Fast permutation learning

Tony Wauters[1], Katja Verbeeck[1] and Patrick De Causmaecker[2], and Greet Vanden Berghe[1]

[1] CODeS group, KAHO Sint-Lieven, Gent, Belgium
[2] CODeS group, K.U. Leuven campus Kortrijk, Kortrijk, Belgium

Abstract. Permutations occur in a great variety of optimization problems, such as routing, scheduling and assignment problems. The present paper introduces the use of learning automata for the online learning of good quality permutations. Several centralized and decentralized methods using individual and common rewards are presented. The performance, memory requirement and scalability of the presented methods is analyzed. Results on well known benchmark problems show interesting properties. It is also demonstrated how these techniques are successfully applied to multi-project scheduling problems.

1 Introduction

The process of creating a permutation, i.e. arrangement of objects or values into a particular order, occurs often in combinatorial optimization problems. The permutations can represent a full or a partial solution of such problems. Typical examples can be found in routing and scheduling. The traveling salesman problem (TSP) for instance, aims at finding a tour of minimum distance through a number of cities. A solution can be represented by a permutation, which defines the order in which to visit the cities. Many solutions for scheduling problems also contain some permutation representation. A solution for the permutation flow shop scheduling problem (PFSP) is such an example. In the PSFP a number of jobs have to be sequenced in order to be processed on a predefined number of resources. All these problems have a search space that is exponential in the number of inputs n (cities, jobs, ...). Due to the very nature of permutations there are at least $n!$ different solutions. An objective function, which represents the quality of the solutions, has to be optimized. If a solution to a problem can be represented by a permutation, then the objective function states how good the permutation is.

In fact, we can imagine the following general problem (see Figure 1): given a permutation π, a function f can give a value for that permutation $f(\pi)$. Function f can be the optimization problem under study, and it is assumed that the function is not known. It is a black box. Since all values can be normalized, we can assume that $f(\pi) \in [0, 1]$, with a value $f(\pi) = 0$ meaning the worst permutation and $f(\pi) = 1$ the best or optimal permutation.

In the present contribution, this general permutation problem is tackled using simple reinforcement learning devices, called Learning Automata (LA). It is

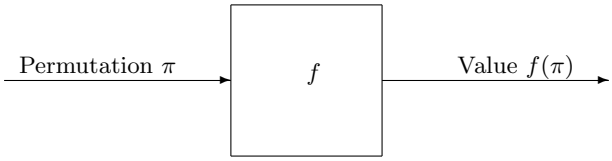

Fig. 1. General permutation problem seen as a black box function.

shown how these methods are capable of learning permutations of good quality ($f(\pi)$ close to 1) without the use of problem specific information or domain knowledge. However, it is not the goal to outperform existing optimization methods for these problems, but only to show the strength of LA for online permutation learning.

The present paper is structured as follows. Section 2 shows related work on the learning of permutations and the use of learning automata for solving optimization problems. Section 3 gives a small overview on learning automata. In Section 4 different categories of permutation functions and their properties are discussed. Section 5 presents some centralized and decentralized methods based on learning automata for the online learning of permutations. In Section 6 the presented methods are analyzed on some well known benchmarks, and a successful application to project scheduling is demonstrated. A conclusion and final remarks are given in Section 7.

2 Related work

Population based incremental learning (PBIL) [1] is a method for solving optimization problems, which is related to genetic algorithms. It however maintains a real-valued probability vector for generating solutions. PBIL is very similar to a cooperative system of finite learning automata where the learning automata choose their actions independently and update with a common reward. COMET [2] incorporates probabilistic modeling in conjunction with fast search algorithms for application to combinatorial optimization problems. The method tries to capture inter-parameter dependencies by creating a tree-shaped probabilistic network. PermELearn [7] is an online algorithm for learning permutations. The approach makes use of a doubly stochastic[3] weight matrix to represent estimations of the permutations, together with exponential weights and an iterative procedure to restore double stochasticity. [15] introduces a method using multiple learning automata to cooperatively find good quality schedules for the multi-mode resource-constrained project scheduling problem (MRCPSP). A common reward, based on the makespan of the scheduling solution is used. [14] present a method using learning automata combined with a dispersion game for solving

[3] A matrix is doubly stochastic if all its elements are nonnegative, all the rows sum to 1, and all the columns sum to 1.

the decentralized resource-constrained multi project scheduling problem (DR-CMPSP). To date the method belongs to the state-of-the-art for this problem[4].

3 Learning Automata

Learning Automata (LA) [11, 13] are simple reinforcement learning components for adaptive decision making in unknown environments. An LA operates in a feedback loop with its environment and receives feedback (reward or punishment) for the actions taken. A single learning automaton maintains a probability vector p over its actions, which it updates according to a reinforcement scheme. Several reinforcement schemes with varying convergence properties are available in the literature. Examples of linear reinforcement schemes are linear reward-penalty, linear reward-inaction and linear reward-ϵ-penalty. The philosophy of these schemes is to increase the probability of selecting an action in the event of success and to decrease it when the response is a failure. The general update scheme is given by:

$$\begin{aligned} p_m(t+1) &= p_m(t) + \alpha_{reward}(1-\beta(t))(1-p_m(t)) \\ &\quad - \alpha_{penalty}\beta(t)p_m(t) \\ &\quad \text{if } a_m \text{ is the action taken at time } t \end{aligned} \tag{1}$$

$$\begin{aligned} p_j(t+1) &= p_j(t) - \alpha_{reward}(1-\beta(t))p_j(t) \\ &\quad + \alpha_{penalty}\beta(t)[(r-1)^{-1} - p_j(t)] \\ &\quad \text{if } a_j \neq a_m \end{aligned} \tag{2}$$

With $p_i(t)$ the probability of selecting action i at time step t. The constants α_{reward} and $\alpha_{penalty}$ are the reward and penalty parameters. When $\alpha_{reward} = \alpha_{penalty}$, the algorithm is referred to as linear reward-penalty (L_{R-P}), when $\alpha_{penalty} = 0$, it is referred to as linear reward-inaction (L_{R-I}) and when $\alpha_{penalty}$ is small compared to α_{reward}, it is called linear reward-ϵ-penalty ($L_{R-\epsilon P}$). $\beta(t)$ is the reward received by the reinforcement signal for an action taken at time step t. r is the number of actions. In the present paper, learning automata with finite actions and linear reinforcement schemes will be used. More specifically we will only use the linear reward-inaction update scheme, because it has nice theoretical convergence results.

4 Permutation Functions

A permutation function, i.e. a function mapping permutations to values, can take several forms. Most of them are highly non-linear. The most straightforward function is one that gives a value to each individual position. Take for example

[4] Multi project scheduling problem library: http://www.mpsplib.com ; accessed on September: 23, 2011

an assignment problem where a matrix defines the cost for assigning an agent to a task. A permutation, where task i at position j is performed by agent j, is a possible solution representation for this problem.

	T0	T1	T2	T3
A0	3	4	1	3
A1	3	2	3	1
A2	3	4	2	2
A3	2	3	4	4

Table 1. Cost matrix for an example assignment problem of size $n = 4$

The assignment problem with the cost matrix from Table 1 results in a permutation function as shown in Figure 2. The total cost for each permutation is plotted. The search space has an optimal solution $[2, 1, 3, 0]$ with a total cost of 7. This solution denotes that task 2 is performed by agent 0, task 1 is performed by agent 1, task 3 is performed by agent 2 and task 0 is performed by agent 3.

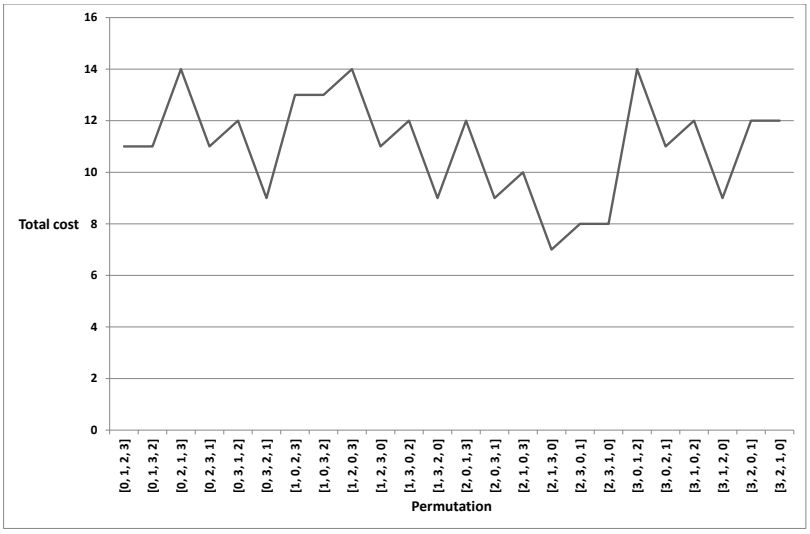

Fig. 2. Permutation function for an example assignment problem.

For many problems using a permutation representation the total cost or value is determined by the adjacent elements. For example, if element A is directly

followed by element B in the permutation, there is a cost of c_{AB}. These costs can be symmetric or asymmetric. In this category of permutation functions we can make a distinction between cyclic and non-cyclic functions. A typical example where the permutation function is based on adjacency costs and is also cyclic is a TSP.

To summarize, we distinguish the following default permutation function categories:

- **individual position**
- **adjacent positions (cyclic and non-cyclic)**

Many permutation functions use or combine elements from these default categories.

Some optimization problems have additional constraints on the permutations. For example, one can have precedence constraints, imposing that one element must occur before or after another element. Examples include the sequential ordering problem (SOP) which is a TSP with precedence constraints) and project scheduling problems. Yet another additional constraint can be that several elements must be adjacent to each other and form groups. One can incorporate these additional constraints by adding a high penalty to the cost value of the permutation.

5 Learning Permutations

In order to learn permutations of good quality one or more learning components are put in a feedback loop with the permutation evaluation function f (i.e. the environment), as is shown in Figure 3. The rest of this section describes a number of centralized and decentralized approaches for performing this learning task.

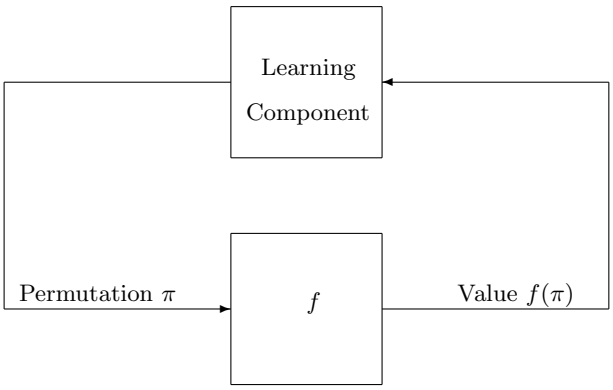

Fig. 3. Learning component in a feedback loop with the permutation learning problem

5.1 Naive approach

A naive and centralized approach (Figure 4) to learning permutations using learning automata would be to assign one action per permutation. This results in a total of $n!$ actions, which is impractical for larger n both with respect to calculation time and memory usage.

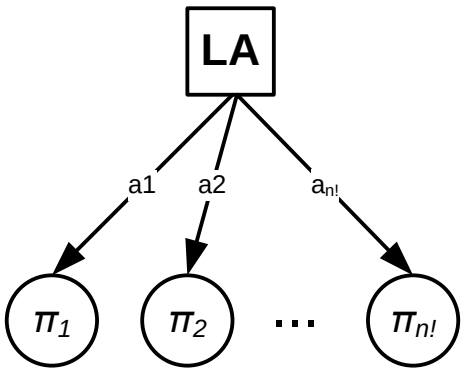

Fig. 4. A naive approach, one LA with one action per permutation.

5.2 Hierarchical approach

A better approach would be to divide the learning task among different learning automata, more specifically a tree of learning automata, called a hierarchical learning automaton [12]. An example of such a hierarchical learning automaton for $n = 3$ is shown in Figure 5. An LA at depth $d \in \{1, 2, \ldots, n\}$ in the tree is responsible for choosing the element at position d in the permutation. Each LA at depth d has $n + 1 - d$ actions, excluding all the actions chosen in the LA in the path from this LA to the root of the tree. The advantage of this hierarchical approach is that each individual LA has a smaller action space (maximum n). There is also a drawback. In case of a large exploration, the whole search tree is visited in the worst case, which results in $\sum_{d=1}^{n} \frac{n!}{d!}$ LAs. Since all action selection probabilities for each LA have to be stored, this can be very memory intensive.

5.3 Probability matrix approach

To deal with the large memory requirements of the hierarchical approach we developed an approach with a compact representation. Similar to the method in [7], we use a doubly stochastic matrix P with n rows and n columns. The column and row sums are always 1. P_{ij} is the probability for element i to be on position j in the permutation. The approach works as follows:

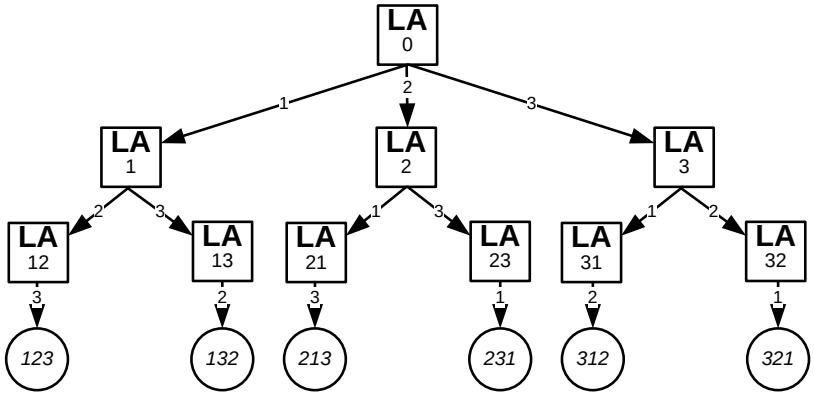

Fig. 5. An example of a hierarchical learning automaton for $n = 3$.

1. generate a uniform doubly stochastic matrix P with: $\forall_{i=1..n} \forall_{j=1..n} P_{ij} = \frac{1}{n}$
2. select a permutation π using P
3. retrieve a reward $r = f(\pi)$ for the selected permutation
4. update P using reward r
5. repeat from step 2 until some stopping condition is met.

These steps are now described in more detail.

Selecting a permutation from P: Several methods can be used to select a permutation from a doubly stochastic matrix. A first method is to uniformly select a row i and then use a probabilistic selection (e.g. roulette wheel selection) on this row for determining on which position j we have to put element i in the permutation. After that we reduce the matrix by removing row i and column j. Then we normalize the remaining rows, and repeat the process until all rows (and also all columns) have been selected once. We then have a complete permutation. Another permutation selection method is based on entropy. The method is similar to the explanation above, but the next row is now selected based on minimum entropy.

$$\underset{i}{argmin}\ H = -\sum_{j=1..n} P_{ij} log\left(P_{ij}\right)$$

Updating P with reward r: The probability matrix P is updated with an LA update scheme for each row i, and the selected action is determined by j. After updating all rows, the matrix P remains doubly stochastic, which is not the case in the PermELearn algorithm [7].

5.4 Decentralized approach

The following method is similar to the 'probability matrix method', but uses a more decentralized approach. Each position in the permutation is determined

by an individual agent. An agent employs a learning automaton for choosing its individual position from the full set of positions. Thus, there are n agents (LA) with n actions each, resulting in the same memory footprint as the 'probability matrix approach'. The agents play a dispersion game [6] for constructing a permutation. In a dispersion game, the number of agents is equal to the number of actions. In order to form a permutation, all agents need to select a distinct action so that the assignment of actions to agents is maximally dispersed. For example, if three agents select the following distinct actions: agent 1 selects position 2, agent 2 selects position 3 and agent 3 selects position 1, then the permutation becomes $[3, 1, 2]$.

A Basic Simple Strategy (BSS) was introduced in [6], allowing agents to select maximally dispersed actions in a logarithmic (in function of the number of agents) number of rounds, where a naive approach would be exponential. BSS does not incorporate the agents' preferences, it uses uniform selection. To take the agents' preferences into account, we introduce a probabilistic version of BSS, which we call Probabilistic Basic Simple Strategy (PBBS). The PBBS works as follows. Given an outcome $o \in O$ (selected actions for all agents), and the set of all actions A, an agent using the PBBS will:

- select action a with probability 1 in the next round, if the number of agents selecting action a in outcome o is 1 ($n_a^o = 1$).
- select an action from the **probabilistic** distribution over actions $a' \in A$ for which $n_{a'}^o \neq 1$, otherwise.

The probabilistic distribution over actions is obtained from the agents' LA. Once a permutation is constructed by playing the game, a common reward (permutation function) or individual reward can be obtained. These common or individual rewards are then given to the agents' LA, which consequently update their probabilistic distribution. Experiments have shown that this decentralized approach has very similar performance characteristics as the 'probability matrix approach'.

6 Experiments

As an illustration of the methods' behaviour, some experiments were performed on a fictitious permutation function and simple benchmark problems, the TSP and an assignment problem. Subsequently, application to a more extensive multi-project scheduling problem is given, which shows the real advantage of the described methods. The following experiments report on the decentralized approach unless mentioned otherwise. Similar properties were observed for the hierarchical and probability matrix approach. All results are averaged over 100 runs and the experiments were performed on an Intel Core i7 2600 3.4Ghz processor, using the Java version 6 programming language.

6.1 Peaked permutation function

Consider the following permutation function definition. If the decimal number of the permutation π is $dec(\pi)$ according to the factorial number system (Lehmer code) [9]. Then the value of the permutation is defined as:

$$f(\pi) = \left(\frac{2dec(\pi)}{n!}\right)^{10} \text{ if } dec(\pi) \leq \frac{n!}{2} \tag{3}$$

$$= \left(\frac{2(n! - dec(\pi))}{n!}\right)^{10} \text{ if } dec(\pi) > \frac{n!}{2} \tag{4}$$

This function has a peak value in the middle of the permutation range. Figure 6 shows this permutation function for $n = 9$.

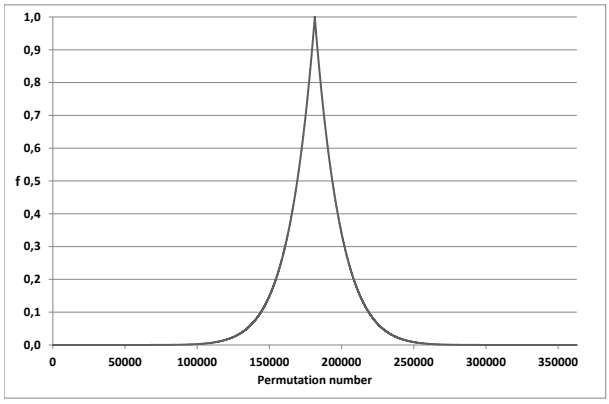

Fig. 6. Peaked permutation function landscape $n = 9$.

Figure 7 compares the average calculation time (in milliseconds) of the presented approaches. For each size $n = 2..20$, 5000 iterations are performed on the peaked permutation function. A learning rate of 0.005 is used for each approach. All but the naive approach show good calculation time properties.

Figure 8 shows the maximum function value over a number of iterations (50,100,500) for different learning rates when applying the decentralized approach with common reward (i.e. the permutation function value). The results show that for a particular range of learning rates better permutations are learned, compared to random sampling (i.e. learning rate equal to 0). When more iterations are given for learning, the difference between random sampling and learning becomes smaller. Bad performance can be observed when the learning rates are too high, and thus premature convergence occurs.

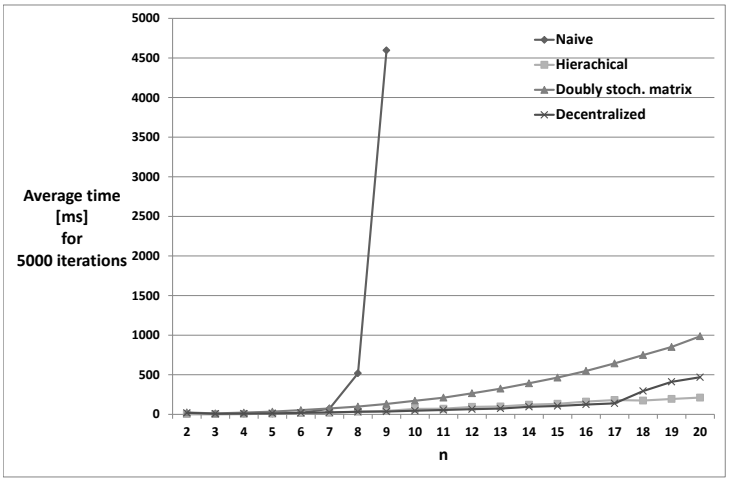

Fig. 7. Average calculation time in milliseconds for performing 5000 iterations of the presented approaches for different permutation sizes on the peaked permutation function.

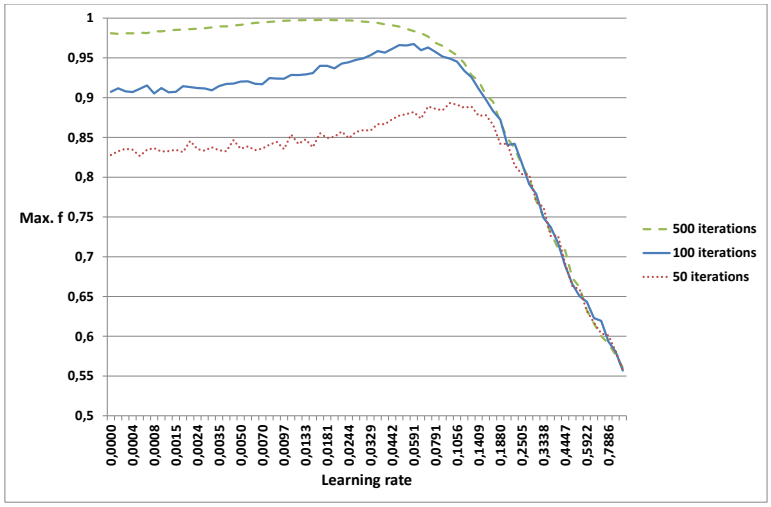

Fig. 8. Max. objective function value for different number of iterations on a peaked permutation function of size $n = 15$.

6.2 TSP

We tested the decentralized approach with common reward on a number of TSP instances from TSPLIB[5]. The distance was scaled to a value between 0 and 1, such that a value of 1 corresponds to an optimal distance and 0 corresponds to an upper bound on the distance. Figure 9 shows the maximum function value over a number of iterations (1000, 5000, 10000, 50000) for different learning rates on a size $n = 17$ instance with name 'gr17'. For a particular range of learning

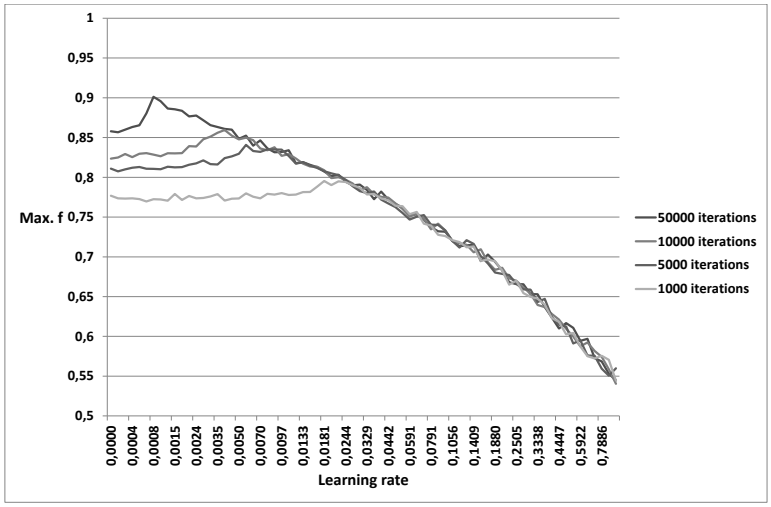

Fig. 9. Max. objective function value for different number of iterations on a TSP instance (gr17) of size $n = 17$.

rates better solution values can be observed, compared to random sampling. If more iterations are given, then the best solutions occur for lower learning rates. Again, too high learning rates lead to worse solutions.

6.3 Assignment problem

Assignment problems belong to the category of permutation functions where the total cost of the permutation is equal to the sum of the individual costs. Therefore both individual and global rewards can be given to the agents. Figure 10 shows a

[5] TSPLIB website: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ ; last check of address September: 23, 2011

comparison of the maximum objective function value for individual and common rewards on a random assignment problem of size $n = 9$ with the cost matrix of Table 2. For each learning rate a run of 1000 iterations is performed where the maximal objective function value is measured. The objective function is scaled between 0 and 1 such that 1 corresponds to the optimal solution. The results show that individual rewards produce better solutions than common rewards. For a particular range of learning rates the method performs better than random sampling, and the optimal solution can be found by using individual rewards .

	T0	T1	T2	T3	T4	T5	T6	T7	T8
A0	7	8	5	3	9	3	9	4	7
A1	3	6	9	3	2	9	6	5	7
A2	6	3	5	1	3	6	9	2	7
A3	8	1	9	3	3	6	3	6	3
A4	7	3	5	7	3	8	9	3	2
A5	4	2	8	2	7	5	4	6	4
A6	7	8	8	9	4	8	9	8	8
A7	7	4	7	8	9	8	1	3	5
A9	9	3	9	7	6	1	5	2	8

Table 2. Cost matrix for a random assignment problem of size $n = 9$

6.4 Multi-project scheduling

The decentralized method introduced in the present paper, was used for solving the decentralized resource-constrained multi project scheduling problem (DRCMPSP) [14]. The DRCMPSP was introduced in [4, 5] and extended in [8]. It is a generalization of the Resource Constrained Project Scheduling Problem (RCPSP) [3, 10] and can be stated as follows. A set of n projects have to be scheduled simultaneously using autonomous and self-interested decision makers. Each individual project contains a set of jobs or activities, precedence relations between the jobs, and resource requirements for executing each job. These resource requirements constitute local renewable resources, and global renewable resources shared among all projects. A global objective value must be minimized, examples include but are not limited to: the average project delay(APD), the total makespan (TMS), and the deviation of the project delay (DPD). The project delay of a project is its makespan minus the critical path duration. The remainder of the current section will concentrate on the APD objective. A constructive schedule generation scheme was applied to solve the DRCMPSP, requiring the project order and the job order for each project as input. The project order is a permutation of projects, while the job order is a permutation with precedence constraints. The decentralized method with the dispersion game was used to find good quality project orders, leading to schedules with a low APD objective value. Each project was represented by one project order decision maker.

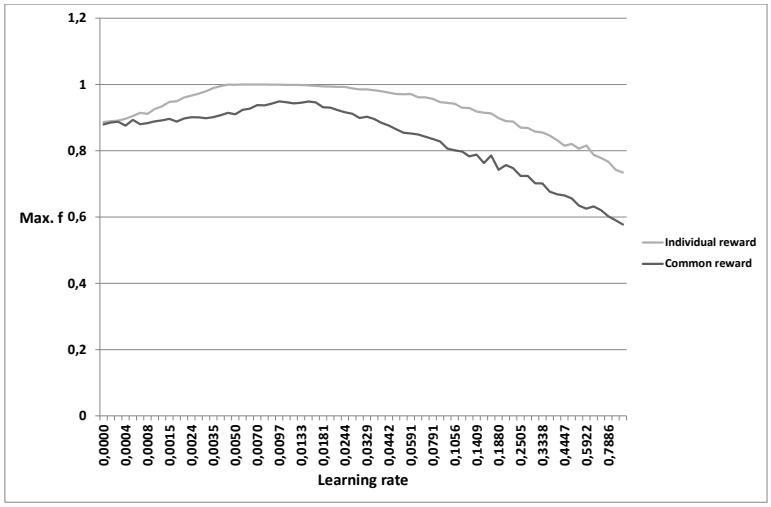

Fig. 10. Comparison of max. objective function value for individual and common reward on a assignment problem of size $n = 9$.

Instances from the Multi project scheduling problem library[6] have been experimented on. To date, the method belongs to the state-of-the-art for this problem, showing 104 best solutions out of 140, with respect to the average project delay objective.

7 Conclusion

In this paper we have presented several centralized and decentralized methods using learning automata for the online learning of good quality permutations. Different permutation functions have been discussed. The capabilities of the methods have been analyzed and demonstrated on well known benchmark problems, and we demonstrated how to successfully apply these methods to a multi-project scheduling problem. The methods are very general because they do not use any problem specific information or domain knowledge, which makes them well suited for application within general optimization methods, like hyper-heuristics. In the future we can apply these methods to other optimization problems and make them core elements of new intelligent optimization methods.

[6] Multi project scheduling problem library: http://www.mpsplib.com ; retrieved September: 23, 2011

Acknowledgment
Thanks to Erik Van Achter for his help on improving the quality of this text.

References

1. Baluja, S.: Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Tech. Rep. CMU-CS-94-163, School of Computer Science, Carnegie Mellon University (1994)
2. Baluja, S., Davies, S.: Fast probabilistic modeling for combinatorial optimization. In: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence. pp. 469–476. AAAI '98/IAAI '98, American Association for Artificial Intelligence, Menlo Park, CA, USA (1998), http://dl.acm.org/citation.cfm?id=295240.295718
3. Blazewicz, J., Lenstra, J., Rinnooy Kan, A.: Scheduling subject to resource constraints: classification and complexity. Discrete Applied Mathematics 5, 11–24 (1983)
4. Confessore, G., Giordani, S., Rismondo, S.: An auction based approach in decentralized project scheduling. In: Proc. of PMS 2002 International Workshop on Project Management and Scheduling. Valencia. pp. 110–113 (2002)
5. Confessore, G., Giordani, S., Rismondo, S.: A market-based multi-agent system model for decentralized multi-project scheduling. Annals of Operational Research 150, 115–135 (2007)
6. Grenager, T., Powers, R., Shoham, Y.: Dispersion games: general definitions and some specific learning results (2002), citeseer.ist.psu.edu/grenager02dispersion.html
7. Helmbold, D.P., Warmuth, M.K.: Learning permutations with exponential weights. Journal of Machine Learning Research 10, 1705–1736 (2009)
8. Homberger, J.: A multi-agent system for the decentralized resource-constrained multi-project scheduling problem. International Transactions in Operational Research 14, 565–589 (2007)
9. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley (1973)
10. Kolisch, R.: Serial and parallel resource-constrained project scheduling methods revisited: theory and computation. European Journal of Operational Research 90, 320–333 (1996)
11. Narendra, K., Thathachar, M.: Learning Automata: An Introduction. Prentice-Hall International, Inc (1989)
12. Thathachar, M., Ramakrishnan, K.: A hierarchical system of learning automata. IEEE Transactions On Systems, Man, And Cybernetics 11 (3), 236–241 (1981)
13. Thathachar, M., Sastry, P.: Networks of Learning Automata: Techniques for Online Stochastic Optimization. Kluwer Academic Publishers (2004)
14. Wauters, T., Verbeeck, K., De Causmaecker, P., Vanden Berghe, G.: A game theoretic approach to decentralized multi-project scheduling (extended abstract). In: Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010). No. R24 (2010)
15. Wauters, T., Verbeeck, K., Vanden Berghe, G., De Causmaecker, P.: Learning agents for the multi-mode project scheduling problem. Journal of the operational research society 62 (2), 281–290 (February 2011)