STOCHASTIC APPROXIMATE SCHEDULING BY NEURODYNAMIC LEARNING

Balázs Csanád Csáji [*] **László Monostori** [*,**]

[*] *Computer and Automation Research Institute, Hungarian Academy of Sciences*
[**] *Faculty of Mechanical Engineering, Budapest University of Technology and Economics*
{*csaji, monostor*}*@sztaki.hu*

Abstract: The paper suggests a stochastic approximate solution to scheduling problems with unrelated parallel machines. The presented method is based on neurodynamic programming (reinforcement learning and feed-forward artificial neural networks). For various scheduling environments (static-dynamic, deterministic-stochastic) different variants of episodic Q-learning rules are proposed. A way to improve the avoidance of local minima is also discussed. Some investigations on the exploration strategy, function approximation and parallelizing the solution are made. Finally, a few experimental results are shown.

Keywords: scheduling algorithms, machine learning, manufacturing systems, agents, Markov decision processes, neural networks, stochastic approximation

1. INTRODUCTION

Informatics of present time often has to face severe difficulties which arise from *incomplete* and *uncertain* information, in addition, the environment, in which the applications have to work in, can *change dynamically*, it can be *non-stationary*. Moreover, *complexity* problems also have to be faced, viz. even by static, highly simplified and abstract problems, the available computation power can be not enough to achieve an optimal solution in practice (e.g. in the case of NP-hard problems).

One way to overcome these difficulties is to apply *machine learning* techniques. It means designing systems which can adapt their behavior to the current state of the environment, can extrapolate their knowledge to unknown, unexperienced cases and can learn how to find and optimize the solutions of the problems that they have to deal with.

Production control is an important manufacturing problem, which has all the difficulties that were mentioned so far. One of the key problems in production control is the allocation of resources over time, namely *scheduling*. The paper suggests applying adaptive algorithms, such as reinforcement learning, artificial neural networks and simulated annealing, to give a stochastic approximate iterative solution for a generalized scheduling problem. Most of the presented ideas can be also applied to other kinds of resource allocation problems.

The structure of the paper is as follows: first, a static deterministic scheduling problem is defined and its complexity is highlighted. Next, a rooted directed graph representation is shown. Then, a brief introduction to reinforcement learning is given and variants of episodic Q-learning to different scheduling environments are proposed with a method that helps to avoid getting stucked in local minimum places. After that, the exploration strategy with simulated annealing is analyzed. Remarks on function approximation and on the parallelization of the algorithm are also presented.

2. THE PROBLEM OF SCHEDULING

Now, a static deterministic scheduling problem with unrelated parallel machines is considered: an instance of the problem consists of a finite set of *tasks* $V = \{v_1, v_2, \dots v_n\}$ together with a directed acyclic graph $G = \langle V, E \rangle$, where $E \subseteq V \times V$ represents the *precedence constraints* (a partial ordering) between the tasks. A finite set of *machines* M is also given with a partial function that defines the *processing times* of the tasks on the machines, $p : V \times M \to \mathbb{R}^+$, where $\mathbb{R}^+$ denotes the set of strictly positive real numbers. The tasks are supposed to be non-preemptive (they may not be interrupted) thus a *schedule* s can be defined as an ordered pair $s = \langle \tau, \mu \rangle$ where $\tau : V \to \mathbb{R}_0^+$ gives the starting time of the tasks and $\mu : V \to M$ defines that which machine will process which task. A schedule s is called *feasible* if and only if the following three properties are satisfied:

(s1) All machines process at most one operation at a time: $\neg\exists(m \in M \wedge v, v' \in V) : \mu(v) = \mu(v') = m \wedge \tau(v) \leq \tau(v') < \tau(v) + p(v, m)$;

(s2) Every machine can process the tasks which were assigned to it: $\forall v \in V : \langle v, \mu(v) \rangle \in D_p$;

(s3) The precedence constraints of the tasks are kept: $\forall \langle v, v' \rangle \in E : \tau(v) + p(v, \mu(v)) \leq \tau(v')$;

If an *earliest start time* ($e : V \to \mathbb{R}_0^+$) and a *due date* ($d : V \to \mathbb{R}_0^+$) for each task is also given, then the feasibility requirements can be extended by:

(s4) Each task must be processed between its release and due date: $\forall v \in V : \tau(v) \geq e(v) \wedge \tau(v) + p(v, \mu(v)) \leq d(v)$;

The set of all feasible schedules is denoted by S, which is supposed to be non-empty (thus, e.g. $\forall v \in V : \exists m \in M : \langle v, m \rangle \in D_p$). The objective of scheduling is to produce a schedule that minimizes (or maximizes) a *performance measure* $\kappa : S \to \mathbb{R}$, which is usually depends on the task completion times, only. For example, if the completion time of the task $v \in V$ (according to the schedule $s \in S$) is denoted by $C_v(s) = \tau(v) + p(v, \mu(v))$ then a commonly used performance measure can be defined by $C_{max}(s) = \max\{C_v(s) \mid v \in V\}$, which is often called total production time or make-span.

Naturally, not *any* function is allowed as a performance measure. The performance measures are restricted to those functions which have the property that the schedule can be uniquely generated from the order in which the jobs are processed through the machines, e.g., by semi-active *timetabling*. For example, *regular* measures, which are monotonic in completion times, have this property. Note that all of the usually used performance measures are regular. In this way, the scheduling problem becomes a *combinatorial optimization* problem determined by $\langle V, E, M, p, \kappa \rangle$.

It is easy to see that the presented parallel machine scheduling problem is a generalization of the standard *job-shop* scheduling problem, which is known to be strongly NP-hard (Lawler *et al.*, 1993), therefore this problem is also strongly NP-hard. Thus, unless $P = NP$, no polynomial time optimal algorithm exists. Moreover, if the used performance measure is C_{max}, there is no good polynomial time approximation of the optimal scheduling algorithm (Williamson *et al.*, 1997).

The presented problem is both *static* and *deterministic*. However, it may be argued that most practical scheduling problems are both *dynamic* and *stochastic*. The stochastic variant of the problem arises, when the processing times are given by independent random variables. In stochastic scheduling there are some information that will only be available during the execution of the plan. According to the usage of these data, two basic types of scheduling techniques are considered.

A *static* (*off-line*) scheduler has to make all decisions before the schedule actually being executed and it cannot take the actual evolution of the process into account. It has to build a schedule that can be executed with high probability.

For a *dynamic* (*on-line*) scheduler it is allowed to make the decisions as the scheduling process actually evolves and more information becomes available. The paper focuses on dynamic scheduling techniques. Note that a dynamic solution is not a simple $\langle \tau, \mu \rangle$ pair, but instead a scheduling *policy* (defined later) which controls the production.

First, the static and deterministic problem will be investigated, however later different learning rules for the dynamic and stochastic cases will also be suggested. Experimental results for the dynamic, on-line scheduling case will be presented, as well.

3. GRAPH REPRESENTATION

Many combinatorial optimization problems can be represented as the application of some operators to an initial element. In this case S is extended with *partial objects*, such as the empty object. The set of complete objects, in which the optimum is searched, is denoted by $S_c \subseteq S$. For the investigated scheduling problem, the initial element is the empty schedule and applying an operator to an object means that a task is scheduled to one of the machines. Formally, this representation contains an initial element $s_0 \in S$, a set of possible operators $A = \{a : D_a \to S \mid D_a \subseteq S\}$ and a function $\mathcal{A} : S \to \mathcal{P}(A)$ which assigns to each object a set of operators that can be applied to it. It follows that these problems have a rooted directed graph representation: the nodes are the elements of S, the root is s_0 and the edges are labeled with A. For

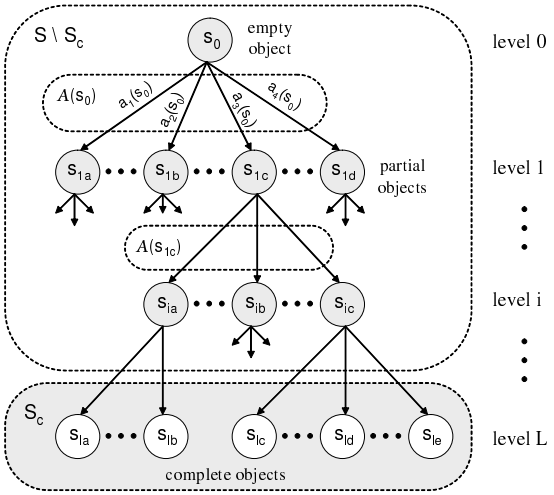

Fig. 1. Graph representation

a lot of problems (e.g. scheduling problems, the *traveling salesman problem*[1]) this rooted graph has special properties (see Fig. 1.):

(g1) The graph is a tree and its leaves are labeled exactly with the elements of S_c;
(g2) All paths form the root to the leaves have the same path length (that is denoted by $L \in \mathbb{N}$);
(g3) The operators change the performance of the object, which they have applied to, with only a "small" amount: $\exists\, \alpha, \beta \ll (\kappa_{max} - \kappa_{min}) : \forall s \in S : \forall a \in \mathcal{A}(s) : \alpha \leq \kappa(a(s)) - \kappa(s) \leq \beta$;

where $\kappa_{max} = \max\{\kappa(s) \mid s \in S_c\}$ and similarly for κ_{min}. E. g., for the TSP: $L = |V| - 1$ (where V is the set of "cities") and $\alpha = \min\{w(e) \mid e \in E\}$, $\beta = \max\{w(e) \mid e \in E\}$. For the scheduling problem: $L = |V|$, and if $\kappa = C_{max}$ then $\alpha = 0$ and $\beta = \max\{p(v, m) \mid v \in V \wedge m \in M\}$.

If the algorithm searches by moving from the root to a leaf, then at every $s \in S \setminus S_c$ it must make a decision which operator to apply from $\mathcal{A}(s)$. The algorithm can safely skip those s nodes which have the property: $\kappa(s) + (L - i)\,\alpha \geq \kappa_{tm}$, where i denotes the level of the node (its distance from the root) and κ_{tm} (temporary minimum) denotes the performance of the so far found best solution.

Algorithms which use some bounding procedures (such as the aforementioned) to purge the search tree but otherwise they search exhaustively are often referred as *branch and bound*.

Note that a natural way of building domain specific knowledge into the system is to control the generation of $\mathcal{A}(s)$ and do not allow operators which result in an object that is surely (or with high probability) not lead to an optimal element (e.g., some cutting conditions are met).

[1] TSP: given a complete weighted graph $G = \langle V, E, w \rangle$ find a Hamilton circuit with the smallest possible weight.

4. REINFORCEMENT LEARNING

Reinforcement learning (RL) is a behavioral learning method, which is performed through *interactions* between the learning system and its environment. The modern approach of reinforcement learning is often called *neurodynamic programming* because its theoretical foundation is based on dynamic programming and its learning capacity is often provided by artificial neural networks.

The operation of a general reinforcement learning system can be characterized as follows:

(r1) The environment evolves by probabilistically occupying a finite set of discrete states, S.
(r2) For each state $s \in S$ there exits a finite set of possible actions that may be taken, $\mathcal{A}(s)$.
(r3) Each time the system takes an action $a \in \mathcal{A}(s)$, a certain reward is incurred, $r \in \mathbb{R}$.
(r4) States s_t, actions a_t and rewards r_t are taken place at discrete time steps, $t \in \mathbb{N}$.

The goal of the system is to maximize its *expected cumulative reward* (the profit in the long run).

The used notations show that the building of a schedule from an empty one by applying operators from $\mathcal{A}(s)$ fits well into the RL framework.

Using reinforcement learning for job-shop scheduling was first proposed in (Zhang and Dietterich, 1995). They used the $TD(\lambda)$ method with iterative repair to solve a static scheduling problem, namely the NASA space shuttle payload processing problem. A multi-agent based scheduling with learning agents was presented in (Brauer and Weiß, 1998), which used a simplified version of *Q-learning* with selfish and non-cooperating agents. Aydin and Öztemel developed an improved version of Q-learning, which they called *Q-III learning* that they have built into an agent-based system, however, scheduling was made in a centralized way (Aydin and Öztemel, 2000). The authors of this paper have introduced the first version of the presented ideas in (Csáji *et al.*, 2003).

The paper suggests using *episodic* learning, thus the system has to face the schedule building process repeatedly. It gets 0 immediate reward in every $s \in S \setminus S_c$ and $\kappa(s)$ reward at the end of each episode, when $s \in S_c$. Its aim is to *minimize* these values. An ostensible difference from the classical viewpoint of RL is that the scheduling problem is a minimization problem, however it is straightforward to modify the different learning formulas by this criteria. In the following investigations, the minimization point of view will be applied.

The *Markov property* is satisfied in these kinds of systems. Markov decision processes are based on a *policy* π for selecting actions in the problem space. The policy defines the actions to be performed in

each state. Formally, a policy $\pi : S \to \Delta(A)$ is a function from states to probability distributions over actions. The probability of taking action a in state s is denoted by $\pi(s,a)$. The *action-value function* $Q^\pi : S \times A \to \mathbb{R}$ is defined by the expected total reward when the system is in state s, it takes action a and it follows π thereafter:

$$Q^\pi(s,a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k} \mid s_t = s, a_t = a\right], \quad (1)$$

where r_t is the reward at time t and $\gamma \in [0,1]$ is a parameter called the *discount rate*. The *state-value* function for policy π is defined by:

$$V^\pi(s) = \mathbb{E}\left[Q^\pi(s,\pi(s))\right], \quad (2)$$

where $\pi(s)$ denotes a random variable that gives the action selected by control policy π in state s. In the presented system episodic learning is used during scheduling, therefore the system learns from episode-by-episode (and not step-by-step). One episode is the building of a complete schedule, or in other words: moving in the representation tree from the root to a leaf. The system gets reward (penalty) at the end of each episode, only and the environment is deterministic, therefore the value function can be written in the form:

$$V^\pi(s) = \begin{cases} \kappa(s) & \text{if } s \in S_c \\ \sum_{a\in\mathcal{A}(s)} \pi(s,a)V^\pi(a(s)) & \text{if } s \notin S_c \end{cases} \quad (3)$$

In these type of systems γ has no role, thus $\gamma = 1$ was supposed. A policy π is better than or equal to a policy π' if and only if $V^\pi(s) \geq V^{\pi'}(s)$ for all $s \in S$. There is always at least one policy, the *optimal policy*, that is better than or equal to all other policies. These policies are denoted by π^*. Although there may be many optimal policies, they all share the same value function, called the *optimal value function*, denoted by V^* or Q^*.

As in most reinforcement learning work, the aim of the presented learning system is to learn the optimal value function rather than directly learning an optimal policy. To learn the value function a modified version of *Q-learning* is applied. The general version of the one-step Q-learning rule is:

$$\delta_t = \alpha_t\left[r_{t+1} - Q_t(s_t,a_t) + \gamma \min_{a\in\mathcal{A}(s_{t+1})} Q_t(s_{t+1},a))\right],$$
$$Q_{t+1}(s_t,a_t) \leftarrow Q_t(s_t,a_t) + \delta_t, \quad (4)$$

where α_t is a sequence that defines the *learning rates* of the system. The Q values converge almost surely to the optimal action-value function if the following three conditions are satisfied:

(q1) $\sum_{t=1}^{\infty}\alpha_t = \infty$ (thus, any value can be reached in finitely many steps);
(q2) $\sum_{t=1}^{\infty}\alpha_t^2 < \infty$ (which is required to show the convergence with probability one);
(q3) Each state action performed infinitely often.

Note that (q1) and (q2) can be easily satisfied, for example, by $\alpha_t = 1/t$. Satisfying (q3) requires an *exploration* strategy (e.g. an ϵ-soft strategy). The next section will briefly investigate such methods. If both (q1), (q2) and (q3) are satisfied then

$$\mathbb{P}\left[\lim_{t\to\infty}\|Q_t - Q^*\|_\infty = 0\right] = 1, \quad (5)$$

for a proof and learning rates for Q-learning see (Even-Dar and Mansour, 2003). If the environment is deterministic, the Q-learning formula can be rewritten to use state-value functions:

$$\delta_t = \alpha_t\left[r_{t+1} - V_t(s_t) + \gamma \min_{a\in\mathcal{A}(s_t)} V_t(a(s_t))\right],$$
$$V_{t+1}(s_t) \leftarrow V_t(s_t) + \delta_t, \quad (6)$$

this approach has the advantage that it requires a much smaller storage space and the convergence of learning will also be faster, since the algorithm has to maintain a much smaller number of values. This version can be called *V-learning*[2] . If episodic learning is applied, this rule can be simplified:

$$V_{i+1}(s_{ij}) = \begin{cases} \kappa(s_{ij}) & \text{if } s_{ij} \in S_c, \\ \min_{a\in\mathcal{A}(s_{ij})} V_i(a(s_{ij})) & \text{if } s_{ij} \notin S_c, \end{cases} \quad (7)$$

where $i \in \mathbb{N}$ denotes the episode number, $j \in \{0,\ldots,L\}$ denotes the level number in the tree, and $s_{ij} \in S$ are the states which were visited during episode i. If the system is also static, as e.g. in the scheduling problem presented in Section 2, this learning rule can be even further simplified:

$$V_{i+1}(s_{ij}) \leftarrow \min\{V_i(s_{ij}), \kappa(s_{iL})\}, \quad (8)$$

where $\kappa(s_{iL})$ is the reward in episode i ($s_{iL} \in S_c$). In that case, the V values should set to extremal ones prior to learning, e.g., $V(s) = \kappa(s_0) + L \cdot \beta$ for all $s \in S$ is a good choice. For stochastic problems the Q-learning version should be used, however the learning formula presented in (4) can be modified for episodic decision processes:

$$\delta_{ij} = \alpha_i[\kappa(s_{iL}) - Q_i(s_{ij},a_{ij}) + \min_{a\in\mathcal{A}(s_{ij+1})} Q_i(s_{ij+1},a))]$$
$$Q_{i+1}(s_{ij},a_{ij}) \leftarrow Q_i(s_{ij},a_{ij}) + \delta_{ij}, \quad (9)$$

Summing up, the paper suggests using episodic Q-learning for the scheduling problem, but depending on the task and the environment (whether it is deterministic or stochastic, static or dynamic) different learning rules are suggested. Finally, Table 1. summarizes the advised learning formulas.

Table 1. Advised episodic learning rules

	Static env.:	*Dynamic env.*:
Deterministic:	V-learning (8)	V-learning (7)
Stochastic:	Q-learning (9)	Q-learning (9)

[2] Note that sometimes $TD(\lambda)$ is also called V-learning.

5. EXPLORATION STRATEGY

In this section the exploration strategy for the static (stationary) case is investigated. In order to ensure the convergence of Q-learning, one must guarantee that each state-action pair is continue to update. To ensure that, the system has to make explorations. The class of ϵ-*soft* algorithms can be defined by the property that they work in two ways. Mostly they make *exploiting* actions, in which they use the previously gathered information and select an object which will be "good" with high probability, however, sometimes they make *exploring* actions as well and select solutions randomly, to gather more information on S.

An often used technique to balance between exploration and exploitation is the *Boltzmann formula*. In the suggested algorithm the *policy* π is computed from the value-function estimations with a modified Boltzmann formula. This formula is modified to avoid the property of finding the same solutions over and over again (getting stucked in a local minimum). To present this two new functions are required. $T_i : S \to \mathbb{N}$ is a *trial counting* function, $T_i(s)$ shows that how many times visited the algorithm the state s until iteration i. $N : S \to \mathbb{N}$ is a *terminal node number estimating* function, $N(s)$ gives the number of leaves that can be reached from the node s, or at least it gives an *upper* estimation of this. The Boltzmann formula:

$$\pi_i(s,a) = \frac{e^{\varrho_i(a(s))/\tau}}{\sum\limits_{b \in \mathcal{A}(s)} e^{\varrho_i(b(s))/\tau}}, \qquad (10)$$

where i is the episode number and τ is the Boltzmann (or Gibbs) temperature. High temperatures cause the actions to be (nearly) equiprobable, low ones cause a greater difference in selection probability for actions that differ in their value estimations. The ϱ_i function is defined by:

$$\varrho_i(s) = \left(1 - \frac{T_i(s)}{N(s)}\right) \cdot \frac{1}{V_i(s)}, \qquad (11)$$

if $\varrho_i(s)$ approaches 0 for an $s \in S$ that means the subtree starting at s was roughly exhaustively searched. Note that $\forall s \in S : T_0(s) = 0$ and normally $T_i(s)$ is incremented by 1 in every episode if s was visited during that episode, however if there was a cut at a node s' somewhere under s then $T_i(s)$ must be increment by $N(s')$ instead.

This Boltzmann formula can be easily rewritten to use action-value functions ($Q(s,a)$).

This approach is often extended by *simulated annealeing* and τ is decreased over time. One can even set different temperatures for different states, however, it can be shown that the rate of decreasing should be bounded from below by $o(1/ln(T_i(s)))$ to ensure sufficient exploration.

6. FUNCTION APPROXIMATION

It is possible that for large systems, the state space S of reinforcement learning is too big to fit to the memory. In this case a *numerical function approximator* should be used to approximate the optimal value function. These techniques could further increase the performance of the system by extrapolating the estimations to the states which were never experienced (generalization).

If the value function $V(s)$ is represented by a function $\forall s \in S : f(s,w) \approx V(s)$ where $w \in \mathbb{R}^d$ is a vector containing the parameters of the approximation (e.g., weights in the case of artificial neural networks) then to learn the optimal value function, $V(\lambda)$ could be applied. At step $t+1$, the error at time t can be computed as (V-learning):

$$\delta_t = r_{t+1} - f(s_t,w) + \gamma \min_{a \in \mathcal{A}(s_t)} f(a(s_t),w), \quad (12)$$

then the smoothed gradient can be computed:

$$e_t = \nabla_w f(s_t,w) + \lambda\, e_{t-1}, \qquad (13)$$

finally, the parameters can be updated:

$$\Delta w = \alpha_t\, \delta_t\, e_t, \qquad (14)$$

where $\lambda \in [0,1]$ is a smoothing parameter that combines the previous gradients with the current one (e_t), and α_t is the learning parameter. Again, it is straightforward to modify this method for action-value functions ($f(s,a,w) \approx Q(s,a)$).

7. DISTRIBUTED OPTIMIZATION

Because its complexity, scheduling is a computationally very expensive process and therefore it is important to calculate it in a distributed way. If a common main memory is available to the processors, then it is straightforward to parallelize the given solution: each processor searches in S independently, but they all share the same value function (and the same trial counting function).

A more interesting case, when the scheduling is made in a multi-agent system and not only the computation but also the memory is local to the agents, thus the global information is eliminated. A multi-agent system is called *heterarchical*, if the agents communicate as peers, no fixed master/slave relationships exist. The advantages these heterarchical systems include: self-configuration, scalability, fault tolerance, massive parallelism, reduced complexity, increased flexibility, reduced cost and emergent behavior (Ueda *et al.*, 2001).

If the agents cooperate intelligently, each agent can benefit from the other agents information (Tan, 1993). There are several ways of cooperation: the agents can communicate instantaneous information, episodes or policies, etc. For

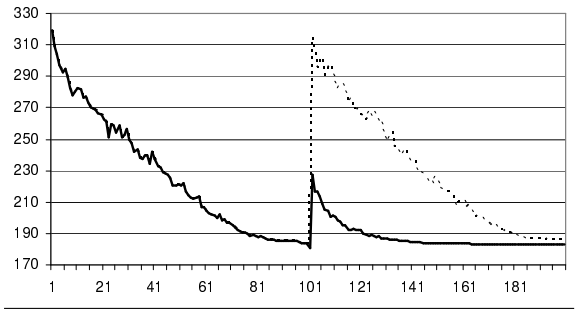

Fig. 2. Managing changes during scheduling

an overview on distributed reinforcement learning, see (Barto and Mahadevan, 2003). For the scheduling case, this paper suggests that the agents periodically communicate a fixed number of their best episodes since the last synchronization (and by this way they help improving each others state or action value functions), but otherwise they do their stochastic search independently.

8. EXPERIMENTAL RESULTS

In order to verify the above algorithm in dynamic environments, experiments were carried out. In the test program V-learning (7) was used and the aim of scheduling was to minimize the maximum completion time (C_{max}). The adaptive features of the algorithm was tested by confronting it with unexpected events, such as: machine breakdown, new machine, new job and job cancellation.

In Fig. 2. the x-axis represents time, while the y-axis the achieved performance measure. It was made by averaging hundred random samples (runtime results). In this test 20 machines were used with few dozens of jobs. In all test cases at time $t = 100$ there were unexpected events. The results show that the presented system is adaptive, because it did not recompute the whole schedule from scratch, but it tried to use previously gathered information from the past. The performance measure which would arise if it recomputed the whole schedule is drawn in a broken line.

9. CONCLUDING REMARKS

The paper presented a neurodynamic programming based solution to a scheduling problem with unrelated parallel machines. Different Q-learning rules for various scheduling environments were proposed. The exploration strategy, parametric function approximation and the parallelization of the solution were briefly investigated. Some preliminary experimental results were also showed.

There are several further research directions. For practical reasons, it would be important to handle set up times, transportations, storage spaces, production costs, etc., as well. The optimal representation in the interest of function approximation is also an open question. Finally, application of the described algorithm to other combinatorial optimization problems, would also be promising.

10. ACKNOWLEDGEMENTS

This research was partially supported by the National Research and Development Programme (NKFP), Hungary, Grant No. 2/010/2004 and by the Hungarian Scientific Research Fund (OTKA), Grant No. T049481 and T043547.

REFERENCES

Aydin, M. E. and E. Öztemel (2000). Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems* **33**, 169–178.

Barto, A. and S. Mahadevan (2003). Recent advances in hierarchical reinforcement learning. *Special Issue on Reinforcement Learning, Discrete Event Systems* **13**, 41–77.

Brauer, W. and G. Weiß (1998). Multi-machine scheduling - a multi-agent learning approach. In: *Proceedings of the 3rd International Conference on Multi-Agent Systems*. pp. 42–48.

Csáji, B. Cs., B. Kádár and L. Monostori (2003). Improving multi-agent based scheduling by neurodynamic programming. In: *First International Conference on Holonic and Multi-Agent Systems for Manufacturing*. Lecture Notes in Computer Science. pp. 110–123.

Even-Dar, E. and Y. Mansour (2003). Learning rates for Q-learning. *Journal of Machine Learning Research* **5**, 1–25.

Lawler, E. L., J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys (1993). Sequencing and scheduling: algorithms and complexity. *Handbooks in Operations Research and Management Science* **4**, 445–522.

Tan, M. (1993). Multi-agent reinforcement learning: independent vs. cooperative learning. In: *Proceedings of the 10th International Conference on Machine Learning*. pp. 330–337.

Ueda, K., A. Márkus, L. Monostori, H. J. J. Kals and T. Arai (2001). Emergent synthesis methodologies for manufacturing. *Annals of the CIRP* **50**(2), 535–551.

Williamson, D. P., L. A. Hall, J. A. Hoogeveen, C. A. J. Hurkens, J. K. Lenstra, S. V. Sevastjanov and D. B. Shmoys (1997). Short shop schedules. *Operations Research* **45**, 288–294.

Zhang, W. and T. Dietterich (1995). A reinforcement learning approach to job-shop scheduling. In: *14th International Joint Conference on Artificial Intelligence*. pp. 1114–1120.