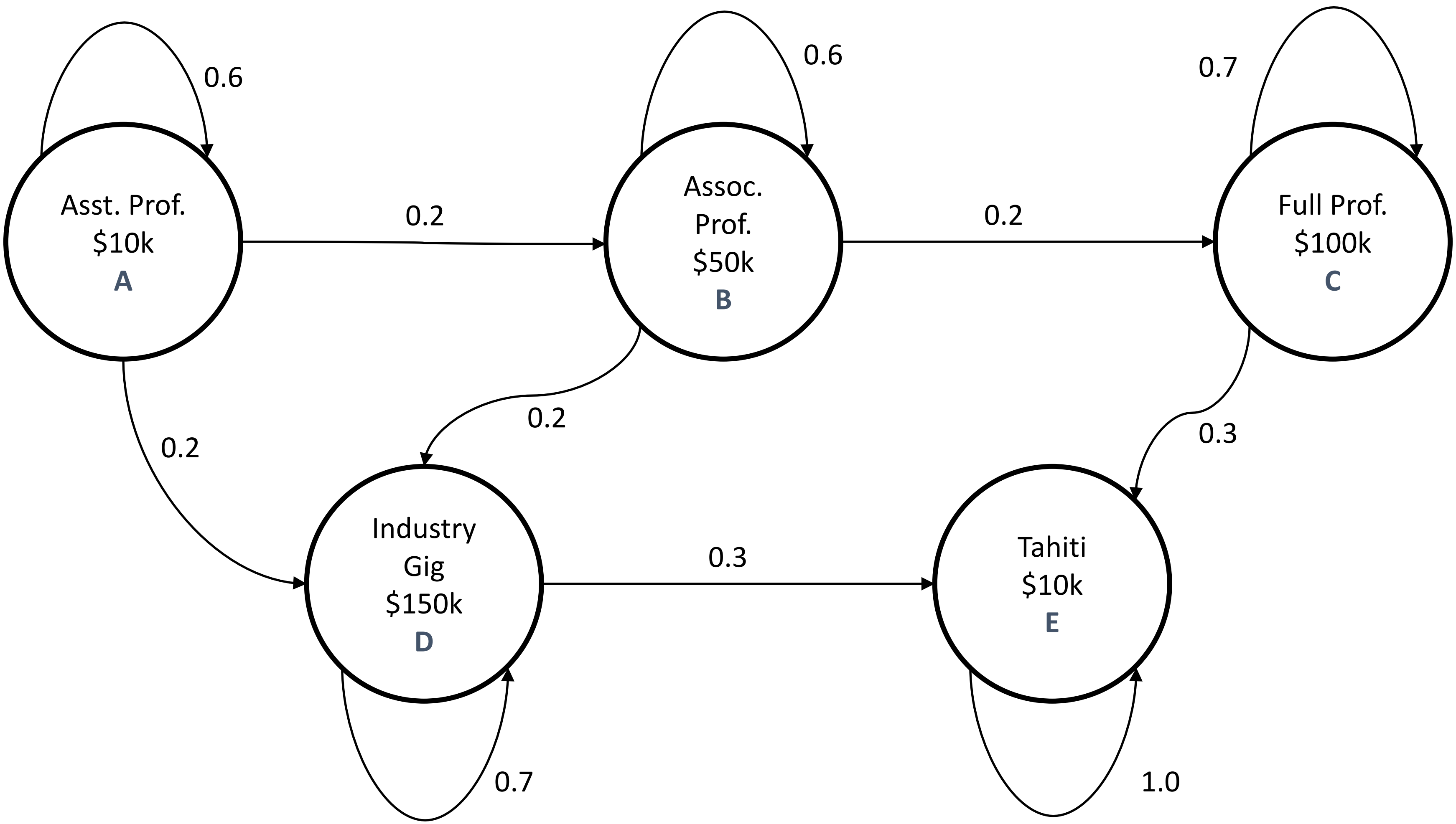
An Introduction to Reinforcement Learning

Raghu Ramanujan
Dept. of Mathematics and Computer Science
Davidson College

AlphaGo

Markov Chains

A Markov Chain



Markov Chain Formalism

- A finite Markov Chain is a tuple *(S, P, R, $\boldsymbol{\gamma}$)* where:
  - *S* is a finite set of states

$$S = \{s_1, s_2, \ldots, s_n\}$$

  - *P* is the transition matrix

$$P = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ & & \vdots & \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{pmatrix}$$

  - *R* is the reward function

$$R : S \mapsto \mathbb{R}$$

  - $\boldsymbol{\gamma}$ is the discount factor

$$\gamma \in [0, 1)$$

Value Iteration

- Can we solve these equations a different way?

- One idea: use iterative approach
  - $V^0(s_i)$: expected discounted sum of rewards from s_i after 0 steps
  - $V^1(s_i)$: expected discounted sum of rewards from s_i after 1 step
  - $V^2(s_i)$: expected discounted sum of rewards from s_i after 2 steps
  - …

- Algorithm: compute $V^0(s_i)$, $V^1(s_i)$, $V^2(s_i)$, … for all s_i
  - Stop when:

$$\max_{s_i} |V^{k+1}(s_i) - V^k(s_i)| < \epsilon$$

  - Claim: this converges to $V^*(s_i)$ as $k \rightarrow \infty$

Markov Decision Processes

Markov Decision Process Formalism

- A finite Markov Decision Process (MDP) is a 5-tuple *(S, A, T, R, γ)* where:
  - *A* is a set of actions the agent can take

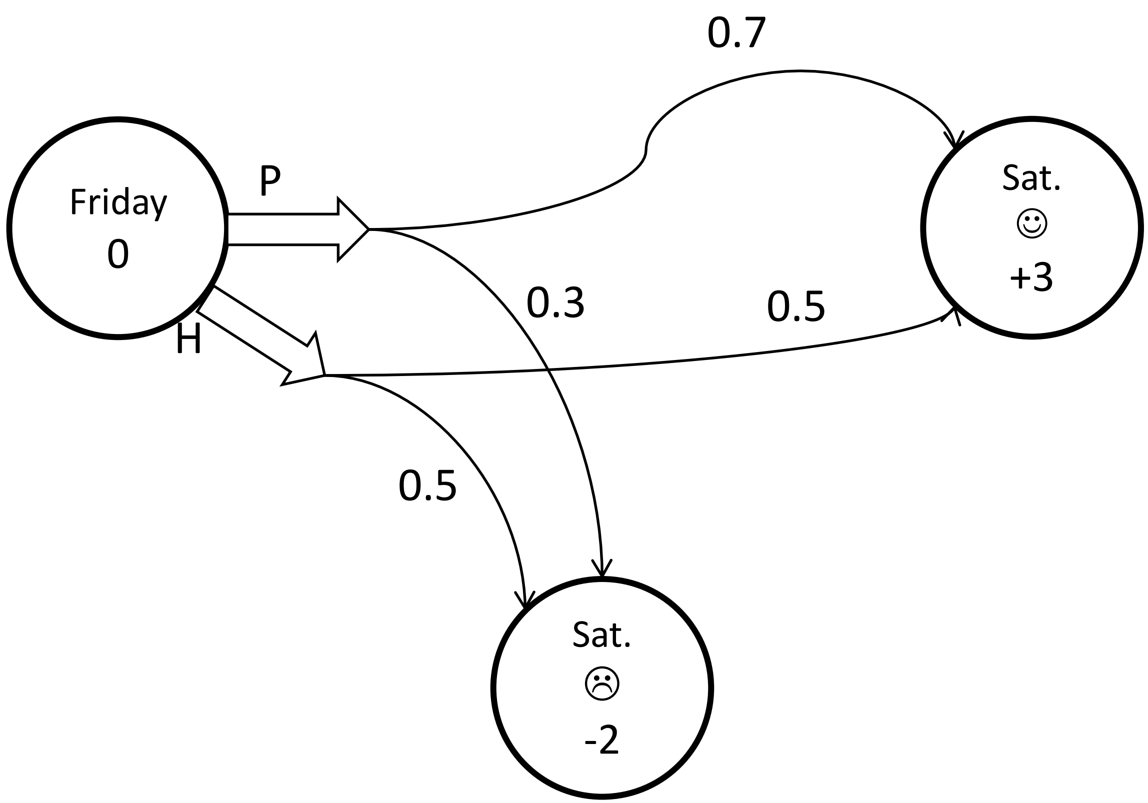
$$A = \{a_1, a_2, \ldots, a_m\}$$

  - *T* is the transition model
    - No longer a matrix with entries p_{ij}, but a tensor with entries p^a_{ij}
    - "If I'm in s_i and perform action a, what is the probability I end up in s_j"

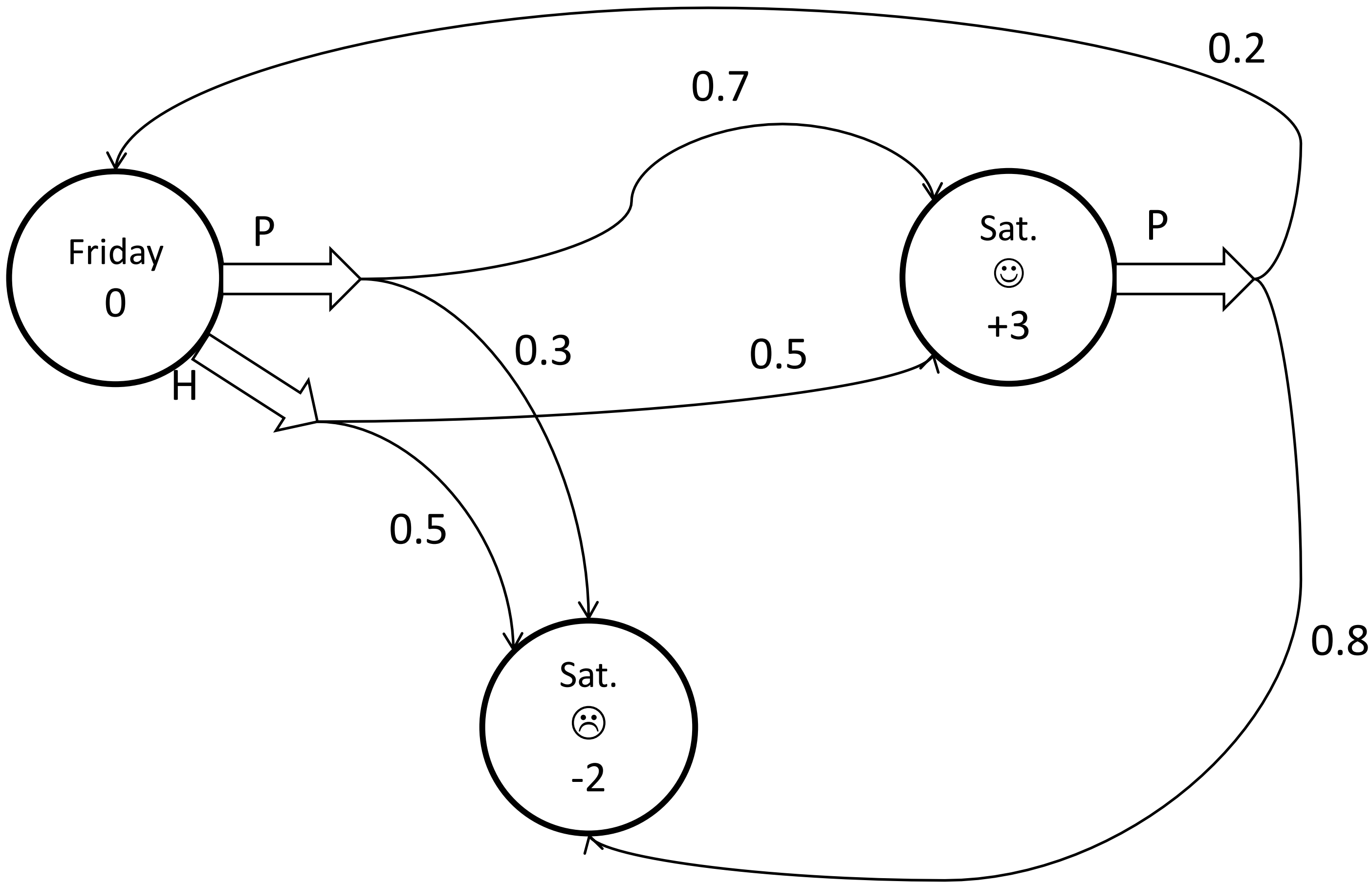
$$P(s_j | s_i, a)$$

  - *S*, *R* and γ are as before

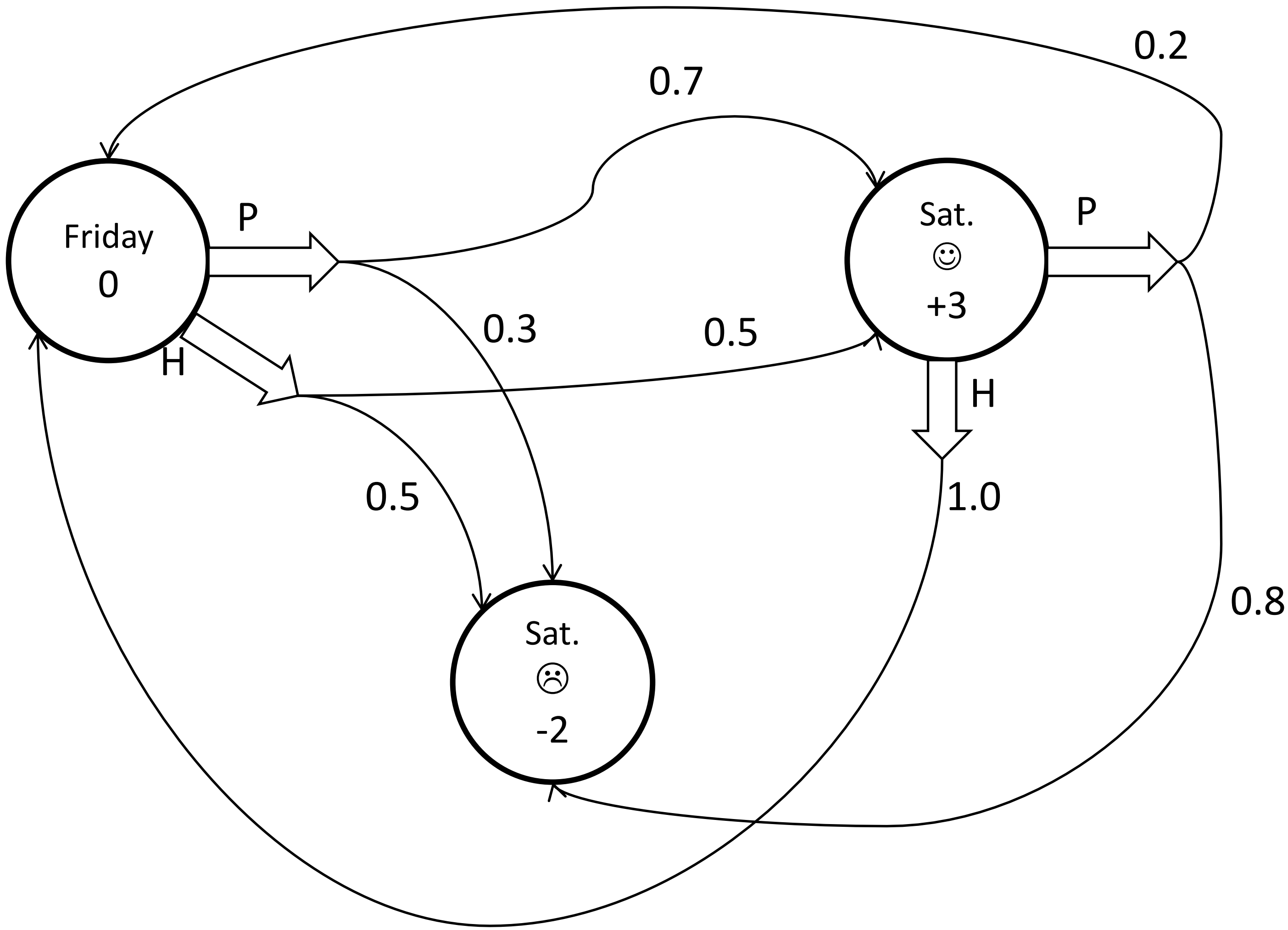
MDP Example



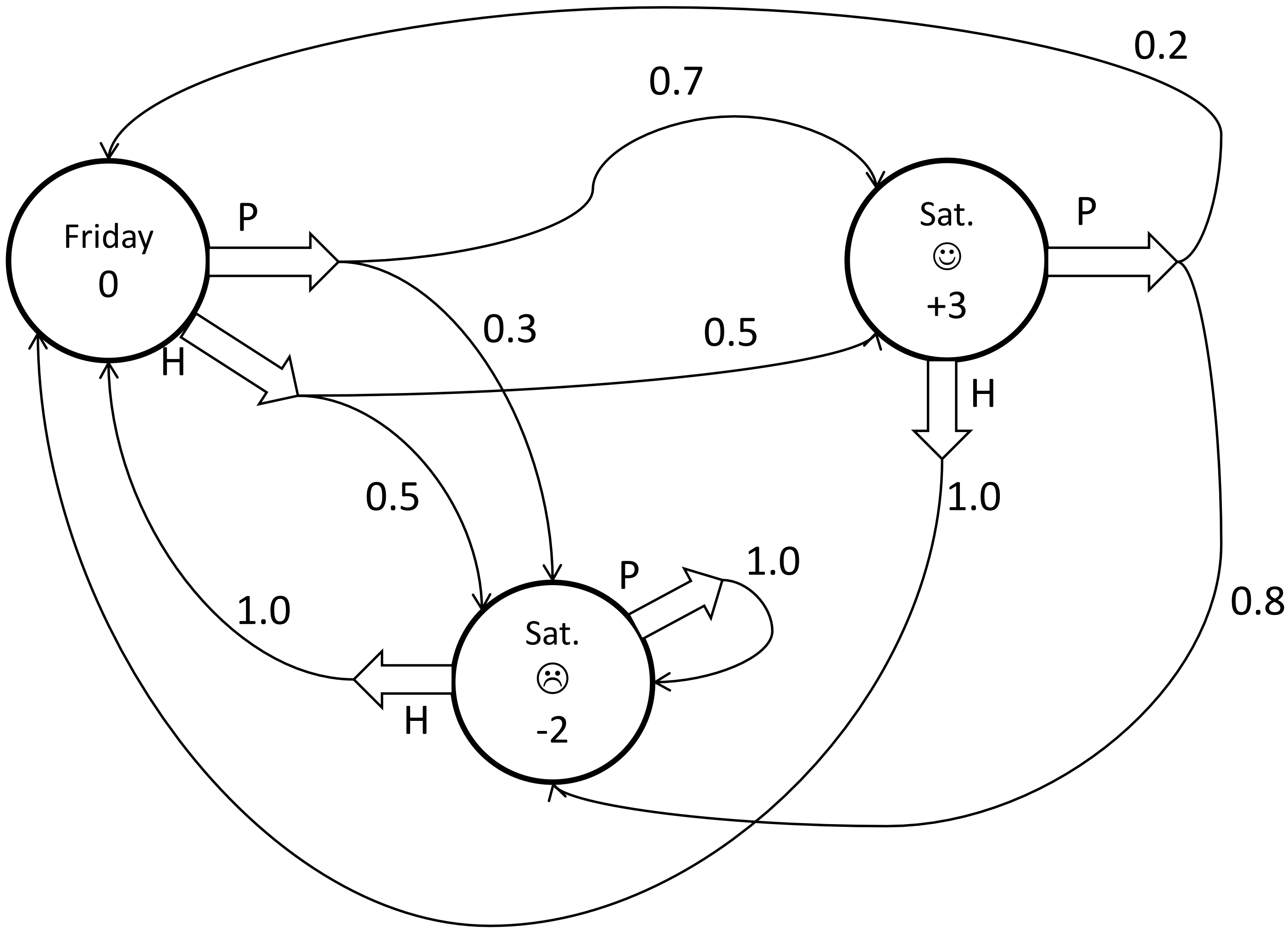
MDP Example



MDP Example



MDP Example



Making Decisions

- Main question: what actions should the agent take?

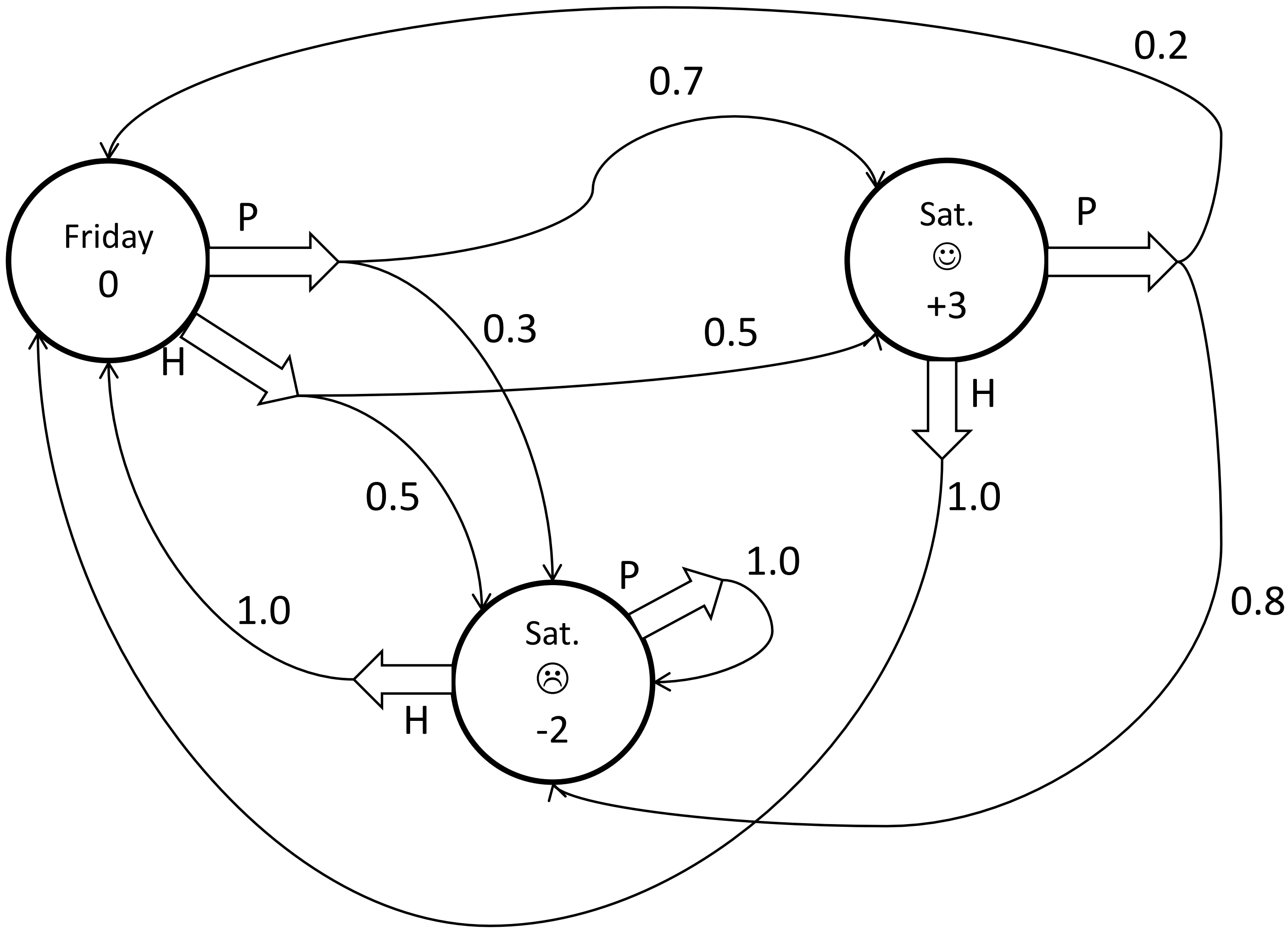
- We are searching for a *policy*:

$$\pi : S \mapsto A$$

- We want the *optimal* policy π^*, one that maximizes expected sum of discounted future rewards.

- Intuition: solve for V^*, and extract π^* from it.

MDP Example



Value Iteration for MDPs

- The Bellman Equation:

$$V^*(s_i) = R(s_i) + \max_a \gamma \sum_{j=1}^{n} p_{ij}^a V^*(s_j)$$

- Bellman equation for dynamic programming:

$$V^k(s_i) = R(s_i) + \max_a \gamma \sum_{j=1}^{n} p_{ij}^a V^{k-1}(s_j)$$

- Compute $V^0(s_i)$, $V^1(s_i)$, $V^2(s_i)$, … for all s_i until convergence

- Extract policy using greedy 1-step look-ahead:

$$\pi^*(s_i) = \operatorname{argmax}_a R(s_i) + \gamma \sum_{j=1}^{n} p_{ij}^a V^*(s_j)$$

Value Iteration for MDPs

- An MDP is a 5-tuple *(S, A, T, R, γ)*

VALUE-ITERATION(*S, A, T, R, γ*):

initialize value function $V^0(s_i) = R(s_i)$

$k \leftarrow 1$

loop until convergence:

for each s_i:

$$V^k(s_i) = R(s_i) + \max_a \gamma \sum_{j=1}^{n} p_{ij}^a V^{k-1}(s_j)$$

$k \leftarrow k + 1$

return $\pi^*(s_i) = \operatorname*{argmax}_a R(s_i) + \gamma \sum_{j=1}^{n} p_{ij}^a V^*(s_j)$

Policy Iteration

POLICY-ITERATION(*S, A, T, R, γ*):

initialize π^0 to a random policy

$k \leftarrow 1$

loop until convergence:

for each s_i:

$$V^{\pi^k}(s_i) = R(s_i) + \gamma \sum_{j=1}^{n} p_{ij}^{\pi^k(s_j)} V^{\pi^k}(s_j)$$

Policy evaluation

for each s_i:

$$\pi^{k+1}(s_i) = \operatorname{argmax}_a R(s_i) + \gamma \sum_{j=1}^{n} p_{ij}^{a} V^{\pi^k}(s_j)$$

Policy improvement

$k \leftarrow k + 1$

return π^*

A More Realistic Scenario

- An MDP is a 5-tuple (S, A, T, R, γ)

- When T and R are known:
  - An ***offline*** learning problem: agent can just think for a while "in its own head"
  - Can use value or policy iteration

- When T or R are unknown:
  - An ***online*** learning problem: agent needs to actually interact with the real world to get anywhere
  - Need other techniques

Q Learning

Q-Learning

- *Q*(s, a)*: expected sum of discounted future rewards if I take action *a* in state *s*, and act optimally thereafter.

- How does *Q*(s, a)* relate to *V*(s)* and *π*(s)*?

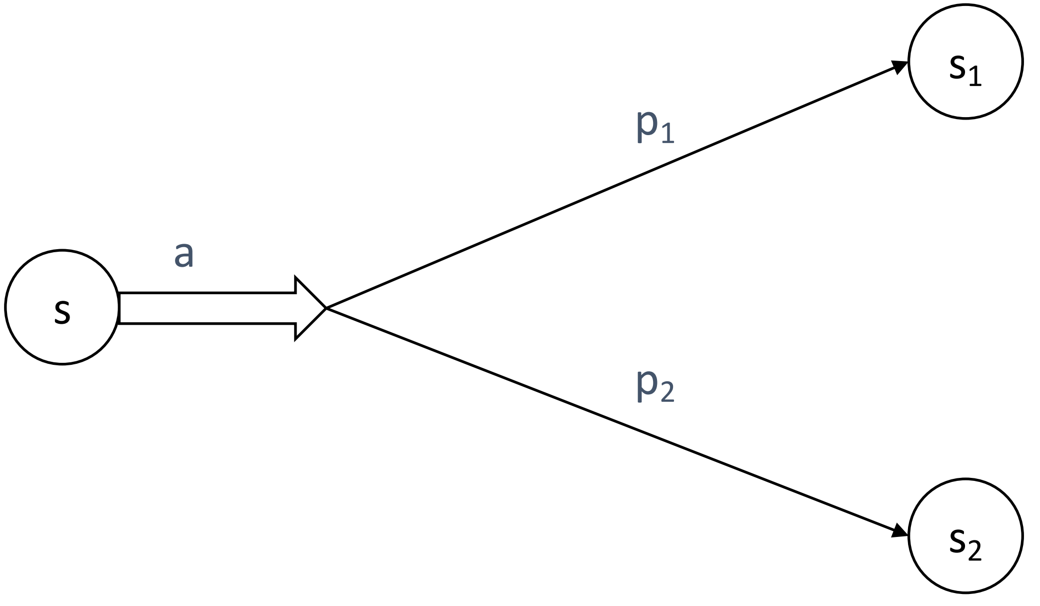
$$V^*(s) = \max_a Q^*(s, a)$$

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

- Can we recursively express *Q*(s, a)*?

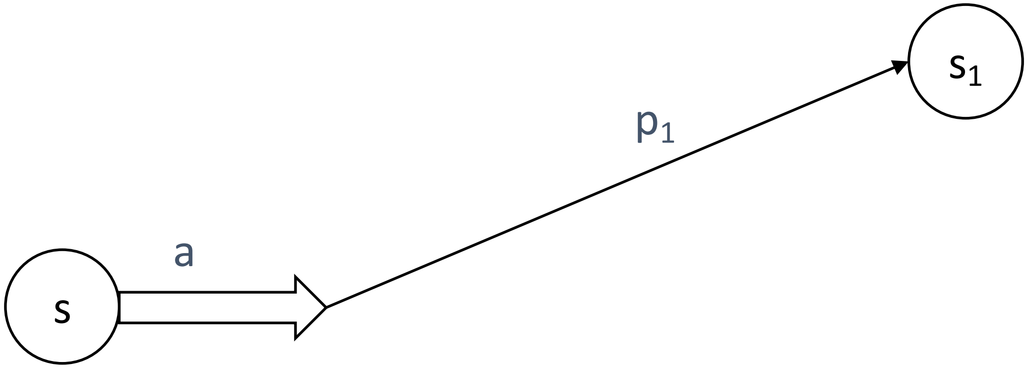
$$Q^*(s_i, a) = R(s_i) + \gamma \sum_j p_{ij}^a \max_{a'} Q^*(s_j, a')$$

Q-Learning



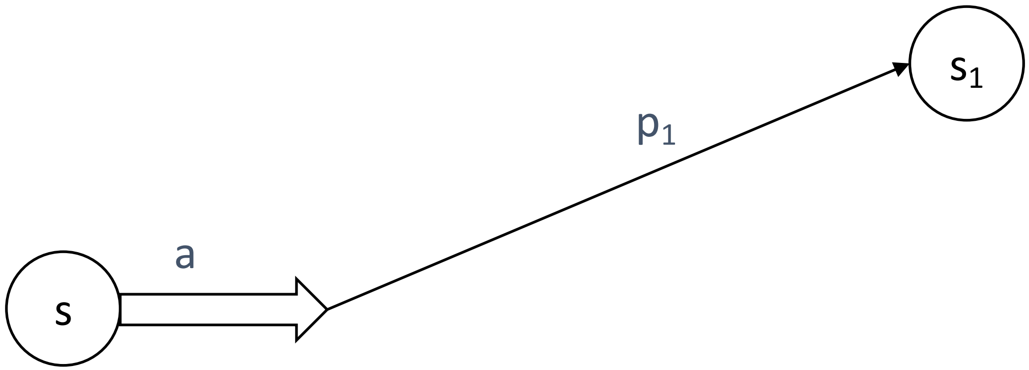
$$Q^*(s,a) = R(s) + \gamma\{p_1 \max_{a'} Q^*(s_1,a') + p_2 \max_{a''} Q^*(s_2,a'')\}$$

Q-Learning



$$R(s) + \gamma \max_{a'} Q^*(s_1, a')$$

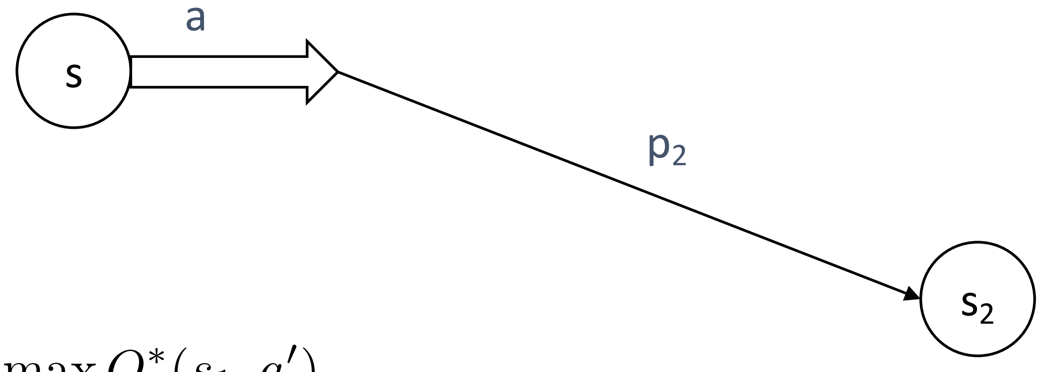
Q-Learning



$$R(s) + \gamma \max_{a'} Q^*(s_1, a')$$

$$R(s) + \gamma \max_{a'} Q^*(s_1, a')$$

Q-Learning

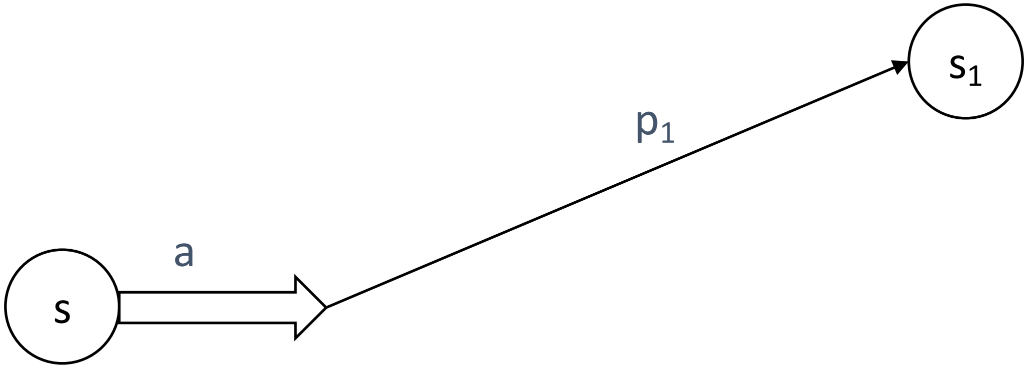


$$R(s) + \gamma \max_{a'} Q^*(s_1, a')$$

$$R(s) + \gamma \max_{a'} Q^*(s_1, a')$$

$$R(s) + \gamma \max_{a''} Q^*(s_2, a'')$$

Q-Learning



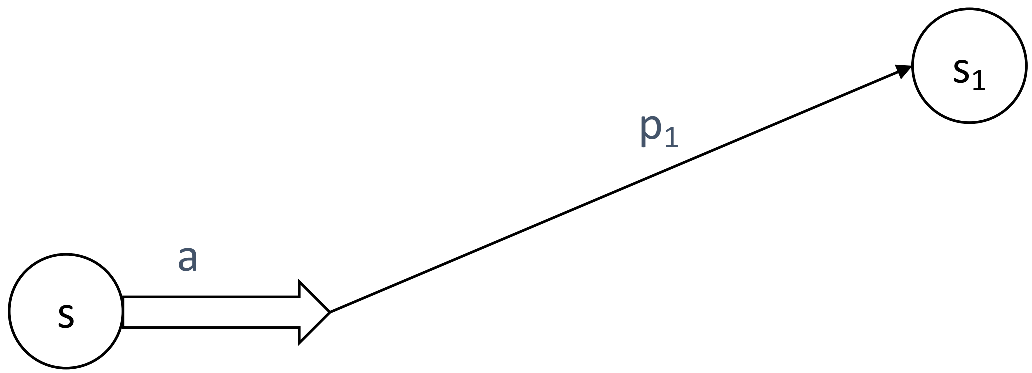
$$R(s) + \gamma \max_{a'} Q^*(s_1, a')$$

$$R(s) + \gamma \max_{a'} Q^*(s_1, a')$$

$$R(s) + \gamma \max_{a''} Q^*(s_2, a'')$$

$$R(s) + \gamma \max_{a'} Q^*(s_1, a')$$

Q-Learning



$$\left.\begin{array}{l} R(s) + \gamma \max_{a'} Q^*(s_1, a') \\ R(s) + \gamma \max_{a'} Q^*(s_1, a') \\ R(s) + \gamma \max_{a''} Q^*(s_2, a'') \\ R(s) + \gamma \max_{a'} Q^*(s_1, a') \end{array}\right\} R(s) + \gamma \left\{ \frac{3}{4} \max_{a'} Q^*(s_1, a') + \frac{1}{4} \max_{a''} Q^*(s_2, a'') \right\}$$

Q-Learning

- *Q*(s, a)*: expected sum of discounted future rewards if I take action *a* in state *s*, and act optimally thereafter.

- How does *Q*(s, a)* relate to *V*(s)* and *π*(s)*?

$$V^*(s) = \max_a Q^*(s,a)$$

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

- Can we recursively express *Q*(s, a)*?

$$Q^*(s_i, a) = R(s_i) + \gamma \sum_j p_{ij}^a \max_{a'} Q^*(s_j, a')$$

Q-Learning

- *Q*(s, a)*: expected sum of discounted future rewards if I take action *a* in state *s*, and act optimally thereafter.

- How does *Q*(s, a)* relate to *V*(s)* and *π*(s)*?

$$V^*(s) = \max_a Q^*(s, a)$$

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

- Q-learning update rule:

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

Deep Q-Networks (DQN)

- What if the state space is large?
  - Use a function approximator to represent *Q**

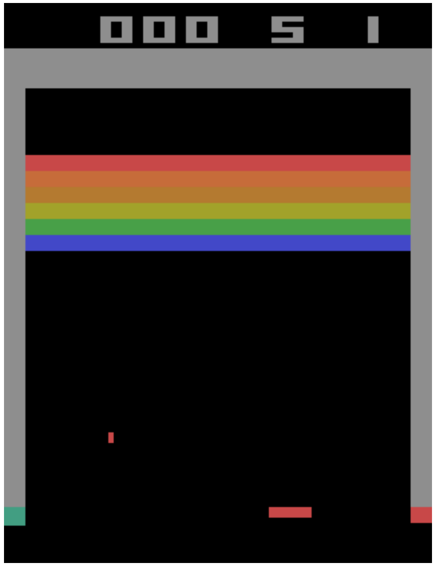
$$Q(s,a) \leftarrow Q(s,a) + \alpha \left\{ \underbrace{R(s) + \gamma \max_{a'} Q(s',a')}_{\text{“target”}} - Q(s,a) \right\}$$

Deep Q-Networks (DQN)

- What if the state space is large?
  - Use a function approximator to represent *Q**

- The DQN algorithm
  - Plays 49 different Atari games
  - Learns from raw pixel inputs
  - Uses a deep CNN for predicting *Q**

- Other important techniques:
  - *Experience replay*
  - *Target network*
  - And a slew of other tricks…



Policy Gradient Methods

An Alternative Approach

- If we care about the policy, why not learn it directly?

- Pros:
  - More reliable/stable: we're optimizing what we care about
  - Works with continuous action spaces

- Cons:
  - Less sample efficient (learning tends to happen *on-policy*)

- General idea:
  - Use a *stochastic* policy

$$a \sim \pi(.|s)$$

  - Use a function approximator (deep neural network) to represent policy
  - Use gradient-based optimization: encourage good actions, discourage bad ones

The REINFORCE Algorithm

- Define the *return*:

$$\mathcal{R}(\tau) = \sum_{t=0}^{\infty} \gamma^t R(s_t)$$

- Assume a parameterized stochastic policy π_θ
- Goal: "Find the parameters that maximize expected return"

$$\underset{\theta}{\operatorname{argmax}}\, J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\mathcal{R}(\tau)\right]$$

- How?

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$$

Deriving the Policy Gradient

The REINFORCE Algorithm

- To summarize, in order to perform:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$$

- We estimate:

$$\hat{g} = \frac{1}{n} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t | s_t) \mathcal{R}(\tau)$$

- And perform:

$$\theta \leftarrow \theta + \alpha \hat{g}$$

- Enhancements:
  - Lower the variance in gradient estimates
  - Use *trust region* updates: TRPO, PPO

Summary

- Have a sequential decision or control problem? RL may help
- Deep RL = classic RL algorithms + deep neural networks
  - Can be challenging to use/tune
  - PPO is a good default
- High-quality implementations are available