LEARNING PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES USING ABSTRACT ACTIONS

by

HAMED JANZADEH

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2012

To my family.

ACKNOWLEDGEMENTS

First of all, I would like to express the deepest appreciation to my supervising professor, Dr. Manfred Huber. His deep knowledge of the topic has made him my main source of education throughout this research project. His patience and skill in teaching, his enthusiasm in learning and his kindness and understanding in all situations will always be a great role model of my life.

I am grateful to my thesis committee members Dr. Farhad Kamangar and Dr. Vassilis Athitsos for their interest in my research and for taking time to serve in my thesis committee. I like to thank Dr. Gergely Zaruba, Mr. David Levine, Dr. Victoria Chen, Dr. Dick Schoech and Dr. Manfred Huber for giving me the opportunity to serve as a member in the Luminant and the Teleherence projects. I also like to thank Dr. Bahram Khalili, the graduate advisor, who has always been helpful and kind towards me and all other students.

Finally, I would like to express my deep gratitude to my parents, Hossein and Zahra, my sister Niloofar and my brother in law Syamak for their supportive role and for enduring my absence during all these years. I like to express an special thank to Bahar who made it possible for me to have an ease of mind and work on my thesis by helping me with my personal life and also by preparing the graphs and figures used in this document.

November 25, 2012

ABSTRACT

LEARNING PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES USING ABSTRACT ACTIONS

HAMED JANZADEH, M.S.

The University of Texas at Arlington, 2012

Supervising Professor: Manfred Huber

Transfer learning and Abstraction are among the new and most interesting research topics in AI and address the use of learned knowledge to improve learning performance in subsequent tasks. While there has been significant recent work on this topic in fully observable domain, it has been less studied for Partially Observable MDPs. This thesis addresses the problem of transferring skills from the previous experiences in POMDP models using high-level actions (Options) in two different kind of algorithms: value iteration and expectation maximization. To do this, this thesis first proves that the optimal value function remains piecewise-linear and convex when policies are made of high-level actions, and explains how value iteration algorithms should be modified to support options. The resulting modifications could be applied to all existing variations of the value iteration and its benefit is demonstrated in an implementation with a basic value iteration algorithm. While the value iteration algorithm is useful for the smaller problems, it is strongly dependent on knowledge of the model. To address this, a second algorithm is developed. In particular, expectation maximization algorithm is modified to learn faster from a set of sampled experiments

instead of using exact inference calculations. The goal here is not only to accelerate learning but also to reduce the learner's dependence on complete knowledge of the system model. Using this framework, it is also explained how to plug options in the model when learning the POMDP using a hierarchical EM algorithm. Experiments show how adding options could speed up the learning process.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

Nowadays, computers are playing a major role in solving our everyday problems and it is no longer possible to accomplish most of our routine tasks without their aid. The way we solve our real-life problems using computers is through first modeling the problems in some mathematical form and second developing computer algorithms to solve the formalized problems. Among all the different mathematical models we have to formalize our problems, the *Markov* processes are very useful when it comes to planning or decision making under uncertainty. This makes it very important to develop effective methods for mapping problems into these models and to design efficient learning algorithms to solve them.

The *Partially Observable Markov Decision Processes* (POMDPs) provide a broader definition and a better modeling of the uncertainties in the environment compared to MDPs. As a result, POMDPs are usually better models for formalizing the real-world problems and there already exist a significant number of different kinds of learning algorithms for these models, including, Value Iteration [1], Policy Gradient [2], Expectation Maximization algorithms [3], etc. However, almost all of these algorithms are still not effective for problems with more than a few thousand states. On the other hand, real-world problems usually require a huge (if not infinite) number of states if directly modeled as a Markov process.

While developing faster algorithms is effective for addressing this issue, another approach is to re-evaluate the methods we use to formalize our problems. *Abstraction* and *Transfer Learning* [4] are among the most interesting new topics in AI that

take advantage of this concept and help in solving more complicated problems using the existing algorithms but through systematic down-scaling of the problem size and reusing of already learned skills.

Unfortunately, Abstraction and Transfer Learning are not well studied topics for POMDPs so far. In this thesis, we show how transferring knowledge and abstraction is possible in the POMDP models through use of high-level actions (Options) and we present two different algorithms to learn optimal policies for these models created from high-level actions. The first learning algorithm we introduce is based on the conventional value iteration algorithms that currently exist for POMDPs. Some variations of the value iteration algorithm are among the fastest algorithms that exist for learning POMDPs. The second algorithm is an *Expectation Maximization* method. In this new approach we show how to learn high-level policies for model-less POMDP problems. In a model-less problem there is no knowledge of the environment's probability functions including those for state transition, observations and rewards.

In this work, a new POMDP problem is also introduced which is a simplified version of the Robocup Rescue Simulation[5] problem. This problem is very scalable and could be a useful testbed for evaluating abstraction methods on POMDPs.

Throughout the rest of this chapter, we explain some well-known re-enforcement learning concepts like Markov models, belief states, finite state controllers and options. Chapter 2 gives a review of some existing learning algorithms for the POMDP models that are related to this research work. In Chapter 3, we explain our two learning algorithms for abstract POMDP policies. Finally, the simulation environment, the implementation of the algorithms and the experimental results are explained in Chapter 4.

1.1 Markov Models

Consider the problem of navigating a robot in a room or a hallway. The robot has several actions to execute, including, move forward, turn left, turn right and etc. The aim could be to find the exit doorway or to search for an object in the environment.

The robot actions are not necessarily accurate. For example, the robot could try to move 10 inches forward but it is not able to do that accurately all the time. It could end up moving a little bit more or a little less than 10 inches. The robot could also diverge from moving in a straight line.

In this world, let us assume the time is discrete and each step of the time is called a time-step. The robot takes an action in each time-step and each action takes exactly one time-step to be executed. The robot has to make decisions about which action to take at each time-step in order to reach the defined goal and the method it uses to choose actions is called its policy.

In each problem, all different parameters that define what the world looks like at each time-step are called the world state. For example, the location of the robot and the obstacles in the environment could define the state of the world in the robot navigation problem. Each time a robot takes an action the world transits from one state to another state according to some probability distributions defined for this specific world or environment. As explained earlier, these state transitions are not necessarily deterministic because a robot could end up in different world states after executing an action because the action execution is not accurate.

The environment or the world also provides rewards to the robot at each time-step. The reward is simply a number and defines how well the robot has performed in each time-step and the robot is trying to maximize the total reward it receives or its *utility*. Using a reward system reduces the specific problem of designing a robot

that learns performing a particular task into the general problem of making a robot that maximizes the total reward it gains. In the navigation problem, the robot could receive a big positive reward when it finds the exit doorway or the object it is looking for and a reward of zero at any other time. It could also receive a small negative reward for the non-rewarding situations. The negative rewards could persuades the robot to find the goal as soon as possible if it wants to maximize its total reward. The amount of reward the robot receives depends on the action and the current state of the world.

A reward could also be discounted which means the reward for a given state and action is going to be less significant if it happens further in the future. With a discounted reward system, the robot could be motivated to find the goal as soon as possible even without adding negative rewards for actions. This means it makes the robot solve the problem efficiently. A discounted reward utility also has some other nice mathematical properties that will be discussed later.

A world like the one just explained is a *Markov model*, if the probability of the next state of the world depends only on the current state and the action chosen by the robot at the current time-step [6]. For example, in the navigation problem, the location of the robot in the next time-step should depend merely on the current location and the type of action chosen by the robot at the current time-step. If the next location depend on where the robot was or what have happened in the past, the model is not Markov.

1.1.1 MDPs

A specific type of Markov models which is called a *Markov Decision Process* (MDP) [6] is one in which the robot knows the state of the world accurately in each

time-step. In the robot navigation problem, the world is a MDP if the robot knows all the world parameters, e.g. its current location, very accurately all the time.

MDPs are not very realistic but they have really nice properties. A problem that is modeled by an MDP could be very easily solved using a significant number of efficient learning algorithms already developed for these models [7].

A finite MDP is formally defined by the tuple $< S, A, T, R, \gamma >$ where:

- S is a finite set of all the states of the world;
- A is a finite set of all the actions available to the agent;
- $T : S \times A \rightarrow \Pi(S)$ is the *State Transition Probability Function* and $T^{sa}_{s'} = P(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability of transiting to the state s' after taking the action a in the state s;
- $R : S \times A \rightarrow \mathbb{R}$ is the *Reward Function* and $R_{sa} = R(s, a)$ is the reward to receive after taking the action a in the state s;
- $\gamma \in [0, 1]$ is a *discount factor*. Discount factor defines how much a potential reward loses its value at each time-step. For example, if the discount factor is 0.8 and there is a reward of 10 for an specific state-action pair, the reward's effect on the utility of the agent is going to be equal to 8 if it happens in the second time-step, equal to 0.64 if it happens in the third time-step and so on.

1.1.2 POMDPs

A *Partially Observable Markov Decision Process* (POMDP) uses a broader definition of the Markov models. The difference with MDPs is that the world state is not known by the agent in a POMDP model; instead, probabilistic observations corresponding to the underlying state are received from the environment after taking each action.

The POMDP version of the robot navigation problem could be the robot not knowing its current location by default; instead, the robot could receive *observations* from the environment. For example, the robot bumper could sense when it hits an obstacle or the robot could have a radar sensor that measures its distance from the wall or any object in front of it. It is important to note that the observations could also be inaccurate in the same way as the state transitions. In the robot navigation problem, for example, the distance sensors might have a measurement error. This could be modeled by a probability distribution.

Compared to MDPs, POMDP models are more realistic representations of real-world environments and therefore being able to solve POMDP problems is a positive step towards solving more realistic problems.

A finite POMDP is formally defined by the tuple $< S, A, \Omega, I, T, Z, R, \gamma >$ where:

- S is a finite set of all the states of the world;
- A is a finite set of all the actions available to the agent;
- Ω is a finite set of all the observations the agent could receive from the environment;
- $I : \Pi(S)$ is the *Initial State Probability Function* and I_s is the probability of the world starting in state s;
- $T : S \times A \rightarrow \Pi(S)$ is the *State Transition Probability Function* and $T^{sa}_{s'} = P(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability of transiting to the state s' after taking the action a in the state s;
- $Z : S \times A \rightarrow \Pi(\Omega)$ is the *Observation Probability Function*. This defines $Z^{sa}_{z} = P(z_t = z | s_t = s, a_{t-1} = a)$ the probability of receiving the observation z after taking the action a and ending up in the state s;

- $R : S \times A \to \mathbb{R}$ is the *Reward Function* and $R_{sa} = R(s, a)$ is the reward to receive after taking the action a in the state s;
- $\gamma \in [0, 1]$ is the *Discount Factor*.

1.2 POMDP Policies

As explained through examples, the main reason for modeling an environment with a Markov model is to find a way to learn how to perform a task. *Learning* here means determining what action to take at each time-step in order to perform the task. This is called the agent's *policy*. A policy π therefore is a function that returns a choice of action at each time-step. An *optimal policy* is one that makes the optimal action choices to finish the task efficiently or in other words optimizes the expected utility.

In a MDP model, the agent can make decisions based on the current state of the world which is known to it. Therefore, the policy can be defined as a mapping from the world states to actions. A *deterministic policy* $\pi : S \to A$ is a direct mapping from the states to actions and a *stochastic policy* $\pi : S \to \Pi(A)$ is a mapping from each of the states to a probability distribution on all actions $\Pi(A)$.

In a POMDP, on the other hand, the world state is not known by the agent and it can therefore not define policies by mapping states to actions. That is why there should be a different way to model policies for POMDP models. Defining policies as a mapping from *Belief States* to actions [8] and also as a mapping from the agent's *Memory States* to actions (using *Finite State Controllers*) [9] are two of the most important methods that address this problem and that we also use in our work.

1.2.1 Belief State Mapping

A *Belief State* b is a probability distribution over all the world states. Belief state defines the probability of being at each of the individual states. In a discrete and finite Markov model, a belief state b is a vector of length $|S|$ of probability values and $b_s = P(s|b)$ defines the s^{th} element of this vector which is the probability of being in state s given b.

When the world starts in a POMDP model, the agent has an *Initial Belief State* I which is defined by the POMDP model. The initial belief state could be a uniform probability distribution over all the world states which means the robot has no idea where it is in the beginning.

The agent updates its belief state over time as it explores the world and receives observations. It can do this updating because the agent has the POMDP model, so it knows all the world states, the state transition probabilities, the observation probabilities, etc. Hence, when the agent takes and action and receives an observation, using inference it could update its assumptions about the probability of being in different states of the world. For example, when the navigating robot's world is an empty room and it hits an object, the robot figures it should be somewhere next to one of the walls and could update its belief state accordingly.

Formally, a belief state b is updated to the belief state b' or $\mathcal{T}(b, a, z)$ after taking action a and receiving observation z as defined below:

$$\mathcal{T}_{s'}(b, a, z) = b'_{s'} = \frac{\sum_s b_s T^{sa}_{s'} Z^{s'a}_z}{\sum_{s''} \sum_s b_s T^{sa}_{s''} Z^{s''a}_z} \tag{1.1}$$

Now, a POMDP policy can be defined as a function that maps from the current belief state to actions. This makes the policy function very complex because it is defined over the infinite space of belief states instead of the finite space of the world states. However, at least, it defines a way to model POMDP policies.

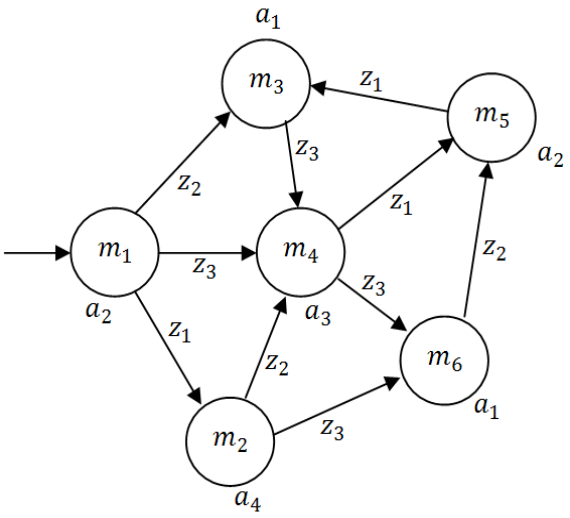

Figure 1.1 A sample Finite State Controller.

1.2.2 Finite State Controllers

Using *Finite State Controllers* (FSCs) is another well-known method to model POMDP policies. A FSC policy is a state machine with a finite number of states called *Memory States*. The agent has an internal memory and takes actions according to its memory state. Therefore, the function that maps the memory states to actions defines the policy. There is also a memory state transition function involved. The current memory state transits to another memory state at each time-step, based on the observation received from the environment. Figure 1.1 shows a sample finite state controller with 6 states.

Formally, a FSC is defined by the tuple $< M, A, \Omega, \pi, \mu, \nu, \eta >$ where:

- M is a finite set of memory states;
- A is the finite set of actions;
- Ω is the finite set of observations;
- π is the *action probability function* and $\pi_a^m = P(a_t = a | m_t = m)$ is the probability of taking action a in memory state m;

- μ is the *memory state transition function* and $\mu_{m'}^{mz} = P(m_{t+1} = m' | m_t = m, z_t = z)$ is the probability of transiting from memory state m to memory state m' if observation z is received;
- ν is the *initial memory state probability function* and $\nu_m = P(m_0 = m)$ is the probability of starting in memory state m; and
- η is the *termination probability function* and $\eta_m = P(Terminate|m)$ is the probability of terminating in memory state m.

1.3 Skill Abstraction and Options

There has been a significant amount of research on how to optimize the learning algorithms for planning on Markov models, yet most of the problems that can be solved in a reasonable time using the existing algorithms are still relatively small problems with at most a few thousand states. This is no way close to solving real-world problems like those we solve in our every day life as human beings. Out of the reasons, one could say maybe the learning algorithms are still very slow and that's true to some extent. However, the other important factor to consider is that humans are very efficient at abstracting problem models before planning. We make high-level decisions on how to reach a goal and then we plan for the details hierarchically, or we might already know how to handle some of the details from our past experience.

Abstraction is fortunately studied in the AI world too, although, mostly as part of traditional planning and on MDPs. For the later, Precup, Sutton, and Singh in 1998 [10], introduce abstract actions for the *Semi-MDP* models which they call *Options*. An option is an already learned policy for performing a sub-task that could be transferred and used in solving other problems as an additional action in the MDP model. For example, when a robot learns how to go from one room to another room,

it could use that as one of its skills or actions when learning to plan for another problem like cleaning the house.

Learning the optimal policy for the MDPs when options are added is very similar to the original case. Each option is a regular policy that uses lower-level actions to execute. For each option, after being learned, some statistics including the expected reward and transition probability functions are calculated and preserved. The expected reward function R_{so} defines the expected sum of the discounted rewards that could be obtained if executing the option o in state s and before the option terminates. The state transition function $T_{s'}^{so}$ gives the probability of option o terminating in state s' if executed at the state s. These functions obviously look very similar to those of the regular actions, and with slight modifications to the algorithms and the model, options can be added to the model's set of actions and used for learning of future problems.

The only reason options could not be directly applied to the MDPs is that they take longer than a single time-step to execute. In fact, an option could take a variable amount of time - from one time-step to infinitely long - before terminating. This contradicts the definition of the Markov models and disturbs the reward discounting. Semi Markov Decision Processes (SMDPs) [11], however, redefine the state probability dependency and allow the options to be used in the model without any contractions. The discounting problem is also solvable if the discount factor is included in the option's state transition function. At this point, the Bellman equations of the regular MDPs with a small modification can be used again to benefit from options.

CHAPTER 2

RELATED WORK: LEARNING POMDP POLICIES

The previous section explained how POMDP policies could be represented formally. However, the actual goal is to find the *Optimal Policy*. The optimal policy is a policy that maximizes the agent's utility.

We explained earlier that a policy which optimizes the expected total reward or expected utility is equivalent to the one that performs the task optimally if the reward function is defined properly for the given task. As a result, we are trying to address the problem of finding the policy that maximizes the expected utility. This is called *Reinforcement Learning*.

For a MDP model, the utility or the value function of a given policy is a function of the world states $V^{\pi} : S \rightarrow \mathbb{R}$ that defines the expected total reward to gain using policy π if starting from the state s. This value function can be easily defined recursively using the Bellman equations [6] as follows:

$$V^{\pi}(s) = \sum_{a} \pi(s,a)\Big(R_{sa} + \gamma \sum_{s'} T_{s'}^{sa} V^{\pi}(s')\Big). \tag{2.1}$$

In Equation 2.1, $\pi(s,a) = P(a|s,\pi)$ is the probability of taking action a at state s using policy π. Using dynamic programming with this equation, the value of a policy could be easily calculated for each state. However, the goal is not just to calculate the value of a single policy, but to find the optimal policy. This is possible, too. Let us define V^* as the value of the optimal policy. Then, we will have:

$$V^{*}(s) = \max_{a} \Big(R_{sa} + \gamma \sum_{s'} T_{s'}^{sa} V^{*}(s')\Big). \tag{2.2}$$

This value function can also be calculated very straightforwardly. We set all the values equal to zero in the beginning and then will iteratively update the function value for all of the states using Equation 2.2. This algorithm is called *Value Iteration*. A discounted reward function shows its importance here and guarantees the value function will converge eventually in all of the MDP problems. Having the optimal value function in hand, now the optimal policy is one that in each state chooses the action which maximizes the utility according to the Equation 2.2.

The value iteration algorithm that was just explained works only for MDPs because it defines the optimal policy as a function of states and also relies on the fact that a policy is a mapping from states to actions. This is not going to work for a POMDP model, because the agent has no knowledge of the current state. Even if it computes the optimal value function this way, it is not able to execute a policy which is a function of the states.

2.1 Value Iteration Algorithm

The value function for a POMDP model could be defined as a function of the current belief state which is available to the agent though:

$$V_n^*(b) = \max_a \sum_s b_s \Big(R_{sa} + \gamma \sum_{s'} \sum_z T_{s'}^{sa} Z_{s'}^{sa} V_{n-1}^*(\mathcal{T}(b,a,z)) \Big), \tag{2.3}$$

where, $V_n^*(b)$ is the value of the optimal policy at belief state b in the n^{th} iteration and $\mathcal{T}_{s'}(b,a,z) = b'(s')$ is the probability of state s' under the updated belief state b' after taking action a and observing z [8].

2.1.1 Modeling the Value Function using α-vectors

In Equation 2.3, value function is again defined recursively but over belief states instead. This unfortunately poses a huge problem: it is not possible to directly solve

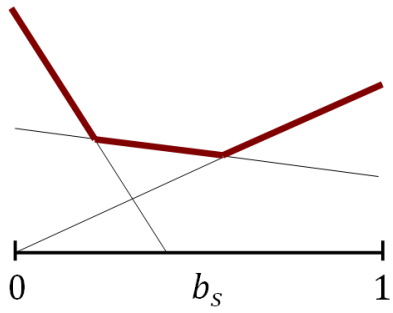

Figure 2.1 A sample optimal value function for a two state POMDP problem.

this equation using dynamic programming or any other finite time algorithm like we did for MDPs, because this function is defined over a continuous space of belief states instead of a finite set of states. As a result, one iteration of dynamic programming or value iteration algorithm would require an infinite amount of time to compute. Also, representing the value function requires an infinite amount of space.

In 1973, Smallwood [8] shows that the optimal value function for POMDPs has some nice properties that makes it possible to be solved using iterative algorithms. Smallwood shows that the optimal value function is *Piece-wise Linear and Convex* over the space of belief states and could be represented by a finite set of vectors. Figure 2.1 illustrates what a piece-wise linear and convex value function looks like for a sample POMDP problem with two states.

As can bee seen, there are a number of linear functions in the space, each could be represented by a vector called *alpha vector*, and the value at each belief state is equal to the maximum value that any of the linear functions return for that belief state. This is formally shown in Equation 2.4:

$$V_n^*(b) = \max_{\alpha} \left[\sum_s b_s \alpha_s(n) \right] \tag{2.4}$$

where, $\alpha(n)$ is one of the α-vectors on iteration n. The size of each alpha vector is equal to the number of states $|S|$.

Using α-vectors, it is now possible to formulate the optimal value function in a finite space and it just remains to find an algorithm to calculate these vectors.

2.1.2 Calculation of α-vectors

Using an iterative method again, α-vectors can be calculated recursively using the α-vectors of the previous iteration:

$$\alpha_s(n) = R_{sa} + \gamma \sum_{s'} T_{s'}^{sa} Z_z^{sa} \alpha_{s'}(n-1) \quad \forall a, z, \alpha. \tag{2.5}$$

This is called the value iteration algorithm for POMDPs. After computing the optimal value function, a policy is extracted from this function. Each alpha vector has a mapping action which is the one it was created with (in Equation 2.5). The optimal policy is the one that returns the action associated to the α-vector which maximizes the value function for a given belief state:

$$\pi^*(b) = \underset{a(\alpha)}{argmax} \sum_s b_s \alpha_s \tag{2.6}$$

This algorithm has its own weakness yet. The problem is that the number of α-vectors grows rapidly after each iteration. This can make the algorithm computationally very expensive or even implausible, if it is implemented the exact way. However, there already exists a lot of work since 1973 on fixing this problem and making different variations of the value iteration algorithm that are computationally less expensive. Most of these algorithms usually work by approximating the α-vectors through either sampling the belief state space or by defining bounds on the value function [1][12].

2.2 Expectation Maximization Algorithm

Optimizing the policy using the *Expectation Maximization* (EM) algorithm [13] is another approach to find the optimal policy for POMDP models [3]. This rela-

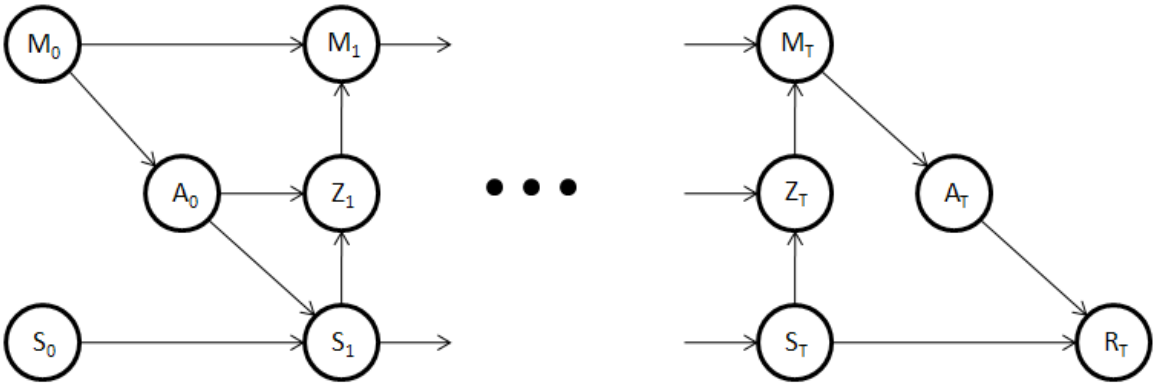

Figure 2.2 A Bayesian Network that defines the probability of receiving a boolean reward at time-step T.

tively new learning method which is almost as fast as some advanced value iteration algorithms provides a powerful framework for applying various useful techniques like hierarchical learning and approximation.

2.2.1 Modeling the Problem using Bayesian Networks

In [3], Toussaint et al. reduce the problem of policy learning for POMDPs into a parameter learning problem for Bayesian Networks (BN). This method uses a Bayesian network to define the probability of receiving a boolean reward $\hat{r} \in \{0, 1\}$ at the end of a finite horizon using a policy that is modeled by a Finite State Controller (FSC). All the variables that contribute to this probability value and their dependencies are depicted in a BN using graphical models.

Figure 2.2 illustrates a Bayesian network that includes all of the variables contributing to the probability $\mathrm{P}(\hat{r}_T = 1)$ and defines their conditional dependencies. $\hat{r}_T$ is a boolean reward that is received at the end of the finite horizon of length T. This network consists of T time-slices each for one time-step $t = 1...T$. Each slice of the network defines the probabilities of variables s_t (the state of the world), a_t (the action taken), z_t (the observation received) and m_t (the memory state of the

FSC policy) at the time-step t. Connections define conditional dependencies of the variables. For example, the probability of taking an action depends on the memory state of that time-step, so there is an arrow from a memory state to an action in each slice. According to the definition of the Markov processes, the reward received at the end of horizon depends only on the last action taken and the state of the world at the last time-step.

In order to follow the same definition for the reward function as explained for the Markov processes, the probability of receiving the boolean reward $\hat{r}$ is considered to be equal to the normalized value of the actual reward defined by the Markov model:

$$\hat{r}_T = P(\hat{r} = 1|T) = \frac{R(s_T, a_T)}{\sum_s R(s, a_T)}. \tag{2.7}$$

In this network, the *Conditional Probability Distributions* (CPDs) of the variables are defined either by the POMDP model or the FSC policy. The POMDP model defines the probabilities of the states s_t, the observations z_t and the reward as follows:

- $P(s_0 = s) = I_s$ is the initial state probability,
- $P(s_{t+1} = s'|s_t = s, a_t = a) = T^{sa}_{s'}$ is the state transition probability,
- $P(z_t = z|s_t = s, a_{t-1} = a) = Z^{sa}_z$ is the observation probability, and
- $P(\hat{r}_T = 1|a_T = a, s_T = s) = \hat{r}_T$ is the boolean reward probability as defined in Equation 2.7.

The FSC also defines the probabilities of actions a_t and the memory states m_t.

- $P(m_0 = m) = \nu_m$ is the initial memory state probability,
- $P(m_{t+1} = m'|m_t = m, z_t = z) = \mu^{mz}_{m'}$ is the memory state transition probability, and
- $P(a_t = a|m_t = m) = \pi^m_a$ is the action probability.

Now, using a BN optimization technique we could maximize the probability of receiving a reward at the end of horizon $P(\hat{r} = 1|T)$ through modifying the CPDs that are related to the policy (ν, μ and π), while the CPDs that are related to the POMDP model remain fixed. This is because we are adjusting the policy not the world model.

The problem is not solved yet: we aim to maximize the total discounted reward not the expected reward to receive at a specific time-step. Toussaint et al. [3] solve this problem by defining one BN for the probability of reward at the end of each horizon limit $T = [0...\infty]$ and redefining the optimization objective function as follows:

$$\theta^* = \underset{\theta}{argmax} \sum_{T=0}^{\infty} \gamma^T P(\hat{r} = 1|T, \theta), \tag{2.8}$$

wherein, $\theta = \{\pi, \mu, \nu\}$ is the set of all CPDs or parameters that define the policy and θ^* is the set of parameters for the optimal policy.

Before doing anything to solve this equation, let's first make our life easier by defining the objective function using the log-likelihood of the utility which won't change the result:

$$\theta^* = \underset{\theta}{argmax} \log \Big(\sum_{T=0}^{\infty} \gamma^T P(\hat{r} = 1|T, \theta) \Big) \tag{2.9}$$

2.2.2 Learning the Optimal Policy using EM

To solve the optimization problem given in Equation 2.9, we need to calculate the probability $P(\hat{r} = 1|T, \theta)$ for each θ and T. This probability is calculated using inference on the BN model and in the simplest form this could be shown by:

$$\theta^* = \underset{\theta}{argmax} \log \Big(\sum_{T=0}^{\infty} \gamma^T \sum_{X} P(\hat{r} = 1, X|T, \theta) \Big) \tag{2.10}$$

wherein, $X =< m_0, s_0, z_0, a_0..., m_T, s_T, z_T, a_T >$ is the set of all of the hidden variables in the network.

Using Equation 2.10, one can use any available optimization technique (e.g., expectation maximization or gradient ascent) to find the optimal policy. In this section, we explain how to solve this problem using the EM algorithm.

The first thing to do before EM could be used is to convert the log of sums equation into a sum of logs by replacing the objective function with a lower bound obtained using Jensen's inequality [13]. This gives a mathematically much nicer formulation of the problem. For Jensen's inequality to be applied, we need to convert the objective function in Equation 2.10 to this form: $log(\sum_X P(X)F(X))$. In order to do this, first, the objective function is multiplied by a constant factor $1-\gamma$. Second, it is multiplied and divided by an arbitrary probability distribution over X that we name $q(X)$. These two operations do not change the result of optimization, but will let us have $P(X) = q(X)$ and $F(X) = (1-\gamma)\gamma^T P(\hat{r}=1, X|T,\theta)/q(X)$. The result is:

$$\theta^* = \underset{\theta}{argmax} \log\Big(\sum_{T=0}^{\infty}\sum_{X}(1-\gamma)\gamma^T \frac{q(X)P(\hat{r}=1,X|T,\theta)}{q(X)}\Big). \tag{2.11}$$

Applying Jenson's inequality changes the objective function to this form:

$$\theta^* \approx \underset{\theta}{argmax} \sum_{T=0}^{\infty}\sum_{X}(1-\gamma)\gamma^T q(X) log\big(P(\hat{r}=1,X|T,\theta)\big) \tag{2.12}$$

Now, the EM algorithm can be used to optimize the policy using Equation 2.12 through several iterations of the E (Expectation) and M (Maximization) steps. In the E-step, the objective function is maximized w.r.t. $q(X)$ while holding θ fixed which results in:

$$q(X) = P(X|\hat{r}=1,T,\theta). \tag{2.13}$$

In the M-step, the objective function is maximized w.r.t. θ while holding $q(X)$ fixed:

$$\theta^{n+1} = \underset{\theta^*}{argmax} \sum_{T=0}^{\infty}\sum_{X}\Big[(1-\gamma)\gamma^T P(X|\hat{r}=1,T,\theta^n) log P(\hat{r}=1,X|T,\theta^*)\Big]. \tag{2.14}$$

The M-step can be solved by finding where the gradient of Equation 2.14 w.r.t. θ is equal to zero. Lagrange Multipliers [13] are needed to enforce the optimization constraints which basically say all of the probability values should remain in the range $[0, 1]$ and should add up to 1.

Optimizing Equation 2.14 using Lagrange Multipliers will give us the following results for the E and the M Steps. In the E-step:

$$E[\pi_a^m] = \sum_{T=0}^{\infty} \sum_{t=0}^{T} \gamma^T P(m_t = m, a_t = a | \hat{r} = 1, T, \theta^{(n)}) \tag{2.15}$$

$$E[\mu_{m'}^{mz}] = \sum_{T=0}^{\infty} \sum_{t=0}^{T-1} \gamma^T P(m_t = m, z_{t+1} = z, m_{t+1} = m' | \hat{r} = 1, T, \theta^{(n)}) \tag{2.16}$$

$$E[\nu_m] = \sum_{T=0}^{\infty} \gamma^T P(m_0 = m | \hat{r} = 1, T, \theta^{(n)}) \tag{2.17}$$

And, in the M-step:

$$(\pi_a^m)^{(n+1)} = \frac{E[\pi_a^m]}{\sum_{m'} E[\pi_a^{m'}]} \tag{2.18}$$

$$(\mu_{m'}^{mz})^{(n+1)} = \frac{E[\mu_{m'}^{mz}]}{\sum_{m''} E[\mu_{m''}^{mz}]} \tag{2.19}$$

$$(\nu_m)^{(n+1)} = \frac{E[\nu_m]}{\sum_{m'} E[\nu_{m'}]} \tag{2.20}$$

It is obvious the E-step is still too complex, because it requires too many inference operations (in fact an infinite number of them) to be calculated on the Bayesian networks we built. Toussaint et al. [3] suggest that if the *Forward-Backward* algorithm is used to calculate the inferences (which is a rational way to do inference), the Forward and Backward messages could be calculated on all of the networks simultaneously, because they are all subsets of each other. Also, both the Forward and the Backward messages could be truncated at some time length when the impact of the message is negligible due to the effect of the discount factor. The later suggestion gets rid of the problem with the infinite number of inferences required, because the

sum in the E-step is not calculated for the T values up to infinity. Please refer to the original paper for details.

CHAPTER 3

LEARNING ABSTRACT POMDP POLICIES

Abstraction and Transfer learning are effective methods to reduce the problem complexity and effectively speed up the algorithms when learning optimal policies for Markov decision processes [4].

This topic is studied extensively on the Markov Decision Processes; however, the Partially Observable Markov Decision Processes are still suffering from the absence of effective models and learning algorithms that support abstraction and transffering of skills. Unfortunately, the techniques developed for MDP models are not directly applicable to the POMDPs because the learning algorithms are completely different and a lot more complicated in the latter case. Most of this comes from the value function being defined over the infinite space of belief states and complex methods being used to formalize it using a finite number of parameters.

In this chapter, we explain two different methods we developed for learning policies with high-level actions (options) for POMDP models. Learning policies using abstract actions and removing the unnecessary primitive actions could reduce the model size and speed up the learning algorithm. In addition, if an already learned policy to perform one task could be transferred to another environment as a new action, it could speed up the process of learning how to perform other tasks.

3.1 POMDP Options

As explained earlier too, an option is a policy that defines how to perform a sub-task or an abstract action. As a result, a POMDP option is also a POMDP

policy and is defined the same way as the POMDP policies are defined. This means a POMDP option could be defined by mapping belief states to actions using α-vectors or by mapping memory states to actions using FSCs. An option has some sort of start and termination condition too.

3.1.1 Belief State Mapping Options

If an option is defined as a mapping of belief states to actions, there needs to be a method to represent it using a finite space of parameters. Since a POMDP option is a regular POMDP policy, it is possible to represent its value function using a finite set of vectors. To distinct these vectors from the vectors defining the actual policy's value function, let us name the vectors representing an option's value function as β-vectors.

The equation for the expected discounted reward to get using a POMPD option o at a particular belief state b is similar to the equation for the value function of a regular POMDP policy at belief state b. As a result, we have:

$$r_b^o = R(b, o) = \max_l \sum_s b_s \beta_s^{o,l}, \tag{3.1}$$

where, r_b^o is the expected reward to get executing option o at belief state b and $\beta_s^{o,l}$ is the l^{th} β-vector of option o. The option's policy is also defined as follow:

$$\pi^o(b, a) = \begin{cases} 1 & \text{if } a = \underset{a(\beta^o)}{argmax} \sum_s b_s \beta_s^o, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

This means, the action $a(\beta^o)$ associated to vector β^o is the action to be chosen at belief state b by option o, if the corresponding vector is the one maximizing Equation 3.1 at belief state b. As you can see, the policy function is piece-wise constant over b, because it returns same results in a belief state area where a β-vector is dominant.

We define the initialization and termination probabilities also as functions of belief states: $\nu^o(b)$ and $\eta^o(b)$ respectively. Later on, we explain what conditions and limitations should be applied to these functions in order for our value iteration algorithm to work.

3.1.2 Finite State Controller Options

An option could also be defined using a FSC the same way as a regular POMDP policy is defined using FSCs. Therefore, for each option, there are a set of memory states, a function π that maps from memory states to actions, a function μ that defines the memory state transition, and functions ν and η defining the initialization and termination probabilities over memory states. Please read the formal definition of FSCs in Chapter 1.

The expected reward to get with an option defined this way could be calculated using dynamic programming with equation below:

$$r^o_{ms} = \sum_a \pi^o(m,a)\Big(R(s,a) + \gamma \sum_{s',z,m} T^{sa}_{s'} Z^{s'a}_z \mu^{mz}_{m'} r^o_{m's'}\Big), \tag{3.3}$$

where, r^o_{ms} is the expected discounted reward to get with option o at state s and memory state m. Therefore, the expected reward to get running option o at belief state b is:

$$r^o_b = R(b,o) = \sum_{s,m} b_s \nu^o_m r^o_{ms}. \tag{3.4}$$

In practice, we do not need to calculate the expected rewards for each option, if we are using the EM algorithm (as will be explained later). However, it is possible to use FSC options for value iteration algorithm too where the expected rewards must be precomputed in this case.

The option's initialization and termination probabilities are defined as functions of memory states, when an option is defined using FSCs.

3.2 Value Iteration for Abstract Policies

The first method we suggest for learning abstract policies for POMDPs is a modified version of the value iteration algorithm. Some variations of the regular value iteration algorithm for learning POMDP policies from primitive actions are among the fastest that currently exist.

It is possible to learn POMDP policies with primitive actions using iterative algorithms, first of all because it has been proven that the optimal value function made from primitive actions is piece-wise linear and convex and as a result could be formalized with a finite set of vectors. Secondly, it is possible to calculate these vectors recursively using an iterative algorithm we call value iteration (as described in section 2.1).

In this section, we prove the optimal POMDP value function calculated using abstract actions remains piece-wise linear and convex and could be represented using a finite set of vectors. We also show what the recursive equation for calculating these vectors looks like. This will give us a new value iteration algorithm that works with abstract actions.

3.2.1 Generalized Belief State Update Function

One of the main pre-requisites of the value iteration algorithm for POMDPs is to have a belief state because the value function is defined over the belief state space. As a result, it is very important to show how the belief state is updated if options are available in the POMDP model.

Here, we present a more generalized definition of the belief state update to handle the options in the value function. In the conventional form, the belief state is updated once after each action is executed and an observation is received. An option, however, is a high level action and it executes a sequence of lower-level actions before

terminated. Therefore, in the generalized form we define a belief state update over a sequence or trajectory of action-observation pairs we call a *History*.

A history h is a trajectory of action-observation pairs defined as:

$$h =< a_1, z_1, a_2, z_2, ..., a_L, z_L >, \tag{3.5}$$

wherein, a_t is the action taken and z_t is the observation obtained at time-step t. L is the length of the history. Lets also define h_k to be a prefix of h which contains the first k pairs.

Now, we show how a belief state could be updated for a given history. $b' = \mathcal{T}(b, h)$ is the updated belief state b after experiencing the history h.

$$\mathcal{T}_{s'}(b, h_k) = \frac{\sum_s \mathcal{T}_s(b, h_{k-1}) T_{s'}^{sa_k} Z_{z_k}^{s'a_k}}{\sum_{s''} \sum_s \mathcal{T}_s(b, h_{k-1}) T_{s''}^{sa_k} Z_{z_k}^{s''a_k}} \tag{3.6}$$

where, $T_{s'}^{sa} = T(s, a, s')$ and $Z_z^{sa} = Z(s, a, z)$ are the shorter version definitions of the state transition and the observation functions. If we expand this recursive function we will see that the belief state function is still a linear function of the starting belief state. We use the $\psi_{ss'}^h$ variable just to simplify the equations in the later sections.

$$\begin{aligned}
\mathcal{T}_s(b, h_1) &= \frac{\sum_{s_0} b_{s_0} T_s^{s_0 a_1} Z_{z_1}^{sa_1}}{\sum_{s_0} \sum_{s_1} b_{s_0} T_{s_1}^{s_0 a_1} Z_{z_1}^{s_1 a_1}}, \\
\mathcal{T}_s(b, h_2) &= \frac{\sum_{s_0} \sum_{s_1} b_{s_0} T_{s_1}^{s_0 a_1} Z_{z_1}^{s_1 a_1} T_s^{s_1 a_2} Z_{z_2}^{sa_2}}{\sum_{s_0} \sum_{s_1} \sum_{s_2} b_{s_0} T_{s_1}^{s_0 a_1} Z_{z_1}^{s_1 a_1} T_{s_2}^{s_1 a_2} Z_{z_2}^{s_2 a_2}}, \\
&\vdots \\
\mathcal{T}_s(b, h_k) &= \frac{\sum_{s_0, s_1, ... s_{k-1}} b_{s_0} T_{s_1}^{s_0 a_1} Z_{z_1}^{s_1 a_1} ... T_s^{s_{k-1} a_k} Z_{z_k}^{s, a_k}}{\sum_{s_0, s_1, ... s_k} b_{s_0} T_{s_1}^{s_0 a_1} Z_{z_1}^{s_1 a_1} ... T_{s_k}^{s_{k-1} a_k} Z_{z_k}^{s_k, a_k}} \qquad (3.7) \\
&= \frac{\sum_{s_0} b_{s_0} \psi_{s_0 s}^{h_k}}{\sum_{s'} \sum_{s_0} b_{s_0} \psi_{s_0 s'}^{h_k}}. \qquad (3.8)
\end{aligned}$$

3.2.2 Value Function for the Optimal Abstract Policy

In this section, we will show that the nice property of the optimal value function which is being piecewise linear and convex will be preserved after incorporating the high-level actions in the optimal policy.

First, we need to re-define the POMDP value function that was first explained in Equation 2.3 again for the case where options are available. The value function could be re-defined using the generalized form of the belief state update function we just explained:

$$V_n^*(b) = \max_o \Big(R_b^o + \sum_{h \in H} P_h^{o,b} \gamma^L V_{n-1}^*(\mathcal{T}(b,h)) \Big), \tag{3.9}$$

where, $V_n^*(b)$ is the value of the optimal policy at the belief state b in the iteration number n, h is a history and H is the finite set of possible histories in the limited horizon, L is the length of the specific history h.

R_b^o is the expected reward to get after executing the option o at the belief state b. We assume an option is a regular POMDP policy itself; therefore, like all POMDP policies an option's utility or its expected reward function is piecewise linear on b and could be represented with a set of vectors. Let's call these vectors each representing a linear equation the β-vectors.

$$R_b^o = \max_i \sum_s b_s \beta_s^{o,i} \tag{3.10}$$

$P_h^{o,b} = P(h|o,b)$ is the probability of the history h happening if option o starts the execution on the belief state b and terminates at the end of history h:

$$
\begin{aligned}
P(h_k|o,b) &= \sum_{s_0} \Big[\nu^o(b) b_{s_0} \\
&\quad \sum_{s_1} \Big[\pi^o(b,a_1) T_{s_1}^{s_0 a_1} Z_{z_1}^{s_1 a_1} \big(1-\eta^o(b)\big) \\
&\quad\quad \sum_{s_2} \Big[\pi^o(\mathcal{T}(b,h_1),a_2) T_{s_2}^{s_1 a_2} Z_{z_2}^{s_2 a_2} \big(1-\eta^o(\mathcal{T}(b,h_1))\big) \\
&\quad\quad\quad \vdots \\
&\quad\quad\quad \sum_{s_k} \Big[\pi^o(\mathcal{T}(b,h_k),a_k) T_{s_k}^{s_{k-1} a_k} Z_{z_k}^{s_k,a_k} \eta^o(\mathcal{T}(b,h_k)) \Big] ... \Big] \Big] \qquad (3.11) \\
&= \pi^o(b,a_1) ... \pi^o(\mathcal{T}(b,h_k),a_k) \\
&\quad \nu^o(b) \big(1-\eta^o(b)\big) ... \big(1-\eta^o(\mathcal{T}(b,h_{k-1}))\big) \eta^o(\mathcal{T}(b,h_k)) \\
&\quad \sum_{s_0,...,s_k} \Big[b_{s_0} T_{s_1}^{s_0 a_1} Z_{z_1}^{s_1 a_1} ... T_{s_k}^{s_{k-1} a_k} Z_{z_k}^{s_k,a_k} \Big] \qquad (3.12) \\
&= A_{h_k}^o W_{h_k}^b \qquad (3.13)
\end{aligned}
$$

where, $\pi^o(b,a)$ is the probability of action a being selected by option o at belief state b, $\nu^o(b)$ is the probability of belief state b being an initial state for option o and $\eta^o(b)$ is the probability of option o terminating in belief state b.

We have split the history probability into two terms: the *Agent Effect* A_h^o and the *Model Effect* W_h^o as also introduced in [14]. The Agent Effect is the product of all the action, initialization and termination probability terms, while the Model Effect is the product of all initial state, state transition and observation terms. Splitting the equation this way, will help us simplify the math later in this section.

Theorem: The optimal value function $V_n^*(b)$ in Equation 3.9 is piecewise linear and convex, if the option's initialization and termination functions return constant values in bounded areas of the belief state simplex:

$$V_n^*(b) = \max_l \Big(\sum_s b_s \alpha_s^l(n) \Big) \tag{3.14}$$

Proof: First of all, the claim is true for $n = 0$ which is when the last option is executed and the end of our finite horizon is reached. This is because the total reward we could get when we have reached the end of time is equal to zero and zero is linear $V_0^*(b) = 0$. Here, n is the iteration number which also means how many options could be executed before reaching the end of horizon. Obviously, after iterating for infinite times we get the converged value function for an infinite horizon.

Using induction now we could show $V_n^*(b)$ is piecewise-linear and convex in b for all of the values of n. For this reason, we assume the hypothesis holds for V_{n-1}, then we prove this applies to V_n as well. By using Equation 3.14 for V_{n-1}, we have:

$$V_{n-1}^*(\mathcal{T}(b,h)) = \max_l \Big(\sum_s \mathcal{T}_s(b,h) \alpha_s^l(n-1) \Big). \tag{3.15}$$

Plugging equations 3.15 and 3.10 into Equation 3.9, we get:

$$V_n^*(b) = \max_o \Big(\Big[\max_i \sum_{s_0} b_{s_0} \beta_{s_0}^{o,i} \Big] + \sum_{h \in H} P_h^{o,b} \gamma^L \Big[\max_l \sum_s \mathcal{T}_s(b,h) \alpha_s^l(n-1) \Big] \Big). \tag{3.16}$$

Finding the location of a particular belief state inside the belief state simplex, we can select the vector that provides the largest value for the given belief state [8]. Calling this vector as α^*, we can remove the max operation of the α-vectors from the Equation 3.16. The same argument applies to the β-vectors of the option:

$$V_n^*(b) = \max_o \Big(\sum_{s_0} b_{s_0} \beta_{s_0}^{o,*} + \sum_{h \in H} P_h^{o,b} \gamma^L \sum_s \mathcal{T}_s(b,h) \alpha_s^*(n-1) \Big). \tag{3.17}$$

In the Equation 3.17, moving $P_h^{o,b}$ inside the inner sum and multiplying it with $\mathcal{T}_s(b,h)$ will cause the Model Effect W_h^b from the Equation 3.13 to be cancelled out with the

normalizing factor (the denominator) in the Equation 3.7. Applying these changes to the Equation 3.17 will result in:

$$\begin{aligned} V_n^*(b) &= \max_o \Big(\sum_{s_0} b_{s_0} \beta_{s_0}^{o,*} + \sum_{h \in H} \gamma^L \sum_s \Big[A_h^o \sum_{s_0} b_{s_0} \psi_{s_0 s}^{h_k} \Big] \alpha_s^*(n-1) \Big) \\ &= \max_o \Big(\sum_{s_0} b_{s_0} \Big[\beta_{s_0}^{o,*} + \sum_s \alpha_s^*(n-1) \big(\sum_{h \in H} \gamma^L A_h^o \psi_{s_0 s}^{h_k} \big) \Big] \Big). \end{aligned} \tag{3.18}$$

Please notice that all of the terms inside the brackets in Equation 3.18 except for A_h^o are very clearly constant with respect to belief state b. Here, we will show that A_h^o is also constant in a vicinity of the belief state point b.

A_h^o is a product of option's action probabilities, initialization probability and termination probabilities (see Equation 3.12). The product of action probabilities get a value of either 0 or 1 for each belief state point b depending on what the β-vectors look like and this value is constant in the vicinity of that point as also visible in Equation 3.2. This is because the policy function is piecewise linear and the max operation will return the same result in an area where the corresponding β-vector has the superior value. If we assume the initialization and termination probabilities ν^o and η^o are also piece-wise constant over b, then for a bounded area of the belief simplex around the belief state point b the value of A_h^o is constant and is not a function of b.

Having piece-wise constant initialization and termination probability functions means the probability of initialization and termination should be constant in a vicinity of the given belief state point. This will partition the belief state simplex into a finite set of sections. For example, if the probability of termination is either 0 or 1 for each belief state, this will partition the belief state simplex in two sections and would respect our assumption.

Hence, given this condition, $\hat{\Psi}^o_{s_0 s} = \sum_{h \in H} \gamma^L A^o_h \psi^{h_k}_{s_0 s}$ is a constant value with respect to b and we can rewrite the Equation 3.18 as:

$$V^*_n(b) = \max_o \Big(\sum_{s_0} b_{s_0} \Big[\beta^{o,*}_{s_0} + \sum_s \alpha^*_s(n-1) \hat{\Psi}^o_{s_0 s} \Big] \Big). \tag{3.19}$$

As a result, the Equation 3.19 is also piecewise linear and convex in b if the option's initialization and termination functions return constant values in bounded areas of the belief state simplex and the proof is complete. □

3.2.3 Value Iteration Algorithm

Having the recursive equation for calculating the α-vectors in hand, it is going to be very simple to build the value iteration algorithm for learning the optimal policy using it. The value iteration algorithm is just going to create more α-vectors out of the existing ones in each iteration and at the end the dominating vectors will be in this set. The different algorithms used for pruning the dominated α-vectors and speeding up the process by sampling will all remain the same as for the regular POMDP case. In this thesis, we just explain how value iteration algorithms could be modified to calculate the α-vectors using options.

To calculate the α-vectors we use the results we got in Equation 3.19. This means our recursive equation for calculating the α-vectors is as follows:

$$\alpha_s(n) = \beta^{o,*}_s + \sum_{s'} \alpha_{s'}(n-1) \hat{\Psi}^o_{ss'} \quad \forall o, \alpha. \tag{3.20}$$

In practice, we don't need to go through all the possible histories to calculate the value of $\Psi^o_{ss'}$ as explained in the Equation 3.19. Instead, we could use a sample set of histories to estimate the expected value for this parameter and we could pre-compute that for each option before-hand.

Another method to expedite the calculations is to approximate the value function by running the value iteration algorithm using a finite set of sampled belief points

that properly cover the belief simplex [12]. We could also go one step further and instead of sampling the belief points that cover the entire simplex, sample those that are more probable to be reached from the current belief state [1]. This will enable us to approximate the value function more accurately for the current belief state through sampling more belief points in the more plausible area of the belief simplex.

3.3 Expectation Maximization for Abstract Policies

The value iteration algorithm requires full knowledge of the environment's (POMDP's) probability functions, including state transition probability, observation probability and initial state probability functions. However, this knowledge is not always given in all the problems. The second method suggested in this chapter for learning abstract POMDP polices is an Expectation Maximization algorithm based on the EM learning algorithm explained for regular policies in the previous chapter.

This method finds the optimal policy by using sample trajectories an agent takes while observing the world. As a result, this algorithm could improve the policy or learn in a realtime fashion. The EM algorithm is first modified to learn using a sample set of trajectories, then the structure of the Bayesian networks is modified to use abstract policies. Using this sampling method will make the learning algorithm much faster, because first the computational complexity posed by inference operations on the Bayesian networks will be eliminated and second the complexity of the algorithm becomes independent of the number of states in the model. It is very important because for abstraction we need to use complex problems with a large number of world states and exact inference methods won't be able to solve those models in a reasonable amount of time. The other important benefit of using sampling is that the learning algorithm will no longer require the probability functions of the POMDP

model. This makes our realtime learning algorithm work for the model-less POMDP problems as well.

The first part of this section explains how the sampling method works and the second part explains how to modify the BN structure to incorporate abstract actions.

3.3.1 Realtime Learning for Model-less POMDPs using Sampling

In this section, we propose a new computationally fast and simple learning algorithm for POMDP problems that is based on EM and uses a set of sampled trajectories (memory states, actions, observations and rewards) to optimize the policy. The complexity of the algorithm is independent of the number of world states and could be used to learn larger POMDP problems compared to exact inference methods. This method also does not require knowledge of the POMDP's probability distributions. Since computation cost is low, many iterations of the EM algorithm could be applied in a fixed amount of time compared to exact inference methods and this dramatically improves the convergence.

Let us define $\rho =< m_0, z_0, a_0, r_0, m_1, z_1, a_1, r_1, ... >$ to be a trajectory of memory states, observations, actions and rewards that are all observable by the agent. Also, let ρ_T be the first T steps of this trajectory. We are trying to calculate the inference using a sampled set of these trajectories $\mathcal{P}$.

Now, we will show how equations 2.15, 2.16 and 2.17 could be modified to calculate the expected values in the E-step from a sampled set of trajectories $\mathcal{P}$. To do this, we first modify these equations to calculate the probabilities from the set of

all possible trajectories, then we show how to estimate the expected values using a set of sampled trajectories. For convenience, these equations are repeated here again:

$$E[\pi_a^m] = \sum_{T=0}^{\infty}\sum_{t=0}^{T}\gamma^T P(m_t = m, a_t = a|\hat{r} = 1, T, \theta^{(n)}), \tag{3.21}$$

$$E[\mu_{m'}^{mz}] = \sum_{T=0}^{\infty}\sum_{t=0}^{T-1}\gamma^T P(m_t = m, z_{t+1} = z, m_{t+1} = m'|\hat{r} = 1, T, \theta^{(n)}), \tag{3.22}$$

$$E[\nu_m] = \sum_{T=0}^{\infty}\gamma^T P(m_0 = m|\hat{r} = 1, T, \theta^{(n)}). \tag{3.23}$$

We have to modify all of these three equations, but let us start by modifying the Equation 3.21. Also, let us define the variable δ to be:

$$\delta_{ma}^t(\rho) = P(m_t = m, a_t = a|\hat{r} = 1, T, \theta^{(n)}, \rho). \tag{3.24}$$

This is the same probability term as the one inside Equation 3.21, but conditioned on a trajectory ρ being experienced. Please notice that when the trajectory ρ is given, the probability of $m_t = m$ and $a_t = a$ is independent of the other conditions. As a result, $\delta_{ma}^t(\rho) = P(m_t = m, a_t = a|\rho)$ is a binary variable and is equal to:

$$\delta_{ma}^t(\rho) = \begin{cases} 1, \text{ if } m_t = m \text{ and } a_t = a \text{ in } \rho; \text{ and} \\ \\ 0, \text{ otherwise.} \end{cases} \tag{3.25}$$

Using δ, Equation 3.21 could be redefined this way:

$$E[\pi_a^m] = \sum_{T=0}^{\infty}\sum_{t=0}^{T}\gamma^T \sum_{\forall\rho} P(\rho_T|\hat{r} = 1, T, \theta^{(n)})\delta_{ma}^t(\rho). \tag{3.26}$$

Using Bayes law we have:

$$P(\rho|\hat{r} = 1, T, \theta^{(n)}) = \frac{P(\rho|T, \theta^{(n)})P(\hat{r} = 1|\rho, T, \theta^{(n)})}{\sum_{\rho'} P(\rho'|T, \theta^{(n)})P(\hat{r} = 1|\rho', T, \theta^{(n)})}. \tag{3.27}$$

Again, given the trajectory, the probability of observing a reward is independent of the policy. Therefore, $r_\rho^T = P(\hat{r} = 1|\rho, T, \theta^{(n)}) = P(\hat{r} = 1|T, \rho)$ is the reward observed

at step T of the trajectory ρ. Replacing this in Equation 3.27 and plugging it into Equation 3.26 we get:

$$E[\pi_a^m] = \sum_{T=0}^{\infty} \sum_{t=0}^{T} \gamma^T \frac{\sum_\rho P(\rho_T|T, \theta^{(n)}) r_\rho^T \delta_{ma}^t(\rho_T)}{\sum_{\rho'} P(\rho'_T|T, \theta^{(n)}) r_{\rho'}^T}. \tag{3.28}$$

Equation 3.28 tells us how to calculate $E[\pi_a^m]$ from the set of all possible trajectories. What we need instead is how to calculate this expected value from a set of sampled trajectories. We know that if $\mathcal{S}(A)$ is a set of samples drawn from the original set A using a probability distribution P, then:

$$\sum_{a \in A} P(a) f_a \approx \frac{1}{|\mathcal{S}(A)|} \sum_{s \in \mathcal{S}(A)} f_s. \tag{3.29}$$

Using this fact and applying it to Equation 3.28 we could calculate $E[\pi_a^m]$ from a set of sampled trajectories $\rho \in \mathcal{P}$:

$$\begin{aligned} E[\pi_a^m|\mathcal{P}] &\approx \sum_{T=0}^{\infty} \sum_{t=0}^{T} \gamma^T \frac{\sum_{\rho \in \mathcal{P}} r_\rho^T \delta_{ma}^t(\rho)}{\sum_{\rho' \in \mathcal{P}} r_{\rho'}^T} \\ &= \sum_{T=0}^{\infty} \gamma^T \frac{\sum_{\rho \in \mathcal{P}} r_\rho^T \sum_{t=0}^{T} \delta_{ma}^t(\rho)}{\sum_{\rho' \in \mathcal{P}} r_{\rho'}^T} \\ &\approx \sum_{T=0}^{T_{max}} \gamma^T \frac{\sum_{\rho \in \mathcal{P}} r_\rho^T \chi_{ma}^T(\rho)}{\sum_{\rho' \in \mathcal{P}} r_{\rho'}^T}. \end{aligned} \tag{3.30}$$

wherein, $\chi_{ma}^T(\rho) = \sum_{t=0}^{T} \delta_{ma}^t(\rho)$ adds up the $\delta_{ma}^t(\rho)$ values for $t = [0, ..., T]$. In other words, $\chi_{ma}^T(\rho)$ counts how many times the pair of memory state m and action a has occurred in the sub-trajectory ρ_T. We have also truncated the infinite loop of T at some T_{max} value the same way as in [3], because γ^T becomes exponentially smaller as T is getting larger, so the effect of those terms is almost negligible at some point.

Please notice how computationally less expensive Equation 3.30 is compared to the Equation 3.21 computed using exact inference algorithms (e.g. Forward-Backward or Junction Networks) on a huge Bayesian network. The complexity order of Equation 3.30 is T_{max} squared times the number of sampled trajectories: $O(T_{max}^2|\mathcal{P}|)$.

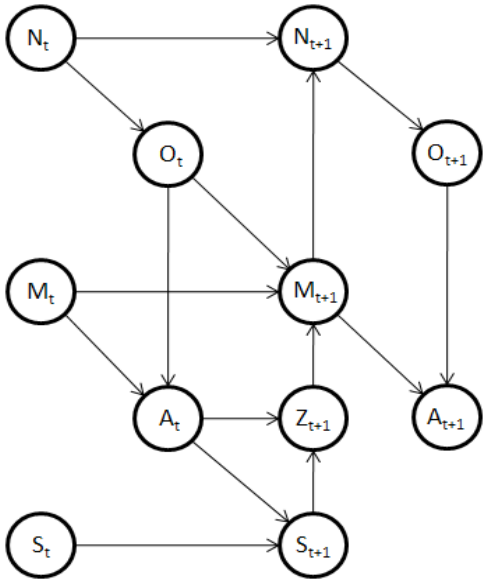

Figure 3.1 A single slice of the Bayesian network for an abstract policy with options used for calculating the reward probability. The original Bayesian network was illustrated in Figure 2.2.

Using the same analysis, we can convert the two other E-step equations (3.22 and 3.23) to be estimated from our set of sampled trajectories $\mathcal{P}$ to:

$$E[\mu_{m'}^{mz}|\mathcal{P}] \approx \sum_{T=0}^{T_{max}-1} \frac{\gamma^T \sum_{\rho \in P} r_\rho^T \phi_{mzm'}^T(\rho)}{\sum_{\rho' \in T} r_{\rho'}^T}, \tag{3.31}$$

$$E[\nu_m|\mathcal{P}] \approx \sum_{T=0}^{T_{max}} \frac{\gamma^T \sum_{\rho \in P} r_\rho^T \xi_m(\rho)}{\sum_{\rho' \in T} r_{\rho'}^T}. \tag{3.32}$$

Here, $\phi_{mzm'}^T(\rho)$ counts how many times the transition from memory state m and observation z to the memory state m' has occurred in the sub-trajectory ρ_T. $\xi_m(\rho)$ returns one if trajectory ρ starts with the memory state m and 0 otherwise.

3.3.2 Hierarchical Policy

In order to apply hierarchical policies in the EM learning algorithm, we have to add other layers of memory state variables for the high-level actions to the Bayesian

network first depicted in the Figure 2.2 [15]. The new Bayesian network after adding high-level actions is going to look like the one in Figure 3.1.

Figure 3.1 shows one slice of the actual BN at time-steps t and $t+1$. The random variable n_t defines the value of the memory state for the high-level policy's FSC. The variable o_t is the option being executed at time-step t and is selected according to the memory state of the high-level policy (n_t). The option itself is a FSC too and has an internal memory state which is defined by m_t. At each time-step, the low-level action is selected based on what option is executing and according to its current memory state and action probability function. The high-level memory state transits to another state when an option is terminated conditioned on its current state and the last memory state of the terminated option.

When running EM on this new BN, the option's probability functions (initial memory state, memory state transition, action probability) are fixed and not going to be learned. Therefore, the CPDs defining these functions are not included in the set of parameters of the Maximization step θ. This will let the order of the EM algorithm remain the same both in the exact and the approximate methods as in the original algorithm.

CHAPTER 4

EXPERIMENTAL RESULTS

This chapter explains the methods used and the results of evaluating the two proposed learning algorithms for abstract policies in POMDPs. In the first section, a new sample problem is proposed for evaluation. The next two sections explain the implementation and the experimental results for each of the proposed algorithms separately.

4.1 The Ambulance Problem

The Robocup Rescue Simulation [5] is one of the well-known AI test-beds that provides a comprehensive environment for research on challenging problems such as decision making under uncertainty, multi-agent planning, realtime learning, etc. The scalability potential of this simulation environment is completely unique compared to most of the existing official AI problems. One could very easily create different subsets of this problem from a very simple single-agent environment with a few states to a very complex multi-agent environment with infinite number of states.

In our research we also needed a problem that could easily be scaled from a simple case used for benchmarking the algorithms in absence of abstraction to more complex cases that could better show the advantage of the abstraction methods. Most of the existing problems however are designed to be simple and solvable using the existing algorithms. We therefore created a simplified version of the Robocup Rescue Simulation problem which is scalable from a very simple version to as complex as the actual problem for the future algorithms.

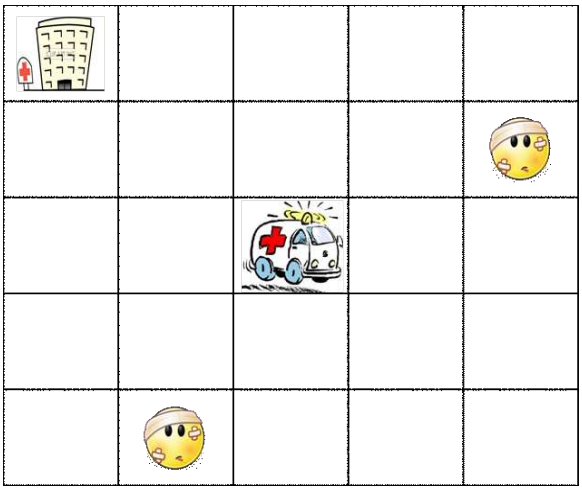

Figure 4.1 A sample Ambulance problem with two civilians.

In our *Ambulance problem*, there exists a rectangular grid world on which there are an ambulance agent, a set of civilians and one of the grid cells defined as the refuge. The civilians have experienced a disaster and need to be carried to the refuge for protection and cure. The ambulance agent should find the civilians moving randomly around, load them, carry them to the refuge one by one and unload them there. The ambulance doesn't know its own location and the civilians'. It could see the civilians though if they are in the same cell and could hear their voice probabilistically based on their distance. The ambulance also observes when it hits one of the walls which are considered to be the edges of the grid-world. The agent has actions to move one step *Up*, *Down*, *Right* or *Left* on the grid world and also can *Load* or *Un-load* a civilian. A civilian can be loaded if it is in the same cell as the ambulance, but can be un-loaded everywhere. There is a reward of +1000 to un-load a civilian at the refuge cell and a -1 for all other actions. A civilian can not be loaded from the refuge and is not going to escape from there.

We have defined two very simple options for this environment to run our experiments. The *Search* option finds a cell wherein there is a civilian and then terminates.

The *Carry* option moves until it reaches the refuge cell and terminates. It is very obvious that the optimal policy is to repeat the sequence of Search, Load, Carry and Unload actions. However, it's not that simple for the learning algorithm to figure that out because it needs to evaluate all different sequences of actions including primitive actions and options and also within that all different possible trajectories from executing each option. In practice, we modeled the options with FSCs and used only three memory states for each one.

4.2 Value Iteration

4.2.1 Implementation

Using the value function we derived in the previous section, it is going to be easy to adopt any of the existing POMDP value iteration algorithms to support learning with the high-level actions. All different variations of the POMDP value iteration algorithms are using different methods to address the problem of increasing complexity due to fast growing number of the Alpha-vectors. We do not address this problem here though. We have based our implementation on the point-base value iteration methods and we have only updated the *Backup Function* as explained in Algorithm 1. Please refer to the HSVI paper [1] or any other point-based value iteration algorithm for more details on the implementation.

Algorithm 1 $\rho = \text{backup}(\Gamma, b)$

$\beta^{o,*} \leftarrow arg\max_{\beta^o}(\beta^o.b)$

$\rho^o_s \leftarrow \beta^{o,*}_s + \max_{\alpha\in\Gamma}(\sum_{s'} \alpha_{s'} \hat{\Psi}^o_{ss'})$

$\rho \leftarrow arg\max_{\rho^o}(\rho^o.b)$

4.2.2 Evaluation

We used the Ambulance problem to run experiments and evaluate the learning algorithm. We are obviously expecting to see faster learning speed after using high-level actions. That is the whole purpose of abstraction.

Two different scales of the Ambulance problem are used for experiments. In the first case, there is a grid world of size 3x3, one civilian and 162 states. The second one has a grid world of size 4x4 with one civilian and 512 states. The parameters defining the state are the ambulance location, the civilians' location and the load status. There is one *Found-civilian* and four *Hit-wall* (for different sides) observations. We did not use the *Heared-civilian* observation in our experiments. There are also 6 primitive actions: *Up*, *Down*, *Right*, *Left*, *Load* and *Un-load*. The reward to rescue a civilian is 1000 and the discount factor is 0.95.

Figure 4.2.2 depicts the computation times of the algorithm solving the two problems, each in three different settings. In the first case (circles), all of the primitive actions are used and there are no high-level actions available. In fact, this is the normal POMDP value iteration algorithm. In the second case (rectangles), the Search and Carry options are added to the list of the available actions. All of the primitive actions are also preserved. This will increase the number of actions to 8. In the last setting (triangles), we add the two options and removed the four primitive move actions (Up, Down, Left and Right). The picture on the top shows the results for the Ambulance-3x3 problem while the other picture is the results from the Ambulance-4x4 problem.

The results show that adding options improves the learning speed even though it makes the model more complex by increasing the number of actions. It also shows that as the number of states are increasing the effectiveness of the high-level actions in speeding up the learning algorithm also increases. In the Ambulance-3x3 problem,

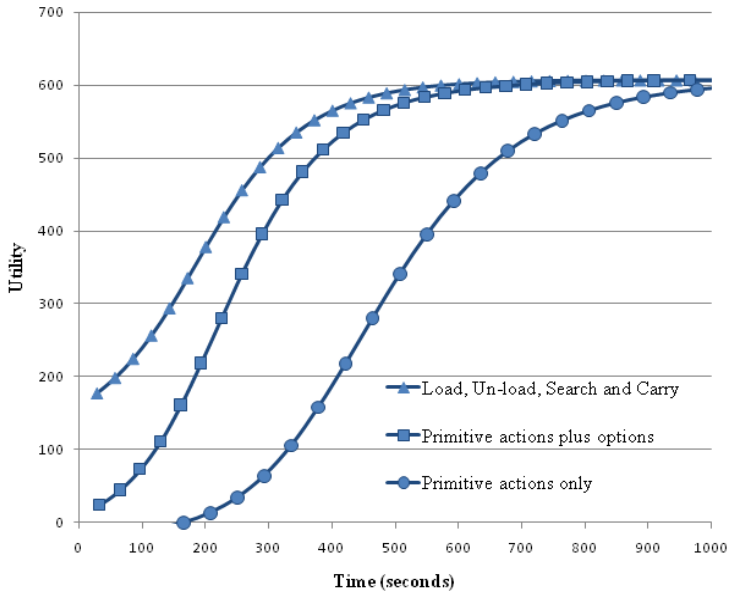

Figure 4.2 The computation times for learning an Ambulance-3x3 problem.

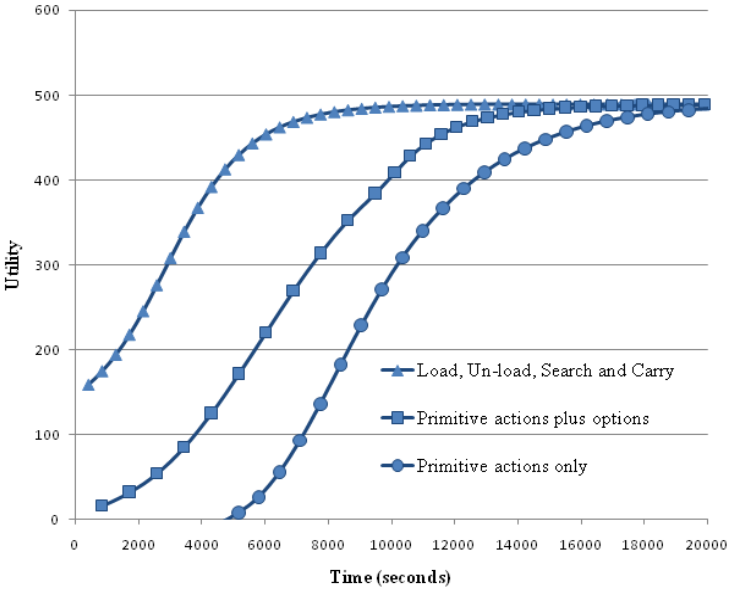

Figure 4.3 The computation times for learning an Ambulance-4x4 problem.

the utility of the optimal policy reaches 90% of the maximum value using options two times faster than the case of primitive actions only. This is almost three times faster in the Ambulance-4x4 problem which has almost three times more states. The reason is that options propagate more information about the future rewards in each iteration of the algorithm and will let the value function be updated faster [10].

It is obvious form the experiments that after new high-level skills are added, the learning speed could be further increased if less useful primitive actions are removed from the model. We do not address the problem of which actions should be removed in this thesis; however, this is studied for MDP models already [16].

We also didn't explain in this thesis how options should be extracted from the previous experiments automatically. One way to achieve this is to find the useful policies that might not directly perform a complete task but are repeatedly used as a subset of the other policies [17].

We have used the Cassandra POMDP learning tools [18] to parse and load the POMDP files, for benchmarking with existing value iteration algorithms and also as a base to add our implementation on it. All the experiments were run on a PC with an Intel Pentium 4 CPU (3.40GHz) running a Redhat Enterprise 5 GNU/Linux operating system.

4.3 Expectation Maximization

4.3.1 Implementation

In this section we explain how to implement the sampling based EM algorithm, which in fact is very simple and straight forward compared to most of the other POMDP learning algorithms we discussed in this thesis. There are two separate parts involved in the learning process. The first part runs experiments and collects

Algorithm 2 Sample_Trajectory()

$m_0 \leftarrow$ Draw_Sample($\nu_M^{(n)}$)

$\xi_{m_0} \leftarrow \xi_{m_0} + 1$

for $t = 0 : T_{max}$ **do**

$a_t \leftarrow$ Draw_Sample($\pi_{m_t A}^{(n)}$)

Take_Action(a_t)

$r_t \leftarrow$ Receive_Reward()

$z_t \leftarrow$ Receive_Observation()

$m_{t+1} \leftarrow$ Draw_Sample($\mu_{m_t z_t M}^{(n)}$))

$\chi_{m_t a_t}^t \leftarrow \chi_{m_t a_t}^{t-1} + 1$

$\phi_{m_t z_t m_{t+1}}^t \leftarrow \phi_{m_t z_t m_{t+1}}^{t-1} + 1$

$\omega^t \leftarrow \omega^t + r_t$

end for

statistics from the environment. The second part uses the EM algorithm to adjust and improve the policy over the course of time.

The *Sample_Trajectory* function in Algorithm 2 is responsible for running experiments using the latest policy in hand and also collecting statistics. Each call of this function performs one experiment of length T_{max}. The states of the POMDP are considered non-absorbing and, as explained earlier, the trajectory is truncated at a point were the reward's discount is too close to zero. The algorithm starts by picking an initial memory state for the FSC. To do that, we pick a random memory state based on the $\nu^{(n)}$ probability distribution. The function *Draw_Sample* generates a random index from the discretized probability distribution vector given as an argument. In each following time-step, the algorithm first selects an action according to the $\pi^{(n)}$ distribution and given the current memory state. After taking the action,

Algorithm 3 EM_Iterate()

for N times **do**

Sample_Trajectory()

end for

$\pi^{(n+1)} \leftarrow \vec{0}$

$\mu^{(n+1)} \leftarrow \vec{0}$

$\nu^{(n+1)} \leftarrow \vec{0}$

for $T = 0 : T_{max}$ **do**

$\pi_{ma}^{(n+1)} \leftarrow \pi_{ma}^{(n+1)} + \gamma^T \chi_{ma}^T / \omega^T \qquad \forall ma$

$\mu_{mzm'}^{(n+1)} \leftarrow \mu_{mzm'}^{(n+1)} + \gamma^T \phi_{mzm'}^T / \omega^T \qquad \forall mzm'$

$\nu_m^{(n+1)} \leftarrow \nu_m^{(n+1)} + \gamma^T \xi_m / \omega^T \qquad \forall m$

end for

Normalize($\pi_{mA}^{(n+1)}$) $\quad \forall m$

Normalize($\mu_{mzM}^{(n+1)}$) $\quad \forall m, z$

Normalize($\nu_M^{(n+1)}$)

it receives observation and reward from the environment. Then, it updates the FSC memory state according to the $\mu^{(n)}$ distribution given the current memory state and the latest observation. It finally updates the χ, ψ and ω statistics and goes to the next time-step. The φ variable is updated only once at time-step zero, since it only counts the initial memory state.

The *EM_Iterate* function in Algorithm 3 performs the task of updating the policy in one iteration of the EM algorithm and using the statistics provided by sampling. This algorithm first calls the *Sample_Trajectory* function to run some experiments and collect statistics. Then it updates the FSC's different probability distributions (π, μ and ν) using Equations 3.30, 3.31 and 3.32. To complete the EM

iteration the algorithm normalizes the probability functions as explained in Equations 2.18, 2.19 and 2.20 as well. The *Normalize* function takes a vector as an input argument and divides the elements by their sum to let those values add up to 1.

As you have probably already noticed, it seems such a waste to run experiments and collect statistics for each iteration and toss all this information out in the next step. Fortunately, the Importance Sampling techniques help avoiding that. In [14], Shelton explains how trajectories produced from other policies could be used to evaluate the current policy using Importance Sampling. The exact same concepts could be applied here, too. When using Importance Sampling, a selection of the past trajectories could be saved in the memory and reused for adjusting the policy in the current iteration. Each old sample will receive an importance weight which is equal to the probability of it happening under the current policy divided by its probability under the original policy it was generated from. Then, the weighted samples are included in the E-step evaluations while their expected rewards are multiplied by their importance weights. To calculate the importance weights, there is fortunately no need to know the world model's probability distributions which is consistent with our goal that these distributions are considered unknown in our learning algorithm.

4.3.2 Evaluation

Unfortunately, the result of experiments with the EM algorithm did not fulfill the expectations in our tries. The sampling algorithm is much faster than the exact EM method on very simple problems like the Tiger problem; however, on most of the larger size problems the sampling method gets stuck in some local optimum point in the value function. We tried several different problems including the ambulance problem from different start points, but the algorithm gets stock in a local optimum point in most of the cases.

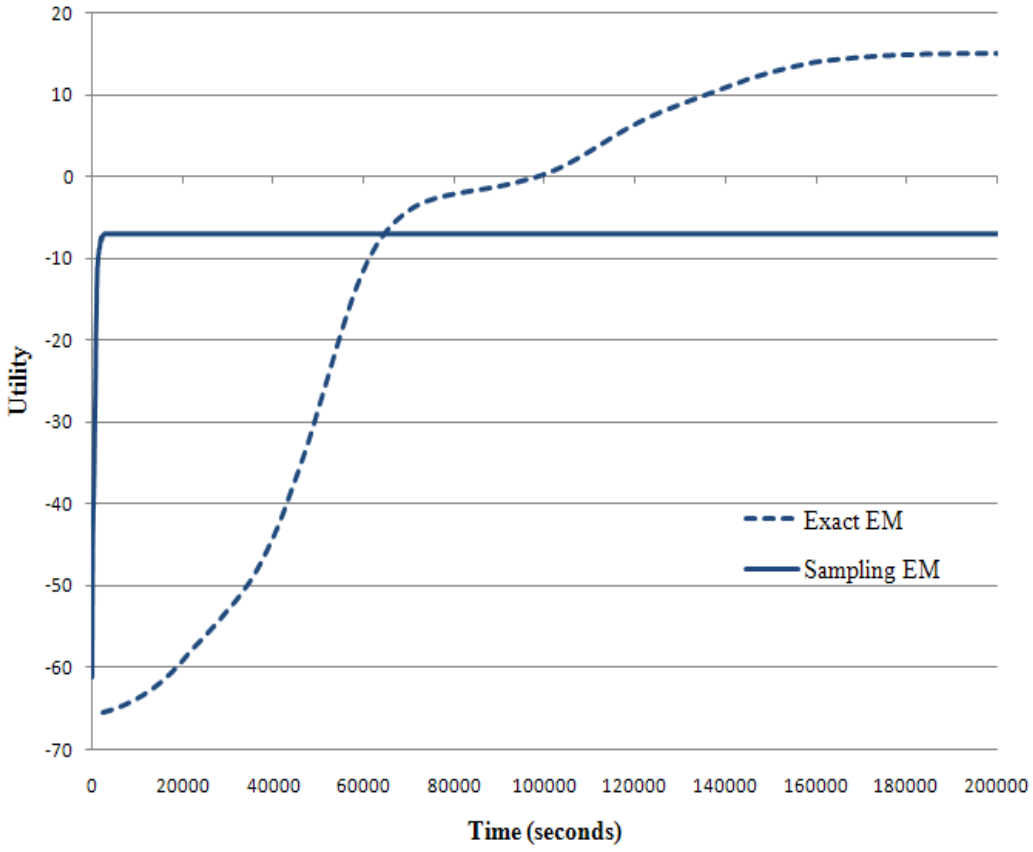

Figure 4.4 Optimal policy's utility vs. computation time on a problem with 100 states. The dashed curve shows the performance of the EM method with exact inference and the solid line is for the EM method with sampling .

Figure 4.4 shows one run of both the exact EM and the sampling EM algorithm on a POMDP problem with 100 states. The exact algorithm (dashed curve) is very slow and takes several days to learn this problem. The sampling method (solid line) is very fast but usually gets stock in some local optimum point.

We have added some randomness in the sample generator function in order to avoid biased samples. This has helped to some extent and has delayed faulty convergences but the problem is not resolved.

Simulated annealing was also implemented to let the algorithm jump out of the local optimum points. In most of the cases this helped the algorithm to jump out of a few first local optima points but usually gets stuck jumping around when the result is closer to the global optimum.

The EM algorithm is in general prone to local optima; however, we are not sure if the sampling method has magnified this weakness or if there are other problems we have neglected. The original algorithm has also gotten stuck in a local optima in some trials, but the occurrence of this problem has been much more frequent in the sampling method.

CHAPTER 5

CONCLUSION

In this thesis, we proposed two learning algorithms based on value iteration and expectation maximization that enable the use of high-level actions or options for POMDP models. Using options makes it possible to transfer the knowledge gained from the past experiences to solve other problems.

We show that the optimal value function remains piece-wise linear and convex after options are added and it is possible to benefit from the existing value iteration algorithms with some modifications. An expectation maximization based algorithm is also modified to learn the optimal policy from a set of sampled trajectories which is a much faster algorithm compared to the original version that uses exact inference. A change of architecture is suggested for the EM algorithm to support learning policies using high-level actions. We have also introduced a new problem for benchmarking abstraction on POMDPs that we called the ambulance problem.

Our experiments show adding options makes the value iteration algorithm faster and more effective when the number of states is increased. The EM algorithm becomes more likely to get stuck in local optimum points when the sampling method is implemented. The presented value iteration algorithm relies on having the model's distribution functions (i.e. state transition probability, observation probability, etc.) which is a similar requirement for all of other value iteration algorithms, too. On the other hand, the sampling based EM algorithm does not require knowledge of the probability functions and in fact could perform realtime learning as the agent is exploring the world.

In this work we are not addressing how sub-goals could be defined for learning options and how to reduce the complexity by removing less useful primitive actions from the model.

REFERENCES

[1] T. Smith and R. Simmons, "Point-based pomdp algorithms: Improved analysis and implementation," *arXiv preprint arXiv:1207.1412*, 2012.

[2] R. Sutton, H. Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvári, and E. Wiewiora, "Fast gradient-descent methods for temporal-difference learning with linear function approximation," in *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, 2009, pp. 993–1000.

[3] M. Toussaint, S. Harmeling, and A. Storkey, "Probabilistic inference for solving (po) mdps," 2006.

[4] M. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey," *The Journal of Machine Learning Research*, vol. 10, pp. 1633–1685, 2009.

[5] S. Amraii, B. Behsaz, M. Izadi, H. Janzadeh, F. Molazem, A. Rahimi, M. Ghinani, and H. Vosoughpour, "Sos 2004: An attempt towards a multi-agent rescue team," in *Proceedings of the 8th RoboCup International Symposium*, 2004.

[6] R. Bellman, "A markovian decision process," DTIC Document, Tech. Rep., 1957.

[7] R. Sutton and A. Barto, *Reinforcement learning: An introduction*. Cambridge Univ Press, 1998, vol. 1, no. 1.

[8] R. Smallwood and E. Sondik, "The optimal control of partially observable markov processes over a finite horizon," *Operations Research*, vol. 21, no. 5, pp. 1071–1088, 1973.

[9] E. Hansen, "Finite-memory control of partially observable systems," 1998.

[10] D. Precup, R. Sutton, and S. Singh, "Theoretical results on reinforcement learning with temporally abstract options," *Machine Learning: ECML-98*, pp. 382–393, 1998.

[11] R. Sutton, D. Precup, S. Singh, *et al.*, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1, pp. 181–211, 1999.

[12] J. Pineau, G. Gordon, S. Thrun, *et al.*, "Point-based value iteration: An anytime algorithm for pomdps," in *International joint conference on artificial intelligence*, vol. 18. LAWRENCE ERLBAUM ASSOCIATES LTD, 2003, pp. 1025–1032.

[13] Z. Ghahramani, "Learning dynamic bayesian networks," *Adaptive Processing of Sequences and Data Structures*, pp. 168–197, 1998.

[14] C. Shelton, "Policy improvement for pomdps using normalized importance sampling," in *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 2001, pp. 496–503.

[15] M. Toussaint, L. Charlin, and P. Poupart, "Hierarchical pomdp controller optimization by likelihood maximization," *Uncertainty in AI (UAI)*, 2008.

[16] M. Asadi, "Learning state and action space hierarchies for reinforcement learning using action-dependent partitioning," Ph.D. dissertation, University of Texas at Arlington, 2006.

[17] M. Asadi and M. Huber, "Effective control knowledge transfer through learning skill and representation hierarchies," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 2007, pp. 2054–2059.

[18] A. Cassandra, "Tonys pomdp page," *website http://www. cs. brown. edu/research/ai/pomdp*, 1999.

BIOGRAPHICAL STATEMENT

Hamed Janzadeh was born in Qazvin, Iran, in 1982. He received his B.S. and M.S. degrees from Amirkabir University of Technology (Tehran's Polytechnic), Tehran, Iran in 2005 and 2008, respectively, in Computer Engineering and Information Technology. He received his second M.S. degree in Computer Science from The University of Texas at Arlington in 2012. Hamed is working in the R&D department of the ClearCorrect, Inc. as a project manager since 2011.