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

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

Modeling the relation between variables **probabilistically**

Learning the relation between variables from data

Markov Decision Processes and Reinforcement Learning

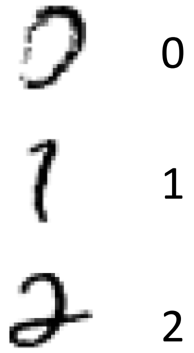
- Search
  - Single-agent search
  - Multi-agent search
  - Constraint satisfaction
  - Logic
- Probabilistic Modeling
  - Bayesian network
  - (Hidden) Markov models
- Machine Learning
  - Learning the model from data

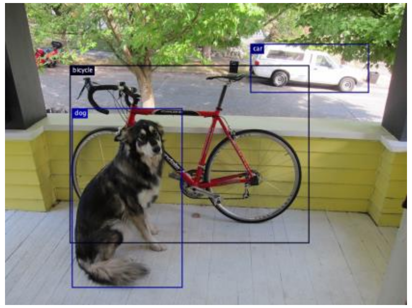
Probabilistic model for search problems
(Markov decision processes)

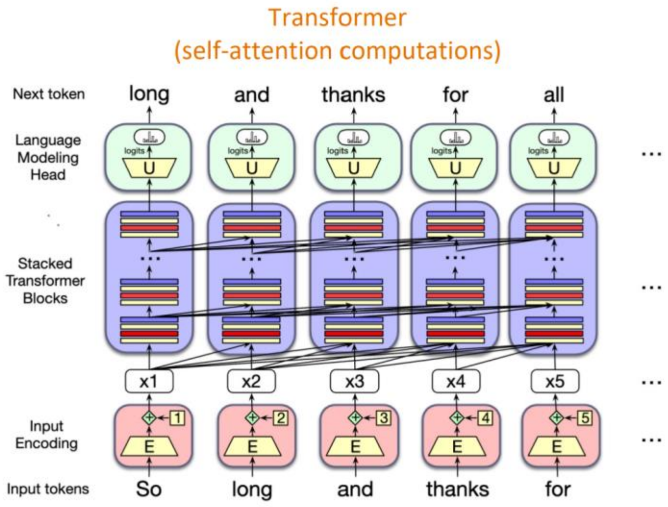
Searching while learning the model
(Reinforcement Learning)

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

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

Y: x_i

supervised learning

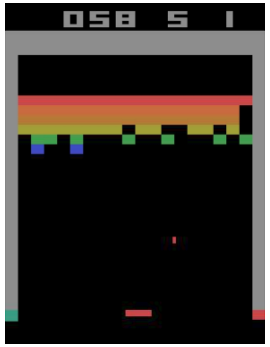
self-supervised learning

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



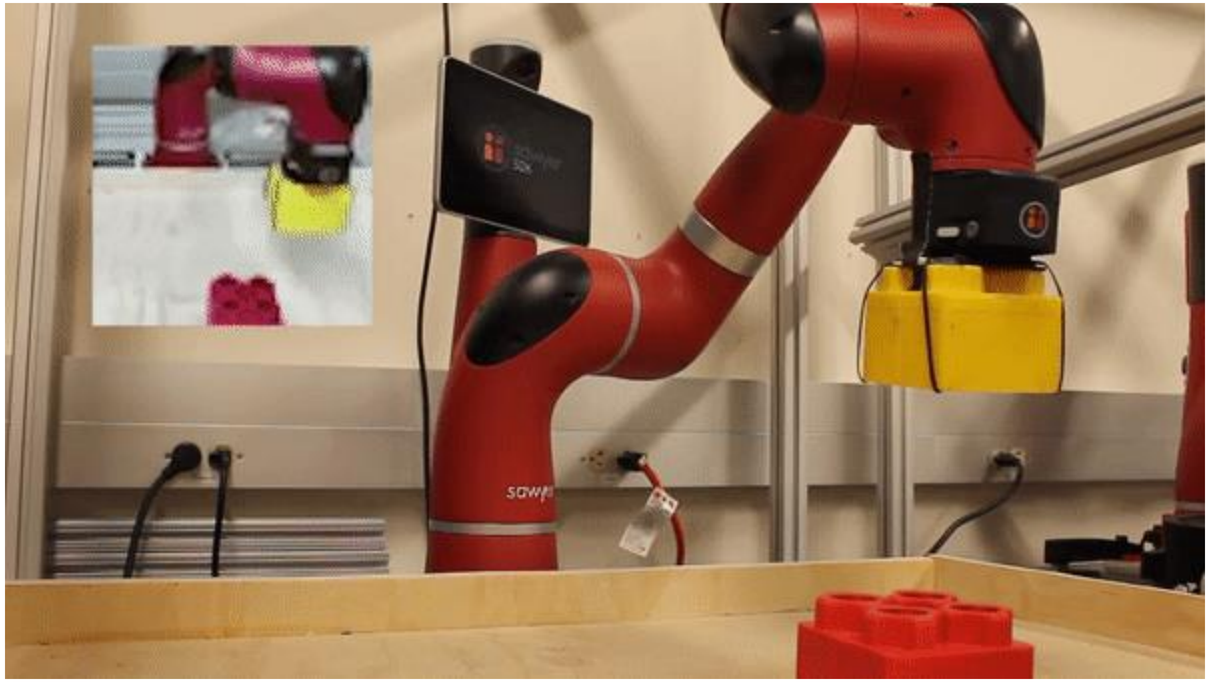
X: View of the game   Y: Action (left or right)

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

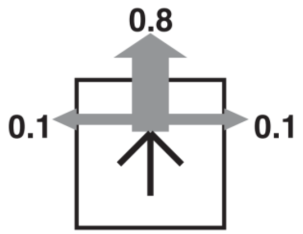
(Just a probabilistic model for search problems --- no “learning”)

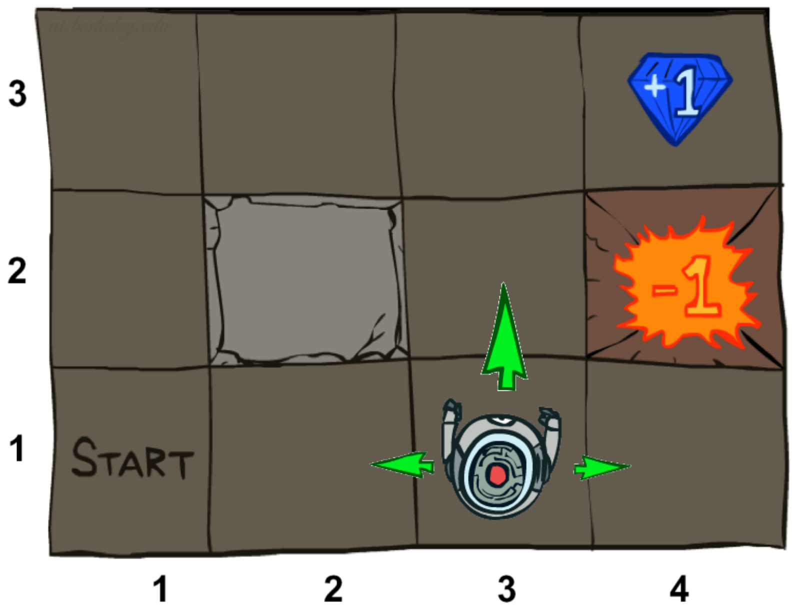
Example: Grid World

- 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)

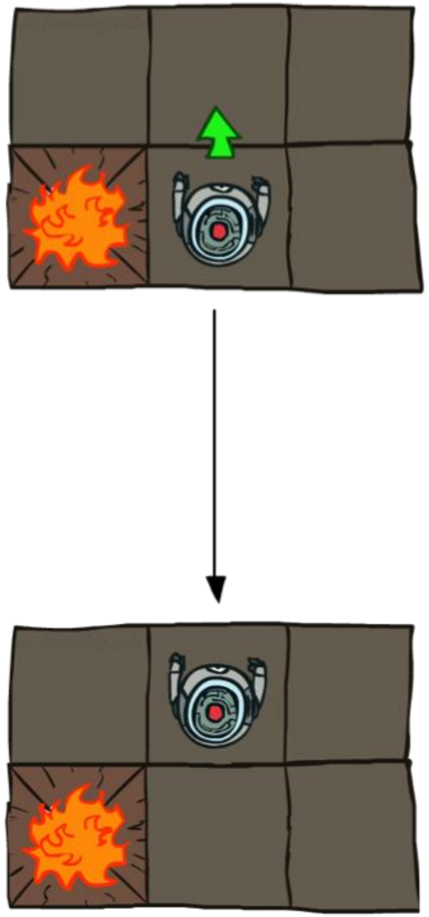
- Goal: maximize sum of rewards



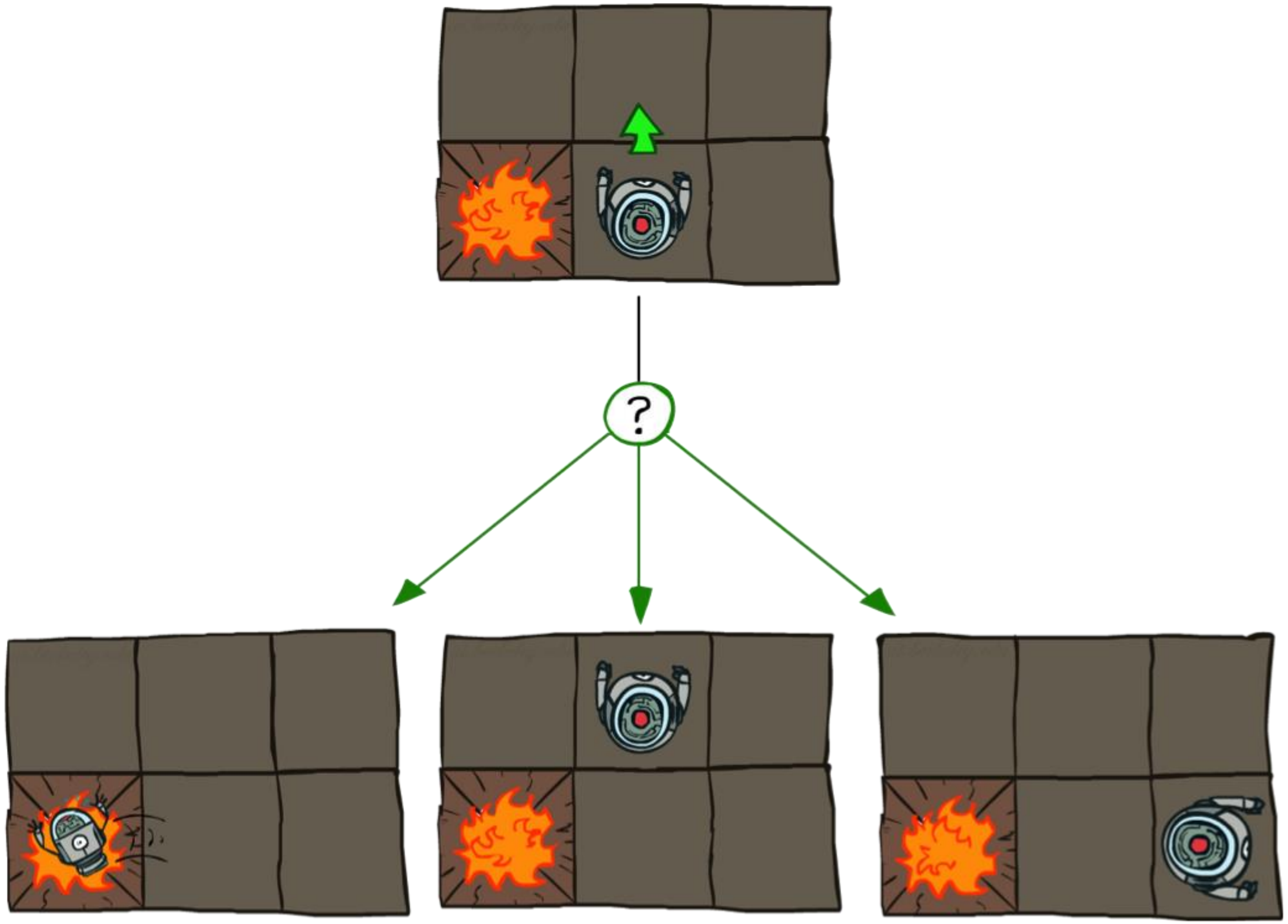


Grid World Actions

Deterministic Grid World

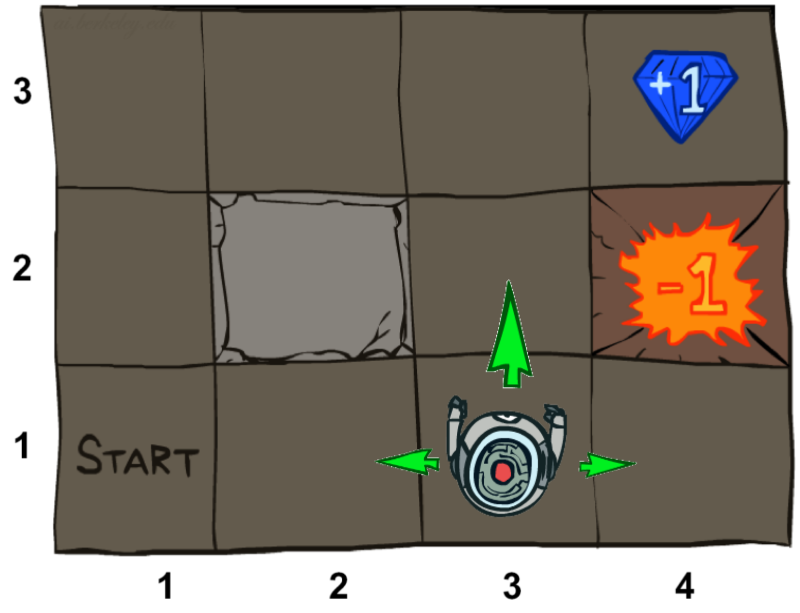


Stochastic Grid World



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

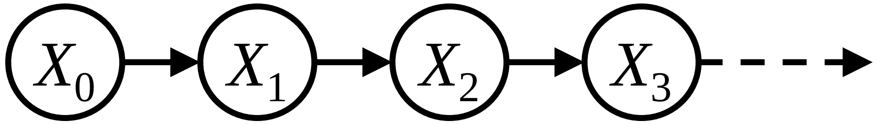
$\bigcirc \to \bigcirc \to \bigcirc \to \cdots$

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

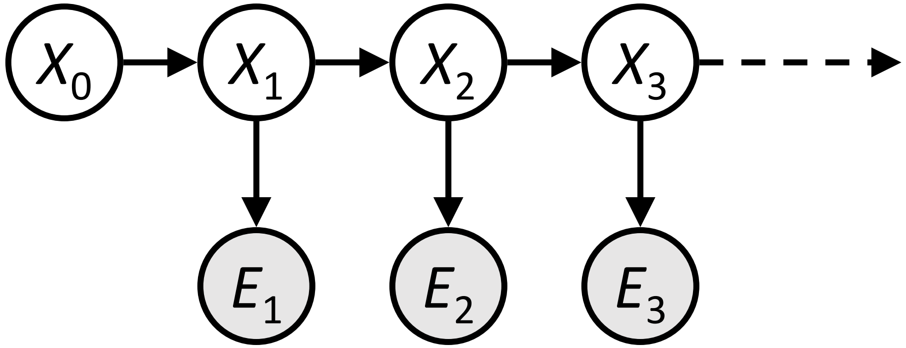
$$
\begin{aligned}
& P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0) \\
= \; & P(S_{t+1} = s' | S_t = s_t, A_t = a_t)
\end{aligned}
$$

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

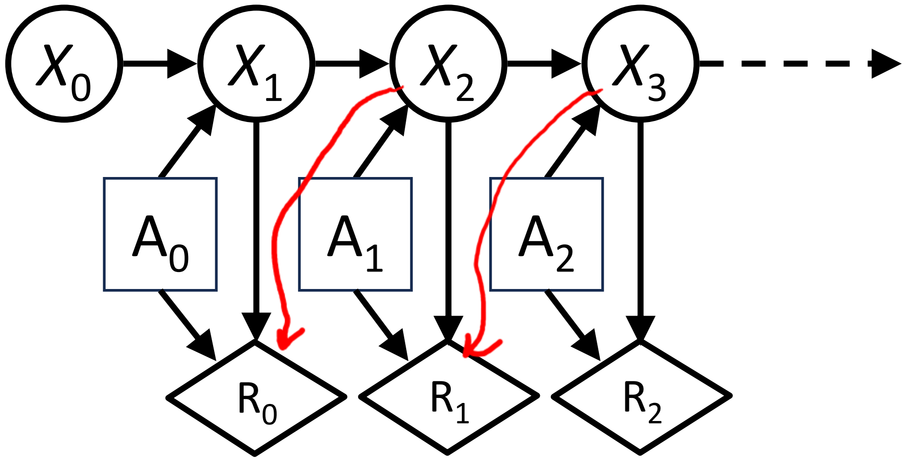
“Markov” as in Markov Chains? HMMs?



Markov Model (Markov Chain)



Hidden Markov Model



Markov Decision Process

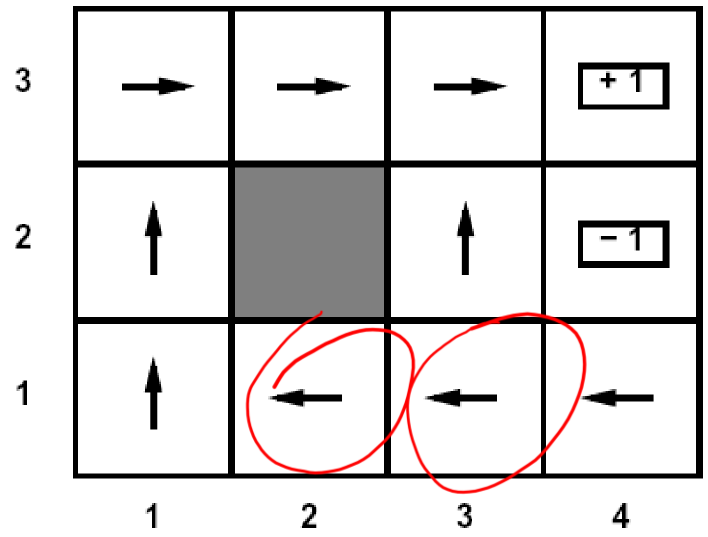
???

Partially Observable Markov Decision Process

Policies

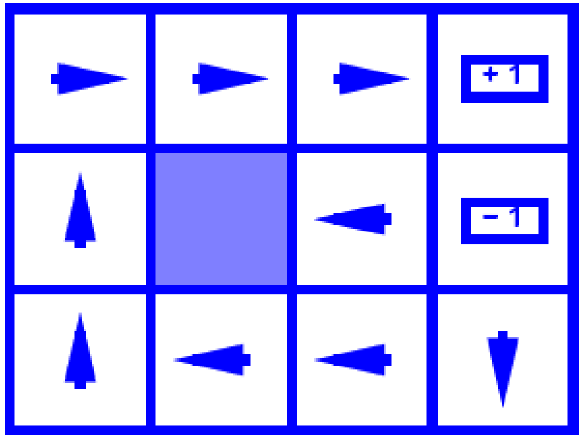
- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal

- 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

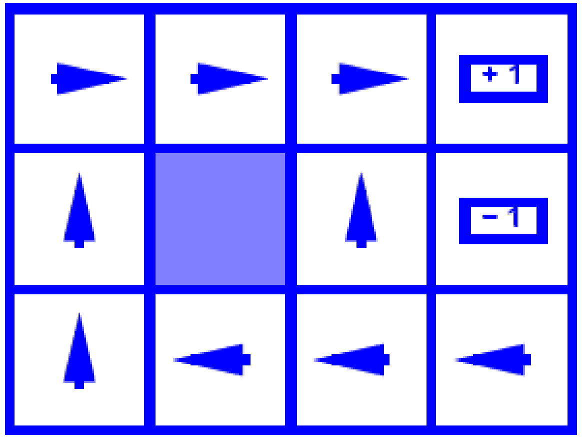


living reward = −0.1

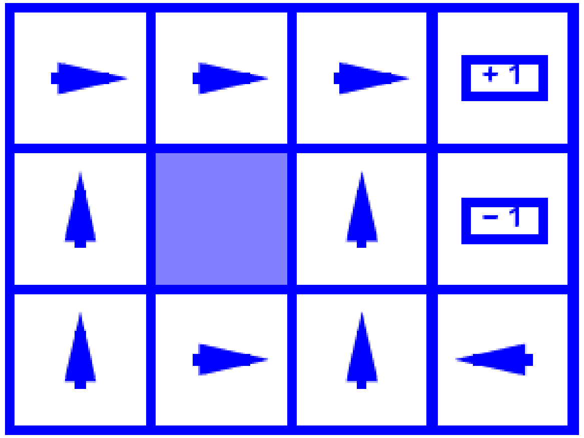
Optimal Policies



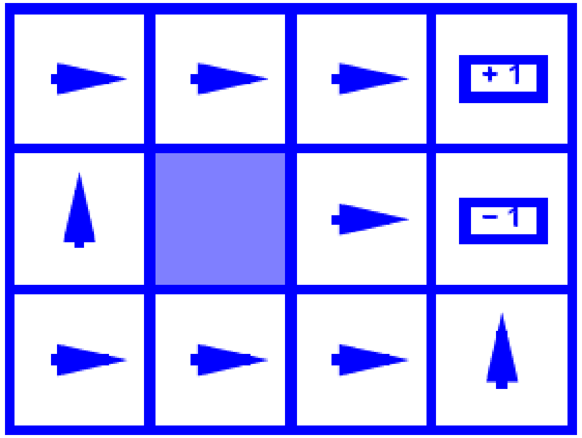
R(s) = -0.01



R(s) = -0.03

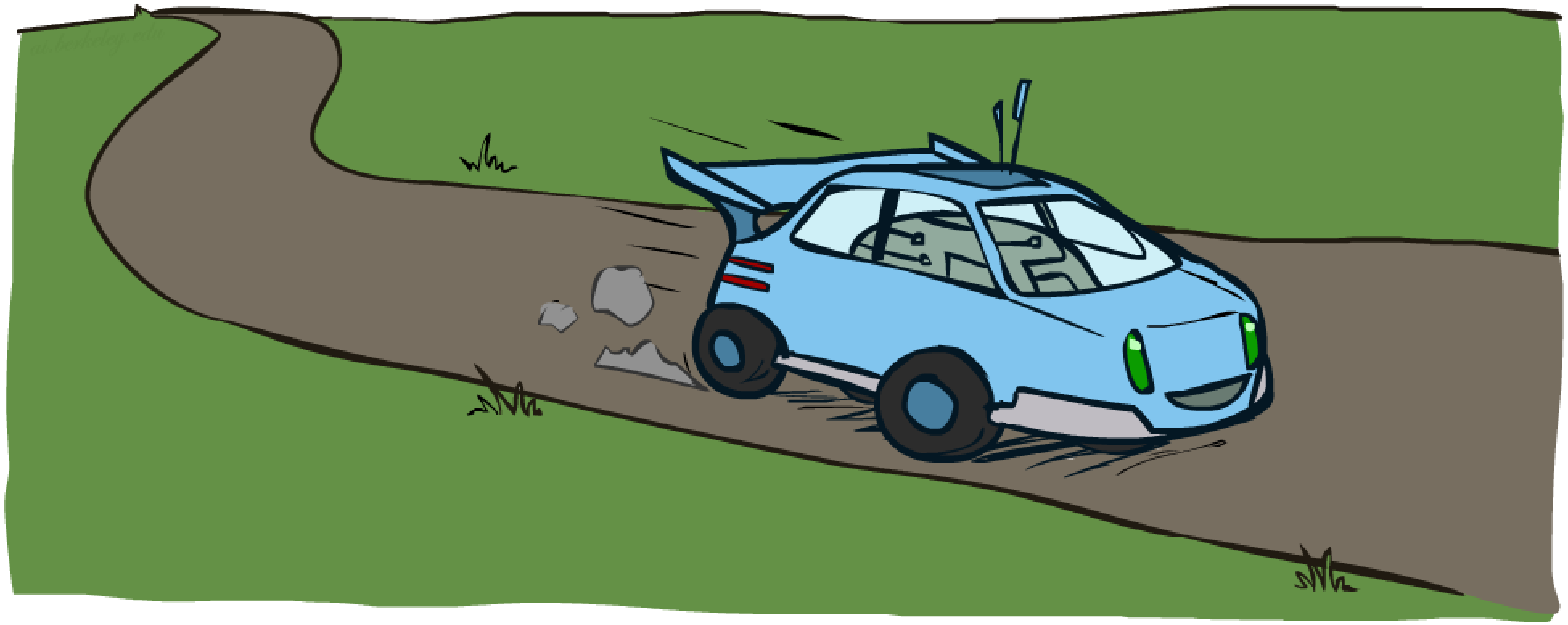


R(s) = -0.4



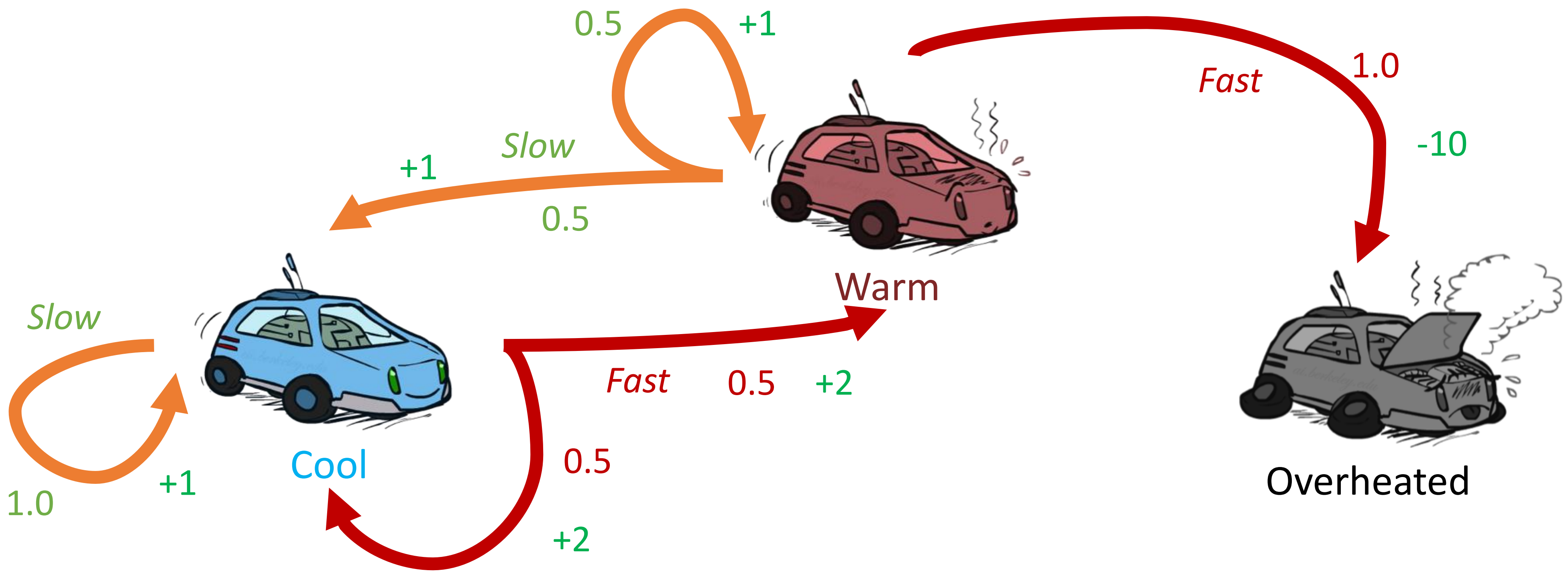
R(s) = -2.0

Example: Racing



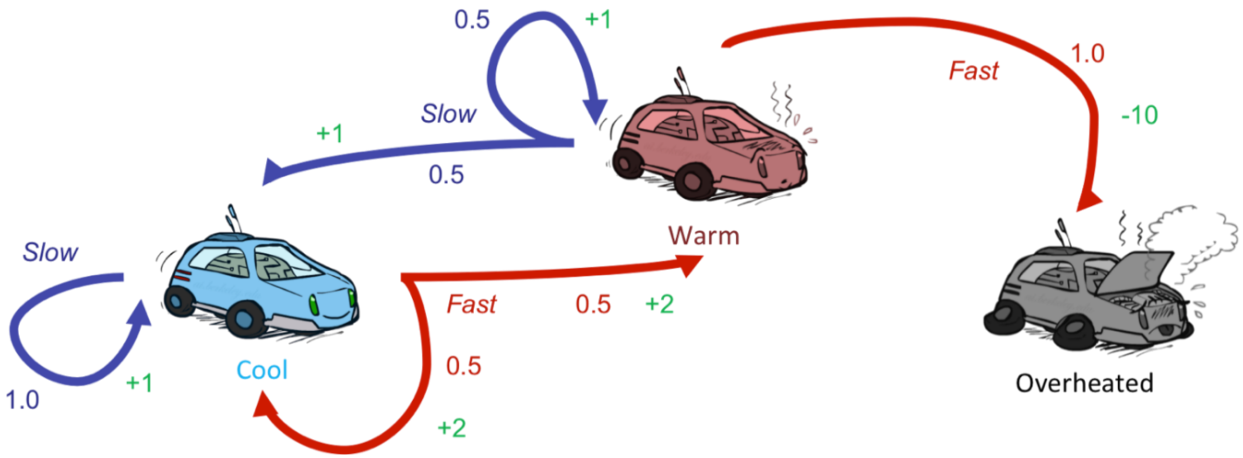
Example: Racing

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



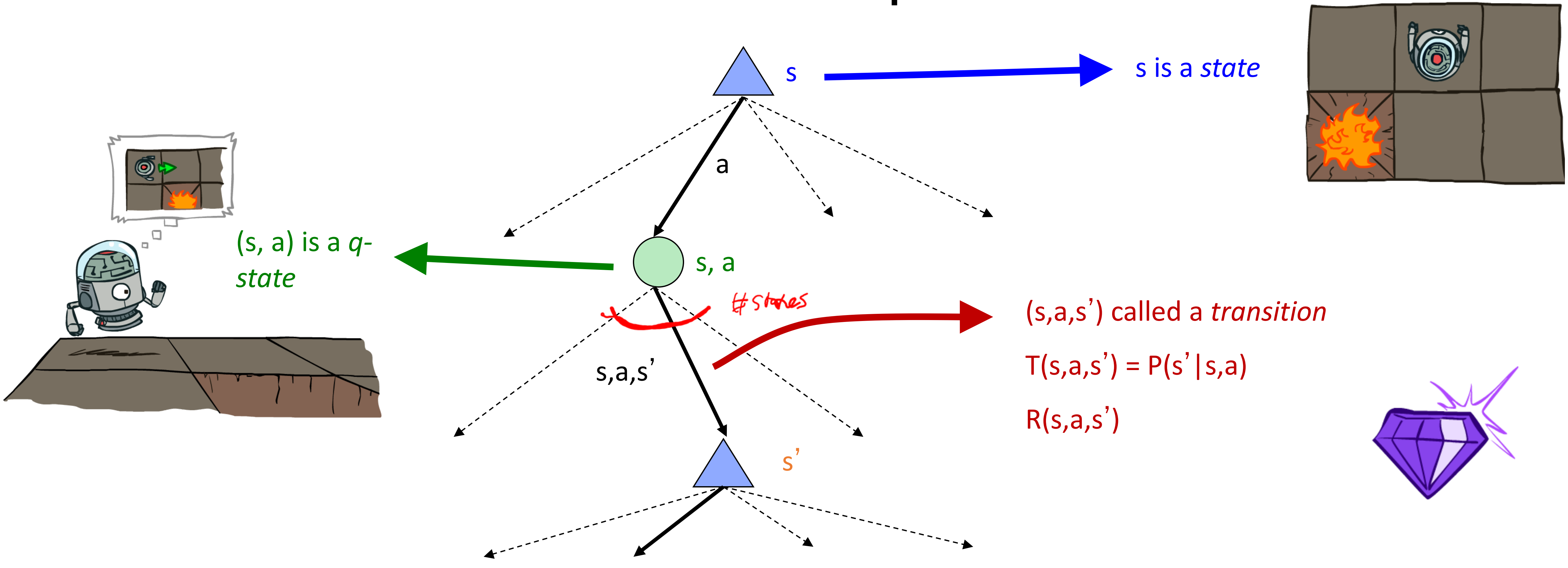
Example: Racing

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

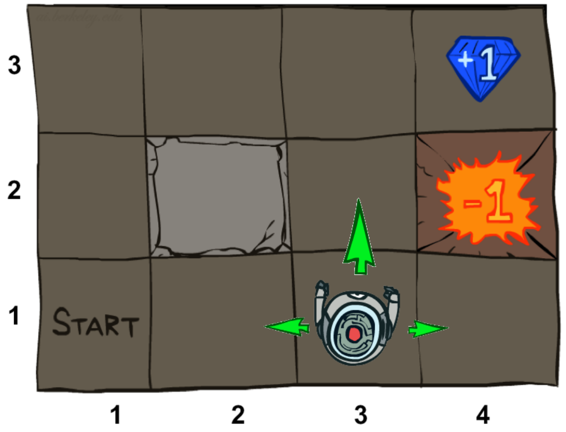
- MDP search tree can be viewed as an **expectimax** search tree



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



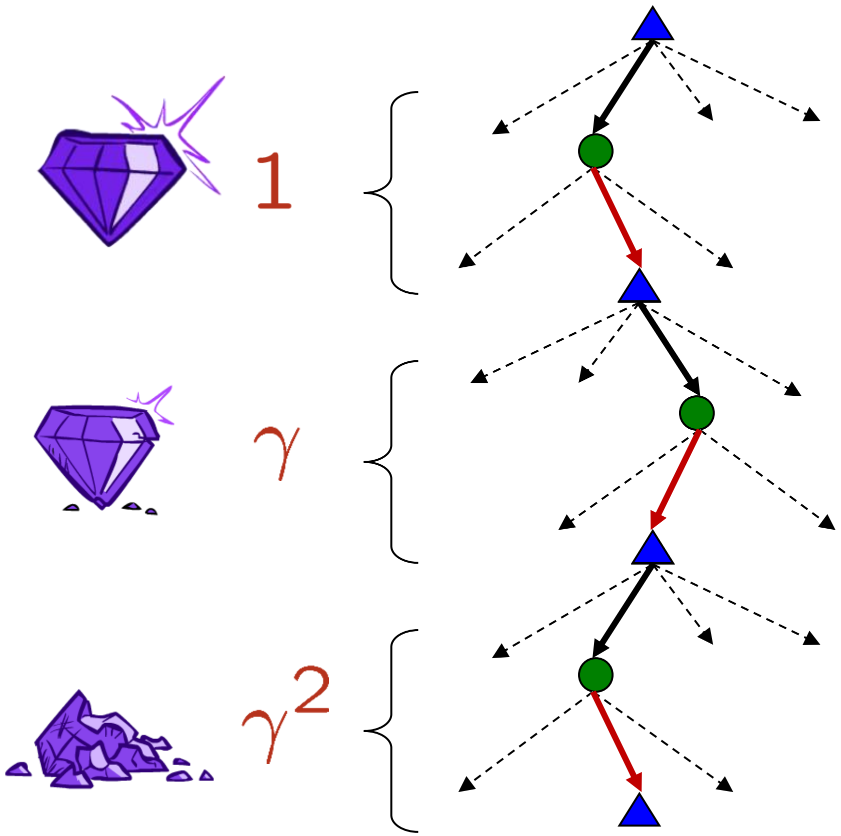
$100 right now

vs.

$110 next year

Discounting $\boxed{0 < \gamma < 1}$

- 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])

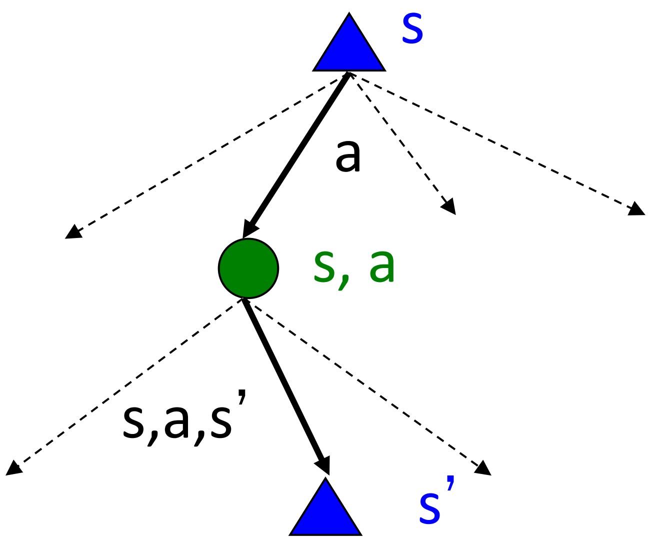


Value Functions and Optimal Policies

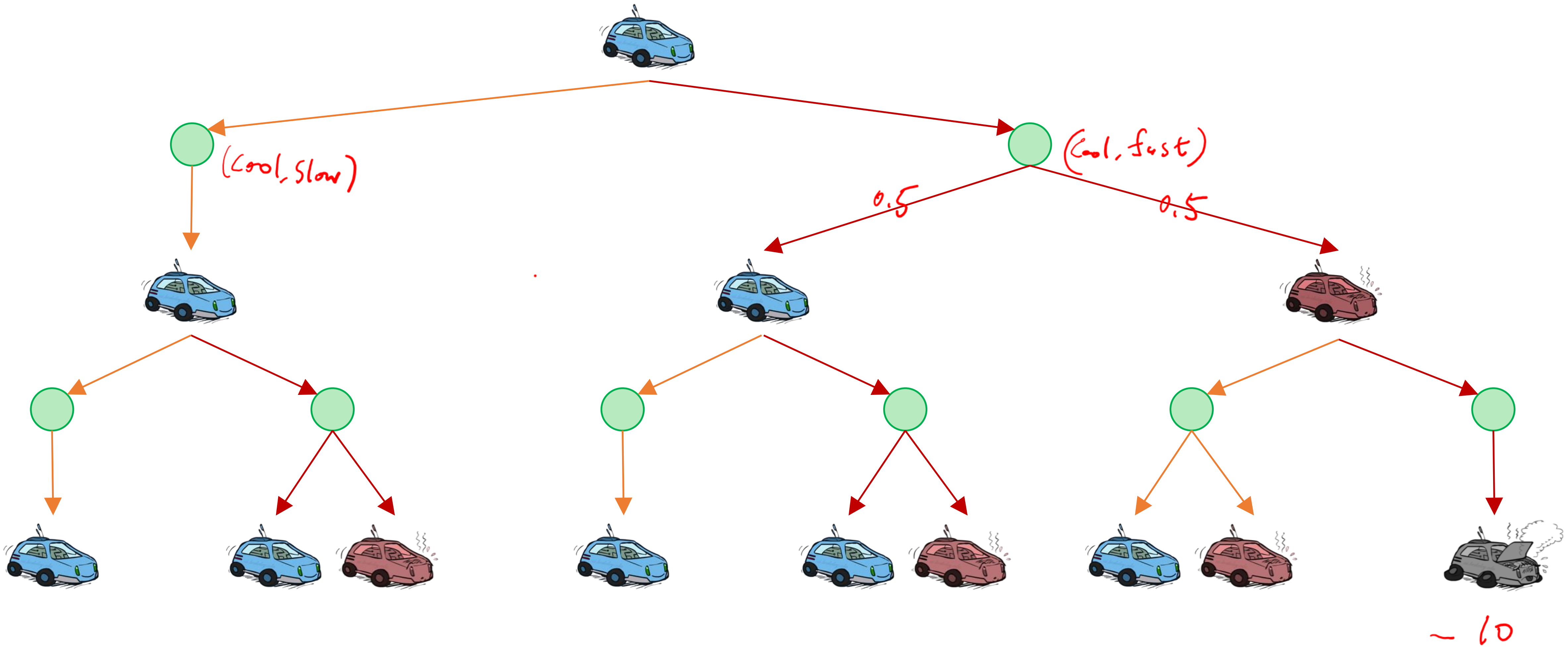
Recap: Defining MDPs

- 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)

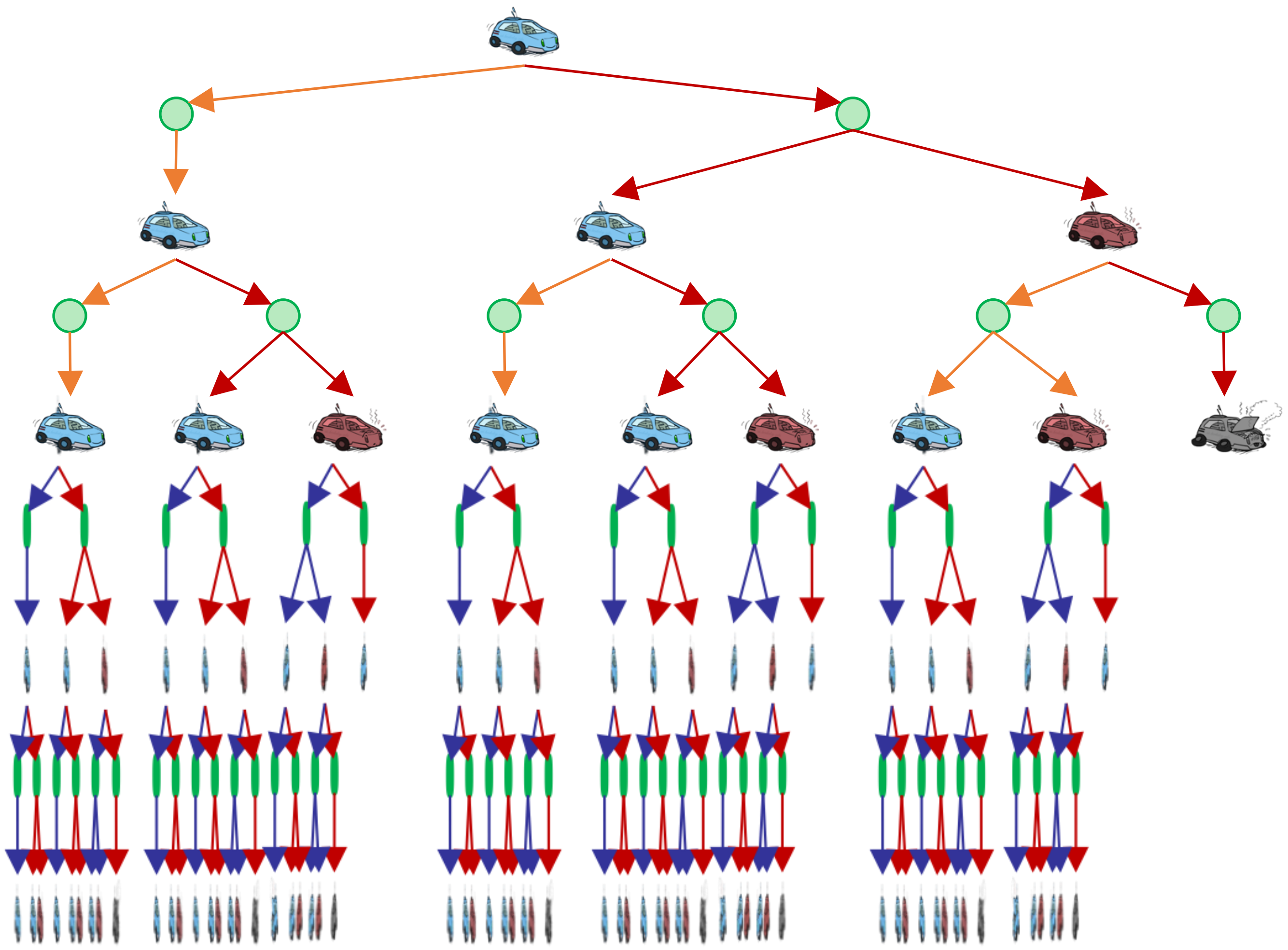
- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility or Return = sum of (discounted) rewards



Racing Search Tree

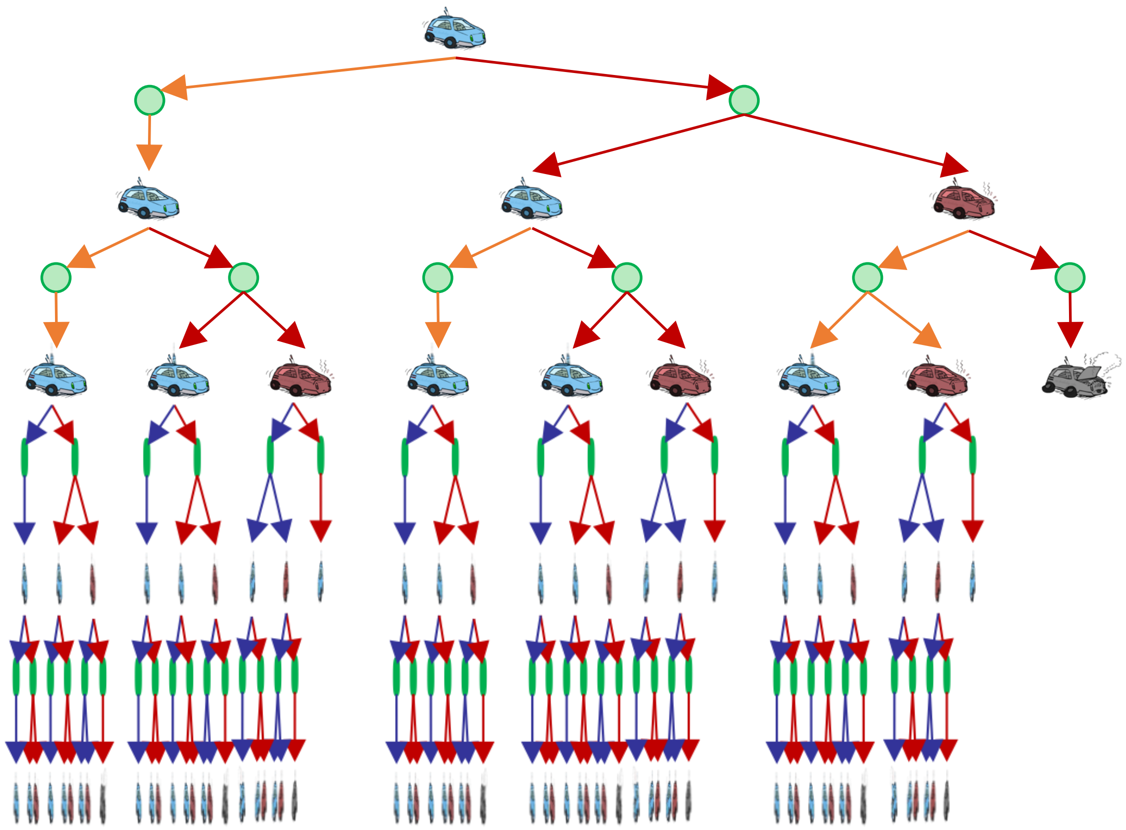


Racing Search Tree

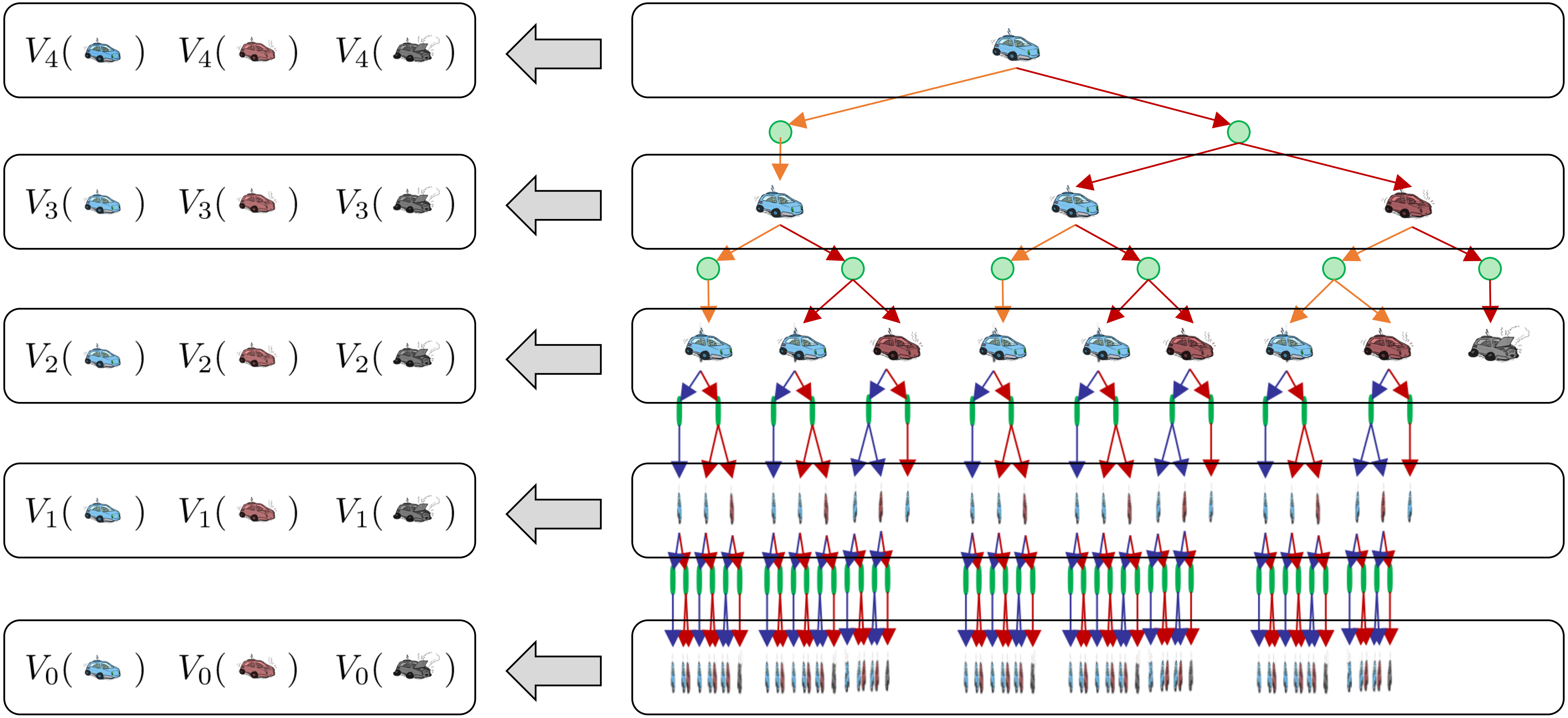


Racing Search Tree

- 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 $\gamma < 1$

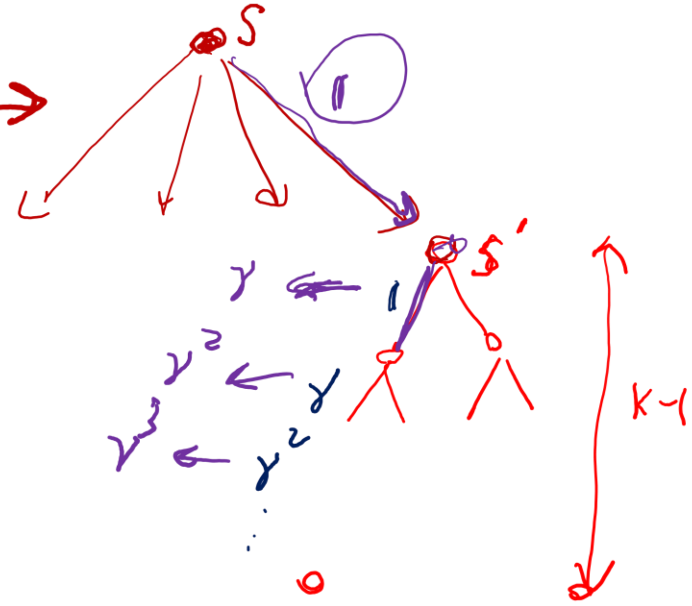


Computing Time-Limited Values



Time-Limited Values

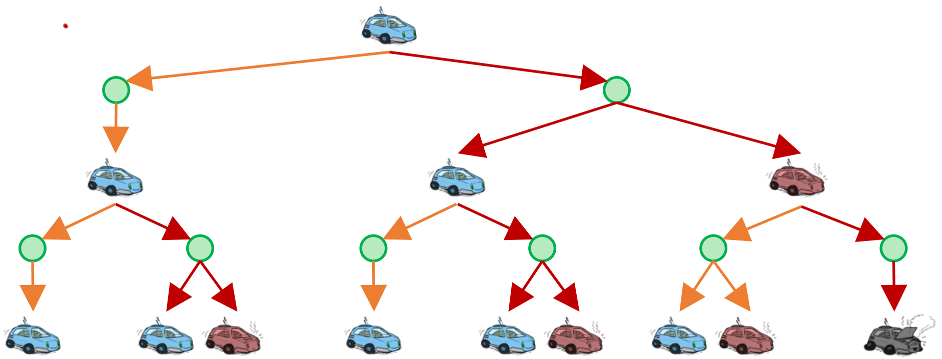
- Define $V_k(s)$ to be the optimal value of s if the game ends in k more time steps



$$V_0(s) = 0$$

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

recursively for $k \geq 1$



Example

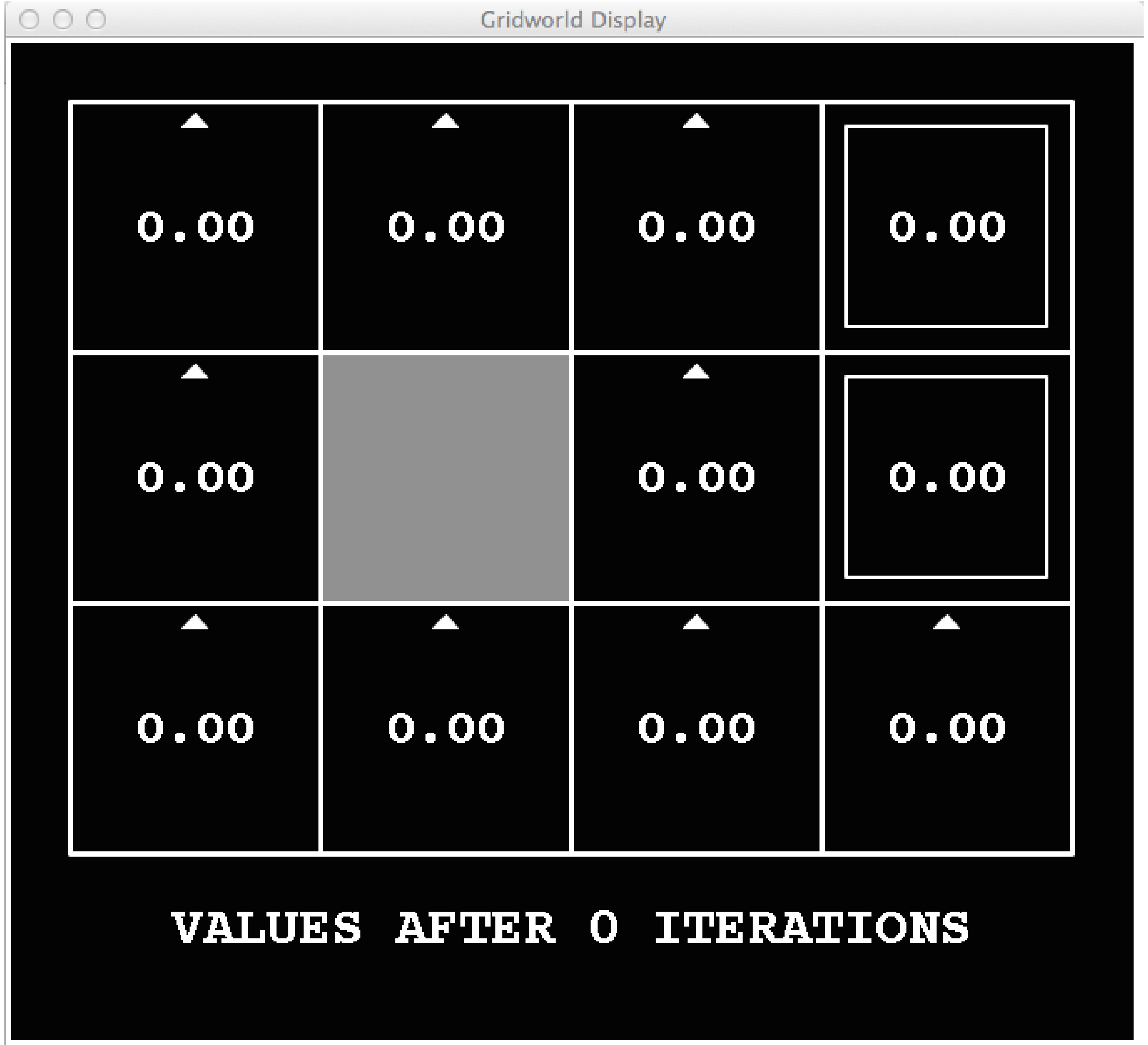
s	a	s'	T(s,a,s’)	R(s,a,s’)
	Slow		1.0	+1
	Fast		0.5	+2
	Fast		0.5	+2
	Slow		0.5	+1
	Slow		0.5	+1
	Fast		1.0	–10
	(end)		1.0	0

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

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

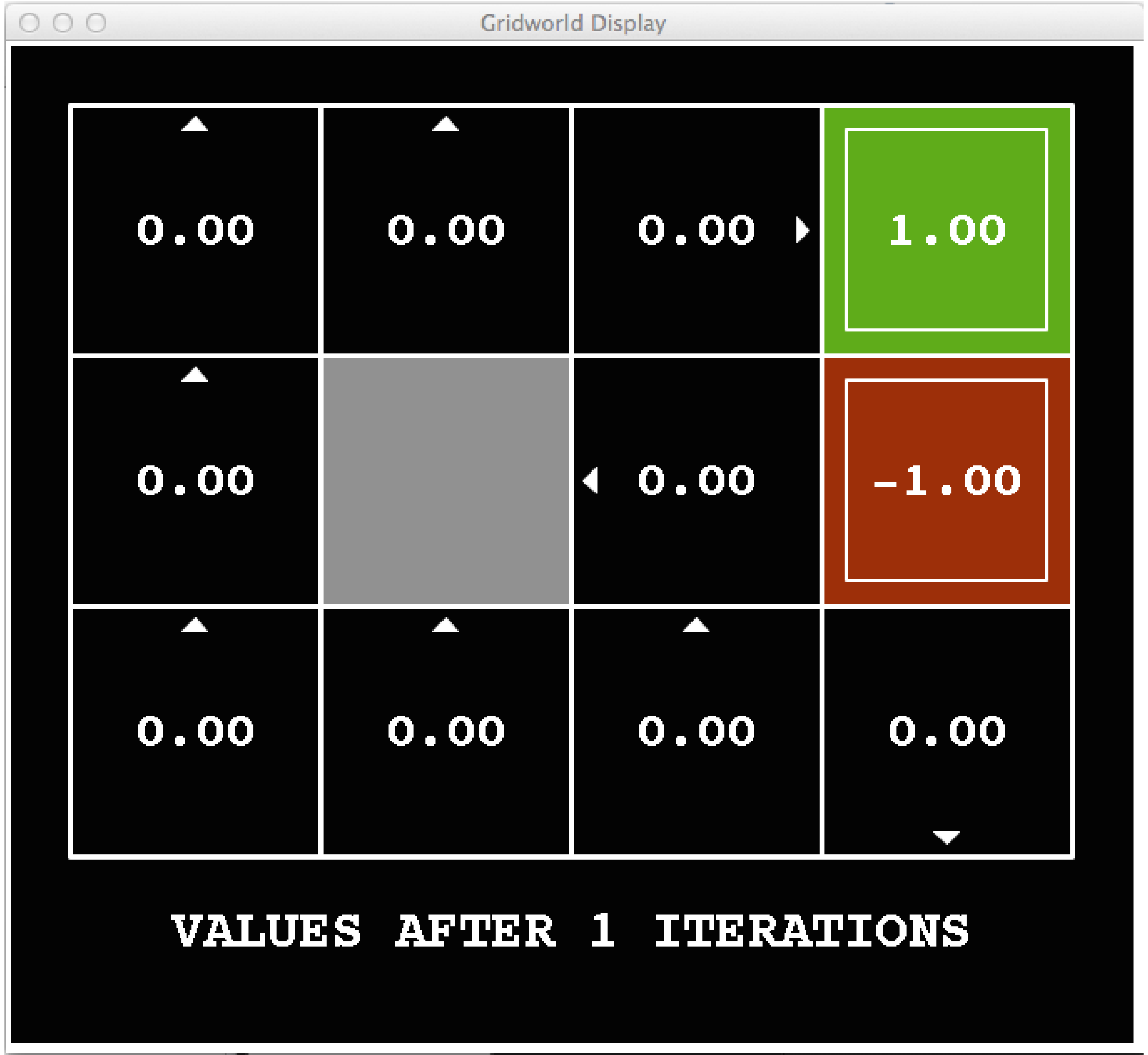
Assume no discount $(\gamma = 1)$

k=0



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1



Noise = 0.2
Discount = 0.9
Living reward = 0

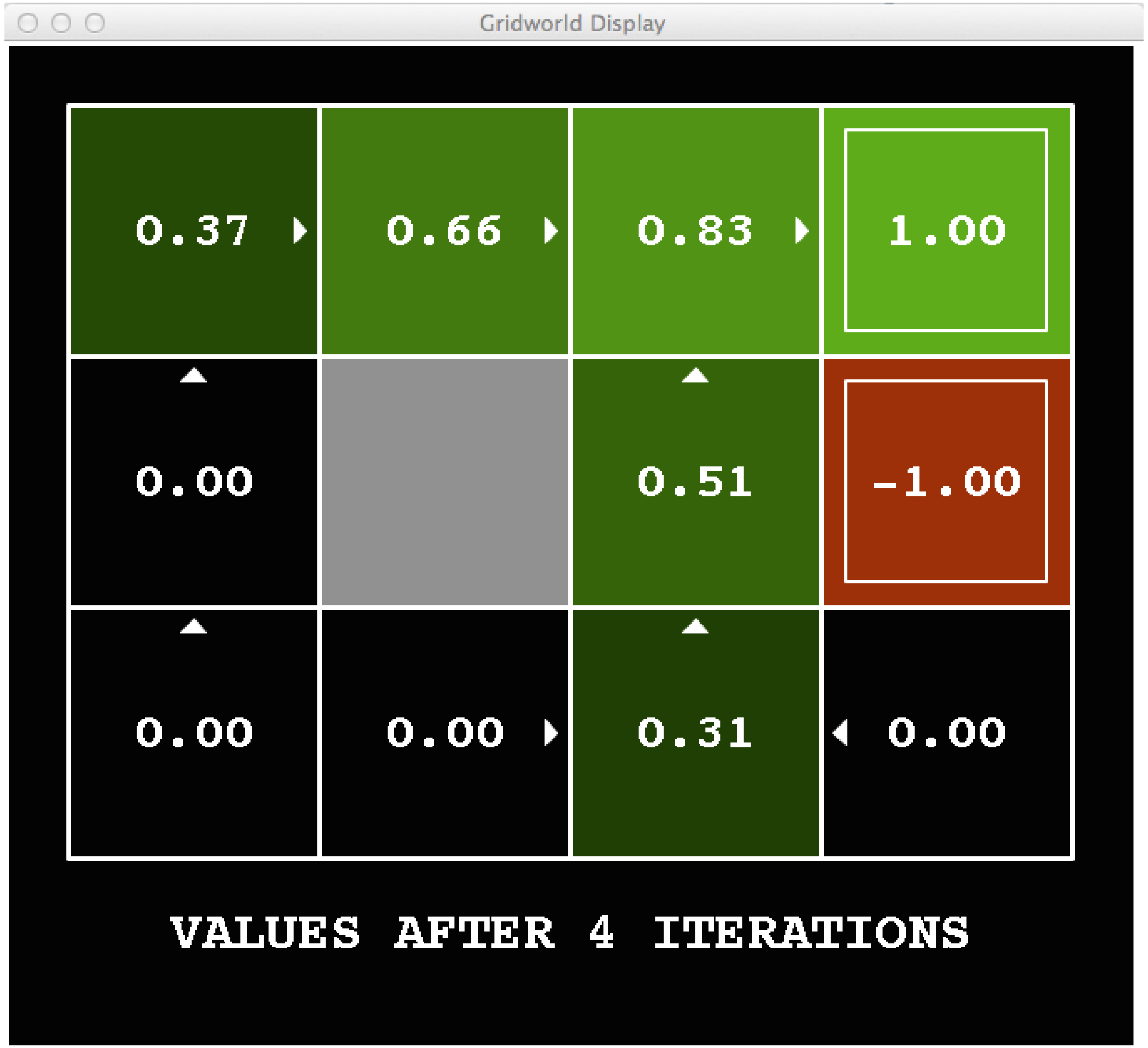
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

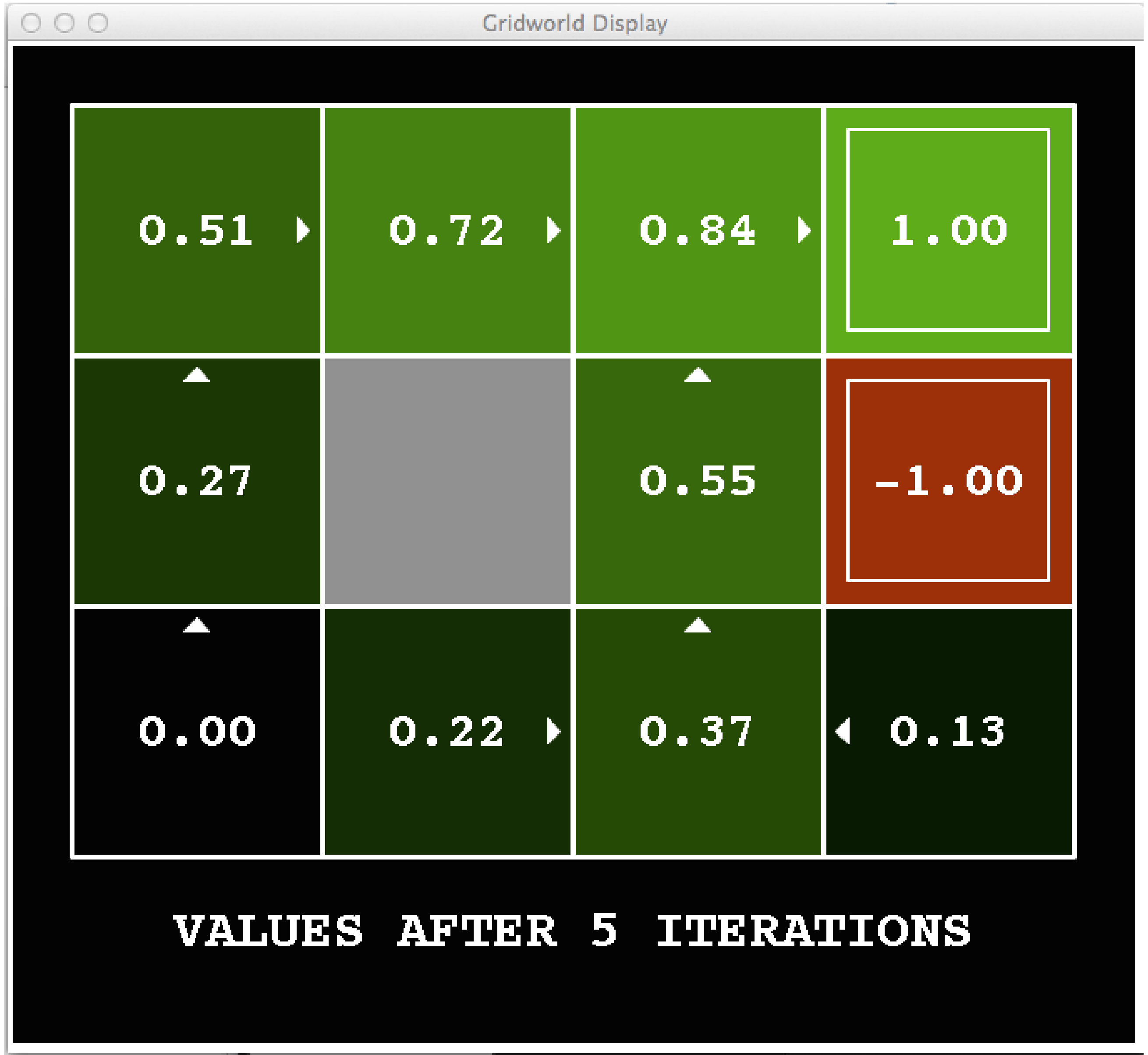
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



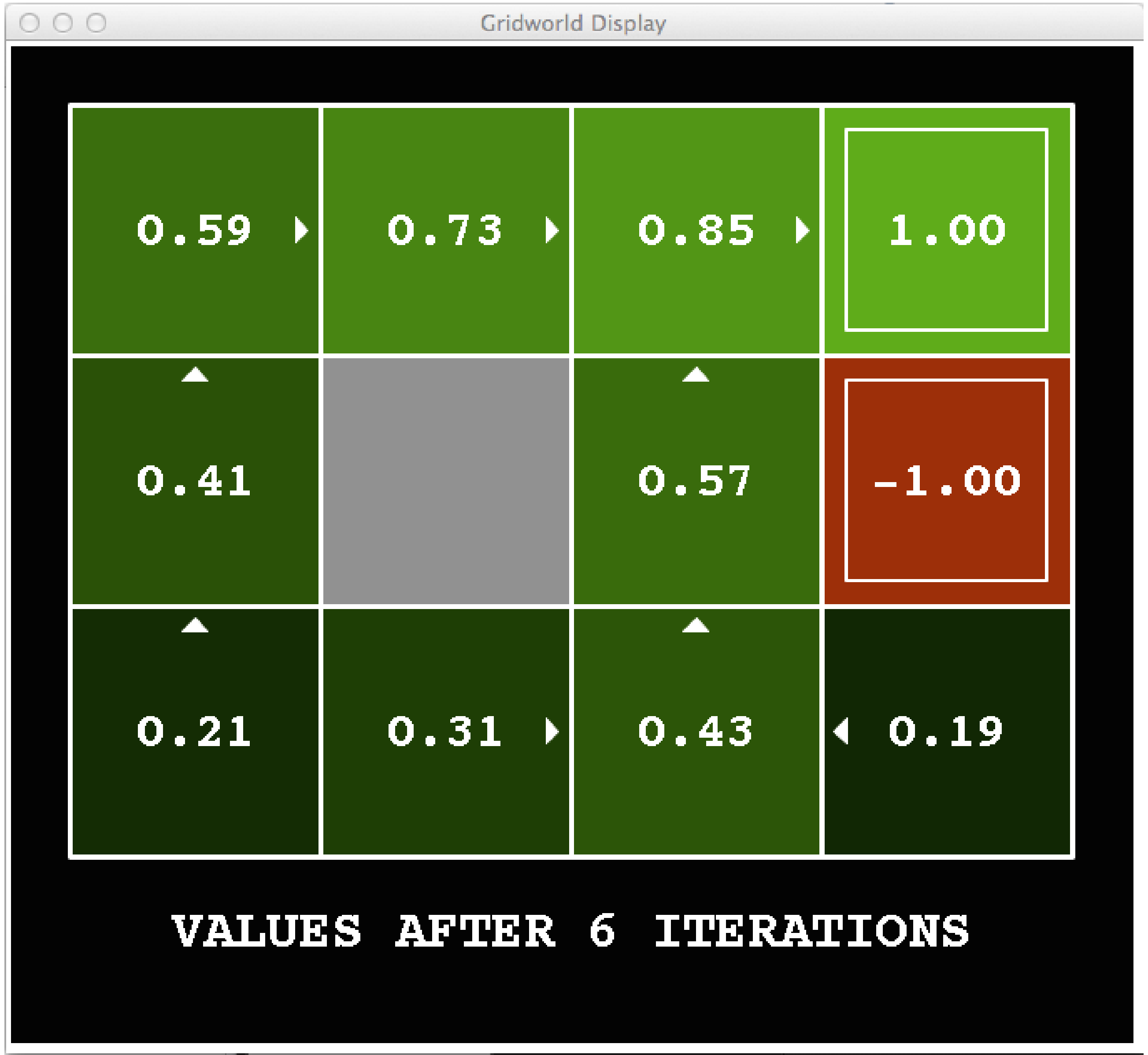
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



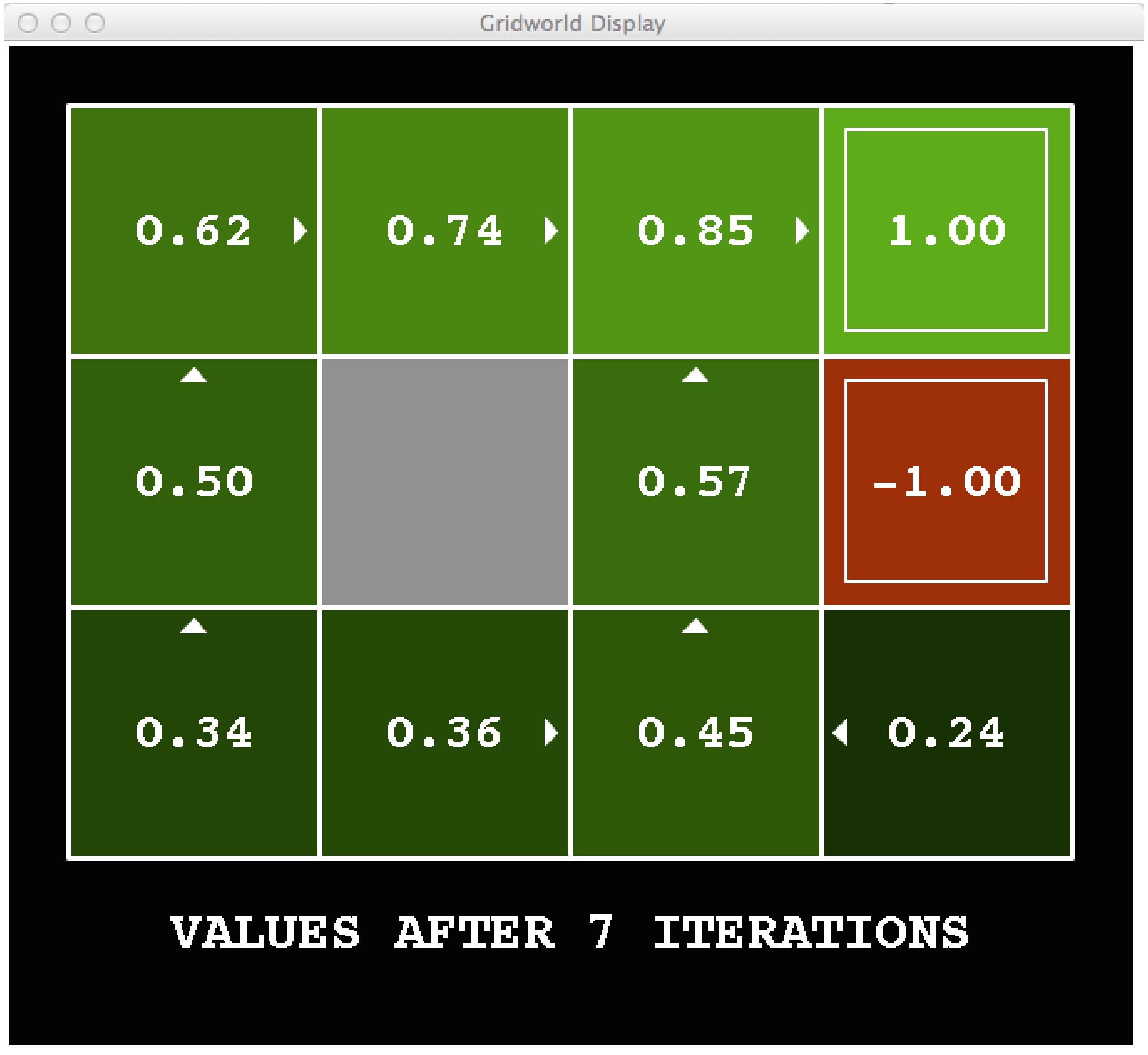
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



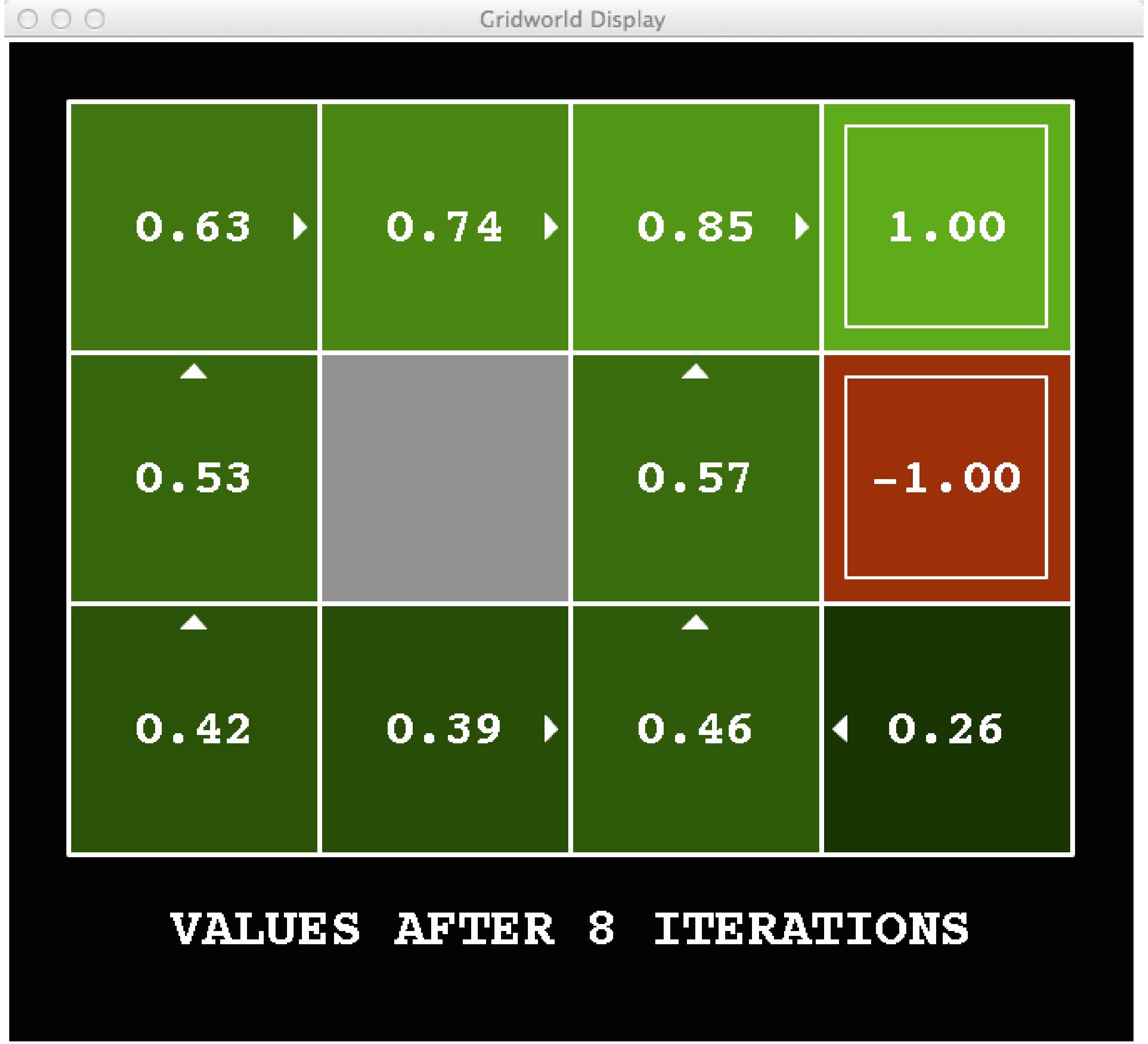
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7



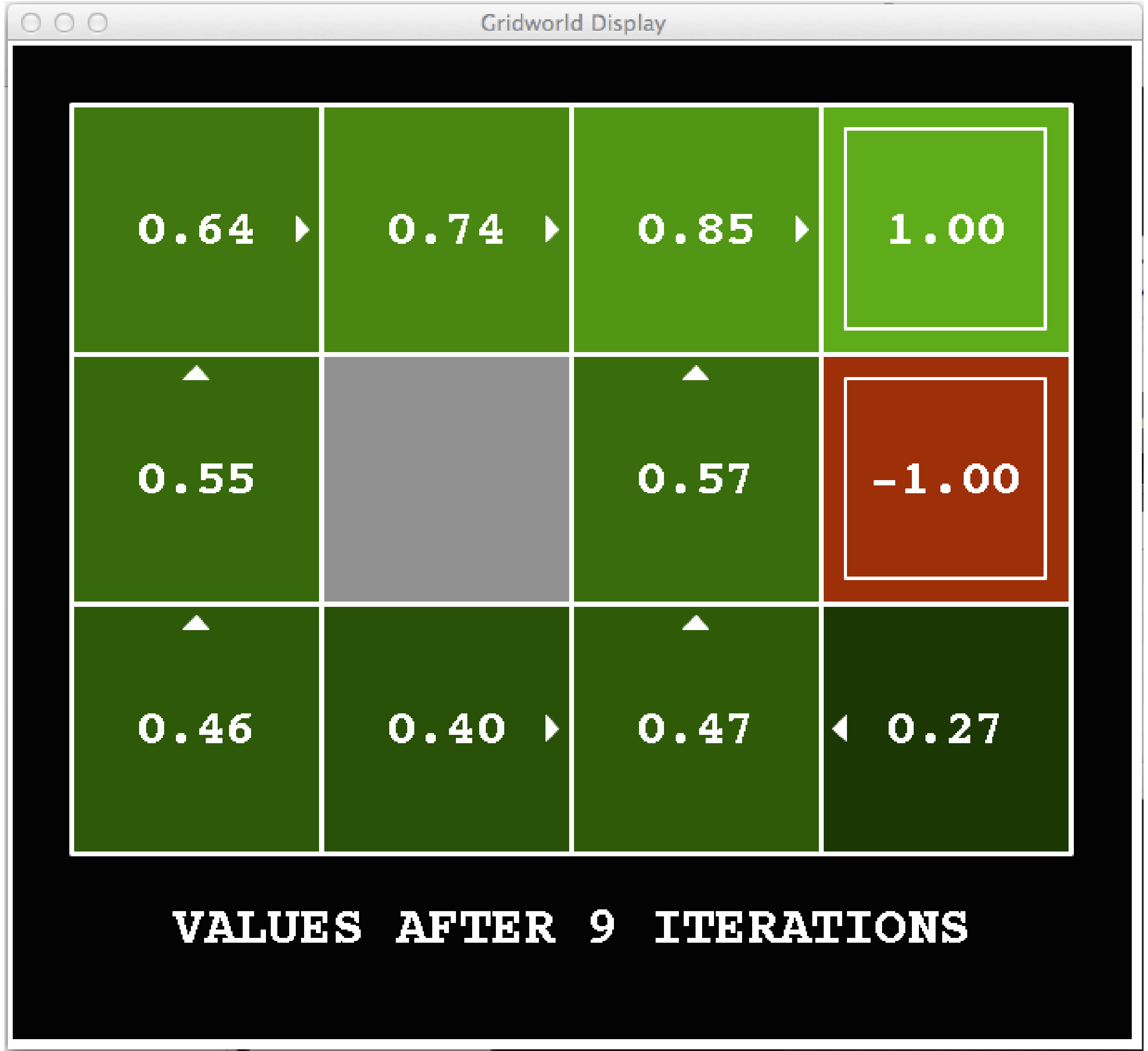
Noise = 0.2
Discount = 0.9
Living reward = 0

k=8



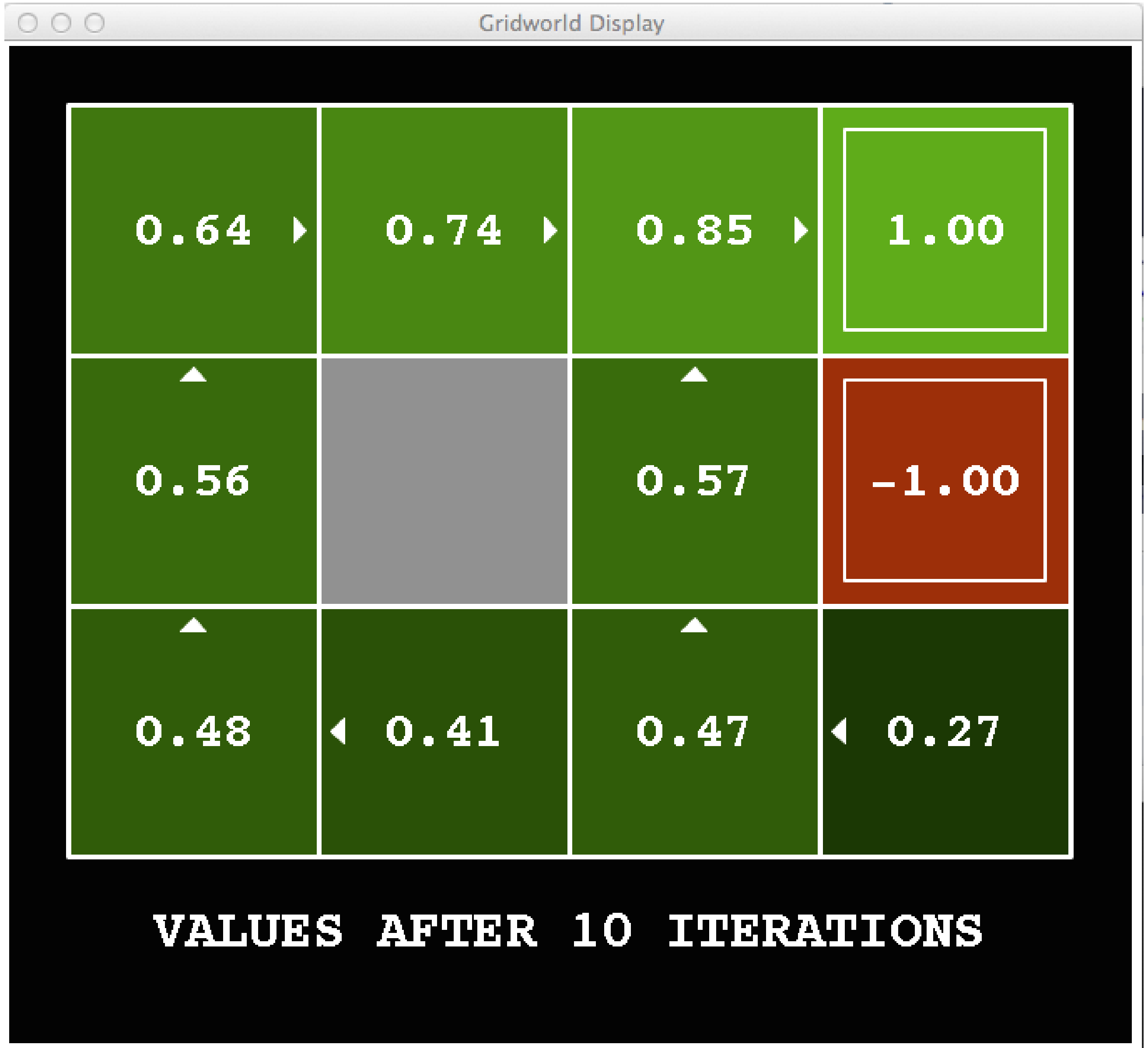
Noise = 0.2
Discount = 0.9
Living reward = 0

k=9



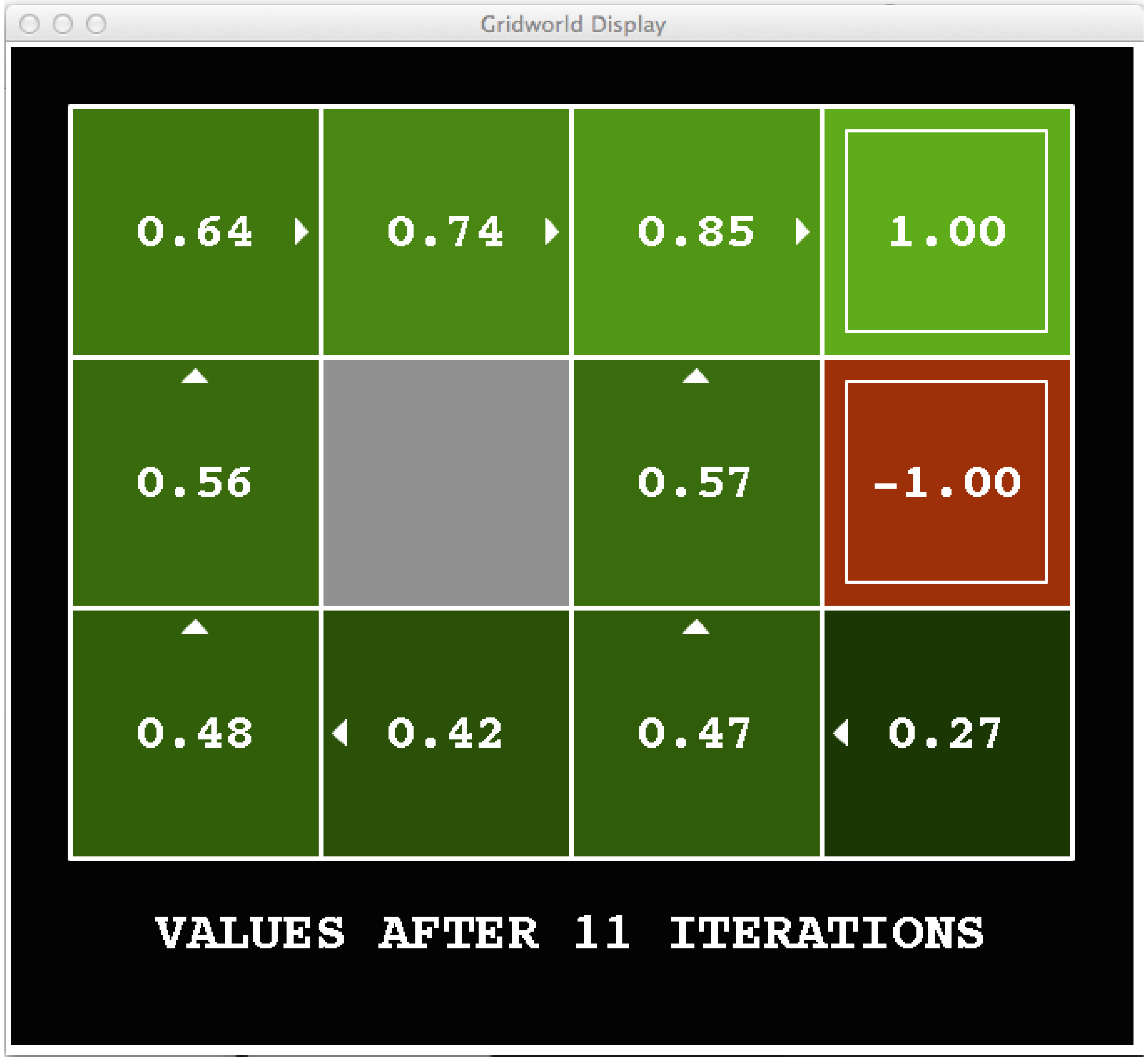
Noise = 0.2
Discount = 0.9
Living reward = 0

k=10



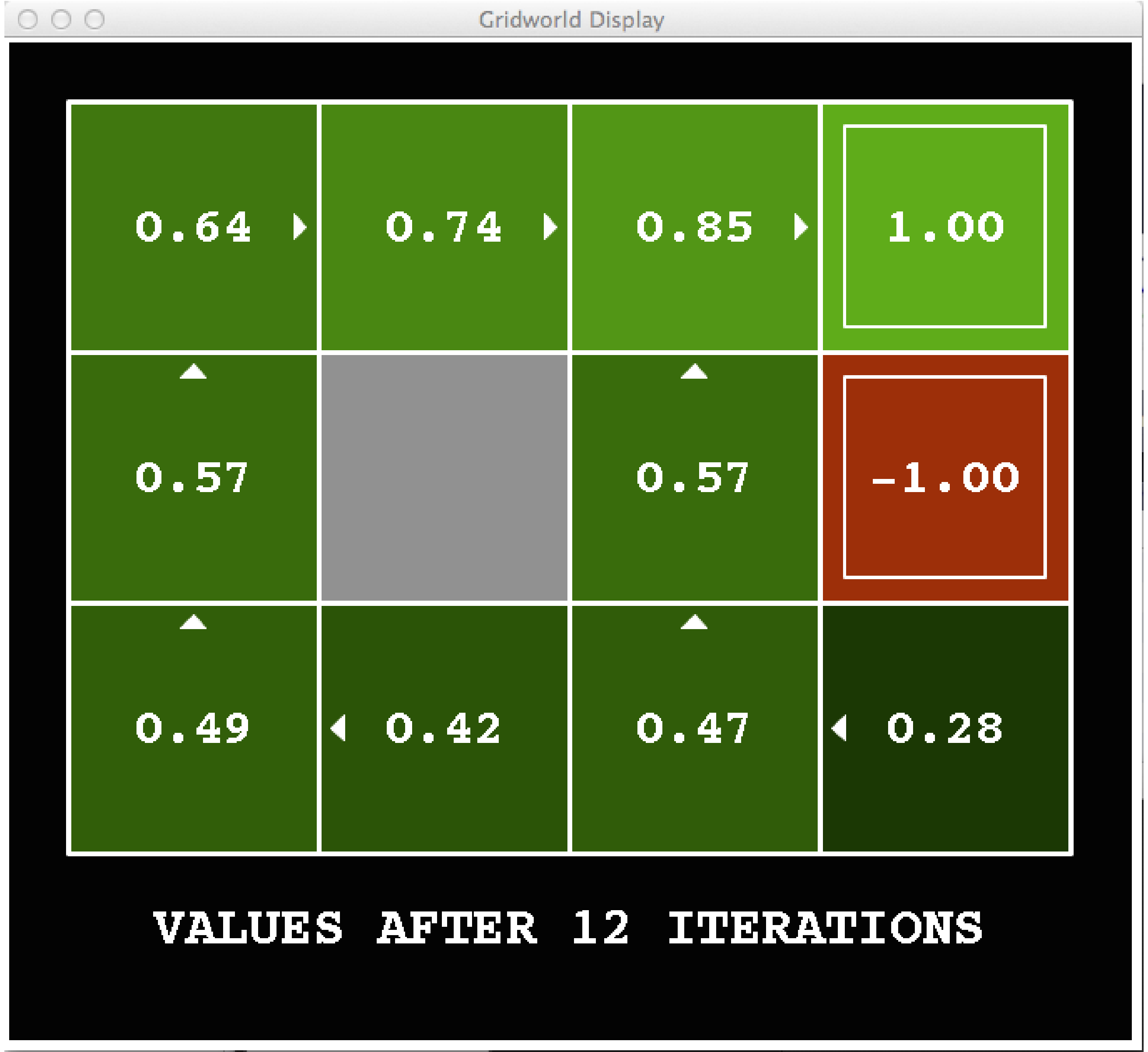
Noise = 0.2
Discount = 0.9
Living reward = 0

k=11



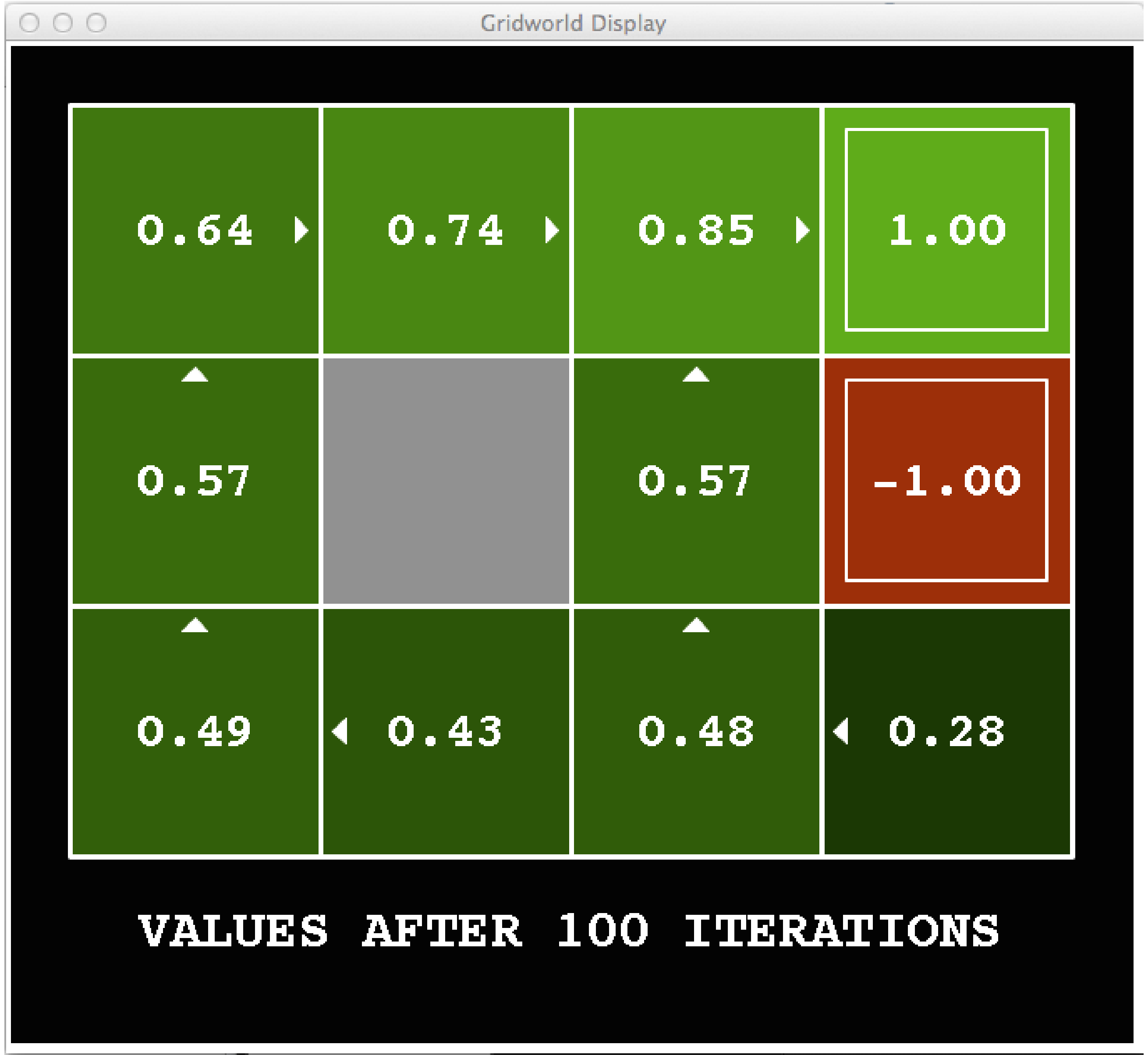
Noise = 0.2
Discount = 0.9
Living reward = 0

k=12



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100



Noise = 0.2
Discount = 0.9
Living reward = 0

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

$$V_0(s) = 0$$

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

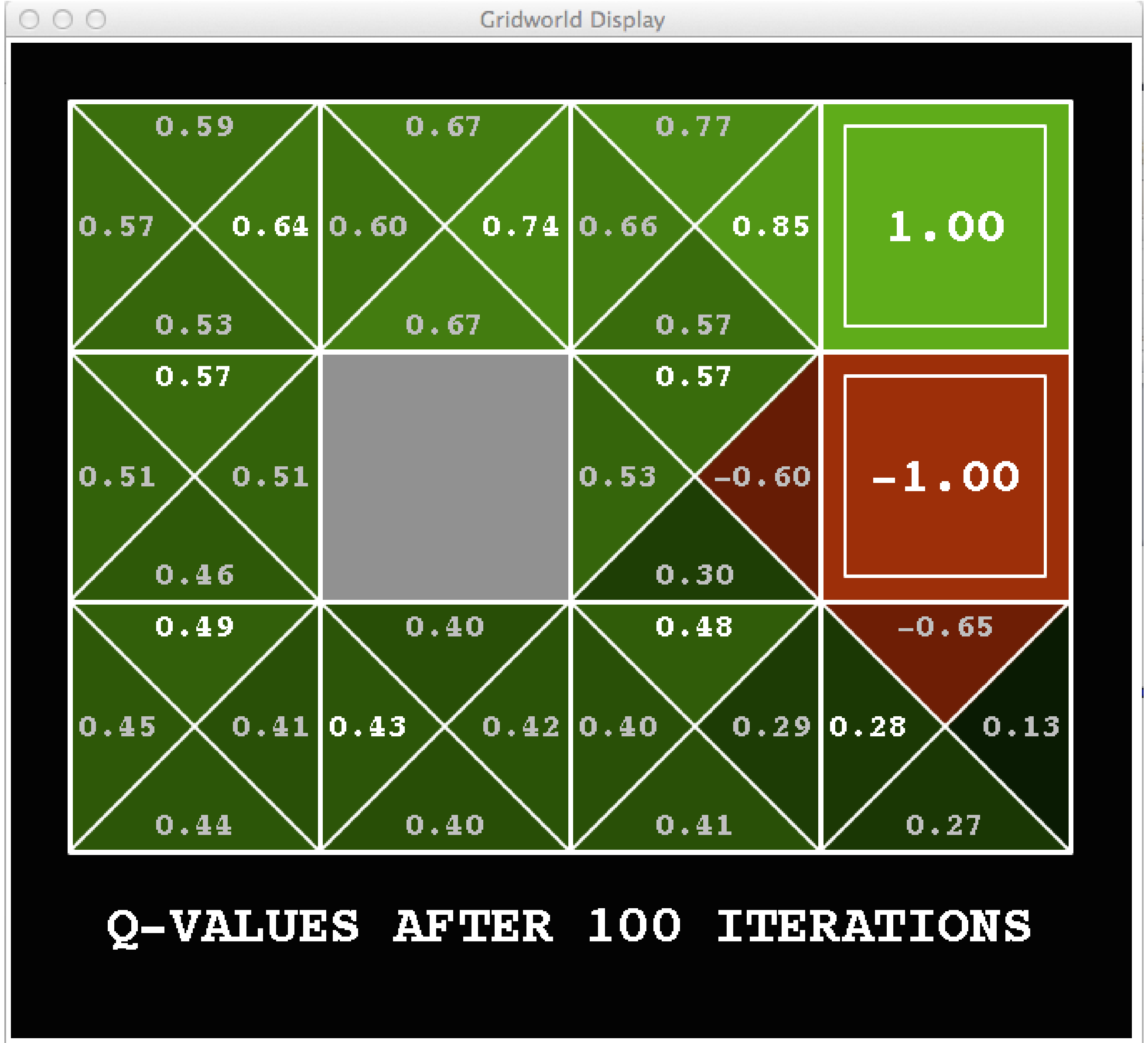
$$\Downarrow$$

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

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

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

Q Values

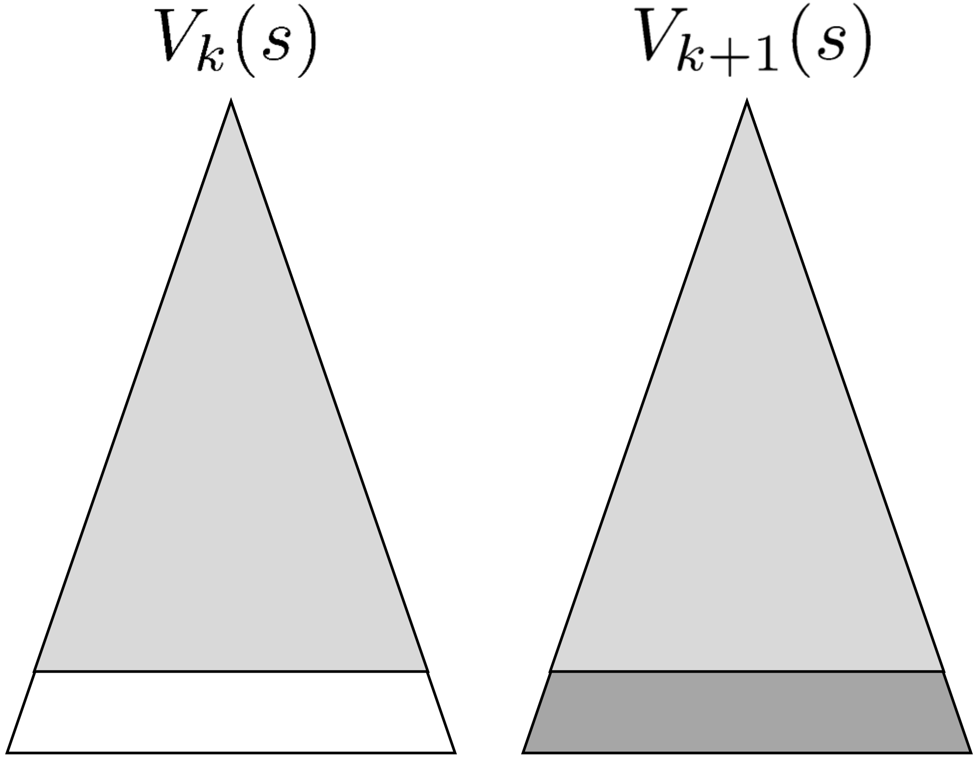


Noise = 0.2
Discount = 0.9
Living reward = 0

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

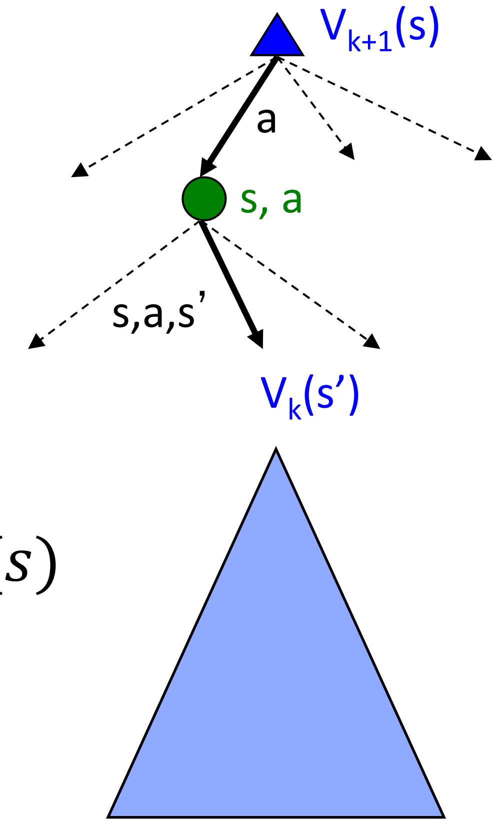


Value Iteration

- Start with $V_0(s) = 0$
- Given vector of $V_k(s)$ values, perform the following from each state:

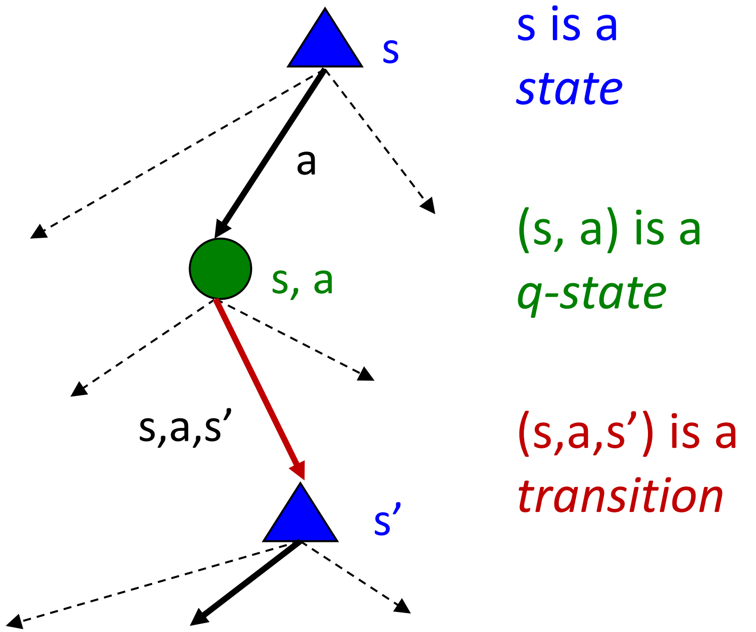
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

- Repeat until convergence: $|V_{k+1}(s) - V_k(s)| \leq \epsilon$ for all s
- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values $V_k(s) \rightarrow V^\star(s)$



Recap: Value Functions

- The state value function:
  - $V^\star(s)$ = expected utility starting from s and acting optimally
- The state-action value function:
  - $Q^\star(s,a)$ = expected utility 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:
  - $\pi^\star(s)$ = optimal action from state s
  - $\pi^\star(s) = \underset{a}{\operatorname{argmax}}\, Q^\star(s,a)$



Remark:
In limited-time game, optimal policy is time-dependent, while for unlimited-time game, it is time independent.