# CS181 Lecture 4 — MDPs and POMDPs

Today we continue the discussion of Markov Decision Processes. We will complete the analysis of value iteration from last time, and use the ideas to develop another algorithm, policy iteration. We will then turn our attention to situations in which the agent does not know the state of the world, but only gets observations about the state (Partially Observable Markov Decision Processes or POMDPs). Finally, we discuss the situation where the agent does not know the model of the world, which is the topic of reinforcement learning.

## 1 Policy Iteration

As we have seen, for a stationary policy π, the values of states under π must satisfy the following equations.

$$V^{\pi}(i) = R(i, \pi(i)) + \gamma \sum_j T_{ij}^{\pi(i)} V^{\pi}(j) \tag{1}$$

Furthermore, an optimal stationary policy $\pi^*$ must satisfy the following optimality equations.

$$\pi^*(i) = \arg\max_a [R(i, a) + \gamma \sum_j T_{ij}^a V^{\pi^*}(j)] \tag{2}$$

$$V^{\pi^*}(i) = \max_a [R(i, a) + \gamma \sum_j T_{ij}^a V^{\pi^*}(j)] \tag{3}$$

As we discussed, Equation (3) can be viewed as defining a fixpoint equation on the space of value functions. The infinite horizon value iteration algorithm solves the fixpoint equation. It begins with any value function $V_0$. Applying the right hand side of (3) to $V_0$ yields a new value function $V_1$. Applying the right hand side again yields a new value function $V_2$, and so on.

The policy iteration algorithm is based on the observation that we can also view Equation (2) as defining a fixpoint equation on the space of stationary policies. Notice that the left hand side, $\pi_*$, appears on the right hand side. This is precisely the shape of a fixpoint equation. This observation suggests that we can use a similar procedure: we begin with any policy $\pi_0$, apply the right hand side of (2) to generate a new policy $\pi_1$, apply the right hand side again to get $\pi_2$, and so on.

The process is a little more complicated than for value iteration. In value iteration, applying the right hand side of (3) to any previous value function V is easy. For policy iteration, it is not quite as straightforward to apply the right hand side of (2) to a policy $\pi$. We need to get from the policy $\pi$ to its value function. But we know how to do that, because for a stationary policy, the value function $V^{\pi}$ satisfies Equation (1). So, given $\pi$, we solve (1), which is a system of linear equations, to obtain $V^{\pi}$. We then apply the right hand side of (2) to get a new policy. The algorithm is as follows:

```
PolicyIteration(γ) =
  // Takes the discount factor γ
  // Returns the optimal policy for each state
  Let π be any policy
  Repeat
    πold = π
    Solve the system of equations
      $V(i) = R(i, \pi_{old}(i)) + \gamma \sum_j T_{ij}^{\pi old(i)} V(j)$
      to obtain V (i)
    For each state i
      For each action a
        $Q(i, a) = R(i, a) + \gamma \cdot \sum_j T^a_{ij} \cdot V(j)$
      $\pi new(i) = \arg\max_a Q(i, a)$
        // Assume ties are broken in a consistent way
    π = πnew
  Until πnew = πold
  Return π.
```

Policy iteration is a neat idea. Does it actually work? The homework asks you to prove that it does. The key steps in the proof are to prove the following two claims:

1. If policy iteration returns policy π, then π is an optimal policy.

2. The policies produced by policy iteration get better and better.

From these one can deduce that policy iteration always terminates with the optimal policy.

How do value iteration and policy iteration compare? It is a fact that policy iteration takes at most as many iterations to reach the optimal policy as value iteration does. In practice, it usually takes far fewer iterations. In policy iteration, the policy always changes every iteration. In contrast, in value iteration, the value function changes every iteration, but the optimal policy relative to that value function, i.e., the optimal policy given that that value function will be achieved in future, may stay the same for several successive iterations. On the other hand, each individual iteration of policy iteration takes longer, because it requires solving the complete system of linear equations of Equation (1). In general, however, policy iteration is considered the better algorithm in practice.

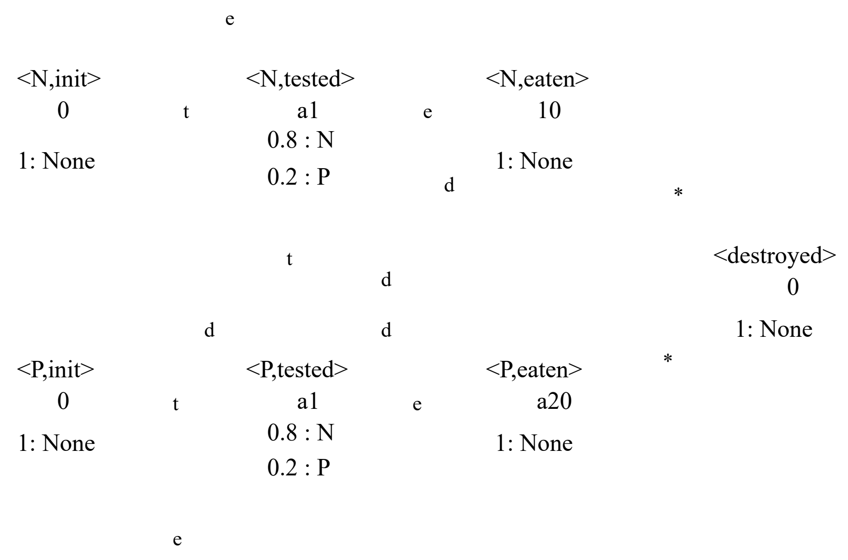
One way to make policy iteration more efficient is to dispense with solving Equation (1) exactly, and to be satisfied with an approximate solution. Note that Equation (1) is also a fixpoint equation, and can be solved by the same kind of iterative method we have been using for solving the optimality equations. I.e., we start with an arbitrary value function, and apply the right hand side of (1) to get better and better approximations to the $V^{\pi}$. This method of approximating the value function is called value propagation, and can be used instead of solving the equations in the policy iteration algorithm.

## 2 Partial Observability

Let us now drop the assumption that the agent always knows the state of the environment when it has to make a decision. This is much more realistic for most situations that agents encounter. The agent is not normally told exactly what the state of the world is. Rather, it gets observations about the state of the world through its sensors, that allow it to form probabilistic beliefs about the world state.

We modify the MDP setup to introduce observations. The new framework is called a partially observable Markov decision process, or POMDP as most people call it. There is a set O of possible observations, and an observation model or sensor model that tells us what the agent is likely to observe in each state of the world: $P(o \mid i)$ is the probability of receiving observation o in state i. Also, we introduce an initial distribution $P_0$ into the model, to model the agent's initial beliefs. $P_0(i)$ is the probability that the initial state of the world is i. If there is a fixed, known, initial state i, this is modeled by setting $P_0(i) = 1$.

Consider an example for our plant-eating robot. The robot is confronted by a particular plant, which may be either nutritious or poisonous. The robot may eat or destroy the plant, but in addition, it may conduct a series of tests to try to determine if it is nutritious. We can model this situation as a POMDP. The states determine the possible observations, so we must include enough information in the state space to specify the observation model. We will use two variables to describe the state space: the first, Boolean variable, indicates whether or not the plant is nutritious or poisonous. The second, three-valued variable indicates the state of the game: initial state, a state in which the agent has just tested the plant, and a state in which the plant has just been eaten. We also add one more terminal state to indicate that the game is over. There are three possible observations: None, which is received at all times except after a test, and Nutritious and Poisonous, which are received after a test. The test is fallible: it has an 80% chance of providing the correct information. However, the test can be repeated, and multiple applications of the same test are independent of each other. The transition and reward models are very simple — in fact, the transition model is deterministic. We can describe the whole problem as a finite state machine, as shown in the following figure.



t

fsm1 node stands for a state, while edges are labeled by actions (“e”,”d” and “t” for eat, destroy and test). An edge from i to j labeled by action a means that taking action a in state i causes the world to transition to state j. Each node contains three pieces of information: the name of the state (e.g. < N, init > for the initial state where the plant is nutritious); the reward associated with the state; and the observation model for the state, specified as a probability distribution over observations. Also not shown is $P_0$, the agent’s initial distribution over the state of the world. If the agent decides, prior to performing any tests, that the probability of the plant being nutritious is p, then $P_0$ will assign probability p to state < N, init > and 1 − p to state < P, init >.[1]

Why are POMDPs hard? There are two main reasons. The first reason is that the optimal action at any given point in time depends not only on the agent’s current observation but on the history of observations and actions it has taken. In our example, suppose the agent has just tested, and wants to decide what to do next. The answer depends not only on the current test result, but on the whole history of test results. This situation is in contrast to MDPs. Because the current state fully determines the expected current and future utility for each policy, and in an MDP the agent knows the current state, the agent can ignore history in choosing its policy. However, in POMDPs, where the agent does not know the current state, the history is relevant to determining what the current state is likely to be. Therefore, the agent may need to compute a separate policy for every possible history, which is prohibitively expensive.

The second reason POMDPs are hard is that in making its decisions, an agent needs to take into account not only the effects of its actions on the environment, but also on its own future beliefs. It is possible that the agent needs to

[1]This kind of drawing can also be used to illustrate MDPs and POMDPs with probabilistic transition models, by labeling the edges with probabilities as well as actions.

perform an action that is not optimal in any state, just to gather the information it needs to make decisions in the future. Such actions are called information-gathering actions. In our example, the test action is an information-gathering action. It is not optimal in any state. If the plant is nutritious, the optimal action is to eat it. If the plant is poisonous, the agent should destroy it. Basically, if the agent knows what the state is, it should take the action appropriate to that state, rather than wasting energy on tests. However, if the agent is sufficiently uncertain about what the state is, it should perform the information-gathering action in order to find out what it is.

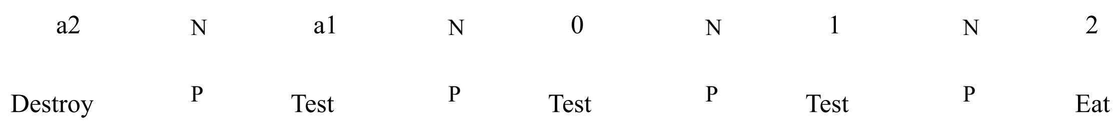
Because of these issues, solving POMDPs to obtain optimal policies is extremely hard. In the finite horizon case, the cost is exponential in the horizon.

In the infinite horizon case, the problem of whether or not there exists a policy that achieves a certain value is undecidable! [Madani, Hanks & Condon, AAAI99]

Nevertheless, many actual real-world situations are best characterized as POMDPs. Since finding an optimal policy is too hard for all but the most trivial of domains, people are generally satisfied if they can implement a good policy, without requiring it to be optimal. There are several particular kinds of policies that are often implemented:

Finite memory policy An agent with a finite memory can only remember the last k observations. So a finite memory policy is a function from the previous k percepts to an action. A typical policy for our example is to continue testing until two identical observations are obtained in succession, at which point the agent should eat if the observations were Nutritious and destroy if they were Poisonous.

Finite state controller Just as the world is modeled as a finite state machine, the agent can be implemented as a finite state controller. The nodes of the controller do not correspond to states of the world; rather, they correspond to internal states of the agent, depending on the observations it has received. There is a nice symmetry here. Just as the agent acts on the world via its actions, so the world acts on the agent by giving it observations. In the finite state machine describing an agent controller, nodes are labeled by the action to take in that node, while edges are labeled by observations. In the following finite state controller, the nodes are labeled with an integer representing the number of Nutritious observations minus the number of Poisonous observations. The agent starts in state 0. Note that this is not exactly the same policy as the finite memory policy described earlier. If the percept sequence is P,N,N, the finite memory policy will say to eat, but the finite state controller will still require a test.



Belief state policy At any point in time, the agent has a probability distribution over the current state of the world. This distribution is called its belief state, which is a very useful notion in thinking about POMDPs. The belief state allows us to formulate an agent's policy directly in terms of its beliefs. Instead of implementing a policy as a function from histories or internal states to actions, a policy is implemented by dividing the belief state into regions, and specifying an action in each region. A belief state policy in our example might be as follows:

| | |
|---|---|
| If $P(\text{Nutritious}) > 0.9$, | eat |
| If $0.2 < P(\text{Nutritious}) \leq 0.9$ | test |
| If $P(\text{Nutritious}) \leq 0.2$, | destroy |

An agent that implements a belief state policy needs to maintain its current belief state. At any point in time, the agent must know the current distribution over possible states, given its history of observations and actions taken. I.e., at time t + 1 it must compute $P(S_{t+1} \mid O_1,\ldots,O_{t+1})$.

How can an agent maintain its belief state in a reasonably efficient way? The key idea is to use the Markov property: namely that the future is independent of the past given the present. The Markov property implies that $S_{t+1}$ is conditionally independent of $O_1,\ldots,O_t$ given $S_t$. A second useful property is that the observation depends only on the current state, and not on previous history, so $O_{t+1}$ is conditionally independent of $O_1\ldots O_t$ given $S_{t+1}$. Combining these two facts, we can work as follows. We write $P_t$ to indicate the belief state at time t (consistent with the notation $P_0$ for the initial distribution over states).

$$
\begin{aligned}
P_{t+1}(j) &= P(S_{t+1} = j \mid O_1,\ldots,O_t,O_{t+1}) \\
&= \frac{P(S_{t+1}=j \mid O_1,\ldots,O_t)P(O_{t+1} \mid S_{t+1}=j,O_1,\ldots,O_t)}{P(O_{t+1} \mid O_1,\ldots,O_t)} \\
&\propto P(S_{t+1} = j \mid O_1,\ldots,O_t)P(O_{t+1} \mid S_{t+1} = j, O_1,\ldots,O_t) \\
&= P(S_{t+1} = j \mid O_1,\ldots,O_t)P(O_{t+1} \mid S_{t+1} = j) \\
&= \textstyle\sum_i P(S_t = i \mid O_1,\ldots,O_t)P(S_{t+1} = j \mid S_t = i, O_1,\ldots,O_t)P(O_{t+1} \mid S_{t+1} = j) \\
&= \textstyle\sum_i P(S_t = i \mid O_1,\ldots,O_t)P(S_{t+1} = j \mid S_t = i)P(O_{t+1} \mid S_{t+1} = j) \\
&= \textstyle\sum_i P_t(i)T^a_{ij}P(O_{t+1} \mid S_{t+1} = j)
\end{aligned}
$$

where a is the action taken at time t

The second line is an application of Bayes rule. In the third line, the notation $\propto$ means "is proportional to" — we drop the normalizing factor. The fourth line follows from the second independence assumption mentioned above. The fifth line is reasoning by cases. The sixth line follows from the Markov propery. To summarize:

$$P_{t+1}(j) \propto \sum_{i} P_t(i) T^a_{ij} P(O_{t+1} \mid S_{t+1} = j) \tag{4}$$

Look what we have: the belief state at time t + 1 can be computed in two steps. First we take the belief state at time t and propagate it through the transition model $T_a$. This produces a prior distribution over the state at time t + 1. We then condition using the sensor model to obtain the posterior distribution over the state at time t+ 1, which is the belief state at time t + 1.