Learning Finite State Controllers from Simulation

Matteo Leonetti[1], Luca Iocchi[1], and Subramanian Ramamoorthy[2]

[1] Department of Computer and System Sciences, Sapienza University of Rome, via Ariosto 25, Rome 00185, Italy
[2] School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh EH8 9AB, United Kingdom

Abstract. We propose a methodology to automatically generate agent controllers, represented as state machines, to act in partially observable environments. We define a multi-step process, in which increasingly accurate models - generally too complex to be used for planning - are employed to generate possible traces of execution by simulation. Those traces are then utilized to induce a state machine, that represents all *reasonable* behaviors, given the approximate models and planners previously used. The state machine will have multiple possible choices in some of its states. Those states are *choice points*, and we defer the learning of those choices to the deployment of the agent in the real environment. The controller obtained can therefore adapt to the actual environment, limiting the search space in a sensible way.

Keywords: Agent programming, Planning, Reinforcement Learning, Partial Observability

1 Introduction

Decision making in *real-world* environments is characterized by uncertainty at many and different levels. Part of the uncertainty can be captured by models for planning under partial observability, to which a great deal of attention has been payed in recent years. A paramount source of uncertainty lies in the assumptions behind such models themselves, especially if the problem has not been synthesized, but arises from an existing application. This is true regardless of how accurately the model has been designed or learned. Such uncertainty cannot be dealt with at planning time, and requires to monitor the execution in order to identify any discrepancies between what is expected and what is perceived [15]. Consider, for instance, a non-stationary environment incorrectly modeled as stationary. Even after having learned a good approximation of the initial environment, when the environment and the model drift apart, plans may start failing. Re-planning in the faulty model is of no use.

Reinforcement Learning (RL) can both learn models from data, or avoiding models altogether. The amount of experience necessary to learn complex behaviors is still overwhelming in robotic applications, and any naive implementation

is bound to limited scalability. Moreover, any learning mechanism involved - such as function approximation - is characterized by a bias. On the one hand, a bias necessary for any learning to take place, and on the other hand it may let the representation diverge from what it aims to represent. RL has been extensively studied in the case of full observability, and the techniques developed have been applied to a number of robotic scenarios. Learning and optimization is successfully carried out on low-level control procedures, since the variables that statistically determine what needs to be predicted are either readings of proprioceptive sensors, or are, more in general, available to the perceptive system. High-level decision making, on the other hand, has to cope with aspects that cannot be observed, and often have no obvious model. A typical example of this category of problems are those that involve other agents.

Hierarchical RL (HRL) [1] allows the designer to provide structure to the policies searched, constraining the exploration in fully observable domains. This is a fundamental aspect for real-world applications, as time is a strictly limited resource, and robotic agents are subject to wearing and tearing. Such structure can be provided either in the form of an hierarchy of tasks [6], a state machine [14], or a Lisp program [11]. The automatic definition of the aforementioned structures is still an open problem, and is usually carried out by hand.

In this paper, we propose a methodology to automatically generate agent controllers, represented as state machines, combining several ideas developed in the literature of planning under partial observability and reinforcement learning. We define a multi-step process, in which increasingly accurate models - generally too complex to be used for planning - are employed to generate possible traces of execution by simulation. Those traces are then utilized to induce a state machine, that represents all *reasonable* behaviors, given the approximate models and planners previously used. The state machine will have multiple possible choices in some of its states. Those states are *choice points*, and we defer the learning of those choices to the deployment of the agent in the real environment. Thus, the exploration in the real environment is constrained by the controller - similarly to HRL, but applied to a partially observable domain - which is the composition of the available low-level procedures, the rationality of a planner, and a model of the problem to solve.

2 Background and Related Work

This work lies at the intersection of reinforcement learning and planning in partially observable domains. It also makes use of induction of state machines. We assume the reader to be familiar with the basic definitions of RL [19], and summarize the background behind symbolic representation of planning problems, and finite-state automata.

2.1 Symbolic action theories

Reasoning about actions comprises the use of formal systems to describe dynamic systems. Regardless of the actual logic employed, all theories of actions share a few elements [17], described in the following.

The state space is represented through *fluents*, propositions whose truth values depend on the current *situation*. A *situation* is the result of the application of actions to some initial assignment of truth values. Such an assignment determines the *initial situation*. Actions are described by pre-conditions, that is formulas that must hold for the action to be applicable, and post-conditions (or effects), that is formulas that hold after an action has been executed. *Axioms* are formulas added to the theory in order to enforce constraints that must always hold. Physic laws, for instance, enforce that moving an object causes all the objects on top of it to be moved as well. Such an atemporal formula (differently from a fluent) must hold in every situation. Action specifications (in terms of pre- and post-conditions) and axioms determine the way in which fluents change, therefore the dynamics of the system represented.

A *goal* is represented as a formula, and a situation fulfills the goal when the truth assignment for that situation models the goal formula.

2.2 Regular languages and automata

Let Σ be a finite, non-empty, set of symbols called an *alphabet*. A *string* (or *word*) is a sequence of symbols, and a set of strings is called *language*. A *deterministic finite-state automaton* is a tuple $DFA = \langle Q, F, \Sigma, \delta, q_0 \rangle$, where:

- Q is a finite set of states
- $F \subseteq Q$ is a set of *final* states
- $\delta : Q \times \Sigma \to Q$ is the transition function
- q_0 is the initial state

Given a finite string $w = w_0 w_1 \dots w_n$, a DFA generates a unique trajectory $\langle q_0, w_0, q_1, w_1, \dots, q_n, w_n \rangle$. Let Σ^* denote the set of all finite sequences of symbols in Σ. A DFA *accepts* a language $L \subseteq \Sigma^*$ iff $\forall w \in L$ s.t. q_n is the last state of the trajectory produced by the DFA over w, and $q_n \in F$. A DFA *recognizes* a language L, if it accepts all and only the strings in L. A DFA that accepts a language L may also accept strings that do not belong to L, while a DFA that recognizes L must not. A language L is regular iff there exists a DFA that accepts L.

2.3 Controllers and partial observability

The work most closely related to our own follows two lines: hierarchical RL, and automated planning for partially observable domains.

The methods for hierarchical RL (MAXQ [6], HAM [14], and ALisp [11]) all assume full observability. In this work, we assume not only partial observability but also partial information, as we do not require a specification of the state

space, let alone the transition function. Our approach shares with those methods the idea of constraining the space of possible solutions, by exploiting some structure that carries information about the task to solve (not the domain). We do require an intermediate model, but subsequent learning in the real domain alleviates the structural uncertainty behind it. Parr mentions the possibility for HAMs to be extended to deal with partially observable domains [14], but to the best of our knowledge it had not been done yet. Moreover, none of the mentioned methods provide a way to derive the controlling structure (state machine, task hierarchy, or Lisp program) automatically. We do so by learning from both simulation and the actual domain.

For what concerns automatic planning, in the last few years several authors have pointed to state machines as effective ways to represent controllers in partially observable domains. They are more compact than *contingent trees*, and policies over *belief states*, and allow to generalize to some extent [3]. For some particular cases, it is possible to derive a controller by logic [3, 4, 7]. Although, in general, planning under partial observability is infeasible (at least EXP-hard) [16]. About making use of existing execution traces, Srivastava et al.[18] proposed a method to compute generalized plans. It aims at identifying parts of given trajectories that might be able to solve the planning problem starting from intermediate positions in an existing plan. The result is an extremely expressive structure with sensing and loops. The main difference between our approach and all the planning ones is that we do not require the model to be exploitable for *computing* solutions, but just to *verify* that a given trajectory is indeed a solution. Moreover, with respect to Srivastava's work we also provide a way of generating such trajectories.

Interestingly, the work on planning can be integrated in our framework. The initial decision maker may be based on a planner, and let act in a simulated environment as will be formally described in the following.

3 Generating Controllers from Simulation

In this section we define the process to generate a finite state controller for a given problem under partial information. In the following we make use of a running example to explain our method. The example is a simple partially observable problem, that has been used as a benchmark in several RL publications [9, 10].

Consider the partially observable problem of Figure 1 (c). The agent starts from any of the squares and has to reach the top-right corner. The actions available are to `move` one step `north`, `south`, `east`, and `west`, and their effect is deterministic. The agent, however, can only observe at any given time the eight squares around its current position.

3.1 Environments, problems, and controllers

We begin with the definition of a *dynamic environment* $\mathcal{E} = \langle \mathcal{A}, \mathcal{O}, \mathcal{S}, \mathcal{I}, \Delta, \Omega \rangle$ in which:

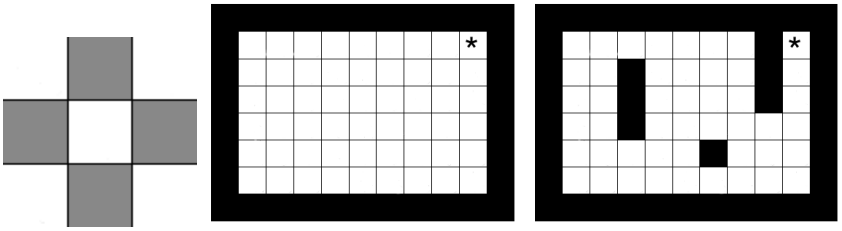


Fig. 1. The different representations at different stages. (a) The initial decision maker takes into account only obstacles in the four directions; (b) a model without obstacles; (c) the grid of the actual problem

- $\mathcal{A}$ is a finite set of actions
- $\mathcal{O}$ is a set of observations
- $\mathcal{S}$ is a set of states
- $\mathcal{I} \subseteq \mathcal{S}$ is a set of initial states
- $\Delta : \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation
- $\Omega : \mathcal{S} \to \mathcal{O}$ is the observation function

In this work, we assume to have available only $\mathcal{A}$ and $\mathcal{O}$, while all the other components of the environment are unknown.

A *generalized planning problem* over an environment is a tuple (defined by Bonet et al. [3]) $P = \langle F, I, A, G, R, O, D \rangle$ where:

- F is a set of primitive fluents
- I is a set of F-clauses representing the initial situation
- A is a set of actions
- G is a set of literals representing the goal situation
- R is a set of non-primitive fluents
- $O \subseteq R$ is the set of axioms defining the fluents in R

In the literature [3, 4], as mentioned in Section 2, this problem specification is used to derive, by logic, a controller. Due to the uncertainty on the definition of the environment, we only assume to have A, and G. That is, we assume to know the available actions and to be able to recognize the goal states. Hence, such a problem cannot be solved directly by any planner, nor learned as a POMDP (no description of the underlying state space is given). Such a situation is described as *partial information*, which includes partial observability. Being a problem on a real environment, however, experience can be gathered by acting in it.

A *controller* is a tuple $\mathcal{C} = \langle Q, A^*, O^*, \delta, q_0 \rangle$ where Q is a set of states, $q_0 \in Q$ is the initial state, and A^*, O^*, and δ are the finite set of actions, set of observations, and transition relation respectively. The (partial) transition relation δ maps pairs $\langle q_i, o_i \rangle$ of controller states and observations into actions, and next states q_{i+1}. The controller is *deterministic* if given a pair $\langle q_i, o_i \rangle$ the action, and consequently the next states, are uniquely determined. A deterministic

controller C over an environment $\mathcal{E}$ produces, from each initial state s_0, a single trajectory $t_C(s_0) = \langle o_0, q_0, o_1, q_1, \ldots, o_f, q_f \rangle$. A non-deterministic controller, on the other hand, can produce a set of trajectories that we denote with $T_C(s_0)$.

A trajectory $t(s_0)$ is a solution of P from an initial state s_0 iff the terminating observation o_f is such that $o_f \models G$. We are assuming that $o_f \models G \Rightarrow s_f \models G$, that is, if an observation fulfills the goal specification, the underlying, unobservable, state is a goal state. A deterministic controller *solves* a problem P over an environment $\mathcal{E}$ iff each trajectory $t_C(s_0)$, from each initial state s_0, is a solution from s_0. We say that a non-deterministic controller C *can solve* a problem P if $\exists t \in T_C(s_0)$ such that t is a solution from s_0.

Finally, we define a *restriction* of a problem P to $\hat{I} \subseteq I$ as the sub-problem $P(\hat{I}) = \langle F, \hat{I}, A, G, R, O, D \rangle$.

3.2 Simulators

The specification of the real environment is severely limited by uncertainty. We assume that such an environment exists, that we can perform a set of actions on it, and that we are able to perceive some observations. Traditional RL would require at least a description of the state space, although no unique definition is in general available from the problem specification. In fact, designing a representation is a delicate task, that requires a great deal of experience and understanding of the problem at hand.

In this work we take a different stand. We assume the existence of another environment $\mathcal{E}'$ on which we can define a problem P'. Informally, $\mathcal{E}'$ is a simulator for $\mathcal{E}$, as those commonly available in robotics. The characteristic of a simulator is to be a model that provides an approximation of the environment $\mathcal{E}$, that is usually too complex to be used for planning. In such a model, however, experience can be gathered much more cheaply than in $\mathcal{E}$. We do not define any direct relationship between $\mathcal{E}$ and $\mathcal{E}'$, we shall rather establish one through controllers.

Although we have a complete specification of both $\mathcal{E}'$ and P', we only use them through simulation, that is, to generate trajectories. P' can be partially observable, but does not necessarily have to. Furthermore, we assume the existence of a *decision maker* that can solve P' in $\mathcal{E}'$. For finite observation spaces, such a decision maker always exists, and in the worst case it would just produce all possible sequences of observations and actions of finite length, thus solving P' if a finite solution exists. The aim of the decision maker, however, is to employ some *rationality*, so that exploratory moves that are clearly wrong can be avoided. The designer may embed in the decision maker any previous knowledge about possible solutions.

The decision maker is deployed in $\mathcal{E}'$ to generate trajectories $t'(s'_0)$ that are solutions to P'. The observations in such trajectories are removed, as they are, in general, not related to the observations that the agent would experience in the real environment. Later, a new mapping from observations to actions will be learned in $\mathcal{E}$, exploiting a controller derived in $\mathcal{E}'$.

Considering each action in A' as a symbol, the set of trajectories that are solutions to P', from which observations are removed, form a *language*. We

induce a finite deterministic automaton $\mathcal{C}' = \langle Q', A', \emptyset, \delta', q'_0 \rangle$ that accepts such a language. Note how this is equivalent to a controller with an empty observation set. There is a vast literature about inducing such acceptors [5, 8], and we mention a few methods later in the example.

Finally, we expand the edges of $\mathcal{C}'$ to accommodate the observations of P. We define a controller $\hat{\mathcal{C}} = \langle Q', A', O, \hat{\delta}, q'_0 \rangle$ obtained from $\mathcal{C}'$ such that for each observation $o \in O$, $\hat{\delta}$ connects a state q'_i to a state q'_j when observing o and by executing $a' \in A'$, if and only if δ' connects q'_i to q'_j by executing a'.

If the controller $\hat{\mathcal{C}}$ obtained through the process just described can solve a reduction of P in $\mathcal{E}$ we say that the composition of the decision maker, $\mathcal{E}'$, P', and the method used to induce the automaton is *admissible*. This can be verified by executing $\hat{\mathcal{C}}$ in $\mathcal{E}$.

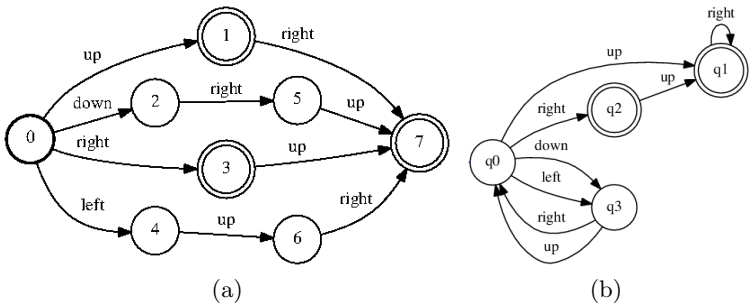
Step 1: definition of an initial decision maker We begin with the definition of a simple decision maker able to act in the domain presented at the beginning of Section 3. To represent high-level procedures that take into account local variables, we define four non-atomic actions with which the agent moves in each direction until it encounters an obstacle. We denote these non-atomic actions with `up`, `down`, `left`, and `right`, with obvious effects. The precondition for each of those actions is the absence of an obstacle in the direction in which the agent would go.

Our decision maker chooses randomly among the non-atomic actions available until it reaches the goal. To add a bit of *rationality*, from which the subsequent phases can benefit, it also avoids the action opposite to the one just chosen. Thus, for instance, if the last action was `right`, the next action cannot be `left`. This because otherwise the same effect of moving right and then left could have been achieved by moving left in the first place. In order to act accordingly to the decision maker just defined, the agent needs only to observe the four squares in the cardinal directions, as represented in Figure 1 (a), and to recognize the goal state.

Step 2: application of the decision maker to a model At this stage we assume to have an approximated model of the environment $\mathcal{E}'$ and the problem to solve P', but too complex to be exploited by a planner.

In our simple example, we assume the model in Figure 1 (b), that is the same grid as the real domain but without obstacles. We let our agent act on such a model for a number of episodes, and record the sequences of actions that led to the goal state within a certain threshold. In this case we ran 1000 episodes, and accepted any trajectory able to lead from the random initial state to the goal in at most 2 times the Manhattan distance. Taking all trajectories regardless of a measure of performance is possible but likely to produce extremely sub-optimal plans. The set of sequences of actions generated is the following: { ⟨ `up` ⟩, ⟨ `right` ⟩, ⟨ `right`, `up` ⟩, ⟨ `up`, `right` ⟩, ⟨ `down`, `right`, `up` ⟩, ⟨ `left`, `up`, `right` ⟩ }.

The output of the this second step is a *deterministic finite state automaton* that accepts the strings produced by the simulation. By standard techniques for



(a) (b)

Fig. 2. Two finite state automata that both solve the given problem

automaton composition and minimization [12] we obtain the state machine in Figure 2 (a), which recognizes the language of the strings produced. A possible alternative consists in using algorithms like Bierman [2] and RPNI [13] providing them both positive and negative examples. Negative examples (sequences the automaton must not accept) may be taken from traces that fail to reach the goal, and sub-sequences of both successful and failing traces (as only complete traces must be accepted), unless they have reached the goal at least once. For instance, ⟨`right`⟩ is a sub-sequence of ⟨`right`, `up`⟩, but must be taken as a positive example, as it reached the goal at least once. On the other hand, both ⟨`down`⟩ and ⟨`down`, `right`⟩ are sub-sequences of ⟨`down`, `right`, `up`⟩ and must be taken as negative examples, as they never reached the goal themselves. The resulting machine is shown in Figure 2 (b). Note that the language accepted by automaton (a) is a sub-set of the language accepted by automaton (b), and the latter is a generalization of the strings (possible plans) provided as examples.

3.3 Reinforcement learning on controllers

The controller $\mathcal{C}$, obtained through simulation and automaton induction, provides at the same time a constraint to the possible behaviors, and a partial specification of a solution for a problem that could not otherwise be solved. If such an automaton is deterministic no further improvement is possible, and it constitutes a completely specified solution to P (with no performance guarantees). More interestingly, if such a controller is non-deterministic, learning in the real environment can determine (and be limited to) the behavior at choice states. In order to generate a non-deterministic controller $\mathcal{C}$, either the decision maker must be able to produce more than a solution trajectory, from at least some initial states, or the induction method must be able to generalize over sample trajectories. The two options are not mutually exclusive.

Non-deterministic controllers may help alleviate the structural uncertainty behind the model $\mathcal{E}'$, without having to give up on modeling completely. The trajectory that proved best in $\mathcal{E}'$ is not necessarily the best one in $\mathcal{E}$ too, but

limiting the search to a set of *reasonable* trajectories in the real environment is of capital importance.

In this section we define a controllable stochastic process from a controller $\mathcal{C}$ in order to apply reinforcement learning to it. As previously mentioned, if $\mathcal{E}$ is fully observable such a definition already exists, and HAMs [14] can be used to learn the optimal policy compliant with the constraints imposed by the controller. We define a decision problem $DP = \langle S_d, A, D, \rho \rangle$ such that:

1. $S_d = Q' \times O$
2. D connects a state $\langle q'_i, o_i \rangle$ to $\langle q'_j, o_j \rangle$ through action $a \in A$, if and only if $\hat{\delta}(q'_i, o_i) = q'_j$
3. $\rho : S_d \times A \to \Re$ is a reward function

D is a probability distribution over $S_d \times A \times S_d$. By saying that D *connects* $s_i \in S_d$ to $s_j \in S_d$ through $a \in A$, we only mean that the probability to go from s_i to s_j by executing a is not necessarily zero. If the two states are not connected by D, on the other hand, such a probability is set to zero. In general, however, the probabilities not set by definition to zero are unknown, as they depend on the probability to land on a state that returns a given observation in the real environment.

The decision problem DP may or may not be Markovian, depending on whether the memory embedded in the controller is sufficient to statistically justify the reward. This problem is very well understood in the literature of POMDPs. Therefore, a learning algorithm must be chosen carefully, preferring methods that search in policy space if the decision problem proves to be Non-Markovian.

This general definition of a decision problem in the real environment allows the designer to implement a different *abstraction* at each choice state. This may reduce the expansion at point 2 of the definition of DP if the observation space is too large or continuous. If the underlying environment is fully observable, therefore having one observation per state, DP is exactly equivalent to a HAM flattened over the underlying MDP.

The controller might get stuck because of discrepancies between $\mathcal{E}'$ and $\mathcal{E}$, such that actions that were possible in the former happen to be not applicable in the latter. In such a case, the agent has to improvise reverting to the decision maker, or in the worst case to a random behavior, and hope for the best. Any successful trajectory generated in the real environment can be included in the controller (re-inducing it), modifying the structure on the fly and avoiding such issues in future executions.

Once the learning phase has terminated, the reward function can be used to compare different controllers, along with the particular reduction that they solve. In general a maximal reduction (possibly the whole initial problem itself) is preferable. If from some initial states a particularly high reward can be obtained, however, a controller that solves a smaller reduction including those states might be better. The reward function can also be used in the previous phase to exclude trajectories that solve P' with poor performance.

The whole process can in principle be iterated, with the newly generated controller acting as one of the available procedures, for a problem at a higher level of the hierarchy.

Step 3: learning in the real environment If the domain were fully observable, either one of the machines derived at the former step could be used to learn as a HAM on the underlying MDP. The automata are deterministic as acceptors of the sequences of actions, since each state never has two outgoing edges that recognize the same symbol. From the point of view of the agent, however, each state with more than one outgoing edge is a *choice state*, since more than one action is possible from it. Those remaining choices, and only those, will be learned directly in the real environment. We performed the transformation previously defined of the automata into decision processes, and used an algorithm for searching in policy space to learn in choice states. We define the problem as episodic imposing a maximum length of 30 actions. The reward function returns -1 after each atomic action is executed. Therefore a non-atomic action that moved the agent for n squares would receive a reward of $-n$. Finally, the problem is considered undiscounted. Figure 3 shows the results of learning. Both

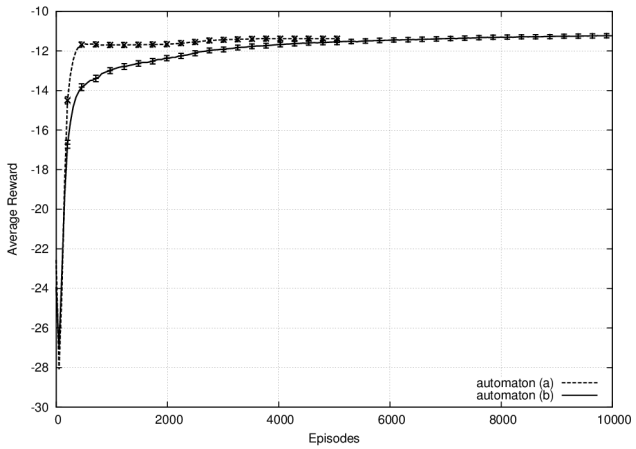


Fig. 3. Results of learning for both automaton

automata reach the same level of performance, although automaton (a) is able to learn faster. Note how the number of episodes taken by automaton (a) to reach the optimum value compatible with its structure is extremely low. With 46 initial states and 2 to 4 actions available per state, about 300 episode is close to performing each action in each initial state only once. Automaton (a), however, is not able to solve the problem from the initial state right on top of the left-most obstacle. From that state the two available paths both run into walls.

Automaton (b), on the other hand, is able to execute the sequence ⟨ left, right, down, right, up ⟩ that is sub-optimal but reaches the goal.

4 Conclusions and Future Directions

We proposed a method to generate agent programs, in the form of state machines, by combining different components: an initial decision maker, available to the designer to include any previous knowledge about the task; a simulator, that is, a model too complex to be used for planning, but from which possible trajectories can be extracted; the induction of an automaton that accept the extracted trajectories; and finally reinforcement learning, on the derived state machine, directly in the real domain. Simulators are commonly employed, but are rarely integral parts in the development of agent programs.

Constraining the final learning, through the interaction of all those components, significantly limit the search space in the real domain. This has a twofold effect: on the one hand the optimal behavior might be lost, and only the best behavior compliant with the final state machine can be learned. On the other hand, limiting the exploration to those trajectories that are related to both the rationality of the designer and a model of the actual task, significantly reduces the number of episodes required to learn. In real-world environments, this trade-off inevitably arises and sensible ways of limiting the search space are necessary.

Different automata may be induced by different methods. The obtained automaton also depends on the positive and negative sample trajectory provided. Thoroughly considering the available options in the future can make the method increasingly effective.

An interesting future direction consists in replacing the simulator with a human, that drives the agent in the real environment. In this case, sequences of both actions and observations might be recorded, and a state machine that mimics and generalizes the behavior of the human be induced.

State abstraction is another aspect that may be considered in the future. The automaton allows to specify a different abstraction at each choice state, partially decoupling the problem of effectively reacting to observation, with back-chaining action sequences in order to reach the goal. Since plausible sequences are already encoded in the machine itself, learning an abstraction should result simpler.

Finally, adaptivity to the real environment may be achieved by training the agent on different sub-problems, and inducing the machine with the union of their trajectories. The resulting controller may be able to solve all the different problems, and discover from the real environment which one it is actually facing. For instance, the agent might be trained in simulation against a number of different adversaries, learning different tactics. When playing against a specific agent, it might adapt to the best of the tactics available.

References

1. A. G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, 2003.

2. A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *Computers, IEEE Transactions on*, 100(6):592–597, 1972.
3. B. Bonet, H. Palacios, and H. Geffner. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. of ICAPS*, pages 34–41, 2009.
4. G. De Giacomo, F. Patrizi, and S. Sardina. Agent programming via planning programs. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 491–498. International Foundation for Autonomous Agents and Multiagent Systems, 2010.
5. C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recogn.*, 38:1332–1348, September 2005.
6. T. G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.
7. Y. Hu and G. De Giacomo. A generic framework and solver for synthesizing finite-state controllers. In *Proc of the AAAI 2011 Workshop on Generalized Planning (GenPlan'11)*, 2011.
8. M. Leucker. Learning meets verification. In *Proceedings of the 5th international conference on Formal methods for components and objects*, pages 127–151. Springer-Verlag, 2006.
9. M. L. Littman. Memoryless policies: Theoretical limitations and practical results. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 238–247, 1994.
10. J. Loch and S. Singh. Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 323–331, 1998.
11. B. Marthi, S. J. Russell, D. Latham, and C. Guestrin. Concurrent hierarchical reinforcement learning. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 779–785, 2005.
12. M. Mohri. Minimization algorithms for sequential transducers. *Theoretical Computer Science*, 234(1-2):177–201, 2000.
13. P. Oncina, J.; Garca. *Identifying regular languages in polynomial time*, chapter -. World Scientific Publishing, 1992.
14. R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, pages 1043–1049, 1998.
15. O. Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.
16. J. Rintanen. Complexity of planning with partial observability. In *Proc. of ICAPS*, volume 4, pages 345–354, 2004.
17. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2005. Third Edition.
18. S. Srivastava, N. Immerman, and S. Zilberstein. Merging example plans into generalized plans for non-deterministic environments. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 1341–1348. International Foundation for Autonomous Agents and Multiagent Systems, 2010.
19. R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.