Markov Decision Processes

Jesse Hoey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, CANADA, N2L3G1
jhoey@cs.uwaterloo.ca

1 Definition

A Markov Decision Process (MDP) is a probabilistic temporal model of an agent interacting with its environment. It consists of the following:

- a set of states, $\mathcal{S}$,
- a set of actions $\mathcal{A}$,
- a transition function $T(s, a, s')$,
- a reward function $R(s)$,
- a discount factor γ.

At each time, t, the agent is in some state $s_t \in \mathcal{S}$, and takes an action $a_t \in \mathcal{A}$. This action causes a transition to a new state $s_{t+1} \in \mathcal{S}$ at time $t + 1$. The transition function gives the probability distribution across the states at time $t + 1$, such that $T(s_t, a_t, s_{t+1}) = Pr(s_{t+1}|s_t, a_t)$. The reward function $R(s)$ specifies the reward for being in state s. Most MDP treatments consider the reward to be over $R(s, a, s')$, but here we will consider this slightly simpler case.

The state-action-reward space can be compactly represented in graphical form as a Dynamic Bayesian network (DBN), as shown in Figure 1. The states are nodes in the graph, which is usually represented using only two time slices (as shown). The full DBN would be obtained by unrolling the graph for as many time steps as you want. Technically, this is not really a DBN, since neither the reward function nor the actions are random variables, so we call it a Dynamic Decision Network, or DDN.

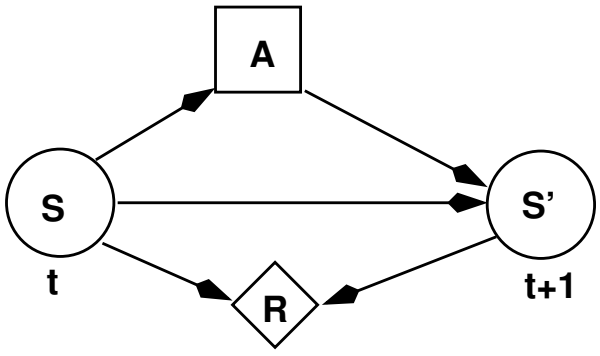


Figure 1: Decision network representation of an MDP. This should be unrolled in time to give the full network. Sometimes R depends on A as well. It is drawn here as $R(s, s')$, but we only treat it as $R(s)$.

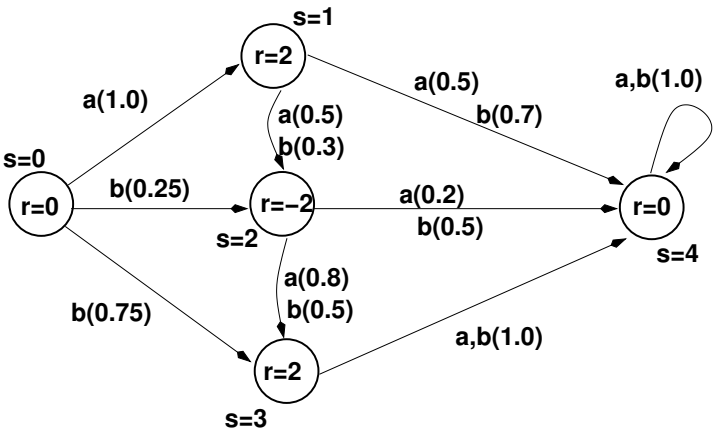


Figure 2: State space MDP graph. The nodes are labeled with the reward for that state.

2 State-Space Graph

Alternatively, the MDP can be represented extensively as a state-space graph, where each node represents a single state. For example, Figure 2 shows a state-space graph for a simple MDP example with 5 states ($S = \{0, 1, 2, 3, 4\}$) and 2 actions ($A = \{a, b\}$). Each arc in the graph denotes a possible transition, and is labeled with the action that causes it, and the probability of that transition happening given the labeled action is taken. **This is NOT a DBN**.

This same graph, represented as a decision network, would have the following factors:

$$P(S'|S, A = a) = \begin{bmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.5 & 0.0 & 0.5 \\ 0.0 & 0.0 & 0.0 & 0.8 & 0.2 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$P(S'|S, A = b) = \begin{bmatrix} 0.0 & 0.0 & 0.25 & 0.75 & 0.0 \\ 0.0 & 0.0 & 0.3 & 0.0 & 0.7 \\ 0.0 & 0.0 & 0.0 & 0.5 & 0.5 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$R(S) = \begin{bmatrix} 0 \\ 2 \\ -2 \\ 2 \\ 0 \end{bmatrix}$$

3 Policies and Values

The goal for an agent is to figure out what action to take in each of the states: this is its *policy* of action, $\pi(s) = a$. The optimal policy, π^*, is the one that guarantees that the system gets the maximum expected reward:

$$\sum_{t=0}^{\infty} \gamma^t R(s_t) \tag{1}$$

The value of being in a state s with t stages to go can be computed using dynamic programming, by evaluating all possible actions and all possible next states, s', and taking the action that leads to the best next state. The next states values are computed recursively using the same equation. Thus, starting with $V^0(s) = R(s)$, we can compute for $t > 0$:

$$V^t(s) = \max_a \left[R(s) + \gamma \sum_{s'} Pr(s'|s,a) V^{t-1}(s') \right] \tag{2}$$

The policy with t stages to go is simply the actions that maximize Equation 2:

$$\pi^t(s) = \arg\max_a \left[R(s) + \gamma \sum_{s'} Pr(s'|s,a) V^{t-1}(s') \right] \tag{3}$$

The optimal value function, V^* is the value function computed with ∞ stages to go, and satisfies Bellman's equation:

$$V^*(s) = \max_a \left[R(s) + \gamma \sum_{s'} Pr(s'|s,a) V^*(s') \right] \tag{4}$$

and the optimal policy is again simply the the actions that maximize Equation 4. In practice, V^* is found by iterating Equation 2 until some convergence measure is obtained: until the difference between V^t and V^{t-1} becomes smaller than some threshold.

4 Simple Derivation of the Value Iteration Equation 2

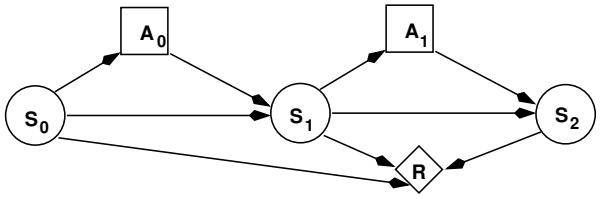


Figure 3: Decision network representation of an MDP for 2 time steps. Note here we have written the general case where R depends jointly on all s, but below we show an additive assumption that would make it a different R for each time step.

Using the variable elimination algorithm, we define factors

$$\begin{aligned} f_1(s_2, s_1, a_1) &= P(s_2|s_1, a_1) \\ f_2(s_0, s_1, s_2) &= R(s_0, s_1, s_2) \\ f_3(s_1, s_0, a_0) &= P(s_1|s_0, a_0) \\ f_4(s_0) &= P(s_0) \end{aligned}$$

We will make the assumption that the reward function is *additive* and *discounted*, such that

$$f_2(s_0, s_1, s_2) = R(s_0, s_1, s_2) = R^0(s_0) + \gamma R^1(s_1) + \gamma^2 R^2(s_2)$$

this essentially converts the single reward node in Figure 3 into three reward nodes, $R^0(s_0)$, $R^1(s_1)$ and $R^2(s_2)$.

First, we sum out variables that are not parents of a decision node (s_2 only):

$$\begin{aligned} f_5(s_0, s_1, a_1) &= \sum_{s_2} f_1(s_2, s_1, a_1) f_2(s_0, s_1, s_2) \\ &= \sum_{s_2} f_1(s_2, s_1, a_1) \left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) \right] \\ &= R(s_0) + \gamma R(s_1) + \left[\gamma^2 \sum_{s_2} f_1(s_2, s_1, a_1) R(s_2) \right] \end{aligned}$$

Now, we max out the decision node with no children (a_1):

$$\begin{aligned} f_6(s_0, s_1) &= \max_{a_1} f_5(s_0, s_1, a_1) \\ &= \max_{a_1} \left[R(s_0) + \gamma R(s_1) + \left[\gamma^2 \sum_{s_2} f_1(s_2, s_1, a_1) R(s_2) \right] \right] \\ &= R(s_0) + \gamma R(s_1) + \left[\max_{a_1} \gamma^2 \sum_{s_2} f_1(s_2, s_1, a_1) R(s_2) \right] \end{aligned}$$

Now, we can sum out s_1

$$\begin{aligned} f_7(s_0, a_0) &= \sum_{s_1} f_3(s_1, s_0, a_0) f_6(s_0, s_1) \\ &= \sum_{s_1} f_3(s_1, s_0, a_0) \left[R(s_0) + \gamma R(s_1) + \left[\max_{a_1} \gamma^2 \sum_{s_2} f_1(s_2, s_1, a_1) R(s_2) \right] \right] \\ &= R(s_0) \sum_{s_1} f_3(s_1, s_0, a_0) + \sum_{s_1} f_3(s_1, s_0, a_0) \left[\gamma R(s_1) + \max_{a_1} \gamma^2 \sum_{s_2} f_1(s_2, s_1, a_1) R(s_2) \right] \\ &= R(s_0) + \gamma \sum_{s_1} f_3(s_1, s_0, a_0) \left[R(s_1) + \max_{a_1} \gamma \sum_{s_2} f_1(s_2, s_1, a_1) R(s_2) \right] \end{aligned}$$

Where we used the fact that $\sum_{s_1} f_3(s_1, s_0, a_0) = \sum_{s_1} P(s_1|s_0, a_0) = 1.0$ In the last step, we simply factored out one γ.
Now max out a_0

$$\begin{aligned} f_8(s_0) &= \max_{a_0} f_7(s_0, a_0) \\ &= R(s_0) + \max_{a_0} \left[\gamma \sum_{s_1} P(s_1|s_0, a_0) \left[R(s_1) + \max_{a_1} \gamma \sum_{s_2} f_1(s_2, s_1, a_1) R(s_2) \right] \right] \end{aligned}$$

letting $V(s_2) = R(s_2)$ and putting back $f_1(s_2, s_1, a_1) = P(s_2|s_1, a_1)$, we define

$$V(s_1) = R(s_1) + \max_{a_1} \gamma \sum_{s_2} f_1(s_2, s_1, a_1) V(s_2)$$

and so we get

$$V(s_0) = f_8(s_0) = R(s_0) + \max_{a_0} \left[\gamma \sum_{s_1} P(s_1|s_0, a_0) V(s_1) \right]$$

so we can now see the recursion developing as

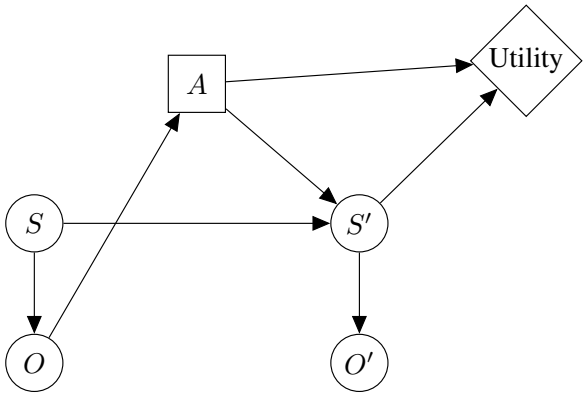
$$V(S_t) = R(s_t) + \max_{a_t} \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) V(s_{t+1})$$

5 Partially Observable Markov Decision Processes (POMDP)

A POMDP is like an MDP, but some variables are not observed. It is a tuple $\langle S, A, T, R, O, \Omega \rangle$ where

- S: finite set of unobservable states
- A: finite set of agent actions
- $T : S \times A \to S$ transition function
- $R : S \times A \to \mathcal{R}$ reward function
- O: set of observations
- $\Omega : S \times A \to O$ observation function

The decision network is shown as:



Where, without loss of generality, we have made the utility function depend only on S' and A (not on S).

5.1 Exact solution

Recall the value iteration Equation 2 (and making R dependent on a as well as on s):

$$V^t(s) = \max_a \left[R(s, a) + \gamma \sum_{s'} Pr(s'|s, a) V^{t-1}(s') \right] \tag{5}$$

In the partially observable case, the states are replaced with belief states, $b(s)$, and the sum over next states is now an integral:

$$V^t(b(s)) = \max_a \left[\sum_s R(s,a)b(s) + \gamma \int_{b(s')} Pr(b(s')|b(s),a)V^{t-1}(b(s')) \right] \tag{6}$$

In a POMDP, we can show that the value functions always remain piecewise linear and convex [KLC98, Lov91]. This is due to the linearity of the utility function's definition, i.e., that the utility of a lottery that obtains $o_i, i = 1 \ldots k$ with probability p_i is the weighted sum of the utilities of each o_i:

$$u([o_1 : p_1, o_2 : p_2, \ldots, o_k : p_k]) = \sum_i p_i u(o_i)$$

That is, we can write

$$V^{t-1}(b(s')) = \max_{\alpha \in \mathbf{V}^{t-1}} \left(\sum_{s'} \alpha(s')b(s') \right)$$

where $\alpha(s)$ is a value function on s which we will call an *alpha vector*, and $\mathbf{V}^{t-1}$ in the $\max$ is a set of alpha vectors, such that:

$$V^t(b(s)) = \max_a \left[\sum_s R(s,a)b(s) + \gamma \int_{b(s')} Pr(b(s')|b(s),a) \max_{\alpha \in V^{t-1}} \sum_{s'} \alpha(s')b(s') \right] \tag{7}$$

We also know there is one $b(s')$ for each a, o pair:

$$b(s') = b_o^a(s') = \sum_s P(o|s')P(s'|s,a)b(s).$$

The integration over $b(s')$ is a sum over all possible next belief states, each of which is defined according to a a, o pair as b_o^a. What will this look like for a particular $b(s)$ and a particular a? For each possible observation o, it would lead to a b_o^a that would select a particular $\alpha(s')$ in the $\max_{\alpha \in V^{t-1}}$ operation. Let's denote by $\alpha_a^{o,b}$ the particular $\alpha(s')$ selected after a was taken in $b(s)$ and o was observed. For this $b(s)$ and o combination, the term $P(b(s')|b(s),a) = \delta(b(s'), b_o^a(s'))$,[1] and so the integration becomes a sum over observations, with the term for o using $\alpha_a^{o,b}$, such that the term:

$$\int_{b(s')} Pr(b(s')|b(s),a) \max_{\alpha \in V^{t-1}} \sum_{s'} \alpha(s')b(s')$$

becomes:

$$\sum_o \left[\sum_{s'} \alpha_a^{o,b}(s')b_o^a(s') \right]$$

But remember, this sum exists *for every value of* $b(s)$, and the set of $\alpha_a^{o,b}(s')$ used for each a, o pair may be different for each of them! How many such sets are there? Let us call the set of $\alpha_a^{o,b}(s') \forall o$ chosen in the sum for $b(s)$ as $\boldsymbol{\alpha}_a^b$, such that there are $|O|$ elements in the set, one alpha vector for each possible observation (out of $|O|$ possible observations). Then notice that it is possible that there is *some* belief point $b(s)$ that would "choose" each and every possible different set $\boldsymbol{\alpha}_a^b$. If there are $|V|$ alpha

[1] $\delta(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$

vectors at step $t-1$, then there are $|V|^{|O|}$ possible sets $\boldsymbol{\alpha}_a^b$ (each observation "chooses" one possible α vector leading to an exponential number of combinations - the number of ways of assigning one of $|V|$ elements to each of $|O|$ options). Denoting this set of sets as $\boldsymbol{\alpha}_a$, we can compute the integral in Equation 7 by computing a cross-sum[2] over $\boldsymbol{\alpha}_a$, creating a new set of $|A||V|^{|O|}$ alpha vectors:

$$\boldsymbol{\alpha}'_a \leftarrow R(s,a) + \gamma \oplus_o \boldsymbol{\alpha}_a(s)$$

where the set $\boldsymbol{\alpha}_a(s)$ is the set of "backed up" alpha vectors for action a from the previous value function. Denoting a set of possibilities as $\{f(x)\}_x = \{f(x_o), f(x_1), \ldots\}$, we have:

$$\boldsymbol{\alpha}_a(s) = \left\{ \left\{ \sum_{s'} \alpha(s') P(o|s') P(s'|s,a) \right\}_{\alpha} \right\}_{o} \tag{8}$$

This set has an element for each observation o which is itself a set of α vectors for each α vector in V^{t-1}. Thus, the set $\boldsymbol{\alpha}'_a$ is the set of all possible ways of choosing one α vector in V^{t-1} for each observation o and summing them. We are doing this because, for each action a, there may be some belief point that might make use of every single one of these combinations. For this belief point, it is precisely that combination of next α vectors that is best. We won't know until we generate all of them and then seek to prune the dominated ones.

We then prune all vectors that are dominated at all belief points $b(s)$, yielding the new set of alpha vectors that form V^t. Each alpha vector in this set has an associated action a which the optimal action for the set of belief points that have this alpha vector as maximal in $\boldsymbol{\alpha}_a$. That is, V^t has a new α vector to "represent" each belief point $b(s)$, such that each piecewise linear piece of the new value function (a new alpha vector) will have been computed using a sum over a specific combination of observations and old alpha vectors. We may, of course, be doing too much work here in computing alpha vectors that are dominated everywhere, but this naïve method is guaranteed to get them all. Notwithstanding this additional computation at each step, the number of alpha vectors may increase exponentially at each step, leading to an infinitude of alpha vectors (one for each belief point) in the worst case. Many other improvements have been made in order to speed up exact value iteration for POMDPs, but most of the major recent improvements came from using so-called "point-based" methods.

5.2 Point-Based solution

In a point-based method [SV05, PGT03], instead of computing the value function for every belief point, we start from an initial belief (the current belief), say b^0, and compute a reachable set of belief points by iteratively computing b_o^a for every combination of a and o. The exact method of computing the reachable set is not as important as finding a set that spans the regions one expects to reach over a given horizon. One can also start from a set of beliefs (e.g. all the possible starting beliefs for an agent in this situation).

Once the reachable set is defined, we compute backups for each belief sample:

$$\alpha_a^{k+1}(s) = R(s,a) + \gamma \sum_o \left(\sum_{s'} P(o|s') P(s'|s,a) \arg\max_{\alpha_i} b_o^a(s') \alpha_i^k(s') \right)$$

where $\alpha_i^k(s)$ is the i^{th} alpha vector in the k-stage to go value function and $b_o^a(s')$ is defined as above. We then take the best of these over a at each belief sample, and finally throw out any completely dominated vectors we have created to obtain a new set of alpha vectors.

[2] A cross-sum $P \oplus Q = \{p+q | p \in P, q \in Q\}$ and a cross-sum over a set $\mathbf{P} = \{P_1, P_2, \ldots\}$ is $\oplus_i \mathbf{P} = P_1 \oplus P_2 \oplus \ldots$

5.3 Forward Search

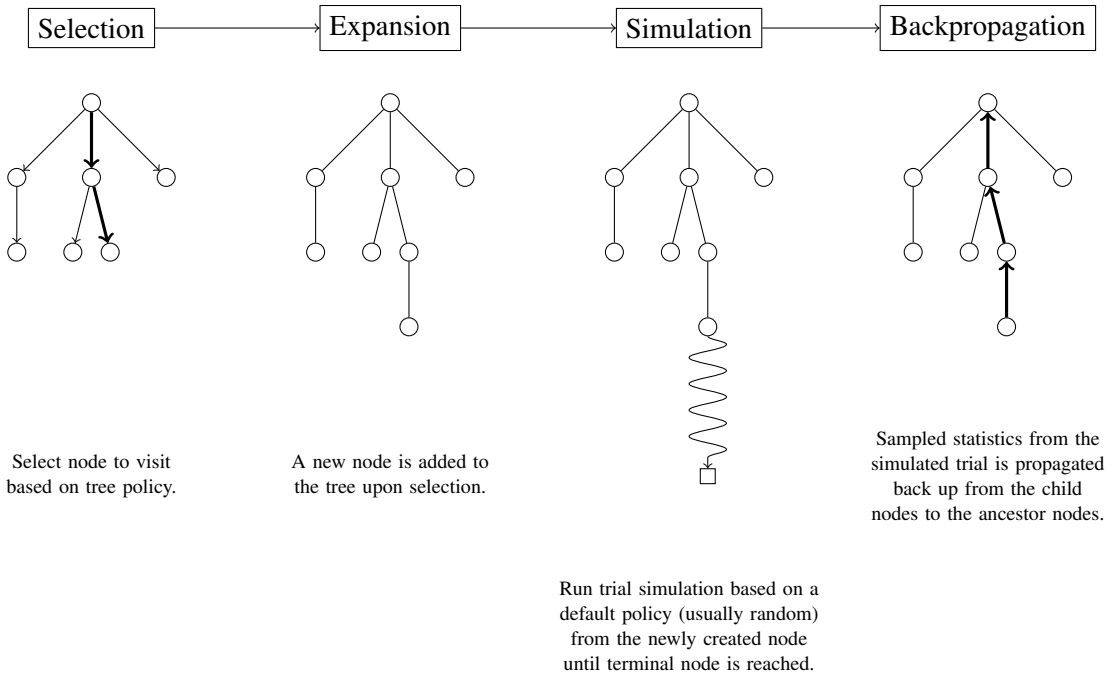
Finally, in a *forward search* method, we also start from an initial belief point (a single one this time) and we iteratively grow the search tree using a Monte-Carlo method. That is, we start with a tree with a single node with b^0, then add children for each a, o combination. Each child node has a new belief b_o^a, and an associated value $V(b_o^a) = \sum_s R(s,a) b_o^a(s)$. The parent node (the root node in this case) can then compute the expected value for taking each action a by weighting the values of each child (a, o) by the probability of that branch being followed should action a be taken, $P(o|s,a)$. Calling these expectations Q_a, The expected value for this node (the root node in this case) is then $V(b) = \sum_s R(s,a) b(s) + \gamma \max_a Q_a$. The method is then applied recursively to each child node, and the resulting value estimate at the root is then recomputed based on the new expected values of each child.

The challenge with this method is how to expand the tree in a sensible way such that more effort is spent expanding the parts of the tree that are likely to be reached, and simultaneously likely to yield high values. This challenge has two parts:

1. Expanding those branches corresponding to observations that are more likely at each node. This can be accomplished through Monte-Carlo sampling at each node and expansion of the more likely children

2. Expanding those branches corresponding to actions that are likely to yield high rewards. This corresponds to the bandit problem, or the exploration/exploitation tradeoff in reinforcement learning, and is usually approached by defining some exploration bonus (e.g. based on confidence bounds [KS06]).

The resulting family of algorithms are known as Monte-Carlo Tree Search, or MCTS algorithms, originally explored for POMDPs in [SV10], and famously applied to the game of GO in [SSS$^+$17]. A key idea behind these methods is the use of "rollouts" which are fast and deep "probes" into the tree, usually done for a POMDP using a single belief point that is rapidly updated and evaluated with respect to the reward. The idea is shown graphically here:



The idea is to descend the tree using the policy defined by the tree (or could be another policy e.g. including an exploration bonus), expand by one level or node when a leaf is reached, and then run a series of rollouts at that leaf to get a rough estimate of what its value might be. Whatever new information is gained is then propagated back up the tree and value estimates are adjusted on the way back up. Essentially, the tree is grown in a direction defined by these fast deep probes which give a rough estimate that is then refined by the more precise growing of the tree nodes. Its like building a road through a dense forest into an unknown land by continuously sending out scouts who report on promising looking directions that the road builders follow. The scouts may miss an easy route and the road would be built over a more difficult terrain, but with sufficent scouts going in different directions this is less likely to occur. AlphaGo [SSS$^+$17] uses an MCTS method where the action choices are based on a deep reinforcement learning network, and the expected values at each node are represented with a second deep network.

6 Questions

1. Show that V^t converges to V^* as $t \to \infty$. Hint: show that the difference between successive iterations goes to zero as t gets large.
   SOLUTION: To do this you must write out the complete calcuation for V^t (or at least imagine what it would look like "unrolled"), and do the same for V^{t+1}. You will notice that these two formulae differ by a single term that is $O(\gamma^{t+1})$. Since $\gamma < 1.0$, this term goes to 0, so the iterations converge.
   Writing the time indices as meaning the steps to go to the goal (at $t = 0$):

$$V(S_t) = R(s_t) + \max_{a_t} \gamma \sum_{s_{t-1}} P(s_{t-1}|s_t, a_t) V(s_{t-1})$$

   leaving out the max and sum operators for clarity and writing P^t as simply P for any t, let's write this in shorthand as:

$$V_t = r + \gamma P V_{t-1}$$

we can expand to a horizon of T as:

$$\begin{aligned} V_t &= r + \gamma P \left[r + \gamma P V_{t-1} \right] \\ &= r + \gamma P r + \gamma^2 P^2 \left[r + \gamma P V_{t-2} \right] \\ &= r + \gamma P r + \gamma^2 P^2 r + \gamma^3 P^3 r + \ldots + \gamma^t P^t r \end{aligned}$$

Whereas

$$\begin{aligned} V_{t+1} &= r + \gamma P \left[r + \gamma P V_t \right] \\ &= r + \gamma P r + \gamma^2 P^2 \left[r + \gamma P V_{t-1} \right] \\ &= r + \gamma P r + \gamma^2 P^2 r + \gamma^3 P^3 r + \ldots + \gamma^t P^t r + \gamma^{t+1} P^{t+1} r \end{aligned}$$

Such that

$$V_{t+1} - V_t = \gamma^{t+1} P^{t+1} r$$

which goes to zero as $t \to \infty$ for $\gamma < 1$.

2. What is the optimal policy and value function for the MDP in Figure 2, given a discount factor of $\gamma = 0.9$. What if $\gamma = 0.8$? What if $\gamma = 0.7$?
SOLUTION: We apply the value iteration algorithm in Matlab as follows:

```
pA=[0,1,0,0,0;
    0,0,0.5,0,0.5;
    0,0,0,0.8,0.2;
    0,0,0,0,1;
    0,0,0,0,1];
pB=[0,0,0.25,0.75,0;
    0,0,0.3,0,0.7;
    0,0,0,0.5,0.5;
    0,0,0,0,1;
    0,0,0,0,1];
R=[0,2,-2,2,0];
V0=R;

%first iteration
Q1=[R'+0.9*pA*V0',R'+0.9*pB*V0']';
[V1,pi1]=max(Q1);
Q1=1.8000    1.1000   -0.5600    2.0000         0
    0.9000    1.4600   -1.1000    2.0000         0
V1=1.8000    1.4600   -0.5600    2.0000         0
pi1 =     1     2     1     1     1

% second iteration
Q2=[R'+0.9*pA*V1',R'+0.9*pB*V1']';
```

```
[V2,pi2]=max(Q2);

Q2=1.3140    1.7480   -0.5600    2.0000         0
   1.2240    1.8488   -1.1000    2.0000         0
V2=1.3140    1.8488   -0.5600    2.0000         0
pi2=1     2     1     1     1

%run to convergence
V=V2
converged=false;
epsilon=0.1
threshold=epsilon*(1-0.9)/0.9;
while converged==false
  Vnew=max([R'+0.9*pA*V',R'+0.9*pB*V']');
   converged=(max(abs(V-Vnew))<threshold);
   V=Vnew;
end

% value function is
V = 1.6639    1.8488   -0.5600    2.0000         0

%one more iteration
Qstar=[R'+0.9*pA*V1',R'+0.9*pB*V1']';
[Vstar,pistar]=max(Qstar);

Qstar=1.6639    1.7480   -0.5600    2.0000         0
    1.2240    1.8488   -1.1000    2.0000         0

%V has converged to optimal value function
Vstar = 1.6639    1.8488   -0.5600    2.0000         0

%optimal policy is do A=b in state 1, A=a otherwise
pistar = 1     2     1     1     1
```

3. Studenbot is a robot who is designed to behave just like a Waterloo student. He has four actions available to him:
   - **study**: studentbot's knowledge increases, studentbot gets tired
   - **sleep**: studentbot gets less tired
   - **party**: studentbot has a good time, but gets tired and loses knowledge
   - **take test**: studentbot takes a test (can take test anytime)

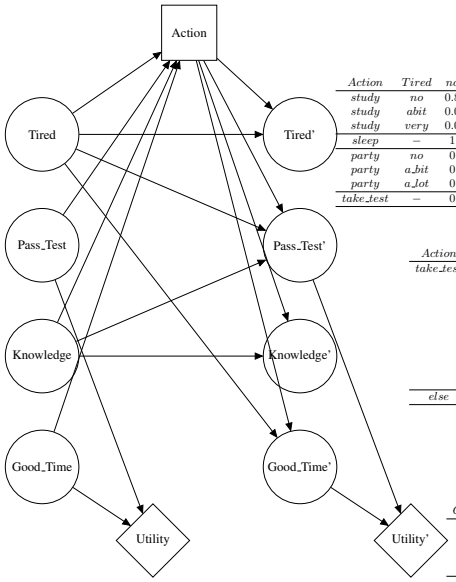
   He has four state variables and a total of 48 states:
   - **tired**: studentbot is tired (no/a bit/very)
   - **passtest**: studentbot passes test (no/yes)
   - **knows**: studentbot's state of knowledge (nothing/a bit/a lot/everything)

- **goodtime**: studentbot has a good time (no/yes)

He gets a big reward for passing the test, and a small one for partying. However, he can only reliably pass the test if his knowledge is everything and if he's not very tired. He gets tired if he studies, takes the test, or parties, and he recovers if he sleeps. He has a good time if he parties when he's not tired. His knowledge increases if he studies and decreases if he parties (he forgets things). His knowledge resets to nothing when he takes the test (so he has to start all over again), and stays the same if he sleeps.
The following figure shows the MDP for studentbot as a dynamic decision network, along with the conditional probability tables and utility function:

Action	*Tired*	*Knowledge*	*Knowledge'* *nothing*	*a_bit*	*a_lot*	*everything*
study	*no*	*nothing*	0.5	0.5	0	0
	no	*a_bit*	0	0.5	0.5	0
	no	*a_lot*	0.0	0.0	0.5	0.5
	no	*everything*	0		0	1.0
	a_bit	*nothing*	0.5	0.5	0	0
	a_bit	*a_bit*	0	0.5	0.5	0.0
	a_bit	*a_lot*	0	0	0.5	0.5
	a_bit	*everything*	0	0	0	1.0
	very	*nothing*	1.0	0	0	0
	very	*a_bit*	0	1.0	0.0	0.0
	very	*a_lot*	0	0	1.0	0.0
	very	*everything*	0	0	0	1.0
party	–	*nothing*	1.0	0.0	0	0
	no	*a_bit*	0.5	0.5	0.0	0.0
	no	*a_lot*	0.0	0.5	0.5	0.0
	no	*everything*	0	0	0.5	0.5
sleep	–	*nothing*	1.0	0.0	0	0
	–	*a_bit*	0.0	1.0	0.0	0.0
	–	*a_lot*	0.0	0.0	1.0	0.0
	–	*everything*	0	0.0	0.0	1.0
take_test	–	–	1.0	0.0	0	0

Action	*Tired*	*Good_Time'* *no*	*yes*
party	*no*	0.0	1.0
party	*a_bit/very*	1.0	0.0
other	–	1.0	0.0



Action	*Tired*	*Tired'* *no*	*a_bit*	*very*
study	*no*	0.8	0.2	0.0
study	*abit*	0.0	0.8	0.2
study	*very*	0.0	0.0	1.0
sleep	–	1	0	0
party	*no*	0	1	0
party	*a_bit*	0	0	1
party	*a_lot*	0	0	1
take_test	–	0	0	1

Action	*Tired*	*Knowledge*	*PassTest'* *no*	*yes*
take_test	*no*	*nothing*	1	0
	no	*a_bit*	0.9	0.1
	no	*a_lot*	0.7	0.3
	no	*everything*	0.1	0.9
	a_bit	*nothing*	1	0
	a_bit	*a_bit*	0.9	0.1
	a_bit	*a_lot*	0.7	0.3
	a_bit	*everything*	0.1	0.9
	very	–	1	0
else	–	–	1	0

Good_Time	*Pass_Test*	*U(Good_Time, Pass_Test)*
yes	*yes*	22
yes	*no*	2
no	*yes*	20
no	*no*	0

What is the optimal policy for studentbot?
SOLUTION: Let us denote the variables using a shorthand as $A = Action$, $GT = Good\_Time$, $K = Knowldge$, $PT = Pass\_Test$, $T = Tired$, and assign the following factors:

$$f_0(GT', A, T) = P(GT'|A, T)$$
$$f_1(K', A, K, T) = P(K'|A, K, T)$$
$$f_2(PT', A, K, T) = P(PT'|A, K, T)$$
$$f_3(T', A, T) = P(T'|A, T)$$
$$f_4(PT', GT') = Utitity(PT', GT')$$

Then, we carry out a variable elimination step on the network by summing out GT', K', PT', T'

(in that order), then max-ing out A and then relabeling all variables to be primed again:

$$
\begin{aligned}
f_5(A, PT') &= \sum_{GT'} f_0(GT', A, T) f_4(PT', GT') \\
f_6(A, K, T) &= \sum_{K'} f_3(K', A, K, T) = 1.0 \\
f_7(A, K, T) &= \sum_{PT'} f_5(A, PT') f_2(PT', A, K, T) \\
f_8(A, K, T) &= \sum_{T'} f_3(T', A, T) = 1.0 \\
f_9(K, T) &= \max_A f_7(A, K, T) f_6(A, K, T) f_8(A, K, T)
\end{aligned}
$$

Note that the two factors f_6 and f_8 are both 1.0, and so in fact, we did not need to sum over K' and T'. This is because they are not ancestors of the utility function at the last time step. They will be so, however for each earlier time step. We now add $\gamma f_9(K', T')$ to the set $f_0 \ldots f_4$ and start again:

$$
\begin{aligned}
f_{10}(A, PT') &= \sum_{GT'} f_0(GT', A, T) f_4(PT', GT') \\
f_{11}(A, K, T, T') &= \gamma \sum_{K'} f_3(K', A, K, T) f_9(K', T') \\
f_{12}(A, K, T) &= \sum_{PT'} f_{10}(A, PT') f_2(PT', A, K, T) \\
f_{13}(A, K, T) &= \sum_{T'} f_3(T', A, T) f_{11}(A, K, T, T') \\
f_{14}(K, T) &= \max_A f_{12}(A, K, T) f_{13}(A, K, T)
\end{aligned}
$$

We now add $\gamma f_{14}(K', T')$ to the set $f_0 \ldots f_4$ and start again...

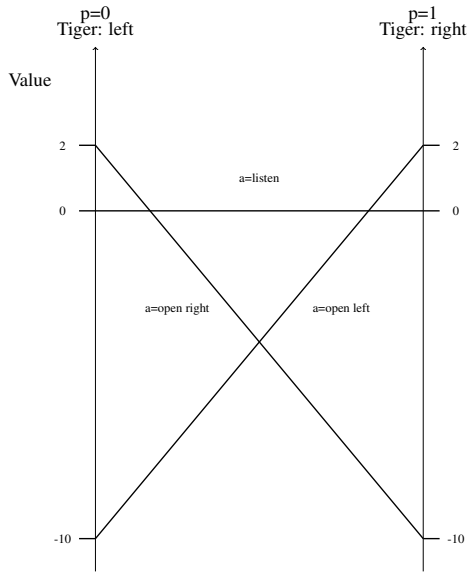
$$
\begin{aligned}
f_{15}(A, PT') &= \sum_{GT'} f_0(GT', A, T) f_4(PT', GT') \\
f_{16}(A, K, T, T') &= \gamma \sum_{K'} f_3(K', A, K, T) f_{14}(K', T') \\
f_{17}(A, K, T) &= \sum_{PT'} f_{15}(A, PT') f_2(PT', A, K, T) \\
f_{18}(A, K, T) &= \sum_{T'} f_3(T', A, T) f_{16}(A, K, T, T') \\
f_{19}(K, T) &= \max_A f_{17}(A, K, T) f_{18}(A, K, T)
\end{aligned}
$$

Note that this recursion is the same as the previous step, and creates a factor on K, T only which is the value function. It only depends on K and T because it has the same values for all values of the other variables. When this function stops changing, we have the optimal value function, and the $\arg\max$ of the last product of factors is the optimal policy.

4. In Section 4 we derived the value iteration algorithm for MDPs by applying the standard variable elimination algorithm for decision networks. Attempt to do the same for a POMDP. What happens and why does this not scale properly?
   SOLUTION:
   In this case, the state variables S are not parents of the decision nodes anymore, and so must all be summed out at the start, leading to an exponential factor blow up.
5. The tiger problem is a classic "minimal working example" POMDP problem usually stated as follows. You are in front of two doors, behind one of which is a tiger and behind the other is a bag of money. Opening the door with the tiger has a value of -10, whereas opening the door with the money has a value of $+2$. You can also listen which reveals the location of the tiger with probability 0.8. You initially don't know which door the tiger is behind. The discount factor $\gamma = 0.9$, so you can listen and then open a door, or you can open a door straight away. Listening doesn't yield any reward, but allows you to gather information that will lead to a better decision (even though the delayed reward so obtained will be worth less).
   SOLUTION:
   A quick calcuation shows that opening a door right away from a belief of $[0.5, 0.5]$ that the tiger is behind the $[left, right]$ door gives you $0.5 \times 2 + 0.5 \times -10 = -4$. Once you listen once, you know the location of the tiger with probability 0.8 so the reward of opening the door (not the one you heard the tiger sounds coming from!) after listening is $0.8 * 2 + 0.2 * (-10) = -0.4$. If you listen twice and hear the tiger behind the same door twice, your belief in the tiger being behind that door goes up to 0.94, and so your payoff for opening the door goes up to $0.94 * 2 + 0.06 * (-10) = 1.28$.
   Optimal solution: We write α vectors as tuples $[v_{left}, v_{right}]$ representing the linear function of p, the probability that the tiger is behind the right door, of $(1-p)v_{left} + pv_{right}$. We start with a single α vector which we can write as $[0, 0]$, and consider the set defined by Equation (8) as

action	α
listen	$[0, 0]$
open left	$[-10, 2]$
open right	$[2, -10]$

These α vectors are shown here:



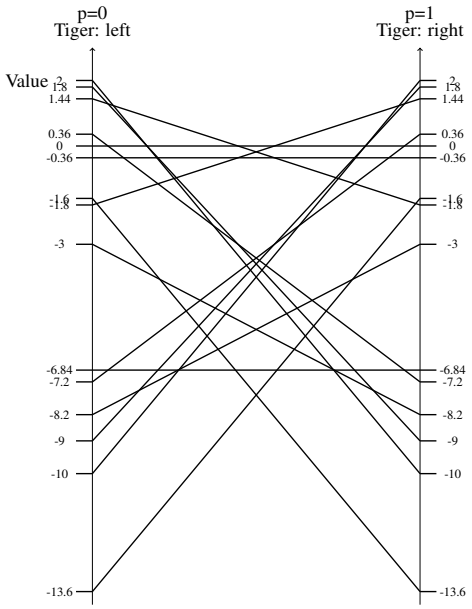
Now, we create a new set of α vectors, one for each action and cross-sum of the two observations (hear left and hear right), using Equation (8). First compute the backed up α vectors for each action, previous α vector and observation. The actions of opening doors have the same value for each observation since the observation is non-informative (or isn't made):

a	$\alpha(s')$	o	$\alpha(s) = \sum_{s'} \alpha(s')P(o\|s')P(s'\|s,a)$
listen	$[0, 0]$	hear left	$[0 * 0.5 * 0.5 + 0 * 0.5 * 0.5, 0 * 0.5 * 0.5 + 0 * 0.5 * 0.5] = [0, 0]$
listen	$[0, 0]$	hear right	$[0 * 0.5 * 0.5 + 0 * 0.5 * 0.5, 0 * 0.5 * 0.5 + 0 * 0.5 * 0.5] = [0, 0]$
listen	$[-10, 2]$	hear left	$[-10 * 0.8 * 1.0 + 2 * 0.2 * 0.0, -10 * 0.8 * 0.0 + 2 * 0.2 * 1.0] = [-8, 0.4]$
listen	$[-10, 2]$	hear right	$[-10 * 0.2 * 1.0 + 2 * 0.8 * 0.0, -10 * 0.2 * 0.0 + 2 * 0.8 * 1.0] = [-2, 1.6]$
listen	$[2, -10]$	hear left	$[2 * 0.8 * 1.0 - 10 * 0.2 * 0.0, 2 * 0.8 * 0.0 - 10 * 0.2 * 1.0] = [1.6, -2]$
listen	$[2, -10]$	hear right	$[2 * 0.2 * 1.0 - 10 * 0.8 * 0.0, 2 * 0.2 * 0.0 - 10 * 0.8 * 1.0] = [0.4, -8]$
open left	$[0, 0]$	-	$[0 * 0.5 * 0.5 + 0 * 0.5 * 0.5, 0 * 0.5 * 0.5 + 0 * 0.5 * 0.5] = [0, 0]$
open left	$[-10, 2]$	-	$[-10 * 0.5 * 0.5 + 2 * 0.5 * 0.5, -10 * 0.5 * 0.5 + 2 * 0.5 * 0.5] = [-2, -2]$
open left	$[2, -10]$	-	$[2 * 0.5 * 0.5 - 10 * 0.5 * 0.5, 2 * 0.5 * 0.5 - 10 * 0.5 * 1.5] = [-2, -2]$
open right	$[0, 0]$	-	$[0 * 0.5 * 0.5 + 0 * 0.5 * 0.5, 0 * 0.5 * 0.5 + 0 * 0.5 * 0.5] = [0, 0]$
open right	$[-10, 2]$	-	$[-10 * 0.5 * 0.5 + 2 * 0.5 * 0.5, -10 * 0.5 * 0.5 + 2 * 0.5 * 0.5] = [-2, -2]$
open right	$[2, -10]$	-	$[2 * 0.5 * 0.5 - 10 * 0.5 * 0.5, 2 * 0.5 * 0.5 - 10 * 0.5 * 0.5] = [-2, -2]$

Then, compute the cross sums. In the following table, the third and fifth columns each contain one of the three alpha vectors from the previous set backed up by multiplying by $P(o|s')$ and $P(s'|s, a)$ (which is $\delta(s, s')$ for $a = listen$ and 0.5 for the other two actions, as shown in the table above), while the second and fourth columns show the corresponding next action that will be taken given that observation. The last column is the sum of both discounted by γ and added to the reward for taking that action.

	hear left		hear right		
action	action	value	action	value	α
listen	listen	$[0, 0]$	listen	$[0, 0]$	$[0, 0] * 0.9 + [0, 0] * 0.9 + [0, 0] = [0, 0]$
	listen	$[0, 0]$	open left	$[-2, 1.6]$	$[0, 0] * 0.9 + [-2, 1.6] * 0.9 + [0, 0] = [-1.8, 1.44]$
	listen	$[0, 0]$	open right	$[0.4, -8]$	$[0, 0] * 0.9 + [0.4, -8] * 0.9 + [0, 0] = [0.36, -7.2]$
	open left	$[-8, 0.4]$	listen	$[0, 0]$	$[-8, 0.4] * 0.9 + [0, 0] * 0.9 + [0, 0] = [-7.2, 0.36]$
	open left	$[-8, 0.4]$	open left	$[-2, 1.6]$	$[-8, 0.4] * 0.9 + [-2, 1.6] * 0.9 + [0, 0] = [-9, 1.8]$
	open left	$[-8, 0.4]$	open right	$[0.4, -8]$	$[-8, 0.4] * 0.9 + [0.4, -8] * 0.9 + [0, 0] = [-6.84, -6.84]$
	open right	$[1.6, -2]$	listen	$[0, 0]$	$[1.6, -2] * 0.9 + [0, 0] * 0.9 + [0, 0] = [1.44, -1.8]$
	open right	$[1.6, -2]$	open left	$[-2, 1.6]$	$[1.6, -2] * 0.9 + [-2, 1.6] * 0.9 + [0, 0] = [-0.36, -0.36]$
	open right	$[1.6, -2]$	open right	$[0.4, -8]$	$[1.6, -2] * 0.9 + [0.4, -8] * 0.9 + [0, 0] = [1.8, -9]$
open left	listen	$[0, 0]$	listen	$[0, 0]$	$[0, 0] * 0.9 + [0, 0] * 0.9 + [-10, 2] = [-10, 2]$
	-	$[-2, -2]$	-	$[-2, -2]$	$[-2, -2] * 0.9 + [-2, -2] * 0.9 + [-10, 2] = [-13.6, -1.6]$
open right	listen	$[0, 0]$	listen	$[0, 0]$	$[0, 0] * 0.9 + [0, 0] * 0.9 + [-10, 2] = [2, -10]$
	-	$[-2, -2]$	-	$[-2, -2]$	$[-2, -2] * 0.9 + [-2, -2] * 0.9 + [2, -10] = [-1.6, -13.6]$

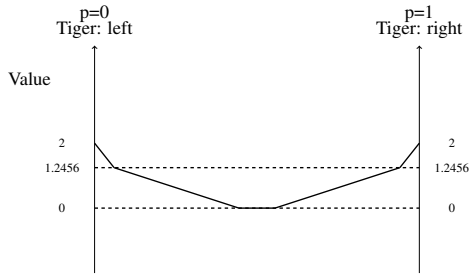
These vectors are shown here



Upon inspection (this could be done by sorting through the vectors and finding the dominated ones), we see only three vectors remain, as shown in the table below, along with the intervals over which they are maximal ($[p_{min}, p_{max}]$)

action	observation		α	range as max α	
$t = 1$	hear left	hear right	$t = 1$	p_{min}	p_{max}
listen	listen	listen	$[0, 0]$	0.444	0.556
	listen	open left	$[-1.8, 1.44]$	0.556	0.94
	open right	listen	$[1.44, -1.8]$	0.06	0.444
open left	listen	listen	$[-10, 2]$	0.94	1.0
open right	listen	listen	$[2, -10]$	0.0	0.06

And the optimal 2-stage to go value function thus looks like this:

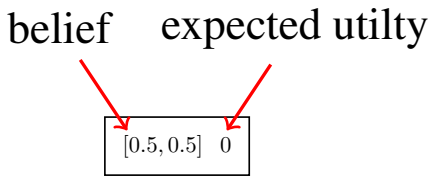


The two extreme α vectors represent opening a door immediately and then listening on the last round. This only works if beliefs are skewed to within 0.06 of certainty. The α vector in the middle represents listening twice, and those in the mid-range (from 0.06 to 0.444 and from 0.555 to 0.94) represent the policy of listening, then opening the door if it agrees with the belief, otherwise listening again. That is, suppose the belief is $p = 0.75$ (the agent believes the tiger is behind the right door with probability 0.75), then listening is optimal, and if the observation is "hear right", opening the left door is optimal, but if "hear left" is the observation, then listening again is optimal.

Forward search: Let us expand the game tree for this problem starting from a belief of $[0.5, 0.5]$- write out all combinations up to a depth of 4 decisions and then use that to decide what the opening move should be. Do this with a slight change to the reward function by making the money bag worth 4 instead of 2 (this makes the decision tree a bit more interesting).

We begin with a single node showing the initial belief and the expected value at that belief, computed as

$$U(b) = \sum_s R(s)b(s)$$



We then add the first set of branches, one for each action-observation pair, where the new beliefs are formed as:

$$P(s'|b(s), a, o') \propto P(o'|s')P(s'|b(s), a) = P(o'|s') \sum_s P(s'|s, a)b(s)$$

e.g. for $b(s) = [0.5, 0.5]$ and denoting $l = left, r = right, li = listen$:

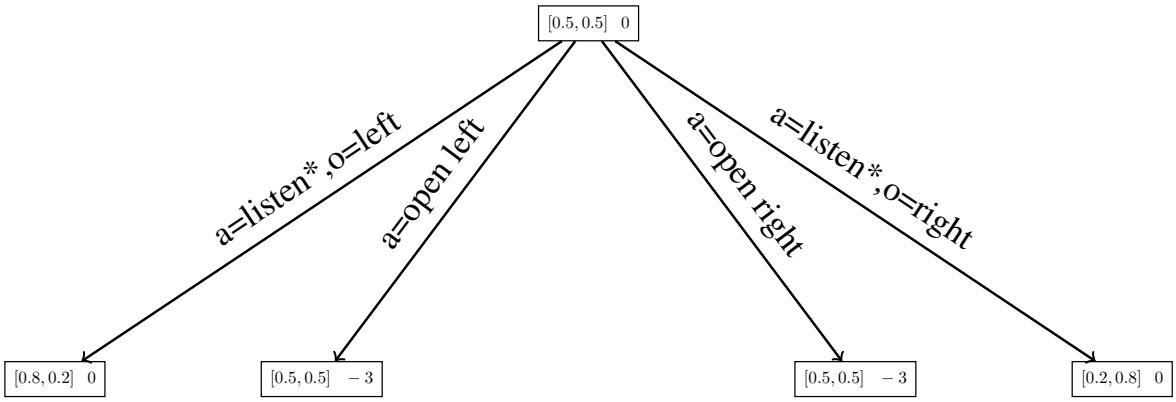
$$\begin{aligned} P(s' = l|b(s), a = li, o' = l) \propto P(o' = l|s' = l) \big[&P(s' = l|s = l, a = li)b(s = l)+ \\ &P(s' = l|s = r, a = li)b(s = r)\big] \\ = 0.8 * [1.0 * 0.5 + 0.0 * 0.5] = 0.4& \end{aligned}$$

and

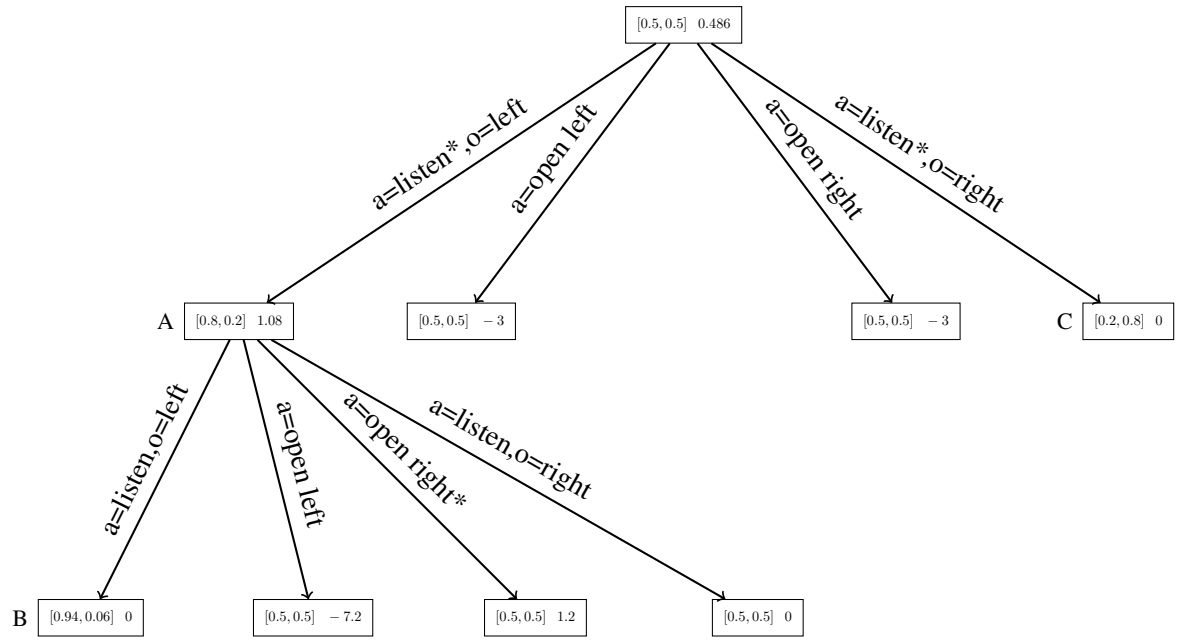
$$\begin{aligned} P(s' = r|b(s), a = li, o' = l) \propto P(o' = l|s' = r) \big[&P(s' = r|s = l, a = li)b(s = l)+ \\ &P(s' = r|s = r, a = li)b(s = r)\big] \\ = 0.2 * [0.0 * 0.5 + 1.0 * 0.5] = 0.1& \end{aligned}$$

so that, after normalization

$$P(s'|b(s), a = li, o' = l) = \frac{1}{0.5}[0.4, 0.1] = [0.8, 0.2]$$



At this stage, we can consider each option and choose the best one, which will be to listen (as it yields an expected value of 0, whereas either open action yields -3). The best action to take is indicated with a *. Let's see what happens when we expand the left-most node one more level

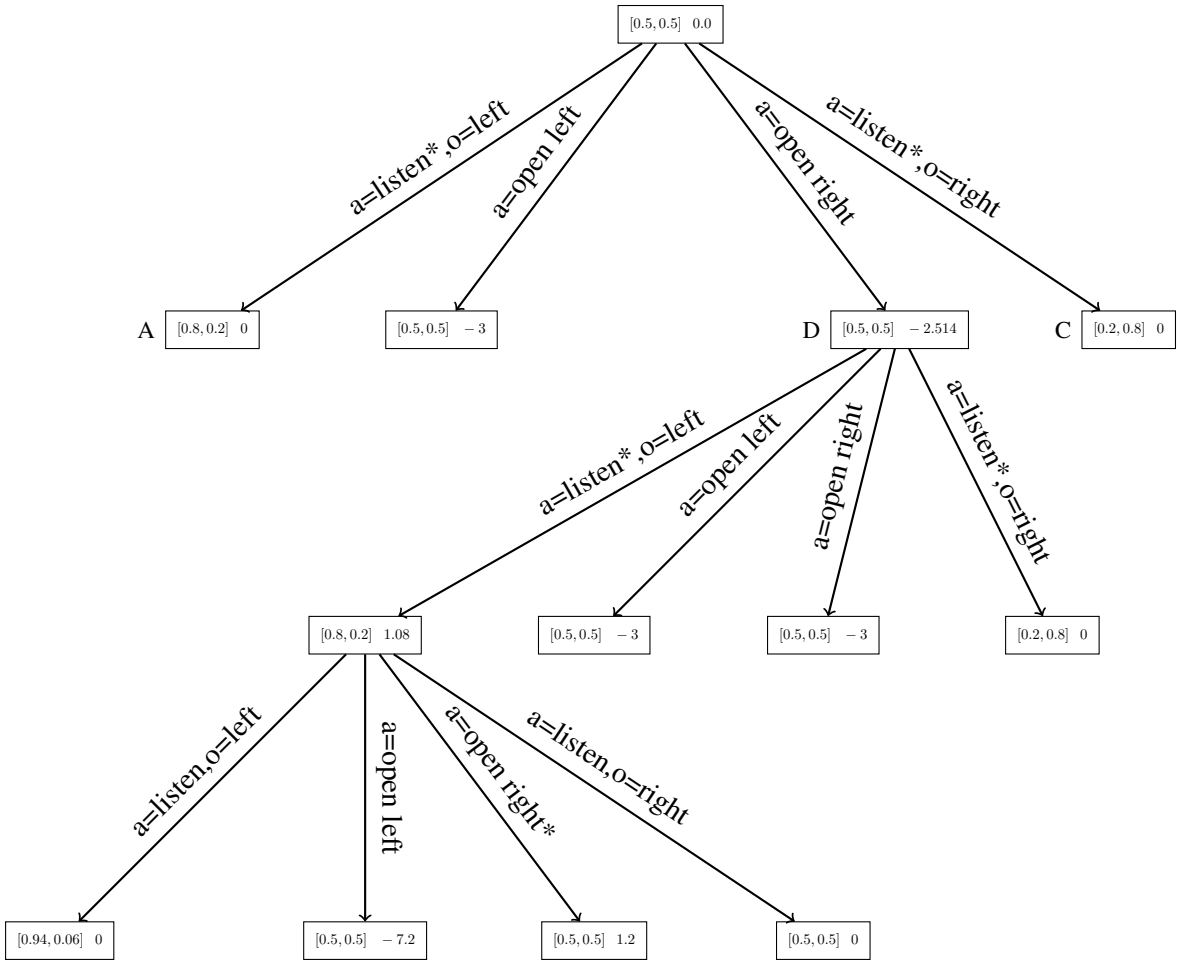


We get expected values for opening the left door at $0.8 * (-10) + 0.2 * (4) = -7.2$ and for opening the right door at $0.8 * 4 + 0.2 * (-10) = 1.2$, so things have improved because of the information gained. The best thing to do, at this second stage is to open the right door, and this has an expected value of 1.2. At the parent node (labeled A), this is discounted to give $1.2 * 0.9 + 0.0 = 1.08$. At the root node, the expected value is now the weighted sum over both observations when listening, but we've only expanded one of the two nodes, so the other one still has an expected value of 0. Thus, the root has $(0.5 * 1.08 + 0.5 * 0) * 0.9 + 0.0 = 0.486$.

If we also expanded the rightmost node (labeled C), it would get the same value as node A (1.08) due to the symmetry of the problem (the right-most branch from the root, if expanded, looks the same as the left-most branch except with right/left swapped), so the value of the root node would be $(0.5 * 1.08 + 0.5 * 1.08) * 0.9 + 0.0 = 0.972$. The best policy with two stages to go is to listen and then open a dooor. Let's expand one more level to see if we should listen twice though!

Now we see that opening the right door after listening twice and hearing the tiger from the left twice has an expected value of 3.16. However, discounted this is 2.84, which becomes the value for the parent node (labeled B). The same value, discounted again becomes 2.04 of the new expected value for node A, because the probability of observing $o' = left$ when at node B is 0.8 (and $2.84 * 0.9 * 0.8 = 2.04$). The other part comes from observing $o' = right$ from node B, and this has an expected value of 0 if expanded out one more level (not shown). Thus, the expected value at node B is now $(2.84 * 0.9 * 0.8 + 0 * 0.9 * 0.2 = 2.04)$. The best action to take at node B has now changed from opening the right door to listening a second time. The value at the root node is now the expected value of listening, which, since we still have not expanded out the right-most node (C), is $2.04 * 0.9 * 0.5 + 0.0 * 0.9 * 0.5 + 0.0 = 0.918$. If we also expanded the symmetric right-most branch two levels, the root node value would be $2.04 * 0.9 * 0.5 + 2.04 * 0.9 * 0.5 = 1.836$ but the best action to take (listen) stays the same.

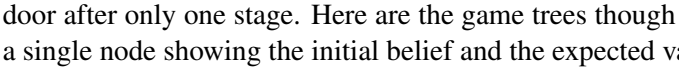
Note that we didn't expand the middle nodes (e.g. node D), but we could have done this instead of expanding node A, yielding



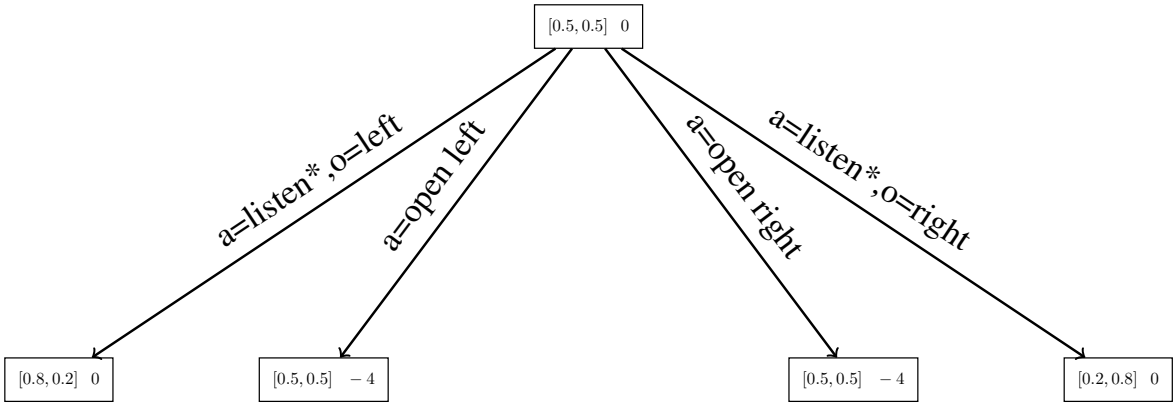
And the value at node D is now $0.5 * 1.08 * 0.9 + 0.5 * 0 * 0.9 - 3.0 = -2.514$, but the value at the root stays 0 because the optimal action is still to listen. **Monte-Carlo rollouts**: Notice that we had to choose at each step which node we were going to expand. This is not a trivial choice and in general will significantly influence the policy that is discovered in a limited amount of time. In order to get a rough estimate of how good each of the children of node B are, we could start from each of them and do a deep "probe" into the tree, simulating a single randomly selected action and observation at each step, then propagating the values obtained all the way down this long branch back up and adding to the current value at each child node. Doing this multiple times and averaging the results gives the rough estimate we seek.

We can do this for the original rewards of 2 and -10, but the policy at node B is not to open the door after only one stage. Here are the game trees though as a another example. We begin with a single node showing the initial belief and the expected value at that belief, computed as

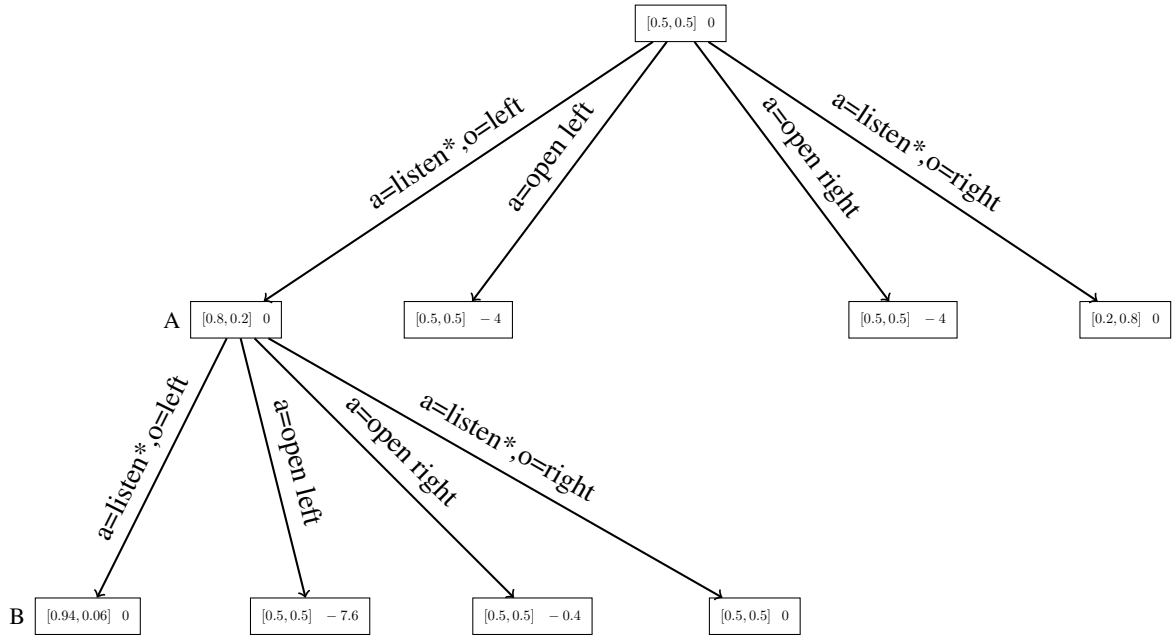
$$U(b) = \sum_s R(s)b(s)$$



We then add the first set of branches, one for each action-observation pair, where the new beliefs are formed in the same way as before



At this stage, we can consider each option and choose the best one, which will be to listen (as it yields an expected value of 0, whereas either open action yields -4). Let's see what happens when we expand the left-most node one more level



We get expected values for opening the left door at $0.8 * (-10) + 0.2 * (2) = -7.6$ and for opening the right door at $0.8 * (2) + 0.2 * (-10) = -0.4$, so things have improved because of the information gained. The best thing to do, however, is again to listen and so the rewards at the parent node (labeled A) stay the same (at 0). We need to expand one level further to find anything interesting:

Now we see that opening the right door after listening twice and hearing the tiger from the left twice has an expected value of 1.28. However, discounted this is 1.15, which becomes the value for the parent node (labeled B). The same value, discounted again becomes 0.8 of the new expected value for node A, because the probability of observing $o' = left$ when at node B is 0.8. The other part comes from observing $o' = right$ from node B, and this has an expected value of 0 if expanded out one more level (not shown). Thus, the expected value at node B is now $(1.15 * 0.9 * 0.8 + 0 * 0.9 * 0.2 = 0.82)$. The value at the root node is now the expected value of listening, which, since the right-most branch is not expanded, is $0.82 * 0.9 * 0.5 + 0 * 0.9 * 0.5 = 0.369$ but the best action to take (listen) stays the same.

7 Further Reading

The standard text on MDPs is Puterman's book [Put94], while this book gives a good introduction [MK12].

References

[KLC98] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of European Conference on Machine Learning*, 2006.

[Lov91] W. S. Lovejoy. A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28:47–66, 1991.

[MK12] Mausam and Andrey Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Morgan Claypool, June 2012.

[PGT03] Joelle Pineau, Geoff Gordon, and Sebastian Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In *Proc. IJCAI*, 2003.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, NY., 1994.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 10 2017.

[SV05] Matthijs T. J. Spaan and Nikos Vlassis. Perseus: Randomized point-based value iteration for POMDPs. *Journal of Artificial Intelligence Research*, 24:195–220, 2005.

[SV10] David Silver and Joel Veness. Monte-carlo planning in large POMDPs. In J.D. Lafferty, C.K.I. Williams, J. Shawe-Taylor, R.S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems (NIPS) 23*, pages 2164–2172. Curran Associates, Inc., 2010.