ADAPTIVE REACTIVE JOB-SHOP SCHEDULING WITH REINFORCEMENT LEARNING AGENTS

Thomas GABEL, **Martin RIEDMILLER**

Neuroinformatics Group
Institute of Cognitive Science
Department of Mathematics and Computer Science
University Osnabrück
49069 Osnabrück, Germany
E-mail: {thomas.gabel|martin.riedmiller}@uos.de

Abstract

Traditional approaches to solving job-shop scheduling problems assume full knowledge of the problem and search for a centralized solution for a single problem instance. Finding optimal solutions, however, requires an enormous computational effort, which becomes critical for large problem instance sizes and, in particular, in situations where frequent changes in the environment occur. In this article, we adopt an alternative view on production scheduling problems by modelling them as multi-agent reinforcement learning problems. In fact, we interpret job-shop scheduling problems as sequential decision processes and attach to each resource an adaptive agent that makes its job dispatching decisions independently of the other agents and improves its dispatching behavior by trial and error employing a reinforcement learning algorithm. The utilization of concurrently and independently learning agents requires special care in the design of the reinforcement learning algorithm to be applied. Therefore, we develop a novel multi-agent learning algorithm, that combines data-efficient batch-mode reinforcement learning, neural network-based value function approximation, and the use of an optimistic inter-agent coordination scheme. The evaluation of our learning framework focuses on numerous established Operations Research benchmark problems and shows that our approach can very well compete with alternative solution methods.

Keywords: reinforcement learning, multi-agent systems, job-shop scheduling, neural networks

To appear in: International Journal of Information Technology and Intelligent Computing.

1 Introduction

The basic idea behind reinforcement learning (RL) is to let (software) agents acquire a control policy on their own on the basis of trial and error by repeated interaction within their environment [33]. The empirical evaluation of reinforcement learning 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 aim at bridging the gap between focusing on artificial RL benchmark problems and real-world applications. We spotlight job-shop scheduling problems (JSSPs), a specific class of problems from the field of production scheduling, as an interesting type of benchmark problems that feature both the character of standardized, well-defined task descriptions as well as the property of representing application-oriented and extremely challenging problems.

In production scheduling, tasks have to be allocated to a limited number of resources in such a manner that one or more objectives are optimized. Though various classical approaches can be shown to provide optimal solutions to various scheduling problem variants, they typically do not scale with problem size, suffering from an exponential increase in computation time. In previous work [32, 13], we have explored a novel alternative approach to production scheduling that performs reactive scheduling and is capable of producing approximate solutions in minimal time. Here, each resource is equipped with a scheduling agent that makes the decision on which job to process next based solely on its local view on the plant. As each agent follows its own decision policy, thus rendering a central control unnecessary, this approach is particularly suitable for environments where unexpected events, such as the arrival of new tasks or machine breakdowns, may occur and, hence, frequent re-planning would be required.

We employ reinforcement learning to let the scheduling agents adapt their behavior policy, based on repeatedly collecting experience within their environment and on receiving positive or negative feedback (reinforcement signals) from that environment. After that *learning* phase, each agent will have obtained a purposive, reactive behavior for the respective environment. Then, during the *application* phase, e.g. during application in a real plant, each agent can make its scheduling decisions very quickly by utilizing its reactive behavior.

Reinforcement learning and job-shop scheduling depict the two central

concepts covered in this article. Accordingly, in Section 2 we start off by briefly introducing notation and reviewing some basics of RL and job-shop scheduling. Moreover, we discuss basic modelling alternatives for solving job-shop scheduling problems by means of using reinforcement learning, point to related work, and clarify similarities and differences between our approach to solving JSSPs and other techniques from the fields of Operations Research and Artificial Intelligence. Section 3 presents in detail our multi-agent reinforcement learning approach for performing reactive scheduling. In Section 4, we we focus on different advanced research questions that arise when aiming at the application of our learning framework for large-scale problems of current standards of difficulty. The experimental part of this article (Section 5) concentrates on established Operations Research benchmark problems for job-shop scheduling and contrasts the performance of our adaptive approach to several analytical and heuristic ones. Furthermore, we analyze the generalization capabilities of the learned dispatching policies, discuss the results, and prospect important topics for future work.

2 Foundations

This article is concerned with a number of different lines of research. Therefore, this section introduces the notation used subsequently and covers basics of reinforcement learning, job-shop scheduling, and multi-agent scheduling that are relevant in the scope of this article. Furthermore, relevant related work is highlighted.

2.1 Basics of Reinforcement Learning

One of the general aims of machine learning is to produce intelligent software systems, sometimes called agents, by a process of learning and evolving. Reinforcement learning represents one approach that may be employed to reach that goal. In an RL learning scenario the agent interacts with its initially unknown environment, observes the results of its actions, and adapts its behavior appropriately. To some extent, this imitates the way biological beings learn.

In each time step, an RL agent observes the environmental state and makes a decision for a specific action, which, on the one hand, may incur some immediate costs[1] (also called reinforcement) generated by the agent's environ-

[1]The notion of *rewards* is basically equal to the notion of costs we are employing. Costs correspond to negative rewards.

ment and, on the other hand, transfers the agent into some successor state. The agent's goal is not to minimize the immediate costs, but its long-term, expected costs. To do so it must learn a decision policy π that is used to determine the best action for a given state. Such a policy is a function that maps the current state $s \in S$ to an action a from a set of viable actions A.

The basic reinforcement learning paradigm is to learn the mapping $\pi : S \rightarrow A$ only on the basis of the reinforcement signals the agent gets from its environment. By repeatedly performing actions and observing corresponding costs, the agent tries to improve and fine-tune its policy. Research in RL has brought about a variety of algorithms that specify how experience from past interaction is used to adapt the policy. Assuming that a sufficient amount of states has been observed and costs/rewards have been received, the optimal decision policy will have been found and the agent following that policy will behave perfectly in the particular environment.

The standard approach to modelling reinforcement learning problems is to use Markov Decision Processes (MDP). An MDP is a 4-tuple (A, S, c, p) where S and A denote the state and action spaces, respectively, $p : S \times A \times S \rightarrow [0, 1]$ is a probabilistic state transition function with $p(s, a, s')$ describing the probability to end up in s' when taking action a in state s. Moreover, $c : S \times A \rightarrow \mathbb{R}$ is a cost function that denotes the immediate costs that arise when taking a specific action in some state. In search of an optimal behavior, the learning agent must differentiate between the value of possible successor states or the value of taking a specific action in a certain state. Typically, this kind of ranking is made by computing a state or state-action value function, $V : S \rightarrow \mathbb{R}$ or $Q : S \times A \rightarrow \mathbb{R}$, which bear information about the prospective value of states or state-action pairs, respectively. Having determined the optimal value function, i.e. the one that correctly reflects the expected costs to go for a state or state-action pair for the respective environment, this function can easily be employed to induce the best action in a given state, e.g. by evaluating $\arg\min_{b \in A} Q(s, b)$. For more basics and a thorough review of state-of-the-art reinforcement learning methods we refer to [33].

2.2 Basics of 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).

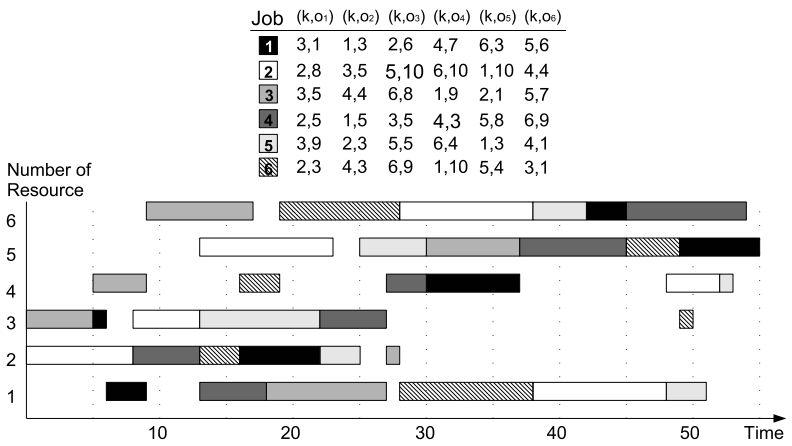

Figure 1. A simple job-shop scheduling problem instance (ft6) with 6 resources and 6 jobs and a Gantt chart representation of an optimal solution.

Figure 1 shows a 6×6 (6 resources and 6 jobs, $m = n = 6$) problem instance from [22]. 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 [4]). 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 ($C_{max} = 55$) whose makespan is 55. For further details on scheduling theory and its applications the reader is referred to [27].

2.3 Multi-Agent View on Job-Shop Scheduling Problems

Basically, there are two ways of modelling a JSSP as a Markov Decision Process (MDP, [29]). 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, [10]). 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 [9]. 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 re-planning.

- 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.

2.4 Related Work

Job-shop scheduling has received an enormous amount of attention in the research literature. As mentioned in Section 2.3, research in production scheduling traditionally distinguishes predictive and reactive solution approaches. Assuming complete knowledge about the entire scheduling problem to be solved (thus about all jobs, their operations and belonging durations as well as about the resources and which operations must be executed on which resource), and aiming at the achievement of global coherence in the process of job dispatching, Operations Research has brought about a variety of predictive scheduling algorithms that yield optimal solutions for individual problem instances – at least up to certain problem sizes, since the computational effort scales exponentially with problem size. By contrast, reactive scheduling approaches support decentralized, local decision making, which is beneficial when no centralized control can be instantiated (e.g. when a factory's resource does not know about the current workload at any other resource) or when quick responses to unexpected events are required. Most of the approaches that utilize ideas from research in Artificial Intelligence to solve scheduling problems belong to the realm of predictive scheduling.

Classical, predictive approaches to solving job-shop scheduling problems cover, for instance, disjunctive programming, branch-and-bound algorithms [2], or the shifting bottleneck heuristic [1]—a thorough overview is given in [28]. Moreover, there is a large number of local search procedures to solve job-shop scheduling problems. These include beam search [25], simulated annealing [34], tabu search [23], greedy randomized adaptive search procedures (GRASP, [8]), as well as squeaky wheel optimization [16]. Furthermore, various different search approaches have been suggested based on evolutionary techniques and genetic algorithms (e.g. [3] or [24]).

In contrast to these analytical methods yielding to search for a single prob-

lem's best solution, our RL-based approach belongs to the class of reactive scheduling techniques. Most relevant references for reactive scheduling cover simple as well as complex dispatching priority rules (see [26] or [7]). Focusing on job-shop scheduling with blocking and no-wait constraints, in [20] the authors develop heuristic dispatching rules (such as AMCC, cf. Section **??**) that are suitable for online control. but that benefit from having a global view onto the entire plant when making their dispatch decisions.

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 [38] 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 simplifying specialization of a scheduling problem. They show that the adaptive resources are able to effectively learn when they have to request maintenance [19], and that introducing a hierarchical decomposition of the learning task is beneficial for obtaining superior results [35]. Another repair-based approach relying on an intelligent computing algorithm is suggested by [37] who make use of case-based reasoning and a simplified reinforcement learning algorithm to achieve adaptation to changing optimization criteria.

3 A Multi-Agent Reinforcement Learning Approach to Reactive Production Scheduling

In this section, we propose an approach to production scheduling problems that allows to combine the desire for obtaining near-optimal solutions with the ability to perform reactive scheduling, realized by resource-coupled scheduling agents that make their dispatching decisions in real-time. The dispatching rules implemented by the agents are, however, not fixed, but are autonomously adapted instead by getting feedback of the overall dynamic behavior of the whole production plant.

3.1 System Architecture

As described in Section 2.3 we adopt a multi-agent perspective on performing job-shop scheduling. So, to each of the resources we attach an adaptive agent that is capable of improving its dispatching behavior with the help of

reinforcement learning. The agents' task is to decide which job to process next out of the set of jobs that are currently waiting for further processing at some resource. Accordingly, an agent cannot take an action at each discrete time step t, but only after its resource has finished one operation. Therefore, the agent makes its decisions after time intervals whose lengths Δt_k are determined by the durations of the operations processed.

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) 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 partial observability 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.

The feature vectors representing states $s_k \in S$ and actions/jobs $a_k \in A_k$, as generated by the resources' local view, 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. Moreover, it is advisable to define features that represent properties of typical problem classes instead of single problem instances, so that acquired knowledge is general and valid for similar problems as well. With respect to the desired real-time applicability of the system, the features should also be easy to compute, enabling a maximum degree of reactivity. State features depict the current situation of the resource by describing its processing state and the set A_k of jobs currently waiting at that resource. That job set characterization includes the resource's current workload, an estimation of the earliest possible job completion times, or the estimated makespan. Furthermore, we capture characteristics of A_k by forming relations between minimal and maximal values of certain job properties over the job set (like operation duration times or remaining job processing

times). Action features characterize single jobs a_k from A_k currently selectable by k. Here, we aim at describing makespan-oriented properties of individual jobs (like processing time indices), as well as immediate consequences to be expected when processing that job next, viz the properties of the job's remaining operations (e.g. the relative remaining processing time). Apart from that, action features cover the significance of the next operation $o_{j_i,next}$ of job j_i (e.g. its relative duration).

3.2 Details of the Learning Algorithm

When there is no explicit model of the environment and of the cost structure available, Q learning [36] is one of the RL methods of choice to learn a value function for the problem at hand, from which a control policy can be derived. The Q function $Q : S \times A \rightarrow \mathbb{R}$ 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

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

where α is the learning rate, γ the discount factor, and where the successor state $\overline{s}$ and the immediate costs $c(s,a,\overline{s})$ are generated by simulation or by interaction with a real process. For the case of finite state and action spaces where the Q function can be represented using a look-up table, there are convergence guarantees that say that Q learning converges to the optimal value function $Q^\star$, assumed that all state-action pairs are visited infinitely often and that α diminishes appropriately. Given convergence to $Q^\star$, the optimal policy $\pi^\star$ can be induced by greedy exploitation of Q according to $\pi^\star(s) = \arg\min_{a \in A(s)} Q^\star(s,a)$.

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):

$$\begin{aligned} Q_k(s_k(t), a_k(t)) \quad := \quad & C_{sa}(t, \Delta t_k) \\ & + \gamma \min_{b \in A_k(t+\Delta t_k)} Q_k(s_k(t+\Delta t_k), b) \end{aligned} \tag{3}$$

This learning rule establishes a relation between the local dispatching decisions and the overall optimization goal, since the global immediate costs are taken into consideration (e.g. costs caused due to tardy jobs). Since a resource is not allowed to take actions at each discrete time step[2], C_{sa} collects the immediate

[2] After having started operation o_{j_i} the resource remains busy until that operation is finished.

global costs arising between t and the next decision time point $t+\Delta t_k$ according to

$$C_{sa}(t, \Delta t_k) := \sum_{i=t}^{t+\Delta t_k} C(s, a, i). \tag{4}$$

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

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

As indicated in the Introduction, 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, 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 clear 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, 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. The focal point of our research, however, is not to concentrate just on individual problem instances, but on arbitrary ones. Hence, we need to assume the domain of Q to be infinite or even continuous, and will have to employ a function approximation mechanisms to represent it.

A crucial precondition for our adaptive agent-based approach to learning to make sophisticated scheduling decisions is that the global direct costs (as feedback to the learners) coincide with the overall objective of scheduling. We define the global costs C to be the sum of the costs that are associated with the resources (sum over k) and jobs (sum over i):

$$C(s, a, t) := \sum_{k=1}^{m} u_k(s, a, t) + \sum_{i=1}^{n} r_{j_i}(s, a, t) \tag{6}$$

When focusing on minimizing overall tardiness, it is possible to set $u_k \equiv 0$ and to let r_{j_i} capture the tardiness $T_{j_i} = max(0, c_{j_i} - d_{j_i})$ of the jobs by

$$r_{j_i}(s, a, t) := \begin{cases} T_{j_i}, & \text{if } j_i \text{ is being finished at } t \\ 0, & \text{else} \end{cases} \tag{7}$$

A disadvantage of that formulation is that the cost function does not reflect when the tardiness actually occurs. Since that information may help the learning algorithm, we prefer the following, equivalent formulation, which assigns costs at each time step during processing:

$$r_{j_i}(s,a,t) := \begin{cases} 1\,, & \text{if } j_i \text{ is tardy at } t \\ 0\,, & \text{else} \end{cases} \tag{8}$$

Equations 7 and 8 are no longer useful when the overall objective is to minimize the makespan C_{max} of the resulting schedule. Accordingly, information about tardy jobs or finishing times c_{j_i} of individual jobs provide no meaningful indicator relating to the makespan. However, the makespan of the 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 [27], i.e. the minimal makespan of a non-delay schedule[3] is achieved when the number of time steps can be minimized during which jobs are waiting for processing at the resources' queues. This argument gives rise to setting $r_{j_i} \equiv 0$ and to defining

$$u_k(s,a,t) := |\{ j_i \mid j_i \text{ queued at } k\}| \tag{9}$$

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.

3.3 Value Function Approximation with Neural Networks

Since an agent's value function Q_k has an infinite domain, we need to employ some function approximation technique to represent it. In this work, we use multilayer 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 $f : D \to \mathbb{R}$ that is continuous on a bounded set D [15]. 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. Input to a neural net are the features (cf. Section 3.1) describing the situation of the resource as well as single waiting jobs[4]. Thus, the neural

[3] Concerning the discussion of considering non-delay vs. delay schedules we refer to Section 5.5.

[4] In the experiments whose results we describe in Section 5, we made use of seven state features and six action features, hence having 13 inputs to the neural network.

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 (see Figure 2).

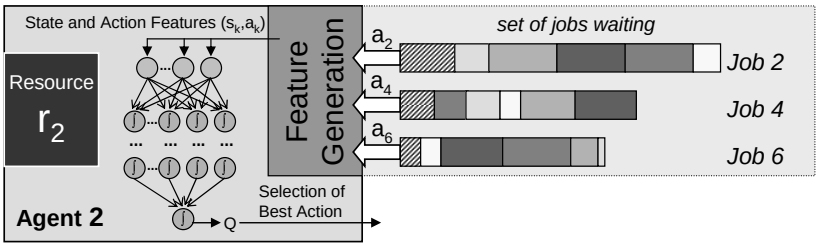

Figure 2. Representing the state-action value function with a neural network whose input are state and action features describing the resource's current situation. The first operation of each of the jobs 2, 4, and 6 has to be processed on resource r_2.

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 [14] 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 value function-based 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 article, we refer to relevant literature [5, 6, 21].

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 [5]): 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 (see Section 4.1). 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.

4 Fitted Q Iteration with Neural Networks and Optimistic Assumption

In Section 3, we have outlined the general characteristics and several design decision of our learning framework. The important issue of how to update the agents' state-action value functions Q_k, however, has been touched only briefly: Equation 4 provides an adaptation of the general Q learning update rule to the type of learning problems we are considering, by means of which data-inefficient online Q learning without an attempt to enforce coordination between multiple agents may be realized. In the following, however, we address the problems of utmost efficient training data utilization and adequate inter-agent coordination, which are of fundamental importance for obtaining learning results of high quality.

4.1 Training Data Utilization

In Section 3.2 we have pointed to the distinction between the learning and the application phase of our adaptive scheduling approach. During the learning phase, 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, i.e. to not greedily exploit their Q_k, 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, in the light of problem dimensions that are considerable from a reinforcement learning perspective, it is inevitable to foster fast improvements of the learned policy by exploiting the training data as efficiently as possible. For this purpose, we revert to neural *fitted Q iteration*.

Fitted Q iteration denotes a batch (also termed off-line) reinforcement learning framework, in which an approximation of the optimal policy is computed from a finite set of four-tuples [12]. 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:

1. increment q
2. 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 $out^i = c^i + \gamma \min_{b \in A(s^i)} \tilde{Q}^{q-1}(\overline{s}^i, b)$

3. use the regression algorithm and the training set $\mathbb{F}$ to induce an approximation $\tilde{Q}^q : S \times A \to \mathbb{R}$

Subsequently, we consider neural fitted Q iteration (NFQ, [30]), 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 (step 3). NFQ is an effective and efficient RL 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 discuss an adaptation of NFQ to be used in the scope of this work in the next section.

4.2 Inter-Agent Coordination

In the literature on multi-agent learning, a distinction between joint-action learners and independent learners is made [11]. The former can observe their own, as well as the other agents' action choices. Consequently, in that case the multi-agent MDP can be reverted to a single-agent MDP with an extended action set and be solved by some standard method. Here, however, we concentrate on independent learners because of the following reasons:

1. We want to 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.

2. Decision-making shall take place in a distributed, reactive manner. Hence, no agent will be aware of the jobs being processed next on other resources.

3. The consideration of joint-action learners with global view on the plant would take us nearer to giving all agents the ability to, e.g., construct a disjunctive graph for the scheduling problem at hand and use some classical solution method to solve it. With respect to 1) and 2), the intended application of learned scheduling policies to unknown situations and in presence of unexpected events, this is exactly what we intend to avoid.

We are, of course, aware that the restrictions that we impose on our learning agents depict a significant problem aggravation when compared to the task of finding an optimal schedule with some analytical algorithm and full problem knowledge.

Given the fact that the scheduling benchmarks to which we intend to apply our learning framework 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 [17] 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')

$$Q_k(s,a) := \max\{Q_k(s,a), r(s,a) + \gamma \max_{b \in A(s')} Q_k(s',b)\} \tag{10}$$

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 [17].

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 9). 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 3, 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 during the learning phase, 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 $\mathbb{T}_k = \{(s^i, a^i, c^i, \vec{s}^{\,i}) | i = 1, \ldots, p\}$

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
   $\mathbb{O} = \{(in^j, out^j) | j = 1, \ldots, p'\}$ 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)
   $$out^j := \min_{\substack{(s^i, a^i, c^i, \vec{s}^{\,i}) \in \mathbb{T}_k, \\ (s^i, a^i) = in^j}} \left(c_i + \gamma \min_{b \in A(s')} Q_k^{(q)}(s_i', b) \right) \qquad (11)$$

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

Figure 3. 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 [31] we use). Apart from those net training issues, the pseudo-code of OA-NFQ in Figure 3 reflects also the policy screening technique (cf. Section 3.3): 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. [12] 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 5.

5 Empirical Evaluation

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 our approach, utilizing the algorithms we have described in Section 4, to let the agents acquire high-quality dispatching policies for problem instances of current standards of difficulty. Furthermore, we want to investigate, whether the learned policies generalize to other, similar benchmark problems, too.

Benchmark problems abz5-9 were generated by Adams et al. [1], problems orb01-09 were generated by Applegate and Cook [2], and finally, problems la01-20 are due to Lawrence [18]. Although these benchmarks are of different sizes, they have in common that no recirculation occurs and that each job has to be processed on each resource exactly once ($\nu_{j_i} = m, i \in \{1, \ldots, n\}$).

5.1 Experiment Overview

Within this evaluation, we compare four different types of algorithms to solve scheduling problems, each of them being subject to different restrictions and following different paradigms in generating schedules.

1. **Analytical Scheduling Algorithms** that perform predictive scheduling and interpret a given JSSP as a single MDP of which they have full knowledge. They find the solution for one specific problem instance, being unable to generalize. Since our focus is not on predictive scheduling, we do not consider individual example algorithms of this group. Instead we let this group be represented by a JSSP's optimal solution

(minimal makespan) that may in principle be found by various analytical scheduling algorithms, when given infinite computational resources.

2. **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 [20]).

3. **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).

4. Our approach to adaptive reactive job-shop scheduling with **Reinforcement Learning Agents**.

Regarding the restrictions our approach is subject to (MMDP interpretation of a JSSP with local view) a comparison to group 3) is most self-evident. However, by approaching or even surpassing the performance of algorithms from group 2) or by reaching the theoretical optimum, we can make a case for the power of our approach.

5.2 Example Benchmark

To start with, we consider the notorious problem ft10 proposed by Fisher and Thompson [22], that had remained unsolved for more than twenty years. Here, during the learning phase, $\mathcal{S}_L = \{p_{ft10}\}$ is processed 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 neural fitted Q iteration with optimistic assumption and 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).

We compare the performance of eight different scheduling algorithms

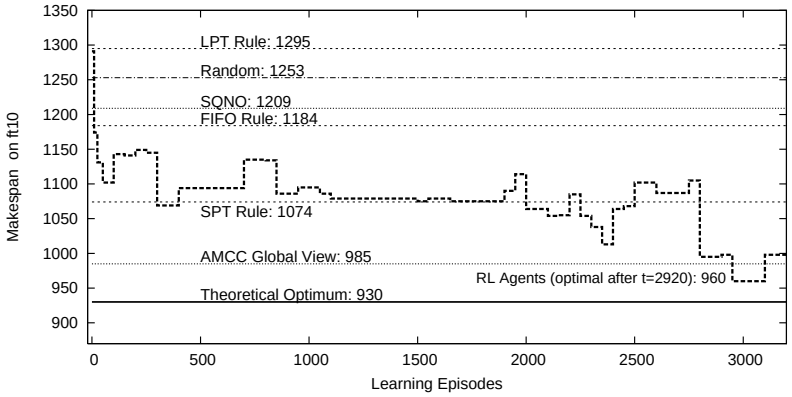

Figure 4. Learning process for the notorious ft10 problem.

- a purely random dispatcher
- three basic local view dispatching rules (LPT, SPT, and FIFO)
- two more sophisticated global view dispatching rules (SQNO and AMCC)
- our adaptive agent-based approach to reactive scheduling
- the theoretical optimum ($C_{max,opt} = 930$)

Figure 4 visualizes the learning progress for the ft10 instance. The best solution found by the learning approach was discovered after 2920 repeated processings of $\mathcal{S}_L$ (see Figure 4). 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 learnt dispatching policy (represented by the agents' neural networks representing their Q functions) by means of the policy screening method described in Section 3.3.

5.3 Benchmark Results

Next, we studied the effectiveness of our agent-based scheduling approach for a large number of different-sized benchmark problems, ranging from job-shops with 5 resources and 10 jobs to 15 resources and 20 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 NFQ with optimistic assumption ($N = 20$ iterations of the Q iteration loop) in order to reach the vicinity of a near-optimal Q function as quickly as possible.

Name & Size	1) Simple DPRs			2) Cmplx.DPRs		3) Central.Methods		4) Adaptive	Remain.
	FIFO	LPT	SPT	SQNO	AMCC	GRASP	Optimum	RL Agents	Err.(%)
ft6 6×6	65	77	88	73	55	55	*55*	**57**	3.64
ft10 10×10	1184	1295	1074	1209	985	938	*930*	**960**	3.23
ft20 5×20	1645	1631	1267	1476	1338	1169	*1165*	**1235**	6.01
abz5 10×10	1467	1586	1352	1397	1318	1238	*1234*	**1293**	4.78
abz6 10×10	1045	1207	1097	1124	985	947	*943*	**981**	4.03
abz7 15×20	803	903	849	823	753	723	*667*	**723**	8.40
abz8 15×20	877	949	929	842	783	729	*670*	**741**	10.60
abz9 15×20	946	976	887	865	777	758	*691*	**779**	12.74
Avg. *abz*	1033.6	1124.2	1022.8	1010.2	923.2	879.0	*841.0*	**903.4**	8.11
la01 5×10	772	822	751	988	666	666	*666*	**666**	0.00
la02 5×10	830	990	821	841	694	655	*655*	**687**	4.89
la03 5×10	755	825	672	770	735	604	*597*	**648**	8.54
la04 5×10	695	818	711	668	679	590	*590*	**611**	3.56
la05 5×10	610	693	610	671	593	593	*593*	**593**	0.00
Avg. $la_{5\times10}$	732.4	829.6	713.0	787.6	673.4	621.6	*620.2*	**641.0**	3.40
la06 5×15	926	1125	1200	1097	926	926	*926*	**926**	0.00
la07 5×15	1088	1069	1034	1052	984	890	*890*	**890**	0.00
la08 5×15	980	1035	942	1058	873	863	*863*	**863**	0.00
la09 5×15	1018	1183	1045	1069	986	951	*951*	**951**	0.00
la10 5×15	1006	1132	1049	1051	1009	958	*958*	**958**	0.00
Avg. $la_{5\times15}$	1003.6	1108.8	1054.0	1065.4	955.6	917.6	*917.6*	**917.6**	0.00
la11 5×20	1272	1467	1473	1515	1239	1222	*1222*	**1222**	0.00
la12 5×20	1039	1240	1203	1202	1039	1039	*1039*	**1039**	0.00
la13 5×20	1199	1230	1275	1314	1161	1150	*1150*	**1150**	0.00
la14 5×20	1292	1434	1427	1438	1305	1292	*1292*	**1292**	0.00
la15 5×20	1587	1612	1339	1400	1369	1207	*1207*	**1207**	0.00
Avg. $la_{5\times20}$	1277.8	1396.6	1343.4	1373.8	1222.6	1182.0	*1182*	**1182.0**	0.00
la16 10×10	1180	1229	1156	1208	979	946	*945*	**996**	5.40
la17 10×10	943	1082	924	955	800	784	*784*	**793**	1.15
la18 10×10	1049	1114	981	1111	916	848	*848*	**890**	4.95
la19 10×10	983	1062	940	1069	846	842	*842*	**875**	3.92
la20 10×10	1272	1272	1000	1230	930	907	*902*	**941**	4.32
Avg. $la_{10\times10}$	1085.4	1151.8	1000.2	1114.6	894.2	865.4	*864.2*	**899.0**	3.95
orb1 10×10	1368	1410	1478	1355	1213	1070	*1059*	**1154**	8.97
orb2 10×10	1007	1293	1175	1038	924	889	*888*	**931**	4.84
orb3 10×10	1405	1430	1179	1378	1113	1021	*1005*	**1095**	8.96
orb4 10×10	1325	1415	1236	1362	1108	1031	*1005*	**1068**	6.27
orb5 10×10	1155	1099	1152	1122	924	891	*887*	**976**	10.03
orb6 10×10	1330	1474	1190	1292	1107	1013	*1010*	**1064**	5.35
orb7 10×10	475	470	504	473	440	397	*397*	**424**	6.80
orb8 10×10	1225	1176	1107	1092	950	909	*899*	**956**	6.34
orb9 10×10	1189	1286	1262	1358	1015	945	*934*	**996**	6.64
Avg. *orb*	1164.3	1226.1	1142.6	1163.3	977.1	907.3	*898.2*	**962.7**	7.13
Overall Avg.	1054.2	1137.6	1037.3	1080.7	932.9	882.6	*874.6*	**908.9**	4.17

Table 1. Learning results on OR job-shop benchmark problems.

In Table 1, we compare the capabilities of four different groups of algorithms to the theoretical optimum. Simple dispatching priority rules (group 1) consider only the local situation at the resource for which they make a dispatching decision. The same holds for our adaptive agents approach (4) whose results are given in the table's last two columns. Moreover, two examples of more sophisticated heuristic rules are considered (group 2) that are not subject to that local view restriction.

Group 3 comprises centralized methods. Here, an instance of a metaheuristic as well as the best known solution (which for the considered benchmarks coincides with the optimal solution), as it may be found by a predictive scheduling algorithm like a branch-and-bound or disjunctive programming method, are provided. Of course, there exists a large variety of centralized methods such as heuristic search procedures or evolutionary approaches to tackle JSSPs. All those algorithms work under superior preconditions compared to local dispatchers because they have full problem knowledge of the task. Accordingly, a comparison of centralized methods' results to the results of our adaptive agent-based approach is not very meaningful. For the sake of completeness, however, we have included the performance of a single representative (GRASP, [8]) of those methods in Table 1. Note that the "Remaining Error" of our learning approach is also calculated with respect to the theoretical optimum.

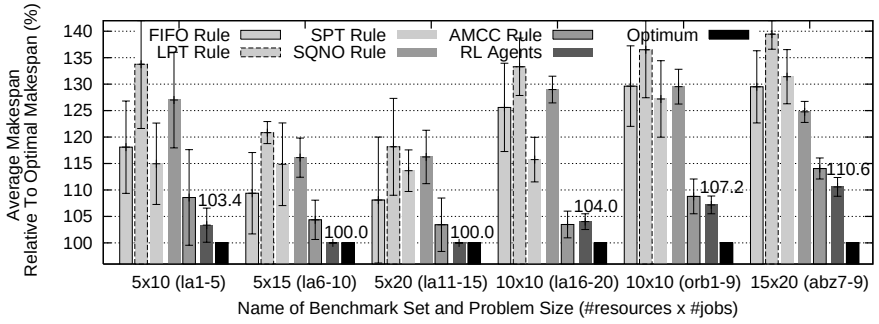

Figure 5. For different sets of benchmark problems with equal size, this figure visualizes the average performance of different approaches in terms aiming at a minimized makespan of the resulting schedules. The results are given relative to the makespan of the optimal schedule (100%, black data series). Data series colored in light gray correspond to static rules having local state information only, whereas medium gray-colored ones are not subject to that restriction. The data series termed "Adaptive" corresponds to the learning approach we have suggested (performance reported belongs to the makespan achieved when *training* for single benchmark problems), which is restricted to the local view, too.

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), 10×10 (la16-20, orb1-9) sets, only a small relative error of less than ten percent compared to the optimal makespan remains (3.4/4.0/7.1%). 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 mixed abz benchmarks involving also instances with 15 resources and 20 jobs per problem, the average relative error increases to 8.1%, yet the rule-based schedulers are surpassed, still. For a better illustration of the main findings we also have grouped the results on individual benchmark problems into classes with respect to the numbers of resources and jobs to be processed (Figure 5).

5.4 Generalization to Unknown Problems

Some analytical search procedure (like a tabu search) finds a suitable schedule for one specific problem instance. By contrast, our learning approach – after having learned for a set of one or more training problems – will have yielded dispatching policies that are generally applicable. To empirically investigate the generalization capabilities of the learned dispatching policies, we designed a further 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.

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%
RL Agents (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 2. Generalization Capabilities: During its application phase, the learned dispatching policies are used for problems not covered during training. Average makespan and remaining errors relative to the optimum are provided.

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 2 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 our adaptive RL agents correspond to their performance on independent test problem instances, i.e. to scheduling scenarios that were not included in the respective training sets $\mathcal{S}_L$ during cross-validation. From that numbers it is obvious that all static local view dispatchers, to which the results of our approach 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 our adaptive RL agents, 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 our learning approach which is logical for two reasons. First, AMCC works under superior conditions compared to our learning approach as it is allowed to make use of global state information (information about the situation of other resources). Second, when training our RL agents for single 10×10 problems (see Table 1), the resultant average performance ($C_{max}^{avg,orb} = 962.3$) was only slightly better than the performance of the AMCC rule. Consequently, it is logical to expect that the AMCC rule outperforms our agents when their learned dispatching policies are applied to problems, they have never seen before.

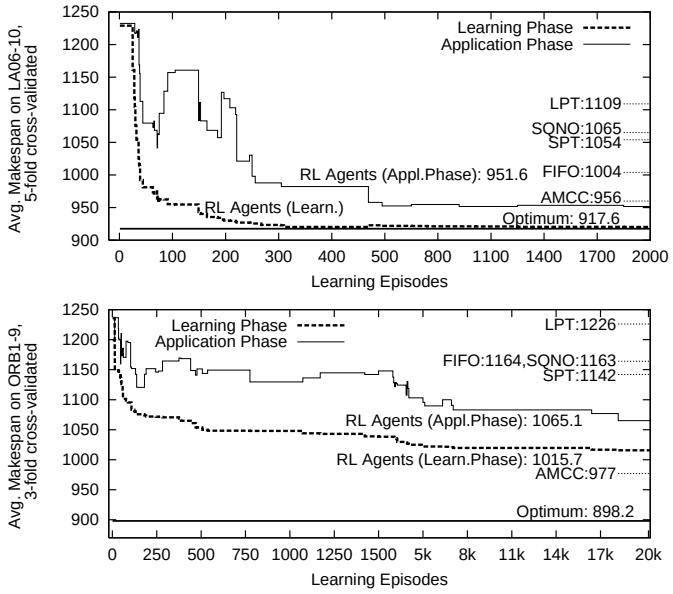

Figure 6. Using ν-fold cross-validation, the adaptive agents' dispatching policies are trained on the $\mathcal{S}_{la}^{5\times15}$ (top) and $\mathcal{S}_{orb}^{10\times10}$ (bottom) benchmark suites, respectively, and, during the application phase, are evaluated on independent test scenarios. At the chart's right hand side, the average makespans achieved by several static dispatching rules are given for comparison.

Figure 6 illustrates the corresponding learning progress over time (up to 20000 training episodes) for this experiment. For the 5×15 problems $\mathcal{S}_{la}^{5\times15}$, our learning approach succeeds in entirely capturing the characteristics of the training problems in $\mathcal{S}_L$ during training: When a plant utilizing the dispatching policies learned processes the problems from $\mathcal{S}_L$, the theoretic optimum is almost reached, i.e. schedule decisions resulting in minimal makespan are yielded. More importantly, even on the independent problem instances from $\mathcal{S}_A$ that were not experienced during training, excellent results are achieved: With an average makespan of 951.6 during the application phase, the acquired dispatching policies outperform not just dispatching rules that work under the same basic conditions as our learning agents do, but even those that have full state information (like AMCC). Furthermore, the gap in performance compared to the theoretically best schedules is only 3.7% in terms of average C_{max}.

The local dispatching rules obtained for the $\mathcal{S}_{orb}^{10\times10}$ benchmark suite feature a remaining relative error of 18.6% compared to the theoretic optimum in terms of minimal makespan. Although for these more intricate benchmark

problems the results are less impressive, they allow us to draw two empiric conclusions: First, traditional dispatching priority rules that solely employ local state information at the regarding the respective resource (just as our learning approach does) are clearly outperformed. And, second, the resulting dispatching policies acquired during training feature generalization capabilities and, hence, can effectively be applied to similar, yet unknown, scheduling problem instances.

5.5 Discussion

Our approach to model the scheduling task as a sequential decision problem and to make reactive scheduling decisions features the disadvantage that currently the resulting schedules correspond to solutions from the set of non-delay schedules, only: 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. Our approach does not allow a resource to remain idle, if there is more work to be done.

From scheduling theory, however, it is well-known that for certain scheduling problem instances the optimal schedule may very well be a delay schedule. In fact, the following subset inclusion holds for three sub-classes of non-preemptive schedules

$$\mathbb{S}_{nondelay} \subsetneq \mathbb{S}_{active} \subsetneq \mathbb{S}_{semiactive} \subsetneq \mathbb{S} \tag{12}$$

where $\mathbb{S}$ denotes the set of all possible schedules [27]. The optimal schedule for a particular problem, however, is always within $\mathbb{S}_{active}$, but not necessarily within $\mathbb{S}_{nondelay}$.

We expect that, in future work, we will be able to further boost the performance of our learning approach. Currently, our adaptive agents can generate schedules of the class $\mathbb{S}_n$ of *non-delay* schedules exclusively: As a consequence, our approach 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 5). Yet, an extension of our learning framework towards delay schedules depicts an important and promising issue for future work.

6 Conclusion

Job-shop problems are NP-hard. We have pursued an alternative approach to scheduling where each resource is assigned a decision-making agent that

decides which job to process next, based on its partial view on the production plant. We use neural reinforcement learning to enable the agents to learn a dispatching policy from repeated interaction with the plant and to adapt their behavior to the environment. This way, we obtain a reactive scheduling system, where the final schedule is not calculated beforehand, viz before execution time, where online dispatching decisions are made, and where the local dispatching policies are aligned with the global optimization goal. So, not just the adaptation of the agents' behavior during learning is decentralized, but also decision-making during application proceeds without a centralized control.

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 learning 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 the multi-agent reinforcement learning framework proposed, 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 Operations Research benchmarks. Our experiments for such large-scale benchmark problems lets us come up with the conclusion that problems of current standards of difficulty can very well be effectively solved by the learning method we suggest: The dispatching policies our learning agents acquire clearly surpass traditional dispatching rules and, in some cases, are able to reach the theoretically optimal solution. Notwithstanding the inherent difficulties in facing partial state observability and agent-independent learning, the dispatching policies acquired do also generalize to unknown situations without retraining, i.e. they are adequate for similar scheduling problems not covered during the learning phase.

References

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

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

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

[4] Beasley J., 2005, *OR-Library*, `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`.

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

[6] Bertsekas D., Tsitsiklis J., 1996, *Neuro Dynamic Programming*, Athena Scientific, Belmont, USA.

[7] Bhaskaran K., Pinedo M., *Dispatching*, 1977, In: Salvendy G. (Ed.), Handbook of Industrial Engineering, John Wiley, New York, USA, pp. 2184–2198.

[8] Binato S., Hery W., Loewenstern D., Resende M., *A GRASP for Job Shop Scheduling*, 2001, In: Hansen P., Ribeiro C. (Ed.), Essays and Surveys in Metaheuristics, Kluwer Academic Publishers, New York, USA, pp. 177–293.

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

[10] Boutilier C., 1999, *Sequential Optimality and Coordination in Multiagent Systems*, Proceedings of IJCAI'99, Stockholm, Sweden, pp. 478–485.

[11] Claus C., Boutilier C., 1998, *The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems*, Proceedings of AAAI'98, Menlo Park, USA, pp. 746–752.

[12] Ernst D., Geurts P., Wehenkel L., 2005, *Tree-Based Batch Mode Reinforcement Learning*, Journal of Machine Learning Research, Vol. 6, pp. 504–556.

[13] Gabel T., Riedmiller M., 2006, *Reducing Policy Degradation in Neuro-Dynamic Programming*, Proceedings of ESANN'06, Bruges, Belgium, pp. 653–658.

[14] Gordon G., *Stable Function Approximation in Dynamic Programming*, 1995, Proceedings of ICML'95, San Francisco, USA, pp. 261–268.

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

[16] Joslin D., Clements D., 1999, *Squeaky Wheel Optimization*, Journal of Artificial Intelligence Research, Vol. 10, pp. 353–373.

[17] Lauer M., Riedmiller M., 2000, *An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems*, Proceedings of ICML'00, Stanford, USA, pp. 535–542.

[18] Lawrence S., 1984, *Supplement to Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques*, Techn. Report, Carnegie Mellon University, Pittsburgh, USA.

[19] Mahadevan S., Marchalleck N., Das T., Gosavi A., 1997, *Self-Improving Factory Simulation Using Continuous-Time Average-Reward Reinforcement Learning*, Proceedings of ICML'97, Nashville, USA, pp. 202–210.

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

[21] Munos R., 2003, *Error Bounds for Approximate Policy Iteration*, Proceedings of ICML'03, Washington, USA, pp. 560–567.

[22] Muth J. and Thompson G. (Ed.), 1963, *Industrial Scheduling*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

[23] Nowicki E., Smutnicki C., 1996, *A Fast Taboo Search Algorithm for the Job Shop Problem*, Management Science, Vol. 42, pp. 797–813.

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

[25] Ow P., Morton T., 1988, *Filtered Beam Search in Scheduling*, International Journal of Production Research, Vol. 26, pp. 297–307.

[26] Panwalkar S., Iskander W., 1977, *A Survey of Scheduling Rules*, Operations Research, Vol. 25, pp. 45–61.

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

[28] Pinson E., *The Job Shop Scheduling Problem: A Concise Survey and Some Recent Developments*, 1995, In: Chretienne P., Coffman E., Lenstra J. (Ed.), *Scheduling Theory and Applications*, pp. 177–293.

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

[30] Riedmiller M., 2005, *Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method*, Machine Learning: ECML 2005, Porto, Portugal, pp. 317–328.

[31] Riedmiller M., Braun H., 1993, *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm*, Proceedings of ICNN'93, San Francisco, USA, pp. 586–591.

[32] Riedmiller S., Riedmiller M., 1999, *A Neural Reinforcement Learning Approach to Learn Local Dispatching Policies in Production Scheduling*, Proceedings of IJCAI'99, Stockholm, Sweden, pp. 764–771.

[33] Sutton R., Barto A., 1998, *Reinforcement Learning. An Introduction*, MIT Press/A Bradford Book, Cambridge, USA.

[34] van Laarhoven P., Aarts E., Lenstra J., 1992, *Job Shop Scheduling by Simulated Annealing*, Operations Research, Vol. 40, pp. 113–125.

[35] Wang G., Mahadevan S., 1999, *Hierarchical Optimization of Policy-Coupled Semi-Markov Decision Processes*, Proceedings of ICML'99, San Francisco, USA, pp. 464–473.

[36] Watkins C., Dayan P., 1992, *Technical Note Q-Learning*, Machine Learning, Vol. 8, pp. 279–292.

[37] Zeng D., Sycara K., 1995, *Using Case-Based Reasoning as a Reinforcement Learning Framework for Optimization with Changing Criteria*, Proceedings of ICTAI'95, Washington, USA, pp. 56–62.

[38] Zhang W., Dietterich T., 1995, *A Reinforcement Learning Approach to Job-Shop Scheduling*, Proceedings of IJCAI'95, Montreal, Canada, pp. 1114–1120.