Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2007)

On a Successful Application of Multi-Agent Reinforcement Learning to Operations Research Benchmarks

Thomas Gabel and Martin Riedmiller
Neuroinformatics Group
Department of Mathematics and Computer Science, Institute of Cognitive Science
University of Osnabrück, 49069 Osnabrück, Germany
Email: thomas.gabel@uos.de, martin.riedmiller@uos.de

***Abstract*— In this paper, we suggest and analyze the use of approximate reinforcement learning techniques for a new category of challenging benchmark problems from the field of Operations Research. We demonstrate that interpreting and solving the task of job-shop scheduling as a multi-agent learning problem is beneficial for obtaining near-optimal solutions and can very well compete with alternative solution approaches. The evaluation of our algorithms focuses on numerous established Operations Research benchmark problems.**

I. Introduction

The empirical evaluation of reinforcement learning (RL) algorithms frequently focuses on established benchmark problems such as the cart-pole, the mountain car, or the bicycle benchmark. These problems are clearly defined and allow for a distinct comparison of RL methods, notwithstanding the fact that, from a practitioner's point of view, they are still far away from the problem sizes to be tackled in real-world problems. In this work, we want to bridge the gap between focusing on artificial RL benchmark problems and real-world applications. It is our goal to spotlight job-shop scheduling problems as a new class of benchmark problems of approximate RL algorithms that feature both the character of standardized, well-defined task descriptions as well as the property of representing application-oriented and extremely challenging problems.

Job-shop scheduling problems (JSSPs) are well-known to be NP-hard and have been in the focus of Operations Research since decades. In between, numerous problem instances have established as benchmarks and are frequently used to compare different analytical solution methods against one another. In this paper, we will start off by providing a thorough description of these types of problems and delineate why they are well suited to be employed as RL benchmarks, too. In this regard, we also discuss different possibilities on how to model a JSSP as a sequential decision problem. In Section II, we present MAPS (Multi-Agent Production Scheduling), our framework to solving job-shop scheduling problems with the help of RL and describe the learning algorithms we are utilizing. Section III presents our results of applying MAPS to the scheduling benchmarks mentioned, and Section IV concludes.

A. What is Job-Shop Scheduling?

The goal of scheduling is to allocate a specified number of jobs (also called tasks) to a limited number resources (also called machines) in such a manner that some specific objective is optimized. In job-shop scheduling n jobs must be processed on m machines in a given order. Each job j ($j \in \{1, \ldots, n\}$) consists of v_j operations $o_{j,1}, \ldots, o_{j,v_j}$ that have to be handled on a specific resource for a certain duration. A job is finished after completion of its last operation (completion time c_j).

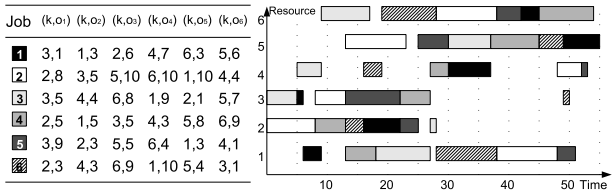

Fig. 1. Simple 6×6 benchmark problem instance (ft6) and a Gantt chart representation of an optimal solution.

Figure 1 shows a 6×6 (6 resources and 6 jobs, $m = n = 6$) problem instance from [1]. In this example, job 2 must first be processed on resource 2 for 8 time units, then go to resource 3 for 5 time steps, and so on. Resource 3 may start processing with job 1, 3, or 5. Over the years, numerous benchmark problem instances like this have been proposed and are publicly available (e.g. from the OR Library [2]). Most of them are, of course, much more complex and certain examples remained unsolved for decades. For other, larger-scale instances there is still no optimal solution known. Common characteristic of these JSSPs is that usually no recirculation is allowed, i.e. that each job has to be processed exactly once on each resource, implying that $v_j = m$. Though there are also stochastic scheduling problems, in the scope of this work we focus on deterministic ones only.

In general, scheduling objectives to be optimized all relate to the completion times of the jobs. For example, it may be desired to minimize the jobs' due date violations or the number of jobs that are late. In this paper, however, we focus on the objective of minimizing maximum makespan C_{max}, which is the length of the schedule ($C_{max} = max\{c_j\}$), since most publications on results achieved for JSSP benchmarks focus on that objective, too. The Gantt chart in Figure 1 shows an optimal solution for that 6×6 benchmark ($C_{max} = 55$).

B. MDP vs. MMDP Modelling of Scheduling Problems

Basically, there are two ways of modelling a JSSP as Markov Decision Process (MDP, [3]). The straightforward alternative is to interpret a scheduling problems as a *single* MDP. We can represent the state $s(t) \in S$ of the system by the situation of all resources as well as the processing status of all jobs. Additionally, a terminal state s_f describes the situation when all jobs have been finished, i.e. $s_k(t) = s_f$ for all $t \geq C_{max}$. An action $a(t) \in A$ describes the decision of which jobs are to be processed next on the resources. Moreover, we can say that the overall goal of scheduling is to find a policy $\pi^\star$ that minimizes production costs $c(s, a, t)$ accumulated over time

$$\pi^\star := \min_{\pi} \sum_{t=1}^{C_{max}} c(s, a, t). \tag{1}$$

Costs may depend on the current situation, as well as on the selected decision, and have to relate closely to the desired optimization goal (e.g. they may occur when a job violates its due date).

The second modelling alternative extends the first one by interpreting JSSPs as a *multi-agent* Markov Decision Process (MMDP, [4]). Here, we associate to each of the m resources an agent k that locally decides on elementary actions $a_k(t)$. So, an element $a(t) = (a_1(t), \ldots, a_m(t))$ of the joint action space is a vector composed of m elementary actions that are assumed to be executed concurrently. For example, starting to process the example in Figure 1 the agent associated to resource $k = 3$ must decide which job to process next at this resource, where its set of actions at $t = 1$ is $A_3(1) = \{1, 3, 5\}$.

In scheduling theory, a distinction between predictive production scheduling (also called analytical scheduling or offline-planning) and *reactive* scheduling (or online control) is made [5]. While the former assumes complete knowledge over the tasks to be accomplished, the latter is concerned with making local decisions independently. Obviously, a single MDP modelling gives rise to analytical scheduling, whereas the MMDP formulation corresponds to performing reactive scheduling. In the following we prefer the MMDP modelling, hence, doing reactive scheduling, for the following reasons.

- Reactive scheduling features the advantage of being able to react to unforeseen events (like a machine breakdown) appropriately without the need to do complete replanning.
- Operations Research has to the bigger part focused on analytical scheduling and yielded numerous excellent algorithms (e.g. branch-and-bound) capable of finding the optimal schedule in reasonable time when the problem dimension ($m \times n$) is not too large and when being provided with complete knowledge over the entire problem. By contrast, reactive scheduling approaches are decentralized by definition and, hence, the task of making globally optimal decisions is aggravated. Accordingly, many interesting open research questions arise.
- From a practical point of view, a centralized control cannot always be instantiated, why the MMDP formulation is of higher impact to real-world applications.

C. Related Work

Job-shop scheduling has received an enormous amount of attention in the research literature. Classical approaches to solving job-shop scheduling problems cover, for instance, disjunctive programming, branch-and-bound algorithms, or the shifting bottleneck heuristic—a thorough overview is given in [6]. Moreover, there is a large number of local search procedures to solve job-shop scheduling problems. These include beam search, simulated annealing, or tabu search (see [7] for details). Furthermore, various different search approaches have been suggested based on genetic algorithms (e.g. [8] or [9]).

In contrast to these analytical methods yielding to search for a single problem's best solution, this paper's focus is on the class of reactive scheduling techniques. Most relevant references for reactive scheduling cover simple as well as complex dispatching priority rules (see [10] or [11]). Focusing on job-shop scheduling with blocking and no-wait constraints, in [12] the authors develop heuristic dispatching rules (such as AMCC, cf. Section III) that are suitable for online control.

Using our reactive scheduling approach, the finally resulting schedule is not calculated beforehand, viz before execution time. Insofar, our RL approach to job-shop scheduling is very different from the work of Zhang and Dietterich [13] who developed a repair-based scheduler that is trained using the temporal difference reinforcement learning algorithm and that starts with a critical-path schedule and incrementally repairs constraint violations. Mahadevan et al. have presented an average-reward reinforcement learning algorithm for the optimization of transfer lines in production manufacturing which resembles a specialization of a scheduling problem. They show that the adaptive resources are able to effectively learn when they have to request maintenance [14], and that introducing a hierarchical decomposition of the learning task is beneficial for obtaining superior results [15].

II. Reinforcement Learning for Multi-Agent Production Scheduling

In this section, we first introduce MAPS (multi-agent production scheduling), our approach to solving job-shop scheduling problems by adopting a multi-agent perspective. The key concepts of MAPS provide possible answers to questions regarding a suitable state and action modelling in the considered multi-agent domain, as well as to questions on how to benefit from employing a reinforcement learning algorithm at the core of the approach. Besides a short review on the basic ideas of our relevant previous work [16], [17], we suggest several extensions to render MAPS applicable to the types of large-scale benchmark problems we are considering in this paper.

A. Foundations of MAPS

As indicated, the global decision $a(t)$ is a vector of single decisions $a_k(t)$ that are made by agents associated to the m resources. For an agent taking an action means deciding

which job to process next out of the set $A_k(t)$ of currently waiting jobs at the corresponding resource k. Accordingly, an agent cannot take an action at each discrete time step t, but only after its resource has finished one operation, because each resource can only work on processing one job at a time. Therefore, the agent makes its decisions after time intervals whose lengths Δt_k are determined by the durations of the operations processed.

While in the previous work mentioned we primarily considered settings where only one learning agent was part of the system, we now turn to real multi-agent settings. In the multi-agent literature, there is a distinction between joint-action learners and independent learners [18]. The former can observe their own, as well as the other agents' action choices. Consequently, in that case the MMDP can be reverted to a single-agent MDP with an extended action set and be solved by some standard method. In this paper, however, we concentrate on the case of independent learners, where each agent k knows only its own contribution to the joint action.

The global view $s(t)$ on the plant, including the situation at all resources and the processing status of all jobs, would allow some classical solution algorithm (like a branch-and-bound method), being run within each agent, to construct a disjunctive graph for the problem at hand and solve it. In this respect, however, we introduce a significant aggravation of the problem: First, we require a reactive scheduling decision in each state to be taken in real-time, i.e. we do not allot arbitrary amounts of computation time. Second, we restrict the amount of state information the agents get. Instead of the global view, each agent k has a local view $s_k(t)$ only, containing condensed information about its associated resource and the jobs waiting there. On the one hand, this restriction increases the difficulty in finding an optimal schedule. On the other hand, it allows for complete decentralization in decision-making since each agent is provided with information only that are relevant for making a local decision at resource k. This is particularly useful in applications where no global control can be instantiated and where communication between distributed working centers is impossible. Nevertheless, the number of features provided to a single agent, viz the local view, is still large and forces us tackle a high-dimensional continuous state-action space.

B. Using RL to Learn Dispatching Policies

When there is no explicit model of the environment available, Q learning [19] is one of the reinforcement learning methods of choice to learn a value function for the problem at hand, from which a control policy can be derived. The Q function $Q : S \times A \to \mathbb{R}$ expresses the expected costs when taking a certain action in a specific state. The Q update rule directly updates the values of state-action pairs according to

$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha(c(s,a,s') + \gamma \min_{b \in A(s')} Q(s',b)) \tag{2}$$

where α is the learning rate, γ the discount factor, and where the successor state s' and the immediate costs $c(s,a,s')$ are generated by simulation or by interaction with a real process.

Since our approach enforces a distributed decision-making by independent agents, the Q update rule is implemented within each learning agent and adapted to the local decision process ($\alpha = 1$ for better readability):

$$\begin{aligned} Q_k(s_k(t), a_k(t)) \quad := \quad & c(s,a,t) \\ & +\gamma \min_{b \in A_k(t_{next})} Q_k(s_k(t_{next}), b) \end{aligned} \tag{3}$$

This learning rule establishes a relation between the local dispatching decisions as made by agent k and the overall optimization goal, since the global immediate costs are taken into consideration (e.g. costs caused due to a tardy job in the system).

A crucial precondition for MAPS to learning to make sophisticated scheduling decisions is that the global direct costs (as feedback to the learners) coincide with the our overall objective of scheduling (minimizing maximal makespan C_{max}). It is well-known that the makespan of a schedule is minimized, if as many resources as possible are processing jobs concurrently and if as few as possible resources with queued jobs are in the system: Usually, a high utilization of the resources implies a minimal makespan [7]. This argument gives rise to defining

$$c(s,a,t) := \sum_{k=1}^{m} |\{j \mid j \text{ queued at } k\}| \tag{4}$$

so that high costs are incurred when many jobs that are waiting for further processing are in the system and, hence, the overall utilization of the resources is poor.

If we assume convergence of Q_k to the optimal local value function $Q_k^\star$, we obtain a predictor of the expected accumulated global costs that will arise, when in state s_k a job denoted by a_k would be processed next. Then, a policy π that exploits Q_k *greedily* will lead to optimized performance of the scheduling agent. A policy greedily exploiting the value function chooses its action $a_k(t)$ as follows

$$a_k(t) := \pi(s_k, a_k, t) = \min_{b \in A_k(t)} Q_k(s_k(t), b) \tag{5}$$

We distinguish between the learning and the application phases of the agents' dispatching policies. During the latter, the learned Q_k function is exploited greedily according to Equation 5. During the former, however, updates to Q_k are made (cf. Equation 4) and an exploration strategy is pursued which chooses random actions with some probability.

Assuming a typical $m \times n$ job-shop scheduling problem, it is easy to see that the transition graph of the system is acyclic and the number of states till reaching s_f is finite. Therefore, all policies are always proper and the problem horizon is finite, which is why γ can safely be set to one (no discounting).

When considering a single job-shop problem, the number of possible states is, of course, finite since we consider deterministic JSSPs. The motivation of our research, however, is not to just focus on individual problem instances, but on arbitrary ones. Accordingly, we need to assume the domain of Q to be infinite or even continuous and will have to employ function approximation mechanisms to represent it.

C. Representing the State-Action Value Function

Since we consider value functions Q_k with infinite domain, we need to employ some function approximation technique to represent it. In this work, we use multi-layer perceptron neural networks to represent the state-action value function. On the one hand, feed-forward neural networks are known to be capable of approximating arbitrarily closely any function $f : D \rightarrow \mathbb{R}$ that is continuous on a bounded set D [20]. On the other hand, we aim at exploiting the generalization capabilities of neural networks yielding general dispatching policies, i.e. policies which are not just tuned for the situations encountered during training, but which are general enough to be applied to unknown situations, too.

1) Net In-/Output: Input to the neural net is the local view, i.e. the features describing the situation of the resource as well as single waiting jobs. The neural network's output value $Q_k(s_k, a_k)$ directly reflects the priority value of the job corresponding to action a_k depending on the current state s_k.

We represent states $s_k \in S$ and actions/jobs $a_k \in A_k$ by feature vectors generated by the resources' local view. A thorough feature description is beyond the scope of this paper. We note, however, that the features have to exhibit some relation to the future expected costs, hence to the makespan, and must allow for a comprehensive characterization of the current situation. So, state features depict the current situation of the resource describing its processing state and the set A_k of jobs currently waiting at that resource. Action features a_k characterize single jobs from A_k currently selectable by k. Here, we aim at describing makespan-oriented properties of individual jobs (like processing time or relative waiting time indices) as well as immediate consequences to be expected when processing that job next.

2) Fitted Q Iteration: In Section II-B, we have pointed to the distinction between the learning and the application phase of MAPS. During learning, a set $\mathcal{S}_L$ of scheduling problem instances is given—these problems are processed on the plant repeatedly, where the agents are allowed to schedule jobs randomly with some probability, obtaining new experiences that way. In principle, it is possible to perform an update on the state-action value function according to Equation 4 after each state transition. However, to foster fast improvements of the learned policy by exploiting the training data as efficiently as possible, we make use of fitted Q iteration.

Fitted Q iteration denotes a batch (also termed off-line) reinforcement learning framework, in which an approximation of the optimal value function is computed from a finite set of four-tuples [21]. The set of four-tuples $\mathbb{T} = \{(s^i, a^i, c^i, \overline{s}^i) | i = 1, \ldots, p\}$ may be collected in any arbitrary manner and corresponds to single "experience units" made up of states s^i, the respective actions a^i taken, the immediate costs c^i incurred, as well as the successor states $\overline{s}^i$ entered.

The basic algorithm takes $\mathbb{T}$, as well as a regression algorithm as input, and after having initialized $\tilde{Q}$ and a counter q to zero, repeatedly processes the following three steps until some stop criterion becomes true:

- increment q
- build up a training set $\mathbb{F}$ for the regression algorithm according to: $\mathbb{F} := \{(in^i, out^i) | i = 1, \ldots, p\}$ where $in^i = (s^i, a^i)$ and $out^i = c^i + \gamma \min_{b \in A(s^i)} \tilde{Q}_{q-1}(\overline{s}^i, b)$
- use the regression algorithm and the training set $\mathbb{F}$ to induce an approximation $\tilde{Q}^q : S \times A \rightarrow \mathbb{R}$

Subsequently, we consider neural fitted Q iteration (NFQ, [22]), a realization of fitted Q iteration where multi-layer neural networks are used to represent the Q function and an enhanced network weight update rule is employed. NFQ is an effective and efficient reinforcement learning method for training a Q value function that requires reasonably few interactions with the scheduling plant to generate dispatching policies of high quality. We will develop an optimistic version of NFQ to be used in the scope of this work in Section II-D.

3) Convergence Problems: A critical question concerns the convergence of the learning technique to a (near-)optimal decision policy when used in conjunction with value function approximation. In spite of a number of advantages, neural networks are known to belong to the class of "exaggerating" value function approximation mechanisms [23] and as such feature the potential risk of diverging. There are, however, several methods for coping with the danger of non-convergent behavior of a reinforcement learning method and to reduce the negative effects of phenomenons like chattering and policy degradation. Since a thorough discussion of that concern is beyond the scope of this paper, we refer to relevant literature [24]–[26].

In order to be able to safely apply our learning approach to reactive scheduling to complex benchmark problems, we rely on *policy screening*, a straightforward, yet computationally intensive method for selecting high-quality policies in spite of oscillations occurring during learning (suggested by Bertsekas and Tsitsiklis [24]): We let the policies generated undergo an additional evaluation based on simulation (by processing problems from a separate set of screening scheduling problems $\mathcal{S}_S$), which takes place in between single iterations of the NFQ learning algorithm. As a result, we can determine the actual performance of the policy represented by the Q function in each iteration of the algorithm and, finally, detect and return the best policy created.

D. Optimistic Neural Fitted Q Iteration

Remember that we take a fully distributed view on multi-agent scheduling: The agents are completely decoupled from one another, get local state information, and are not allowed to share information via communication. Thus, unless the policies of other agents are stable, the environment as experienced by a single agent appears to be non-stationary. Furthermore, a crucial problem of cooperative independent learners is their inability to attribute the global costs correctly to different joint actions: Taking some action a_k in some state may at different

times incur different costs/state transitions depending on the other agents' action choices.

Given the fact that the scheduling benchmarks to which we intend to apply MAPS are deterministic, we can employ a powerful mechanism for cooperative multi-agent learning during the learning phase permitting all agents to learn in parallel. Lauer and Riedmiller [27] suggest an algorithm for distributed Q learning of independent learners using the so called optimistic assumption (OA). Here, each agent *assumes* that all other agents act optimally, i.e. that the combination of all elementary actions forms an optimal joint-action vector. Given the standard prerequisites for Q learning, it can be shown that the optimistic assumption Q iteration rule (with current state s, action a, successor state s')

$$Q_k(s,a) := \max\{Q_k(s,a), r(s,a) + \gamma \max_{b \in A(s')} Q_k(s',b)\} \quad (6)$$

to be applied to agent-specific local Q functions Q_k converges to the optimal $Q^\star$ function in a deterministic environment, if initially $Q_k \equiv 0$ for all k and if the immediate rewards $r(s,a)$ are always larger or equal zero. Hence, the basic idea of that update rule is that the expected returns of state-action pairs are captured in the value of Q_k by successively taking the maximum. For more details on that algorithm and on the derivation of coordinated agent-specific policies we refer to [27].

For the benchmark problems we are tackling in this work, we have to take the following two facts into consideration: First, we are using the notion of costs instead of rewards, so that small Q values correspond to "good" state-action pairs incurring low expected costs (though we assume all immediate global costs to be larger or equal zero, cf. Equation 4). Second, we perform batch-mode learning by first collecting a large amount of training data (state transition tuples) and then calculating updated values to the Q functions.

To comply with these requirements, we suggest an offline reinforcement learning method that adapts and combines neural fitted Q iteration and the optimistic assumption Q update rule. In Figure 2, we give a pseudo-code realization of neural fitted Q iteration using the optimistic assumption (OA-NFQ). The distinctive feature of that algorithm lies in step 1 where a reduced (optimistic) training set $\mathbb{O}$ with $|\mathbb{O}| = p'$ is constructed from the original training tuple set $\mathbb{T}$ ($|\mathbb{T}| = p \geq p'$). In a deterministic environment where scheduling scenarios from a fixed set $\mathcal{S}_L$ of problems are repeatedly processed, the probability of entering some state s_k more than once is larger than zero. If in s_k a certain action $a_k \in A(s_k)$ is taken again when having entered s_k for a repeated time, it may eventually incur very different costs because of different elementary actions selected by other agents. The definition of $\mathbb{O}$ basically realizes a partitioning of $\mathbb{T}$ into p' clusters with respect to identical values of s_k and a_k (steps 1a and 1b). In step 1c the optimistic assumption is applied which corresponds to implicitly assuming the best joint action vector covered by the experience collected so far, i.e. assuming the other agents have taken optimal elementary actions that are most appropriate for the current state and the agent's own elementary action a_k. Thus, the target value out^j for some state-action pair $in^j = (s^j, a^j)$ is the minimal sum of the immediate costs and discounted costs to go over all tuples $(s^j, a^j, \cdot, \cdot) \in \mathbb{T}$.

Input: number of Q iterations $N \in \mathbb{N}$, training set

$$\mathbb{T}_k = \{(s_k(t), a_k(t), c(s,a,t), s_k(t_{next})) \mid t \in \text{set of decision time points}\}$$

for better readability abbreviated as $\mathbb{T}_k = \{(s^i, a^i, c^i, \overline{s}^i) | i = 1, \ldots, p\}$

Output: state-action value function $Q_k^{(top)}$

init $Q_k^{(0)} \equiv 0$

for $q = 0$ **to** $N-1$ *//Q iterations*

1) **generate** optimistic training set

$$\mathbb{O} = \{(in^j, out^j) | j = 1, \ldots, p'\} \text{ with } p' \leq p$$

with (a) $\forall in^j \ \exists i \in \{1, \ldots, p\}$ with $in^j = (s^i, a^i)$

(b) $in^i \neq in^j \ \forall i \neq j \ (i, j \in \{1, \ldots, p'\})$

(c) $$out^j := \min_{\substack{(s^i, a^i, c^i, \overline{s}^i) \in \mathbb{T}_k \\ (s^i, a^i) = in^j}} \left(c_i + \gamma \min_{b \in A(s')} Q_k^{(q)}(s'_i, b) \right)$$

2) **train** neural network given $\mathbb{O}$ to induce a new Q function $Q_k^{(q+1)}$

3) **do** policy screening to evaluate $\pi_{Q_k^{(q+1)}}$ and memorize $Q_k^{(top)} := Q_k^{(q+1)}$ in case of a policy improvement

Fig. 2. OA-NFQ, a realization of neural fitted Q iteration in deterministic multi-agent settings based on the optimistic assumption. The value function's superscripts in $Q^{(q)}$ denote the number q of the respective Q iteration, the subscripts k indicate the agent.

After having constructed the training set $\mathbb{O}$ any suitable neural network training algorithm can be employed for the regression task at hand (e.g. standard backpropagation or the faster Rprop algorithm [28] we use). Apart from those net training issues, the pseudo-code of OA-NFQ in Figure 2 reflects also the policy screening technique (cf. Section II-C): In between individual Q iterations we let the current value function $Q_k^{(q)}$ and the corresponding dispatching policy, respectively, undergo an additional evaluation based on simulating a number of screening scheduling problems from a set $\mathcal{S}_S$. Via that mechanism the best Q iteration and its belonging Q function $Q_k^{(top)}$ is detected and finally returned.

We need to stress that in presence of using a neural value function approximation mechanism to represent Q and providing agents with local view information only, neither the convergence guarantees for certain (averaging) types of fitted Q iteration algorithms (see Ernst et al. [21] for a thorough discussion), nor the convergence proof of the OA Q learning algorithm (supporting finite state-action spaces, only) endure. Nevertheless, it is possible to obtain impressive empirical results despite the approximations we employ, as we will show in Section III.

III. Benchmarking

In this section, we evaluate the use of approximate reinforcement learning for the Operations Research benchmark

problems that are in the center of our interest. In particular, we want to address the questions, if it is possible to use the MAPS approach, utilizing the algorithms we have described in Section II, to let the agents acquire high-quality dispatching policies for problem instances of current standards of difficulty. Furthermore, we want to investigate, whether the learnt policies generalize to other, similar benchmark problems, too.

Benchmark problems abz7-9 were generated by Adams et al. [29], problems orb1-9 are from Applegate and Cook [30], and finally, problems la01-20 are due to Lawrence [31].

A. Preliminary Remarks

Within this evaluation, we compare three different types of algorithms to solve scheduling problems, each of them being subject to different restrictions.

1) **Global View Dispatching Rules** perform reactive scheduling and correspond to the MMDP view on job-shop scheduling. They take local dispatching decisions at the respective resources, but are allowed to get hold of more than just local state information. Instances of this group are the SQNO rule (heuristic violating the local view restriction by considering information of the queue lengths at the resources where the waiting jobs will have to be processed next) or the powerful AMCC rule (heuristic to avoid the maximum current C_{max} based on the idea to repeatedly enlarge a consistent selection, given a general alternative graph representation of the scheduling problem [12]).
2) **Local View Dispatching Rules** perform reactive scheduling as well and make their decisions which job to process next based solely on their local view on the respective resource. In the following, we consider three instances of this group (LPT/SPT rule chooses operations with longest/shortest processing times first, FIFO rule considers how long operations had to wait at some resource).
3) The **MAPS approach** presented in this paper.

Additionally, for each benchmark considered, we provide the theoretical optimum, i.e. the makespan of a schedule with minimal makespan. Regarding the restrictions MAPS is subject to (MMDP interpretation of a JSSP with local view) a comparison to group 2) is most self-evident. However, by approaching or even surpassing the performance of algorithms from group 1), we can make a case for the power of the MAPS approach.

B. Example Benchmark

We start with the notorious problem ft10 proposed by Fisher and Thompson published in the book by Muth and Thompson [1], that had remained unsolved for more than twenty years. Here, during the learning phase, MAPS processes $\mathcal{S}_L = \{ft10\}$ repeatedly on the simulated plant, where the agents associated to the ten resources follow ε-greedy strategies ($\varepsilon = 0.5$) and sample experience while adapting their behaviors using the OA-NFQ algorithm in conjunction with the policy screening method (we set $\mathcal{S}_L = \mathcal{S}_S$, i.e. the screening set contains the same problems as the training set).

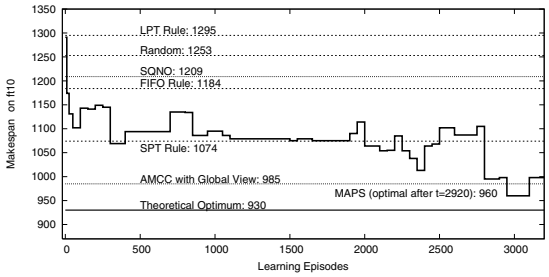

Fig. 3. Learning process for the notorious ft10 problem.

Figure 3 visualizes the learning progress for the ft10 instance. We compare the performance the three groups of methods listed in Section III-A plus a purely random dispatching policy. The best solution found by MAPS was discovered after 2920 repeated processings of $\mathcal{S}_L$. The makespan $C_{max} = 960$ of the corresponding schedule thus has a relative error of 3.2% compared to the optimal schedule. We note that we have detected the optimal learned dispatching policy (represented by the agents' neural networks representing their Q functions) by means of the policy screening method described in Section II-C.

C. Benchmark Suites

Next, we studied the effectiveness of our agent-based scheduling approach for a large number of different-sized benchmark problem suites, ranging from job-shop scheduling problems with five resources and ten jobs to fifteen resources and twenty jobs. We allowed the agents to sample training data tuples in an ε-greedy manner for maximally 25000 processings of $\mathcal{S}_L$ with $\mathcal{S}_L = \mathcal{S}_S$ and permitted intermediate calls to OA-NFQ ($N = 20$ iterations of the Q iteration loop) in order to reach the vicinity of a near-optimal Q function as quickly as possible.

For a better illustration of the findings we have grouped the results on individual benchmark problems into classes with respect to the numbers of resources and jobs to be processed (Figure 4). For the 5×15 (la6-10) and 5×20 (la11-15) benchmark problems, the optimal solution can be found by our learning approach in all cases, and for the 5×10 (la1-5) and 10×10 (la16-20, orb1-9) sets, only a small relative error of less than ten percent compared to the optimal makespan remains ($3.4/4.0/7.2\%$). As to be expected, dispatching rules, even those disposing of more than just local state information (like AMCC or SQNO), are clearly outperformed. For the larger benchmarks (abz) involving 15 resources and 20 jobs per problem instance the relative error increases to 10.6%, but the rule-based schedulers can be outperformed still.

D. Generalization Capabilities

To empirically investigate the generalization capabilities of the learned dispatching policies, we designed a further

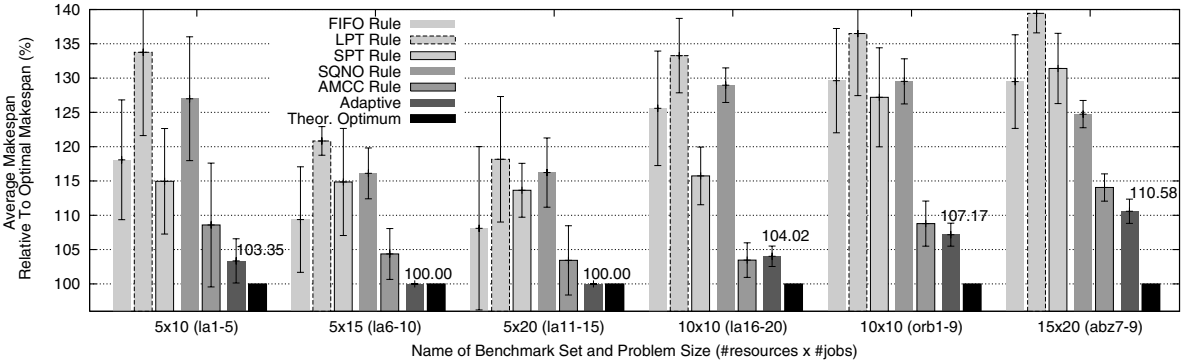

Fig. 4. For different sets of benchmark problems with equal size, this figure visualizes the average performance of different approaches in terms of minimized makespan of the resulting schedules. The results are given relative to the makespan of the optimal schedule (100%, black data series). Data series colored in light gray correspond to static rules having local state information only (group 2), whereas medium gray-colored ones are not subject to that restriction (group 1). The dark gray-colored data series corresponds to the MAPS approach we have suggested (group 3, performance reported belongs to the makespan achieved when *training* for single benchmark problems), which is restricted to the local view, too. Error bars indicate the standard deviations of the relative errors over the problem instances within the respective benchmark problem sets.

Benchmark Suite Name		$\mathcal{S}_{la}^{5\times15}$		$\mathcal{S}_{orb}^{10\times10}$	
Problem Instances		$la06,\ldots,la10$		$orb1,\ldots,orb9$	
Local View	FIFO	1003.6	9.4%	1164.3	29.6%
	LPT	1108.8	20.9%	1226.1	36.5%
	SPT	1054.0	14.9%	1142.6	27.1%
Global View	AMCC	955.6	4.2%	977.1	8.8%
	SQNO	1065.4	16.1%	1163.3	29.5%
MAPS (local view)		**951.6**	**3.7%**	**1065.1**	**18.6%**
with Cross-Validation		5-fold		3-fold	
Avg. Optimum ($C_{max,opt}^{avg}$)		*917.6*		*898.2*	

TABLE I

GENERALIZATION CAPABILITIES: DURING ITS APPLICATION PHASE, MAPS' LEARNED DISPATCHING POLICIES ARE USED FOR PROBLEMS NOT COVERED DURING TRAINING. AVERAGE MAKESPAN AND REMAINING ERRORS RELATIVE TO THE OPTIMUM ARE PROVIDED.

experiment. Here, the learning agents were presented three sets of scheduling problems

- the training set $\mathcal{S}_L$ for the learning phase,
- the screening set $\mathcal{S}_S$ for intermediate policy screening rollouts (as before, we set $\mathcal{S}_L = \mathcal{S}_S$),
- and an application set $\mathcal{S}_A$ containing independent problem instances to evaluate the quality of the learning results on problem instances the agents have not seen before ($\mathcal{S}_L \cap \mathcal{S}_A = \emptyset$).

Of course, it would be unrealistic to expect the dispatching policies that were trained using, for instance, a training set with 5×15 problems, to bring about reasonable scheduling decisions for very different problems (e.g. for 10×10 benchmarks). Therefore, we have conducted experiments for benchmark suites $\mathcal{S}$ consisting of problems with identical sizes that were provided by the same authors. From an applicatory point of view, this assumption is appropriate and purposeful, because it reflects the requirements of a real plant where usually variations in the scheduling tasks to be solved occur according to some scheme and depending on the plant layout, but not in an entirely arbitrary manner.

Moreover, since $|\mathcal{S}|$ is rather small under these premises, we performed ν-fold cross-validation on $\mathcal{S}$, i.e. we disjointed $\mathcal{S}$ into $\mathcal{S}_L$ and $\mathcal{S}_A$, trained on $\mathcal{S}_L$ and assessed the performance of the learning results on $\mathcal{S}_A$, and finally, repeated that procedure ν times to form average values.

In Table I we summarize the learning results for a benchmark suite of 5×15 problems $\mathcal{S}_{la}^{5\times15} = \{la06,\ldots,la10\}$ as well as for the more intricate suite of 10×10 problems $\mathcal{S}_{orb}^{10\times10} = \{orb1,\ldots,orb9\}$. We emphasize that the average makespan values reported for MAPS correspond to its performance on independent test problem instances, i.e. to scheduling scenarios that were not included in the respective training sets $\mathcal{S}_L$[1] during cross-validation. From that numbers (average remaining errors of $3.7\%/18.6\%$ compared to the theoretic optimum) it is obvious that all static local view dispatchers, to which MAPS must naturally be compared, are clearly outperformed. Interestingly, for the $\mathcal{S}_{la}^{5\times15}$ problem suite not just dispatching rules working under the same conditions as MAPS, but even the AMCC rule is beaten, which exhaustively benefits from its global view on the plant. For the $\mathcal{S}_{orb}^{10\times10}$ suite, AMCC brings about better performance than MAPS which is logical since it was capable of yielding nearly the same results when training MAPS for single 10×10 problem instances in the experiment described in Section III-C.

We expect that, in future work, we will be able to further boost the performance of MAPS. In its current version MAPS can generate schedules of the class $\mathbb{S}_n$ of *non-delay* schedules exclusively: If a resource has finished processing one operation and has at least one job waiting, the dispatching agent immediately continues processing by picking one of the waiting jobs. From scheduling theory, however, it is known that for certain scheduling problem instances the optimal schedule may be

[1]Concerning the performance on the training problem instances from $\mathcal{S}_L$ (these values are not included in Table I), MAPS achieves a relative remaining error of only 0.3% for the la problems and 13.1% for the orb problems.

a delay schedule from the set of active schedules $\mathbb{S}_a \supsetneq \mathbb{S}_n$, i.e. a schedule where some resource has to remain idle for some time units in order to achieve minimal makespan. As a consequence, MAPS is currently able to produce near-optimal schedules from $\mathbb{S}_n$ and may miss the best schedule possible, though in many cases the optimum is indeed found (cf. Figure 4). Yet, an extension of MAPS towards delay schedules depicts an important and promising issue for future work.

IV. Conclusion

Numerous instances of job-shop scheduling problems have established themselves as prevalent benchmark problems in the field of Operations Research. In this paper, we suggested the interpretation of JSSPs as sequential decision processes and to use them as a new class of benchmark problems for approximate reinforcement learning methods.

Although it is possible to adopt a global view on a given scheduling problem and model it as a single MDP, we decided to interpret and solve it as a multi-agent learning problem using our MAPS approach relying on reinforcement learning. On the one hand, we therefore have to cope with a problem complication due to independently learning agents. But, on the other hand, we derive the benefit of being enabled to perform reactive scheduling including the capability to react to unforeseen events. Furthermore, a decentralized view on a scheduling task is of higher relevance to practice since a central control cannot always be instantiated.

In addition to introducing the integral concepts and modelling specifics of MAPS we also presented a new reinforcement learning method for deterministic multi-agent environments (OA-NFQ). This algorithm realizes a combination of data-efficient batch-mode reinforcement learning in conjunction with neural value function approximation, and the utilization of an optimistic inter-agent coordination.

Despite the numerous approximations that we have made, the empirical part of this paper contains several convincing results for classical OR benchmarks. Obviously, the dispatching policies our learning agents acquire clearly surpass traditional dispatching rules and, in some cases, are able to reach the theoretically optimal solution. Moreover, the acquired policies feature generalization capabilities, being adequate for similar scheduling problems not covered during the learning phase.

Acknowledgment

This research has been supported by the German Research Foundation (DFG) under grant number Ri-923/2-3.

References

[1] J. Muth and G. T. (Eds.), *Industrial Scheduling*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1963.

[2] J. Beasley, "Or-library," 2005, `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`.

[3] M. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. USA: Wiley-Interscience, 2005.

[4] C. Boutilier, "Sequential Optimality and Coordination in Multiagent Systems," in *Proceedings of IJCAI-99*. Stockholm, Sweden: Morgan Kaufmann, 1999, pp. 478–485.

[5] J. Blazewicz, K. Ecker, G. Schmidt, and J. Weglarz, *Scheduling in Computer and Manufacturing Systems*. Berlin: Springer, 1993.

[6] E. Pinson, "The Job Shop Scheduling Problem: A Concise Survey and Some Recent Developments," in *Scheduling Theory and Applications*, P. Chretienne, E. Coffman, and J. Lenstra, Eds., 1995, pp. 177–293.

[7] M. Pinedo, *Scheduling. Theory, Algorithms, and Systems*. USA: Prentice Hall, 2002.

[8] J. Bean, "Genetics and Random Keys for Sequencing and Optimization," *ORSA Journal of Computing*, vol. 6, pp. 154–160, 1994.

[9] B. Ombuki and M. Ventresca, "Local Search Genetic Algorithms for the Job Shop Scheduling Problem," *Applied Intelligence*, vol. 21, pp. 99–109, 2004.

[10] S. Panwalkar and W. Iskander, "A Survey of Scheduling Rules," *Operations Research*, vol. 25, pp. 45–61, 1977.

[11] K. Bhaskaran and M. Pinedo, "Dispatching," in *Handbook of Industrial Engineering*, G. Salvendy, Ed., 1977, pp. 2184–2198.

[12] A. Mascis and D. Pacciarelli, "Job-Shop Scheduling with Blocking and No-Wait Constraints," *European Journal of Operational Research*, vol. 143, pp. 498–517, 2002.

[13] W. Zhang and T. G. Dietterich, "A reinforcement learning approach to Job-shop Scheduling," in *Proceedings of the International Joint Conference on Artificial Intellience*, 1995.

[14] S. Mahadevan, N. Marchalleck, T. Das, and A. Gosavi, "Self-Improving Factory Simulation Using Continuous-Time Average-Reward Reinforcement Learning," in *Proceedings of ICML 1997*. Nashville, USA: Morgan Kaufmann, 1997, pp. 202–210.

[15] G. Wang and S. Mahadevan, "Hierarchical Optimization of Policy-Coupled Semi-Markov Decision Processes," in *Proceedings of ICML 1999*. San Francisco, USA: Morgan Kaufmann, 1999, pp. 464–473.

[16] S. Riedmiller and M. Riedmiller, "A Neural Reinforcement Learning Approach to Learn Local Dispatching Policies in Production Scheduling," in *IJCAI 1999*, Stockholm, Sweden, 1999, pp. 764–771.

[17] W. Hunger and M. Riedmiller, "Scheduling with Adaptive Agents - an Empirical Evaluation," in *Proceedings of EWRL-5, European Workshop on Reinforcement Learning*, 2001.

[18] C. Claus and C. Boutilier, "The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems," in *Proceedings of AAAI-98*. Menlo Park, USA: AAAI Press, 1998.

[19] C. Watkins and P. Dayan, "Q-Learning," *Machine Learning*, vol. 8, 1992.

[20] K. Hornick, M. Stinchcombe, and H. White, "Multilayer Feedforward Networks Are Universal Approximators," *Neural Networks*, vol. 2, pp. 359–366, 1989.

[21] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-Based Batch Mode Reinforcement Learning," *Journal of Machine Learning Research*, 2005.

[22] M. Riedmiller, "Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method," in *Machine Learning: ECML 2005*. Porto, Portugal: Springer, 2005.

[23] G. J. Gordon, "Stable Function Approximation in Dynamic Programming," in *Proceedings of ICML 1995*. San Francisco, USA: Morgan Kaufmann, 1995, pp. 261–268.

[24] D. Bertsekas, M. Homer, D. Logan, S. Patek, and N. Sandell, "Missile Defense and Interceptor Allocation by Neuro-Dynamic Programming," in *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 30, 2000, pp. 42–51.

[25] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro Dynamic Programming*. Belmont, USA: Athena Scientific, 1996.

[26] R. Munos, "Error Bounds for Approximate Policy Iteration," in *Proceedings of the International Conference on Machine Learning*, 2003.

[27] M. Lauer and M. Riedmiller, "An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems," in *Proceedings of ICML 2000*. Stanford, USA: AAAI Press, 2000, pp. 535–542.

[28] M. Riedmiller and H. Braun, "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm," in *Proceedings of ICNN 1993*, San Francisco, USA, 1993, pp. 586–591.

[29] J. Adams, E. Balas, and D. Zawack, "The Shifting Bottleneck Procedure for Job Shop Scheduling," *Management Science*, vol. 34, pp. 391–401, 1988.

[30] D. Applegate and W. Cook, "A Computational Study of the Job-Shop Scheduling Problem," *ORSA Journal on Computing*, vol. 3, pp. 149–156, 1991.

[31] S. Lawrence, "Supplement to Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques," Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, USA, Tech. Rep., 1984.