Lecture 5: Monte Carlo

Recall: Monte Carlo Methods

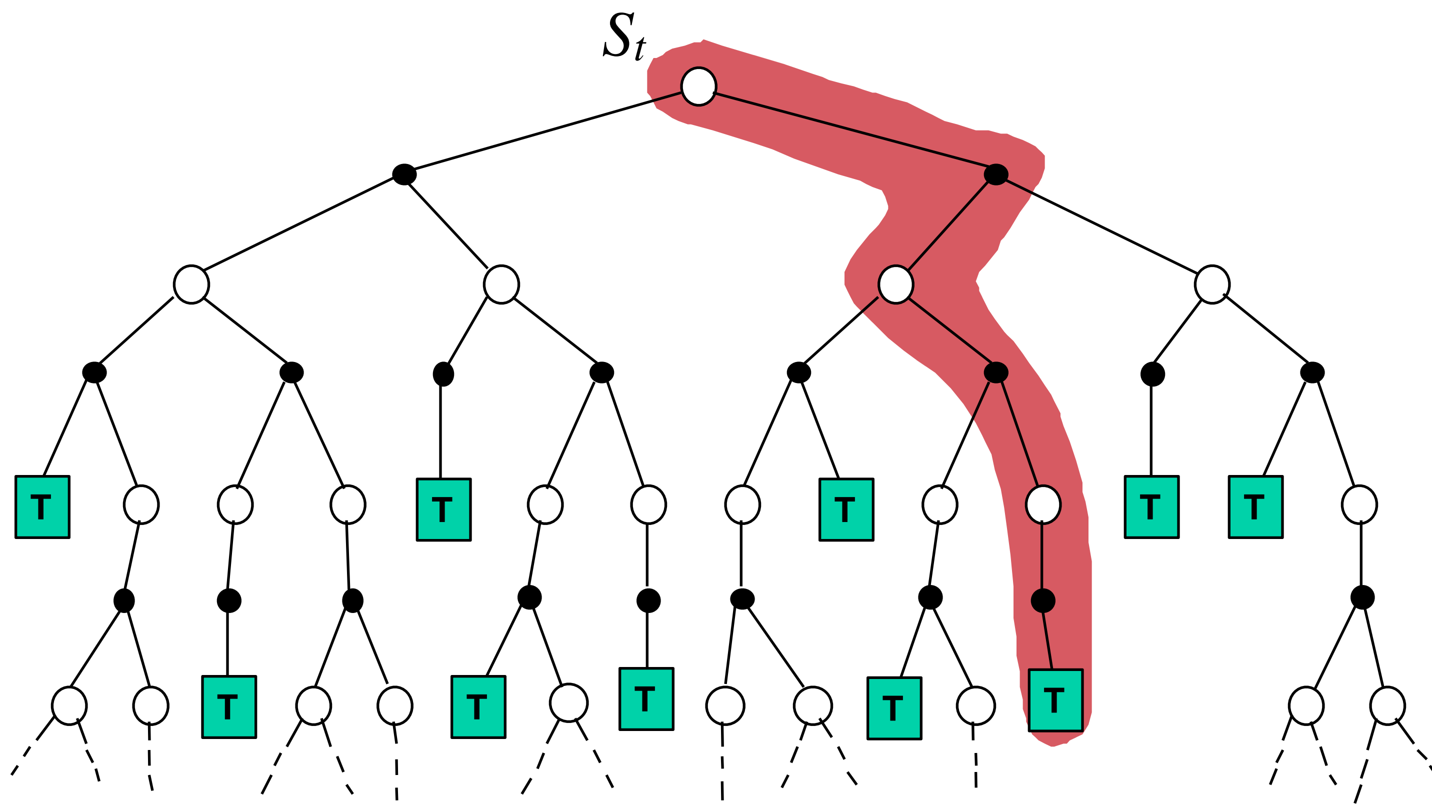
- Learning methods for sequential decision making
  Experience → values, policy
- Monte Carlo methods learn from *complete* sample returns
  - Defined for episodic tasks (in the book)

$$G_t = R_{t+2} + R_{t+2} + \ldots + R_T = \sum_{k=1}^{T-t} R_{t+k}$$

- Like an associative version of a bandit method: associate return to state or state-action pair

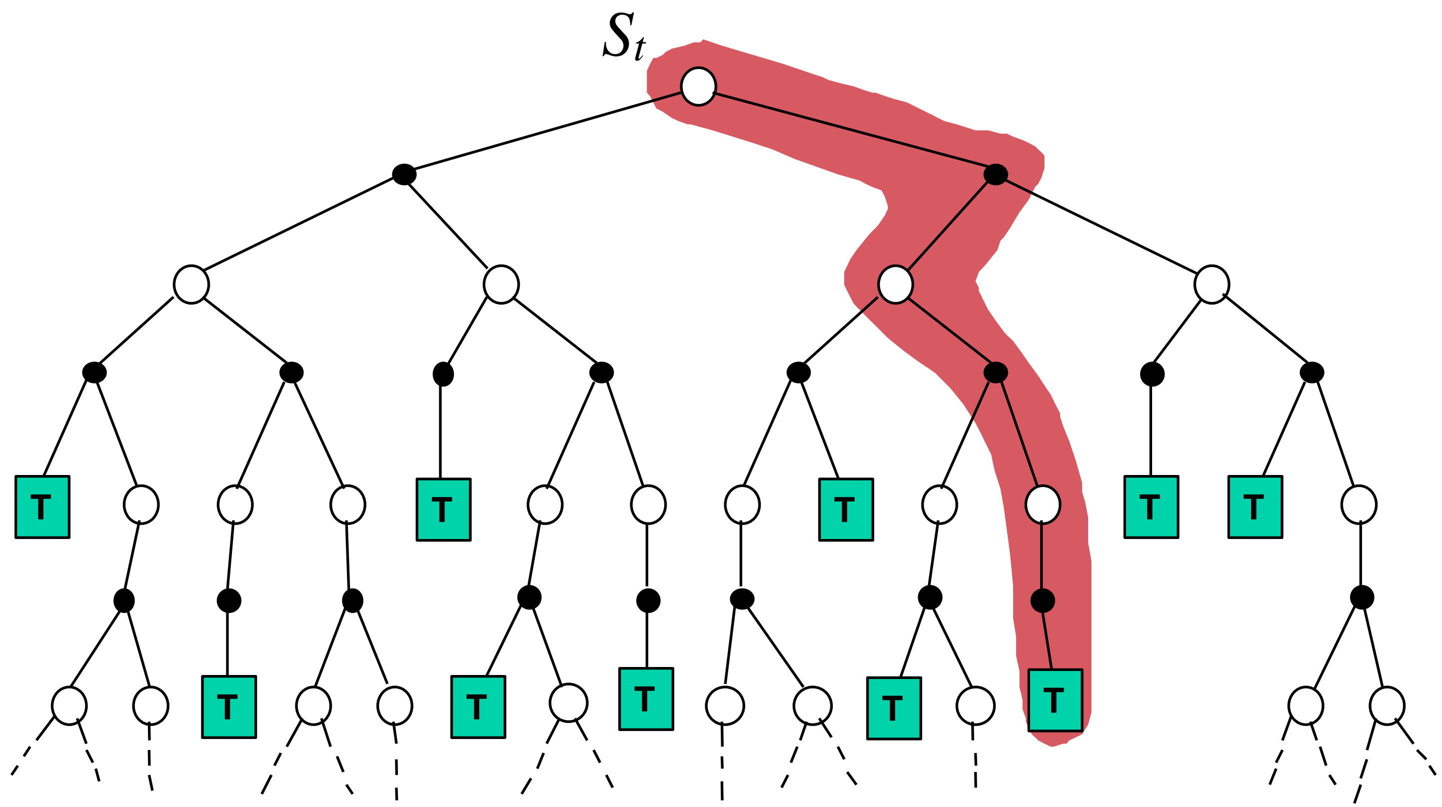
Recall: Simple Monte Carlo

$$V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right]$$



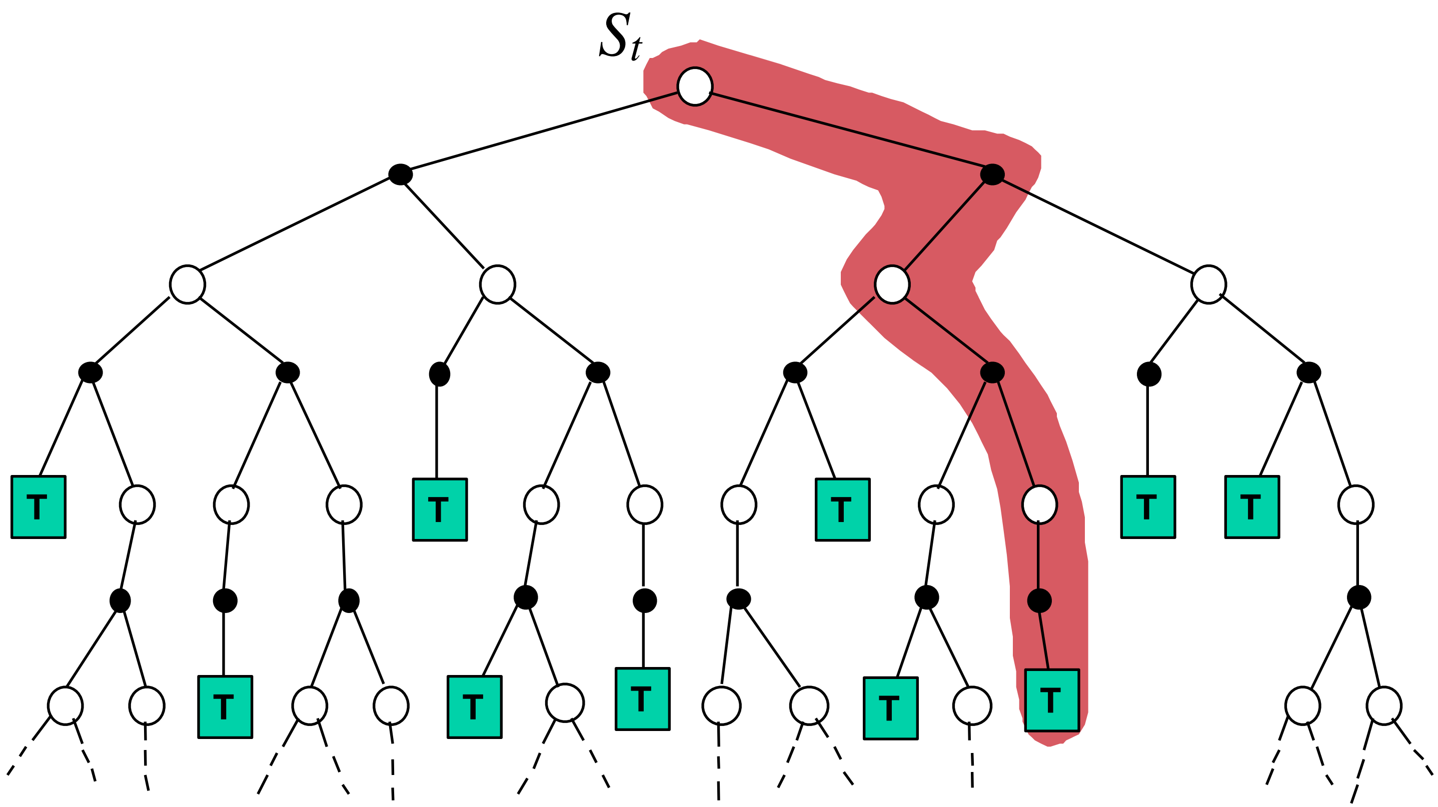
Example: Chess

S: board position;
Action: legal move;
Reward: at the end of the game, +1, -1 or 0 (win, loss, draw)



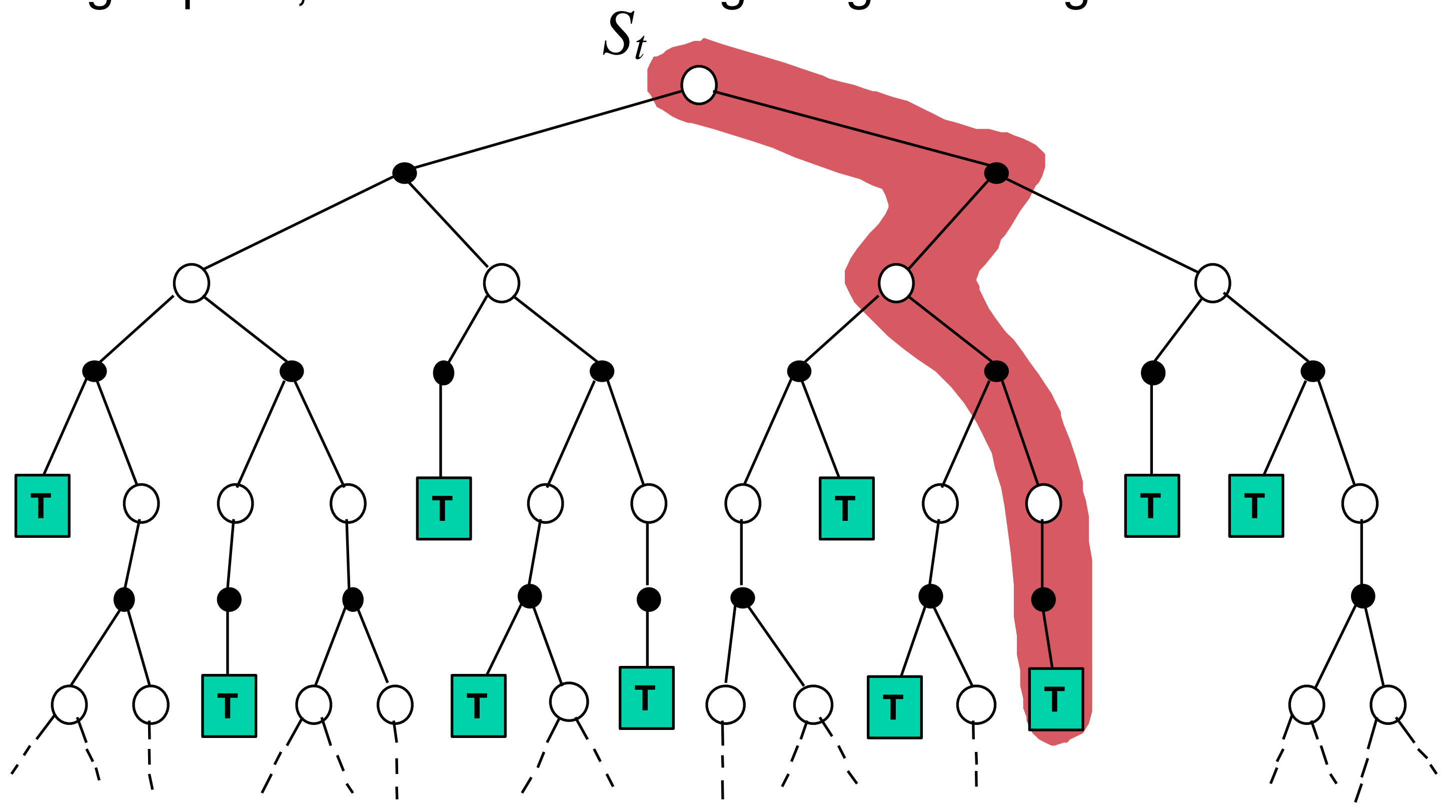
Example: Dialogue

S: What has been said so far
Action: next word/sentence
Reward: user satisfaction



Example: Robotics navigation

S: Positions and velocities of various joints; camera images
Action: joint torques
Reward: -1 for bumping into an obstacle, -0.1 per time step to encourage speed, +1000 for reaching the goal configuration



Value Functions

	state values	action values
prediction	v_π	q_π
control	v_*	q_*

- All theoretical objects, mathematical ideals (expected values)

- Algorithm will maintain estimate from data:

$$V_t(s) \qquad Q_t(s,a)$$

Values are *expected* returns

- The value of a state, given a policy:

$$v_\pi(s) = \mathbb{E}\{G_t \mid S_t = s, A_{t:\infty} \sim \pi\} \qquad v_\pi : \mathcal{S} \to \Re$$

- The value of a state-action pair, given a policy:

$$q_\pi(s,a) = \mathbb{E}\{G_t \mid S_t = s, A_t = a, A_{t+1:\infty} \sim \pi\} \qquad q_\pi : \mathcal{S} \times \mathcal{A} \to \Re$$

- The optimal value of a state:

$$v_*(s) = \max_\pi v_\pi(s) \qquad v_* : \mathcal{S} \to \Re$$

- The optimal value of a state-action pair:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \qquad q_* : \mathcal{S} \times \mathcal{A} \to \Re$$

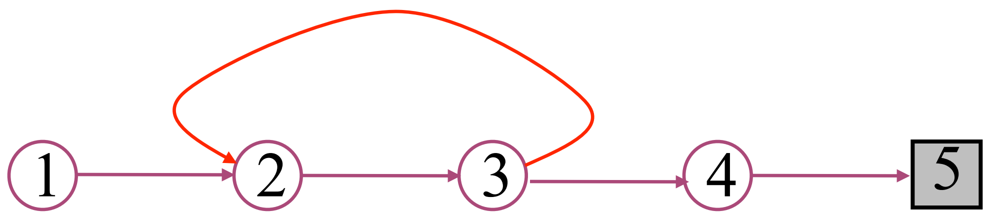
- Optimal policy: π_* is an optimal policy if and only if

$$\pi_*(a|s) > 0 \text{ only where } q_*(s,a) = \max_b q_*(s,b) \qquad \forall s \in \mathcal{S}$$

  - in other words, π_* is optimal iff it is *greedy* wrt q_*

Monte Carlo Policy Evaluation

- *Goal:* learn $v_\pi(s)$
- *Given:* some number of episodes under π which contain *s*
- *Idea:* Average returns observed after visits to s



- *Every-Visit MC:* average returns for *every* time *s* is visited in an episode
- *First-visit MC:* average returns only for *first* time *s* is visited in an episode
- Both converge asymptotically

First-visit Monte Carlo policy evaluation

Initialize:

$\pi \leftarrow$ policy to be evaluated

$V \leftarrow$ an arbitrary state-value function

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:

Generate an episode using π

For each state s appearing in the episode:

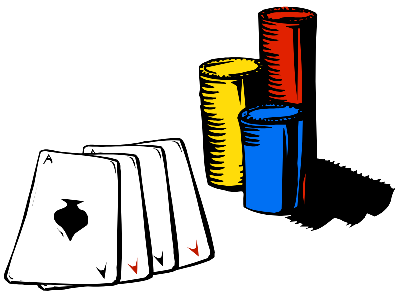
$G \leftarrow$ return following the first occurrence of s

Append G to $Returns(s)$

$V(s) \leftarrow$ average$(Returns(s))$

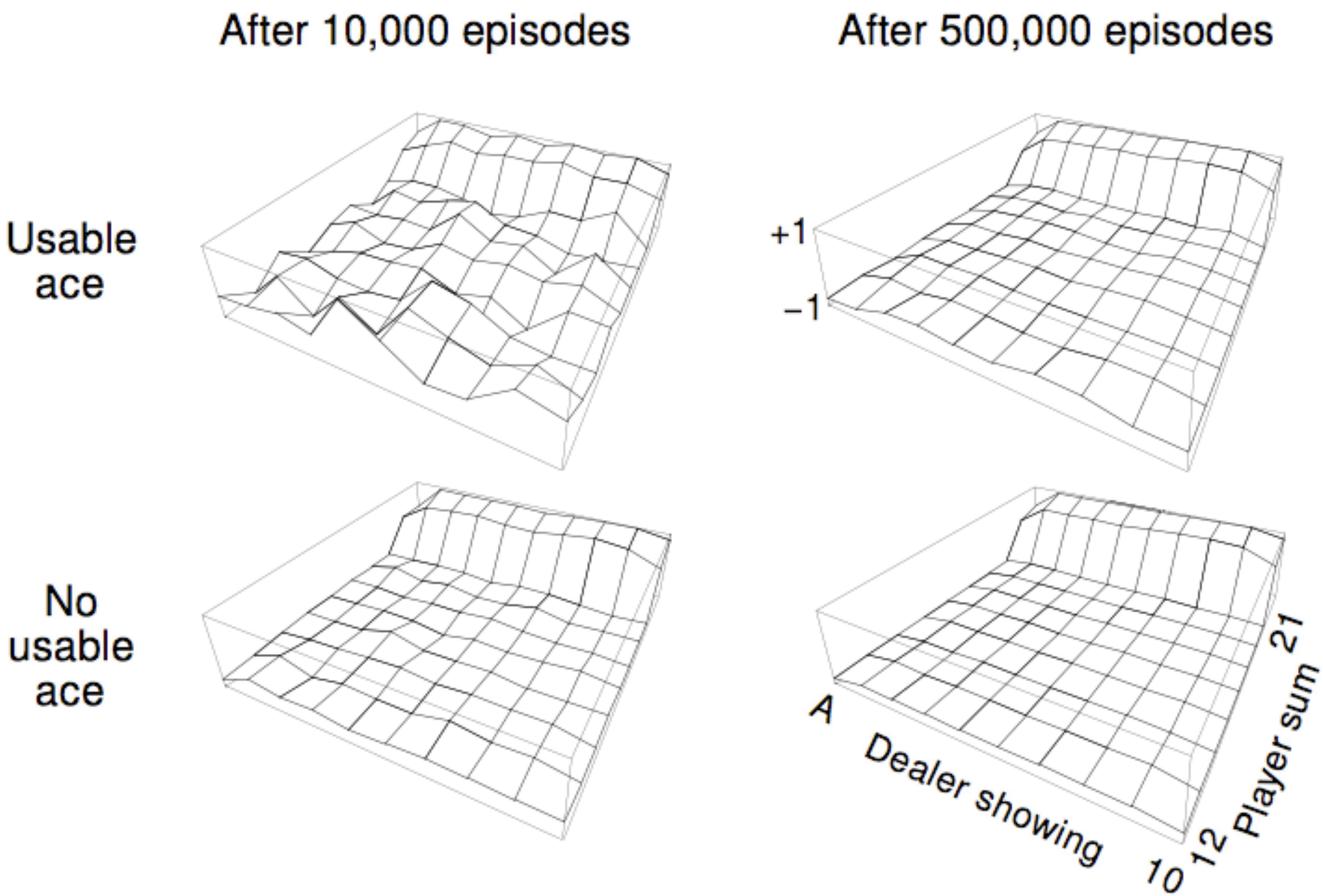
Blackjack example

- *Object:* Have your card sum be greater than the dealer's without exceeding 21.
- *States* (200 of them):
  - current sum (12-21)
  - dealer's showing card (ace-10)
  - do I have a useable ace?

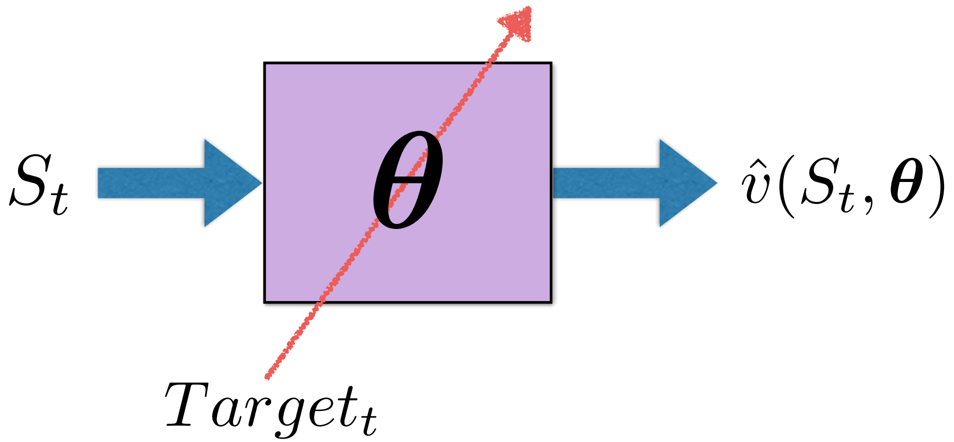


- *Reward:* +1 for winning, 0 for a draw, -1 for losing
- *Actions:* stick (stop receiving cards), hit (receive another card)
- *Policy:* Stick if my sum is 20 or 21, else hit

Learned blackjack state-value functions



Value function approximation (VFA)



Target depends on the agent's behavior!

Objective: minimize Mean Square Value Error

$$\mathrm{MSVE}(\boldsymbol{\theta}) \doteq \sum_{s \in \mathcal{S}} d(s) \Big[v_\pi(s) - \hat{v}(s, \boldsymbol{\theta}) \Big]^2$$

where $d(s)$ is the fraction of time steps spent in *s*

Use G_t instead of v_π

Monte Carlo will provide *samples of the expectation*

- Use *sample return* instead of v_π
- Use *actual visited states* instead of $d(s)$

Gradient Monte Carlo Algorithm for Approximating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^n \to \mathbb{R}$

Initialize value-function weights $\boldsymbol{\theta}$ as appropriate (e.g., $\boldsymbol{\theta} = \mathbf{0}$)
Repeat forever:
    Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
    For $t = 0, 1, \ldots, T-1$:
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \big[G_t - \hat{v}(S_t,\boldsymbol{\theta}) \big] \nabla \hat{v}(S_t,\boldsymbol{\theta})$

MC vs supervised regression

- Target returns can be viewed as a supervised label (true value we want to fit)
- State is the input
- We can use any function approximator to fit a function from states to returns! Neural nets, linear, nonparametric…

- *Unlike supervised learning: there is strong correlation between inputs and between outputs!*
- Due to the lack of iid assumptions, theoretical results from supervised learning cannot be directly applied

State aggregation is the simplest kind of VFA

- States are partitioned into disjoint subsets (groups)
- One component of $\boldsymbol{\theta}$ is allocated to each group

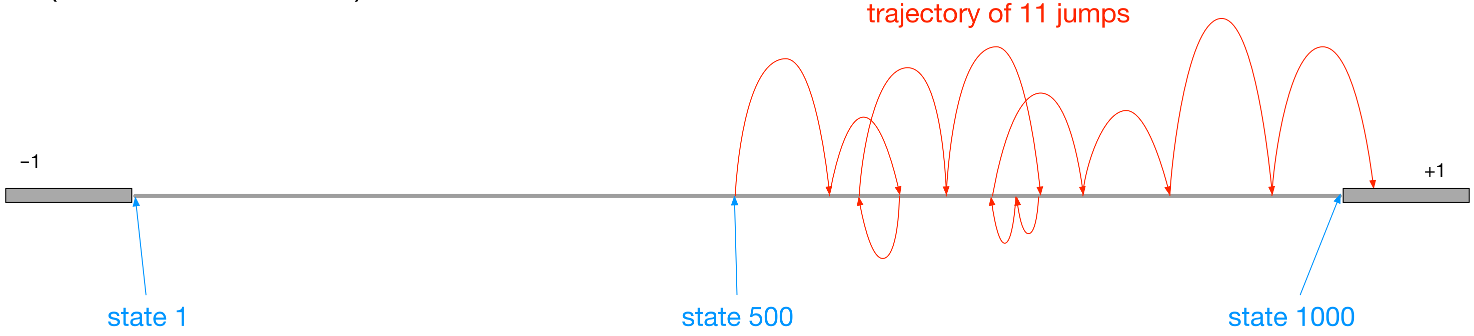
$$\hat{v}(s, \boldsymbol{\theta}) \doteq \theta_{group(s)}$$

$$\nabla_{\boldsymbol{\theta}}\, \hat{v}(s, \boldsymbol{\theta}) \doteq [0, 0, \ldots, 0, 1, 0, 0, \ldots, 0]$$

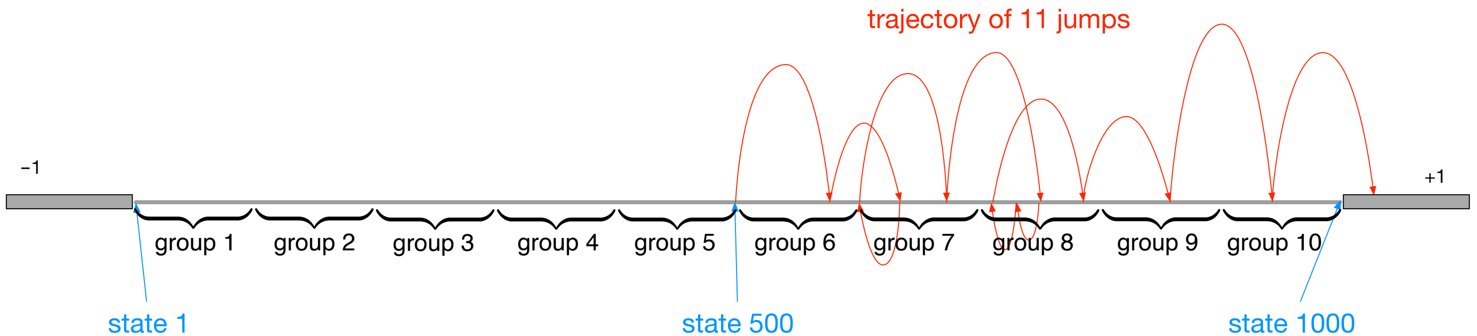
Recall: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \left[Target_t - \hat{v}(S_t, \boldsymbol{\theta}) \right] \nabla_{\boldsymbol{\theta}}\, \hat{v}(S_t, \boldsymbol{\theta})$

Example: The 1000-state random walk

- States are numbered 1 to 1000
- Walks start in the near middle, at state 500 $S_0 = 500$
- At each step, *jump* to one of the 100 states to the right, or to one of the 100 states to the left $S_1 \in \{400..499\} \cup \{501..600\}$
- If the jump goes beyond 1 or 1000, terminates with a reward of -1 or +1 (otherwise R_t=0)



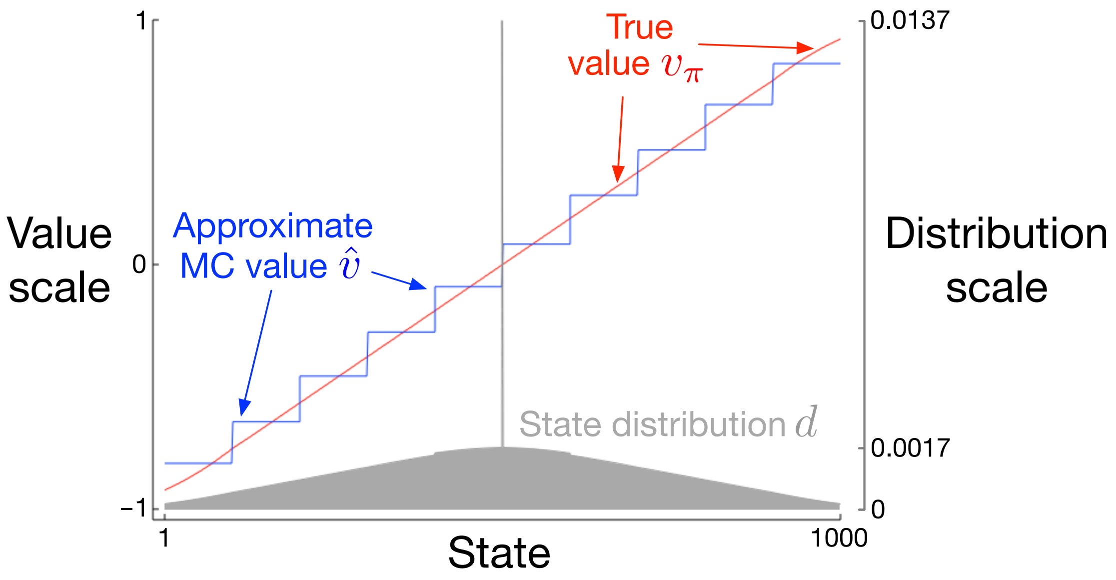
Example: State aggregation into 10 groups



The whole value function over 1000 states will be approximated with 10 numbers!

Example: Gradient MC with *state aggregation*

- 10 groups of 100 states
- after 100,000 episodes
- $\alpha = 2 \times 10^{-5}$
- state distribution affects accuracy



What about control?

Policy at step t = π_t =

a mapping from states to action probabilities

$\pi_t(a \mid s) =$ probability that $A_t = a$ when $S_t = s$

Special case - *deterministic policies*:

$\pi_t(s)$ = the action taken with prob=1 when $S_t = s$

- Reinforcement learning methods specify how the agent changes its policy as a result of experience.
- Roughly, the agent's goal is to get as much reward as it can over the long run.

Monte Carlo Estimation of Action Values (Q)

- Monte Carlo is most useful when a model is not available
  - We want to learn q_*
- $q_\pi(s,a)$ - average return starting from state s and action a then following π
- Converges asymptotically *if* every state-action pair is visited
- *Exploring starts:* Every state-action pair has a non-zero probability of being the starting pair

Monte Carlo Exploring Starts

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

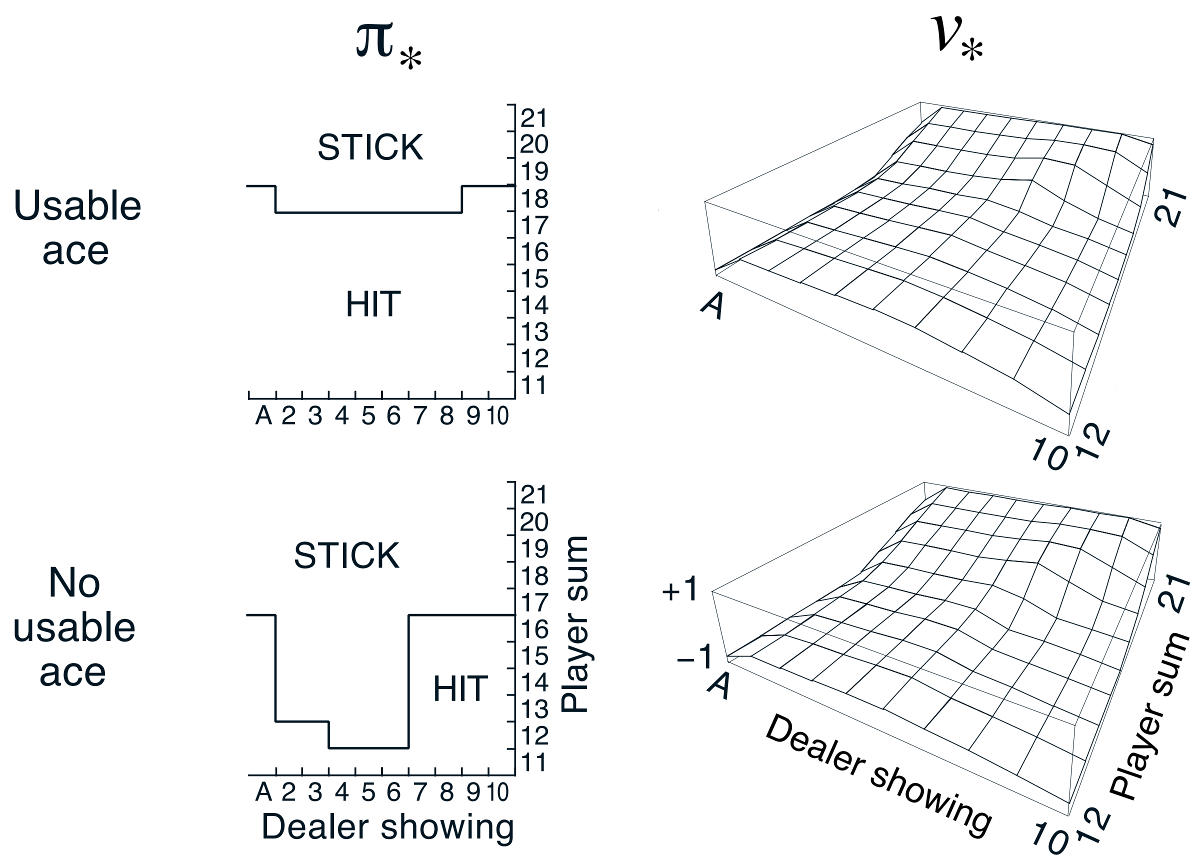
$Q(s, a) \leftarrow$ arbitrary
$\pi(s) \leftarrow$ arbitrary
$Returns(s, a) \leftarrow$ empty list

Repeat forever:

Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability > 0
Generate an episode starting from S_0, A_0, following π
For each pair s, a appearing in the episode:
$G \leftarrow$ return following the first occurrence of s, a
Append G to $Returns(s, a)$
$Q(s, a) \leftarrow$ average($Returns(s, a)$)
For each s in the episode:
$\pi(s) \leftarrow \arg\max_a Q(s, a)$

Blackjack example continued

- Exploring starts
- Initial policy as described before



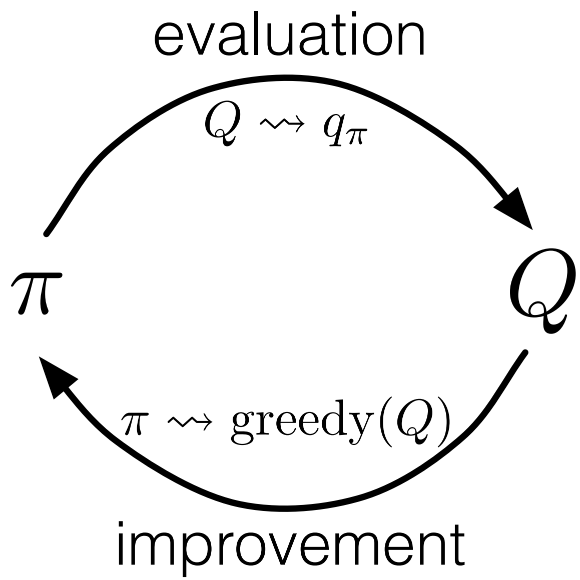
On-policy Monte Carlo Control

- *On-policy:* learn about policy currently executing
- How do we get rid of exploring starts?
  - The policy must be eternally *soft*:
    - $\pi(a|s) > 0$ for all s and a
  - e.g. ε-soft policy:
    - probability of an action = $\frac{\epsilon}{|\mathcal{A}(s)|}$ or $1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$

      non-max max (greedy)

- An instance of *policy iteration*: move policy *towards* greedy policy (e.g., ε-greedy)

Monte Carlo Control



- MC policy iteration: Policy evaluation using MC methods followed by policy improvement
- Policy improvement step: greedify with respect to value (or action-value) function

On-policy MC Control

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$Q(s,a) \leftarrow$ arbitrary

$Returns(s,a) \leftarrow$ empty list

$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:

(a) Generate an episode using π

(b) For each pair s, a appearing in the episode:

$G \leftarrow$ return following the first occurrence of s, a

Append G to $Returns(s,a)$

$Q(s,a) \leftarrow \text{average}(Returns(s,a))$

(c) For each s in the episode:

$A^* \leftarrow \arg\max_a Q(s,a)$

For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

MC Control with function approximation

- Always learn the action-value function of the current policy
- Always act near-greedily wrt the current action-value estimates (eg soft max, epsilon-greedy)
- The learning rule:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big[U_t - \hat{q}(S_t, A_t, \boldsymbol{\theta}_t)\Big]\nabla\hat{q}(S_t, A_t, \boldsymbol{\theta}_t)$$

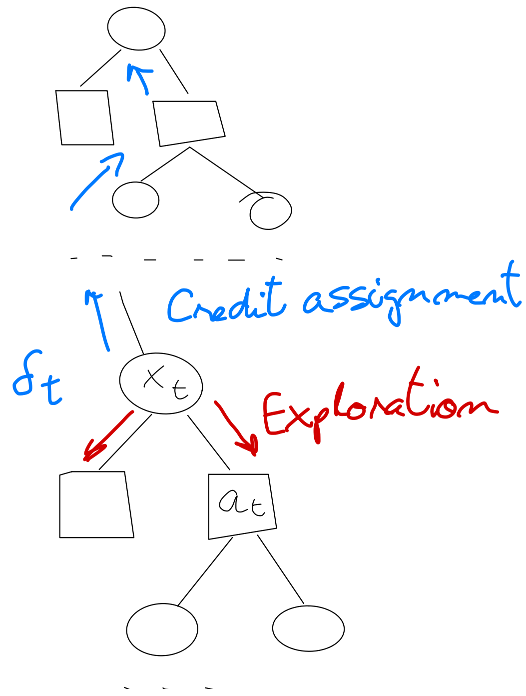
- For MC, $U_t = G_t$

Monte Carlo Summary

- Easy to implement for any sequential decision making problem!
- Can leverage the power of function approximation
- Policies leverage randomized exploration ideas
- But: we have to wait until the end of an episode to make any updates!
- That can be really long! Eg sequential treatment design
- Even in games that can be too long! Eg opponent adaptation
- Can we do something more efficient? More responsive?

Recall: sequential decision making

- At time t, agent receives an observation from set $\mathcal{X}$ and can choose an action from set $\mathcal{A}$ (think finite for now)
- Goal of the agent is to maximize long-term return



- Recall the infinite tree of possible interactions of the agent and environment - is finite horizon the only assumption we can make?

Finite clustering assumption

- The infinite paths cluster into a finite number of clusters!
- The means *similar situations will recur*
- So we can generalize!

One more step: Markovian assumption

- The way we got to some specific situation is not relevant for the future!

- All that matters is our current observation X_t

- Alternatively, if we should have remembered something, we will consider it part of X_t

- We will call such an observation *state*

The Markov Property

- By "the state" at step *t*, the book means whatever information is available to the agent at step *t* about its environment.
- The state can include immediate "sensations," highly processed sensations, and structures built up over time from sequences of sensations.
- Ideally, a state should summarize past sensations so as to retain all "essential" information, i.e., it should have the **Markov Property**:

$$\mathbf{Pr}\{R_{t+1} = r, S_{t+1} = s' \mid S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\} =$$

$$p(s', r | s, a) = \mathbf{Pr}\{R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t\}$$

- for all $s' \in \mathcal{S}^+, r \in \mathcal{R}$, and all histories $S_0,\ A_0,\ R_1,\ \ldots,\ S_{t-1},\ A_{t-1},\ R_t,\ S_t,\ A_t$.

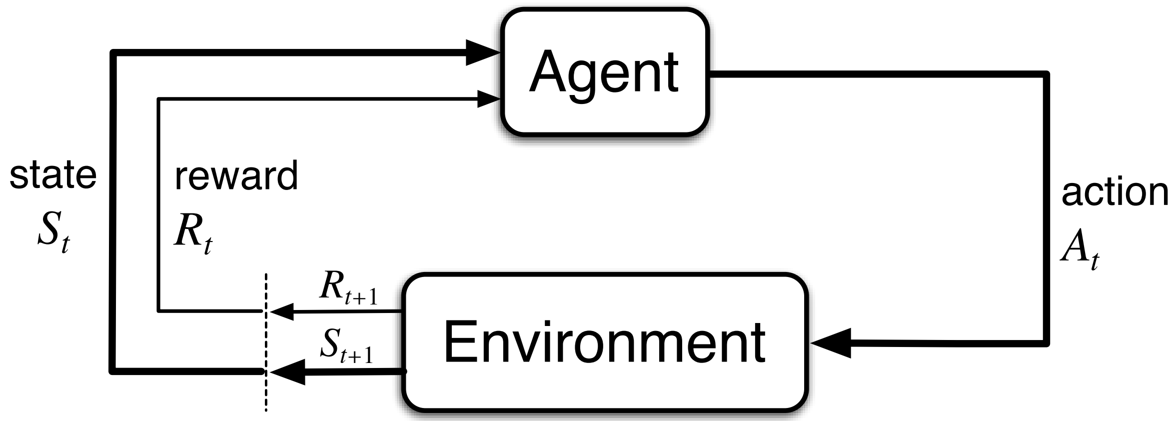
Markov Property

- An assumption about the environment
- Next state and reward depend only on the previous state and action, and noting else that happened in the past

$$p(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a) = p(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a, \tau_t), \forall \tau_t$$

- The assumption is useful to develop, analyze and understand algorithms
- It does NOT mean it has to always hold

The Agent-Environment Interface



Agent and environment interact at discrete time steps: $t = 0, 1, 2, 3, \ldots$

Agent observes state at step t: $S_t \in \mathcal{S}$

produces action at step t : $A_t \in \mathcal{A}(S_t)$

gets resulting reward: $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$

and resulting next state: $S_{t+1} \in \mathcal{S}^+$

Markov Decision Processes

- If a reinforcement learning task has the Markov Property, it is basically a **Markov Decision Process (MDP)**.
- If state and action sets are finite, it is a **finite MDP**.
- To define a finite MDP, you need to give:
  - **state and action sets**
  - one-step "dynamics"

$$p(s', r|s, a) = \mathbf{Pr}\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\}$$

$$p(s'|s, a) \doteq \Pr\{S_{t+1} = s' \mid S_t = s, A_t = a\} = \sum_{r \in \mathcal{R}} p(s', r|s, a)$$

$$r(s, a) \doteq \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a)$$

An Example Finite MDP

Recycling Robot

- At each step, robot has to decide whether it should (1) actively search for a can, (2) wait for someone to bring it a can, or (3) go to home base and recharge.
- Searching is better but runs down the battery; if runs out of power while searching, has to be rescued (which is bad).
- Decisions made on basis of current energy level: `high`, `low`.
- Reward = number of cans collected

Recycling Robot MDP

$\mathcal{S} = \{\texttt{high}, \texttt{low}\}$

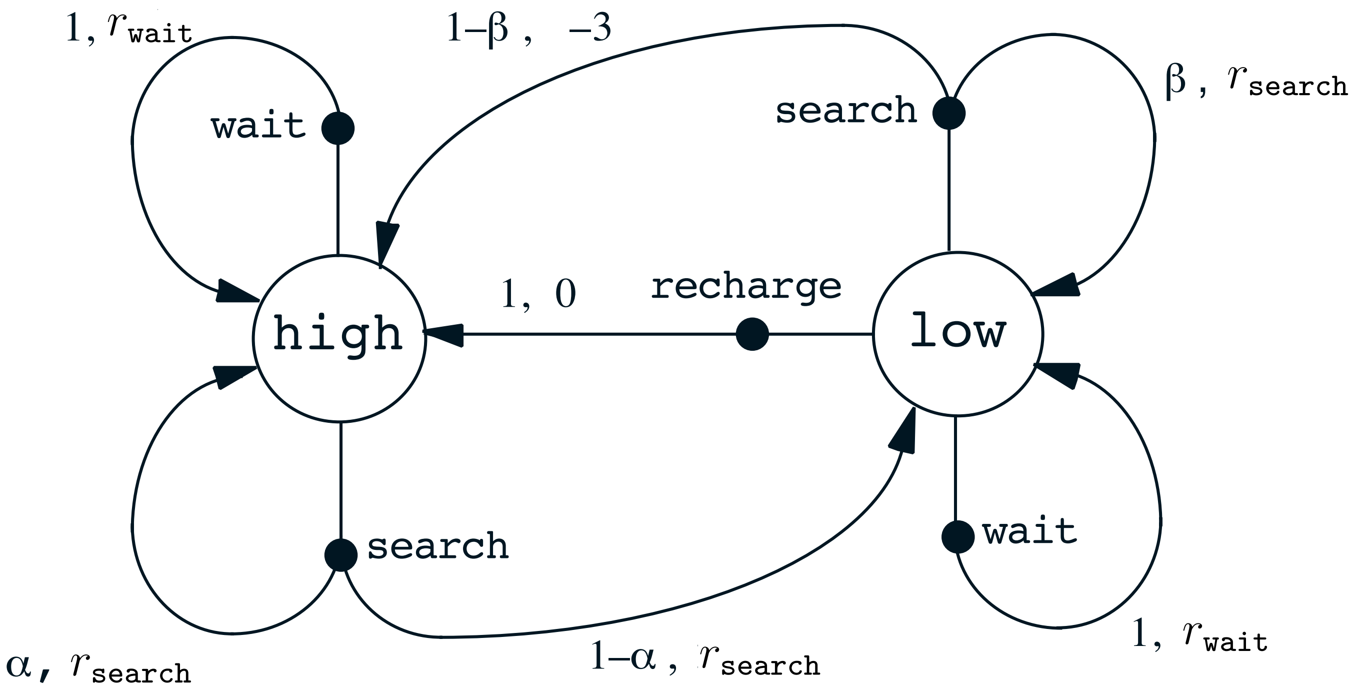
$\mathcal{A}(\texttt{high}) = \{\texttt{search}, \texttt{wait}\}$

$\mathcal{A}(\texttt{low}) = \{\texttt{search}, \texttt{wait}, \texttt{recharge}\}$

$r_{\texttt{search}}$ = expected no. of cans while searching

$r_{\texttt{wait}}$ = expected no. of cans while waiting

$r_{\texttt{search}} > r_{\texttt{wait}}$



Return

Suppose the sequence of rewards after step t is:

$$R_{t+1}, R_{t+2}, R_{t+3}, \ldots$$

What do we want to maximize?

At least three cases, but in all of them,

we seek to maximize the **expected return**, $E\{G_t\}$, on each step t.

- Total reward, G_t = sum of all future reward in the episode
- Discounted reward, G_t = sum of all future *discounted* reward
- Average reward, G_t = average reward per time step

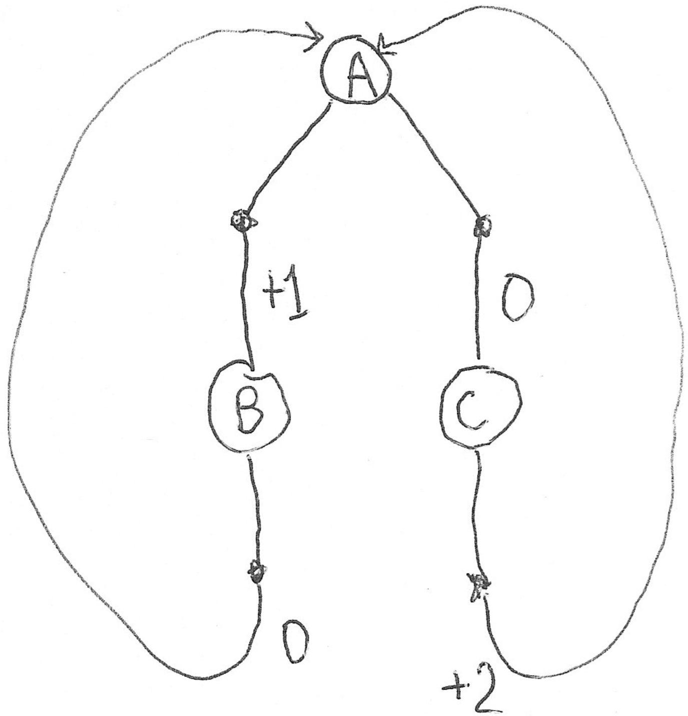
Rewards and returns

- The objective in RL is to maximize long-term future reward
- That is, to choose A_t so as to maximize $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$
- But what exactly should be maximized?
- The discounted *return* at time t:

the *discount rate*

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \qquad \gamma \in [0, 1)$$

γ	Reward sequence	Return
0.5(or any)	1 0 0 0…	
0.5	0 0 2 0 0 0…	
0.9	0 0 2 0 0 0…	
0.5	-1 2 6 3 2 0 0 0…	



What policy is optimal?

A: left

B: Right

C: other

If $\gamma = 0$?

If $\gamma = .99$

If $\gamma = \frac{1}{2}$?

Recall: Episodic Tasks

Episodic tasks: interaction breaks naturally into episodes, e.g., plays of a game, trips through a maze

In episodic tasks, we almost always use simple *total reward*:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T ,$$

where T is a final time step at which a **terminal state** is reached, ending an episode.

Continuing Tasks

Continuing tasks: interaction does not have natural episodes, but just goes on and on...

In this class, for continuing tasks we will always use *discounted return*:

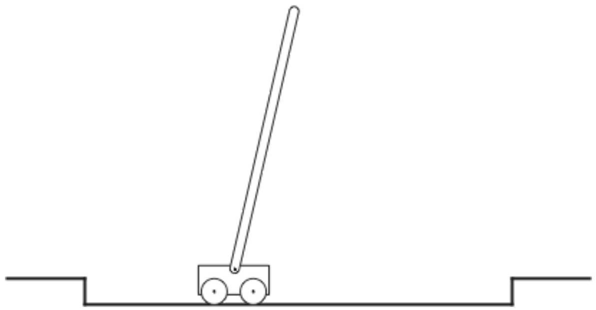
$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where γ, $0 \le \gamma \le 1$, is the **discount rate**.

shortsighted $0 \leftarrow \gamma \rightarrow 1$ farsighted

Typically, $\gamma = 0.9$

An Example: Pole Balancing



Avoid **failure:** the pole falling beyond a critical angle or the cart hitting end of track

As an **episodic task** where episode ends upon failure:

reward = +1 for each step before failure

$\Rightarrow$ return = number of steps before failure

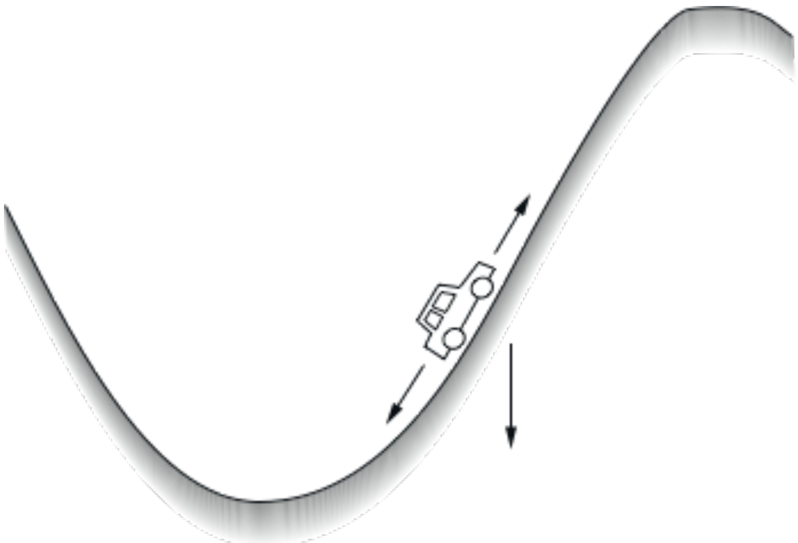
As a **continuing task** with discounted return:

reward = −1 upon failure; 0 otherwise

$\Rightarrow$ return = $-\gamma^k$, for k steps before failure

In either case, return is maximized by avoiding failure for as long as possible.

Another Example: Mountain Car



Get to the top of the hill
as quickly as possible.

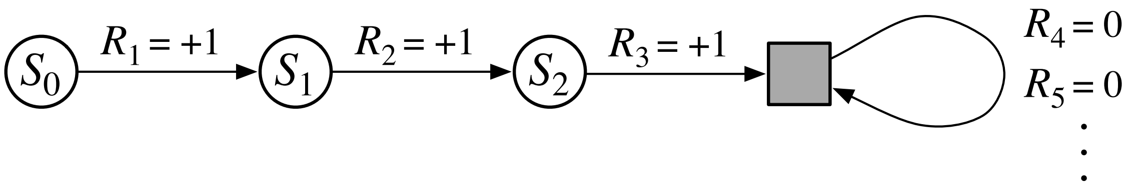
reward = −1 for each step where **not** at top of hill

⇒ return = − number of steps before reaching top of hill

Return is maximized by minimizing
number of steps to reach the top of the hill.

A Trick to Unify Notation for Returns

- In episodic tasks, we number the time steps of each episode starting from zero.
- We usually do not have to distinguish between episodes, so instead of writing $S_{t,j}$ for states in episode j, we write just S_t
- Think of each episode as ending in an absorbing state that always produces reward of zero:



- We can cover <u>all</u> cases by writing $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$

where γ can be 1 only if a zero reward absorbing state is always reached.