Adaptive Sampling Based Large-Scale Stochastic Resource Control*

Balázs Csanád Csáji
Computer and Automation Research Institute,
Hungarian Academy of Sciences,
13–17. Kende utca, H-1111, Budapest, Hungary
e-mail: balazs.csaji@sztaki.hu

László Monostori
Computer and Automation Research Institute,
Hungarian Academy of Sciences; and
Faculty of Mechanical Engineering,
Budapest University of Technology and Economics
e-mail: laszlo.monostori@sztaki.hu

Abstract

We consider closed-loop solutions to stochastic optimization problems of resource allocation type. They concern with the dynamic allocation of reusable resources over time to non-preemtive interconnected tasks with stochastic durations. The aim is to minimize the expected value of a regular performance measure. First, we formulate the problem as a stochastic shortest path problem and argue that our formulation has favorable properties, e.g., it has finite horizon, it is acyclic, thus, all policies are proper, and moreover, the space of control policies can be safely restricted. Then, we propose an iterative solution. Essentially, we apply a reinforcement learning based adaptive sampler to compute a suboptimal control policy. We suggest several approaches to enhance this solution and make it applicable to large-scale problems. The main improvements are: (1) the value function is maintained by feature-based support vector regression; (2) the initial exploration is guided by rollout algorithms; (3) the state space is partitioned by clustering the tasks while keeping the precedence constraints satisfied; (4) the action space is decomposed and, consequently, the number of available actions in a state is decreased; and, finally, (5) we argue that the sampling can be effectively distributed among several processors. The effectiveness of the approach is demonstrated by experimental results on both artificial (benchmark) and real-world (industry related) data.

Introduction

Resource Allocation Problems (RAPs) arise in many divers fields, such as manufacturing production control (e.g., production scheduling), fleet management (e.g., freight transportation), personnel management, scheduling of computer programs (e.g., in massively parallel GRID systems), managing a construction project or controlling a cellular mobile network. In general, they can be described as optimization problems which include the assignment of a finite set of reusable resources to interconnected tasks that have temporal extensions. RAPs have a huge literature, see e.g. (Pinedo 2002), however, in real-world applications the problems are often very large, the environment is uncertain and can even change dynamically. Complexity and uncertainty seriously limit the applicability of classical approaches to RAPs.

Machine learning techniques represent a promising new way to deal with stochastic resource allocation problems. RAPs can often be formulated as *Markov Decision Processes* (MDPs) and can be solved by *Reinforcement Learning* (RL) algorithms. However, most RL methods in their standard forms cannot effectively face large-scale problems. In the paper we first formulate the RAP as a stochastic shortest path problem (a special MDP) and then apply sampling-based fitted Q-learning to approximate the optimal action-value function, and hence, to get a suboptimal resource control policy. We suggest several approaches to make this solution effectively applicable to large-scale problems. Improvements like this are: the value function is maintained by feature-based kernel regression; the initial exploration is guided by limited-lookahead rollout algorithms; the state space is partitioned in a feasible way; the action space is decomposed to decrease the available actions in the states; and the sampling is done in a distributed way. Our solution can show up not only fast convergence properties but, additionally, it scales well with the size of the problem and can quickly adapt to disturbances and changes, such as breakdowns. These properties are demonstrated by numerical experiments on both benchmark and industry related data.

Using RL for scheduling problems was first proposed in (Zhang & Dietterich 1995). They used the TD(λ) method with iterative repair to solve the NASA space shuttle payload processing problem. Since then, a number of papers have been published that suggested using RL for different RAPs. Most of them, however, only investigated centralized open-loop solutions of small-scale deterministic problems. A closed-loop scheduling system was described in (Schneider, Boyan, & Moore 1998) which applied the ROUT and the RTDP algorithms. In (Powell & Van Roy 2004) a formal framework for high-dimensional RAPs were presented and *Approximate Dynamic Programming* (ADP) was used to get a dynamic solution. The application of *Support Vector Regression* (SVR) to maintain value functions was first presented in (Dietterich & Wang 2001). Recently, SVR has been applied to improve iterative repair (local search) strategies of deterministic *Resource Constrained Project Scheduling Problems* (RCPSPs) in (Gersmann & Hammer 2005).

*This research was partially supported by the NKFP Grant No. 2/010/2004 and by the OTKA Grant No. T049481, Hungary.

Scheduling and Resource Control

First, we consider a basic deterministic scheduling problem: an instance of the problem consists of a finite set of *tasks* $\mathcal{T}$ together with a partial ordering $\mathcal{C} \subseteq \mathcal{T} \times \mathcal{T}$ that represents the *precedence constraints* between the tasks. A finite set of *resources* $\mathcal{R}$ is also given with a partial function that defines the *durations* of the tasks depending on the executing resource, $d : \mathcal{T} \times \mathcal{R} \to \mathbb{N}$. The tasks are supposed to be non-preemptive (they may not be interrupted), thus, a *schedule* can be defined as an ordered pair $\langle \sigma, \varrho \rangle$ where $\sigma : \mathcal{T} \to \mathbb{N}$ gives the starting times of the tasks and $\varrho : \mathcal{T} \to \mathcal{R}$ defines which resource will execute which task. A schedule is called *feasible* if the following three properties are satisfied:

(s1) Each resource executes at most one task at a time:
$\neg\exists(r \in \mathcal{R} \wedge u, v \in \mathcal{T}) : \varrho(u) = \varrho(v) = r \wedge$
$\wedge\, \sigma(u) \leq \sigma(v) < \sigma(u) + d(u, r)$

(s2) Every resource can execute the tasks which were assigned to it: $\forall v \in \mathcal{T} : \langle v, \varrho(v) \rangle \in dom(d)$

(s3) The precedence constraints of the tasks are kept:
$\forall \langle u, v \rangle \in \mathcal{C} : \sigma(u) + d(u, \varrho(u)) \leq \sigma(v)$

Note that $dom(d) \subseteq \mathcal{T} \times \mathcal{R}$ denotes the domain set of function d. The set of all feasible schedules is denoted by S, which is supposed to be non-empty. The objective is to minimize a *performance measure* $\kappa : S \to \mathbb{R}$ that usually only depends on the task completion times. If, for example, the completion time of the task $v \in \mathcal{T}$ is denoted by $C(v) = \sigma(v) + d(v, \varrho(v))$, then a commonly used performance measure, which is often called total completion time, can be defined by $C_{max} = \max\{\, C(v) \mid v \in \mathcal{T} \,\}$.

We restrict ourselves to *regular* performance measures, which are monotone in completion times. They have the property that a schedule can be uniquely generated from the order in which the tasks are executed on the resources. As a consequence, S can be safely restricted to a finite number of schedules and, thus, the problem becomes a *combinatorial optimization* problem characterized by $\langle \mathcal{T}, \mathcal{R}, \mathcal{C}, d, \kappa \rangle$.

As a generalization of the job-shop scheduling problem, this problem is *strongly NP-hard* and, moreover, in case of C_{max} there is no good polynomial time approximation of the optimal scheduling algorithm (Williamson *et al.* 1997).

The stochastic variant of the presented problem arises when the task durations are given by independent finite random variables. The randomized version of function d is denoted by D, therefore, for each $\langle v, r \rangle \in dom(D) : D(v, r)$ is a random variable. In the stochastic case, the performance of a schedule is also a random variable and the objective is, usually, to minimize the expected value of the performance.

An *open-loop* solution of a RAP has to make all decisions before the tasks are being executed and it cannot take the actual evolution of the process into account. Regarding a *closed-loop* solution, it is allowed to make the decisions on-line as more data become available. In this case, the resources are continually controlled and, hence, these approaches are called *resource control*. Note that a closed-loop solution is not a simple $\langle \sigma, \varrho \rangle$ pair but, instead, a resource control policy (defined later). In the paper we will only focus on closed-loop solutions and will formulate the RAP as an acyclic stochastic shortest path problem.

Markov Decision Processes

Sequential decision making under uncertainty is often modeled by MDPs. This section contains the basic definitions and some preliminaries. By a (finite, discrete-time, stationary, fully observable) *Markov Decision Process* (MDP) we mean a stochastic system that can be characterized by an 8-tuple $\langle \mathbb{X}, \mathbb{T}, \mathbb{A}, \mathcal{A}, p, g, \alpha, \beta \rangle$, where the components are: $\mathbb{X}$ is a finite set of discrete *states*, $\mathbb{T} \subseteq \mathbb{X}$ is a set of *terminal states*, $\mathbb{A}$ is a finite set of control *actions*. $\mathcal{A} : \mathbb{X} \to \mathcal{P}(\mathbb{A})$ is the *availability function* that renders each state a set of actions available in that state where $\mathcal{P}$ denotes the power set. The *transition function* is given by $p : \mathbb{X} \times \mathbb{A} \to \Delta(\mathbb{X})$ where $\Delta(\mathbb{X})$ is the space of probability distributions over $\mathbb{X}$. Let us denote by $p(y|x, a)$ the probability of arrival at state y after executing action $a \in \mathcal{A}(x)$ in state x. The *immediate cost function* is defined by $g : \mathbb{X} \times \mathbb{A} \times \mathbb{X} \to \mathbb{R}$, where $g(x, a, y)$ is the cost of arrival at state y after taking action $a \in \mathcal{A}(x)$ in state x. We consider discounted MDPs and the *discount rate* is denoted by $\alpha \in [0, 1)$. Finally, $\beta \in \Delta(\mathbb{X})$ determines the *initial probability distribution* of the states.

Theoretically, the terminal states can be treated as states with only one available control action that loops back to them with probability one and zero immediate cost.

A (stationary, randomized, Markov) control *policy* is a function from states to probability distributions over actions, $\pi : \mathbb{X} \to \Delta(\mathbb{A})$. We denote by $\pi(x, a)$ the probability of executing control action a in state x. If policy π reaches a terminal state with probability one, it is called *proper*.

The initial probability distribution β, the transition probabilities p together with a control policy π completely determine the progress of the system in a stochastic sense, namely, it defines a homogeneous Markov chain on $\mathbb{X}$.

The *cost-to-go* or *action-value* function of a control policy is $Q^\pi : \mathbb{X} \times \mathbb{A} \to \mathbb{R}$, where $Q^\pi(x, a)$ gives the expected cumulative [discounted] costs when the system is in state x, it takes control action a and it follows policy π thereafter

$$Q^\pi(x, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \alpha^t G_t^\pi \;\middle|\; X_0 = x, A_0 = a \right], \quad (1)$$

where $G_t^\pi = g(X_t, A_t^\pi, X_{t+1})$, A_t^π is selected according to policy π and the next state X_{t+1} has $p(X_t, A_t^\pi)$ distribution.

A policy $\pi_1 \leq \pi_2$ if and only if $\forall x \in \mathbb{X}, \forall a \in \mathbb{A} : Q^{\pi_1}(x, a) \leq Q^{\pi_2}(x, a)$. A policy is called (uniformly) *optimal* if it is better than or equal to all other control policies. The objective in MDPs is to compute a near-optimal policy.

There always exits at least one optimal (even stationary and deterministic) control policy. Although, there may be many optimal policies, they all share the same unique optimal action-value function, denoted by Q^*. This function must satisfy a (Hamilton-Jacoby-) *Bellman type optimality equation* (Bertsekas 2001) for all $x \in \mathbb{X}$ and $a \in \mathbb{A}$:

$$Q^*(x, a) = \mathbb{E}\Big[g(x, a, Y) + \alpha \min_{B \in \mathcal{A}(Y)} Q^*(Y, B) \Big], \quad (2)$$

where Y is a random variable with $p(x, a)$ distribution.

From an action-value function it is straightforward to get a policy, e.g., by selecting in each state in a greedy way an action producing minimal costs with one-stage lookahead.

For more details on this issue, see (Bertsekas 2001).

Markovian Resource Control

Now, we formulate the resource control problem as an MDP. The actual task durations will be incrementally available during the execution and the decisions will be made on-line.

The states are defined as 6-tuples $x = \langle t, \mathcal{T}_S, \mathcal{T}_F, \sigma, \varrho, \varphi \rangle$, where $t \in \mathbb{N}$ is the actual time, $\mathcal{T}_S \subseteq \mathcal{T}$ is the set of tasks which have been started before time t and $\mathcal{T}_F \subseteq \mathcal{T}_S$ is the set of tasks that have been finished, already. The functions $\sigma : \mathcal{T}_S \to \mathbb{N}$ and $\varrho : \mathcal{T}_S \to \mathcal{R}$, as previously, give the starting times of the tasks and the task-resource assignments. The function $\varphi : \mathcal{T}_F \to \mathbb{N}$ stores the task completion times. The starting state $x_0 = \langle 0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ corresponds to the situation at time zero when none of the tasks are started. The initial probability distribution β renders one to state x_0.

A state x is considered as a a terminal state if $\mathcal{T}_F = \mathcal{T}$. If the system reaches a terminal state (all tasks are finished) then we treat the resource control process completed.

At every time t the system is informed which tasks have been finished, and it can decide which tasks will be started (and on which resources). The action space contains task-resource assignments $a_{vr} \in \mathbb{A}$ $(v \in \mathcal{T}, r \in \mathcal{R})$ and a special a_ω control that corresponds to the action when the system does not start a new task at the present time, it "waits".

The availability function $\mathcal{A}$ allows an a_{vr} action only if r is idle, it can execute v and the preceding tasks of v have been finished, already; action a_ω is allowed only if $\mathcal{T}_S \neq \mathcal{T}_F$.

If an a_{vr} is executed, the system assigns task v to resource r and the resource starts processing the task immediately, t does not increase. The effect of a_ω action is that the system does not take any active action and the current time t increases. This action can result in task completions.

The cost function, for a given κ performance measure (which depends on the task completion times, only), is defined as follows. Let $x, y \in \mathbb{X}$ be two states and φ_x, φ_y the corresponding task completion time functions. Then, the cost function is $\forall a \in \mathcal{A}(x) : g(x, a, y) = \kappa(\varphi_y) - \kappa(\varphi_x)$.

It is easy to see that these MDPs have finite state spaces and their transition graphs are acyclic. Therefore, all policies are proper and the horizon is finite, consequently, the discount rate α can be safely omitted, without risking that the expectation in the Q function is not well-defined. These problems are often called stochastic shortest path problems. Regarding the effective computation of a policy, it is important to try reducing the number of states. If κ is regular (which is almost always the case in practice), then an optimal policy can be found among the policies which start new tasks only at times when another task has been finished or at the initial state. We restrict ourselves to these polices.

Approximate Q-learning

In the previous section we have formulated a resource control task as an acyclic MDP. Now, we face the challenge of finding a good control policy. The paper suggests using a variant of Q-learning to compute a near optimal policy. Like in most RL methods, the aim is to approximate the optimal cost-to-go function rather than directly learning a policy.

We iteratively simulate the possible occurrences of the resource control process with the model, starting from x_0. Each trial produces a sample trajectory that can be described as a sequence of state-action pairs. After each trial, we make updates on the approximated values of the visited pairs.

The one-step Q-learning rule is $Q_{t+1} = TQ_t$, where T is

$$(TQ_t)(x,a) = (1 - \gamma_t(x,a))\, Q_t(x,a) + \\ + \gamma_t(x,a) \left[g(x,a,y) + \alpha \min_{b \in \mathcal{A}(y)} Q_t(y,b) \right], \qquad (3)$$

where y and $g(x, a, y)$ are generated from the pair (x, a) by simulation, that is, according to distribution $p(x, a)$; the coefficients $\gamma_t(x, a)$ are called the *learning rate* and $\gamma_t(x, a) \neq 0$ only if (x, a) was visited during trial t. It is known well (Bertsekas 2001) that if for all x and a: $\sum_{t=1}^{\infty} \gamma_t(x,a) = \infty$ and $\sum_{t=1}^{\infty} \gamma_t^2(x,a) < \infty$, the Q-learning algorithm will converge with probability one to the optimal value function in the case of lookup table representation. Because the problem is acyclic, it is advised to apply *prioritized sweeping*, and perform the backups in an opposite order in which they appeared during simulation, starting from a terminal state.

To balance between *exploration* and *exploitation*, and so to ensure the convergence of Q-learning, we use the standard Boltzmann formula (soft action selection) (Bertsekas 2001).

In systems with large state spaces, the action-value function is usually approximated by a (typically parametric) function. Let us denote the space of action-value functions over $\mathbb{X} \times \mathbb{A}$ by $\mathbb{Q}(\mathbb{X} \times \mathbb{A})$. The method of fitted Q-learning arises when after each trial the action-value function is projected onto a suitable function space $\mathcal{F}$ with a possible error $\epsilon > 0$. The update rule becomes $Q_{t+1} = \Phi\, T\, Q_t$, where Φ denotes a projection operator to function space $\mathcal{F}$. For simplicity, we assume that $\mathcal{F}$ is dense in $\mathbb{Q}(\mathbb{X} \times \mathbb{A})$. We suggest feature-based kernel regression to maintain the Q function, which idea was first applied in (Dietterich & Wang 2001).

Value Estimation by Kernel Regression

A promising solution for compactly representing the action-value function is to apply *Support Vector Regression* (SVR) from statistical learning theory. We suggest using ν-SVRs, proposed by (Schölkopf *et al.* 2000). They have an advantage over classical ϵ-SVRs that through the new hyper-parameter ν, the number of support vectors can be controlled. Additionally, the ϵ parameter can be eliminated.

In general, SVR faces the problem as follows. We are given a sample $\{\langle x_1, y_1 \rangle, \ldots, \langle x_l, y_l \rangle\}$, such that $x_i \in \mathcal{X}$ is an input, $\mathcal{X}$ is a measurable space, and $y_i \in \mathbb{R}$ is the target output. For simplicity, we assume that $\mathcal{X} \subseteq \mathbb{R}^k$. The aim is to find a function $f : \mathcal{X} \to \mathbb{R}$ with a small empirical risk

$$R[f] = \frac{1}{2} \|w\|^2 + C \frac{1}{l} \sum_{i=1}^{l} |f(x_i) - y_i|_\epsilon, \qquad (4)$$

where $|f(x) - y|_\epsilon = \max\{0, |f(x) - y| - \epsilon\}$, w and C are explained later. In ν-SVR the problem can be formulated as finding a function f, which is a linear combination of given ϕ_j functions, by adjusting the weights w_j to minimize

$$\Phi(w, \xi, \xi^*, \epsilon) = \frac{1}{2} \|w\|_2^2 + C \left(\nu\epsilon + \frac{1}{l} \sum_{i=1}^{l} (\xi_i + \xi_i^*) \right), \qquad (5)$$

subject to the constraints on w, b, ξ, ξ^* and ϵ as follows

$$w^T\phi(x_i) + b - y_i \leq \epsilon + \xi_i\,, \tag{6}$$

$$y_i - w^T\phi(x_i) - b \leq \epsilon + \xi_i^*\,, \tag{7}$$

$$\xi_i, \xi_i^* \geq 0, i \in \{1, \ldots, l\}\,, \epsilon \geq 0\,, \tag{8}$$

where $\phi(x) = \langle \phi_1(x), \phi_2(x), \ldots \rangle$. The parameters ξ_i and ξ_i^* are *slack variables* to cope with the otherwise often infeasible problem. Parameter $\epsilon > 0$ defines the amount of deviation that we totally tolerate. Constant $C > 0$ determines the trade-off between the flatness of the regression and the amount up to which deviations larger than ϵ are tolerated.

Using Lagrange multiplier techniques, we can rewrite the regression problem in its dual form and solve it as a quadratic programming problem. The resulting regression estimate then takes a linear approximation form as follows

$$f(x) = \sum_{i=1}^{l} (\alpha_i^* - \alpha_i) K(x_i, x) + b\,, \tag{9}$$

where K is an *inner product kernel* defined by $K(x, y) = \langle\!\langle \phi(x), \phi(y) \rangle\!\rangle$, where $\langle\!\langle \cdot, \cdot \rangle\!\rangle$ denotes an inner product. Note that $\alpha_i, \alpha_i^* \neq 0$, usually, only holds for a small subset of training samples (Schölkopf *et al.* 2000). In our numerical experiments we applied Gaussian type kernels, $K(x, y) = \exp(-\|x - y\|^2 / (2\mu^2))$, where $\mu > 0$ is a parameter.

Then, the action-value function approximation takes the form of (9). Some peculiar features of the training data constitute the inputs of the regression and the target output is the estimated action-value. Basically, the ν-SVR representation should be re-calculated after each trial or batch of trials, however, there are incremental SVR algorithms, as well.

As features, which describe the peculiar characteristics of the state and the selected action, we can use properties as follows. The *expected relative ready time* of each resource (that is zero if the resource is idle); the *estimated future load* of every resource that can be computed from the cumulative expected processing times of the tasks that can be processed by the resource. Regarding the chosen action (task-resource assignment), we can calculate its *expected finish time* and also its *deviation*. The total expected finish time of the tasks, which *succeeds* the selected task, is also a feature.

To efficiently apply SVR, we need a large number of samples both from the relevant part of the state space, which states can appear during the execution of an optimal policy, but also from the irrelevant part, to have some estimations on situations having high costs. Hence, we suggest using two kinds of training samples: (1) a set of examples corresponding to states with high costs. They can be generated, e.g., prior to learning by applying random control; (2) naturally, the important samples are generated during the application of the actual control policy, determined by the current Q function and the Boltzmann formula. Preferably, a few thousand of samples of each type should be maintained.

Rollout Algorithms

There are two issues why we suggest the application of rollout algorithms in the initial stage of learning. Firstly, we need several initial samples before the first application of SVR and these first samples can be generated by simulations guided by a rollout policy. Secondly, the Q-learning method performs quite poorly in practice without any initial guidance. Because Q-learning is an *off-policy* method, it can learn from simulated samples generated by rollout algorithms. This, usually, speeds up the learning considerably.

A *rollout policy* is a limited-lookahead policy with the optimal cost-to-go approximated by the value function of a (usually suboptimal) base policy. In our experiments we applied a greedy policy with respect to the immediate costs as a base policy and the rollout algorithm made one-step lookahead. In scheduling theory, it would be called a dispatching rule. Regarding rollout algorithms, see (Bertsekas 2001).

Decomposition and Partitioning

In large-scale problems the set of available actions in a state can be very large, which can slow down the system significantly. In the current formulation of the RAP, the number of available actions in a state is $O(|\mathcal{T}|\,|\mathcal{R}|)$. Though, even in real-world situations $|\mathcal{R}|$ is, usually, not very large, but $\mathcal{T}$ could contain thousands of tasks. Here, we suggest decomposing the action space for these RAPs in a way as follows. First, the system selects a task, only, and it moves to a new state where this task is fixed and an executing resource should be selected. In that case the state description can be extended by a variable $\tau \in \mathcal{T} \cup \{\emptyset\}$, where $\emptyset$ denotes the case when no task has been selected yet. In every other case the system should select an executing resource for the selected task. Consequently, the new action space is $\mathbb{A} = \mathbb{A}_1 \cup \mathbb{A}_2$, where $\mathbb{A}_1 = \{\, a_v \mid v \in \mathcal{T} \,\} \cup \{a_\omega\}$ and $\mathbb{A}_2 = \{\, a_r \mid r \in \mathcal{R} \,\}$. As a result, we radically decreased the number of available actions, however, the number of possible states is increased. Our experiments show that it is a reasonable trade-off.

The idea of divide-and-conquer is widely used in artificial intelligence and recently it has appeared in the theory of dealing with large-scale MDPs. Partitioning a problem into several smaller subproblems is also often applied to decrease computational complexity in combinatorial optimization. We propose a simple and yet efficient partitioning method for a practically very important class of performance measures. In real-world situations the tasks very often have release dates and due dates, and the performance measure depends on meeting the deadlines, e.g., total lateness, number of tardy tasks. Note that these measures are regular. We denote the functions defining the release and due dates of the tasks by $\rho : \mathcal{T} \to \mathbb{N}$ and $\delta : \mathcal{T} \to \mathbb{N}$, respectively. Then, we can define the *weighted expected slack times* of the tasks, by

$$S(v) = \sum_{r \in \Gamma(v)} w(r)\, \mathbb{E}\left[\delta(v) - \rho(v) + D(v, r)\right], \tag{10}$$

where $\Gamma(v)$ denotes the set of resources that can process task v, formally $\Gamma(v) = \{\, r \in \mathcal{R} \mid \langle v, r \rangle \in dom(D) \,\}$, and $w(r)$ are weights corresponding, e.g., to the estimated workload of the resources, or they can be simply $w(r) = 1/\,|\Gamma(v)|$.

We suggest clustering the tasks in $\mathcal{T}$ into successive disjoint subsets $\mathcal{T}_1, \ldots, \mathcal{T}_k$ according to the expected slack times. If $\mathcal{T}_i$ and $\mathcal{T}_j$ are two clusters and $i < j$, then tasks

in $\mathcal{T}_i$ should have smaller expected slack times than tasks in $\mathcal{T}_j$. However, the precedence constraints must be also taken into account, thus if $\langle u, v\rangle \in \mathcal{C}$, $u \in \mathcal{T}_i$ and $v \in \mathcal{T}_j$, then $i \leq j$ must hold. During learning, first, tasks in $\mathcal{T}_1$ are allocated to resources, only. After some episodes, we fix the allocation policy concerning tasks in $\mathcal{T}_1$ and we start sampling to achieve a good policy for tasks in $\mathcal{T}_2$, and so on.

Naturally, clustering the tasks is a two-edged weapon, making too small clusters can seriously decrease the performance of the best achievable policy, making too large clusters can considerably slow down the system. This technique, however, has several advantages, e.g., (1) it effectively decreases the search space; (2) it also further reduces the number of available actions in the states; and, additionally (3) it speeds up the learning, since the trajectories become smaller (only a small part of tasks is allocated in a trial and, consequently, the variance of the possible costs is also decreased).

Experimental Results

In our experiments we partially applied the LIBSVM free library for support vector machines (Chang & Lin 2001). After centering and scaling the data into interval $[0, 1]$, we used Gaussian kernels and shrinking techniques. We always applied rollout algorithms and action decomposition, but partitioning was only used in tests shown in Figure 4.

In all cases $V^*(x_0) = \min_a Q^*(x_0, a)$ was known (due to the special construction of the test problems) and the error was computed as the average of the $(G_t - V^*(x_0))$ values, where G_t shows the cumulative incurred costs in trial t.

dataset	parallel	1000 iter.	5000 iter.	10000 iter.
sdata	1	8.54 %	5.69 %	3.57 %
edata	1.2	12.37 %	8.03 %	5.26 %
rdata	2	16.14 %	11.41 %	7.14 %
vdata	5	10.18 %	7.73 %	3.49 %
average	**2.3**	**11.81 %**	**8.21 %**	**4.86 %**

Figure 1: benchmark dataset of flexible job-shop problems

We have tested our RL based approach on Hurink's benchmark dataset (Hurink, Jurisch, & Thole 1994). It contains flexible job-shop scheduling problems with 6–30 jobs (30–225 tasks) and 5–15 machines (resources). The performance measure is make-span, thus, the total completion time has to be minimized. These problems are "hard", which means, e.g., that standard dispatching rules or heuristics perform poorly on them. This dataset consists of four subsets, each subset containing about 60 problems. The subsets (sdata, edata, rdata, vdata) differ on the ratio of resource interchangeability, shown in the "parallel" column in the table (Figure 1). The columns with label "x iter." show the average error after carrying out altogether "x" iterations. The execution of 10000 simulated trials (after on the average the system has achieved a solution with less than 5% error) takes only few seconds on an common computer of today.

The best performance on this dataset was achieved by (Mastrolilli & Gambardella 2000). Though, their algorithm performs slightly better than ours, their solution exploits the (unrealistic) specialties of the dataset, e.g., the durations do not depend on the resources; the tasks are linearly ordered in the jobs; each job consists of the same number of tasks. Therefore, the comparison of the solutions is hard.

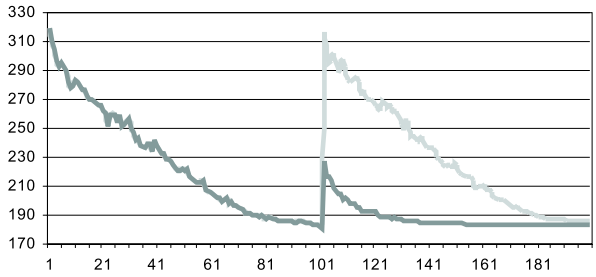

Figure 2: adaptation to changes and disturbances

The adaptive features of the algorithm was tested by confronting it with unexpected events, such as, resource breakdowns and task cancellations. In Figure 2 the horizontal axis represents time, while the vertical one, the achieved performance measure (C_{max}). In these tests a fixed number of 20 resources were used with approximately 100 tasks. In all test cases at time $t = 100$ there were unexpected events. The results (dark gray line) show that the system presented is adaptive, because it did not re-compute the whole solution from scratch. The performance measure which would arise if it recomputed the whole solution is drawn in light gray.

We initiated experiments on a simulated factory by modeling the structure of a real plant producing customized mass-products. We have used randomly generated orders (jobs) with random due dates. The tasks and the process-plans of the jobs, however, covered real products. In this plant the machines require product-type dependent setup times, and another specialty of the plant is that, at some previously given time points, preemptions are allowed. The applied performance measure was to minimize the number of late jobs and an additional secondary performance measure

resources	tasks	1000 iter.	5000 iter.	10000 iter.
16	140	4.26 %	3.28 %	2.45 %
25	280	7.05 %	4.15 %	3.61 %
30	560	7.56 %	5.96 %	4.57 %
50	2000	8.69 %	7.24 %	6.04 %
100	**10000**	**15.07 %**	**10.31 %**	**9.11 %**

Figure 3: industry related simulation experiments

was to minimize the total cumulative lateness, which can be applied to comparing two situations having the same number of late jobs. In Figure 3 the convergence speed (average error) relative to the number of resources and tasks is demonstrated. The workload of the resources was approximately 90%. The results show, that our RL based resource control algorithm can perform efficiently on large-scale problems.

We studied the effectiveness of clustering on industry related data. We considered a system with 60 resources and 1000 tasks distributed among 2–300 jobs (there were approximately 3–4000 precedence constraints). On each cluster we applied 10000 iterations. The computational time in case of using only one cluster was treated as a unit. In Figure 4 the average error and the computational speedup are shown relative to the number (and the size) of the clusters. The results demonstrate that partitioning the search space not only results in greater speed, but often accompanied by better solutions. The latter phenomenon can be explained by the fact that using smaller sample trajectories generates smaller variance that is preferable for learning.

clusters	cluster size	speedup	average error
1	1000	1.00	6.53 %
5	200	1.52	4.13 %
10	100	3.01	3.49 %
20	**50**	**6.37**	**1.95 %**
30	33	7.07	3.02 %
40	25	7.29	3.55 %

Figure 4: the effectiveness of clustering

We have also investigated the parallelization of the method, namely, the speedup of the system relative to the number of processors. The average number of iterations was studied, until the system could reach a solution with less than 5% error on Hurink's dataset. We have treated the average speed of a single processor as a unit (cf. with the data in Figure 1). In Figure 5 the horizontal axis represents the number of applied processors, while the vertical axis shows the relative speedup achieved. We applied two kinds of parallelization: in the first case (dark gray bars), each processor could access a global value function. It means that all of the processors could read and write the same global action-value function, but otherwise, they searched independently. In that case the speedup was almost linear. In the second case (light gray bars), each processor had its own, completely local action-value function and, after the search was finished, these individual functions were combined. The experiments show that the computation of the RL based resource control can be effectively distributed, even if there is not a commonly accessible action-value function available.

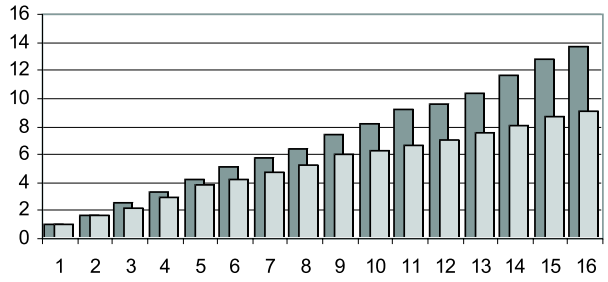

Figure 5: average speedup in case of distributed sampling

Concluding Remarks

The efficient allocation of reusable resources over time is an important problem in many real-world applications. We proposed an adaptive sampling-based closed-loop solution to a stochastic resource control problem. First, the problem was formulated as an MDP. We have highlighted that this formulation has favorable properties. Next, we applied reinforcement learning to approximate a good policy. Several approaches to make the solution applicable to large-scale problems were considered, such as: (1) the value function was maintained by feature-based support vector regression; (2) the initial exploration was guided by rollout algorithms; (3) the set of tasks were clustered in case of due date dependent measures; (4) the action space was decomposed; and, finally, (5) the sampling was done in a distributed way. The effectiveness of the approach was demonstrated by experimental results on both benchmark and industry related data.

References

Bertsekas, D. P. 2001. *Dynamic Programming and Optimal Control*. Athena Scientific, 2nd edition.

Chang, C. C., and Lin, C. J. 2001. LIBSVM: A library for support vector machines. Software available on-line at http://www.csie.ntu.edu.tw/∼cjlin/libsvm.

Dietterich, T. G., and Wang, X. 2001. Batch value function approximation via support vectors. *Advances in Neural Information Processing Systems* 14:1491–1498.

Gersmann, K., and Hammer, B. 2005. Improving iterative repair strategies for scheduling with the SVM. *Neurocomputing* 63:271–292.

Hurink, E.; Jurisch, B.; and Thole, M. 1994. Tabu search for the job shop scheduling problem with multi-purpose machines. *Operations Research Spektrum* 15:205–215.

Mastrolilli, M., and Gambardella, L. M. 2000. Effective neighborhood functions for the flexible job shop problem. *Journal of Scheduling* 3(1):3–20.

Pinedo, M. 2002. *Scheduling: Theory, Algorithms, and Systems*. Prentice-Hall.

Powell, W. B., and Van Roy, B. 2004. *Handbook of Learning and Approximate Dynamic Programming*. IEEE Press, Wiley-Interscience. chapter Approximate Dynamic Programming for High-Dimensional Resource Allocation Problems, 261–283.

Schneider, J.; Boyan, J.; and Moore, A. 1998. Value function based production scheduling. In *Proceedings of the 15th International Conference on Machine Learning*.

Schölkopf, B.; Smola, A.; Williamson, R. C.; and Bartlett, P. L. 2000. New support vector algorithms. *Neural Computation* 12:1207–1245.

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

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