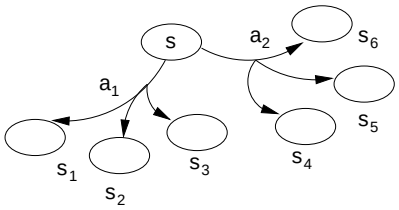
MAKING COMPLEX DECISIONS

Introduction

- Last week we looked at how an agent might make a decision under uncertainty.
- Problem structure:
  - Set of non-deterministic actions.
  - Resulting states have utilities associated with them.
- We use probability + utility theory (= decision theory) to establish the best choice of action.
- This week wee'll look at more complex situations.
- But first we'll recap.

Simple decisions

- Consider an agent with a set of possible actions A available to it.
- Each $a \in A$ has a set of possible outcomes s_a:



- For the outcomes s_a of each action each a, the agent can calculate:

$$E(u(s_a)) = \sum_{s' \in s_a} u(s').P(s_a = s')$$

- A *rational* agent will then choose a^* such that:

$$a^* = \arg\max_{a \in A} \sum_{s' \in s_a} u(s').P(s_a = s')$$

- That is it picks the action that has the greatest expected utility.
- Here "rational" means "rational in the sense of maximising expected utility".
- As we will see, there are other ways to define "rational", but the MEU definition is the most widely accepted at this time.

An example

- As an example, consider an agent which has to choose between tossing a coin, rolling a die, or receiving a payoff of $ 1.
- If the coin is chosen, then the agent gets $1.50 a head and $0.5 for a tail.
- If the die is chosen, the agent gets $5 if a six is rolled, $1 if a two or three is rolled, and nothing otherwise.
- What is the rational choice, assuming that the agent's preferences are (for once) modelled by monetary value?

- Well, we need to calculate the expected outcome of each choice.
- For doing nothing, we have:

  a_1 = "receive payoff",
  s_{a_1} = {get $1},
  $u(\text{get } \$1) = 1$, and
  $\Pr(s_{a_1} = \text{get } \$1) = 1$.
- Thus:

$$E(u(s_{a_1})) = 1$$

- If the coin is chosen, we have a_2 = "coin", $s_{a_2} = \{\text{head}, \text{tail}\}$,

$$\begin{aligned} u(\text{head}) &= \$1.50 \\ u(\text{tail}) &= \$0.5 \end{aligned}$$

and

$$\begin{aligned} \Pr(s_{a_2} &= \text{head}) = 0.5 \\ \Pr(s_{a_2} &= \text{tail}) = 0.5 \end{aligned}$$

- Thus the expected utility is:

$$\begin{aligned} E(u(s_{a_2})) &= 0.5 \times 1.5 + 0.5 \times 0.5 \\ &= 1 \end{aligned}$$

- Action a_3, rolling the die, can be analysed in a similar way, giving:

$$E(u(s_{a_3})) = 1.17$$

- Choosing to roll the die is the rational choice.

Other kinds of "rational"

- There are other criteria for decision-making than maximising expected utility.
- One approach is to look at the option which has the least-bad worst outcome.
- This *maximin* criterion can be formalised in the same framework as MEU, making the rational (in this sense) action:

$$a^* = \arg\max_{a \in A} \{\min_{s' \in s_a} u(s')\}$$

- Its effect is to ignore the probability of outcomes and concentrate on optimising the worst case outcome.

- The opposite attitude, that of optimistic risk-seeker, is captured by the *maximax* criterion:

$$a^* = \arg\max_{a \in A} \{\max_{s' \in s_a} u(s')\}$$

- This will ignore possible bad outcomes and just focus on the best outcome of each action.

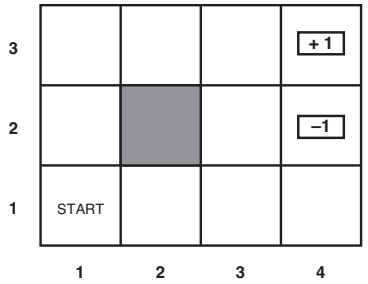
Sequential decision problems

- These approaches give us a battery of techniques to apply to individual decisions by agents.
- However, they aren't really sufficient.
- Agents aren't usually in the business of taking single decisions
  - Life is a series of decisions.

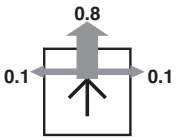
  The best overall result is not necessarily obtained by a greedy approach to a series of decisions.
- The current best option isn't the best thing in the long-run.

- Otherwise I'd only ever eat chocolate cake.

- Need to think about *sequential decision problems* where the agent's utility depends on a sequence of decisions.



(a)



(b)

- The agent has to pick a sequence of actions.

$$A(s) = \{Up, Down, Left, Right\}$$

for all states s.

- The world is fully observable. End states have values $+1$ or -1.

- If the world were deterministic, the choice of actions would be easy here.

$$Up, Up, Right, Right, Right$$

- But actions are stochastic.
- 80% of the time the agent moves as intended, but 20% of the time the agent moves perpendicular to the intended direction.

  Half the time to the left, half the time to the right. The agent doesn't move if it hits a wall.
- So $Up, Up, Right, Right, Right$ succeeds with probability:

$$0.8^5 = 0.32768$$

- (Also a small chance of going around the obstacle the other way.)

- We can write a *transition* model to describe these actions.
- Since the actions are stochastic, the model looks like:

$$P(s'|s, a)$$

where a is the action that takes the agent from s to s'.

- Transitions are assumed to be Markovian.
- So, we could write a large set of probability tables that would describe all the possible actions executed in all the possible states.

  This would completely specify the actions.

- The full description of the problem also has to include the utility function.
- This is defined over sequences of states.
- We will assume that in each state s the agent receives a reward $R(s)$.
- In general, this may be positive or negative.
- Here we will set the reward for non-terminal states to -0.04.
- We will assume that the utility of a run is the sum of the utilities of states, so the -0.04 is an incentive to take fewer steps to get to the terminal state.

  (You can also think of it as the cost of an action).

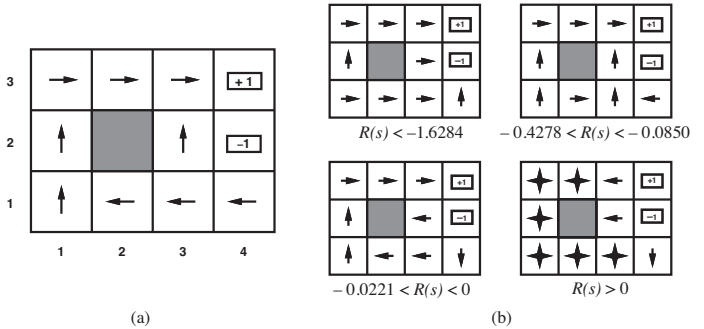
Markov decision process

- The overall problem the agent faces here is a *Markov decision process* (MDP)
- That is any fully observable non-deterministic environment with a Markovian transition model and additive rewards.
- Mathematically we have:
  - A set of states $s \in S$ with an initial state s_0.
  - A set of actions $A(s)$ in each state.
  - A transition model $P(s'|s, a)$; and
  - A reward function $R(s)$.

- What does a solution to an MDP look like?

- A solution is a *policy*, which we write as π.
- This is a choice of action for *every* state.
  - that way if we get off track, we still know what to do.
- In any state s, $\pi(s)$ identifies what action to take.
- We already met the idea of a policy in Lecture 8.

- Naturally we'd prefer not just any policy but the *optimum* policy.
  - But how to find it?
- Need to compare policies by the reward they generate.
- Since actions are stochastic, policies won't give the same reward every time.
  - So compare the expected value.
- The optimum policy π^* is the policy with the highest expected value.
- At every stage the agent should do $\pi^*(s)$.



(a) Optimal policy for the original problem.

(b) Optimal policies for different values of $R(s)$.

- $R(s) \leq -1.6284$, life is painful so the agent heads for the exit, even if is a bad state.
- $-0.4278 \leq R(s) \leq -0.0850$, life is unpleasant so the agent heads for the $+1$ state and is prepared to risk falling into the -1 state.
- $-0.0221 < R(s) < 0$, life isn't so bad, and the optimal policy doesn't take any risks.
- $R(s) > 0$, the agent doesn't want to leave.

How utilities are calculated

- So far we have assumed that utilities are summed along a run.
  - Not the only way.
- In general we need to compute $U_r([s_0, s_1, \ldots, s_n])$.
- Can consider *finite* and *infinite* horizons.
  - Is it "game over" at some point?
- Turns out that infinite horizons are mostly easier to deal with.
  - That is what we will use.
- Also have to consider whether utilities are *stationary* or *non-stationary*.
  - Does the same state always have the same value?

- Normally if we prefer one state to another
  - Passing the mid-term to failing it

  when we have the exam, today or next week, is irrelevant.
- So we say that utilities are *stationary*.

An aside

- Assuming that utilties are stationary makes no sense over the long term.
  - How many people like exactly the same things now that they liked 10 years ago?
  - Or 20 years ago?
- However, it may be reasonable that an agent's utilities stay somewhat fixed over a short period.

Back to sequential decisions

- With stationary utilities, there are two ways to establish $U_r([s_0, s_1, \ldots, s_n])$ from $R(s)$.
- *Additive* rewards:

$$U_r([s_0, s_1, \ldots, s_n]) = R(s_0) + R(s_1) + \ldots + R(s_n)$$

  as above.
- *Discounted* rewards:

$$U_r([s_0, s_1, \ldots, s_n]) = R(s_0) + \gamma R(s_1) + \ldots + \gamma^n R(s_n)$$

  where the *discount factor* γ is a number between 0 and 1.
- The discount factor models the preference of the agent for current over future rewards.

- There is an issue with infinite sequences with additive, undiscounted rewards.
  - What will the utility of a policy be?

- There is an issue with infinite sequences with additive, undiscounted rewards.
  - What will the utility of a policy be?
- ∞ or $-\infty$.
- This is problematic if we want to compare policies.
- Some solutions as follows.

- *Proper policies* always end up in a terminal state eventually. Thus they have a finite expected utility.

- We can compute the *average reward* per time step.

- With discounted rewards the utility of an infinite sequence is finite:

$$\begin{aligned} U_r([s_0, s_1, \ldots, s_n]) &= \sum_{t=0}^{\infty} \gamma^t R(s_t) \\ &\leq \sum_{t=0}^{\infty} \gamma^t R_{max} \\ &\leq \frac{R_{max}}{(1-\gamma)} \end{aligned}$$

  where $0 \leq \gamma < 1$ and rewards are bounded by $\pm R_{max}$

Optimal policies

- With discounted rewards we compare policies by computing their expected values.
- The expected utility of executing π starting in s is given by:

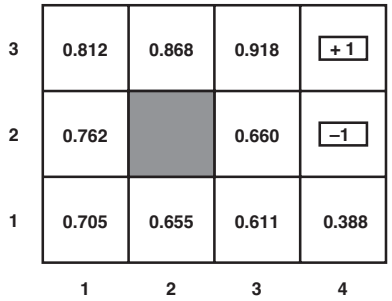
$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]$$

where S_t is the state the agent gets to at time t.

- S_t is a random variable and we compute the probability of all its values by looking at all the runs which end up there after t steps.
- The optimal policy is then:

$$\pi^* = \arg\max_{\pi} U^{\pi}(s)$$

and it turns out that this is independent of the state the agent starts in.



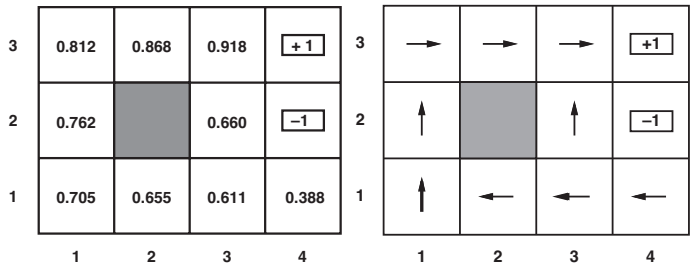
- Here we have the values of states

$U^{\pi^*}(s)$

if the agent executes an optimal policy

- If we have these values, the agent has a simple decision process
- It just picks the action a that maximises the expected utility of the next state:

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a) U^{\pi^*}(s')$$



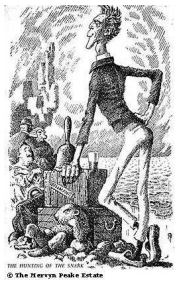
- The big question is how to compute $U^{\pi^*}(s)$.

Value iteration

- The key to computing the utility of a state is the *Bellman equation*:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s')$$

- We already met this (in Lecture 8) in a form suitable for deterministic actions.
- This version says the utility of a state is the reward for being in that state plus the expected discounted reward of being in the next state, assuming the agent picks the optimal action.



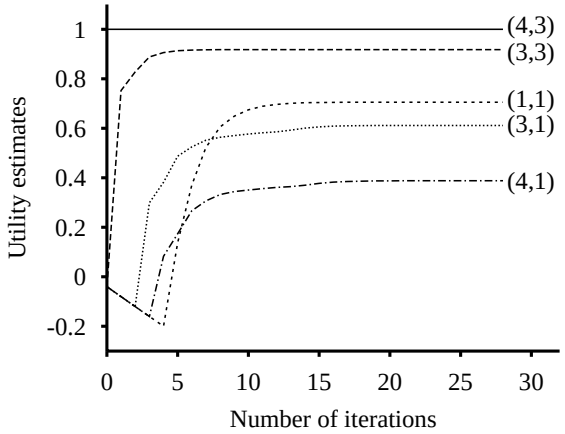
- There will be a set of Bellman equations, one for each state.
- We need to solve this set of (non-linear) equations.
  - Hard

  Because of the non-linearity.
- Luckily an iterative approach works.
  - Same basic process as we saw in Lecture 8.

- Start with *arbitrary* values for states and apply the Bellman update:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U_i(s')$$

  simultaneously to all the states.
- Continue until the values of states do not change.
- After an infinite number of applications, the values will converge on the optimal values.
- In practice we stop updating once the values of all states stop changing significantly.



- How the values of states change as updates occur.

- Intuition: works for the same reason as the deterministic case.
  - Eventually the right value propagates back from the goal.

Policy iteration

- Rather than compute optimal utility values, *policy iteration* looks through the space of possible policies.
- Starting from some initial policy π_0 there are two steps:
  - *Policy evaluation*
  - *Policy improvement*
- These steps are then repeated.

- *Policy evaluation*
  Given a policy π_i, calculate $U_i = U^{\pi_i}$.
- *Policy improvement*
  Calculate a new policy π_{i+1} by applying:

$$\pi_{i+1}(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

- At each iteration, improvement is based on our better understanding of the utility of each state.

- The iteration will terminate when there is no improvement in utility from one iteration to the next.
- At this point the utility U_i is a fixed point of the Bellman update and so π_i must be optimal.

- How do we calculate the utility of each step given the policy π_i?
- Turns out not to be so hard.
- Given a policy, the choice of action in a given state is fixed (that is what a policy tells us) so:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s')$$

- Again there are lots of simultaneous equations, but now they are linear (no $\max$) and so standard linear algebra solutions will work.

Bellman redux

- The Bellman equation(s)/update are widely used.

- D. Romer, It's Fourth Down and What Does the Bellman Equation Say? A Dynamic Programming Analysis of Football Strategy, NBER Working Paper No. 9024, June 2002

This paper uses play-by-play accounts of virtually all regular season National Football League games for 1998-2000 to analyze teams' choices on fourth down between trying for a first down and kicking. Dynamic programming is used to estimate the values of possessing the ball at different points on the field. These estimates are combined with data on the results of kicks and conventional plays to estimate the average payoffs to kicking and going for it under different circumstances. Examination of teams' actual decisions shows systematic, overwhelmingly statistically significant, and quantitatively large departures from the decisions the dynamic-programming analysis implies are preferable.

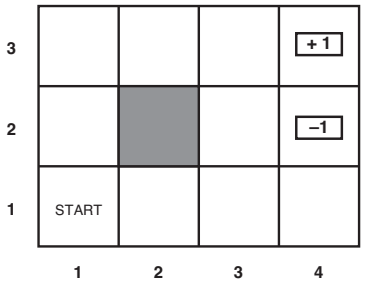
Partially observable MDPs

- MDPs made the assumption that the environment was fully observable.
  - Agent always knows what state it is in.
- The optimal policy only depends on the current state.
- Not the case in the real world.
  - We only have a belief about the current state.
- POMDPs extend the model to deal with partial observability.

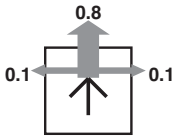
- Basic addition to the MDP model is the *sensor* model:

$$P(e|s)$$

  probability of perceiving e in state s.
- As a result of noise in the sensor model, the agent only has a belief about which state it is in.
- Probability distribution over the possible states.



(a)



(b)

$P(s_{1,1}) = 0.05, P(s_{1,2}) = 0.01, \ldots$

- If $b(s)$ was the distribution before an action and an observation, then afterwards the distribution is:

$$b'(s') = \alpha P(e|s') \sum_s P(s'|s,a) b(s)$$

- Everything in a POMDP hinges on the belief state b.
  - Including the optimal action.
- Indeed, the optimal policy is a mapping $\pi^*(b)$ from beliefs to actions.

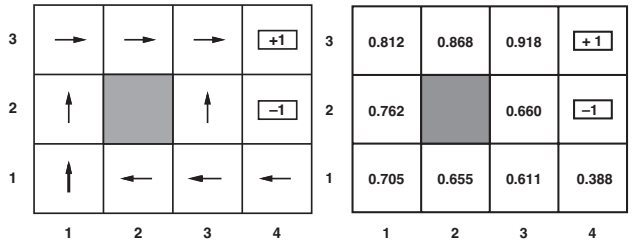
  "If you think you are next to the wall, turn left"
- The agent executes the optimal action given its beliefs, receives a percept e and then recomputes the belief state.

- The big issue in solving POMDPs is that beliefs are continuous.
- When we solved MDPs, we could search through the set of possible actions in each state to find the best.
- To solve a POMDP, we need to look through the possible actions for each belief state.

  But belief is continuous, so there are a lot of belief states.
- Exact solutions to POMDPs are intractable for even small problems (like the example we have been using).
- Need (once again) to use approximate techniques.

Reinforcement learning

- Remember this world which we solved as an MDP:



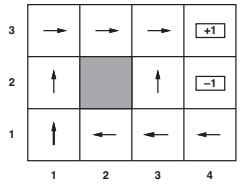
- In *passive learning* the agent's policy is fixed:
  - In state s it always executes $\pi(s)$.
- It has to learn the utility function $U^{\pi}(s)$.
- Comparing with the MDP case, the agent doesn't know the transition model:

  $$P(s'|s,a)$$

  and it doesn't know the reward function

  $$R(s)$$
- How can it learn them?

- It learns them by carrying out runs through the environment.



- A run is a sequence of states and actions that continues until the agent reaches the terminal state:

  $$(1,1)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow$$
- Note that we have reward as well.

- As the agent moves it can calculate a sample estimate of $P(s'|s, \pi(s))$
  - Each time it moves it creates a new sample for one state.
- Each reward is a contribution to the computation of utility.

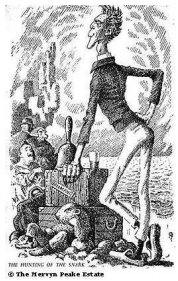
- We could estimate the utility of a state by the rewards generated along the run from that state.
  - Direct utility estimation.
- Thus a sample reward for $(1, 1)$ from the run above is the sum of the rewards all the way to a goal state.
- The same run will produce two samples for $(1, 2)$ and $(1, 3)$.
- You can do the calculation with or without discount.

Adaptive dynamic programming

- We can improve on the direct estimation by remembering the Bellman equation for a fixed policy:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s))U^{\pi}(s')$$

- The utility of a state is the reward for being in that state plus the expected discounted reward of being in the next state.
- This is the formula from page 46.

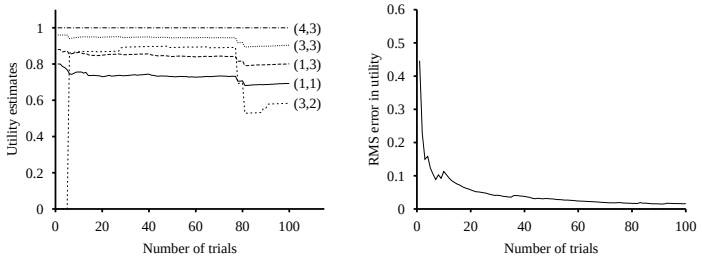


- Since we are using the fixed policy version of the Bellman equation we don't have the $\max$ that makes the original so hard to solve.
- Can just plug results into an LP solver
  - As we discussed when talking about policy iteration.
- Can also use value iteration, using:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s, \pi(s))U_i(s')$$

to update utilities.

- Results:



- Now, to get the utilities, the agent started with a fixed policy, so it always knew what action to take.
- It used this to get utilities.
- Having gotten the utilities, it could use them to choose actions.
  - Just picks the action with the best expected utility in a given state.
- However, there is a problem with doing this.
- What is it?

- Might not yet have experienced the bad effects of an action:

- Textbook uses the example of successfully running a red light.
- Of course, this kind of over-reliance on not-full-explored state/action spaces is what people do all the time.

- In addition, as the textbook points out, there are ways to get around this.
- There is no way to be sure that the action your reinforcement learner is picking doesn't have possible bad outcomes.
- But there are ways to try to mitigate the issue.

Active reinforcement learning

- The passive reinforcement learning agent is told what to do.
  - Fixed policy
- An active reinforcement learning agent must decide what to do.
- We'll think about how to do this by adapting the passive learner.
- We can use exactly the same approach to estimating the transition function.
  - Sample average of the transitions we observe.
- But computing utilities is more complex.

- When we had a policy, we could use the simple version of the Bellman equation:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) U^{\pi}(s')$$

- When we have to choose actions, we need to solve the full Bellman equation:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U(s')$$

  with its pesky $\max$.
- What to do?

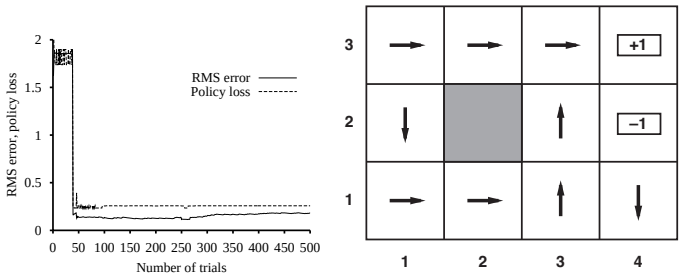
- Well, we know what to do, we use value iteration.
- At any stage, we can run:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U_i(s')$$

  to stability to compute a new set of utilities.

- Deciding what to do, what action to take, is the next issue.
- Normally after running value iteration we would choose the action with the highest expected utility.
  - Greedy agent
- This turns out not to be so great an idea.

- Typically a greedy agent will not learn the optimal policy:



- The issue is that once the agent finds a run that leads to a good reward, it tends to stick to it.
  - It stops exploring.
- May never find the best action for a given state.

- To do better we can force exploration using, for example, the ϵ-greedy approach.
- If a_t^* is the action we compute gives highest utility, ϵ-greedy action selection:

$$a_t = \begin{cases} a_t^* & \text{with probability } 1-\epsilon \\ \textit{random action} & \text{with probability } \epsilon \end{cases}$$

  for some $0 < \epsilon < 1$.
- This can be slow to converge.

- A better approach is to change the estimated utility assigned to states in value iteration.
- For example we can use:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} f\left(\sum_{s'} P(s'|s,a)U_i(s'), N(s,a)\right)$$

where $N(s,a)$ counts how many times we have done a in s, and $f(u,n)$ provides an exploration-happy estimate of the utility of a state.

- For example:

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

R^+ is an optimistic reward, and N_e is the number of times we want the agent to be forced to pick an action in every state.

Q-learning

- Q-learning is a *model-free* approach to reinforcement learning.
  - It doesn't need to learn $P(s'|s,a)$.
- Revolves around the notion of $Q(s,a)$, which denotes the value of doing a in s.

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

- We can write:

$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) max_{a'} Q(s',a')$$

and we could do value-iteration style updates on this.
(Wouldn't be model-free)

- However, we can write the update rule as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

and recalculate everytime that a is executed in s and takes the agent to s'.

- α is a learning rate, just like the learning rate in linear regression.
  - Controls how quickly we update the Q-value when we have new information.

Summary

- Today we looked at practical decision making for agents.
  - Practical in the sense that agents will need this kind of decision making to do the things they need to do.
- We looked in detail at solutions for techniques that work in fully observable worlds
  - MDPs
- We also briefly mentioned the difficulties of extending this work to partially observable worlds.
- We also looked at how reinforcement learning could be used to solve problems for which we don't have action models.