

CMPUT 366: Intelligent Systems

Instructor: Rich Sutton
Dept of Computing Science
University of Alberta
richsutton.com

Intelligent Systems

- Introduction to Artificial Intelligence (AI)
- Introduction to the Science and Technology of Mind
 - touches on control theory, psychology, operations research, philosophy, and neuroscience
- A technical and conceptual foundation for understanding this large and complex set of issues



Boston Dynamics

The coming of artificial intelligence

- When people finally come to understand the principles of intelligence—what it is and how it works—well enough to design and create beings as intelligent as ourselves
- A fundamental goal for science, engineering, the humanities, ...for all mankind
- It will change the way we work and play, our sense of self, life, and death, the goals we set for ourselves and for our societies
- But it is also of significance beyond our species, beyond history
- It will lead to new beings and new ways of being, things inevitably *much more powerful than our current selves*

Milestones in the development of life on Earth

The Age of
Replicators

The Age of
Design

year	Milestone	
14Bya	Big bang	
4.5Bya	formation of the earth and solar system	
3.7Bya	origin of life on earth (formation of first replicators)	
	DNA and RNA	
1.1Bya	sexual reproduction	
	multi-cellular organisms	
	nervous systems	
1Mya	humans	Self-replicated things most prominent
	culture	
100Kya	language	
10Kya	agriculture, metal tools	
5Kya	written language	
200ya	industrial revolution	
	technology	
70ya	computers	Designed things most prominent
?	nanotechnology	
	artificial intelligence	
	super-intelligence	
	...	

AI is a great scientific prize

- cf. the discovery of DNA, the digital code of life, by Watson and Crick (1953)
- cf. Darwin's discovery of evolution, how people are descendants of earlier forms of life (1860)
- cf. the splitting of the atom, by Hahn (1938)
 - leading to both atomic power and atomic bombs

Socrative.com, Room 568225

When will we understand the principles of intelligence well enough to create, using technology, artificial minds that rival our own in skill and generality?

Which of the following best represents your current views?

- A. Never
- B. Not during your lifetime
- C. During your lifetime, but not before 2045
- D. Before 2045
- E. Before 2035

Is human-level AI *possible*?

- If people are biological machines, then eventually we will reverse engineer them, and understand their workings
- Then, surely we can make improvements
 - with materials and technology not available to evolution
 - how could there not be something we can improve?
 - design can overcome local minima, make great strides, try things much faster than biology

Yes

If AI is possible, then will it *eventually*, inevitably happen?

- No. Not if we destroy ourselves first
- If that doesn't happen, then there will be strong, multi-incremental economic incentives pushing inexorably towards human and super-human AI
- It seems unlikely that they could be resisted
 - or successfully forbidden or controlled
 - there is too much value, too many independent actors

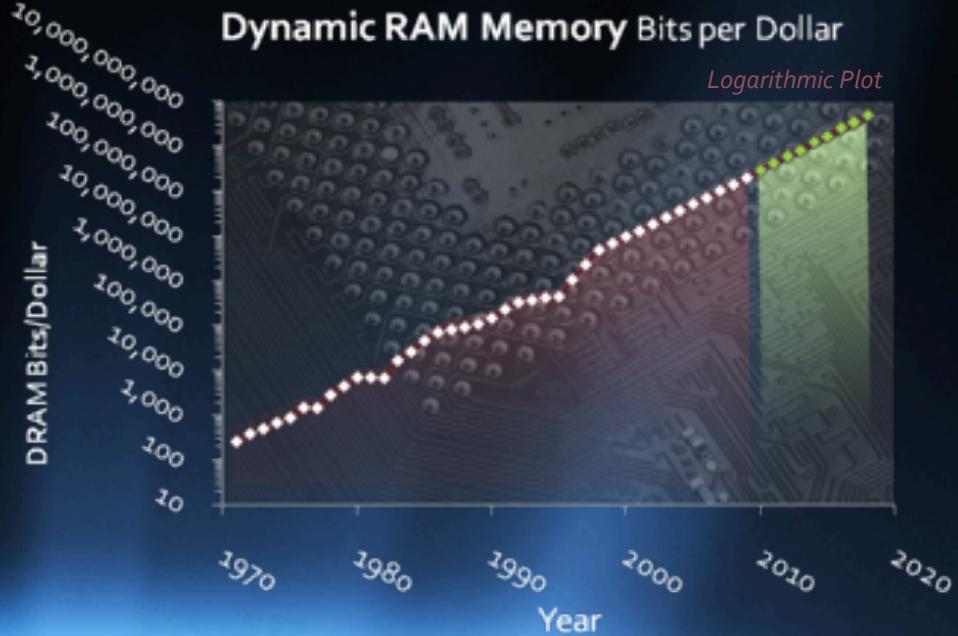
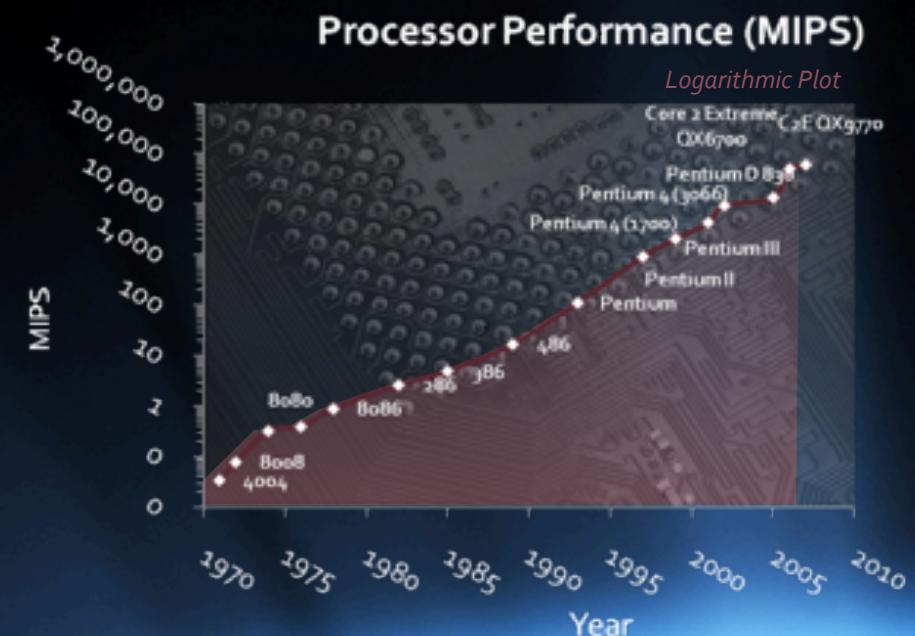
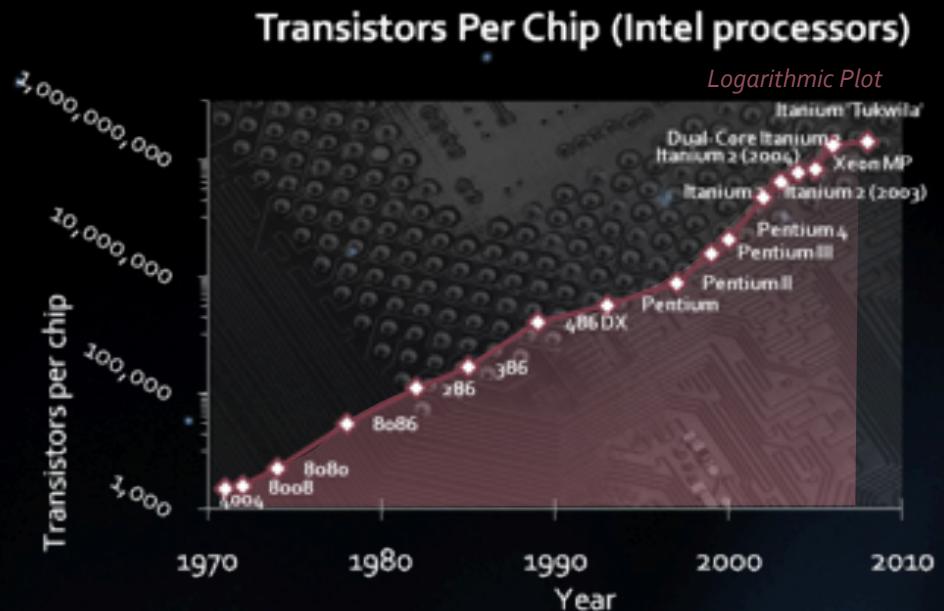
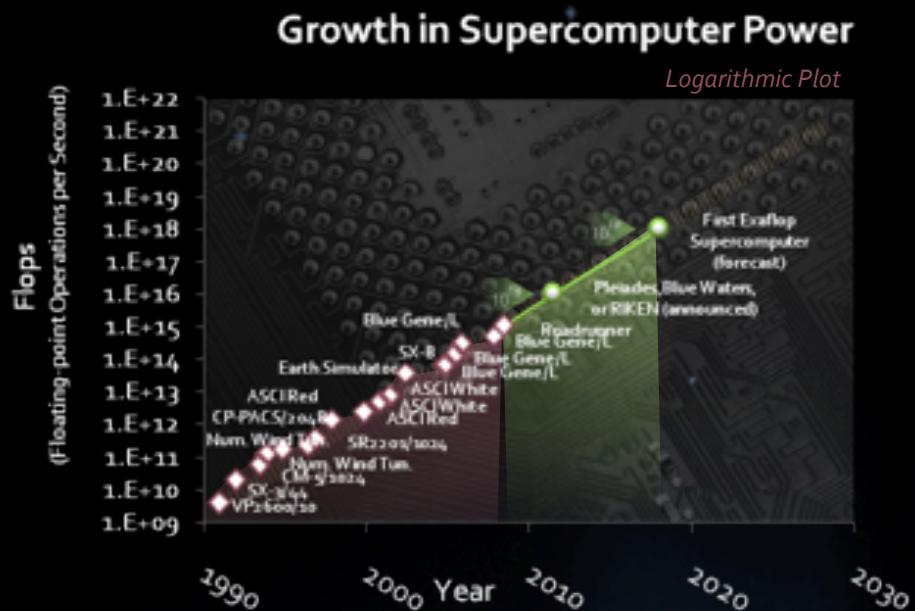
Very probably, say 90%

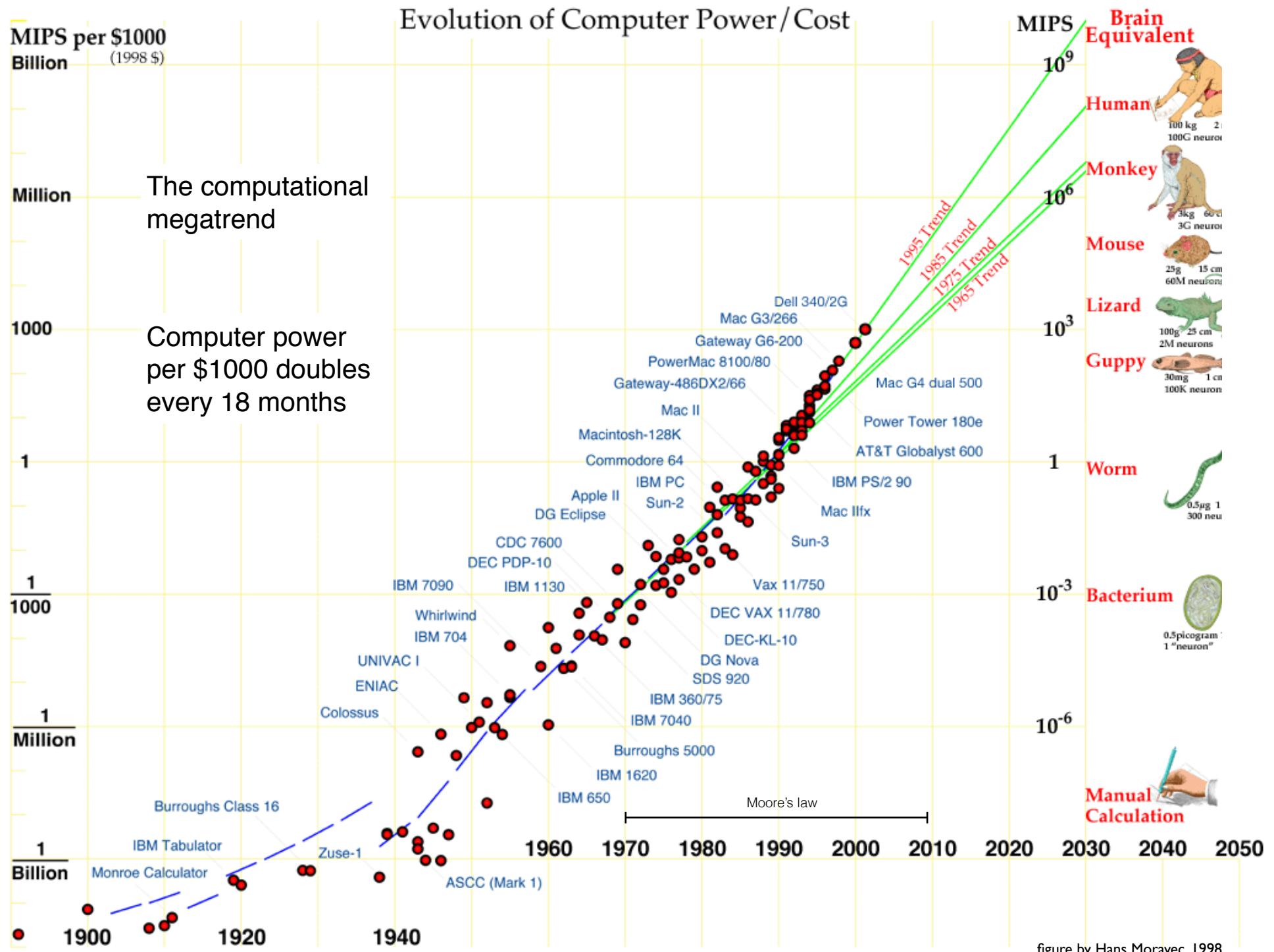
When will human-level AI first be created?

- No one knows of course; we can make an educated guess about the probability distribution:
 - 25% chance by 2030
 - 50% chance by 2040
 - 10% chance never
- Certainly a significant chance within all of our expected lifetimes
 - We should take the possibility into account in our career plans

the computational mega-trend

- [effective computation per \$ increases exponentially, with a doubling time of 18-24 months
- [this trend has held for the last sixty years
- [and will continue for the foreseeable future





The possibility of AI is near

— [“We are nearing an important milestone in the history of life on earth, the point at which we can construct machines with the potential for exhibiting an intelligence comparable to ours.” – David Waltz, 1988 (recent president of AAAI)]

— [Should occur in ≈2030 for ≈\$1000]

— [We don’t yet have the needed AI “software” (designs, ideas)]

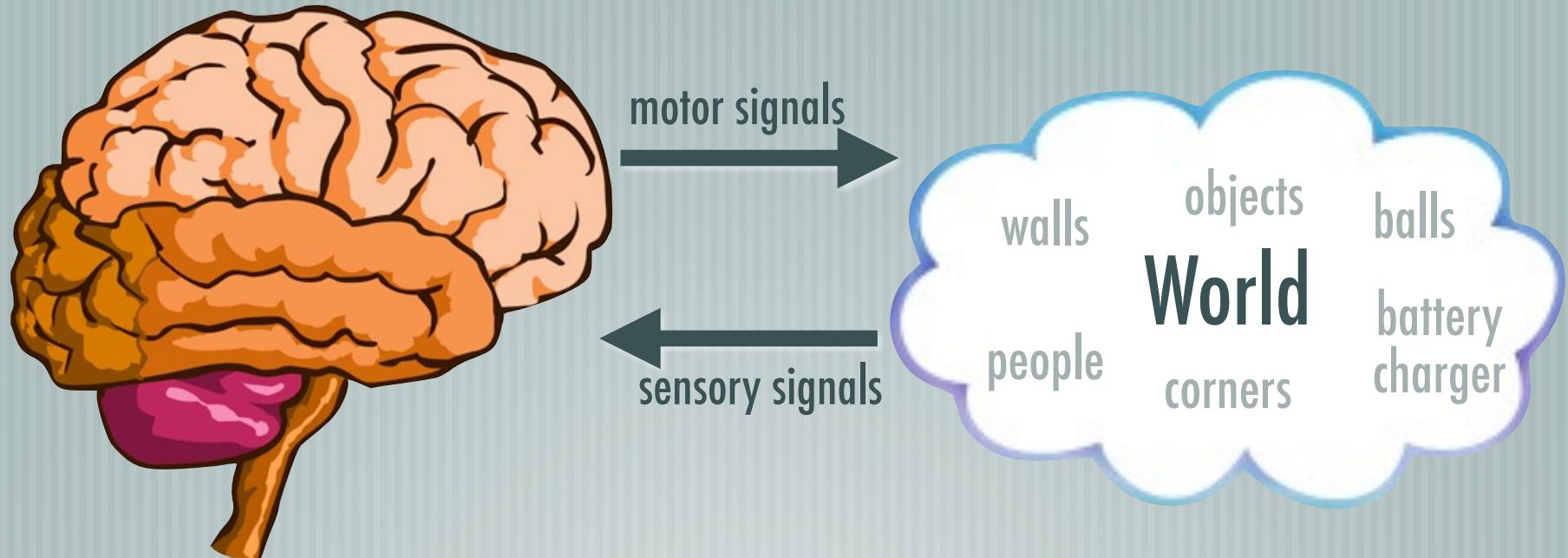
— [But the hardware will be a tremendous economic spur to development of the ideas...perhaps at nearly the same time]

For you, which of the following are essential abilities of an intelligent system that you would like to learn about (say in this course)?

The ability to:

- A. sense and perceive the external world
- B. choose actions that affect the world
- C. use language and interact with other agents
- D. predict the future
- E. fool people into thinking that you are a person
- F. have and achieve goals
- G. reason symbolically, as in logic and mathematics
- H. reason in advance about courses of action before picking the best
- I. learn by trying things out and subsequently picking the best
- J. have emotions, pleasure and pain
- K. other?

Minds are sensori-motor information processors



the mind's job is to predict and control its sensory signals

Course Overview

- Main Topics:
 - Learning (by trial and error)
 - Planning (search, reason, thought, cognition)
 - Prediction (evaluation functions, knowledge)
 - Control (action selection, decision making)
- Recurring issues:
 - Demystifying the illusion of intelligence
 - Purpose (goals, reward) vs Mechanism

Order of Presentation

- Control: Bandits and Markov decision processes
- Stochastic planning (dynamic programming)
- Model-free reinforcement learning
- Planning with a learned model
- Learning with approximations

Instruction Team

- Prof: Rich Sutton
- Secondary instructor: Harm van Seijen
- TAs (grad students doing research in AI)
 - Mohammad Ajallooeian
 - Pooria Joulani
 - Robert Post
 - (Rupam Mahmood)

Provisional Schedule of Classes and Assignments

class num	date	lecture topic	Reading assignment (in advance)	Assignment due
1	Tue, Sep 1, 2015	The Magic of Artificial Intelligence; reasons for taking the course	Read section 1 of the Wikipedia entry for “the technological singularity” and the other things listed in topic 2 on the course moodle page	
2	Thu, Sep 3, 2015	Bandit problems	Read Sutton & Barto Chapters 1 and 2 (Section 2.7 optional)	
3	Tue, Sep 8, 2015	Defining “Intelligent Systems”	Read the definition given for artificial intelligence in Wikipedia and in the Nilsson book on p13; google for and read “John McCarthy basic questions”, and “the intentional stance (dictionary of philosophy of mind)”	
4	Thu, Sep 10, 2015	Markov decision problems	Read Sutton & Barto Chapter 3 thru Section 3.5	
5	Tue, Sep 15, 2015	Returns, value functions	Read rest of Sutton & Barto Chapter 3	W1
6	Thu, Sep 17, 2015	Returns, value functions	See Sutton & Barto Summary of Notation	
7	Tue, Sep 22, 2015	Dynamic programming (planning)	Sutton & Barto Chapter 4	W2
8	Thu, Sep 24, 2015	Monte Carlo Learning	Sutton & Barto Chapter 5	
9	Tue, Sep 29, 2015	More Monte Carlo Learning	Sutton & Barto Chapter 5	W3
10	Thu, Oct 1, 2015	Temporal-difference learning	Sutton & Barto Chapter 6 thru Section 6.3	
11	Tue, Oct 6, 2015	Temporal-difference learning	Sutton & Barto rest of Chapter 6	
12	Thu, Oct 8, 2015	Models and planning	Sutton & Barto Chapter 8 thru Section 8.3	W4
13	Tue, Oct 13, 2015	Models and planning	Sutton & Barto rest of Chapter 8	
14	Thu, Oct 15, 2015	Review	Sutton & Barto Chapters 2-6	W5
15	Tue, Oct 20, 2015	Midterm Exam	No new reading	
16	Thu, Oct 22, 2015	Eligibility traces	Sutton & Barto Chapter 7	
17	Tue, Oct 27, 2015	Online linear supervised learning	Nilsson Sec. 2.2.1 and Ch. 4	W6
18	Thu, Oct 29, 2015	Value Function Approximation	Sutton & Barto Ch. 9.1-3	
19	Tue, Nov 3, 2015	Value Function Approximation	Sutton & Barto rest of Chapter 9	
20	Thu, Nov 5, 2015	Deterministic Tree-based Planning	Wikipedia on Iterative deepening depth-first search, and alpha-beta pruning	P1
21	Tue, Nov 17, 2015	Heuristic Search (Holte)	Wikinedia on the A* search algorithm	

Course Information

- Course Moodle page <https://eclass.srv.ualberta.ca/course/view.php?id=13130>
 - some official information
 - discussion list!
- Course Dropbox (see moodle page for link)
 - schedule, assignments, slides, projects
- Lab is on Monday, 5-7:50
 - a good place to do your assignments

Textbooks

- Readings will be from web sources plus the following two textbooks (both of which are available as online electronically and open-access):
 - *Reinforcement Learning: An Introduction*, by R Sutton and A Barto, MIT Press.
 - we will use the in-progress, online 2nd edition
 - *The Quest for AI*, by N Nilsson, Cambridge, 2010 (pdf)

Evaluation

- Final Exam – 40%
- Midterm – 20%
- ≈1 assignment per week, due at the beginning of class
 - 5-6 written assignments – 23%
 - 3 programming projects – 17%
(later in the course)

Grades

- Will be assigned on an *absolute scale* based on weighted % of points received:

A+	[90 -- 100]%
A	[85 -- 90]%
A-	[80 -- 85)%
B+	[75 -- 80)%
B	[70 -- 75)%
B-	[65 -- 70)%
C+	[60 -- 65)%
C	[55 -- 60)%
C-	[50 -- 55)%
D+	[45 -- 50)%
D	[40 -- 45)%
F	[0 -- 40)%

Collaboration

- Working together to solve the problems is encouraged
- But you must write-up your answers individually (or in teams of two for the programming projects)
- You must acknowledge all the people you talked with in solving the problems
- You must completely understand and be able to justify your answers
- I reserve the right to check your understanding and possibly revise your marks

Use the Labs!

- Formally, the labs are optional, but they are a good place to
 - get practice with problems like those on the assignments and exams
 - discuss the assignments and get questions answered
 - find a partner for the programming projects
 - cooperate to understand the problems, projects, and the textbook material

Contacting us...

- Use the course discussion feature on the moodle page
 - Start a discussion
 - Read by prof and TAs
 - Remember: public!
 - Meeting w/ profs, TAs: at office hours or by arrangement

Prerequisites

- Some comfort or interest in thinking abstractly and with mathematics
- Elementary statistics, probability theory
 - conditional expectations of random variables
 - there will be a lab session devoted to a tutorial review of basic probability
- Basic linear algebra: vectors, vector equations, gradients
- Basic programming skills (Python)
 - If Python is a problem, choose a partner who is already comfortable with Python

Policies on Integrity

- Do not cheat on assignments:
Discuss only general approaches to problem
- Do not take written notes on other's work
- Respect the lab environment. Do not:
 - Interfere with operation of computing system
 - Interfere with other's files
 - Change another's password
 - Copy another's program
 - etc.
- Cheating is reported to university whereupon it is out of our hands
- Possible consequences:
 - A mark of 0 for assignment
 - A mark of 0 for the course
 - A permanent note on student record
 - Suspension / Expulsion from university

Academic Integrity

- The University of Alberta is committed to the highest standards of academic integrity and honesty. Students are expected to be familiar with these standards regarding academic honesty and to uphold the policies of the University in this respect. Students are particularly urged to familiarize themselves with the provisions of the Code of Student Behavior (online at www.ualberta.ca/secretariat/appeals.htm) and avoid any behavior which could potentially result in suspicions of cheating, plagiarism, misrepresentation of facts and / or participation in an offence. Academic dishonesty is a serious offence and can result in suspension or expulsion from the University.

for next time...

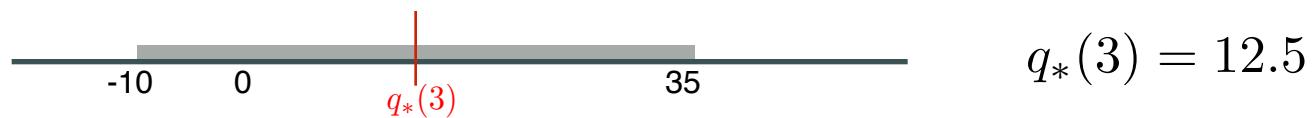
- Read Chapters 1 & 2 of Sutton & Barto text
(online)

Multi-arm Bandits

Sutton and Barto, Chapter 2

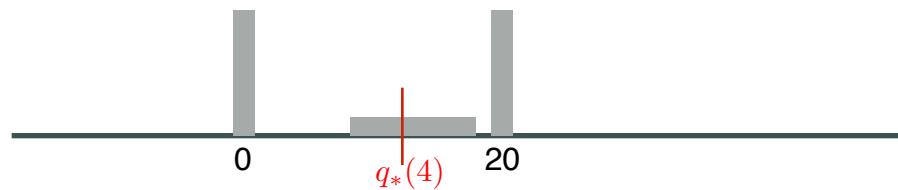
You are the algorithm! (bandit I)

- Action 1 — Reward is always 8
 - value of action 1 is $q_*(1) = 8$
- Action 2 — 88% chance of 0, 12% chance of 100!
 - value of action 2 is $q_*(2) = .88 \times 0 + .12 \times 100 = 12$
- Action 3 — Randomly between -10 and 35, equiprobable



$$q_*(3) = 12.5$$

- Action 4 — a third 0, a third 20, and a third from $\{8,9,\dots,18\}$



$$q_*(4) = \frac{1}{3} \times 0 + \frac{1}{3} \times 20 + \frac{1}{3} \times 13 = 0 + \frac{20}{3} + \frac{13}{3} = \frac{33}{3} = 11$$

The k -armed Bandit Problem

- On each of an infinite sequence of *time steps*, $t=1, 2, 3, \dots$, you choose an action A_t from k possibilities, and receive a real-valued *reward* R_t
- The reward depends only on the action taken; it is identically, independently distributed (i.i.d.):

$$\mathbb{E}[R_t | A_t = a] \doteq q_*(a) \quad \text{true values}$$

- These true values are *unknown*. The distribution is unknown
- Nevertheless, you must maximize your total reward
- You must both try actions to learn their values (*explore*), and prefer those that appear best (*exploit*)

The Exploration/Exploitation Dilemma

- Suppose you form estimates

$$Q_t(a) \approx q_*(a), \quad \forall a \qquad \text{action-value estimates}$$

- Define the *greedy action* at time t as

$$A_t^* \doteq \arg \max_a Q_t(a)$$

- If $A_t = A_t^*$ then you are *exploiting*
If $A_t \neq A_t^*$ then you are *exploring*
- You can't do both, but you need to do both
- You can never stop exploring, but maybe you should explore less with time. Or maybe not.

Action-Value Methods

- Methods that learn action-value estimates and nothing else
- For example, estimate action values as *sample averages*:

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken upto } t}{\text{number of times } a \text{ taken upto } t} = \frac{\sum_{i=1}^{t-1} R_t \cdot \mathbb{1}\{A_t = a\}}{\sum_{i=1}^{t-1} \mathbb{1}\{A_t = a\}}$$

- The sample-average estimates converge to the true values

$$\lim_{N_t(a) \rightarrow \infty} Q_t(a) = q_*(a)$$

- If the action is taken an infinite number of times

ϵ -Greedy Action Selection

- In greedy action selection, you always exploit
- In ϵ -greedy, you are usually greedy, but with probability ϵ you instead pick an action at random (possibly the greedy action again)
- This is perhaps the simplest way to balance exploration and exploitation

The 10-armed Testbed

- $k = 10$ possible actions
- Each reward is chosen from a normal distribution with unit variance:

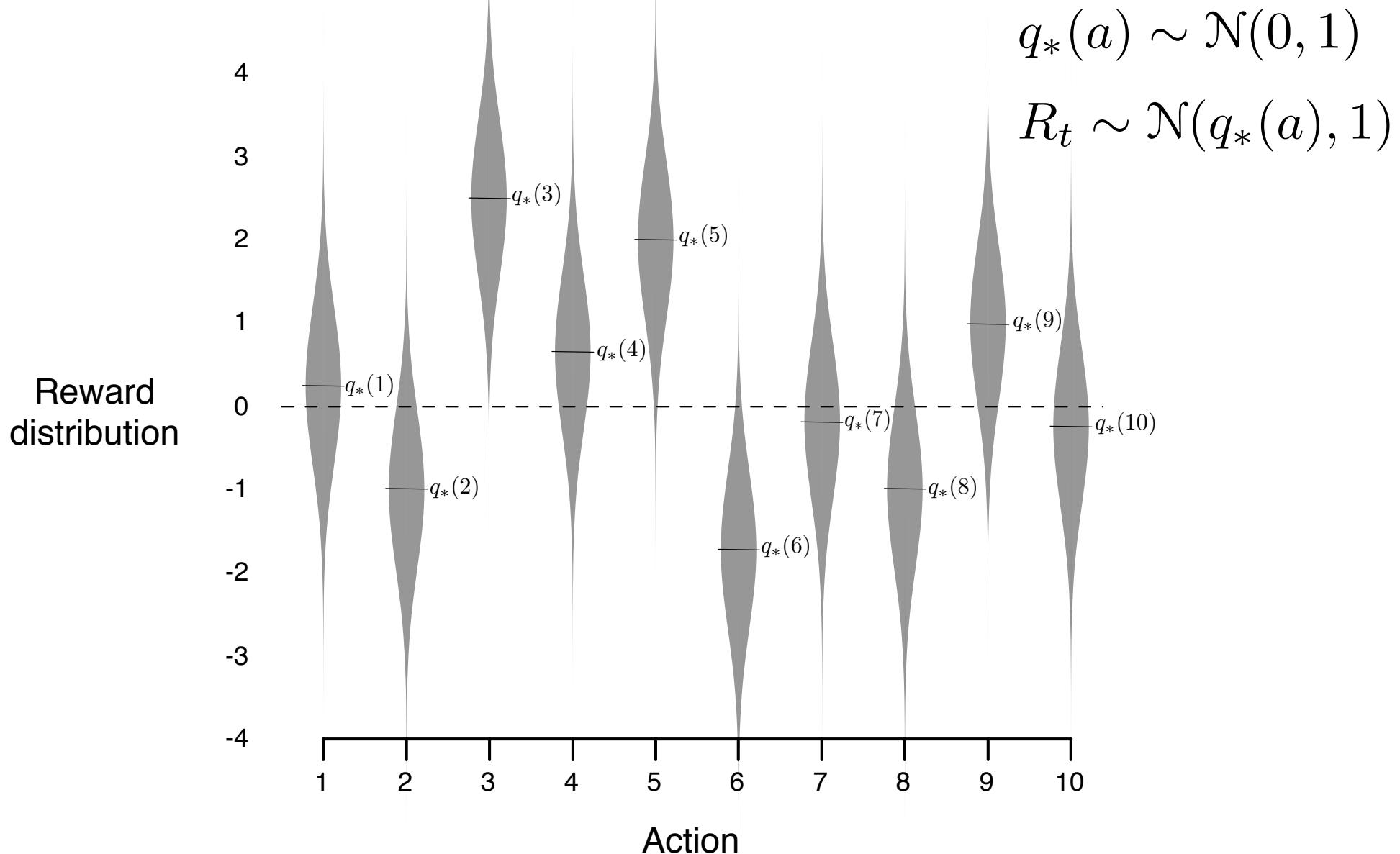
$$R_t \sim \mathcal{N}(q_*(a), 1)$$

- The true values are also selected from a normal distribution:

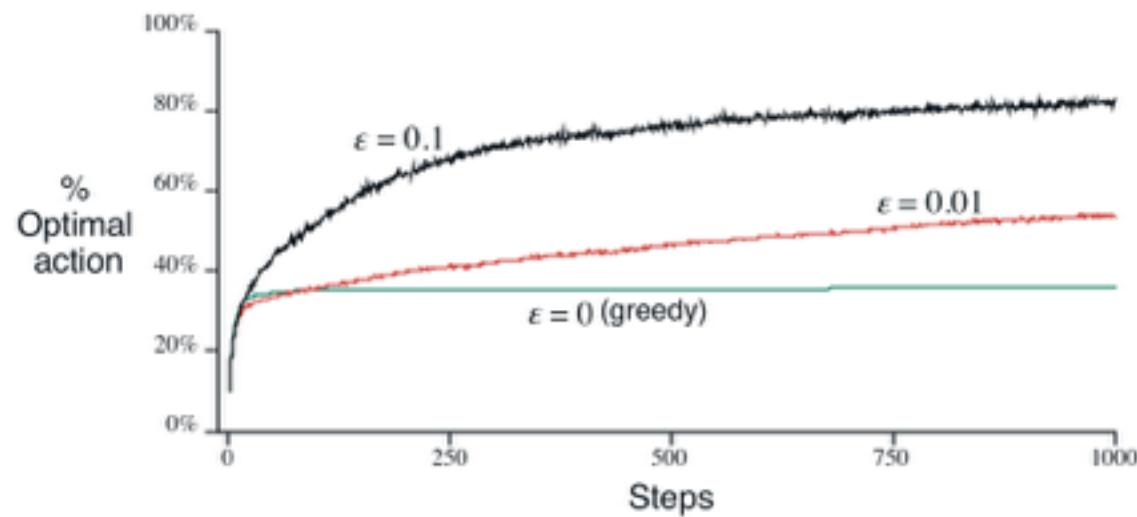
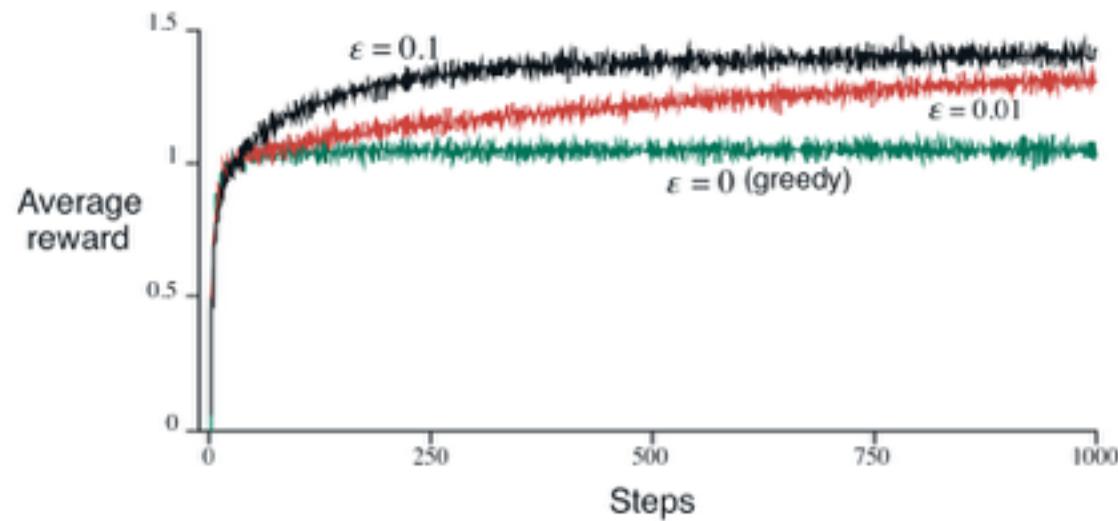
$$q_*(a) \sim \mathcal{N}(0, 1)$$

- This creates a single bandit task
- The testbed consists of 2000 tasks. Each runs for 1000 steps

The 10-armed Testbed



ϵ -Greedy Methods on the 10-Armed Testbed



Incremental Implementation

- To simplify notation, let us focus on one action
 - We consider only its rewards, and its estimate after $n+1$ rewards:
$$Q_{n+1} \doteq \frac{1}{n} \sum_{i=1}^n R_i$$
- How can we do this incrementally (without storing all the rewards)?
- Could store a running sum and count (and divide), or equivalently:
$$Q_{n+1} = Q_n + \frac{1}{n} [R_n - Q_n]$$
- This is a standard form for learning/update rules:

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize} [\text{Target} - \text{OldEstimate}]$$

Tracking a Nonstationary Problem

- Suppose the true action values change slowly over time
 - then we say that the problem is *nonstationary*
- In this case, sample averages are not a good idea (Why?)
- Better is an “exponential, recency-weighted average”:

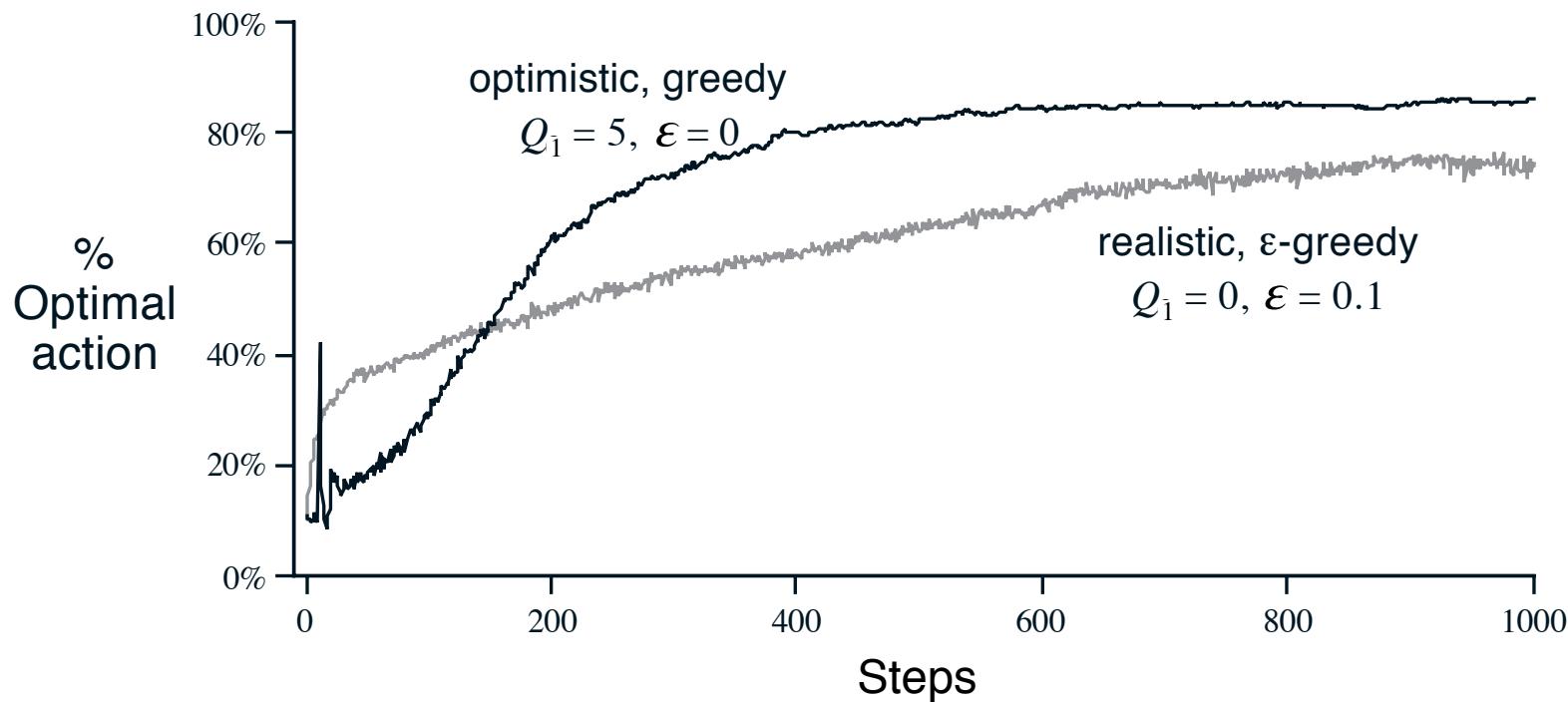
$$\begin{aligned} Q_{n+1} &= Q_n + \alpha [R_n - Q_n] \\ &= (1 - \alpha)^n Q_1 + \sum_{i=1}^n \alpha(1 - \alpha)^{n-i} R_i \end{aligned}$$

where α is a constant, *step-size parameter*, $0 < \alpha \leq 1$

- There is bias due to Q_1 that becomes smaller over time

Optimistic Initial Values

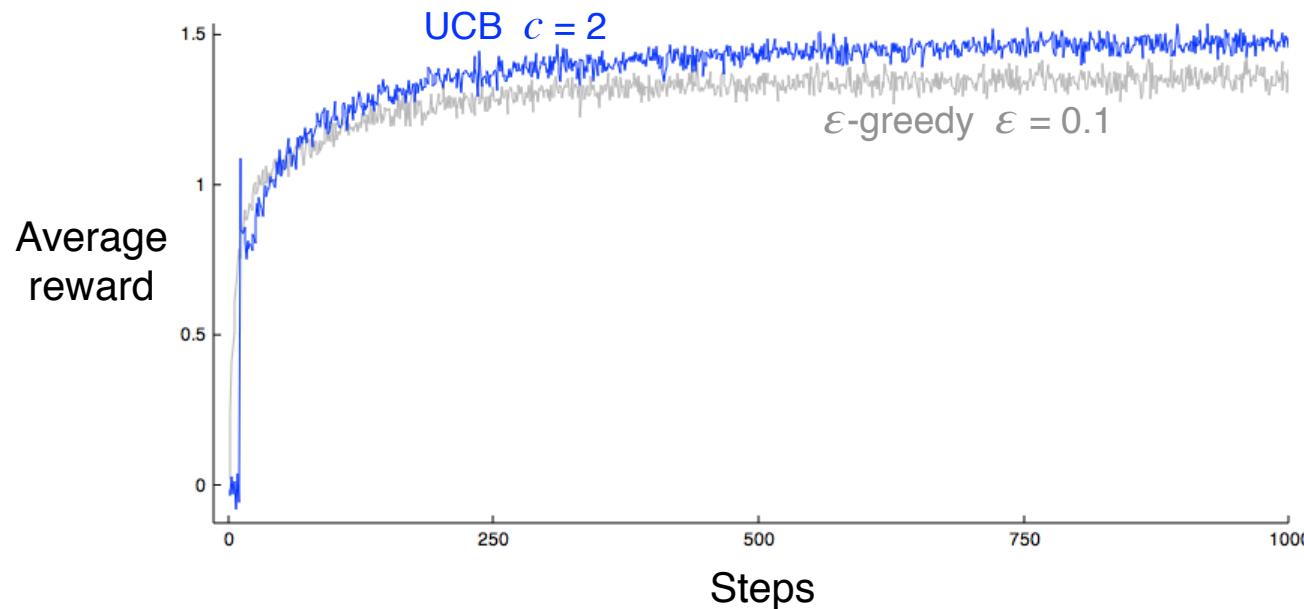
- All methods so far depend on $Q_1(a)$, i.e., they are biased.
So far we have used $Q_1(a) = 0$
- Suppose we initialize the action values *optimistically* ($Q_1(a) = 5$),
e.g., on the 10-armed testbed (with $\alpha = 0.1$)



Upper Confidence Bound (UCB) action selection

- A clever way of reducing exploration over time
- Estimate an upper bound on the true action values
- Select the action with the largest (estimated) upper bound

$$A_t \doteq \arg \max_a \left[Q_t(a) + c \sqrt{\frac{\log t}{N_t(a)}} \right]$$



Gradient-Bandit Algorithms

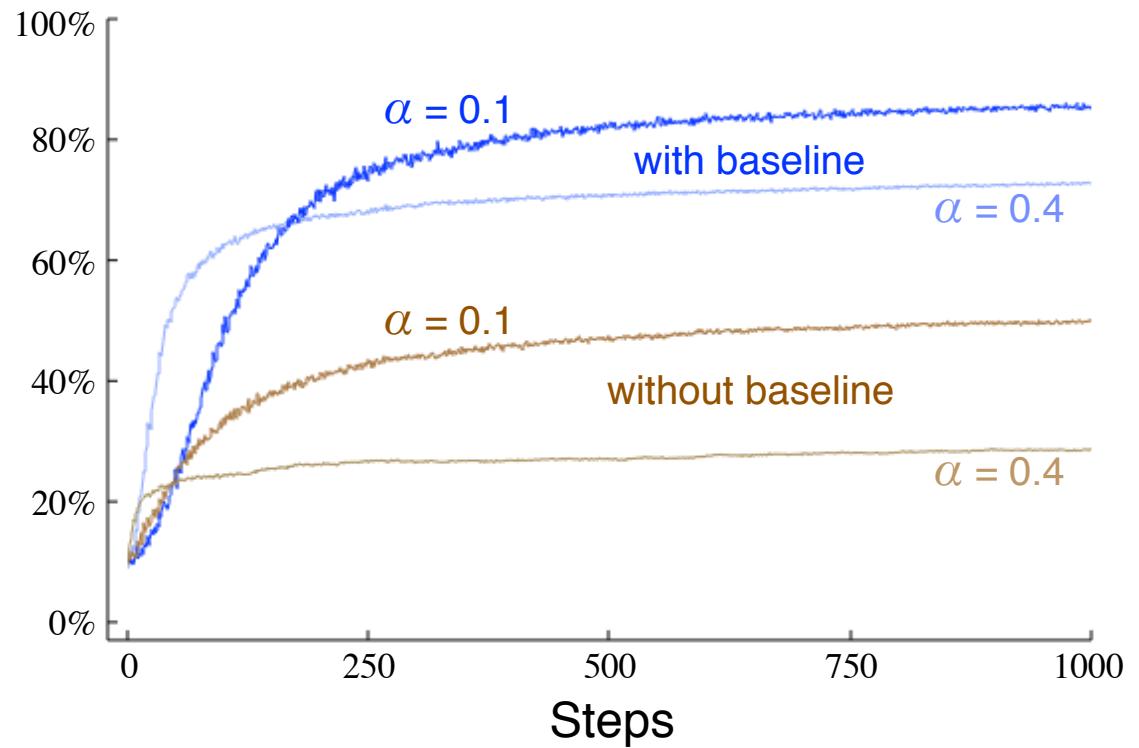
- Let $H_t(a)$ be a learned preference for taking action a

$$\Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^k e^{H_t(b)}} \doteq \pi_t(a)$$

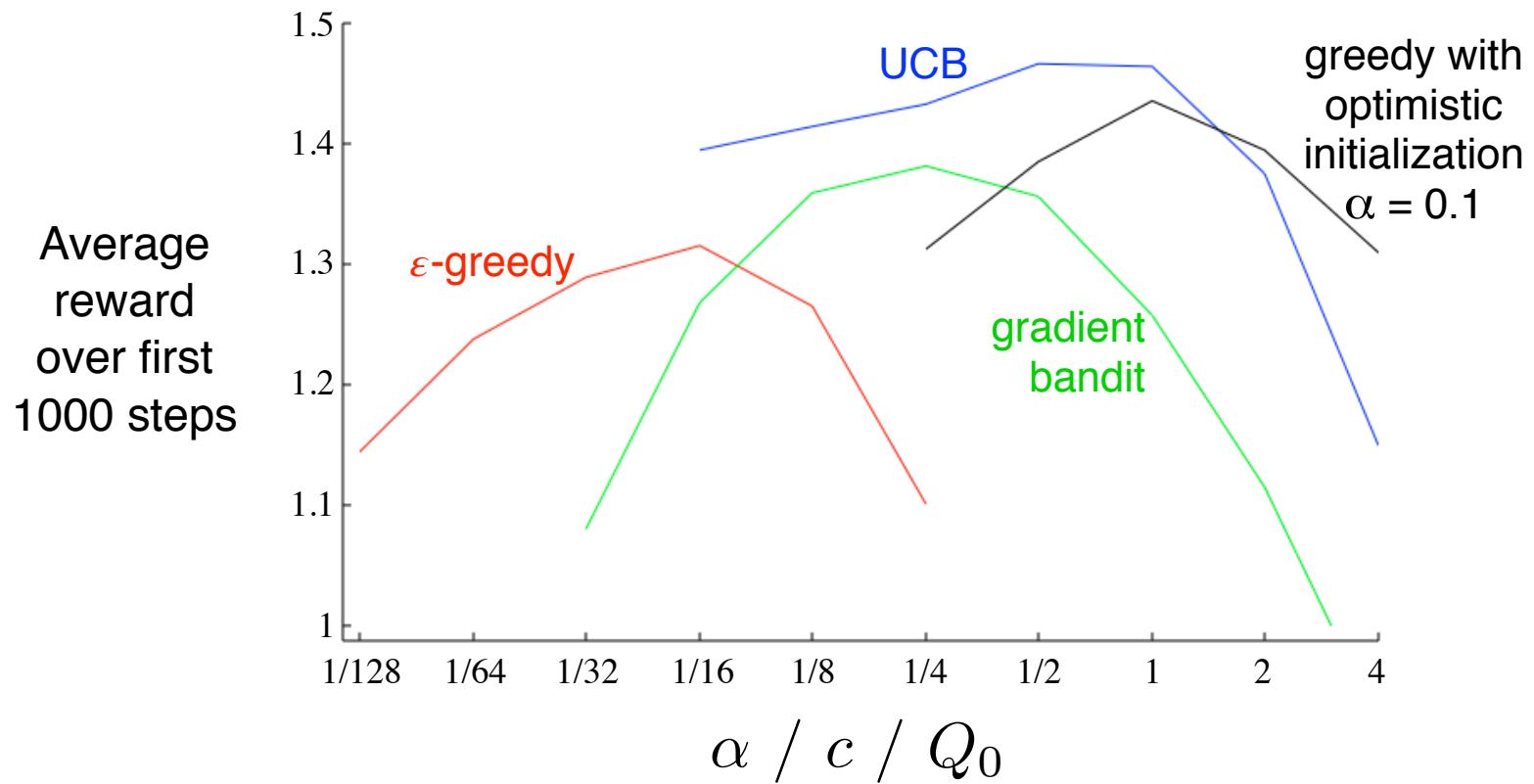
$$H_{t+1}(A_t) \doteq H_t(A_t) + \alpha \left(R_t - \bar{R}_t \right) \left(\mathbb{1}\{A_t = a\} - \pi_t(A_t) \right)$$

$$\bar{R}_t \doteq \sum_{i=1}^t R_i$$

%
Optimal
action



Summary Comparison of Bandit Algorithms

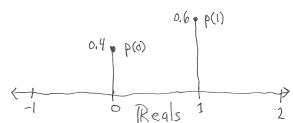


Conclusions

- These are all simple methods
 - but they are complicated enough—we will build on them
 - we should understand them completely
 - there are still open questions
- Our first algorithms that learn from evaluative feedback
 - and thus must balance exploration and exploitation
- Our first algorithms that appear to have a goal
 - that learn to maximize reward by trial and error

Random Variables, Expectations, Estimates, and a Learning Rule

Let $R \in \{0, 1\}$ $p(0) = 0.4$ $p(1) = 0.6$ $p(r) = \Pr[R=r]$



In general, consider a r.v. $R \in \mathbb{R} \subset \text{Reals}$ $\{R\} < \infty$

$R \sim P$ $P: \mathbb{R} \rightarrow [0, 1]$ s.t. $\sum_{r \in \mathbb{R}} p(r) = 1$

The expectation, or expected value of R is

$$\text{Defn: } E[R] = \sum_{r \in \mathbb{R}} p(r) \cdot r$$

Consider a sequence of r.v.s $R_t \sim P$ $t=1, 2, 3, \dots$

that are i.i.d. - identically, independently distributed

Consider the sequence of sample averages:

$$Q_{t+1} = \frac{R_1 + R_2 + \dots + R_t}{t} \approx E[R_t]$$

$$\lim_{t \rightarrow \infty} Q_t = E[R_t]$$

$$\text{Var}[Q_t] = E[(Q_t - E[R_t])^2] \propto \frac{1}{t}$$

Sample averages can be computed incrementally:

$$Q_{t+1} = Q_t + \frac{1}{t} [R_t - Q_t]$$

Which is a special case of our standard learning rule:

$$\text{NewEst.} = \text{OldEst} + \text{StepSize} \cdot [\text{Target} - \text{OldEst}]$$

$\underbrace{\phantom{\text{Target} - \text{OldEst}}}_{\text{error}}$

n-Armed Bandits

n actions $\in \{1, 2, 3, \dots, n\} = A$ $|A| = n$ "policy"

A sequence of r.v.s $A_t \in A$ $A_t \sim \pi_t$ \leftarrow not iid

$$t = 1, 2, 3, \dots \quad \pi_t : A \rightarrow [0, 1], \quad \sum_{a \in A} \pi_t(a) = 1$$

A sequence of r.v.s $R_t \in R \subset \text{Reals}$

$$p(r|a) = \Pr[R_t = r | A_t = a]$$

$$p : R \times A \rightarrow [0, 1]$$

$$\sum_{r \in R} p(r|a) = 1 \quad \forall a \in A$$

The value of action a :

$$q^*(a) = E[R_t | A_t = a] = \sum_{r \in R} p(r|a) \cdot r$$

Estimates

$$Q_t(a) \approx q^*(a)$$

greedy policy: $A_t = \arg \max_a Q_t(a)$

OR:

$$\pi_t(a) = \begin{cases} 1 & \text{if } a = \arg \max_{a'} Q_t(a') \\ 0 & \text{otherwise} \end{cases}$$

ϵ -greedy policy:

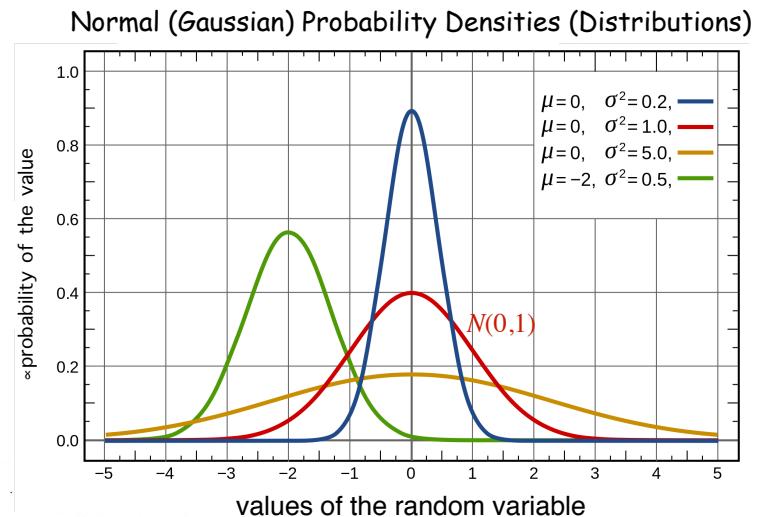
$$\pi_t(a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{n} & \text{if } a = \arg \max_{a'} Q_t(a') \\ \frac{\epsilon}{n} & \text{otherwise} \end{cases}$$

Ten-Armed Testbed

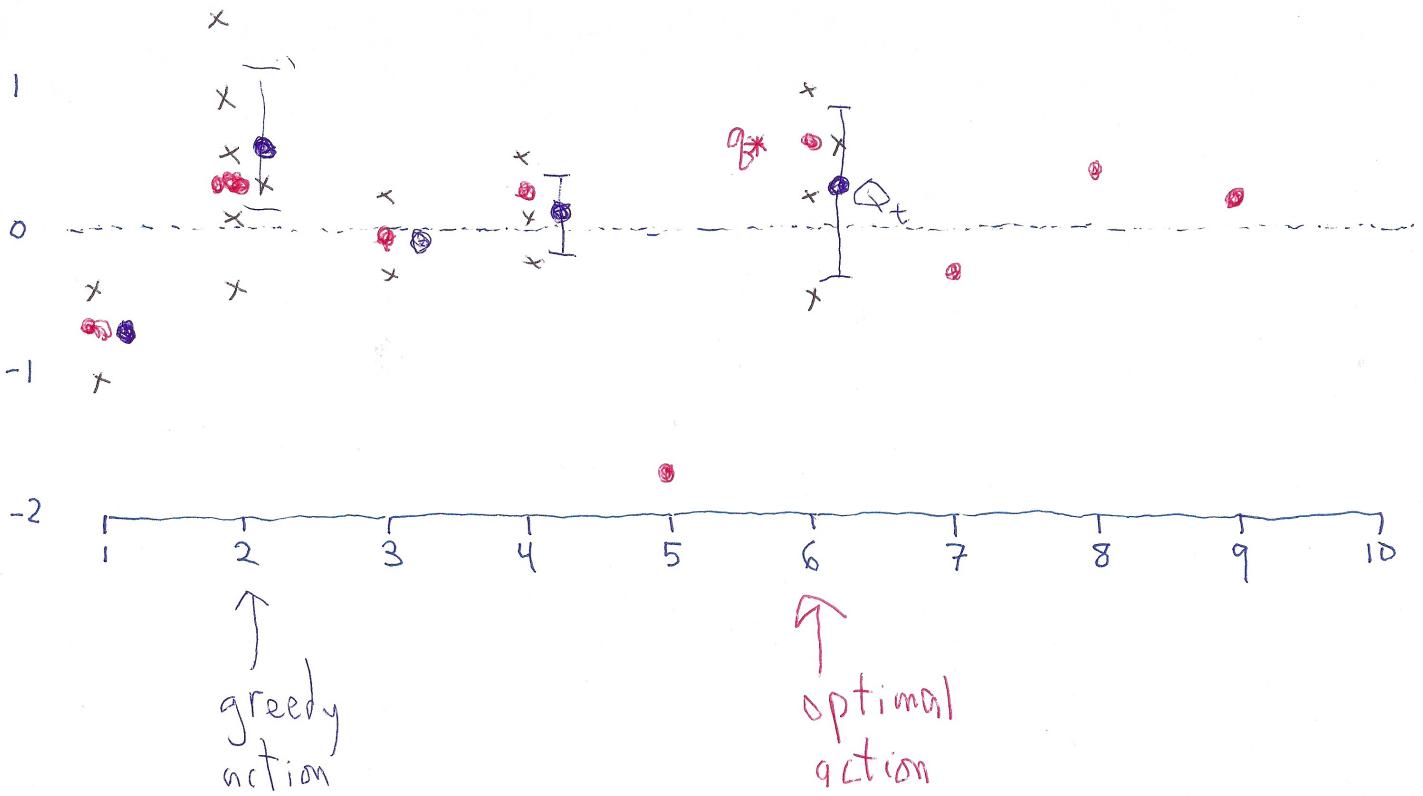
10 actions, $n=10$

$$q^*(a) \sim N(0, 1)$$

mean Variance



2



and

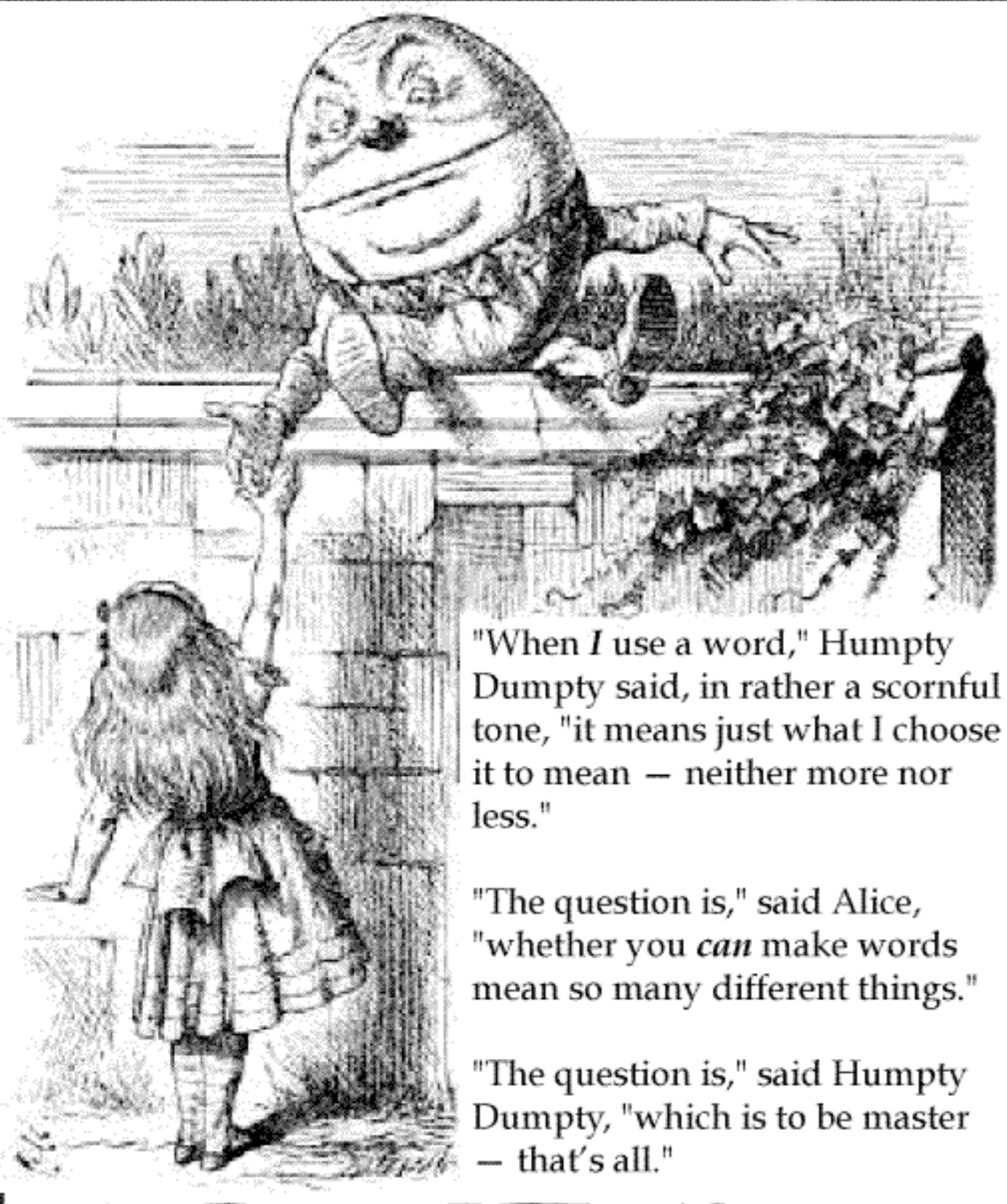
$$Q_{t+1}(A_t) = Q_t(A_t) + \alpha [R_t - Q_t(A_t)]$$

$$Q_{t+1}(a) = Q_t(a) \quad \forall a \neq A_t \quad a \in \mathcal{A}$$

What you have learned from bandits

- ϵ -greedy policies
- the difference between a sample, an estimate, and a true expected value - R_t, Q_t, q^*
- the difference between the greedy action and the optimal action
- a learning rule; how learning can be seen as computing an average in an incremental way
- seen a complete example of goal-seeking - both the problem and the solution methods
- seen a complete example of mathematical formalization of an AI problem & solution

Defining “Intelligent Systems”



"When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean — neither more nor less."

"The question is," said Alice, "whether you *can* make words mean so many different things."

"The question is," said Humpty Dumpty, "which is to be master — that's all."

From "Through the Looking Glass"
by Lewis Carroll, 1871

Multiple Perspectivism

- That there are multiple useful perspectives to take
 - All are valid; They vary in their usefulness
- For example:
 - Reality is objective, material, existing separate from us
 - Reality is a construct, an idea we create to explain our input
 - Both are useful viewpoints, let us keep both
- The most liberal of philosophies
- We can still (respectfully) debate over usefulness

Defining “System”

- A thing
 - with some recognizable identity over time (need not be physical)
 - usually with some inputs and outputs
 - may have state
 - sometimes with a goal/purpose

What is intelligence?
What is a mind?

Intelligence is not physical

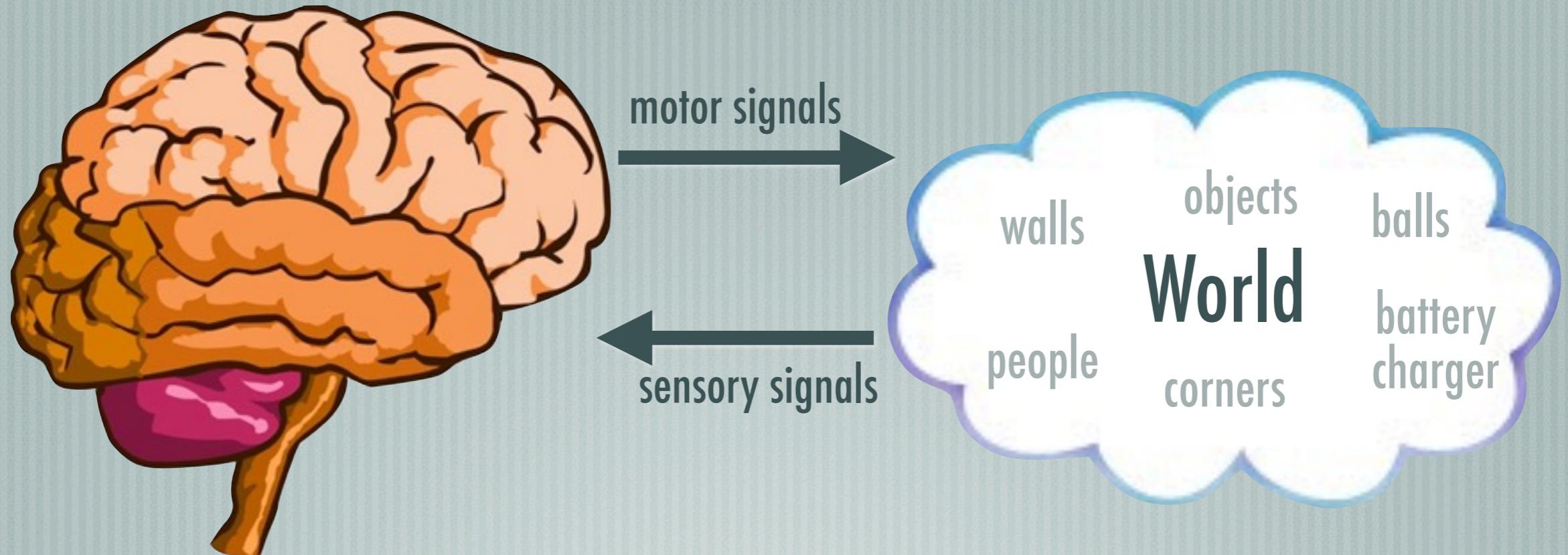
- Brains are physical
- Certainly intelligence is implemented physically
- But the idea of intelligence is part of another perspective
 - that is not physical, but functional?
 - It is computational
- When we talk of intelligence, we are talking about an informational or *computational ability*

Ques: What is the right definition of intelligence?

Ans: The computational part of the ability to:

- A. improve over time with experience
- B. use language, communicate and cooperate with other intelligent agents
- C. achieve goals
- D. predict and control your input signals
- E. There is no such thing as a right definition

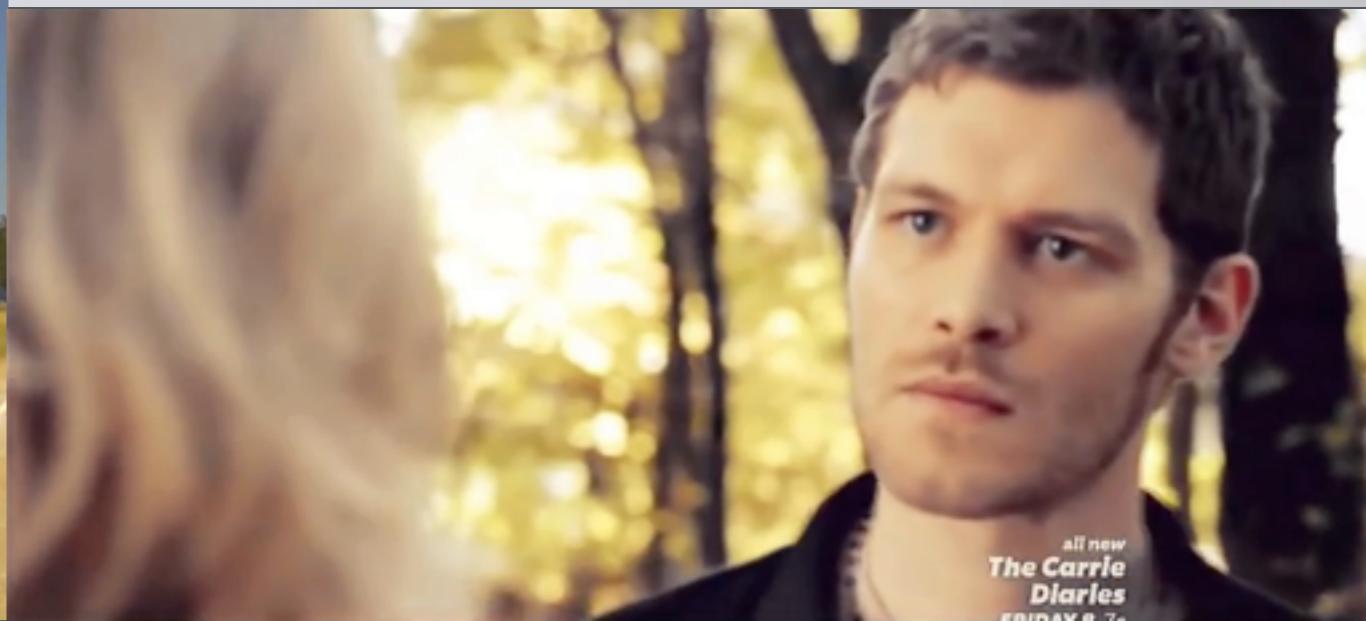
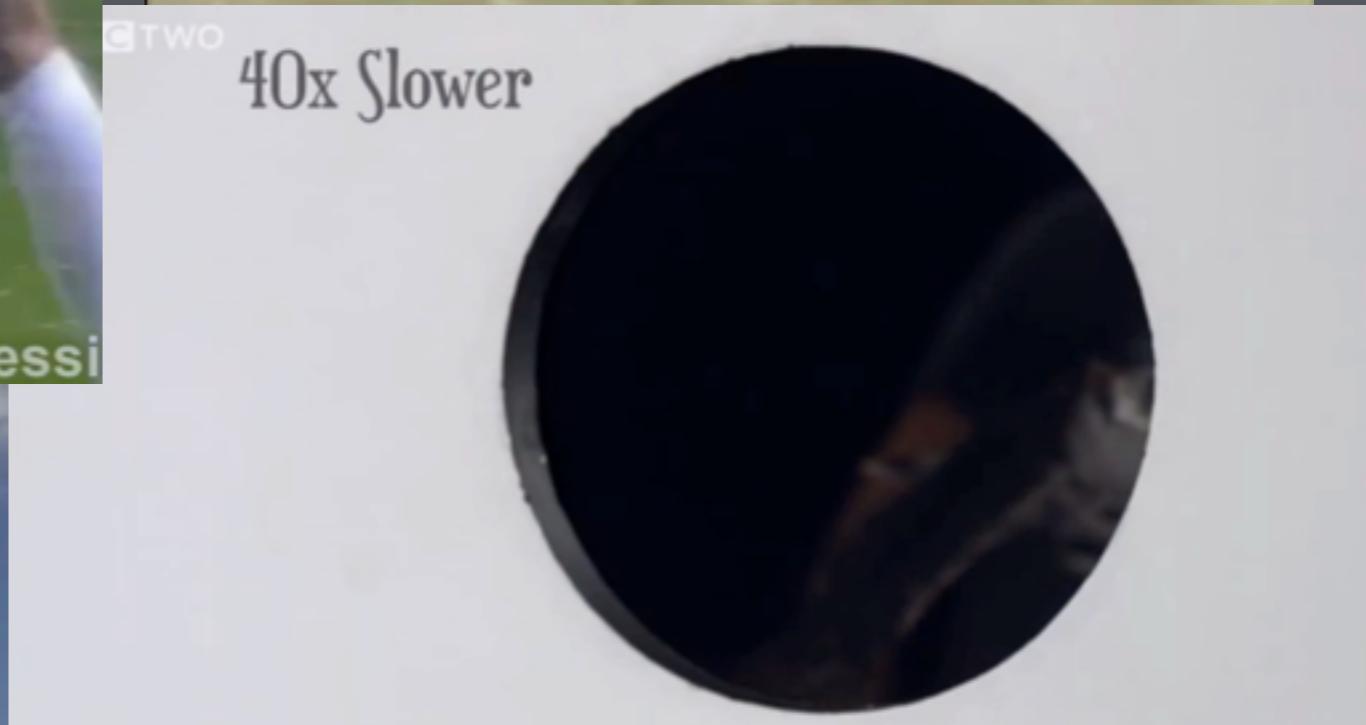
Minds are sensori-motor information processors



the mind's job is to predict and control its sensory signals

the mind's first responsibility is
real-time sensorimotor information processing

- Perception, action, & anticipation
- as fast and reactive as possible



A new view of the AI problem

and its implications for (and against) solution methods

- Minds are real-time information processors interacting with a firehose of data from a complex and arbitrary world
 - we must find *scalable* and *general* methods, to *learn* arbitrary stuff (no domain knowledge, no taking advantage of structure)
- We have immense computational resources, but it's never enough; the complexity of the world is always vastly greater
 - we seek *computationally frugal* methods for finding *approximate* solutions (optimality is a distraction; relying on it is untenable)
- We have immense data, but not labeled examples
 - we must be able to learn from *unsupervised* interaction with the world, a.k.a. *self-labelling* (no human labels, not even from the web)

Intelligence is the ability to achieve goals

- “Intelligence is the most powerful phenomena in the universe” —Ray Kurzweil, c 2000
 - The phenomena is that there are systems in the universe that are well thought of as goal-seeking systems
- What is a goal-seeking system?
 - “Constant ends from variable means is the hallmark of mind” —William James, c 1890
 - a system that is better understood in terms of *outcomes* than in terms of *mechanisms*

Intelligent \equiv outcomes > mechanisms

- An information processing system is said to be intelligent, with respect to an observer and a purpose,
 - if the system is well thought of (by the observer, for the purpose) as having goals, i.e.,
 - i.e., if the system's behavior is more usefully predicted, described, or controlled in terms of *outcomes* than in terms of *mechanisms*
- Not a property of the thing itself, but only in relation to an observer and a purpose
- Otherwise, it is perfectly objective

Intelligent \equiv agent works well for many envs

- Intelligence is a property of an agent that can be connected to environments
- An agent is said to be intelligent if it performs well (gets a lot of reward) for a broad range of environments
- Thus, intelligence is
 - a matter of degree
 - an objective property
 - relative to an observer and purpose

Artificial Intelligence (definition)

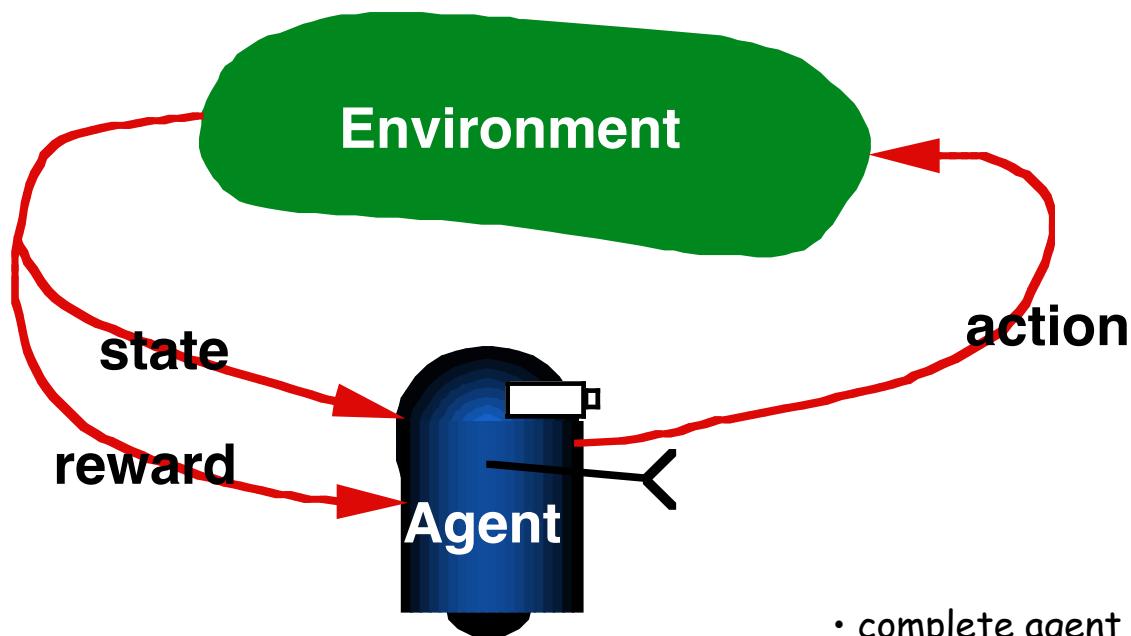
- ❑ the science and technology of information processing systems with goals
 - that is, of information processing systems that observers tend to find it useful to think about in terms of goals
 - that is, in terms of outcomes rather than in terms of mechanisms

Conclusions

- A good way of understanding intelligent systems is as systems that try to find good ways of understanding their input
 - understanding = predicting and controlling
 - good = useful for predicting and controlling
 - controlling = influencing towards a goal
- Intelligence is an appearance
 - Intelligence is in the eye of the beholder, thus relative to her abilities and purpose, not in the thing itself, not binary
 - But then so is everything; this does not make them less real or valuable
- Intelligence is the most powerful appearance in the universe

Examples and Videos of Markov Decision Processes (MDPs) and Reinforcement Learning

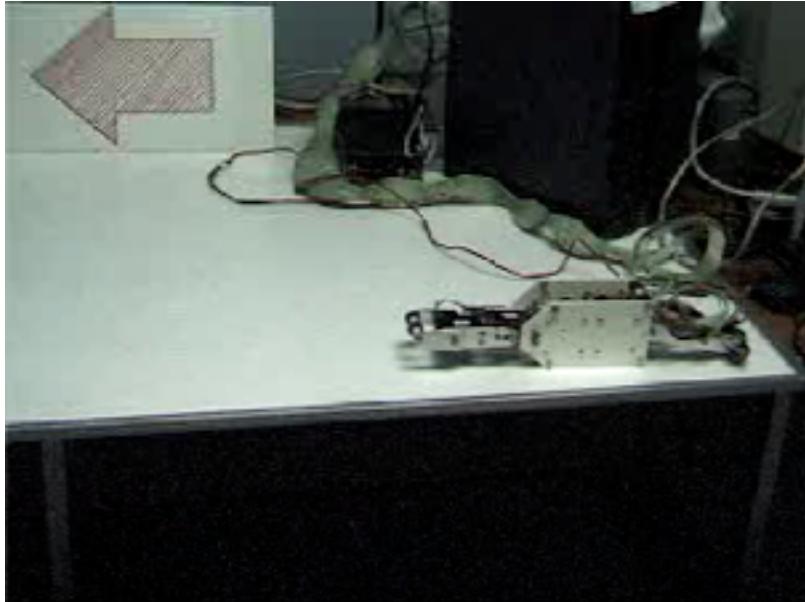
Artificial Intelligence is
interaction to achieve a goal



- complete agent
- temporally situated
- continual learning & planning
- object is to affect environment
- environment stochastic & uncertain

States, Actions, and Rewards

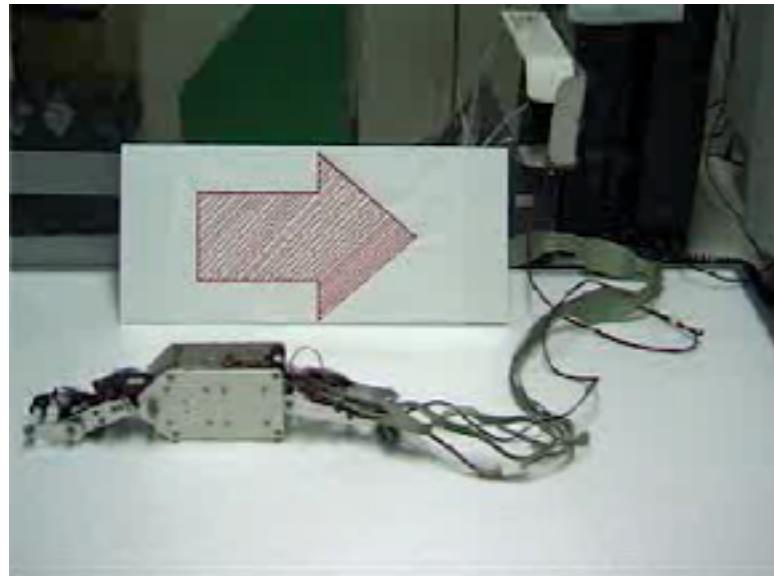
Hajime Kimura's RL Robots



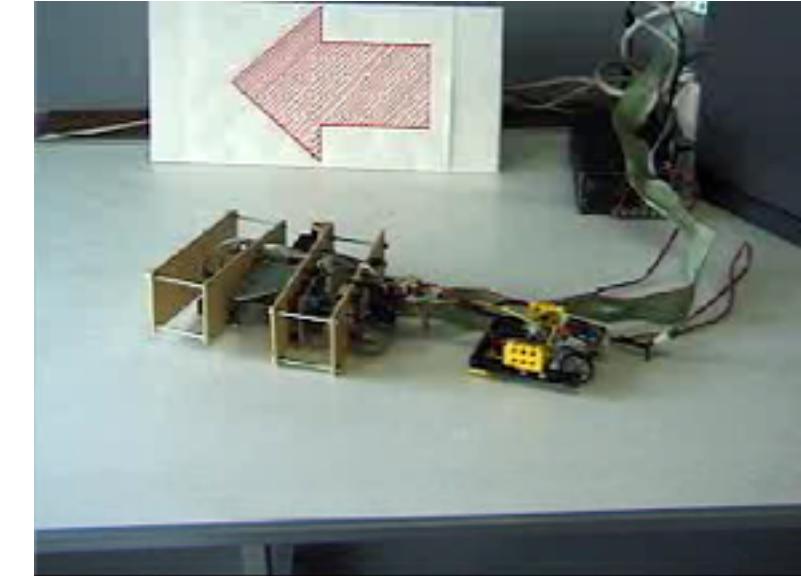
Before



After

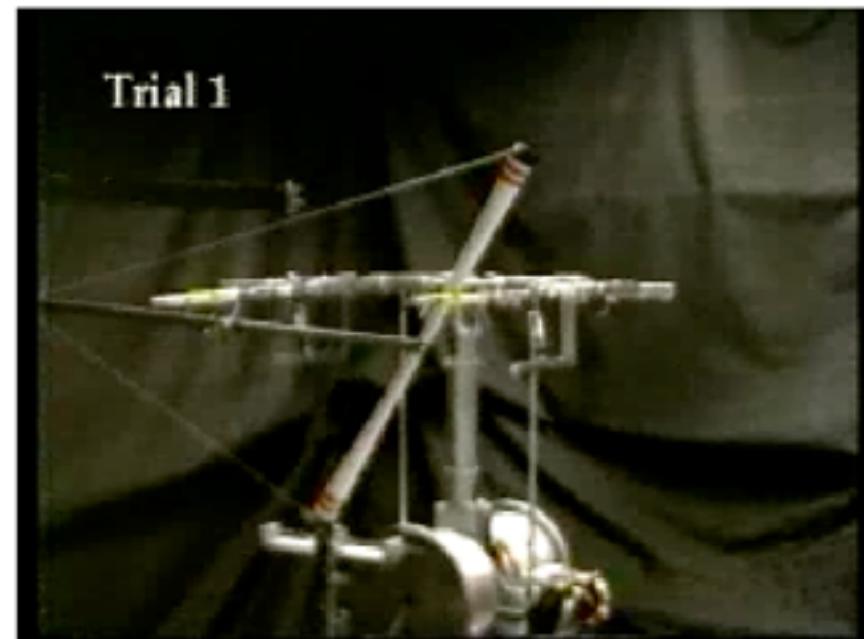
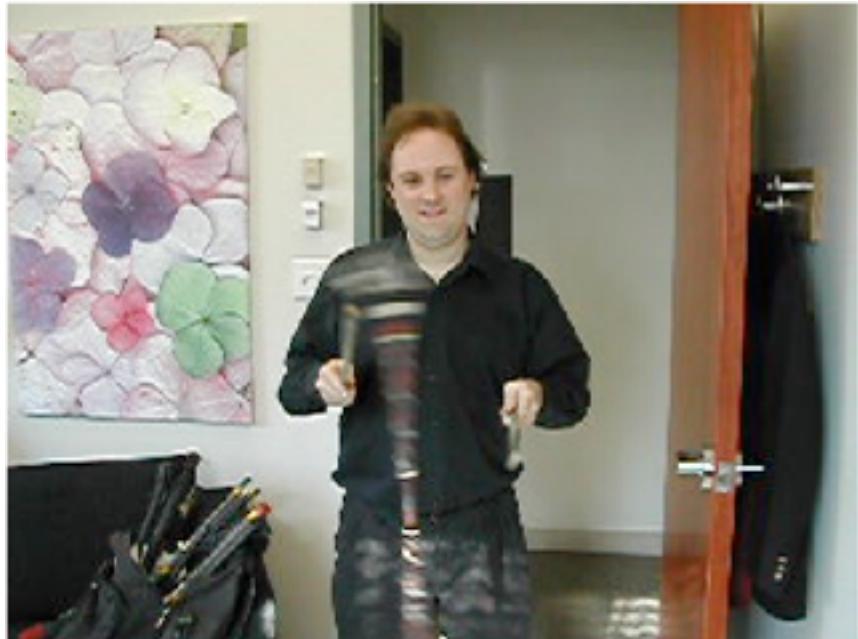


Backward



New Robot, Same algorithm

Devilsticking

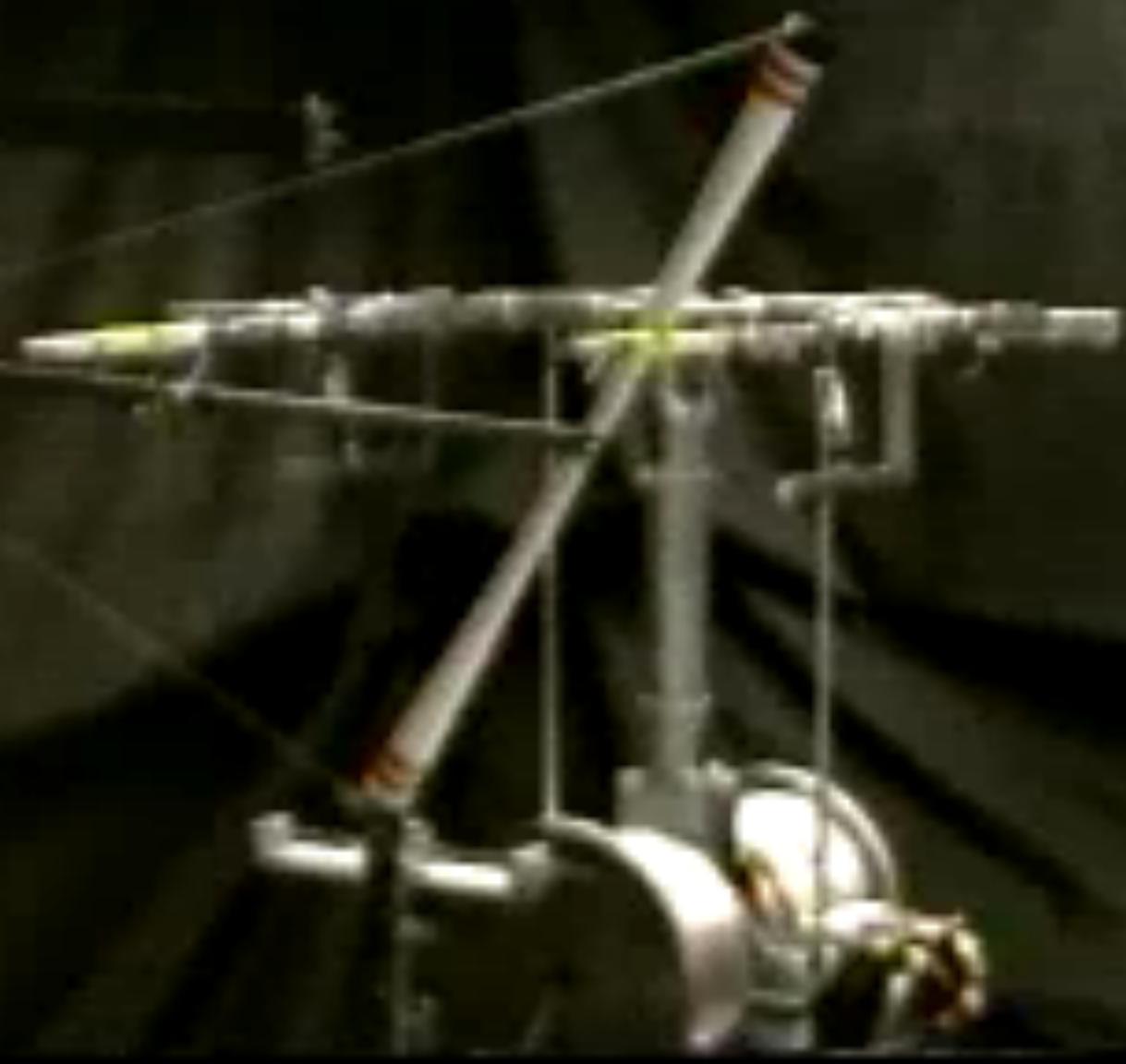


Finnegan Southey
University of Alberta

Stefan Schaal & Chris Atkeson
Univ. of Southern California
“Model-based Reinforcement
Learning of Devilsticking”



Trial 1



The RoboCup Soccer Competition



Autonomous Learning of Efficient Gait

Kohl & Stone (UTexas) 2004









Policies

- A **policy** maps each state to an action to take
 - Like a stimulus–response rule
- We seek a policy that maximizes cumulative reward
- The policy is a subgoal to achieving reward

The Reward Hypothesis

The goal of intelligence is to maximize the cumulative sum of a single received number:
“reward” = pleasure - pain

Artificial Intelligence = reward maximization

Value

Value systems are hedonism with foresight

We value situations according to how much reward we expect will follow them

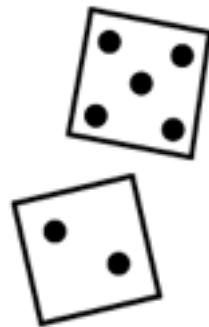
All efficient methods for solving sequential decision problems determine (learn or compute) “value functions” as an intermediate step

Value systems are a *means* to reward, yet we *care more* about values than rewards

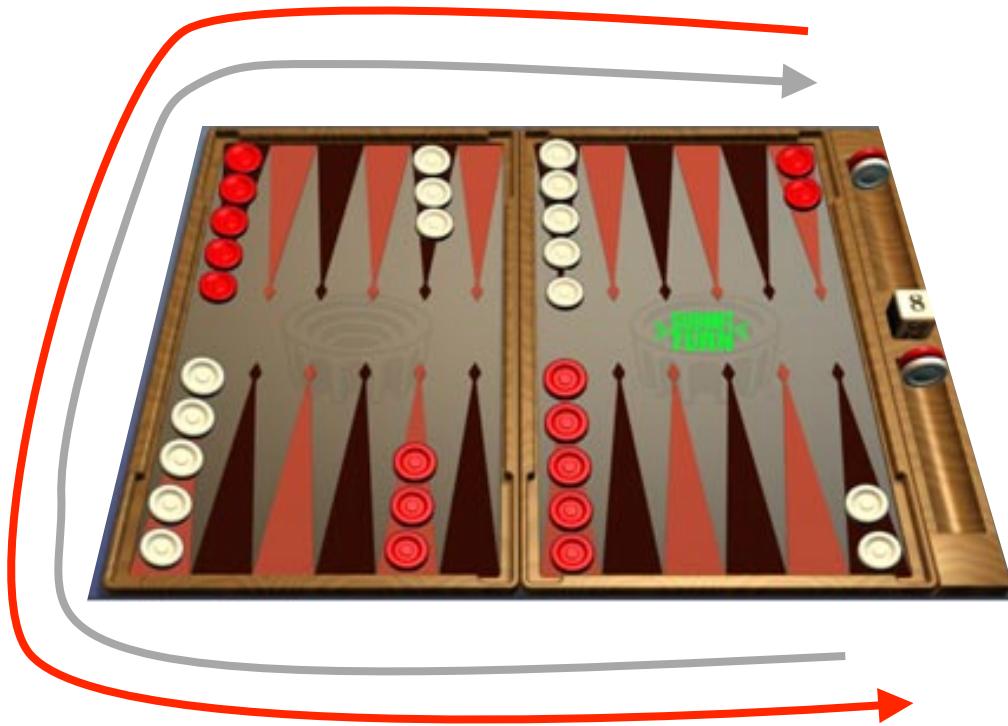
Pleasure = Immediate Reward ≠ good = Long-term Reward

“Even enjoying yourself you call evil whenever it leads to the loss of a pleasure greater than its own, or lays up pains that outweigh its pleasures.... Isn't it the same when we turn back to pain? To suffer pain you call good when it either rids us of greater pains than its own or leads to pleasures that outweigh them.”

—Plato, Protagoras



Backgammon



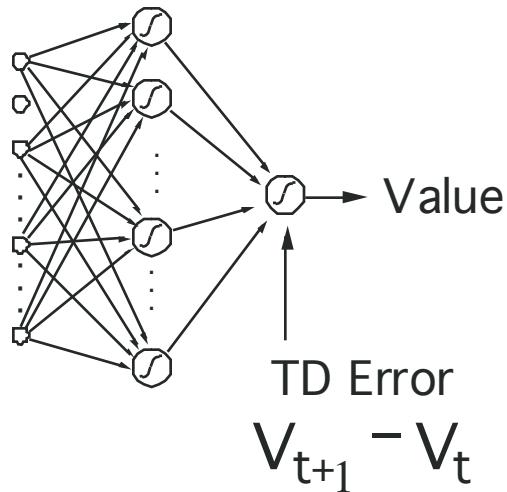
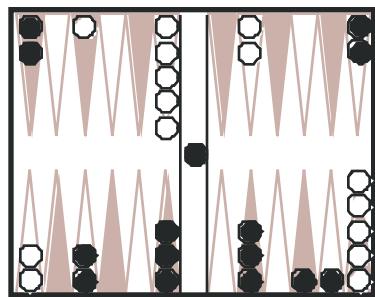
STATES: configurations of the playing board ($\approx 10^{20}$)

ACTIONS: moves

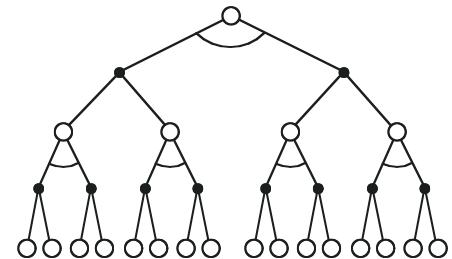
REWARDS: win: +1
lose: -1
else: 0

a “big” game

TD-Gammon



Action selection
by 2-3 ply search



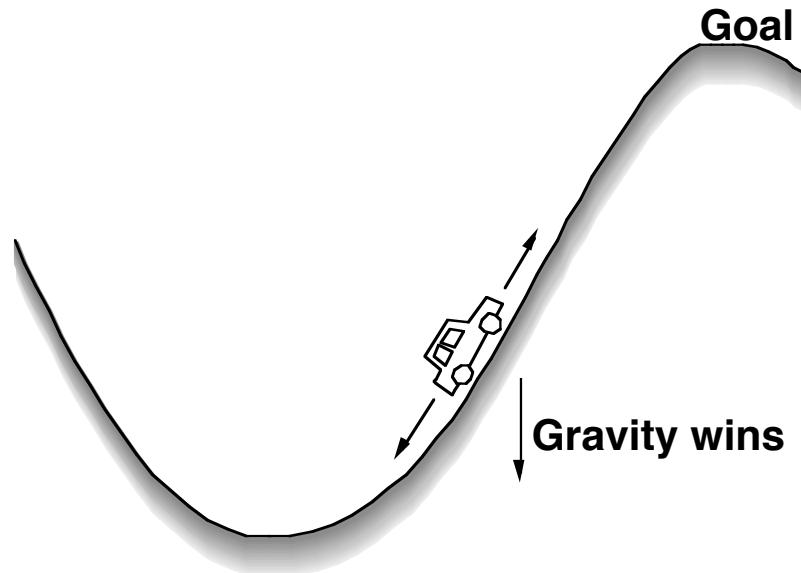
Start with a random Network

Play millions of games against itself

Learn a value function from this simulated experience

Six weeks later it's the best player of backgammon in the world

The Mountain Car Problem



SITUATIONS: car's position and velocity

ACTIONS: three thrusts: forward, reverse, none

REWARDS: always -1 until car reaches the goal

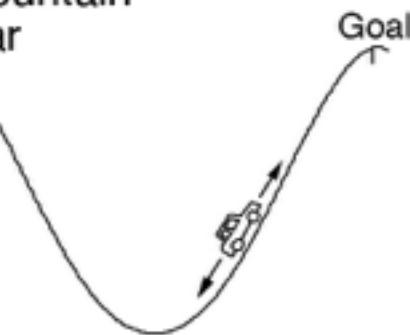
No Discounting

Minimum-Time-to-Goal Problem

Moore, 1990

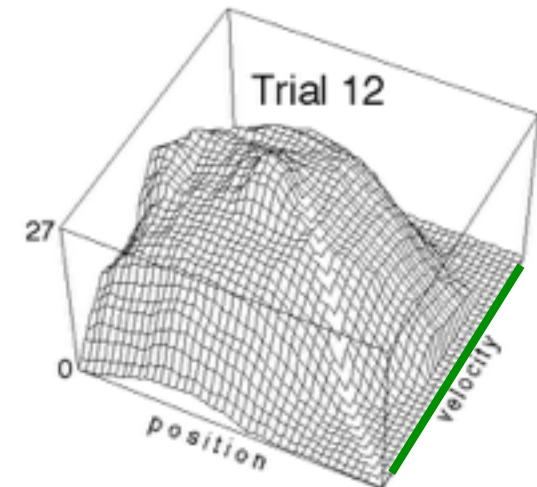
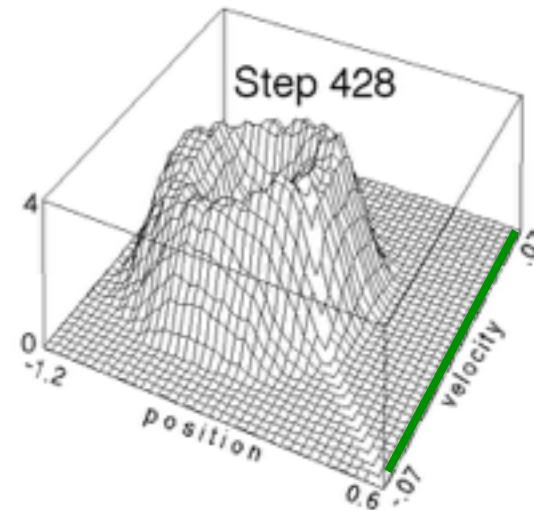
Value Functions Learned while solving the Mountain Car problem

Mountain
Car

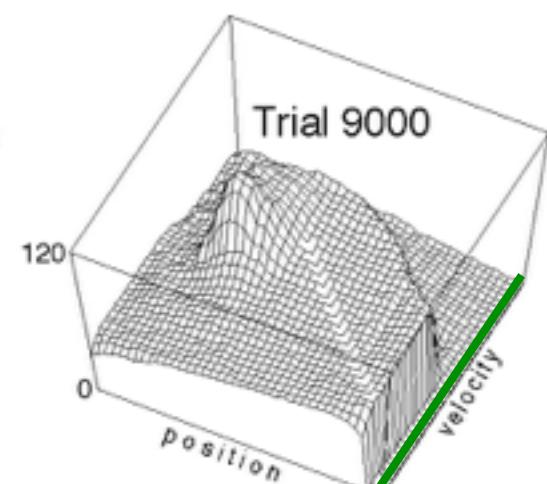
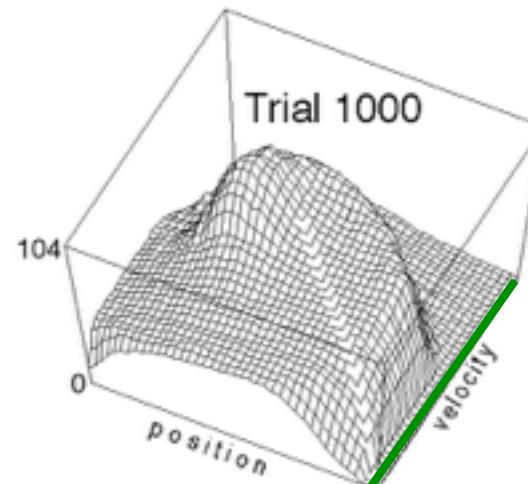
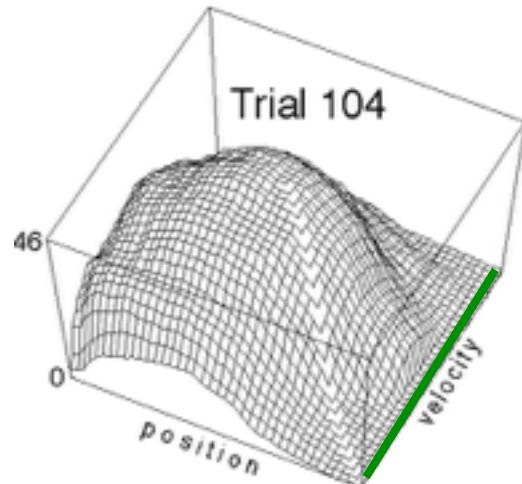


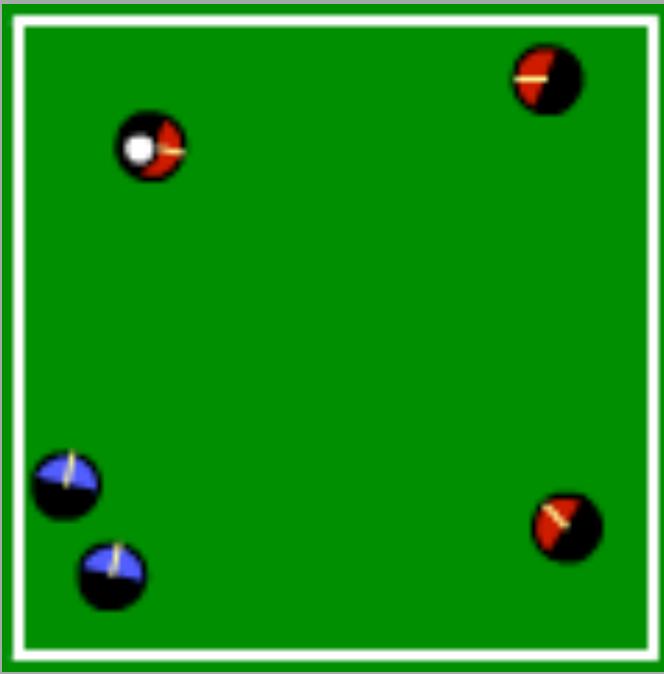
Minimize Time-to-Goal

Value = estimated time to goal

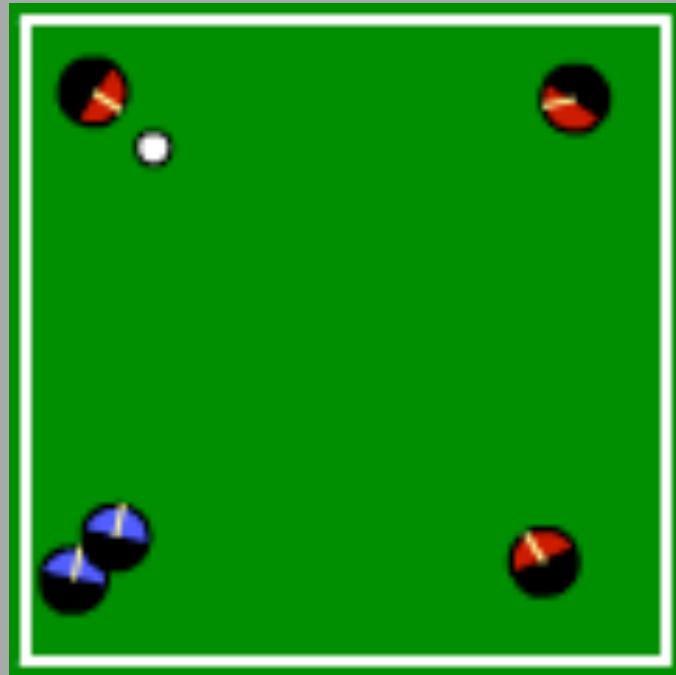


Goal
region

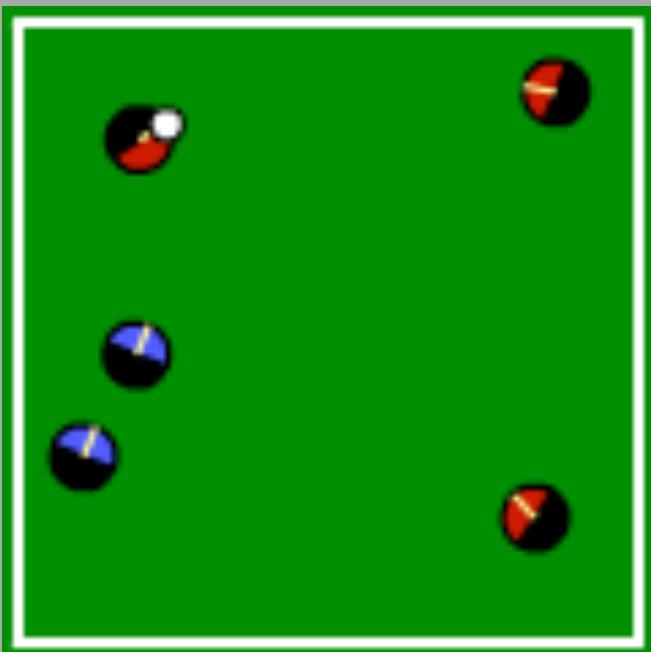




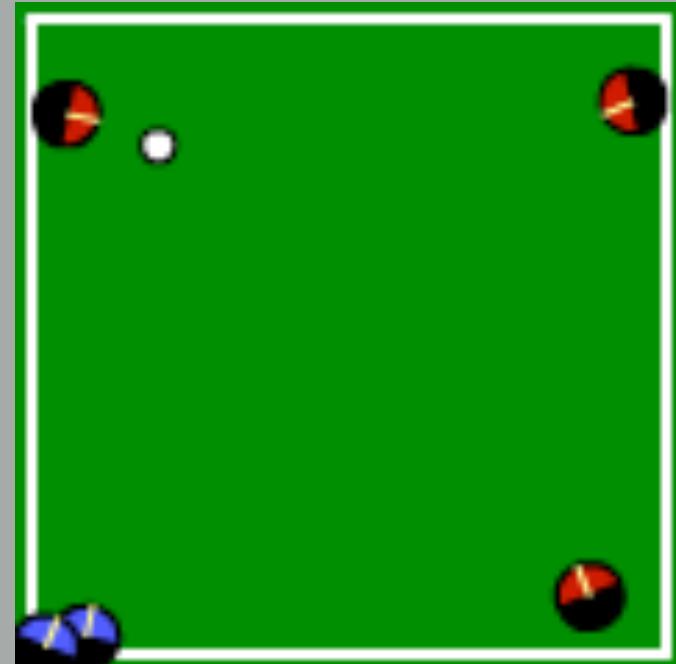
Random



Learned



Hand-coded



Hold



25



15



8



5



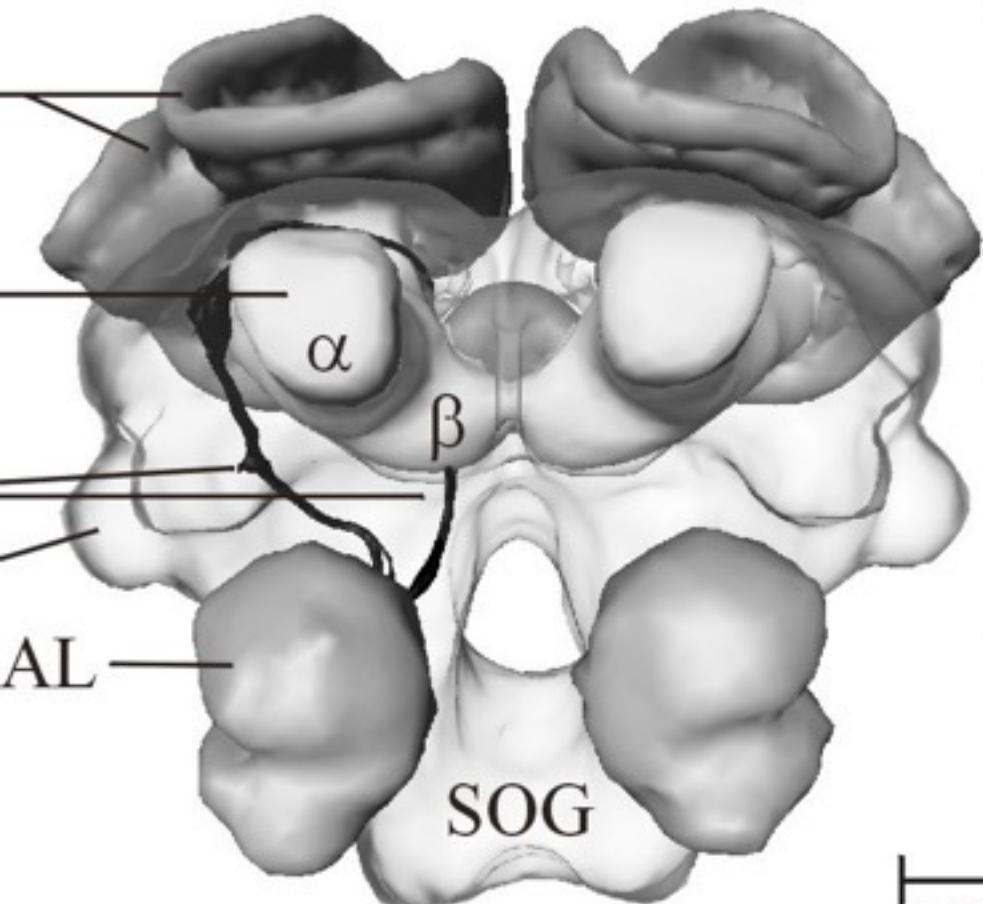
10



Temporal-difference (TD) error

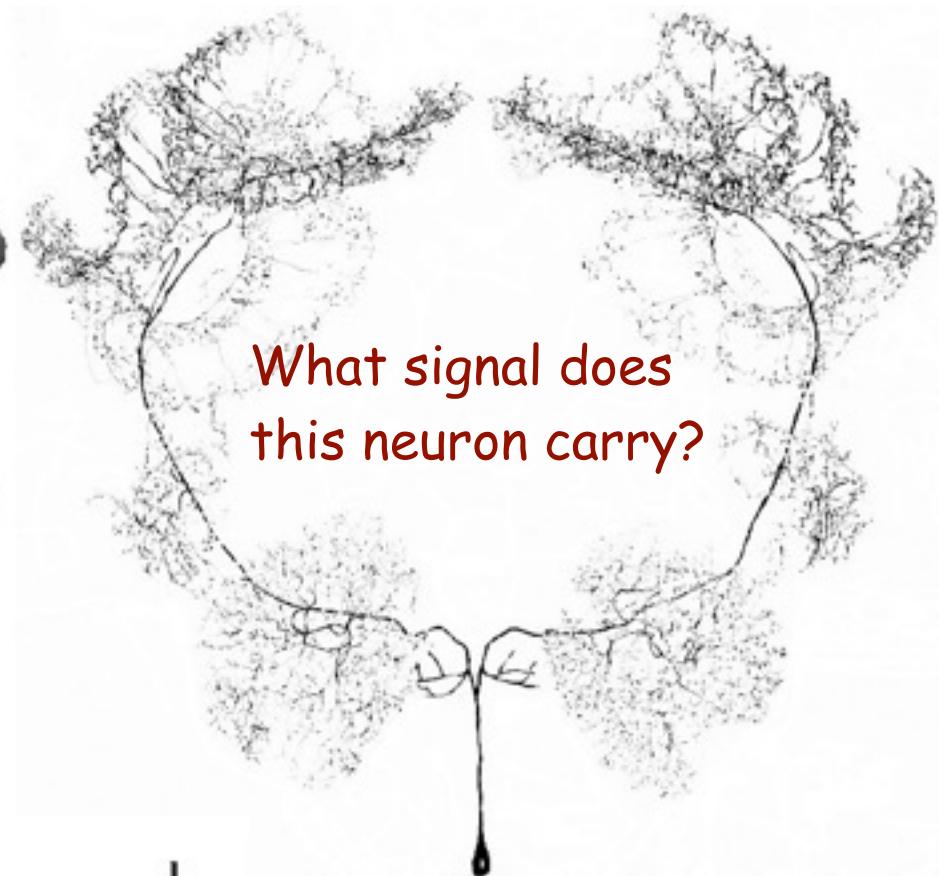
Do things seem to be getting better or worse,
in terms of long-term reward,
at this instant in time?

Brain reward systems



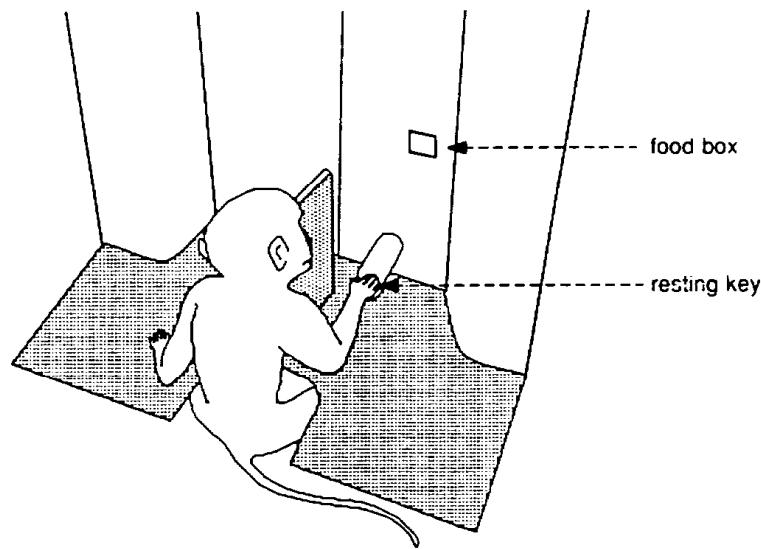
Honeybee Brain

250 μm



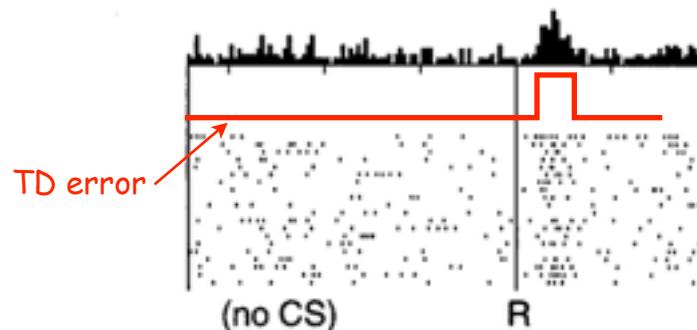
VUM Neuron

Brain reward systems seem to signal TD error

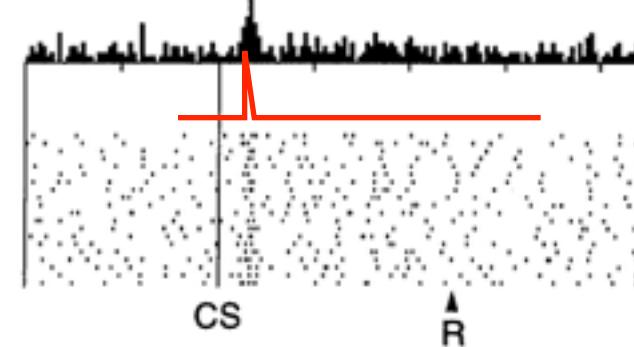


Wolfram Schultz, et al.

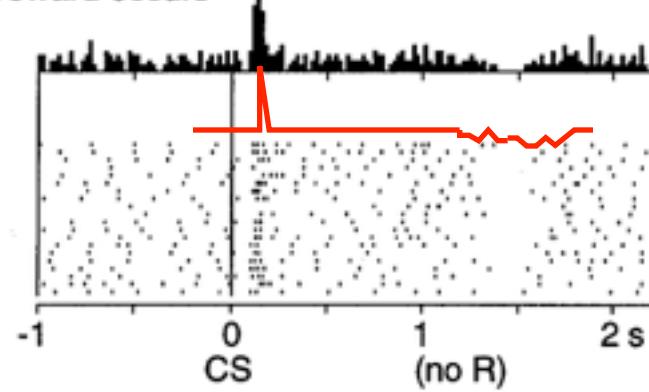
No prediction
Reward occurs



Reward predicted
Reward occurs

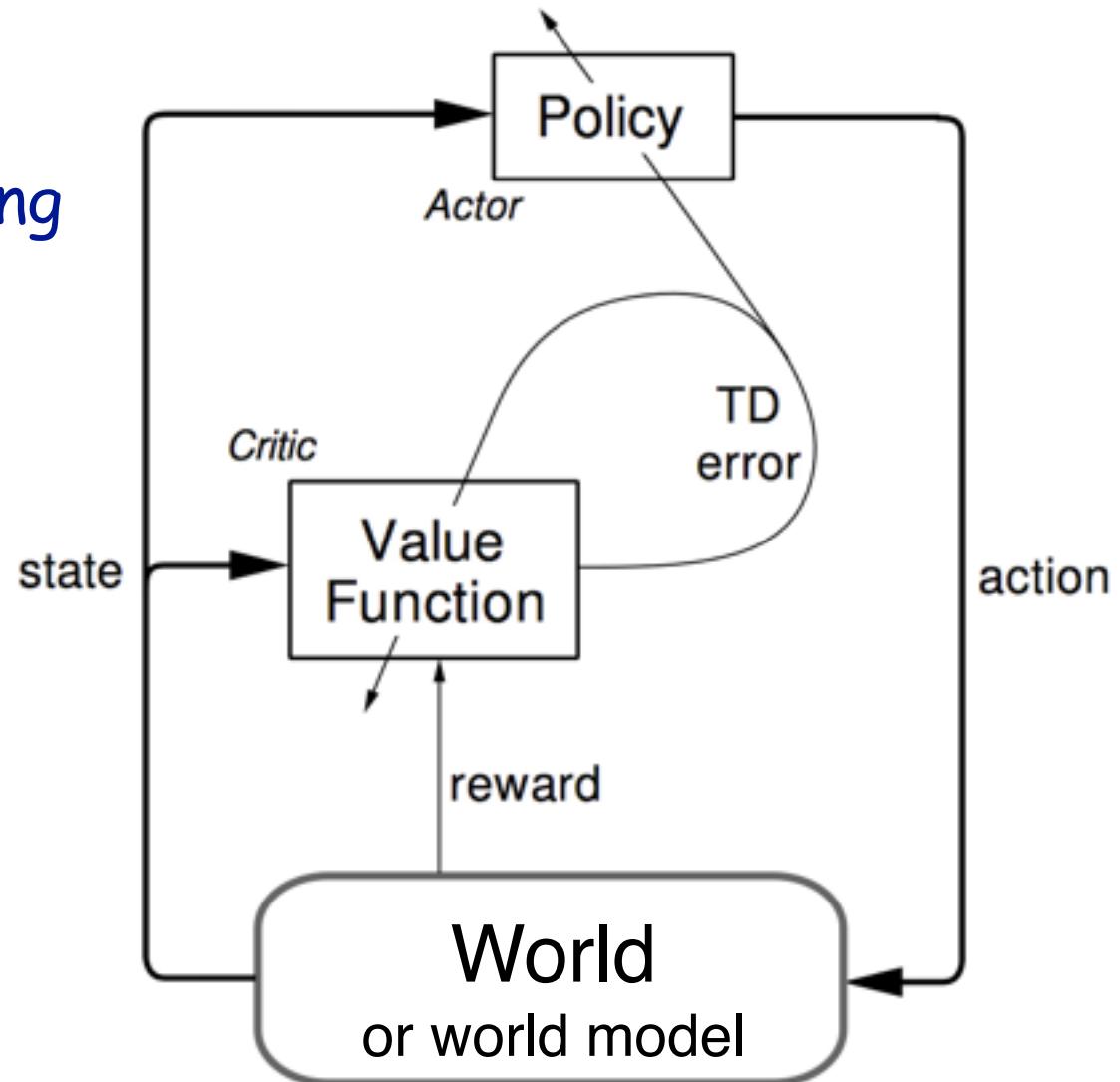


Reward predicted
No reward occurs



World models

the actor-critic reinforcement learning architecture



“Autonomous helicopter flight via Reinforcement Learning”

Ng (Stanford), Kim, Jordan, & Sastry (UC Berkeley) 2004

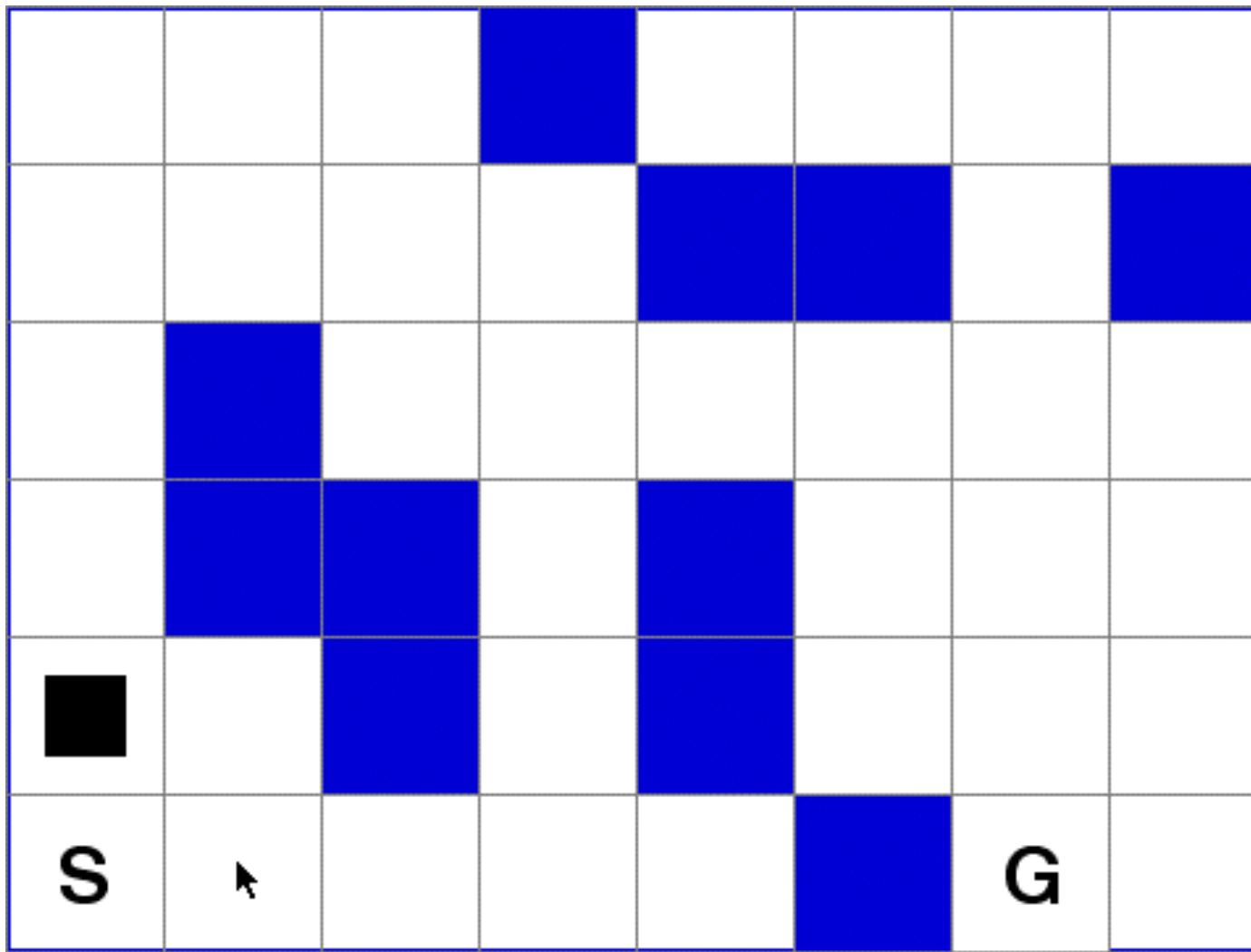




Reason as RL over Imagined Experience

1. Learn a predictive model of the world's dynamics
transition probabilities, expected immediate rewards
2. Use model to generate imaginary experiences
internal thought trials, mental simulation (Craik, 1943)
3. Apply RL as if experience had really happened
vicarious trial and error (Tolman, 1932)

GridWorld Example



Go

Step

Policy

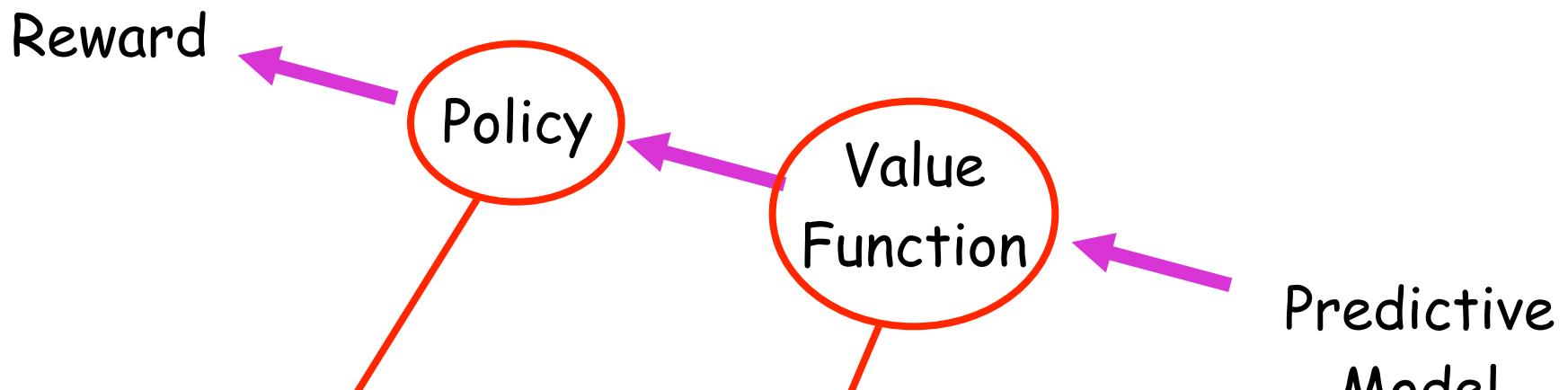
Values

Faster

Slower

0|1|0

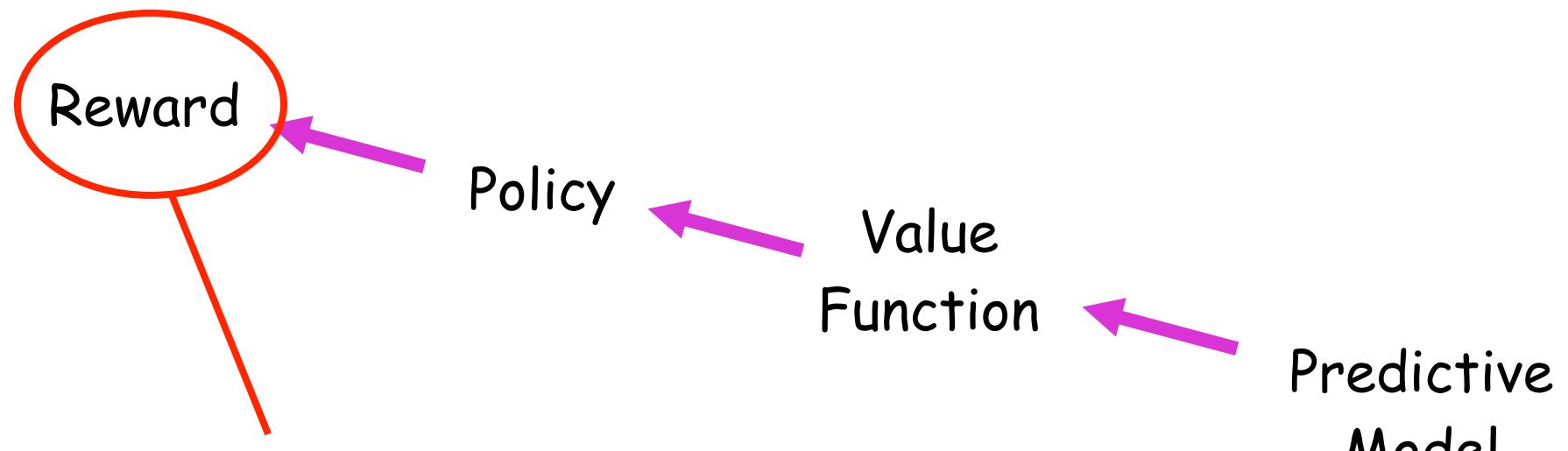
Summary: RL's Computational Theory of Mind



A learned, time-varying prediction of imminent reward
Key to all efficient methods for finding optimal policies

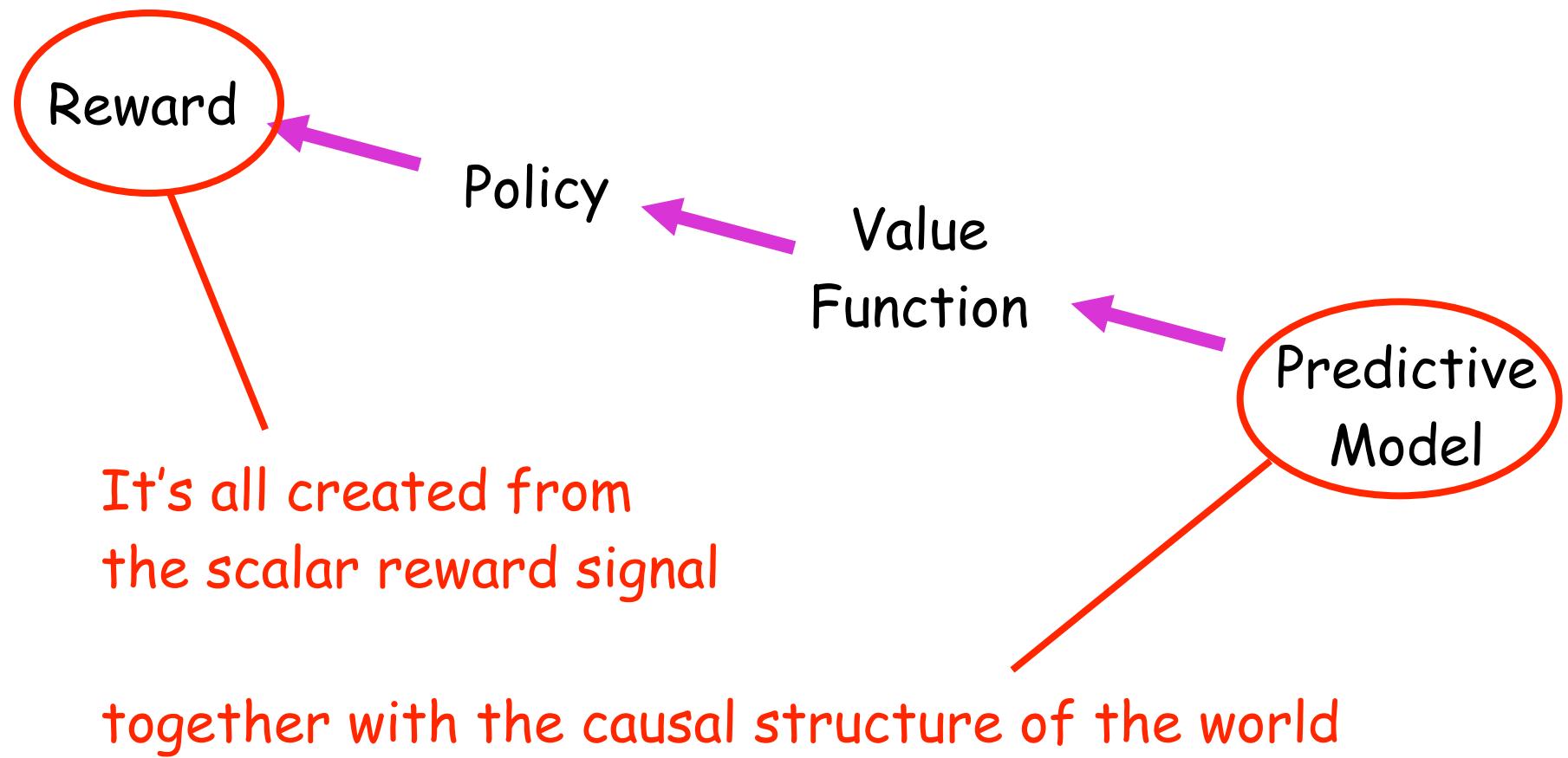
This has nothing to do with either biology or computers

Summary: RL's Computational Theory of Mind



It's all created from
the scalar reward signal

Summary: RL's Computational Theory of Mind



Finite Markov Decision Processes (MDPs)

$$MDP = \langle \underbrace{S, A, R_0, P}_{\text{finite sets}}, \gamma, s_0 \rangle$$

dynamics

$$P: S \times R_0 \times S \times A \rightarrow [0,1] \quad \sum_{s' \in S} \sum_{r \in R_0} P(s', r | s, a) = 1$$

$$= P(s', r | s, a) = \Pr[S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a] \quad \begin{matrix} s \in S \\ a \in A \end{matrix}$$

$t = 0, 1, 2, \dots$

Trajectory

$$S_0, A_0(R_0, S_1), A_1(R_1, S_2), A_2, R_2, S_3, A_3, \dots$$

Policy

stochastic: $\pi: A \times S \rightarrow [0,1] \quad \pi(a | s) = \Pr[A_t = a | S_t = s]$

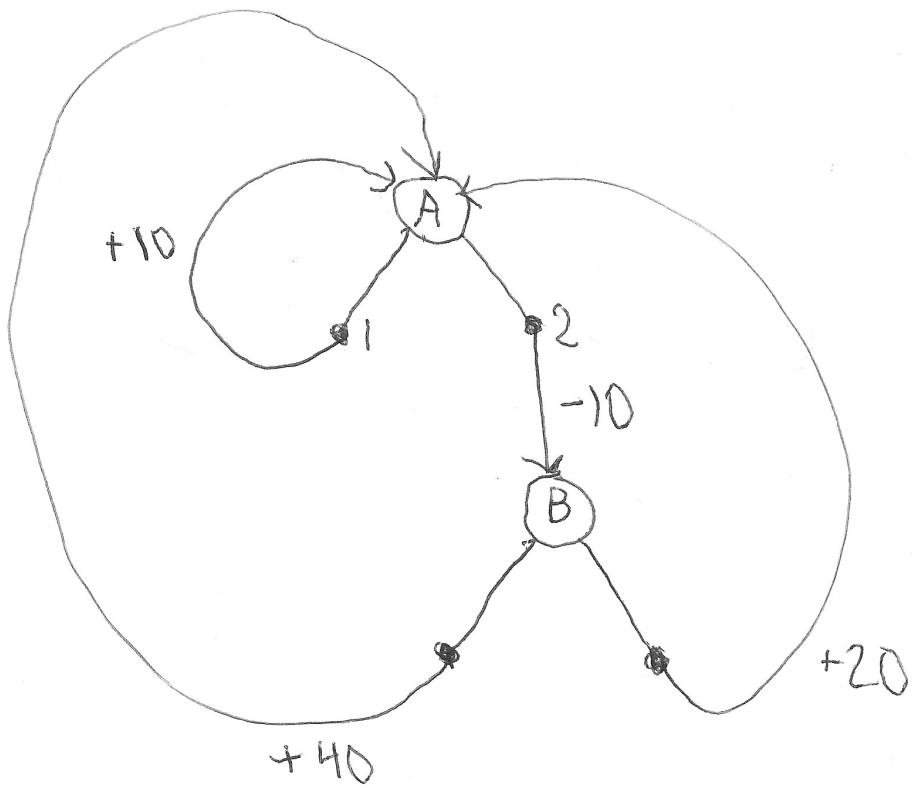
deterministic: $\pi: S \rightarrow A \quad A_t = \pi(S_t)$

Optimal policy

Average Reward per step Setting

$$\pi^* = \arg \max_{\pi} \lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=1}^T E[R_{t+1} | S_t = s]$$

Markov Decision Process



1 - left
2 - right

π

$\pi: A: 2 \text{ (right)}$
 $\pi: B: 1 \text{ (left)}$

B 1 40 A 2 -10 B 1 40 A ...

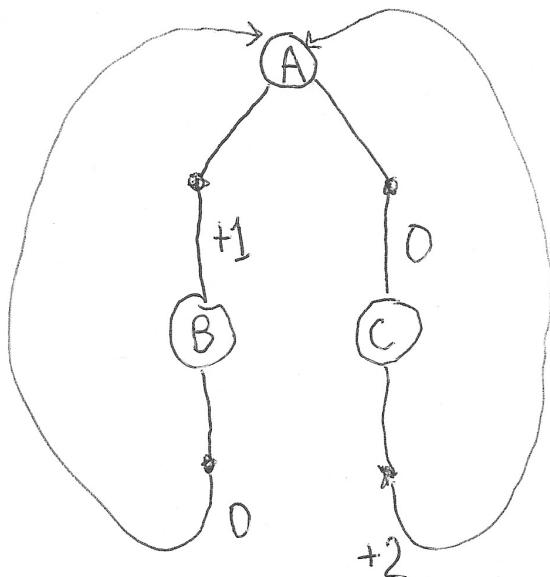
Discounting

$\gamma \in [0,1] \quad 0 \leq \gamma < 1$ discount rate

Return

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots$$

0 1 0 1 0 1
 0 2 0 2 0 2



What policy is optimal?

A: left

B: Right C: Other

If $\gamma=0$?

If $\gamma=.99$

If $\gamma=\frac{1}{2}$?

Value function for a given policy

$$v_\pi(s) = E_\pi[G_t | S_t = s]$$

$v_\pi: \mathcal{S} \rightarrow \mathbb{R}$ Reals

Optimal policy

$$\pi_* = \arg \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S}$$

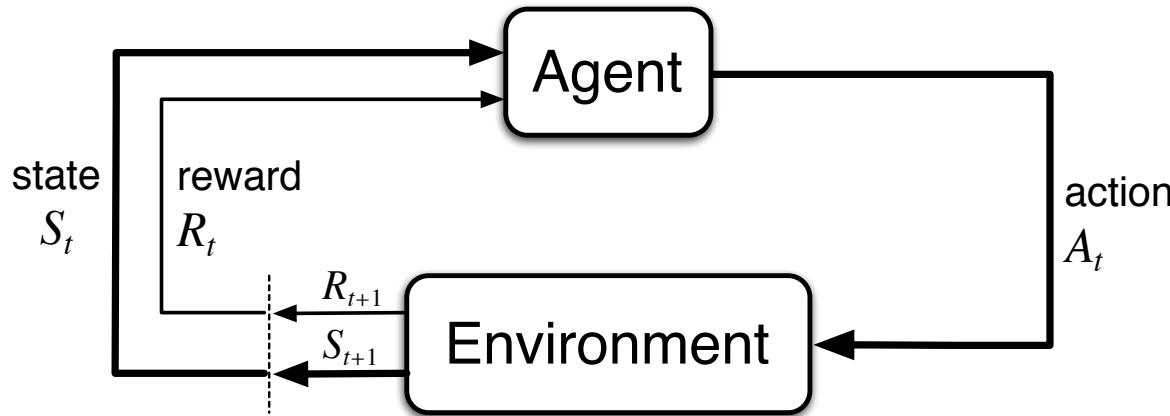
Chapter 3: The Reinforcement Learning Problem

(Markov Decision Processes, or MDPs)

Objectives of this chapter:

- present Markov decision processes—an idealized form of the AI problem for which we have precise theoretical results
- introduce key components of the mathematics: value functions and Bellman equations

The Agent-Environment Interface



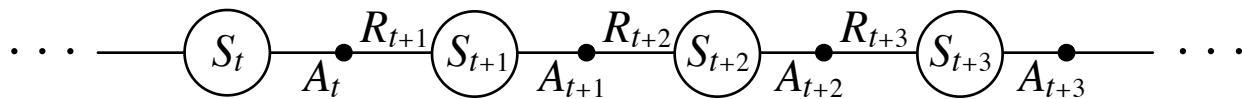
Agent and environment interact at discrete time steps: $t = 0, 1, 2, 3, \dots$

Agent observes state at step t : $S_t \in \mathcal{S}$

produces action at step t : $A_t \in \mathcal{A}(S_t)$

gets resulting reward: $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$

and resulting next state: $S_{t+1} \in \mathcal{S}^+$



Markov Decision Processes

- If a reinforcement learning task has the Markov Property, it is basically a **Markov Decision Process (MDP)**.
- If state and action sets are finite, it is a **finite MDP**.
- To define a finite MDP, you need to give:
 - **state and action sets**
 - one-step “dynamics”

$$p(s', r | s, a) = \Pr\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\}$$

The Agent Learns a Policy

Policy at step t = π_t =

a mapping from states to action probabilities

$\pi_t(a \mid s)$ = probability that $A_t = a$ when $S_t = s$

Special case - *deterministic policies*:

$\pi_t(s)$ = the action taken with prob=1 when $S_t = s$

- Reinforcement learning methods specify how the agent changes its policy as a result of experience.
- Roughly, the agent's goal is to get as much reward as it can over the long run.

The Markov Property

- By “the state” at step t , the book means whatever information is available to the agent at step t about its environment.
- The state can include immediate “sensations,” highly processed sensations, and structures built up over time from sequences of sensations.
- Ideally, a state should summarize past sensations so as to retain all “essential” information, i.e., it should have the **Markov Property**:

$$\Pr\{R_{t+1} = r, S_{t+1} = s' \mid S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\} = \\ p(s', r | s, a) = \Pr\{R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t\}$$

- for all $s' \in \mathcal{S}^+, r \in \mathcal{R}$, and all histories $S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$.

The Meaning of Life (goals, rewards, and returns)

Rewards and returns

- The objective in RL is to maximize long-term future reward
- That is, to choose A_t so as to maximize $R_{t+1}, R_{t+2}, R_{t+3}, \dots$
- But what exactly should be maximized?
- The discounted return at time t :

the *discount rate*

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \quad \gamma \in [0, 1)$$

γ	Reward sequence	Return
0.5(or any)	1 0 0 0...	
0.5	0 0 2 0 0 0...	
0.9	0 0 2 0 0 0...	
0.5	-1 2 6 3 2 0 0 0...	

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \qquad \gamma \in [0,1)$$

Values are *expected* returns

- The value of a state, given a policy:

$$v_\pi(s) = \mathbb{E}\{G_t \mid S_t = s, A_{t:\infty} \sim \pi\} \quad v_\pi : \mathcal{S} \rightarrow \mathbb{R}$$

- The value of a state-action pair, given a policy:

$$q_\pi(s, a) = \mathbb{E}\{G_t \mid S_t = s, A_t = a, A_{t+1:\infty} \sim \pi\} \quad q_\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$$

- The optimal value of a state:

$$v_*(s) = \max_\pi v_\pi(s) \quad v_* : \mathcal{S} \rightarrow \mathbb{R}$$

- The optimal value of a state-action pair:

$$q_*(s, a) = \max_\pi q_\pi(s, a) \quad q_* : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$$

- Optimal policy: π_* is an optimal policy if and only if

$$\pi_*(a|s) > 0 \text{ only where } q_*(s, a) = \max_b q_*(s, b) \quad \forall s \in \mathcal{S}$$

- in other words, π_* is optimal iff it is *greedy* wrt q_*

4 value functions

	state values	action values
prediction	v_π	q_π
control	v_*	q_*

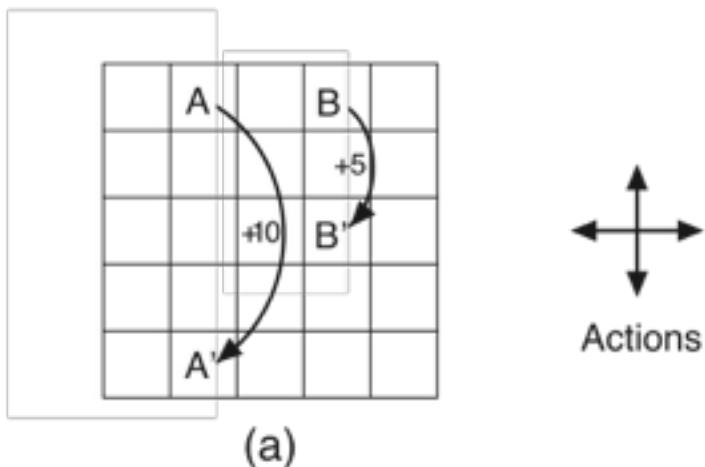
- All theoretical objects, mathematical ideals (expected values)
- Distinct from their estimates:

$$V_t(s) \quad Q_t(s, a)$$

optimal policy example

Gridworld

- Actions: north, south, east, west; deterministic.
- If would take agent off the grid: no move but reward = -1
- Other actions produce reward = 0, except actions that move agent out of special states A and B as shown.



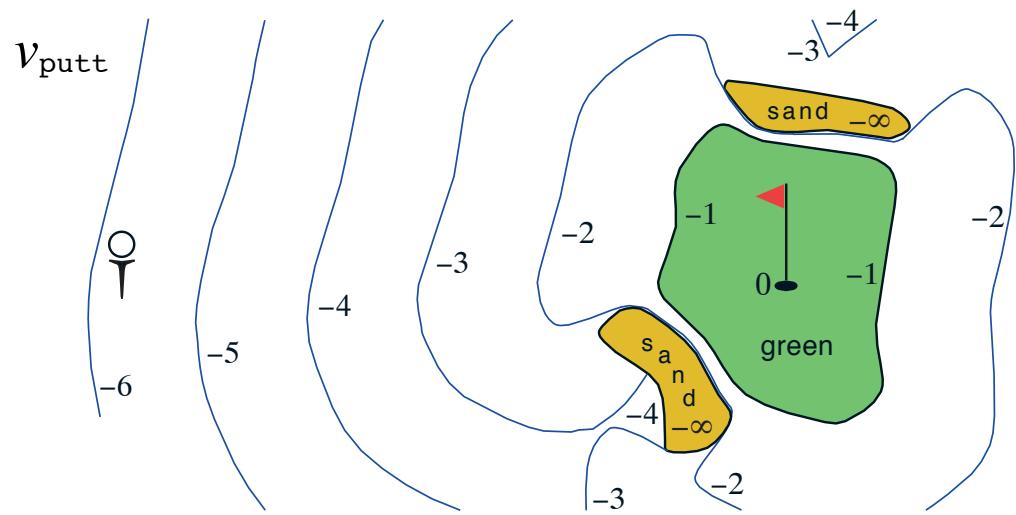
3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

(b)

State-value function
for equiprobable
random policy;
 $\gamma = 0.9$

Golf

- State is ball location
- Reward of -1 for each stroke until the ball is in the hole
- Value of a state?
- Actions:
 - **putt** (use putter)
 - **driver** (use driver)
- **putt** succeeds anywhere on the green



Optimal Value Functions

- For finite MDPs, policies can be **partially ordered**:
$$\pi \geq \pi' \quad \text{if and only if } v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in \mathcal{S}$$
- There are always one or more policies that are better than or equal to all the others. These are the **optimal policies**. We denote them all π_* .
- Optimal policies share the same **optimal state-value function**:

$$v_*(s) = \max_{\pi} v_{\pi}(s) \quad \text{for all } s \in \mathcal{S}$$

- Optimal policies also share the same **optimal action-value function**:

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a) \quad \text{for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}$$

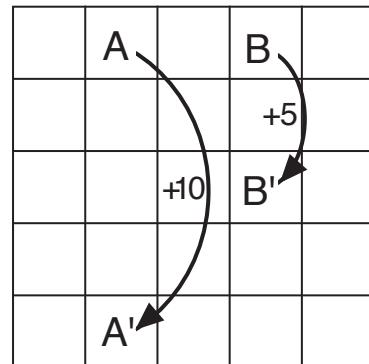
This is the expected return for taking action a in state s and thereafter following an optimal policy.

Why Optimal State-Value Functions are Useful

Any policy that is greedy with respect to v_* is an optimal policy.

Therefore, given v_* , one-step-ahead search produces the long-term optimal actions.

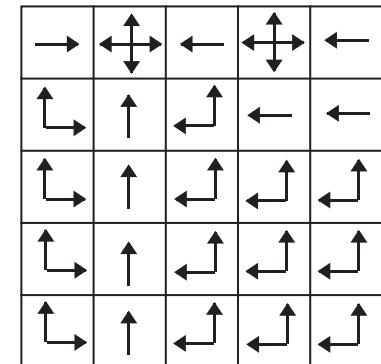
E.g., back to the gridworld:



a) gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

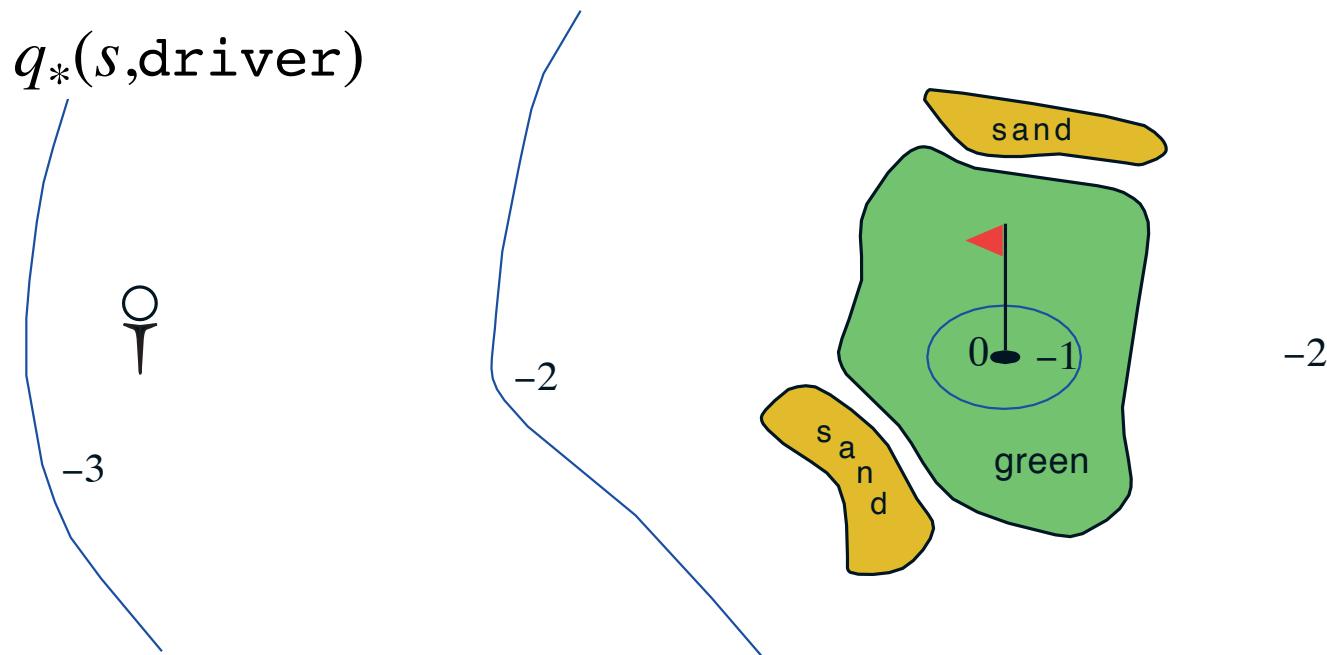
b) v_*



c) π_*

Optimal Value Function for Golf

- We can hit the ball farther with `driver` than with `putter`, but with less accuracy
- $q_*(s, \text{driver})$ gives the value of using `driver` first, then using whichever actions are best



What About Optimal Action-Value Functions?

Given q_* , the agent does not even have to do a one-step-ahead search:

$$\pi_*(s) = \arg \max_a q_*(s, a)$$

Value Functions

x 4

Bellman Equations

x 4

Bellman Equation for a Policy π

The basic idea:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\ &= R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

So:

$$\begin{aligned} v_\pi(s) &= E_\pi \left\{ G_t \mid S_t = s \right\} \\ &= E_\pi \left\{ R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s \right\} \end{aligned}$$

Or, without the expectation operator:

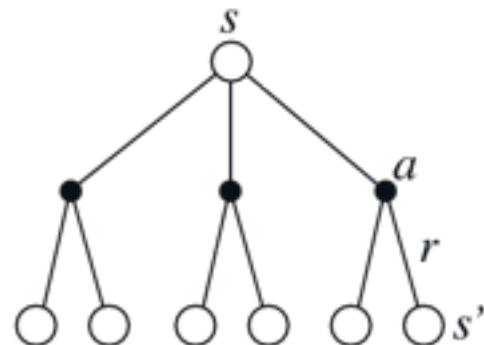
$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a) [r + \gamma v_\pi(s')]$$

More on the Bellman Equation

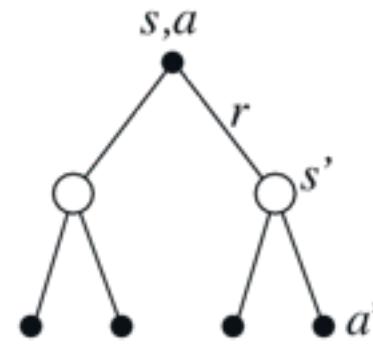
$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_\pi(s')]$$

This is a set of equations (in fact, linear), one for each state.
The value function for π is its unique solution.

Backup diagrams:



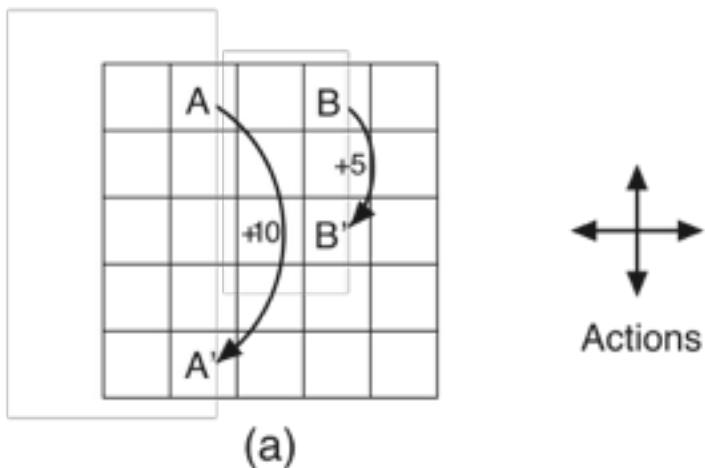
for v_π



for q_π

Gridworld

- Actions: north, south, east, west; deterministic.
- If would take agent off the grid: no move but reward = -1
- Other actions produce reward = 0, except actions that move agent out of special states A and B as shown.



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

(b)

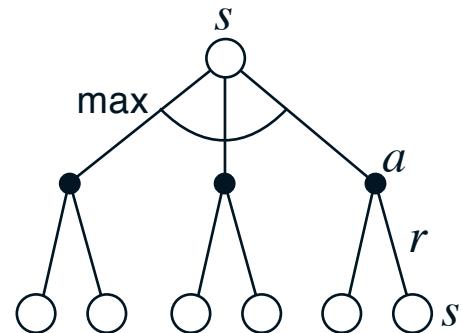
State-value function
for equiprobable
random policy;
 $\gamma = 0.9$

Bellman Optimality Equation for v_*

The value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} v_*(s) &= \max_a q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_*(s')]. \end{aligned}$$

The relevant backup diagram:

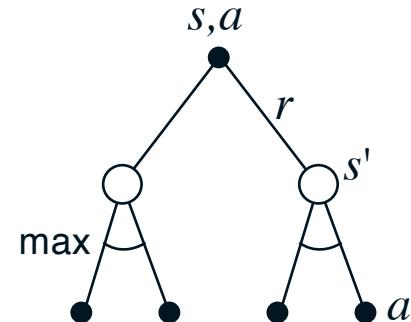


v_* is the unique solution of this system of nonlinear equations.

Bellman Optimality Equation for q_*

$$\begin{aligned} q_*(s, a) &= \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a \right] \\ &= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \max_{a'} q_*(s', a') \right]. \end{aligned}$$

The relevant backup diagram:



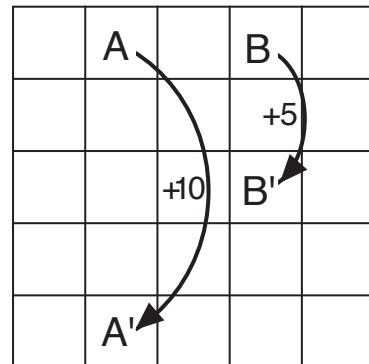
q_* is the unique solution of this system of nonlinear equations.

Why Optimal State-Value Functions are Useful

Any policy that is greedy with respect to v_* is an optimal policy.

Therefore, given v_* , one-step-ahead search produces the long-term optimal actions.

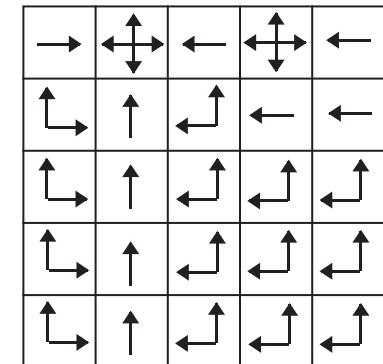
E.g., back to the gridworld:



a) gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

b) v_*



c) π_*

Solving the Bellman Optimality Equation

- ❑ Finding an optimal policy by solving the Bellman Optimality Equation requires the following:
 - accurate knowledge of environment dynamics;
 - we have enough space and time to do the computation;
 - the Markov Property.
- ❑ How much space and time do we need?
 - polynomial in number of states (via dynamic programming methods; Chapter 4),
 - BUT, number of states is often huge (e.g., backgammon has about 10^{20} states).
- ❑ We usually have to settle for approximations.
- ❑ Many RL methods can be understood as approximately solving the Bellman Optimality Equation.

Summary

- Agent-environment interaction
 - States
 - Actions
 - Rewards
- Policy: stochastic rule for selecting actions
- Return: the function of future rewards agent tries to maximize
- Episodic and continuing tasks
- Markov Property
- Markov Decision Process
 - Transition probabilities
 - Expected rewards
- Value functions
 - State-value function for a policy
 - Action-value function for a policy
 - Optimal state-value function
 - Optimal action-value function
- Optimal value functions
- Optimal policies
- Bellman Equations
- The need for approximation

Chapter 4: Dynamic Programming

Objectives of this chapter:

- Overview of a collection of classical solution methods for MDPs known as dynamic programming (DP)
- Show how DP can be used to compute value functions, and hence, optimal policies
- Discuss efficiency and utility of DP

Policy Evaluation

Policy Evaluation: for a given policy π , compute the state-value function v_π

Recall: **State-value function for policy π**

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right]$$

Recall: **Bellman equation for v_π**

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_\pi(s')]$$

—a system of $|S|$ simultaneous equations

Iterative Methods

$$v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k \rightarrow v_{k+1} \rightarrow \cdots \rightarrow v_\pi$$

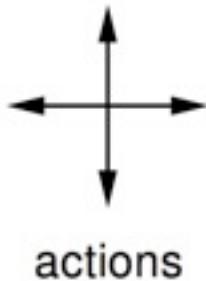
a “sweep” 

A sweep consists of applying a **backup operation** to each state.

A **full policy-evaluation backup**:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')] \quad \forall s \in \mathcal{S}$$

A Small Gridworld



	1	2	3
4	5	6	7
8	9	10	11
	12	13	14

$R = -1$
on all transitions

$\gamma = 1$

- An undiscounted episodic task
- Nonterminal states: 1, 2, . . . , 14;
- One terminal state (shown twice as shaded squares)
- Actions that would take agent off the grid leave state unchanged
- Reward is -1 until the terminal state is reached

Iterative Policy Eval for the Small Gridworld

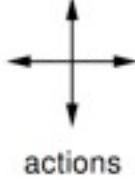
V_k for the
Random Policy

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

π = equiprobable random action choices

$k = 1$



	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$R = -1$
on all transitions

$k = 2$

$\gamma = 1$

$k = 3$

- An undiscounted episodic task
- Nonterminal states: 1, 2, . . . , 14;
- One terminal state (shown twice as shaded squares)
- Actions that would take agent off the grid leave state unchanged
- Reward is -1 until the terminal state is reached

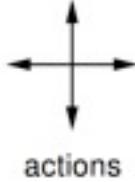
$k = 10$

$k = \infty$

Iterative Policy Eval for the Small Gridworld

V_k for the
Random Policy

π = equiprobable random action choices



	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$R = -1$
on all transitions

$\gamma = 1$

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

$k = 3$

- An undiscounted episodic task
- Nonterminal states: 1, 2, . . . , 14;
- One terminal state (shown twice as shaded squares)
- Actions that would take agent off the grid leave state unchanged
- Reward is -1 until the terminal state is reached

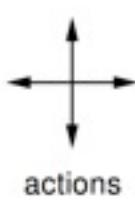
$k = 10$

$k = \infty$

Iterative Policy Eval for the Small Gridworld

V_k for the
Random Policy

π = equiprobable random action choices



	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$R = -1$
on all transitions

$\gamma = 1$

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

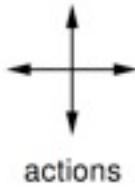
- An undiscounted episodic task
- Nonterminal states: 1, 2, . . . , 14;
- One terminal state (shown twice as shaded squares)
- Actions that would take agent off the grid leave state unchanged
- Reward is -1 until the terminal state is reached

$k = 10$

$k = \infty$

Iterative Policy Eval for the Small Gridworld

π = equiprobable random action choices



	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$R = -1$
on all transitions

$$\gamma = 1$$

- An undiscounted episodic task
- Nonterminal states: 1, 2, . . . , 14;
- One terminal state (shown twice as shaded squares)
- Actions that would take agent off the grid leave state unchanged
- Reward is -1 until the terminal state is reached

$$k = \infty$$

V_k for the
Random Policy

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$$k = 0$$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$$k = 1$$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$$k = 2$$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$$k = 3$$

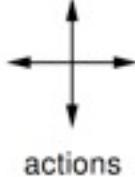
$$k = 10$$

$$k = \infty$$

Iterative Policy Eval for the Small Gridworld

V_k for the
Random Policy

π = equiprobable random action choices



	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$R = -1$
on all transitions

$\gamma = 1$

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Iterative Policy Evaluation – One array version

Input π , the policy to be evaluated

Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$

Repeat

$$\Delta \leftarrow 0$$

For each $s \in \mathcal{S}$:

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$ (a small positive number)

Output $V \approx v_\pi$

Value Iteration

Recall the **full policy-evaluation backup**:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')] \quad \forall s \in \mathcal{S}$$

Here is the **full value-iteration backup**:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')] \quad \forall s \in \mathcal{S}$$

Value Iteration – One array version

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat

$$\Delta \leftarrow 0$$

For each $s \in \mathcal{S}$:

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$ (a small positive number)

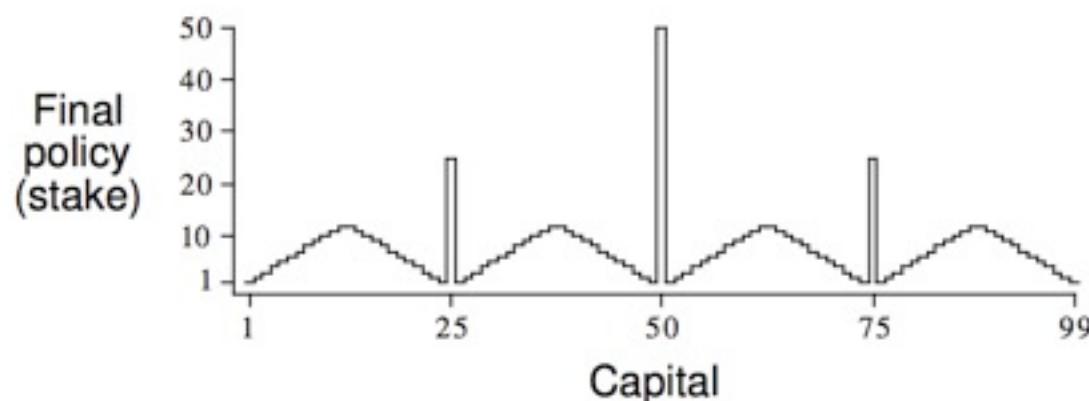
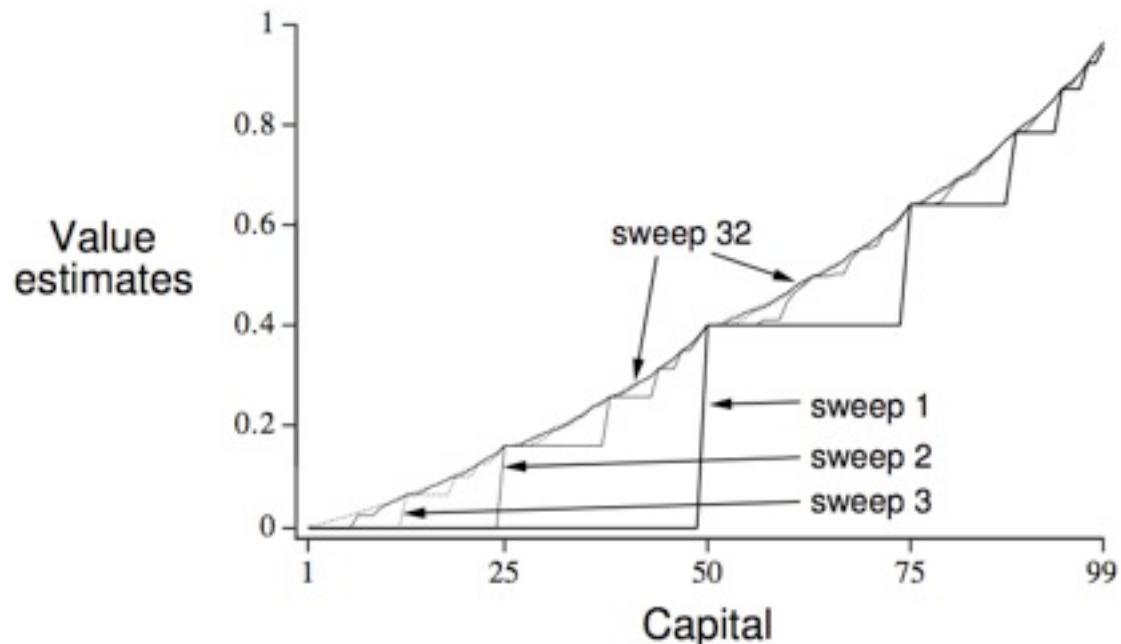
Output a deterministic policy, π , such that

$$\pi(s) = \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

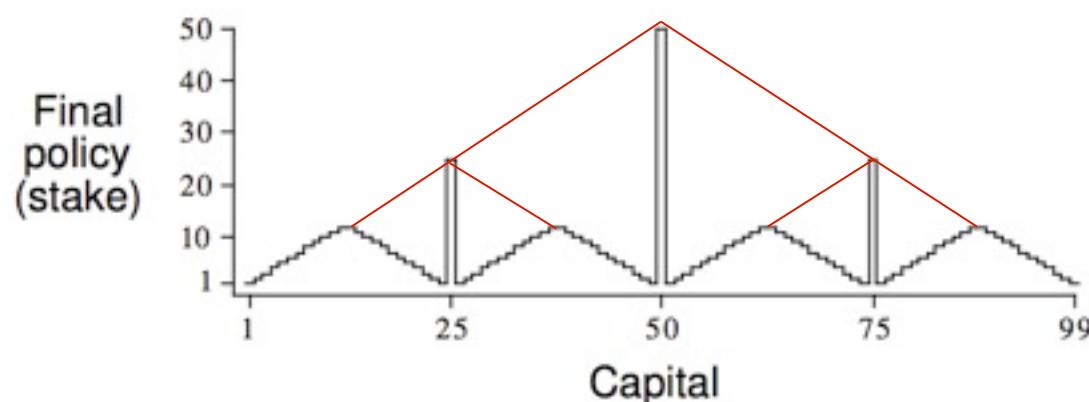
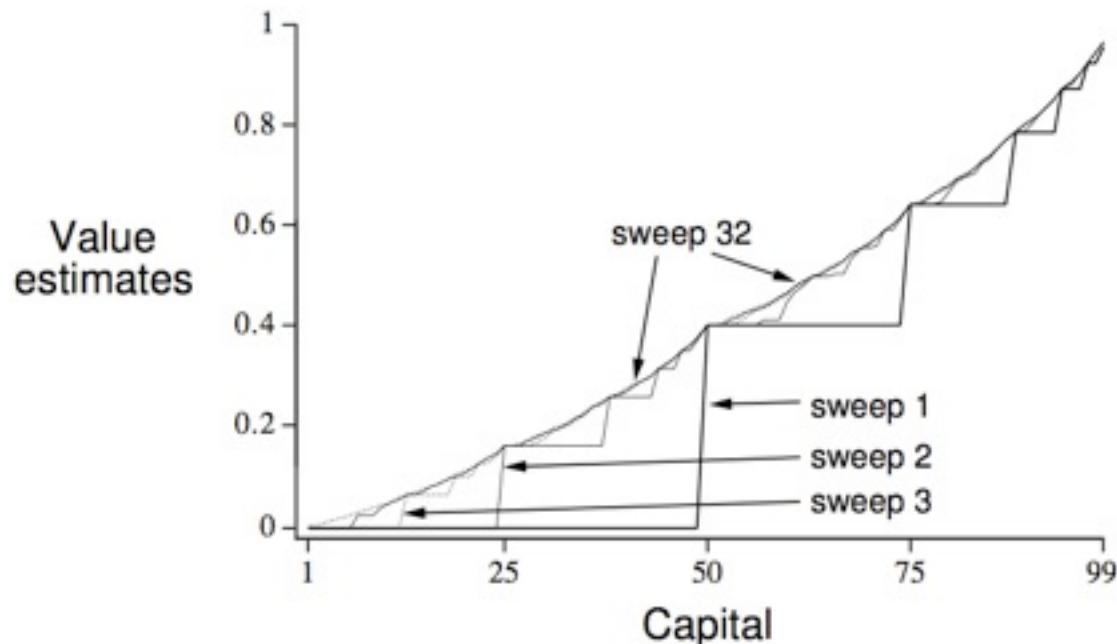
Gambler's Problem

- Gambler can repeatedly bet \$ on a coin flip
- Heads he wins his stake, tails he loses it
- Initial capital $\in \{\$1, \$2, \dots \$99\}$
- Gambler wins if his capital becomes \$100
loses if it becomes \$0
- Coin is unfair
 - Heads (gambler wins) with probability $p = .4$
- States, Actions, Rewards?

Gambler's Problem Solution



Gambler's Problem Solution



Policy Improvement

Suppose we have computed v_π for a deterministic policy π .

For a given state s ,

would it be better to do an action $a \neq \pi(s)$?

It is better to switch to action a for state s if and only if

$$q_\pi(s, a) > v_\pi(s)$$

And, we can compute $q_\pi(s, a)$ from v_π by:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')]. \end{aligned}$$

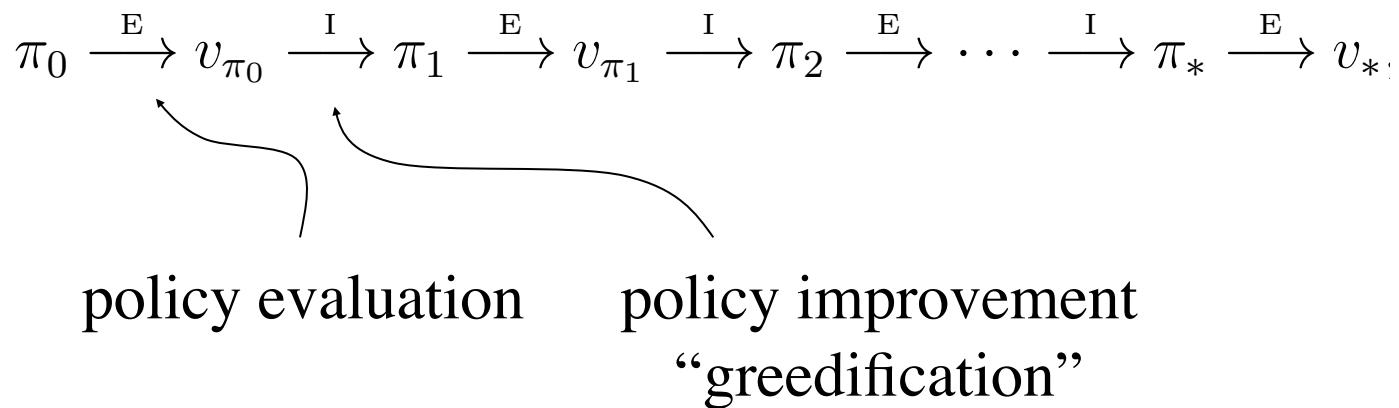
Policy Improvement Cont.

Do this for all states to get a new policy $\pi' \geq \pi$ that is **greedy** with respect to v_π :

$$\begin{aligned}\pi'(s) &= \arg \max_a q_\pi(s, a) \\ &= \arg \max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \arg \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')],\end{aligned}$$

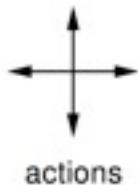
What if the policy is unchanged by this?
Then the policy must be optimal!

Policy Iteration



Iterative Policy Eval for the Small Gridworld

π = equiprobable random action choices



	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$R = -1$
on all transitions

$\gamma = 1$

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

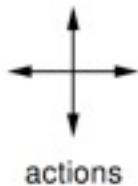
V_k for the
Random Policy

Greedy Policy
w.r.t. V_k

random
policy

Iterative Policy Eval for the Small Gridworld

π = equiprobable random action choices

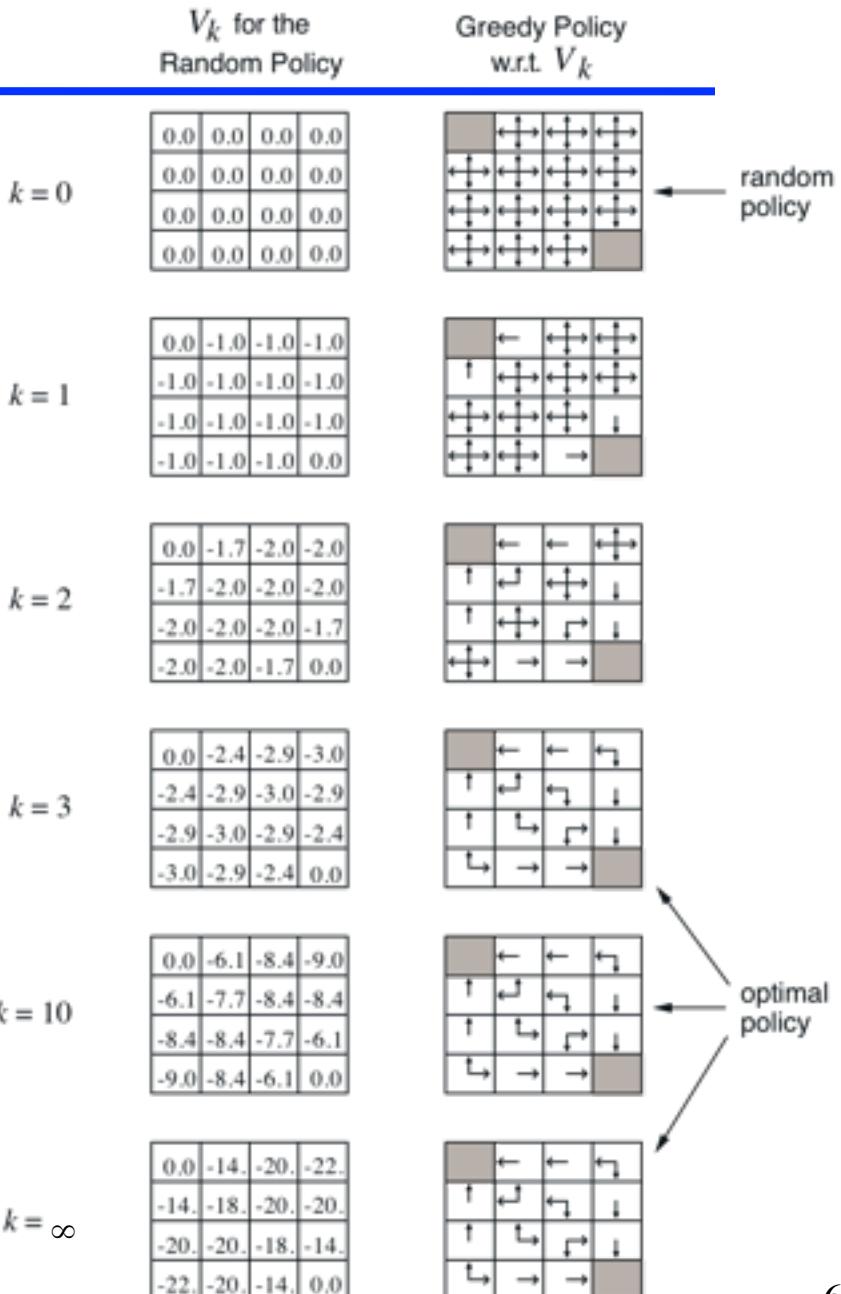


	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$R = -1$
on all transitions

$\gamma = 1$

- An undiscounted episodic task
- Nonterminal states: 1, 2, . . . , 14;
- One terminal state (shown twice as shaded squares)
- Actions that would take agent off the grid leave state unchanged
- Reward is -1 until the terminal state is reached



random policy

optimal policy

Policy Iteration – One array version (+ policy)

1. Initialization

$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation

Repeat

$$\Delta \leftarrow 0$$

For each $s \in \mathcal{S}$:

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_{s',r} p(s', r | s, \pi(s)) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$ (a small positive number)

3. Policy Improvement

policy-stable \leftarrow true

For each $s \in \mathcal{S}$:

$$a \leftarrow \pi(s)$$

$$\pi(s) \leftarrow \arg \max_a \sum_{s',r} p(s', r | s, a) [r + \gamma V(s')]$$

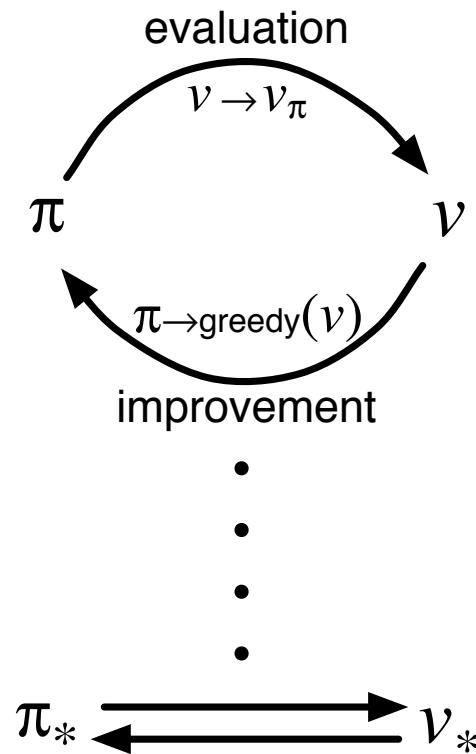
If $a \neq \pi(s)$, then *policy-stable* \leftarrow false

If *policy-stable*, then stop and return V and π ; else go to 2

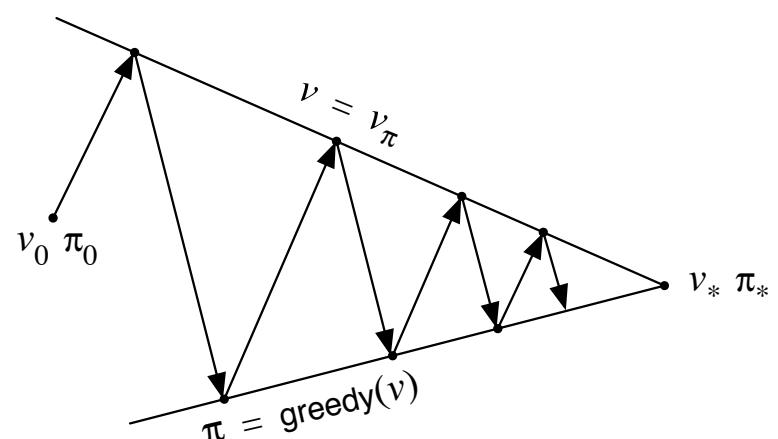
Generalized Policy Iteration

Generalized Policy Iteration (GPI):

any interaction of policy evaluation and policy improvement,
independent of their granularity.



A geometric metaphor for convergence of GPI:

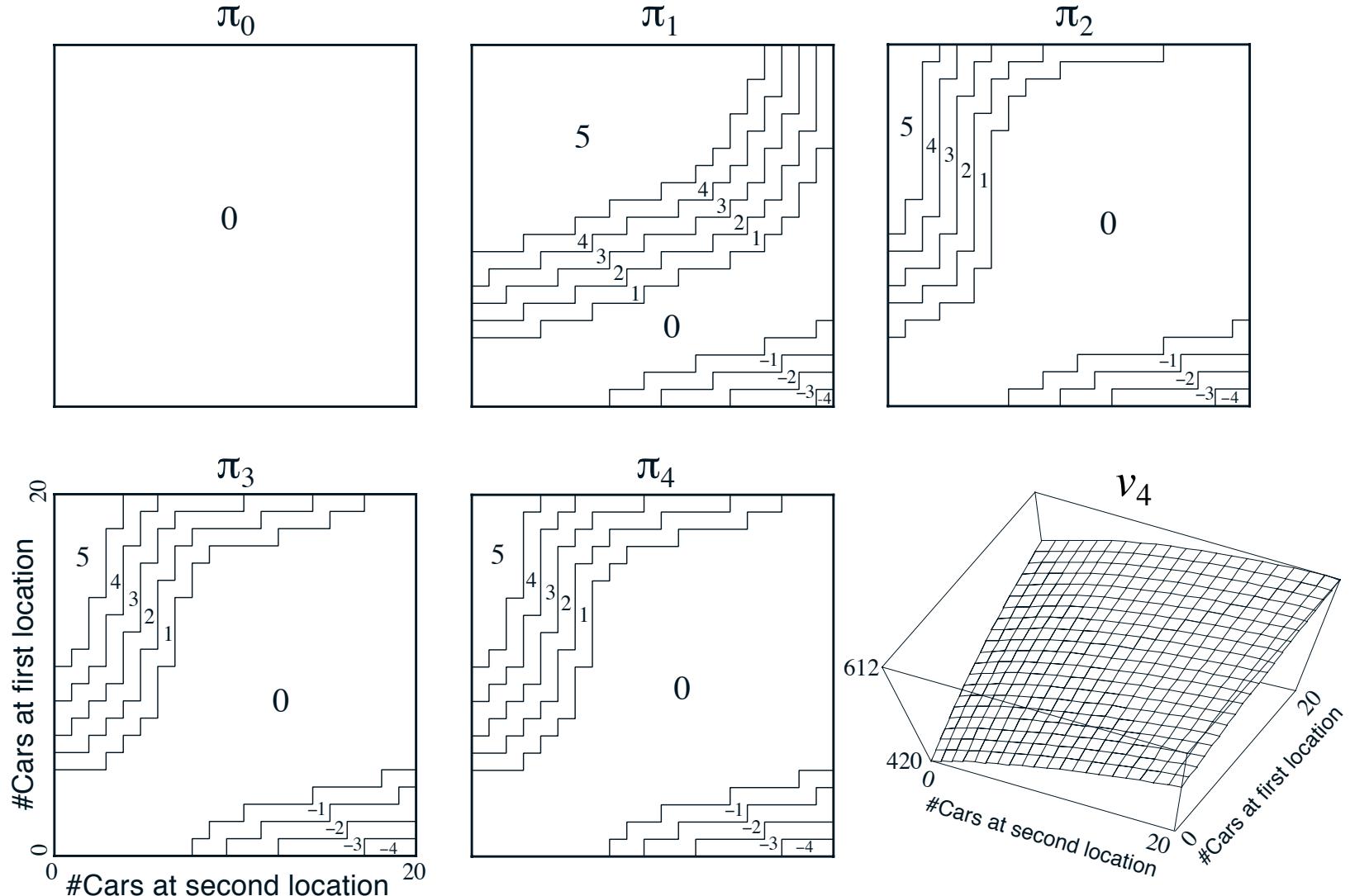


Jack's Car Rental

- \$10 for each car rented (must be available when request rec'd)
- Two locations, maximum of 20 cars at each
- Cars returned and requested randomly
 - Poisson distribution, n returns/requests with prob $\frac{\lambda^n}{n!} e^{-\lambda}$
 - 1st location: average requests = 3, average returns = 3
 - 2nd location: average requests = 4, average returns = 2
- Can move up to 5 cars between locations overnight

- States, Actions, Rewards?
- Transition probabilities?

Jack's Car Rental



Jack's CR Exercise

- Suppose the first car moved is free
 - From 1st to 2nd location
 - Because an employee travels that way anyway (by bus)
- Suppose only 10 cars can be parked for free at each location
 - More than 10 cost \$4 for using an extra parking lot
- Such arbitrary nonlinearities are common in real problems

Asynchronous DP

- All the DP methods described so far require exhaustive sweeps of the entire state set.
- Asynchronous DP does not use sweeps. Instead it works like this:
 - Repeat until convergence criterion is met:
 - Pick a state at random and apply the appropriate backup
- Still need lots of computation, but does not get locked into hopelessly long sweeps
- Can you select states to backup intelligently? YES: an agent's experience can act as a guide.

Efficiency of DP

- To find an optimal policy is polynomial in the number of states...
- BUT, the number of states is often astronomical, e.g., often growing exponentially with the number of state variables (what Bellman called “the curse of dimensionality”).
- In practice, classical DP can be applied to problems with a few millions of states.
- Asynchronous DP can be applied to larger problems, and is appropriate for parallel computation.
- It is surprisingly easy to come up with MDPs for which DP methods are not practical.

Summary

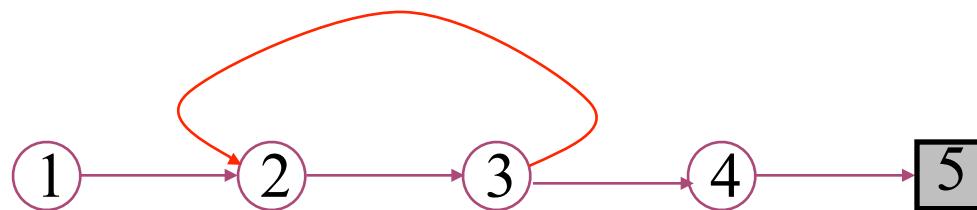
- Policy evaluation: backups without a max
- Policy improvement: form a greedy policy, if only locally
- Policy iteration: alternate the above two processes
- Value iteration: backups with a max
- Full backups (to be contrasted later with sample backups)
- Generalized Policy Iteration (GPI)
- Asynchronous DP: a way to avoid exhaustive sweeps
- **Bootstrapping**: updating estimates based on other estimates
- Biggest limitation of DP is that it requires a *probability model* (as opposed to a generative or simulation model)

Chapter 5: Monte Carlo Methods

- Monte Carlo methods are learning methods
Experience → values, policy
- Monte Carlo methods can be used in two ways:
 - *On-line*: No model necessary and still attains optimality
 - *Simulated*: No need for a *full* model
- Monte Carlo methods learn from *complete* sample returns
 - Only defined for episodic tasks (in this book)
- Like an associative version of a bandit method

Monte Carlo Policy Evaluation

- ❑ *Goal:* learn $v_\pi(s)$
- ❑ *Given:* some number of episodes under π which contain s
- ❑ *Idea:* Average returns observed after visits to s



- ❑ *Every-Visit MC:* average returns for *every* time s is visited in an episode
- ❑ *First-visit MC:* average returns only for *first* time s is visited in an episode
- ❑ Both converge asymptotically

First-visit Monte Carlo policy evaluation

Initialize:

$\pi \leftarrow$ policy to be evaluated

$V \leftarrow$ an arbitrary state-value function

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:

 Generate an episode using π

 For each state s appearing in the episode:

$G \leftarrow$ return following the first occurrence of s

 Append G to $Returns(s)$

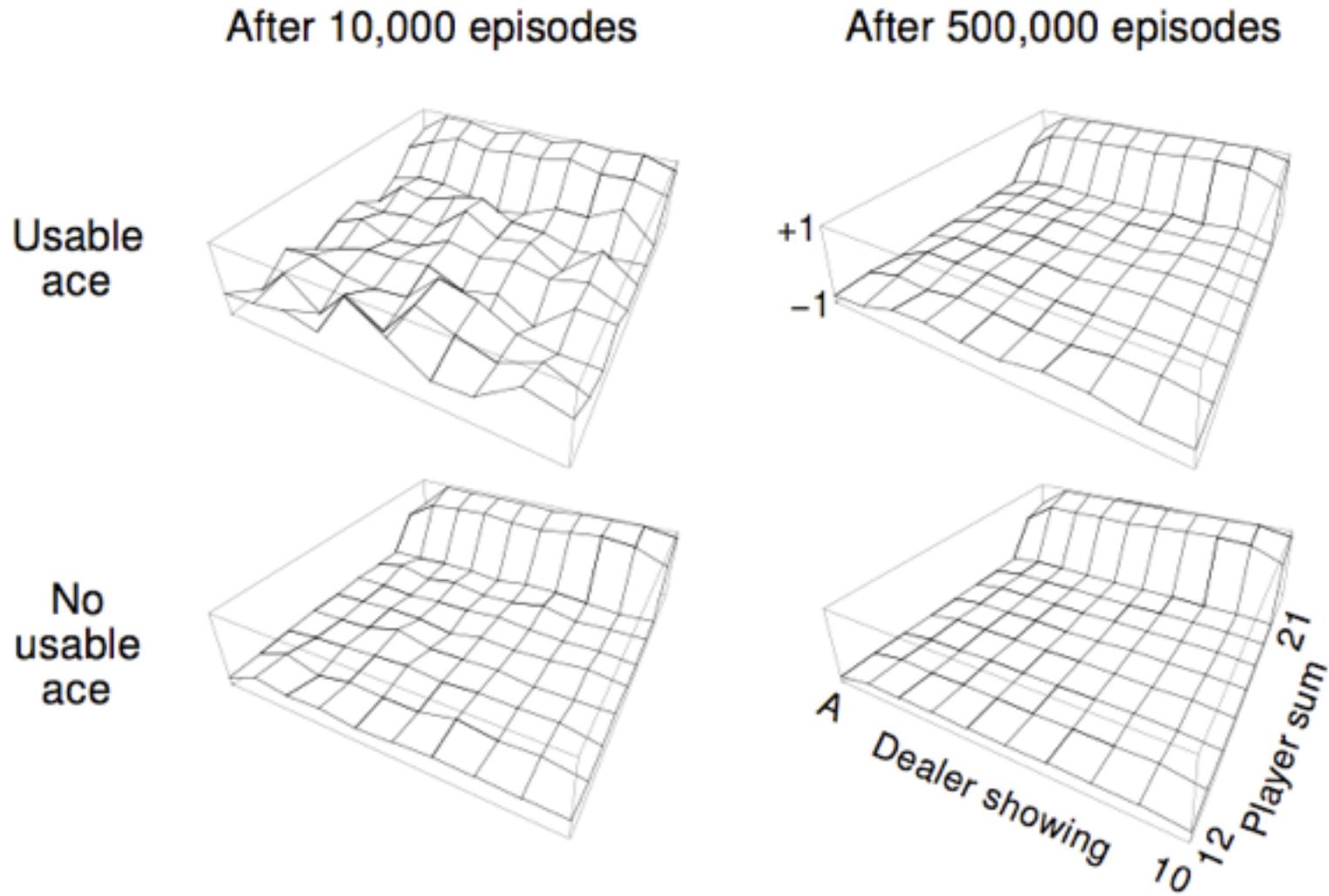
$V(s) \leftarrow$ average($Returns(s)$)

Blackjack example

- *Object*: Have your card sum be greater than the dealer's without exceeding 21.
- *States* (200 of them):
 - current sum (12-21)
 - dealer's showing card (ace-10)
 - do I have a useable ace?
- *Reward*: +1 for winning, 0 for a draw, -1 for losing
- *Actions*: stick (stop receiving cards), hit (receive another card)
- *Policy*: Stick if my sum is 20 or 21, else hit

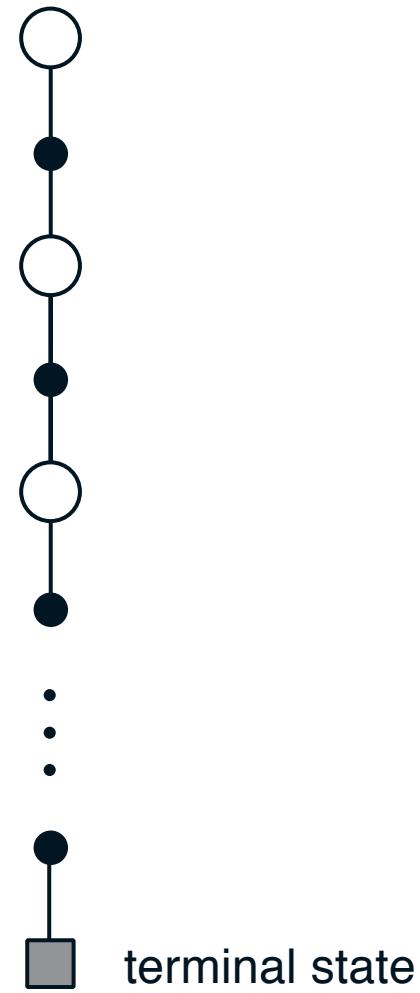


Learned blackjack state-value functions



Backup diagram for Monte Carlo

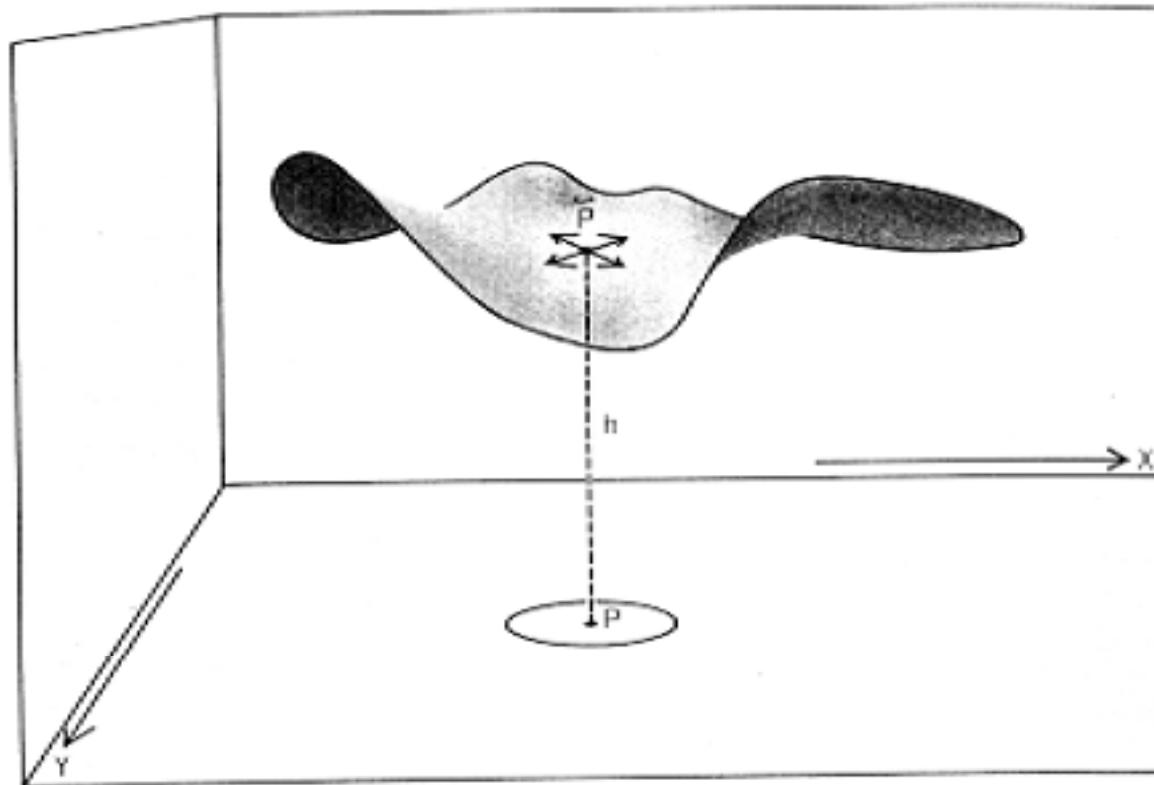
- ❑ Entire rest of episode included
- ❑ Only one choice considered at each state (unlike DP)
 - thus, there will be an explore/exploit dilemma
- ❑ Does not bootstrap from successor states's values (unlike DP)
- ❑ Time required to estimate one state does not depend on the total number of states



The Power of Monte Carlo

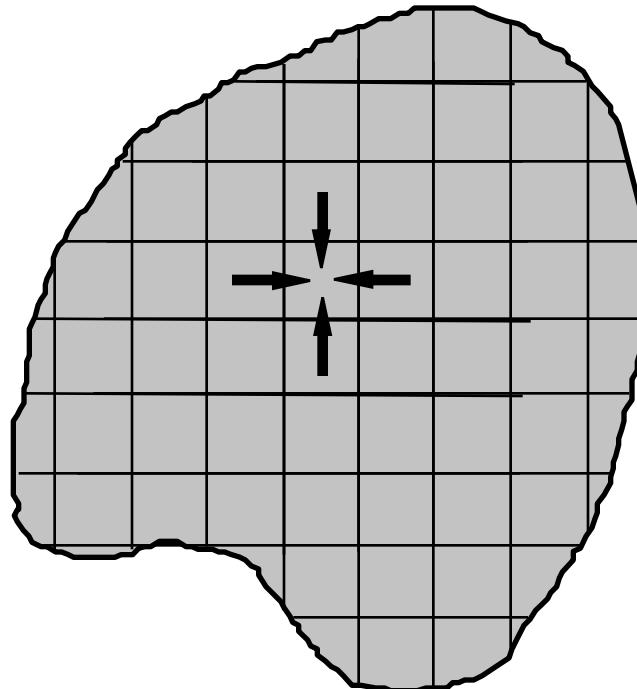
e.g., Elastic Membrane (Dirichlet Problem)

How do we compute the shape of the membrane or bubble?

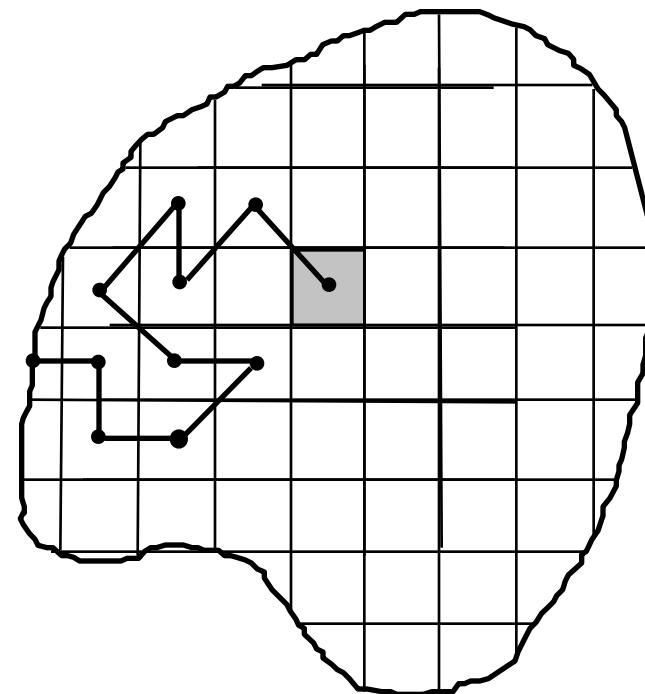


Two Approaches

Relaxation



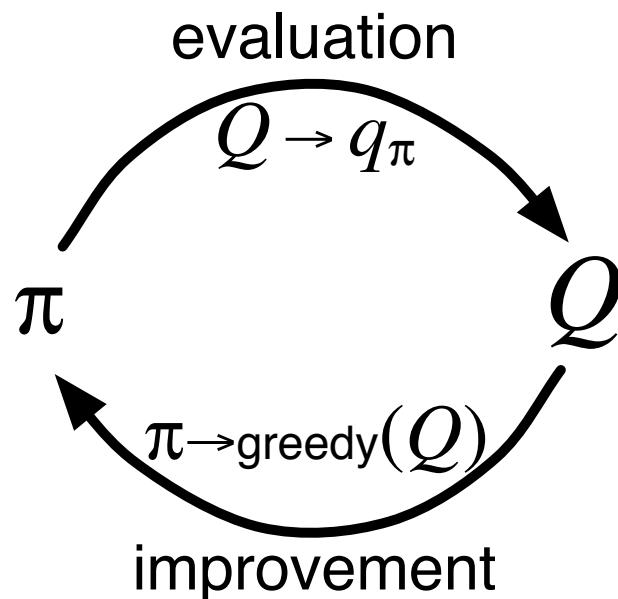
Kakutani's algorithm, 1945



Monte Carlo Estimation of Action Values (Q)

- Monte Carlo is most useful when a model is not available
 - We want to learn q^*
- $q_\pi(s,a)$ - average return starting from state s and action a following π
- Converges asymptotically *if* every state-action pair is visited
- *Exploring starts*: Every state-action pair has a non-zero probability of being the starting pair

Monte Carlo Control



- MC policy iteration: Policy evaluation using MC methods followed by policy improvement
- Policy improvement step: greedify with respect to value (or action-value) function

Convergence of MC Control

- Greedified policy meets the conditions for policy improvement:

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg \max_a q_{\pi_k}(s, a)) \\ &= \max_a q_{\pi_k}(s, a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &\geq v_{\pi_k}(s). \end{aligned}$$

- And thus must be $\geq \pi_k$ by the policy improvement theorem
- This assumes exploring starts and infinite number of episodes for MC policy evaluation
- To solve the latter:
 - update only to a given level of performance
 - alternate between evaluation and improvement per episode

Monte Carlo Exploring Starts

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$$Q(s, a) \leftarrow \text{arbitrary}$$

$$\pi(s) \leftarrow \text{arbitrary}$$

$$Returns(s, a) \leftarrow \text{empty list}$$

Fixed point is optimal policy π^*

Now proven (almost)

Repeat forever:

Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability > 0

Generate an episode starting from S_0, A_0 , following π

For each pair s, a appearing in the episode:

$G \leftarrow$ return following the first occurrence of s, a

Append G to $Returns(s, a)$

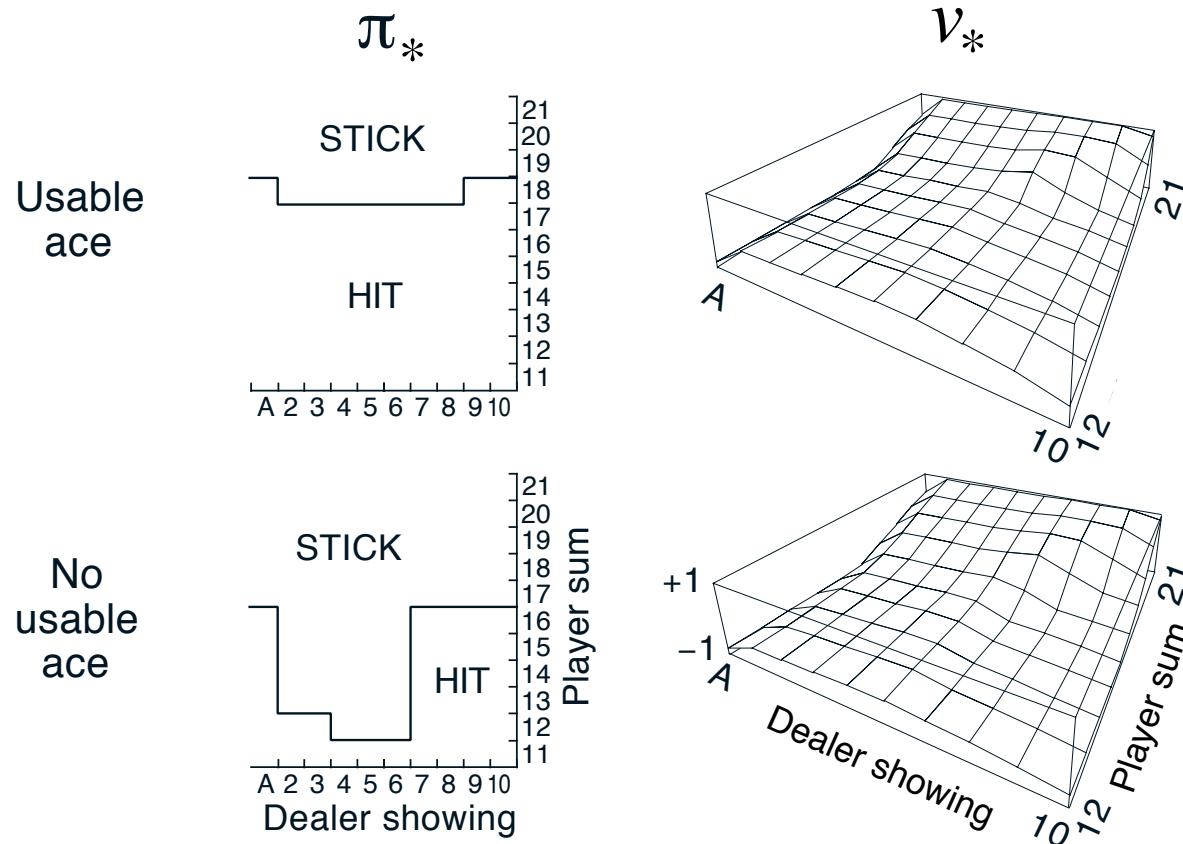
$Q(s, a) \leftarrow \text{average}(Returns(s, a))$

For each s in the episode:

$\pi(s) \leftarrow \arg \max_a Q(s, a)$

Blackjack example continued

- ❑ Exploring starts
- ❑ Initial policy as described before



On-policy Monte Carlo Control

- *On-policy*: learn about policy currently executing
- How do we get rid of exploring starts?
 - Need *soft* policies: $\pi(a|s) > 0$ for all s and a
 - e.g. ϵ -soft policy:

$$\begin{array}{ll} \frac{\epsilon}{|\mathcal{A}(s)|} & 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} \\ \text{non-max} & \text{greedy} \end{array}$$

- Similar to GPI: move policy *towards* greedy policy (i.e. ϵ -soft)
- Converges to best ϵ -soft policy

On-policy MC Control

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$Q(s, a) \leftarrow$ arbitrary

$Returns(s, a) \leftarrow$ empty list

$\pi(a|s) \leftarrow$ an arbitrary ε -soft policy

Repeat forever:

(a) Generate an episode using π

(b) For each pair s, a appearing in the episode:

$G \leftarrow$ return following the first occurrence of s, a

Append G to $Returns(s, a)$

$Q(s, a) \leftarrow \text{average}(Returns(s, a))$

(c) For each s in the episode:

$A^* \leftarrow \arg \max_a Q(s, a)$

For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

Off-policy prediction

- Learn the value of target policy π from experience due to behavior policy μ
- Using “importance sampling” — weighting each return by its relative likelihood under the two policies:

$$\rho_t^T \doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} \mu(A_k|S_k) p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{\mu(A_k|S_k)}$$

- Ordinary importance sampling:

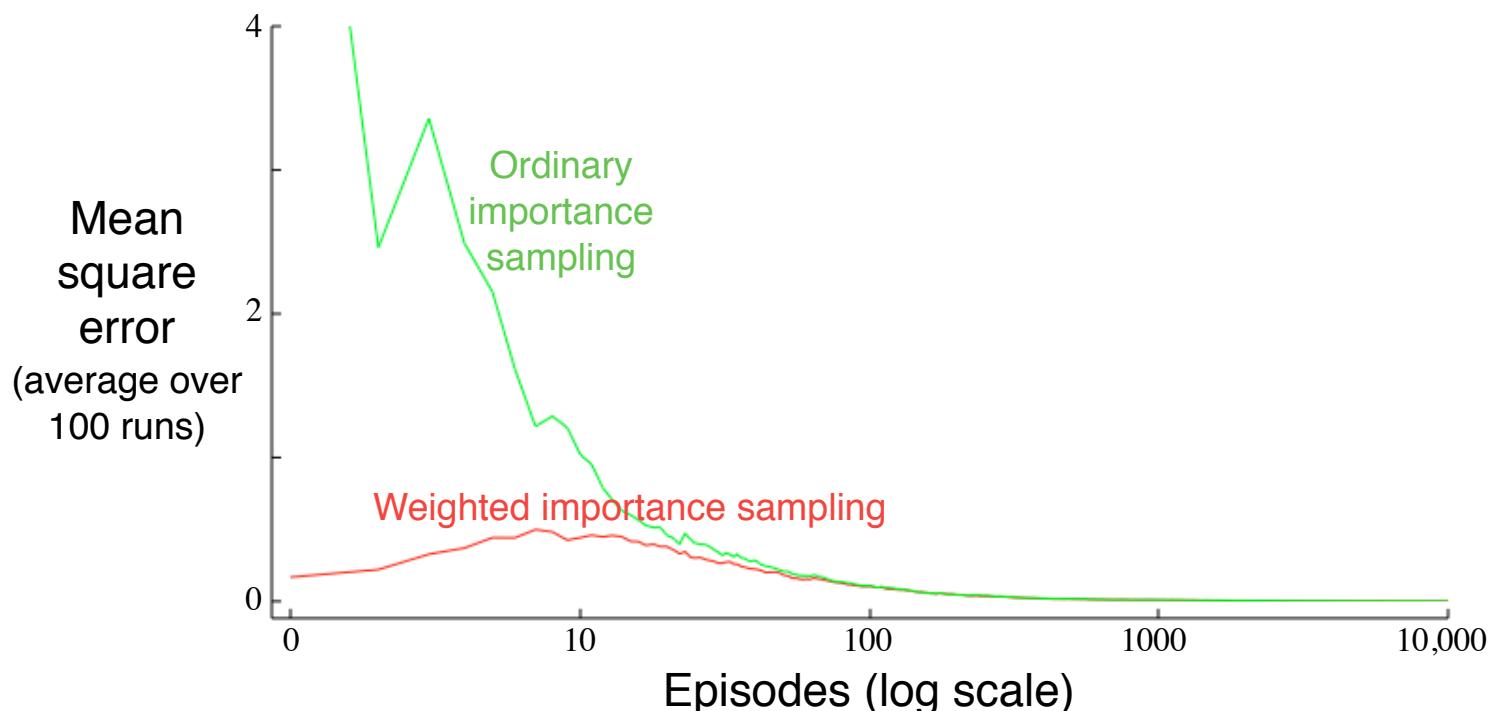
$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_t^{T(t)} G_t}{|\mathcal{T}(s)|}$$

- Weighted importance sampling:

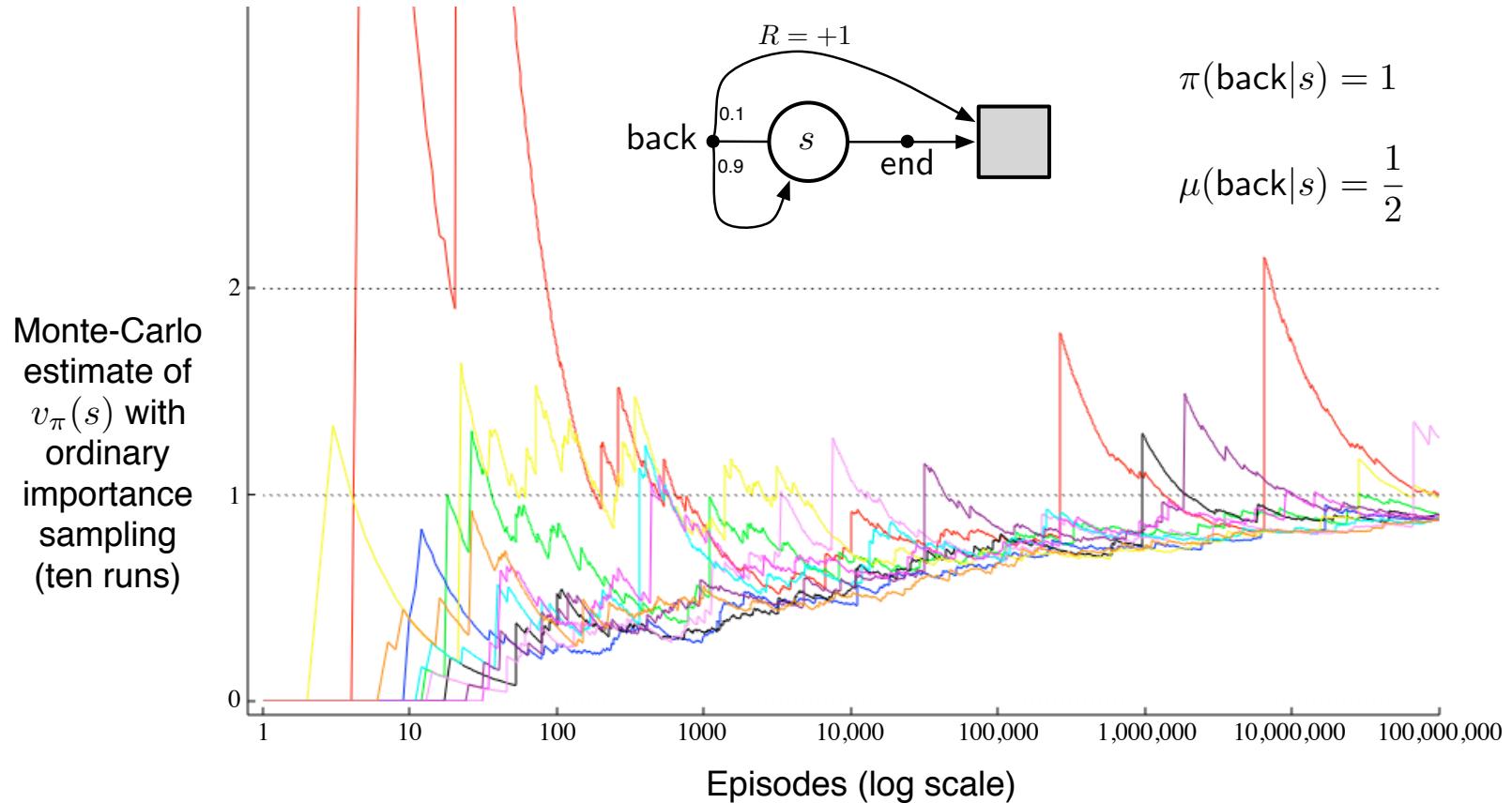
$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_t^{T(t)} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_t^{T(t)}}$$

Off-policy Estimation of a Blackjack State Value

- State is player-sum 13, useable ace, dealer-showing 2
- Policy is stick only on 20 or 21
- True value ≈ -0.27726



Example of infinite variance under ordinary importance sampling



What is the variance under *weighted* importance sampling?

Incremental off-policy policy evaluation

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$$Q(s, a) \leftarrow \text{arbitrary}$$

$$C(s, a) \leftarrow 0$$

$\mu(a|s) \leftarrow \text{an arbitrary soft behavior policy}$

$\pi(a|s) \leftarrow \text{an arbitrary target policy}$

Repeat forever:

Generate an episode using μ :

$$S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T, S_T$$

$$G \leftarrow 0$$

$$W \leftarrow 1$$

For $t = T - 1, T - 2, \dots$ downto 0:

$$G \leftarrow \gamma G + R_{t+1}$$

$$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$$

$$W \leftarrow W \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}$$

If $W = 0$ then ExitForLoop

Off-policy Monte Carlo control

- *Off-policy*: learn about a policy different than the one being executed

Target policy is one being learned about

Behavior policy is the one executed, generating behavior

- Idea: importance sampling
 - Average returns from behavior policy, each weighted by the ratio of the probabilities of the it having occurred under the target and behavior policies

Incremental off-policy MC control

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$$Q(s, a) \leftarrow \text{arbitrary}$$

$$C(s, a) \leftarrow 0$$

$\pi(s) \leftarrow$ a deterministic policy that is greedy with respect to Q

Repeat forever:

Generate an episode using any soft policy μ :

$$S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T, S_T$$

$$G \leftarrow 0$$

$$W \leftarrow 1$$

For $t = T - 1, T - 2, \dots$ downto 0:

$$G \leftarrow \gamma G + R_{t+1}$$

$$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$$

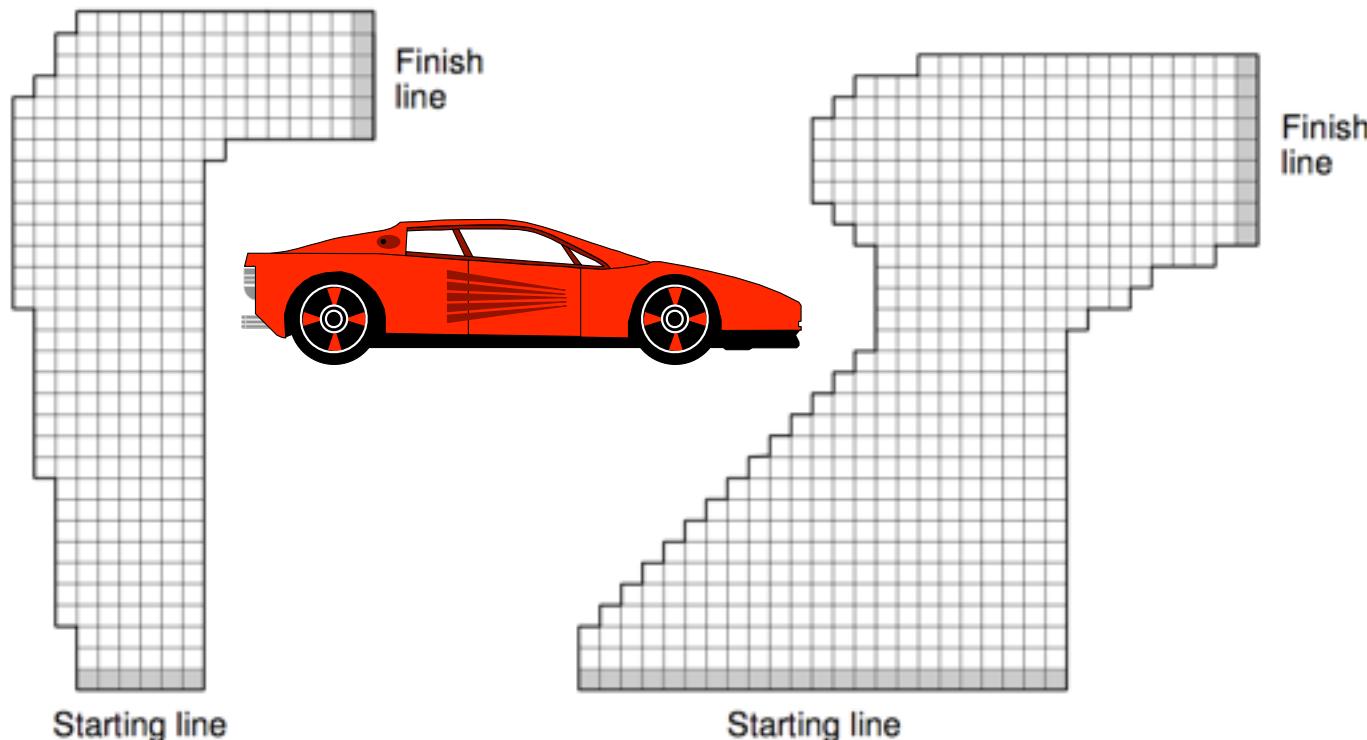
$$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a) \quad (\text{with ties broken arbitrarily})$$

$$W \leftarrow W \frac{1}{\mu(A_t | S_t)}$$

If $W = 0$ then ExitForLoop

Racetrack Exercise

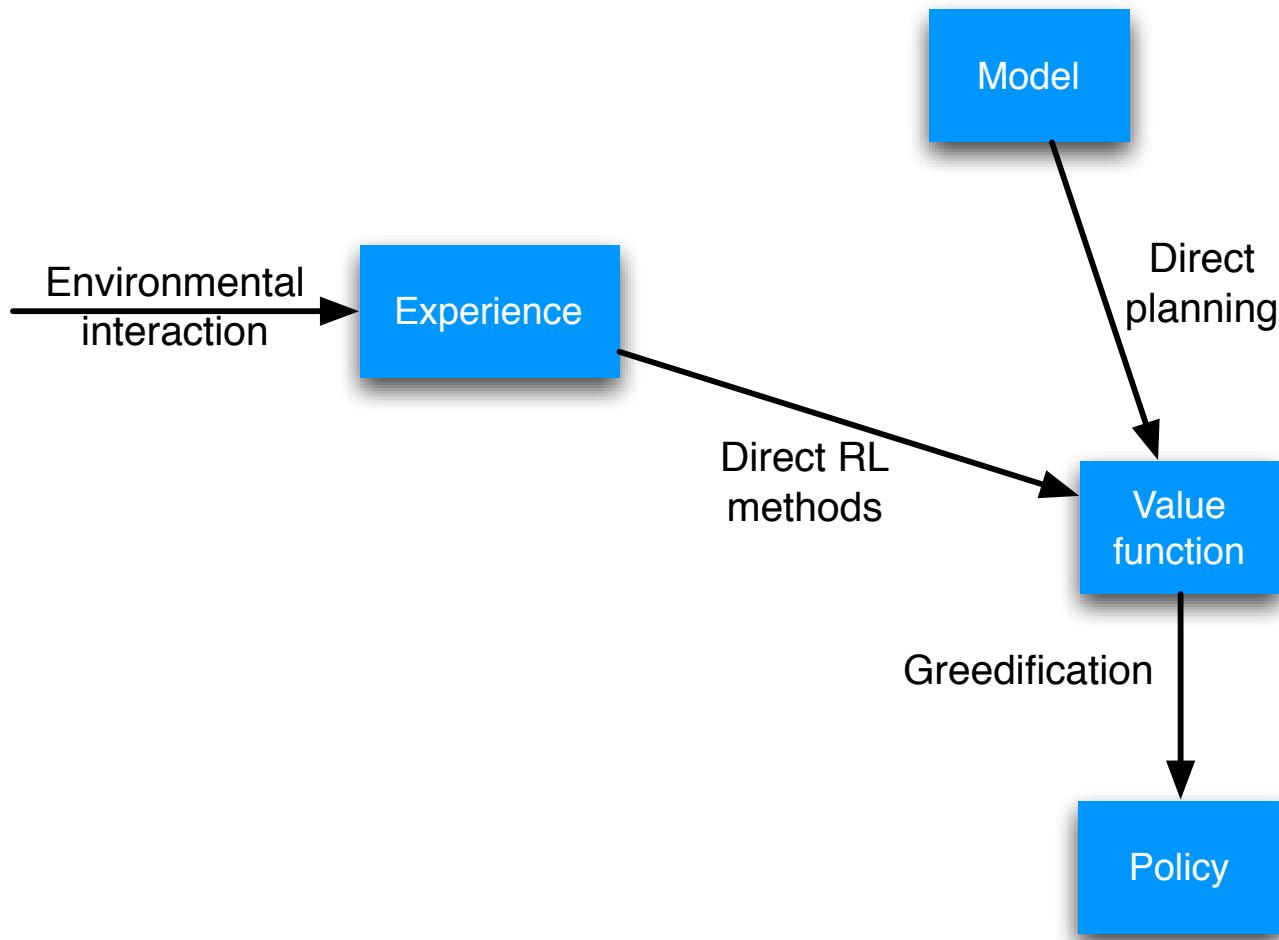
- *States*: grid squares, velocity horizontal and vertical
- *Rewards*: -1 on track, -5 off track
- *Actions*: +1, -1, 0 to velocity
- $0 < \text{Velocity} < 5$
- Stochastic: 50% of the time it moves 1 extra square up or right



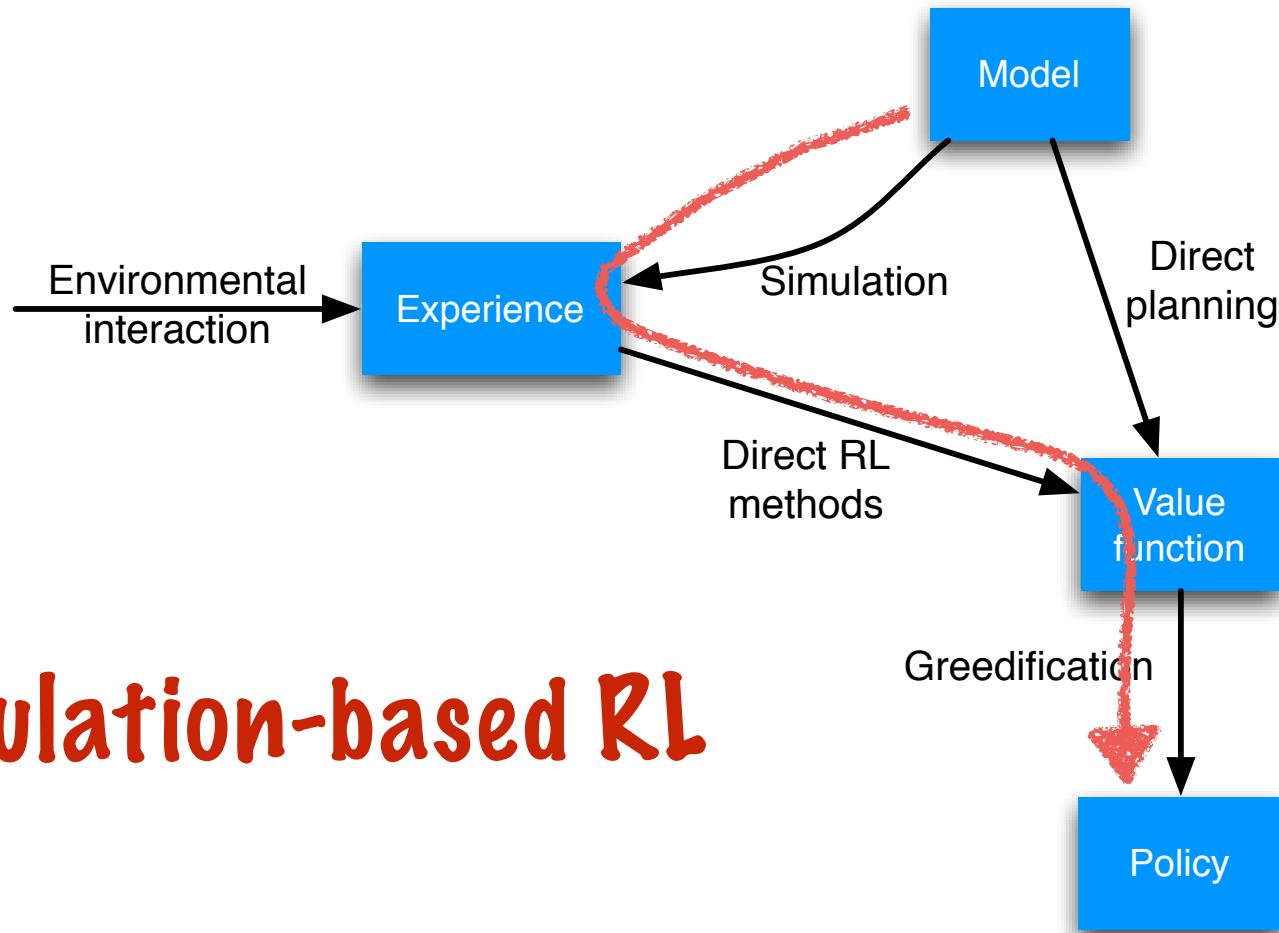
Summary

- MC has several advantages over DP:
 - Can learn directly from interaction with environment
 - No need for full models
 - No need to learn about ALL states
 - Less harmed by violating Markov property (later in book)
- MC methods provide an alternate policy evaluation process
- One issue to watch for: maintaining sufficient exploration
 - exploring starts, soft policies
- Introduced distinction between *on-policy* and *off-policy* methods
- No bootstrapping (as opposed to DP)

Paths to a policy

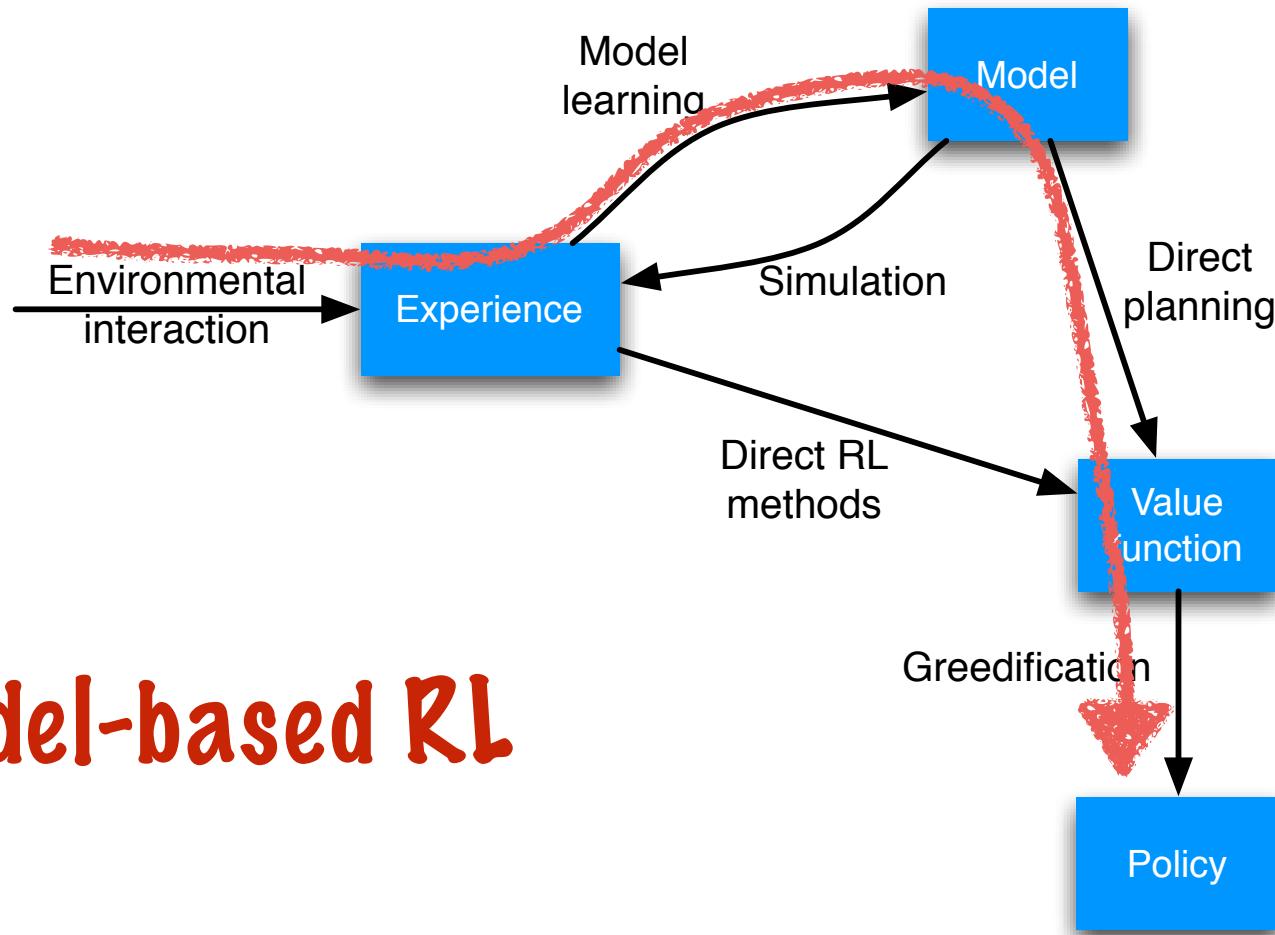


Paths to a policy

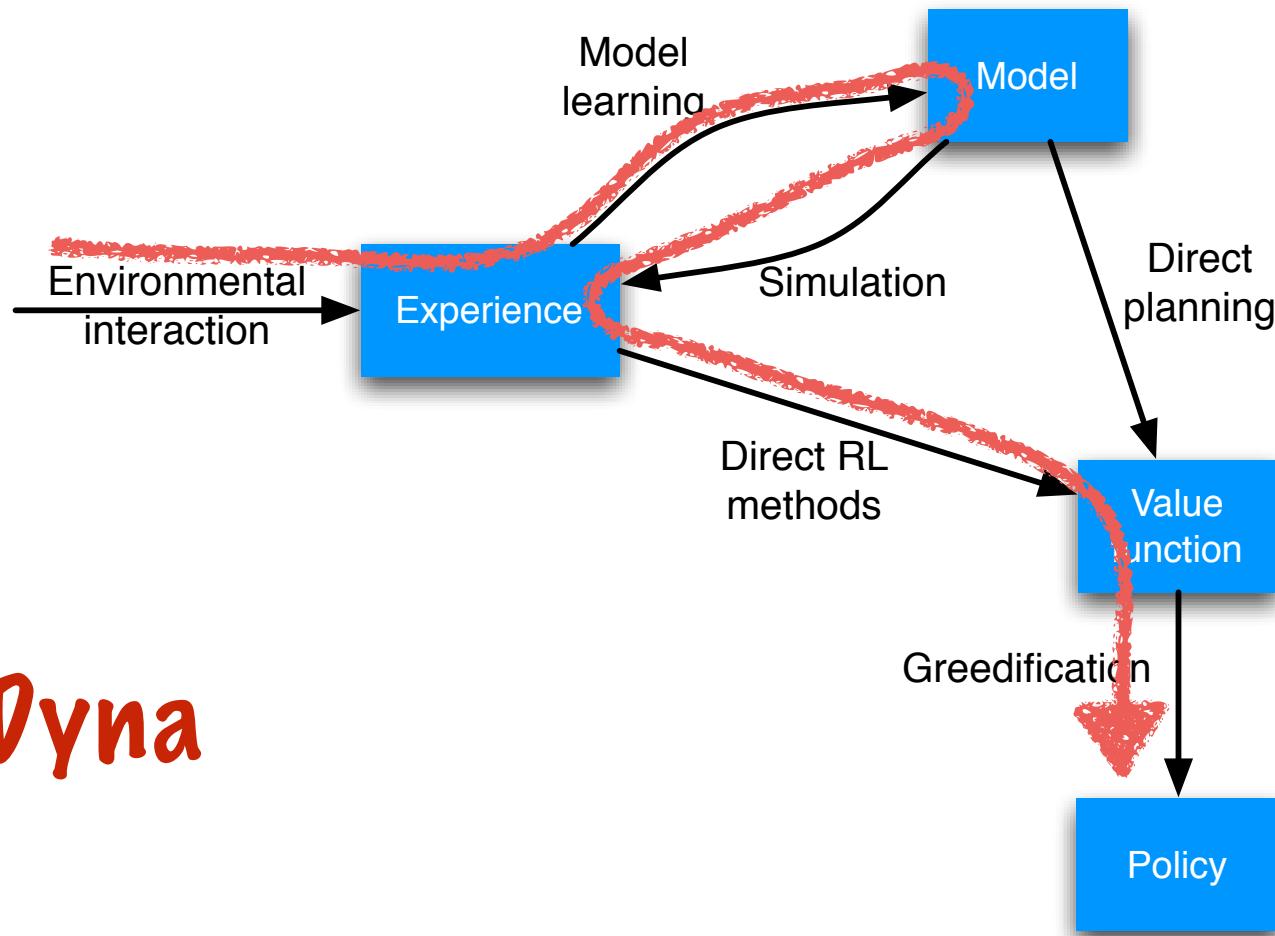


Simulation-based RL

Paths to a policy



Paths to a policy



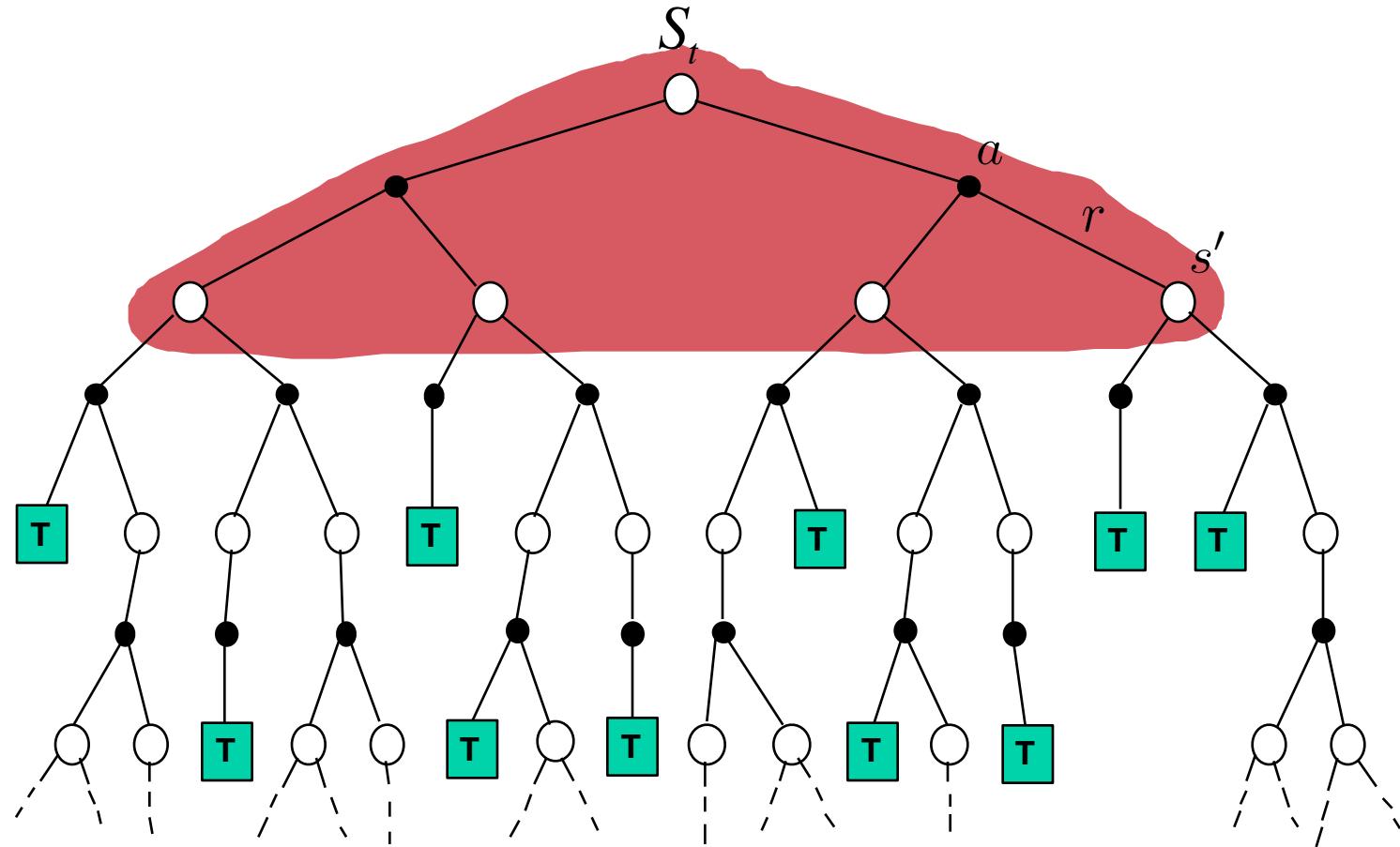
Chapter 6: Temporal Difference Learning

Objectives of this chapter:

- Introduce Temporal Difference (TD) learning
- Focus first on policy evaluation, or prediction, methods
- Compare efficiency of TD learning with MC learning
- Then extend to control methods

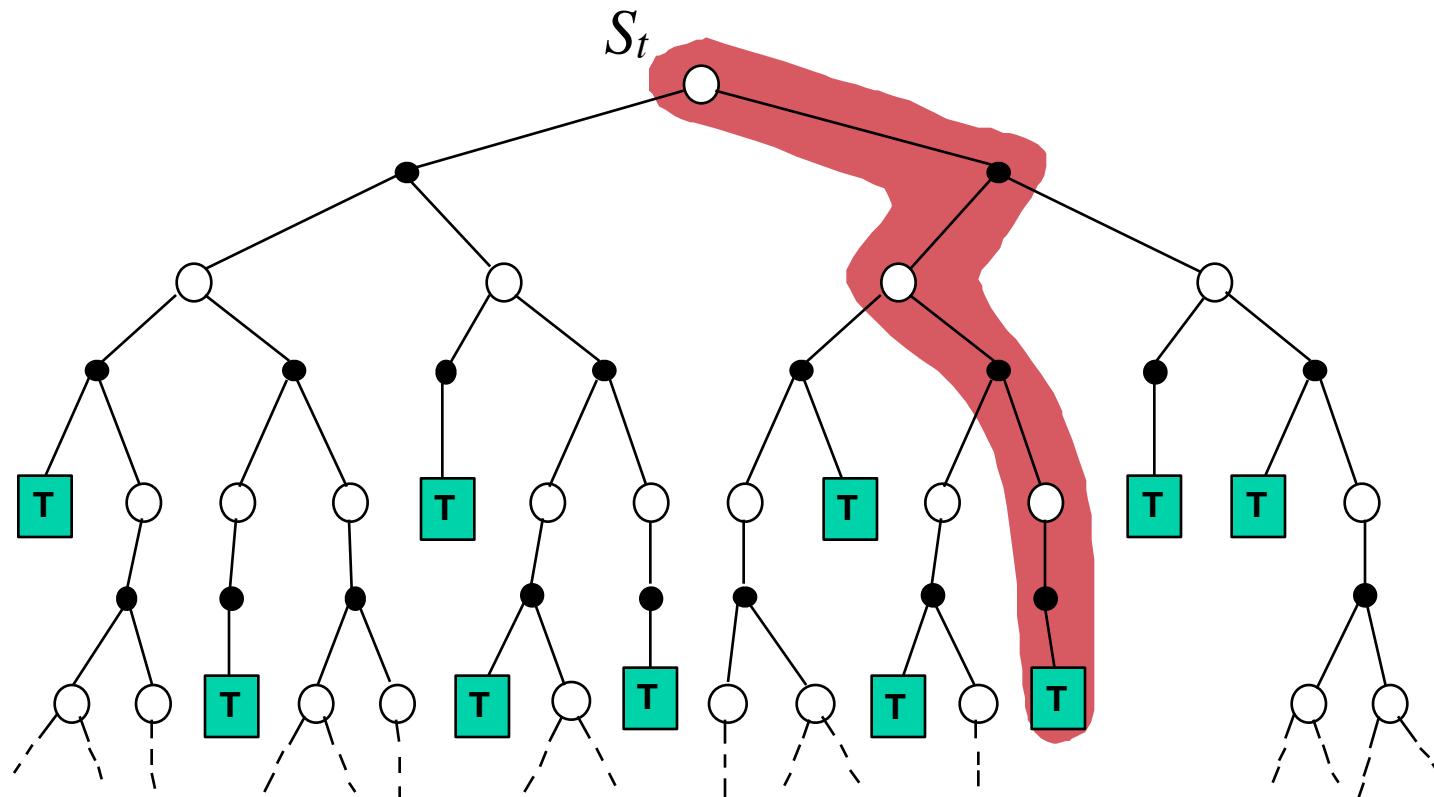
cf. Dynamic Programming

$$V(S_t) \leftarrow E_{\pi} \left[R_{t+1} + \gamma V(S_{t+1}) \right] = \sum_a \pi(a|S_t) \sum_{s',r} p(s',r|S_t,a) [r + \gamma V(s')]$$



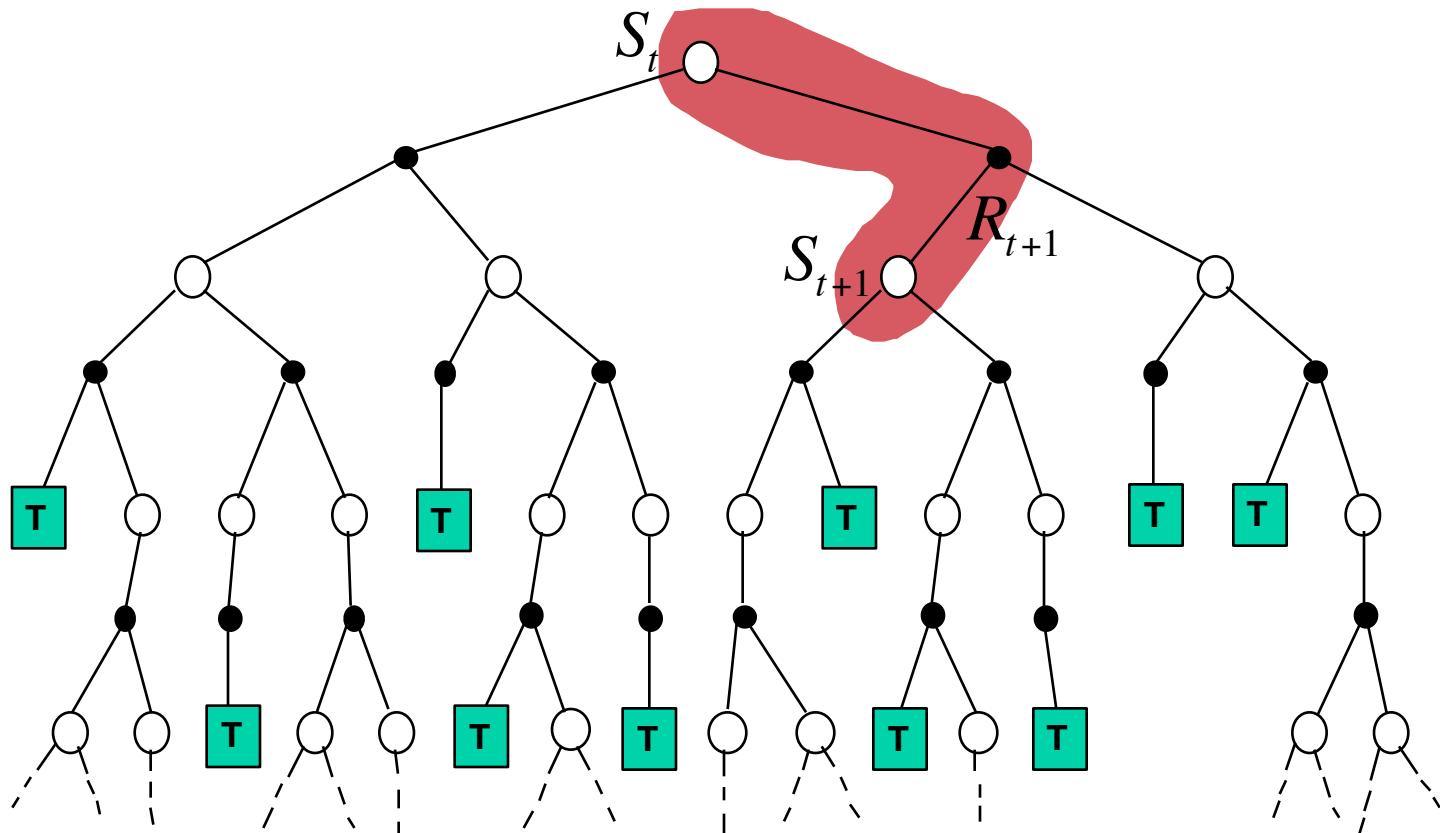
Simple Monte Carlo

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)]$$



Simplest TD Method

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$



TD methods bootstrap and sample

- **Bootstrapping:** update involves an *estimate*
 - MC does not bootstrap
 - DP bootstraps
 - TD bootstraps
- **Sampling:** update does not involve an *expected value*
 - MC samples
 - DP does not sample
 - TD samples

TD Prediction

Policy Evaluation (the prediction problem):

for a given policy π , compute the state-value function v_π

Recall: Simple every-visit Monte Carlo method:

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)]$$


target: the actual return after time t

The simplest temporal-difference method TD(0):

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

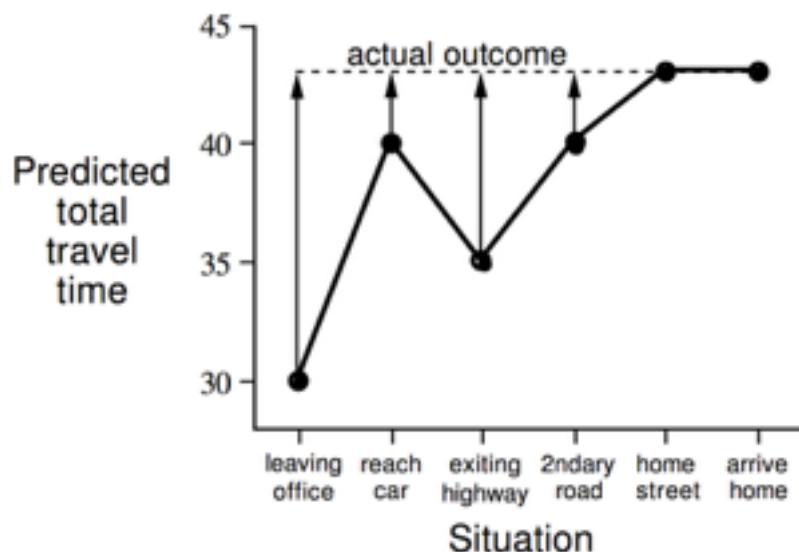

target: an estimate of the return

Example: Driving Home

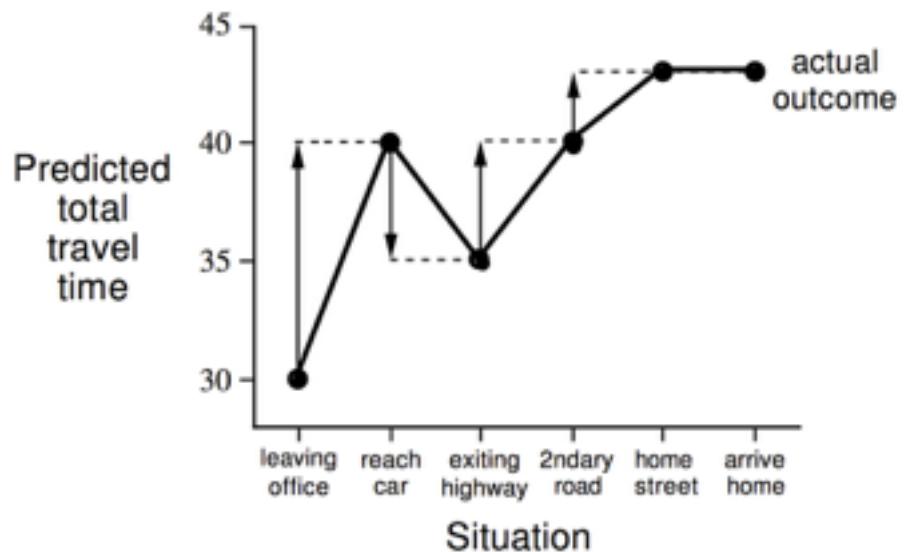
<i>State</i>	<i>Elapsed Time</i> (minutes)	<i>Predicted</i> <i>Time to Go</i>	<i>Predicted</i> <i>Total Time</i>
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

Driving Home

Changes recommended by
Monte Carlo methods ($\alpha=1$)



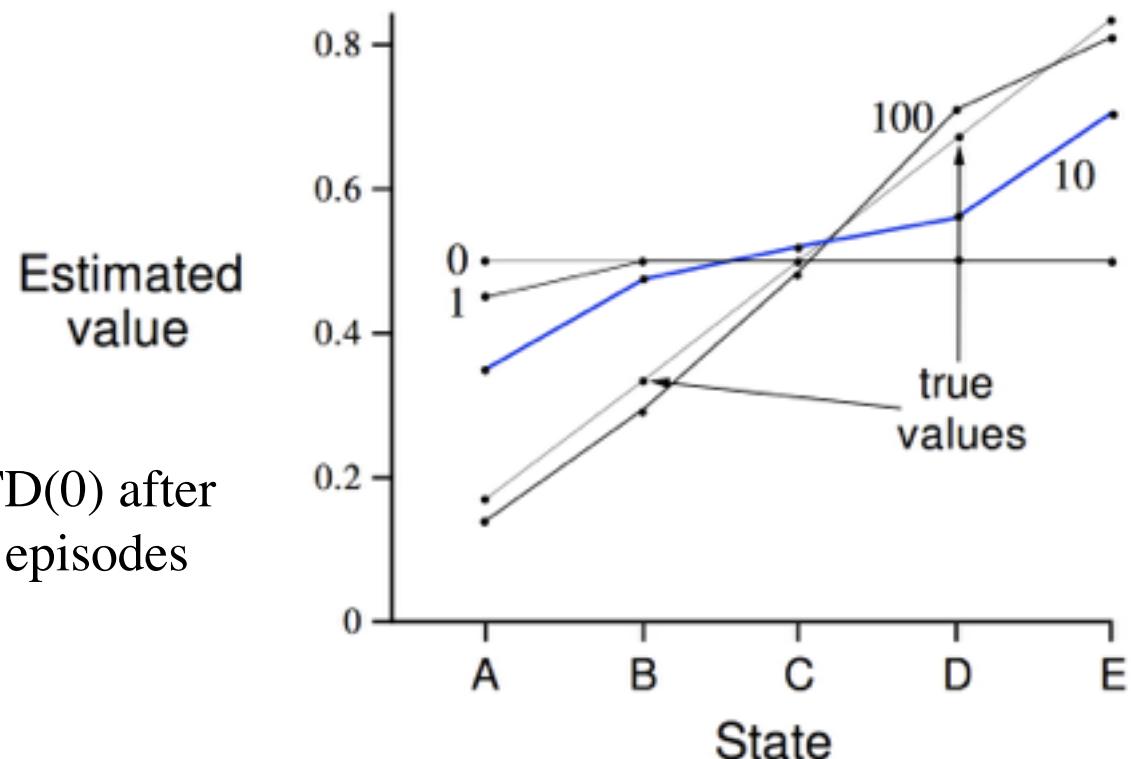
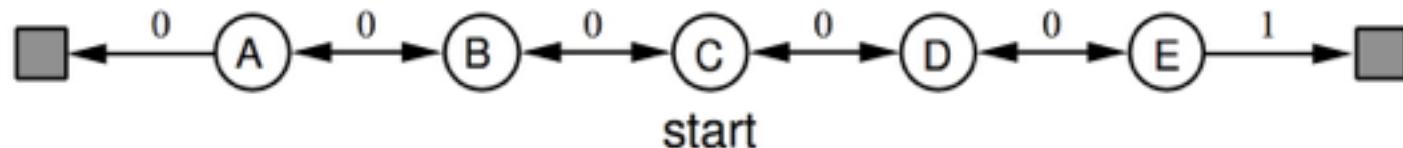
Changes recommended
by TD methods ($\alpha=1$)



Advantages of TD Learning

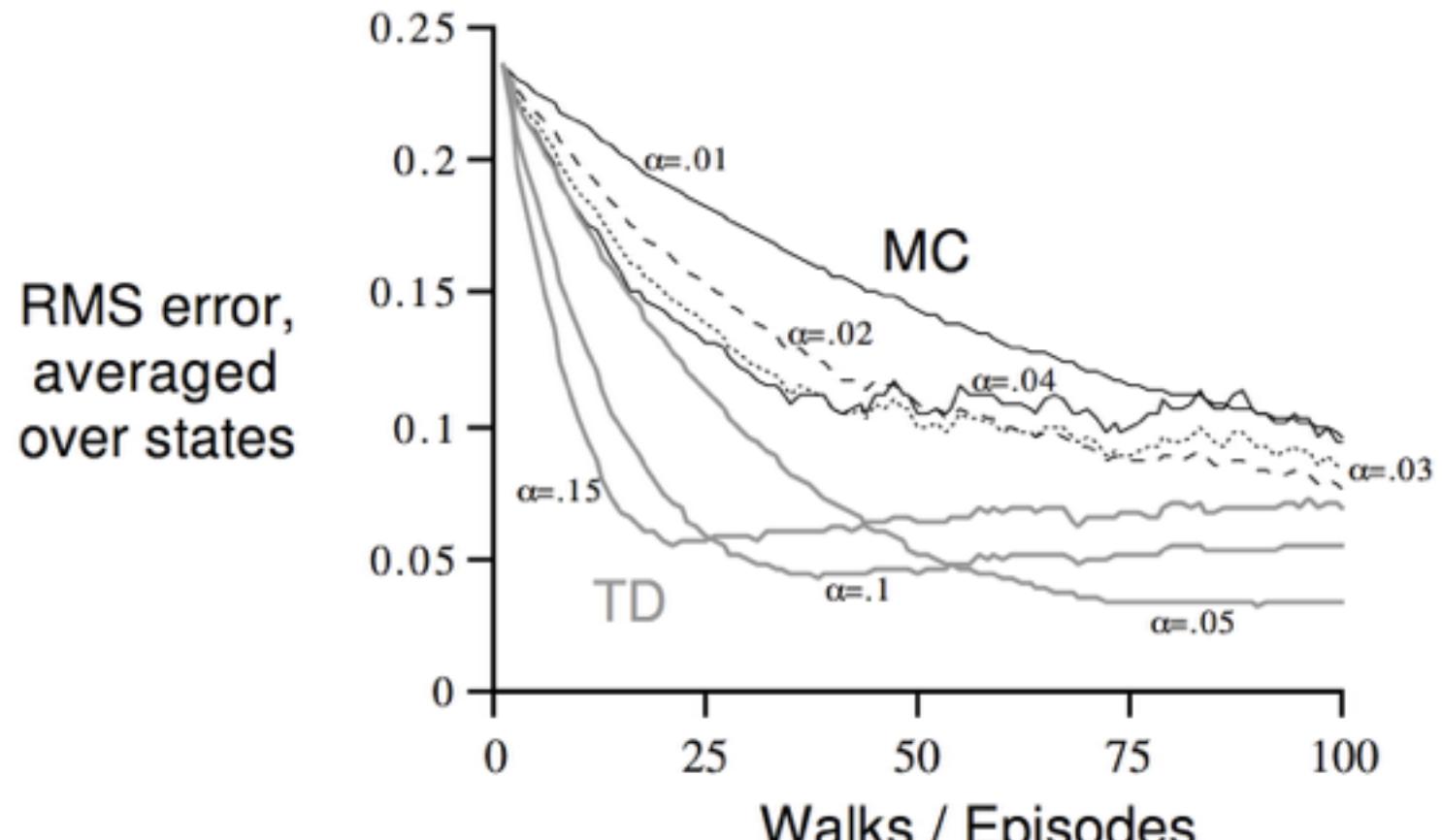
- TD methods do not require a model of the environment, only experience
- TD, but not MC, methods can be fully incremental
 - You can learn **before** knowing the final outcome
 - Less memory
 - Less peak computation
 - You can learn **without** the final outcome
 - From incomplete sequences
- Both MC and TD converge (under certain assumptions to be detailed later), but which is faster?

Random Walk Example



Values learned by TD(0) after various numbers of episodes

TD and MC on the Random Walk



Data averaged over
100 sequences of episodes

Batch Updating in TD and MC methods

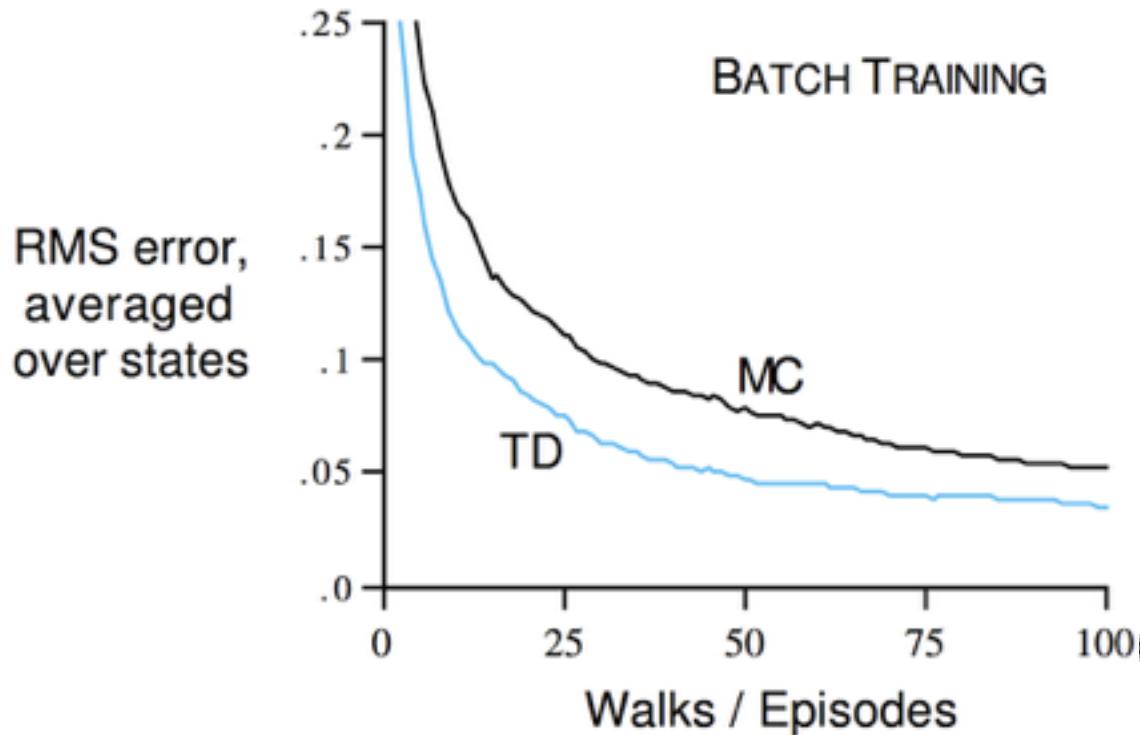
Batch Updating: train completely on a finite amount of data,
e.g., train repeatedly on 10 episodes until convergence.

Compute updates according to TD(0), but only update
estimates after each complete pass through the data.

For any finite Markov prediction task, under batch updating,
TD(0) converges for sufficiently small α .

Constant- α MC also converges under these conditions, **but to a difference answer!**

Random Walk under Batch Updating



After each new episode, all previous episodes were treated as a batch, and algorithm was trained until convergence. All repeated 100 times.

You are the Predictor

Suppose you observe the following 8 episodes:

A, 0, B, 0

B, 1

B, 1

$V(B)? \quad 0.75$

B, 1

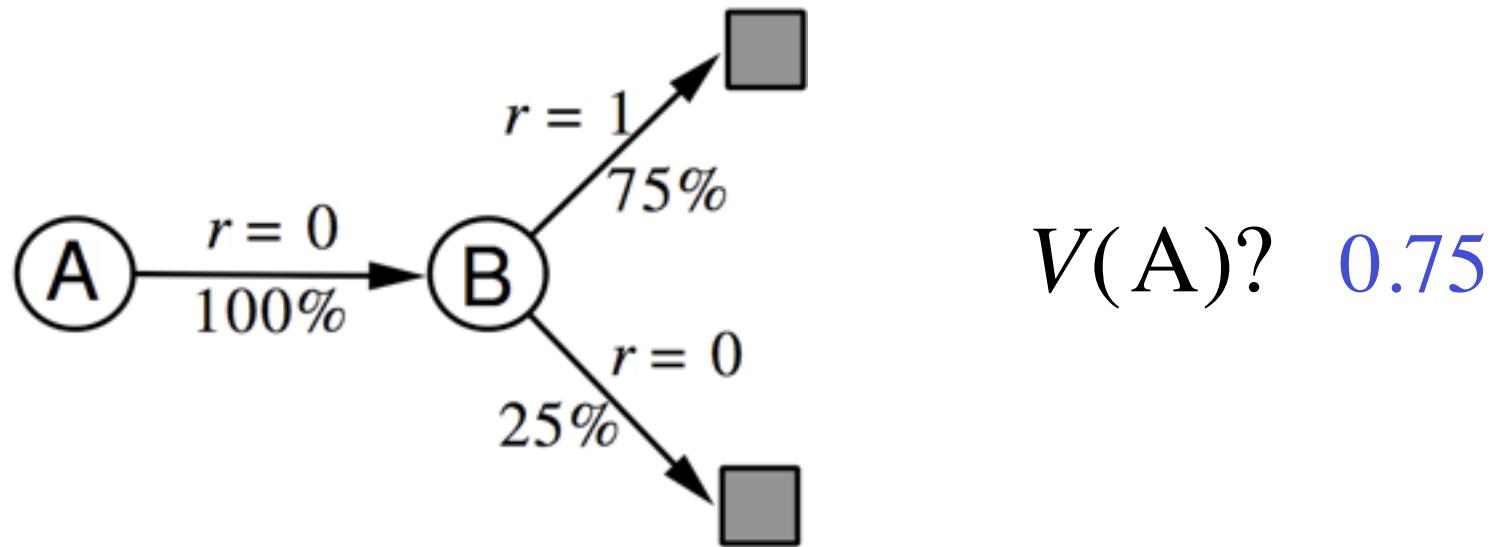
$V(A)? \quad 0?$

B, 1

B, 1

B, 0

You are the Predictor



You are the Predictor

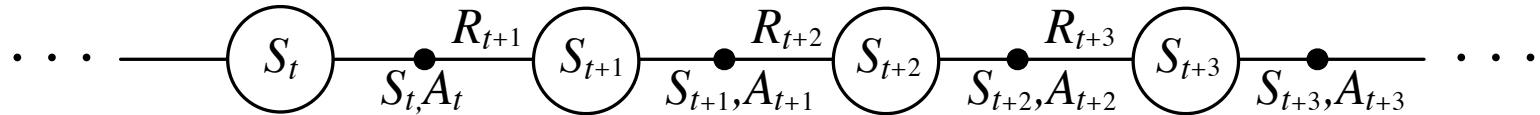
- The prediction that best matches the training data is $V(A)=0$
 - This **minimizes the mean-square-error** on the training set
 - This is what a batch Monte Carlo method gets
- If we consider the sequentiality of the problem, then we would set $V(A)=.75$
 - This is correct for the **maximum likelihood** estimate of a Markov model generating the data
 - i.e., if we do a best fit Markov model, and assume it is exactly correct, and then compute what it predicts (how?)
 - This is called the **certainty-equivalence estimate**
 - This is what TD(0) gets

Summary (so far)

- Introduced *one-step tabular model-free TD methods*
- These methods bootstrap and sample, combining aspects of DP and MC methods
- TD prediction
- If the world is truly Markov, then TD methods will learn faster than MC methods
- MC methods have lower error on past data, but higher error on future data
- Extend prediction to control by employing some form of GPI
 - On-policy control: Sarsa
 - Off-policy control: Q-learning

Learning An Action-Value Function

Estimate q_π for the current policy π



After every transition from a nonterminal state, S_t , do this:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

If S_{t+1} is terminal, then define $Q(S_{t+1}, A_{t+1}) = 0$

Sarsa: On-Policy TD Control

Turn this into a control method by always updating the policy to be greedy with respect to the current estimate:

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

 Initialize S

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Repeat (for each step of episode):

 Take action A , observe R, S'

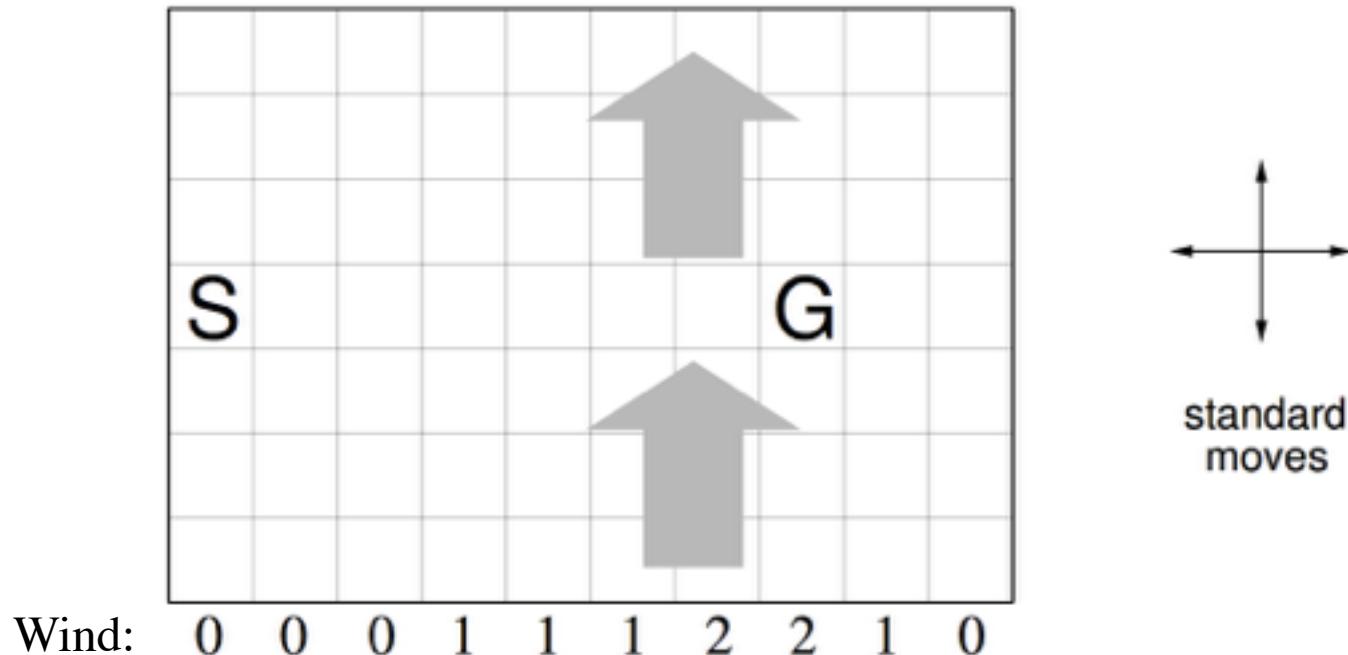
 Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$$

$$S \leftarrow S'; A \leftarrow A';$$

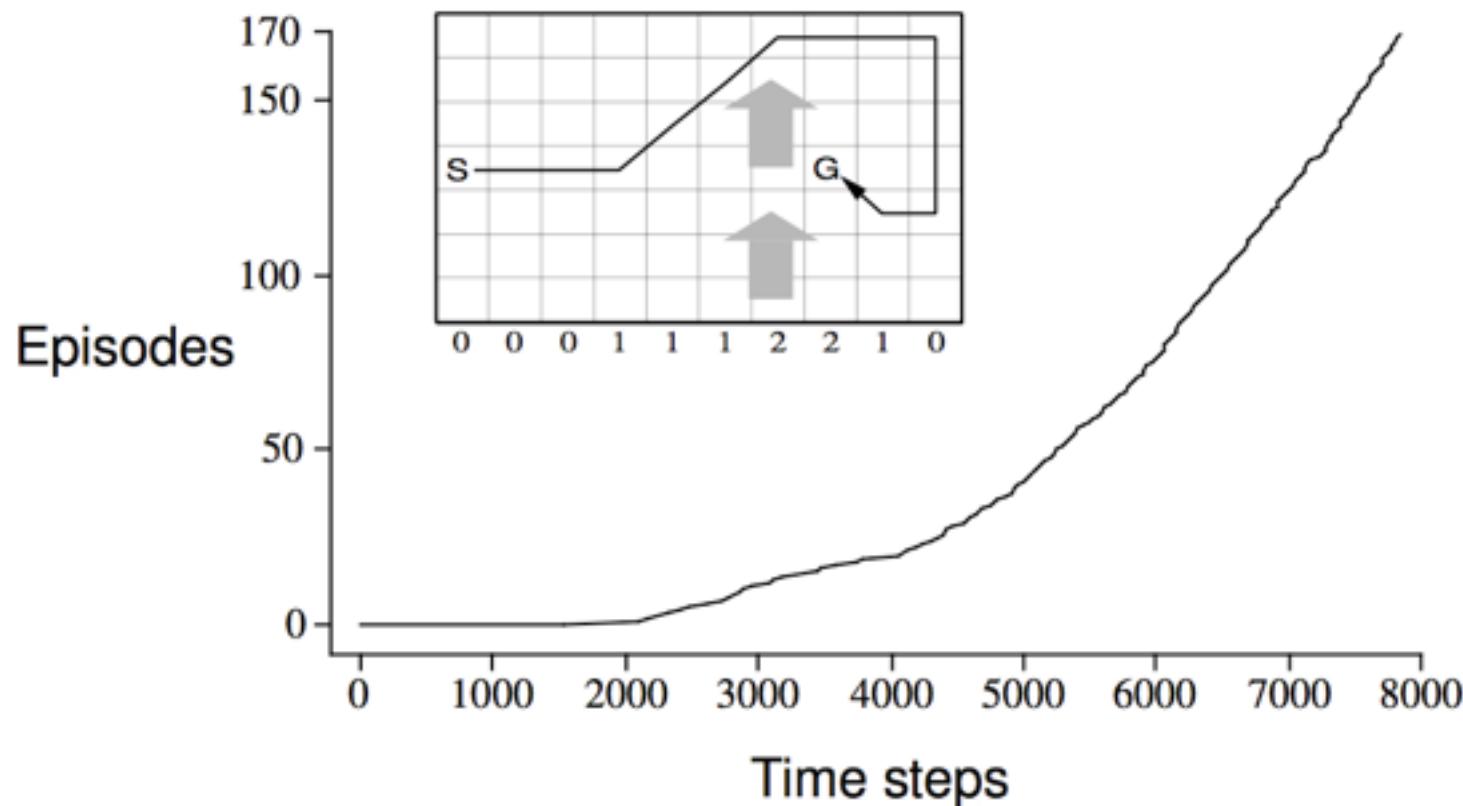
 until S is terminal

Windy Gridworld



undiscounted, episodic, reward = -1 until goal

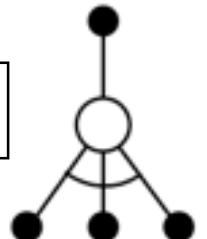
Results of Sarsa on the Windy Gridworld



Q-Learning: Off-Policy TD Control

One-step Q-learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$



Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

 Initialize S

 Repeat (for each step of episode):

 Choose A from S using policy derived from Q (e.g., ε -greedy)

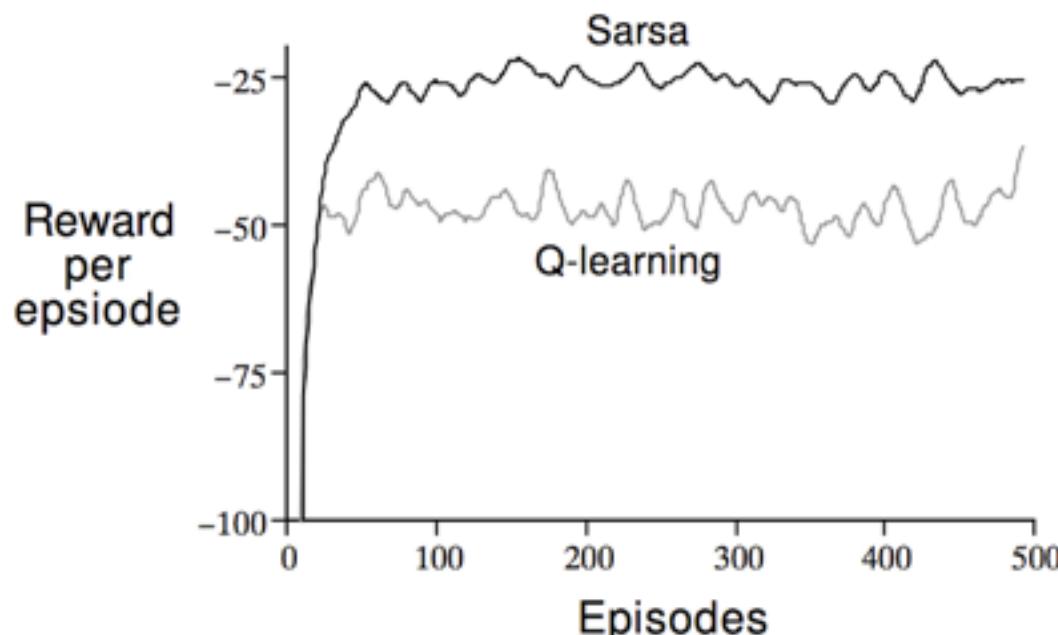
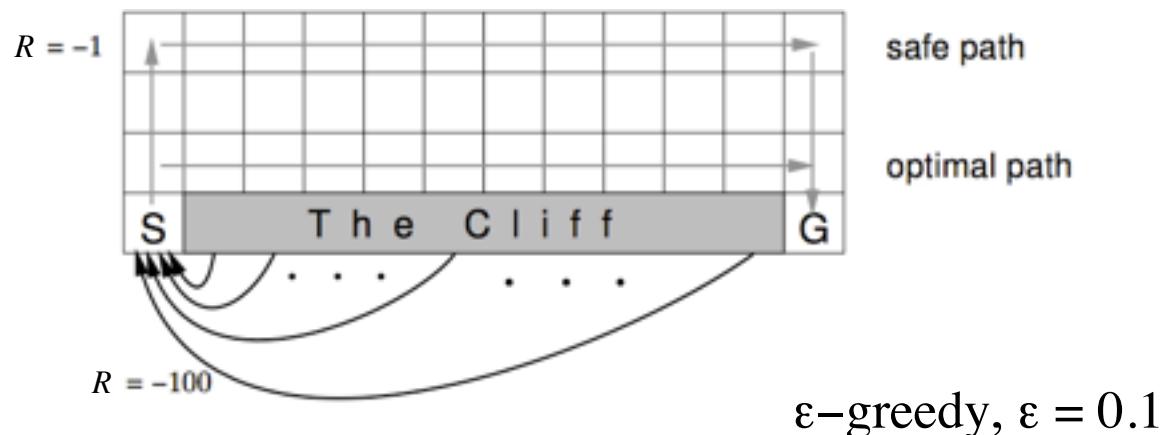
 Take action A , observe R, S'

$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$;

 until S is terminal

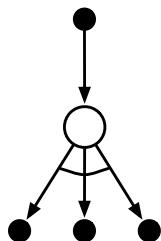
Cliffwalking



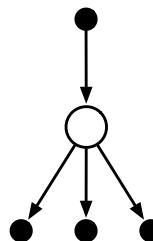
Expected Sarsa

- Instead of the *sample* value-of-next-state, use the expectation!

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t) \right] \\ &\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a) - Q(S_t, A_t) \right] \end{aligned}$$



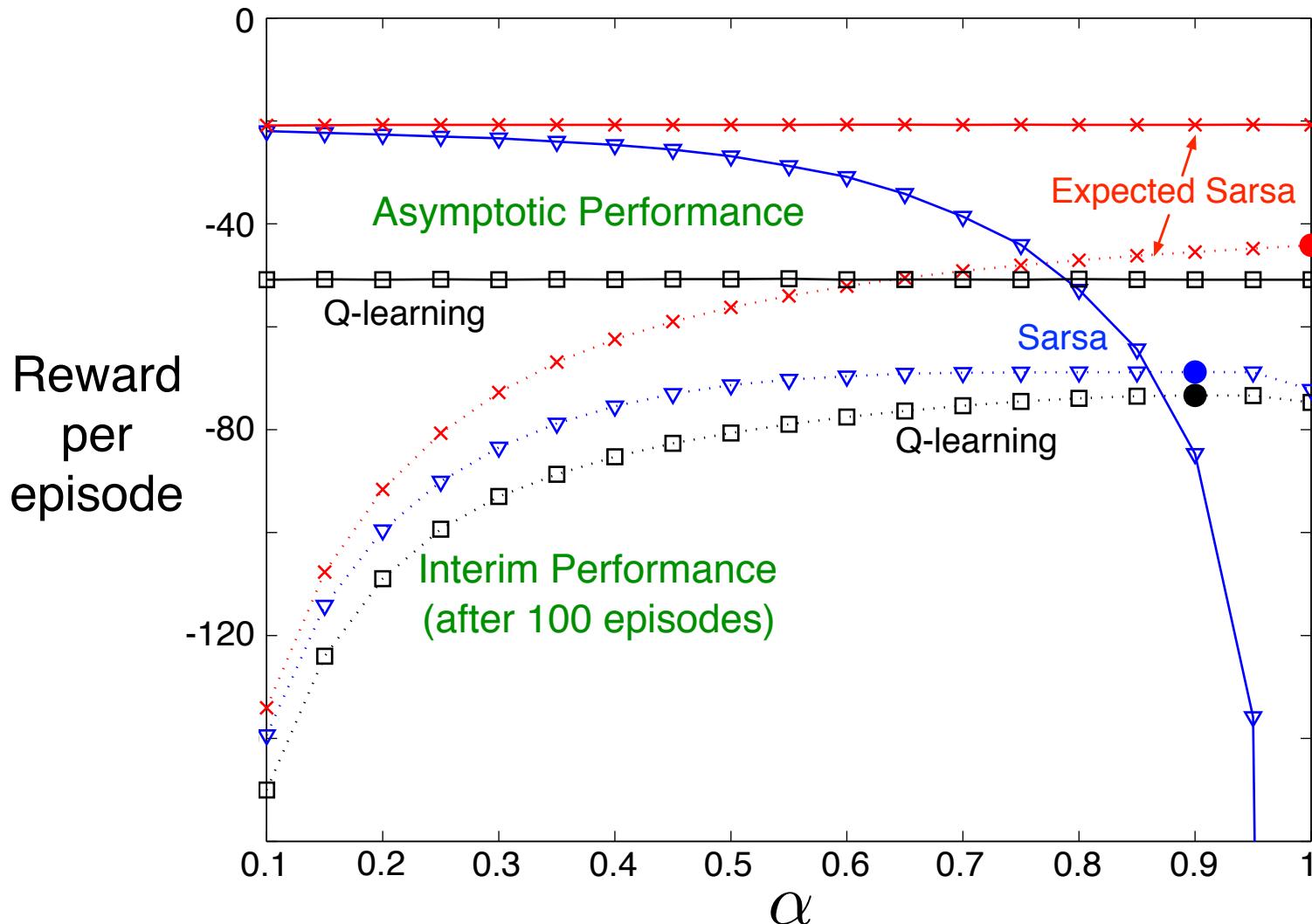
Q-learning



Expected Sarsa

- Expected Sarsa's performs better than Sarsa (but costs more)

Performance on the Cliff-walking Task

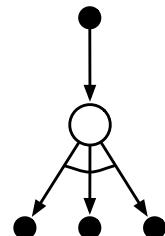


Off-policy Expected Sarsa

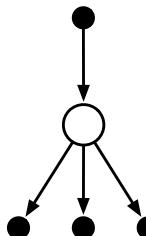
- Expected Sarsa generalizes to arbitrary behavior policies μ
 - in which case it includes Q-learning as the special case in which π is the greedy policy

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t) \right] \\ &\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a) - Q(S_t, A_t) \right] \end{aligned}$$

Nothing
changes
here



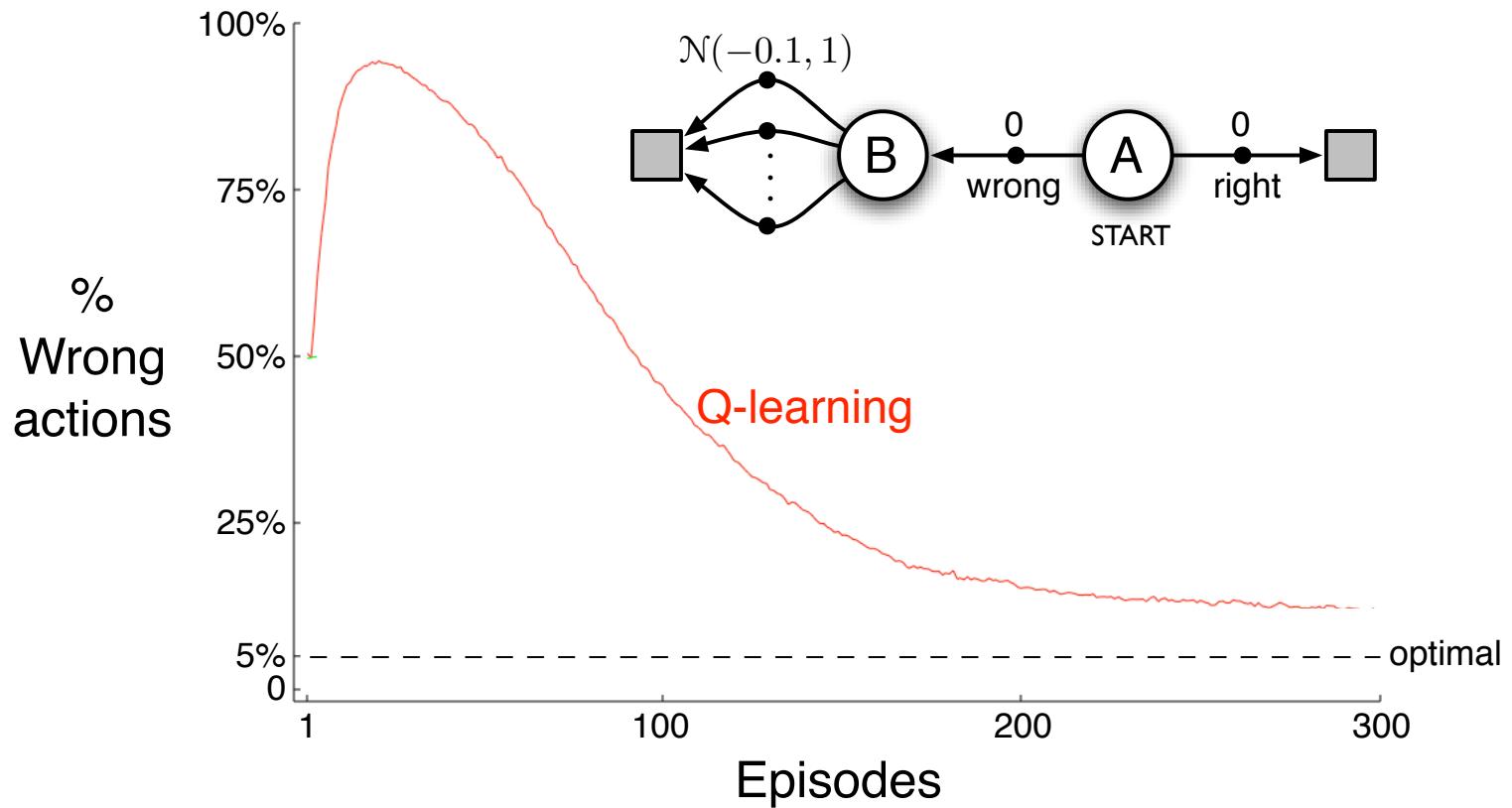
Q-learning



Expected Sarsa

- This idea seems to be new

Maximization Bias Example



Tabular Q-learning:
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

Double Q-Learning

- Train 2 action-value functions, Q_1 and Q_2
- Do Q-learning on both, but
 - never on the same time steps (Q_1 and Q_2 are indep.)
 - pick Q_1 or Q_2 at random to be updated on each step
- If updating Q_1 , use Q_2 for the value of the next state:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left(R_{t+1} + Q_2\left(S_{t+1}, \arg \max_a Q_1(S_{t+1}, a)\right) - Q_1(S_t, A_t) \right)$$

- Action selections are (say) ε -greedy with respect to the sum of Q_1 and Q_2

Double Q-Learning

Initialize $Q_1(s, a)$ and $Q_2(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily

Initialize $Q_1(\text{terminal-state}, \cdot) = Q_2(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

 Initialize S

 Repeat (for each step of episode):

 Choose A from S using policy derived from Q_1 and Q_2 (e.g., ε -greedy in $Q_1 + Q_2$)

 Take action A , observe R, S'

 With 0.5 probability:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha \left(R + \gamma Q_2(S', \arg\max_a Q_1(S', a)) - Q_1(S, A) \right)$$

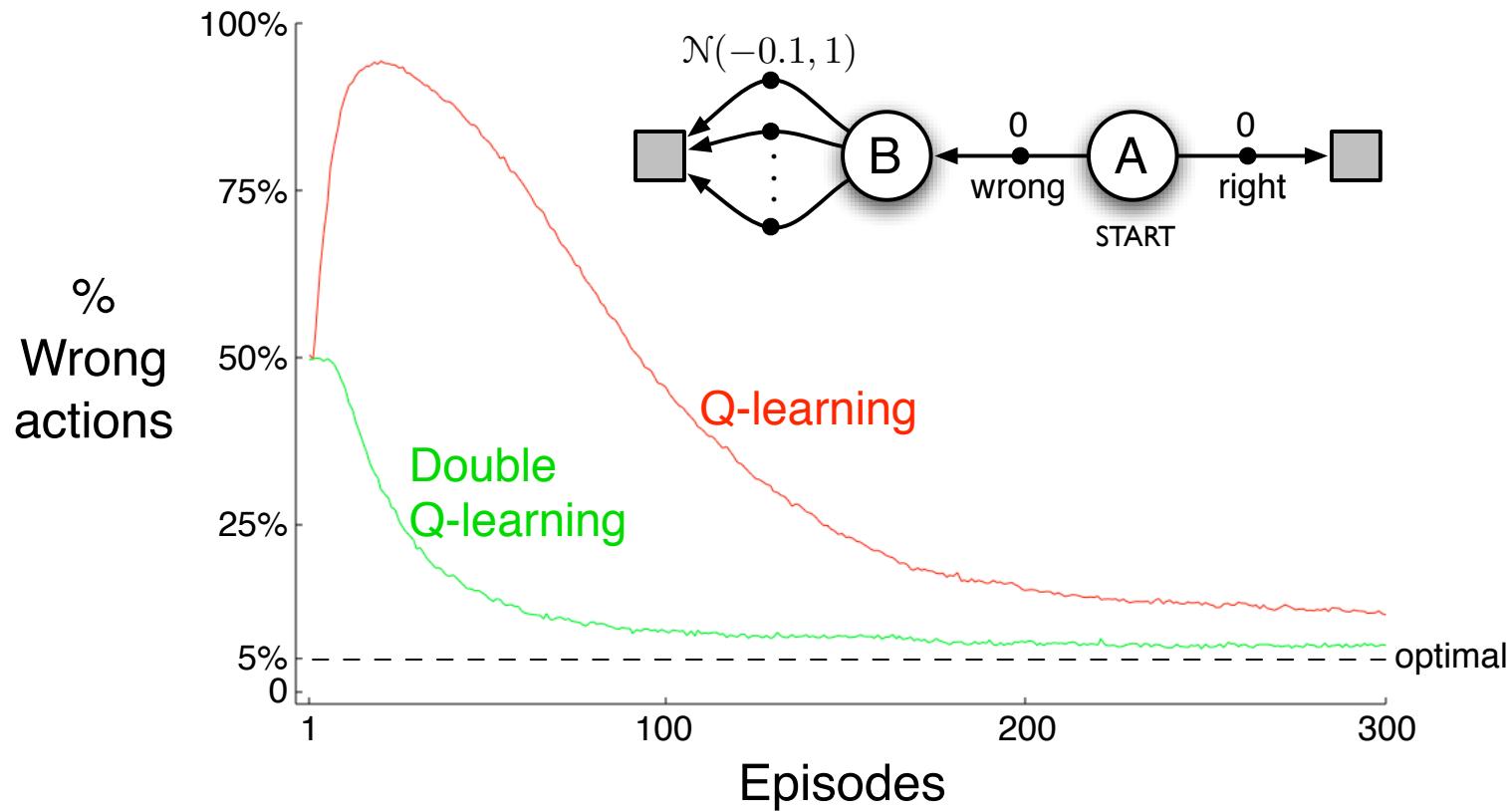
 else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha \left(R + \gamma Q_1(S', \arg\max_a Q_2(S', a)) - Q_2(S, A) \right)$$

$S \leftarrow S'$;

 until S is terminal

Example of Maximization Bias

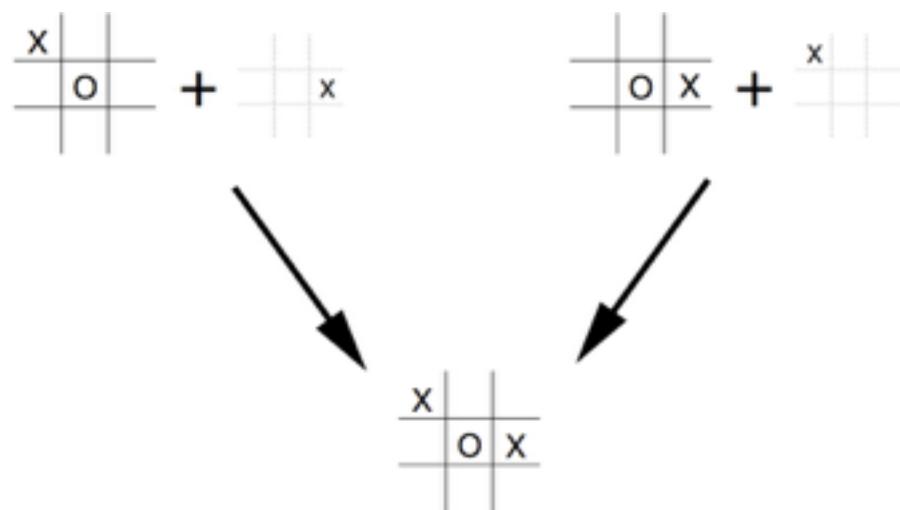


Double Q-learning:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left(R_{t+1} + Q_2(S_{t+1}, \arg \max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right)$$

Afterstates

- Usually, a state-value function evaluates states in which the agent can take an action.
- But sometimes it is useful to evaluate states **after** agent has acted, as in tic-tac-toe.
- Why is this useful?

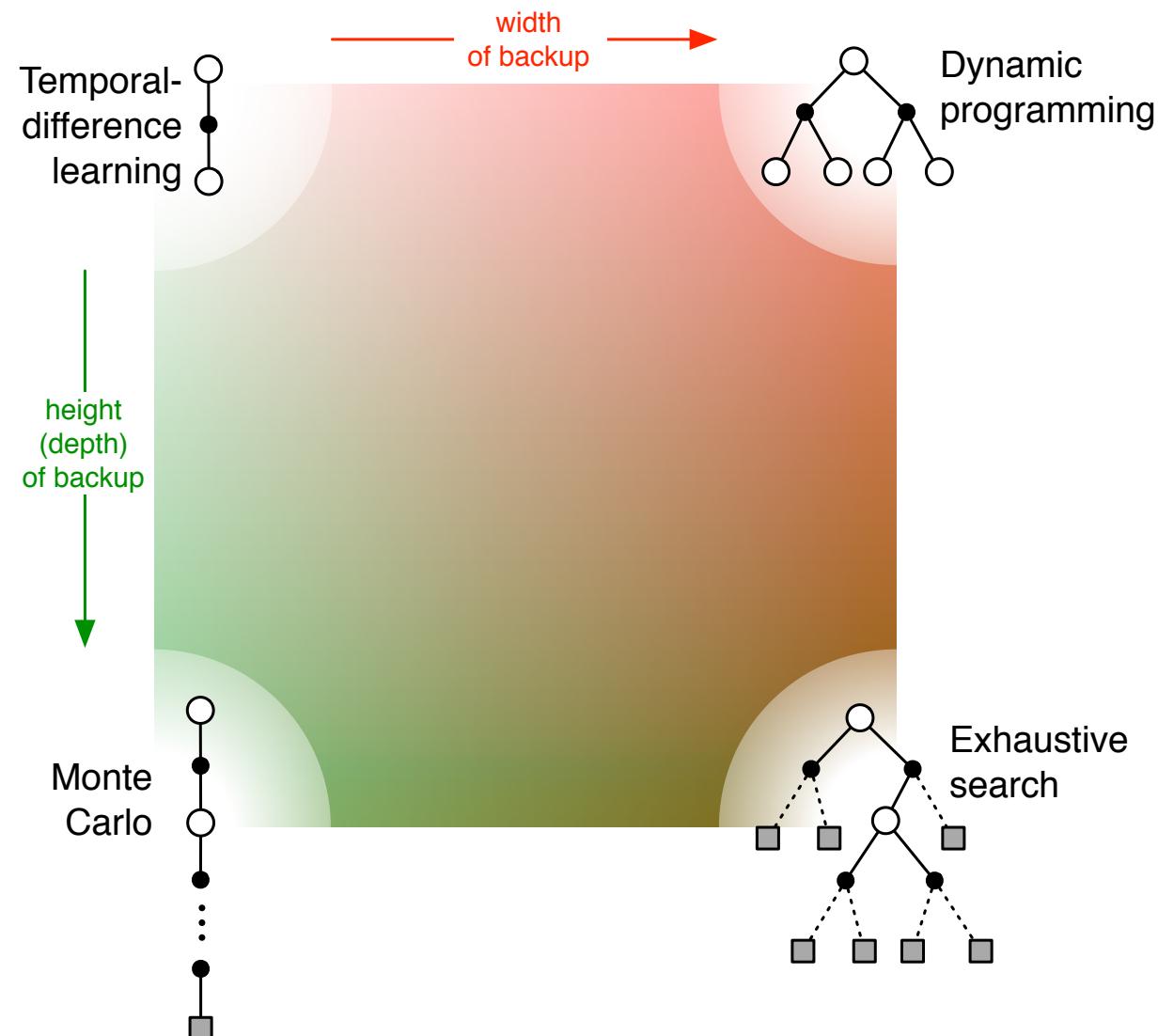


- What is this in general?

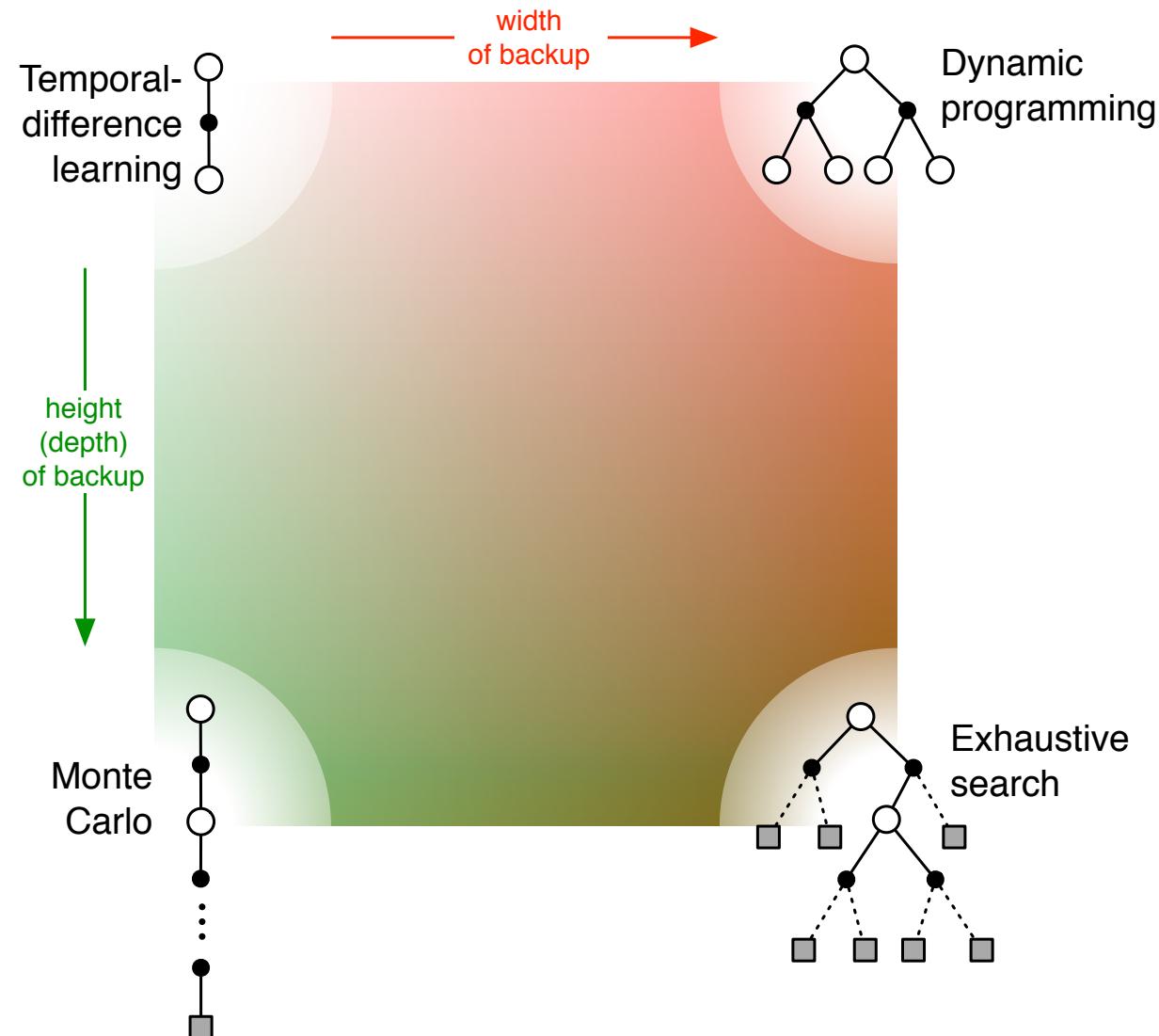
Summary

- Introduced *one-step tabular model-free TD methods*
- These methods bootstrap and sample, combining aspects of DP and MC methods
- TD prediction
- If the world is truly Markov, then TD methods will learn faster than MC methods
- MC methods have lower error on past data, but higher error on future data
- Extend prediction to control by employing some form of GPI
 - On-policy control: *Sarsa, Expected Sarsa*
 - Off-policy control: *Q-learning, Expected Sarsa*

Unified View



Unified View

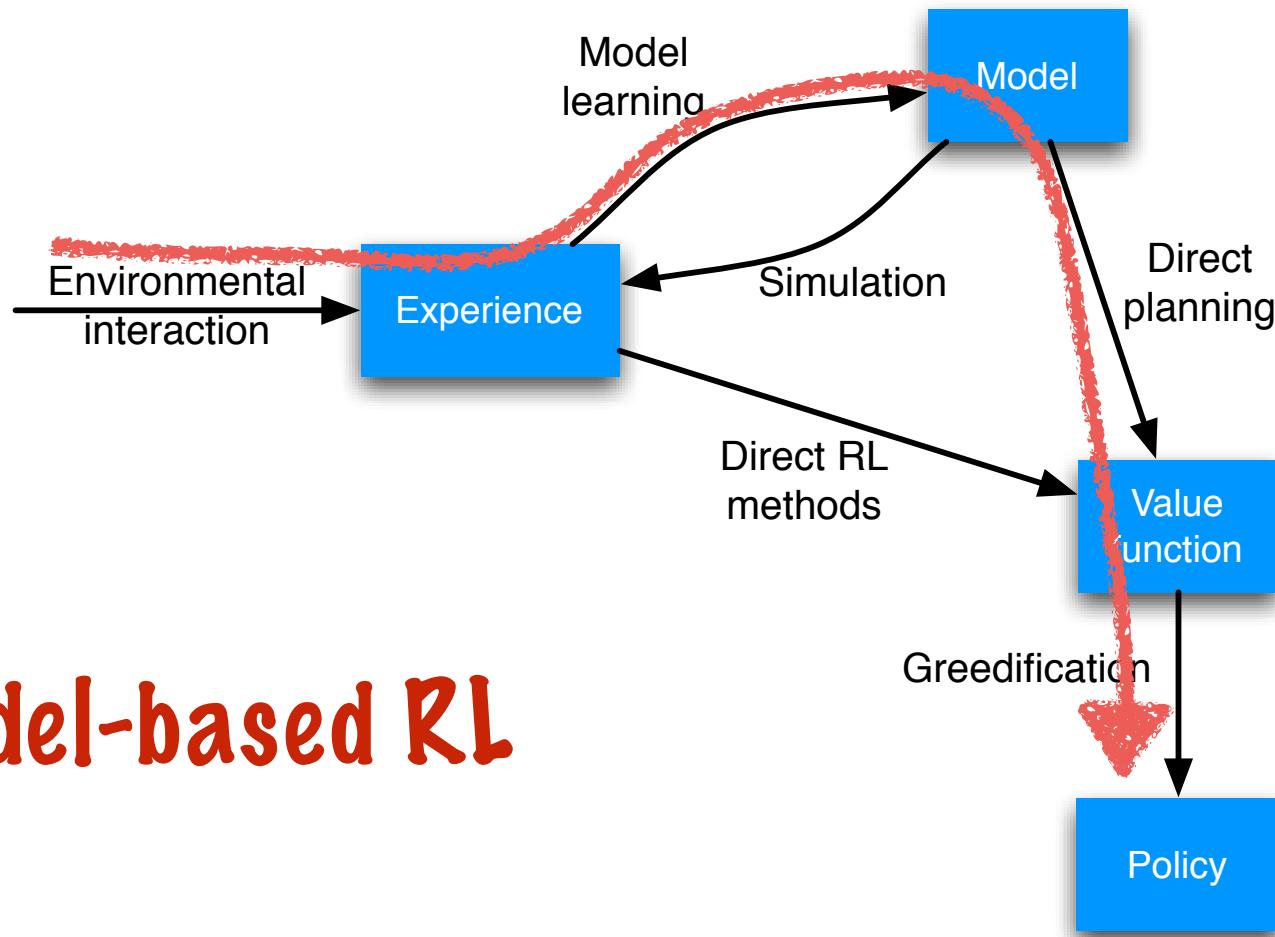


Chapter 8: Planning and Learning

Objectives of this chapter:

- To think more generally about uses of environment models
- Integration of (unifying) planning, learning, and execution
- “Model-based reinforcement learning”

Paths to a policy

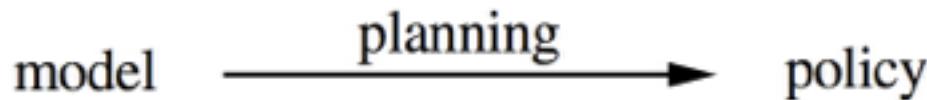


Models

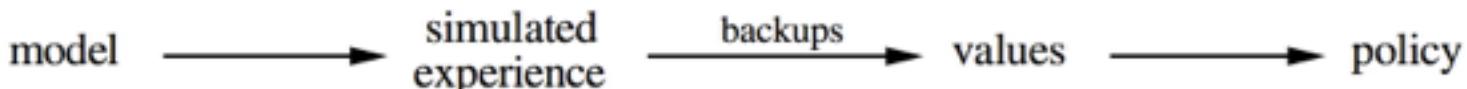
- **Model**: anything the agent can use to predict how the environment will respond to its actions
- **Distribution model**: description of all possibilities and their probabilities
 - e.g., $\hat{p}(s', r \mid s, a)$ for all s, a, s', r
- **Sample model**, a.k.a. a simulation model
 - produces sample experiences for given s, a
 - allows reset, exploring starts
 - often much easier to come by
- Both types of models can be used to produce **hypothetical experience**

Planning

- **Planning:** any computational process that uses a model to create or improve a policy



- Planning in AI:
 - state-space planning
 - plan-space planning (e.g., partial-order planner)
- We take the following (unusual) view:
 - all state-space planning methods involve computing value functions, either explicitly or implicitly
 - they all apply backups to simulated experience



Planning Cont.

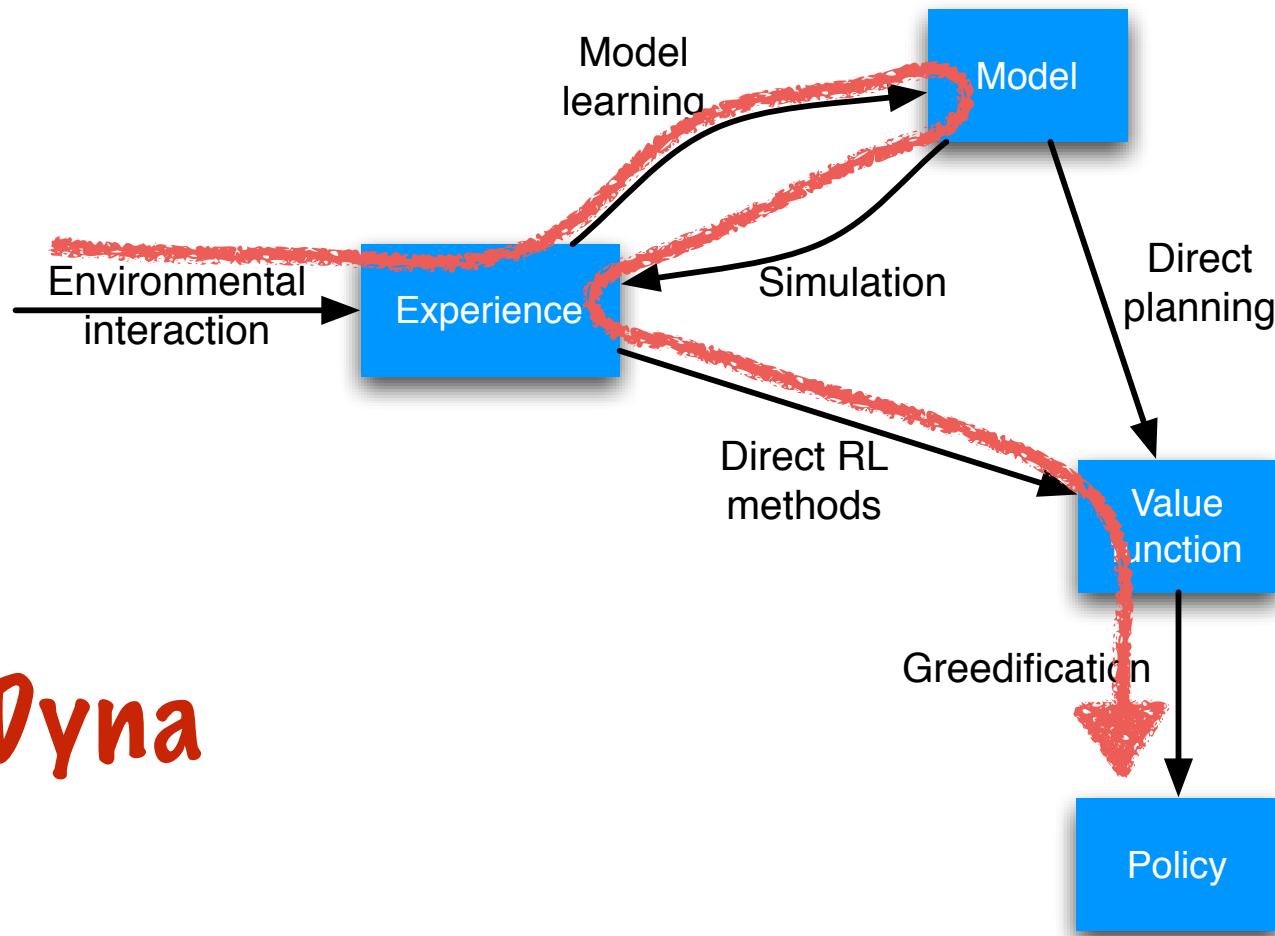
- Classical DP methods are state-space planning methods
- Heuristic search methods are state-space planning methods
- A planning method based on Q-learning:

Do forever:

1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(s)$, at random
2. Send S, A to a sample model, and obtain
a sample next reward, R , and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S' :
$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$$

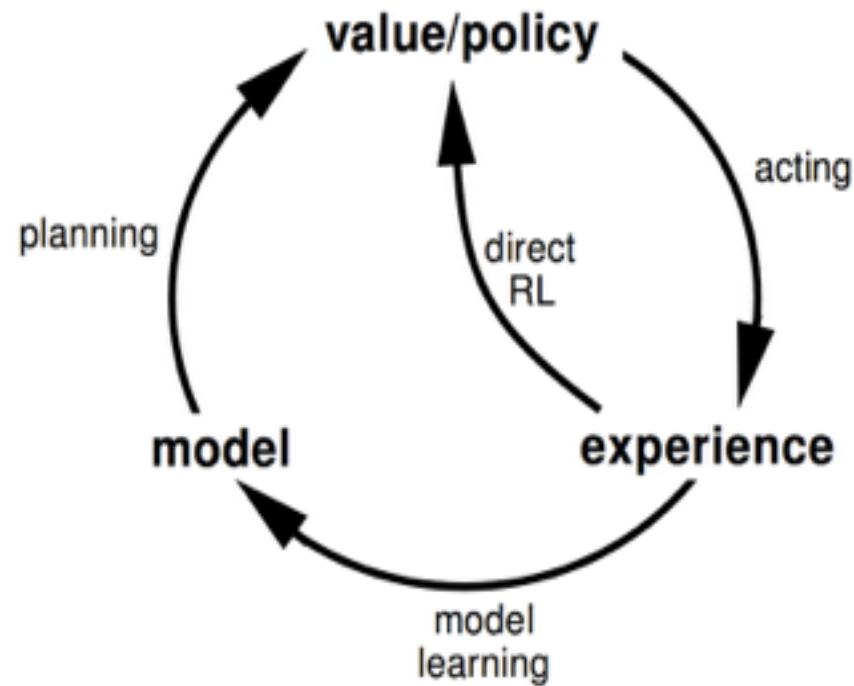
Random-Sample One-Step Tabular Q-Planning

Paths to a policy



Learning, Planning, and Acting

- Two uses of real experience:
 - **model learning**: to improve the model
 - **direct RL**: to directly improve the value function and policy
- Improving value function and/or policy via a model is sometimes called **indirect RL**. Here, we call it **planning**.



Direct (model-free) vs. Indirect (model-based) RL

- Direct methods

- simpler
- not affected by bad models

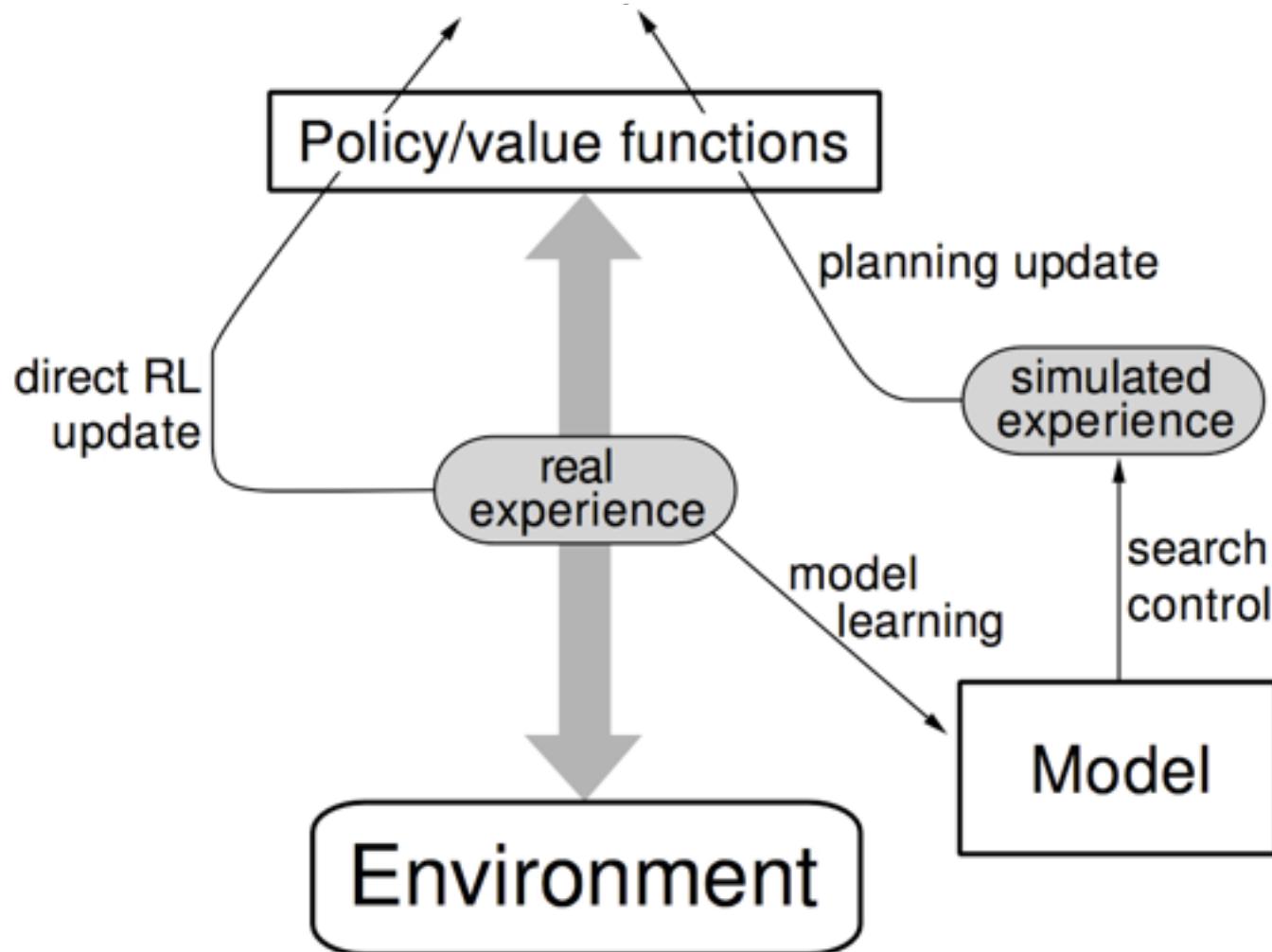
- Indirect methods:

- make fuller use of experience: get better policy with fewer environment interactions

But they are very closely related and can be usefully combined:

planning, acting, model learning, and direct RL can occur simultaneously and in parallel

The Dyna Architecture

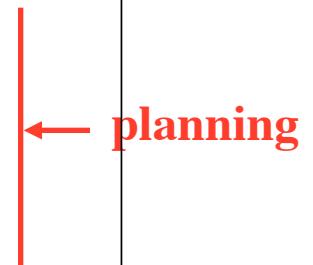


The Dyna-Q Algorithm

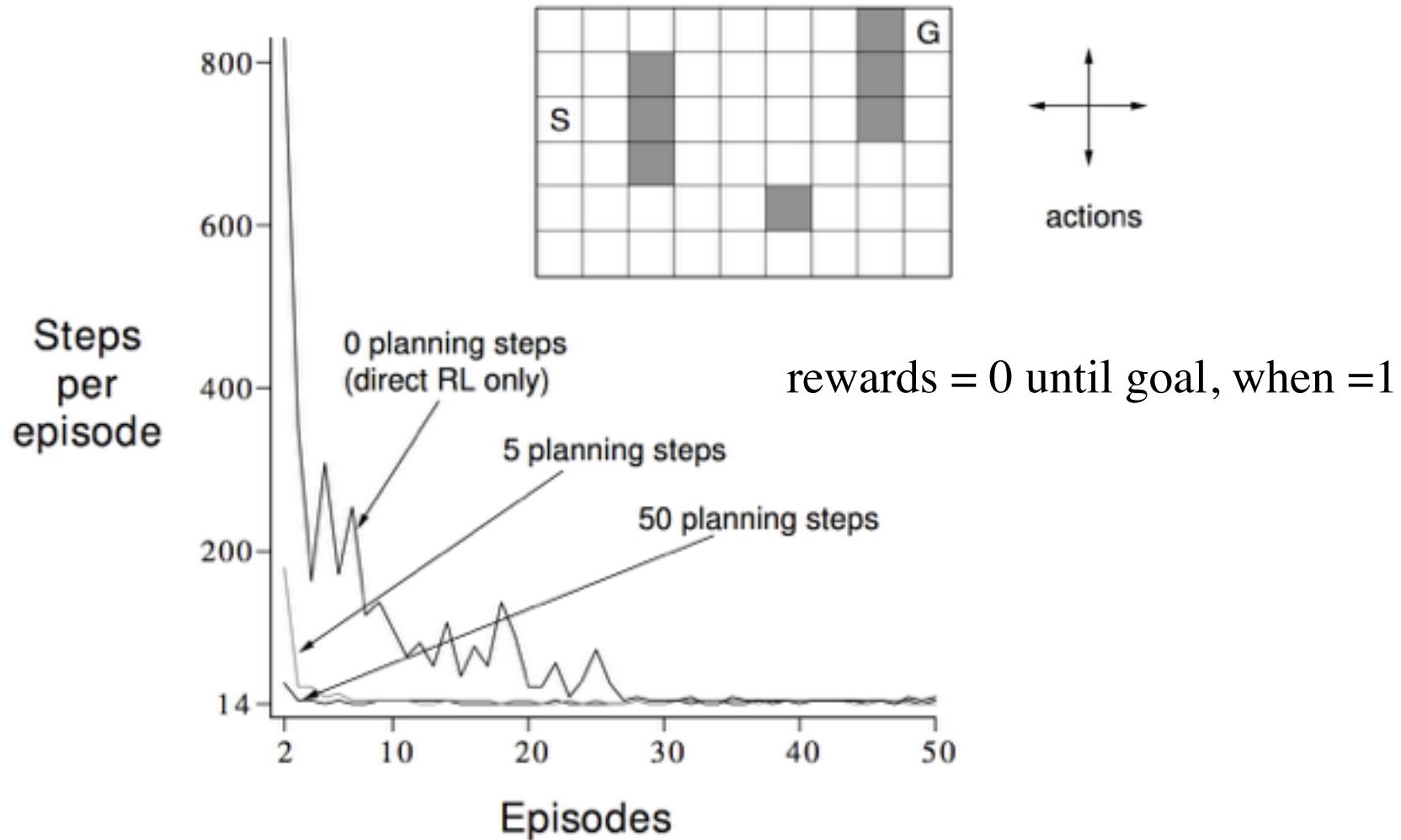
Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Do forever:

- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon\text{-greedy}(S, Q)$
- (c) Execute action A ; observe resultant reward, R , and state, S'
- (d) $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$ ← direct RL
- (e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment) ← model learning
- (f) Repeat n times:
 $S \leftarrow$ random previously observed state
 $A \leftarrow$ random action previously taken in S
 $R, S' \leftarrow Model(S, A)$
 $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

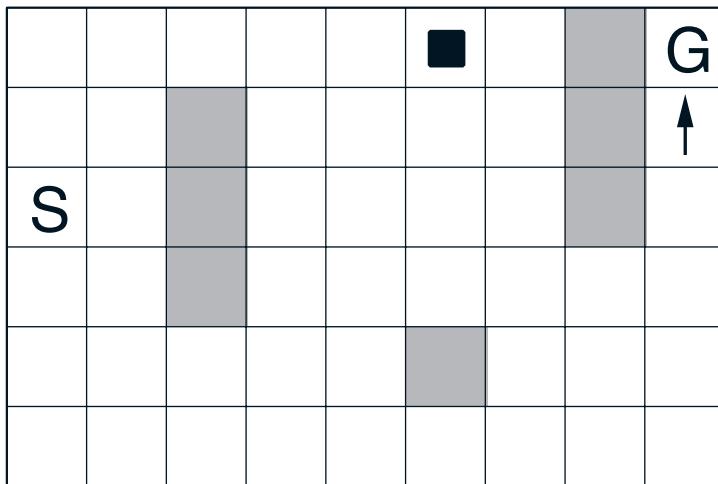


Dyna-Q on a Simple Maze

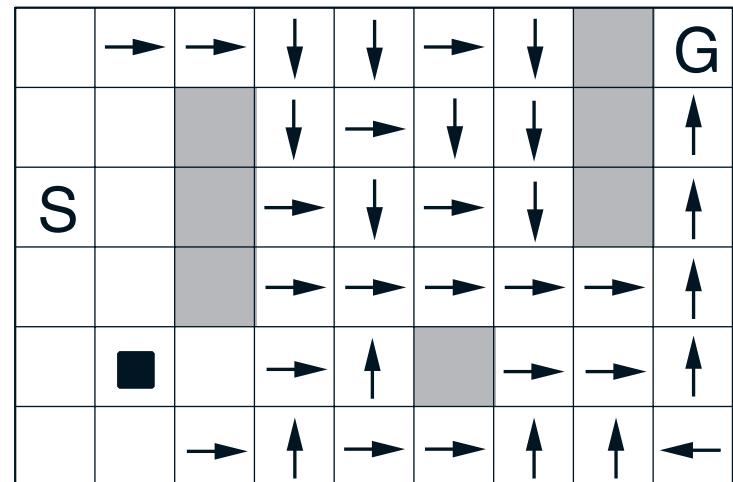


Dyna-Q Snapshots: Midway in 2nd Episode

WITHOUT PLANNING ($n=0$)

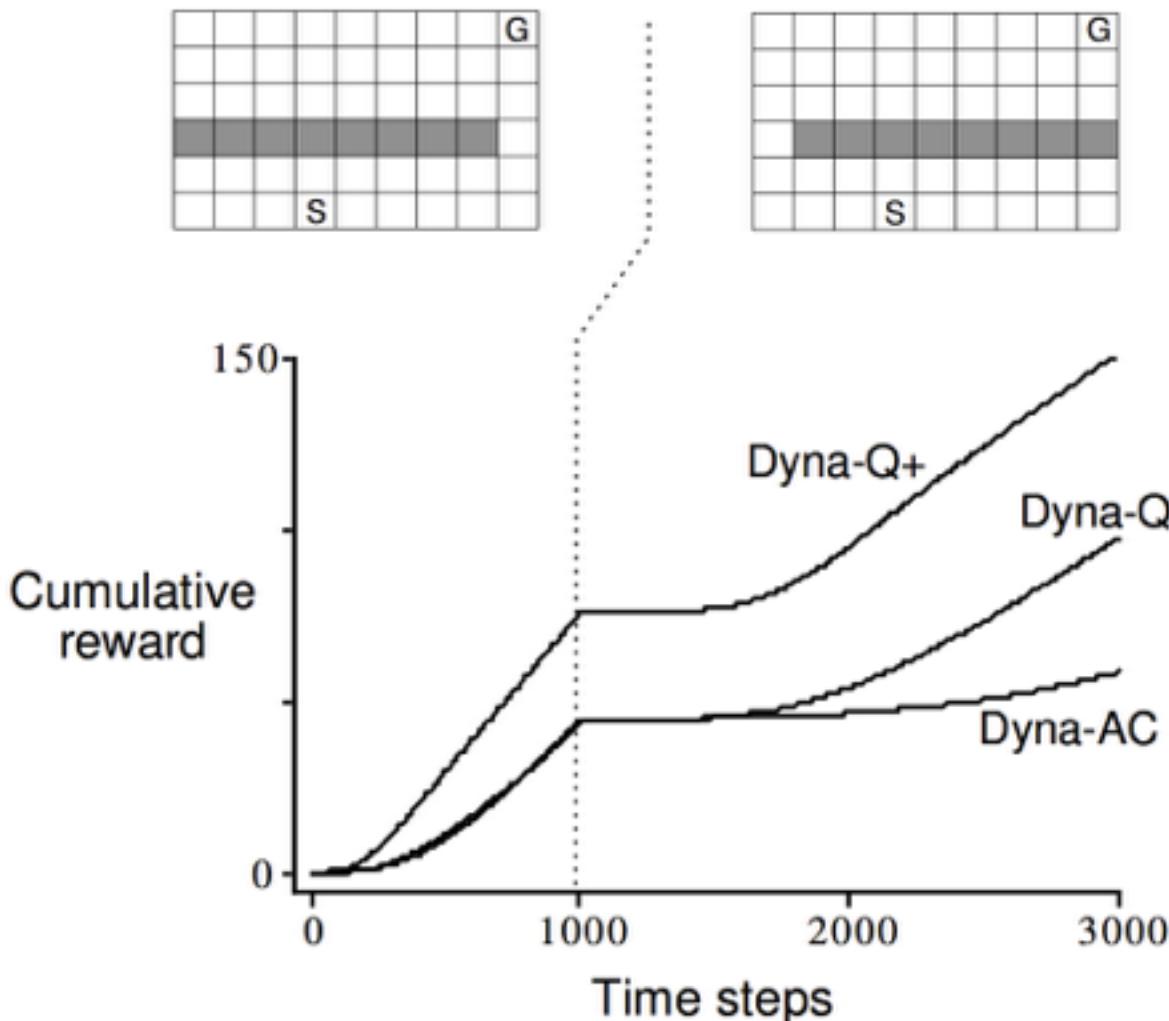


WITH PLANNING ($n=50$)



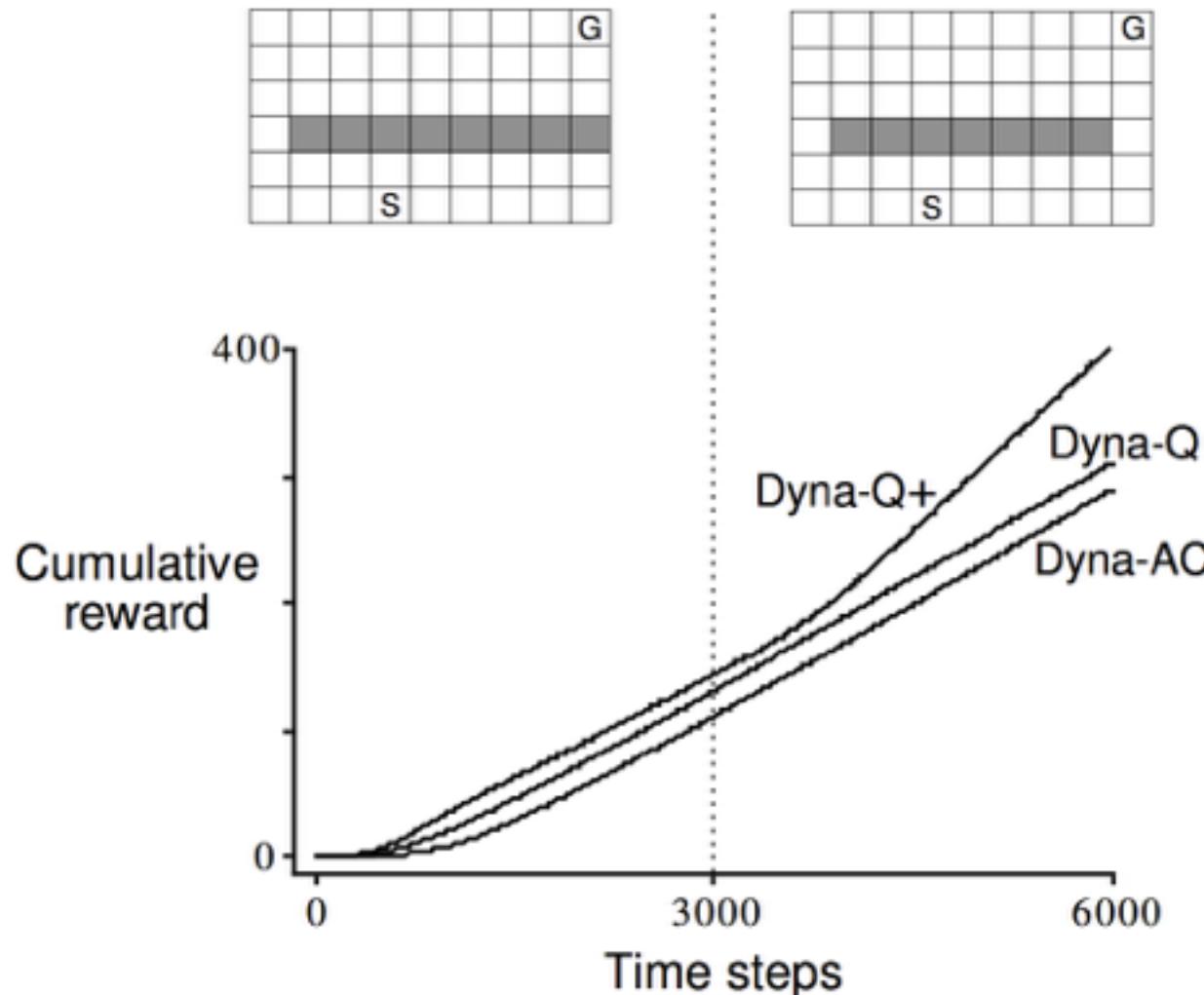
When the Model is Wrong: Blocking Maze

The changed environment is harder



When the Model is Wrong: Shortcut Maze

The changed environment is easier



What is Dyna-Q+?

- Uses an “exploration bonus”:
 - Keeps track of time since each state-action pair was tried for real
 - An extra reward is added for transitions caused by state-action pairs related to how long ago they were tried: the longer unvisited, the more reward for visiting

$$R + \kappa \sqrt{\tau}$$

time since last visiting
the state-action pair

- The agent actually “plans” how to visit long unvisited states

Prioritized Sweeping

- Which states or state-action pairs should be generated during planning?
- Work backwards from states whose values have just changed:
 - Maintain a queue of state-action pairs whose values would change a lot if backed up, prioritized by the size of the change
 - When a new backup occurs, insert predecessors according to their priorities
 - Always perform backups from first in queue
- Moore & Atkeson 1993; Peng & Williams 1993
- improved by McMahan & Gordon 2005; Van Seijen 2013

Prioritized Sweeping

Initialize $Q(s, a)$, $Model(s, a)$, for all s, a , and $PQueue$ to empty

Do forever:

- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow policy(S, Q)$
- (c) Execute action A ; observe resultant reward, R , and state, S'
- (d) $Model(S, A) \leftarrow R, S'$
- (e) $P \leftarrow |R + \gamma \max_a Q(S', a) - Q(S, A)|.$
- (f) if $P > \theta$, then insert S, A into $PQueue$ with priority P
- (g) Repeat n times, while $PQueue$ is not empty:

$S, A \leftarrow first(PQueue)$

$R, S' \leftarrow Model(S, A)$

$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

Repeat, for all \bar{S}, \bar{A} predicted to lead to S :

$\bar{R} \leftarrow$ predicted reward for \bar{S}, \bar{A}, S

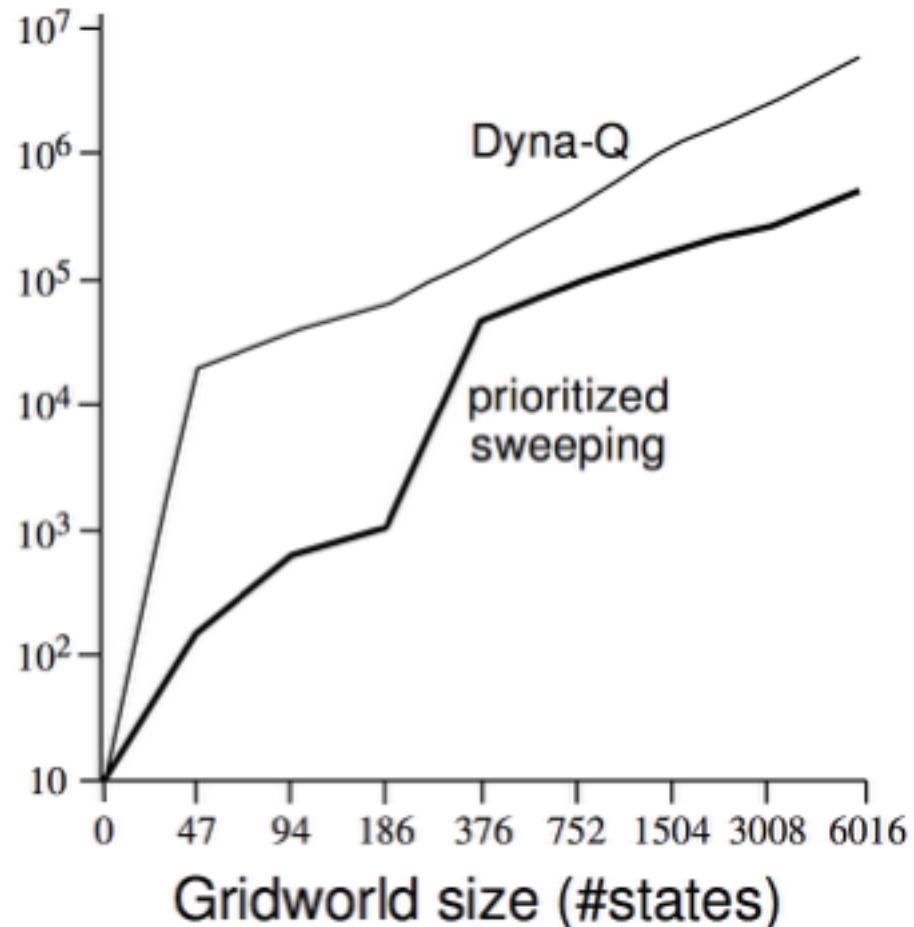
$P \leftarrow |\bar{R} + \gamma \max_a Q(S, a) - Q(\bar{S}, \bar{A})|.$

if $P > \theta$ then insert \bar{S}, \bar{A} into $PQueue$ with priority P

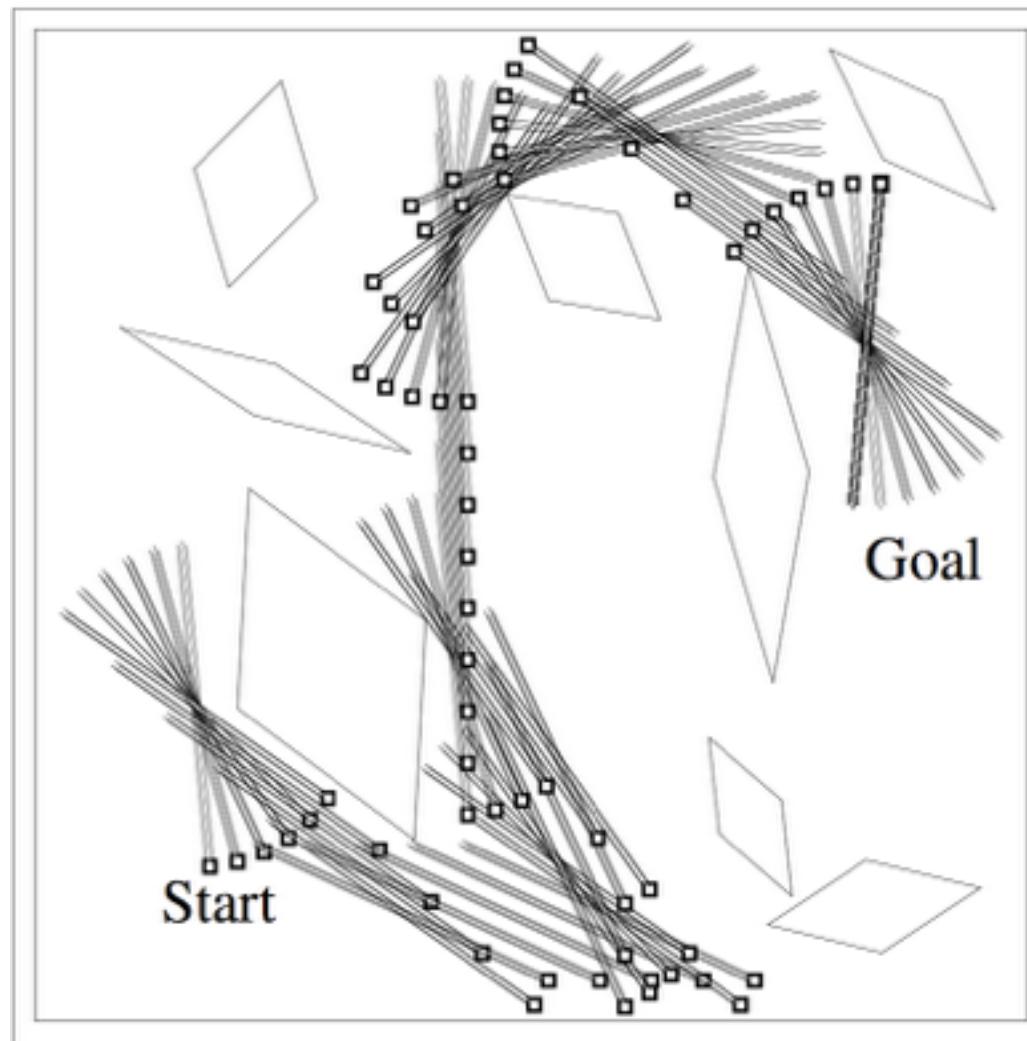
Prioritized Sweeping vs. Dyna-Q

Both use $n=5$ backups per environmental interaction

Backups until optimal solution



Rod Maneuvering (Moore and Atkeson 1993)



Improved Prioritized Sweeping with Small Backups

- Planning is a form of state-space search
 - a massive computation which we want to control to maximize its efficiency
- Prioritized sweeping is a form of search control
 - focusing the computation where it will do the most good
- But can we focus better?
- Can we focus more tightly?
- Small backups are perhaps the smallest unit of search work
 - and thus permit the most flexible allocation of effort

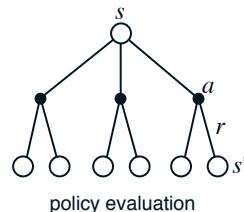
Full and Sample (One-Step) Backups

Value
estimated

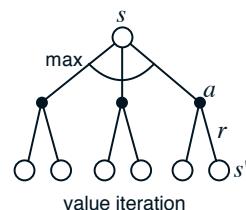
Full backups
(DP)

Sample backups
(one-step TD)

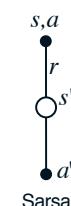
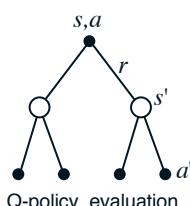
$$\mathcal{V}_\pi(s)$$



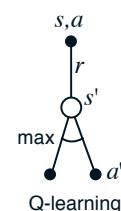
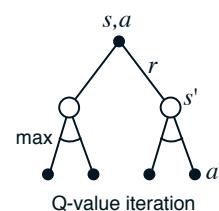
$$\mathcal{V}_*(s)$$



$$q_\pi(a,s)$$

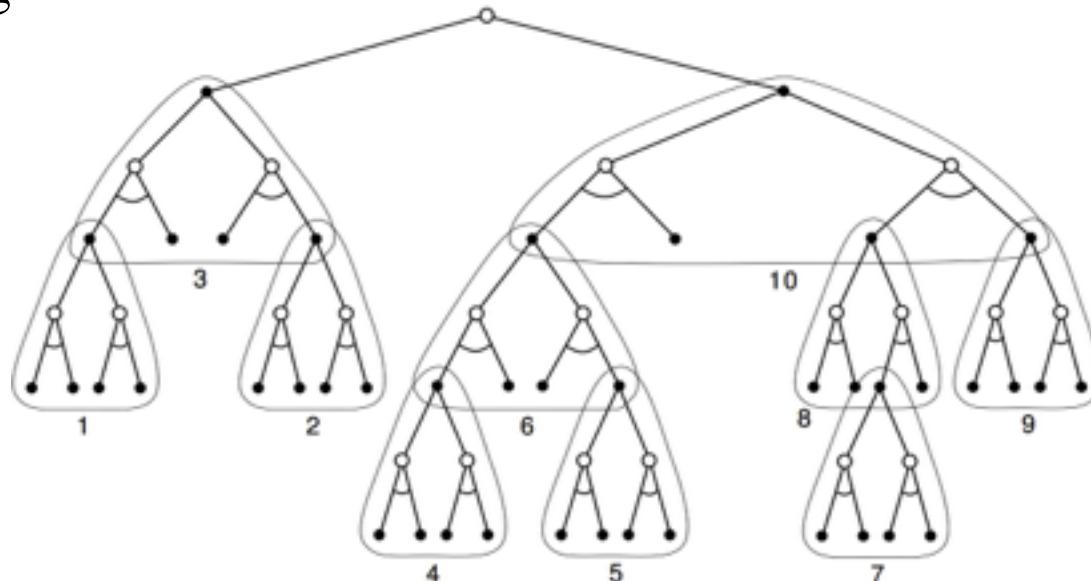


$$q_*(a,s)$$



Heuristic Search

- Used for action selection, not for changing a value function (=heuristic evaluation function)
- Backed-up values are computed, but typically discarded
- Extension of the idea of a greedy policy — only deeper
- Also suggests ways to select states to backup: smart focusing:



Summary

- Emphasized close relationship between planning and learning
- Important distinction between **distribution models** and **sample models**
- Looked at some ways to integrate planning and learning
 - synergy among planning, acting, model learning
- Distribution of backups: focus of the computation
 - prioritized sweeping
 - small backups
 - sample backups
 - trajectory sampling: backup along trajectories
 - heuristic search
- Size of backups: full/sample/small; deep/shallow

~~$V(s) \approx v_\pi(s) \approx \hat{v}(s, \theta) \doteq \theta^\top \phi(s) \doteq \sum_{i=1}^n \theta_i \cdot \phi_i(s) = 1.71$~~

transpose
inner product
*i*th components

$\theta \in \mathbb{R}^n$, e.g., $\theta =$ parameter vector

$$\begin{bmatrix} 2.1 \\ 0.01 \\ -1.1 \\ 1.2 \\ -0.1 \\ 0.01 \\ 4.93 \\ 0.5 \end{bmatrix}, \quad \phi(s) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \phi : \mathcal{S} \rightarrow \mathbb{R}^n$$

feature vector

~~$Q(s, a) \approx q_\pi(s, a) \approx \hat{q}(s, a, \theta) \doteq \theta^\top \phi(s, a) \doteq \sum_{i=1}^n \theta_i \cdot \phi_i(s, a)$~~

Chapter 9: Generalization and Function Approximation

Objectives of this chapter:

- Look at how experience with a limited part of the state set be used to produce good behavior over a much larger part.
- Overview of function approximation (FA) methods and how they can be adapted to RL

Value Prediction with Function Approx.

As usual: Policy Evaluation (the prediction problem):
for a given policy π , estimate the state-value function v_π

In earlier chapters, value functions were stored in lookup tables.

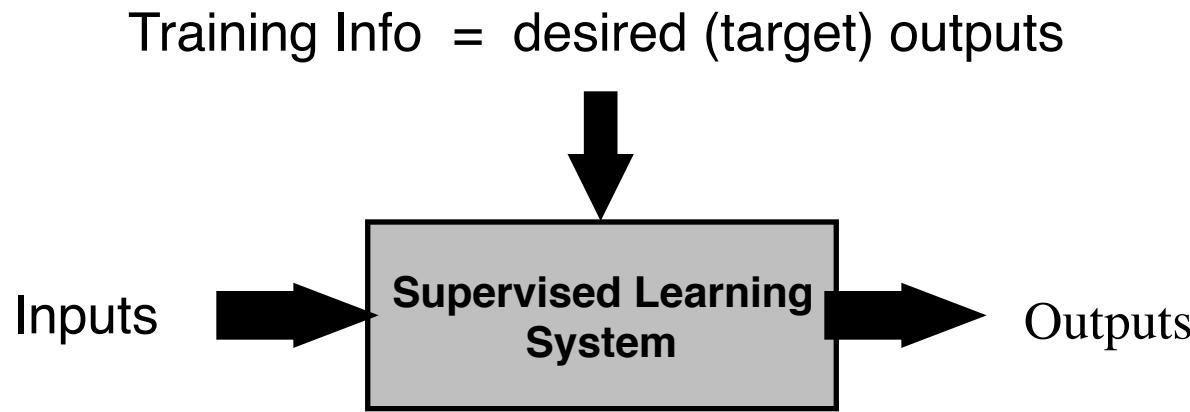
Now, the value function estimate at time t , V_t , depends
on a vector of parameters θ :

$$\hat{v}(s, \theta) \approx v_\pi(s)$$

only the parameters are updated

e.g., θ could be the modifiable connection weights and
thresholds of a deep neural network

Adapt Supervised Learning Algorithms



Training example = {input, target output}

Error = (target output – actual output)

Backups as Training Examples

For example, the TD(0) backup:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]$$

As a training example:

$$\text{features of } S_t \longrightarrow R_{t+1} + \gamma V(S_{t+1})$$

↑ ↑
input target output

Any FA Method?

□ In principle, yes:

- artificial neural networks
- decision trees
- multivariate regression methods
- etc.

□ But RL has some special requirements:

- usually want to learn while interacting (online)
- ability to handle nonstationarity
- other?

Gradient Descent Methods

$$\boldsymbol{\theta} \doteq (\theta_1, \theta_2, \dots, \theta_n)^\top$$

transpose 

Assume $\hat{v}(s, \boldsymbol{\theta})$ is a differentiable function of $\boldsymbol{\theta}$, for all $s \in \mathcal{S}$

Assume, for now, training examples of this form:

features of $S_t \longrightarrow v_\pi(S_t)$

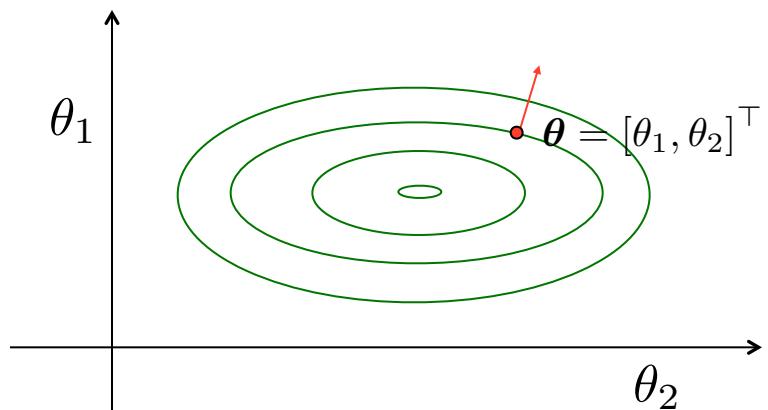
Gradient Descent

Let $f(\theta)$ be a function to be minimized, e.g., an error
Its gradient with respect to θ is

$$\nabla f(\theta) \doteq \frac{\partial f(\theta)}{\partial \theta} \doteq \left(\frac{\partial f(\theta)}{\partial \theta_1}, \frac{\partial f(\theta)}{\partial \theta_2}, \dots, \frac{\partial f(\theta)}{\partial \theta_n} \right)^\top$$

Iteratively move “down”
the gradient:

$$\theta \leftarrow \theta - \alpha \nabla f(\theta)$$



Performance Measures

- Many are applicable but...
- a common and simple one is the mean-squared error (MSE) over a distribution d :

$$\text{MSVE}(\boldsymbol{\theta}) = \sum_{s \in \mathcal{S}} d(s) \left[v_\pi(s) - \hat{v}(s, \boldsymbol{\theta}) \right]^2$$

- Why d ?
- Why minimize MSVE?
- Let us assume that d is always the distribution of states at which backups are done.
- The **on-policy distribution**: the distribution created while following the policy being evaluated. Stronger results are available for this distribution.

Gradient Descent Derivation

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \alpha \nabla \text{MSVE}(\boldsymbol{\theta}_t) \\&= \boldsymbol{\theta}_t - \alpha \nabla \sum_{s \in \mathcal{S}} d(s) [v_\pi(s) - \hat{v}(s, \boldsymbol{\theta}_t)]^2 \\&= \boldsymbol{\theta}_t - \alpha \sum_{s \in \mathcal{S}} d(s) \nabla [v_\pi(s) - \hat{v}(s, \boldsymbol{\theta}_t)]^2 \\&= \boldsymbol{\theta}_t - 2\alpha \sum_{s \in \mathcal{S}} d(s) [v_\pi(s) - \hat{v}(s, \boldsymbol{\theta}_t)] \nabla [v_\pi(s) - \hat{v}(s, \boldsymbol{\theta}_t)] \\&= \boldsymbol{\theta}_t + \alpha \sum_{s \in \mathcal{S}} d(s) [v_\pi(s) - \hat{v}(s, \boldsymbol{\theta}_t)] \nabla \hat{v}(s, \boldsymbol{\theta}_t) \\&\quad (\text{sampling}) \\&= \boldsymbol{\theta}_t + \alpha [v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{\theta}_t)] \nabla \hat{v}(S_t, \boldsymbol{\theta}_t)\end{aligned}$$

Since each sample gradient is an **unbiased estimate** of the true gradient, this converges to a local minimum of the MSVE if α decreases appropriately with t .

But We Don't have these Targets

Suppose we just have targets V_t instead :

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left[V_t - \hat{v}(S_t, \boldsymbol{\theta}_t) \right] \nabla \hat{v}(S_t, \boldsymbol{\theta}_t)$$

If each V_t is an unbiased estimate of $v_\pi(S_t)$,
i.e., $E\{V_t\} = v_\pi(S_t)$, then gradient descent converges
to a local minimum (provided α decreases appropriately).

e.g., the Monte Carlo target $V_t = G_t$ (unbiased):

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left[G_t - \hat{v}(S_t, \boldsymbol{\theta}_t) \right] \nabla \hat{v}(S_t, \boldsymbol{\theta}_t)$$

What about TD(λ) Targets?

What about the λ -return, G_t^λ ?

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left[G_t^\lambda - \hat{v}(S_t, \boldsymbol{\theta}_t) \right] \nabla \hat{v}(S_t, \boldsymbol{\theta}_t)$$

Unfortunately, G_t^λ is biased for $\lambda < 1$
⇒ standard gradient descent results don't apply

But we do it anyway!

first, some meta comments on

Understanding Algorithms

- I. Do I understand the symbols and their meaning?
 - Could I write a program to do it?
 - Does it make intuitive sense?
2. Can I derive the algorithm from some objective?
3. Can I prove that the algorithm converges to some objective?
4. Can I prove something about the rate of convergence?

and some meta comments on
Efficient Scaling

3 Kinds of Efficiency in Machine Learning & AI

- I. **Data** efficiency (rate of learning)
2. **Computational** efficiency (memory, computation, communication)
3. **User** efficiency (autonomy, ease of setup, lack of parameters, priors, labels, expertise)

Computational Resources

1. Memory
2. Computation
3. Communication (wires)

Natural Scaling

- Every learning system has two parts
 1. the thing that is learned (e.g., the neural network and its weights)
 2. the algorithm that learns it (e.g., the algorithm that learns the weights)
- *Natural scaling* is when the computational complexities of the two parts scale similarly

Gradient-based TD(λ), backwards view

$$\delta_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \theta_t) - \hat{v}(S_t, \theta_t)$$

$$\mathbf{e}_t \doteq \gamma \lambda \mathbf{e}_{t-1} + \nabla \hat{v}(S_t, \theta_t)$$

$$\theta_{t+1} \doteq \theta_t + \alpha \delta_t \mathbf{e}_t$$

On-Line Gradient-Descent TD(λ)

Initialize $\boldsymbol{\theta}$ as appropriate for the problem, e.g., $\boldsymbol{\theta} = \mathbf{0}$

Repeat (for each episode):

$$\mathbf{e} = \mathbf{0}$$

$S \leftarrow$ initial state of episode

Repeat (for each step of episode):

$A \leftarrow$ action given by π for S

Take action A , observe reward, R , and next state, S'

$$\delta \leftarrow R + \gamma \hat{v}(S', \boldsymbol{\theta}) - \hat{v}(S, \boldsymbol{\theta})$$

$$\mathbf{e} \leftarrow \gamma \lambda \mathbf{e} + \nabla \hat{v}(S, \boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \delta \mathbf{e}$$

$$S \leftarrow S'$$

until S' is terminal

Linear Methods

Represent states as feature vectors:

for each $s \in \mathcal{S}$:

$$\hat{v}(s, \boldsymbol{\theta}) \doteq \boldsymbol{\theta}^\top \boldsymbol{\phi}(s) = \sum_{i=1}^n \theta_i x_i(s)$$

$$\nabla \hat{v}(s, \boldsymbol{\theta}) = \boldsymbol{\phi}(s)$$

Nice Properties of Linear FA Methods

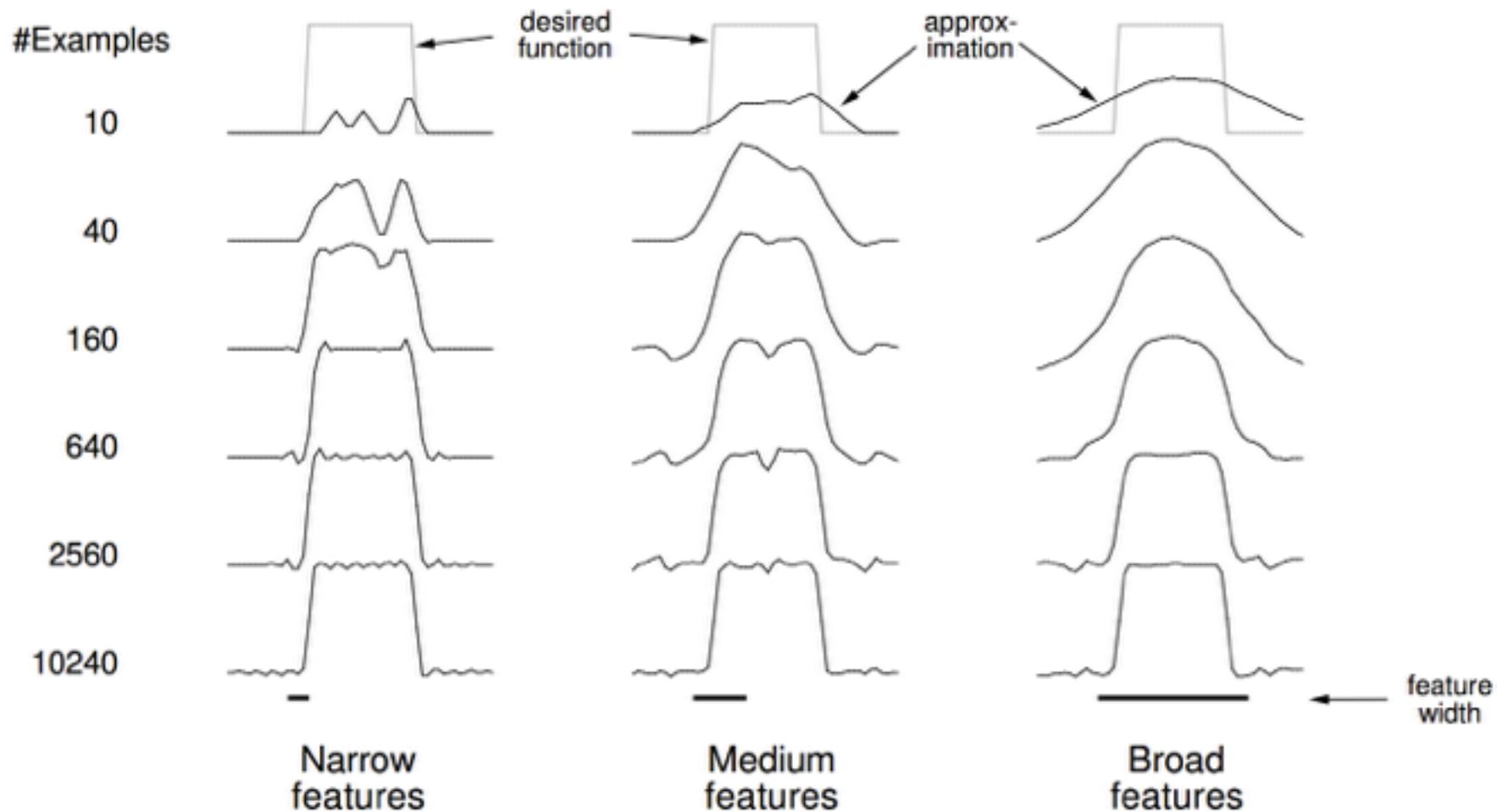
- The gradient is very simple: $\nabla \hat{v}(s, \theta) = \phi(s)$
- For MSE, the error surface is simple: quadratic surface with a single minimum.
- Linear gradient descent TD(λ) converges:
 - Step size decreases appropriately
 - On-line sampling (states sampled from the on-policy distribution)
 - Converges to weight vector θ_∞ with property:

$$\text{MSVE}(\theta_\infty) \leq \frac{1 - \gamma\lambda}{1 - \gamma} \text{MSVE}(\theta^*)$$

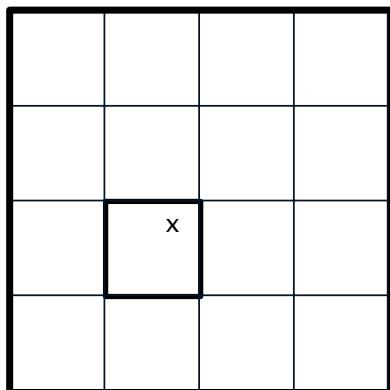
(Tsitsiklis & Van Roy, 1997)

best weight vector

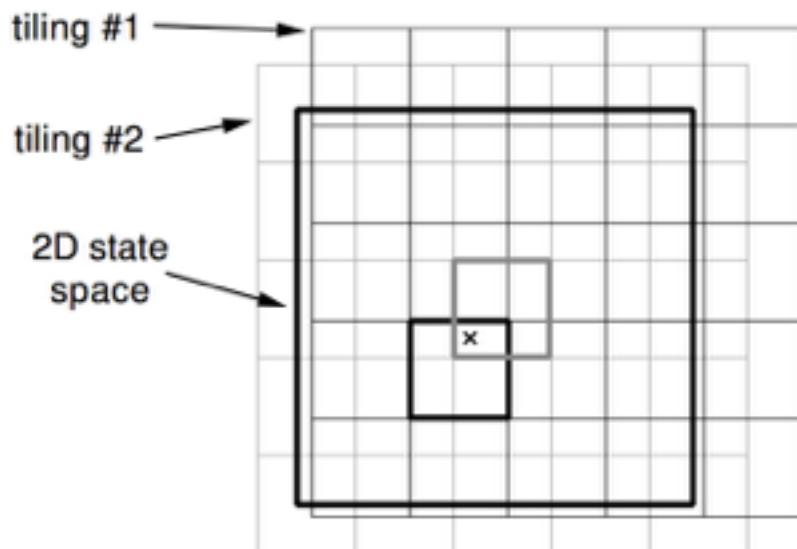
Learning and Coarse Coding



Tile Coding



- Binary feature for each tile
- Number of features present at any one time is constant
- Binary features means weighted sum easy to compute
- Easy to compute indices of the features present

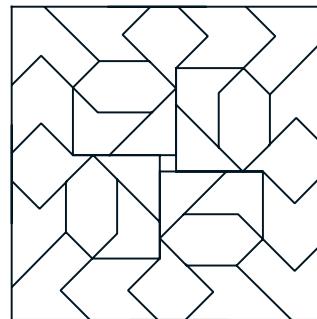


Shape of tiles \Rightarrow Generalization

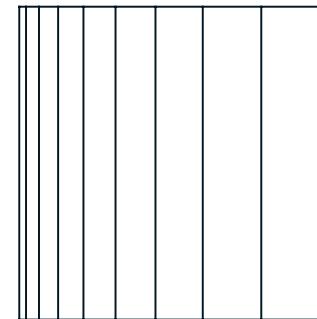
#Tilings \Rightarrow Resolution of final approximation

Tile Coding Cont.

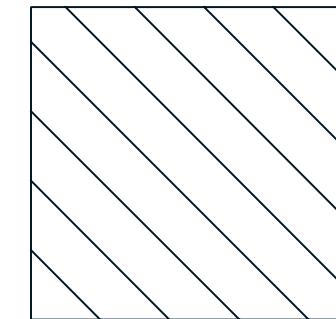
Irregular tilings



a) Irregular

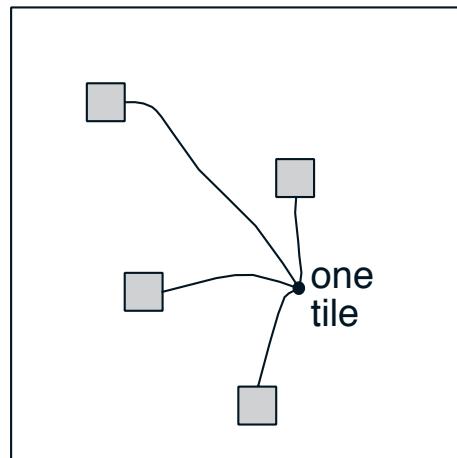


b) Log stripes



c) Diagonal stripes

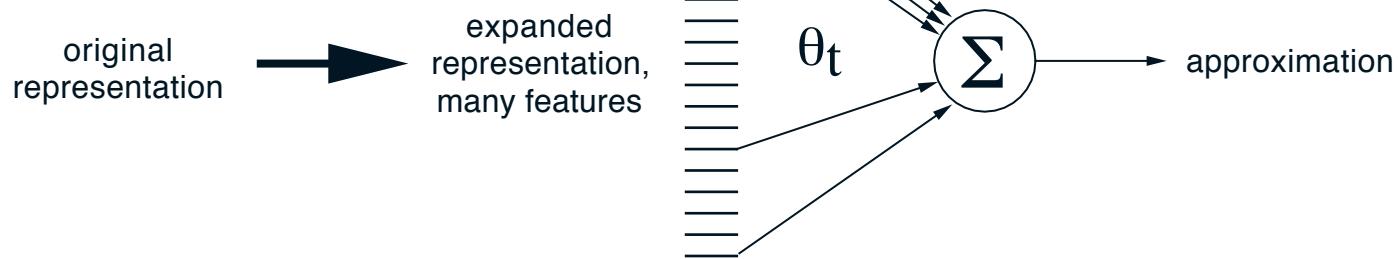
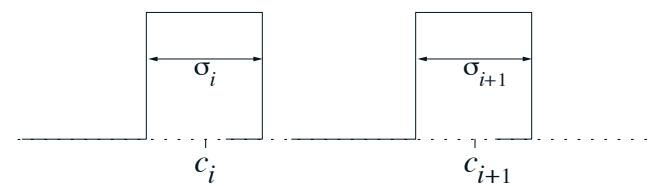
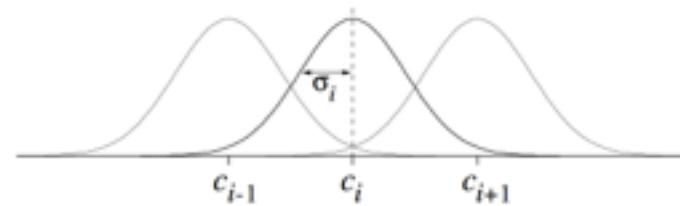
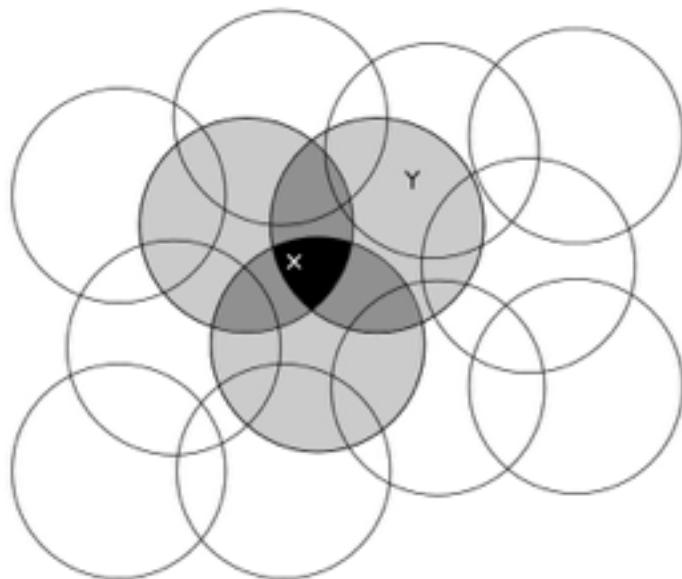
Hashing



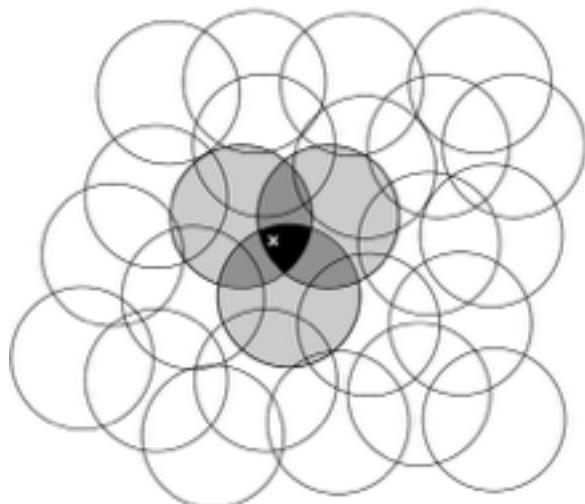
CMAC

“Cerebellar model arithmetic computer”
Albus 1971

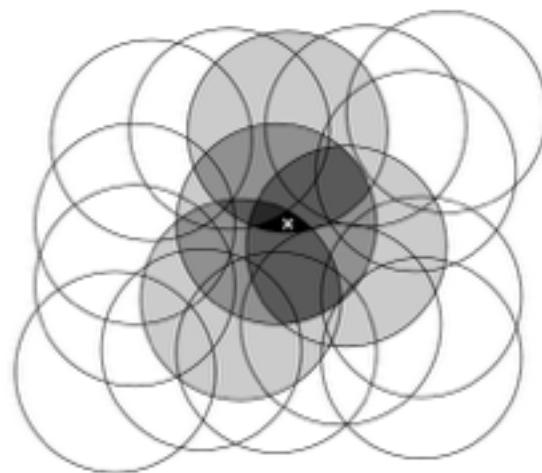
Coarse Coding



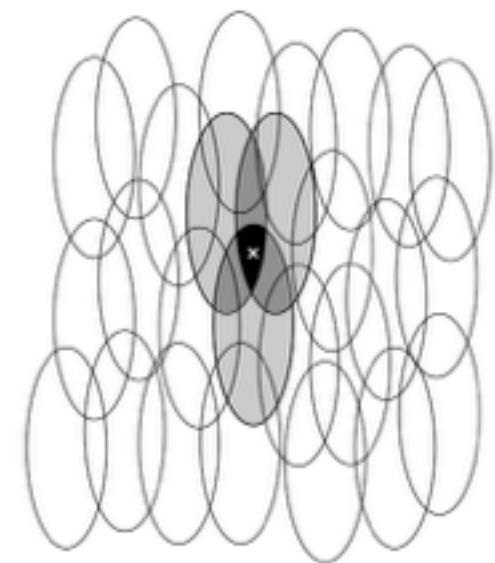
Shaping Generalization in Coarse Coding



a) Narrow generalization



b) Broad generalization



c) Asymmetric generalization

Can you beat the “curse of dimensionality”?

- Can you keep the number of features from going up exponentially with the dimension?
- Function complexity, not dimensionality, is the problem.
- Kanerva coding:
 - Select a bunch of binary **prototypes**
 - Use hamming distance as distance measure
 - Dimensionality is no longer a problem, only complexity
- “Lazy learning” schemes:
 - Remember all the data
 - To get new value, find nearest neighbours and interpolate
 - e.g., locally-weighted regression

Control with FA

□ Learning state-action values

Training examples of the form:

$$\{\text{description of } (S_t, A_t), Q_t\}$$

□ The general gradient-descent rule:

$$\theta_{t+1} \doteq \theta_t + \alpha [Q_t - \hat{q}(S_t, A_t, \theta_t)] \nabla \hat{q}(S_t, A_t, \theta_t)$$

□ Gradient-descent Sarsa(λ) (backward view):

$$\theta_{t+1} \doteq \theta_t + \alpha \delta_t \mathbf{e}_t$$

where: $\delta_t \doteq R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \theta_t) - \hat{q}(S_t, A_t, \theta_t)$

$$\mathbf{e}_t \doteq \gamma \lambda \mathbf{e}_{t-1} + \nabla \hat{q}(S_t, A_t, \theta_t)$$

Linear Gradient-based Sarsa(λ)

Let $\boldsymbol{\theta}$ and \mathbf{e} be vectors with one component for each possible feature

Let \mathcal{F}_a , for every possible action a , be a set of feature indices, initially empty

Initialize $\boldsymbol{\theta}$ as appropriate for the problem, e.g., $\boldsymbol{\theta} = \mathbf{0}$

Repeat (for each episode):

$$\mathbf{e} = \mathbf{0}$$

$S, A \leftarrow$ initial state and action of episode (e.g., ε -greedy)

$\mathcal{F}_A \leftarrow$ set of features present in S, A

Repeat (for each step of episode):

For all $i \in \mathcal{F}_A$:

$$e_i \leftarrow e_i + 1 \quad (\text{accumulating traces})$$

$$\text{or } e_i \leftarrow 1 \quad (\text{replacing traces})$$

Take action A , observe reward, R , and next state, S'

$$\delta \leftarrow R - \sum_{i \in \mathcal{F}_A} \theta_i$$

If S' is terminal, then $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \delta \mathbf{e}$; go to next episode

For all $a \in \mathcal{A}(S')$:

$\mathcal{F}_a \leftarrow$ set of features present in S', a

$$Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta_i$$

$A' \leftarrow$ new action in S' (e.g., ε -greedy)

$$\delta \leftarrow \delta + \gamma Q_{A'}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \delta \mathbf{e}$$

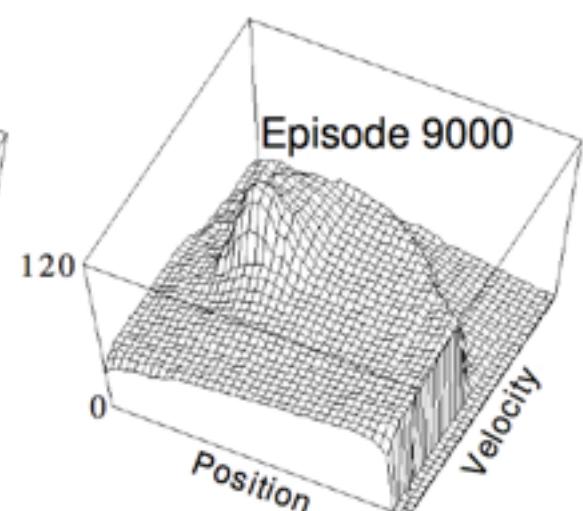
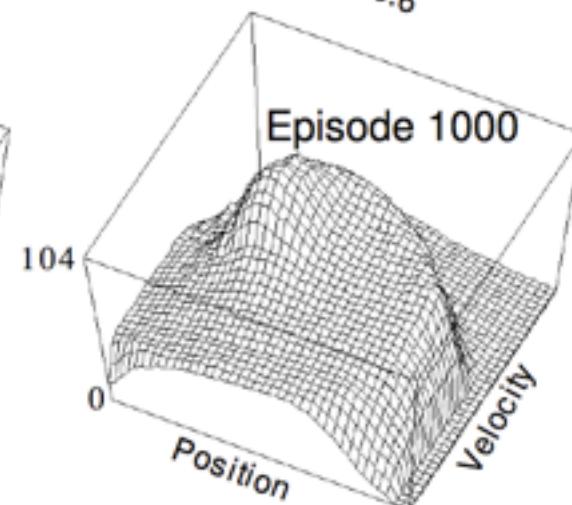
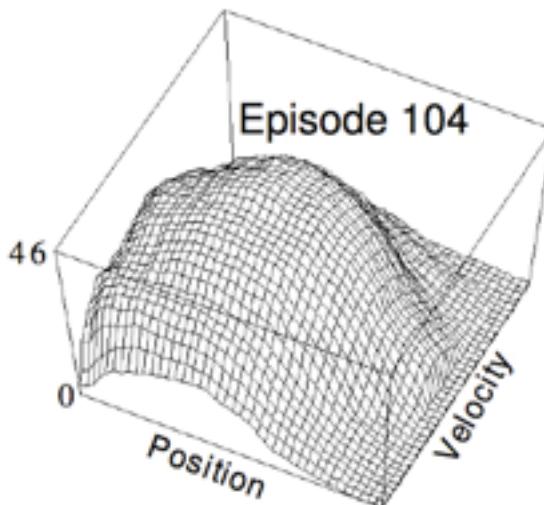
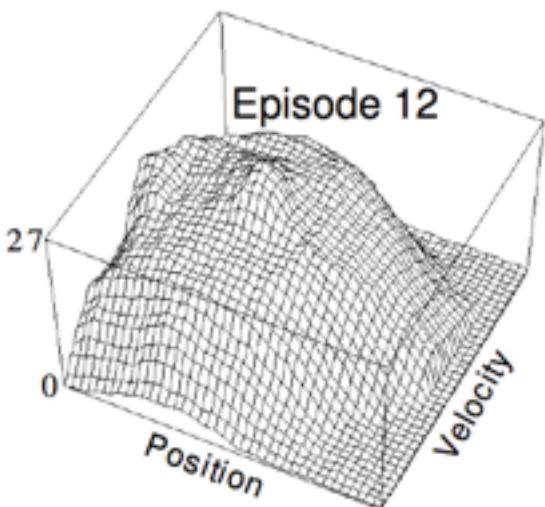
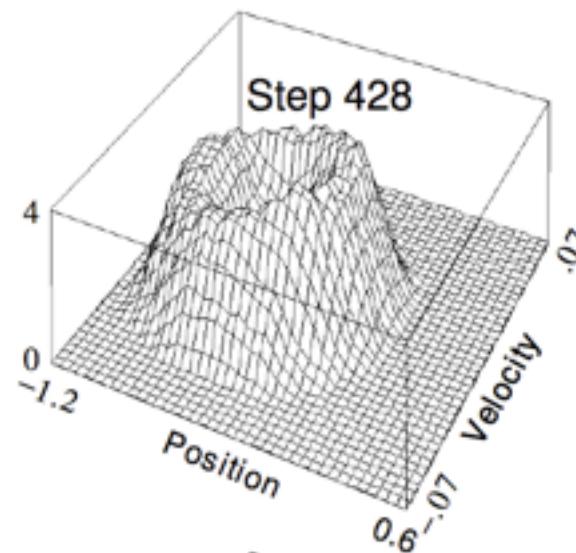
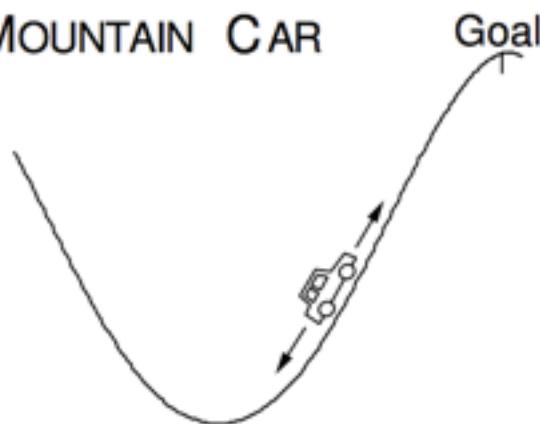
$$\mathbf{e} \leftarrow \gamma \lambda \mathbf{e}$$

$$S \leftarrow S'$$

$$A \leftarrow A'$$

Approx Value Functions on Mountain-Car Task

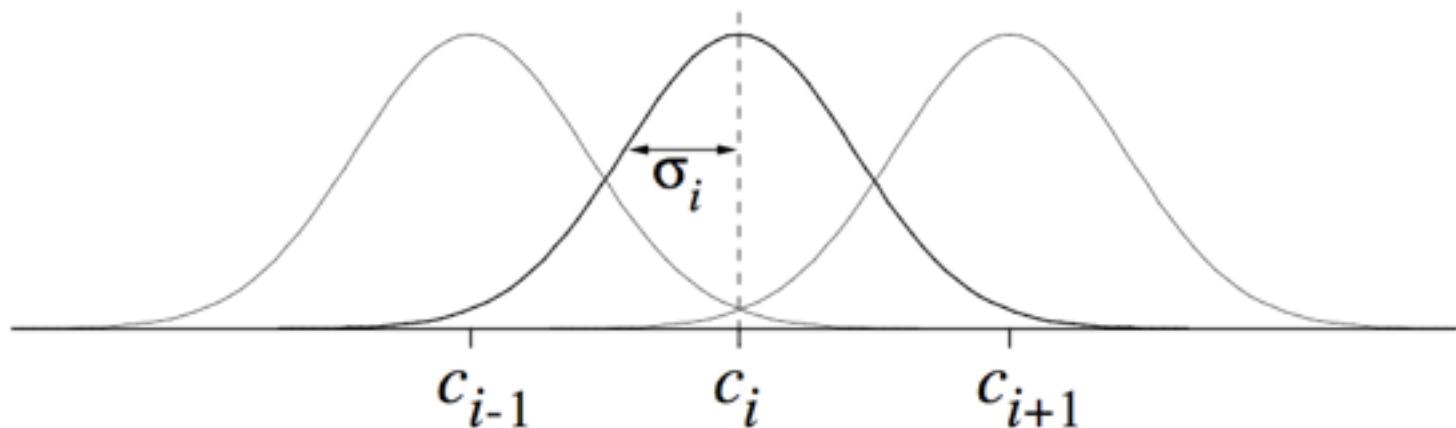
MOUNTAIN CAR



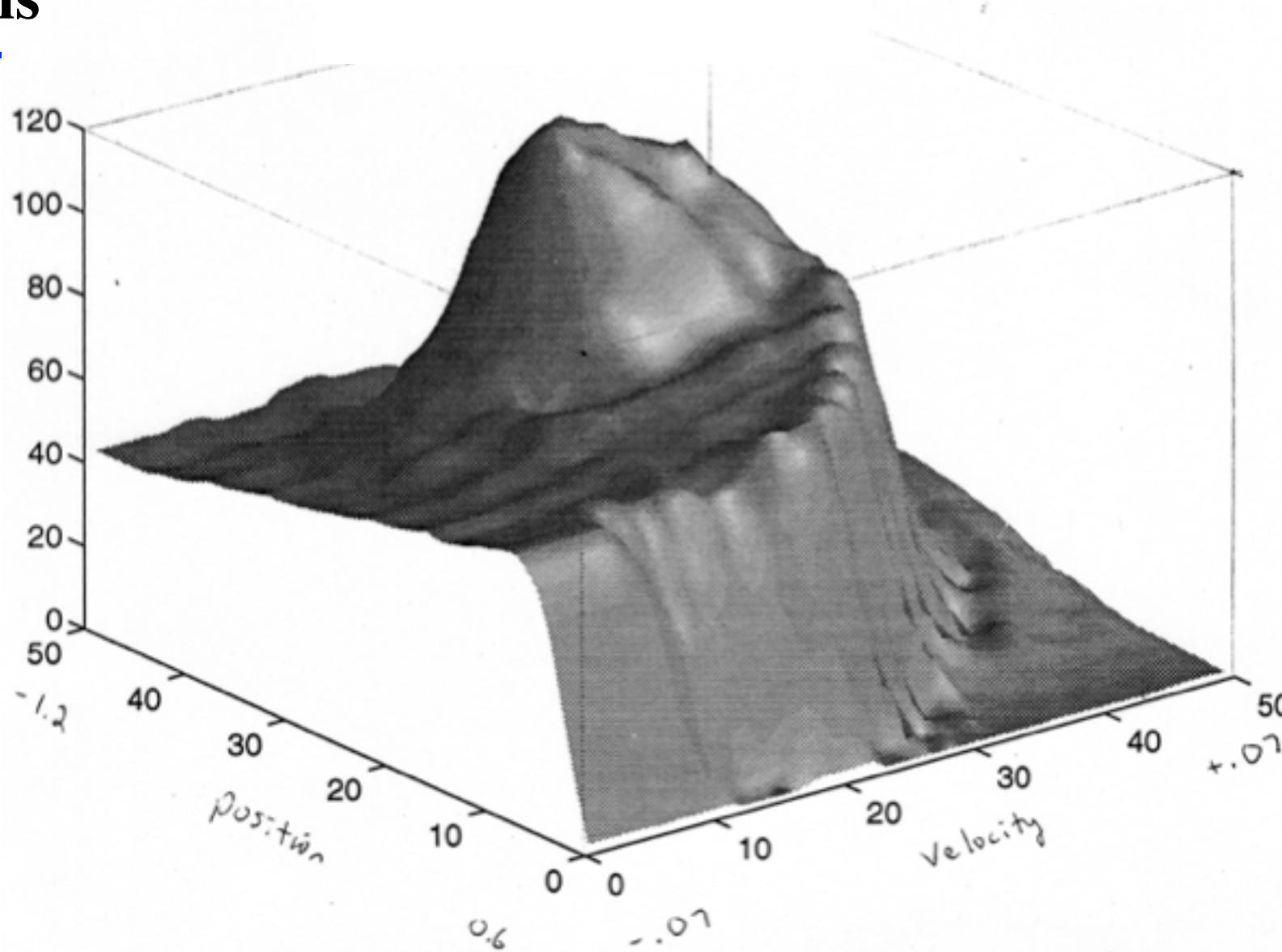
Radial Basis Functions (RBFs)

e.g., Gaussians

$$x_i(s) = \exp\left(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right)$$



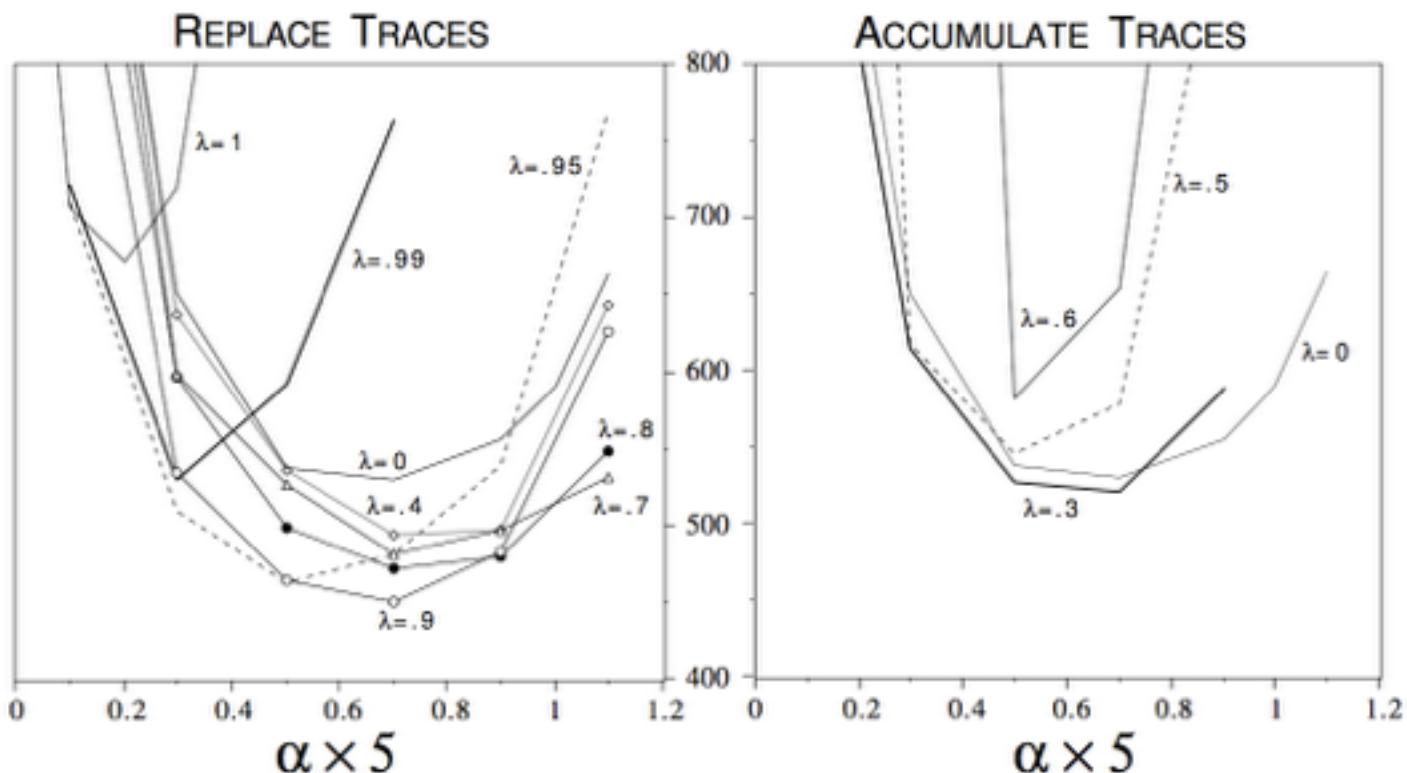
Mountain Car with Radial Basis Functions



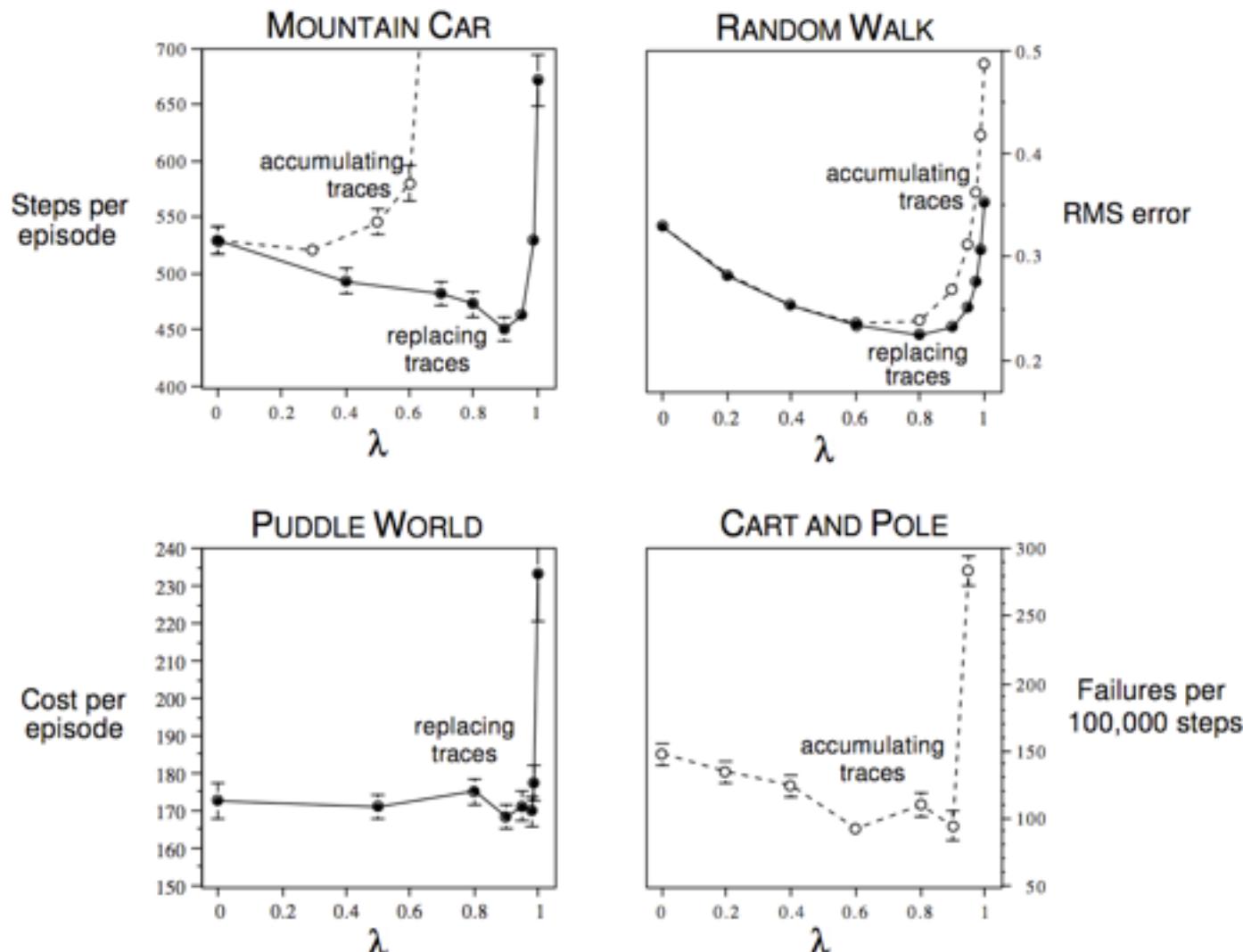
Matt Kretchmar,

Mountain-Car Results

Steps per episode
averaged over
first 20 trials
and 30 runs



Should We Bootstrap?



Summary

- ❑ Generalization
- ❑ Adapting supervised-learning function approximation methods
- ❑ Gradient-descent methods
- ❑ Linear gradient-descent methods
 - Radial basis functions
 - Tile coding
 - Kanerva coding

Eligibility Traces

Unifying Monte Carlo and TD

key algorithms: $\text{TD}(\lambda)$, $\text{Sarsa}(\lambda)$

Mathematics of N-step TD Prediction

- Monte Carlo: $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$
- TD(0): $G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1})$
 - Use V to estimate remaining return
- n -step return:
 - 2 step return: $G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})$
 - n -step return: $G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$

Backup Diagrams

TD (1-step)



2-step



3-step



n -step



Monte Carlo



Learning with n -step Backups

- On-line update:

$$V_{t+1}(S_t) = V_t(S_t) + \alpha[G_t^{(n)} - V_t(S_t)]$$

$$V_{t+1}(s) = V_t(s) \quad \text{for } s \neq S_t$$

- Alternative formulation:

$$\Delta_t(s) \doteq \begin{cases} \alpha[G_t^{(n)} - V_t(S_t)] & \text{if } s = S_t \\ 0 & \text{if } s \neq S_t \end{cases}$$

$$V_{t+1}(s) = V_t(s) + \Delta_t(s) \quad \text{for all } s$$

Off-line Updating

- On-line updating: values change at every time step
- Off-line updating: values change only at the last time step

- On-line update:

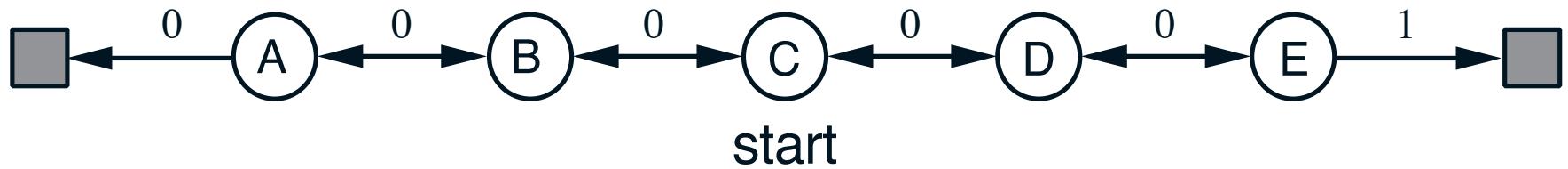
$$V_{t+1}(s) = V_t(s) + \Delta_t(s) \quad \text{for } 0 \leq t < T$$

- Off-line update:

$$V_{t+1}(s) = \begin{cases} V_t(s) & \text{for } 0 \leq t < T - 1 \\ V_t(s) + \sum_{i=0}^{T-1} \Delta_i(s) & \text{for } t = T - 1 \end{cases}$$

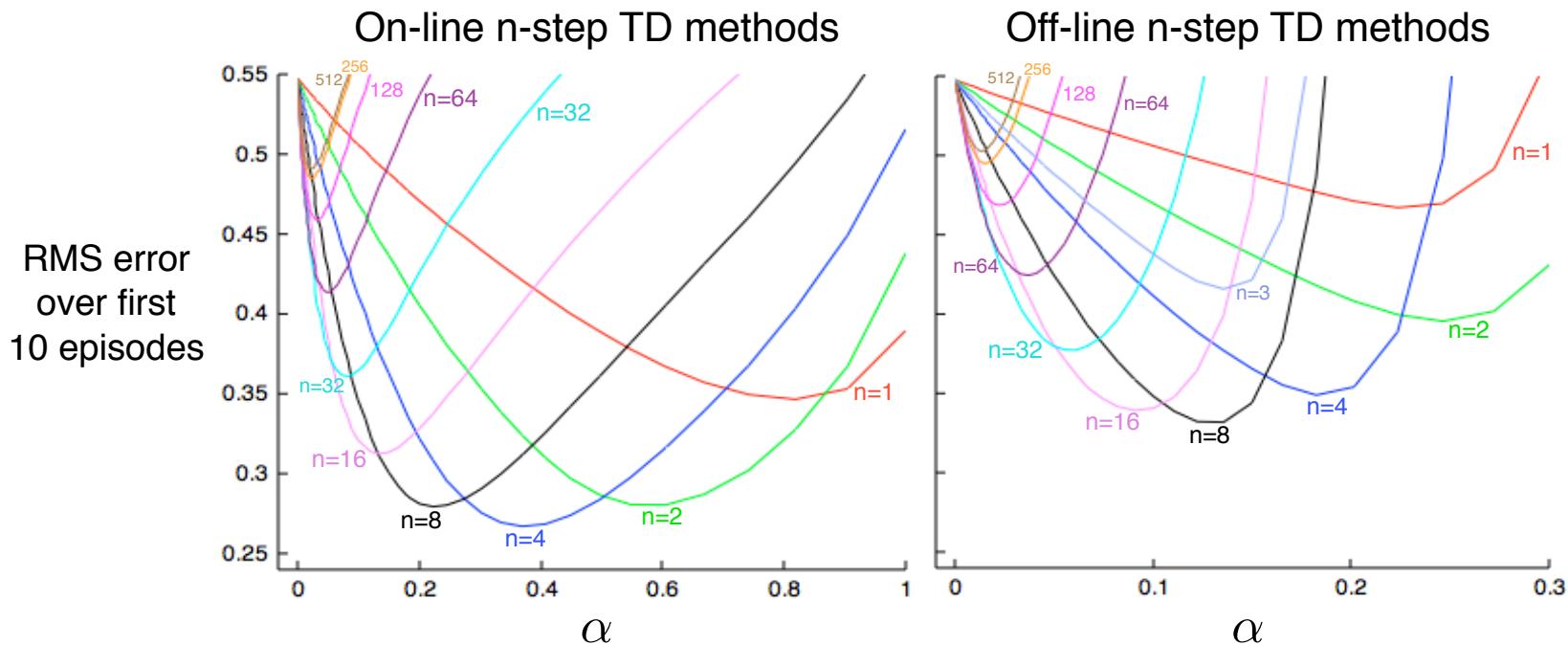
What are some of the advantages of on-line
updating over off-line updating?

Random Walk Examples



transitions to the left and to the right occur with equal probability

Results on 19-state Random Walk



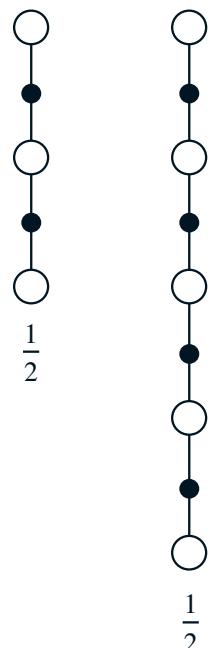
Combining Multiple N-step Returns

- Idea: backup an average of several returns
 - e.g. backup half of 2-step and half of 4-step

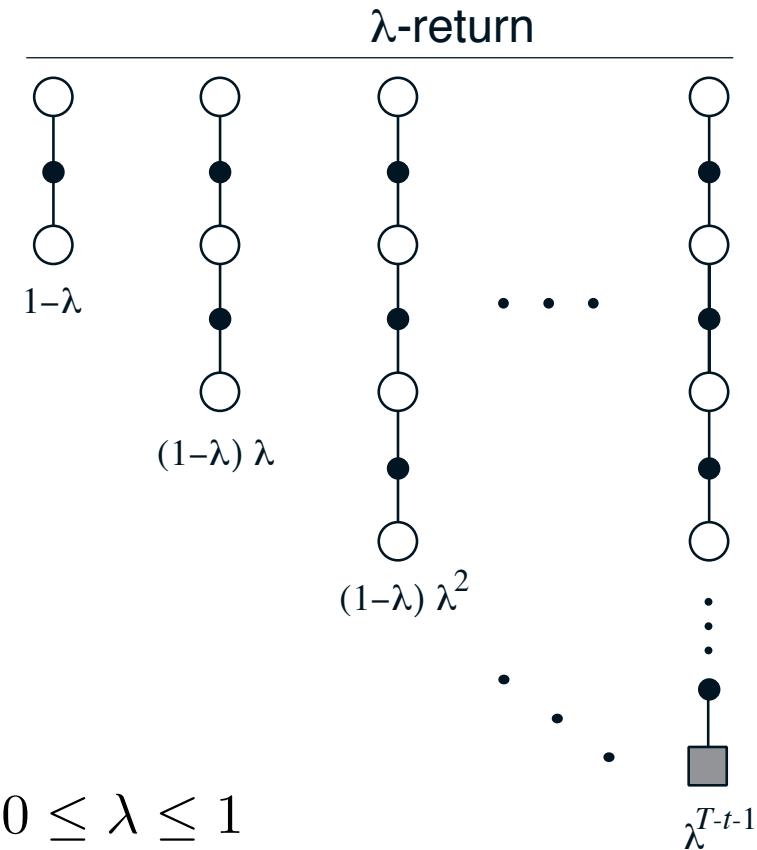
$$G_t^{ave} = \frac{1}{2}G_t^{(2)} + \frac{1}{2}G_t^{(4)}$$

- Called a complex backup
 - Draw each component
 - Label with the weights for that component

One backup



The λ -return



$$(1 + \lambda) + (1 + \lambda)\lambda + (1 + \lambda)\lambda^2 + \dots + \lambda^{T-t-1} = 1$$

The λ -return

- The λ -return can be rewritten as:

$$G_t^\lambda = \sum_{n=1}^{T-t-1} w_n G_t^{(n)} + w_{T-t} G_t$$

with $w_n = \begin{cases} (1 - \lambda) \lambda^{n-1} & \text{if } 1 \leq n < T - t \\ \lambda^{n-1} & \text{if } n = T - t \end{cases}$ $\left(\sum_{n=1}^{T-t} w_n = 1 \right)$

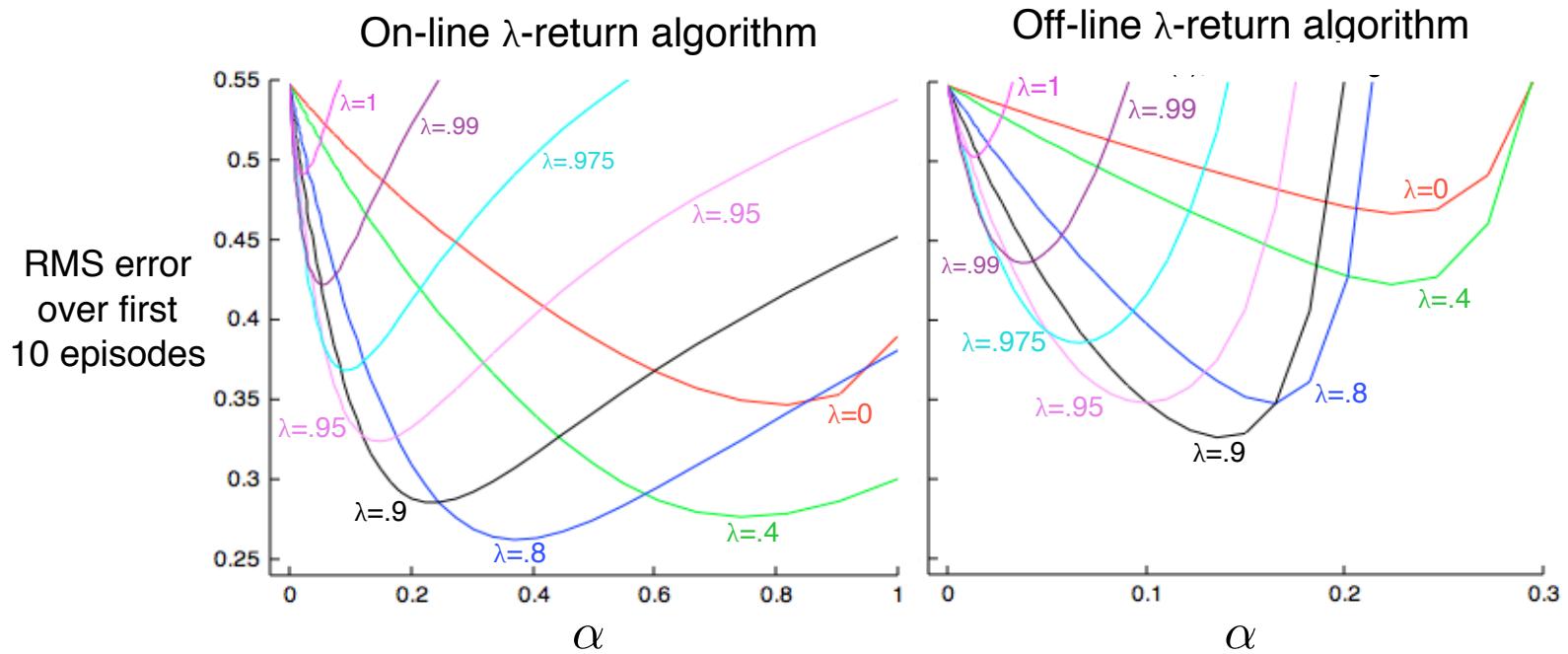
- If $\lambda = 1$: $G_t^\lambda = (1 - 1) \sum_{n=1}^{T-t-1} 1^{n-1} G_t^{(n)} + 1^{T-t-1} G_t = G_t$

- If $\lambda = 0$: $G_t^\lambda = (1 - 0) \sum_{n=1}^{T-t-1} 0^{n-1} G_t^{(n)} + 0^{T-t-1} G_t = G_t^{(1)}$

Forward View of TD(λ)

- The algorithm that performs updates with the λ -return is called the λ -return algorithm
- This algorithm represents the forward view of TD(λ)
- There is an on-line version as well as an off-line version
- Mainly interesting for analytical purposes, impractical to implement

Results on 19-state Random Walk



Eligibility Traces - Backward view

- update equations:

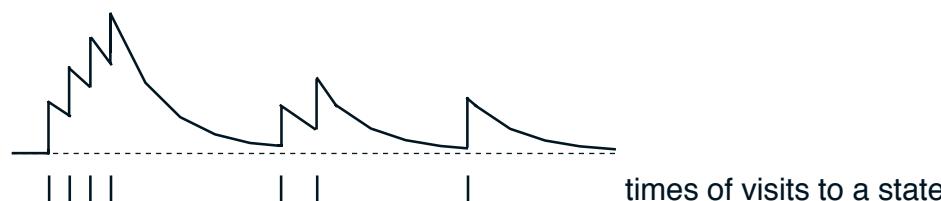
$$V_{t+1}(s) = V_t(s) + \alpha E_t(s) [R_{t+1} + V_t(S_{t+1}) - V_t(S_t)]$$

\downarrow

$$\delta_t \quad (\text{one-step TD error})$$

- new variable, the *eligibility trace*, E_t

$$E_t(s) = \begin{cases} \gamma\lambda E_{t-1}(s) + 1 & \text{if } s = S_t \\ \gamma\lambda E_{t-1}(s) & \text{if } s \neq S_t \end{cases} \quad (\text{accumulating trace})$$



On-line Tabular TD(λ)

Initialize $V(s)$ arbitrarily

Repeat (for each episode):

 Initialize $E(s) = 0$, for all $s \in \mathcal{S}$

 Initialize S

 Repeat (for each step of episode):

$A \leftarrow$ action given by π for S

 Take action A , observe reward, R , and next state, S'

$\delta \leftarrow R + \gamma V(S') - V(S)$

$E(S) \leftarrow E(S) + 1$

 For all $s \in \mathcal{S}$:

$V(s) \leftarrow V(s) + \alpha \delta E(s)$

$E(s) \leftarrow \gamma \lambda E(s)$

$S \leftarrow S'$

 until S is terminal

Forward View/Backward View Equivalence

V^f : value estimate computed by λ -return algorithm

V^b : value estimate computed by TD(λ) with acc. traces

- With off-line learning:

$$V_T^f(s) = V_T^b(s), \quad \text{for all } s$$

- With on-line learning, for appropriately small α :

$$V_T^f(s) \approx V_T^b(s), \quad \text{for all } s$$

Forward view is mainly used for analytical purposes,
while backward view is used for implementation.

Why?

Advantage Backward View

- Computational efficiency
- TD(λ) can be applied to non-episodic tasks
- Values computed by λ -return algorithm can only be computed in hindsight, at the end of an episode

Other Traces

- Accumulating Traces:

$$E_t(s) \doteq \begin{cases} \gamma\lambda E_{t-1}(s) + 1 & \text{if } s = S_t \\ \gamma\lambda E_{t-1}(s) & \text{if } s \neq S_t \end{cases}$$

- Replacing Traces:

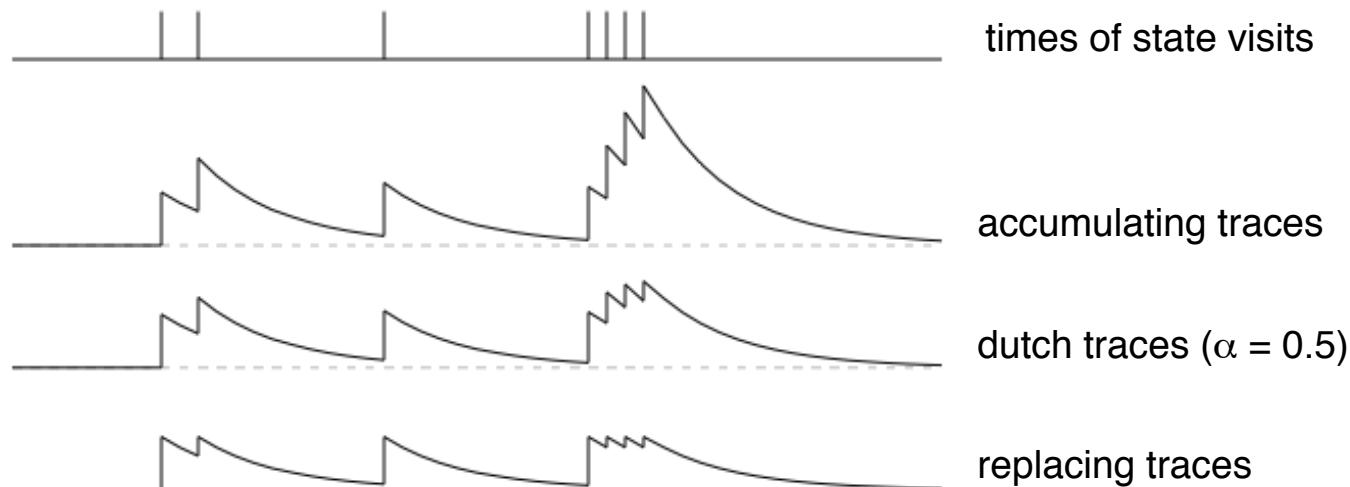
$$E_t(s) \doteq \begin{cases} 1 & \text{if } s = S_t \\ \gamma\lambda E_{t-1}(s) & \text{if } s \neq S_t \end{cases}$$

- Dutch Traces:

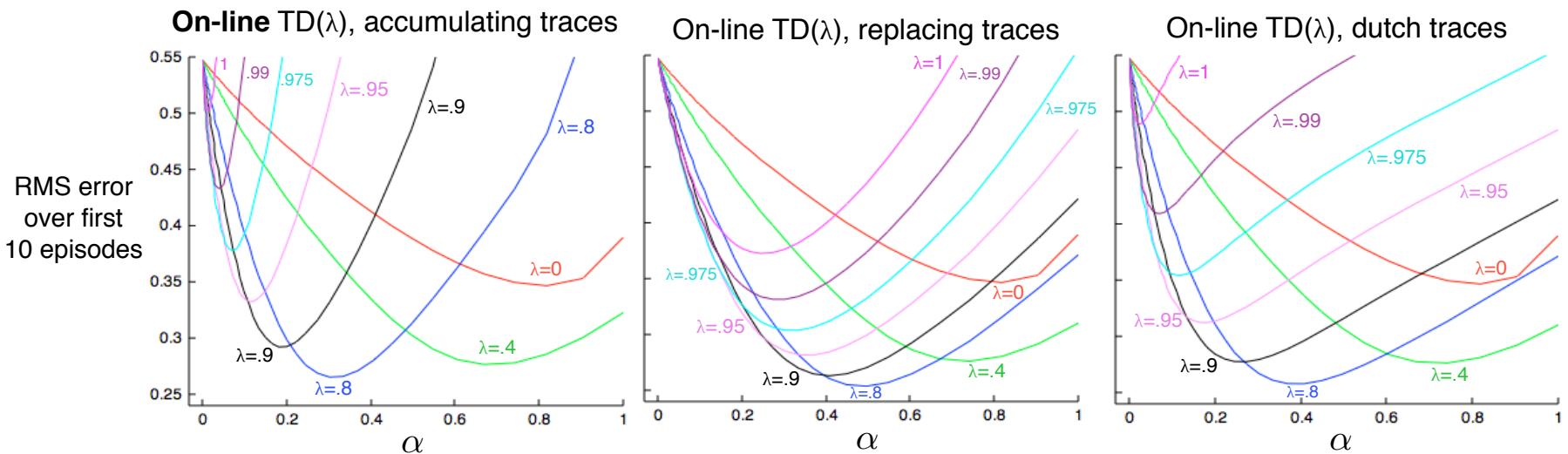
$$E_t(s) \doteq \begin{cases} (1 - \alpha)\gamma\lambda E_{t-1}(s) + 1 & \text{if } s = S_t \\ \gamma\lambda E_{t-1}(s) & \text{if } s \neq S_t \end{cases}$$

Traces Comparison

- Bounds, for $0 \leq \alpha \leq 1$:
 - Accumulating traces: $0 \leq \alpha E_t(s) \leq \alpha/(1 - \gamma\lambda)$
 - Replacing traces: $0 \leq \alpha E_t(s) \leq \alpha$
 - Dutch traces: $0 \leq \alpha E_t(s) \leq 1$



Results on 19-state Random Walk



Control: Sarsa(λ)

- Everything changes from states to state-action pairs

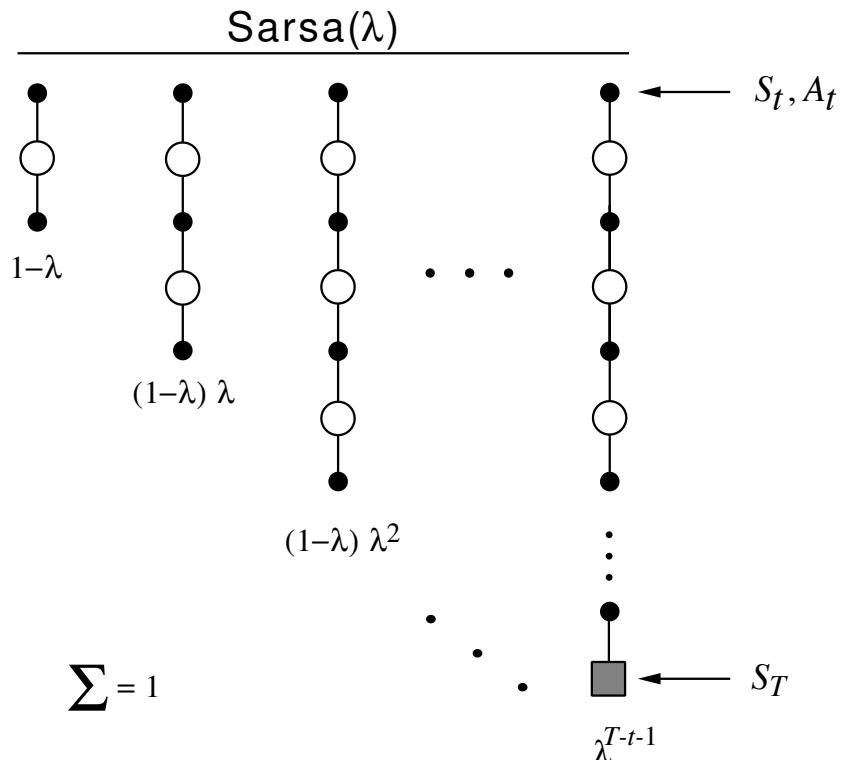
$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha \delta_t E_t(s, a), \quad \forall s, a$$

where

$$\delta_t = R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t) \quad \sum = 1$$

and

$$E_t(s, a) = \begin{cases} \gamma \lambda E_{t-1}(s, a) + 1 & \text{if } s = S_t \text{ and } a = A_t; \\ \gamma \lambda E_{t-1}(s, a) & \text{otherwise.} \end{cases} \quad \text{for all } s, a$$



Sarsa(λ) Algorithm

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

Repeat (for each episode):

$$E(s, a) = 0, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s)$$

Initialize S, A

Repeat (for each step of episode):

Take action A , observe R, S'

Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$$\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$$

$$E(S, A) \leftarrow E(S, A) + 1$$

For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:

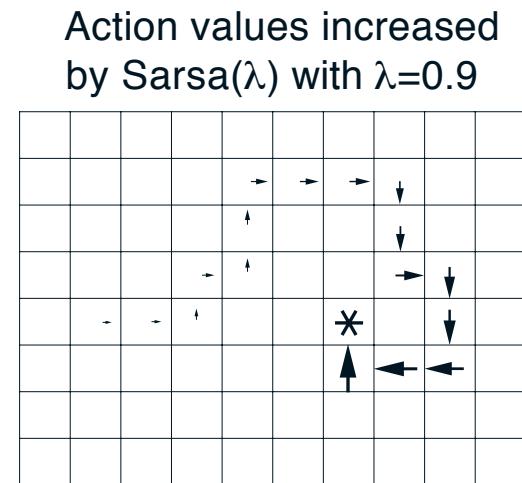
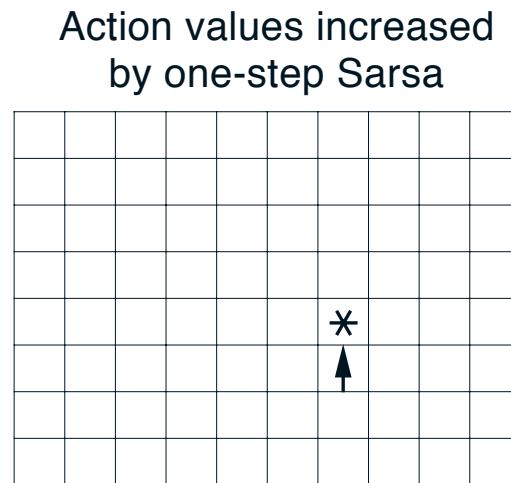
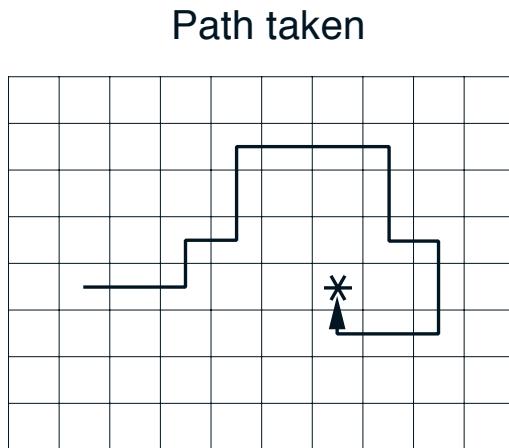
$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta E(s, a)$$

$$E(s, a) \leftarrow \gamma \lambda E(s, a)$$

$$S \leftarrow S'; A \leftarrow A'$$

until S is terminal

Sarsa(λ) Gridworld Example



- With one trial, the agent has much more information about how to get to the goal
 - not necessarily the *best* way
- Can considerably accelerate learning

Other Variations

- $Q(\lambda)$: off-policy control method using eligibility traces
- true online $TD(\lambda)$ / true online Sarsa(λ) : dutch traces + modified value update; results in exact equivalence with forward view
- time-dependent λ

Summary

- The update targets of TD(0) and MC lie at the extremes of a spectrum of possible update targets.
- The λ -return is a parameterized complex backup, that spans the whole spectrum.
- By tuning λ to the task at hand, significant speed-ups can be obtained.
- TD(λ) approximates the λ -return algorithm by using eligibility traces.
- TD(λ) has the advantage over the λ -return algorithm that it is computationally efficient, can be applied to non-episodic tasks and that values can be computed in real-time.
- TD(λ) can be easily extended to control: Sarsa(λ).

Deterministic Tree Search

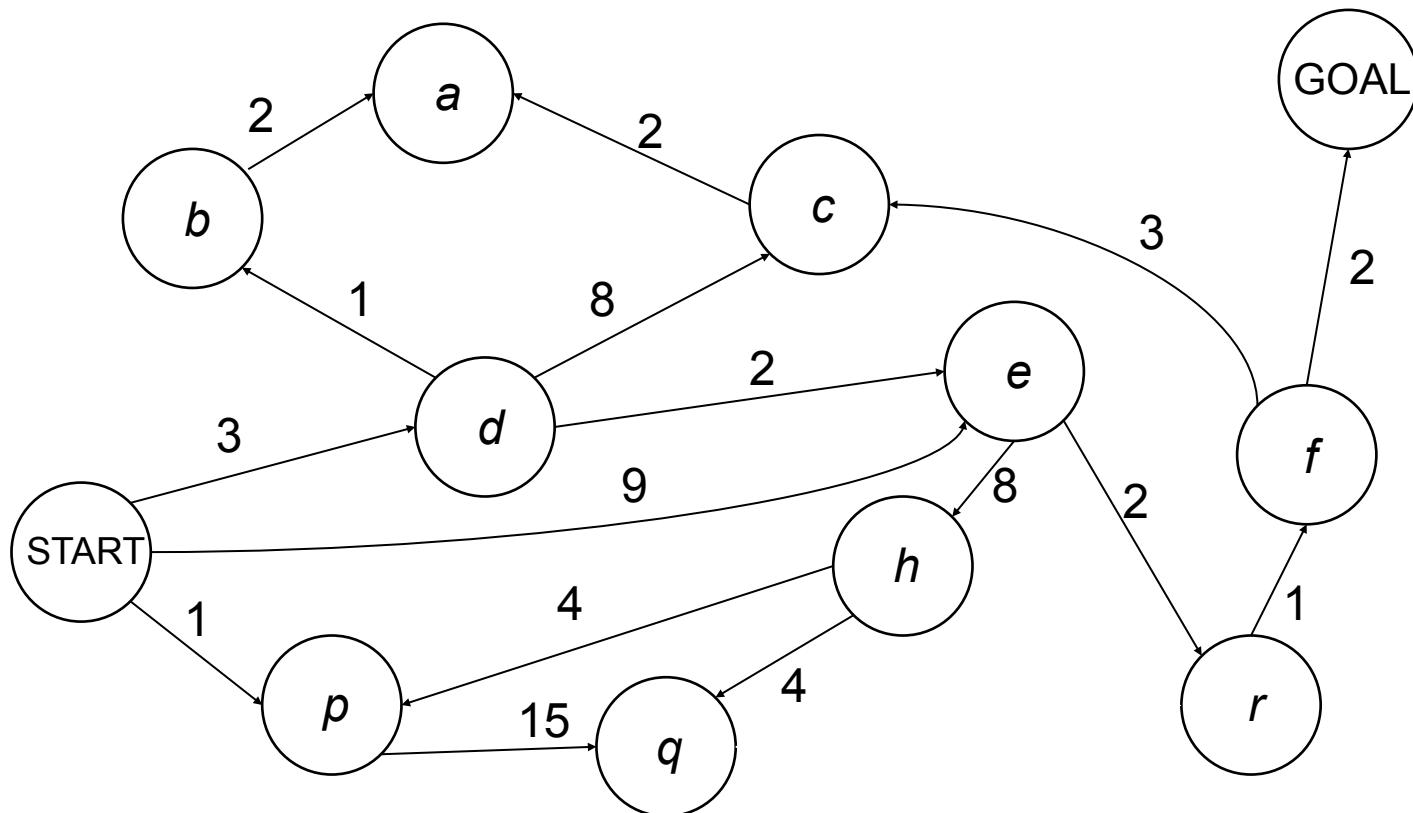
aka Deterministic Tree-based Planning
aka “Search”

finding the shortest path from start state to goal state

Goals for today

- Learn (or remember) basic ideas of breadth-first and depth-first search
 - their strengths and weaknesses
- Introduce Iterative-Deepening and Uniform-Cost Search
- Introduce Adversarial (minimax) Search with evaluation functions and alpha-beta pruning
 - try doing alpha-beta pruning yourself
- Gain some perspective on these search algorithms vs dynamic programming

Search Graph



Search Algorithm Properties

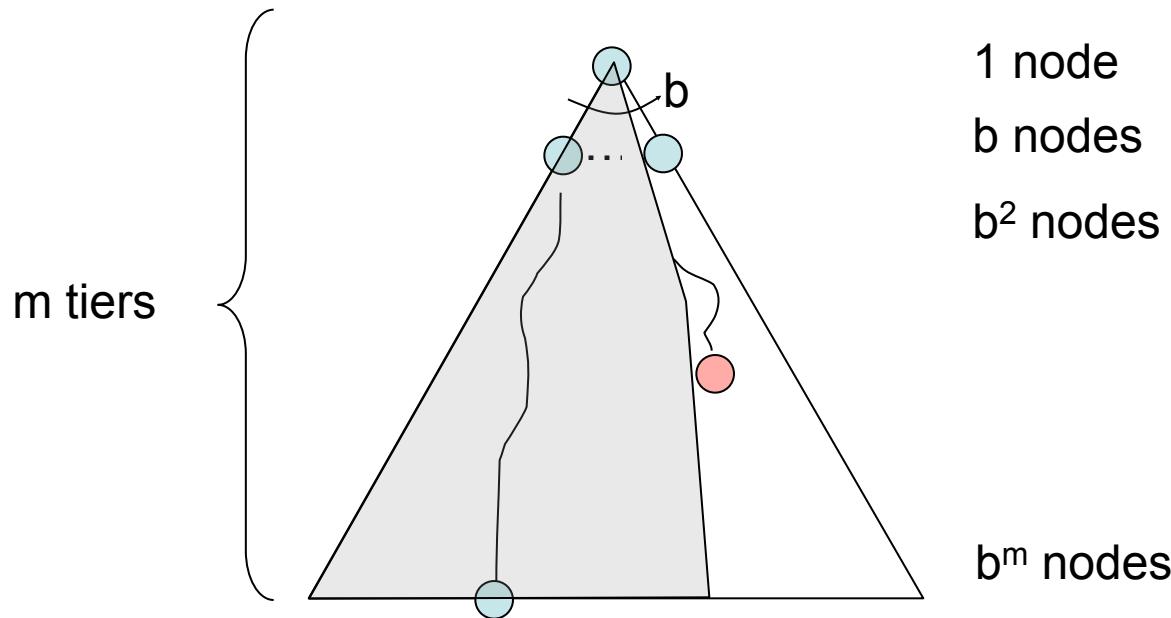
- **Complete?** Guaranteed to find a solution if one exists?
- **Optimal?** Guaranteed to find the least cost path?
- **Time complexity?**
- **Space complexity?**

Variables:

n	Number of states in the problem
b	The average branching factor B (the average number of successors)
s	Depth of the shallowest solution
m	Max depth of the search tree

Depth-First Search (DFS)

- With cycle checking, DFS is complete.

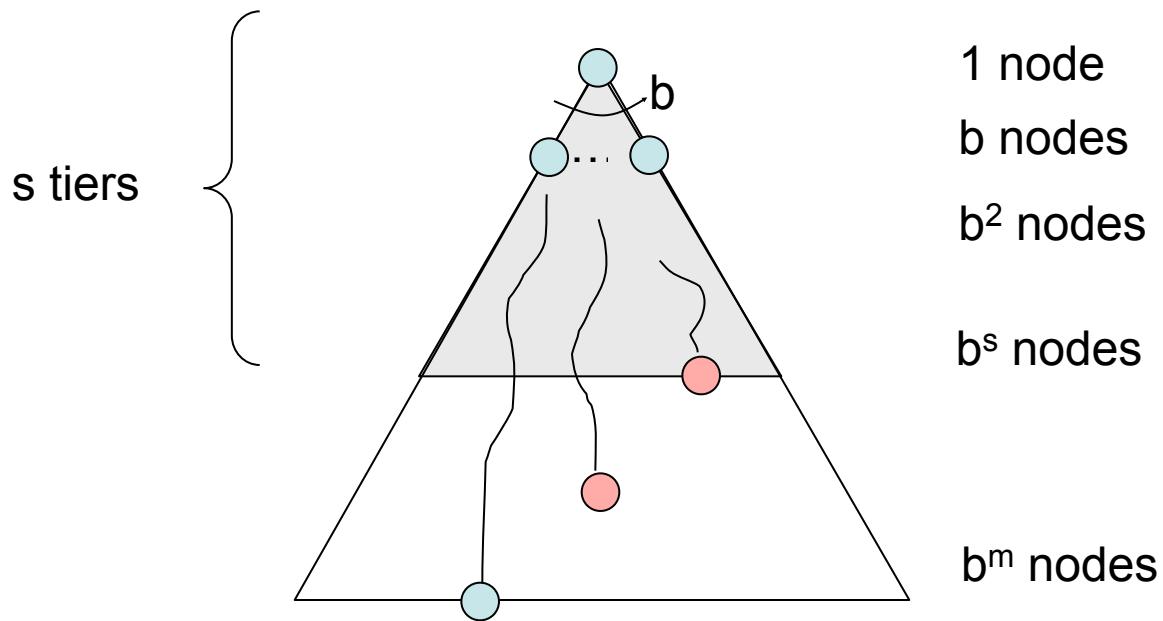


Algorithm	Complete	Optimal	Time	Space
DFS w/cycle checking	Y	N	$O(b^{m+1})$	$O(bm)$

- When is DFS optimal?

Breadth-First Search (BFS)

Algorithm	Complete	Optimal	Time	Space
DFS w/cycle checking	Y	N	$O(b^{m+1})$	$O(bm)$
BFS	Y	N*	$O(b^{s+1})$	$O(b^s)$



- When is BFS optimal?

Comparisons

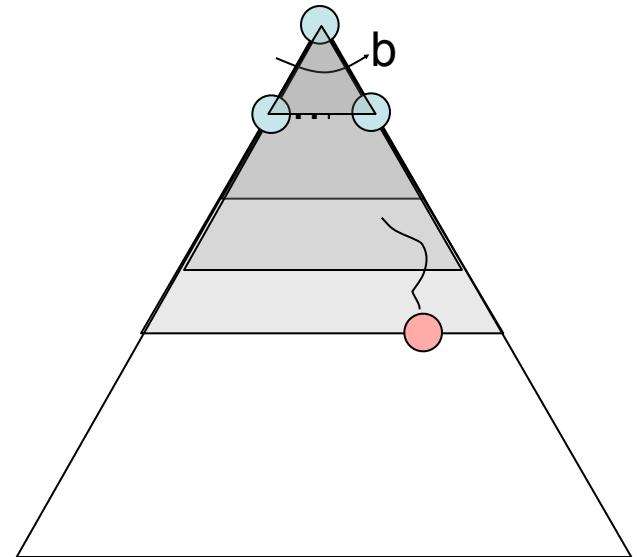
- When will BFS outperform DFS?
- When will DFS outperform BFS?

Iterative Deepening

Iterative deepening uses DFS as a subroutine:

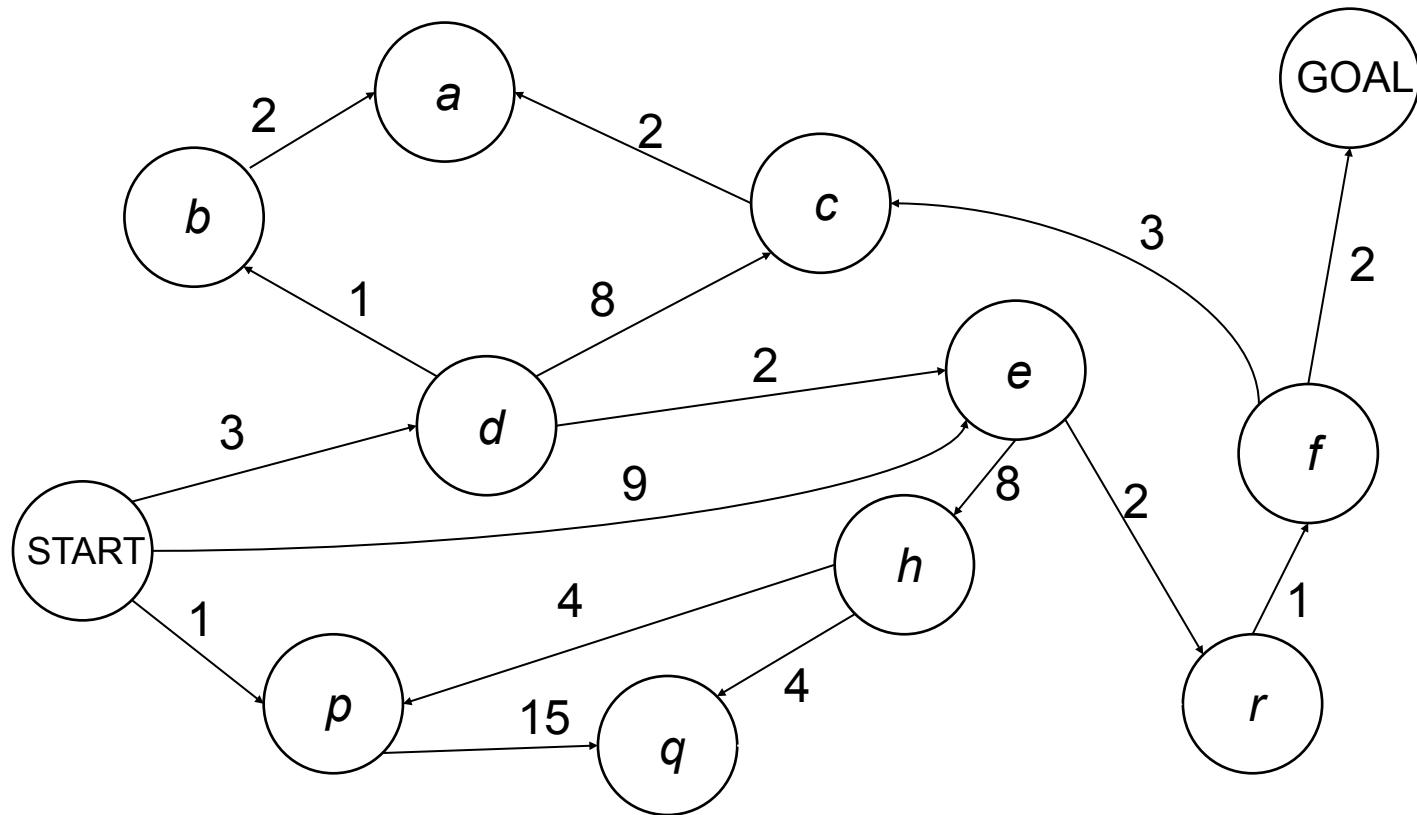
1. Do a DFS which only searches for paths of length 1 or less.
 2. If “1” failed, do a DFS which only searches paths of length 2 or less.
 3. If “2” failed, do a DFS which only searches paths of length 3 or less.

....and so on.



Algorithm		Complete	Optimal	Time	Space
DFS	w/cycle checking	Y	N	$O(b^{m+1})$	$O(bm)$
BFS		Y	N*	$O(b^{s+1})$	$O(b^s)$
ID		Y	N*	$O(b^{s+1})$	$O(bs)$

Costs on Actions



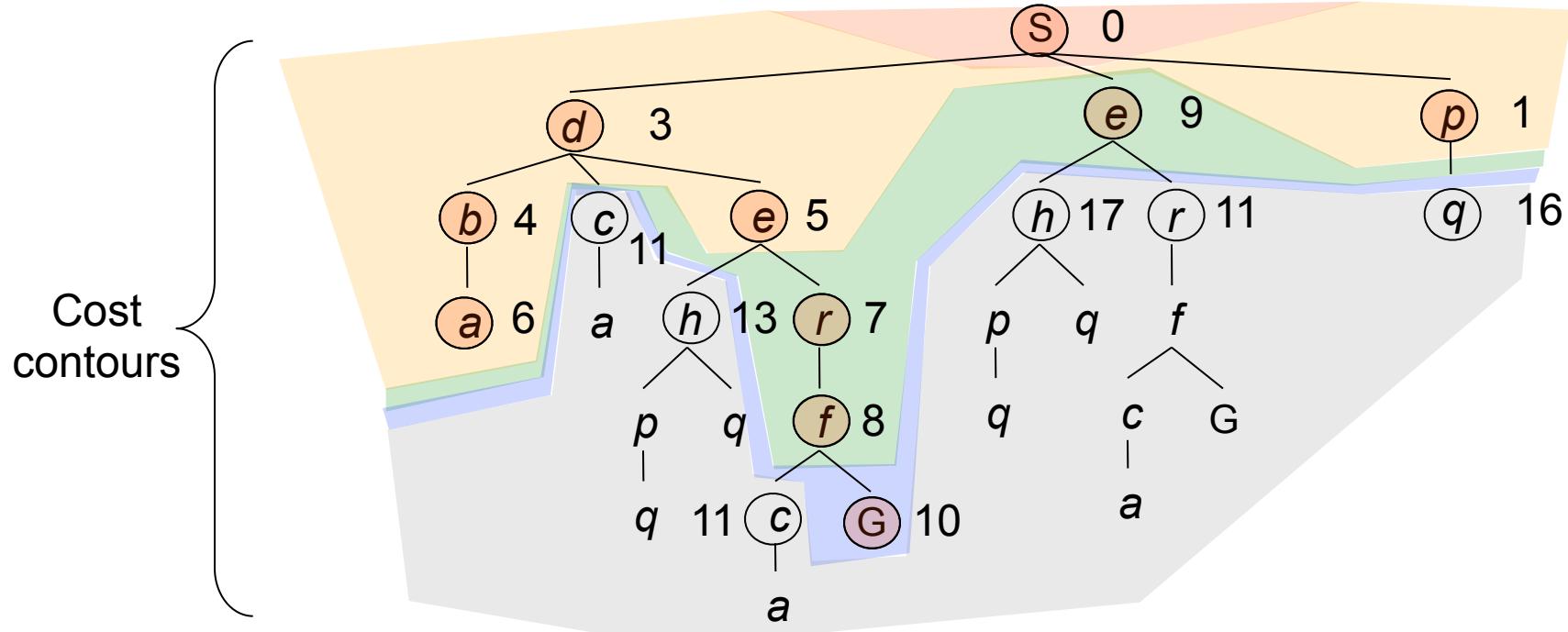
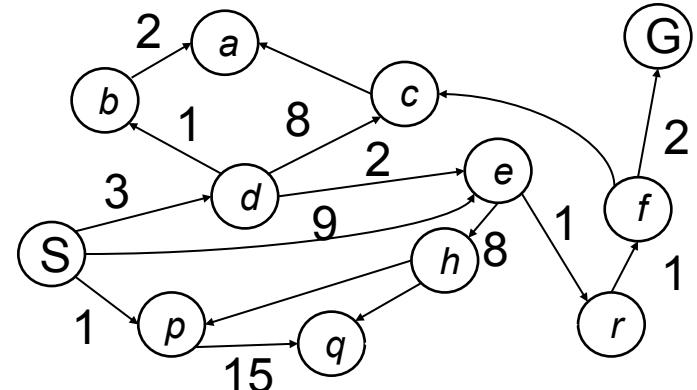
Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.

We will now cover an algorithm which does find the least-cost path.

Uniform Cost Search

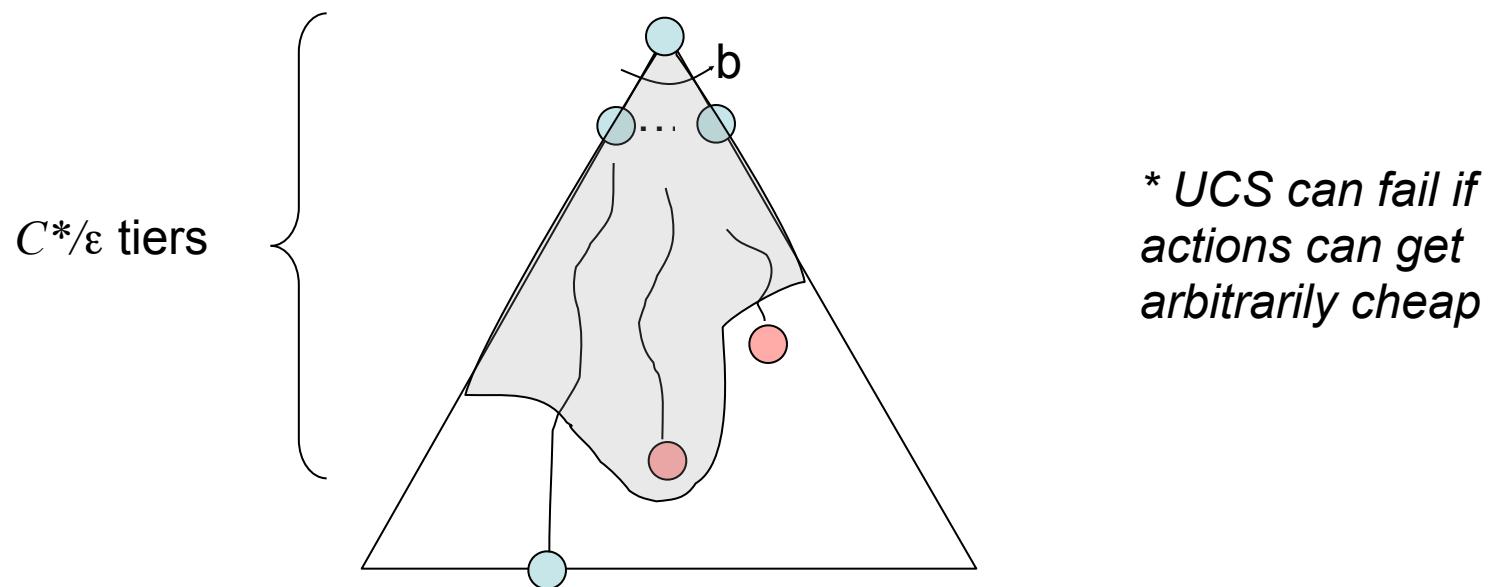
Expand cheapest node first:

Fringe is a priority queue



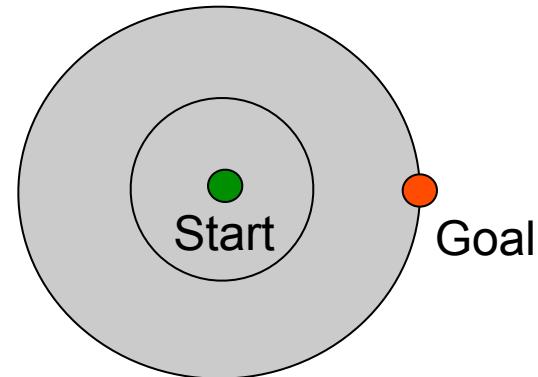
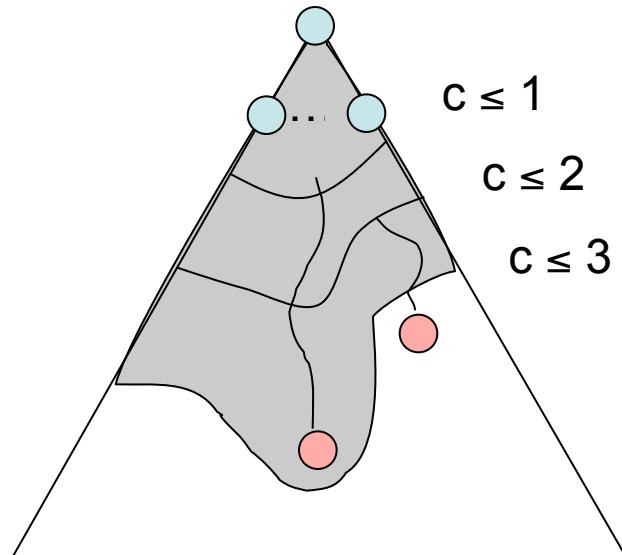
Uniform Cost Search

Algorithm	Complete	Optimal	Time	Space
DFS w/cycle checking	Y	N	$O(b^{m+1})$	$O(bm)$
BFS	Y	N	$O(b^{s+1})$	$O(b^s)$
UCS	Y^*	Y	$O(b^{C^*/\epsilon})$	$O(b^{C^*/\epsilon})$



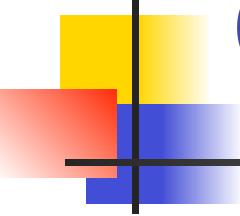
Uniform Cost Issues

- Remember: explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
 - Explores options in every “direction”
 - No information about goal location
 - “blind” search



Recap: Search

- **Search problem:**
 - States (configurations of the world)
 - Deterministic transitions: a function from states to lists of (next state, cost) pairs; drawn as a graph
 - Start state and goal test
- **Search tree:**
 - Nodes: represent plans for reaching states
 - Plans have costs (sum of action costs)
- **Search Algorithm:**
 - Systematically builds a search tree
 - Chooses an ordering of the fringe (unexplored nodes)

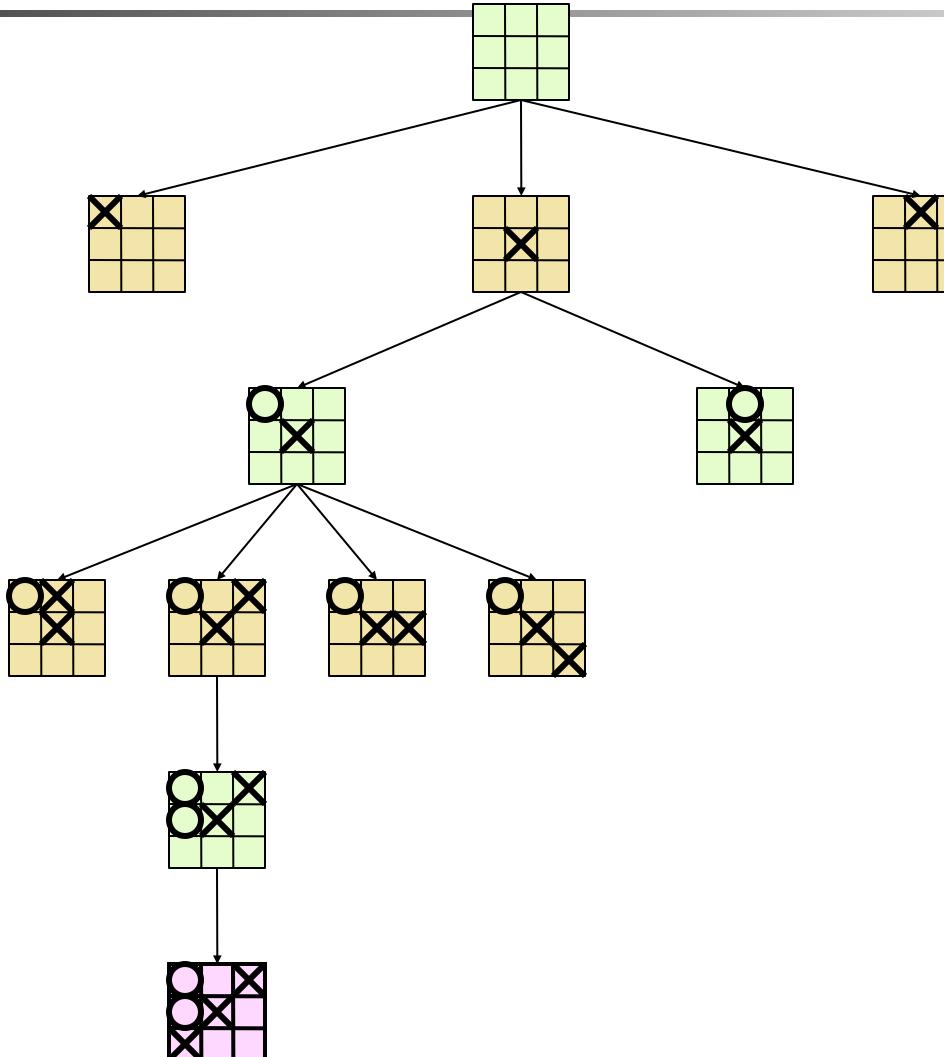


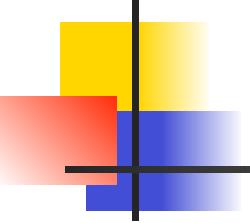
Game-tree search

- Approximate
 - Evaluation functions
 - “Anytime” algorithms such as iterative deepening
- Adversarial
 - Minimax search
 - Alpha-Beta pruning

based on work by R Greiner, D Lin, Jean-Claude Latombe, N Nilsson

Partial Game Search Tree for Tic-Tac-Toe

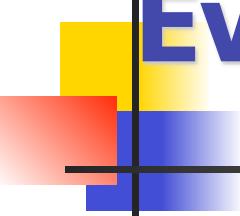




But... in general the search tree is too big to make it possible to reach the terminal states!

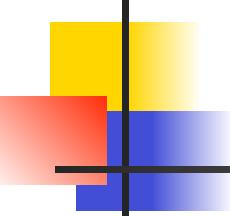
Examples:

- Checkers: $\sim 10^{40}$ nodes
- Chess: $\sim 10^{120}$ nodes



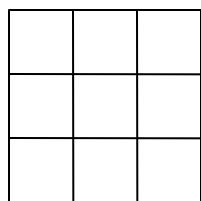
Evaluation Function of a State

- $e(s) = +\infty$ if s is a win for MAX
- $e(s) = -\infty$ if s is a win for MIN
- $e(s)$ = a measure of how “favorable”
is s for MAX
 - > 0 if s is considered favorable to MAX
 - < 0 otherwise

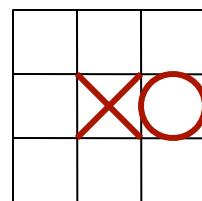


Example: Tic-Tac-Toe

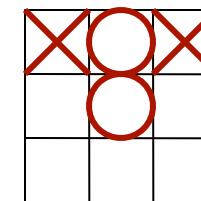
$e(s) =$ number of rows, columns, and diagonals open for MAX
- number of rows, columns, and diagonals open for MIN



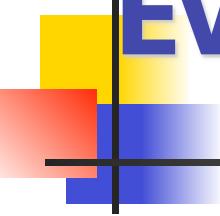
$$8-8 = 0$$



$$6-4 = 2$$



$$3-3 = 0$$

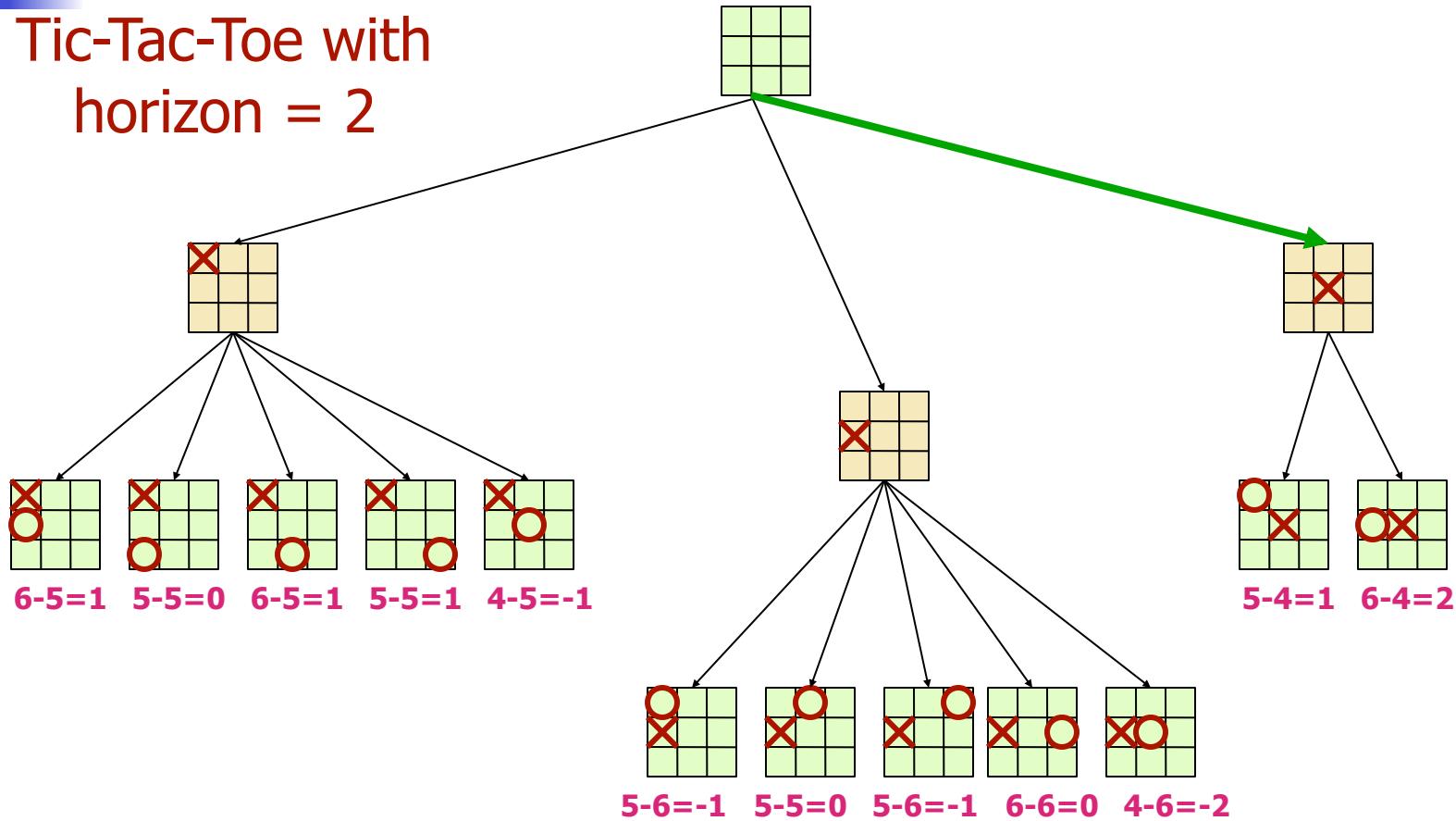


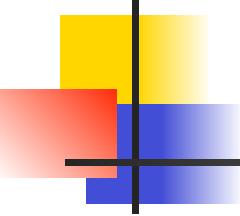
Evaluation Function for chess

- $e(s) = \text{weighted sum of feature}$
$$e(s) = w_1x_1 + w_2x_2 + \dots + w_kx_k$$
- Features
 - # of white pawns – # of black pawns
 - # of white bishops – # of black bishops
 - # of white rooks – # of black rooks
 - ...
- Weights
 - 1 for pawns, 3 for bishops, 5 for rooks

Example

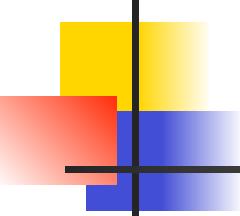
Tic-Tac-Toe with
horizon = 2





Minimax

- Achieves “Perfect” play for deterministic, perfect-information games
- Idea: choose move leading to position with highest minimax value
 - best achievable payoff against best play

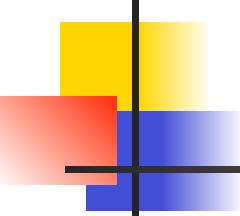


Minimax (back of the envelope)

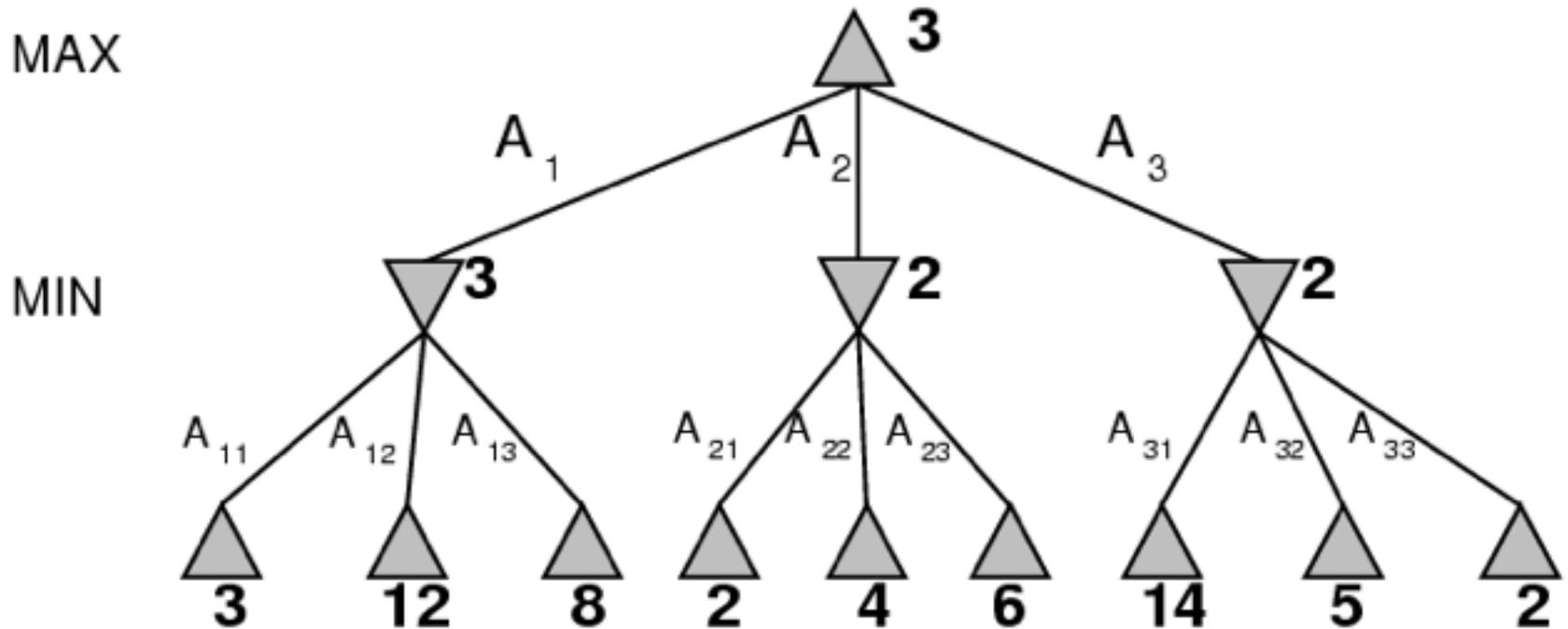
- Does minimax work in practice?

In chess: we can do about $b^m = 10^9$
thus, if $b=35$, then $m=6$
but 6-ply lookahead is a weak chess player!

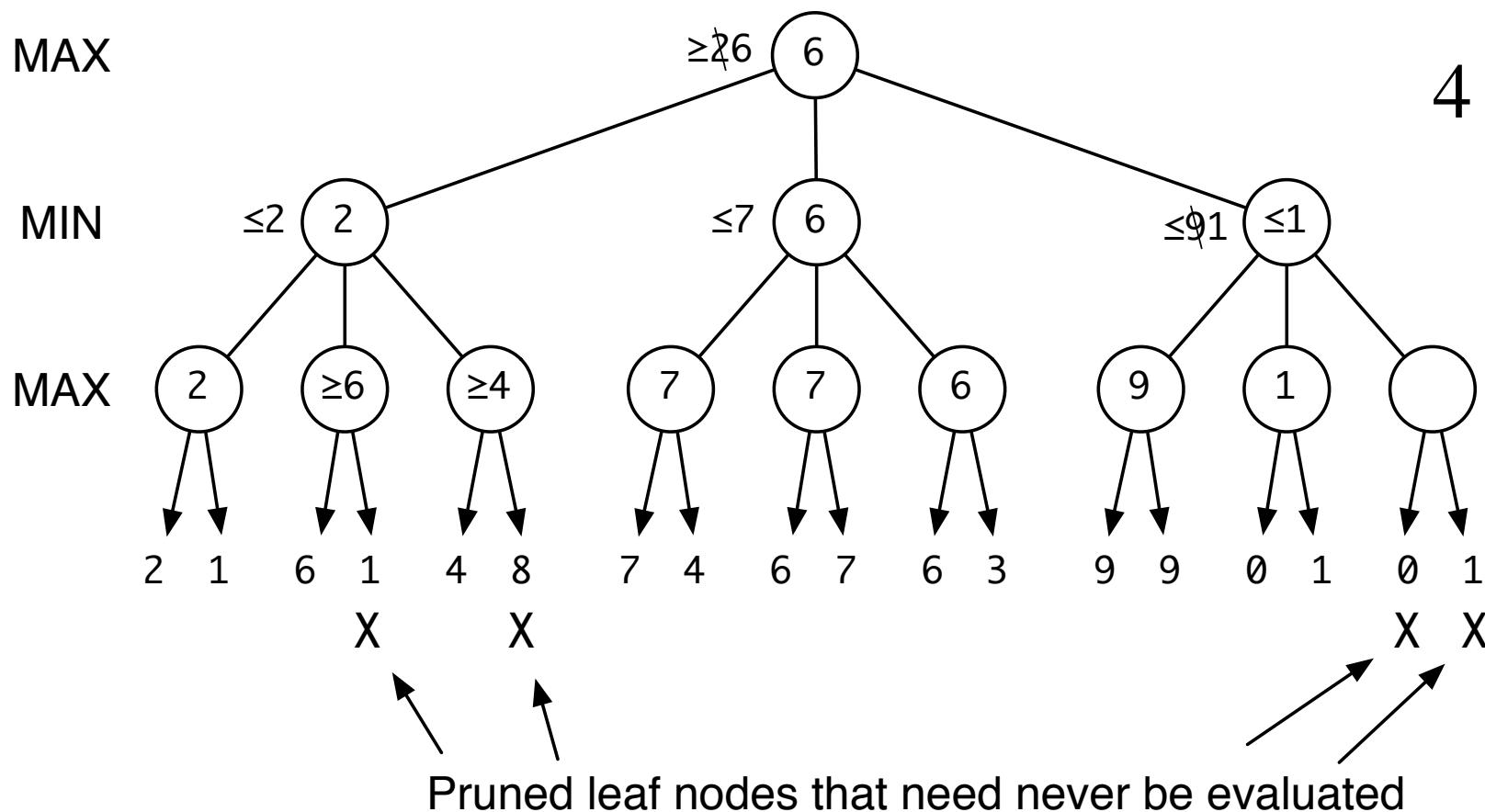
- 4-ply = human novice
8-ply = slow PC, human master
12-ply = Deep Blue, human grandmaster



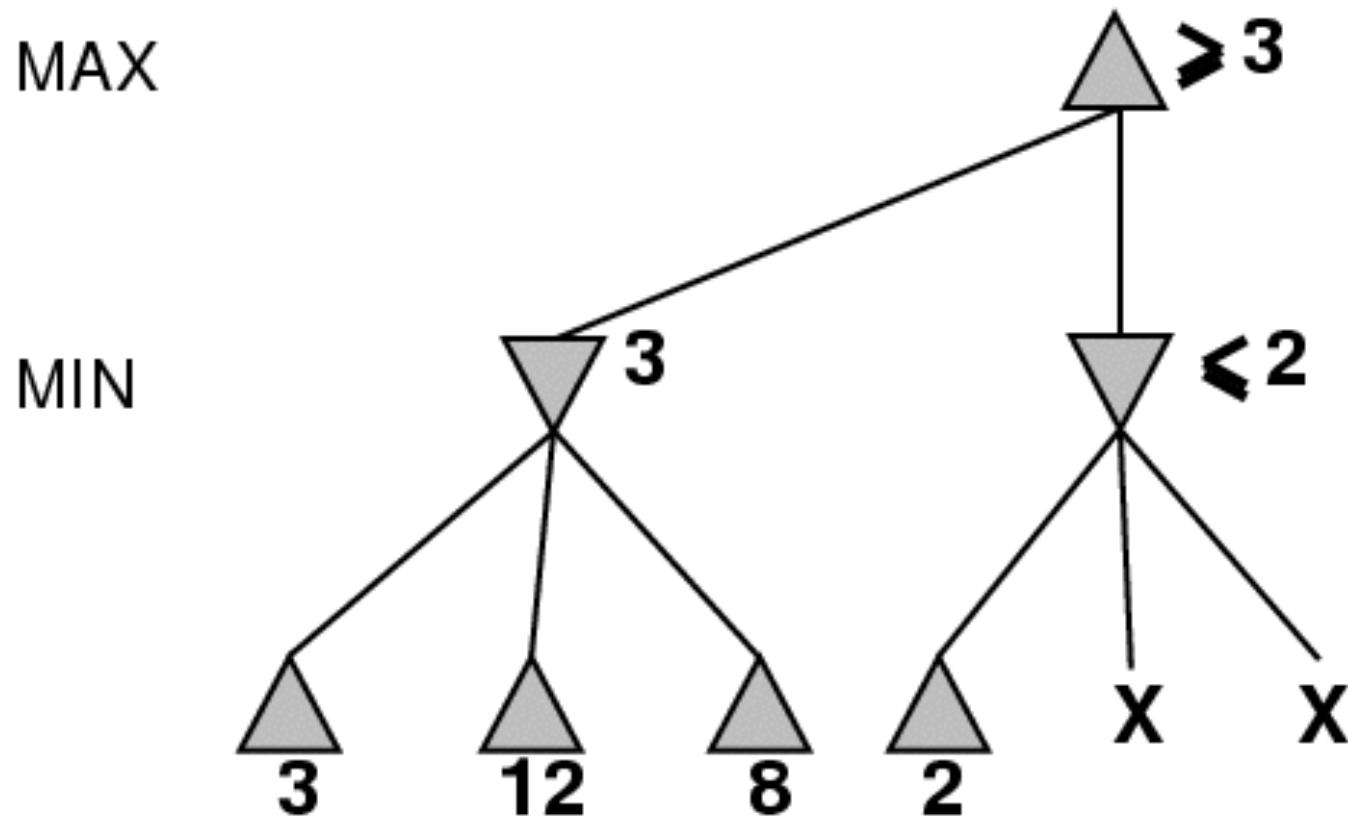
Example: a 2-ply game

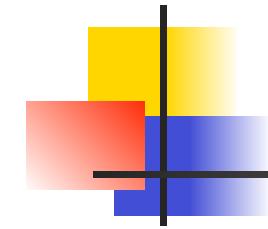


alpha-beta example



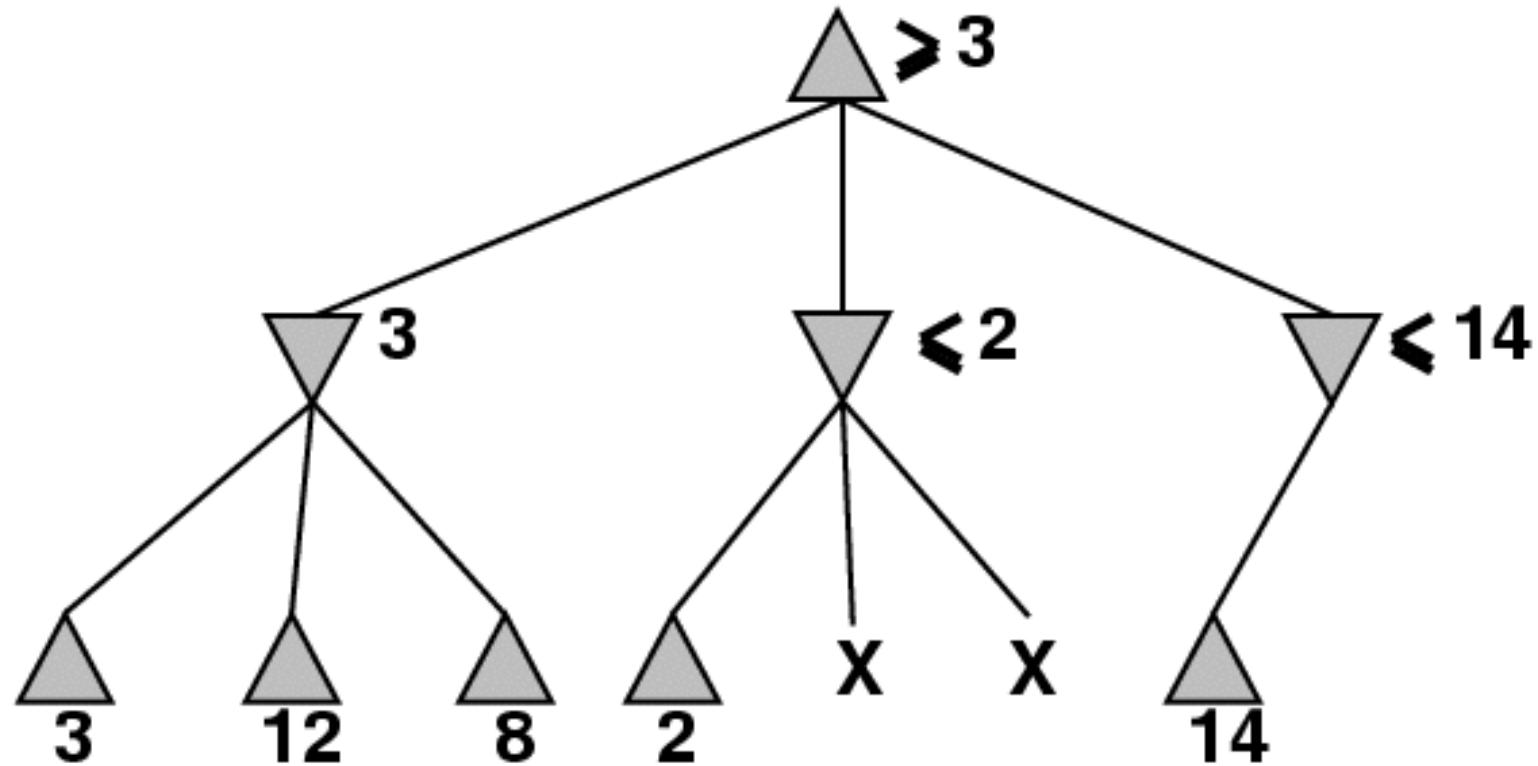
α - β Pruning

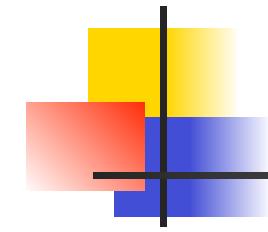




MAX

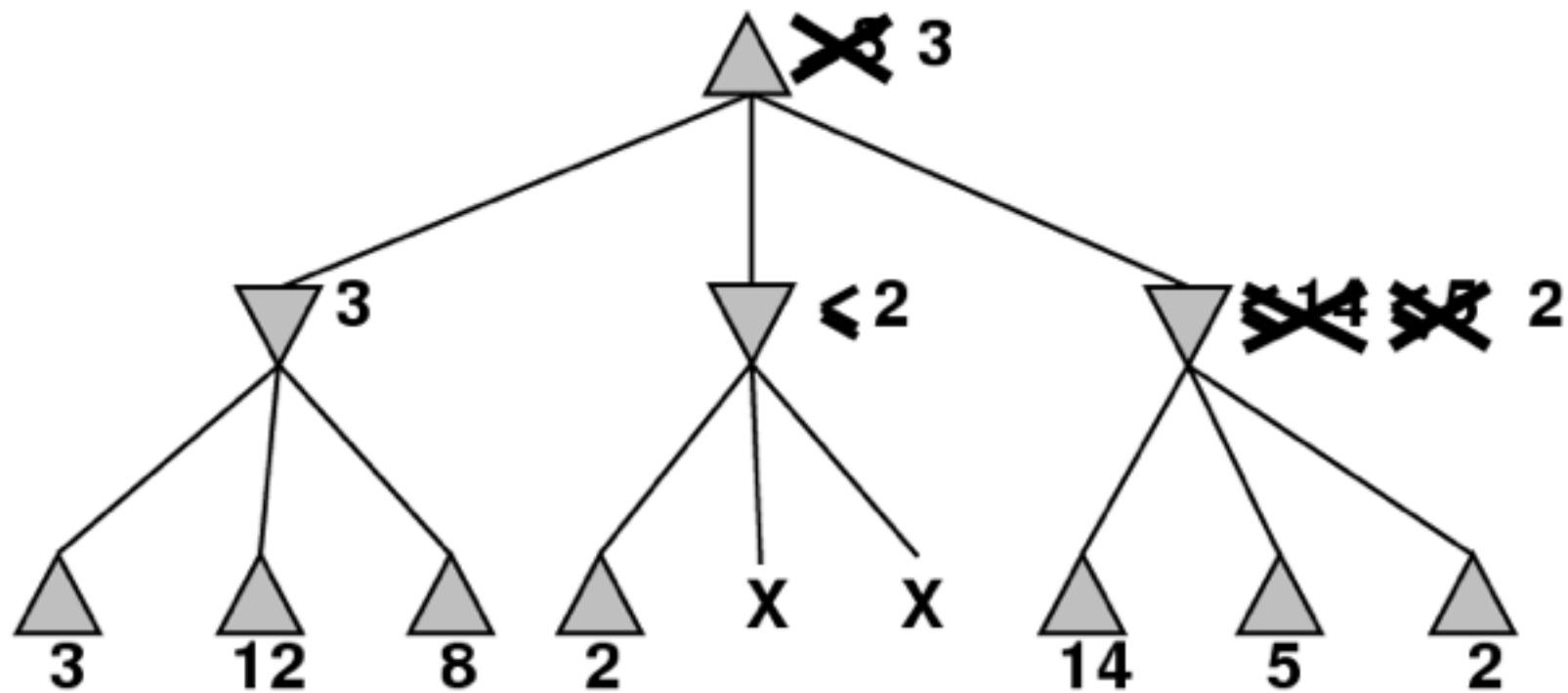
MIN



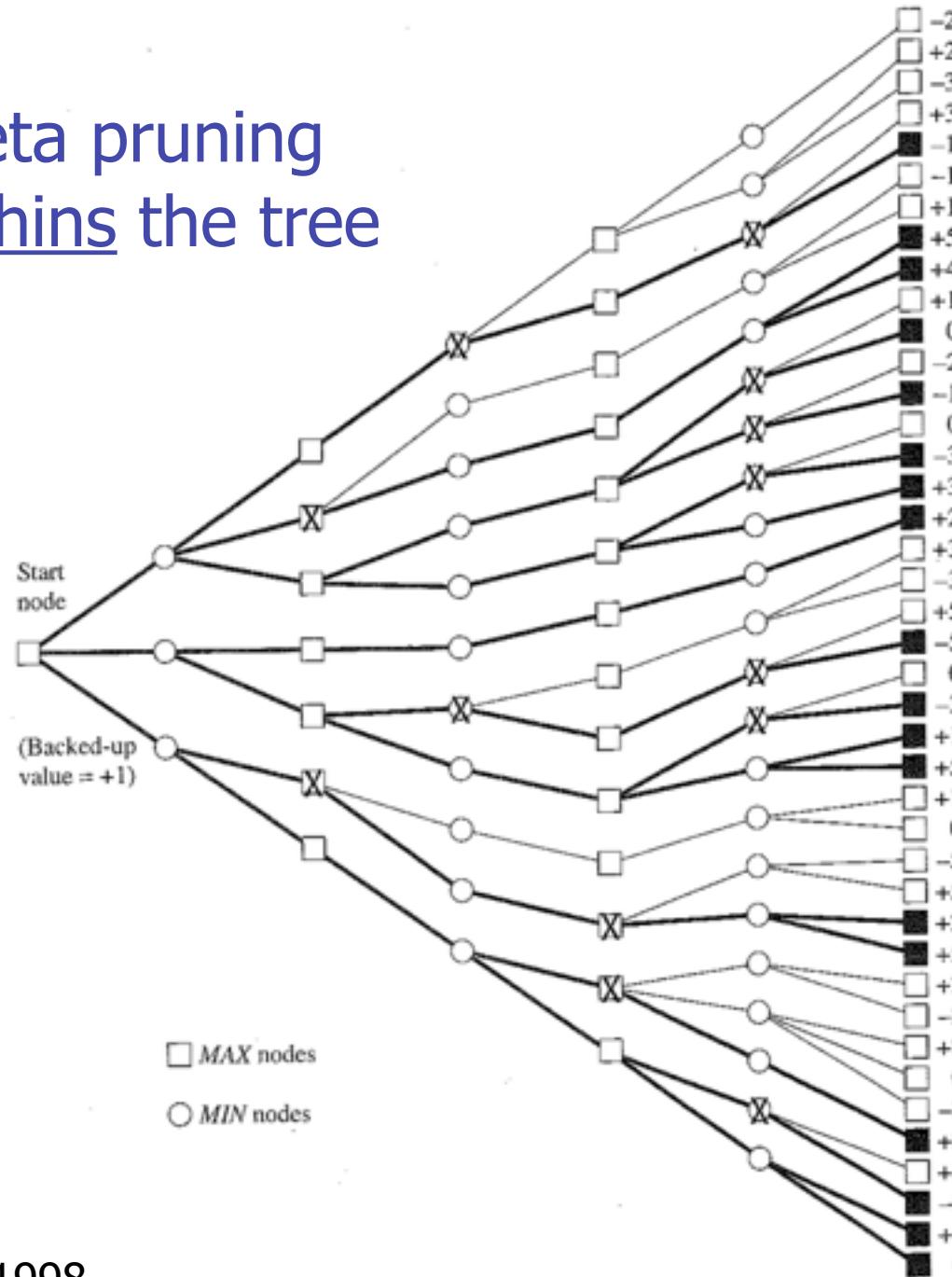


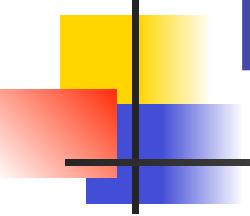
MAX

MIN



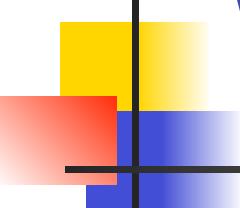
Alpha-beta pruning greatly thins the tree



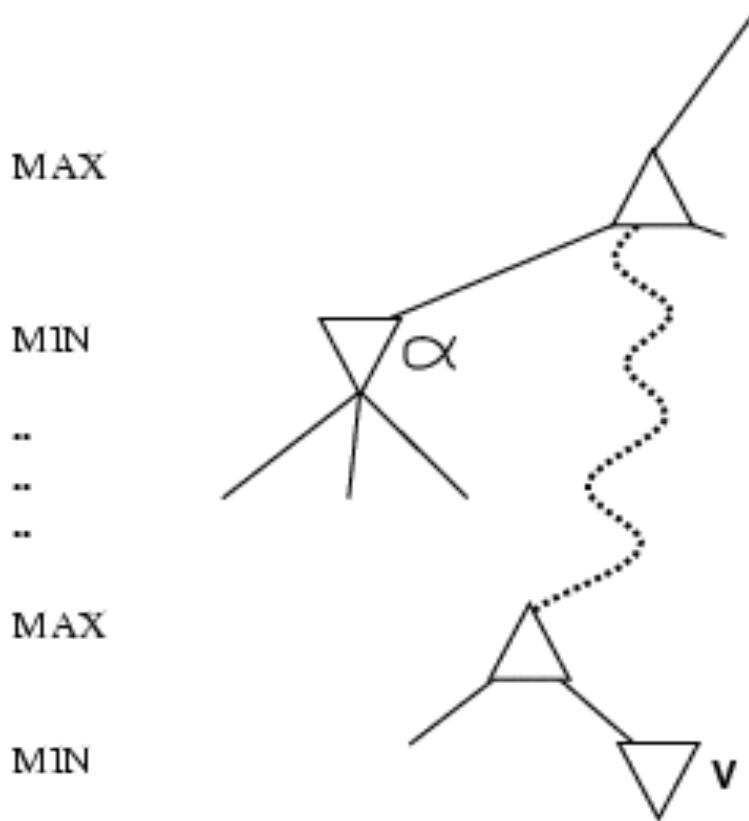


Properties of α - β Search

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With “perfect ordering”:
 - time complexity = $O(b^{m/2})$
 - doubles depth of search
 - can easily reach depth 12
 - play grandmaster chess!

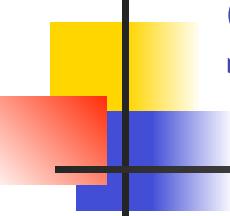


Why is it called α - β ?



- α = best value (for max) found so far, off the current path
- If v is worse than α , max will avoid it
 - prune that branch
- Define β similarly for min

- Could you use BFS for game-tree search?
- No, BFS is inherently a 'forward' search
- Whereas DFS is inherently 'backward'
- Dynamic programming is also 'backward'
- In stochastic or adversarial settings it seems to me necessary to search backward



Summary

- There are lots of different kinds of search
 - With different optimizations and guarantees
- All search involves computing how good it is to be in each state
- The problem characteristics determine which search method is best
- All today's methods are blind search, with no hint or guess of what comes ahead, unlike heuristic search

Heuristic Single-Agent Search

Robert Holte

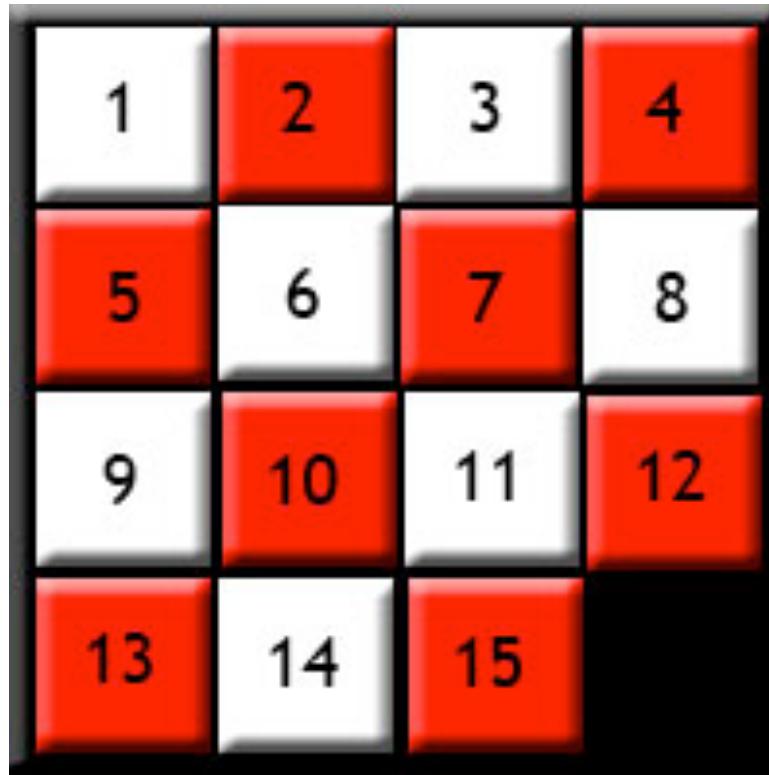


A Heuristic Function Estimates

Distance to Goal



Heuristics Speed up Search



10,461,394,944,000 states

heuristic search examines 36,000

Example Heuristic Functions

- $h(n)$ estimates cost of cheapest path from node n to goal node
- Example: 8-puzzle

5		8
4	2	1
7	3	6

n

1	2	3
4	5	6
7	8	

goal

$$h_1(n) = \text{number of misplaced tiles} \\ = 6$$

Example Heuristic Functions

- $h(n)$ estimates cost of cheapest path from node n to goal node
- Example: 8-puzzle

5		8
4	2	1
7	3	6

n

1	2	3
4	5	6
7	8	

goal

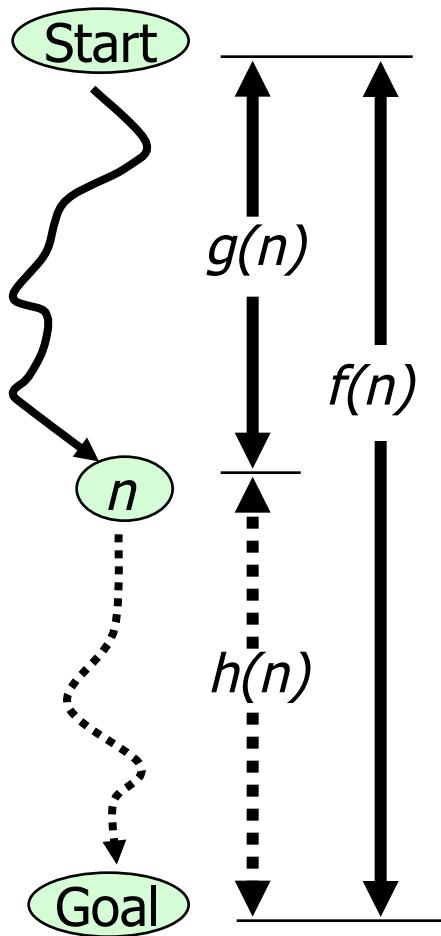
$$h_1(n) = \text{number of misplaced tiles} \\ = 6$$

$$h_2(n) = \text{sum of the distances of every tile to its goal position} \\ = 3 + 1 + 3 + 0 + 2 + 1 + 0 + 3 \\ = 13$$

How to Use a Heuristic Function to Speed Up Search ?



Notation



- $g(n)$ = distance from start to node n along our current path (not necessarily optimal)
- $h(n)$ = estimated distance from n to goal
- $f(n) = g(n)+h(n)$ = estimated distance from start to goal via n (using our current path to n)

More Notation

- $g^*(n)$ = true distance from start to node n
- $h^*(n)$ = true distance from n to goal
- $f^*(n)$ = true distance from start to goal via n
 $= g^*(n) + h^*(n)$

How to Use a Heuristic Function to Speed Up Search ?



(1) Pruning

- If we have a bound B on total solution cost (e.g. iterative deepening) we can prune n from the search space if $f(n) > B$.
- But this might prune all the optimal paths. One condition that makes pruning safe is ...
- $g^*(n)+h(n) \leq f^*(n)$, which is true if $h(n)...$
- never overestimates, $h(n) \leq h^*(n)$.
- Such a heuristic is called “admissible”.

Iterative Deepening Heuristic

Search (IDA*)

- Using this pruning idea, when you generate a node n you only explore beneath it (with a recursive call) if $f(n) \leq B$, the current cost bound (not depth bound).
- How do you update the cost bound from one iteration to the next ?

(2) Node Ordering

Use the heuristic to decide which node to expand next. Two main options.

1. Choose the node with minimum $h(n)$.
Uniform-cost search using this selection rule is called “Pure Heuristic Search”
2. Choose the node with minimum $f(n)$.
Uniform-cost search using this selection rule is called A*.

Terminology

- Expanding a state means generating its successors (children).
- A state is open if it has been generated but not expanded.
- A state is closed if it has been expanded.
- Open list = data structure holding all currently open states.
- Closed list = data structure indicating which states are closed.

Dijkstra's Algorithm*

- Put $(\text{start}, 0)$ in OPEN
- Repeat
 - If OPEN is empty, exit with failure
 - Select $(n, g(n))$ on OPEN with minimum $g(n)$
 - If n is the goal, return path from start to goal
 - Move $(n, g(n))$ from OPEN to CLOSED
 - For each successor x of n :
 - If $(x, g(x))$ is on OPEN and $g(x) > g(n) + \text{cost}(n, x)$ then replace $(x, g(x))$ on OPEN with $(x, g(n) + \text{cost}(n, x))$
 - If $(x, _)$ is on neither OPEN nor CLOSED then add $(x, g(n) + \text{cost}(n, x))$ to OPEN

A* Algorithm

- Put $(\text{start}, \mathbf{f}(\text{start}))$ in OPEN
- Repeat
 - If OPEN is empty, exit with failure
 - Select $(n, \mathbf{f}(n))$ on OPEN with minimum $\mathbf{f}(n)$
 - If n is the goal, return path from start to goal
 - Move $(n, \mathbf{f}(n))$ from OPEN to CLOSED
 - For each successor x of n :
 - If $(x, \mathbf{f}(x))$ is on OPEN and $\mathbf{f}(x) > g(n) + \text{cost}(n, x) + \mathbf{h}(x)$ then replace $(x, \mathbf{f}(x))$ on OPEN with $(x, g(n) + \text{cost}(n, x) + \mathbf{h}(x))$
 - If $(x, \underline{})$ is on neither OPEN nor CLOSED then add $(x, g(n) + \text{cost}(n, x) + \mathbf{h}(x))$ to OPEN
 - If $(x, \mathbf{f}(x))$ is on CLOSED and $\mathbf{f}(x) > g(n) + \text{cost}(n, x) + \mathbf{h}(x)$ then remove $(x, \mathbf{f}(x))$ from CLOSED and add $(x, g(n) + \text{cost}(n, x) + \mathbf{h}(x))$ to OPEN

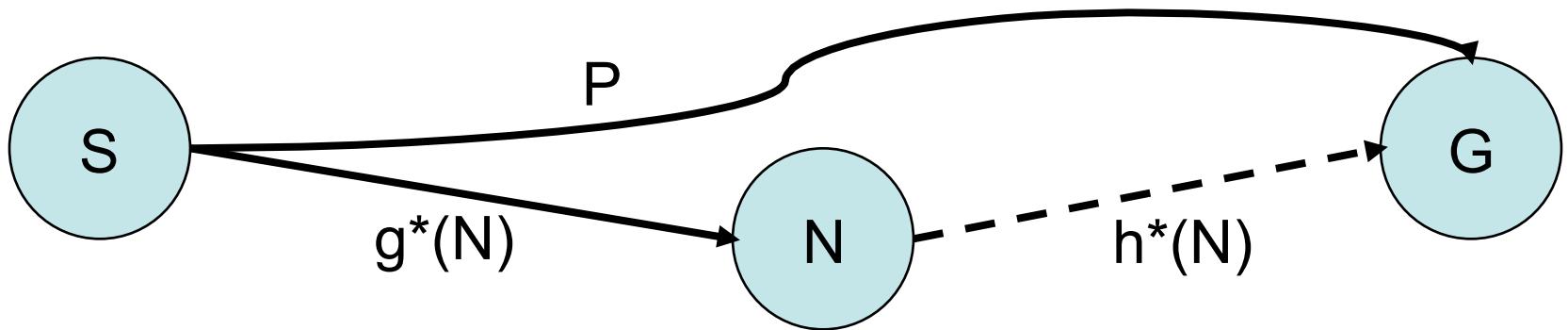
Question

When these algorithms stop (first remove the goal from Open), under what conditions are we guaranteed that this path to the goal is optimal ?

Dijkstra: all edge weights are non-negative

A^{*}: impose conditions on the heuristic

Condition on the Heuristic



Require $\langle N, g^*(N)+h(N) \rangle$ lower cost than $\langle G, P \rangle$

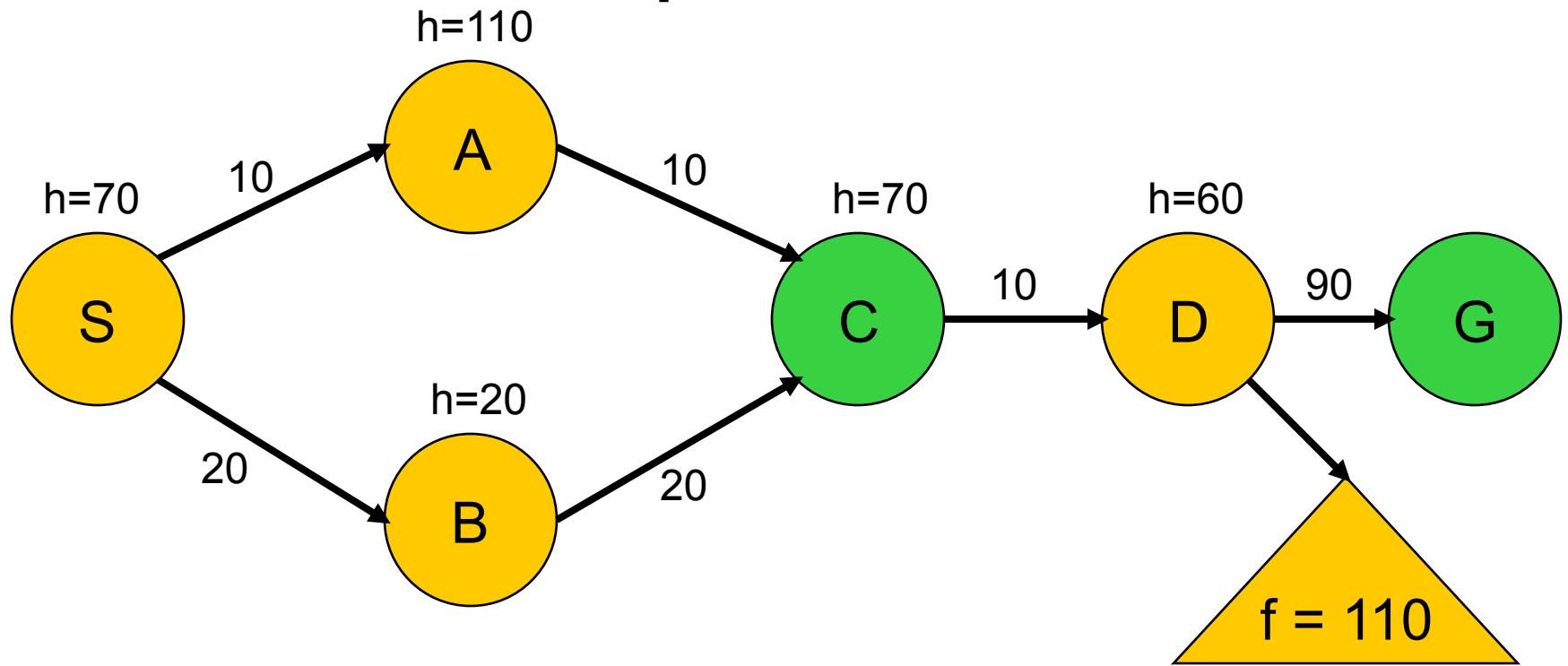
$$g^*(N)+h(N) < P$$

$$\Leftrightarrow h(N) < P - g^*(N)$$

$$\Leftrightarrow h(N) \leq h^*(N) \quad (\text{because } h^*(N) < P - g^*(N))$$

Admissible Heuristic \Rightarrow A* stops with an optimal path to goal

A* must re-open closed nodes



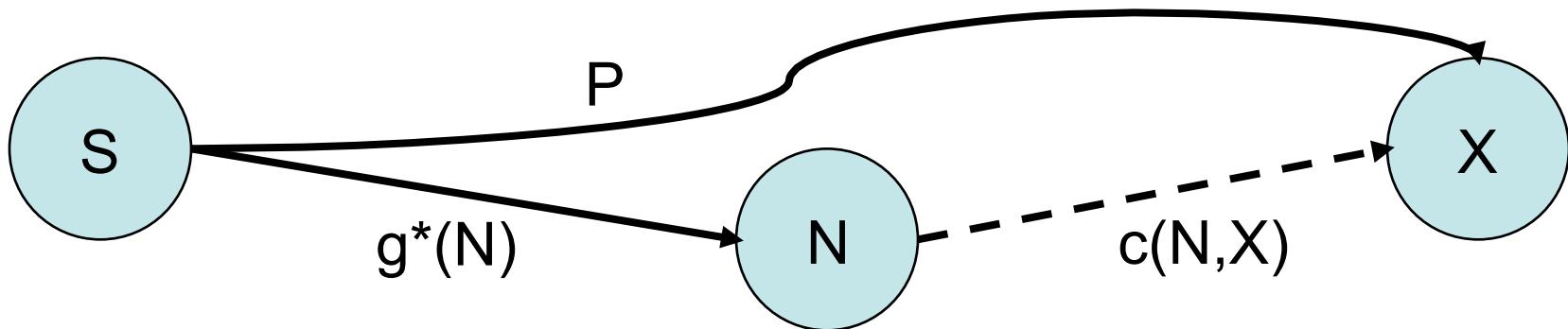
OPEN: (G,140), (C,90)

CLOSED: (S,70), (B,40), (C,110) , (D,110), ... (A, 120)

How Bad Can This Be ?

- If N is the number of distinct nodes that A^* expands, the total number of node expansions could be as bad as $O(2^N)$.
- Some A^* variations reduce the worst-case to $O(N^2)$.
- Alternatively, if the heuristic has a special property, basic A^* never re-opens a node once it is closed, so is $O(N)$.

Consistent Heuristic



Require $\langle N, g^*(N)+h(N) \rangle$ lower cost than $\langle X, P+h(X) \rangle$

$$g^*(N)+h(N) < P+h(X)$$

$$\Leftrightarrow h(N) - h(X) < P - g^*(N)$$

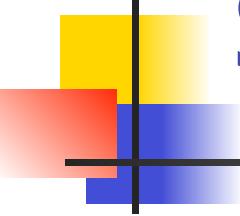
$$\Leftrightarrow h(N) - h(X) \leq c(N,X) \quad (\text{because } c(N,X) < P - g^*(N))$$

A heuristic is consistent if $h(N) \leq c(N,X) + h(X)$
for all N and **all** X.

Consistent \Rightarrow first path to X off Open is optimal for all X

The Need for Automatically Generated Heuristics





Summary

- There are lots of different kinds of search
 - With different optimizations and guarantees
- All involve planning – using your knowledge of the world's dynamics to anticipate the consequences of your action, and then picking the best
- All search involves computing how good it is to be in each state
 - And benefits from a good initial estimate

From Deep Blue to Monte Carlo: An Update on Game Tree Research

Akihiro Kishimoto and Martin Müller

AAAI-14 Tutorial 5:
Monte Carlo Tree Search

Presenter:
Martin Müller, University of Alberta

Tutorial 5 – MCTS - Contents

Part 1:

- Limitations of alphabeta and PNS
- Simulations as evaluation replacement
- Bandits, UCB and UCT
- Monte Carlo Tree Search (MCTS)

Tutorial 5 – MCTS - Contents

Part 2:

- MCTS enhancements: RAVE and prior knowledge
- Parallel MCTS
- Applications
- Research challenges, ongoing work

Go: a Failure for AlphaBeta

- ↗ Game of Go
- ↗ Decades of Research on knowledge-based and alphabeta approaches
- ↗ Level weak to intermediate
- ↗ AlphaBeta works much less well than in many other games
- ↗ Why?

Problems for AlphaBeta in GO

- ↗ Reason usually given: Depth and width of game tree
 - ↗ 250 moves on average
 - ↗ game length > 200 moves
- ↗ **Real reason: Lack of good evaluation function**
 - ↗ Too subtle to model: very similar looking positions can have completely different outcome
 - ↗ Material is mostly irrelevant
 - ↗ Stones can remain on the board long after they “die”
 - ↗ Finding safe stones and estimating territories is hard

Monte Carlo Methods to the Rescue!

- Hugely successful
- Backgammon (Tesauro 1995)
- Go (many)
- Amazons, Havannah, Lines of Action, ...
- Application to deterministic games pretty recent
(less than 10 years)
- Explosion in interest, applications far beyond games
- Planning, motion planning, optimization, finance,
energy management,...

Brief History of Monte Carlo Methods

- ↗ 1940's – now Popular in Physics, Economics, ...
to simulate complex systems
- ↗ 1990 (Abramson 1990) expected-outcome
Brügmann, *Gobble*
- ↗ 1993 Bouzy, Monte Carlo experiments
- ↗ 2003 – 05 Coulom, *Crazy Stone*, **MCTS**
- ↗ 2006 (Kocsis & Szepesvari2006) UCT
- ↗ 2007 – now *MoGo*, *Zen*, *Fuego*, many others
- ↗ 2012 – now MCTS survey paper (Browne et al 2012);
huge number of applications

Idea: Monte Carlo Simulation

- ↗ No evaluation function? No problem!
- ↗ Simulate rest of game using random moves (easy)
- ↗ Score the game at the end (easy)
- ↗ Use that as evaluation (hmm, **but...**)

The GIGO Principle

- Garbage In, Garbage Out
- Even the best algorithms do not work if the input data is bad
- How can we gain any information from playing random games?

Well, it Works!

- For many games, anyway
- Go, NoGo, Lines of Action, Amazons, Konane, DiskConnect, ..., ...
- Even random moves often preserve *some* difference between a good position and a bad one
- The rest is statistics...
- ...well, not quite.

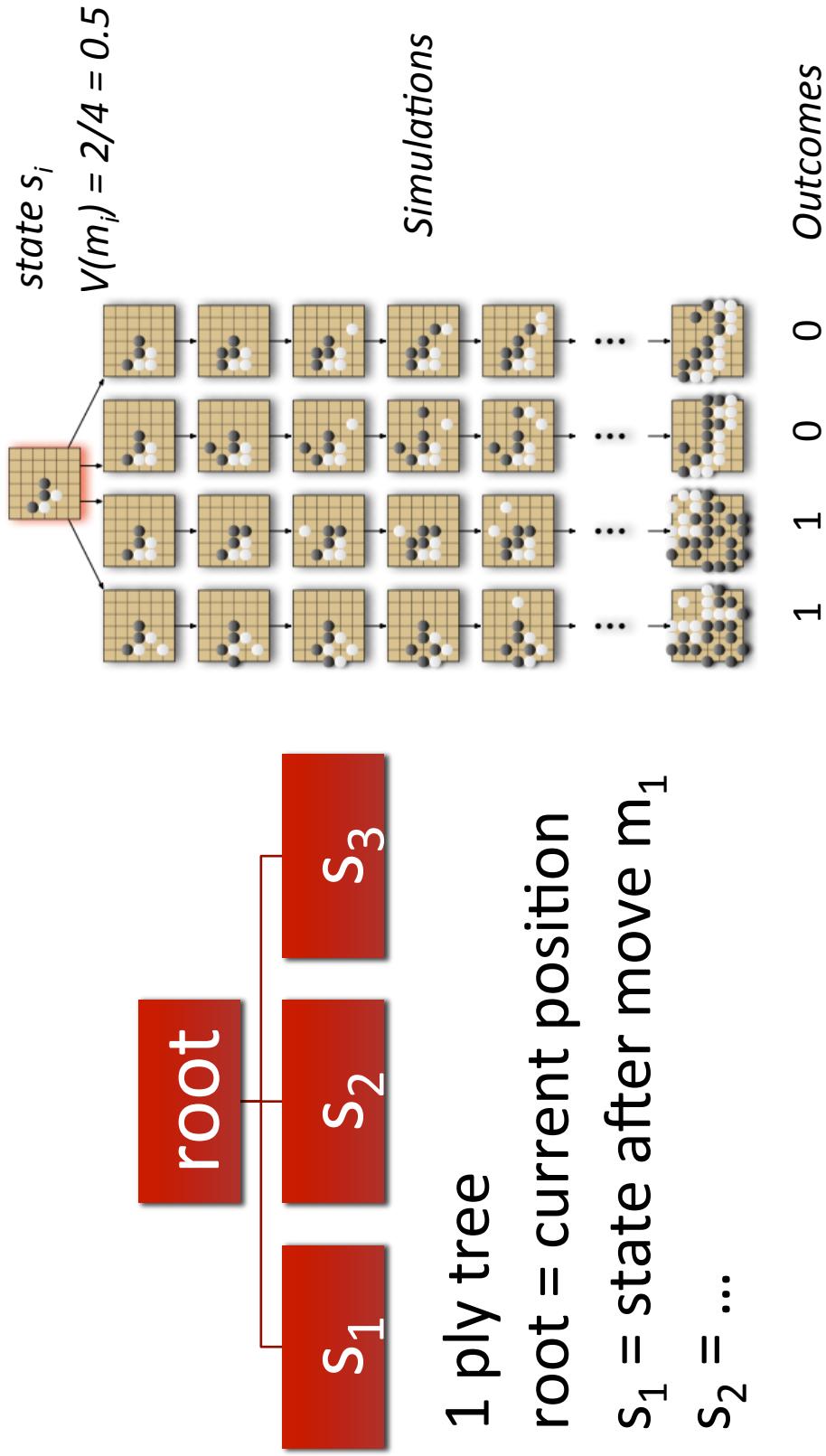
(Very) Basic Monte Carlo Search

- Play lots of random games
 - start with each possible legal move
- Keep winning statistics
 - Separately for each starting move
- Keep going as long as you have time, then...
- Play move with best winning percentage

Simulation Example in NoGo

- Demo using *GoGui* and *BobNoGo* program
- Random legal moves
- End of game when *ToPlay* has no move (loss)
- Evaluate:
 - +1 for win for current player
 - 0 for loss

Example – Basic Monte Carlo Search



Example for NoGo

- Demo for NoGo
- 1 ply search plus random simulations
- Show winning percentages for different first moves

Evaluation

- ↗ Surprisingly good e.g. in Go - much better than random or simple knowledge-based players
- ↗ Still limited
- ↗ Prefers moves that work “on average”
- ↗ Often these moves fail against the best response
- ↗ Likes “silly threats”

Improving the Monte Carlo Approach

- Add a game tree search (Monte Carlo Tree Search)
 - Major new game tree search algorithm
- Improved, better-than-random simulations
 - Mostly game-specific
- Add statistics over move quality
 - RAVE, AMAF
- Add knowledge in the game tree
 - human knowledge
 - machine-learnt knowledge

Add game tree search (Monte Carlo Tree Search)

- Naïve approach and why it fails
- Bandits and Bandit algorithms
 - Regret, exploration-exploitation, UCB algorithm
- Monte Carlo Tree Search
 - UCT algorithm

Naïve Approach

- Use simulations directly as an evaluation function for $\alpha\beta$
- Problems
 - Single simulation is very noisy, only 0/1 signal
 - running many simulations for one evaluation is very slow
- Example:
 - typical speed of chess programs **1 million eval/second**
 - Go: 1 million moves/second, 400 moves/simulation,
100 simulations/eval = **25 eval/second**
- Result: Monte Carlo was ignored for over 10 years in Go

Monte Carlo Tree Search

- ↗ Idea: use results of simulations to guide growth of the game tree
- ↗ **Exploitation:** focus on promising moves
- ↗ **Exploration:** focus on moves where uncertainty about evaluation is high
- ↗ Two contradictory goals?
 - ↗ Theory of *bandits* can help

Bandits



- ↗ Multi-armed bandits
(slot machines in Casino)
- ↗ Assumptions:
 - ↗ Choice of several arms
 - ↗ each arm pull is independent of other pulls
 - ↗ Each arm has *fixed, unknown average payoff*
- ↗ Which arm has the best *average payoff*?
- ↗ Want to minimize *regret* = loss from playing non-optimal arm

Example (1)

- ↗ Three arms A, B, C
- ↗ Each pull of one arm is either
 - ↗ a win (payoff 1) or
 - ↗ a loss (payoff 0)
- ↗ Probability of win for each arm is fixed but *unknown*:
 - ↗ $p(A \text{ wins}) = 60\%$
 - ↗ $p(B \text{ wins}) = 55\%$
 - ↗ $p(C \text{ wins}) = 40\%$
- ↗ A is best arm (but we don't know that)

Example (2)

- How to find out which arm is best? Which arm is best?
- The only thing we can do is play them
- Play each arm many times
 - the empirical payoff will approach the (unknown) true payoff
- Example:
 - Play A, win
 - Play B, loss
 - Play C, win
 - Play A, loss
 - Play B, loss
- How to choose which arm to pull in each round?

Applying the Bandit Model to Games

- Bandit arm \approx move in game
- Payoff \approx quality of move
- Regret \approx difference to best move

Explore and Exploit with Bandits

- ↗ *Explore* all arms, but also:
- ↗ *Exploit*: play promising arms more often
- ↗ Minimize *regret* from playing poor arms

Formal Setting for Bandits

- One specific setting, more general ones exist
- K arms (actions, possible moves) named 1, 2, ..., K
- $t \geq 1$ time steps
- X_i random variable, payoff of arm i
 - Assumed *independent of time* here
- Later: discussion of *drift* over time, i.e. with trees
- Assume $X_i \in [0 \dots 1]$ e.g. 0 = loss, 1 = win
- $\mu_i = E[X_i]$ expected payoff of arm i
- r_t reward at time t
 - realization of random variable X_i from playing arm i at time t

Formalization Example

- Same example as with A, B, C before, but use formal notation
- K=3 .. 3 arms, arm 1 = A, arm 2 = B, arm 3 = C
- X_1 = random variable – pull arm 1
 - $X_1 = 1$ with probability 0.6
 - $X_1 = 0$ with probability 1 - 0.6 = 0.4
 - similar for X_2, X_3
- $\mu_1 = E[X_1] = 0.6, \mu_2 = E[X_2] = 0.55, \mu_3 = E[X_3] = 0.4$
- Each r_t is either 0 or 1, with probability given by the arm which was pulled.
 - Example: $r_1 = 0, r_2 = 0, r_3 = 1, r_4 = 1, r_5 = 0, r_6 = 1, \dots$

Formal Setting for Bandits (2)

- *Policy*: Strategy for choosing arm to play at time t
 - given arm selections and outcomes of previous trials at times $1, \dots, t - 1$.
- $l_t \in \{1, \dots, K\}$.. arm selected at time t
- $T_i(t) = \sum_{s=1}^t \mathbb{I}(l_s = i)$
 - .. total number of times arm i was played from time $1, \dots, t$

Example

- Example: $|_1 = 2, |_2 = 3, |_3 = 2, |_4 = 3, |_5 = 2, |_6 = 2$
- $T_1(6) = 0, T_2(6) = 4, T_3(6) = 2$
- Simple policies:
 - Uniform - play a least-played arm, break ties randomly
 - Greedy - play an arm with highest empirical payoff
 - Question – what is a *smart* strategy?

Formal Setting for Bandits (3)

- ↗ Best possible payoff: $\mu^* = \max_{1 \leq i \leq K} \mu_i$
- ↗ Expected payoff after n steps: $\sum_{i=1}^K \mu_i \mathbb{E}[T_i(n)]$
- ↗ Regret after n steps is the difference:
$$n\mu^* - \sum_{i=1}^K \mu_i \mathbb{E}[T_i(n)]$$
- ↗ Minimize regret: minimize $T_i(n)$ for the non-optimal moves, especially the worst ones

Example, continued

- ↗ $\mu_1 = 0.6, \mu_2 = 0.55, \mu_3 = 0.4$
- ↗ $\mu^* = 0.6$
- ↗ With our fixed exploration policy from before:
 - ↗ $E[T_1(6)] = 0, E[T_2(6)] = 4, E[T_3(6)] = 2$
 - ↗ expected payoff $\mu_1 * 0 + \mu_2 * 4 + \mu_3 * 2 = 3.0$
 - ↗ expected payoff if always plays arm 1: $\mu^* * 6 = 3.6$
 - ↗ Regret = $3.6 - 3.0 = 0.6$
- ↗ Important: regret of a policy is **expected regret**
 - ↗ Will be achieved in the limit, as average of many repetitions of this experiment
 - ↗ In any single experiment with six rounds, the payoff can be anything from 0 to 6, with varying probabilities

Formal Setting for Bandits (4)

- ↗ (Auer et al 2002)
- ↗ Statistics on each arm so far
- ↗ \bar{X}_i average reward from arm i so far
- ↗ n_i number of times arm i played so far
(same meaning as $T_i(t)$ above)
- ↗ n total number of trials so far

UCB1 Formula (Auer et al 2002)

- Name UCB stands for Upper Confidence Bound
- Policy:
 1. First, try each arm once
 2. Then, at each time step:
 - choose arm i that maximizes the *UCB1 formula* for the upper confidence bound:

$$\bar{x}_i + \sqrt{\frac{2 \ln(n)}{n_i}}$$

UCB Demystified - Formula

$$\bar{x}_i + \sqrt{\frac{2 \ln(n)}{n_i}}$$

- ↗ Exploitation: higher observed reward \bar{x}_i is better
- ↗ Expect “true value” μ_i to be in some *confidence interval* around \bar{x}_i .
- ↗ “Optimism in face of uncertainty”: choose move for which the upper bound of confidence interval is highest

UCB Demystified – Exploration Term

$$\bar{x}_i + \sqrt{\frac{2 \ln(n)}{n_i}}$$

- ↗ Interval is large when number of trials n_i is small.
Interval shrinks in proportion to $\sqrt{n_i}$
- ↗ High uncertainty about move
 - ↗ large exploration term in UCB formula
 - ↗ move is explored
- ↗ $\sqrt{\ln(n)}$ term, intuition:
explore children more if parent is important
(has many simulations)

Theoretical Properties of UCB1

- Main question: rate of convergence to optimal arm
- Huge amount of literature on different bandit algorithms and their properties
- Typical goal: regret $O(\log n)$ for n trials
- For many kinds of problems, cannot do better asymptotically (Lai and Robbins 1985)
- UCB1 is a simple algorithm that achieves this asymptotic bound for many input distributions

Is UCB What we Really Want??

- ↗ No.
- ↗ UCB minimizes *cumulative regret*
- ↗ Regret is accumulated over all trials
- ↗ In games, we only care about the final move choice
 - ↗ We do not care about simulating bad moves
- ↗ *Simple regret:* loss of our final move choice, compared to best move
 - ↗ Better measure, but theory is much less developed for trees

The case of Trees: From UCB to UCT

- ↗ UCB makes a single decision
- ↗ What about sequences of decisions (e.g. planning, games)?
- ↗ Answer: use a lookahead tree (as in games)
- ↗ Scenarios
 - ↗ Single-agent (planning, all actions controlled)
 - ↗ **Adversarial** (as in games, or worst-case analysis)
 - ↗ Probabilistic (average case, “neutral” environment)



Our
Focus

Monte Carlo Planning - UCT

- Main ideas:
- Build lookahead tree (e.g. game tree)
- Use rollouts (simulations) to generate rewards
- Apply UCB – like formula in interior nodes of tree
 - choose “optimistically” where to expand next

Generic Monte Carlo Planning Algorithm

MonteCarloPlanning(state)

```
repeat search(state, 0) until Timeout
    if Terminal(state) then return 0
    if Leaf(state, depth) then return Evaluate(state)
    action := selectAction(state, depth)
    (nextstate, reward) := simulate (state, action)
    q := reward +  $\gamma$  search(nextstate, depth + 1)
    UpdateValue(state, action, q, depth)
return q
```

- Reinforcement-learning-like framework
(Kocsis and Szepesvari 2006)
- Rewards at every time step
 - future rewards discounted by factor γ
- Apply to games:
 - 0/1 reward, only at end of game
 - $\gamma = 1$ (no discount)

Generic Monte Carlo Tree Search

- ↗ Select leaf node L in game tree
- ↗ Expand children of L
- ↗ Simulate a randomized game from (new) leaf node
- ↗ Update (or backpropagate) statistics on path to root

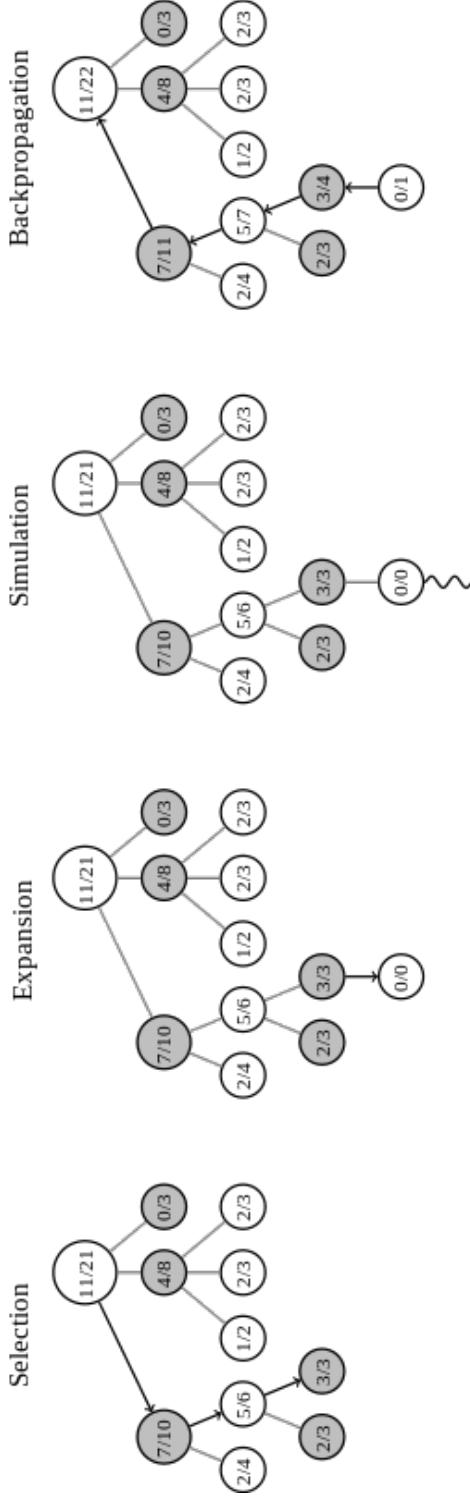


Image source: http://en.wikipedia.org/wiki/Monte-Carlo_tree_search

Drift

- In basic bandit framework, we assumed that payoff for each arm comes from a *fixed* (stationary) distribution
- If distribution changes over time, UCB will still converge under some relatively weak conditions
- In UCT, the tree changes over time
 - payoffs of choices within tree also change
 - Example: better move is discovered for one of the players

Convergence Property of UCT

- Very informal presentation here.
- See (K+S 2006), Section 2.4 for precise statements.
- Assumptions:
 1. average payoffs converge for each arm /
 2. “tail inequalities”: probability of being “far off” is very small
- Under those conditions:
 - probability of selecting a suboptimal move approaches zero in the limit

Towards Practice: UCB1-tuned

- ↗ Finite-time Analysis of the Multiarmed Bandit Problem (Auer et al 2002)
- ↗ UCB1 formula simply assumes variance decreases with $1/\sqrt{n_i}$ of number of trials n_i
- ↗ UCB1-tuned idea: take *measured variance* of each arm (move choice) into account
- ↗ Compute upper confidence bound using that measured variance
 - ↗ Can be better in practice
- ↗ We will see many more extensions to UCB ideas

MoGo – First UCT Go Program

- Original MoGo technical report (Gelly et al 2006)
- Modify UCB1-tuned, add two parameters:
 - *First-play urgency* - value for unplayed move
 - *exploration constant c* (called ρ in first paper) - controls rate of exploration
 $\rho = 1.2$ found best empirically for early MoGo

$$\bar{X}_j + \rho \sqrt{\frac{\log n}{T_j(n)} \min\{1/4, V_j(n_j)\}}$$

Formula from original MoGo report

Move Selection for UCT

- Scenario:
 - run UCT as long as we can
 - run simulations, grow tree
- When out of time, which move to play?
 - Highest mean
 - Highest UCB
 - **Most-simulated move**
 - later refinement: most wins

Summary – MCTS So Far

- UCB, UCT are very important algorithms in both theory and practice
- Well founded, convergence guarantees under relatively weak conditions
- Basis for extremely successful programs for games and many other applications

MCTS Enhancements

- Improved simulations
 - Mostly game-specific
 - We will discuss it later
- Improved in-tree child selection
 - General approaches
 - Review – the history heuristic
 - AMAF and RAVE
- Prior knowledge for initializing nodes in tree

Improved In-Tree Child Selection

- Plain UCT: in-tree child selection by UCB formula
- Components: exploitation term (mean) and exploration term
- Enhancements: modify formula, add other terms
 - Collect other kinds of statistics – AMAF, RAVE
 - Prior knowledge – game specific evaluation terms
- Two main approaches
 - Add another term
 - “Equivalent experience” – translate knowledge into (virtual, fake) simulation wins or losses

Review - History Heuristic

- Game-independent enhancement for alphabeta
- Goal: improve move ordering
(Schaeffer 1983, 1989)
- Give bonus for moves that lead to cutoff
Prefer those moves at other places in the search
- Similar ideas in MCTS:
 - all-moves-as-first (AMAF) heuristic, RAVE

Assumptions of History Heuristic

- Abstract concept of *move*
 - Not just a single edge in the game graph
 - identify *class of all moves* e.g. “Black F3” - place stone of given color on given square
- History heuristic: quality of such moves is correlated
 - tries to exploit that correlation
- Special case of reasoning by similarity:
in similar state, the same action may also be good
 - Classical: if move often lead to a beta cut in search, try it again, might lead to similar cutoff in similar position.
 - MCTS: if move helped to win previous simulations, then give it a bonus for its evaluation - will lead to more exploration of the move

All Moves As First (AMAF) Heuristic

- ↗ (Brügmann 1993)
- ↗ Plain Monte Carlo search:
 - ↗ no game tree, only simulations, winrate statistics for each first move
- ↗ AMAF idea: bonus for *all* moves in a winning simulation, not just the first.
 - ↗ Treat all moves like the first
 - ↗ Statistics in *global table*, *separate* from winrate
- ↗ Main advantage: statistics accumulate much faster
- ↗ Disadvantage: some moves good only if played right now - they will get a very bad AMAF score.

RAVE - Rapid Action Value Estimate

- Idea (Gelly and Silver 2007): compute separate AMAF statistics in *each node* of the MCTS tree
- After each simulation, update the RAVE scores of all ancestors that are in the tree
- Each move i in the tree now also has a RAVE score:
 - number of simulations $n_{i,RAVE}$
 - number of wins $V_{i,RAVE}$
 - RAVE value $X_{i,RAVE} = V_{i,RAVE}/n_{i,RAVE}$

RAVE Illustration

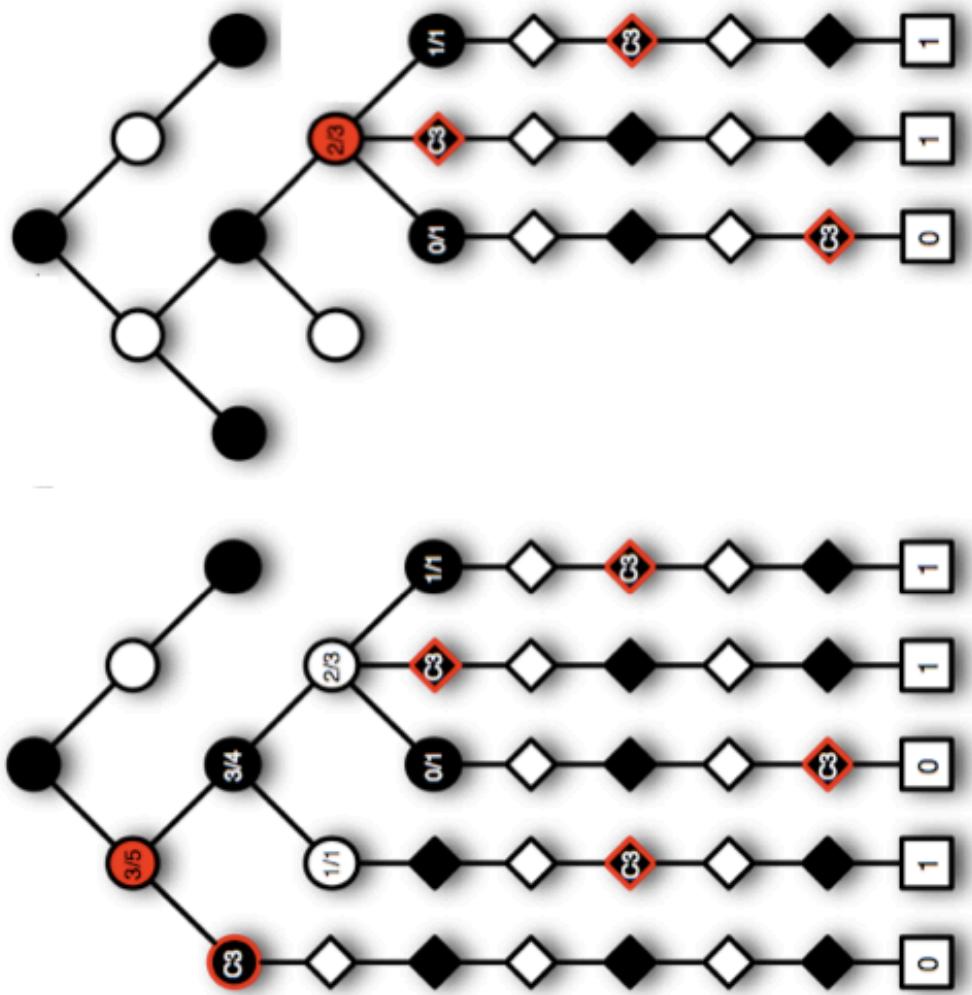


Image source: (Silver 2009)

Adding RAVE to the UCB Formula

- Basic idea: replace mean value x_i with weighted combination of mean value and RAVE value
$$\beta x_i + (1 - \beta) x_{i,RAVE}$$
- How to choose β ?
Not constant, depends on all statistics
- Try to find best combined estimator given x_i and $x_{i,RAVE}$

Adding RAVE (2)

- Original method in MoGo (Gelly and Silver 2007):
- equivalence parameter k = number of simulations when mean and RAVE have equal weight
 - When $n_i = k$, then $\beta = 0.5$
- Results were quite stable for wide range of $k=50...10000$

- Formula

$$\beta(s, a) = \frac{k}{\sqrt{3n(s) + k}}$$

Adding RAVE (3)

- ↗ (Silver 2009, Chapter 8.4.3)
 - ↗ Assume independence of estimates
 - ↗ not true in real life, but useful assumption
 - ↗ Can compute optimal choice in closed form (!)
 - ↗ Estimated by machine learning, or trial and error

Adding RAVE (4) – Fuego Program

- General scheme to combine different estimators
- Combining mean and RAVE is special case
 - Very similar to Silver's scheme
- General scheme: each estimator has:
 1. *initial slope*
 2. *final asymptotic value*
- Details: <http://fuego.sourceforge.net/fuegodoc-1.1/smartergame-doc/sguctsearchweights.html>

Using Prior Knowledge

- (Gelly and Silver 2007)
- Most nodes in the game tree are leaf nodes (exponential growth)
- Almost no statistics for leaf nodes - only simulated once
- Use domain-specific knowledge to initialize nodes
 - “equivalent experience” - a number of wins and losses
 - additive term (Rosin 2011)
- Similar to heuristic initialization in proof-number search

Types of Prior Knowledge

- ↗ (Silver 2009) machine-learned 3x3 pattern values
- ↗ Later Mogo and Fuego: hand-crafted features
- ↗ Crazy Stone: many features, weights trained by Minorization-Maximization (MM) algorithm (Coulom 2007)
- ↗ Fuego today:
 - ↗ large number of simple features
 - ↗ weights and interaction weights trained by *Latent Feature Ranking* (Wistuba et al 2013)

Example – Pattern Features (Coulom)

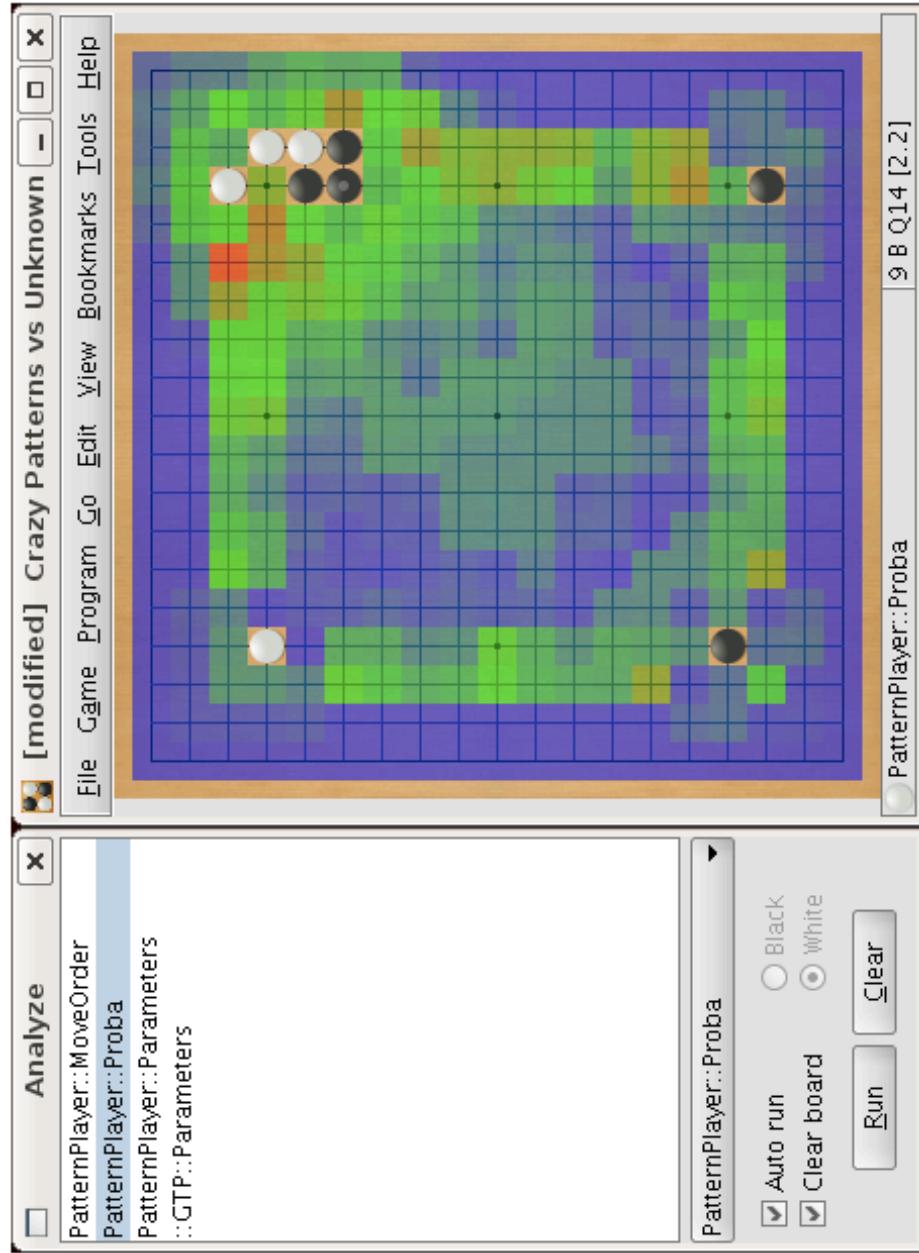


Image source: Remi Coulom

Improving Simulations

- ↗ Goal: strong correlation between initial position and result of simulation
- ↗ Preserve wins and losses
- ↗ How?
 - ↗ Avoid blunders
 - ↗ “Stabilize” position
 - ↗ Go: prefer local replies
 - ↗ Go: urgent pattern replies

Improving Simulations (2)

- Game-independent techniques
 - If there is an immediate win, then take it (1 ply win check)
 - Avoid immediate losses in simulation (1 ply mate check)
 - Avoid moves that give opponent an immediate win (2 play mate check)
- Last Good Reply – next slide

Last Good Reply

- ↗ Last Good Reply (Drake 2009),
Last Good Reply with Forgetting (Baier et al 2010)
- ↗ Idea: after winning simulation, store (opponent move,
our answer) move pairs
 - ↗ Try same reply in future simulations
 - ↗ Forgetting: delete move pair if it fails
- ↗ Evaluation: worked well for Go program with simpler
playout policy (Orego)
- ↗ Trouble reproducing success with stronger Go programs
- ↗ Simple form of adaptive simulations

Hybrid Approaches

- Combine MCTS with “older” ideas from the alphabeta world
- Examples
 - Prove wins/losses
 - Use evaluation function
 - Hybrid search strategy MCTS+alphabeta

Hybrids: MCTS + Game Solver

- Recognize leaf nodes that are wins/losses
- Backup in minimax/proof tree fashion
- Problem: how to adapt child selection if some children are proven wins or losses?
 - At least, don't expand those anymore
- Useful in many games, e.g. Hex, Lines of Action, NoGo, Havannah, Konane,...

Hybrids: MCTS + Evaluation

- Use evaluation function
 - Standard MCTS plays until end of game
 - Some games have reasonable and fast evaluation functions, but can still profit from exploration
 - Examples: Amazons, Lines of Action
- Hybrid approach (Lorentz 2008, Winands et al 2010)
 - run short simulation for fixed number of moves (e.g. 5-6 in Amazons)
 - call static evaluation at end, use as simulation result

Hybrids: MCTS + Minimax

- 1-2 ply lookahead in playouts (discussed before)
- Require strong evaluation function
- (Baier and Winands 2013) add minimax with no evaluation function to MCTS
- Playouts
 - Avoid forced losses
- Selection/Expansion
 - Find shallow wins/losses

Towards a Tournament-Level Program

- ↗ Early search termination – best move cannot change
- ↗ Pondering – think in opponent's time
- ↗ Time control – how much time to spend for each move
- ↗ Reuse sub-tree from previous search
- ↗ Multithreading (see later)
- ↗ Code optimization
- ↗ Testing, testing, testing, ...

Machine Learning for MCTS

- ↗ Learn better knowledge
 - ↗ Patterns, features (discussed before)
- ↗ Learn better simulation policies
 - ↗ Simulation balancing (Silver and Tesauro 2009)
 - ↗ Simulation balancing in practice (Huang et al 2011)
- ↗ Adapt simulations online
 - ↗ Dyna2, RLGo (Silver et al 2012)
 - ↗ Nested Rollout Policy Adaptation (Rosin 2011)
 - ↗ Last Good Reply (discussed before)
 - ↗ Use RAVE (Rimmel et al 2011)

Parallel MCTS

- MCTS scales well with more computation
- Currently, hardware is moving quickly towards more parallelism
- MCTS simulations are “embarrassingly parallel”
- Growing the tree is a sequential algorithm
 - How to parallelize it?

Parallel MCTS - Approaches

- root parallelism
 - shared memory
 - distributed memory
-
- New algorithm: depth-first UCT (Yoshizoe et al 2011)
 - Avoid bottleneck of updates to the root

Root Parallelism

- ↗ (Cazenave and Jouandeau 2007, Soejima et al. 2010)
- ↗ Run n independent MCTS searches on n nodes
- ↗ Add up the top-level statistics
- ↗ Easiest to implement, but limited
- ↗ Majority vote may be better

Shared Memory Parallelism

- ↗ n cores together build one tree in shared memory
- ↗ How to synchronize access? Need to write results (changes to statistics for mean and RAVE), add nodes, and read statistics for in-tree move selection
- ↗ Simplest approach: lock tree during each change
- ↗ Better: lock-free hash table (Coulom 2008) or tree (Enzenberger and Müller 2010)
- ↗ Possible to use spinlock

Limits to Parallelism

- Loss of information from running n simulations in parallel as opposed to sequentially
- Experiment (Segal 2010)
 - run single-threaded
 - delay tree updates by $n - 1$ simulations
- Best-case experiment for behavior of parallel MCTS
- Predicts upper limit of strength over 4000 Elo above single-threaded performance

Virtual Loss

- ↗ Record simulation as a loss at start
 - ↗ Leads to more variety in UCT-like child selection
- ↗ Change to a win if outcome is a win
- ↗ Crucial technique for scaling
- ↗ With virtual loss, scales well up to 64 threads
- ↗ Can also use *virtual wins*

Fuego Virtual Loss Experiment

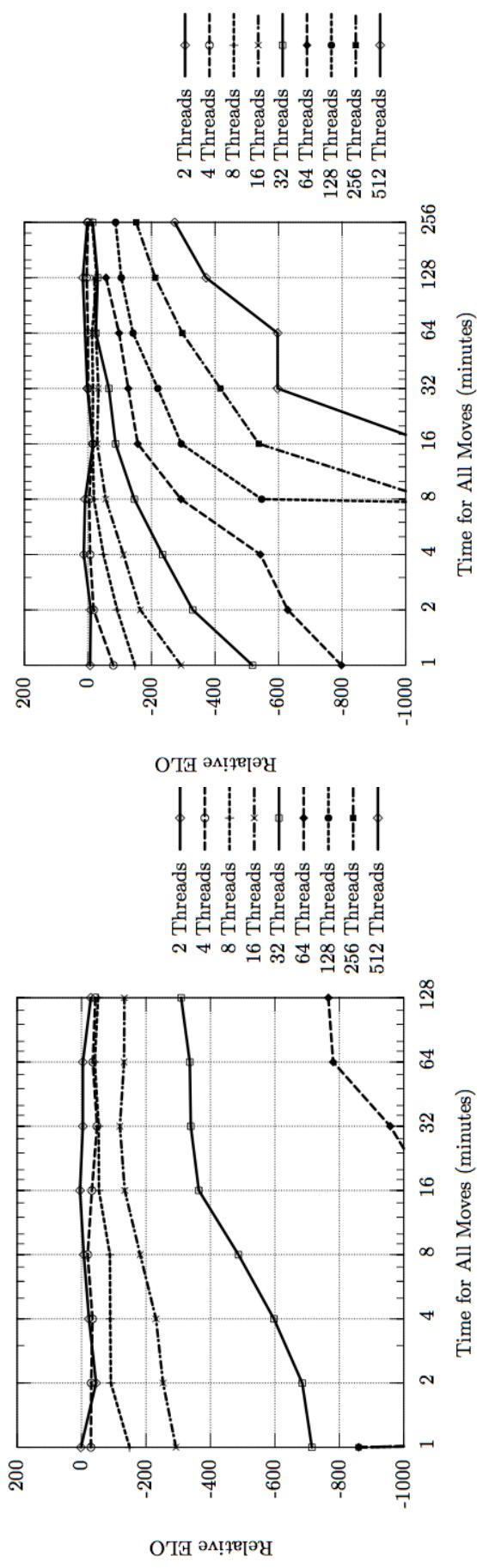


Fig. 2. Self-play of N threads against a uni-processor with equal total computation.

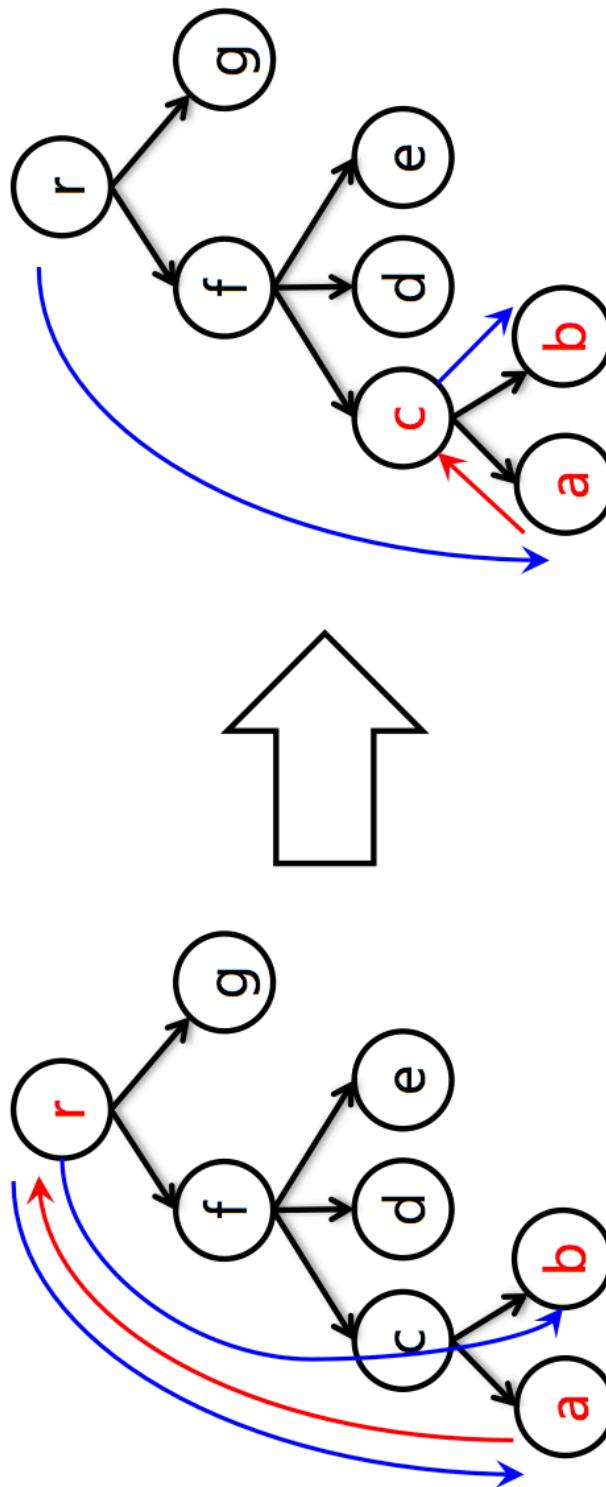
Fig. 4. Self-play of N threads against a uni-processor and virtual loss enabled.

Image source: (Segal 2010)

Distributed Memory Parallelism

- ↗ Many copies of MCTS engine, one on each compute node
- ↗ Communicate by message passing (MPI)
- ↗ MoGo model:
 - ↗ synchronize a few times per second
 - ↗ synchronize only “heavy” nodes which have many simulations
- ↗ Performance depends on
 - ↗ hardware for communication
 - ↗ shape of tree
 - ↗ game-specific properties, length of playouts

Normal UCT vs. Depth-first UCT



Depth First UCT

returns only if needed

Normal UCT

always return to root

Depth-first UCT

- Bottleneck of updates to “heavy” nodes including root
- Depth-first reformulation of UCT
 - stay in subtree while best-child selection is unlikely to change
 - about 1 - 2% wrong child selections
 - Delay updates further up the tree
 - Similar idea as df-pn
- Unlike df-pn, sometimes the 3rd-best (or worse) child can become best

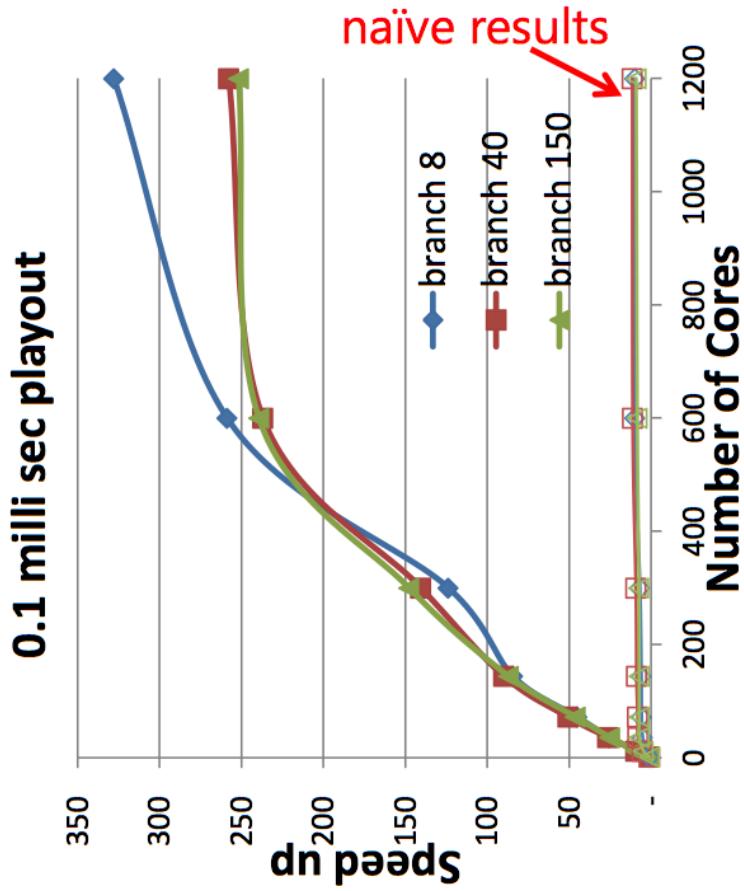
Distributed Memory: TDS

- TDS – Transposition Table Driven Scheduling
(Romein et al 1999)
- Single global hash table
 - Each node in tree owned by one processor
 - Work is sent to the processor that owns the node
 - In single-agent search, achieved almost perfect speedup on mid-size parallel machines

TDS-df-UCT

- Use TDS approach to implement df-UCT on (massively) parallel machines
 - TSUBAME2 (17984 cores)
 - SGI UV-1000 (2048 cores)
- Implemented artificial game (P-game) and Go (MP-Fuego program)
 - In P-game: measure effect of playout speed (artificial slowdown for fake simulations)

TDS-df-UCT Speedup - 1200 Cores



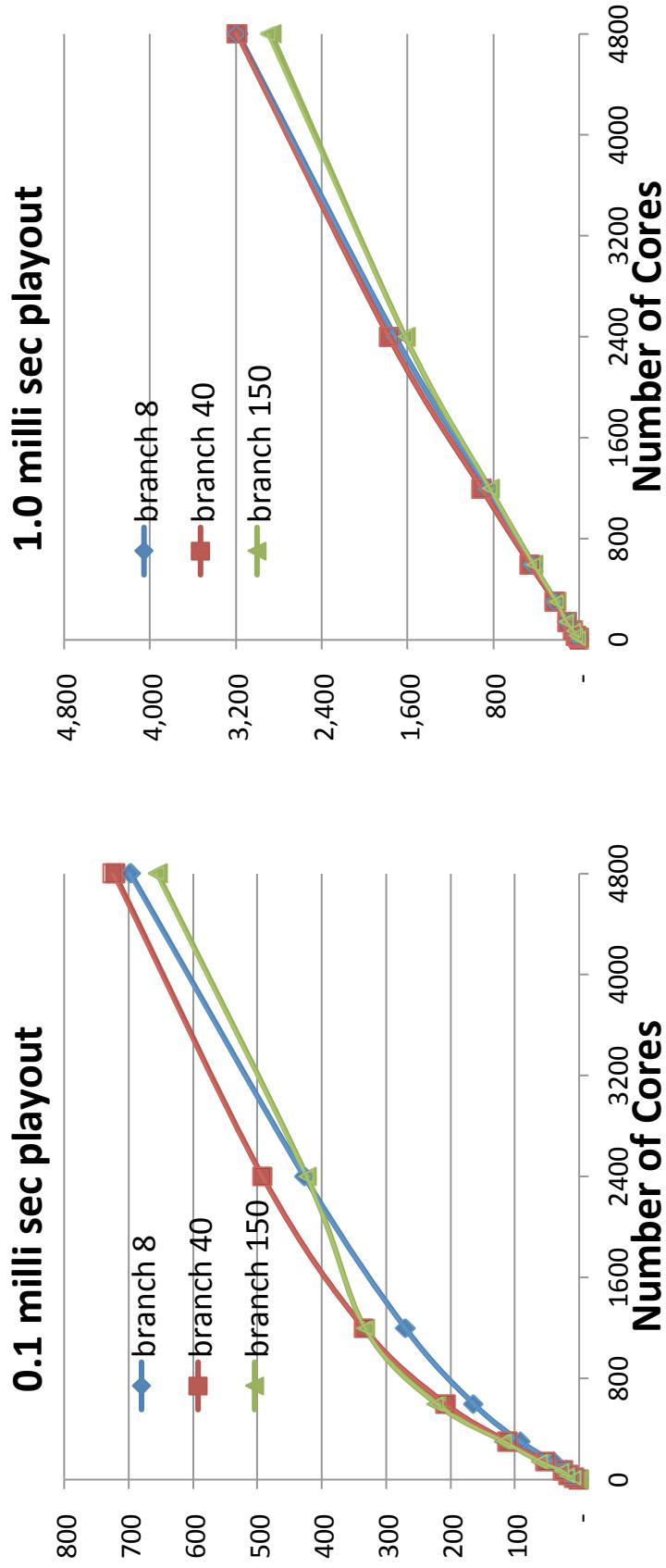
1.0 milli sec playout



naïve results

330 fold speedup for 0.1 ms playout
740 fold speedup for 1.0 ms playout

P-game 4,800 Cores



job number
= cores x 10
700-fold for 0.1 ms payout
3,200-fold for 1.0 ms payout

TDS-df-UCT = TDS + depth first UCT

Speedup including Go



Hardware1: TSUBAME2 supercomputer

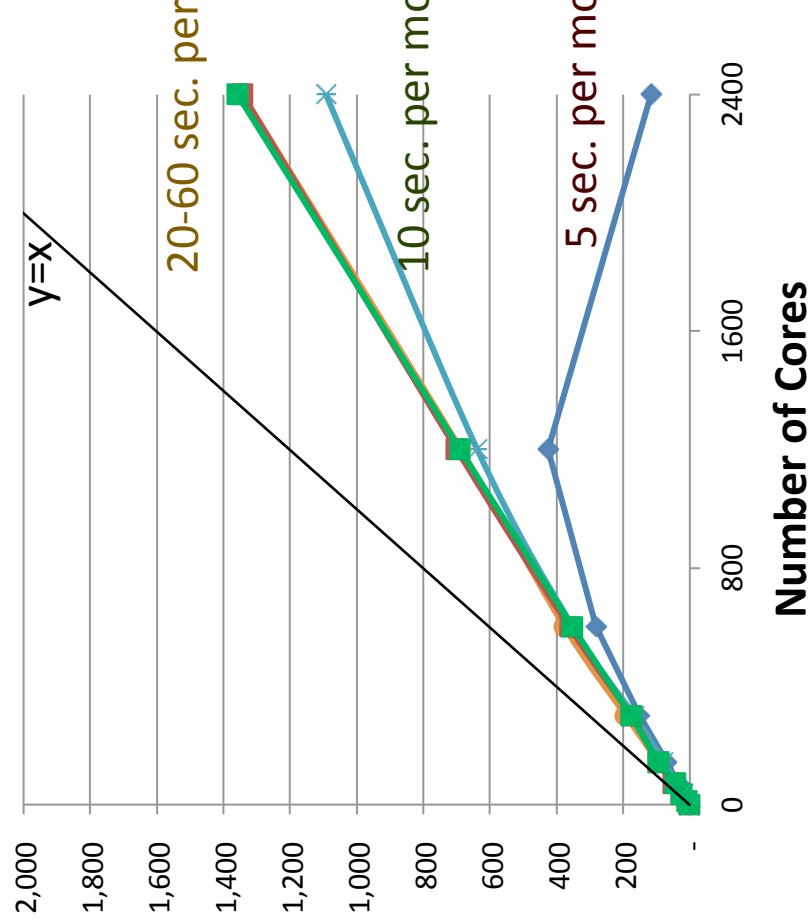


Hardware2: SGI UV1000 (Hungabee)

Image source: K. Yoshizoe

Search Time and Speedup

MP-Fuego speedup (19x19)



→ Short thinking time =
slower speedup

→ One major difficulty in
massive parallel search

Summary – MCTS Tutorial so far...

- Reviewed algorithms, enhancements, applications
 - Bandits
 - Simulations
 - Monte Carlo Tree Search
 - AMAF, RAVE, adding knowledge
 - Hybrid algorithms
 - Parallel algorithms
- Still to come: impact of MCTS, research topics

Impact - Applications of MCTS

- Classical Board Games
 - Go, Hex
 - Amazons
 - Lines of Action, Arimaa, Havannah, NoGo, Konane, ...
- Multi-player games, card games, RTS, video games
- Probabilistic Planning, MDP, POMDP
- Optimization, energy management, scheduling, distributed constraint satisfaction, library performance tuning, ...

Impact – Strengths of MCTS

- Very general algorithm for decision making
- Works with very little domain-specific knowledge
 - Need a simulator of the domain
- Can take advantage of knowledge when present
- Successful parallelizations for both shared memory and massively parallel distributed systems

Current Topics in MCTS

- ↗ Recent progress, Limitations, random half-baked ideas, challenges for future work,...
- ↗ Dynamically adaptive simulations
- ↗ Integrating local search and analysis
- ↗ Improve in-tree child selection
- ↗ Parallel search
 - ↗ Extra simulations should never hurt
 - ↗ Sequential halving and SHOT

Dynamically Adaptive Simulations

- Idea: adapt simulations to specific current context
 - Very appealing idea, only modest results so far
 - Biassing using RAVE (Rimmel et al 2010) – small improvement
- Last Good Reply (with Forgetting) (Drake 2009, Baier et al 2010)

Integrating Local Search and Analysis

- Mainly For Go
 - Players do much local analysis
 - Much of the work on simulation policies and knowledge is about local replies
- Combinatorial Game Theory has many theoretical concepts
- Tactical alphabeta search (Fuego, unpublished)
- Life and death solvers

Improve In-tree Child Selection

- ↗ Intuition: want to maximize if we're certain, average if uncertain
- ↗ Is there a better formula than average weighted by number of simulations? (My intuition: there has to be...)
- ↗ Part of the benefits of iterative widening may be that the max is over fewer sibling nodes – measure that
 - ↗ Restrict averaging to top n nodes

Extra Simulations Should Never Hurt

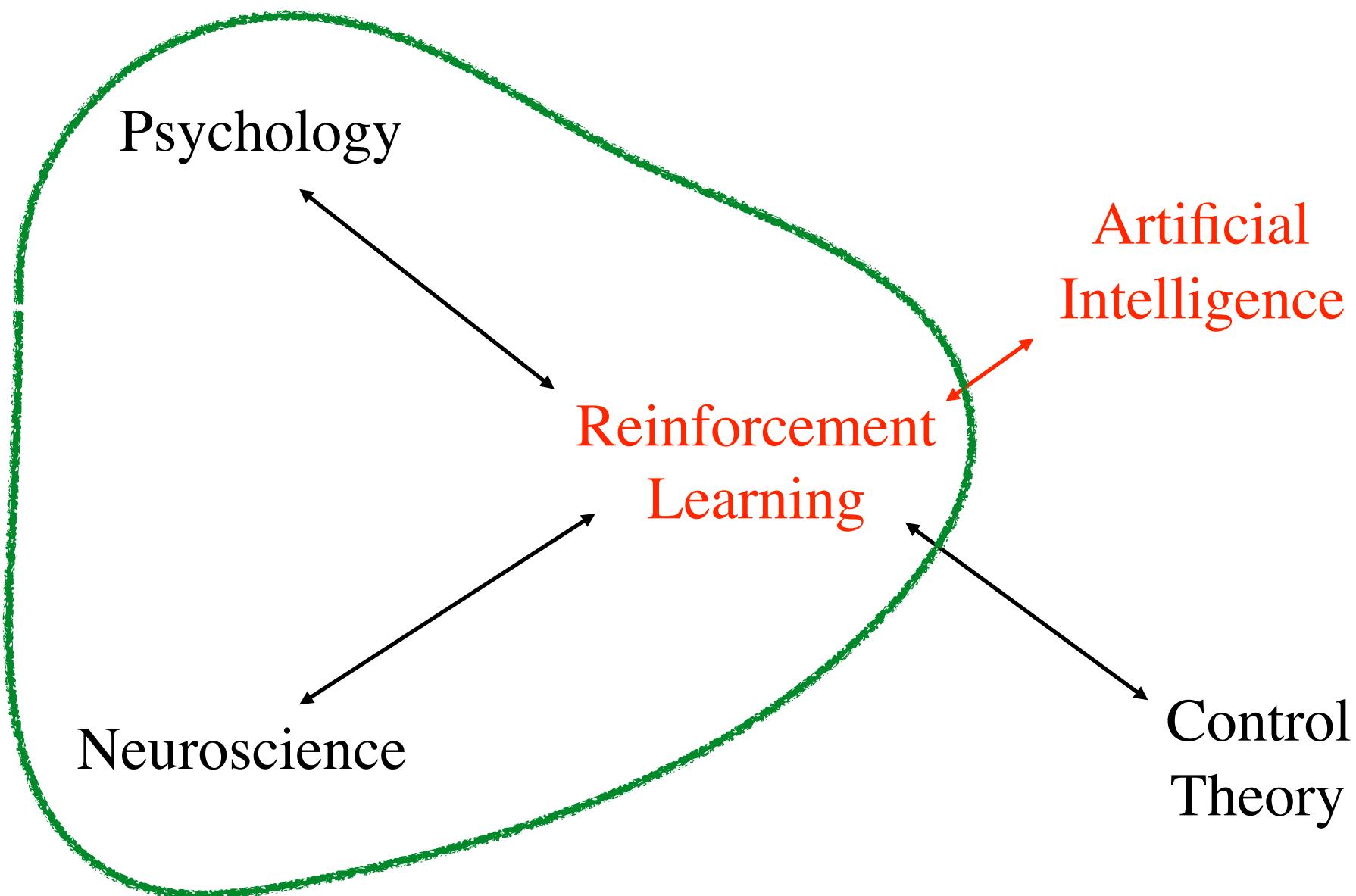
- Ideally, adding more search should never make an algorithm weaker
- For example, if you search nodes that could be pruned in alphabeta, it just becomes slower, but produces the same result
- Unfortunately it is not true for MCTS
- Because of averaging, adding more simulations to bad moves hurts performance - it is worse than doing nothing!

Extra Simulations Should Never Hurt (2)

- Challenge: design a MCTS algorithm that is robust against extra search at the “wrong” nodes
- This would be great for parallel search
- A rough idea: keep two counters in each node - total simulations, and “useful” simulations
- Use only the “useful” simulations for child selections
- Could also “disable” old, obsolete simulations?

Sequential Halving, SHOT

- Early MC algorithm: successive elimination of empirically worst move (Bouzy 2005)
- Sequential halving (Karnin et al 2013):
 - Rounds of uniform sampling
 - keep top half of all moves for next round
- SHOT (Cazenave 2014)
 - Sequential halving applied to trees
 - Like UCT, uses bandit algorithm to control tree growth
 - Promising results for NoGo
 - Promising for parallel search



Any information processing system can be understood at multiple “levels”

- **The Computational Theory Level**
 - *What* is being computed?
 - *Why* are these the right things to compute?
- **Representation and Algorithm Level**
 - *How* are these things computed?
- **Implementation Level**
 - How is this implemented physically?



David Marr, 1972

Reinforcement Learning in Psychology and Neuroscience



with thanks to
Elliot Ludvig
University of Warwick

Psychology has identified two primitive kinds of learning

- *Classical Conditioning*
- *Operant Conditioning* (a.k.a. Instrumental learning)
- Computational theory:
 - ❖ *Classical* = Prediction
 - What is going to happen?
 - ❖ *Operant* = Control



Classical Conditioning

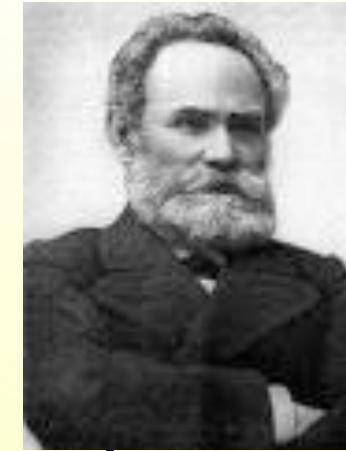


Classical Conditioning as Prediction Learning

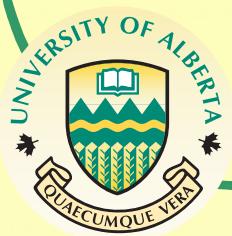
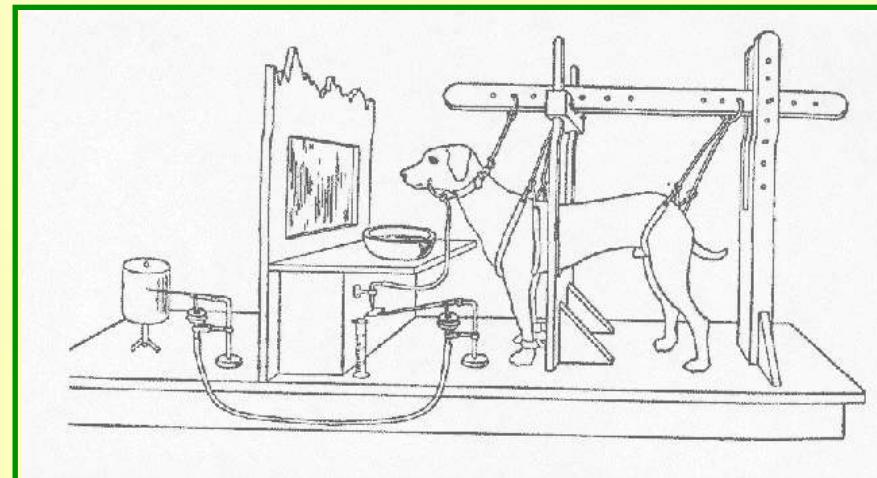
- Classical Conditioning is the process of learning to predict the world around you
 - ❖ Classical Conditioning concerns (typically) the subset of these predictions to which there is a hard-wired response



Pavlov



- Russian physiologist
- Interested in how learning happened in the brain
- Conditional and Unconditional Stimuli



Is it really predictions?



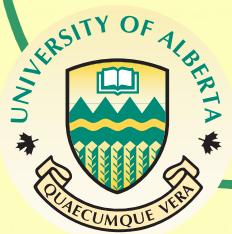
Maybe Contiguity?

- Foundational principle of classical associationism (back to Aristotle)
 - ❖ Contiguity = Co-occurrence
 - ❖ Sufficient for association?

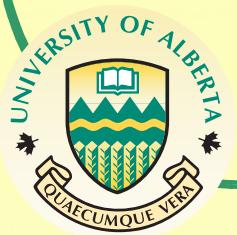
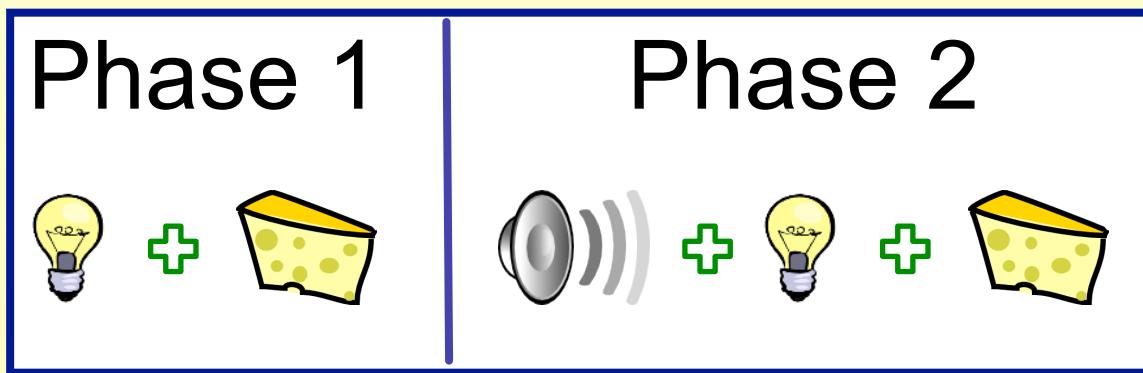


Contiguity Problems

- Unnecessary:
 - ❖ Conditioned Taste Aversion
- Insufficient:
 - ❖ Blocking
 - ❖ Contingency Experiments



Blocking

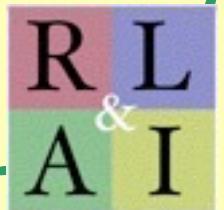




Rescorla-Wagner Model (1972)



- Computational model of conditioning
 - ❖ Widely cited and used
- Learning as violation of expectations
 - ❖ TD learning as extension of RW

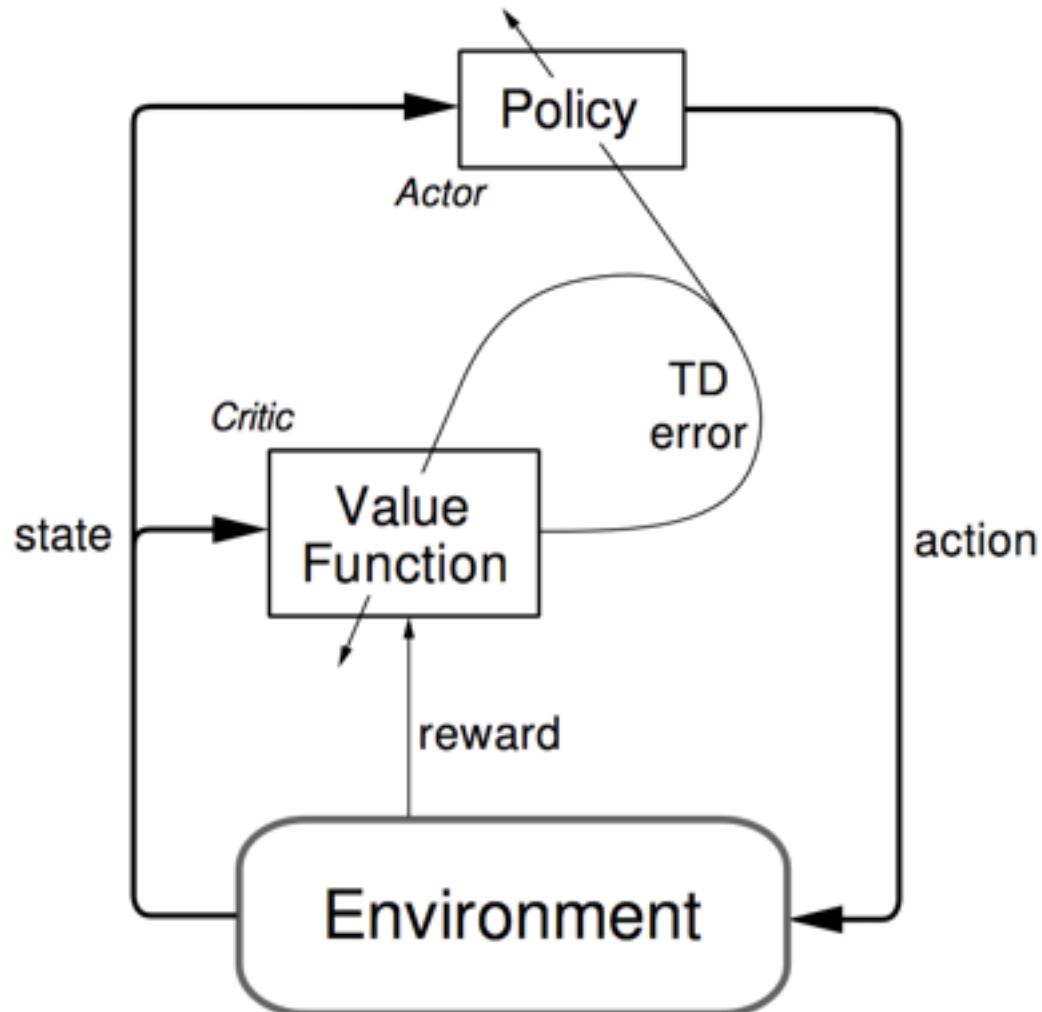


Operant Learning

- Operant Conditioning is all about choice in 3 main ways:
 - ❖ Decide **which** response to make?
 - ❖ Decide **how much** to respond?
 - ❖ Decide **when** to respond?

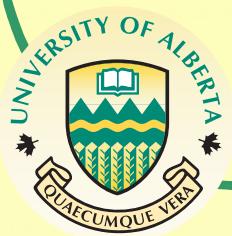
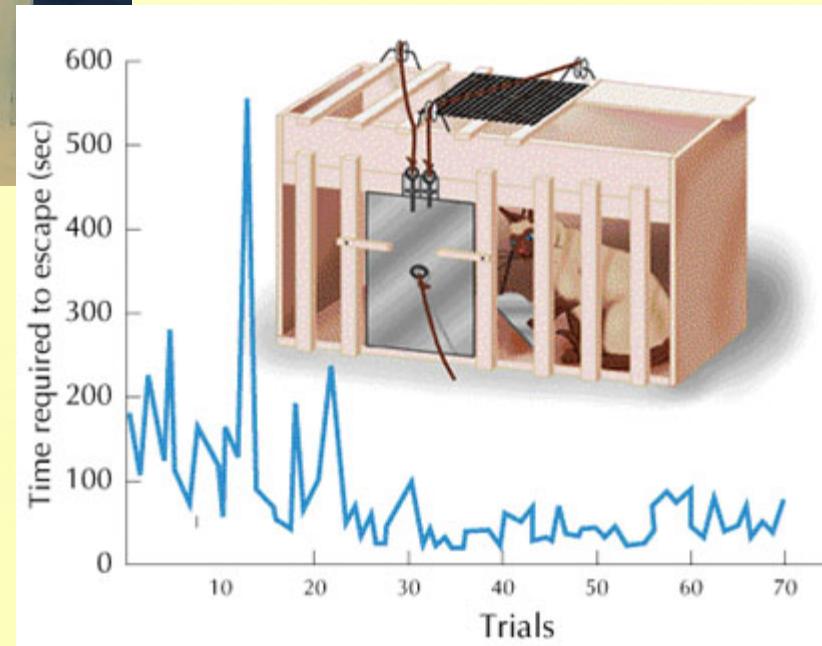
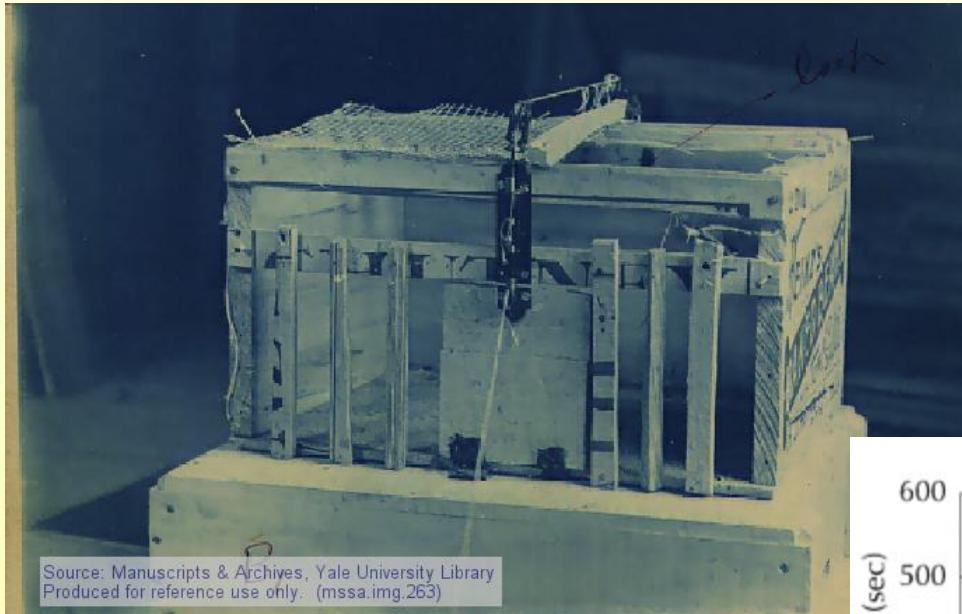


Actor-Critic Methods



- Explicit representation of policy as well as value function
- Minimal computation to select actions
- Can learn an explicit stochastic policy
- Can put constraints on policies
- Appealing as psychological and neural models

Thorndike's Puzzle Box



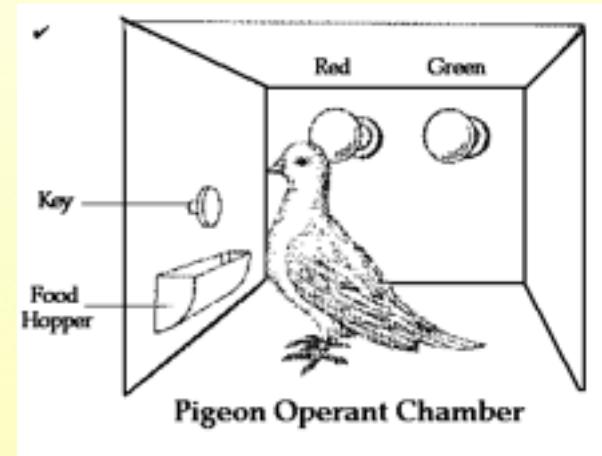
Law of Effect



- “Of several **responses** made to the same situation, those which are accompanied by or closely followed by **satisfaction** to the animal will, other things being equal, be more firmly **connected with the situation**, so that, when it recurs, they will be more likely to recur...” - Thorndike (1911), p. 244



Operant Chambers



Complex Cognition



Any information processing system can be understood at multiple “levels”

- The Computational Theory Level
 - *What* is being computed?
 - *Why* are these the right things to compute?
- Representation and Algorithm Level
 - *How* are these things computed?
- Implementation Level
 - How is this implemented *physically*?



David Marr, 1972

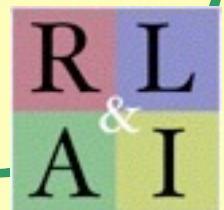
The Basic TD Model

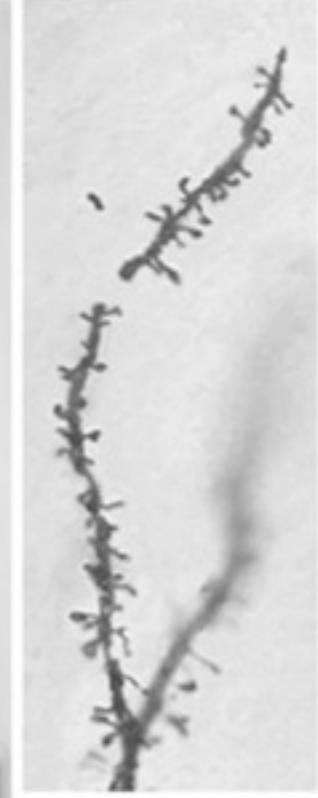
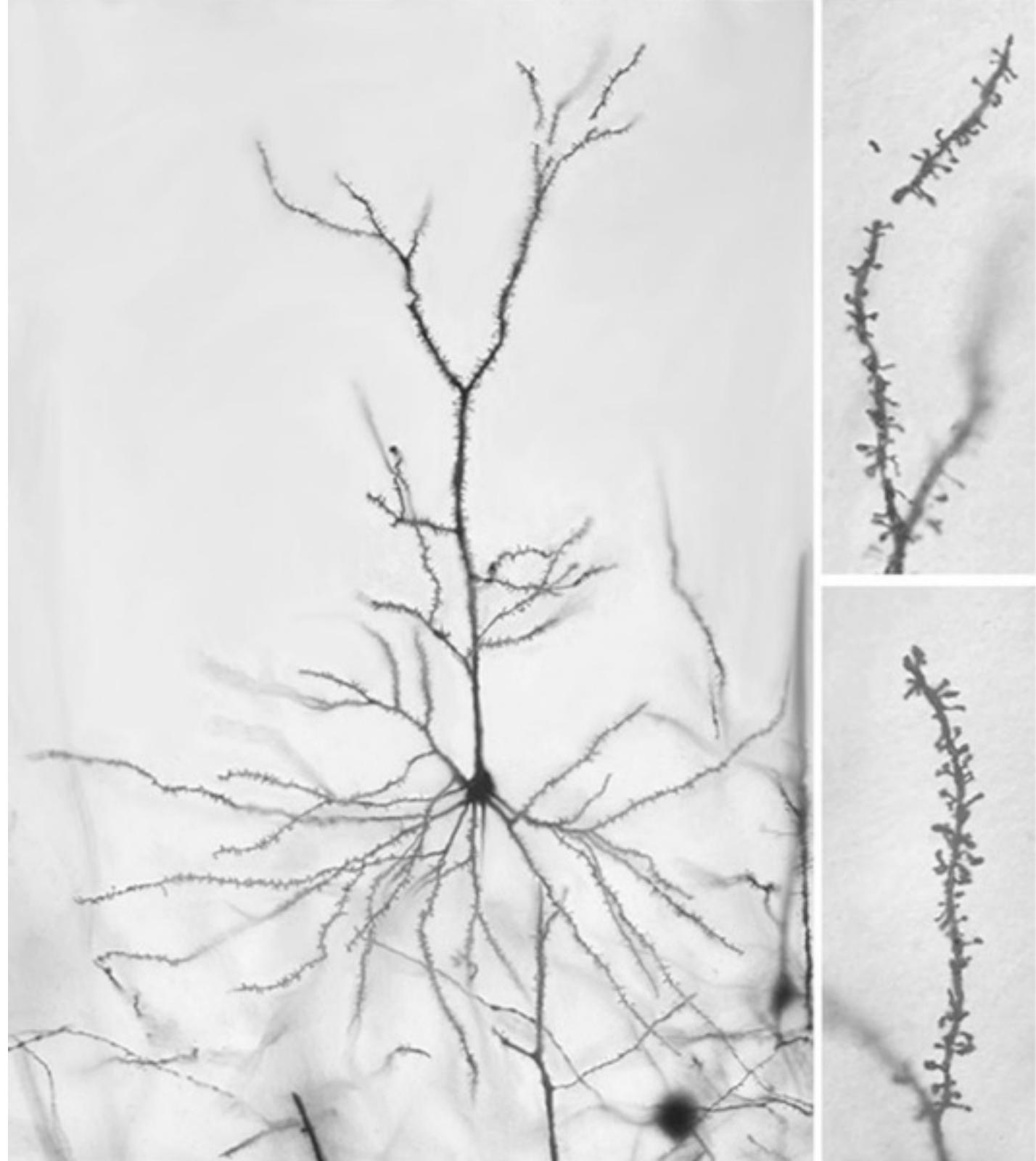
- Learn to predict discounted sum of upcoming reward through TD with linear function approximation:

$$V_t = \mathbf{w}_t^T \mathbf{x}_t = \sum_{i=1}^n w_t(i)x_t(i)$$

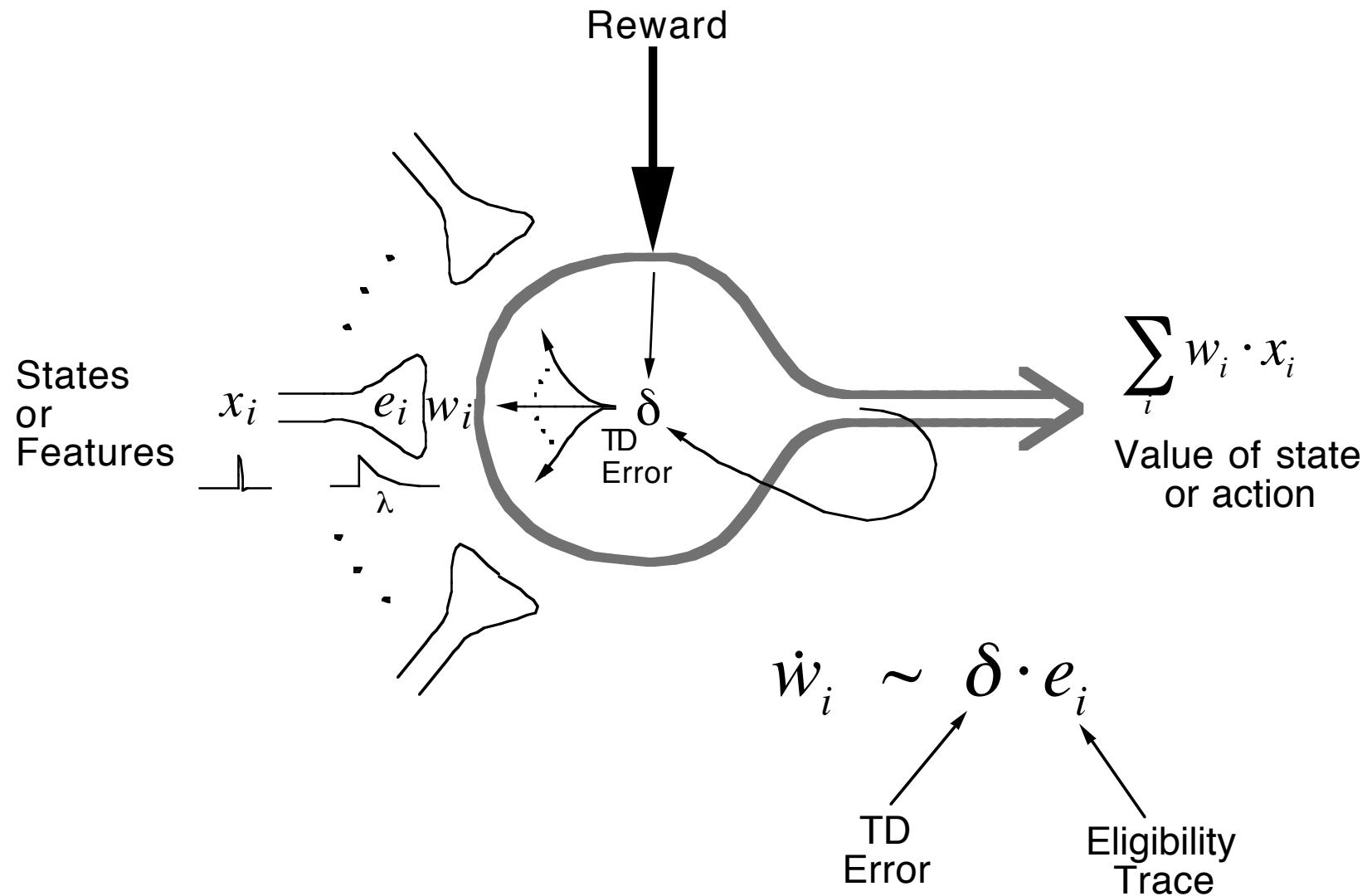
- The TD error is calculated as:

$$\delta_t = r_{t+1} + \gamma V_{t+1} - V_t$$

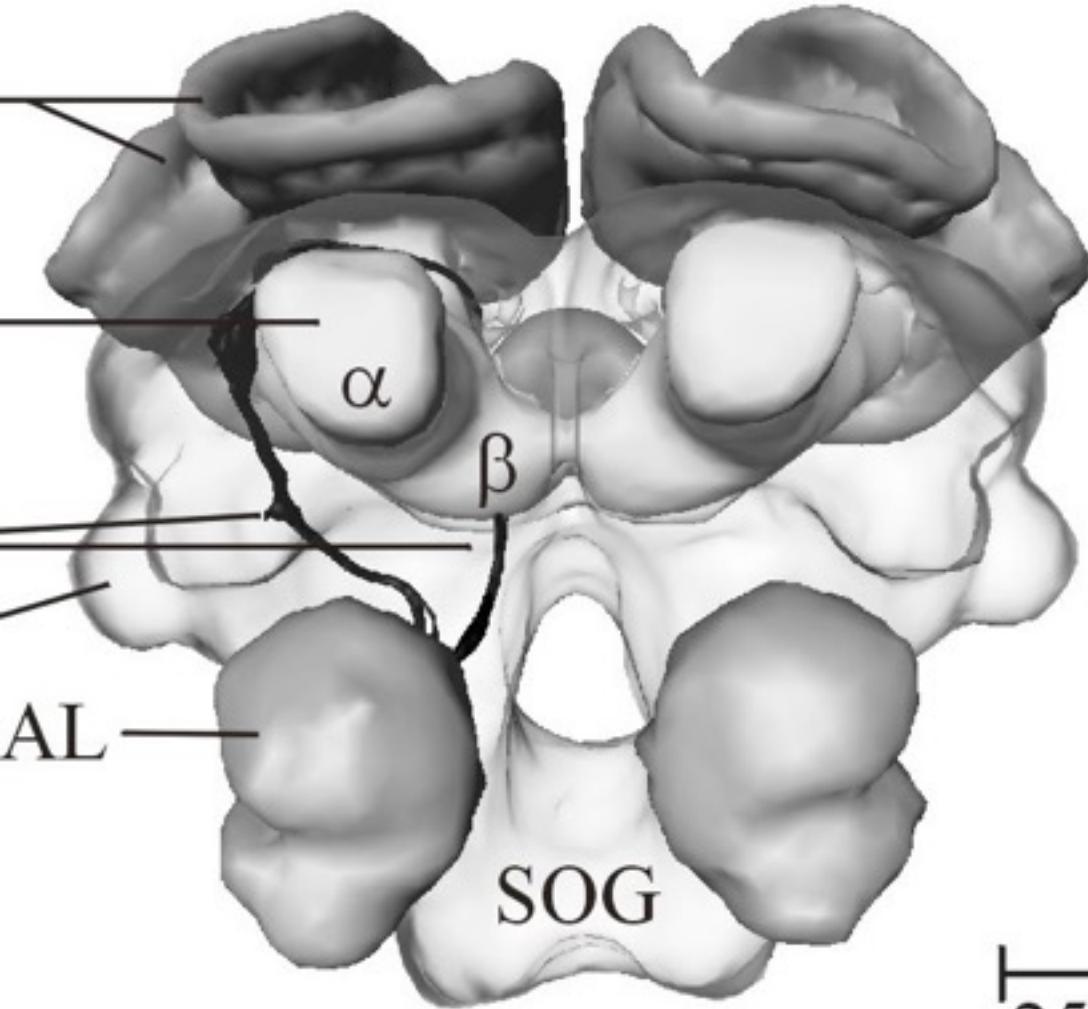




TD(λ) algorithm/model/neuron

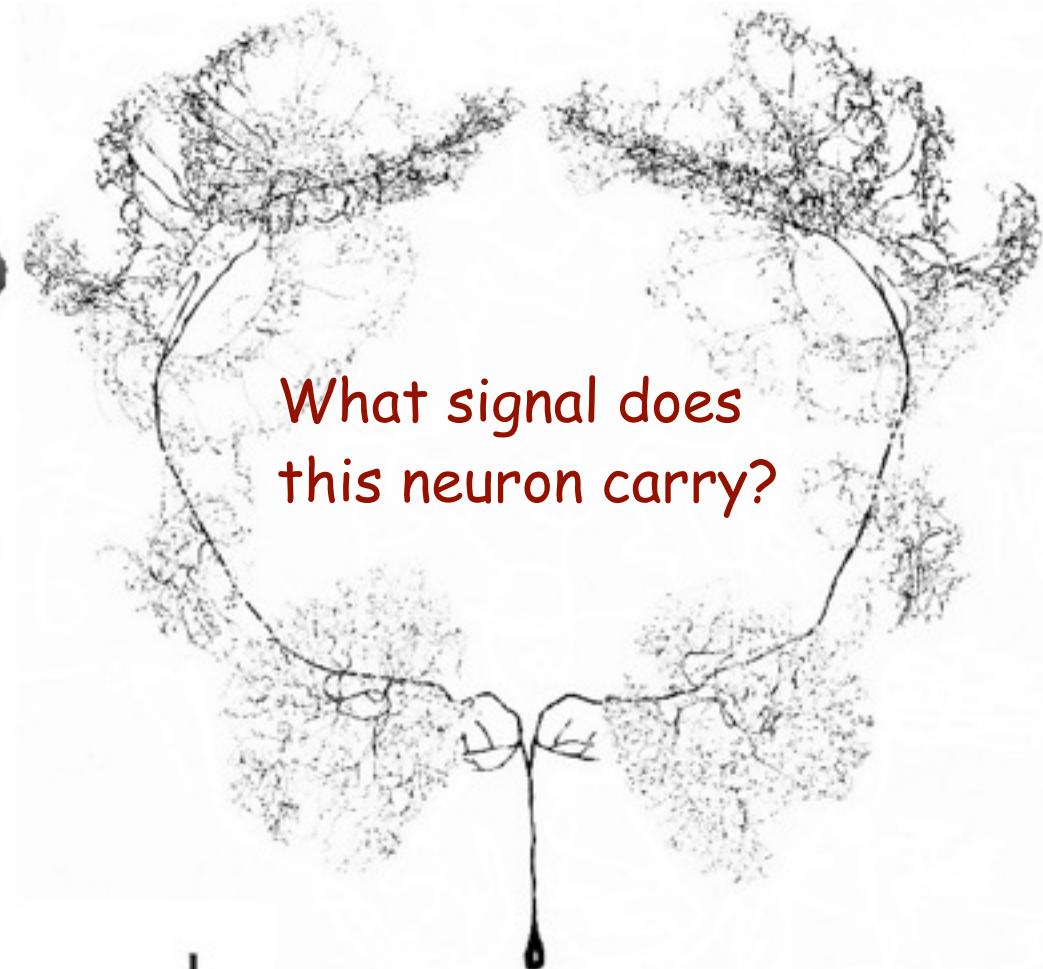


Brain reward systems



Honeybee Brain

250 μm



VUM Neuron

Dopamine

- Small-molecule Neurotransmitter
 - ❖ Diffuse projections from mid-brain throughout the brain

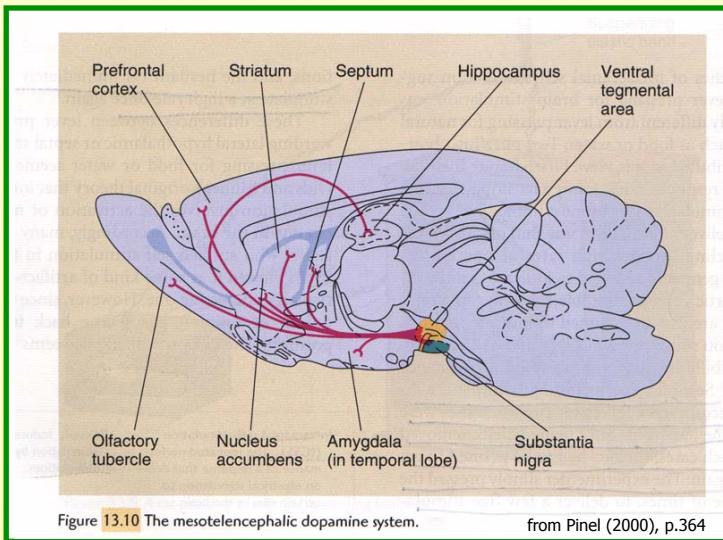


Figure 13.10 The mesotelencephalic dopamine system.

from Pinel (2000), p.364

Key Idea: Phasic change in baseline dopamine responding = reward prediction error

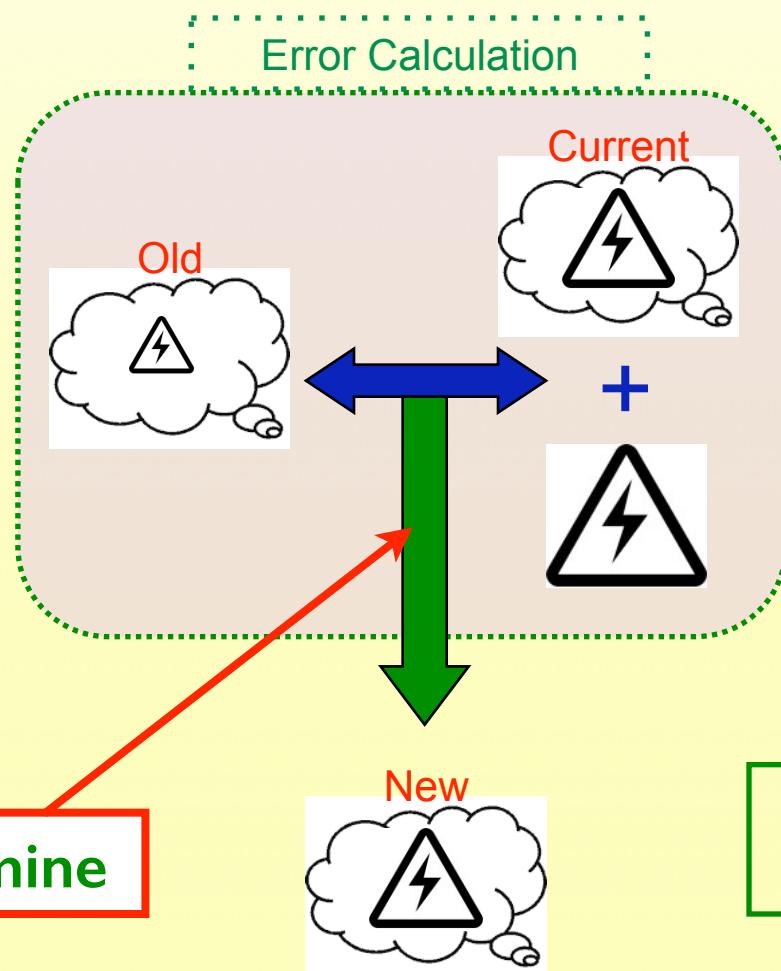


What does Dopamine Do?

- Hedonic Impact
- Motivation
- Motor Activity
- Attention
- Novelty
- Learning



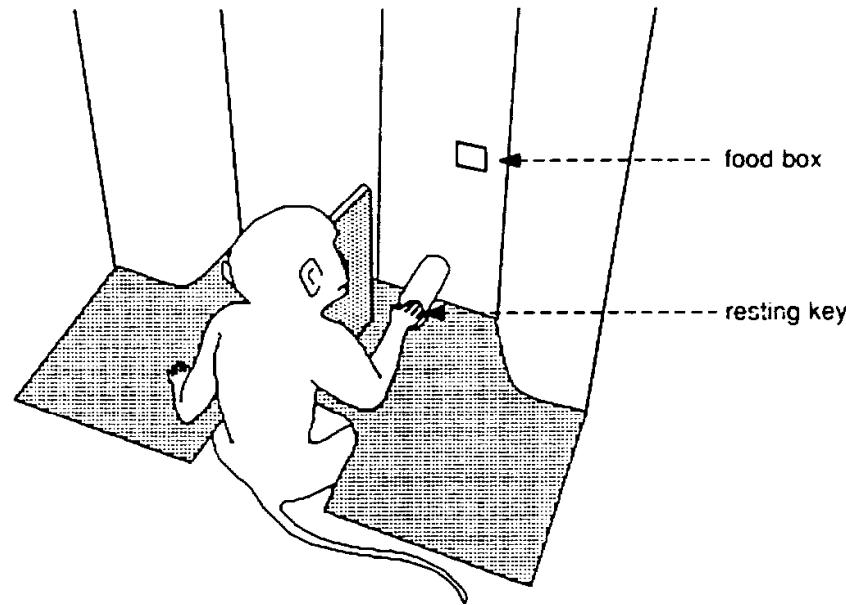
TD Error = Dopamine



Schultz et al., (1997);
Montague et al. (1996)

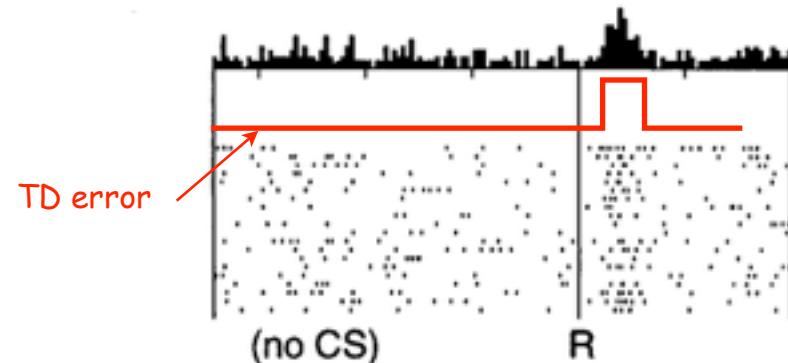


Dopamine neurons signal the error/change in prediction of reward

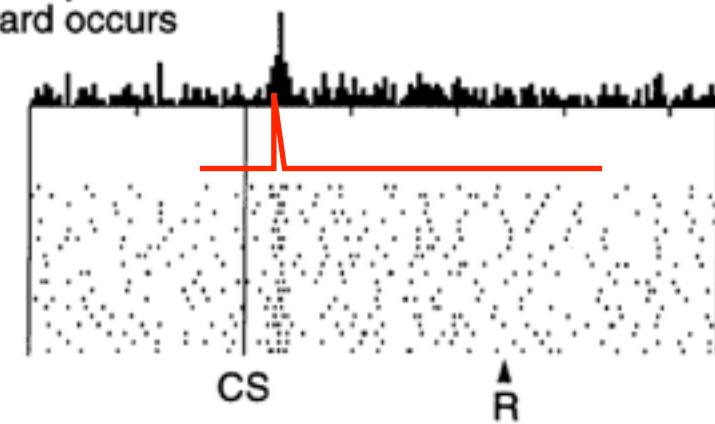


Wolfram Schultz, et al.

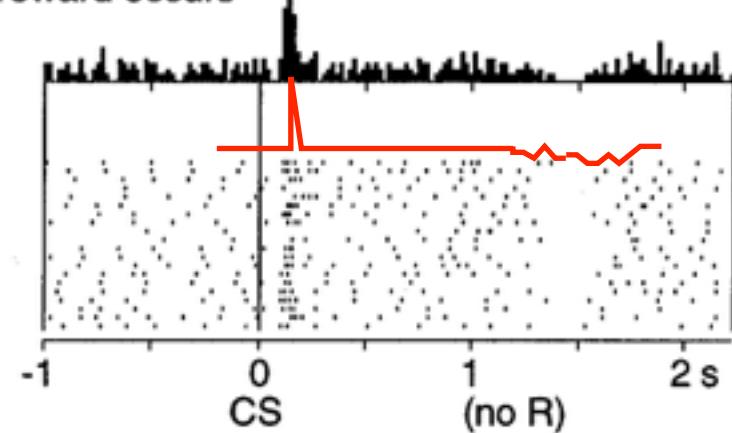
No prediction
Reward occurs



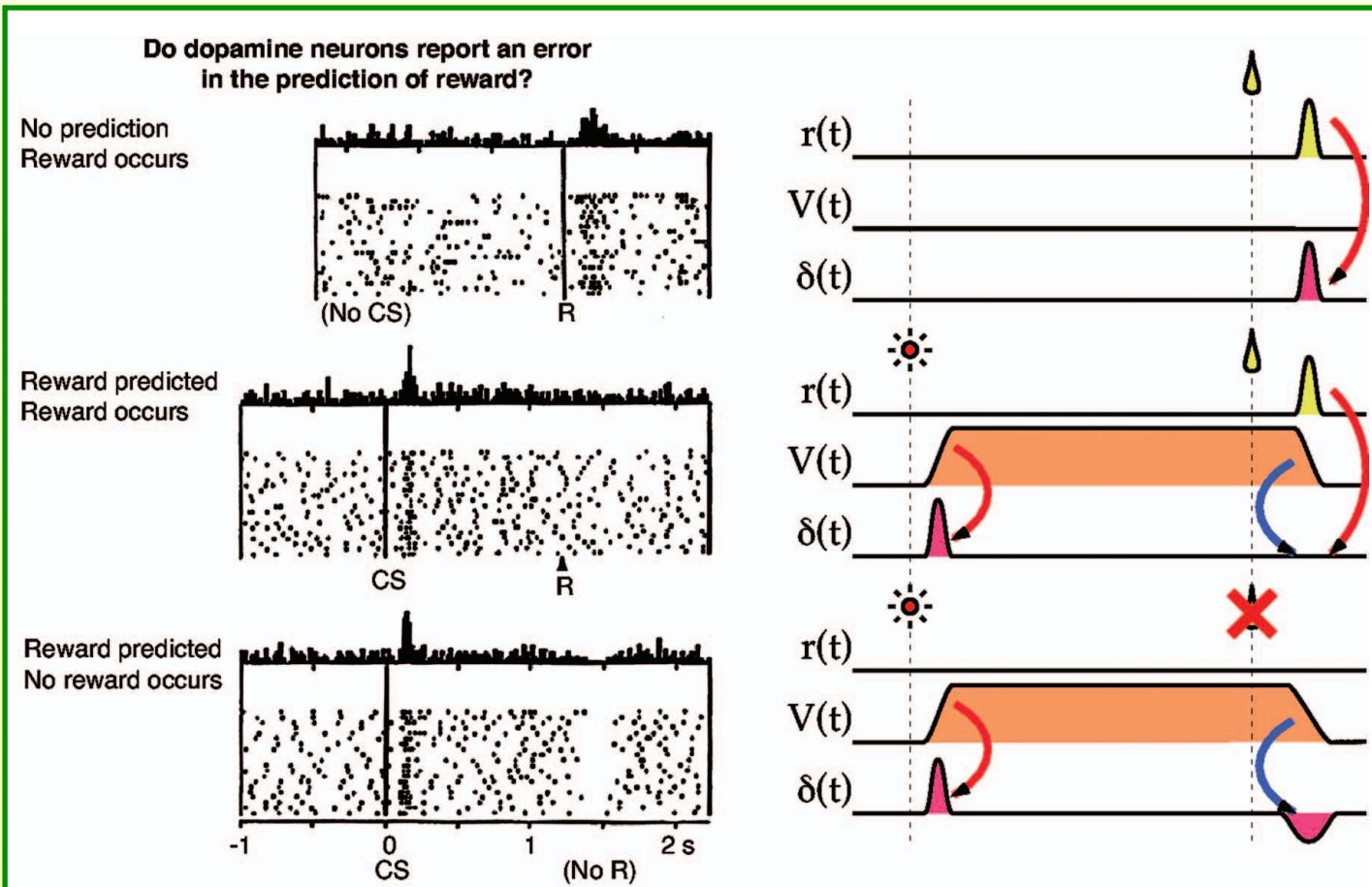
Reward predicted
Reward occurs



Reward predicted
No reward occurs

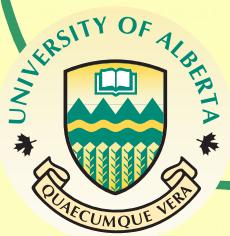


Dopamine Neurons



Schultz et al., 1997

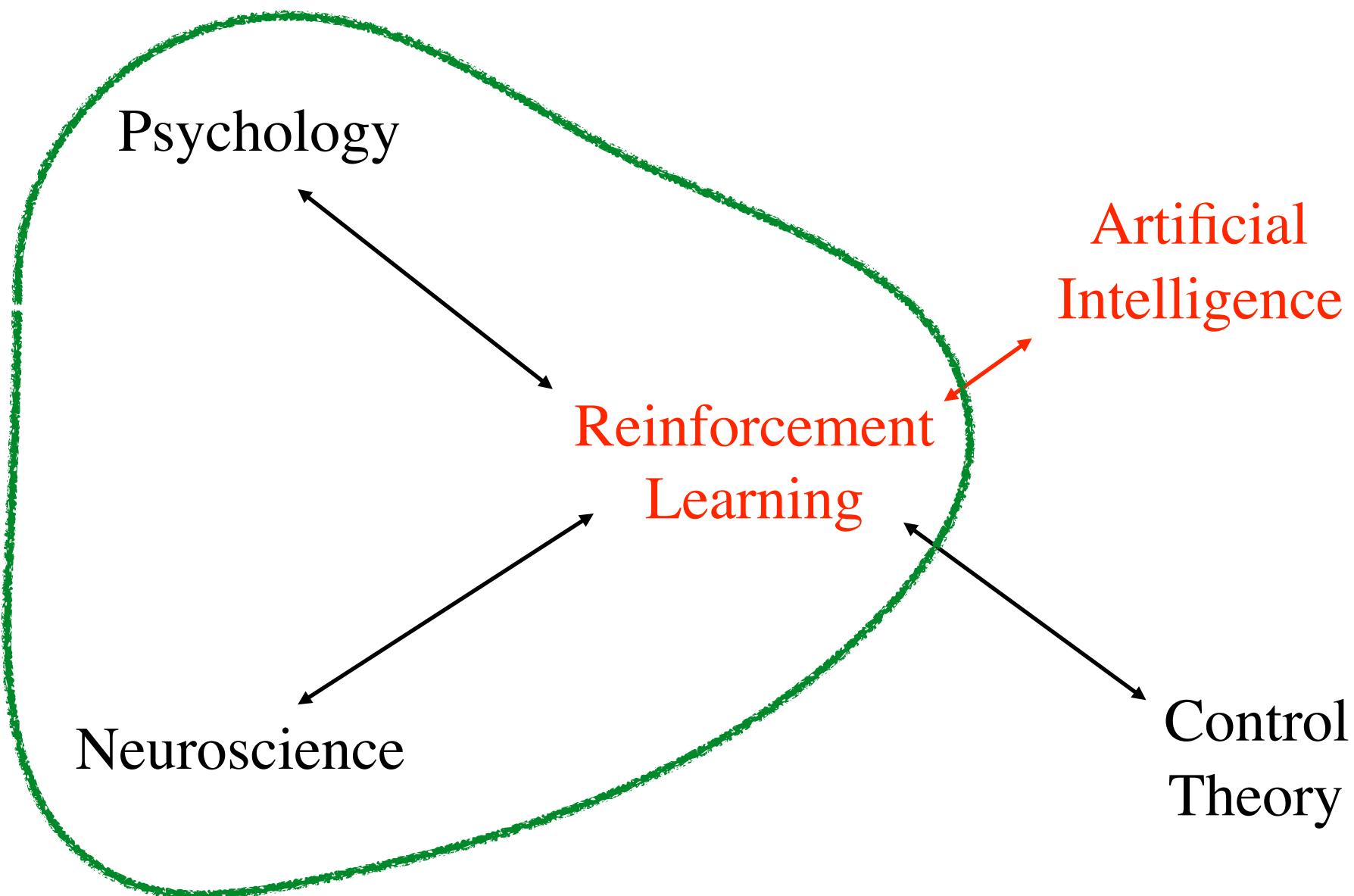
Doya, 2007



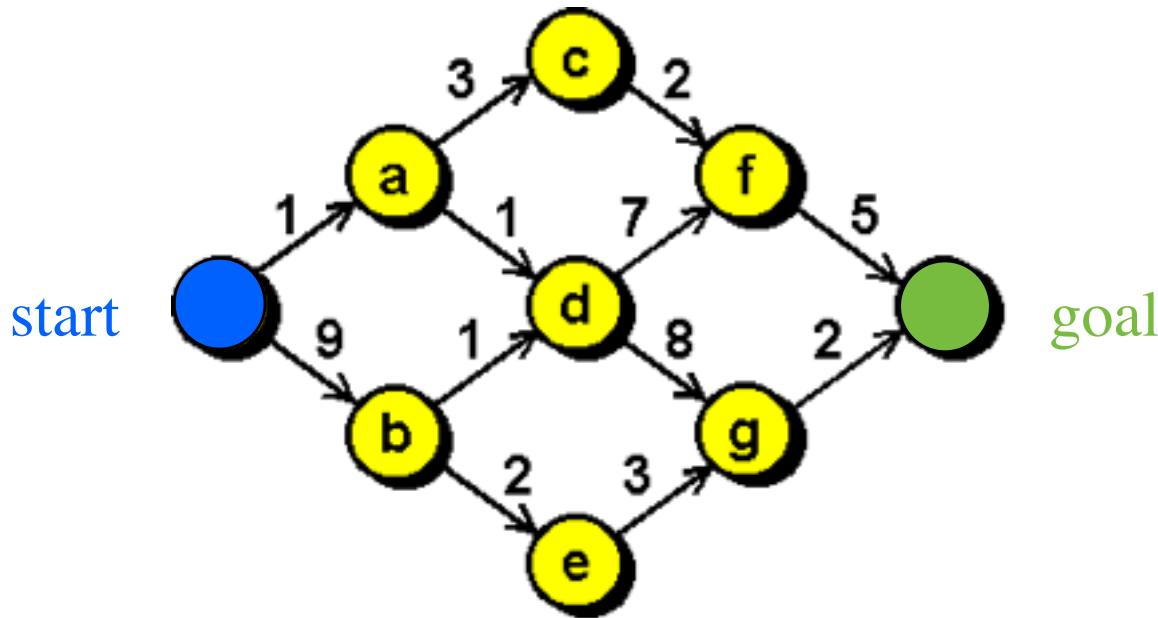
The theory that *Dopamine = TD error*
is one of the *most important interactions ever*
between artificial intelligence and neuroscience

What have you learned about in this course (without buzzwords)?

- “Decision-making over time to achieve a long-term goal”
 - includes learning and planning
 - makes plain why value functions are so important
 - makes plain why so many fields care about these algorithms
 - AI
 - Control theory
 - Psychology and Neuroscience
 - Operations Research
 - Economics
 - all involve decision, goals, and time...
 - the essence of...



Shortest Path Problem



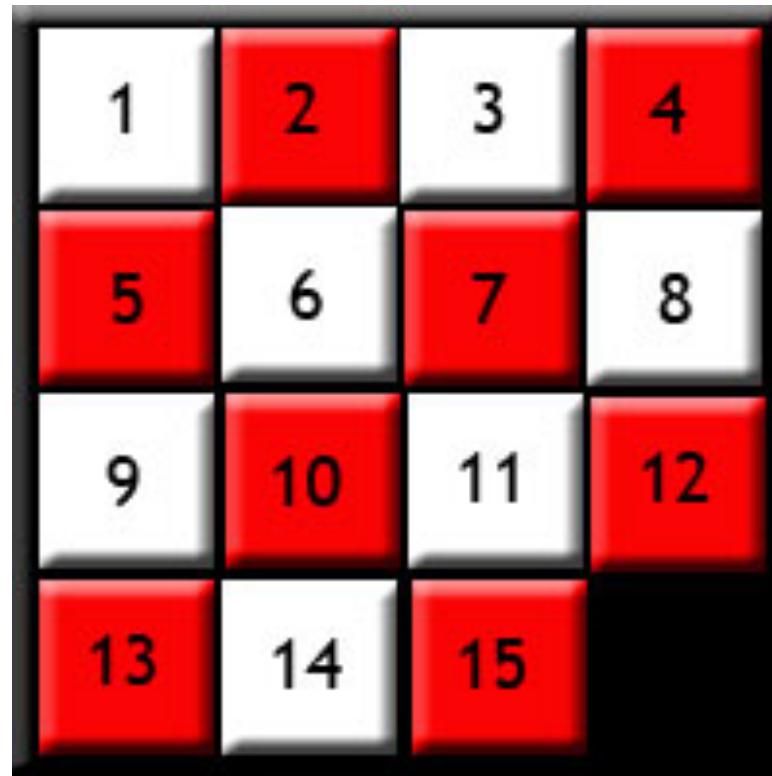
Goal: find the shortest path from **start** to **goal**.

shortest path: path with minimal total cost

Shortest Path Problem

- Shortest path problem is a special case of an episodic MDP:
 - deterministic transitions
 - only positive costs (negative rewards)
 - $\gamma = 1$
- *path*: ordered set of states from start to goal.
- Each path has a *total cost* associated with it, corresponding to the sum of the costs that is encountered when following the path.
- Goal is not to find the optimal policy, but to find the path from start to goal that has minimal total cost.

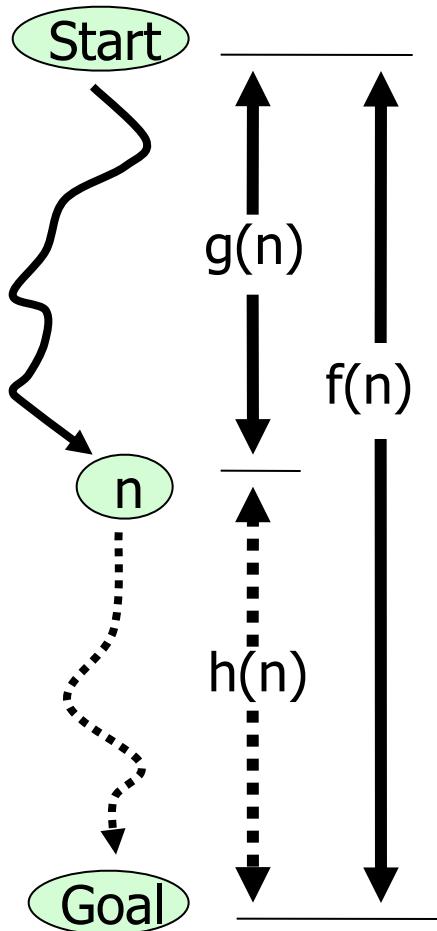
Heuristic Search Speed Up



10,461,394,944,000 states

heuristic search examines 36,000

Notation



- $g(n)$ = **cost-so-far**: cost from start to n along current path
- $h(n)$ = **cost-to-go** (heuristic): estimated cost from n to goal
- $f(n) = g(n)+h(n)$ = **total cost**: estimated cost from start to goal via n (using the current path to n)

More Notation

- $g^*(n)$: optimal cost from start to n
- $h^*(n)$: optimal cost from n to goal
- $f^*(n)$: optimal cost from start to goal via n

General Strategy

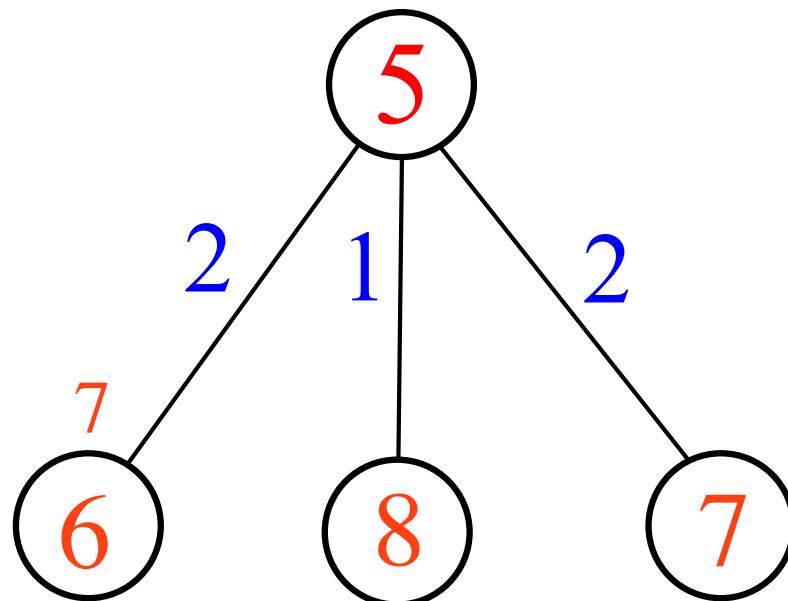
- Algorithms for shortest path problem aim to find $g^*(x)$.
- Once $g^*(x)$ is found, an optimal path can be easily derived.
- Main operator for updating $g(x)$, the estimate of $g^*(x)$, is the *expansion* operator.

Expansion

for each child:

- compute tentative cost
- if tentative cost < current cost of child, then replace current cost with tentative cost

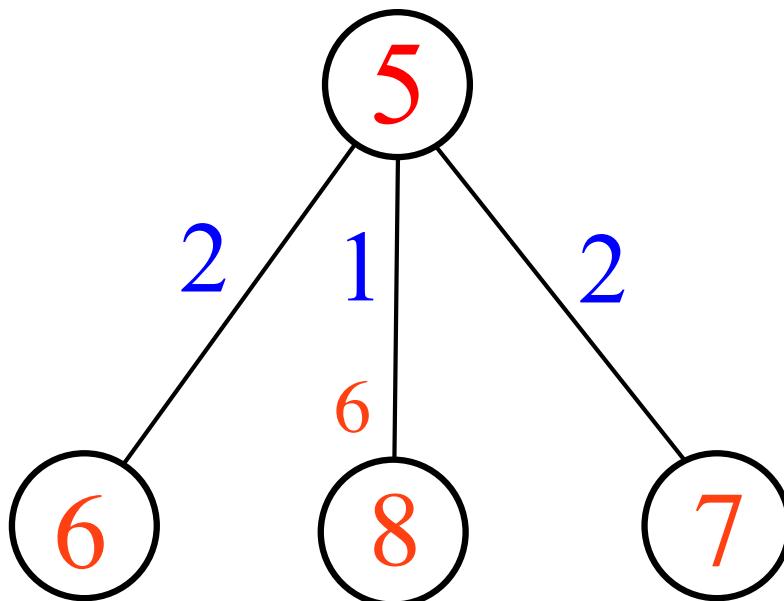
Expansion Example



red : $g(x)$

blue: $\text{cost}(x,y)$

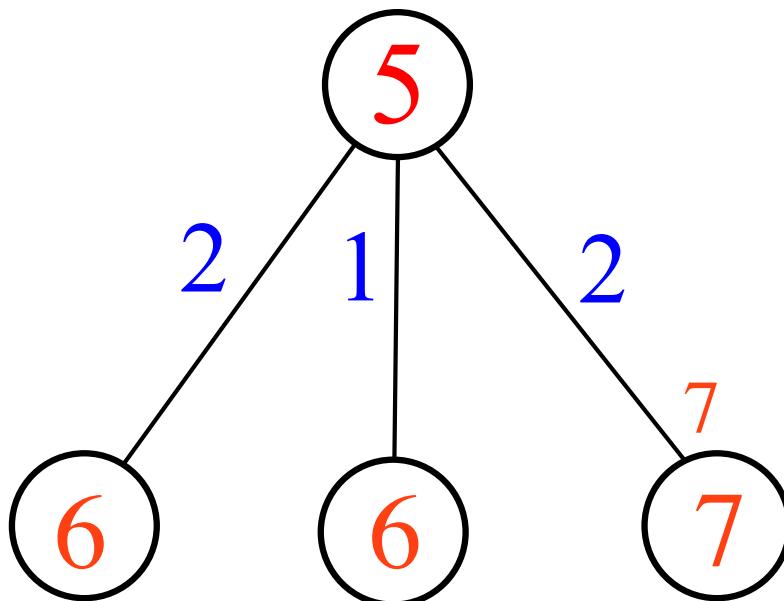
Expansion Example



red : $g(x)$

blue: $\text{cost}(x,y)$

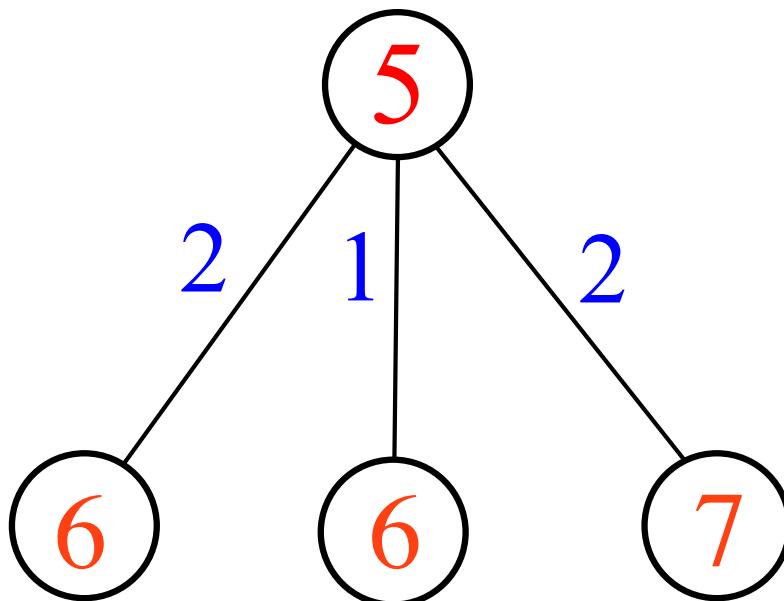
Expansion Example



red : $g(x)$

blue: $\text{cost}(x,y)$

Expansion Example



red : $g(x)$

blue: $\text{cost}(x,y)$

A* algorithm properties

- ‘best-first’ search: always expands the node with best total cost $f(n) = h(n) + g(n)$ first.
- during execution, the total cost, $f(n)$, and the cost-so-far, $g(n)$, change value; the cost-to-go, $h(n)$, is fixed.
- if $h(n) = 0$, A* is the same as uniform-cost search (Dijkstra’s algorithm)
- guaranteed to find shortest path if heuristic $h(x)$ is an optimistic estimate, that is, $h(x) \leq h^*(x)$
- such a heuristic is called ‘admissible’

Admissible Heuristic

- an admissible heuristic is an optimistic estimate of the cost-to-go:
$$h(x) \leq h^*(x) \quad \text{for all states } x$$
- higher heuristic values are better
- admissible heuristics guarantee that the optimal path will be found

Example Admissible Heuristic

- $h(n)$ estimates cost of cheapest path from node n to goal node
- Example: 8-puzzle

5	2	8
4	2	1
7	3	6

n

1	2	3
4	5	6
7	8	

goal

$$h_1(n) = \text{number of misplaced tiles} \\ = 6$$

$$h_2(n) = \text{sum of the distances of every tile to its goal position} \\ = 3 + 1 + 3 + 0 + 2 + 1 + 0 + 3 \\ = 13$$

Consistent Heuristics

- In general, it is possible that A* expands the same node more than once.
- If a heuristic is *consistent* it is guaranteed that a node will never expand more than once
- A heuristic is consistent if, for all states n:

$$h(n) \leq c(n,x) + h(x)$$

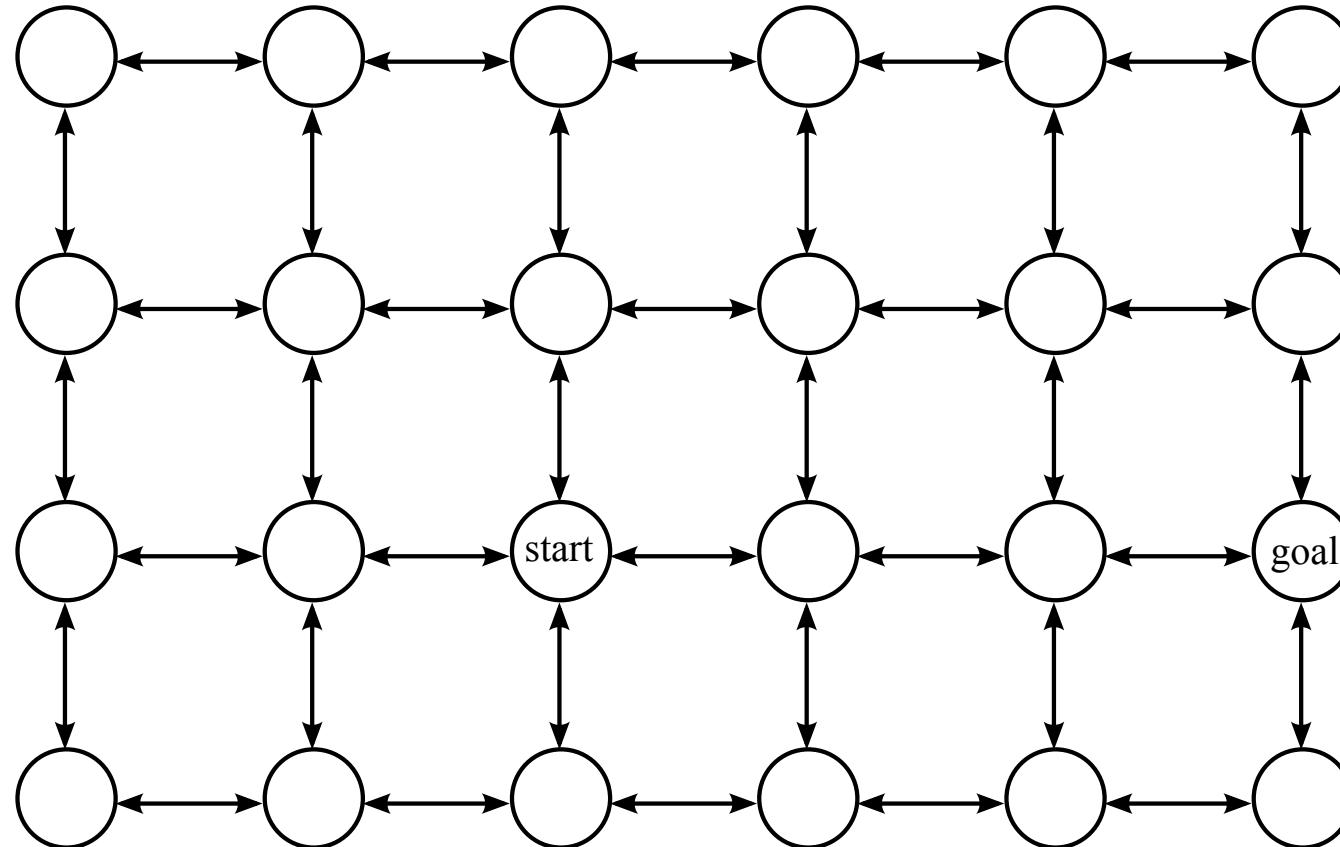
where x is the child of n
and $c(n,x)$ is cost from state n to state x

- Every consistent heuristic is also admissible. However, not every admissible heuristic is consistent.

A* algorithm

- Initialize:
 - put (start, 0) on OPEN; CLOSED is empty list
 - define cost-to-go function $h(x)$
- Do {until goal is on CLOSED}
 - if OPEN is empty, exit with failure
 - Select state n on OPEN with minimum $f(n) = g(n) + h(n)$
 - For each child x of n :
 - compute $g_n(x) = g(n) + \text{cost}(n,x)$
 - if x neither on OPEN nor on CLOSED:
add $(x, g_n(x))$ to OPEN
 - if x on OPEN and $g_n(x) < g(x)$:
replace $(x, g(x))$ with $(x, g_n(x))$
 - if x on CLOSED and $g_n(x) < g(x)$:
remove $(x, g(x))$ from CLOSED and
add $(x, g_n(x))$ to OPEN

A* example

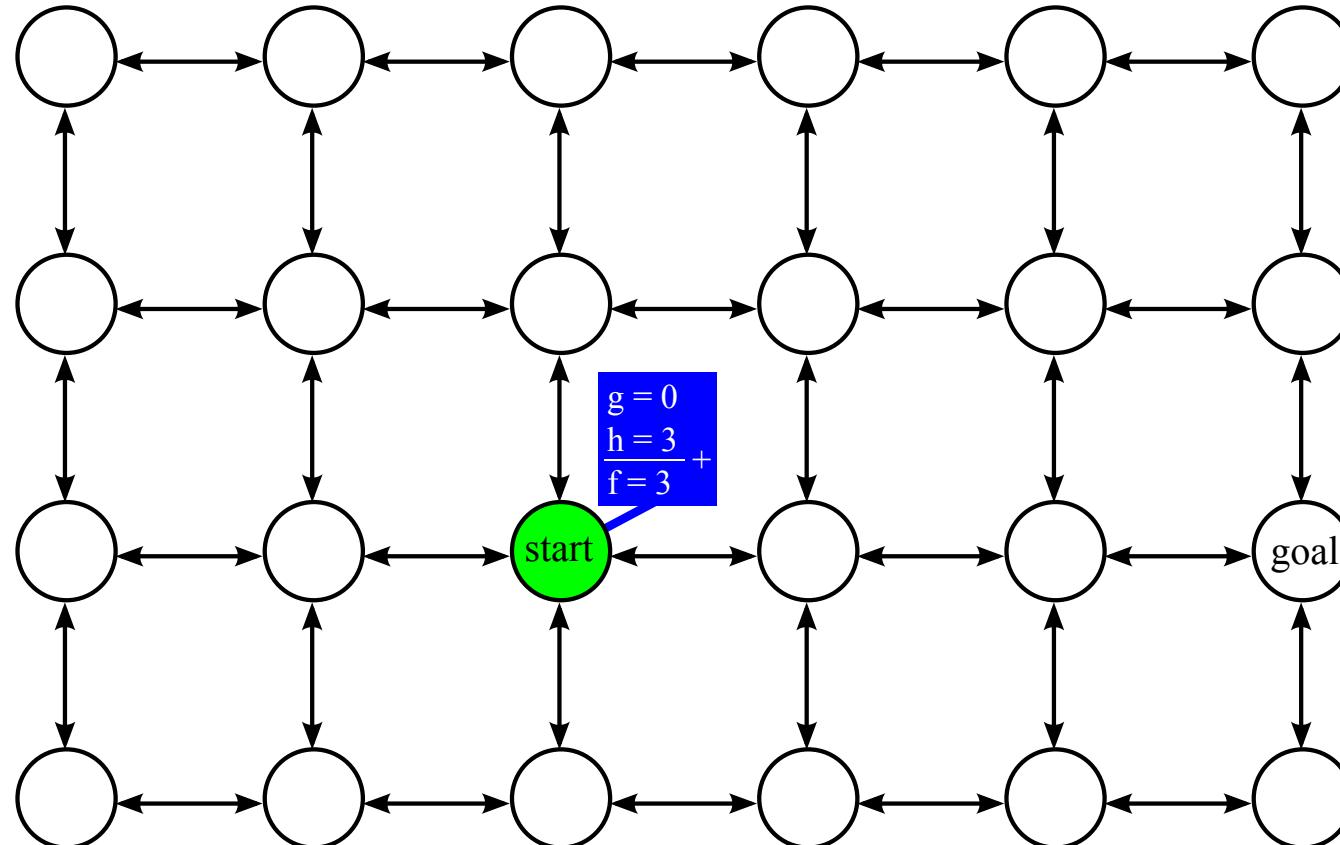


open list :

closed list ::

$h = h^*$
all costs: 1

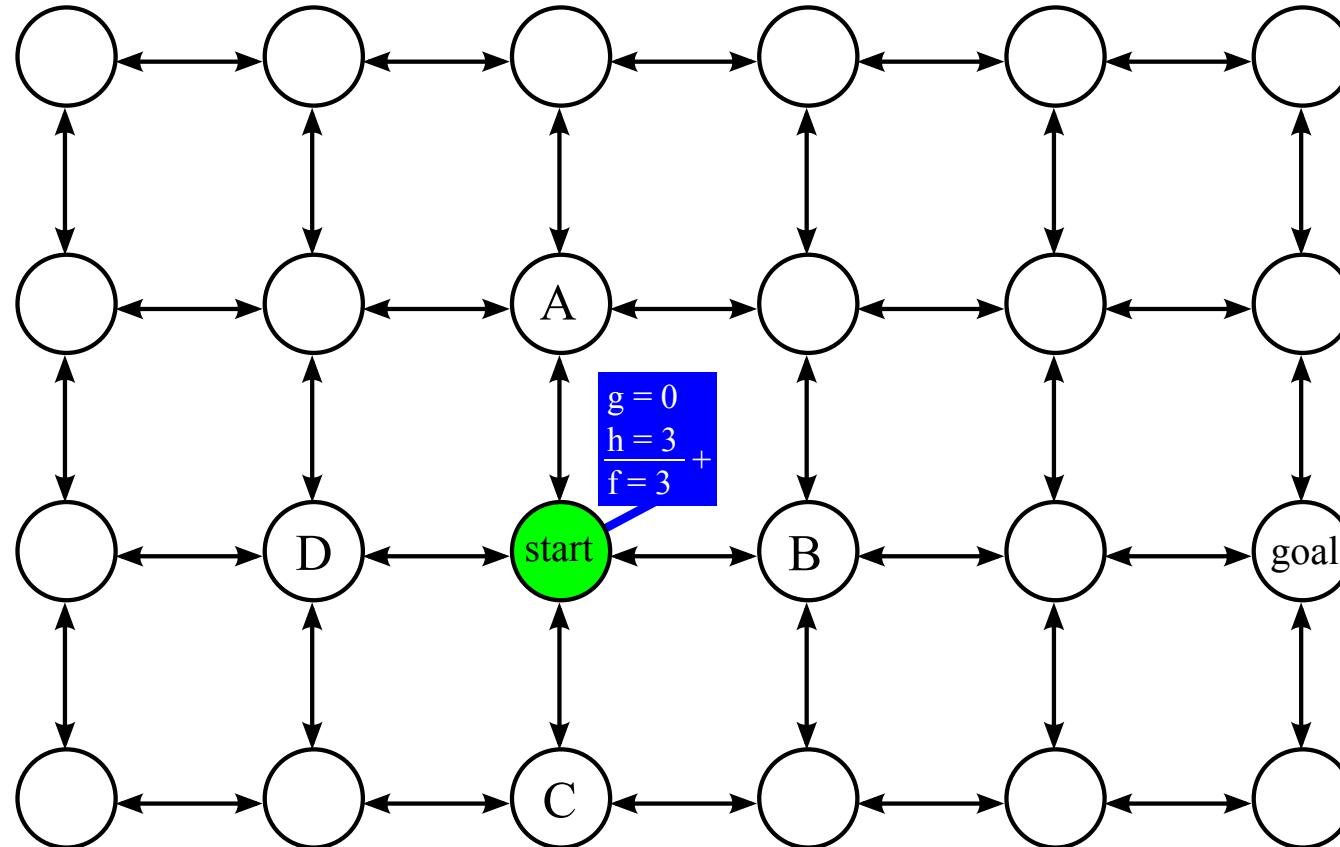
A* example



open list : start
closed list ::

h = h*
all costs: 1

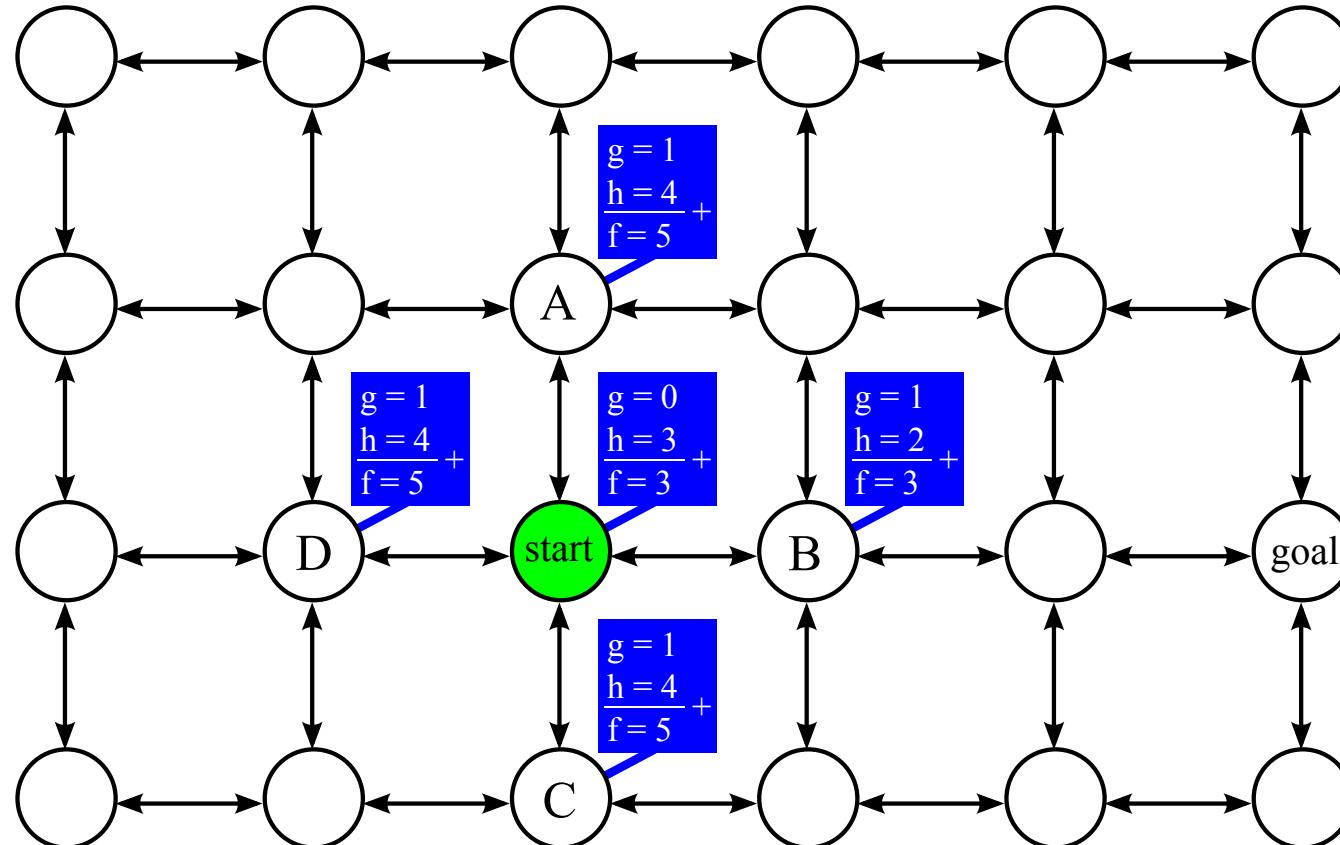
A* example



open list : start
closed list ::

$h = h^*$
all costs: 1

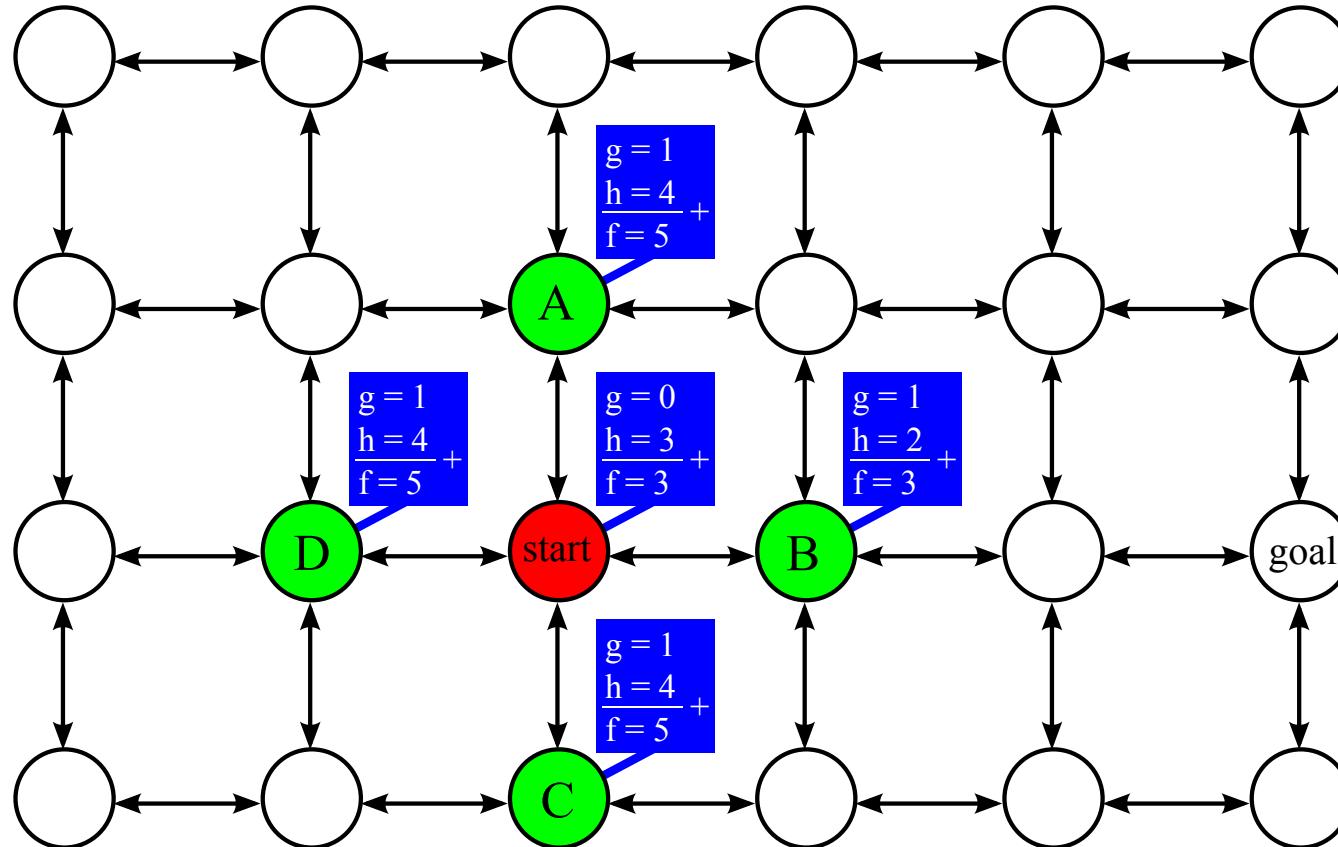
A* example



open list : start
closed list ::

$h = h^*$
all costs: 1

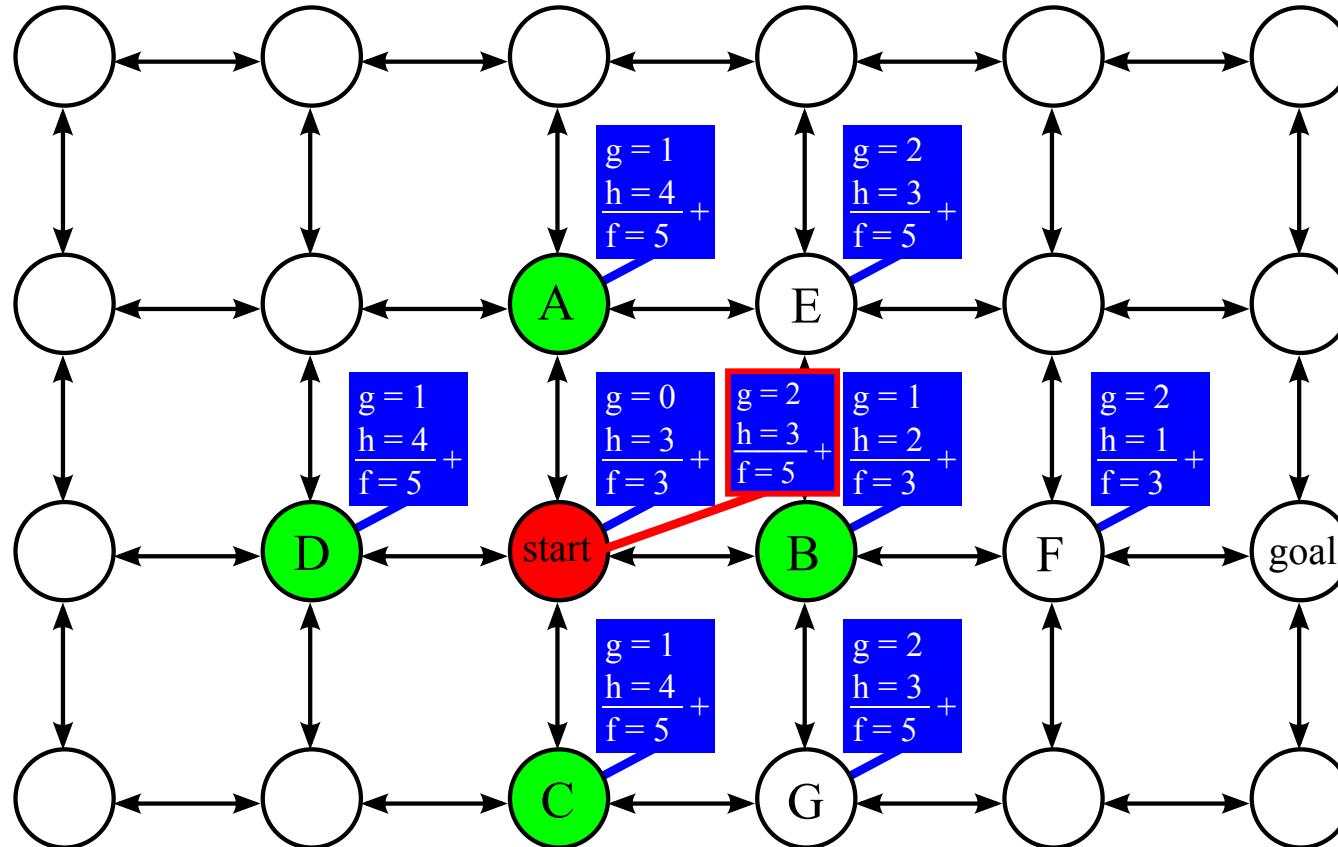
A* example



open list : A, B, C, D
closed list : start

$h = h^*$
all costs: 1

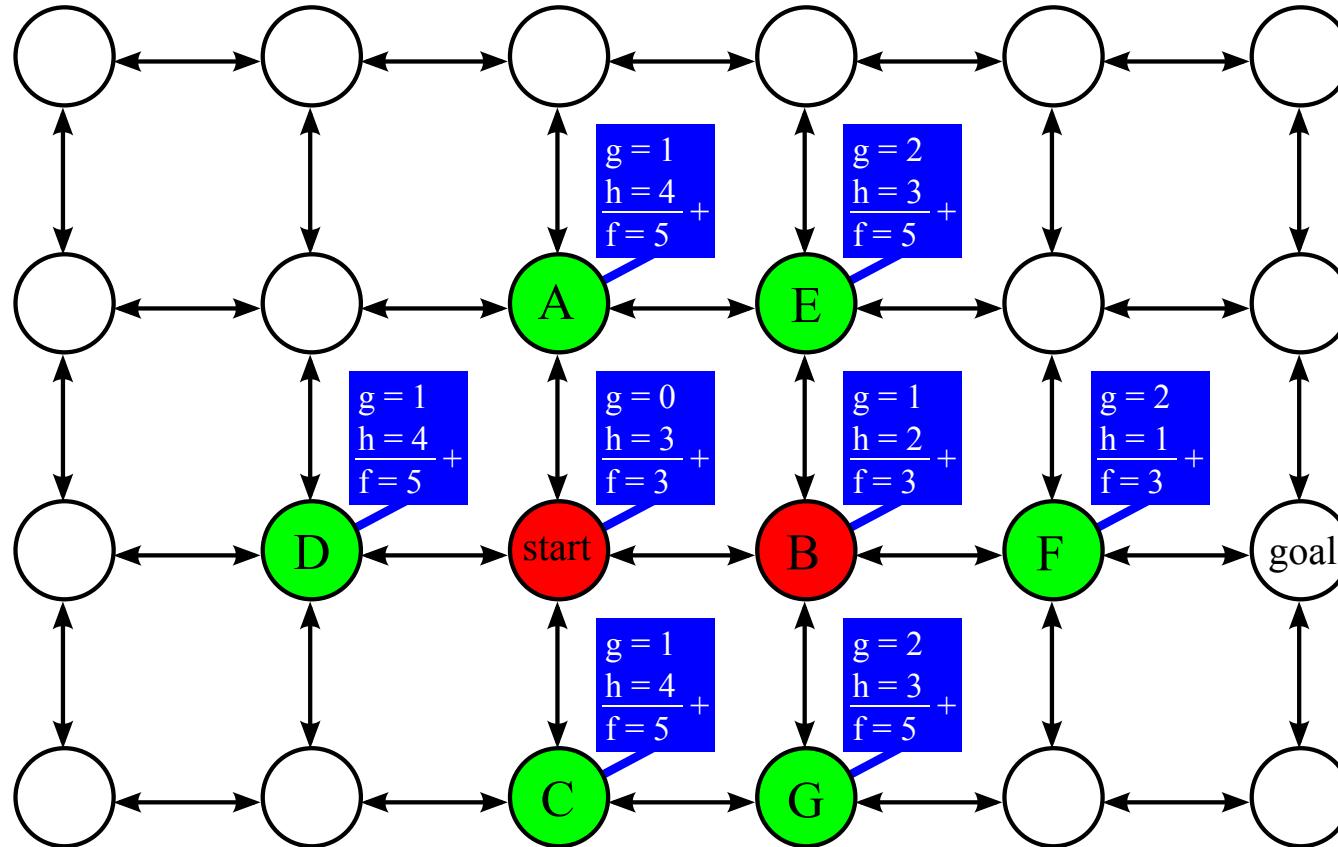
A* example



open list : A, B, C, D
closed list : start

$h = h^*$
all costs: 1

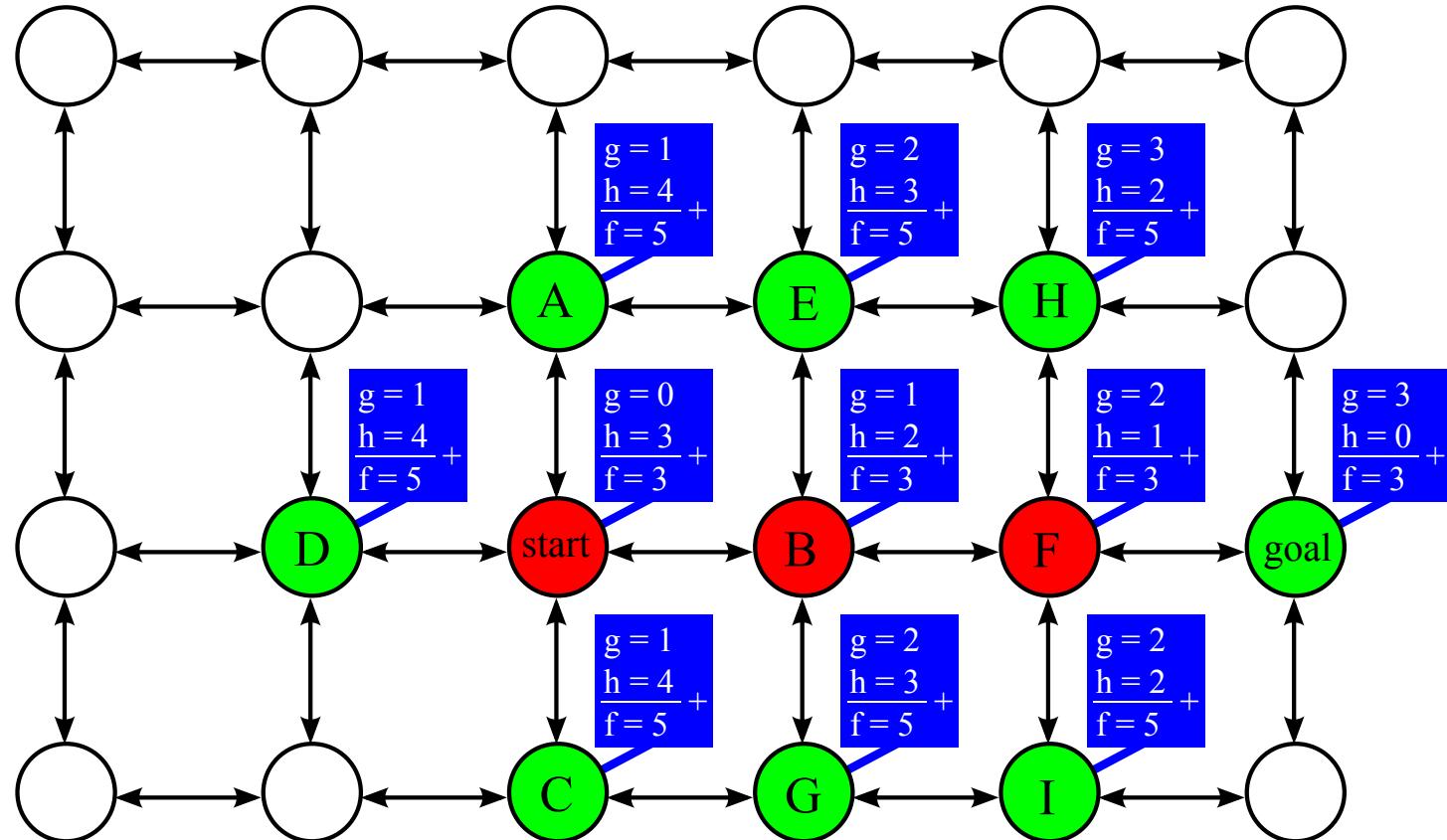
A* example



open list : A, C, D, E, F, G
closed list : start, B

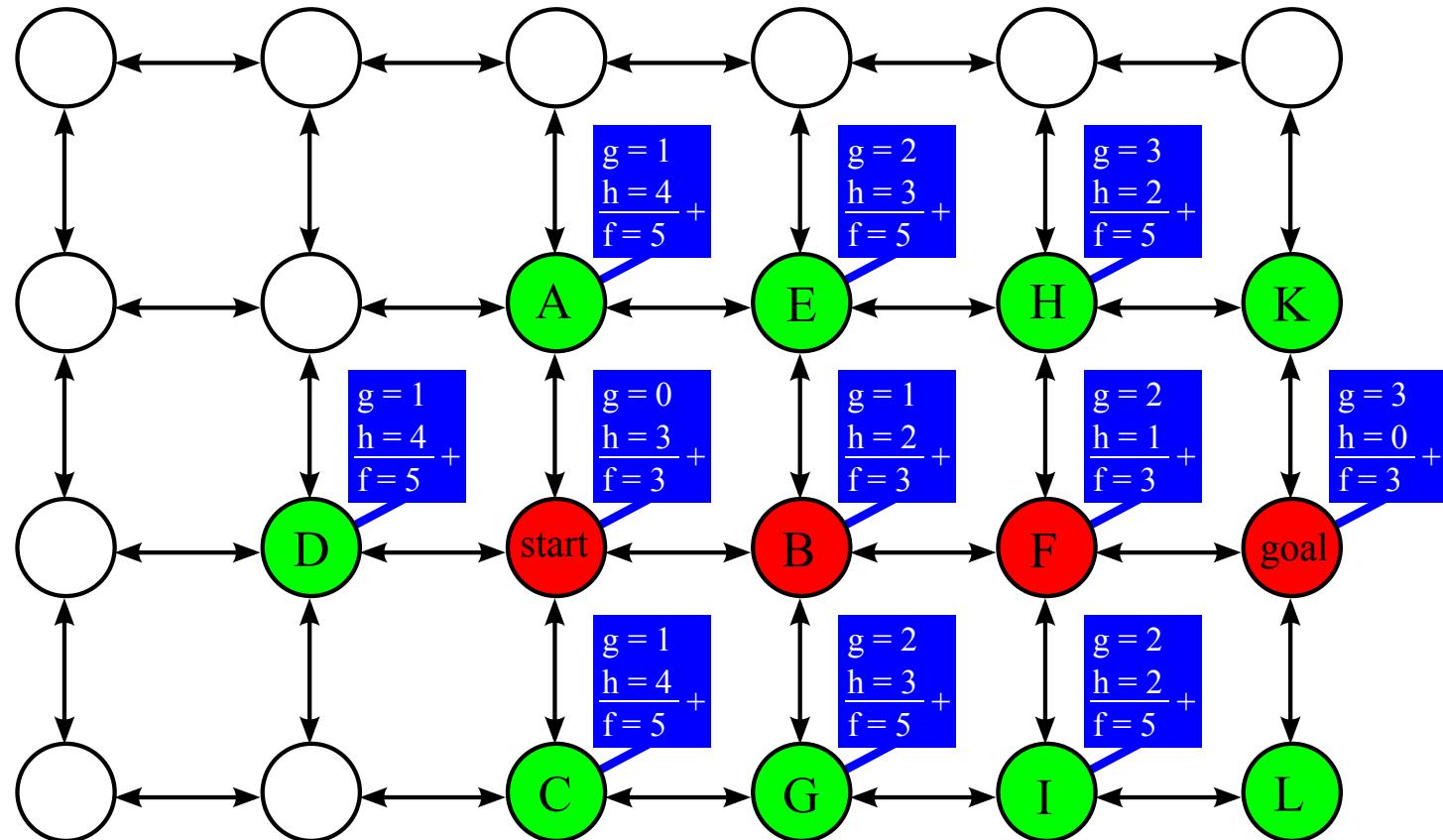
$h = h^*$
all costs: 1

A* example



$h = h^*$
 all costs: 1

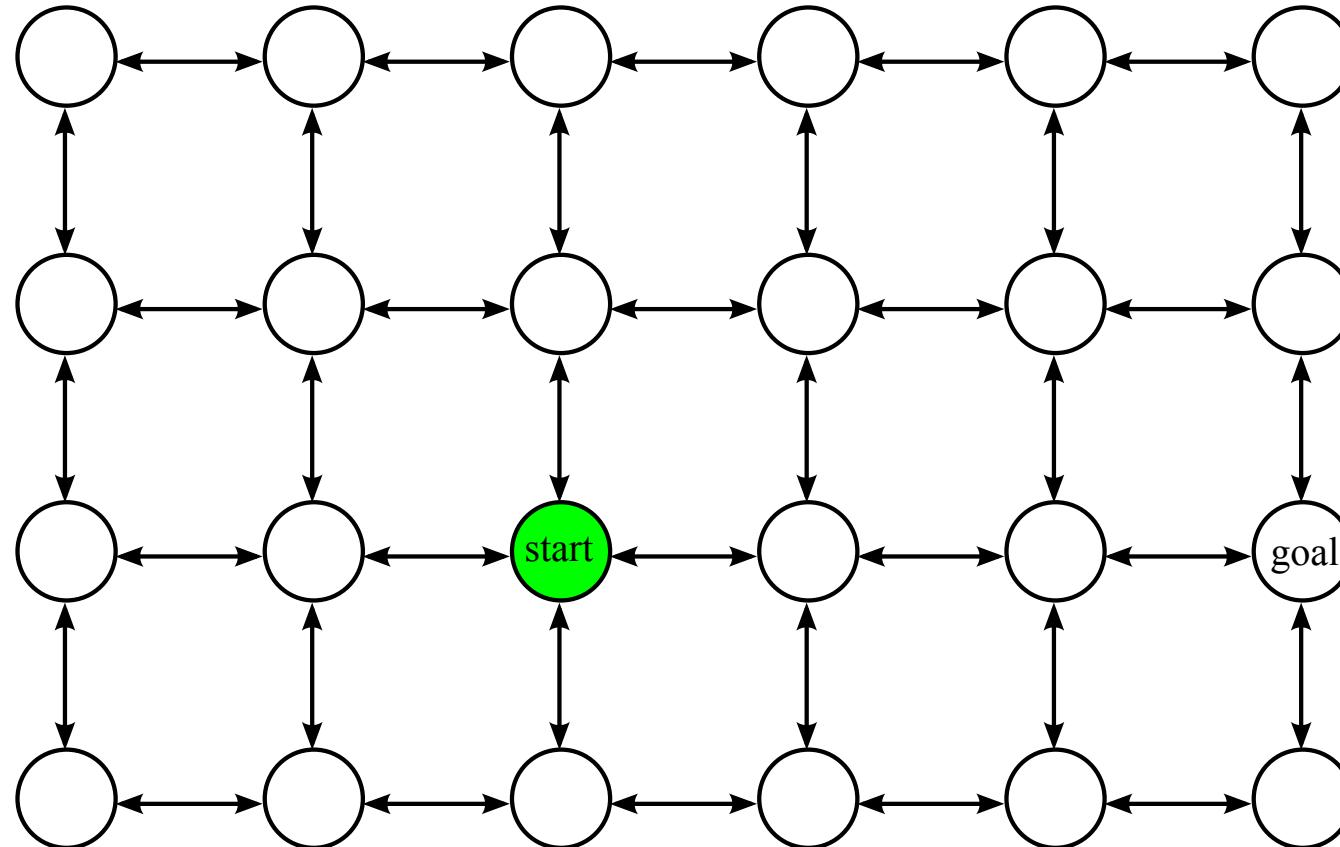
A* example



open list : A, C, D, E, G, H, I, K, L
closed list : start, B, F, goal

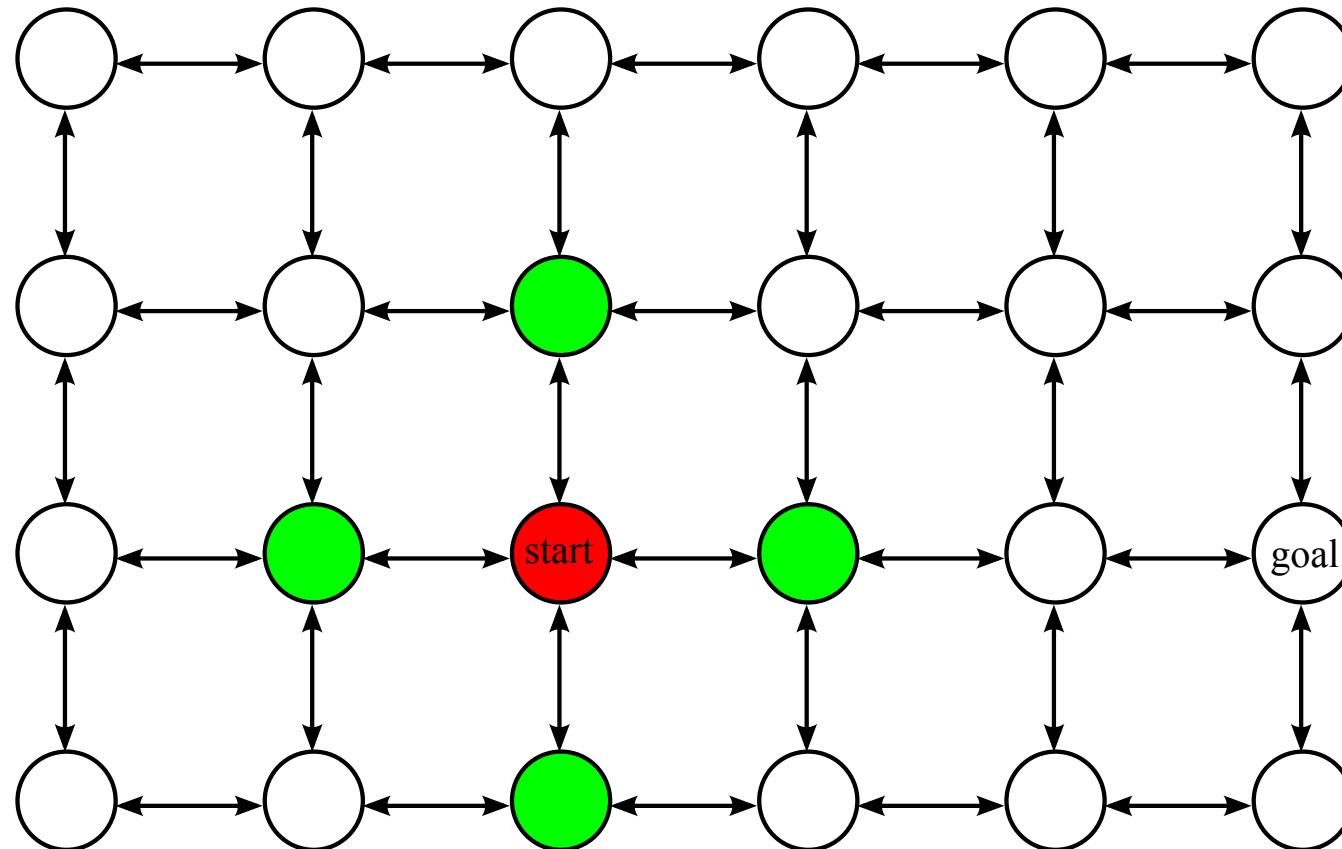
$h = h^*$
 all costs: 1

A* example



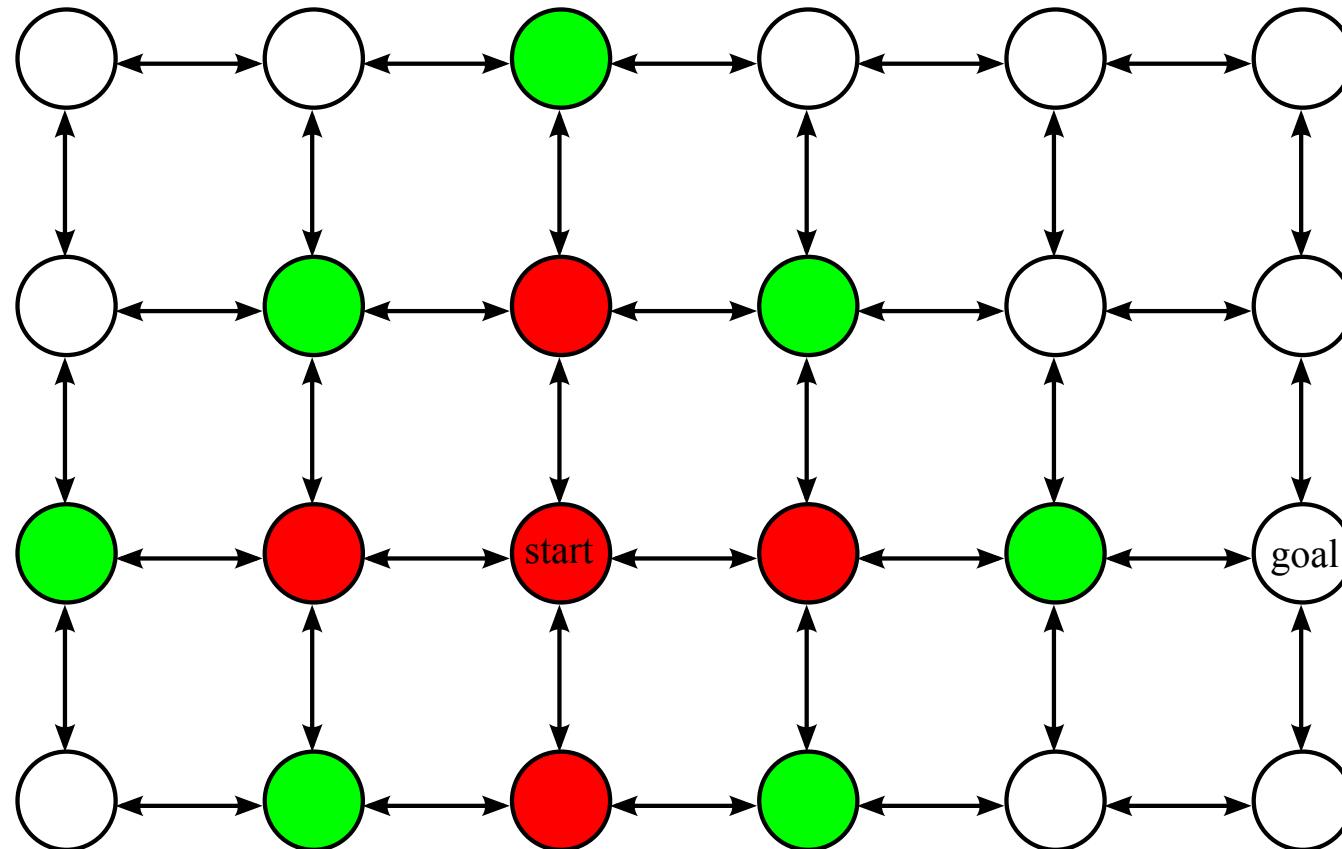
$h = 0$
all costs: 1

A* example



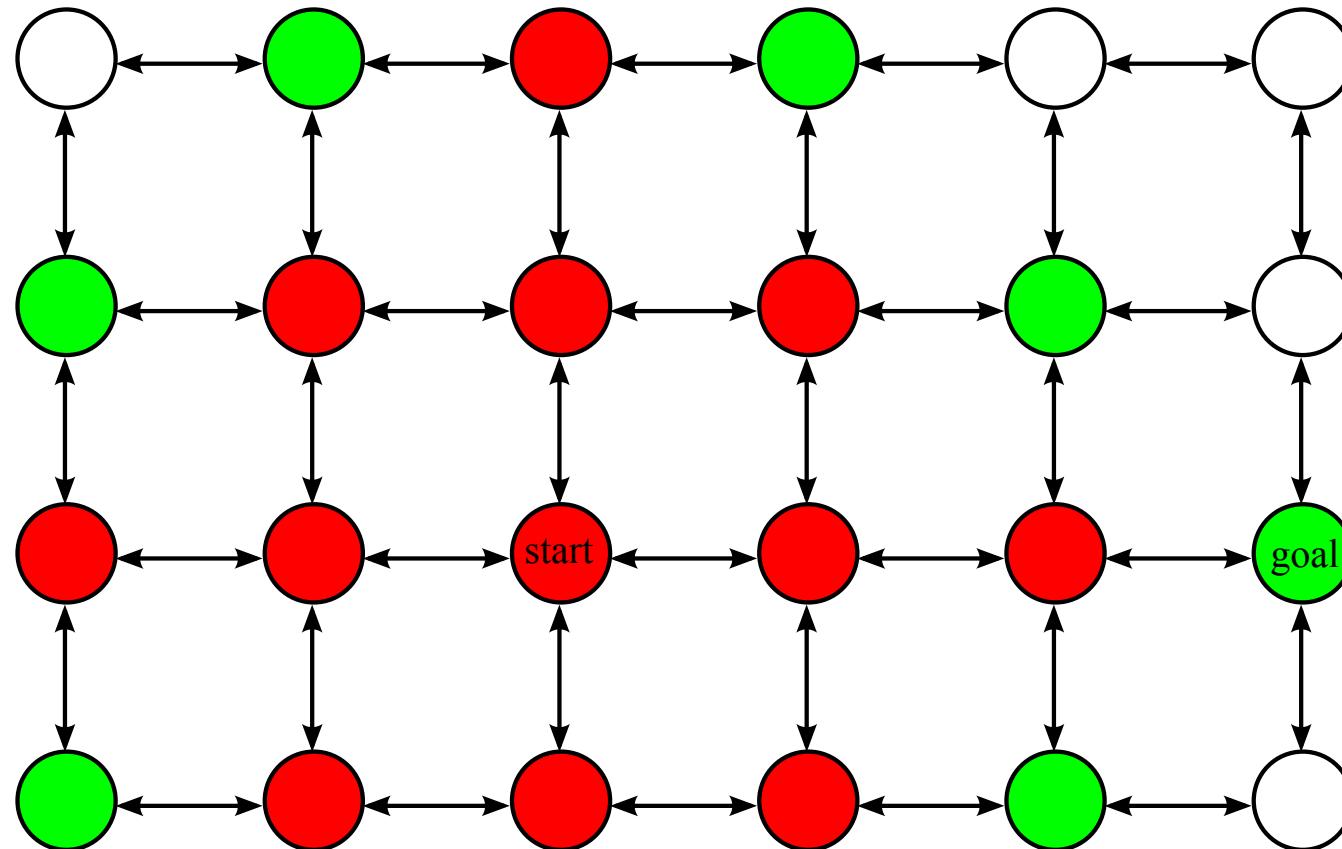
$h = 0$
all costs: 1

A* example



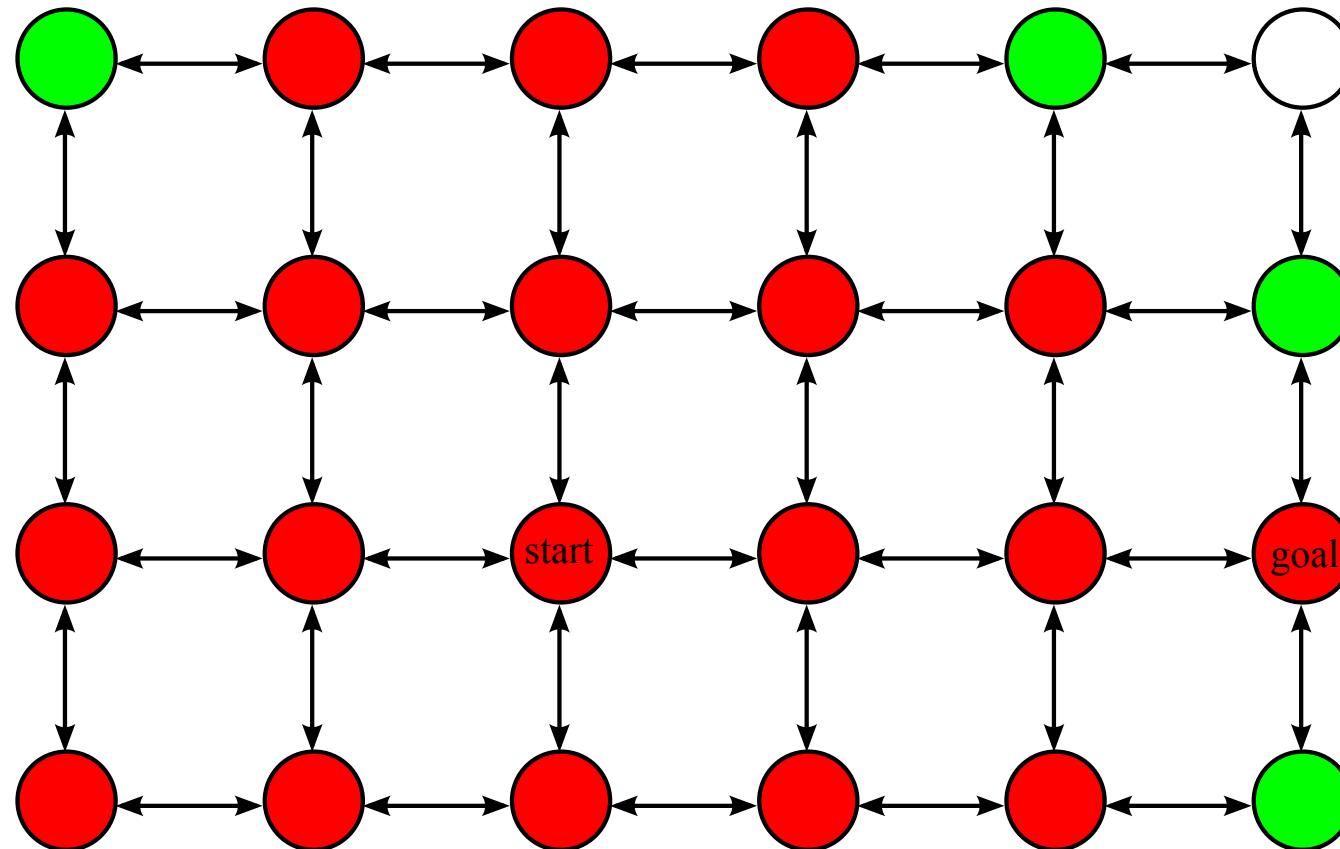
$h = 0$
all costs: 1

A* example



$h = 0$
all costs: 1

A* example



$h = 0$
all costs: 1

Quiz

Is the following statement **true** or **false**?

Statement 1:

If $h_1(x)$ is admissible and $h_2(x) = h_1(x) / 2$ for all x ,
then $h_2(x)$ is guaranteed to be admissible as well.

Quiz

Is the following statement **true** or **false**?

Statement 2:

If $h_1(x)$ is admissible and $h_2(x) = h_1(x) * 2$ for all x ,
then $h_2(x)$ is guaranteed to be admissible as well.

Quiz

Is the following statement **true** or **false**?

Statement 3:

If $h_1(x)$ and $h_2(x)$ are both admissible and
 $h_3(x) = \min(h_1(x), h_2(x))$ for all x ,
then $h_3(x)$ is guaranteed to be admissible as well.

Quiz

Is the following statement **true** or **false**?

Statement 4:

If $h_1(x)$ and $h_2(x)$ are both admissible and
 $h_3(x) = \max(h_1(x), h_2(x))$ for all x ,
then $h_3(x)$ is guaranteed to be admissible as well.

Quiz

Is the following statement **true** or **false**?

Statement 5:

If $h_1(x)$ and $h_2(x)$ are both admissible and
 $h_3(x) = \max(h_1(x), h_2(x))$ for all x ,
then $h_3(x)$ is at least as good as $h_1(x)$ and $h_2(x)$.

A* example

<http://www.vision.ee.ethz.ch/~cvcourse/astar/AStar.html>

Three kinds of evaluation function

- Heuristic functions of heuristic search
- "Static" Evaluation functions of a-b game-tree search
- Value functions of RL

Are these the same ideas, or significantly different?
What clear differences can you identify?

Goals for today

- Learn that policies can be optimized directly, without learning value functions, by *policy-gradient methods*
 - Glimpse a new goal for learning that does not involve discounting
 - Glimpse how one could learn real-valued (continuous) actions
- Glimpse how to handle hidden state

Policy-gradient methods

A new approach to control

Approaches to control

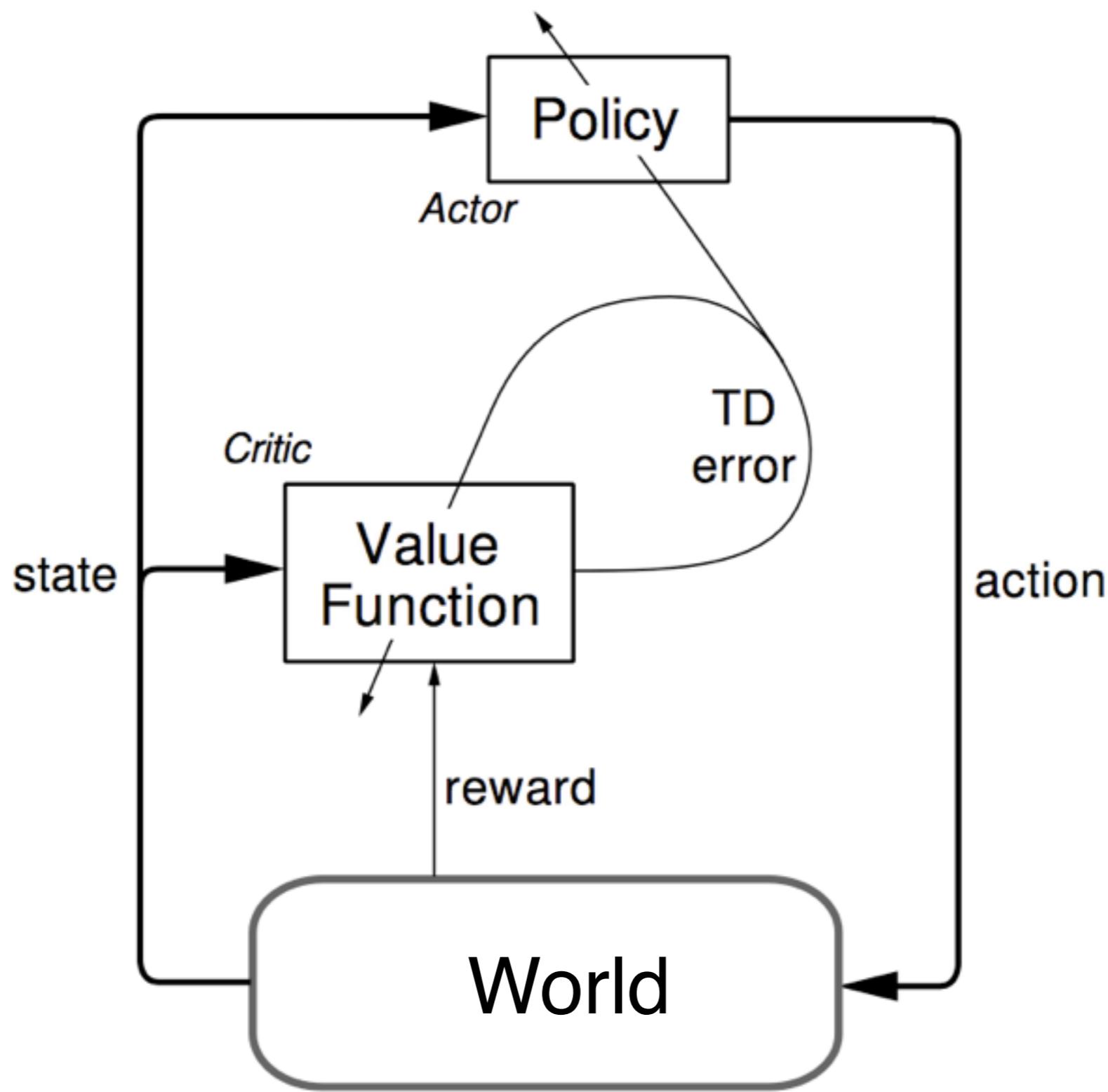
I. Previous approach: *Action-value methods*:

- learn the value of each action;
- pick the max (usually)

2. New approach: *Policy-gradient methods*:

- learn the parameters of a stochastic policy
- update by gradient ascent in performance
- includes *actor-critic methods*, which learn *both* value and policy parameters

Actor-critic architecture



Why approximate policies rather than values?

- In many problems, the policy is simpler to approximate than the value function
- In many problems, the optimal policy is stochastic
 - e.g., bluffing, POMDPs
- To enable smoother change in policies
- To avoid a search on every step (the max)
- To better relate to biology

Policy Approximation

- Policy = a function from state to action
 - How does the agent select actions?
 - In such a way that it can be affected by learning?
 - In such a way as to assure exploration?
- Approximation: there are too many states and/or actions to represent all policies
 - To handle large/continuous action spaces

Gradient-bandit algorithm

- Store action preferences $H_t(a)$ rather than action-value estimates $Q_t(a)$
- Instead of ε -greedy, pick actions by an exponential soft-max:

$$\Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^k e^{H_t(b)}} \doteq \pi_t(a)$$

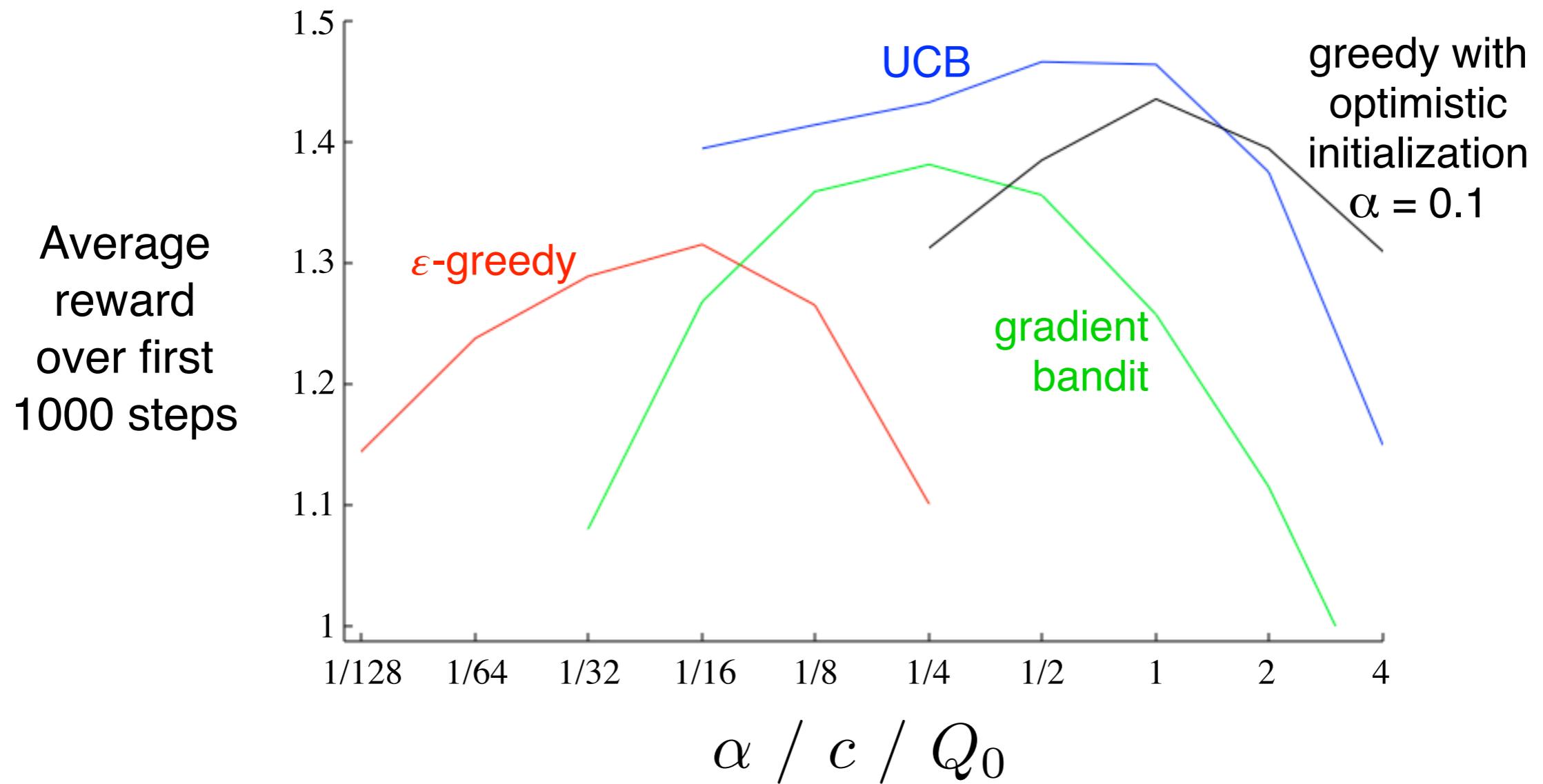
- Also store the sample average of rewards as \bar{R}_t
- Then update:

$$H_{t+1}(a) = H_t(a) + \alpha(R_t - \bar{R}_t)(1_{a=A_t} - \pi_t(a))$$

I or 0, depending on whether
the predicate (subscript) is true

$$\frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t)$$

Summary Comparison of Bandit Algorithms



Gradient-bandit algorithms on the 10-armed testbed

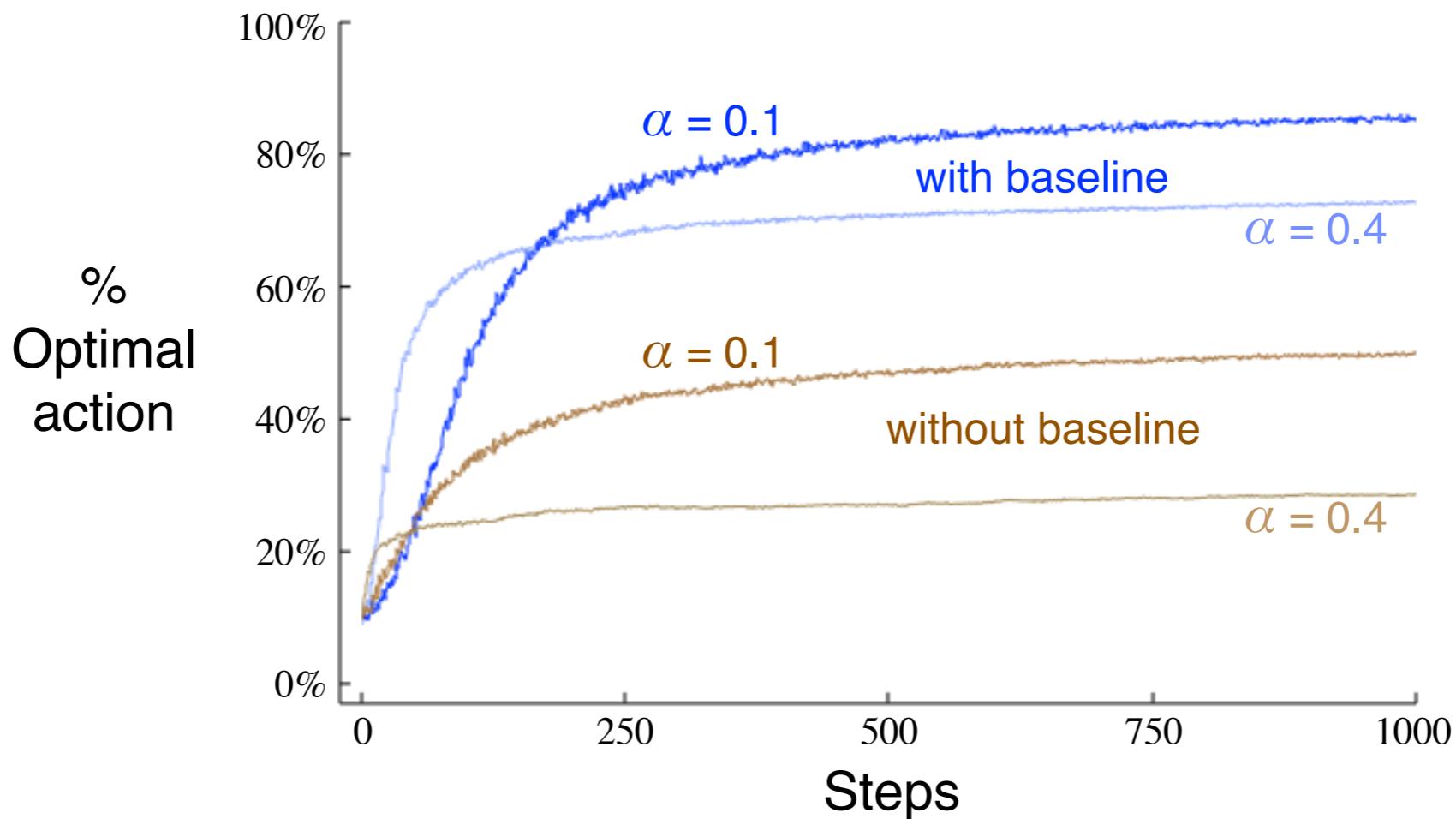


Figure 2.6: Average performance of the gradient-bandit algorithm with and without a reward baseline on the 10-armed testbed when the $q_*(a)$ are chosen to be near +4 rather than near zero.

eg, linear-exponential policies (discrete actions)

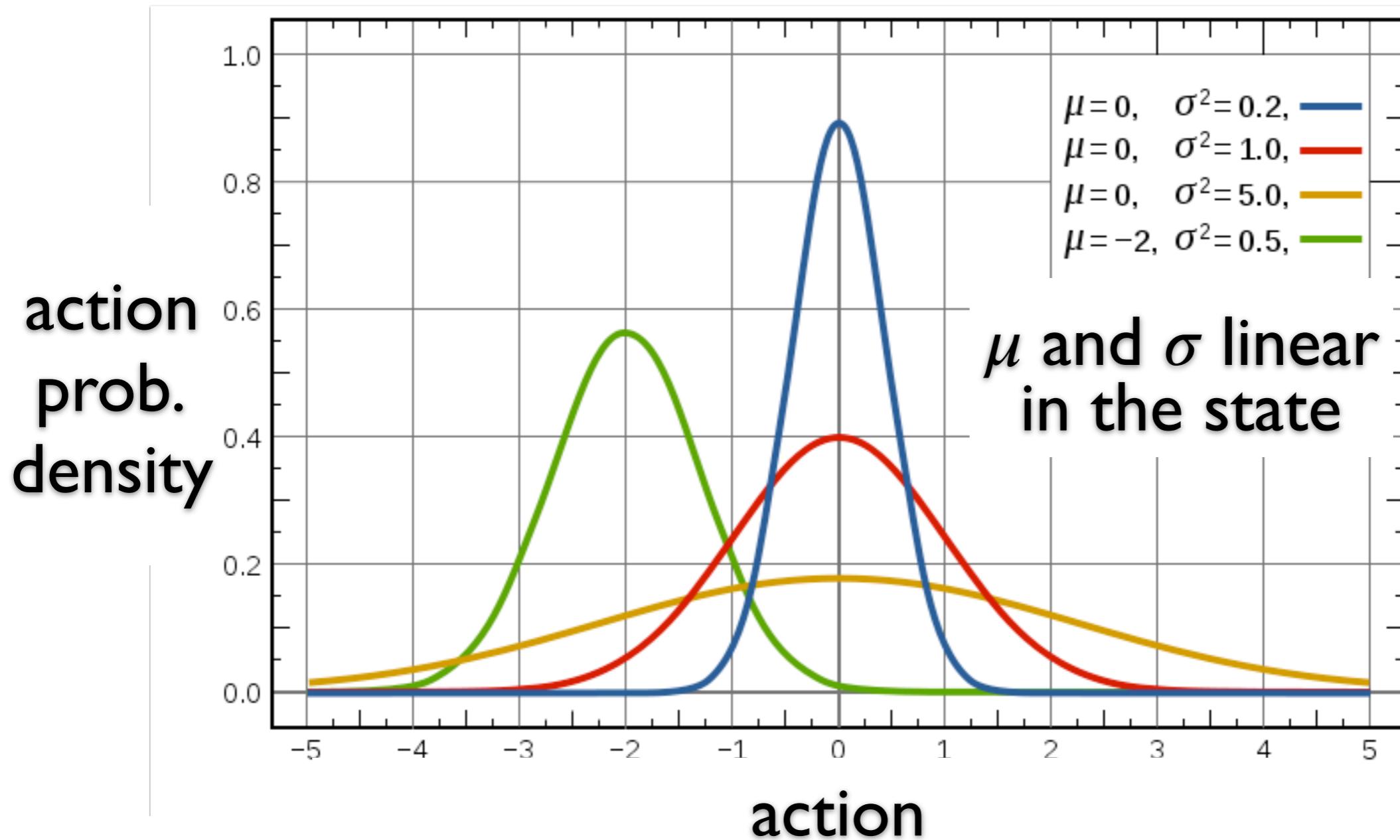
- The “preference” for action a in state s is linear in θ and a state-action feature vector $\phi(s,a)$
- The probability of action a in state s is exponential in its preference

$$\pi(a|s, \theta) \doteq \frac{\exp(\theta^\top \phi(s, a))}{\sum_b \exp(\theta^\top \phi(s, b))}$$

- Corresponding eligibility function:

$$\frac{\nabla \pi(a|s, \theta)}{\pi(a|s, \theta)} = \phi(s, a) - \sum_b \pi(b|s, \theta) \phi(s, b)$$

eg, linear-gaussian policies (continuous actions)



eg, linear-gaussian policies (continuous actions)

- The mean and std. dev. for the action taken in state s are linear and linear-exponential in

$$\theta \doteq (\theta_\mu^\top; \theta_\sigma^\top)^\top \quad \mu(s) \doteq \theta_\mu^\top \phi(s) \quad \sigma(s) \doteq \exp(\theta_\sigma^\top \phi(s))$$

- The probability density function for the action taken in state s is gaussian

$$\pi(a|s, \theta) \doteq \frac{1}{\sigma(s)\sqrt{2\pi}} \exp\left(-\frac{(a - \mu(s))^2}{2\sigma(s)^2}\right)$$

The generality of the policy-gradient strategy

- Can be applied whenever we can compute the effect of parameter changes on the action probabilities, $\nabla \pi(A_t | S_t, \theta)$
- E.g., has been applied to spiking neuron models
- There are many possibilities other than linear-exponential and linear-gaussian
 - e.g., mixture of random, argmax, and fixed-width gaussian; learn the mixing weights, drift/diffusion models

Steps to understanding Policy-gradient methods

- Policy approximation $\pi(a|s, \theta)$
- The average-reward (reward rate) objective $\bar{r}(\theta)$
- Stochastic gradient ascent/descent $\Delta\theta_t \approx \alpha \frac{\partial \bar{r}(\theta)}{\partial \theta}$
- The policy-gradient theorem and its proof
- Approximating the gradient
- Eligibility functions for a few cases
- A complete algorithm

We should never discount
when approximating policies!



γ is ok if there is a
start state/distribution

Average reward setting

- All rewards are compared to the average reward

$$q_{\pi}(s, a) = \mathbb{E} \left[\sum_{t=1}^{\infty} R_t - \bar{r}(\pi) \middle| S_0 = s, A_0 = a \right]$$

- where

$$\bar{r}(\pi) = \lim_{t \rightarrow \infty} \frac{1}{t} \mathbb{E} [R_1 + R_2 + \cdots + R_t \mid A_{0:t-1} \sim \pi]$$

- and we learn an approximation

$$\bar{R}_t \approx \bar{r}(\pi_t)$$

Average-reward Q-learning (R-learning)

Initialize \bar{R} and $Q(s, a)$, for all s, a , arbitrarily

Repeat forever:

$S \leftarrow$ current state

Choose action A in S using behavior policy (e.g., ϵ -greedy)

Take action A , observe R, S'

$\delta \leftarrow R - \bar{R} + \max_a Q(S', a) - Q(S, A)$

$Q(S, A) \leftarrow Q(S, A) + \alpha\delta$

If $Q(S, A) = \max_a Q(S, a)$, then:

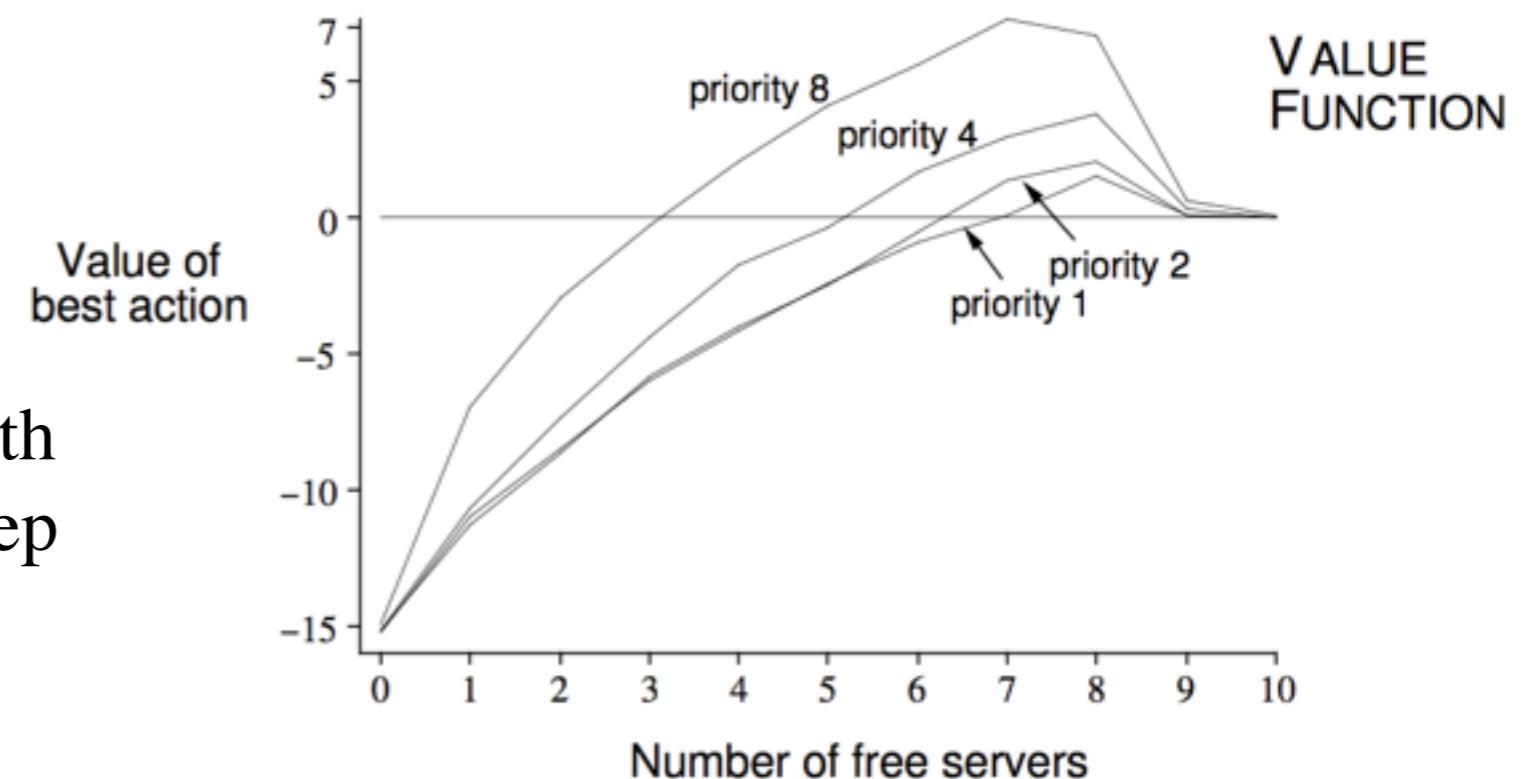
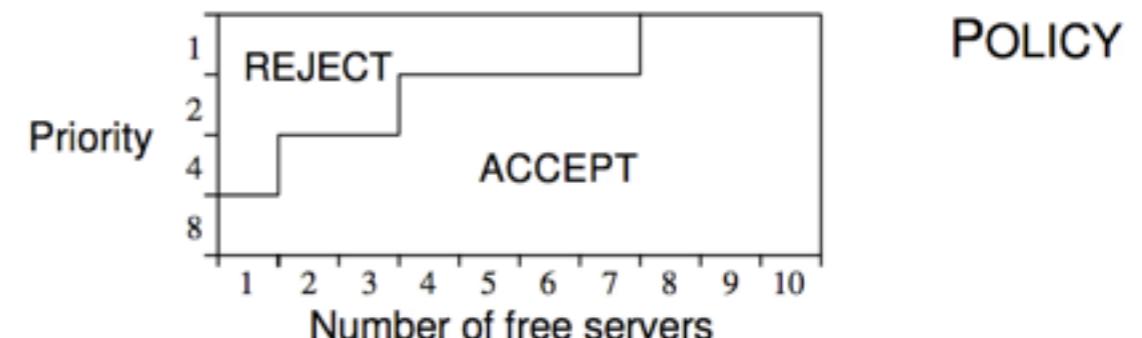
$\bar{R} \leftarrow \bar{R} + \beta\delta$

Access-Control Queuing Task

- n servers
- Customers have four different priorities, which pay reward of 1, 2, 4, or 8, if served
- At each time step, customer at head of queue is accepted (assigned to a server) or removed from the queue
- Proportion of randomly distributed high priority customers in queue is h
- Busy server becomes free with probability p on each time step
- Statistics of arrivals and departures are unknown

Apply R-learning

$$n=10, h=.5, p=.06$$



On-policy average-reward with traces and linear FA

$$\theta_{t+1} \doteq \theta_t + \alpha \delta_t \mathbf{e}_t$$

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \theta_t^\top \phi_{t+1} - \theta_t^\top \phi_t$$

$$\bar{R}_{t+1} \doteq \bar{R}_t + \beta \delta_t$$

Complete PG algorithm

Initialize parameters of policy $\theta \in \mathbb{R}^n$, and state-value function $w \in \mathbb{R}^m$

Initialize eligibility traces $e^\theta \in \mathbb{R}^n$ and $e^w \in \mathbb{R}^m$ to 0

Initialize $\bar{R} = 0$

On each step, in state S :

Choose A according to $\pi(\cdot|S, \theta)$

Take action A , observe S', R

$$\delta \leftarrow R - \bar{R} + \hat{v}(S', w) - \hat{v}(S, w)$$

form TD error from critic

$$\bar{R} \leftarrow \bar{R} + \alpha^\theta \delta$$

update average reward estimate

$$e^w \leftarrow \lambda e^w + \nabla_w \hat{v}(S, w)$$

update eligibility trace for critic

$$w \leftarrow w + \alpha^w \delta e^w$$

update critic parameters

$$e^\theta \leftarrow \lambda e^\theta + \frac{\nabla \pi(A|S, \theta)}{\pi(A|S, \theta)}$$

update eligibility trace for actor

$$\theta \leftarrow \theta + \alpha^\theta \delta e^\theta$$

update actor parameters

Goals for today

- Learn that policies can be optimized directly, without learning value functions, by *policy-gradient methods*
 - Glimpse a new goal for learning that does not involve discounting
 - Glimpse how one could learn real-valued (continuous) actions
- Glimpse how to handle hidden state

Hidden State

What it is
What to do about it

What is hidden state?

- Sometimes the environment includes state variables that are not visible to the agent
 - the agent sees only *observations*, not state
 - e.g., the object in the box, or in other rooms, velocities, even real positions as distinct from sensor readings
- This makes the environment **Non-Markov**

- So all real problems involve extensive hidden state
- The agent's approximation to the hidden state of the environment will be imperfect and non-Markov
- But all of our methods rely on the Markov (state) property to some extent
- What to do?

**DON'T
PANIC**

The usual over-reaction

- You could introduce a new mathematical theory like POMDPs (Partially Observable MDPs) or HMMs (Hidden Markov Models)
- Relying on *complete models* of the hidden underlying environment and observation generators (even though these things are all hidden)
- Thereby making both learning and planning *intractably complex*

There may be nothing you can do

- If the agent's approximate state is very poor, then any policy based on it will be poor

Use your tools!

I. Function approximation

- Features can be anything; they can be an arbitrary summary of past observations
 - Nothing in our theory relies on the features being Markov
- ∴ FA will work ok with non-Markov features

Use your tools!

2. Eligibility traces

- Monte Carlo methods are much less reliant on having a good state approximation
 - because they don't bootstrap
- Eligibility traces allow our learning methods to be fully or partly Monte Carlo
 - and thus resistant to hidden state

Remember: the bound of approximation accuracy as function of λ

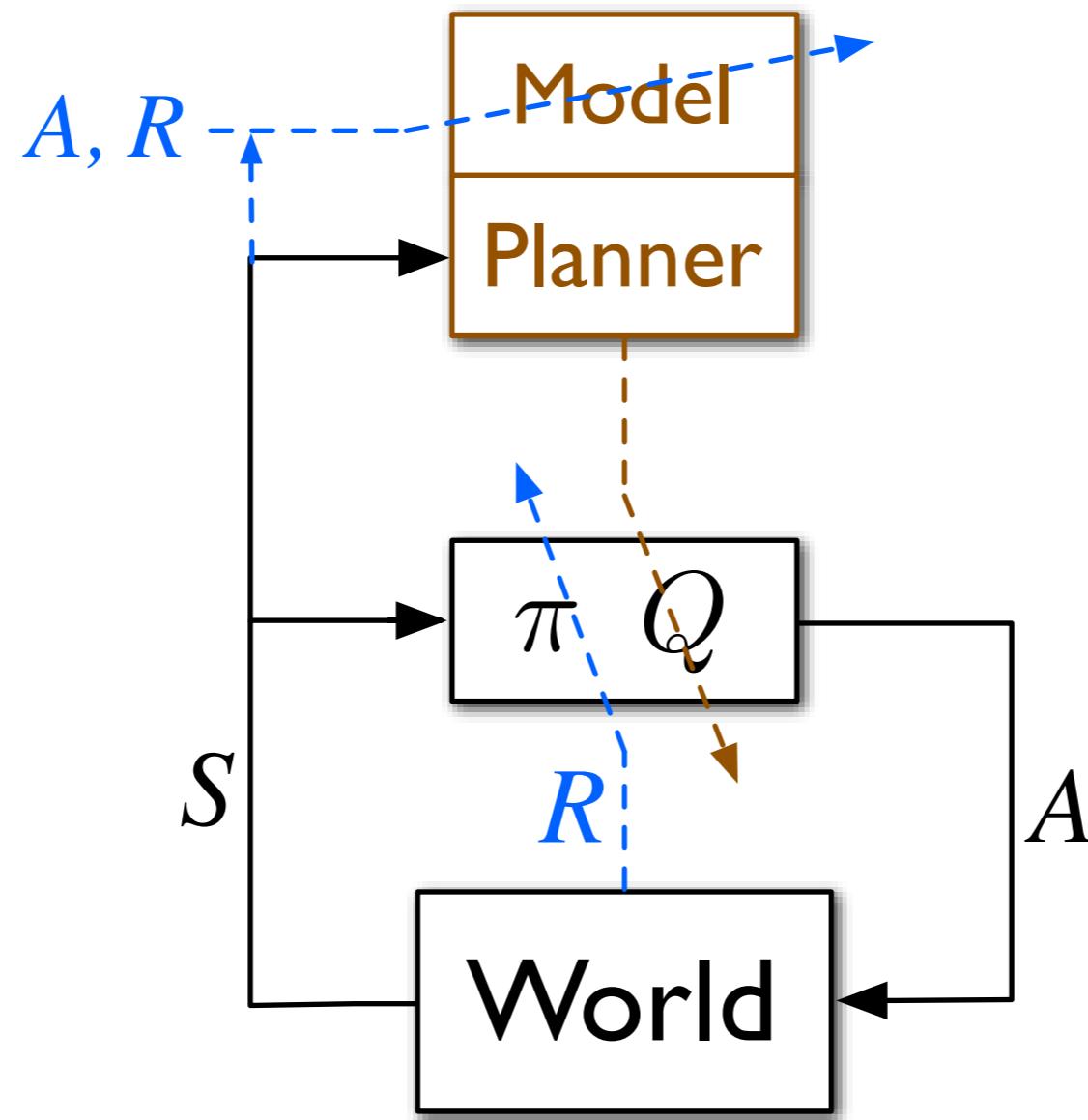
Remember: why do we ever bootstrap?

The long-term solution

- Don't panic
- Use your tools
- Embrace approximation
- Develop a recurrent process for updating the agent's approximate state
 - Accept that it will be approximate, imperfect
 - And that it will have to monitored, debugged, improved...forever approximate

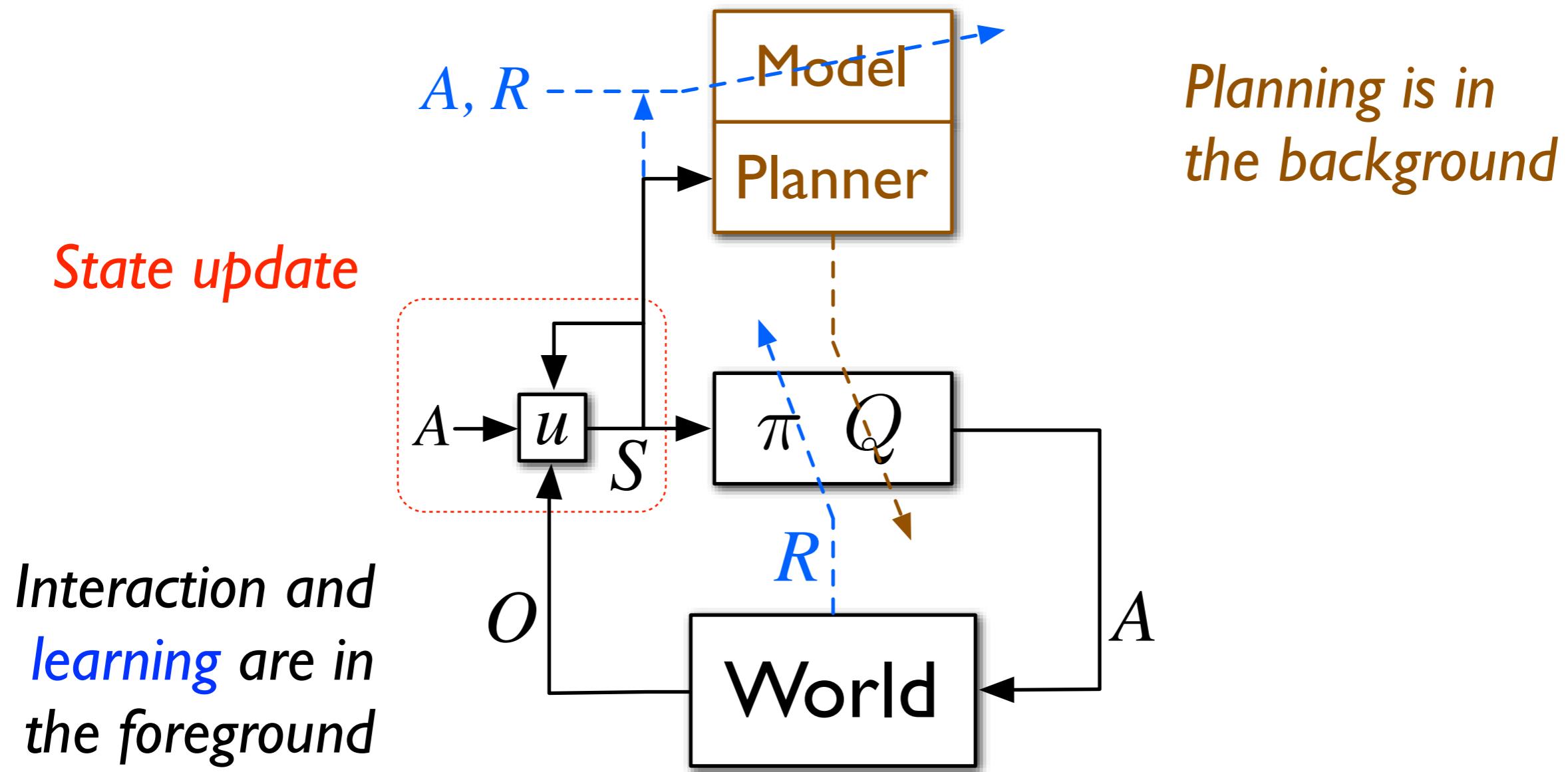
Foreground-background architecture

Interaction and learning are in the foreground

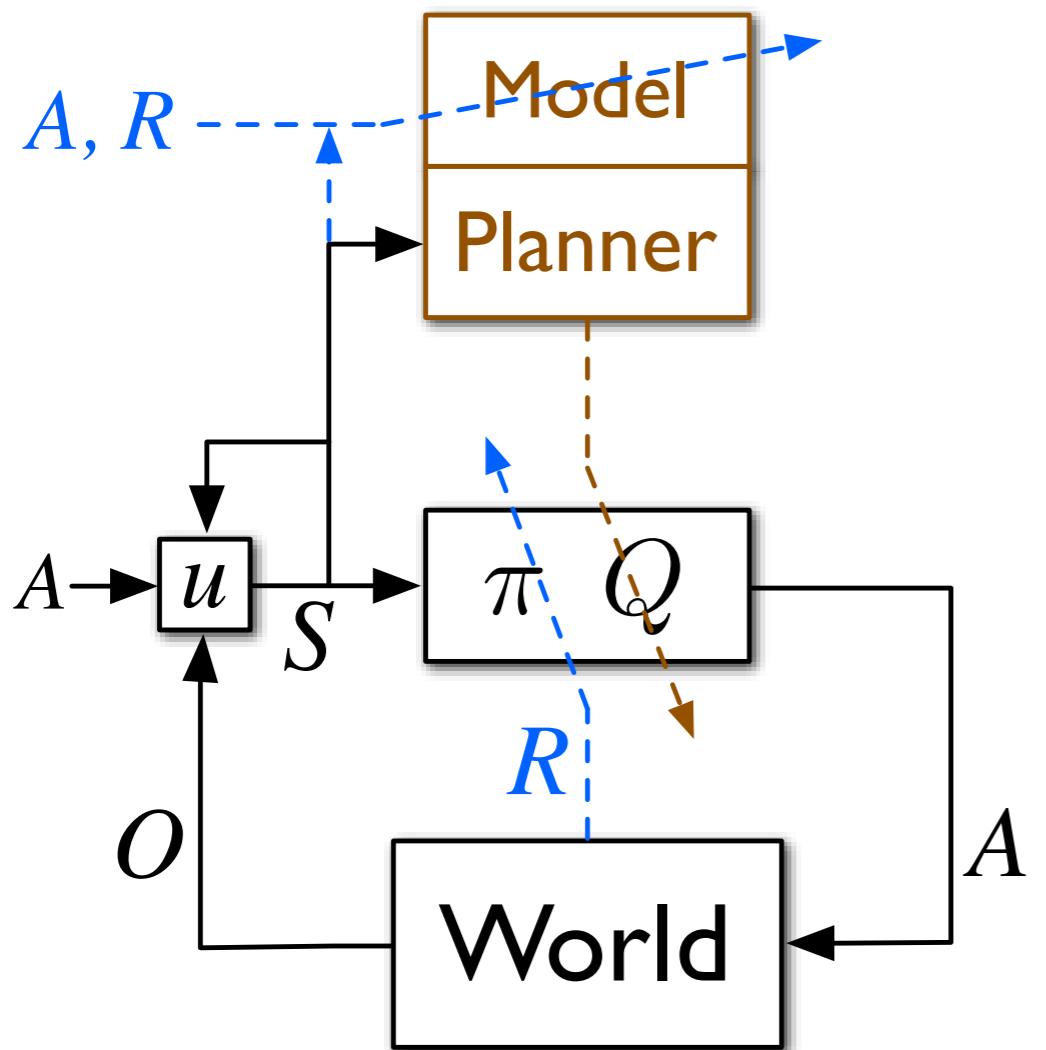


Planning is in the background

Foreground-background architecture with *partial observability*

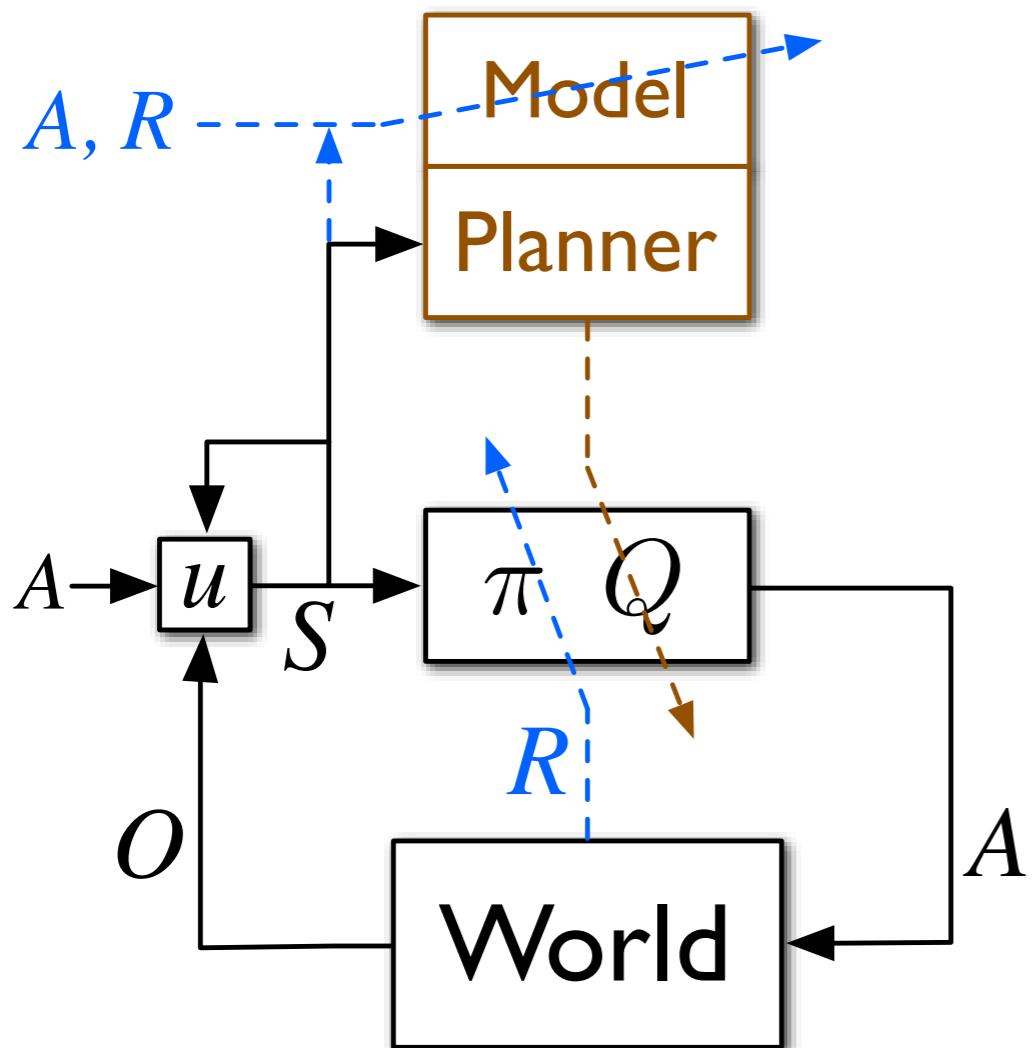


Agent state and its update



- Agent state is whatever the agent uses as state
 - in policy, value fn, model...
 - may differ from env state and information state
- State update:
$$S_{t+1} = u(S_t, A_t, O_{t+1})$$
 - e.g., Bayes rule, k-order Markov (history), PSRs, predictions

Planning should be state-to-state



$$S_{t+1} = u(S_t, A_t, O_{t+1})$$

- State update is in the foreground!
 - Planner and model see *only* states, never observations
 - We lost this with POMDPs; Why?
 - Classical and MDP planning were always state-to-state
 - Planning can always be state-to-state in information state
 - Function approximation makes planning in the info state a natural, flexible, and scalable approach

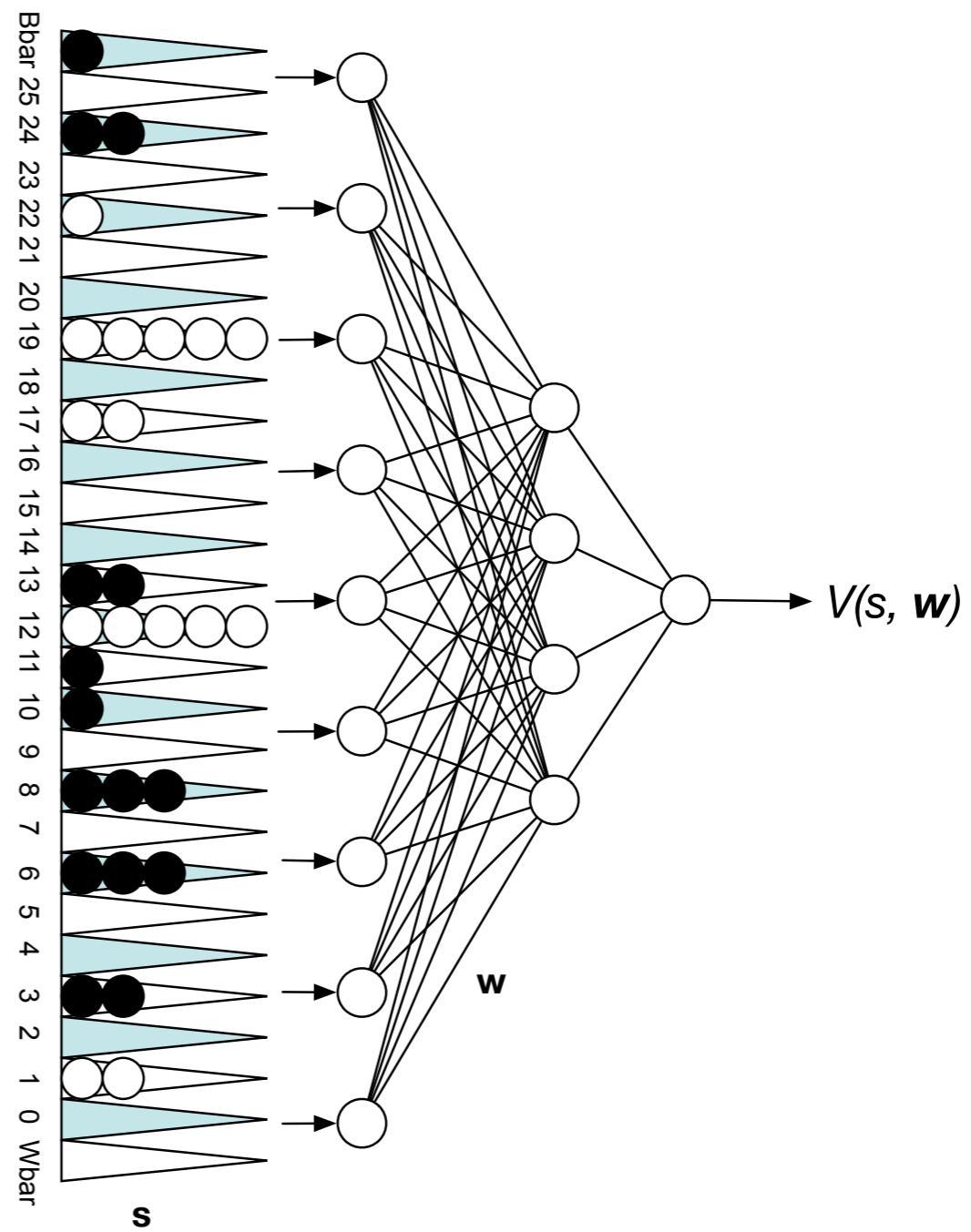
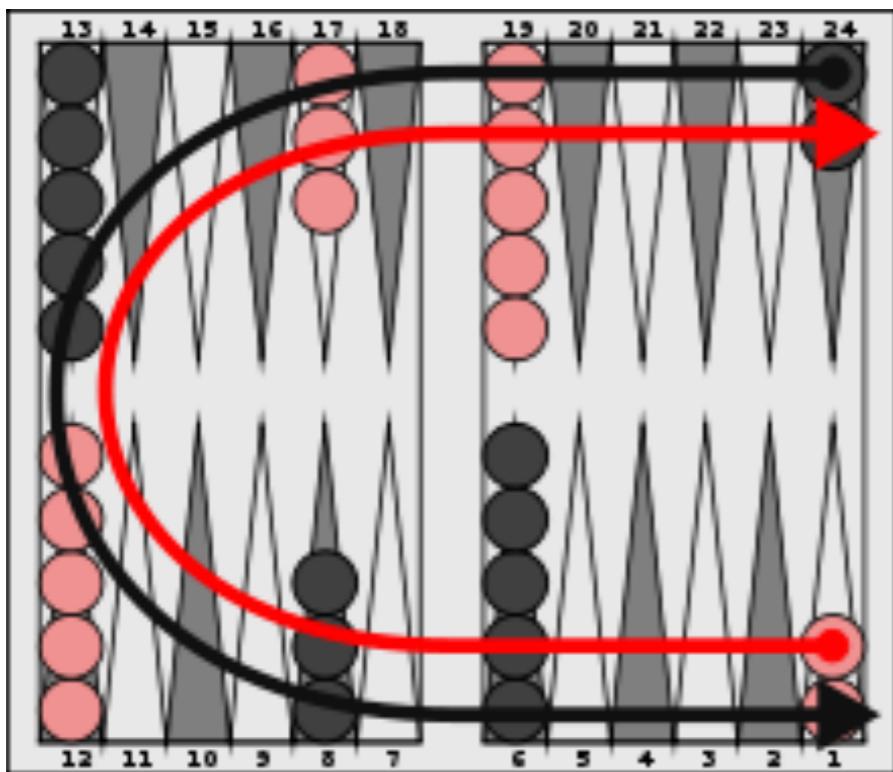
Goals for today

- Learn that policies can be optimized directly, without learning value functions, by *policy-gradient methods*
 - Glimpse a new goal for learning that does not involve discounting
 - Glimpse how one could learn real-valued (continuous) actions
- **Glimpse how to handle hidden state**

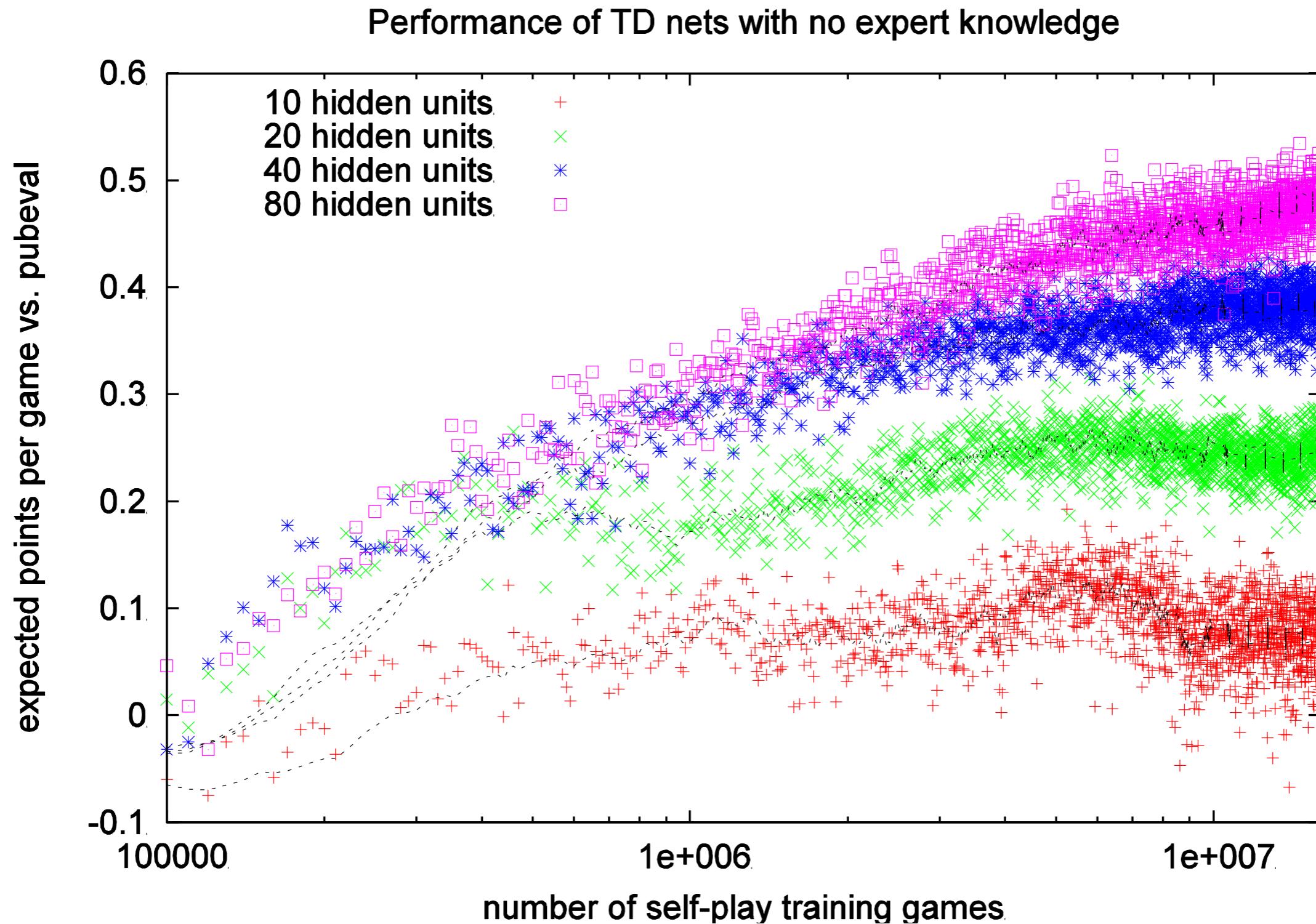
Deep Learning
= Neural Networks

Deep RL = RL + NN

Example: TD Gammon



New TD-Gammon Results



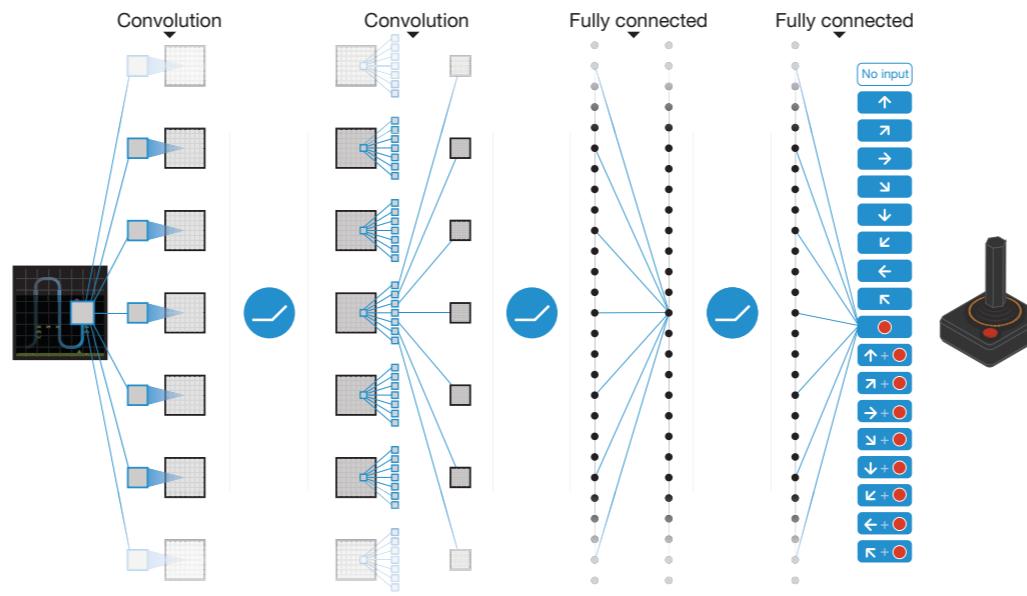
RL + DL applied to Classic Atari Games

Google Deepmind 2015, Bowling et al. 2012



- Learned to play 49 games for the Atari 2600 game console, without labels or human input, from self-play and the score alone

mapping raw screen pixels



to predictions of final score for each of 18 joystick actions

- Learned to play better than all previous algorithms and at human level for more than half the games

Same learning algorithm applied to all 49 games!
No human tuning

RL + DL Performance on Atari Games

(Deep Learning)



Space Invaders

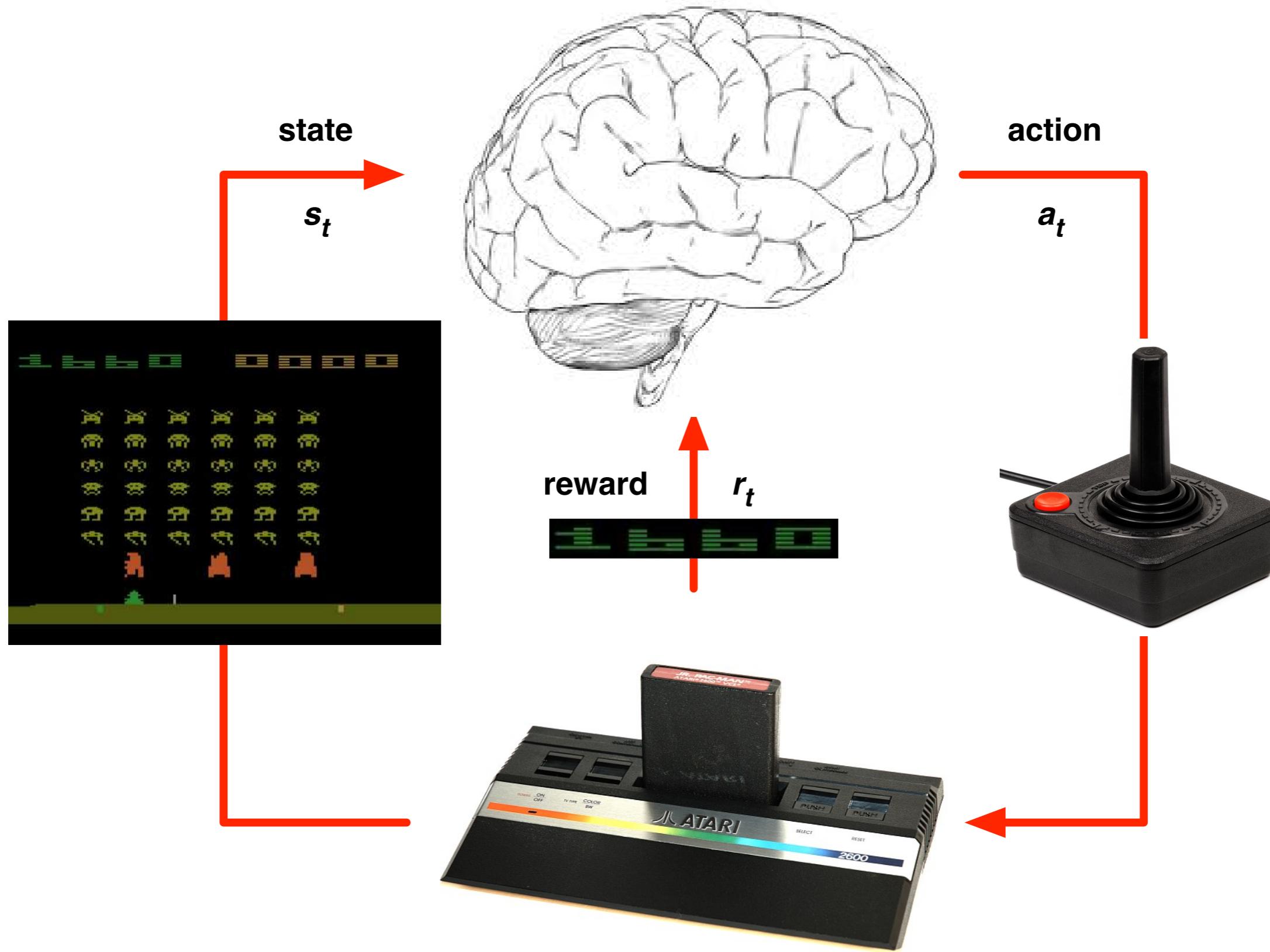


Breakout



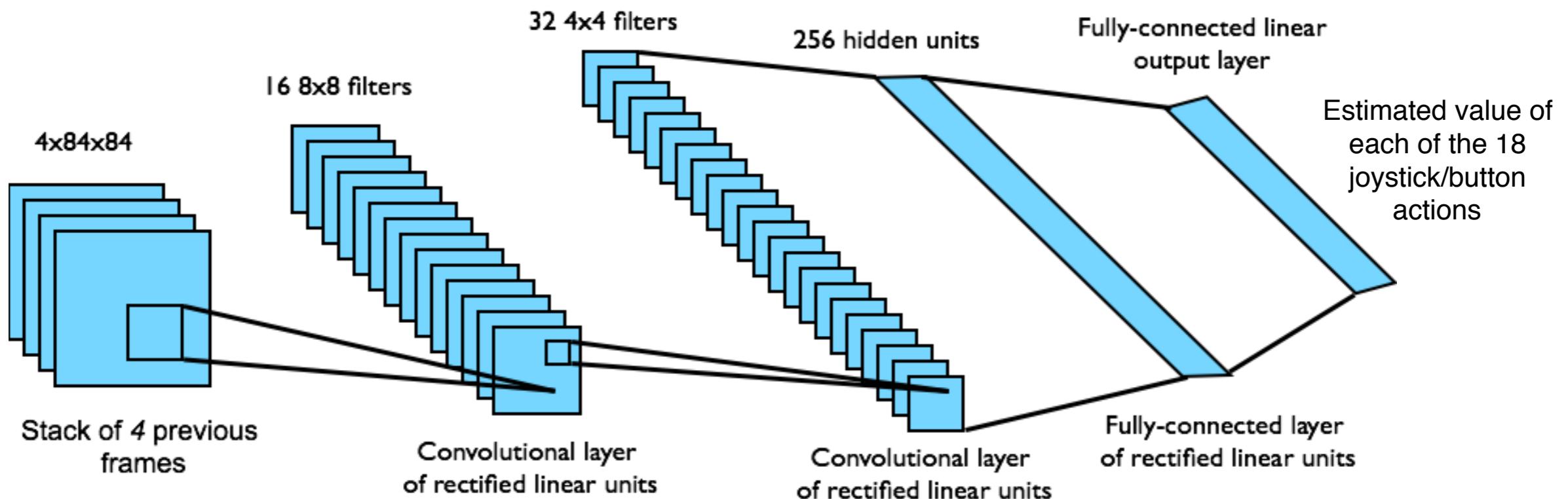
Enduro

Reinforcement Learning in Atari



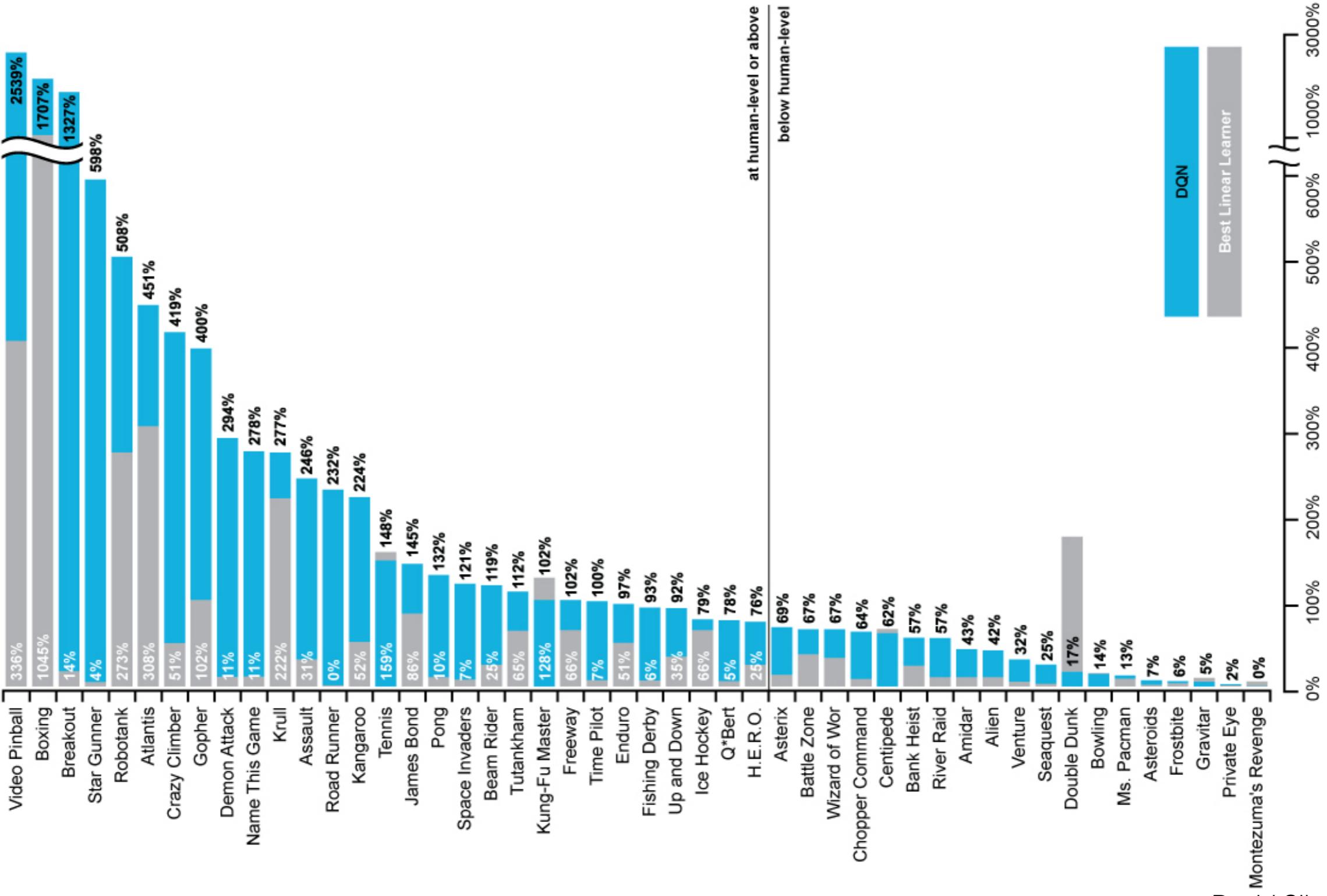
DQN in Atari

- ▶ End-to-end learning of values $Q(s, a)$ from pixels s
- ▶ Input state s is stack of raw pixels from last 4 frames
- ▶ Output is $Q(s, a)$ for 18 joystick/button positions
- ▶ Reward is change in score for that step



Network architecture and hyperparameters fixed across all games
[Mnih et al.]

DQN Results in Atari



Nick Bostrum on what kind of AI will be successfull



“Artificial Intelligence: Are we engineering our own obsolescence?”
https://www.youtube.com/watch?v=u0R6pV_aeLc