Forced periodic optimal scheduling policy for graph reinforcement learning

Miguel S. E. Martins[1[0000-0002-6285-8737]], Susana Vieira[1[0000-0001-7961-1004]], and João M. C. Sousa[1[0000-0002-8030-4746]]

IDMEC, Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal
miguelsemartins@tecnico.ulisboa.pt

Abstract. Efficient scheduling is fundamental when tackling large and complex industrial problems. Heuristic methods provide good enough solutions quickly, but their potential is limited. Exact methods and metaheuristics can reach very good results but might require prohibitively high computational times. Alternatively, a reinforcement learning agent can explore an environment by trial and error and create solutions based on the current environment state. This paper proposes periodically replacing the current agent policy with an optimal policy, which can be obtained by using exact methods on moderate sized problems or from known benchmark solutions. This results in a better final policy, even for problems slightly larger than those of the original training dataset. The scheduling problem is encoded as a graph. The agent is a graph neural network trained with Proximal Policy Optimization. Note that a single trained model can solve problems of any size. Results show that the trained agent can outperform heuristics. Also, it is competitive with other reinforcement learning approaches when solving problems in the order of magnitude of the training instances.

Keywords: Scheduling, reinforcement learning, construction heuristics.

1 Introduction

Every resource-constrained service and industrial process is concerned with efficient resource allocation. Scheduling optimization is an extensively studied subject. Many problem formulations and constraints exist, which can represent most real world problems. Three types of approaches are often used solve them: exact methods, heuristics and metaheuristics.

Exact methods, such as branch-and-bound, are exhaustive search procedures. They rely on efficient mathematical formulations and optimized implementations to navigate the solution space and reach optimal solutions. Heuristic methods create a single solution by following a defined set of rules. These are usually generic enough that they can quickly create a solution for different problems, even if large-sized [9]. However, it is often far from the optimal. Metaheuristics are black box approaches suitable for different problems as long as the problem encoding is compatible. They also follow predetermined steps to search the

solution space. However, they have strategies to avoid getting stuck in local minimum solutions. While metaheuristics almost always outperform heuristic methods, they are probabilistic and thus can require long run times.

All three options solve problems following a fixed set of instructions. Only in the design phase, be it rule setting, problem formulation or encoding, is there opportunity to adjust the approach to better tackle the current problem instances. However, using frameworks that can learn and adapt to different problems erases the need for initial expert knowledge and hyper specific approaches. This can be done with machine learning methods such as reinforcement learning (RL). It learns by interacting with an environment and exploring the available actions, without needing large volumes of high quality data to train. Instead, it relies on a simulation or similar representation of the problem, the environment.

In this paper, a novel modification to a reinforcement learning framework is proposed. During training, periodically, the agent is forced to follow an optimal policy, which is known for benchmark problems with discovered optimal solutions. By switching between current and optimal policy, can the agent improve its performance? Does it also translate to better results on larger, unseen problem instances? The job shop scheduling problem is encoded as a conjunctive graph, which is then given to a graph neural network that predicts what the next best action to schedule is.

In section 2 a brief introduction to RL concepts is presented, followed by a review of existing RL approaches to iterative schedule construction. The problem to solve is introduced in section 3. In section 4 the proposed approach is further detailed. The results are presented and discussed in section 5. Finally, in section 6, the key takeaways and the next research steps are presented.

2 Reinforcement learning

2.1 Brief introduction

Reinforcement learning is a subset of ML where agents learn by interacting with an environment iteratively [14]. In the agent-environment loop, figure 1, the *agent* observes an *environment* and takes a certain *action*. The environment changes and the new configuration, the *state*, is returned to the agent. Alongside comes a *reward* signal, representing how good it was to reach this state. Since the goal of the agent is to maximize the cumulative rewards, the reward signal guides the agent towards a desired behaviour.

The agent mapping between states and actions is called *policy*. By interacting with the environment, the agent can estimate the expected cumulative reward of each state, or state-action pair. The *value function* relates to the selection of a new state, and following current policy until the episode end, while *Q-value* is the expected reward for the immediate next step.

Many algorithms can optimize the value function or the policy. On-policy methods, like SARSA, improve their policy while interacting with the environment following that same policy. Alternatively, off-policy methods learn and

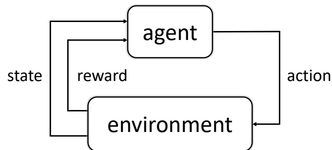

Fig. 1. Agent-environment interaction loop, based on [14].

observe from experiences generated by a different policy, with the goal of being more sample-efficient. For example, the Q-Learning update, Equation 1, gives the Q-value of taking action a at current state s. It updates its current estimate based on observed rewards and the maximum expected future reward, defined as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{1}$$

where α is the *learning rate*, which controls the magnitude of the update, r is the immediate reward, γ is the discount factor, which sets the importance of future rewards, and s' and a' are the future state and action. The double Q-learning variant uses two networks, one to estimate the target and another the current Q-values. This helps with overestimation bias.

Tabular representations are lookup tables can store the Q-values explicitly, but are only applicable when both state and action spaces are small and discrete. Alternatively, *approximate representations* are parametrized functions to approximate these values. While they may introduce approximation errors, they enable RL agents to work with large or continuous spaces. Deep Q-learning (DQN) and Double Deep Q-learning (DDQN) extend the previous Q-learning algorithms by using a deep neural network instead of a Q-table.

All previous methods mentioned are *value-based*. They learn the optimal value function by iteratively updating their estimates with temporal difference or Monte Carlo methods. On the other hand, *policy-based* methods focus on directly learning the policy. REINFORCE collects trajectories of states, actions, and rewards to compute the gradients of the policy parameters.

Actor-critic methods learn the policy via an actor network while simultaneously learning the value function via a critic network. DDPG is an off-policy actor-critic method that extends DQN into continuos action spaces. It learns from a replay buffer of experiences and directly outputs the action. Advantage actor-critic (A2C) is an on-policy method that introduces the advantage of taking an action versus the average action. Outputs the action probability distribution.

Proximal Policy Optimization (PPO) [13] is an on-policy algorithm that is both policy gradient and actor-critic. It has both actor and critic networks, but utilizes a policy gradient approach to optimize the actor network. It presents a clipped surrogate objective to improve sample efficiency and stabilize training.

2.2 Literature approaches

This chapter summarizes RL approaches equivalent to heuristic rules. Proposals that construct solutions by sequentially selecting operations or resources are organized in table 1. Job shop approaches that construct solutions iteratively often deal with flexible, dynamic, of flow-shop variations. In the reviewed papers, different types of neural networks (NN) are used: fully connected, convolutional (CNN), graph (GNN) and recurrent (RNN). It is also of note that [18] uses a type of GNN called graph isomorphic network and [2] a transformer model.

The most common way to represent the problem is as a list of metrics [8, 12, 16], either resource averages [8, 12] or individual operations, for example processing times and status [15, 18]. Instead of lists, 3D matrices can be used [15], which when combined with CNNs [7] can leverage the matrix neighbourhood positions for extra information.

Graph-based [10, 17, 18] and recurrent [2, 4] approaches can use disjunctive and conjunctive graph structures to represent the problem. Besides node and edge features, graph structures allow the edges to implicitly encode problem dynamics like job precedences or machine compatibility. Nodes and edges can have individual features, such as task processing time and setup costs, respectively.

There are two main ways for the agent to build the schedule. The agent can select the next job, operation or resource [2, 4, 15, 18]. All the reviewed approaches use action masking, making infeasible options impossible to select. Alternatively, a set of dispatching rules is the action space. When each operation can only go to one machine, dispatching rules select the order [7, 8, 16, 17]. When machines are also needed, either the approach selects the machine simultaneously [10] or a resource-dedicated agent selects the job [12].

Overall, these methods are often compared with heuristics rules. Some examples of dispatching heuristics include *Shortest Processing Time* (SPT) or *First in, First out* (FIFO) approaches [7, 15], and *Most Work Remaining* (MWKR) [18]. Other used baselines include metaheuristics [15, 16] and RL approaches [7, 17]. Most papers use popular job shop benchmarks[1] and show that that RL job dispatching approaches are competitive with and often outperform heuristics.

3 Job shop scheduling graph representation

The job shop scheduling problem (JSP) is a very popular formulation in operations research [11]. A number of jobs, each consisting of multiple operations, must be allocated to a limited number of machines. Each operation has a specified machine and processing time, and each operation must be done in a specified order. The objective is to minimize the *makespan*, C_{max}, the maximum total completion time of all scheduled operations.

This problem can be represented as disjunctive and conjunctive graph [10, 17, 18], also called a directed graph [11], as can be seen in figure 2. Considering nodes as operations, the directed edges between nodes, the conjunctive edges,

[1] OR-Library: http://people.brunel.ac.uk/ mastjjb/jeb/info.html

Table 1. Summary of papers with iterative solution construction for scheduling.

Ref	Publication	Problem	Model	State	Action	Algorithm
[2]	IEEE Transactions on Industrial Informatics	JSP	encoder-decoder	disjunctive graph embedding	operations	DDPG
[4]	International Journal of Simulation Modelling	flexible JSP	RNN	disjunctive graph	job and machine combo	PPO
[7]	IEEE Access	dynamic JSP	CNN	3D matrix	job dispatching rule	DDPG
[8]	International Journal of Production Research	dynamic JSP	NN	list of environment metrics	job dispatching rules	DDQN
[10]	International Journal of Production Research	JSP	GNN	disjunctive and conjunctive graph	operation	PPO
[12]	Alexandria Engineering Journal	flow-shop	NN	list of environment metrics	job dispatching rule	SARSA, Q-learning
[15]	Computer & Networks	dynamic JSP	NN	3D matrix	operation	PPO
[16]	International Journal of Production Research	flow shop	NN	list of environment metrics	job selection rules	A2C
[17]	Processes	dynamic JSP	GNN	disjunctive graph	job dispatching rules	DDQN
[18]	Applied soft computing	JSP	GNN	list of features per operation	operation	PPO

are precedence constraints. Machine compatibilities can be added as undirected edges between nodes that share the same machine, the disjunctive edges. Both nodes and edges can have additional features.

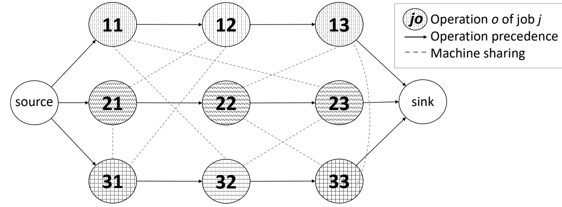

Fig. 2. Job shop problem represented as a disjunctive and conjunctive graph.

4 Proposed approach

To minimize the makespan of job shop scheduling problems, an agent is trained to create new schedules. It observes the current graph-encoded schedule and selects the next operation to add. The reward signal is based on the iteration increment of the maximum completion time.

A single agent must solve problems of different sizes. A GNN is used as agent model, which fits the previous requirement, and PPO is used to train the model. A naive lower bound is computed, summing all processing times and dividing it by the total number of machines. This is used to normalize all features dealing with processing times.

This approach resembles [10, 17]. However, state representation, model architecture and algorithm used are different. More importantly, in this paper the periodic optimal policy switch is proposed, as described in subsection 4.2.

4.1 Agent-environment interactions

State representation: machine and operation allocations are encoded as a conjunctive graph, as explained in section 3, but without source and sink nodes or disjunctive machine edges. Thus, the representation contains two matrices: node features and edge values. The node, or operation, features used are:

- Boolean representation if operation can be selected;
- Boolean representation if operation is already done;
- Normalized operation processing time;
- Normalized job processing time;
- Normalized subsequent operations' processing time;
- Subsequent operations' processing time, as percentage of full job duration;
- Total number of operations of job;
- Number of subsequent operations;
- ID of machine attributed to operation;
- Earliest possible start time at attributed machine;

– Operation free time, or -1 if not yet scheduled.

The edge features matrix is a $[n \times m, 10]$ matrix, where n is the total number of jobs and m the total number of operations per job. The edge values matrix is a $[2, (2m-1)n]$ matrix. Precedence constraints between operations are pairs of rows, identifying the origin and destination node of each edge. Self-loops are needed to send its own information to the GNN embedding layers. These edges start and end at each node, bringing the total number of edges to $(2m-1)n$.

Action space: the agent outputs the single operation to schedule next. Internally, the agent is using action masking to assure only feasible operations are selected. When compared to the same approach without action masking, the agent takes much longer to converge to a worse solution.

Reward signal: the objective is to minimize the makespan. However, it is only possible to get its value at the end of the episode, when all the operations are scheduled. Thus, the reward signal used is the increment to the maximum completion time in this iteration, divided by the lower bound. No change to the maximum completion time means zero reward. The agent penalty is proportional to the increase in makespan, normalized by the lower bound. This usually keeps the cumulative penalty per episode between $[-2.5, -1]$.

4.2 Forced periodic optimal policy

This paper proposes an off-policy PPO modification where the policy alternates between the current policy and an optimal policy periodically. For the training instances used the optimal solution is known. Thus, it is possible to create a sequence of agent actions that would lead to this optimal solution by ordering the start times of each operation. Following this sequence of actions for the corresponding problem creates an optimal policy.

This modification is inspired by the *elitism* procedure in a genetic algorithm. In this metaheuristic, new solutions are generated every iteration. However, to keep high quality solutions in the genetic pool, some of the best solutions from the previous iteration have guaranteed spots in the following generation.

Since optimal solutions will not always be available for training, only the simplest four instances from the *la* job shop benchmarks are used to train. Simpler problems are usually easier to solve with exact methods. This will clarify if it is possible to train the agent on simpler problems and still be performant on larger instances. The periodicity of the policy swap is evaluated in the next section.

4.3 Agent model

The agent is represented by a GNN model [10]. It uses *SAGEConv* [3] node embedding layers to interpret connections between nodes and edges of the graph representation. For each node, sequential embedding layers aggregate information from sequentially connected neighbours.

GNNs take the disjunctive graph information, as previously described, and output a value for each node in the input graph. The action masking will invalidate any outputs that translate into infeasible actions by replacing the model

output of infeasible actions with $-\inf$. Then, a *softmax* layer turns these real numbers into probabilities, from where the next action is sampled.

The approach presented uses PPO [5, 1, 13], a popular algorithm for JSP as seen in table 1. Both actor and critic networks have the same structure. Typically the actor and the critic share some of the layers, but for the current implementation it was not beneficial. Table 4.3 presents the parameters.

The agent was trained and tested on JSP benchmarks. Specifically, *la01* to *la40* [6]. The number of machines is equal to the number of jobs, an each job has the same number of operation. Each operation is attributed a specific machine ID and a processing time. Inside a single job, all operations must be processed in a pre-determined order and all use different machines.

The agent trained with 4 parallel environments, each running a different instance: *la01*, *la02*, *la03* and *la04*. Learning rate annealing was used to drive its value to zero at the end of the episode, which takes 500000 steps. The Adam optimizer is used with 4 mini-batches and 4 epochs to update the policy. Each policy rollout runs for 256 steps. For the models, each have 5 *SAGEConv* layers with 516 neurons, using ReLU as activation function and one final linear layer.

Table 3. PPO parameters used to train the GNN model.

Parameters	**Values**
Learning rate	2×10^{-5}
Discount factor (γ)	0.99
GAE parameter	0.95
Clipping parameter	0.2
Value function coefficient	1.0
Entropy bonus coefficient	0.01
Maximum gradient norm	0.5

The remaining *la* instances not used for training are used to evaluate the models versus the selected baselines. The baselines used are the SPT, FIFO and MWKR heuristic rules. While [10, 18] use slightly different approaches, they solve the same problems and so are also used as baselines.

The implementation relies on *torch* and *torch_geometric* Python libraries. All training and testing was done with the following specifications: AMD Ryzen 7 3700X 8-Core Processo CPU, 4.00 GHz speed; 32 GB RAM; Windows 10 Pro operating system; NVIDIA GeForce RTX 2700 SUPER graphics card.

5 Results and Discussion

This section presents the results for the proposed approach. All *la* benchmarks were tested, but tables 4 and 5 only show the results for the first two problem instances of every complexity. The overall results reflect the conclusions drawn from these truncated tables, as can be seen in figure 3, where the average results

for all benchmark instances are presented. For example, the values for 10×5 are the average of all 5 instances with 10 jobs and 5 operations per job.

Table 4 presents the results regarding the periodic optimal policy frequency parameter. *Every rollout* means that after the policy update, in the next agent-environment exploration phase, the first episode of each environment will follow the optimal policy. The other optimal policy frequencies tested are every ten rollouts, every hundred and never.

Table 4. Selected *la* results for different optimal policy frequency. The results presented are the best results obtained in ten different runs. †Problem instances used during training.

Benchmark	n × m	Never	100 rollouts	10 rollouts	Every rollout
la01†	10 × 5	900	943	768	**732**
la02†	10 × 5	**782**	863	795	789
la06	15 × 5	1131	1050	985	**944**
la07	15 × 5	1149	1239	1045	**999**
la11	20 × 5	1363	1405	**1263**	1264
la12	20 × 5	1182	1125	**1061**	1108
la16	10 × 10	**1135**	1661	1304	1161
la17	10 × 10	1340	1534	1228	**995**
la21	15 × 10	1645	1937	1640	**1429**
la22	15 × 10	1428	1686	1412	**1246**
la26	20 × 10	1796	2180	1752	**1667**
la27	20 × 10	**1725**	2256	1842	1781
la31	30 × 10	2448	3475	2486	**2228**
la32	30 × 10	**2386**	3228	2526	2689
la36	15 × 15	**1901**	3230	1985	2100
la37	15 × 15	**2066**	2761	2252	2192

Results show that for the current implementation, forcing the periodic optimal policy every rollout improvements it. It achieves the best result more frequently and has the lowest average gap to optimal solution (28%). While never using the optimal policy still reaches the best solution many times, it is the third average gap (37%), after every ten policy rollouts (35%).

Note that using 100 rollouts frequency results in a worse agent (67%) than with no forced optimal policy. This shows that repetition is important for the agent to learn with an optimal policy. It is also interesting that for higher complexities, not forcing optimal policy leads to better performance. That might be because forcing an optimal policy on small instances will make agents better, but only up to a certain degree of complexity close to the training examples. After that, it has a detrimental effect.

Table 5 compares the best trained model, using elitism every policy rollout, with the benchmarks. Figure 3 shows the gap to the optimal solution for all *la* benchmarks, grouped by similar problem instances.

Table 5. Selected *la* benchmark results for baselines: heuristics, RL approaches [10, 18] and proposed approach. For the proposed approach, it is shown the average of ten runs, as well as the best result of those ten. †Problem instances used during training.

Instances	n × m	Optimal	SPT	FIFO	MWKR	[10]	[18]	Ours	
								Average	Best
la01†	10 × 5	666	751	772	773	805	**670**	818.9	732
la02†	10 × 5	655	821	830	803	**687**	756	827.1	789
la06	15 × 5	926	1200	**926**	**926**	**926**	932	1032.8	944
la07	15 × 5	890	1034	1088	1030	**931**	1012	1059.8	999
la11	20 × 5	1222	1473	1272	1238	1276	**1228**	1348.8	1264
la12	20 × 5	1039	1203	**1039**	**1039**	**1039**	1050	1192.2	1108
la16	10 × 10	945	1156	1180	1060	1134	**1051**	1296.1	1161
la17	10 × 10	784	924	943	859	953	**857**	1055.4	995
la21	15 × 10	1046	1324	1265	1234	1309	**1179**	1527.9	1429
la22	15 × 10	927	1180	1312	1187	1158	**1057**	1312.1	1246
la26	20 × 10	1218	1498	**1372**	1453	1553	1383	1817.8	1744
la27	20 × 10	1235	1784	1644	1473	1642	**1515**	1919.4	1799
la31	30 × 10	1784	1951	1918	1832	**1817**	1894	2366.4	2227
la32	30 × 10	1850	2165	2110	1995	1977	**1902**	2869.5	2689
la36	15 × 15	1268	1799	1516	1508	1489	**1426**	2220.9	2100
la37	15 × 15	1397	1655	1873	**1594**	1623	1649	2261.4	2192

The trained agent cannot overcome the reinforcement learning baselines. It is competitive for the simpler problems, even for the average values, and it can compete with heuristic rules up to medium complexity. However, the performance deteriorates after that. Thus, forcing optimal policy on an agent that trains only with small problems leads to some improvements, but it cannot reliably solve harder problems. In figure 3 can be seen that all approaches worsen with rising problem complexity. However, the scalability of the proposed approach is the worst one. This reinforces the conclusion from before. Note that while the current implementation cannot surpass the state-of-the-art RL methods, periodically forcing the optimal policy was ultimately a good strategy. From the values of table 4, never switching to the optimal policy is approximately 7% worse than forcing it every rollout. This value is closer to 9% when evaluating all *la* instances not used for training.

6 Conclusions

In this paper, an off-policy modification to the PPO is proposed. Results show that for the current implementation, periodically switching between an optimal policy and the current policy, at every policy rollout, leads to a 9% improvement. Thus, adopting this alternating strategy can potentially improve other RL approaches as well. Comparatively to state-of-the-art approaches, the current implementation is competitive for smaller scale problems.

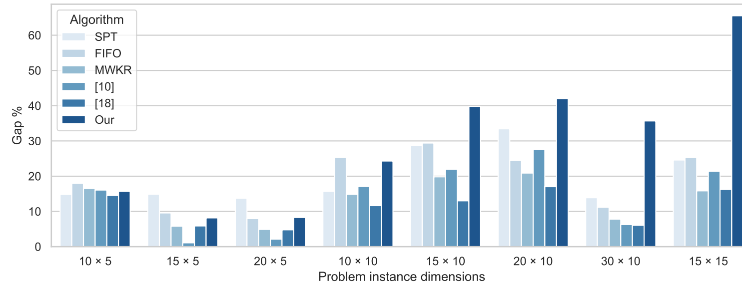

Fig. 3. Gap (%) to optimal solution, shown as average of all instances with the same dimensions, per algorithm. Note that four of the five 10×5 problem instances were used during training.

As problem complexity increases, the performance drops rapidly comparatively to the baselines. Having a more balanced dataset, with some problem instances of larger sizes, could mitigate this issue. Also, forcing the optimal policy when training with more complex problems, which usually have more steps, might be even more valuable than for shorter problems. This is also applicable to flexible job shop problems, since having the agent select both the job and the appropriate machine greatly increases the state and action spaces.

Finally, the proposed approach assumes that the optimal solution is known. If that was not the case, how different would the performance improvement be? What if the agent is forced to follow a heuristic solution instead? Would it be better for the agent to start with a heuristic solution, and only later be see the higher quality solution? It also seems promising to investigate dynamic changes to the optimal policy period, either changing the frequency over time or using the optimal policy to keep the agent from converging prematurely.

Acknowledgments. This work was supported by FCT, Fundação para a Ciência e a Tecnologia, I.P., under the PhD scholarship 10.54499/2020.08776.BD (Miguel S.E. Martins). The authors acknowledge Fundação para a Ciência e a Tecnologia (FCT) for its financial support via the project LAETA Base Funding (DOI: 10.54499/UIDB/50022/2020) and via the project LAETA Programatic Funding (DOI: 10.54499/UIDP/50022/2020).

References

1. Andrychowicz, M., Raichuk, A., Stańczyk, P., Orsini, M., Girgin, S., Marinier, R., Hussenot, L., Geist, M., Pietquin, O., Michalski, M., Gelly, S., Bachem, O.: What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study (2020), http://arxiv.org/abs/2006.05990
2. Chen, R., Li, W., Yang, H.: A Deep Reinforcement Learning Framework Based on an Attention Mechanism and Disjunctive Graph Embedding for the Job-Shop Scheduling Problem. IEEE Transactions on Industrial Informatics **19**(2), 1322–1331 (2023). https://doi.org/10.1109/TII.2022.3167380

3. Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. Advances in Neural Information Processing Systems **2017-Decem**(Nips), 1025–1035 (2017)
4. Han, B.A., Yang, J.J.: A deep reinforcement learning based solution for flexible job shop scheduling problem. International Journal of Simulation Modelling **20**(2), 375–386 (2021). https://doi.org/10.2507/IJSIMM20-2-CO7
5. Huang, S., Dossa, R.F.J., Ye, C., Braga, J., Chakraborty, D., Mehta, K., Araújo, J.G.: CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms. Journal of Machine Learning Research **23**(274), 1—-18 (2022). https://doi.org/https://doi.org/10.48550/arXiv.2111.08819
6. Lawrence, S.: An experimental investigation of heuristic scheduling techniques. Supplement to resource constrained project scheduling (1984)
7. Liu, C.L., Chang, C.C., Tseng, C.J.: Actor-critic deep reinforcement learning for solving job shop scheduling problems. IEEE Access **8**, 71752–71762 (2020). https://doi.org/10.1109/ACCESS.2020.2987820
8. Liu, R., Piplani, R., Toro, C.: Deep reinforcement learning for dynamic scheduling of a flexible job shop. International Journal of Production Research **60**(13), 4049–4069 (2022). https://doi.org/10.1080/00207543.2022.2058432
9. Martins, M.S.E., Viegas, J.L., Coito, T., Firme, B., Costigliola, A., Figueiredo, J., Vieira, S.M., Sousa, J.M.C.: Minimizing total completion time in large-sized pharmaceutical quality control scheduling. Journal of Heuristics (2023). https://doi.org/10.1007/s10732-023-09509-8
10. Park, J., Chun, J., Kim, S.H., Kim, Y., Park, J.: Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. International Journal of Production Research **59**(11), 3360–3377 (2021). https://doi.org/10.1080/00207543.2020.1870013
11. Pinedo, M.L.: Planning and Scheduling in Manufacturing and Services. Springer, 2nd edn. (2009). https://doi.org/10.1007/978-1-4614-2361-4
12. Ren, J., Ye, C., Yang, F.: Solving flow-shop scheduling problem with a reinforcement learning algorithm that generalizes the value function with neural network. Alexandria Engineering Journal **60**(3), 2787–2800 (2021). https://doi.org/10.1016/j.aej.2021.01.030
13. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal Policy Optimization Algorithms pp. 1–12 (2017), http://arxiv.org/abs/1707.06347
14. Sutton, R.S., Barto, A.G.: Reinforcement Learning. MIT Press (2020)
15. Wang, L., Hu, X., Wang, Y., Xu, S., Ma, S., Yang, K., Liu, Z., Wang, W.: Dynamic job-shop scheduling in smart manufacturing using deep reinforcement learning. Computer Networks **190**(November 2020), 107969 (2021). https://doi.org/10.1016/j.comnet.2021.107969
16. Yang, S., Xu, Z.: Intelligent scheduling and reconfiguration via deep reinforcement learning in smart manufacturing. International Journal of Production Research **60**(16), 4936–4953 (2022). https://doi.org/10.1080/00207543.2021.1943037
17. Yang, Z., Bi, L., Jiao, X.: Combining Reinforcement Learning Algorithms with Graph Neural Networks to Solve Dynamic Job Shop Scheduling Problems. Processes **11**(5) (2023). https://doi.org/10.3390/pr11051571
18. Yuan, E., Cheng, S., Wang, L., Song, S., Wu, F.: Solving job shop scheduling problems via deep reinforcement learning. Applied Soft Computing **143**, 110436 (2023). https://doi.org/10.1016/j.asoc.2023.110436