Evolvable Hardware:From Biology to Hardware(ICES2001),LNCS 2210,Springer,2001,pp.267-278.

Solving partially observable problems by evolution and learning of finite state machines

Eduardo Sanchez[1], Andrés Pérez-Uribe[2], and Bertrand Mesot[1]

(1) Logic Systems Laboratory
Computer Science Department
Swiss Federal Institute of Technology-Lausanne
Email: Eduardo.Sanchez@epfl.ch, Bertrand.Mesot@epfl.ch
(2)Parallelism and Artificial Intelligence Group
Department of Informatics
University of Fribourg, Switzerland
Email: Andres.PerezUribe@unifr.ch

Abstract. Finite state machines (FSM) have been successfully used to implement the control of an agent to solve particular sequential tasks. Nevertheless, finite state machines must be hand-coded by the engineer, which might be very difficult for complex tasks. Researchers have used evolutionary techniques to evolve finite state machines and find automatic solutions to sequential tasks. Their approach consists on encoding the state-transition table defining a finite state machine in the genome. However, the search space of such approach tends to be innecesarily huge. In this article, we propose an alternative approach for the automatic design of finite state machines using artificial evolution and learning techniques: the *SOS-algorithm*. We have obtained very impresive results on experimental work solving partially observable problems.

1 Introduction

In digital systems we talk about combinational and sequential systems. In the first case, the output of such a system depends uniquely on the input data at the moment, while the behavior of sequential systems depends on the sequence of inputs over a period of time [14]. Thus, in the latter case, a memory is required to save a trace of the sequence of inputs: that is, the *state* of the system. Given the finite character of the memory, the number of states is necessarily finite; therefore, one often speaks of sequential systems as finite state machines (FSM).

One may use a finite state machine to implement the controller of an agent to solve a particular sequential task. However, such a finite state machine must be hand-coded by the user, which may be a tedious work when the tasks are highly complex. Evolutionary techniques may reduce the user's effort to solve such a task. For example, one may use genetic algorithms to evolve finite state machines instead of designing them by hand.

Nevertheless, in some problems, like maze solving, evolutionary techniques may require quite much more computational effort than, for instance, reinforcement learning techniques. Indeed, contrary to reinforcement learning techniques

Evolvable Hardware:From Biology to Hardware(ICES2001),LNCS 2210,Springer,2001,pp.267-278.

that learn by interacting with their environment, evolutionary methods do ignore much of the useful structure of the problem: the states an agent passes through during its lifetime, which actions it selects, etc. [18]

In the reinforcement learning framework, the agent makes its decisions as a function of a signal from the environement, called the environment's *state*. However, it should be noticed that in such framework, the state is any information available to the agent, e.g., a local *observation* of the environment. Thus, in certain reinforcement learning problems, the state does not summarize the past sensations of the agent (i.e., the history of the agent). In the reinforcement learning framework, we speak of Markov states if they succed in retaining all relevant information of the history of the agent, as it is the case of a state in a finite state machine.

Many of the learning techniques developed within the reinforcement learning framework can learn optimal policies if the problem is Markovian; on the other hand, they can only approximate the optimal policy. Partially observable mazes are an example of non-Markovian tasks.

In this paper, we introduce an algorithm for evolving and learning finite state machines, and test its capabilities solving partially observable mazes. In Section 2, we describe the maze learning task and the concept of partial observability. In Sections 3, 4, and 5, we briefly describe finite state machines (FSM), reinforcement learning, and evolutionary techniques. Section 6 describes our *SOS-algorithm* for evolving and learning finite state machines, Section 7 delineates the experimental results, and finally, Section 8 presents some concluding remarks.

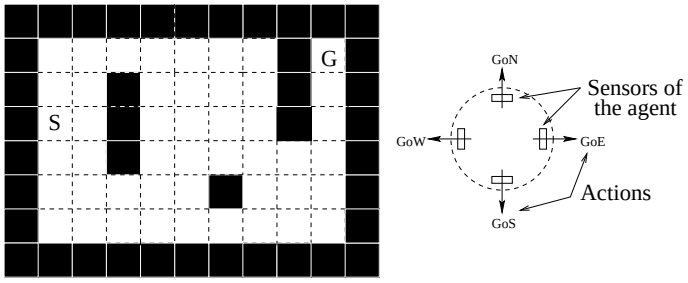

Fig. 1. Example of a maze problem and of an agent. The S location indicates the starting postion, the G location is the goal. The agent is able to detect an obstacle in any of the four directions, and may choose one among four possible actions, or a NOP (no operation) action.

2 Maze learning and partial observability

Mazes have long been used as a benchmark for machine learning techniques. Maze learning is presented here as an abstraction of the real navigation problem

Evolvable Hardware:From Biology to Hardware(ICES2001),LNCS 2210,Springer,2001,pp.267-278.

in mobile autonomous robots: One has an a-priori partition of the sensor space of the robot, a perfect vision, and a discrete world. An autonomous robot is faced to the problem of finding a goal (G) starting from a valid position (S) in a labyrinth-like environment (see figure 1). In some experiments, the robot can only detect an obstacle lying in front of it, and may move one position forward, or turn right or left. In other experiments, the robot can detect an obstacle to the north, south, left and to the right, and may move one position in the same directions. The problem here is to find the correct actions to take, such that the robot avoids obstacles and finds the shortest path to the goal.

In mobile robotics, the agent has only partial information about the current state of the environment, that is, it does not know the state of the whole world from the sensory input alone (i.e., the observations). The agent is said to suffer from *hidden-state* [8] or *perceptual aliasing* [19], and the environment is said to be partially observable.

For an agent to behave in a partially observable environment, a sort of structural abstraction is needed. One technique performs *spatial abstraction*: it consists on aggregating similar observations and treating them to some degree as the same. Spatial abstraction, or function approximation seeks to develop mappings which are more compact than lookup tables. Nearest neighbor, neural networks, fuzzy logic, and other approaches have been widely used [11, 15]. Another technique performs *temporal abstraction*, in which certain sequences of actions are treated as higher level actions [12].

In general, the partially observable Markov decision processes (POMDP) provides a formal framework for studying these problems [2, 5]. The difficulty of acting and planning in partially observable environments is illustrated by the fact that the optimal policy may need the use of the complete previous history of the system, that is, the whole sequence of observations, actions, and rewards to determine the next action to perform [9].

Researchers attempt to aleviate such difficulty by developing hierarchical reinforcement learning techniques [3], incremental learning methods [13], multi-agent reinforcement learning techniques [16], etc. In this paper, we present an alternative method for solving partially observable mazes by using artificial evolution and learning of finite state machines.

3 Finite State Machines

The behavior of sequential systems depends on the sequence of inputs over a period of time. A memory is required to save a trace of the sequence of inputs: that is, the *state* of the system. Given the finite character of the memory, the number of states is necessarily finite; therefore, one often speaks of sequential systems as finite state machines (FSM) [14].

The state of a sequential system is therefore stored by the state variables of the system; these variables contain all the information concerning the past necessary for calculation of the future behavior.

next state = f(present state, inputs)

The output variables of a sequential system (i.e., the actions of the system) can depend on the present state and present value of inputs, or solely on the present state: one speaks in the first case of Mealy machines and of Moore machines in the second case.

outputs $= \delta$*(present state, inputs)* (Mealy machine)
outputs $= \delta$*(present state)* (Moore machine)

In order to better visualize the state sequences of the system, the truth tables of the functions f and δ are put together in a single table called *state table* (or state-transition table). The lines in this table correspond to the states, and the columns correspond to combinations of input variables, where for each entry one places the values corresponding to the functions f and δ (value of the next state and outputs).

It is similarly common to give the information of the state tables under another form, more visual: the *state diagram* or state-transition diagram. This is an oriented graph where each state transition is represented by an arrow, linking the present state and the next state. On each arrow, the input values producing the change of state and the ouput values are written.

4 Reinforcement learning

Reinforcement learning is a computational approach to learning from interaction. This approach offers an attractive paradigm to automating goal-directed learning and decision making [18].

Reinforcement learning tasks are generally treated in discrete time steps: at each time step t, the learning system receives some representation of the environment's *state* s^t, it *tries* an action a, and one step later it is *reinforced* by receiving a scalar evaluation r^t. Finally, it finds itself in a new state s^{t+1}. This is what is called learning by trial-and-error in artificial systems. To solve a reinforcement learning task, the system attempts to maximize the total amount of reward it receives in the long run [18]. To achieve this, the system tries to minimize the so called *temporal-difference error*, computed as the difference between predictions at successive time steps. This is called *temporal difference learning* or *TD*-learning [17].

Reinforcement learning techniques involve the use of a *value function* which can be simply implemented by a look-up table, and is composed of *state values* or *action values*. A state value (denoted as $V(s)$) indicates the expected cumulative reinforcement an agent can receive starting from a given state s; and, an action value (denoted by $Q(s, a)$) indicates the expected cumulative reinforcement an agent can receive taking action a from state s. In the general case, the *policy* of the system (i.e., the way of behaving of an agent) is a simple function of the value

function. Therefore, adaptation here is achieved by updating the value function using the scalar evaluation received while interacting with the environment.

5 Artificial evolution

The idea of applying the biological principle of natural evolution to artificial systems has been used in a wide range of applications. In this work, we use *genetic algorithms*, an iterative procedure that consists of a population of individuals, each one represented by a finite string of symbols (the genome), encoding a possible solution to a given problem [10]. The algorithm starts with an initial population of individuals that is generated at random. At every evolutionary step (a generation), the individuals in the current population are decoded and evaluated according to some predefined quality criterion (the fitness). To form a new population (the next generation), individuals are selected according to their fitness and then modified by using two standard (genetic) operators: mutation (it is applied by flipping one or more random bits in a string with a probability equal to the mutation rate) and crossover (it takes two individuals called parents and produces two new individuals called the offspring by swapping parts of the parents). The genetic algorithm may eventually find an acceptable solution.

Koza [7] and other researchers have used evolutionary techniques to evolve finite state machines. Their approach consists on encoding the state-transition table defining a finite state machine in the genome. Such approach is straighforward, but the size of the genome tends to be innecesarily huge: in a finite state machine of S states (normally selected a-priori), and O combination of inputs (possible observtions), the genome will have a length of $S \times O$ genes, each gene encoding the next state and the action to be executed.

6 Evolution and learning of finite state machines

The resemblance between a FSM state-table and a state-action value function led us to adopt a different strategy to solve partially observable mazes. Instead of considering a table of state-action values $Q(s, a)$, we used a lookup table similar to the FSM state-table. As shown above, a FSM state-table is accesed by a pair (present-state, current-input), and outputs the following state for the FSM. Now, if we consider a Moore machine, the output action of the finite state machine is only a function of the state (i.e., it does not depend on the inputs).

In our approach, to determine the next-state transition for a given pair (state,input), we use a value-state-like function ($V(s)$-like) that will estimate the current best next state for the agent. Thus, to enable the evolution and learning of finite state machines, our lookup table is accessed by the triplet (present-state, current-observation, estimate-value-of-next-state) and outputs the current best next state for the agent. Each state has a unique associated action, which is taken by the agent after its selection of a state transition. Of course, this implies an a-priori association of actions and states. Therefore, we used evolutionary techniques to solve such design problem.

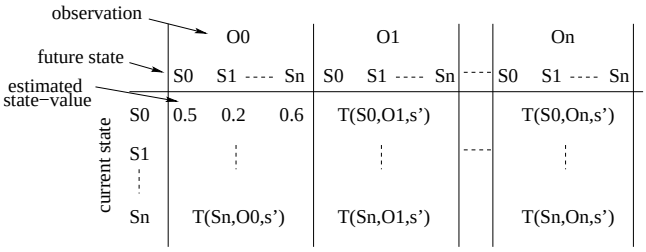

Fig. 2. Lookup table of estimations of the state values $T(s, o, s')$, where s is the current state of the environment, o is the current observation of the environment made by the agent, and s' is the current best next state.

We have denoted the value function by $T(s, o, s')$, where s is the current state of the environment, o is the current observation of the environment made by the agent, and s' is the current best next state. This triplet gives rise to the name of our algorithm. Such a function is implemented by a lookup table as shown in Figure 2. The estimates of the value of the states, used by the agent to choose its next state, are updated using a temporal difference formula as shown in Figure 3.

The algorithm starts by using a population of POP individuals whose genomes associate actions to the states of the finite state machine. The length of the genome is equal to the maximal number of states SMAX multiplied by the number of bits (BITACT) needed to code the set of possible actions (notice that the same action can be associated to several different states). Each individual is tested in the maze for a TRIAL number of trials. During each trial, the agent interacts with the maze and uses the learning algorithm of Figure 3 to update its $T(s, o, s')$ value function, until it reaches the goal or completes a number of STEPS agent performed actions (or number of state transitions). The fitness of an individual is computed by a measure of performance depending on the task (e.g., -1 $\times$ mean number of steps needed to achieve the goal). The evolutionary process is run for up to G generations. In our experiments, both, mutation and crossover operate at the level of the genes, so, for instance, a mutation is defined as a random change of a gene (i.e., the action associated to a particular state) instead of a random binary change.

7 Experimental results

7.1 Case 1: the Santa Fe trail

The Santa Fe trail was proposed by Christopher Langton [7]: an artificial ant is placed on a 32×32 grid of cells, its objective being to collect as many food pellets strewn about as possible in a given number of times steps. The ant starts out facing east, and positioned in the upper left-hand cell of the grid (See figure 4). At each time step it can turn in any direction, walk one cell in that direction,

1. Initialize the $T(s, o, s')$ value function to a constant value, equal to the maximal reward value (optimistic initialization to force exploration), where s and o are the current state and observation, and s' is the next state,
2. Set the agent to the start position, initialize its state ($s^t = s_0$), observe the environment, and take a NOP (no-operation) action,
3. Make $s^{t-1} = s^t$, $o^{t-1} = o^t$
4. Observe the environment and choose as present state, the state with the higher estimate $T(s^{t-1}, o^{t-1}, s^t)$. If several states are possible, choose the one with the smaller index in table $T(s^{t-1}, o^{t-1}, s^t)$.
5. Take the action a, associated (by the genome) to the current state s^t,
6. Update the estimate value $T(s^{t-1}, o^{t-1}, s^t)$ as follows: $T(s^{t-1}, o^{t-1}, s^t) = T(s^{t-1}, o^{t-1}, s^t) + \alpha(r + \gamma T(s^t, o^t, s^{t+1}) - T(s^{t-1}, o^{t-1}, s^t))$, where α is the step size, r is the reward, and γ is a discount reward factor, o^t is the observation at the new state, and s^{t+1} is the next state,
7. Go to step 3 until the state s^t is a terminal state, or a number of permitted steps is reached,
8. Repeat steps 2 to 7 for a pre-defined number of trials.

Fig. 3. Learning loop of the *SOS-algorithm*.

or select the no-operation (NOP) action. The actions are encoded as follows: action 0: NOP, 1: turn left, 2: turn right, and 3: move forward one step. When the agent collects a food pellet, it is reward with positive signal (r=+1). The main difficulty faced by the artificial ant lies in the positions of the pellets, which are not arranged in a simple, consecutive manner. The whole trail contains 89 food pellets.

We have used our *SOS-algorithm* using the parameters shown in the following table:

G	POP	$SMAX$	$TRIAL$	$STEPS$	α	γ
10	100	10	100	500	0.6	0.9

After runing the algorithm for less than 10 generations (in average), each agent, needing less than 30 trials, learns a finite state machine solution of less than 7 states. Typically, the system is able to learn a 5-state finite state machine without using the evolutionary search after 1 to 25 trials. One such example is presented in Figure 4. As a comparison, Koza [7] describes the use of evolutionary techniques to evolve a finite state machine of up to 32 states, whose complete state-transition table is encoded on the genome, to solve the Santa Fe trail problem. We do not have information of the performance of Koza's method, but instead of our 20-bit genome and the effective use of learning, he used a genome of 65 genes (453 bits). Another solution is given by Angeline et al. [1]; they

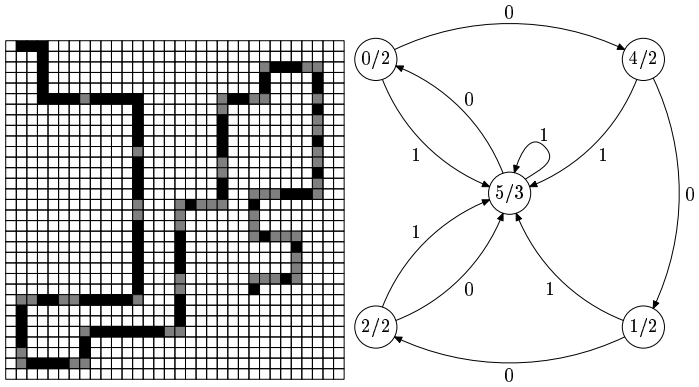

Fig. 4. The Santafe trail and the corresponding finite state machine solution found by the evolutionary and learning method. The numbers on the arrows indicate the input value (0: no food, 1: food), the numbers on the circles indicate the state and the associated action (state/action).

found a 5-state FSM solution after evolving a population of 100 recurrent neural networks for 2090 generations.

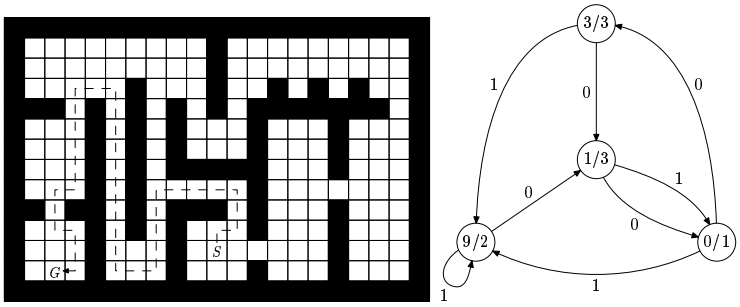

Fig. 5. Complex partially-observable maze and the best state diagram found by the evolutionary and learning method. The numbers on the arrows indicate the input value (0: no obstacle, 1: obstacle), the numbers on the circles indicate the state and the associated action (state/action).

7.2 Case 2: a partially-observable maze

Once our *SOS-algorithm* successfully solved the Santa Fe trail problem, we designed a more complex maze shown in Figure 5. It was motivated by the fact of having multiple paths to reach the goal and a delayed reward signal (i.e., there

is a unique reward signal while reaching the goal). In particular, we used our SOS-algorithm and the parameters shown below:

G	POP	$SMAX$	$TRIAL$	$STEPS$	α	γ
10	200	10	500	500	0.6	0.9

After runing the algorithm for less than 4 generations (in average), each agent needing less than 250 trials, learns a finite state machine solution of less than 6 states. Typically, the system is able to learn a 9-state finite state machine which guides the robot to the goal in 116 steps, without using the evolutionary search, a 7-state finite state machine which guides the robot to the goal in 96 steps, after the second generation, and a 4-state finite state machine which guides the robot to the goal in 92 steps, after the third generation. This problem was very interesting because of the diverse solutions we obtained: for instance, some solutions needed few states, but needed more steps to reach the goal than other solutions were the FSM was larger but guided the agent through the shortest path. Finally, we succeded in optimizing both aspects. Our next testbed is a maze with several possible paths to reach the goal we found on the literature.

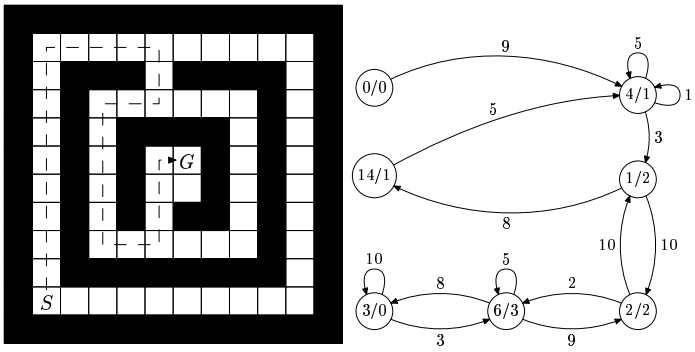

Fig. 6. Wiering-Schmidhuber labyrinth and the best state diagram found by the SOS-algorithm. The numbers on the arrows indicate the input value computed as a funtion of its four sensors (e.g., 0: no obstacle, 1: obstacle to the right, 2: obstacle to the north, 6: obstacle to the north and to the west, etc.), the numbers on the circles indicate the state and the associated action (state/action).

7.3 Case 3: Wiering and Schmidhuber maze

Wiering and Schmidhuber learns finite sequences of (memoryless) reactive policies using an implicit memory of some of the previous observations [20]. They defined a multi-agent system, where each agent has a Q-table, and an HQ-table that serves as a control transfer unit (except for the last agent which only has a

Q-table). The Q-table stores estimates of actual observation/action values and is used to select the next action. The HQ-table stores estimated subgoal values and is used to generate a subgoal once the agent is made active. To test their algorithm, they defined the 12×12 maze shown in Figure 6. The system has to discover a path leading from start position S to goal G. There are four actions: go west, go north, go east, go south, and 16 possible observations. Although there are 62 possible agent positions, there are only 9 highly ambiguous inputs. To solve such a maze, they used groups of 3,4,6, 8 and 12 agents using HQ-learning [20]. A reward was only delivered when the goal was reached. Within 20,000 trials all systems almost always found near-optimal solutions. In most cases a 30-step solutions were found, though in 1 out of 8 cases, the optimal 28-step solution was found. The better systems performed using between 3 and 6 agents.

Our evolving and learning approach use the paramters shown below:

G	POP	$SMAX$	$TRIAL$	$STEPS$	α	γ
50	50/200	15	250	1000	0.6	0.9

After runing the algorithm using a population of 50 individuals, in less than 10 generations (in average), each agent, needing less than 250 trials, learns a finite state machine solution of 9 states, which reaches the goal in 28 steps (i.e., the optimal solution). Typically, if the system is run using 200 individuals per generation, it is able to learn a 9-state finite state machine without using the evolutionary search after only 127 learning trials. Figure 6 shows the best solution found by the SOS-algorithm. With that 7-state finite state machine, the agent learns to reach the goal in 28 steps after running the evolutionary search on a population of 100 individuals for 2 generations.

8 Concluding remarks

In this work we show how to effectively use the evolution and learning of finite state machines to solve complex partially observable problems. Instead of using evolutionary techniques by encoding in the genome the whole state-transition table defining a finite state machine, we provide an approach with a very compact genome, where learning by interaction completes the design of the finite state machine. We compared our results with other approaches in two different tasks, the well-known Santa Fe trail and a 12×12 maze proposed in [20], and provide an intermediate test on a complex maze proposed by one of the authors. The results were very impressive given the compact representation of the genome we used, and the very few learning steps needed by the learning by interaction algorithm, to find optimal solutions.

Our approach (the *SOS-algorithm*) does not solve the curse of dimentionality problem, but provides an efficient algorithm for the design of finite state machines. It is the result of taking into account the normal procedure for designing finite state machines by hand, and the use of artificial evolution and learning techniques. Given that any procedural program can be implemented by a finite

state machine, our approach is closely related to genetic programming (GP) [7]. Indeed, a predecesor of GP, evolutionary programming (EP), was proposed in the 1960's by L. Fogel [4], and one of the premises of his work was the automatic development of programs by means of finite state machine evolution [6].

Although evolution was not essential in solving the problems discussed in this paper, in a more general way, evolution enables the solution of the problems using smaller populations. Indeed, without evolution, the learning part of the SOS-algorithm can solve these problems only if the initial population contains all the needed actions, in the sufficient number (i.e., when a maze-solution uses the x action in n different states, the learning part would need an individual with n times the x action in its genome). This condition can only be respected, for any problem, by using big populations.

References

1. P.J. Angeline, G.M. Saunders, and J. Pollack. A evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, January 1994.
2. A.R. Cassandra. *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, Brown University, 1998.
3. Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proc. 15th International Conf. on Machine Learning*, pages 118–126. Morgan Kaufmann, San Francisco, CA, 1998.
4. L.J. Fogel and A.J. Owens. *Artificial Intelligence through Simulated Evolution*. John Wiley and sons, New York, 1966.
5. M. Hauskrecht. *Planning and Control in Stochastic Domains with Imperfect Information*. PhD thesis, MIT, Cambridge, MA, 1997.
6. C. Jacob. *Illustrating Evolutionary Computing with Mathematica*. Morgan Kaufmann, San Francisco, 2001.
7. J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.
8. J.L. Lin and T.M. Mitchell. Reinforcement learning with hidden states. In J-A. Meyer, H.L. Roitblat, and S.W. Wilson, editors, *From Animals to Animats: Proceedings of the Second International Conference on Simulation of Adaptive Behavior (SAB92)*, pages 281–290, 1992.
9. N. Meuleau, L. Peshkin, K.E. Kim, and L.P. Kaebling. Learning Finite-State Controllers for Partially Observable Environments. In *Proceedings of the Conference on Uncertainty and Artificial Intelligence UAI'99*, Stockholm, Sweden, 1999.
10. Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1992.
11. A. Pérez-Uribe. *Structure-Adaptable Digital Neural Networks, Ph.D Thesis 2052*. PhD thesis, Swiss Federal Institute of Technology-Lausanne, 1999.
12. D. Precup, R.S. Sutton, and S. Singh. Theoretical Results on Reinforcement Learning with Temporally Abstract Behaviors. In *Proceedings of the 10th European Conference on Machine Learning ECML'98*, pages 382–393, Chemnitz, Germany, 1998.
13. M.B. Ring. *Continual learning in reinforcement environments*. PhD thesis, The University of Texas at Austin, August 1994.

14. E. Sanchez. An Introduction to Digital Systems. In D. Mange and M. Tomassini, editors, *Bio-Inspired Computing Machines: Toward Novel Computational Ma chines*, pages 13–48. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.
15. Satinder P. Singh, Tommi Jaakkola, and Michael I. Jordan. Reinforcement learning with soft state aggregation. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 361–368. The MIT Press, 1995.
16. R. Sun and T. Peterson. Multi-agent reinforcement learning with bidding for segmenting action sequences. In J-A. Meyer, A. Bethoz, D. Floreano, D. Roitblat, and S. Wilson, editors, *From Animals to Animats: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior (SAB2000)*, pages 325–332, 2000.
17. R.S. Sutton. Learning to predict by the methods of Temporal Differences. *Machine Learning*, 3:9–44, 1988.
18. R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, 1998.
19. S.D. Whitehead and D.H. Ballard. Learning to Perceive and Act by Trial. *Machine Learning*, 7(1):45–83, 1991.
20. M. Wiering and J. Schmidhuber. HQ-Learning. *Adaptive Behavior*, 6(2), 1997.