Lecture 5: Computational Cognitive Modeling

Reinforcement Learning (pt. 2)

course website:
https://brendenlake.github.io/CCM-site/

Reinforcement Learning

Three levels of description *(David Marr, 1982)*

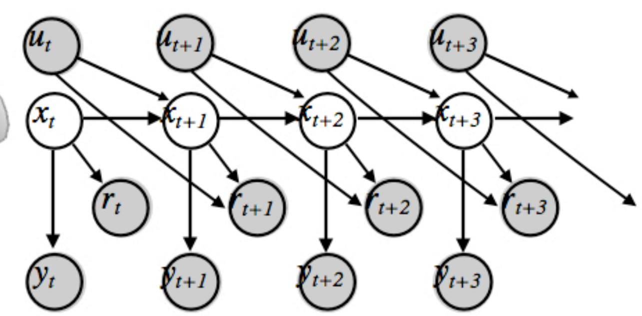
Computational
Why do things work the way they do?
What is the goal of the computation?
What are the unifying principles?

maximize:

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$

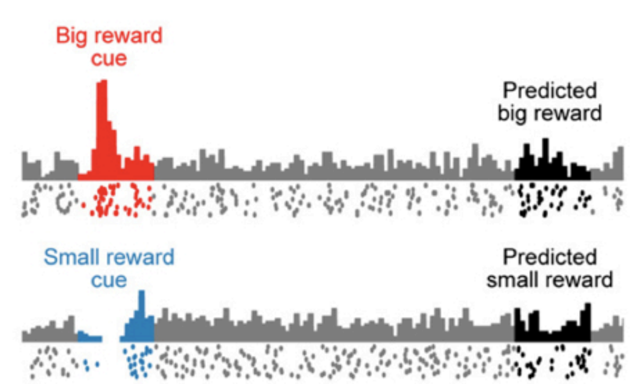
Bellman

Algorthmic
What representations can implement such computations?
How does the choice of representations determine the algorithm?



Dynamic programming, TD methods, Monte Carlo

Implementational
How can such a system be built in hardware?
How can neurons carry out the computations?

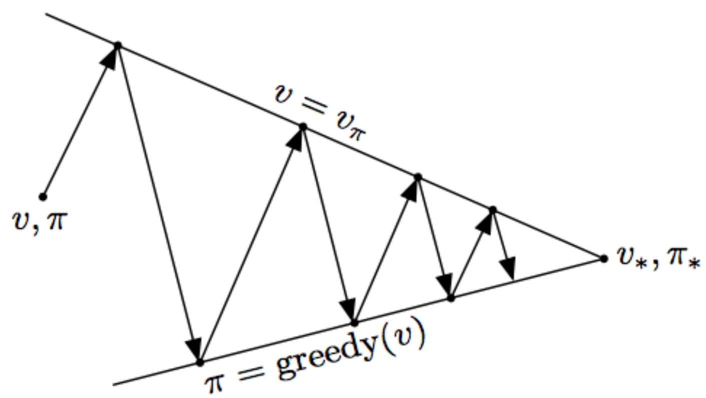


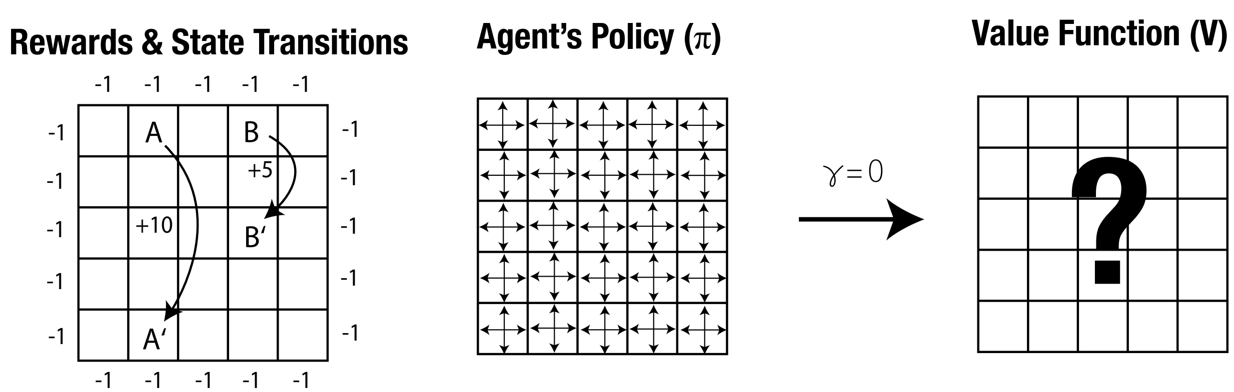
Neural firing patterns, prediction errors, system level neuroscience

Overview for Today

- Temporal difference methods
- The explore-exploit dilemma
- Generalization and function approximation

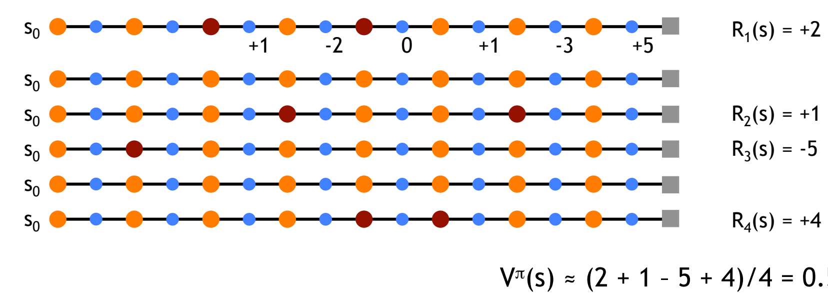
Dynamic Programming/Value iteration





- Generally require "model" of environment (i.e., knowledge of state transitions, reward, and policy at at point in environment)
- Curse of dimensionality
- Proveably converges to optimal
- Solution benefits from "bootstrapping"

Monte Carlo



- Does **not** require "model"
- May not even estimate some part of environment
- Convergence more sensitive to issues like sufficient exploration
- Solution does not benefit from "bootstrapping"

Blending the ideas….

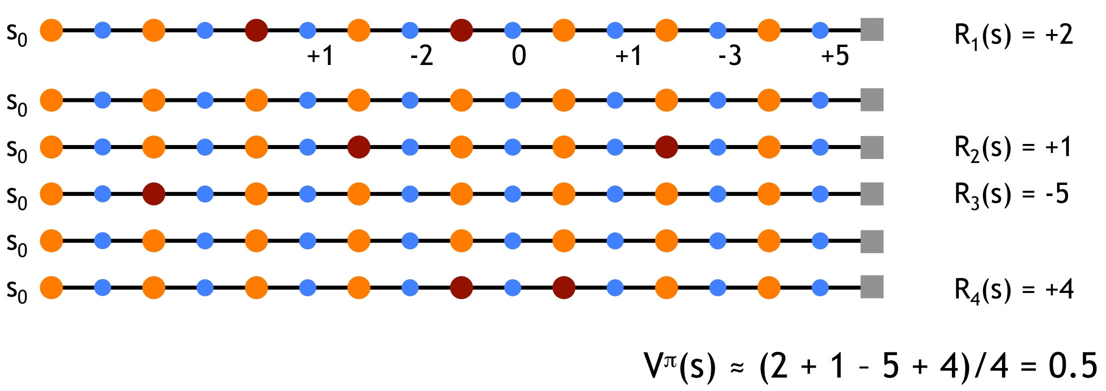
- The first-visit MC algorithm has following steps

Let R be the return following first visit to state s. Append R to list Returns[s]. V(s) = average(Returns[s])

- Incremental implementation:

$$V(s) = V(s) + \frac{1}{n(s)}[R - V(s)]$$

where n(s) is number of first visits to s.

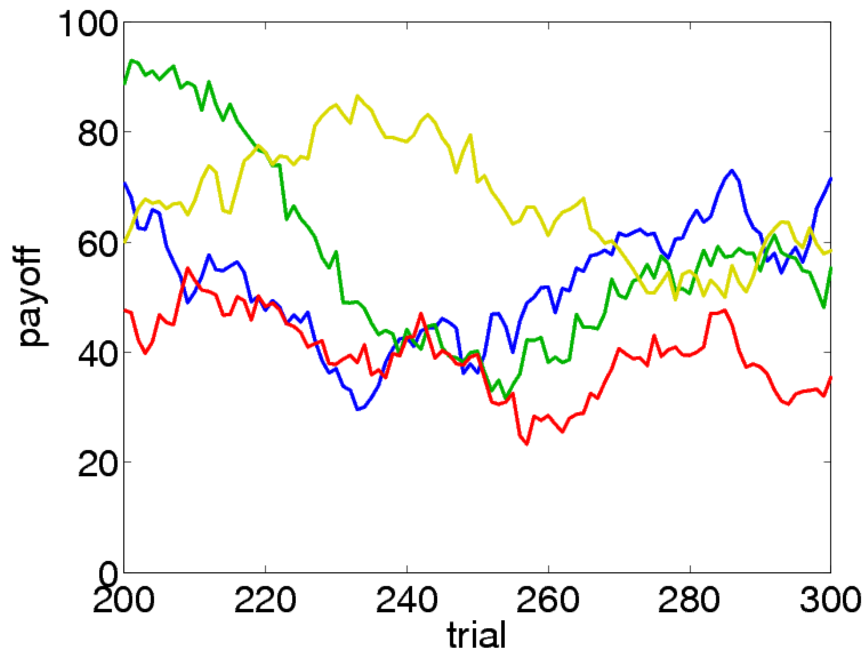


Blending the ideas….

Now consider a constant step size Monte-carlo update:

$$V(s) = V(s) + \alpha[R - V(s)]$$

Why might this be useful?



(hint)

Temporal difference prediction

Policy evaluation is often referred to as a prediction problem: we are trying to predict how much return we'll get from being in state s and following our policy.

Monte carlo incremental update

$$V(s) = V(s) + \alpha[R - V(s)]$$

target: *actual* return from s_t to end of episode

Still have to wait until episode terminates…

Temporal Difference update TD(0):

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

target: *estimate* of the return… using BOOTSTRAPPING!

Evaluating the world when you don't know anything about it

$\alpha = 0.9 \quad \gamma = 1 \quad \pi$ - random

Initialize

d	e	f
a	b	c

$\Rightarrow$

0	0	0
0	0	0

$c \rightarrow f \qquad V(c) \leftarrow 0 + 0.9[100 + 0 - 0] = 90$

0	0	0
0	0	0

$\Rightarrow$

0	0	0
0	0	90

Evaluating the world when you don't know anything about it

$a \rightarrow b \quad V(a) \leftarrow 0 + 0.9[0 + 0 - 0] = 0$

0	0	0
0	0	90

0	0	0
0	0	90

$e \rightarrow f \quad V(e) \leftarrow 0 + 0.9[100 + 0 - 0] = 90$

0	0	0
0	0	90

0	90	0
0	0	90

Evaluating the world when you don't know anything about it

$b \rightarrow c \quad V(b) \leftarrow 0 + 0.9[0 + 90 - 0] = 81$

0	90	0
0	0	90

0	90	0
0	81	90

$d \rightarrow e \quad V(d) \leftarrow 0 + 0.9[0 + 90 - 0] = 81$

0	90	0
0	81	90

81	90	0
0	81	90

Evaluating the world when you don't know anything about it

$$a \rightarrow b \quad V(a) \leftarrow 0 + 0.9[0 + 81 - 0] \approx 73$$

81	90	0
0	81	90

81	90	0
73	81	90

$$c \rightarrow f \quad V(c) \leftarrow 90 + 0.9[100 + 0 - 90] = 99$$

81	90	0
73	81	90

81	90	0
73	81	99

Evaluating the world when you don't know anything about it

$$e \rightarrow f \quad V(e) \leftarrow 90 + 0.9[100 + 0 - 90] = 99$$

81	90	0
73	81	99

81	99	0
73	81	99

$$c \rightarrow b \quad V(c) \leftarrow 99 + 0.9[0 + 81 - 99] \approx 83$$

81	99	0
73	81	99

81	99	0
73	81	83

Evaluating the world when you don't know anything about it

$\gamma = 0.9$

52	66	0
49	57	76

bellman solution!

Temporal difference prediction

Temporal Difference update TD(0):

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

Bellman recurrence relation

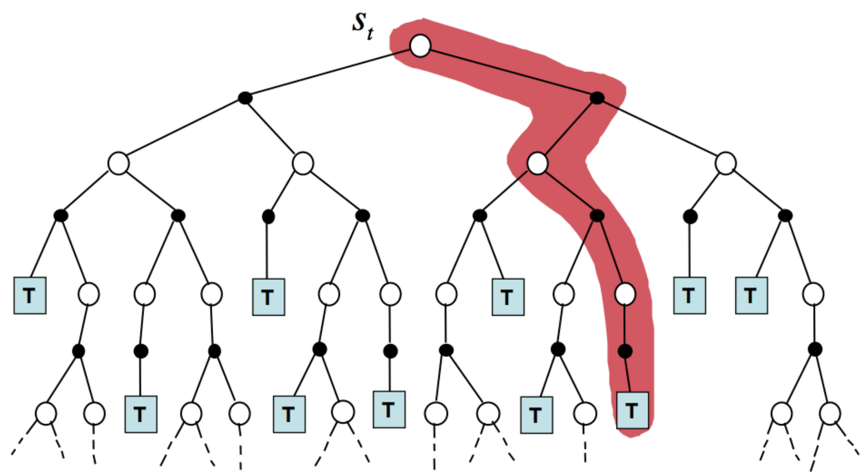
$$V^{\pi}(s) = E_{\pi}\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s\}$$

$$V^{\pi}(s) = \sum_{a} \pi(s, a) \sum_{s'} \mathcal{P}^{a}_{ss'} [\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s')]$$

Simple Monte Carlo

$$V(s_t) \leftarrow V(s_t) + \alpha\left[R_t - V(s_t)\right]$$

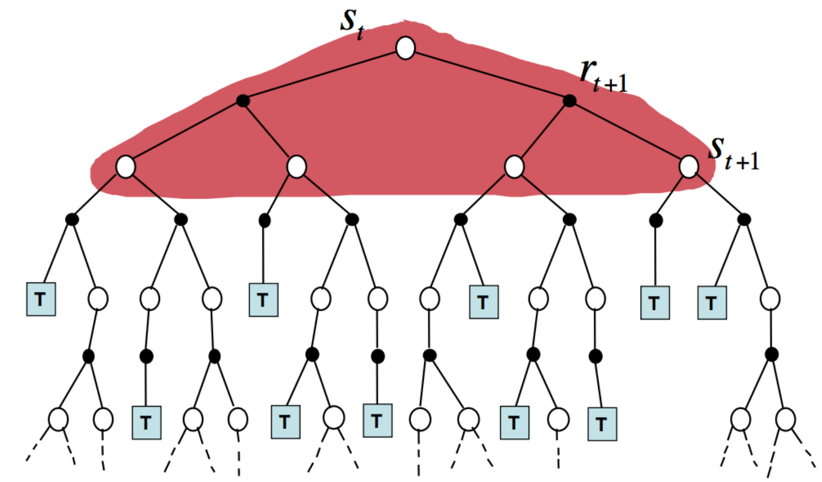
where R_t is the actual return following state s_t.



Monte Carlo uses an estimate of the actual return.

Dynamic Programming

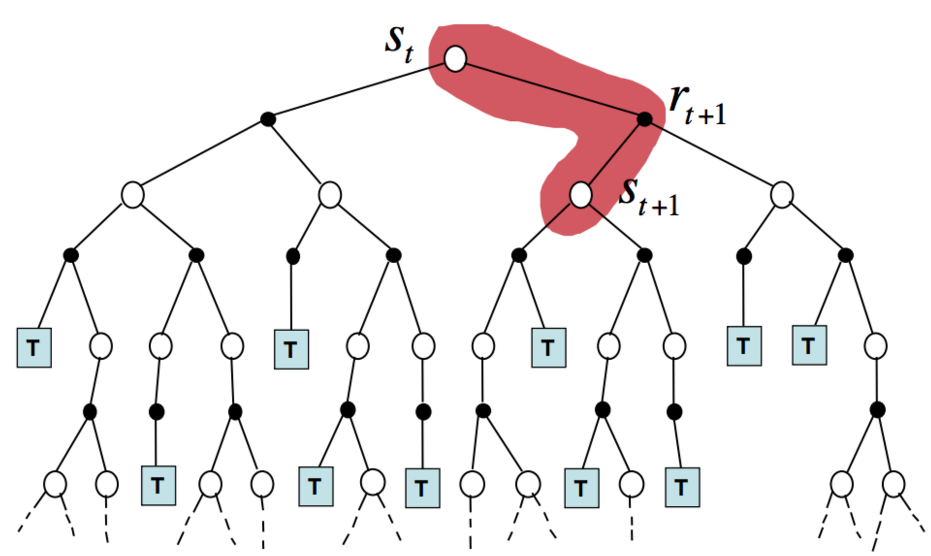
$$V(s_t) \leftarrow E_\pi\left\{r_{t+1} + \gamma V(s_t)\right\}$$



The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because V^π is not known and the current estimate is used instead.

Simplest TD Method

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$



TD samples the expected value and uses the current estimate of the value.

Advantages of TD learning methods

- Don’t need a model of the environment
- Online and incremental so can be fast (don’t need to wait until end of episode as in MC)
- Update based on actual experience (r_{t+1})
- Converges to the true values if you lower step size/learning rate as learning continues
- TD bootstraps: it updates estimate based on other estimates (like DP/value iteration).
- TD samples: updates are based on a single run/path through the state space (like MC)

TD(0) is still kind of slow: Eligibility traces

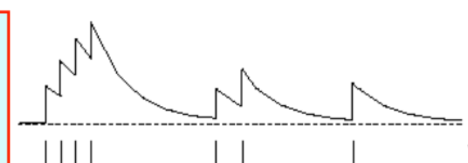
- The benefits of bootstrapping only extend between adjacent states (s to s'). As a result you have to cross that particular state transition many times for the value to "propogate" backwards

- New variable called *eligibility trace*. The eligibility trace for state at time is denoted

$$e_t(s) \in \Re^+$$

On each step, decay all traces by $\gamma\lambda$ and increment the trace for the current state by

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$



γ discount rate

λ trace-decay parameter

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$$

$$\Delta V_t(s) = \alpha \delta_t e_t(s)$$

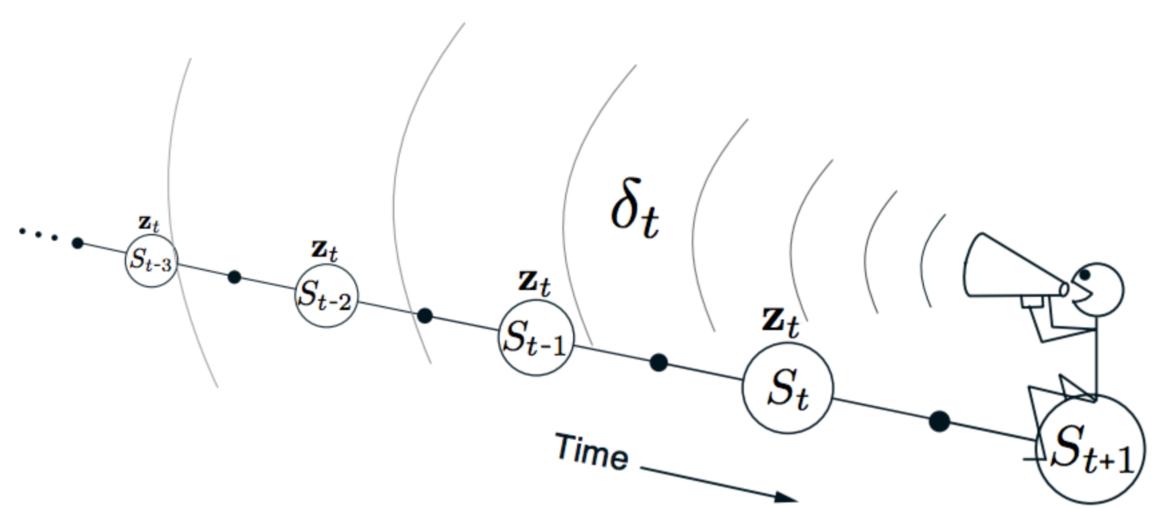
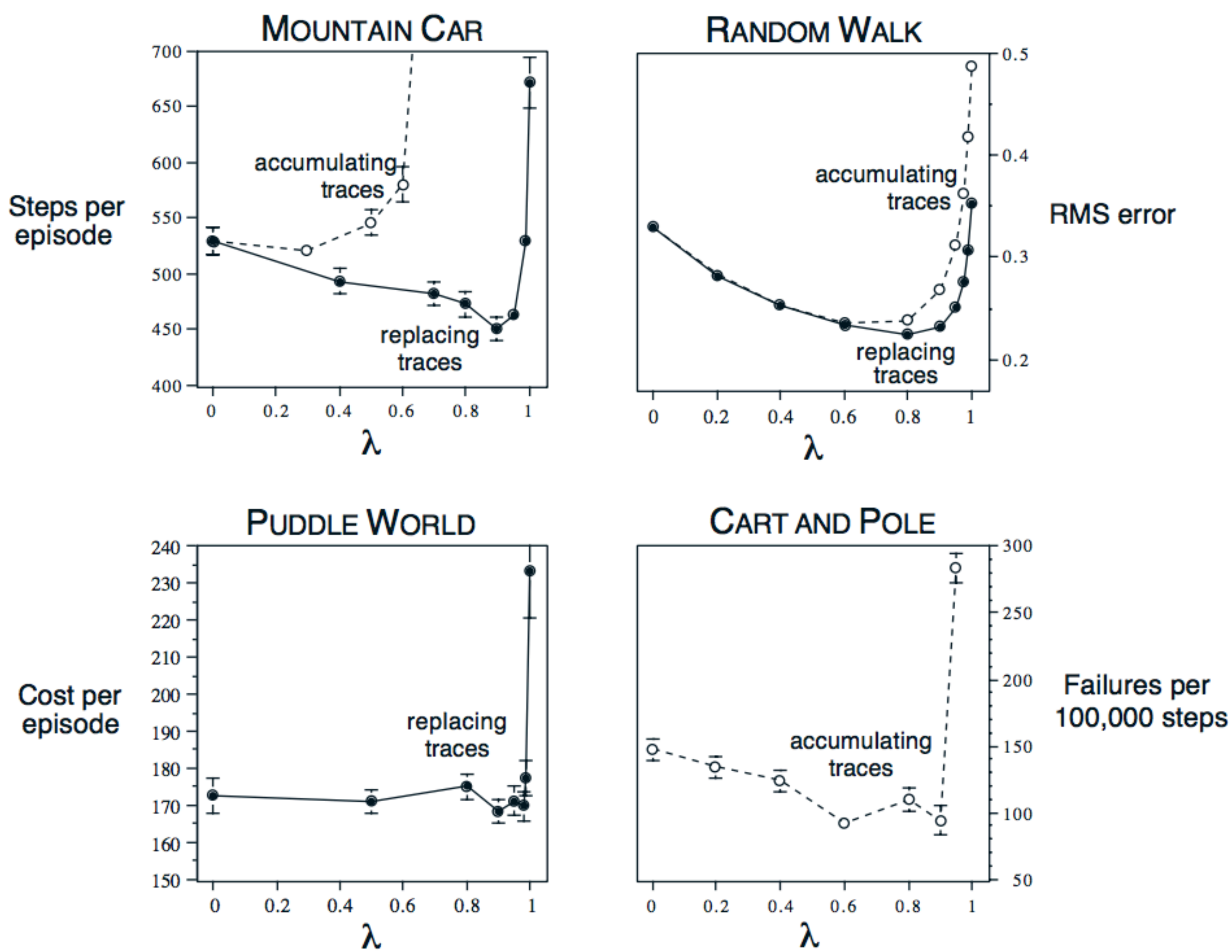


Figure 12.5: The backward or mechanistic view. Each update depends on the current TD error combined with the current eligibility traces of past events.

TD(0) is still kind of slow: Eligibility traces



intermediate values empirically work best!

Learning for control: Learning Q-values

- Learning the value of different states can be a little obtuse because what you really want to do is learn how to act!
- Instead can make sense to learn $Q^{\pi}(s, a)$

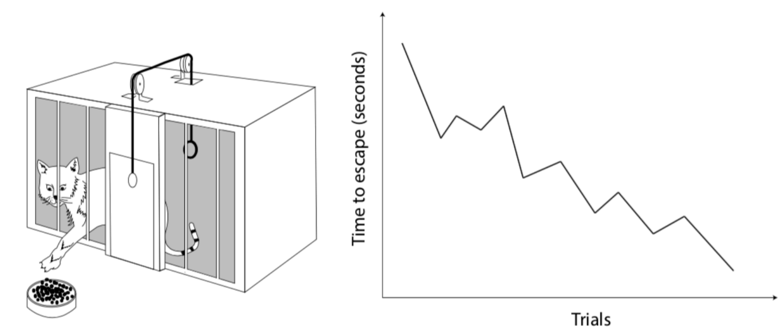


Figure 1: *Left*: An illustration of Thorndike's puzzle box experiments. *Right*: The time recorded to escape the box is reduced over repeated trials as the cat becomes more efficient at selecting the actions which lead to escape.

SARSA update rule:

$$\Delta Q_t(s_t, a_t) = \alpha[r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$$

- Choose a policy and estimate the Q-values using SARSA rule. Change policy toward greediness with respect to Q values.
- Converges with probability 1 to optimal policy and Q-value if you visit all state-action pairs infinitely many times and the policy converges to be a greedy policy.
- Easy to know what to do! Just choose the action with highest Q value!

Learning for control: Learning Q-values

SARSA update rule:

$$\Delta Q_t(s_t, a_t) = \alpha[r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$$

- Initialise $Q(s, a)$
- Repeat `many times`
  - Pick s, a
  - Repeat `each step to goal`
    - Do a, observe r, s'
    - Choose a' based on $Q(s', a')$ ϵ-`greedy`
    - $Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s', a') - Q(s,a)]$
    - $s = s'$, $a = a'$
  - Until s terminal (where $Q(s', a') = 0$)

Use with policy iteration, i.e. change policy each time to be greedy wrt current estimate of Q

sarsa is known as an
on-policy learning rule…

Learning for control: Learning Q-values

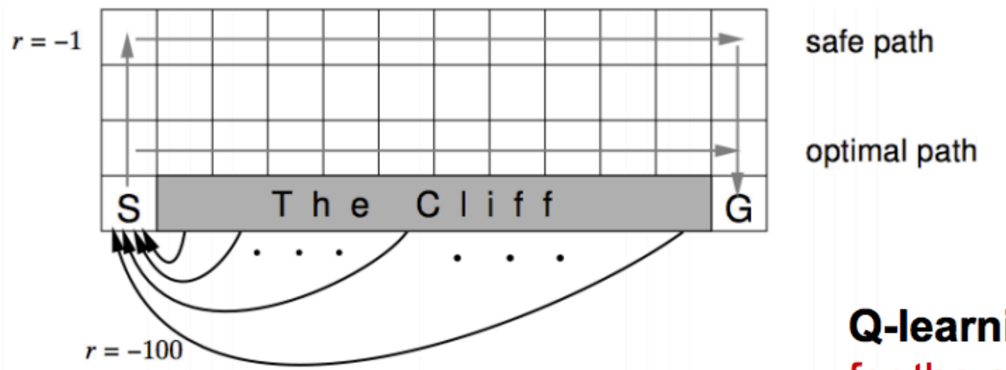
Q-learning update rule:

$$\Delta Q_t(s_t, a_t) = \alpha[r_{t+1} + \gamma max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)]$$

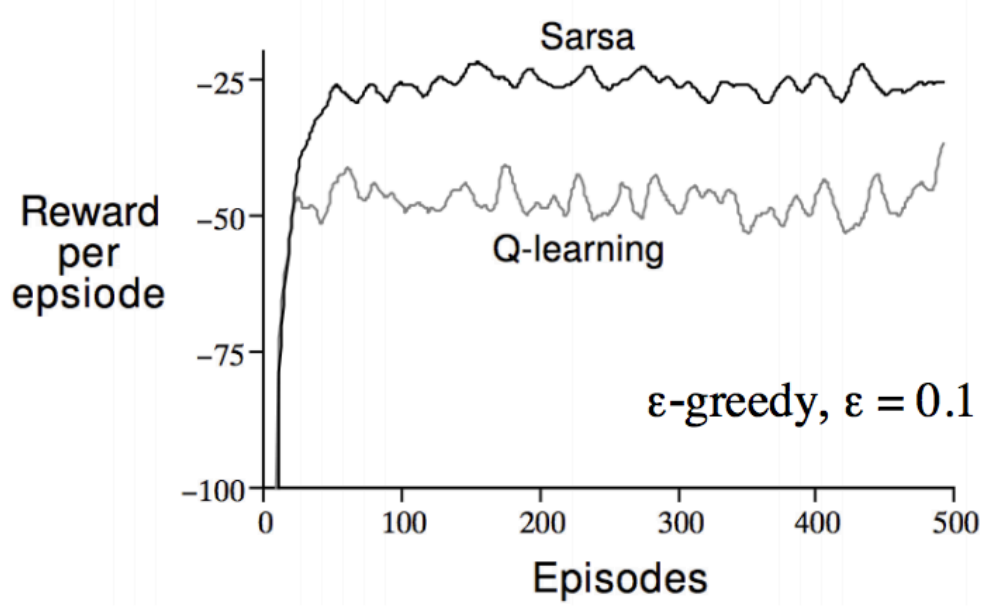
- Initialise $Q(s, a)$
- Repeat `many times`
  - Pick s `start state`
  - Repeat `each step to goal`
    * Choose a based on $Q(s, a)$ ϵ`-greedy`
    * Do a, observe r, s'
    * $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    * $s = s'$
  - Until s terminal

Q-learning is known as an
off-policy learning rule…
always update Q value with maximally
best action in next state, even if you
won't necessarily take that step
yourself.

Q-learning versus SARSA (Cliffwalking)



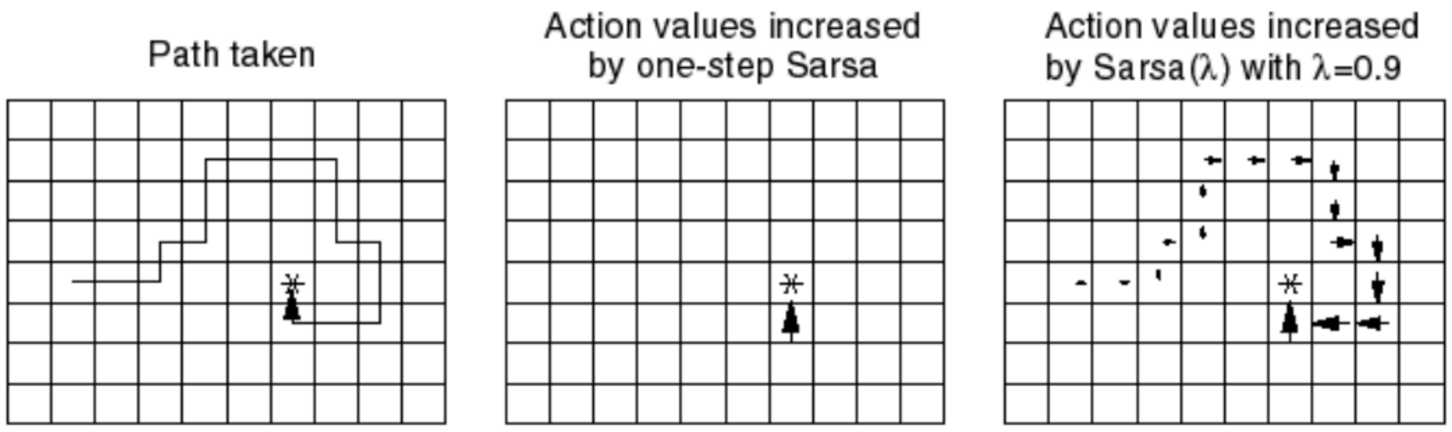
Reward is on all transitions -1 except those into the the region marked "The Cliff."



Q-learning learns quickly values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ε-greedy action selection.
Sarsa takes the action selection into account and learns the longer but safer path through the upper part of the grid.

If ε were gradually reduced, then both methods would asymptotically converge to the optimal policy.

SARSA(lambda)



- 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

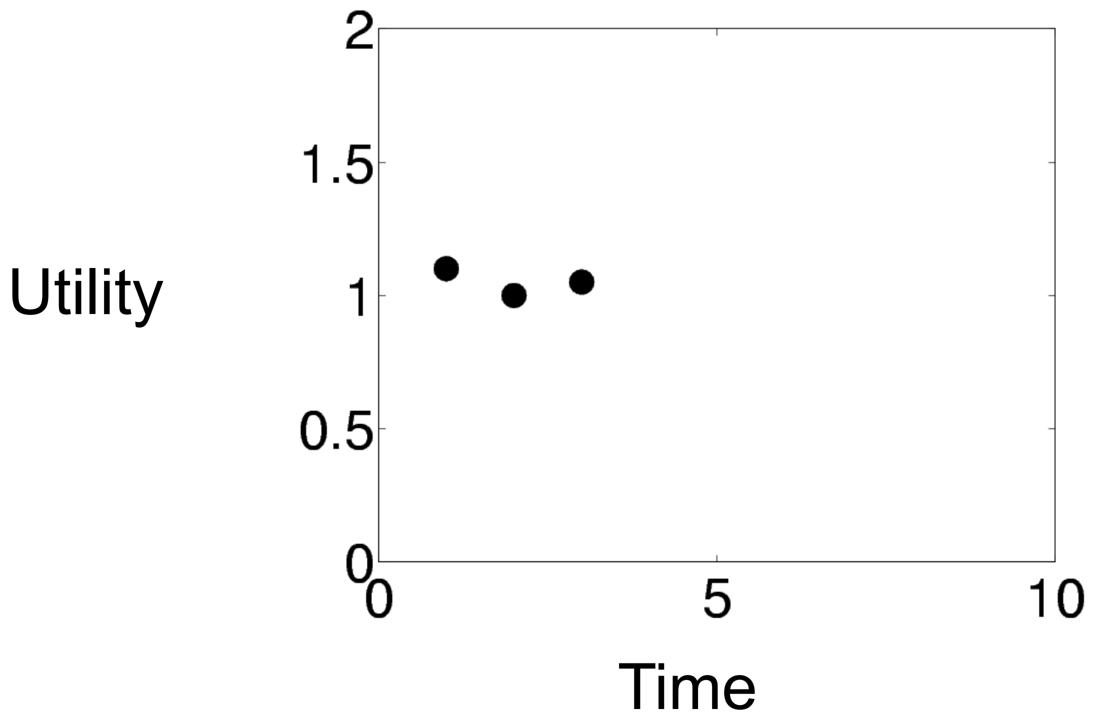
The Explore-Exploit Dilemma

TD methods require a bit of randomness in order to properly search the state space (we call this search process **exploration**). Reward maximization requires choosing what seems like the best action (**exploitation**). Effective learning in unknown environment requires proper balance of these tensions.

Classic dilemma in learned decision making

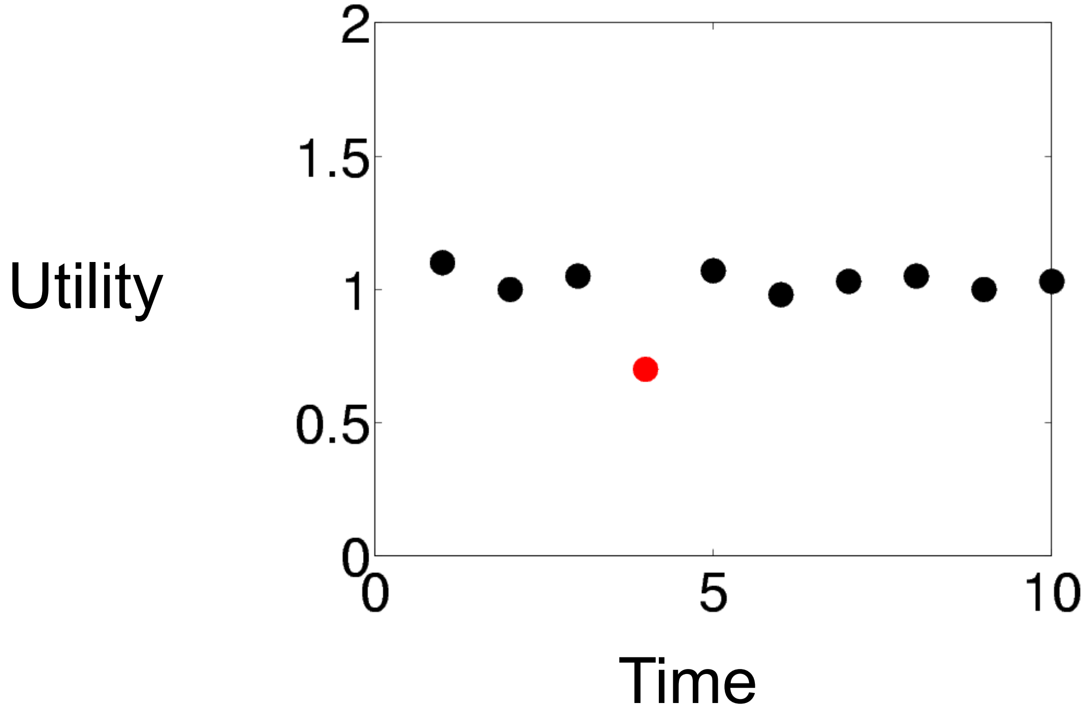
For unfamiliar outcomes, how to trade of learning about their quality/ value against exploiting knowledge already gained.

Exploration vs. exploitation



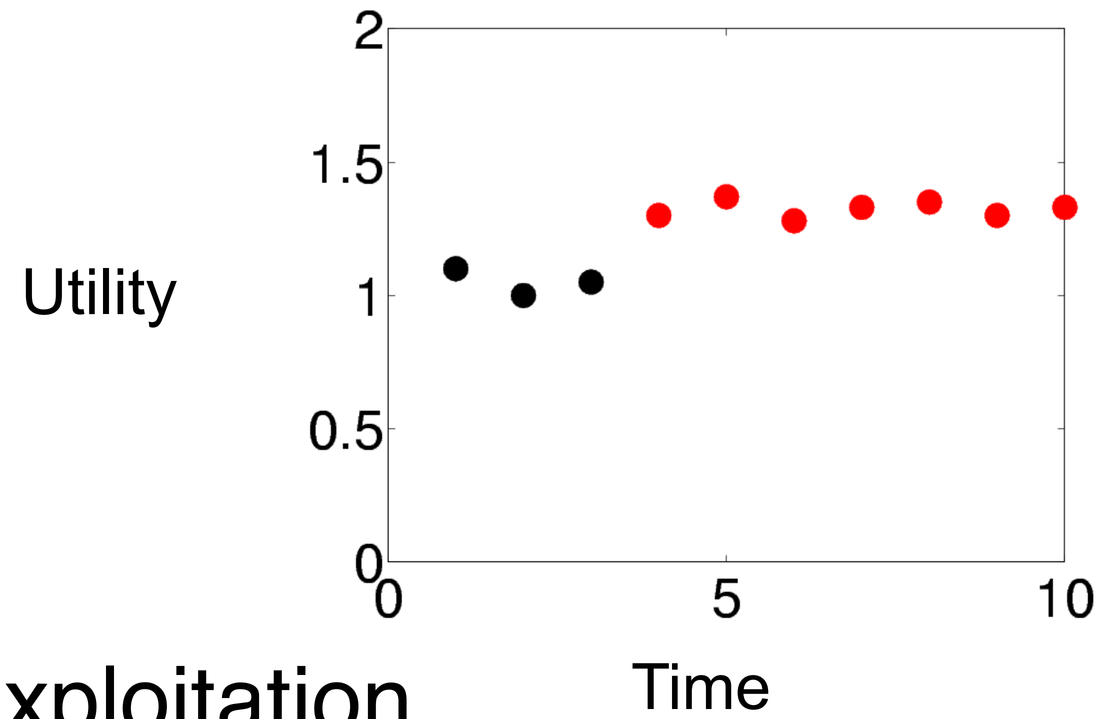
- Exploitation
  - Choose action expected to be best
  - May never discover something better

Exploration vs. exploitation



- Exploitation
  - Choose action expected to be best
  - May never discover something better
- Exploration:
  - Choose action expected to be worse

Exploration vs. exploitation



- Exploitation
  - Choose action expected to be best
  - May never discover something better
- Exploration:
  - Choose action expected to be worse
  - Balanced by the long-term gain if it turns out better

the N-armed bandit

another name for a popular psychology/neuroscience task:

- repeated choice between lotteries…
- ...whose properties are learned experientially
- (assume each bandit is just a weighted coin: no weird time-based lotteries)

overall approach:

1. learn Q-values for options
2. choose the best ??

1.Greedy methods (e.g., epsilon greedy)

2.Softmax

3.Optimal exploration

Action Selection

Greedy: select the action a^* for which Q is highest:

$$Q_t(a^*) = \max_a Q_t(a)$$

So $a^* = \arg\max_a Q_t(a)$ – and * means "best"

Example: 10-armed bandit

Snapshot at time t for actions 1 to 10

$Q_t(a) \rightarrow$

0	0.3	0.1	0.1	0.4	0.05	0	0	0.05	0

$Q_t(a^*) = 0.4$ and $a^* =$?

Maximises reward

ϵ-**greedy**: Select *random* action ϵ of the time, else select greedy action

Sample all actions infinitely many times

So as $k_a \to \infty$, Qs converge to Q^*

Can reduce ϵ over time

Softmax Action Selection

ϵ-greedy: even if worst action is very bad, it will still be chosen with same probability as second-best – we may not want this. So:

Vary selection probability as a function of estimated goodness

Choose a at time t from among the n actions with probability

$$\frac{\exp(Q_t(a)/\tau)}{\sum_{b=1}^{n} \exp(Q_t(b)/\tau)}$$

Gibbs/Boltzmann distribution, τ is temperature (from physics)

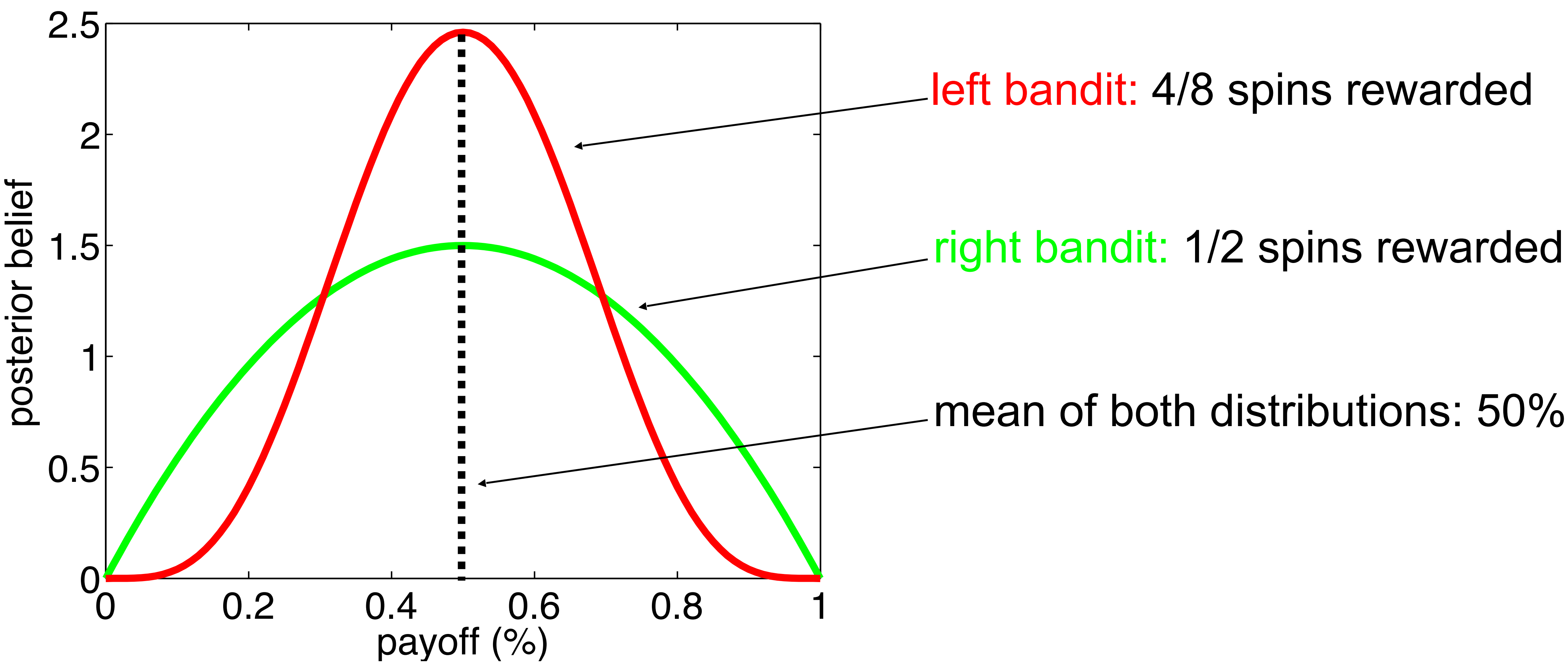
Effect of $|\ \tau\ |$

As $\tau \to \infty$, probability $\to 1/n$

As $\tau \to 0$, probability $\to$ greedy

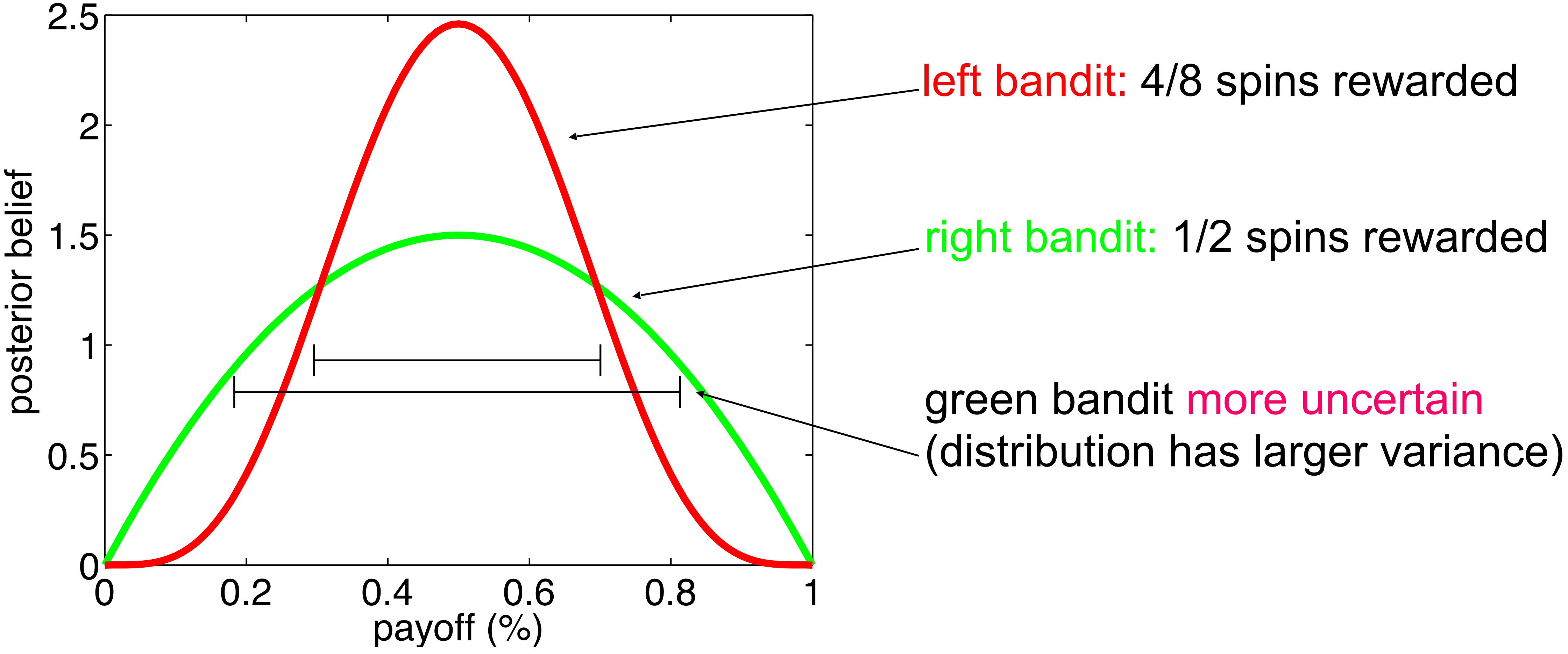
wwBd?

assign belief according to posterior probability of different chances of heads



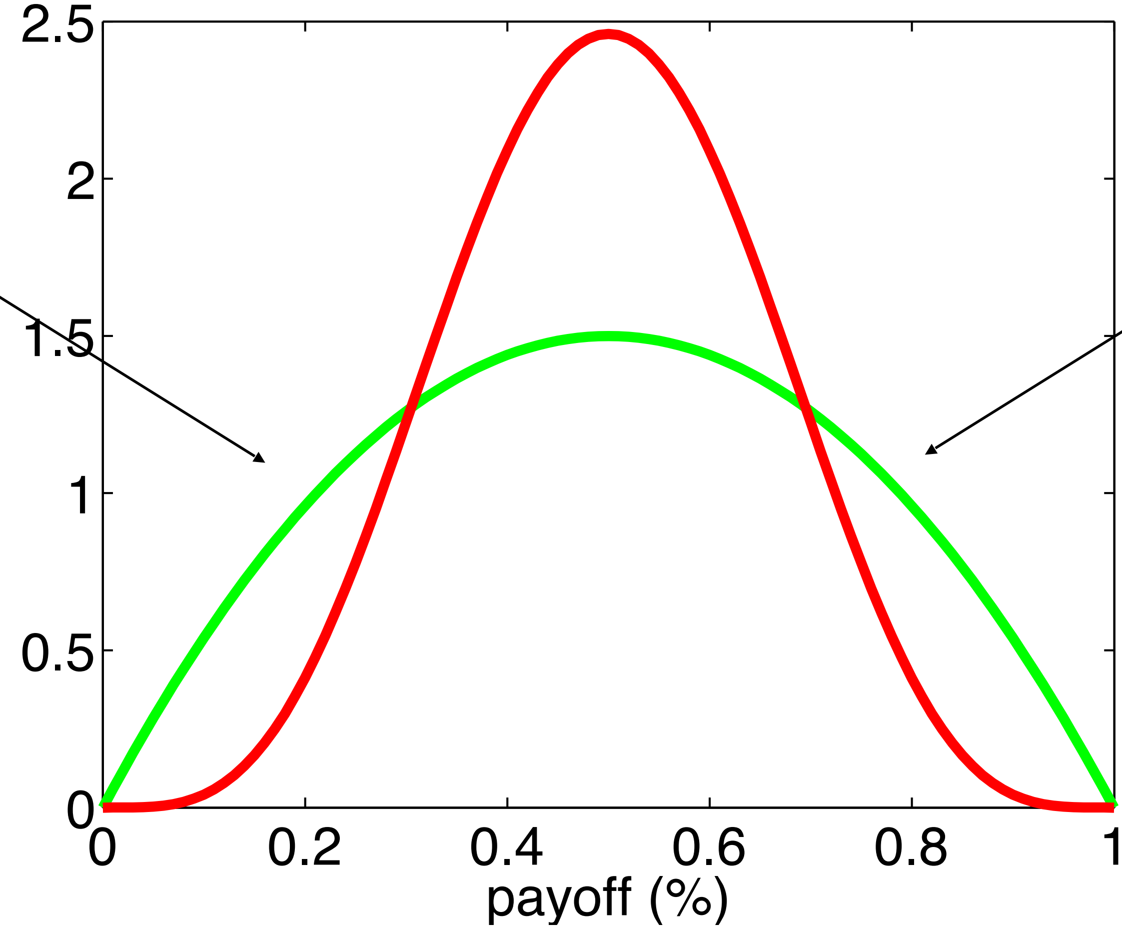
wwBd?

assign belief according to posterior probability of different chances of heads



Gittins index

although green bandit has a larger chance of being worse…



… it also has a larger chance of being better

…which would be useful to find out, if true

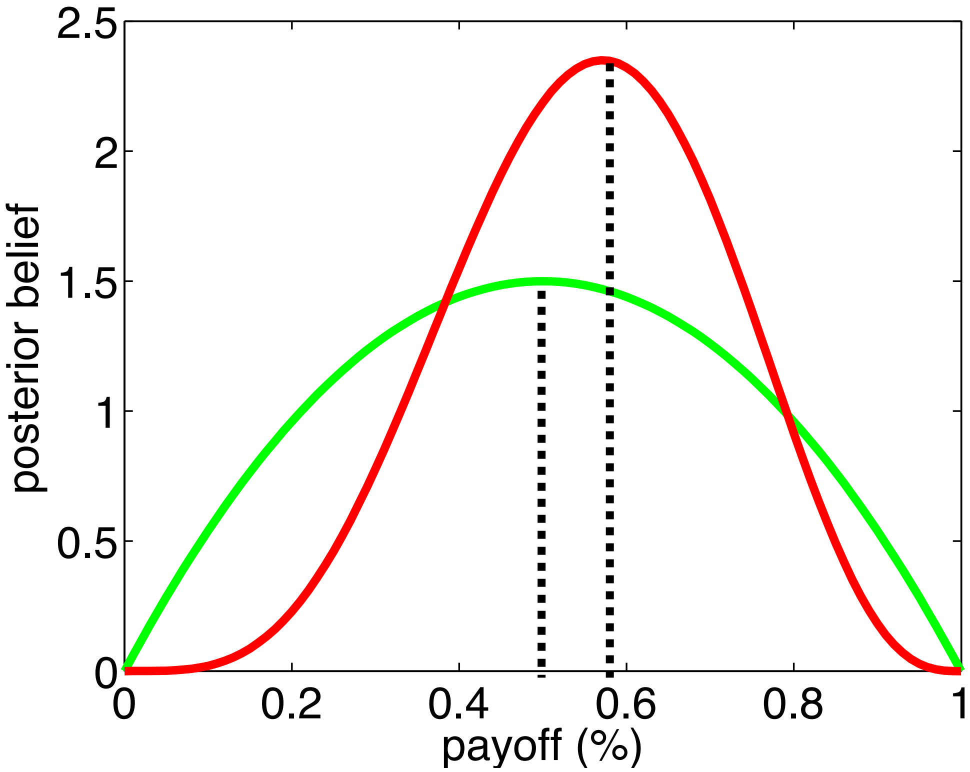
"Gittins index":
- choose on the basis of expected payoff (50%) plus "uncertainty bonus"
- quantifies "value of information" : chance of finding something better & improving my future prospects
- (very difficult to work out exactly, he solves for simple problems)

→ note that Rescorla/Wagner model doesn't track uncertainty, only mean

horizon

suppose I have so far been rewarded:

- 4 out of 7 spins on the left bandit (57%)
- 1 out of 2 spins on the right bandit (50%)



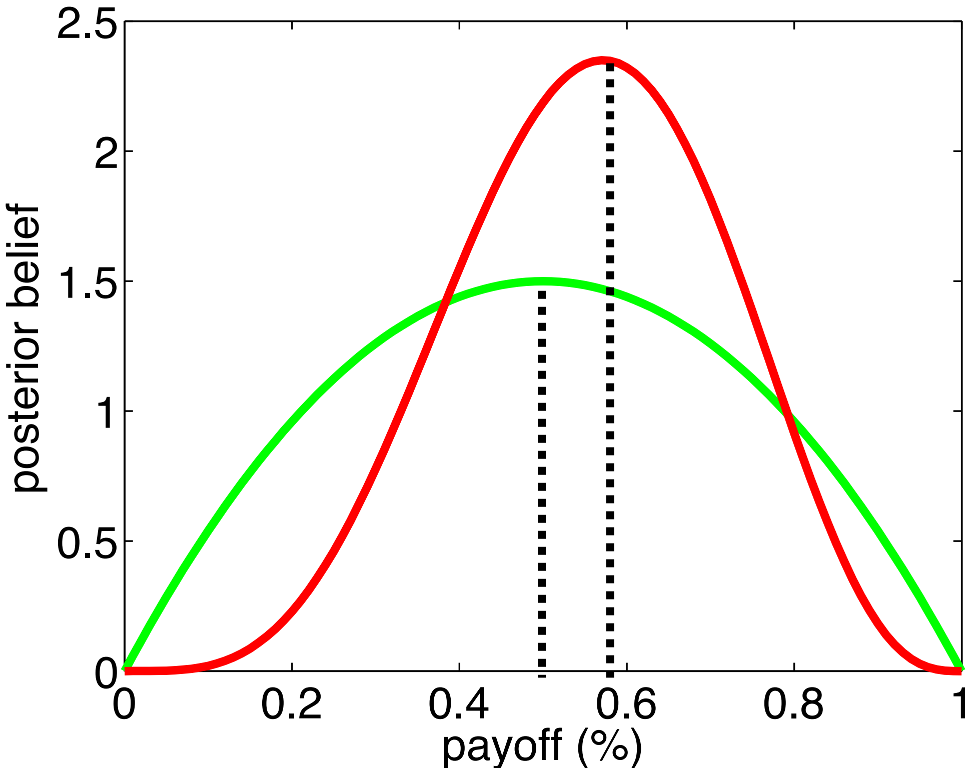
horizon

suppose I have so far been rewarded:

- 4 out of 7 spins on the left bandit (57%)
- 1 out of 2 spins on the right bandit (50%)

… and I am allowed only one more spin

now which should I choose?

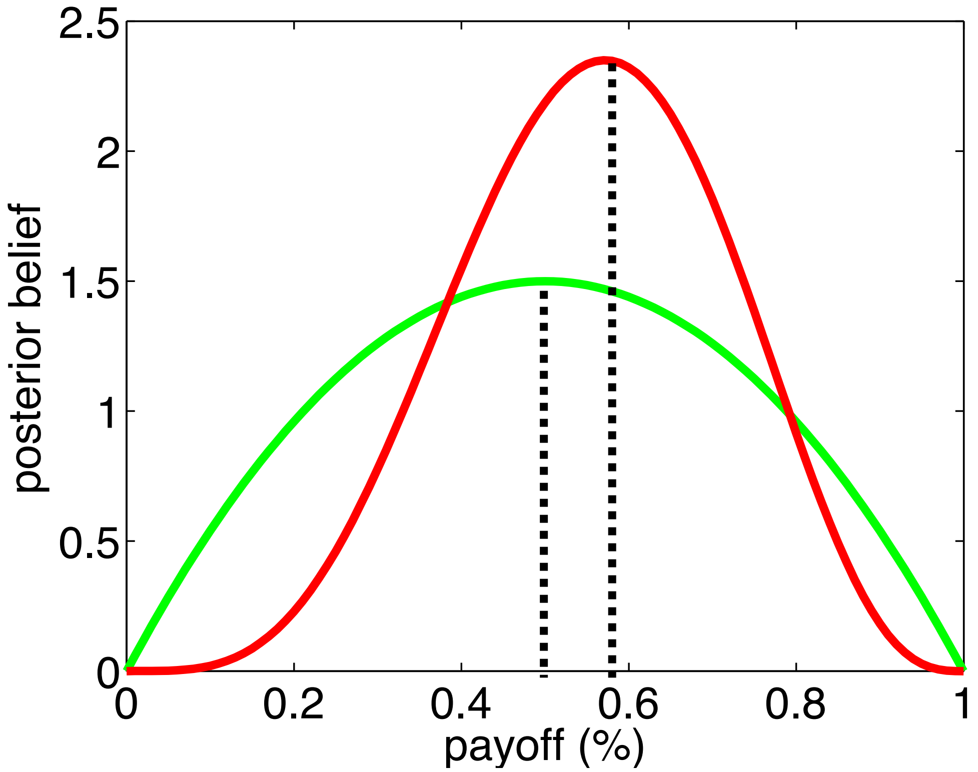


horizon

suppose I have so far been rewarded:
- 4 out of 7 spins on the left bandit (57%)
- 1 out of 2 spins on the right bandit (50%)

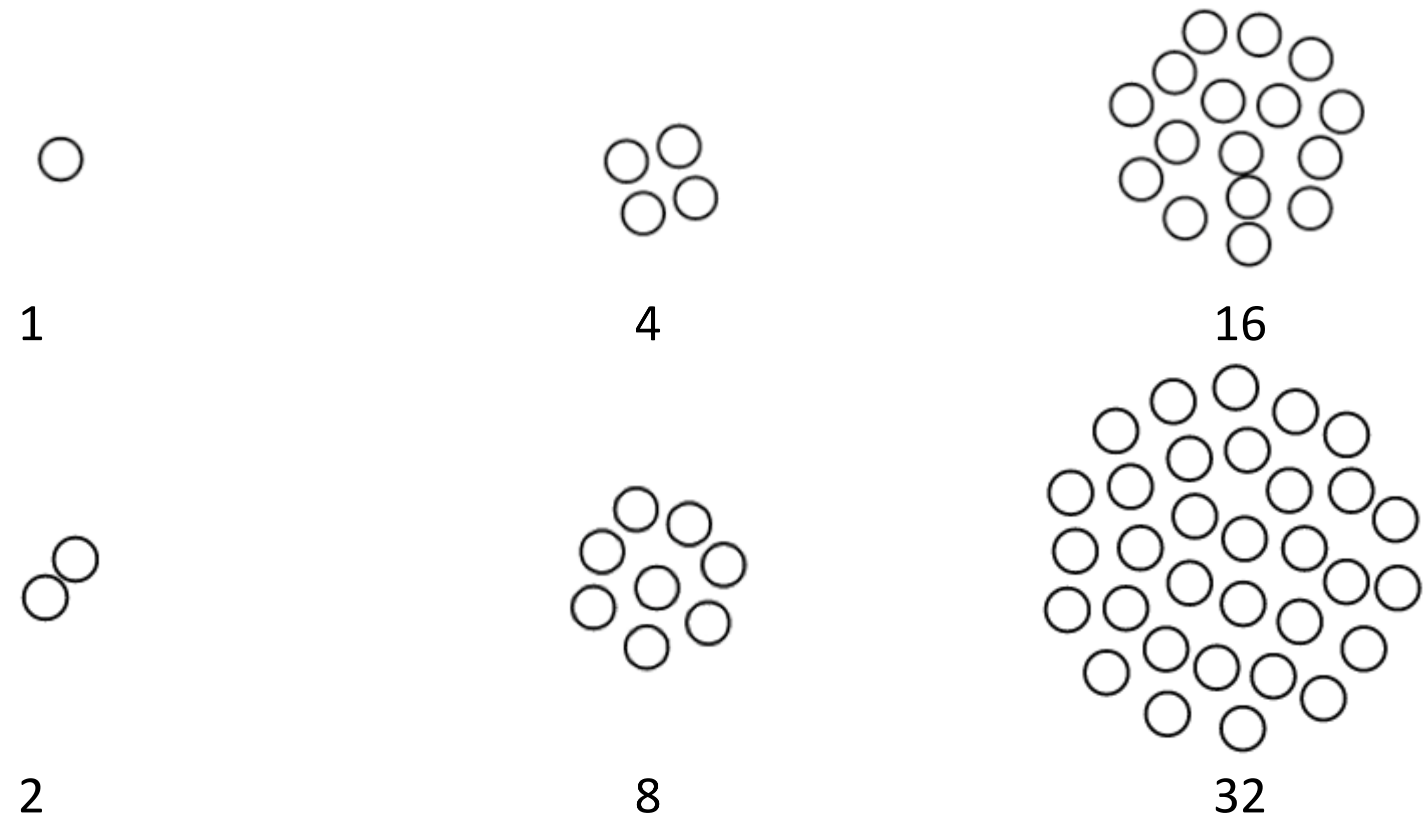
… and I am allowed only one more spin
now which should I choose?

→ value of exploration depends on temporal horizon



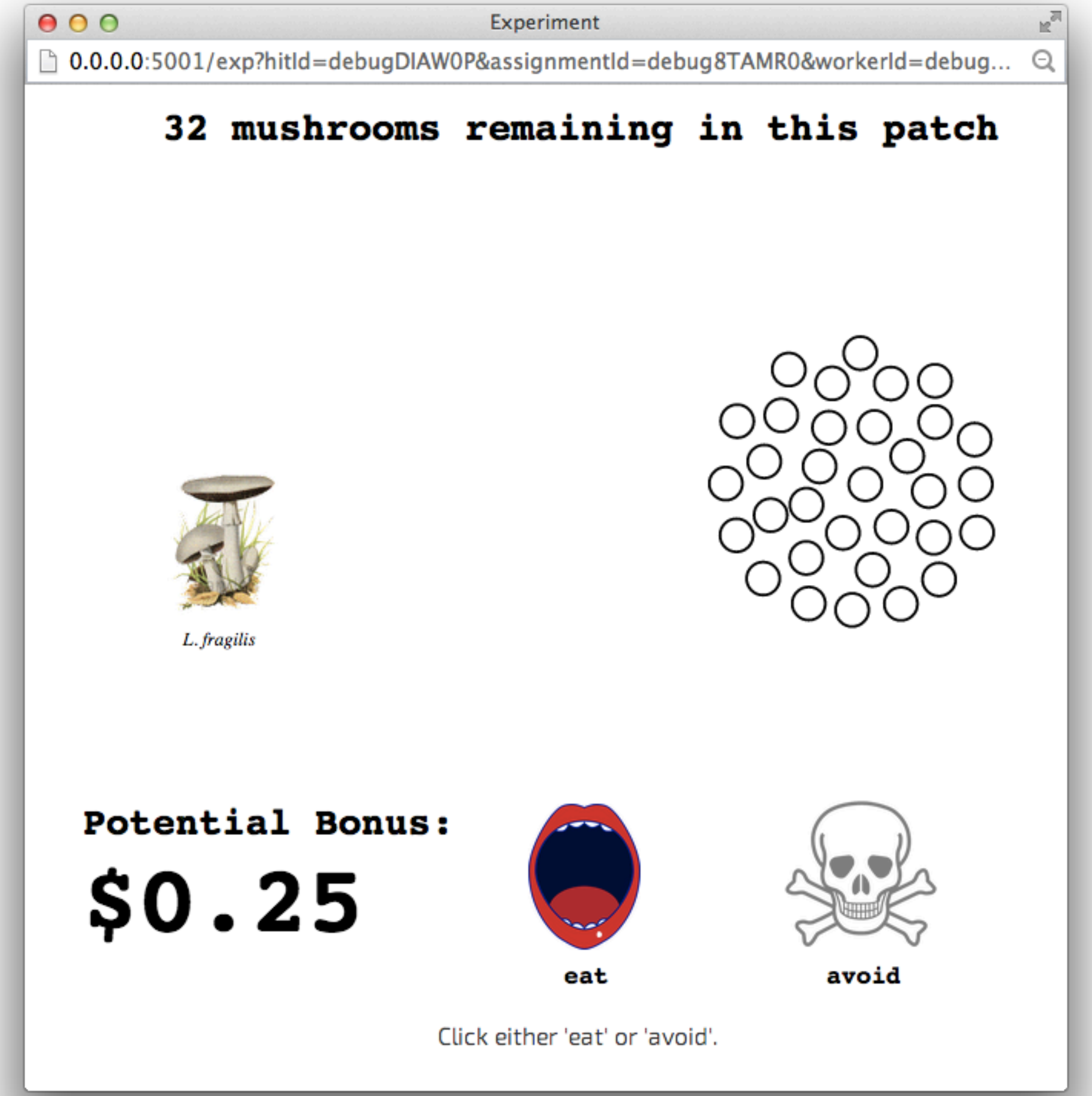
experiment 1

are people more likely to approach (i.e., explore) a uncertain prospect when they expect to encounter it a greater number of time in the future?

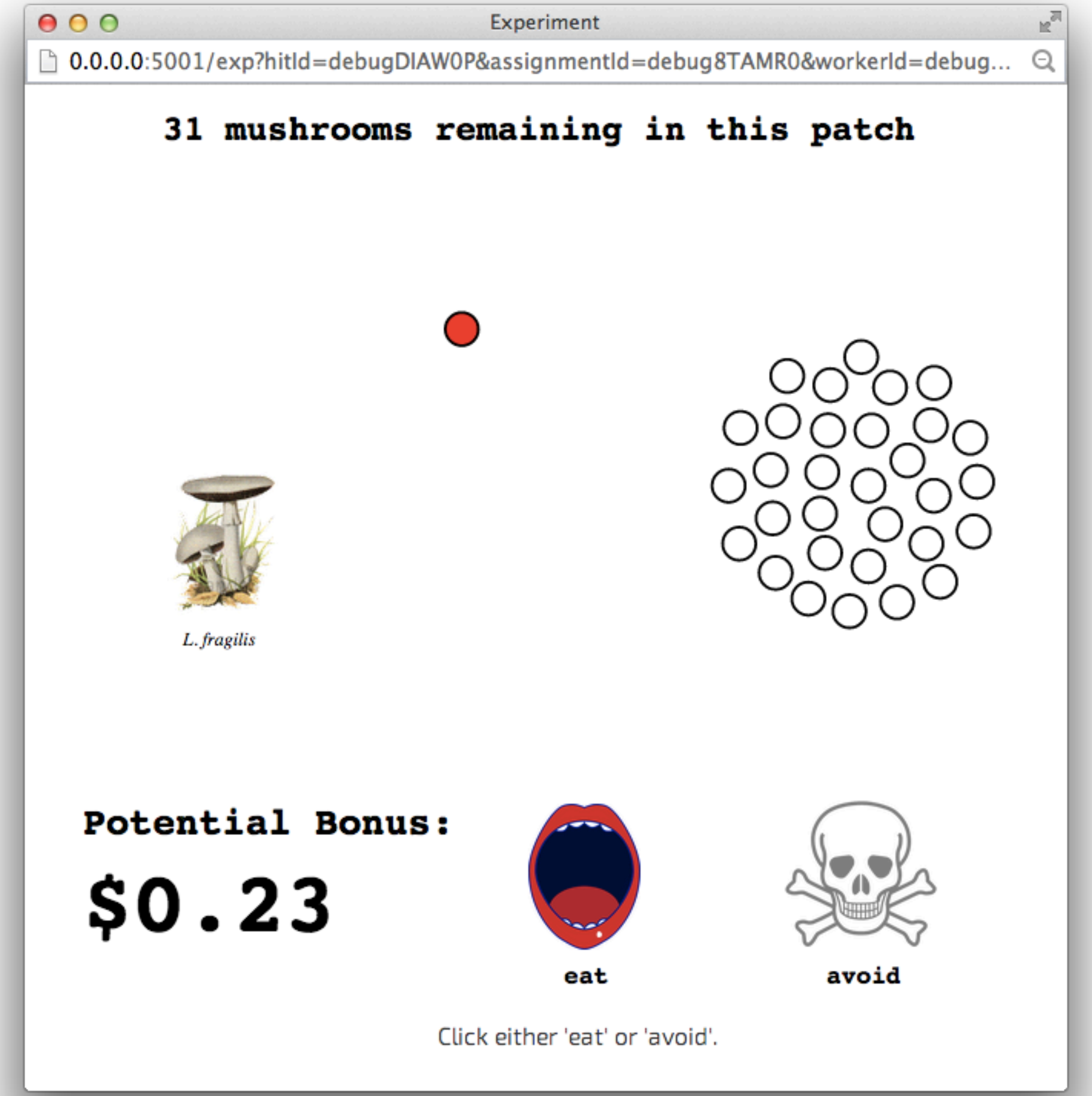


N = 143

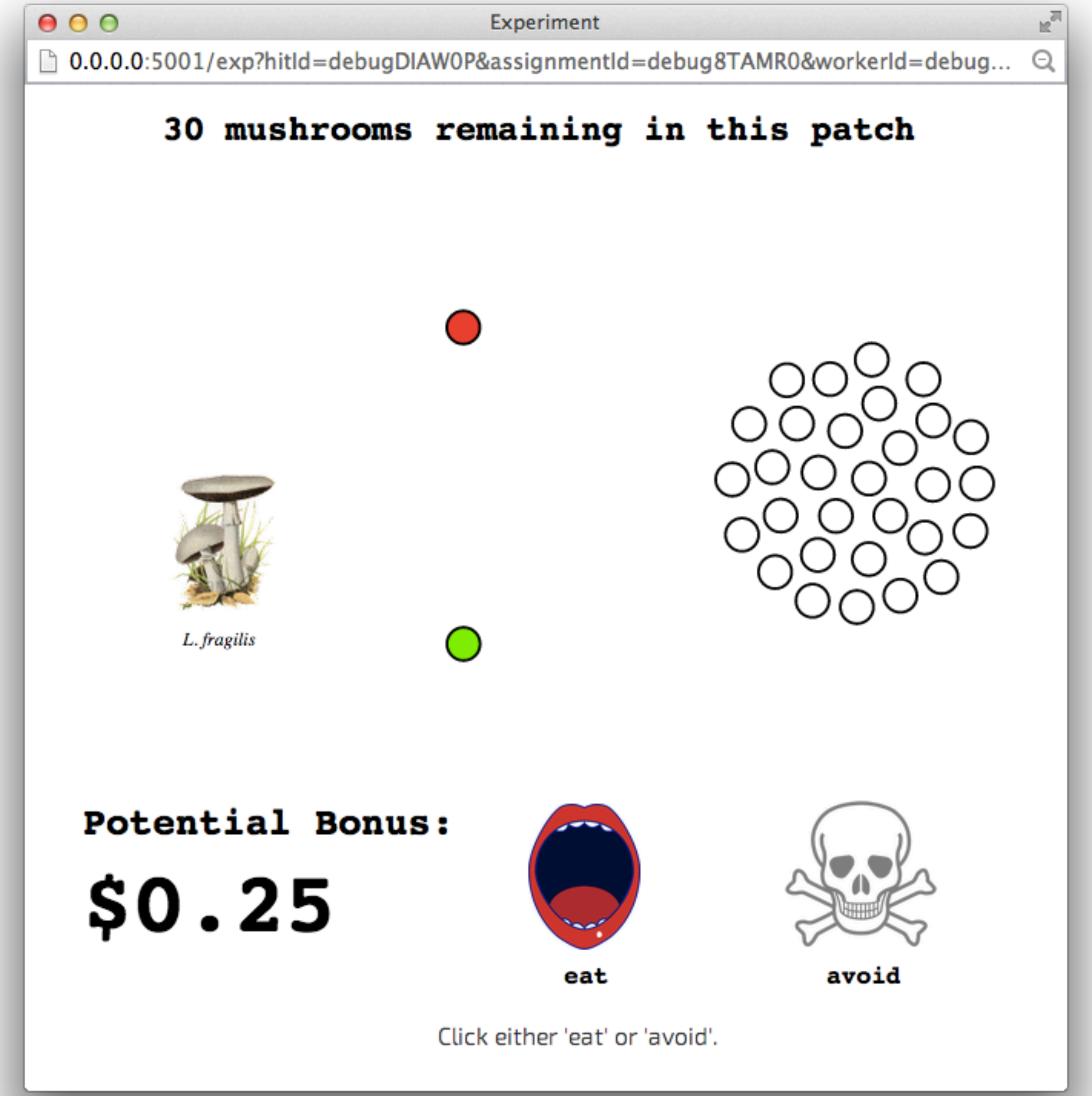
experiment 1



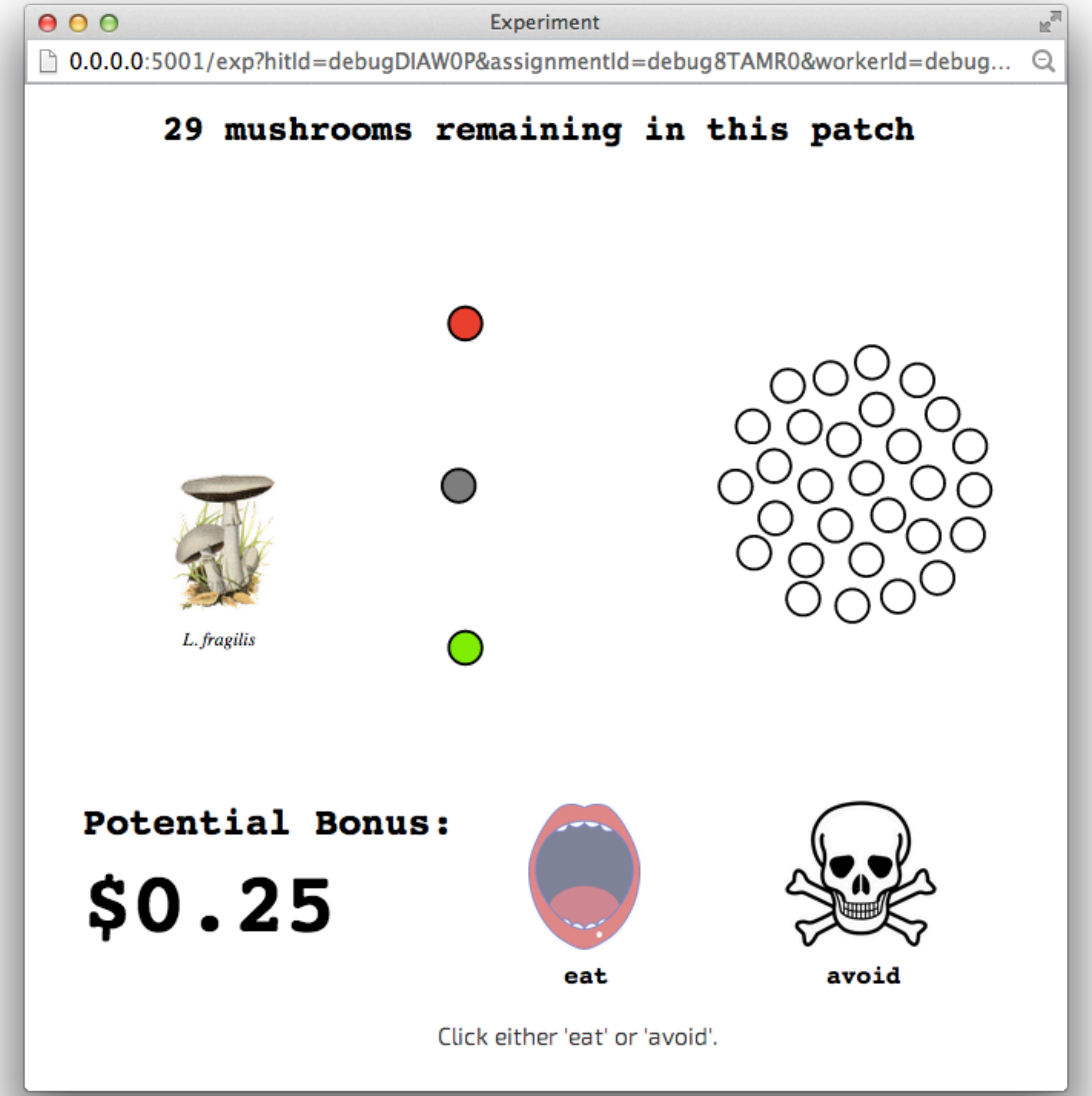
experiment 1



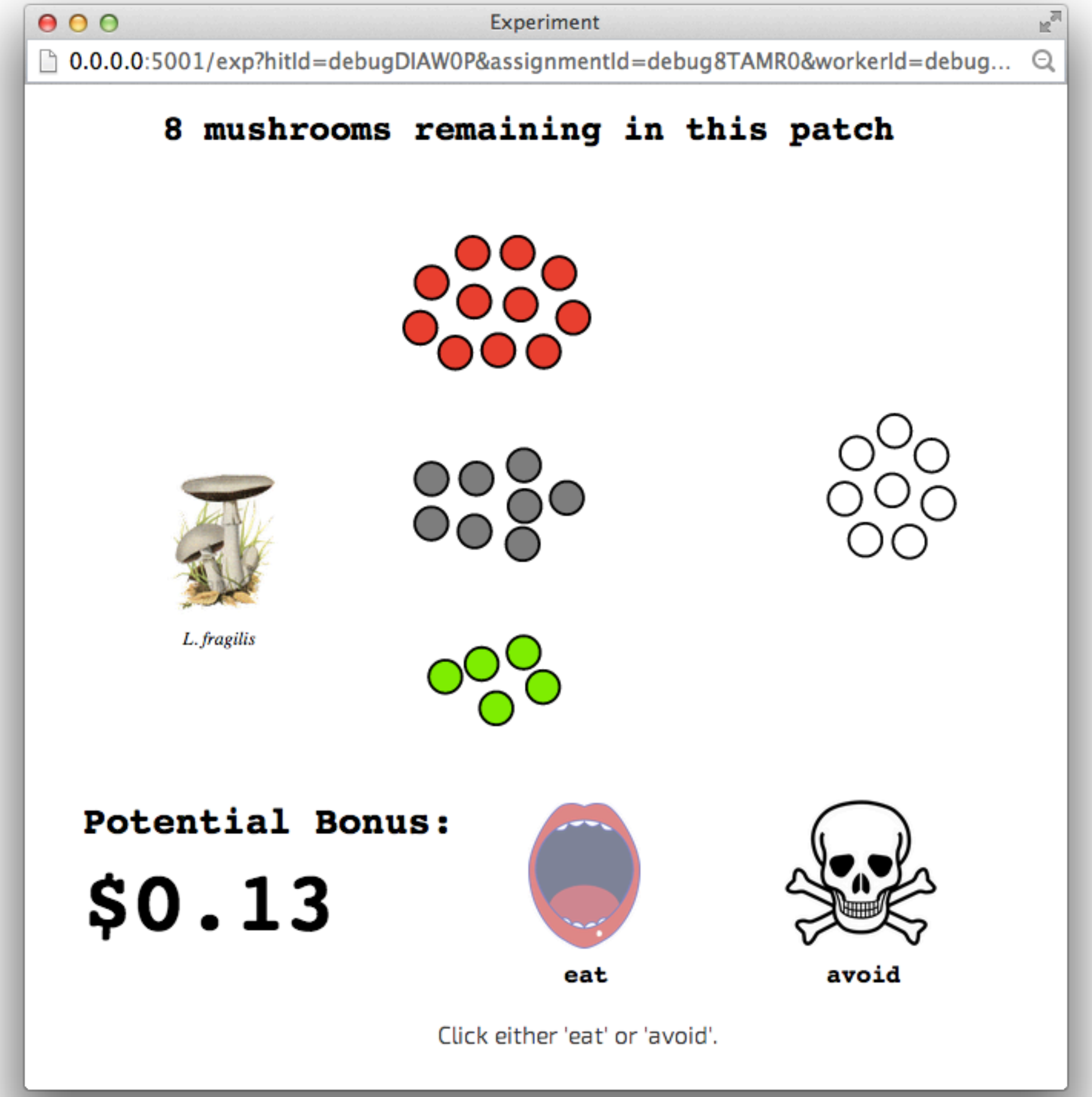
experiment 1



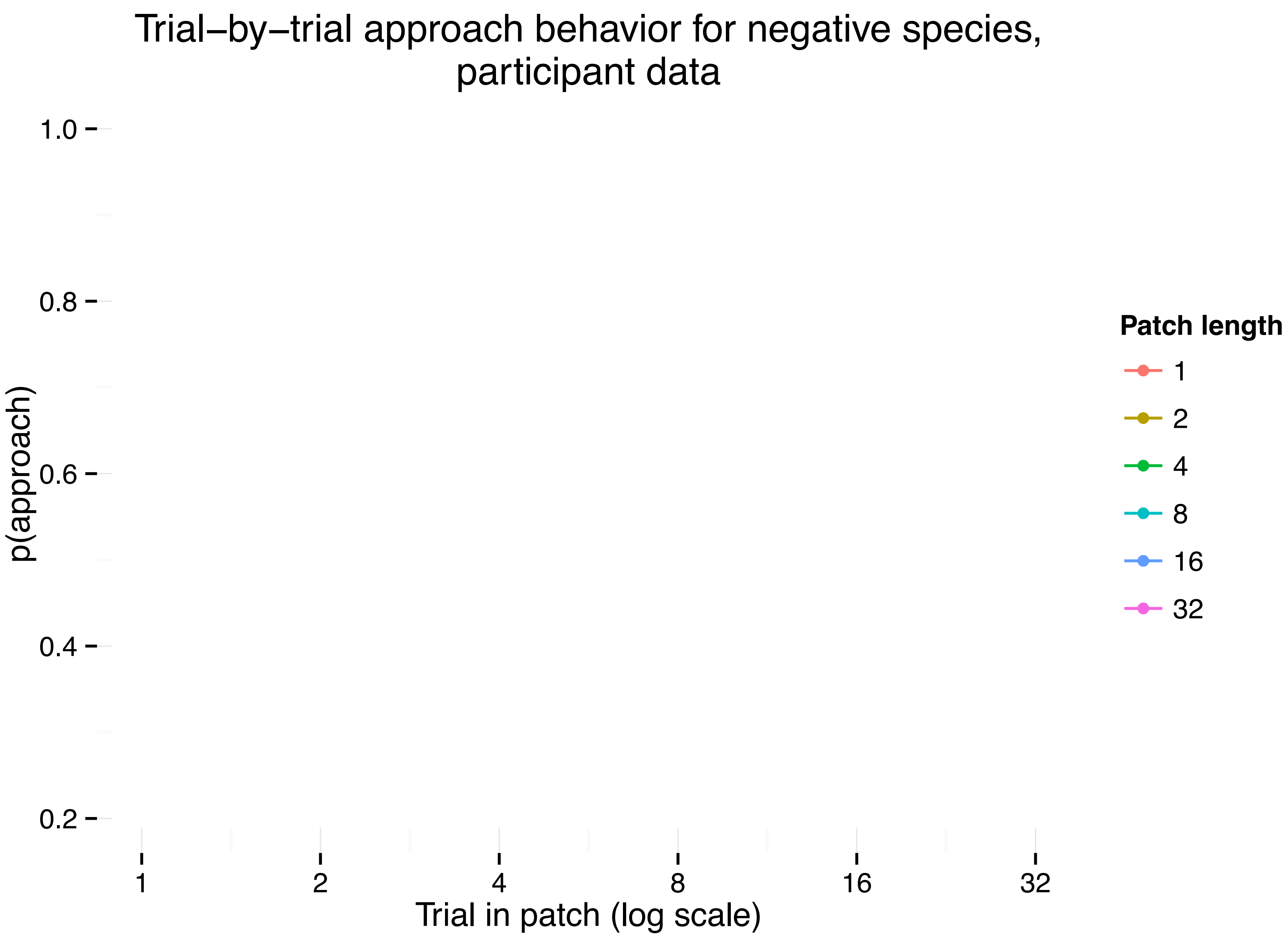
experiment 1



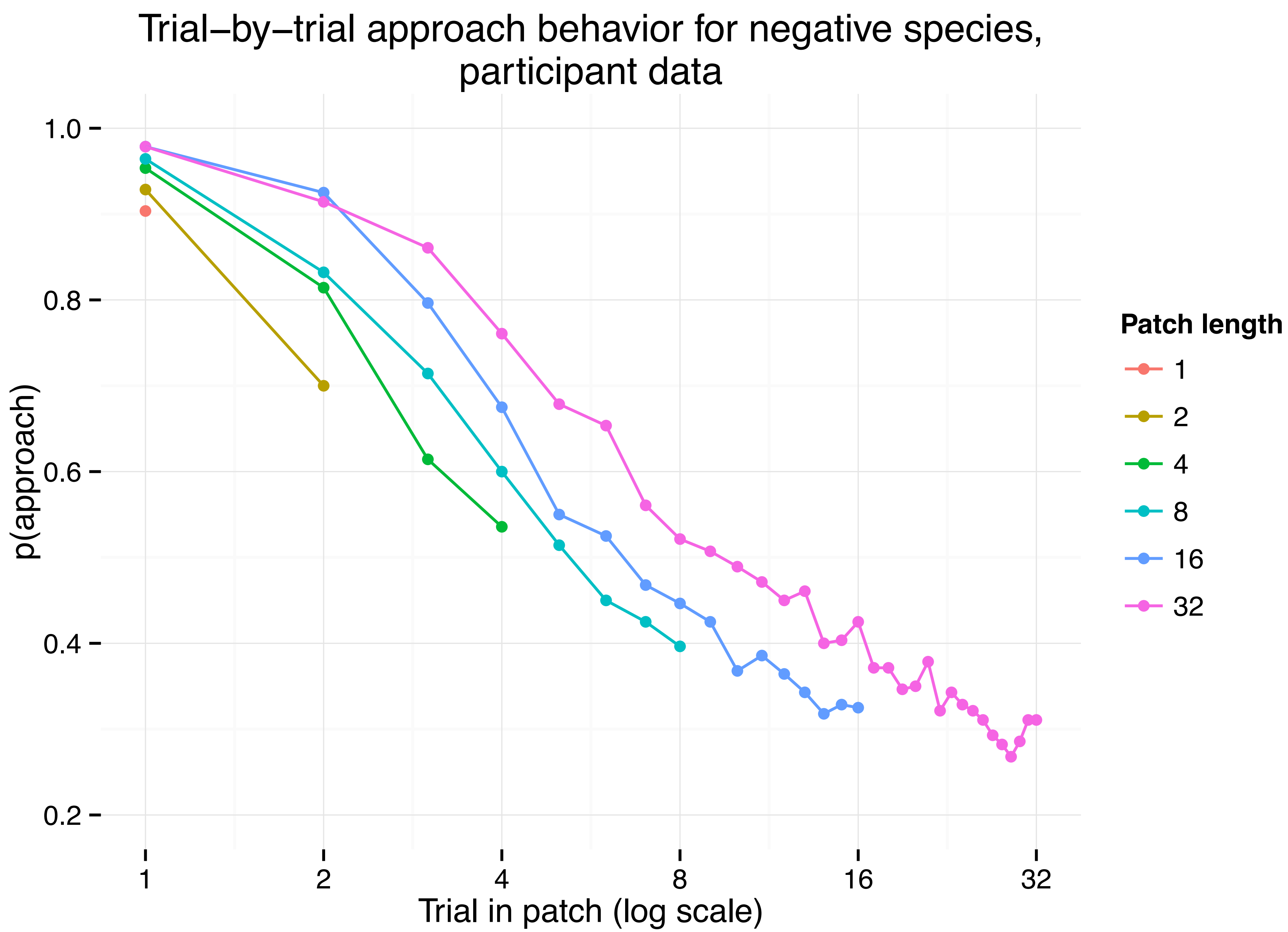
experiment 1



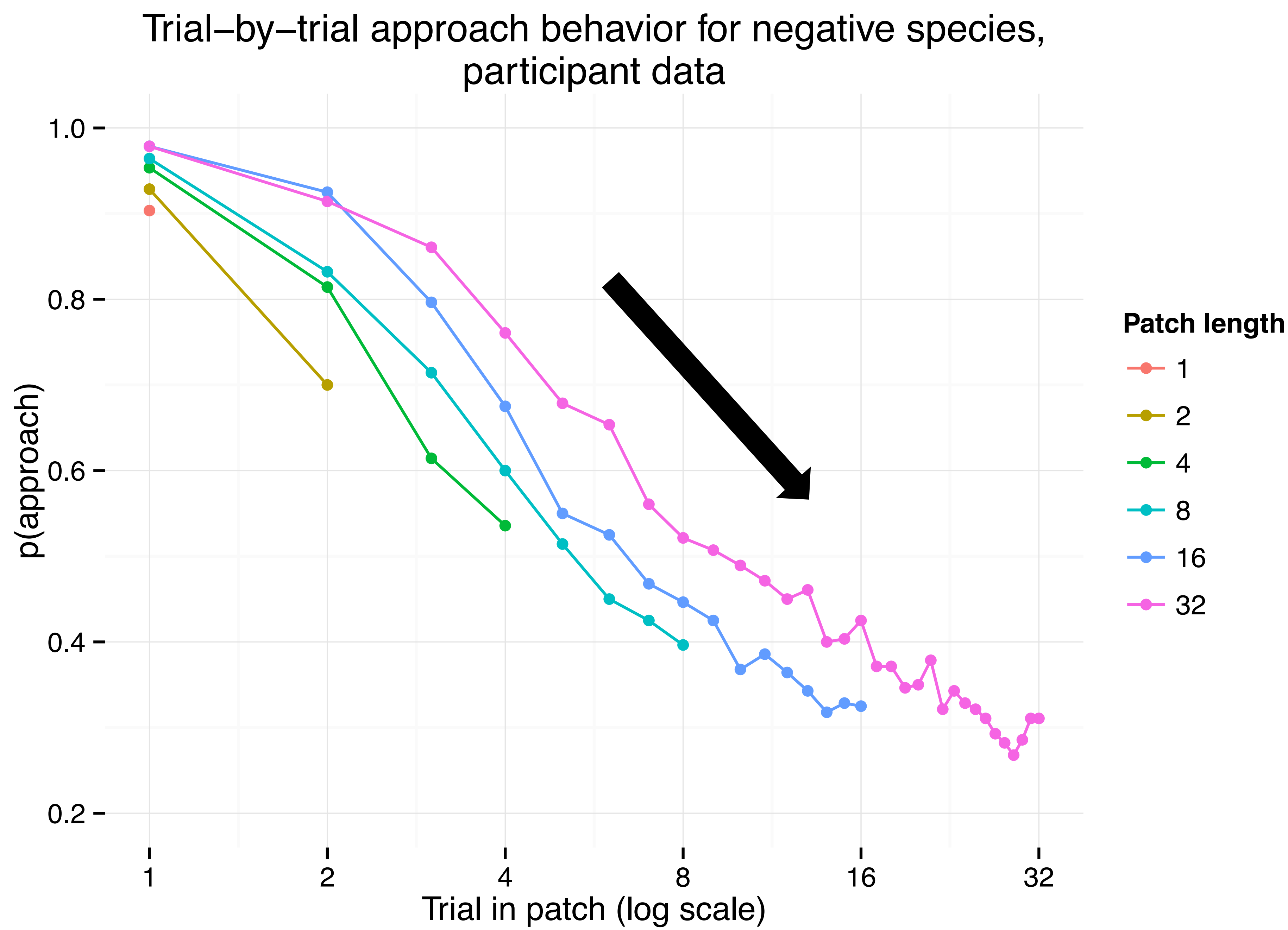
experiment 1 - results



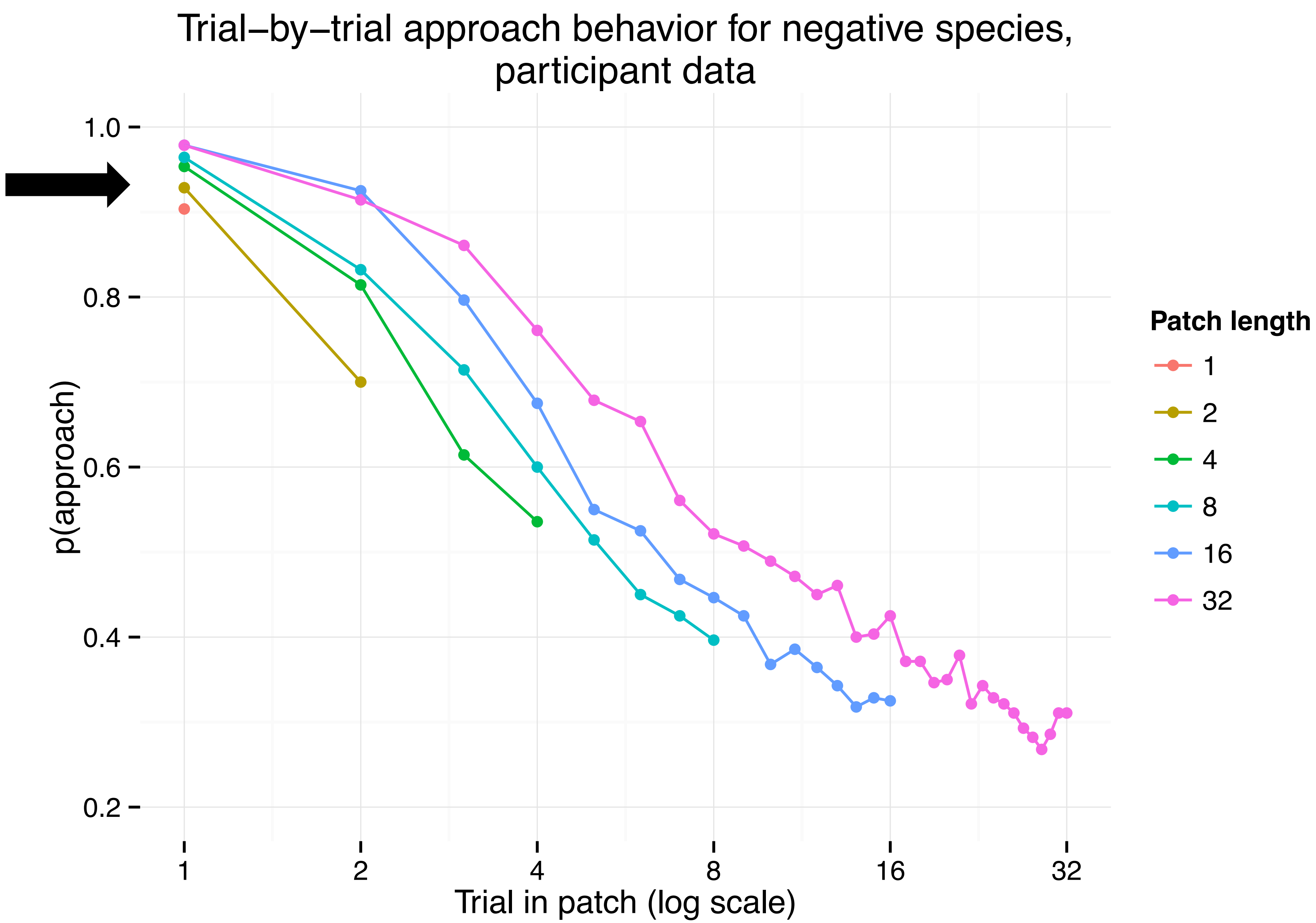
experiment 1 - results



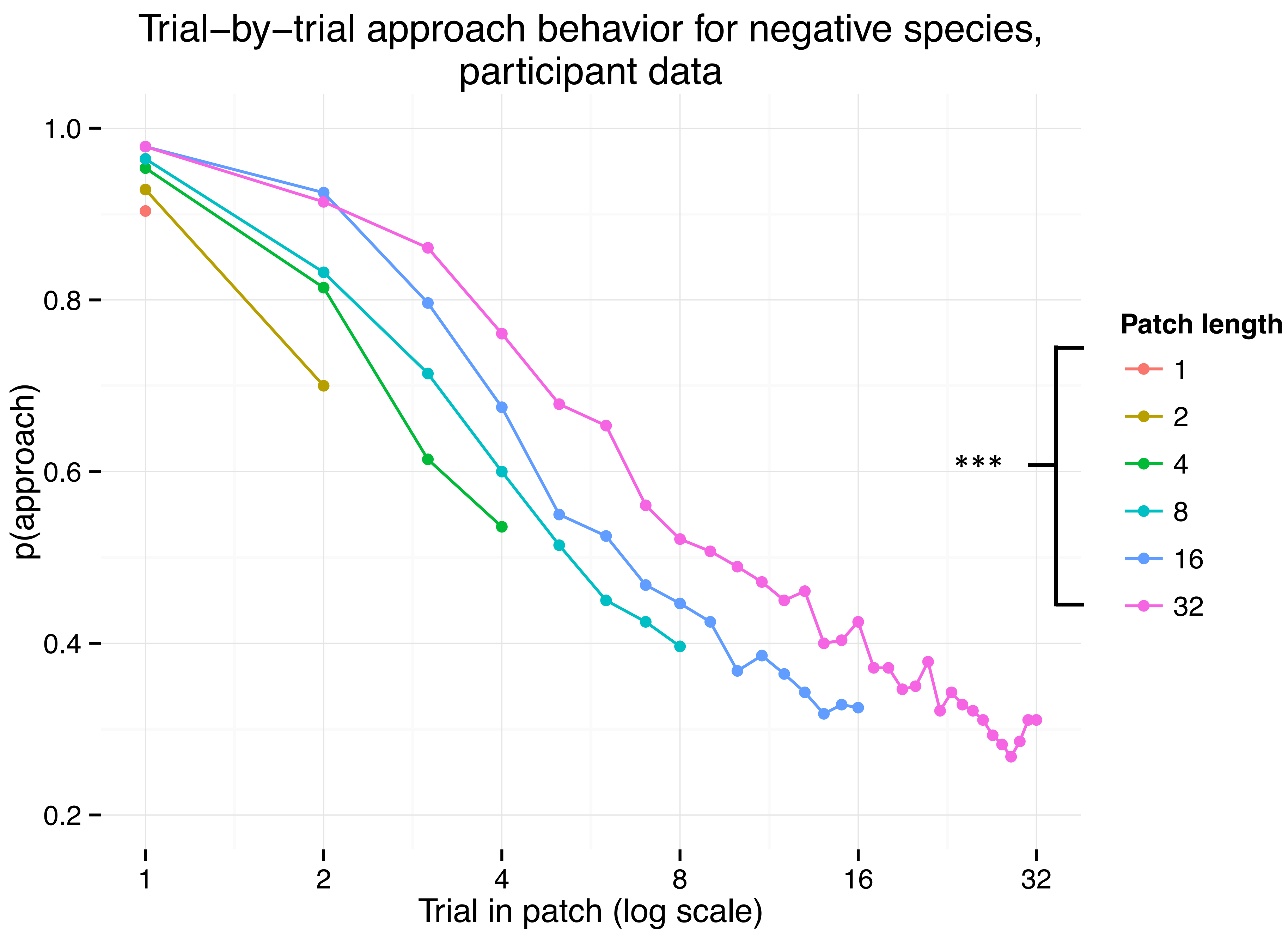
experiment 1 - results



experiment 1 - results



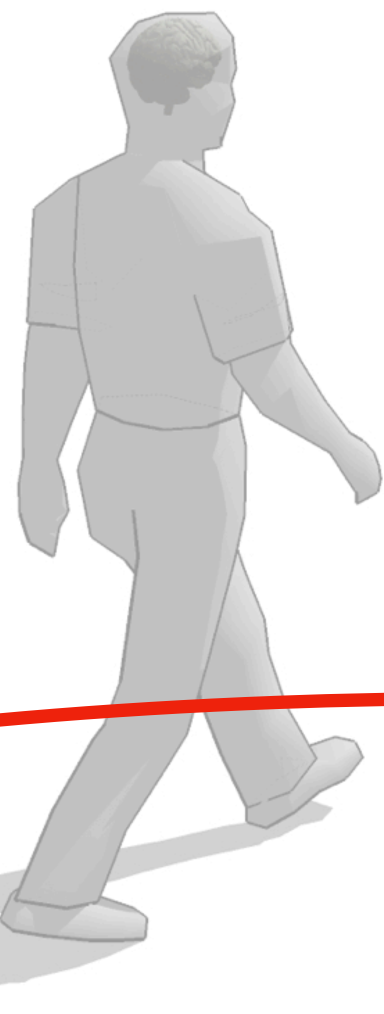
experiment 1 - results



Next time

- **Model-based RL/Planning**

Three levels of description *(David Marr, 1982)*



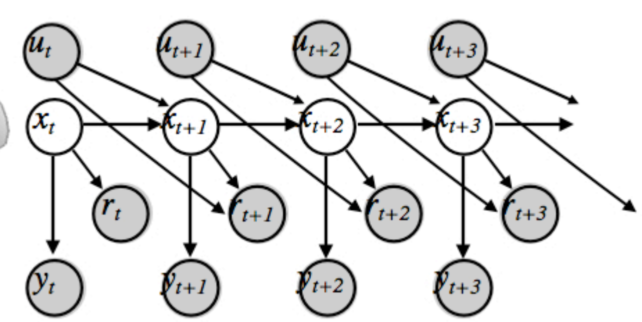
Computational
Why do things work the way they do?
What is the goal of the computation?
What are the unifying principles?

maximize:

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$

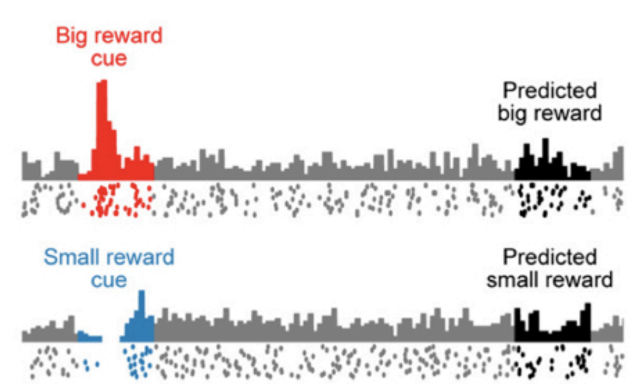
Bellman

Algorthmic
What representations can implement such computations?
How does the choice of representations determine the algorithm?



Dynamic programming, TD methods, Monte Carlo

Implementational
How can such a system be built in hardware?
How can neurons carry out the computations?



Neural firing patterns, prediction errors, system level neuroscience

Slide Credits

Nathaniel Daw (exploration/gittins)
Alex Rich
Gillian Hayes (TD methods/explore)
Rich Sutton (general approach)
Andy Barto (general approach)