Artificial Intelligence Programming

Markov Decision Processes

Chris Brooks

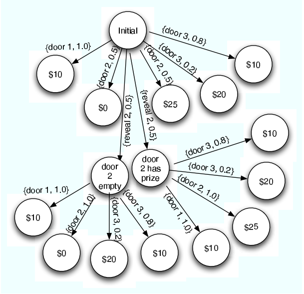
Department of Computer Science
University of San Francisco

23-2: Making Sequential Decisions

- Previously, we've talked about:
  - Making one-shot decisions in a deterministic environment
  - Making sequential decisions in a deterministic environment
    - Search
    - Inference
    - Planning
  - Making one-shot decisions in a stochastic environment
    - Probability and Belief Networks
    - Expected Utility
- What about sequential decisions in a stochastic environment?

23-3: Sequential Decisions

- We've thought a little bit about this in terms of value of information.
- We can model this as a state-space problem.
- We can even use a minimax-style approach to determine the optimal actions to take.



23-4: Expected Utility

- Recall that the expected utility of an action is the utility of each possible outcome, weighted by the probability of that outcome occurring.
- More formally, from state s, an agent may take actions $a_1, a_2, ..., a_n$.
- Each action a_i can lead to states $s_{i1}, s_{i2}, ..., s_{im}$, with probability $p_{i1}, p_{i2}, ..., p_{im}$
- $$EU(a_i) = \sum p_{ij} s_{ij}$$
- We call the set of probabilities and associated states the state transition model.
- The agent should choose the action a' that maximizes EU.

23-5: Markovian environments

- We can extend this idea to sequential environments.
- Problem: How to determine transition probabilities?
  - The probability of reaching state s given action a might depend on previous actions that were taken.
  - Reasoning about long chains of probabilities can be complex and expensive.
- The Markov assumption says that state transition probabilities depend only on a finite number of parents.
- Simplest: a first-order Markov process. State transition probabilities depend only on the previous state.
  - This is what we'll focus on.

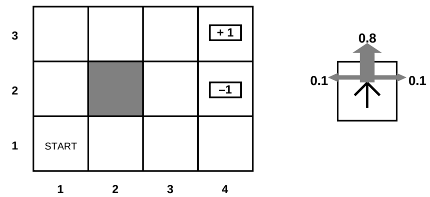
23-6: Stationary Distributions

- We'll also assume a *stationary distribution*
- This says that the probability of reaching a state s' given action a from state s with history H does not change.
- Different histories may produce different probabilities
- Given identical histories, the state transitions will be the same.
- We'll also assume that the utility of a state does not change throughout the course of the problem.
  - In other words, our model of the world does not change while we are solving the problem.

23-7: Solving sequential problems

- If we have to solve a sequential problem, the total utility will depend on a sequence of states $s_1, s_2, ..., s_n$.
- Let's assign each state a reward $R(s_i)$.
- Agent wants to maximize the sum of rewards.
- We call this formulation a Markov decision process.
  - Formally:
  - An initial state s_0
  - A discrete set of states and actions
  - A Transition model: $T(s, a, s')$ that indicates the probability of reaching state s' from s when taking action a.
  - A reward function: $R(s)$

23-8: Example grid problem



(a) (b)

- Agent moves in the "intended" direction with probability 0.8, and at a right angle with probability 0.2
- What should an agent do at each state to maximize reward?

23-9: MDP solutions

- Since the environment is stochastic, a solution will not be an action sequence.
- Instead, we must specify what an agent should do in any reachable state.
- We call this specification a *policy*
  - "If you're below the goal, move up."
  - "If you're in the left-most column, move right."
- We denote a policy with π, and $\pi(s)$ indicates the policy for state s.

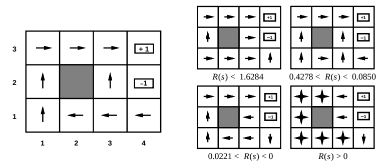
23-10: MDP solutions

- Things to note:
  - We've wrapped the goal formulation into the problem
    - Different goals will require different policies.
  - We are assuming a great deal of (correct) knowledge about the world.
    - State transition models, rewards
    - We'll touch on how to learn these without a model.

23-11: Comparing policies

- We can compare policies according to the expected utility of the histories they produce.
- The policy with the highest expected utility is the *optimal policy*.
- Once an optimal policy is found, the agent can just look up the best action for any state.

23-12: Example grid problem



(a) (b)

Left figure: R(s) = -0.04. Note - there are typos in this figure - all non-zero rewards should be negative.

- As the cost of being in a nonterminal state changes, so does the optimal policy.
- Very high cost: Agent tries to exit immediately, even through bad exit.
- Middle ground: Agent tries to avoid bad exit
- Positive reward for nonterminals: Agent doesn't try to exit.

23-13: More on reward functions

- In solving an MDP, an agent must consider the value of future actions.
- There are different types of problems to consider:
- Horizon - does the world go on forever?
  - Finite horizon: after N actions, the world stops and no more reward can be earned.
  - Infinite horizon; World goes on indefinitely, or we don't know when it stops.
    - Infinite horizon is simpler to deal with, as policies don't change over time.

23-14: More on reward functions

- We also need to think about how to value future reward.
- $100 is worth more to me today than in a year.
- We model this by *discounting* future rewards.
  - γ is a *discount factor*
- $U(s_0, s_1, s_2, s_3, ...) =$ $R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \gamma^3 R(s_3) + ..., \gamma \in [0, 1]$
- If γ is large, we value future states
- if γ is low, we focus on near-term reward
- In monetary terms, a discount factor of γ is equivalent to an interest rate of $(1/\gamma) - 1$

23-15: More on reward functions

- Discounting lets us deal sensibly with infinite horizon problems.
  - Otherwise, all EUs would approach infinity.
- Expected utilities will be finite if rewards are finite and bounded and $\gamma < 1$.
- We can now describe the optimal policy π^* as:
-

$$\pi^* = argmax_\pi EU(\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi)$$

23-16: Value iteration

- How to find an optimal policy?
- We'll begin by calculating the expected utility of each state and then selecting actions that maximize expected utility.
- In a sequential problem, the utility of a state is the expected utility of all the state sequences that follow from it.
- This depends on the policy π being executed.
- Essentially, $U(s)$ is the expected utility of executing an optimal policy from state s.

23-17: Utilities of States

3	**0.812**	**0.868**	**0.918**	**+ 1**
2	**0.762**		**0.660**	**–1**
1	**0.705**	**0.655**	**0.611**	**0.388**
	1	**2**	**3**	**4**

- Notice that utilities are highest for states close to the +1 exit.

23-18: Utilities of States

- The utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action.
-

$$U(s) = R(s) + \gamma max_a \sum_{s'} T(s, a, s') U(s')$$

- This is called the Bellman equation
- Example:

$$\begin{aligned} U(1,1) = -0.04 + \gamma max(&0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1) \\ &0.9U(1,1) + 0.1U(1,2), \\ &0.9U(1,1) + 0.1U(2,1), \\ &0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1)) \end{aligned}$$

23-19: Dynamic Programming

- The Bellman equation is the basis of dynamic programming.
- In an acyclic transition graph, you can solve these recursively by working backward from the final state to the initial states.
- Can't do this directly for transition graphs with loops.

23-20: Value Iteration

- Since state utilities are defined in terms of other state utilities, how to find a closed-form solution?
- We can use an iterative approach:
  - Give each state random initial utilities.
  - Calculate the new left-hand side for a state based on its neighbors' values.
  - Propagate this to update the right-hand-side for other states,
  - Update rule: $U_{i+1}(s) = R(s) + \gamma max_a \sum_{s'} T(s, a, s')U_i(s')$
- This is guaranteed to converge to the solutions to the Bellman equations.

23-21: Value Iteration algorithm

```
do
   for s in states
      U(s) = R(s) + max T(s,a,s') U(s')
until
     all utilities change by less then delta
```

- where $\delta = error * (1 - \gamma)/\gamma$

23-22: Discussion

- Strengths of Value iteration
  - Guaranteed to converge to correct solution
  - Simple iterative algorithm
- Weaknesses:
  - Convergence can be slow
  - We really don't need all this information
  - Just need *what to do* at each state.

23-23: Policy iteration

- Policy iteration helps address these weaknesses.
- Searches directly for optimal policies, rather than state utilities.
- Same idea: iteratively update policies for each state.
- Two steps:
  - Given a policy, compute the utilities for each state.
  - Compute a new policy based on these new utilities.

23-24: Policy iteration algorithm

```
Pi = random policy vector indexed by state
do
  U = evaluate the utility of each state for Pi
  for s in states
    a = find action that maximizes expected utility for that state
    Pi(s) = a
while some action changed
```

23-25: Learning a Policy

- So far, we've assumed a great deal of knowledge
- In particular, we've assumed that a model of the world is known.
  - This is the state transition model and the reward function
- What if we don't have a model?
- All we know is that there are a set of states, and a set of actions.
- We still want to learn an optimal policy

23-26: Q-learning

- Learning a policy directly is difficult
- Problem: our data is not of the form: <state, action>
- Instead, it's of the form $s_1, s_2, s_3, ..., R$.
- Since we don't know the transition function, it's also hard to learn the utility of a state.
- Instead, we'll learn a value function $Q(s, a)$. This will estimate the "utility" of taking action a in state s.

23-27: Q-learning

- More precisely, $Q(s, a)$ will represent the value of taking a in state s, then acting optimally after that.
- $Q(s, a) = R(s, a) + \gamma max_a \sum_{s'} T(s, a, s')U(s')$
- The optimal policy is then to take the action with the highest Q value in each state.
- If the agent can learn $Q(s, a)$ it can take optimal actions even without knowing either the reward function or the transition function.

23-28: Learning the Q function

- To learn Q, we need to be able to estimate the value of taking an action in a state even though our rewards are spread out over time.
- We can do this iteratively.
- Notice that $U(s) = max_a Q(s, a)$
- We can then rewrite our equation for Q as:
- $Q(s, a) = R(s, a) + \gamma max_{a'} Q(s', a')$

23-29: Learning the Q function

- Let's denote our estimate of $Q(s, a)$ as $\hat{Q}(s, a)$
- We'll keep a table listing each state-action pair and estimated Q-value
- the agent observes its state s, chooses an action a, then observes the reward $r = R(s, a)$ that it receives and the new state s'.
- It then updates the Q-table according to the following formula:
- $\hat{Q}(s, a) = r + \gamma max_{a'} \hat{Q}(s', a')$

23-30: Learning the Q function

- The agent uses the estimate of $\hat{Q}$ for s' to estimate $\hat{Q}$ for s.
- Notice that the agent doesn't need any knowledge of R or the transition function to execute this.
- Q-learning is guaranteed to converge as long as:
  - Rewards are bounded
  - The agent selects state-action pairs in such a way that it each infinitely often.
  - This means that an agent must have a nonzero probability of selecting each a in each s as the sequence of state-action pairs approaches infinity.

23-31: Exploration

- So how to guarantee this?
- Q-learning has a distinct difference from other learning algorithms we've seen:
- The agent can select actions and observe the rewards they get.
- This is called *active learning*
- Issue: the agent would also like to maximize performance
  - This means trying the action that currently looks best
  - But if the agent never tries "bad-looking" actions, it can't recover from mistakes.
- Intuition: Early on, $\hat{Q}$ is not very accurate, so we'll try non-optimal actions. Later on, as $\hat{Q}$ becomes better, we'll select optimal actions.

23-32: Boltzmann exploration

- One way to do this is using Boltzmann exploration.
- We take an action with probability:
- $P(a|s) = \frac{k^{\hat{Q}(s,a)}}{\sum_j k^{\hat{Q}(s,a_j)}}$
- Where k is a temperature parameter.
- This is the same formula we used in simulated annealing.

23-33: Reinforcement Learning

- Q-learning is an example of what's called reinforcement learning.
- Agents don't see examples of how to act.
- Instead, they select actions and receive rewards or punishment
- An agent can learn, even when it takes a non-optimal action.
- Extensions deal with delayed reward and nondeterministic MDPs.

23-34: Summary

- Q-learning is sometimes referred to as model-free learning.
- Agent only needs to choose actions and receive rewards.
- Problems:
  - How to generalize?
  - Scalability
  - Speed of convergence
- Q-learning has turned out to be an empirically useful algorithm.