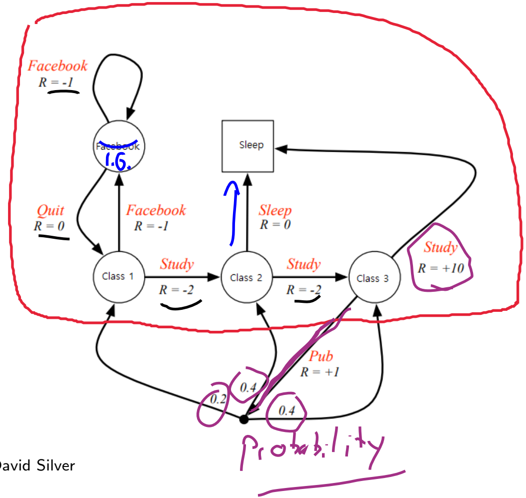
Nov 4 Markov Decision Processes



1 decisions

2 Uncertainty

3 motivation

1) Rewards

David Silver

Hidden Markov Models

Smoothing: Filter up to each step then backwards steps

1) **Filtering:** The goal is to predict X_{t+1} *given* all the evidence available $E_{1:t+1}$.

Method: Forward-stepping, starting at X_0.

2) **Prediction:** The goal is to predict X_{t+k+1} *given* all the evidence available $E_{1:t+1}$.

Method: Forward-stepping until evidence ends, then evolve the *Markov Model* to desired future times.

3) **Smoothing:** The goal is to try to update probabilities of prior states X based on current evidence.

Method: Backward-stepping. Requires forward steps up to desired state X_k then backwards steps from the future states with evidence.

Most Likely Sequence: The goal is to find the *most likely* sequence of X values that gave rise to our evidence.

Method: Viterbi algorithm. Instead of computing the exact probabilities (as a product) of a specific X sequence, only compute their log-sums and recursively ask what the most likely state of X_k was for *each* possible level of X_{k+1}.

prior time

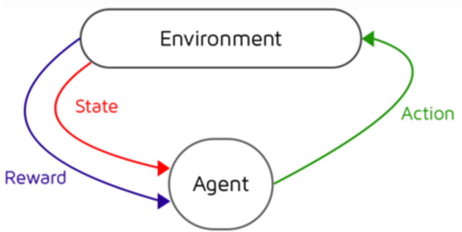
Markov Decision Process

A *Markov Decision Process* (MDP) is a sequential decision problem that is the combination of a Markov Model and a decision-making agent. It asks the question: how do we maximize utility if there are uncertainties associated with the successor states of each action? To do this, we require:

1. A fully observable, stochastic environment
2. A Markov transition model that gives probabilities of states *given* decisions
3. An additive reward structure

They are often used for

1. Inventory management
2. Routing/logistics
3. Games
4. Planning under uncertainty



MDPs

Consider an agent-based game. We win if we reach the treasure. We lose if we run into the internet troll or the dragon.

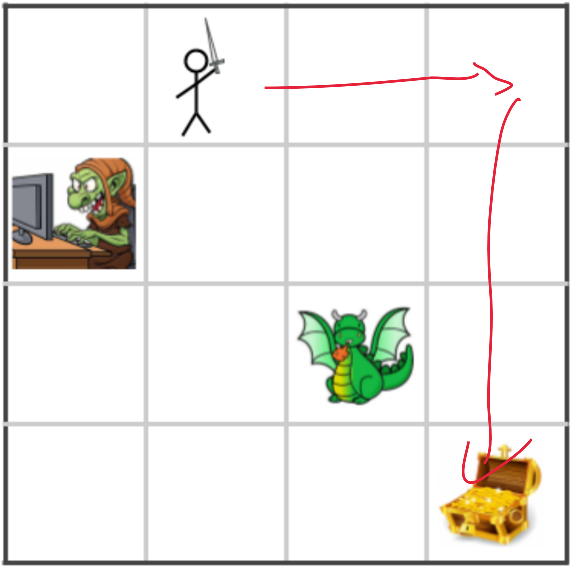
Goal: describe the appropriate set of moves from our current location to the treasure.

MDPs

Consider an agent-based game. We win if we reach the treasure. We lose if we run into the internet troll or the dragon.

Goal: describe the appropriate set of moves from our current location to the treasure.

Suppose the Dragon and Troll can't move. What do we do?



MDPs

Consider an agent-based game. We win if we reach the treasure. We lose if we run into the internet troll or the dragon.

Goal: describe the appropriate set of moves from our current location to the treasure.

Suppose the Dragon and Troll can't move. What do we do?

Twist: suppose, as in our trip to Taco Bell, we sometimes get disoriented, and move in a *different* direction than the one we choose!

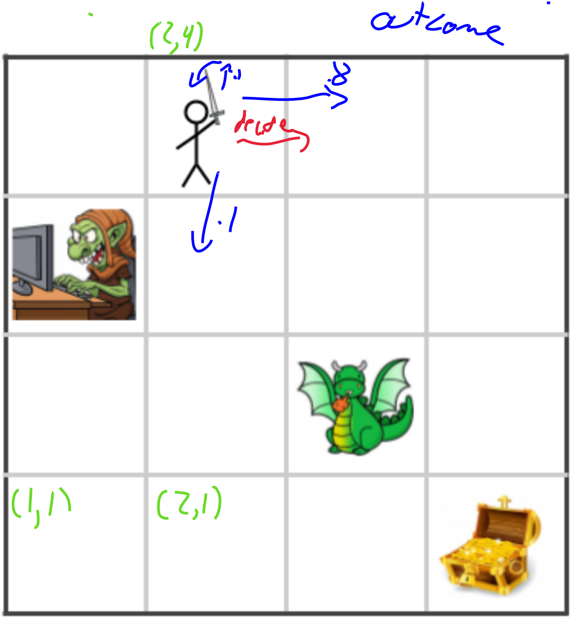
MDP Uncertainty and Choice

The MDP is meant to describe a real world process where actions are not perfectly reliable. Suppose we describe a transition model:

1. *Given* our intended action, the probability we move where we intend is .8.
2. *Given* our intended action, the probability we move to either side (90°) is .1 each.

Definition: A *policy* is what we would tell our agent to do in any given possible state s. Denote $\pi(s)$ by the policy chosen at state s.

$\pi((2,4)) =$ "right"



MDP Rewards

To choose a policy we need a notion of what makes a move good or bad. Suppose that:

1. Moving to the dragon or troll achieves a reward of -1, and ends the game.
2. Moving to the treasure achieves a reward of 1, and ends the game. reward is state
3. We can define a reward $\boxed{R(s)}$ (or maybe $R(s \to s'))$ associated with moving to any state. () rewards at movements.

We may even encode a reward for the non-movement $s \to s$. For example, $R(s \to s) > 0$, an agent will rarely move, whereas a reward of $R(s \to s) = -2$ will create a frenetic, always-moving agent.

MDP Utility

Since rewards may not exist on all actions, we need to conceptualize an *expected* rewards or a *long-run* rewards. These ~~like~~ live in the *utility* associated with each state.

Utility is our long-run gain.

1. It depends on the entire *sequence* of states visited, $[s_0, s_1, \ldots s_{50}, s_{51}, \ldots]$
2. Informal definition: utility is the *sum* of rewards achieved over a set of states/movements: $U[s_0, s_1, \ldots s_{50}, s_{51}, \ldots] = R(s_0) + R(s_1) + \ldots$

Classically, two terms are added to clarify and add tuning to the concept of utility.

MDP Utility

Definition: The *Time Horizon* of an MDP can be either:

1. *Finite Horizon*, where after a fixed time N no actions matter. Here we consider the rewards or utility $U[s_0, s_1, \dots s_N, \cancel{s_{N+1}, \dots}] = U[s_0, s_1, \dots s_N]$. The length of the horizon may impact your decisions.
   Example: It's the first/last lap of your game of Mario Kart. Should you save your star piece for when someone tries to shoot you?

2. *Infinite Horizon*, where there is never a reason to behave differently in the same state at different times.
   Example: *When* you get the treasure doesn't matter, only whether you get there.

Definition: The *discount factor* of an MDP is a multiplicative punishment γ for taking longer to reach rewards. It's common in finance as it represents an increased value of immediate rewards over future rewards. $\gamma \in [0,1]$.

$$U[s_0, s_1, \dots s_{50}, s_{51}, \dots] = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \gamma^3 R(s_3) + \dots$$

MDP Utility

Definition: The *discount factor* of an MDP is a multiplicative punishment γ for taking longer to reach rewards. It's common in finance as it represents an increased value of immediate rewards over future rewards.

$$U[s_0, s_1, \ldots s_{50}, s_{51}, \ldots] = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \gamma^3 R(s_3) + \ldots$$

MDP Goal

So we have:

1. A Markov chain that gives successors *given* actions
2. Rewards associated with each state (or each state transition)
3. A utility that may track rewards of a *sequence* of states
4. A possible discount on *when* we get rewards
5. A preference for how many total moves matter

All told, an MDP is defined as a 5-tuple! **States,** **Actions,** **Rewards,** **Transition Probabilities**, and **Discounting**

nodes edges nodes given actions

Goal: The output of an MDP is an *optimal policy* that specifies where to move from a given starting state s. We maximize the expected utility under policy π:

utility of decision "π" from state "s"

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

MDP Rewards

Suppose we start at the state $(3, 1)$. What is the *expected utility* of the policy of “move to the right”?

At time $t = 0$, we have utility of $\boxed{\gamma^0 R((3, 1))}$.

1. 80% chance we actually go right and achieve a reward of $+1$, for utility γ^1.
2. 10% chance we actually go up and achieve a reward of -1, for utility $-\gamma^1$.
3. 10% chance we actually go right and achieve a reward of... whatever the discounted *utility* of tile $(2, 1)$ is.

MDP Rewards

Suppose we start at the state $(3, 1)$. What is the *expected utility* of the policy of "move to the right"?

At time $t = 0$, we have utility of $\gamma^0 R((3, 1))$.

1. 80% chance we actually go right and achieve a reward of $+1$, for utility γ^1.
2. 10% chance we actually go up and achieve a reward of -1, for utility $-\gamma^1$.
3. 10% chance we actually go right and achieve a reward of... whatever the discounted *utility* of tile $(2, 1)$ is.

MDP Rewards

Suppose we start at the state $(3,1)$. What is the *expected utility* of the policy of "move to the right"?

At time $t=0$, we have utility of $\gamma^0 R((3,1))$.

1. 80% chance we actually go right and achieve a reward of $+1$, for utility γ^1.
2. 10% chance we actually go up and achieve a reward of -1, for utility $-\gamma^1$.
3. 10% chance we actually go right and achieve a reward of... whatever the discounted *utility* of tile $(2,1)$ is.

So at $t=1$, $U^\pi((3,1)) =$

$$\gamma\,[.1R(3,2) + .8R(4,1) + .1R(3,1)]$$

infinite horizon: we're not saying "t". time.

MDP Rewards: What to consider

Our utility gained after one attempt to move right was

$$
\begin{aligned}
U^\pi((3,1)) &= \gamma\,[.1R(3,2) + .8R(4,1) + .1R(3,1)] \\
&= \gamma\,[.1(-1) + .8(1) + .1\underbrace{R(3,1)}_{((3,1),\, t=1)}]
\end{aligned}
$$

If we're allowed to take more moves, the $R(3,1)$ term should get it's own policy: where should we move from $(3,1)$ if it's currently $t = 1$?

But the value of $U^\pi((3,1))$ at $t = 1$ isn't necessarily the same as the left-hand side of $U^\pi((3,1))$ at $t = 0$! This now depends on our *horizon*. If we only had one more left, it might be utility of zero!

MDP Algorithm:

γ

Our goal with an MDP is to maximize the expected discounted utility after playing the game to its horizon. So we compute the utility associated with any given policy π and choose the best one, the *optimal policy* $\pi^*(s)$:

$$\underbrace{\pi^*(s)}_{\text{best policy}} = \arg\max_{\pi} U^{\pi}(s)$$

Definition: The *true utility* of a state is its utility when associated with the optimal policy π^*. We denote it $U^{\pi^*}(s)$ or just $U(s)$.

U(s): utility achieved from s given optimal policies

Result: The *true utility* of a state is the expected utility gained by choosing the best successor state. This is the sum of the discounted utilities of all the possible successors of state s under optimal decision a.

MDP Algorithm:

Is we can calculate that true utility for each state $U(s)$, we can pick actions that maximize it, or at least it's expected value.

Suppose we are in a state s. Then we have a **set** of possible outcomes of taking action a.

Denote:

$\pi(s) =$ action a

$$P(s'|s,a) := \text{Probability of ending up in state } s' \text{ given action } a \text{ from state } s.$$

$s' :=$ { up, left, right } P = .8, .1, .1

$a =$ "up"

The resulting expected utility of action a is:

$$\boxed{\sum_{s'} P(s'|s,a)U(s')}$$

$= P(\text{"up"}) \cdot U(s \text{ that "up"}) + P(\text{"left"}) \cdot U(\text{"left"})$

In other words: we sum the utilities of the successor states multiplied by their respective probabilities. The optimal policy is the a that is maximal:

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

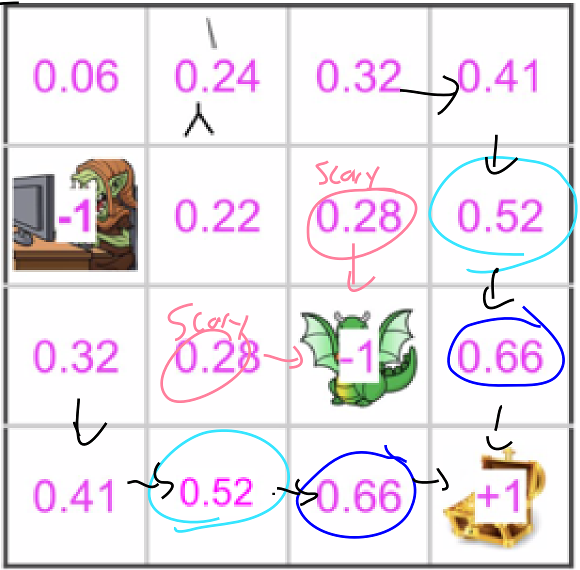
MDP Algorithm:

Policies

So our goal is to compute *utilities.* Supposing that time doesn't matter (infinite horizon), this is a large set of calculations that depend on one another.

$$\sum_{s'} P(s'|s,a)U(s')$$

One way to solve this is to set up a *large* linear system. This tends to be tedious to solve!



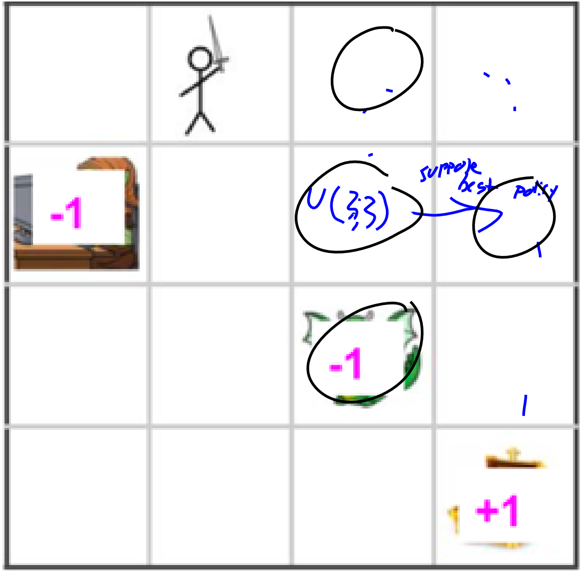
MDP Value Iteration:

$$U(3,3) = \gamma[.8U(4,3) + .1U(3,4) + .1U(2,3)]$$

The algorithm to find values on an infinite horizon starts with stating known terminal states and their values.
We also often have minor punishments for taking a long time to find a terminal state, such as $R(s) = -0.03$ for each non-terminal state.
Consider a discount factor of $\gamma = 0.9$.

linear system $\quad O(n^3)$

Balance:

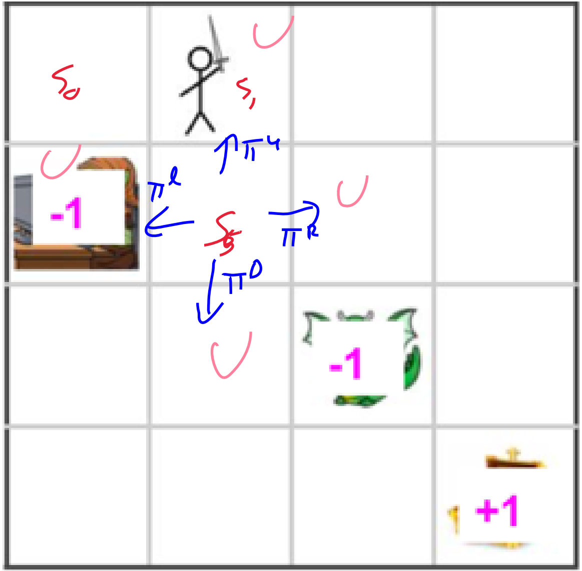


MDP Value Iteration:

Value Iteration Algorithm:

1. Start with *candidate* utilities for each state. (rewards?)
2. Do the following *many* times: (until bored/convergence)
   For each state s:
   - 2.1 Collect the set of valid actions $a \in A$
   - 2.2 For each a, consider the successor of that action and their associate probabilities: $P(s'|s,a)$, **then** calculate the expected utility of action a.
   - 2.3 Assign $U(s)$ to the max of the discounted expected utilities, plus any immediate reward for s.

$O(n \cdot \#iter)$



MDP Value Iteration:

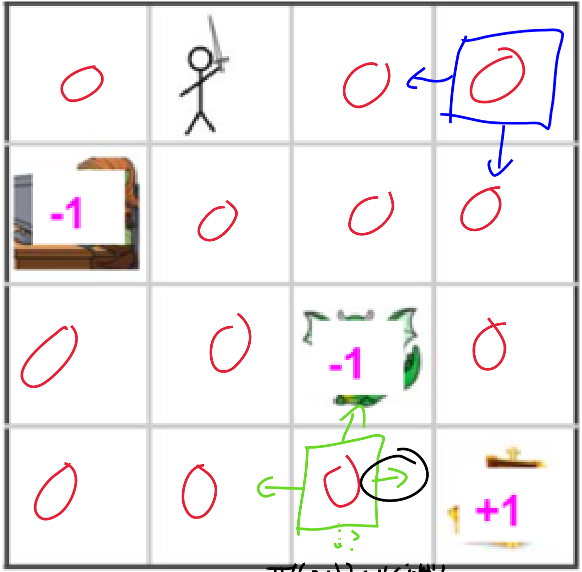
Value Iteration Algorithm:

1. Start with *candidate* utilities for each state.
2. Do the following *many* times:
   For each state s: $U_{i+1}(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$

Example: Find U_0 and U_1 for each state.

U_0:

$U_1((4,1)) = 0$



$\pi((3,1))$:

MDP Value Iteration:

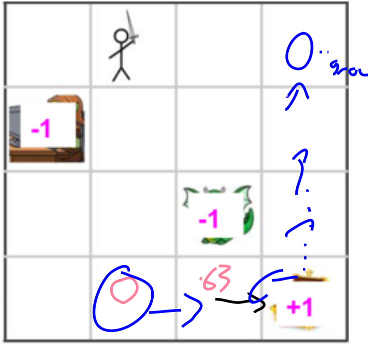
Example: Find U_0 and U_1 for each state.

U_2

$$U_1((3,1)) = \gamma \left[.8\, U_0((4,1)) + .1\, U_0((3,2)) + .1\, U_0((3,1)) \right]$$

$$= .9 \left[.8(1) + .1(-1) + .1(0) \right]$$

$$= .9[.7] = .63 = U_1[(3,1)]$$



MDP Value Iteration:

Example: Find U_0 and U_1 for each state.

Partial Soln: Consider (2,1). Each of it's neighbors has utility of 0, so *any* movement has utility of zero. Then $U_1((2,1)) = -.03 + 0 = -.03$.

Consider (3,1). This one is interesting!

Action: "right" has

$$U_1^R((3,1)) = .8U_0((4,1)) + .1U_0((3,1)) + .1U_0((3,2))$$

Action: "up" has

$$U_1^U((3,1)) = .8U_0((3,2)) + .1U_0((2,1)) + .1U_0((4,1))$$

... and so forth. Right is better, so we would choose that and assign *its* value to $U_1((3,1))$ Notice how the effects of the 1 and the -1's spread further each iteration!

Notice how the effects of the 1 and the -1's spread further each iteration!

Value Iteration Pseudocode

```
def value_iteration(mdp, tolerance):

    # initilize utility for all states

    # iterate:

        # make a copy of current utility, to be modified

        # initialize maximum change to 0

        # for each state s:

            # for each available action, what next states
            # are possible, and their probabilities?

            # calculate the maximum expected utility

            # new utility of s = reward(s) +
            #                    discounted max expected utility

            # update maximum change in utilities, if needed

        # if maximum change in utility from one iteration to the
        # next is less than some tolerance, break!

    return # final utility
```

Policy Iteration

Once we have utilities for each state, we need to specify **policies**. This is fairly intuitive!

Policy Iteration Algorithm:
A two-step algorithm that alternates between **policy evaluation** and **policy improvement**

1. **Policy Evaluation**: *Given* a policy π_i, compute $U_i = U_i^\pi$, the utility of each state if that policy is followed. This is only *one* action considered per state. Still have to iterate!

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

2. **Policy Improvement**: After updating the utility calculations in step 1, calculate a new policy π_{i+1} using π_i and U_i. Compare the utility from π_i to alternatives $a \in A$. If

$$\max \_a \in A \sum_{s'} P(s'|s, a)U_i(s') > \sum_{s'} P(s'|s, \pi_i(s))U_i(s')$$

then set $\pi_{i+1}(s) =$ that action.

Policy Iteration Pseudocode

```
def policy_iteration(mdp):

    # initilize utility for all states

    # initialize a policy for each state, being a random action

    # iterate:

        # update utility, using policy evaluation and
        # current estimates of utility and policy

        # initialize unchanged = True

        # for each state s:

            # among the possible actions, which yields
            # the maximum expected utility?

            # if the best action choice is not currently
            # the policy for s, update it

        # if no policy values are changed, break!

    return # final policy (and/or utility)

def policy_evaluation(policy, utility, mdp, n_iter):
    # do a handful of value iteration updates of
    # the input utility, under the given policy
    return # updated utility
```

Moving Forward

- Coming up:
  1. More on MDPs, and their value of information/uncertainty.
  2. MDP NB on Friday.