Preprints of the 21st IFAC World Congress (Virtual)
Berlin, Germany, July 12-17, 2020

Reinforcement Learning for Dual-Resource Constrained Scheduling ⋆

Miguel S. E. Martins * **Joaquim L. Viegas** * **Tiago Coito** *
Bernardo Marreiros Firme * **João M. C. Sousa** *
João Figueiredo ** **Susana M. Vieira** *

* *IDMEC, Instituto Superior Técnico, Universidade de Lisboa, (e-mail: miguelsemartins@tecnico.ulisboa.pt; joaquim.viegas@tecnico.ulisboa.pt; tiagoascoito@tecnico.ulisboa.pt; bernardo.firme@tecnico.ulisboa.pt; jmsousa@tecnico.ulisboa.pt; susana.vieira@tecnico.ulisboa.pt)*
** *Department of Physics, Universidade de Évora, Évora, Portugal (e-mail: jfig@uevora.pt)*

Abstract: This paper proposes using reinforcement learning to solve scheduling problems where two types of resources of limited availability must be allocated. The goal is to minimize the makespan of a dual-resource constrained flexible job shop scheduling problem. Efficient practical implementation is very valuable to industry, yet it is often only solved combining heuristics and expert knowledge. A framework for training a reinforcement learning agent to schedule diverse dual-resource constrained job shops is presented. Comparison with other state-of-the-art approaches is done on both simpler and more complex instances that the ones used for training. Results show the agent produces competitive solutions for small instances that can outperform the implemented heuristic if given enough time. Other extensions are needed before real-world deployment, such as deadlines and constraining resources to work shifts.

Keywords: Production planning and control, Job and activity scheduling, Intelligent manufacturing systems

1. INTRODUCTION

Scheduling is an integral part of manufacturing and service industries. Efficiently assigning work to the available resources can have a big impact on overall efficiency. One of the most common approaches to modelling manufacturing units is the *job shop* (JS) formulation, as can be seen in Pinedo (2009).

The JS is concerned with efficiently assigning the workload, the *jobs*, to the available resources, commonly *machines*. For over forty years the JS has been a widely researched topic. It has many extensions and a wide range of solution approaches dedicated to each, as detailed in Morshed et al. (2017). If multiple equivalent resources exist it is called a *flexible* JP problem.

In many practical applications it is of interest to also allocate other auxiliary resources when these are limited and shared by all jobs. The *dual-resource constrained* (DRC) problem is used to model capacity constraints by two types of shared resources. Typically, these two constraints represent labour and machines. Labour commonly refers to worker availability for job processing or machine handling, as can be seen in Cunha et al. (2019). However, the constraining auxiliary resource could also a transporter between machines, as seen in Nouri et al. (2016). Moreover, a second material resource might also be needed simultaneously with machines. As detailed in Huiyuan et al. (2009), in a mass injection molding case study, neither machines nor moulds can be scheduled independently of each other. The same can be said on photolithography processes for semiconductor production, where machines and reticles are need simultaneous as presented in Ham (2018). A schematic of the DRC methodology considered for this paper can be seen on figure 1.

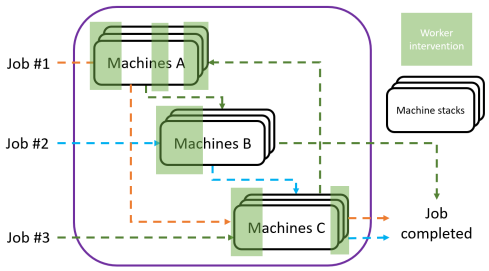

Fig. 1. A DRC flexible job shop combining machines (white rectangles) and workers (green rectangles).

Job shop literature considering the DRC is limited, as can be seen in Dhiflaoui et al. (2018). Only a handful of publications model workers who only need to be present for smaller periods of machine run time, such as: Shen et al. (2018); Cunha et al. (2019). On these most recent publications, worker intervention is modelled with the desired level of freedom of multiple varying interventions per operation. The formulation considered for the work

⋆ This work was supported by FCT, through IDMEC, under LAETA, project UIDP/50022/2020

presented this paper is based on Cunha et al. (2019), as explained in section 3.

Being a fairly complex formulation, the DRC struggles with the exponential growth of the combinatorial search space, similarly to the JS. It is not uncommon for strategies to efficiently solve smaller instances yet quickly drop performance as problem instances scale, leading to unfeasible computational efforts. Taking as an example the work presented in Cunha et al. (2019), a Mixed Integer Linear Programming (MILP) algorithm is implemented to find the optimal solutions. Here solutions for small problem instances (three machines, three jobs and one worker) are easily found. However, for medium and bigger problems no exact solution can be found in sensible computational times and the algorithm must be forced to stop with no guarantees of optimality.

While common approaches tend to explicitly program the behaviour of the algorithms, there is value in considering strategies able to infer rules and mappings autonomously. A common example of such algorithms are machine learning methods. These are dedicated to inferring patterns from data to perform specific tasks. Some could even be made suitable for online learning, this is, learning directly from streaming data. This way knowledge of the system could be updated in real-time. Some work has been done in Araz (2005) regarding the creation of metamodels using artificial neural networks. The focus of these networks is to effectively represent the real system as a model of lower-complexity for simulation use, increasing computational efficiency.

Dynamic approaches are not uncommon on JSP solutions without the DRC. A noteworthy approach is presented in Zhang and Dietterich (2000). Here, a reinforcement learning (RL) agent is trained to choose which repair actions will be iteratively applied to an unfeasible schedule. The agent is then able to solve other problem instances of different size and complexity and is able to generalize from previous experience.

The key idea behind all the referred strategies is to offload some part of the model representation to a learning structure. By preparing an approach to learn online, not only can a solution overcome modelling bias but it can also be used for adaptative scheduling. Thus, in this paper is presented a routine able to train an agent to solve diverse DRC scheduling problems without knowledge of a system's dynamics.

The concepts presented by Zhang and Dietterich (2000) are the foundation for the solution approach presented in this paper, extended to encompass a dual-resource contained flexible job shop problem. To the best of the authors knowledge, this is the first time an RL agent is applied to a DRC formulation. On the following section relevant concepts from reinforcement learning are detailed.

2. REINFORCEMENT LEARNING CONCEPTS

Reinforcement learning is focused in learning how to act, or how to map situations into actions, in order to maximize a numerical reward signal as described in Sutton and Barto (2018). Without direct instructions on what actions to perform, an agent must discover by trial-and-error what will lead to a biggest cumulative reward over a full run. A key challenge is how to connect specific actions to good rewards, since these rewards can be a consequence of the immediately preceding action or of a very early action. These two concepts, *trial-and-error search* and *delayed rewards* are extremely important in RL design.

The agent iteratively interacts with the environment as schematized on figure 2. At time t, the agent performs action A_t, knowing the current state S_t and previously received reward R_t. The action will cause the environment to move from state S_t to S_{t+1}. This new state is also paired with a corresponding reward R_{t+1}. After advancing a timestep in the system, this is $t = t + 1$, values S_t and R_t become S_{t+1} and R_{t+1}, respectively. The parameters are then feed into the agent, further continuing the cycle until some stopping criteria is reached. For episodic cases this can be a terminal state and for continues cases this can be maximum allotted time.

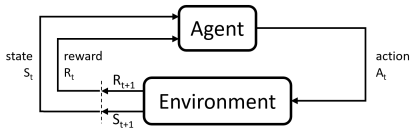

Fig. 2. Agent-environment interaction of a RL agent based on Sutton and Barto (2018)

Solving a RL task means finding the agent behaviour, the *policy* π, which will maximize the cumulative sum of rewards signals collected during a single run. These rewards are designed for each problem, for example by only positively rewarding an agent when reaching the desired goal. Note that *maximizing rewards* can be switched for *minimizing costs* without loss of generality for any of the presented concepts.

The algorithm can either run episodically, limited by time or number of iterations, or continuously. To avoid that the sum of cumulative rewards goes to infinity in continuous tasks, the *discount factor* $\gamma \in [0, 1]$ is used. The discount factor represents the trade-off between immediate and delayed rewards.

The *value function*, $V_\pi(s)$ expresses how desirable a state is regarding immediate and future rewards. Formally it represents the expected return starting from state s and following policy π for the remaining steps, as represented in equation 1.

$$V_\pi(s) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t R_t | s_0 = s\Big] \tag{1}$$

This means that the discount factor will determine if a bigger emphasis is to be given to rewards R close by or to future rewards. Looking at the extreme cases, with $\gamma = 0$ the value function is only affected by immediate rewards while $\gamma = 1$ treats all rewards for the rest of the episode as equally important. When receiving the feedback from the taken action, the agent updates its internal knowledge of the system. Both state representation and type of update depend on the specific RL methodology used.

Monte Carlo methods can use this sequence of states, actions, rewards to learn directly from interactions with the environment without previous knowledge of the system.

They do so by following a fixed policy for a full episode. At the end of it, the discounted reward for each visited state is calculated and the estimated value function for each visited state can be updated. To update these estimates before the episode ends, temporal-difference learning can be used instead.

Temporal-difference learning (TD) is also recurrently employed to learn directly from raw experience without a model of the environment's dynamics. It iteratively updates its estimates without waiting for a final outcome of the interaction with the environment, making it suitable for both episodic and continuous cases. TD uses *bootstrapping*, which consists on updating estimates based on current sample estimates. A simple TD method, one-step TD, makes an update as described in equation 2 by updating the value of a state after moving to another and observing the reward. Here α is the step size, which weights the impact of an update in the current value.

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big] \tag{2}$$

In many problems it can be impractical to individually store estimates for all the possible states due to prohibitively large state spaces. For example, using camera images as states, it is impractical to represent every frame as a unique state both due to memory constraints and due to the required time to iterate through enough data to estimate these values accurately. For these cases, *function approximations* can be used to efficiently express states as lower-complexity representations, from where values can be generalized. The value function can then be expressed as $V(s) \approx \hat{V}(s, \mathbf{w})$, where $\mathbf{w}$ represents a generic parameter vector. One example of a commonly used function estimator is a multi-layer artificial neural network, where $\mathbf{w}$ represents the vector of connection weights between all layers.

Eligibility traces can be used to extend TD algorithms. Depending on the eligibility parameter, λ, a TD method can change its behaviour from a Monte Carlo method ($\lambda = 1$) to a one-step TD ($\lambda = 0$). The eligibility trace, a short-term memory vector denoted as $\mathbf{z}$, is used to influence the impact of weight gradients on the weight update, according to equation 3. This way weight updates will mostly impact specific weights recently big gradient values on previous iterations. This long term impact depends on the decay rate defined by λ.

$$\begin{aligned} \mathbf{z} &= \gamma\lambda\mathbf{z} + \nabla\hat{V}(S_t, \mathbf{w}) \\ \delta &= R_{t+1} + \gamma\hat{V}(S_{t+1}, \mathbf{w}) - \hat{V}(S_t, \mathbf{w}) \\ \mathbf{w} &= \mathbf{w} + \alpha\delta\mathbf{z} \end{aligned} \tag{3}$$

This and the all other RL concepts are extensively detailed in Sutton and Barto (2018). After having briefly introduced the concepts relevant to the scope of this paper, the background section now gives place to the problem statement.

3. PROBLEM STATEMENT

In this section the tackled *dual-resource constrained flexible job shop scheduling problem* is detailed. The complete mathematical formulation used is based on Cunha et al. (2019). Due to this, only a summary of the assumptions to what is presented in the cited paper are presented next.

The Job Shop problem is ultimately concerned with assigning jobs to resources with the goal of minimizing the total schedule length, where resources have limited availability. Two types of resources are considered, machines and workers, which introduces the dual-resource constraint to the JS problem. A solution to this problem needs to detail what is each job's start time and allocated resources. The total time to perform all jobs is known as the makespan, C_{max}, and will be used as minimization goal. Feasible solutions must respect the following constraints:

- Precedence constraints: operations of a job must obey their predefined sequence. This means that an operation can only start after its immediate preceding operation ends, even if they are being processed on different resources. Similarly, an operation must always end before the following one starts.
- Allocation constraints: Every resource has a capacity of one, meaning that, for each resource, at any time, only one task can be processed. This is valid for both machines and workers.
- Sequence constraints: any two operations on the same resource must not overlap, independently of resource type.
- Dual-resource constraint: workers are to be allocated alongside machines, starting at specific time steps that can be different from the machine start time. The length of a worker intervention can be the same as the paired machine processing length, or it can be smaller.

To clarify the last constraint, at multiple timesteps of machine processing a worker can be requested for a certain percentage of time. This way there are multiple periods of worker and machine allocation and other of solo machine allocation. Most DRC formulations do not expect multiple worker interventions, typically either tying a worker to the full machine processing time or allowing partial allocation of the worker at the start of the operation. Note that workers do not have these restrictions and are free to switch between any operations, as long as their unitary capacity is always respected.

4. PROPOSED APPROACH

The goal of the proposed approach is to teach an agent how to schedule DRC JS problems, independently of complexity or dimension of training instances. To do so, the agent will iteratively apply actions on scheduled tasks, altering its starting time or the allocated resources.

Making the connection with the generic RL cycle presented in figure 2, the environment is the current schedule solution where actions will be applied. The actions change an operation's start time or allocated resources.

4.1 Actions available to the agent

Two types actions are available to the agent, *MOVE* and *REASSIGN*. The first action translates a task on the time axis. The second action, *REASSIGN*, changes one of the allocated resource types. The actions are only allowed if the task acted upon lands on a feasible configuration.

To the MOVE operator must be indicated if the task is to be moved forwards or backwards. For either case, the allocated resources remain unchanged. A task is always moved together with its worker/machine pair. When performing a MOVE, all *depended tasks* on the specified direction are unscheduled. Moving forwards, these tasks are the remainder sequential tasks of a job. If moving backwards they are the previously scheduled tasks of the same job.

Possible insertion points along the specified direction include other task end times and other task start times minus the machine processing time for the current pair to move. Starting closest to the original position, the pair is iteratively inserted until a feasible place is found. The dependent tasks are then rescheduled using a *critical path insertion*, even if the dependent tasks land on unfeasible positions. A critical path insertion simply allocates the dependent tasks sequentially in the shortest possible span without overlapping.

The *REASSIGN* operator can be applied to either resource type, worker or machine. Since machine tasks cannot be separated, when reassigning the full operation set is reassigned while worker tasks do not have this restriction. These two actions together with the two move types, forwards and backwards, comprise the allowable actions.

When the current schedule configuration is unfeasible, a task is randomly selected from one of the earliest schedule conflicts. Unlike the original paper, when the schedule is feasible a weighted random selection is used, favouring biggest machine slack times.

4.2 Reward distribution

Instead of only using total length of the schedule, another metric is used to define the rewards given to an agent. The proposed performance measure takes into account both the schedule length and the resource allocation independently of the problem instance size, as introduced by Zhang and Dietterich (2000). In practice, this metric translates the schedule length in relative terms with the initial solution while also being slightly impacted by resource occupation. This assures that different values are returned for schedules with the same length while favouring allocations with less slack times.

To measure the length and over-allocation of a solution s, the *Resource Utilization Index* (RUI) is defined according to equation 4.

$$RUI(s) = \sum_{k=1}^{K} \sum_{t=1}^{T} max\{1, \text{util}(k,t)\} w_t^l \tag{4}$$

where $\text{util}(k,t)$ is the current utilization of resources of type k over time window t. K is the total number of resources, T the total number of windows and w_t^l is the length of window t in time units. A window is defined as a time partition of the schedule where there are no changes in the participating tasks.

The chosen metric to evaluate the value of a schedule during a run is the *Resource Dilatation Factor* (RDF). It takes into account the current RUI and the initial solution's RUI, as defined in equation 5. This is done to normalize this metric and obtain similar RDF values for all problem instances.

$$RDF = \frac{RUI(s)}{RUI(s_0)} \tag{5}$$

The values of RDF are usually between 0.8 and 2, where a lower value corresponds to a higher quality solution. The reward R, given to the agent is derived from this measure, according to the following rules:

- If the current solution is feasible and has the best RDF value found so far, $R = \frac{1}{RDF}$.
- If the current solution is feasible but not an improvement, the percentage difference from the best solution so far is returned minus a small negative number, $R = -\frac{RDF_{current} - RDF_{best}}{RDF_{best}} - 0.001$
- If current solution is unfeasible, $R = -0.01$.

The small negative reward is always given to lightly punish the agent for each action that leads to an infeasible solutions. A smaller negative reward is also added when no better solution is found. By penalizing any solution that is not an improvement from the best solution the agent is always encouraged to search for new configurations, even when there is no difference between current and best solution.

4.3 State representation

Instead of using a full schedule representation as state, a more compact notation is desired. As state S will be used a set of metrics computed from a full schedule configurations. The resulting vector of real numbers is much easier to both store and manipulate than the full schedule and can be used together with the function approximation detailed in the next section. The metrics computed independently for both workers and machines are:

- Mean and standard deviation of free capacity, which measures the percentage of unallocated time units
- Slack times, defined as the time between the start of a task and the end of a preceding task. For the average and the minimum slacks, the median and the standard deviation are calculated, for a total of four metrics per resource type.
- Number and percentage of windows in violation
- Percentage of windows which could be solved by reassignment. This is, percentage of overallocated windows with equivalent resources available for the same time window.
- Time units in violation, relative to total schedule
- Average number of violation windows per resource
- Location of first window in violation, respective to the total number of windows, and the percentage of total violations in next ten windows
- Resource utilization, defined as the average percentage of occupied times in the initial schedule.

Each state also contains the current RDF value and a variable dependent on feasibility. If the solution is not feasible it has a value of 0, else it contains the RDF value. This gives a total of 30 features.

4.4 Function approximation

The used function approximation is an artificial neural network with one fully connected layer of 128 hidden units and a ReLU activation function. As input it uses the state vector of 30 elements and it has 1 output, the value function estimate for the input state.

For this implementation, three parameter ranges create different network configurations for training: step size $\alpha \in [0.001, 0.0001]$, the change of exploration rate, $\Delta\beta \in [0.95, 0.99]$, and the eligibility parameter, $\lambda \in [0.2, 0.7]$. This creates a total of 8 networks to train. A discount factor of $\gamma = 0.99$ is always used. The weight updates are done using equations 3 and stochastic gradient descent.

4.5 Agent training

In the *training algorithm* multiple networks are to be trained on the full set of training instances. Each network has a set of parameters as detailed in subsection 4.4.

The *training algorithm* will go over all train instances iteratively, while there are still networks to train. At each cycle, the algorithm is divided in three main parts:

- Schedule with exploration: for each network, always starting from an initial solution, apply actions to the schedule iteratively.
- Update network: after getting a sequence, perform a weight update on the respective network.
- Stopping criteria check: every so often run a separate validation instance set on the networks to decide when to stop training each network.

The first procedure, *schedule with exploration*, is responsible for creating the schedules. As input, it requires a problem instance and a network. It returns the full scheduling history and respective reward at each step.

To select an action from the available pool, a mechanism related to ϵ-greedy is used. With a random chance $1 - \beta$, the current network in use will predict which of the four actions might lead to the best cumulative sum of rewards. The alternative, with random chance β, is that all available actions at that instance are tested and the most attractive selected. Initially β is set to 1, but with each successive iteration is updated according to $\beta \leftarrow \beta * \Delta\beta$.

On the second procedure, *update networks*, the created scheduling history is used to update the network using as target the TD update formula from equation 2. This update is based on equations 2 and 3, applied to each two consecutive sequence elements, starting with the last two elements and progressing backwards. Every 5 iterations the networks are updated with the best full schedule history saved instead of the most recent. This is done so that the network does not forget good solutions.

At last, the procedure *stopping criteria check* will solve every problem of the validation set on the full validations set. It runs every 5 iterations. If the current average RDF is not an improvement over the previous 5 runs' average times then the network is no longer updated. When no more networks are in queue to be trained, the training algorithm stops.

4.6 Agent for scheduling

After having the networks trained, the *scheduling algorithm* uses the best trained agent on never seen problem instances from the test set. The selected network has the highest performance on the validation set. To decide what action to execute, it uses a similar version to *schedule with exploration*, but with no exploration ($\beta = 0$). This means that it will always choose an operator based on the network's prediction.

In this algorithm is also implemented a simple memory to avoid infinite loops. During scheduling two lists are maintained: one for the states visited and another for keeping track of the number of visits. If any state is visited a second time, the second best prediction is chosen as action to take. The third time a state is visited, the algorithm returns to the previous state and the second best prediction of said state is selected. This continues recursively until an acceptable state is reached.

4.7 Problem instances

To train the agent, multiple problem instances were generated. The number of jobs can be 3, 5, 8 or 10. For instances with 3 jobs, there are 2 machines and 1 worker available. For all other there are 3 machines and 2 workers. For all instances, there is full flexibility of resources. This means that all machines and all workers are equivalent. Six instances of 5 jobs were used on the training algorithm. Four were used for training and two for validation. For the scheduling algorithm four instances were used, each with a different job number.

Individually, each instance presents variations regarding number of operations per job, processing times per operation and number of unique sample types. For all operations it is assumed three worker interventions take place, requiring the following worker's presence relative to machine time: *setup*, from 0 to 5% of machine time; *intermediate*, from 30% to 40% of machine time; *teardown*, from 90% to full machine time.

For every problem instance, an initial solution is generated in one of the following ways. *Critical insertion* is done by randomly distributing the first task of each job over all resources at zero time, disregarding resource capacity. Then, all following tasks are inserted according to their critical path on the same resource. *Feasible insertion* is done by adding all jobs sequentially on only one resource, which returns a feasible but lengthy solution. This was done to force the agent to train using move operators both forwards and backwards. For the training phase half of the initial instances come from critical insertion and half from feasible insertion. For the scheduling part all instances are initiated with critical insertion.

5. RESULTS

To test the trained agents, the best trained network is to be compared with two other strategies from the literature: the MILP integer implementation from Cunha et al. (2019) and an heuristic algorithm as presented in Viegas et al. (2019). The latter is based on a series of rules based on expert knowledge.

On table 1 are presented the results for the test set after training the agent. After testing multiple combinations, the parameters which gave the best network performance are: $\alpha = 0.001$, $\nabla\beta = 0.95$ and $\lambda = 0.7$.

Table 1. Makespan (hours) for each approach, where RL results are averaged from 10 runs

Number of jobs		3	5	8	10
MILP		3.1	5	5	10
Heuristic		4.2	6	5.2	12.2
RL	best	4.0	5.1	5.1	12
	average	4.42	6.03	5.94	13.61

For the MILP results run times are under a second and under a minute for the first two instances, respectively. For the latter two, the algorithm was forced to stop after 30 minutes. All heuristic runs take less than a few seconds. For all RL results, the average of 10 runs is presented. Average run times for instances from 3 to 10 jobs are, respectively: 1 minute, 3 minutes, 5 minutes, 10 minutes.

5.1 Discussion

Average results using the RL agent are close to heuristic performance, however take much longer to compute. Even so, the best solutions found within 10 runs could always achieve better performance than the heuristic at least once. Currently, the choice of which operation to act upon is very randomized. A more reliable task selection, or even moving that effort to the agent itself, could make the agent scheduling more reliable and frequently closer to its demonstrated potential.

One observed problem when using the agent for scheduling is the quality of consecutive feasible solutions. After reaching its first feasible solution, usually only a small improvement upon it is found on that run, if at all. This pattern happens frequently independent of total iteration number or the quality of the first feasible solution. Yet, the initial feasible solution is generally quickly found. This indicates that the agent is better trained to find feasible solutions than to optimize them. Providing more training time and more diverse training sets could be helpful in correcting this imbalance.

Another common observed issue, especially on the bigger instances, is that gaps exist in the schedule. These gaps have no resource being for the whole duration. They are also of considerable size, sometimes well over 1 hour. The gaps could be erased by a post-processing routine for practical application, but for the presented comparison with other approaches these flaws were not removed.

6. CONCLUSIONS

The DRC flexible job shop is a complex problem with little research but direct applicability in industry, where the common approach is to use heuristic methods based on expert knowledge.

The presented RL approach on this problem is new and able to give competitive solutions for small problem instances, even if at the cost of longer computational times. In most runs the agent is able to outperform the implemented heuristic, if given enough time. Due to this extended run time, it is not yet suitable for practical implementation.

As future work, a less probabilistic (or better weighted) action selection should be considered. Both presented issues in section 5.1 could indicate that the agent does not correctly value moving backwards, which could signal a problem with the reward system.

Before real-world deployment, some additions to the formulation are needed. Slack times between operation subtasks need to be accounted for since machines can stay idle until workers are available. Other two important practical constraints are worker shifts and due dates.

Other uses could also be given to the agent. Since it gives a feasible solution relatively faster, it could be used to construct starting solutions for other methods. Another interesting application could be to use the agent in real-time for fixing deviations to the expected conditions, such as to reassign resources on machine breakdown.

REFERENCES

Araz, Ö.U. (2005). A Simulation Based Multi-criteria Scheduling Approach of Dual-Resource Constrained Manufacturing Systems with Neural Networks. In *AI 2005: Advances in Artificial Intelligence*, 1047–1053. Springer.

Cunha, M., Viegas, J.L., Sousa, M.C., and Vieira, S.M. (2019). Dual-resource Constrained Scheduling for Quality Control Laboratories. *IFAC-PapersOnLine*, 6.

Dhiflaoui, M., Nouri, H.E., and Driss, O.B. (2018). Dual-Resource Constraints in Classical and Flexible Job Shop Problems : A State-of-the-Art Review. *Procedia Computer Science*, 126, 1507–1515.

Ham, A. (2018). Scheduling of Dual Resource Constrained Lithography Production : Using. *IEEE Transactions on Semiconductor Manufacturing*, 31(1), 52–61.

Huiyuan, R., Lili, J., Xiaoying, X., and Muzhi, L. (2009). Heuristic Optimization for Dual-resource Constrained Job Shop Scheduling. In *2009 International Asia Conference on Informatics in Control, Automation and Robotics*, 485–488. IEEE.

Morshed, M.S., Jain, S.A., and Meeran, S. (2017). A State-of-the-art Review of Job-Shop Scheduling Techniques.

Nouri, H.E., Belkahla, O., and Khaled, D. (2016). Hybrid metaheuristics for scheduling of machines and transport robots in job shop environment. 808–828.

Pinedo, M.L. (2009). *Planning and Scheduling in Manufacturing and Services*. Springer, 2 edition.

Shen, L., Dauzère-pérès, S., and Neufeld, J.S. (2018). Solving the flexible job shop scheduling problem with sequence-dependent setup times. *European Journal of Operational Research*, 265(1), 503–516.

Sutton, R.S. and Barto, A.G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, 2nd edition.

Viegas, J.L., Cunha, M., Martins, M., Coito, T., Figueiredo, J., Sousa, J.M., and Vieira, S.M. (2019). A flexible heuristic for the dual-resource constrained scheduling problem in quality control laboratories [abstract]. In *30th European Conference on Operational Research*.

Zhang, W. and Dietterich, T.G. (2000). Solving Combinatorial Optimization Tasks by Reinforcement Learning. *Journal of Artificial Intelligence Research*, 1.