Hierarchical Learning and Planning in Partially Observable Markov Decision Processes

By

Georgios Theocharous

A Dissertation

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science & Engineering

2002

Abstract

Hierarchical Learning and Planning in Partially Observable Markov Decision Processes

By

Georgios Theocharous

Sequential decision making under uncertainty is a fundamental problem in artificial intelligence. This problem is faced by autonomous agents embedded in complex environments (e.g., physical systems, including robots, softbots, and automated manufacturing systems) that need to choose actions to achieve long-term goals efficiently. Incomplete knowledge about the state of the environment, termed as hidden state, and the uncertain effects of their actions, makes sequential decision making difficult.

Partially Observable Markov Decision Processes (POMDPs) are a general framework for sequential decision making in environments where states are hidden and actions are stochastic. A POMDP model represents the dynamics of the environment, such as the probabilistic outcomes of the actions, and the probabilistic relationships between the agents observations and the hidden states. Unfortunately, "learning" the dynamics of the environment, and planning, scales poorly as the application domains get larger.

In this dissertation, we propose and investigate a Hierarchical POMDP (HPOMDP) model to scale learning and planning to large scale partially observable environments. In scaling planning, the key ideas are spatial and temporal abstraction. In spatial abstraction, grouping of lower level states into abstract higher level states reduces uncertainty of the agent. In temporal abstraction, the agent uses longer-duration actions instead of just primitive one-step actions. Macro-actions are crucial, in that they produce long traces of experience, which help the agent to disambiguate perceptually aliased situations. In scaling learning, the main advantage of hierarchy is that a multi-resolution problem representation has advantages in ease of maintenance, interpretability, and reusability. We investigate the new HPOMDP framework within the context of indoor robot navigation.

This dissertation makes the following principal contributions. We formally introduce and describe the HPOMDP model. We derive a hierarchical EM algorithm for learning the parameters of a given HPOMDP, and show empirical convergence. We introduce two approximate training methods, "reuse-training", and "selective-training", which result in fast training and better learned models. We also derive two planning algorithms for approximating the optimal policy for a given HPOMDP. We conduct a detailed experimental study of the learning algorithms for indoor robot navigation, and show how the robot localization improves with our learning methods. We also apply the planning algorithms to indoor robot navigation, both in simulation and in the real world. The results show that the planning algorithms are very successful in taking the robot to any environment state starting from no positional knowledge, and use significantly less number of steps than flat approaches.

To My Parents

Acknowledgments

First I would like to give many thanks to my advisor, Dr. Sridhar Mahadevan for his support, encouragement, and patience. His breadth as well as depth for the field of AI, and his enthusiasm and vision, have been invaluable lessons to me that I hope to carry into my future career. I am also in gratitude to my colleague Khashayar Rohanimanesh for his help, specially during the early stages. I would also like to thank my committee members, Dr. Anil K. Jain, Dr. John J. Weng, Dr. John M. Henderson (from Psychology), and Dr. Sven Koenig (from Georgia Tech) for their advice and feedback. I would like to thank all the members of the AGENTS lab, the Embodied Intelligence lab (EI), the Pattern Recognition and Image Processing lab (PRIP), and the Sequential Information Gathering in Machines and Animals lab (SIGMA) for their interest and feedback on the work presented in this dissertation. My appreciation is also extended to the DARPA MARS grant, and the NSF KDI grant awarded to Dr. Sridhar Mahadevan, which provided the financial support throughout my Ph.D. student years.

Finally, my deepest gratitude goes to my parents whose unconditional support throughout my student career have made this dissertation possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Sequential decision making under uncertainty is a fundamental problem both for artificial intelligence agents, and biological organisms (from insects to humans). It is the problem faced by entities embedded in complex environments that need to choose actions to achieve long-term goals efficiently. Incomplete knowledge about the state of the environment, and uncertainty of the action outcomes, makes this problem difficult. Partially Observable Markov Decision Processes (POMDPs) are a general framework for sequential decision making in environments where states are hidden and actions are stochastic. Unfortunately, learning and planning in POMDPs scales poorly as the application domains gets larger.

In this dissertation, we propose and investigate a Hierarchical POMDP (HPOMDP) model to scale learning and planning to large scale partially observable environments. In scaling planning, the key ideas are spatial and temporal abstraction, which reduce the agent uncertainty. In scaling learning, the main advantage of hierarchy is that a multi-resolution problem representation has advantages in ease of maintenance, interpretability, and reusability. We investigate the new HPOMDP framework within the context of indoor robot navigation.

In this chapter, in Section 1.1, we define the problem of sequential decision making under uncertainty, and provide a generic model of the problem. In Section 1.2 we describe the solution to the sequential decision problem under uncertainty. A solution to this problem can be computed through the rigorous mathematical framework of Markov decision processes. In section 1.3 we describe the notion of learning partially observable Markov decision processes which is achieved through an Expectation Maximization algorithm (EM). EM type algorithms iteratively modify the parameters of the model such that the log likelihood of the training data is increased after every iteration. In Section 1.4 we motivate our hierarchical approach, explain the key ideas behind our proposed solution, and briefly describe the derivation of our hierarchical POMDP model. In Section 1.5 we present an important instance of sequential decision making under uncertainty, which occurs in robot navigation. The robot navigation domain is used throughout the main chapters of the thesis, to demonstrate our hierarchical POMDP approach and its advantages over flat POMDP approaches. In Section 1.6 we list the principal contributions made in this dissertation. In the final Section 1.7, we describe the presentation structure for the rest of the dissertation.

1.1 Sequential decision making under uncertainty

Sequential decision making under uncertainty is a fundamental problem in artificial intelligence. This problem is faced by autonomous agents embedded in complex environments that need to choose actions to achieve long-term goals efficiently. It is also a core problem in many biological systems, examples of which include navigation, medical diagnosis, driving, and general everyday activity (from going to get lunch, to going to graduate school).

For example, in a medical diagnosis task, the physician (agent) cannot directly observe the internal state (hidden state) of the patient (environment) but rather perceives the symptoms (observations). Yet, after observing every symptom he has to issue a treatment (action) which in turn produces a new symptom. Treatments can produce side-effects because they can change the internal state of the patient in an unexpected manner. The goal of the physician is to come up with a treatment plan that will "heal" the patient (i.e., cause a transition in the patient's "state" from being "ill" to being "healthy"). The need to decide what is the best treatment that should be issued now is an instance of the problem of **sequential decision making under uncertainty**.

Formally the problem of sequential decision making under uncertainty is visualized in Figure 1.1, which depicts a generic embedded agent interacting with its environment. The agent is the decision maker while everything outside the agent, which the agent interacts with, is the environment. At each point in time the agent chooses the next action a according to its behavior B. The behavior B is a function that maps the agent's perception z into an action. The transition function T of the environment changes stochastically the state of the environment based on the current state s and current action a. The new state of the environment depends only on the current state and action pair, and not on previous state action pairs. The assumption that knowledge of the current state and action defines the next state is typically referred to as the *Markov assumption*. The new state of the environment produces a numerical reward signal r, which is communicated to the agent through the reward function R, and a perception z, which is communicated to the agent through the observation function O. The agent chooses actions such that its long-term reward is maximized. For some environments, O is the identity function; that is the agent has access to the true state of the environment. In real world domains however, the state of the environment is only partially observable to the agent.

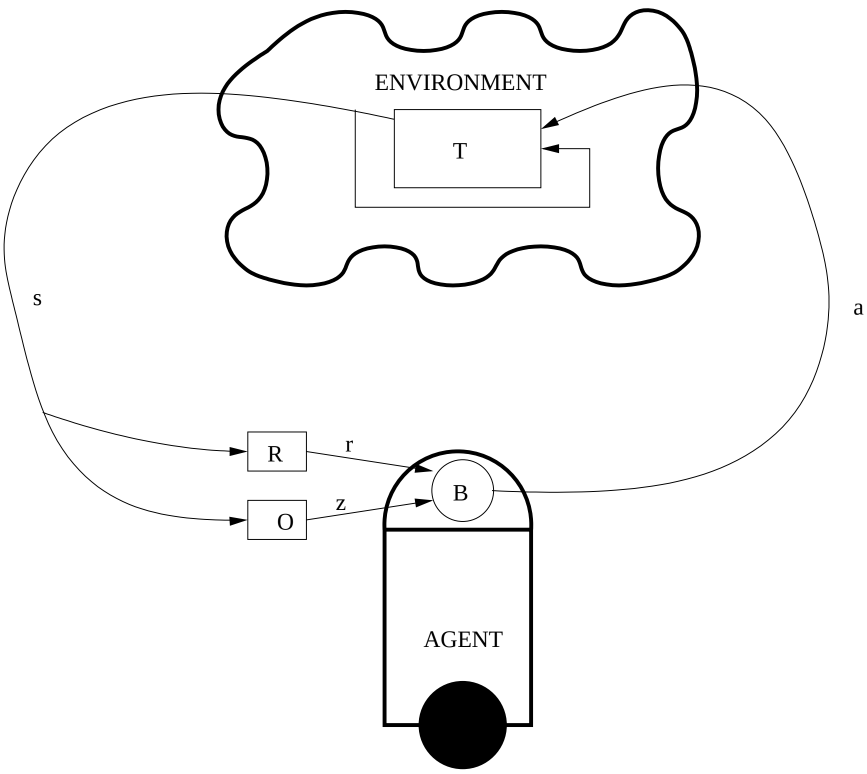

Figure 1.1: The agent interacts with its environment by issuing actions. The environment uses the previous state and current action and transitions into a new state according to the transition function T. The state generates an observation z and a reward signal r. The agent uses z and r to issue the next action. The agent chooses actions according to its behavior B, which maps perceptions to actions, such that that its long-term reward is maximized.

1.2 Planning

A widely adopted framework for modeling the sequential decision making problem under uncertainty is the Markov Decision Process (MDP) framework [3] [4]. MDPs model the states of the environment, the transition function T, the reward function R, and assume that the observation function O is the identity. An agent's task is to compute a policy (also referred to as a plan) π, which maps every state of the environment to an action through interaction with its environment such that the long-term reward is maximized. The policy can be computed either through dynamic programming, if R and T are known [5], or through reinforcement learning, if R and T are unknown [3]. A detailed description of the MDP framework is given later in Section 2.1.

A more general model, which does not assume that the observation function O is identity, is the partially observable Markov decision process model. A POMDP model represents the dynamics of the environment, such as the probabilistic outcomes of the actions (the transition function T), the reward function R, and the probabilistic relationships between the agents observations and the states of the environment (the observation function O). In a POMDP model the states of the environment are typically referred to as "hidden states", since they are not directly observable by the agent.

In a POMDP model the process is no longer Markovian, because an agent cannot predict the next perception based on the current perception and action. For an agent to be able to compute policies in POMDP models, using the MDP formalism, we need to create a new type of state representation, which is completely observable, and adheres to the Markov assumption. Intuitively, such a state representation can be formulated from history of past observations and actions. History, or memory of past observations and actions allows biological and artificial agents to disambiguate perceptually aliased situations (see Figure 1.2).

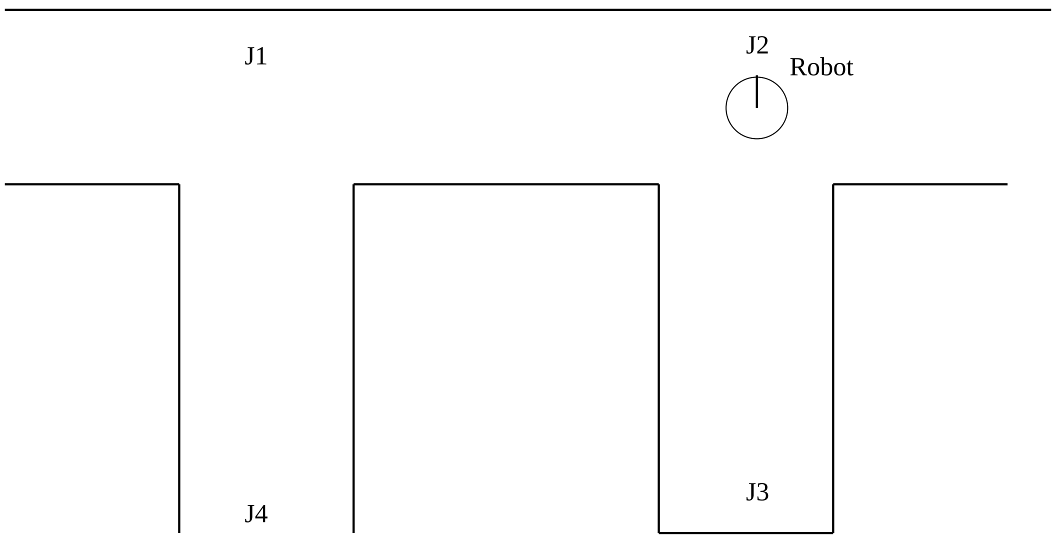

Figure 1.2: The figure shows a perceptually aliased situation where locations J1 and J2 generate the same observation (T-junction). These locations can be disambiguated if the robot uses memory of past observations and actions. For example, if the robot has perceived a dead-end (J3) and then moved forward where it perceived a T-junction (J1 or J2), then it knows it is at location J2.

In POMDPs memory is represented as a probability distribution over the hidden states of the environment. A probability distribution over the hidden states of the environment (otherwise called belief state) is a sufficient information state, in that it implicitly represents a unique history of observations and actions [6]. This type of MDP is called belief MDP and complies with the Markov assumption, in that having information about previous belief states cannot improve the choice of action [6]. The planning problem now becomes one of finding mappings from belief states to actions.

Using belief states as a new type of state representation however, does not scale well for large tasks with long planning horizons, since the number of belief states is infinite, which makes the problem computationally intractable [7], [8].

1.3 Learning

Learning a POMDP model is also difficult because an agent needs to learn the transition and observation functions of the model without true knowledge of where actions were taken and observations perceived. For example, in the robot navigation domain the robot has to learn the probability of perceiving each one of its available observations for every hidden state of the environment, and also the effects of its actions on the hidden states of the environment. This however is difficult, since the robot only maintains a probability distribution over the hidden states of the environment each time an action is taken and an observation perceived.

A general algorithm for learning the observation and transition functions of a POMDP is the Baum-Welch algorithm [9], [10], [11], which is an Expectation Maximization algorithm [12]. The EM procedure iterates between two steps. The expectation step (E-step) fixes the current parameters $\lambda_i = \langle T, O \rangle$, and computes the posterior probabilities over the hidden states S. The maximization step (M-step) maximizes the expected log likelihood of the perceptions Z as a function of the parameters, and produces a new set of parameters λ_{i+1}. After every iteration $\log P(Z|\lambda_{i+1}) \geq \log P(Z|\lambda_i)$.

Later in Section 2.2.2 we present a derivation of the EM algorithm and briefly sketch its convergence proof. We also describe the Baum-Welch algorithm which is an adaptation of the EM algorithm and is traditionally used in learning the parameters of a POMDP model [11].

1.4 Hierarchical learning and planning

In this dissertation, we explore how to scale the solution to large sequential decision making problems through hierarchical representations. Hierarchical learning and planning is well motivated from biology and occurs in many real world situations.

Honeybees for example, use different low resolution features when they are traveling either to the hive or to the food (such as mountains and trees), and fine resolution features when they are close to their goal (such as rocks and flowers) [13]. Humans also do not seem to be modeling the world at a uniform resolution [14]. In fact, humans use a hierarchical representation of spatial knowledge which is acquired through different stages (from sensory motor association to topological maps) [15], [16]. For example, if a human is about to drive from one city to the next, he does not plan based on joint movements of his arms and legs, but rather forms a plan based on a more abstract spatial resolution such as the highways or exits to take.

Hierarchical representations of sequential decision making tasks, where the actions are stochastic and the states are hidden, has the potential of significantly reducing uncertainty. There are two intuitive reasons why uncertainty would be reduced. First, uncertainty reduces at higher levels of abstraction. An intuitive example is human representation of space, where we usually feel more certain as to our world location at a coarse resolution of space, than a finer resolution. The second reason has to do with temporal abstraction, where an agent can execute multi-step actions. Macro-actions are crucial in reducing uncertainty in that they enable an agent to acquire long traces of experience from its environment, which help the agent to disambiguate perceptually aliased situations. For example, if we are lost in some unfamiliar environment, we can choose to follow a particular compass direction until we localize.

From an engineering point of view organizing a problem hierarchically has several 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 train separately 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 just train one of them and reuse it for all the others. In planning, we can construct abstract actions available to each abstract state that can be reused for any global plan without the need of worrying about the details inside the abstract states. For example, if we are given abstract actions to operate in any US city, and any US highway, then given any global destination city we can construct a plan to reach the destination city from any other city by simply using the precomputed abstract actions. Another form of reusing abstract actions would be to precompute abstract actions that can be used in every city (e.g. use the subway).

The cornerstone of our approach to hierarchical sequential decision making under uncertainty is the Hierarchical Hidden Markov Model (HHMM) framework [17]. We extend the HHMM model to a new model called Hierarchical POMDP (HPOMDP) which includes actions and rewards. POMDPs can also be viewed as extending the well-known hidden Markov model to include actions and rewards [10]. Figure 1.3 depicts the relationship between HMMs/POMDPs and HHMMs/HPOMDPs.

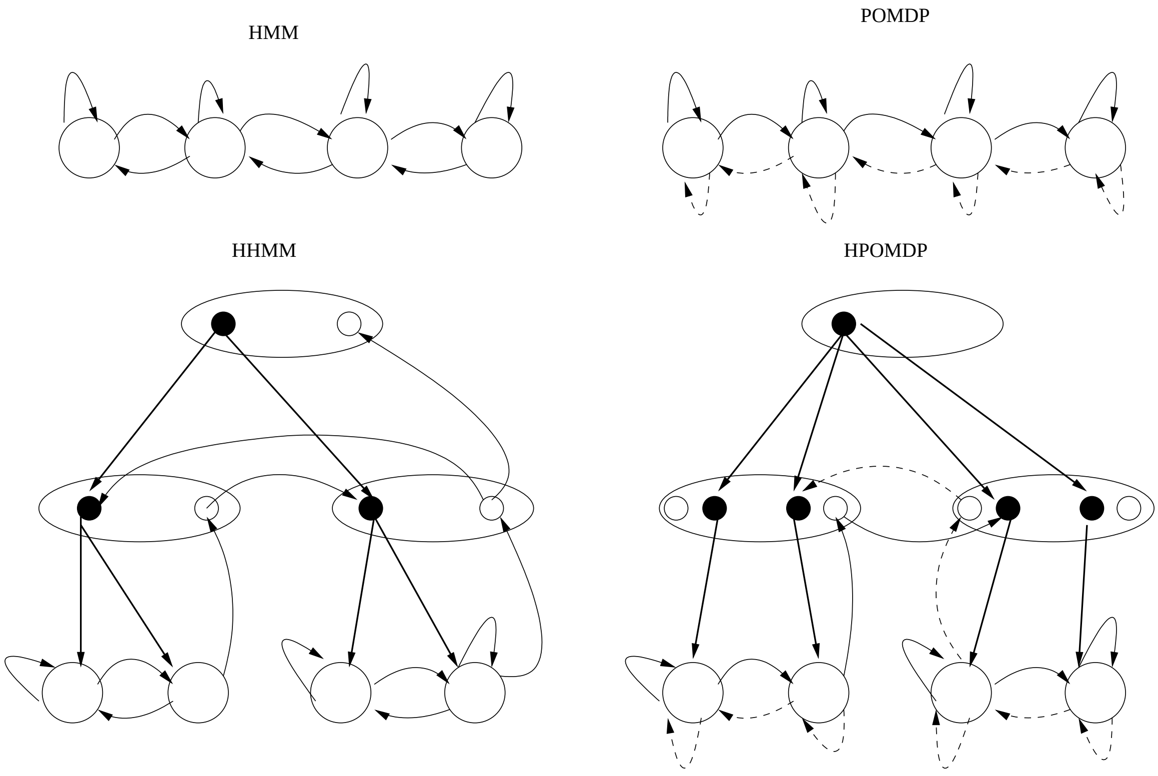

Figure 1.3: POMDPs can be thought of as an extension of HMMs with the addition of a separate transition matrix for each available action. In the same manner HHMMs are extended to HPOMDPs. In these example decision models there are two actions available, one denoted with solid arrows, and the other denoted with dashed arrows.

The HHMMs/HPOMDPs models differ from flat models in that they have abstract states which can activate children states through vertical transition matrices. Abstract states do not directly emit observations but rather produce sequences of observations which are output by their children states. Only primitive states directly emit observations which are the leaves of the tree. Horizontal transitions from abstract state are not allowed to occur until the process under the abstract state has finished (the small white circles inside the abstract states have been entered, see Figure 1.3). Detailed description of the HPOMDP model is presented later in Section 3.2.

1.5 Robot navigation

In this dissertation the HPOMDP model is applied to learning and planning in the context of robot navigation, which is a classic and difficult instance of the problem of sequential decision making under uncertainty. A robot agent acts in physical real world environments where it can sense the world through sensors such as sonar, infrared, laser, and camera, and acts upon the environment through effectors such as motion movements (wheel motors), camera movements (pan-tilt), and arm actions (pickup objects). Figure 1.4 shows a real robot Pavlov (a Nomad 200), which senses its environment (the corridors of the MSU Engineering Building) through sonar sensors and produces perceptions such as *wall*, and *opening*. After every perception, Pavlov can take actions such as “go-forward”, “turn-left”, and “turn-right”. Pavlov’s task is to be able to navigate to any location in the engineering building shown in Figure 1.6.

The major problem for Pavlov however, is the determination of its exact location in its environment. Unfortunately due to noisy sensors readings, and the fact that many location in the environment produce similar sensor values, no amount of image processing of sensor input is guaranteed to yield unique features describing its location. This realization is quite obvious through the corridor picture in Figure 1.4

where many locations in the corridor have similar visible features. Not only a robot agent has trouble identifying its true location but sometimes a human has trouble too.

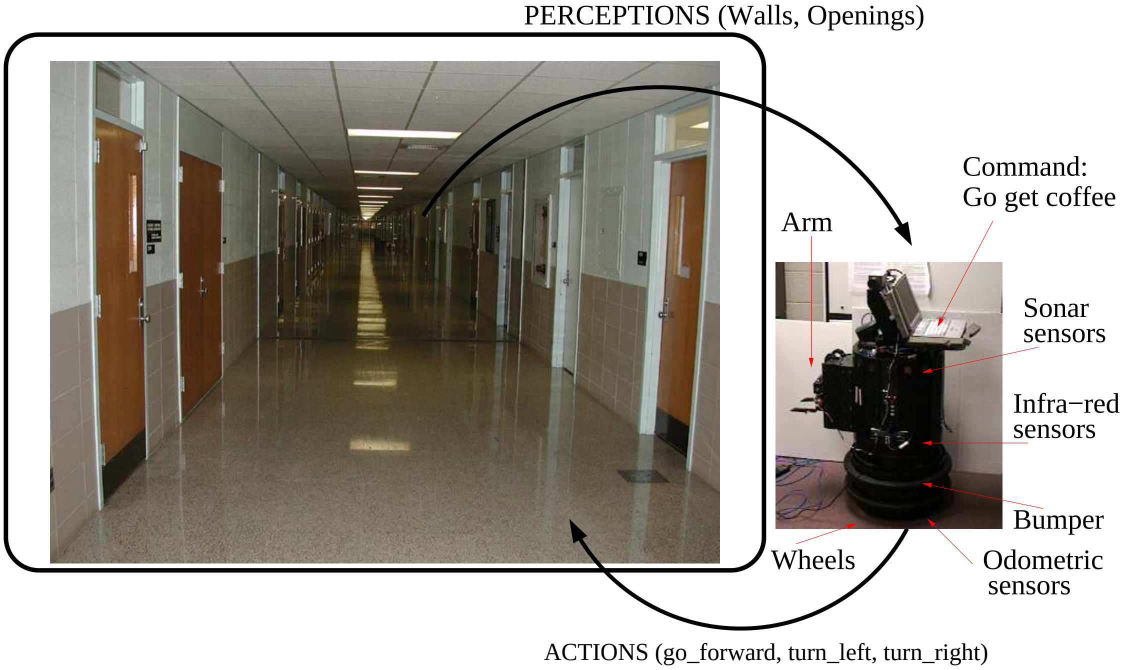

Figure 1.4: The picture on the left shows a section from a corridor on the 3rd floor of the Engineering Building at MSU. The picture on the right shows a real robot named Pavlov (a Nomad 200). Pavlov uses its sonar sensors to perceive walls and openings on the North, West and South direction. After every 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.

In order for Pavlov to be able to navigate to any location in the building it has to solve three problems. First, it has to learn a representation of the environment such as a map. Second, it needs to be able to construct plans which associate actions to map locations for every navigation task. And third, it has to be able to execute every navigation task successfully. A map representation could be a topological graph of the environment, where vertices are locations in the environment and edges are the outcomes of the actions. A topological representation can either be given, or can be learned through robot experience [18]. Given the graph representation then, a plan would be the shortest path from every vertex to the goal state, which can easily be computed using a shortest path algorithm such as Dijkstra's algorithm [19].

There are however two problems with this straightforward graph-theoretic solution: sensing and action uncertainty. In order to execute such a plan, the robot will need to know exactly in which state it is at, so that it can take the action associated with that state. One way to know the exact state of the robot is to start from an initial location and keep track of odometry. Unfortunately, keeping track of odometry (path-integration) is difficult due to wheel slippage, noisy wheel encoders and the fact that the robot may not be traveling a straight line due to dynamic obstacles. What's more, path integration requires that we give the robot the starting position.

An alternative, is to associate each state with a unique landmark that can be perceived by the robot's sensors. Still however, landmark based navigation is extremely prone to errors due to noisy sensors, noisy perception, and the fact that many locations in the environment generate the same observation (perceptual aliasing) [20]. We need a framework that can incorporate sequences of observations as memory from the last distinct landmark. In biology, bees and ants seem to combine both path integration and landmark based navigation to produce robust navigation behavior [21], [13].

The partially observable Markov decision process framework has been proven to be a reliable approach to robot navigation, which seamlessly integrates both motion and sensor reports [22], [23], [24]. Unlike pure odometric and landmark based systems it also incorporates the uncertainty of sensors and actuators. In POMDP based navigation systems the robot does not maintain a single estimate of its current location, but rather a probability distribution over all possible locations (hidden states) in the environment. A Bayesian update routine is used to update this probability distribution after every motion and sensor report. Thus, the robot is never completely lost because it always maintains a belief about its true location.

An example POMDP representation for robot navigation is shown in Figure 1.5, where the states correspond to fixed regions (e.g., each square meter in the environment is modeled as four states of the POMDP, one for each robot orientation). The robot can take non-deterministic actions such as "go-forward", "turn-left", and "tun-right", and in each state the robot can perceive features such as wall, and opening on the front, left, back and right side.

The advantage of the POMDP framework is that it provides rigorous algorithms for learning the transitions matrices and observation models, and for computing mappings from belief states to actions. Modeling a real office environment as a flat POMDP however, may require many thousands of states. The current algorithms scale poorly in terms of learning and planning in large POMDPs. It is difficult to learn large models since the standard learning algorithm (the Baum-Welch [10]) does not provide an obvious way of reusing previously learned sub-models. Exact planning methods based on dynamic programming are also intractable in large models. Additionally, heuristic planning methods such as Most Likely State (MLS) (which only looks at the peak of the distribution) often used in their stead are mainly effective in

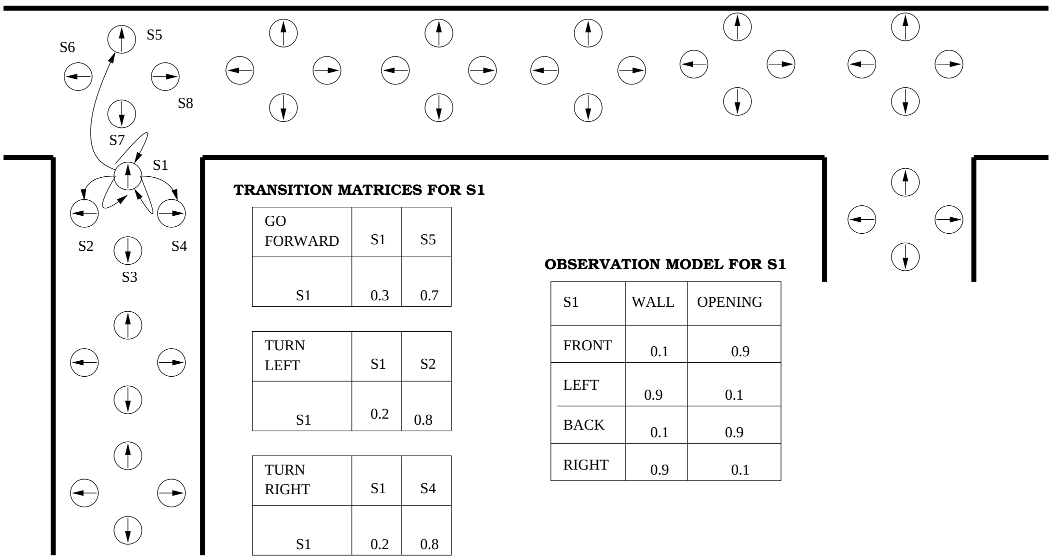

Figure 1.5: A corridor environment modeled as a flat POMDP. The circles are the hidden states of the model and represent the pose of the robot (location and orientation) in the environment. The figure shows the transition matrix for the three available actions for state $S1$, and also the observation model associated with state $S1$.

belief states with unimodal low entropy distributions (see Figure 1.6) [22]. In this dissertation we apply a hierarchical solution to learning and planning in robot navigation using our hierarchical POMDP framework. Figure 1.7 shows an example hierarchical representation of a corridor environment where abstract states group together fine resolution primitive states.

1.6 Contributions

The contributions of this dissertation are summarized as follows:

- We formally introduce and describe the hierarchical partially observable Markov decision process model. This model is derived from the hierarchical hidden Markov model with the addition of actions and rewards. Unlike flat partially

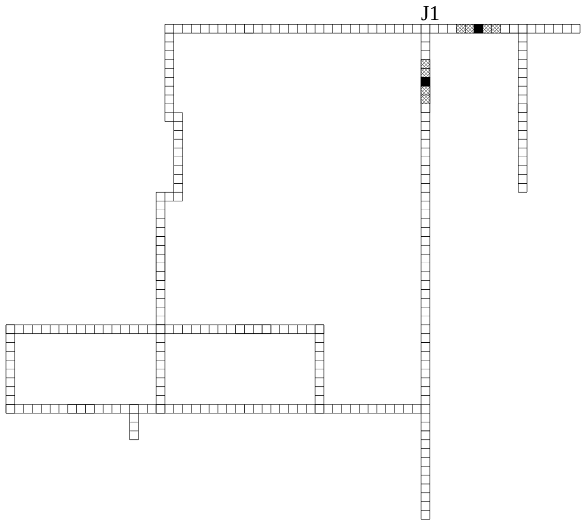

Figure 1.6: The figure shows a map of the third floor of the MSU Engineering Building. The squares indicate two square meter locations in the environment. The shading represents the probability distribution of possible locations, and darker means higher values. Pavlov's task is to be able to get to any one of the squares in the map starting from any other square. A problem that may arise during navigation is that the robot's belief as to its true location may be multimodal. This could happen either because the starting location of the robot was unknown, or due to sensor and motion noise. For example, due to sensor and motion noise at junction $J1$, the robot was unable to distinguish the east from the south corridor.

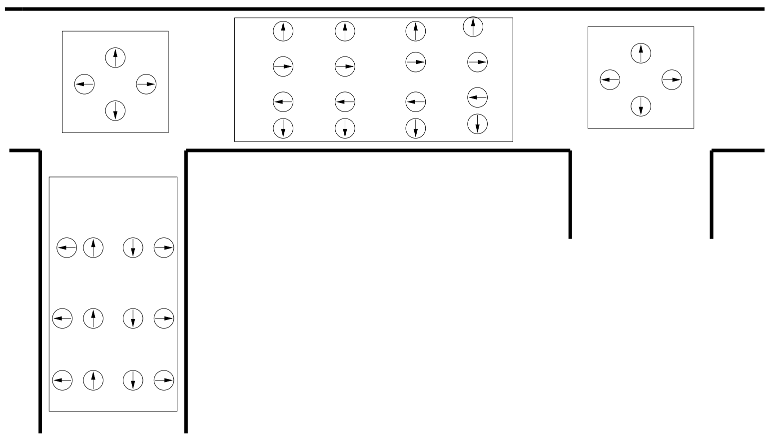

Figure 1.7: The figure illustrates a multi-resolution spatial model. The rectangles indicate abstract states representing corridors and junctions. The circles in the rectangle indicate fine resolution states which are children of the rectangle abstract states.

observable Markov decision process models, it provides both spatial and temporal abstraction. Spatial abstraction is achieved by low resolution abstract states, which group together finer resolution states in a tree like fashion. In temporal abstraction, a transition from an abstract state cannot occur unless the process under the abstract state has finished. This model is presented in Chapter 3.

- We derive an EM algorithm for learning the parameters of a given HPOMDP. We empirically show that the algorithm converges by increasing the log likelihood of the data after every training epoch. The algorithm is presented in Chapter 3, and the empirical convergence results described in Chapter 5.

- We introduce two approximate training methods. For the first method, which we named "selective-training", only selected parts of an overall HPOMDP are allowed to be trained for a given training sequence. This results in faster training. For the second method, which we named "reuse-training", submodels of an HPOMDP model are first trained separately and then reused in the overall model. This results in better learned models. Both algorithms and results are presented in Chapter 5.

- We derive two planning and execution algorithms for approximating the optimal policy in a given HPOMDP. The planning and execution algorithms combine hierarchical solutions for solving Markov decision processes, such as the options framework [25], and approximate flat POMDP solutions [22] (Chapter 6).

- 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 HPOMDP models which results in better robot localization (Chapters 5 and 6). Additionally, we compare our learned models with equivalent flat POMDP models that are trained with the same data. We have developed approximate training methods to learn HPOMDP models that converge much more rapidly than standard EM procedures for flat POMDP models. When the learned HPOMDP 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 (Chapter 5).

- We apply the planning algorithms to indoor robot navigation, both in simulation and in the real world. Our algorithms are highly successful in taking 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 much faster, and use a much smaller number of steps in executing navigation tasks (Chapter 6).

1.7 Outline

In Chapter 2 we cover background material. We formally explain the Markov decision process framework, and the partially observable Markov decision process framework, which extends the MDP framework with hidden states. We describe the EM algorithm for learning hidden Markov models, and POMDPs. Additionally, we present the mainstream approaches to robot navigation, and report on previous studies of robot navigation involving POMDPs.

In Chapter 3 we first describe the hierarchical HMM model which has been the building block of our hierarchical approach. We then introduce the hierarchical POMDP model which is an extension of the HHMM with the addition of actions and rewards. We describe in detail the algorithm for learning HPOMDPs, and present a small example to demonstrate the correctness of the learning algorithm. The chapter continues with a discussion on the relation of hierarchical POMDPs to flat POMDPs. Finally, the chapter concludes with a discussion of related multi-resolution hidden Markov models.

In Chapter 4 we describe the robot navigation domain. We describe the different robot platforms, the navigation environments, and how they are represented as hierarchical POMDPs. We present a description of the features that the robot can perceive, and how they are computed through the use of a neural network. We conclude with a detail description of the overall robot navigation architecture.

In the next two chapters we describe in detail learning and planning in HPOMDPs within the context of robot navigation. Learning and planning are integrated in an overall navigation architecture shown in Figure 1.8. The topological map construction can be produced either by the programmer or can be learned. In our case we haven't implemented any automatic construction of topological maps, but rather we provide it. Automatic topological map construction has been addressed by many researchers

[18], [26], [27], and is not the focus of this dissertation. The topological map can be compiled into an HPOMDP representation, which can be used for navigation tasks. Before a navigation task can begin we construct a plan, which is a mapping from states to actions. During execution of the navigation task the robot updates its belief about its true position after every action and observation, and chooses the next appropriate action to execute according to its plan and belief state. A navigation task produces a sequence of observations and actions, which can be used through an EM learning algorithm to improve the parameters of the HPOMDP model.

In Chapter 5 we describe detailed learning experiments where we train HPOMDP models to represent hierarchical spatial maps of indoor office environments. Before any learning experiments, we present two approximate learning methods that enable fast training of large scale environments. We present a detailed experimental study where we learn the parameters of HPOMDP models for simulated spatial environments, and present learning experiments with actual robot platforms. Additionally, we present experiments for learning the structure of the environment using our learning algorithms. In structure learning, we allow for semi-ergodic models at abstract levels of the hierarchy, and prune away incorrect connections using our learning algorithms.

In Chapter 6 we describe planning and execution algorithms, which combine theory from hierarchical MDP solutions, such as the options framework [25], with approximate POMDP solutions, such as the MLS strategy. We present robot navigation results both in the Nomad 200 simulator and in the real physical world.

In the final Chapter 7, we summarize the main contributions of the dissertation, and describe directions for future research.

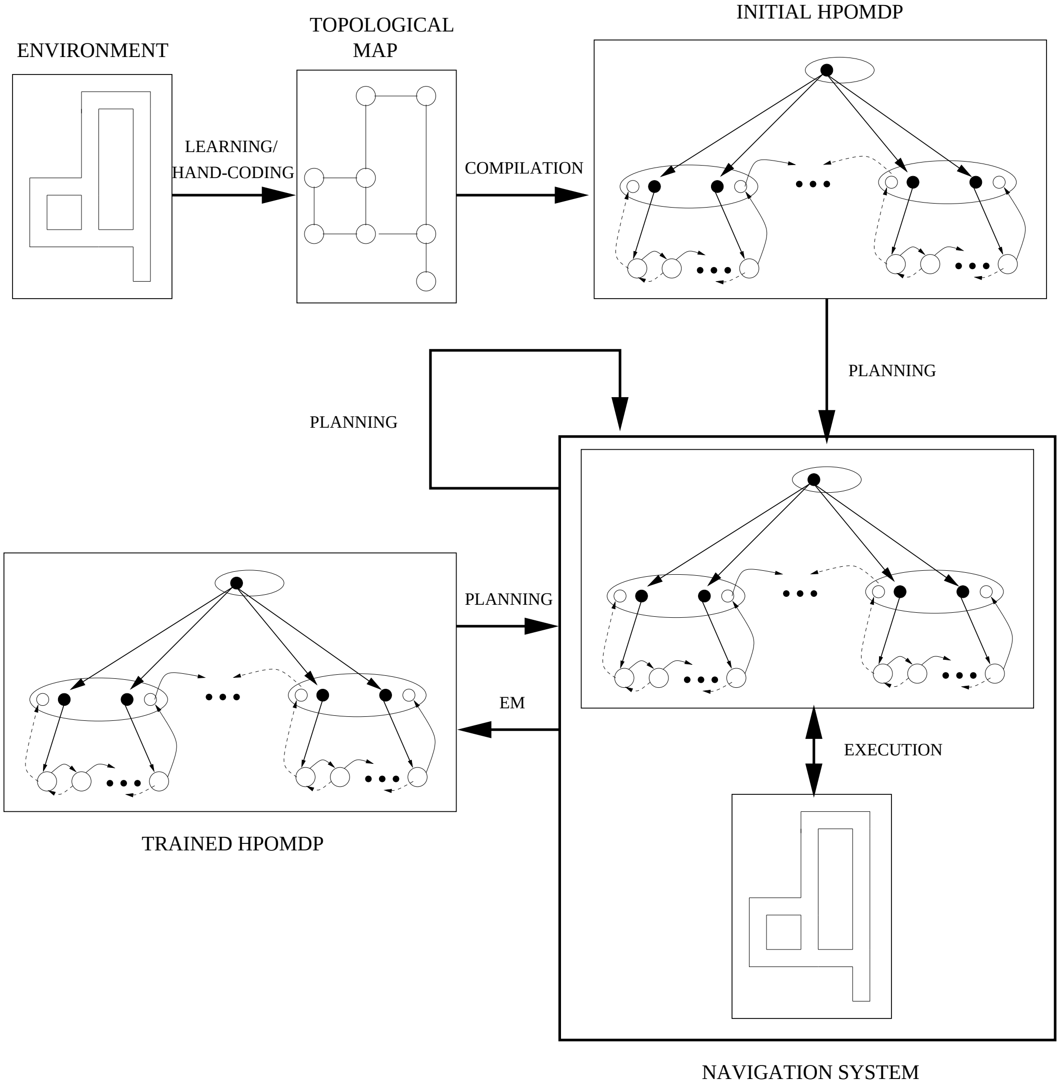

Figure 1.8: Topological map construction can either be learned or provided. The topological map is compiled into an HPOMDP representation which can be used for navigation tasks through planning and execution. Navigation tasks produce sequences of observations and actions that can be used in an EM training procedure to improve the parameters of the HPOMDP model.

Chapter 2

Background

In this chapter we cover background material. In Section 2.1 we explain Markov Decision processes which form the basic solution to sequential decision making. In Section 2.2 we explain the partially observable Markov decision process framework which extends the MDP framework by taking into consideration hidden states. In Section 2.2.1 we present the planning solution in POMDPs, and in Section 2.2.2 we describe the Baum-Welch algorithm for learning POMDP models. In Section 2.3 we present the main stream approaches to robot navigation; such as metric-based, and landmark-based navigation. Finally, in Section 2.4 we describe previous studies that have used POMDP-based navigation systems.

2.1 Markov Decision Processes

Markov Decision Processes (MDPs) model sequential decision making tasks, where an agent at each point in time observes its current state and based on that chooses what action to do next [28]. When an action is chosen, it is carried out and a reward is received from the agent's environment. Figure 2.1 illustrates an MDP modeling a simple navigation task. Formally MDPs are defined as a four tuple $\langle S, A, T, R\rangle$:

- S: A set of states that represents the state of the system at each point in time.
- A: A set of actions that an agent can take (can depend on state).
- $T : A \times S \times S \rightarrow [0, 1]$: The state transition function, which maps each state action pair into a probability distribution over the state space. The next distribution over the state space depends only on the current state action pair and not on previous state action pairs. This requirement ensures the *Markov property* of the process. We write $T(s, a, s')$ for the probability that an agent took action a from state s and reached state s'.
- $R : S \times A \rightarrow \Re$: The immediate reward function which indicates the reward for doing an action in some state.

An agent chooses actions so as to maximize its expected long-term reward. If the lifetime of the agent is finite, for example if the agent expects to live for the next T steps, it should choose to maximize its expected reward only for the next T steps. Such optimality criterion is called *finite-horizon* optimality and is shown in Equation 2.1, which denotes the expected sum of rewards the agent will receive starting from state s and following policy π (mapping from states to actions) for T steps.

$$V^{\pi}(s) = E^{\pi}\left(\sum_{t=0}^{T-1} r_t\right) \tag{2.1}$$

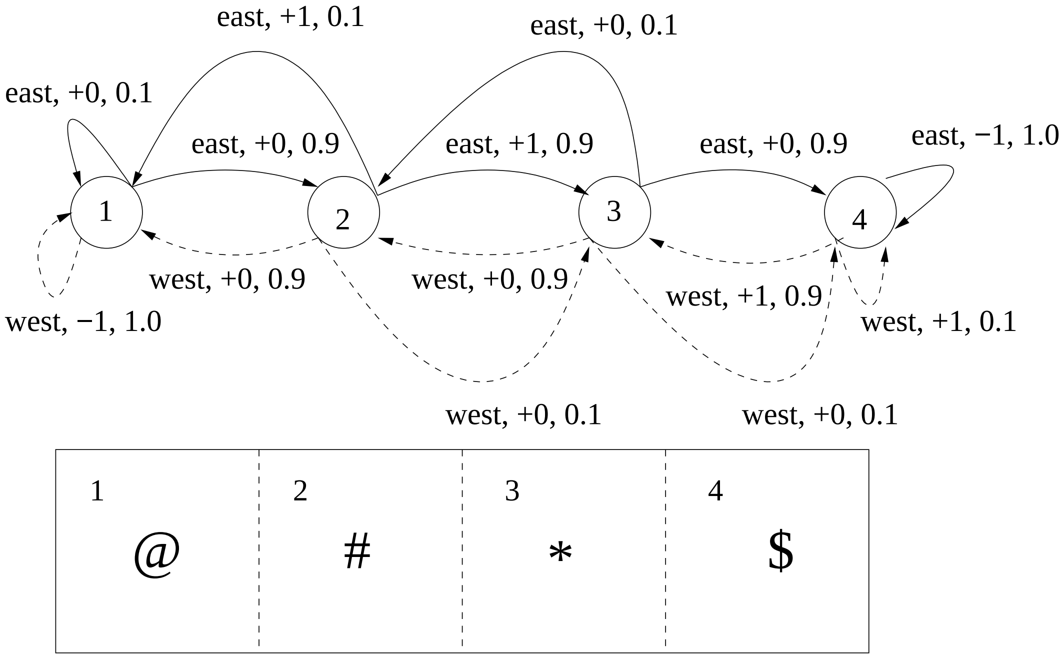

Figure 2.1: An MDP modeling a simple navigation task. There are two actions available *east* and *west* which succeed with probability 0.9. If they fail the robot moves in the opposite direction, or stays in the same state if the opposite direction is a wall. The reward function is structured such that the robot gets a reward of +1 for actions leading into state 3. Otherwise the robot gets a 0 reward except for a negative −1 reward for actions leading onto the wall. All four states are completely observable since they generate unique observation symbols.

However, if the agent's lifetime is infinite, or if the lifetime is very long compared to the length of each action, then what seems to be a better optimality criterion is the *infinite-horizon* which is shown in Equation 2.2. In this model, the agent chooses action so as to maximize its expected infinite discounted sum of rewards. The discount factor γ plays different roles. One reason is to give more weight to the rewards in the near future over the rewards in the far future. Another reason is to make sure that the sum is bounded.

$$V^{\pi}(s) = E^{\pi}\left(\sum_{t=0}^{\infty} \gamma^t r_t\right), \ 0 \leq \gamma \leq 1 \tag{2.2}$$

An agent who is taking actions so as to maximize one of the optimality criteria above is said to be executing a policy. The goal of the agent is to find the best or optimal policy, which will also receive the most reward in the long-run. In the finite-horizon model an optimal policy is typically non-stationary, that is it changes over time. This is because the agent takes into account how many steps are left in its life, and given the same situation (state), may choose a different action depending on how many steps it has left. In the infinite-horizon case, there is shown to be an optimal policy, which is stationary.

In order to evaluate a policy that an agent is executing, we can define the value of a state s given that the agent starts in state s and executes policy π as $V^{\pi}(s)$. In the finite-horizon the value of a state s given a policy π can be defined as the expected sum of reward gained from starting in state s and executing non-stationary policy π for t steps. This value function can be expanded recursively as shown in Equation 2.3.

$$V_t^{\pi}(s) = R(s, \pi_t(s)) + \gamma \sum_{s' \in S} T(s, \pi_t(s), s') V_{t-1}^{\pi}(s') \tag{2.3}$$

In the infinite-horizon case, the value of a state under some policy π is the expected discounted sum of future rewards for starting in state s and executing stationary policy π. This value function is recursively defined in Equation 2.4. The simultaneous solution for the linear set of equations that can be derived from Equation 2.4 (one for every $s \in S$) is unique and gives us the value function of policy π.

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V^{\pi}(s') \tag{2.4}$$

For example, if the policy in the navigation example in Figure 2.1 is $\pi = [east, east, east, west]$, then to compute the value function we need to solve a set of four linear equations shown below:

$$V(s_1) = \gamma \left[0.9V(s_2) + 0.1V(s_1)\right]$$

$$V(s_2) = 1 + \gamma \left[0.9V(s_3) + 0.1V(s_1)\right]$$

$$V(s_3) = \gamma \left[0.9V(s_4) + 0.1V(s_2)\right]$$

$$V(s_4) = 1 + \gamma \left[0.9V(s_3) + 0.1V(s_4)\right]$$

If we set $\gamma = 0.9$ then the solution is $[4.844, 5.442, 4.946, 5.501]$, which represents the values of the four states for policy π.

But, how can we find the optimal policy? One way is to use the optimal value function, from which we can find the optimal policy. First, for the finite-horizon case the optimal action to do for the last step is the one that gives the highest immediate reward (Equation 2.5).

$$\pi_1^*(s) = \text{argmax}_a R(s, a) \tag{2.5}$$

For every other step t, the optimal action to do in a state s is given by Equation 2.6.

$$\pi_t^*(s) = \text{argmax}_a \left(R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V_{t-1}^*(s') \right) \tag{2.6}$$

For the infinite-horizon case, there exists an optimal stationary policy that can be followed from any initial state s that can produce the maximum value for that state [5] . The value function for any optimal policy (note that there can be more than one optimal policy) is unique and is derived from the solution of the simultaneous equations defined in Equation 2.7.

$$V^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right), \ \forall s \in S \tag{2.7}$$

Given the optimal value function we can derive a greedy optimal policy as shown in Equation 2.8.

$$\pi^*(s) = \text{argmax}_a \left(R(s,a) + \gamma \sum_{s \in S} T(s,a,s') V^*(s') \right) \tag{2.8}$$

Thus one way to find optimal policies is to find the optimal value function. Unfortunately due to the max term in Equation 2.7, the set of equations derived is non-linear and therefore we cannot analytically solve the system of equations. However, we can find the optimal value function using an iterative procedure called *Value Iteration.* The value iteration algorithm works by iteratively computing the discounted finite-horizon optimal value functions as the horizon keeps increasing, and can be shown to converge to the correct values [29], [30].

However, if we don't know the transition and reward functions T and R we can use an on-line algorithm such as *Q-learning* [31]. We can do that if we re-express Equation 2.3 in terms of state action values called Q values shown in Equation 2.9.

. *initialize* $V(s)$ *arbitrarily*
. *loop until* $\max|V_t - V_{t-1}| < \epsilon$
. $t = t + 1$
. *loop* $for\ s \in S$
. *loop* $for\ a \in A$
. $V_t(s) = \max_a \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V_{t-1}(s')\right)$
. *end loop*
. *end loop*
. *end loop*

Figure 2.2: The Value Iteration algorithm is a dynamic programming solution to the non-linear set of equations defined by Equation 2.7. The algorithm works in an iterative fashion by computing the discounted finite-horizon optimal value functions as the horizon keeps increasing.

$$Q_t(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^{\pi}_{t-1}(s') \tag{2.9}$$

The intuition behind this re-expression is that if an agent has learned the Q values, it can now choose actions without knowledge of he transition function T and reward function R. An agent who has learned the optimal Q^* values (Equation 2.10) can choose the optimal action for each state my maximizing over the Q^* values (Equation 2.11).

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s') \tag{2.10}$$

$$\pi^*(s) = \mathrm{argmax}_a Q^*(s,a) \tag{2.11}$$

We can also express Equation 2.10 as Equation 2.12, which allows us to devise an iterative routine for calculating the optimal Q^* values.

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') max_{a'} Q^*(s',a') \tag{2.12}$$

The Q^* values can be estimated recursively using Equation 2.13 [31].

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{2.13}$$

If each action is executed in each state an infinite number of times on an infinite run and α is decayed appropriately, the Q values will converge with probability 1 to the optimal Q^* values [31, 32]. The r term is the immediate reward the agent receives from the environment when it executes action a in state s, and α is the learning rate.

2.2 Partially Observable Markov Decision Processes

To use the MDP formalism for planning an agent needs to accurately perceive the state of the environment. Unfortunately however, this is not the case in the real world. For example, imagine a mobile robot whose task is to navigate an indoor office environment and that the only features it can perceive are walls and openings. Obviously, this robot will be unable to tell its exact location in the environment since its local observation may correspond to different places in the world. Providing the robot more observation capabilities could reduce perceptual aliasing at the cost of defining and using more features. Nonetheless, it is always difficult to associate unique features for each possible location in he environment, especially for large scale robot navigation domains. Additionally, too many features can also give rise to perceptual aliasing in that many locations would correspond to similar features. For example, if we are navigating in a forest we are able to see an amazing number of tree shapes and sizes. Yet, for the most part it all appears the same. Thus, we would like to have an approach that deals with perceptual aliasing rather than an approach that tries to create unique features.

A way to to disambiguate these spatial locations that generate the same observations is to use memory, that is, to keep track of all observations and actions that the robot has taken. A popular way of representing memory is provided by the partially observable Markov decision process framework. In the POMDP framework, at each point in time the agent uses the history of past observations and actions to calculate a probability distribution over the underlying environment state (belief states). The belief states are a sufficient statistic for the history of the process [33]. This means that using the belief space as a new type of state representation we can compute optimal policies for POMDPs using the MDP framework.

Intuitively an agent that is able to act based on its uncertainty as to its true state will be more successful than simply acting on its current perception as to what the true state of the environment is. For example, if the robot knows that is highly confused it could take actions just for the sake of reducing its uncertainty. Or, given that it believes that it is at different locations at the same time it could take an action that is the best for all probable locations. Thus planning in POMDPs means that we need to map every possible probability distribution about the agents belief of its true state, to the best action for the task at hand.

2.2.1 Planning in POMDPs

The POMDP framework is a systematic approach that uses belief states to represent memory of past actions and observations. Thus, it enables an agent to compute optimal policies using the MDP framework. Formally a POMDP model is defined as a six tuple $\langle S, A, T, R, Z, O \rangle$:

- S, A, T, and R describe a Markov decision process.
- Z: A set of observations.
- $O : A \times S \times Z \rightarrow [0, 1]$: A function that maps the action at time $t-1$ and the state at time t to a distribution over the observation set. We write $O(s', a, z)$ for the probability of making observation z given that the agent took action a and landed in state s'.

A POMDP agent can be decomposed into two components as shown in figure 2.3. The state estimator takes as input the current observation, the action taken and the previous belief state b; and it outputs a new belief state b'. This calculation is shown in Equation 2.14.

$$\begin{aligned} b'(s') &= P(s'|z, a, b) \\ &= \frac{P(z|s', a, b)P(s'|a, b)}{P(z|a, b)} \\ &= \frac{P(z|s', a, b)\sum_{s \in S} P(s'|a, b, s)P(s|a, b)}{P(z|a, b)} \\ &= \frac{P(z|s', a)\sum_{s \in S} P(s'|s, a)P(s|b)}{P(z|a, b)} \\ &= \frac{O(s', a, z)\sum_{s \in S} T(s, a, s')b(s)}{P(z|a, b)} \end{aligned} \tag{2.14}$$

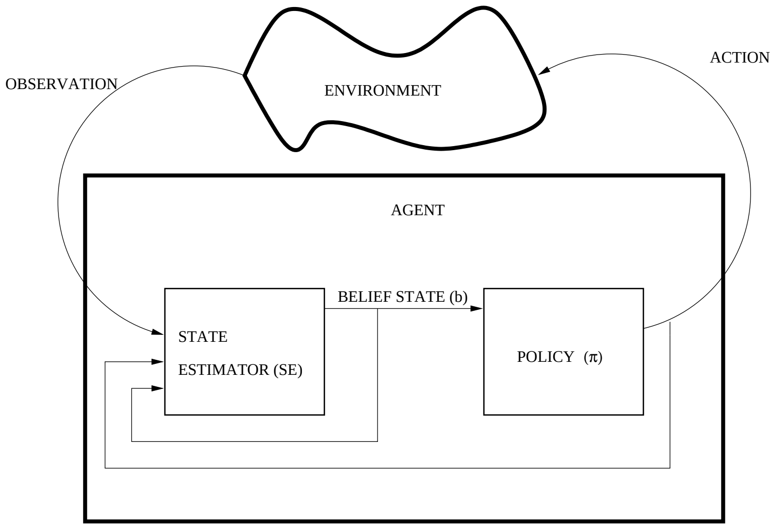

Figure 2.3: A POMDP agent is made of two main components. The state estimator module receives observations from the environment, the last action taken and the previous belief state and produces an updated belief state. The Policy module maps a belief state to an action.

Next, we demonstrate a sample belief state evolution based on the POMDP model described in Figure 2.4 and the belief update procedure described in Equation 2.14. If the starting belief state is [0.33 0.33 0.0 0.33] and the agent takes action *east* and does not perceive the $*$ observation, then the new belief state becomes [0.1 0.45 0.0 0.45]. If it takes the action *east* again and still does not perceive the $*$ observation, then the belief state becomes [0.1 0.164 0.0 0.736], which is a good indication that the agent has moved to the most right state.

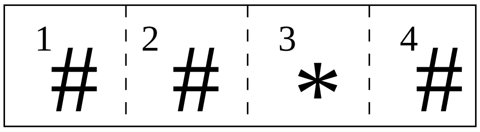

Figure 2.4: In this simple POMDP states 1, 2, and 4 generate the same observation while state 3 generates a different observation. The agent can take actions *east* and *west* which succeed with probability 0.9. If they fail, the agent moves in the opposite direction. If a movement is not possible the agent stays in the same state.

After a belief state estimation, the policy component of the POMDP agent must map the current belief state into an action. Since the belief state is a sufficient statistic, the optimal policy is the solution to a continuous-space belief MDP, and it is defined as follows:

- B: A set of belief states, where each belief state $b : s->[0,1]$ s.t. $\sum_s b(s) = 1$
- A: A set of actions that remain the same as the underlying MDP.
- $\tau : A \times B \times B \rightarrow [0,1]$: The state transition function which maps each belief state action pair into a probability distribution over the belief state space. The state transition function can be calculated according to Equation 2.15.

$$\begin{aligned} \tau(b,a,b') = P(b'|a,b) = \textstyle\sum_{z \in Z} P(b'|a,b,z)P(z|a,b), \\ where\ P(b'|b,a,z) = \begin{cases} 1\ if\ SE(b,a,z) = b' \\ \quad 0\ otherwise \end{cases} \end{aligned} \tag{2.15}$$

- $\rho : B \times A \rightarrow \Re$: The immediate reward function on belief states. The reward function can be calculated as shown in equation 2.16.

$$\rho(b,a) = \sum_{s \in S} b(s)R(s,a) \tag{2.16}$$

Since we can formulate the POMDP problem as an MDP whose states are the belief states of the POMDP, we can solve it using value iteration (see Figure 2.2). That is, determine the optimal policy by iteratively constructing each optimal t-step discounted value function over belief space. Unfortunately, the number of belief states is infinite and iteration over belief states is intractable. Nevertheless, the optimal value function for any finite-horizon POMDP exhibits certain characteristics, which alleviate the computation of optimal policies. In particular, the optimal value function for any finite-horizon POMDP is always piecewise-linear and convex over the belief space [34], [35]. This is not necessarily true for the infinite-horizon discounted value function which remains convex but may have infinitely many facets [36]. Still the optimal infinite-horizon discounted value function can be approximated arbitrarily well by a piecewise-linear and convex value function, or in other words with a finite-horizon value function with sufficiently long horizon [37]. This property of the value function allows for compact representations as well as for the development of algorithms that are able to calculate the optimal t-step discounted value function from the optimal t-1 step discounted value function.

To see how we can compactly represent the optimal finite-horizon value function of POMDPs, we will review the concept of *policy-trees* shown in Figure 2.5 [38]. In a policy-tree the root node represents the action to be taken when there are t steps to go. Depending on the observation made after the action has been taken, the agent chooses the next action with $t-1$ steps to go. If the agent is at a leaf of a tree then there is only one thing to do, take the last action. To calculate the expected discounted value of executing a non-stationary policy for t steps or equivalently the value of executing a policy tree p from a belief state b, we can re-express Equation 2.3 with Equation 2.17.

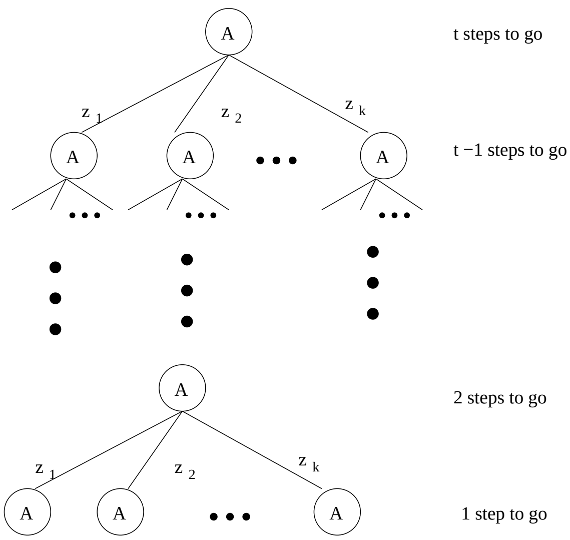

Figure 2.5: The policy tree indicates the non stationary t-step policy to be executed for a POMDP. First the action a defined by the root is executed and then according to the observation z_i perceived the $t-1$ steps to go tree is chosen.

$$\begin{aligned} V_{p,t}(b) &= \rho(b, a(p)) + \gamma \sum_{b' \in B} \tau(b, a(p), b') V_{p,t-1}(b') \\ &= \rho(b, a(p)) + \gamma \sum_{b' \in B} \sum_{z \in Z} P(b'|a(p), b, z) P(z|a(p), b) V_{z(p),t-1}(b') \end{aligned} \tag{2.17}$$

Where $V_{z(p),t-1}$ is the $t-1$ step tree to go that was entered with observation z from its parent $V_{p,t}$. Since the term $P(b'|a(p), b, z)$ (see Equation 2.15) is deterministic Equation 2.17 can be re-expressed as Equation 2.18.

$$V_{p,t}(b) = \rho(b, a(p)) + \gamma \sum_{z \in Z} P(z|a(p), b) V_{z(p),t-1}(SE(b, p(a), z)) \tag{2.18}$$

The term $P(z|a(p), b)$ is the normalizing factor in the belief state estimation in Equation 2.14. Therefore Equation 2.18 can be expressed with Equation 2.19.

$$
\begin{aligned}
V_{p,t}(b) &= \sum_{s\in S} b(s)R(s,a(p)) + \gamma \sum_{z\in Z}\sum_{s'\in S}\sum_{s\in S} b(s)T(s,a(p),s')O(p(a),s',z)V_{z(p),t-1}(s') \\
&= \sum_{s\in S} b(s)\left(R(s,a(p)) + \gamma \sum_{z\in Z}\sum_{s'\in S} T(s,a(p),s')O(p(a),s',z)V_{z(p),t-1}(s')\right) \\
&= \sum_{s\in S} b(s)V_{p,t}(s)
\end{aligned} \tag{2.19}
$$

Where $V_{p,t}(s)$ is the value of executing a policy tree p of depth t starting at state s. The interesting part of Equation 2.19 is that it tells us that the value function of every policy tree $V_{p,t}$ is linear in the belief space. If we let $\alpha_{t,p} = \langle V_{p,t}(s_1), ..., V_{p,t}(s_n)\rangle$, then Equation 2.19 becomes Equation 2.20.

$$
V_{p,t}(b) = b \cdot \alpha_{p,t} \tag{2.20}
$$

If P is the finite set of all possible policy trees of depth t, then the optimal t-step value for a belief state b is the value for executing the best policy tree in that belief state, shown in Equation 2.21.

$$
V_t^*(b) = max_{p\in P}(b \cdot \alpha_{p,t}) \tag{2.21}
$$

Each policy tree p induces a value function $V_{p,t}$ that is linear in b, and V_t^* is the upper surface of this collection of functions, which makes the optimal t-step value function piecewise-linear and convex. The convexity of the value function is intuitive since belief states in the middle of the belief space have high entropy and as a result the agent cannot select actions very appropriately and so tends to gain less long-term reward. An example with a two state POMDP s_1 , s_2 is shown in Figure 2.6 where $b(s_1)$ is enough to describe the belief space.

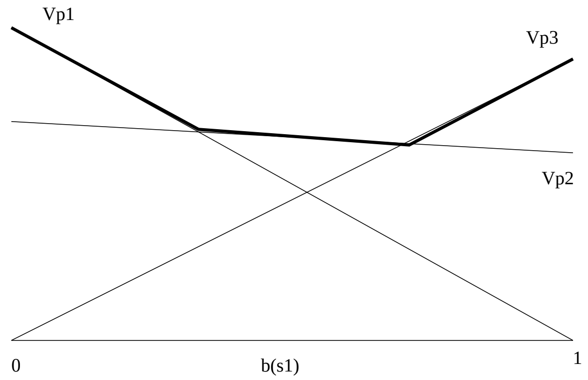

Figure 2.6: The figure shows an example of an optimal t-step value function which is the upper surface of the value functions associated with all t-step policy trees.

If we are given a piecewise-linear and convex value function, then we can calculate the best action to do by projecting the optimal value function onto the belief space. An example is shown in Figure 2.7.

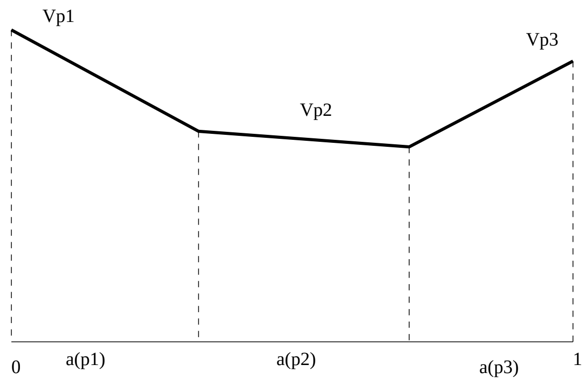

Figure 2.7: The figure shows a mapping from the belief space to the appropriate action defined by the root of a t-step policy tree. The mapping is calculated by projecting the optimal value function onto the belief space.

Now in order to calculate V_t^* from V_{t-1}^* (which is what we want to achieve in a value iteration algorithm) really means to get the set of $\alpha_{p,t}$ vectors from the set of $\alpha_{p,t-1}$ vectors. Nevertheless, the complexity of computing the new set of α vectors grows exponentially in the number of observations $|Z|$. In particular, given the current set of α vectors V_{t-1}, we need to compute $|A||V_{t-1}|^{|Z|}$ α vectors to represent the value function V_t. The reason is that we need to take into consideration all possible lists (of alpha vectors in V_{t-1}) of length $|O|$. Solving POMDP problems, is intrinsically hard. In fact, computing the optimal policy for the finite-horizon is PSPACE-complete [7]

A few algorithms have been developed for solving this problem, [35], [39], [40] [41], [38]. These algorithms however, are only able to solve small state POMDP problems, and even worse, they are based on the the fact that the value function is piecewise-linear and convex which is not necessarily true in the infinite-horizon case [34]. For the infinite-horizon case finding the optimal policy of a given POMDP is actually undecidable [8], and thus may not even be computable.

To avoid the computational intractability in computing the exact value function for a given POMDP, researchers have developed various approximations [42]. Approximate algorithms have been developed both for approximating the optimal value function in belief-state MDPs [43], [44] as well as algorithms for approximating the optimal policy directly [45], [46], [47], [48]. In realistic domains such as robot navigation which require thousands of states exact algorithms are impossible. Instead we have to rely on approximate algorithms for solving POMDPs such as the algorithms that will be presented in Chapter 6.

Finally, we should note that alternatives to POMDPs, in terms of computing belief states, are Kalman filters [49], [50] and temporal Bayesian networks [51]. Kalman filters have been a great contribution into the world history, in that they have enabled the mid-course correction scheme for the Apollo missions to the moon [52]. With

Kalman filters however only restricted sets of probability distributions can be modeled such as Gaussian. Gaussians are unimodal distributions which are not a good representation for robot navigation beliefs which in many situations are multimodal. This limitation is overcome with multi-hypothesis Kalman filters. Multiple-hypothesis Kalman filters represent beliefs using mixtures of Gaussians [53], [54]. Nonetheless, this approach still inherits the Gaussian noise assumption. Temporal Bayesian networks can also model arbitrary probability distributions. Unfortunately the model size of temporal Bayesian nets grows linearly with the temporal look-ahead and this limits their usefulness in planning [55].

2.2.2 Learning POMDPs

POMDPs are basically HMMs with the addition of actions and rewards. HHMMs can be leaned from data through an Expectation Maximization procedure. The learning algorithm is called Baum-Welch and can be derived from the general theory of EM algorithms [9] [56] [57] .

In particular, the EM algorithm is well suited for two types of applications. The first occurs in situations where optimizing the likelihood function is analytically intractable but when the likelihood function can be simplified by assuming the existence of, and values for additional but missing parameters. One such example is the mixture-density parameter estimation [58], [59], [60]. The second occurs when the data has indeed missing values, due to problems with or limitations of the observation process, such as training of hidden Markov models [9], [56], [61]. The second application is what we are interested in, since we need to learn the parameters of a POMDP model based on observation data which do not directly reveal the underling hidden state of the environment. Next, we give a brief derivation of the EM algorithm.

For analytical convenience the likelihood function is expressed as the log likelihood function $L(\Theta) = \ln P(X|\Theta)$, where X is the training data and Θ the set of parameters. The EM algorithm recursively estimates a new set of parameters Θ_{i+1} such that $L(\Theta_{i+1}) \geq L(\Theta_i)$, where Θ_i is the current set of parameters. The difference between the log likelihood of a any new set of parameters Θ and the current set Θ_i is expressed in Equation 2.22.

$$L(\Theta) - L(\Theta_i) = \ln P(X|\Theta) - \ln P(X|\Theta_i) = \ln \frac{P(X|\Theta)}{P(X|\Theta_i)} \tag{2.22}$$

We would like to choose the new parameters Θ such that the right-hand side of Equation 2.22 is maximized. In general, maximization of the right-hand site may be intractable. The idea of the EM algorithm is to introduce a set of hidden variables Z, such that if Z where known, computation of the optimal value of Θ becomes tractable. Equation 2.22 then, is transformed though conditioning into equation 2.23.

$$L(\Theta) - L(\Theta_i) = \ln \frac{\sum_z P(X|z,\Theta)P(z|\Theta)}{P(X|\Theta_i)} \tag{2.23}$$

Unfortunately, in the above expression the logarithm of the sum is not easy to deal with. Fortunately due to Jensen's inequality (Equation 2.24), we can can replace the logarithm of a sum with the sum of logarithms.

$$\ln \sum_j \lambda_j y_j \geq \sum_j \lambda_j \ln y_j \text{ if } \sum_j \lambda_j = 1 \tag{2.24}$$

In our expression we introduce the term $\lambda_z = P(z|X, \Theta_i)$ which transforms Equation 2.23 into Equation 2.25.

$$\begin{aligned} L(\Theta) - L(\Theta_i) &= \ln \frac{\sum_z P(X|z,\Theta)P(z|\Theta)}{P(X|\Theta_i)} \frac{P(z|X,\Theta_i)}{P(z|X,\Theta_i)} \\ &\geq \sum_z P(z|X,\Theta_i) \ln \frac{P(X|z,\Theta)P(z|\Theta)}{P(X|\Theta_i)P(z|X,\Theta_i)} \end{aligned} \tag{2.25}$$

Equation 2.25 can be expressed as Equation 2.26.

$$L(\Theta) \geq L(\Theta_i) + \sum_z P(z|X,\Theta_i) \ln \frac{P(X|z,\Theta)P(z|\Theta)}{P(X|\Theta_i)P(z|X,\Theta_i)} = l(\Theta) \tag{2.26}$$

Now we need to find the parameters Θ_{i+1} that maximize the right hand side of Equation 2.26 as shown in Equation 2.27.

$$\begin{aligned} \Theta_{i+1} &= \text{argmax}_\Theta L(\Theta_i) + \sum_z P(z|X,\Theta_i) \ln \frac{P(X|z,\Theta)P(z|\Theta)}{P(X|\Theta_i)P(z|X,\Theta_i)} \\ &= \text{argmax}_\Theta \sum_z P(z|X,\Theta_i) \ln P(X,z|\Theta) \end{aligned} \tag{2.27}$$

In short, Equation 2.27 is the EM algorithm. The E-step is the estimation of the expectation (expectation of $\ln P(X, z|\Theta)$ with respect to the hidden variables) and the M-step is the maximization. At each iteration Θ_{i+1} maximizes $l(\Theta)$ and therefore $l(\Theta_{i+1}) \geq l(\Theta_i)$. Since Ł$(\Theta_i) = l(\Theta_i)$ (from Equation 2.26) and the fact that for any Θ $L(\Theta) \geq l(\Theta)$, the new log likelihood $L(\Theta_{i+1})$ cannot decrease, and therefore

$L(\Theta_{i+1}) \geq L(\Theta_i)$.

In cases where the complete maximization of the expected log likelihood is intractable we resolve to partial M-step implementations. Such implementations simply improve $l(\Theta)$ using a gradient decent method rather than fully maximizing it. Such algorithms are also guaranteed to increase $L(\Theta)$ and are called Generalized EM (GEM) algorithms [12]. Next we present an adaptation of the EM algorithm, the Baum-Welch algorithm, which learns the parameters of an HMM model.

An HMM model is defined formally as a five tuple $\langle S, T, Z, O, \pi \rangle$:

- S: A set of states.
- $T : S \times S \rightarrow [0, 1]$: A function that maps every state to a distribution over the state set. We write $T(s_1, s_2)$ to indicate the probability of going from state s_1 to state s_2.
- Z: A set of observations.
- $O : S \times Z \rightarrow [0, 1]$: A function that maps every state to a distribution over the observation set. We write $O(s, z)$ for the probability of observation z in state s.
- $\pi : \Pi(S)$: The initial probability distribution.

When we say we learn an HMM model we mean that given an observation sequence $M = z_1, z_2 ... z_T$, and the model parameters $\lambda = \langle T, O, \pi \rangle$, we must adjust the model parameters such that the log likelihood of the observation sequence is maximized. That is, maximize $\log P(M|\lambda)$. In order to develop the algorithm, we first need to define some variables. The first variable is called the forward variable α and is defined in Equation 2.28. The α variable defines the probability of an observation sequence $z_1, ...z_t$ and that the state at time t is i given the model parameters λ.

$$\alpha_t(i) = P(z_1, z_2...z_t, s_t = i|\lambda) \tag{2.28}$$

If we know the α variable then it easy to calculate how well the model fits a sequence of observations M of length T as shown in Equation 2.29

$$P(M|\lambda) = \sum_{s \in S} \alpha_T(s) \tag{2.29}$$

The α variable can easily be calculated recursively. If $t = 1$ then

$$\alpha_1(i) = \pi(i)O(i, z_1)$$

else

$$\alpha_{t+1}(j) = \sum_{i=1}^{S} \alpha_t(i)T(i, j)O(j, z_{t+1})$$

Another variable is the backward variable which defines the probability of an observation sequence $z_{t+1}, ...z_T$ given that the state at time t was i (Equation 2.30).

$$\beta_t(i) = P(z_{t+1}z_{t+2}...z_T|s_t = i, \lambda) \tag{2.30}$$

The backward-variable can also be calculated recursively as follows: If $t = T$ then

$$\beta_T(i) = 1$$

else

$$\beta_t(j) = \sum_{j=1}^{S} T(i,j)O(j, z_{t+1})\beta_{t+1(j)}$$

Note that if we know the β variable, we can rewrite equation Equation 2.29 with Equation 2.31

$$P(M|\lambda) = \sum_{s\in S} \alpha_t(s)\beta_t(s) \tag{2.31}$$

The next variable to be defined is the γ variable which gives the probability that at time t the process was in state i (Equation 2.32)

$$\gamma_t(i) = P(s_t = i|M, \lambda) \tag{2.32}$$

The γ variable can be calculated from the α and β variables as follows:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(M|\lambda)} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{S} \alpha_t(i)\beta_t(i)}$$

The last variable to be defined is the ξ variable which is shown in equation 2.33

$$\xi_t(i,j) = p(s_t = i, s_{t+1} = j, M|\lambda) \tag{2.33}$$

which can also be calculated from the α and β variable as follows:

$$\xi_t(i,j) = \frac{\alpha_t(i)T(i,j)O(j, z_{t+1})\beta_{t+1}(j)}{P(M|\lambda)}$$

Using the calculated variables defined above we re-estimate the transition model as shown in Equation 2.34, the observation model as shown in Equation 2.35, and the initial distribution as shown in 2.36

$$T(ij) = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{2.34}$$

$$O(j,k) = \frac{\sum_{t=1|z_t=k}^{T} \gamma_t(i)}{\sum_{t=1}^{T} \gamma_t(i)} \tag{2.35}$$

$$\pi[i] = \gamma_1(i) \tag{2.36}$$

It has been proven that each time the model parameters get re-estimated that: $P(M|\lambda)_{new} \geq P(M|\lambda)_{previous}$. Even though the maximization does not necessarily converge to the best optimal solution, it converges to local maxima [56]. This limitation requires that a good initial model is provided to the learning algorithm as well as "adequate" training data.

In [11] an extension of the Baum-Welch algorithm for HMMs [10] is described, which learns POMDPs, and we describe next. In this version there is also a scaling factor, which allows the algorithm to run for long training sequences without fear of exceeding the precision range of any machine.

1. First we compute the scaling factor and the α variable for the first observation:

$$scale_1 = \sum_{s \in S} [P(z_1|s)P(s_1 = s)]$$

$$\alpha_1(s) = p(z_1|s)P(s_1 = s)/scale_1 \ \forall s \in S$$

2. Next we compute the forward variable and the scaling factor for all $t = 1$ to $T-1$ in a procedure called "forward propagation":

$$temp(s) = \sum_{s' \in S} [P(s|s', a_t)\alpha_t(s')]P(z_{t+1}|s) \ \forall s \in S$$

$$scale_{t+1} = \sum_{s \in S} temp(s)$$

$$\alpha_{t+1}(s) = temp(s)/scale_{t+1} \forall s \in S$$

3. Next we initialize the backward variable β for time T:

$$\beta_T(s) = 1/scale_T \ \forall s \in S$$

4. Then we estimate the backward variable starting from time $t = T-1$ down to $t = 1$ in a procedure called "backward propagation":

$$\beta_t(s) = \sum_{s' \in S} [P(s'|s, a_t)P(z_{t+1}|s')\beta_{t+1}(s')]/scale_t \ \forall s \in S$$

5. After the estimation of the α and β variables we can now compute the γ variable:

$$\gamma_t(s, s') = \alpha_t(s)P(s'|s, a_t)P(z_{t+1}|s')\beta_{t+1}(s') \ \forall s, s' \in S, \ for\ t = 1\ to\ T-1$$

$$\gamma_t(s) = scale_t\alpha_t(s)\beta_t(s) \ \forall s \in S, \ for\ t = 1\ to\ T$$

6. Using the αs, βs, and γs we can now re-estimate the initial probability distribution, the transition matrices, and the observation models:

$$\pi(s) = \gamma_1(s) \; \forall s \in S$$

$$P(s'|s,a) = \sum_{t=1...T-1|a_t=a} \xi_t(s,s') \sum_{t=1...T-1|a_t=a} \gamma_t(s) \; \forall s, s' \in S \; and \; \forall a \in A$$

$$P(z|s) = \sum_{t=1...T|z_t=z} \gamma_t(s) / \sum_{t=1...T} \gamma_t(s) \; \forall s \in S \; and \; \forall z \in Z$$

2.3 Robot navigation

Robot navigation is a broad topic, covering a large spectrum of different technologies and applications [62]. It draws on some very ancient techniques, as well as some of the most advanced space science and engineering. The general problem of robot navigation can be summarized by three questions: "Where am I?", "Where am I going?", and "How should I get there?", according to [63]. In general, we can classify robot navigation technologies in to two categories: *relative* and *absolute* position measurements [62]. Relative position methods include Dead-reckoning systems such as systems based on odometric information. Absolute position systems include active beacon navigation, landmark-based, and map-based navigation.

The dead-reckoning method (derived from "deduced reckoning" from sailing) is a simple mathematical procedure for determining the present location of a vehicle by advancing some previous position through known course and velocity information over a given length of time. The simplest form of dead-reckoning is often termed as odometry. Odometry-based systems use encoders to measure wheel rotation and steering orientation. The advantage of odometry is that it is self contained (no external feedback from the environment is needed). The disadvantage, is that the position error grows without bounds. The unbounded position error is due to unequal wheel diameters, misalignment of wheels, travel over uneven floors, travel over unexpected obstacles, and wheel slippage. Due to the increasing error, odometric systems are rarely used by them selves but rather are combined with absolute positioning systems [64].

Active beacon based navigation is an ancient technique that ranges from star following to the Global Positioning System (GPS) [65]. Active beacons systems compute the absolute position of the robot, by measuring the direction of incidence of three or more actively transmitted beacons mounted at known locations in the environment. There are two principle methods for determining the vehicle position: *triangulation*, and *trilateration*. Triangulation measures the angles between the vehicles heading and a number of beacons, and uses them to estimate its absolute position. Trilateration uses a measurement of distance between a number of beacons and the vehicle [62].

In landmark-based navigation, the robot computes its absolute position by recognizing distinct features (landmarks) from its sensory input. If the position of landmarks in the environment is known in advance then the robot will be able to localize correctly. Landmark-based approaches can incorporate topological maps which represent the location of the each landmark in the environment (see Figure 2.8). Or, landmark based navigation could simply be a reactive systems where each landmark defines the next robot direction or action [66], [67]. The advantage of having a topological map is that a robot can use it to construct multiple navigation tasks (planning). On the other hand, a topological map may be too general, because it represents multiple navigation tasks. Thus, a separate representation (direct mapping from landmarks to actions) for each navigation task at hand, may be more efficient for that particular task. Nonetheless, it is costly to construct, or learn a separate representation for every possible task.

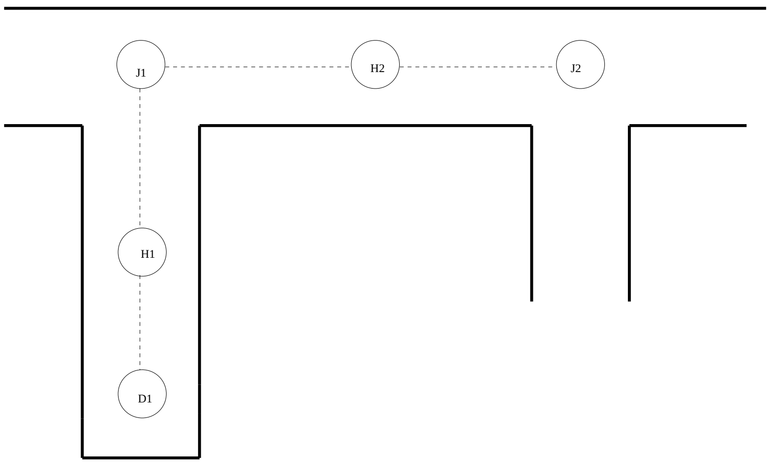

Figure 2.8: The figure show a landmark based topological map representing a corridor environment. Each node in the topology represent a unique landmark that can be identified by the robot. Each node also contains information as to the distance and orientation to the next node.

In general, landmark-based navigation systems are difficult to construct and maintain in large environments for complicated navigation tasks. The reason is that unique landmarks and their relationships need to be discovered. Also, recognition of landmarks may be ambiguous and sensitive to the point of view. If a robot cannot uniquely identify landmarks then it would be executing inefficient navigation strategies that may not even be able to get it to its goal (if two places in the environment are recognized as the same landmark and yet they require a different action). In practice researchers implement hybrid systems where odometry is used to disambiguate landmarks which are perceived the same.

In map-based systems the information acquired from the robot's onboard sensors is compared to a map or world model of the environment. If features from the sensor-based map and the world model map match, then the vehicle's absolute location can be estimated. Map-based positioning often includes improving global maps based on the new sensory observations in a dynamic environment, and integrating local maps into the global map to cover previously unexplored areas. One type of map is the topological map used in landmark-based navigation. A classic example of landmark-based navigation using topological map is presented in [18].

The other type of map is the geometric map. Unlike topological maps, which represent the relationships between landmarks, the geometric maps represent the world in a global coordinate system. A classic example of geometric map navigation is presented in [1] where the environment is represented with a uniform fine grain resolution which can be as high as 1 square *cm*.. Figure 2.9 shows an example occupancy grid map of an office environment taken from [1]. Each grid cell of the occupancy map contains an occupancy value (probability of obstacle). Occupancy values are calculated from multiple sonar sensors which are integrated over time [68]. To build such a map however, a robot needs to know its exact position in the environment. This is achieved by combining various approaches such as odometry, the fact that the robot operates in an office environment (90 degree angles), and by mapping sensor readings to the current global grid map (map correlation).

Nonetheless, it is difficult to construct such maps for cyclic environments where the odometric error renders them unusable for robot navigation. In such situations a better approach is the concurrent mapping and localization method [69], where an EM type algorithm is used to improve the occupancy grid values. The advantage of the EM algorithm is that it does not rely on exact odometric information as to where landmark observations are generated, but rather estimates the probability that a landmark was generated at a particular environment location. Given a sequence of landmarks and motion control signals the E-step estimates the robot position, and the M-step reestimates the occupancy grid values. This approach has also been extended to on-line learning of spatial environments [70], and recently to learning of 3D maps of spatial environments [71].

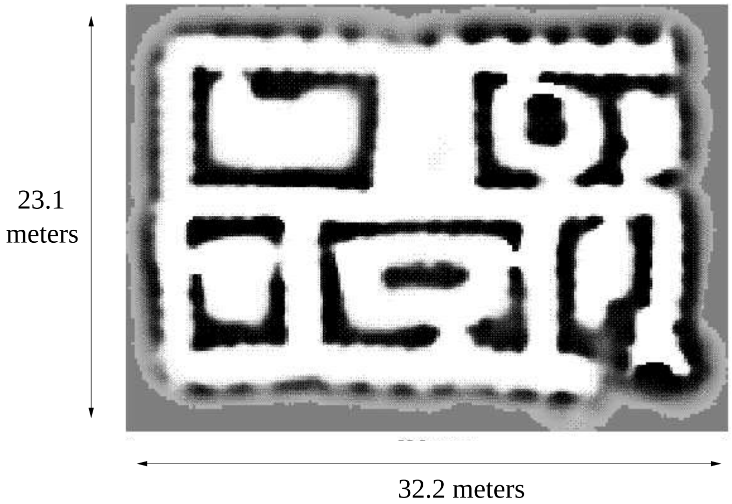

Figure 2.9: The figure shows an occupancy grid map that models an indoor office environment and was taken from [1].

Even though it is easy to build and maintain large grid maps which are also view point independent, planning is inefficient due to the fine-grain resolution, and unecessarily detailed representation. Nonetheless, the fine-grain representation has the advantage of exact robot localization, and facilitates the disambiguation of different places. However, the large number of states imposes significant computational burden, even for the simple procedure of belief state update. To overcome these limitations, researchers have proposed selective updating algorithms [72], tree-based representations that dynamically change their resolution, and Monte Carlo Localization (MCL) methods [73]. Monte carlo localization simply represents the posteriors or the belief state by a random collection of weighted particles which approximates the desired distribution. In the literature this is referred to as the Sampling Importance Resampling (SIR) algorithm [74]. The monte carlo localization approach presented in [73] called Mixture-MCL, is considered to be the state of the art in robot localization solutions, and has been shown to recover gracefully from global localization failures. It is able to recover from localization failures generated from the *kidnapped robot problem* [75], where a well localized robot is picked up and transferred to some random location.

2.4 Application of POMDPs to robot navigation

An autonomous mobile robot that has to navigate in an indoor office environment needs to be able to know its location at each point in time so that it can navigate successfully. Unfortunately indoor office environments are perceptually aliased, which means that different places in the environment are perceived to be the same. Even if the robot had perfect sensors, which is usually not the case, it will not always be able to distinguish where it is in its environment simply by perceiving its surroundings. One way to know exactly where the robot is in the environment is to keep track of odometry, unfortunately this also requires noise-free dead-reckoning and knowledge of the starting position of the robot. One solution then, to the above problems is to model the navigation task as a partially observable Markov decision process [23], [22]. The POMDP framework can be viewed as a method between landmark-based and odometry-based based navigation in that it seamlesly integrates both odometric and sensor reports. In addition, the integration is done in a probabilistic manner which takes into consideration the noisy sensors, and the uncertainty involved with dead-reckoning.

In the POMDP framework for robot navigation, for each fixed region of the environment (e.g., for each square meter) there are four states (one for each robot orientation) modeling the position and orientation of the robot. The robot can take actions such as “go-forward”, “turn-left”, and “tun-right”. In each state it can observe features such as wall, and opening on the front, left, back and right side. Modeling a real office environment in this manner requires thousands of states and non of the exact algorithms can solve such a problem. A simple heuristic solution is to assume that the states of the POMDP are completely observable, and solve it using the MDP framework. Then, one POMDP execution strategy is to choose the action for each belief states that corresponds to the state, for which the belief state is peaked. Other strategies that use the MDP solution over the hidden states are described in [22], [76].

An important issue is how to learn the POMDP model. The obvious way is to use the standard Baum-Welch algorithm, which requires a “good” initial model. In [2] an initial model is created from a topological map which is later fine-tuned with an extension of the Baum-Welch algorithm called GROW-BW. The GROW-BW algorithm examines multiple Markov chains along each hallway and adds longer ones as needed until it finds the best length that fits the data the best. Figure 2.10 shows a POMDP model with multiple chains for each hallway.

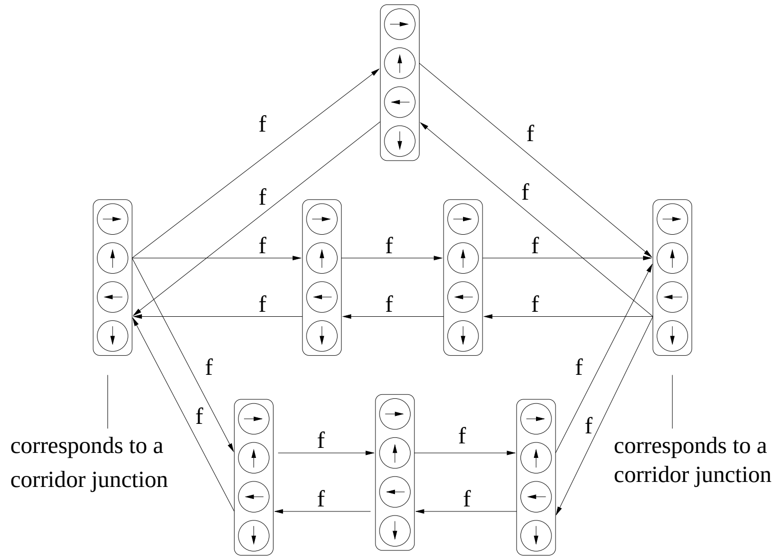

Figure 2.10: The figure shows a POMDP model that has multiple chains for each hallway. The GROW-BW algorithm adds longer chains as needed so as to find the length that fits the data the best. The figure was taken from [2].

Another way of learning POMDP models for robot navigation is the algorithm described in [77], and [78]. This algorithm also extends the Baum-Welch algorithm by adding the concept of *relation-matrices*. The relation-matrices store the means and variances of the distances between states. These matrices are then incorporated in the calculation of the α and β variables in the standard Baum-Welch algorithm. Moreover, odometric information collected with each observation, is used to do an initial clustering of the observations into states, and therefore producing some initial model.

Both of the above approaches use flat representations where they model the environment with uniform resolution. Since flat POMDP representations do not scale well in large scale environments, both in learning and planning, in the rest of this dissertation we model robot navigation as a hierarchical POMDP. We present map learning experiments using a hierarchical EM algorithm as well as experiments using approximate hierarchical planning and execution algorithms.

Chapter 3

Hierarchical Partially Observable Markov Decision Processes

A hierarchical problem representation is not only inspired from biology but it potentially has many advantages from an engineering point of view. As mentioned in Section 1.4, biological organisms (from insects to humans) seem to solve every day sequential decision tasks in a hierarchical fashion. From an engineering point of view, hierarchical modeling has advantages in ease of maintenance, interpretability and reuseability. Reusability could be a great advantage in learning the model parameters in that sub-hierarchies of the model can be trained separately and then reused in the overall hierarchical model. In planning, the advantage is that at higher levels of abstraction (both spatial and temporal) uncertainty is less. Reduced uncertainty implies more efficient planning algorithms (as we shall see in Chapter 6).

In this chapter we first describe in Section 3.1 the hierarchical HMM model which has been the building block of our hierarchical approach. In Section 3.2 we introduce the hierarchical POMDP model which is an extension of the HHMM with the addition of actions and rewards. In Section 3.2.1 we describe in detail the algorithm for learning HPOMDPs. In Section 3.2.2 we present a small example to demonstrate the correctness of the learning algorithm. The chapter continues with Section 3.2.3 where we describe the relation of hierarchical POMDPs to flat POMDPs. Finally, the chapter concludes with a discussion of related multi-resolution hidden Markov models in Section 3.3.

3.1 Hierarchical Hidden Markov Models

The Hierarchical Hidden Markov Model (HHMM) [17] generalizes the standard hidden Markov model (HMM) [10] by allowing hidden states to represent stochastic processes themselves. An HHMM is visualized as a tree structure (see Figure 3.1 in which there are three types of states; product 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 which 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 of an internal state to 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 $|C^s|$. There are three types of states.
  - *Product states*
  - *Abstract states*
  - *End-states*

- $T^s : \{C^s - e^s\} \times C^s \to [0, 1]$ 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 3.1, $T^{s4}(s7, s8) = 0.6$.

- $\Pi^s : \{C^s - e^s\} \to [0, 1]$ 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 3.1, $\Pi^{s1}(s2) = 0.5$.

- Z denotes the set of discrete observations.

- $O^s : C^{s^{product}} \rightarrow [0,1]$ denotes a function that maps every product 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. $C^{s^{product}}$ is the set of all product states which are children of s.

Figure 3.1 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 product 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.

Fine et al. [17] describe a hierarchical Baum-Welch algorithm that is able to re-estimate the model parameters (including transitions matrices, vertical vectors, and observation vectors) of an HHMM from observation sequences. The hierarchical Baum-Welch algorithm operates in a recursive fashion similar to the standard Baum-Welch procedure. In the next section we describe the extension of the HHMM into a decision model, the hierarchical POMDP.

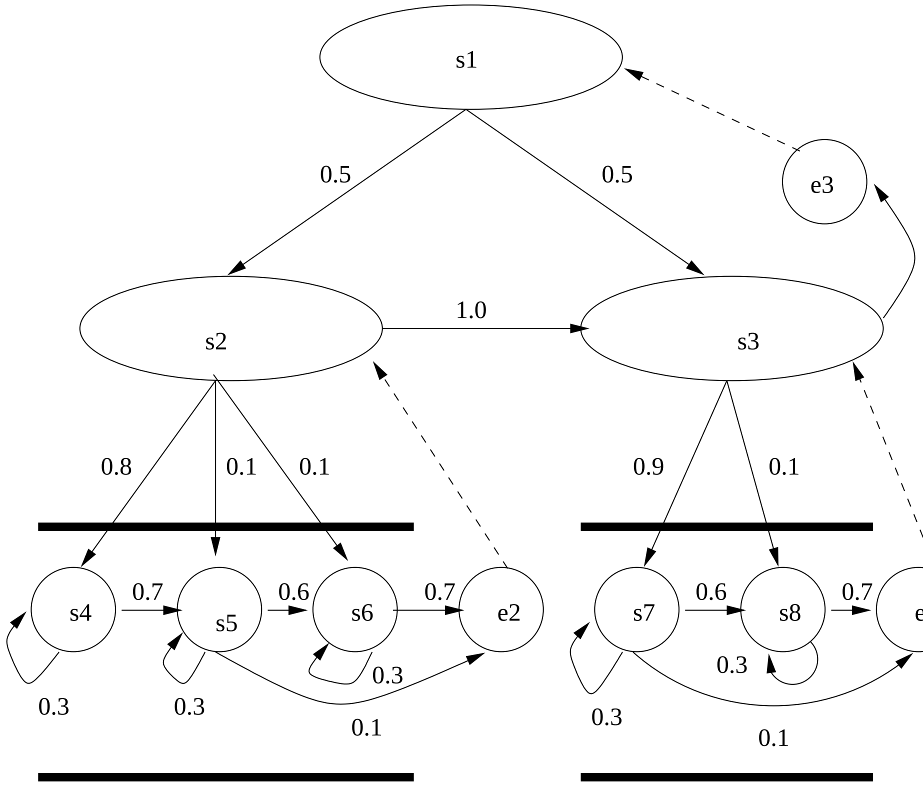

Figure 3.1: An example hierarchical HMM modeling two adjacent corridors. Only leaves (production) states (s4, s5, s6, s7, and s8) have associated observations. The end-states are e1, e2, and e3

3.2 Hierarchical Partially Observable Markov Decision Process Models

Robot navigation is an example of a planning problem that requires extending the HHMM model to include actions, as well as reward functions for specifying goals. We call this extended model *Hierarchical Partially Observable Markov Decision Process* model [79]. Unlike HHMMs, we extend the definition to include multiple entry points into abstract states which we call *entry-states*, and multiple *exit-states* (defined as end-states in HHMMs). Figure 3.2 shows a sample HPOMDP with multiple entry/exit states for corridor environments. The exit and entry states represent the spatial borders of each (corridor) abstract state. If we only had single exit and entry states, the consequences of primitive actions would not be modeled correctly. For example, an exit from the east side of a corridor would have a non-zero transition probability to an adjacent state of the corridor at the west side.

On the other hand, we could have two abstract states representing each corridor (one for each direction). However, having two abstract states for the same spatial locations would mean that the abstract states share children. Even though the HHMM allows abstract states to share children, each abstract state induces its own set of parameters on the children (vertical vectors, horizontal transitions, and observation models). This would imply learning model parameters for each abstract state separately, which is redundant. Since we are interested in learning the dynamics of the world (e.g., effect of actions on states), it would be redundant to learn them more than once (e.g., once when going to the east, and once when going to the west side of the corridor).

In effect, since abstract states do not share parameters the representation becomes equivalent to having each abstract state having its own exclusive set of children. If we allowed abstract states to share parameters then the spatial abstraction is no longer obvious and the model abstraction tends to become more temporal and task specific. For example, if two abstract states shared the same sub-structure (states and parameters) then we would not be able to distinguish the two abstract states based on sequences of observations emitted by the substructure. For example, in robot navigation, it is important that the robot knows its true location or true abstract state that it is at, so that it can take the appropriate action. If the appropriate action for each of the abstract states is different, then the robot will not know which one to take. However, if both abstract states require the same action then sharing sub-structures may be possible. In our HPOMDP definition we assume that abstract states do not share any parameters or any children.

In addition to the above extensions, we have also removed the end-state associated with the root abstract state. Instead, we consider the end of an episode of a sequence observations and actions to be the last child state of the root which has produced an observation (if it is a product state) or the last child state of the root exited (if it an abstract state). By removing the end-state we allow the model to be trained using training sequence that can terminate at any child of the root. This is justified, since we need to use our model for planning where we need to construct arbitrary plans, which can terminate anywhere, and therefore produce arbitrary training data as well. We formally describe the HPOMDP model below:

- S denotes the set of states. Unlike HHMMs, we have an additional type of state called entry states.
  - *Product states* which produce observations.
  - *Exit-states* which return control to their parent abstract state when entered.
  - *Entry-states* which belong to abstract states and when entered activate the children of the abstract state they are associated with.
  - *Abstract states* which group together *product*, *entry*, *exit*, and other *abstract* states. C_p^s denotes the product children of abstract state s. C_a^s denotes the

abstract state children of abstract state s. C^s_x denotes the set of exit states which belong to children abstract states of abstract state s. C^s_n 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 product states.

- $T(s'|s_x, a)$ denotes the horizontal transition probabilities for primitive actions. If s is an abstract state, s_x denotes the current exit state. If s is a product state then s_x refers to the product state itself since there are no exit states for product states. The resulting state s' could be the entry point of some abstract state, an exit state associated with the parent of s, or some product state.

- $V(s'_n|s_n)$ denotes the probability that entry state n which belongs to abstract state s will activate child state s'. s'_n is an entry state of state s' (if s' is an abstract state), and can be the state itself if s' is a product state.

- Z denote the set of discrete observations.

- $O(z|s, a)$ denotes the probability of observation z in product state s after action a has been taken.

- $R(s, a)$ denotes an immediate reward function defined over the product states.

For the example HPOMDP in Figure 3.2, there are 16 distinct observations for each product state (the combination of wall and opening on the four sides of a state). The probability of each observation can be constructed from individual sensor models for each side of the state, which are shown in Table 3.1. Using the different sensor models for each side of the state we can construct a joint probability distribution that gives us the probability of each one of the possible 16 observations as described in Equation 3.1 (assuming independence of the observations, or independence of each side).

$$O(z_W, z_S, z_E, z_N|a, s) = P(z_W|s, a)P(z_S|s, a)P(z_E|s, a)P(z_N|s, a) \qquad (3.1)$$

	Orientation	W	W	S	S	E	E	N	N
State	Action	P(w)	P(o)	P(w)	P(o)	P(w)	P(o)	P(w)	P(o)
s_4	go-left	0.1	0.9	0.9	0.1	0.1	0.9	0.9	0.1
s_4	go-right	0.1	0.9	0.9	0.1	0.1	0.9	0.9	0.1
s_5	go-left	0.1	0.9	0.9	0.1	0.1	0.9	0.9	0.1
s_5	go-right	0.1	0.9	0.9	0.1	0.1	0.9	0.9	0.1
s_6	go-left	0.1	0.9	0.9	0.1	0.1	0.9	0.9	0.1
s_6	go-right	0.1	0.9	0.9	0.1	0.1	0.9	0.9	0.1
s_9	go-left	0.1	0.9	0.9	0.1	0.1	0.9	0.9	0.1
s_9	go-right	0.1	0.9	0.9	0.1	0.1	0.9	0.9	0.1
s_{10}	go-left	0.1	0.9	0.9	0.1	0.1	0.9	0.9	0.1
s_{10}	go-right	0.1	0.9	0.9	0.1	0.1	0.9	0.9	0.1

Table 3.1: The table describes the sensor models of the product states of the HPOMDP in Figure 3.2. For every product state and action there are four sensors, one for every side of the state (W,S,E,N). Each sensor model gives the probability of a wall $P(w|s, a)$ and opening $P(o|s, a)$ given the current state and last action taken.

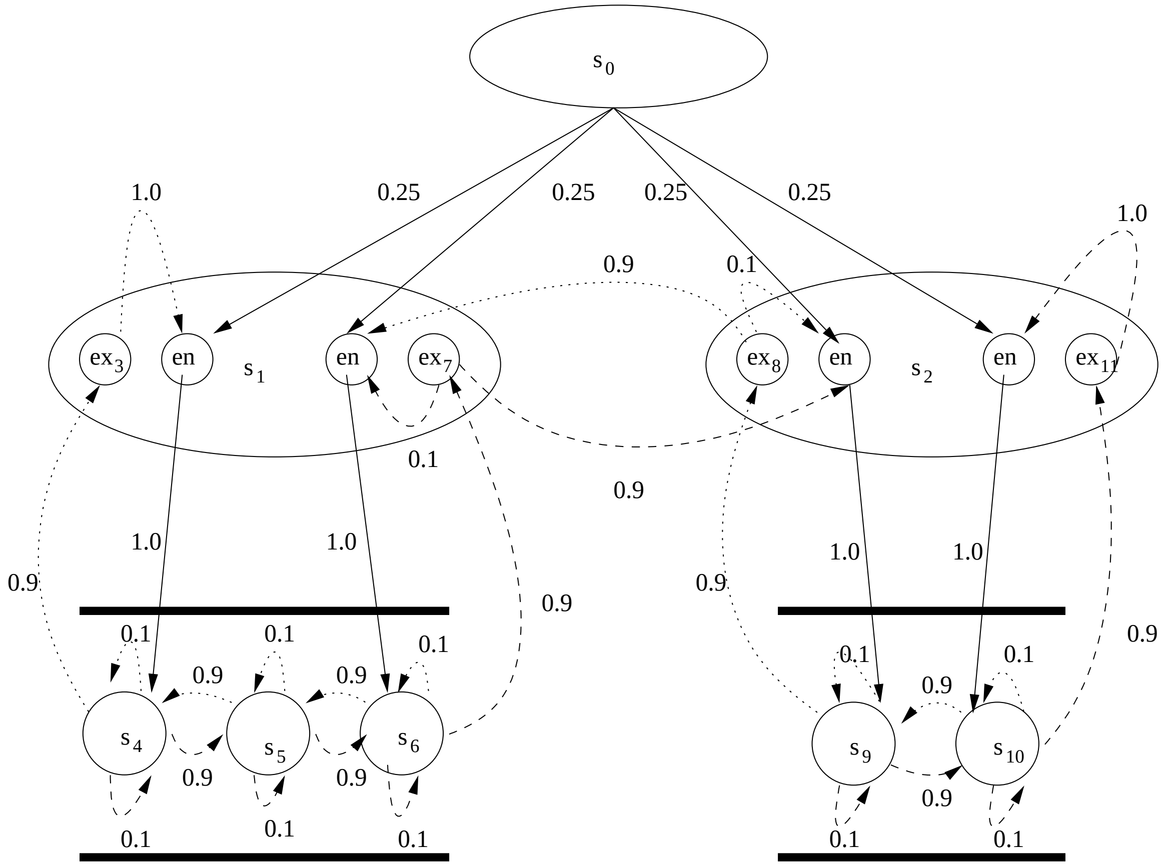

Figure 3.2: An example hierarchical POMDP with two primitive actions, "go-left" indicated with the dotted arrows and and "go-right" indicated with the dashed arrows. This HPOMDP has two abstract states s_1 and s_2 and each abstract state has two entry and two exit states. The product state s_4, s_5, s_6, s_9, and s_{10} are associated with sensor models which are described in Table 3.1.

3.2.1 Hierarchical Baum-Welch for HPOMDPs

A hierarchical Baum-Welch algorithm (shown in Figure 3.3) 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 which modifies the model parameters such that the log $P(Z|A, \lambda)$ is increased after very training epoch, where Z is a sequence of observations, A is a sequence of actions, and λ the current model parameters T,O, and V.

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 which have the same parent), and for every time index in the training sequence. The χ variable is an estimation of a vertical transition from every parent state and child, for every time index in the training sequence.

For the Maximization step, the Baum-Welch algorithm reestimates the model parameters by using the frequencies of occurrence of vertical and horizontal transitions (provided by ξ and χ). For example, to reestimate the horizontal transition probability between two states s and s' under an action a, the algorithm divides the number of times a transition occurred from s to s' under action a over the 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 α (similar to flat hidden Markov models) which is defined in Equation 3.2 and described in Figure 3.4.

$$
\begin{aligned}
&\alpha(p(s)_n, s_x, t, t+k) = \\
&P(z_t, ...z_{t+k},\ s \text{ exited from } s_x \text{ at } t+k \mid \\
&a_{t-1}, a_t...a_{t+k},\ p(s) \text{ started at } t \text{ from entry state } p(s)_n,\ \lambda)
\end{aligned}
\tag{3.2}
$$

We say that a product 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 product state s started at time t, if at time t it produced observation z_t. An abstract state s starts

Procedure HPOMDP-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 product states.
    - V: vertical transition vectors from entry states to children.
- Output
  - A new set of λ' parameters such that:

    $\log P(Z_1, ..., Z_k | A_1, ...A_k, \lambda') \geq \log P(Z_1, ..., Z_k | A_1, ..., A_k, \lambda)$

Figure 3.3: This is the Baum-Welch algorithm for learning the parameters of an HPOMDP model. The algorithm increases the log likelihood of the observation sequences given the actions sequences and current model parameters after every training epoch.

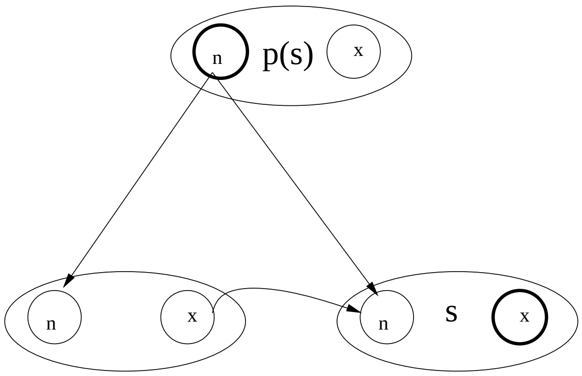

Figure 3.4: The α variables gives the probability of the observation sequence $z_t...z_{t+k}$ ending in state s_x at time $t+k$ (denoted as a bold circle), given that actions $a_t, ...a_{t+k}$ were taken and the parent of s, $p(s)$ was started at time t from entry state $p(s)_n$ (also denoted as bold circle).

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. 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 3.3 and described in Figure 3.5. 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 $t = 1$.

$$\alpha(p(s)_n, s, t, t) = V(s|p(s)_n)O(z_t|s, a_{t-1}) \tag{3.3}$$

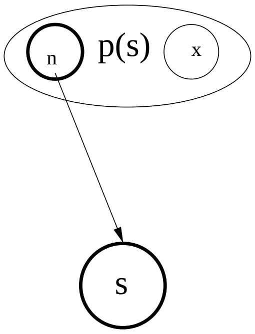

Figure 3.5: Equation 3.3 computes the probability that at time t the parent state of product state s, $p(s)$ produces a single observation z_t. The observation is produced by a vertical activation into product state s from entry state $p(s)_n$.

  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 3.4 and described in Figure 3.6. The equation computes the probability of an observation sequence that started at at time t from entry state $p(s_n)$ and finished at a child state s at time $t+k$. The computation is 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 $t = 1$.

$$\alpha(p(s)_n, s, t, t+k) = \left[\sum_{s'_x \in C_x^{p(s)} \cup C_p^{p(s)}} \alpha(p(s)_n, s'_x, t, t+k-1) T(s|s'_x, a_{t+k-1}) \right] O(z_{t+k}|s, a_{t+k-1}) \tag{3.4}$$

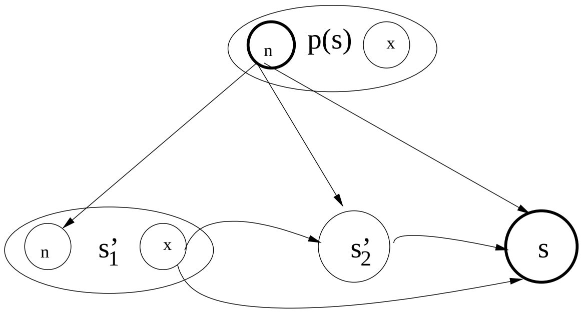

Figure 3.6: Equation 3.4 gives us the probability that the parent of the product state s, $p(s)$ produced observations from time t to time $t+k$, $(k \geq 1)$ and that the last observation was produced by product state s. s'_x is either the exit state of some abstract state s' (such as s'_{1_x}) or a 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 3.5 which can be better understood with Figure 3.7. When $p(s) = root$ we only use this equation for $t = 1$.

$$\begin{aligned}
&\alpha(p(s)_n, s_x, t, t+k) = \\
&\sum_{s_n \in N^s} V(s_n|p(s)_n) \left[\sum_{i_x \in C_x^s \cup C_p^s} \alpha(s_n, i_x, t, t+k) T(s_x, |i_x, a_{t+k}) \right] \\
&+ \left[\sum_{l=0}^{k-1} \left[\sum_{s'_x \in C_x^{p(s)} \cup C_p^{p(s)}} \alpha(p(s')_n, s'_x, t, t+l) T(s_n|s'_x, a_{t+l}) \right] \right. \\
&\left. \left[\sum_{i_x \in C_x^s \cup C_p^s} \alpha(s_n, i_x, t+l+1, t+k) T(s_x, |i_x, a_{t+k}) \right] \right]
\end{aligned} \tag{3.5}$$

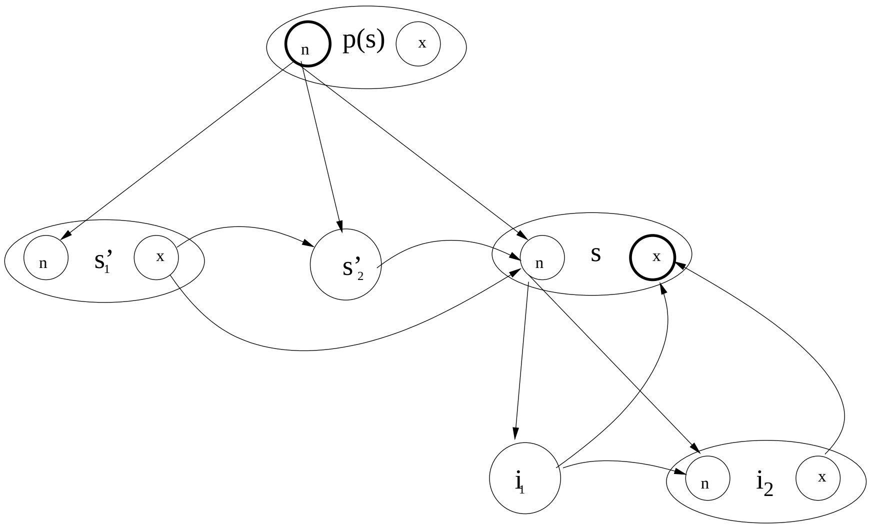

Figure 3.7: The figure illustrates Equation 3.5 which is used for situations where the parent of state s, $p(s)$ starting from entry state $p(s)_n$ produced observations from t to $t+k$ (where $k \geq 0$), and at time $t+k$ abstract state s was exited from s_x. We need to sum over all lengths of observation 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.

Overall, for each sublevel (or internal state) of the model we need to compute approximately $T^2/2$ α terms, where T is the length of the observation sequence (see Figure 3.8). In addition, to compute an α term we might need in the worst case to look at all the states of the model N and all related α variables calculated in the past. Thus the time complexity is $O(NT^3)$.

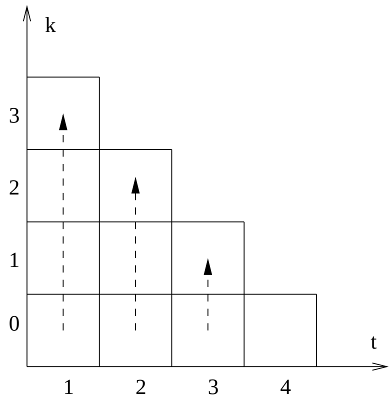

Figure 3.8: Each square is a new α computation for a subsequence $t+k$ for some internal state. The dashed arrows show dependencies on previous computations.

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 as shown in Equation 3.6.

$$P(Z|A,\lambda) = \sum_{i \in C_p^s \cup C_x^s} \alpha(s,i,1,T),\ where\ s = root\ entry \tag{3.6}$$

The next important variable for the expectation step is the backward variable β (similar to HMMs), which is defined in Equation 3.7 and described in Figure 3.9.

$$\begin{aligned} &\beta(p(s)_x, s_n, t, t+k) = \\ &P(z_t, ...z_{t+k}|a_t, ...a_{t+k},\ s_n \text{ started at } t,\ p(s) \text{ generated} \\ &\ z_t...z_{t+k} \text{ and exited from } p(s)_x \text{ at } t+k,\ \lambda) \end{aligned} \tag{3.7}$$

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

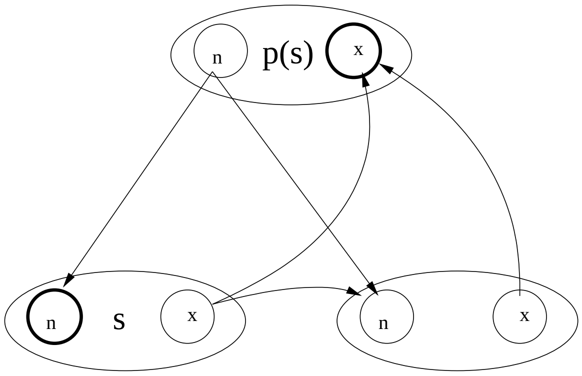

Figure 3.9: The β variable (Equation 3.7) denotes the probability that a state s was entered at time t from entry state s_n and that the observations $z_t...z_{t+k}$ were produced by the parent of s, which is $p(s)$, and actions $a_t, ...a_{t+k}$ were taken, and $p(s)$ terminated at time $t+k$ from exit state $p(s)_x$.

sequence length and $T = N - 1$), Equation 3.8 computes the probability of a single observation and is described in Figure 3.10. When $p(s) = root$ we only use these equations for $t = T$.

$$\beta(p(s)_x, s, t, t) = O(z_t|s, a_{t-1})T(p(s)_x|s, a_t), \quad p(s) \neq root \tag{3.8}$$

$$\beta(p(s), s, t, t) = O(z_t|s, a_{t-1}), \quad p(s) = root \tag{3.9}$$

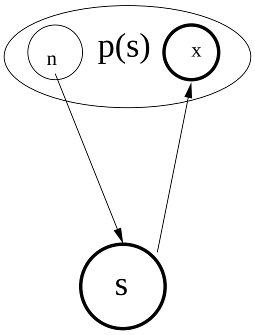

Figure 3.10: The figure illustrates Equation 3.8 which computes the probability that at time t the product state s produced observation z_t and then the system exited the abstract state $p(s)$ from exit state $p(s)_x$.

2. If s is an abstract state at $depth = level$, for $t = N-1...1$, Equation 3.10 computes the probability of observation z_t and is described in Figure 3.11. When $p(s) = root$ we only use these equations for $t = T$.

$$\beta(p(s)_x, s_n, t, t) =$$
$$\sum_{s_x \in X^s} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t) \right] T(p(s)_x|s_x, a_t), \quad p(s) \neq root \tag{3.10}$$

$$\beta(p(s), s_n, t, t) = \sum_{s_x \in X^s} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t) \right], \quad p(s) = root \tag{3.11}$$

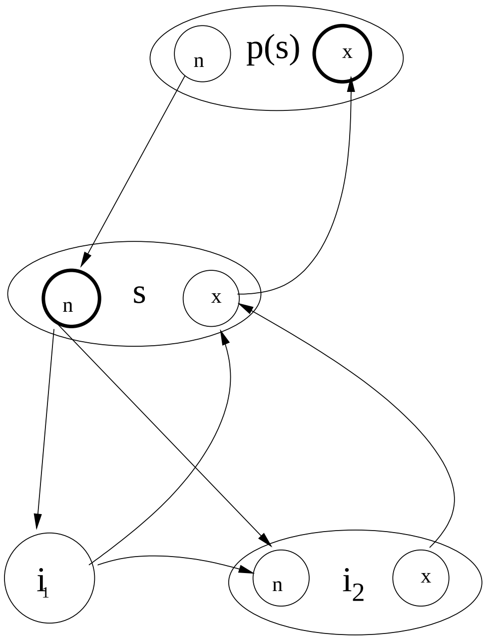

Figure 3.11: Equation 3.10 computes the probability that at time t that any of the children of abstract state s (such as i_1 and i_2) has produced observation z_t and then the parent of s $p(s)$ was exited from exit state $p(s)_x$.

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 3.12 and described in Figure 3.12. When $p(s) = root$ we only use this equation for $t + k = T$.

$$\beta(p(s)_x, s, t, t+k) = \\ O(z_t|s, a_{t-1}) \left[\sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} T(s'_n|s, a_t)\beta(p(s)_x, s'_n, t+1, t+k) \right] \tag{3.12}$$

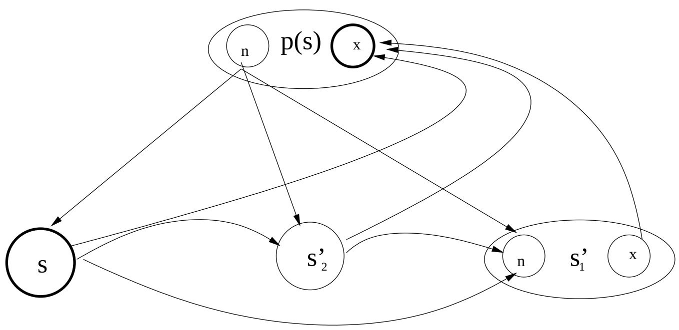

Figure 3.12: The figure illustrates Equation 3.12 which defines the probability that at time t the product state s produced observation z_t and its parent $p(s)$ continued to produce observations $z_{t+1}...z_{t+k}$ and exited at time $t+k$ from exit state $p(s)_x$, where $k \geq 1$ (or at least two 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 3.13 and described in Figure 3.13. When $p(s) = root$ we only use these equations for $t + k = T$.

$$
\begin{aligned}
&\beta(p(s)_x, s_n, t, t+k) = \\
&\sum_{s_x \in X^s} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t+k) \right] T(p(s)_x|s_x, a_{t+k}) \\
&+ \left[\sum_{l=0}^{k-1} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t+l) \right] \right. \\
&\left. \left[\sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} T(s'_n|s_x, a_{t+l})\beta(p(s)_x, s'_n, t+l+1, t+k) \right] \right], \\
&p(s) \neq root
\end{aligned}
\tag{3.13}
$$

$$
\begin{aligned}
&\beta(p(s), s_n, t, t+k) = \\
&\sum_{s_x \in X^s} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t+k) \right] \\
&+ \left[\sum_{l=0}^{k-1} \left[\sum_{i_n \in C_n^s \cup C_p^s} V(i_n|s_n)\beta(s_x, i_n, t, t+l) \right] \right. \\
&\left. \left[\sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} T(s'_n|s_x, a_{t+l})\beta(p(s), s'_n, t+l+1, t+k) \right] \right], \\
&p(s) = root
\end{aligned}
\tag{3.14}
$$

Calculating the β is also an $O(NT^3)$ computation. The computation dependencies are shown in Figure 3.14.

The next variable is ξ, which is one of the results of the expectation steps, is defined in Equation 3.15, and described in Figure 3.15.

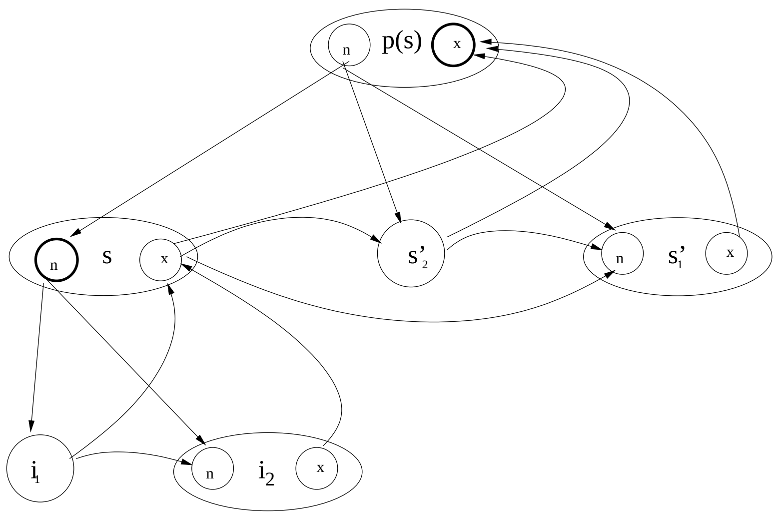

Figure 3.13: The figure describes Equation 3.13 which is the probability that at time t the abstract state s, starting from entry state s_n produced some observations, and then the parent of s, $p(s)$ continued to produce the rest of the observations and exited at time $t+k$ from exit state $p(s)_x$. 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 $k \geq 1$.

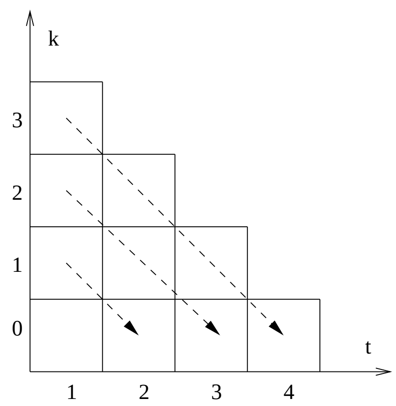

Figure 3.14: Each square is a new β computation for a subsequence $t+k$ for some internal state. The dashed arrows show dependencies on previous computations.

$$\xi(t, s_x, s'_n) = P(s \text{ finished at time } t \text{ from } s_x$$
$$s' \text{ started at time } t+1 \text{ from } s'_n \mid a_0, ...a_T, z_1, ...z_T,\ \lambda) \tag{3.15}$$

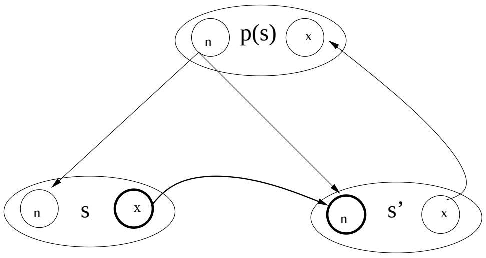

Figure 3.15: The ξ variable denotes the probability of making a horizontal transition from exit state s_x to entry state s'_n at time t, given a sequence of observations and actions and the model parameters.

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 3.16 and described in Figure 3.16.

$$\eta_{in}(t, s_n) = P(z_t, ...z_{t-1},\ s \text{ was entered at } t \text{ from } s_n \mid a_0, ..., a_{t-1},\ \lambda) \tag{3.16}$$

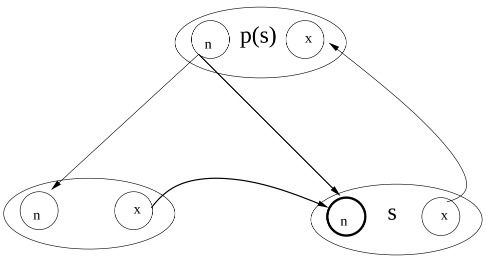

Figure 3.16: The η_{in} variable defines the probability that a state $s$is entered at time t either vertically or horizontally given a sequence of actions and observations up to time $t-1$.

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

3.17, and for $t > 1$ as shown in Equation 3.18

$$\eta_{in}(1, s_n) = V(s_n|p(s)_n),\ where\ p(s)_n = root \tag{3.17}$$

$$\eta_{in}(t, s_n) = \sum_{s'_x \in C_p^{p(s)} \cup C_x^{p(s)}} \alpha(p(s)_n, s'_x, 1, t-1)T(s_n|s'_x, a_t - 1) \tag{3.18}$$

2. If s is not a child of the root

$$\begin{aligned}\eta_{in}(t, s_n) = \\ &\sum_{p(s)_n \in N^{p(s)}} \sum_{k=0}^{t-1} \eta_{in}(k, p(s)_n) \left[\sum_{s'_x \in C_p^{p(s)} \cup C_x^{p(s)}} \alpha(p(s)_n, s'_x, k, t-1)T(s_n|s'_x, a_{t-1}) \right] \\ &+ \eta_{in}(t, p(s)_n)V(s_n|p(s)_n)\end{aligned} \tag{3.19}$$

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 3.20 and described in Figure 3.17. It can also be estimated recursively from root to leaves.

$$\eta_{out}(t, s_x) = P(s \text{ finished at } t,\ \text{from } s_x,\ z_{t+1}, ...z_T \mid a_t, ..., a_T,\ \lambda) \tag{3.20}$$

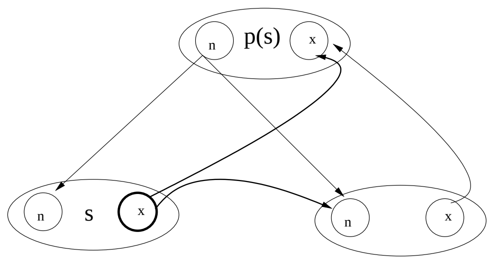

Figure 3.17: The η_{out} variable defines the probability that a state s exited at time t and observations $z_{t+1}, ...z_T$ were perceived and actions $a_t, ..., a_T$ taken.

1. If s is a child of the root.

$$\eta_{out}(t, s_x) = \sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} T(s'_n|s_x, a_t)\beta(p(s), s'_n, t+1, T), \quad t < T, p(s) = root \tag{3.21}$$

$$\eta_{out}(T, s_x) = 1.0 \tag{3.22}$$

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

$$\begin{aligned} &\eta_{out}(t, s_x) = \\ &\quad \sum_{p(s)_x \in X^{p(s)}} \sum_{k=t+1}^{T} \left[\sum_{s'_n \in C_n^{p(s)}, \cup C_p^{p(s)}} T(s'_n|s_x, a_t)\beta(p(s)_x, s'_n, t+1, k) \right] \eta_{out}(k, p(s)_x) \\ &\qquad + T(p(s)_x|s_x, a_t)\eta_{out}(t, p(s)_x) \end{aligned} \tag{3.23}$$

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 3.15 recursively from the root to the leaves of the tree.

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

$$\xi(t, s_x, s'_n) = \frac{\alpha(p(s)_n, s_x, 1, t)T(s'_n|s_x, a_t)\beta(p(s)_x, s'_n, t+1, T)}{P(Z|A, \lambda)}, \quad t < T \tag{3.24}$$

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

2. If s and s' are not root children

$$\begin{aligned} &\xi(t, s_x, s'_n) = \\ &\frac{1}{P(Z|A,\lambda)} \left[\sum_{k=1}^{t} \sum_{p(s)_n \in N^{p(s)}} \eta_{in}(k, p(s)_n)\ \alpha(p(s)_n, s_x, k, t) \right] T(s'_n|s_x, a_t) \\ &\left[\sum_{k=t+1}^{T} \sum_{p(s)_x \in X^{p(s)}} \beta(p(s)_x, s'_n, t+1, k)\ \eta_{out}(k, p(s)_x) \right], \quad t < T \end{aligned} \tag{3.26}$$

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

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.

$$\xi(t, s_x, p(s)_x) = \frac{1}{P(Z|A,\lambda)} \left[\sum_{k=1}^{t} \sum_{p(s)_n \in N^{p(s)}} \eta_{in}(k, p(s)_n)\, \alpha(p(s)_n, s_x, k, t) \right] T(p(s)_x|s_x, a_t)\, \eta_{out}(t, p(s)_x), \quad t \leq T \tag{3.28}$$

The second result of the expectation step is the $\chi(t, p(s)_n, s_n)$ variable, shown in Equation 3.29 and described in Figure 3.18.

$$\chi(t, p(s)_n, s_n) = P(p(s)_n \text{ started at time } t, s_n \text{ started at time } t \mid a_0, ...a_T, z_1, ...z_T, \lambda) \tag{3.29}$$

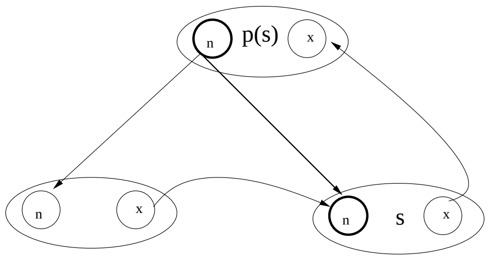

Figure 3.18: The χ defines the probability that entry state s_n of child s was activated by the entry state $p(s)_n$ of its parent $p(s)$ at time t given the action and observation sequence.

We can calculate the χ variable as follows:

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

$$\chi(1, p(s)_n, s_n) = \frac{V(s_n|p(s)_n)\, \beta(p(s)_x, s_n, 1, T, s)}{P(Z|A,\lambda)} \tag{3.30}$$

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

$$\chi(t, p(s)_n, s_n) = \frac{\eta_{in}(t, p(s)_n)\ V(s_n|p(s)_n)}{P(Z|A,\lambda)} \sum_{k=t}^{T} \sum_{p(s)_x \in X^{p(s)}} \beta(p(s)_x, s_n, t, k)\ \eta_{out}(k, p(s)_x) \tag{3.31}$$

Based on these variables we can reestimate the model parameters. The vertical vectors are reestimated in Equations 3.32 and 3.33.

$$V(s_n|p(s)_n) = \chi(1, p(s)_n, s_n),\ if\ p(s) = root,\ s_n \in C_n^{p(s)} \cup C_p^{p(s)} \tag{3.32}$$

$$V(s_n|p(s)_n) = \frac{\sum_{t=1}^{T} \chi(t, p(s)_n, s_n)}{\sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} \sum_{t=1}^{T} \chi(t, p(s)_n, s'_n)},\quad if\ s \neq root \tag{3.33}$$

Equation 3.33 finds the vertical transition to any non-root child by calculating the number of times the child was vertically entered over the total number of time the child and its siblings were entered. The horizontal transition matrices are reestimated in Equation 3.34. The reestimation 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.

$$T(s'_n|s_x, a) = \frac{\sum_{t=1|a_t=a}^{T} \xi(t, s_x, s'_n)}{\sum_{t=1|a_t=a}^{T} \sum_{s''_n \in C_p^{p(s)} \cup C_n^{p(s)}} \xi(t, s_x, s''_n)} \tag{3.34}$$

The observation vectors are reestimated in Equation 3.35. The reestimation of the observation model 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.

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

$$where \tag{3.35}$$

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

All of the above formulas of the hierarchical Baum-Welch 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 sequences are independent of each other (*iid*), then the probability of all sequences can be calculated as the product of the individual sequences as shown in Equation 3.36

$$P(Z_1,...,Z_K|A_1,...A_K,\lambda) = \prod_{k=1}^{K} P(Z_k|A_k,\lambda) \tag{3.36}$$

Since reestimation is based on frequencies of occurrence of various events, the new reestimation formulas can be constructed by adding together the individual frequencies of occurrence for each data sequence. The vertical transition are reestimated in Equations 3.37 and 3.38, the horizontal in Equation 3.39, and the observation models in Equation 3.40.

$$V(s_n|p(s)_n) = \sum_{k=1}^{K} \chi_k(1, p(s)_n, s_n),\ if\ p(s) = root,\ s_n \in C_n^{p(s)} \cup C_p^{p(s)} \tag{3.37}$$

$$\begin{aligned} V(s_n|p(s)_n) =& \frac{\sum_{k=1}^{K} \sum_{t=1}^{T_k} \chi_k(t, p(s)_n, s_n)}{\sum_{k=1}^{K} \sum_{s'_n \in C_n^{p(s)} \cup C_p^{p(s)}} \sum_{t=1}^{T_k} \chi_k(t, p(s)_n, s'_n)}, \\ & if\ s \neq root \end{aligned} \tag{3.38}$$

$$T(s'_n|s_x, a) = \frac{\sum_{k=1}^{K} \sum_{t=1|a_t=a}^{T_k} \xi_k(t, s_x, s'_n)}{\sum_{k=1}^{K} \sum_{t=1|a_t=a}^{T_k} \sum_{s''_n \in C_p^{p(s)} \cup C_n^{p(s)}} \xi_k(t, s_x, s''_n)} \tag{3.39}$$

$$O(z|s, a) = \frac{\sum_{k=1}^{K} \sum_{t=1, z_t=z, a_{t-1}=a}^{T_k} \chi_k(t, p(s)_n, s) + \sum_{t=2, z_t=z, a_{t-1}=a}^{T_k} \gamma_{in_k}(t, s)}{\sum_{k=1}^{K} \sum_{t=1}^{T_k} \chi_k(t, p(s)_n, s) + \sum_{t=2}^{T_k} \gamma_{in_k}(t, s)} \tag{3.40}$$

3.2.2 Learning a small corridor environment

To test the HPOMDP learning algorithm we created a training set of 100 action and observation sequences by sampling the original HPOMDP in Figure 3.2. We then used these sequences to train different models, which were initialized with random transitions (horizontal and vertical) and random observation models. To create sequences of actions and observations, we simulate the model and choose transitions (vertical and horizontal) using the inverse transform technique [80]. For each sequence we first randomly choose an action a_0 and then start from the root and vertically activate submodels until a product state is reached and observation z_1 produced. We then randomly choose the next action and transition to the next state. If the next state is a product state we produce the next observation. If the next state is an exit state we transition to an adjacent state of the parent. If the adjacent state is an abstract state, we transition vertically to submodels until a product state is reached. If the adjacent state is a product state we produce an observation. Figure 3.19 shows a model with random transition probabilities and Table 3.2 shows the initial sensor model probabilities. Figure 3.20, and Table 3.2 show the model parameters after training. It is obvious that the algorithm learns a model which is close to the original.

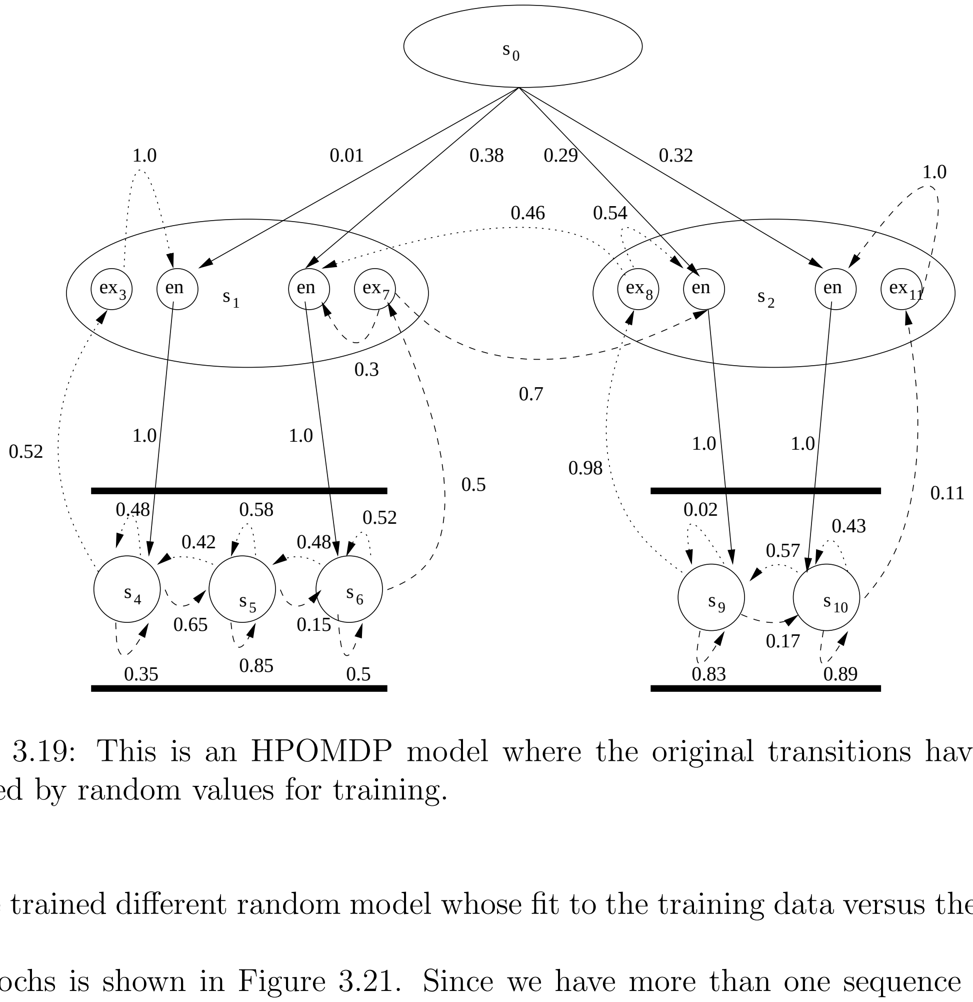

Figure 3.19: This is an HPOMDP model where the original transitions have been replaced by random values for training.

We trained different random model whose fit to the training data versus the training epochs is shown in Figure 3.21. Since we have more than one sequence during training the log-likelihood $\log P(Z|A, \lambda)$ represents all the data sequence (is the sum

	Orientation	W	W	S	S	E	E	N	N
State	Action	P(w)	P(o)	P(w)	P(o)	P(w)	P(o)	P(w)	P(o)
s_4	go-left	0.27	0.73	0.5	0.5	0.64	0.36	0.29	0.71
s_4	go-right	0.52	0.48	0.63	0.37	0.1	0.9	0.38	0.62
s_5	go-left	0.14	0.86	0.37	0.63	0.19	0.81	0.67	0.33
s_5	go-right	0.57	0.43	0.19	0.81	0.31	0.69	0.7	0.3
s_6	go-left	0.57	0.43	0.3	0.7	0.78	0.22	0.4	0.6
s_6	go-right	0.1	0.9	0.2	0.8	0.79	0.21	0.34	0.66
s_9	go-left	0.51	0.49	0.85	0.15	0.53	0.47	0.19	0.81
s_9	go-right	0.5	0.5	0.34	0.66	0.9	0.1	0.57	0.43
s_{10}	go-left	0.87	0.13	0.05	0.95	0.36	0.64	0.45	0.55
s_{10}	go-right	0.55	0.45	0.65	0.35	0.8	0.2	0.36	0.64

Table 3.2: The table shows an initial sensor model which has been created randomly.

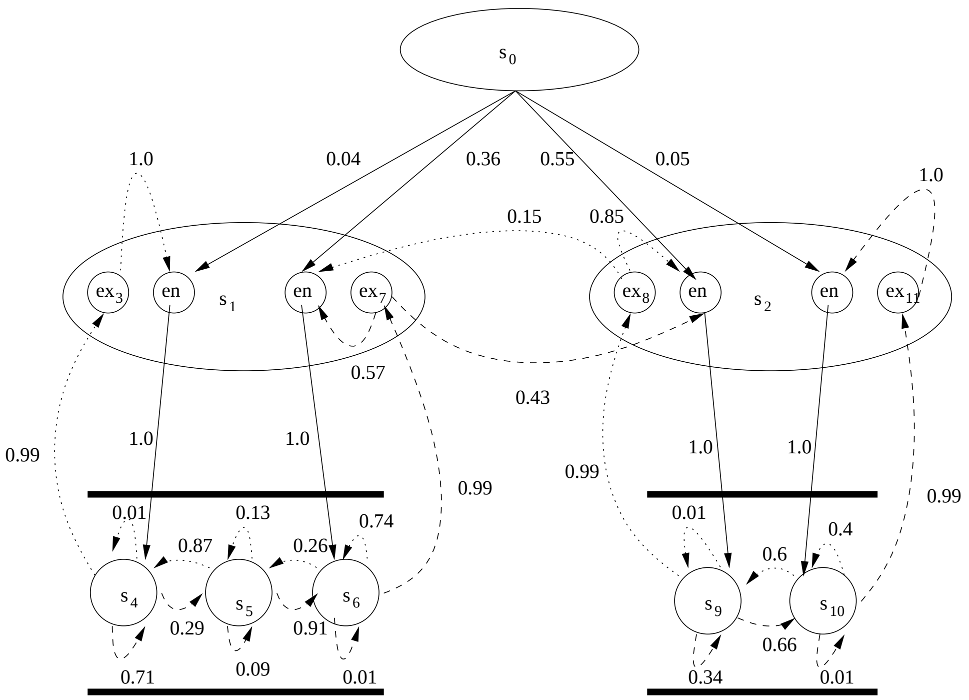

Figure 3.20: The random HPOMDP model after training, is more similar to the original model (Figure 3.2) after training.

	Orientation	W	W	S	S	E	E	N	N
State	Action	P(w)	P(o)	P(w)	P(o)	P(w)	P(o)	P(w)	P(o)
s_4	go-left	0.10	0.90	0.92	0.08	0.20	0.80	0.92	0.08
s_4	go-right	0.0	1.0	1.0	0.0	0.06	0.94	0.99	0.01
s_5	go-left	0.01	0.99	0.77	0.23	0.0	1.0	0.48	0.52
s_5	go-right	0.0	1.0	1.0	0.0	0.05	0.95	1.0	0.0
s_6	go-left	0.0	1.0	0.83	0.17	0.05	0.95	0.95	0.05
s_6	go-right	0.0	1.0	0.81	0.19	0.15	0.85	0.87	0.13
s_9	go-left	0.15	0.85	0.94	0.06	0.13	0.87	0.98	0.02
s_9	go-right	0.12	0.88	0.89	0.11	0.14	0.86	0.94	0.06
s_{10}	go-left	0.26	0.74	0.92	0.08	0.0	1.0	1.0	0.0
s_{10}	go-right	0.17	0.83	0.94	0.06	0.13	0.87	0.91	0.09

Table 3.3: The table shows the sensor model after training. The sensor models are almost the same as the original sensor parameters in Table 3.1.

of the individual log-likelihoods). It is obvious from the plot that the different random models converge fast to the original model.

3.2.3 Converting HPOMDPs to POMDPs

Every HPOMDP can be converted to a flat POMDP. The states of the flat POMDP are the product states of the HPOMDP, and are associated with a global transition matrix that is calculated from the vertical and horizontal transition matrices of the HPOMDP. To construct this global transition matrix of the equivalent flat POMDP, for every pair of states (s_1, s_2), we need to sum up the probabilities of all the paths that will transition the system from s_1 to s_2 under some action a. For example in Figure 3.2 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. On 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 $0.9 \times 0.1.0 = 0.09$. Therefore the total probability is 0.19. The equivalent flat POMDP is shown here in Figure 3.22.

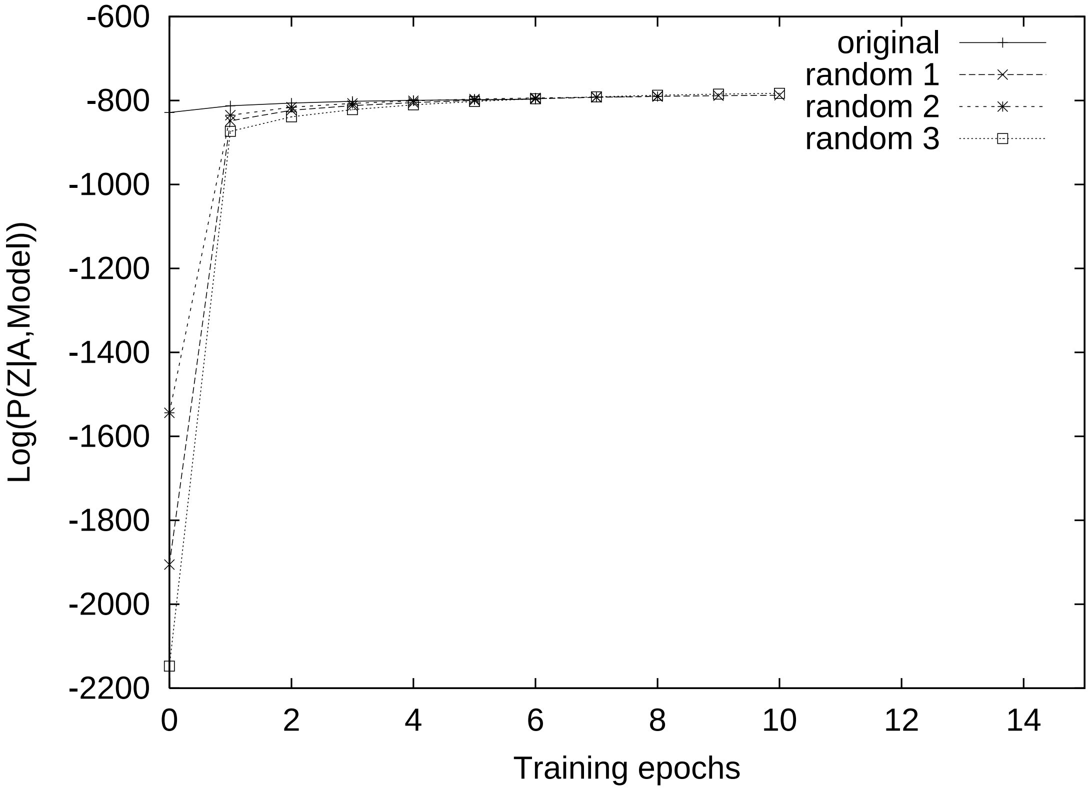

Figure 3.21: Training runs of three different random models shows that the log likelihood converges fast to the original model. The original model does not improve as expected since it was used to create the training data. In the log likelihood function Z is the set of observation sequences, A is the set of action sequences, and Model is the set of parameters that are improved every training epoch (horizontal matrices, vertical vectors, and observation models).

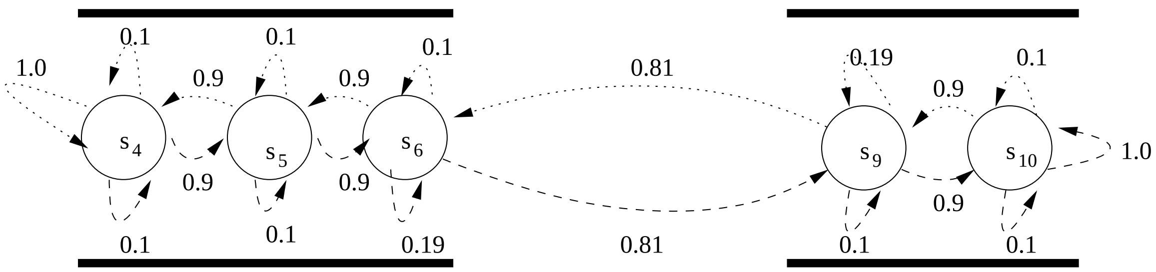

Figure 3.22: This is the equivalent flat POMDP of the hierarchical POMDP shown in Figure 3.2.

Converting an HPOMDP to flat model, however, is not advantageous since we lose the abstract states and the vertical transitions which we can exploit later 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 model from a good initial model. This results in better trained models since the hierarchical Baum-Welch 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 state (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.

Nonetheless, the equivalent flat POMDP will provide different likelihood to the data than the hierarchical. The reason is that the α term for hierarchical models, which defines the probability that an 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 better understood in Figure 3.23. This property of the HHMM/HPOMDP models, in that the next abstract state cannot be entered until the current abstract state has exited, allows us to capture relationships between states that are far apart.

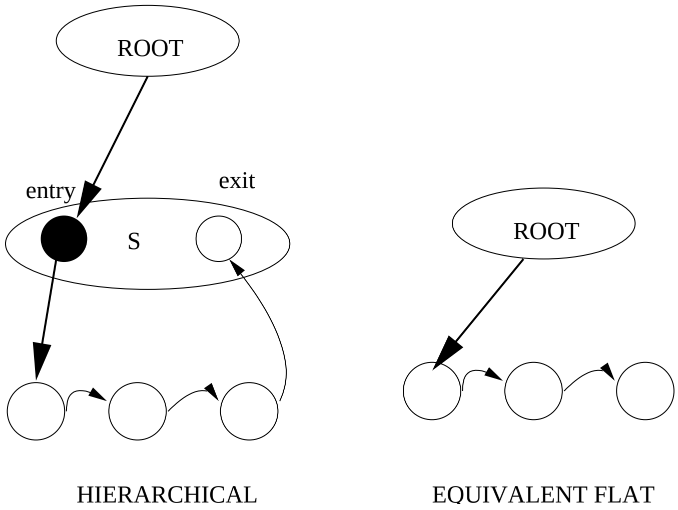

Figure 3.23: For an observation sequence of length 2 the hierarchical likelihood will be zero, since we need at least three observations to exit the abstracts state S. While the likelihood in the equivalent flat model will not be zero.

3.3 Related multi-resolution Hidden Markov Models

HHMMs can also be viewed as a specific case of Stochastic Context Free Grammars (SCFGs). The generalized hierarchical Baum-Welch presented in [17] is inspired by the Inside-Outside algorithm which is an EM type algorithm for learning the probabilities of SCFG rules [81], [82]. The Inside-Outside algorithm is less efficient than the EM algorithm for HHMMs because it also runs in cubic time in terms of the length of the training sequence, but also runs in cubic time in terms of 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 don't share subtrees, and the whole model is of bounded depth. However, it would not be difficult to extend the generalized Baum-

Welch in [17] to handle HHMMs whose subtrees share structure. Since the models of each submodel (vertical vectors, horizontal transitions, and observation models) are defined separately based on the parent abstract state of the submodel, then one way of learning shared substructures models is to average all model parameters that are being shared by different abstract states.

Another way of representing HHMMs is with a special case of Dynamic Bayesian Networks (DBNs) [83]. The transformation of HHMMs to DBNs allows us to train them by combining standard inference techniques in Bayesian nets such as the junction-tree algorithm [84]. The junction-tree algorithm replaces the expectation step in the hierarchical Baum-Welch algorithm. The major advantage is that with this combination HHMMs can be trained in linear time while the conventional hierarchical Baum-Welch algorithm requires cubic time [17]. The disadvantage is that the DBN model grows linearly with the size of the training data. In addition, it still remains to be seen how well such representations scale to larger models and how the learned models compare with models learned by the standard hierarchical Baum-Welch algorithm.

Another closely related model to HHMMs is the Abstract hidden Markov model (AHMM) [85]. This is a DBN representation closely related with the options framework [25]. The states however are not fully observable and the model becomes similar to an HHMM. In this framework at each level of abstraction there is a set of abstract actions available. The states for which these abstract actions are applicable form a decomposition of the state space. Plan execution starts from top of the hierarchy where the most abstract level issues an abstract action. The abstract action may issue 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. Termination of intermediate level abstract actions depends on a special set of destination states available for each level of abstraction. Upon termination of an abstract action in some abstract level the level above chooses the next abstract action. The process terminates when the abstract action at the most abstract level terminates and all other called abstract actions at lower levels terminated as well. The DBN representation models the relationships between the different policies at all levels of abstraction, the termination conditions, the hidden states, and the primitive actions.

The AHMM framework can be used to infer the probability of each possible abstract action at each level given a sequence of observations. The AHMM differs from HMMM in that it does not model the horizontal relationships between abstract states (or spatial decompositions). However, it is more similar to our extension of the HHMM model with actions and rewards, in that it also models multiple actions. This representation models the effects of abstract actions, while as we shall see later in our HPOMDP model usage in Chapter 6, the effect of abstract actions are computed based on the model of the environment.

Embedded hidden Markov models are another special case of the HHMM model, which were applied to face recognition [86]. 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 the segment models [87].

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 [10] [88], or hidden semi-Markov models [89]. These models can be thought as special cases of 2 level HHMMs. Unfortunately, the computational complexity of training durational HHMMs is usually proportional to D^2 where D is the maximum allowed duration [10]. Nonetheless, there has been work that demonstrates that training durational models can be done in time proportional to D rather than D^2 [90].

Finally, another related model, which is also called hierarchical POMDP, is presented in [91]. 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.

This chapter has described the HPOMDP model, its learning algorithm, and a small experiment that showed how the algorithm converges successfully. It has also shown the relation of HPOMDPs to flat POMDPs and concluded with a discussion on related multi-resolution models. In the next chapter we describe the robot navigation domain and show how HPOMDP models are used to model robot navigation environments.

Chapter 4

The robot navigation domain

In this chapter we describe the robot navigation domain. In Section 4.1 we describe the different robot platforms used in the learning and planning experiments of this dissertation. In Section 4.2 we describe the navigation environments used, and how they are represented as hierarchical POMDPs. In Section 4.3 we describe the features that the robot can observed and how it is done through the use of a neural network. In the final Section 4.4, we describe in detail the overall robot navigation architecture.

4.1 Robot platforms

We have conducted experiments using two robot platforms. The first is a real robot platform, which is a Nomad 200 robot called Pavlov shown in Figure 4.1 [92]. The nomad 200 is an integrated mobile robot platform with four sensory modules including tactile, infrared, ultrasonic, and vision. The platform has an onboard computer for sensor and motor control, and host computer communication. The mobile base keeps track of its position and orientation (integrated over time) [92].

Figure 4.1: This is PAVLOV, a Nomad 200. Pavlov is equipped with 16 sonar sensors, 16 infrared-red sensors, two rings of bumper sensors (with a total of 16 switches), an arm, and a camera. Pavlov has an 133 MHz Pentium onboard processor and an external Pentium II 400MHz laptop.

The simulated robot platform simulates the real Nomad 200 robot. In the simulation environment the robot can move in a two dimensional world populated with convex polygonal structures (see Figure 4.2). Sensor modules simulate each of the sensors of the real robot platform and report sensor values according to the current simulation world [93].

4.2 The environment

In our experiment we used two navigation environments. The first one, whose topology is shown in Figure 4.3, is an approximate model of the 3rd floor of the MSU

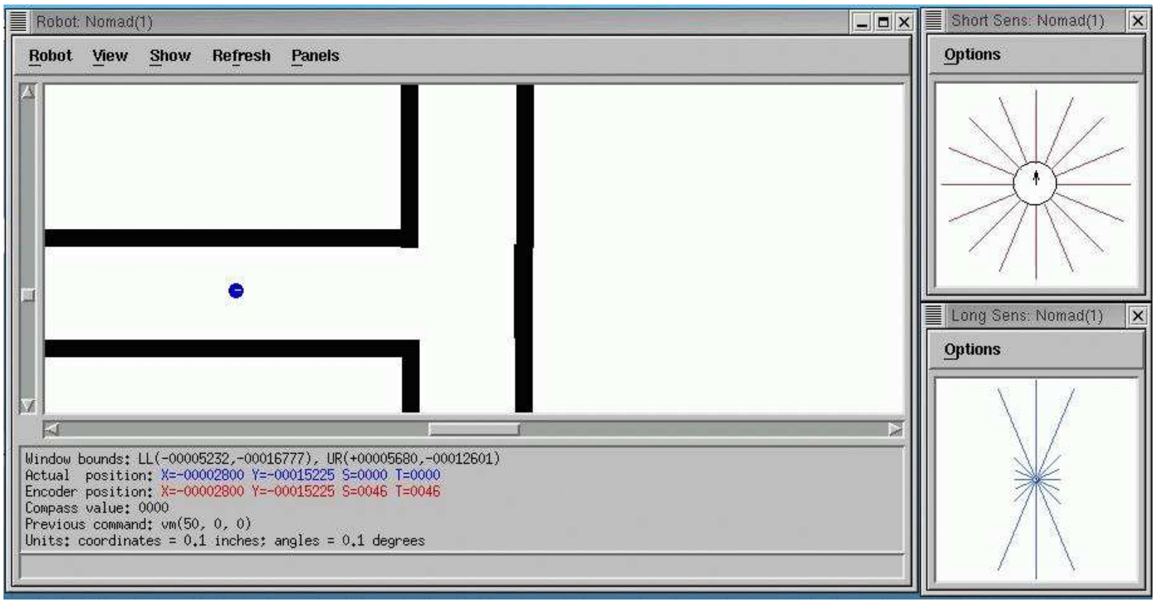

Figure 4.2: This figure illustrates the Nomad 200 simulator. The main window shows a section of the environment and the simulated robot. The two windows on the right show infrared and sonar sensor values at the current location.

engineering building. The second one, whose topology is shown in Figure 4.4, has the same dimensions as the 3rd floor of the MSU engineering building.

Such topological maps can either be learned or provided. The topology of the environment can be learned either by identifying landmark locations and their relationships [18], or by learning a metric representation of the environment and then extracting the topology [1]. In our experiments we assume that we start with topological map which we compile into an HPOMDP representations (see Figure 4.6). Figure 4.5 shows the equivalent flat of the hierarchical model. When we compile the topological map in Figure 4.3, the result is an initial HPOMDP model with 575 states. When this initial HPOMDP model is converted to a flat POMDP the number of states is 465. Compilation of the topological map in Figure 4.4 results in an HPOMDP model with 1385 states. When this HPOMDP model is converted to a flat

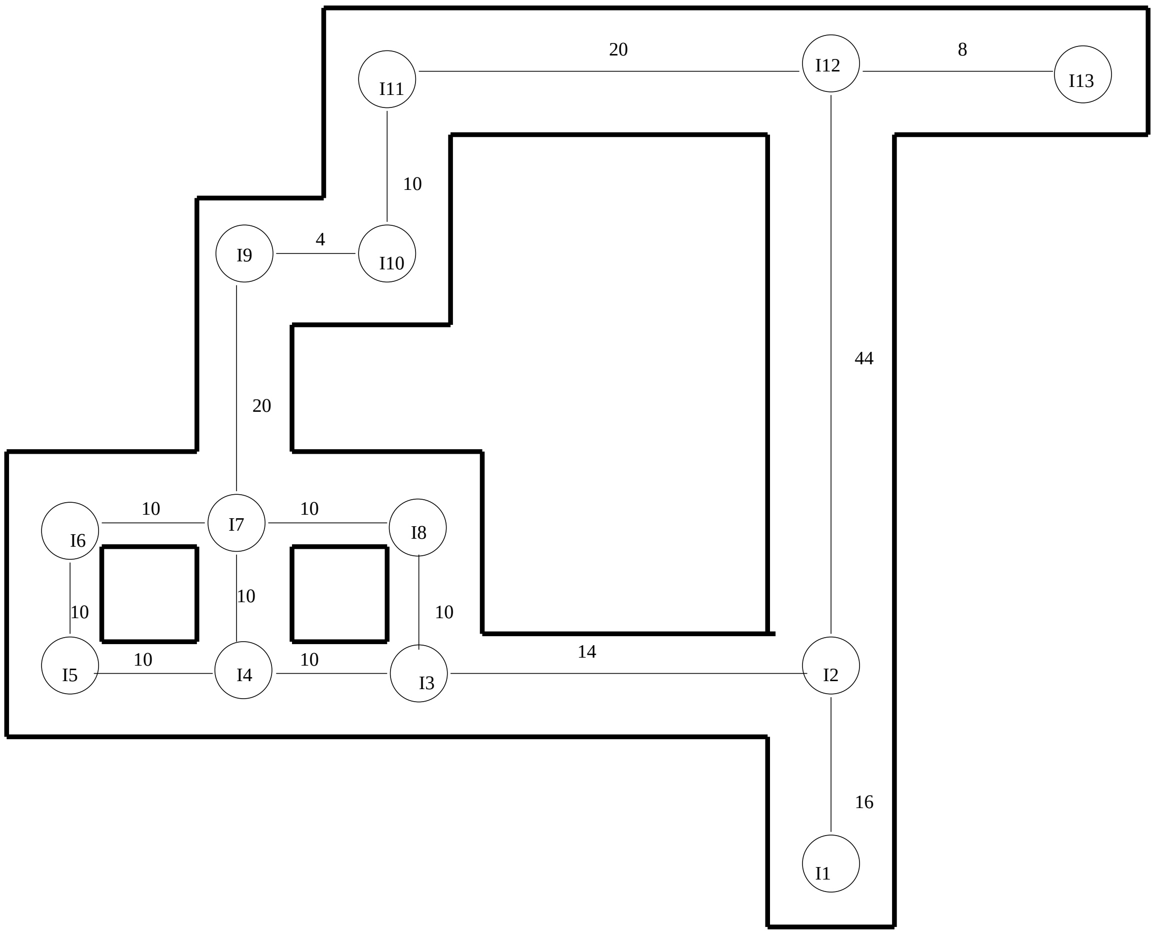

Figure 4.3: The figure shows an approximate model of the 3rd floor of the engineering building that is used with the Nomad 200 simulator. The number next to the edges is the distance between the nodes in meters.

POMDP the number of states is 1285.

To learn the parameters of the HPOMDP models first we need to collect sequences of observations and actions. To collect data we run the robot in the environment and save 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. An observation consists of 4 components, the observation of a wall or opening, on the front left, back, and right side of the robot.

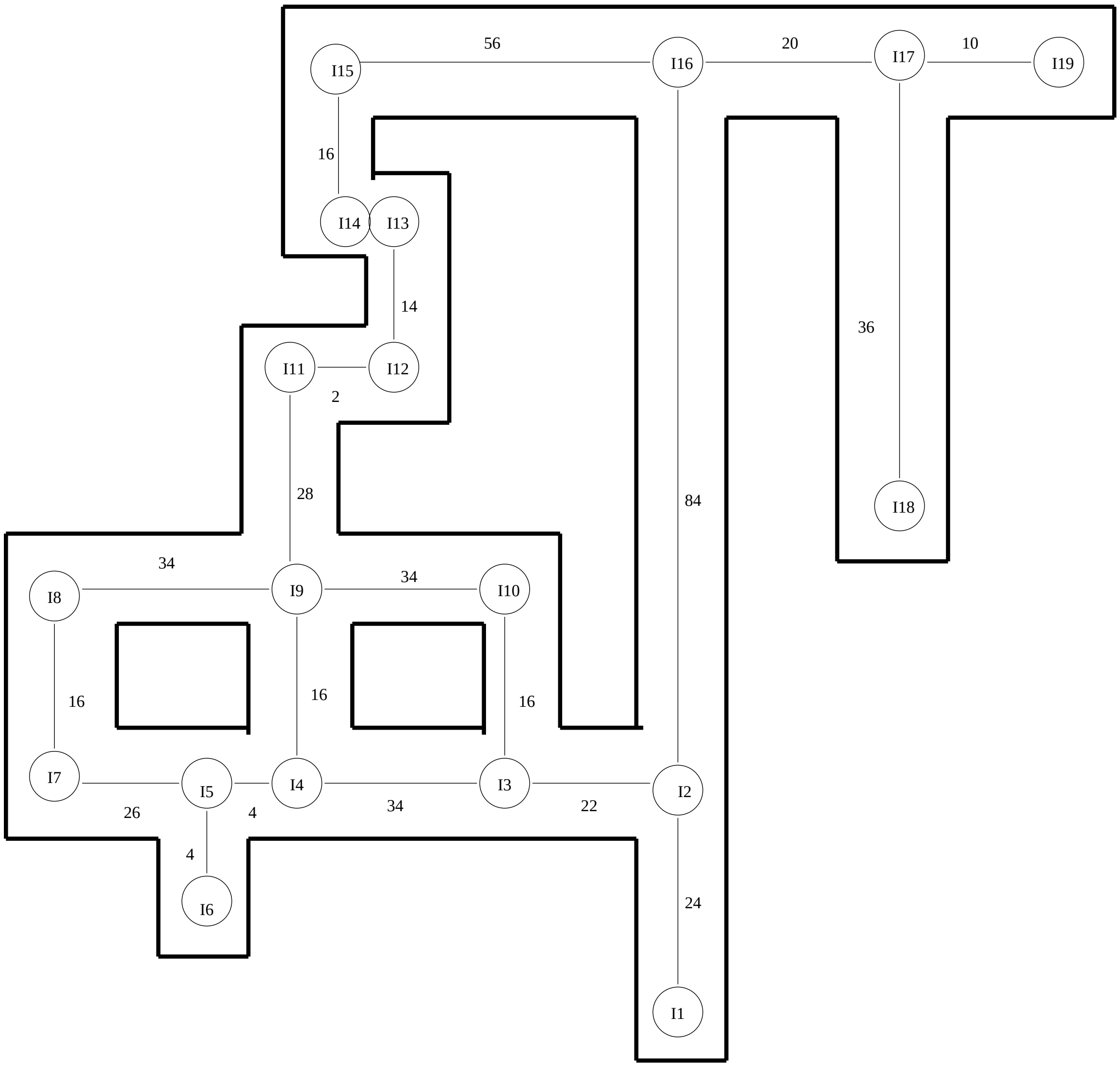

Figure 4.4: The figure shows the real 3rd floor of the engineering building. This environment is used for experiments both, in the Nomad 200 simulator, and with the real robot Pavlov in the actual physical floor. The meters next to the edges denote the distance in meters between the nodes. The nodes represent junction locations in the environment.

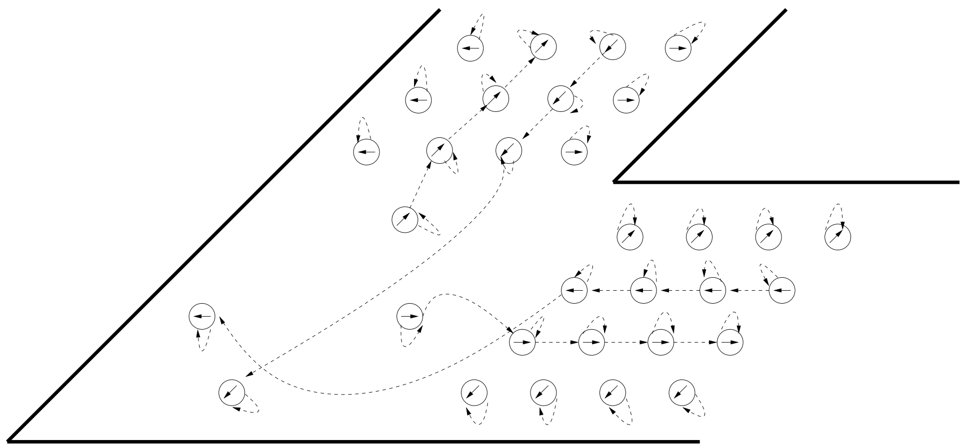

Figure 4.5: This is the equivalent flat model of the HPOMDP in Figure 4.6 for the forward action.

The observations are produces by a neural-net (Figure4.7). In the next section we describe the feature detectors.

4.3 Learning feature detectors

Features such as wall and opening on the four sides of the robot can be detected using the 16 sonar sensors around the robot. However, because the walls of our environment are fairly smooth, sonars can produce specular reflections. These reflections make it difficult to create hard-coded feature detectors for recognizing sonar signatures. Instead, we use sonar and odometric information values to build local occupancy grids which depict a local metric representation of the environment [68]. The local occupancy grids are labeled and then used as training data for a neural net feature detector which was studied earlier in [24] and [94]. Figure 4.7 shows the neural net used in feature detection. The net was trained using the quickprop method [95], an optimized variant of the backpropagation algorithm.

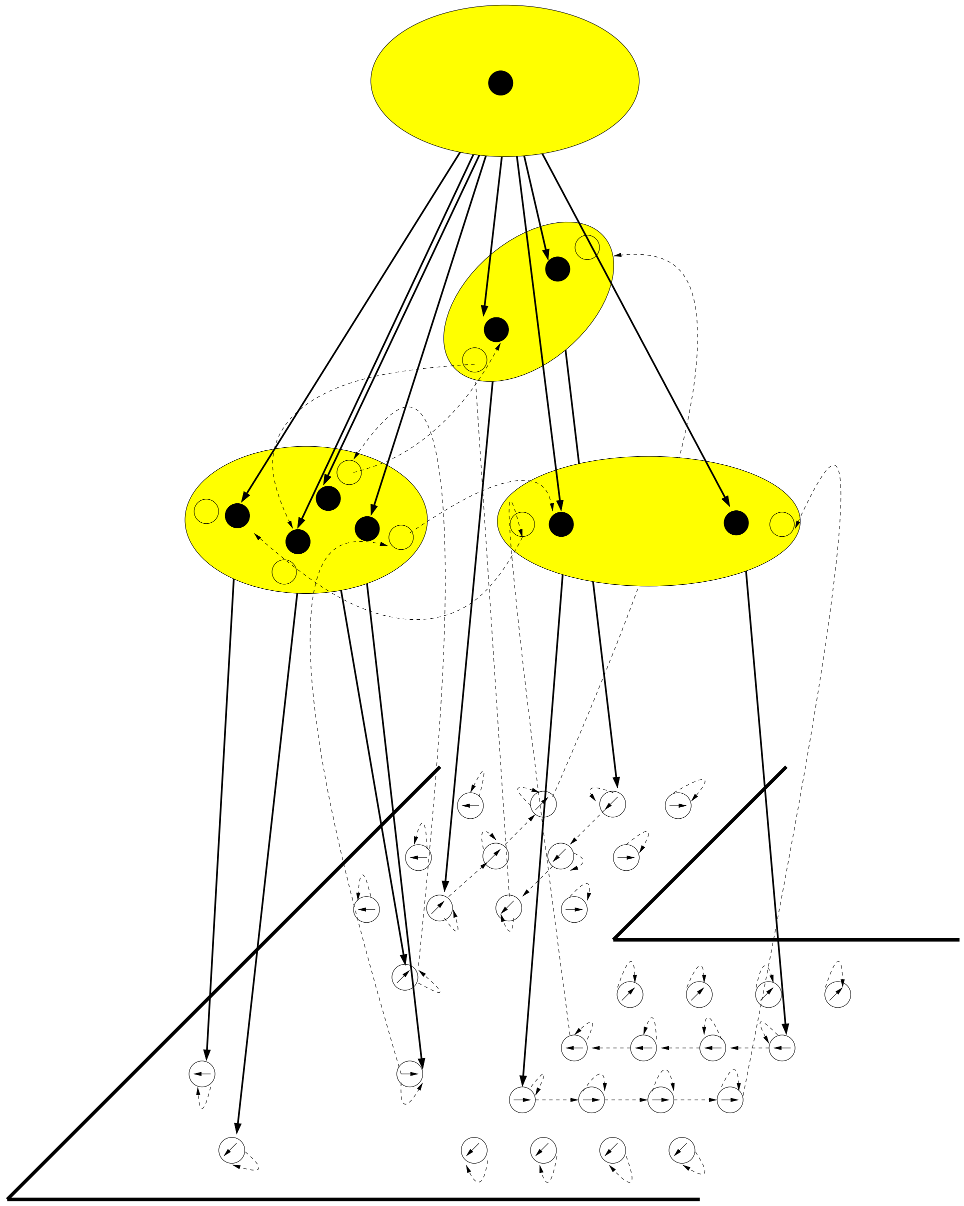

Figure 4.6: This is a sample three-dimensional view of the HPOMD model for the forward action. The solid bold arrows represent the vertical transition matrices. The dashed arrows represent the horizontal transitions. The bold circles inside the abstract states represent entry-states and the empty circle represent exit-states. In this representation all the children of the root are abstract states. Product states are the leafs of the model and represent two meter locations in the environment. The arrows inside a product state represent the orientation of the product state. For each product state there is an observation model which was previously described in Chapter 3.

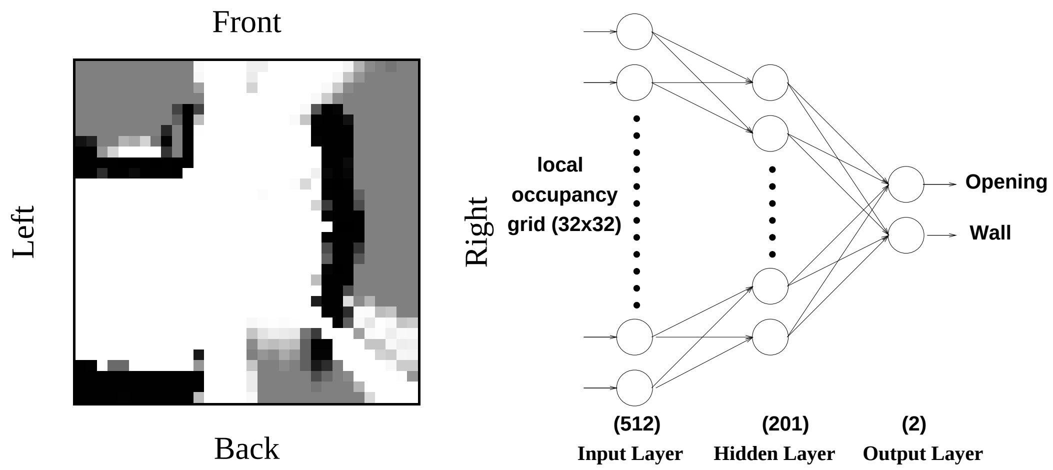

Figure 4.7: A local occupancy grid map, which is decomposed into four overlapping quadrants (left, right, top, bottom), each of which is input to a neural net feature detector. The output of the net is a multi-class label estimating the likelihood of each possible observation (opening, wall). The net is trained on manually labeled real data.

Learning the features could have also be done using any other non-parametric classifier such as decision trees, or a parametric classifier such as Bayesian [96]. Additionally, we can extend the HPOMDP model to handle continues features, or extend it such that neural nets and the parameters of the HPOMDP are trained simultaneously [97].

Sample local occupancy grids were collected by running the robot through the hallways. Each local occupancy grid was then used to produce 4 training patterns. The neural net was trained and tested with 736 hand labeled examples using the m-fold cross-validation method [96]. Since all sensors predict the same set of features, it was only necessary to learn one set of weights. Using the 736 examples we created 8 sets of hand-labeled data, where each time we used 7 for training and 1 for testing. Figure 4.8 shows the learning curves for multiple training data sets using a batch update. The training error shown in the graph is formulated as the sum of the

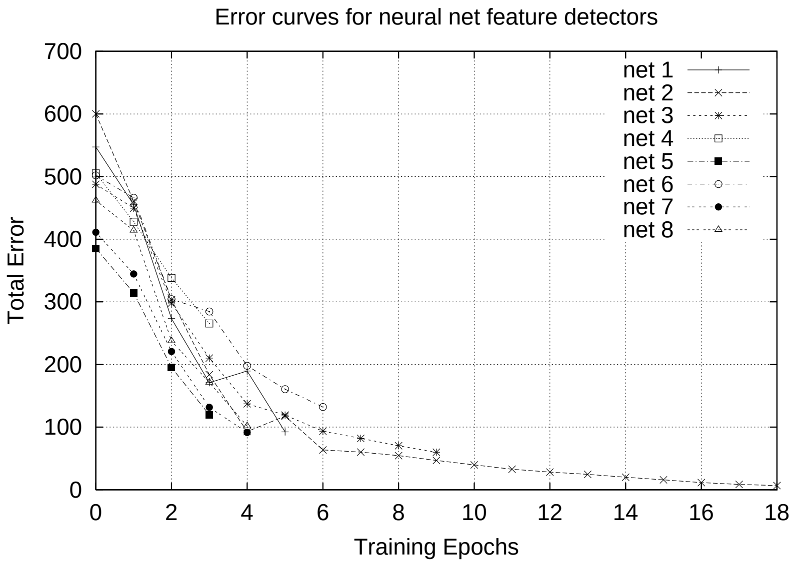

Figure 4.8: Learning curves for training the neural net to recognize features. The nets are trained with 736 examples in an 8-fold cross-validation fashion using quickprop.

square difference between the actual and the desired output over all training patterns, Training error $= \sum_{\text{all training patterns}}(\text{desired} - \text{actual})^2$. For each training run we stopped training when the classification error on the testing data set reached a minimum. A testing pattern was classified correctly if the largest value of the actual output was also the largest value in the desired output. The classification error over all test data sets had a mean value of $\mu = 7.33$ and a standard deviation value of $\sigma = 3.7$. For actual use with the robot we chose the neural net that had the smallest classification error.

Figure 4.9 illustrates the variation in observation data generated during an actual run on Pavlov. In these occupancy grids, free space is represented by white, while black represents occupied space. Gray areas indicate that occupancy information is unknown. The figures are labeled with the virtual sensors and corresponding features, as predicted by the neural net. Despite noise due to specular reflections, and noise due

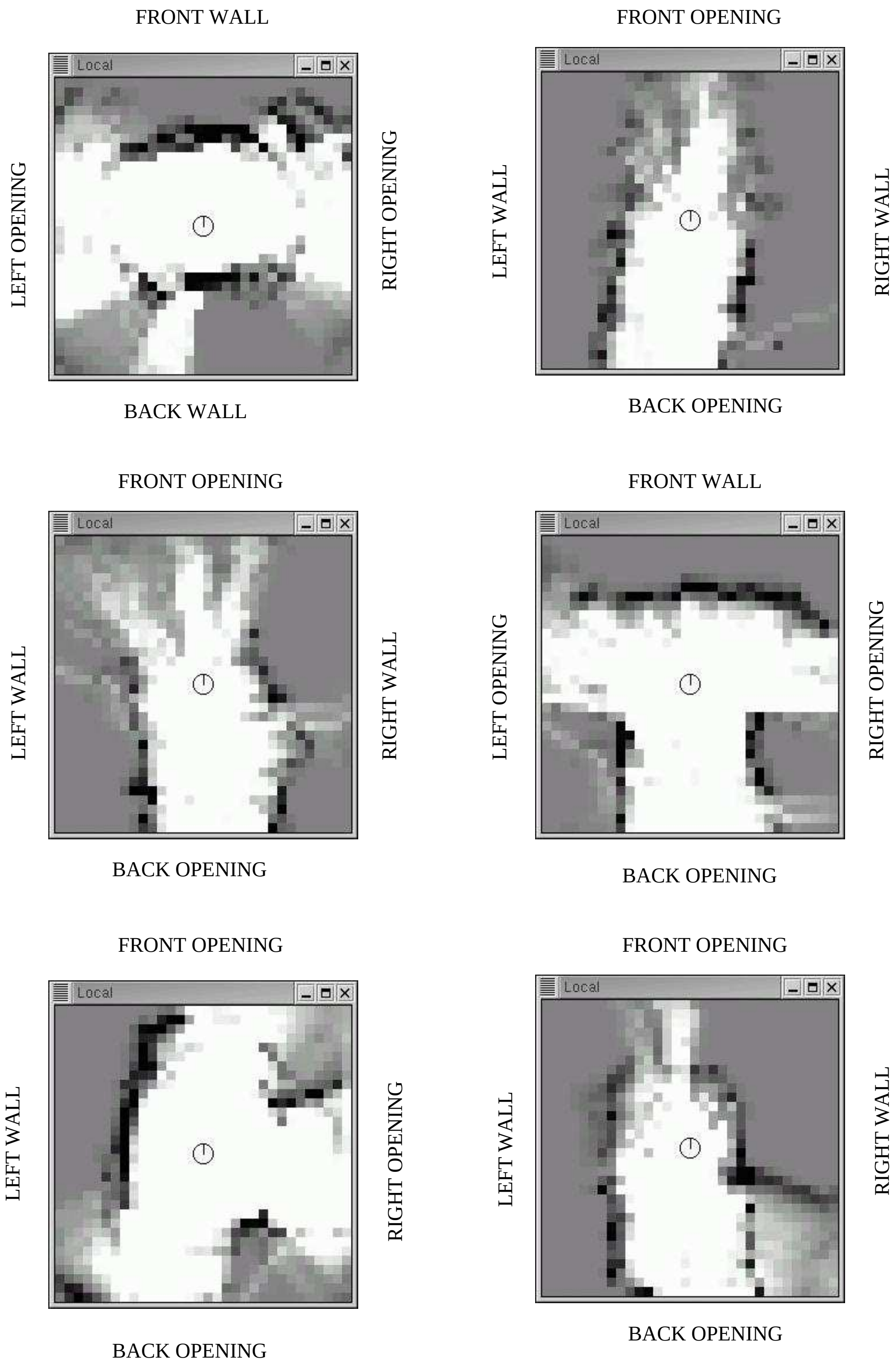

Figure 4.9: Sample local occupancy grids generated over an actual run, with observations output by the trained neural net. Despite significant sensor noise, the net is able to produce fairly reliable observations.

to robot orientation with respect to the features, the neural note is able to accurately predict the observations.

4.4 The navigation architecture

Learning and planning are integrated in an overall navigation architecture shown in Figure 4.10. The topological map construction module produces an initial HPOMDP model. In our case the we haven't implemented any automatic construction of topological maps but we rather provide it. Automatic topological map construction is not difficult to be done and has been done by previous researchers such as in [18], and [1].

The planning layer is activated at the beginning of a navigation task and produces a policy which is a mapping of states to actions. The input to the planning system is the initial or learned learned HPOMDP model and an external reward signal that defines the goal. The execution layer is responsible for updating the robot's belief after every action and observations and issuing the next action according to the plan. The Planning and execution layer are presented in Chapter 6.

The neural net forward pass module is responsible for classifying local occupancy grids into walls and openings. The neural net is trained off-line using labeled local occupancy grids. The occupancy grid module is responsible for constructing local occupancy grids based on robot motion and sonar sensors [68].

The behavior based layer is responsible for executing abstract actions such as move forward two meters, turn left, and turn right, while at the same time avoiding obstacles and keeping the robot aligned with the walls. The behavior-based framework is a widely adopted and successful approach for programming mobile robots [98], [99].

The environment learning layer takes as input an initial HPOMDP and sequences of observations and actions (produced by the run-time system) and produces a learned HPOMDP model. Learning of the model is done-off line according to the EM learning algorithm presented in Section 3.2.1. Next, in Chapter 5 we will present a detail experimental study for learning HPOMDPs in our navigation environments.

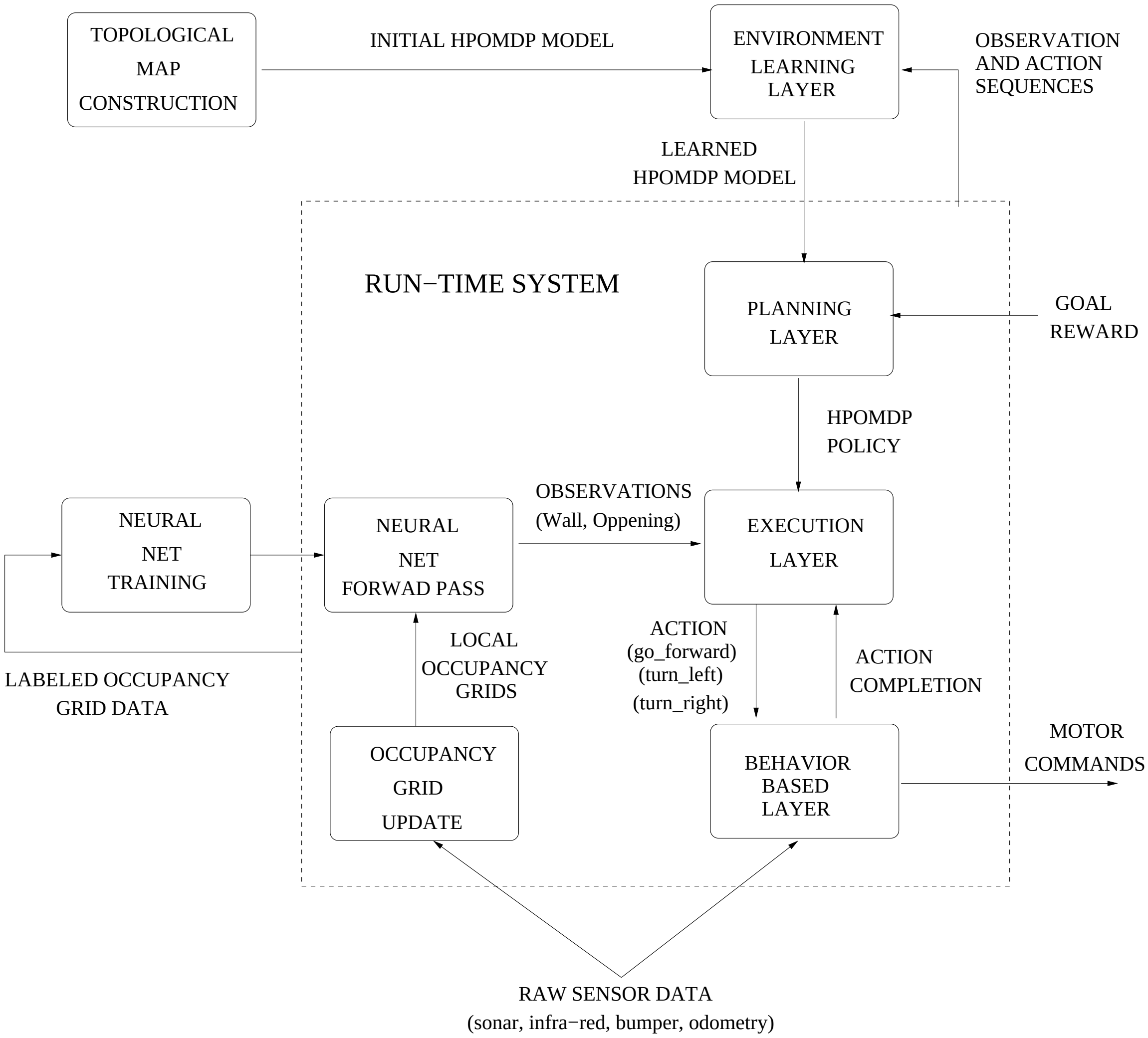

Figure 4.10: The figure shows the overall navigation architecture. The modules inside the dashed box are used during actual navigation while the models outside are used off-line

Chapter 5

Learning hierarchical spatial models for robot navigation

In this chapter we describe detailed learning experiments where we train HPOMDP models to represent hierarchical spatial maps of indoor office environments. Before any learning experiments, in Section 5.1, we present two approximate learning methods that enable fast training of large scale environments. In Section 5.2.1 we present a detail experimental study where we lean the parameters of an HPOMDP model for the spatial environment presented in Figure 4.3 in Chapter 4. For these set of experiments data is collected by sampling some "good" initial HPOMDP model. In Section 5.2.2 we present learning experiments with our robot platforms. We train both the small environment (Figure 4.3) and the larger environment (Figure 4.4) using robot experience. Next, in Section 5.3 we present experiments for learning the structure of the environment using our learning algorithms. In structure learning we allow for semi-ergodic models at abstract levels of the hierarchy and prune away incorrect connections using our learning algorithms. Finally in Section 5.4 we conclude with a discussion about the results and significance of the experiments.

5.1 Approximate learning methods

An interesting question is how long should the data sequences be in order to learn HPOMDP models. Since the time complexity of our algorithm is $O(NT^3)$, long data sequences will require unrealistic training times. Fortunately, in order to learn relationships between states which are children of the root (and the submodel 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 Equation 3.24.

However, with short training sequences there is a danger we will not learn the correct model parameters unless we provide the initial vertical vector from the root, or in other words the first state that the observation sequence was initiated from. With longer sequences, it is more likely that the starting position of the observation sequence will be discovered by itself (see Figure 5.1). In our experiments we used both short training sequences accompanied with their starting position and longer training sequences without their starting position. Obtaining the starting position of an observation sequence is easy, since for every robot run for which data is to be collected, we know the starting position of the robot. If we always need to collect data (whenever the robot is acting in the environment), and don't have an exact initial position, then we will have long training times.

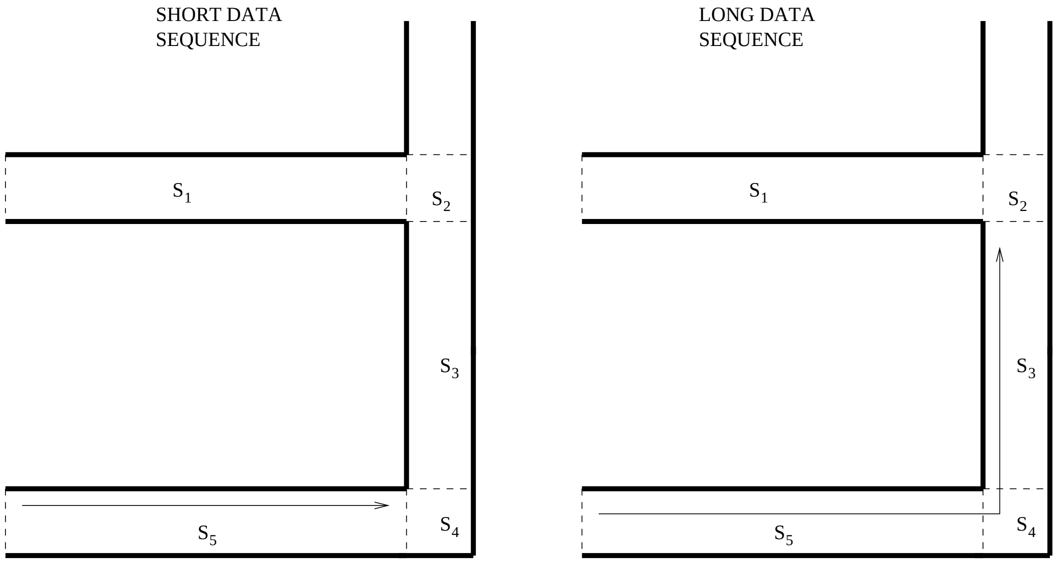

Figure 5.1: When a short training sequence is collected in state S_5, the probability that this sequence occurred in S_1 will be the same as the probability that it occurred in S_5, since S_1 and S_5 are similar. When a long training sequence is obtained from S_5 to S_3, then the probability that this sequence was started from S_1 will be much smaller than the probability that it started in S_5 since the intermediate junction nodes S_4 and S_2 have different observations. Therefore, for short sequences we provide the starting state.

The above realization leads us to create two approximate learning methods that take advantage of the minimum length of the training data. In the first method, which we call *reuse-training*, we collect data for each abstract state separately and train 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, which has similar corridors (abstract states) we can just train one corridor and use the learned model parameters for all other similar corridors. Moreover, once we have trained local abstract states separately of some environment, we can use them as part of any arbitrary global model. Figure 5.2 shows an example HPOMDP model and Figure 5.3 shows how we created the different submodels to be trained.

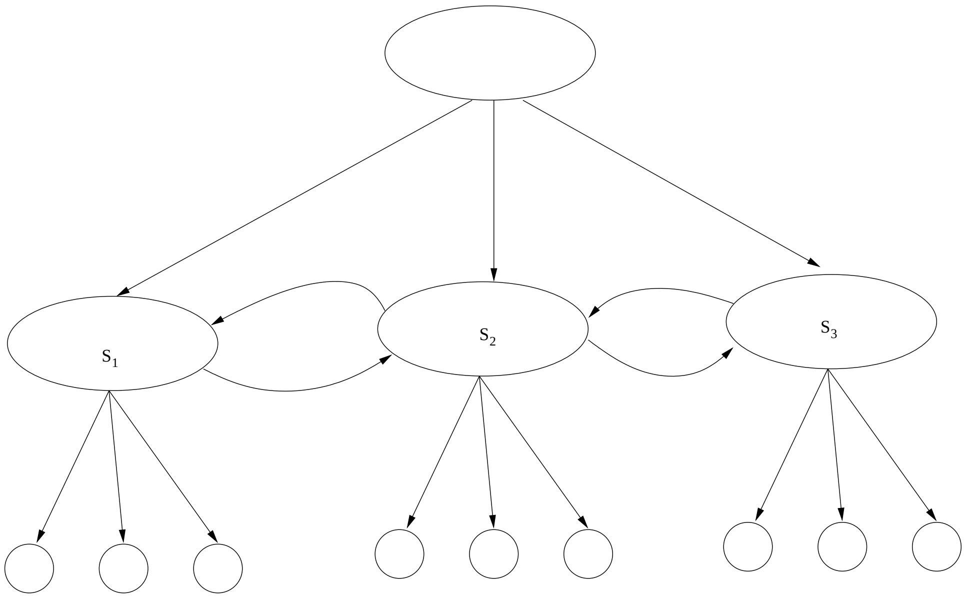

Figure 5.2: An example HPOMDP model for which we illustrate our approximate training methods. For example, in the reuse-training method we train separately each abstract state child of the root (S_1, S_2, S_3) as shown in Figure 5.3

For the second method, which we call *selective-training*, we only compute 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 submodel) 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

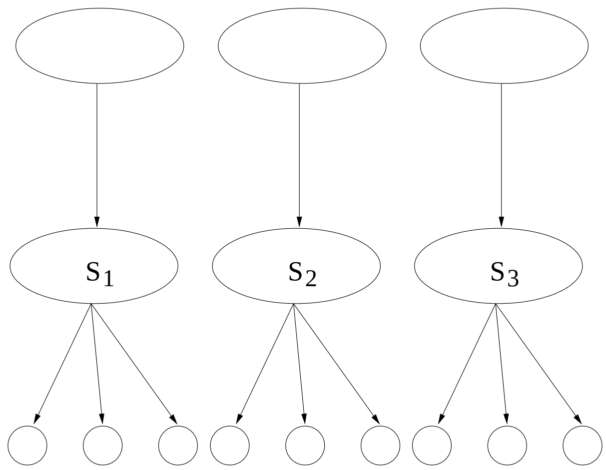

Figure 5.3: Here the submodels of the HPOMDP in Figure 5.2 are trained as separate HPOMDP models. This is the reuse-training method, since the submodels can be used in the complete model in Figure 5.2. In this particular example, we don't need to retrain the whole model in Figure 5.2 (after training separately the various submodels), since the connections between the abstract states are deterministic.

and β of valid states we zero all vertical transitions from the root except the vertical transition to the valid state which the training sequence was initiated from. For each valid abstract state and their 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 3.37, 3.38, 3.39, and 3.40. Figure 5.4 shows a graphical description of the *selective-training* approach.

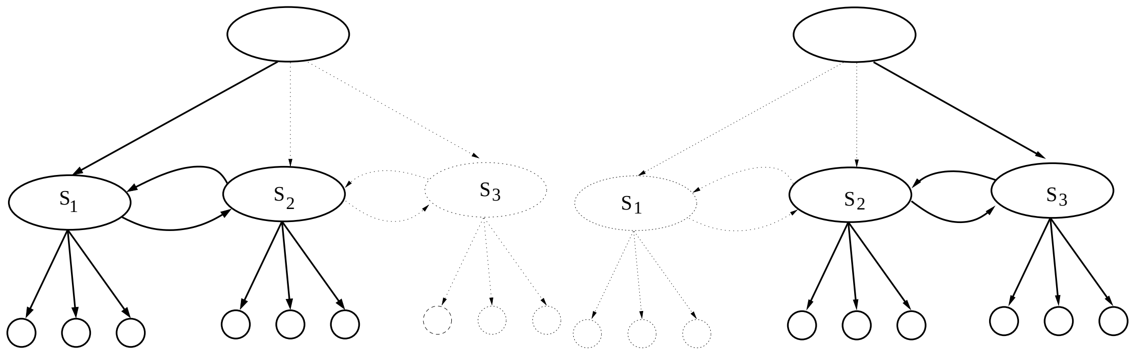

Figure 5.4: In the "selective-training" method only selected parts of the model are trained for each training sequence. Here two data sequences define two valid selections of the whole model which are shown in bold lines.

5.2 Maximum likelihood parameter estimation experiments

5.2.1 Simulated learning experiments

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 simulation environment in Figure 4.3. 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.

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 HPOMDP 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 change 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 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 5 training methods. In the first method, which we name "submodels-only" we simply train the submodels of each abstract state and did not train the model at the abstract level. The second is the "reuse-training" method where we train the submodels separately and then retrain the whole model again. The third method is the "selective-training", the fourth is the standard EM learning algorithm for HPOMDPs, and the fifth 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 3.36 in Chapter 3. 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 HPOMDPs to flat POMDPs. Since a flat POMDP can also be viewed as a Hierarchical POMDP with a single abstract state, the root, then we can compute the log likelihood using Equation 3.36. We used this measure to show both, convergence of the training algorithms, and how the learned HPOMDPs compare in terms of fit to the training data. Plots of the log likelihood for different experiments are shown in Figures 5.5, 5.6, 5.7, 5.9, 5.8, and 5.10.

Distance was measured according to Equation 5.1 [10]. Equation 5.1 is 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 my 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 also be interpreted in other ways including cross entropy, divergence, and discrimination information [100].

$$D(\lambda_1, \lambda_2) = \frac{\log P(Z_2|A_2, \lambda_1) - \log P(Z_2|A_2, \lambda_2)}{T} \tag{5.1}$$

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.

In Table 5.1 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 [22]. To compute the most likely state we first convert the HPOMDP learned to a flat POMDP as was shown in Section 3.2.3. 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 was shown in Equation 2.14 in Section 2.2.1 and 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 $s_{MLS} = \text{argmax}_s b(s)$. Since our test data is created by sampling some good HPOMDP 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 sequence we average over all the errors. Table 5.2 shows robot localization results for the different models trained.

In passing, we should node 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 [101], [10]. 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 original EM learning algorithm for HPOMDPs and the original EM learning algorithm for POMDPs. Such results are shown in Table 5.3

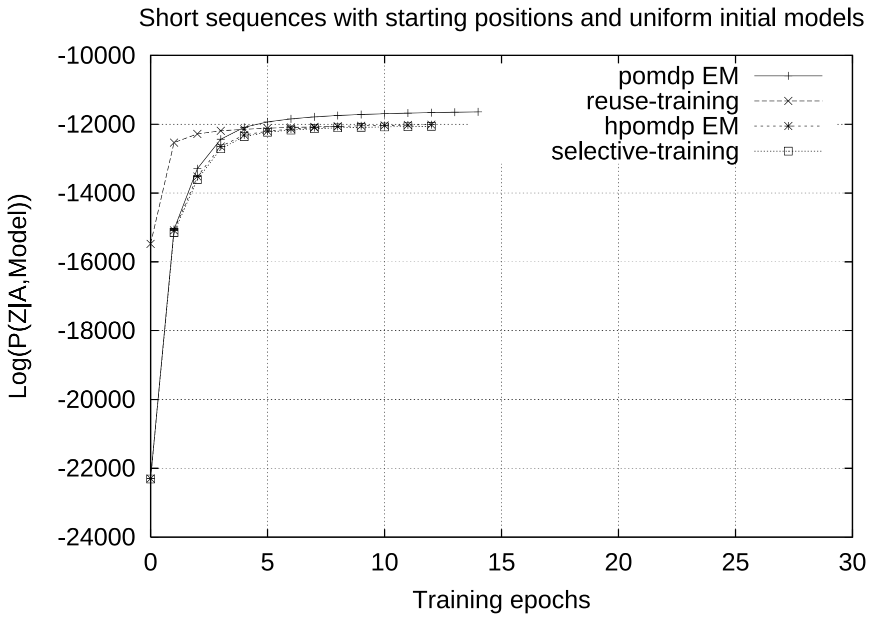

Figure 5.5: The graph describes the fit to the training data during experiment 1. The vertical axis represents the log likelihood over all 738 training sequences. The "reuse-training method" converges faster than all other methods. The EM learning algorithm for POMDPs fits the training data slightly better than the hierarchical methods.

From Tables 5.1 and 5.2, and from graphs 5.5, 5.6, 5.7, 5.9, 5.8, and 5.10. we 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).

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	Figure (5.5)	submodels-only	-0.40, (0.21)
		738,2,yes		reuse-training	**-0.29, (0.18)**
				selective-training	-0.34, (0.20)
				EM for hpomdps	-0.35, (0.21)
				EM for pomdps	-0.80, (0.35)
					Experiment 2
good	-0.95, (0.18)	232,1,yes	Figure (5.6)	submodels-only	-0.38, 0.20
		738,2,yes		reuse-training	**-0.30, (0.19)**
				selective-training	**-0.30, (0.19)**
				EM for hpomdps	**-0.30, (0.19)**
				EM for pomdps	-0.49, (0.25)
					Experiment 3
uniform	-1.3, (0.2)	246,3,no	Figure (5.7)	reuse-training	**-0.72, (0.31)**
				EM for hpomdps	-1.93, (0.83)
				EM for pomdps	-2.15, (0.99)
					Experiment 4
good	-0.95, (0.18)	246,3,no	Figure (5.8)	reuse-training	**-0.72, (0.32)**
				EM for hpomdps	-0.82, (0.35)
				EM for pomdps	-0.85, (0.33)
					Experiment 5
uniform	-1.3, (0.2)	246,3,yes	Figure (5.9)	reuse-training	**-0.48, (0.26)**
				EM for hpomdps	-0.76, (0.30)
				EM for pomdps	-1.01, (0.41)
					Experiment 6
good	-0.95, (0.18)	246,3,yes	Figure (5.10)	reuse-training	**-0.45, (0.24)**
				EM for hpomdps	-0.5, (0.26)
				EM for pomdps	-0.61, (0.21)

Table 5.1: The table shows the mean and average of the distance of the leaned 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 4.3. The bold numbers in the last column show the shortest distance for each experiment. The "reuse-training method" produces the best results. The EM learning algorithm for HPOMDPs always produces better models than the EM learning algorithm for POMDPs.

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)
			EM for hpomdps	37.30, (21.62)
			EM for pomdps	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)**
			EM for hpomdps	33.2, (19.95)
			EM for pomdps	49.17, (22.5)
				Experiment 3
uniform	89.2, (10.0)	246,3,no	reuse-training	**36.69, (20.83)**
			EM for hpomdps	86.02, (10.59)
			EM for pomdps	87.81, (8.85)
				Experiment 4
good	55.38, (23.88)	246,3,no	reuse-training	**36.97, (20.26)**
			EM for hpomdps	43.89, (23.07)
			EM for pomdps	48.62, (23.48)
				Experiment 5
uniform	89.2, (10.0)	246,3,yes	reuse-training	**34.76, (20.95)**
			EM for hpomdps	60.45, (23.06)
			EM for pomdps	69.55, (20.99)
				Experiment 6
good	55.38, (23.88)	246,3,yes	reuse-training	**35.17, (20.29)**
			EM for hpomdps	36.60, (20.28)
			EM for pomdps	45.47, (22.25)

Table 5.2: 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. The EM learning algorithm for HPOMDPs always produces better models than the EM learning algorithm for POMDPs.

Figure 5.6: The graph describes the fit to the training data during experiment 2. The fit to the training data is similar to experiment 1, even though training started with a better initial model.

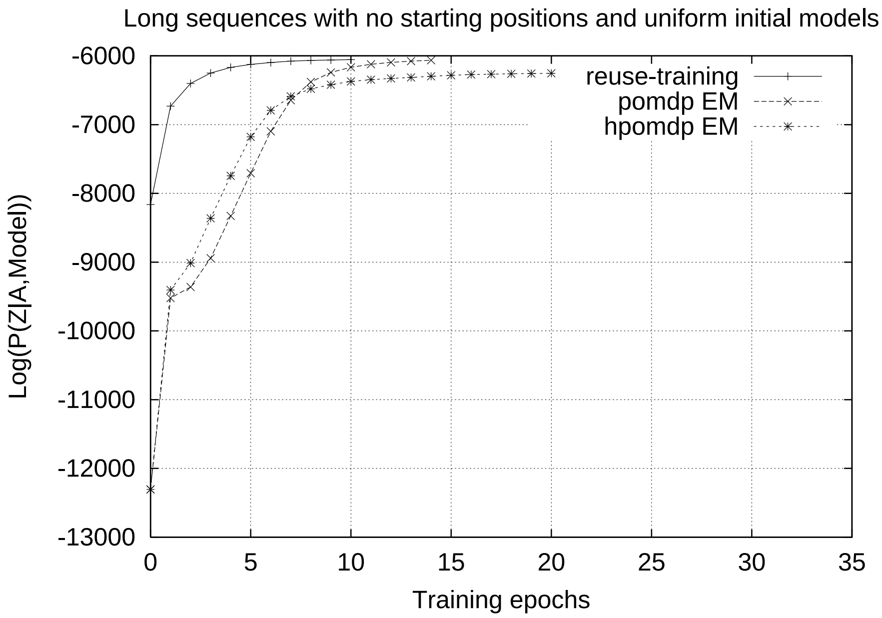

Figure 5.7: The graph describes the fit to the training data during experiment 3. The "reuse-training" method clearly converges faster than the other training methods.

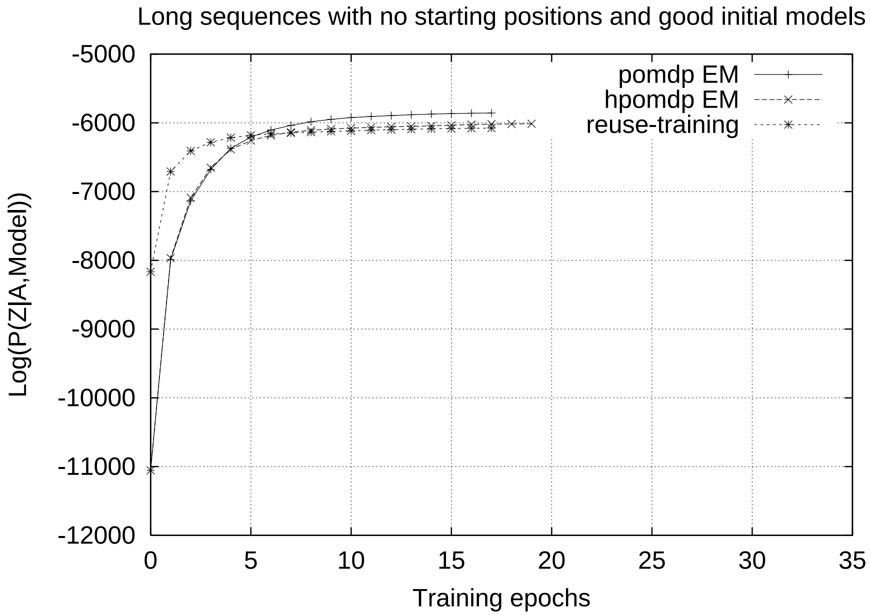

Figure 5.8: The graph describes the fit to the training data during experiment 4. The fit to the data is better than experiment 6 (see Figure 5.10). This is because data is allowed to start from all states. Nevertheless, navigation performance is worse than experiment 6 according to Table 5.2.

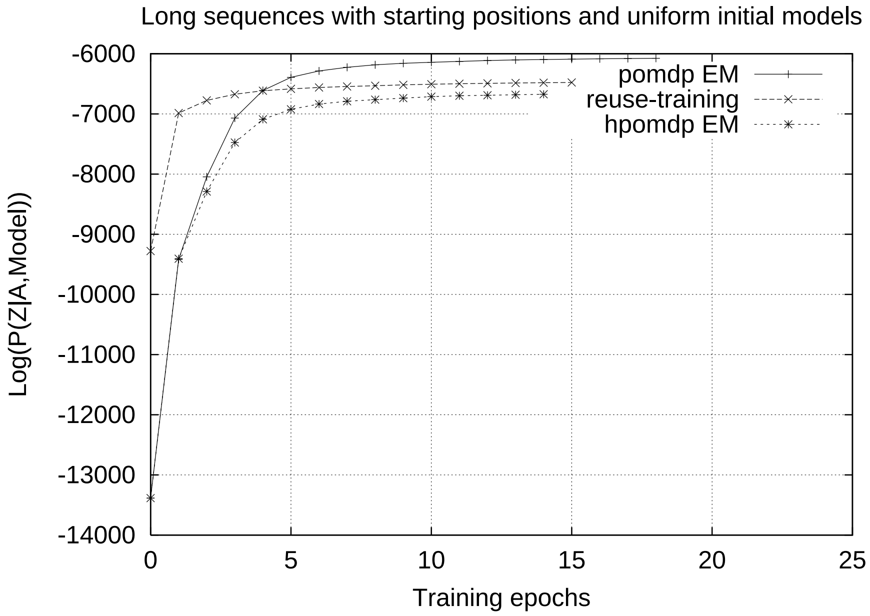

Figure 5.9: The graph describes the fit to the training data during experiment 5. The POMDP model fits the training data the best while the "reuse-training" method converges faster than all other methods.

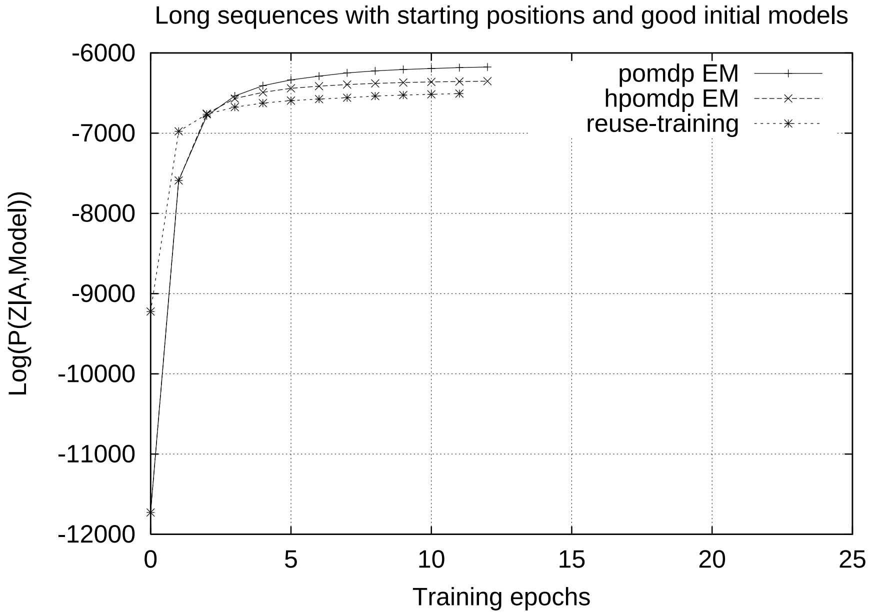

Figure 5.10: The graph describes the fit to the training data during experiment 6. Even though the fit to the data is not as good as in experiment 4 (see Figure 5.8), navigation performance was better. The EM learning algorithm for POMDPs fits the training data the best.

- From Tables 5.1 and 5.2 we can conclude that the shorter the distance of the learned models from the original model, the better the robot localization.

- Almost all of the plots of the log-likelihoods during training show that the EM learning algorithm for POMDPs fits the training data better. Nonetheless, during testing the models learned 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 good 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 HPOMDPs (see Table 5.3).

- It is obvious from Tables 5.1 and 5.2 that the learned models never reach the performance of the original model. This is due to the fact that EM type algorithms converges 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.

- Providing the initial state the data was collected from, 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 number of 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 5.3 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 $O(NT)$. This is is due to the hierarchical representation of the state space which makes the N term in the hierarchical methods smaller than the N term in the flat methods.

Experiment	Training method	Time (sec)/epoch
Figures 5.5, 5.6	selective	449
	flat	3722
	hierarchical	1995

Table 5.3: 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

5.2.2 Learning experiments using the robot platforms

In this set of experiments we trained hierarchical POMDP models for the small floor (Figure 4.3), using robot experience from the Nomad 200 simulator, and experience with the real robot Pavlov from the actual 3rd floor in Figure 4.4. The purpose of these experiments was to investigate if learned models can actually be used for robot navigation.

For both environments we simply trained all submodels of the abstract states separately. To collect data we run 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.

To evaluate the learned models we used some known "good" original model, which we compare to the "learned" model. For the "good" original model, we made all sensor probabilities of perceiving the correct observation to be 0.9 and the probability of perceiving the wrong observation to be 0.1. All self transitions were set to 0.05 and transitions to the correct state were set to 0.9. In these experiments, for all corridor states we also added an overshoot transition, set to 0.05. Learning started from a "bad" model which was created from the "good" original model by increasing the self transitions to 0.47 and decreasing the transitions to the correct state to be 0.47. We evaluated the learned models based on robot localization.

Localization performance was computed different this time. Since 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 5.4. It is obvious that after training, the learned models produce low error robot localization. Also note that unlike Table 5.2 in the previous section, the localization error here is quite low. The reason is that the training and testing data came from actual robot runs. Actual robot navigation tasks produce similar training and testing sequences of observations and actions (e.g traversing of corridors). In the previous section the data was collected trough sampling which could have produced training sequences quite different from the testing sequences. Additionally, the low localization error is due to the fact that the initial observation models provided 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. 4.3)	14 seq. from Nomad 200 simulator (Fig. 4.3)	41.8, (21.4)	11.8, (18.0)
	43 seq. from Pavlov in real floor (Fig. 4.4)	7 seq. from Pavlov in real floor (Fig. 4.4)	64.9, (20.9)	2.8, (3.8)

Table 5.4: 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.

5.3 Inferring the structure of the environment

We also did experiments to discover whether the HPOMDP model has the ability to infer the structure of the environment. If at an abstract level we don't have the correct topology of the environment, then we would expect the HPOMDP learning algorithm to perform much better in learning the correct topology than the flat POMDP learning algorithm. The reason is that the Hierarchical-Baum algorithm does not learn the relationship between two abstract states (which are root children) unless the training data goes all the way to the exit-state of 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 (or $\xi(t, s_x, s'_n)$) in Equation 3.24. 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 better understood in Figure 5.11.

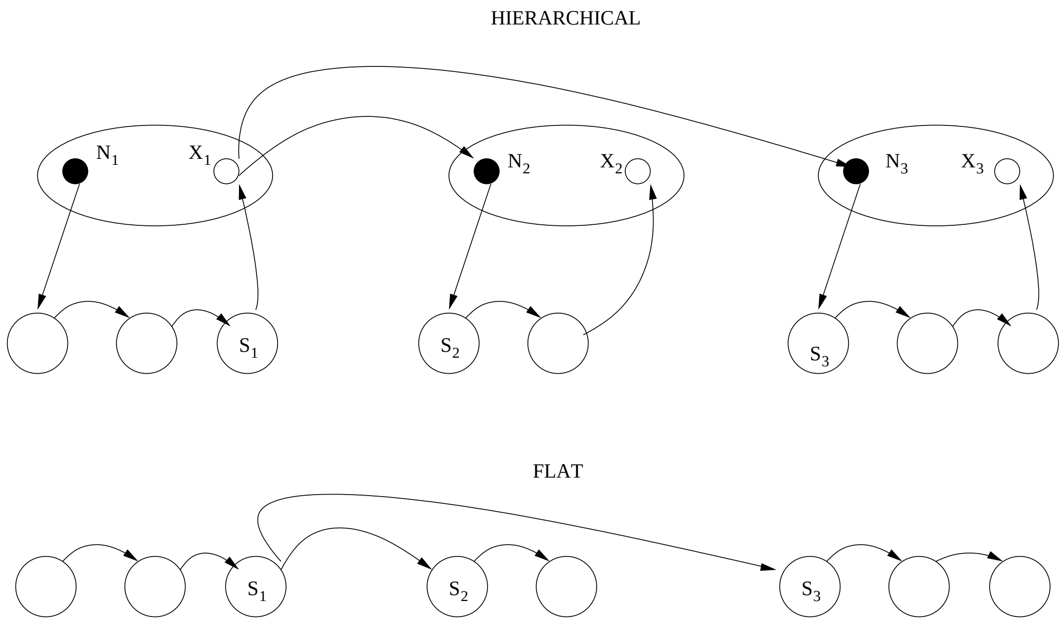

Figure 5.11: Assume that an observation sequence of length 5 is started from entry state N_1 and that, in the correct topology of the environment the transition probability $X_1 \rightarrow N_3$ is zero. In the hierarchical model the topology will be learned correctly since in order for the transition $X_1 \rightarrow N_3$ 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.

5.3.1 Learning the structure of a three level hierarchy

For the simulated environment in Figure 4.3 we trained an HPOMDP 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 HPOMDPs, 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 5.12.

2. A hierarchical model, "model-2", 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 abstract state is connected to a every entry point of 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 in Figure 5.13 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 according to Equation 5.2.

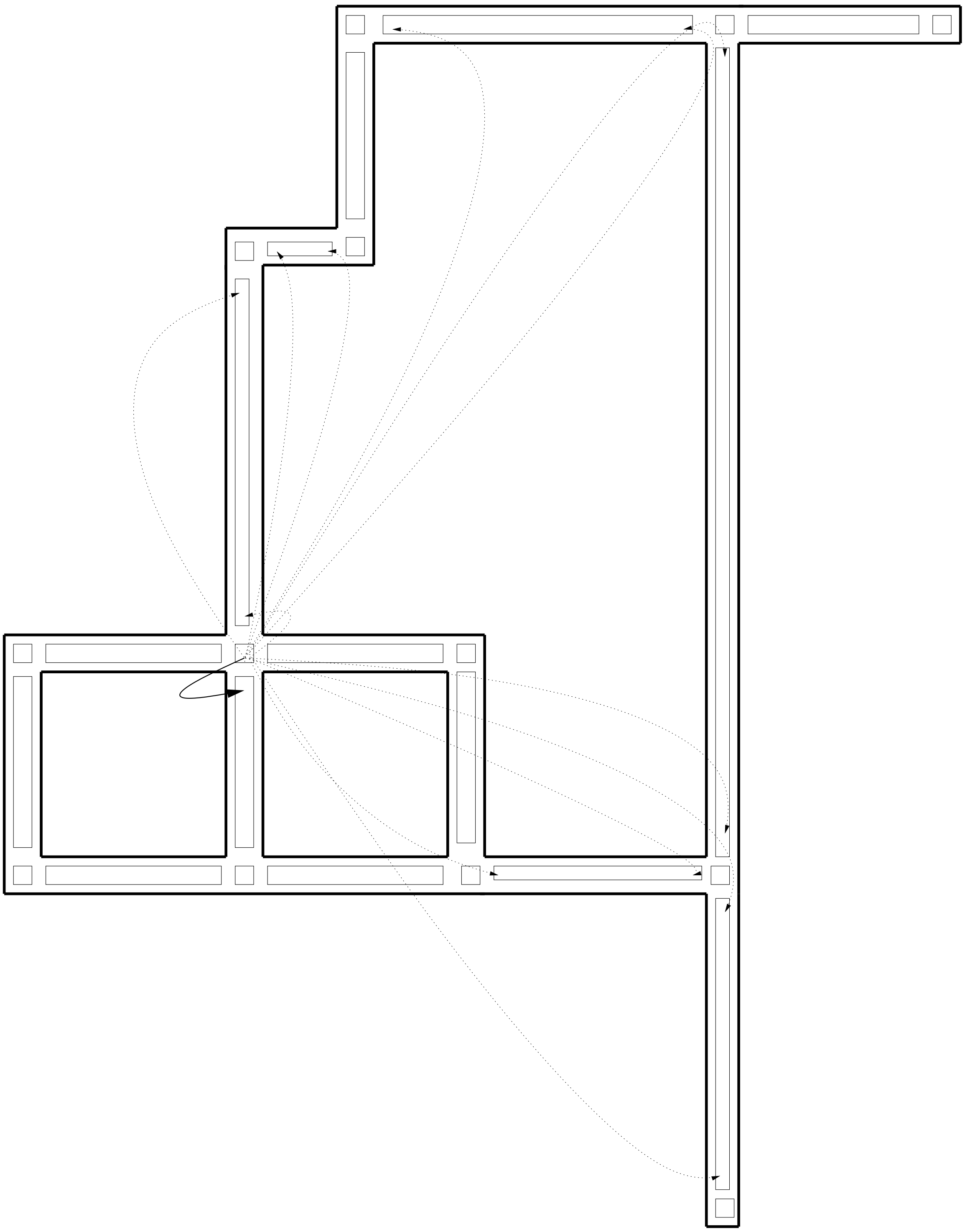

Figure 5.12: The bold arrow indicates the correct structure for the particular south exit point of the junction abstract state. The dotted arrows indicate possible adjacent nodes. We chose adjacent nodes to be corridors of different size from the correct corridor.

$$\text{correct structure} = \frac{\sum_{s \in S, a \in A} \sum_{s' \in S'} T(s'|s,a)}{\sum_{s \in S, a \in A} \sum_{s'' \in S''} T(s''|s,a)} \tag{5.2}$$

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 5.13 over the total transition probability associated with all the arrows (all arrows starting from the same positions as the arrows shown in Figure 5.13). Table 5.5 summarizes the results of all the models, and shows that the hierarchical models improve the structure much more than the flat models. Figures 5.13 and 5.14 show the transition probabilities of the correct structure before and after learning for initial "model-1".

We also plotted the log likelihood ($logP(Z|, A, \lambda)$) as a function of the training steps as shown in Figures 5.15 and 5.16. The graphs show that that the flat models fit the data better. Its also obvious that even though, "model-1" is less ergodic, "model-2" fits the data better. This suggests that for the short length data sequences better fit to the data does not necessarily mean better navigation model. In addition, even though the flat model fit the training data better than then hierarchical models, the

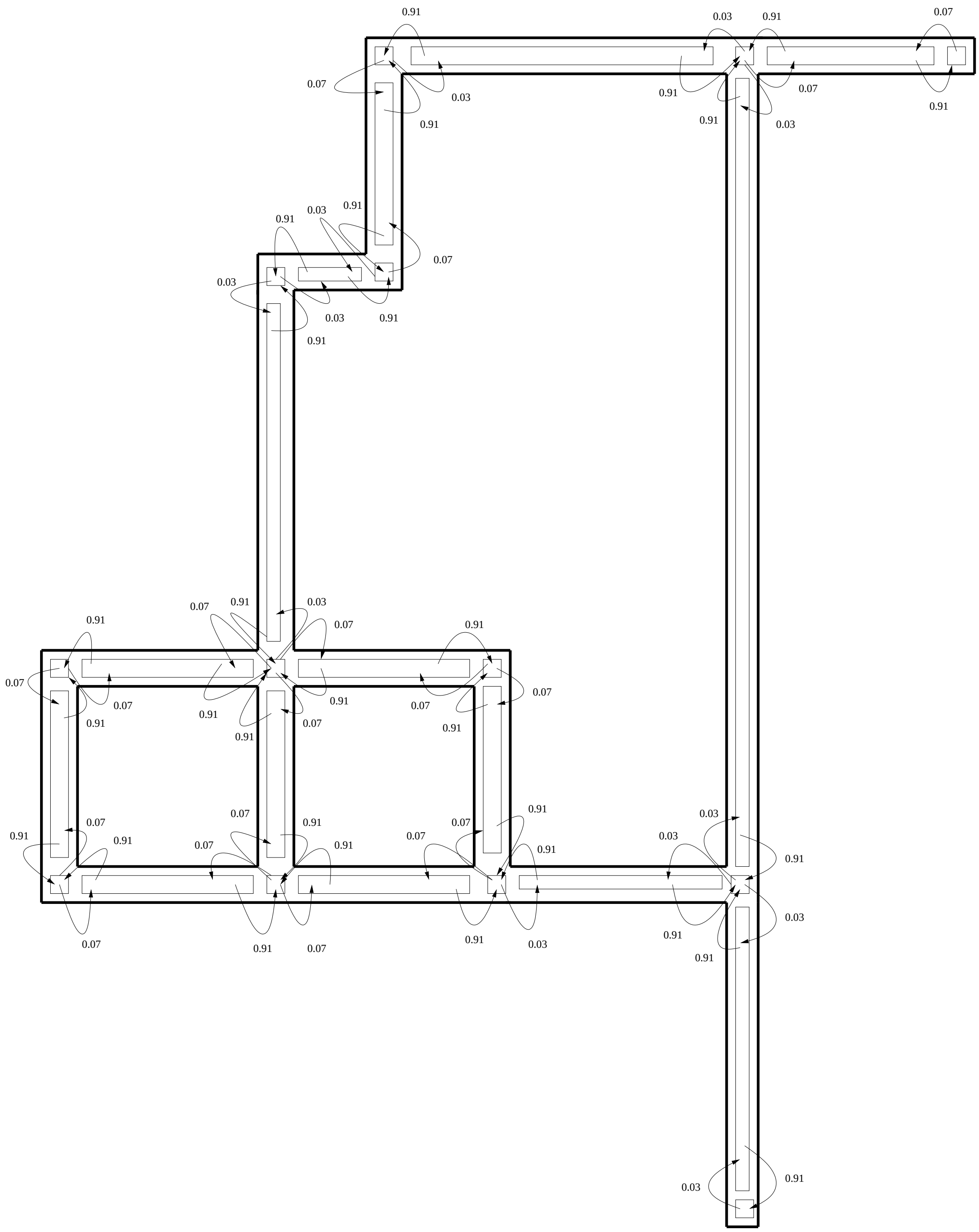

Figure 5.13: The figure shows the transition probabilities that belong to the *correct* structure before training for "model-1". The probabilities shown are the ones corresponding to the HPOMDP model when converted to flat.

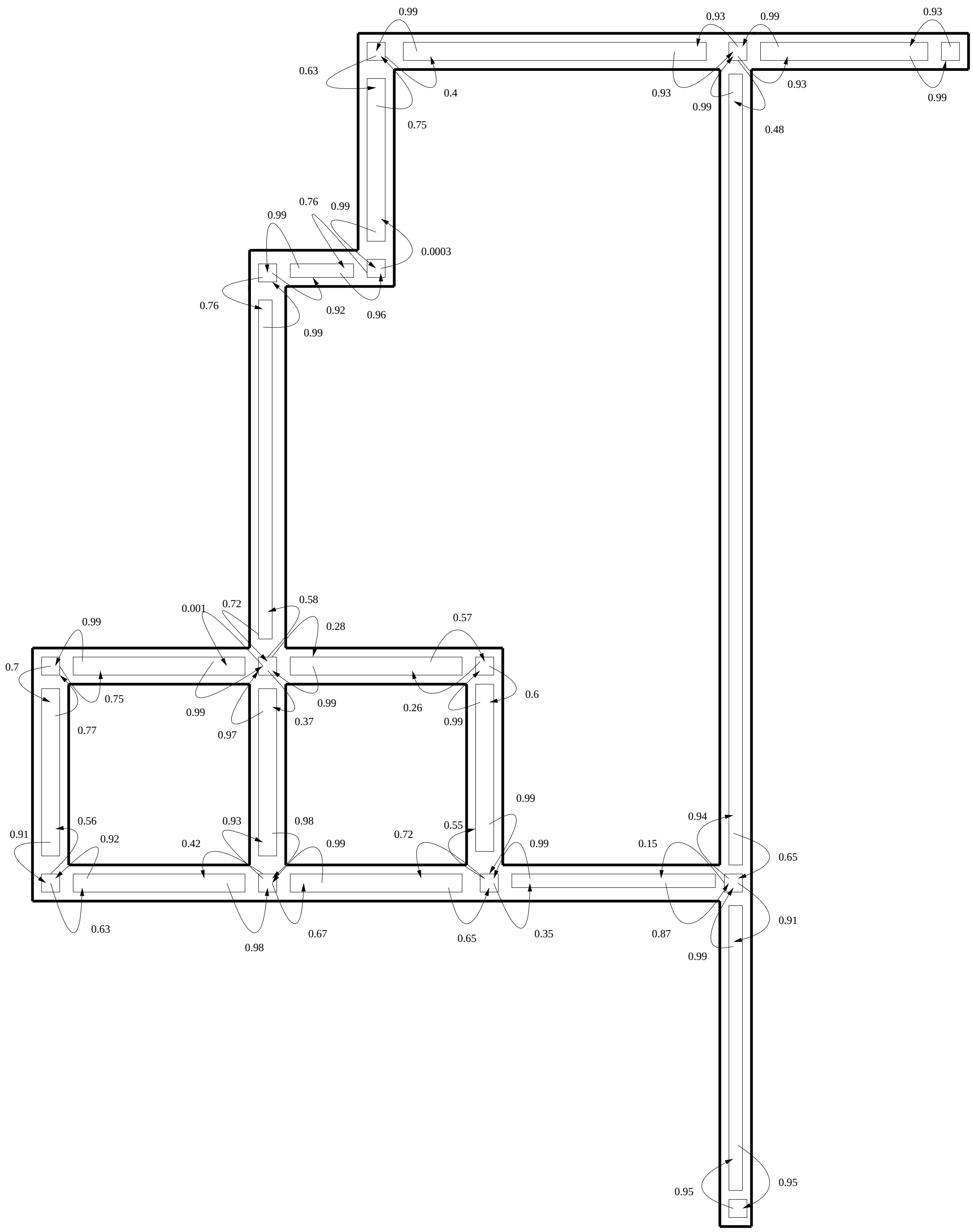

Figure 5.14: The figure shows the transition probabilities that belong to the *correct* structure after training for "model-1", using the "selective-training" method. The probabilities shown are the ones corresponding to the HPOMDP model when converted to flat.

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

Table 5.5: 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.

hierarchical models produces less error in the correct structure in Table 5.5.

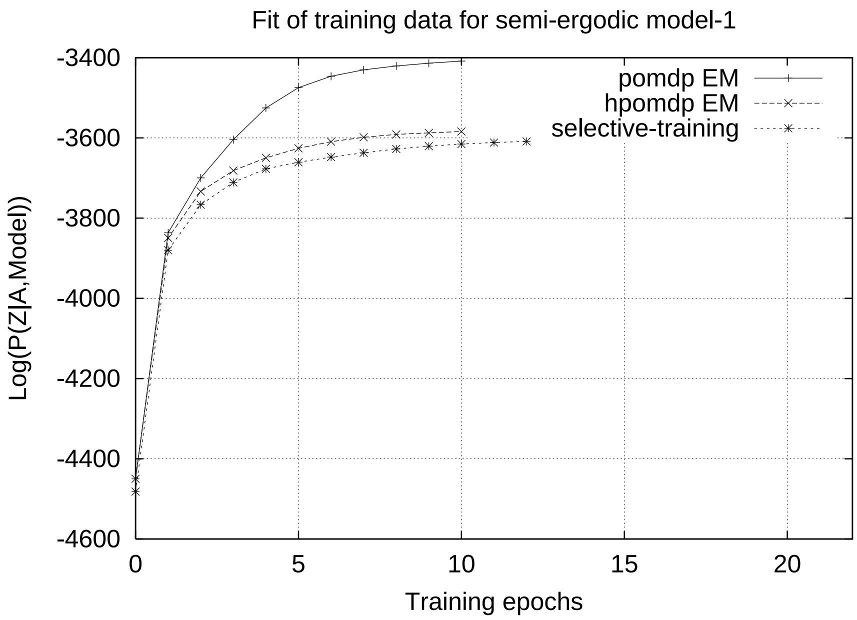

Figure 5.15: The figure shows how “model-1” fits the data over every training epoch, and for the different training methods. The POMDP model fits the training data better than the HPOMDP models.

5.3.2 Learning the structure of a four level hierarchy

We also did structure learning experiments on a four level hierarchical model. The second and third level are shown in figure 5.17. In this set of experiments we started

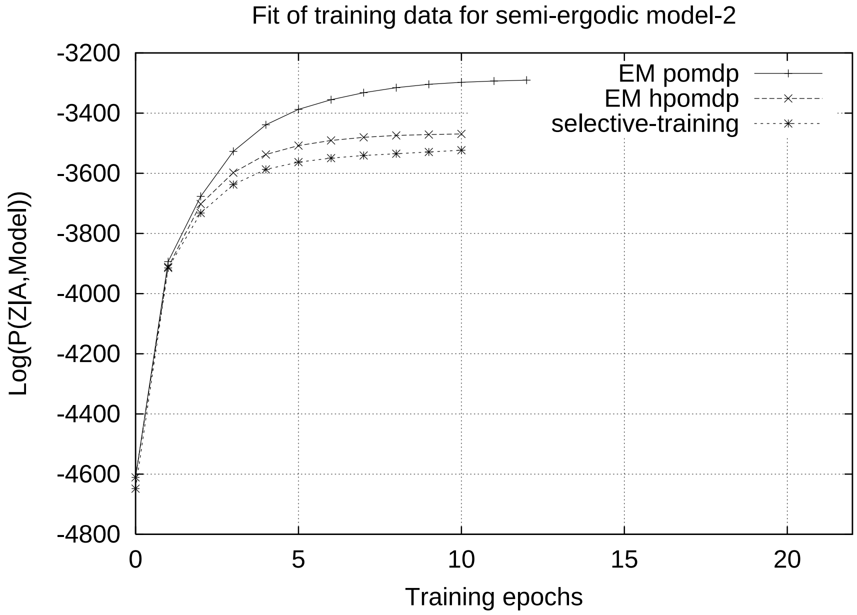

Figure 5.16: The figure shows how "model-2" fits the data over every training epoch, and for the different training methods. The POMDP model fits the data better than the HPOMDP models.

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 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 5.3.1).

Table 5.6 summarizes the result for the different initial models and different training methods while Figures 5.20 and 5.21 show how the different methods and different models fit the data during training. In this experiment, 17 long training sequences were used. For model "4level-1", both the EM learning algorithm for HPOMDPs 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 HPOMDPs 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. Figures 5.19, and 5.18 show the correct structure learned at level 2, for model "4level-2", using the EM learning algorithm for HPOMDPs and the EM algorithm for learning POMDPs.

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 %	EM for hpomdps	13.25 %	0.22 %
			EM for pomdps	12.31 %	0.20 %
4level-2	82.46 %	83.61 %	EM for hpomdps	**32.50** %	**49.90** %
			EM for pomdps	43.85 %	55.34 %

Table 5.6: 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.

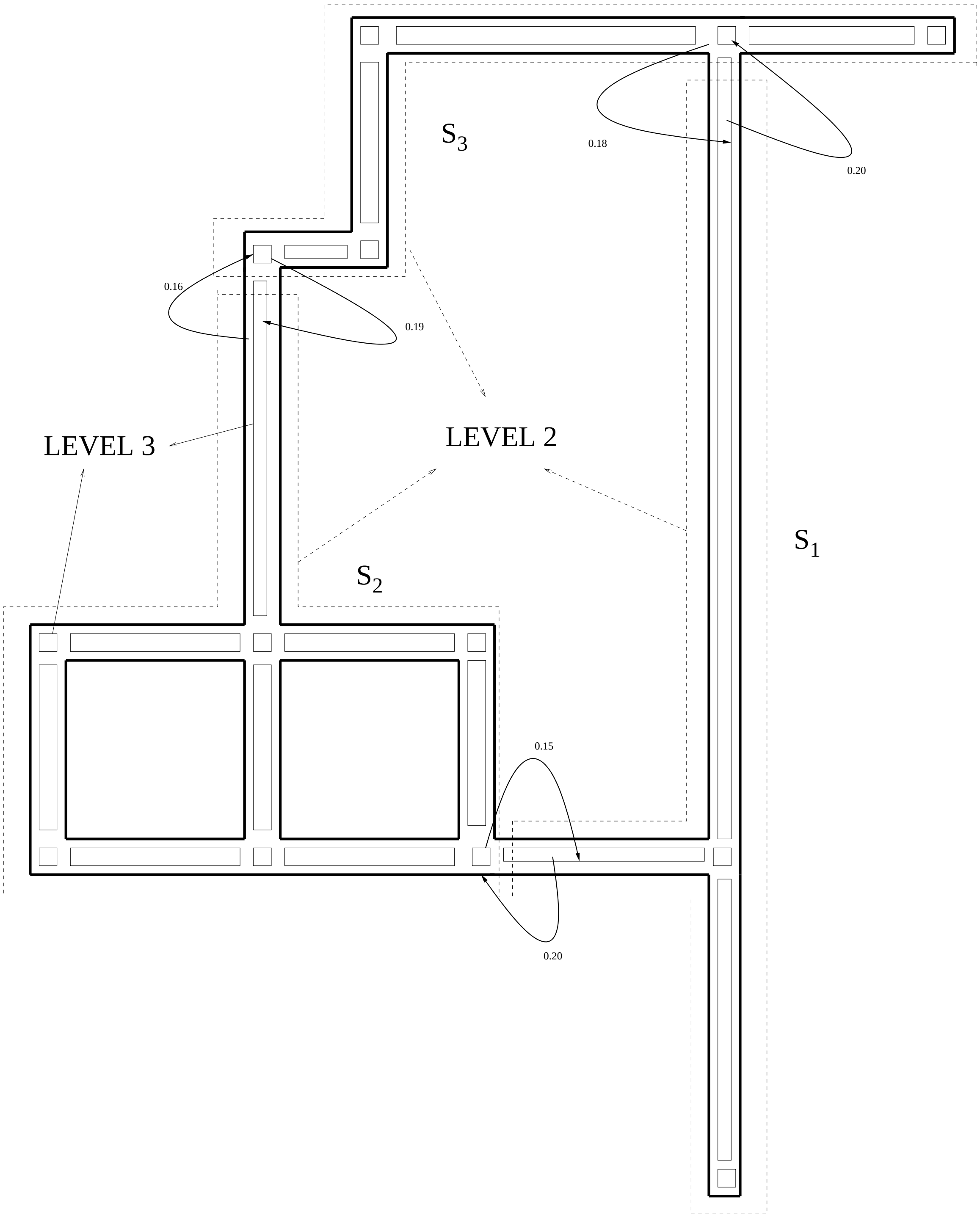

Figure 5.17: This is a four level representation of the environment where entire sections of the building where grouped into abstract states at a higher level. The three large dashed polygons represent the abstraction at level 2 while the smaller rectangles inside the polygons represent the spatial abstraction 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.

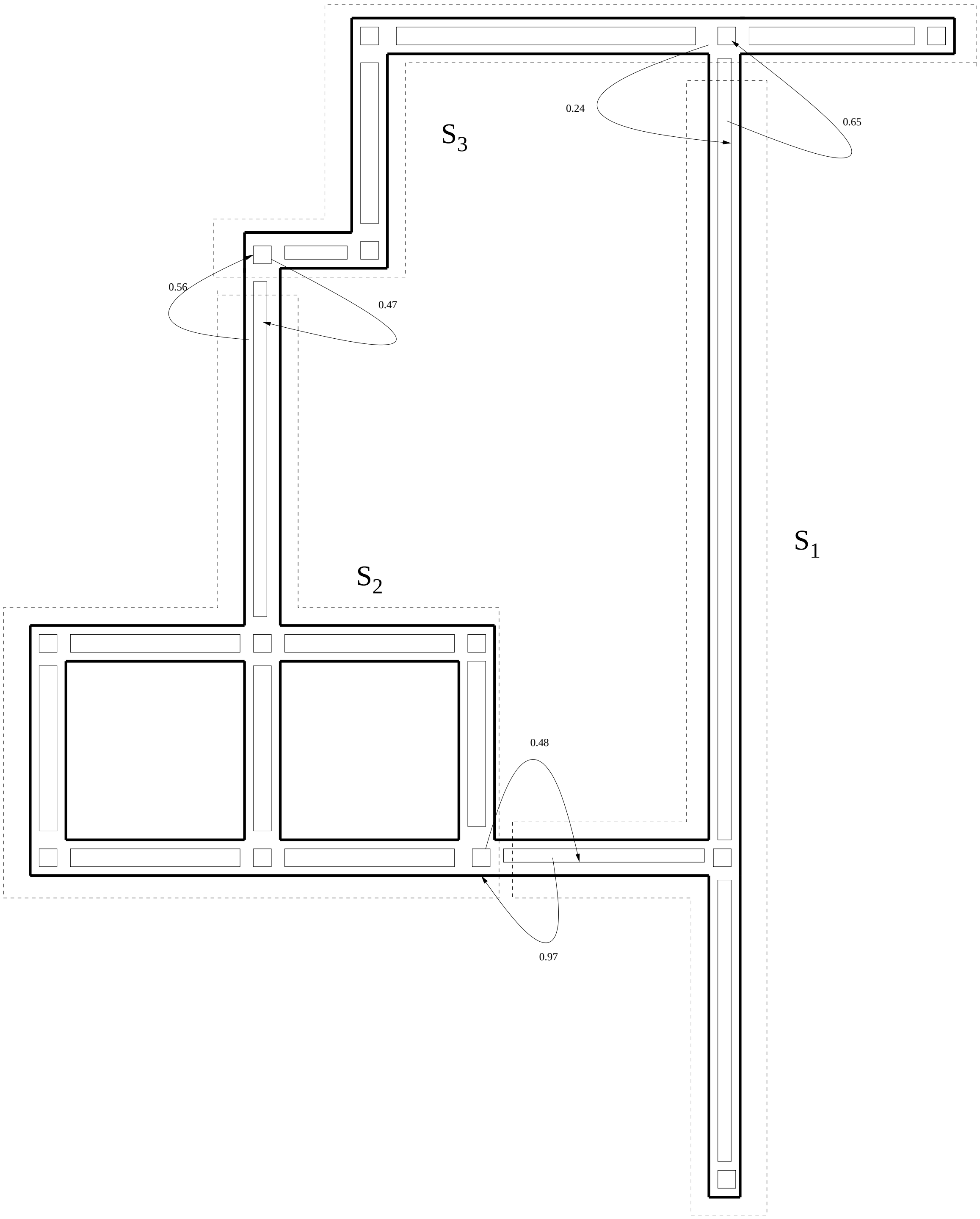

Figure 5.18: The arrows show the transition probabilities of the correct structure, after training model “4level-2” with the EM learning algorithm for POMDPs.

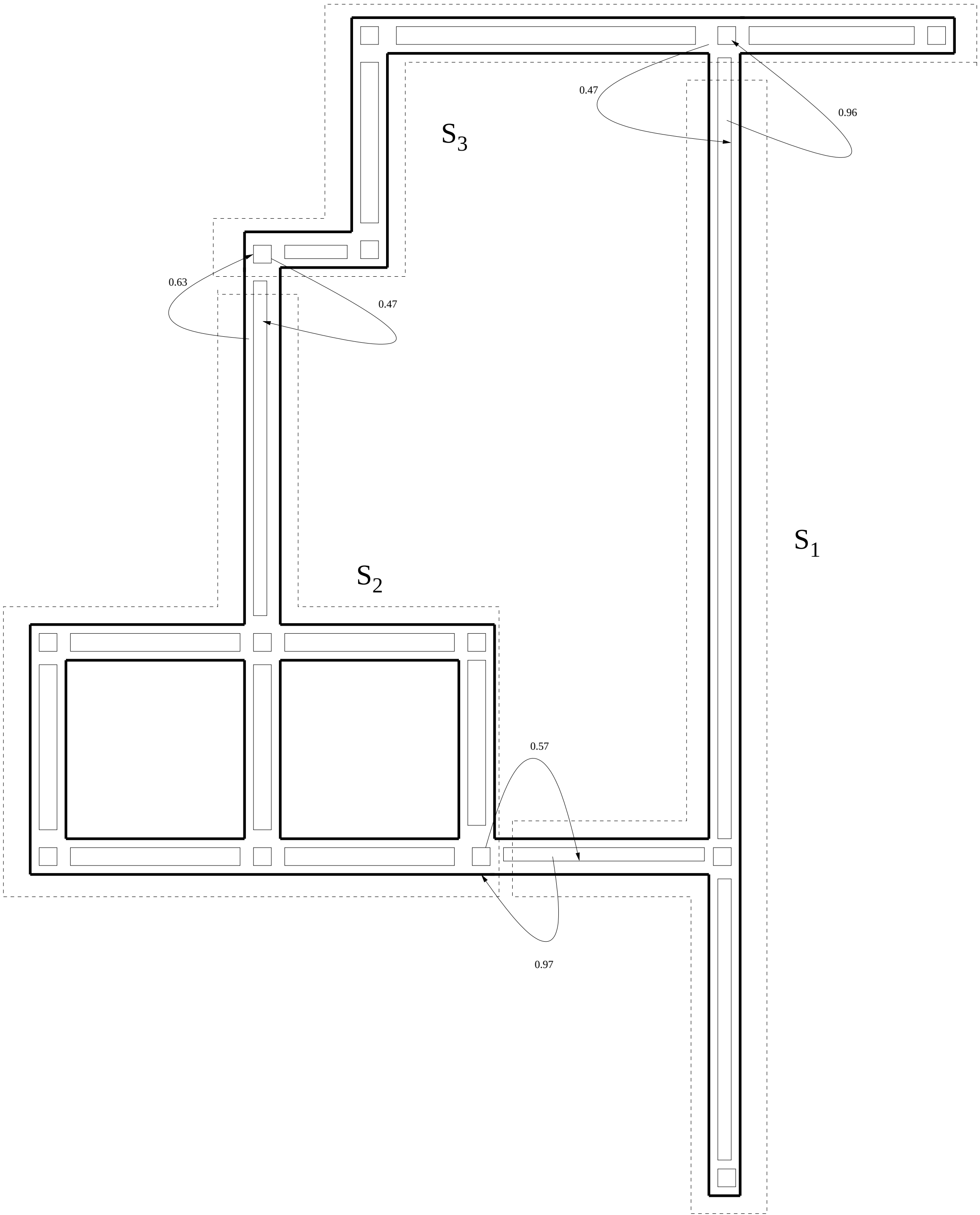

Figure 5.19: The arrows show the transition probabilities of the correct structure, after training model “4level-2” with the EM learning algorithm for HPOMDPs.

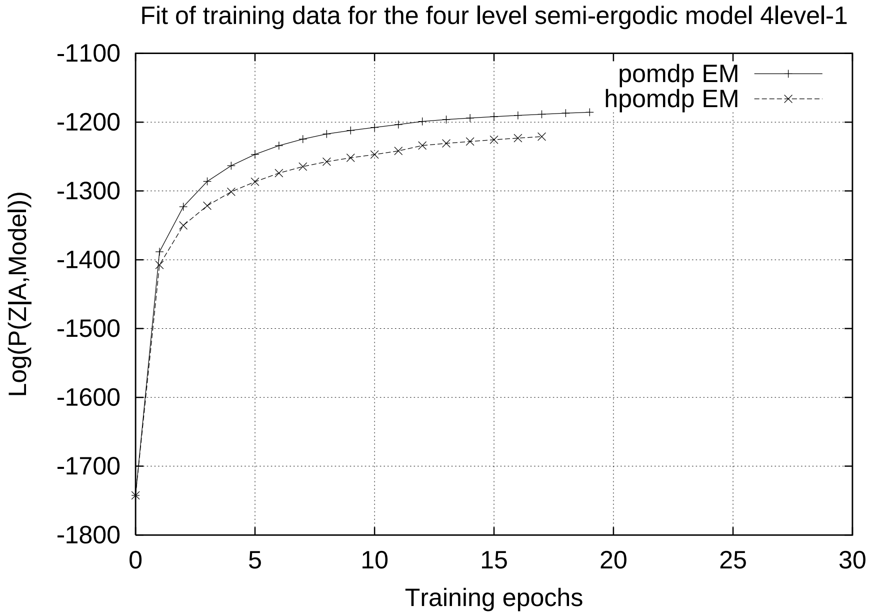

Figure 5.20: The graph shows the fit to the data during training of model “4level-1”.

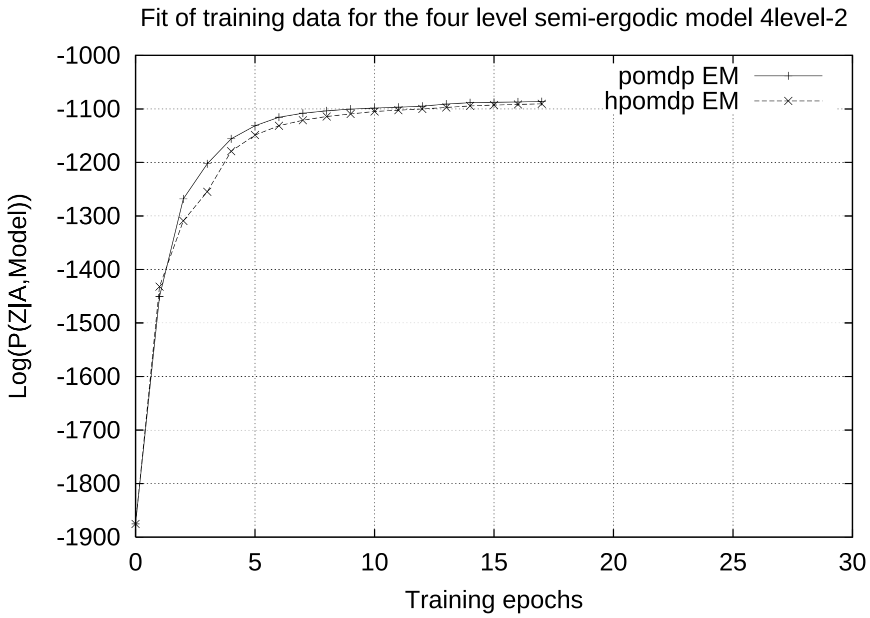

Figure 5.21: The graph shows the fit to training data during training of model “4level-2”. In the graph, the EM algorithm for learning HPOMDPs, and the EM algorithm for learning POMDPs fit the data better than the plots in Figure 5.20 despite the fact that the models here are more ergodic.

5.4 Conclusions

This chapter has described learning experiments, where we used the HPOMDP model to represent spatial maps for indoor office environments. Taking advantage of short training length sequences we have created two approximate training methods. The "reuse-training" method which starts with better initial models and as a result converges faster, and the "selective-training" method which takes a lot less time per iteration than any of the other methods. The two approximate training methods concentrate on smaller parts of the whole model which results in much better models for robot navigation. Advantages of training with smaller training data sequences is that first, it is easier to collect more complete training data sets and second, training is done much faster. A disadvantage of the approximate training methods is that they require knowledge as to where the data was collected from. Additionally, the training sequences need to terminate at some root child.

The experimental results show that our hierarchically learned HPOMDP models are closer to the generative model, from which the training data is produced, than the models learned through the flat EM procedure for POMDPs. Additionally, the hierarchically learned models produce better robot localization than the models learned through the flat EM algorithm for POMDPs.

Overall, the hierarchical Baum-Welch algorithm is better at capturing the relationship between states which are far apart and as a result more meaningful models are learned for use in robot localization. Additionally, the hierarchical Baum-Welch is better in acquiring the initial structure. This algorithm is well suited for domains which emit hierarchical data, such as data sets that start from the root and finish at a child state of the root. In the following chapter we will show how to use the hierarchical POMDP model to perform navigation tasks. We will show hierarchical planning algorithms for approximating the optimal policy in a given HPOMDP and their comparison to flat POMDP methods in robot navigation.

Chapter 6

Hierarchical planning for robot navigation

The HPOMDP model inspires a hierarchical framework for planning, which has two major advantages over flat POMDP solutions. The first advantage is that plans can be constructed in a hierarchical fashion and thus allow us to reuse low level plans in different high level plans. The second advantage is that at higher levels of abstraction, both spatial and temporal, uncertainty is less and therefore plan execution can be done more successfully than in flat POMDPs.

In this chapter we describe a general hierarchical planning algorithm for HPOMDPs and apply the framework to corridor navigation to produce and execute hierarchical plans [102]. At an abstract level we construct abstract plans by computing mappings from abstract states (e.g whole corridors) to macro-actions (e.g., go down the corridor) and within abstract states we compute the macro-action plans which are mapping from product states to primitive actions. Examples of macro-actions within abstract states include "exit a corridor" at one of its exit states, "exit a junction" at one of its exit states, or reach a particular location within some abstract state. Our experiments show that there is less uncertainty associated with the belief states defined over higher levels of the HPOMDP hierarchy. The combination of low entropy at the abstract level, macro-actions, and heuristic POMDP solutions (e.g., choose the action associated with the most likely state) allow the robot to reach locations of its environment starting from no knowledge of its original starting location.

In Section 6.1 we motivate our planning algorithm. In Section 6.2 we review related literature, which forms the building blocks of our planning algorithm. In Section 6.3 we present our planning, and two execution algorithms. In Section 6.4 we present robot navigation results, both in the Nomad 200 simulator and in the real physical world. Finally in Section 6.5, we present our conclusions.

6.1 Robot navigation in corridor environments

It was obvious from Figure 1.4 in Chapter 1 that corridors in the MSU engineering building are quite long (more than 1000 ft.) and locations within the corridors do not have distinguishing characteristics for a robot to localize. However, this problem can be alleviated if we treat whole corridors as single states and thus abstract away information that may be unnecessary for most navigation tasks. The reason is that for most navigation tasks a robot just needs to know in which corridor it is in, rather than the precise corridor location, and therefore it can often ignore location uncertainty that arises within corridors.

Figure 6.1 shows an HPOMDP model for robot navigation and figure 6.2 shows the equivalent flat model. Both models are used to represent corridor environments. The environment is divided into equal fixed regions (e.g two meter regions), where there are four product states (one for each direction). In the HPOMDP model whole corridors are represented with a single abstract state, which in addition to the product states contains two entry-states and two exit-states for the entry and exit points of the corridor. Junction nodes are represented with four product states as shown in Figure 6.1. In the HPOMDP model for corridor environments, we use the term *global product level* to denote the set of all product states, and *abstract level* to denote all the states which are children of the root.

A multi-resolution representation is intuitive for many reasons. First of all, it allows us to scale up to larger environments since we do not need to represent all relationships among product states. Second, for many navigation tasks the robot does not need to know where it is in a corridor but rather that it is in some corridor which it has to traverse. Third, it offers us the capability of learning local policies (or macro-actions as shown in Figure 6.7) for the different abstract states, which we can reuse for any global plan at the abstract level. Finally, as shown in Figure 6.4, the robot's belief (see Equations 6.20 and 6.21) as to where it is at the abstract level, represented as a probability distribution over the states at the abstract level, is less uncertain than its belief as to where it is at a global product level.

The uncertainty of the robots belief about its true location at the abstract and global product level was measured using the normalized entropy. The normalized entropy of an $|S|$ dimensional discrete probability distribution can be computed as

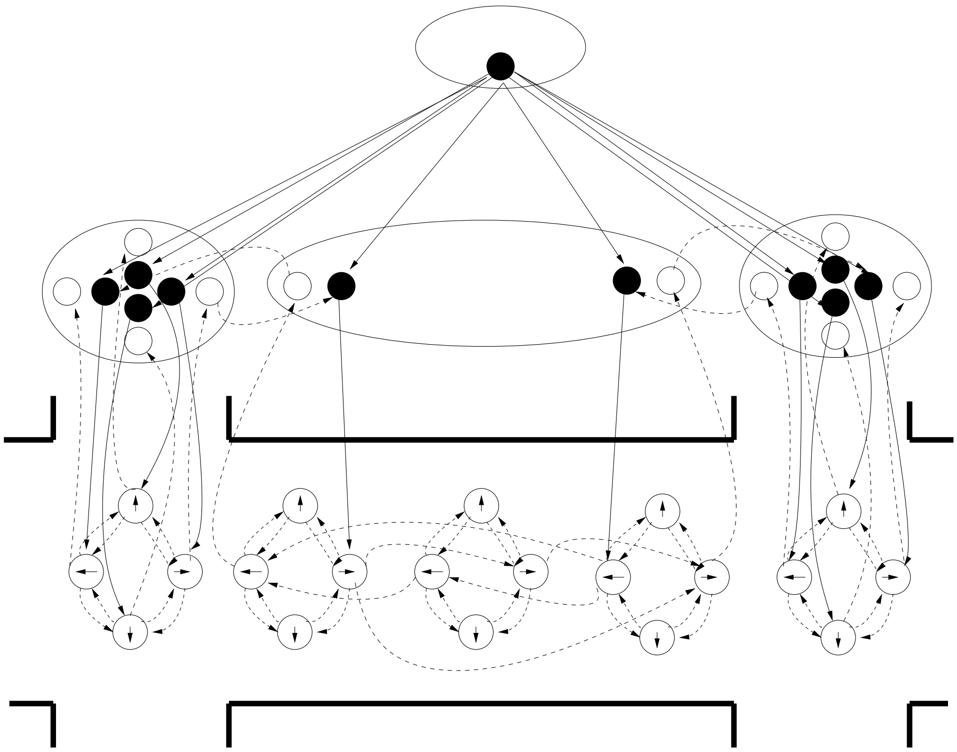

Figure 6.1: This HPOMDP model is used to represent corridor environments. The dotted arrows from product states show non-zero transition probabilities for primitive actions. The black vertical arrows show non-zero vertical activations of product states from the entry states. The dashed arrows from exit states show non-zero transition probabilities from exit-states to adjacent states of the abstract state they are associated with.

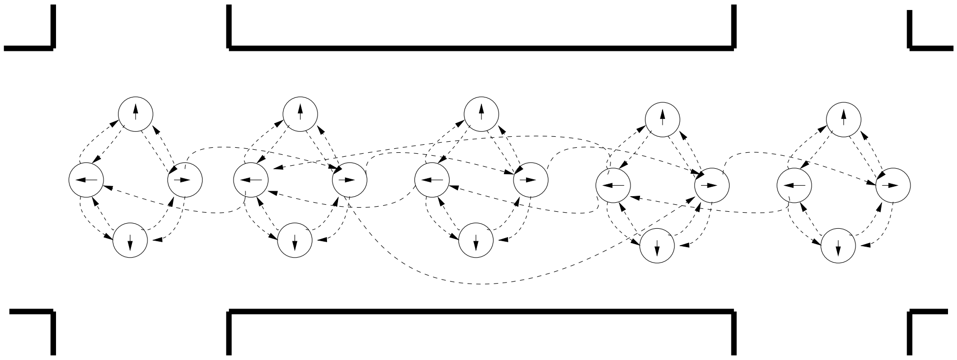

Figure 6.2: This is the equivalent flat POMDP model of the hierarchical POMDP model shown in Figure 6.1. In this model we only represent the product states of the hierarchical model. Global transition matrices for each action are automatically computed from the hierarchical model by considering all the possible paths between the product states under the primitive actions.

shown in Equation 6.1.

$$
\begin{aligned}
\text{normalized entropy} &= \frac{\text{entropy}}{\text{maximum entropy}} \\
&= \frac{-\sum_{s=1}^{S} P(s)\log P(s)}{-\sum_{s=1}^{S} 1/|S|\log(1/|S|)} \\
&= \frac{-\sum_{s=1}^{S} P(s)\log P(s)}{-\log(1/|S|)} \qquad = \frac{-\sum_{s=1}^{S} P(s)\log P(s)}{\log(|S|)}
\end{aligned} \tag{6.1}
$$

In fact, the entropy plot in Figure 6.4 reveals a number of characteristics that motivate our planning algorithm:

1. It is obvious from the plot that uncertainty is less at an abstract level.

2. Uncertainty reduces dramatically at junctions both at the global and abstract levels (see J_1, J_2, J_3 and J_4 in Figures 6.3 and 6.4).

3. At the global product level uncertainty increases as the robot traverses corridors where all observations are usually the same, but at an abstract level it does not increase.

4. Overall uncertainty decreases as the robot traverses a series of corridors.

Since uncertainty is less at an abstract level, an approximate algorithm that treats the states at the abstract level as completely observable should be more successful than an algorithm which treats all product states as completely observable. In addition, a successful algorithm would be one that is able to execute macro-actions (e.g., move from one entry-state of a corridor to the exit-state at the other exit point of the corridor) so that overall uncertainty is gradually reduced. In the next section we

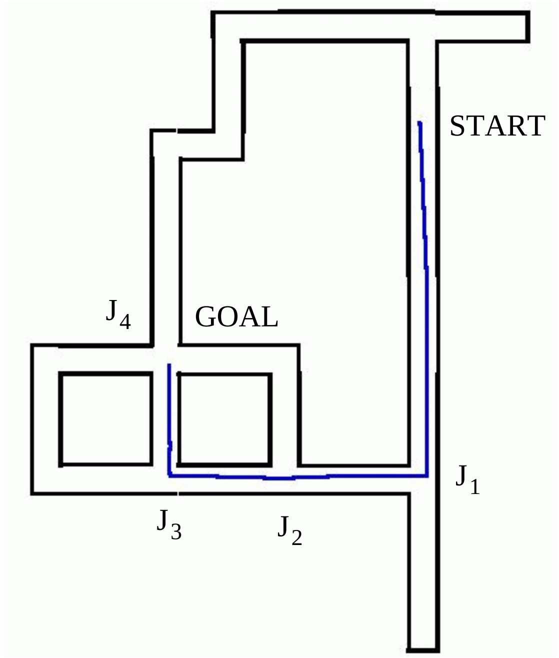

Figure 6.3: This is a robot run on the Nomad 200 simulator, which was used to collect observations for the entropy plot in Figures 6.4.

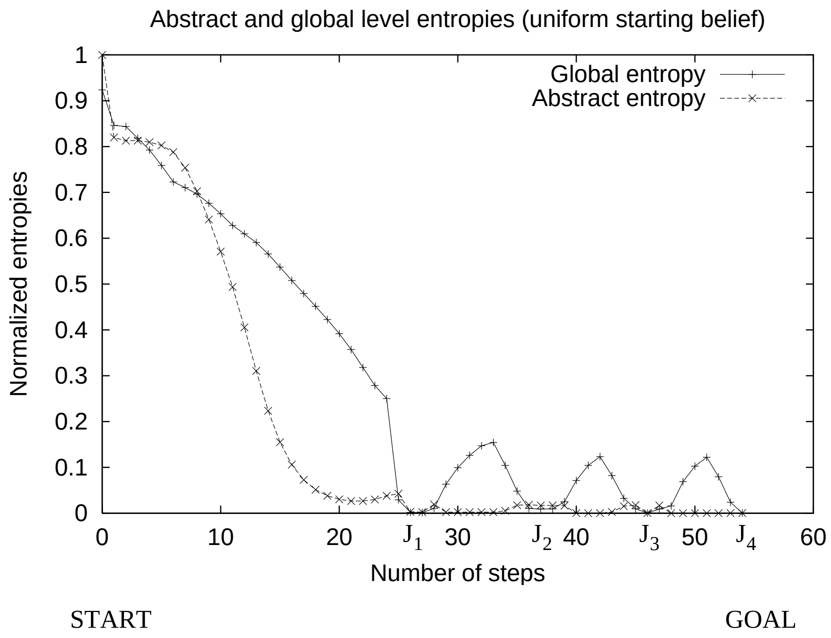

Figure 6.4: The plot shows the normalized entropies of the belief of the robot's location at the global product level and at the abstract level. The initial belief was uniform and observations were collected along the trace shown in Figure 6.3.

present related work from semi Markov decision processes (SMDPs), and heuristic POMDP solutions that formulate our hierarchical planning and execution algorithms for HPOMDPs.

6.2 Related work

6.2.1 Hierarchical planning in MDPs

Hierarchical planning in completely observable Markov decision processes has been explored by many researchers. Most notable are the hierarchical Abstract Machines framework (HAM) [103], [104], the MAXQ framework [105], and the options framework [25]. Here we briefly review the options framework on which we base much of the formulation of our hierarchical planning algorithm for HPOMDPs.

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 taken 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.

An abstract policy μ is a policy over options. However, before a policy among options is computed we need to define its model, which specifies for every state s where an option o may be started, and when it will terminate (see Equation 6.3). The total reward that will be received for executing option o in state s is:

$$r_s^o = E\{r_{t+1} + \gamma r_{t+2} \cdots + \gamma^{k-1} r_{t+k} | \mathcal{E}(o, s, t)\} \tag{6.2}$$

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 prediction model is defined as:

$$\mathcal{P}_{ss'}^o = \sum_{k=1}^{\infty} p(s', k)\gamma^k \tag{6.3}$$

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 for the value of executing an abstract policy μ at some state s as shown in Equation 6.4. The Bellman equation then, allows us to compute the values of executing each option from every state that belong in its initiation state $\mathcal{I}$, and thus be able to construct policies over options. Note that this equation is similar to Equation 2.4 in Section 2.1.

$$\begin{aligned} V^\mu(s) &= E\{r_{t+1} + \gamma r_{t+2} \cdots + \gamma^{k-1} r_{t+k} + \gamma^k V^\mu(s_{t+k}) | \mathcal{E}(\mu, s, t)\} \\ &= r_s^o + \sum_{s'} \mathcal{P}_{ss'}^o V^\mu(s') \end{aligned} \tag{6.4}$$

The transition and reward models of options can be learned using an intra-option learning method [25]. We can express the option models as Bellman equations (see Equations 6.6 and 6.5) and then solve the system of linear equations either explicitly, or through dynamic programming as was shown in Section 2.1.

$$r_s^o = r_s^{\pi(s)} + \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} (1 - \beta(s') r_{s'}^o) \tag{6.5}$$

$$\mathcal{P}_{sx}^o = \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[(1 - \beta(s')) \mathcal{P}_{s'x}^o + \beta(s') \delta_{s'x} \right] \tag{6.6}$$

where $\delta_{s'x} = 1$ if $s' = x$ and is 0 otherwise. Note that in the above Bellman equations (6.4, 6.5 and 6.6) the discount term γ, is included within the transition model. In order to be able to solve these equation using the standard value-iteration algorithm described in Chapter 2 (Figure 2.2) we can modify the definition of the state prediction model as shown in Equation 6.7. Equation 6.4 then becomes Equation 6.8. Equations 6.5 and 6.6 become Equations 6.9 and 6.10. This redefinition of the transition model implies that the transition models for primitive actions (or options made of a single action) are the same as the transition probability it self. Even though the end result (Equations 6.8 and 6.4) for both definitions is the same, this redefinition is easier to interpret, since it uses the same form of value function regardless of whether we compute a policy over options or a policy over primitive actions. Additionally, with the new definition its easier to understand how the options framework is adapted to our macro-action definition, which we present in Section 6.3.

$$\mathcal{P}_{ss'}^o = \sum_{k=1}^{\infty} p(s', k) \gamma^{k-1} \tag{6.7}$$

$$
\begin{aligned}
V^{\mu}(s) &= E\{r_{t+1} + \gamma r_{t+2} \cdots + \gamma^{k-1} r_{t+k} + \gamma^{k} V^{\mu}(s_{t+k}) | \mathcal{E}(\mu, s, t)\} \\
&= r_{s}^{o} + \gamma \sum_{s'} \mathcal{P}_{ss'}^{o} V^{\mu}(s')
\end{aligned}
\tag{6.8}
$$

$$
r_{s}^{o} = r_{s}^{\pi(s)} + \gamma \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} (1 - \beta(s') r_{s'}^{o})
\tag{6.9}
$$

$$
\mathcal{P}_{sx}^{o} = \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[\gamma(1 - \beta(s'))\mathcal{P}_{s'x}^{o} + \beta(s')\delta_{s'x}\right]
\tag{6.10}
$$

6.2.2 Approximate POMDP solutions

Since exact POMDP solutions are intractable, researchers have developed approximate solutions which have been proven to work successfully for robot navigation [23], [22], [76]. Here, we briefly review two approximations which assume that the POMDP is completely observable and solve it as was shown in Section 2.1. The MDP policy computed can then be transformed into a POMDP solution during execution through the following strategies:

- The **Most likely State** heuristic executes the action $\pi(s)$, where π is the completely observable policy that is associated with states s. The chosen state s has the highest belief $b(s)$.

$$
a = \pi(\mathrm{argmax}_{s \in S} b(s))
$$

- The $\mathbf{Q_{MDP}}$ approach does not use the completely observable policy directly, but rather uses the Q values of the state action pairs which were estimated during the value iteration algorithm (Figure 2.2). The QMDP approach chooses actions by assuming that the world will become completely observable after the first step. In other words, the action chosen is optimal if the POMDP becomes completely observable after the first action. To relate the QMDP approach to exact POMDP solutions we can think of the Q values of every action as an α vector (or policy tree value) as was shown in Equation 2.20.

$$\begin{aligned} a &= \text{argmax}_{a\in A} \sum_{s\in S} b(s) \left[r(s,a) + \gamma \sum_{s'\in S} P(s'|s,a) V^{\pi}(s')) \right] \\ &= \text{argmax}_{a\in A} \sum_{s\in S} b(s) Q(s,a) \end{aligned} \tag{6.11}$$

6.3 Planning and execution algorithms

Our planning and execution algorithms combine ideas from hierarchical SMDP planning with approximate solutions to POMDPs. First, we design a set of macro-actions that operate over the abstract states. We then compute the models of those macro-actions, which can be used to produce abstract plans or mappings from entry points of the abstract states to macro-actions. During execution we can either use a hierarchical most likely state (H-MLS) strategy, or a hierarchical QMDP (H-QMDP) heuristic.

In the H-MLS strategy we choose the most likely abstract state and then execute the appropriate macro-action under the abstract state. A macro-action terminates when the parent abstract state which initiated the macro-action is no longer the most likely abstract state. In the H-QMDP heuristic we choose the best macro-action to execute according to the QMDP heuristic in Equation 6.11, where primitive actions are replaced with macro-actions. Next, we describe the planning and execution algorithms in detail which are generalized to arbitrary tree depths.

6.3.1 Planning

To find a completely observable plan at an abstract level, we treat the product states and the entry states (of the abstract states) as the states of an MDP. The set of actions is the set of primitive actions applicable on product states and the set of macro-actions for each abstract state s, which are responsible for taking the agent out of the abstract state s, or to some child state of the abstract state. An exit from an abstract state can occur under any of the primitive actions, which result in one of the exit states. We design HPOMDPs such that only one type of primitive action can enter an exit state. In this manner the policy of each exit state can only be the type of primitive action that leads into it. Thus, the action associated with the exit-state can be used to connect the macro-action with states outside the subtree, in which the macro-action is defined. This idea is better understood in Figure 6.5. We should also node that in the case where an exit state S_x is entered from another exit state S'_x, the policy of exit state S_x is the same as the policy of exit state S'_x (see Figure 6.6).

In our navigation HPOMDP model all exits from abstract states occur under the "go-forward" action. Thus we need one exit-state for each spatial border of each abstract state. Macro-actions for exiting the abstract states can be computed using the MDP framework, where the robot is rewarded for taking the forward action in

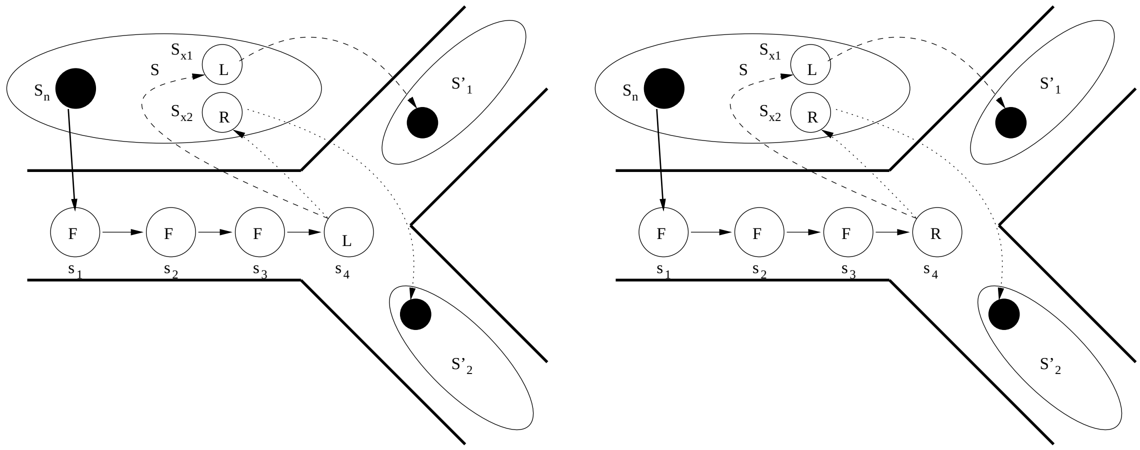

Figure 6.5: The figure shows two macro-actions for abstract state S. The letter F means go-forward, L turn-left, and R turn-right. The solid arrows define the transition matrix of the forward action, the dashed arrows define the transition matrix of the turn-left action, and the dotted arrows define the transition matrix of the turn-right action. The policy of exit-states is always the same since exit-states can only be entered with a single type of primitive action. The two macro-actions differ only on the policy of state s_4. Knowing the policy of exit-states allows us to know the outcome of the macro-actions at the level above.

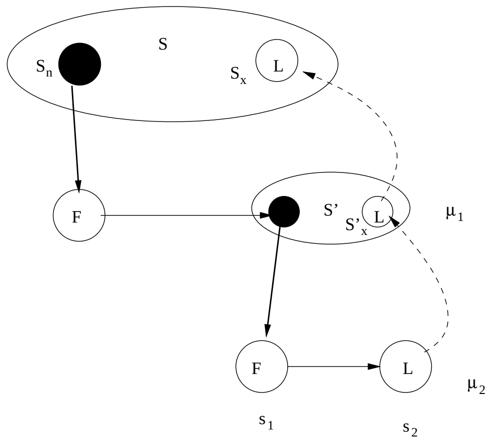

Figure 6.6: When an exit state S_x is entered through another exit state S'_x, the policy of S_x is the same as the policy of S'_x

the particular exit-state. Macro-actions for reaching a goal state in the middle of an abstract state can be computed by simply rewarding the robot for any action that it takes in the goal state. Figure 6.7 shows different macro-actions for corridors.

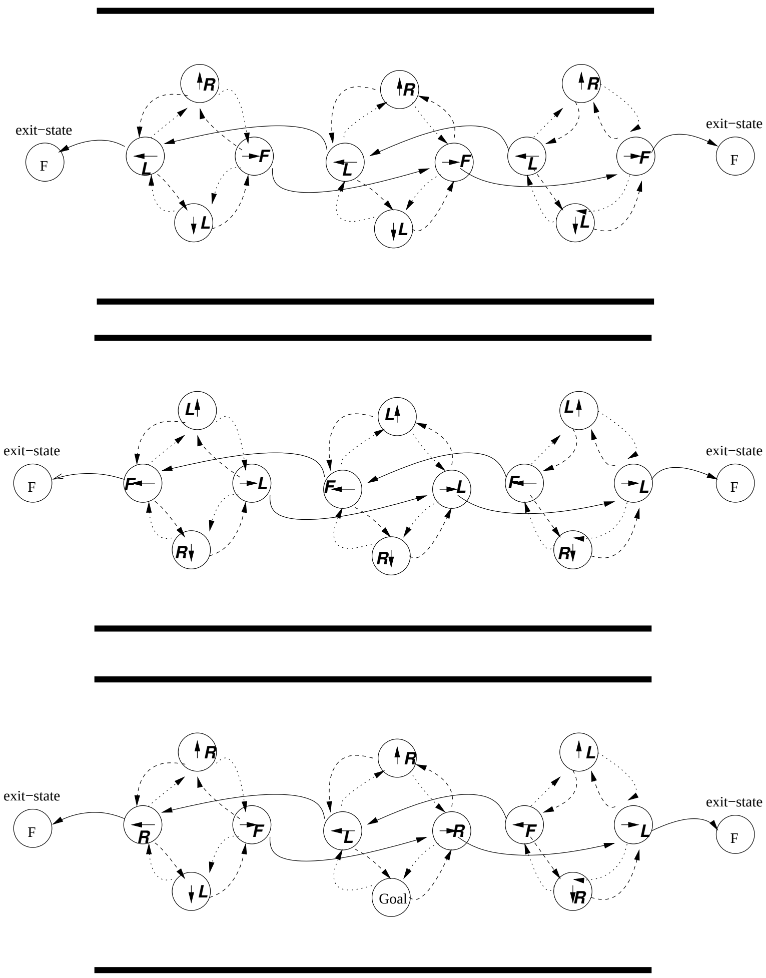

Figure 6.7: F means go forward, R means turn right and L means turn left. The top figure is a macro-action for exiting the East side of the corridor under the "go-forward" action, the middle figure is a macro-action for exiting the West side of the corridor with the go-forward action, and the bottom figure is a macro-action for reaching a particular location within the corridor. The solid arrows represent the transition matrix for the go-forward action, the dashed arrows represent the transition matrix for the turn-left action, and the dotted arrows represent the transition matrix for the turn-right action.

Overall our macro-action representation is a special case of the options framework and is similar to work presented in [106]. Each of our macro-actions can be initiated at some entry state of an abstract state and terminates at the exit states. More precisely, we can say that given a macro-action μ under an abstract state s, which is a policy over all the children, (the set of product states defined as C_p^s, the set of entry states of other abstract states defined as C_n^s, and the set of exit states of abstract state s defined as X^s), we can formulate one option for every entry state $s_n \in N^s$ (where N^s is the set of entry states of abstract state s). This option will have initiation set $\mathcal{I} = i | i \in C_p^s \cup C_n^s, V(i|s_n) \neq 0$, policy μ and termination condition β which will be 0 for all children of s except for all exit states $s_x \in X^s$.

Given the set of macro-actions available for each abstract state s (which we define to be M^s, also explained in Appendix A), we need to evaluate them for each entry state s_n. Equation 6.12 computes 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 product or entry state). Figure 6.8 describes Equation 6.12. 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 product 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 $i \in N^j$).

$$\begin{aligned} &T(s'|s_n, \mu) = \\ &\sum_{\forall s_x \in X^s} \left[\sum_{\forall i \in C_p^s \cup C_n^s} V(i|s_n) TD(s_x|i, \mu) \right] T(s'|s_x, \pi_\mu^s(s_x)) \end{aligned} \tag{6.12}$$

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 a set of linear equations defined in Equation 6.13. Note how this equation can be derived from the

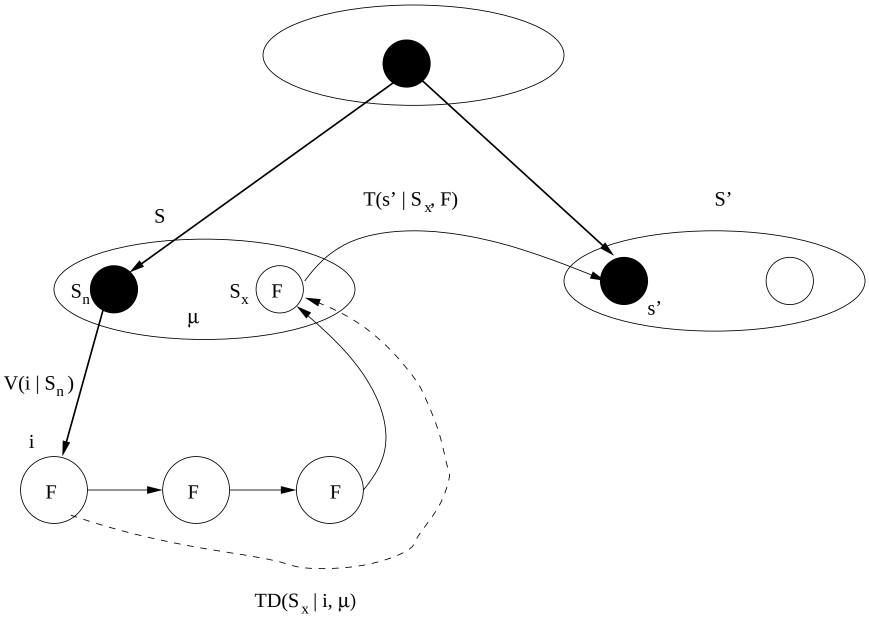

Figure 6.8: The figure shows the operation of of Equation 6.12. The dashed arrow is the discounted transition probability for reaching state S_x form state i. The solid arrows are simply the transition probabilities for the forward action.

option Bellman Equation 6.10 where the termination condition β is 0 for all product and entry children of state s and 1 for all exit states $s_x \in X^s$.

$$\begin{aligned} &TD(s_x|i,\mu) = \\ &T(s_x|i,\pi^s_\mu(i)) + \gamma \sum_{j \in C^s_p \cup C^s_n} T(j|i,\pi^s_\mu(i)) TD(s_x|j,\mu) \end{aligned} \tag{6.13}$$

In a similar manner we can calculate the total reward R that will be gained when an abstract action μ is executed at an entry state s_n as shown in Equation 6.14.

$$R(s_n,\mu) = \sum_{i \in C^s_p \cup C^s_n} V(i|s_n) RD(i,\mu) \tag{6.14}$$

where the term $RD(i,\mu)$ can be calculated by solving a set of linear equations defined in Equation 6.15. Again, this equation is a direct derivation of the option

Bellman Equation 6.9 where β is 0 for all product and entry children of state s and 1 for all exit states $s_x \in X^s$.

$$RD(i, \mu) = \\ R(i, \pi_\mu^s(i)) + \gamma \sum_{j \in C_p^s \cup C_n^s} T(j|i, \pi_\mu^s(i)) RD(j, \mu) \tag{6.15}$$

Given any arbitrary HPOMDP and all the macro-actions M^s of of every abstract state, we can calculate all the reward and transition models at the highest level by starting from the lowest levels and moving toward the top. Once the transition and reward models are calculated, we can construct a plan among product and entry-states at the most abstract level (level 2). To find a plan, we first compute the optimal values of states by solving a set of linear equations defined by the Bellman equation (as shown below).

$$V^*(s_1) = \\ \max_{a \in A | a \in M^{s_1}} \left[R(s_1, a) + \gamma \sum_{s_2}^{C^{root}} T(s_2|s_1, a) V^*(s_2) \right] \tag{6.16}$$

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 as shown in Equation 6.17.

$$\pi^*(s_1) = \\ \text{argmax}_{a \in A | a \in M^{s_1}} \left[R(s_1, a) + \gamma \sum_{s_2}^{C^{root}} T(s_2|s_1, a) V^*(s_2) \right] \tag{6.17}$$

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. If we know at which entry state the robot is, we can issue the appropriate macro-action which in turn may issue another macro-action at a lower level until a primitive action is issued. In reality, we never know if the robot is exactly at an entry-state at the abstract level and therefore the robot may have to decide the next macro-action when it is in the middle of some abstract state. Figure 6.9 illustrates this problem.

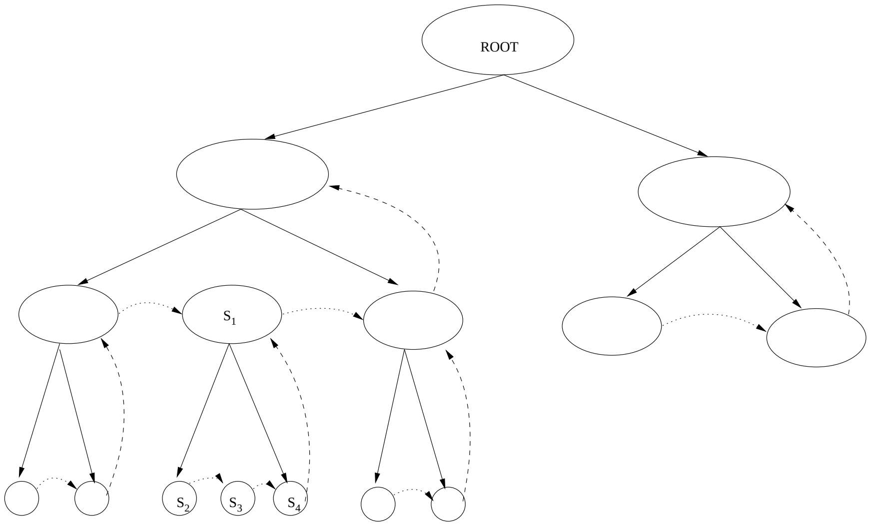

Figure 6.9: S_1, S_2, S_3, and S_4 cannot be activated vertically from states at the abstract level. Therefore, if the agent was to start in any of these states we don't know what is the best macro-action to initiate.

In other words, we need to know what is the best macro-action to issue next if the current macro-action is interrupted before it terminates at one of the exit states. The best macro-action for intermediate states (not direct descendants of the abstract level) needs to be computed by taking into consideration the plan at the above level, and the consequences that each macro-action available on that intermediate state will have on the long term reward. Formally, for any HPOMDP, we compute the values of all states which are not at the abstract level in an iterative fashion from top to bottom as explained in Figure 6.10 and described in Equation 6.18. The best macro-action for each state, which is not a child of the root, is then computed greedily according to equation 6.19.

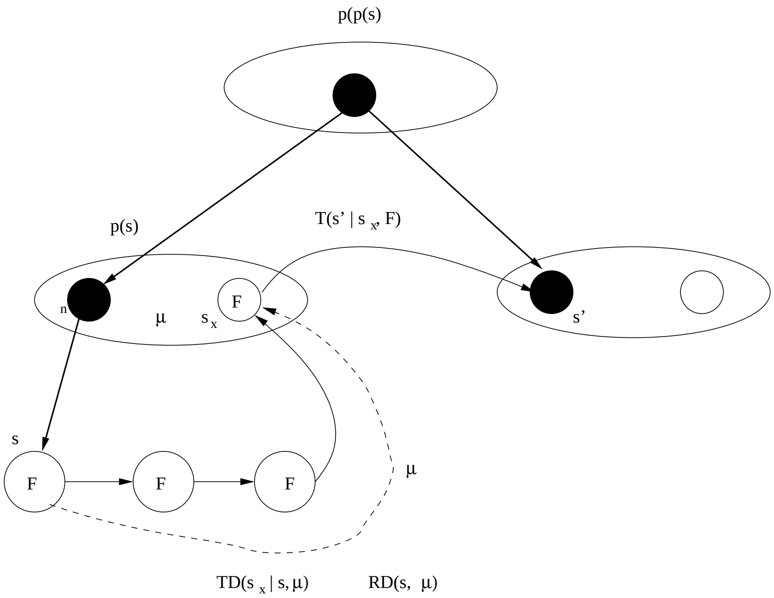

Figure 6.10: The value of a state s under a macro-action μ is computed by taking into consideration the total reward and discounted transition probability for exiting the parent of s, $p(s)$, from all exit states s_x, and the values of the resulting adjacent states to $p(s)$, such as s'. Equation 6.18 defines the value of a state as the maximum value among all macro-actions μ that belong to the set of of macro-actions $M^{p(s)}$ available for the parent of s, $p(s)$.

$$V(s) = \max_{\mu \in M^{p(s)}} \left[RD(s,\mu) + \gamma \sum_{s'}^{C_p^{p(p(s))} \cup C_n^{p(p(s))}} \left(\sum_{\forall s_x \in X^{p(s)}} TD(s_x|s,\mu) T(s'|s_x, \pi_\mu^{p(s)}(s_x) \right) V(s') \right] \tag{6.18}$$

$$\pi(s) = \mathrm{argmax}_{\mu \in M^{p(s)}} \left[RD(s,\mu) + \gamma \sum_{s'}^{C_p^{p(p(s))} \cup C_n^{p(p(s))}} \left(\sum_{\forall s_x \in X^{p(s)}} TD(s_x|s,\mu) T(s'|s_x, \pi_\mu^{p(s)}(s_x) \right) V(s') \right] \tag{6.19}$$

6.3.2 Execution

To execute the abstract plan we keep track of a probability distribution for each level of abstraction (at each tree depth). An efficient way to calculate the belief state for each level of the HPOMDP is to first calculate a global transition matrix among all product states under the primitive actions, and use that transition matrix to update a global belief state among product states after every action and observation as shown in Equation 6.20.

$$b'(s) = \frac{1}{P(z|a,b)} P(z|s,a) \sum_{s'=1}^{|S|} P(s|a,s')b(s') \tag{6.20}$$

where b' is the next belief state, and $P(z|s,a)$ is the probability of perceiving observation z when action a was taken in state s. In the robot navigation domain, there are 16 observations, which indicate the probability of a wall and opening on the four sides of the robot. The observations are extracted from trained neural nets where the inputs are local occupancy grids constructed from sonar sensors and outputs are probabilities of walls and openings [94]. $P(s|a,s')$ is the probability of going from state s' to state s under action a and b is the previous belief state. Using the flat belief state b we can calculate the belief of abstract states in an iterative fashion from leafs to root, where the belief of an abstract state is equal to the sum of the beliefs of its children as shown in Equation 6.21.

$$b(s) = \sum_{c(s,i)}^{C_p^s \cup C_a^s} b(c(s,i)) \text{ if s is an abstract state} \tag{6.21}$$

After every action and observation we update the belief state b, and then choose actions according to two strategies. The first strategy is a hierarchical most likely state strategy (H-MLS), and the second is a hierarchical QMDP strategy (H-QMDP) which we describe next.

H-MLS

For the H-MLS strategy we start from the root and iteratively find the most likely child state until a product state s is reached. To avoid, situations where the robot gets stuck into loop behavior, we don't always choose the state at the peak of the belief, but rather choose randomly among a set of most likely states. For each abstract state we maintain a separate set of most likely states, which are states that share the same parent abstract state (or otherwise states that belong in the same sublevel). A sublevel is a set of states, which all have the same parent. The states that belong in the most likely state set at the current sublevel are chosen such that their belief is is closer than $1/|S|$ to the maximum belief in the current sublevel (where $|S|$ is the total number of states which are at the same tree depth as the current sublevel). Once a product state is reached we choose the best macro-action to execute according to Equation 6.19. The macro-action terminates when one of the ancestors is no longer among the most likely state set of its sublevel. We then jump at the sublevel where the change occurred and restart the process.

To execute a macro-action under some abstract state S, we create a new separate local belief state b' which we initialize according to the most likely child product state of S. The macro-action is then executed using the flat MLS strategy [22]. A goal is reached when the most likely product state is the same as the goal state, and the probability of being at the ancestor, at the most abstract level is larger than 0.9. While Figure 6.11 describes in detail our H-MLS execution algorithm, Figure 6.12 shows an example H-MLS execution.

H-QMDP

For the H-QMDP strategy we find the best macro-action μ to execute under every abstract state S according to Equation 6.22.

$$\begin{aligned}\mu = \text{argmax}_{\mu \in M^S} \sum_{s \in C_p^S} b(s)[RD(s,\mu)+ \\ \gamma \sum_{s'}^{C_p^{p(S)} \cup C_n^{p(S)}} \left(\sum_{\forall s_x \in X^S} TD(s_x|s,\mu) T(s'|s_x, \pi_\mu^S(s_x)) \right) V(s')]\end{aligned} \tag{6.22}$$

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 is executed under the abstract state S until another $< \mu, S >$ pair is maximized (according to the above equation). Executing a macro-action is done in the same manner as for the H-MLS heuristic. Figure 6.13 explains graphically the H-QMDP heuristic.

1. Set current *sublevel* equal to 2 (children of the root), initialize the belief b, and create the most likely state sets for each sublevel. The states that belong in the most likely state set of each sublevel are chosen such that their belief is closer than $1/|S|$ to the maximum belief in the sublevel (where $|S|$ is the total number of states which are at the same tree depth as the sublevel).

2. Recursively choose randomly a state from the most-likely state set at the current sublevel until a product state i is reached.

3. Execute the appropriate macro-action μ from state i. This macro-action is one of the available macro actions that belong to the parent of i, that is, $\mu \in M^{p(i)}$

   (a) To execute the macro-action μ, create a new belief state b' which is defined only for the subtree rooted at the parent of i. Initialize b' such that $b'(i) = 1$ and the belief of all other product states that belong in the subtree is 0. Set $parent = p(i)$.

   (b) Choose the most likely state j among the children of *parent* according to b'.

   - If the most likely state is an abstract state (which may happen in unbalanced HPOMDPs), then we find the most likely entry state j_n. The belief of being at an entry state j_n is calculated by summing the beliefs of the children of j (according to b'), which have non-zero vertical transition from the entry-state j_n. The macro-action μ' to be executed under abstract state j then is defined by the macro-action μ that was initiated under *parent*, $\mu' = \pi(j_n)^{parent}_{\mu}$. We set $parent = j$ and $\mu = \mu'$ and repeat step 3(b).
   - If the most likely state j is a product state, then the primitive action a to be executed is defined by the best macro-action μ that was initiated under *parent*. $a = \pi(j)^{parent}_{\mu}$

   (c) Execute the primitive action, get the action and observation, update both b and b', and reestimate the most likely state sets for each sublevel

   - If the parent of i (or above) is not in the most likely state set of its sublevel, terminate the macro-action, jump to that *sublevel*, and go to step 2.
   - Otherwise go to step 3(b)

Figure 6.11: The figure describes in detail our hierarchical-MLS execution algorithm. The algorithm initiates macro actions by recursively finding the most likely state from root to leaves. Once a macro-action is initiated it runs until one of the ancestors of the state that the macro-action was initiated from, is no longer the most likely state.

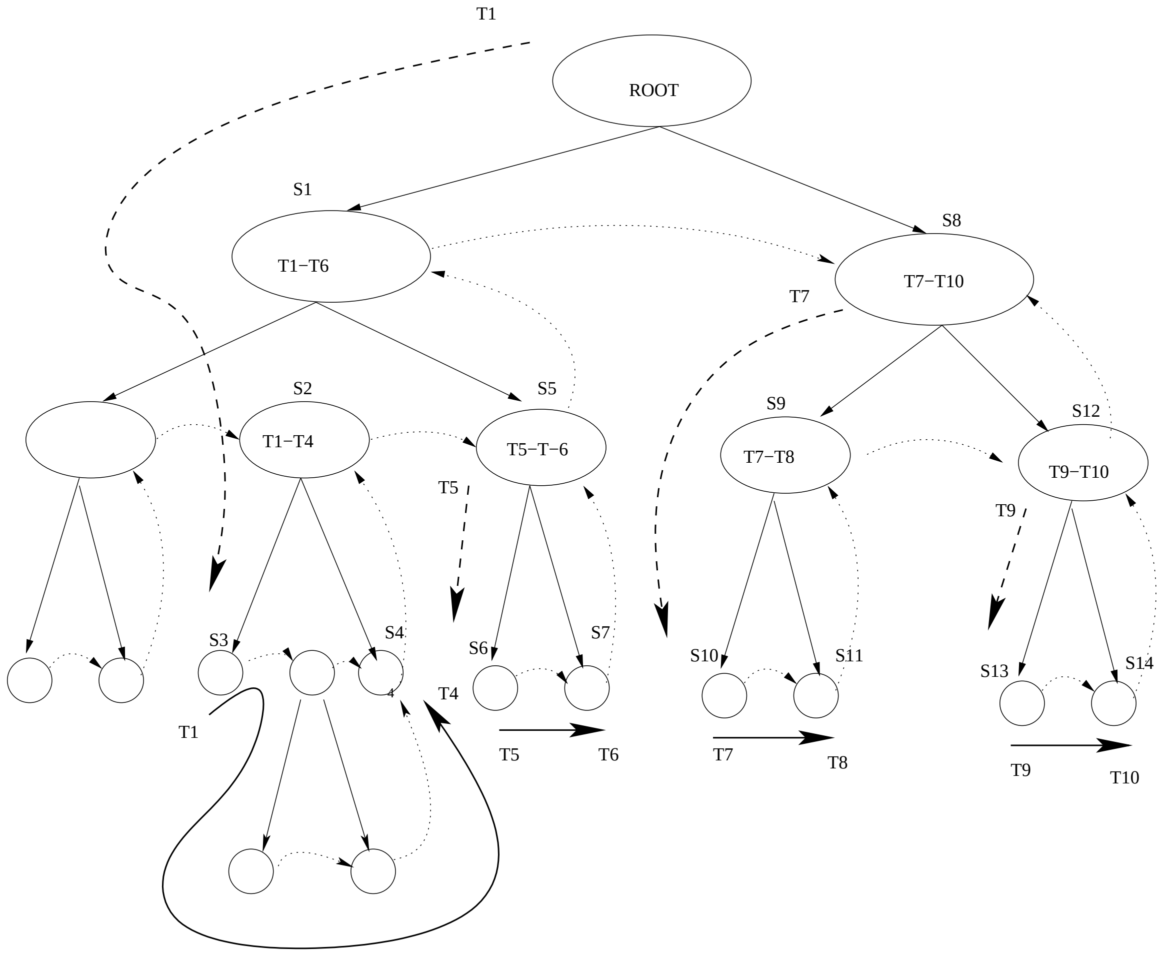

Figure 6.12: The figure shows an example HMLS execution. The bold solid arrows indicate the macro-actions while the bold dashed arrows indicate a new macro-action initiation. The beginning of the bold dashed arrows indicate the state at which, we recursively start choosing the most likely state until a product state is reached, at which point we can decide the appropriate next macro-action. Starting from the root we choose recursively the most likely state, which ends up to be S3. We then begin to execute the appropriate macro-action from S3 until one of its ancestors in no longer the most likely state (among its adjacent states). This happen at time T5 when S5 is the most likely state among the children of S1. Again, we recursively find the most likely product state starting from S5, which ends up to be S6. We begin to execute the appropriate macro-action until time T7 at which point the most likely state among the children of the root has changed to S8. We then again, find recursively the most likely state starting from state S8 which ends up to be state S10. The appropriate macro-action begins to execute until time T9 at which the most likely state among the children of S8 changes to state S12. Again, we find recursively the most likely state starting from S12 which is S13 and execute the appropriate macro-action.

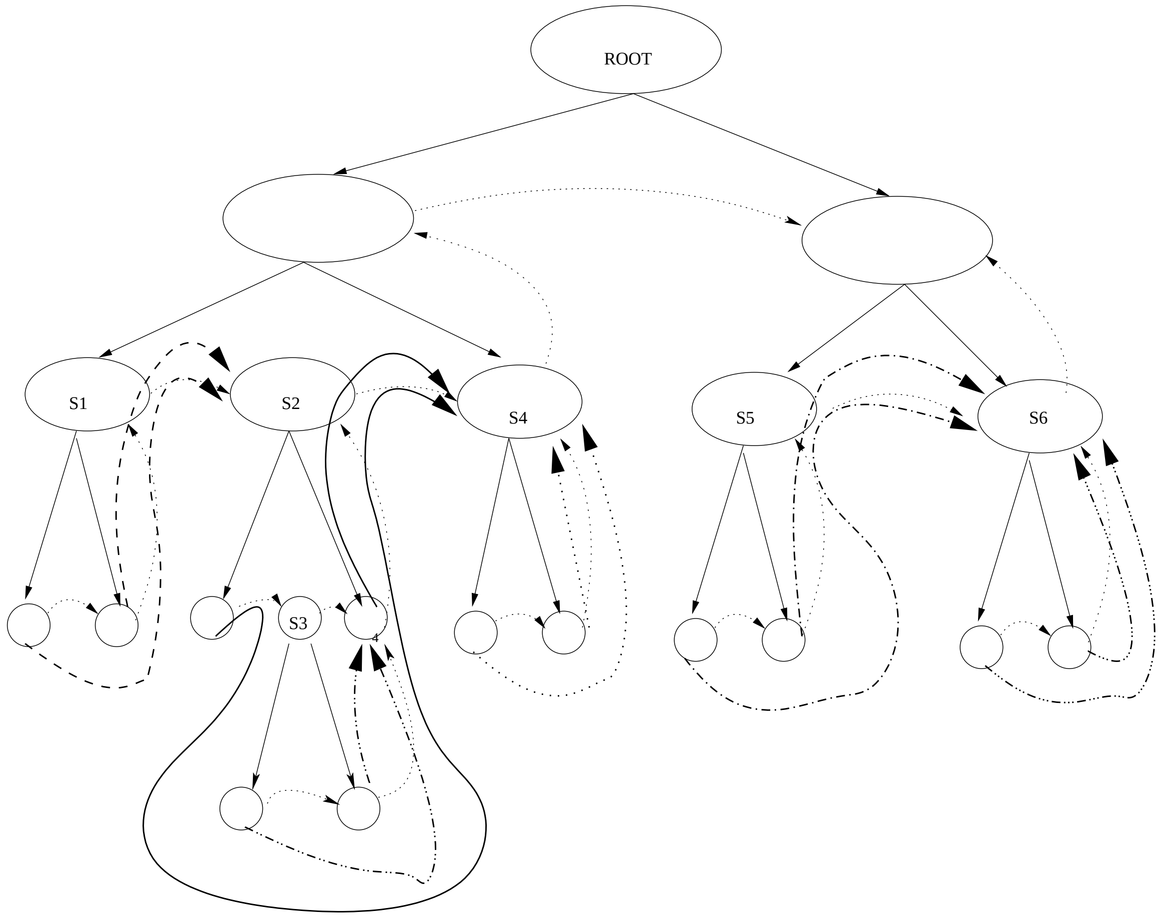

Figure 6.13: Each type of bold arrow represents a different macro-action that can be initiated from a product state. The H-QMDP heuristic multiplies the value of executing every macro-action from a product state with the current belief of the product state. The best macro-action to be executed is the one for which the sum of all the products of values and beliefs is the maximum. Note that choosing a macro-action, also defines the abstract state to be executed under, since every macro-action is unique to a a particular abstract state. In this example we have assumed that we only have one macro-action for each abstract state, but in our navigation models we have multiple macro-actions for each abstract state.

6.4 Planning experiments using the robot platforms

6.4.1 Experiments with hand-coded models

We applied the above algorithms to two robot navigation environments, which were shown in Figures 4.3, and 4.4 in Chapter 4. Both of the environments were represented as HPOMDPs using hand-coded parameters. The reason we used hand-coded parameters, was to separate the evaluation of the planning algorithms from the learning algorithms. Otherwise, it would be difficult to evaluate the planning algorithms since the learned parameters affect robot performance, as we shall see in Section 6.4.2.

In the small navigation environment, 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 6.1. 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.

It is evident from Table 6.1 that the hierarchical approaches perform better than the flat. The MLS strategies produce 100% success due to the randomness in the

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 6.1: 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.

selection of the peak of the belief state. The QMD 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 6.2. We also ran the robot in the real environment using the HPOMDP model. In this experiment we started the robot from 11 different locations and used 5 different goal destinations. The results are summarized in Table 6.2.

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 6.2: 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.

It is obvious from Table 6.2 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 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 run the most successful hierarchical approach, the H-MLS, which proved to work successfully. Comparing Table 6.2 with Table 6.1, 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 6.3 where for the HPOMDP 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 HPOMDP 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 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 $O(S^2T)$.

It is apparent from Table 6.3 that hierarchical plans are constructed faster than flat 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 a new macro-

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

Table 6.3: The table shows the planning times for the different environments and models. The planning times were computed on a 900 MHz Athlon processor

action needs to be computed, the one for the abstract state which contains the goal location.

6.4.2 Experiments with learned models

We also performed experiments with learned HPOMDP models to investigate the effect of learning HPOMDP 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 HPOMDP model. In this HPOMDP 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 junctions 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 product states, which is computed as was shown earlier in Equation 6.1. 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 6.4.

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 6.4: 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 HPOMDP 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.

6.5 Conclusions

We have seen in this chapter that our hierarchical approach to robot navigation using the HPOMDP model outperforms previous flat POMDP based representations. The algorithms presented take less number of steps than the flat POMDP approaches in taking the robot to the goal starting from no positional knowledge. Also, the hierarchical approaches scale gracefully to large environments than the flat approaches ass can be seen from Tables 6.1, and 6.2. Additionally, abstract plans are constructed much faster than flat plan representations as can be seen from Table 6.3.

The better performance in taking the robot to its goal is due to two key reasons. First, the robot takes decisions based on a hierarchical representation of its belief about its true location. A hierarchical belief representation is advantageous, in that at higher levels of abstraction uncertainty is less than lower levels due to the significantly smaller number of states. Second, macro-action termination depends on the abstract belief state which changes more slowly than the global product belief state. As a result, macro-actions execute for multiple steps which results in entropy reduction.

The disadvantage of our algorithms, is that they are is still approximate solutions based on a completely observable hierarchical MDP solution. Even though they perform better than flat POMDP approximations they are still far from the optimal number of steps for reaching the goal, which can be achieved when the initial position is known. A future direction would be the study of this model with respect to exact POMDP solutions. A hierarchical approach for planning and execution in POMDPs that is closer to exact POMDP solutions is described in [91]. In the next chapter we present in detail the main conclusions and contributions of the this dissertation and directions for future research.

Chapter 7

Conclusions and future work

In this final chapter we summarize the main contributions of the thesis and describe directions for future research. Section 7.1 summarizes the dissertation, and Section 7.2 presents the main contributions. Section 7.3 presents a number of concrete directions for future research.

7.1 Summary

In this dissertation we have described a framework for learning and planning using hierarchical models of partially observable environments, and applied it to indoor robot navigation. We introduced a new multi-resolution model called HPOMDP, and presented hierarchical learning and planning algorithms for this model.

In Chapter 3 we defined the HPOMDP model and described an EM type learning algorithm for learning its parameters. In Chapter 4 we described the robot platforms, the extraction of features from the sensor values, the representation of spatial environments as HPOMDPs, and the overall navigation architecture.

In Chapter 5 we introduced two approximate learning strategies, "reuse-training", and "selective-training", and presented detailed experimental learning studies for partially observable indoor corridor environments. The experiments were done on a small simulated environment as well as with robot platforms in the Nomad 200 simulator and in the real world. We compared the hierarchically learned models with models learned using the standard EM for flat POMDPs. The comparisons were done using different metrics such as distance to the generative model, error in robot localization, log likelihood of the training data, and training time. Additionally we used the hierarchical learning algorithms to prune away false transition models at higher level of abstraction both for a three level, and a four level hierarchical model.

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 training strategies which take advantage of short length data sequences have their own advantages. While the "reuse-training" method converges fast (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 the 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 in our implementation the algorithm's time complexity is $O(NT^3)$. Nonetheless, a recently developed training algorithm for HMMs uses linear time [83], and could be adapted to overcome this limitation in our implementation. In addition, training data sequences needs 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 environment training data needs 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.

In Chapter 6 we introduced our hierarchical planning and execution algorithms, and used them in robot navigation experiments. Experiments were performed with good hand-coded models, for the real environment, and for simulation environments in the Nomad 200 simulator. To evaluate the planning algorithms we measured the average number of steps it took for the robot to reach locations in its environment starting from uniform initial belief states. We also performed experiments with learned models, in the Nomad 200 simulator, and showed how the whole learning and planning system is combined in practice. We showed how navigation performance improves after every learning stage. The learning stages were compared based on average steps to reach the goal and the average uncertainty per run as the model parameters improved.

Our planning algorithms use both spatial and temporal abstraction and combine approaches from hierarchical MDP solutions with approximate POMDP strategies. Using these algorithms the robot is able to execute navigation tasks starting from no initial positional knowledge, in less number of 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 comparable to the current state of the art in robot navigation systems, such as the monte carlo localization approach described in [73].

The disadvantage of our planning algorithm is that it assumes abstract and product states are completely observable. The assumption that we have different sets of macro-actions for each abstract state may not be valid in an exact hierarchical POMDP solution. In general, in an exact HPOMDP solution we would have to create POMDP type macro-actions (or POMDP policies as was shown in Section 2.2, which are policy graphs mapping observations to actions) and evaluate their transition and reward models for each abstract state. Nonetheless, computing exact POMDP type macro-actions even for small state spaces is intractable.

7.2 Contributions

We summarize the main contributions of this dissertation below:

- We have formally introduced and described the hierarchical partially observable Markov decision process model. 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. Spatial abstraction is achieved by low resolution abstract states, which group together finer resolution states in a tree like fashion. In temporal abstraction, a transition from an abstract state cannot occur unless the process under the abstract state has finished. This model was presented in Chapter 3.

- We have derived an EM algorithm for learning the parameters of a given HPOMDP. We empirically showed that the algorithm converges by increasing the log likelihood of the data after every training epoch. The algorithm was presented in Chapter 3, and the empirical convergence results described in Chapter 5.

- We have introduced two approximate training methods. For the first method, which we named "selective-training", only selected parts of an overall HPOMDP are allowed to be trained for a given training sequence. This results in faster training. For the second method, which we named "reuse-training", submodels of an HPOMDP model are first trained separately and then reused in the overall model. This results in better learned models. Both algorithms and results were presented in Chapter 5.

- We have derived two planning and execution algorithms for approximating the optimal policy in a given HPOMDP. The planning and execution algorithms combine hierarchical solutions for solving Markov decision processes, such as the options framework [25], and approximate flat POMDP solutions [22] (Chapter 6).

- We conducted a detailed experimental study of the learning algorithms for indoor robot navigation. Experiments were 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 HPOMDP models which results in better robot localization (Chapters 5 and 6). Additionally, we compared our learned models with equivalent flat POMDP models that were trained with the same data. We have developed approximate training methods to learn HPOMDP models that converge much more rapidly than standard EM procedures for flat POMDP models. When the learned HPOMDP models where converted to flat, they gave better robot localization than the flat POMDP models that were trained with the standard flat EM algorithm for POMDPs (Chapter 5).

- We applied the planning algorithms to indoor robot navigation, both in simulation and in the real world. Our algorithms are highly successful in taking 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 much faster, and use a considerable smaller number of steps in executing navigation tasks (Chapter 6).

7.3 Future Directions

We now describe some directions for future research including faster training algorithms, structure learning, extension of the model to sharing substructures, and exact HPOMDP solutions.

7.3.1 Faster training algorithms

It would be useful to think of other approaches to implementing the same learning algorithm where the time complexity is less than $O(NT^3)$. Such an approach that uses the Dynamic Bayesian network formalism has recently been developed for HHMMs in [83]. The transformation of HHMMs to DBNs allows us to train them by combining standard inference techniques in Bayesian nets such as the junction-tree algorithm [84]. The junction-tree algorithm replaces the expectation step in the hierarchical Baum-Welch algorithm. The major advantage is that with this combination HHMMs can be trained in linear time, while the conventional hierarchical Baum-Welch algorithm requires cubic time [17]. We could adapt such an approach to learning HPOMDPs, and at the same time verify it's correctness by comparing it to the original HHMM training algorithm.

We could also explore approximate inference algorithms for computing the Expectation step such as Markov Chain Monte Carlo (MCMC) methods [107]. These methods can be thought of as simulations of Markov processes where they estimate the number of times a state has been visited (similar to the ξ variable in Baum-Welch). Such approaches are used in situations where exact computation of the Expectation step is intractable. They have been applied in training factored HMMs [108], as well as HMMs with continuous action and observation spaces [109]. In our case computation of the expectation step through sampling may reduce our training time, since it will eliminate unnecessary summations (specially when the initial starting state is provided).

Finally, the investigation of an Incremental EM procedure for training HPOMDPs may eliminate the burden of long training times. Incremental versions of the EM for HMMs have been studied in [110], and [111]. In Incremental type EM algorithms the E-step computes sufficient statistics and can be updated after every new observation. The M-step is the same as the batch EM algorithm.

7.3.2 Structure learning

Another future direction would be to learn the initial model structure (number of states, and allowed vertical and horizontal transitions) independent of any specific application domain. In the past researchers have developed such approaches for flat HMMs. One approach is through a top-down state splitting where cross validation is used to determine if a local change to the model is beneficial [112]. Another bottom-up approach, which starts with a specific model consistent with the training data and generalizes the model by merging states together, is presented in [113]. We would like to extend such approaches to learning the structure of HPOMDPs.

Nonetheless, since HHMMs can be converted to dynamic Bayesian nets [83], we can use structure learning methods from the Bayesian Network literature. Structure learning in Bayesian net has been extensively studied by many researchers [114], [115], [116], and [117].

7.3.3 Sharing substructure

Another future direction would be to extend the HPOMDP model such that abstract states share substructure. In this extension, we would be able to scale-up to even larger domains since a single submodel would be able to represent multiple abstract states. In such a representation not only would we have smaller models, but we could also modify the Baum-Welch calculation of the forward and backward variables such that it is only done once for abstract states that share the same substructure and as a result speed up the computation time as well. We could, for example, have at an abstract level a representation of the spatial environment and at lower levels have the representation of the notion of corridors. This idea is illustrated in Figure 7.1.

Such an extension however would impose certain requirements, such as the policies required for each entry point of every abstract state that share the same sub-structure are the same. Otherwise the robot would not know which action to execute, since its belief at being at any of the abstract states that share the same substructure would be the same. Unfortunately, knowledge of the required action means that a policy has already been computed which means that the model is task specific. One possibility is to have a representation where some abstract states always represent fixed policies (e.g., always traverse the corridor). Unlike the approach in [85] where the effects of abstract actions at abstract levels are an integral part of the AHMM specification, we could take a bottom up approach where low level abstractions represent task specific representations and high level spatial abstraction is task independent.

7.3.4 Exact HPOMDP solutions

Finally, a key question that this dissertation may raise is how does spatial and temporal abstraction aid in the exact solutions of POMDPs. If a macro-action is a policy-tree as described in Chapter 2, then when do we initiate and when do we terminate each macro-action. What is an abstract state in a POMDP? Is it a set of physical states or a set of belief states? Do we need to compute macro-actions over a set of belief states or over a set of physical states. Are macro actions applicable everywhere or locally? In POMDPs primitive actions need to be applicable everywhere because the true state is never known; does the same condition hold for macro-actions? If we were to compute macro-actions over the entire POMDP then the computational complexity will be the same as solving the whole POMDP and therefore temporal abstraction may not be as useful. If we assume however, that we are a given a set of macro-actions over the entire POMDP, then how does the computation of any global plan based on the macro-actions compares with a flat approach?.

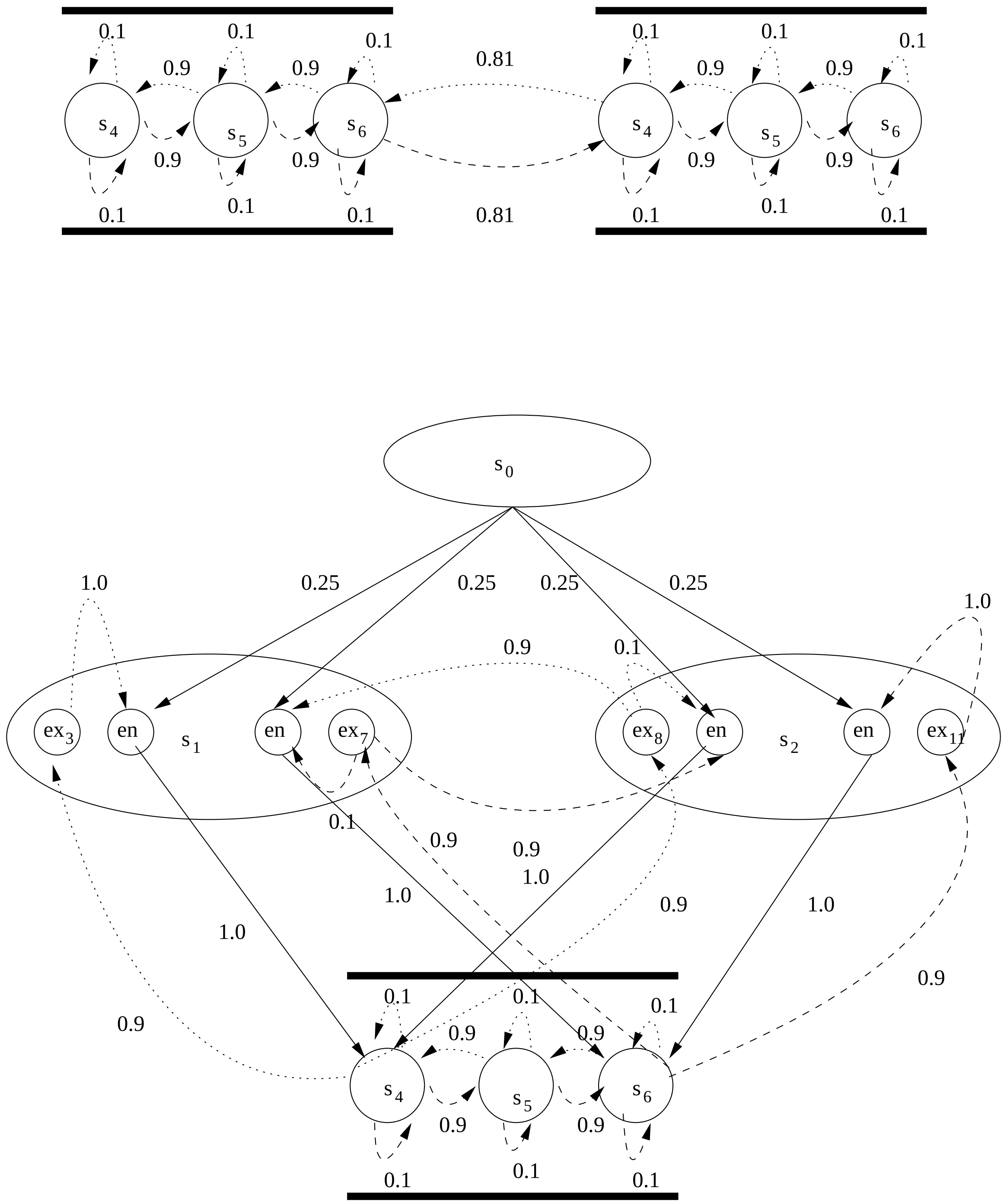

Figure 7.1: The top part show a corridor environment and its flat POMDP representation. The bottom part shows an extended HPOMDP representation where abstract states share submodels. This future direction will have the potential of scaling the algorithm to even larger domains, and applications to problems that have repetitive structure such as the task of reading.

Appendices

Appendix A

A.1 Table of HPOMDP symbols

Symbol	Definition
C_p^s	the set of product 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 A.1: Table of HPOMDP symbols

186

Bibliography

Bibliography

[1] S. Thrun, “Learning metric-topological maps for indoor mobile robot navigation,” *Artificial Intelligence*, no. 1, pp. 21–71, 99.

[2] S. Koenig and R. Simmons, “Passive distance learning for robot navigation,” in *Proceedings of the Thirteenth International Conference on Machine Learning (ICML)*, pp. 266–274, 1996.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning An Introduction*. The MIT Press, 1998.

[4] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of Artificial Intelligence Research*, 1996.

[5] R. A. Howard, *Dynamic Programming and Markov Processes*. The MIT Press, Cambridge, Massachusetss, 1960.

[6] K. J. Anstrom, “Optimal control of Markov decision processes wih incomlete state estimation.,” *Journal of Mathematical Analysis and Applications*, no. 10, 1965.

[7] C. Papadimitriou and J. Tsitsiklis, “The complexity of Markov decision processes,” *Mathematics of Operation Research*, vol. 12(3), 1987.

[8] O. Madani, S. Hanks, and A. Gordon, “On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision processes,” in *Proceedings of the Sixteenth National Conference in Artificial Intelligence*, pp. 409–416, 1999.

[9] L. E. Baum and G. R. Sell, “Growth functions for transformations on manifolds,” *Pac. J. Math.*, vol. 27, no. 2, pp. 211–227, 1968.

[10] L. Rabiner, “Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition,” in *Proceedings of the IEEE*, vol. 77, February 1989.

[11] S. Koenig and R. Simmons, “Unsupervised learning of probabilistic models for robot navigation,” in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2301–2308, 1996.

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

[13] F. C. Dyer, "Spatial cognition: lessons from central-place foraging insects," in *A Synthetic Approach to Studying Animal Cognition* (I. Pepperberg, A. Kamil, and R. Balda, eds.), pp. 119–154, London: Academic Press, 1998.

[14] H. L. Pick, M. R. Heinrichs, D. R. Montello, K. Smith, C. N. Sullivan, and W. B. Thompson, "Topographic map reading," *The Ecology of Human-Machine Systems*, vol. 2, 1995.

[15] J. Piaget and A. Szeminska, *The Child's Conception of Geometry.* New York: Basic Books, 1960.

[16] A. W. Siegel and S. H. White, "The development of spatial representations of large-scale environments," in *Advances in Child Development and Behaviour* (H. W. Reese, ed.), Academic Press, 1975.

[17] S. Fine, Y. Singer, and N. Tishby, "The Hierarchical Hidden Markov Model: Analysis and Applications," *Machine Learning*, vol. 32, pp. 41–62, July 1998.

[18] M. Mataric, "A distributed model for mobile robot environment-learning and navigation," Tech. Rep. TR-1228, MIT, 1990.

[19] G. Chartrand and O. R. Oellerman, *Applied and Algorithmic Graph Theory.* Mc Graw Hill, 1993.

[20] R. Simmons, "Becoming increasingly reliable," in *Proceedings of the International Conference on Artificial Intelligence Planning Systems (AIPS)*, pp. 152–157, 1994.

[21] M. Collett and T. S. Collett, "How do insects use path integration for their navigation," *Biol. Cybern*, no. 83, pp. 245–259, 2000.

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

[23] I. Nourbakhsh, R. Powers, and S. Birchfield, "Dervish: An office-navigation robot," *AI Magazine, 53-60*, vol. 16, no. 2, pp. 53–60, 1995.

[24] N. Khaleeli, "A robust robot navigation architecture using partially observable semi-Markov decision processes," Master's thesis, University of South Florida, 1997.

[25] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, pp. 112:181–211, 1999.

[26] B. Kuipers and Y.-T. Byun, "A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations," *Journal of Robotics and Autonomous Systems*, vol. 8, 1991.

[27] H. Choset, *Sensor based motion planning: The hierarchical generalized voronoi graph.* PhD thesis, California Institute of Technology, 1996.

[28] M. Puterman, *Markov Decision Processes: Discrete Dynamic Stochastic Programming.* John Wiley, 1994.

[29] R. Bellman, *Dynamic Programming.* Princeton University Press, 1957.

[30] D. P. Bersekas, *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall, 1987.

[31] C. J. C. H. Watkins, *Learning from Delayed Rewards.* PhD thesis, Cambridge University, 1989.

[32] J. N. Tsitsiklis and B. V. Roy, "Asynchronous stochastic approximation and Q-learning," *Machine Learning*, vol. 16, September 1994.

[33] K. J. Astrom, "Optimal control of Markov decision processes with incomplete state estimation," *Journal of Mathematical Analysis and Applications*, vol. 10, pp. 174–205, 1965.

[34] E. Sondik, *The Optimal Control of Partially Observable Markov Processes.* PhD thesis, Stanford University, 1971.

[35] R. D. Smallwood and E. J. Sondik, "The optimal control of partially observable Markov processes over a finite horizon," *Operations Research*, no. 21, pp. 1071–1088, 1973.

[36] C. White and D. Harrington, "Applications of Jensen's inequality for adaptive suboptimal design," *Journal of Optimaization Theory and Applications*, pp. 32(1):89–99, 1980.

[37] E. J. Sondik, "The optimal control of partially observable Markov processes over the infinite horizon," *Operations Research*, pp. 26(2):282–304, 1978.

[38] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, pp. 99–134, 1998.

[39] G. E. Monahan, "A survey of partially observable Markov decision processes: Theory, models, and algorithms," *Management Science*, no. 28(1), pp. 1–16, 1982.

[40] H. T. Cheng, *Algorithms for Partially Observable Markov Decision Processes.* PhD thesis, University of British Columbia, 1988.

[41] A. Cassandra, M. L. Littman, and N. L. Zhang, "Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes," in *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 1997.

[42] M. Hauskrecht, "Value-function approximations for partially observable Markov decision processes," *Journal of Artificial Intelligence Research*, vol. 13, pp. 33–94, 2000.

[43] R. Parr and S. Russell, "Approximating optimal policies for partially observable stochastic domains," in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1088–1094, 1995.

[44] R. I. Brafman, "A heuristic variable grid solution method for POMDPs," in *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI)*, 1997.

[45] E. Hansen, "Solving POMDPs by searching in policy space," in *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 1998.

[46] N. Meuleau, L. Peshkin, K.-E. Kim, and L. Kaelbling, "Learning finite-state controllers for partially observable environments," in *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 1999.

[47] N. Meuleau, K.-E. Kim, L. Kaelbling, and A. Cassandra, "Solving POMDPs by searching the space of finite policies," in *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 1999.

[48] L. Peshkin, N. Meuleau, and L. Kaelbling, "Learning policies with external memory," in *Interanational Conference on Machine learning (ICML)*, 1999.

[49] A. Kosaka and A. Kak, "Fast vision-guided mobile robot navigation using model-based reasoning and prediction of uncertainties," in *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, pp. 2177–2186, 1992.

[50] R. Smith and P. Cheeseman, "On the representation and estimation of spatial uncertainty," *The International Journal of Robotics Research*, pp. 5:56–68, 1986.

[51] T. Dean, K. Basye, R. Chekaluk, S. Hyun, M. Lejter, and M. Randazza, "Coping with uncertainty in a control system for navigation and exploration.," in *Proceedings of the the Eighth National Conference on Artificial Intelligence (AAAI)*, pp. 1010–1015, 1990.

[52] R. H. Battin and G. M. Levine, "Application of Kalman filtering techniques to the Apollo program," tech. rep., MIT Instrumentation Laboratory, April 1969. E-2401.

[53] I. J. Cox and J. Leonrad, "Modeling a dynamic environment using a Bayesian multiple hypothesis approach," *Artificial Intelligence*, pp. 311–344, 1994.

[54] S. Roumeliotis and G. Bekey, "Bayesian estimation and Kalman filtering: A unified framework for mobile robot localization," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (San Francisco CA), pp. 2985–2992, 2000.

[55] K. Basye, T. Dean, J. Kirman, and M. Lejter, "A decision-theoretic approach to planning, perception, and control," *IEEE Expert*, vol. 7, no. 4, pp. 58–65, 1992.

[56] J. K. Baker, "The dragon system-an overview," *IEEE Trans. Acoust. Speech Signal Processing*, vol. ASSP-23, pp. 24–29, Feb 1975.

[57] J. A. Bilmes, "A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian Mixture and Hidden Markov Models," tech. rep., International Computer Science Institute, Berkley CA, 94704, April 1998. TR-97-021.

[58] C. M. Bishop, *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[59] M. Jordan and R. Jacobs, "Hierachical mixtures of experts and the EM algoritm," *Neural Computaion*, pp. 6:181–214, 1994.

[60] R. Redner and H. Walker, "Mixture densities, maximum likelihood and the EM algorithm," *SIAM Review*, 26(2), 1984.

[61] G. J. McLachlan and T. Krishnan, *The EM algorithm and extentions.* Wiley, New York, 1997.

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

[63] J. Leonard and H. F. Durrant-Whyte, "Mobile robot localizatin by tracking geometric beacons," in *IEEE Transactions Robotics and Automation*, vol. 7(3), pp. 376–382, 1991.

[64] I. J. Cox, "Blanche- an experiment in guidance and navigation of an autonomous mobile rrobot," in *IEEE Transactions Robotics and Automation*, vol. 7(3), 1991.

[65] I. A. Getting, "The global positioning system," *IEEE Spectrum*, pp. 36–47, December 1993.

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

[67] S. Chen and J. Weng, "State-based SHOSLIF for indoor visual navigation," *IEEE Trans. Neural Networks*, vol. 11, no. 6, pp. 1300–1314, 2000.

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

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

[70] S. Thrun, W. Burgard, and D. Fox, "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), 2000.

[71] Y. Liu, R. Emery, D. Chakrabarti, W. Burgard, and S. Thrun, "Using EM to learn 3D models with mobile robots," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2001.

[72] D. Fox, W. Burgard, and S. Thrun, "Markov localization for mobile robots in dynamic environments," *Journal of Artificial Intelligence Research*, vol. 11, 1999.

[73] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust monte carlo localization for mobile robots," *Artificial Intelligence*, pp. 99–141, 2000.

[74] D. B. Rubin, "Using the SIR algorithm to simulate posterior distributions," in *Bayesian Statistics 3* (J. M. Bernardo, M. H. DeGroot, D. V. Lindley, and A. F. M. Smith, eds.), pp. 395–402, Oxford, UK: University Press, 1988.

[75] S. Engelson, *Passive Map Learning and Visual Place Recognition.* PhD thesis, Yale University, 1994.

[76] A. Cassandra, *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes.* PhD thesis, Brown University, Department of Computer Science, Providence, RI, 1998.

[77] H. Shatkay and L. Kaelbling, "Learning topological maps with weak local odometric information," in *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 1997.

[78] H. Shatkay, *Learning Models for Robot Navigation.* PhD thesis, Brown, 1998.

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

[80] J. Banks, J. Carson, and B. Nelson, *Discrete-Event System Simulation*. Prentice Hall, 1996.

[81] K. Lari and S. Young, "The estimation of stochastic context-free grammars using the Insight-Outside algorithm," *Computer Speech and Language*, 1990.

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

[83] K. Murphy and M. Paskin, "Linear time inference in hierarchical HMMs," in *Neural Information Processing Systems (NIPS)*, 2001.

[84] K. Murphy, "Applying the junction tree alogirthm to variable-length DBNs," tech. rep., U.C Berkley, Computer Science Divison, 2001.

[85] H. Bui, S. Venkatesh, and G. West, "On the recognition of abstract Markov policies," in *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI)*, 2000.

[86] A. Nefian and M. Hayes, "An embedded HMM-based approach for face detection and recognition," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 1999.

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

[88] S. Levinson, "Continuosly variable duration hidden Markov models for automatic speech recognition," *Computer Speech and Language*, 1986.

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

[90] C. Mitchell and L. Jamieson, "On the complexity of explicit duration HMMs," *IEEE Transactions on Speech and Audio Processing*, vol. 3, May 1995.

[91] J. Pineau, N. Roy, and S. Thrun, "A hierarchical approach to POMDP planning and execution," in *Workshop on Hierarchy and Memory in Reinforcement Learning (ICML 2001)*, (Williams College, MA), June 2001.

[92] Nomadic technolgies, Inc., *Nomad 200 Hardware Manual*, 1997.

[93] Nomadic technolgies, Inc., *User's Manual*, 1997.

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

[95] S. Fahlman, "Faster-learning variations on back-propagation: An empirical study," in *Proceedings of 1988 Connectionist Models Summer School*, Morgan Kaufmann, 1988.

[96] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*. Wiley Interscience, 2000.

[97] P. Baldi and S. Brunak, *Bioinformatics The Machine Learning Approach*, ch. 9. MIT Press, 1998.

[98] J. H. Connell, *A Colony Architecture for an Artificial Creature*. PhD thesis, MIT, 1989.

[99] R. A. Brooks, "Intelligence without representation," *Artificial Intelligence Journal*, vol. 47, pp. 139–159, 1991.

[100] B. H. Juang and L. R. Rabiner, "A probabilistic distance measure for hidden Markov models," *AT&T Tech J.*, vol. 64, pp. 391–408, February 1985.

[101] A. J. Viterbi, "Error bounds for convlutional codes and an aymptotically optimal decoding algorithm," in *IEEE Transactions Information Theory*, pp. 260–269, 1967.

[102] G. Theocharous and S. Mahadevan, "Approximate planning with hierarchical partially observable Markov decision processes for robot navigation," in *Proceesings of the IEEE International Conference on Robotics and Automation (ICRA)*, (Washington, D.C.), 2002.

[103] R. Parr and S. Russell, "Reinforcemnet learning with hierarchies of machines," in *Advances in Neural Information Processing Systems*, vol. 10, (Cambridge, MA), pp. 1043–1049, MIT Press, 1997.

[104] R. Parr, *Hierarchical control and learning for Markov decision processes*. PhD thesis, University of California at Berkley, 1998.

[105] T. G. Dietterich, "Hierachical reinforcement learning with the MAXQ value function decomposition," *Journal of Artificial Intelligence Research*, 2000.

[106] M. Hauskrecht, N. Meuleau, L. P. Kaelbling, T. Dean, and C. Boutilier, "Hierachical solution of Markov decision processes using macro-actions," in *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 1998.

[107] R. M. Neal, "Probabilistic inference using Markov chain monte carlo methods," Tech. Rep. CRG-TR-93-1, University of Toronto, 1993.

[108] Z. Ghahramani and M. I. Jordan, "Factorial hidden Markov models," *Machine Learning*, pp. 245–275, 1997.

[109] S. Thrun, J. Langford, and D. Fox, "Monte carlo hidden Markov models: Learning non-parametric models of partially observable stochastic processes," in *Proc. of the International Conference on Machine Learning*, 1999.

[110] Y. Gotoh and H. F. Silverman, "Incremental ML estimation of HMM parameters for efficient training," in *Proceedings of the International Conference on Acoustics Speech and Signal Processing*, pp. 585–588, 1996.

[111] V. Digalakis, "On-line adaptation of hidden Markov models using incremental estimation algorithms," in *Proceedings of the European Conference on Speech Communication and Technology*, September 1997.

[112] D. Freitag and A. McCallum, "Information extraction with HMM structures learned by stochastic optimization," in *AAAI/IAAI*, pp. 584–589, 2000.

[113] A. Stolcke and S. Omohundro, "Hidden Markov model induction by Bayesian model merging," in *Neural Information Processing Systems (NIPS)*, 1992.

[114] D. Heckerman, "A tutorial on learning with Bayesian networks," Tech. Rep. MSR-TR-95-06, Microsoft Research, 1996.

[115] N. Friedman, "The Bayesian structural EM algorithm," in *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 1998.

[116] M. Brad, "Structure learning in conditional probability models via an entropic prior and parameter extinction," *Neural computation*, pp. 11:1155–1182, 1999.

[117] N. Friedman and D. Koller, "Being Bayesian about network structure," in *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 2000.