CMSC 726
Lecture 24:Reinforcement Learning – Part I

Lise Getoor
November 30, 2010

ACKNOWLEDGEMENTS: The material in this course is a synthesis of materials from many sources, including: Hal Daume III, Mark Drezde, Carlos Guestrin, **Andrew Ng**, Ben Taskar, **Eric Xing**, and others. I am very grateful for their generous sharing of insights and materials.

Outline

- Intro to reinforcement learning
- MDP: Markov decision problem
- Dynamic programming:
  - Value iteration
  - Policy iteration

What is Learning?

- Learning takes place as a result of interaction between an agent and the world, the idea behind learning is that
  - Percepts received by an agent should be used not only for understanding/interpreting/prediction, as in the machine learning tasks we have addressed so far,
    but also for **acting**, and further more for improving the agent's ability to behave optimally in the future to achieve the goal.

Types of Learning

- Supervised Learning
  - A situation in which sample (input, output) pairs of the function to be learned can be perceived or are given
  - You can think it as if there is a kind teacher
    - Training data: (X,Y). (features, label)
    - Predict Y, minimizing some loss.
    - Regression, Classification.

- Unsupervised Learning
    - Training data: X. (features only)
    - Find "similar" points in high-dim X-space.
    - Clustering.

Types of Learning (Cont'd)

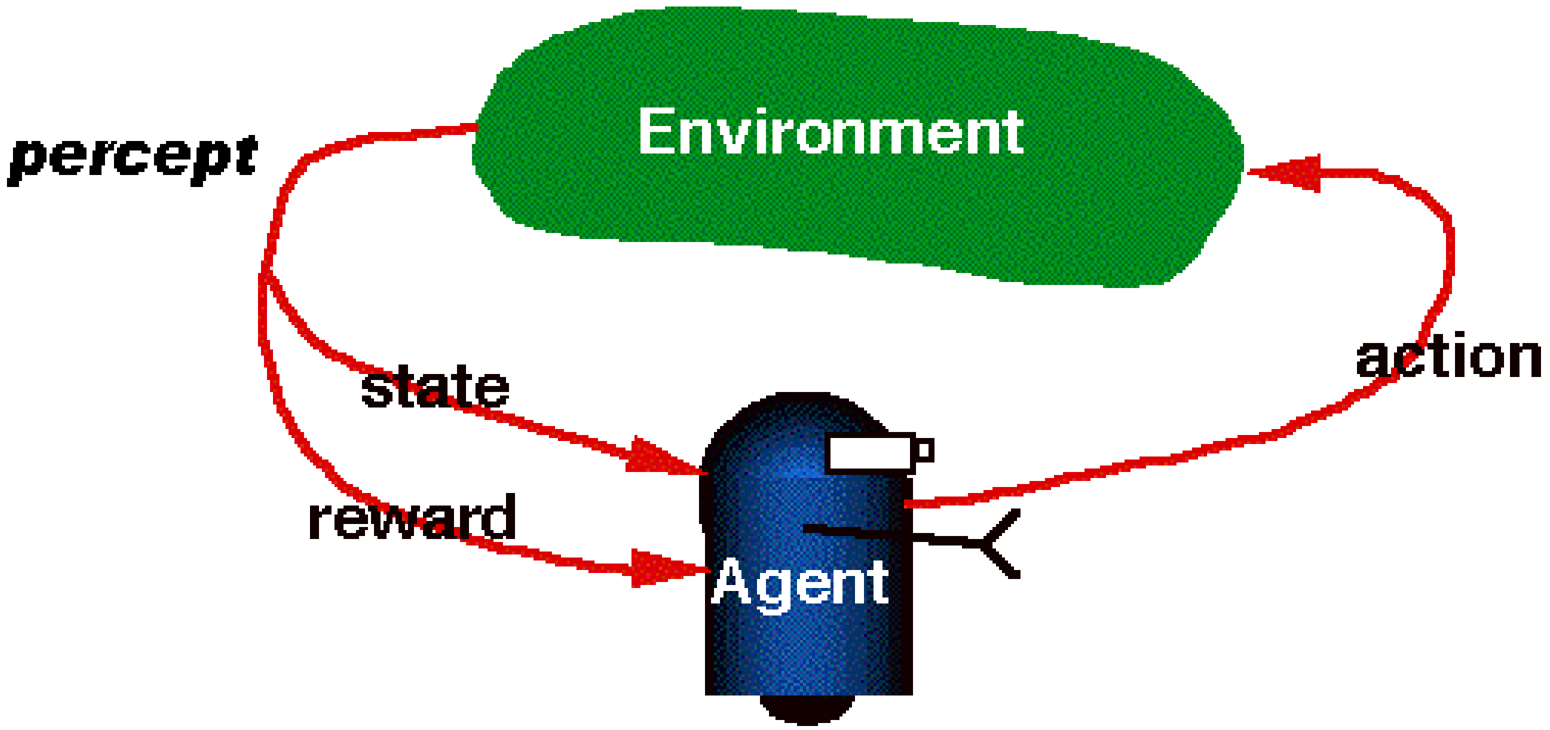
- Reinforcement Learning
  - the agent acts on its environment, it receives some evaluation of its action (reinforcement), but is not told of which action is the correct one to achieve its goal

    – Training data: (S, A, R). (State–Action–Reward)

    – Develop an optimal policy (sequence of decision rules) for the learner so as to maximize its long-term reward.

    – Robotics, Board game playing programs.

RL is learning from interaction



Examples of Reinforcement Learning

- How should a robot behave so as to optimize its "performance"? (Robotics)

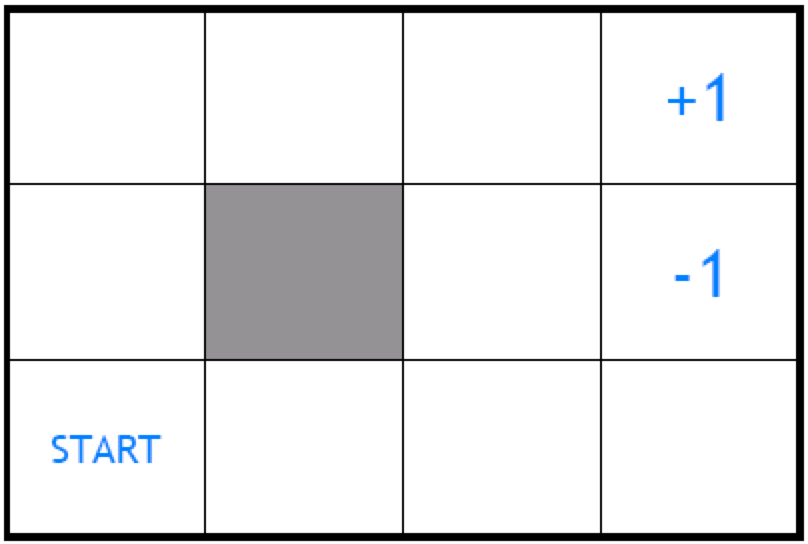
- How to automate the motion of a helicopter? (Control Theory)

- How to make a good chess-playing program? (Artificial Intelligence)

History of Reinforcement Learning

- Roots in the psychology of animal learning (Thorndike,1911).
- Another independent thread was the problem of optimal control, and its solution using dynamic programming (Bellman, 1957).
- Idea of temporal difference learning (on-line method), e.g., playing board games (Samuel, 1959).
- A major breakthrough was the discovery of Q-learning (Watkins, 1989).

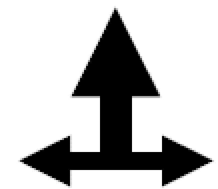
Robot in a room



actions: UP, DOWN, LEFT, RIGHT

UP

80%	move UP
10%	move LEFT
10%	move RIGHT



- reward +1 at [4,3], -1 at [4,2]
- reward -0.04 for each step

what's the strategy to achieve max reward?
what if the actions were deterministic?

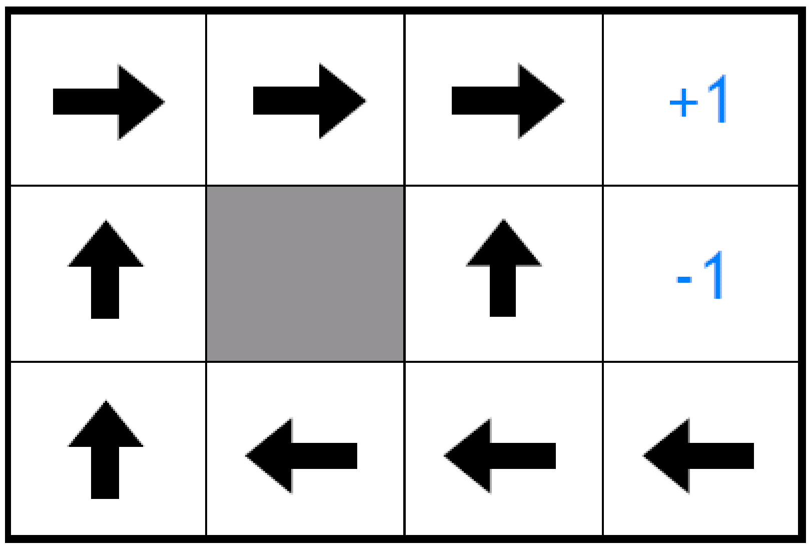
What is special about RL?

- RL is learning how to map states to actions, so as to maximize a numerical reward over time.
- Unlike other forms of learning, it is a multistage decision-making process (often Markovian).
- An RL agent must learn by trial-and-error. (Not entirely supervised, but interactive)
- Actions may affect not only the immediate reward but also subsequent rewards (Delayed effect).

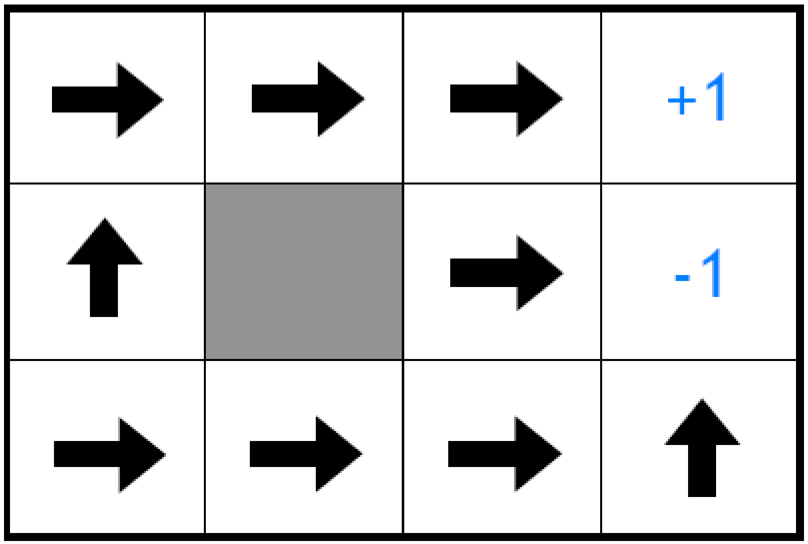
Elements of RL

- A policy
  - A map from state space to action space.
  - May be stochastic.
- A reward function
  - It maps each state (or, state-action pair) to a real number, called reward.
- A value function
  - Value of a state (or, state-action pair) is the total expected reward, starting from that state (or, state-action pair).

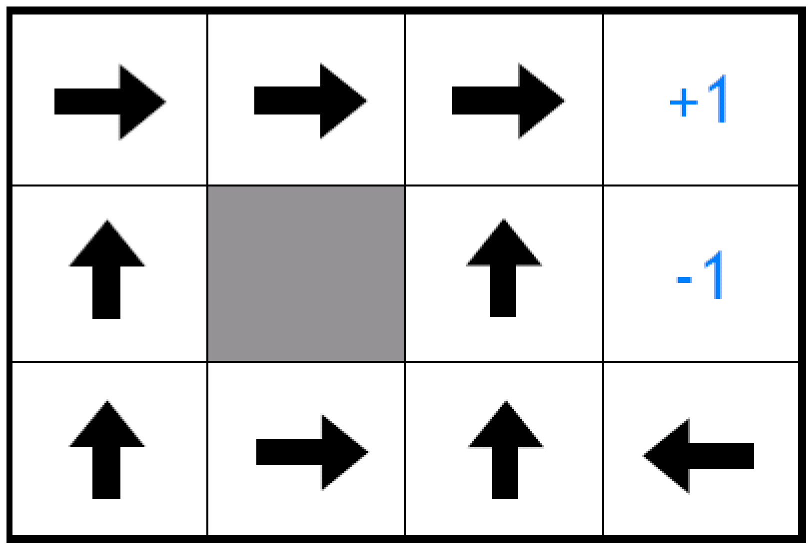
Policy



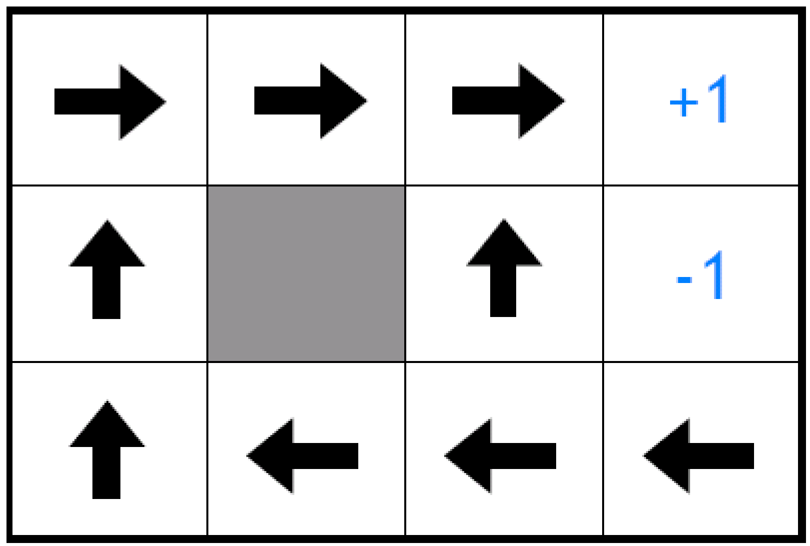
Reward for each step -2



Reward for each step: -0.1



Reward for each step: −0.04



The Precise Goal

- To find a policy that maximizes the Value function.
  - transitions and rewards usually not available
- There are different approaches to achieve this goal in various situations.
- Value iteration and Policy iteration are two more classic approaches to this problem. But essentially both are dynamic programming.
- Q-learning is a more recent approaches to this problem. Essentially it is a temporal-difference method.

Markov Decision Processes

A Markov decision process is a tuple $(S, A, \{P_{sa}\}, \gamma, R)$ where:

- S is a set of **states**. (For example, in autonomous helicopter flight, S might be the set of all possible positions and orientations of the helicopter.)

- A is a set of **actions**. (For example, the set of all possible directions in which you can push the helicopter's control sticks.)

- P_{sa} are the state transition probabilities. For each state $s \in S$ and action $a \in A$, P_{sa} is a distribution over the state space. We'll say more about this later, but briely, P_{sa} gives the distribution over what states we will transition to if we take action a in state s.

- $\gamma \in [0, 1)$ is called the **discount factor**.

- $R : S \times A \mapsto \mathbb{R}$ is the **reward function**. (Rewards are sometimes also written as a function of a state S only, in which case we would have $R : S \mapsto \mathbb{R}$).

The dynamics of an MDP

- We start in some state s_0, and get to choose some action $a_0 \in A$
- As a result of our choice, the state of the MDP randomly transitions to some successor state s_1, drawn according to $s_1 \sim P_{s0a0}$
- Then, we get to pick another action a_1
- …

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

The dynamics of an MDP, (Cont'd)

- Upon visiting the sequence of states s_0, s_0, …, with actions a_0, a_0, …, our total payoff is given by

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots$$

- Or, when we are writing rewards as a function of the states only, this becomes $$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$
  - For most of our development, we will use the simpler state-rewards R(s), though the generalization to state-action rewards R(s; a) offers no special diculties.
- Our goal in reinforcement learning is to choose actions over time so as to maximize the expected value of the total payoff:

$$\mathrm{E}\big[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots\big]$$

Policy

- A policy is any function $\pi : S \mapsto A$ mapping from the states to the actions.
- We say that we are executing some policy if, whenever we are in state s, we take action $a = \pi(s)$.
- We also define the value function for a policy π according to

$$V^{\pi}(s) = \mathrm{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots \mid s_0 = s, \pi\right]$$

  - $V^{\pi}(s)$ is simply the expected sum of discounted rewards upon starting in state s, and taking actions according to π.

Value Function

- Given a fixed policy π, its value function V^π satisfies the **Bellman equations**:

$$V^\pi(s) = \underbrace{R(s)}_{\text{Immediate reward}} + \underbrace{\gamma \sum_{s' \in \mathcal{S}} P_{s\pi(s)}(s') V^\pi(s')}_{\text{expected sum of future discounted rewards}}$$

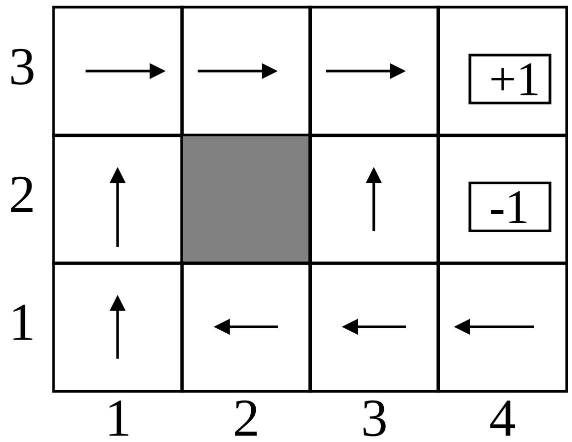
  - Bellman's equations can be used to efficiently solve for V^π (see later)

The Grid world

M = 0.8 in direction you want to go
0.2 in perpendicular < 0.1 left, 0.1 right

Policy: mapping from states to actions

An optimal policy for the stochastic environment:



utilities of states:

	1	2	3	4
3	0.812	0.868	0.912	+1
2	0.762		0.660	-1
1	0.705	0.655	0.611	0.388

Environment < Observable (accessible): percept identifies the state; Partially observable

Markov property: Transition probabilities depend on state only, not on the path to the state.

Markov decision problem (MDP).

Partially observable MDP (POMDP): percepts do not have enough info to identify transition probabilities.

Optimal value function

- We define the optimal value function according to

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{1}$$

  - In other words, this is the best possible expected sum of discounted rewards that can be attained using any policy

- There is a version of Bellman's equations for the optimal value function:

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s') \tag{2}$$

Optimal policy

- We also define a policy : $\pi^* : S \mapsto A$ as follows:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in \mathcal{S}} P_{sa}(s') V^*(s') \qquad (3)$$

- Fact:
  - $$V^*(s) = V^{\pi^*}(s) \geq V^{\pi}(s)$$
  - Policy π^* has property that it is the optimal policy for all states s.
    - It is not the case that if we were starting in some state s, there'd be some optimal policy for that state, and if starting in some other state s_0 then there'd be some other optimal policy for s_0.
    - The same policy π^* attains the maximum above for all states s. This means that we can use the same policy no matter what the initial state of our MDP is.

Algorithm 1: Value iteration

- Consider only MDPs with finite state and action spaces

  $(|S| < \infty, \ |A| < \infty)$

- The value iteration algorithm:

  1. For each state s, initialize $V(s) := 0$.

  2. Repeat until convergence $\{$

     - For every state, update
       $V(s) := R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s')$.

     $\}$

  - **synchronous** update
  - **asynchronous** updates

- It can be shown that value iteration will cause V to converge to V^*. Having found V , we can then use Equation (3) to find the optimal policy.

Algorithm 2: Policy iteration

- The policy iteration algorithm:

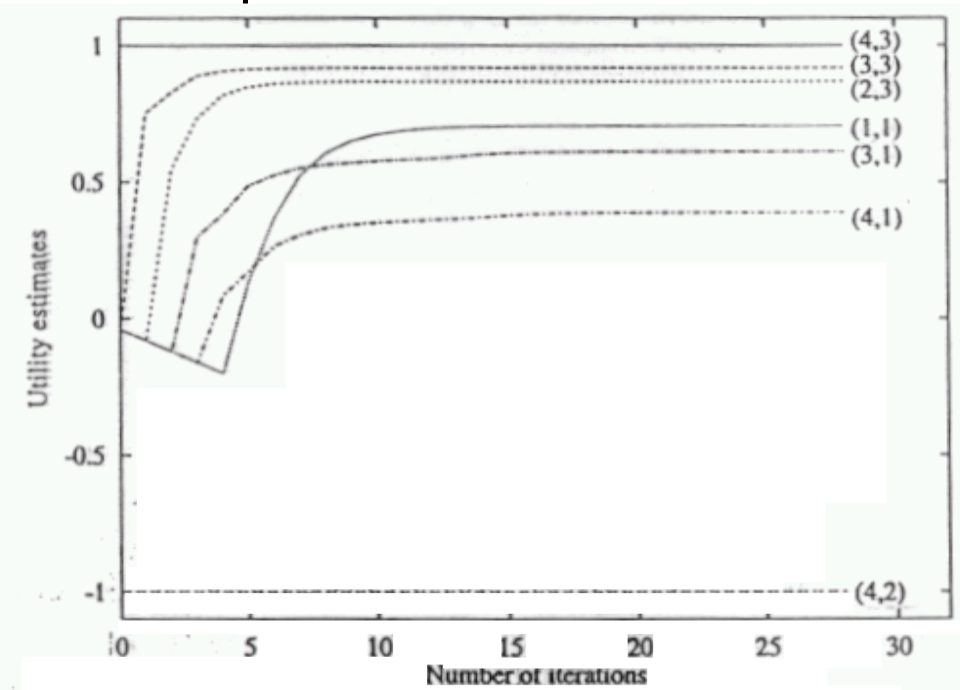
  1. Initialize π randomly.
  2. Repeat until convergence $\{$
     - Let $V := V^{\pi}$
     - For each state s, let $\pi(s) := \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V(s')$.

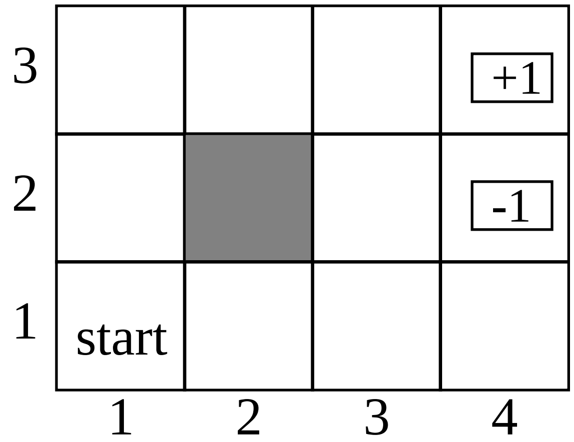
     $\}$

  - The inner-loop repeatedly computes the value function for the current policy, and then updates the policy using the current value function.
  - Greedy update
  - After a finite number of iterations of this algorithm, V will converge to V^* , and π will converge to π^*.

Convergence

- The utility values for selected states at each iteration step in the application of VALUE-ITERATION to the 4x3 world in our example





Thrm: As t→∞, value iteration converges to exact U even if updates are done asynchronously & i is picked randomly at every step.

Convergence

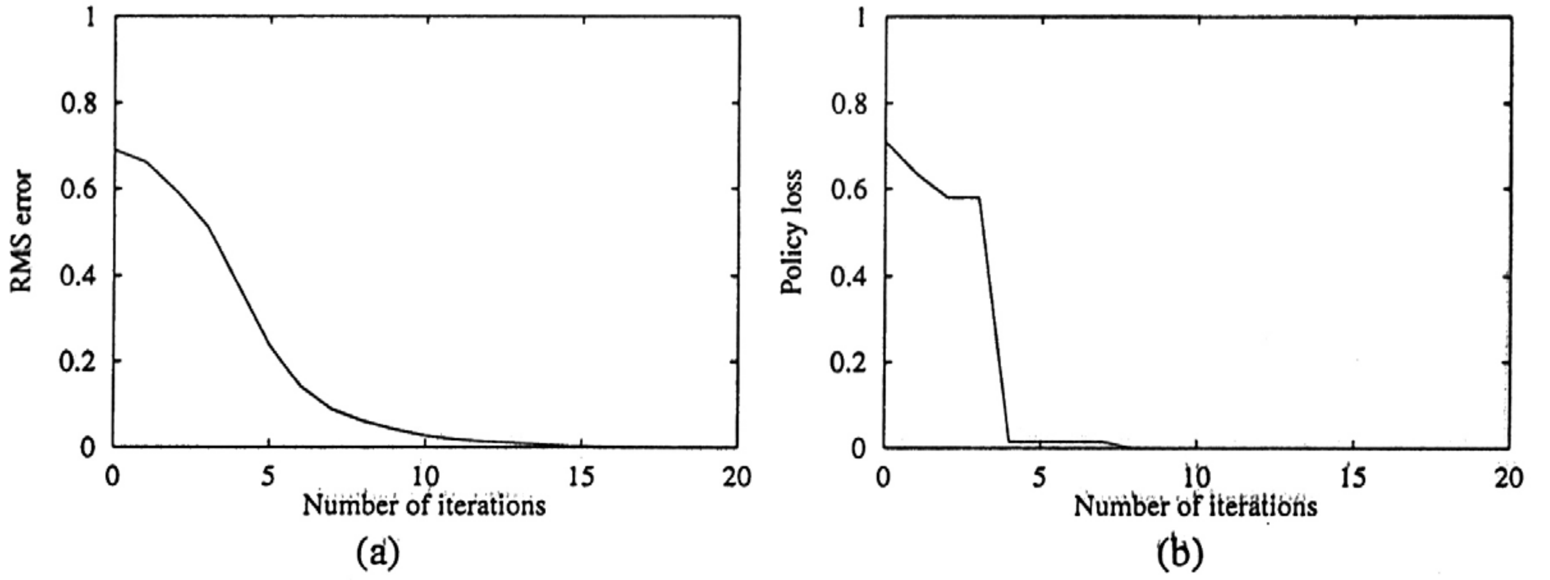


Figure 17.6 (a) The RMS (root mean square) error of the utility estimates compared to the correct values, as a function of iteration number during value iteration. (b) The expected policy loss compared to the optimal policy.

When to stop value iteration?

Summary

- Both value iteration and policy iteration are standard algorithms for solving MDPs, and there isn't currently universal agreement over which algorithm is better.
- For small MDPs, policy iteration is often very fast and converges with very few iterations. However, for MDPs with large state spaces, solving for V explicitly would involve solving a large system of linear equations, and could be difficult.
- In these problems, value iteration may be preferred. For this reason, in practice value iteration seems to be used more often than policy iteration.