Reinforcement Learning

Chen-Yu Wei

Overview on what we have talked about

- Search
  - Single-agent search
  - Multi-agent search
  - Constraint satisfaction
  - Logic
- Probabilistic Modeling
  - Bayesian network
  - (Hidden) Markov models
- Machine Learning
  - Learning from data

Search → Finding a series of decisions or a solution in a large state space (Modeling the relation between variables **deterministically**)

Probabilistic Modeling → Modeling the relation between variables **probabilistically**

Machine Learning → Learning the relation between variables from data

Markov Decision Processes and Reinforcement Learning

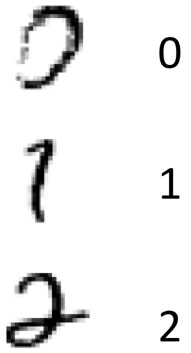
- Search
  - Single-agent search
  - Multi-agent search
  - Constraint satisfaction
  - Logic
- Probabilistic Modeling
  - Bayesian network
  - (Hidden) Markov models
- Machine Learning
  - Learning the model from data

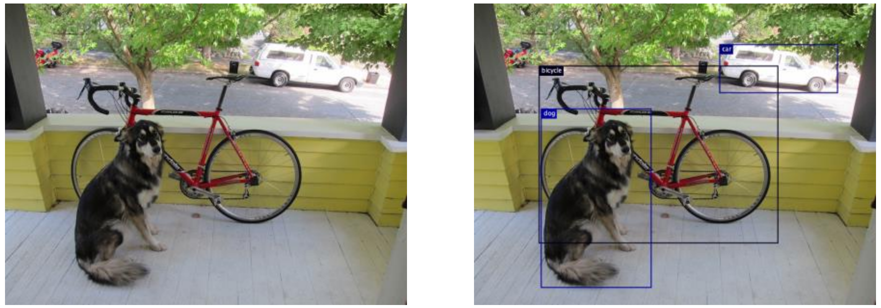
Probabilistic model for search problems
(Markov decision processes)

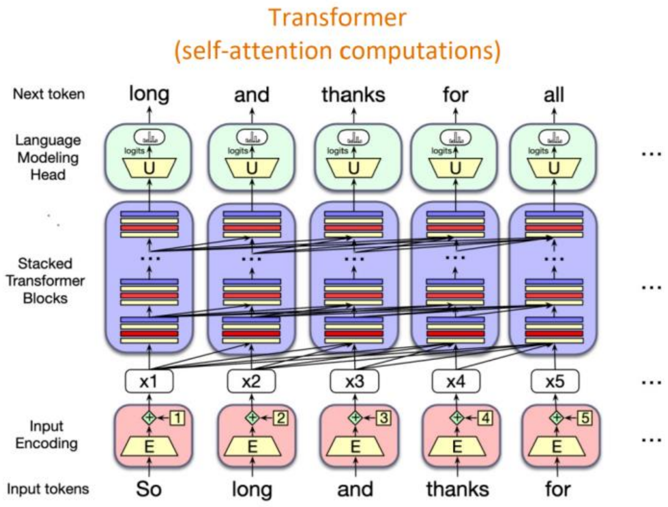
Searching while learning the model
(Reinforcement Learning)

Reinforcement Learning (RL) vs. other ML methods

- How is RL different from the ML methods we have seem so far?







X: image

Y: digit

X: image

Y: bounding box

supervised learning

X: $(x_1, x_2, \ldots, x_{i-1})$

Y: x_i

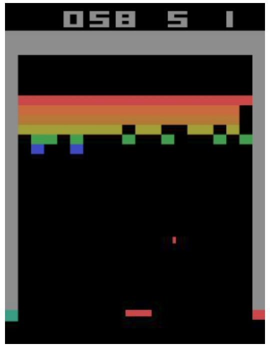
self-supervised learning

Reinforcement Learning (RL) vs. other ML methods

- In supervised learning or self-supervised learning, it is important that we (human) have to collect a big amount of training data (i.e., (X, Y) pairs)
  - Bounding box: human labeling
  - Texts: web crawler

- Reinforcement learning handles problems where the machine has to collect data by itself while learning

Reinforcement Learning



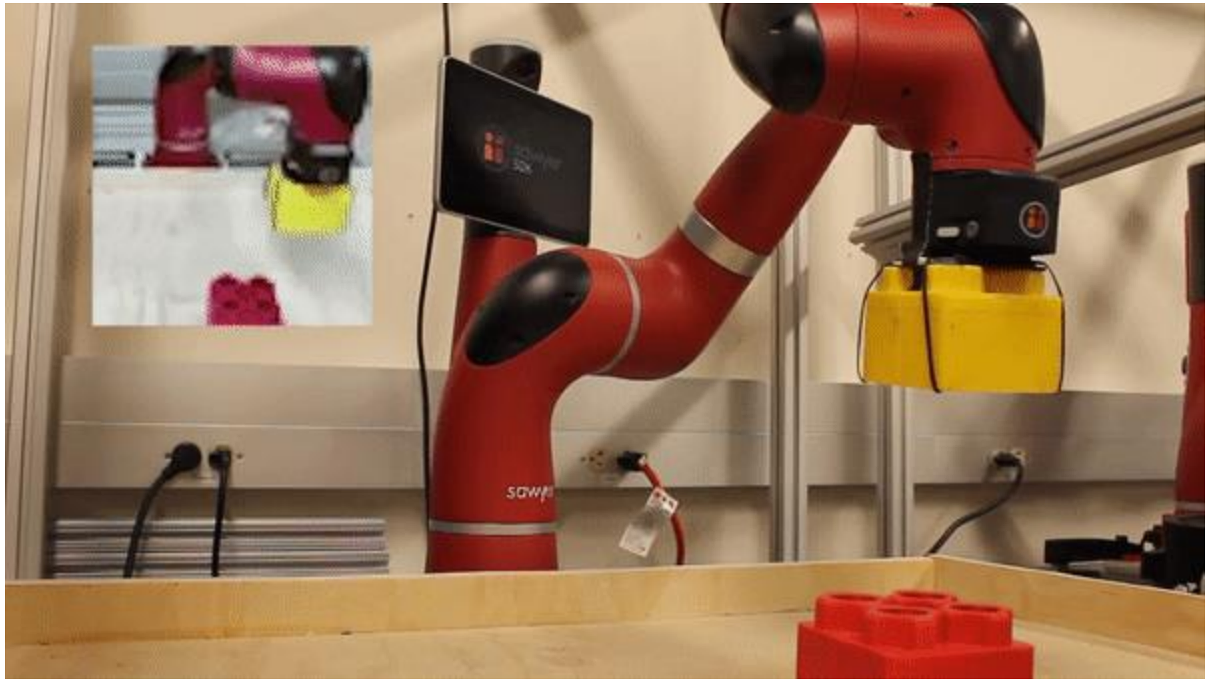
X: View of the game   Y: Action (left or right)

Instead of providing training data to the machine, we let it collect them **by itself** (through trial and error).

Instead of telling the machine which action to take, we only tell it **reward** (like in search problems).

Difference between telling action and telling reward: in the former case, the machine can just follow the action, but in the latter case, the machine still needs to try different actions.

Reinforcement Learning



.

Markov Decision Process

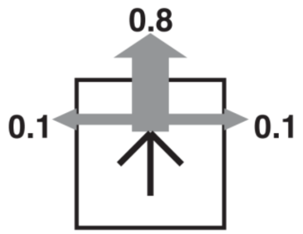
(Just a probabilistic model for search problems --- no “learning”)

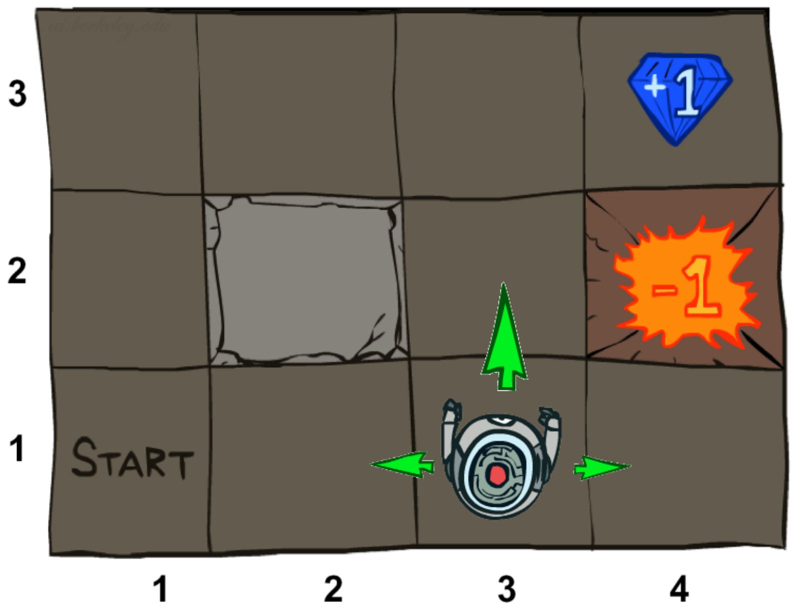
Example: Grid World

- 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

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

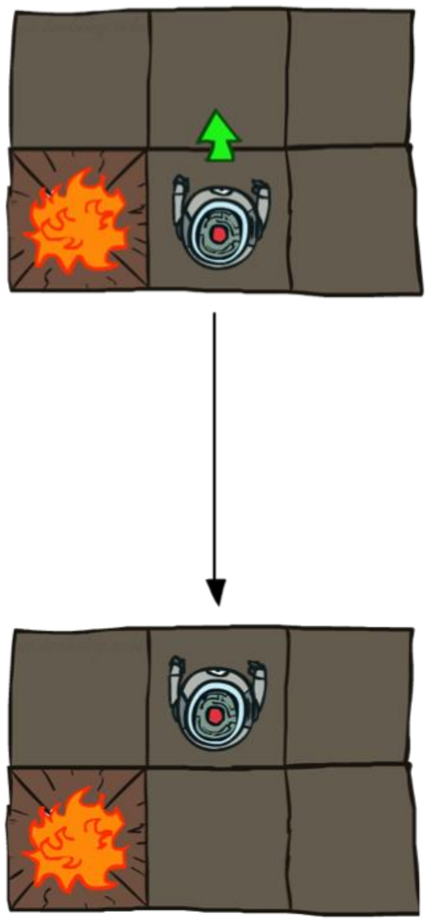
- Goal: maximize sum of rewards



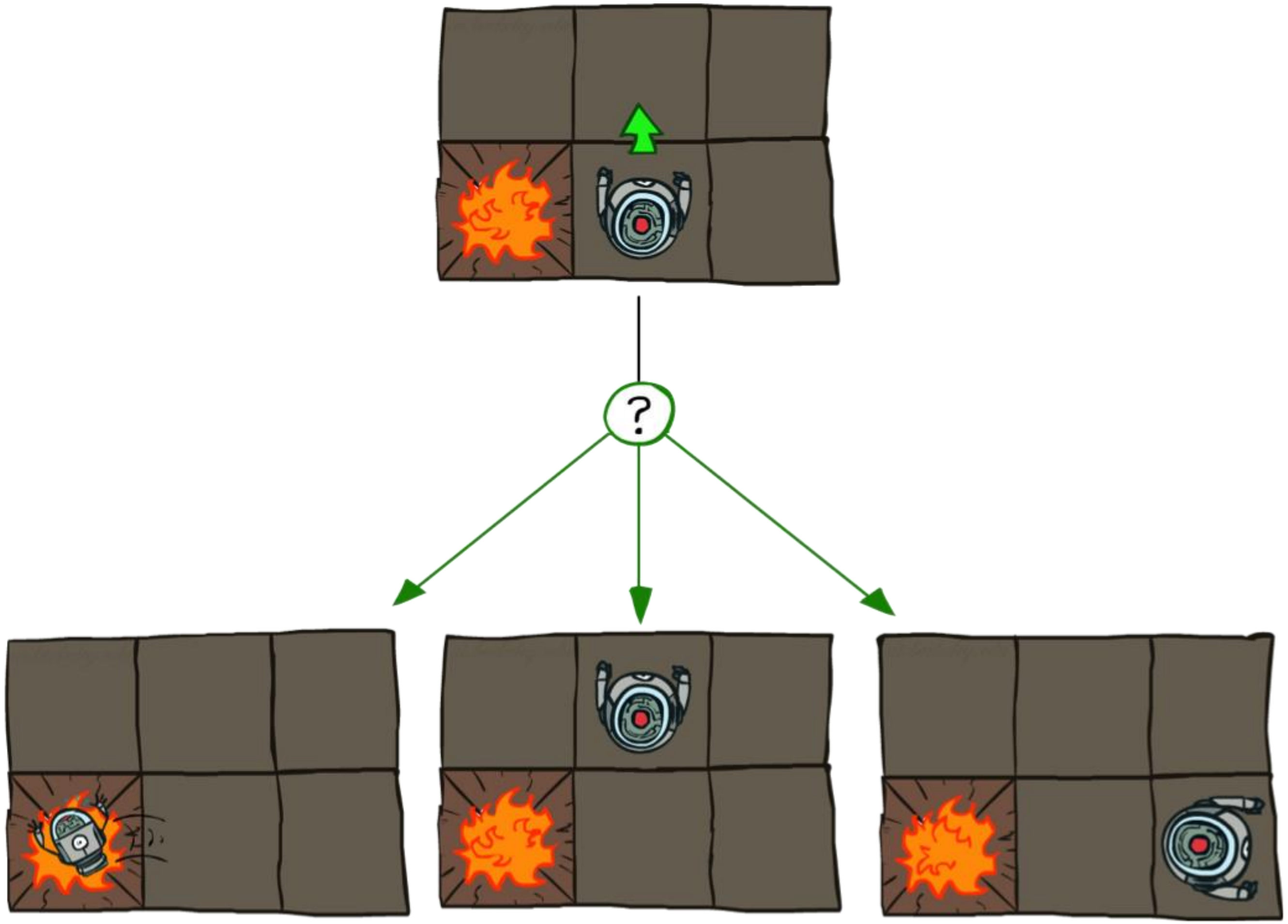


Grid World Actions

Deterministic Grid World

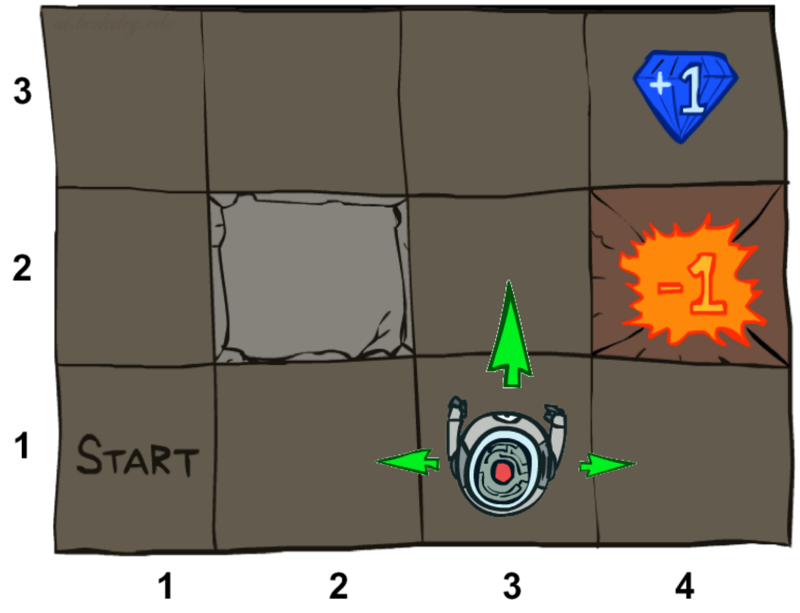


Stochastic Grid World



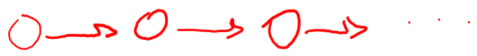
Markov Decision Processes

- 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 $R(s, a, s')$ or $R(s,a)$
    - Sometimes just $R(s)$ or $R(s')$
  - A start state
  - Maybe a terminal state



What is Markov about MDPs?

- "Markov" generally means that given the present state, the future and the past are independent

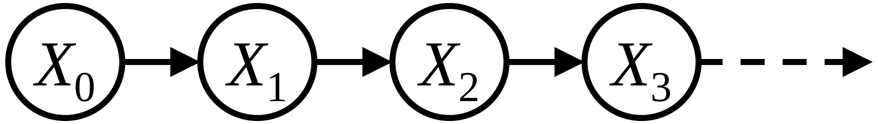


- For Markov decision processes, "Markov" means action outcomes depend only on the current state

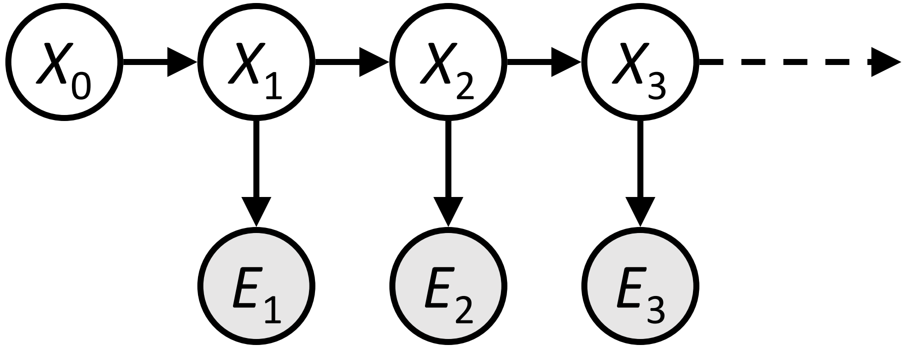
$$
\begin{aligned}
&P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0) \\
=\; &P(S_{t+1} = s' | S_t = s_t, A_t = a_t)
\end{aligned}
$$

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

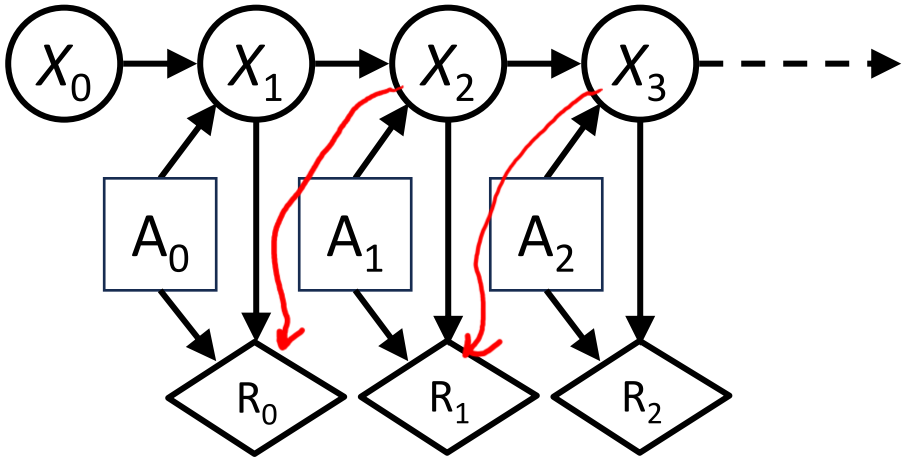
“Markov” as in Markov Chains? HMMs?



Markov Model (Markov Chain)



Hidden Markov Model



Markov Decision Process

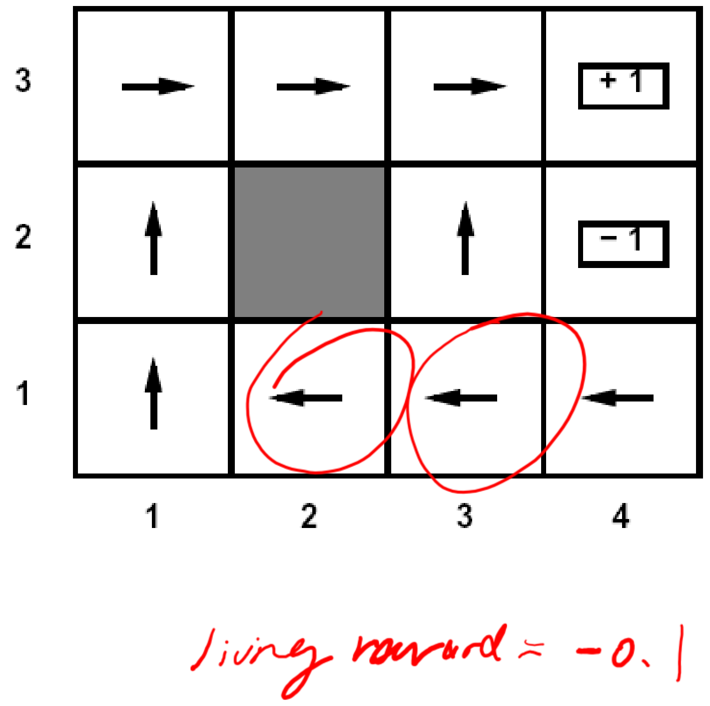
???

Partially Observable Markov Decision Process

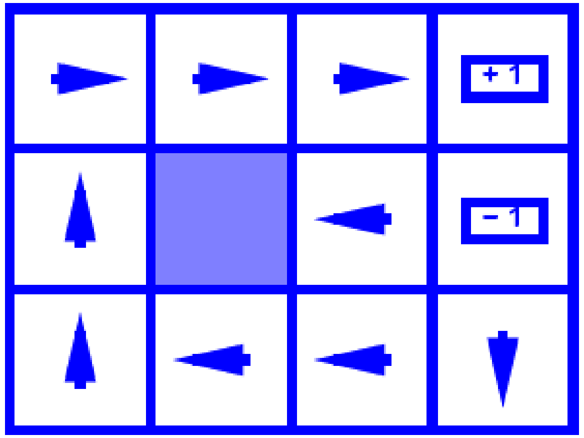
Policies

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal

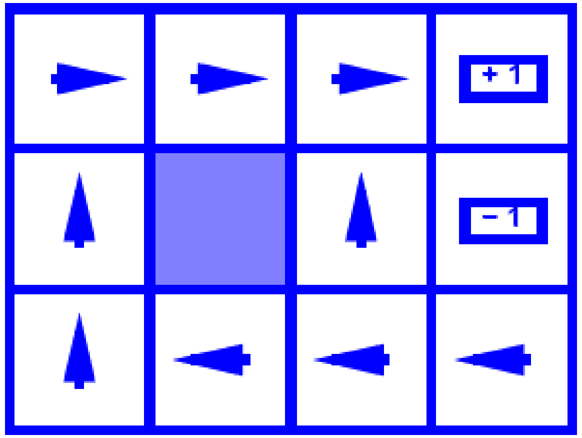
- For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$
  - A policy π gives an action for each state
  - An optimal policy is one that maximizes expected total return



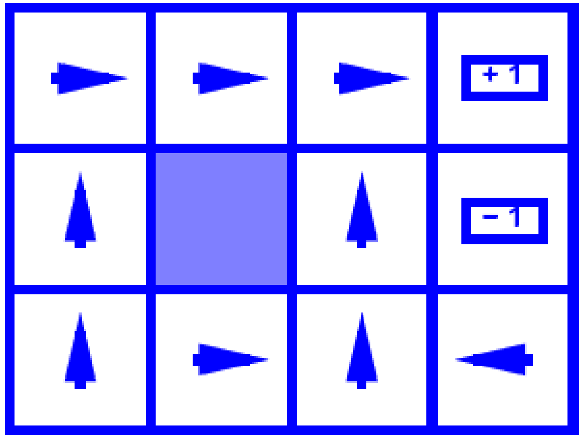
Optimal Policies



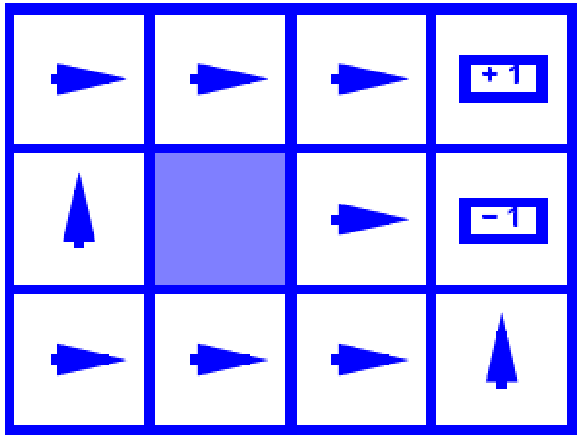
R(s) = -0.01



R(s) = -0.03

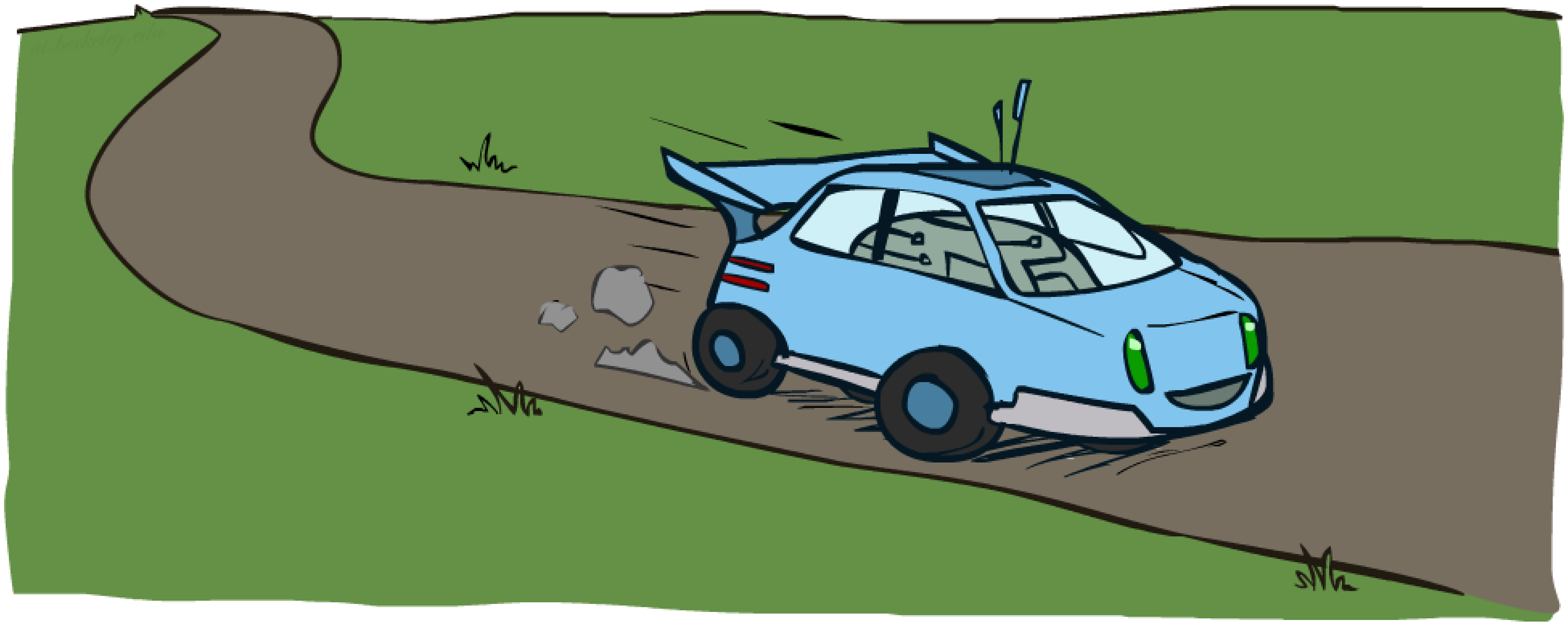


R(s) = -0.4



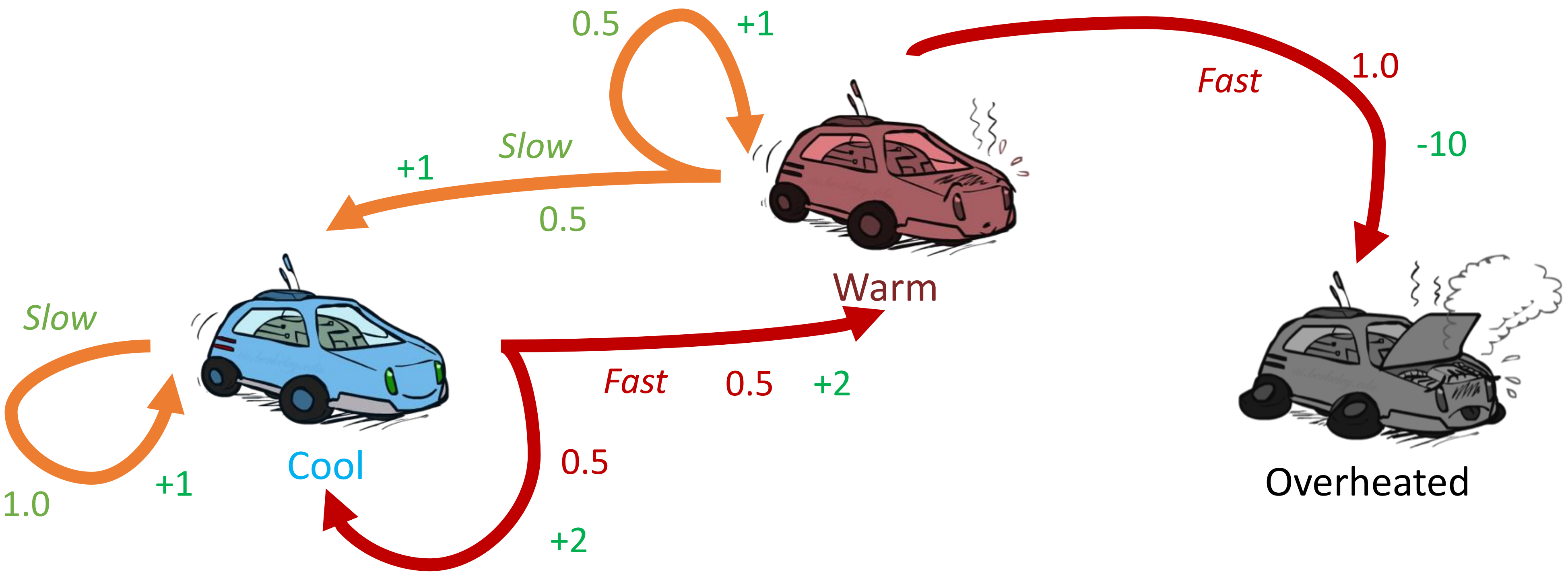
R(s) = -2.0

Example: Racing



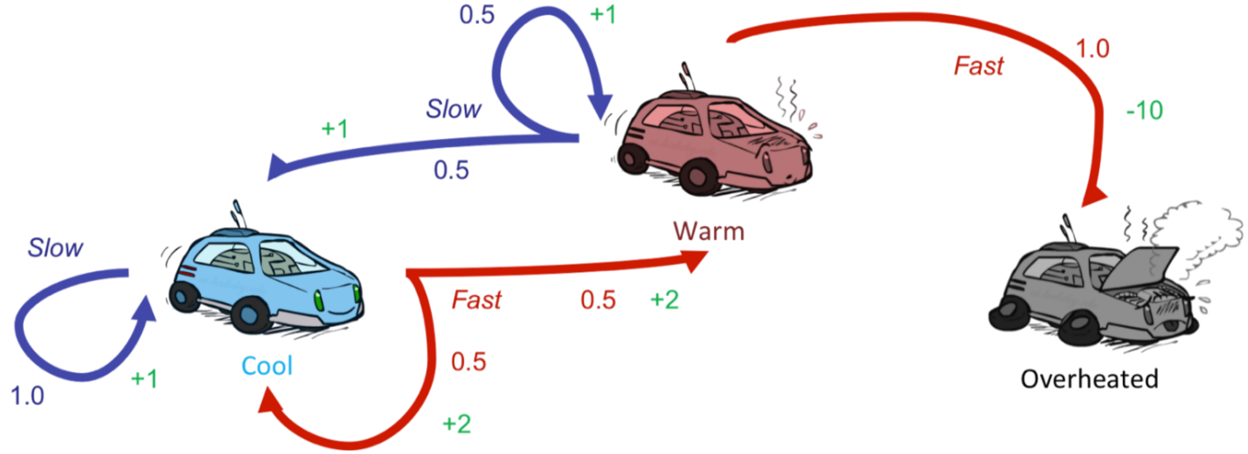
Example: Racing

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



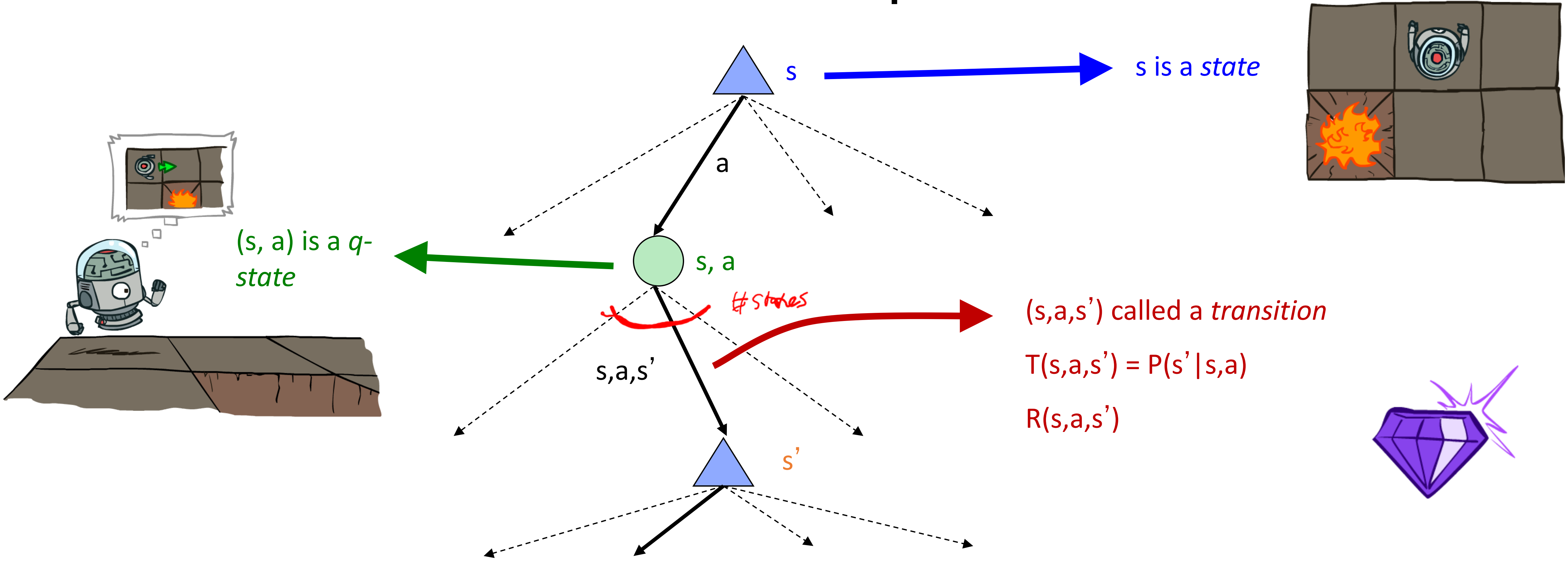
Example: Racing

s	a	s'	T(s,a,s’)	R(s,a,s’)
	Slow		1.0	+1
	Fast		0.5	+2
	Fast		0.5	+2
	Slow		0.5	+1
	Slow		0.5	+1
	Fast		1.0	–10
	(end)		1.0	0



MDP Search Trees

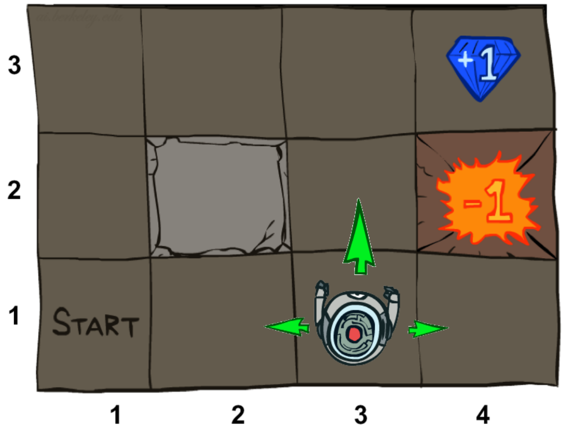
- MDP search tree can be viewed as an **expectimax** search tree



Discounting

Discounting

- Give less importance to reward / cost in the distant future
- There are several reasons to do so
  - When performing reinforcement learning (which will be covered in the next lecture), uncertainty accumulates over time, so it's less meaningful to optimize reward in the distant future
  - In many cases, we prioritize more recent reward



$100 right now

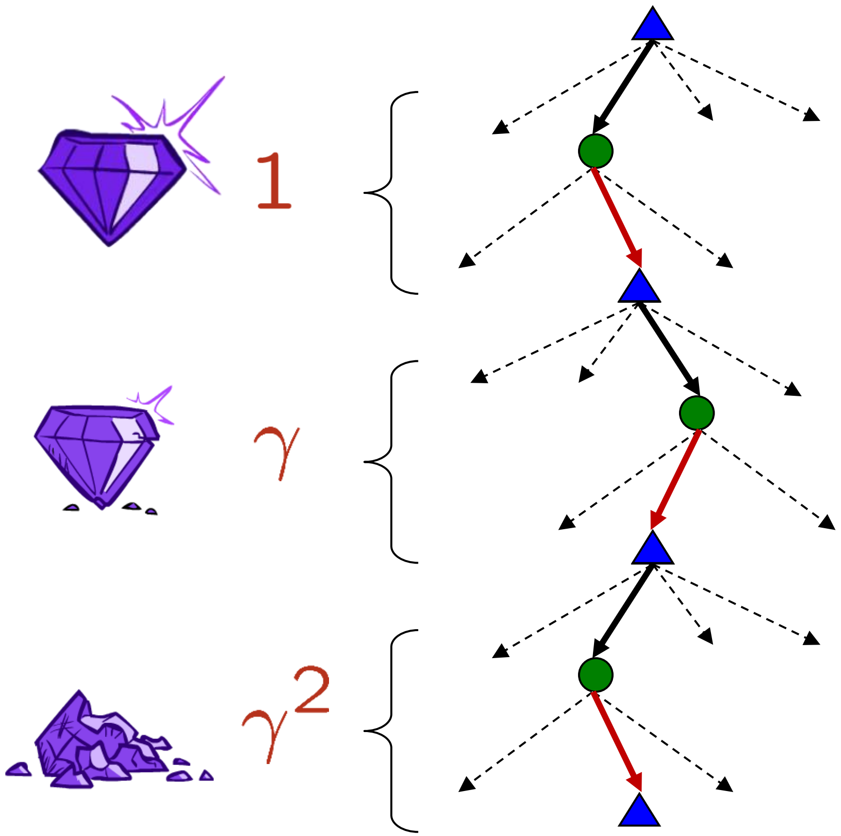
vs.

$110 next year

Discounting

$0 < \gamma < 1$

- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Example: discount of 0.9 $= \gamma$
  - U([1,2,3]) = 1*1 + 0.9*2 + 0.81*3
  - U([1,2,3]) < U([3,2,1])

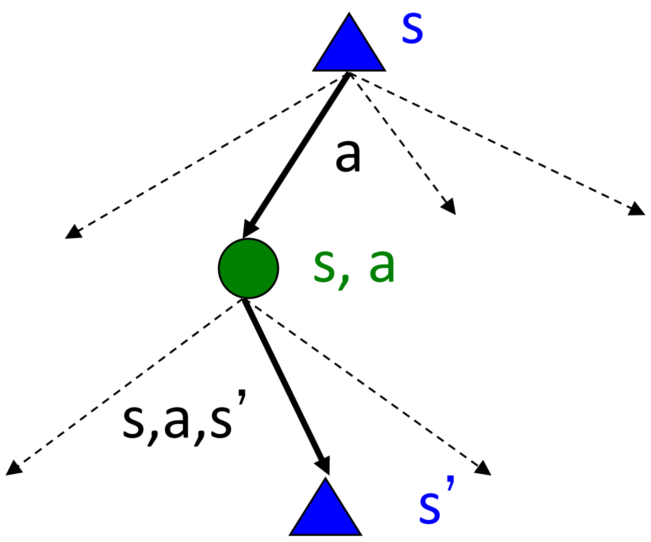


Value Functions and Optimal Policies

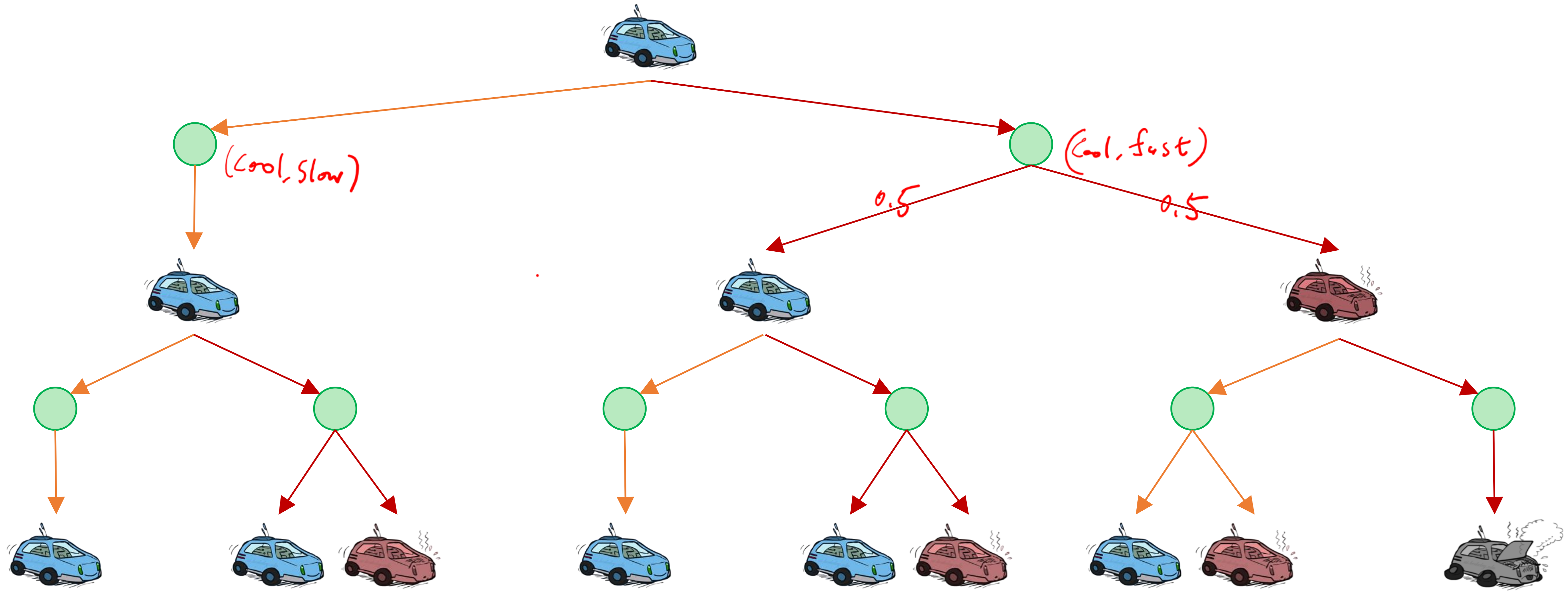
Recap: Defining MDPs

- 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)

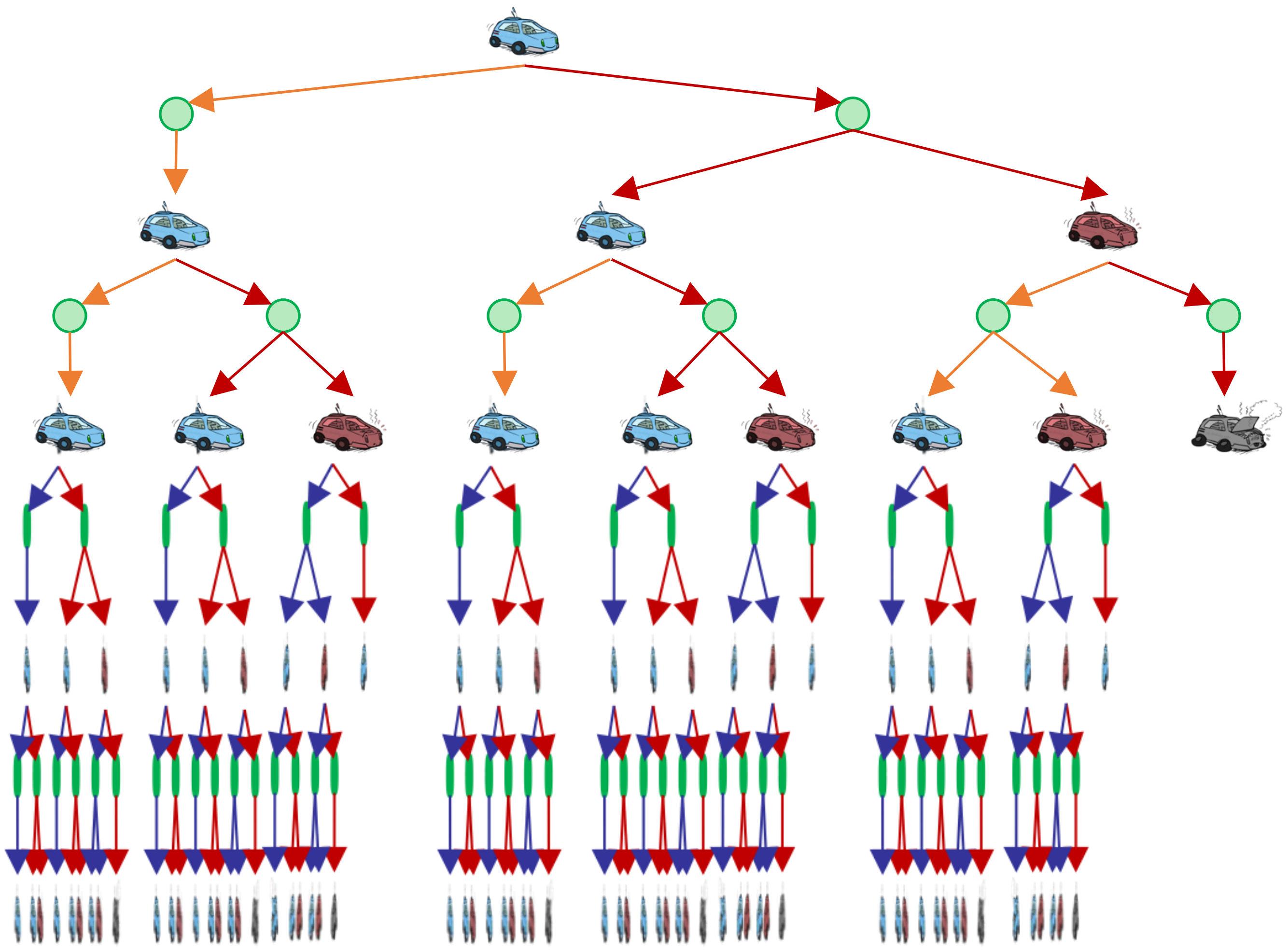
- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility or Return = sum of (discounted) rewards



Racing Search Tree

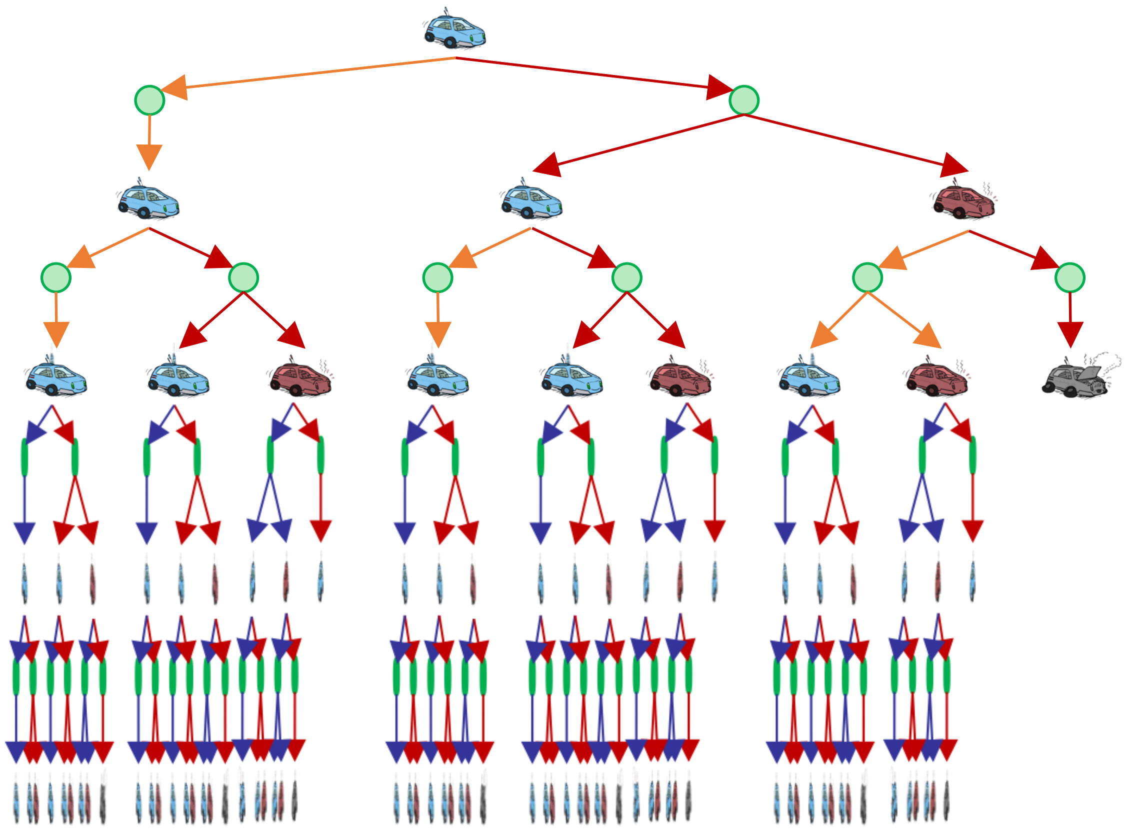


Racing Search Tree

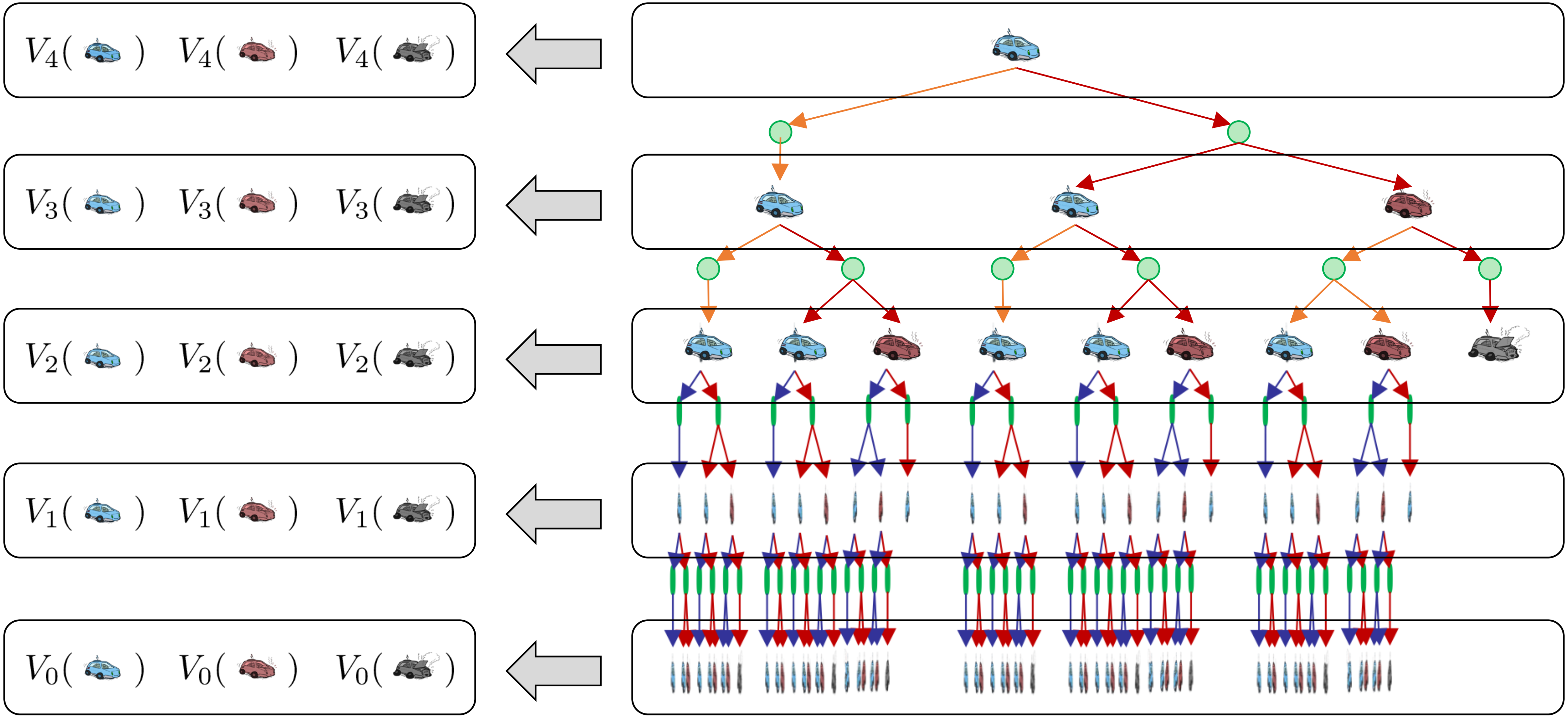


Racing Search Tree

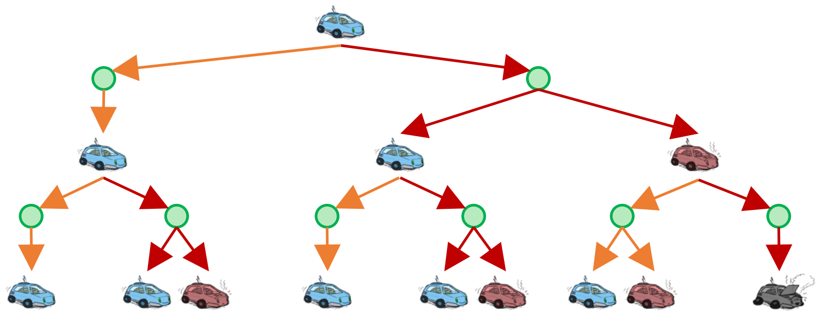
- Problem: States are repeated
  - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
  - Idea: Perform **depth-limited** computation with increasing depths until change is small
  - Note: deep parts of the tree eventually don't matter if $\gamma < 1$



Computing Time-Limited Values



Time-Limited Values



Define $V_k(s)$ to be the optimal value of s if the game ends in at most k more time steps

$$V_0(s) = 0$$

$$V_k(s) = \begin{cases} \max_a \left(\sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_{k-1}(s')) \right) & \text{if } s \text{ is not a terminal state} \\ \max_a \left(\sum_{s'} T(s,a,s') R(s,a,s') \right) & \text{if } s \text{ is a terminal state} \end{cases}$$

recursively for $k \geq 1$

Example

s	a	s'	T(s,a,s')	R(s,a,s')
	Slow		1.0	+1
	Fast		0.5	+2
	Fast		0.5	+2
	Slow		0.5	+1
	Slow		0.5	+1
	Fast		1.0	−10
	(end)		1.0	0

Assume no discount $(\gamma = 1)$

	cool 	warm 	overheated 
V_2	3.5	2.5	0
V_1	2	1	0
V_0	0	0	0

$V_1(\text{cool})$ If $a = \text{slow} \Rightarrow 1 + \gamma \cdot V_0(\text{cool}) = 1$

If $a = \text{fast} \Rightarrow 0.5(2 + \gamma V_0(\text{cool})) + 0.5(2 + \gamma V_0(\text{warm})) = 2$

$$V_k(s) = \begin{cases} \max_a \left(\sum_{s'} T(s,a,s')\left(R(s,a,s') + \gamma V_{k-1}(s')\right) \right) \\ \max_a \left(\sum_{s'} T(s,a,s') R(s,a,s') \right) \end{cases}$$

Slightly Simplifying the Notation

$$V_k(s) = \begin{cases} \max_a \left(\sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_{k-1}(s')) \right) & \text{if } s \text{ is not a terminal state} \\ \max_a \left(\sum_{s'} T(s,a,s')R(s,a,s') \right) & \text{if } s \text{ is a terminal state} \end{cases}$$

It is possible to write them as $V_k(s) = \max_a \left(\sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_{k-1}(s')) \right) \quad \forall s$

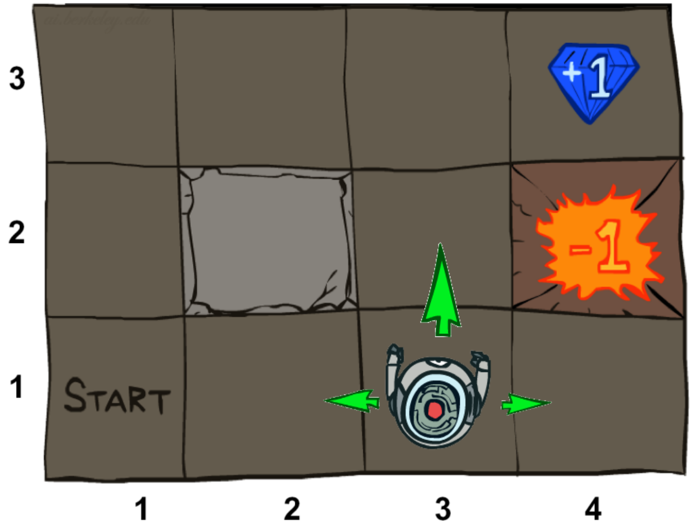
by creating an artificial s_{dum} state so that

$T(s_{\text{ter}}, a, s_{\text{dum}}) = 1$ for any terminal state s_{ter} and any action a
$T(s_{\text{dum}}, a, s_{\text{dum}}) = 1$ for any action a
$R(s_{\text{dum}}, a, s_{\text{dum}}) = 0$ for any action a

We did not have this matter when discussing about search because there we usually assume no reward from the terminal state.

Example

Two ways to incorporate the final reward.
Let s_{ter} be a terminal state, i.e., (4,2) or (4,3)



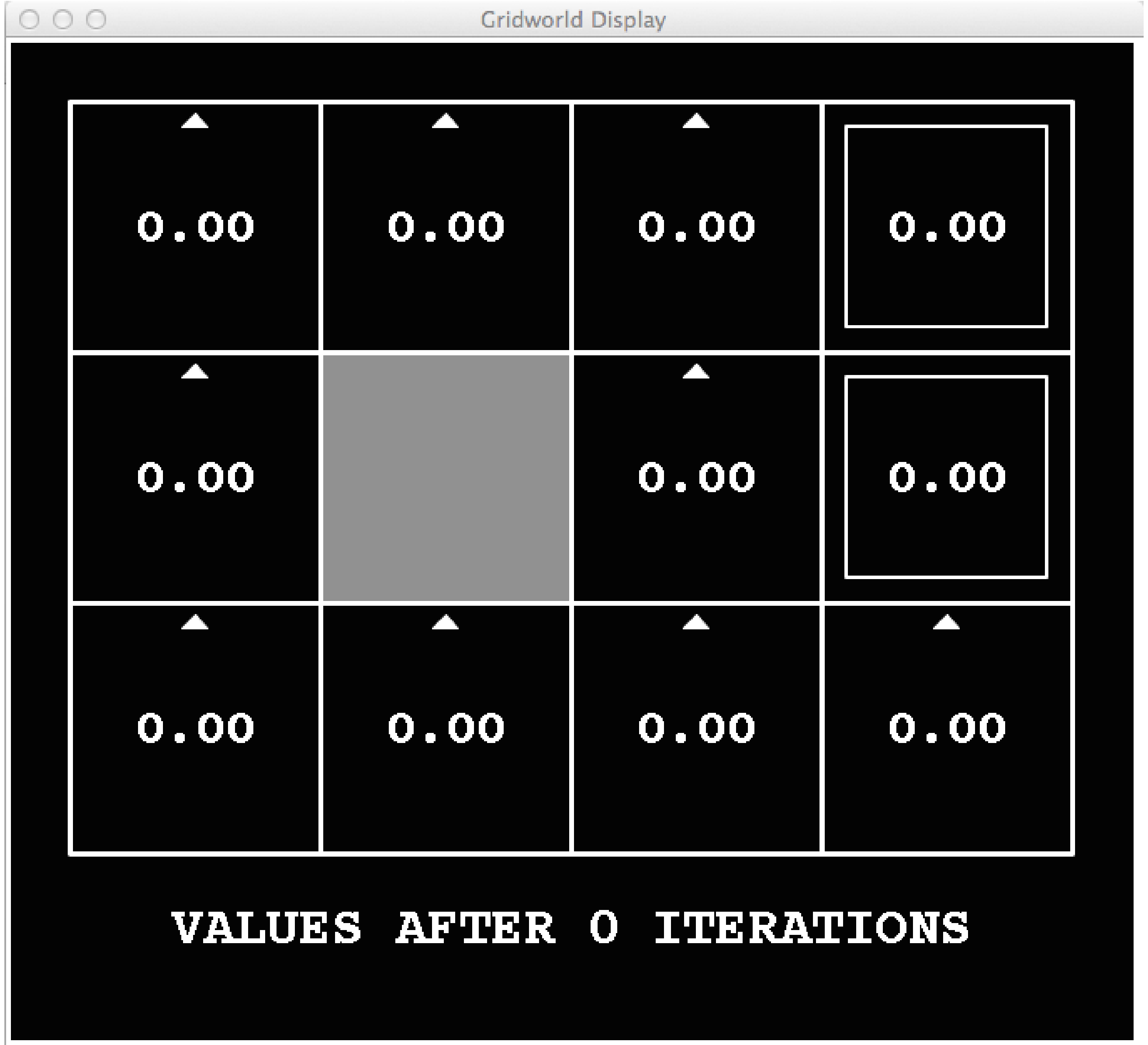
(1) $R(s, a, s_{\text{ter}}) = +1$ (or -1) $R(s_{ter}, a, s') = 0$

$$V_k(s) = \begin{cases} \max_a \left(\sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_{k-1}(s')) \right) & \text{if } s \text{ is not a terminal state} \\ \max_a \left(\sum_{s'} T(s,a,s') R(s,a,s') \right) & \text{if } s \text{ is a terminal state} \end{cases}$$

(2) $R(s_{\text{ter}}, a, s_{\text{dum}}) = +1$ (or -1) (Needs to create a dummy state)

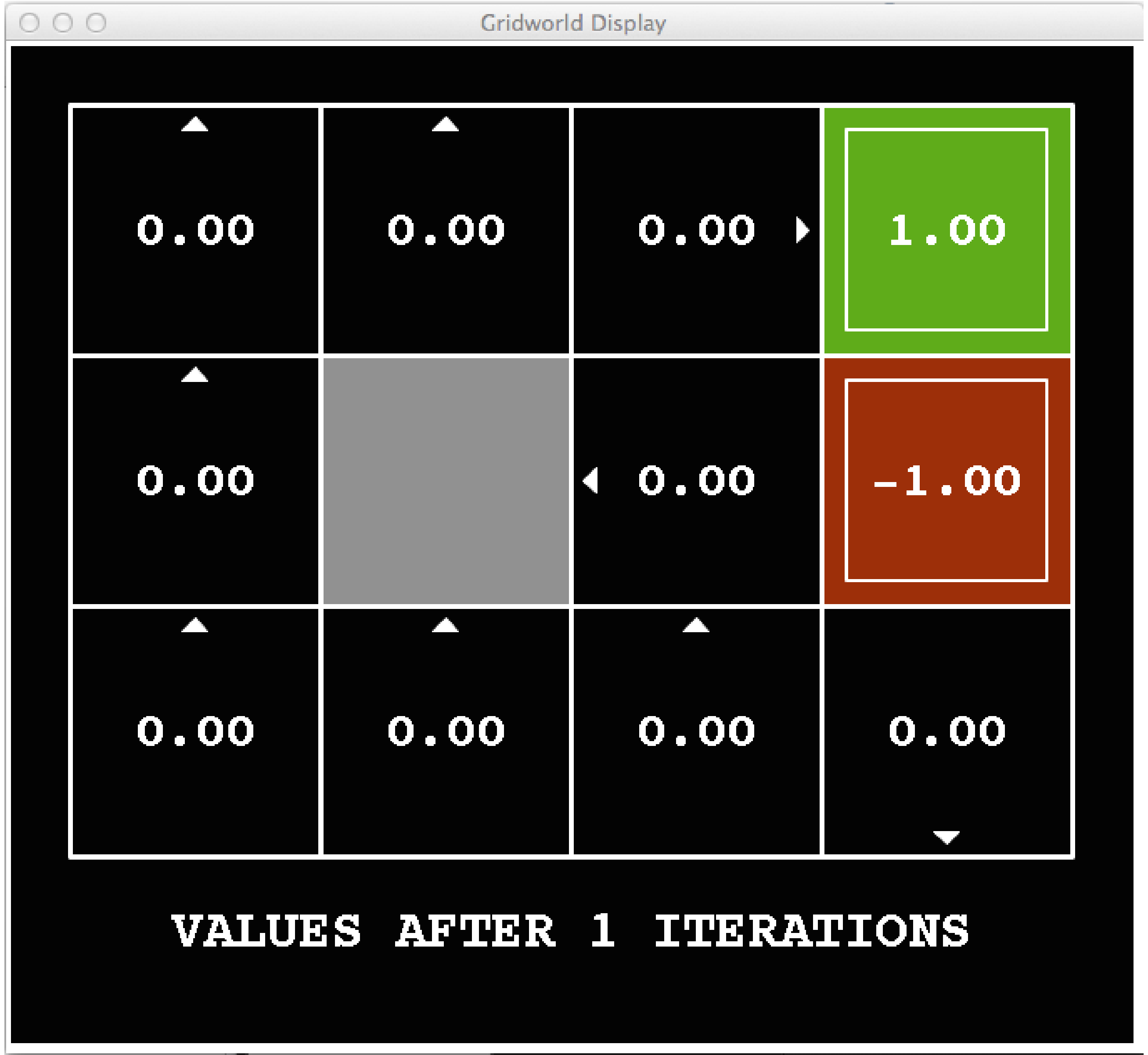
$$V_k(s) = \max_a \left(\sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_{k-1}(s')) \right) \quad \forall s$$

k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

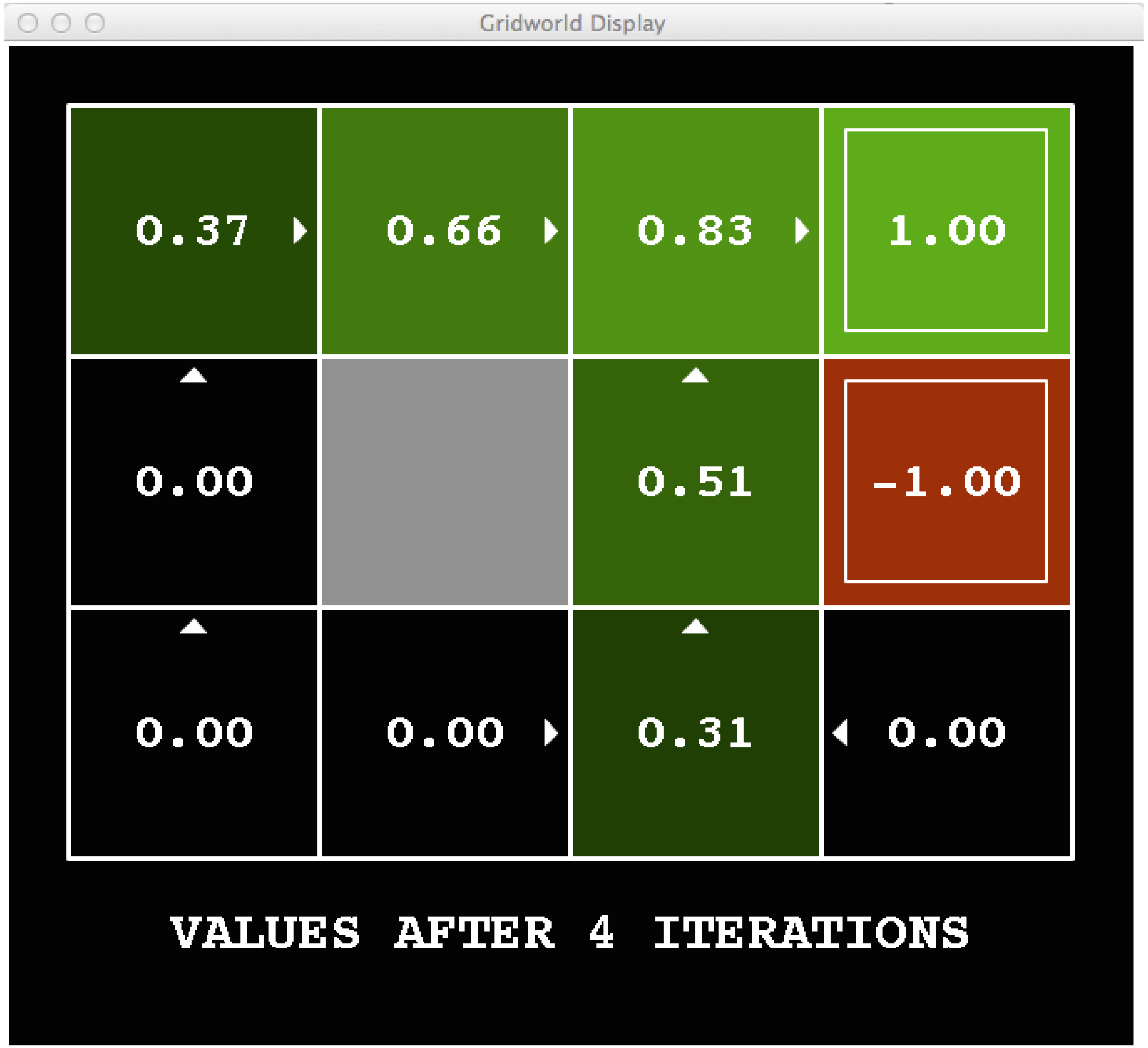
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

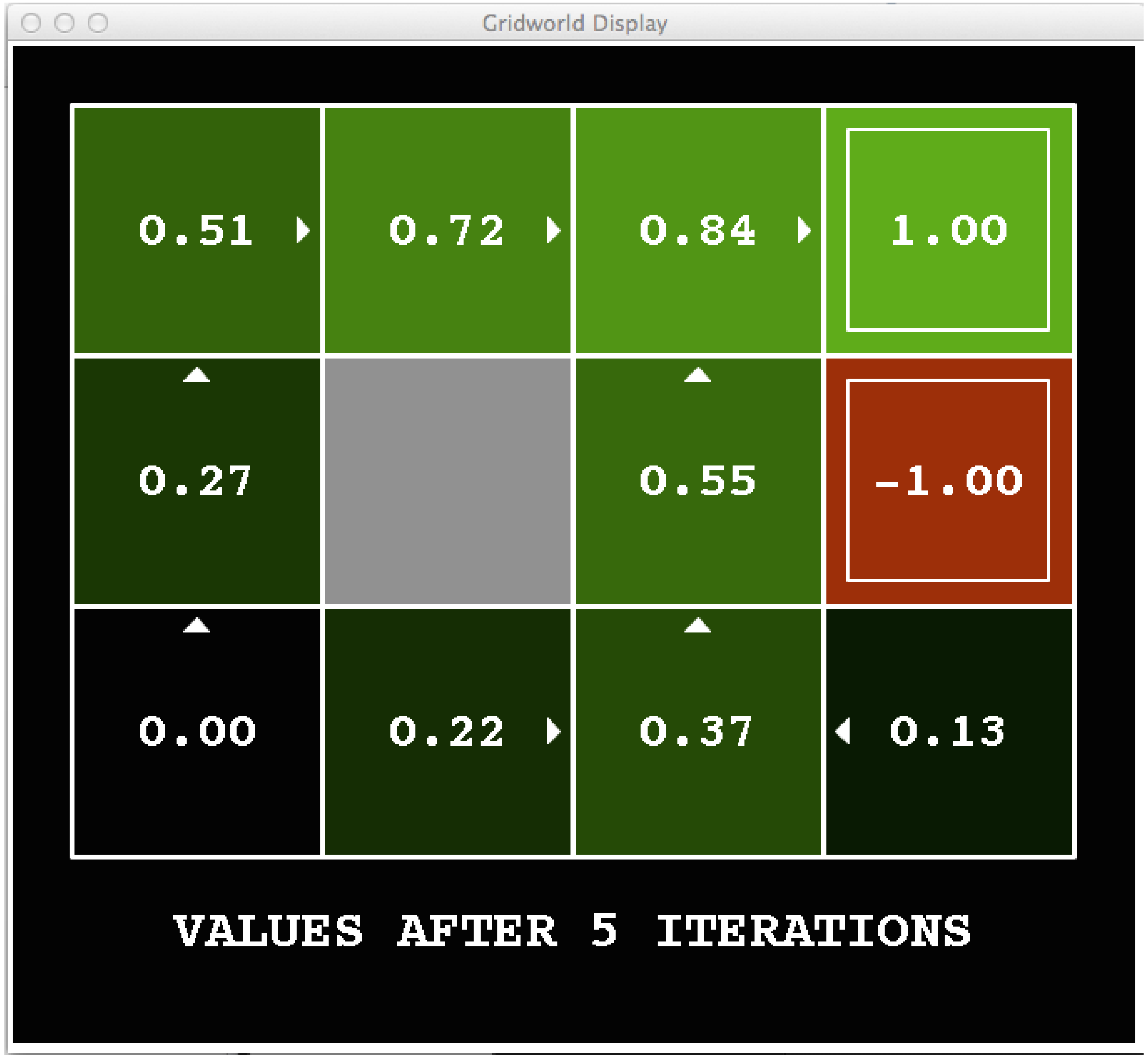
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



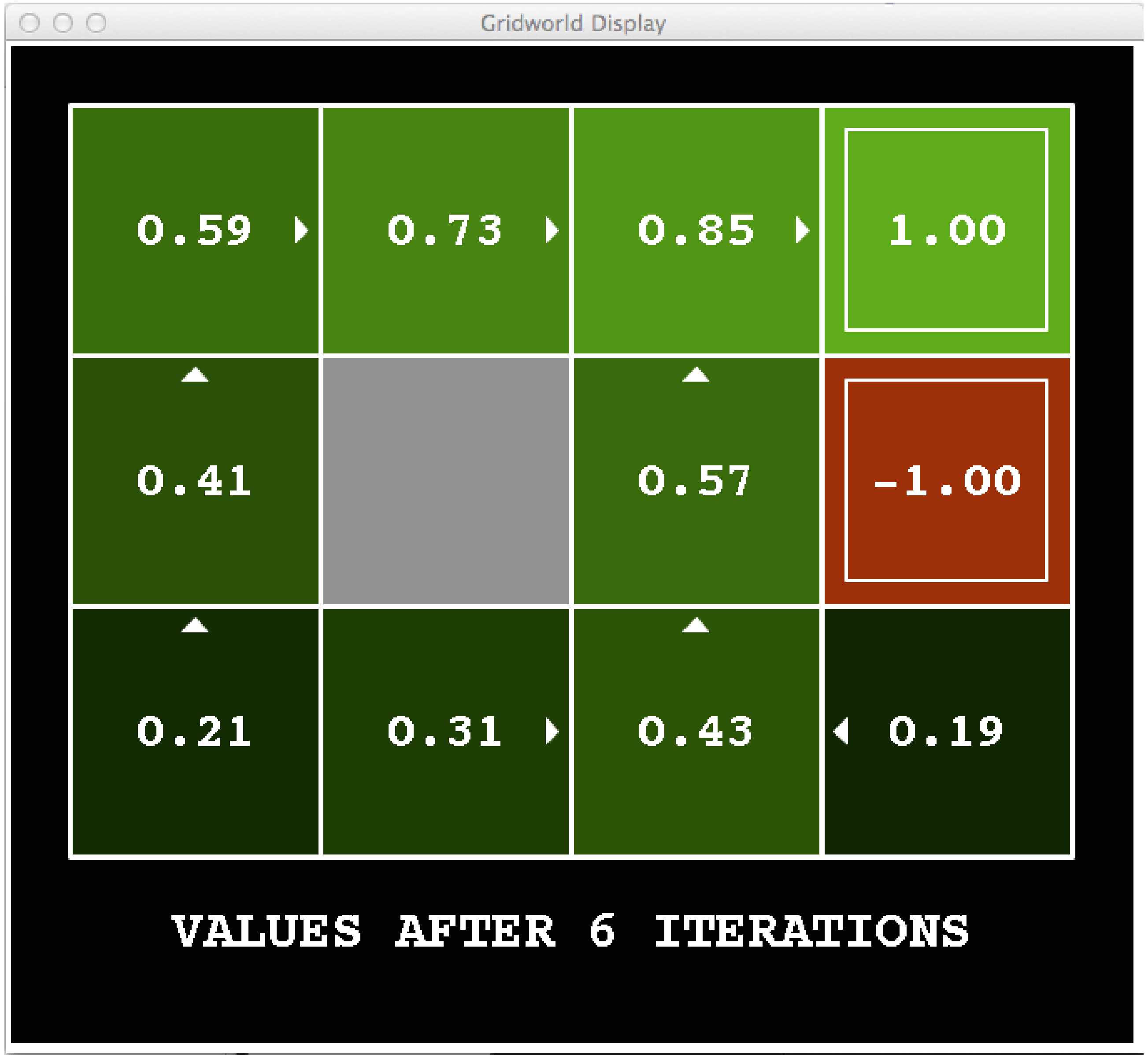
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



Noise = 0.2
Discount = 0.9
Living reward = 0

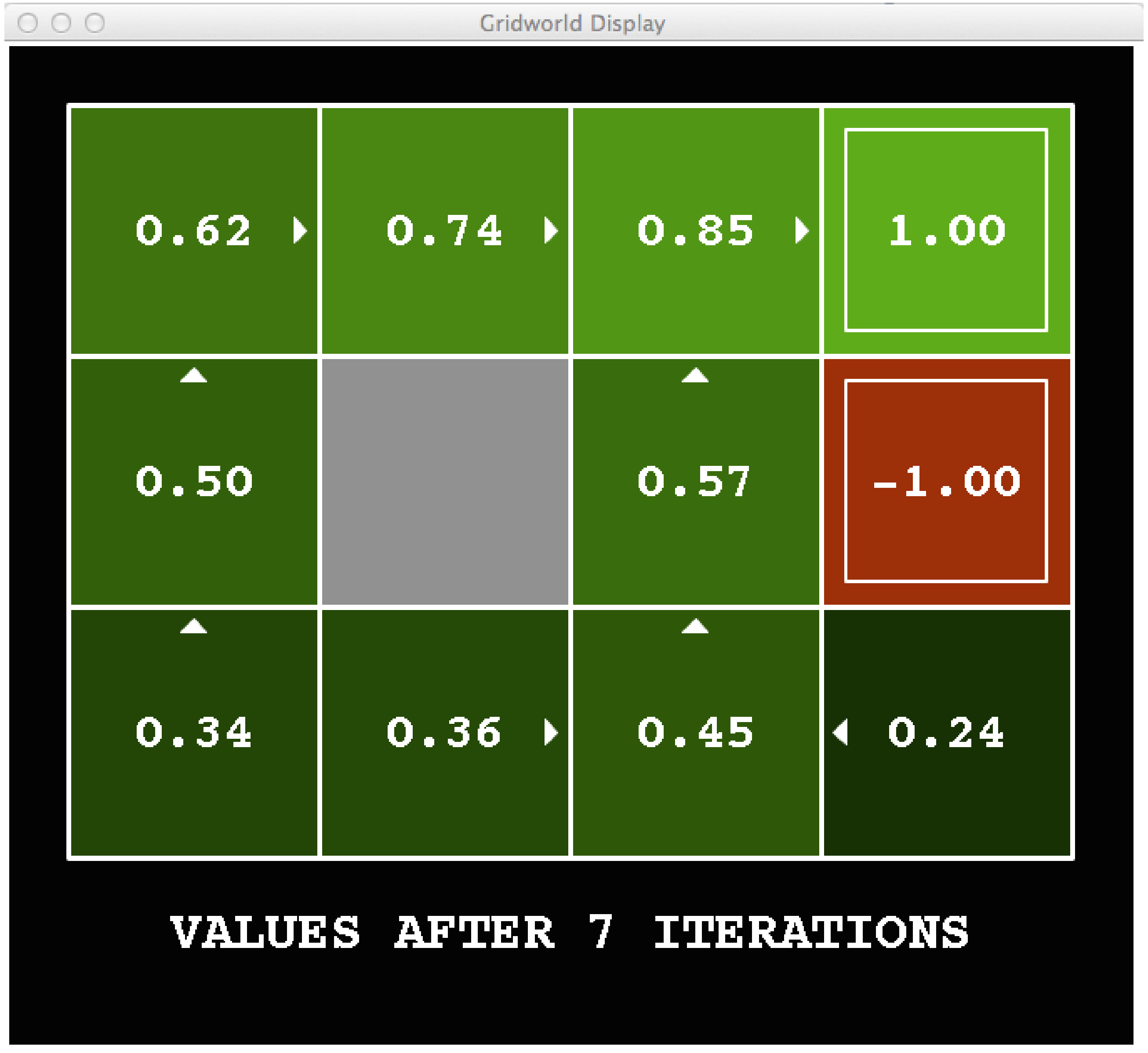
k=6



V_6

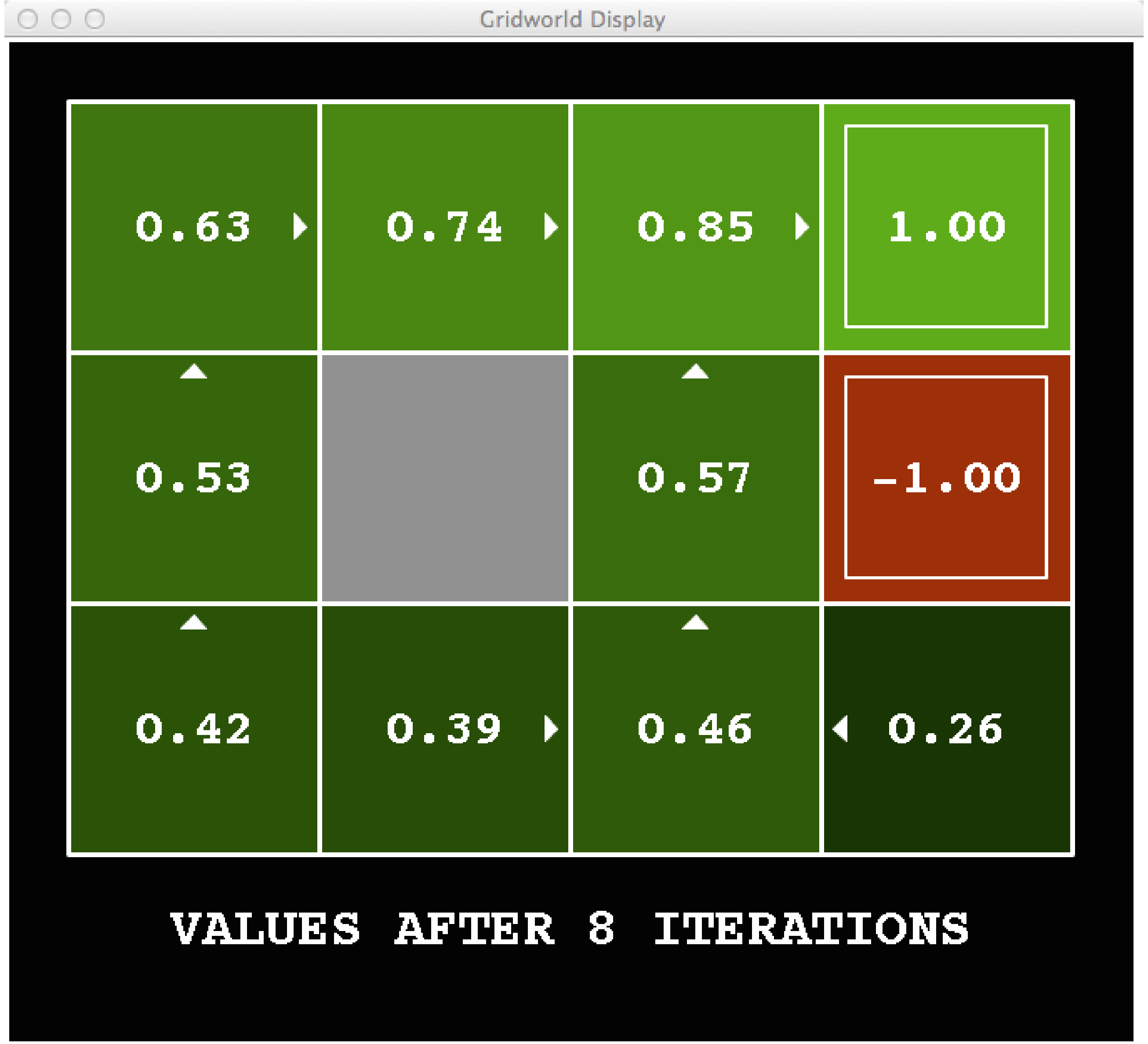
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



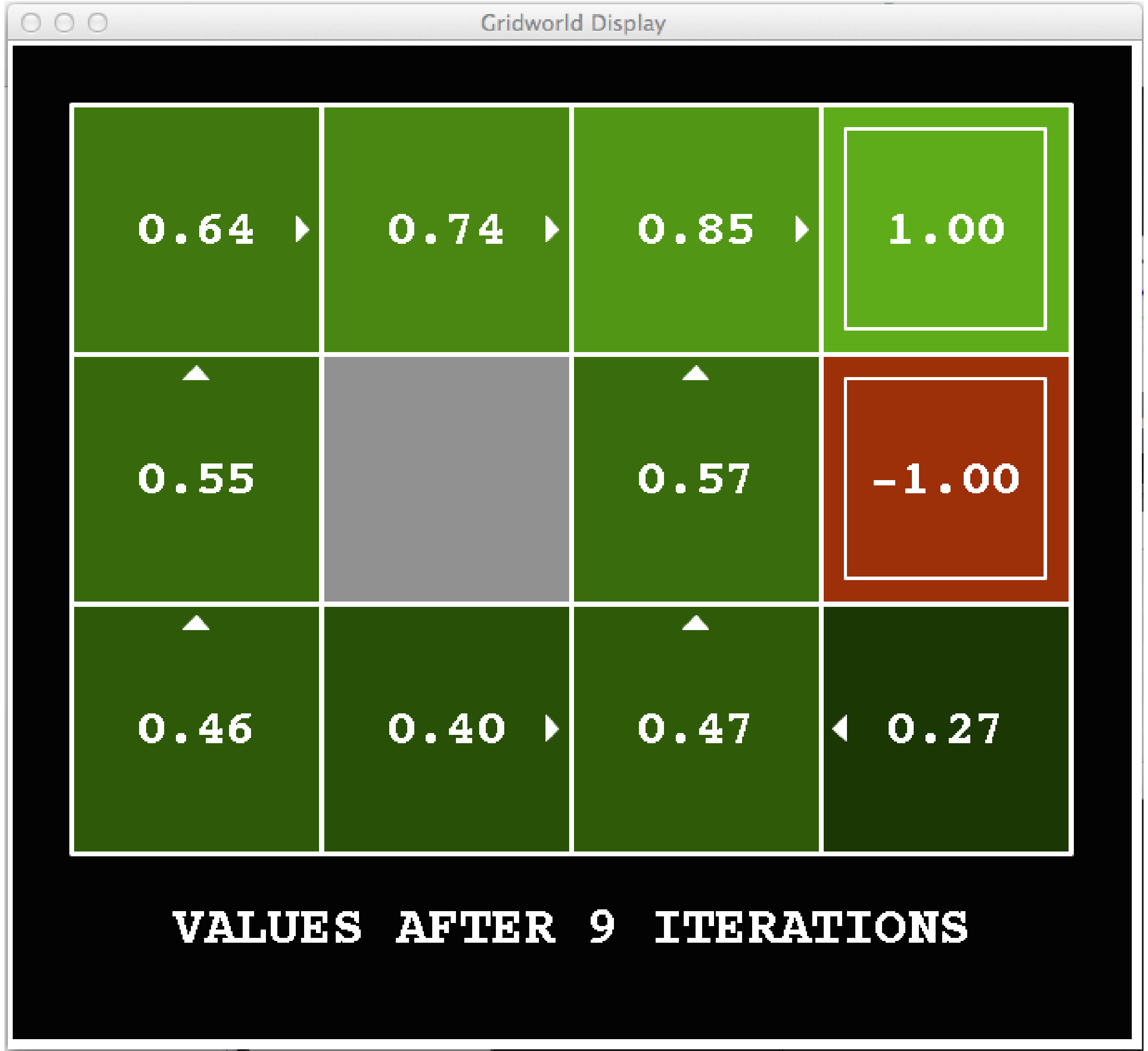
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



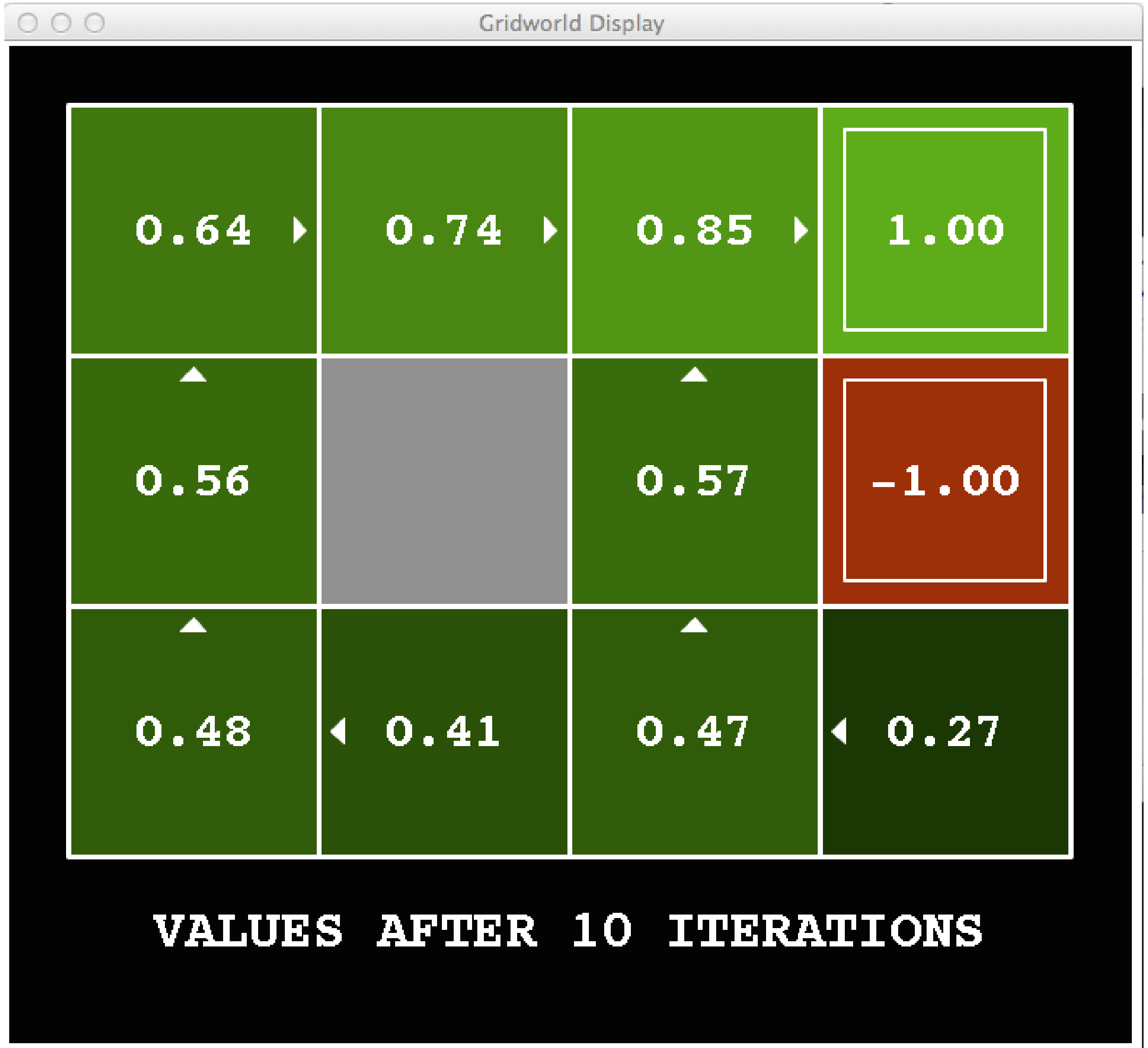
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



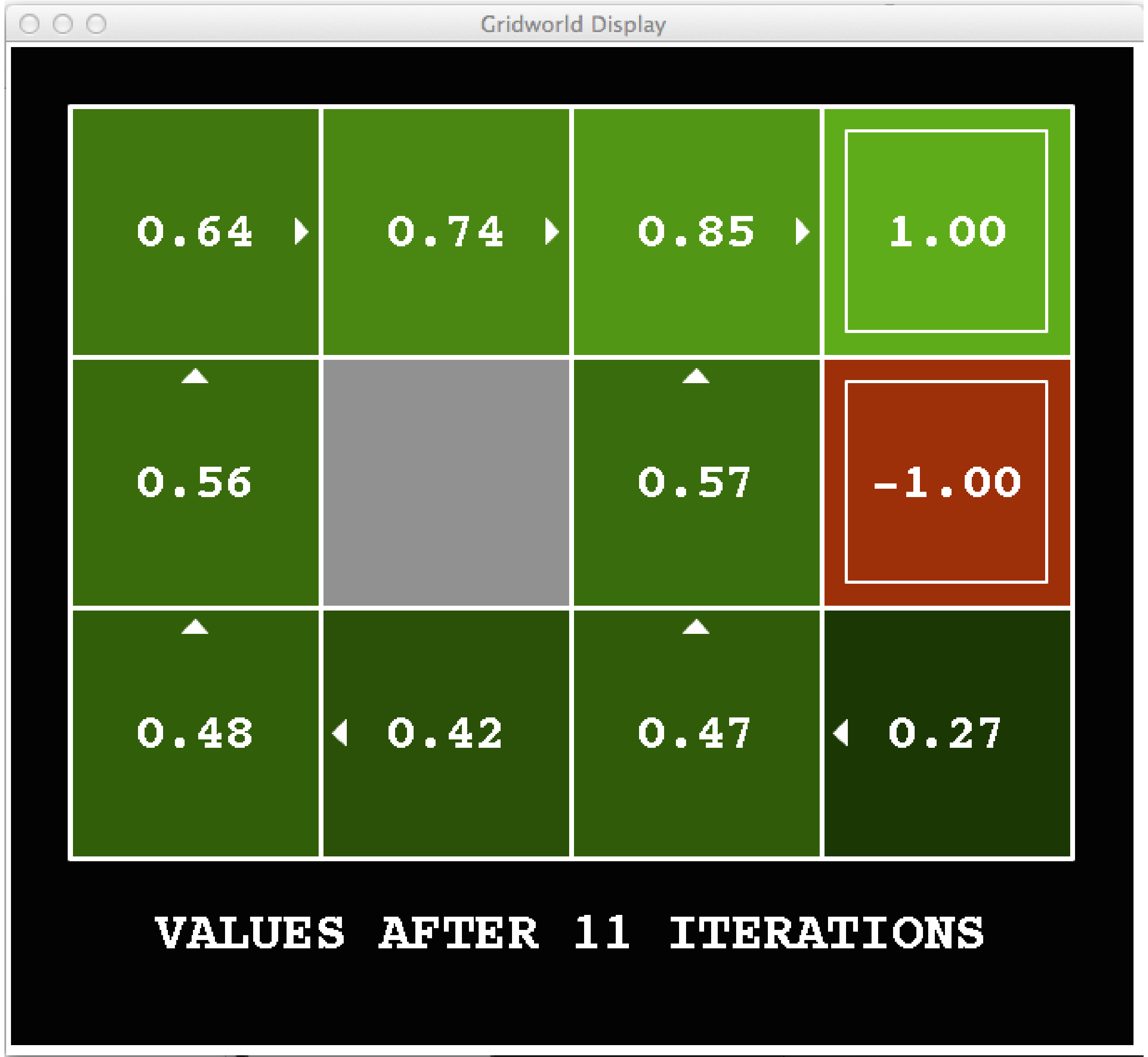
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



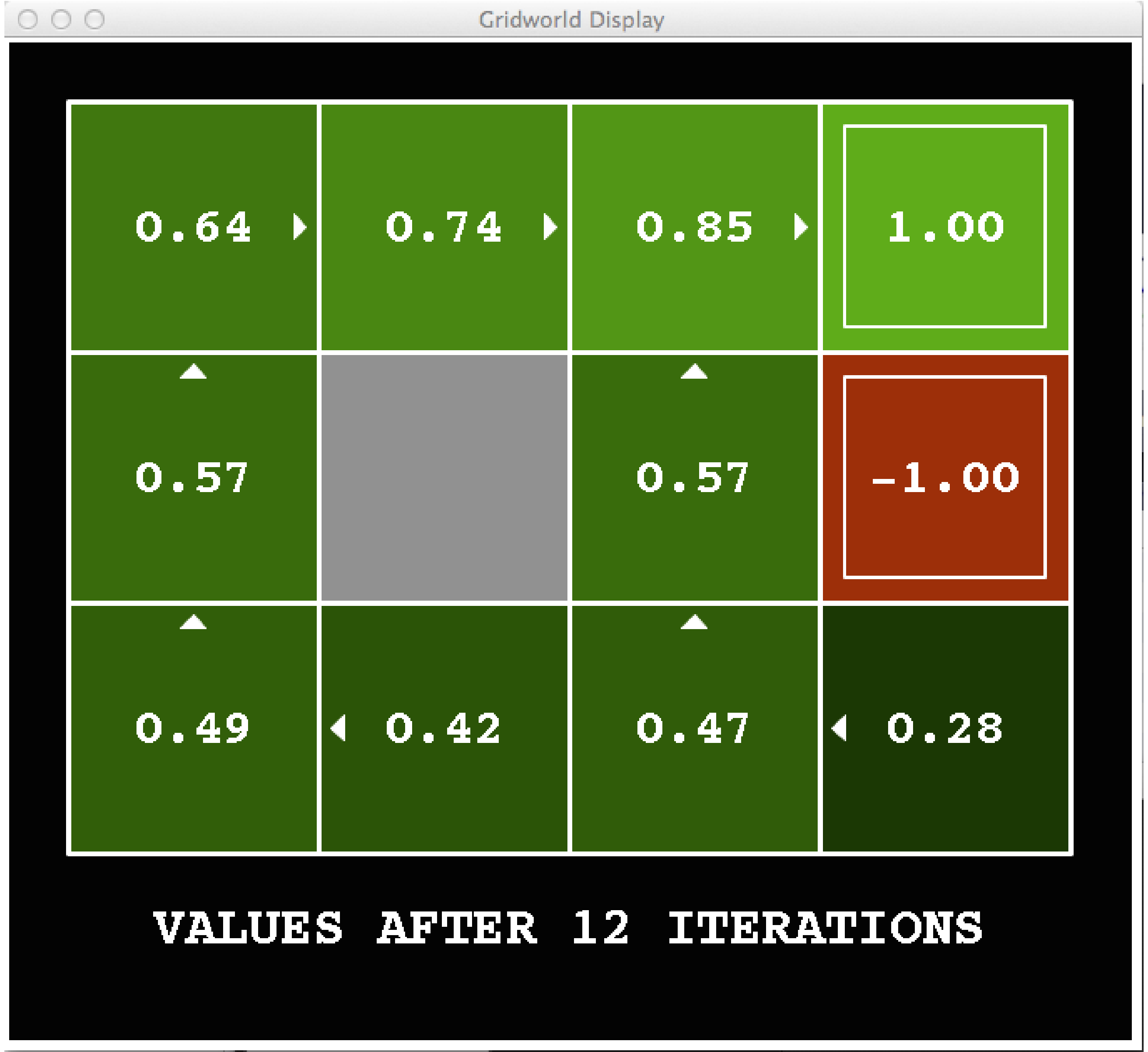
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



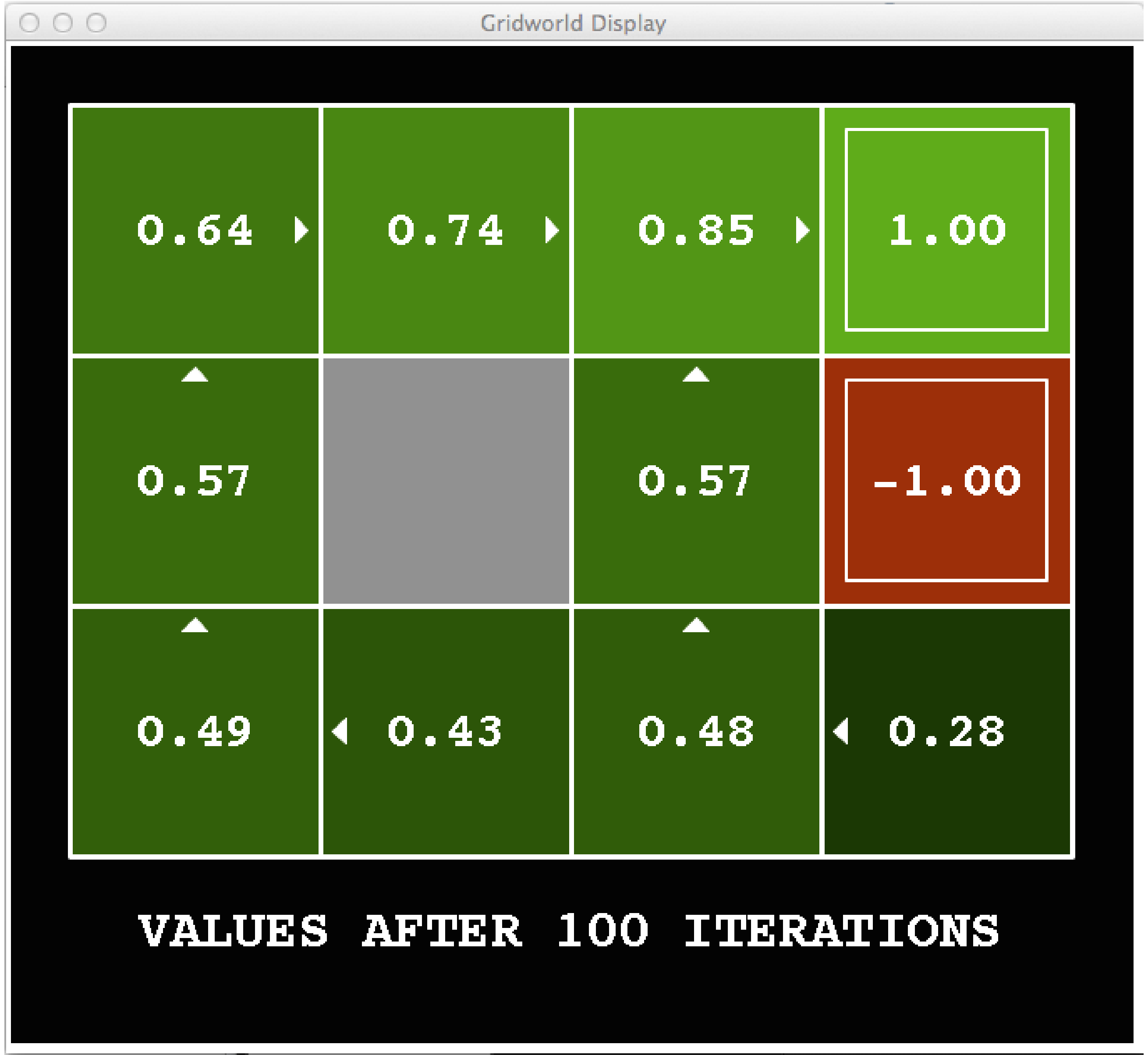
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

State Value (V Value) and State-Action Value (Q Value)

$$V_0(s) = 0$$

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

|| $Q(s,a)$

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

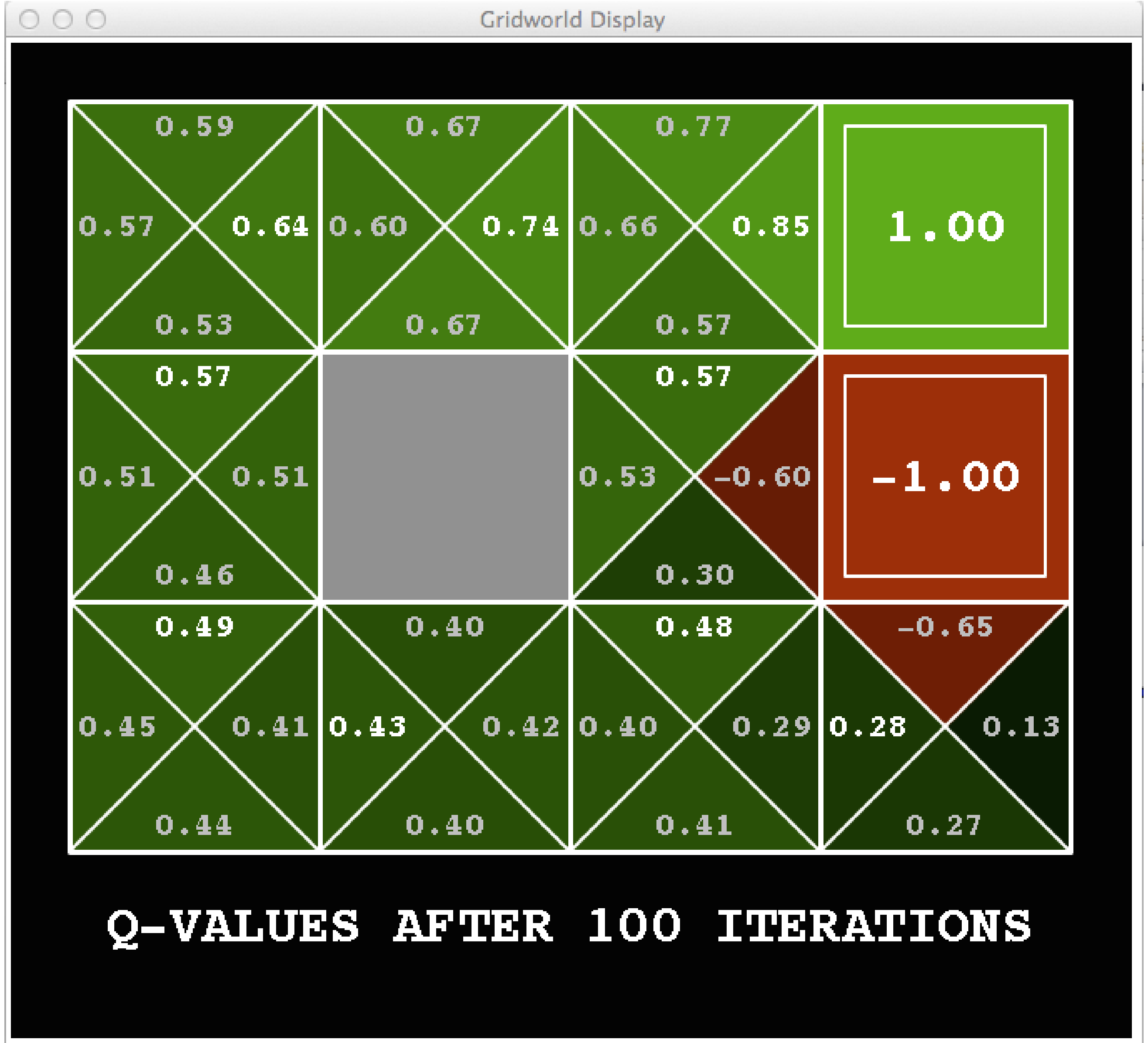
$$V_k(s) = \max_a Q_k(s,a)$$

total / discounted reward

$Q_k(s,a)$ = The optimal value from s if **taking action a in the first step** and then perform optimally in the remaining $k-1$ steps.

$\pi_k(s) = \arg\max_a Q_k(s,a)$

Q Values

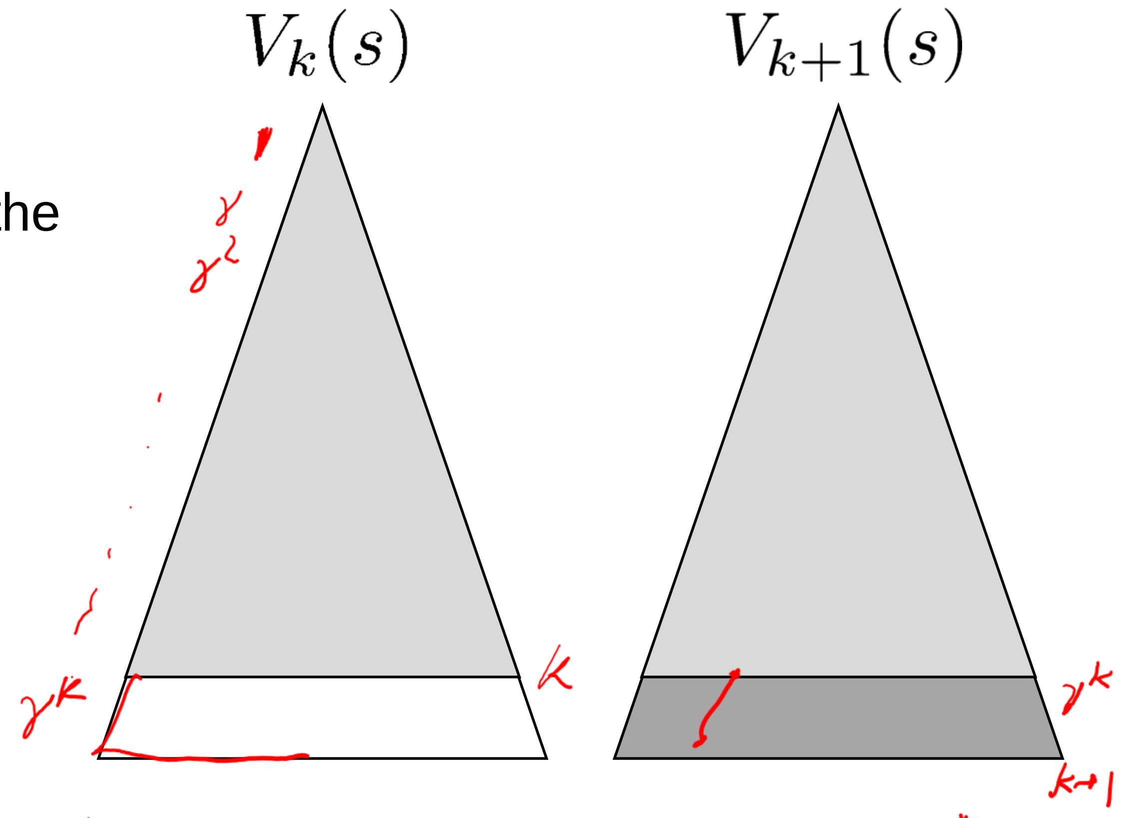


Noise = 0.2
Discount = 0.9
Living reward = 0

Convergence

- Are V_k going to converge?

- If the discount is less than 1
  - The difference between V_k and V_{k+1} is that on the bottom layer, V_{k+1} has actual rewards while V_k has zeros
  - That one-step reward ranges in [-R, R] where R = max |R(s,a,s')|
  - But everything is discounted by γ^k
  - So V_k and V_{k+1} are at most γ^k max|R| different
  - So as k increases, the values converge



$|V_k(s) - V_{k+1}(s)| \leq \gamma^k \max|R|$

$\lim_{k \to \infty} V_k(s) \to V^*(s)$

Value Iteration

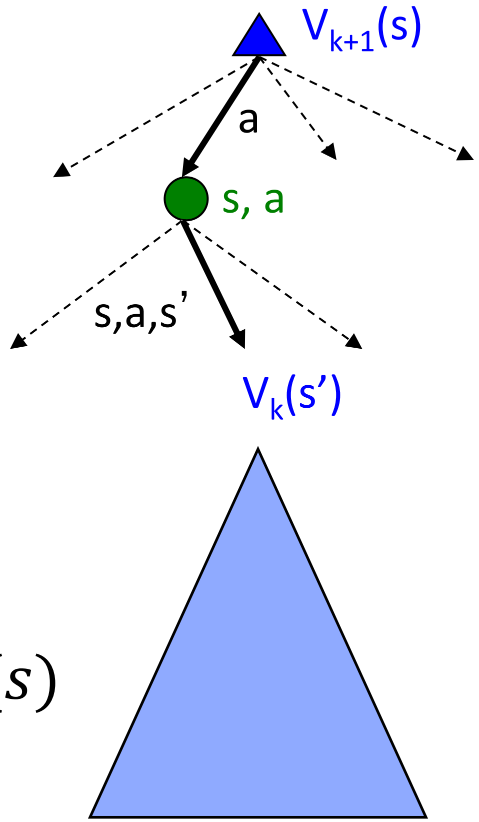
$V^*(s) = V_\infty(s)$

- Start with $V_0(s) = 0$
- Given $V_{k-1}(s)$, perform the following update **for all state s and action a**:

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

$$V_k(s) \leftarrow \max_a Q_k(s,a)$$

- Repeat until convergence: $|V_{k+1}(s) - V_k(s)| \leq \epsilon$ for all s
- (Near) optimal policy: $\pi(s) = \operatorname{argmax}_a Q_k(s,a)$
- **Theorem:** will converge to unique optimal values $V_k(s) \to V^\star(s)$



The Limits of Value Iteration

- The state value function:
  - $V^\star(s)$ = expected **discounted total reward** starting from s and acting optimally

- The state-action value function:
  - $Q^\star(s,a)$ = expected **discounted total reward** starting by taking action a from state s and (thereafter) acting optimally
  - $Q^\star(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^\star(s'))$

- The optimal policy (that maximizes the discounted total reward)
  - $\pi^\star(s)$ = optimal action from state s $= \underset{a}{\operatorname{argmax}}\, Q^\star(s,a)$

Bellman Equation

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

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

As discussed previously, given T and R, one can approximate $Q^\star$ and $V^\star$ that satisfy the Bellman equation through **value iteration**.

This set of equations is an instance of **dynamic programming** (but probably slightly more advanced than what you learned in DSA because it could involve infinite depth)

Q-Learning

(Machine Learning in an MDP)

Recall how we compute the optimal policy in MDPs

Value Iteration

$V_0(s) \leftarrow 0 \quad \forall s$

For $k = 1, 2, \ldots$

Require knowledge about the model

$$Q_k(s,a) \leftarrow \sum_{s'} \boxed{T(s,a,s')} [\boxed{R(s,a,s')} + \underbrace{\gamma V_{k-1}(s')}_{=0 \text{ if } s \text{ is terminal}}] \quad \forall s, a$$

$$V_k(s) \leftarrow \max_a Q_k(s,a) \quad \forall s$$

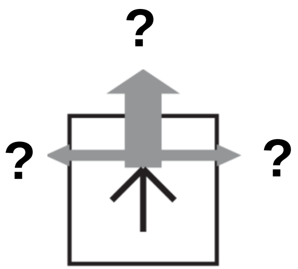
If $|V_k(s) - V_{k-1}(s)| \leq \epsilon$ for all s:

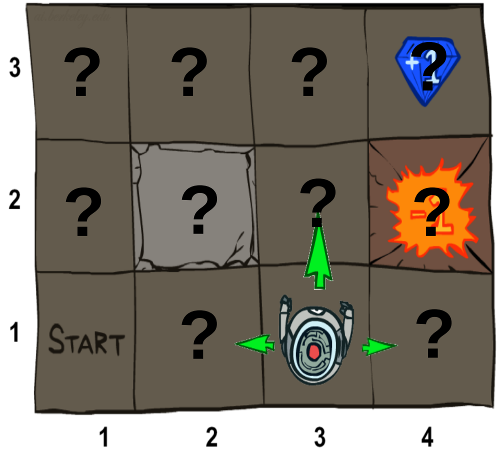
Let $\hat{Q}(s,a) = Q_k(s,a) \;\; \forall s, a$

break

Return policy $\hat{\pi}(s) = \underset{a}{\operatorname{argmax}} \, \hat{Q}(s,a)$

What if we don't know the transition T or the reward R?





Solutions when we don’t know the model

- We want to perform the update

$$Q_k(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_{k-1}(s')] \quad \forall s,a$$

- But we don’t know T and R
- Fortunately, we can get a “sample” for the right-hand side
  - Suppose that we are on some state $\hat{s}$ and we take an action $\hat{a}$
  - The environment will generate next state $\hat{s}'$ and reveal the reward $\hat{R} = R(\hat{s},\hat{a},\hat{s}')$
  - Then we have

$$\mathbb{E}_{\hat{R},\hat{s}'}\left[\hat{R} + \gamma V_{k-1}(\hat{s}')\right] = \sum_{s'} T(\hat{s},\hat{a},s')[R(\hat{s},\hat{a},s') + \gamma V_{k-1}(s')]$$

  - But we cannot simply do $Q_k(\hat{s},\hat{a}) \leftarrow \hat{R} + \gamma V_{k-1}(\hat{s}')$ … why?

Q-Learning

$V_0(s) \leftarrow 0,\ Q_0(s,a) \leftarrow 0 \quad \forall s,a$

Let s_1 be the initial state.

For $k = 1, 2, \ldots$

Take action a_k. Observe next state s_{k+1} and reward $R_k = R(s_k, a_k, s_{k+1})$.

// Slightly modify the values on the visited state-action pair (s_k, a_k):

$Q_k(s_k, a_k) \leftarrow (1-\eta_k)\, Q_{k-1}(s_k, a_k) + \eta_k\, [R_k + \gamma V_{k-1}(s_{k+1})]$ $\eta_k \in (0,1)$: learning rate

$\eta_k \approx 0.001$

$V_k(s_k) \leftarrow \max_a Q_k(s_k, a)$

=0 if s_k is terminal

// Keep other values unchanged:

$Q_k(s,a) \leftarrow Q_{k-1}(s,a)$ and $V_k(s) \leftarrow V_{k-1}(s)$ for $(s,a) \neq (s_k, a_k)$

If s_k is a terminal state:

Reset s_{k+1} to be the initial state.

Continue

Q-Learning

The update

$$Q_k(s_k, a_k) \leftarrow (1 - \eta_k)\, Q_{k-1}(s_k, a_k) + \eta_k\, [R_k + \gamma V_{k-1}(s_{k+1})]$$

has the effect of averaging up multiple samples of $R_k + \gamma V_{k-1}(s_{k+1})$

(so mitigate the effect of randomness)

0, 0, 0, 1, 0, 1, 1, 0, 1, ...

$Q = 0$

$Q \leftarrow (1-\eta) Q + \eta \times 0$

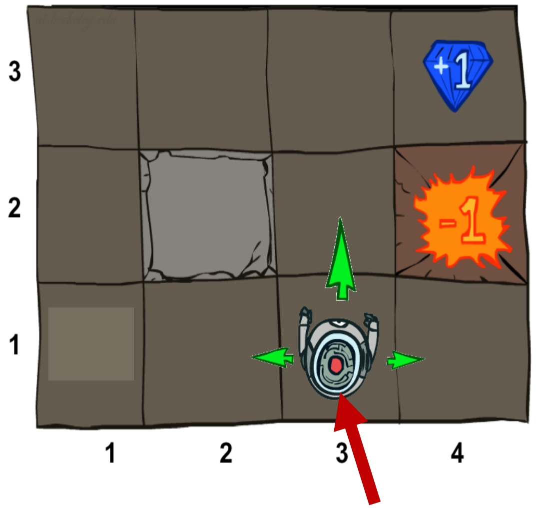
$Q \leftarrow (1-\eta) Q + \eta \times 0$

⋮

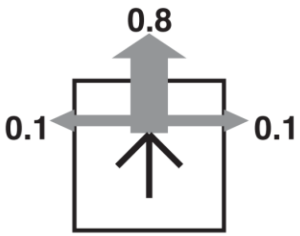
Deep Q-Learning

- Instead of recording $Q(s, a)$ for each individual s, a, use a neural network (NN) to model the mapping (NN input: s, a, NN output: $Q(s, a) \in \mathbb{R}$)
- Notable applications:
  - Playing Atari games: https://www.youtube.com/watch?v=rFwQDDbYTm4

Q-Learning Example



starting state = (3,1)



Initial state: $(3,1)$

Terminal states: $(4,2)$ and $(4,3)$

Actions: NSEW

The learner doesn't know these!

Reward:

$R(s,a,s') = R(s) = -0.2$ for all non-terminal s

$R(s) = -1$ if $s = (4,2)$

$R(s) = +1$ if $s = (4,3)$

Transition:

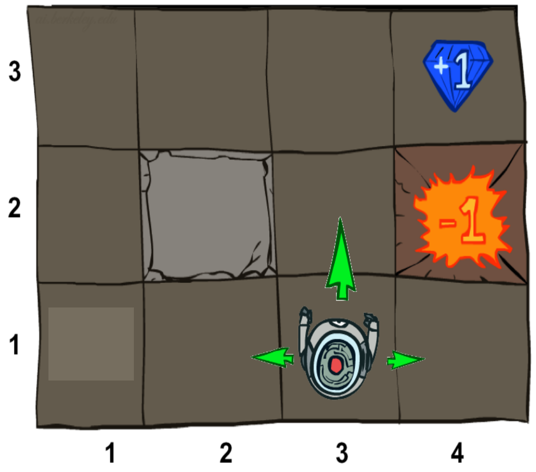
with probability 0.8: transition according to the action

with probability 0.2: transition to two sides (see left figure)

If wall is met, stay in the original square

Discount factor $\gamma = 0.95$

Q-Learning Example



Iteration 1

$s_1 = (3,1)$

Learner take action $a_1 = \text{N}$

Environment sample next state $s_2 = (3,2)$ and reveal reward -0.2

Learner update

$$Q\big((3,1), \text{N}\big) = (1-\eta)Q\big((3,1), \text{N}\big) + \eta[-0.2 + \gamma V\big((3,2)\big)]$$

$$= 0.9 \times 0 + 0.1[-0.2 + 0.95 \times 0] = -0.02$$

$$V\big((3,1)\big) = \max_a Q\big((3,1), a\big) = 0$$

// $Q(s, a)$, $V(s)$ for other s, a remains unchanged.

Iteration 2

Learner take action a_2 = S

Environment sample next state $s_3 = (4,2)$ and reveal reward -0.2

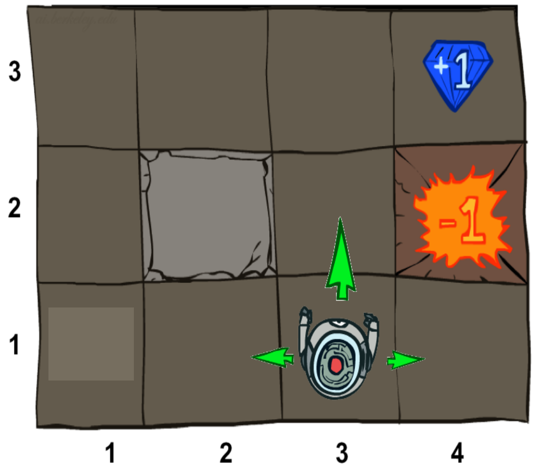
Learner update

$$Q\big((3,2), \text{S}\big) = (1-\eta)Q\big((3,2), \text{S}\big) + \eta[-0.2 + \gamma V\big((4,2)\big)]$$

$$= 0.9 \times 0 + 0.1[-0.2 + 0.95 \times 0] = -0.02$$

$$V\big((3,2)\big) = \max_a Q\big((3,2), a\big) = 0$$

Q-Learning Example



Learner take action $a_3 = \text{W}$

Since $s_3 = (4,2)$ is a terminal state, there is no next state.
Environment reveal reward -1.

Learner update

$$Q\big((4,2),\text{W}\big) = (1-\eta)Q\big((4,2),\text{W}\big) + \eta[-1]$$
$$= 0.9 \times 0 + 0.1[-1] = -0.1$$
$$V\big((4,2)\big) = \max_a Q\big((4,2),a\big) = 0$$

Iteration 3

Restart at $s_4 = (3,1)$ // but the Q, V values continue to update

Learner take action $a_4 = \text{S}$

Environment sample next state $s_5 = (3,2)$ and reveal reward -0.2

Learner update

$$Q\big((3,2),\text{S}\big) = (1-\eta)Q\big((3,2),\text{S}\big) + \eta[-0.2 + 0.9V((4,2))]$$
$$= 0.9 \times -\mathbf{0.02} + 0.1[-0.2 + 0.9 \times 0]$$
$$V\big((3,2)\big) = \max_a Q\big((3,2),a\big) = 0$$

Iteration 4

Q-Learning

Common strategies to pick actions:

- ϵ-Greedy:

$$a_k = \begin{cases} \underset{a}{\operatorname{argmax}}\ Q_{k-1}(s_k, a) & \text{with probability } 1-\epsilon \\ \text{random} & \text{with probability } \epsilon \end{cases}$$

- Boltzmann exploration: sample a_k from the distribution

$$\frac{\exp(Q_{k-1}(s_k,a))}{\sum_{a'} \exp(Q_{k-1}(s_k,a'))}$$

Idea: balancing **exploration** and **exploitation**

Randomly try some new actions (→ exploration)

Try to perform well (get high reward) using the current estimation (→ exploitation)

Theorem

- If every state-action pair is visited infinitely often (which requires exploration), with properly chosen learning rate scheduling η_k, then $\lim_{k \to \infty} Q_k(s, a) = Q^\star(s, a) \quad \forall s, a$

Summary

- Markov Decision Process formulates a search problem (finding a path that maximize the total reward) that has random state transition
- We can use value iteration (a dynamic programming algorithm) to find
  - State-action value function $Q^\star(s, a)$
  - State value function $V^\star(s)$

  The optimal policy is then given by $\pi^\star(s) = \underset{a}{\operatorname{argmax}}\, Q^\star(s, a)$
- Reinforcement Learning estimates the model (machine learning) through interacting with the MDP (search)
- Q-learning $\approx$ value iteration with samples and soft updates

Homework 6

- Choices problems: deadline 12/8 11:59PM
- Programming problem: deadline 12/18 11:59PM
  - Value iteration and Q-learning
- No late submission

Next Lecture

A review for the materials after the midterm