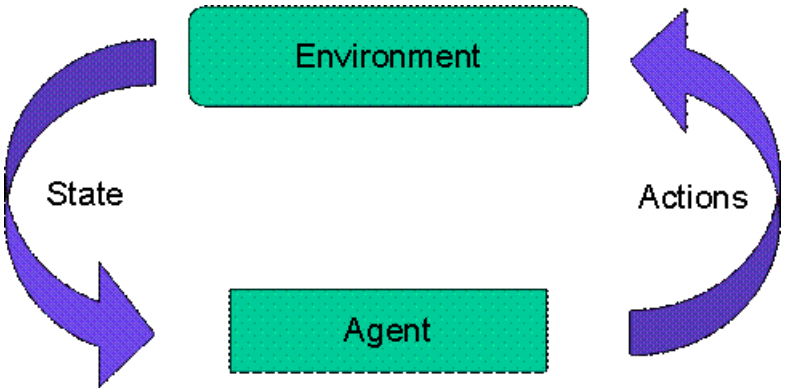
Markov Decision Processes

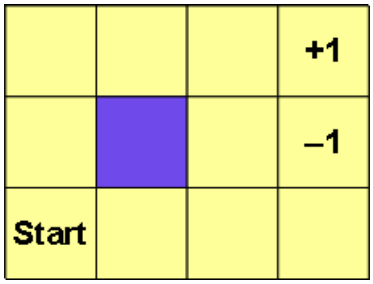
The following is based on material extracted from Barto, Bradtke, and Singh, "Learning to Act Using Real-Time Dynamic Programming," *Artificial Intelligence*, 72 (1): 81–138, 1995. This presentation focuses on the formulation of the original control problem by Richard Bellman and his approaches to solving them. Specifically, the Markov Decision Process was the basis for the development of the dynamic programming method and can be mapped to almost every dynamic programming problem.

Agent-Based Systems

At the foundation of the Markov Decision Process is the desire to solve control and planning problems for autonomous agents. An agent is defined to be a system (whether or hardware, software, biological, or whatever) that perceives the state of its environment and acts upon those perceptions in an attempt to achieve some goal. This definition is depicted graphically as follows:



For example, suppose we have a robot that want to navigate a simple environment defined by 12 blocks:



In this environment, the starting location for the robot is indicated by "Start", and two ending locations are specified by the values $+1$ and -1. These values correspond to a "payoff" that the robot will receive when it reaches either location. All other locations have no return, but movement incurs a fixed cost (say -0.1). The blue square in the center of the environment indicates an obstacle that cannot be occupied by the robot. The robot is able to move up, down, left, or right, and if the robot bumps into a wall or obstacle, it stays in the original location. The object is for the robot to find a path from Start to one of the terminal states and maximize its return (or equivalently minimize its loss).

Two different versions of this problem can be presented. The first is virtually identical to the shortest path problem we saw last week when considering graph algorithms. In this "deterministic" version of the problem, all of the actions yield the expected behavior. In other words, if the robot specifies that it wants to go up, it will go up (as long as it does not hit a wall), and so on for all of the available actions. The second, "nondeterministic" version is the same as the deterministic, except there is noise in the actions. For example, we might find that when a direction is specified, the robot succeeds in moving that direction only 80% of the time. It would move to the left of what was expected 10% of the time and to the right of what was expected 10% of the time. Such nondeterminism makes finding an optimal strategy more difficult than simply applying the greedy approach from Dijkstra's algorithm.

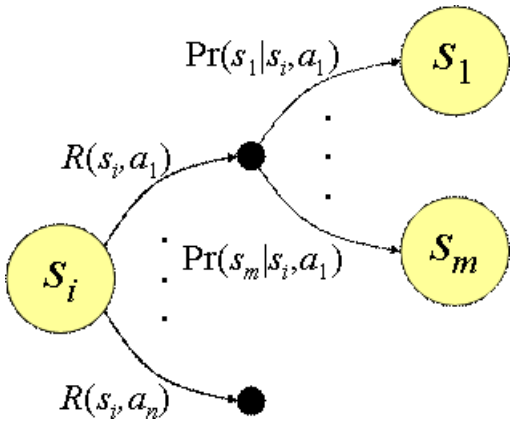
Definition of an MDP

A Markov Decision Process (MDP) is defined formally as follows:

Definition: $M = \langle S, A, T, R, \gamma \rangle$ is a Markov Decision Process where

- $S$is a finite set of states of the environment,
- $A$is a finite set of actions available to the agent,
- $T : S \times A \rightarrow \pi(S)$ is a state transition function where $T(s, a, s') = P(s'|s, a)$ is the probability of transitioning to state s' given the agent takes action a in state s.
- $R : S \times A \rightarrow \Re$ is a reward function where $R(s, a)$ is the expected reward associated with taking action a in state s.
- $\gamma \in [0, 1]$ is a discount factor.

This definition can be shown graphically as follows:



It is interesting to note that the assumptions of S and A being finite can be relaxed. In fact, solving MDPs with infinite state and action spaces is an active area of research.

Definition: A policy $\pi : S \rightarrow A$ is a specification of how an agent will act in all states. A policy can be either stationary or nonstationary.

Definition: A stationary policy is a policy that specifies for each step an action to be taken, independent of the time step in which the state is encountered.

Definition: A nonstationary policy is a policy that specifies a sequence of state-action mappings, indexed by time. Specifically, given $\delta = \langle \pi_t, \ldots, \pi_{t+\tau} \rangle$, π_i is used to choose an action in step i as a function of that state.

Determining the Value of a Policy

Suppose we are presented with a model of an MDP and an associated stationary policy. Then the problem we want to solve is to determine the value of that policy with respect to the MDP. In other words, we want to know what the expected, long run return is if we follow that policy from some state. To find this, let $V^*(s)$ be the expected, discounted, future reward when starting in state s and following stationary policy π. We use the discount factor to ensure this value is well defined; otherwise, an infinite path would yield an infinite cost or reward. Given this, we can find the infinite horizon value of the policy from s as follows:

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

Interpreting, we are saying that the value of state s when following policy can be determined by adding the expected value of the successor state to the reward received when taking action $\pi(s)$ in state s. This is an expected value because the transitions are nondeterministic.

On the surface, this does not appear to help. We have a recurrence that needs to be solved, except we can observe that we can construct a system of |S| linear equations. The only unknowns are the values $V(s)$; therefore, given these equations, we can solve for $V(s)$ using Gaussian elimination or some similar method of solving systems of simultaneous equations.

Finding an Optimal Policy

Suppose now we are presented with an MDP and know the value of its value function based on some unknown optimal policy. Now the problem becomes extracting the policy from this information. As it turns out, this is a relatively simple thing to do. Specifically, the optimal policy involves applying a greedy choice to the value function, namely

$$\pi(s) = \arg\max_{\forall a \in A} \left[R(s,a) + \gamma \sum_{\forall s' \in S} T(s,a,s')V(s') \right].$$

Solving MDPs

Each of the previous two problems had straightforward solutions that did not require dynamic programming. However, suppose we are presented with a situation where we are given the model but do not know either the optimal value function (thus we cannot extract the policy) or the optimal policy (thus we cannot extract the value function). Is there a way to process this model to find both? The answer is "yes," and that is where dynamic programming comes in. In fact, it can be shown that, for any infinite-horizon, discounted MDP as defined here, there exists a stationary policy that is optimal for every state in the MDP. The value function (i.e., the Bellman equation) for finding that policy is as follows:

$$V^*(s) = \max_{\forall a \in A} \left[R(s,a) + \gamma \sum_{\forall s' \in S} T(s,a,s')V^*(s') \right].$$

As we can see, this equation is very similar to the one used to extract the policy, and once we find V, we will be able to use that approach to extract the policy. However, we still need to find V. Unfortunately, we cannot use Gaussian elimination because these equations are no longer linear. To solve, let $V^*(i)$ denote the expected value of the infinite-horizon, discounted reward that will accrue from initial state i given policy π. Let $E_\pi[\cdot]$ denote the expectation over policy π, and let us define $V^*(i)$ as follows:

$$V^*(i) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_t | s_0 = i \right].$$

Now let's define an intermediate function to accumulate values into a Q table (which will be a cost table for dynamic programming) to approximate V. This can be done as follows:

$$Q^V(i,a) = R(i,a) + \gamma \sum_{\forall j \in S} T(i,a,j)V(j).$$

Q represents our approximation of the value of taking action a in state i and then acting optimally thereafter. (As an aside, it is interesting to know that if our MDP is deterministic, the above equation reduces to $Q^V(i,a) = R(i,a) + \gamma V(j)$.) Since the optimal policy is one that is greedy with respect to the value function, we can rewrite the Bellman equation given above in terms of Q as follows:

$$\begin{aligned} V(i) &= \max_{\forall a \in A} \left[R(i,a) + \gamma \sum_{\forall j \in S} T(i,a,j)V(j) \right] \\ &= \max_{\forall a \in A} Q^V(i,a). \end{aligned}$$

Given this definition, we are now in a position to define two different dynamic programming algorithms—value iteration and policy iteration.

Value Iteration

We will compute a sequence V_t using the auxiliary Q function, but we will denote the Q function as $Q_t(s, a)$ to make explicit the relationship between the time step of the algorithm (i.e., the iteration through the loop) and the states and actions in the MDP. Since these Q values change with each step of the algorithm (and so do the corresponding V values), we will be defining a nonstationary policy until convergence. Value iteration will terminate when the maximum difference between two successive iterations drops below a predefined threshold ϵ, known as the Bellman error magnitude. The pseudocode for value iteration is as follows:

Algorithm 1 Value Iteration

```
ValueIteration(MDP, ε)
    for each s ∈ S do
        V_0 ← 0
    t ← 0
    repeat
        t ← t + 1
        for each s ∈ S do
            for each a ∈ A do
                Q_t(s, a) ← R(s, a) + γ Σ_{s'∈S} T(s, a, s') V_{t−1}(s')
            π_t(s) ← arg max_{a∈A} Q_t(s, a)
            V_t(s) ← Q_t(s, π_t(s))
    until max_s |V_t(s) − V_{t−1}(s)| < ε
    return π_t
```

The first thing that happens is that all values of V are initialized to zero. Then the main loop is run and continues until the termination criterion described above is met. For this algorithm, every state-action pair is considered and updated using the equation $Q_t(s, a) = R(s, a) + \gamma \sum_{\forall s' \in S} T(s, a, s') V_{t-1}(s')$. After the Q values are updated, the policy is determined by applying the greedy choice $\pi_t(s) = \arg\max_{\forall a \in A} Q_t(s, a)$, and this is used to determine the new value for $V_t(s)$.

Policy Iteration

Policy iteration is similar to value iteration in that the intermediate Q function is the basis for the updates; however, this approach uses the two algorithms presented at the beginning of this section (deriving a value function from a policy and deriving a policy given a value function) more directly.

Let π_0 be the greedy policy for the initial value function V_0. Let $V_0 = V^*$ be the value function for π_0. If we alternate between finding the optimal policy for a particular value function and the corresponding value function for the new policy, we eventually converge on the optimal policy and value function for the MDP. This time, since the value function is derived directly from a policy, the termination criterion for the algorithm is stronger. We terminate when there is no change in the value function because there is no change in the policy. The pseudocode for policy iteration is as follows:

Algorithm 2 Policy Iteration

```
PolicyIteration(MDP)
   for each s ∈ S do
      V_0 ← 0
   t ← 0
   repeat
      t ← t + 1
      for each s ∈ S do
         for each a ∈ A do
            Q_t(s,a) ← R(s,a) + γ Σ_{s'∈S} T(s,a,s') V_{t−1}(s')
         π_t(s) ← arg max_{a∈A} Q_t(s,a)
         V_t(s) ← EvalMDP(π_t, MDP)
   until V_t(s) = V_{t−1}(s) for all s ∈ S
   return π_t
```

The primary difference between value iteration and policy iteration lies in the line $V_t(s) \leftarrow \text{EvalMDP}(\pi_t, MDP)$. This function solves the system of simultaneous equations to determine V directly from the policy.

Convergence of Synchronous Dynamic Programming

Both value iteration and policy iteration are called "synchronous" dynamic programming because updates at time t are all based on values from time $t-1$. Determining the complexity of synchronous dynamic programming (whether policy iteration or value iteration) is difficult for MDPs in that the analysis depends on determining when the process converges. We note that, given n states and m actions, each iteration of the algorithm will require $O(mn^2)$ operations. Policy iteration then has the expense associated with solving the system of simultaneous equations; however, since each $V(s)$ is defined in terms of only one $V(s')$, solving the equations can be done in linear time. In fact, this process is called "backing up" since we effectively back up values through the stages of the MDP, one stage at a time with each iteration.

For convergence, we find that synchronous dynamic programming converges to the optimal V^* in undiscounted stochastic MDPs (i.e., when $\gamma = 1$) under the following conditions:

- The initial cost of every goal state is zero.
- There exists at least one proper policy.
- All policies that are not proper incur infinite cost for at least one state.

Convergence results with similar conditions apply for discounted MDPs as well.