Spatial and Temporal Abstractions in POMDPs Applied to Robot Navigation

Georgios Theocharous georgios.theocharous@intel.com
Intel Corporation
Santa Clara, CA 95054

Sridhar Mahadevan mahadeva@cs.umass.edu
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Leslie Pack Kaelbling lpk@csail.mit.edu
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Partially observable Markov decision processes (POMDPs) are a well studied paradigm for programming autonomous robots, where the robot sequentially chooses actions to achieve long term goals efficiently. Unfortunately, for real world robots and other similar domains, the uncertain outcomes of the actions and the fact that the true world state may not be completely observable make learning of models of the world extremely difficult, and using them algorithmically infeasible. In this paper we show that learning POMDP models and planning with them can become significantly easier when we incorporate into our algorithms the notions of spatial and temporal abstraction. We demonstrate the superiority of our algorithms by comparing them with previous flat approaches for large scale robot navigation.

1. Introduction

Modeling an agent and its interaction with its environment through actions, perceptions and rewards is a unifying framework for AI (Russell & Norvig, 2003). The fundamental problem faced by all intelligent agents (e.g., physical systems, including robots, softbots, and automated manufacturing systems) is the following: how should an agent choose actions based on its perceptions such that it achieves its long-term goals? One way to formalize this planning problem is to design the agent so that it chooses actions that maximize the long-term expected reward, where rewards represent desirable states of the world. Formulating planning as the problem of maximizing long-term expected sum of rewards is a flexible framework since it prevents agents from acting myopically to pick short-term rewarding actions over decisions that are more rewarding on the long run. Markov decision processes (MDPs) are a widely adopted paradigm for modeling planning problems as maximizing long-term rewards (Sutton & Barto, 1998).

An MDP is specified by a set of states, a set of actions that represent transitions between states, and a reward function on states (or state action pairs). Knowledge of the current state is sufficient for predicting the distribution over the next state (the Markov property).

In turn, the Markov assumption leads to dynamic programming methods for computing optimal policies. Optimal policies are mappings from states to actions, which maximize the long-term reward received by the agent.

A partially observable MDP (POMDP) models the situation where agent's perceptions do not directly reveal the state of the world. However, a probability distribution over the hidden states of the POMDP model (called the "belief state") turns out to be a sufficient information state, in that it implicitly represents histories of observations and actions that map to the same belief state given some initial belief state (Aström, 1965). Belief states satisfy the Markov assumption in that knowledge of the previous belief state, action taken, and observation perceived is sufficient for computing the current belief state. Unfortunately, even though the Markov assumption is satisfied in belief space, computing optimal decisions from belief states is computationally intractable, since the number of belief states is infinite.

Learning the parameters of POMDP models, state-transition and state-observation functions, is also a difficult problem. The learning difficulty is once more a consequence of partial observability, since the agent does not have direct access to the states. A general learning method for POMDPs is the Baum-Welch algorithm (Baum & Sell, 1968), (Rabiner, 1989), which is an expectation maximization (EM) algorithm (Dempster, Laird, & Rubin, 1977). Unfortunately learning flat POMDP models for real domains, using the Baum-Welch algorithm, becomes increasingly harder as the domains get larger, mainly due to local optima (Koenig & Simmons, 1998), (Shatkay, 1998).

In this paper we investigate how spatial and temporal abstractions can make learning and planning in POMDPs practical for large scale domains. The contributions of this work can be summarized as follows:

- We formally introduce and describe the hierarchical partially observable Markov decision process model (H-POMDP) (Theocharous, Rohanimanesh, & Mahadevan, 2001), (Theocharous, 2002). This model is derived from the hierarchical hidden Markov model with the addition of actions and rewards. Unlike flat partially observable Markov decision process models, it provides both spatial and temporal abstraction. The abstraction is achieved by low resolution abstract states, which group together finer resolution states in a tree-like fashion.

- We derive an EM algorithm for learning the parameters of H-POMDPs. The algorithm runs in time linear in the number of states but cubic in the length of the training sequences. It is presented in Section 4 and the empirical results are described in Section 7.

- To avoid the cubic time complexity we introduce two approximate training methods which take advantage of short-length training sequences. In the first method, which we call *selective-training*, only selected parts of an overall H-POMDP are allowed to be trained for a given training sequence. In the second method, which we call *reuse-training*, submodels of an H-POMDP model are first trained separately and then reused in the overall model. Both algorithms are described in Section 4 and their empirical evaluation is described in Section 7.

- We explore the advantages of representing H-POMDPs as dynamic Bayesian networks (DBNs). Representing H-POMDPs as DBNS allows us to train them in linear time with respect to the length of training sequences but quadratic time with respect to the number of states. The DBN formulation provides extensions, such as parameter tying and factoring of variables, which further reduces the time and sample complexity. The DBN representation of H-POMDPs is described in Section 3 and the experiments in Section 7.2.

- We derive two planning and execution algorithms for approximating the optimal policy in a given H-POMDP. The planning and execution algorithms combine hierarchical methods for solving Markov decision processes, such as the options framework (Sutton, Precup, & Singh, 1999), with approximate flat POMDP solutions (Koenig & Simmons, 1998) (Section 5.1).

- We describe a reinforcement learning algorithm, which learns from sample trajectories about what subspace of the belief space is typically inhabited by the process and how to choose actions. When we add macro-actions to the problem, the algorithm considers an even smaller part of the belief space, learns POMDP policies faster, and can do information gathering more efficiently (Section 5.2).

- We conduct a detailed experimental study of the learning algorithms for indoor robot navigation. Experiments are presented for various large-scale models both in simulation and on a real robot platform. The experiments demonstrate that our learning algorithms can efficiently improve initial H-POMDP models which results in better robot localization (Section 7). Additionally, we compare our learned models with equivalent flat POMDP models that are trained with the same data. Our approximate training methods converge faster than standard EM procedures for flat POMDP models. When the learned H-POMDP models are converted to flat, they give better robot localization than the flat POMDP models that are trained with the standard flat EM algorithm for POMDPs (Section 7).

- We apply the planning algorithms to indoor robot navigation, both in simulation and in the real world. Our algorithms take the robot to any environment state starting from no positional knowledge (uniform initial belief state). In comparison with flat POMDP heuristics, our algorithms compute plans faster, and use a smaller number of steps in executing navigation tasks (Section 7.4). The reinforcement learning algorithm we introduce learns to explicitly reason about macro-actions in belief-states and as a result exhibits information gathering behavior (Section 7.5).

2. Hierarchical models

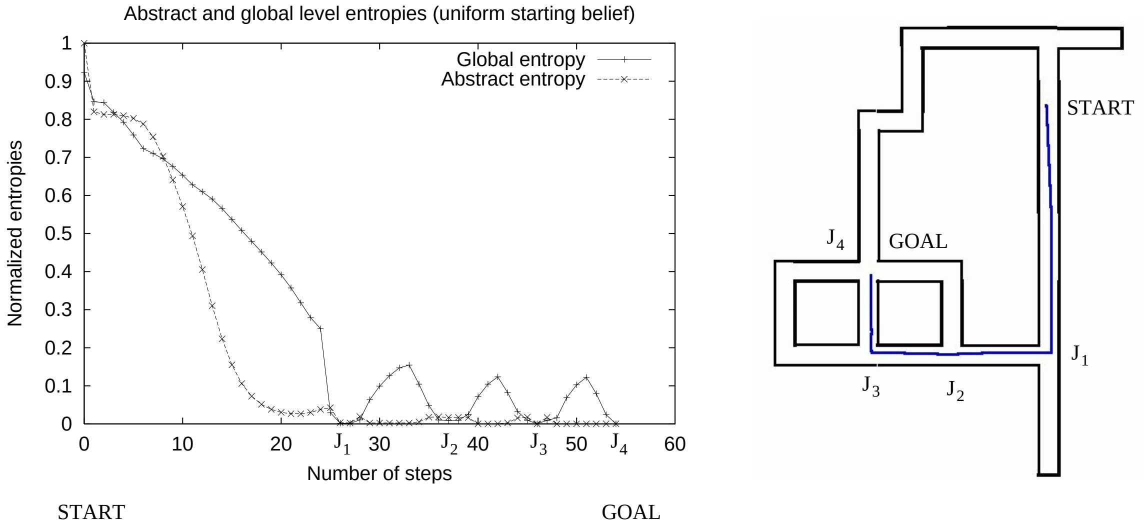
Hierarchy and abstraction have long been viewed as necessary to scale learning and planning methods to large domains. The goal of hierarchical planners is to simplify the search and reasoning process by finding abstract solutions at higher levels where the details are not computationally overwhelming, and then refining them. For example, a classical logic-based planner that uses spatial abstraction is ABSTRIPS (Sacerdoti, 1974), while a classical logic-based planner that uses temporal abstraction, in the form of hierarchical plans, is NOAH (Sacerdoti, 1975). Such systems however, do not address uncertainty or partial observability. In this paper we are investigating spatial and temporal abstraction, in terms of learning and planning, in the presence of uncertainty and partial observability. We show that in sequential decision making tasks under uncertainty, state and temporal abstractions play a larger role than simply reducing computational complexity. They help in reducing uncertainty and partial observability, which are the major obstacles in solving such tasks.

In spatial abstraction, finer grained adjacent states are grouped into higher level abstract states. Our approach is built on the assumption that in many real world domains uncertainty at an abstract level is less than uncertainty at a primitive level. Mathematically however, abstract states do not always reduce uncertainty. For example, if there are four states s_1, s_2, s_3, s_4 with probability distribution $\{0.5, 0.0, 0.5, 0.0\}$ and we group adjacent states s_1, s_2 into an abstract state A and adjacent states s_3, s_4 into an abstract state B, then the normalized entropy at the abstract level will be $-log(0.5)/log(2)$, while at the finer grain level will be $-log(0.5)/log(4)$, where normalized entropy is computed as $-\sum_{s=1}^{S} p(s)log(p(s))/log(|S|)$. Our conjecture is that in the presence of sparse connectivity such situations will not arise. By "sparse connectivity" we mean that a state cannot transition to all other states but only to its neighbors and in turn the neighbors to their neighbors. Such sparse connectivity is quite common in the real world and can easily be visualized in terms of an agent, or even us people moving through physical environments. For example, when driving in the highway we are uncertain where in the highway we are but more certain as to which highway. Having more certainty at the higher level can lead to better estimation of the true state of the agent. This in turn can lead to more informed learning, easier planning algorithms and robust plan execution.

The problem of partial observability can be reduced even further through temporal abstraction, where single step primitive actions are combined to form multi-step macro-actions. In general, any random sequence of actions is not guaranteed to reduce uncertainty, unless it takes the agent to highly distinctive states in the environment. For example, if we don't know where we are in the highway, we may choose to follow a long-term direction (e.g. go north) until we see a familiar landmark. But even if there are no distinctive landmarks, macro-actions could still help in disambiguating perceptually aliased situations. For example, if we don't know in which of two highways we are in and both highways look exactly the same, but we know that one is longer than the other, we can choose to keep going in the current highway until the distance traveled disambiguates them. Figure 1 demonstrates these intuitions in the context of robot navigation.

In addition to uncertainty reduction, hierarchy has several other practical advantages, including interpretability, ease of maintenance, and reusability. Reusability can be an important factor in solving large scale sequential decision-making problems. In learning, we can separately train the different sub-hierarchies of the complete model and reuse them in larger, complete hierarchical models. Or, if different sub-hierarchies have the same dynamics, we could simply train one of them and reuse it for all the others. In planning, we can construct abstract actions available in each abstract state. These can be reused for any global plan without the need of worrying about the details inside the abstract states.

To mathematically model spatial and temporal abstractions in the context of sequential decision-making under uncertainty, we borrow from two established models, each one providing some of the desired properties. The first is a general formalism for hierarchical control in sequential tasks under uncertainty, namely the framework of semi-Markov decision processes. The second component is a general model of hierarchical sequential



(a) (b)

Figure 1: Figure (a) shows two plots of normalized entropy, the "abstract" entropy is at the level of corridors and junctions, and the "global" entropy at the level of finer grain locations in the environment. Figure (b) shows the trace of the robot from START to GOAL. Uncertainty is lower at the abstract level than the global level. The robot's ability to execute multi-step actions before the next decision reduces its location uncertainty significantly. For example, look at the entropy at START and then at J1, after a "go-down-the-corridor" multi-step action.

processes, the hierarchical hidden Markov model. Combining the two frameworks allows us to address partial observability, and hierarchical modeling and control. Next we describe these two fundamental formalisms and other related hierarchical approaches.

2.1 Semi-Markov decision processes

In a Markov decision process at each stage, an agent observes a system's state s contained in a finite set $\mathcal{S}$, and executes an action a selected from a finite set, $\mathcal{A}_s$. The agent receives an immediate reward having expected value $R(s,a)$ and transitions to the next state s' with probability $P(s'|s,a)$. A stationary stochastic policy π specifies that the agent executes action a with probability $\pi(s,a)$ whenever it observes state s. For any policy π, $V^\pi(s)$ denotes the expected infinite-horizon discounted return from s given that the agent uses policy π. This value function is defined as:

$$V^\pi(s) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|s_t = s, \pi\},$$

where s_t is the state at stage t, r_{t+1} denotes the reward for acting at stage t and $0 \leq \gamma < 1$, denotes the discount factor. The value function can also be defined in terms of state action pairs. Given a policy π, the value of (s,a),for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}$, denoted $Q^\pi(s,a)$, is the expected infinite-horizon discounted return for executing a in state s and thereafter following π

$$Q^{\pi}(s,a) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|s_t = s, a_t = a, \pi\}$$

Value functions based on state-action pairs play an important role for on-line learning because they allow an agent to learn by sampling, whereas state-based value functions do not allow model-free learning.

The objective in an MDP is to compute an optimal policy π^*, which returns the maximum value $V^*(s)$ for each state. Such computation can be done by dynamic programming (DP) algorithms which exploit the fact that the value function satisfies the Bellman equations:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^{\pi}(s') \right],$$

$$V^*(s) = \max_{a \in \mathcal{A}} \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right]$$

and the action value equations:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a' \in \mathcal{A}} Q^*(s',a').$$

An example DP algorithm is value iteration, which successively approximates V^* as follows. At each iteration k it updates an approximation V_k of V^* by applying the following operation for each state s:

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V_k(s') \right].$$

Repeated applications of this operation converges to the unique optimal value function V^*.

An alternative approach to DP is reinforcement learning (RL). One of the most popular approaches is Q learning (Watkins, 1989), which approximates Q^* by using the following update rule:

$$Q_{k+1}(s,a) = (1 - \alpha_k) Q_k(s,a) + \alpha_k \left[r + \gamma \max_{a' \in \mathcal{A}} Q_k(s',a') \right],$$

where a was the action executed in state s, r the immediate reward received and α_k a time-varying learning parameter.

Semi-Markov decision processes are a generalization of MDPs where state transitions depend not only the state from which a transition takes place, but also the amount of time that passes between decisions. The amount of time between decisions could be continuous or discrete. In the continuous time SMDPs decision are made at discrete events (Mahadevan, Marchalleck, Das, & Gosavi, 1997), (Puterman, 1994). In the discrete-time SMDP model decisions can only be made at (positive) integer multiples of an underlying time step (Howard, 1971). In both cases it is usual to treat the system as remaining in the same state for a random waiting time, at the end of which an instantaneous transitions occurs to the next state.

Let the random variable τ represent the waiting time for state s when action a is executed. The transition probabilities generalize to give the joint probability that a transition from state s to state s' occurs after τ steps when action a is executed. We write this joint probability as $P(s', \tau|s, a)$. The expected immediate reward $R(s, a)$ now gives the amount of reward expected to accumulate over the waiting time in s given action a. The Bellman equation for V^* becomes:

$$V^*(s) = \max_{a \in \mathcal{A}} \left[R(s,a) + \sum_{s'} \gamma^\tau P(s', \tau|s,a) V^*(s') \right].$$

Q-learning extends to SMDPs as well. The update rule becomes:

$$Q_{k+1}(s,a) = (1-\alpha_k)Q_k(s,a) + \alpha_k \left[r+1, +\gamma r_{t+2} + ... + \gamma^{\tau-1} r_{t+\tau} + \gamma^\tau \max_{a' \in \mathcal{A}} Q_k(s', a') \right].$$

The temporal aspect of the SMDP formalism provides the basis for hierarchical control where temporally extended actions can be composed of sequences of finer time-scale actions. Some of the popular hierarchical frameworks include options (Sutton et al., 1999), MAXQ (Dietterich, 2000), and HAMs (Parr, 1998). Here we briefly review the options framework on which we base our hierarchical planning algorithms.

An option is either a temporally extended course of action (macro-action) or a primitive action. If an option is a macro-action, then it is a policy over an underlying MDP. Options consist of three components: a policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0,1]$, a termination condition $\beta : \mathcal{S}^+ \rightarrow [0,1]$, and an initiation set $\mathcal{I} \subseteq \mathcal{S}$. An option $< \mathcal{I}, \pi, \beta >$ can be started from any state $s \in \mathcal{I}$, at which point actions are selected according to policy π until the option terminates stochastically according to β.

To construct policies among options we first need to define reward and transition models for the options. The total reward that will be received for executing option o in some state s is defined as:

$$r_s^o = E\{r_{t+1} + \gamma r_{t+2} \cdots + \gamma^{k-1} r_{t+k} | \mathcal{E}(o,s,t)\},$$

where $t+k$ is the random time at which o terminates, and $\mathcal{E}(o,s,t)$ is the event that option o started at time t. The state transition model is defined as:

$$\mathcal{P}_{ss'}^o = \sum_{k=1}^{\infty} p(s', k)\gamma^k,$$

which is a combination of the probability value $p(s', k)$ that o terminates in state s' in k steps, together with a measure of how delayed the outcome is according to the discount factor γ. The reward and transition model for options allows us to write the Bellman equation:

$$\begin{aligned} V^\mu(s) &= E\{r_{t+1} + \gamma r_{t+2} \cdots + \gamma^{k-1} r_{t+k} + \gamma^k V^\mu(s_{t+k}) | \mathcal{E}(\mu, s, t)\} \\ &= r_s^o + \sum_{s'} \mathcal{P}_{ss'}^o V^\mu(s') \end{aligned}$$

for the value of executing an abstract policy μ at some state s and thus be able to compute policies over options.

The transition and reward models of options can be learned using an intra-option learning method (Sutton et al., 1999). We can express the option models using Bellman equations and then solve the system of linear equations either explicitly, or through dynamic programming (Bellman, 1957), (Bersekas, 1987). The Bellman equation for the reward model is:

$$r_s^o = r_s^{\pi(s)} + \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} (1 - \beta(s') r_{s'}^o)$$

and the Bellman equation for transition model is:

$$\mathcal{P}_{sx}^o = \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[(1 - \beta(s')) \mathcal{P}_{s'x}^o + \beta(s') \delta_{s'x} \right],$$

where $\delta_{s'x} = 1$ if $s' = x$ and is 0 otherwise.

2.2 Hierarchical hidden Markov models

Now we describe the second component in our approach, the HHMM. The HHMM (Fine, Singer, & Tishby, 1998) generalizes the standard hidden Markov model (HMM) (Rabiner, 1989) by allowing hidden states to represent stochastic processes themselves. HHMMs are tree structured (see Figure 2 with three types of states, production states (leaves of the tree) which emit observations, internal states which are (unobservable) hidden states that represent entire stochastic processes, and end-states which are artificial states of the model that when entered exit the (parent) abstract state they are associated with. Each production state is associated with an observation vector that maintains distribution functions for each observation defined for the model. Each internal state is associated with a horizontal transition matrix and a vertical transition vector. The horizontal transition matrix of an internal state defines the transition probabilities among its children. The vertical transition vectors define the probability that an internal state will activate any of its children. Each internal state is also associated with a single end-state child. The end-states do not produce observations and cannot be activated through a vertical transition from their parent. The HHMM is formally defined as a 5 tuple $\langle S, T, \Pi, Z, O \rangle$:

- S denotes the set of states. The function $p(s)$ denotes the parent of state s. The function $c(s, j)$ returns the j^{th} child of state s. The end-state child of an abstract state s is denoted by e^s. The set of children of a state s is denoted by C^s and the number of children by $|C^s|$.

- $T^s : \{C^s - e^s\} \times C^s \rightarrow [0, 1]$ denotes the horizontal transition functions, defined separately for each abstract state. A horizontal transition function maps each child state of s into a probability distribution over the children states of s. We write $T^s(c(s, i), c(s, j))$ to denote the horizontal transition probability from the i^{th} to the j^{th} child of state s. As an example, in Figure 2, $T^{s4}(s7, s8) = 0.6$.

- $\Pi^s : \{C^s - e^s\} \rightarrow [0, 1]$ denotes the vertical transition function for each abstract state s. This function defines the initial distribution over the children states of state s, except from the end-state child e^s. For example, in Figure 2, $\Pi^{s1}(s2) = 0.5$.

- Z denotes the set of discrete observations.

- $O^s : C^{s^{production}} \rightarrow [0,1]$ denotes a function that maps every production state (child of s) to a distribution over the observation set. We write $O^{p(s)}(s,z)$ for the probability of observing z in state s. $C^{s^{production}}$ is the set of all production states which are children of s.

Figure 2 shows a graphical representation of an example HHMM. The HHMM produces observations as follows:

1. If the current node is the root, then it chooses to activate one of its children according to the vertical transition vector from the root to its children.

2. If the child activated is a production state, it produces an observation according to an observation probability output vector. It then transitions to another state within the same level. If the state reached after the transition is the end-state, then control is returned to the parent of the end-state.

3. If the child is an abstract state then it chooses to activate one of its children. The abstract state waits until control is returned to it from its child end-state. Then it transitions to another state within the same level. If the resulting transition is to the end-state then control is returned to the parent of the abstract state.

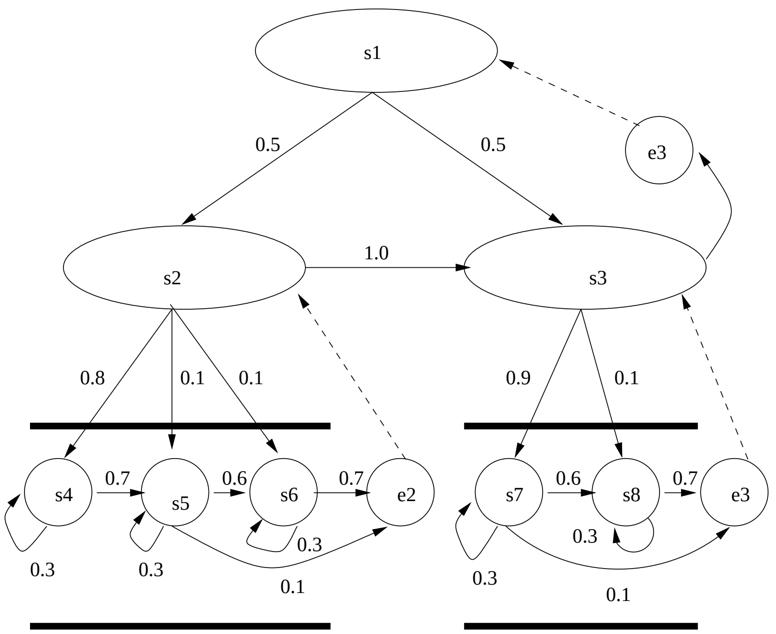


Figure 2: An example hierarchical HMM modeling two adjacent corridors. Only leaves (production) states (s4, s5, s6, s7, and s8) have associated observations. The end-states are e1, e2, and e3

2.3 Related multi-resolution hidden Markov models

HHMMs can also be viewed as a degenerate case of stochastic context free grammars (SCFGs). The generalized hierarchical Baum-Welch algorithm presented in (Fine et al., 1998) was inspired by the inside-outside algorithm which is an EM type algorithm for learning the probabilities of SCFG rules (Lari & Young, 1990), (Prescher, 2001). The inside-outside algorithm is less efficient than the EM algorithm for HHMMs because not only does it run in cubic time in terms of the length of the training sequence, but also in cubic time in the number of non-terminal symbols. EM for HHMMs is cubic in time with respect to the length of the training sequence but linear in the number of non-terminal symbols. Additionally, the likelihood of observed sequences induced by a SCFG varies dramatically with small changes in the parameters of the model. HHMMs differ from SCFGs in that they allow only for bounded stack depths.

Another way of representing HHMMs is with a special case of Dynamic Bayesian Networks (DBNs) (Murphy, 2001). In this paper we extend the H-POMDP model to a DBN representation which we describe in the next section. Figure 3 shows the DBN representation of HHMMs and their extension to H-POMDPs. The transformation of HHMMs to DBNs allows us to train them by applying standard inference techniques in Bayesian nets such as the junction-tree algorithm. The junction-tree algorithm replaces the expectation step in the hierarchical Baum-Welch algorithm. The major advantage of using the DBN representation of HHMMs is that they can be trained in linear time with respect to sequence length, while the conventional hierarchical Baum-Welch algorithm requires cubic time with respect to sequence length (Fine et al., 1998). A disadvantage is that it requires quadratic time with respect to the number of states, while the original algorithm requires linear time with respect to the number of states. This makes the original algorithm better suited for larger models and small length training sequences, while the DBN algorithm is suited for small models and long training sequences.

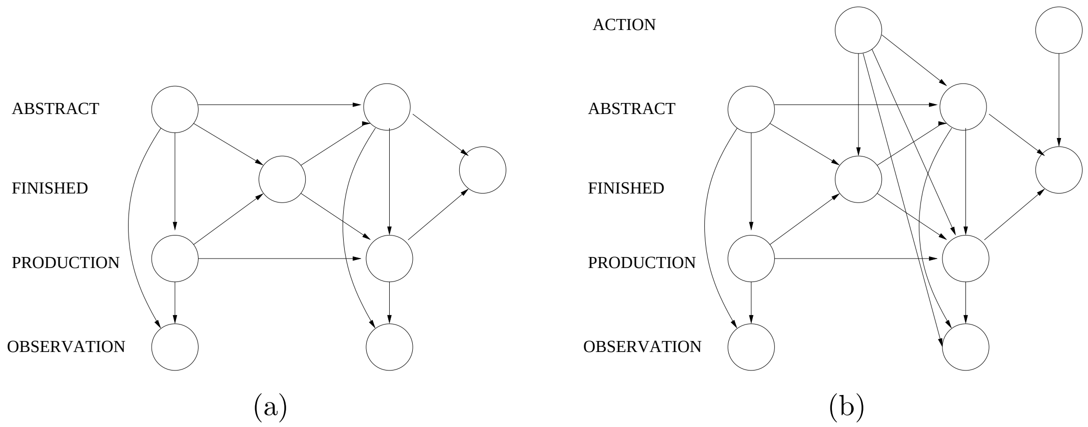


Figure 3: Figure (a) shows the DBN representation of HHMMs and Figure (b) shows the extension to H-POMDPs

Another closely related model to HHMMs is the abstract hidden Markov model (AHMM) (Bui, Venkatesh, & West, 2000), which has been used to model abstract policy recognition.

This is a DBN representation closely related to the *options* framework (Sutton et al., 1999). The states however are not fully observable and the model becomes similar to an HHMM. In the AHMM framework at each level of abstraction there is a set of abstract actions available. Plan execution starts from top of the hierarchy where the most abstract level executes an abstract action. The abstract action may be refined into other abstract actions at the next lower level of abstraction. At the lowest level only primitive actions are executed, which terminate after every time step. The DBN representation of the AHMM exhibits many characteristics which make it similar to HHMMs/H-POMDPs as shown in Figure 4. The basic differences are the following:

1. In an AHMM there is a global variable representing the world state, which every transition in the model depends on.
2. Unlike HHMMs where a process over primitive states can take multiple steps before control is transferred to the calling abstract state, in the AHMM the primitive states transfer control to the calling process every time step. In general, there are no transitions from lower level states to higher level states, which means that when an abstract state finishes, control is transferred to the higher level. In effect there are no horizontal transitions.
3. There is no spatial abstraction. Higher level states share lower level states.
4. The actions (both primitive and policies) depend on state. In H-POMDPs policy selection is not contingent on the lower-level state.

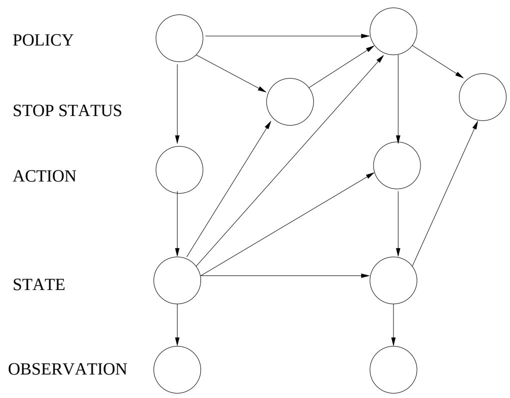


Figure 4: The figure shows a DBN representation of the AHMM. The semantics of the variables and conditional independence relations are defined in such a way that they represent hierarchical policies. These hierarchical policies define mappings of primitive and hierarchical actions on the states of the model. What makes it similar to HHMMs is the fact that the states are hidden. These models are used for hierarchical activity recognition.

Embedded hidden Markov models are another special case of the HHMM model, which were applied to face recognition (Nefian & Hayes, 1999). The basic difference is that they are always of depth 2 and the number of observations produced by each abstract state is known in advance. This leads to efficient algorithms for estimating the model parameters.

Another well studied class of multi-resolution models are segment models (Ostendorf, Digalakis, & Kimball, 1996). In the simplest segment model the self-transitions of abstract states are eliminated and state duration is modeled explicitly either in a parametric or non-parametric fashion. The reason is that self transitions induce an exponential duration while explicit durations can be arbitrary. These simple models are also called variable duration HMMs (Rabiner, 1989) (Levinson, 1986), or hidden semi-Markov models (Russell & Moore, 1985). These models can be thought as special cases of two level HHMMs. Unfortunately, the computational complexity of training durational HHMMs is $O(D^2)$ where D is the maximum allowed duration (Rabiner, 1989).

Finally, in another related hierarchical POMDP approach, (Pineau, Gordon, & Thrun, 2003) the hierarchy comes from a task-based decomposition rather than a state-based decomposition. Even though the underlying dynamics of the world are still represented as a flat POMDP, the model becomes hierarchical based on the solution the authors propose. The POMDP solution is based on first solving smaller POMDPs. The smaller POMDP problems are created based on decompositions of the action space.

3. Hierarchical partially observable Markov decision processes

The H-POMDP model can be derived from the HHMM model (Fine et al., 1998) with the addition of actions and rewards (Theocharous et al., 2001). The H-POMDP model is formally defined as follows:

- S denotes the set of states. Unlike HHMMs, we have an additional type of state called entry states.
  - *Production states* which produce observations.
  - *Exit-states* which return control to their parent abstract state when entered.
  - *Entry-states* which activate children of the abstract state that they are associated with.
  - *Abstract states* which group together *production*, *entry*, *exit*, and other *abstract* states. C_p^s denotes the production children of abstract state s. C_a^s denotes the abstract state children of abstract state s. C_x^s denotes the set of exit states which belong to children abstract states of abstract state s. C_n^s denotes the set of entry states which belong to children abstract states of abstract state s. X^s denotes the set of exit states that belong to s and N^s denotes the set of entry-states that belong to abstract state s. $p(s)$ denotes the parent state of s.
- A denotes the set of primitive actions. For example, in robot navigation, the actions could be *go-forward* one meter, *turn-left* 90 degrees, and *turn-right* 90 degrees. Primitive actions are only defined over production states.
- $T(s'|s_x, a)$ denotes the horizontal transition probabilities for primitive actions. Horizontal transition probabilities are defined for sibling states, or for a child and its parent. The transitions originate at exit-states and terminate at a sibling's entry-state,

or originate at an exit-state and terminate at the parent's exit-state. If a production state is involved then the entry or exit-state is the state itself. More specifically, if s is an abstract state, s_x denotes the current exit state. If s is a production state then s_x refers to the production state itself since there are no exit states for production states. The resulting state s' could be the entry-state of some abstract state, an exit state associated with the parent of s, or some production state.

- $V(s'_n|s_n)$ denotes the vertical transition probability between a parent and its child. It originates at an entry state of a parent and terminates at an entry state of a child. More specifically, it denotes the probability that entry state s_n, which belongs to abstract state s, will activate child state s'. If s' is an abstract state, s'_n is an entry state of state s'. If s' is a production state s'_n is the state itself.
- Z denote the set of discrete observations.
- $O(z|s,a)$ denotes the probability of observation z in production state s after action a has been taken.
- $R(s,a)$ denotes an immediate reward function defined over the production states.

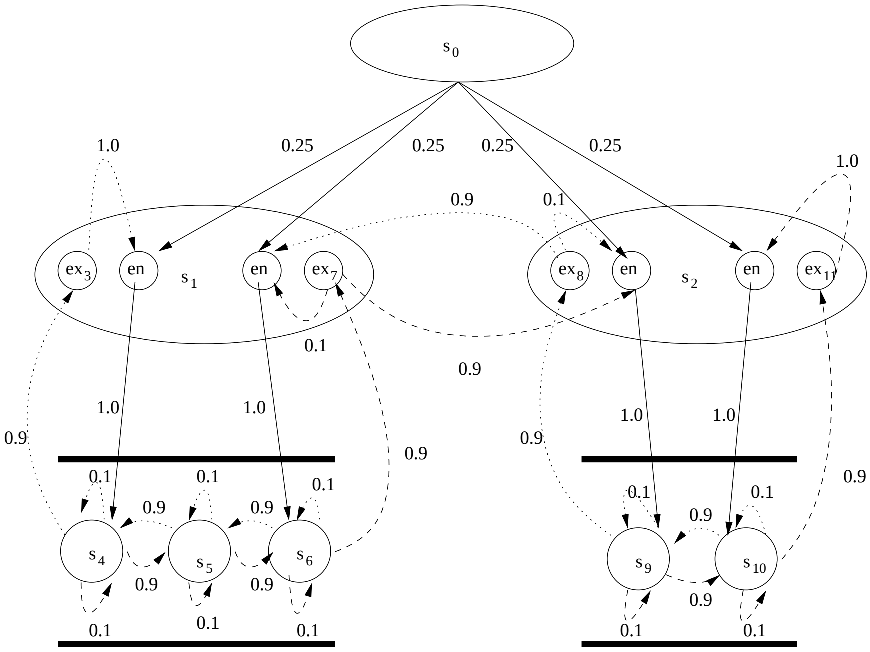


Figure 5: An example hierarchical POMDP with two primitive actions, "go-left" indicated with the dotted arrows and "go-right" indicated with the dashed arrows. This H-POMDP has two abstract states s_1 and s_2 and each abstract state has two entry and two exit states. Only the production states s_4, s_5, s_6, s_9, and s_{10} produce observations.

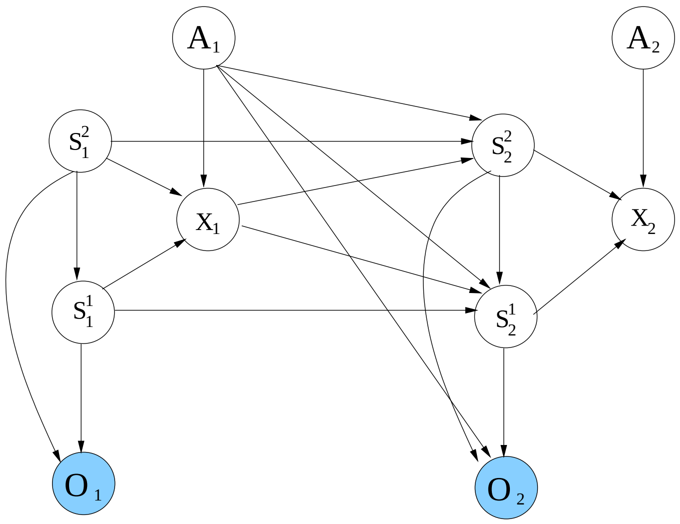


Figure 6: A 2-level H-POMDP represented as a DBN.

Figure 5 shows an example H-POMDP, which can also be transformed into a DBN as shown in Figure 6. This model differs from the model described by (Murphy & Paskin, 2001) in two basic ways: the presence of action nodes A, and the fact that exit nodes X are no longer binary. In the navigation example from Figure 5 the exit node X_t can take on 3 possible values, representing no-exit, west-exit and east-exit. If $X_t =$ no-exit, then we make a horizontal transition at the concrete level, but the abstract state is required to remain the same. If $X_t \neq$ no-exit, then we enter a new abstract state; this abstract state then makes a vertical transition into a new concrete state. The new concrete state, S_t^1, depends on the new abstract state, S_t^2, as well as the previous exit state, X_{t-1}. More precisely we can define the conditional probability distributions of each type of node in the DBN depending on node type: For the abstract nodes,

$$P(S_t^2 = s' | S_{t-1}^2 = s, X_{t-1} = x, A_{t-1} = a) = \begin{cases} \delta(s', s) & \text{if } x = \text{no-exit} \\ T^{root}(s'_x | s_x, a) & \text{otherwise} \end{cases},$$

where $T^{root}(s'_x|s_x, a)$ in the state representation of the H-POMDP model defines the transition probability from abstract state s and exit state x to abstract state s' and entry state x, where x defines the type of entry or exit state (east, west). S is the parent of s and s' in the state transition model.

For the concrete nodes,

$$P(S_t^1 = s' | S_{t-1}^1 = s, S_t^2 = S, X_{t-1} = x, A_{t-1} = a) = \begin{cases} T^S(s'|s, a) & \text{if } x = \text{no-exit} \\ V(s'|S_x) & \text{otherwise} \end{cases},$$

where $V(s'|S_x)$ defines the probability of a vertical transition from abstract state S and entry state of type x to concrete state s'.

For the exit nodes,

$$P(X_t = x | S_t^1 = s, S_t^2 = S, A_t = a) = T^S(S_x|s, a),$$

where $T^S(S_x|s,a)$ is the transition probability from production state s under abstract state S to exit from state S of type x.

For the sensor nodes,

$$P(O_t = z|S_t^1 = s, S_t^2 = S, A_{t-1} = a) = O^S(z|s,a),$$

where $O^S(z|s,a)$ is the probability of perceiving observation z at the sth child node of state S after action a.

For a given observation sequence, the final step in the transformation is to assert that the sequence has finished. Since we do not know which exit state will be used, just that $X_T =$ no-exit, we assign "soft" or "virtual" evidence to the X_T node: the local evidence becomes $(0, 0.5, 0.5)$, which encodes the fact that we can exit by any direction with equal likelihood, but the no-exit condition is impossible.

Every H-POMDP can be converted to a flat POMDP. The states of the flat POMDP are the production states of the H-POMDP, and are associated with a global transition matrix that is calculated from the vertical and horizontal transition matrices of the H-POMDP. To construct this global transition matrix of the equivalent flat POMDP, for every pair of states (s_i, s_j), we need to sum up the probabilities of all the paths that will transition the system from s_i to s_j under some action a. For example, in Figure 5 the transition from s_9 back to itself under the go-left action is not simply 0.1, but rather the summation of two paths. One path is s_9, s_9 which gives probability of 0.1, and another path is s_9, s_2, s_9 which gives probability $0.9 \times 0.1 \times 1.0 = 0.09$. Therefore the total probability is 0.19. The equivalent flat POMDP is shown here in Figure 7.

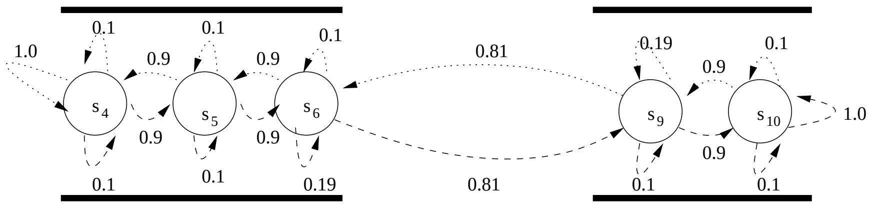


Figure 7: This is the equivalent flat POMDP of the hierarchical POMDP shown in Figure 5. Converting an H-POMDP to flat model is not advantageous, since we lose the abstract states and the vertical transitions which we can exploit later in learning and planning.

Converting an H-POMDP to a flat model, however, is not advantageous since we lose the abstract states, abstract horizontal transitions, and the vertical transitions, which we can exploit in learning and planning. For example, in model learning the fact that we have abstract states allows us to initially train submodels independently, and as a result start training of the whole model from a good initial model. This results in better trained models since the hierarchical Baum-Welch algorithm is an expectation maximization algorithm and converges to local log-likelihood maxima depending on the initial model. In planning, we can use the abstract states to construct abstract belief state representations and also to construct macro-actions that execute under the abstract states. As a result we can construct faster and more successful plans. The vertical vectors help to compute hierarchical plans in that they give us the initial relationship between abstract states and their children.

In addition to the lack of the hierarchical structure, inference in the flat POMDP model will produce different results than in the hierarchical model. The reason is that the probability that an observation sequence starts from some abstract state S and finishes at the exit state of some child state s_x, will be zero unless the child state is actually able to produce part of the observation sequence. This concept is illustrated in Figure 8. This property of the HHMM/H-POMDP models, that the next abstract state cannot be entered unless the process under the abstract state finishes, allows us to capture relationships between states that are far apart. Note however, that it is possible to force inference in flat models to give the same results as in the hierarchical by enforcing the fact that observation sequences must terminate at particular states (namely the states that have non-zero transitions to exit-states). Nonetheless, during training, after the first training epoch, inference will not give the same results due to the fact that flat models do not represent or learn parameters for abstract horizontal transition, transitions to exit-states and vertical transitions.

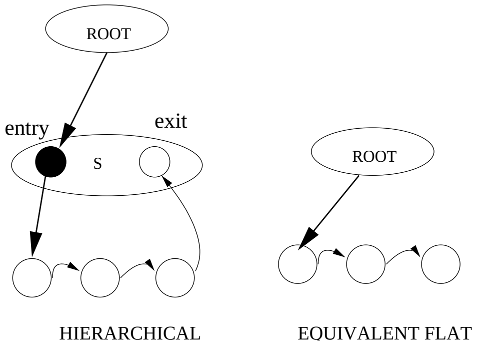


Figure 8: For an observation sequence of length 2 the hierarchical likelihood will be zero, since we need at least three observations to exit the abstract state S. However, the likelihood in the corresponding flat model will not be zero. This is one advantage of retaining the hierarchical structure of the model.

4. Learning hierarchical POMDP models

In this section we describe in detail various algorithms for learning H-POMDPs. In Section 4.1 we describe two exact learning approaches. The first method uses a non-factored state representation, while the second uses a factored state representation. In Section 4.2 we describe two heuristic learning approaches. In Section 4.3 we describe an approach for learning the structure at the abstract level.

4.1 Exact learning algorithms

A hierarchical Baum-Welch algorithm (see Figure 9) for learning the parameters of a hierarchical POMDP can be defined by extending the hierarchical Baum-Welch algorithm for HHMMs. The Baum-Welch algorithm is an expectation maximization algorithm that maximizes the log likelihood of the model parameters (T,O, and V) given observation sequences Z and actions sequences A.

Procedure H-POMDP-Baum-Welch

- Input
  - Sequences of observations $Z_1, ..., Z_k$ and actions $A_1, ..., A_k$.
    - Z_i: a sequence of observations $z_1, ...z_T$.
    - A_i: a sequence of actions $a_0, ..., a_T$.
  - Model parameters $\lambda = T, O, V$.
    - T: horizontal transition matrices.
    - O: observation models of production states.
    - V: vertical transition vectors from entry states to children.
- Output
  - A new set of λ' parameters that locally maximizes the likelihood of the observed data
$$P(Z_1, ..., Z_k | A_1, ..., A_k, \lambda')$$
- Procedure
  - The algorithm works in an iterative fashion where at each iteration it chooses new parameters λ_t that maximize the expectation of the likelihood of the observation sequences with respect to the hidden states
$$\lambda_t = \text{argmax}_\lambda \sum_{S_1,...,S_k} P(S_1, ...S_k | Z_1, ..., Z_k, A_1, ..., A_k, \lambda_{t-1}) P(Z | A, S_1, ..., S_k \lambda),$$
    where S_i is a hidden state path for observation sequence Z_i. This guarantees that $P(Z_1, ..., Z_k | A_1, ...A_k, \lambda_t) \geq P(Z_1, ..., Z_k | A_1, ..., A_k, \lambda_{t-1})$.

Figure 9: The figure describes the inputs, outputs and procedure of the Baum-Welch algorithm for H-POMDPs. The algorithm learns the model parameters, that maximize the likelihood of the observation sequences given the actions sequences. The algorithm works in an iterative hill-climbing fashion where on each iteration the likelihood is better than on the previous one. The algorithm is sensitive to the initial model parameters and is not guaranteed to reach a globally optimal solution. The implementation details are described in Section 4.1 and Appendix B.

For the expectation step, the Baum-Welch algorithm estimates two variables, the horizontal variable ξ, and the vertical variable χ. The ξ variable is an estimation of the probability of a horizontal transition from every state to every other state (both of whom have the same parent), and for every time index in the training sequence. The χ variable is an estimation of vertical transition probabilities from every parent state and child, for every time index in the training sequence.

For the maximization step, the Baum-Welch algorithm estimates the model parameters by using the frequencies of occurrence of vertical and horizontal transitions (provided by ξ and χ). For example, to estimate the horizontal transition probability between two states s and s' under an action a, the algorithm divides the expected number of times a transition occurred from s to s' under action a over the expected number of times a transition occurred from state s under action a. Next, we describe in detail the expectation and maximization steps.

A major part of the expectation step is the computation of the "forward" variable α (similar to flat hidden Markov models):

$$\begin{aligned}
&\alpha(p(s)_n, s_x, t, t+k) = \\
&P(z_t, ...z_{t+k},\ s \text{ exited from } s_x \text{ at } t+k \mid \\
&a_{t-1}, a_t...a_{t+k},\ p(s) \text{ started at } t \text{ from entry state } p(s)_n,\ \lambda),
\end{aligned}$$

which is graphically described in Figure 10.

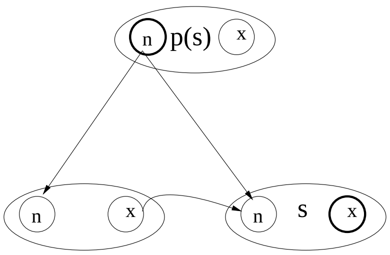


Figure 10: The α variables represents the probability of the observation sequence $z_t...z_{t+k}$ ending in state s_x at time $t+k$ (denoted as a bold circle), given that actions $a_t, ...a_{t+k}$ were taken and the parent of s, $p(s)$ was started at time t from entry state $p(s)_n$ (also denoted as bold circle).

We say that a production state finishes at time t after the production of observation z_t and an abstract state finishes when its exit state s_x is entered after observation z_t has been produced and action a_t initiated. We also say that a production state s started at time t, if at time t it produced observation z_t. An abstract state s starts at time t, if at time t one of its children produced observation z_t but observation z_{t-1} was generated before s was activated by its parent or any other horizontal transition.

Once the α variable is calculated it can also be used to estimate the probability of an observation sequence $Z = z_1, ...z_T$ given an action sequence $A = a_0...a_T$ and the model parameters λ:

$$P(Z|A, \lambda) = \sum_{i \in C_p^s \cup C_x^s} \alpha(s, i, 1, T),\ \textit{where } s = \textit{root entry}.$$

The next important variable for the expectation step is the backward variable β (similar to HMMs):

$$\beta(p(s)_x, s_n, t, t+k) =$$
$$P(z_t, ...z_{t+k}|a_t, ...a_{t+k},\ s_n \text{ started at } t,\ p(s) \text{ generated}$$
$$z_t...z_{t+k} \text{ and exited from } p(s)_x \text{ at } t+k,\ \lambda),$$

which is graphically described in Figure 11.

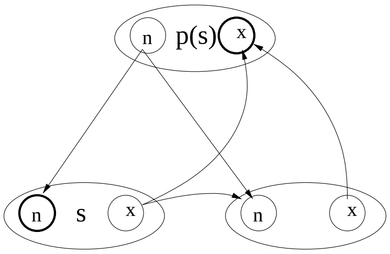


Figure 11: The β variable denotes the probability that a state s was entered at time t from entry state s_n and that the observations $z_t...z_{t+k}$ were produced by the parent of s, which is $p(s)$, and actions $a_t, ...a_{t+k}$ were taken, and $p(s)$ terminated at time $t+k$ from exit state $p(s)_x$.

The next variable is ξ, which is one of the results of the expectation step:

$$\xi(t, s_x, s'_n) = P(s \text{ finished at time } t \text{ from } s_x$$
$$s' \text{ started at time } t+1 \text{ from } s'_n \mid a_0, ...a_T, z_1, ...z_T,\ \lambda)$$

and is graphically described in Figure 12.

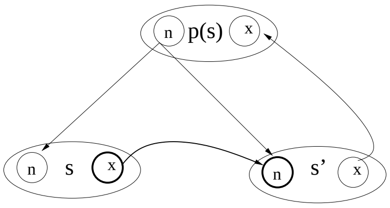


Figure 12: The ξ variable denotes the probability of making a horizontal transition from exit state s_x to entry state s'_n at time t, given a sequence of observations and actions and the model parameters.

The second result of the expectation step is the $\chi(t, p(s)_n, s_n)$ variable:

$$\chi(t, p(s)_n, s_n) = P(p(s)_n \text{ started at time } t, s_n \text{ started at time } t \mid$$
$$a_0, ...a_T, z_1, ...z_T, \lambda),$$

which is graphically described in Figure 13.

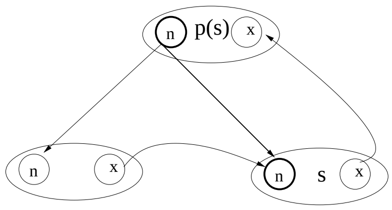


Figure 13: The χ defines the probability that entry state s_n of child s was activated by the entry state $p(s)_n$ of its parent $p(s)$ at time t given the action and observation sequence.

The exact computation of all the above variables, α, β, ξ, and χ is described in detail in Appendix B. Based on these variables we can estimate the model parameters. The vertical transition probability from the root is estimated by

$$V(s_n|p(s)_n) = \chi(1, p(s)_n, s_n),\ if\ p(s) = root,\ s_n \in C_n^{p(s)} \cup C_p^{p(s)}.$$

The vertical transition probability to any non-root child is estimated by deriving the number of times the child was vertically entered over the total number of times the child and its siblings were entered:

$$V(s_n|p(s)_n) = \frac{\sum_{t=1}^{T} \chi(t, p(s)_n, s_n)}{\sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} \sum_{t=1}^{T} \chi(t, p(s)_n, s'_n)}$$
$$if\ s \neq root.$$

The horizontal transition matrices are estimated by

$$T(s'_n|s_x, a) = \frac{\sum_{t=1|a_t=a}^{T} \xi(t, s_x, s'_n)}{\sum_{t=1|a_t=a}^{T} \sum_{s''_n \in C_p^{p(s)} \cup C_n^{p(s)}} \xi(t, s_x, s''_n)},$$

which calculates the average number of times the process went from state s to state s' over the number of times the process exited state s under action a. The observation vectors are estimated by:

$$O(z|s, a) = \frac{\sum_{t=1, z_t=z, a_{t-1}=a}^{T} \chi(t, p(s)_n, s) + \sum_{t=2, z_t=z, a_{t-1}=a}^{T} \gamma_{in}(t, s)}{\sum_{t=1}^{T} \chi(t, p(s)_n, s) + \sum_{t=2}^{T} \gamma_{in}(t, s)}$$

where

$$\gamma_{in}(t, s) = \sum_{s'_x \in C_x^{p(s)} \cup C_p^{p(s)}} \xi(t-1, s'_x, s),$$

which calculates the average number of times the process was in state s and perceived observation z over the number of times the process was in state s.

All of the above formulas for the hierarchical Baum-Welch algorithm are used to estimate the model parameters, given a single observation and action sequence. However, in order to have sufficient data to make reliable estimates of all model parameters, one has to use multiple observation and action sequences. If we had K sequences of data, then for each iteration the algorithm needs to adjust the model parameters such that the probability of all sequences is maximized. Assuming the data sequences are drawn from identical distributions and independent of each other (*iid*), then the probability of all sequences can be calculated as the production of the individual sequences:

$$P(Z_1, ..., Z_K|A_1, ...A_K, \lambda) = \prod_{k=1}^{K} P(Z_k|A_k, \lambda). \tag{1}$$

Since estimation is based on frequencies of occurrence of various events, the new estimation formulas can be constructed by adding together the individual frequencies of occurrence for each data sequence. The vertical transitions are estimated by

$$V(s_n|p(s)_n) = \sum_{k=1}^{K} \chi_k(1, p(s)_n, s_n),\ if\ p(s) = root,\ s_n \in C_n^{p(s)} \cup C_p^{p(s)} \tag{2}$$

and

$$\begin{aligned} V(s_n|p(s)_n) = & \frac{\sum_{k=1}^{K} \sum_{t=1}^{T_k} \chi_k(t, p(s)_n, s_n)}{\sum_{k=1}^{K} \sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} \sum_{t=1}^{T_k} \chi_k(t, p(s)_n, s'_n)}, \\ & if\ s \neq root, \end{aligned} \tag{3}$$

the horizontal transitions by

$$T(s'_n|s_x, a) = \frac{\sum_{k=1}^{K} \sum_{t=1|a_t=a}^{T_k} \xi_k(t, s_x, s'_n)}{\sum_{k=1}^{K} \sum_{t=1|a_t=a}^{T_k} \sum_{s''_n \in C_p^{p(s)} \cup C_n^{p(s)}} \xi_k(t, s_x, s''_n)} \tag{4}$$

and the observation models by

$$\begin{aligned} & O(z|s, a) = \\ & \frac{\sum_{k=1}^{K} \sum_{t=1, z_t=z, a_{t-1}=a}^{T_k} \chi_k(t, p(s)_n, s) + \sum_{t=2, z_t=z, a_{t-1}=a}^{T_k} \gamma_{in_k}(t, s)}{\sum_{k=1}^{K} \sum_{t=1}^{T_k} \chi_k(t, p(s)_n, s) + \sum_{t=2}^{T_k} \gamma_{in_k}(t, s)}. \end{aligned} \tag{5}$$

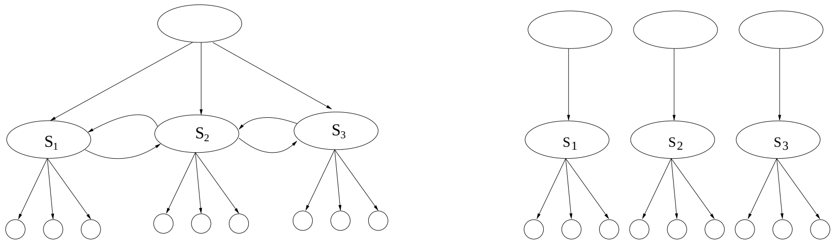
A drawback of the above algorithm is that the time complexity of the forward and backward pass is $O(NT^3)$, where N is the number of states, which may be unacceptable for long training sequences. The reason for the cubic time complexity is that for each abstract state and observation sequence of length T we need to compute T^2 α terms for all the possible subsequences of observations under the abstract state. Computation of the α term for each subsequence may depend at the most on T previous α term computations, which gives rise to cubic time complexity.

An alternative approach over the cubic time-complexity is to represent H-POMDPs as Dynamic Bayesian networks (DBN) and then use standard Bayesian network inference algorithms (Theocharous, Murphy, & Kaelbling, 2004) which take in the worst case $O(N^2T)$ time. Representing H-POMDPS as DBNS then has the advantage that we can train for long observation sequences. Nonetheless, for short observation sequences and large models the original algorithm may be faster, since it is linear in the number of states.

4.2 Approximate learning methods

To get the advantage of linear time complexity with respect to the size of the model we have explored various approximate training schemes that use training data with short observation sequences. In the H-POMDP model in order to learn relationships between states which are children of the root (and the sub model below them), training data should at least go through both of those states. This intuition can be better understood by close examination of the calculation of the probability of going from one state s_x at the abstract level to another state s'_n at time t (or $\xi(t, s_x, s'_n)$) in Appendix B.

The above realization leads to two approximate learning methods. In the first method, which we call *reuse-training*, we collect data for each abstract state separately and train submodels under each abstract state separately. If the horizontal transition model at the abstract level is deterministic, then we are done. If the horizontal transition model at the abstract level is not deterministic, then we can collect data for the entire model and use the submodels learned with the *reuse-training* method as initial submodels. In general, this method allows us to come up with better initial models. Since the hierarchical Baum-Welch algorithm is an expectation maximization algorithm, it is highly sensitive to the initial model. In addition, if we have some large environment, with similar corridors (abstract states) we can just train the model in one corridor and use the learned model parameters for all other similar corridors. Moreover, once we have trained local abstract states separately, we can use them as part of any arbitrary global model. Figure 14(a) shows an example H-POMDP model and Figure 14(b) shows how we created the different submodels to be trained.



(a) (b)

Figure 14: Figure (a) shows an example H-POMDP model for which we illustrate our approximate training methods. Figure (b) demonstrates the reuse-training method, where we train separately submodels under each abstract state child of the root (S_1, S_2, S_3)

For the second method, which we call *selective-training*, we only carry out the estimation step of the EM algorithm for selected parts of the model. More precisely, we only compute the α and β variables for the abstract states and their children which are *valid*. The validity of a state depends on the current training sequence. For each training sequence the valid states are the abstract states (and the states in their sub model) from which the data sequence was initiated and its adjacent states to which there is a non-zero transition probability. In addition to computing the α and β variables of valid states we zero all vertical transitions from the root except the vertical transition to the valid state from which the training sequence was initiated. For each valid abstract state and its submodels we estimate the new updates of the parameters. A combined update for all model parameters from all training sequences is done at the end of every epoch (after all training sequences) according to Equations 2, 3, 4, and 5. Figure 15 shows a graphical description of the selective-training approach.

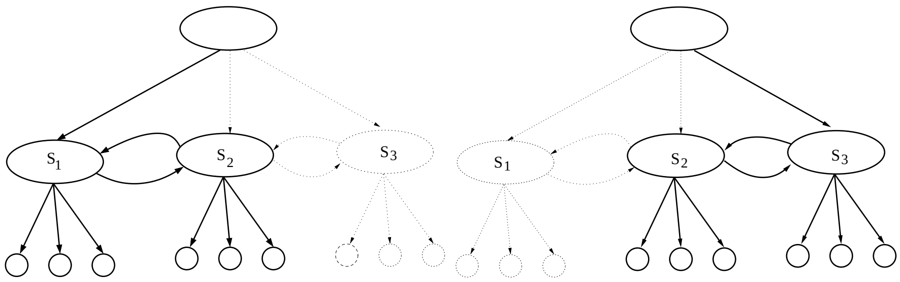


Figure 15: In the selective-training method only selected parts of the model are trained for each training sequence. Here two data sequences define two valid selections of the whole model which are shown in bold lines.

4.3 Structure learning

We also investigated the use of the H-POMDP model to infer the spatial structure of corridor environments from weak prior knowledge. To accomplish structure learning, we train models from different initial priors on the environment topology. The difference among the initial models is how ergodic they are (that is, how many connections exist between states). For example, in some initial models, every corridor is connected to all junctions and every junction is connected to multiple corridors. Our results show that for the Hierarchical POMDP model, the trained models are much closer to the true structure of the environment than equivalent uniform resolution flat POMDPs, trained using the same data. The reason is that the Hierarchical Baum-Welch algorithm does not learn the relationship between two abstract states (which are children of the root) unless the training data has a non-zero probability of exiting the second abstract state. Again, this observation can be better understood by close examination of the calculation of the probability of going from one state s'_x at the abstract level to another state s'_n at time t. This probability will be zero unless state s'_n exits. In the flat case the algorithm will still learn relationships between the children of the abstract states even if the training data does not go all the way through the second abstract state. This idea is illustrated in Figure 16.

5. Planning

In this section we present two approximate planning algorithms for POMDPs that take advantage of spatial and temporal abstractions. The unifying theme of both algorithms is their usage of the semi-Markov decision process framework for partially observable domains. In the first approach (Section 5.1), the spatial and temporal abstraction is derived from the

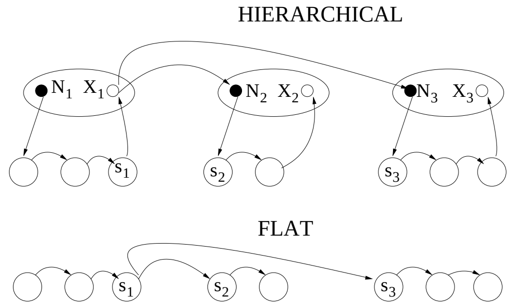


Figure 16: Assume that an observation sequence of length 5 is started from entry state N_1 and that, in the correct topology of the environment the transition probability $X_1 \rightarrow N_3$ is zero. In the hierarchical model the topology will be learned correctly since in order for the transition $X_1 \rightarrow N_3$ to be non-zero we need an observation sequence of at least length 6. In the flat model the topology will not be learned correctly since the transition probability $S_1 \rightarrow S_3$ will not be zero. The hierarchical algorithm is able to better discover the relationships between groups of states and has the potential for inferring the structure at the abstract level.

hidden state abstraction of the H-POMDP model. The planning part uses a special case of the options framework where initiation and termination of long-term actions is defined based on the hierarchical state abstraction. These hierarchical plans are heuristically mapped to belief states during execution. In the second approach (Section 5.2) the spatial and temporal abstraction is explicitly derived from agent trajectories over the belief space. The agent uses model-based reinforcement learning for computing a value function at these points. The reinforcement learning backups use the Q-learning rule for SMDPs.

5.1 Hierarchical planning in POMDPs using the H-POMDP structure

The first approach combines ideas from SMDP planning in completely observable domains (Sutton et al., 1999) with approximate POMDP planning algorithms (Koenig & Simmons, 1998). First, we design a set of macro-actions that operate within the abstract states. For example, for each abstract state we design or learn a policy (or macro-action) for exiting each one of the exit-states of the abstract state. We then compute reward and transition models for executing those macro-actions under every abstract state. Using the reward and transition models of the macro-action we can compute a plan, a mapping from entry points of the abstract states to macro-actions. During execution of the plan we map every probability distribution over the hidden production states of the model to the best macro-action.

To compute the plan we treat the production states and the entry states (of the abstract states) as the states of an SMDP. The set of actions is the set of primitive actions applicable in production states, and the set of macro-actions for each abstract state s, which transition the agent out of the abstract state s, or to some child state of the abstract state. For our corridor environment the number of macro-actions available for each abstract state is equal to the number of exit states that belong to the abstract state. The goal abstract state has an additional macro-action that can take the robot to a goal production state within the abstract state. We can design macro-actions for each corridor by either hard-coding them or by computing them using the MDP framework where we only give a goal reward. Figure 17 shows different macro-actions for corridors.

During execution we can use a hierarchical most likely state (H-MLS) strategy, where we hierarchically compute the most likely state and then execute the best macro-action assuming the world is completely observable. A macro-action terminates when the parent abstract state which initiated the macro-action is no longer the most likely abstract state. Another approach is to use a hierarchical QMDP (H-QMDP) heuristic where we hierarchically choose the best action for current belief state at the current level. This heuristic assumes that the world will become completely observable at the next time step.

But first, we need to compute an SMDP plan over the entry states of the abstract states. The first step is to evaluate the given set of macro-actions available for each abstract state s (which we define to be M^s). We need to evaluate their transition and reward model for each entry state s_n. The transition probability from entry-state $s_n \in N^s$ to an adjacent state s' of the abstract state s under some macro-action μ (s' could be a production or entry state) is computed by

$$T(s'|s_n,\mu) = \sum_{\forall s_x \in X^s} \left[\sum_{\forall i \in C_p^s \cup C_n^s} V(i|s_n) TD(s_x|i,\mu) \right] T(s'|s_x, \pi_\mu^s(s_x))$$

and is graphically explained in Figure 18. An exit from state s can occur from any exit state $s_x \in X^s$ under the primitive action $\pi_\mu^s(s_x)$, where π_μ^s defines the policy of macro action μ on all states i that are either in the C_p^s, C_n^s, or X^s sets. If i is a production or exit state, then $\pi_\mu^s(i)$ is a primitive action. If i is an entry state, then $\pi_\mu^s(i)$ is a macro action that belongs to M^j (where $i \in N^j$). The term $TD(s_x|i,\mu)$ is the expected discounted transition probability for exiting the parent of state i starting from i and can be calculated by solving the following set of linear equations:

$$TD(s_x|i,\mu) = T(s_x|i,\pi_\mu^s(i)) + \gamma \sum_{j \in C_p^s \cup C_n^s} T(j|i,\pi_\mu^s(i)) TD(s_x|j,\mu).$$

In a similar manner we can calculate the reward R that will be gained when an abstract action μ is executed at an entry state s_n as shown below:

$$R(s_n,\mu) = \sum_{i \in C_p^s \cup C_n^s} V(i|s_n) RD(i,\mu),$$

where the term $RD(i,\mu)$ can be calculated by solving a set of linear equations:

$$RD(i,\mu) = R(i,\pi_\mu^s(i)) + \gamma \sum_{j \in C_p^s \cup C_n^s} T(j|i,\pi_\mu^s(i)) RD(j,\mu).$$

Given any arbitrary H-POMDP and all the macro-actions M^s of every abstract state we can calculate all the reward and transition models at the highest level by starting from

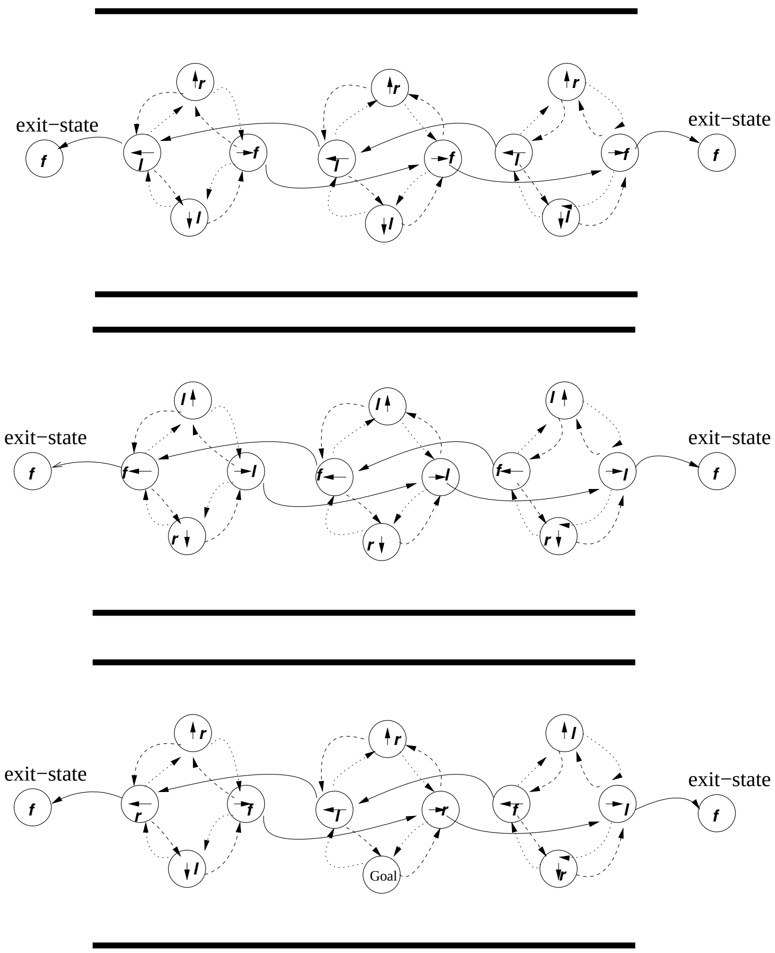


Figure 17: H-POMDP macro-actions for driving down the corridor. The circles and arrows within the circles define state and orientation. The arrows from one circle to the next define non-zero transition probabilities. The different dotting schemes indicate transition dynamics for the three actions, f which means go forward, r which means turn right and l which means turn left. The top figure is a macro-action for exiting the east side of the corridor under the "go-forward" action, the middle figure is a macro-action for exiting the west side of the corridor with the go-forward action, and the bottom figure is a macro-action for reaching a particular location within the corridor.

the lowest levels and moving toward the top. Once the transition and reward models are calculated we can construct an abstract plan among production and entry-states at the abstract level. To find a plan, we first compute the optimal values of states by solving the Bellman equations:

$$V^*(s_1) = \max_{a \in A \cup M^{s_1}} (R(s_1, a) + \gamma \sum_{s_2 \in C^{root}} T(s_2|s_1, a) V^*(s_2)),$$

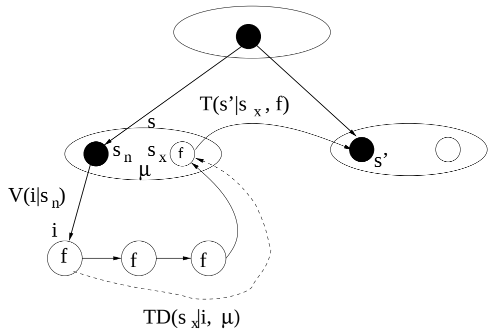


Figure 18: The dashed arrow is the discounted transition probability for reaching state S_x form state i. The solid arrows are simply the transition probabilities for the forward action.

where a is either a primitive action or a macro action. We can then associate the best action for each state (or optimal policy π^*) greedily:

$$\pi^*(s_1) = \text{argmax}_{a \in A \cup M^{s_1}} (R(s_1, a) + \gamma \sum_{s_2 \in C^{root}} T(s_2|s_1, a) V^*(s_2)).$$

Executing an abstract plan means that the robot can decide the next macro-action only when it enters an entry-state at the abstract level. In reality, we never know exactly when the robot is at an entry-state at the abstract level. Therefore, we compute the values of all states which are not at the abstract level in an iterative fashion from top to bottom as shown in the next equation:

$$\begin{aligned} V(s) = & \max_{\mu \in M^{p(s)}} [RD(s, \mu) + \\ & \gamma \sum_{s' \in C_p^{p(p(s))} \cup C_n^{p(p(s))}} \left(\sum_{\forall s_x \in X^{p(s)}} TD(s_x|s, \mu) T(s'|s_x, \pi_\mu^{p(s)}(s_x)) \right) V(s')] \end{aligned} \tag{6}$$

and explained in Figure 19. The best macro-action is then computed greedily by:

$$\begin{aligned} \pi(s) = & \text{argmax}_{\mu \in M^{p(s)}} [RD(s, \mu) + \\ & \gamma \sum_{s' \in C_p^{p(p(s))} \cup C_n^{p(p(s))}} \left(\sum_{\forall s_x \in X^{p(s)}} TD(s_x|s, \mu) T(s'|s_x, \pi_\mu^{p(s)}(s_x)) \right) V(s')]. \end{aligned} \tag{7}$$

To execute the abstract plan we keep track of a probability distribution (or belief state) at each level of abstraction. An efficient way to calculate a probability distribution at each level of abstraction is to first calculate a global transition matrix among all production states under the primitive actions, and use that transition matrix to update a global belief state among production states after every action a and observation z as shown in the next equation:

$$b'(s) = \frac{1}{P(z|a, b)} P(z|s, a) \sum_{s'=1}^{|S|} P(s|a, s') b(s'),$$

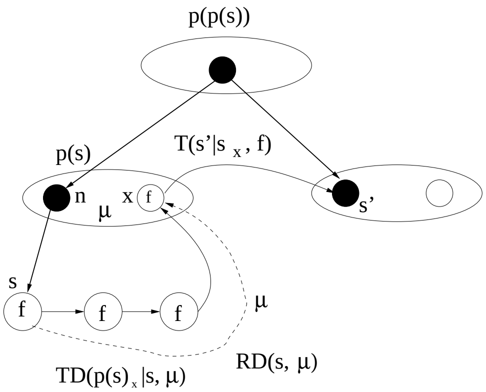


Figure 19: The value of a state s under a macro-action μ is computed by taking into consideration the total reward and discounted transition probability for exiting the parent of s, $p(s)$, from all exit states s_x, and the values of the resulting adjacent states to $p(s)$, such as s'. Equation 6 defines the value of a state as the maximum value among all macro-actions μ that belong to the set of macro-actions $M^{p(s)}$ available for the parent of s, $p(s)$.

where b' is the next belief state, $P(z|s,a)$ is the probability of perceiving observation z when action a was taken in state s. $P(s|a,s')$ is the probability of going from state s' to state s under action a and b is the previous belief state. Using the belief of production states we can calculate the belief of abstract states in an iterative fashion from leaves to root as shown below:

$$b(s) = \sum_{c(s,i) \in C_p^s \cup C_a^s} b(c(s,i)) \text{ if s is an abstract state.}$$

After every action and observation we update the belief state b, and then choose actions according to any of the below strategies:

- The first is a *hierarchical most likely state* strategy (H-MLS). Here, we start from the root and iteratively find the most likely child state at each sublevel until a production state s is reached. To avoid situations where the robot gets stuck into a loop behavior, we don't always choose the state at the peak of the belief, but rather choose randomly one of the states whose belief is closer than $1/|S|$ to the maximum belief in the current sublevel (where $|S|$ is the number of states at the same depth as the sublevel). We then choose the best macro-action to execute according to Equation 7. The macro-action terminates when one of the ancestors is no longer among the most likely state peaks. We then jump to the level where the change occurred and restart the process.

- The second method is a *hierarchical* Q_{MDP} strategy (H-Q_{MDP}). Here, we find the best macro-action μ to execute under every abstract state S by

$$\mu = \text{argmax}_{\mu \in M^S} \sum_{s \in C_p^S} b(s)[RD(s,\mu)+$$

$$\gamma \sum_{s' \in C_p^{p(S)} \cup C_n^{p(S)}} \left(\sum_{\forall s_x \in X^S} TD(s_x|s,\mu) T(s'|s_x, \pi_\mu^S(s_x) \right) V(s')]$$

The overall best macro-action μ and abstract state S is the one which is best over all abstract state and macro-action pairs. The macro-action action is executed under the abstract state S until another $< \mu, S >$ pair is maximized (according to the above equation).

To execute a macro-action under some abstract state S, we create a new separate local belief state which we initialize according to the most likely child production state of S. The macro-action is then executed using the flat MLS strategy (Koenig & Simmons, 1998).

5.2 Hierarchical planning in POMDPs using generic macro-actions

The second approach explicitly takes into consideration the outcomes of macro-actions in belief-space (Theocharous & Kaelbling, 2004). It works by using a dynamically-created finite-grid approximation to the belief space, and then using model-based reinforcement learning to compute a value function at the grid points. Our algorithm takes as input a POMDP model, a resolution r, and a set of macro-actions (described as policies or finite state automata). The output is a set of grid-points (in belief space) and their associated action-values, which via interpolation specify an action-value function over the entire belief space, and therefore a complete policy for the POMDP.

Dynamic Grid Approximation A standard method of finding approximate solutions to POMDPs is to discretize the belief space by covering it with a uniformly-spaced grid (otherwise called a regular grid) as shown in Figure 20, then solve an MDP that takes those grid points as states (Hauskrecht, 2000). Unfortunately, the number of grid points required rises exponentially in the number of dimensions in the belief space, which corresponds to the number of states in the original space.

Recent studies have shown that in many cases, an agent actually travels through a very small subpart of its entire belief space. Roy and Gordon find a low-dimensional subspace of the original belief space, then discretize that uniformly to get an MDP approximation to the original POMDP (Roy & Gordon, 2003). This is an effective strategy, but it might be that the final uniform discretization is unnecessarily fine.

In our work, we allocate grid points from a uniformly-spaced grid dynamically by simulating trajectories of the agent through the belief space. At each belief state experienced, we find the grid point that is closest to that belief state and add it to the set of grid points that we explicitly consider. In this way, we develop a set of grid points that is typically a very small subset of the entire possible grid, which is adapted to the parts of the belief space typically inhabited by the agent.

In particular, given a grid resolution r and a belief state b we can compute the coordinates (grid points g_i) of the belief simplex that contains b using an efficient method called

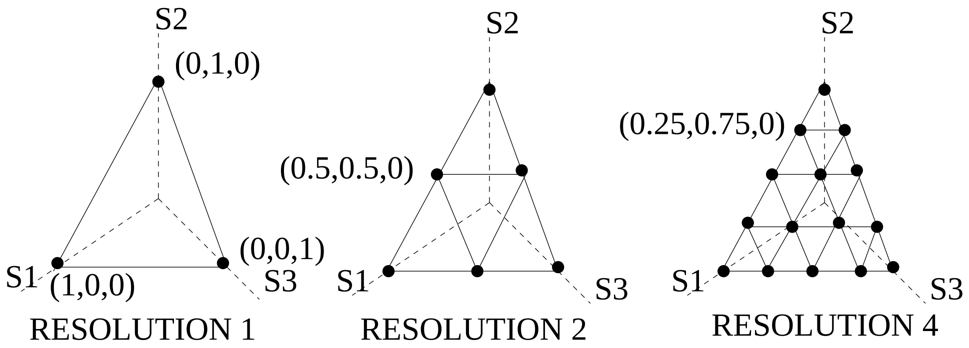


Figure 20: The figure depicts various regular discretizations of a 3 dimensional belief simplex. The belief-space is the surface of the triangle, while grid points are the intersection of the lines drawn within the triangles. Using resolution of powers of 2 allows finer discretizations to include the points of coarser discretizations.

Freudenthal triangulation (Lovejoy, 1991). In addition to the vertices of a sub-simplex, Freundenthal triangulation also produces *barycentric* coordinates λ_i, with respect to g_i, which enable effective interpolation for the value of the belief state b from the values of the grid points g_i (Hauskrecht, 2000). Using the barycentric coordinates we can also decide which is the closest grid-point to be added in the state space.

Algorithm Our algorithm works by building a grid-based approximation of the belief space while executing a policy made up of macro actions. The policy is determined by "solving" the finite MDP over the grid points. Computing a policy over grid points equally spaced in the belief simplex, otherwise called regular discretization, is computationally intractable since the number of grid-points grows exponentially with the resolution (Lovejoy, 1991). Nonetheless, the value of a belief point in a regular discretization can be interpolated efficiently from the values of the neighboring grid-points (Lovejoy, 1991). On the other hand, in variable resolution non-regular grids, interpolation can be computationally expensive (Hauskrecht, 2000). A better approach is variable resolution with regular discretization which takes advantage of fast interpolation and increases resolution only in the necessary areas (Zhou & Hansen, 2001). Our approach falls in this last category with the addition of macro-actions, which exhibit various advantages over approaches using primitive actions only. Specifically, we use a reinforcement-learning algorithm (rather than dynamic programming) to compute a value function over the MDP states. It works by generating trajectories through the belief space according to the current policy, with some added exploration. Reinforcement learning using a model, otherwise called real time dynamic programming (RTDP), is not only better suited for huge spaces but in our case is also convenient in estimating the necessary models of our macro-actions over the experienced grid points.

Figure 21 gives a graphical explanation of the algorithm. Below, we sketch the entire algorithm in detail:

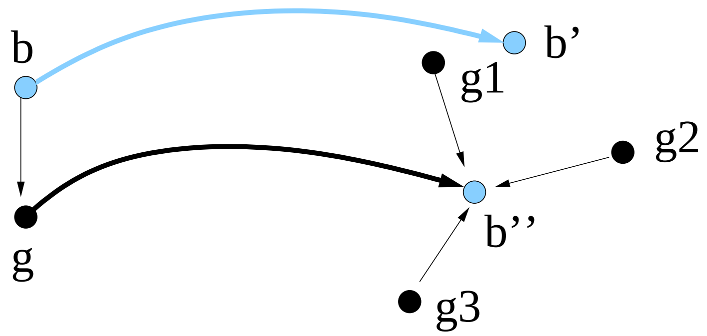


Figure 21: The agent finds itself at a belief state b. It maps b to the grid point g, which has the largest barycentric coordinate among the sub-simplex coordinates that contain b. In case the nearest grid-point g is missing, it is interpolated from coarser resolutions and added to the representation. If the resolution is 1, the value of g is initialized to zero. Now, it needs to do a value backup for that grid point. It chooses a macro action and executes it starting from the chosen grid-point, using the primitive actions and observations that it executes along the way to update its belief state. It needs to get a value estimate for the resulting belief state b''. It does so by using the barycentric coordinates from the grid to interpolate a value from nearby grid points $g1$, $g2$, and $g3$. In case the nearest grid-point g_i is missing, it is interpolated from coarser resolutions. The agent executes the macro-action from the same grid point g multiple times so that it can approximate the probability distribution over the resulting belief-states b''. Finally, it can update the estimated value of the grid point g and execute the macro-action chosen from the true belief state b. The process repeats from the next true belief state b'.

1. Assume a current true state s. This is the physical true location of the agent, and it should have support in the current belief state b (that is $b(s) \neq 0$). It is chosen randomely from the starting belief state.

2. Discretize the current belief state $b \rightarrow g_i$, where g_i is the closest grid-point (with the maximum barycentric coordinate) in a regular discretization of the belief space. If g_i is missing add it to the table. If the resolution is 1 initialize its value to zero otherwise interpolate its initial value from coarser resolutions.

3. Choose a random action ϵ% of the time. The rest of the time choose the best macro-action μ by interpolating over the Q values of the vertices of the sub-simplex that contains b: $\mu = \text{argmax}_{\mu \in \mathcal{M}} \sum_{i=1}^{|S|+1} \lambda_i Q(g_i, \mu)$.

4. Estimate $E\left[R(g_i, \mu) + \gamma^t V(b')\right]$ by sampling:

   (a) Sample a state s from the current grid-belief state g_i (which like all belief states represents a probability distribution over world states).

      i. Set $t = 0$

ii. Choose the appropriate primitive action a according to macro-action μ.
iii. Sample the next state s' from the transition model $T(s, a, \cdot)$.
iv. Sample an observation z from observation model $O(a, s', \cdot)$.
v. Store the reward $R(g_i, \mu) := R(g_i, \mu)^1 + \gamma^t R(s, a)$.
vi. Update the belief state: $b'(j) := \frac{1}{\alpha} O(a, j, z) \sum_{i \in S} T(i, a, j)$, for all states j, where α is a normalizing factor.
vii. Set $t = t + 1$, $b = b'$, $s = s'$ and repeat from step 4(a)ii until μ terminates.

(b) Compute the value of the resulting belief state b' by interpolating over the vertices in the resulting belief sub-simplex: $V(b') = \sum_i^{|S|+1} \lambda_i V(g_i)$. If the closest grid-point (with the maximum barycentric coordinate) is missing, interpolate it from coarser resolutions.

(c) Repeat steps 4a and 4b multiple times, and average the estimate $\left[R(g_i, \mu) + \gamma^t V(b')\right]$.

5. Update the state action value: $Q(g_i, \mu) = (1 - \beta) Q(g_i, \mu) + \beta E\left[R(g_i, \mu) + \gamma^t V(b')\right]$.

6. Update the state value: $V(g_i) = \text{argmax}_{\mu \in \mathcal{M}} Q(g_i, \mu)$.

7. Execute the macro-action μ starting from belief state b until termination. During execution, generate observations by sampling the POMDP model, starting from the true state s. Set $b = b'$ and $s = s'$ and go to step 2.

8. Repeat this learning epoch multiple times starting from the same b.

6. The robot navigation domain

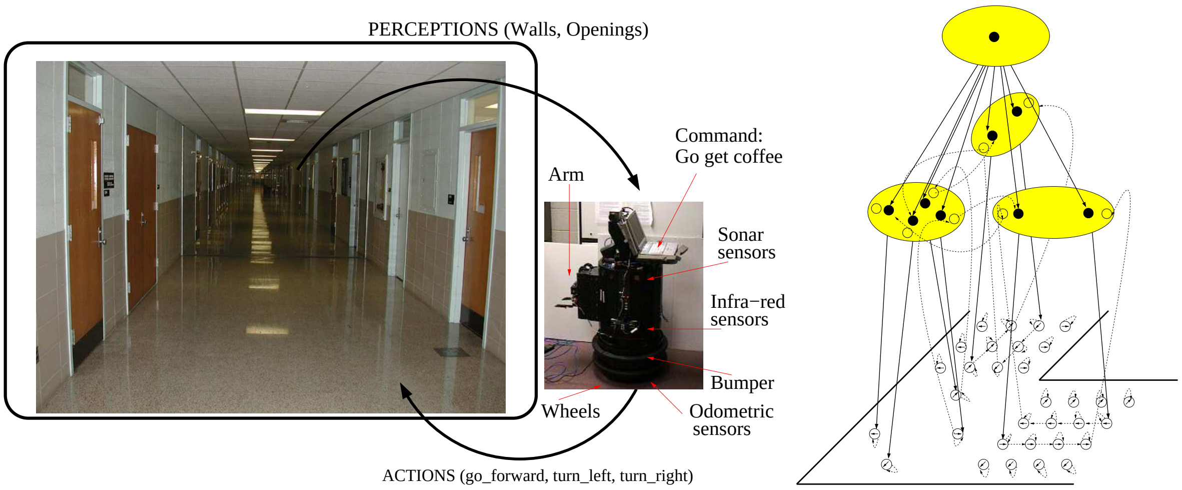
We tested our learning and planning algorithms in the domain of robot navigation. In this section we describe the task, the robot platforms, and how we represent spatial environments as POMDPs/H-POMDPs (Figure 22).

We have conducted experiments using three robot platforms. The first two are real robot platforms and the third a simulated robot (Figure 23). The simulated robot platform simulates a Nomad 200 robot. In the simulation environment the robot can move in a two dimensional world populated with convex polygonal structures (see Figure 23). Sensor modules simulate each of the sensors of the real robot platform and report sensor values according to the current simulation world.

The robotic platforms operated in two navigation environments (see Figure 24). The first is an approximate model of the 3rd floor of the Michigan State University engineering building. The second environment has the same dimensions as the 3rd floor of the MSU engineering building. The size difference between the two algorithms was used for demonstrating how our hierarchical learning and planning algorithms scale much better to larger environments than flat approaches.

Topological maps of the environment can either be learned or provided. The topology of the environment can be learned either by identifying landmark locations and their relationships (Mataric, 1990), or by learning a metric representation of the environment and then

1. For faster learning we use reward-shaping: $R(g_i, \mu) := R(g_i, \mu) + \gamma^{t+1} V(s') - \gamma^t V(s)$, where $V(s)$ are the values of the underlying MDP (Ng, Harada, & Russell, 1999)



(a) (b)

Figure 22: Figure (a) shows a section from a corridor on the 3rd floor of the Engineering Building at Michigan State University (MSU) and a real robot named Pavlov (a Nomad 200). Pavlov uses its sonar sensors to perceive walls and openings on the North, West, South, and East direction. After every perception, Pavlov takes actions such as go-forward, turn-left, and turn-right to veer toward a goal location. Navigation in such environments is challenging due to noisy sensors and actuators, long corridors, and the fact that many locations in the environment generate similar sensor values. Figure (b) shows a hierarchical POMDP representation of spatial indoor environments. Abstract states (shaded ovals) represent corridors, junctions and buildings. Production states (leafs) represent robot location and orientation.

extracting the topology (Thrun, 99). In our experiments we assume that we start with a topological map which we compile into an H-POMDP representation (see Figure 25). When we compile the small topological map in Figure 24, the result is an initial H-POMDP model with 575 states. When this initial H-POMDP model is converted to a flat POMDP the number of states is 465. Compilation of the large topological map in Figure 24 results in an H-POMDP model with 1385 states. When this H-POMDP model is converted to a flat POMDP the number of states is 1285, with a much denser transition matrix.

To learn the parameters of the H-POMDP models, first we need to collect sequences of observations and actions. To collect data we run the robot in the environment and record every action taken and every observation perceived. There are three actions available, "go-forward" which takes the robot forward approximately two meters, "turn-left," which makes the robot turn left 90 degrees, and "turn-right" which makes the robot turn right 90 degrees. These actions are implemented in a behavior-based manner, where during execution of the action, the robot is also avoiding obstacles and keeps aligned with the walls. The observation consists of 4 components, the observation of a wall or opening, on the front left, back, and

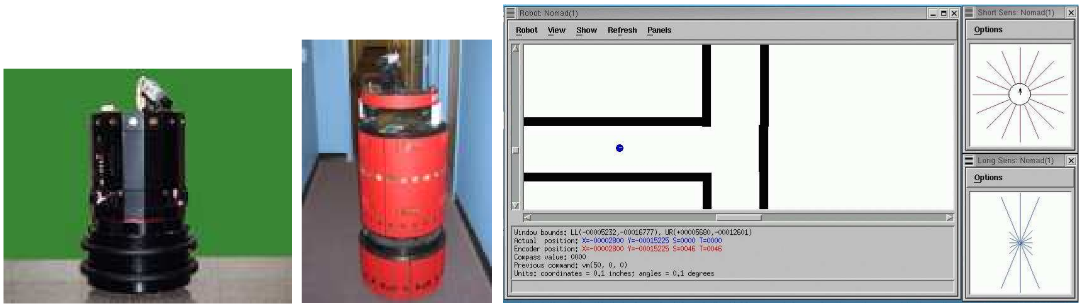
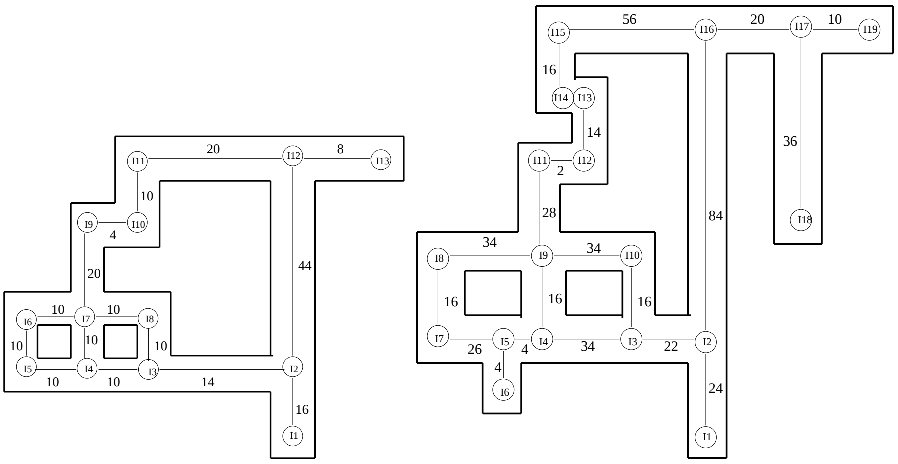


Figure 23: On the left are two pictures of real robots Pavlov, a Nomad 200 and Erik the Red, a B21r. On the right is a snapshot of the Nomad 200 simulator. The main window shows a section of the environment and the simulated robot. The two windows on the right show infrared and sonar sensor values at the current location



(a) (b)

Figure 24: The figure on the left shows an approximate model of the 3rd floor of the engineering building that is used with the Nomad 200 simulator. The figure on the right shows the real 3rd floor of the engineering building. This environment is used for experiments both, in the Nomad 200 simulator, and with the real robot Pavlov in the actual physical environment. The number next to the edges is the distance between the nodes in meters.

right side of the robot. The observations are produced by a neural-net trained with hand-labeled examples (Mahadevan, Theocharous, & Khaleeli, 1998).

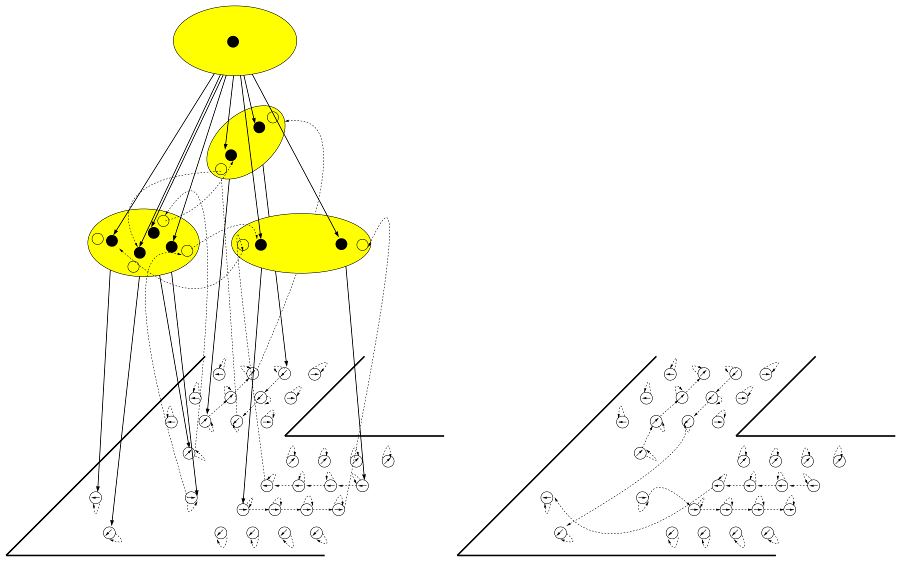


Figure 25: The figure on the left is a sample three-dimensional view of the H-POMDP model for the forward action. The solid bold arrows represent the vertical transition matrices. The dashed arrows represent the horizontal transitions. The bold circles inside the abstract states represent entry-states and the empty circles represent exit-states. In this representation all the children of the root are abstract states. Production states are the leaves of the model and represent two-meter locations in the environment. The arrows inside a production state represent the orientation of the production state. The figure on the right is the equivalent flat model of the H-POMDP.

7. Experimental results

Next we present experimental results in the navigation domain of all of our learning and planning algorithms and demonstrate their advantages over flat approaches.

7.1 Learning experiments in the state representation

To investigate the feasibility of the hierarchical Baum-Welch algorithm and the approximate training methods in learning large scale corridor environments, we performed experiments using the small simulation environment in Figure 24. First we created artificial data using a good hand-coded model (which we refer to as the original model), and for which all transitions to the next state were set to 0.9 and all self transitions set to 0.1. For the sensor models we initialized the probabilities of perceiving the correct observation to be 0.9 and all probabilities of perceiving an incorrect observation to be 0.1. Transitions at the abstract level were set deterministically to represent the correct topology of junction and corridors. For example, a forward exit from a corridor transitions to the correct entry state of a junction abstract state.

Using different sets of initial models, different lengths of data sequences, some of which were labeled with the starting state, and some of which were not labeled with the starting state, we trained both POMDP and H-POMDP models for the simulated environment. We used "short" data sequences where data was collected across two abstract states and "long" data sequences where data was collected across three abstract states. We used "uniform" initial models where all matrices in the "original" model were changed to uniform (e.g. 0.1, 0.9 was changed to 0.5, 0.5), and "good" initial models where all values of "0.9" in the original model were changed to 0.6 and all values of 0.1 were changed to 0.4. We collected 738 short training sequences, 246 long sequences, and 500 test sequences of length 40. We also collected 232 train sequences for training the submodels of the abstract states separately with the "reuse-training" method. All testing and training data was collected by sampling the good original model.

We trained uniform and good initial model using five training methods. In the first method, which we name "submodels-only" we trained the submodels of each abstract state and did not train the model at the abstract level. The second method is the "reuse-training" method where we trained the submodels separately and then retrained the whole model again. The third method is the "selective-training", the fourth is the standard EM learning algorithm for H-POMDPs, and the fifth method is the standard EM algorithm for learning POMDPs. We evaluated and compared the learned models based on robot localization, log likelihood of the training data, distance of the learned model from the original model, and computation time.

Log likelihood of the training data was measured using the log likelihood ($\sum_k^K$ log-$P(Z_k|A_k, \lambda_f)$) function defined in Equation 1, where Z_k is the observation sequence for training data set k, and A_k is action sequence k. The term λ_f refers to the parameters of a flat POMDP (T,O, and initial probability distribution π). The reason λ_f refers to the parameters of a flat POMDP is because we need to be able to compare hierarchical and flat POMDPs. Therefore, before any comparisons are made we convert the learned H-POMDPs to flat POMDPs. Since a flat POMDP can also be viewed as a Hierarchical POMDP with a single abstract state, the root, we can compute the log likelihood using Equation 1. We used this measure to show both convergence of the training algorithms, and how the learned H-POMDPs compare in terms of fit to the training data. Plots of the log likelihood for different experiments are shown in Figure 26.

Distance (or relative log-likelihood) was measured according to Equation 8 (Rabiner, 1989), which defines a method for measuring the similarity between two POMDP models λ_1, and λ_2. It is a measure of how well model λ_1 matches observations generated by model λ_2. The sign of the distance indicates which model fits the data better. A negative sign means λ_2 is better, while a positive sign means that λ_1 is better. The equation can be derived from the general definition of KL divergence (Juang & Rabiner, 1985):

$$D(\lambda_1, \lambda_2) = \frac{\log P(Z_2|A_2, \lambda_1) - \log P(Z_2|A_2, \lambda_2)}{T}, \tag{8}$$

where Z_2 and A_2 refer to a sequence of observations and actions that were produced by model λ_2, and T refers to the length of the sequence. In our experiments λ_2 refers to the good "original" model which we used to create 500 test sequences. Before any distance measurement was made, all the hierarchical models were converted to flat models. In Table

1 we report the means and standard deviations of the distances of the learned models from the original model.

Robot localization was done using the most likely state heuristic (Koenig & Simmons, 1998). To compute the most likely state we first convert the H-POMDP learned to a flat POMDP as was shown in Section 3. Then, for a test data sequence (of actions and observations), we compute the probability distribution over the states of the model after every action and observation as summarized below:

$$b'(s') = \frac{O(s', a, z) \sum_{s \in S} T(s, a, s') b(s)}{P(z|a, b)}.$$

The most likely state for each belief state b then is $\textsc{mls} = \text{argmax}_s b(s)$. Since our test data is created by sampling some good H-POMDP model we know the true state sequence. Thus, we estimate the error on robot localization for a particular test sequence by counting the number of times the true state was different from the most likely state. For multiple test sequences we average over all the errors. Table 2 shows robot localization results for the different models trained.

In passing, we should note the most likely state heuristic is a naive way of decoding the underlying state sequence in an HMM. A better approach is the Viterbi algorithm which computes the most likely state sequence rather than the most likely state at every time index (Viterbi, 1967), (Rabiner, 1989). However, the Viterbi algorithm requires the whole observation sequence before-hand which is not possible in the navigation domain.

Computation time was done by simply measuring computer time. Computation times are used to compare training times for the approximate learning algorithms with the exact learning algorithm for H-POMDPs and the flat learning algorithm for POMDPs. These results are shown in Table 3

From Tables 1 and 2, and from the graphs in Figure 26, we can draw the following conclusions:

- In almost all of the experiments described the "reuse-training" method performs the best. Even when we start with uniform initial models and do not provide the starting position the "reuse-training" method produces good results both in robot localization and in distance of the learned models from the original (Experiments 1, 3, and 5).

- From Tables 1 and 2 we can conclude that the shorter the distance to the learned models from the original model, the better the robot was able to localize.

- Almost all of the log-likelihood plots during training show that the EM learning algorithm for POMDPs fits the training data better. Nonetheless, during testing the models trained with the EM learning algorithm for POMDPs performed the worst. This observation shows that the EM learning algorithm for POMDPs over-trains the learned models (compared to the hierarchical algorithms) and does not generalize well to unseen test sequences.

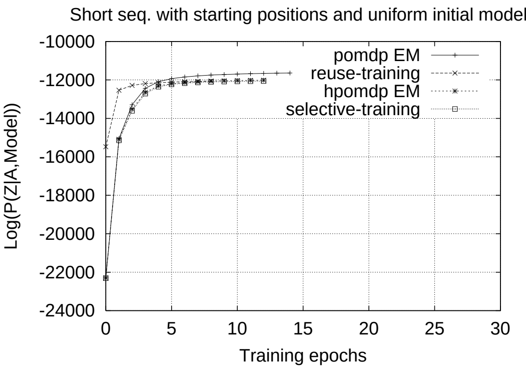
- The "selective-training" method works almost as well as the "reuse-training" method (Experiment 1 and 2). This is a pleasing result since the "selective-training" method trains much faster than the "reuse-training" method and the EM learning algorithm for H-POMDPs (see Table 3).

Initial model	Distance D(initial, original) before training μ, (σ)	Training data # of seq., # of abs. states traversed, label	log likelihood during training	Training method	Distance D(learned, original) after training μ, (σ)
original	0.0, (0.0)				
					Experiment 1
uniform	-1.3, (0.2)	232, 1,yes		submodels-only	-0.40, (0.21)
		738,2,yes	Figure (26.1)	reuse-training	**-0.29, (0.18)**
				selective-training	-0.34, (0.20)
				hierarchical	-0.35, (0.21)
				flat	-0.80, (0.35)
					Experiment 2
good	-0.95, (0.18)	232,1,yes		submodels-only	-0.38, 0.20
		738,2,yes	Figure (26.2)	reuse-training	**-0.30, (0.19)**
				selective-training	**-0.30, (0.19)**
				hierarchical	**-0.30, (0.19)**
				flat	-0.49, (0.25)
					Experiment 3
uniform	-1.3, (0.2)	246,3,no	Figure (26.3)	reuse-training	**-0.72, (0.31)**
				hierarchical	-1.93, (0.83)
				flat	-2.15, (0.99)
					Experiment 4
good	-0.95, (0.18)	246,3,no	Figure (26.4)	reuse-training	**-0.72, (0.32)**
				hierarchical	-0.82, (0.35)
				flat	-0.85, (0.33)
					Experiment 5
uniform	-1.3, (0.2)	246,3,yes	Figure (26.5)	reuse-training	**-0.48, (0.26)**
				hierarchical	-0.76, (0.30)
				flat	-1.01, (0.41)
					Experiment 6
good	-0.95, (0.18)	246,3,yes	Figure (26.6)	reuse-training	**-0.45, (0.24)**
				hierarchical	-0.5, (0.26)
				flat	-0.61, (0.21)

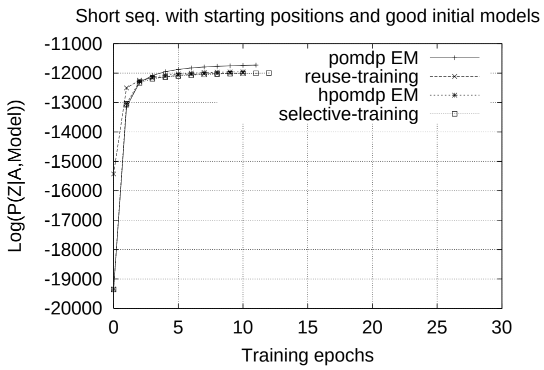
Table 1: The table shows the mean and average of the distance of the learned model from the original model. The distance was measure using 500 test sequences of length 40, which were produced by sampling the original model in the small environment in Figure 24. The bold numbers in the last column show the shortest distance for each experiment. The "reuse-training method" produces the best results. Hierarchical learning always produces better models than the flat learning. The "uniform" initial model was derived from the "original" model by changing all non-zero-entries of probability distributions to uniform. The "good" initial models where derived from the "original" model where "0.9" values were changed to 0.6 and 0.1 values were changed to 0.4.

Initial model	Localization error before training μ %, (σ)	Training data # of seq., # of abs. states traversed, label	Training method	Localization error after training μ %, (σ)
original	17.7, (14.0)			
				Experiment 1
uniform	89.2, (10.0)	232,1,yes	submodels-only	38.01, (21.47)
		738,2,yes	reuse-training	**33.04, (19.60)**
			selective-training	37.36, (21.50)
			hierarchical	37.30, (21.62)
			flat	66.00, (21.61)
				Experiment 2
good	55.38, (23.88)	232,1,yes	submodels-only	36.6, (21.75)
		738,2,yes	reuse-training	33.16, (19.13)
			selective-training	**32.84, (19.61)**
			hierarchical	33.2, (19.95)
			flat	49.17, (22.5)
				Experiment 3
uniform	89.2, (10.0)	246,3,no	reuse-training	**36.69, (20.83)**
			hierarchical	86.02, (10.59)
			flat	87.81, (8.85)
				Experiment 4
good	55.38, (23.88)	246,3,no	reuse-training	**36.97, (20.26)**
			hierarchical	43.89, (23.07)
			flat	48.62, (23.48)
				Experiment 5
uniform	89.2, (10.0)	246,3,yes	reuse-training	**34.76, (20.95)**
			hierarchical	60.45, (23.06)
			flat	69.55, (20.99)
				Experiment 6
good	55.38, (23.88)	246,3,yes	reuse-training	**35.17, (20.29)**
			hierarchical	36.60, (20.28)
			flat	45.47, (22.25)

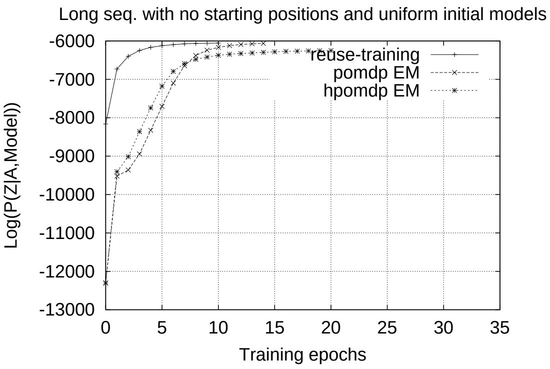
Table 2: The table shows the localization error which was computed using the 500 test sequences of length 40. The "reuse-training" method produces the best models with an exception in experiment 2 where "selective-training" is slightly better. Hierarchical learning always produces better models than flat learning. The "uniform" initial model was derived from the "original" model by changing all non-zero-entries of probability distributions to uniform. The "good" initial models where derived from the "original" model where "0.9" values were changed to 0.6 and 0.1 values were changed to 0.4



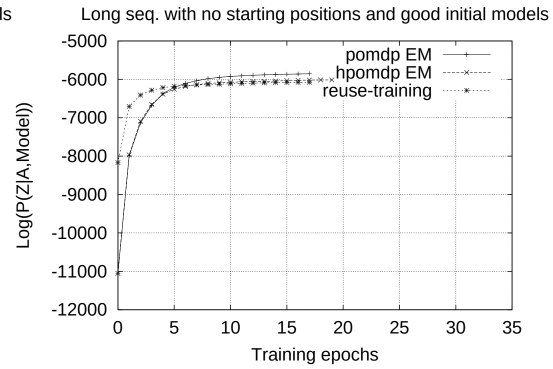
Experiment 1



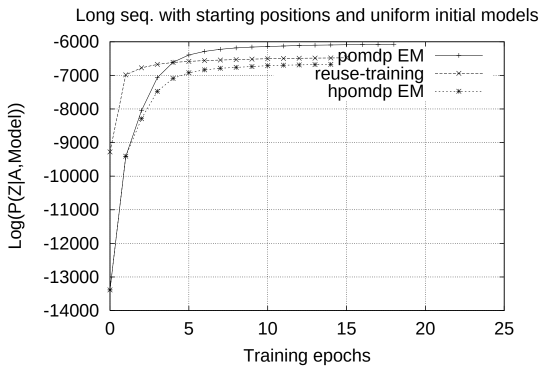
Experiment 2



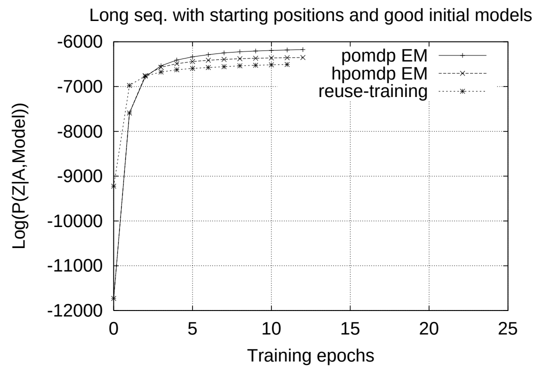
Experiment 3



Experiment 4



Experiment 5



Experiment 6

Figure 26: The graphs describe the fit to the training data during experiments 1 through 6. In almost all the plots, the flat POMDPs fit the training data better. Nonetheless, during testing the flat POMDPs performed the worst. This suggests that flat POMDPs overfit the data.

- It is obvious from Tables 1 and 2 that the learned models never reach the performance of the original model. This is due to the fact that EM type algorithms converge to local maxima and are highly dependent on the initial model. If we compare the results of the EM algorithms in experiments 1 and 2, and in experiments 3 and 4, and in experiments 5 and 6, we see that training that started from good initial models produces better learned models than training that started from uniform initial models.
- Knowing the start-state of the training data produces better learned models. This is obvious if we compare experiment 3 with experiment 5, and experiment 4 with experiment 6.
- The worst results were produced in experiment 3 where we started with uniform initial models and did not provide the initial starting position.
- As the size of the training data increases so does the performance. Despite the fact that Experiments 1 and 2 use shorter length sequences than Experiments 3, 4, 5, and 6, they produce better learned models, mainly due to the significantly larger number of training sequences provided.

We also measured the training times for "short" length sequences (sequences that were collected through two abstract states) and the different learning methods. Table 3 summarizes the results. For short training sequences the "selective-training" method is much faster than both the "flat" and "hierarchical" training methods. Even though the time complexity of the "hierarchical" training methods is $O(NT^3)$, for short training sequences they are faster than flat models whose time complexity is $O(N^2T)$. This is due to the fact that the number of states becomes a more significant factor than the length of training sequences.

Experiment	Training method	Time (sec)/epoch
1, 2	selective	449
	flat	3722
	hierarchical	1995

Table 3: The table shows the training times for short training sequences and different training methods. All training times were calculated on a 900 MHz Athlon processor running Redhat Linux 7.1

We also trained hierarchical POMDP models for the small floor plan (Figure 24), using robot data from the Nomad 200 simulator, and data with the real robot Pavlov from the actual 3rd floor in Figure 24. The purpose of these experiments was to investigate how well the learned models would perform in the robot navigation task.

For both environments we simply trained all submodels of the abstract states separately. To collect data we ran the robot separately in each abstract state. In corridors we would collect data by simply running the robot from every entry state of a corridor to the opposite exit state. In junction states, we would run the robot from every entry state to each one of the other exit states.

Learning started from a "bad" model, which was created from some known "good" original model by dividing the total probability from the self and transition to the correct state, equally among the two. The "good" original model was hand-tuned, such that it would provide reasonable navigation behavior when used. In addition, we used the "good" model for localization comparisons with the "bad" and "learned" models.

Because it is hard to compute the true location of the robot, we compared the most likely state given by the "bad" and "learned" model with the most likely state given by the "good" original model. Since the "good" model is simply based on our insight of what the parameters should be, we relaxed the MLS comparison. The error increases only when the distance between two states is longer than a single transition. That is, we don't increase the error if the most likely state given by the "good" model is the same, or adjacent to the most likely state given by the "learned" model.

Localization results are summarized in Table 4. It is obvious that after training, the learned models result in low error robot localization. Also note that unlike Table 2 in the previous section, the localization error here is quite low. This is due to the relaxed way of measuring the localization error and the fact that the initial observation models of the bad model were fairly descent (same as the "good" original model).

Training method	Training data	Test data	Localization error before training μ %, (σ)	Localization error after training μ %, (σ)
submodels-only	30 seq. from Nomad 200 simulator (Fig. 24.(a))	14 seq. from Nomad 200 simulator (Fig. 24.(a))	41.8, (21.4)	11.8, (18.0)
	43 seq. from Pavlov in real floor (Fig. 24.(b))	7 seq. from Pavlov in real floor (Fig. 24.(b))	64.9, (20.9)	2.8, (3.8)

Table 4: The table show the localization error as to the true state of the robot before and after training, in the Nomad 200 simulator and in the real environment with the actual robot Pavlov.

7.2 Learning experiments in the DBN representation

To investigate the advantages of learning H-POMDPs represented as DBNs, we performed experiments using the small simulation environment shown in Figure 24. First we created artificial data using a good hand-coded model (which we call the "original" model) for which all transitions to the correct state were set to 0.9 and the rest of the probability mass, 0.1, was divided between self and overshoot-by-one transitions. For the sensor models we ini-

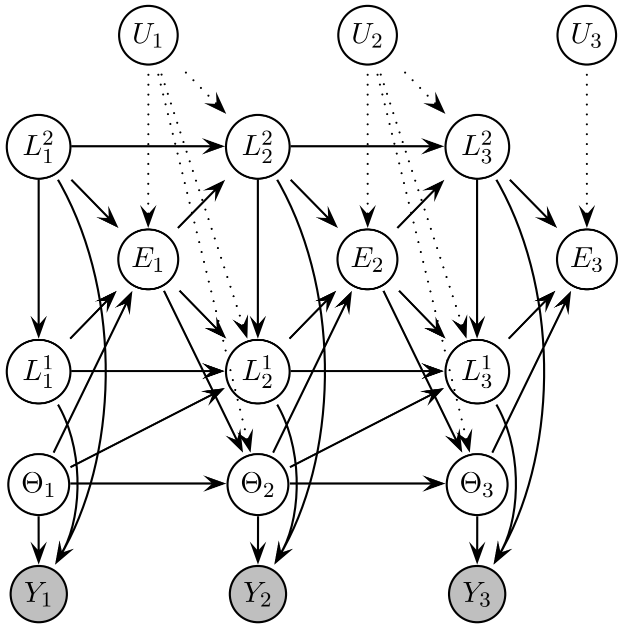


Figure 27: A 2-level factored H-POMDP represented as a DBN. We refer to this model as the "factored hierarchical DBN". The arcs from the action node, U_t, are shown dotted merely to reduce clutter. The L_t^2 nodes denote the abstract state, the L_t^1 nodes denote the concrete location, the Θ_t nodes denote the orientation, the E_t nodes denote the state of the exit variable, and Y_t denotes the state of the observation variables.

tialized the probabilities of perceiving the correct observation to be 0.9 and all probabilities of perceiving an incorrect observation to be 0.1. The control policy was a random walk.

We then used the artificial data to train four different models. We trained a flat POMDP, a hierarchical POMDP, a hierarchical POMDP with factored orientation (see Figure 27), and a hierarchical POMDP with factored orientation and for which we did parameter tying on the observation model. With parameter tying we mean that we treat certain observation probability parameters as being the same. For example, the probability of wall on the left side when the robot is facing east is the same as the probability of a wall on the right side when the robot is facing west. We used "uniform" initial models for training, where all non-zero parameters in the "original" model were changed to uniform (e.g., $\langle 0.1, 0.9\rangle$ was changed to $\langle 0.5, 0.5\rangle$). Also, we used a uniform Dirichlet prior on all non-zero probabilities, to prevent us from incorrectly setting probabilities to zero due to small training sets.

For testing, we created 500 sequences of length 30 by sampling the "original" model using a policy that performs a random walk. We evaluated the different models using log-likelihood relative to the original model as we did in Section 7.1. In addition , we compared the algorithms in terms of localization and computation time. The relative log-likelihood results are shown in Figure 28. It is clear that the hierarchical models require much less data than the flat model. Furthermore, factoring and parameter tying help even more.

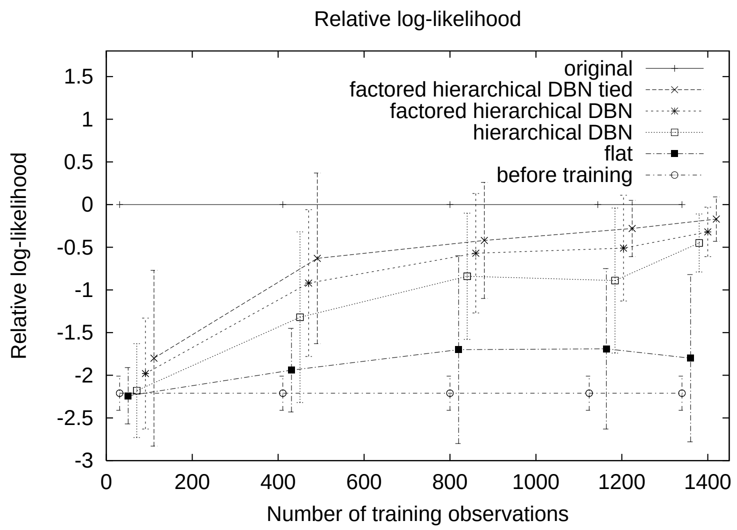


Figure 28: Relative log-likelihood for the four models averaged over 500 test sequences. The curves are as follows, from bottom to top: flat, hierarchical, factored hierarchical, factored hierarchical with tying, The bottom flat line is the model before training, the top flat line is the generating model.

To assess the ability of the models to estimate the robot's position after training (a more relevant criterion than likelihood), we computed the total probability mass assigned to the true state sequence (which is known, since we generated the data from a simulator): $\sum_{t=1}^{T} b_t(s)$, where $b_t(s) = P(X_t = s|y_{1:t}, u_{1:t})$ is the belief state at time t. If the belief state was a delta function, and put all its mass on the correct state, then $\sum_{t=1}^{T} b_t(s) = T$; this is the score that an algorithm with access to an oracle would achieve. The best score that is achievable without access to an oracle is obtained by computing the belief state using the original generating model; this score is 87%. The scores of the other algorithms are shown in Figure 29. (This is the total probability mass assigned to the true states in all the test sequences.) The rank ordering of the algorithms is the same as before: the flat model performs the worst, then hierarchical, then factored hierarchical, and the factored hierarchical with tying is the best.

To investigate the scalability of our algorithm, we also plotted the time per training epoch with respect to the length of the training sequence (see Figure 30). The results are shown in Figure 30. The $O(T^3)$ behavior of the original H-POMDP algorithm is clear; the flat and DBN algorithms are all $O(T)$, although the constant factors differ. The flat algorithm was implemented in C, and therefore runs faster than the hierarchical DBN (although not by much!); the factored hierarchical DBN is the fastest, even though it is also implemented in Matlab, since the state space is much smaller.

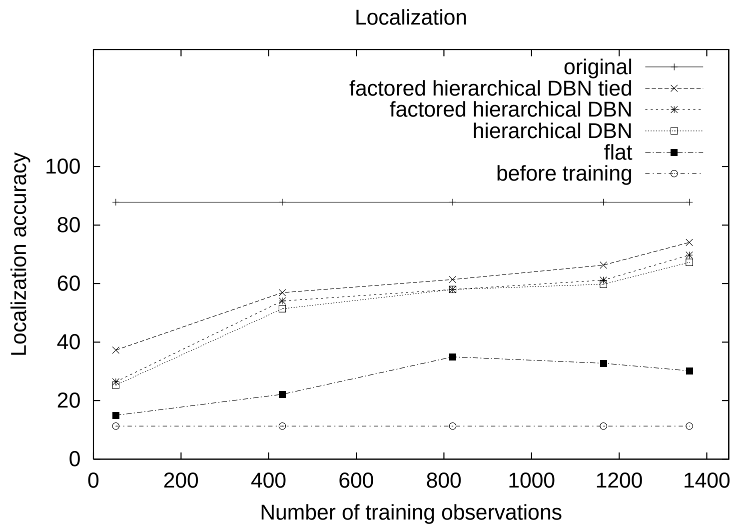


Figure 29: Percentage of time the robot estimates its location correctly (see text for precise definition), as a function of the amount of training data. The curves are as follows, from bottom to top: flat, hierarchical, factored hierarchical, factored hierarchical with tying. The bottom flat line is the model before training, the top flat line is the generating model.

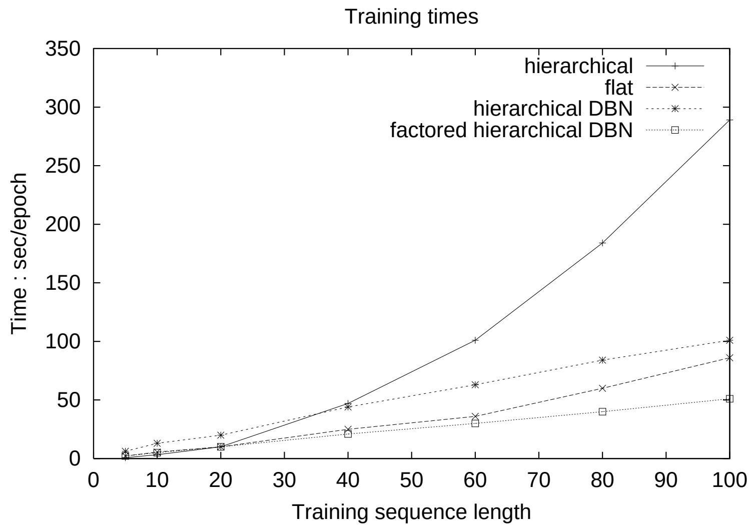


Figure 30: Running time (in elapsed user seconds) vs sequence length for the different algorithms. The curves from bottom to top are: factored hierarchical DBN (Matlab), flat (C), hierarchical DBN (Matlab), original hierarchical H-POMDP (C).

To verify the applicability of these ideas to the real world, we conduced an experiment with a B21R mobile robot. We created a topological map by hand of the 7th floor of the MIT AI lab; this has about 600 production states (representing $1m \times 1m$ grid cells), and 38 abstract states (representing corridors, junctions and open space). We manually drove the robot around this environment, and collected data using a laser-range finder. The data was classified using a neural network to indicate the presence of a wall or opening on the front, left, right and back sides of the robot. We then created several initial models, based on the true topology but with unknown (uniform) observations, and tried learning their parameters from several training sequences totaling about 600 observations (corresponding to about 3 laps around the lab). The results are qualitatively similar to the simulation results. The tied model always does significantly better, however, since it is able to learn the appearance of all 4 directions in a single pass through a corridor.

7.3 Structure learning experiments

In this section we show hat it is possible to learn the structure at the abstract level just by doing parameter estimation in the H-POMDP. We assume that the initial abstract transition matrix is almost fully interconnected (ergodic), but with the constraint that corridor states could only connect to junction states, and vice versa. The resulting parameters, when thresholded, recovers the correct topology. By contrast, attempts to learn flat HMM topology using EM have generally been considered unsuccessful, with the notable exception of (Brand, 1999), who uses a minimum entropy prior to encourage structural zeros.

7.3.1 Learning the structure of a three level hierarchy

For the simulated environment in Figure 24 we trained an H-POMDP model using 245 short training sequences (collected by sampling through two abstract states) labeled with their starting positions. We used three type of training methods, the "selective-training" method, the standard EM learning algorithm for H-POMDPs, and the standard flat EM learning method for POMDPs. We trained the model starting from different sets of initial parameters. Each set of parameters defined a different type of "semi-ergodic" model. Below we describe the different initial models:

1. A hierarchical model "model-1", where every exit point of every junction abstract state is connected to the correct corridor and entry-state, and all entry points of other corridors whose size is different than the correct corridor. Every corridor is connected to a single junction. A partial example of this type of initial model is shown in Figure 31.

2. A hierarchical model "model-2", where connections from junctions are same as "model-1". To investigate the strength of our approach in recovering the correct structure as the ergodicity at the abstract level gets larger, we made this model more ergodic. We connected every corridor abstract state to every junction abstract state.

To evaluate each model we examine the percentage of the total transition probability that is associated with the *correct* structure of the environment at the abstract level. The arrows

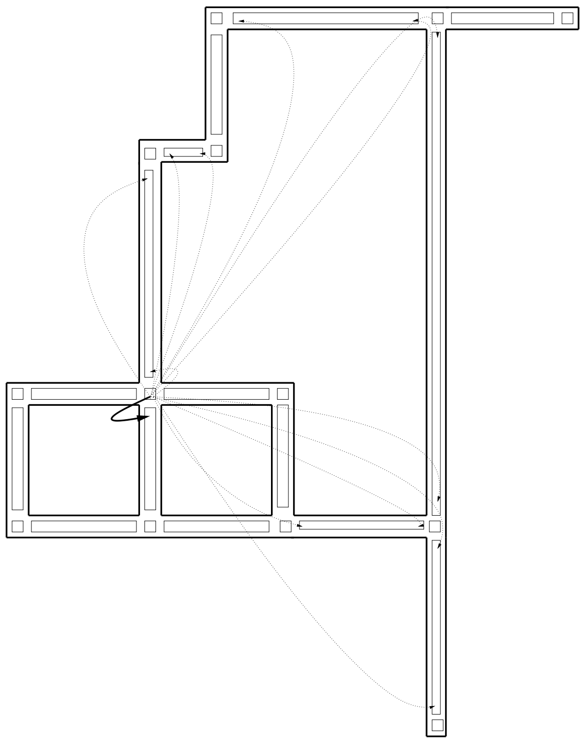


Figure 31: The bold arrow indicates the correct structure for the particular south exit point of the junction abstract state. The dotted arrows indicate possible adjacent nodes. We chose adjacent nodes to be corridors of different size from the correct corridor.

in Figure 32 define the *correct* structure. In an ergodic model there are more connections among states (at level 2) than the arrows shown in the figure.

Correct structure was measured as follows:

$$\% \text{ correct structure} = \frac{\sum_{s\in S, a\in A}\sum_{s'\in S'} T(s'|s,a)}{\sum_{s\in S, a\in A}\sum_{s''\in S''} T(s''|s,a)},$$

where S' is the set of states that belong to abstract states different from the parent of s, $p(s)$, such that s' and s belong to topologically adjacent abstract states. S'' is the set of states that belong to abstract states different from the parent of s, $p(s)$ (s and s'' do not necessarily belong to topologically adjacent abstract states). In order to be able to compare flat and hierarchical learning algorithm the transition functions $T(s'|s,a)$ and $T(s''|s,a)$ refer to the transition probabilities when the model is represented as a flat POMDP.

Overall, our evaluation criterion finds the percentage of transition probability associated with the arrows in Figure 32 over the total transition probability associated with all the arrows (all arrows starting from the same positions as the arrows shown in Figure 32). Table 5 summarizes the results of all the models, and shows that the hierarchical models improve the structure much more than the flat models. Figure 32 shows the transition probabilities of the correct structure before and after learning for initial “model-1”.

Model	Correct structure error before training	Training method	Correct structure error after training
model-1	46.91 %	selective-training	**15.47 %**
		hierarchical	15.63 %
		flat	41.79 %
model-2	96.02 %	selective-training	**59.36** %
		hierarchical	60.71 %
		flat	87.20 %

Table 5: The table shows the percentage of transition probabilities that do not belong to the correct structure before and after learning for the three level hierarchical model. For model-2 the results are not as good as model-1 mainly due to the high initial error. Nonetheless, in both cases the hierarchical approaches improve the structure considerably, while the flat approaches do not.

7.3.2 Learning the structure of a four level hierarchy

We also carried out structure learning experiments on a four level hierarchical model. The second and third level are shown in figure 33. In this set of experiments we started with two types of initial models:

1. In the first model, "4level-1", only the structure at level 2 was semi-ergodic where every state at level two was connected to every other state at level 2.
2. In the second model, "4level-2', all states at the second level where connected the same as in the first model "4level-1". At the third level (for all submodels of the abstract states at level 2) corridor states were connected to all junction states, and junction states were connected to all corridor states whose size was different from the correct corridor (similar to "model-2" in section 7.3.1).

Table 6 summarizes the result for the different initial models and different training methods. In this experiment, 17 long training sequences were used. For model "4level-1", both the EM learning algorithm for H-POMDPs and the EM learning algorithm for POMDPs learn the structure of the model equally well. The reason the EM learning algorithm for POMDPs performs this well is because we have used long data training sequence across the abstract states at level 2, and the fact that the abstract states at level 2 look sufficiently different, thus enabling the EM learning algorithm for POMDPs to discover the correct state sequence.

For model "4level-2" however, the EM algorithm for learning H-POMDPs outperforms the EM algorithm for learning POMDPs. The reason is that by adding ergodicity at level 3 the abstract states at level 2 do not look substantially different and as a result the EM algorithm for learning POMDPs is unable to discover the correct sequence of states.

In all the above structure learning experiments the hierarchical algorithms perform better or equally well as the flat. They perform significantly better when the training data sequences are short. Increasing the level of hierarchy requires longer training data to

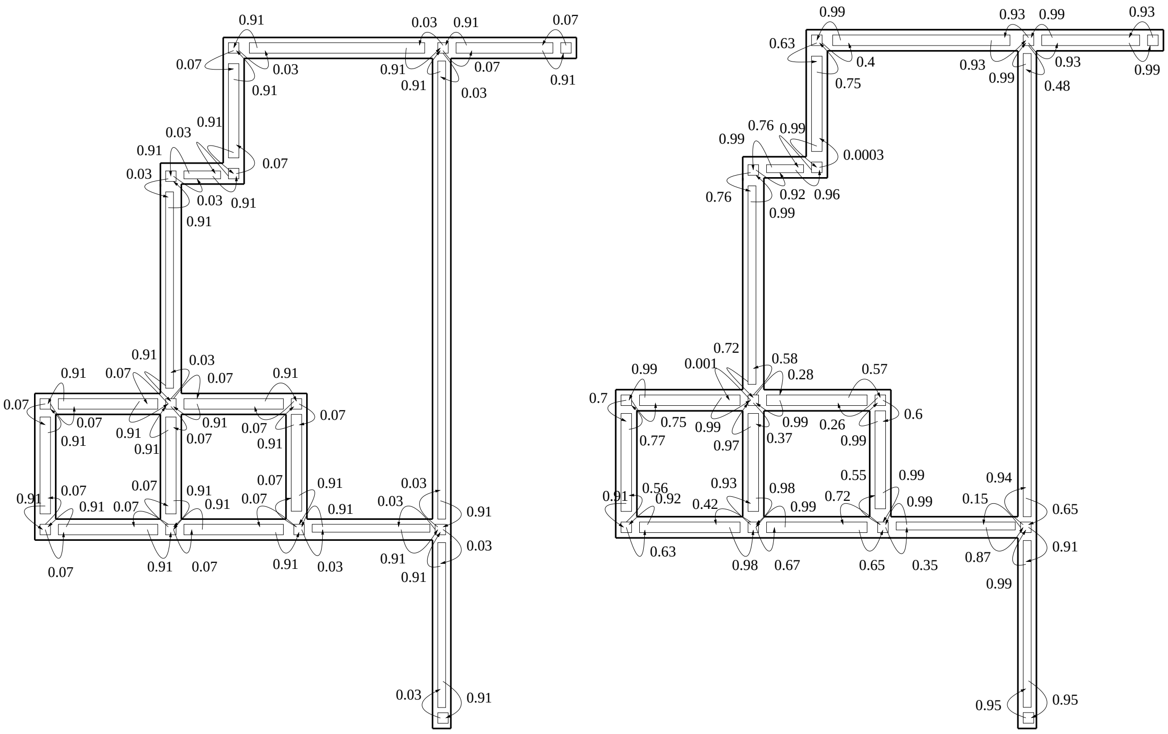


Figure 32: The figure on the left shows the transition probabilities that belong to the *correct* structure before training for "model-1". The figure on the left shows the transition probabilities that belong to the *correct* structure after training for "model-1", using the "selective-training" method. The probabilities shown are the ones corresponding to the H-POMDP model when converted to flat.

learn the relationships at the highest level. Having longer sequences enables flat algorithms to perform as well, specially when the structure uncertainty is small. As the structure uncertainty increases the hierarchical approaches outperform the flat ones despite the long data sequences.

Model	Correct structure error before training		Training method	Correct structure error after training	
	second level	third level		second level	third level
4level-1	77.27 %	4.31 %	hierarchical	13.25 %	0.22 %
			flat	12.31 %	0.20 %
4level-2	82.46 %	83.61 %	hierarchical	**32.50** %	**49.90** %
			flat	43.85 %	55.34 %

Table 6: The table shows the percentage of transition probabilities that do not belong to the correct structure before and after learning, for the four level hierarchical models.

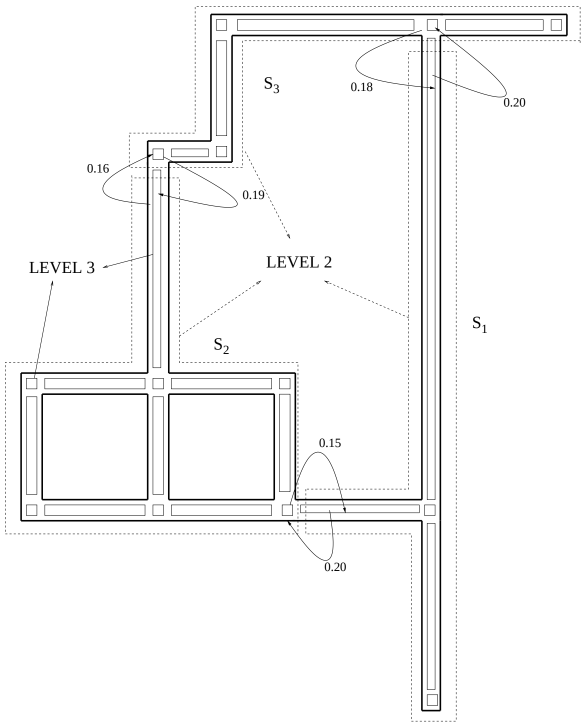


Figure 33: This is a four level representation of the environment where entire sections of the building where grouped into abstract states at a higher level. The three large dashed polygons represent the abstraction at level 2 while the smaller rectangles inside the polygons represent the spatial abstraction at level 3. The bold arrows indicate the correct structure at level 2 and the number next to the arrows indicate the transition probabilities before any learning for model “4level-2”. The probabilities shown are the ones that belong to the correct structure when the model is converted to flat.

7.4 Planning experiments using the H-POMDP structure

In the small navigation environment (Figure 24), we set up two navigation tasks. One task was to get to the four-way junction in the middle of the environment (node I7) and the other task was to get to the middle of the largest corridor (between nodes I2 and I12). The reason we set up these two navigation tasks is because they form two opposite and extreme situations. The four-way junction emits a unique observations in the environment, while the middle of the largest corridor is the most perceptually aliased location. Experiments in the small navigation environment were done using the Nomad 200 simulator. We also performed experiments in the Nomad 200 simulator using the larger environment. In these set of experiments, again we set up two navigation tasks, where one was to reach the four-way junction (node I9), and the other was to reach the middle of the largest corridor (between nodes I2 and I 16). Additionally, we performed experiments using the real robot Pavlov in the real environment. In this set of experiments we formulated 5 different goals.

For each navigation task, we created macro-actions for every abstract state. For every abstract state, we created one macro-action for each of its exit states (a macro-action that would lead the robot out of the abstract state through the particular exit state). For the abstract state that contained the goal location we created an additional macro-action that could take the robot to the goal state. We then gave a global reward, where the robot is penalized by -1 for every primitive action executed, and rewarded it by $+100$ reward for primitive actions leading to the goal state. In this way the robot would learn to choose shorter routes so as to minimize the cost. Based on the reward function we computed the reward and transition models of the macro-actions, and then computed a plan using macro-actions over the children of the root.

For the small simulated environments we compared the H-MLS algorithm with the flat most likely state strategy (F-MLS), and the H-QMDP algorithm with the flat QMDP (F-QMDP). For each navigation task we started the robot 3 times from 6 different locations and recorded the number of steps it took to reach the goal. We also recorded the number of times the robot was successful in reaching its goal, over the total number of runs. The results are summarized in Table 7. All experiments were started both from a uniform initial belief state, where the starting position was not known, and from a localized initial belief state where the starting position was known.

Envir.	Initial position	Planning method	Steps	Macro steps	Success
small sim.	unknown	H-MLS	**56.1**	15.5	100 %
		F-MLS	73.15		100 %
		H-QMDP	**47**	9.6	100 %
		F-QMDP	47		66.6%
	known	All	29	5.5	100 %

Table 7: The table shows the average number of primitive steps, and the average number of decisions at the abstract level. The results are for the small environment in the Nomad 200 simulator.

It is evident from Table 7 that the hierarchical approaches perform better than the flat. The MLS strategies produce 100% success due to the randomness in the selection of the peak of the belief state. The QMDP approaches perform better than the MLS approaches. However, the flat QMDP approach has a low success rate. Starting from known initial positions, all approaches perform the same.

For the large simulated environment we tested all the algorithms by starting the robot 3 times from 7 different locations and measured the steps it took to reach the goal. The results are summarized in Table 8. We also ran the robot in the real environment using the H-POMDP model. In this experiment we started the robot from 11 different locations and used 5 different goal destinations. The results are summarized in Table 8.

It is obvious from Table 8 that the hierarchical approaches outperform the flat when the starting belief state is uniform. Due to the low success rate in the flat F-QMDP method (starting from unknown position) we did not report the number of steps. Failures to reach the goal occur when the robot gets stuck into loop behavior. The MLS strategies

Envir.	Initial position	Planning method	Steps	Macro steps	Success
large sim.	unknown	H-MLS	**90.5**	15.8	100 %
		F-MLS	163		93 %
		H-QMDP	**74**	12.5	85 %
		F-QMDP			28 %
	known	All	46	6.7	100 %
large real	unknown	H-MLS	130	22.3	100 %
	known	H-MLS	53.8	7.2	100 %

Table 8: The table shows the average number of primitive steps, and the average number of decisions at the abstract level. The results are for the large environment in the Nomad 200 simulator, and in the real world.

produce high success rates due to the randomness in choosing the peak of the belief state. When the initial position is known, all the methods produce the same result. For the real environment we ran the most successful hierarchical approach, the H-MLS, which proved to work successfully. Comparing Table 8 with Table 7, we can conclude that the hierarchical approaches scale more gracefully than flat approaches to large environments.

We also measured the time it took for the robot to construct a plan, including the time to construct the macro-action that leads to the goal state. The results are summarized in table 9 where for the H-POMDP model it takes significantly less time to compute a plan. The time complexity of the value-iteration algorithm is $O(S^2T)$ where S is the number of states and T is the planning horizon. If we construct perfect H-POMDP trees where all leaves are at the same level, and each abstract state has the maximum number of children, then the time complexity of value iteration over the children of the root would be $O(S^{\frac{2}{d}}NT)$, where S is the total number of production states, d is the depth of the tree and N is the maximum number of entry states for an abstract state. Usually we don't construct perfect trees, and therefore, time complexity lies between $O(S^{\frac{2}{d}}NT)$ and $O(S^2T)$.

Environment	Goal location	Model	Planning time (sec)
small	I7	H-POMDP	**2.11**
		POMDP	15.34
	I2-I12	H-POMDP	**5.7**
		POMDP	14.67
large	I9	H-POMDP	**5.05**
		POMDP	88.02
	I2-I16	H-POMDP	**26.12**
		POMDP	83.57

Table 9: The table shows the planning times for the different environments and models. The planning times were computed on a 900 MHz Athlon processor

It is apparent from Table 9 that hierarchical plans are constructed faster than flat plans due to the small number of states at the abstract level, and due to the fact that we can reuse precomputed macro-actions. For each navigation task only one new macro-action needs to be computed, the one for the abstract state which contains the goal location.

We also performed experiments with learned H-POMDP models to investigate the effect of learning H-POMDP models on the navigation system as a whole. The experiments were done in the Nomad 200 simulator using the smaller navigation environment. The overall training procedure was done as follows:

1. First we started with an initial H-POMDP model. In this H-POMDP model we set all probabilities of observing the correct observation to be 0.6. In all corridor states we set the overshoot transition to 0.6, the self transitions to 0.2, and transitions to the correct states to 0.2. In all junction states, we set the self transitions to 0.5 and transitions to the correct state to 0.5. We trained this model by simply training all the submodels of the abstract states separately ("submodels-only" strategy). Training data was collected by running the robot separately in each abstract state. In corridors, we would collect data by simply running the robot from every entry state of a corridor to the opposite exit state. We collected 30 training sequences in this manner, two for each corridor.

2. Given the trained model we were able to run different navigation tasks. For each navigation task we collected data for the "selective-training" strategy. A data sequence was recorded if the robot entered an abstract state and was 99% sure that it was at the particular entry state. Recording stopped when two abstract states were traversed. We collected 38 training sequences in this manner. We then retrained the model learned in the first stage, using the new training data, with the "selective-training" method.

To evaluate the different stages of learning we started the robot from 12 different locations and provided the initial starting position. The goal was to get to the four-way junction in the middle of the environment (node I7). Using our hierarchical planning and execution algorithm H-MLS, we measured the average number of steps, the success rate of reaching the goal, and the mean of the average entropy per run. We measured the average entropy of all runs to be $\bar{H} = \frac{1}{T}\sum_t H_t$, where T is the total number of steps for all runs. H_t is the normalized entropy of the probability distribution over all production states, which is computed as:

$$\begin{aligned}\text{normalized entropy} &= \frac{\text{entropy}}{\text{maximum entropy}} = \frac{-\sum_{s=1}^{S} P(s)\log P(s)}{-\sum_{s=1}^{S} 1/|S|\log(1/|S|)} \\ &= \frac{-\sum_{s=1}^{S} P(s)\log P(s)}{-\log(1/|S|)} = \frac{-\sum_{s=1}^{S} P(s)\log P(s)}{\log(|S|)}.\end{aligned}$$

A run was considered a failure if the robot traveled more than 70 steps and still was unable to reach the goal. We measured the average number of steps in reaching the goal over successful trials, but we measured the average entropy over both successful and unsuccessful trials. The results are summarized in Table 10.

Training stage	Average number of steps	Average entropy	Success
before training	51	0.29	16.6 %
submodels-only	27	0.14	100 %
selective-training	27	0.10	100 %

Table 10: The table shows the average number of steps to reach the goal, and average entropy per run after every training stage. Simply training the submodels provides an H-POMDP model that can be used successfully for robot navigation tasks. Retraining the already learned model with additional data from the navigation tasks improves robot localization.

7.5 Planning experiments for POMDPs with generic macro-actions

We tested our second algorithm by applying it to the problem of robot navigation domain in the large environment in Figure 24. A macro-action is implemented as a behavior (could be a POMDP policy) that takes observations as inputs and outputs actions. In our experiments we have a macro-action for going down the corridor until the end.

We ran the algorithm starting with resolution 1. When the average number of training steps stabilized we increased the resolution by multiplying it by 2. The maximum resolution we considered was 4. Each training episode started from the uniform initial belief state and was terminated when the four-way junction was reached or when more than 200 steps were taken. We ran the algorithm with and without the macro-action go-down-the-corridor. We compared the results with the QMDP heuristic which first solves the underlying MDP and then given any belief state, chooses the action that maximizes the dot product of the belief and Q values of state action pairs: $QMDP_a = \text{argmax}_a \sum_{s=1}^{|S|} b(s)Q(s,a)$.

With Reward Shaping The learning results in Figure 34 demonstrate that learning with macro-actions requires fewer training steps, which means the agent is getting to the goal faster. In addition, training with-macro-actions produces better performance as we increase the resolution. Training with primitive actions only does not scale well as we increase the resolution, because the number of states increases dramatically.

In general, the number of grid points used with or without macro-actions is significantly smaller than the total number of points allowed for regular discretization. For example, for a regular discretization the number of grid points can be computed by the formula given in (Lovejoy, 1991), $\frac{(r+|S|-1)!}{r!(|S|-1)!}$, which is 5.4^{10} for $r = 4$ and $|S| = 1068$. Our algorithm with macro actions uses only about 1000 and with primitive actions only about 3500 grid points.

We tested the policies that resulted from each algorithm by starting from a uniform initial belief state and a uniformly randomly chosen world state and simulating the greedy policy derived by interpolating the grid value function. We tested our plans over 200 different sampling sequences and report the results in Figure 35. A run was considered a success if the robot was able to reach the goal in fewer than 200 steps.

From Figure 35 we can conclude that the QMDP approach can never be 100% successful. It is also evident from Figure 35 that as we increase the resolution, the macro-action algorithm performs better while the primitive-action algorithm performs worse, mainly due to the fact that it adds more grid-points. It is also obvious from the above figures that with

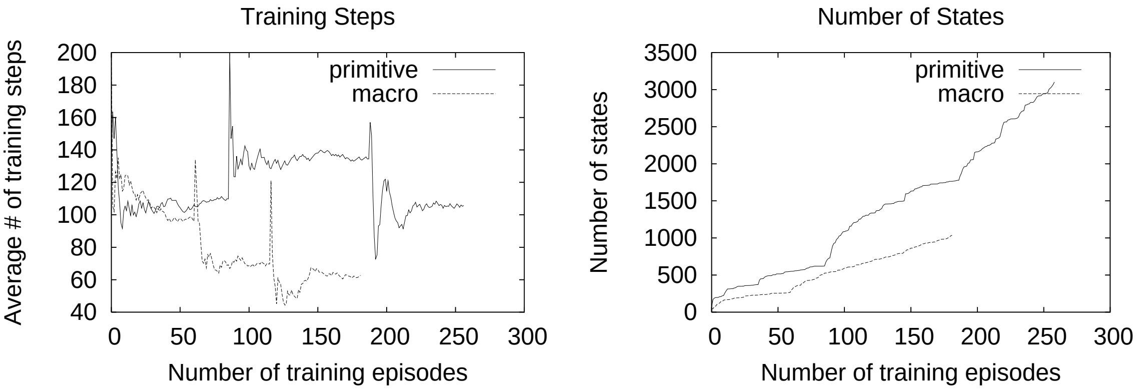


Figure 34: The graph on the left shows the average number of training-steps per episode as a function of the number of episodes. The graph on the right shows the number of grid-points added during learning. The sharp changes in the graph are due to the resolution increase. Learning with macros requires fewer grid points and fewer steps in reaching the goal. The performance gets better as the resolution increases. Learning with primitive actions only requires more points. Due to the larger number of points required as the resolution increases, performance does not improve monotonically.

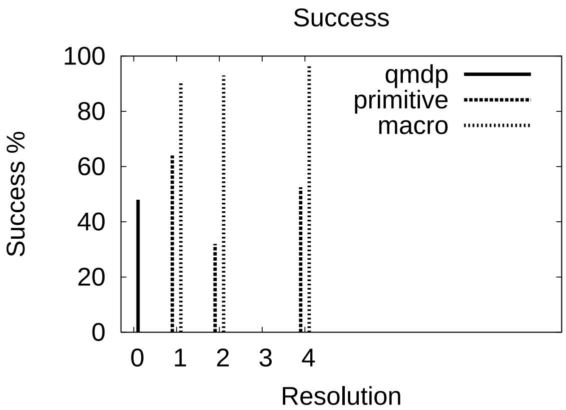


Figure 35: The figure shows the success percentage for the different methods during testing. The results are reported after training for each resolution.

macro-actions we get a better quality policy, which takes fewer steps to reach the goal, and is more successful than the QMDP heuristic, and the primitive-actions algorithm.

Without Reward Shaping We also performed experiments to investigate the effect of reward-shaping. Figure 36 shows that with primitive actions only, the algorithm fails completely. However, with macro-actions the algorithm still works, but not as well as with reward-shaping.

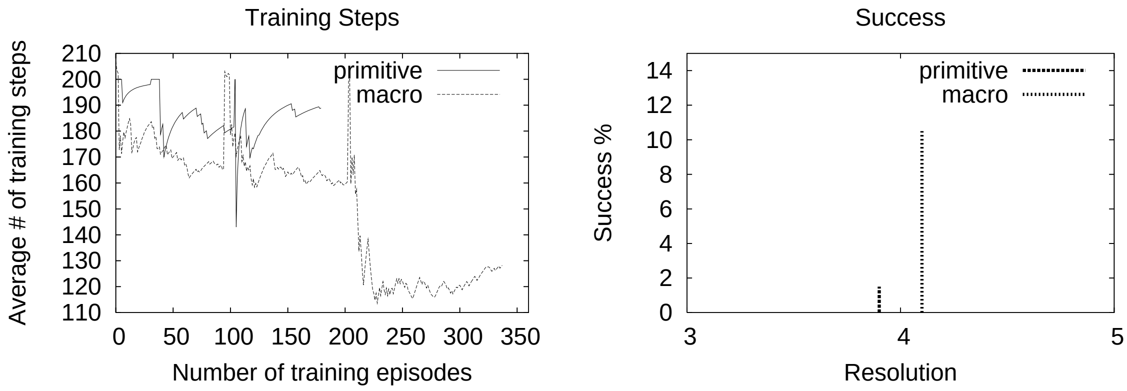


Figure 36: The graph on the left shows the average number of training-steps (without reward shaping). The figure on the right shows the success percentage. The primitive actions only algorithm fails completely.

Information Gathering Apart from simulated experiments we also wanted to compare the performance of QMDP with the macro-action algorithm on a platform more closely related to a real robot. We used the Nomad 200 simulator and describe a test in Figure 37 to demonstrate how our algorithm is able to perform information gathering, as compared to QMDP.

8. Conclusions

In this paper we have described learning and planning algorithms for POMDPs that take into consideration spatial and temporal abstraction. We demonstrated the advantages of our approach in the domain of robot navigation. There is a diverse literature on robot navigation. One of the most popular approaches, is simultaneous localization and mapping (SLAM), which can build on-line maps and localize the robot at the same. Even though this framework has been recently shown to scale to really large navigation domains (Leonard & Newman, 2003), (Paskin, 2003), (Thrun, Liu, Koller, Ng, Ghahramani, & Durrant-Whyte, 2003), it is still just a specific approach to robot navigation. Our algorithms on the other hand, do a an excellent job is solving the problem of robot navigation while at the same time providing a general framework for sequential decision making under uncertainty. Our approach exploits spatial and temporal abstractions and as a result perform significantly better than analogous flat approaches.

Specifically, the results show that the hierarchical learning algorithms produce learned models which are statistically closer to the generative model according to our distance metric. Also, the hierarchically learned models produce better robot localization. Additionally, the hierarchical learning algorithms are better at inferring the structure of the environment at the abstract level than the EM algorithm for learning flat POMDPs. The approximate

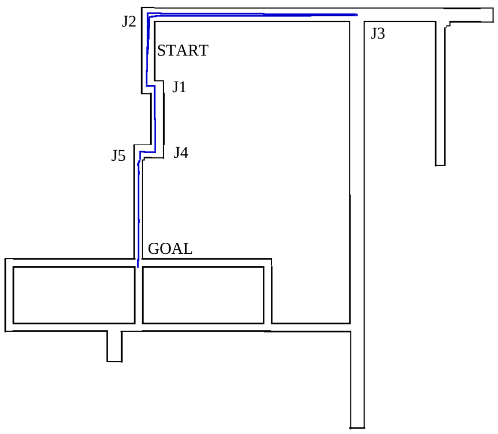


Figure 37: The figure shows the actual floor as it was designed in the Nomad 200 simulator. For the QMDP approach, the robot starts from START with uniform initial belief. After reaching J2 the belief becomes bi-modal concentrating on J1 and J2. The robot then keeps turning left and right. On the other hand, with our macro-action algorithm, the robot again starts from START and a uniform initial belief. Upon reaching J2 the belief becomes bimodal over J1 and J2. The agent resolves its uncertainty by deciding that the best action to take is the go-down-the-corridor macro, at which point it encounters J3 and localizes. The robot then is able to reach its goal by traveling from J3, to J2 , J1, J4, and J5.

training strategies which take advantage of short length data sequences have their own advantages. While the "reuse-training" method converges quickly (takes a small number of training epochs) and usually produces better models than all other learning algorithms, the "selective-training" method in practice trains faster (takes less time per training epoch) than any of the learning algorithms.

In general the advantage of hierarchical learning methods is their ability to better recognize the relationship between states which are spatially far apart (due to the way transitions models are learned at the abstract levels). A disadvantage is that training data sequences need to start at some root child and finish at some root child. This implies that training sequences are in some respect labeled. For example, in corridor environments training sequences need to start at an entry state of an abstract state (which may be a corridor or a junction) and finish at the exit state of an abstract state. Nonetheless, this was our assumption, in that corridor navigation exhibits structure, such as sequences of observations where the corridors are followed by junctions and junctions by corridors.

Our planning algorithms use both spatial and temporal abstraction and combine the ideas of SMDP planning and partial observability. Using these algorithms the robot is able to execute navigation tasks starting from no initial positional knowledge, in fewer steps than flat POMDP strategies. The advantages of these algorithms were even more apparent as the environment model got larger. Other advantages of hierarchical planning is that macro-action models can be computed and evaluated once, and then reused for multiple navigation tasks. Additionally, the fact that abstract plans are mappings from entry states (which is a small part of the total state space) to macro-actions reduces planning time. The ability to successfully execute navigation tasks starting from an unknown initial position makes this navigation system a strong candidate among state of the art navigation systems. Moreover, we have introduced an RL approach for POMDPs with macros that can be potentially applied to arbitrary POMDPs.

A limitation of our system is that a good initial H-POMDP model needs to be provided. Therefore, we are currently exploring structure learning. We are interested in learning the number of states as well as the vertical and horizontal relationships. We are investigating general Bayesian model selection methods (Stolcke & Omohundro, 1992), (Brand, 1999), techniques for learning compositional hierarchies (Pfleger, 2002) and other general approaches from natural language processing (de Marcken, 1996). We are also exploring methods for automatically learning the topology (structure) of spatial environments while at the same time estimating the robot's position. It seems that the H-POMDP model as it is currently applied in robot navigation naturally combines topological (abstract levels) and metric (production states) approaches for robot navigation and mapping. We are beginning to explore methods that combine topological and metric SLAM (Tomatis, Nourbakhsh, & Siegwart., 2003).

We are also exploring the dynamic creation of macro-actions and the application of the algorithms to arbitrary POMDP problems, that have appeared in the literature. In addition we are attempting to solve some real world problems. We are currently applying hierarchical POMDP models and planning algorithms to the problem of activity monitoring (e.g., cooking) and prompting. In this scenario a person uses objects in his/her house (all marked with RFID tags). The problem is to build hierarchical models of activity for that person, monitor the person's behaviors and guide the person in completing his/her activities when needed.

In the future we will try to combine the idea of decomposing the hidden state space with the idea of explicit planning with macros over belief space. In many real world situations it might be the case that an agent may only need to reduce its uncertainty over a specific set of variables rather than the whole set. We believe that a desirable way of scaling up the problem of sequential-decision making under uncertainty is by discovering generic primitives that can be combined for learning more complicated behaviors. We have taken some initial steps in this paper by examining primitives in terms of hidden state space abstractions belief discretizations, factored representations and temporal abstractions.

References

Aström, K. J. (1965). Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications*, *10*, 174–205.

Baum, L. E., & Sell, G. R. (1968). Growth functions for transformations on manifolds. *Pac. J. Math.*, *27*(2), 211–227.

Bellman, R. (1957). *Dynamic Programming.* Princeton University Press.

Bersekas, D. P. (1987). *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall.

Borenstein, J., Everett, H. R., & Feng, L. (1996). *Navigating Mobile Robots: Sensors and Techniques*. A. K. Peters, Ltd, Wellesley, MA.

Brand, L. (1999). Structure learning in conditional probability models via an entropic prior and parameter extinction. *Neural Computation Journal*, 1155–1182.

Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence Journal*, *47*, 139–159.

Bui, H., Venkatesh, S., & West, G. (2000). On the recognition of abstract Markov policies. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI)*.

Chen, S., & Weng, J. (2000). State-based SHOSLIF for indoor visual navigation. *IEEE Trans. Neural Networks*, *11*(6), 1300–1314.

Connell, J. H. (1989). *A Colony Architecture for an Artificial Creature*. Ph.D. thesis, MIT.

Cox, I. J. (1991). Blanche- an experiment in guidance and navigation of an autonomous mobile rrobot. In *IEEE Transactions Robotics and Automation*, Vol. 7(3).

de Marcken, C. (1996). *Unsupervised language acquisition*. Ph.D. thesis, MIT AI lab.

Dempster, A., Laird, N., & Rubin, D. (1977). Maximum likelihood from incomlete data via the EM algorithm. *Journal of the Royal Statistical Society, 39 (Series B)*, 1–38.

Dietterich, T. G. (2000). Hierachical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*.

Fine, S., Singer, Y., & Tishby, N. (1998). The Hierarchical Hidden Markov Model: Analysis and Applications. *Machine Learning*, *32*(1), 41–62.

Gaussier, P., Joulain, C., Zrehen, S., & Banquet, J.-P. (1997). Visual navigation in an open environment without a map. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, Grenoble.

Hauskrecht, M. (2000). Value-function approximations for partially observable Markov decision processes. *Journal of Artificial Intelligence Research*, *13*, 33–94.

Howard, R. A. (1971). *Dynamic Probabilistic Systems: Semi-Markov and Decision Processes*. New York:Wiley.

Juang, B. H., & Rabiner, L. R. (1985). A probabilistic distance measure for hidden Markov models. *AT&T Tech J.*, *64*(2), 391–408.

Koenig, S., & Simmons, R. (1998). A robot navigation architecture based on partially observable Markov decision process models. In Kortenkamp, D., Bonasso, R., & Murphy, R. (Eds.), *Artificial Intelligence Based Mobile Robotics:Case Studies of Successful Robot Systems*, pp. 91–122. MIT press.

Lari, K., & Young, S. (1990). The estimation of stochastic context-free grammars using the Insight-Outside algorithm. *Computer Speech and Language*.

Leonard, J., & Durrant-Whyte, H. F. (1991). Mobile robot localizatin by tracking geometric beacons. In *IEEE Transactions Robotics and Automation*, Vol. 7(3), pp. 376–382.

Leonard, J., & Newman, P. (2003). Consistent, convergent, and constant-time slam. In *International Joint Conference in Artificial Intelligence (IJCAI)*.

Levinson, S. (1986). Continuosly variable duration hidden Markov models for automatic speech recognition. *Computer Speech and Language*.

Lovejoy, W. S. (1991). Computationally feasible bounds for partially observed Markov decision processes. *Operations Research*, *39(1)*, 162–175.

Mahadevan, S., Marchalleck, N., Das, T., & Gosavi, A. (1997). Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *Proceedings of the four-teenth INternational Conference on Machine Learning*.

Mahadevan, S., Theocharous, G., & Khaleeli, N. (1998). Fast concept learning for mobile robots. *Machine Learning and Autonomous Robots Journal (joint issue)*, *31/5*, 239–251.

Mataric, M. (1990). A distributed model for mobile robot environment-learning and navigation. Tech. rep. TR-1228, MIT.

Moravec, H., & Elfes, A. (1985). High resolution maps from wide angle sonar. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 116–121.

Murphy, K. (2001). Applying the junction tree alogirthm to variable-length DBNs. Tech. rep., U.C Berkley, Computer Science Divison.

Murphy, K., & Paskin, M. (2001). Linear time inference in hierarchical HMMs. In *Neural Information Processing Systems (NIPS)*.

Nefian, A., & Hayes, M. (1999). An embedded HMM-based approach for face detection and recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing*.

Ng, A. Y., Harada, D., & Russell, S. (1999). Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*.

Nourbakhsh, I., Powers, R., & Birchfield, S. (1995). Dervish: An office-navigation robot. *AI Magazine, 53-60*, *16*(2), 53–60.

Ostendorf, M., Digalakis, V., & Kimball, O. (1996). From HMM's to segment models: a unified view of stochastic modeling for speech recognition. *IEEE Trans. on Speach and Audio Processing*, 4(5):360–378.

Parr, R. (1998). *Hierarchical control and learning for Markov decision processes*. Ph.D. thesis, University of California at Berkley.

Paskin, M. A. (2003). Thin junction tree filters for simultaneous localization and mapping. In *International Joint Conference in Artificial Intelligence (IJCAI)*.

Pfleger, K. (2002). *On-Line Learning of Predictive Compositional Hierarchies*. Ph.D. thesis, Stanford University.

Pineau, J., Gordon, G. J., & Thrun, S. (2003). Policy-contingent abstraction for robust robot control. In *Conference on Uncertainty in Articifical Intelligence*.

Prescher, D. (2001). Inside-Outside estimation meets dynamic EM. In *Proceedings of the 7th International Workshop on Parsing Technologies (IWPT-01)*, Beijing, China.

Puterman, M. (1994). *Markov Decision Processes: Discrete Dynamic Stochastic Programming*. John Wiley.

Rabiner, L. (1989). Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. In *Proceedings of the IEEE*, Vol. 77.

Roy, N., & Gordon, G. (2003). Exponential family PCA for belief compression in POMDPs. In *Advances in Neural Information Processing Systems*.

Russell, M., & Moore, R. (1985). Explicit modeling of state occupancy in hidden Markov models for automatic speech recognition. *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 5–8.

Russell, S. J., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (2nd edition). Prentice Hall.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, *5*, 115–135.

Sacerdoti, E. D. (1975). The nonlinear nature of plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pp. 206–214.

Shatkay, H. (1998). *Learning Models for Robot Navigation*. Ph.D. thesis, Brown.

Stolcke, A., & Omohundro, S. (1992). Hidden Markov model induction by Bayesian model merging. In *Neural Information Processing Systems (NIPS)*.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning An Introduction*. The MIT Press.

Theocharous, G. (2002). *Hierarchical Learning and Planning in Partially Observable Markov decision Processes*. Ph.D. thesis, Michigan State University.

Theocharous, G., & Kaelbling, L. P. (2004). Approximate planning in pomdps with macro-actions. In *Neural Information Processing Systems (NIPS)*.

Theocharous, G., Murphy, K., & Kaelbling, L. P. (2004). Representing hierarchical pomdps as DBNs for multi-scale robot localization. In *International Conference on Robotics and automation (ICRA)*.

Theocharous, G., Rohanimanesh, K., & Mahadevan, S. (2001). Learning hierarchical partially observable Markov decision processes for robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Seoul, Korea.

Thrun, S., Liu, Y., Koller, D., Ng, A., Ghahramani, Z., & Durrant-Whyte, H. (2003). Simultaneous localization and mapping with sparse extended information filters. Submitted for journal publication.

Thrun, S. (99). Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, pp. 21–71.

Thrun, S., Burgard, W., & Fox, D. (1998). A probabilistic approach to concurrent mapping and localization for mobile robots. *Machine Learning and Autonomous Robots (joint issue)*, 1–25.

Thrun, S., Burgard, W., & Fox, D. (2000). A real-time algorithm for mobile robot mapping with applications to multi-robot and 3D mapping. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, San Fransisco, CA.

Tomatis, N., Nourbakhsh, I., & Siegwart., R. (2003). Hybrid simultaneous map-building: A natural integration of topological and metric. *Autonomous Systems Journal*. In print.

Viterbi, A. J. (1967). Error bounds for convlutional codes and an aymptotically optimal decoding algorithm. In *IEEE Transactions Information Theory*, pp. 260–269.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, Cambridge University.

Zhou, R., & Hansen, E. A. (2001). An improved grid-based approximation algorithm for POMDPs. In *Proceedings of the Seventeenth International Conference in Artificial intelligence (IJCAI-01)*, Seattle, WA.

Appendix A. Table of H-POMDP symbols

Symbol	Definition
C_p^s	the set of production children of abstract state s
C_a^s	the set of abstract children of state s
C_n^s	the set of entry states that belong to abstract states children of s
X^s	the set of exit states that belong to abstract state s
N^s	the set of entry states that belong to abstract state s
$p(s)$	the parent of state s
s_x	exit state that belongs to abstract state s
s_n	entry state that belongs to abstract state s
$T(s'_n\|s_x, a)$	horizontal transition probability
$V(s'_n\|s_n)$	vertical transition probability
$O(z\|s, a)$	observation probability
$R(s, a)$	immediate reward function
$b(s)$	belief of state s
$TD(s_x\|i, \mu)$	Discounted transition model
$RD(i, \mu)$	Discounted reward model
M^s	the set of macro-actions available under abstract state s
π_μ^s	policy of macro-action μ under abstract state s

Table 11: Table of H-POMDP symbols

Appendix B. Computing α, β, ξ, and χ

B.1 Computing α

We can calculate the α variable recursively from the leaves of the tree to the root in ascending time order as follows:

- For $T = 1...N-1$ where N is the sequence length and for $level = D...1$ where D is the tree depth, we calculate the α terms. For all internal states we calculate the α terms for all sub-sequences $t, t+k$ and for the root only the sequence $1, T$.

  1. If s is a leaf state at $depth = level$, then for $t = T$ we can calculate α as shown in Equation 9 and described in Figure 38. When $p(s) = root$ we only use this equation for $t = 1$, because an observation sequence starts from the root and can only start at time $t = 1$.

  $$\alpha(p(s)_n, s, t, t) = V(s|p(s)_n)O(z_t|s, a_{t-1}) \tag{9}$$

  2. If s is a leaf state at $depth = level$ then for $t = 1...T-1$, and $k = T-t$ we can calculate α as shown in Equation 10 and described in Figure 39. The equation computes the probability of an observation sequence that started at time t from entry state $p(s_n)$ and finished at a child state s at time $t+k$. The computation is

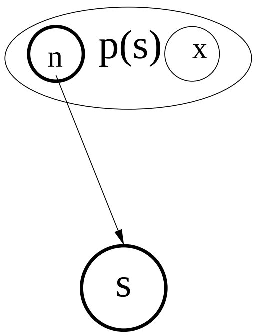


Figure 38: Equation 9 computes the probability that at time t the parent state of product state s, $p(s)$ produces a single observation z_t. The observation is produced by a vertical activation into product state s from entry state $p(s)_n$.

done iteratively by considering the probability of the observation sequence which finished at some child s'_x a time step earlier $(t+k-1)$. When $p(s) = root$ we only use this equation for $t = 1$, because an observation sequence starts from the root and can only start at time $t = 1$.

$$\alpha(p(s)_n, s, t, t+k) = \left[\sum_{s'_x \in C_x^{p(s)} \cup C_p^{p(s)}} \alpha(p(s)_n, s'_x, t, t+k-1) T(s|s'_x, a_{t+k-1}) \right] O(z_{t+k}|s, a_{t+k-1}) \quad (10)$$

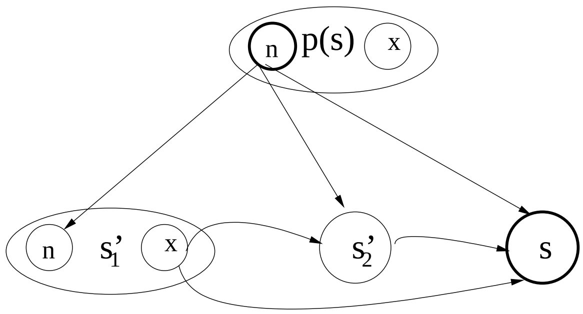


Figure 39: Equation 10 gives us the probability that the parent of the product state s, $p(s)$ produced observations from time t to time $t+k$, $(k \geq 1)$ and that the last observation was produced by product state s. s'_x is either the exit state of some abstract state s' (such as s'_{1_x}) or a product state s' (such as s'_2).

3. If s is an abstract state at $depth = level$ then for $t = 1...T$, and $k = T - t$ we can calculate α as shown in Equation 11 which can be better understood with Figure 40. When $p(s) = root$ we only use this equation for $t = 1$.

$$
\begin{aligned}
&\alpha(p(s)_n, s_x, t, t+k) = \\
&\sum_{s_n \in N^s} V(s_n|p(s)_n) \left[\sum_{i_x \in C_x^s \cup C_p^s} \alpha(s_n, i_x, t, t+k) T(s_x, |i_x, a_{t+k}) \right] \\
&+ \left[\sum_{l=0}^{k-1} \left[\sum_{s'_x \in C_x^{p(s)} \cup C_p^{p(s)}} \alpha(p(s')_n, s'_x, t, t+l) T(s_n|s'_x, a_{t+l}) \right] \right. \\
&\left. \left[\sum_{i_x \in C_x^s \cup C_p^s} \alpha(s_n, i_x, t+l+1, t+k) T(s_x, |i_x, a_{t+k}) \right] \right]
\end{aligned}
\tag{11}
$$

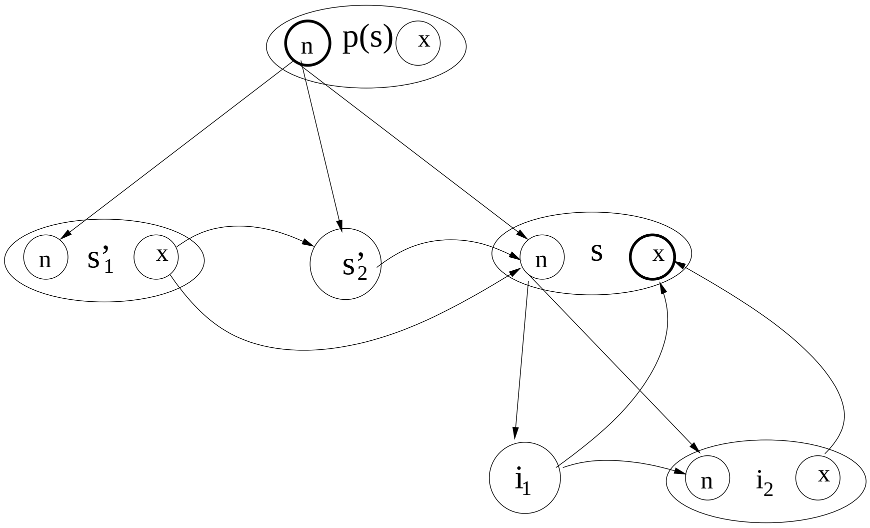


Figure 40: The figure illustrates Equation 11 which is used for situations where the parent of state s, $p(s)$ starting from entry state $p(s)_n$ produced observations from t to $t+k$ (where $k \geq 0$), and at time $t+k$ abstract state s was exited from s_x. We need to sum over all lengths of observation sequences that could have been produced by abstract state s. That is, s could have produced all observations $z_t ... z_{t+k}$, or in the other extreme z_t, z_{t+k-1} could have been produced by the adjacent states of s (such as s'_1 and s'_2), and only the last observation z_{t+k} produced by s.

B.2 Computing β

The beta variable can be calculated recursively from leaves to root. The time variable is considered in descending order as follows:

- For $level = D...1$ where D is the tree depth.

  1. If s is a leaf state at $depth = level$, for $t = N-1...1$ (where N is the sequence length and $T = N-1$), Equation 12 computes the probability of a single ob-

servation and is described in Figure 41. When $p(s) = root$ we only use these equations for $t = T$.

$$\beta(p(s)_x, s, t, t) = O(z_t|s, a_{t-1})T(p(s)_x|s, a_t), \;\; p(s) \neq root \tag{12}$$

$$\beta(p(s), s, t, t) = O(z_t|s, a_{t-1}), \;\; p(s) = root \tag{13}$$

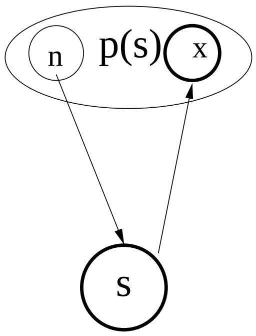


Figure 41: The figure illustrates Equation 12 which computes the probability that at time t the product state s produced observation z_t and then the system exited the abstract state $p(s)$ from exit state $p(s)_x$.

2. If s is an abstract state at $depth = level$, for $t = N-1...1$, Equation 14 computes the probability of observation z_t and is described in Figure 42. When $p(s) = root$ we only use these equations for $t = T$.

$$\beta(p(s)_x, s_n, t, t) =$$
$$\sum_{s_x \in X^s} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t) \right] T(p(s)_x|s_x, a_t), \;\; p(s) \neq root \tag{14}$$

$$\beta(p(s), s_n, t, t) = \sum_{s_x \in X^s} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t) \right], \;\; p(s) = root \tag{15}$$

3. For $t = N-2...1$

   (a) If s is a leaf state at $depth = level$, then for $k = 1...N-1-t$, we can calculate β as shown in Equation 16 and described in Figure 43. When $p(s) = root$ we only use this equation for $t + k = T$.

$$\beta(p(s)_x, s, t, t+k) =$$
$$O(z_t|s, a_{t-1}) \left[\sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} T(s'_n|s, a_t)\beta(p(s)_x, s'_n, t+1, t+k) \right] \tag{16}$$

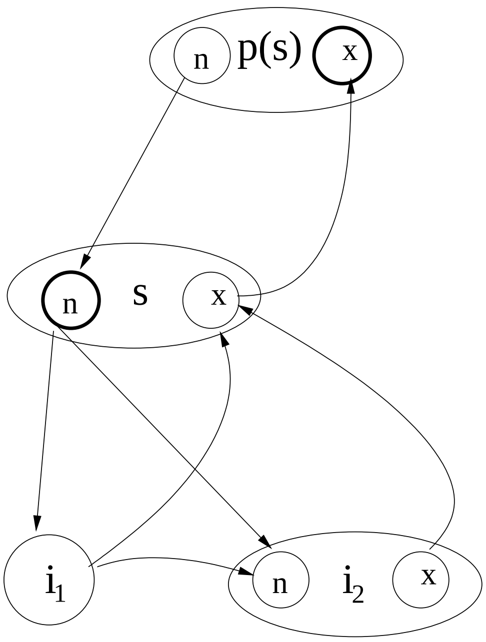


Figure 42: Equation 14 computes the probability that at time t that any of the children of abstract state s (such as i_1 and i_2) has produced observation z_t and then the parent of s $p(s)$ was exited from exit state $p(s)_x$.

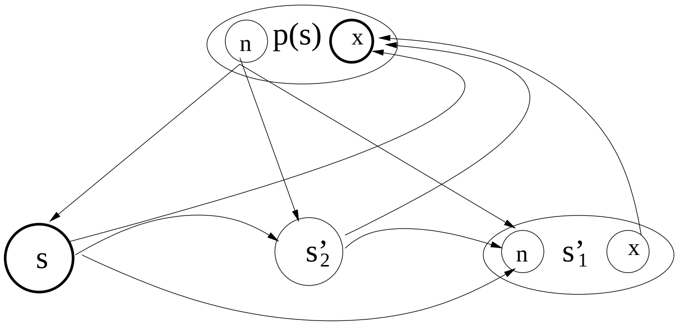


Figure 43: The figure illustrates Equation 16 which defines the probability that at time t the product state s produced observation z_t and its parent $p(s)$ continued to produce observations $z_{t+1}...z_{t+k}$ and exited at time $t+k$ from exit state $p(s)_x$, where $k \geq 1$ (or at least two observations are produced).

(b) If s is an abstract state at $depth = level$, then for $k = 1...N-1-t$, we can calculate β as shown in Equation 17 and described in Figure 44. When $p(s) = root$ we only use these equations for $t + k = T$.

$$\beta(p(s)_x, s_n, t, t+k) =$$
$$\sum_{s_x \in X^s} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t+k) \right] T(p(s)_x|s_x, a_{t+k})$$
$$+ \left[\sum_{l=0}^{k-1} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t+l) \right] \right. \tag{17}$$
$$\left. \left[\sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} T(s'_n|s_x, a_{t+l})\beta(p(s)_x, s'_n, t+l+1, t+k) \right] \right],$$
$$p(s) \neq root$$

$$\beta(p(s), s_n, t, t+k) =$$
$$\sum_{s_x \in X^s} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t+k) \right]$$
$$+ \left[\sum_{l=0}^{k-1} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t+l) \right] \right. \tag{18}$$
$$\left. \left[\sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} T(s'_n|s_x, a_{t+l})\beta(p(s), s'_n, t+l+1, t+k) \right] \right],$$
$$p(s) = root$$

B.3 Computing ξ

To simplify calculation of the ξ variable we define two auxiliary variables called η_{in} and η_{out}. The η_{in} is defined in Equation 19 and described in Figure 45.

$$\eta_{in}(t, s_n) = P(z_t, ...z_{t-1},\ s \text{ was entered at } t \text{ from } s_n \mid a_0, ..., a_{t-1},\ \lambda) \tag{19}$$

We can estimate η_{in} recursively from the root to the leaves of the tree.

1. If s is a child of the root and $t = 1$ we can calculate η_{in} as shown in Equation 20, and for $t > 1$ as shown in Equation 21

$$\eta_{in}(1, s_n) = V(s_n|p(s)_n),\ where\ p(s)_n = root \tag{20}$$

$$\eta_{in}(t, s_n) = \sum_{s'_x \in C_p^{p(s)} \cup C_x^{p(s)}} \alpha(p(s)_n, s'_x, 1, t-1)T(s_n|s'_x, a_t - 1) \tag{21}$$

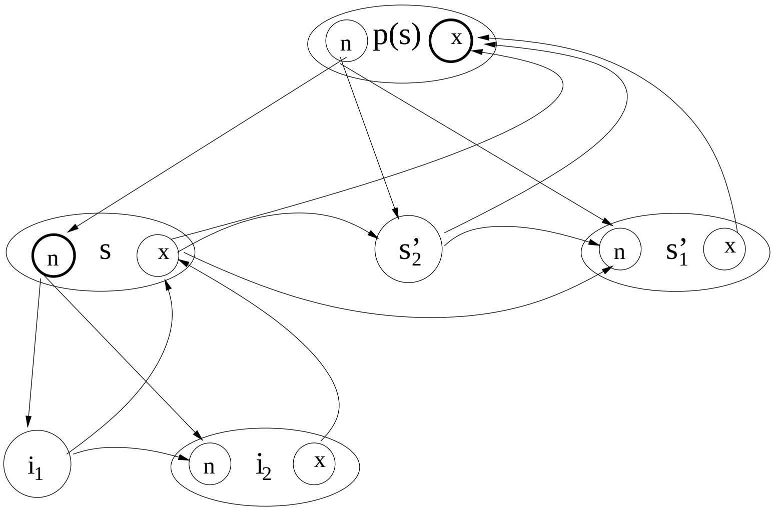


Figure 44: The figure describes Equation 17 which is the probability that at time t the abstract state s, starting from entry state s_n produced some observations, and then the parent of s, $p(s)$ continued to produce the rest of the observations and exited at time $t+k$ from exit state $p(s)_x$. Since s is an abstract state we have to consider all possible lengths of observations that could have been produced by abstract state s. So s could have produced a single observation z_t, up to all the observations $z_t ... z_{t+k}$, where $k \geq 1$.

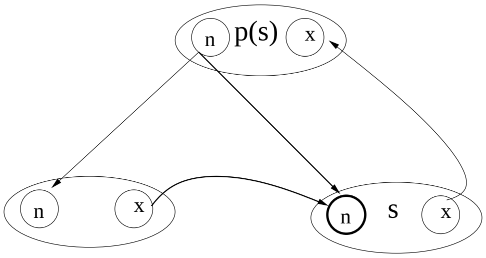


Figure 45: The η_{in} variable defines the probability that a state $s$is entered at time t either vertically or horizontally given a sequence of actions and observations up to time $t-1$.

2. If s is not a child of the root

$$\eta_{in}(t, s_n) = \\ \sum_{p(s)_n \in N^{p(s)}} \sum_{k=0}^{t-1} \eta_{in}(k, p(s)_n) \left[\sum_{s'_x \in C_p^{p(s)} \cup C_x^{p(s)}} \alpha(p(s)_n, s'_x, k, t-1) T(s_n | s'_x, a_{t-1}) \right] \\ + \eta_{in}(t, p(s)_n) V(s_n | p(s)_n) \tag{22}$$

which, is the probability that state s, which is not a root child, was entered at time t from entry state s_n. To calculate this probability we have to consider all the possible times $0...t-1$ and all possible entry states $p(s)_n$ that the parent of state s, $p(s)$ was entered.

The η_{out} variable is defined in Equation 23 and described in Figure 46. It can also be estimated recursively from root to leaves.

$$\eta_{out}(t, s_x) = P(s \text{ finished at } t,\ \text{from } s_x,\ z_{t+1}, ...z_T \mid a_t, ..., a_T,\ \lambda) \tag{23}$$

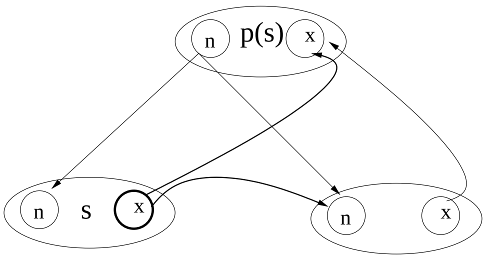


Figure 46: The η_{out} variable defines the probability that a state s exited at time t and observations $z_{t+1}, ...z_T$ were perceived and actions $a_t, ..., a_T$ taken.

1. If s is a child of the root.

$$\eta_{out}(t, s_x) = \sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} T(s'_n|s_x, a_t)\beta(p(s), s'_n, t+1, T), \quad t < T, p(s) = root \tag{24}$$

$$\eta_{out}(T, s_x) = 1.0 \tag{25}$$

2. If s is not a child of the root and for $t \leq T$, where $T = N - 1$.

$$\eta_{out}(t, s_x) =$$
$$\sum_{p(s)_x \in X^{p(s)}} \sum_{k=t+1}^{T} \left[\sum_{s'_n \in C_n^{p(s)}, \cup C_p^{p(s)}} T(s'_n|s_x, a_t)\beta(p(s)_x, s'_n, t+1, k) \right] \eta_{out}(k, p(s)_x)$$
$$+ T(p(s)_x|s_x, a_t)\eta_{out}(t, p(s)_x) \tag{26}$$

which is the probability that state s, which is not a root child, was exited at time t from exit state s_x and then the observations $z_{t+1}...z_T$ were produced. To calculate this probability we need to consider all the lengths of observation sequences from $t+1...T$, including a zero length observation sequence that could have been produced by the parent of s, $p(s)$ before $p(s)$ exits from exit state $p(s)_x$.

Now we can calculate ξ in Equation 1 recursively from the root to the leaves of the tree.

1. If s and s' are children of the root $p(s) = p(s)_x = p(s)_n$

$$\xi(t, s_x, s'_n) = \frac{\alpha(p(s)_n, s_x, 1, t) T(s'_n|s_x, a_t) \beta(p(s)_x, s'_n, t+1, T)}{P(Z|A, \lambda)}, \quad t < T \tag{27}$$

$$\xi(T, s_x, s'_n) = 0$$

2. If s and s' are not root children

$$\begin{aligned} &\xi(t, s_x, s'_n) = \\ &\frac{1}{P(Z|A,\lambda)} \left[\sum_{k=1}^{t} \sum_{p(s)_n \in N^{p(s)}} \eta_{in}(k, p(s)_n)\ \alpha(p(s)_n, s_x, k, t) \right] T(s'_n|s_x, a_t) \\ &\left[\sum_{k=t+1}^{T} \sum_{p(s)_x \in X^{p(s)}} \beta(p(s)_x, s'_n, t+1, k)\ \eta_{out}(k, p(s)_x) \right], \quad t < T \end{aligned} \tag{28}$$

$$\xi(T, s_x, s'_n) = 0$$

which calculates the ξ variable by considering all the possible times $1..t$ and entry states $p(s)_n$ that the parent state of s, $p(s)$ was entered and then at time t a transition was made from child exit state s_x to child entry state s'_n and then consider all the possible times from $t+1...T$ and all possible exit states $p(s)_x$ that $p(s)$ was exited from. The equation below shows the probability of going from the non-root child state s_x to the exit state $p(s)_x$ at time t.

$$\begin{aligned} \xi(t, s_x, p(s)_x) = \frac{1}{P(Z|A,\lambda)} &\left[\sum_{k=1}^{t} \sum_{p(s)_n \in N^{p(s)}} \eta_{in}(k, p(s)_n)\ \alpha(p(s)_n, s_x, k, t) \right] \\ &T(p(s)_x|s_x, a_t)\ \eta_{out}(t, p(s)_x), \quad t \leq T \end{aligned} \tag{29}$$

B.4 Computing χ

We can calculate the χ variable as follows:

1. If s is a child of the root $p(s) = p(s)_n = p(s)_x$

$$\chi(1, p(s)_n, s_n) = \frac{V(s_n|p(s)_n)\ \beta(p(s)_x, s_n, 1, T, s)}{P(Z|A, \lambda)} \tag{30}$$

2. Otherwise for $1 \geq t \leq T$

$$\begin{aligned} &\chi(t, p(s)_n, s_n) = \\ &\frac{\eta_{in}(t, p(s)_n)\ V(s_n|p(s)_n)}{P(Z|A,\lambda)} \sum_{k=t}^{T} \sum_{p(s)_x \in X^{p(s)}} \beta(p(s)_x, s_n, t, k)\ \eta_{out}(k, p(s)_x) \end{aligned} \tag{31}$$