Improving Multi-Agent Based Scheduling by Neurodynamic Programming

Balázs Csanád Csáji, Botond Kádár, László Monostori

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,
E-mail: {csaji, kadar, laszlo.monostori}@sztaki.hu

Abstract. Scheduling problems, e.g., a job-shop scheduling, are classical NP-hard problems. In the paper a two-level adaptation method is proposed to solve the scheduling problem in a dynamically changing and uncertain environment. It is applied to the heterarchical multi-agent architecture developed by Valckenaers et al. Their work is improved by applying machine learning techniques, such as: neurodynamic programming (reinforcement learning + neural networks) and simulated annealing. The paper focuses on manufacturing control, however, a lot of these ideas can be applied to other kinds of decision-making, as well.

1 Introduction

Continuous, steady improvement is a key requirement for manufacturing enterprises that necessitates flat and flexible organizations, life-long learning of employees on the one hand, and information and material processing systems with adaptive, learning abilities on the other hand [11]. The paper outlines an attempt to enhance the performance of an agent-based manufacturing system by using adaptation and machine learning techniques. A two-level adaptation method is proposed to solve the scheduling problem in a dynamically changing and uncertain environment. It is applied to the heterarchical multi-agent architecture developed by Valckenaers et al. [19], which was inspired by food forging ants. First, the general job-shop scheduling problem is described and its complexity is highlighted. A short introduction to the advantages of multi-agent systems is given and the PROSA architecture overviewed including its improvement with mobile agents (ants). Some properties of neurodynamic programming are also described and, finally, an adaptation method is presented which was applied to a multi-agent system. It is capable of solving the job-shop scheduling efficiently and with great fault tolerance. The applicability and the effectiveness of the proposed method are illustrated by the results of experimental runs.

2 Scheduling

Scheduling is the allocation of resources over time to perform a collection of jobs or tasks. The problem of scheduling is important in manufacturing: near-optimal scheduling is a prerequisite for the efficient utilization of resources and hence, for the profitability of the en-

terprise. 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 Job-Shop Scheduling

One of the basic scheduling problems is the so-called job-shop scheduling. We begin by defining the general job-shop problem: suppose that we have n jobs $J = \{J_1, J_2, ..., J_n\}$ to be processed through m machines $M = \{M_1, M_2, ..., M_m\}$. The processing of a job on a machine is called operation. 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 denotes the set of all operations. Every job has a set of operation sequences: the possible process plans, $o: J \rightarrow P(O^*)$. These sequences give the precedence constraints of the operations. The processing time of an operation on a machine is given by $p: M \times O \rightarrow R^+$ which is a partial function. Due dates are given for every job: $d: J \rightarrow R^+$, $d(J_i)$ is the time by which we would like to have J_i completed ideally. It is not easy to state our objectives in scheduling. They are complex, and often conflicting. Generally, we can say that the objective is to produce a schedule that minimizes (or maximizes) a performance measure f, which is usually a function of job completion times (e.g.: maximum completion time, mean flow time, mean tardiness, number of tardy jobs, etc.). Therefore the job-shop scheduling is an optimization problem.

2.2 Complexity of Scheduling

The general job-shop scheduling is a very complex problem. If we suppose, e.g., that every job consists of exactly m operations and each machine can do every operation, then 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 $n = 20$ 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[1]. Except for some strongly restricted special cases[2], the job-shop scheduling is an *NP-hard* optimization problem [14]. It means that no polynomial time algorithm exists that always gives us the *exact* optimal schedule, unless[3] $P = NP$. Moreover, there is no good polynomial time approximation of the optimal scheduling algorithm: Williamson et al. [22] gave the first nontrivial theoretical evidence that shop-scheduling problems are hard to solve even approximately. They showed that if f (the performance measure) is the maximum completion time (C_{max}), then if a (polynomial time) ε–approximator with $\varepsilon < 5/4$ exists for the job-shop problem then it implies that $P = NP$. Because of these properties, the job-shop scheduling has earned the reputation of being notoriously difficult to solve.

---

[1] According to Arthur Eddington (*Mathematical Theory of Relativity*) the number of particles in the known universe is $\approx 3.1495 \cdot 10^{79}$ and $(20!)^{10} \approx 7.2651 \cdot 10^{183}$.

[2] Like single or double machine makespan / maximum lateness problems.

[3] Which is the largest unsolved problem in complexity theory, however we have a strong intuition that the two sets are different.

3 Multi-Agent Systems

Many different approaches such as *integer programming* [15], *Lagrangian relaxation* [4], *branch and bound algorithms* [13] [2], *simulated annealing* [10], *genetic algorithms* [3] [6], *neural networks* [9], *reinforcement learning* [23], etc. have been tried for solving the job-shop scheduling problem. Though, some of them, e.g., integer programming, have elegant mathematics they scale exponentially and are only able to solve highly simplified "toy" instances. Others, such as genetic algorithms, have some promising experimental results but it is not clear how to put these approaches into a distributed (multi-agent) system. As we stated before, our objective is to propose an adaptive extension to a previously designed heterarchical multi-agent architecture. Before we continue our investigation on job-shop scheduling, let us give a short review on multi-agent systems in general and in manufacturing. According to Baker [1], 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. 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. Advantages of these heterarchical multi-agent systems include: *self-configuration*, *scalability*, *fault tolerance*, *emergent behaviour*, and *massive parallelism* [1]. Other authors claim that the advantages of heterarchical architectures also include *reduced complexity*, *increased flexibility*, and *reduced cost* [20], [7], [5], [16], [18] as well. 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.

3.1 The PROSA Architecture

PROSA, developed by Van Brussel et al. [21], is a *holonic* reference architecture for manufacturing systems. It is a starting point for the design and development of multi-agent manufacturing control. The structure of the architecture identifies three kinds of agents (holons), their responsibilities, and the way they interact. The basic architecture consists of three types of basic agents: order agents (internal logistics), product agents (process plans), and resource agents (resource handling). *Resource agents* correspond to physical parts (production resources in the manufacturing system, like: factories, shops, machines, furnaces, conveyors, pipelines, material storages, personnel, etc.), and contain an information processing part that controls the resource. *Product agents* include the process and product knowledge to ensure the correct production. They act as information servers to other agents. *Order agents* represent a task or job in the manufacturing system. They are responsible for performing the assigned work correctly and on time.

3.2 Ant Colonies

Many approaches in multi-agent systems were inspired by heterarchical biological systems such as wasp nests, bird flocks, fish schools, wolf packs, termite hills and ant colonies. Valckenaers et al. [19] presented a coordination and control technique, which was inspired by food-forging ants. They identified the key achievement of this biological example: lim-

ited exposure of the individuals, combined with the emergence of robust and optimized overall system behavior. In their work the environment was agentified and made it part of the solution. The PROSA reference architecture was applied to separate resource, logistic, and process concerns and new type of agents were introduced, called ants, which are mobile and they gather and distribute information in the manufacturing system. Their main assumption is that the agents are much faster than the ironware that they control, and that makes the system capable for forecasting. Agents are faster and, therefore, can emulate the system's behavior several times before the actual decision is made. Scheduling in this system is done based on local decisions. Each order agent sends ants moving downstream in a virtual manner. They gather information about the possible schedules from the resource agents and than they return to the order agent with the information. The order agent chooses a schedule and sends ants to book the needed resources. After that the order agent regularly sends ants to rebook the previously found best schedule, because if the booking is not refreshed, it *evaporates* (like the pheromone in the analogy of food-forging ants) after a while. From time to time, the order agent sends ants to survey the possible new (and better) schedules. If they find a better solution, the order agent books the resources that are needed for the new schedule and the old booking information simply *evaporates*.

3.3 Adaptive Approach

The multi-agent manufacturing system inspired by food-forging ants is very promising, but as to the scheduling problem it can be regarded as a framework only. From what we stated above on the complexity of scheduling, it is clear that even if the mobile agents are much faster than the ironware, they cannot survey every possible schedule. They should select schedules from an initial set for investigation. They should adapt to the current state of the system and make guesses about the presumably good schedules. If the system changes, they should *explore* the new situation. If the system stabilizes, they should *exploit* the information they have gathered. In the paper we propose a two-level adaptation mechanism with these properties. Our below approach applies *neurodynamic programming* (*reinforcement learning + neural networks*) and *simulated annealing*.

4 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-organised (unsupervised) learning* and *reinforcement learning*. Supervised learning is a "cognitive" learning method performed with the support of a teacher: this requires the availability of an adequate set of input-output examples. In the contrary, reinforcement learning is a "behavioral" learning method, which is performed through *interaction* between the learning system and its environment. The fact that this interaction proceeds without a teacher makes reinforcement learning particularly attractive for dynamic situations. We refer to the modern approach of reinforcement learning as *neurodynamic programming*, because dynamic programming provides its theoretical foundation and neural networks provide its learning capability. The operation of a reinforcement learning system is characterized as follows [8]:

1. The environment evolves by probabilistically occupying a finite set of discrete states.

2. For each state there is a finite set of possible actions that may be taken.
3. Every time the learning system takes an action, a certain reward is incurred.
4. Sates are observed, actions are taken, and rewards are incurred at discrete time steps.

The agents' goal is to maximize their *cumulative profit* (reward). This does not mean maximizing immediate gains, but the profit in the long run. In our scheduling system the rewards are computed from the performance measures of the achieved schedules.

4.1 Markov Property

An important concept in reinforcement learning is the *Markov property*. The environment satisfies the Markov property if its state signal compactly summarizes the past without degrading the ability to predict the future. Formally, if we denote the state at time t by s_t, the action taken by a_t and the reward received by r_t, we can compute how the environment might response at time $t+1$ to the action taken at time t. If the system has Markov property, the response of the environment at $t+1$ depends only on the state and action representations at t. A state signal has Markov property if and only if:

$$P(s_{t+1} = s, r_{t+1} = r \mid s_t, a_t) = P(s_{t+1} = s, r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0)$$

for all s, r, s_t, a_t and for all histories s_t, a_t, r_t, …., r_1, s_0, a_0. In this case the environment and the task as a whole are also said to have the Markov property [17]. A reinforcement learning task that satisfies the Markov property is called a *Markov Decision Process (MDP)*. In our work we presuppose that our system states satisfy the Markov property. We may safely assume so, because the only thing that matters if we want to develop an incremental scheduling algorithm, is the current schedule of the resources. It is not relevant how this schedule has evolved.

4.2 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 π: $S \times A \rightarrow [0,1]$ a mapping (partial function) from state and actions to the probability $\pi(s, a)$ of taking action a in state s. A *value* of a state s under a policy π is the expected return when starting in s and following π, thereafter,

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

where γ is a parameter, $0 \leq \gamma \leq 1$, called *discount rate*. (If $\gamma = 0$, the agent is "myopic", and as γ approaches 1, the agent becomes more farsighted.) As in most reinforcement learning work, we attempt to learn the value function of the optimal policy π^*, denoted by V^* rather than directly learning π^*:

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

To learn the value function we can apply the method of *temporal difference learning* known as $TD(\lambda)$, developed by Sutton [17]. A value function $V^{\pi}(s)$ is represented by a function approximator $f(s, w)$, where w is a vector that holds the parameters of the approximation e.g., weights if we use neural networks. 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 step t as:

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

Then we compute the smoothed gradient:

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

Finally, we can update the parameters according to:

$$\Delta w = \alpha \delta_t e_t,$$

where λ is a smoothing parameter that combines the previous gradients with the current gradient e_t, and α is the learning rate. 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 look ahead). After applying this action, we get a reward, and update our value function estimation. This means that the policy continuously changes during the learning process. $TD(\lambda)$ still converges under these conditions [17].

5 Adaptive Scheduling

Now, we can return to the problem of scheduling and present our approach. As we stated before, we propose a two-level adaptation improvement to the framework designed by Valckenaers et al. [19]. Intuitively, the function of the first level is to *route* the mobile agents (ants) so they should not have to investigate every possible schedule. Some presumably good schedules are selected on the basis of information gathered previously. The second level of adaptation controls how *greedy* the system is, more precisely, the ratio of *exploitation* and *exploration*.

5.1 Markov Decision Process

First, we investigate the first level of adaptation. Recall that the order agents are responsible for the processing of the jobs and the resource agents control the ironware. When an order agent sends mobile agents (ants) to gather information from the resource agents about the feasible schedules, they cannot investigate every possible schedule (see the part about the complexity of scheduling) and, therefore, when they reach a decision point, they have to choose only *some* possibilities but they should not choose *all* of them except in case of problems of very small size. We suggest that they use the information gathered by the re-

source agents. The resource agents can learn the possible good schedules of the actual state of the system and only send ants in directions which are presumably good. Every resource agent can learn how to direct ants passing it. To learn the promising scheduling traces we use temporal difference learning. State s that we use for deciding which action is to be taken is the remaining operation sequence of the job that the ant investigates with the earliest start time. Consequently, the state space is $O^* \times R$. A possible action a is the sending of the ant to a resource agent whose machine can process the next operation of the job (the first operation of the remaining operation sequence). Note that it is possible to send an ant to the sender resource agent. $\pi(s, a)$ is the probability of taking action a in the state s if we use policy π. The randomization of ant routing is not only needed for exploration but it also helps us to avoid pathologic behavior.

A difference form the "classical" Markov Decision Processes is that a resource agent is not restricted to send the ants in one direction only. We threat every $\pi(s, a)$ as independent probabilities and it is possible to take two or more actions in a state. This is called *branching*. Thus, we can explore several schedules. Naturally, we have to be very careful about the branching factor we use, otherwise, the system could turn intractable. This way, the ants move from agent to agent and sometimes they branch. When an ant virtually investigated a schedule (its remaining operation sequence is empty), it computes the performance measure of the achieved schedule and then it starts traveling back to the resource agents and the order agent that started it to support feedback. The feedback information is communicated during the back traveling process. The rewards for the $TD(\lambda)$ reinforcement learning mechanism are computed form the achieved performance measures. If an ant arrives back at a resource agent where there was previously a branching, then it waits until all of the other ants arrive back, and than only the ant with the best schedule travels back to the previous resource agent, the others terminate (let us call this *unbranching*). Every time an ant arrives back at a resource agent, it gives a reward according to the schedule that was investigated. The resource agent uses this information to learn the optimal scheduling routes of the current system. Similarly to the authors of [19] we, too, assume that the agents work much faster than the ironware they control, so they can repeat this process several times before the system changes. Thus, from the viewpoint of the agents the system changes very slowly.

5.2 Simulated Annealing

At this point two questions arise, namely, how we should set the branching factor of ants and what happens if the system changes? These questions are answered on the second level of adaptation. A *simulated annealing* mechanism [12] controls the "temperature" of the system, which is the expected branching number of ants at a resource agent. To keep things as simple as possible, let us suppose that this branching factor is equal regarding all of the resource agents. Consequently, we can define the "temperature" by $T = E\{\pi(s, a)\}$ for a state s. Note that the expected number of branches is not necessarily an integer. If the system changes, then we raise the temperate (the expected number of branches) to force the system to *explore* the new situation. If the system stabilizes, we slowly cool the system down by lowering the temperature to achieve that the agents *exploit* the information they have gathered. We can change the temperature by simply rescaling the probabilities.

Although it is a difficult question in itself and investigating that was not the main goal of our research, some ideas are provided on how the system has been changed. If an order agent books a new schedule, the situation changes, but in that case the order agent can

inform the system (the other order agents) about it. Machine(s) may break down or a new machine(s) may become available, but in these cases the resource agents, which control the changed machines, should inform the system about the change(s). Because our learning algorithm can work in relatively slowly changing non-stationary environments, not recognizing system changes does not disastrously effect the whole system. But if we can recognize a change, we can force the system to make more exploration.

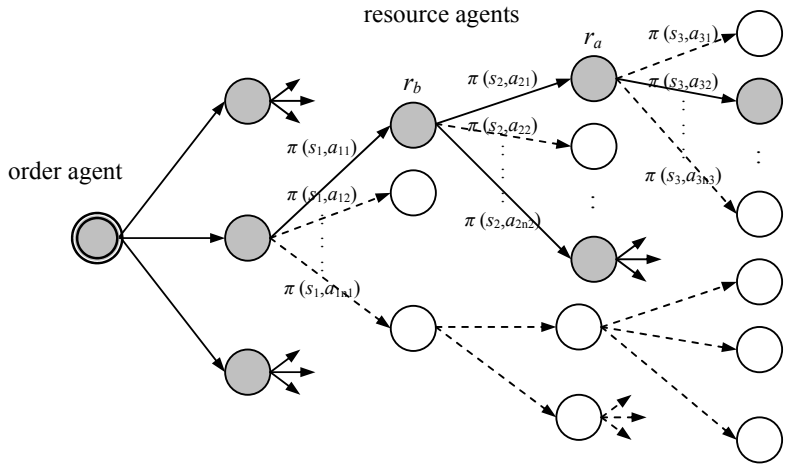

Fig. 1. This figure shows a possible way of ants in the system. They all start from an order agent and they visit resource agents only which are colored gray. At an agent they continue their way in a direction with probability $\pi(s, a)$. Resource agent r_a does a normal routing but at r_b there is a branching

6 Experimental Results

In order to verify the above algorithm, experiments were initiated and carried out. Although the evaluation and analysis of this method is far from being over, some preliminary results are presented here. In the test program, the aim of scheduling was to minimize the maximum completion time (C_{max}). We have extended the job-shop scheduling problem by considering also the distances between machines (note that the distance between two machines can be infinite as well). We did not use any function approximator in our program, because the state space (of reinforcement learning) was relatively small, however, for large problems one cannot avoid using an approximator (e.g., neural networks are candidates for such an approximator). The methodology of testing was as follows: first, we made an advance schedule by scheduling random orders with simple dispatching rules. We did it to generate a non-empty schedule. Then we generated order agents to given jobs and tested how quickly they can find a good (or the optimal) schedule with different parameter values. We also compared our results with other scheduling algorithms (such as branch and bound). We have tested the exploration / exploitation features of our algorithm, as well. Since our solution is a randomized algorithm, we have generated our results, which we

present here, by averaging several (usually a hundred) runtime outcomes. We also give the (standard) deviation of our samples.

Table 1. This table shows a comparison between different scheduling algorithms. It shows how many steps they required (step = virtually putting an operation to a machine) and the performance that they achieved. The BF is the "Brute Force" algorithm (it tries every possible variation). It is only shown because from that column, one can see the size of search space and the optimal performance. The BB column shows the data for the "Branch and Bound" algorithm. It always finds the optimal schedule. The ND(X) columns show the data for our "Neurodynamic" algorithm with branching factor X. The number of iterations is also shown. We have generated the data for our algorithm by averaging several runtime results (for each type of branching factor)

	BF	**BB**	**ND(2.0)**	**ND(2.0)**	**ND(2.0)**	**ND(3.0)**	**ND(4.0)**
Iterations:	-	-	10	50	100	35	25
Steps:	$1.5 \cdot 10^{10}$	$5.6 \cdot 10^{6}$	$1.2 \cdot 10^{4}$	$1.3 \cdot 10^{5}$	$3.8 \cdot 10^{5}$	$5.4 \cdot 10^{5}$	$1.5 \cdot 10^{6}$
Sample Size:	-	-	100	100	100	100	100
(Mean) Performance:	262	262	291.5	268.3	267.1	263.4	262.5
(Standard) Deviation:	-	-	14.36	7.43	7.21	3.19	2.73

While there are many ways to compute decision probabilities from the previously learnt return estimations, in our test program we used a particular one, called *Boltzmann distribution*. We modified the original formula to let the system have higher (than one) expected branching numbers. Here, we briefly present, how we computed the probabilities of sending ants from the learnt value estimations: suppose that a resource agent r has estimation for the expected return value if it sends an ant with the remaining operation sequence γ to the resource agent a at time t. Let us denote that by $V_a(\gamma, t)$. Let us denote the branching factor by β. If our aim is to *maximize* the achieved return values, we can compute the probabilities of sending ants from these data in the following way (note that this formula is slightly more complicated when we consider distances as well):

$$P(\text{sending an ant to } a) = \min\left\{1, \quad \frac{e^{V_a(\gamma,t)/\tau}}{\sum_b e^{V_b(\gamma,t)/\tau}}\beta\right\}, \tag{1}$$

where τ is the so-called *Boltzmann temperature*. High temperatures cause the actions to be all (nearly) equiprobable. Low temperature causes a greater difference in selection probability for actions that differ in their value estimation. As we can see, high Boltzmann temperature also causes the system to make more explorations. The formula that we should use if we want to *minimize* the achieved return values, like in the case of minimizing C_{max}, can be easily computed from (1). As we stated before, in our test program we also consider

distances (transportation time) as well between machines. One can further modify the Boltzmann distribution by adding transportation time to the formula. If we denote the transportation time between the resource r and a by $d(r, a)$, we can modify the formula (1) by adding that value to the time parameter. Thus, $V_a(\gamma, t)$ will be $V_a(\gamma, t + d(r, a))$. One of our future plans is to schedule transportation resources too (such as AGVs, trolleys, conveyor belts). Considering distances between resources (machines) is a step into that direction.

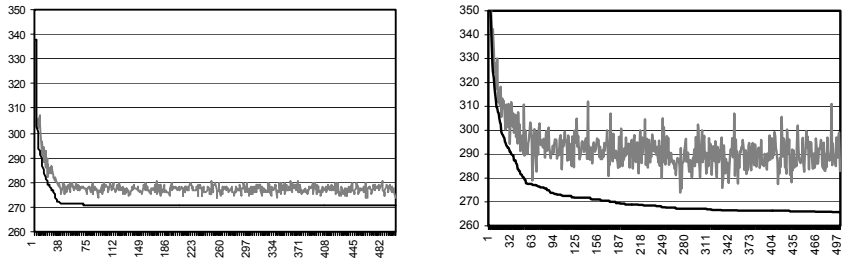

Fig. 2. This figure shows a comparison between different Boltzmann temperatures. Axis x shows the number of iterations. The black line (down) shows the best-achieved performance measure, the gray one (up) the actually achieved performance. The algorithm showed on the left-hand side used 0.3 as Boltzmann temperature. That made the algorithm converge faster, but the probability of choking in a local minimum is also higher. The algorithm on the right-hand side had 0.9 as Boltzmann temperature and that made it take more risk (it made more explorations). It converged slower but it did not choked in local minimums. (Both algorithms had 2.0 as branching factor and 0.1 as learning rate.) We have generated these figures by averaging results of twenty runs

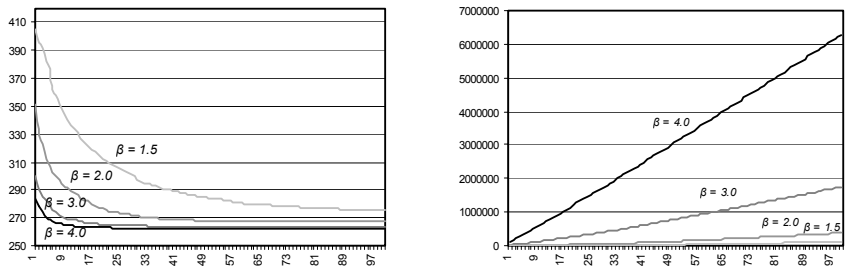

Fig. 3. This figure shows a comparison between different branching factors. Axis x shows the number of iterations. In the left figure the best-achieved performance measure is shown. The diagram on the right-hand side shows the number of required steps. The algorithm with the lightest color had 1.5 as branching factor, the darkest one had 4.0. The other two had 2.0 and 3.0. Every algorithm had 0.1 as learning rate and 0.4 as Boltzmann temperature. We have generated these figures by averaging results of a hundred runs

7 Concluding Remarks

In the paper a two-level adaptation mechanism was presented to improve the performance of a previously developed multi-agent based manufacturing control system. The system can learn the routing of mobile agents, called ants, which gather information about the possible schedules of a particular job. In the proposed system the resource agents route the ants. Temporal difference learning is used for learning the directions (probabilities) of the possible good schedules. The branching factor of ants is controlled by simulated annealing. The main advantages of the algorithm are as follows:

1. *Parallel*: the computation of schedule is massively parallel: all of the order and resource agents work independently.
2. *Robust*: the system can handle changes: for example, the constant learning rate makes the system "forget" old information. Using neural networks as function approximator also makes the system more stable, because one of the well-known properties of neural networks is that they have great fault tolerance.
3. *Iterative*: the algorithm can support us (sub-optimal) solutions in settable time slices. It permanently tries to improve the previously found best schedule.
4. *Flexible*: one can change the working of the algorithm by changing the *branching temperature* or the *Boltzmann temperature* or even the *learning rate*.
5. *Fast convergence*: the solution statistically finds the optimal schedule much faster than the "classical" *branch and bound algorithm*.

Some experimental results, too, were presented in the paper. Future research direction could be to improving the system by localizing the temperate of the system in a way that every agent should be to control its own temperature. We also plan to schedule transportation and storage resources. The presented method will be tuned with different function approximators, as well. The approach to the reinforcement-learning problem is slightly different from the classical one, because in the system an agent can take more than one action at once (it can send ants to more than one direction).It is supposed that this feature does not affect the learning abilities of temporal difference learning, however, this may need further investigations.

Acknowledgements

This work was partially supported by the National Research Foundation, Hungary, Grant No. T034632, No. T043547 and the 5th Framework GROWTH Project MPA (G1RD-CT2000-00298).

References

1. Baker, A. D.: 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, 297-320, (1998).
2. Baker, J. R., and McMahon, G. B.: Scheduling the General Job-Shop, Management Science, May, 31(5), 594-498 (1985).

3. Davis, L.: Job Shop Scheduling with Genetic Algorithms. Int'l Conf. on Genetic Algorithms and Their Application, Pittsburgh, PA, 136-140 (1985).
4. Della Croce, F., Menga, G., Tadei, R., Cavalotto, M., and Petri, L.: Cellular Control of Manufacturing Systems, Eurapean Journal of Operational Research, Vol. 69, 489-509 (1993).
5. Duffie, N. A., Piper, R. S., Humphrey, B. J., Hartwick, J. P.: Hierarchical and Non-Hierarchical Manufacturing Cell Control with Dynamic Part-Oriented Scheduling. Proc. of the 14th North American Mfg. Research Conf., Minneapolis, 504-507 (1986).
6. Falkenauer E., Bouffouix S.: A Genetic Algorithm for Job Shop. IEEE Int'l Conf. on Robotics and Automation, Sacramento, CA, Apr. 9-11, 824-829 (1991).
7. Hatvany, J.: Intelligence and Cooperation in Heterarchic Manufacturing Systems. Robotics & Computer-Integrated Manufacturing, Vol. 2, No. 2, 101-104 (1985).
8. Haykin, S.: Neural Networks, A Comprehensive Foundation. 2nd Edition, Prentice Hall (1999).
9. Jain, A. S., Meeran, S.: Job-Shop Scheduling Using Neural Networks. International Journal of Production Research, 36(5), 1249-1272 (1998).
10. Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon C.: Optimisation by Simulation Annealing: An Experimental Evaluation. Part II. (Graph Coloring and Number Partitioning), Operations Research, Vol. 39, 378-406 (1991).
11. Kádár, B., Monostori, L.: Approaches to Increase the Performance of Agent-Based Production Systems. Engineering of Intelligent Systems, Lecture Notes in AI 2070, IEA/AIE-01, 14th International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, June 4 - 7, 2001, Budapest, Hungary, Springer, 612-621 (2001).
12. Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P.: Optimisation by Simulated Annealing. Science, Number 4598, 671-681 (1983).
13. Lageweg, B. J., Lenstra, J. K., and Rinnooy Kan, A. H. G.: Job-Shop Scheduling by Implicit Enumeration, Management Science, December, 24(4), 441-450 (1977).
14. Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G., and Shmoys, D. B.: Sequencing and Scheduling: Algorithms and Complexity. In: Handbooks in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory, S. C. Graves et al., editors, Elsevier, 445-522 (1993).
15. Manne, A. S.: On Job-Shop Scheduling Problem, Operations Research, Vol. 8. 219-223 (1960).
16. Monostori, L., Márkus, A., Van Brussel, H., Westkämper, E.: Machine Learning Approaches to Manufacturing, CIRP Annals, Vol. 45, No. 2, 675-712 (1996).
17. Sutton, R. S., Barto, A. G.: Reinforcement Learning. The MIT Press (1998).
18. Ueda, K.; Márkus, A.; Monostori, L.; Kals, H.J.J.; Arai, T.: Emergent synthesis methodologies for manufacturing, Annals of the CIRP, Vol. 50, No. 2, 535-551, (2001).
19. Valckenaers, P., Van Brussel, H., Kollingbaum, M., and Bochmann O.: Multi-Agent Coordination and Control Using Stigmergy Applied to Manufacturing Control. European Agent Systems Summer School, Prague, 317-334 (2001).
20. Vámos, T.: Co-operative Systems - An Evolutionary Perspective. IEEE Continuous Systems Magazine, Vol. 3, No. 2, 9-14 (1983).
21. Van Brussel, H., Jo Wyns, Valckenaers, P., Bongaerts, L., Peeters, P.: Reference Architecture for Holonic Manufacturing Systems: PROSA. Computers in Industry, Vol. 37, 255-274 (1998).
22. Williamson, D. P., Hall, L. A., Hoogeveen, J. A., Hurkens, C. A. J., Lenstra, J. K., Sevastjanov, S. V., and Shmoys, D. B.: Short Shop Schedules. Operations Research, 45(2): 288-294, March-April (1997).
23. Zhang, W., Dietterich, T. G.: A Reinforcement Learning Approach to Job-Shop Scheduling. In: Proceedings of Fourteens International Joint Conference on Artificial Intelligence, 1114-1120 (1995).