
Machine Learning

Topic 15: Reinforcement Learning

(thanks in part to Bill Smart at Washington University in St. Louis)

Learning Types

- Supervised learning:
 - (Input, output) pairs of the function to be learned can be perceived or are given.

Back-propagation in Neural Nets

- Unsupervised Learning:
 - No information about desired outcomes given

K-means clustering

- Reinforcement learning:
 - Reward or punishment for actions

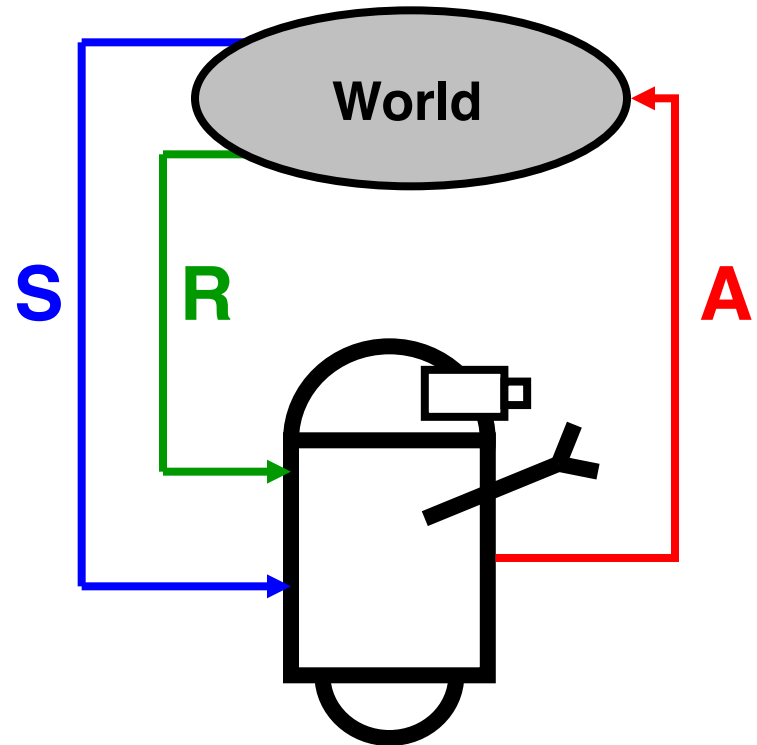
Q-Learning

Reinforcement Learning

- Task
 - Learn how to behave to achieve a goal
 - Learn through experience from trial and error
- Examples
 - Game playing: The agent knows when it wins, but doesn't know the appropriate action in each state along the way
 - Control: a traffic system can measure the delay of cars, but not know how to decrease it.

Basic RL Model

1. Observe state, s_t
2. Decide on an action, a_t
3. Perform action
4. Observe new state, s_{t+1}
5. Observe reward, r_{t+1}
6. Learn from experience
7. Repeat



•Goal: Find a control policy that will maximize the observed rewards over the lifetime of the agent

An Example: Gridworld

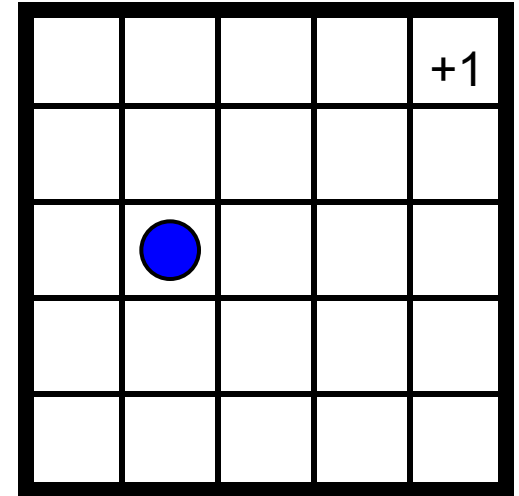
- Canonical RL domain

States are grid cells

4 actions: N, S, E, W

Reward for entering top right cell

-0.01 for every other move

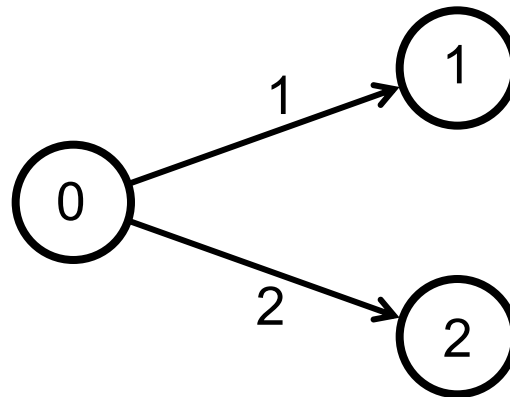


Mathematics of RL

- Before we talk about RL, we need to cover some background material
 - Simple decision theory
 - Markov Decision Processes
 - Value functions
 - Dynamic programming

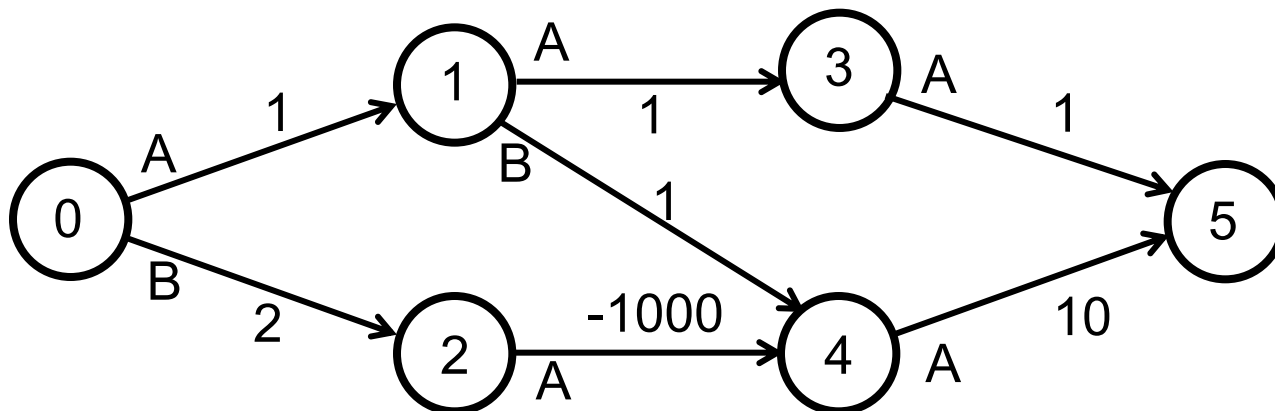
Making Single Decisions

- Single decision to be made
 - Multiple discrete actions
 - Each action has a reward associated with it
- Goal is to maximize reward
 - Not hard: just pick the action with the largest reward
- State 0 has a value of 2
 - Sum of rewards from taking the best action from the state



Markov Decision Processes

- We can generalize the previous example to multiple sequential decisions
 - Each decision affects subsequent decisions
- This is formally modeled by a Markov Decision Process (MDP)



Markov Decision Processes

- Formally, a MDP is
 - A set of states, $S = \{s_1, s_2, \dots, s_n\}$
 - A set of actions, $A = \{a_1, a_2, \dots, a_m\}$
 - A reward function, $R: S \times A \times S \rightarrow \mathbb{R}$
 - A transition function, $P_{ij}^a = P(s_{t+1} = j | s_t = i, a_t = a)$
 - Sometimes $T: S \times A \rightarrow S$
- We want to learn a policy, $\pi: S \rightarrow A$
 - Maximize sum of rewards we see over our lifetime

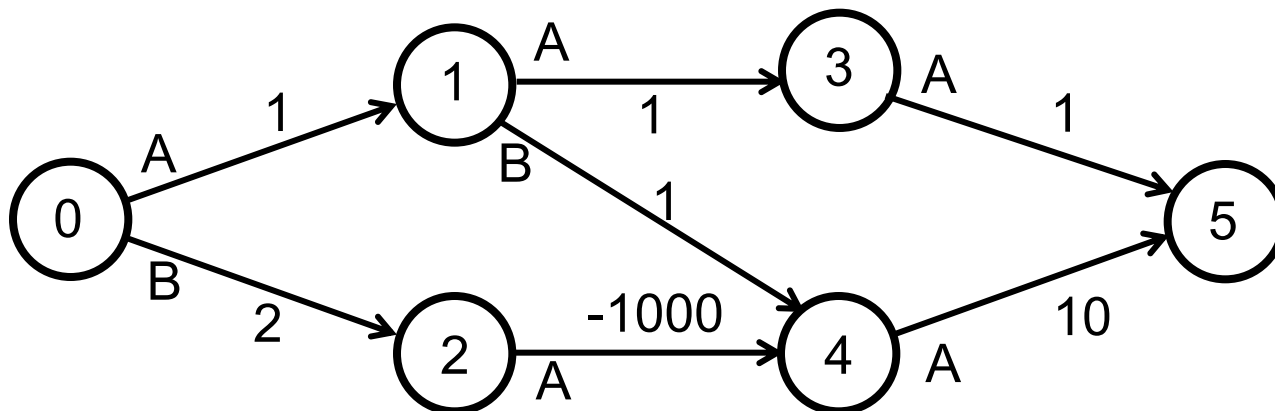
Policies

- A policy $\pi(s)$ returns what action to take in state s .
- There are 3 policies for this MDP

Policy 1: $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$

Policy 2: $0 \rightarrow 1 \rightarrow 4 \rightarrow 5$

Policy 3: $0 \rightarrow 2 \rightarrow 4 \rightarrow 5$



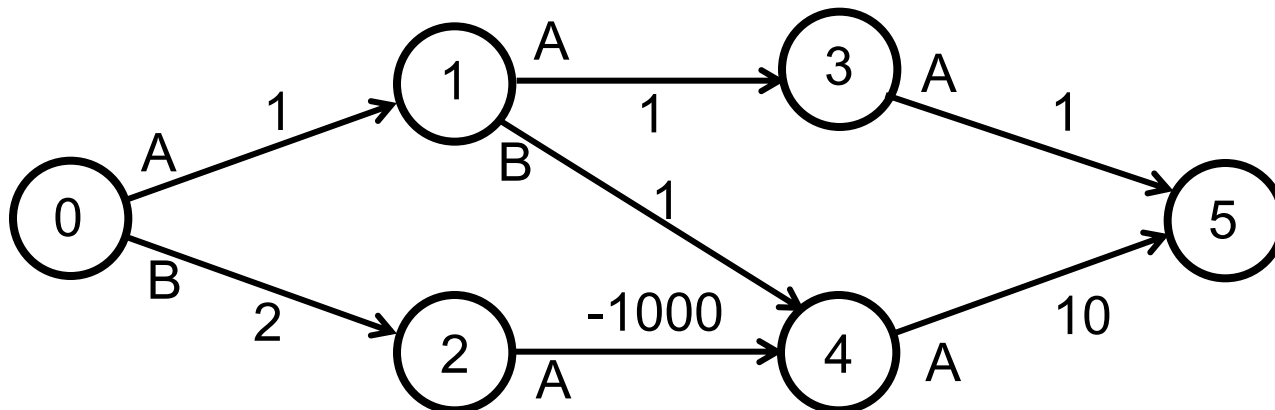
Comparing Policies

- Which policy is best?
- Order them by how much reward they see

Policy 1: $0 \rightarrow 1 \rightarrow 3 \rightarrow 5 = 1 + 1 + 1 = 3$

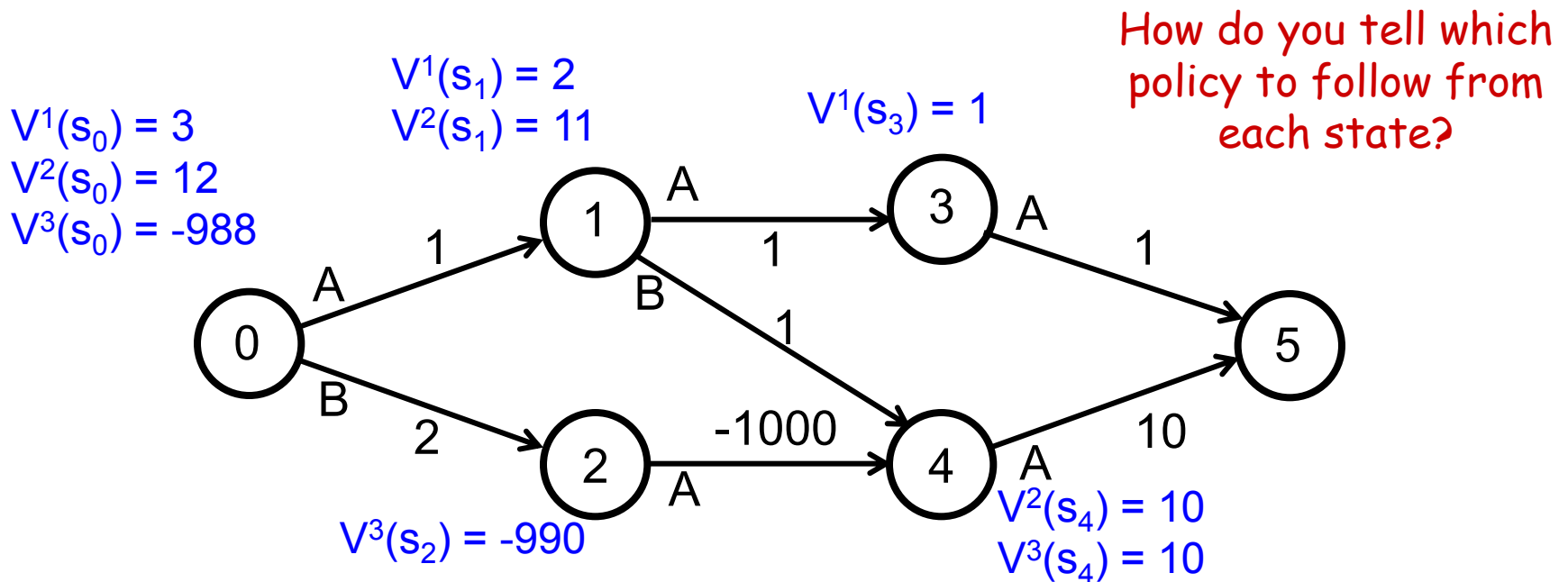
Policy 2: $0 \rightarrow 1 \rightarrow 4 \rightarrow 5 = 1 + 1 + 10 = 12$

Policy 3: $0 \rightarrow 2 \rightarrow 4 \rightarrow 5 = 2 - 1000 + 10 = -988$



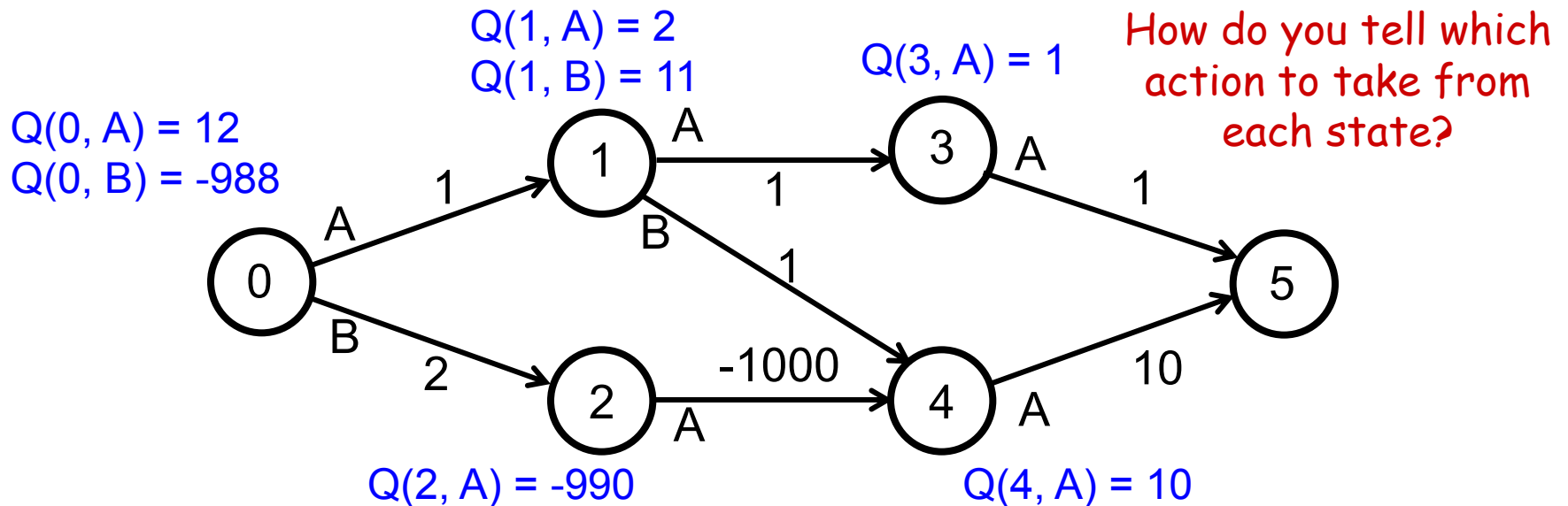
Value Functions

- We can associate a value with each state
 - For a fixed policy
 - How good is it to run policy π from that state s
 - This is the state value function, V



Q Functions

- Define value without specifying the policy
 - Specify the value of taking action A from state S and then performing optimally, thereafter



Value Functions

- So, we have two value functions

$$V^{\pi}(s) = R(s, \pi(s), s') + V^{\pi}(s')$$

s' is the
next state

a' is the
next action

$$Q(s, a) = R(s, a, s') + \max_{a'} Q(s', a')$$

- Both have the same form
 - Next reward plus the best I can do from the next state

Value Functions

- These can be extended to probabilistic actions (for when the results of an action are not certain, or when a policy is probabilistic)

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

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

Getting the Policy

- If we have the value function, then finding the optimal policy, $\pi^*(s)$, is easy...just find the policy that maximized value

$$\pi^*(s) = \arg \max_a (R(s, a, s') + V^\pi(s'))$$

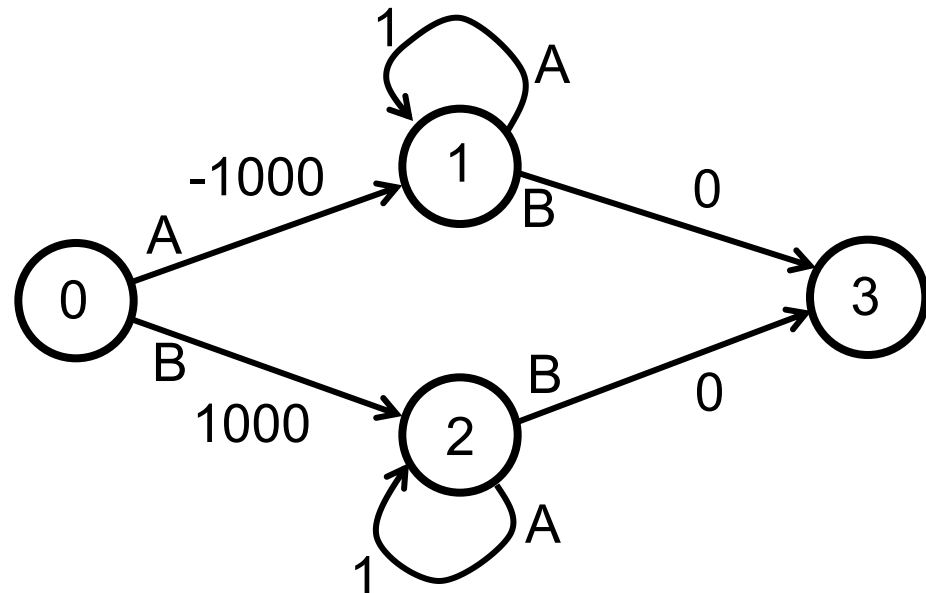
$$\pi^*(s) = \arg \max_a Q(s, a)$$

Problems with Our Functions

- Consider this MDP
 - Number of steps is now unlimited because of loops
 - Value of states 1 and 2 is infinite for some policies

$$\begin{aligned}Q(1, A) &= 1 + Q(1, A) \\&= 1 + 1 + Q(1, A) \\&= 1 + 1 + 1 + Q(1, A) \\&= \dots\end{aligned}$$

- This is bad
 - All policies with a non-zero reward cycle have infinite value



Better Value Functions

- Introduce the **discount factor** γ , to get around the problem of infinite value
 - Three interpretations
 - Probability of living to see the next time step
 - Measure of the uncertainty inherent in the world
 - Makes the mathematics work out nicely

Assume $0 \leq \gamma \leq 1$

$$V^\pi(s) = R(s, \pi(s), s') + \gamma V^\pi(s')$$

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

Better Value Functions

Value now depends
on the discount, γ

$$Q(0, A) = -1000 + \frac{\gamma}{1-\gamma}$$

$$Q(1, B) = 1000 + \frac{\gamma}{1-\gamma}$$

$$Q(1, A) = \frac{1}{1-\gamma}$$

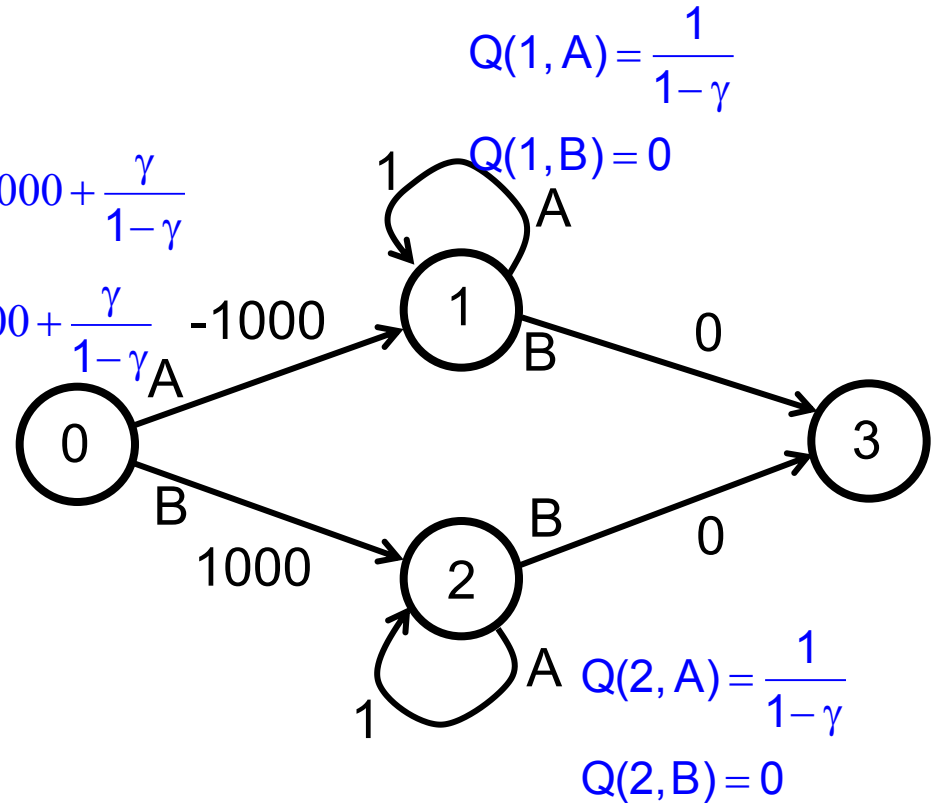
$$Q(1, B) = 0$$

- Optimal Policy:

$$\pi(0) = B$$

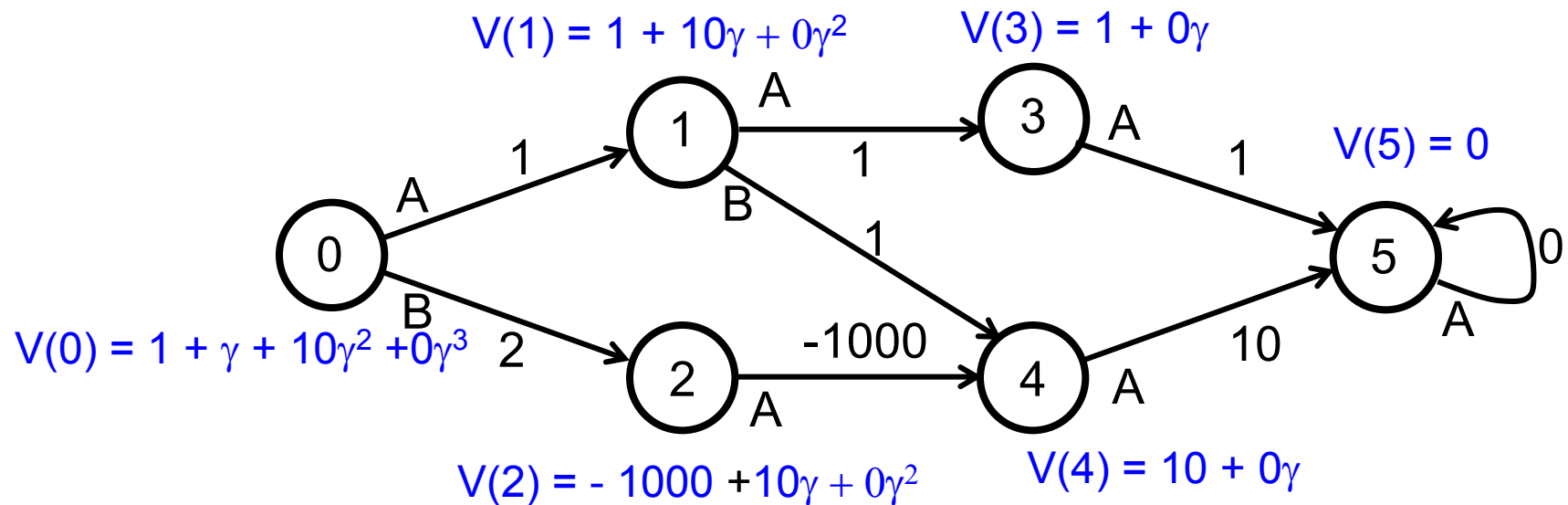
$$\pi(1) = A$$

$$\pi(2) = A$$



Dynamic Programming

- Given the complete MDP model, we can compute the optimal value function directly

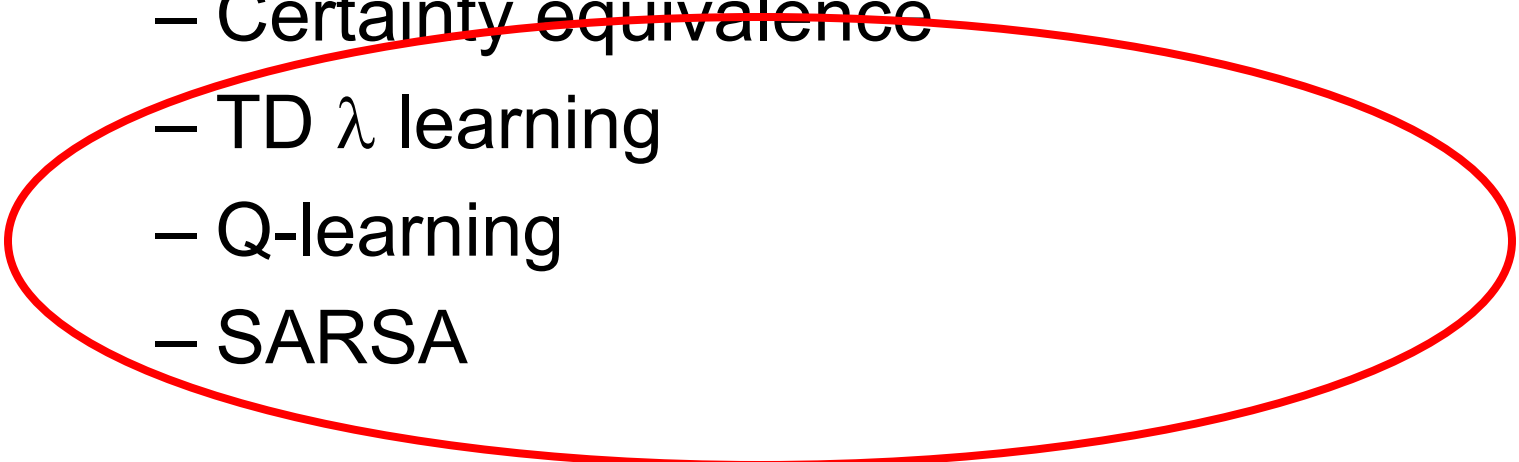


[Bertsekas, 87, 95a, 95b]

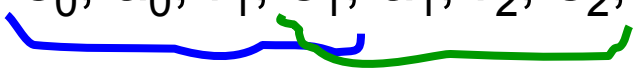
Reinforcement Learning

- What happens if we don't have the whole MDP?
 - We know the states and actions
 - We don't have the system model (transition function) or reward function
- We're only allowed to sample from the MDP
 - Can observe experiences (s, a, r, s')
 - Need to perform actions to generate new experiences
- This is Reinforcement Learning (RL)
 - Sometimes called Approximate Dynamic Programming (ADP)

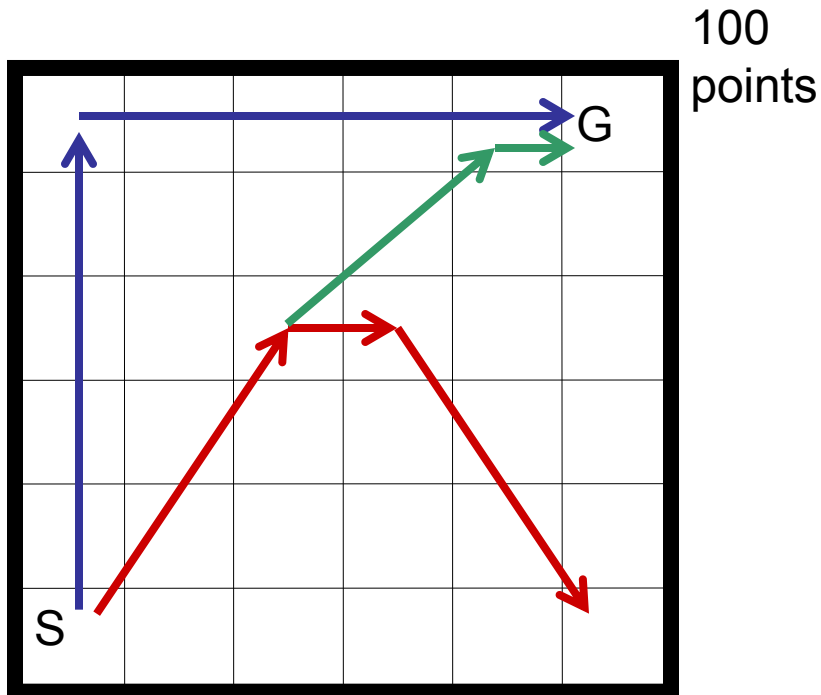
Learning Value Functions

- We still want to learn a value function
 - We're forced to approximate it iteratively
 - Based on direct experience of the world
 - Four main algorithms
 - Certainty equivalence
 - TD λ learning
 - Q-learning
 - SARSA
- 

Certainty Equivalence

- Collect experience by moving through the world
 - $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, r_4, s_4, a_4, r_5, s_5, \dots$ 
- Use these to estimate the underlying MDP
 - Transition function, $T: S \times A \rightarrow S$
 - Reward function, $R: S \times A \times S \rightarrow \mathbb{R}$
- Compute the optimal value function for this MDP
- And then compute the optimal policy from it

How are we going to do this?



- Reward whole policies?
 - That could be a pain
- What about incremental rewards?
 - Everything has a reward of 0 except for the goal
- Now what???

Exploration vs. Exploitation

- We want to pick good actions most of the time, but also do some exploration
- Exploring means we can learn better policies
- But, we want to balance known good actions with exploratory ones
- This is called the **exploration/exploitation** problem

On-Policy vs. Off Policy

- On-policy algorithms
 - Final policy is influenced by the exploration policy
 - Generally, the exploration policy needs to be “close” to the final policy
 - Can get stuck in local maxima
- Off-policy algorithms
 - Final policy is independent of exploration policy
 - Can use arbitrary exploration policies
 - Will not get stuck in local maxima

*Given enough
experience*

Picking Actions

ϵ -greedy

- Pick best (greedy) action with probability ϵ
- Otherwise, pick a random action
- Boltzmann (Soft-Max)
 - Pick an action based on its Q-value

$$P(a | s) = \frac{e^{\left(\frac{Q(s, a)}{\tau}\right)}}{\sum_{a'} e^{\left(\frac{Q(s, a')}{\tau}\right)}}$$

...where τ is the “temperature”

TD(λ)

- TD-learning estimates the value function directly
 - Don't try to learn the underlying MDP [Sutton, 88]
- Keep an estimate of $V^\pi(s)$ in a table
 - Update these estimates as we gather more experience
 - Estimates depend on exploration policy, π
 - TD is an on-policy method

TD(0)-Learning Algorithm

- Initialize $V^\pi(s)$ to 0
- Make a (possibly randomly created) policy π
- For each ‘episode’ (episode = series of actions)
 1. Observe state s
 2. Perform action according to the policy $\pi(s)$
 3. $V(s) \leftarrow (1-\alpha)V(s) + \alpha[r + \gamma V(s')]$
 4. $s \leftarrow s'$
 5. Repeat until out of actions
- Update policy given newly learned values
- Start a new episode

Note: this formulation is from Sutton & Barto’s “Reinforcement Learning”

r = reward
 α = learning rate
 γ = discount factor

(Tabular) TD-Learning Algorithm

1. Initialize $V^\pi(s)$ to 0, and $e(s) = 0 \forall s$
2. Observe state, s
3. Perform action according to the policy $\pi(s)$
4. Observe new state, s' , and reward, r
5. $\delta \leftarrow r + \gamma V^\pi(s') - V^\pi(s)$
6. $e(s) \leftarrow e(s) + 1$
7. For all states j
 $V^\pi(s) \leftarrow V^\pi(s) + \alpha \delta e(j)$
 $e(j) \leftarrow \gamma \lambda e(j)$
8. Go to 2

γ = future returns
discount factor
 λ = eligibility discount
 α = learning rate

TD-Learning

- $V^\pi(s)$ is guaranteed to converge to $V^*(s)$
 - After an infinite number of experiences
 - If we decay the learning rate

$$\sum_{t=0}^{\infty} \alpha_t = \infty \quad \sum_{t=0}^{\infty} \alpha_t^2 < \infty$$

$$\alpha_t = \frac{c}{c+t} \quad \text{will work}$$

- In practice, we often don't need value convergence
 - Policy convergence generally happens sooner

SARSA

- SARSA iteratively approximates the state-action value function, Q
 - Like Q-learning, SARSA learns the policy and the value function simultaneously
- Keep an estimate of $Q(s, a)$ in a table
 - Update these estimates based on experiences
 - Estimates depend on the exploration policy
 - SARSA is an on-policy method
 - Policy is derived from current value estimates

SARSA Algorithm

1. Initialize $Q(s, a)$ to small random values, $\forall s, a$
 2. Observe state, s
 3. $a \leftarrow \pi(s)$ (pick action according to policy)
 4. Observe next state, s' , and reward, r
 5. $Q(s, a) \leftarrow (1-\alpha)Q(s, a) + \alpha(r + \gamma Q(s', \pi(s')))$
 6. Go to 2
- $0 \leq \alpha \leq 1$ is the learning rate
 - We should decay this, just like TD

Q-Learning

[Watkins & Dayan, 92]

- Q-learning iteratively approximates the state-action value function, Q
 - We won't estimate the MDP directly
 - Learns the value function and policy simultaneously
- Keep an estimate of $Q(s, a)$ in a table
 - Update these estimates as we gather more experience
 - Estimates do not depend on exploration policy
 - Q-learning is an off-policy method

Q-Learning Algorithm

1. Initialize $Q(s, a)$ to small random values, $\forall s, a$
(what if you make them 0? What if they are big?)
2. Observe state, s
3. Randomly (or ϵ greedy) pick action, a
4. Observe next state, s' , and reward, r
5. $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$
6. $s \leftarrow s'$
7. Go to 2

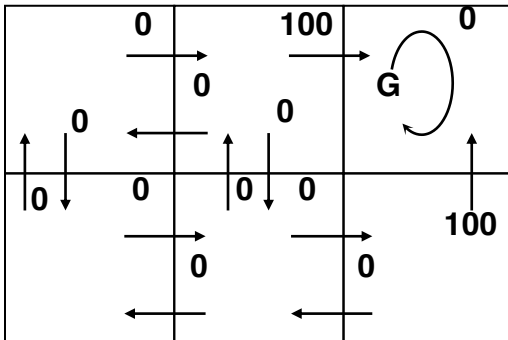
$0 \leq \alpha \leq 1$ is the learning rate & we should decay α , just like in TD

Note: this formulation is from Sutton & Barto's "Reinforcement Learning"

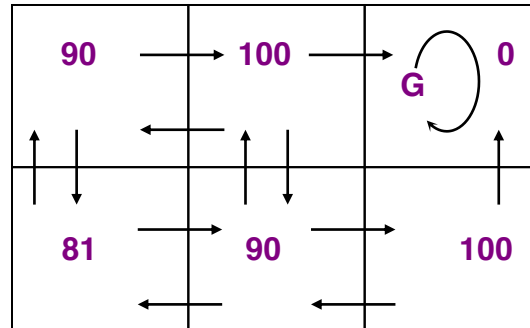
This is not identical to Mitchell's formulation, which does not use learning rate.

Q-learning

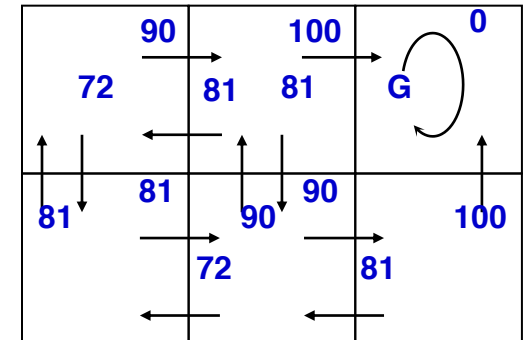
- Q-learning, learns the expected utility of taking a particular action **a** in state **s**



$r(\text{state}, \text{action})$
immediate reward values



$V^*(\text{state})$ values



$Q(\text{state}, \text{action})$ values

Convergence Guarantees

- The convergence guarantees for RL are “in the limit”
 - The word “infinite” crops up several times
- Don't let this put you off
 - Value convergence is different than policy convergence
 - We're more interested in policy convergence
 - If one action is significantly better than the others, policy convergence will happen relatively quickly

Rewards

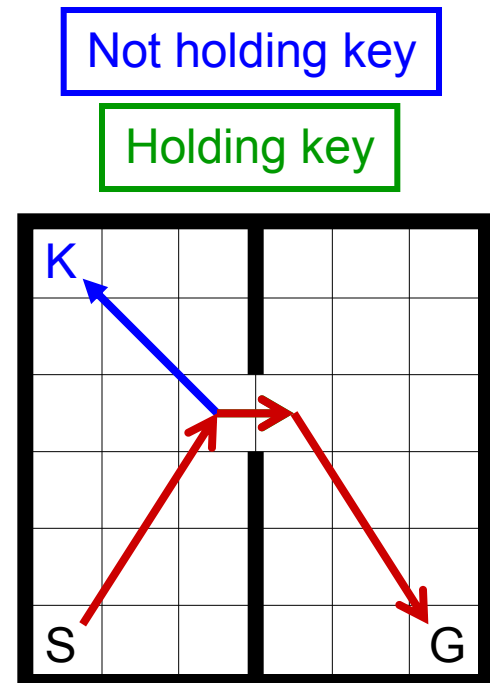
- Rewards measure how well the policy is doing
 - Often correspond to events in the world
 - Current load on a machine
 - Reaching the coffee machine
 - Program crashing
 - Everything else gets a 0 reward
- Things work better if the rewards are incremental
 - For example, distance to goal at each step
 - These reward functions are often hard to design

*These are
sparse rewards*

*These are
dense rewards*

The Markov Property

- RL needs a set of states that are Markov
 - Everything you need to know to make a decision is included in the state
 - Not allowed to consult the past
- Rule-of-thumb
 - If you can calculate the reward function from the state without any additional information, you're OK



But, What's the Catch?

- RL will solve all of your problems, but
 - We need lots of experience to train from
 - Taking random actions can be dangerous
 - It can take a long time to learn
 - Not all problems fit into the MDP framework

Learning Policies Directly

- An alternative approach to RL is to reward whole policies, rather than individual actions
 - Run whole policy, then receive a single reward
 - Reward measures success of the whole policy
- If there are a small number of policies, we can exhaustively try them all
 - However, this is not possible in most interesting problems

Policy Gradient Methods

- Assume that our policy, p , has a set of n real-valued parameters, $q = \{q_1, q_2, q_3, \dots, q_n\}$
 - Running the policy with a particular q results in a reward, r_q
 - Estimate the reward gradient, $\frac{\partial R}{\partial \theta_i}$, for each q_i

$$\theta_i \leftarrow \theta_i + \alpha \frac{\partial R}{\partial \theta_i}$$

This is another
learning rate

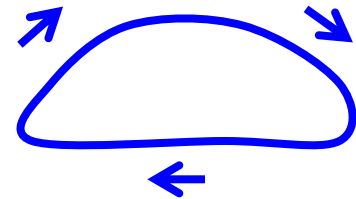
Policy Gradient Methods

- This results in hill-climbing in policy space
 - So, it's subject to all the problems of hill-climbing
 - But, we can also use tricks from search, like random restarts and momentum terms
- This is a good approach if you have a parameterized policy
 - Typically faster than value-based methods
 - “Safe” exploration, if you have a good policy
 - Learns locally-best parameters *for that policy*

An Example: Learning to Walk


[Kohl & Stone, 04]

- RoboCup legged league
 - Walking quickly is a *big* advantage
- Robots have a parameterized gait controller
 - 11 parameters
 - Controls step length, height, etc.
- Robots walk across soccer pitch and are timed
 - Reward is a function of the time taken



An Example: Learning to Walk

- Basic idea
 1. Pick an initial $\theta = \{\theta_1, \theta_2, \dots, \theta_{11}\}$
 2. Generate N testing parameter settings by perturbing θ
 $\theta^j = \{\theta_1 + \delta_1, \theta_2 + \delta_2, \dots, \theta_{11} + \delta_{11}\}, \quad \delta_i \in \{-\varepsilon, 0, \varepsilon\}$
 3. Test each setting, and observe rewards
 $\theta^j \rightarrow r_j$
 4. For each $\theta_i \in \theta$
Calculate $\theta_1^+, \theta_1^0, \theta_1^-$ and set $\theta'_i \leftarrow \theta_i + \begin{cases} \delta & \text{if } \theta_i^+ \text{ largest} \\ 0 & \text{if } \theta_i^0 \text{ largest} \\ -\delta & \text{if } \theta_i^- \text{ largest} \end{cases}$
 5. Set $\theta \leftarrow \theta'$, and go to 2



Average reward
when $q_i^n = q_i - d_i$

An Example: Learning to Walk



Initial



Final

<http://utopia.utexas.edu/media/features/av.qtl>

Video: Nate Kohl & Peter Stone, UT Austin

Value Function or Policy Gradient?

- When should I use policy gradient?
 - When there's a parameterized policy
 - When there's a high-dimensional state space
 - When we expect the gradient to be smooth
- When should I use a value-based method?
 - When there is no parameterized policy
 - When we have no idea how to solve the problem