Artificial Intelligence

Markov Decision Processes (AIMA 17)

Christopher Simpkins

Kennesaw State University

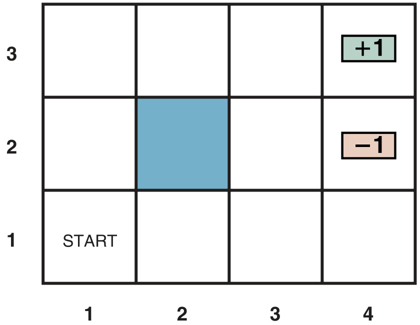
Sequential Decisions

In **sequential decision problems**, the agent's utility depends on a sequence of decisions.

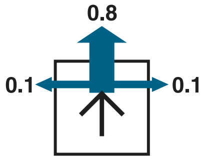
Sequential decision problems incorporate utilities, uncertainty, and sensing, and include search and planning problems as special cases.

- Markov decision processes (MDPs)
- k-Armed bandits
- Partially observable MPDs (POMDPs)

Worlds



(a)



(b)

Markov Decision Processes (MDPs)

A Markov decision process (MDP) is a 4-tuple $(S, A, Pr(s' \mid s, a), R(s))$, where

- S is a set of states,
- A, or $Action(s)$ is a set of actions, and
- $Pr(s' \mid s, a)$ is a transition function giving the probability that executing action a in state s will result in s'.
  - Many authors use $T(s, a, s')$
- $R(s, a, s')$ is the reward the world provides to an agent for arriving in state s' after executing action a in state s, bounded by $\pm R_{max}$
  - Many authors use $R(s')$, which is easier to think about – the reward for arriving in state s' regardless of the s, a pair in the previous time step.

Some definitions of MDPs include an initialization function, $I(s)$, which specifies the probability the the agent will start in some state $s \in S$, others specify a particular state s_0 from S as the start state.

MDP Solutions: Policies

Due to stochastic action results, fixed plans don't work. We need a function that returns a recommended action for every state. Such a function is called a **policy**:

$$\pi(s)$$

The policy can also be stochastic, $\pi(a \mid s)$, but for now we'll assume deterministic policies.

By the maximum expected utility principle, for a policy to be optimal the action it recommends for each state must have the highest **value** among all the action choices in that state.

The book uses state/action utility instead state/action value, but this is inconsistent with the book's previous definition of utility and its relationsip to preformance measures, and it's inconsistent with the terminology used by the reinforcement learning community, which is where this is headed. So we'll use value instead of utility.

Values and Trajectories

Since we're in the realm of sequential decisions, the value of an action depends on the *trajectory* – the sequence of states and actions – to which it leads.

Here again, we depart from the book's terminology. The book uses history, but the reinforcement learning community uses the term trajectory, τ, so we'll use τ. We'll also add reward to the trajectories, in line with reinforcement learning literature.

An experience sequence through an MDP is called a *trajectory*:

$$\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots$$

And the values in an MDP are defined in terms of a trajectory:

$$V_\tau(s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{t-1}, a_{t-1}, r_t)$$

We'll define these values after we learn about rewards.

Goals and Rewards

The purpose or goal of the agent is formalized in terms of a special signal, called the reward, passing from the environment to the agent.

- At each time step, the reward is a simple number, $R_t \in \mathbb{R}$.
- The agent's goal is to maximize the total amount of reward it receives.
  - This means maximizing not immediate reward, but cumulative reward in the long run.

We can clearly state this informal idea as the reward hypothesis:

> *All of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).*

So we represent goal states as the states giving the highest reward, with other states giving less reward according to some structure, which we'll discuss later. First, how do we use these rewards to determine values, which is what we need to derive policies . . .

Returns and Finite Time Horizons

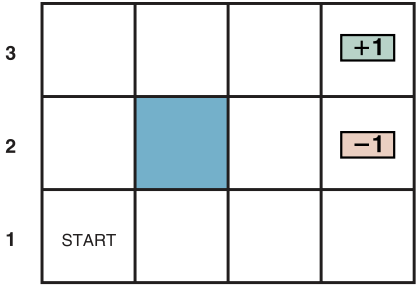
As the agent traverses an MDP along some trajectory, it collects reward on each state transition. The total reward collected in a trajectory is called the **return**, G. The simplest way to calculate return is simple addition. For a trajectory of length T:

$$G_t \doteq r_{t+1} + r_{t+2} + \cdots + r_T$$

With a finite horizon, there is an end time after which nothing happens, so if the end is time N:

$$V_\tau(s_0, a_0, r_1, s_1, a_1, \ldots, r_{N+k}, s_{N+k}) = V_\tau(s_0, a_0, r_1, s_1, a_1, \ldots, r_N, s_N)$$

Finite time horizons lead to **nonstationary** policies, i.e., policies that differ based on time. Consider that happens if $N = 3$ vs $N > 6$ in our 4×3 grid world:



Infinite Horizon Return

An infinite horizon gives us a **stationary** policy, because there is no time limit influencing the decision. From the current time step t:

$$G_t = r_{t+1} + r_{t+2} + \cdots + r_{t+k+1} = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

γ is the *discount factor* which says how much we discount future rewards. With $\gamma < 1$ it decays toward zero.

If rewards are bounded by $\pm R_{max}$ and $0 \leq \gamma < 1$, then, using the standard sum of an infinite geometric series.

$$V_\tau(s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{t-1}, a_{t-1}, r_t, s_t) = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \leq \sum_{t=0}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma} \tag{17.1}$$

So the values of infinite trajectories are finite.

- We will use inifinite horizon models.
- We can convert any finite horizon model into a infinite one by making terminal states *absorbing states*, which include a single action that loops to the absorbing state and gives zero reward.

State Values and Optimal Policies

The expected value of executing π in starting in state s is:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t), s_{t+1})\right]$$

The expectation $\mathbb{E}$ is with respect to the probability distribution over state sequences determined by s and π. Remember that $\pi(s_t)$ returns an action a_t. One or more policies, π^*, will have the highest of all values.

$$\pi^*(s) = \operatorname*{argmax}_{a \in A(s)} \sum_{s'} Pr(s' \mid s, a)\,[R(s, a, s') + \gamma V(s')] \tag{17.4}$$

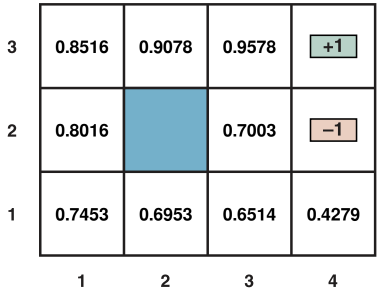
The value of a state is the expected reward for the next transition plus the discounted utility of the next state, assuming that the agent chooses the optimal action. That is, the utility of a state is given by

$$V(s) = \max_{a \in A(s)} \sum_{s'} Pr(s' \mid s, a)\,[R(s, a, s') + \gamma V(s')] \tag{17.5}$$

This is the **Bellman equation**, which is the basis of the value iteration algorithm we'll see soon.

Bellman Equation Example

What is the optimal action in state (1, 1) given the following state values?



We plug the state values above and $\gamma = 1$ into:

$$\max\Big([0.8(-0.04+\gamma V(1,2)) + 0.1(-0.04+\gamma V(2,1)) + 0.1(-0.04+\gamma V(1,1))], \qquad \text{(Up)}$$
$$[0.9(-0.04+\gamma V(1,1)) + 0.1(-0.04+\gamma V(1,2))], \qquad \text{(Left)}$$
$$[0.9(-0.04+\gamma V(1,1)) + 0.1(-0.04+\gamma V(2,1))], \qquad \text{(Down)}$$
$$[0.8(-0.04+\gamma V(2,1)) + 0.1(-0.04+\gamma V(1,2)) + 0.1(-0.04+\gamma V(1,1))]\Big) \qquad \text{(Right)}$$

which yields Up as the optimal action because Up is the action that maximizes $V((1,1))$.

Action Values and Optimal Policies

$$V(s) = \max_a Q(s, a)$$

The optimal action value function is:

$$Q^*(s_t, a_t) = \max_\pi \left(\mathbb{E}[G_t | s_t, a_t^\pi] \right)$$

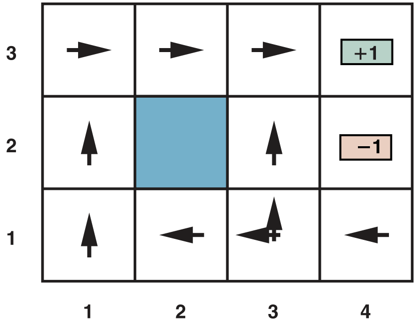
If we know Q^* we can use it to derive an optimal policy, π^*:

$$\pi^*(a_t | s_t) \leftarrow \operatorname*{argmax}_{a_t} (Q^*(s_t, a_t))$$

We'll see this idea when we solve MDPs using dynamic programming.

Optimal Policies

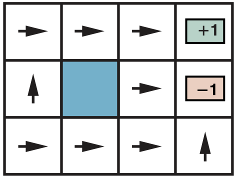
Notice that in State (3, 1) there are two optimal actions:



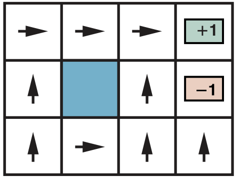
This results from the successor states having equal values, giving rise to multiple optimal policies.

Reward Structures

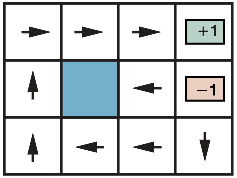
Notice how different reward structures for non-goal states influences the optimal policy:



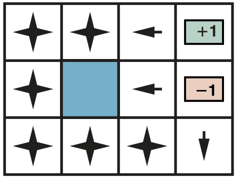
$r < -1.6497$



$-0.7311 < r < -0.4526$



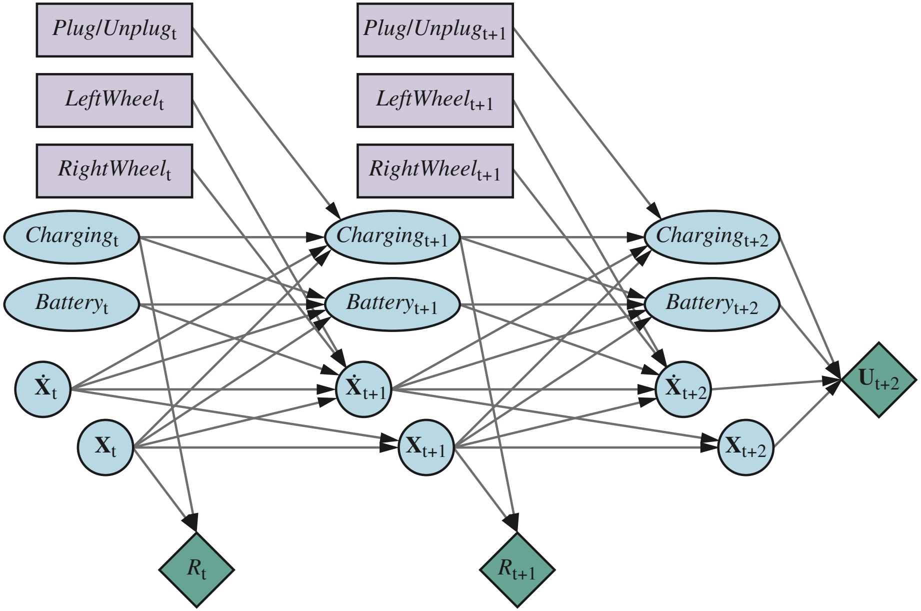
$-0.0274 < r < 0$



$r > 0$

In general, negative rewards "motivate" the agent to seek the goal quickly.

Representing MDPs



Bellman Value Update Rule

The value iteration algorithm initializes each state's value to a random value, then iteratively update these values by turning the Bellman equation into an update rule (the Bellman update):

$$V_{i+1}(s) \leftarrow R(s) + \max_{a \in A} \sum_{s'} T(s, a, s') V_i(s') \tag{1}$$

These updates are applied at the same time for all states, i.e., the values in iteration $i + 1$ are calculated from the values in iteration i. The value iteration algorithm is shown in Algorithm 1.

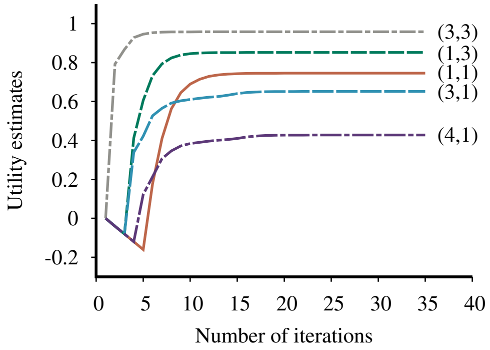
Algorithm 1 Value Iteration

```
$V \leftarrow$ random initial values
**repeat**
    $V' \leftarrow V$
    **for** each $s \in S$ **do**
        $V'(s) \leftarrow R(s) + \max_{a \in A} \sum_{s'} T(s, a, s') V(s')$
    $V \leftarrow V'$
**until** $V$ changes by a sufficiently small amount
```

Value Iteration Algorithm

```
function VALUE-ITERATION($mdp, \epsilon$) returns a utility function
  inputs: $mdp$, an MDP with states $S$, actions $A(s)$, transition model $P(s' \mid s, a)$,
            rewards $R(s, a, s')$, discount $\gamma$
          $\epsilon$, the maximum error allowed in the utility of any state
  local variables: $U$, $U'$, vectors of utilities for states in $S$, initially zero
                   $\delta$, the maximum relative change in the utility of any state

  repeat
      $U \leftarrow U'$; $\delta \leftarrow 0$
      for each state $s$ in $S$ do
          $U'[s] \leftarrow \max_{a \in A(s)}$ Q-VALUE($mdp, s, a, U$)
          if $|U'[s] - U[s]| > \delta$ then $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta \leq \epsilon(1-\gamma)/\gamma$
  return $U$
```

Value Iteration Converges Quickly



Policy Iteration

In policy iteration [?] we start with a random initial values and policy and alternate between two steps for each iteration i:

- **Policy evaluation.** Use policy π_i to calculate the values of each state using the discounted current values of their successor states. Since we are calculating the values under a particular policy, we drop the $\max$ operator:

$$V_{i+1}(s) = R(s) + \gamma \sum_{s'} T(s, a, s') V(s') \tag{2}$$

- **Policy improvement.** Calculate policy π_{i+1} using the values calculated in the previous step.

When policy improvement does not change the policy, an optimal policy has been found and policy iteration terminates.

Note that since the update equation used in policy evaluation is linear, we can use linear algebra to solve the set of simultaneous linear equations in $O(n^3)$. This method works fine for smaller state spaces but may be too expensive for large state spaces. A solution to this problem is known as modified policy iteration [?, ?], which combines policy iteration with value iteration by using a bounded number of Bellman updates to perform the policy evaluation step.

Policy Iteration Algorithm

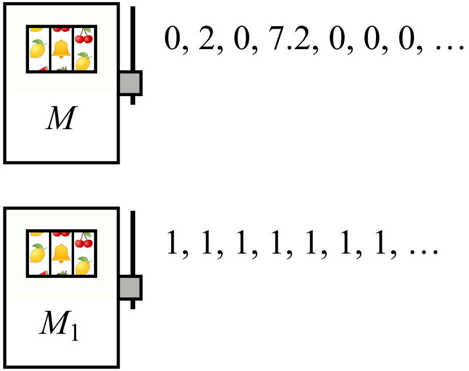
```
**function** POLICY-ITERATION($mdp$) **returns** a policy
  **inputs**: $mdp$, an MDP with states $S$, actions $A(s)$, transition model $P(s' \mid s, a)$
  **local variables**: $U$, a vector of utilities for states in $S$, initially zero
                       $\pi$, a policy vector indexed by state, initially random

  **repeat**
      $U \leftarrow$ POLICY-EVALUATION($\pi, U, mdp$)
      $unchanged? \leftarrow$ true
      **for each** state $s$ **in** $S$ **do**
          $a^* \leftarrow \underset{a \in A(s)}{\text{argmax}}\ \text{Q-VALUE}(mdp, s, a, U)$
          **if** $\text{Q-VALUE}(mdp, s, a^*, U) > \text{Q-VALUE}(mdp, s, \pi[s], U)$ **then**
              $\pi[s] \leftarrow a^*$; $unchanged? \leftarrow$ false
  **until** $unchanged?$
  **return** $\pi$
```

k-Armed Bandits

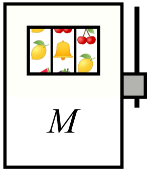
In Las Vegas, a *one-armed bandit* is a slot machine with one. A k-armed bandit is k levers, each of which gives a sequence of rewards according to an unknown probability distribution. Problem: which arm should agent pull next?

Many practical applications:

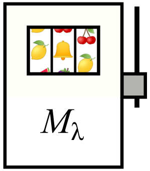
- deciding between k treatments to cure a disease,
- deciding between k investments,
- deciding between k research projects to fund,
- deciding between k advertisements to show a web page visitor,
- A/B testing.



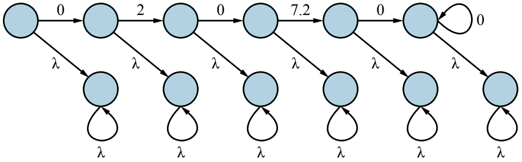
Optimal Bandit Policy via Gittins Index



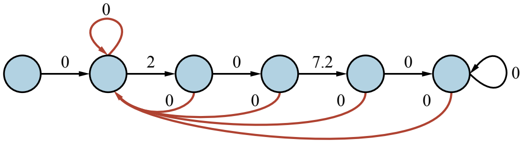
$R_0, R_1, R_2, R_3, R_4, \ldots$



$\lambda, \lambda, \lambda, \lambda, \lambda, \lambda, \lambda, \ldots$

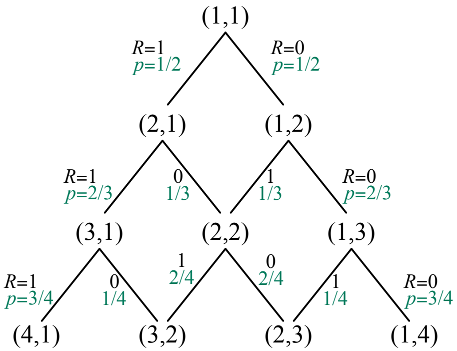


(a)

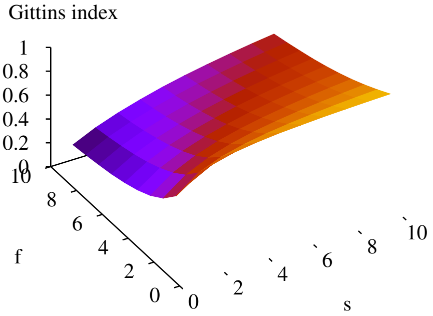


(b)

Bernoulli Bandit



(a)

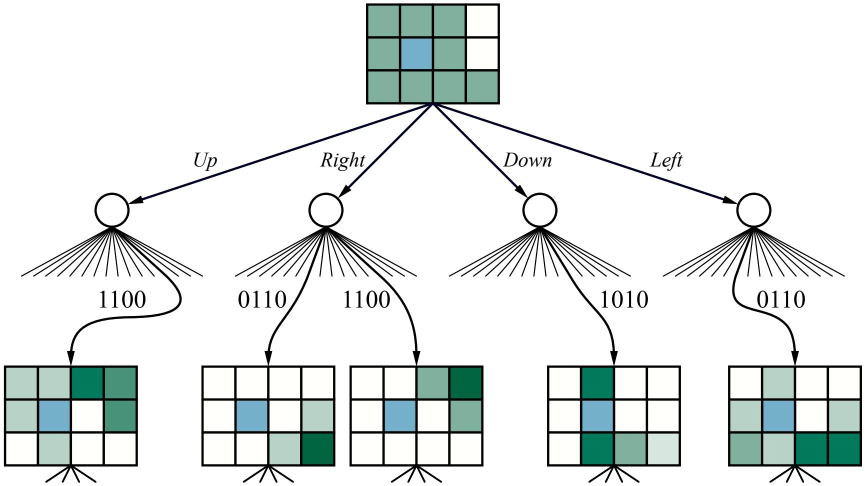


(b)

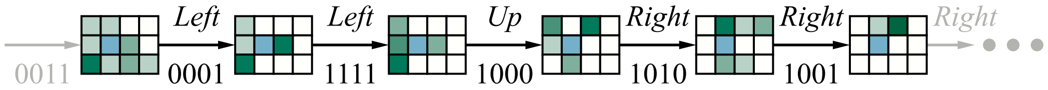
POMDP Value Iteration Algorithm

```
**function** POMDP-VALUE-ITERATION($pomdp, \epsilon$) **returns** a utility function
    **inputs**: $pomdp$, a POMDP with states $S$, actions $A(s)$, transition model $P(s' \mid s, a)$,
                sensor model $P(e \mid s)$, rewards $R(s, a, s')$, discount $\gamma$
            $\epsilon$, the maximum error allowed in the utility of any state
    **local variables**: $U$, $U'$, sets of plans $p$ with associated utility vectors $\alpha_p$

    $U' \leftarrow$ a set containing all one-step plans $[a]$, with $\alpha_{[a]}(s) = \sum_{s'} P(s' \mid s, a)\, R(s, a, s')$
    **repeat**
        $U \leftarrow U'$
        $U' \leftarrow$ the set of all plans consisting of an action and, for each possible next percept,
            a plan in $U$ with utility vectors computed according to Equation (16.18)
        $U' \leftarrow$ REMOVE-DOMINATED-PLANS($U'$)
    **until** MAX-DIFFERENCE$(U, U') \leq \epsilon(1-\gamma)/\gamma$
    **return** $U$
```

Online POMDP Algorithms



POMDP Belief Updates


0011
Left
0001
Left
1111
Up
1000
Right
1010
Right
1001
Right