SARSA-based reinforcement learning for motion planning in Serial Manipulators

Ignazio Aleo, Paolo Arena and Luca Patané.

***Abstract*— In this paper we investigate an application in which a serial manipulator is engaged in a task driven state transition learning through a set of basic behaviours (i.e. inherited actions). The approach is based on an extension of the SARSA reinforcement learning algorithm.**

In particular, the case under study consists in the control of the end-effector position sequences of a custom serial manipulator (i.e. the MiniARM) in a constrained shortest path problem.

In order to test performances of the overall algorithm and the improvement beyond the state of the art, those strategies have been implemented both in simulation and in a real hardware environment. Results have been analyzed in terms of learning time and iterations needed to complete the assigned task.

I. Introduction

Reinforcement Learning (RL) is the straightforward learning paradigm for bio-inspired architectures.

In the last few decades the state of the art for robot learning has moved towards RL [1] [2]. Unfortunately in robotics the common trial-and-error practice is not so trivial: the low-cost mechanical structures have very different compliance and timing capabilities from their biological counterpart. A large number of different approaches have been proposed to overcome this kind of problems [3] [4].

Generally speaking, the underling idea is to use computational power to minimize necessity of real environment interaction. Eligibility traces, model based approaches (for instances Prioritized Sweeping [5]) and probabilistic methods (Ant-Colony Optimization [6]) increase computational costs in order to reduce the number of environment examples needed [7] [8]. A commonly used strategy is to perform multiple iterations based on past observations between two real experiences [9].

In this paper an application of a modified version of a simple State-Action-Reward-State-Action (SARSA) algorithm is presented in order to cope with a discrete shortest path problem with a redundant serial manipulator.

The task taken into consideration includes a robotic arm equipped with a pointer as end-effector. The robot, hereafter referred to as the agent, should learn which is the best way to position the end-effector on all the black squares in a given custom checkerboard.

Multiple levels have been used to hierarchically control the hardware. A high level, behavioral, control is first performed in a host computer. The algorithm output (i.e. a discrete checkerboard position) is then given as input to a kinematic inversion algorithm able to cope with the redundant serial structure. A low level control is then performed both in a custom designed control board and in the distributed control system of the robot.

As in any other Q-learning based algorithms, for a given state and a policy, the next action is chosen, reward is evaluated, and therefore the action-value (Q-function) for state-action pair is updated iteratively. The particular reward function determines the overall behaviour of the agent.

In contrast to what previously discussed, for a given state the most suitable action (i.e. the one that leads to the estimated best next state) is simulated and through the uncomplete model the estimated reward is assigned. Furthermore the learning process is strongly accelerated with action simulation performed by the agent based on what previously learnt: starting from a particular state the agent explores multiple parallel simulated trials and respectively evaluates rewards creating a tree of all chosen possibilities.

It is clear that the problem is better achieved via *n*-step prediction algorithms [9]. Eligibility traces have been used in order to further keep trace of meaningful state-action pairs.

The system performances have been evaluated using the Ant-Colony Optimization meta-heuristic method [6], that is one of the most suitable approaches for this kind of problem.

In the first section, the considered general SARSA model is introduced. In the second section, an improvement with a model-based prediction is discussed. Finally, in the other two sections the experimental architecture is presented and the experimental results are shown.

II. Model description

The problem of the shortest path is herewith introduced in a unconventional task-driven way. The environment is a modified checkerboard with white and black squares: when the agent is on a black ($s_v = 0$) square, the selected area changes its state to white ($s_v = 1$). The goal is reached when all the squares become white. Both the state and the action spaces are discrete. In particular, state set S includes all the bistable $M \times N$ checkerboard squares configurations together with agent position for a total of n_s possible states:

$$S = \{s_1, s_2, ..., s_{n_s}\} \qquad \text{with } n_s = M \cdot N \cdot 2^{M \cdot N}. \tag{1}$$

The actions from action set A can drive the end-effector to one of all possible squares:

$$A = \{a_1, a_2, ..., a_{n_a}\} \qquad \text{with } n_a = M \cdot N. \tag{2}$$

A representation of one possible state from the state-space set S is depicted in Fig. 1. As previously described, for a given state all the possible actions $a \in A$ are those sketched in Fig. 2. Therefore considering the actual state s and what

Ignazio Aleo, Paolo Arena, and Luca Patané are with the Dipartimento di Ingegneria Elettrica, Elettronica e dei Sistemi (DIEES), Universitá degli Studi di Catania, Italy (email: {ialeo,parena,lpatane}@diees.unict.it)

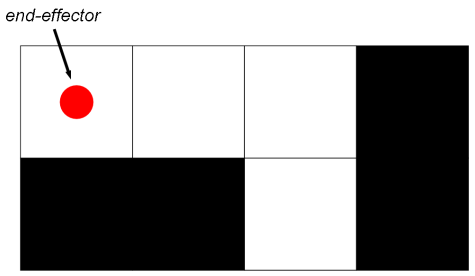

Fig. 1. Example of four by two checkerboard together with the end-effector position representation (circle).

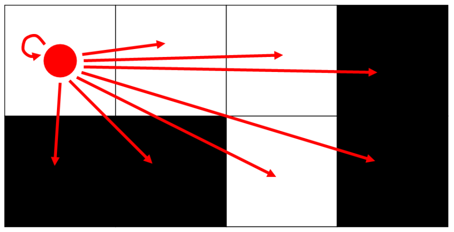

Fig. 2. Example of all possible actions (arrows) in a four by two state-space checkerboard by the agent (circle).

since now learned, the most rewarding action a^* is chosen, using Q-learning,with the following method:

$$a^* = \text{argmax}_a(Q(s,a)). \quad (3)$$

In contrast to what happens in Q-learning, shown in equation (3), in SARSA-based algorithms the performed action is not always the one with the highest value in Q-function (for a given state). For instances, the so called e-greedy policy can be followed by the agent as described in the next pseudo-code block.

```
...
if  $(rand < \varepsilon)$
then $a^* \leftarrow \text{randint}([1, n_A])$;
else $a^* \leftarrow \text{argmax}_a(Q(s^*, a))$;
endif
...
```

The simplest SARSA block diagram is shown in Fig. 3 where, after parameters and variables initialization performed by the **init** block, the (s, a, r, s^*, n^*) vector is collected iteration after iteration. Moreover the action value function **Q** is updated at each iteration as in the following assignment:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s^*, a^*) - Q(s,a)), \quad (4)$$

where α is the step-size of the learning process, r is the current step reward and γ is the discount factor while s^* is the next step state.

Considering that the largest part of the movement time of the real manipulator from one cell to another is not dependent on the distance, though white cells are not a physical constraint, the best path should avoid them. Therefore

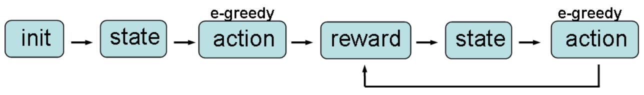

Fig. 3. Block diagram of single episode cycle. After an initialization phase implemented in the **init** block, the (s, a, r, s^*, n^*) vector is built at each iteration.

the problem can be easily split into two sub-problems: the agent should learn to avoid white squares and than learn the shortest path through black squares.

As can be easily understood, the former is not suitable for one-step prediction: too many iterations are needed to obtain low errors (i.e. big deal for real robot implementation).

A. *Eligibility Traces*

In order to reduce the number of iterations despite the increase in computational cost, eligibility traces have been introduced [9] [10].

When a state-action pair (s, a) is visited, the corresponding value in eligibility function is updated as follows:

$$e(s,a) \quad \leftarrow \quad e(s,a) + 1. \quad (5)$$

The Q-function update is modified as in the following equations:

$$\delta = r + \gamma Q(s^*, a^*) - Q(s,a), \quad (6)$$

and then for all state-action pairs (s, a)

$$\begin{aligned} Q(s,a) &\leftarrow Q(s,a) + \alpha\delta e(s,a), \\ e(s,a) &\leftarrow \lambda e(s,a), \end{aligned} \quad (7)$$

where λ is the eligibility discount factor and α is again the step-size of the learning process. From a computational point of view the straightforward application of eligibility trace implies a single episode problem dimensionality to increase from $O(steps)$ to $O(steps \cdot n_s)$, where $steps$ is the number of action performed in order to reach the goal.

Therefore in the real application a modified version of the algorithm has been implemented to hold information about the (s, a) pairs for which the following inequality is verified:

$$e(s,a) \geq \varepsilon \qquad with \quad \varepsilon \geq 0. \quad (8)$$

This increases the complexity from $O(steps)$ to $O(steps!)$ (it must be noticed that typically $n_s \gg steps$).

B. *Reward cost function*

The definition of the reward function is an important aspect of the algorithm in the proposed application. For shortest path solution, the coefficient γ is set to $\gamma = 1$, as suggested in [9], and a negative reward for all non-terminal states is given, while zero reward is assigned for goal achievement.

The cost function used as reward is state-dependent and evaluated through the following system:

$$r = \begin{cases} -(x - a_x)^2 - (y - a_y)^2 & if \quad s_v = 1, \\ -v_b \quad with \quad v_b \geq 0 & if \quad s_v = 0, \end{cases} \quad (9)$$

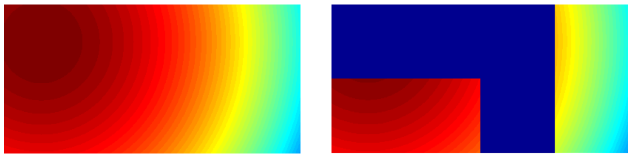

Fig. 4. Cost function part for initial state depicted in Fig. 2 in the case of movement towards black squares with $s_v = 1$ (on the left). State-dependent overall cost function used as reward (on the right).

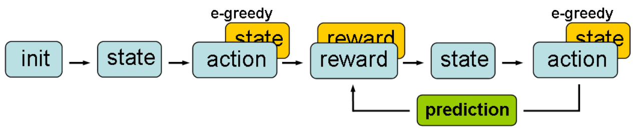

Fig. 5. Block diagram of cycle with state-to-state prediction. After an initialization phase implemented in the **init** block, the (s, a, r, s^*, n^*) vector is built each iteration, model Q_s is updated and agent prediction is evaluated by the **prediction** block and darker **reward** block.

where it is clear how all non-terminal states are negatively rewarded according to the squared step length, in order to progressively reduce overall traveled distance. Actions that lead to a white square (i.e. in which $s_v = 1$) are negatively biased to further increase convergence speed of the algorithm. An example of the cost function used as reward is shown Fig. 4.

III. Predictive Model

As already described, the model has been enhanced to improve the performances in terms of number of episodes needed for a complete learning, in order to meet real hardware timing necessities. The approach consists in reducing the number of real actions needed for learning by introducing a virtual agent that will simulate the outcomes of the environment based on what previously learnt by the real agent.

A state-to-state transition matrix Q_s is built incrementally through iterations based on actual state-action pair (s, a) next state s_n and respective reward r. Therefore the updated block diagram is shown in the diagram in Fig. 5, where the agent movement prediction is performed (by the **prediction** block) and evaluated at each iteration using the vector (s, a, r, s^*, n^*) updating the Q_s function.

$$Q_s(s, s_n) \leftarrow f(s, a, r, s_n), \tag{10}$$

where f is a reinforcement-based state-to-state function. An example of parallel agent motion simulation that forms a state predicted tree is shown in Fig. 6.

It must be noticed that even for a trivial case of $M = 4$ and $N = 2$, Q_s is quite a huge matrix 2048 by 2048. Nevertheless, if we choose to initialize it as zeros matrix instead of random, this results sparse: the possible transitions from one state to any other are fewer, as computed in the equation:

$$n_a^* = M \cdot N \cdot n_{bk}, \tag{11}$$

where n_{bk} is the number of black cells in the state space. Therefore the worst case is $n_a^* = (M \cdot N)^2$, in our example is $n_a^* = 64$.

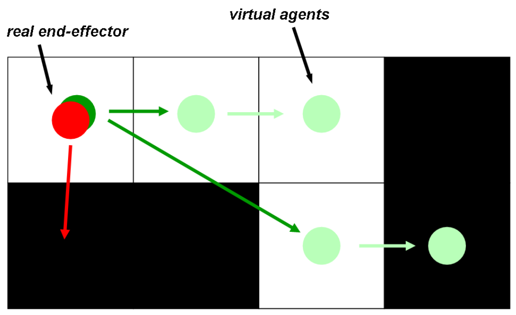

Fig. 6. Parallel agent simulation in state predicted tree for a four by two state-space checkerboard. The light arrows and circles indicates possible transitions and states of the virtual agent.

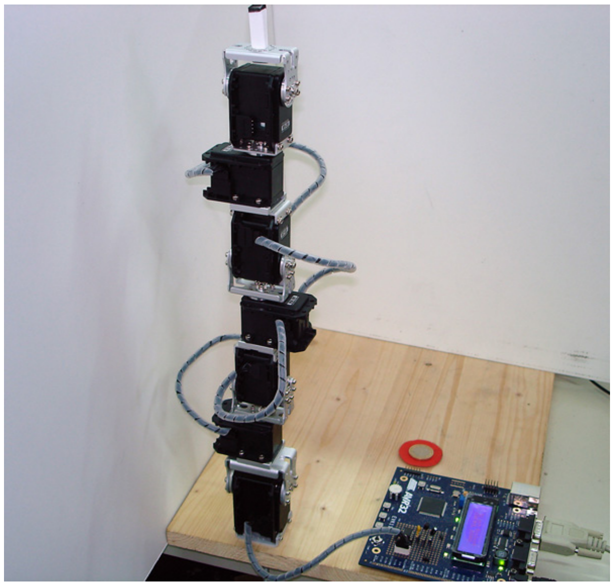

Fig. 7. MiniARM manipulator designed and built at DIEES, University of Catania. It is a serial seven degrees of freedom manipulator with a pointer as end effector.

IV. Experimental Setup

The algorithm has been validated both in a three dimensional simulation environment and with a real manipulator.

The considered robot is the MiniARM [11] (shown in Fig. 7), a custom built seven degrees of freedoms manipulator with revolute joints.

The architecture for both simulation and hardware control is sketched in a diagram in Fig. 8 Virtual Reality Modeling Language (VRML) models of simple environment and manipulator (depicted in Fig. 9) have been realized in order to have the same function interfaces as the real hardware control devices. The highest level control is achieved with a PC, while the low level control is performed with a 32 bit mcu-based board. The PC-board communication is done with a USB-serial interface, while the robot is position-controlled through a RS485 serial bus.

Although positions are known and few, in the case study the end-effector positioning is realized with inverse kinematic

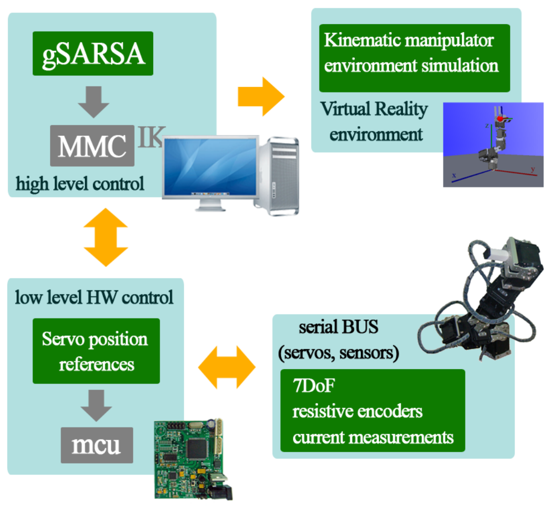
gSARSA
MMC
IK
high level control
Kinematic manipulator environment simulation
Virtual Reality environment
low level HW control
Servo position references
mcu
serial BUS (servos, sensors)
7DoF resistive encoders current measurements

Fig. 8. Hardware block diagram of the adopted experimental setup. High level algorithms run on a PC-based platform. Both tri-dimensional kinematic simulation and real hardware control have been implemented. Low level control of the custom manipulator (i.e. the MiniARM) is achieved, through a USB-to-serial converter, thanks to a 32 bit microcontroller-based custom designed board. Lowest level motor control and angular joint readings are decentralized in each servo.

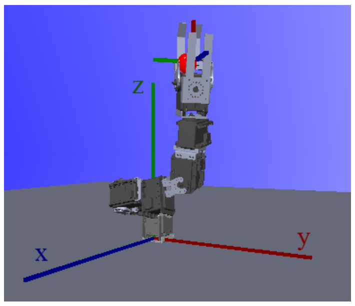

Fig. 9. Simulated manipulator and target in the virtual environment.

algorithm for generality (an iterative novel strategy called Mean of Multiple Computation MMC [12], [13]).

V. Experimental Results and Comparisons

A reinforcement learning algorithm, because of the high parameter and particular strategy dependency, is not easy to be numerically compared to other approaches. Under this point of view a cross-comparison between predictive and non predictive algorithm seemed a straightforward comparison. Moreover the well known ACO meta-heuristic method [6] is used as gold-reference to give idea of best achievable performance without explicitly computing it.

Two different performance indexes have been chosen to compare the proposed algorithm with respect to other classical SARSA-based approaches. The first is the normalized number of steps needed for a single task achievement.

$$n_s = \frac{n_{steps}}{n_{bk}} \tag{12}$$

where n_{steps} is the overall number of steps executed by the agent while n_{bk} is the number of black squares in the checkerboard.

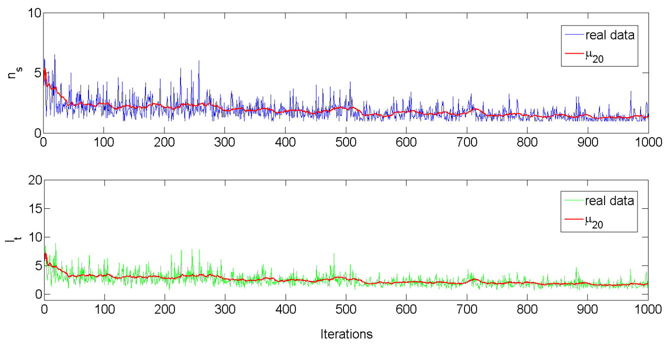

Fig. 10. Analyzed indexes in an experimental test. The dashed line indicates the real data and the solid line (μ_{20}) is the moving average computed with the most recent twenty samples. Both n_s and l_t have been investigated in order to understand different learning capabilities on tasks (i.e. avoid white squares and choose shortest black squares path.

The second performance index is:

$$l_t = \frac{d - d_b^*}{n_{bk}}, \tag{13}$$

where d is the total distance traveled by the agent and d_b^* is the minimum distance between the end-effector starting position and the black squares. It is clear how though the minimum value for l_t is $l_t^* = 1$, not all configurations admit this value as optimal sequence because of the distance between black squares in initial state. A typical dynamical evolution of the two indexes, chosen to evaluate system performance, through iteration is shown is Fig. 10. As it is possible to see, both indexes show a fast decrease through iterations. Considering an average time per movement $t_{avg} = 1s$, the overall computational time requested per iteration is far less and therefore, as discussed above, the number of real agent actions can be kept low.

In order to compare the performances, a number of iteration of 1000 was chosen (i.e. number of real robot actions). Each robot experiment is completed in about $t_{max} = 20\ minutes$ (i.e. all together with computational time).

As shown in Fig. 11 and in Fig. 12, the presence of the state transition model improves the learning time in terms of number of actions performed from the real robot: both chosen indexes show faster decreasing in presence of a predictive model. It must be remarked that the ACO algorithm has been implemented, for simplicity, to solve shortest path sub-task without considering the actions leading to white squares. Moreover, by using the ACO strategy the problem is solved iteratively for a given static environment configuration and it is not possible to extract a general model from the data. Nevertheless, it is a quite good benchmark algorithm because it provides a fast numerical sub-optimal solution for a particular problem. For each RL iteration, $i_{ACO} = 500$ have been performed for a computational time $t_{ACO} = 0.12s$ per each solution. The color map depicted in Fig. 13 shows the state-action weights matrix $\mathbf{Q}$ before learning. Iteration after iteration the weight matrix changes its values. The color map in Fig. 14 shows the same matrix after learning: as

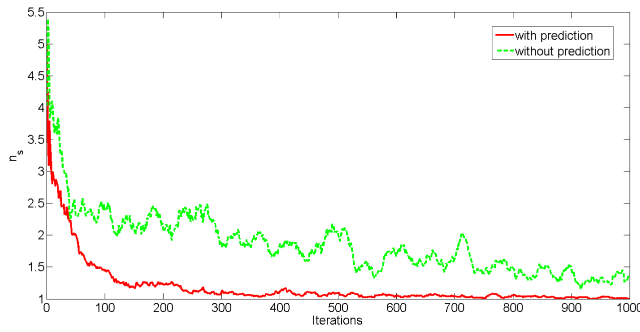

Fig. 11. Normalized number of steps needed for task achievement. It must be remarked that when the agent is able to avoid white squares (first task achieved) $n_s = 1$. Dashed line exploits the performance of the first tested algorithm, without prediction. Solid line indicates the predictive model performances.

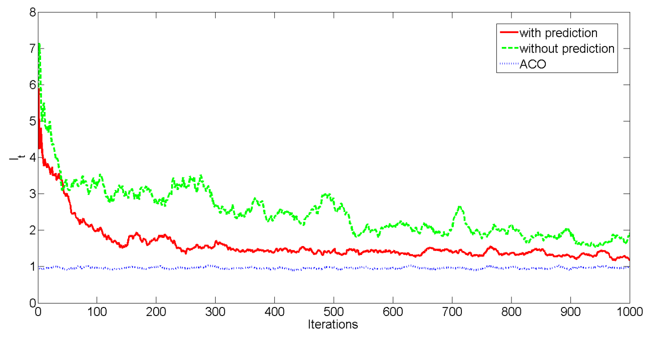

Fig. 12. Summary of the experimental results on shortest path task performance evaluation expressed through the index l_t that is a function of the over-length of the path traveled. Dashed line exploits the performance of the first tested algorithm, without prediction. Solid line depicts the predictive model performances, while the dotted line are the Ant-Colony Optimization performances.

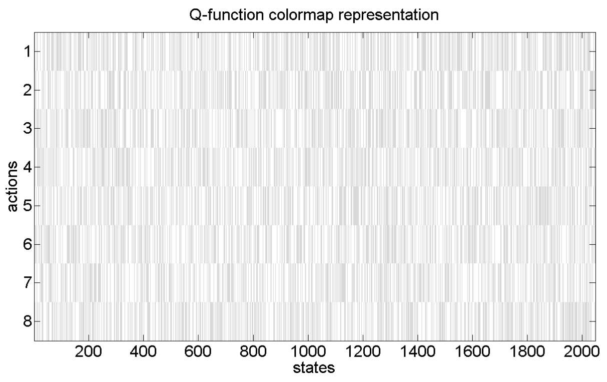

Fig. 13. Example Q-function before learning. In abscissa there are all the possible 2048 states, in ordinate there are all the possible actions. As shown, random high value (lighter) initializations have been chosen.

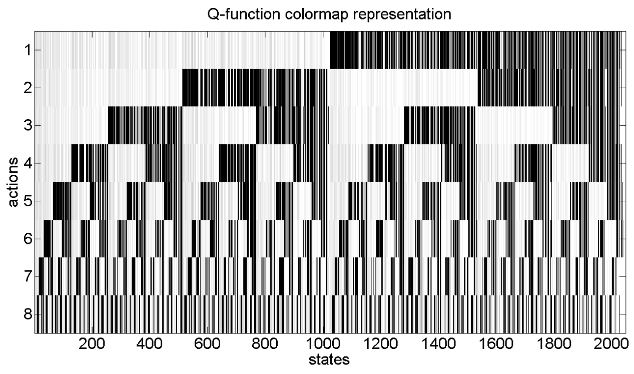

Fig. 14. Example of Q-function after learning. In abscissa there are all the possible 2048 states, in ordinate there are all the possible actions. Due to the binary coding of the states, the trained **Q**-function looks like a binary tree, confirming the correct training. Nevertheless, high value stripes (lighter) in low value areas (darker) indicates non-explored state-action pairs.

it is possible to see, all non-rewarding actions values are decreased. As previously introduced, all described learning results have been obtained using both simulations and the real robot. Up to now no sensors have been applied for state transition check. A laptop LCD monitor has been used for solution visualization. Nevertheless, a touch sensitive panel is a straightforward upgrade that can be used in order to close the loop with the real hardware.

Snapshots from the real manipulator learning are shown in Fig. 15.

VI. Conclusions

An application to shortest path discrete problem in real hardware is presented. Both simulation and experimental results show the improvement achieved thanks to the state-transition model.

It must be remarked that a considerable improvement was obtained in the real application because the number of actions performed by the real agent (end-effector manipulator) was heavily reduced. In fact, the virtual agent performs a lot of simulated actions based on what previously learnt by the real agent.

Comparison results with probabilistic methods, as Ant-Colony Optimization, show that, though the latter shows a lower index value than final performance index value of the proposed strategy, these kinds of approaches cannot extract the general model from the data.

Acknowledgements

The authors acknowledge the support of the European Commission under the project ICT 216227 - SPARK II "Spatial-temporal patterns for action-oriented perception in roving robots II: an insect brain computational model".

References

[1] Honglak Lee, Yirong Shen, Chih-Han Yu, Gurjeet Singh and Andrew Y. Ng, "Quadruped Robot Obstacle Negotiation via Reinforcement Learning", *ICRA*, 2006.

[2] Arthur P. S. Braga, Aluizio F. R. Araujo, "A topological reinforcement learning agent for navigation" *Neural Comput. and Applic.*, vol. 12, pp. 220-236, 2003.

[3] Bram Bakker, Viktor Zhumatiy, Gabriel Gruener and Jürgen Schmidhuber, "Quasi-Online Reinforcement Learning for Robots", Proc. *IEEE International Conference on Robotics and Automation*, 2006.

[4] Kolter, Abbeel and Ng, "Hierarchical Apprenticeship Learning with Application to Quadruped Locomotion", *NIPS 2008*

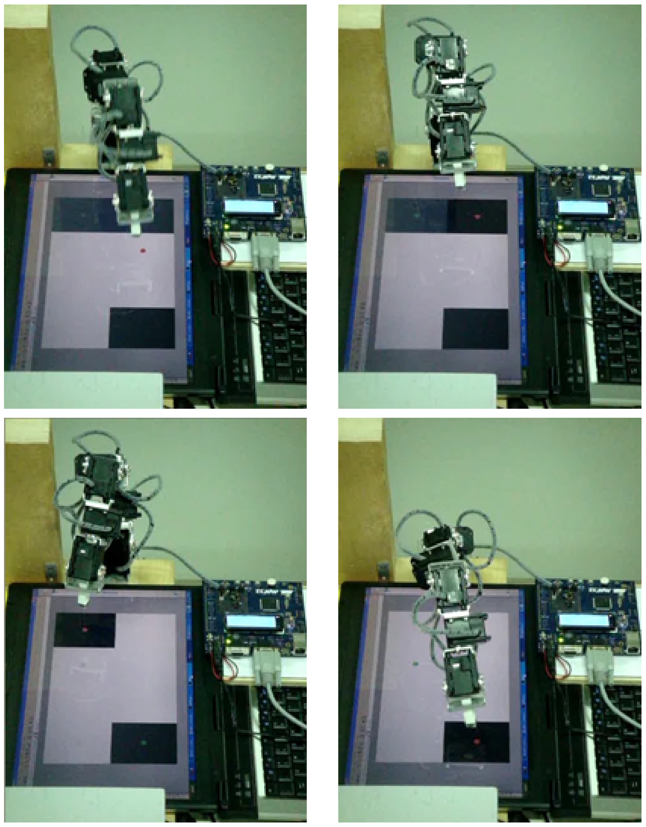

Fig. 15. Example of sequence reproduction after learning process has been successfully performed. A laptop LCD was adopted to visualize the solution.

[5] Andrew W. Moore and Christopher G. Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less time", *Machine Learning*, Springer, vol. 13, pp. 103-130, 1993.

[6] Marco Dorigo and Thomas Stützle, "Ant Colony Optimization", *MIT Press*, 2004.

[7] Leslie P. Kaelbling, Michael L. Littman and Andrew W. Moore, "Reinforcement Learning: A Survey", *Journal of Artilcial Intelligence Research*, vol. 4, 1996.

[8] Andrew G. Barto and Sridhar Mahadevan "Recent advances in hierarchical reinforcement learning", *Discrete Event Dynamic Systems: Theory and Applications*, vol. 13, pp. 41-77, 2003.

[9] Sutton, R.G, Barto, A.G, "Reinforcement Learning: An Introduction" 2 ed. (sl): Mit Press. pp. 51-185,1998.

[10] Ribeiro, C.H.C., "A Tutorial on Reinforcement Learning Techniques" in Proc. of *International Conference on Neural Networks*, INNS Press, Washington, DC, USA, pp. 1-23, 1999.

[11] EU Project SPARK II, website online at *www.spark2.diees.unict.it/MiniArm.html*.

[12] H. Cruse and U. Steinkuhler, "Solution of the direct and inverse kinematics problems by a common algorithm based on the mean of multiple computations", *Biol. Cybernetics* vol. 69, pp. 345-351 2, 1993.

[13] P. Arena and L. Patané, "Spatial Temporal Patterns for Action Oriented Perception in Roving Robots", Springer, Series: Cognitive Systems Monographs, vol. 1, 2009.