Decision-theoretic Planning with Temporally Abstract Actions

Milos Hauskrecht
Computer and Information Sciences Department
Temple University
Wachman Hall 313
1805 North Broad Street
Philadelphia, PA 19122
milos@joda.cis.temple.edu

Nicolas Meuleau
IRIDIA, Universite Libre de Bruxelles
Avenue Franklin Roosevelt 50, CP194/6
B-1050 Brussels Belgium
nmeuleau@iridia.ulb.ac.be

Leslie Pack Kaelbling
MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139
lpk@ai.mit.edu

Thomas Dean
Computer Science Department, Box 1910
Brown University
Providence, RI 02912
tld@cs.brown.edu

Craig Boutilier
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 3H5
cebly@cs.toronto.edu

Abstract

Markov decision processes (MDPs) provide an elegant mathematical framework for modeling and solving planning and decision problems in the presence of uncertainty. While MDPs can be solved in time polynomial in the size of the state and action spaces using traditional dynamic programming techniques, for many practical problems these spaces are too large to be explicitly enumerated. This has prompted considerable research into various methods for accelerating and approximating the solution of MDPs. In this work, we consider extensions of MDPs to *temporally abstract actions* or *macro-actions*, that allow the treatment of sequences of actions or local policies as primitive actions. We investigate two models that exploit macro-actions: one in which macro-actions are used to enrich a set of primitive actions and leave the state space unchanged; and a hierarchical (abstract) model where macro-actions replace primitive actions, and an abstract state space is constructed, with the potential to significantly reduce the size of the MDP. We discuss several ways to generate macro-actions in order to ensure good solution quality. Finally, we investigate the use of macros in the solution of multiple, related MDPs, and demonstrate how this can justify the computational overhead of macro generation.

1. Introduction

Markov decision processes (MDPs) [17, 32, 4] have proven tremendously useful as models of stochastic planning and decision problems. While MDPs can be solved in time polynomial in the size of the state and action spaces using traditional dynamic programming techniques, for many practical problems these spaces are too large to be explicitly enumerated. Thus, considerable research has been directed toward the solution of MDPs with large state and action spaces through various speed-ups and approximations. These include function approximation [2, 15, 38, 20], reachability analyses [9, 3], structured approaches [11, 5, 24], decomposition [10] and state aggregation techniques [8].[1]

In this work we focus on an alternative approach to the computational difficulties associated with classical MDP solution techniques. We consider the use of *temporally abstract actions*, or *macro-actions*, that allow one to apply a preselected (or precomputed) *local policy* within the MDP, potentially alleviating certain computational bottlenecks associated with solving MDPs. A local policy or macro-action dictates behavior within a specific region of state space. This provides two computational advantages.

First, given a set of macro-actions, and descriptions of their effects, an agent can determine quickly (in one step) the consequences of applying a macro, thus, avoiding substantial reasoning with long sequences of primitive actions. Second, by restricting the agent to making decisions only at the boundaries of regions and to choosing only among the macro-actions covering these regions, the burden of making decisions at *each* state of the MDP is alleviated, leading to a much smaller *abstract MDP*. This form of abstraction can be used to generate a *hierarchy* of abstract MDPs, where the solution of more abstract MDPs is used to direct the solution of less abstract MDPs.

Our approach is motivated by classical planning in deterministic domains, where macro-actions (i.e., sequences of deterministic actions) are often used to accelerate the planning or problem-solving process by reducing the (operator) distance to the goal state. Macros are often created by caching and reuse of parts of the solution for previously solved, related problems [13, 25, 21]. We will see similar benefits in the application of macros to MDPs. Another motivation for our model is the use of abstraction and hierarchical approaches to planning. In this work, one solves a hierarchy of planning problems on different levels of abstraction, starting from the highest abstraction level [33, 19, 22], and using abstract solutions to guide the solution of less abstract versions of the planning problem. Similar ideas have been applied to MDPs by Dearden and Boutilier [11], but using a very different form of abstraction than that considered here.

1. See the survey paper by Boutilier, Dean and Hanks [4] for a review of various approximation techniques.

Recent research has seen the application of temporally abstract behaviors in stochastic settings emerge as an important method for tackling stochastic decision problems. This work has been quite diverse, adopting various guises and applied to many different types of decision problems. For example, many researchers have investigated the role of hierarchies and macros in the reinforcement learning setting [34, 18, 7, 37, 31, 36, 12], while others have focused predominantly on planning problems [30, 16, 28, 27]. The objective of this work is the development of a basic framework for integrating macro-actions into stochastic planning problems and for the generation of solutions using hierarchical techniques.

The model we develop is based on region-based decomposition of MDPs, and finds its roots in the work of Dean and Lin [10]. We use region-based decompositions to define sets of possible local policies (i.e., macros) an agent can consider. The main difficulty in incorporating macro-actions into the standard MDP framework lies in the fact that the execution of different macro-actions may extend over different periods of time. To deal with this problem, Precup, Sutton and Singh [35, 30, 31] have developed *multi-time models* and applied them to planning with MDPs. Multi-time models allow one to represent "actions" of different duration uniformly within an MDP and to apply standard solution algorithms such as value and policy iteration [1, 17]. We draw heavily on these ideas, using multi-time models to provide the basic semantics of macros.

We consider two ways in which macros can be used to accelerate the solution of decision-theoretic planning problems. In the first model, the *augmented MDP model*, macro-actions are simply added to the underlying MDP (i.e., added to the set of primitive actions). This model has been proposed by Sutton [35], and studied in depth by Precup, Sutton, and Singh [30, 31, 36], who demonstrated empirically that a good set of macros can increase the convergence rate of value iteration. We show that, in fact, this improvement is not always guaranteed; specifically, we demonstrate that convergence is sensitive to the value function estimate used to initialize the value iteration procedure.

Although the augmented model allows one to integrate macros into MDPs and speed up their solution, it still relies on explicit dynamic programming over the original state space (and a larger action space). Thus it does not address the most pressing computational bottleneck. The second model we consider, the *abstract MDP model*, requires an agent to use *only* macro-actions and, most importantly, allows one to reduce the number of states of the underlying MDP by restricting decisions to a subset of states of the original MDP. These abstract states are traversed using macro-actions. The abstract model is particularly well-suited to MDPs that exhibit a natural hierarchical decomposition (possibly with multiple state and macro-action levels). It can be viewed as a stochastic counterpart of the hierarchical approaches in deterministic domains [33, 19, 22]. The abstract model we propose is similar to those considered by Forestier and Variaya [14], Kaelbling [18], and Parr [28, 29, 27].

Two key limitations of the abstract model relate to the solution quality of the policies generated by solving an abstract MDP, and the computational overhead needed to generate a suitable set of macro-actions. Since the policy obtained by solving the abstract MDP consists only of macro-actions, certain behaviors cannot be realized. Thus, the resulting abstract policy may be suboptimal when viewed from the perspective of the original MDP. To ensure high quality solutions, the set of macros used must provide an adequate "coverage;" that is, it must allow for a range of policies sufficient to include close-to-optimal behavior. In Section 4 we describe different techniques for generating a set of suitable macro-actions.

In many cases, a suitable set of macros can be provided by a knowledgeable designer, who understands the domain well enough to constrain an agent's decision space without precluding good behavior. In other cases, we may be interested in the *automatic generation* of a macro set. The time required to generate a reasonable set of macros can also be prohibitive, generally requiring that we perform some form of dynamic programming within specific regions of state space. Since our regions *cover* state space, macro generation can be computationally very expensive. The challenge here is to adequately balance the trade-off between solution quality admitted by a set of macros and the time required to generate that set. The effect of this computational overhead can be diminished if we produce macros off-line and use them on-line, or reuse the same set of macros to solve multiple problems. We also analyze the requirements for feasible macro reuse and propose two models for doing this. Thus the presence of macros can ensure fast on-line response to certain types of changes in the original MDP specification. The use of macros for the on-line solution of multiple related MDPs is one of the main advantages of our abstract MDP model.

The paper is organized as follows. Section 2 introduces the concept of macro-actions in context of MDPs. Section 3 explores two models that use macro-actions in the solution of MDPs: the augmented and the abstract MDP models. Methods for automatic construction of good-quality macro-actions are the focus of Section 4. Section 5 presents two approaches for reusing macro-actions in the solution of multiple related MDPs: locally revised abstract MDPs and hybrid MDPs. We conclude in Section 6 with discussion of some open issues.

2. Macro-actions

In this section we develop a framework for the definition and investigation of macro-actions grounded in the region-based decomposition of MDPs [10] and the use of multi-time models [35, 31]. We first present basic background on MDPs. We then define region-based macro-actions, describe how one can construct suitable multi-time models for MDPs, and then discuss one method for generating suitable macro-actions by solving a *local MDP*.

2.1 Markov Decision Processes

A (finite) *Markov decision process* is a tuple $\langle S, A, T, R\rangle$ where: S is a finite set of states; A is a finite set of actions; T is a transition function $T : S \times A \times S \to [0, 1]$, defining a family of probability distributions over S for each $s, \in S$, $a \in A$; and $R : S \times A \to \mathbb{R}$ is a bounded reward function. Intuitively, $T(s, a, s')$ denotes the probability of moving to state s' when action a is performed at state s, while $R(s, a)$ is the expected immediate reward associated with executing action a at state s.

Given an MDP, the objective is to construct a *policy* that maximizes expected accumulated reward over some horizon of interest. We focus on *infinite horizon, discounted* decision problems, where we adopt a policy that maximizes $E(\sum_{t=0}^{\infty} \beta^t \cdot r_t)$, where r_t is the reward obtained at time t and $0 \leq \beta < 1$ is a discount factor. In such a setting, we restrict our attention to stationary policies of the form $\pi : S \to A$, with $\pi(s)$ denoting the action to be executed in state s. The value of a state given a policy π satisfies [17]

$$V_\pi(s) = R(s, \pi(s)) + \beta \sum_{s' \in S} T(s, \pi(s), s') \cdot V_\pi(s').$$

A policy π is *optimal* if $V_\pi(s) \geq V_{\pi'}(s)$ for all $s \in S$ and policies π'. The *optimal value function* $V^* : S \to \mathbb{R}$ is the value V_{π^*} of any optimal policy π^*.

Given a value function V, the *Bellman backup operator* produces an "improved" value function V' as follows:

$$V'(s) = \max_{a \in A} \{R(s, a) + \beta \sum_{s' \in S} T(s, a, s') \cdot V(s')\}. \tag{1}$$

We let $H_M : B \to B$ (where B is the set of bounded, real-valued value functions over S), denote this operator for a specific MDP M (we drop the subscript M when the MDP is clear from context). The optimal value function $V^* : S \to \mathbb{R}$ satisfies Bellman's fixpoint equation [1]:

$$V^* = HV^*.$$

A number of techniques for constructing optimal policies exist. An especially simple algorithm is *value iteration* [1]. In this algorithm we compute a sequence of value functions V^i starting from an arbitrary V^0, and defining

$$V^{i+1}(s) = \max_{a \in A} \{R(s, a) + \beta \sum_{s' \in S} T(s, a, s') \cdot V^i(s')\} = (HV^i)(s) \ . \tag{2}$$

The sequence of functions V^i converges to V^* in the limit.[2] After some finite number n of iterations, the choice of maximizing action for each s forms an optimal policy π^* and V^n approximates its value. We refer to Puterman [32] for a discussion of appropriate termination criteria.

2.2 Macro-actions as Local Policies

Macro-actions are used to model complex behaviors or collections of actions in an MDP. For example, the designer or programmer of a robot might provide typical or standardized policies or programs, such as navigating a hallway, grasping an object, or even more complex behaviors. These prespecified behaviors can be pieced together to form more complex behaviors, or used as primitives in the robot's planning process. This perspective is quite common. In reinforcement learning, Thrun and Schwartz [37] consider how an agent might learn to reuse policy fragments or *skills*. Parr and Russell [29] propose using finite-state controllers to specify partial policies and make decisions about how best to piece these

2. Convergence follows from the contraction property of the mapping H [1, 32].

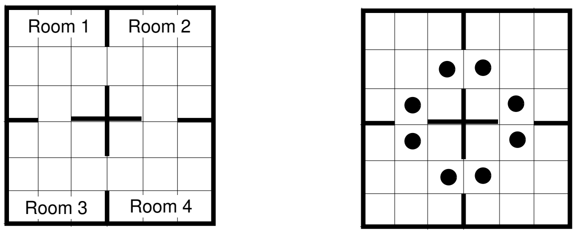


Figure 1: (a) A four-room example; (b) Peripheral states for a room partitioning.

together. Boutilier, Reiter, Soutchanski, and Thrun [6] propose a programming language in which behaviors and policy constraints can be specified, with an MDP solution procedure again used to determine how best to use these behaviors in the planning process. One benefit of using macros is the ability to exploit a programmer's knowledge of the structure of an optimal (or reasonable) policy. If it is known that an agent should act using *some* combination of the set of predefined behaviors, restricting attention to these behaviors can significantly reduce the complexity of the decision process an agent needs to solve.

To illustrate this notion, consider the simple robot navigation problem in Figure 1(a), in which a robot can move in each of the four compass directions. In addition to the basic move actions, a programmer may have provided a set of macros (or programs) that enables the robot to exit each of the rooms through each door. The execution of a macro thus consists of multiple steps involving multiple primitive move actions.[3] If the robot decides that exiting a room through a specific door is appropriate at a given point in the process, the relevant macro can be executed—the robot needn't *plan* how to exit the room.

Macro-actions can be modeled in many different ways. For example, macros can be represented as programs with arbitrary termination conditions as suggested by Precup, Sutton and Singh [35, 30, 31] or using a finite-state machine representation as proposed by Parr and Russell [29]. In our work, we define a somewhat simpler macro-action model, in which every macro-action is represented as a *local policy* restricted to some region of a state space. Formally, our model relies on a *region-based decomposition* of an MDP which finds its roots in the work of Dean and Lin [10].

Definition A *region-based decomposition* Π of an MDP $M = \langle S, A, T, R \rangle$ is a partitioning $\Pi = \{S_1, \cdots, S_n\}$ of the state space S. We call the elements S_i of Π the *regions* of M. For any region S_i, the *exit periphery* of S_i is

$$XPer(S_i) = \{s' \in S - S_i : T(s, a, s') > 0 \text{ for some } a \in A, s \in S_i\}.$$

The *entrance periphery* of S_i is

$$EPer(S_i) = \{s' \in S_i : T(s, a, s') > 0 \text{ for some } a \in A, s \in S - S_i\}.$$

3. Besides being given *a priori* by a programmer, macros can be constructed by reusing a solution obtained for a related MDP, or generated in a systematic fashion by covering a certain set of possible agent behaviors. We explore issues related to the automatic generation of macro-actions in Section 4.

We call elements of $XPer(S_i)$ *exit states* for S_i and elements of $EPer(S_i)$ *entrance states*. The collection of all peripheral states is denoted

$$\mathrm{Per}_\Pi(S) = \cup_i\{EPer(S_i) : i \leq n\} = \cup_i\{XPer(S_i) : i \leq n\}.$$

Notice that the set of states in the exit periphery of some region correspond to the set of states in the entrance periphery of some region. Figure 1(b) shows the set of peripheral states obtained if we partition the problem of Figure 1(a) into the four regions corresponding to different rooms.

Definition A *macro-action* for region S_i is a local policy $\pi_i : S_i \rightarrow A$.

A *macro-action* is simply a local policy defined for a particular region S_i. Intuitively, this policy can be executed whenever an agent enters or is in the region and terminates when the agent leaves the region (if ever). In our example, the macro-action corresponding to exiting a room through a specific door would dictate action choices at each state in that room (e.g., move in a direction that is expected to take the robot closer to the door). We note that other definitions of macro-actions (e.g., [30, 29]) are more general, allowing for arbitrary starting and termination conditions, and non-Markovian policies.

The main problem with integrating macro-actions into the MDP is that the execution of macro-actions can extend over different periods of time. A key insight of Precup, Sutton and Singh [35, 30, 31] is that one can treat a macro-action as a *primitive action* in the original MDP if one has an appropriate reward and transition model for the macro. They propose the following method of modeling macros.

Definition A *discounted transition model* $T_i(\cdot, \pi_i, \cdot)$ for macro π_i (defined on region S_i) is a mapping $T_i : S_i \times (S_i \rightarrow A) \times XPer(S_i) \rightarrow [0, 1]$ such that

$$\begin{aligned} T_i(s, \pi_i, s') &= E_\tau(\beta^{\tau-1} \cdot \Pr(s^\tau = s' \mid s^0 = s, \pi_i)), \\ &= \sum_{t=1}^{\infty} \beta^{t-1} \cdot \Pr\left(\tau = t, s^t = s' \mid s^0 = s, \pi_i\right) \end{aligned}$$

where the expectation is taken with respect to time τ of termination of π_i. A *discounted reward model* $R_i(\cdot, \pi_i)$ for π_i is a mapping $R_i : S_i \times (S_i \rightarrow A) \rightarrow I\!R$ such that

$$R_i(s, \pi_i) = E_\tau(\sum_{t=0}^{\tau} \beta^t R(s^t, \pi_i(s^t)) \mid s^0 = s, \pi_i),$$

where the expectation is taken with respect to completion time τ of π_i.

The discounted transition model specifies the probability of leaving S_i via a specific exit state, similar to a standard stochastic transition matrix, with one exception: the probability is *discounted* according to the expected time at which that exit occurs. This clever addition allows the transition model to be used as a normal transition matrix in any standard MDP solution technique, such as policy or value iteration [30, 31].[4] The reward model is similar, simply measuring the expected discounted reward accrued during execution of π_i starting from a particular state in S_i.

2.3 Computing Macro-action Parameters

The discounted transition and reward models are critical to our ability to treat macros as if they were primitive actions within the MDP, as we will see in Section 3. We now discuss how to compute transition and reward models, that is, the parameters needed to summarize the effects of macro-actions.

Let π_i be a macro defined on S_i. The discounted transition probability $T_i(s, \pi_i, s')$ for $s \in S_i$, macro π_i and $s' \in XPer(S_i)$ satisfies

$$T_i(s, \pi_i, s') \quad = \quad T(s, \pi_i(s), s') + \beta \sum_{s'' \in S_i} T(s, \pi_i(s), s'')T_i(s'', \pi_i, s').$$

This leads to $|XPer(S_i)|$ systems of linear equations, one set for every exit state s'. Each system consists of $|S_i|$ equations with $|S_i|$ unknowns. The systems can be solved either directly or using iterative methods. Thus, the time complexity of finding all transition probability parameters is $O(|XPer(S_i)||S_i|^3)$.

We can construct the reward model in a similar fashion. Let $R_i(s, \pi_i)$ be the expected discounted reward for following the policy π_i starting at state $s \in S_i$. Then we have:

$$R_i(s, \pi_i) = R(s, \pi_i(s)) + \beta \sum_{s' \in S_i} T(s, \pi_i(s), s')R_i(s', \pi_i).$$

This defines a set of $|S_i|$ linear equations, which can be solved in $O(|S_i|^3)$ time.

Overall, the computation of macro parameters takes $O(|XPer(S_i)||S_i|^3)$ time per macro. Note that if a large number of macros per region is used, the cost paid for computing the necessary parameters can easily outweigh the benefit resulting from applying macro-actions to solve the original MDP. We discuss this in some depth in Section 4.

2.4 Local MDPs

As discussed above, we can view macro actions as behavior fragments provided by a programmer whose knowledge of the domain allows her to confidently assert that this behavior fragment could be used as part of an optimal policy. Often, however, a programmer may not know the precise behavior to be implemented within a region, but does know (or can estimate) the value of leaving a region through each exit state. By specifying the value of the exit states of a region, a programmer permits the automatic construction of an optimal local policy within the region, through the solution of a *local MDP*.

4. Our definition of the discounted transition model is consistent with Equation 1, while Precup, Sutton and Singh fold the discount factor into the transition model. Thus their transition model is obtained by multiplying our variable T_i by the constant β.

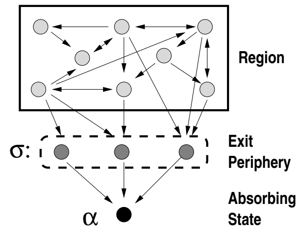


Figure 2: A Local MDP for Macro Generation

A local MDP for region S_i provides the means for combining the rewards associated with states within S_i with (estimates of) the expected values associated with the region's exit states.

Definition Let S_i be a region of MDP $M = \langle S, A, T, R \rangle$, and let $\sigma : XPer(S_i) \rightarrow I\!R$ be a *seed function* for S_i. The *local MDP* $M_i(\sigma)$ associated with S_i and σ consists of: (a) state space $S_i \cup XPer(S_i) \cup \{\alpha\}$; (b) actions, dynamics and rewards associated with S_i as in M; (c) a reward $\sigma(s)$ associated with each $s \in XPer(S_i)$; and (d) a single cost-free action applicable at each $s \in XPer(S_i)$ that leads with certainty to α (a cost-free absorbing state).

A local MDP is depicted graphically in Figure 2. Solving $M_i(\sigma)$ results in a local policy π_i whose behavior is optimal with regard to the seed function σ.

For example, if our programmer can estimate the value of leaving room 1 by each of its two exits, she can assign those values to the exit states, solve the local MDP, and use the resulting local policy as a macro-action for the original MDP. If the states in the interior of the room have a complex reward structure, or complex dynamics, it may be much simpler to construct such behavior automatically by specifying seed values than by spelling out the local policy directly.

Using local MDPs in the fashion described above requires that the programmer have some knowledge of the optimal value function for the underlying MDP: unless one specifies seed values that reflect the true value of reaching an exit state, the local policy generated may be arbitrarily far from optimal. We will see in Section 4 how local MDPs can be used to create a set of macro-actions that cover the space of "reasonable" behaviors when this knowledge is not directly available.

3. Solving MDPs Using Macro-actions

Suppose we have an MDP M and a set of macros defined for each region S_i induced by some state space partitioning Π. There are two ways in which macro-actions can be used to accelerate the solution of M:

1. The *augmented MDP model* [35, 30, 31] treats macros as if they were new primitive actions that can be used by the agent in the original MDP. A new MDP is constructed by extending the action space A with the set of macro-actions.

2. The *abstract MDP model* treats macros as the only actions available to the agent. An abstract MDP is formed from M by restricting attention to macro-actions and peripheral states. One of the main contributions of this work is the development and study of a specific model of abstract MDPs. The form of abstraction we investigate has been studied by others as well [14, 27, 28].

We consider each of these models in turn.

3.1 Augmented MDPs

The *augmented MDP* model is constructed by adding macro-actions (with their discounted transition and reward models) to the original set of primitive actions. Although the augmented model uses discounted transition matrices for macro-actions and thus it is not a classical MDP, it can be still solved using standard methods, such as value iteration. Because all *base level* actions (those in A) are present, the policy found is guaranteed to be optimal. In addition, the presence of macros may enhance the convergence of value iteration, as demonstrated empirically by Precup, Sutton, Singh [30, 31] and others.[5] This happens because one "application" of a macro can propagate values through a large number of states and over a large period of time in a single step. Thus, by reducing the number of iterations required by value iteration to reach convergence, macros in the augmented MDP model offer (potentially) significant computational savings.

However, the potential for enhanced convergence is not always guaranteed. In this section we prove that the convergence rate and the speed-ups are sensitive to the initial value function estimate, thus the inclusion of macro-actions may not help in all cases. Interestingly, this is also the case if only optimal macros (i.e., local policies for regions formed by restricting a globally optimal policy to these regions) are used to enrich the primitive action set.

At first, it may seem that extending an MDP with macro-actions is always useful. However, there are situations in which a macro-action (even if it is optimal) never affects the value of the value function during value iteration; in such situations, we pay a computational penalty for adding the macro-action and derive no benefit from it.

Lemma 1 *Let M and M' be MDPs such that M' differs from M only by having additional actions; that is, $A_M \subset A_{M'}$. Then for any value function V such that $V \geq V^*_M$ (i.e., V upper-bounds the optimal value function for M), $H_M V \leq H_{M'} V$.*

Proof The value function for a state s obtained after a Bellman backup is

$$V^+(s) = \max_{a \in A} Q^+(s, a)$$

5. We note that these savings do not necessarily account for the possible overhead associated with generating macros and constructing an appropriate model for each macro, which will be the focus of the next section.

where A is a set of actions and $Q^+(s,a)$ is an action-value function defined as

$$Q^+(s,a) = R(s,a) + \sum_{s' \in S} T(s,a,s')V(s').$$

Then if $A_M \subset A_{M'}$, regardless of the transition and reward models associated with additional actions, it must hold that

$$\max_{a \in A_M} Q^+(s,a) = H_M V(s) \leq \max_{a \in A_{M'}} Q^+(s,a) = H_{M'} V(s).$$

□

The above theorem is general and holds for arbitrary MDPs having this relationship, whether or not the additional actions are macro-actions. The result implies that the value iteration procedure, when initialized with an upper-bound value-function estimate, is no worse on the original MDP (with primitive actions only) in terms of the number of iteration steps than on any augmented MDP. As an update for an augmented MDP involves the computation of more Q-functions (for both action and macro-actions) this choice also makes the augmented model inferior in terms of the running time, as stated in the following. (Recall that the optimal value function for both an augmented MDP M' and the original MDP M from which it is formed are the same.)

Theorem 2 *Let M' be an augmented MDP formed from MDP M by the addition of a set of macro-actions. If value iteration for both M and M' starts from the initial upper bound $V_0 \geq V^*$, then value iteration on M' converges no faster than value iteration on the base model M.*

Proof An augmented MDP converges to the same fixed point solution as the original MDP, that is, $V^* = H_M V^* = H_{M'} V^*$. By Lemma 1, the value function $H_{M'}$ is an upper bound on H_M. Also, both H_M and $H_{M'}$ are isotone [32]. Then, assuming that an initial value function V_0 satisfies $V^* \leq V_0$, we obtain $V^* \leq H_M V_0 \leq H_{M'} V_0$ after the first update step, $V^* \leq H_M^2 V_0 \leq H_{M'}^2 V_0$ after two steps and $V^* \leq H_M^i V_0 \leq H_{M'}^i V_0$ in i steps. Thus, the value function obtained through value iteration on M is always better (closer to the optimal value function) than that based on M'. As the model augmented with macro-actions requires us to compute Q-values for every macro-action, the method is clearly inferior and must lead to slower running times. □

This theorem suggests that one should not blindly apply value iteration to MDPs with macro-actions as the potential benefit and intended speed-up is not guaranteed. To avoid the suboptimal behavior one should always initialize the value function estimate with a lower bound. This choice makes macro-actions useful and has the potential to improve the convergence rate.

Theorem 3 *Let M' be an augmented MDP formed from MDP M by the addition of a set of macro-actions. If value iteration for both M and M' starts from the initial lower bound $V_0 \leq V^*$, then value iteration on M' converges no more slowly than value iteration on the base model M.*

Proof The proof is similar to the previous case. Starting from the lower bound $V_0 \le V^*$, knowing that $H_{M'}$ and H_M are isotone contractions and that $H_{M'}$ upper bounds H_M, it must hold $H^i_M V_0 \le H^i_{M'} V_0 \le V^*$. Therefore the value function obtained using the value iteration procedure for M' converges to the optimal value function no slower than value iteration for M. $\square$

Intuitively, longer and better macro-actions can provide better Q-values as compared to primitive actions, thus influencing the convergence of value iteration more. Note, however, that starting from the appropriate bound is not sufficient to guarantee the actual speed-up in terms of running time. With an extended set of actions, the slowdown resulting from the computation of additional Q-value updates for every macro may outweigh the benefit obtained by accelerated convergence.

3.2 Abstract MDPs

To solve an augmented model requires explicit value iteration over the complete state space. Thus, despite its potential to speed up the convergence rate, it does not alleviate the problem of large state space size. To address this problem we develop an *abstract MDP model.*

3.2.1 Defining Abstract MDPs

In the abstract model we consider only peripheral states and macro-actions connecting them. The advantage of this is that if the number of peripheral states and macro-actions is small, the model leads to a much smaller decision problem, taking advantage of the fact that, by committing to the execution of some macro, decisions need only be made at peripheral states, not at states that lie strictly within a region. Thus the planning process can be performed more efficiently as compared to the original problem. We note that the abstract model is closely related to approaches studied by Forestier and Varaiya [14] and Parr [27, 28], as well as the "landmark" technique developed by Kaelbling [18].

Definition Let $\Pi = \{S_1, \cdots, S_n\}$ be a decomposition of MDP $M = \langle S, A, T, R \rangle$, and let $\mathbf{A} = \{A_i : i \le n\}$ be a collection of macro-action sets, where $A_i = \{\pi_i^1, \cdots, \pi_i^{n_i}\}$ is a set of macros for region S_i. The *abstract MDP* $M' = \langle S', A', T', R' \rangle$ induced by Π and $\mathbf{A}$, is given by:

- $S' = Per_\Pi(S) = \cup\{EPer(S_i) : i \le n\}$;
- $A' = \cup_i A_i$ with $\pi_i^k \in A_i$ feasible only at states $s \in EPer(S_i)$;
- $T'(s, \pi_i^k, t)$ is given by the discounted transition model for π_i^k, for any $s \in EPer(S_i)$ and $t \in XPer(S_i)$; $T'(s, \pi_i^k, t) = 0$ for any $t \notin XPer(S_i)$;
- $R'(s, \pi_i^k)$ is given by the discounted reward model for π_i^k, for any $s \in EPer(S_i)$.

The transition and reward models required by the abstract MDP are restricted to peripheral states and make no mention of states "internal" to a region. Due to discounting in T' these definitions do not describe an MDP, but they still preserve the Markov property;

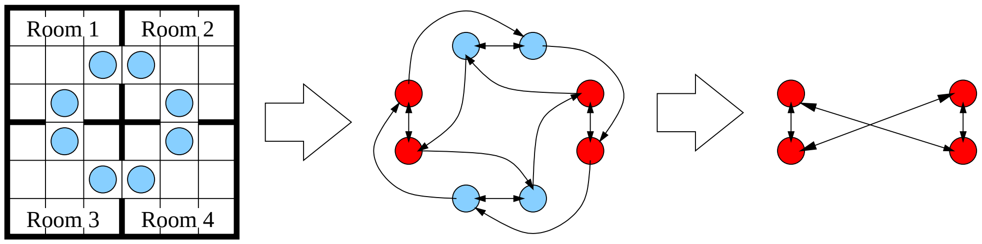


Figure 3: Abstract MDP for a four-room example. Grey circles mark peripheral states of the original MDP, i.e., states of the abstract MDP. The abstraction process is recursive and one can reduce the complexity of an abstract MDP further using the same decomposition mechanism, and creating a hierarchy of abstract MDPs.

specifically, the discounted probability of moving from any entrance state to an exit state for a given macro is independent of previous history. Thus, we may use dynamic programming techniques (value iteration, policy iteration, or even linear programming approaches) to solve the abstract MDP.

An example of an abstract MDP for our earlier four-room navigation problem is shown in Figure 3. Regions are formed by the rooms and the peripheral states make up the abstract MDP. We assume macros exist that can take the robot out of any room through any door, accounting for the connectivity of the abstract MDP. The abstract MDP effectively reduces the size of the planning problem to one involving only periphery states and macro-actions allowing the robot to move from room to room.

Abstract MDPs are particularly well-suited to modeling problems whose solutions naturally decompose hierarchically. In our running example, it is quite intuitive to view policies abstractly as involving moves from room to room, while the moves within each room are considered distinctly. This form of abstraction is also amenable to multiple levels of abstraction: we can create a hierarchy of abstract MDPs, such that a higher level (more abstract) MDP is defined in terms of states and macro-actions of a lower level abstract model. In our running example, we might further abstract the navigation problem by treating each floor of an office building as a region at a higher level of abstraction, where its states are the less abstract (but still, not primitive) rooms on that floor. In Figure 3, for example, the most abstract MDP on the right is produced by grouping rooms 1 and 2 and grouping rooms 3 and 4 (perhaps because 1 and 2 are on the same floor, as are 3 and 4). Such an approach can be viewed as a stochastic counterpart of the deterministic hierarchical planning methods. It represents a good alternative to a variety of approximation methods for solving large MDPs.

The main advantage of the abstract MDP M' induced by a given decomposition is that it can be substantially smaller than the original MDP, especially if the problem can be decomposed into a number of regions with relatively small peripheries. Various agent navigation problems of the type considered by Precup, Sutton, and Singh [30, 31], for example, are particularly well-suited to this form of decomposition.

The primary disadvantage of the abstract model is that in order for it to be computationally advantageous, the decision maker can consider only a limited range of behaviors (macros) and these need to be prepared in advance. Unfortunately, one does not always have sufficient understanding of a domain to be able to restrict the range of "reasonable" behaviors to a small set for each region. Thus, if a small set of macros is used, one runs the risk of generating a macro-policy that is suboptimal when viewed from the perspective of the original (non-abstract) MDP. Therefore, the abstract MDP model is most useful when good rather than optimal behavior is acceptable, or when strong *a priori* knowledge can be used to restrict the set of macro-actions used in the abstract MDP. In this sense it is best viewed as an alternative to various approximation methods for solving large MDPs [11, 2, 10, 8].

3.2.2 Solving Abstract MDPs

We call a policy $\pi' : S' \to A'$ for M' that maps peripheral states to macro-actions a *macro-policy*. Such a policy π', when considered in the context of the original MDP M, defines a *non-Markovian* policy π; that is, the choice of action at a state s can depend on previous history. In particular, the action $\pi(s)$ to be executed at some state $s \in S_i$ will generally depend on the state s_e by which S_i was most recently entered: $\pi(s) = \pi'(s_e)(s)$.

Given an abstract MDP M', its solution is the macro-policy produced by solving the abstract MDP M' using any standard optimal policy construction algorithm, such as value iteration or policy iteration. The quality of the optimal macro-policy as compared to the optimal policy in the underlying MDP depends critically on the macros being used. We discuss this issue in depth in Section 4.

One problem with macro-policies is that they do not dictate which macro-action to take if the process begins at an arbitrary internal state s of some region S_i. Thus, we need to first identify a macro most suitable for that internal state. To do this, we rely on the parameters R_i and T_i for each macro, which are computed prior to solving the abstract MDP. Let V' be the optimal value function for the abstract MDP (defined only on peripheral states), $s \in S_i$ be an internal start state of the region S_i, and A_i be a set of macros for S_i. Then the best macro-action for an arbitrary state s is

$$\pi_i^s = \arg\max_{\pi_i \in A'} \left[R_i(s, \pi_i) + \beta \sum_{s' \in XPer(s_i)} T_i(s, \pi_i, s') V'(s') \right] \tag{3}$$

We refer to this approach as one-shot macro selection. The approach determines the macro with highest expected value at the initial state s and commits to that macro. Notice that selecting this macro is not computationally demanding. This selection retains the spirit of the abstract MDP approach, simply extending it to an macro selection at an arbitrary initial state.

3.2.3 Policy Refinement in Region Interiors

The solution to abstract MDP tells us how to act in different states. The optimal policy based on an abstract MDP is non-Markovian with regard to the underlying model M, that is, the choice of the action in an arbitrary internal state s depends on the state used to enter the region or start state. However, using the solution to the abstract MDP, it is possible to construct a Markovian policy with respect to the M, that is, a policy that associates actions with states in the interior of a region that is independent of how the region was entered. During this process, one can improve the value of the new policy as compared to the original macro-policy. This process is best viewed as a refinement of the abstract MDP model. In the case of a hierarchy of abstract MDPs, the refinement process can run on multiple levels, starting from the most abstract MDP, and gradually removing the dependencies on entrance periphery states in less abstract MDPs. In the following we consider two refinement techniques.

The first technique, *greedy refinement*, is similar to the one-shot macro selection, but allows one to apply different macros at different states within the region. Specifically, the agent can switch from one macro to another as it moves through the region—it need not commit to the macro selected initially. More precisely, let S_i be a region, and for each $s \in S_i$, define π_i^s as in Equation 3. The greedy refinement of a local policy $\mu : S_i \to A$ is given by $\mu(s) = \pi_i^s$. Thus at each state the action dictated by the best macro at that state is executed. The advantage of using such a policy is that it tends to improve control as compared to committing to a single macro-action. It is clear that the value of the policy obtained through greedy refinement of regions can be no worse than that obtained by following the macro-policy. We note that a similar idea has been investigated independently also by Sutton *et al.* [36].

Theorem 4 *Let $\pi' : S' \to A'$ be the optimal macro-policy obtained by solving the abstract MDP, and $V' : S' \to R$ be the expected reward corresponding to such a policy. Then $\pi : S \to A$ constructed using the greedy refinement method seeded with values V', satisfies $V'(s) \leq V_\pi(s)$ for all $s \in S'$.*

The proof of the theorem is similar to the proof for the local MDP refinement method stated below. A similar theorem (with a proof) appears also in Sutton et al. [36].

The second method, *local MDP refinement*, constructs a policy by solving a set of local MDPs, one for each region, using the value function values V' obtained from the solution of the abstract MDP as the seed values for exit states. That is, at a region S_i, we set $\sigma(s) = V'(s)$ for all $s \in XPer(S_i)$). As with greedy refinement, it is possible to show that the new policy is no worse than the macro-policy, that is, it always yields higher or equal expected rewards.

Theorem 5 *Let $\pi' : S' \to A'$ be the optimal macro-policy obtained by solving the abstract MDP, and $V' : S' \to R$ be the expected reward corresponding to such a policy. Then $\pi : S \to A$ constructed using the local MDP refinement approach seeded with values V', satisfies $V'(s) \leq V_\pi(s)$ for all $s \in S'$.*

Proof Let S_i be a region of S. The local MDP refinement method computes the policy π_i^σ that is optimal in S_i given the set of seeds σ. Thus, for all $s \in EPer(S_i)$, it must hold that

$$V'(s) \leq V^\sigma_{\pi_i}(s).$$

In other words, the local policy always leads to expected rewards (for a given set of seeds) that are no worse than the non-Markovian policy based on macros only. As seeds used for a local MDP correspond to values obtained by the optimal macro-policy and every local policy found is no worse than the optimal macro choice for periphery state, the combination of local policies (local polices are strung together) cannot lead to smaller expected rewards. Thus, the resulting global policy is no worse than the optimal macro-policy. □

While both policy refinement techniques guarantee the improvement over the macro-policy, the questions of how these compare to each other and whether one refinement strategy produces a policy whose value dominates the other remain open.[6]

To illustrate the effect of refinements in our running example, assume that we have two macros that push the agent to exit room 1 via one of the two doors, and that the expected value for exiting through each door is roughly the same. Then, if the agent is positioned closer to one door it should move toward that door by invoking the appropriate macro-action. However, if moves are highly stochastic, the agent, while attempting to exit via one door, could be diverted closer to the other. In that case it would be appropriate to switch to a macro-action that takes the agent towards the other (closer) exit. While one-shot macro selection does not allow us to switch macros (the original macro-action is always executed until the originally selected exit is reached), the greedy strategy does allow for policy switching. The local MDP method allows one to construct a new local policy, distinct from each of the macros, that could be loosely interpreted as allowing the agent to "move to the closest exit." Note that, while the computational effort of the region refinement is greater for the local MDP method, this MDP will still generally be quite small when compared to the original MDP, since it involves only states in the initial region S_i.

3.3 Experimental results

To demonstrate the computational savings made possible by using macro-actions in planning tasks, we have performed experiments on the simple navigation problem in Figure 4. These experiments are merely suggestive (we consider more complicated domains below when we study macro reuse). The agent can move in any compass direction to an adjacent cell or stay in place. The move actions are stochastic, so the agent actually moves in an unintended direction with some small probability. The objective is to maximize the expected discounted reward (or minimize cost) incurred by navigating the maze, with each state, except the zero-cost absorbing goal state, incurring some negative reward. The rewards and transition probabilities are not uniform across the maze.

We compared the results of value iteration for the original MDP, the augmented MDP and the abstract MDP, the latter two formed using the rooms in the problem as regions.

6. We conjecture that there is no clear dominance between the two refinement methods.

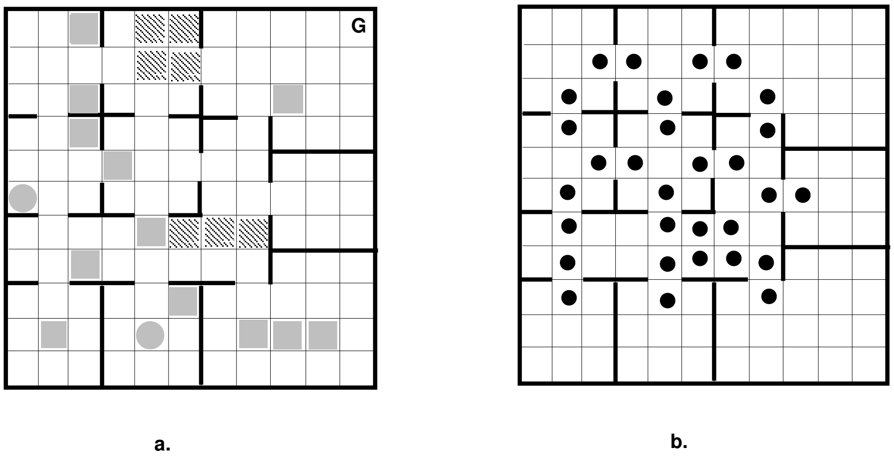


Figure 4: (a) Test problem Maze 121. Shaded squares denote locations with higher cost, patterned squares represent areas in which moves are more uncertain (a move in the intended direction is less likely). Shaded circles denote absorbing states with a finite positive cost, G stands for a zero-cost goal state; (b) peripheral states for the partitioning into 11 rooms (regions).

The macros were formed heuristically using the simple strategy described in Section 4, giving $|XPer(S_i)| + 1$ macros for every region S_i. Figure 5 shows how the estimated value of a particular state improves with the time (in seconds) taken by value iteration on each of the three models. When the initial value function estimate is a lower bound both the augmented MDP and the abstract MDP lead to faster convergence of the value function. In the augmented MDP, the ability of macros to propagate value through a large number of states produces large changes in the value function in a single iteration step, overcoming the increased number of actions. Note, however, that when the initial estimate of the value function is an upper bound, the augmented MDP actually performs worse than the original MDP, as proved earlier.

The abstract MDP has significantly reduced the size of both state and action spaces. Although in general, macros can lead to suboptimal value functions (and subsequently policies), in our example, the abstract MDP produced nearly optimal policies (and did so very quickly). The average time (in seconds) taken per value iteration step in this example is 0.045 for the original MDP, 0.12 for the augmented MDP, and 0.019 for the abstract MDP. This reflects the increased action space of the augmented MDP and the reduced action and state spaces for the abstract MDP, as expected.

While solutions obtained for the base level and augmented MDPs allow us to get the optimal Markovian policies directly, solving the abstract MDP results in a non-Markovian policy which uses macros and periphery states only. A non-Markovian policy could be

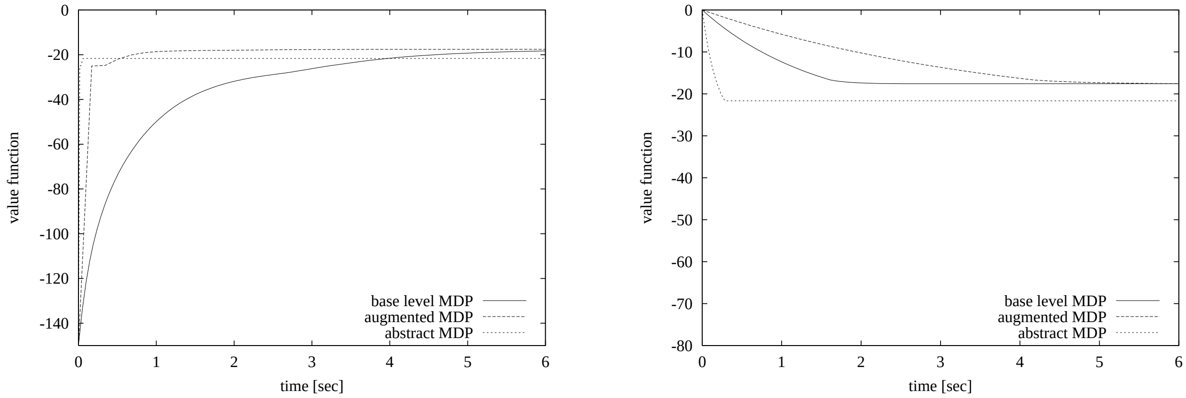


Figure 5: Value function estimates versus time for various models. Results with a properly initialized value function (w.r.t. the augmented MDP) are shown on the left. Results for a poor initial function are on the right. In the latter case, the augmented MDP converges more slowly than the original MDP.

converted into a Markovian policy using the local MDP refinement method or the greedy refinement described above. This allowed us to compare polices obtained for the original MDP and abstract MDP methods. In our agent-navigation example the Markovian policy we obtained for the abstract-MDP solution via the local MDP refinement turned out to be equivalent to the optimal base-level policy. The policy obtained via the greedy refinement method (best choice dictated by any macro-action) was very close to the optimal one, but different at a few states.

4. Generating Macro-actions

Up to this point, we have assumed that the set of macro-actions used to improve the ability to solve an MDP have been defined, specified, and implemented in advance by the designer or programmer of an agent; for example, standardized procedures for navigating a hallway, exiting a room, and so forth may have been provided. Generally, these assume some knowledge of reasonable agent behavior on the part of the designer. However, if these are not known we might still wish to generate a suitable set of macros in order to enhance our ability to solve an MDP. In order to prove useful, an automatically generated set of macro actions should: (1) ensure that good quality solutions to the underlying MDP can be found; and at the same time, (2) require that the computational cost to generate the multi-time models is, at least, compensated for by the savings in solving the new MDP model.

In general, a better set of macro-actions allows one to improve the convergence rate of value iteration if the augmented MDP model is used, or to ensure that a good quality approximation to the optimal (base-level) policy is obtained if solving the abstract MDP model. In this section, we focus largely on the problem of macro generation for the abstract MDP model.

It can be hard to guess the optimal or even a close-to-optimal set of macros directly. In order to ensure good quality behavior, one approach to macro generation is to ensure that the set of macros generated for a specific region is diverse enough to span the range of all reasonable behaviors. The larger (and more diverse) the set of macros, the more likely one is to find a macro policy whose value (w.r.t. the original MDP) is reasonably close to optimal. However, increasing the number of macros per region also leads to a more complex model which makes the abstract MDP somewhat more difficult to solve, but, more critically, makes macro generation more difficult. The key concern lies with our ability to generate macro-action models (that is, their discounted transition and reward models T_i and R_i) effectively. The primary challenge is to come up with a small set of good quality macros, and to ensure that the computational overhead associated with macro-model construction does not exceed the cost of solving the original MDP directly.

In the following, we investigate techniques for generating a good set macros that ensures that good quality policies are found. The question of ensuring that the computational cost of macro generation pays off is deferred to Section 5.

4.1 Policy Coverage Methods

A simple approach to constructing a set of macros for a region would be to consider all possible local policies. Such an approach ensures that all policies for a base-level model M would be considered and hence it leads to the optimal base-level solution. Unfortunately, it would be also computationally very expensive; the number of possible macro-actions for a region S_i with $|S_i|$ states and $|A|$ primitive actions is $|S_i|^{|A|}$, which is exponential in the number of actions. Such an approach is unlikely ever to pay dividends.

4.2 Value Coverage Methods

An alternative approach to macro generation is to construct local MDPs for each region, and seed the local MDPs defined for each region with an estimate of the optimal value function at the exit states for the region (this is the technique described in Section 2.4). If it is known, for example, that exiting room 1 via one door has value 1 and exiting via the other door has value 0, the local MDP using these seed values could be solved to produce the relevant macro for room 1. Macros so created would be part of an optimal policy.

The main problem with generating macros using local MDPs is to chose periphery seeds that most closely reflect the true expected reward. Intuitively, if we could seed the exit periphery of each local MDP using a function σ within ε of the true value function at these states, we could generate a single macro for each region, and "string them together" to obtain an approximately optimal policy. More precisely, we have:

Theorem 6 *Let $\Pi = \{S_1, \cdots, S_n\}$ be a decomposition of MDP M, and let V be the optimal value function for M. Let $\mathbf{A} = \cup\{A_i : i \leq n\}$ be a set of macro-actions such that each A_i*

contains some macro π_i generated by the local MDP $M_i(\sigma_i)$ where $|\sigma_i(s) - V(s)| \leq \varepsilon$ for all $s \in XPer(S_i)$. If M' is the abstract MDP induced by Π using action set $\mathbf{A}$, and V' is the optimal value function for M', then

$$|V'(s) - V(s)| \leq \frac{2\varepsilon\beta}{1-\beta}$$

for all $s \in S'$ (the abstract state space). Furthermore, if τ is a lower bound on the completion time of all macros, then

$$|V'(s) - V(s)| \leq \frac{2\varepsilon\beta^{\tau}}{1-\beta^{\tau}}.$$

Proof The macro-action generated by an ε-optimal seed guarantees its expected reward accrued within a region S_i to be at most $2\beta\varepsilon$ from the optimal local expected reward. This follows directly from the contraction property of the value iteration update with a discount factor $\beta < 1$. The cumulative error resulting from an infinite sequence of ε-optimal macros is then less than or equal to $\frac{2\varepsilon\beta}{1-\beta}$. This assumes that every macro stays in its region for at least one step. However, if we know that every macro stays in the region for at least τ steps, the discount factor β^{τ} can be used instead of β. This once again follows from the contraction property of the value iteration update for τ steps. □

The above error bounds for ε-optimal macros are very rough and more precise error bounds are possible. These are based on the idea of "effective" discounting of macro-transitions rather than discounting based on the minimal macro-action length.

Definition Let $T_i(s, \pi_i, .) : XPer(S_i) \to [0, 1]$ denote discounted transition model for an initial state $s \in S_i$ and macro-action π_i. Then $\beta(s, \pi_i) = \sum_{s' \in XPer(S_i)} T_i(s, \pi_i, s')$ is called an effective discount factor for state s and macro-action π_i.

In other words, the effective discount factor represents the discounting one would apply to state s if a macro-action is treated as a single action, and transition probabilities sum to one.

Once the abstract MDP with ε-optimal macros is solved, the error for state s can be bounded by an error function $E : S_i \to R$, obtained by solving the following system of equations:

$$E(s) = \sum_{s' \in XPer(S_i)} T_i(s, \pi_i^s, s') \left[2\varepsilon + E(s')\right]$$

where π_i^s denotes a macro dictated by the macro-policy obtained by solving the abstract MDP. The right-hand side of the equation consists of two parts: the maximum error ε incurred within the region S_i and the maximum error we can incur after we exit the region. Note that the effective discount factor is "hidden" in the discounted transition model T_i.

The previous result indicates that knowledge of the (optimal) value function for an MDP can give rise to good macros. Unfortunately, such prescience is rare: if we knew the value function, we would have no decision problem to solve. However, this basic idea can be extended to create a *set* of macros for a given region. The *value coverage* technique requires that one solve multiple versions of the local MDP for a given region, differing only in the value seeds used. The seeds σ should cover the range of possible values the exit states take under the optimal value function. We often have heuristic knowledge regarding the range of the value function at certain states, or constraints on its possible values. It is precisely this type of knowledge that comes into play when one imposes partial policies (say, in the form of a control routine). Even *some* information of this type can be used to construct a good set of macros that guarantees approximately optimal performance. We consider several methods for exploiting such knowledge.

If one knows the *range* of the value function, this can be used to construct a set of macros systematically. For instance, when constructing macros for room 1 in Figure 3, suppose we know that the values of the two exit states lie between $V_{\min}$ and $V_{\max}$.[7] Then in order to generate a set of macros for room 1 that is guaranteed to contain a good macro, we can use the *ε-coverage technique*: intuitively, for each of the two exit states, we consider values that lie in the range $[V_{\min}, V_{\max}]$ spaced some ε apart; that is, we consider a grid or mesh covering $[V_{\min}, V_{\max}]^2$. By constructing macros corresponding to optimal local policies for each σ lying on a grid point, we are assured that one such σ is within $\frac{1}{2}\varepsilon$ of the optimal value function and that (assuming other regions have "good" macros from which to choose) close-to-optimal behavior results when the abstract MDP is solved.

The ε-coverage technique can be extremely expensive: given such a generic knowledge of the value function, we will generate $[(V_{\max} - V_{\min})/\varepsilon)]^{|XPer(S_i)|}$ different seeds per region, and for every seed we will construct a new macro corresponding to the optimal local policy. However, we can often do much better.

First, the number of macros is usually smaller than the number of grid points covering $[V_{\min}, V_{\max}]$. Thus it is often more appropriate to search a local policy space. One technique for finding a set of macros with ε-precision guarantees was developed recently by Parr [27]. This technique builds up a macro cache for a region incrementally, basing its decisions to create a new macro on the "range" of values covered by the existing macro set. Essentially, only "useful" macros will be added to the set, in contrast to the ε-coverage technique (which for various combinations of seeds might construct essentially the same local policy). Unfortunately, this method requires the solution of multiple linear programming problems for every region, which may be computationally prohibitive and impractical as well. However, we conjecture that Parr's method will generally provide a much tighter set of macros for a given region than the ε-coverage technique. Parr provides error bounds much like ours on the quality of the optimal macro policy with respect to the underlying base-level MDP.

Second, we can apply various forms of domain-specific knowledge. For instance, the values of several exit states for a region S_i may not be known, but we may know that these values are (approximately) the same (e.g., they are equidistant from any rewarding or dangerous states). This effectively reduces the dimensionality of the required grid. Tighter constraints on the value function can reduce the range of values that need to be tried. Furthermore, in circumstances where no reward can be obtained within the region, only

7. Bounds on the value function are easily obtainable using the maximum and minimum rewards.

differences in the *relative values* of exit states impact the local policy: this too can reduce the number of macros needed.

4.3 Heuristic Macro Sets

The systematic coverage techniques (policy or ε-coverage) can be computationally very costly to apply in practical settings. Thus, unless tight constraints on the value function or policy constraints are known, this can involve substantial overhead and, in many instances, be unprofitable. *Heuristic* methods for macro generation can alleviate this difficulty if they require the construction of a small number of macros. We propose a strategy (based in part on the proposal of Precup, Sutton, and Singh [31]) that uses $|XPer(S_i)| + 1$ macros for every region S_i: one macro per exit state leading the agent towards that exit, plus a "stay-in-region" macro encouraging the agent not to move out of the region. The set of heuristic macros is generated by seeding a local MDP with appropriate values: using V_{max} for the target exit and V_{min} for all other exits when generating the macro for a specific target exit; and using V_{min} values for all exits when generating the "stay-in-region" macro.

In general, the above heuristic strategy assures that exits and potential goals within the region will not be overlooked while planning at the abstract level. Note, however, that this technique does not guarantee that the necessary coverage will be obtained. For example, while implementing a policy to exit in one way, the agent may find itself actually "slipping" closer to another exit due to uncertainty in its actions. However, the policy will ensure the agent persists in its attempt to leave as planned. If both exit states have equal value, forcing the agent to choose one or the other can be far from optimal. Instead, we would like to use a third macro that takes the agent to the *nearest* exit. However, we cannot discard the original macros unless we know *in advance* that the values are similar. In addition, unless one accounts for potential variability in the actual value assigned to an exit state, sound decisions to stay within a region or leave it cannot be made.

4.4 Iterative Macro Refinement

In general, it is hard to generate a small set of good macros in one step. This problem can be alleviated by constructing a good set of macros gradually using iterative macro refinement techniques. A simple refinement strategy uses the value function produced by solving the abstract MDP as seeds for an entirely new set of macros. In particular, we choose an initial set of seeds, generate a single macro per region, then solve the induced abstract MDP. The resulting value function is used as a seed to generate a new set of macros (again one per region), and the new abstract MDP is solved.

This iterative macro-refinement method is a special case of asynchronous policy iteration [2] and is in many respects similar to Dantzig-Wolfe decomposition techniques [10, 23]. The method guarantees that after every refinement step the new macro-policy obtained by solving the abstract MDP improves over the macro-policy used in the previous step and eventually, it converges to the optimal policy. This is captured in the following theorem.

Theorem 7 *The simple iterative macro-refinement method converges to the optimal policy and the new macro-policy obtained after the refinement step is no worse than the previous step's macro-policy.*

Proof The fact that the new macro-policy improves the previous step policy follows directly from Theorem 5. Now we need to show that in the case that all local macros remain unchanged in two consecutive steps the solution policy equals the optimal policy. Assume that this does not hold and that there is a policy that is better than the policy π^* found by the iterative refinement method.[8] But then there is at least one state s for which the current action choice dictated by the policy π^* can be improved locally (as follows from the policy improvement theorem used in the standard policy iteration technique), i.e. there exist an action a such that $Q(s,a) > Q(s,\pi^*(s))$. But this is a contradiction as such an action would be found and incorporated into π^* by solving the local MDP with the same set of seeds. Thus the policy found by the simple macro-refinement method must be optimal. □

There are various modifications of the basic macro-refinement method one can apply to solve MDPs. One such technique involves refinement of the grid or mesh used to generate seed values for local MDPs. If knowledge of the value function is sparse, we may generate a set of macros using a coarse grid for a certain region (thus generating few macros). The abstract MDP can then be solved using these macros, and the abstract value function, with error bounds determined by the coarseness of the grid, will generally provide substantially improved knowledge of the true value function. This can be used to generate new seed values, with much finer precision, over a greatly reduced range, which in turn can be used to generate a small number of macros with greatly improved accuracy.

In general, iterative macro-refinement methods overcome the threat of poor initial seeding (and the generation of poor macros) by gradually improving the macro set using information as it becomes available. This requires the repeated computation of a macro model for every newly generated macro-action, which may limit their applicability.

5. Multiple MDPs and the Reuse of Macros

Generating a set macro-actions and constructing their transition and reward models is an intensive process, requiring explicit state-space enumeration. In many instances the overhead associated with this process will outweigh any speed-up macros can provide in solving the underlying MDP. Thus our hierarchical approach (or any approach requiring macro model generation) may not be worthwhile as a technique for solving a single MDP.

The main reason to incur the overhead of macro construction lies in the reuse of macros to solve multiple related MDPs. For example, in our running example, the robot may have constructed a policy that gets it to the goal consistently, but at some point the goal location might change, or the penalties associated with other locations might be revised,

8. Note that the macro-refinement method generates only one macro per region, which means that the local policy is always the part of the solution.

or perhaps the environment (or the robot's abilities) might change so that the uncertainty associated with its moves at particular locations increases. Any of these changes requires the solution of a new MDP, reflecting a change in reward structure or change in system dynamics. However, the changes to the MDP are often *local*: the reward function and the dynamics remain the same in all but a few regions of state space. For instance, it may be that the goal location in our navigation example moves within room 1, but no other part of the reward function changes.

Local changes in MDP structure can induce global changes in the value function (and can induce dramatic qualitative changes in the optimal behavior). If macros have been generated for a region such that they cover a set of different behaviors, they can be applied and reused in solving these revised MDPs. However, there is one impediment to the application of macro-actions to revised MDPs, namely, the fact that revising an MDP requires that the local information (rewards or dynamics) for some region(s) must change. For example, while the macros developed for most regions (rooms) in our navigation problem can be reused, those generated for room 1 do not reflect the revisions in the rewards, and the goal location and must be revised. Our objective is to revise the model such that solution efficiency and quality is affected as little as possible.

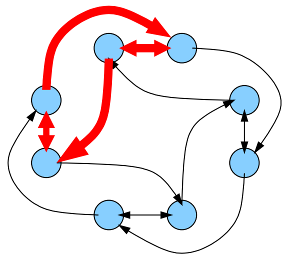
In this section, we investigate two models for reusing macros in the abstract MDP setting: the *locally revised abstract MDP* method; and the *hybrid MDP* model.

5.1 Locally-revised Abstract MDPs

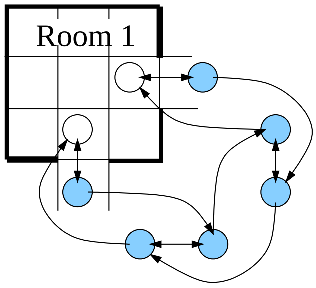
One way to account for local changes in the reward and transition functions is to generate a new set of macros for all regions in which the MDP has changed. More precisely, for each region in which the reward or transition function has changed, we first generate a new set of macros using our macro generation technique of choice, create the macro parameters for these new macros, then replace the actions in the abstract MDP with these newly created macros to obtain a *locally revised abstract MDP*. The process is illustrated in Figure 6(a), where changes in the dynamics or rewards in room 1 lead to changes in the macros associated with that abstract state. Formally, we have:

Definition Let $\Pi = \{S_1, \cdots, S_n\}$ be a decomposition of MDP $M = \langle S, A, T, R \rangle$, and let $M' = \langle S', A', T', R' \rangle$ be the abstract MDP induced by Π and macro-action set $\mathbf{A} = \{A_i : i \le n\}$. Let $\overline{M} = \langle S, A, \overline{T}, \overline{R} \rangle$ be a *local revision* of M with regard to region S_i; that is, $T(s,a,t) = \overline{T}(s,a,t)$ and $R(s,a) = \overline{R}(s,a)$ for all $s \notin S_i$ and all $a \in A$. Let $MC(S_i) = \{A''_i\}$ be a set of macro-actions for the (revised) region S_i obtained through some macro-generation method. The *locally revised abstract MDP* $M^* = \langle S^*, A^*, T^*, R^* \rangle$ with respect to $\overline{M}$ and $MC(S_i)$ is given by:

- $S^* = S'$;
- $A^* = \cup\{A_j \in \mathbf{A} : j \ne i\} \cup A''_i$, such that $\pi_j^k \in A_j$ is feasible only at states $s \in EPer(S_j)$, and $\pi_i^k \in A''_i$ is feasible only at $s \in EPer(S_i)$;
- $T^*(s, \pi_j^k, t)$ is given by the discounted transition model for $\pi_j^k \in A_j$, for any $s \in EPer(S_j)$ and $t \in XPer(S_j)$; and



(a) locally-revised abstract MDP



(b) hybrid MDP

Figure 6: Two methods for adapting an abstract MDP to local changes.

- $R^*(s, \pi_j^k)$ is given by the discounted reward model for π_j^k for any $s \in S_j$.

The obvious advantage of using a locally revised abstract MDP is that for on-line computations we do not have to modify macros for the regions that are not modified. The drawback is that the computational cost (and delays) incurred during the generation of the new set of macros for all revised regions may adversely affect the time required for the on-line solution of the revised MDP. Since the computational cost of this step depends on the size of the macro set generated for all revised regions, it may easily outweigh the benefit of the reduced state space provided by the abstract MDP. Typically, the approach is useful and can significantly improve on the base-level MDP if the newly generated macro sets are small and provide adequate coverage, and the total number of peripheral states of the abstract model is small compared to the number of states of the base-level model.

5.2 Hybrid MDPs

An alternative approach to solving a revised MDP is to solve the problem in one step using a *hybrid MDP* that combines both abstract and base-level states. Intuitively, when the dynamics (or reward function) in a specific region of the base-level MDP is revised, we can solve the new MDP by retaining the macros for all unaltered regions, and simply re-solving the base-level decision problem in the region that has changed. The abstract MDP and macros can be used to summarize all relevant information at the unaltered states. We thus end up with an MDP with base-level states from the altered region(s), and abstract states corresponding to the unaltered regions. This is depicted graphically in Figure 6(b), where again the dynamics have changed only within room 1. Formally, we have:

Definition Let $\Pi = \{S_1, \cdots, S_n\}$ be a decomposition of MDP $M = \langle S, A, T, R\rangle$, and let $M' = \langle S', A', T', R'\rangle$ be the abstract MDP induced by Π and macro set $\mathbf{A} = \{A_i : i \leq n\}$. Let $\overline{M} = \langle S, A, \overline{T}, \overline{R}\rangle$ be a *local revision* of M with regard to region S_i; that is,

$T(s,a,t) = \overline{T}(s,a,t)$ and $R(s,a) = \overline{R}(s,a)$ for all $s \notin S_i$ and all $a \in A$. The *hybrid expansion* $M^* = \langle S^*, A^*, T^*, R^* \rangle$ of M' by $\overline{M}$ is:

- $S^* = Per_\Pi(S) \cup S_i$;
- $A^* = \cup\{A_j \in \mathbf{A} : j \neq i\} \cup A$, where $\pi_j^k \in A_j$ is feasible only at states $s \in EPer(S_j)$, and $a \in A$ is feasible only at S_i;
- $T^*(s, \pi_j^k, t)$ is given by the discounted transition model for π_j^k, for any $s \in EPer(S_j)$ and $t \in XPer(S_j)$ $(j \neq i)$; $T^*(s, \pi_j^k, t) = 0$ for any $t \notin XPer(S_j)$; $T^*(s,a,t) = \overline{T}(s,a,t)$ for any $s \in S_i$ and $t \in S^*$; and
- $R^*(s, \pi_j^k)$ is given by the discounted reward model for π_j^k for any $s \in S_j (j \neq i)$, while $R^*(s,a) = \overline{R}(s,a)$ for any $s \in S_i$.

The hybrid MDP M^*, constructed when the structure within region S_i changes, consists of the original abstract MDP with the abstract states in $EPer(S_i)$ replaced by the region S_i itself. We note that this expansion is easily defined for changes in any number of regions as well as for a hierarchy of abstract MDPs.

Hybrid MDPs have a considerable advantage over the base-level MDP when real-time response is required to changing circumstances. Given a new MDP M_i that differs from the original MDP M in a single region S_i (or, more generally, some small set of regions), this new problem can be solved using a hybrid MDP of size $|S'| + |S_i - EPer(S_i)|$ (recall S' is the set of peripheral states, or states in the abstract MDP). For example, if an MDP is partitioned into k regions of roughly uniform size, and the average size of the entrance periphery of any region is p, then a hybrid MDP with one expanded region has roughly $kp + \frac{|S|}{k}$ states. Without the use of macros and abstract/hybrid MDPs, the solution of a new problem requires value or policy iteration over the entire state space of size $|S|$. Thus a new problem can be solved much more quickly as compared to the base-level MDP. In addition, the derived policy on S_i can be cached as a macro-action to be used in the future when the MDP changes in a different region.

5.3 Experimental results

To illustrate the potential of macro-actions to accelerate the solution of multiple, related MDPs, we compared solutions times for base-level MDPs with both the locally revised abstract MDP and the hybrid MDP on three sequences of related problems. We examined three agent navigation problems of increasing complexity, shown in Figure 7: Maze 36 with 36 states and 4 regions; Maze 66 with 66 states and 7 regions; and Maze 121 with 121 states and 11 regions. In each instance, the underlying MDP was modified locally by changing the goal, represented by a zero-cost absorbing state (this required changes to both the dynamics and the reward model).

Table 1 summarizes results obtained for a sequence of 25 problem instances (with different randomly selected goal states) and four different solution methods: (1) solving the base-level MDPs for each instance; (2) solving the locally revised abstract MDPs for each instance, using one macro-action for the revised goal region; (3) solving the locally revised

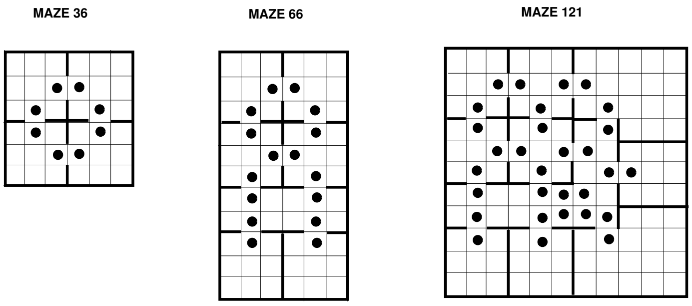


Figure 7: Problems used to test the benefits of macro-reuse. Circles denote peripheral states assumed by the hybrid-MDP method.

abstract MDP for each instance, using full heuristic macro-coverage (from Section 4) in the revised region; and (4) solving the hybrid MDP for each instance.

Each region S_i in the abstract MDP is covered initially by a heuristic set of macro-actions (containing $|XPer(S_i)|+1$ macros), as described in Section 4. The macro set and its parameters were computed off-line. The delay columns in the table list the times it took to compute the initial macro set for all abstract methods. Only the local revisions to models (due to the changes in the goal location) affected the on-line computational cost. These are summarized in average time entries in the table.

To illustrate the dependency between the number of macros in a revised macro set and the on-line computational cost for generating such set, we used two versions of the locally revised abstract MDP approach, one in which the revised goal region S_i was always covered using full heuristic coverage with $XPer(S_i) + 1$ macros, and the other in which only the "stay-in-region" macro was used.[9]

To solve an MDP (whether it is a standard, abstract, or hybrid MDP) we used value iteration. As an initial value function estimate for the revised models, we always choose the solution for the original MDP. The iteration was stopped dynamically whenever the precision of 0.01 cost units was achieved.

The results illustrate that the two reuse models, given a set of suitable macros, can solve new problem instances much more quickly than the MDP with the original state and action spaces (compare average time entries in the table). We also see that the savings offered by both models are greater for larger problems, exactly as expected. This is due to the fact that local changes affect a significantly smaller proportion of the original model in larger MDPs than in smaller MDPs. For both macro-reuse models this means that most of the

9. Since we want to solve a sequence of goal achievement tasks, a "stay-in-region" macro from our heuristic coverage is the macro that encourages the agent towards the goal most and thus reflects best our intentions. All other macros in the heuristic coverage, in addition to the goal, try to press the agent towards one of the exits. Thus we always choose the "stay-in-region" macro to define the one-macro coverage in our experiments.

MDP model	Maze 36			Maze 66			Maze 121		
	delay	av.time	AEC	delay	av.time	AEC	delay	av.time	AEC
base	0	1.22	5.96	0	2.61	8.55	0	5.94	9.96
revised abstract (1 macro)	5.52	0.91	6.12	12.62	1.73	9.92	24.47	3.25	10.94
revised abstract (heur. coverage)	5.52	1.52	6.06	12.62	2.83	9.85	24.47	4.98	10.88
hybrid	5.52	0.96	6.01	12.62	2.04	9.73	24.47	4.89	10.72

Table 1: Results obtained by standard MDP, revised abstract MDP and hybrid MDP methods on 25 randomly selected goals and three navigation problems. The delay (in seconds) measures the time spent to prepare the initial set of macro-actions, av. time is the average time (in seconds) to obtain the solution for the revised model (with the same precision stopping criterion), AEC measures the solution quality, and is computed by averaging expected cost over all peripheral states and task instances.

MDP model	Maze 36	Maze 66	Maze 121
revised abstract (1 macro)	18	15	10
revised abstract (heur. coverage)	-	-	26
hybrid	22	23	24

Table 2: Amortization thresholds for sequence of related problems on three maze problems. The threshold reflects the number of tasks one has to solve to beat the sequence of the base-level MDP tasks.

structure of the abstract MDP is preserved and only the regions in which the change has occurred are elaborated.

Generating a set of initial macros for both of the reuse frameworks can be quite costly as witnessed by initial delays in Table 1. However, if the primary macro construction process is performed off-line, this delay may be unimportant in relation to the improved ability to solve new problem instances quickly. Alternatively, the initial delay can be justified when the computational cost could be amortized over multiple problem instances. Table 2 illustrates the average number of tasks we have to solve so that the reuse method would start to dominate in terms of the total solution time (counting both the initial macro-generation delay and time to solve n tasks). For example, the hybrid MDP improves on the base model after 22, 23, and 24 tasks are solved for Maze 36, Maze 66, and Maze 121, respectively. The situation is slightly different for the revised abstract model with the full heuristic macro-coverage; the on-line revision is too costly for the first two problems and it starts to pay off only on Maze 121. Notice that the amortization threshold (the number of tasks after which the initial macro preparation delay "pays off") decreases with the problem size for the revised abstract model and increases very slowly for the hybrid model, even though this sequence of problems is such that a more complex maze has roughly double the state space size of its predecessor. This trend is very promising for the application of macros in very large domains with multiple related tasks.

The results in Tables 1 and 2 illustrate, in addition to the computational advantage of macro-action reuse, a spectrum of multiway trade-offs among different models. A relatively straightforward one is the explanation of the effect of macro-coverage complexity on the solution time of the revised abstract MDPs. Simply, solutions are slower if a more complex macro-coverage is used for a modified region. This can be seen by comparing average running times for solving the revised abstract model through single-macro versus full heuristic macro-set approaches.

Relations between the revised abstract and hybrid models and analysis of their advantages are much harder to characterize. Notice, for example, that the difference in solution time between the hybrid and revised abstract approach with the full heuristic coverage shrinks from Maze 66 to Maze 121. This phenomenon can be explained by the addition of two large rooms with small peripheries to the Maze 66 layout. In the case of the hybrid model, the addition of large rooms increases the size of the state space of the hybrid MDP we have to solve whenever the goal is in one of these rooms. In contrast to this, a small number of peripheral states and sparse heuristic macro-coverage of the modified region lead to: (1) smaller on-line delays for computing the new set of macros of the revised region; and (2) much simpler revised abstract MDPs with smaller number of states. More generally, the on-line solution of the revised abstract MDPs is affected by—in addition to the obvious parameters, like number of states and actions—the number of peripheral states of the region together with the precision of the coverage. On the other hand, the hybrid model is independent of the number of peripheral states, as well as any on-line precision parameters. Thus, one should expect the revised abstract model to perform better in cases in which modifications involve regions with a small number of peripheral states (they lead to a smaller macro-coverage) and small state spaces. On the other hand, the hybrid model should do better if regions are small and their peripheries are larger.

In terms of the solution quality, the two frameworks used in our experiments rely on a heuristically generated set of macros. Although, the macro set is relatively small, it performed quite well on the set of maze navigation problems we tested. This is documented by comparing AEC scores, measuring average expected cost for all peripheral states and for 25 randomly generated goal tasks. The slight increase in the cost score for larger problems is caused by an increase in distances between peripheral and possible goal states.

Analyzing and quantifying the trade-offs among different reuse models exactly and ahead of time is a hard task. It is also possible that for the same initial MDP, the revisions in one region are better handled by one reuse approach, while the other reuse approach is more convenient for the other region. Since it is hard to guess which modification is better in advance we can make our choice only after we learn more about possible solutions. Simply, by solving a sequence of related planning tasks we can first obtain an estimate of solution times for each modified region and for both revision methods, and then choose the one with the best solution time.[10] In a similar way, we may obtain estimates of the solution quality for each region and technique and combine these with solution-time estimates for the best cost-benefit revision approach.

6. Conclusions

We have analyzed two possible models for solving MDPs using macro-actions: the augmented MDP model, in which the set of actions is enhanced by a set of macro-actions representing a local policies; and the abstract MDP model, in which only macro-actions are used during planning. The augmented MDP does not reduce the size of the state space, instead reducing the number of iterations one needs to solve an MDP by allowing values to propagate more quickly to distant states. The abstract MDP works with macro-actions only and allows one to reduce significantly the size of the state space by restricting decision making to the peripheral states defined by a region-based decomposition of the MDP, and restricting choices to the set of macro actions. The abstract approach forms the basis of various hierarchical MDP solution techniques. Its main deficiency lies in the fact that control may be suboptimal due to the restrictions on behavior dictated by macro-actions. We have elaborated conditions and macro construction techniques that provide guarantees on solution quality. Within this model, anytime tradeoffs can be made rather easily. Furthermore, with the locally-revised abstract and *hybrid* MDPs we have techniques that allows macros to be reused to solve multiple MDPs, providing for fast, on-line decision making, and allowing macro construction costs to be amortized over many problem solving episodes.

There are a number of questions and open issues that remain to be addressed and many interesting directions in which this work can be extended. For example, apart from the handcrafted decompositions we used, one can imagine several strategies for automatic decomposition. There are multiple tradeoffs that need to be considered in such a case: larger regions often lead to smaller peripheries, which result in smaller abstract MDPs (which in turn can be solved more readily), and increase the odds that a revision of the MDP will be localized to a small number of regions; smaller regions, in contrast, allow macros to be generated more quickly when revisions are required. A systematic investigation of tradeoffs is needed. When solving multiple MDPs with a known distribution over problem instances, the savings associated with macro reuse adds another dimension to the problem of automatic partitioning: if we have information pertaining to the ways in which system dynamics and reward functions may be revised, we'd like to exploit it in forming our decomposition of state space and the associated macro-actions. One method for automatic construction of a multi-level hierarchy is explored by Moore, Baird, and Kaelbling [26], for the special case of a domain in which the only problems to be solved are navigation problems with the objective of getting from one location to another as quickly as possible.

10. The idea of adaptive adjustment becomes harder and may not work if modifications affect many regions at once. In that case, it may not be feasible to remember statistics for all possible combination of regions.

Other interesting direction is related to the use of compact MDP representations (e.g., Bayesian network representations) to form decompositions and to solve local, abstract and hybrid MDPs. Related is the issue of compact representation of macros and macro models without explicit enumeration of the state space. Dietterich's MaxQ method [12], for example, is able to use different state-space approximations in different regions of the state space.

As well as generalization over similar states within a region, we may also be able to exploit generalization over similar regions: for instance, after learning a policy to navigate through a room and out a door, it might be possible to apply it to navigating out other doors of other rooms. Such generalizations will hinge on appropriate representations of the state and actions spaces within regions, so that their commonalities are revealed. However, if this type of generalization is possible, it can justify the computational expense of (automatic) macro generation even in a single MDP, since the creation of macros may require local optimization only over a small subset of state space, with macros created for certain regions applied with suitable modifications to other regions.

Hierarchy and abstraction play a crucial role in human planning and an increasingly important role in automatic planning. This paper has explored the use of abstraction in time (macro actions) and in space (regions) to increase efficiency of planning in uncertain domains. The time taken to form an abstraction is usually only warranted in the case in which it can be re-used, but we believe that this is the typical situation in most planning domains. There are many further refinements and extensions that can be made to this approach; they will be necessary before we have a truly practical method for solving very large uncertain planning problems.

Acknowledgments

We would like to thank Ronald Parr for his motivating discussion on macro-actions and for pointing out additional references.

This work was supported in part by DARPA/Rome Labs Planning Initiative grant F30602-95-1-0020 and in parts by NSF grants IRI-9453383 and IRI-9312395.

Craig Boutilier was supported by NSERC and the Institute for Robotics and Intelligent Systems. Some of this work was undertaken while the author was visiting Brown University. Thanks also to the generous support of the Killam Foundation.

Nicolas Meuleau was also supported by a Marie Curie Fellowship (Contract No. HPMFCT-2000-00230). The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication.

References

[1] Richard E. Bellman. *Dynamic programming.* Princeton University Press, Princeton, NJ, 1957.

[2] Dimitri P. Bertsekas and John. N. Tsitsiklis. *Neuro-dynamic Programming.* Athena Scientific, 1996.

[3] Craig Boutilier, Ronen I. Brafman, and Christopher Geib. Structured reachability analysis for Markov decision processes. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 24–32, Madison, WI, 1998.

[4] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[5] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1104–1111, Montreal, 1995.

[6] Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 355–362, Austin, TX, 2000.

[7] Peter Dayan and Geoffrey Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5*, pages 271–278, 1993.

[8] Thomas Dean and Robert Givan. Model minimization in Markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 106–111, 1997.

[9] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76:35–74, 1995.

[10] Thomas Dean and Shieu-Hong Lin. Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1121–1127, Montreal, 1995.

[11] Richard Dearden and Craig Boutilier. Abstraction and approximate decision theoretic planning. *Artificial Intelligence*, 89:219–283, 1997.

[12] Thomas G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[13] Richard Fikes, Peter Hart, and Nils Nilsson. Learning and executing generalized robot plan. *Artificial Intelligence*, 3:251–288, 1972.

[14] J. P. Forestier and P. Varaiya. Multilayer control of large Markov chains. *IEEE Transactions on Automatic Control*, 23:298–304, 1978.

[15] Geoffrey J. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, 1995.

[16] Milos Hauskrecht, Nicolas Meuleau, Craig Boutilier, Leslie Pack Kaelbling, and Thomas Dean. Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 220–229, Madison, WI, 1998.

[17] Ronald A. Howard. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, 1960.

[18] Leslie Pack Kaelbling. Hierarchical reinforcement learning: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 167–173, 1993.

[19] Craig Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 686–691, 1991.

[20] Daphne Koller and Ronald Parr. Policy iteration for factored MDPs. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 326–334, Stanford,CA, 2000.

[21] Richard Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[22] Richard Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

[23] Harold J. Kushner and Ching-Hui Chen. Decomposition of systems governed by Markov chains. *IEEE Transactions on Automatic Control*, 19(5):501–507, 1974.

[24] Nicolas Meuleau, Milos Hauskrecht, Kee-Eung Kim, Leonid Peshkin, Craig Boutilier, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Solving very large weakly coupled Markov decision processes. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 162–172, 1998.

[25] Steven Minton. Selectively generalizing plans for problem solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 596–599, Boston, 1985.

[26] Andrew Moore, Leemon Baird, and Leslie P. Kaelbling. Multi-value functions: Efficient automatic action hierarchies for multiple goal MDPs. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1316–1323, Stockholm,, 1999.

[27] Ronald Parr. Flexible decomposition algorithms for weakly coupled Markov decision processes. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, 1998.

[28] Ronald Parr. *Hierarchical control and learning for Markov decision processes.* PhD thesis, University of California, Berkeley, 1998.

[29] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 11*. MIT Press, Cambridge, 1998.

[30] Doina Precup and Richard S. Sutton. Multi-time models for temporally abstract planning. In *Advances in Neural Information Processing Systems 11*. MIT Press, Cambridge, 1998.

[31] Doina Precup, Richard S. Sutton, and Satinder Singh. Theoretical results on reinforcement learning with temporally abstract behaviors. In *Proceedings of the Tenth European Conference on Machine Learning*, Chemnitz, Germany, 1998. To appear.

[32] Martin L. Puterman. *Markov decision processes: discrete stochastic dynamic programming.* John Wiley, New York, 1994.

[33] Earl Sacerdoti. Planning in hierarchy of abstraction spaces. *Artificial Intelligence*, 7:231–277, 1974.

[34] Satinder P. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339, 1992.

[35] Richard S. Sutton. Td models: Modeling the world at a mixture of time scales. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 531–539, Lake Tahoe, Nevada, 1995.

[36] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and Semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. *Artificial Intelligence*, 112:181–211, 1999.

[37] Sebastian Thrun and Anton Schwartz. Finding structure in reinforcement learning. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, Cambridge, MA, 1995. MIT Press.

[38] John N. Tsitsiklis and Benjamin Van Roy. Feature-based methods for large-scale dynamic programming. *Machine Learning*, 22:59–94, 1996.