STEVENS
INSTITUTE OF TECHNOLOGY
1870

Artificial Intelligence

Computer Science, CS541 - A

Jonggi Hong

Announcements

- HW #4
  - Has been released! Due 11:59 pm, Tuesday, April 25.
- Grades of the midterm project report will be released soon.
- Homework for extra credit
  - Will be released next week
  - Not mandatory

STEVENS
INSTITUTE OF TECHNOLOGY
1870

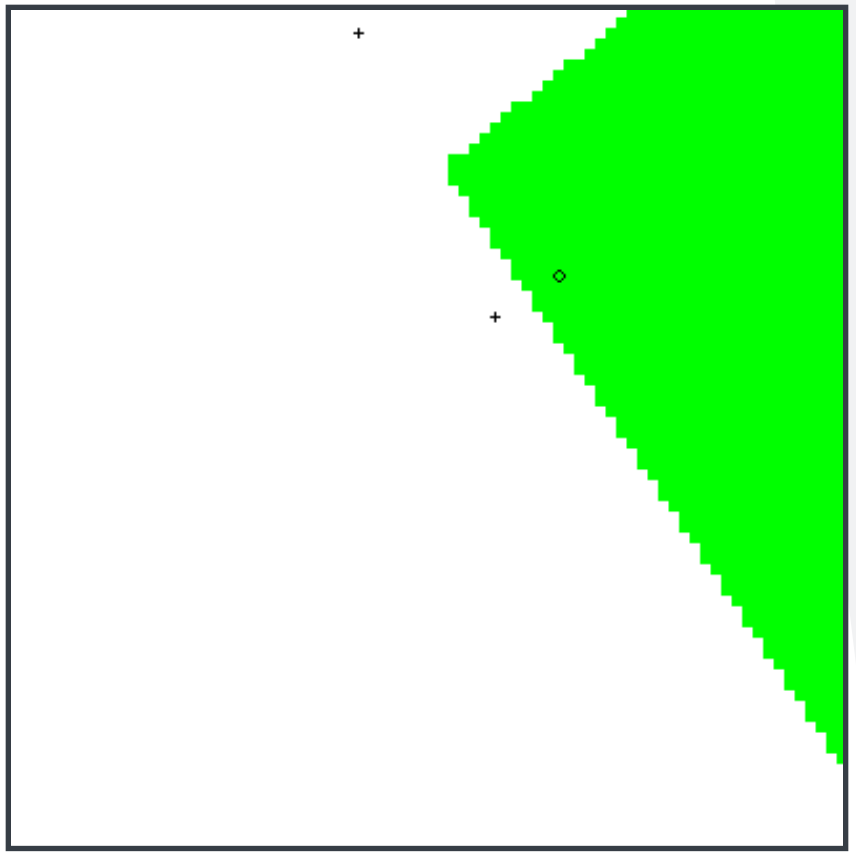
Recap

Uninformed search

Case-Based Learning

Case-Based Reasoning

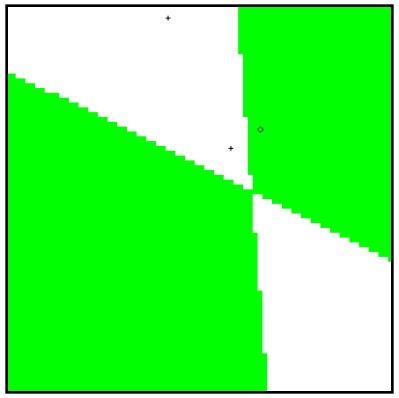
- Classification from similarity
  - Case-based reasoning
  - Predict an instance's label using similar instances
- Nearest-neighbor classification
  - 1-NN: copy the label of the most similar data point
  - K-NN: vote the k nearest neighbors (need a weighting scheme)
  - Key issue: how to define similarity
  - Trade-offs: Small k gives relevant neighbors, Large k gives smoother functions



http://www.cs.cmu.edu/~zhuxj/courseproject/knndemo/KNN.html

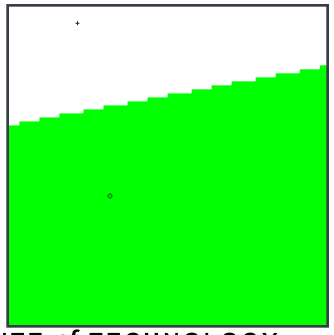
Parametric / Non-Parametric

- Parametric models:
  - Fixed set of parameters
  - More data means better settings
- Non-parametric models:
  - Complexity of the classifier increases with data
  - Better in the limit, often worse in the non-limit
- (K)NN is **non-parametric**

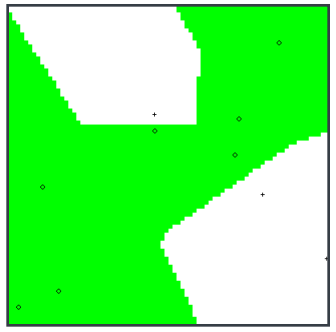


Truth

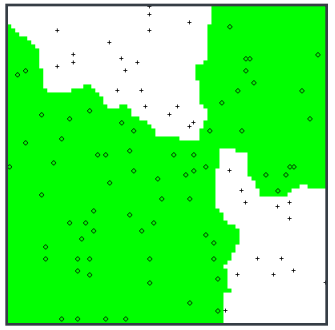
2 Examples



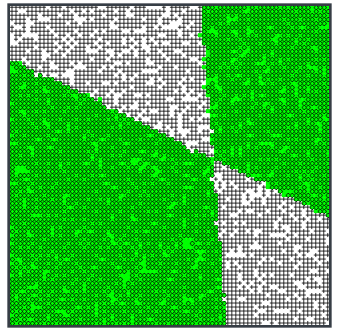
10 Examples



100 Examples



10000 Examples



Nearest-Neighbor Classification

- Nearest neighbor for digits:
  - Take new image
  - Compare to all training images
  - Assign based on closest example
- Encoding: image is vector of intensities:

 $= \langle 0.0 \;\; 0.0 \;\; 0.3 \;\; 0.8 \;\; 0.7 \;\; 0.1 \ldots 0.0 \rangle$

- What's the similarity function?
  - Dot product of two images vectors?

$$\text{sim}(x, x') = x \cdot x' = \sum_i x_i x'_i$$

  - Usually normalize vectors so ||x|| = 1
  - min = 0 (when?), max = 1 (when?)

 0

 1

 2

 0

 1

 2

Basic Similarity

- Many similarities based on **feature dot products**:

$$\text{sim}(x, x') = f(x) \cdot f(x') = \sum_i f_i(x) f_i(x')$$

- If features are just the pixels:

$$\text{sim}(x, x') = x \cdot x' = \sum_i x_i x'_i$$

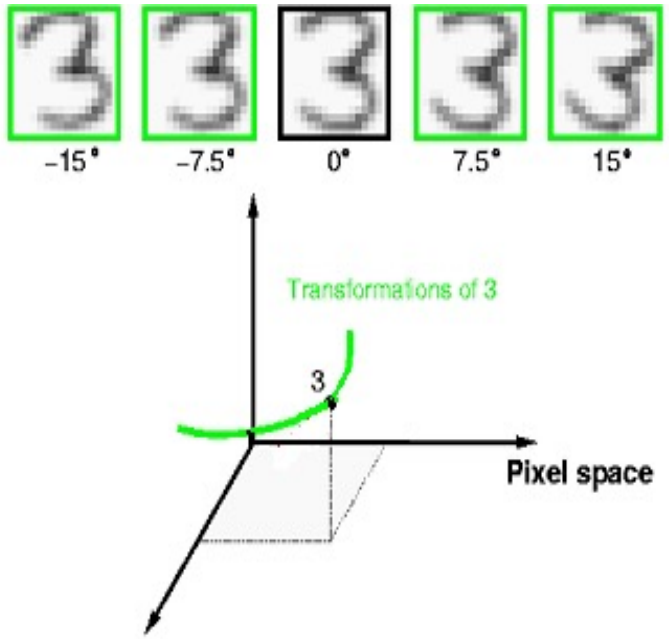
- Note: not all similarities are of this form

Invariant Metrics

- Better similarity functions use knowledge about vision
- Example: invariant metrics:
  - Similarities are invariant under certain transformations
  - Rotation, scaling, translation, stroke-thickness…
  - E.g:

    - 16 x 16 = 256 pixels; a point in 256-dim space
    - These points have small similarity in R^{256} (why?)
  - How can we incorporate such invariances into our similarities?

Rotation Invariant Metrics



- Each example is now a curve in R^{256}
- Rotation invariant similarity:

s'=max s(r(), r())

- E.g., highest similarity between images' rotation lines

Unsupervised Learning

Clustering

- Clustering systems:
  - **Unsupervised learning**
  - **Detect patterns** in unlabeled data
    - E.g. group emails or search results
    - E.g. find categories of customers
    - E.g. detect anomalous program executions
  - Useful when don’t know what you’re looking for
  - Requires data, but no labels
  - Often get gibberish

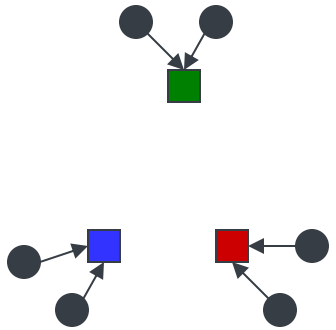
K-Means

K-Means as Optimization

- Consider the total distance to the means:

$$\phi(\{x_i\}, \{a_i\}, \{c_k\}) = \sum_i \text{dist}(x_i, c_{a_i})$$

points — assignments — means



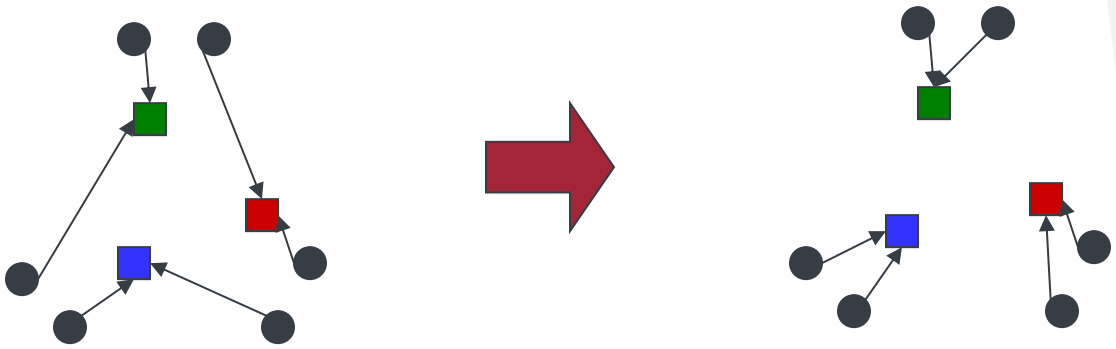
- Each iteration reduces ϕ

- Two stages each iteration:
  - Update assignments: fix means c, change assignments a
  - Update means: fix assignments a, change means c

Phase I: Update Assignments

- For each point, re-assign to closest mean:

$$a_i = \underset{k}{\text{argmin}}\, \text{dist}(x_i, c_k)$$



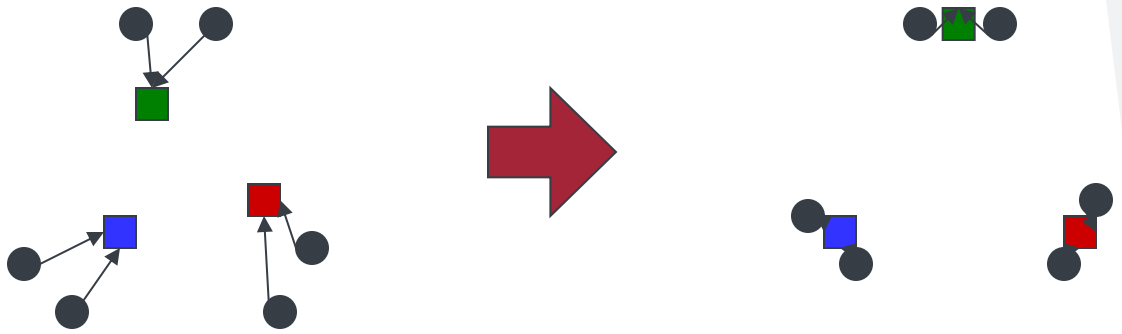
- Can only decrease total distance ϕ!

$$\phi(\{x_i\}, \{a_i\}, \{c_k\}) = \sum_i \text{dist}(x_i, c_{a_i})$$

Phase II: Update Means

- Move each mean to the average of its assigned points:

$$c_k = \frac{1}{|\{i : a_i = k\}|} \sum_{i:a_i=k} x_i$$



- Also, can only decrease total distance… (Why?)

- Fun fact: the point y with minimum squared Euclidean distance to a set of points {x} is their mean

Initialization

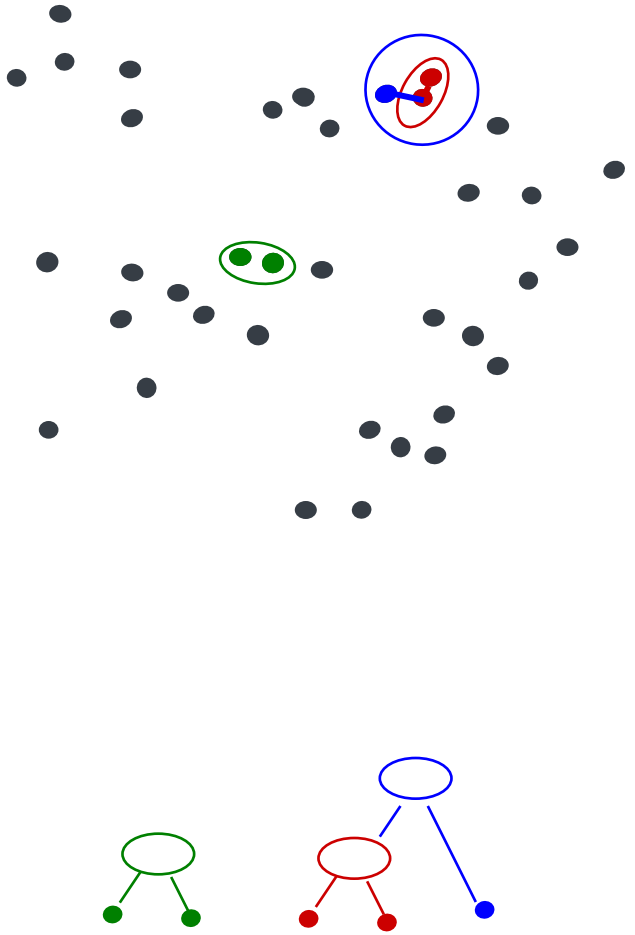
- K-means is non-deterministic
  - Requires initial means
  - It does matter what you pick!
  - What can go wrong?

  - Various schemes for preventing this kind of thing: variance-based split / merge, initialization heuristics

Agglomerative Clustering

Agglomerative Clustering

- Agglomerative clustering:
  - First merge very similar instances
  - Incrementally build larger clusters out of smaller clusters
- Algorithm:
  - Maintain a set of clusters
  - Initially, each instance in its own cluster
  - Repeat:
    - Pick the two **closest** clusters
    - Merge them into a new cluster
    - Stop when there's only one cluster left
- Produces not one clustering, but a family of clusterings represented by a **dendrogram**



Expectation Maximization

Expectation-Maximization (EM)

- EM algorithm
  - Guess h_{ML}
  - Iterate
    - Expectation: based on h_{ML} compute expectation of (missing) values
    - Maximization: based on expected (missing) values, compute new h_{ML}
- K-means algorithm is a special case of EM
  - EM with one hidden variable (*i.e.*, assignment of clusters to points) and one parameter (*i.e.*, mans of clusters)

Gaussian Mixture Models

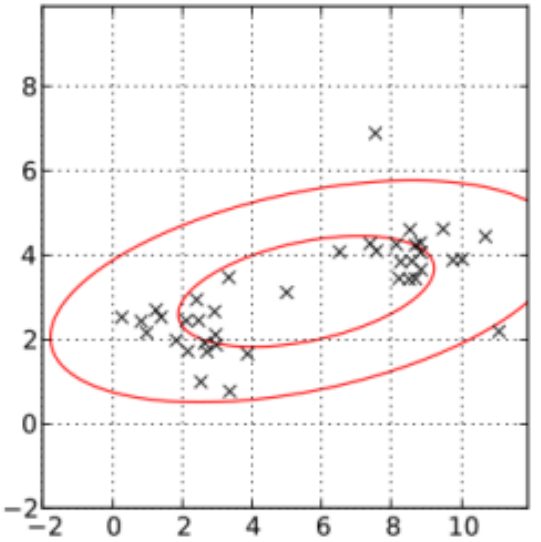
- Gaussian mixture models
  - The probability of points $p(\boldsymbol{x})$ are from a mix of Gaussian probability distributions ($\mathcal{N}$).

$$p(\boldsymbol{x}) = \sum_{i=1}^{K} w_i \mathcal{N}(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$$

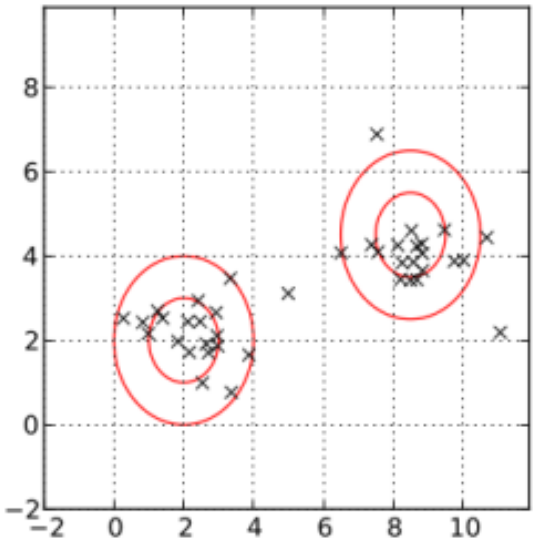
$$\mathcal{N}(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) = \frac{1}{\sqrt{(2\pi)^K |\boldsymbol{\Sigma}_i|}} \exp\left(-\frac{1}{2} (\boldsymbol{x} - \boldsymbol{\mu}_i)^T \Sigma_i^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_i) \right)$$

$$\sum_{i=1}^{K} w_i = 1$$

K=1



K=2



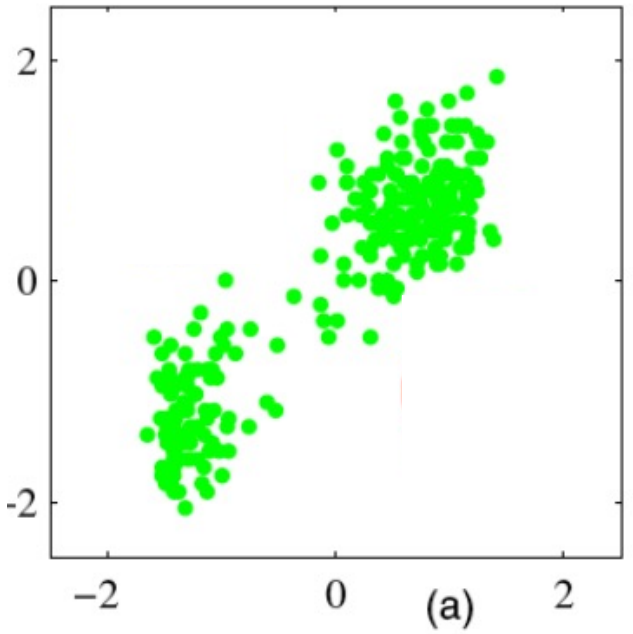
Gaussian Mixture Models

- Gaussian mixture models
  - The probability of points $p(\boldsymbol{x})$ are from a mix of Gaussian probability distributions ($\mathcal{N}$).

$$p(\boldsymbol{x}) = \sum_{i=1}^{K} w_i \mathcal{N}(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$$

$$\mathcal{N}(\boldsymbol{x}, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) = \frac{1}{\sqrt{(2\pi)^K |\boldsymbol{\Sigma}_i|}} \exp\left(-\frac{1}{2} (\boldsymbol{x} - \boldsymbol{\mu}_i)^T \Sigma_i^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_i) \right)$$

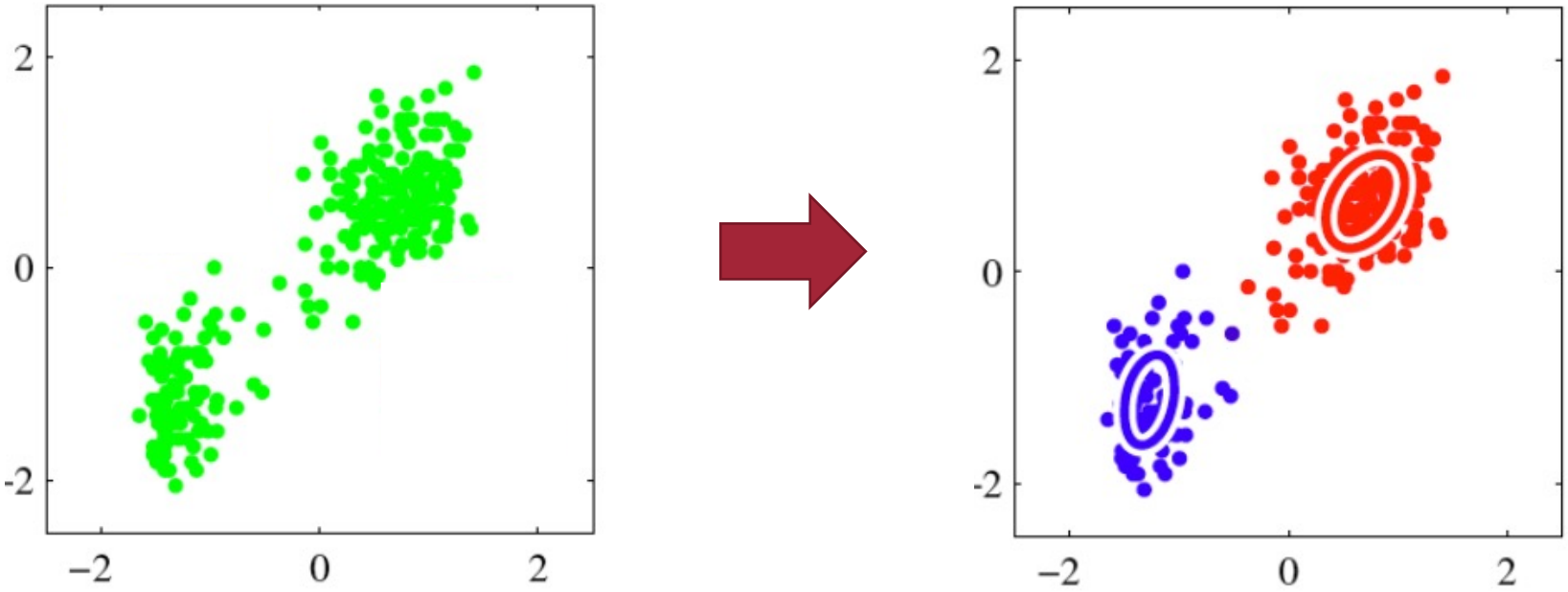
$$\sum_{i=1}^{K} w_i = 1$$



K=2

Learn $w_i, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i$!

EM Example: Learning Mixtures of Gaussians



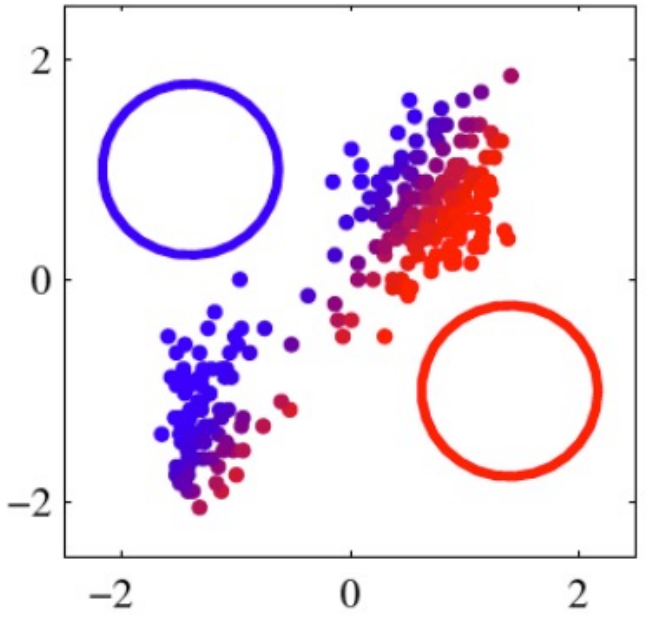
K=2

Maximize the log likelihood

$$w, \boldsymbol{\mu}, \boldsymbol{\Sigma} = \underset{w,\boldsymbol{\mu},\boldsymbol{\Sigma}}{\operatorname{argmax}} \log p(\boldsymbol{X}|w, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \underset{w,\boldsymbol{\mu},\boldsymbol{\Sigma}}{\operatorname{argmax}} \sum_{n=1}^{N} \log \left(\sum_{i=1}^{K} w_i \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \right)$$

EM Example: Learning Mixtures of Gaussians

- Initialization
  - Randomly assign values to the component **mean** estimates $\hat{\mu}_1, \hat{\mu}_2, \ldots, \hat{\mu}_K$.
    - e.g., K=2, $\hat{\mu}_1 = (-1.5, 1), \hat{\mu}_2 = (1.5, -1)$
  - Set all component variance estimates to the sample **variance** $\hat{\Sigma}_1^2, \hat{\Sigma}_2^2, \ldots, \hat{\Sigma}_K^2 = \frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{x}_n - \overline{\boldsymbol{x}})^2$, where $\overline{\boldsymbol{x}}$ is the sample mean $\overline{\boldsymbol{x}} = \frac{1}{N}\sum_{n=1}^{N}\boldsymbol{x}_n$.
  - Set all component distribution **prior** estimates to the uniform distribution $\hat{w}_1, \hat{w}_2, \ldots, \hat{w}_K = \frac{1}{K}$

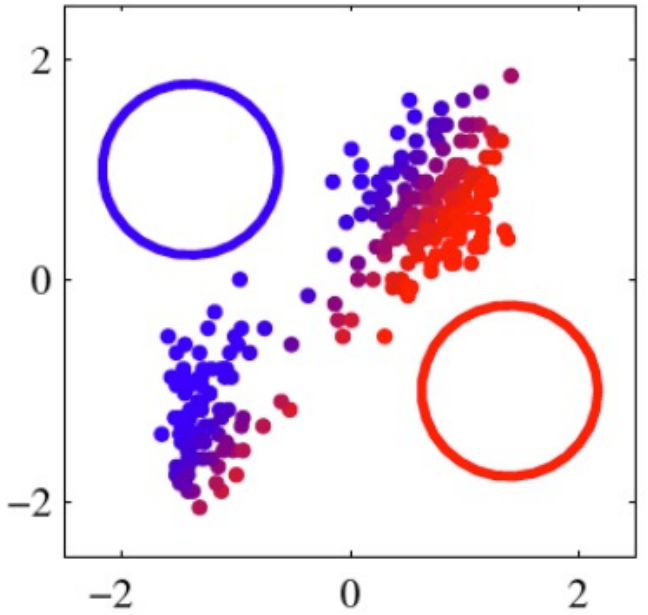


EM Example: Learning Mixtures of Gaussians

- Expectation (E) step
  - Calculate probability ($p_{i,n}$) for all clusters ($\forall i$) and points ($\forall n$)

$$p_{i,n} = P(C = i|\boldsymbol{x}_n, w, \boldsymbol{\mu}, \boldsymbol{\Sigma})$$

$$= \frac{w_i \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)}{\sum_{j=1}^{K} w_j \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

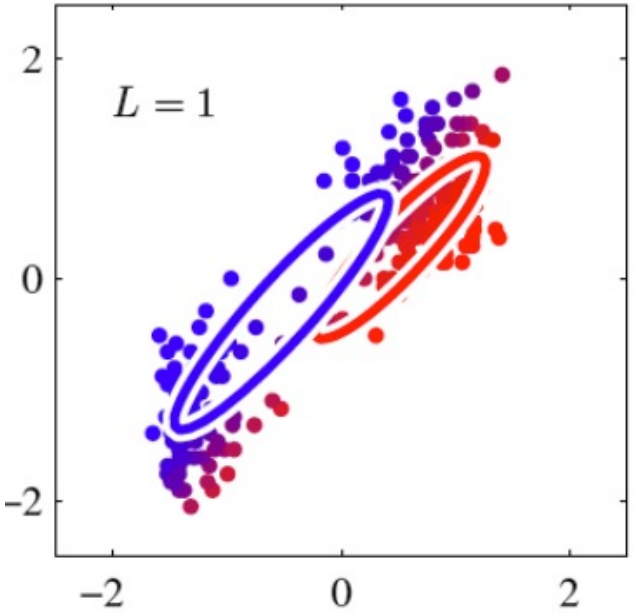


EM Example: Learning Mixtures of Gaussians

- Maximization (M) step
  - Using the $p_{i,n}$ in the expectation step, calculate the following, $\forall i$:

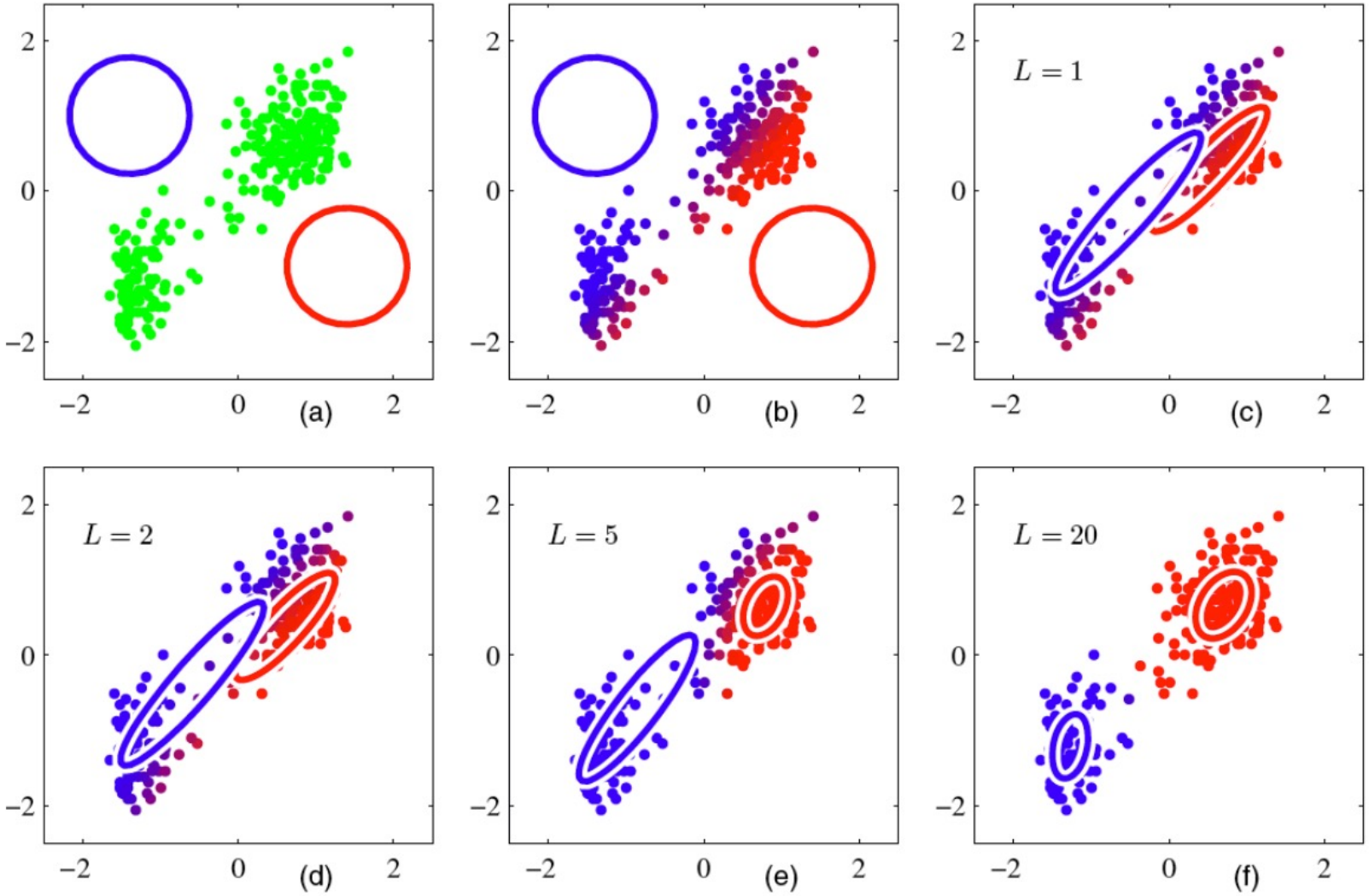
$$\widehat{w}_i = \sum_{n=1}^{N} \frac{p_{i,n}}{N} \qquad \widehat{\boldsymbol{\mu}}_i = \frac{\sum_{n=1}^{N} p_{i,n}\boldsymbol{x}_n}{\sum_{n=1}^{N} p_{i,n}}$$

$$\boldsymbol{\Sigma}_i = \frac{\sum_{n=1}^{N} p_{i,n}(\boldsymbol{x}_n - \widehat{\boldsymbol{\mu}}_i)^2}{\sum_{n=1}^{N} p_{i,n}}$$



EM Example: Learning Mixtures of Gaussians

- Iterate
  - Expectation (E) step
  - Maximization (M) step



The General Form of the EM Algorithm

The General Form of the EM Algorithm

- The examples involve two steps:
  - Step 1 (E): computing expected values of hidden variables for each example
  - Step 2 (M): recomputing the parameters, using the expected values as if they were observed values.

The General Form of the EM Algorithm

- The general form

$$\theta^{(i+1)} = \underset{\theta}{\operatorname{argmax}} \sum_{\boldsymbol{Z}} P\left(\boldsymbol{Z} = \boldsymbol{z} \middle| \boldsymbol{x}, \boldsymbol{\theta}^{(i)}\right) L(\boldsymbol{x}, \boldsymbol{Z} = \boldsymbol{z} | \boldsymbol{\theta})$$

  where $\boldsymbol{Z}$ denote all the hidden variables for all the examples, L is log likelihood, and $\boldsymbol{\theta}$ be all the parameters for the probability model.

- e.g., Gaussian Mixture Models
  - Hidden variables ($\boldsymbol{Z}$): Z_{ij} where Z_{ij} is 1 if example j was generated by component I
  - Parameters ($\boldsymbol{\theta}$): $w, \boldsymbol{\mu}, \boldsymbol{\Sigma}$

The General Form of the EM Algorithm

- The general form

$$\theta^{(i+1)} = \underset{\theta}{\operatorname{argmax}} \underbrace{\sum_{\boldsymbol{Z}} P\left(\boldsymbol{Z} = \boldsymbol{z} \middle| \boldsymbol{x}, \boldsymbol{\theta}^{(i)}\right) L(\boldsymbol{x}, \boldsymbol{Z} = \boldsymbol{z} | \boldsymbol{\theta})}$$

- The E-step is the computation of summation
  - The expectation of the log likelihood of the "completed" data with respect to the distribution $P\left(\boldsymbol{Z} = \boldsymbol{z} \middle| \boldsymbol{x}, \boldsymbol{\theta}^{(i)}\right)$

The General Form of the EM Algorithm

- The general form

$$\theta^{(i+1)} = \underset{\theta}{\operatorname{argmax}} \sum_{\boldsymbol{Z}} P\left(\boldsymbol{Z} = \boldsymbol{z} \middle| \boldsymbol{x}, \boldsymbol{\theta}^{(i)}\right) L(\boldsymbol{x}, \boldsymbol{Z} = \boldsymbol{z} | \boldsymbol{\theta})$$

- The M-step is maximizing the summation with θ

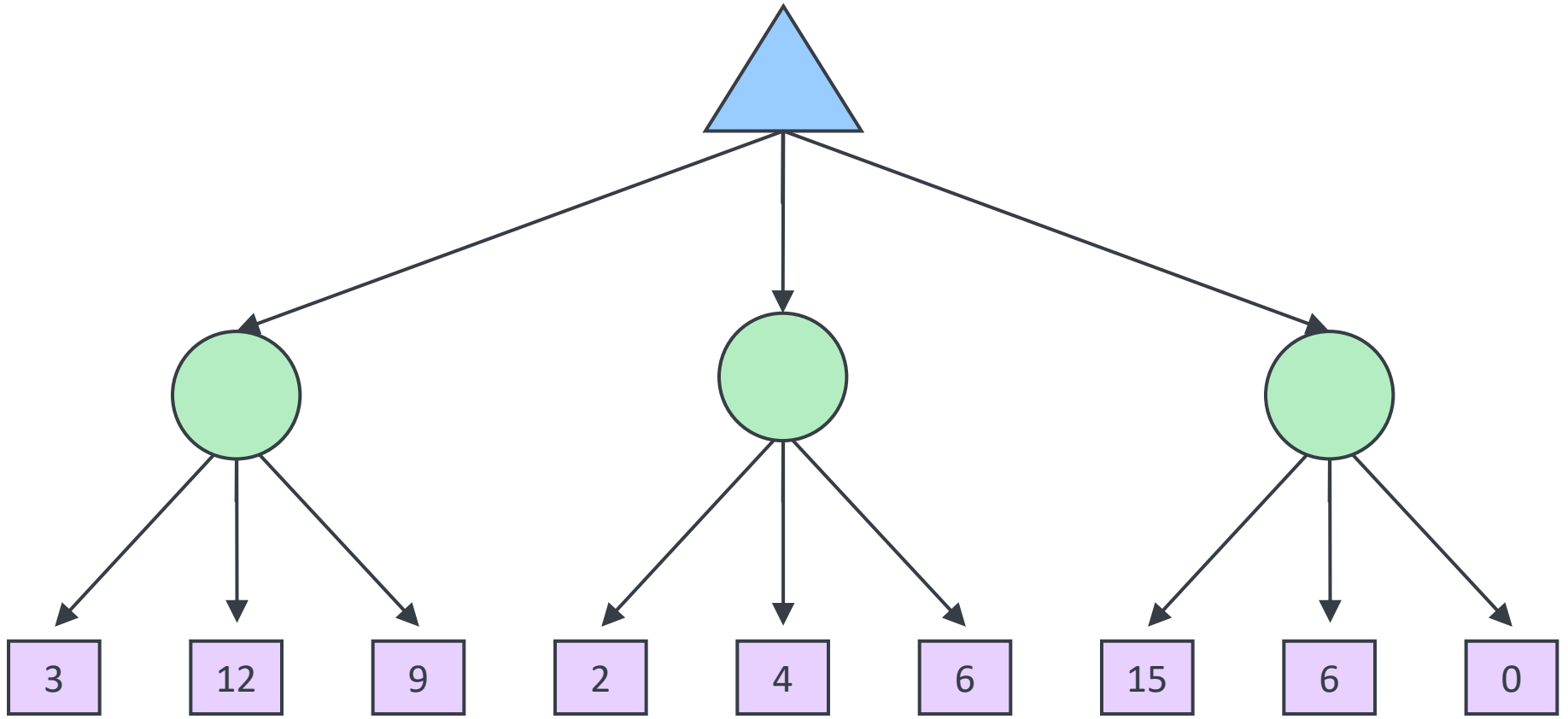
STEVENS
INSTITUTE OF TECHNOLOGY
1870

Markov Decision Process

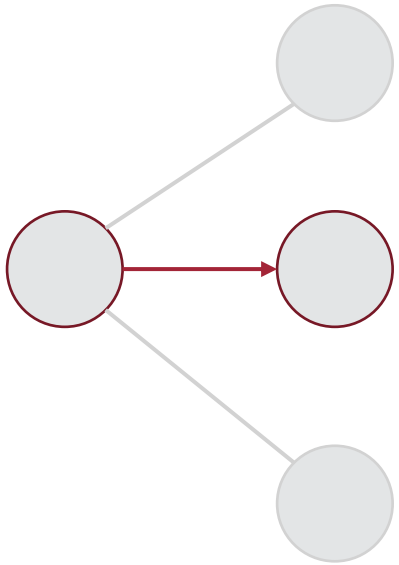
Chapter 17

Markov Decision Process (MDP)

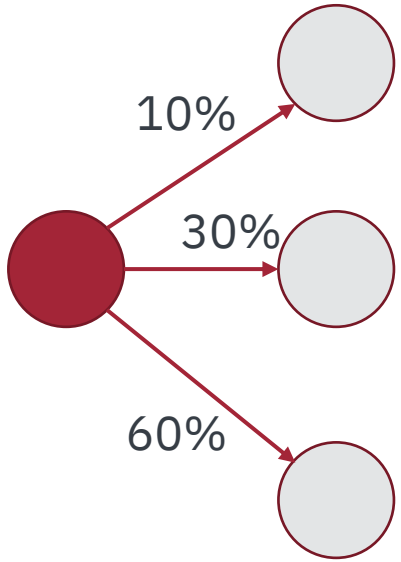
Reminder: Expectimax Search Tree



Non-Deterministic Search



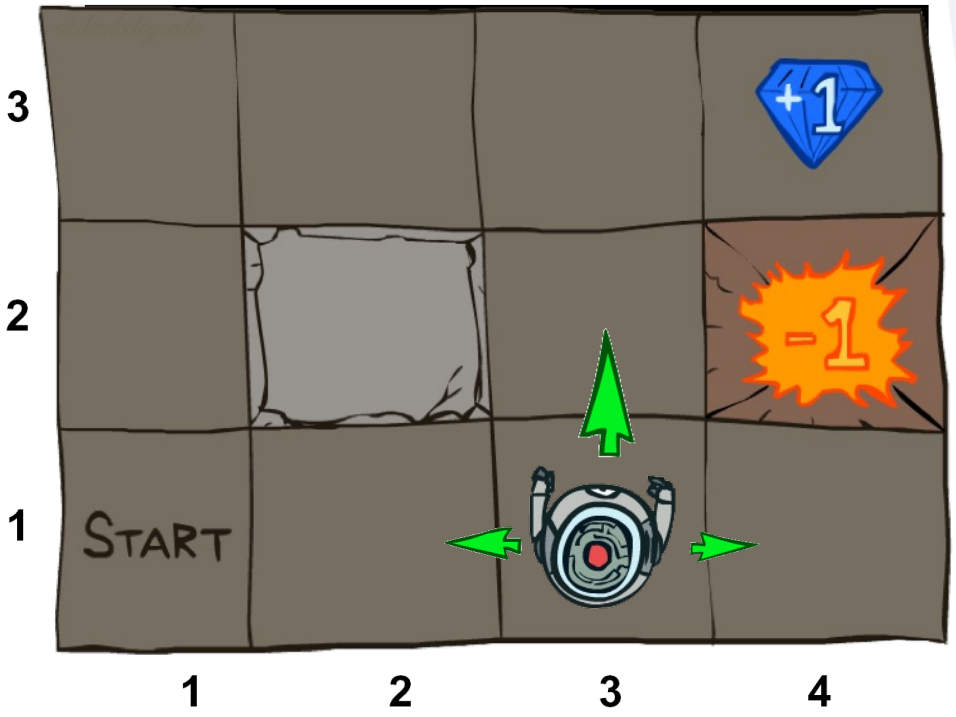
Deterministic



Non-deterministic

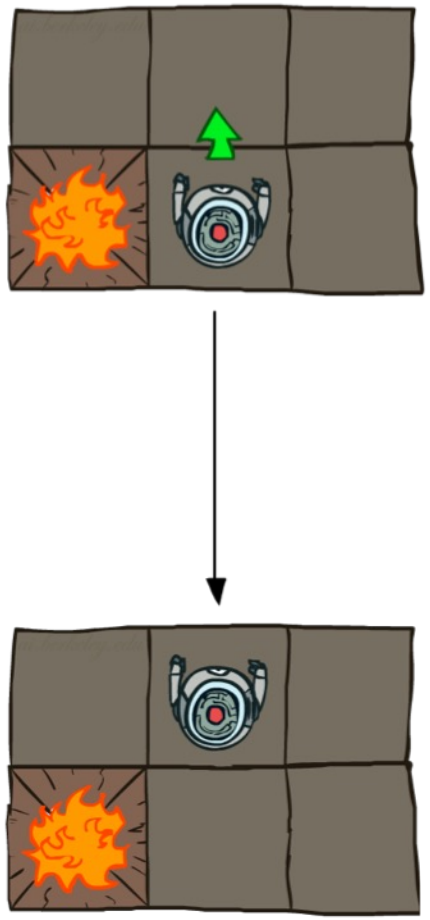
Example: Grid World

- 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 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

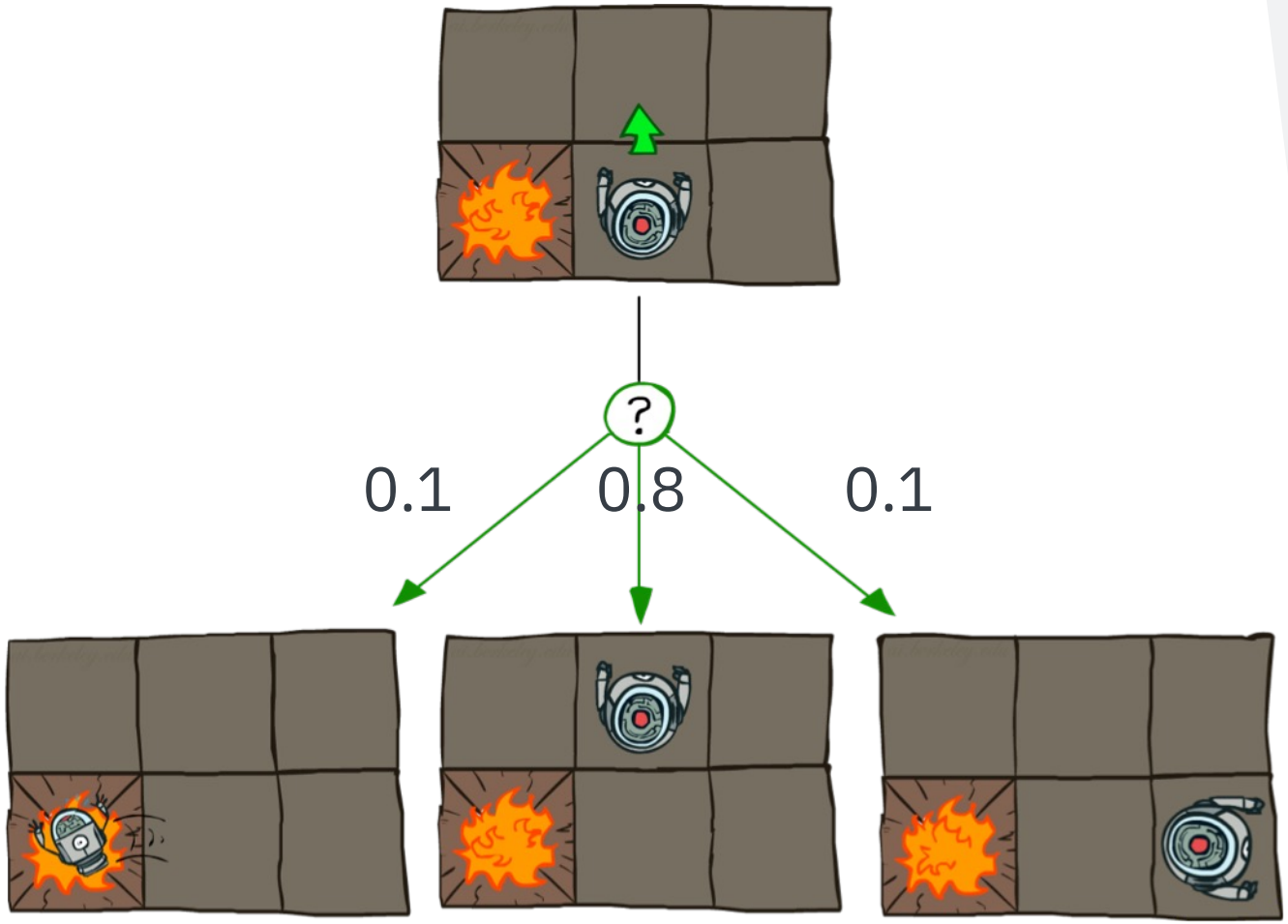


Grid World Actions

Deterministic Grid World

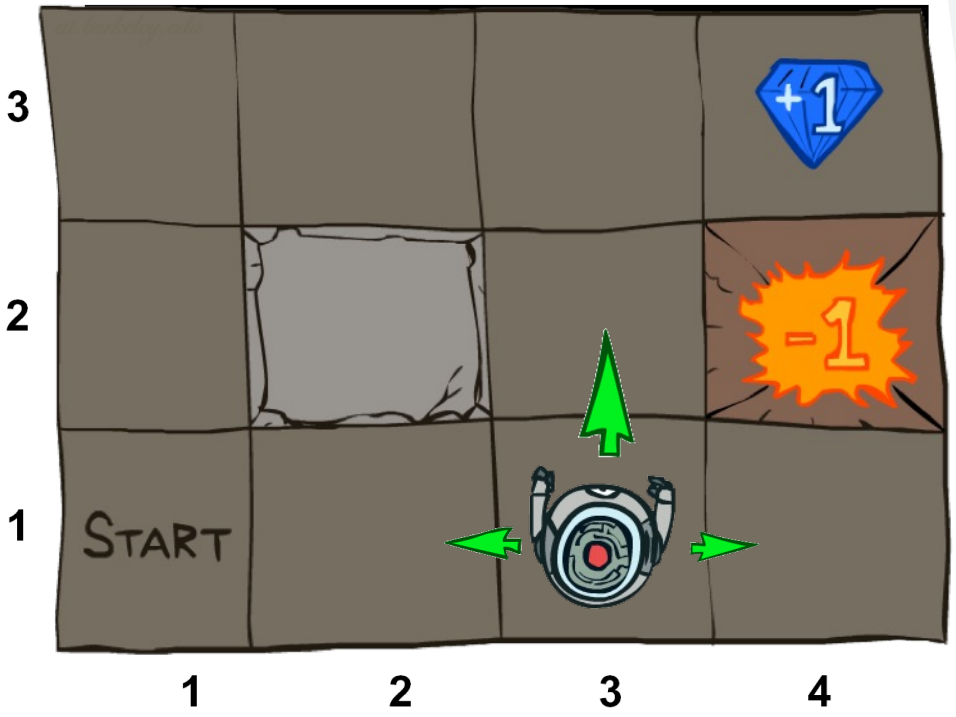


Stochastic Grid World



Markov Decision Processes

- An MDP is defined by:
  - A set of states s ∈ S
  - A set of actions a ∈ 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



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

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

$$=$$

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

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

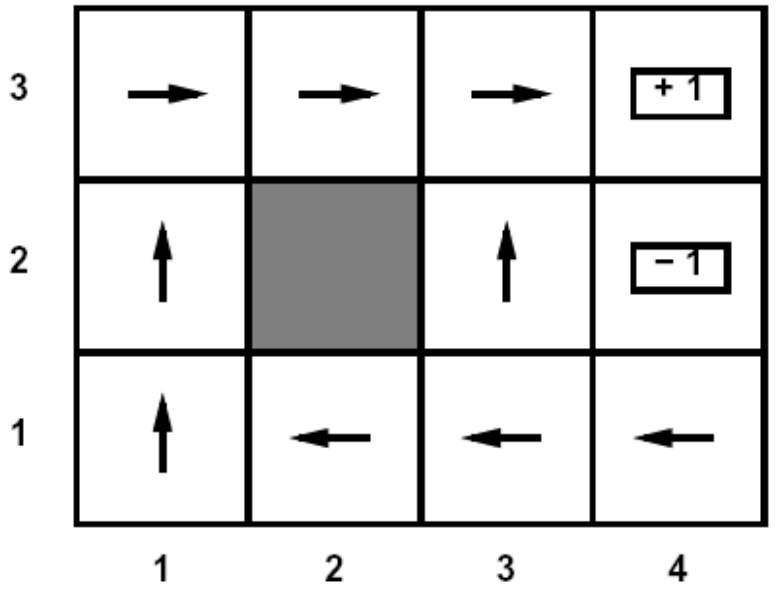
Andrey Markov
(1856-1922)

Policies

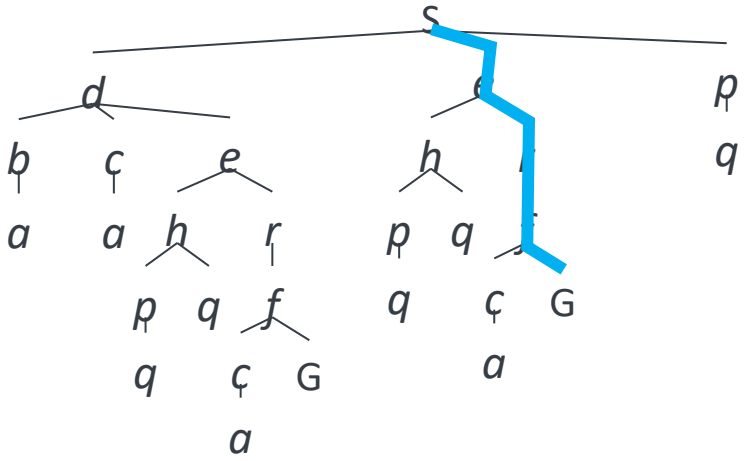
- In deterministic single-agent search problems, we wanted an optimal **plan**, or sequence of actions, from start to a goal

- 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

- Expectimax didn't compute entire policies
  - It computed the action for a single state only

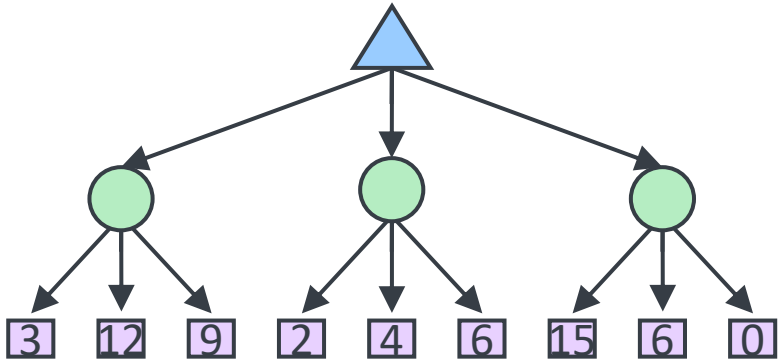


Policies



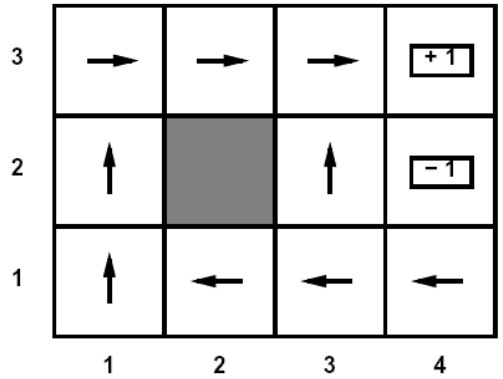
Deterministic search

Sequence of actions from **the root state**



Expectimax search

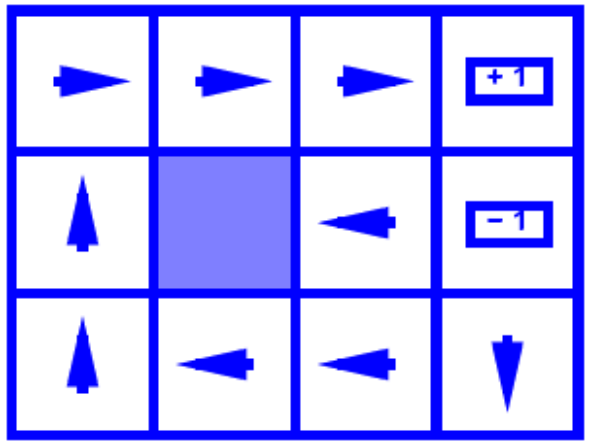
Optimal action for **the root state**



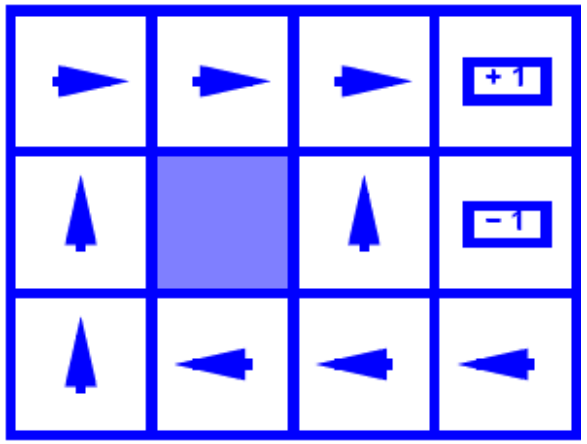
MDP

Policy of actions **for all states**

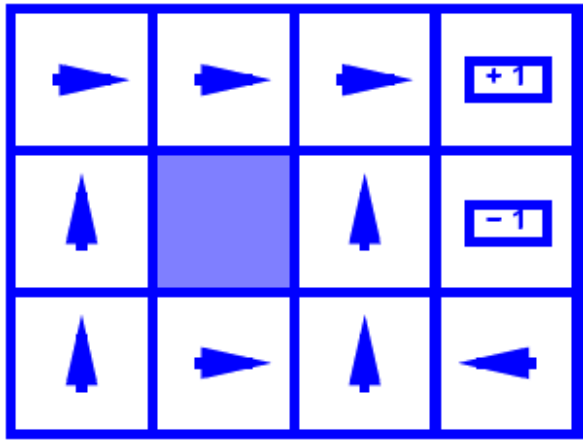
Optimal Policies



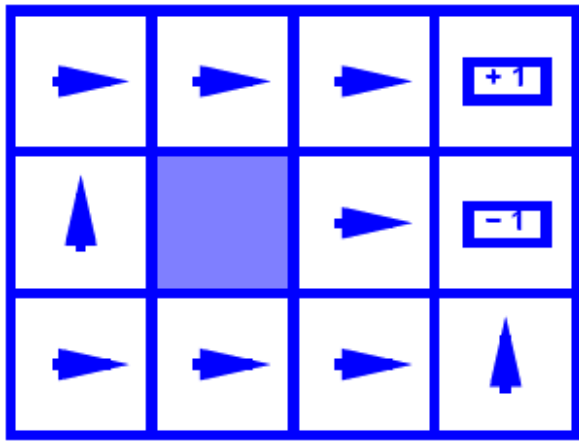
R(s) = -0.01



R(s) = -0.03



R(s) = -0.4

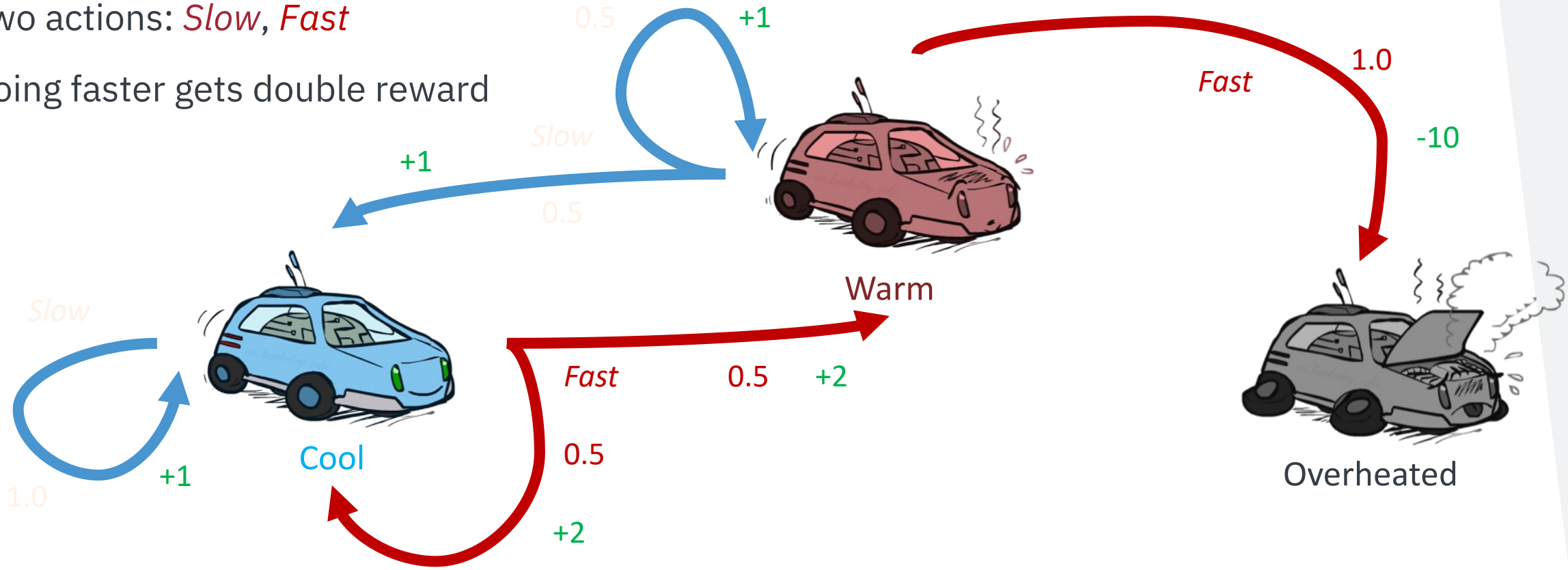


R(s) = -2.0

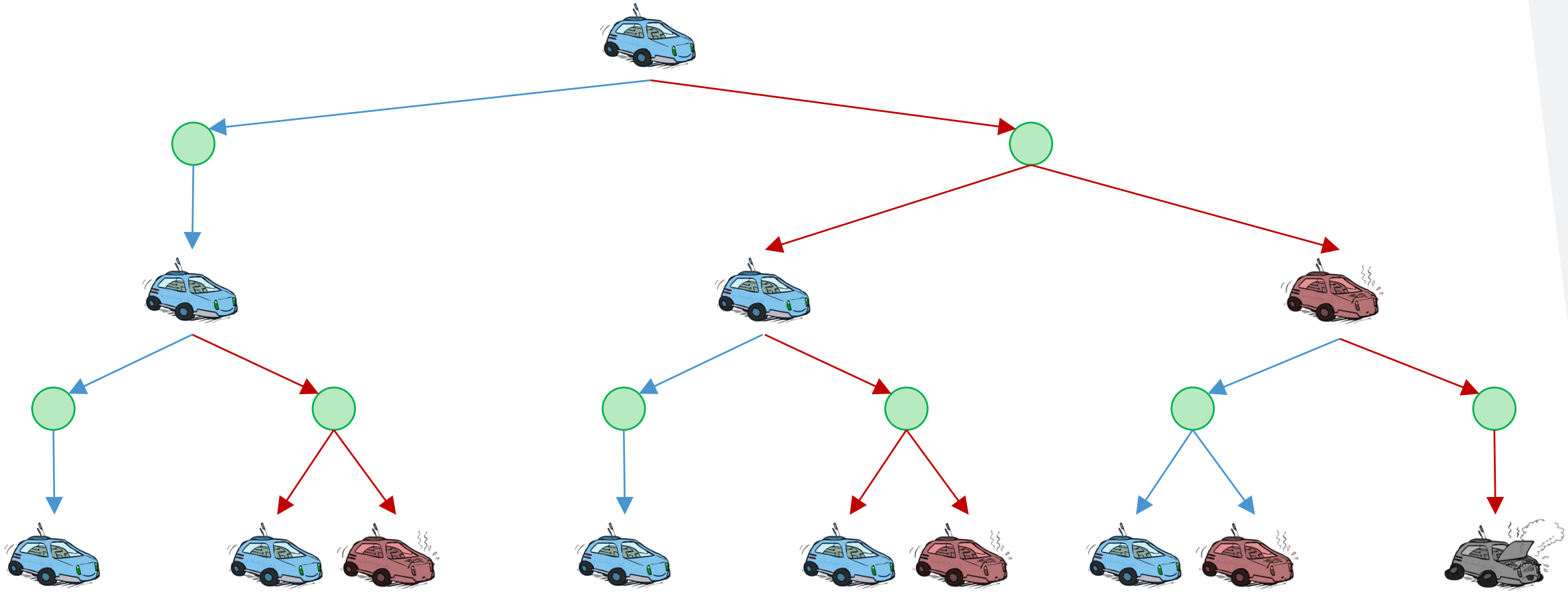
Example: Racing

Example: Racing

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

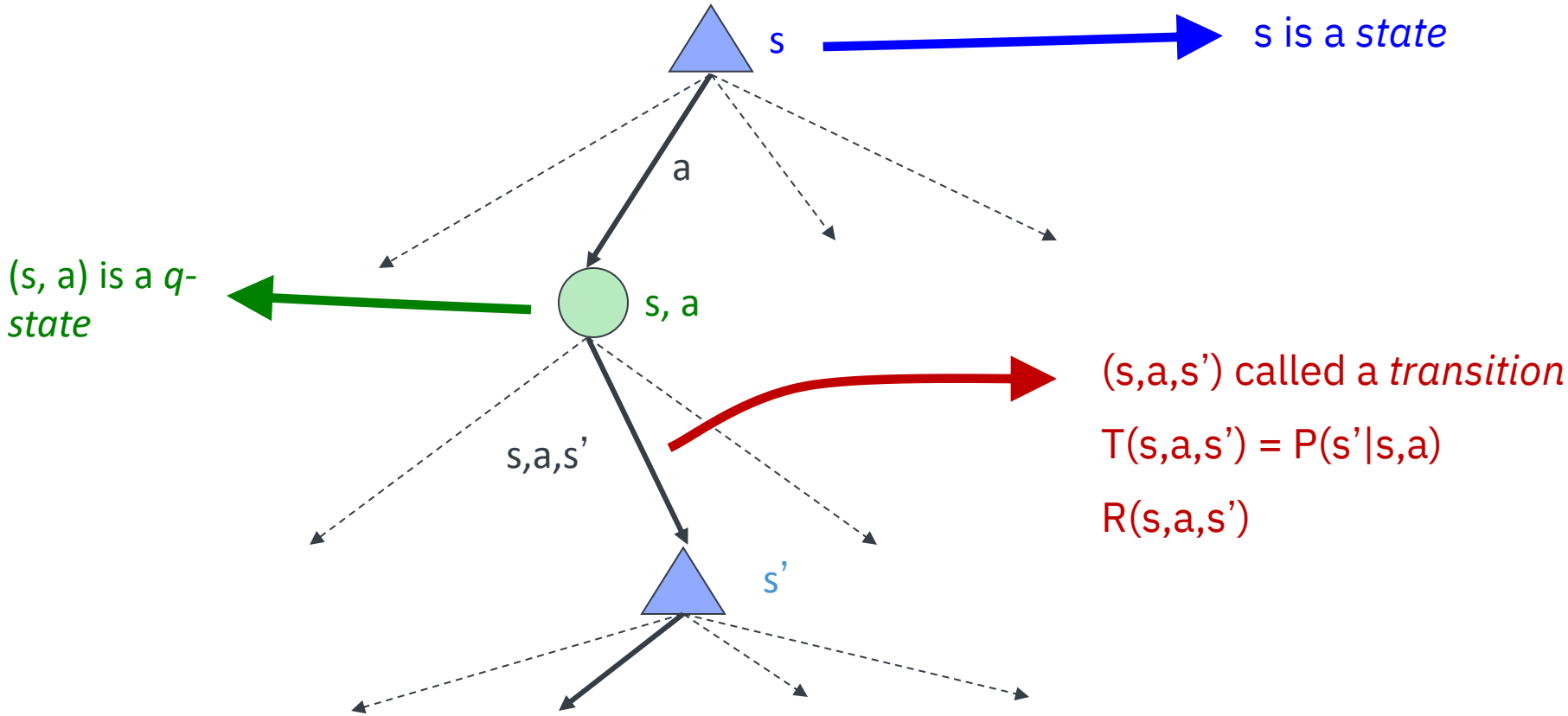


Racing Search Tree



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

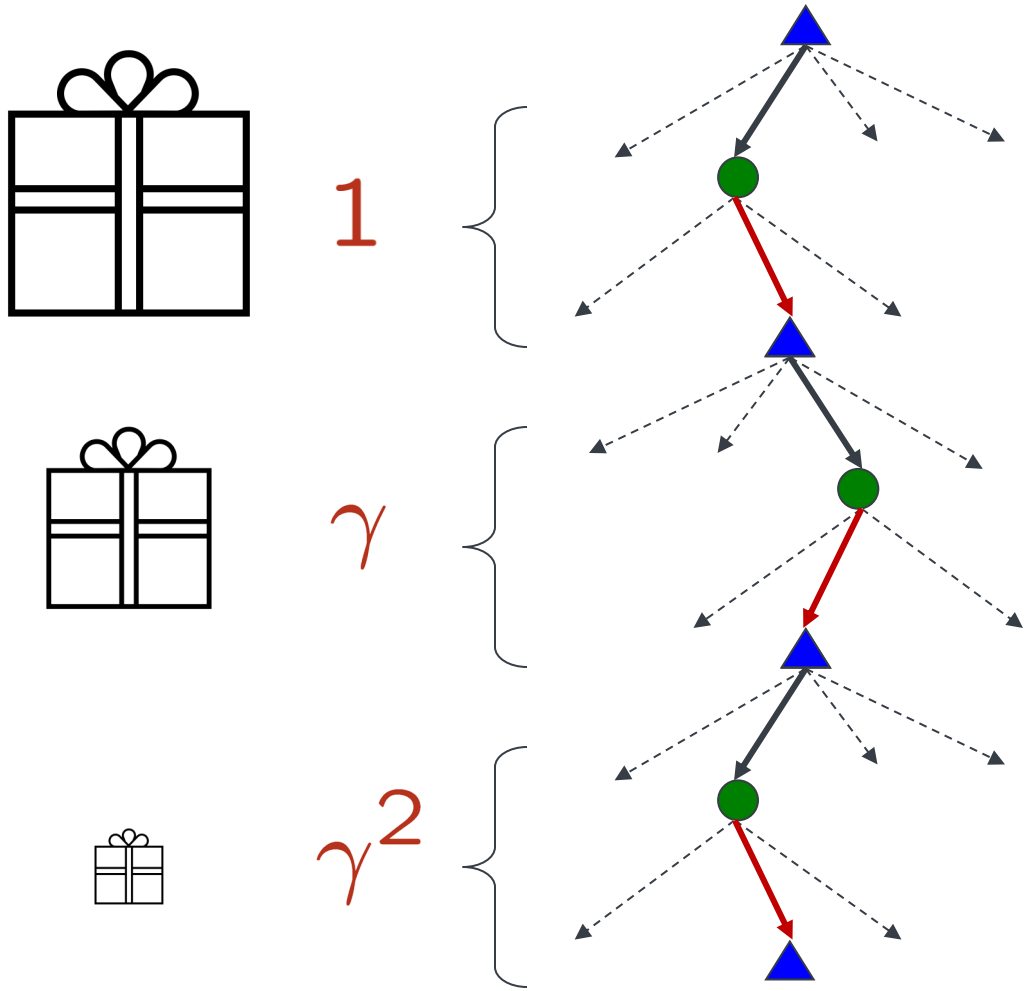
Worth Next Step

γ^2

Worth In Two Steps

Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Why discount?
  - Sooner rewards probably do have higher utility than later rewards
  - 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])



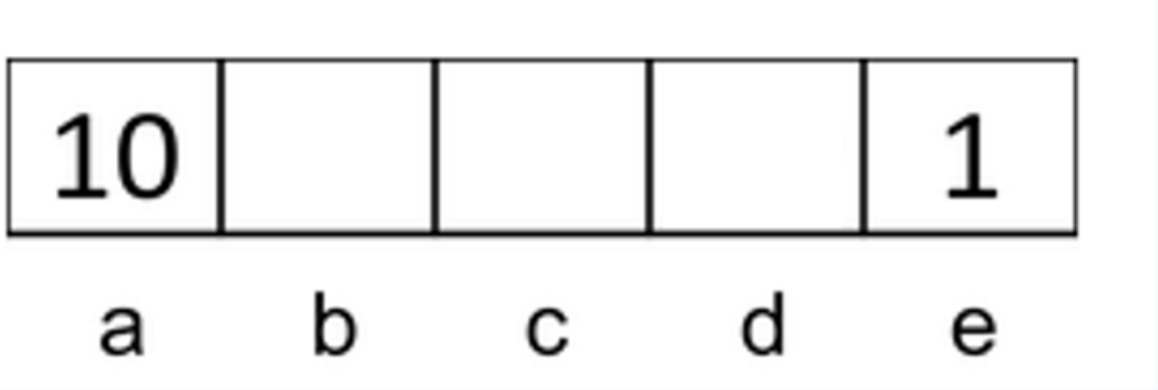
Stationary Preferences

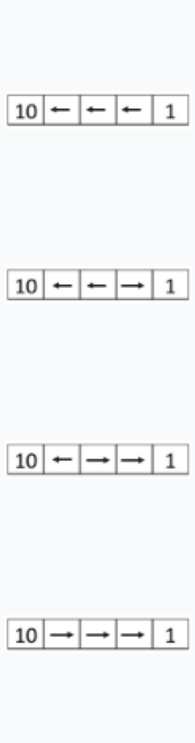
- Theorem: if we assume **stationary preferences**:

$$[a_1, a_2, \ldots] \succ [b_1, b_2, \ldots]$$
$$\Updownarrow$$
$$[r, a_1, a_2, \ldots] \succ [r, b_1, b_2, \ldots]$$

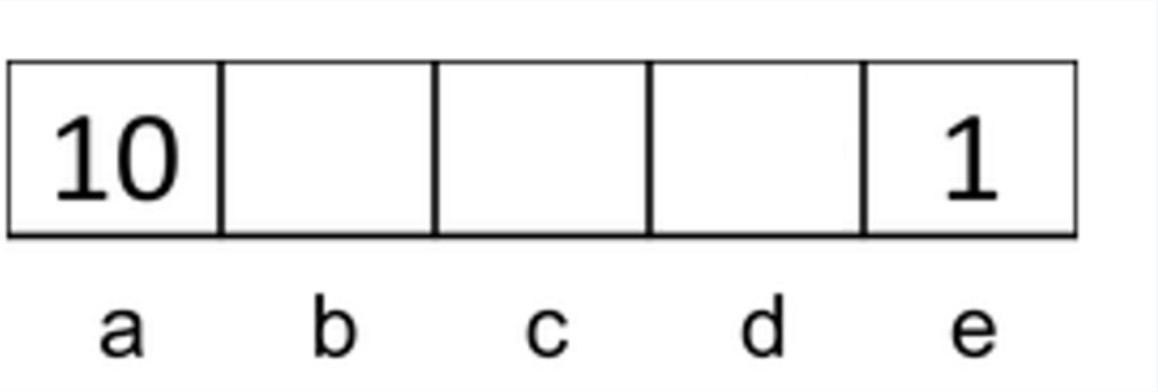
- Then: there are only two ways to define utilities
  - Additive utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + r_1 + r_2 + \cdots$
  - Discounted utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \cdots$

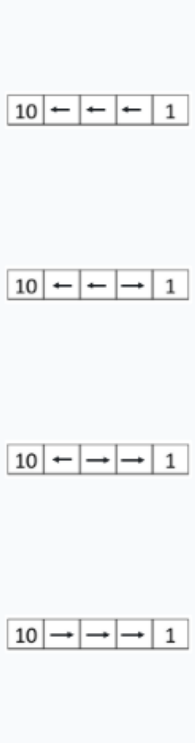
For γ=1, what is the optimal policy? (Trainsitions are deterministic.)





For γ=0.1, what is the optimal policy? (Trainsitions are deterministic.)





Infinite Utilities?!

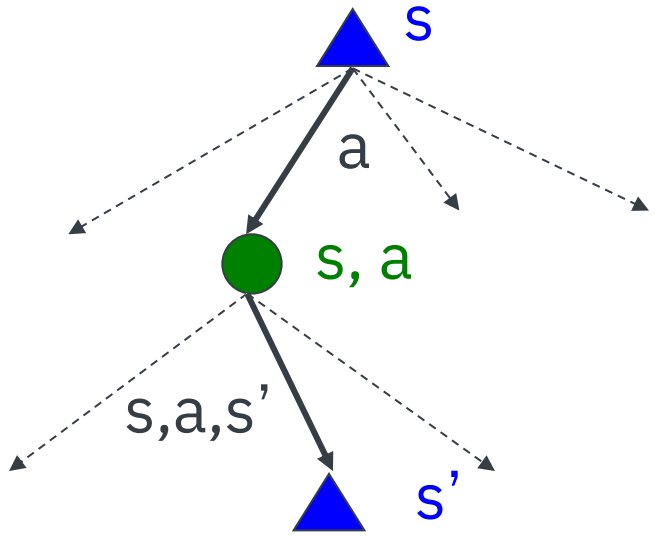
- Problem: What if the game lasts forever? Do we get infinite rewards?
- Solutions:
  - Finite horizon: (similar to depth-limited search)
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives nonstationary policies (π depends on time left)
  - Discounting: use $0 < \gamma < 1$

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

    - 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)

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

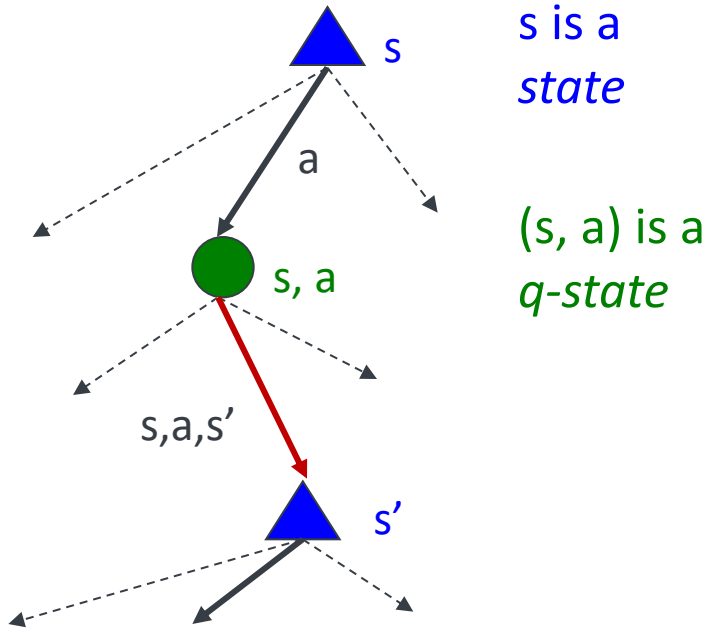
Solving MDPs

Optimal Quantities

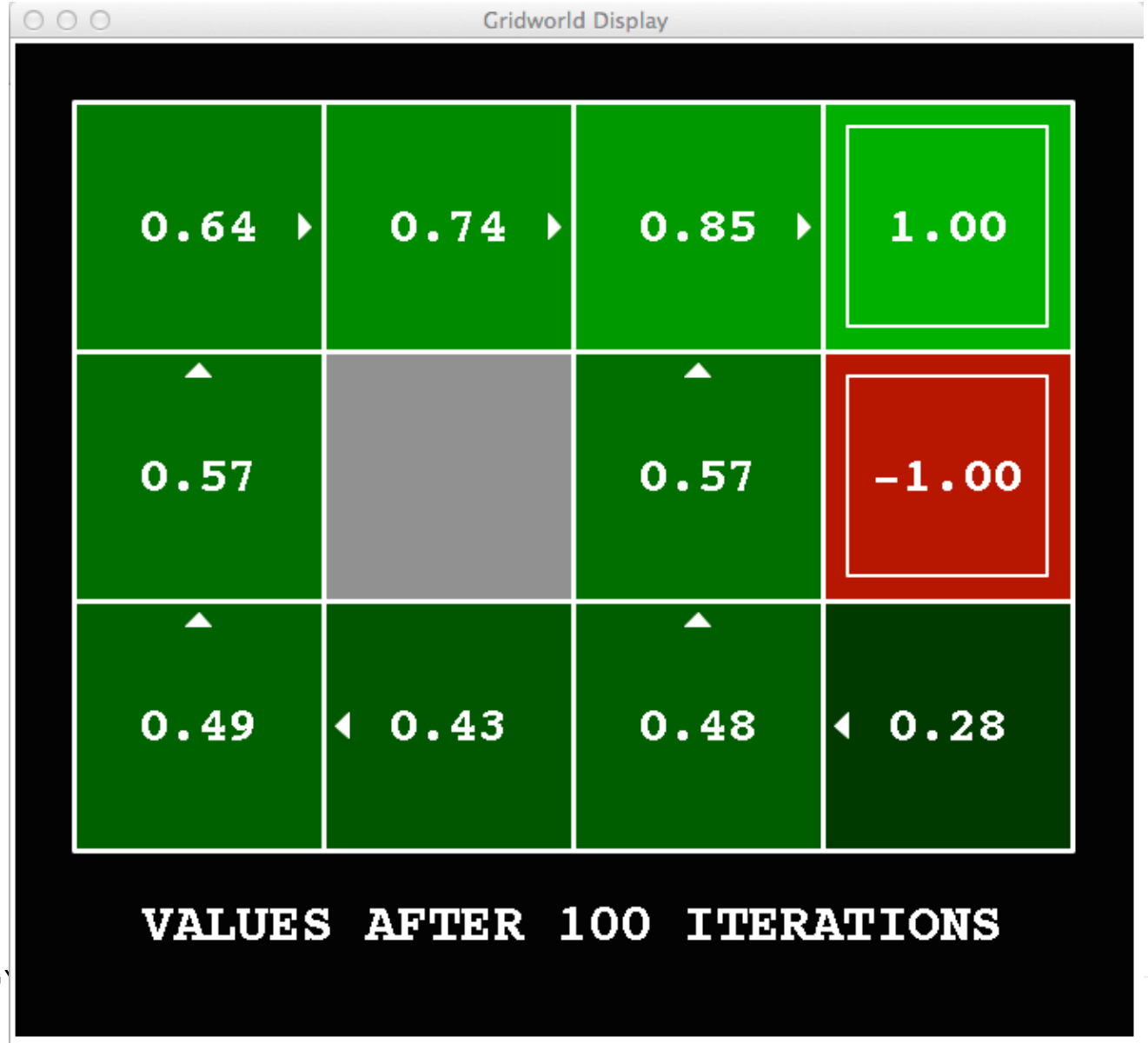
- The value (utility) of a state s:
  - V*(s) = expected utility starting in s and acting optimally

- 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:
  - π*(s) = optimal action from state s

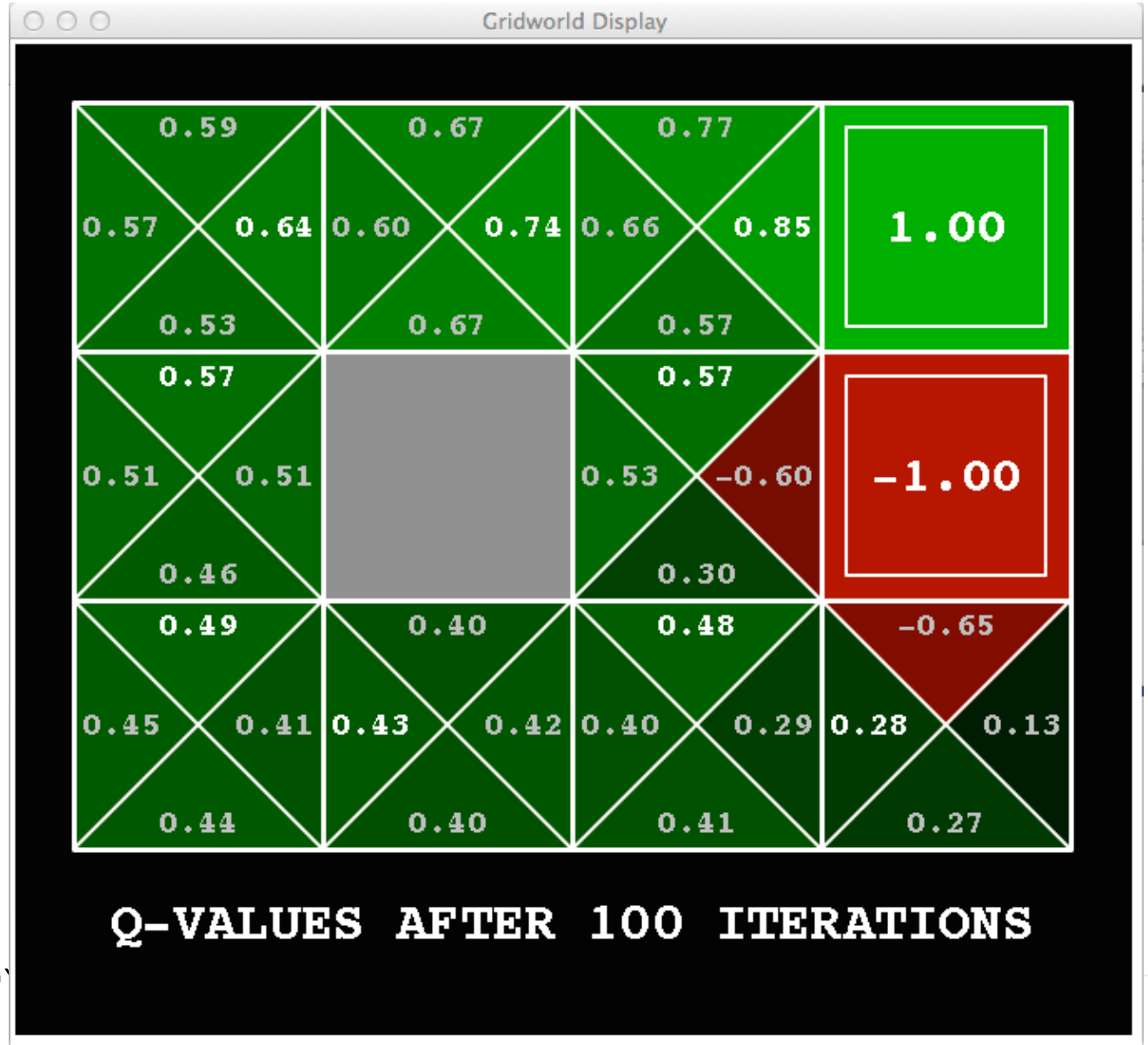


Snapshot of Demo – Gridworld V Values



Noise = 0.2
Discount = 0.9
Living reward = 0

Snapshot of Demo – Gridworld Q Values



Noise = 0.2
Discount = 0.9
Living reward = 0

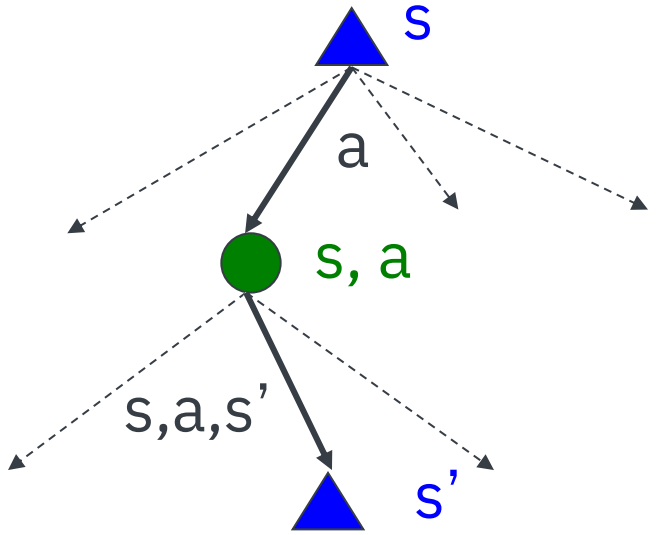
Values of States

- Fundamental operation: compute the (expectimax) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards
  - This is just what expectimax computed!
- The Bellman equations:

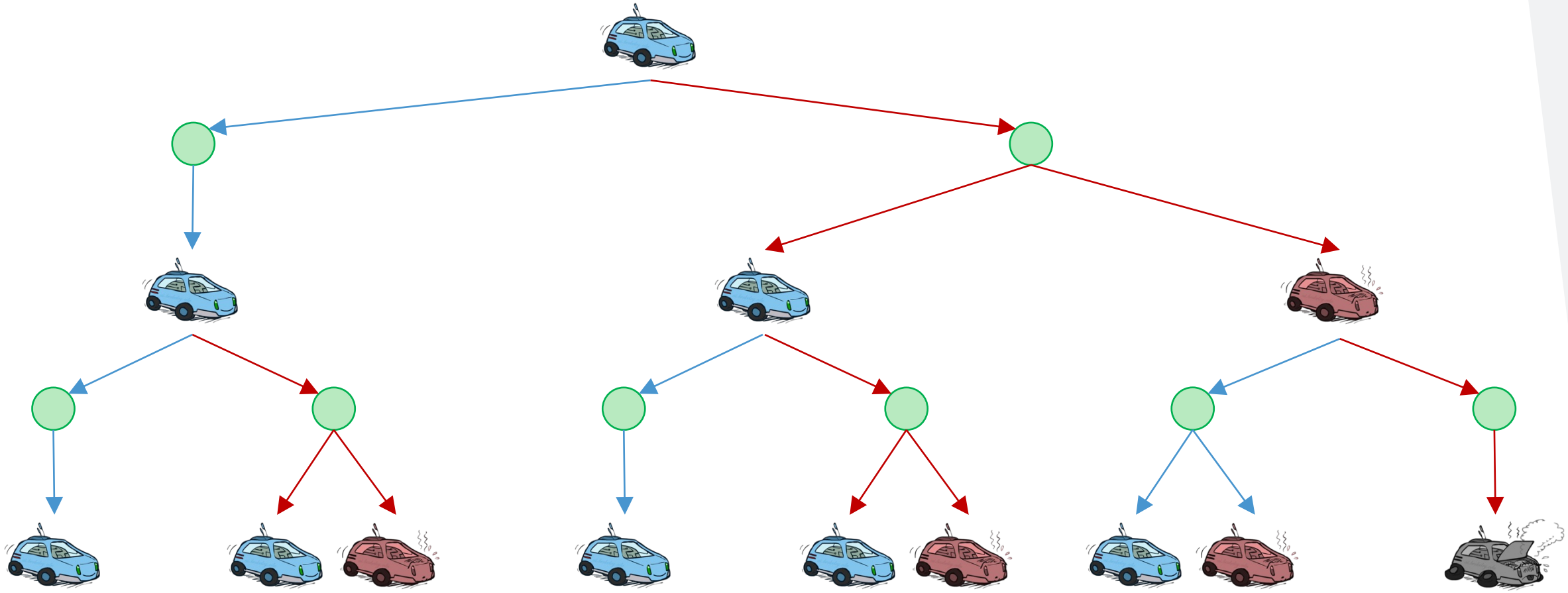
$$V^*(s) = \max_a Q^*(s,a)$$

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

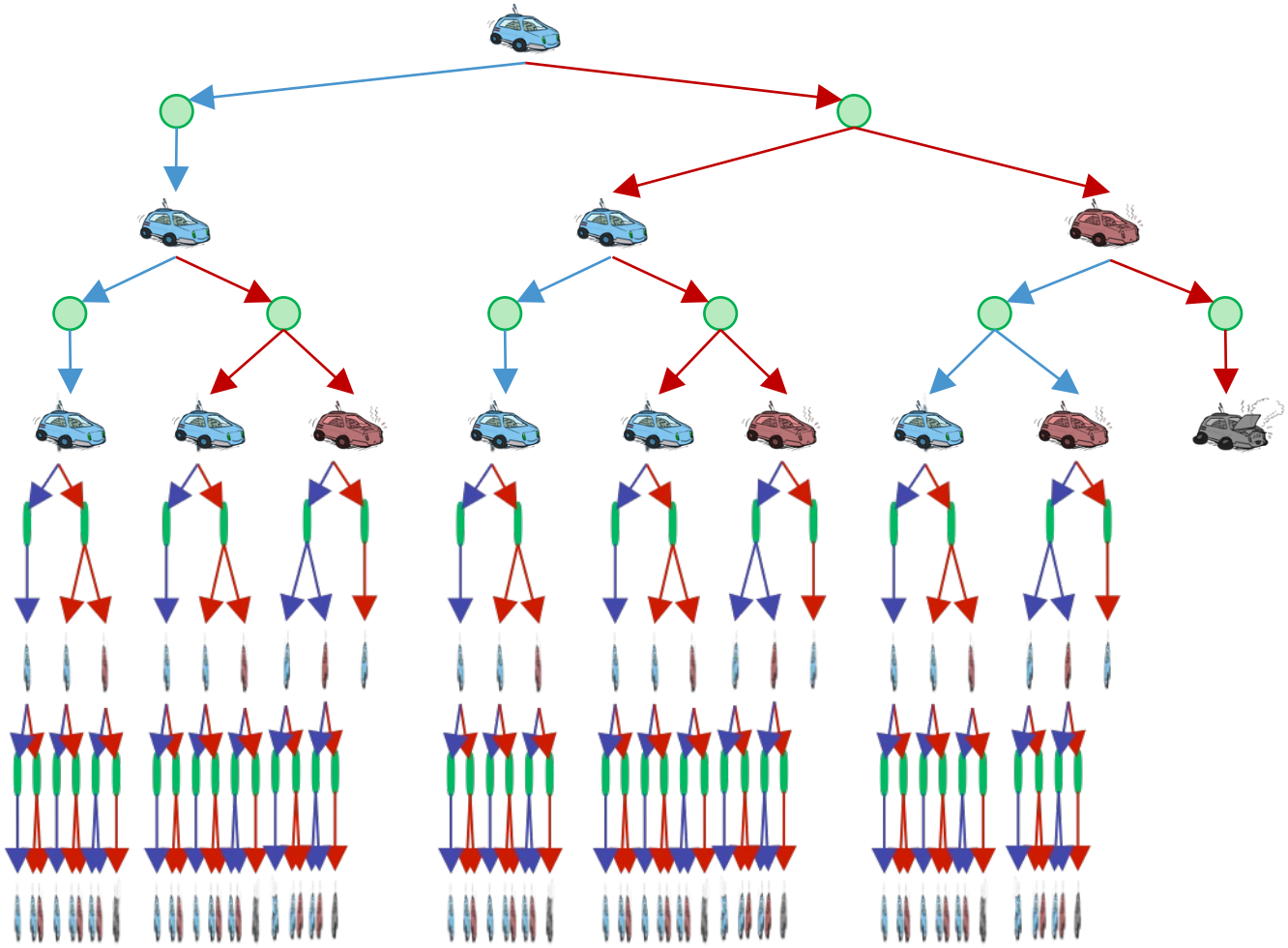
$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$



Racing Search Tree

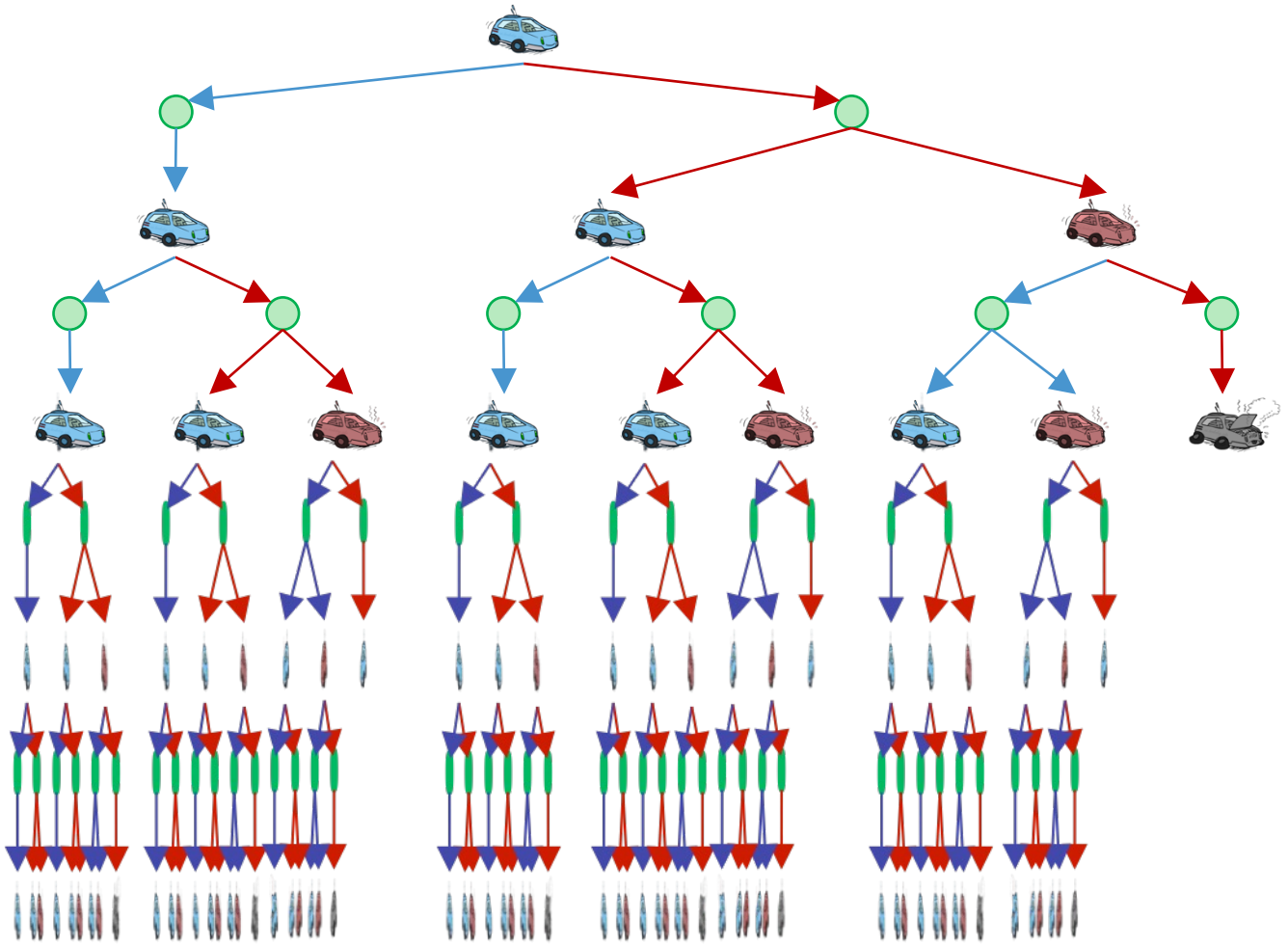


Racing Search Tree



Racing Search Tree

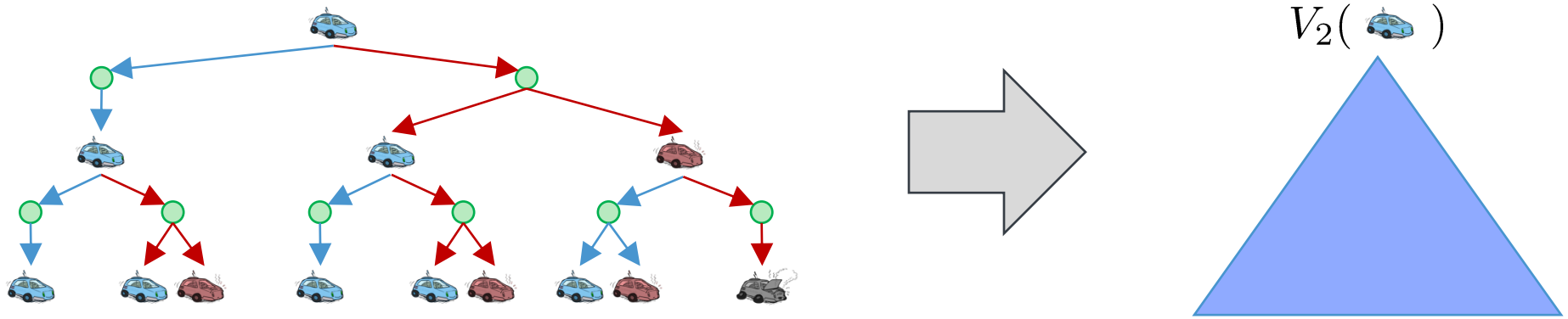
- We're doing way too much work with expectimax!
- Problem: States are repeated
  - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: deep parts of the tree eventually don't matter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



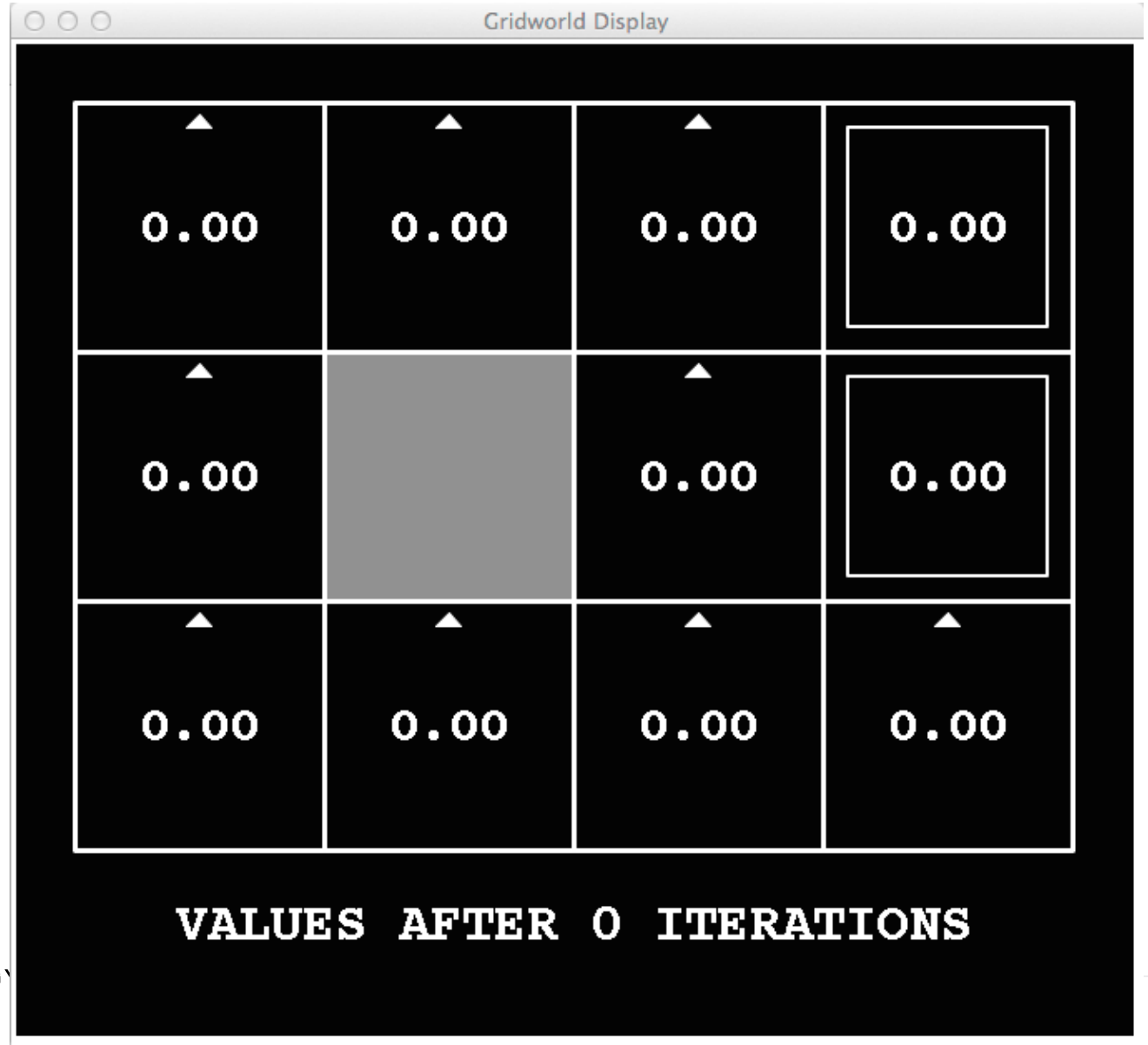
Time-Limited Values

- 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

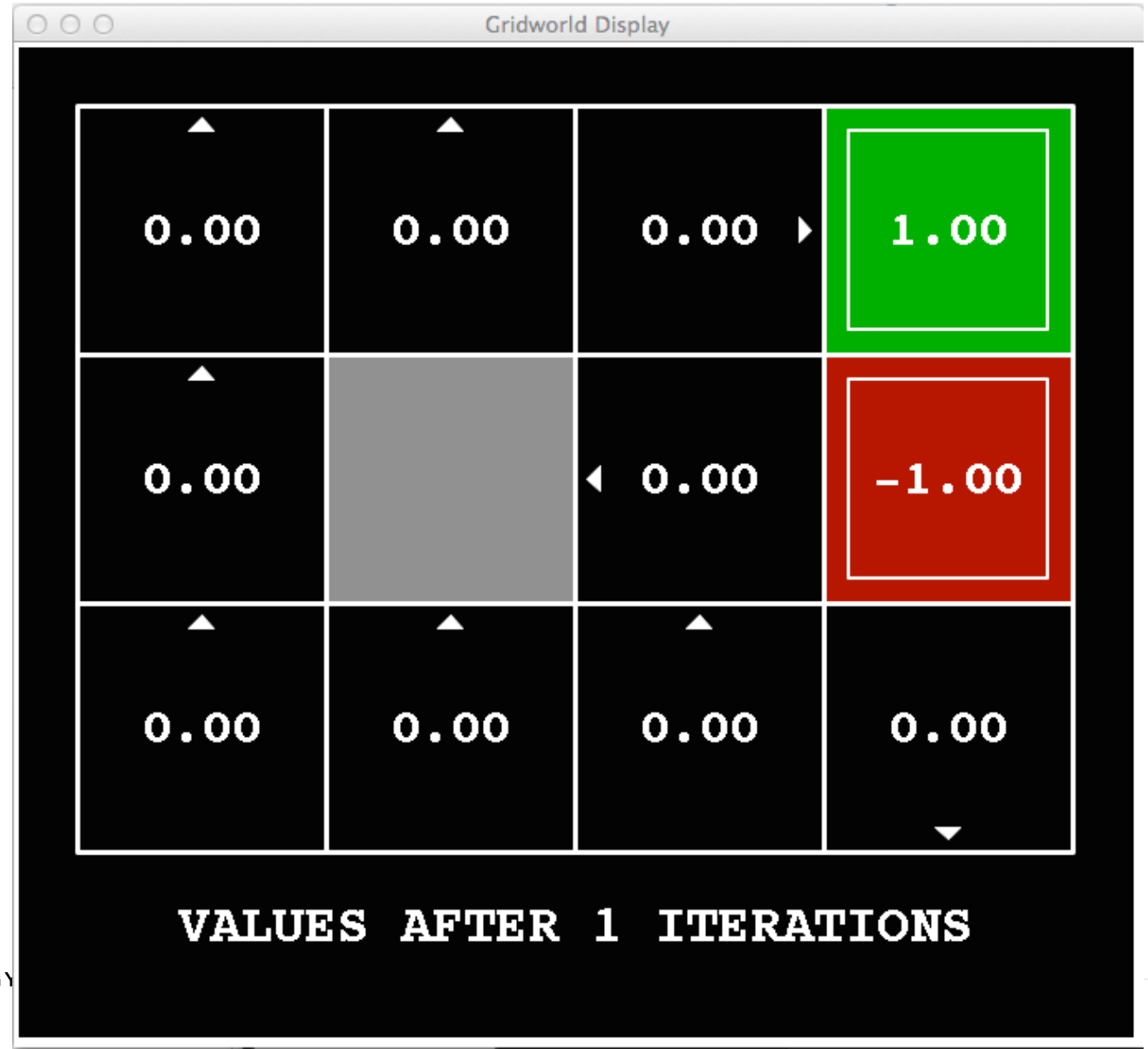


k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

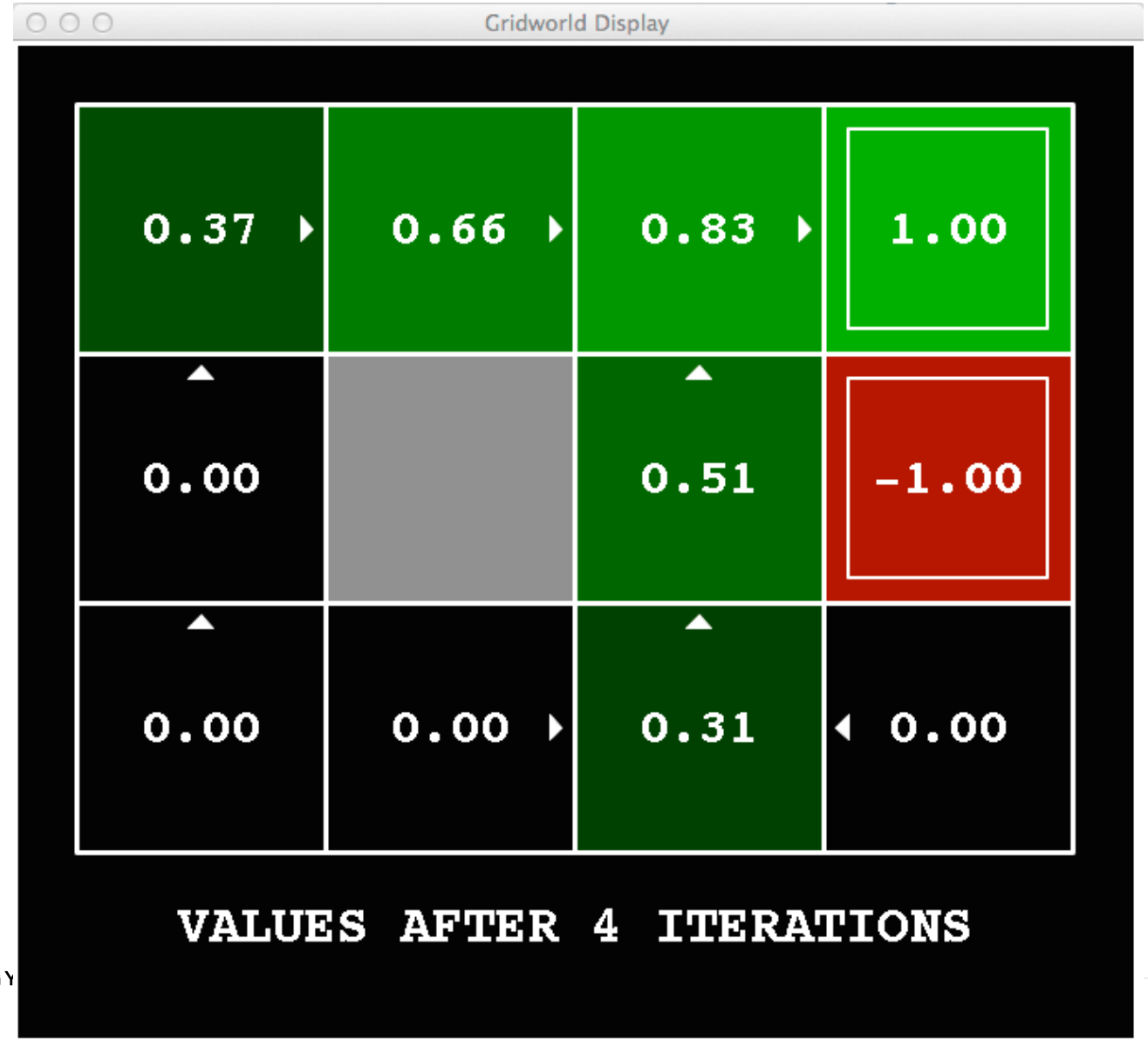
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

Noise = 0.2
Discount = 0.9
Living reward = 0

k=4

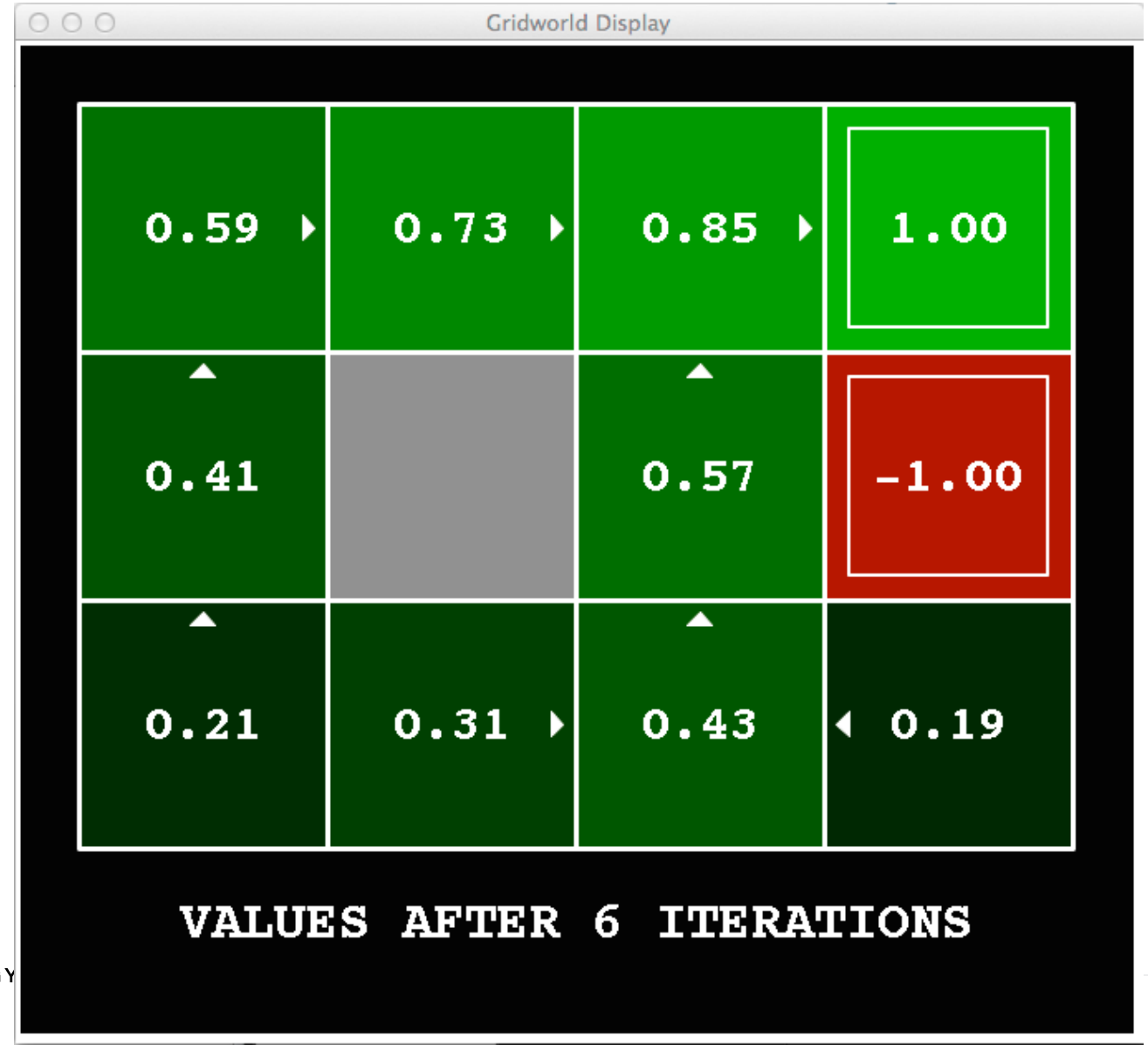


Noise = 0.2
Discount = 0.9
Living reward = 0

k=5

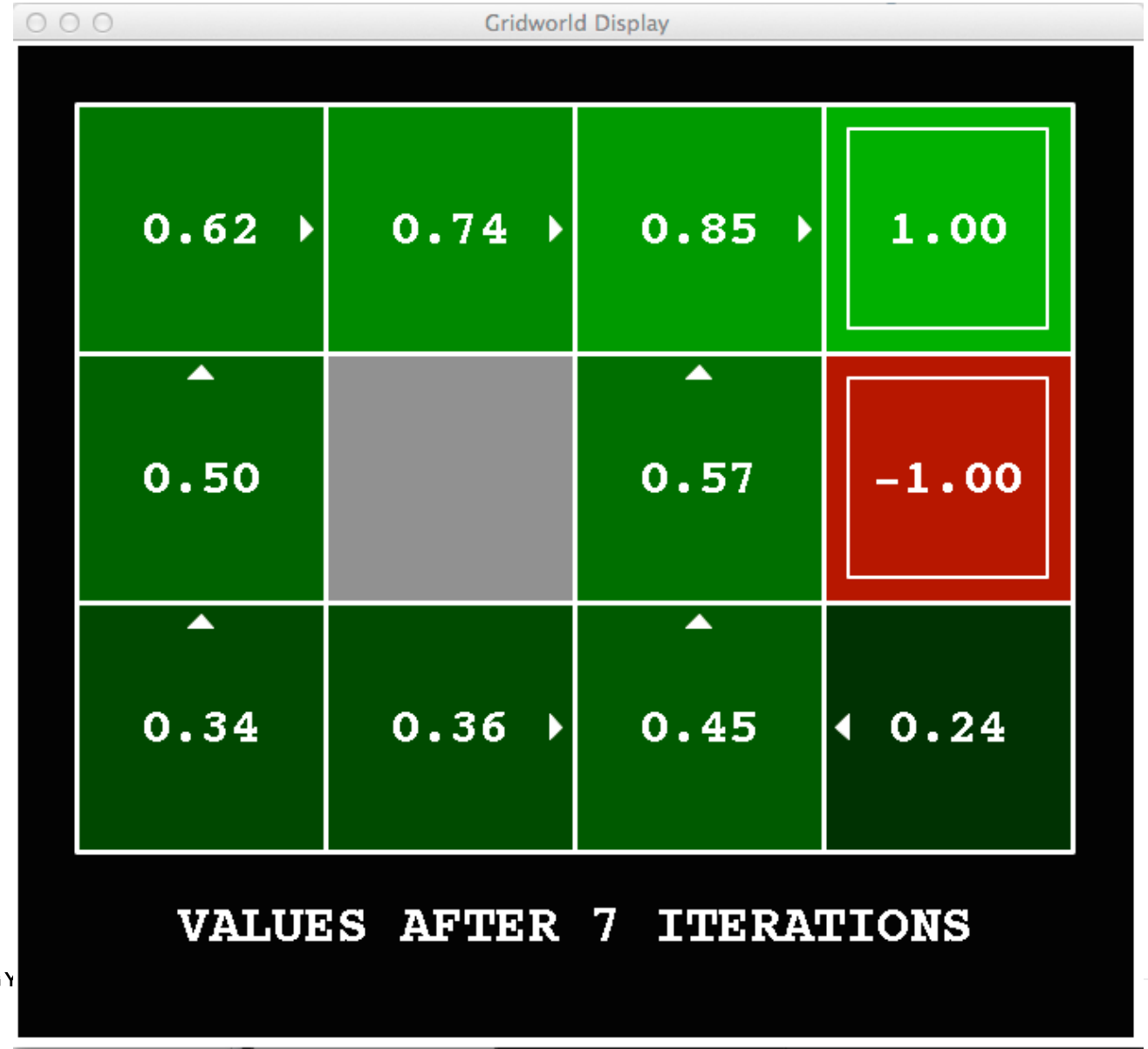
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



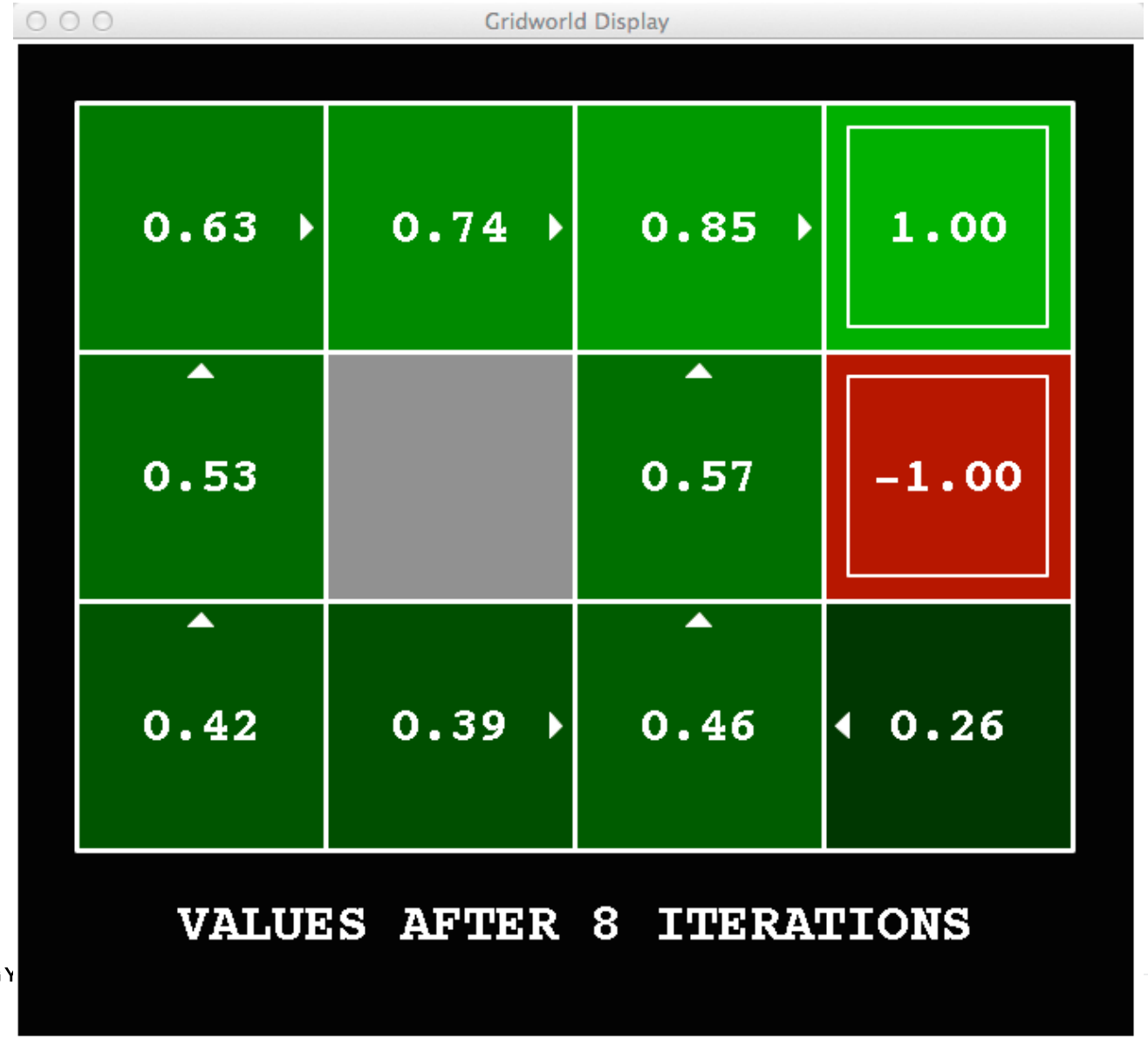
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



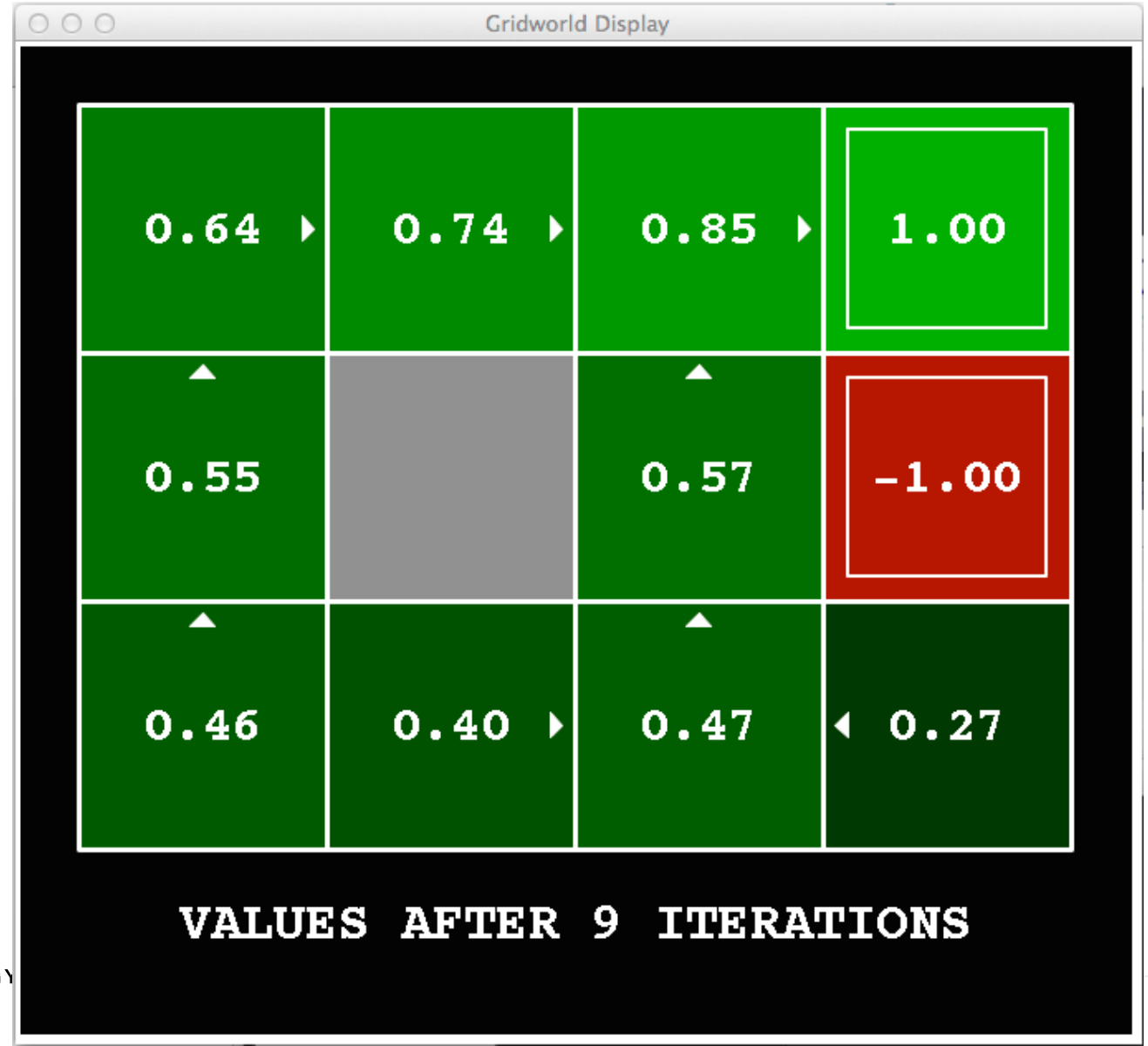
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



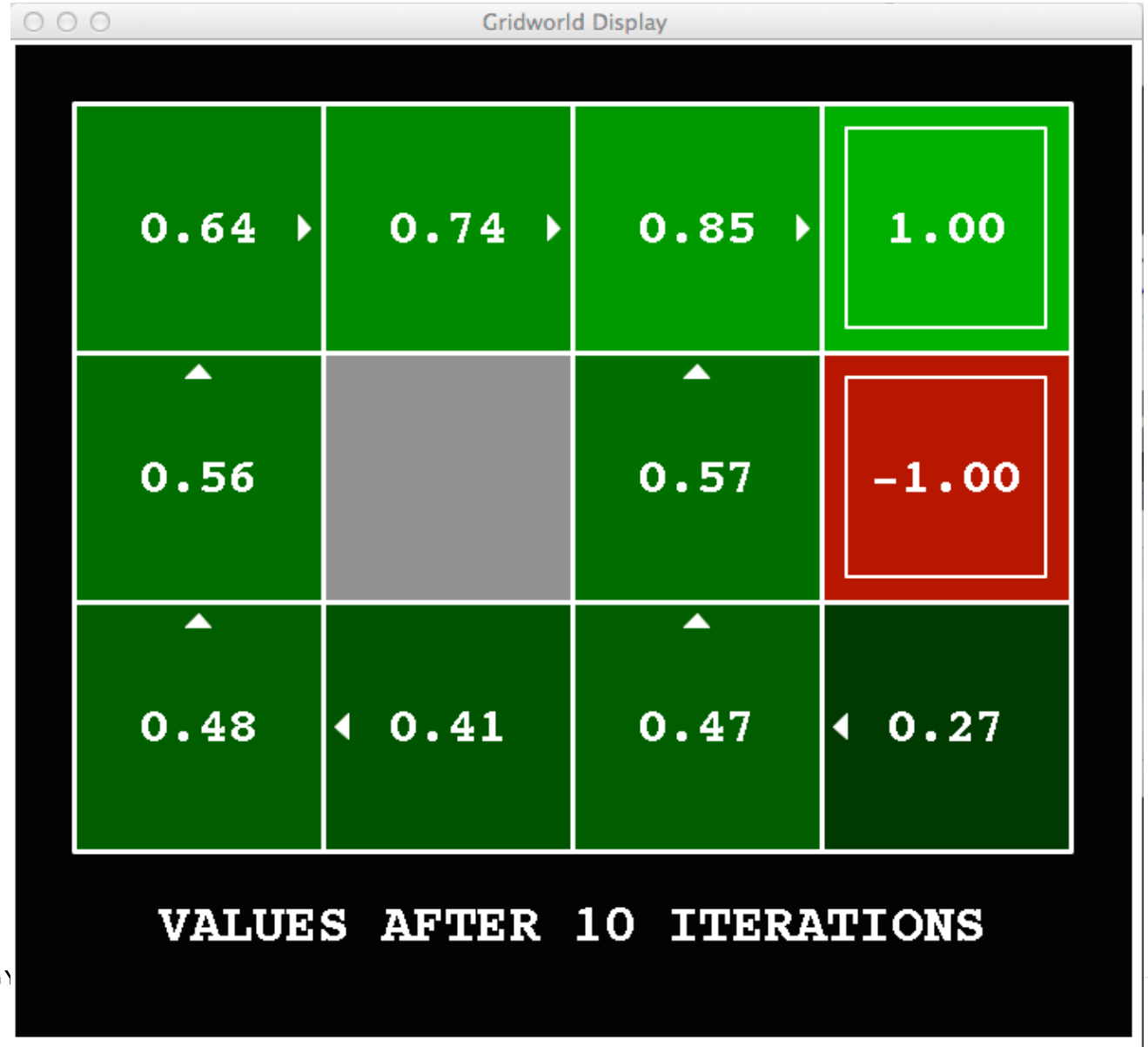
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



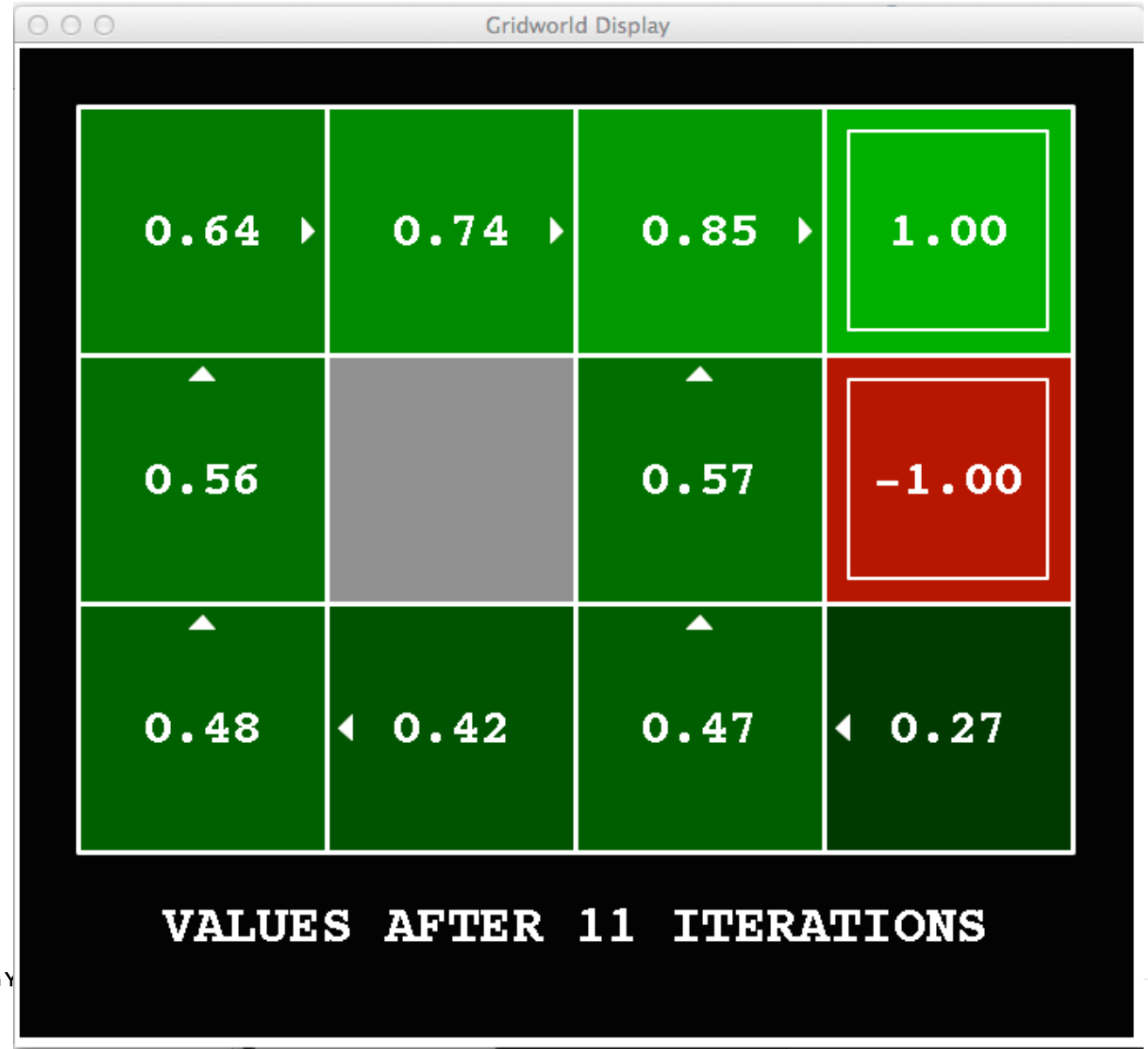
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



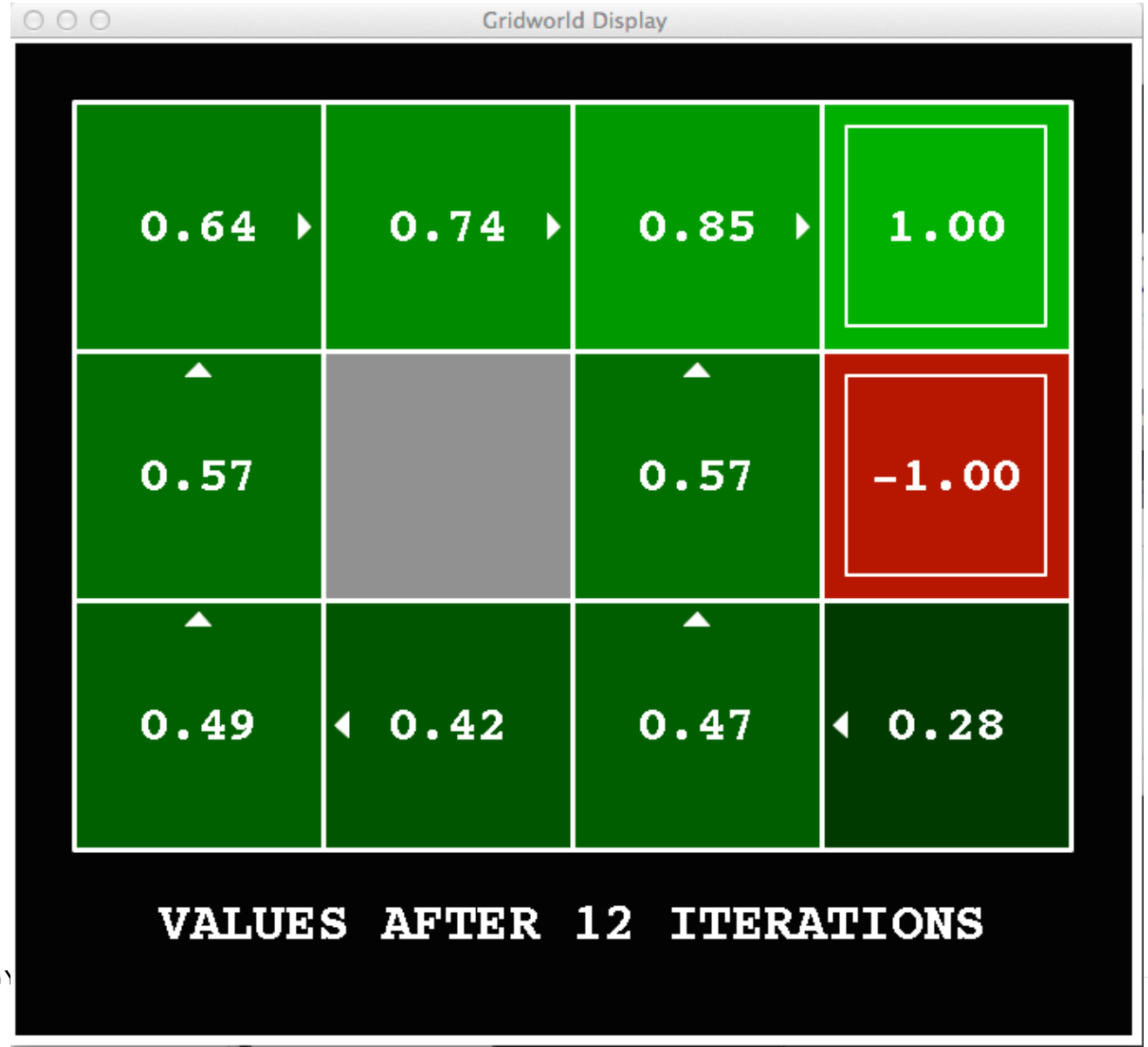
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



Noise = 0.2
Discount = 0.9
Living reward = 0

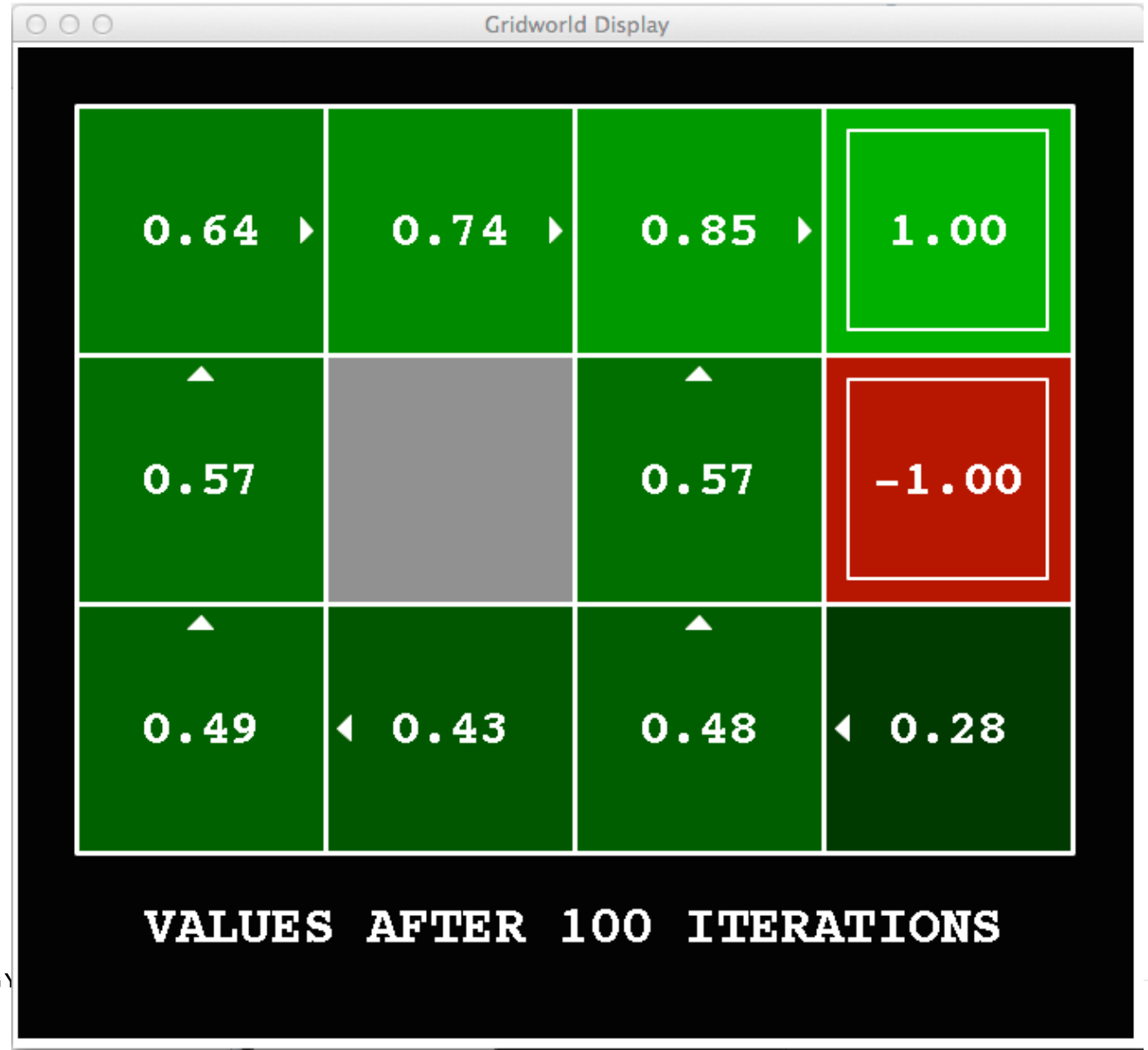
k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

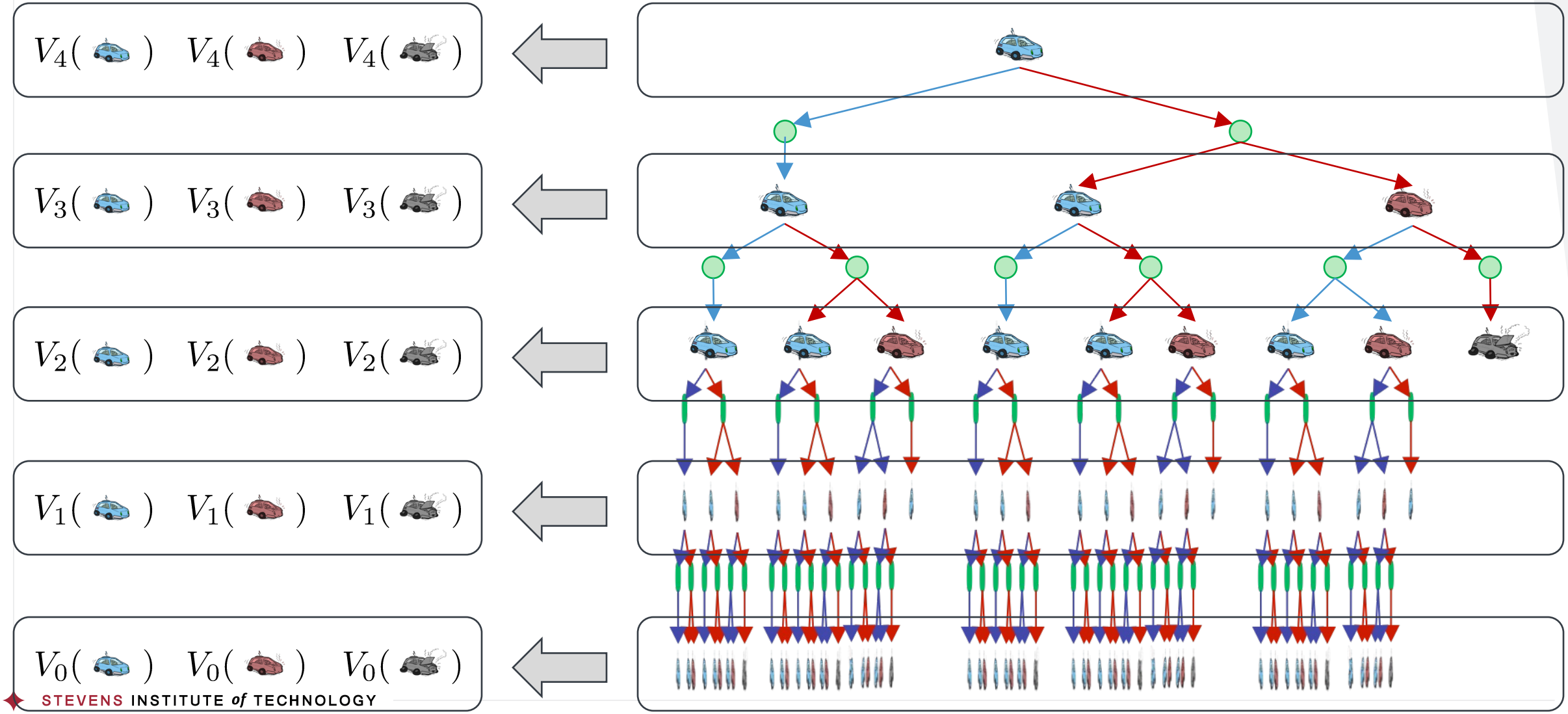
k=100

The values converge!



Noise = 0.2
Discount = 0.9
Living reward = 0

Computing Time-Limited Values

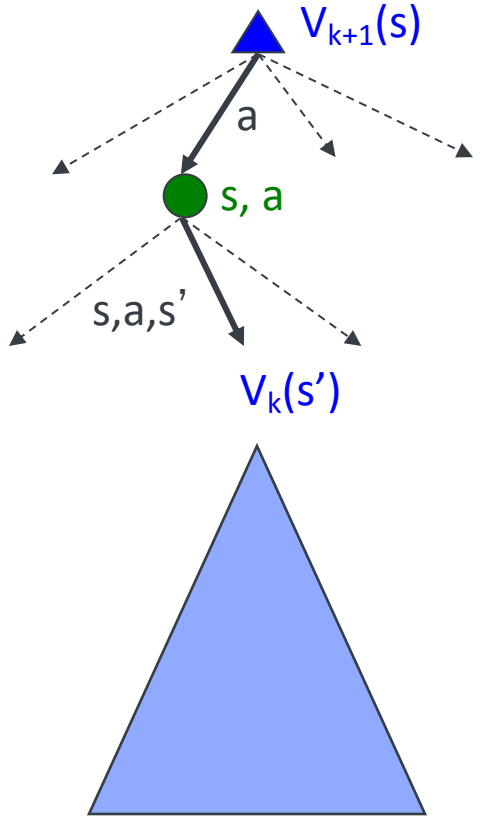


Value Iteration

Value Iteration

- 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:

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



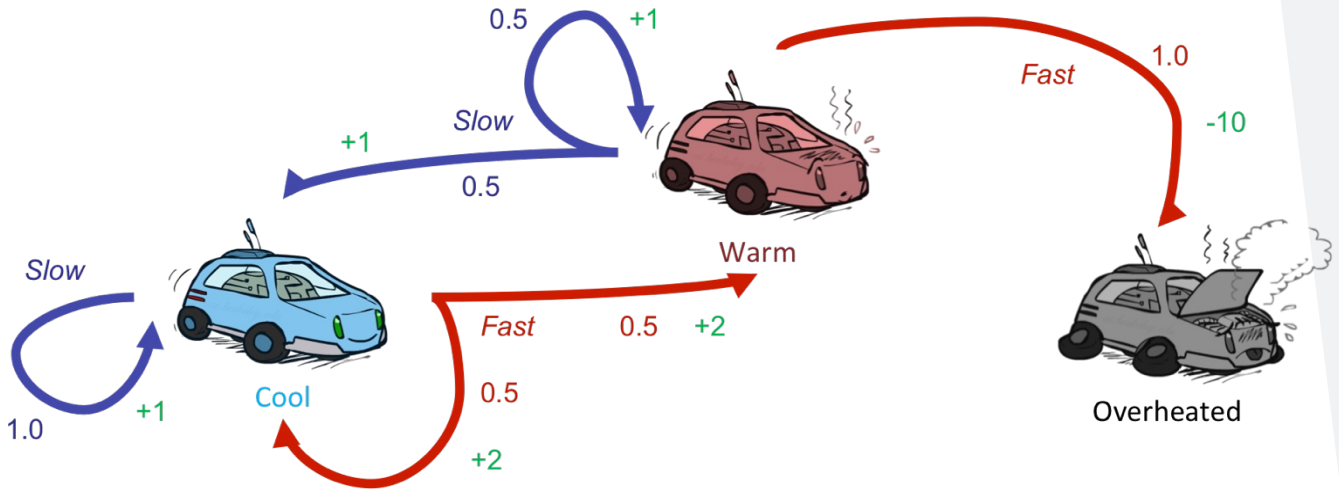
- Repeat until convergence
- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do

Example: Value Iteration

V_2

V_1 2 1 0

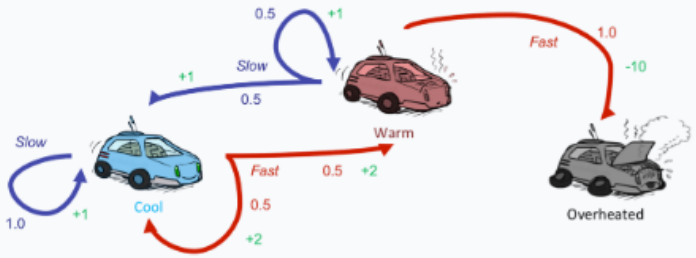
V_0 0 0 0



Assume no discount!

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

What are the values of V2(s)?



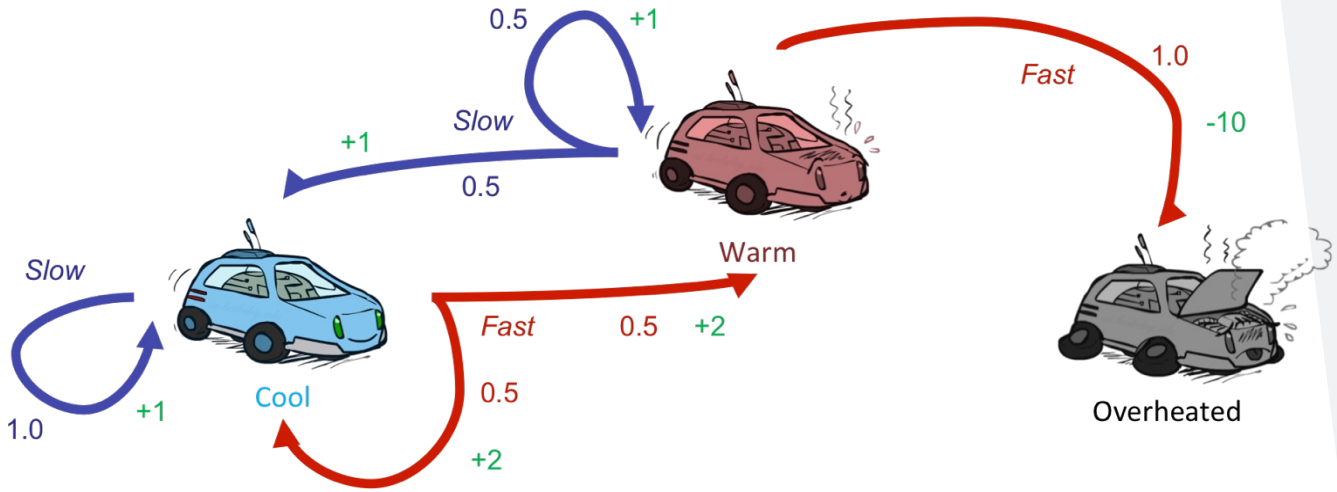
Assume no discount!

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

- 4, 2, 0
- 3.5, 2.5, 0
- 3.5, 2.5, 0.5
- 3, 2, 0

Example: Value Iteration

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



Assume no discount!

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

Example: Value Iteration

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

0	0	0	+1
0		0	-1
0	0	0	0

V_1

0	0	0.72	+1
0		0	-1
0	0	0	0

V_2

0	0.52	0.78	+1
0		x	-1
0	0	0	0

V_3

Noise = 0.2
Discount = 0.9
Living reward = 0

What is the value of x?

0	0.52	0.78	+1
0		x	-1
0	0	0	0

V_3

- 0.52
- 0.72
- 0.43
- 0.28
- None of the above

Example: Value Iteration

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

0	0	0	+1
0		0	-1
0	0	0	0

V_1

0	0	0.72	+1
0		0	-1
0	0	0	0

V_2

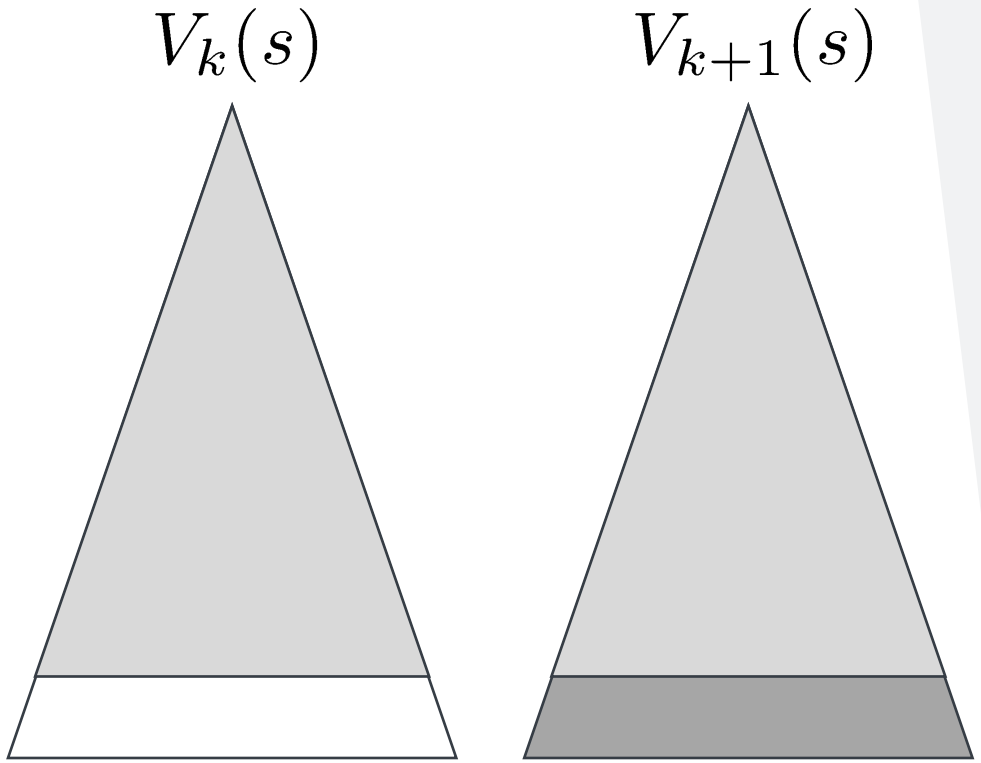
0	0.52	0.78	+1
0		0.43	-1
0	0	0	0

V_3

Noise = 0.2
Discount = 0.9
Living reward = 0

Convergence

- How do we know the V_k vectors are going to converge?
- Case 1: If the tree has maximum depth M, then V_M holds the actual untruncated values
- Case 2: If the discount is less than 1
  - 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 γ^k max|R| different
  - So as k increases, the values converge



STEVENS
INSTITUTE OF TECHNOLOGY
1870

15 min. break

Optimal Quantities

- The value (utility) of a state s:

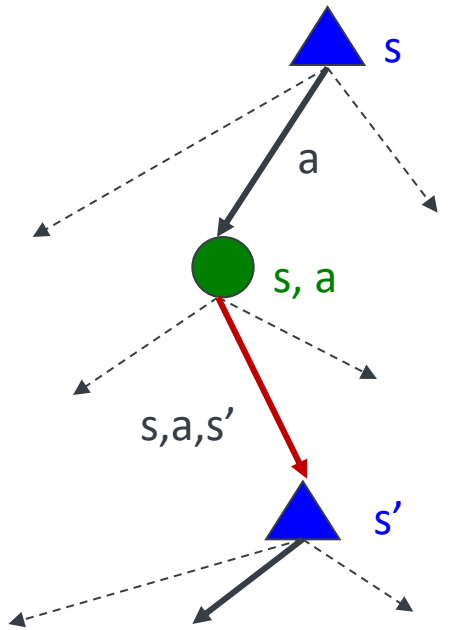
  $V^*(s)$ = expected utility starting in s and acting optimally

- 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*

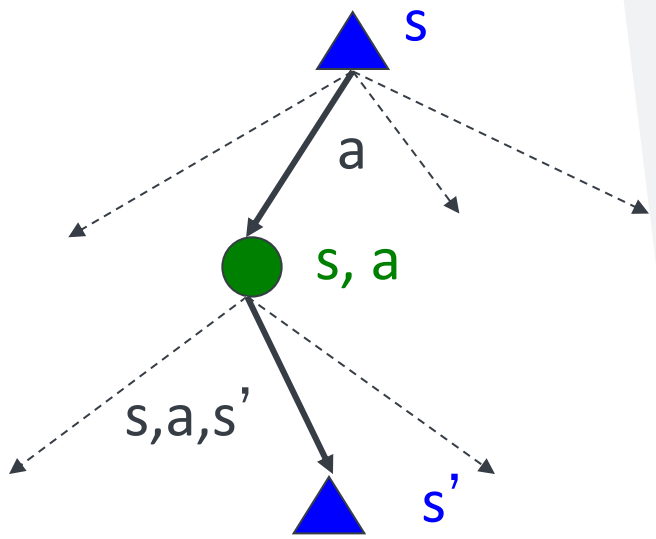
The Bellman Equations

- Definition of “optimal utility” via expectimax recurrence gives a simple one-step lookahead relationship amongst optimal utility values

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

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

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

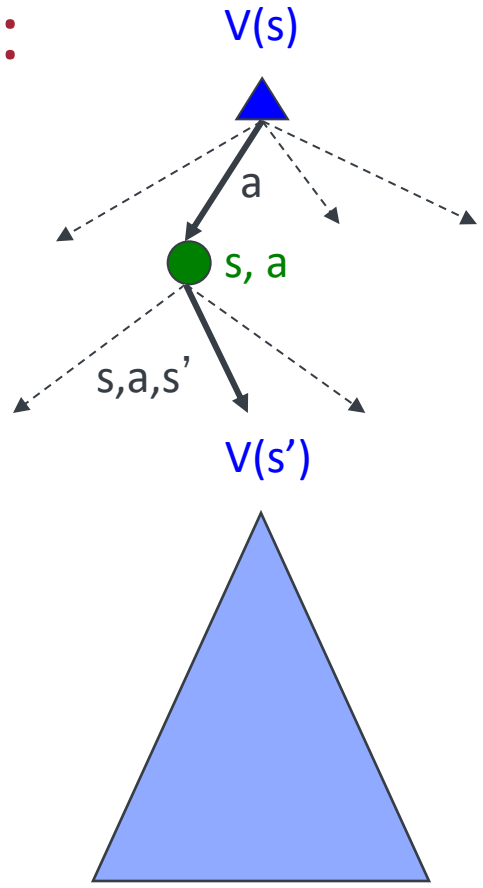


- These are the Bellman equations, and they characterize optimal values in a way we’ll use over and over

Value Iteration

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

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

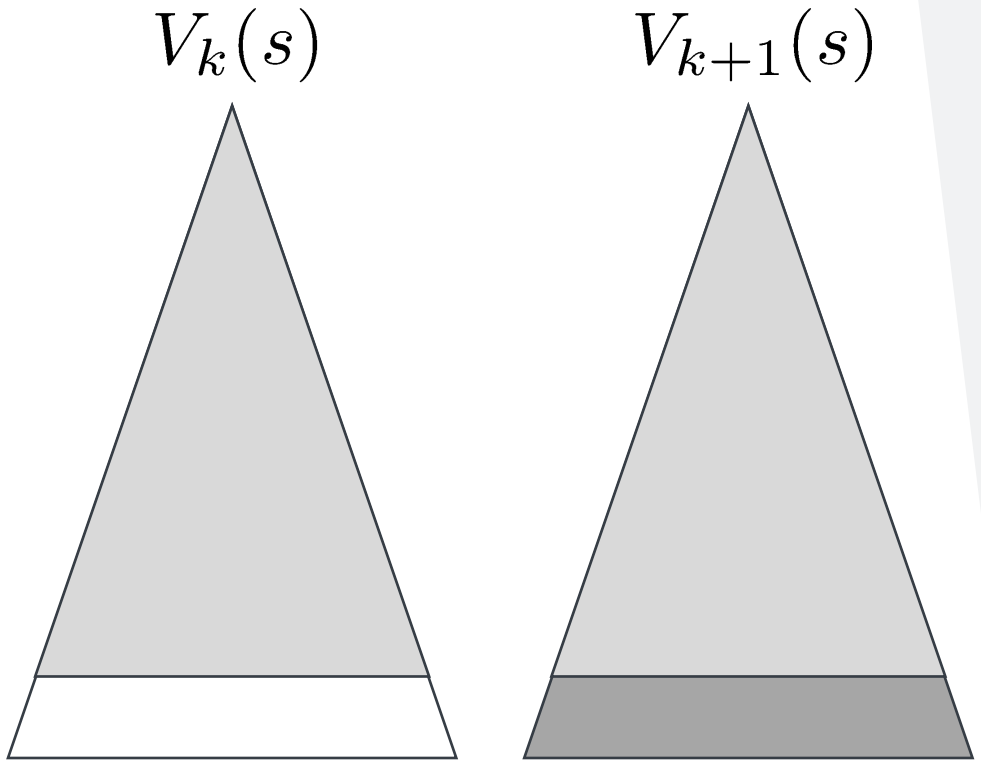


- Value iteration **computes** them:

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

Convergence

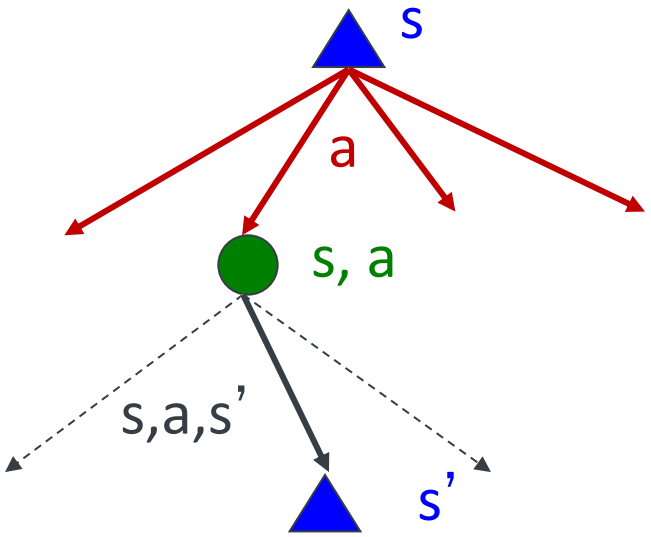
- How do we know the V_k vectors are going to converge?
- Case 1: If the tree has maximum depth M, then V_M holds the actual untruncated values
- Case 2: If the discount is less than 1
  - Sketch: For any state V_k and V_{k+1} can be viewed as depth k+1 expectimax results in nearly identical search trees
  - 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 γ^k max|R| different
  - So as k increases, the values converge



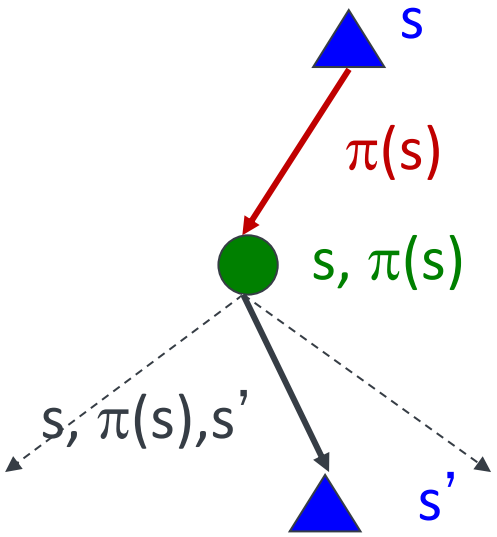
Policy Evaluation

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 π(s), then the tree would be simpler – only one action per state
  - … though the tree's value would depend on which policy we fixed

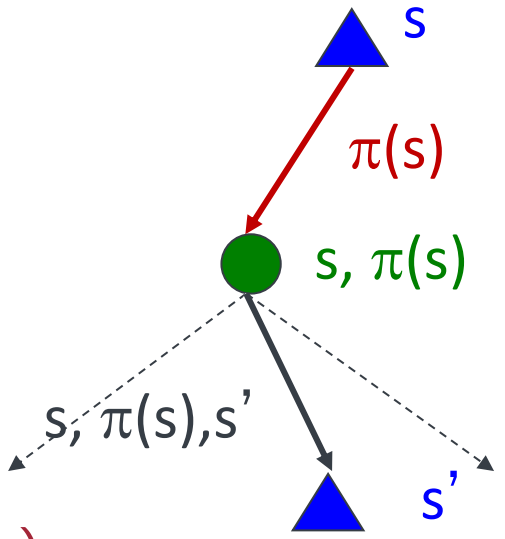
Utilities for a Fixed Policy

- 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):

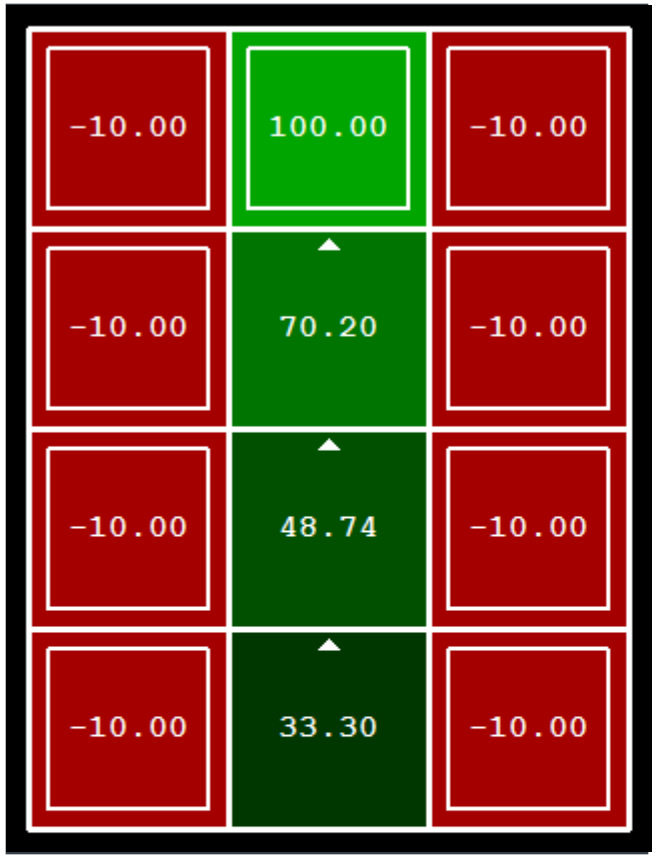


$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

Example: Policy Evaluation

Always Go Right

Always Go Forward

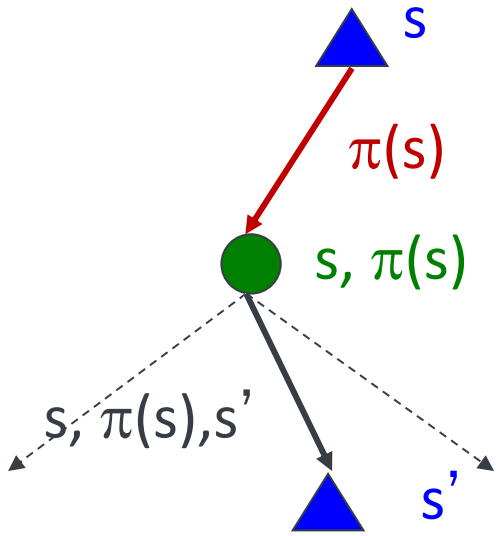


Policy Evaluation

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

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

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

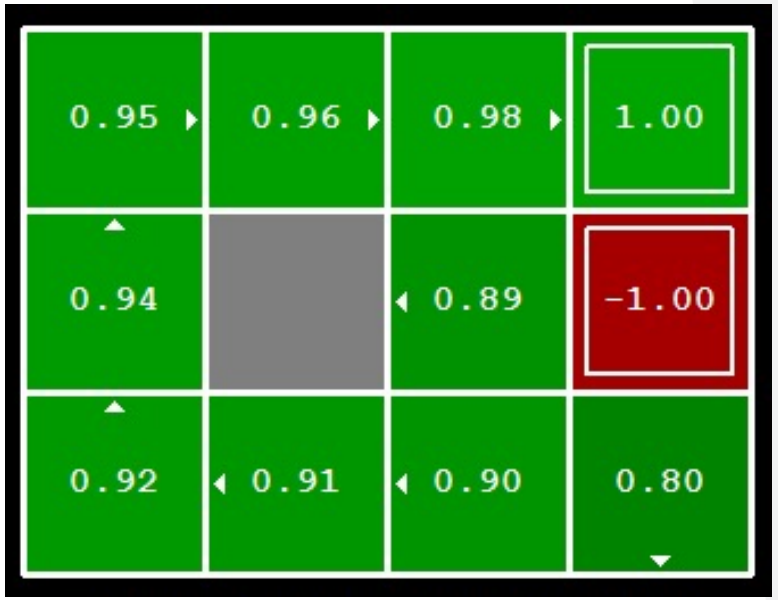


- Efficiency: $O(S^2)$ per iteration
- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve with Matlab (or your favorite linear system solver)

Policy Extraction

Computing Actions from Values

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



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

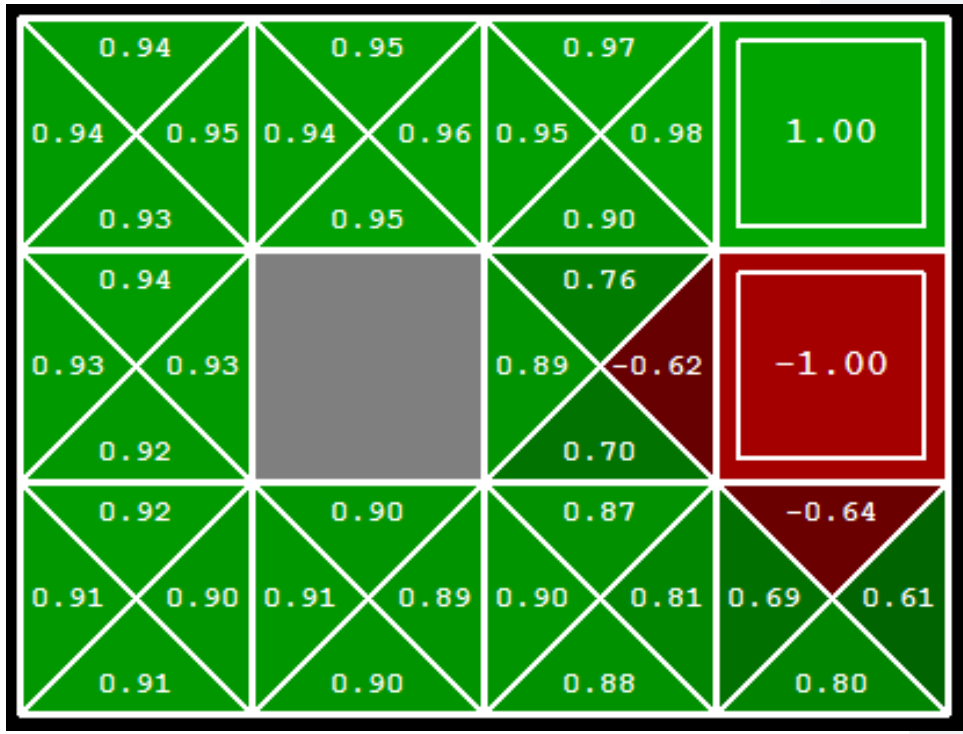
- We need to do a mini-expectimax (one step)

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

- 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!

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

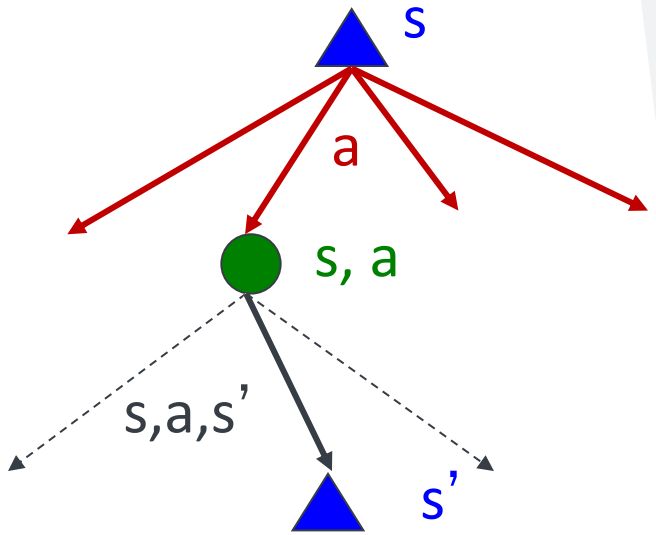
- Important lesson: actions are easier to select from q-values than values!

Policy Iteration

Problems with Value Iteration

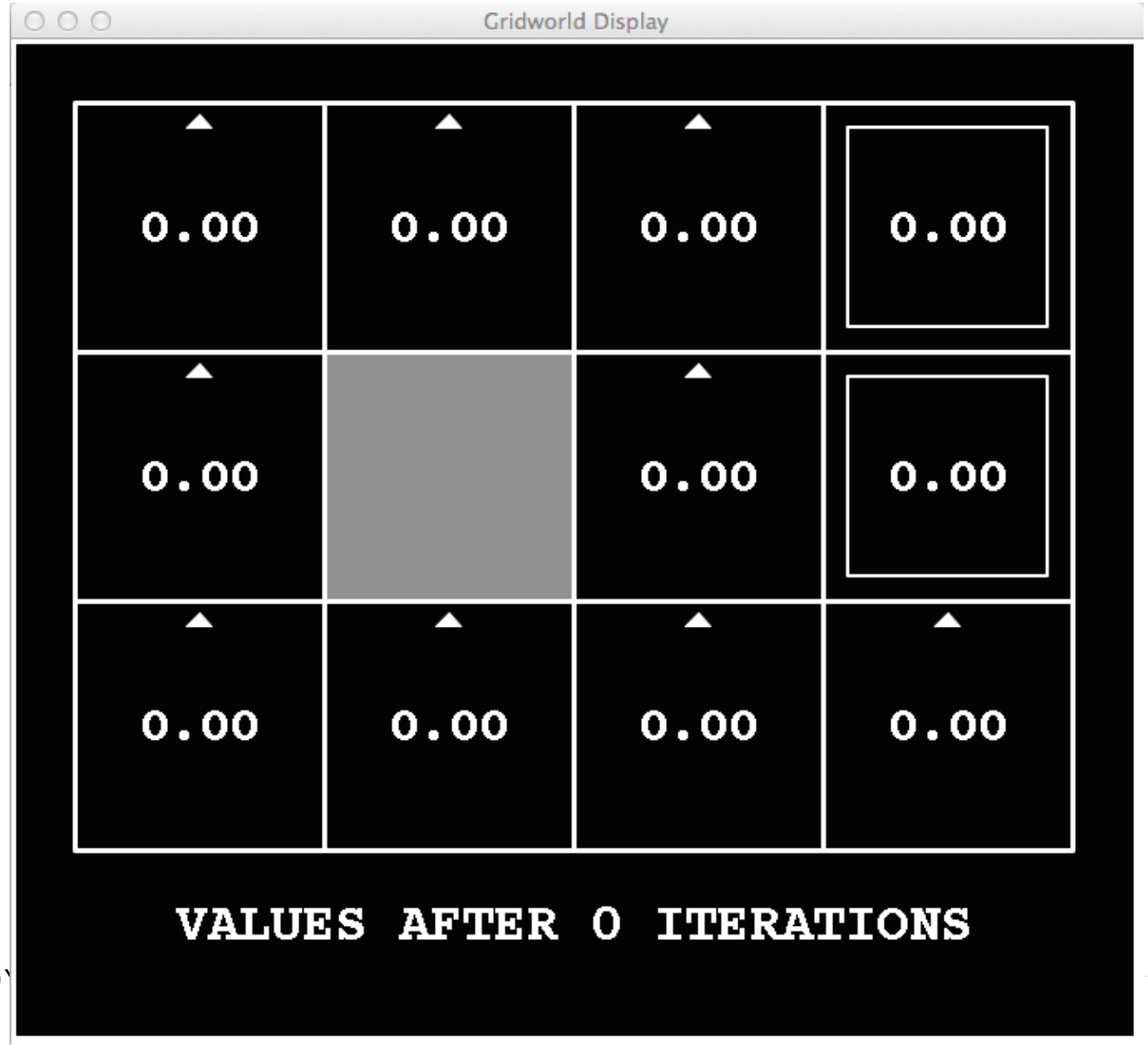
- Value iteration repeats the Bellman updates:

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



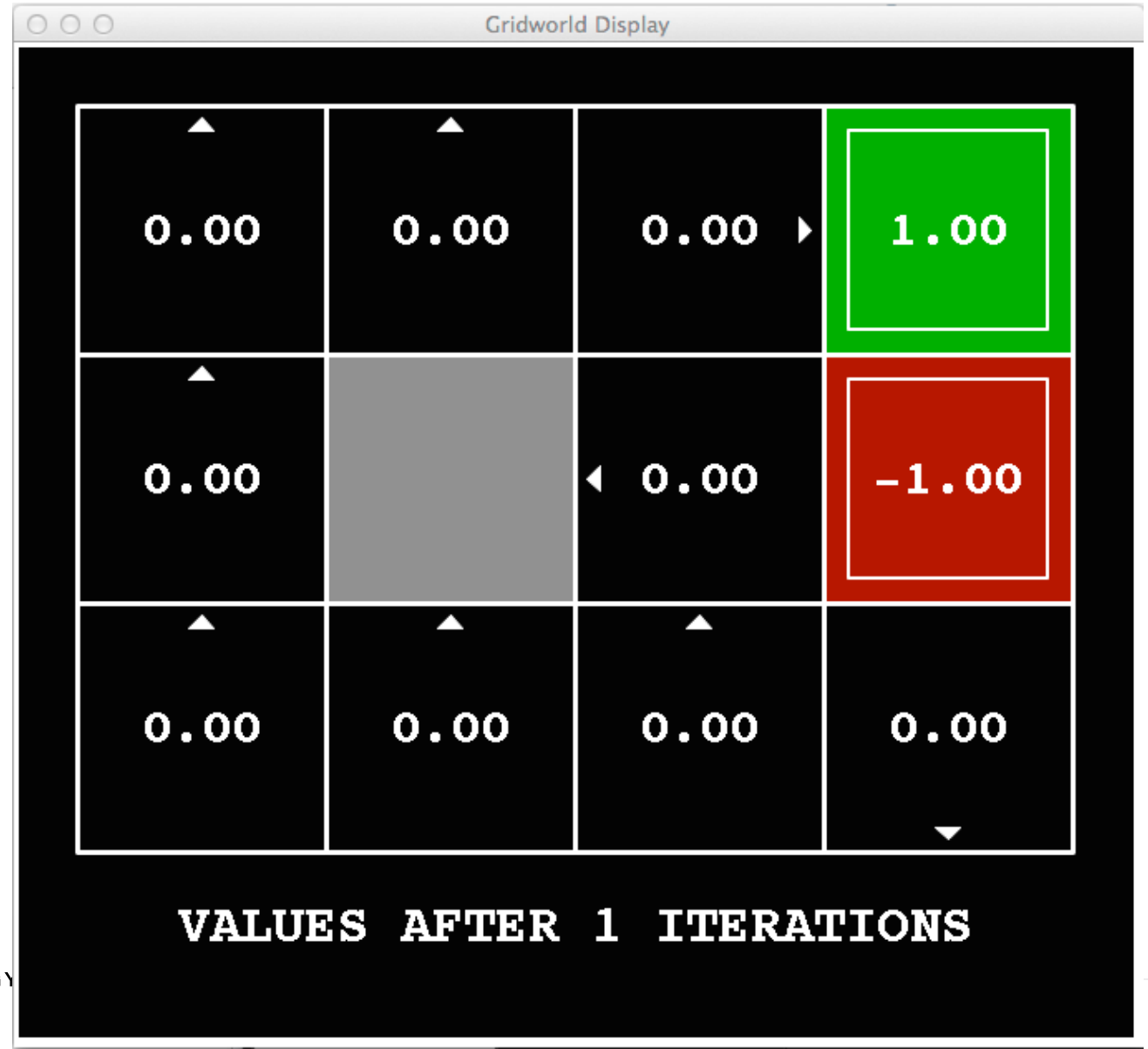
- Problem 1: It's slow – $O(S^2A)$ per iteration
- Problem 2: The "max" at each state rarely changes
- Problem 3: The policy often converges long before the values

k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

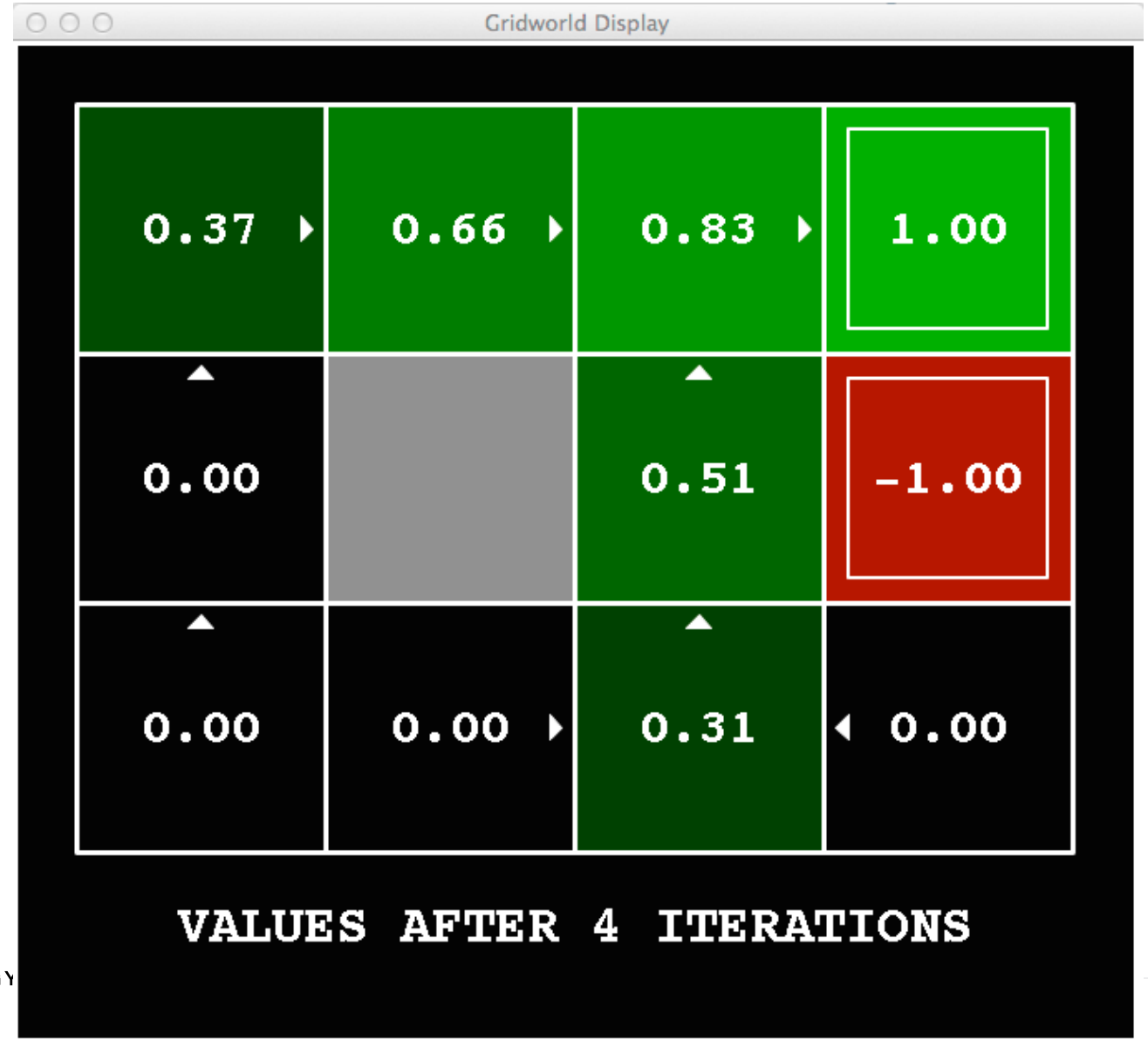
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

Noise = 0.2
Discount = 0.9
Living reward = 0

k=4

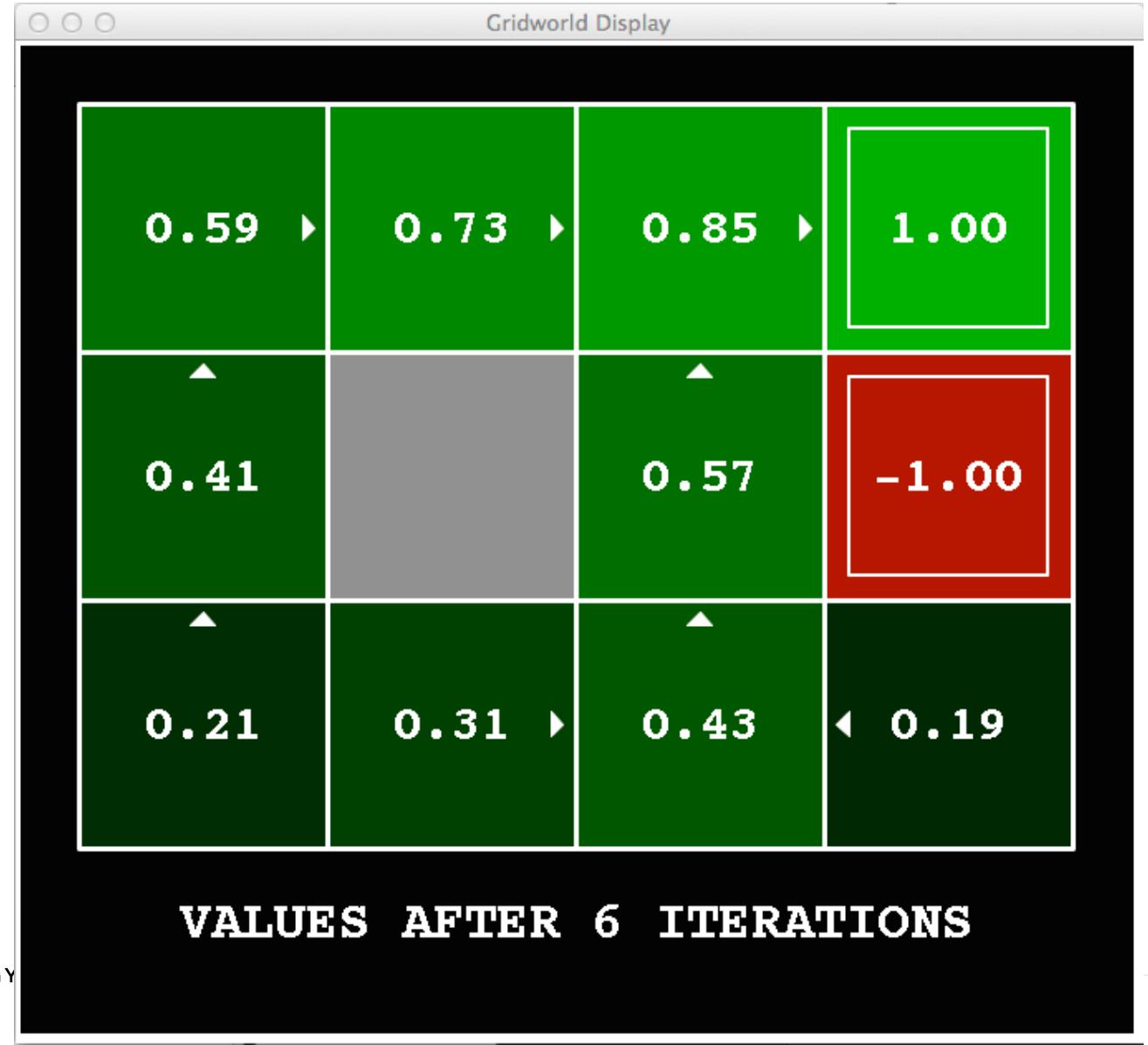


Noise = 0.2
Discount = 0.9
Living reward = 0

k=5

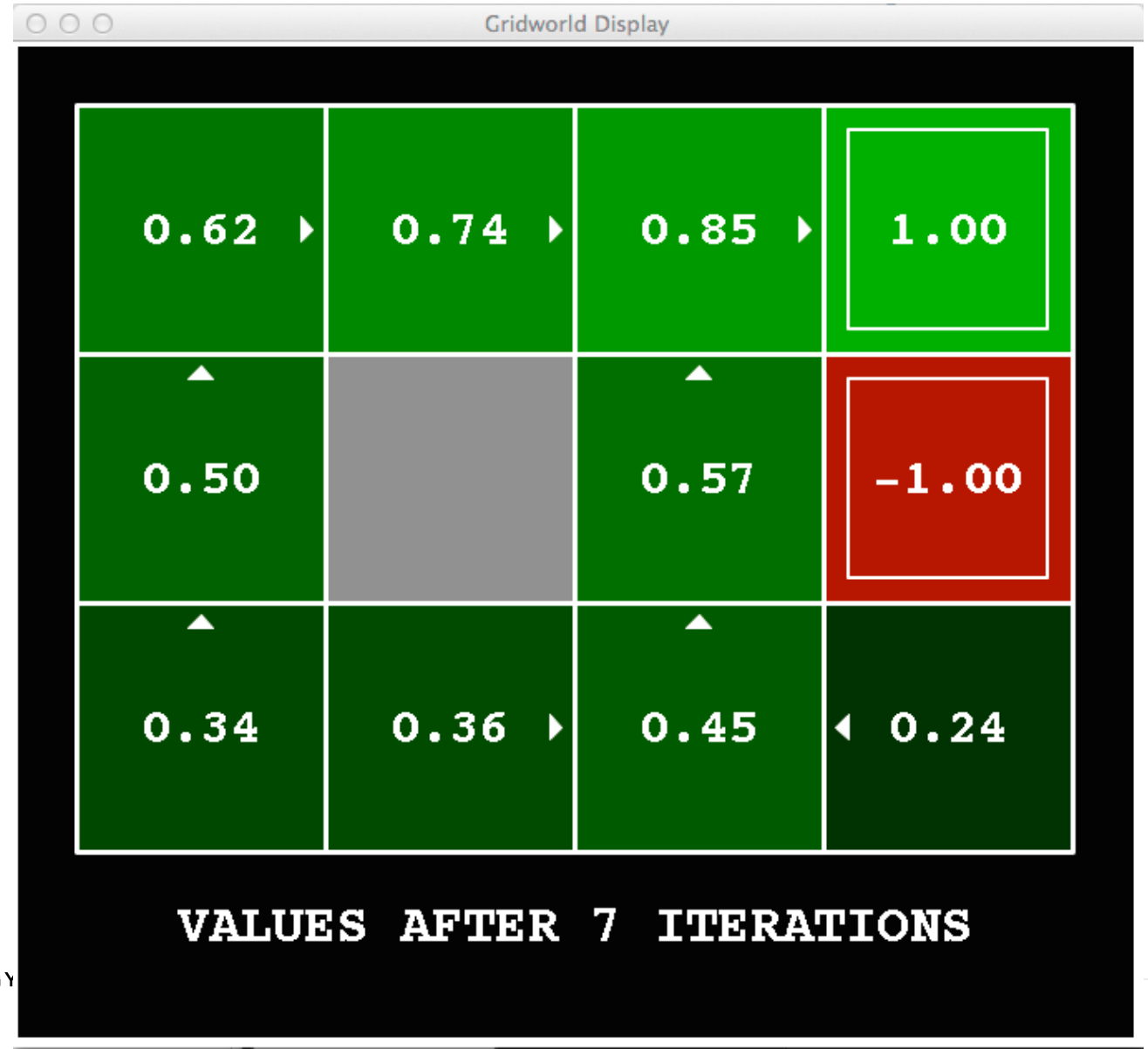
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



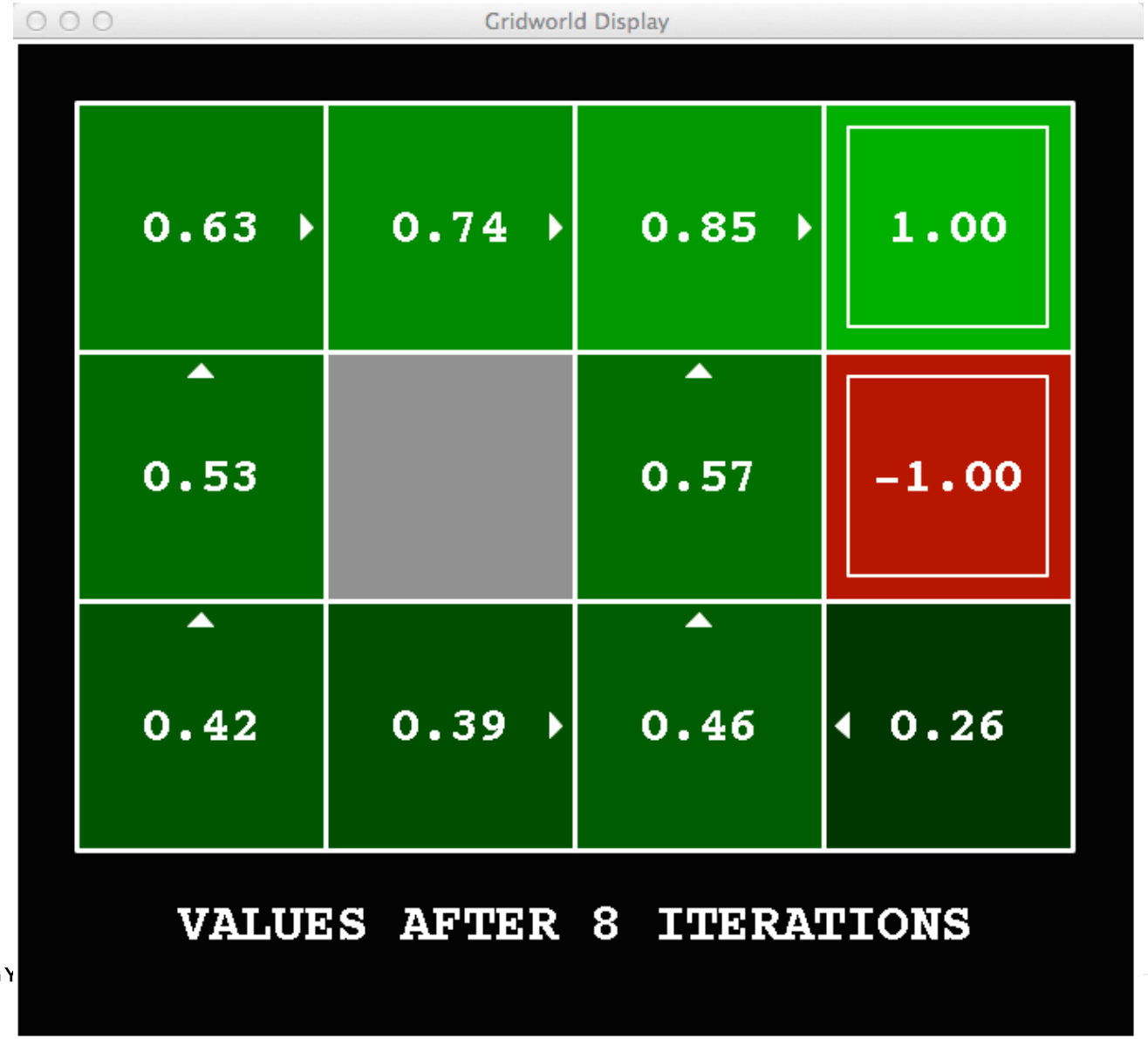
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



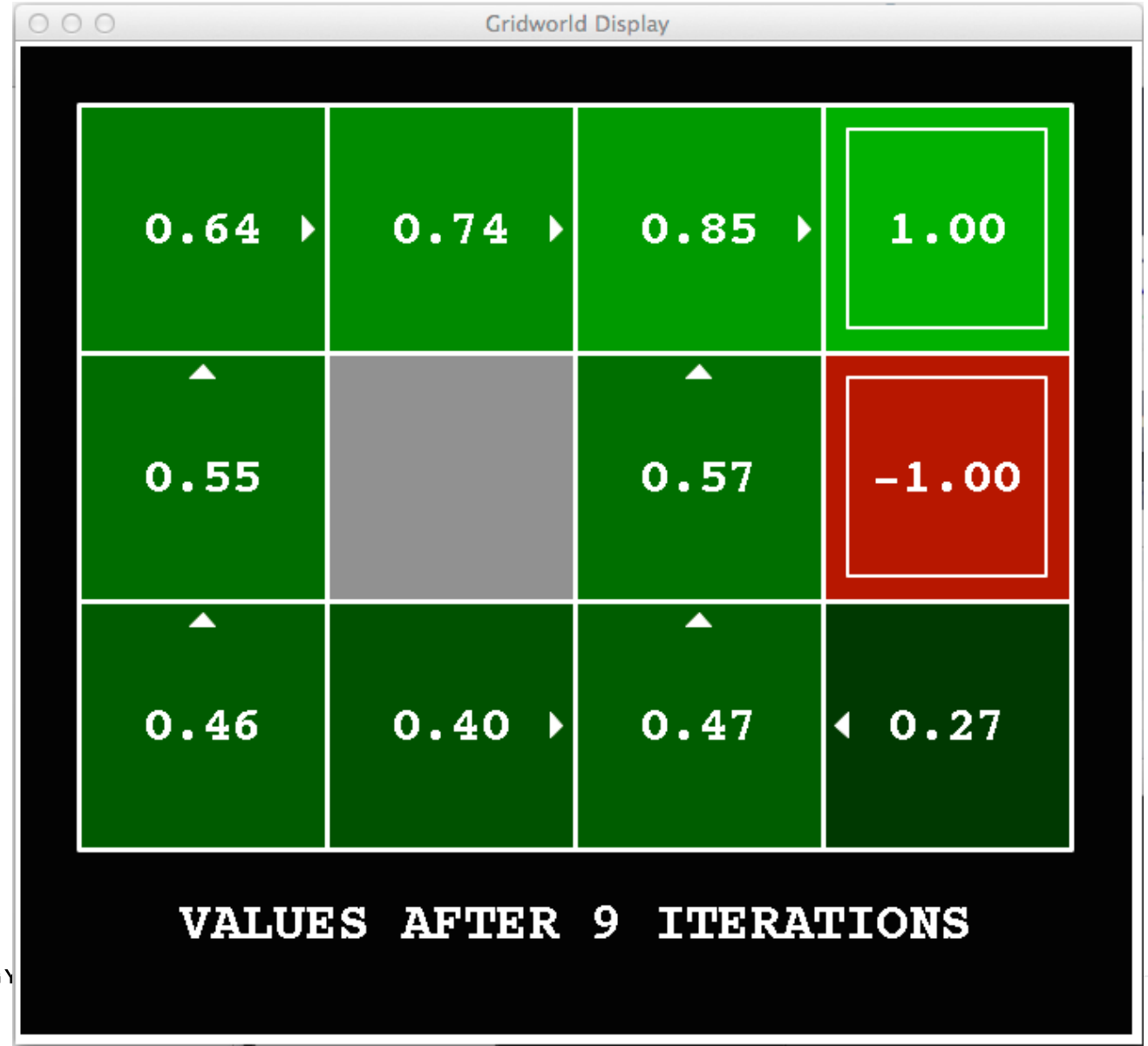
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



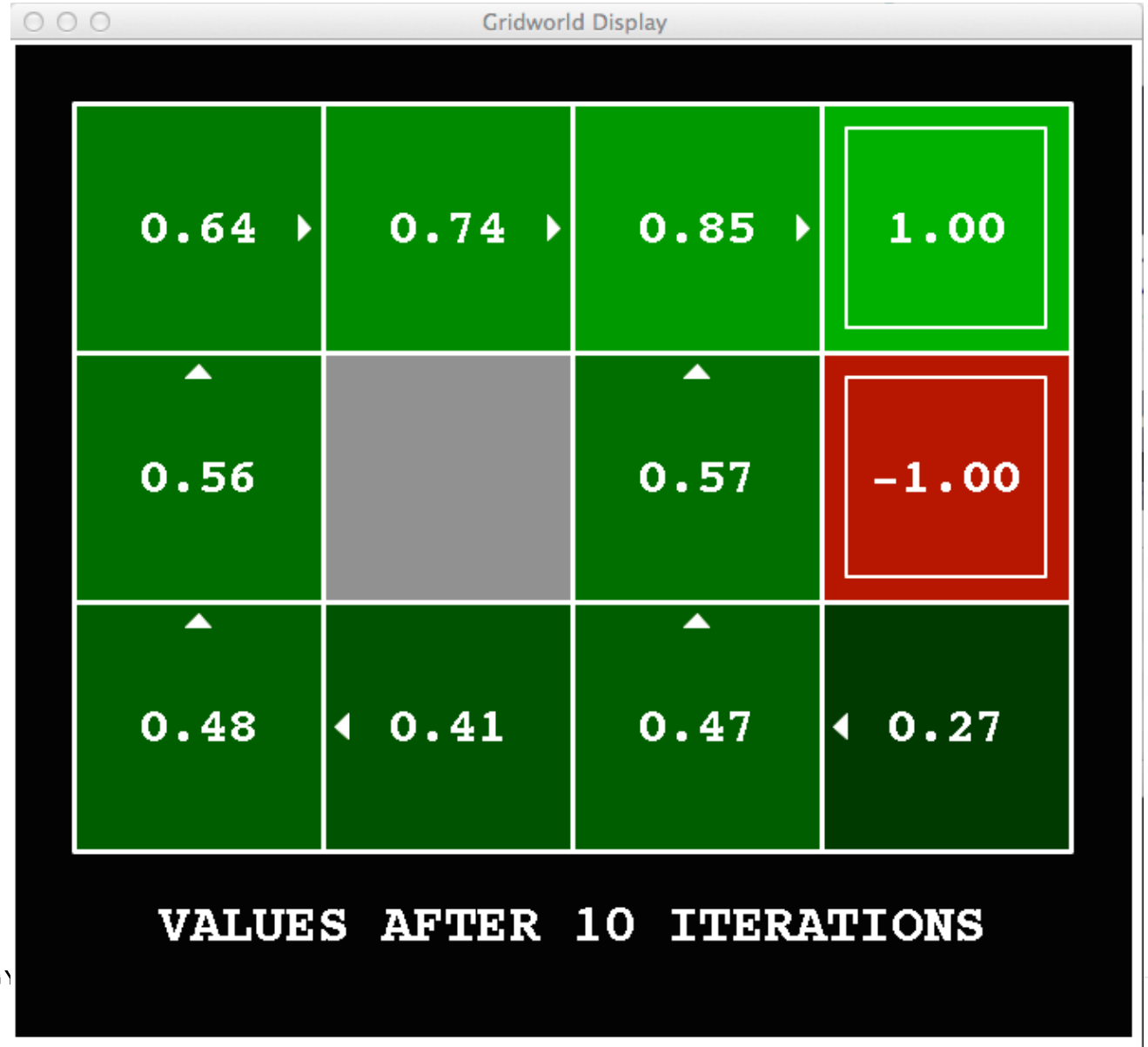
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



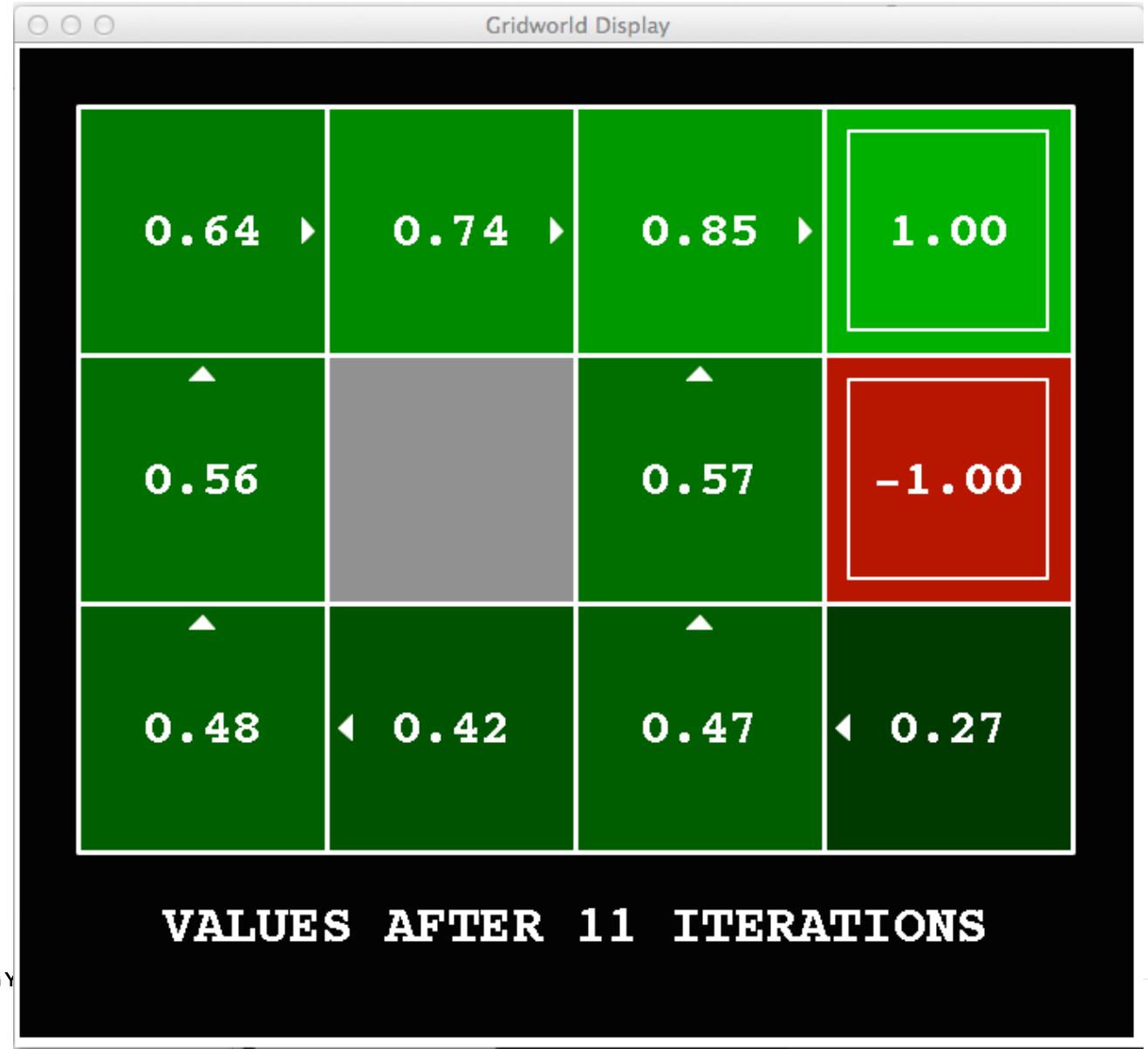
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



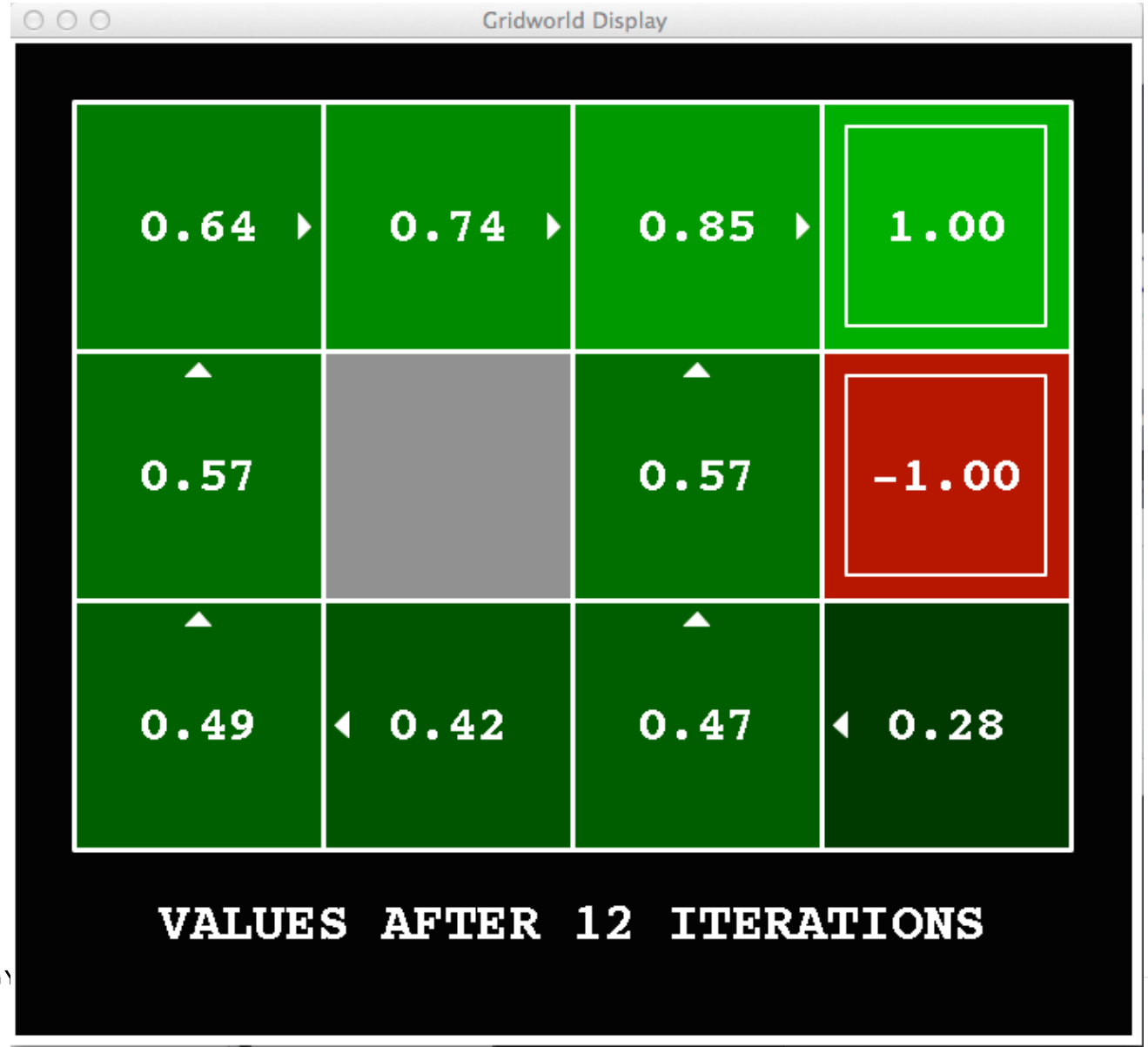
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



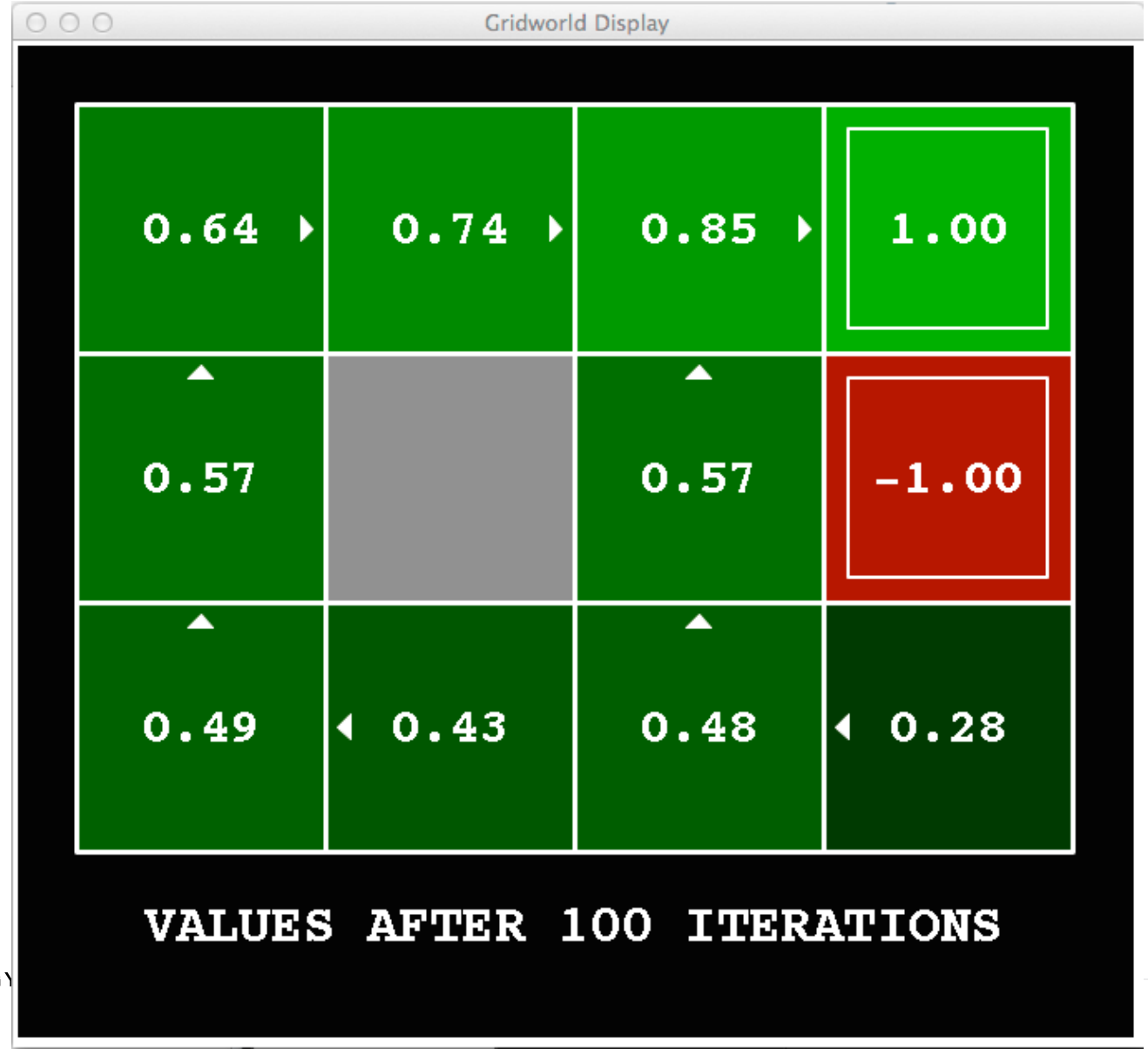
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

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 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

- Value iteration

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

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

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

- Policy iteration

Evaluation

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

Improvement

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

Comparison

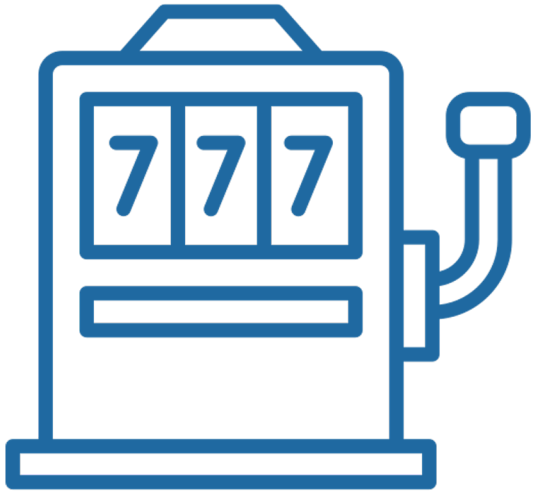
- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don’t track the policy, but taking the max over actions implicitly recomputes it
- In policy iteration:
  - We do 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 programs for solving MDPs

Summary: MDP Algorithms

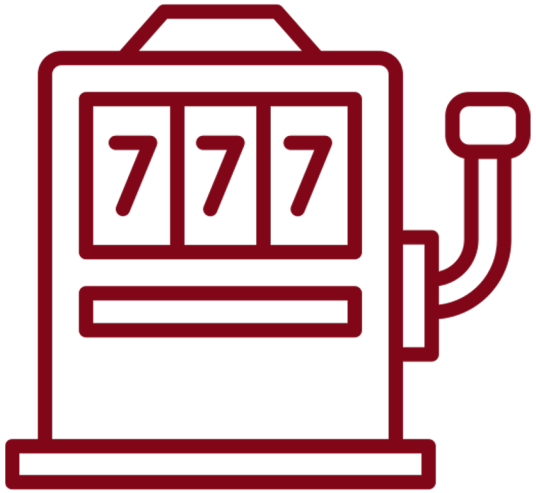
- So you want to....
  - Compute optimal values: use value iteration or policy iteration
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)

- 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



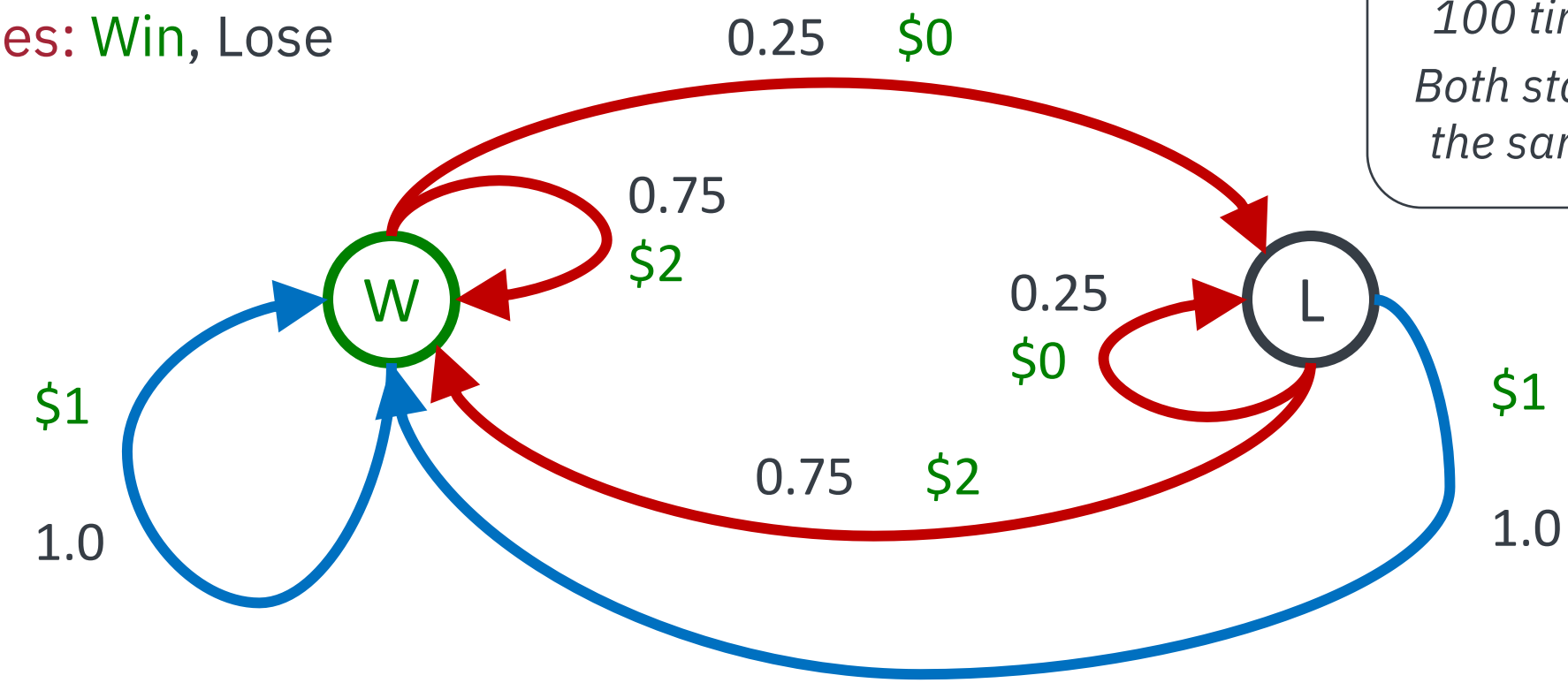
$1: 100%



$2: 75%
$0: 25%

Double-Bandit MDP

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



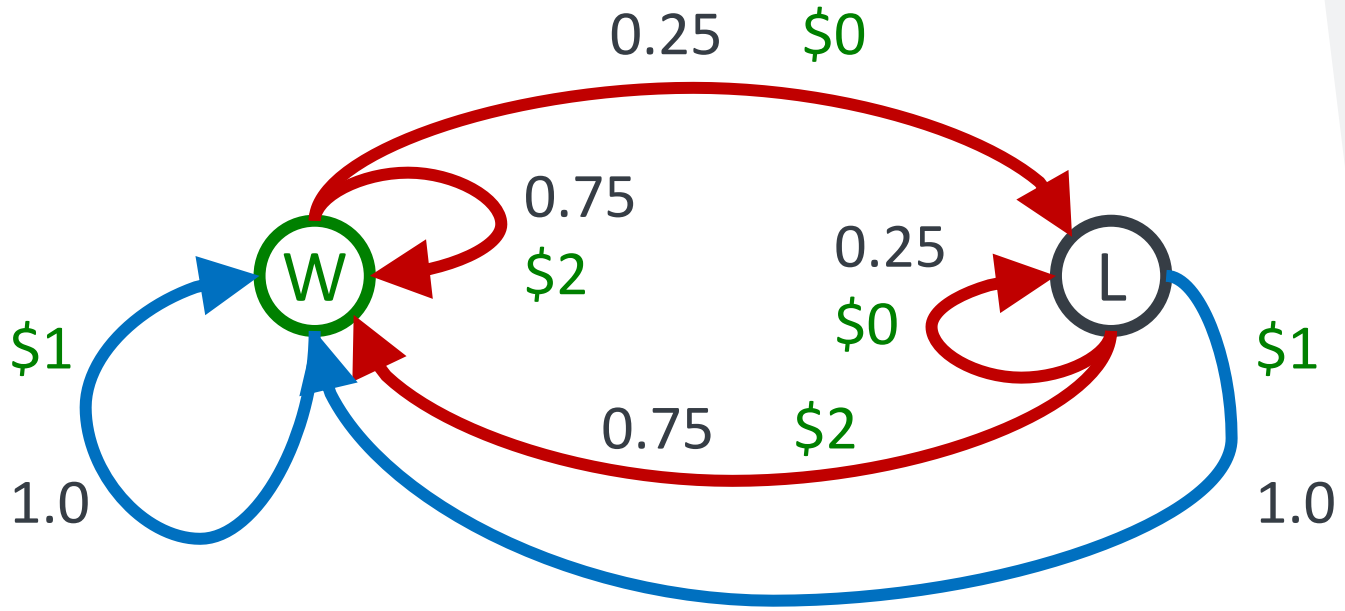
No discount
100 time steps
Both states have the same value

Offline Planning

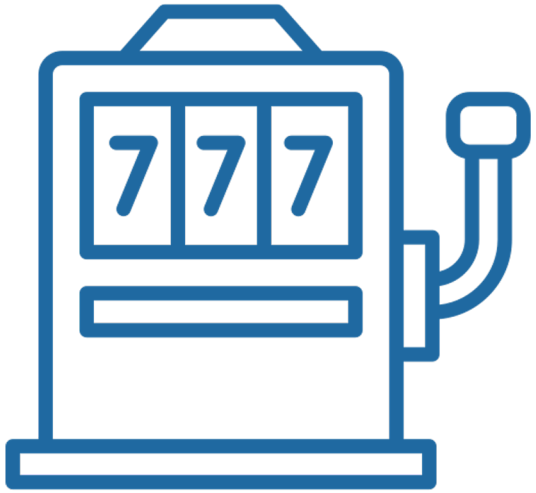
- Solving MDPs is offline planning
  - You determine all quantities through computation
  - You need to know the details of the MDP
  - You do not actually play the game!

No discount
100 time steps
Both states have the same value

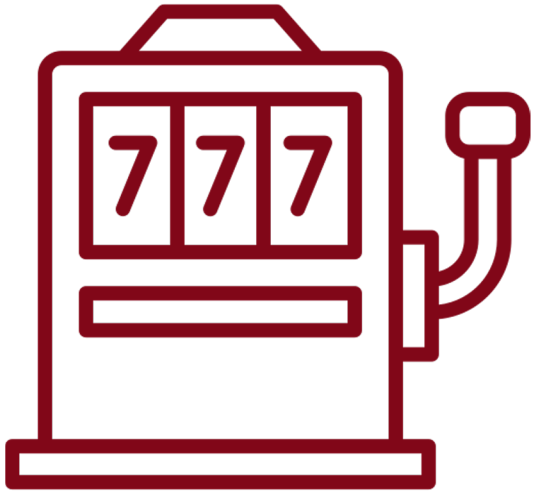
	Value
Play Red	150
Play Blue	100



Let’s Play!



$1: 100%



$2: 75%
$0: 25%

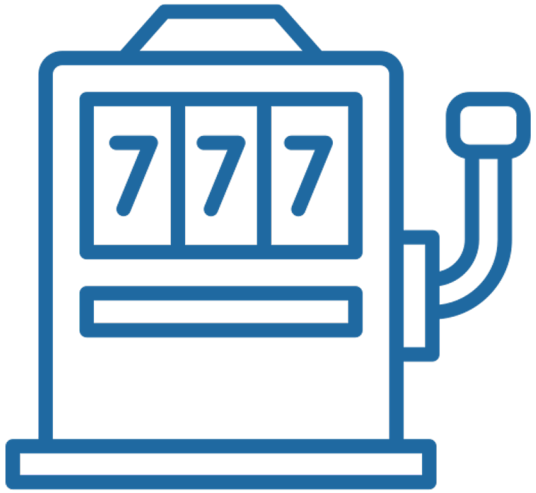
$2 $2 $0 $2 $2

$2 $2 $0 $0 $0

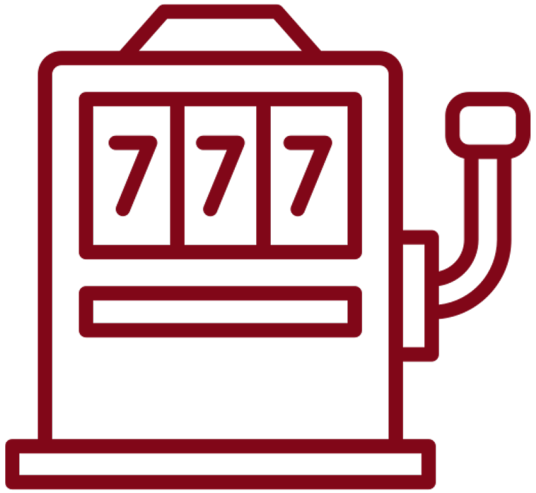
Online Planning

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

Let’s Play!



$1: 100%



$2: ?%
$0: ?%

$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

STEVENS
INSTITUTE OF TECHNOLOGY
1870

Questions?

Acknowledgement

Fahiem Bacchus, University of Toronto
Dan Klein, UC Berkeley
Kate Larson, University of Waterloo