

Chapter 8: Planning and Learning

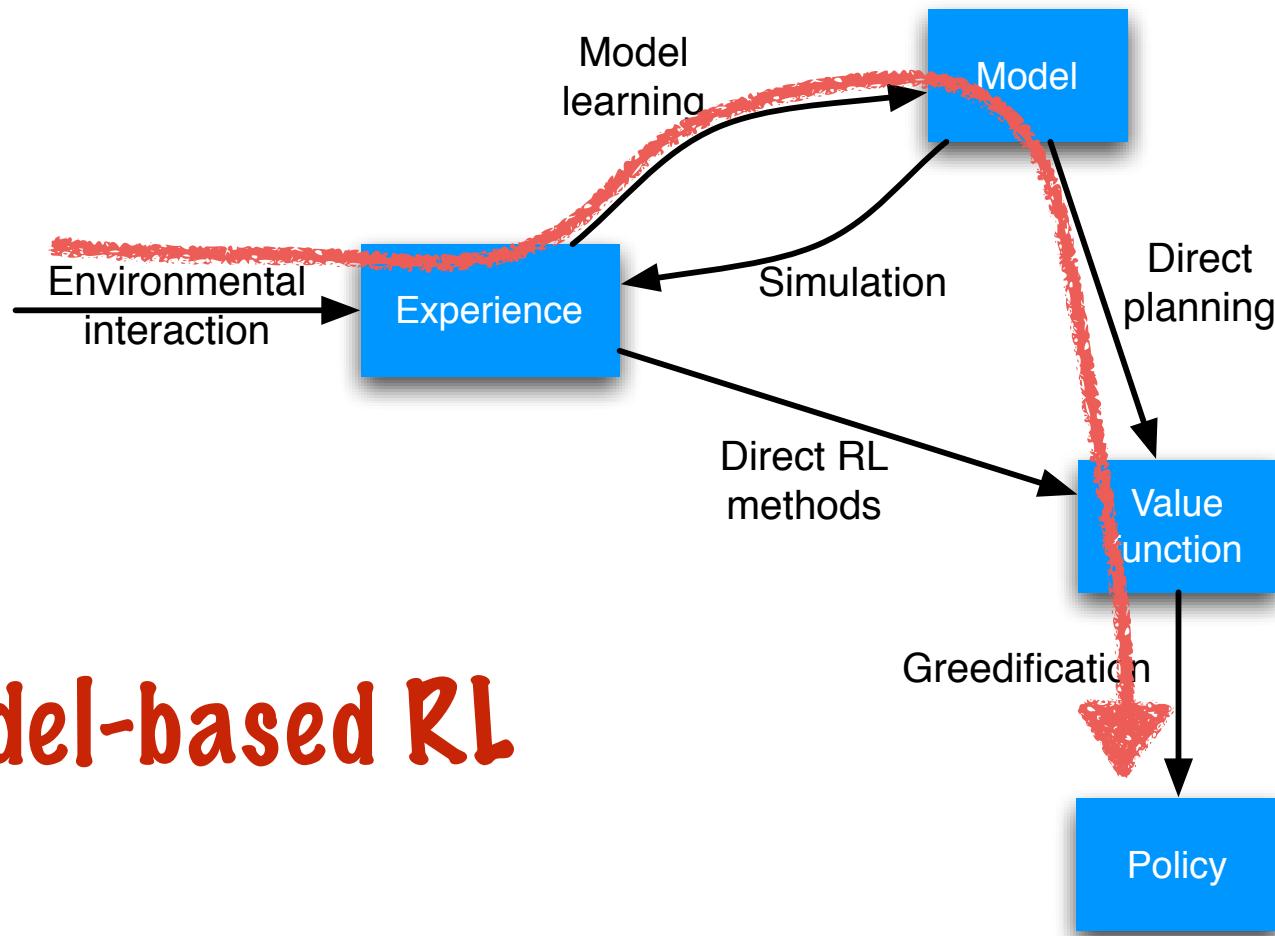
Objectives of this chapter:

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

DP with Distribution models

- In Chapter 4, we assumed access to a model of the world
 - These models describe all possibilities and their probabilities
 - We call them **Distribution models**
 - e.g., $p(s', r | s, a)$ for all s, a, s', r
- In Dynamic Programming we sweep the states:
 - in each state we consider all the possible rewards and next state values
 - the model describes the next states and rewards and their associated probabilities
 - using these values to update the value function
- In Policy Iteration, we then improve the policy using the computed value function

Paths to a policy



Sample Models

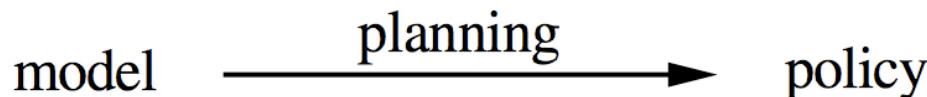
- **Model**: anything the agent can use to predict how the environment will respond to its actions
- **Sample model**, a.k.a. a simulation model
 - produces sample experiences for given s, a
 - sampled according to the probabilities
 - allows reset, exploring starts
 - often much easier to come by
- Both types of models can be used mimic or simulate experience: to produce **hypothetical experience**

Models

- Consider modeling the sum of two dice
 - A *distribution model* would produce all possible sums and their probabilities of occurring
 - A *sample model* would produce an individual sum drawn according to the correct probability distribution
- When we solved the Gambler's problem with value iteration, we used the distribution model
- When you solved the Gambler's problem with Monte-Carlo, you implemented a sample model in your environment code

Planning

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



- We take the following (unusual) view:
 - update value functions using both real and simulated experience
 - all state-space planning methods involve computing value functions, either explicitly or implicitly
 - they all apply updates from 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:

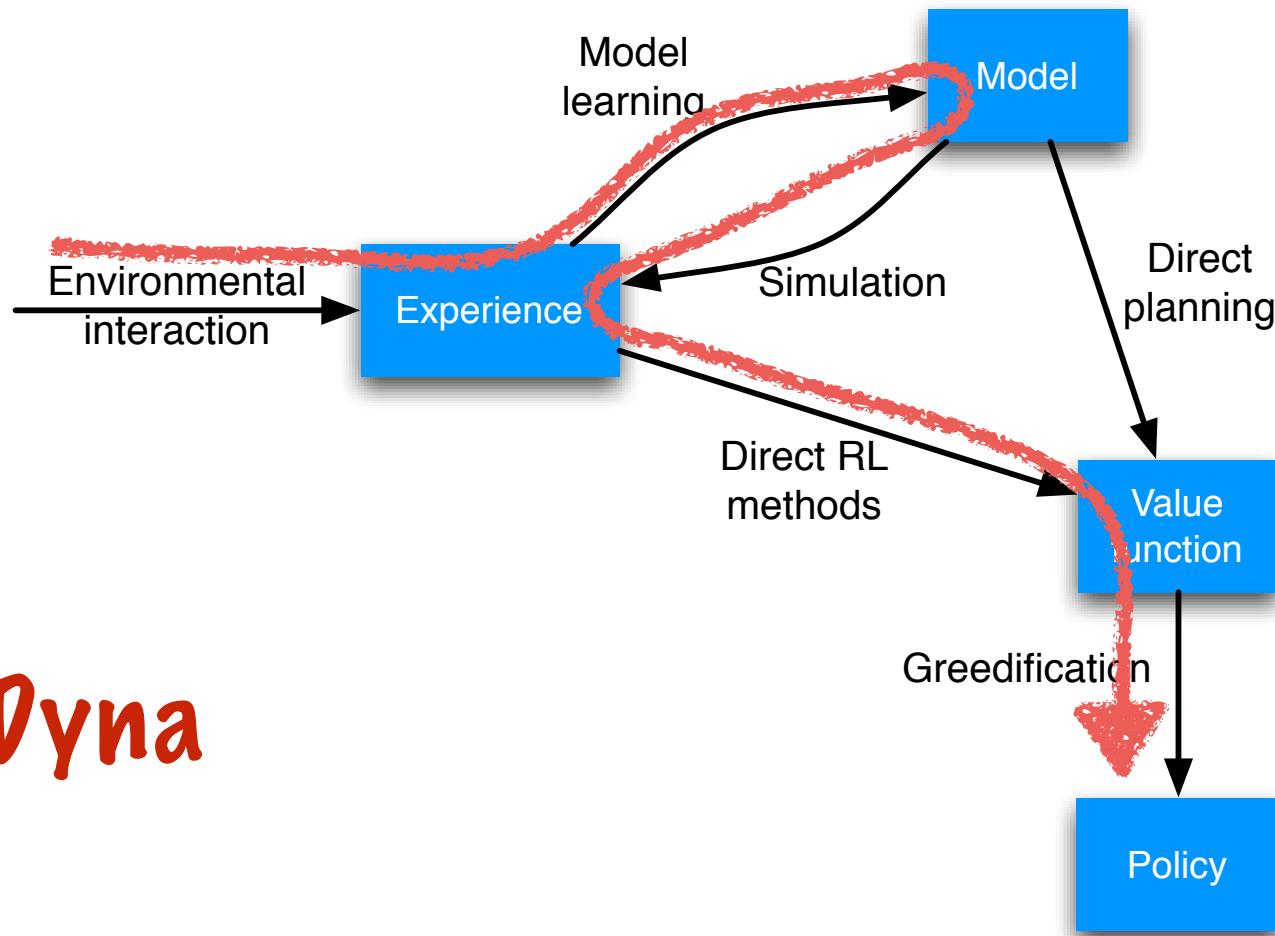
Random-sample one-step tabular Q-planning

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

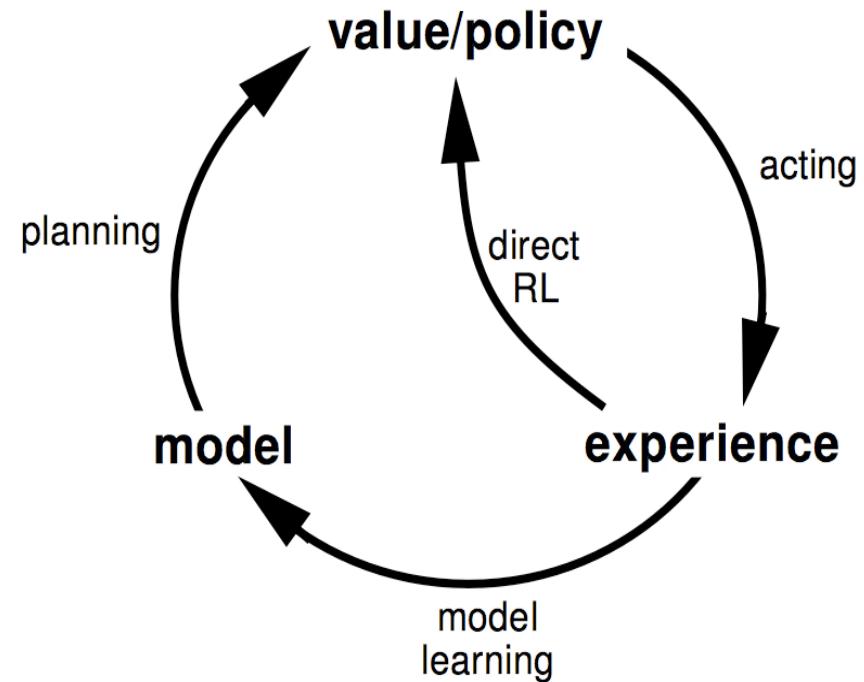
Environment program
Experiment program
Agent program

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

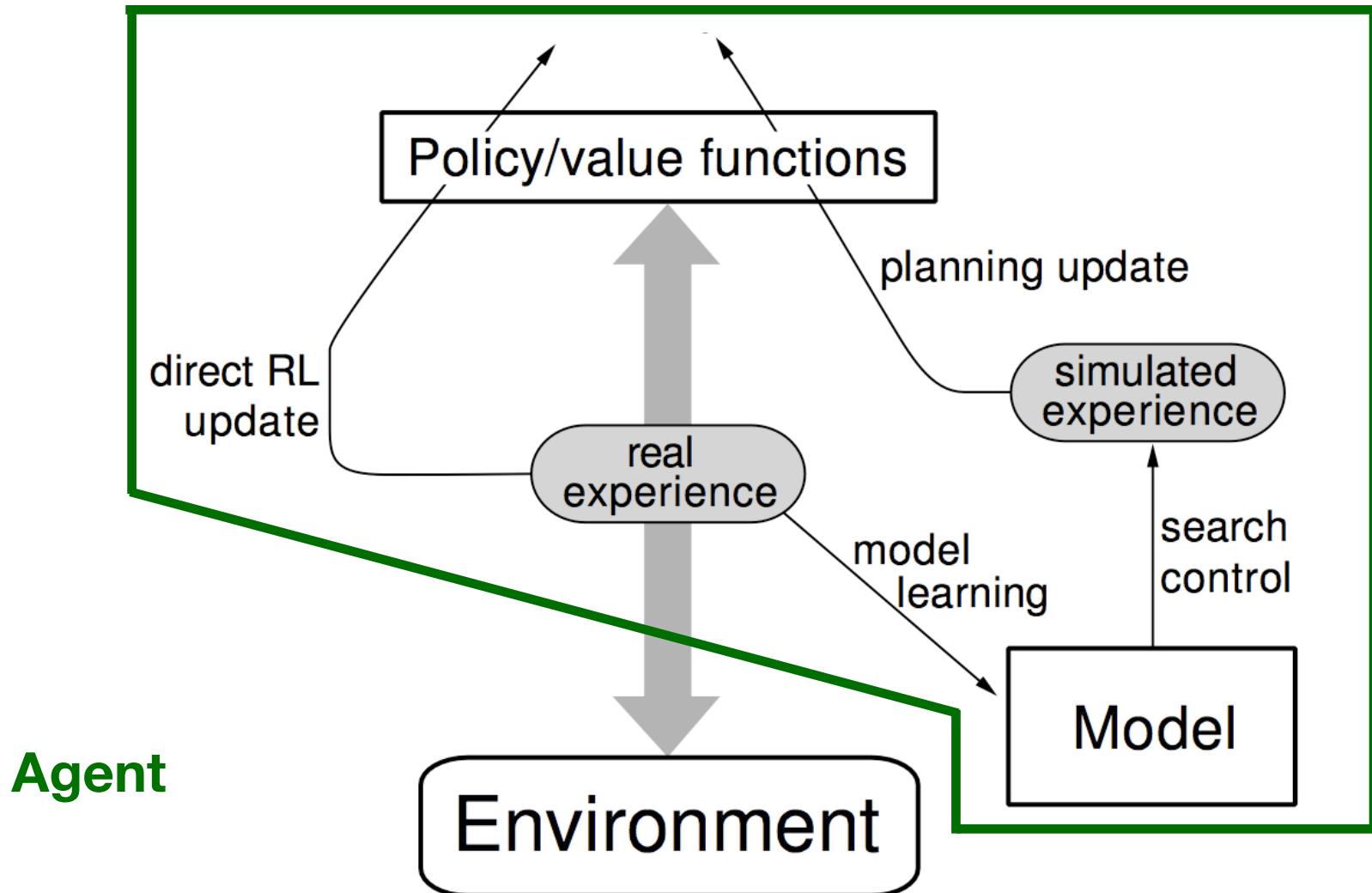
- Indirect methods:

- 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

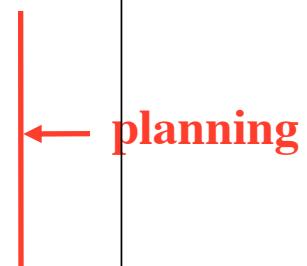
(f) Repeat n times:

$S \leftarrow$ random previously observed state

$A \leftarrow$ random action previously taken in S

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

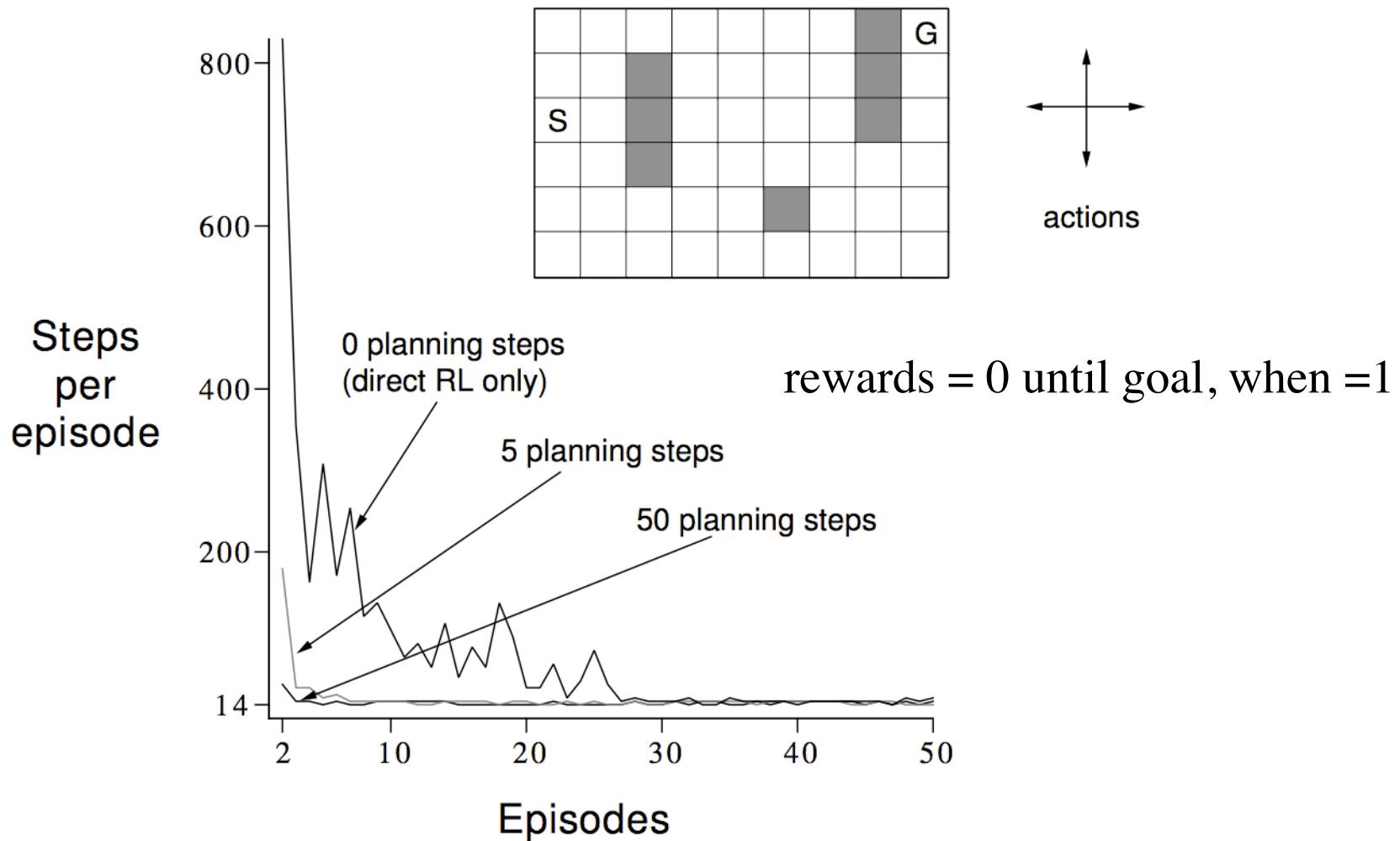
$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$



A simple maze: problem description

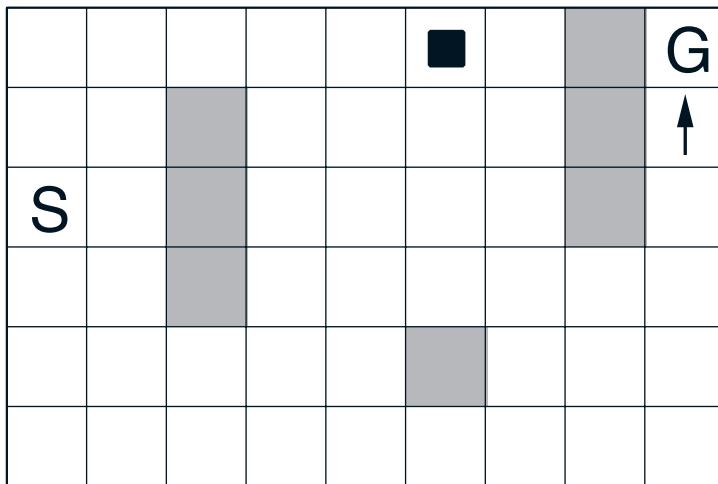
- 47 states, 4 actions, deterministic dynamics
- Obstacles and walls
- Rewards are 0 except +1 for transition into goal state
- $\gamma = 0.95$, discounted episodic task
- Agent parameters:
 - $\alpha = 0.1$, $\epsilon = 0.1$
 - Initial action-values were all zero
- Let's compare one-step tabular Q-learning and Dyna-Q with different values of n

Dyna-Q on a Simple Maze

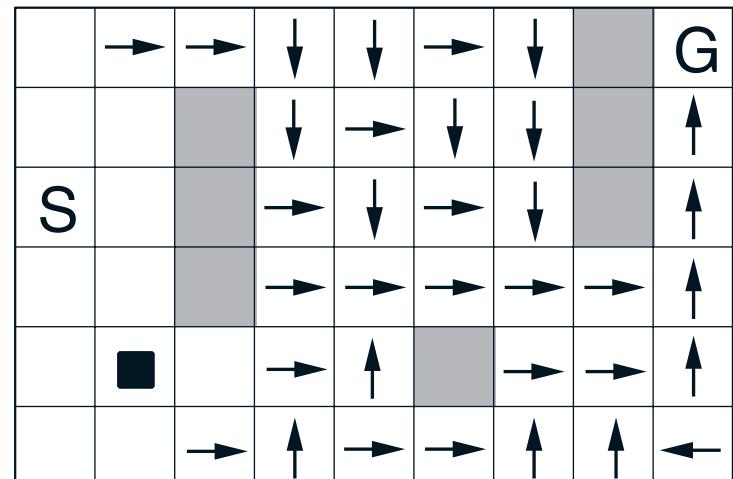


Dyna-Q Snapshots: Midway in 2nd Episode

WITHOUT PLANNING ($n=0$)



WITH PLANNING ($n=50$)



Implementation details

- Notice during the demo, that the one-step Q-learning agent and Dyna-Q agent appeared to be equally reactive
- Updating the value function and selecting a new action is very fast
 - Thus, there is usually some time left over
 - We can use that time to run a planning loop
 - Planning with $n=5$ helps a lot
- What are other ways we could integrate planning with learning and acting?
- Planning of this form is *anytime*

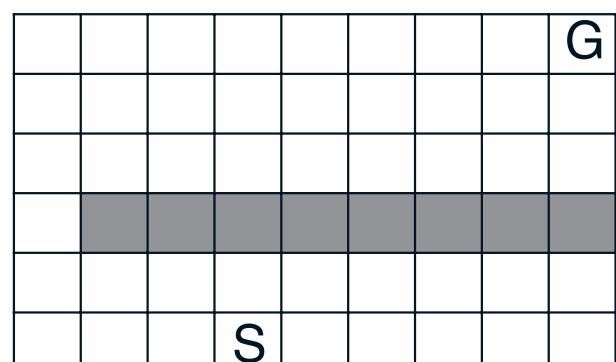
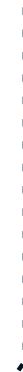
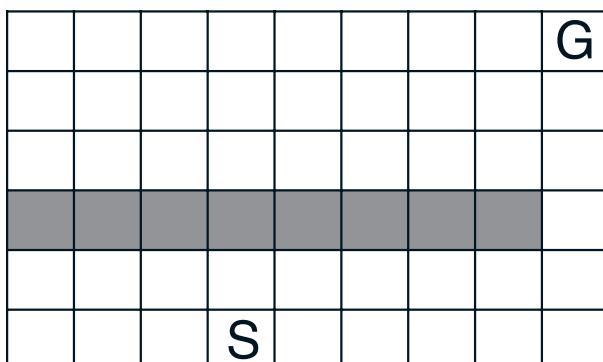
When the model is wrong

- So far we have considered models, that:
 - Start empty and are always updated with correct info.
- The model can be wrong! Because:
 - environment might be stochastic and we have only seen a few samples
 - the environment has changed
- Planning is likely to compute a suboptimal policy in this case

- Imagine the world changed, and:
 - The suboptimal policy leads to discovery and correction of the modeling error

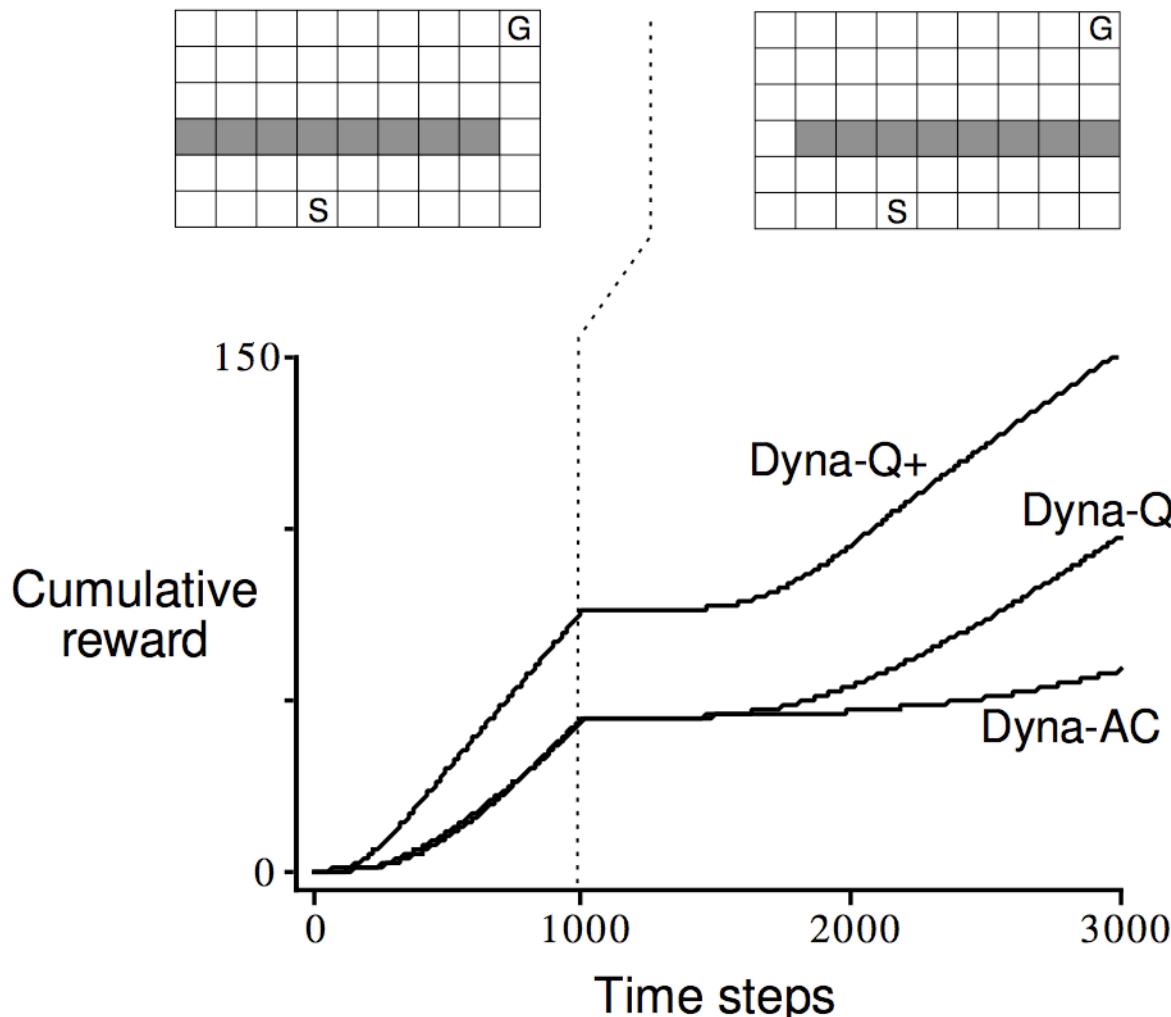
Blocking Maze problem

- Consider the Gridworld with a partition wall
- Dyna-Q can find the policy for reaching the goal state
- Then after 1000 time steps, we change the world:
 - we block the path of the existing planned policy
 - and open up a new path



When the Model is Wrong: Blocking Maze

The changed environment is harder

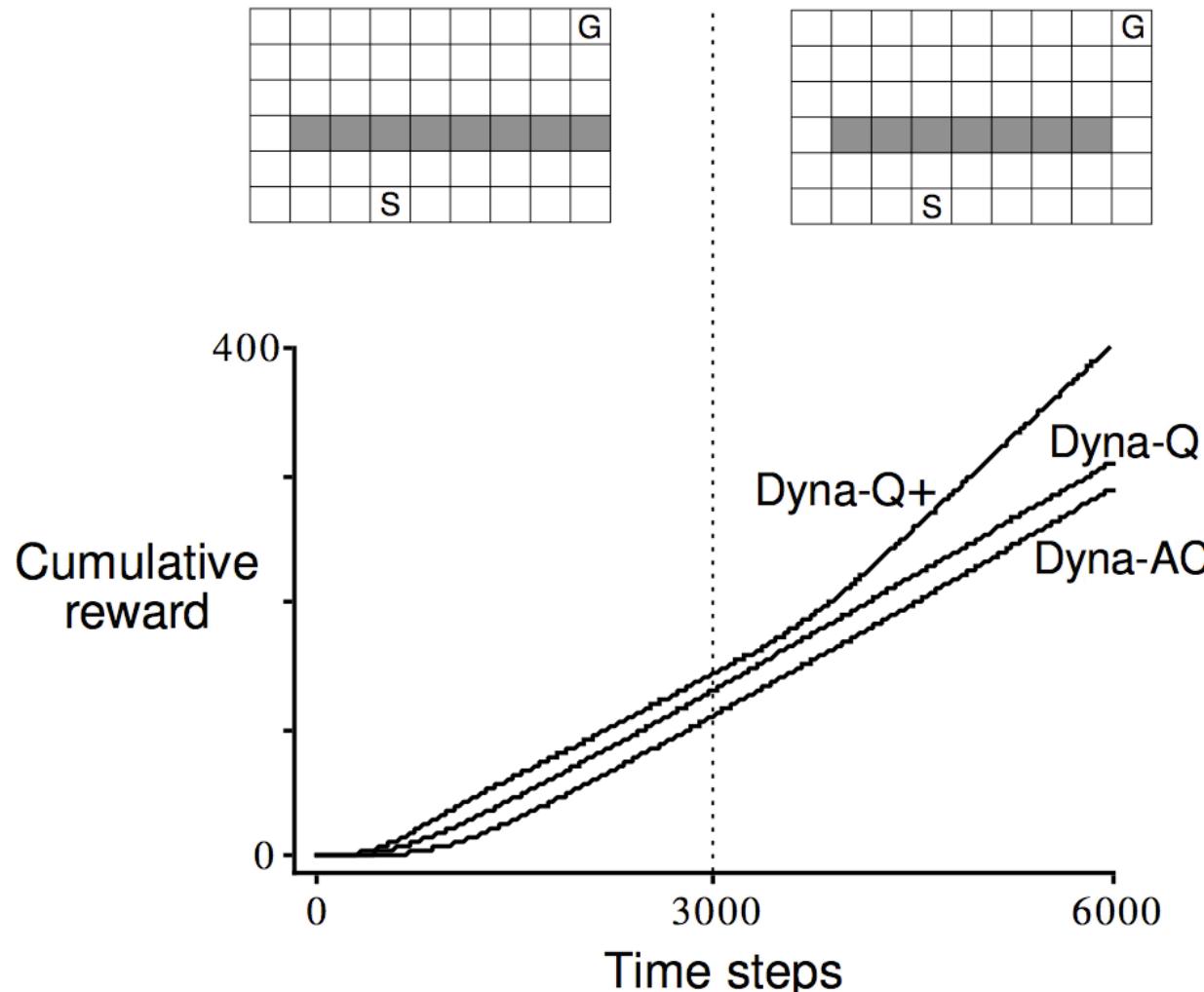


A change for the better

- If the world changes, such that a better path is possible:
 - the formerly correct policy will not reveal the improved situation
 - the modeling error may not be detected for a long time!

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

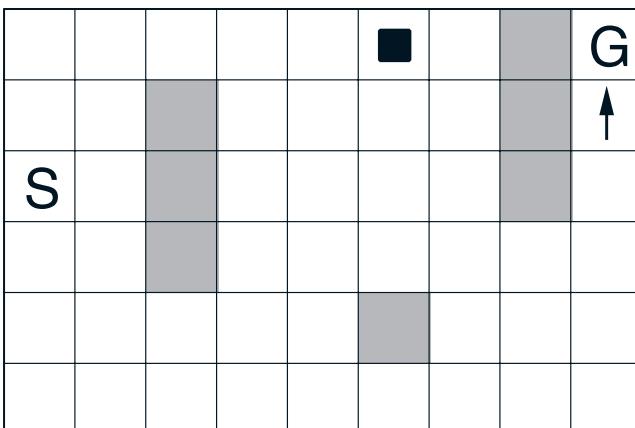
The conflict between exploration and exploitation

- Exploration in planning: trying actions that improve the model
 - Make it more accurate
 - Make it a better match with the environment
 - Proactively discover when the model is wrong
- Exploitation: behaving optimally with respect to the current model
- Simple heuristics can be effective

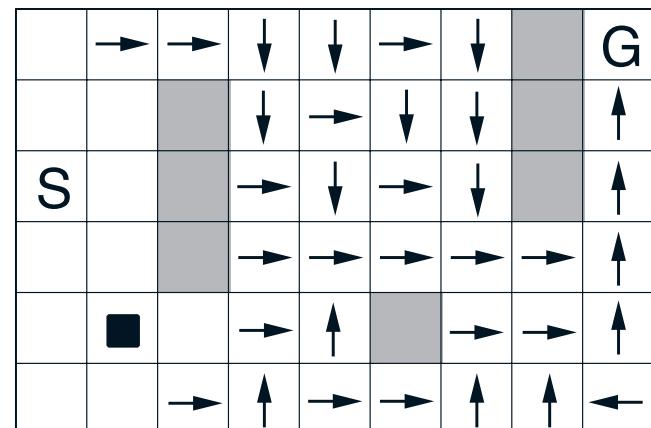
Prioritizing Search Control

- Consider the second episode in the Dyna maze
 - The agent has successfully reached the goal once...

WITHOUT PLANNING ($n=0$)

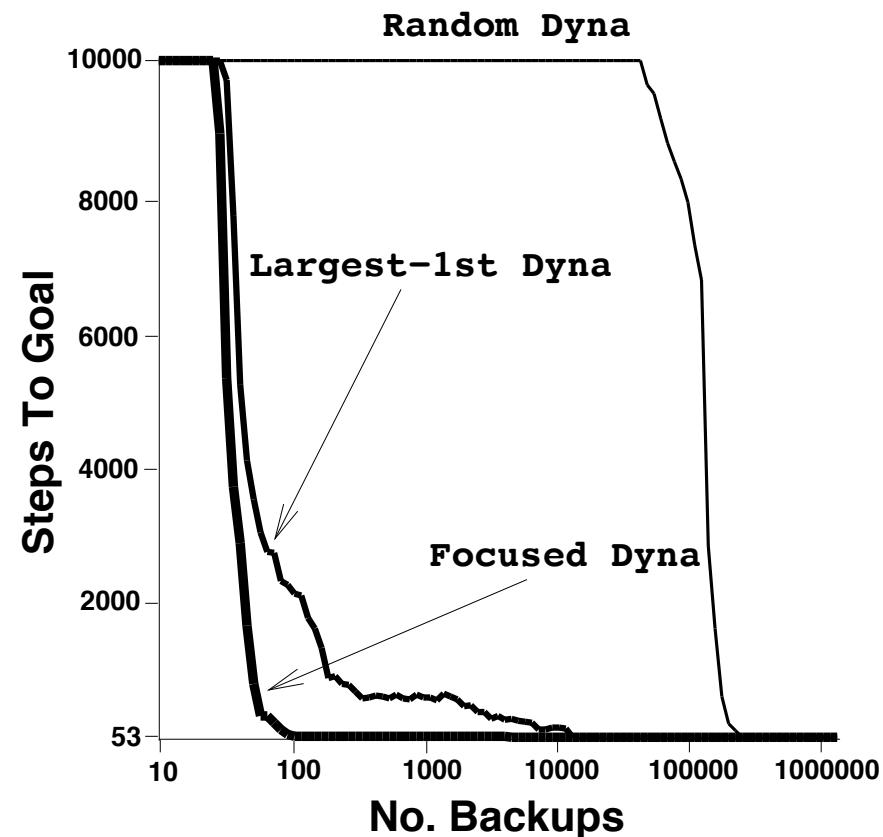
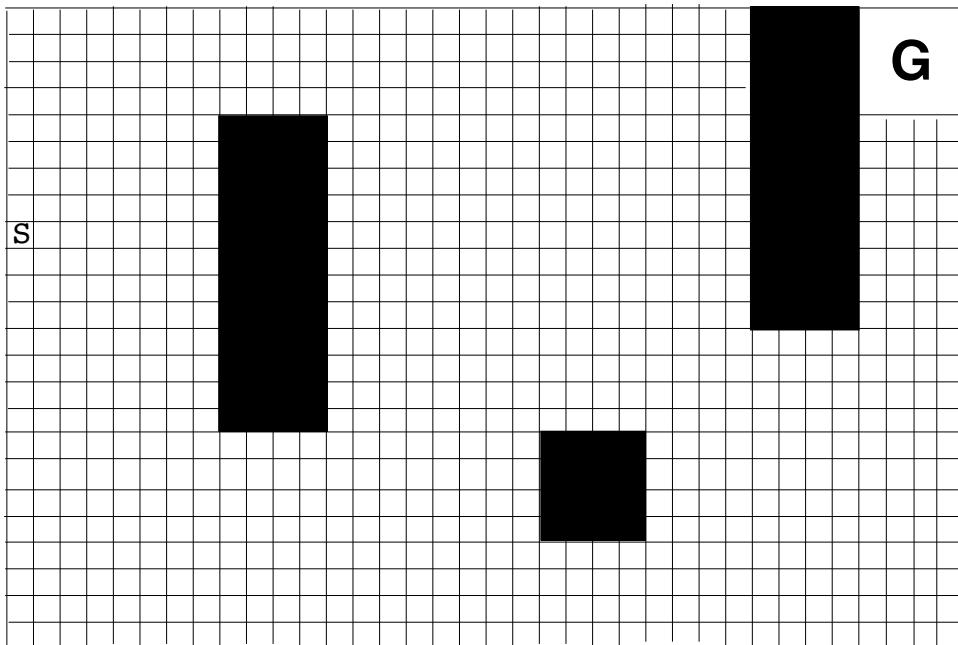


WITH PLANNING ($n=50$)



- In larger problems, the number of states is so large that unfocused planning would be extremely inefficient

Large maze and random search control



(Peng and Williams, 1993)

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:

$S, A \leftarrow first(PQueue)$

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

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

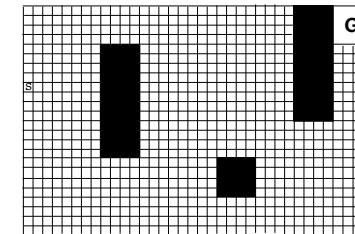
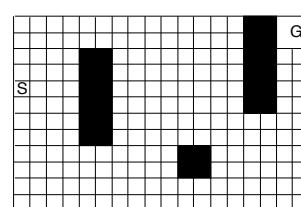
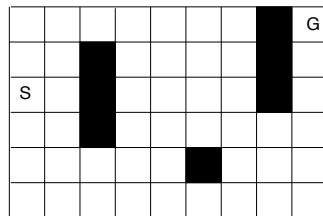
Repeat, for all \bar{S}, \bar{A} predicted to lead to S :

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

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

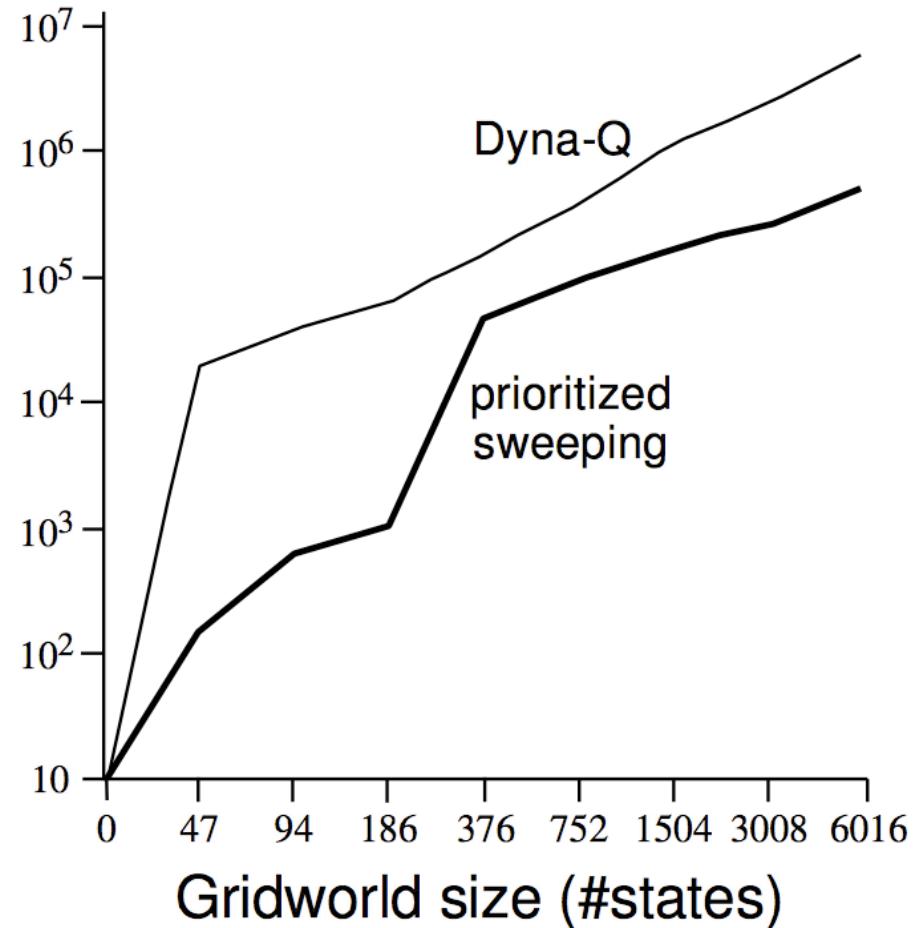
if $P > \theta$ then insert \bar{S}, \bar{A} into $PQueue$ with priority P

Prioritized Sweeping vs. Dyna-Q

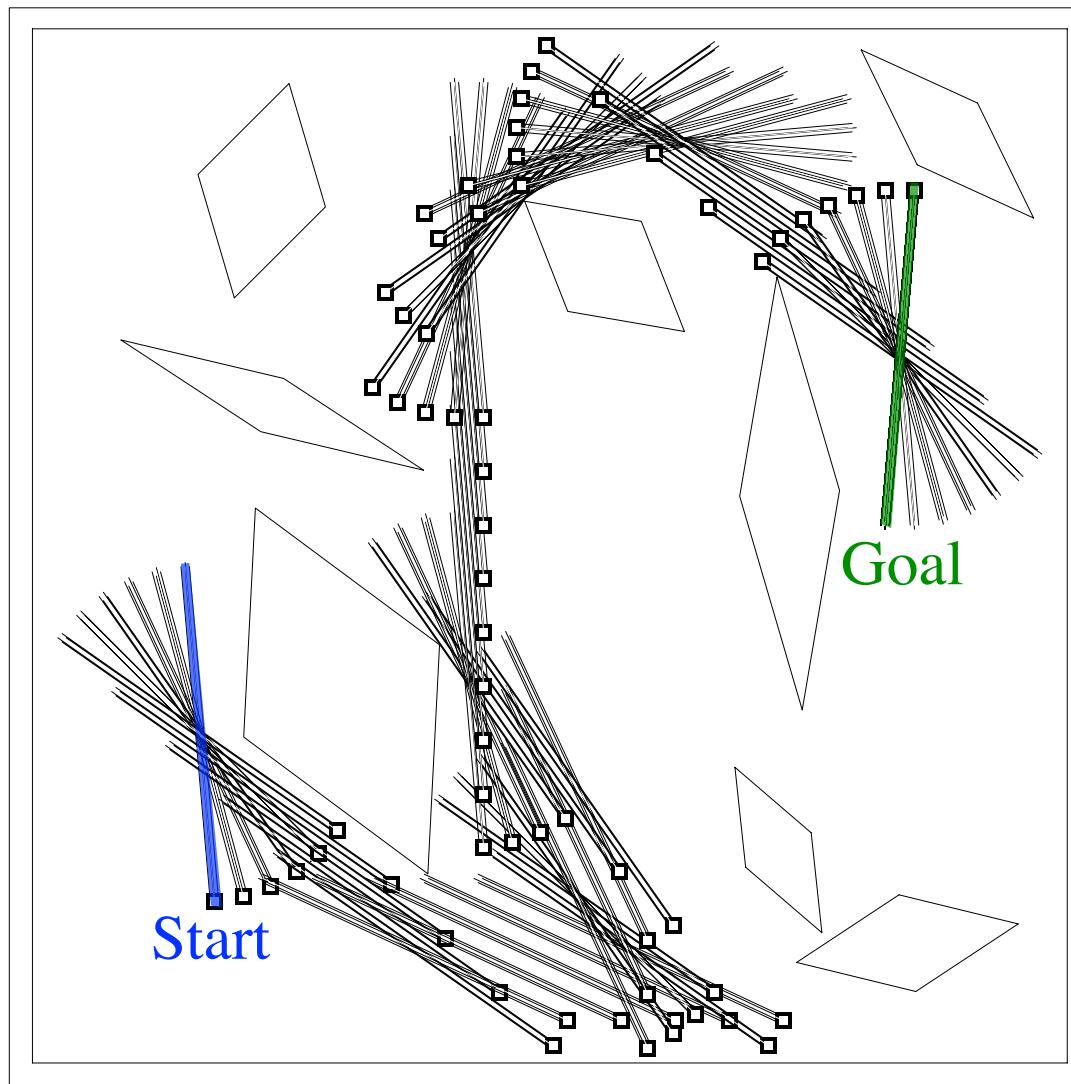


Backups
until
optimal
solution

Both use $n=5$ backups per environmental interaction



Rod Maneuvering (Moore and Atkeson 1993)



Improved Prioritized Sweeping with Small Backups

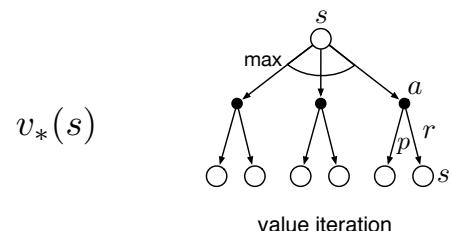
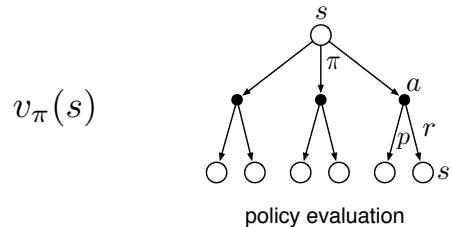
- Planning is a form of state-space search
 - a massive computation which we want to control to maximize its efficiency
- Prioritized sweeping is a form of search control
 - focusing the computation where it will do the most good
- But can we focus better?
- Can we focus more tightly?
- Small backups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

Expected and Sample Backups (One-Step)

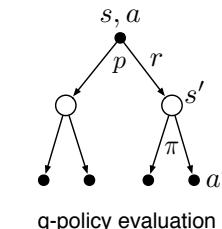
Value
estimated

Expected updates
(DP)

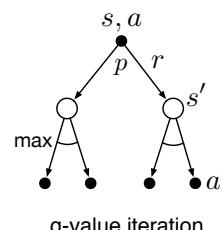
Sample updates
(one-step TD)



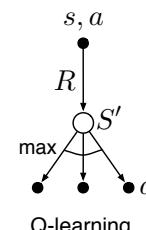
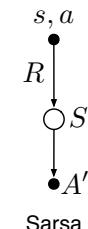
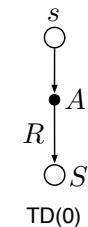
$q_\pi(s, a)$



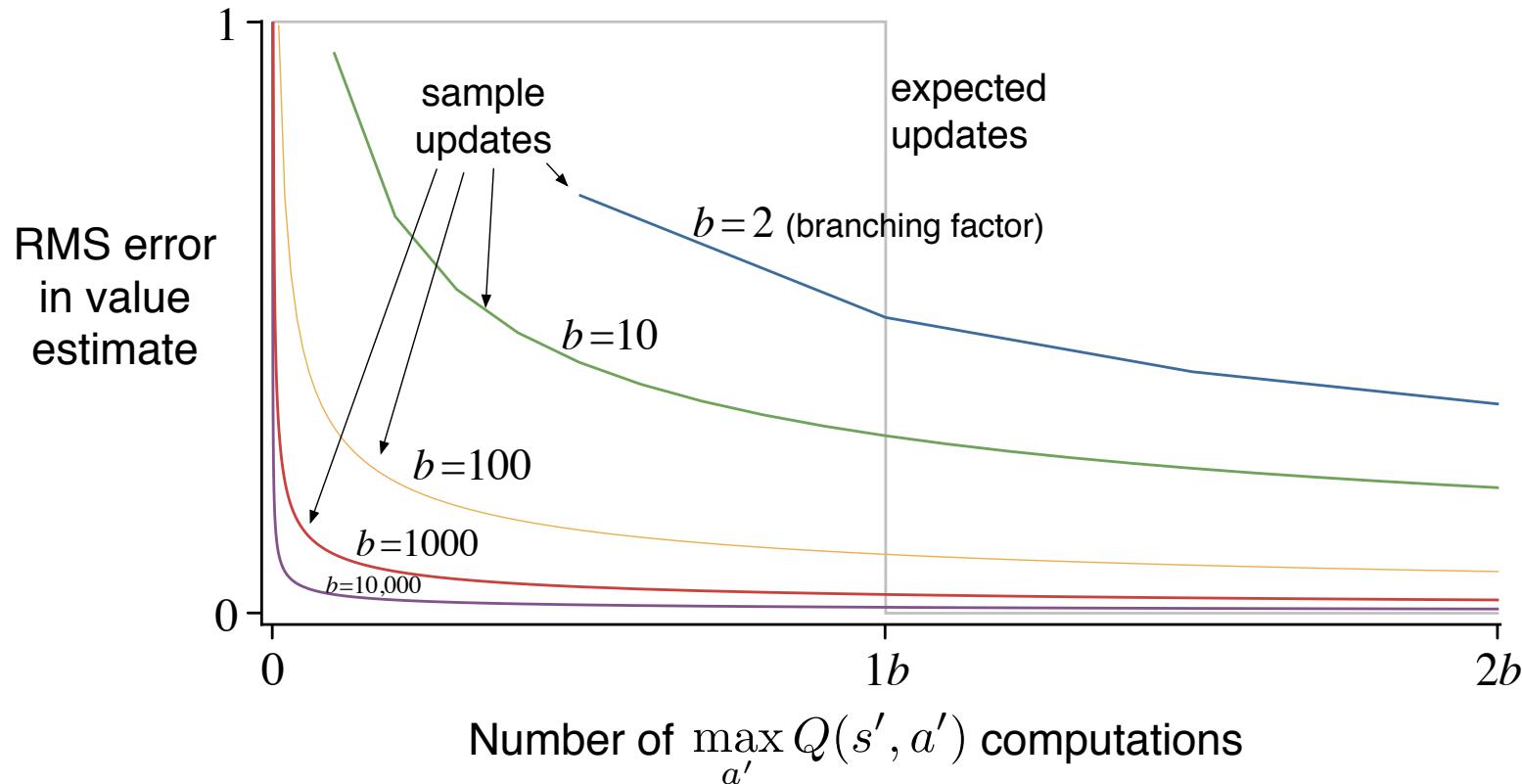
$q_*(s, a)$



Sample updates
(one-step TD)



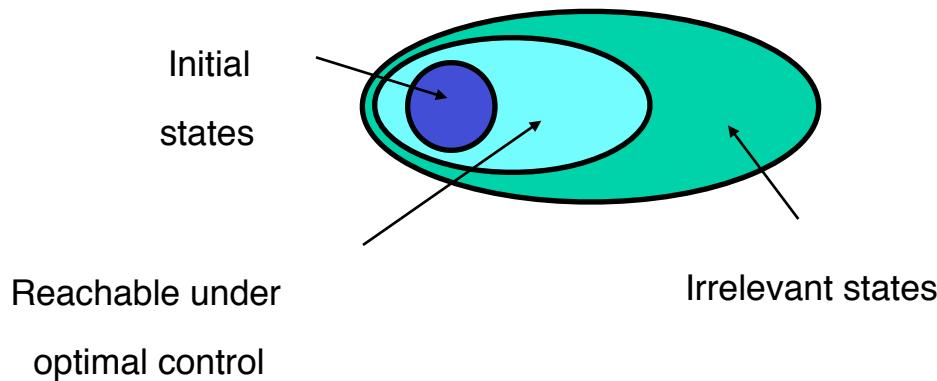
Full vs. Sample Backups



b successor states, equally likely; initial error = 1;
assume all next states' values are correct

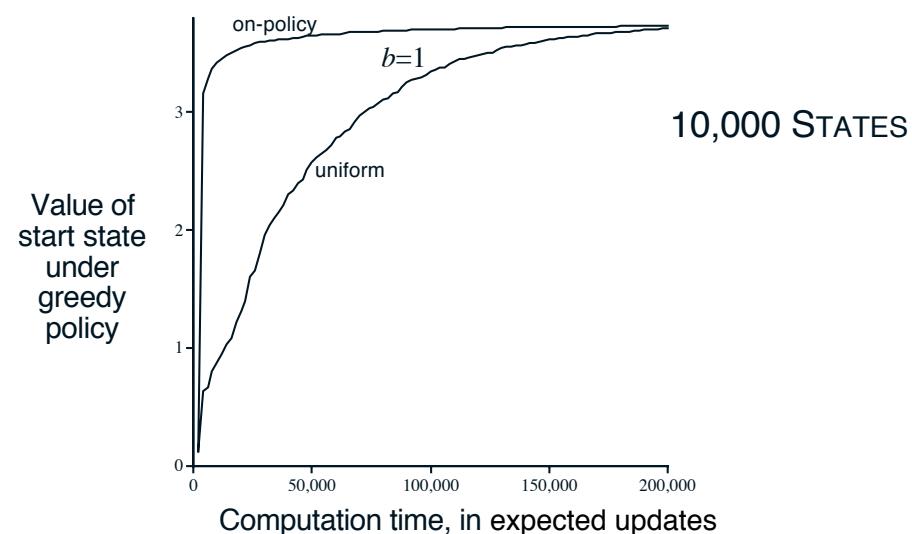
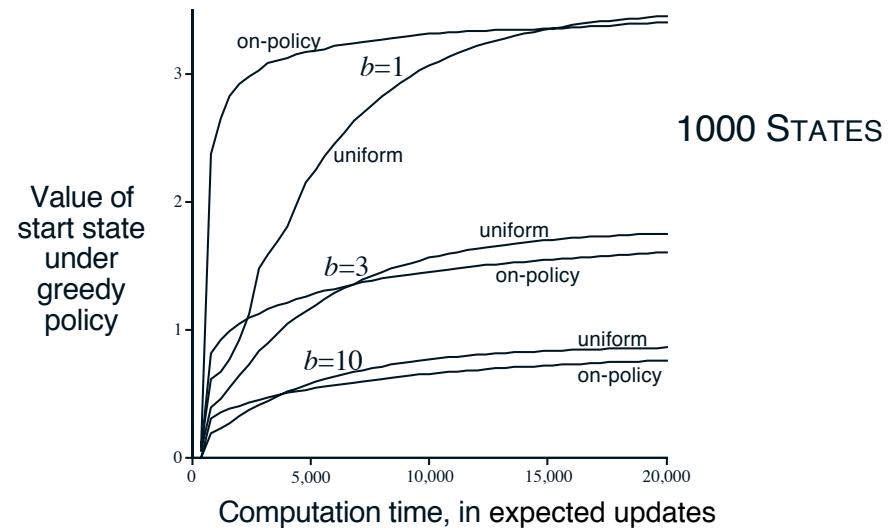
Trajectory Sampling

- **Trajectory sampling:** perform updates along simulated trajectories
- This samples from the on-policy distribution
- Advantages when function approximation is used (Part II)
- Focusing of computation:
can cause vast uninteresting parts of the state space to be ignored:



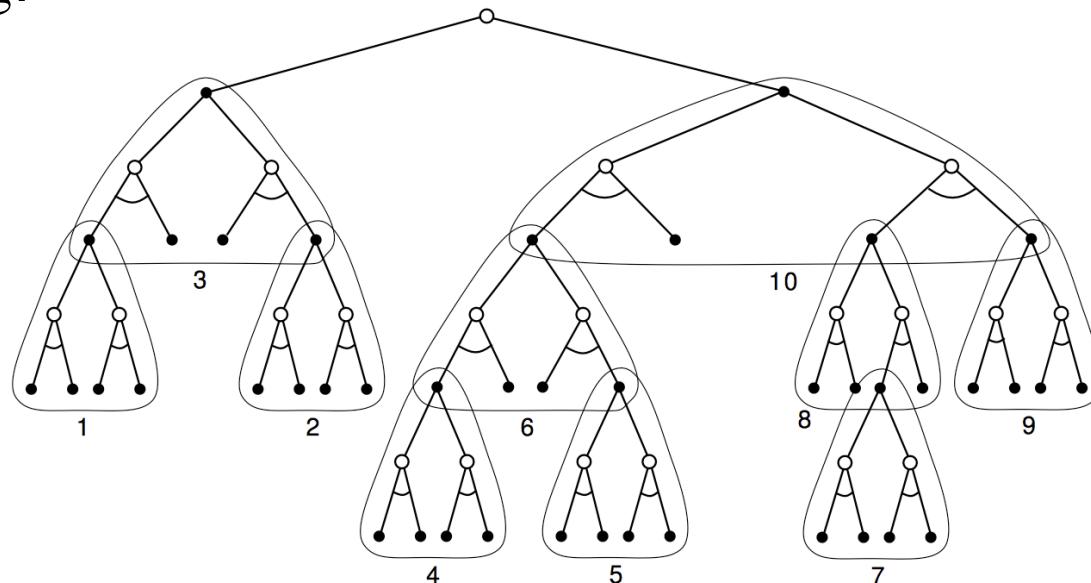
Trajectory Sampling Experiment

- one-step full tabular updates
- uniform: cycled through all state-action pairs
- on-policy: backed up along simulated trajectories
- 200 randomly generated undiscounted episodic tasks
- 2 actions for each state, each with b equally likely next states
- 0.1 prob of transition to terminal state
- expected reward on each transition selected from mean 0 variance 1 Gaussian



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 of Chapter 8

- 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; deep/shallow

Summary of Part I: Dimensions

- Core ideas common to all methods:
 - estimation of value functions
 - backing-up updates
- Backups can be
 - fat or skinny
(expected or sample)
 - short or tall (depth)
- Problem dimensions:
 - prediction vs control
 - action values vs state values
 - on-policy vs off-policy
 - episodic vs continuing
- Other method dimensions:
 - online vs offline

