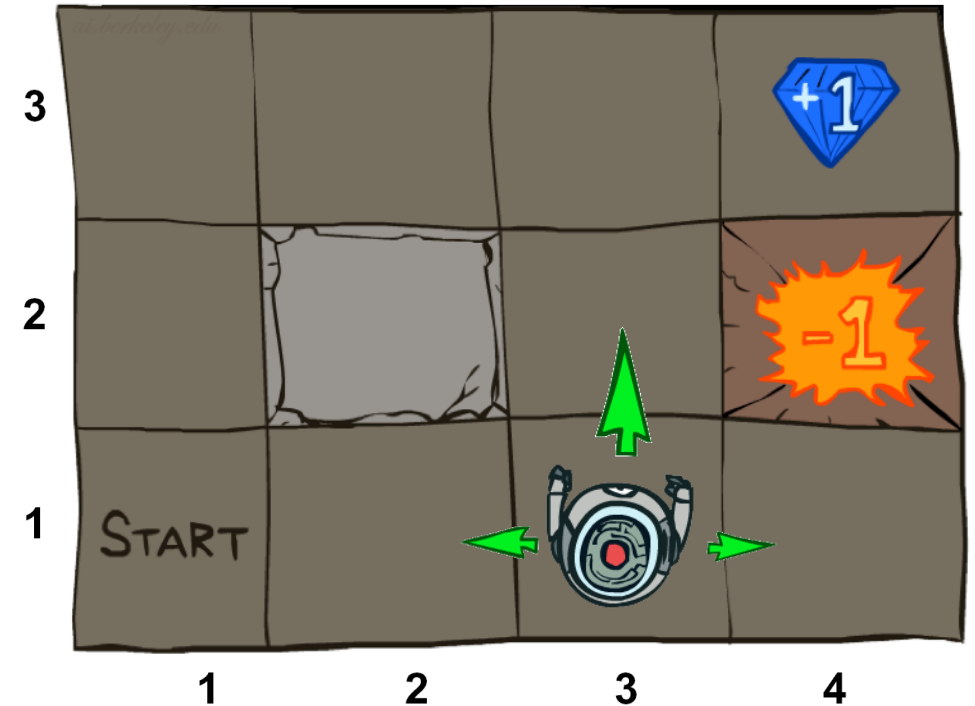


Announcement

- Project group registration
 - Each group only needs to register once.
 - Due: Dec. 6 (this Sunday), 11:59pm
 - Registration link: <https://wj.qq.com/s2/7551413/2fd0/>

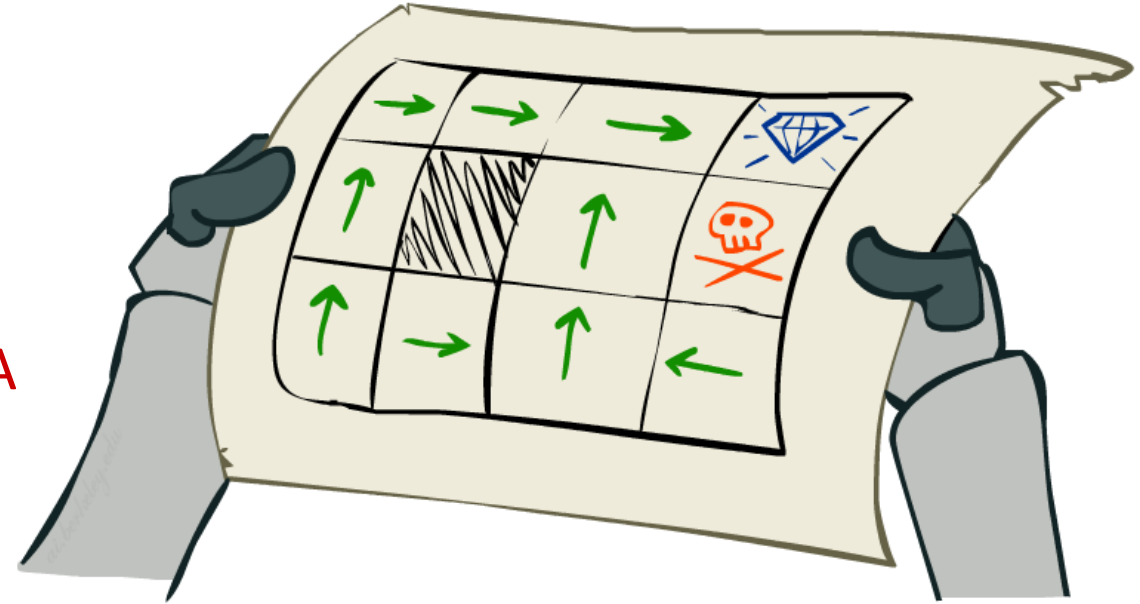
Markov Decision Processes

- An MDP is defined by:
 - A **set of states** $s \in S$
 - A **set of actions** $a \in A$
 - A **transition function** $T(s, a, s')$
 - Probability that a from s leads to s' , i.e., $P(s' | s, a)$
 - Also called the model or the dynamics
 - A **reward function** $R(s, a, s')$
 - Sometimes just $R(s)$ or $R(s')$
 - A **start state**
 - Maybe a **terminal state**
- MDPs are non-deterministic search problems
 - One way to solve them is with expectimax search
 - We'll have a new tool soon



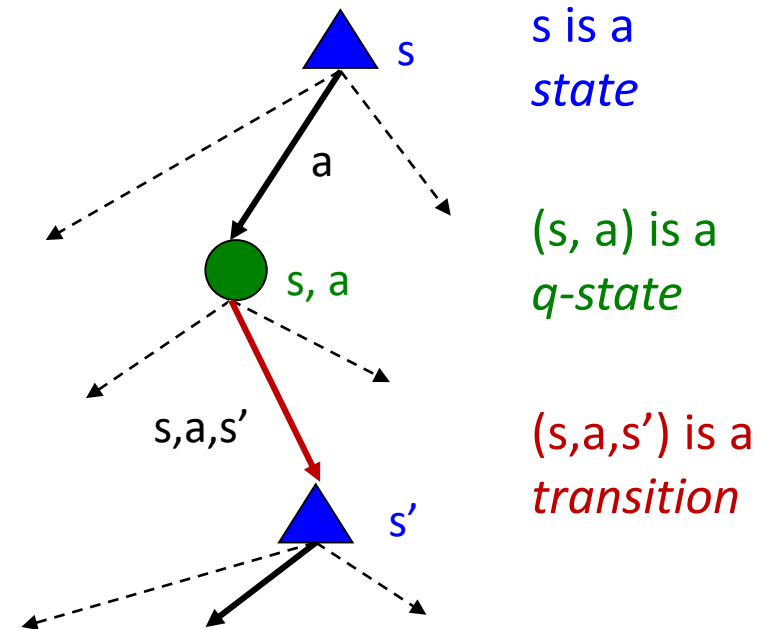
Policies

- In deterministic single-agent search problems, we wanted an optimal **plan**, or sequence of actions, from start to a goal
- For MDPs, we want an optimal **policy** $\pi^*: S \rightarrow A$
 - A policy π gives an action for each state
 - An optimal policy is one that maximizes expected utility if followed
 - An explicit policy defines a reflex agent



Optimal Quantities

- The value (utility) of a state s :
 $V^*(s)$ = expected utility starting in s and acting optimally
- The value (utility) of a q-state (s,a) :
 $Q^*(s,a)$ = expected utility starting out having taken action a from state s and (thereafter) acting optimally
- The optimal policy:
 $\pi^*(s)$ = optimal action from state s



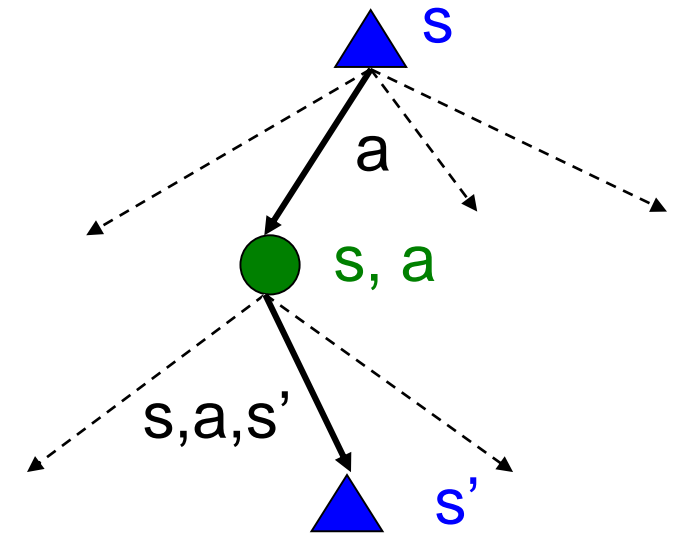
Values of States

- How to compute the value of a state
 - Expected utility under optimal action
 - This is just what expectimax computed!
- Recursive definition of value:

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

$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$



The Bellman Equation

Q-Value Iteration

- Value iteration: find successive (depth-limited) values

- Start with $V_0(s) = 0$
- Given V_k , calculate the depth $k+1$ values for all states:

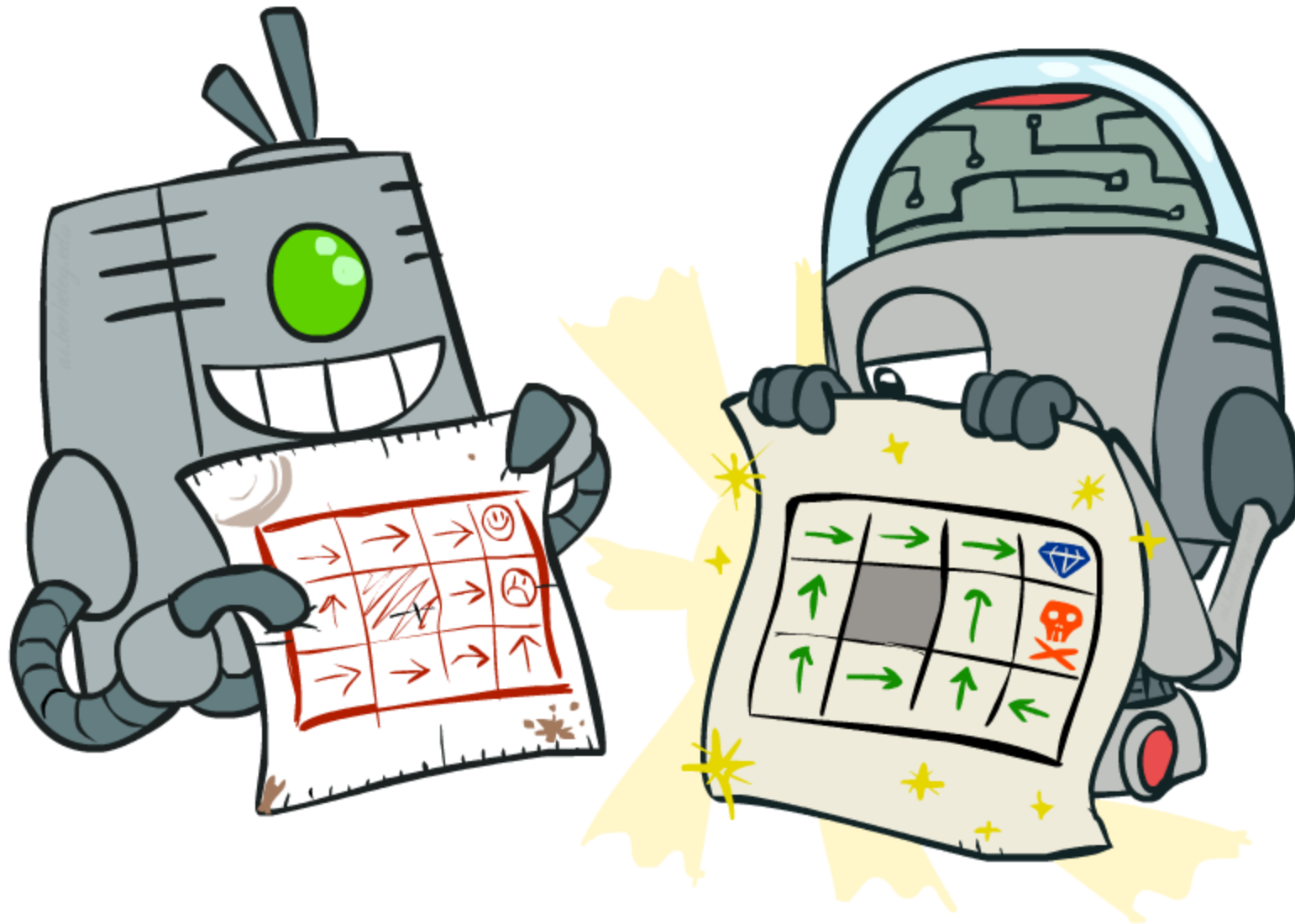
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

- But Q-values are more useful, so compute them instead

- Start with $Q_0(s,a) = 0$
- Given Q_k , calculate the depth $k+1$ q-values for all q-states:

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \max_{a'} Q_k(s', a')]$$

Policy Methods

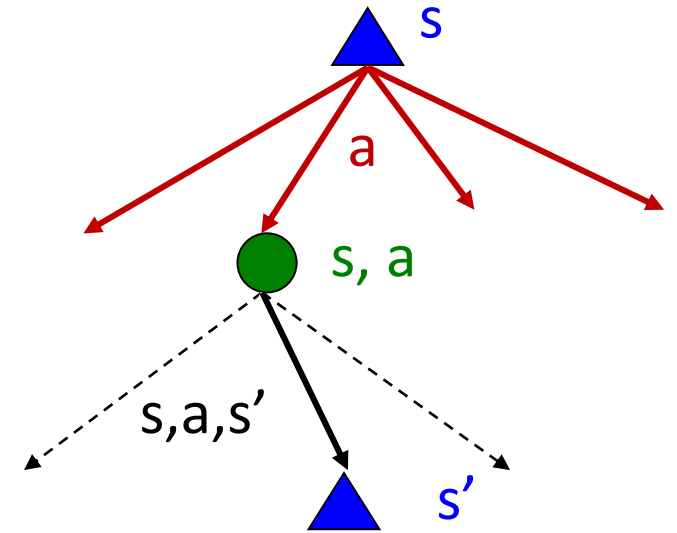


Problems with Value Iteration

- Value iteration repeats the Bellman updates:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

- Problem 1: It's slow – $O(S^2A)$ per iteration
- Problem 2: The “max” at each state rarely changes
 - The policy often converges long before the values



k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100

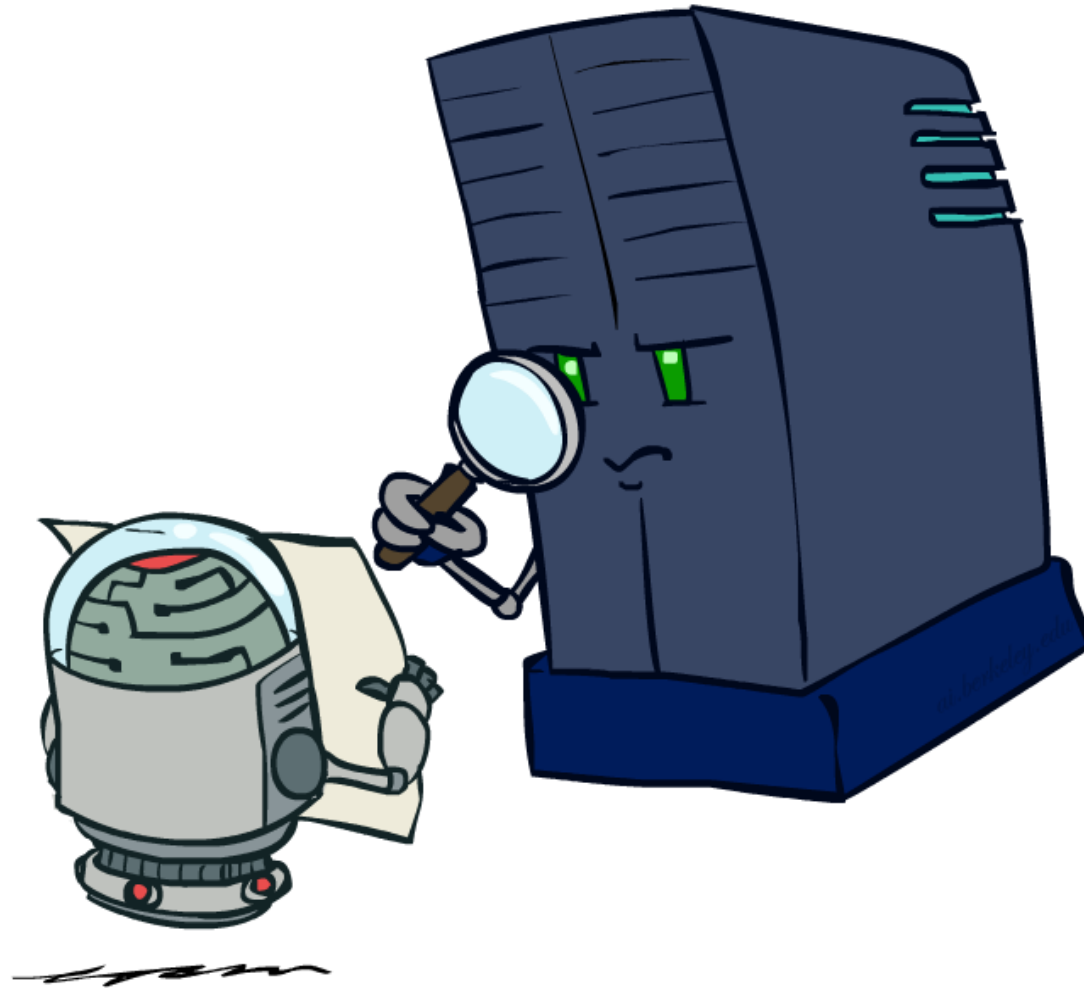


Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

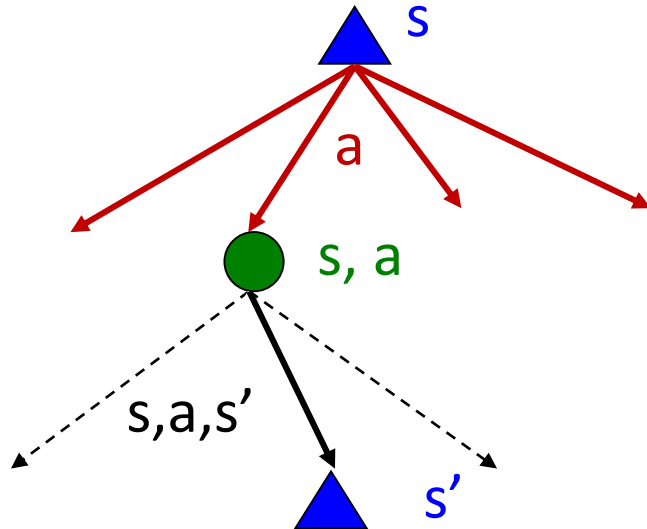
- **Policy iteration:** an alternative approach for value iteration
 - **Step 1: Policy evaluation:** calculate utilities for some fixed (not optimal) policy
 - **Step 2: Policy improvement:** update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
 - Repeat steps until policy converges
- It's still optimal!
- Can converge (much) faster under some conditions

Step 1: Policy Evaluation

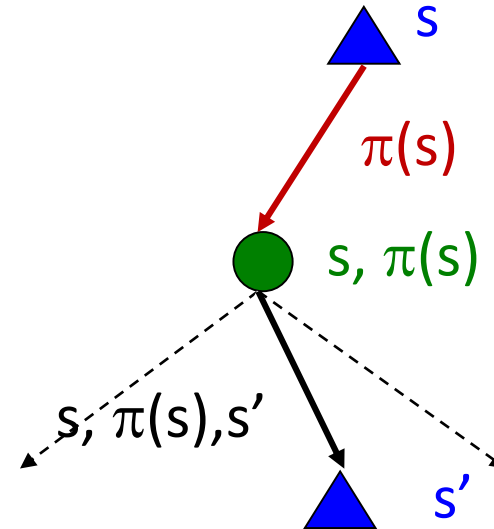


Fixed Policies

Do the optimal action



Do what π says to do



- Expectimax trees max over all actions to compute the optimal values
- If we fixed some policy $\pi(s)$, then the tree would be simpler – only one action per state

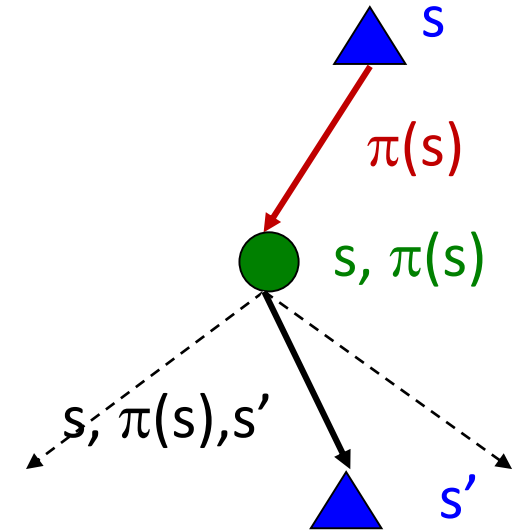
Utilities for a Fixed Policy

- The utility of a state s , under a fixed policy π :

$V^\pi(s)$ = expected utility starting in s and following π

- Recursive relation (one-step look-ahead):

$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V^\pi(s')]$$



Policy Evaluation

- How do we calculate the values under a fixed policy π ?

- Idea 1: Iterative updates (like value iteration)

- Start with $V_0^\pi(s) = 0$
- Given V_k^π , calculate the depth $k+1$ values for all states:

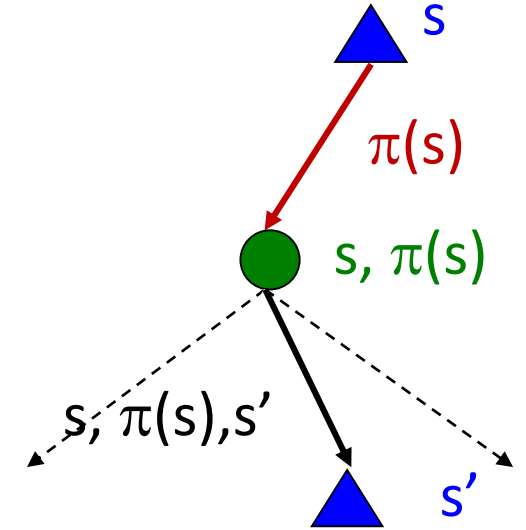
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$

- Repeat until convergence
- Efficiency: $O(S^2)$ per iteration

- Idea 2: Without the maxes, the Bellman equations are just a linear system

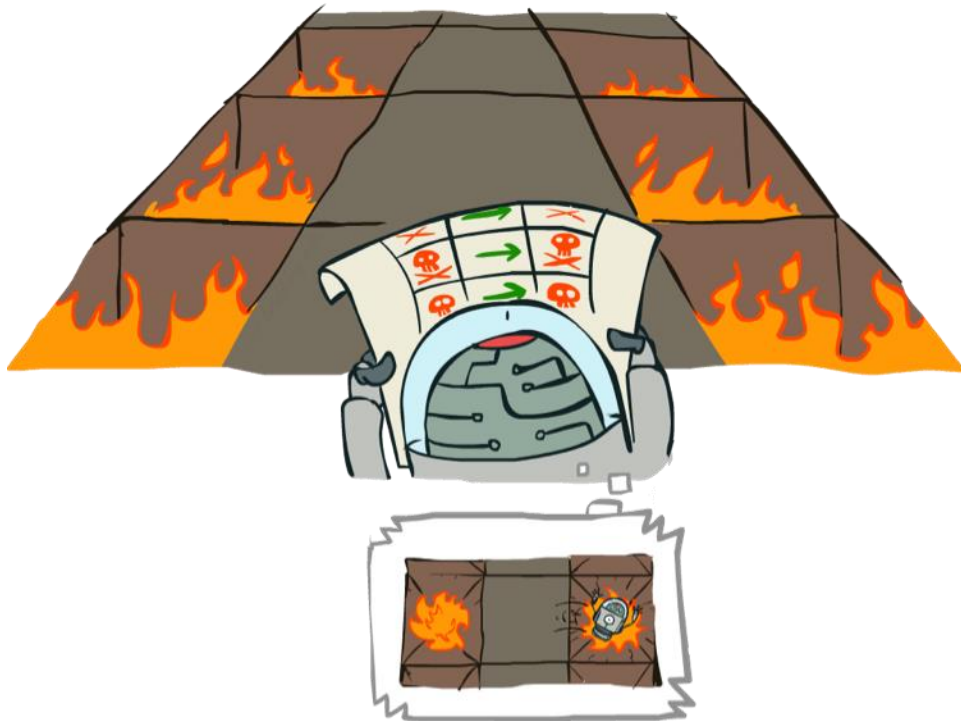
$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V^\pi(s')]$$

- Solvable with a linear system solver



Example: Policy Evaluation

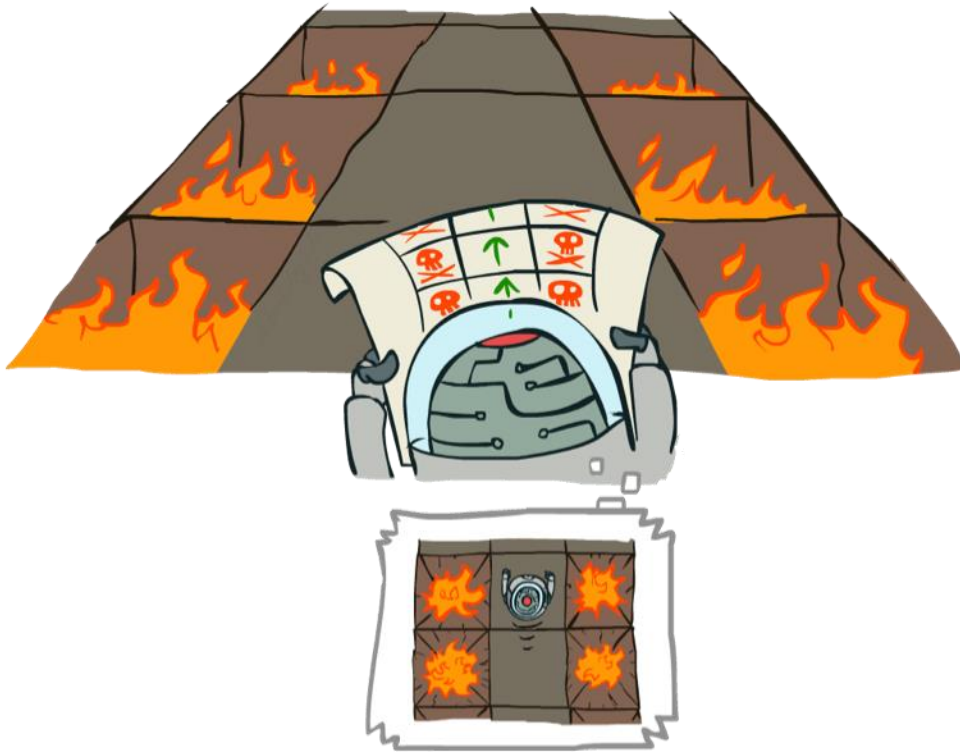
Always Go Right



-10.00	100.00	-10.00
-10.00	1.09 ▶	-10.00
-10.00	-7.88 ▶	-10.00
-10.00	-8.69 ▶	-10.00

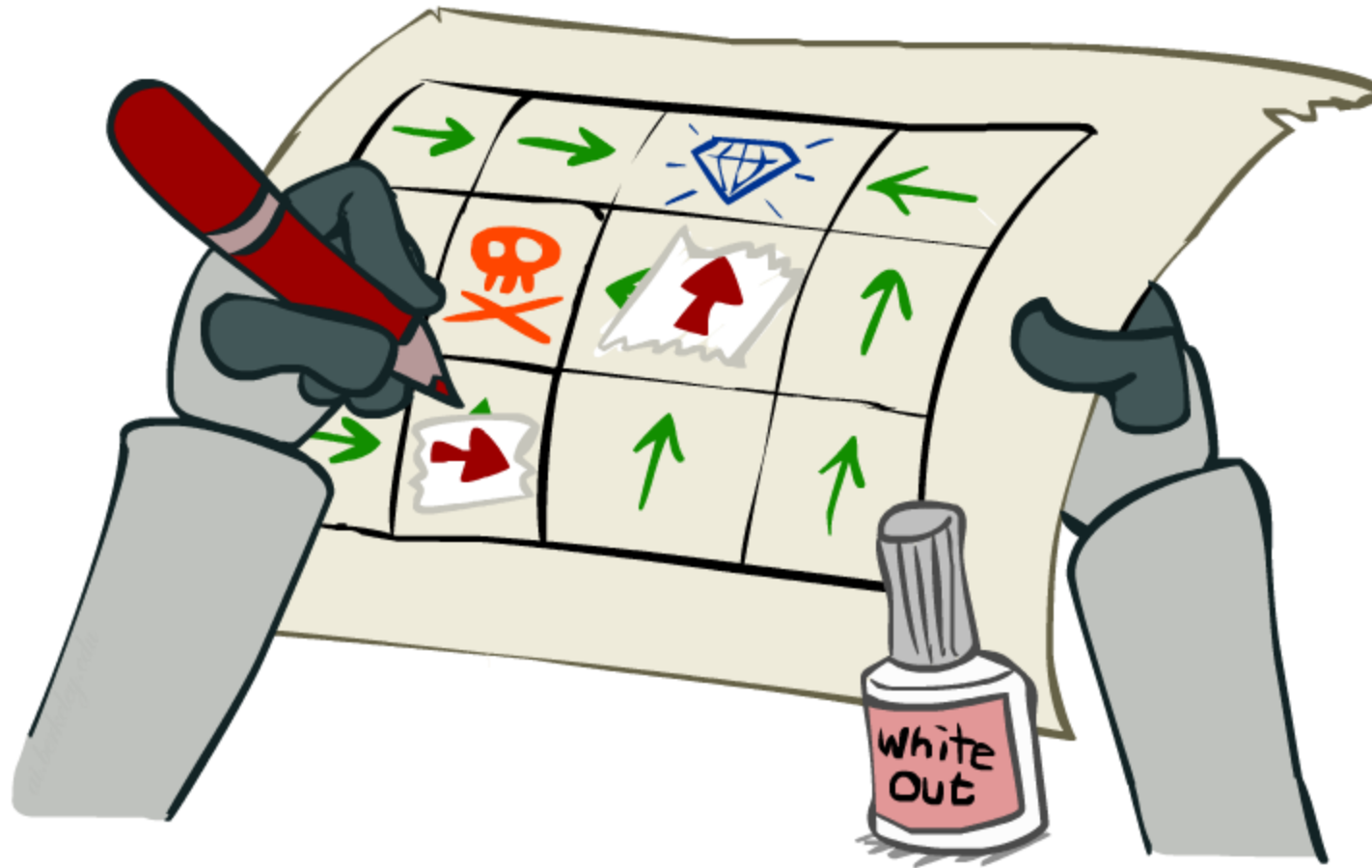
Example: Policy Evaluation

Always Go Forward



-10.00	100.00	-10.00
-10.00	70.20	-10.00
-10.00	48.74	-10.00
-10.00	33.30	-10.00

Step 2: Policy Improvement



Policy Improvement

- Step 2: Improvement: For fixed values, get a better policy using policy extraction
 - One-step look-ahead:

$$\pi_{i+1}(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^{\pi_i}(s')]$$

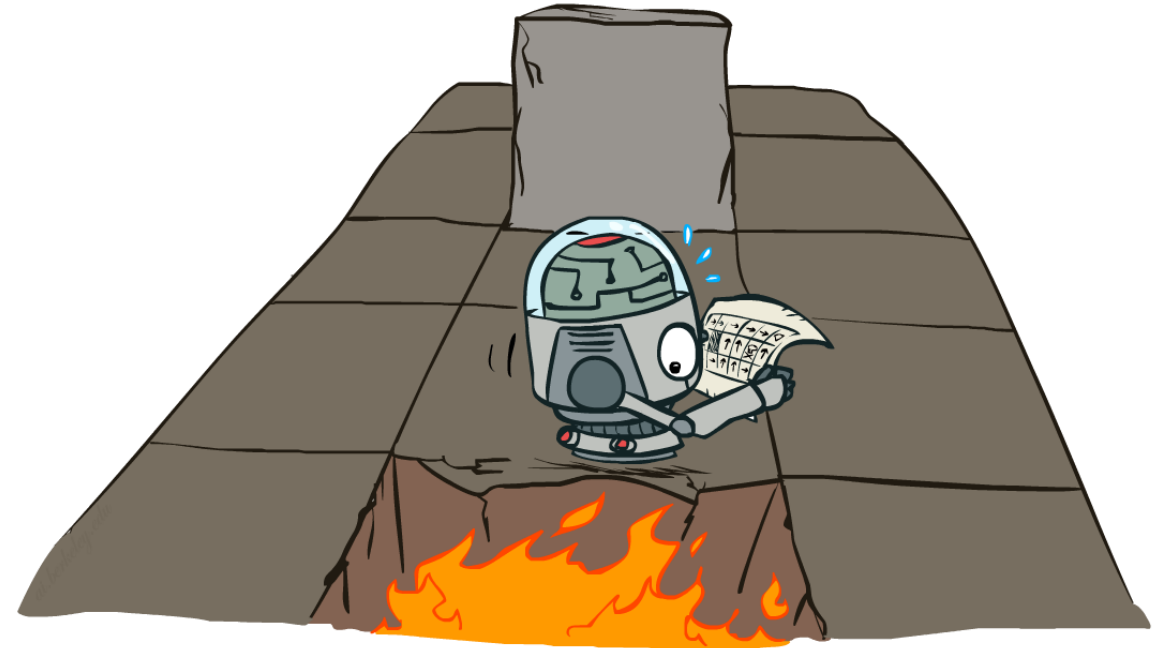
- Policy Iteration: repeat the two steps until policy converges

Value Iteration vs. Policy Iteration

- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
 - Every iteration updates both the values and (implicitly) the policy
 - We don't track the policy, but taking the max over actions implicitly recomputes it
- In policy iteration:
 - We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
 - After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
 - May converge faster
- Both are dynamic programs for solving MDPs

Summary

- Markov Decision Process
 - States S , Actions A , Transitions $P(s' | s, a)$, Rewards $R(s, a, s')$
- Quantities:
 - Policy, Utility, Values, Q-Values
- Solve MDP
 - Value iteration
 - Policy iteration

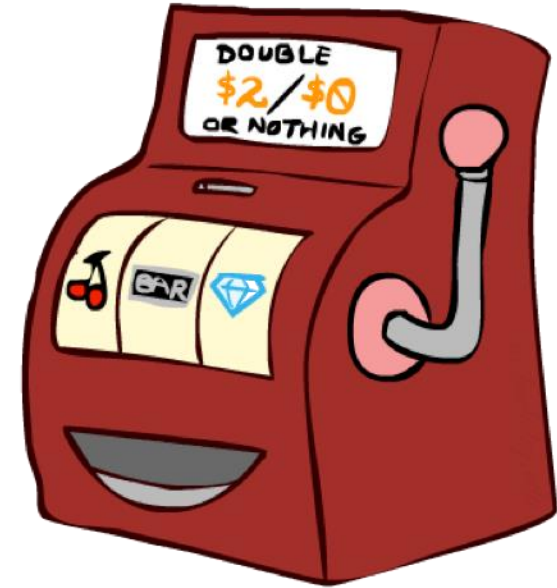
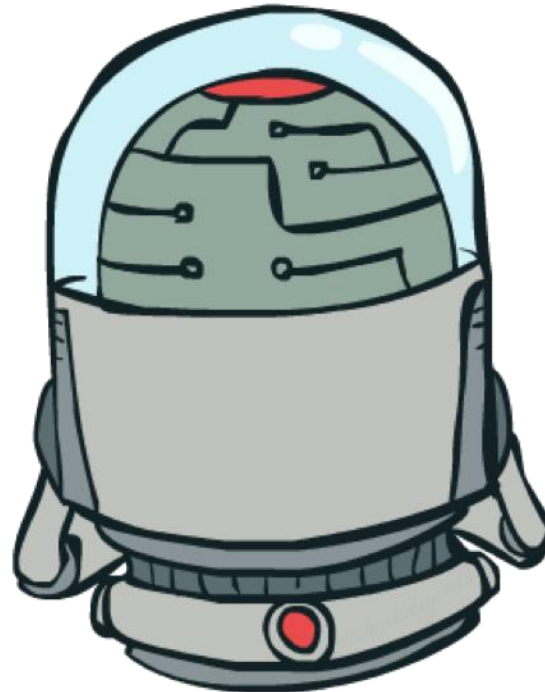


Reinforcement Learning



AIMA Chapter 21

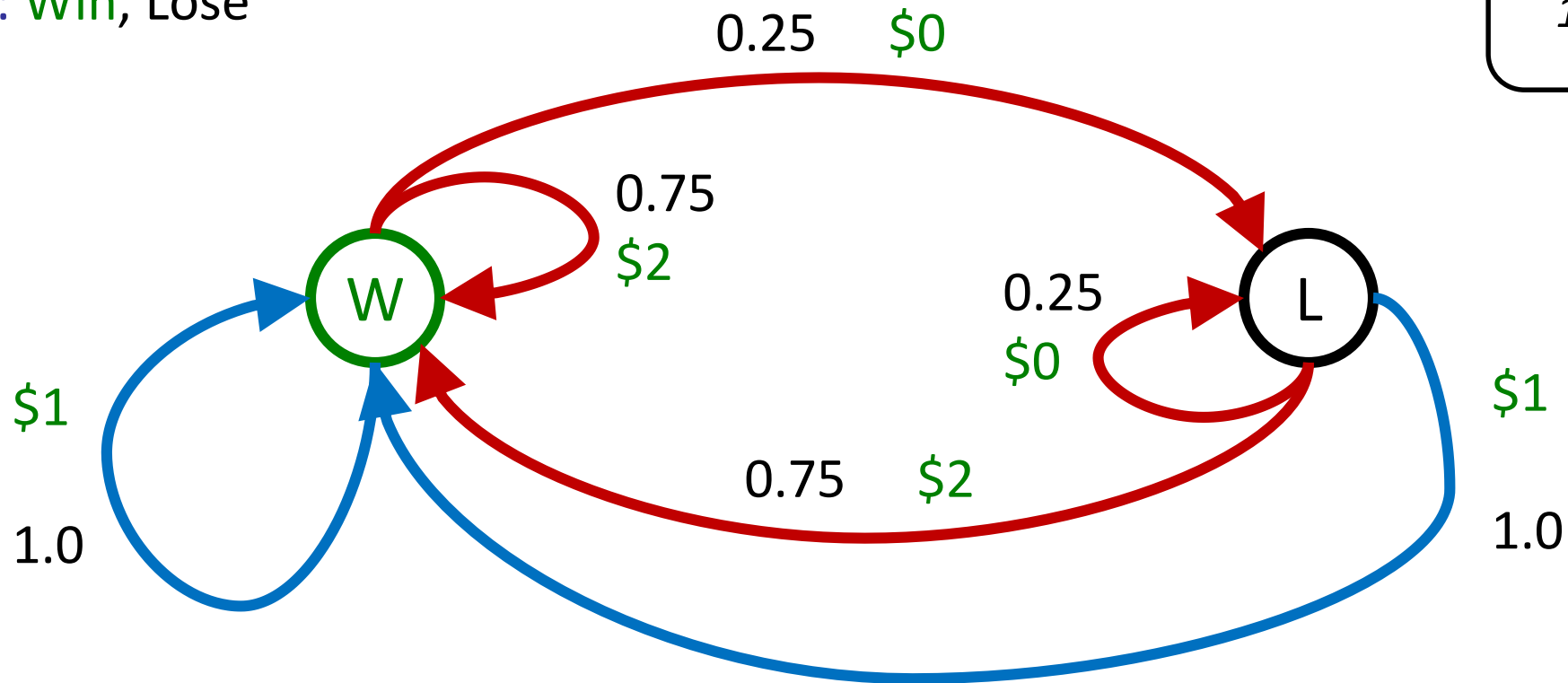
Double Bandits



Double-Bandit MDP

- Actions: *Blue*, *Red*
- States: *Win*, Lose

No discount
100 time steps

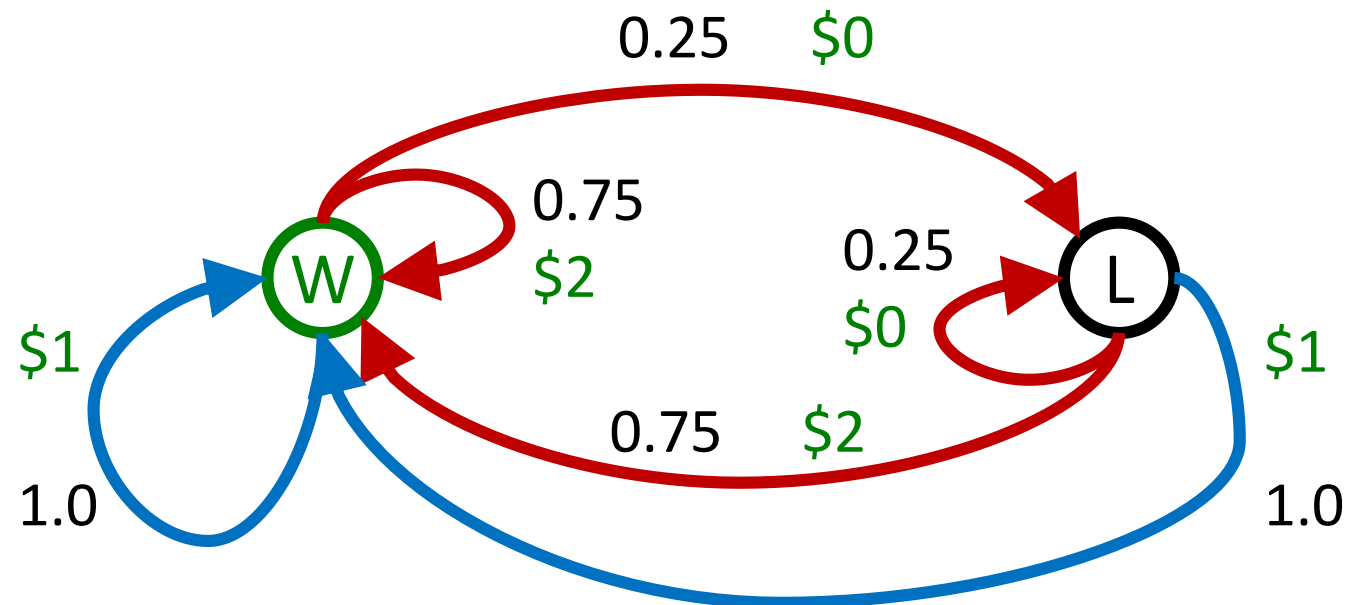


Offline Planning

- Solving MDPs is offline planning
 - You determine all quantities through computation
 - You need to know the details of the MDP
 - You do not actually play the game!

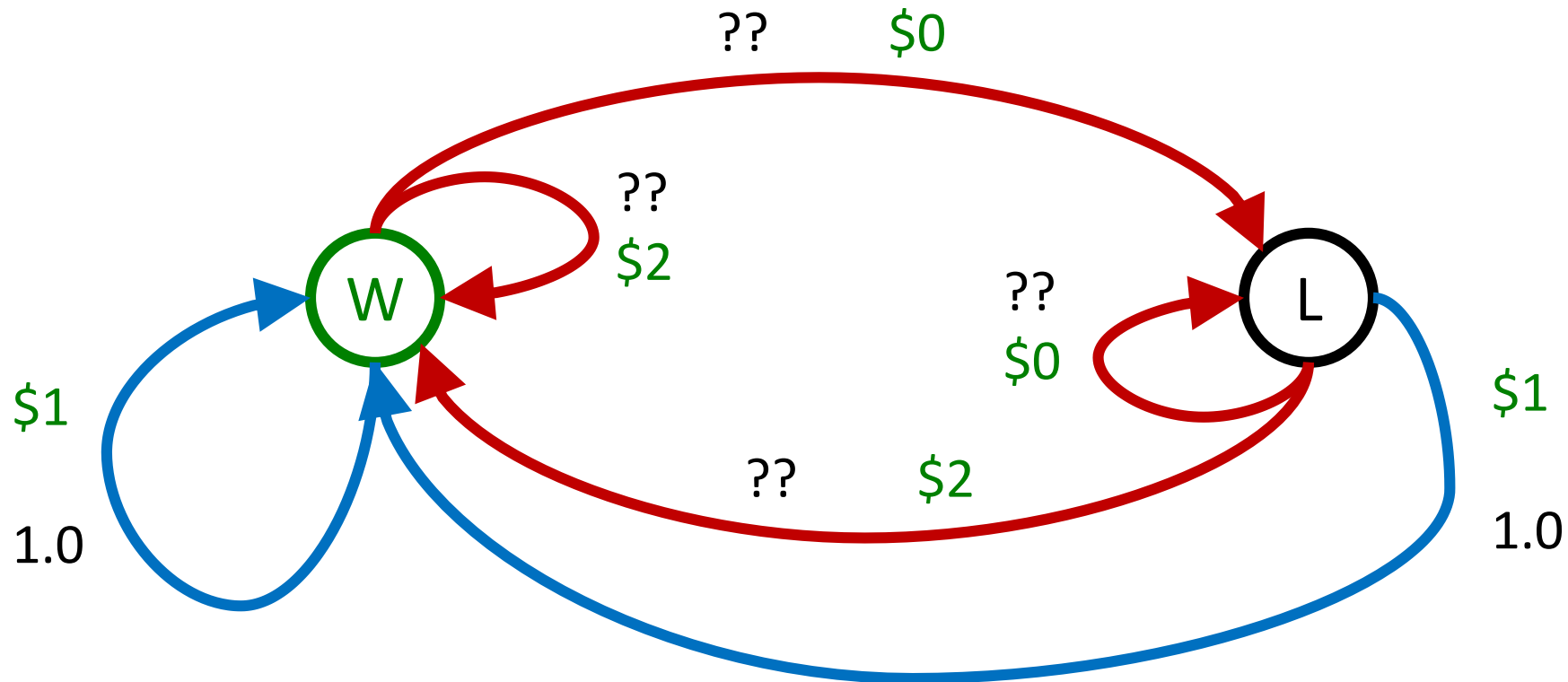
No discount
100 time steps

	Value
Play Red	150
Play Blue	100

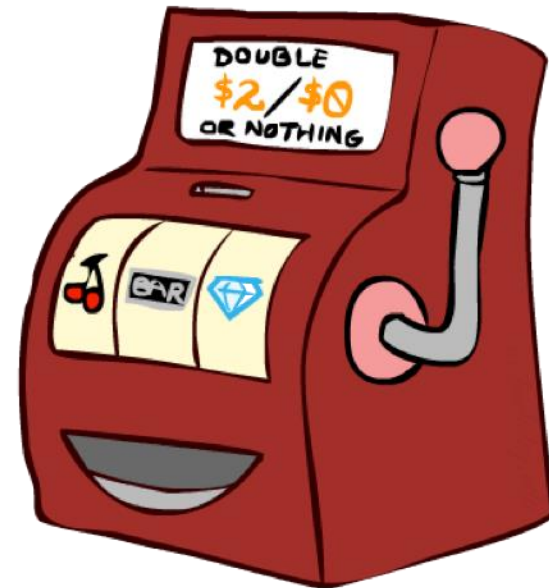


Online Planning

- Rules changed! Red's win chance is different.



Let's Play!



\$0 \$0 \$0 \$2 \$0
\$2 \$0 \$0 \$0 \$0

What Just Happened?

- That wasn't planning, it was learning!
 - Specifically, reinforcement learning
 - There was an MDP, but you couldn't solve it with just computation
 - You needed to actually act to figure it out
- Important ideas in reinforcement learning that came up
 - Exploration: you have to try unknown actions to get information
 - Exploitation: eventually, you have to use what you know
 - Regret: even if you learn intelligently, you make mistakes
 - Sampling: because of chance, you have to try things repeatedly
 - Difficulty: learning can be much harder than solving a known MDP

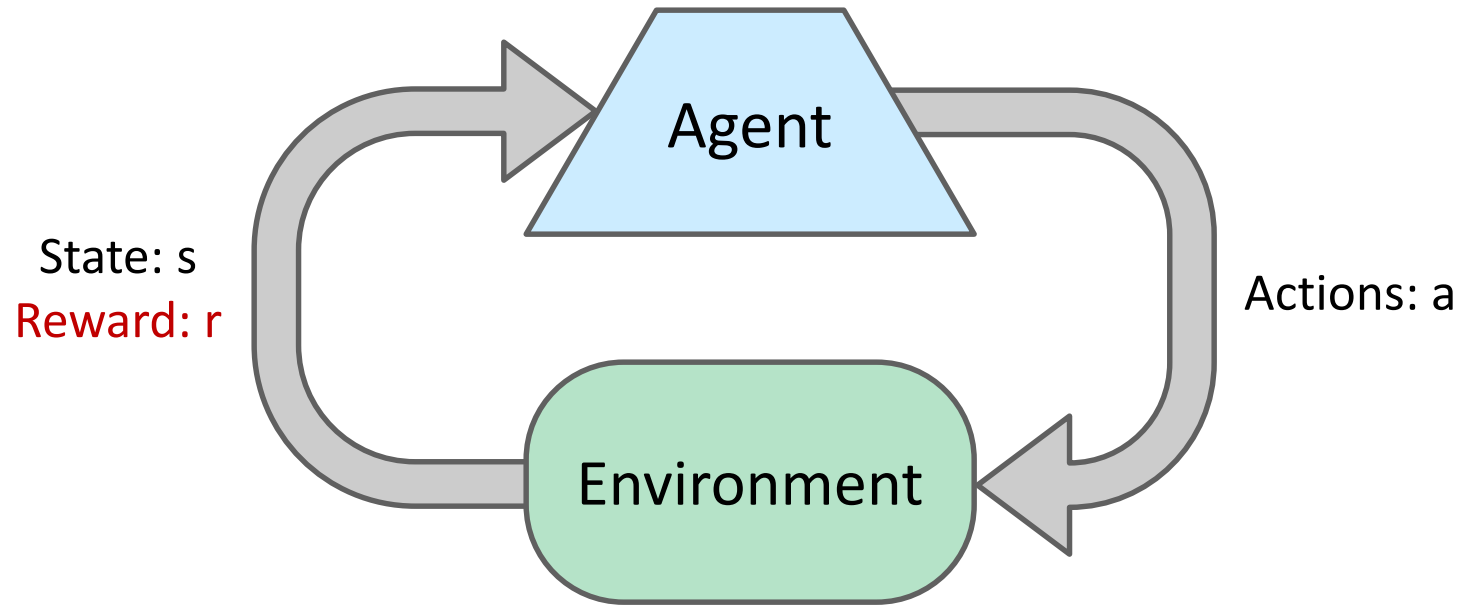


Reinforcement Learning

- Still assume a Markov decision process (MDP):
 - A set of states $s \in S$
 - A set of actions (per state) A
 - A model $T(s,a,s')$
 - A reward function $R(s,a,s')$
- Still looking for a policy $\pi(s)$
- New twist: don't know T or R
 - I.e. we don't know which states are good or what the actions do
 - Must actually try actions and states out to learn



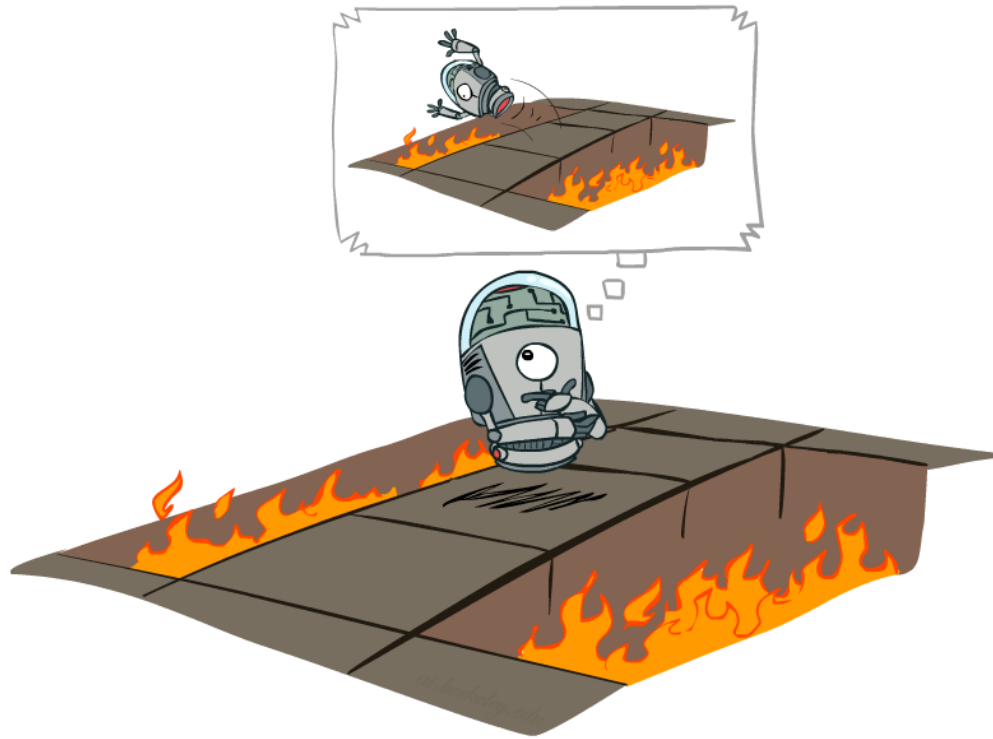
Reinforcement Learning



- **Basic idea:**

- Take actions and observe outcomes (new states, rewards)
- Learning is based on observed samples of outcomes
- Must (learn to) act so as to **maximize expected rewards**

Offline (MDPs) vs. Online (RL)



Offline Solution

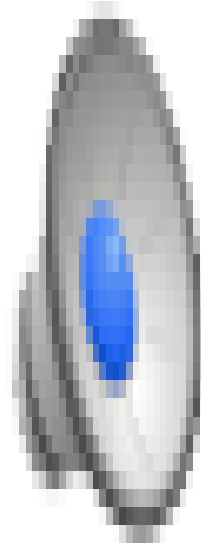


Online Learning

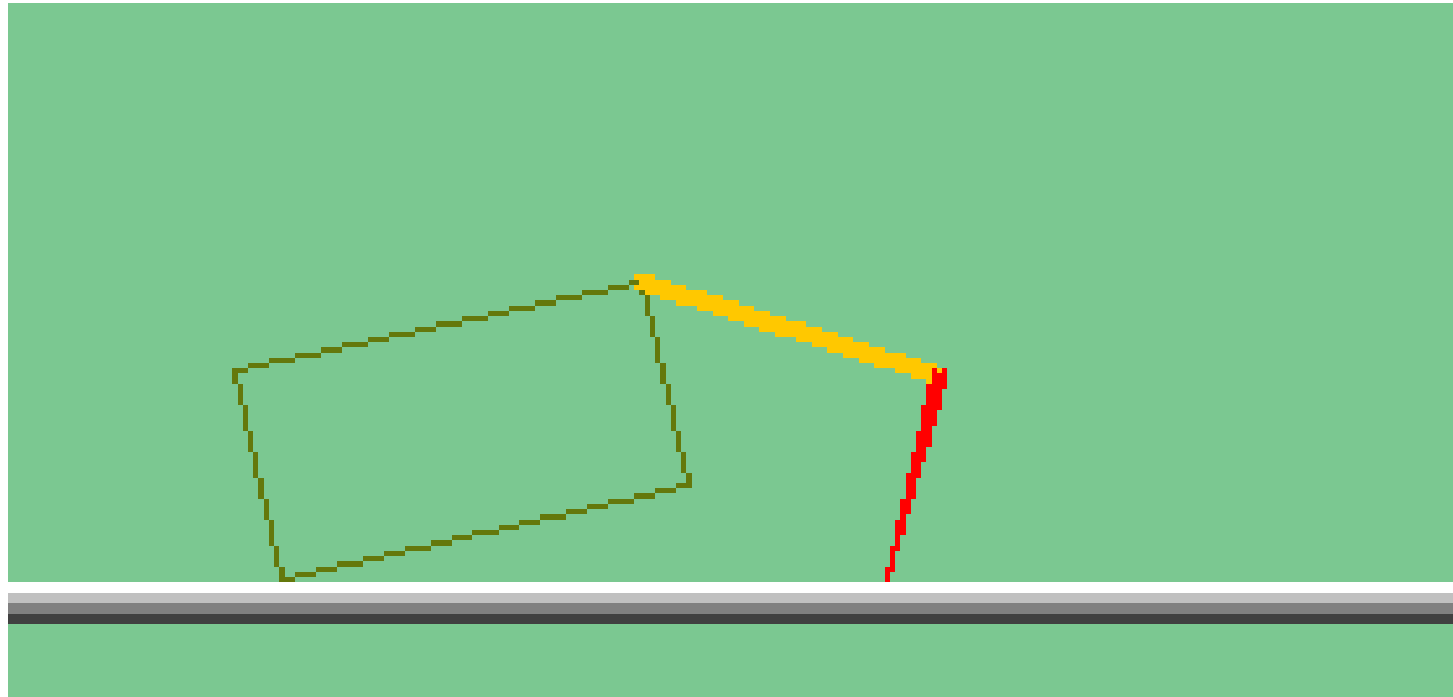
Cheetah



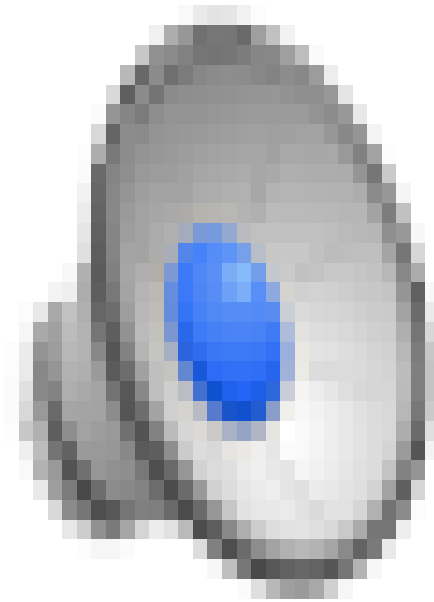
Atari



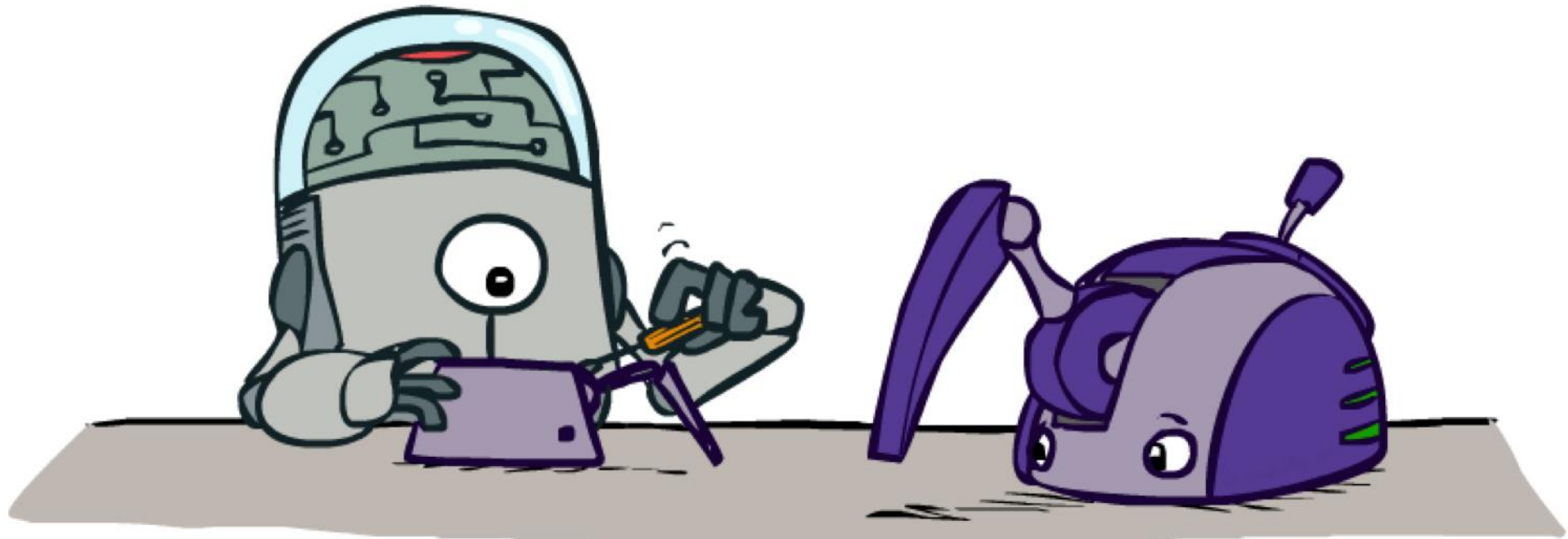
The Crawler!



Video of Demo Crawler Bot

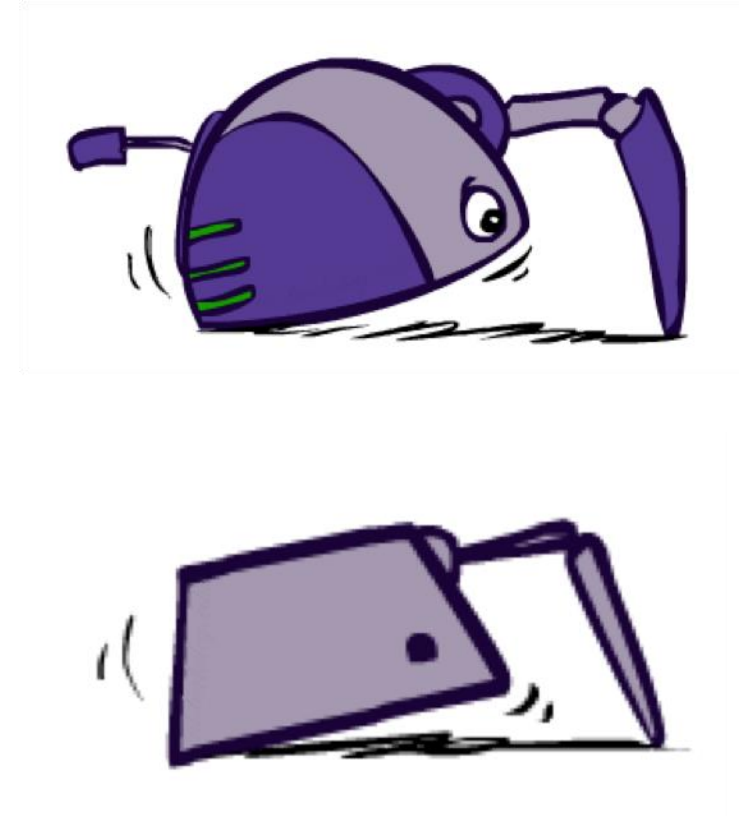


Model-Based Learning

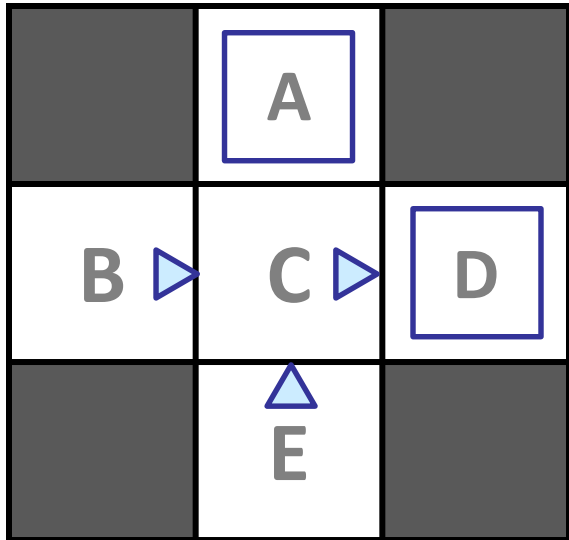


Model-Based Learning

- **Model-Based Idea:**
 - Learn an approximate model based on experiences
 - Solve for values as if the learned model was correct
- **Step 1: Learn empirical MDP model**
 - Count outcomes s' for each s, a
 - Normalize to give an estimate of $\hat{T}(s, a, s')$
 - Discover each $\hat{R}(s, a, s')$ when we experience (s, a, s')
- **Step 2: Solve the learned MDP**
 - For example, use value iteration, as before



Example: Model-Based Learning



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Learned Model

$$\hat{T}(s, a, s')$$

$T(B, \text{east}, C) = 1.00$
 $T(C, \text{east}, D) = 0.75$
 $T(C, \text{east}, A) = 0.25$
...

$$\hat{R}(s, a, s')$$

$R(B, \text{east}, C) = -1$
 $R(C, \text{east}, D) = -1$
 $R(D, \text{exit}, x) = +10$
...

Model-Based vs. Model-Free

Goal: Compute expected age of ShanghaiTech students

Known $P(A)$

$$E[A] = \sum_a P(a) \cdot a = 0.35 \times 20 + \dots$$

Without $P(A)$, instead collect samples $[a_1, a_2, \dots, a_N]$

Unknown $P(A)$: “Model Based”

$$\hat{P}(a) = \frac{\text{num}(a)}{N}$$
$$E[A] \approx \sum_a \hat{P}(a) \cdot a$$

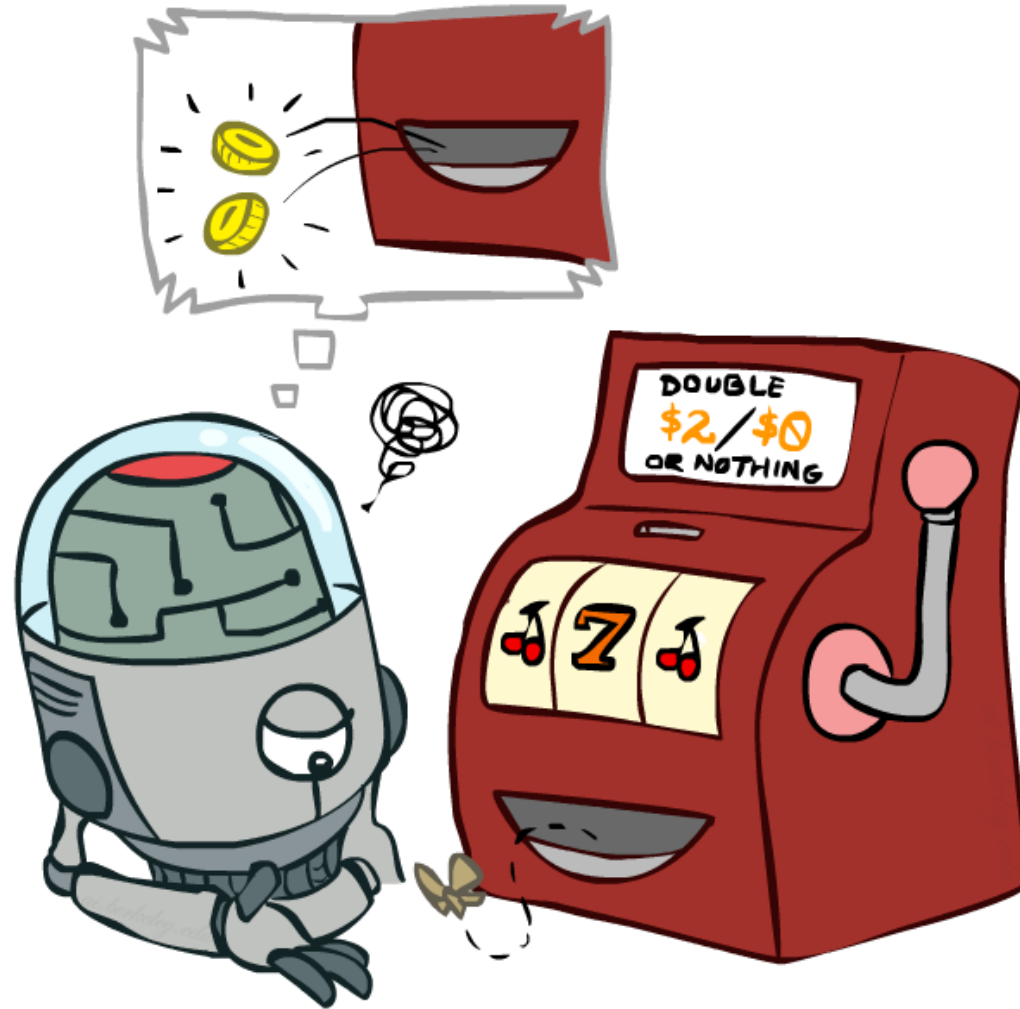
Why does this work? Because eventually you learn the right model.

Unknown $P(A)$: “Model Free”

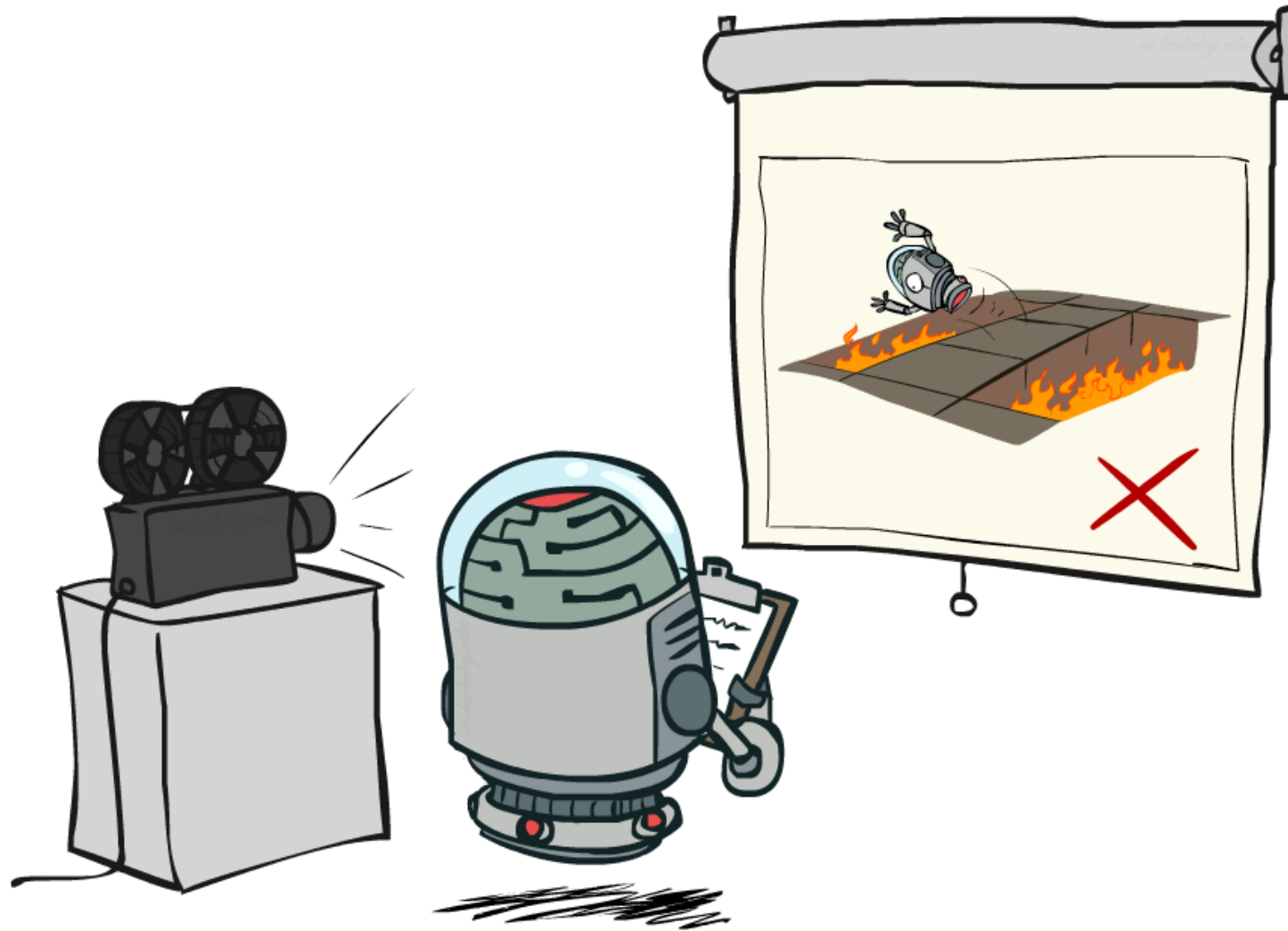
$$E[A] \approx \frac{1}{N} \sum_i a_i$$

Why does this work? Because samples appear with the right frequencies.

Model-Free Learning



Passive Reinforcement Learning



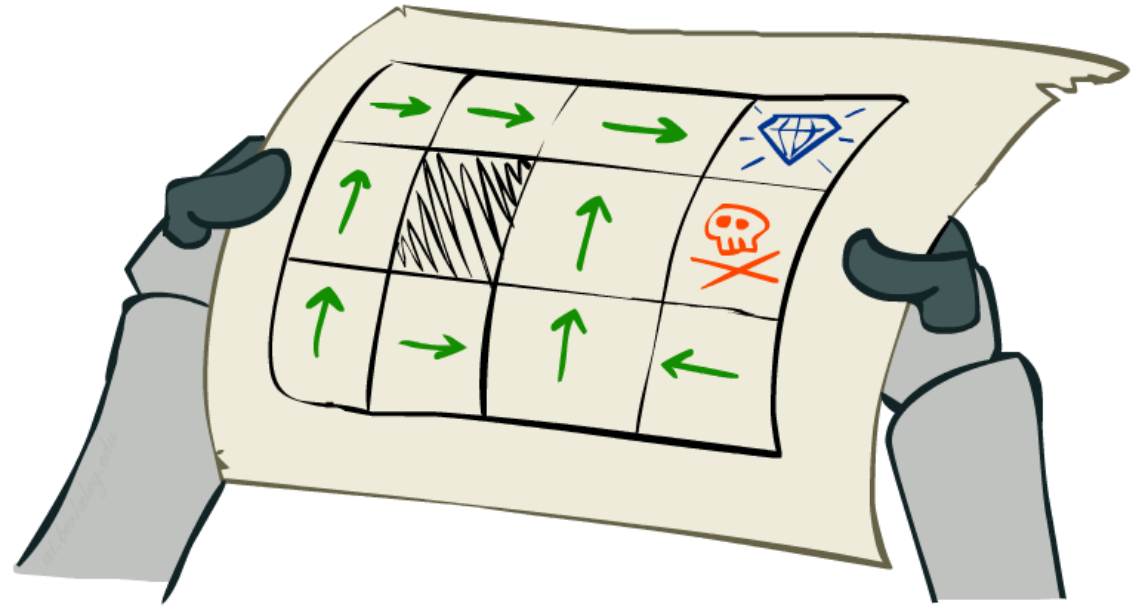
Passive Reinforcement Learning

- Simplified task: policy evaluation

- Input: a fixed policy $\pi(s)$
- You don't know the transitions $T(s,a,s')$
- You don't know the rewards $R(s,a,s')$
- Goal: learn the state values

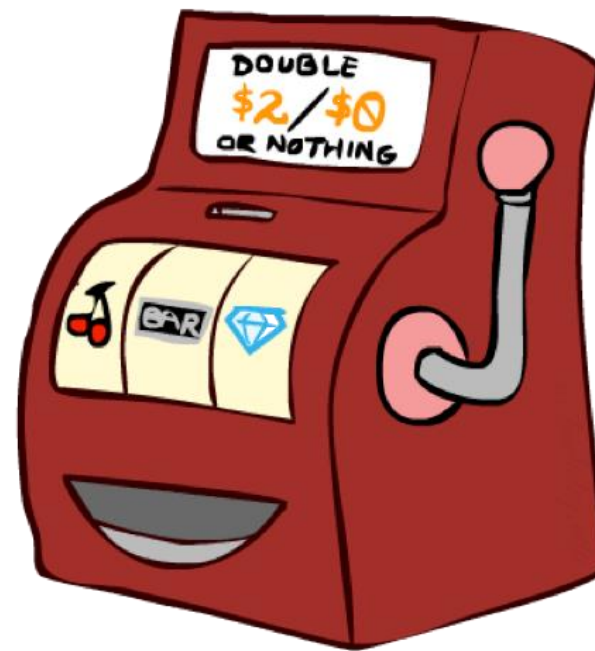
- In this case:

- No choice about what actions to take
- Just execute the policy and learn from experience
- This is NOT offline planning! You actually take actions in the world.



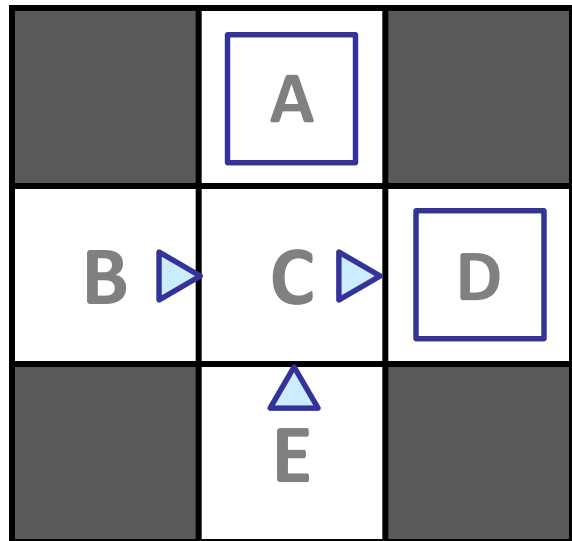
Direct Evaluation

- Goal: Compute values for each state under π
- Idea: Average together observed sample values
 - Act according to π
 - Every time you visit a state, write down what the sum of discounted rewards turned out to be
 - Average those samples
- This is called direct evaluation



Example: Direct Evaluation

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

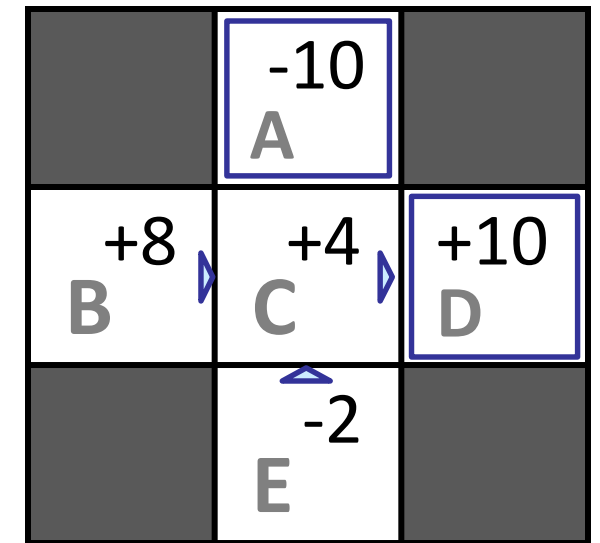
Output Values

	-10 A	
+8 B	+4 C	+10 D
	-2 E	

Problems with Direct Evaluation

- What's good about direct evaluation?
 - It's easy to understand
 - It doesn't require any knowledge of T , R
 - It eventually computes the correct average values, using just sample transitions
- What bad about it?
 - It wastes information about state connections
 - Each state must be learned separately
 - So, it takes a long time to learn

Output Values



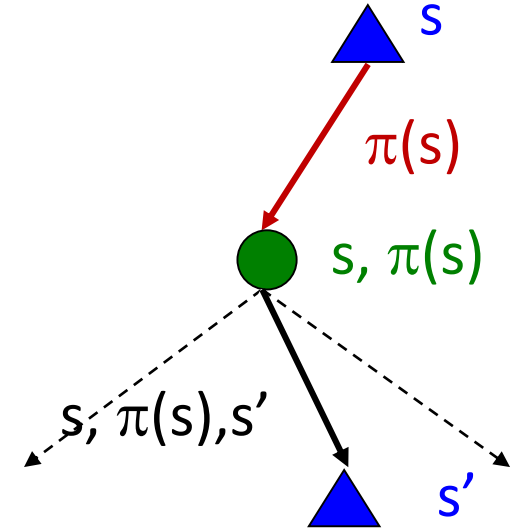
B goes to C, so we may use Bellman equation

Why Not Use Policy Evaluation?

- Simplified Bellman updates calculate V for a fixed policy:

$$V_0^\pi(s) = 0$$

$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$



- This approach fully exploits the connections between the states
- Unfortunately, we need T and R to do it!
- Key question: how can we do this update to V without knowing T and R ?
 - In other words, how do we take a weighted average without knowing the weights?

Sample-Based Policy Evaluation?

- We want to compute these averages:

$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$

- Idea: Take samples of outcomes s' (by doing the action!) and average

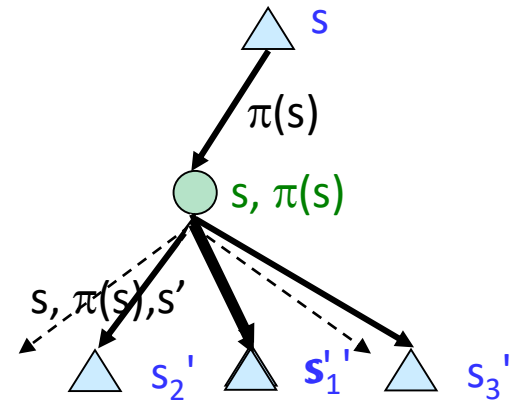
$$sample_1 = R(s, \pi(s), s'_1) + \gamma V_k^{\pi}(s'_1)$$

$$sample_2 = R(s, \pi(s), s'_2) + \gamma V_k^{\pi}(s'_2)$$

...

$$sample_n = R(s, \pi(s), s'_n) + \gamma V_k^{\pi}(s'_n)$$

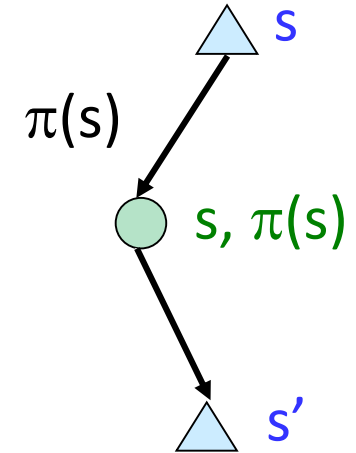
$$V_{k+1}^{\pi}(s) \leftarrow \frac{1}{n} \sum_i sample_i$$



*But we can't rewind time
to get sample after
sample from state s !*

Temporal Difference Learning

- Big idea: learn immediately from every experience!
 - Update $V(s)$ each time we experience a transition (s, a, s', r)
- Temporal difference learning of values
 - (Policy still fixed, still doing evaluation!)
 - Move the value towards the sample



Sample of $V(s)$: $sample = R(s, \pi(s), s') + \gamma V^\pi(s')$

Update to $V(s)$: $V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + (\alpha)sample$

Same update: $V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s))$

Exponential Moving Average

- Exponential moving average

- The running interpolation update: $\bar{x}_n = (1 - \alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n$
- Makes recent samples more important
- Forgets about the past (distant past values were wrong anyway)

$$\bar{x}_n = \frac{x_n + (1 - \alpha) \cdot x_{n-1} + (1 - \alpha)^2 \cdot x_{n-2} + \dots}{1 + (1 - \alpha) + (1 - \alpha)^2 + \dots}$$

- Decreasing learning rate (alpha) can give converging averages