CS 3346: Artificial Intelligence I

Markov Decision Processes II
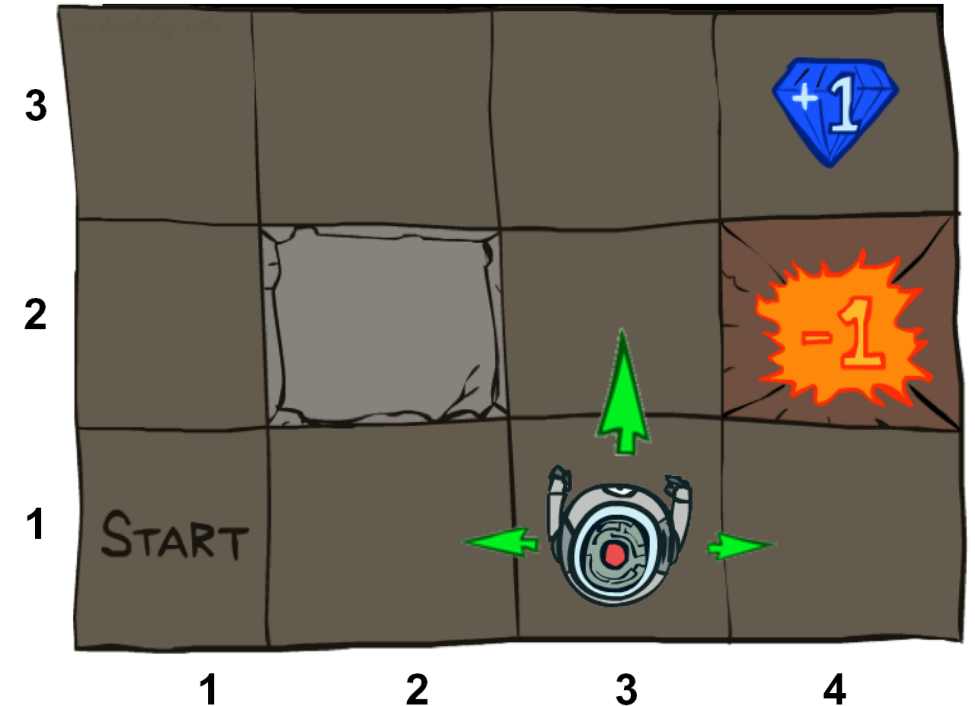

Instructor: Yalda Mohsenzadeh, Western University

Example: Grid World

- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path

- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put

- The agent receives rewards
  - Small "living" reward each step (can be negative)
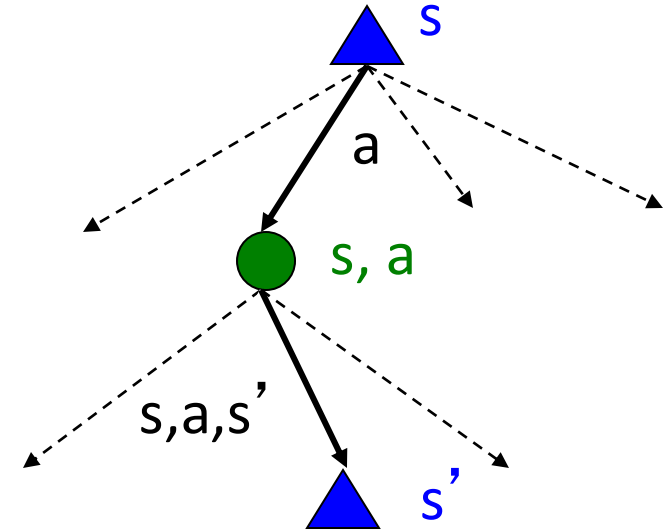 - Big rewards come at the end (good or bad)

- Goal: maximize sum of rewards
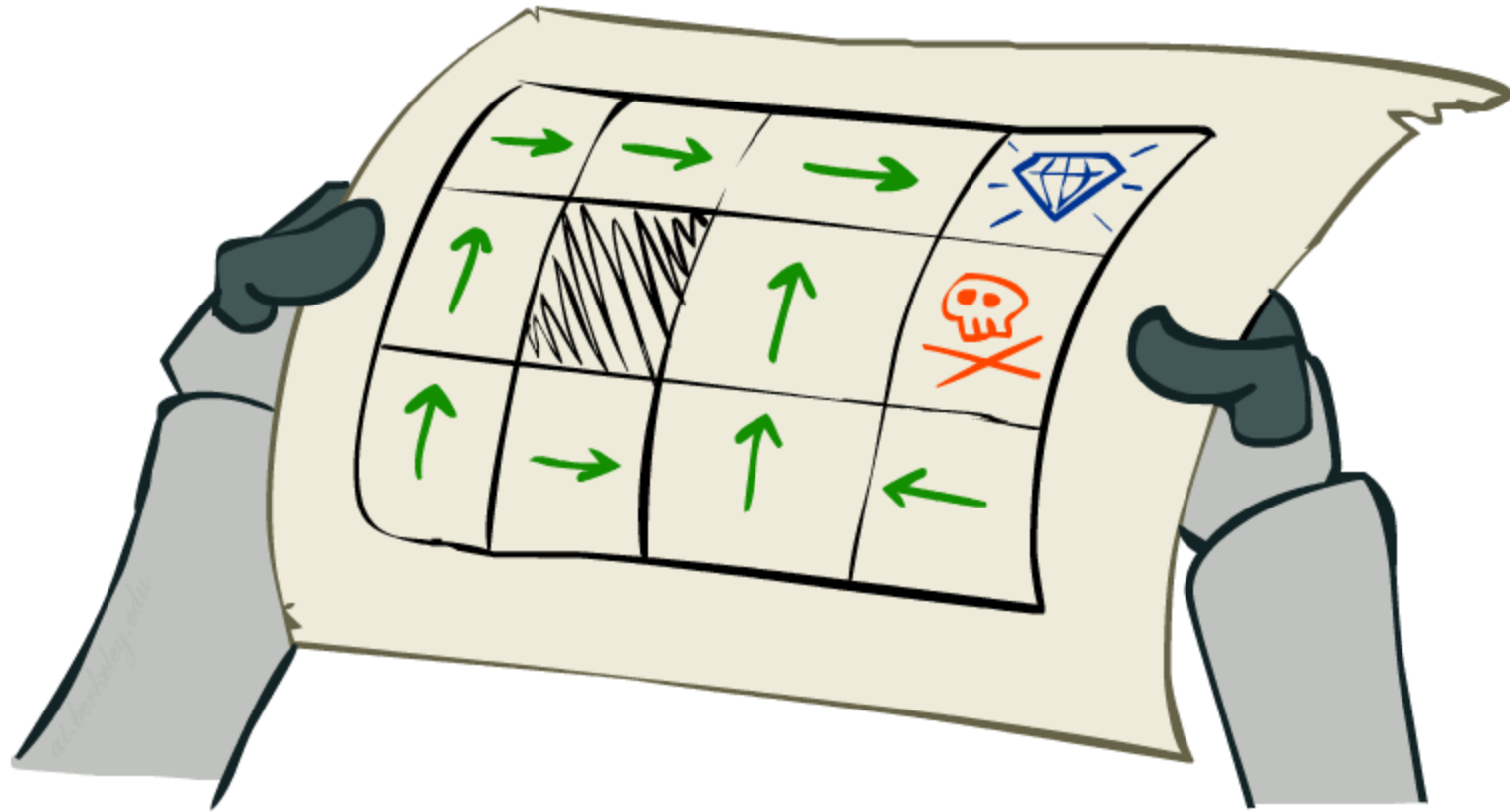
Recap: MDPs

o **Markov decision processes:**
  o Set of states S
  o Start state s_0
  o Set of actions A
  o Transitions P(s'|s,a) (or T(s,a,s'))
  o Rewards R(s,a,s') (and discount γ)

o **MDP quantities:**
  o Policy = map of states to actions
  o Utility = sum of (discounted) rewards
  o Values = expected future utility from a state (max node)
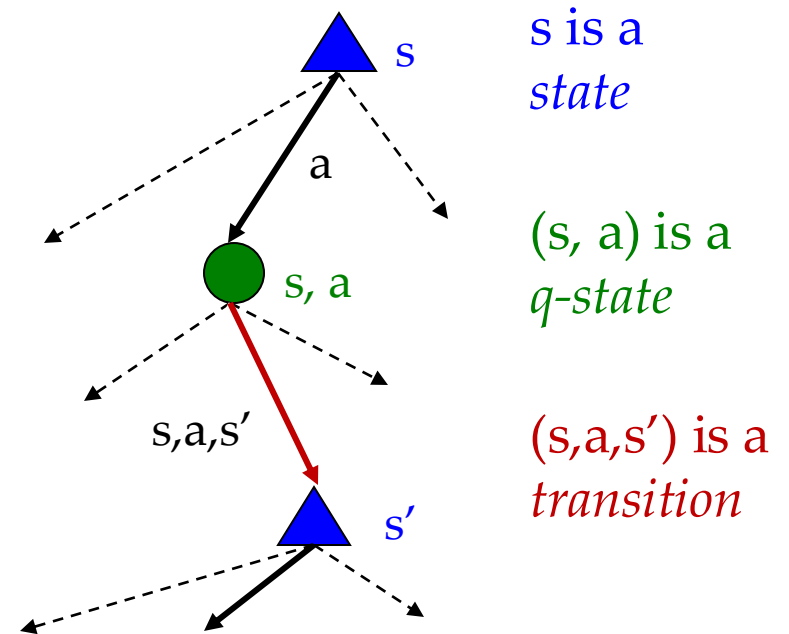 o Q-values = expected future utility from a q-state (chance node)
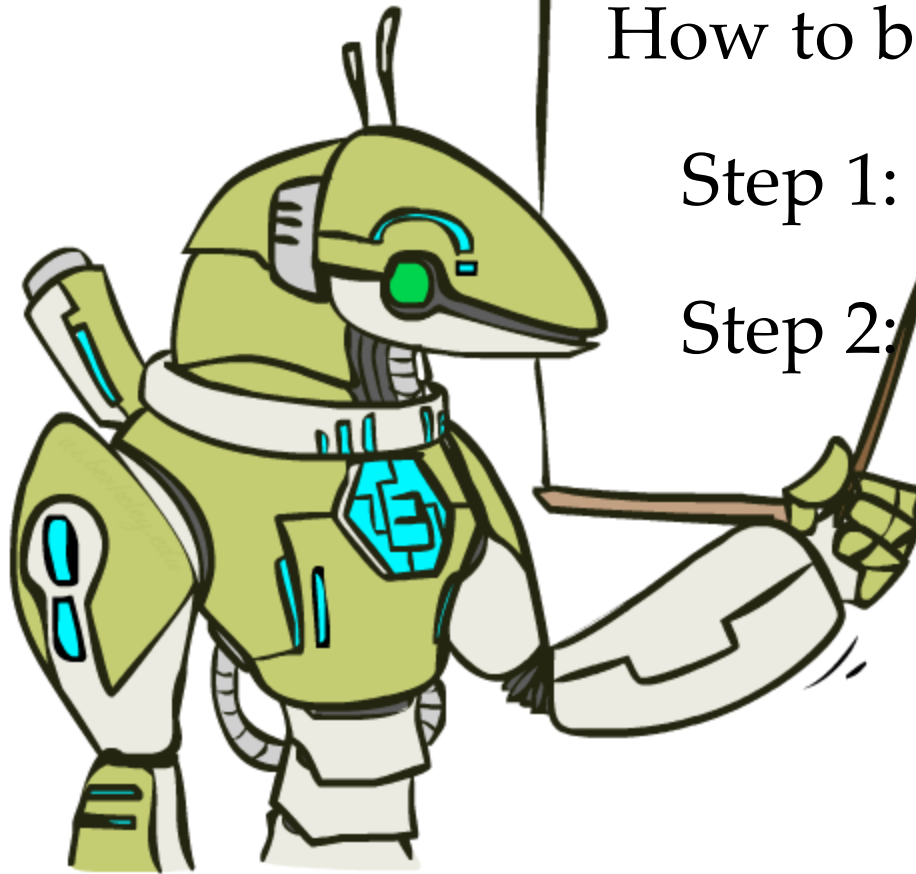
Solving MDPs

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

s is a *state*

(s, a) is a *q-state*

(s,a,s') is a *transition*

s

a

s, a

s,a,s'

s'

The Bellman Equations

How to be optimal:

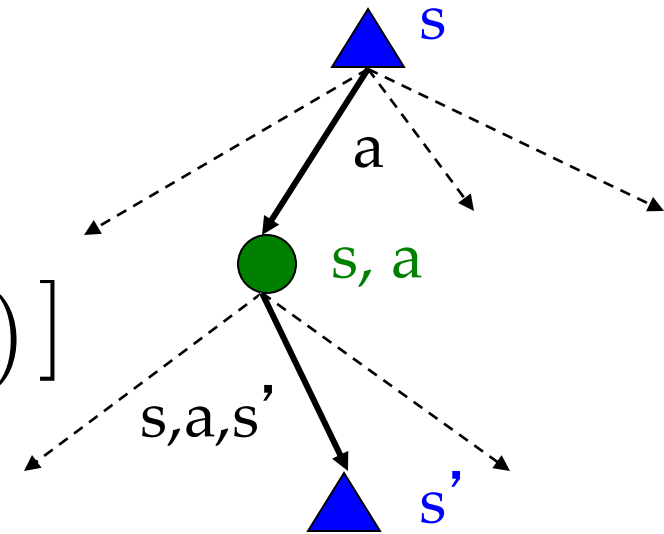Step 1: Take correct first action

Step 2: Keep being optimal

Values of States: Bellman Equation
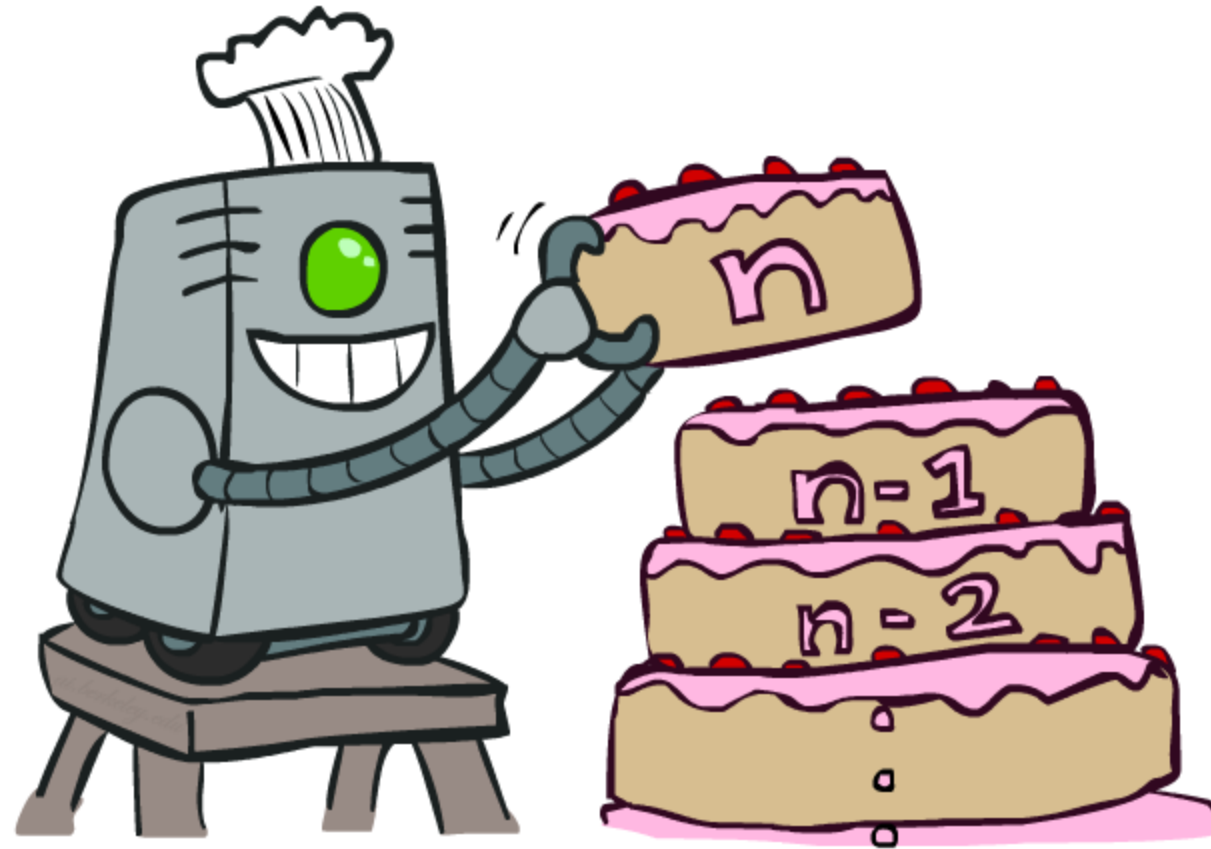
o Recursive definition of value:

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

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

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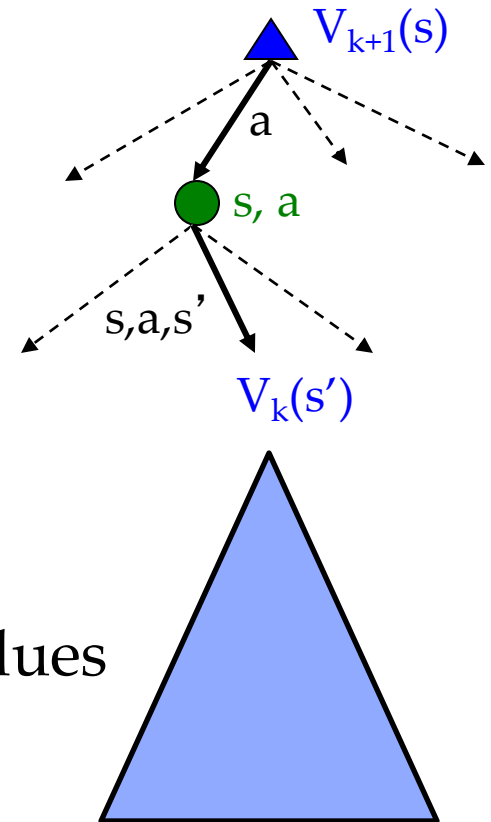
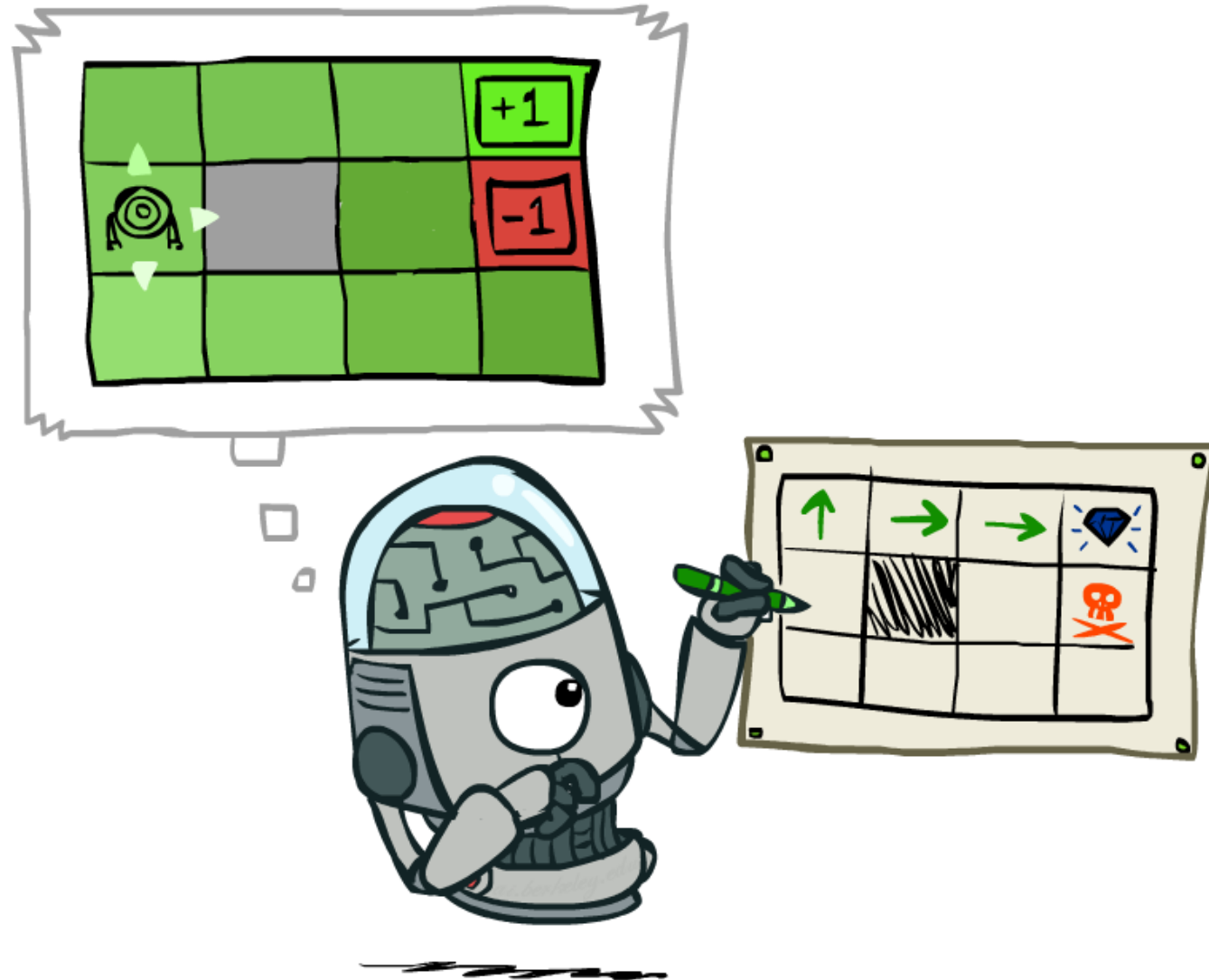# Value Iteration

Value Iteration: Dynamic Programming

o Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero

o Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

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

$V_{k+1}(s)$

a

s, a

s,a,s'

$V_k(s')$

o V = B(V) Where B is the Bellman update operator

o Repeat until convergence, which yields V*

o Complexity of each iteration: $O(S^2 A)$

o Theorem: Value Iteration will converge to unique optimal values
  o Basic idea: approximations get refined towards optimal values
  o Policy may converge long before values do

Policy Extraction

Computing Actions from Values

○ Let's imagine we have the optimal values V*(s)

○ How should we act?
  ○ It's not obvious!

○ We need to do a mini-expectimax (one step)

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

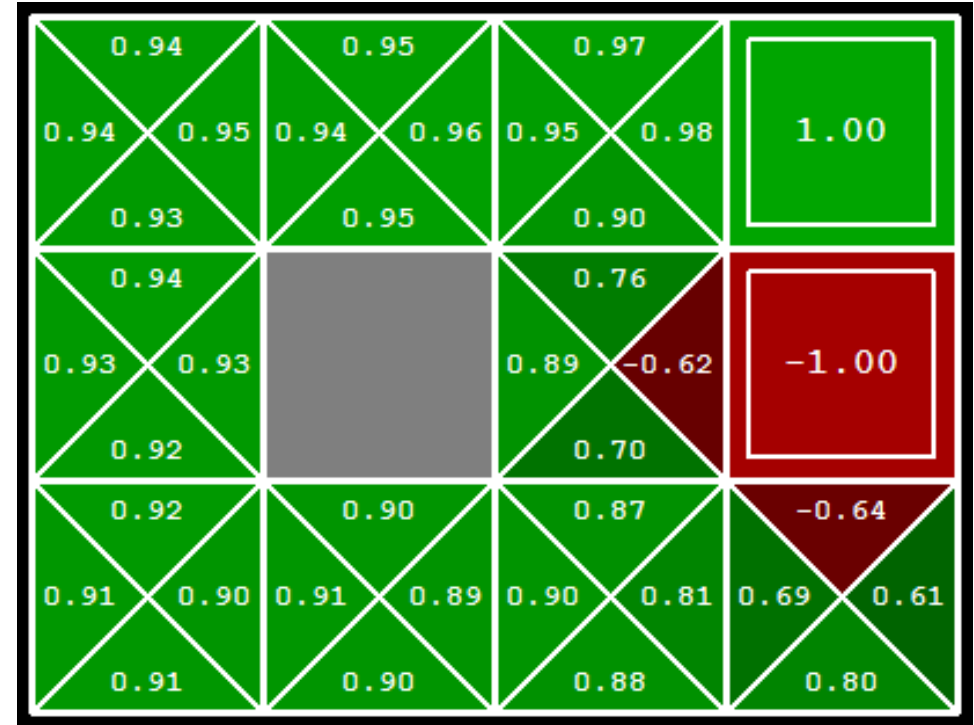○ This is called policy extraction, since it gets the policy implied by the values

Computing Actions from Q-Values

○ Let's imagine we have the optimal q-values:
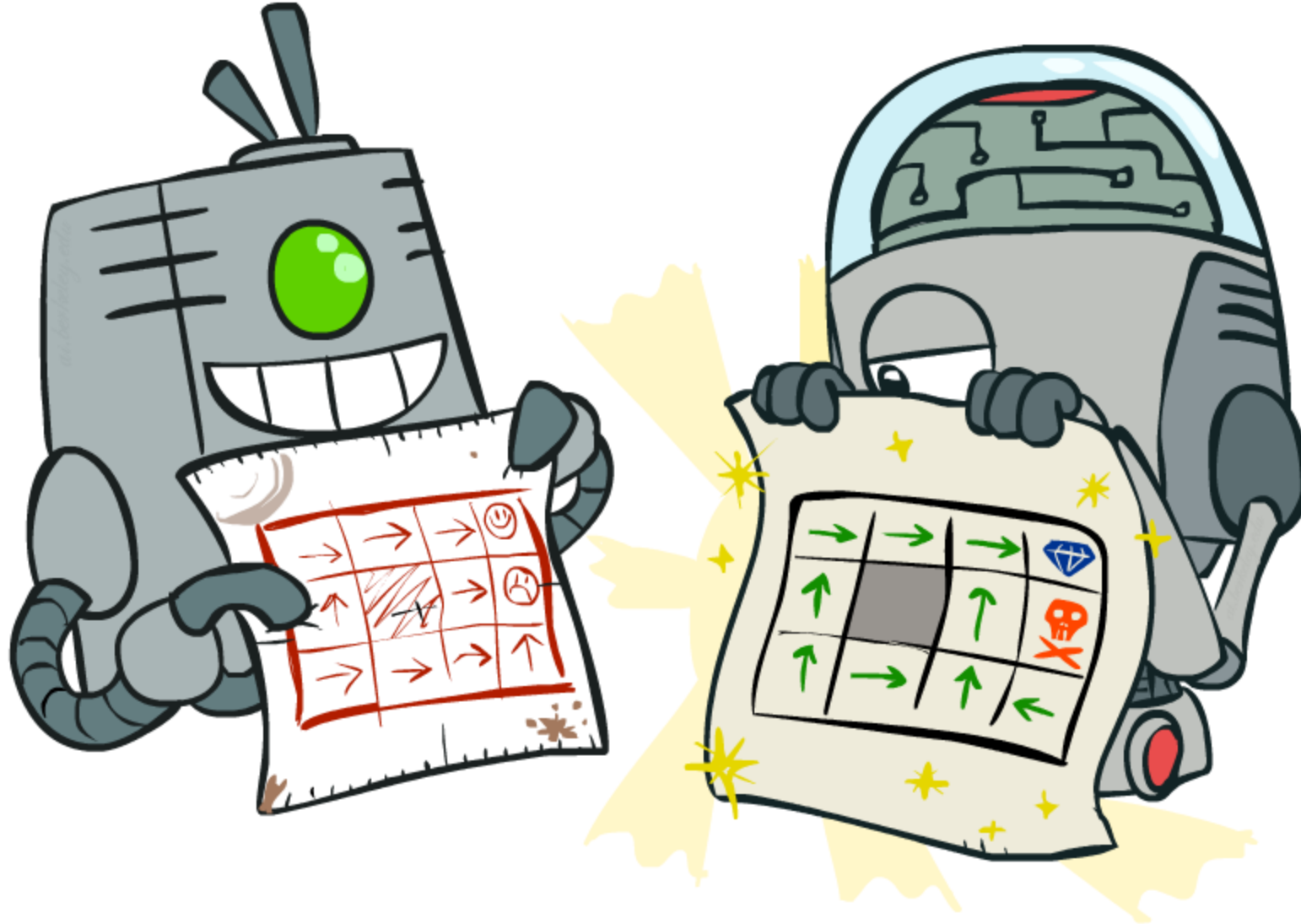
○ How should we act?
  ○ Completely trivial to decide!

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

○ Important lesson: actions are easier to select from q-values than values!
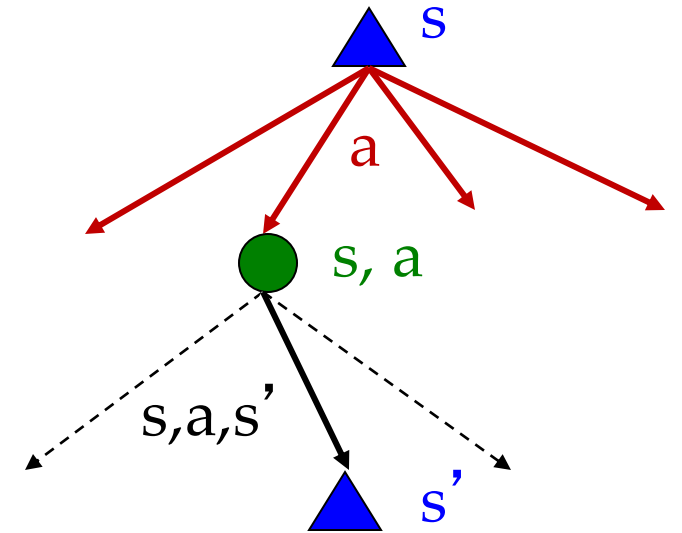
Policy Methods

Problems with Value Iteration

○ Value iteration repeats the Bellman updates:

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

s

a

s, a

s,a,s'

s'

○ Problem 1: It's slow – O(S²A) per iteration

○ Problem 2: The "max" at each state rarely changes

○ Problem 3: The policy often converges long before the values

k=12

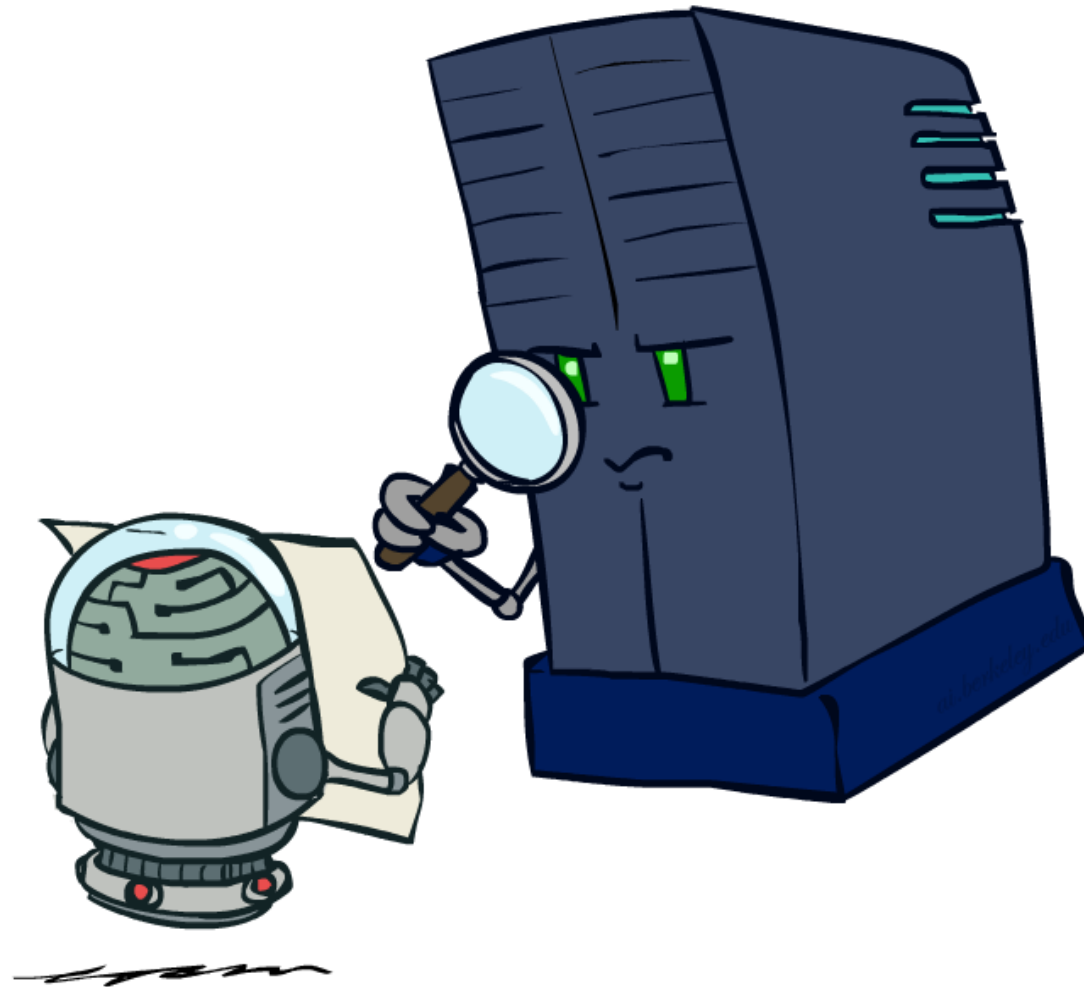Noise = 0.2
Discount = 0.9
Living reward = 0

k=100

Noise = 0.2
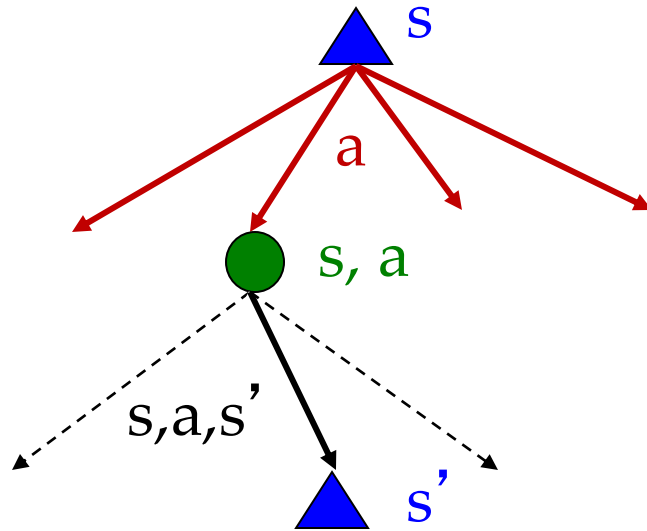Discount = 0.9
Living reward = 0

Policy Iteration

o Alternative approach for optimal values:

  o Step 1: Policy Evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence

  o Step 2: Policy Improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values

  o Repeat steps until policy converges

o This is Policy Iteration

  o It's still optimal!

  o Can converge (much) faster under some conditions
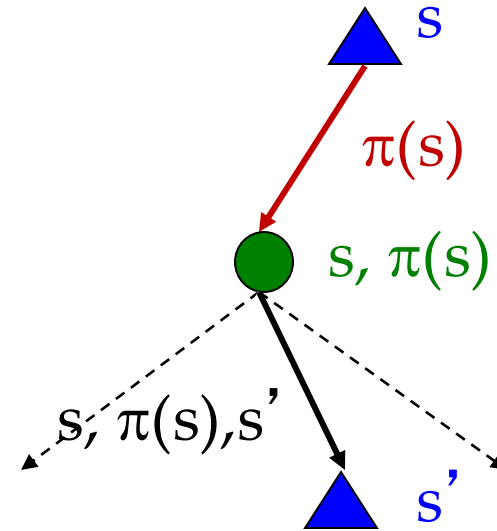
Policy Evaluation

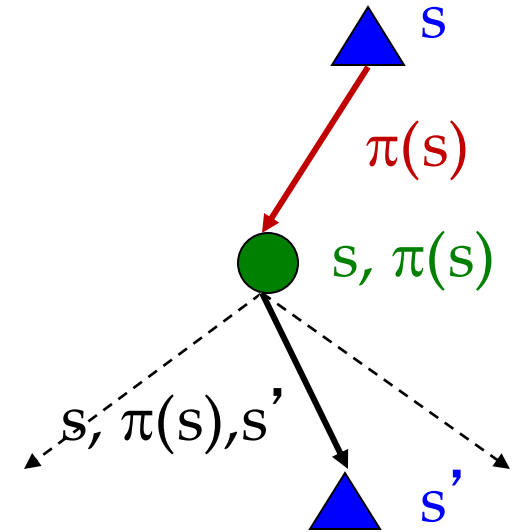# Fixed Policies

Do the optimal action

Do what π says to do

o Expectimax trees max over all actions to compute the optimal values

o If we fixed some policy π(s), then the tree would be simpler – only one action per state

    o … though the tree's value would depend on which policy we fixed
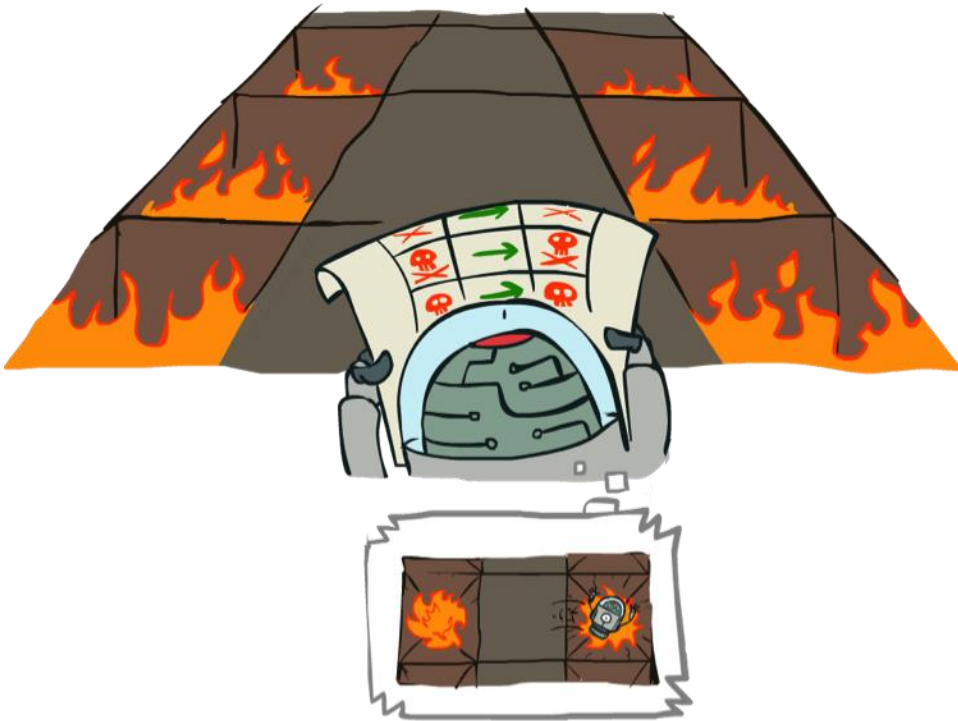
Utilities for a Fixed Policy

○ Another basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy

○ Define the utility of a state s, under a fixed policy π:

Vπ(s) = expected total discounted rewards starting in s and following π

○ Recursive relation (one-step look-ahead / Bellman equation):

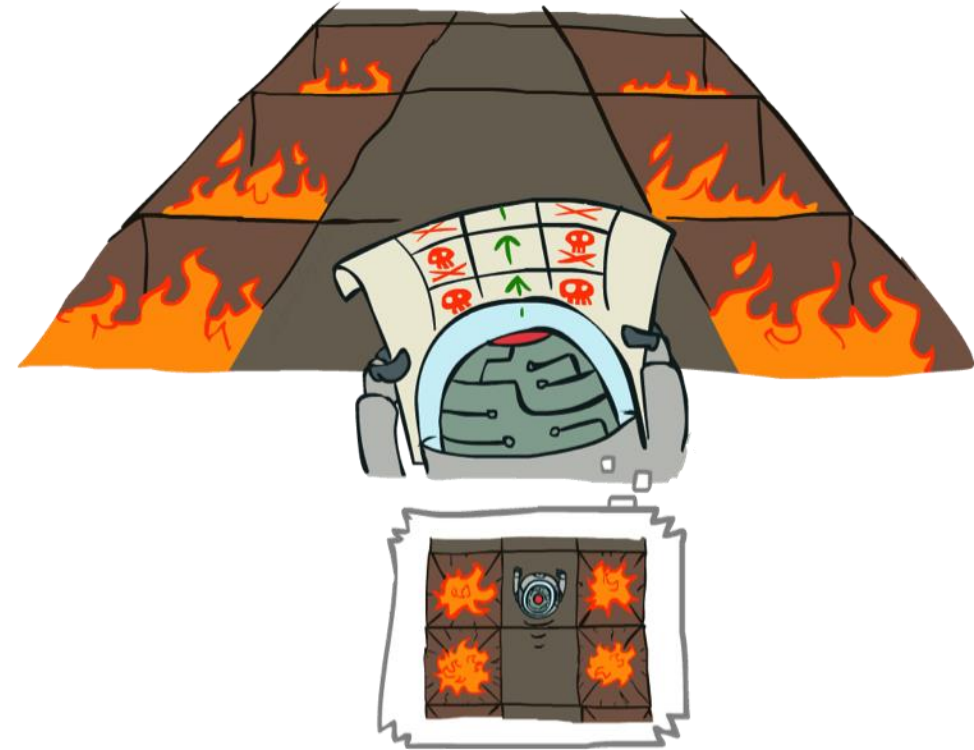$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^\pi(s')]$$

s

π(s)

s, π(s)

s, π(s),s'

s'

Example: Policy Evaluation

Always Go Right

Always Go Forward

Example: Policy Evaluation
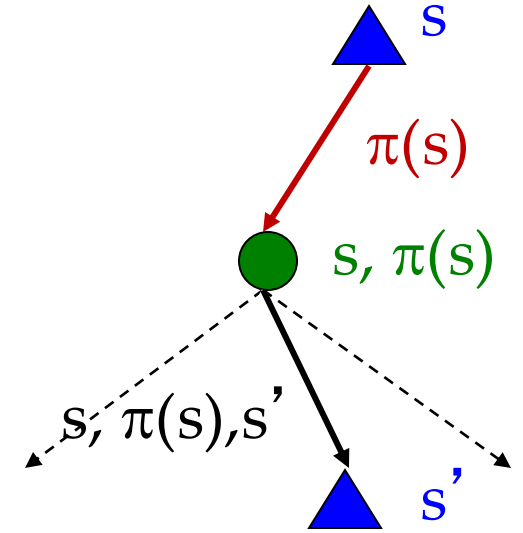
Always Go Right

Always Go Forward

Policy Evaluation

○ How do we calculate the V's for a fixed policy π?

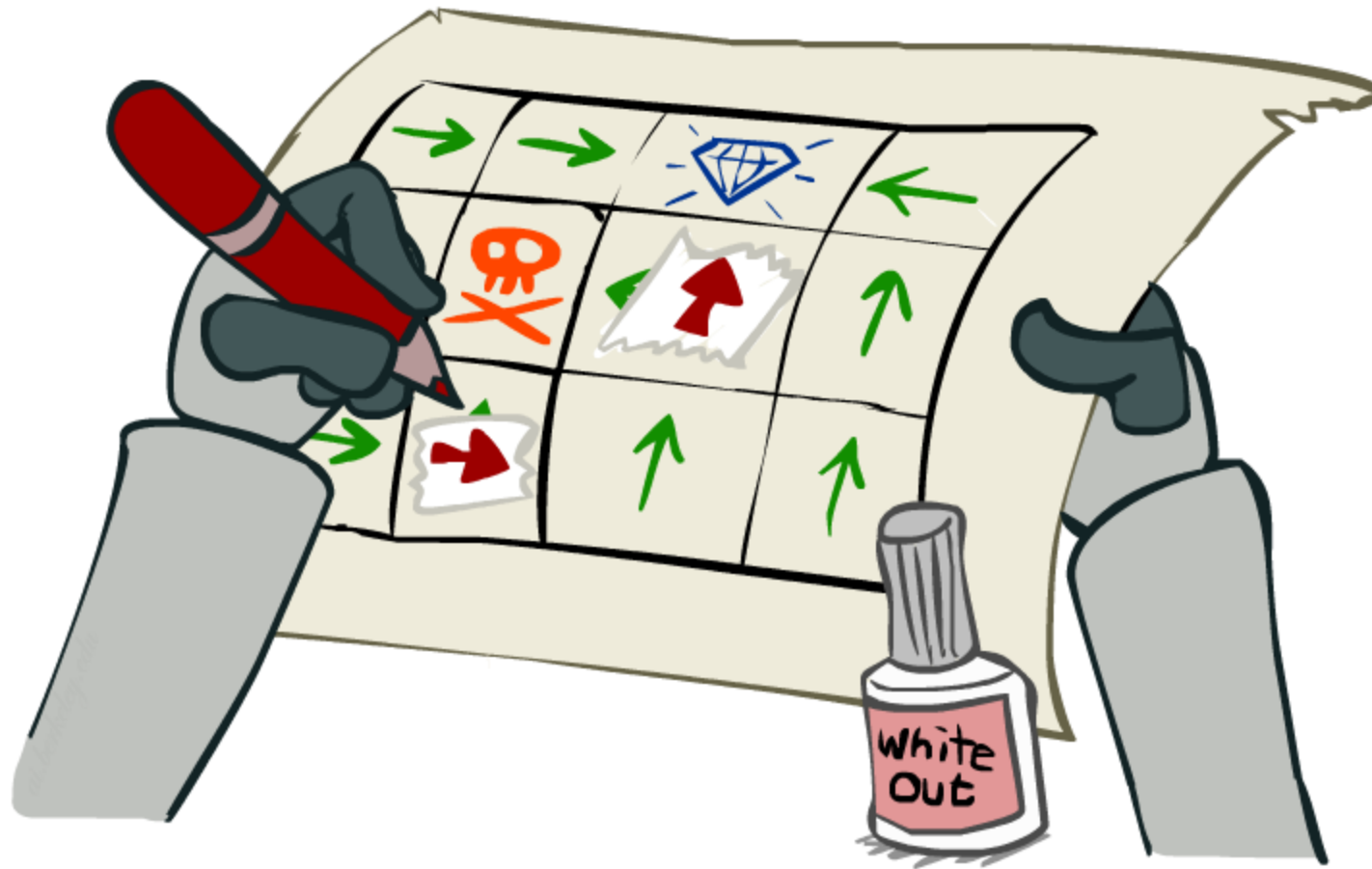○ Idea 1: Turn recursive Bellman equations into updates
(like value iteration)

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

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

○ Efficiency: O(S²) per iteration

○ Idea 2: Without the maxes, the Bellman equations are just a linear system
  ○ Solve the system of equations

s

π(s)

s, π(s)

s, π(s),s'

s'

Policy Iteration

Policy Iteration

- Evaluation: For fixed current policy π, find values with policy evaluation:
  - Iterate until values converge:

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

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

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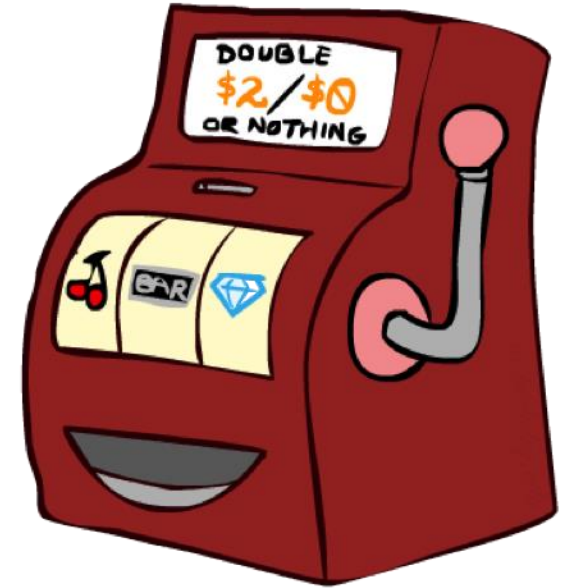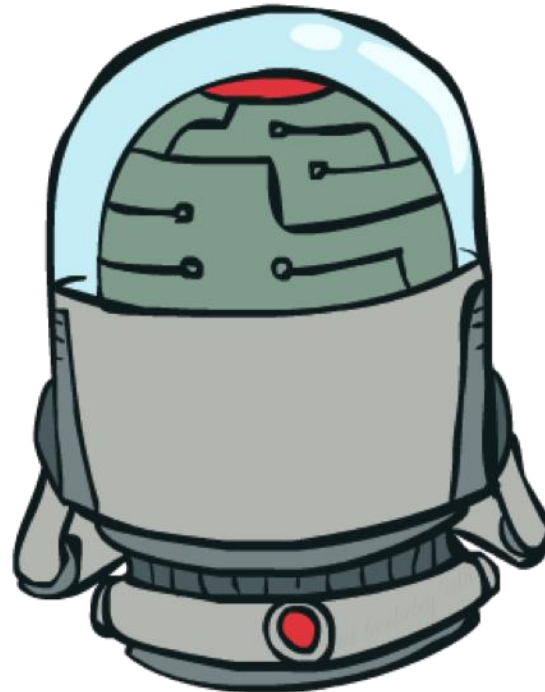
Comparison

o Both value iteration and policy iteration compute the same thing (all optimal values)

o In value iteration:
   o Every iteration updates both the values and (implicitly) the policy
   o We don't track the policy, but taking the max over actions implicitly recomputes it

o In policy iteration:
   o We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
   o After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
   o The new policy will be better (or we're done)

o Both are dynamic programming approaches for solving MDPs
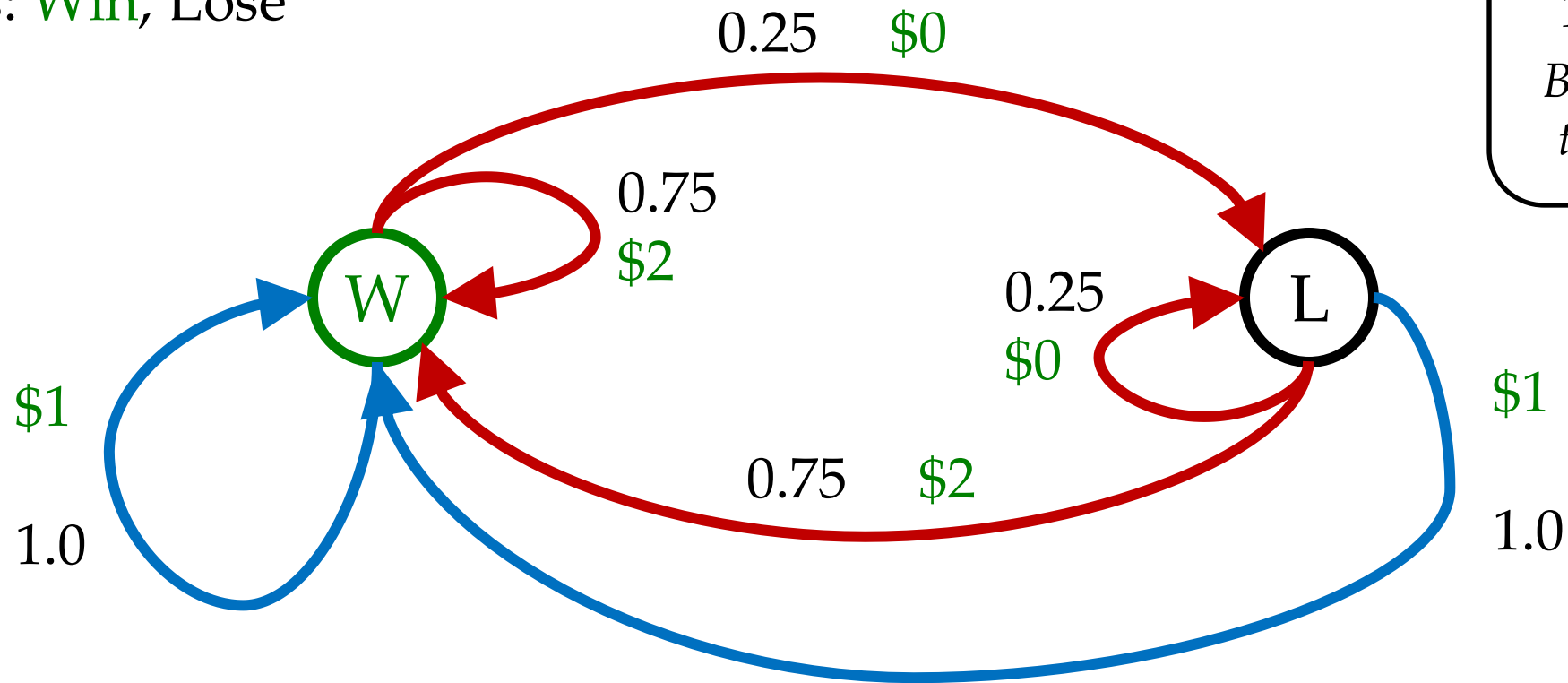
Summary: MDP Algorithms

o **So you want to….**
  o Compute optimal values: use value iteration or policy iteration
  o Compute values for a particular policy: use policy evaluation
  o Turn your values into a policy: use policy extraction (one-step lookahead)

o **These all look the same!**
  o They basically are – they are all variations of Bellman updates
  o They all use one-step lookahead expectimax fragments
  o They differ only in whether we plug in a fixed policy or max over actions

Double Bandits

Double-Bandit MDP

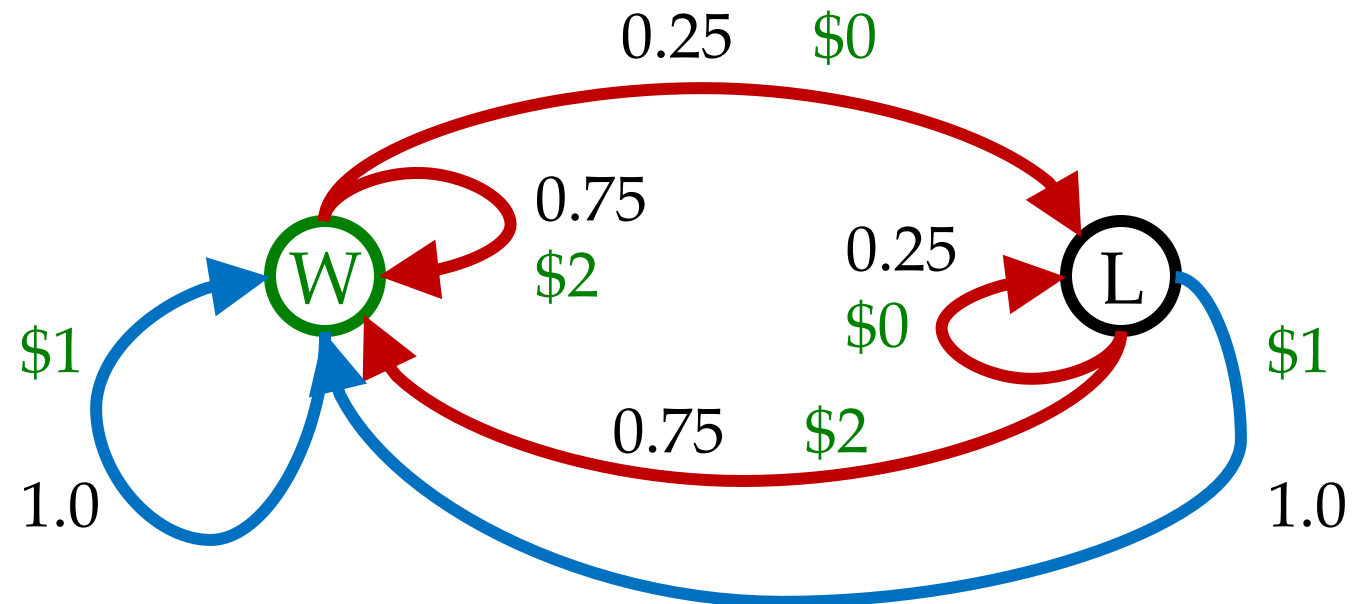- Actions: *Blue*, *Red*
- States: Win, Lose

No discount
100 time steps
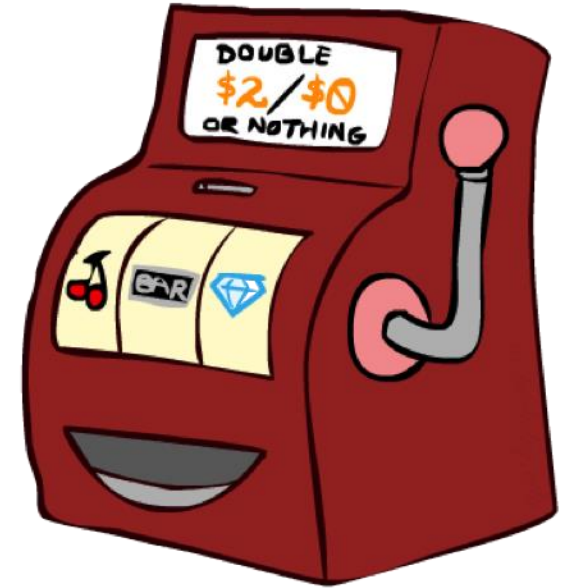Both states have
the same value

Offline Planning

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

No discount
100 time steps
Both states have the same value

	Value
Play Red	150
Play Blue	100

Let's Play!

$2 $2 $0 $2 $2

$2 $2 $0 $0 $0

Online Planning

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

Let's Play!

DOUBLE
$2/$0
OR NOTHING

$0 $0 $0 $2 $0
$2 $0 $0 $0 $0

What Just Happened?

o That wasn't planning, it was learning!
  o Specifically, reinforcement learning
  o There was an MDP, but you couldn't solve it with just computation
  o You needed to actually act to figure it out

o Important ideas in reinforcement learning that came up
  o Exploration: you have to try unknown actions to get information
  o Exploitation: eventually, you have to use what you know
  o Regret: even if you learn intelligently, you make mistakes
  o Sampling: because of chance, you have to try things repeatedly
  o Difficulty: learning can be much harder than solving a known MDP