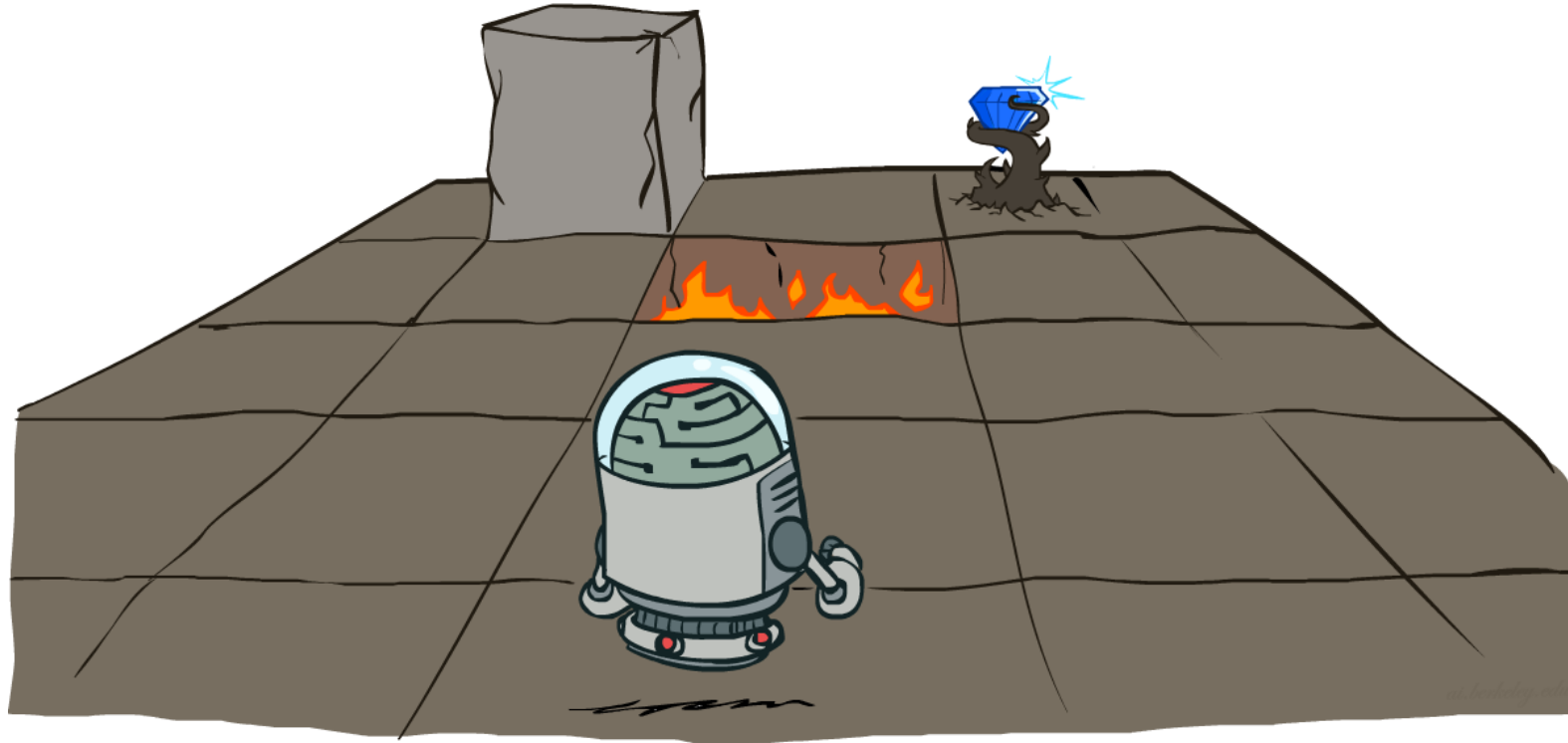


CS 188: Artificial Intelligence

Markov Decision Processes



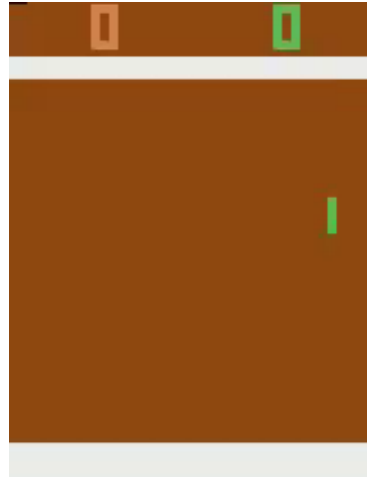
Instructor: Evgeny Pobachienko – UC Berkeley

[Slides adapted from Dan Klein, Pieter Abbeel, Ketrina Yim, Stuart Russell, Satish Rao, and many others.]

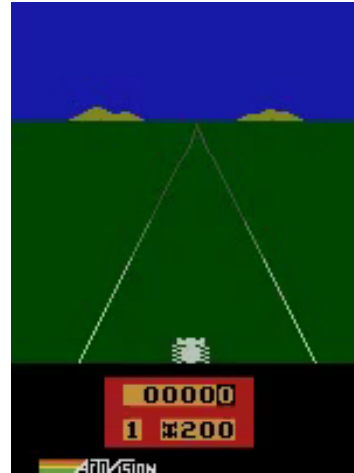
Deep Reinforcement Learning

2013

Atari (DQN)
[Deepmind]



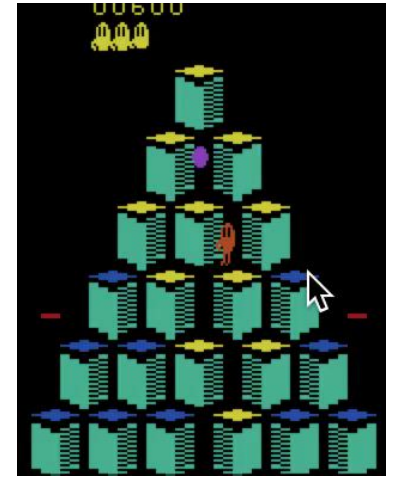
Pong



Enduro



Beamrider



Q*bert

Deep Reinforcement Learning

2013

Atari (DQN)
[Deepmind]

2015

AlphaGo
[Deepmind]



AlphaGo Silver et al, Nature 2015

AlphaGoZero Silver et al, Nature 2017

AlphaZero Silver et al, 2017

Tian et al, 2016; Maddison et al, 2014; Clark et al, 2015

Deep Reinforcement Learning

2013

Atari (DQN)
[Deepmind]

2015

AlphaGo
[Deepmind]

2016

3D locomotion (TRPO+GAE)
[Berkeley]



[Schulman, Moritz, Levine, Jordan, Abbeel, ICLR 2016]

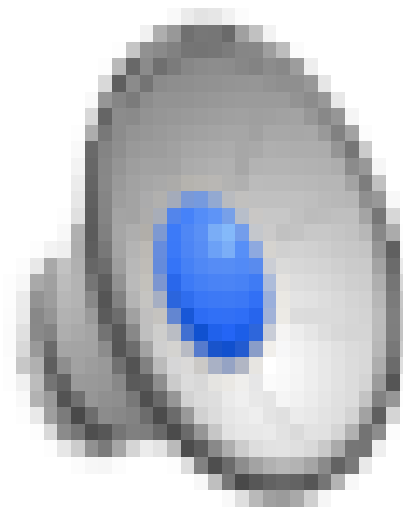
Deep Reinforcement Learning

2013
Atari (DQN)
[Deepmind]

2015
AlphaGo
[Deepmind]

2016
3D locomotion (TRPO+GAE)
[Berkeley]

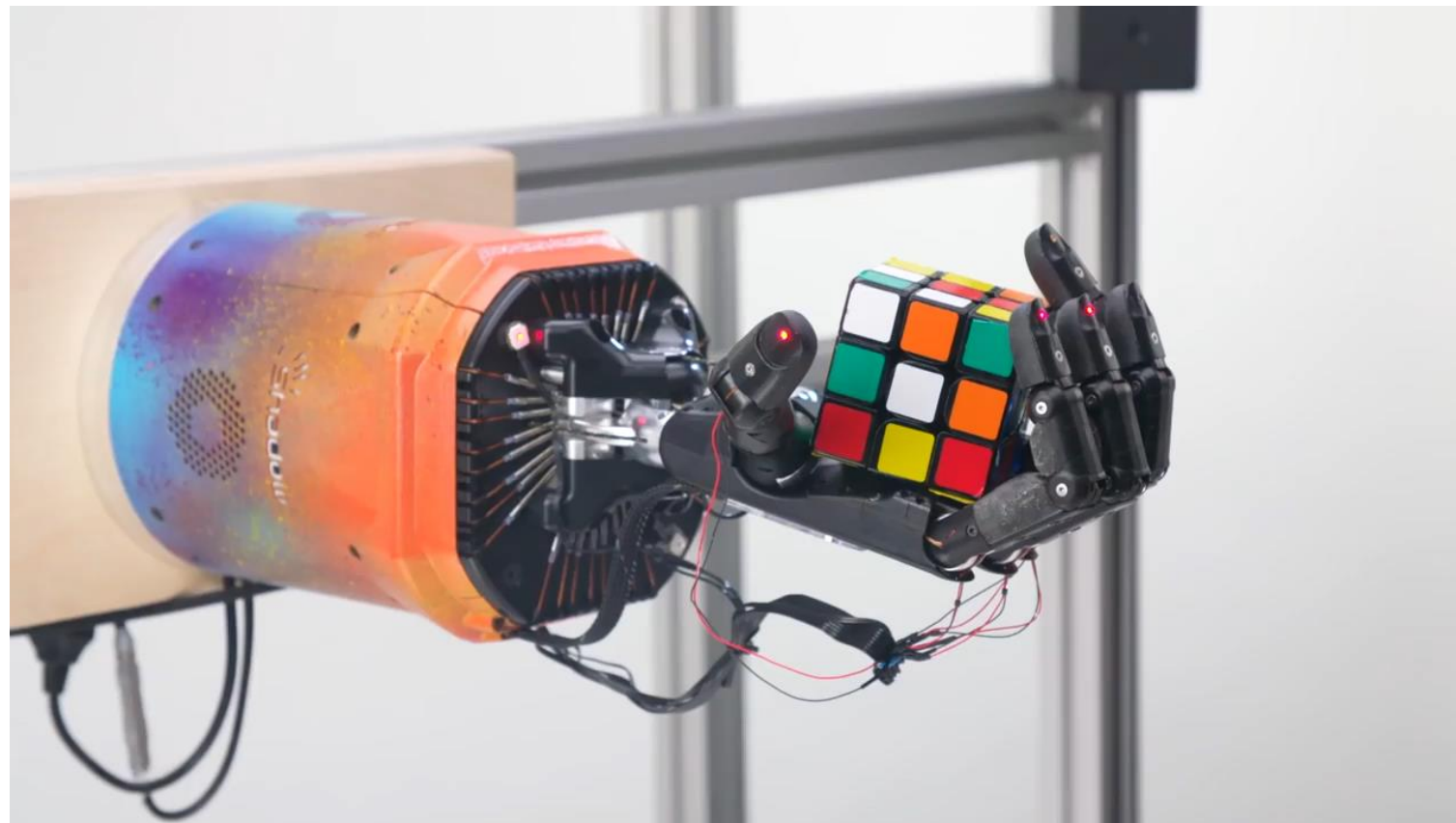
2016
Real Robot Manipulation (GPS)
[Berkeley]



[Levine*, Finn*, Darrell, Abbeel, JMLR 2016]

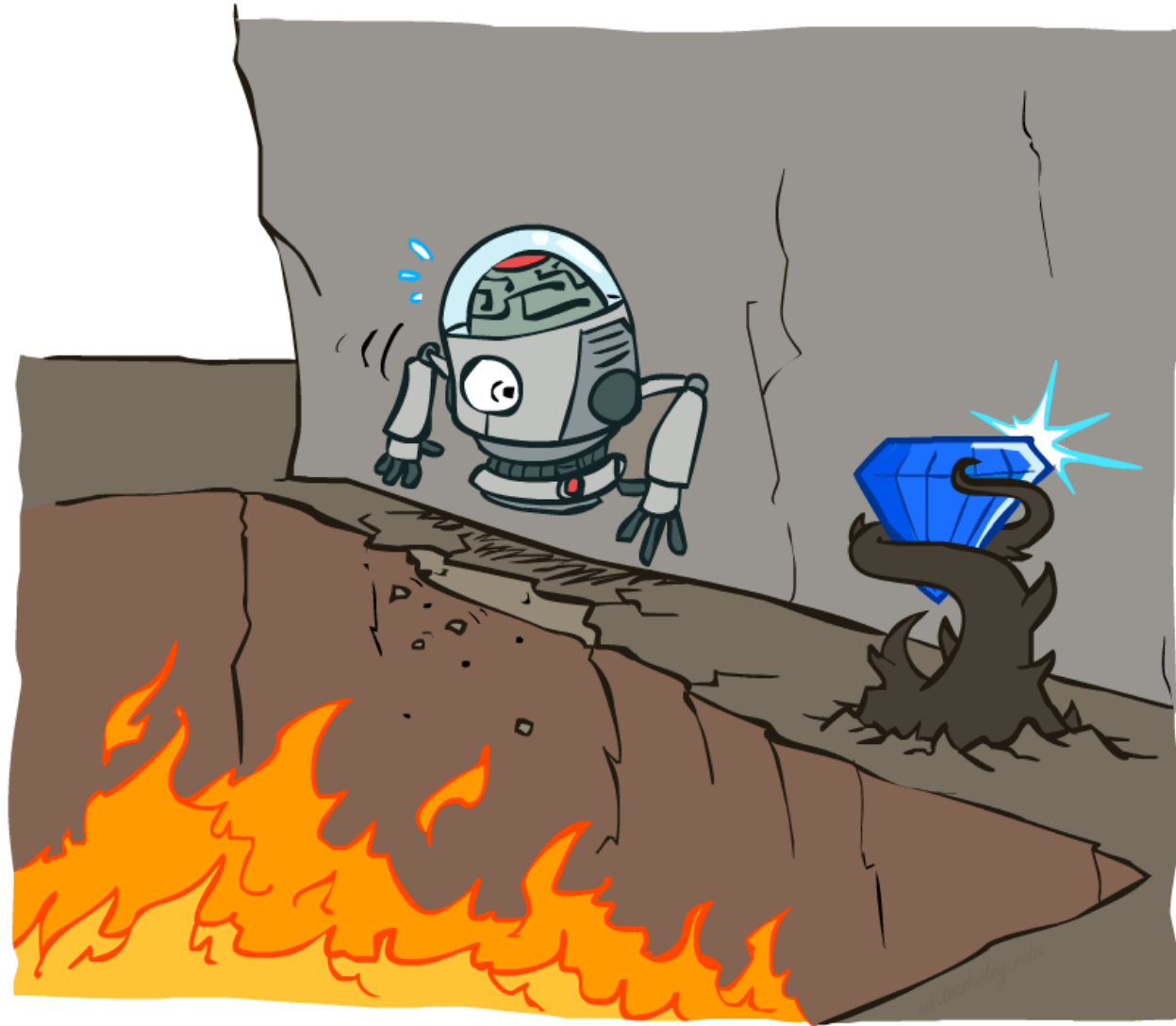
Deep Reinforcement Learning

- 2013 Atari (DQN)
[Deepmind]
- 2015 AlphaGo
[Deepmind]
- 2016 3D locomotion (TRPO+GAE)
[Berkeley]
- 2016 Real Robot Manipulation (GPS)
[Berkeley]
- 2019 **Rubik's Cube (PPO+DR)**
[OpenAI]



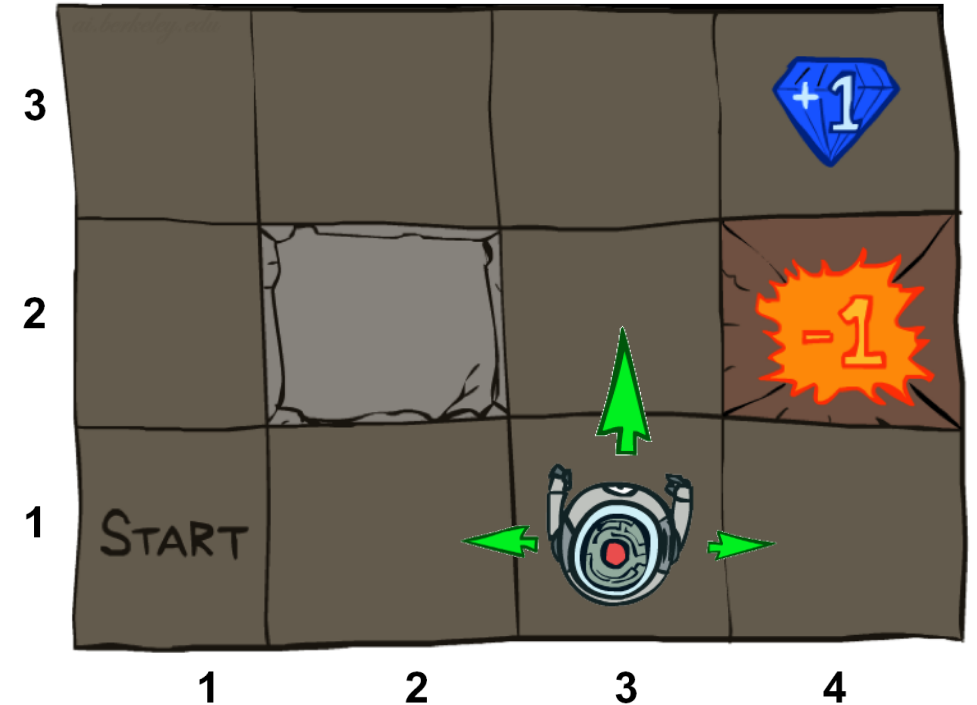
OpenAI

Non-Deterministic Search



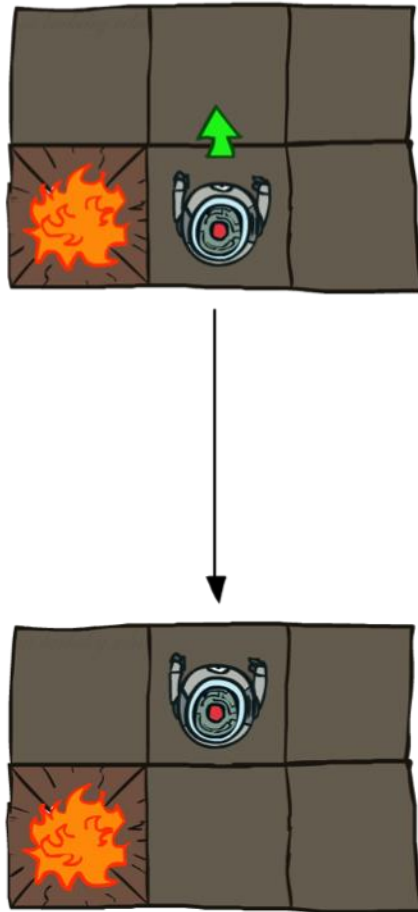
Example: Grid World

- 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

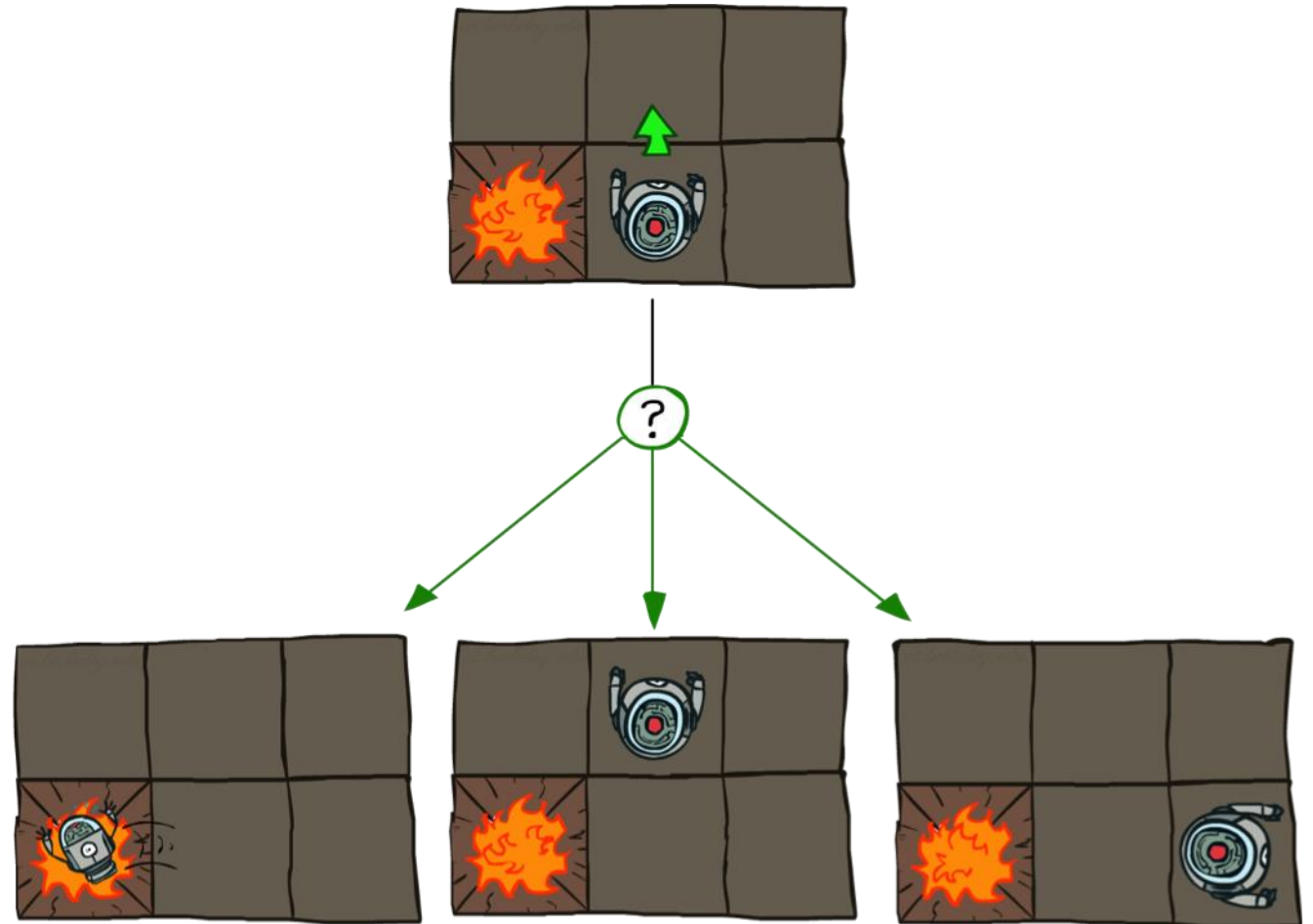


Grid World Actions

Deterministic Grid World

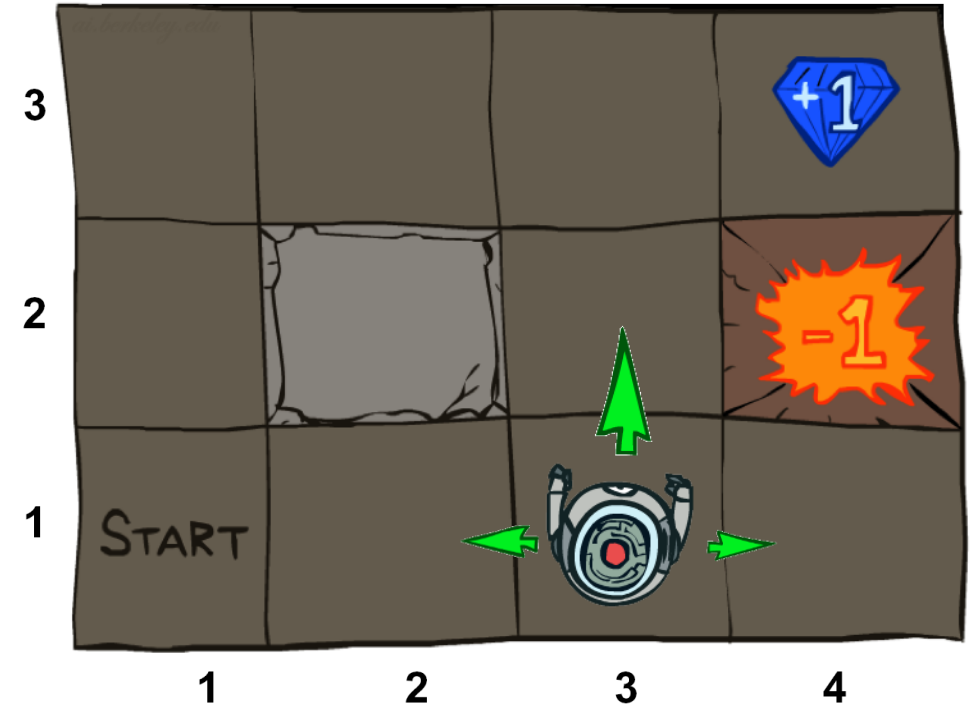


Stochastic Grid World

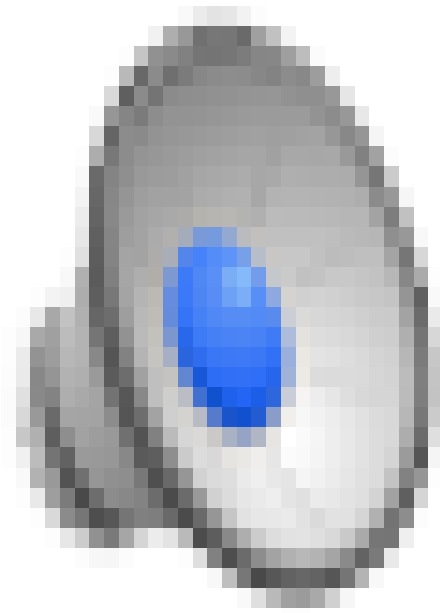


Markov Decision Processes

- 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**



Video of Demo Gridworld Manual Intro



What is Markov about MDPs?

- “Markov” generally means that given the present state, the future and the past are independent
- For Markov decision processes, “Markov” means action outcomes depend only on the current state

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \dots, S_0 = s_0)$$

=

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

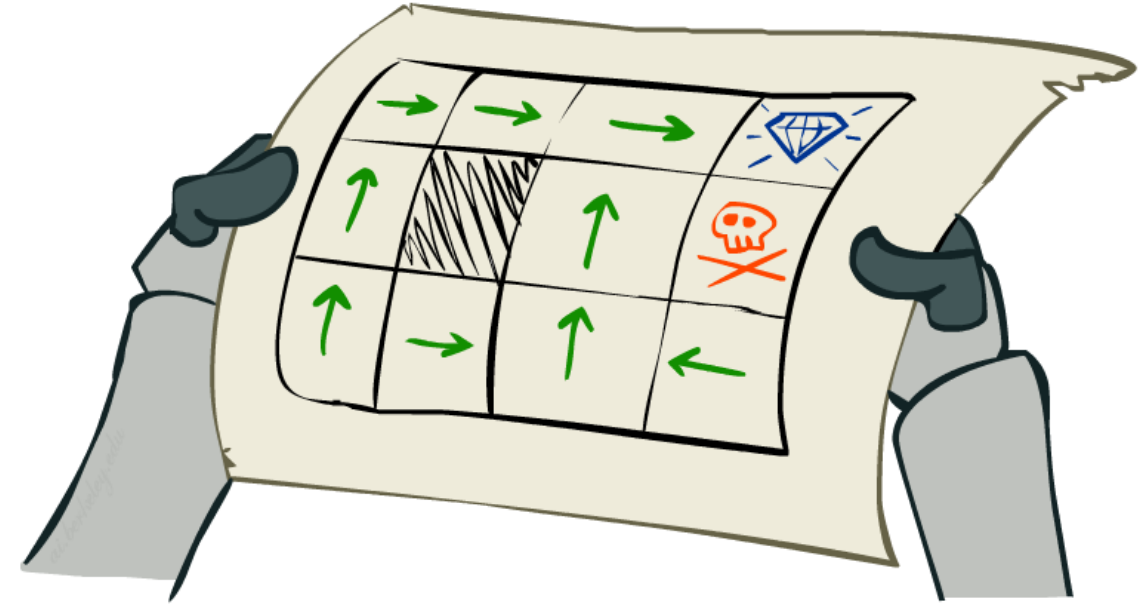
- This is just like search, where the successor function could only depend on the current state (not the history)



Andrey Markov
(1856-1922)

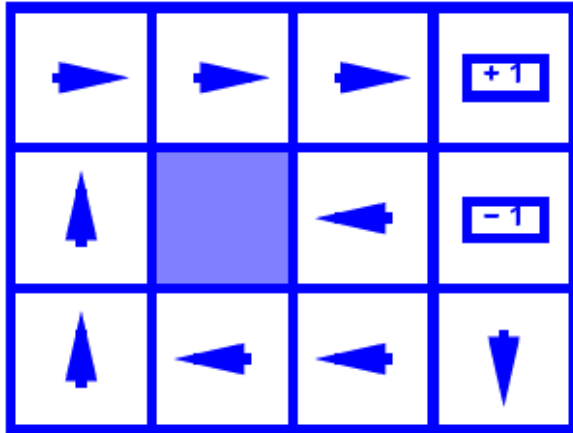
Policies

- 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

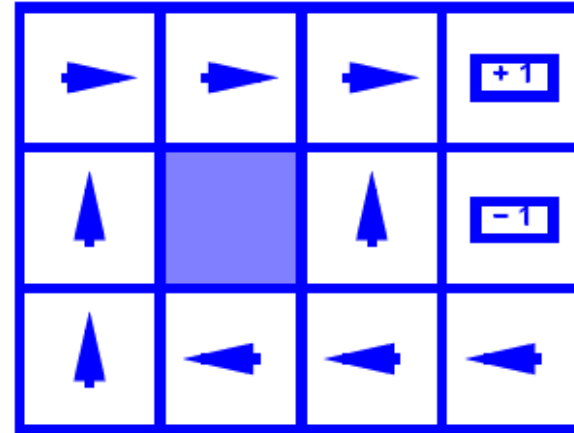


Optimal policy when $R(s, a, s') = -0.03$
for all non-terminals s

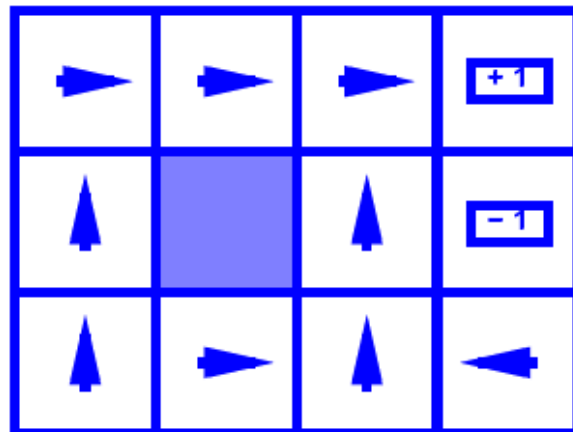
Optimal Policies



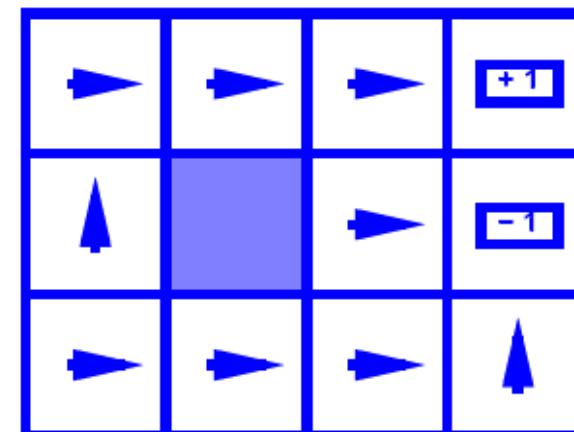
$R(s) = -0.01$



$R(s) = -0.03$

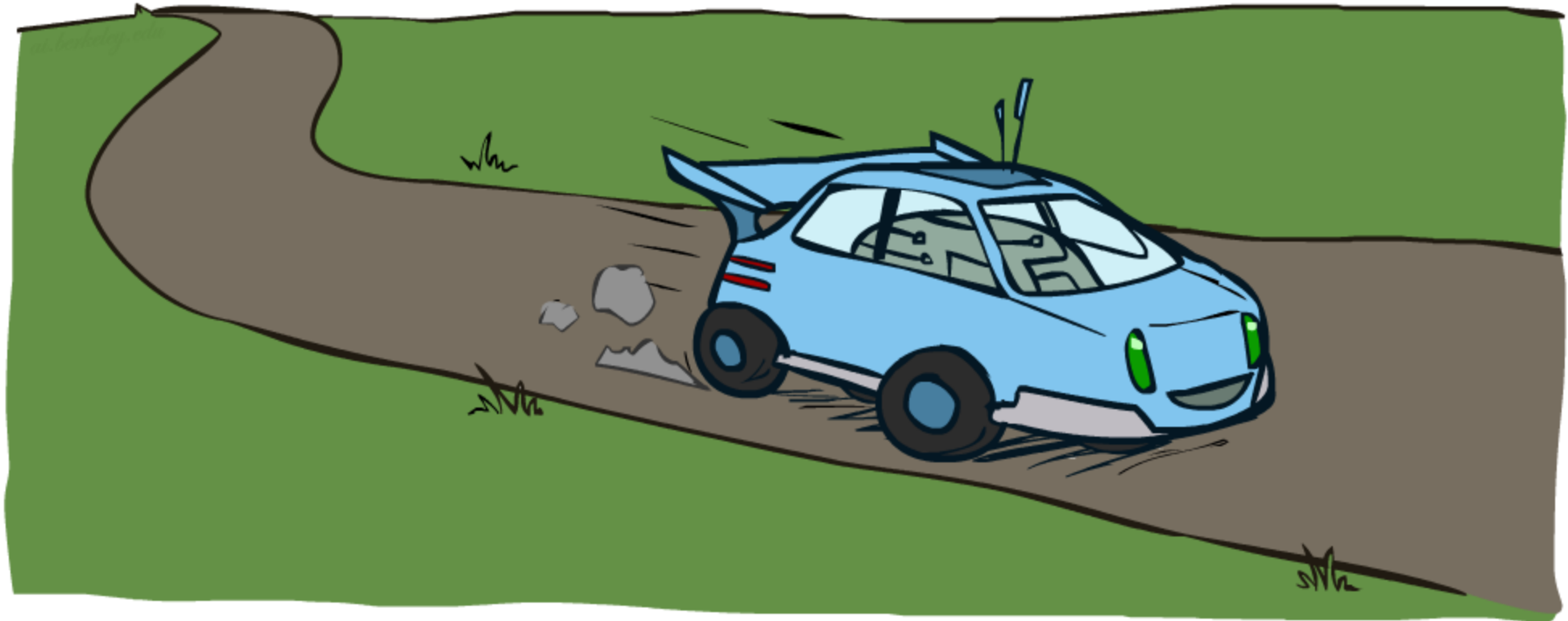


$R(s) = -0.4$



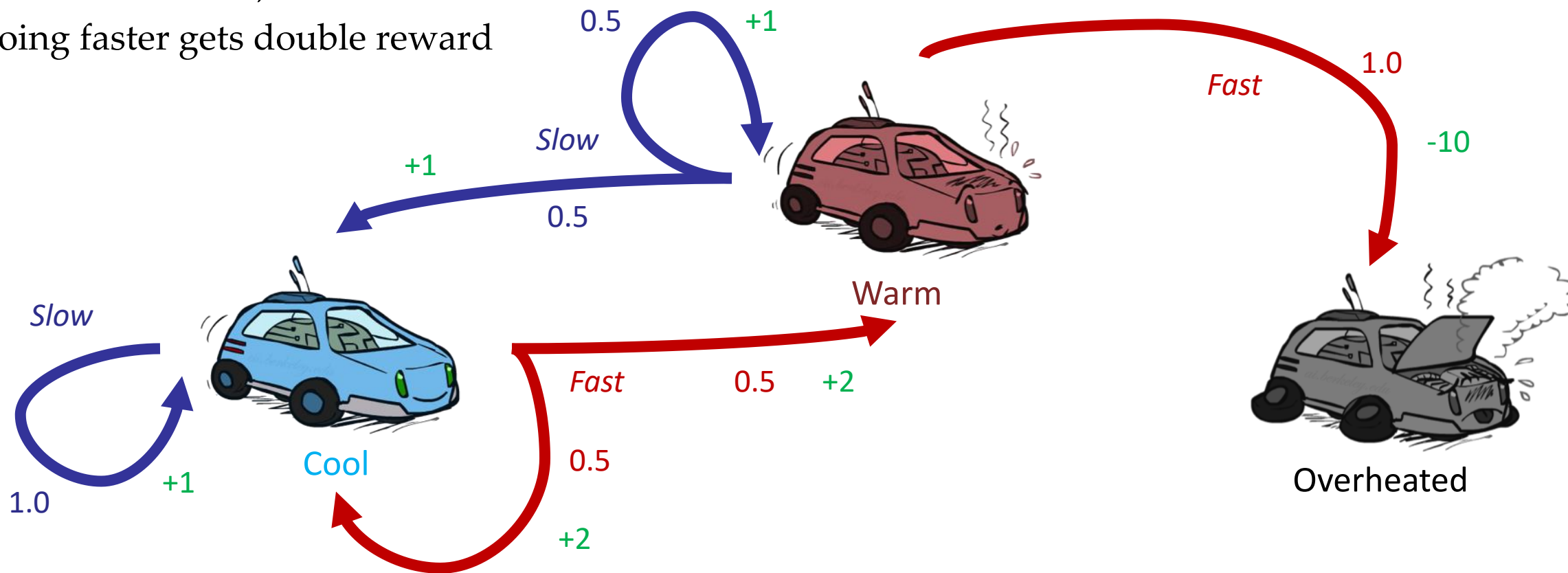
$R(s) = -2.0$

Example: Racing

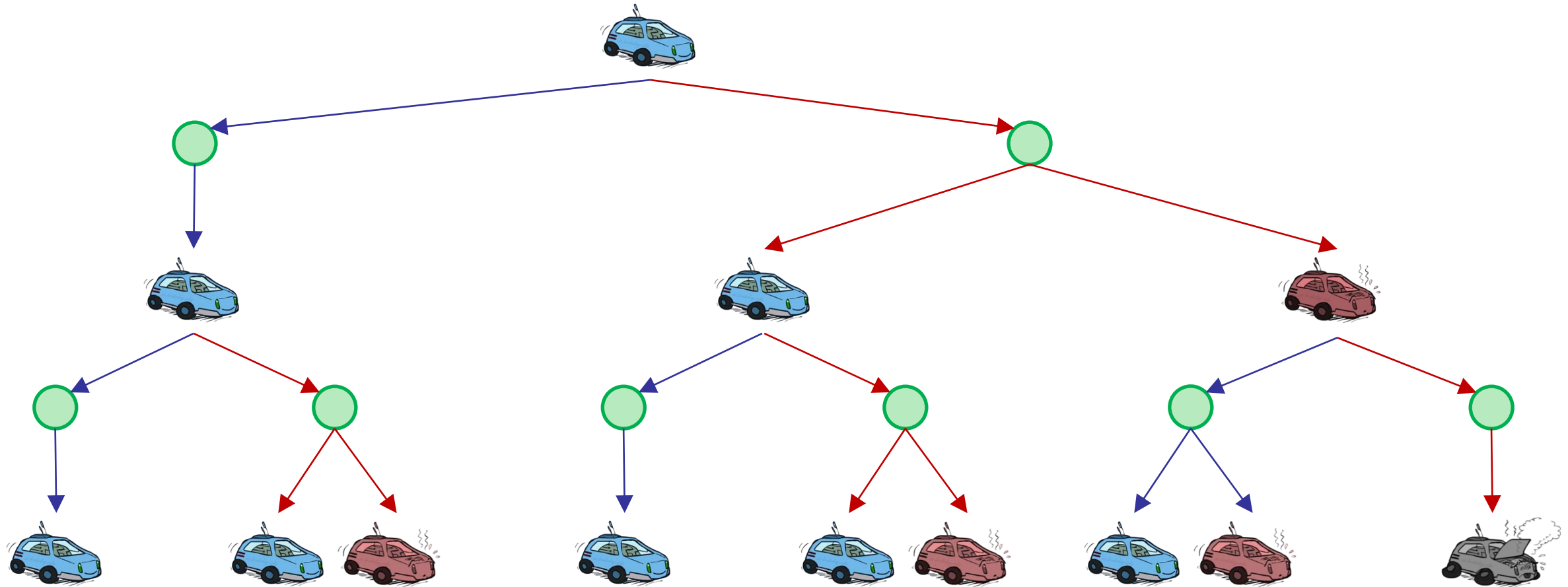


Example: Racing

- A robot car wants to travel far, quickly
- Three states: **Cool**, **Warm**, Overheated
- Two actions: *Slow*, *Fast*
- Going faster gets double reward

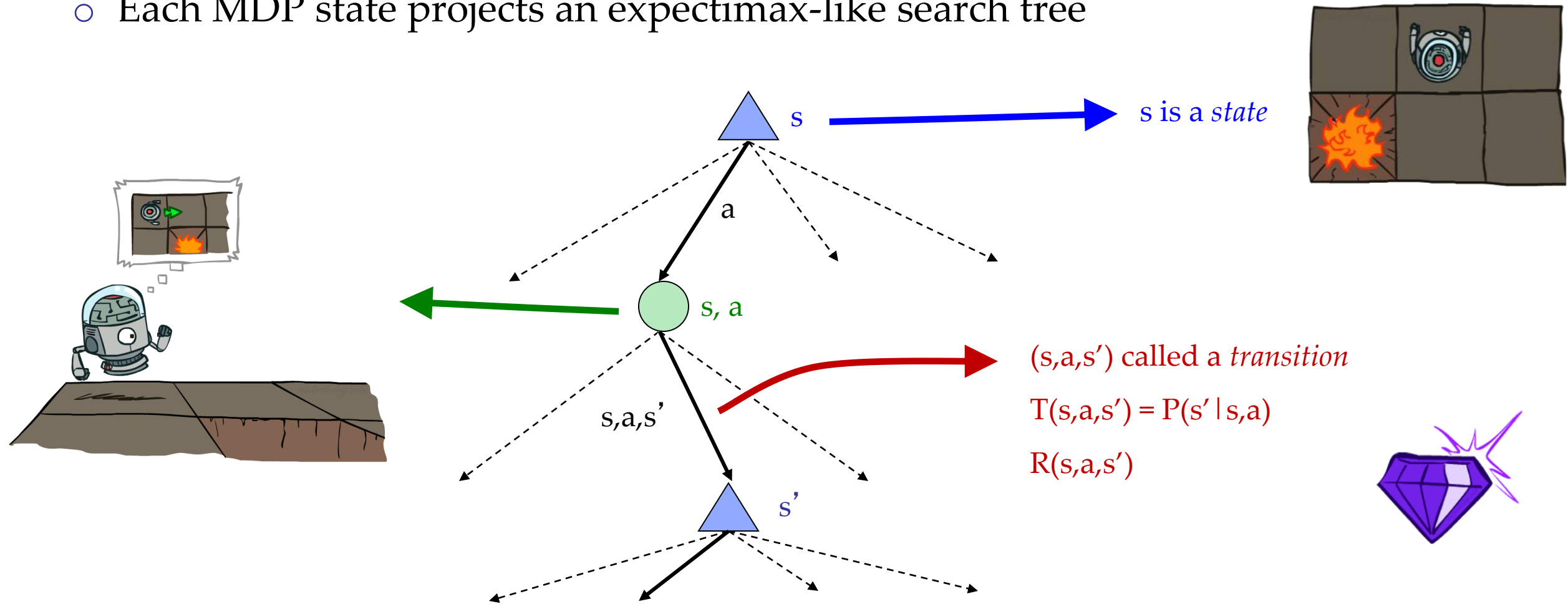


Racing Search Tree

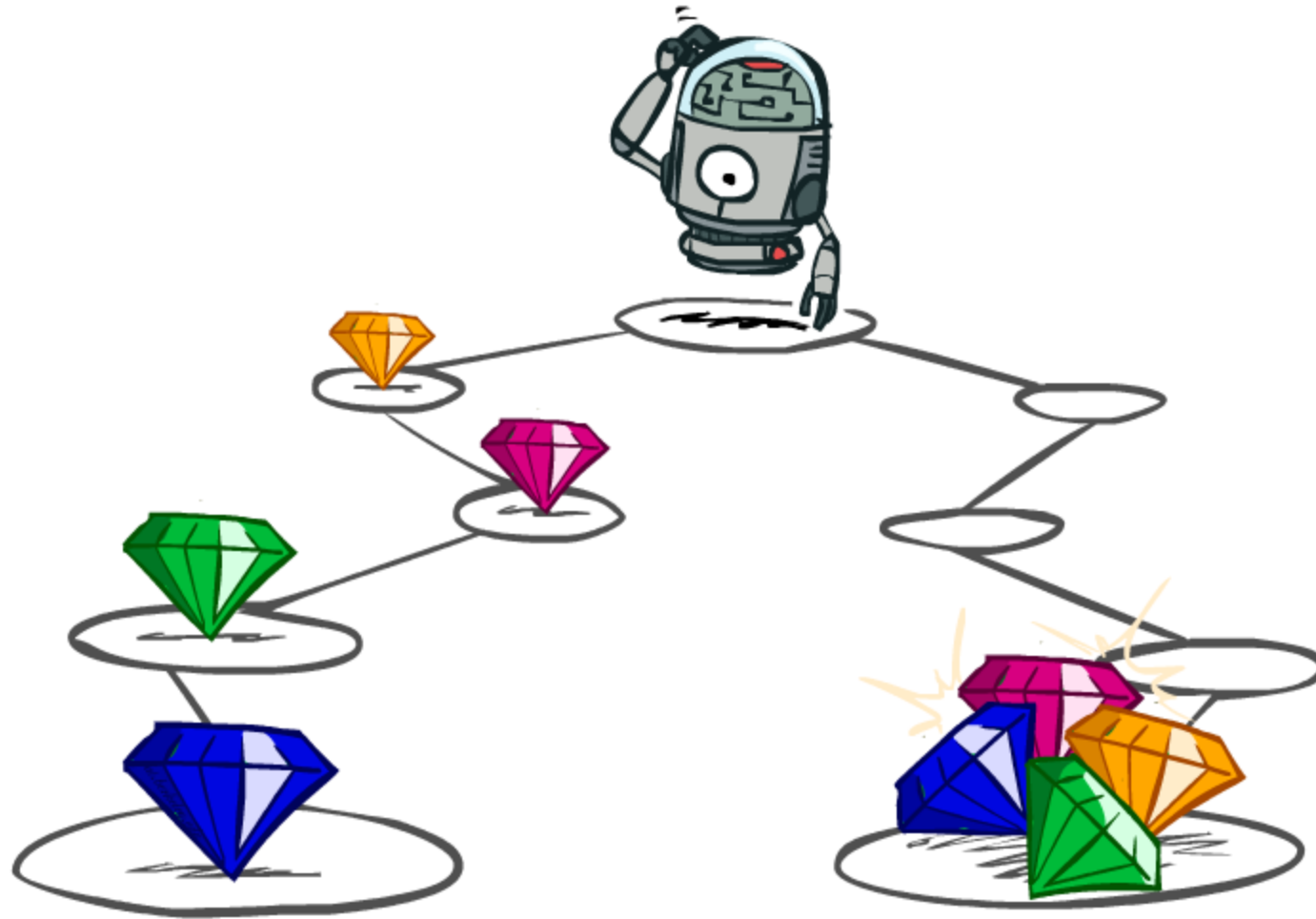


MDP Search Trees

- Each MDP state projects an expectimax-like search tree

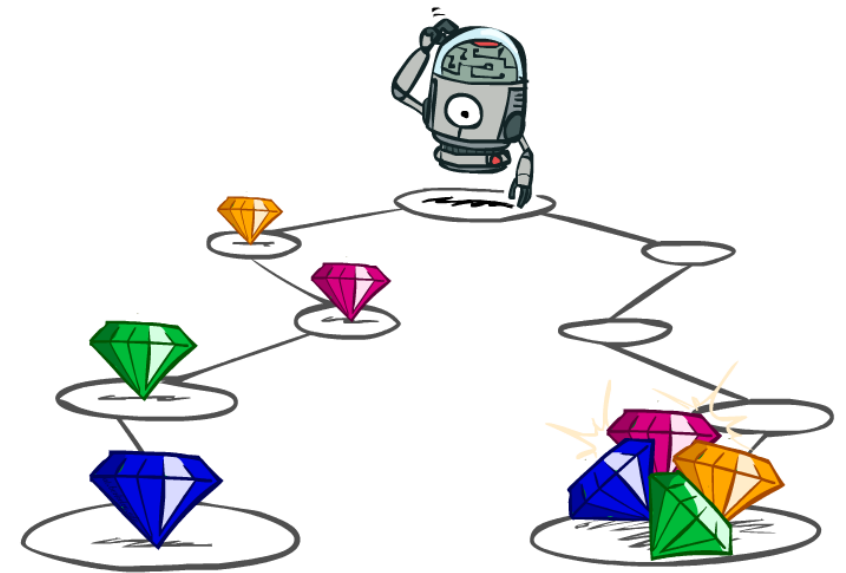


Utilities of Sequences



Utilities of Sequences

- What preferences should an agent have over reward sequences?
- More or less? $[1, 2, 2]$ or $[2, 3, 4]$
- Now or later? $[0, 0, 1]$ or $[1, 0, 0]$



Discounting

- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially



1

Worth Now



γ

Worth Next Step



γ^2

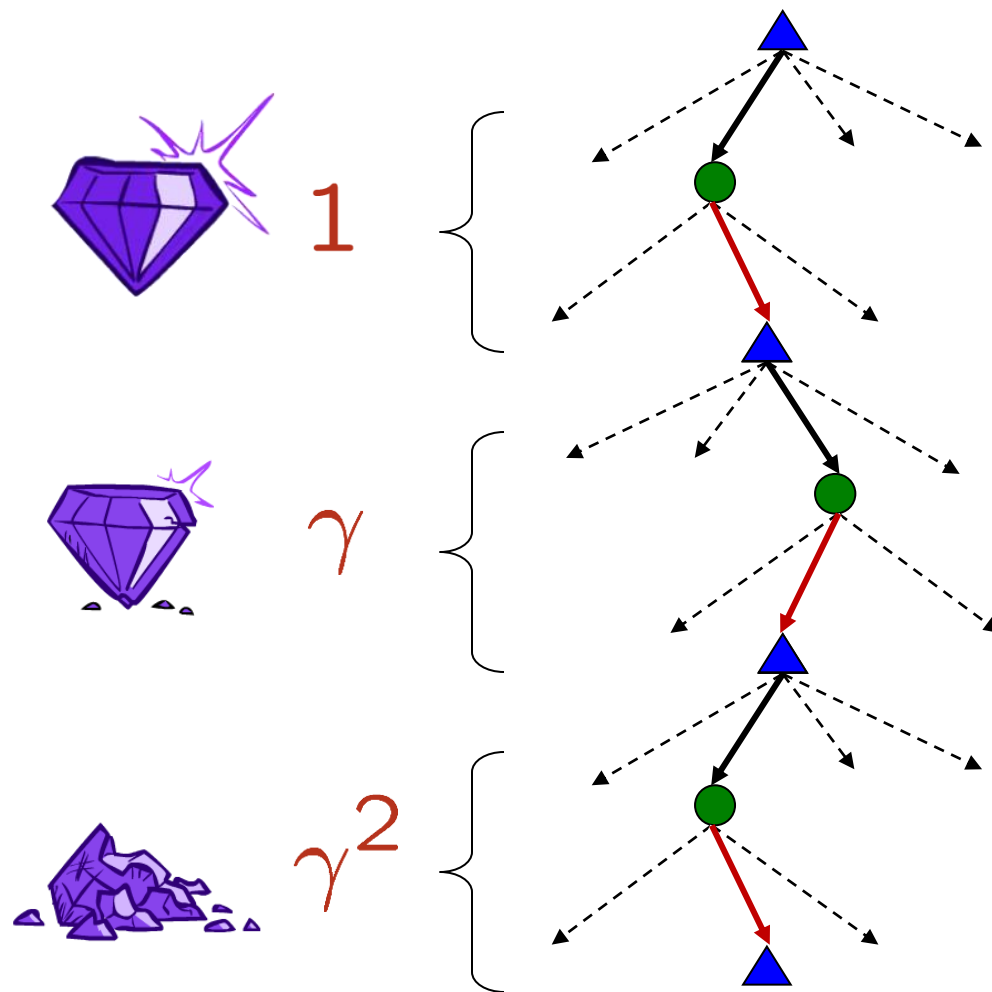
Worth In Two Steps

Discounting

- How to discount?
 - Each time we descend a level, we multiply in the discount once

- Why discount?
 - Reward now is better than later
 - Can also think of it as a $1-\gamma$ chance of ending the process at every step
 - Also helps our algorithms converge

- Example: discount of 0.5
 - $U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3$
 - $U([1,2,3]) < U([3,2,1])$



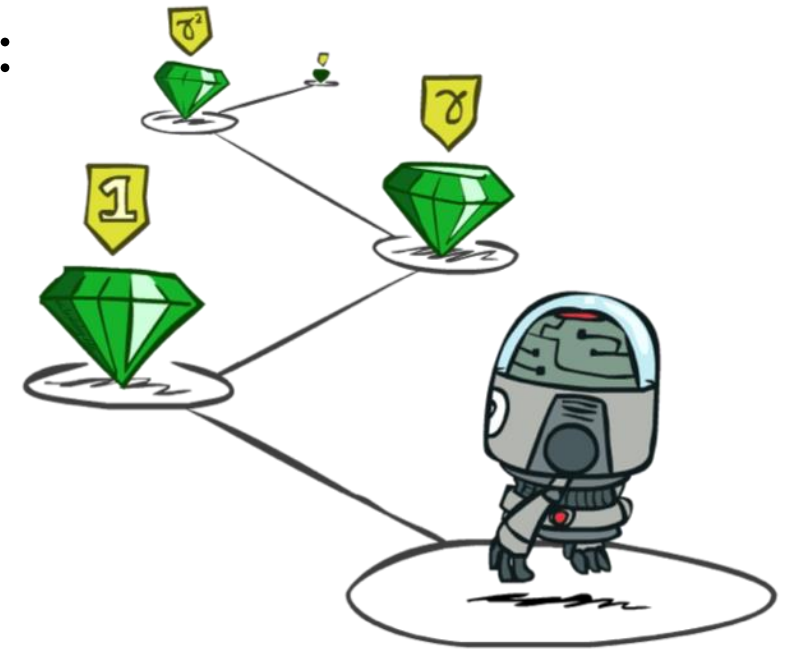
Stationary Preferences

- Theorem: if we assume **stationary preferences**:

$$[a_1, a_2, \dots] \succ [b_1, b_2, \dots]$$



$$[r, a_1, a_2, \dots] \succ [r, b_1, b_2, \dots]$$



- Then: there is only ways to define utilities

- Additive discounted utility:

$$U([r_0, r_1, r_2, \dots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \dots$$

Quiz: Discounting

○ Given:

10				1
a	b	c	d	e

- Actions: East, West, and Exit (only available in exit states a, e)
- Transitions: deterministic

○ Quiz 1: For $\gamma = 1$, what is the optimal policy?

10	<-	<-	<-	1
----	----	----	----	---

○ Quiz 2: For $\gamma = 0.1$, what is the optimal policy?

10	<-	<-	->	1
----	----	----	----	---

○ Quiz 3: For which γ are West and East equally good when in state d?

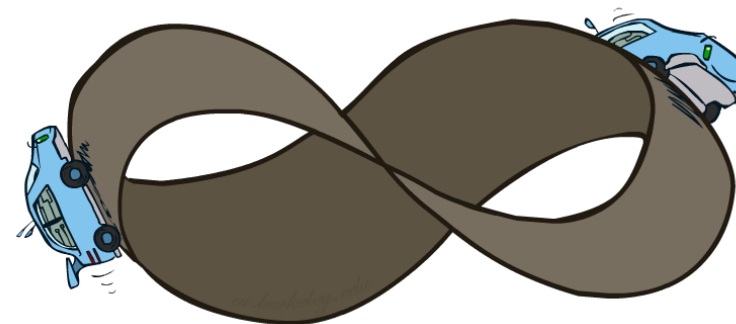
$$10\gamma = \gamma^3$$

Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?

- Solutions:

- Finite horizon: (similar to depth-limited search)
 - Terminate episodes after a fixed T steps (e.g. life)
 - Gives nonstationary policies (π depends on time left)



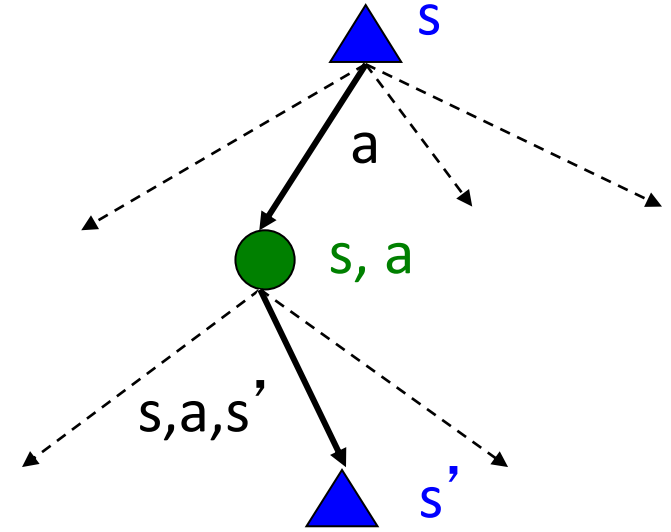
- Discounting: use $0 < \gamma < 1$

$$U([r_0, \dots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max} / (1 - \gamma)$$

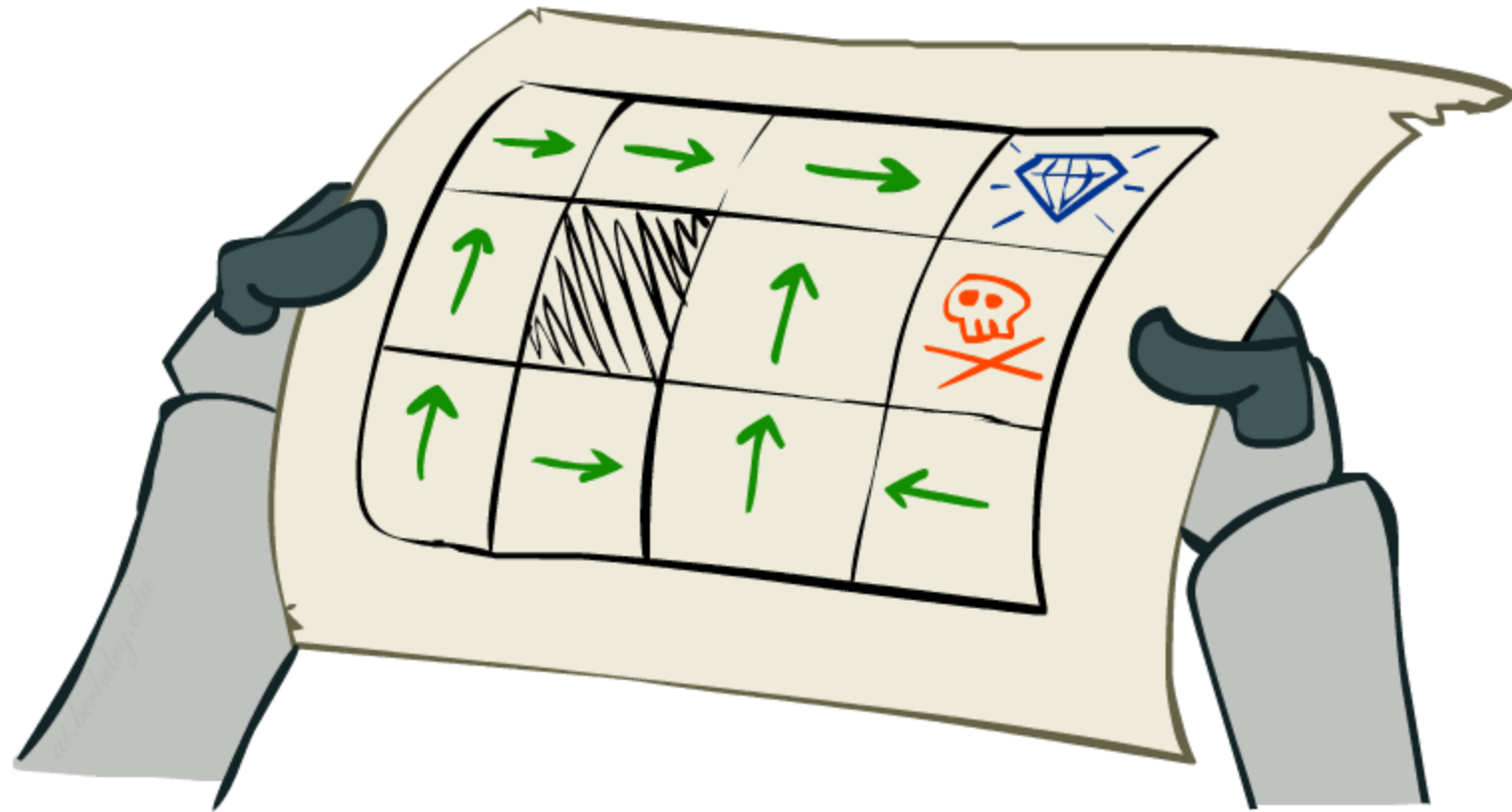
- Smaller γ means smaller “horizon” – shorter term focus
- Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like “overheated” for racing)

Recap: Defining MDPs

- Markov decision processes:
 - Set of states S
 - Start state s_0
 - Set of actions A
 - Transitions $P(s' | s, a)$ (or $T(s, a, s')$)
 - Rewards $R(s, a, s')$ (and discount γ)
- MDP quantities so far:
 - Policy = Choice of action for each state
 - Utility = sum of (discounted) rewards

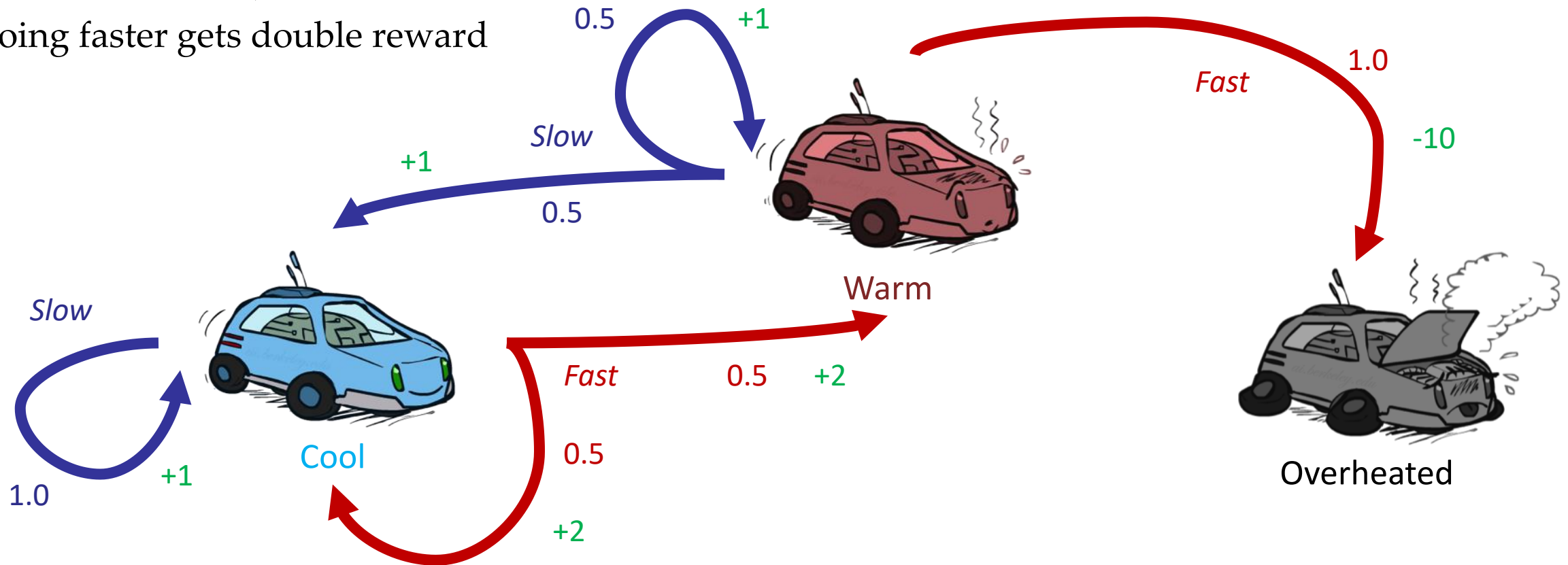


Solving MDPs

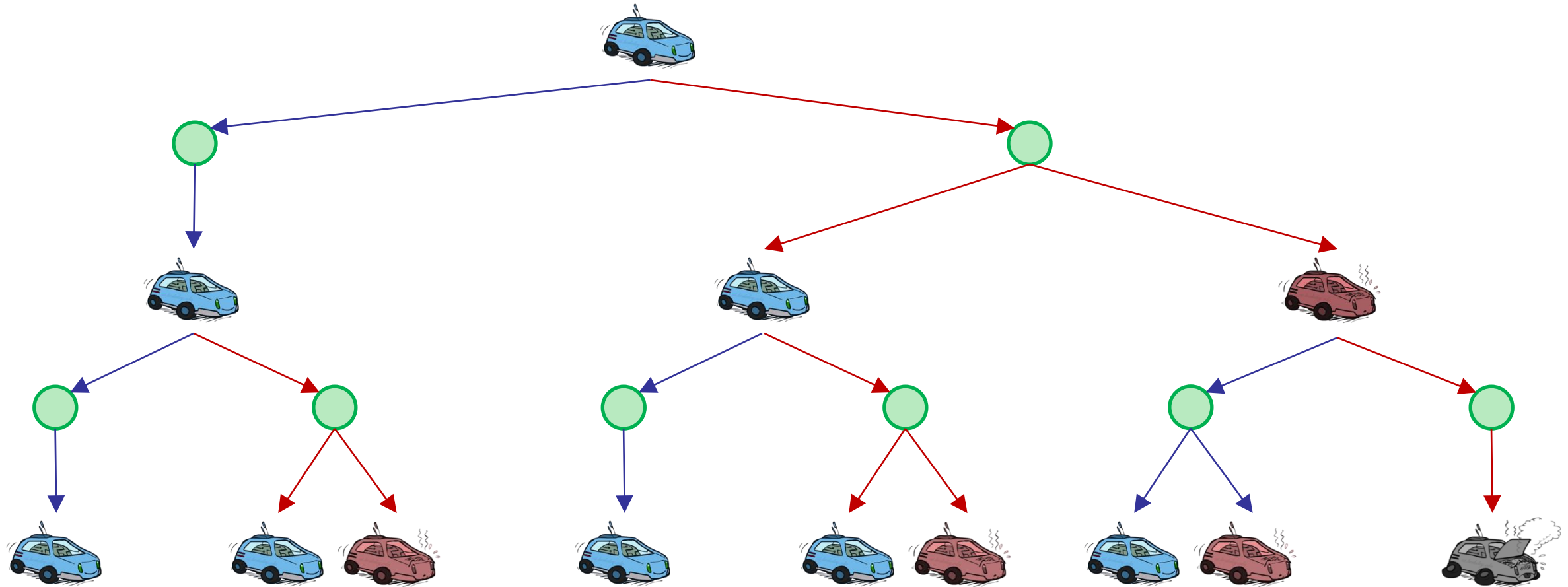


Recall: Racing MDP

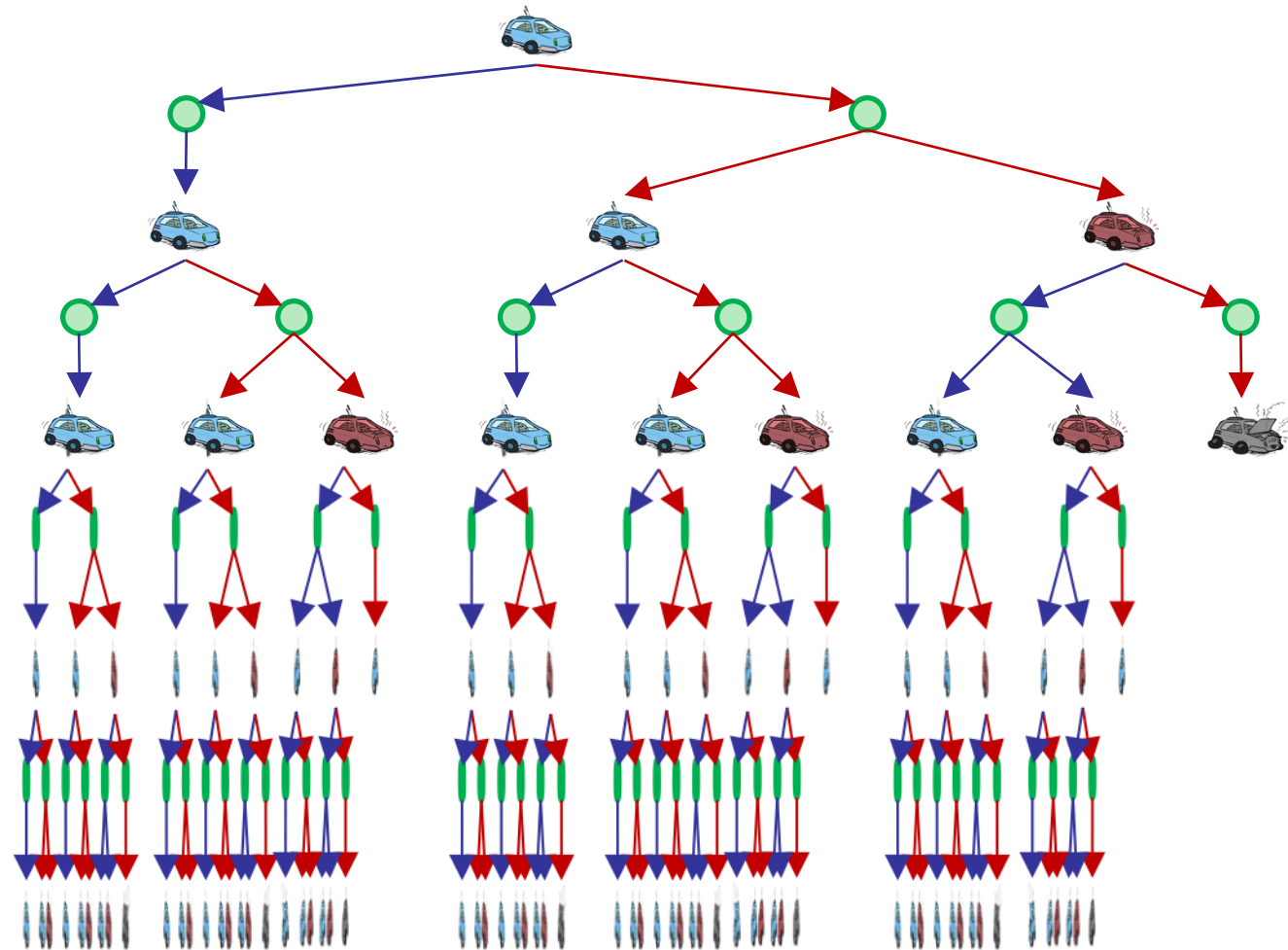
- A robot car wants to travel far, quickly
- Three states: **Cool**, **Warm**, Overheated
- Two actions: *Slow*, *Fast*
- Going faster gets double reward



Racing Search Tree

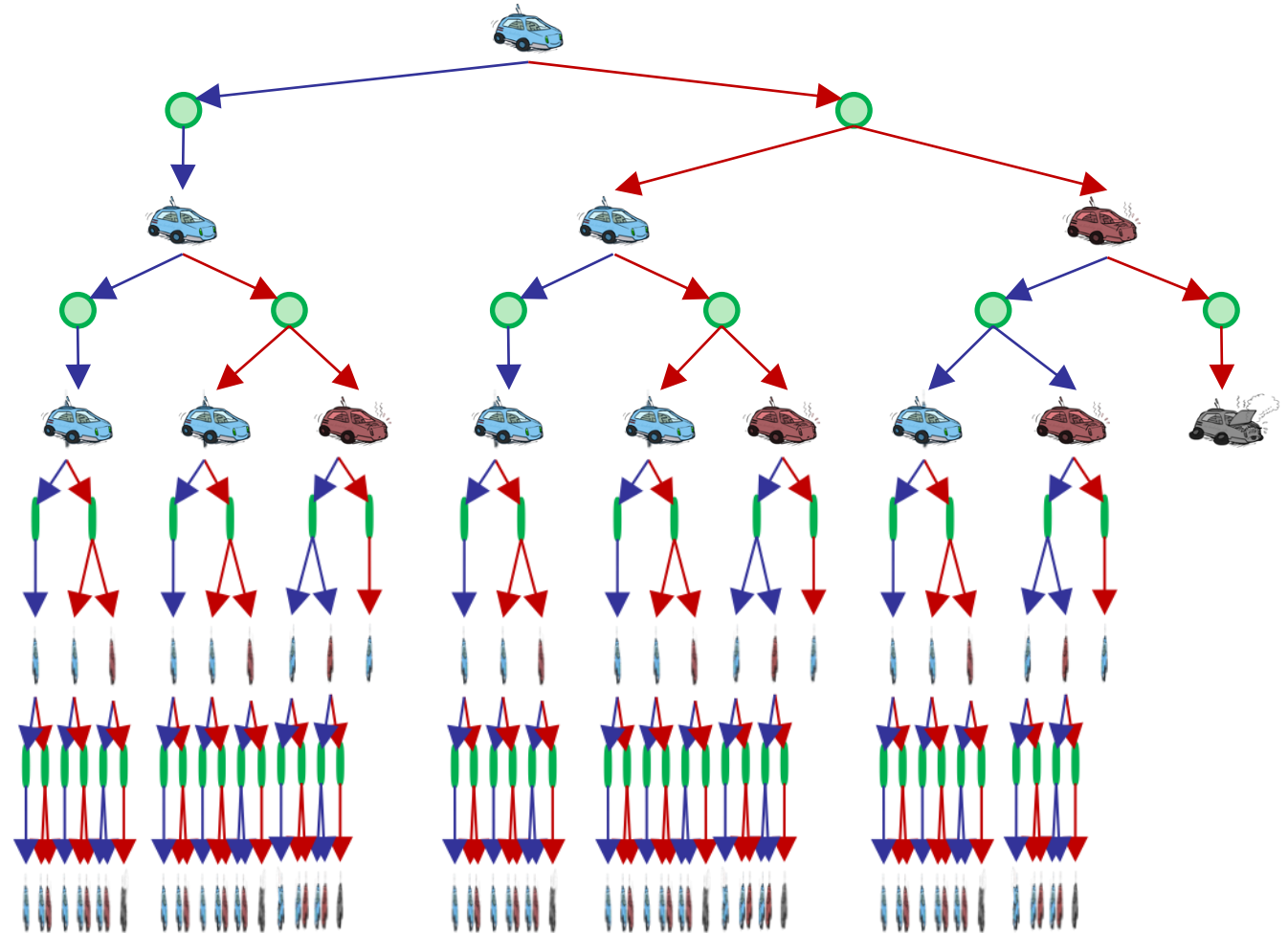


Racing Search Tree



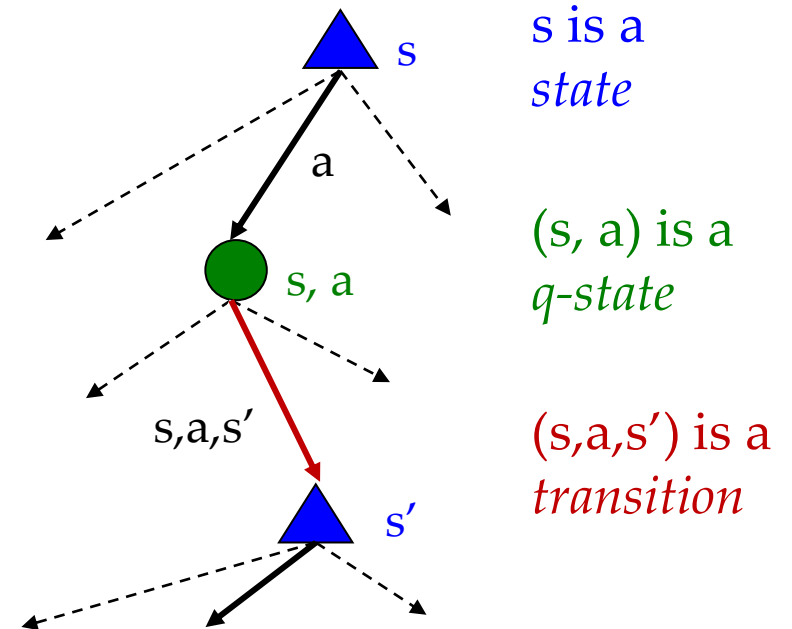
Racing Search Tree

- We're doing way too much work with expectimax!
- Problem: States are repeated
 - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
 - Idea: Do a depth-limited computation, but with increasing depths until change is small
 - Note: deep parts of the tree eventually don't matter if $\gamma < 1$



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

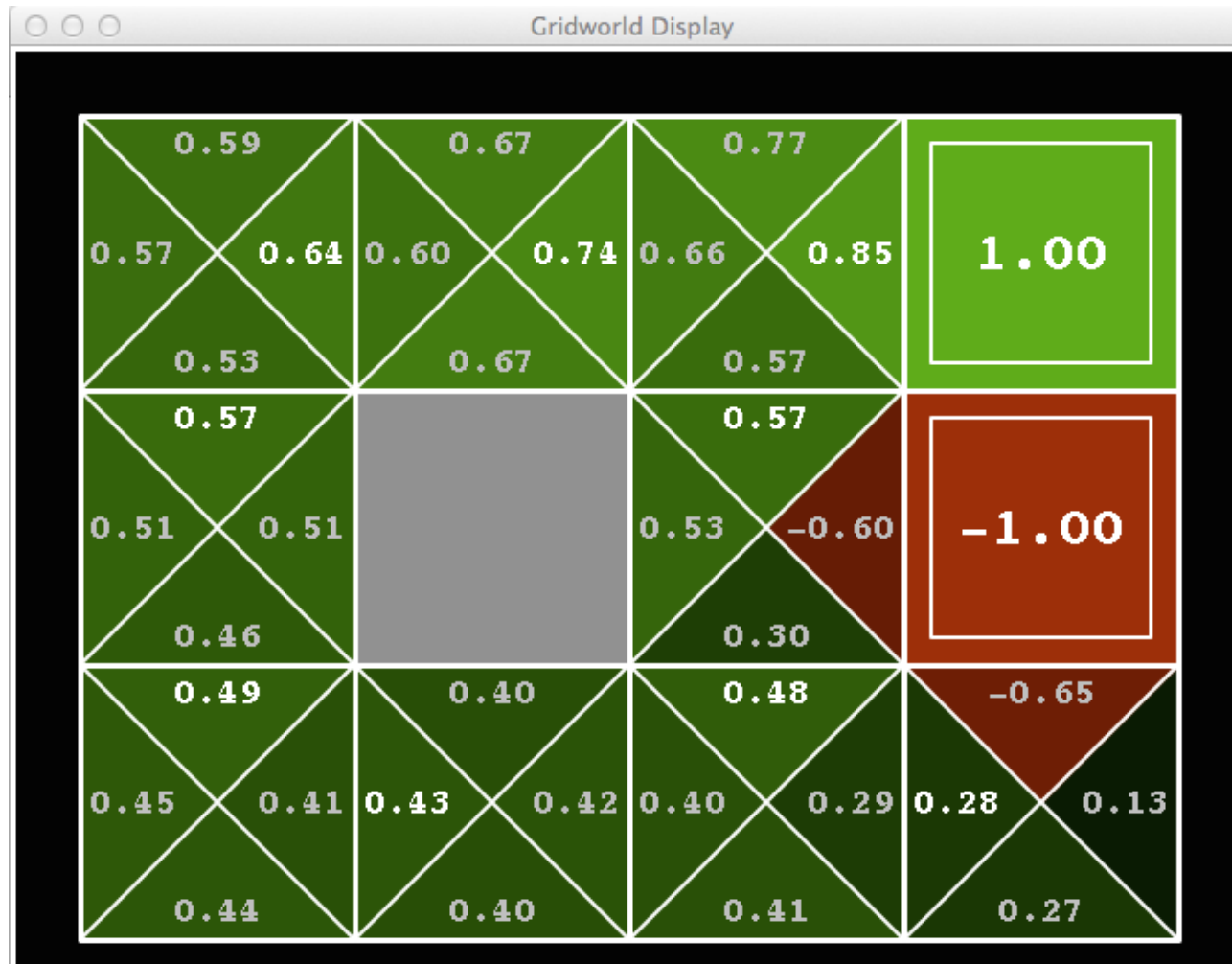


Gridworld V^* Values



Noise = 0.2
Discount = 0.9
Living reward = 0

Gridworld Q^* Values



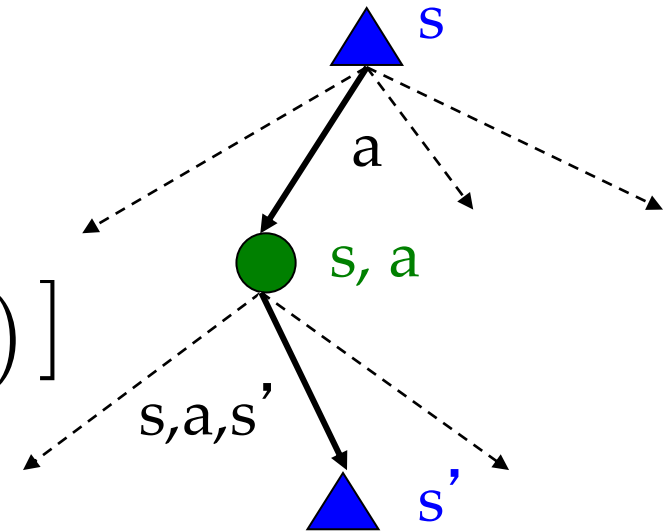
Noise = 0.2
Discount = 0.9
Living reward = 0

Values of States: Bellman Equation

- Recursive definition of value:

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

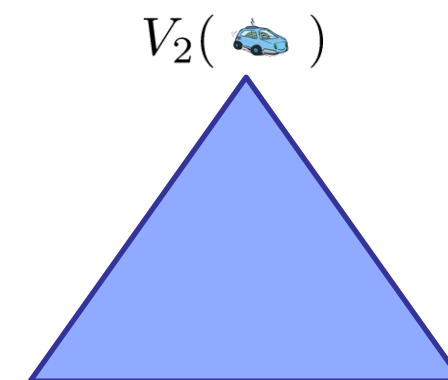
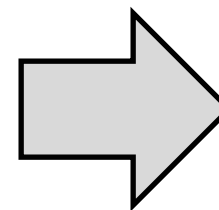
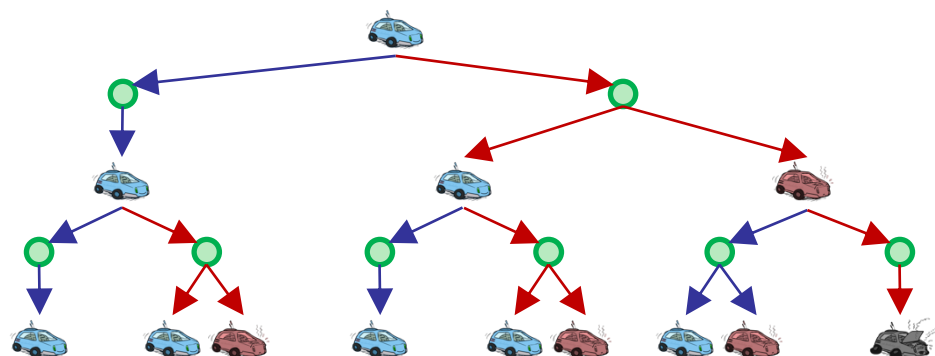
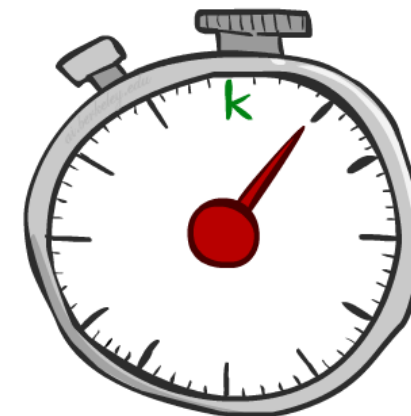
$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$



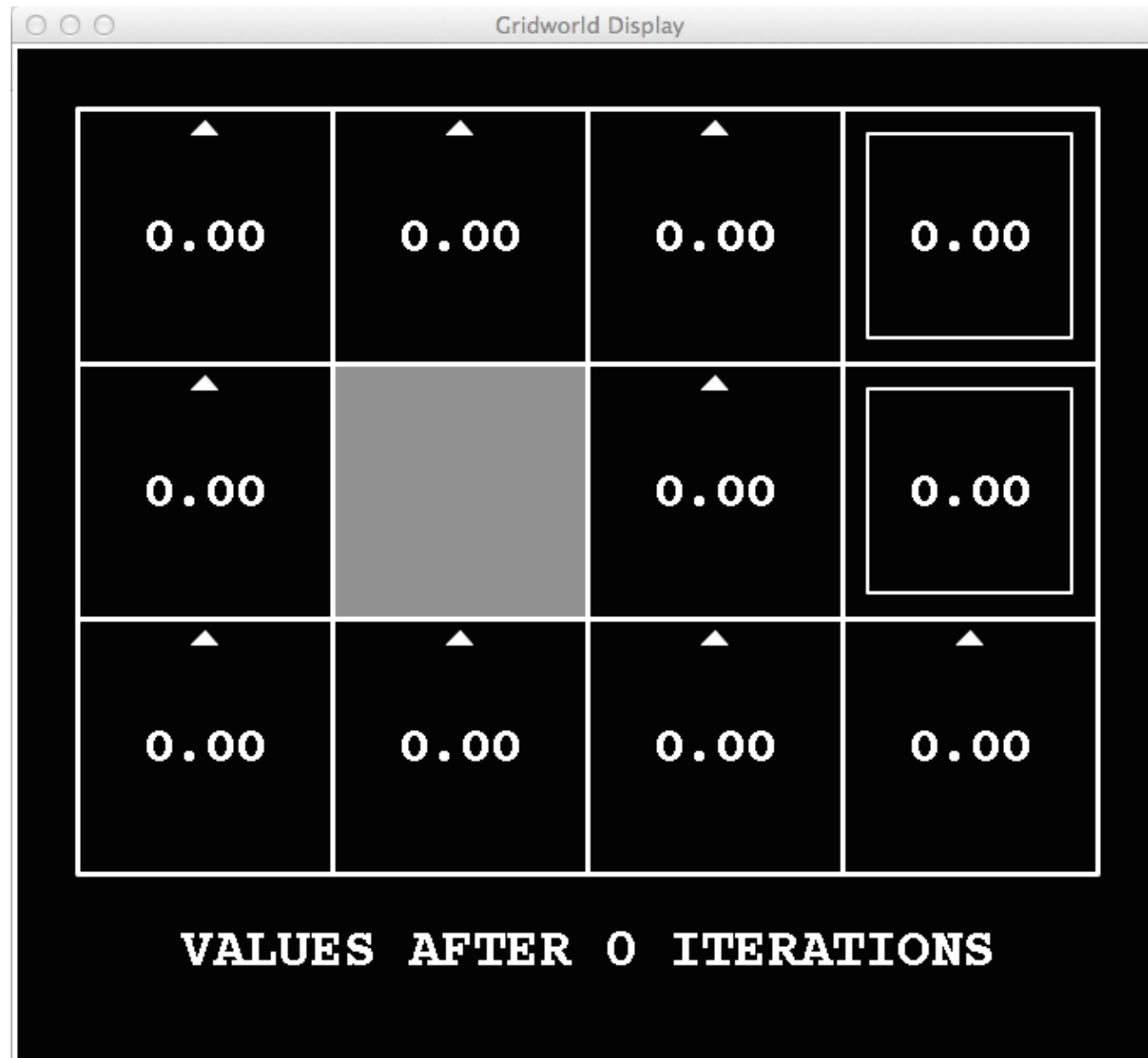
$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

Time-Limited Values

- Key idea: time-limited values
- Define $V_k(s)$ to be the optimal value of s if the game ends in k more time steps
 - Equivalently, it's what a depth- k expectimax would give from s



$k=0$



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



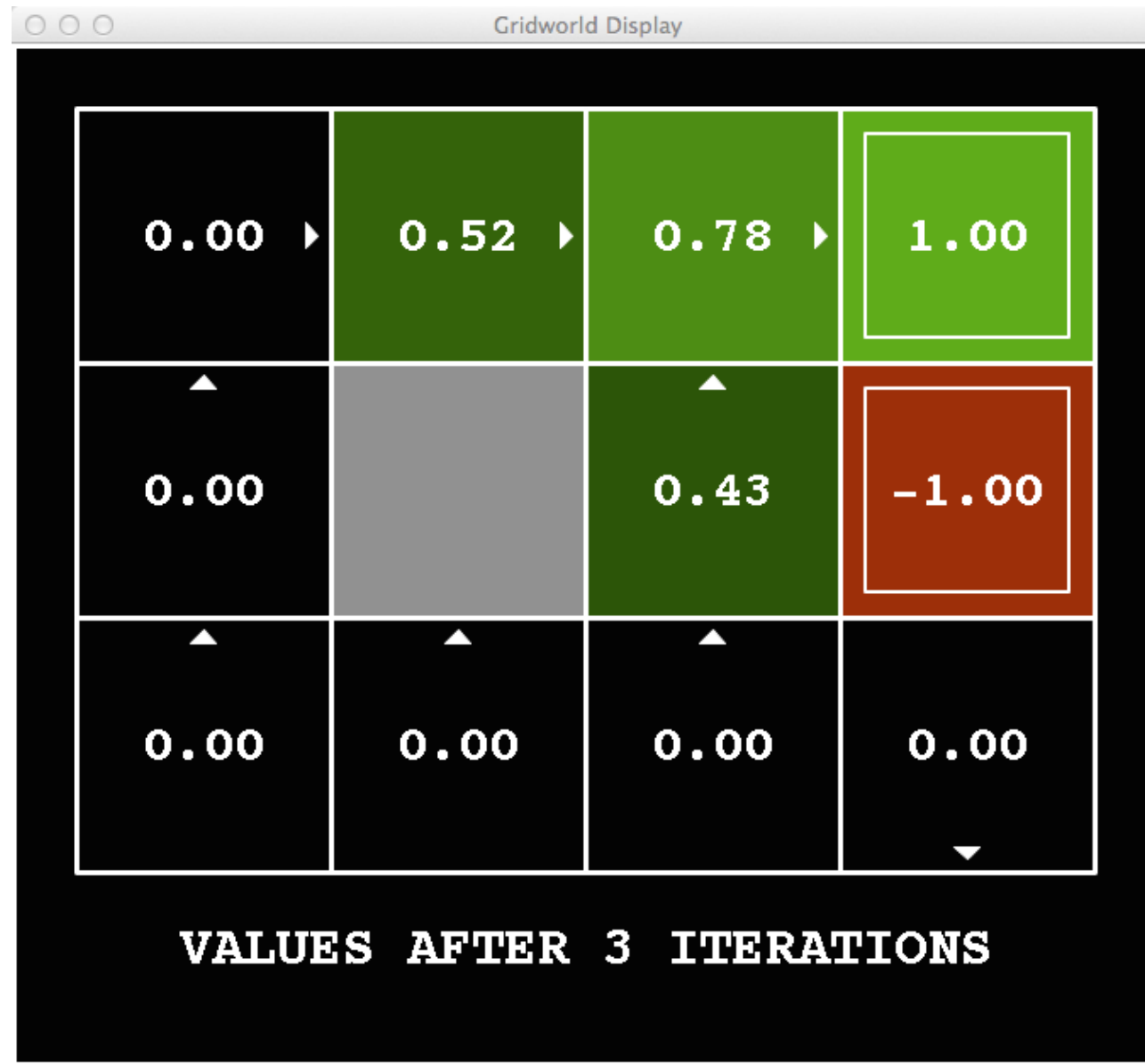
Noise = 0.2
Discount = 0.9
Living reward = 0

k=2



Noise = 0.2
Discount = 0.9
Living reward = 0

k=3



Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



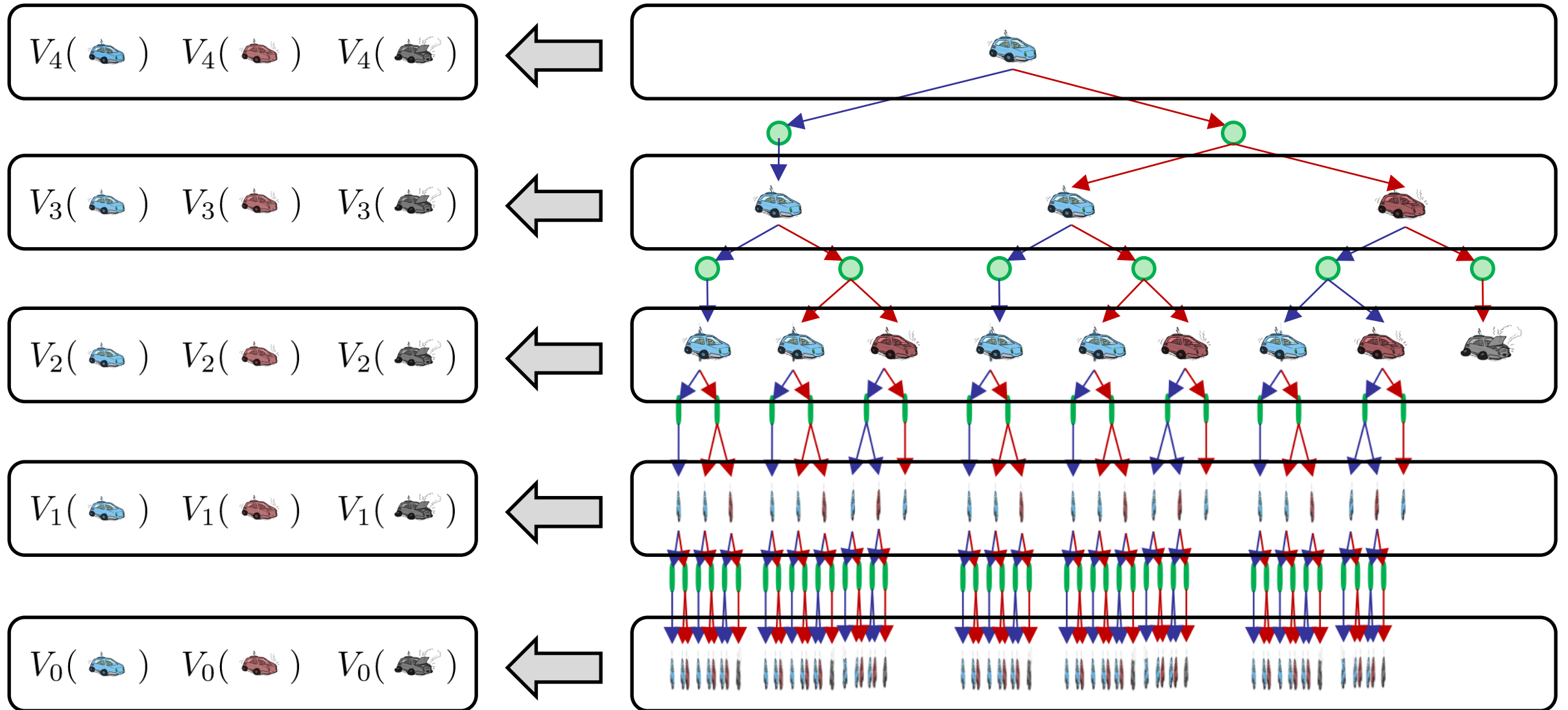
Noise = 0.2
Discount = 0.9
Living reward = 0

k=100

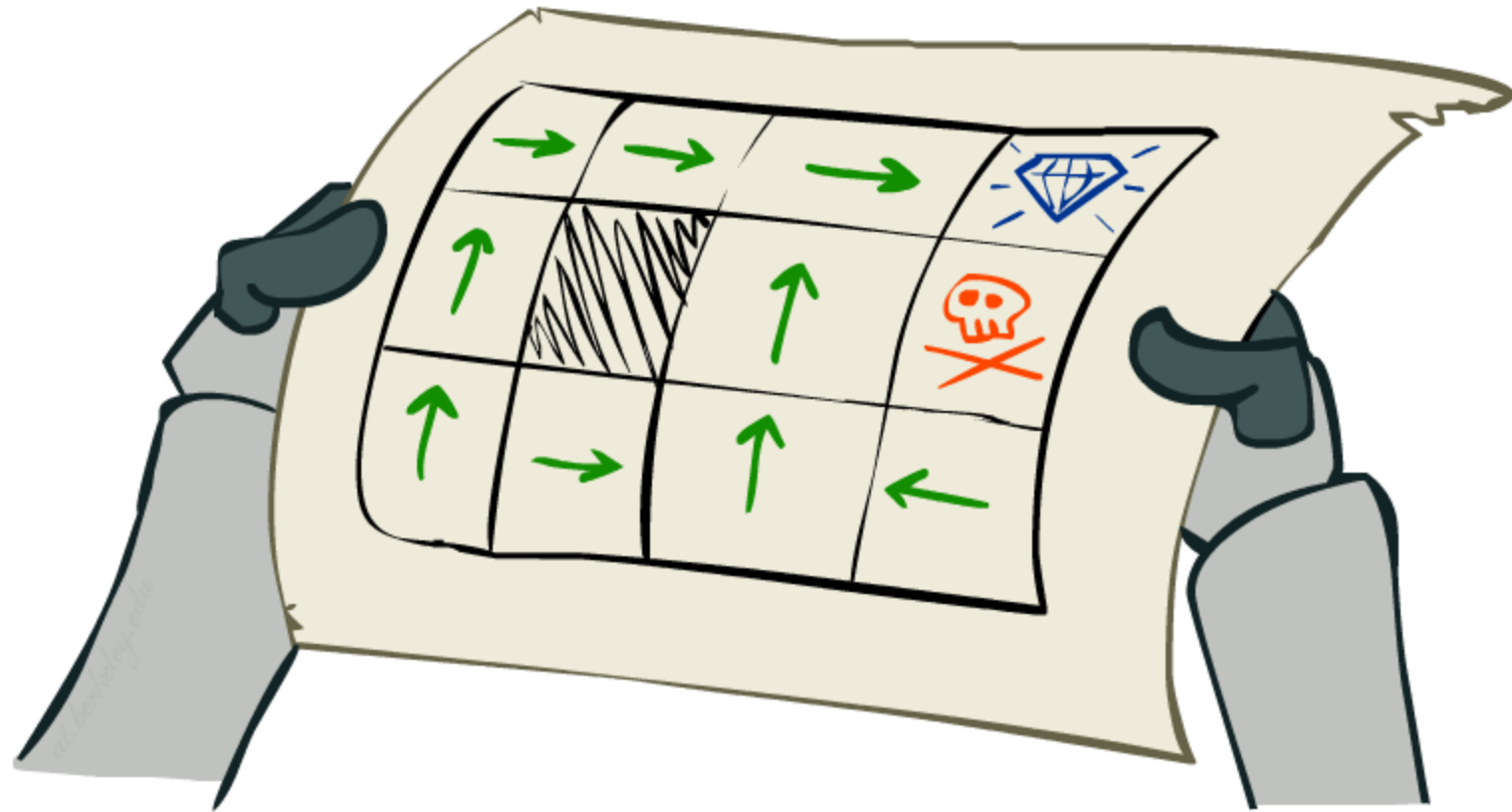


Noise = 0.2
Discount = 0.9
Living reward = 0

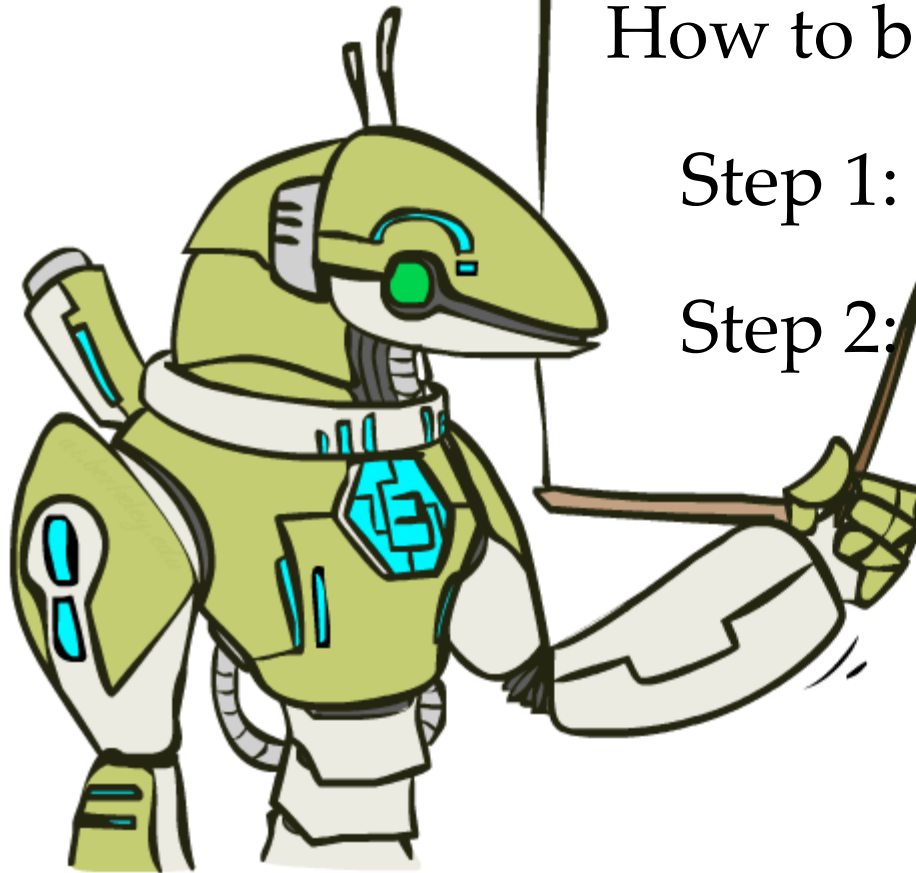
Computing Time-Limited Values



Solving MDPs



The Bellman Equations



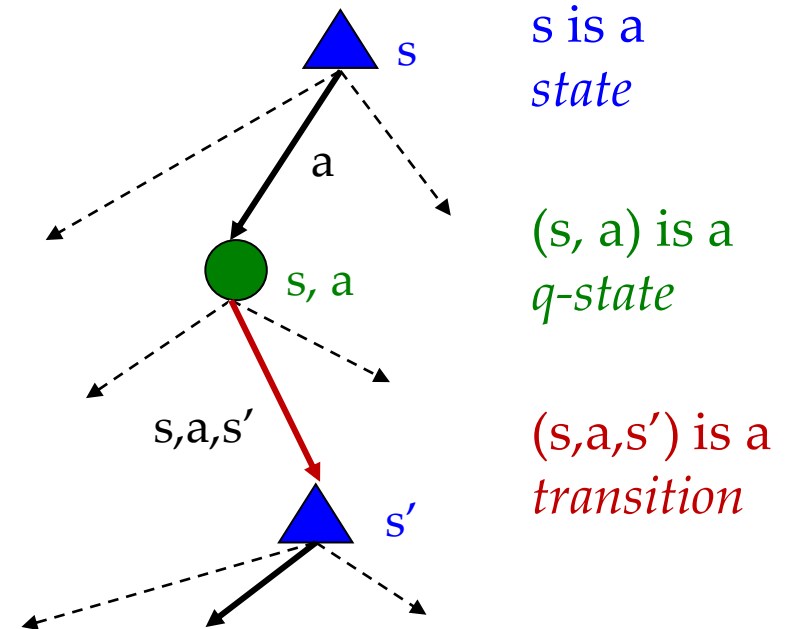
How to be optimal:

Step 1: Take correct first action

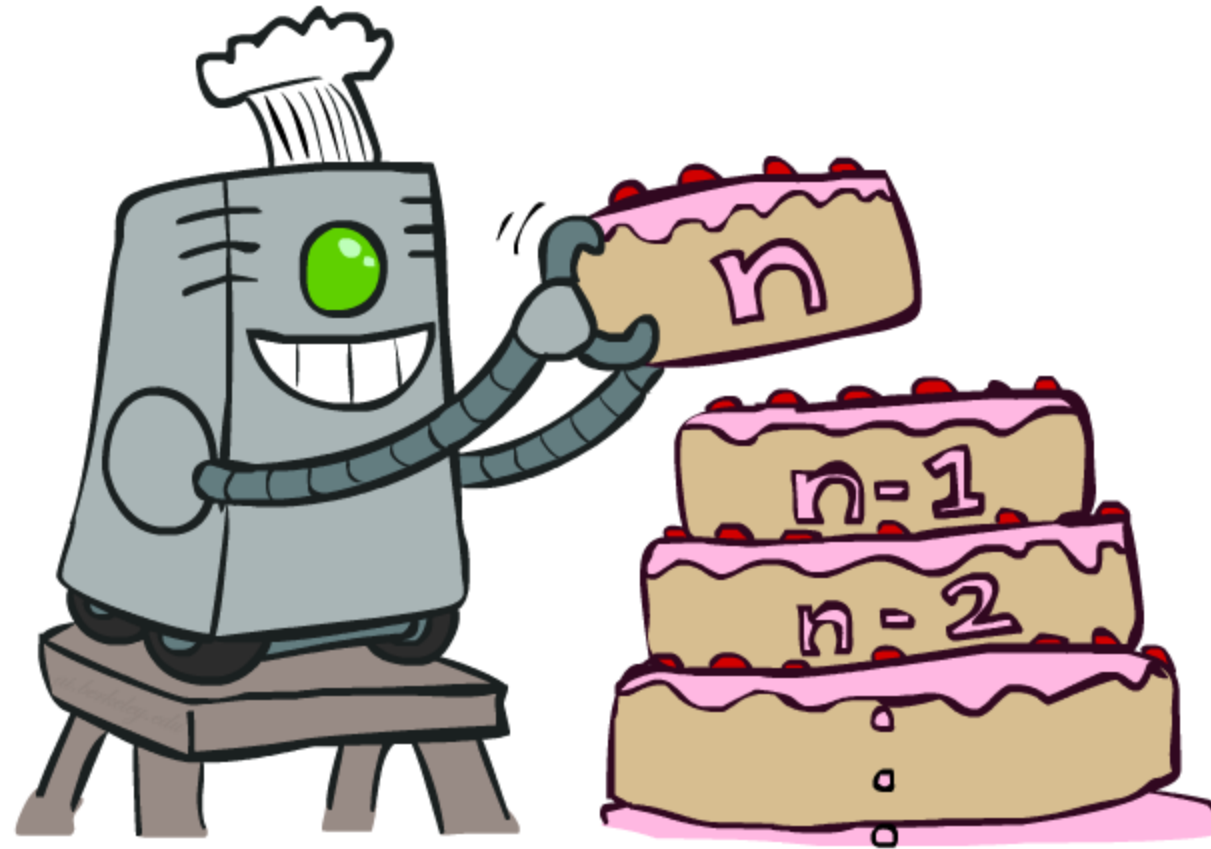
Step 2: Keep being optimal

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



Value Iteration



Value Iteration

- Bellman equations **characterize** the optimal values:

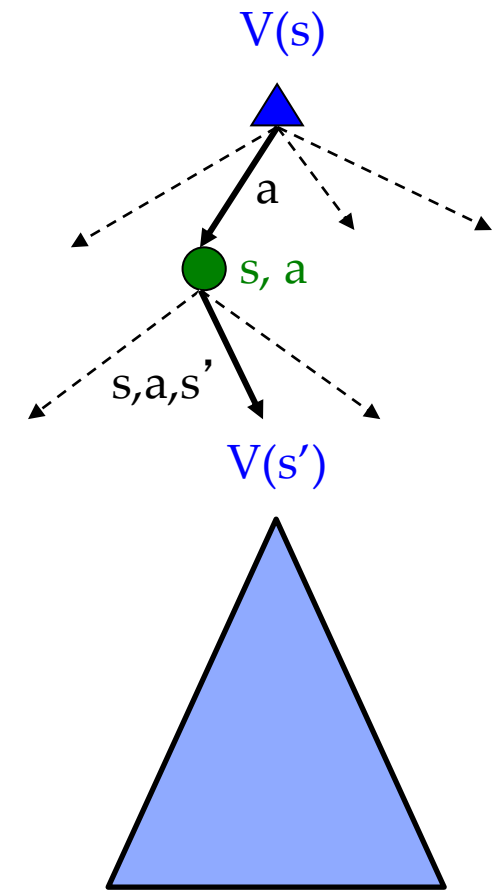
$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

- Value iteration **computes** them:

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

“Bellman Update”

- Value iteration is just a fixed point solution method
 - ... though the V_k vectors are also interpretable as time-limited values



Similarly, can have Q-Value Iteration

- Bellman Equation: recursive **definition** of Q-values

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

- Q-Value Iteration: Dynamic Programming

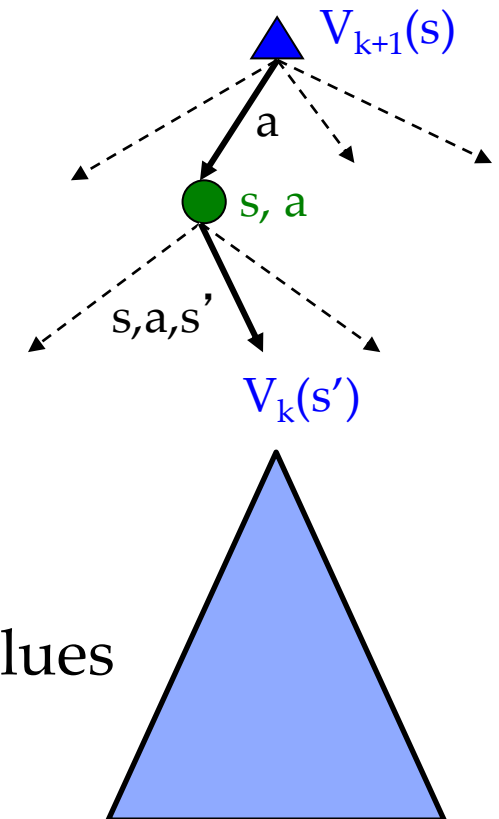
- $Q_{k+1}(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \max_{a'} Q_k(s', a')]$

Value Iteration: Dynamic Programming

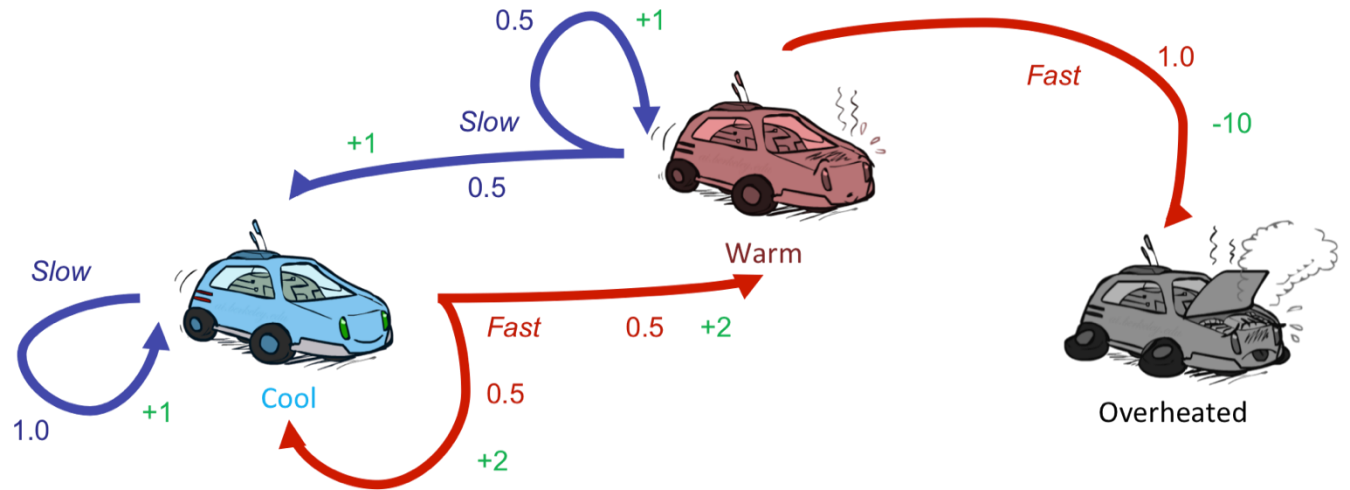
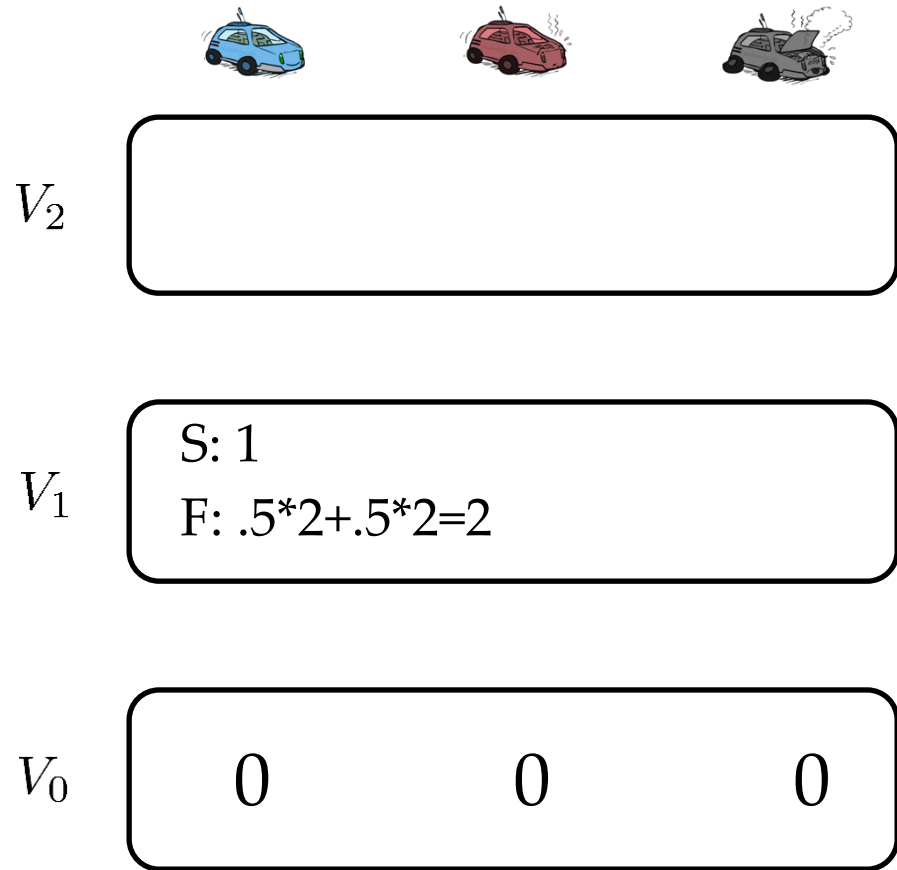
- Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

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

- $V = B(V)$ Where B is the Bellman update operator
- Repeat until convergence, which yields V^*
- Complexity of each iteration: $O(S^2A)$
- Theorem: Value Iteration will converge to unique optimal values
 - Basic idea: approximations get refined towards optimal values
 - Policy may converge long before values do



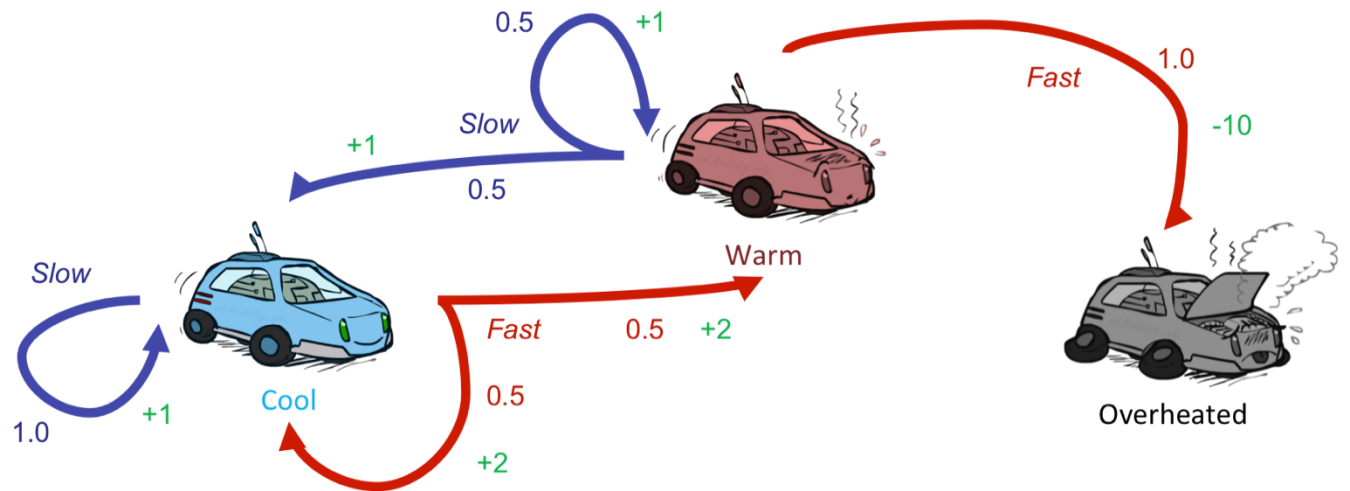
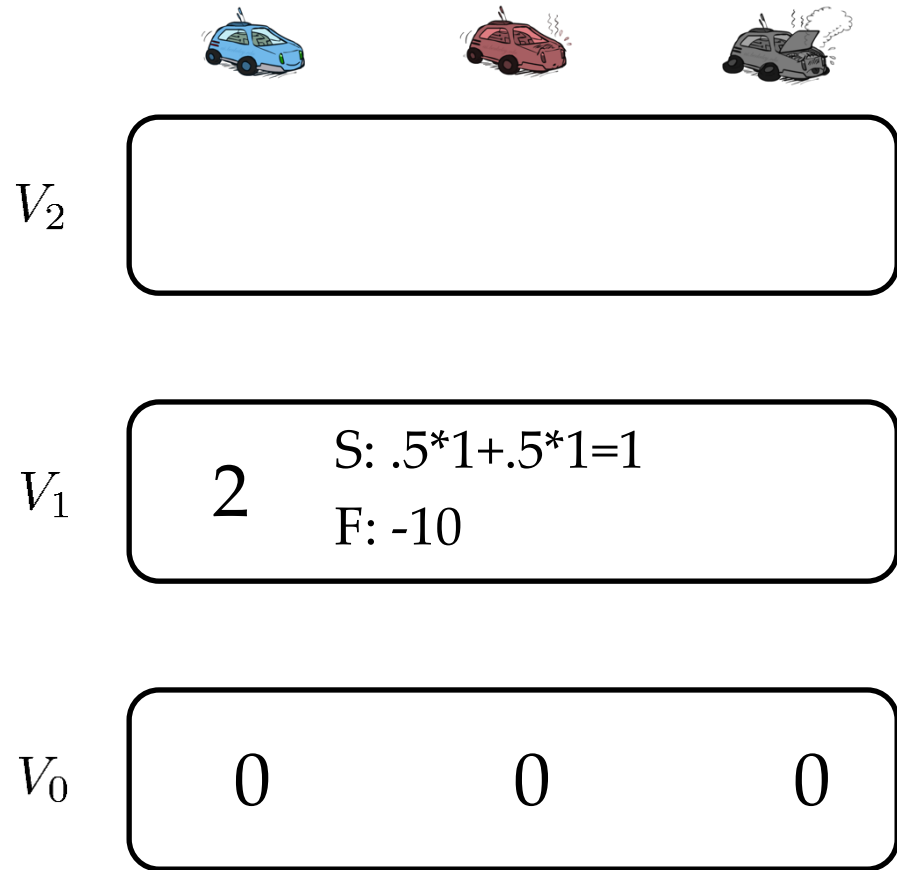
Example: Value Iteration



Assume no discount!

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




Example: Value Iteration

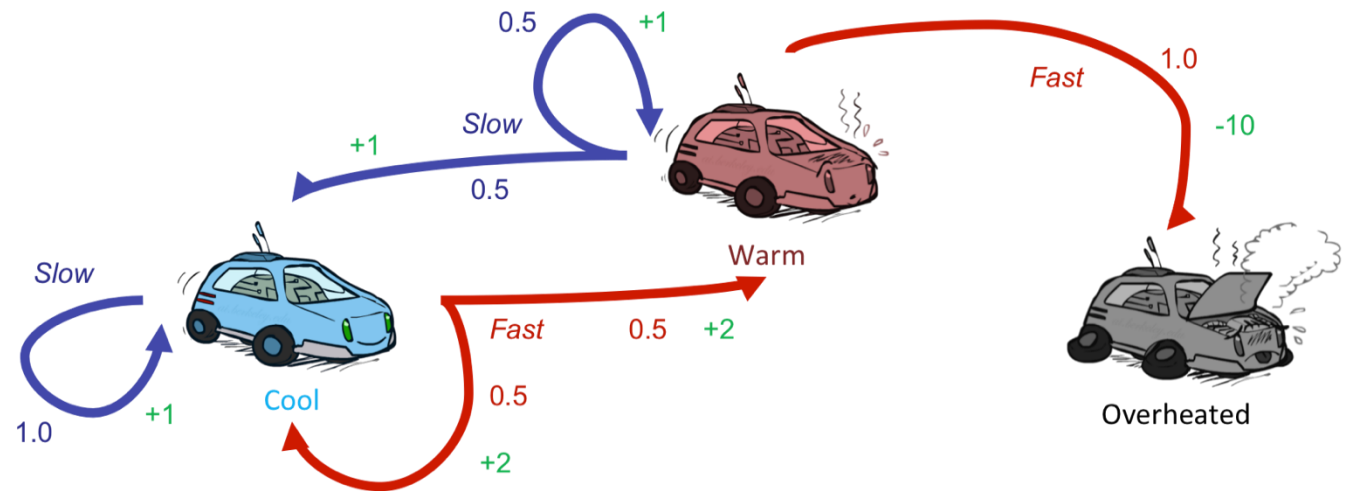


Assume no discount!

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

Example: Value Iteration

			
V_2			
V_1	2	1	0
V_0	0	0	0



Assume no discount!

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

Example: Value Iteration



V_2

S: $1+2=3$

F: $.5*(2+2)+.5*(2+1)=3.5$

V_1

2

1

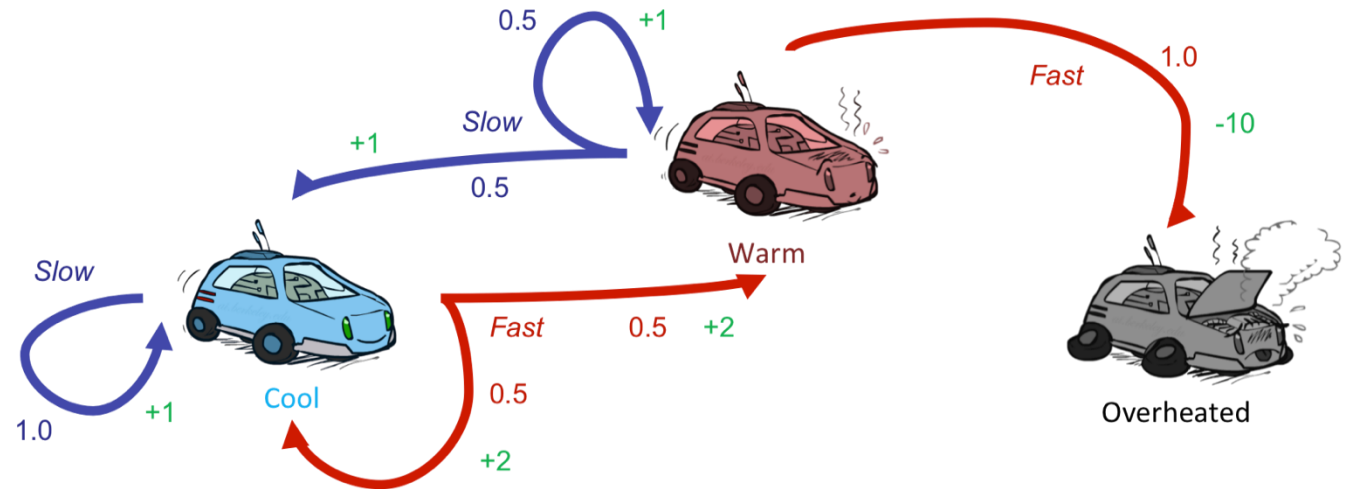
0

V_0

0

0




0

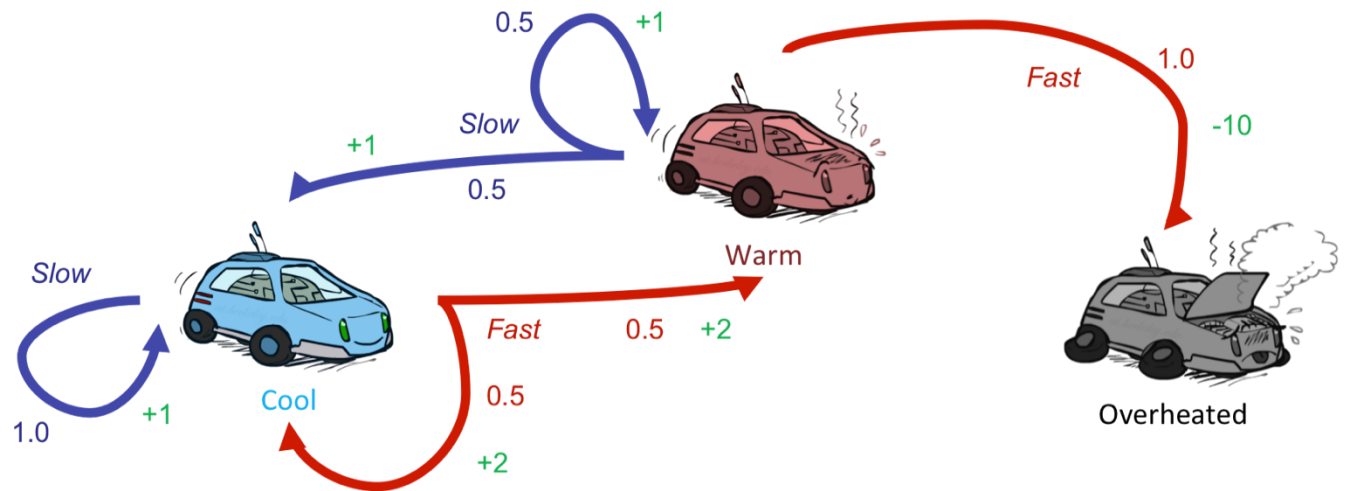


Assume no discount!

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

Example: Value Iteration

			
V_2	3.5	2.5	0
V_1	2	1	0
V_0	0	0	0

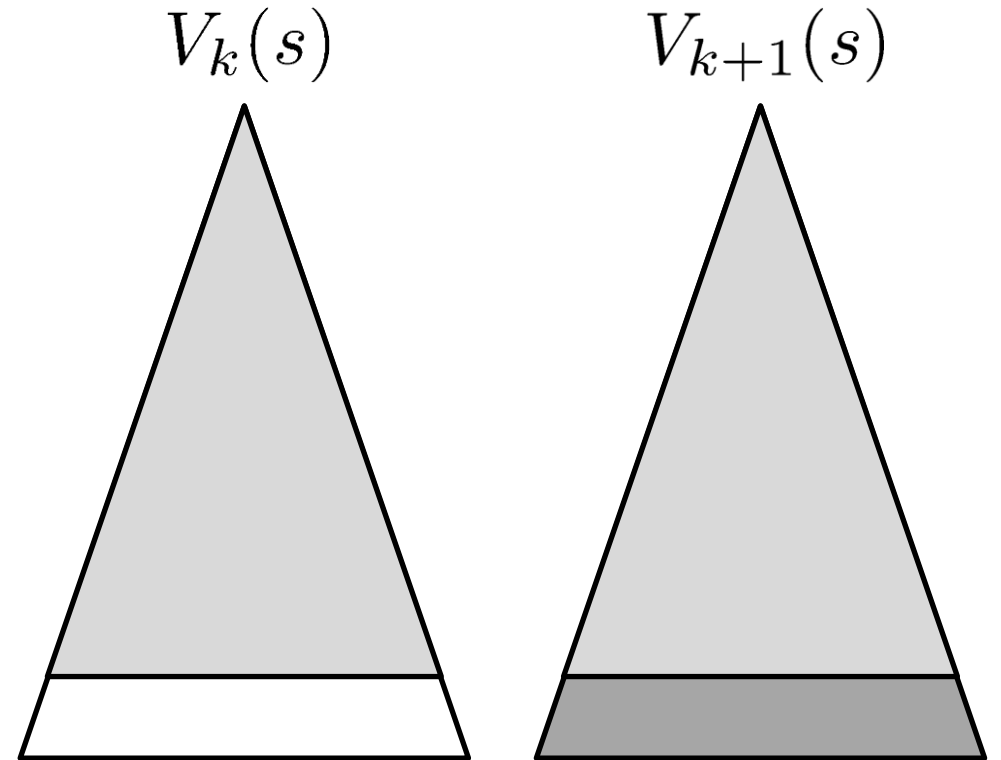


Assume no discount!

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

Convergence

- How do we know the V_k vectors are going to converge? (assuming $0 < \gamma < 1$)
- Proof Sketch:
 - For any state V_k and V_{k+1} can be viewed as depth $k+1$ expectimax results in nearly identical search trees
 - The difference is that on the bottom layer, V_{k+1} has actual rewards while V_k has zeros
 - That last layer is at best all R_{MAX}
 - It is at worst R_{MIN}
 - But everything is discounted by γ^k that far out
 - So V_k and V_{k+1} are at most $\gamma^k \max |R|$ different
 - So as k increases, the values converge



Convergence of Value Iteration: Contraction

- New concept: **contraction**
 - If some operator **F** is a contraction by a factor, it brings any pair of objects **closer** to each other (according to some metric $d(,)$)
 - For any x, y , we have $d(F(x), F(y)) \leq cd(x, y)$ where $c < 1$
 - If **F** is a contraction it has a unique fixed point z (i.e., $F(z)=z$)
- Since Value iteration is just $V_{k+1} = B(V_k)$, the Bellman update **B** is a contraction by γ
- Metric is the **max norm**: $\|V - W\| = \max_s |V(s) - W(s)|$
- What's the fixed point for B?
- $BV^* = V^*$

Speed of Convergence

- Look at what happens to the distance between V_k and V^*
- $\|V_{k+1} - V^*\|$
- $= \|BV_k - V^*\|$ (definition of V_{k+1} from VI update)
- $= \|BV_k - BV^*\|$ (V^* is the fixed point of B)
- $\leq \gamma \|V_k - V^*\|$ (B is a contraction by γ)
- I.e., the error is reduced by at least a factor γ on every iteration
- Exponentially fast convergence!

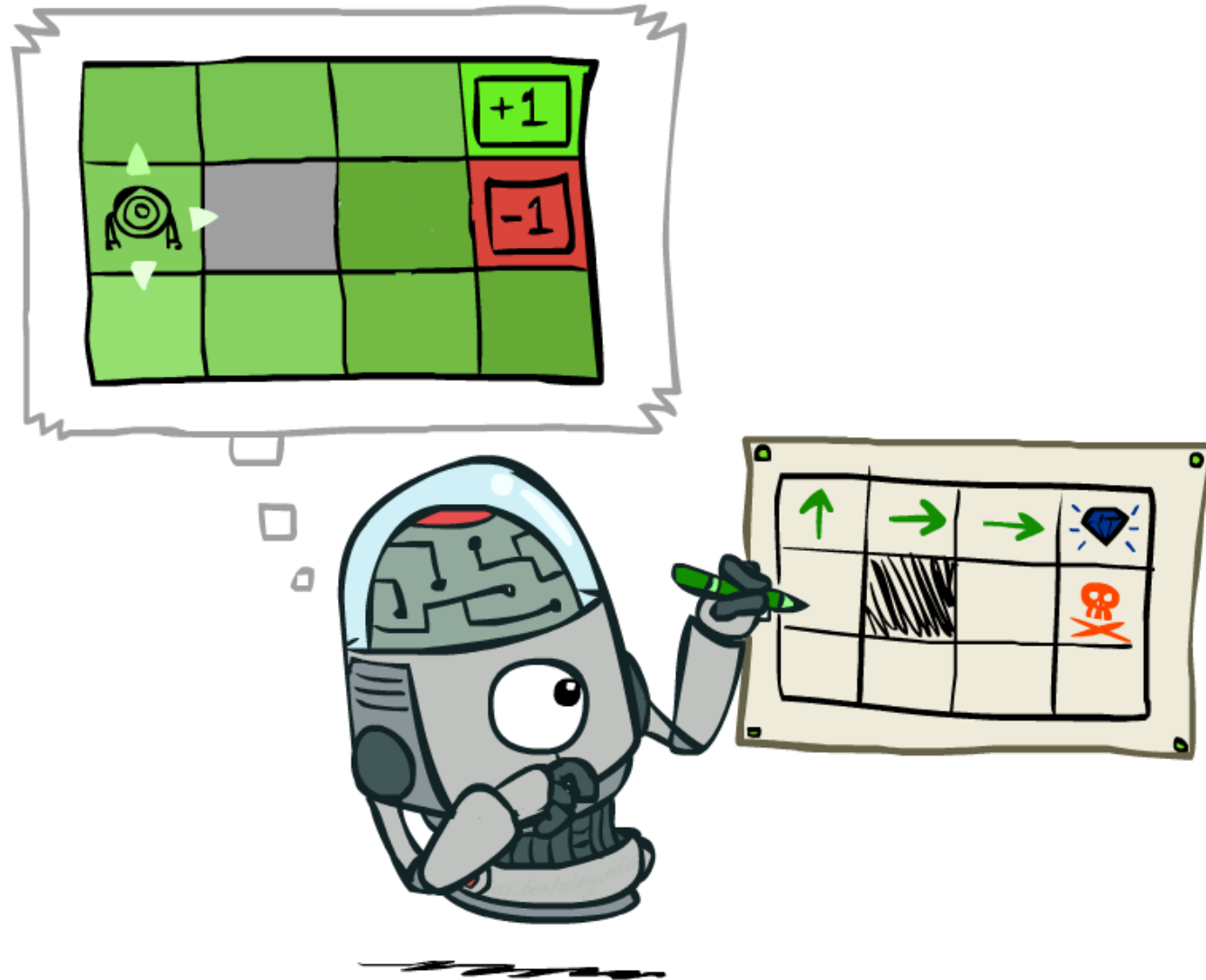
Correctness of Convergence

- Don't usually converge exactly; stops when change $< \frac{\epsilon(1-\gamma)}{\gamma}$
- I.e. $\|V_{k+1} - V_k\| < \frac{\epsilon(1-\gamma)}{\gamma}$
- What about $\|V_{k+1} - V^*\|$ when $\|V_{k+1} - V_k\| < \frac{\epsilon(1-\gamma)}{\gamma}$
- Useful properties:
 - Contraction: $\|V_{k+1} - V^*\| \leq \gamma \|V_k - V^*\|$
 - Triangle inequality: $\|V_k - V^*\| \leq \|V_{k+1} - V_k\| + \|V_{k+1} - V^*\|$

Correctness of Convergence

- Value Iteration: stop when $\|V_{k+1} - V_k\| < \frac{\epsilon(1-\gamma)}{\gamma}$
- What about $\|V_{k+1} - V^*\|$ when $\|V_{k+1} - V_k\| < \frac{\epsilon(1-\gamma)}{\gamma}$?
- Triangle inequality: $\|V_k - V^*\| \leq \|V_{k+1} - V_k\| + \|V_{k+1} - V^*\|$
- $\frac{1}{\gamma} \|V_{k+1} - V^*\| \leq \|V_{k+1} - V_k\| + \|V_{k+1} - V^*\|$ B(V) is contraction by γ
- $\left(\frac{1}{\gamma} - 1\right) \|V_{k+1} - V^*\| \leq \|V_{k+1} - V_k\|$
- $\left(\frac{1}{\gamma} - 1\right) \|V_{k+1} - V^*\| \leq \frac{\epsilon(1-\gamma)}{\gamma}$ when we have converged
- $\|V_{k+1} - V^*\| \leq \epsilon$

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)



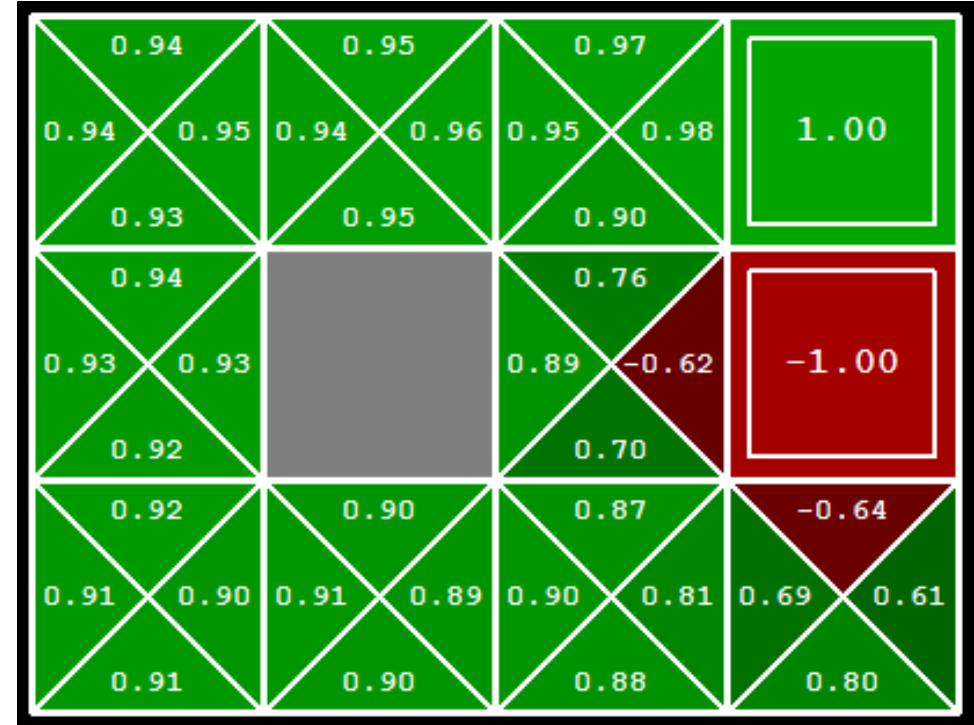
$$\pi^*(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

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

Computing Actions from Q-Values

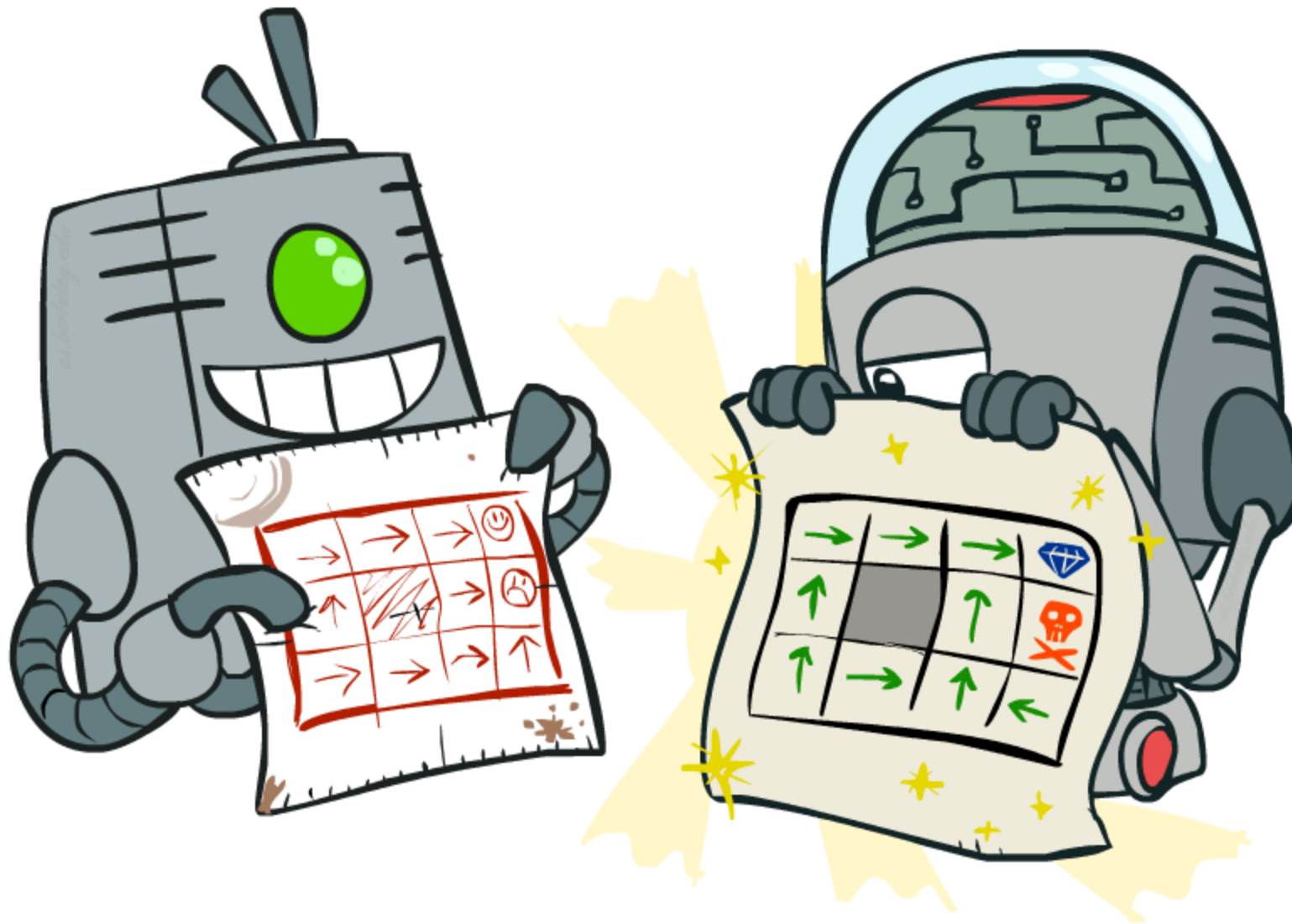
- Let's imagine we have the optimal q-values:
- How should we act?
 - Completely trivial to decide!

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



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

Policy Methods

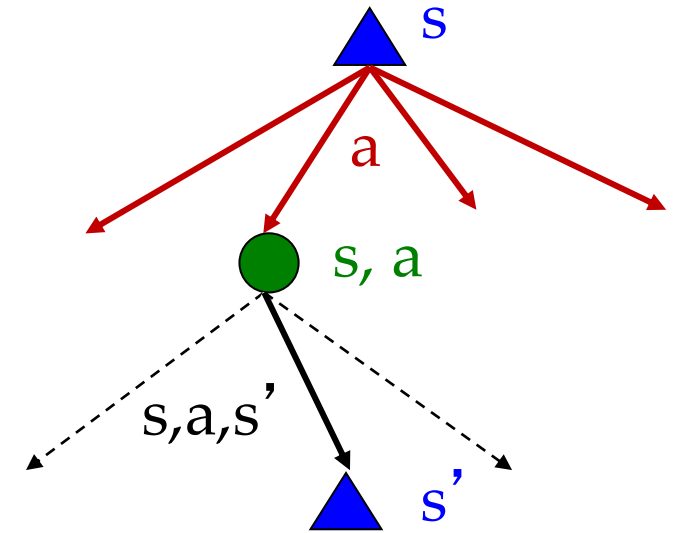


Problems with Value Iteration

- Value iteration repeats the Bellman updates:

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

- Problem 1: It's slow – $O(S^2A)$ per iteration
- Problem 2: The “max” at each state rarely changes
- Problem 3: The policy often converges long before the values



k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100

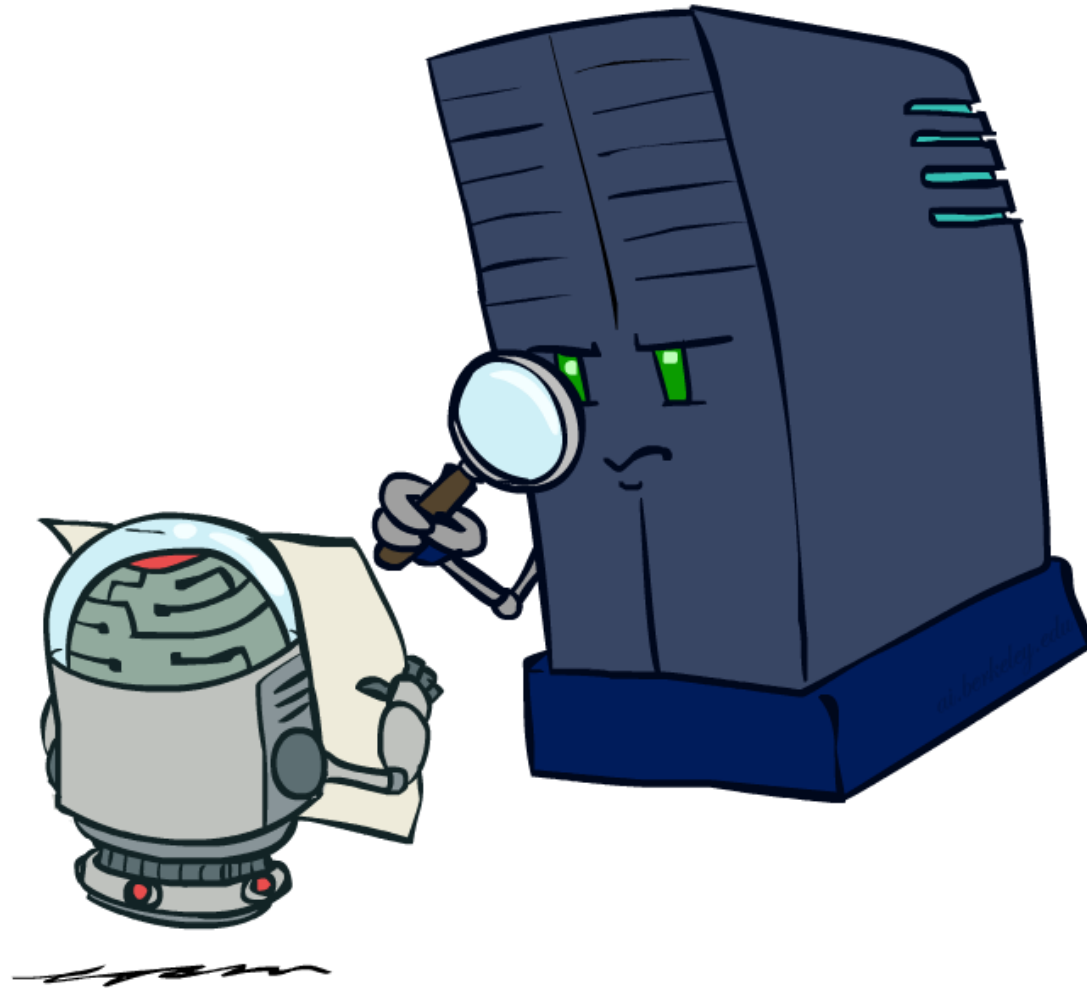


Noise = 0.2
Discount = 0.9
Living reward = 0

Policy Iteration

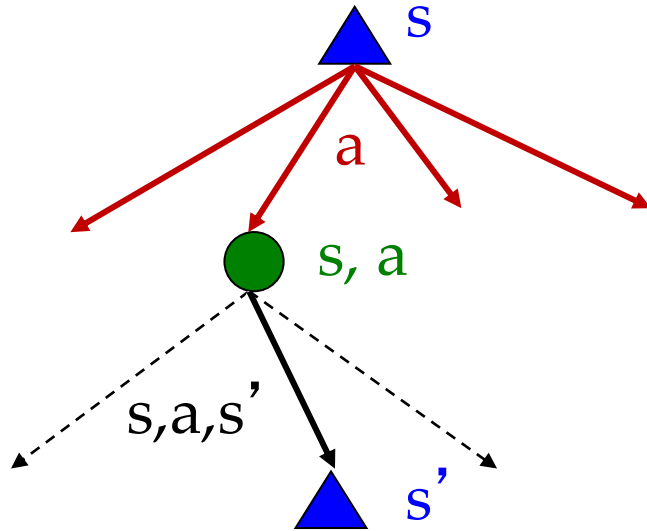
- Alternative approach for optimal values:
 - **Step 1: Policy Evaluation:** calculate utilities for some fixed policy (not optimal utilities!) until convergence
 - **Step 2: Policy Improvement:** update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
 - Repeat steps until policy converges
- This is **Policy Iteration**
 - It's still optimal!
 - Can converge (much) faster under some conditions

Policy Evaluation

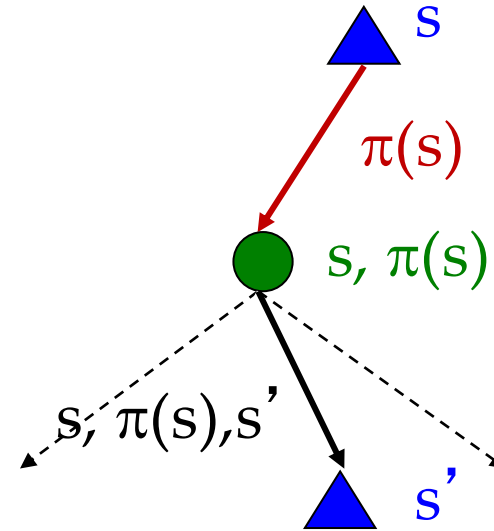


Fixed Policies

Do the optimal action



Do what π says to do

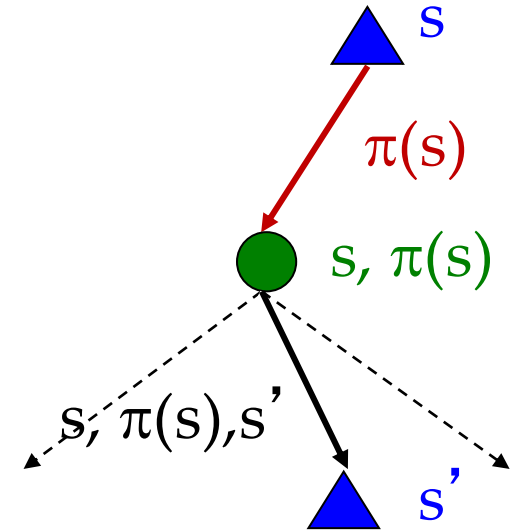


- Expectimax trees max over all actions to compute the optimal values
- If we fixed some policy $\pi(s)$, then the tree would be simpler – only one action per state
 - ... though the tree's value would depend on which policy we fixed

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 :
 $V^\pi(s)$ = expected total discounted rewards starting in s and following π
- Recursive relation (one-step look-ahead / Bellman equation):

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

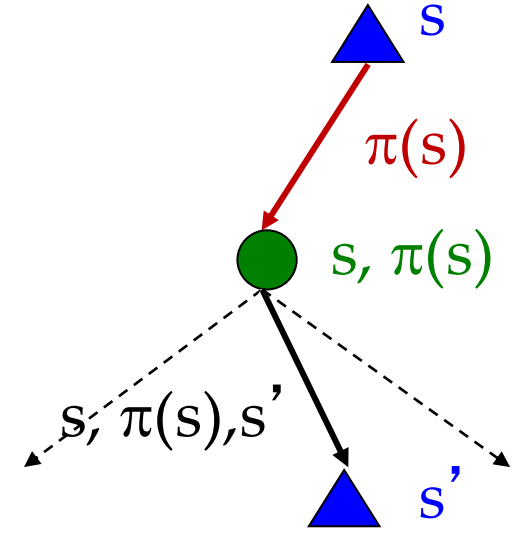


Policy Evaluation

- How do we calculate the V 's for a fixed policy π ?
- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

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

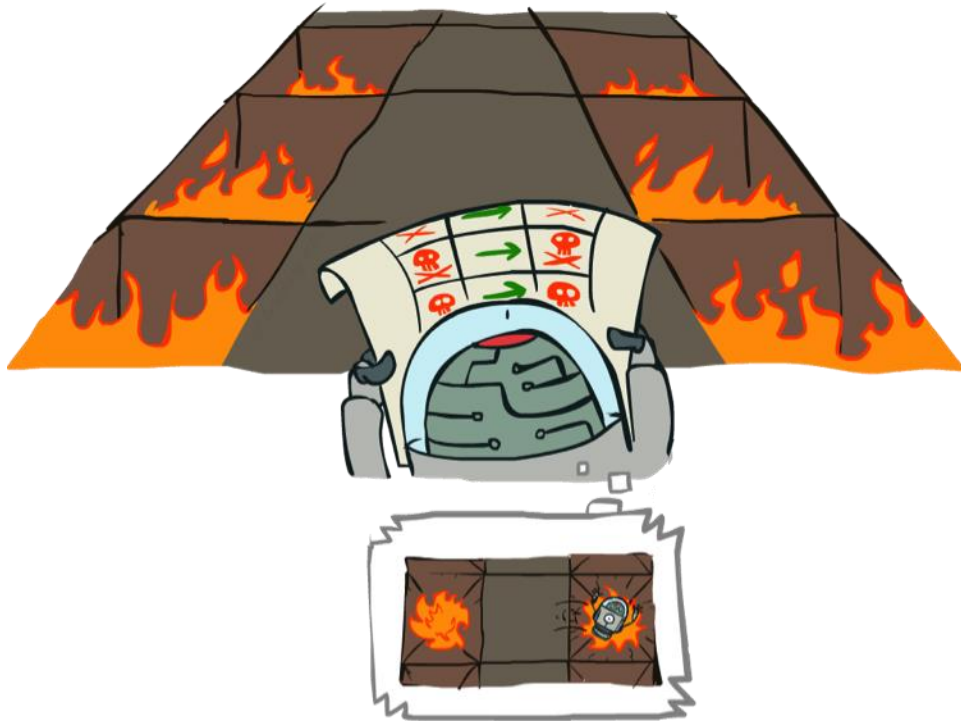
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$



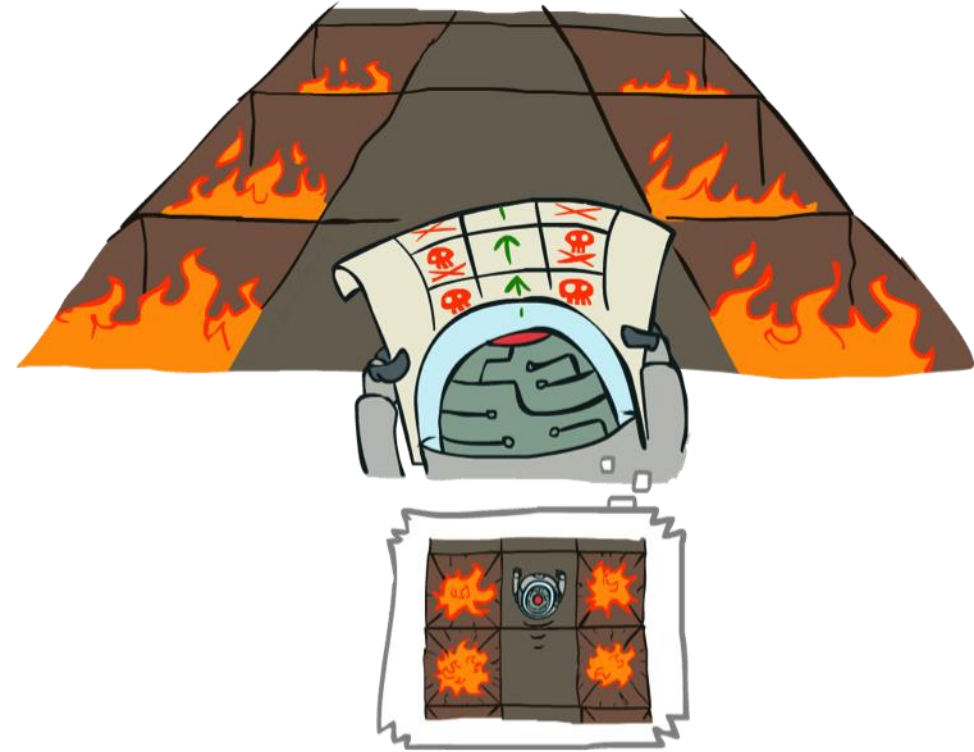
- Efficiency: $O(S^2)$ per iteration
- Idea 2: Without the maxes, the Bellman equations are just a linear system
 - Solve the system of equations

Example: Policy Evaluation

Always Go Right



Always Go Forward

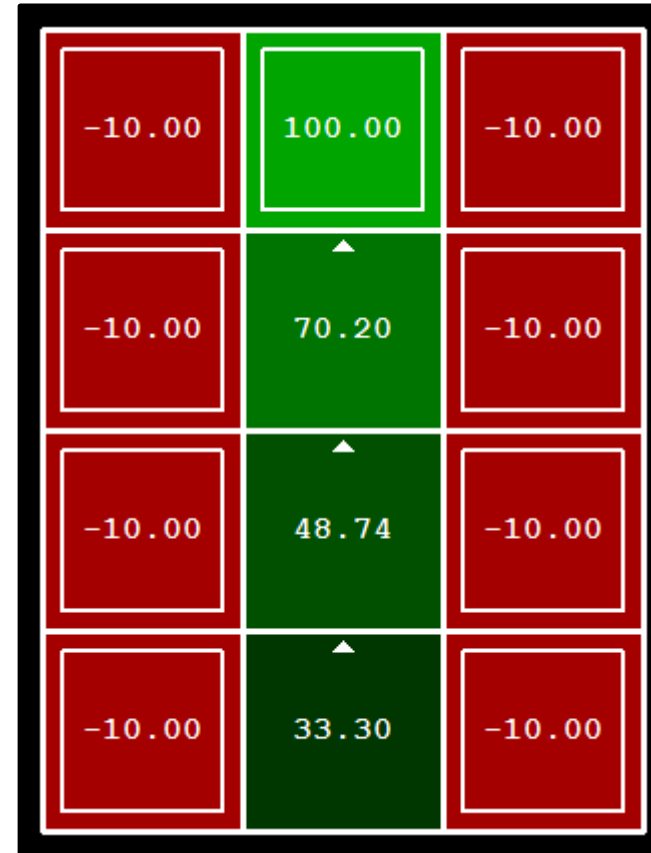


Example: Policy Evaluation

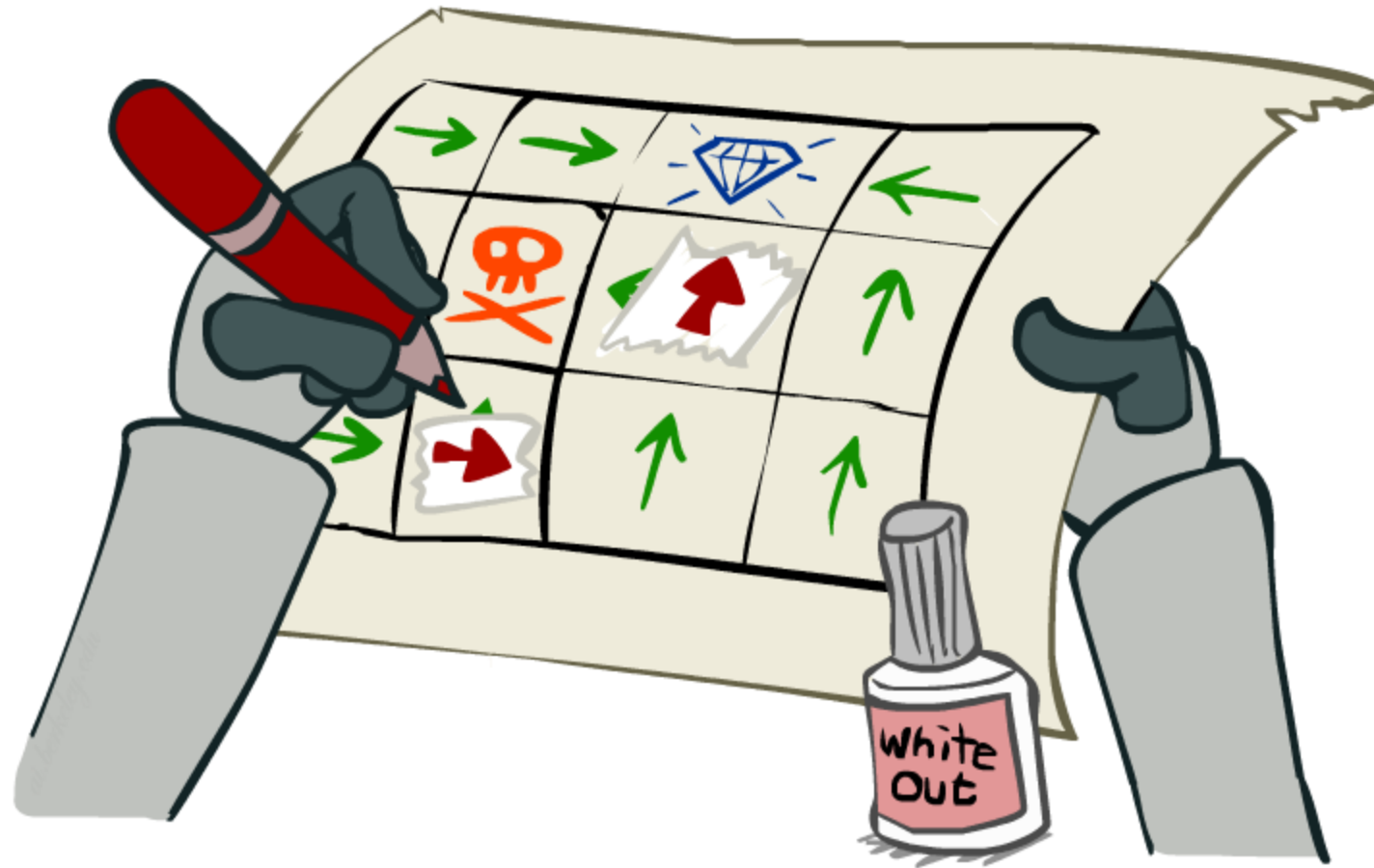
Always Go Right



Always Go Forward



Policy Iteration



Policy Iteration

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

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

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

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

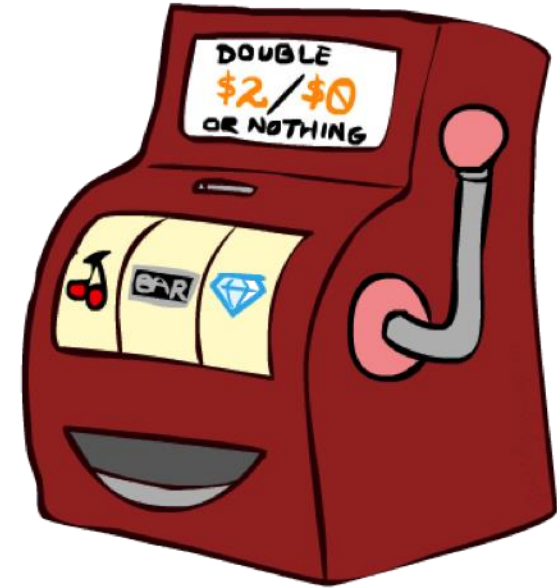
Comparison

- 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)
 - The new policy will be better (or we're done)
- Both are dynamic programming approaches for solving MDPs

Summary: MDP Algorithms

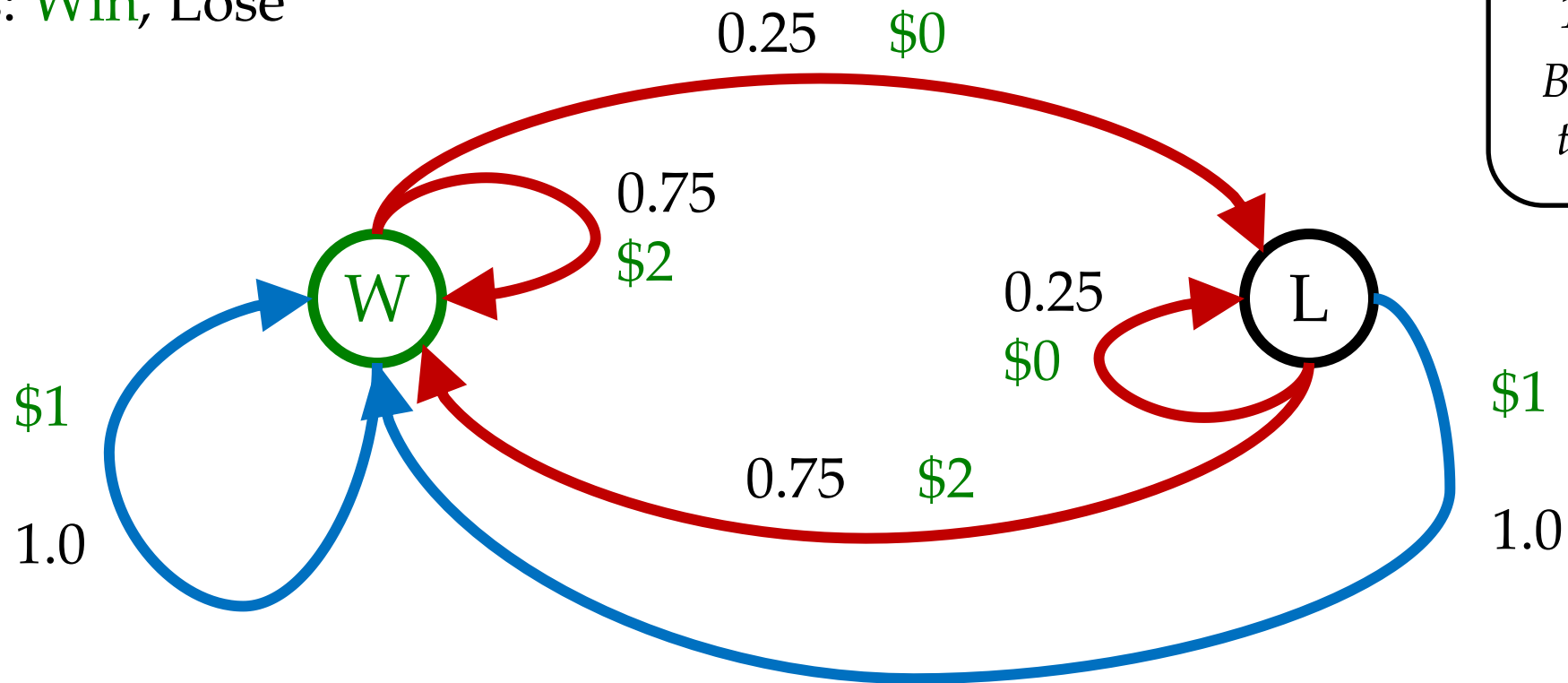
- So you want to....
 - Compute optimal values: use value iteration or policy iteration
 - Compute values for a particular policy: use policy evaluation
 - Turn your values into a policy: use policy extraction (one-step lookahead)
- These all look the same!
 - They basically are – they are all variations of Bellman updates
 - They all use one-step lookahead expectimax fragments
 - 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*



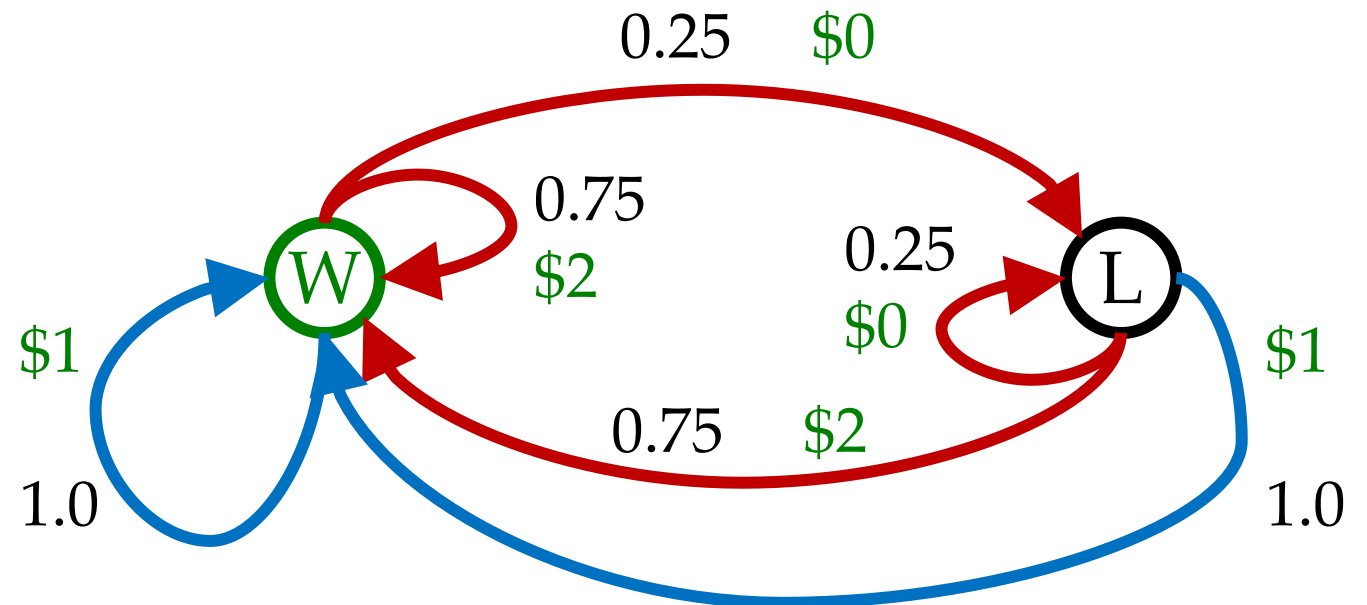
*No discount
100 time steps
Both states have
the same value*

Offline Planning

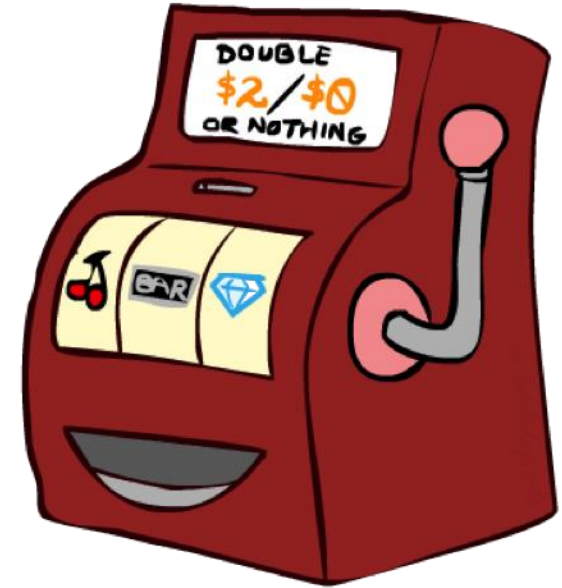
- Solving MDPs is offline planning
 - You determine all quantities through computation
 - You need to know the details of the MDP
 - You do not actually play the game!

*No discount
100 time steps
Both states have
the same value*

	Value
Play Red	150
Play Blue	100



Let's Play!

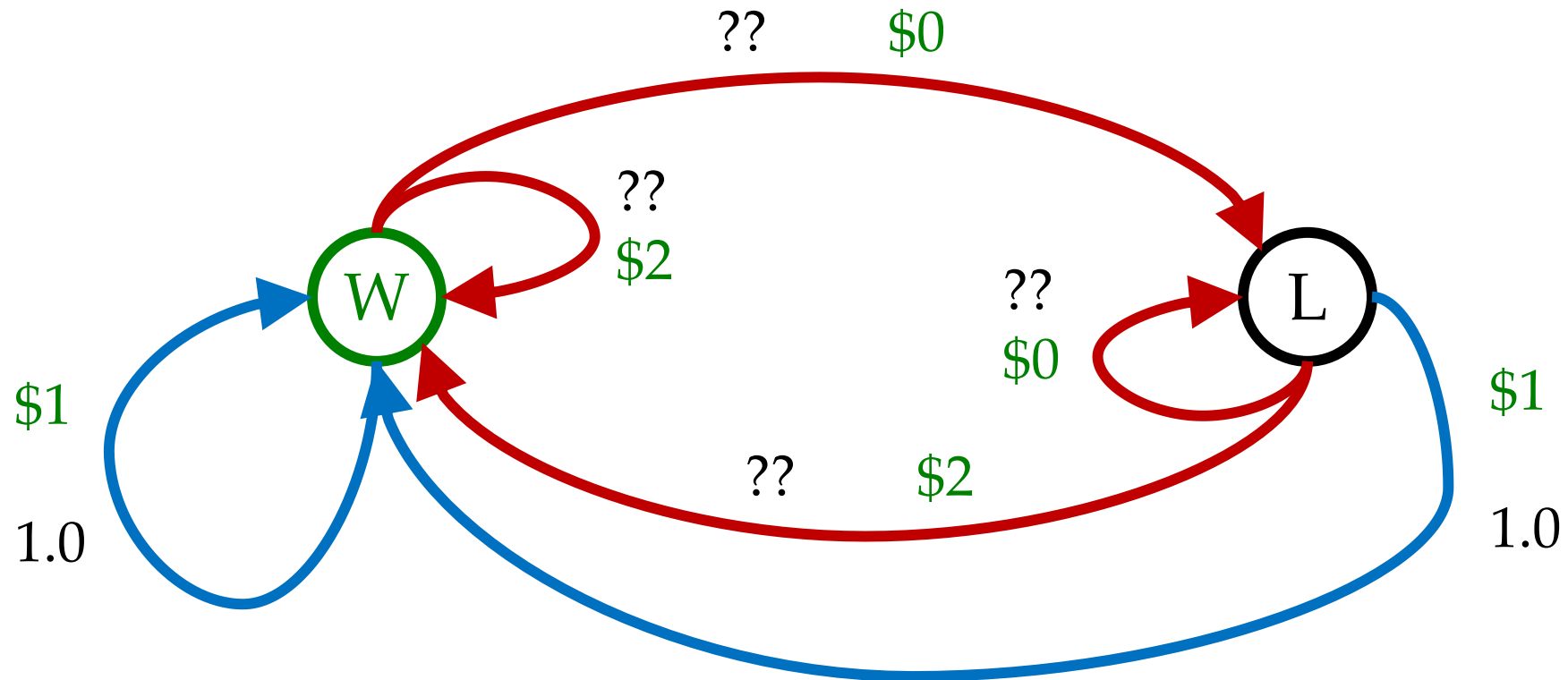


\$2 \$2 \$0 \$2 \$2

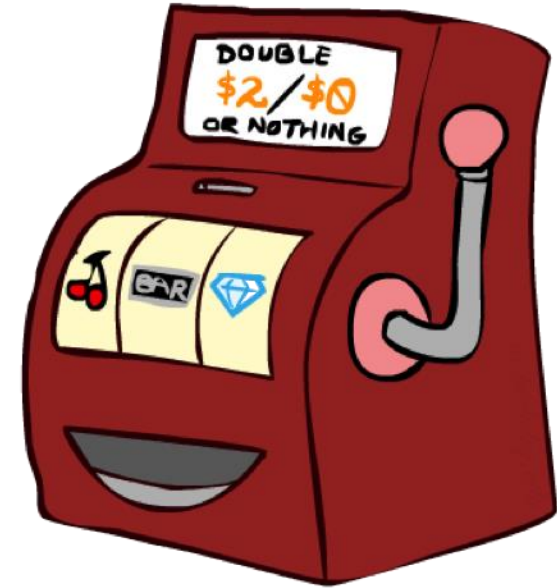
\$2 \$2 \$0 \$0 \$0

Online Planning

- 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