Automatic Generation and Learning of Finite-State Controllers

Matteo Leonetti[1], Luca Iocchi[2], and Fabio Patrizi[2]

[1] Department of Advanced Robotics, Istituto Italiano di Tecnologia, via Morego, 30, 16163 Genova
[2] Department of Computer, Control, and Management Engineering, Sapienza University of Rome, via Ariosto 25, 00185 Rome

Abstract. We propose a method for generating and learning agent controllers, which combines techniques from automated planning and reinforcement learning. An incomplete description of the domain is first used to generate a non-deterministic automaton able to act (sub-optimally) in the given environment. Such a controller is then refined through experience, by learning choices at non-deterministic points. On the one hand, the incompleteness of the model, which would make a pure-planning approach ineffective, is overcome through learning. On the other hand, the portion of the domain available drives the learning process, that otherwise would be excessively expensive. Our method allows to adapt the behavior of a given planner to the environment, facing the unavoidable discrepancies between the model and the environment. We provide quantitative experiments with a simulator of a mobile robot to assess the performance of the proposed method.

Keywords: Agent Programming, Planning, Reinforcement Learning, Controller Induction

1 Introduction

Decision making in domains for which no detailed model description of the underlying system is available is one of the most challenging problems in Artificial Intelligence. This condition often characterizes applications related to cognitive robots acting in a real environment. Two different perspectives have been applied to this problem. On the one hand, the use of automatic planning techniques assume the presence of a suitable model of the domain. Solutions can be found, by planning, only if they are represented or characterized by this model. On the other hand, the use of model-free machine learning techniques prove to be too sample inefficient (in the number of trials) to converge in real environments, even for simple tasks.

The integration of the two perspectives allows to overcome these main difficulties. Planning-based methods provide an effective way to properly reduce the solution space, while learning-based methods allow for exploring such a solution space. The resulting behavior improves from experience, and can take into account non-modeled aspects of the task.

More specifically, in this paper, we consider the following problem: an agent has to optimize its performance over a complex task. A model of the task and the environment is given with limited accuracy (i.e., not all the relevant properties may be considered in the model). The setting is such that: 1) a pure planning method would not be effective, because: i) it may not be able to provide an effective solution since the model is not sufficiently accurate; ii) it returns a strategy with constant average performance (i.e., it will not adapt to the real environment and improve over time); 2) a pure learning method will not be effective, because the number of experiments needed for convergence to either a "good" or optimal solution is not feasible for the system.

In this paper we propose a method that allows for improving the performance over time with limited sample complexity, so to be suitable also for real systems (like robots) acting in a real environment.

Although many different representations can be used for the behavior of an intelligent agent, in this paper we focus on finite state machines, because they can realize compact controllers and have been largely used for this purpose. As already mentioned, however, we consider a scenario in which both automatic generation from a model, and learning from data separately present non-optimal performance due to the complexity of the task. We present a combined method that is able to generate and learn a finite state machine representing the controller of a complex system. It exploits both an approximated and incomplete model of the domain and experience on the field. In particular, a planning-based technique is used to generate a set of possible solutions to the problem given the approximated model, while learning is applied to refine and adapt these solutions to the actual environment.

2 Background and Related Work

In this section, we introduce the notation and provide the background, behind planning and reinforcement learning, required in the this paper. We also describe the methods of Hierarchical Reinforcement Learning closer to our own, to clarify the novel aspects.

2.1 Planning Problems

A *non-deterministic planning domain* is a tuple $\mathcal{D} = \langle A, Prop, S, s_0, T \rangle$, where: (a) A is the finite set of actions; (b) $Prop$ is the finite set of propositions; (c) $S \subseteq 2^{Prop}$ is the finite set of states; (d) $s_0 \in S$ is the initial state; (e) $T : S \times A \mapsto 2^S$ is the transition function. Each state s implicitly defines a propositional interpretation $\nu_s : Prop \mapsto \{\top, \bot\}$ s.t. $\nu_s(p) = \top$ iff $p \in s$. Given a propositional formula φ (over $Prop$), a state s is said to *satisfy* φ, written $s \models \varphi$, iff ν_s is a model of φ. A *goal* G is a set of states, i.e., $G \subseteq S$, possibly represented as a propositional (goal) formula φ, in which case $G \doteq \{s \in S \mid s \models \varphi\}$. When $T(s, a) \neq \emptyset$ we say that action a is *executable* in s. If $s' \in T(s, a)$, s' is called a *successor* of s on a. A $\mathcal{D}$*-trace* is a sequence $\tau = s_0, s_1, \ldots, s_n$ s.t. $s_{i+1} \in T(s_i, a)$, for some $a \in A$. A *policy* for $\mathcal{D}$ is a function $\pi : S \mapsto A$. A $\mathcal{D}$-trace τ is induced by a policy π on $\mathcal{D}$ iff $s_{i+1} \in T(s_i, \pi(s_i))$.

A planning domain and a goal define a *planning problem*. We say that a policy π is a *strong solution* [3] to a problem $Prob = \langle \mathcal{D}, G \rangle$, if every $\mathcal{D}$-trace τ induced by π contains a state $s' \in G$. Intuitively, strong solutions are those which guarantee the achievement of a goal despite the non-determinism of all the executed actions.

The planning setting we consider here is that of non-deterministic planning under full observability. Model Based Planner (MBP) [3] is a very natural choice to compute strong solutions. Nonetheless, we chose a tool for verification and synthesis, Temporal Logic Verifier (TLV) [7], as it offers the possibility of easily building custom algorithms over transition systems. We plan to take advantage of such a flexibility in future extensions of this work. It is worth noting that, similarly to MBP, TLV exploits ordered binary decision diagrams (OBDDs) [2] to limit the space required by the search.

2.2 Markov Decision Processes

A Markov Decision Process is a tuple $MDP = \langle S, A, T, \rho \rangle$ where: S is a set of *states*, A is a set of *actions*, $T : S \times A \times S \to [0, 1]$ is the transition function. $T(s, a, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that the current state changes from s to s' by executing action a. $\rho : S \times \mathbb{R} \to [0, 1]$ is the reward function. $\rho(s, r) = Pr(r_{t+1} = r | s_t = s)$ is the probability to get a reward r from being in state s. MDPs are the model of choice for domains in Reinforcement Learning (RL). Our method takes a planning problem as input, and produces an MDP suitably constrained to learn more quickly than in the original domain. The behavior of the agent is represented as a function $\pi_t : S \times A \to [0, 1]$ called a *policy*, where $\pi_t(s, a)$ is the probability of selecting action a in state s at time t. The expected cumulative discounted reward, that is the reward expected from each state s executing action a, and following a policy π, is defined as: $Q : S \times A \to \mathbb{R}$. Its value is given by: $Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t \rho(s_{t+1}, \cdot) \,|s_0 = s,\ a_0 = a\right]$ where $0 < \gamma \leq 1$ is a discount factor. The discount factor is allowed to be 1 only in *episodic* tasks, where the MDP has *absorbing* states which once entered cannot be left. The reward is accumulated by executing a in s and following π thereafter.

2.3 Hierarchies of Abstract Machines

From the learning perspective, the problem of acting optimally in a fully observable environment has been studied under Reinforcement Learning. In particular, Hierarchical Reinforcement Learning (HRL) [1] is the field of RL most related to our work. In HRL, the designer provides structure to the policies searched, constraining the exploration in fully observable domains. Such structure can be provided in different forms, among which state machines.

A Hierarchy of Abstract Machines (HAM) [6] is a set of state machines that encodes a sketch of the solution for a given problem. The environment is described as a Markov Decision Process. The hierarchy constraints the policies allowed, effectively reducing possible exploration. At any given time, a machine either specifies an action to be executed or is in a *choice* state. When in a choice state, the next action is determined by the underlying MDP, performing learning only in those states. HAMs are hierarchical machines, but in this paper we only make use of one-level machines, therefore we report a simplified description. The complete definition of HAMs is available in the original paper [6].

The composition of a machine H and the MDP M, $H \circ M$, is obtained as described in the following. First, the Cartesian product of the state spaces of H and M is computed. Then, a new MDP over such a state space is defined by picking transitions either from the machine, or the MDP. An action is borrowed from H if the state is an action state, and from M if it is a choice state. Thus, only choice states have more than one action available. The reward is taken from the MDP if the transition is an action, while it is zero otherwise. The resulting process is still an MDP [6], and will be leveraged to learn in the representation we create through the planner. It is proved that there is no need to store the entire state space, since there exists always a Semi-Markov Decision Process (SMDP) such that its optimal policy is the same as $H \circ M$. Such an SMDP has only the states $\langle q, s \rangle \in H \times M$ in which q (the machine state) is a choice state.

The definition of the machine H is still an open problem, and is usually carried out by hand. To the best of our knowledge this is the first attempt to generate such a structure automatically. Our work differs from automated option discovery [8], as this usually consists in the identification of sub-goals, and learning of a procedure to reach them in the framework of Semi-Markov Decision Processes. Our work does not concern discovering a suitable goal, nor a representation of the sub-task, as both are part of the initial planning problem. Our method produces a structure that allows subsequent learning for a given task, and differently from the main focus of option learning, does not create new tasks.

3 Controller Generation and Learning

Our method for the specification of a controller is divided into two phases: generation and learning. At planning time the available model is used to produce a finite-state automaton as described in Section 3.1. Then, the learning phase begins by generating from such an automaton a HAM, as described in Section 3.2. Finally, at run time, the learning phase is carried out in the real environment.

3.1 Finite-State Controller Generation

We start from a formal representation of the environment, encoded as a non-deterministic planning domain. The domain is given as input to the TLV system, which we exploit to compute a set of policies guaranteed to achieve the goal.

Our aim is to build, for a given domain $\mathcal{D}$, a set of policies that are strong solutions. To do so, we created a variant of the algorithm for strong planning

Algorithm 1 Computes minimal-cost strong solutions

```
Input: a planning domain problem $Prob = \langle \mathcal{D}, G \rangle$
Output: a set of minimal strong solutions
Let $Z := G$, $Z' := \emptyset$, $T_G := \emptyset$;
While($s_0 \notin Z \wedge Z \neq Z'$) do
  Let $N := \{\langle s,a \rangle \in (S \setminus Z) \times A \mid T(s,a) \neq \emptyset \wedge T(s,a) \subseteq Z\}$; Let $T_G := T_G \cup \{\langle s,a,s' \rangle \in S \times A \times S \mid \langle s,a \rangle \in N \wedge s' \in T(s,a)\}$;
  Let $Z' := Z$;
  Let $Z := \{s \in S \mid \exists a. \langle s,a \rangle \in N\} \cup Z$;
End
if($s_0 \in Z$) return $T_G$; else return $\emptyset$;
```

described by Cimatti et al. [3]. Intuitively, the algorithm computes the set of all *minimal-cost* strong solutions for $Prob$, where by *cost* of a policy π we mean the length of the longest trace induced by π, containing a goal state only as its last state. To do so, the algorithm iteratively computes the set Z of all the states from which a policy that achieves a goal state exists. At the end of the i-th iteration, Z contains the set of states from which a goal state can be achieved in at most (depending on the outcome of non-deterministic actions) i steps. The relation N associates a state s with those actions whose successors are all members of Z. The relation T_G completes N with the information on the transitions that take place as actions are executed.

Although, in this paper, we focus on strong solutions, weaker notions of solutions can be in principle considered. For instance, one could use weak solutions, i.e., those that guarantee at least a possibility of reaching a goal, or strongly cyclic ones, i.e., those that guarantee goal achievement under a fairness assumption on the action effects. Also mixing these notions is possible.

The shortest solutions are not necessarily the optimal ones, as the true cost depends on the environment and cannot in general be modeled. Therefore, while policies that extend the minimal cost (in terms of number of actions) may be considered of lower quality by the planner, they may yield better performance in practice. This is the subject of the following learning phase.

3.2 Finite-State Controller Learning

The controller generated by the previous phase (cf. Section 3.1) can be translated into a one-level HAM as follows.

The output of the controller generator is a graph $G = \langle S, T_G \rangle$ where S is the set of states of the original domain $\mathcal{D}$. Essentially, when the domain is in state s, the agent is guaranteed that any action a, s.t. for some s' $T_G(s,a,s')$ holds, is a valid option to move closer to the goal.

Considering G as an acceptor, with the actions on the transitions as its symbols, a HAM H is obtained as the minimal automaton [5] that accepts such a language. The minimization process removes the environment state information, and the one-to-one correspondence between the states of the automaton and the states of the domain is not maintained in the new representation.

During learning, this state machine is used in conjunction with the observations (the states of the domain's MDP) as described in Section 2.3. The dynamics of the MDP needs not to be known, as learning at choice points can be model-free. If the state space is fully observable the system falls in the category of Semi-Markov Decision Processes, and the related theory guarantees convergence.

As an example, consider the simple planning problem described in the following. Bar-bot has to help a customer, by serving either coffee or tea. The actions available are *goto(location)*, *pickup(object)*, *pour*, *call-barman*, and *serve*. *Pour* is a non-deterministic action, that can either succeed of fail. If it fails bar-bot can recover by calling the barman. The customer likes one between coffee and tea more than the other, but which one is unknown to the agent. A pure learning approach would start trying actions, initially at random, to discover that bar-bot has to go to the fridge before picking up coffee; or that it has to have picked up coffee, before being able to pour it. The model is fed to the planner, together with a goal description in Linear Temporal Logic. TLV produces the graph in Figure 1. This graph represents two different strong plans (as they recover from non-determinism): one to serve coffee and one to serve tea. There is still a one-to-one correspondence between states of the graph and states of the environment at this stage. Once the graph is minimized, such a correspondence is lost, together with the information of whether an action is actually permitted in a given state. In Figure 2, the minimized automaton is shown, which constitutes our automatically generated HAM. In this automaton you can see

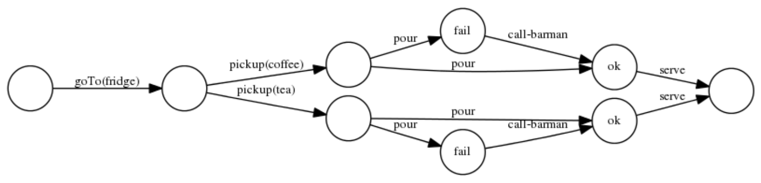

Fig. 1. The automaton created by the generation phase

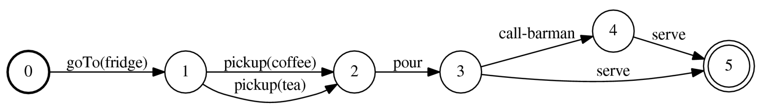

Fig. 2. The HAM obtained by minimizing the previous automaton

that *call-barman* and *serve* are both available in state 3. Call-barman, however, makes sense only if the previous *pour* action has failed. The HAM will introduce the state, if necessary, through the Cartesian product, as explained in Section 2.3. State 1 is a non-deterministic choice point, as both actions will be simultaneously available for execution. This decision will be learned, by getting reward, and discovering whether the client actually prefers coffee or tea. Such information was not available at planning time, and a pure planning agent cannot do better than choosing randomly among the options available.

4 Experimental evaluation

We begin our experimental section by introducing the printing domain, where we performed experiments on a realistic simulator. The task considered in this paper consists in receiving printing requests, sending the papers to an appropriate printer, picking up papers, delivering them to the corresponding person and returning to a home location. In this setting we modeled several kinds of uncertainties: robot sensor noise affecting localization and navigation, probability of paper jams in the printers, variable time to fix the printer by a human operator.

The state space is composed by the following variables: *location* $\in$ {home, at_printer_1, at_printer_2}, denotes the current position of the robot; *docsReady* $\in$ {0,1}, indicates whether there are documents ready to be printed. For each printer: *docsOnPrinter* $\in \mathbb{N}^+$ is the number of papers printed ready for pick-up; *isStuck* $\in$ {0,1}, indicates whether this printer is stuck and must be fixed; *helpRequested* $\in$ {0,1}, indicates whether this printer is stuck and help has been requested. Moreover, the robot has a fixed number of trays, 3 in our experiments, that can accommodate different papers. For each tray, a positive integer represents the number of papers in the tray.

The actions available to the robot are the following: `retriveDocs` that loads papers to be printed, affecting *numDocsReady*; `print_n` that sends the next document to be processed to the printer *n*; `goTo_l` that moves the robot to location *l*; `collect` that, executed from the location of a printer, picks up all the papers from a printer and puts them on the first empty tray; `askForHelp` that, executed from the location of a stuck printer, signals that the printer is stuck, and gets the printer fixed by a human operator; `deliver` that delivers all the papers in the trays to the appropriate people, terminating the episode.

The performance of the task is measured in terms of a score that considers both the number of papers printed and delivered, and the time to terminate each episode. For each paper successfully delivered the agent obtains a reward of 500. If two or more documents are mixed in a tray it receives 300 per document. Each action other than `deliver`, returns a negative reward equal to the opposite of the time, in seconds, taken by the action. Note how this information, as much as the probability of each printer to be jammed, is not known in advance. Therefore, it cannot be part of the model and exploited during the planning phase, but the agent must adapt during the execution.

4.1 Experimental setting

The main goal of the presented experiments is to show the capabilities of the proposed method to quickly learn an optimal strategy, providing a comparison with the execution of the controller generated in the planning phase without learning, and with pure learning.

Given the domain introduced above, a domain description is encoded in the language SMV, the input language for TLV, using a compilation scheme similar to the one described by Giunchiglia and Traverso [4]. The algorithm described in Section 3.2 returns a finite-state controller that is given in input to the learning phase. In this example, the returned controller has over 100 states and over 200 transitions.

The environment setting during the experiments (not known to the robot) is the following: the printer closest to the robot home position is faulty (0.8 probability of paper jam) and there is a quick response for fixing the printer (about a third of the time needed to go to the other printer); the other printer is faulty with a probability 0.2.

We perform three series of experiments: 1) using the controller generated by the planning phase with a random choice of actions in case of multiple possible actions in a state; 2) using a pure learning approach on the MDP built for this scenario; 3) using the combined planning+learning method described in this paper, where the controller generated by the planning step is refined through reinforcement learning.

We used Sarsa(λ) for both the learning cases, optimizing the parameters for the different representations. Common to both, $\lambda = 0.9, \gamma = 0.999, \alpha = 0.1$. The value function is initialized optimistically, and the exploration is performed following a ϵ-greedy strategy, where ϵ decays at different rates, depending on the representation. On the full MDP, ϵ going from 1 to 0.05 in 800 episodes performed best. With our method, the best decaying rate proved to be from 1 to 0.02 in 300 episodes. Each run has been repeated 100 times to average over the stochasticity of the domain.

4.2 Results

The results are shown in Figure 3. At the beginning of the learning phase, and for the first episodes, the 'plan' and the 'plan+mdp' approaches have the same values since they both act randomly over the generated controllers. However, after the initial exploratory phase of 300 episodes, learning allows to significantly improve on the pure planning approach, which keeps a constant average performance. On the other hand, the pure learning approach starts from a lower solution, since it is exploring a large space that contains many non-optimal behaviors. The figure clearly shows the increased learning rate of the 'plan+mdp' approach that is operating on a much smaller search space, with respect to 'mdp'.

In this case, the globally optimal solution was not part of the generated controller therefore the mdp eventually passes learn+mdp. This is common to all HRL methods, since limiting the search space cannot prevent from excluding the optimal path. Nonetheless, the learned behavior is guaranteed to be an improvement on the pure planning approach, as every solution computed by the planner is always part of the generated controller. Moreover, learning alone reaches the optimal behavior after a number of episodes which is usually not feasible when dealing with real systems. Last, focusing on exploration, note how exploring in the constrained space is always safe, as all the trajectories lead to the goal state. Unconstrained exploration, on the other hand, is in general dangerous in addition to costly. Therefore, the planning+learning method described in this paper

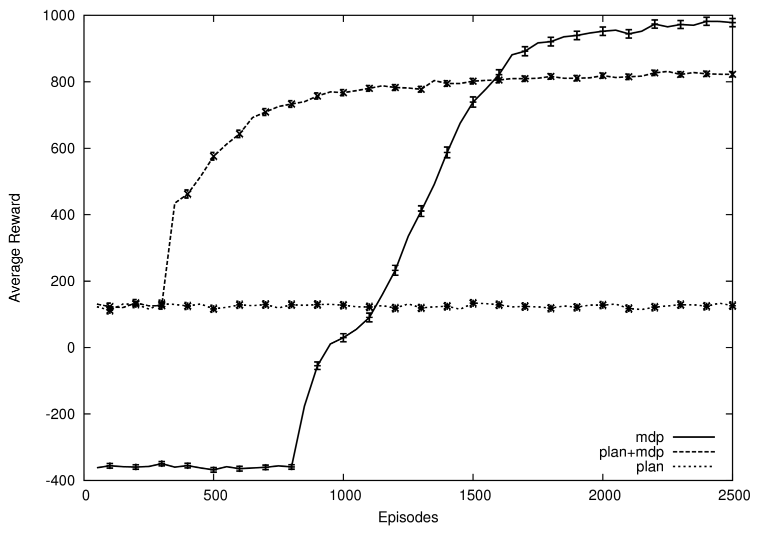

Fig. 3. The average cumulative reward obtained with learning alone, planning alone, and their proposed combination

guarantees to reach a “good” (close to optimal) performance with respect to a pure learning approach with significantly fewer experiments to be performed.

5 Discussion and Conclusion

We presented a method to generate and learn finite state controllers, combining techniques from both automatic planning and reinforcement learning. This method allows for suitably integrating the advantages of a planning approach in reducing the set of possible trajectories in a transition system (discarding those that do not lead to a solution or that are too expensive, with the advantages of a learning approach that is able to improve performance over time from on-line experiments in the environment.

The application of the proposed method to problems involving real systems in real environments (for example, robotic applications) has the advantage of finding “good” or optimal solutions for a complex task in the presence of an incomplete description of the domain with a small number of experiments. This is not possible when using either pure planning or pure learning methods.

We evaluated our method in a realistic domain, where the uncertainty behind the model constitutes the main difficulty, rather than the size of the domain itself. Such uncertainty can be alleviated by providing structure in the space of the task, and adapting to the environment. As the adaptation is constrained, both tracking non-stationarity in the environment (traditionally a hard problem) and exploring is faster and safer.

While in this paper we focus only on a two-step process, where planning is followed by learning, our approach can be extended to an iterative process where planning and on-line learning are interleaved. Specifically, at every iteration the learning phase returns a refined version of the controller that is used to refine the planning domain, thus allowing for the generation of a new controller, provided again as input to the learning phase.

References

1. Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, 2003.
2. Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.
3. Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artif. Intell.*, 147(1-2):35–84, 2003.
4. Fausto Giunchiglia and Paolo Traverso. Planning as Model Checking. In *Proc. of ECP*, 1999.
5. M. Mohri. Minimization algorithms for sequential transducers. *Theoretical Computer Science*, 234(1-2):177–201, 2000.
6. R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, pages 1043–1049, 1998.
7. Amir Pnueli and Elad Shahar. A Platform for Combining Deductive with Algorithmic Verification. In *Proc. of CAV*, 1996.
8. M. Stolle. *Automated discovery of options in reinforcement learning.* PhD thesis, McGill University, 2004.