LEARNING AND COOPERATION IN A DISTRIBUTED MARKET-BASED PRODUCTION CONTROL SYSTEM

Balázs Csanád Csáji, László Monostori, Botond Kádár
Computer and Automation Research Institute,
Hungarian Academy of Sciences
Kende u. 13-17, Budapest, H-1111, Hungary
Phone: (36 1) 297-6115, Fax: (36 1) 4667-503,
{*csaji, monostor, kadar*}*@sztaki.hu*

Abstract

The paper presents an adaptive iterative distributed scheduling algorithm that operates in a market-based production control system. The manufacturing system is agentified, thus every machine and job is associated with its own software agent. Each agent learns how to select presumably good schedules, by this way the size of the search space can be reduced. In order to get adaptive behavior and search space reduction, a triple-level learning mechanism is proposed. The top level of learning incorporates a simulated annealing algorithm, the middle (and the most important) level contains a reinforcement learning system, while the bottom level is done by a numerical function approximator, such as an artificial neural network. The paper suggests a cooperation technique for the agents, as well. It also analyses the time and space complexity of the solution and presents some experimental results.

Keywords: dynamic scheduling, multi-agent systems, reinforcement learning, market-based scheduling

1. Introduction

Computer science has made an explosion-like progress since the middle of the past century. However, as computer science broke out from laboratories and classrooms and started to deal with "real world" problems, it had to face major difficulties. Namely, in practice, we mostly have only *incomplete* and *uncertain* information on the environment (surrounding world) that we must work in, in addition, this environment could be *non-stationary.* Moreover, we also have to face *complexity* problems, viz. even if we deal with static, highly simplified and abstract problems and we know that the solution exists and can be attained in finitely many steps, we may not have enough computation power to achieve it in practice (as this is the case, for example, with NP-hard problems).

One way to overcome these difficulties is to use *machine learning* techniques. It means designing systems which can adapt their behavior to the current state of the environment, extrapolate their knowledge to the unknown cases and can learn how to optimize the solutions of the problems that they should deal with. The importance of learning was recognized even by the founders of computer science. It is well known, for example, that the Hungarian John von Neumann was keen on artificial life and, besides many other things, he designed self-organizing (and reproducting) automata (Neumann, 1948). The English Alan Turing in his famous paper (Turing, 1950), which we can treat as one of the starting articles of artificial intelligence research, wrote that instead of designing extremely complex and large systems, we should desing programs which can learn how to work efficiently by themselves.

In the paper we try to apply machine learning techniques to an important manufacturing problem, which has all the difficulties we have mentioned in the previous parts, namely: production control. In manufacturing systems of our days, difficulties arise from unexpected tasks and events, non-linearities, and a multitude of interactions while attempting to control various activities in dynamic shop floors. Complexity and uncertainty seriously limit the effectiveness of conventional control scheduling approaches. Multi-agent based (holonic) control architectures offer prospects of reduced complexity, high flexibility and a high robustness against disturbances. Summing up, we present an *adaptive iterative distributed scheduling algorithm.*

The structure of the paper is as follows: first, we give a brief introduction to the problem of scheduling and highlight its complexity. Next, we overview the importance of distributed approaches to this problem, then we introduce a market-based production control system with resource and order agents. After that we overview some results from the theory of neurodynamic programming and demonstrate how this technique can be used for achieving efficient adaptive scheduling. We will use a modified version of reinforcement learning theory, in which it is allowed to make branches, but we will argue that this version can behave in the same way, as the original approach did. After that, we present some ideas on the extra cooperation of order agents, analyse the time and space complexity of our solution and, finally, we demonstrate some experimental results.

2. Scheduling

Scheduling is the allocation of resources over time to perform a collection of jobs. Near-optimal scheduling is a prerequisite for the efficient utilization of resources and hence for the profitability of the enterprise. Thus, scheduling is a key problem in a production control system. Moreover, much of what we can learn about scheduling can be applied to other kinds of decision-making and therefore, is of general practical value.

2.1 Static Job-Shop Scheduling

One of the basic scheduling problems is the problem of *job-shop* scheduling. We begin by defining the general job-shop problem. Suppose that we have n jobs $J = \{J_1, J_2, \dots J_n\}$ to be processed through m machines $M = \{M_1, M_2, \dots, M_m\}$. The processing of a job on a machine is called operation or task. The operations are non-preemptive (they may not be interrupted) and each machine can process at most one operation at a time (capacity constraint). Each job may be processed by at most one machine at a time (disjunctive constraint). $O = \{O_1, O_2, \dots, O_k\}$ denotes the set of all operations. Every job has a set of operation sequences: the possible process plans, $o : J \to \mathcal{P}(O^*)$. These sequences give the precedence constraints of the operations. The processing time of an operation on a machine is given by $p : M \times O \to \mathbb{R}^+$ which is a partial function. The job-shop problem (especially in practice) is very often extended with due dates. These dates are assigned to every job: $d : J \to \mathbb{R}^+$, where $d(J_i)$ shows the time by which we would like to have J_i completed ideally. Note that we can include these kinds of constraints in the performance measure. It is not easy to state our objectives in scheduling. They are complex, and often conflicting. Generally, we may say that the objective is to produce a schedule that maximizes (or minimizes) a performance measure f, which is usually a function of job completion times. Naturally, we do not allow *any* function as a performance measure. We will restrict ourselves to performance measures which have the property that the schedule can be uniquely generated from the order in which the jobs are processed through the machines (called a *sequence*), e.g., by semi-active timetabling. Regular measures, for example, satisfy this property. Let C_i denote the completion time of J_i, i.e., the time at which processing of J_i finishes. We call a performance measure *regular* if it is monotonic in completion time, for example, if $C_1 \leq C_1^{'}, C_2 \leq C_2^{'}, \dots, C_n \leq C_n^{'}$ then $f(C_1, C_2, \dots, C_n) \leq f(C_1^{'}, C_2^{'}, \dots, C_n^{'})$. All of the usually used performance measures are regular (e.g.: maximum completion time, mean flow time, mean tardiness, number of tardy jobs, etc.). We can conclude that the *general* job-shop scheduling problem is a combinatorial optimization problem defined by the six-tuple: $JSP = (J, M, O, p, o, f)$. Note that in the *standard* or *classical* job-shop scheduling problem *none* of the machines are interchangeable, each job has only one process plan and every job must be processed on every machine once, only[1].

2.2 Complexity of Job-Shop Scheduling

The job-shop scheduling problem, even the standard variant and except for some strongly restricted special cases[2], is an NP-hard optimization problem (Lawler *et al.*, 1993). It means that unless $P = NP$, no polynomial time algorithm exists that always computes the exact optimal schedule. A very common performance measure is $C_{max} = max\{C_1, C_2, \dots, C_n\}$ which is also called total production time or make-span. Williamson *et al.* proved that, if our performance measure is C_{max}, there is no good polynomial time approximation of the optimal scheduling algorithm (Williamson *et al.*, 1997). Even a much simpler version of the standard job-shop scheduling problem when all the jobs share the same production order, called the *flow-shop* problem, is NP-hard (Baker, 1998) if our performance measure is C_{max} and $m > 2$. We can highlight the seriousness of the problem by giving an example. In the case of the standard job-shop problem the size of the search space is $(n!)^m$. Thus, it is not possible to try every potential solution, not even in cases such as $n = 12$ and $m = 10$ because even in this case the size of the search space is much larger than the number of particles in the known Universe[3]. Because of these properties, job-shop scheduling has earned the reputation of being notoriously difficult to solve.

2.3 Dynamic Scheduling

In Section 2.1 the standard static scheduling problem was presented. In that problem all information was available initially and it did not change over time. Most of the solutions in the literature concerning scheduling concentrates on this static problem. However, in "real world" situations the scheduling is very seldom static. Events such as arrivals of new jobs or machine breakdowns are, in some situations, impossible to predict. We can generate a dynamic problem (we can call it $DJSP$) from the static problem of JSP by considering time. If, for example, the set of jobs can vary over time, we can modify their definition by

[1]Thus, every job consists of exactly m operations and for each operation there is exactly one machine that can process it.

[2]Like single/double machine problems if the performance measure is the maximum completion time/lateness/tardiness.

[3]According to Arthur Eddington (Mathematical Theory of Relativity), the number of particles in the known Universe is $\approx 3.1495 \cdot 10^{79}$ and $(12!)^{10} \approx 6.3587 \cdot 10^{86}$.

$J_t = \{J_{t,1}, J_{t,2}, \dots, J_{t,n(t)}\}$, where $t \in \mathbb{R}$, and similarly for M_t and o_t. Naturally, we must presume that the situation changes relatively slowly.

Many different approaches, such as integer programming, Lagrangian relaxation, branch and bound algorithms, simulated annealing, genetic algorithms, neural networks, reinforcement learning, etc. have been tried for solving the job-shop scheduling problem (Baker, 1998). Though, some of them, e.g., integer programming, have elegant mathematics, their time requirement grows exponentially. Others, such as genetic algorithms, provide some promising experimental results but it is not clear how to put these approaches into a distributed (multi-agent) system, and moreover, they solve the static scheduling problem, only.

One can generate a dynamic scheduler from a static one, by systematically recomputing the whole schedule if the system changes. However, as we saw, computing an optimal (or even a relatively good) schedule is extremely time-consuming, thus we should avoid these ideas. We have to use all the information that we have gathered from the previous scheduling attempts and use these data as a starting point for our new schedule. Our adaptive scheduling algorithm try to overcome uncertainties and dynamic changes but, additionally, it also makes an attempt to decrease computation costs, thus it tries to generalize information from the past.

3. Multi-Agent Systems and Manufacturing

Before we continue our investigation on adaptive job-shop scheduling, let us give a short review on multi-agent systems in general and in manufacturing.

According to Baker, an agent is basically a self-directed software object. It is an object with its own value system and a means to communicate with other objects like this. Unlike a lot of software, which must be specifically called upon to act, the agent software continuously acts on its own initiative (Baker, 1998).

In a heterarchical architecture, agents communicate as peers, no fixed master/slave relationships exist, each type of agents is usually replicated many times, and global information is eliminated. The advantages of these heterarchical multi-agent systems include: self-configuration, scalability, fault tolerance, massive parallelism and emergent behaviour (Ueda *et al.*, 2001). Other authors claim that the advantages of heterarchical architectures also include reduced complexity, increased flexibility and reduced cost. This approach is useful for manufacturers who often need to change the configuration of their factories by adding or removing machines, workers, product lines, manufacturers who cannot predict the possible manufacturing scenarios according to which they will need to work in the future.

An agent-based (holonic) reference architecture for manufacturing systems is PROSA (Van Brussel *et al.*, 1998). The general idea underlying this approach is to consider both the machines and the jobs as active entities. The basic architecture of the PROSA approach consists of three types of basic agents: order agents (internal logistics), product agents (process plans), and resource agents (resource handling). A further improvement of this architecture can be found in (Hadeli *et al.*, 2004) where a coordination and control technique was presented, which was inspired by food-foraging ants. The PROSA architecture was extended with mobile agents, called ants. The key achievement of this biological example can be identified as the limited exposure of the individuals, combined with the emergence of robust and optimized overall system behavior.

Multi-agent based or holonic manufacturing with adaptive agents received a great deal of recent attention (Monostori *et al.*, 2002). They became a promising tool for managing complexity, changes and uncertainties in manufacturing. These approaches, like ours, use machine-learning techniques, but in a distributed way: learning is shared among several agents and none of them is required to solve the learning task alone.

Using reinforcement learning for job-shop scheduling was first proposed by Zhang and Dietterich in 1995. They used the $TD(\lambda)$ method, which we present later, in a centralized way to solve a static scheduling problem, namely the NASA space shuttle payload processing problem (Sutton, 1998). 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 (2000) 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. The first version of our solution with a triple-level learning system was presented in (Csáji *et al.*, 2003).

4. Market-Based Production Control

As we stated before, our objective is to propose an adaptive iterative distributed scheduling algorithm that operates in a market-based production control system. We call it "market-based", however, it is just a metaphor, it helps to understand the basic ideas and the general structure of the solution, but (like every metaphor) it breaks at some point.

The idea of market or negotiation based scheduling has emerged long before, for example, a holonic market approach with cooperative agents and local problem solving can be found in (Márkus *et al.*, 1996).

Now, we informally define the basic frame of our approach. In a multi-agent based manufacturing system, autonomous agents control different real world entities. In the presented system the two most important types of agents are the *resource agents* and the *order agents*. Resource agents control physical parts (like machines, furnaces, conveyors, pipelines, material storages, etc.), while order agents control the production of a job.

In our market-based production control system if a new job arrives at the system, a new order agent is created and associated with that job. An order agent or a group of cooperating order agents announces a sequence of operations and the resource agents can bid for that sequence. Only resource agents being able to do at least the first operation of that job are allowed to bid. Before an agent bids, it gathers information about the possible costs of making that sequence. If the sequence contains only one operation, the agent has all the information it needs, however, if the sequence contains other operations as well, which probably cannot be processed by the machine of the agent, it starts to search for subcontractors. It becomes a partial order agent and announces the remaining part of the sequence. The other resource agents, which can do the next operation, may bid for the remaining operation sequence. Consequently, a recursive announce-bid process begins. At the end, when all the possible costs of that (partial) job are known, the agent bids. If the order agent, which announced that job, is contented with it (it is the best bidder), the agent (and its subcontractors) get the job (award). Thus, the schedule generation is a recursive, iterative process with announce-bid-award cycles based on market mechanisms.

5. Learning Agents

The main problem with the mechanism described above is the combinatorial explosion of the possible schedules. More precisely, it makes a complete enumeration, in some sense, and thus, its time complexity makes it unusable in practice (see the example in Section 2.2 on complexity). The agents should not investigate every potential schedule, because this can be extremely time-consuming. If an agent wants to bid for an operation sequence and it needs information about the production costs of the part of the job, which it cannot do, it should not announce the part to every resource agent. It should make only a restricted tendering among the agents that will give a presumably good bid. The paper suggests that the agents should use *neurodynamic programming* for learning the presumably good bidders for every operation sequence in a given time. Before we present our solution, we review the basic ideas of neurodynamic programming.

5.1 Neurodynamic Programming

Two main paradigms of machine-learning are known: learning with a teacher, which is called *supervised learning*, and *learning without a teacher.* The paradigm of learning without a teacher is subdivided into *self-organized (unsupervised)* and *reinforcement learning.* Supervised learning is a "cognitive" learning method performed under a tutelage of a teacher: this requires the avaibility of an adequate set of input-output examples. In contrary, reinforcement learning is a "behavioral" learning method, which is performed throught *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 reinforcement learning system is characterized as follows:

(1) The environment evolves by probabilistically occupying a finite set of discrete states, S.

(2) For each state $s \in S$ there exits a finite set of possible actions that may be taken, $\mathcal{A}(s)$.

(3) Every time the learning system takes an action $a \in \mathcal{A}(s)$, a certain reward is incurred, $r \in \mathbb{R}$.

(4) States s_t are observed, actions a_t are taken, and rewards r_t are incurred at discrete time steps $t \in \mathbb{N}$.

The goal of the learning system is to maximize its *commulative reward.* This does not mean maximizing immediate gains, but the profit in the long run.

5.1.1 Markov Property

An important assumption in reinforcement learning is the *Markov property.* A sequence of random variables $\{X_t\}$ where $t \in \mathbb{N}$ have the Markov property if they satisfy the following equality:

$$P(X_t = j \mid X_0 = i_0, X_1 = i_1, \ldots, X_{t-1} = i_{t-1}) = \\ = P(X_t = j \mid X_{t-1} = i_{t-1}) \quad (1)$$

Or in other words, the present is conditionally independent of the past. The environment satisfies this property if its state signal compactly summarizes the past without degrading the ability to predict the future. A reinforcement learning task is called a *Markov Decision Process* or *MDP* if its environment satisfies the Markov property (Sutton, 1998).

Markov states provide the best possible basis for choosing actions. Even if the state signal is non-Markov, it is appropriate to consider it as an approximation to a Markov state. During the paper we presuppose that our system satisfies this property.

5.1.2 Temporal Difference Learning

Reinforcement learning methods 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 is $\pi : S \times A \rightarrow [0, 1]$ a partial function from state and actions to the probability $\pi(s, a)$ of taking action a in state s. The *value function* $V^\pi : S \rightarrow \mathbb{R}$ is defined by the expected return (sum of rewards) when starting in a state s and following π thereafter:

$$V^\pi(s) = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\}, \quad (2)$$

where r_t is the reward at time t and $\gamma \in \mathbb{R}$ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate* (if $\gamma = 0$, the system is "myopic", and as γ approaches 1, the agent becomes more and more "farsighted").

A policy π is better than or equal to a policy π' if and only if $\forall s \in S : V^\pi(s) \geq V^{\pi'}(s)$. There is always at least one policy, the *optimal policy*, that is better than or equal to all other policies. We denote these policies by π^*. Although there may be many optimal policies, they all share the same value function, called the *optimal state-value function*, and defined as:

$$\forall s \in S : V^*(s) = \max_{\pi} V^{\pi}(s) \tag{3}$$

As in most reinforcement learning work, the aim of our system is to learn the optimal value function V^* rather than directly learning a π^*. To learn the value function we can apply the method of *temporal difference learning* known as $TD(\lambda)$, developed by Sutton (1998). The value function $V^{\pi}(s)$ is represented by a function approximator $f(s, w)$ where $w \in \mathbb{R}^d$ is a vector containing the parameters of the approximation (e.g., weights if we use an artificial neural network). If the policy π were fixed, $TD(\lambda)$ could be applied to learn the value function V^{π} as follows. At step $t + 1$, we can compute the temporal difference error at time t as:

$$\delta_t = r_{t+1} + \gamma\, f(s_{t+1}, w) - f(s_t, w), \tag{4}$$

then we can compute the smoothed gradient:

$$e_t = \nabla_w f(s_t, w) + \lambda\, e_{t-1}, \tag{5}$$

finally, we can update the parameters according to:

$$\Delta w = \alpha\, \delta_t\, e_t, \tag{6}$$

where λ is a smoothing parameter that combines the previous gradients with the current one (e_t), and α is the learning parameter. This way, $TD(\lambda)$ could learn the value function of a *fixed* policy. But we want to learn the value function of the *optimal* policy. Fortunately, we can do this by the method of *value iteration.* During the learning we continually choose an action that maximizes the predicted value of the resulting state (with one step lookahead). After applying this action, we get a reward, and update our value function estimation. This means that the policy continually changes during the learning process. $TD(\lambda)$ still converges under these conditions (Sutton, 1998).

5.2 Iterative Probing

The general idea of our solution as follows: we consider the problem as a tree (graph), which nodes are (partial) schedules and its edges represent consigning a (partial) operation sequence to a resource agent. Naturally, we do not investigate all of the possible combinations, but make *probes* in the search space and from the attained bids (from the performance measures of the achieved schedules), we learn which routes in the graph we should investigate in the next iterations. The agents learn estimations of the expected profit if they announce a job to a specific agent. If we do not have estimations on a node yet, we can use local heuristics (such as dispatching rules) for better results. The agents learn these estimations with neurodynamic programming. In our system every agent learns independently. The *states* of the used reinforcement learning for deciding which action is to be taken is an operation sequence with its earliest start time. An *action* is the announcement of the sequence to a resource agent whose machine can do the next operation of the job. The *rewards* are computed from the given bids of the invited agents. The space-complexity can be reduced and the speed of convergence can be increased if an agent stores estimations about itself, only.

Naturally, an agent may announce a job to a lot of resource agents which means that it takes more than one action in a state. In that case, all of the agents which were invited become active and all of them try to search for further subcontractors. We call the expected number of invited resource agents the *branching factor* of the system. Note that it is not necessarily an integer. Taking multiple actions multiplies the number of actual states (a lot of agents can search for subcontractors simultaneously), however, it is easy to see that we can construct a new system in which it is not allowed to take more than one action in a state, but it is functionally equivalent to the original system. Suppose, for example, that S and A denote the set of all states and actions in our original system. The actions that can be taken in a state s is denoted by $\mathcal{A}(s) \subseteq A$. An action $a \in \mathcal{A}(s)$ is a function[4] $a : S \to S$. The received reward in state s is denoted by $r(s) \in \mathbb{R}$. The rewards in the orginal system are computed by some function g (e.g. average, minimum) of the received rewards in all of the actual states (if we took more than one action). Then we can construct a new system by defining the elements of the system in the following way: the new state set $\hat{S} = \mathcal{P}(S)$ is the power set of the original state set and the new action set $\hat{A}$ is defined by this way: an $\hat{a} : \hat{S} \to \hat{S}$ is included in $\hat{\mathcal{A}}(\hat{s})$ for an $\hat{s} \in \hat{S}$ if and only if there exists an $\alpha \in \mathcal{P}(S \times A)$ so that:

$$\begin{gathered} \forall (s, a) \in \alpha : s \in \hat{s} \,\wedge\, a \in \mathcal{A}(s) \,\wedge \\ \wedge\; \hat{a}(\hat{s}) = \{ s' \in S \mid \exists (s, a) \in \alpha : s' = a(s) \} \end{gathered} \tag{7}$$

The rewards are computed by $\hat{r}(\hat{s}) = g(\{ r(s) \mid s \in \hat{s} \})$. If we allow only one action in a state, this system functions in the same way as the original system did, but naturally, it has much more states. This remark is important because the convergence of temporal difference learning is proved for systems only in which it is not allowed to take multiple actions in a state. Our viewpoint of the system speeds up the learning process and reduces the needed storage space, as well.

The *policy* π, which is a partial function from state and actions to the probability of taking action a in state s, is computed from the cost estimations of feasible resource agents with a modified Boltzmann formula. Suppose, for example, that an estimation is available for the agent r for the expected return value if it announces a job with the remaining operation sequence $\gamma \in O^*$ to the resource agent p at time t. Let us denote this estimation by $V_p(s)$, where $s = (\gamma, t)$, $s \in S$ is the state signal. The branching factor of r is denoted by $\beta \in \mathbb{R}$. The set of all agents whose machine can do the next

[4]Generally, this is a relation, but in our case it is deterministic, so it is a function.

operation of the job is denoted by $R(\gamma_1)$. If our aim is to *maximize* the achieved return values, the probability of taking action a, which is the announcement of γ to the agent $p(a) \in R(\gamma_1)$, can be computed by:

$$\pi(s,a) = min \left\{ 1, \beta \, \frac{e^{V_{p(a)}(s)/\tau}}{\sum\limits_{q \in R(\gamma_1)} e^{V_q(s)/\tau}} \right\}, \qquad (8)$$

where 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. In the limit as $\tau \to 0$ the policy becomes the greedy policy. However, if we want to *minimize* the performance measure instead (e.g., in the case of C_{max}) then we can use the following formula:

$$\pi(s,a) = max \left\{ 0, 1 - (\alpha - \beta) \, \frac{e^{V_{p(a)}(s)/\tau}}{\sum\limits_{q \in R(\gamma_1)} e^{V_q(s)/\tau}} \right\}, \quad (9)$$

where α denotes the number of feasible agents, thus $\alpha = |R(\gamma_1)|$, $\forall a \in \mathcal{A}(s) : p(a) \in R(\gamma_1)$ and $0 < \beta \leq \alpha$.

In Figure 1 we can see how the system searches in the space of possible schedules by recursively inviting resource agents to bid for the remaining operation sequences. An agent is invited with $\pi(s,a)$ probability.

5.3 Numerical Function Approximators

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 we should use a *numerical function approximator* to approximate the optimal state-value function V^*. These techniques could further increase the performance of the system by extrapolating the estimations to the states which were never experienced (generalization). The type of the used approximator is not restricted, it can be a polynom, a spline, fourier series, wavelets, an artificial neural network, etc. However, it is important to have an *on-line* learning algorithm for the used method. This criteria and other useful properties, such as fault-tolerance and massive parallelism, make neural networks (e.g., support vector machines, radial basis function networks or multi-layer perceptrons) a promising choice.

5.4 Simulated Annealing

Another important idea is to balance the ratio of explorations and exploitations in the system. The paper suggests using *simulated annealing* to control this ratio. The *temperature* of the system is either the expected number of agents which will be invited to submit a tender for an operation sequence or it could be the Boltzmann temperature of the system, as well. If the system is stable, we can slowly cool the temperature down to exploit the information which was gathered. However, if the system changes, for example an unexpected event happens (such as a machine breakdown or a new job arrives, etc.), we can raise the temperature to force the system to make more explorations.

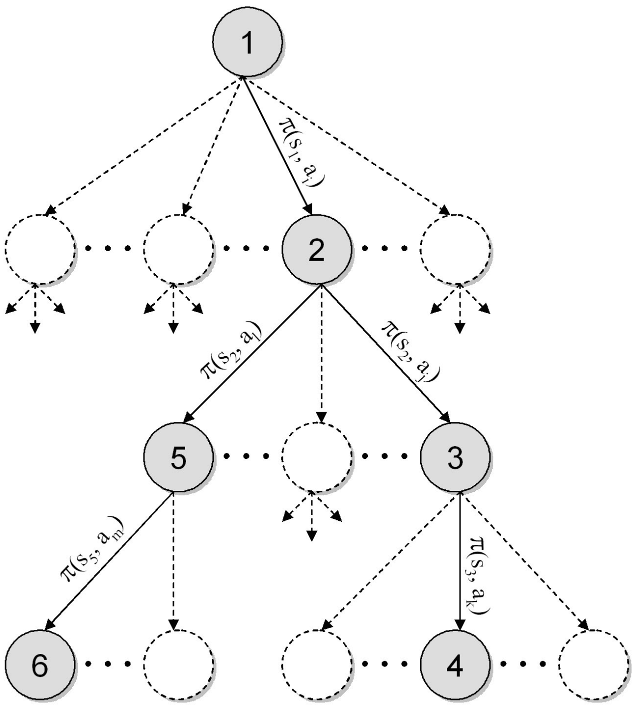

Fig. 1 Searching in the schedule graph

6. Cooperative Agents

The system described in Section 4 already contains a cooperative element, namely the resource agents cooperate in a subcontractor likewise when they bid for an operation sequence. In this section the cooperation of the order agents is presented. The use of selfish autonomous agents in the system does not guarantee the attainment of the global optimum. The problem is that if every order agent announces its job independently, the system will not count the possible schedules which arise if we exchange the operation orders of different jobs and, therefore, the global optimum is not guaranteed. However, it is possible for the order agents to make alliances. In the simplest case, if some order agents cooperate, they announce all of their interleaved operation sequences. First, every order agent works alone and tries to find a solution in itself. If it still has time after this it can search for another order agent to make a group of two agents. If they finished the search, they can leave each other, and search for another partner. If the process of making two element groups with all of the possible partners has been completed, it can search for two other agents and make a group of three agents, and so on. If all of the agents cooperate, the system will find the global optimum with one probability if time goes to infinity.

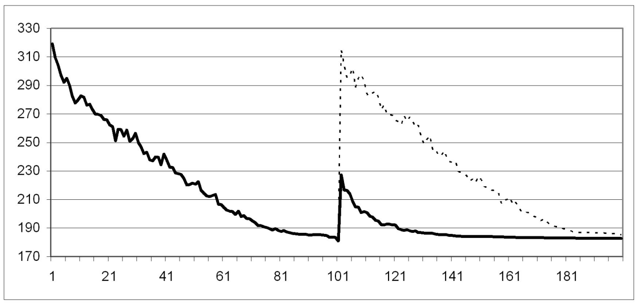

Fig. 2 Machine breakdown (at $t = 100$)

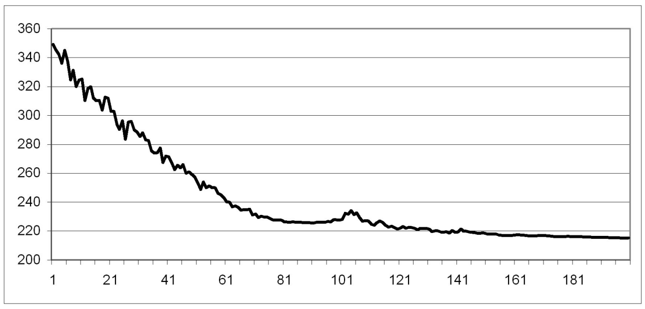

Fig. 3 A job is cancelled (at $t = 100$)

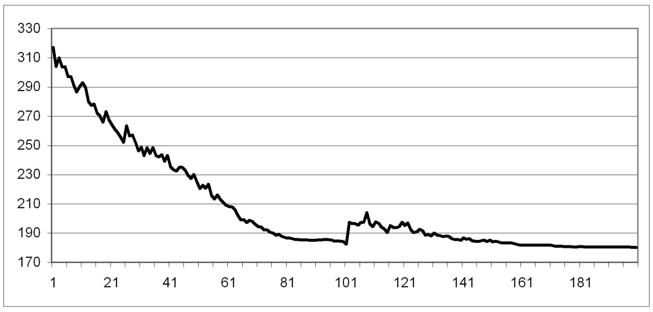

Fig. 4 A new machine is available (at $t = 100$)

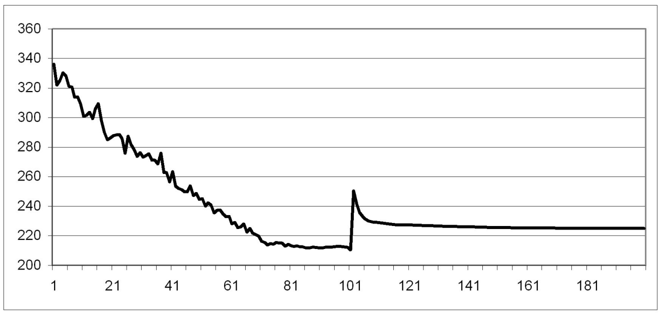

Fig. 5 A new job enters the system (at $t = 100$)

7. Comparative Algorithm Analysis

To illustrate the given method its resource requirement, namely, its time and space complexity, was analysed and compared with other solutions. In the following investigation let us restrict ourselves to the case when there is only one process plan for every job and each job consists of exactly k operations.

The *complete enumeration* or *brute force* algorithm investigates $O((nm)^{nk})$ schedules[5]. Its space complexity is $O(1)$ because it stores only one schedule, namely the best schedule found so far, but if we take a detailed look, it means storing $O(nk)$ data, because for each operation of each job, we must store the machine that will process it. Due to its time complexity this method is unusable in practice.

The classical *branch and bound* algorithm investigates only $O(1)$ schedules in the best case, but in the worst case, it could not decrease the upper bound of the complete enumeration algorithm. It is difficult to predict its exact time requirement in the average case. However, in real-life test cases it proved to speed up the search considerably. Good heuristics can seriously decrease the number of schedules that must be investigated. Regarding storage, its space complexity is the same as the complete enumeration algorithm has.

In the presented solution, if there is no cooperation between the order agents, they compute every sub-schedule (a schedule of only one job) independently, and the whole schedule is built up from the sub-schedules. In each iteration step, the *expected* number of computed schedules is $O(\beta^k)$. Since in case there are n jobs in the system, there are n order agents as well, consequently to compute a whole schedule, the algorithm computes $O(n\beta^k)$ schedules in an iteration. The expected number of iterations we need to get the error below a given $\epsilon > 0$ is an open question; however, the fact that it strongly depends on β is obvious. The experimental results show, that even if $\beta \ll m$, the convergence rate of the algorithm is relatively high. If we allow the order agents to cooperate, let c be the number of cooperating order agents in a group. Cooperation gives the system the ability to find better global solutions, however, it increases the amount of the needed computations as well. In this case the expected number of computed schedules by the given group of c cooperating order agent is: $O((c\beta)^{ck})$, in the worst case.

The space complexity of this solution is $O(1)$ sub-schedules for each order agent, consequently $O(n)$ for the whole system, however, if we take a detailed look, one sub-schedule contains $O(k)$ data only, thus, the whole system needs to store $O(nk)$ data for the schedules only, like the branch and bound. At the same time, we must remember the storage that is required by our learning machinery. There are m machines, thus there are m resource agents in the system. Each resource agent stores the value function of its reinforcement learning which contains estimations about the expected performance measure of each partial job (a tail of the operation sequence of a job) with the earliest starting time t, so each resource agent stores $O(nkt)$ data where t is a time scale parameter. Thus the whole system has $O(mnkt)$ space complexity regarding the reinforcement learning. This storage requirement could be a problem

[5]This estimation is not strict, a better estimation would be $O(n!\, n^{n(k-1)} m^{nk})$, and we can give $\Omega((n!)^k m^{nk})$ as a lower estimation when each machine can process every operation.

in practice if we deal with very large problems. Fortunately, function approximators (such as artificial neural networks) can approximate the value function with as small memory as we have (naturally, the error of the approximation will depend on the amount of memory that they can use).

8. Experimental Results

In order to verify the above algorithm, experiments were initiated and carried out. Although the evaluation and analysis of this method is not over, we present some preliminary results. In the test program, the aim of scheduling was to minimize the maximum completion time (C_{max}). We tested the adaptive features of our solution by confronting it with unexpected events. In Figures 2–5 there are four types of unexpected events: machine breakdown, new machine, new job and job cancellation. The x-axis represents time, while the y-axis the achieved performance measure (which is the total production time that we want to minimize). The figures presented here were made by averaging hundred random samples (runtime results). In this test we used 20 machines with few dozens of jobs and non-cooperative agents. In all cases at time $t = 100$ there were unexpected events. The results show that our system is adaptive, because it did not recompute the whole schedule from scratch, but it tried to use as much information from the past as possible. In Figure 2 the performance measure which would arise if it recomputed the whole schedule is drawn in a broken line.

9. Concluding Remarks

In the paper a market-based distributed production control system with learning and cooperative agents was described. The learning was done by a triple-level learning mechanism. The top level of learning consists of a simulated annealing algorithm, the middle (and the most important) level contained a reinforcement learning system, while the bottom level was done by numerical function approximators, such as artificial neural networks. The proposed system can be used for solving the general dynamic job-shop scheduling problem in a distributed, iterative and robust way. The time and space complexity of the solution were analysed and compared with classical approaches. Some experimental results, too, were presented in the paper.

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. Moreover, it would be promising to generalize the solution to other combinatorial optimization problems.

10. Acknowledgements

This work was partially supported by the Nat. Res. Fund., Hungary, Grant Nos. T034632 and T043547.

References

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

Baker, A. D. (1998). A Survey of Factory Control Algorithms That Can Be Implemented in a Multi-Agent Heterarchy: Dispatching, Scheduling, and Pull. *Journal of Manufacturing Systems*, Vol. 17, No. 4, pp. 297–320.

Brauer, W.; Weiß, G. (1998). Multi-Machine Scheduling - A Multi-Agent Learning Approach. *Proceedings of the 3rd International Conference on Multi-Agent Systems*, Paris, France, pp. 42–48.

Csáji, B. Cs.; Kádár, B.; Monostori, L. (2003). Improving Multi-Agent Based Scheduling by Neurodynamic Programming. Holonic and Multi-Agent Systems for Manufacturing, *Lecture Notes in Computer Science*, Vol. 2744; Lecture Notes in Artificial Intelligence, *HoloMAS 2003*, pp. 110–123.

Hadeli; Valckenaers, P.; Kollingbaum, M.; Van Brussel, H. (2004). Multi-agent coordination and control using stigmergy. *Computers in Industry*, Vol. 53, Elsevier, pp. 75–96.

Lawler, E. L.; Lenstra, J. K.; Rinnooy Kan, A. H. G.; and Shmoys, D. B. (1993). Sequencing and Scheduling: Algorithms and Complexity. In: *Handbooks in Operations Research and Management Science*, Vol. 4: Logistics of Production and Inventory, Elsevier, pp. 445–522.

Márkus, A.; Kis, T.; Váncza, J.; Monostori, L. (1996). A Market Approach to Holonic Manufacturing. *Annals of the CIRP*, Vol. 45(1), pp. 433–436.

Monostori, L.; Kádár, B.; Csáji, B. (2002). The Role of Adaptive Agents in Distributed Manufacturing. *The 4th International Workshop on Emergent Synthesis (IWES02)*, Kobe, Japan, pp. 135–142.

Neumann, J. (1948). The general and logical theory of automata. In: *Papers of John von Neumann on Computing and Computer Theory.* The MIT Press, Cambridge, 1987. pp. 391–431.

Sutton, R. S.; Barto, A. G. (1998). Reinforcement Learning. The MIT Press.

Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind*, Vol. 59, No. 236, pp. 4-30.

Ueda, K.; Márkus, A.; Monostori, L.; Kals, H. J. J.; Arai, T. (2001). Emergent synthesis methodologies for manufacturing. *Annals of the CIRP*, Vol. 50, No. 2, pp. 535–551.

Van Brussel, H.; Jo Wyns; Valckenaers, P.; Bongaerts, L.; Peeters, P. (1998). Reference Architecture for Holonic Manufacturing Systems: PROSA. *Computers in Industry*, Vol. 37, pp. 255–274.

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