

Reinforcement Learning

Ref: Chapter 21

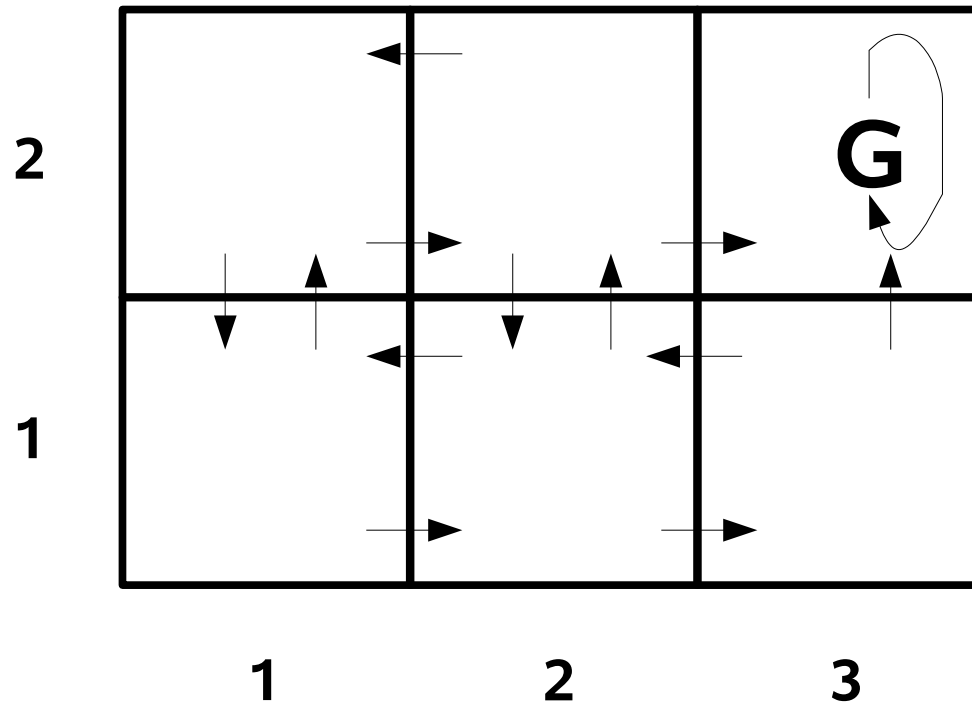
Learning without an oracle

- Agent is alone in some *world*
 - the *world* is some set of states.
 - a number of actions are possible in each state.
 - some *reward* is provided to the agent after taking an action.
- The goal is to learn an optimal *control policy*
 - for each state, what is the best action to take to maximize the cumulative reward collected (over some possibly infinite time period).

Markov Decision Process

- We generally make a Markov assumption:
 - the best action to take in any state is independent of how the agent arrived at that state.
 - the problem is still hard, but much simpler if we can make this assumption.
 - many real-life reinforcement learning problems are clearly Markov:
 - most games.
 - robot control systems
 - factory scheduling

An Example World



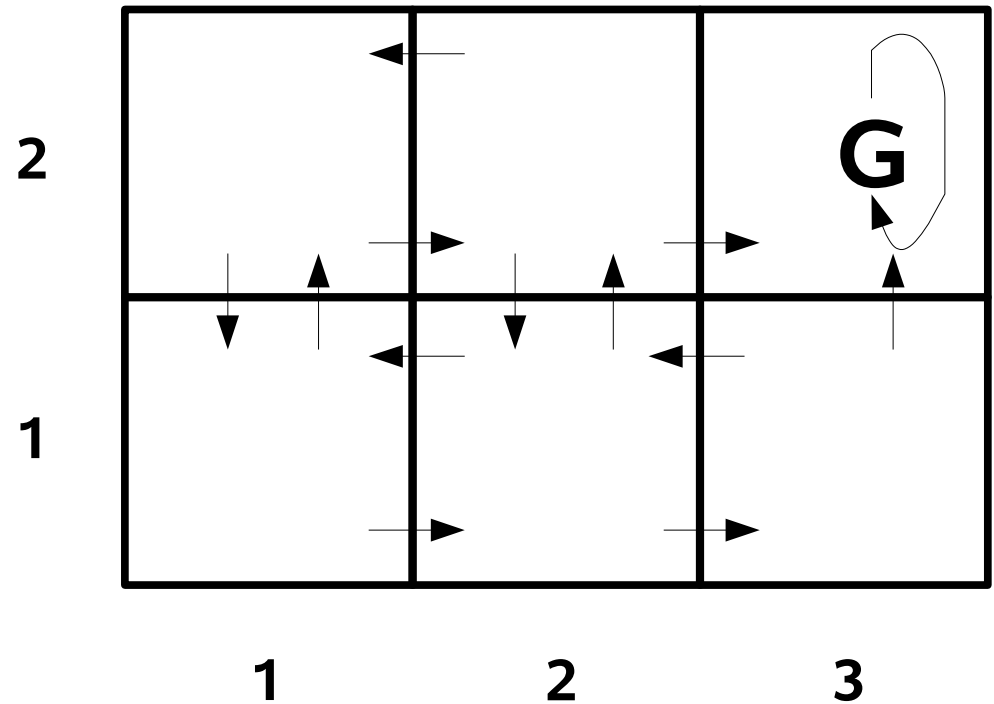
State 2,3 is the goal state (an absorbing state).

The reward for entering state 2,3 is 100.

The reward for entering all other states is 0.

The arrows show the possible actions.

Issues



- Reward is delayed.
 - *credit assignment problem.*
- Initially the agent knows nothing.
- We want the agent to *learn* an optimal *policy*.

Formalization

S: set of states.

A: set of actions

At time t the agent is in state s_t (and knows this).

The agent selects action a_t

The environment gives the agent a reward $r(s_t, a_t)$

The agent moves to state $s_{t+1} = \delta(s_t, a_t)$

↑
transition function

↑
reward function

MDP

We are assuming that both the reward function

$$r(s_t, a_t)$$

and the transition function

$$\delta(s_t, a_t)$$

depend on the current state (not on previous states).

For now we assume both functions are deterministic.

A Policy

- The task is to have the agent learn an optimal *policy*.
- Policy $\pi: S \rightarrow A$
- A policy is a set of rules that the agent uses to decide what action to take in each state.
- We want to select a policy that maximizes the cumulative reward received from the current state into the future.
 - this is independent of how we got to the current state or how much reward we have collected so far!

The goal

$$V^\pi(s_t) = r_t + r_{t+1} + r_{t+2} + \dots$$

$V^\pi(s_t)$ is the sum of the rewards obtained when following policy π starting at state s_t

More General Goal

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

$$0 \leq \gamma \leq 1$$

$V^\pi(s_t)$ is the sum of the *discounted* rewards obtained when following policy π starting at state s_t

The general idea is that sometimes immediate rewards are more important than future rewards (γ is used to discount future rewards).

Learning is a search

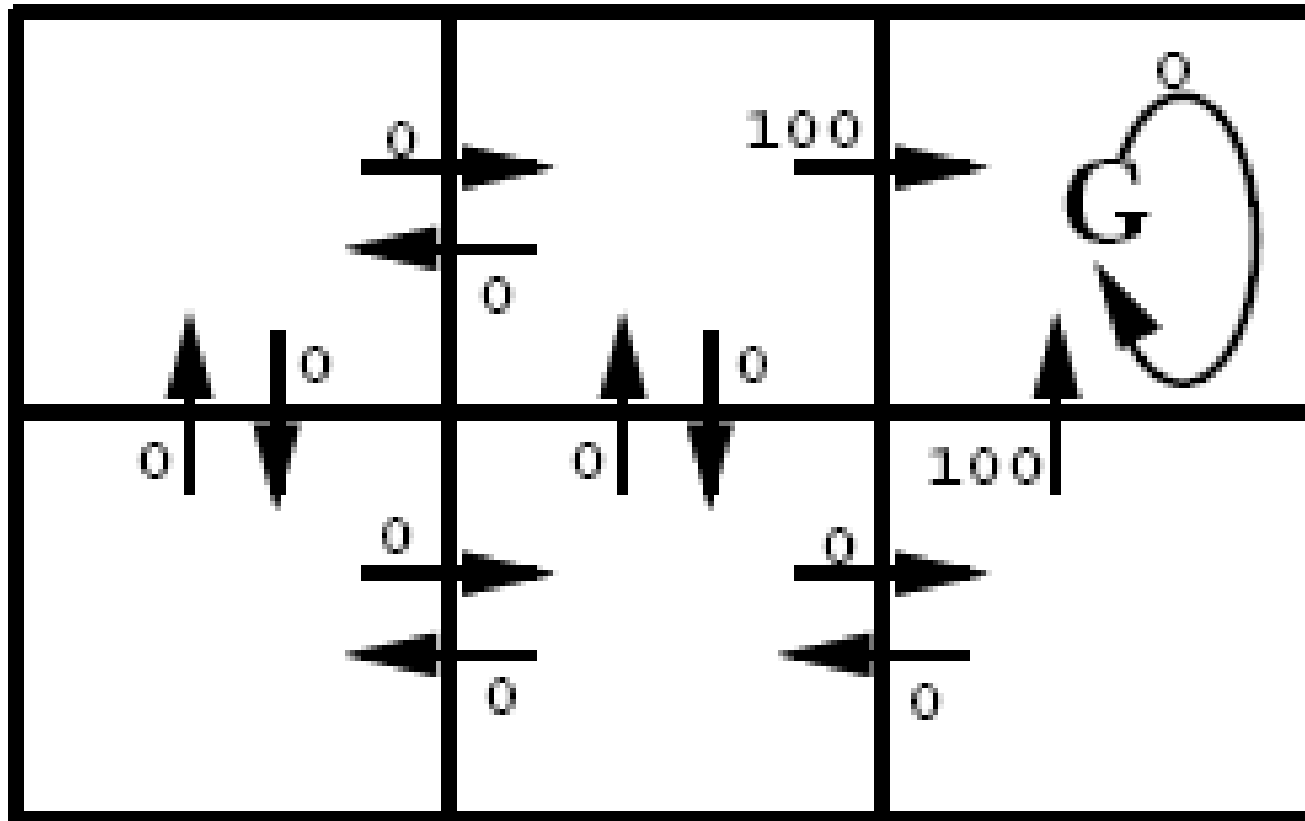
Our agent needs to search for a policy π^* that maximized the (possibly discounted) cumulative reward.

$$\pi^* = \operatorname{argmax}_{\pi} V^{\pi}(s_t), (\forall s)$$

The policy π which maximizes the cumulative sum V^{π} over all states.

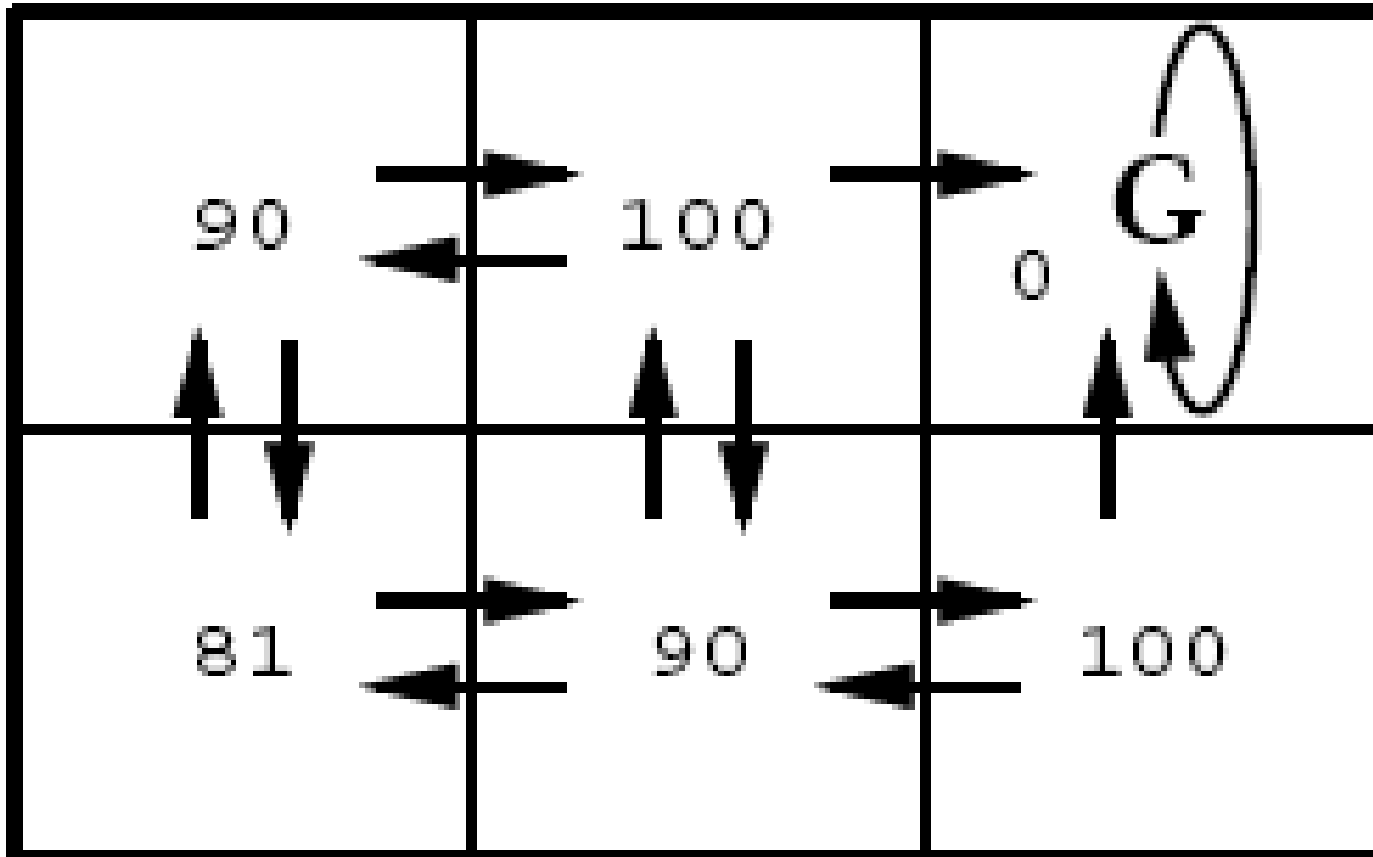
The set of rules that when used, result in the maximum reward from any state.

Immediate Rewards

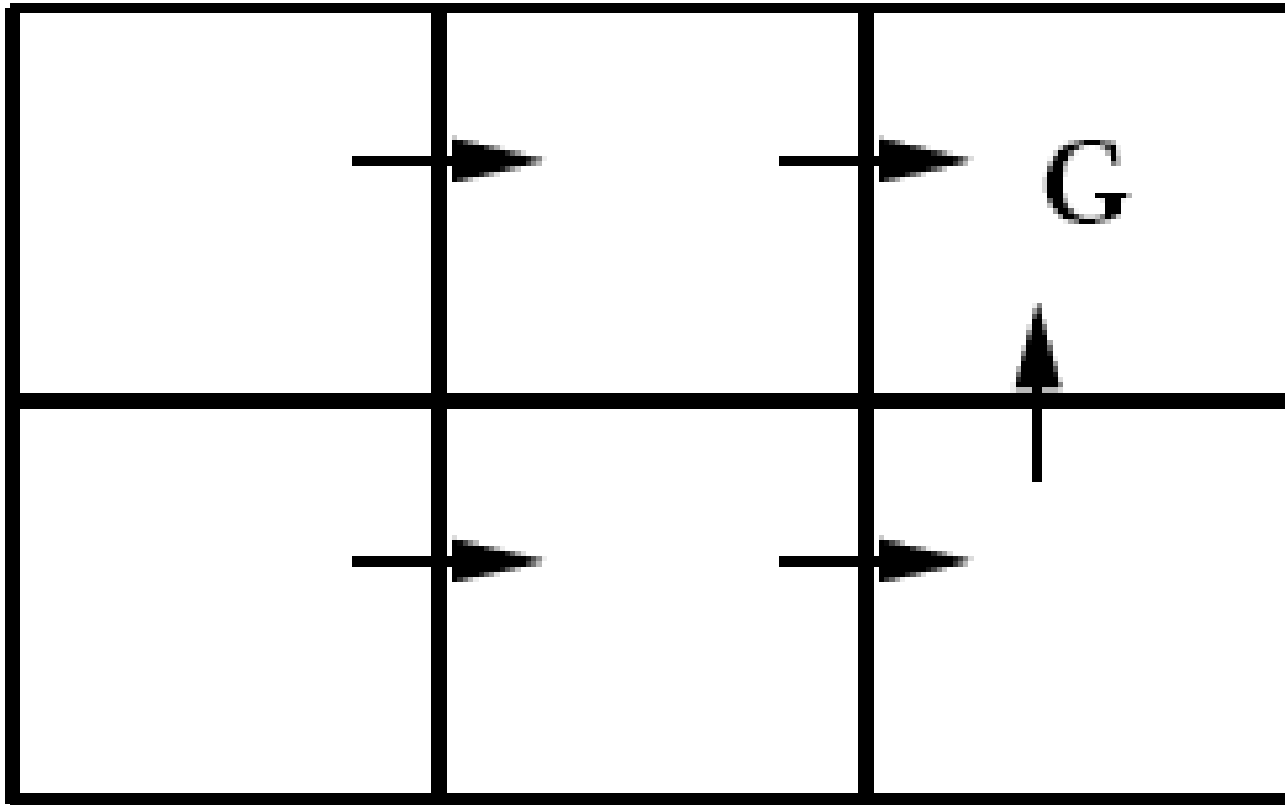


Cumulative Rewards $V^*(s_t)$

$$\gamma = 0.9$$



An Optimal Policy



What to learn?

- The agent could try to learn $V^*(s_t)$
 - the maximum cumulative reward from state s_t

$$\pi^* = \operatorname{argmax}_a [r(s, a) + \gamma V^*(\delta(s, a))]$$

The policy is: the agent picks that action a for which the immediate reward + cumulative reward for the next state is maximum.

the agent needs to know r and δ

How to learn?

- Since we are assuming the agent knows r and δ , it can simply compute the cumulative payoff for each state.
 - basically it can evaluate all possible policies without ever actually taking any actions.
 - This is generally not possible!

r and δ are not known

Dumb Agent

- Assume the only information that agent can get is the rewards obtained by actually taking actions (moving around, making moves in a game, whatever).
- We can still learn an optimal policy (under the right conditions).
- The algorithm for doing this is call Q-learning

Q-Learning – What to Learn

$$Q(s,a) = r(s,a) + \gamma V^*(\delta(s, a))$$

If the agent can learn Q (instead of V^*), it can still come up with an optimal policy:

$$\pi^* = \operatorname{argmax}_a Q(s,a)$$

a is the action that maximizes Q in state s

Learning Q vs. V^*

- Learning and using V^* sounds easier:
 - pick the action that will lead to the greatest combination of immediate reward + future reward.
- Learning Q :
 - pick the *action* that provides the greatest cumulative reward.
 - a restatement of the problem!
 - no explicit distinction between immediate reward and future reward.

How to learn Q ?

- Iterative approximation
 - agent starts with some initial guess as to the value of Q at each state, for each action.
 - each time an action a is taken from a state s , update the approximation of $Q(s, a)$ based on the reward received and the approximations of Q for the new state.

Update Rule Derivation

$$V^*(s) = \max_{a'} Q(s, a')$$

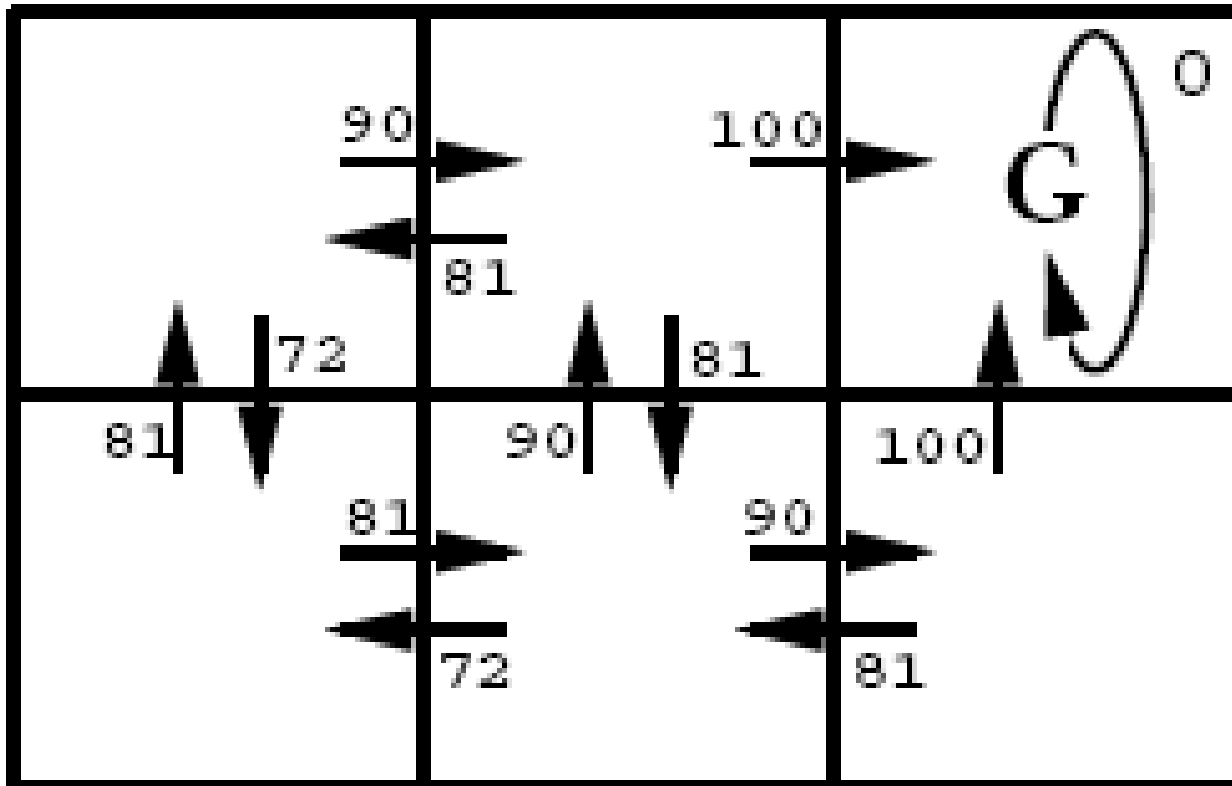
$$Q(s, a') = r(s, a) + \gamma \max_{a'} (\delta(s, a), a')$$

$\hat{Q}(s, a')$ is the approximation of $Q(s, a')$

Update Rule: $\hat{Q}(s, a') \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$

$s' = \delta(s, a)$ (the state reached when applying action a')

Q values : $\gamma = 0.9$



Note that without discounts, all actions would have Q value 100

How to learn?

- The agent will need to visit the states many times in order to accumulate information about the rewards.
 - each time it reaches the goal state, it picks a random state and starts over.
- How will the agent select actions?
 - it could use the best policy found so far?
 - it could pick randomly (while learning)?

Exploitation vs. Exploration

- When learning, the agent clearly needs to visit all the states to come up with accurate estimates for each $Q(s,a)$
- In most circumstances we want the agent to be looking for reward while learning!
- The agent needs to find a balance between exploring lots of states, and exploiting it's current knowledge about the world (to collect reward).

n-Armed Bandit Problems

- You are at a slot machine with n different arms.
- You are told that each arm is different, in that some pay more than others (on average).
 - each arm has some fixed payoff distribution, but some provide better average payoff than others
- Your job is to come up with a strategy that will make the most money in 1000 total pulls.

Simple Strategy

pull each arm $1000/n$ times.

we end up with the average average payoff.

If one arm pays off a lot more than the others, we could have done much better.

Another Possible Strategy

pull each arm once and keep track of the payoffs.

pull the arm that looks best $1000-n$ more times.

This assumes that the first n pulls were a representative sample.

Better strategy?

- Pull each arm once.
- For the remaining $1000-n$ pulls:
 - pull the arm which has shown the highest average payoff seen.
 - we need to constantly update the average payoff seen for each arm.
- If we pick the wrong arm first, eventually it's average payoff will drop below the initial payoff of some other arm (we hope).

Better strategies?

- This a well studied problem, with lots of variations involving lots of solutions.
- In general, the best strategy is to allocate increasing pulls to the best arms.
- For example, with 2 arms:
 - pull each arm 10 times.
 - for the next 980 pulls: pull arms probabilistically, with $p(\text{best}) = .5 + (\# \text{ pulls}) / (980 * 2)$

Q Learning Algorithm

- Set all Q values (one for each action) to 0.
- pick some starting state s
- Do forever
 - pick an action a (for the current state s)
 - take the action, moving to state s' and receive reward r
 - Update $Q(s, a)$ to be $r + \max_{a'} Q(s', a')$
 - $s = s'$

An Example World

3				+100
2				-100
1				
	1	2	3	4

Non deterministic environment

In some situations the functions $r(s,a)$ and $\delta(s, a)$ are not deterministic!

- reward is not fixed, but is drawn from some fixed distribution and is bounded.
- successor state is not fixed, but depends on some distribution.
- We can still learn, but now we need to base all computations on *expected values*. Q learning is still possible (see the book).

More general algorithm: TD(λ)

- Q learning looks ahead one state when adjusting Q estimates.
- We can generalize this to looking ahead multiple states
 - combine the estimates, giving estimates from far away states less weight when updating Q.
- The best backgammon program (TD-Gammon) uses this algorithm.