Proceedings of 2008 ISFA
2008 International Symposium on Flexible Automation
Atlanta, GA, USA June 23-26, 2008

ISFA2008U_120

A SCHEDULING REINFORCEMENT LEARNING ALGORITHM

Amit Gil, Helman Stern, Yael Edan, and Uri Kartoun
Department of Industrial Engineering and Management
Ben-Gurion University of the Negev, Beer Sheva 84105, Israel
{gilami, Helman, yael, Kartoun}@bgu.ac.il

ABSTRACT

This paper presents a scheduling reinforcement learning algorithm designed for the execution of complex tasks. The algorithm presented here addresses the high-level learning task of scheduling a single transfer agent (a robot arm) through a set of sub-tasks in a sequence that will achieve optimal task execution times. In lieu of fixed inter-process job transfers, the robot allows the flexibility of job movements at any point in time. Execution of a complex task was demonstrated using a Motoman UP-6 six degree of freedom fixed-arm robot, applied to a toast making system. The algorithm addressed the scheduling of a sequence of toast transitions with the objective of minimal completion time. Experiments performed examined the trade-off between exploration of the state-space and exploitation of the information already gathered, and its effects on the algorithm's performance. Comparison of the suggested algorithm to the Monte-Carlo method and a random search method demonstrated the superiority of the algorithm over a wide range of learning conditions. The results were assessed against the optimal solution obtained by Branch and Bound.

Index Terms **- hierarchical reinforcement learning, scheduling, robot learning.**

INTRODUCTION

To introduce robots into flexible manufacturing systems, it is necessary for them to perform in unpredictable and large-scale environments. Since it is impossible to model all environments and task conditions, robots must perform independently and learn how to interact with the world surrounding them. One approach to learning is reinforcement learning (RL), an unsupervised learning method [1], [2]. In RL the robot acts in a process guided by reinforcements from the environment, indicating how well it is performing the required task. The basic notion is that an agent (robot) observes its current state (s_t) and chooses an action from a set of possible actions (a_t), with the objective of achieving a defined goal. Throughout the process, the agent receives reinforcements from the environment (r_t), indicating how well it is performing the required task. The robot's goal is to optimize system responses by minimizing a cost function suited for the desired task [3].

RL is an attractive alternative for programming autonomous systems (agents), as it allows the agent to learn behaviors on the basis of sparse, delayed reward signals provided only when the agent reaches desired goals [4]. Furthermore, RL do not require training examples as it creates its own examples during the learning process. However, standard RL methods do not scale well for larger, more complex tasks. Although RL has many advantages over other learning methods, and has been used in many robotic applications, it has several drawbacks: (i) expensive computability, (ii) long learning times (until convergence to an optimal policy) in large state-action spaces, and (iii) the fact that it allows only one goal in the learning task. These drawbacks present significant difficulties when dealing with complex tasks consisting of several sub-tasks.

One promising approach to scaling up RL is hierarchical reinforcement learning (HRL) [1], [4]. Low-level policies, which emit the actual actions, solve only parts of the overall task. Higher-level policies solve the overall task, considering only a few abstract, high-level observations and actions. This reduces each level's search space and facilitates temporal credit assignment [5]. Moreover, HRL allows a learning process to consist of more than one goal.

The notion of HRL presented in this paper has been applied for various problems. An application of HRL to the problem of negotiating obstacles with a quadruped robot is based on a two-level hierarchical decomposition of the task [6]. In a *Hierarchical Assignment of Subgoals to Subpolicies LEarning* algorithm (HASSLE) [5] the high-level policies select the next sub-goal to be reached by a lower-lever policy, in addition to defining sub-goals represented as desired abstract observations which cluster raw input data. Testing of the HASSLE algorithm in a navigation task in a simulated "office" grid world showed that HASSLE outperformed standard RL methods in

deterministic and stochastic tasks, and learned significantly faster.

Similarly to HRL, Compositional *Q*-Learning (*CQ-L*) [7] is a modular approach to learning to perform composite tasks made up of several elemental tasks by RL. Successful applications include a simulated two-linked manipulator required to drive the manipulator from an arbitrary starting arm configuration to one where its end-effector is brought to a fixed destination [8].

This paper presents a reinforcement learning scheduling algorithm developed to provide an optimal sequence of sub-tasks. The algorithm was evaluated by applying it to a test-bed learning application - a toast making system. The complex task of multi-toast making is addressed here by decomposing it into a two-level learning hierarchy to be solved by HRL. The high-level consists of learning the desired sequence of execution of basic sub-tasks and the low-level consists of learning how to perform each of the sub-tasks required. In this application a high-level scheduling algorithm is used to generate a sequence of toast transitions through the system stations, to achieve completion of toast making in minimum time. It is assumed here that the solutions of the low-level tasks are known *a-priori*. The system, however, has no *a-priori* knowledge regarding the efficient sequencing policies of the toast transitions and it learns this knowledge from experience.

The sequencing problem is an extension of a flow-shop problem with one transport robot. This problem is known to be NP hard, therefore classical scheduling algorithms can not be expected to reach optimal solutions in reasonable time and heuristic or approximate methods are required. RL was selected since it can learn to solve complex problems in reasonable time.

The paper is organized as follows: Section II presents the new scheduling algorithm. The test-bed learning application is described in section III, followed by experimental results and conclusions presented in sections IV and V, respectively.

SCHEDULING ALGORITHM

RL Scheduling Algorithms

The goal of scheduling is defined as finding the best sequence of different activities (*e.g.*, processing operations, goods delivery) given a set of constraints imposed by the real world processes [9]. Several authors have used RL to solve scheduling problems. A RL-based algorithm was designed using *Q*-learning [1] to give a quasi-optimal solution to the m-machine flow-shop scheduling problem [9]. The goal was to find an appropriate sequence of jobs that minimizes the sum of machining idle times. Results indicated that the RL-scheduler was able to find close-to-optimal solutions. An adaptive method of rules selection for dynamic job-shop scheduling was developed in [10]. A *Q*-learning agent performed dynamic scheduling based on information provided by the scheduling system. The goal was to minimize mean tardiness. The *Q*-learning algorithm showed superiority over most of the conventional rules compared. An intelligent agent-based scheduling system, consisting of a RL agent and a simulation model was developed and tested on a classic scheduling problem, the Economic Lot Scheduling Problem [11]. This problem refers to the production of multiple parts on a single machine, with the restriction that no two parts may be produced at the same time. The agent's goal was to minimize total production costs, through selection of a job sequence and batch size. The agent successfully identified optimal operating policies for a real production facility.

A great advantage of solving scheduling problems with RL is the relatively easy modeling of the problem. There is no need for predefining desirable or undesirable intermediate states, which is very hard to do in such problems. All that must to be done is to construct a fairly simple rewarding policy (*e.g.*, higher reward for shorter completion times) and the algorithm will supply a solution.

The RL Multi-Toast Algorithm

The proposed algorithm, described in pseudo code in Appendix A, was developed to solve a difficult version of the flow-shop sequencing problem. The complication arises because there is a single job transfer agent (a robot arm) with a capacity of one, and non-zero empty robot return times. The objective is to schedule the transfer of jobs (sliced bread pieces) through a sequence of operations (toasting, buttering, etc.) so as to minimize the total completion time of all jobs.

Problem states, denoted as $s_t \in S$, are defined as system's overall state at time step t. Problem actions, $a_t \in A$, traversing the system from state to state, are defined in accordance to the specific problem. A value Q, associated with a state-action pair, (s_t,a_t), represents how "good" it is to perform a specific action a_t when at state s_t. A learning episode is defined as a finite sequence of time steps, during which the agent traverses from the starting state to the goal state. A learning session is a whole learning process, containing a series of N learning episodes.

Action selection in the algorithm is performed using an adaptive ε-greedy method [2], in which the agent behaves greedily by selecting an action according to Max Q most of the time, and with a small probability ε, selects a random action instead. The probability ε starts with a relatively high value, and is adaptively reduced over time. At the beginning of the learning session, when the agent has not gathered much information, a large value of ε encourages exploration of the state-space by allowing more random actions. As the learning session progresses, and the agent has more information about the environment, the probability ε decreases, reducing the number of random actions, and allowing the agent to exploit the information already gathered and perform better. Equation (1) shows the change in ε as a function of the number of episodes:

$$\varepsilon = \frac{1}{n^{\beta}} \qquad (1)$$

where ε specifies the probability, [0,1], of a random action being chosen, n is the number of episodes already performed during the current learning session, and β is a positive parameter specifying how fast ε will exponentially decrease towards zero, meaning how greedily the algorithm will act as the learning proceeds.

To solve the scheduling problem there is a need to address the state-action value updates not only step by step, but also taking into account the sequence of steps as a whole. The reason is that the policy's performance can only be evaluated at the end of the learning episode, when the task completion time is known. This is also the reason why standard RL algorithms, such as *Q*-learning, updating value estimates on a step-to-step basis, can not be applied here. Hence, the algorithm includes two updating methods.

The *first* method is performed after each step, similar to the SARSA control algorithm [2]. The difference is that because of the characteristics of the scheduling problem, there is no way of evaluating whether a certain action taken is good or not, from the narrow perspective of a single step. Therefore, it is impossible to assign an effective instantaneous reward. Equation (2) describes a one step update of the state-action values:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[\gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \qquad (2)$$

where $Q(s_t,a_t)$ is the value of performing action a_t when the system is in state s_t at time step t, α is the learning rate which controls how much weight is given to the new Q estimate, as opposed to the old one, and γ is the discount rate, determining the present value of future rewards.

The *second* update method is performed at the end of the learning episode n, when it is possible to evaluate the performance of the policy used. At this stage there is an update of all the steps in the episode sequence, by multiplying their Q values with a factor (the reward) indicating how good the last episode was. Two reward factor calculations can be used, both assigning higher values to lower task completion times. A *type A* reward factor receives a value of 1 if task completion time T_n, achieved at episode n, was less than or equal to the best time found so far. Otherwise, the factor will be smaller than 1, proportional to the difference between the current episode's time and the best time achieved so far. This way, Q values of states visited during a "good" sequence remain the same, while Q values of states included in worse sequences are decreased. The reward factor is calculated according to Eq. (3):

$$R_n = \begin{cases} 1, & if\ T_n \le T^*_{n-1} \\ 1/(T_n - T^*_{n-1} + a) + b, & if\ T_n > T^*_{n-1} \end{cases} \qquad (3)$$
$$Where\ T^*_{n-1} = \underset{k=0,\ldots,n-1}{Min}\{T_k\}$$

where R_n is the reward factor at episode n, T_n is the time achieved at the current episode n, and T^*_{n-1} is the best time achieved up to episode *n-1*. The parameters a and b are used to adjust the reward factor to exhibit the desired values. The *type B* reward factor is simply set to be $1/T_n$, achieving the desired inverse proportion between the factor and the task completion time T_n.

CASE STUDY

Experimental Setup

The system (Fig. 1) is comprised of six stations, two of which are processing stations (toaster and butter applier), and a transferring "agent", a fixed-arm six degrees of freedom Motoman UP-6 robot (Fig. 2), advancing the toasts through the system, one toast at a time. The processing and transfer times are predetermined (Table I). The system allows the user to choose the number of toasts in a session (one to four).

The objective of the system is to produce butter covered toasts from raw bread slices as fast as possible. It is assumed that low-level task times achieved via optimal robot motions are known, such that only the high-level scheduling task must be solved.

TABLE I
Processing and Transition Times

Action	Time (sec)	Action	Time (sec)
Toasting process	90	4 to 2	20
Buttering process	90	4 to 3	30
Station 1 to station 2	20	4 to 5	40
1 to 3	30	5 to 1	30
2 to 1	30	5 to 2	20
2 to 3	30	5 to 3	30
2 to 4	20	5 to 6	20
2 to 5	30	6 to 1	30
3 to 1	30	6 to 2	20
3 to 4	30	6 to 3	20
3 to 5	50	6 to 4	30
4 to 1	30		

* Transition combinations not specified are not applicable

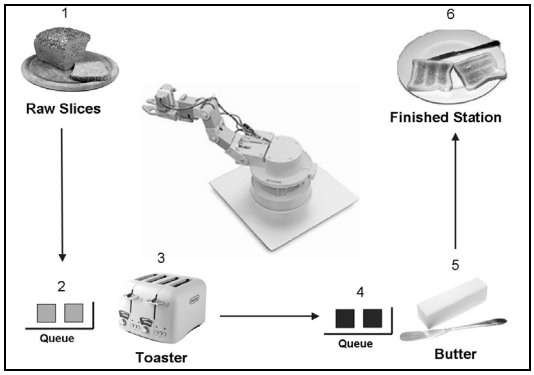

Figure 1: General scheme of the multi-toast making system.

Learning the high-level sequencing task is performed off-line using an event-based Matlab simulation. On-line fixed-arm robot motions are performed only after the simulation supplied the desired sequence. The use of simulation allows fast learning, since real robot manipulations are extremely time-consuming. Furthermore, the simulation constituted a convenient and powerful tool for analyzing the performance of the algorithm, by conducting various virtual experiments off-line.

The simulated system consists of six stations: 1- raw slices of bread, 2- a queue in front of toaster, 3- toaster (with a capacity of one slice), 4- a queue in front of butter applier, 5- butter applier (butter can be applied to only one slice at a time), and 6- finished plate. Each toast has to go through all of the stations in the specified order, except for the queue stations which are used only when needed. The model receives robot transition times and machine processing times as input data, and a robot schedule of toast moves is sought to minimize total completion time.

Task Definition

The objective of the learning task is to generate a sequence of toast transitions through the system stations that will minimize total completion time of the desired number of toasts. The sequencing problem presented by the system resembles flow-shop scheduling problems, which require sequencing of parts (jobs) with different processing times through a set of machines. The difference is that here the parts have identical processing times, and the requirement is to sequence the

transitions between the system stations. Furthermore, the problem is more complex due to the requirement of a transfer agent with limited capacity (movement of only one toast at a time). Other unique characteristics are: (i) toasts can be transferred only one at a time, because there is only one robotic arm, and toast transitions take time, (ii) there are dynamic queues (unlimited) in front of the processing stations, which are not a part of the technological path, and are used only when a station is busy, and (iii) the fact that robot's arm movements while empty must be considered (duration depends on the source and target locations).

To solve the sequencing problem using the algorithm, it is formulated as a RL problem. The system's overall state at time step t, denoted as $s_t \in S$, is defined by the current locations of the toasts. For example, in the three toast problem, states can be: (1,1,1), (3,1,1), (3,2,1), (5,2,1) etc. A solution is a specific sequence of toast transfers: "move toast 1 to its next station, move toast 3 to its next station, move toast 1 to its next station, move toast 3…;" presented as a vector: [1,3,1,3,2,1,3,2,2].

The goal state of the learning task is state (6,6,6), when all the toasts have reached the finished plate. In this context, it is important to understand the distinction between the goal state of the toasting system, which is, as noted, (6,6,6), and the goal of the learning task, which is to find the sequence of steps that would achieve state (6,6,6) as fast as possible.

An action at step t is denoted as $a_t \in A$, where A is the action space of all possible actions. It is to be noted that the action space is state dependent. The execution of an action constitutes the advancement of a toast to its next station in the processing sequence. For example, at state (3,2,1) there are two possible actions: (i) advance toast number one from station 3 to station 5, and (ii) advance toast number three from station 1 to station 2. Toast number two cannot be moved to station 3 because the station is still loaded with another toast.

As aforementioned, learning is achieved by updating the state-action values both during the learning episode, step by step, and at the end of the episode, according to the performance. A learning episode starts from the state where all the slices lie on the raw slice plate, and ends when the last slice arrives to the serving plate, toasted and covered with butter. A step is the transition from one system state to another.

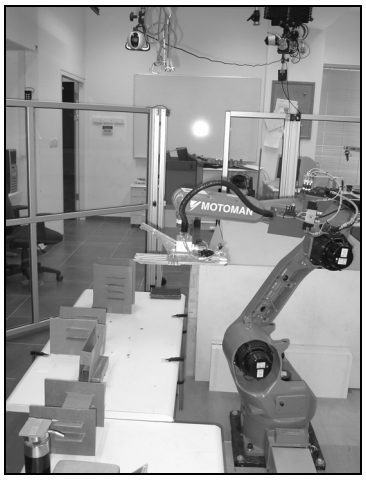

Figure 2: Experimental setup.

Performance was evaluated using the following measures: (i) distance from the optimal solution of the scheduling problem described, (ii) number of learning episodes required to reach convergence, and (iii) percentage of learning sessions reaching the optimal solution, in a set of learning tasks. Convergence in this aspect means not only reaching the optimal solution, but eventually come to the understanding it is the best solution possible, and continuing to produce it from now onward. As described in Eq. (4), the episode at which convergence occurs, n^*, is defined as the episode after which there is no change in the performance over an interval of [n, $n+k$], meaning the algorithm supplied the same solution for k consecutive episodes.

$$\begin{aligned} &n^* = Min\{n\} \text{ over all intervals } [n, n+k] \\ &\text{such that } \Delta(n+j) = 0, \forall\ j = 0,\ldots,k \\ &\qquad n = 0,\ldots,N-k \end{aligned} \tag{4}$$

where $\Delta(n) = T_{n+1} - T_n$ is the change in performance at episode n and N is the length of the learning session.

Experiments

The algorithm's performance was tested on two problems: a "3-toast" problem and a "4-toast" problem. The 3-toast problem allows better understanding of the algorithm characteristics, and its optimal solution can be found in reasonable time and compared to the solution reached by the algorithm. The 4-toast problem is closer to real-world problems, having a significantly larger state-space.

Based on the Matlab simulation, three simulated experiments were conducted, each one twice, once using a *type A* reward factor and once using a *type B* reward factor. The *first* experiment was designed to show the convergence of the suggested scheduling algorithm to a solution, and to examine the differences in performance using various action selection parameters. The parameter β was varied during the analysis, due to its presumed significant influence on the algorithm's performance. Sensitivity analysis was performed using four different β values (1, 1.2, 1.5 and 1.7). Each value was evaluated by performing 10 simulation runs, each run containing 100 learning sessions with 200 learning episodes. For each simulation run the average number of episodes until convergence and the percentage of sessions reaching optimum were measured. To evaluate the performance of the algorithm in solving the 3-toast problem, a *second* experiment was set. The experiment was designed to compare the performance of the scheduling algorithm to the Monte-Carlo method [2] and a random search algorithm. Monte-Carlo (MC) methods are ways of solving the RL problem based on averaging sample returns. It is only upon the completion of an episode that value estimates and policies are changed, thus incremental in an episode-by-episode sense, but not in a step-by-step sense. Here the Q values are simply the average rewards received after visits to the states during the episodes. The reward for a specific episode is set to be $1/T_n$, assigning a higher reward for lower times, and is accumulated and averaged for each state-action pair encountered during the episode. The action selection is similar to the one used for the scheduling algorithm. When applying the random search, actions are chosen with equal probability, using a uniform distribution. Comparisons were made using a range of ten learning session lengths (from 15 episode learning sessions to 60 episode ones, in increments of

5). Each length was evaluated by performing 10 simulation runs, each run containing 100 sessions of that length, and counting the number of sessions reaching the optimal solution at each run. Each session length was evaluated three times, once for each method (scheduling algorithm, MC method and random search). In terms of equation (1), for the random search $\beta = 0$ ($\varepsilon = 1$ for all n) was used, while for the scheduling algorithm and MC, using the adaptive ε-greedy method, a value of $\beta = 1$ was used. To examine the performance in solving the more complex 4-toast problem, a *third* experiment was conducted. Similarly, this experiment consisted of learning sessions of eight different lengths (50 to 400 learning episodes, in increments of 50), each evaluated using 10 simulation runs for each method. Here $\beta = 0.5$ was used for the scheduling algorithm and MC method.

EXPERIMENTAL RESULTS

The best solution produced by the algorithm for the 3-toast problem using the moving and process times shown in Table I, achieved a total completion time of 700 seconds for all the three toasts (Fig. 3). This value was verified by a Branch and Bound general search technique as being optimal. The solution achieved was to schedule the toasts advancing as follows (from left to right): [1,2,1,2,1,2,3,2,3,3].

Examining the influence of the action selection parameters on the algorithm's performance (Fig. 4), revealed that when using a relatively small β ($\beta = 1$) the algorithm reaches the optimal solution with very high percentage of success, yet with the cost of a high number of episodes required for convergence. As β increases, the percentage of success in reaching the optimal solution decreases, but fewer episodes are required to achieve convergence.

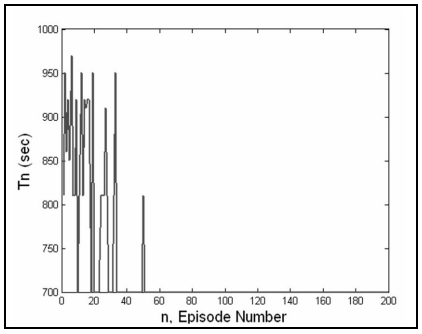

Figure 3: Convergence to the scheduling problem's solution.

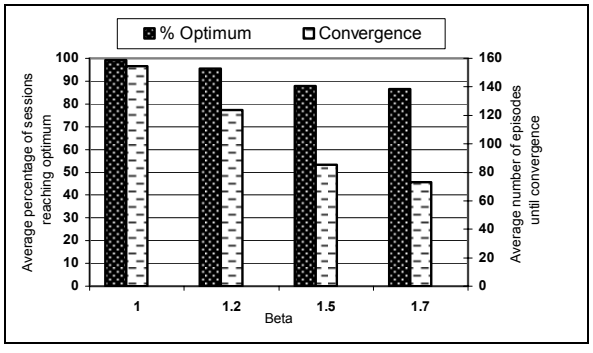

Figure 4: Action selection sensitivity analysis, *type A* factor.

The reason for this behavior lies in the action selection method. When using a small β, the probability of choosing a random action remains relatively high when the episode number rises. The action selection rule allows much exploration, resulting in a higher percentage of sessions reaching the optimal solution, but also a higher number of episodes required for convergence. When using larger values of β, the probability of choosing a random action decreases very fast, resulting in less exploration of the environment, and more exploitation of the information already gathered. This allows much faster convergence to a solution, but not necessarily the optimal one. Generally, the use of a *type A* reward factor achieves fast learning and good results in a low number of episodes, while when using the *type B* reward factor the algorithm requires more episodes in order to achieve good results, but ultimately outperforms the *type A* results.

Comparison of the suggested algorithm to the MC method and random search in solving the 3-toast problem (Fig. 5), demonstrates the superiority of the algorithm over a wide range of learning conditions (20-60 episode sessions), reaching up to a 37 percent difference from the MC method and a 16 percent difference from the random search method. This implies that fewer episodes are needed for the same success percentage.

For 15 episode sessions, the agent does not achieve enough interaction with the environment, therefore does not have sufficient information, and its Q values does not reflect the real state-action values. At this state, the algorithm acts *de facto* as a random search method, hence the similarity in the performances. From 20 to 50 episodes, the agent obtains sufficient information on the environment, allowing it to update the Q values to be closer to the real ones, and reach optimality more times than the other methods.

For the 55-60 episode sessions the algorithm performance becomes steady, reaching approximately 95 percent of success. The reason for this phenomenon lies in the action selection method. After 55 episodes, ε is very close to zero (0.018), implying there are almost no random actions taken. At this state the agent is acting greedily according to its current knowledge, and the algorithm converges to a solution. At about 5 percent of the sessions, the agent reaches 55 episodes with insufficient knowledge, leading it to converge to a local optimum solution.

The MC method acts similarly, but converges to worse solutions. For the random search on the other hand, more episodes means a greater chance of reaching the optimal solution in one of them, hence its success percentage continues to rise along the full range.

Comparison of performance for the 4-toast problem (Fig. 6) reveals the superiority of the algorithm in reaching the best possible result of 900 seconds in the higher range of session lengths (300-400). Due to its complexity, the 4-toast problem requires much more learning episodes to reach the best solution, and the algorithm requires more experience to reach good results. Here, as opposed to the 3-toast problem, the MC method shows better results than the random search.

CONCLUSIONS

A RL-based scheduling algorithm is used for learning high-level policies in a decomposed complex task, where there is a need to sequence the execution of a set of sub-tasks in order to optimize a target function. In such learning tasks, where there is

a need to consider the sequence of steps as a whole, standard step-by-step update RL methods can not be applied.

As implied by the experimental results, the algorithm produces good results, outperforming both the Monte-Carlo and the random search when allowed sufficient experience.

The algorithm can be adjusted to achieve desired performance. In applications where it is critical to achieve a high percentage of success reaching an optimal solution, lower values of β for the adaptive ε-greedy selection probability will achieve the desired effect, but result in longer learning times. If rapid convergence is required at the expense of certainty in reaching the optimum, higher values of β will achieve the proper results. The scheduling algorithm can be adjusted to suit other scheduling problems, especially those with job transfer agents.

Future work includes evaluating the scheduling algorithm's performance in stochastic environments (stochastic processing and robot transfer times), applying learning methods to perform the basic low-level sub-tasks required, and adding human-robot collaboration aspects to the system, allowing acceleration of the learning process. Furthermore, other soft computing methods such as genetic algorithms can be applied to solve the sequencing problem. Undergoing research aims to integrate the low and high level learning into one framework.

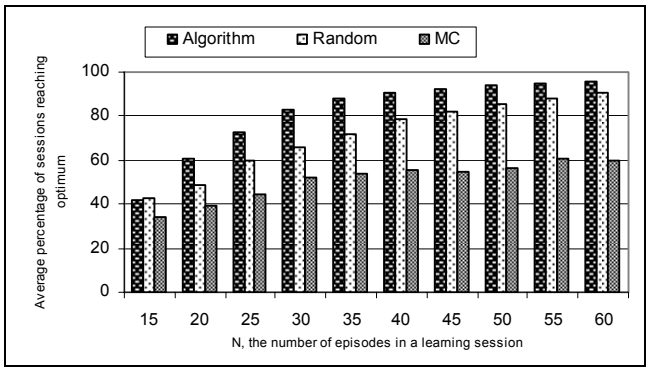

Figure 5: Performance comparison - 3-toast problem, *type A* factor.

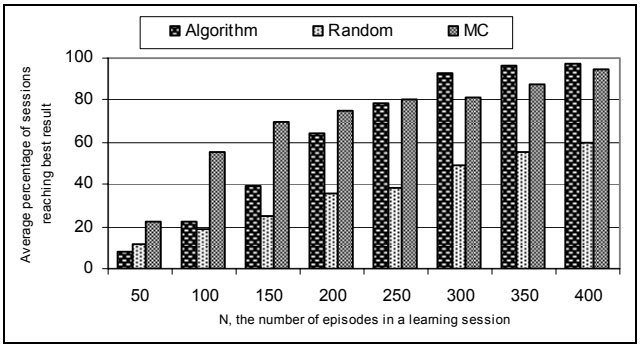

Figure 6: Performance comparison - 4-toast problem, *type B* factor.

ACKNOWLEDGEMENTS

This research was partially supported by the Paul Ivanier Center for Robotics Research and Production Management, and by the Rabbi W. Gunther Plaut Chair in Manufacturing Engineering, Ben-Gurion University of the Negev.

APPENDIX A - RL algorithm pseudo code

Initialize $Q(s,a)=1$ for a learning session
Repeat (for each learning episode *n*):
  Initialize state s_t as starting state (all toasts at starting station)
  Repeat (for each step *t* of episode):
    Take action a_t, observe next state s_{t+1}
    Choose a_{t+1} for s_{t+1} using a certain rule (*e.g.*, ε-greedy)
    $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[\gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$
    $s_t \leftarrow s_{t+1}; a_t \leftarrow a_{t+1}$
  Until a stopping condition (*i.e.*, reached goal state)
  Calculate R_n (*type A* or *type B*)
  For all (s_t,a_t) visited during the episode:
    $Q(s_t,a_t) \leftarrow R_n * Q(s_t,a_t)$
Until a stopping condition (desired number of learning episodes)

REFERENCES

[1] C. J. C. H. Watkins, *Learning from Delayed Rewards*, Ph.D. Dissertation, Cambridge University, 1989.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Cambridge, MA: MIT Press, 1998.

[3] C. Ribeiro, "Reinforcement learning agents," *Artificial Intelligence Review*, 2002, vol. 17, no. 3, pp. 223-250.

[4] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," *Journal of Artificial Intelligence Research*, 1999, vol. 13, pp. 227-303.

[5] B. Bakker and J. Schmidhuber, "Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization," *Proc. of the 8th Conf. on Intelligent Autonomous Systems*, 2004, pp. 438-445, Amsterdam, The Netherlands.

[6] L. Honglak, S. Yirong, Y. Chin-Han, S. Gurjeet, and Y. N. Andrew, "Quadruped robot obstacle negotiation via reinforcement learning," *Proc. of the 2006 IEEE Conf. on Robotics and Automation*, 2006, Orlando, Florida.

[7] S. Singh, "Transfer of learning by composing solutions of elemental sequential tasks," *Machine Learning*, 1992, vol. 8, pp. 323-339.

[8] C. K. Tham and R. W. Prager, "A modular Q-learning architecture for manipulator task decomposition," *Int. Conf. on Machine Learning*, 1994.

[9] P. Stefan, "Flow-Shop scheduling based on reinforcement learning algorithm," *Production Systems and Information Engineering*, 2003, vol. 1, pp. 83-90.

[10] Y. Wei and M. Zhao, "Composite rules selection using reinforcement learning for dynamic job-shop scheduling," *Proc. of the 2004 IEEE Conf. on Robotics, Automation and Mechatronics*, 2004, Singapore.

[11] D. C. Creighton and S. Nahavandi, "The application of a reinforcement learning agent to a multi-product manufacturing facility," *IEEE Conf. on Industrial Technology*, 2002, pp. 1229-1234, Bangkok, Thailand.