



Reinforcement Learning

Planning and Learning with Tabular Methods

U Kang
Seoul National University



In This Lecture

- Unified view of planning and learning
- Model-based methods and model-free methods
- Planning at decision time



Overview

- Develop a unified view of RL methods that require a model of the environment, such as dynamic programming and heuristic search, and methods that can be used without a model, such as MC and TD methods
- These are respectively called model-based and model-free RL methods
- Model-based methods rely on planning as their primary component, while model-free methods primarily rely on learning
- Despite their differences, there are also great similarities in them; the heart of both kinds of methods is the computation of value functions
- Moreover, all the methods are based on looking ahead to future events, computing a backed-up value, and then using it as an update target for an approximate value function
- Integration of model-based and model-free methods



Outline

- ➔ **Models and Planning**
- Dyna: Integrated Planning, Acting, and Learning
- When the Model Is Wrong
- Prioritized Sweeping
- Expected vs. Sample Updates
- Trajectory Sampling
- Real-time DP
- Planning at Decision Time
- Heuristic Search
- Rollout Algorithms
- MC Tree Search
- Conclusion



Models and Planning

- Model of the environment: anything that an agent can use to predict how the environment will respond to its actions
- Given a state and an action, a model produces a prediction of the resultant next state and next reward
- Two types: distribution models and sample models
- Distribution models
 - Produce a description of all possibilities and their probabilities
 - E.g., in modeling the sum of dices, produce all possible sums and their probabilities of occurring
 - Used in DP
- Sample models
 - Produce just one of the possibilities, sampled according to the probabilities
 - E.g., in modeling the sum of dices, produce an individual sum drawn according to this probability distribution
 - Used in blackjack example of MC



Models and Planning

- Distribution models are stronger than sample models in that they can always be used to produce samples
- However, in many applications it is much easier to obtain sample models than distribution models
- E.g., dices
 - Sample models: it is easy to write a computer program to simulate the dice rolls and return the sum
 - Distribution models: it is harder and more error-prone to figure out all the possible sums and their probabilities



Models and Planning

- Models can be used to mimic or simulate experience
- Given a starting state and action, a sample model produces a possible transition, and a distribution model generates all possible transitions weighted by their probabilities of occurring
- Given a starting state and a policy, a sample model could produce an entire episode, and a distribution model could generate all possible episodes and their probabilities
- In either case, we say the model is used to simulate the environment and produce *simulated experience*



Models and Planning

- Planning: any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment

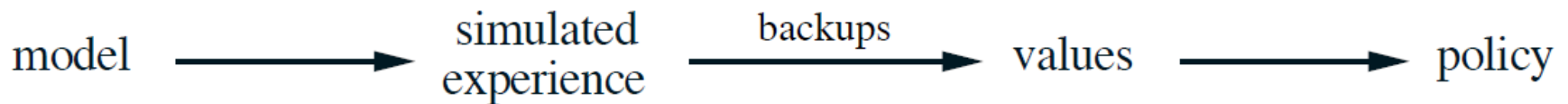


- In RL, we focus on state-space planning: search through the state space for an optimal policy or an optimal path to a goal; actions cause transitions from state to state, and value functions are computed over states



Models and Planning

- Unified view: all state-space planning methods share a common structure
- Main idea 1: they involve computing value functions as a key intermediate step toward improving the policy
- Main idea 2: they compute value functions by updates or backup operations applied to simulated experience



- E.g., DP
 - Make sweeps through the space of states, generating for each state the distribution of possible transitions
 - Each distribution is then used to compute a backed-up value (update target) and update the state's estimated value



Models and Planning

- The heart of both learning and planning methods is the estimation of value functions by backing-up update operations
 - Difference: planning uses simulated experience generated by a model, while learning methods use real experience generated by the environment
- This common structure means that many ideas and algorithms can be transferred between planning and learning
- In many cases a learning algorithm can be substituted for the key update step of a planning method; learning methods require only experience as input, and in many cases they can be applied to simulated experience just as well as to real experience



Models and Planning

- Random-sample one-step tabular Q-planning
 - Planning method based on one-step tabular Q-learning and on random samples from a sample model
 - Converges to the optimal policy for the model under the same conditions that one-step tabular Q-learning converges to the optimal policy for the real environment (each state-action pair must be selected an infinite number of times in Step 1, and α must decrease appropriately over time)

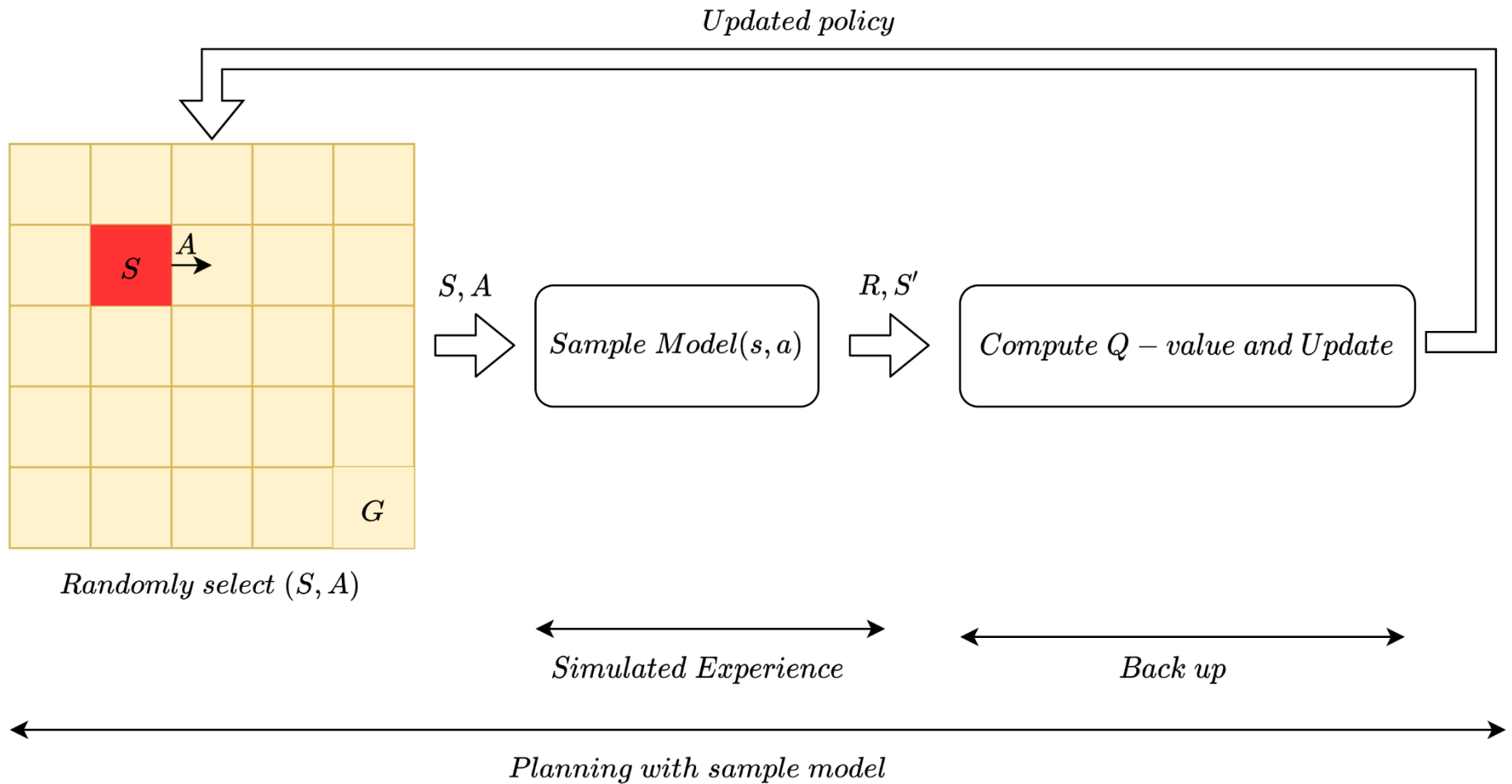
Random-sample one-step tabular Q-planning

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



Models and Planning






Models and Planning

- Planning in small, incremental steps
 - Enables planning to be interrupted or redirected at any time with little wasted computation, a key requirement for efficiently intermixing planning with acting and with learning of the model
 - Planning in very small steps may be the most efficient approach even on pure planning problems if the problem is too large to be solved exactly



Outline

- Models and Planning
-  **Dyna: Integrated Planning, Acting, and Learning**
- When the Model Is Wrong
- Prioritized Sweeping
- Expected vs. Sample Updates
- Trajectory Sampling
- Real-time DP
- Planning at Decision Time
- Heuristic Search
- Rollout Algorithms
- MC Tree Search
- Conclusion



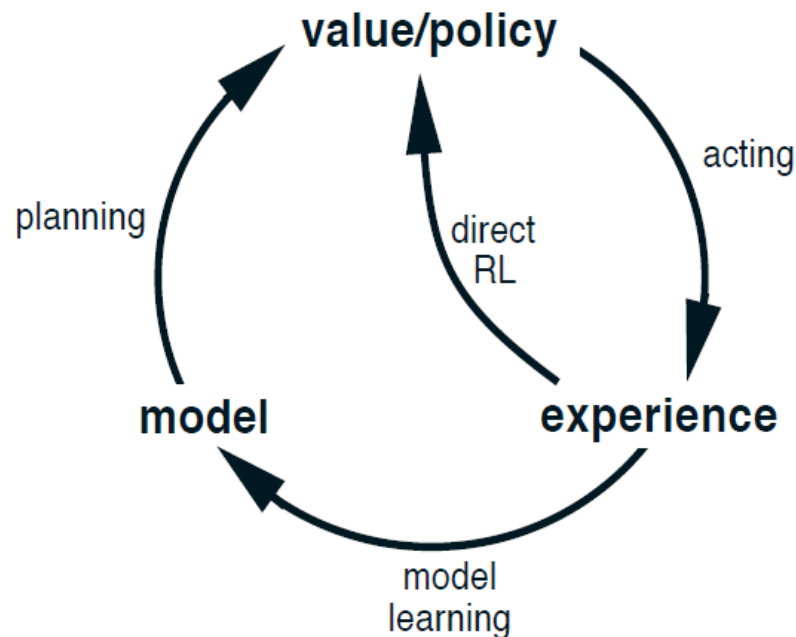
Dyna: Integrated Planning, Acting, and Learning

- When planning is done online, while interacting with the environment, a number of interesting issues arise
- New information gained from the interaction may change the model and thereby interact with planning; it may be desirable to customize the planning process to the states or decisions *currently under consideration, or expected in the near future*
- The available computational resources may need to be divided between decision making and model learning



Dyna: Integrated Planning, Acting, and Learning

- Two roles of real experience within a planning agent
 - Model learning: improve the model (to make it more accurately match the real environment)
 - Direct RL: directly improve the value function and policy
- Experience can improve value functions and policies either directly or indirectly via the model





Dyna: Integrated Planning, Acting, and Learning

- Both direct and indirect methods have advantages and disadvantages
- Indirect methods: often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions
- Direct methods: much simpler, and are not affected by biases in the design of the model
- Some have argued that indirect methods are always superior to direct ones, while others have argued that direct methods are responsible for most human and animal learning
- However, they have similarities as well: e.g., DP and TD are related, even though DP was designed for planning and TD was for model-free learning



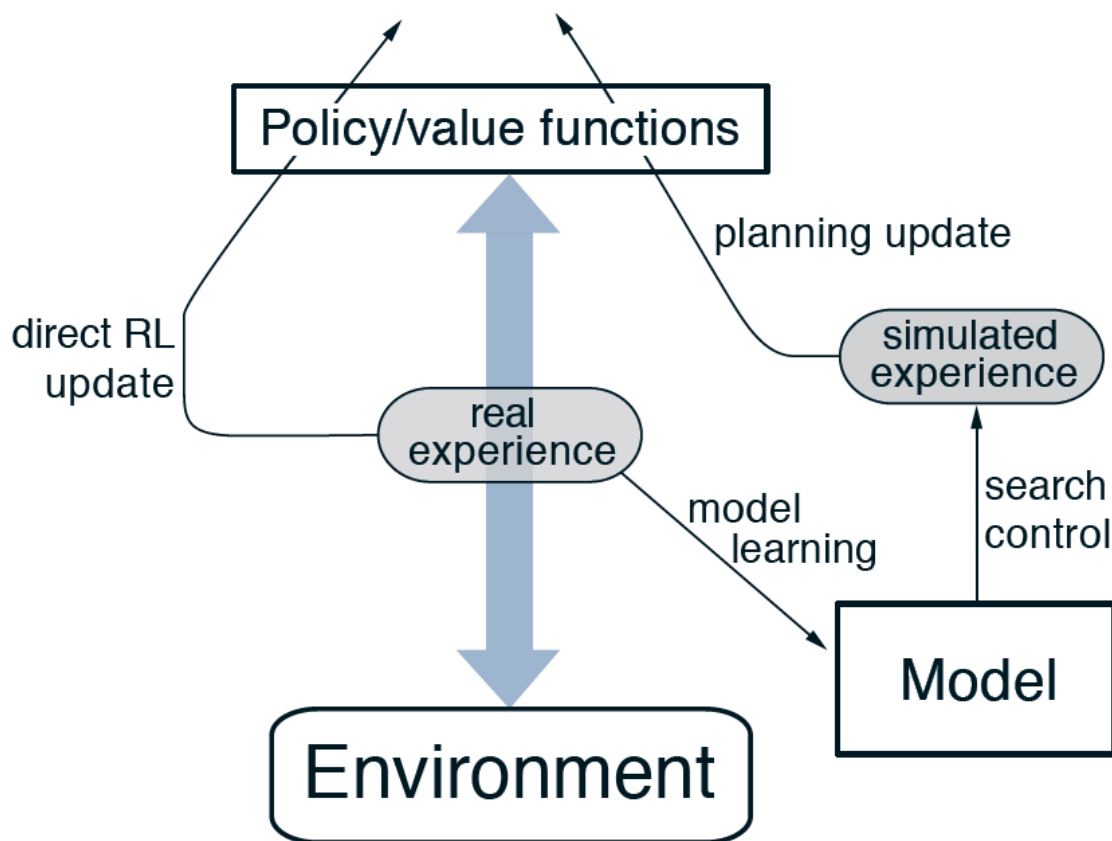
Dyna: Integrated Planning, Acting, and Learning

- Dyna-Q includes all of the processes: planning, acting, model-learning, and direct RL—all occurring continually
- Planning: random-sample one-step tabular Q-planning method
- Direct RL: one-step tabular Q-learning
- Model-learning: table-based, and assumes the environment is deterministic. After each transition $S_t, A_t \rightarrow R_{t+1}, S_{t+1}$, the model records in its table entry for S_t, A_t , the prediction R_{t+1}, S_{t+1} that will deterministically follow. Thus, if the model is queried with a state–action pair that has been experienced before, it simply returns the last-observed next state and next reward as its prediction
- During planning, the Q-planning algorithm randomly samples only from state–action pairs *that have previously been experienced*, so the model is never queried with a pair about which it has no information



Dyna: Integrated Planning, Acting, and Learning

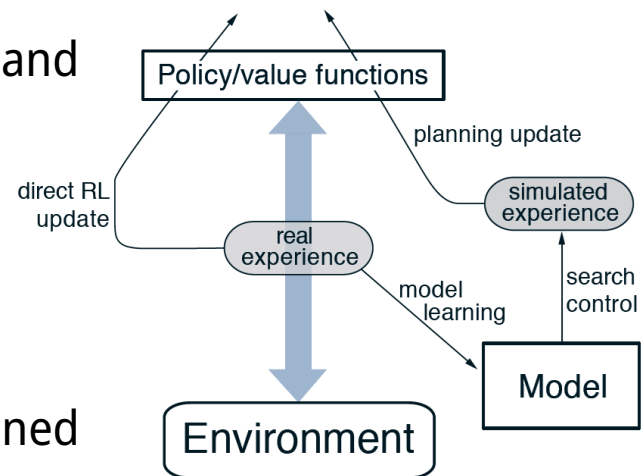
- Dyna architecture





Dyna: Integrated Planning, Acting, and Learning

- The central column represents the basic interaction between agent and environment
- The arrow on the left: direct RL on real experience to improve the value function and the policy
- Right side: the model is learned from real experience and gives rise to simulated experience
- Search control: the process that selects the starting states and actions for the simulated experiences generated by the model
- Planning is achieved by applying RL methods to the simulated experiences just as if they had really happened
- The same RL method is used both for learning from real experience and for planning from simulated experience
- Learning and planning are deeply integrated; they share almost all the same machinery, differing only in the source of their experience



Sutton and Barto,
Reinforcement
Learning, 2018



Dyna: Integrated Planning, Acting, and Learning

- In Dyna-Q, the acting, model-learning, and direct RL processes require little computation; the remaining time in each step can be devoted to the planning process, which is inherently computation-intensive
- In the pseudocode below, $Model(s, a)$ denotes the contents of the (predicted next state and reward) for state–action pair (s, a)
- Without (e) and (f), the algorithm would be one-step tabular Q-learning

Tabular Dyna-Q

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

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state

(b) $A \leftarrow \varepsilon$ -greedy(S, Q)

(c) Take 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) Loop repeat n times:

→ planning

$S \leftarrow$ random previously observed state

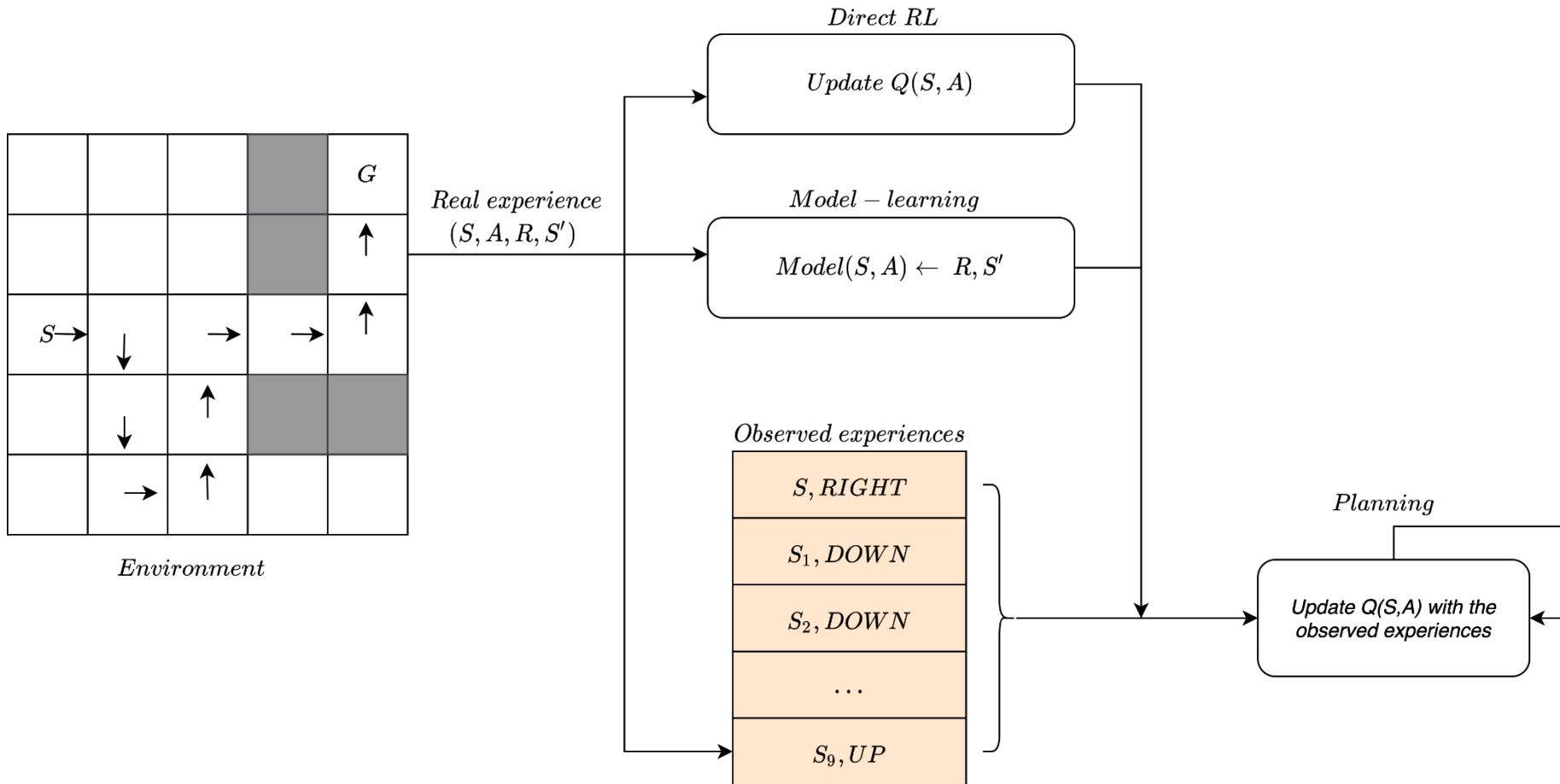
$A \leftarrow$ random action previously taken in S

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

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



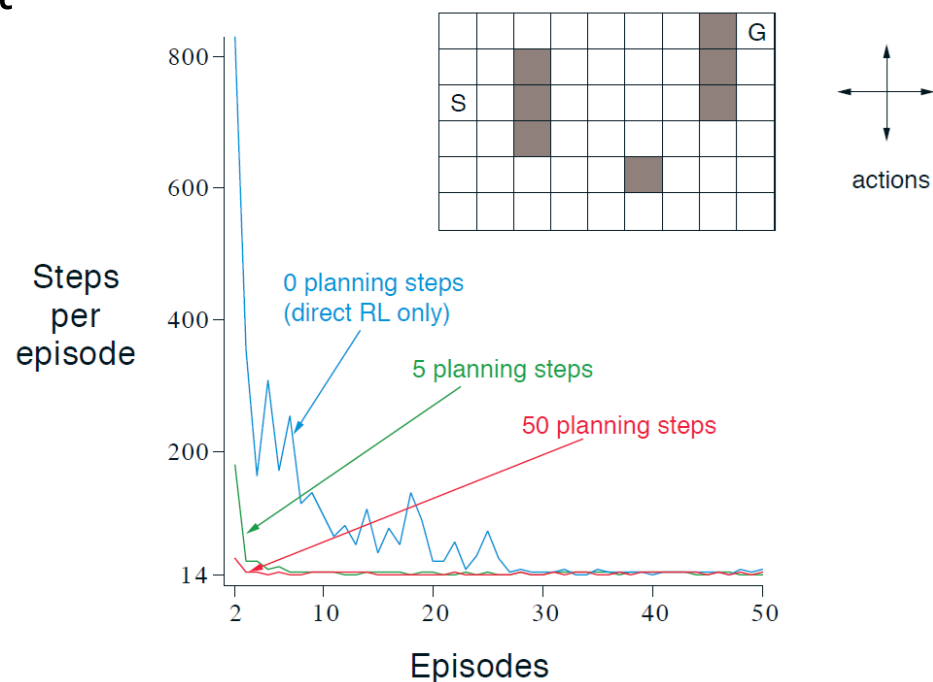
Example: Simple Dyna Maze





Example: Dyna Maze

- There are four actions, up, down, right, and left, which take the agent deterministically to the corresponding neighboring states, except when movement is blocked by an obstacle or the edge of the maze, in which case the agent remains where it is
- Reward is zero on all transitions, except those into the goal state, on which it is +1; after reaching the goal state (G), the agent returns to the start state (S) to begin a new episode

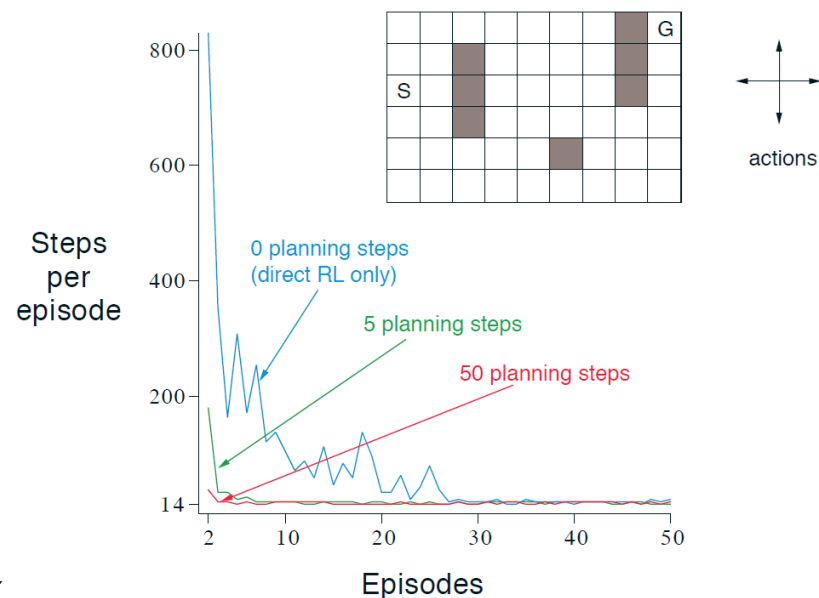


Sutton and Barto,
Reinforcement
Learning, 2018



Example: Dyna Maze

- The curves show the number of steps taken by the agent to reach the goal in each episode, averaged over 30 repetitions of the experiment
- After the first episode, performance improved for all values of n , but much more rapidly for larger values
- Recall that the $n = 0$ agent is a nonplanning agent, using only direct RL (one-step tabular Q-learning); this was by far the slowest agent on this problem, despite the fact that the parameter values (α and ϵ) were optimized for it. The nonplanning agent took about 25 episodes to reach (ϵ -)optimal performance, whereas the $n = 5$ agent took about five episodes, and the $n = 50$ agent took only three episodes



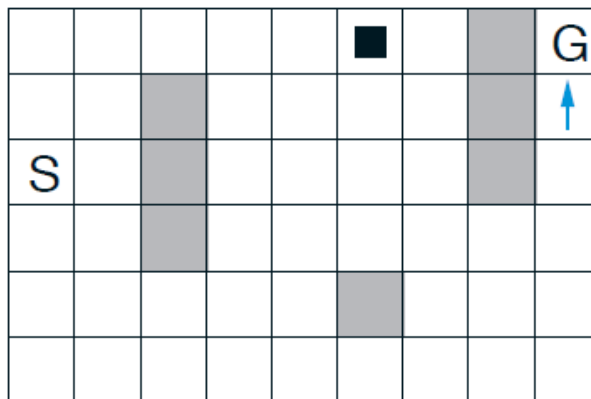
Sutton and Barto,
Reinforcement
Learning, 2018



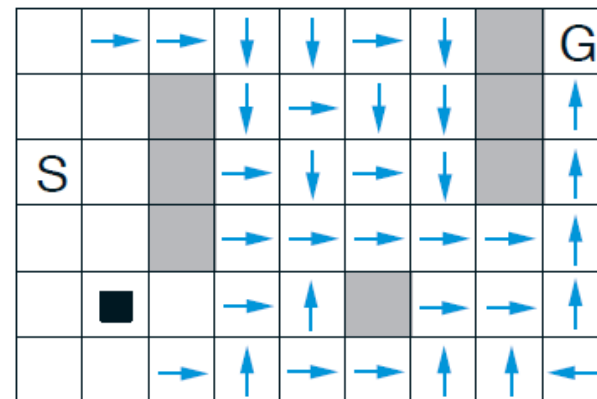
Example: Dyna Maze

- Why does the planning agents found the solution so much faster than the nonplanning agent?
- Without planning ($n = 0$), each episode adds only one additional step to the policy, and so only one step (the last) has been learned so far
- With planning, the agent learns for many steps during an episode

WITHOUT PLANNING ($n=0$)



WITH PLANNING ($n=50$)



Sutton and Barto,
Reinforcement
Learning, 2018




Dyna: Integrated Planning, Acting, and Learning

- In Dyna-Q, learning and planning are accomplished by exactly the same algorithm, operating on real experience for learning and on simulated experience for planning
- Both proceed as fast as they can. The agent is always reactive and always deliberative, responding instantly to the latest sensory information and yet always planning in the background
- Also ongoing in the background is the model-learning process. As new information is gained, the model is updated to better match reality. As the model changes, the ongoing planning process will gradually compute a different way of behaving to match the new model.



Outline

- Models and Planning
- Dyna: Integrated Planning, Acting, and Learning
-  **When the Model Is Wrong**
- Prioritized Sweeping
- Expected vs. Sample Updates
- Trajectory Sampling
- Real-time DP
- Planning at Decision Time
- Heuristic Search
- Rollout Algorithms
- MC Tree Search
- Conclusion

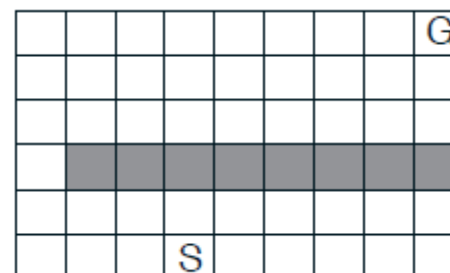
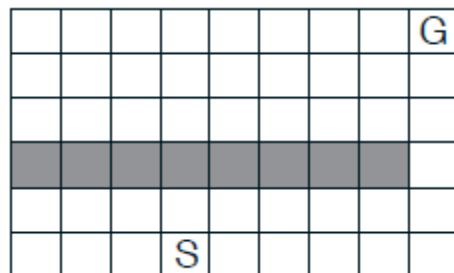


When the Model is Wrong

- Models may be incorrect
 - The environment is stochastic and only a limited number of samples have been observed
 - The model was learned using function approximation that has generalized imperfectly
 - The environment has changed and its new behavior has not yet been observed
- When the model is incorrect, the planning process is likely to compute a suboptimal policy
- In some cases, the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error



Example: Blocking Maze



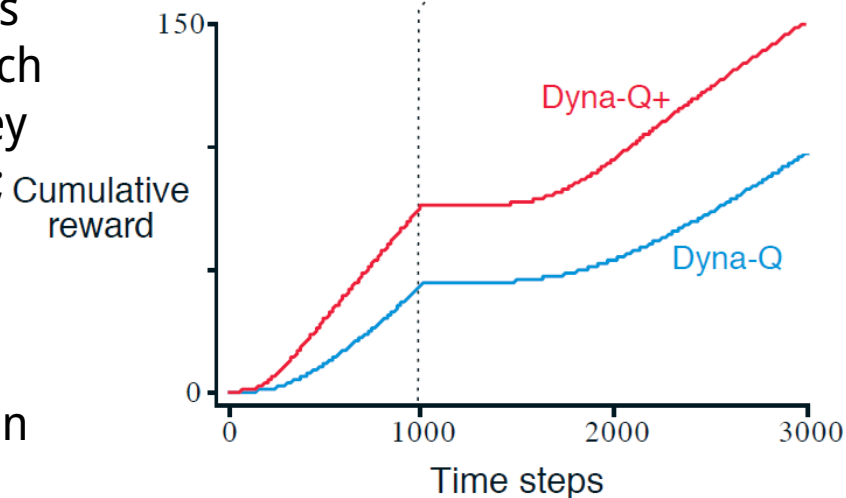
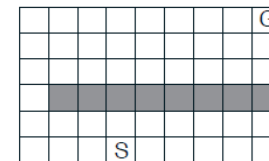
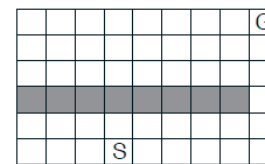
Sutton and Barto,
Reinforcement
Learning, 2018

- Initially, there is a short path from start to goal, to the right of the barrier
- After 1000 time steps, the short path is “blocked,” and a longer path is opened up along the left-hand side of the barrier



Example: Blocking Maze

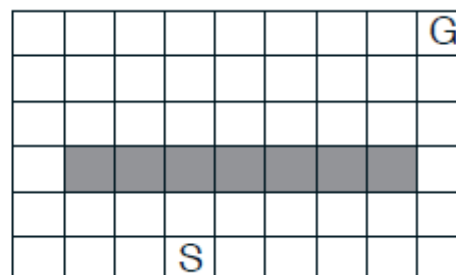
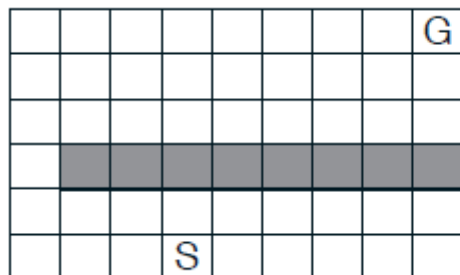
- Average cumulative reward for a Dyna-Q agent and an enhanced Dyna-Q+ agent
- The first part of the graph shows that both Dyna agents found the short path within 1000 steps
- When the environment changed, the graphs become flat, indicating a period during which the agents obtained no reward because they were wandering around behind the barrier; after a while, however, they found the new opening and the new optimal behavior
- Greater difficulties arise when the environment changes to become better than it was before, and yet the formerly correct policy does not reveal the improvement; in these cases the modeling error may not be detected for a long time, if ever



Sutton and Barto,
Reinforcement
Learning, 2018



Example: Shortcut Maze



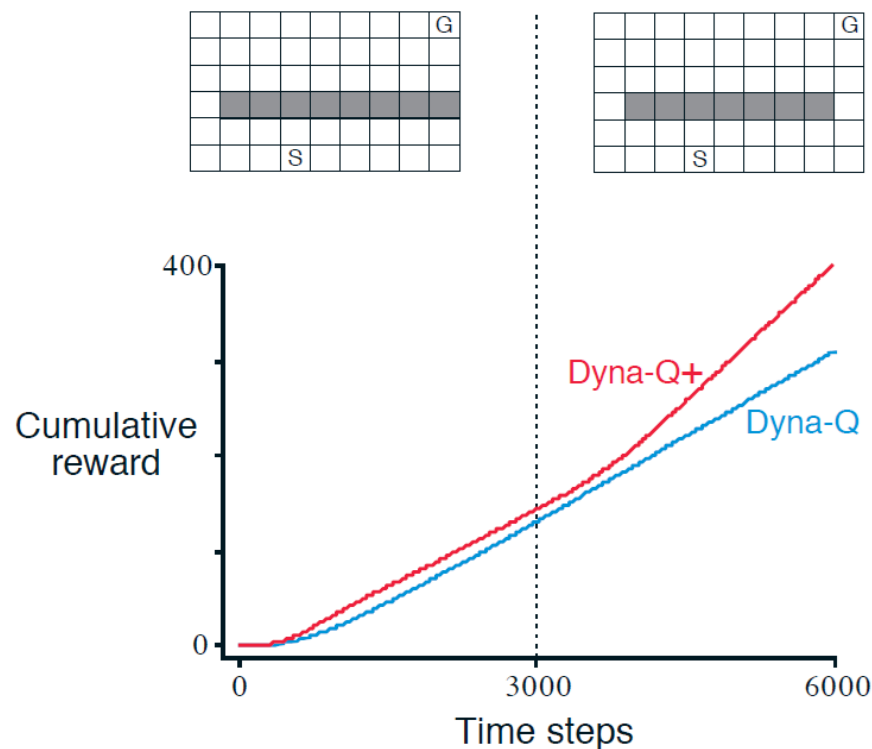
Sutton and Barto,
Reinforcement
Learning, 2018

- Initially, the optimal path is to go around the left side of the barrier (upper left)
- After 3000 steps, however, a shorter path is opened up along the right side, without disturbing the longer path (upper right)



Example: Shortcut Maze

- The regular Dyna-Q agent never switched to the shortcut; in fact, it never realized that it existed
- Its model said that there was no shortcut, so the more it planned, the less likely it was to step to the right and discover it
- Even with an ϵ -greedy policy, it is very unlikely that an agent will take so many exploratory actions as to discover the shortcut



Sutton and Barto,
Reinforcement
Learning, 2018



When the Model is Wrong

- The general problem here is another version of the conflict between exploration and exploitation
- In a planning context, exploration means trying actions that improve the model, whereas exploitation means behaving in the optimal way given the current model
- We want the agent to explore to find changes in the environment, but not so much that performance is greatly degraded
- There is no solution that is both perfect and practical, but simple heuristics are often effective; Dyna-Q+ agent that did solve the shortcut maze uses such heuristic

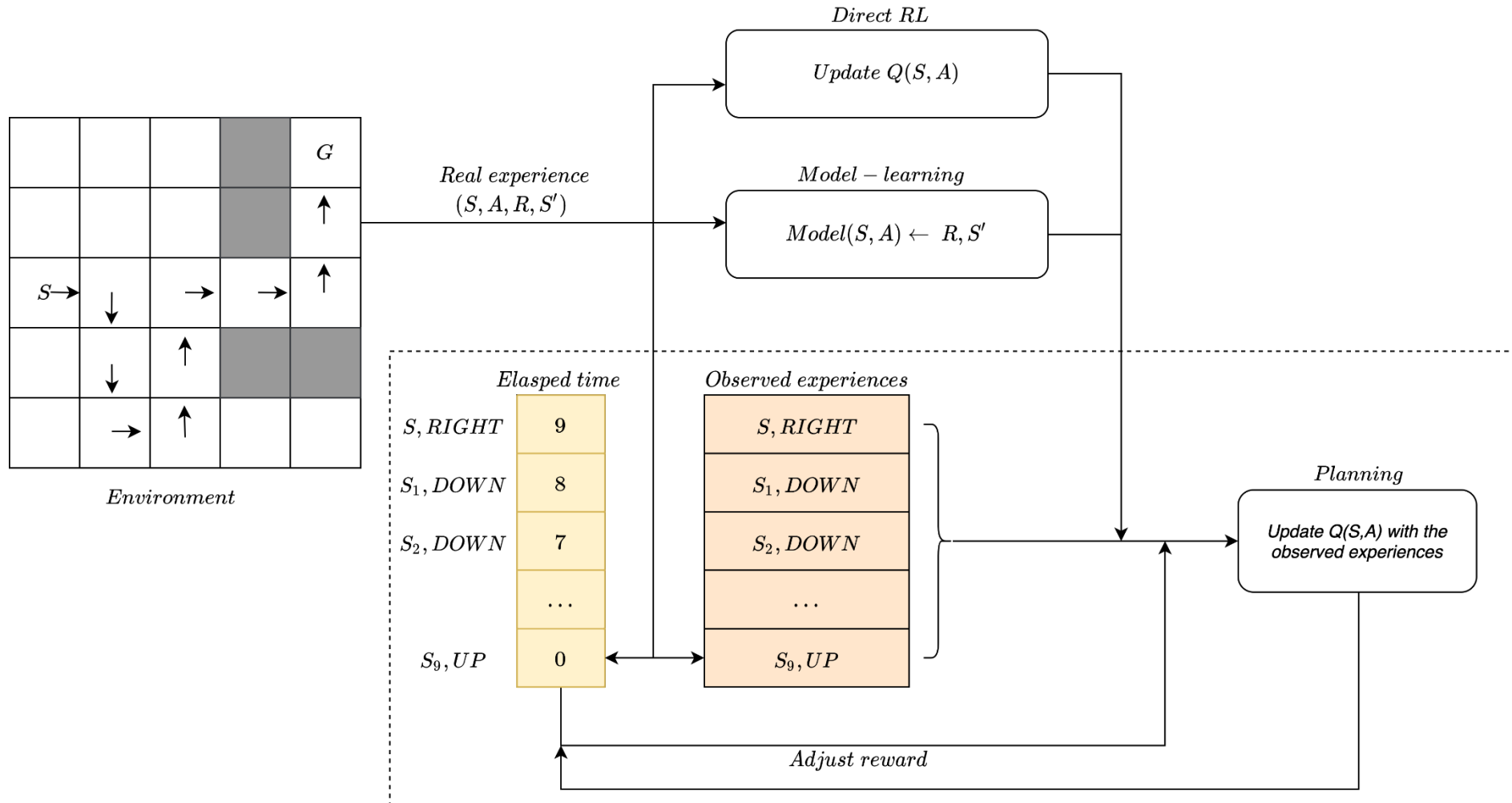


When the Model is Wrong

- Dyna-Q+ agent keeps track for each state–action pair of how many time steps have elapsed since the pair was last tried in a real interaction with the environment
- The more time that has elapsed, the greater (we might presume) the chance that the dynamics of this pair has changed and that the model of it is incorrect
- To encourage behavior that tests long-untried actions, a special “bonus reward” is given on simulated experiences involving these actions
- In particular, if the modeled reward for a transition is r , and the transition has not been tried in τ time steps, then planning updates are done as if that transition produced a reward of $r + k\sqrt{\tau}$, for some small k
- This encourages the agent to keep testing all accessible state transitions and even to find long sequences of actions in order to carry out such tests




When the Model is Wrong





Outline

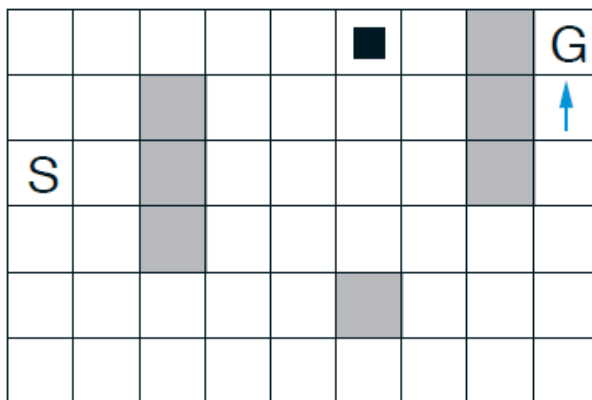
- Models and Planning
- Dyna: Integrated Planning, Acting, and Learning
- When the Model Is Wrong
-  **Prioritized Sweeping**
- Expected vs. Sample Updates
- Trajectory Sampling
- Real-time DP
- Planning at Decision Time
- Heuristic Search
- Rollout Algorithms
- MC Tree Search
- Conclusion



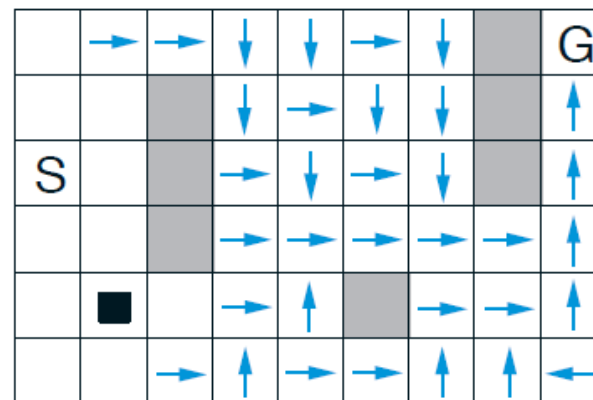
Prioritized Sweeping

- In the Dyna agents presented before, simulated transitions are started in state–action pairs selected uniformly at random from all previously experienced pairs
- Planning can be much more efficient if simulated transitions and updates are focused on particular state–action pairs
- E.g., second episode of the maze task

WITHOUT PLANNING ($n=0$)



WITH PLANNING ($n=50$)



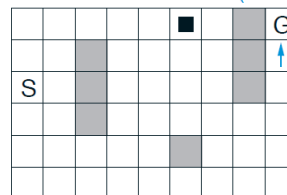
Sutton and Barto,
Reinforcement
Learning, 2018



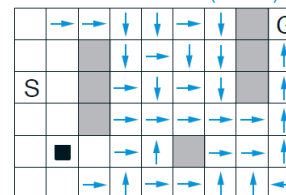
Prioritized Sweeping

- At the beginning of the second episode, only the state–action pair leading directly into the goal has a positive value; the values of all other pairs are still zero
- It is pointless to perform updates along almost all transitions, because they take the agent from one zero-valued state to another, and thus the updates would have no effect
- Only an update along a transition into the state just prior to the goal, or from it, will change any values
- If simulated transitions are generated uniformly, then many wasteful updates will be made before stumbling onto one of these useful ones
- As planning progresses, the region of useful updates grows, but planning is still far less efficient than it would be if focused where it would do the most good
- In the much larger problems that are our real objective, the number of states is so large that an unfocused search would be extremely inefficient

WITHOUT PLANNING ($n=0$)



WITH PLANNING ($n=50$)





Prioritized Sweeping

- *Backward focusing* of planning computations
 - Search might be usefully focused by working backward from goal states
 - Of course, we do not really want to use any methods specific to the idea of “goal state”. We want methods that work for general reward functions: work back not just from goal states but from any state whose value has changed
 - Suppose that the values are initially correct given the model
 - Suppose now that the agent discovers a change in the environment and changes its estimated value of one state, either up or down
 - This will imply that the values of many other states should also be changed, but the only useful one-step updates are those of actions that lead directly into the one state whose value has been changed
 - If the values of these actions are updated, then the values of the predecessor states may change in turn. If so, then actions leading into them need to be updated, and then their predecessor states may have changed. In this way one can work backward from arbitrary states that have changed in value, either performing useful updates or terminating the propagation



Prioritized Sweeping

■ *Prioritized Sweeping*

- As the frontier of useful updates propagates backward, it often grows rapidly, producing many state–action pairs that could usefully be updated
- But not all of these will be equally useful. The values of some states may have changed a lot, whereas others may have changed little
- The predecessor pairs of those that have changed a lot are more likely to also change a lot
- In a stochastic environment, variations in estimated transition probabilities also contribute to variations in the sizes of changes and in the urgency with which pairs need to be updated. It is natural to prioritize the updates according to a measure of their urgency, and perform them in order of priority
- A queue is maintained of every state–action pair whose estimated value would change nontrivially if updated, prioritized by the size of the change
- When the top pair in the queue is updated, the effect on each of its predecessor pairs is computed; if the effect is greater than some small threshold, then the pair is inserted in the queue with the new priority
- In this way the effects of changes are efficiently propagated backward until quiescence



Prioritized Sweeping

Prioritized sweeping for a deterministic environment

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

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state

(b) $A \leftarrow policy(S, Q)$

(c) Take 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) Loop repeat n times, while $PQueue$ is not empty:

$S, A \leftarrow first(PQueue)$

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

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

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

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

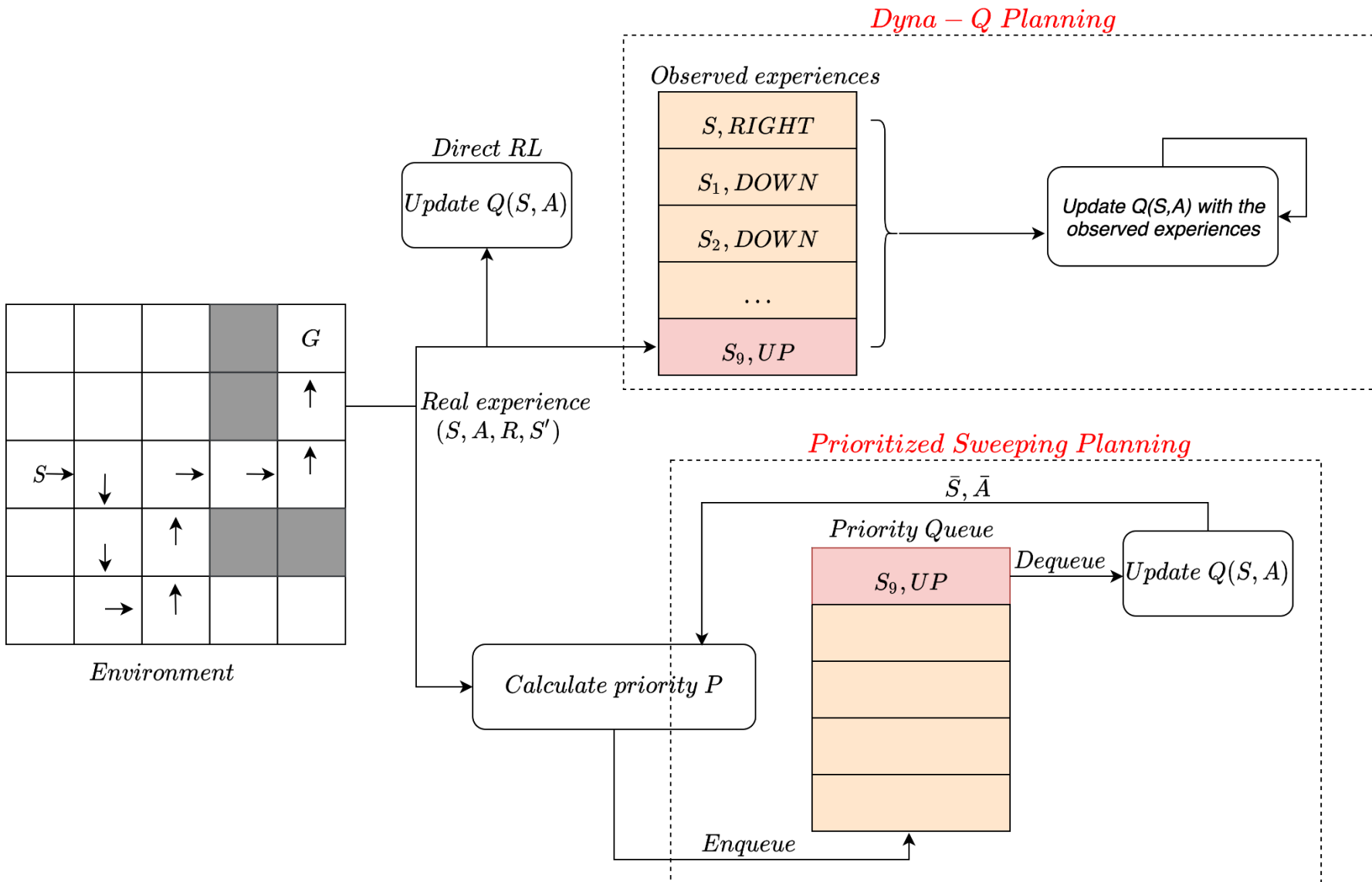
$P \leftarrow |\bar{R} + \gamma \max_a Q(S, a) - Q(\bar{S}, \bar{A})|$.

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

Sutton and Barto,
Reinforcement
Learning, 2018



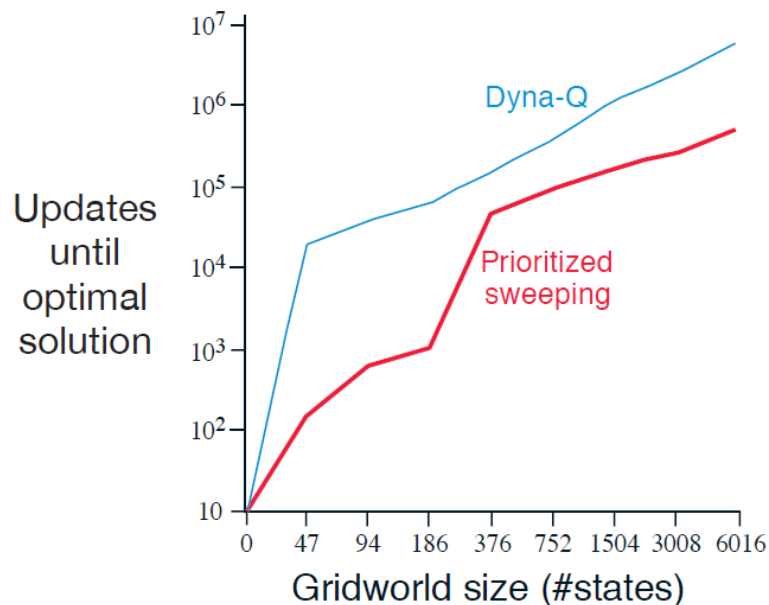
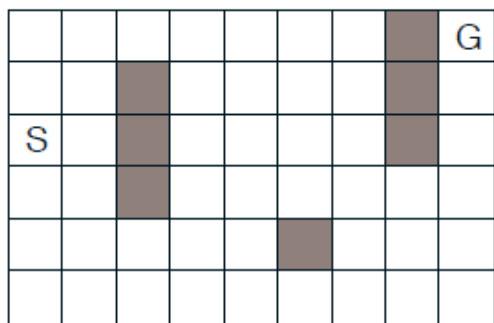
Prioritized Sweeping





Example: Prioritized Sweeping on Mazes

- Prioritized sweeping dramatically increases the speed at which optimal solutions are found in maze tasks, often by a factor of 5 to 10
- Prioritized sweeping maintained a decisive advantage over unprioritized Dyna-Q

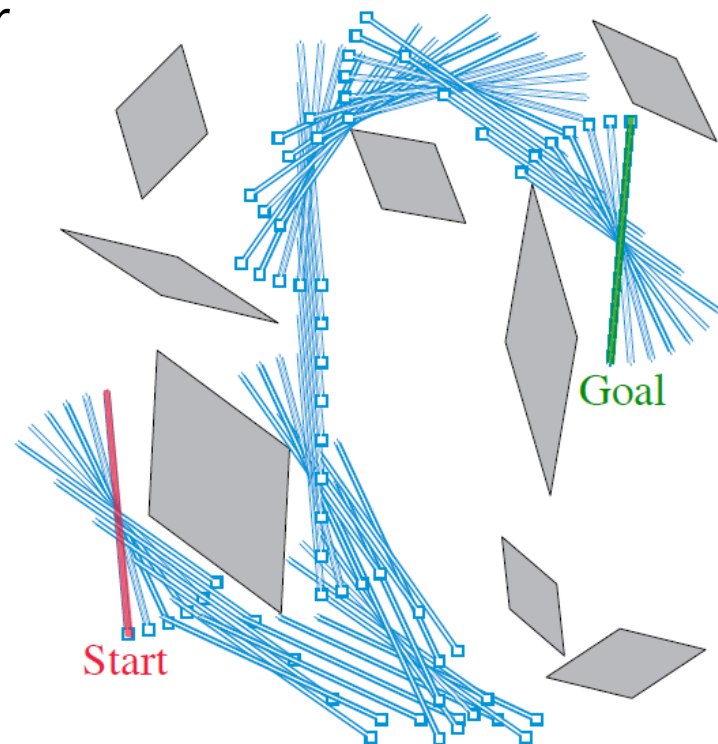


Sutton and Barto,
Reinforcement
Learning, 2018



Example: Prioritized Sweeping for Rod Maneuvering


- Maneuver a rod around some awkwardly placed obstacles within a limited rectangular work space to a goal position in the fewest number of steps
- The rod can be translated along its long axis or perpendicular to that axis, or it can be rotated in either direction around its center
- This problem has four actions and 14,400 potential states (some of these are unreachable because of the obstacles); This problem is probably too large to be solved with unprioritized methods



Sutton and Barto,
Reinforcement
Learning, 2018



Outline

- Models and Planning
- Dyna: Integrated Planning, Acting, and Learning
- When the Model Is Wrong
- Prioritized Sweeping
-  **Expected vs. Sample Updates**
- Trajectory Sampling
- Real-time DP
- Planning at Decision Time
- Heuristic Search
- Rollout Algorithms
- MC Tree Search
- Conclusion



Expected vs. Sample Updates

- We have discussed different kinds of value-function updates
- One-step updates: vary primarily along three binary dimensions
- Dim 1: whether they update state values or action values
- Dim 2: whether they estimate the value for the optimal policy or for an arbitrary given policy
- These two dimensions give rise to four classes of updates for approximating the four value functions, q_* , v_* , q_π , v_π
- Dim 3: whether the updates are expected updates, considering all possible events that might happen, or sample updates, considering a single sample of what might happen



Expected vs. Sample Updates

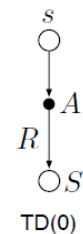
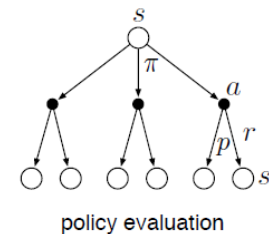
- These three binary dimensions give rise to eight cases, seven of which correspond to specific algorithms
- Any of these one-step updates can be used in planning methods

Value
estimated

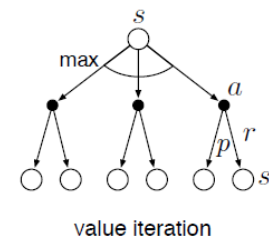
Expected updates
(DP)

Sample updates
(one-step TD)

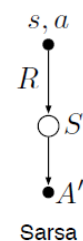
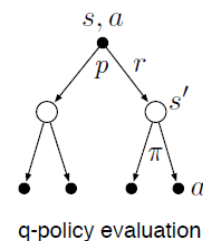
$v_{\pi}(s)$



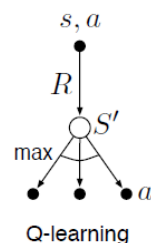
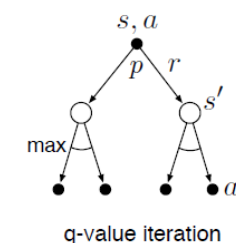
$v_{*}(s)$



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



$q_{*}(s, a)$





Expected vs. Sample Updates

- In the absence of a distribution model, expected updates are not possible, but sample updates can be done using sample transitions from the environment or a sample model
- Are expected updates preferable to sample updates?
- Expected updates certainly yield a better estimate because they are uncorrupted by sampling error, but they also require more computation, and computation is often the limiting resource in planning
- To properly assess the relative merits of expected and sample updates for planning we must control for their different computational requirements



Expected vs. Sample Updates

- Consider the expected and sample updates for approximating q_* , and the special case of discrete states and actions, a table-lookup representation of the approximate value function Q , and a model in the form of estimated dynamics, $\hat{p}(s', r|s, a)$

- The expected update for a state–action pair, s, a , is:

$$Q(s, a) \leftarrow \sum_{s', r} \hat{p}(s', r|s, a) [r + \gamma \max_{a'} Q(s', a')]$$

- The corresponding sample update for s, a , given a sample next state S' and reward R (from the model), is the Q-learning-like update:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [R + \gamma \max_{a'} Q(S', a') - Q(s, a)]$$

where α is the usual positive step-size parameter



Expected vs. Sample Updates

- If only one next state is possible, then the expected and sample updates given above are identical (taking $\alpha=1$)
- For many possible next states, there may be significant differences
- In favor of the expected update is that it is an exact computation, resulting in a new $Q(s, a)$ whose correctness is limited only by the correctness of the $Q(s', a')$ at successor states. The sample update is in addition affected by sampling error
- On the other hand, the sample update is cheaper computationally because it considers only one next state, not all possible next states. In practice, the computation required by update operations is usually dominated by the number of state–action pairs at which Q is evaluated



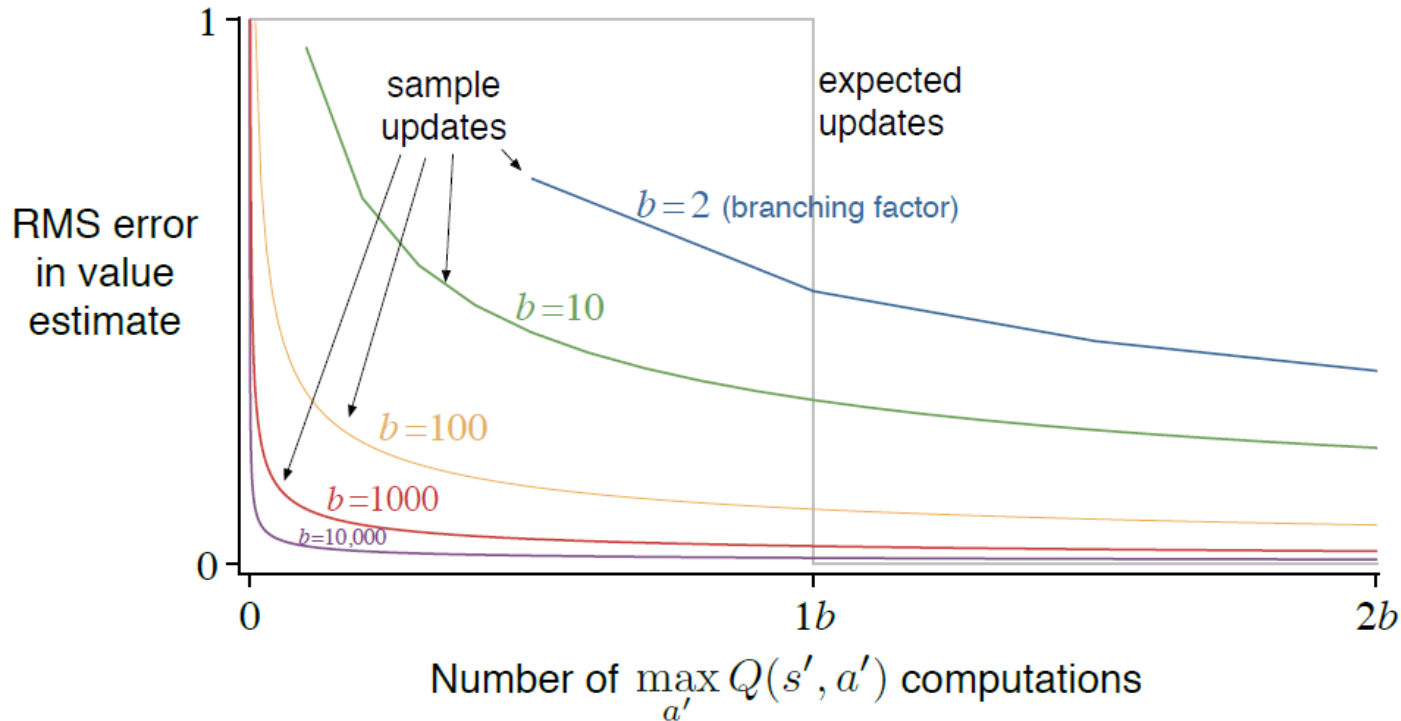
Expected vs. Sample Updates

- For a particular starting pair, s, a , let b be the branching factor (i.e., the number of possible next states, s' , for which $\hat{p}(s'|s, a) > 0$). Then an expected update of this pair requires roughly b times as much computation as a sample update
- If there is enough time to complete an expected update, then the resulting estimate is generally better than that of b sample updates because of the absence of sampling error
- But if there is insufficient time to complete an expected update, then sample updates are always preferable because they at least make some improvement in the value estimate with fewer than b updates



Expected vs. Sample Updates

- Given a unit of computational effort, is it better devoted to a few expected updates or to b times as many sample updates?



Sutton and Barto,
Reinforcement
Learning, 2018

- For moderately large b the error falls dramatically with a tiny fraction of b updates




Expected vs. Sample Updates

- In a real problem, the values of the successor states would be estimates that are themselves updated
- By causing estimates to be more accurate sooner, sample updates will have a second advantage in that the values backed up from the successor states will be more accurate
- These results suggest that sample updates are likely to be superior to expected updates on problems with large stochastic branching factors and too many states to be solved exactly



Outline

- Models and Planning
- Dyna: Integrated Planning, Acting, and Learning
- When the Model Is Wrong
- Prioritized Sweeping
- Expected vs. Sample Updates
-  **Trajectory Sampling**
- Real-time DP
- Planning at Decision Time
- Heuristic Search
- Rollout Algorithms
- MC Tree Search
- Conclusion



Trajectory Sampling

- Compare two ways of distributing updates
- Approach 1: exhaustive sweeps (from DP)
 - Perform sweeps through the entire state (or state–action) space, updating each state (or state–action pair) once per sweep
 - This is problematic on large tasks because there may not be time to complete even one sweep
 - In many tasks the vast majority of the states are irrelevant because they are visited only under very poor policies or with very low probability
 - Exhaustive sweeps implicitly devote equal time to all parts of the state space rather than focusing where it is needed; but it is not necessary to do that
 - In principle, updates can be distributed in any way one likes (to assure convergence, all states or state–action pairs must be visited in the limit an infinite number of times)



Trajectory Sampling

- Approach 2: trajectory sampling
 - Sample from the state or state–action space according to some distribution
 - One could sample uniformly, as in the Dyna-Q agent, but this would suffer from some of the same problems as exhaustive sweeps
 - More appealing is to distribute updates according to the on-policy distribution, that is, according to the distribution observed when following the current policy
 - One advantage of this distribution is that it is easily generated; one simply interacts with the model, following the current policy
 - In an episodic task, one starts in a start state (or according to the starting-state distribution) and simulates until the terminal state; in a continuing task, one starts anywhere and just keeps simulating. In either case, sample state transitions and rewards are given by the model, and sample actions are given by the current policy
 - Trajectory sampling simulates explicit individual trajectories and performs updates at the state or state-action pairs encountered along the way



Trajectory Sampling

- Intuition of on-policy distribution of updates
 - If you are learning to play chess, you study positions that might arise in real games, not random positions of chess pieces
 - One might expect on-policy focusing to significantly improve the speed of planning

- But, what are the experimental results?
 - Focusing on the on-policy distribution could be beneficial because it causes vast, uninteresting parts of the space to be ignored, or it could be detrimental because it causes the same old parts of the space to be updated over and over



Trajectory Sampling

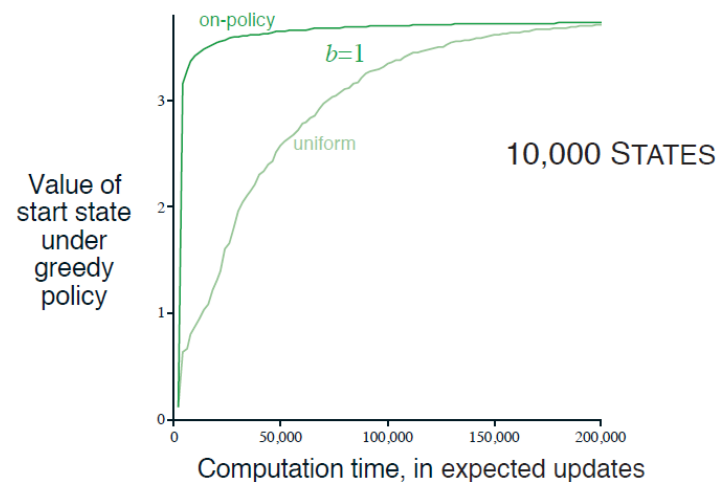
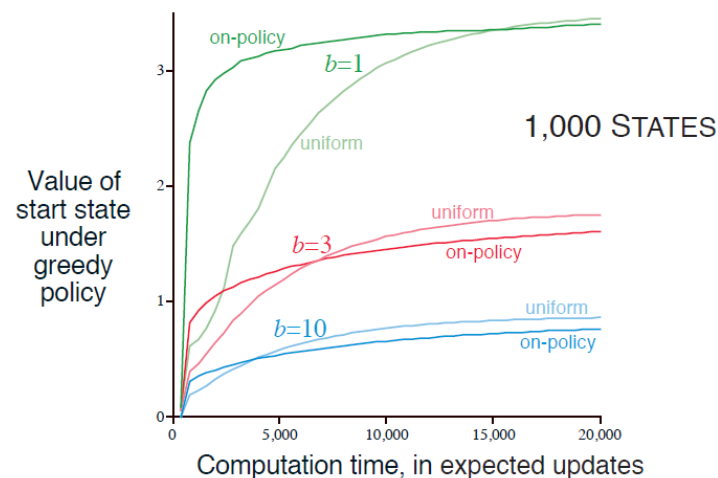
- Experiment to assess the effect of on-policy distribution
 - Use entirely one-step expected tabular updates
 - Uniform case: cycle through all state–action pairs, updating each in place
 - On-policy case: simulate episodes, all starting in the same state, updating each state–action pair that occurred under the current ϵ -greedy policy ($\epsilon=0.1$)
 - The tasks were undiscounted episodic tasks, generated randomly
 - From each of the $|S|$ states, two actions were possible, each of which resulted in one of b next states, all equally likely, with a different random selection of b states for each state–action pair
 - The branching factor, b , was the same for all state–action pairs
 - On all transitions there was a 0.1 probability of transition to the terminal state, ending the episode. The expected reward on each transition was selected from a Gaussian distribution with mean 0 and variance 1



Trajectory Sampling

Sutton and Barto,
Reinforcement
Learning, 2018

- Top figure: results averaged over 200 sample tasks with 1000 states and branching factors of 1, 3, and 10
 - In all cases, sampling according to the on-policy distribution resulted in faster planning initially and retarded planning in the long run
 - The effect was stronger, and the initial period of faster planning was longer, at smaller branching factors
- Bottom figure: results for a branching factor of 1 for tasks with 10,000 states
 - The advantage of on-policy focusing is large and long-lasting
 - The effects are stronger as the number of states increases






Trajectory Sampling

- In the short term, sampling according to the on-policy distribution helps by focusing on states that are near descendants of the start state
- If there are many states and a small branching factor, this effect will be large and long-lasting
- In the long run, focusing on the on-policy distribution may hurt because the commonly occurring states all already have their correct values; sampling them is useless, whereas sampling other states may actually perform some useful work
- This presumably is why the exhaustive, unfocused approach does better in the long run, at least for small problems
- These results are not conclusive because they are only for problems generated in a particular, random way, but they do suggest that sampling according to the on-policy distribution can be a great advantage for large problems, in particular for problems in which a small subset of the state–action space is visited under the on-policy distribution



Outline

- Models and Planning
- Dyna: Integrated Planning, Acting, and Learning
- When the Model Is Wrong
- Prioritized Sweeping
- Expected vs. Sample Updates
- Trajectory Sampling
-  **Real-time DP**
- Planning at Decision Time
- Heuristic Search
- Rollout Algorithms
- MC Tree Search
- Conclusion



Real-time DP

- Real-time dynamic programming (RTDP): an on-policy trajectory-sampling version of the value-iteration algorithm of dynamic programming (DP)
- Because it is closely related to conventional sweep-based policy iteration, RTDP illustrates in a particularly clear way some of the advantages that on-policy trajectory sampling can provide
- RTDP updates the values of states visited in actual or simulated trajectories by means of expected tabular value-iteration updates

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



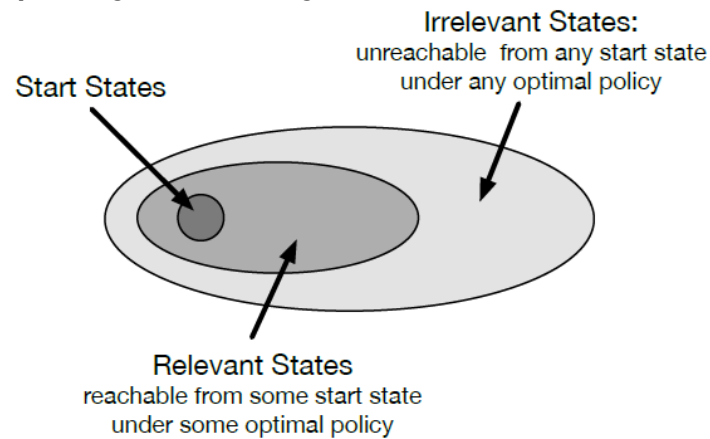
Real-time DP

- RTDP is an example of an asynchronous DP algorithm
- Asynchronous DP algorithms are not organized in terms of systematic sweeps of the state set; they update state values in any order whatsoever, using whatever values of other states happen to be available
- In RTDP, the update order is dictated by the order states are visited in real or simulated trajectories



Real-time DP

- If trajectories can start only from a designated set of start states, and if you are interested in the prediction problem for a given policy, then on-policy trajectory sampling allows the algorithm to completely skip states that cannot be reached by the given policy from any of the start states: such states are irrelevant to the prediction problem
- For a control problem, where the goal is to find an optimal policy, there might well be states that cannot be reached by any optimal policy from any of the start states, and there is no need to specify optimal actions for these irrelevant states
- What is needed is an optimal partial policy: a policy that is optimal for the relevant states but can specify arbitrary actions, or even be undefined, for the irrelevant states





Real-time DP

- For certain types of problems satisfying reasonable conditions, RTDP is guaranteed to find a policy that is optimal on the relevant states without visiting every state infinitely often, or even without visiting some states at all
- This can be a great advantage for problems with very large state sets, where even a single sweep may not be feasible
- The tasks for which this result holds are undiscounted episodic tasks for MDPs with absorbing goal states that generate zero rewards
- At every step of a real or simulated trajectory, RTDP selects a greedy action (breaking ties randomly) and applies the expected value-iteration update operation to the current state. It can also update the values of an arbitrary collection of other states at each step



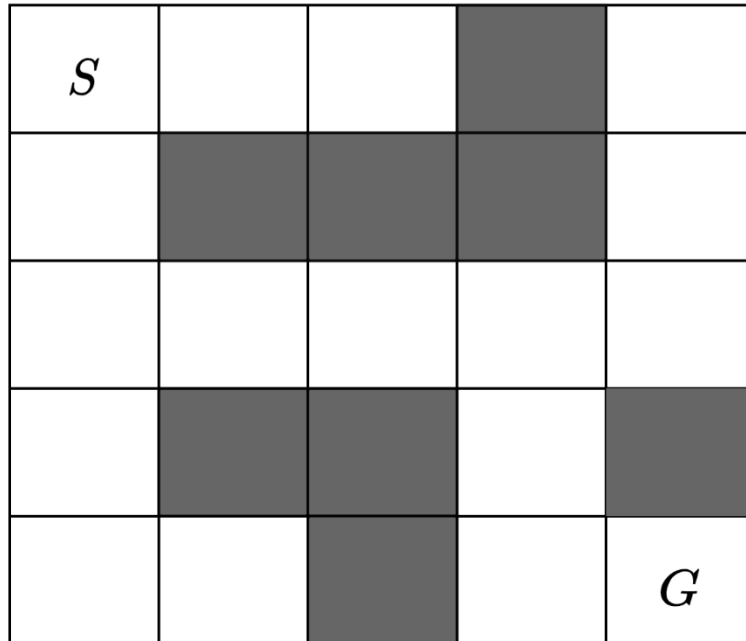
Real-time DP

- For these problems, with each episode beginning in a state randomly chosen from the set of start states and ending at a goal state, RTDP converges with probability 1 to a policy that is optimal for all the relevant states if
 - 1) the initial value of every goal state is zero
 - 2) there exists at least one policy that guarantees that a goal state will be reached with probability 1 from any start state
 - 3) all rewards for transitions from non-goal states are strictly negative
 - 4) all the initial values are equal to, or greater than, their optimal values (which can be satisfied by simply setting the initial values of all states to 0)
- Tasks having these properties are examples of stochastic optimal path problems
 - Example: 1) minimum-time control tasks, where each time step required to reach a goal produces a reward of -1 , and 2) Golf whose objective is to hit the hole with the fewest strokes

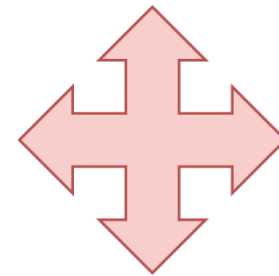


Example: RTDP on Simple Maze

Initial state



Terminal state

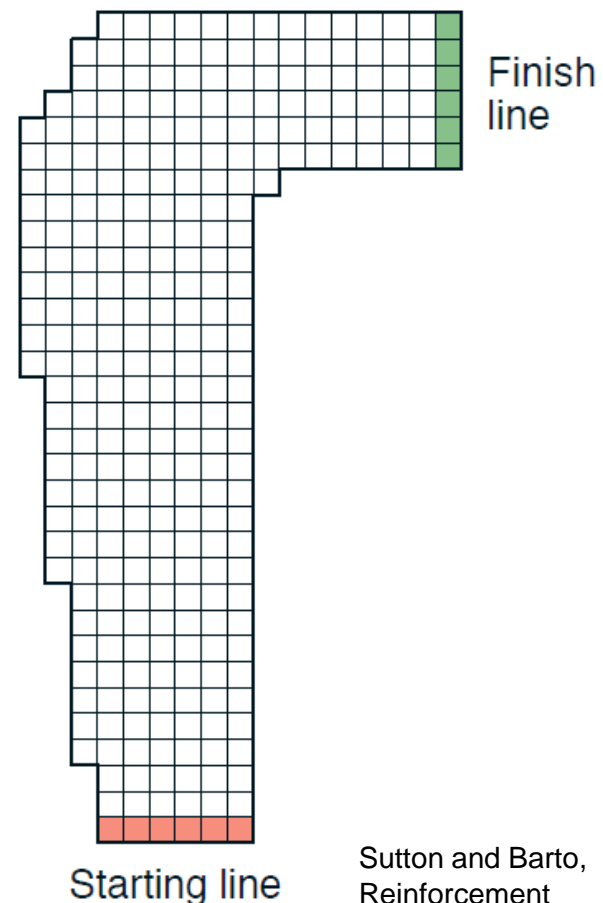


Action space



Example: RTDP on Racetrack

- A stochastic optimal path problem
- An agent has to learn how to drive a car around a turn, and cross the finish line as quickly as possible while staying on the track
- Start states are all the zero-speed states on the starting line; the goal states are all the states that can be reached in one time step by crossing the finish line from inside the track
- Each episode begins in a randomly selected start state and ends when the car crosses the finish line
- Rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random start state, and the episode continues



Sutton and Barto,
Reinforcement
Learning, 2018



Example: RTDP on Racetrack

- Consider a racetrack with 9,115 states reachable from start states by any policy, only 599 of which are relevant, meaning that they are reachable from some start state via some optimal policy
- How well do DP and RTDP solve this problem?
- Conventional DP: value iteration using exhaustive sweeps of the state set, with values updated one state at a time in place, meaning that the update for each state uses the most recent values of the other states
- Initial values were all zero for each run of both methods
- DP was judged to have converged when the maximum change in a state value over a sweep was less than 10^{-4} , and RTDP was judged to have converged when the average time to cross the finish line over 20 episodes appeared to stabilize at an asymptotic number of steps
- This version of RTDP updated only the value of the current state on each step



Example: RTDP on Racetrack

| | DP | RTDP |
|--|-----------|---------------|
| Average computation to convergence | 28 sweeps | 4000 episodes |
| Average number of updates to convergence | 252,784 | 127,600 |
| Average number of updates per episode | — | 31.9 |
| % of states updated ≤ 100 times | — | 98.45 |
| % of states updated ≤ 10 times | — | 80.51 |
| % of states updated 0 times | — | 3.18 |

- Both methods produced policies averaging between 14 and 15 steps to cross the finish line, but RTDP required only roughly half of the updates that DP did
- This is the result of RTDP's on-policy trajectory sampling: while the value of every state was updated in each sweep of DP, RTDP focused updates on fewer states
- In an average run, RTDP updated the values of 98.45% of the states no more than 100 times and 80.51% of the states no more than 10 times; the values of about 290 states were not updated at all




Real-time DP

- Another advantage of RTDP is that as the value function approaches the optimal value function v_* , the policy used by the agent to generate trajectories approaches an optimal policy because it is always greedy with respect to the current value function
- This is in contrast to the situation in conventional value iteration
- In practice, value iteration terminates when the value function changes by only a small amount in a sweep. At this point, the value function closely approximates v_* , and a greedy policy is close to an optimal policy
- However, it is possible that policies that are greedy with respect to the latest value function were optimal, long before value iteration terminates
- For this racetrack, a close-to-optimal policy emerged after 15 sweeps of value iteration, or after 136,725 value-iteration updates. This is considerably less than the 252,784 updates DP needed to converge to v_* , but still more than the 127,600 updates RTDP required
- RTDP achieved nearly optimal control with about 50% of the computation required by sweep-based value iteration



Outline

- Models and Planning
- Dyna: Integrated Planning, Acting, and Learning
- When the Model Is Wrong
- Prioritized Sweeping
- Expected vs. Sample Updates
- Trajectory Sampling
- Real-time DP
-  **Planning at Decision Time**
- Heuristic Search
- Rollout Algorithms
- MC Tree Search
- Conclusion



Planning at Decision Time

- Planning can be used in at least two ways
- Option 1: background planning
 - Use planning to gradually improve a policy or value function on the basis of simulated experience obtained from a model (either a sample or a distribution model)
 - Example: dynamic programming and Dyna
 - Selecting actions is then a matter of comparing the current state's action values obtained from a table in the tabular case we have thus far considered, or by evaluating a mathematical expression in the approximate methods
 - Well before an action is selected for any current state S_t , planning has played a part in improving the table entries, or the mathematical expression, needed to select the action for many states, including S_t
 - Used this way, planning is not focused on the current state



Planning at Decision Time

- Option 2: decision-time planning
 - The other way to use planning is to begin and complete it after encountering each new state S_t , as a computation whose output is the selection of a single action A_t
 - On the next step planning begins anew with S_{t+1} to produce A_{t+1} , etc.
 - The simplest, and almost degenerate, example of this use of planning is when only state values are available, and an action is selected by comparing the values of model-predicted next states for each action (e.g., tic-tac-toe)
 - More generally, planning used in this way can look much deeper than one-step-ahead and evaluate action choices leading to many different predicted state and reward trajectories
 - Unlike Option 1, Option 2 focuses on a particular state



Planning at Decision Time

- Even when planning is only done at decision time, we can still view it as proceeding from simulated experience to updates and values, and ultimately to a policy
- It is just that now the values and policy are specific to the current state and the action choices available there, so much so that the values and policy created by the planning process are typically discarded after being used to select the current action
- In many applications this is not a problem because there are many states and we are unlikely to return to the same state for a long time
- In general, one may want to do a mix of both: focus planning on the current state and store the results of planning, since one might return to the same state later




Planning at Decision Time

- Decision-time planning is most useful in applications in which fast responses are not required
- In chess playing programs, for example, one may be permitted seconds or minutes of computation for each move, and strong programs may plan dozens of moves ahead within this time
- On the other hand, if low latency action selection is the priority, then one is generally better off doing planning in the background to compute a policy that can then be rapidly applied to each newly encountered state



Outline

- Models and Planning
- Dyna: Integrated Planning, Acting, and Learning
- When the Model Is Wrong
- Prioritized Sweeping
- Expected vs. Sample Updates
- Trajectory Sampling
- Real-time DP
- Planning at Decision Time
-  **Heuristic Search**
- Rollout Algorithms
- MC Tree Search
- Conclusion

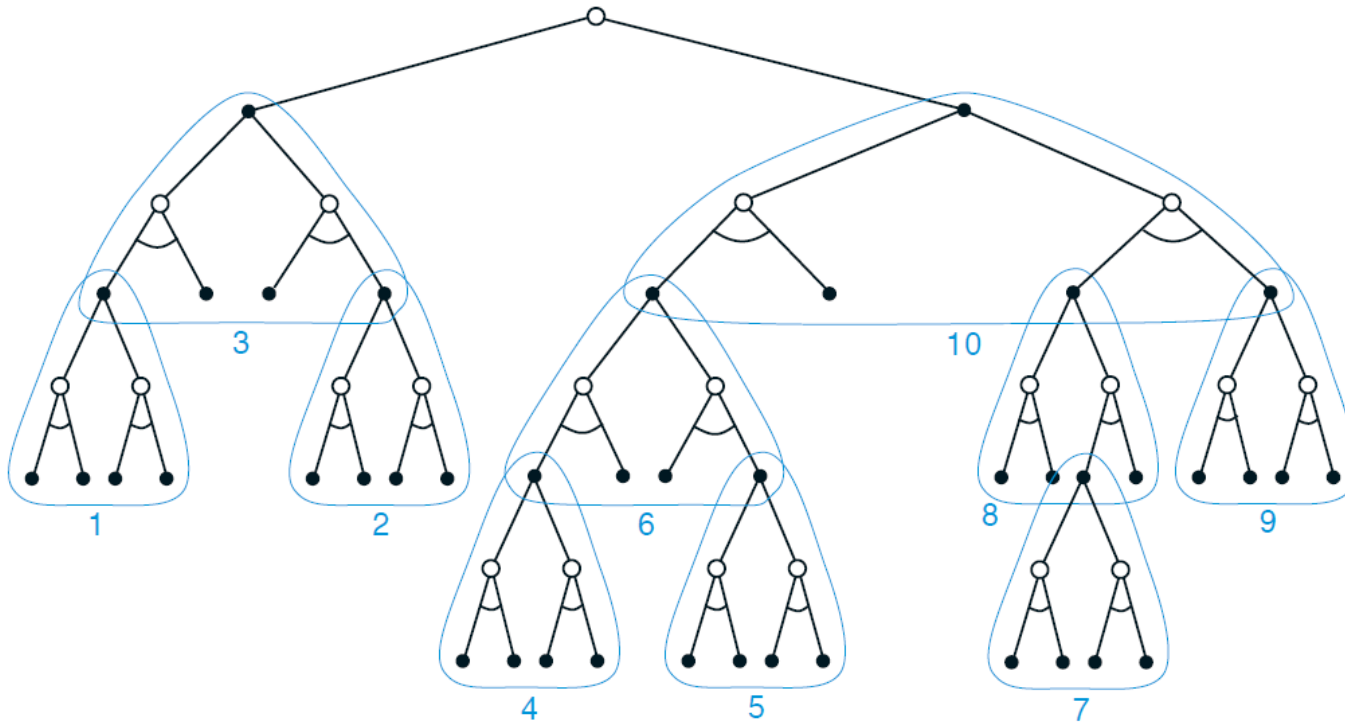


Heuristic Search

- The classical state-space planning methods in AI are decision-time planning methods collectively known as heuristic search
- In heuristic search, for each state encountered, a large tree of possible continuations is considered
- The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root
- The backing up within the search tree is just the same as in the expected updates with maxes (those for v_* and q_*)
- The backing up stops at the state–action nodes for the current state
- Once the backed-up values of these nodes are computed, the best of them is chosen as the current action, and then all backed-up values are discarded



Heuristic Search



Sutton and Barto,
Reinforcement
Learning, 2018



Heuristic Search

- In conventional heuristic search no effort is made to save the backed-up values by changing the approximate value function
- In fact, the value function is generally designed by people and never changed as a result of search
- However, it is natural to consider allowing the value function to be improved over time, using either the backed-up values computed during heuristic search or any of the other methods
- Example of non-saving method: ϵ -greedy
 - To compute the greedy action, we must look ahead from each possible action to each possible next state, take into account the rewards and estimated values, and then pick the best action
 - Just as in conventional heuristic search, this process computes backed-up values of the possible actions, but does not attempt to save them
 - Thus, heuristic search can be viewed as an extension of the idea of a greedy policy beyond a single step



Heuristic Search

- The point of searching deeper than one step is to obtain better action selections. If one has a perfect model and an imperfect action-value function, then in fact deeper search will usually yield better policies
- Certainly, if the search is all the way to the end of the episode, then the effect of the imperfect value function is eliminated, and the action determined in this way must be optimal
- If the search is of sufficient depth k such that γ^k is very small, then the actions will be correspondingly near optimal
- On the other hand, the deeper the search, the more computation is required, usually resulting in a slower response time



Heuristic Search

- Example: TD-Gammon
 - Uses TD learning to learn an afterstate value function through many games of self-play, using a form of heuristic search to make its moves
 - As a model, TD-Gammon used a priori knowledge of the probabilities of dice rolls and the assumption that the opponent always selected the actions that TD-Gammon rated as best for it
 - Tesauro found that the deeper the heuristic search, the better the moves made by TD-Gammon, but the longer it took to make each move
 - Backgammon has a large branching factor, yet moves must be made within a few seconds
 - It was only feasible to search ahead selectively a few steps, but even so the search resulted in significantly better action selections



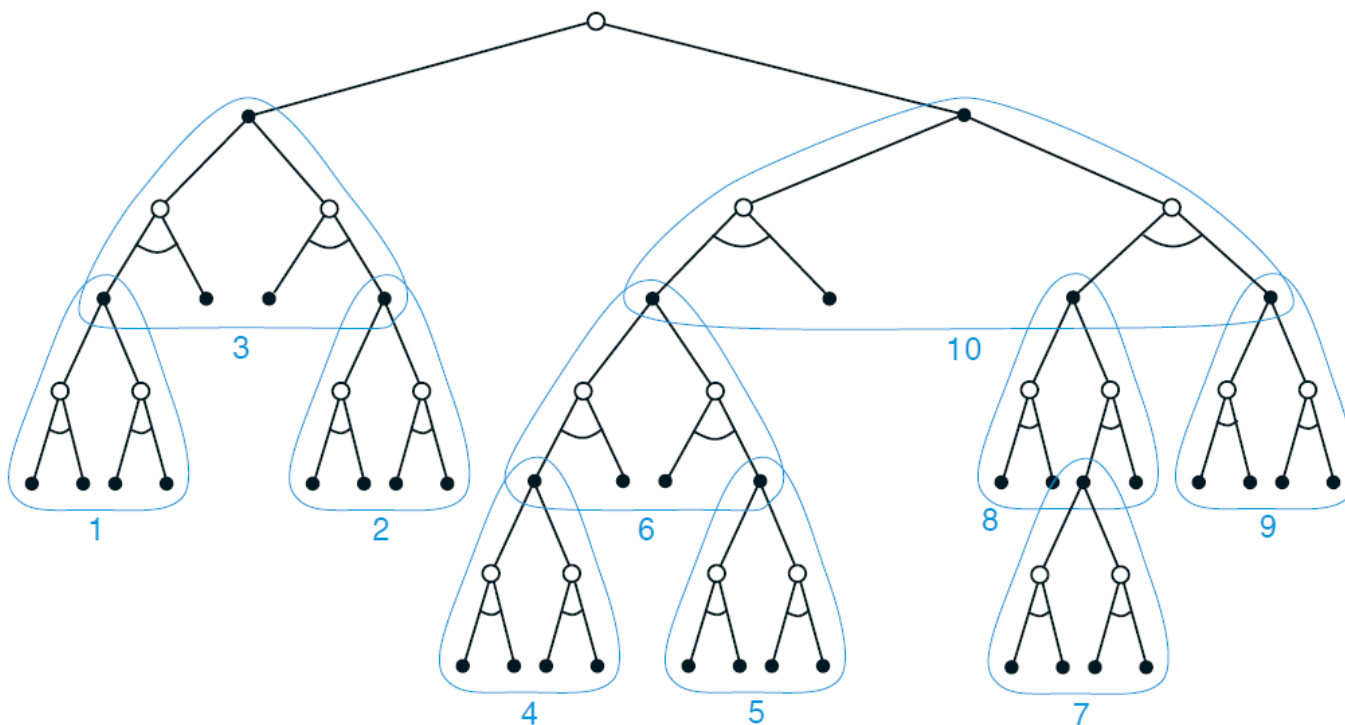
Heuristic Search

- Much of the effectiveness of heuristic search is due to its search tree being tightly focused on the states and actions that might immediately follow the *current state*
- No matter how you select actions, it is these states and actions that are of highest priority for updates and where you most urgently want your approximate value function to be accurate
- The computation and memory should be devoted to imminent events
- In chess, there are far too many possible positions to store distinct value estimates for each of them, but chess programs based on heuristic search can easily store distinct estimates for the millions of positions they encounter looking ahead from a single position
- This great focusing of memory and computational resources on the current decision is presumably the reason why heuristic search can be so effective




Heuristic Search

- The distribution of updates can be altered in similar ways to focus on the current state and its likely successors
- As a limiting case we might use exactly the methods of heuristic search to construct a search tree, and then perform the individual, one-step updates from bottom up





Outline

- Models and Planning
- Dyna: Integrated Planning, Acting, and Learning
- When the Model Is Wrong
- Prioritized Sweeping
- Expected vs. Sample Updates
- Trajectory Sampling
- Real-time DP
- Planning at Decision Time
- Heuristic Search
-  **Rollout Algorithms**
- MC Tree Search
- Conclusion



Rollout Algorithms

- Rollout algorithms are decision-time planning algorithms based on MC control applied to simulated trajectories that all begin at the current environment state
- They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy
- When the action-value estimates are considered to be accurate enough, the action (or one of the actions) having the highest estimated value is executed, after which the process is carried out anew from the resulting next state



Rollout Algorithms

- Unlike the MC control algorithms described in Chapter 5, the goal of a rollout algorithm is not to estimate a complete optimal action-value function, q_* , or a complete action-value function, q_π , for a given policy π
- Instead, they produce MC estimates of action values only for each current state and for a given policy usually called the rollout policy
- As decision-time planning algorithms, rollout algorithms make immediate use of these action-value estimates, then discard them
- This makes rollout algorithms relatively simple to implement because there is no need to sample outcomes for every state-action pair, and there is no need to approximate a function over either the state space or the state-action space



Rollout Algorithms

- What then do rollout algorithms accomplish?
- The policy improvement theorem says that given any two policies π and π' that are identical except that $\pi'(s) = a \neq \pi(s)$ for some state s , if $q_\pi(s, a) \geq v_\pi(s)$, then policy π' is as good as, or better, than π
- Moreover, if the inequality is strict, then π' is in fact better than π
- This applies to rollout algorithms where s is the current state and π is the rollout policy
- Averaging the returns of the simulated trajectories produces estimates of $q_\pi(s, a')$ for each action $a' \in A(s)$
- Then the policy that selects an action in s that maximizes these estimates and thereafter follows π is a good candidate for a policy that improves over π



Rollout Algorithms

- In other words, the aim of a rollout algorithm is to improve upon the rollout policy (not to find an optimal policy)
- Rollout algorithms can be surprisingly effective
- For example, Tesauro and Galperin (1997) were surprised by the dramatic improvements in backgammon playing ability produced by the rollout method
- In some applications, a rollout algorithm can produce good performance even if the rollout policy is completely random
- But the performance of the improved policy depends on properties of the rollout policy and the ranking of actions from the MC value estimates
- Intuition suggests that the better the rollout policy and the more accurate the value estimates, the better the policy produced by a rollout algorithm is likely be



Rollout Algorithms

- This involves important tradeoffs because better rollout policies typically mean that more time is needed to simulate enough trajectories to obtain good value estimates
- As decision-time planning methods, rollout algorithms usually have to meet strict time constraints
- The computation time needed by a rollout algorithm depends on
 - the number of actions that have to be evaluated for each decision
 - the number of time steps in the simulated trajectories needed to obtain useful sample returns
 - the time it takes the rollout policy to make decisions, and
 - the number of simulated trajectories needed to obtain good MC action-value estimates



Rollout Algorithms

- Balancing these factors is important in any application of rollout methods, though there are several ways to ease the challenge
- Because the MC trials are independent of one another, it is possible to run many trials in parallel on separate processors
- Another approach is to truncate the simulated trajectories short of complete episodes, correcting the truncated returns by means of a stored evaluation function (which brings into play all that we have said about truncated returns and updates in the preceding chapters)
- It is also possible to monitor the MC simulations and prune away candidate actions that are unlikely to turn out to be the best, or whose values are close enough to that of the current best that choosing them instead would make no real difference



Rollout Algorithms

- Rollout algorithms are not considered as learning algorithms because they do not maintain long-term memories of values or policies
- However, these algorithms take advantage of some of the features of RL
- As instances of MC control, they estimate action values by averaging the returns of a collection of sample trajectories, in this case trajectories of simulated interactions with a sample model of the environment
- In this way they are like RL algorithms in avoiding the exhaustive sweeps of DP by trajectory sampling, and in avoiding the need for distribution models by relying on sample, instead of expected, updates
- Rollout algorithms take advantage of the policy improvement property by acting greedily with respect to the estimated action values



Outline

- Models and Planning
- Dyna: Integrated Planning, Acting, and Learning
- When the Model Is Wrong
- Prioritized Sweeping
- Expected vs. Sample Updates
- Trajectory Sampling
- Real-time DP
- Planning at Decision Time
- Heuristic Search
- Rollout Algorithms
- MC Tree Search**
- Conclusion





Monte Carlo Tree Search

- Monte Carlo Tree Search (MCTS) is a recent and strikingly successful example of decision-time planning
- At its base, MCTS is a rollout algorithm, but enhanced by the addition of a means for accumulating value estimates obtained from the MC simulations in order to successively direct simulations toward more highly-rewarding trajectories
- MCTS is largely responsible for the improvement in computer Go from a weak amateur level in 2005 to a grandmaster level in 2015
- MCTS has proved to be effective in a wide variety of competitive settings, including general game playing, but it is not limited to games; it can be effective for other single-agent sequential decision problems if there is an environment model simple enough for fast multistep simulation



Monte Carlo Tree Search

- MCTS is executed after encountering each new state to select the agent's action for that state; it is executed again to select the action for the next state, and so on
- As in a rollout algorithm, each execution is an iterative process that simulates many trajectories starting from the current state and running to a terminal state (or until discounting makes any further reward negligible as a contribution to the return)
- Main idea of MCTS: successively focus multiple simulations starting at the current state by extending the initial portions of trajectories that have received high evaluations from earlier simulations
- MCTS does not have to retain approximate value functions or policies from one action selection to the next, though in many implementations it retains selected action values likely to be useful for its next execution

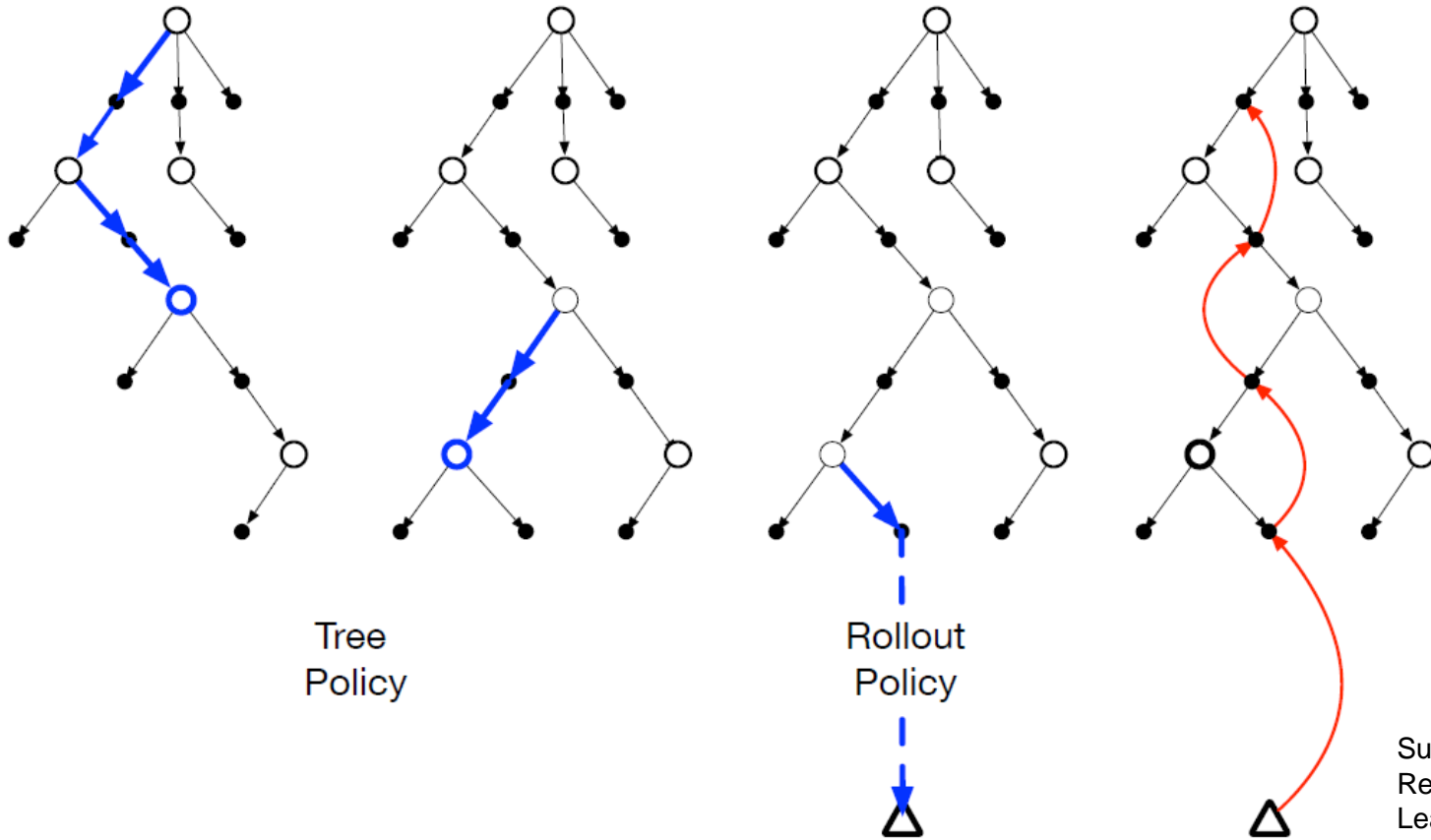
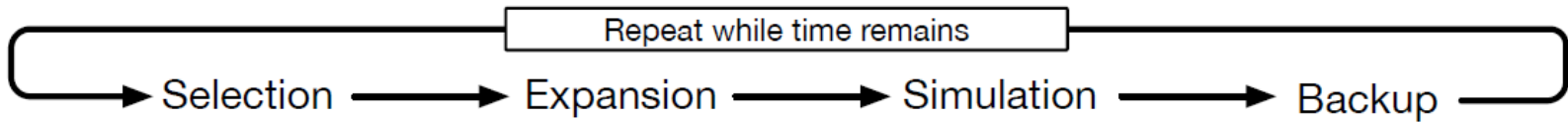


Monte Carlo Tree Search

- For the most part, the actions in the simulated trajectories are generated using a simple policy called a rollout policy
- When both the rollout policy and the model do not require a lot of computation, many simulated trajectories can be generated in a short period of time
- As in any tabular MC method, the value of a state–action pair is estimated as the average of the (simulated) returns from that pair
- MC value estimates are maintained only for the subset of state–action pairs that are most likely to be reached in a few steps, which form a tree rooted at the current state



Monte Carlo Tree Search



Sutton and Barto,
Reinforcement
Learning, 2018



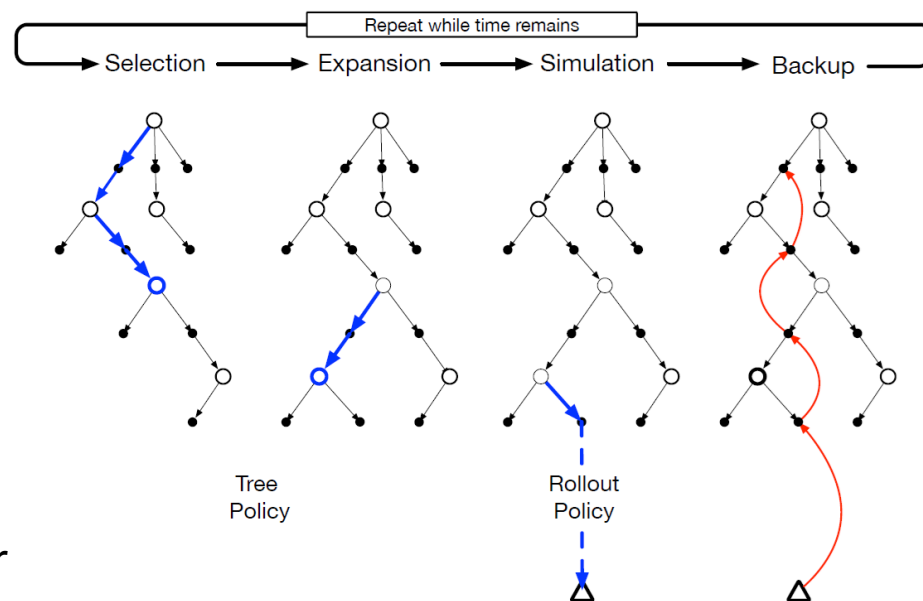
Monte Carlo Tree Search

- MCTS incrementally extends the tree by adding nodes representing states that look promising based on the results of the simulated trajectories
- Any simulated trajectory will pass through the tree and then exit it at some leaf node
- Outside the tree and at the leaf nodes the rollout policy is used for action selections, but at the states inside the tree something better is possible
- For these states we have value estimates for of at least some of the actions, so we can pick among them using an informed policy, called the tree policy, that balances exploration and exploitation
- For example, the tree policy could select actions using an ϵ -greedy or UCB selection rule



Monte Carlo Tree Search

- Each iteration of MCTS has four steps
- 1. Selection
 - Starting at the root node, a tree policy based on the action values attached to the edges of the tree traverses the tree to select a leaf node
- 2. Expansion
 - The tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions

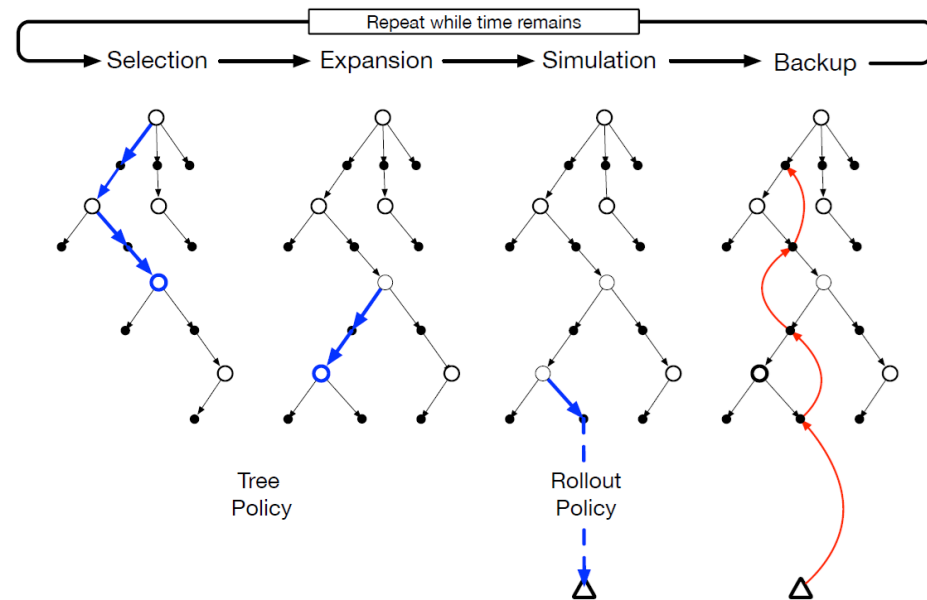


Sutton and Barto,
Reinforcement
Learning, 2018



Monte Carlo Tree Search

- 3. Simulation
 - From the selected node, or from one of its newly-added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is an MC trial with actions selected first by the tree policy and beyond the tree by the rollout policy



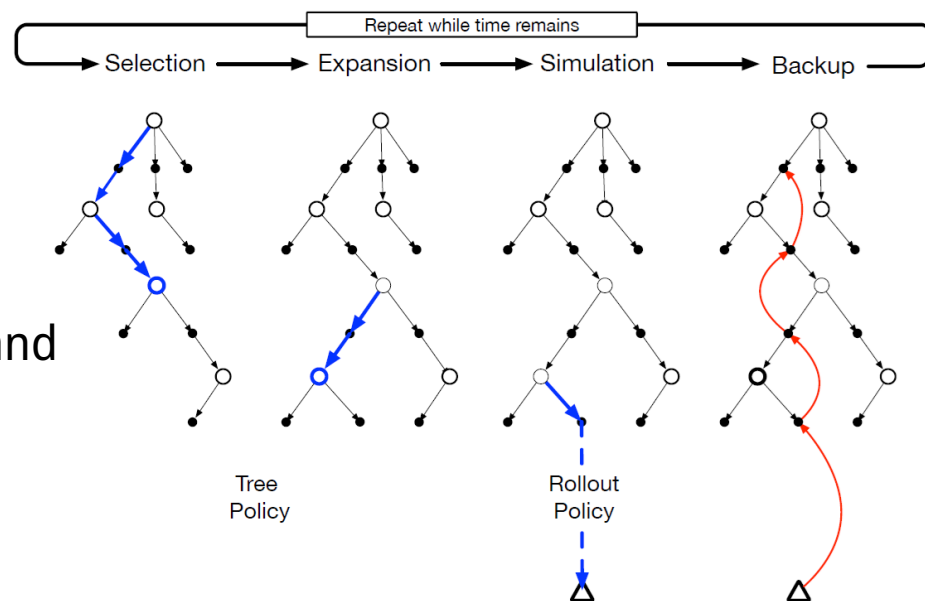
Sutton and Barto,
Reinforcement
Learning, 2018



Monte Carlo Tree Search

■ 4. Backup

- The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS
- No values are saved for the states and actions visited by the rollout policy beyond the tree



Sutton and Barto,
Reinforcement
Learning, 2018



Monte Carlo Tree Search

- MCTS continues executing these four steps, starting each time at the tree's root node, until no more time is left, or some other computational resource is exhausted
- Then, finally, an action from the root node (which still represents the current state of the environment) is selected according to some mechanism that depends on the accumulated statistics in the tree; for example, it may be an action having the largest action value of all the actions available from the root state, or perhaps the action with the largest visit count to avoid selecting outliers
- After the environment transitions to a new state, MCTS is run again, sometimes starting with a tree of a single root node representing the new state, but often starting with a tree containing any descendants of this node left over from the tree constructed by the previous execution of MCTS; all the remaining nodes are discarded, along with the action values associated with them



Monte Carlo Tree Search

- MCTS was first proposed to select moves in programs playing two-person competitive games, such as Go
- For game playing, each simulated episode is one complete play of the game in which both players select actions by the tree and rollout policies
- An extension of MCTS used in the AlphaGo program combines the MC evaluations of MCTS with action values learned by a deep artificial neural network via self-play reinforcement learning



Monte Carlo Tree Search

- MCTS is a decision-time planning algorithm based on MC control applied to simulations that start from the root state; that is, it is a kind of rollout algorithm. It therefore benefits from online, incremental, sample-based value estimation and policy improvement
- Beyond this, it saves action-value estimates attached to the tree edges and updates them using RL's sample updates
- This has the effect of focusing the MC trials on trajectories whose initial segments are common to high-return trajectories previously simulated
- By incrementally expanding the tree, MCTS effectively grows a lookup table to store a partial action-value function, with memory allocated to the estimated values of state–action pairs visited in the initial segments of high-yielding sample trajectories
- MCTS thus avoids the problem of globally approximating an action-value function while it retains the benefit of using past experience to guide exploration



Outline

- Models and Planning
- Dyna: Integrated Planning, Acting, and Learning
- When the Model Is Wrong
- Prioritized Sweeping
- Expected vs. Sample Updates
- Trajectory Sampling
- Real-time DP
- Planning at Decision Time
- Heuristic Search
- Rollout Algorithms
- MC Tree Search
- Conclusion**





Conclusion

- Planning requires a model of the environment
- A distribution model consists of the probabilities of next states and rewards for possible actions; a sample model produces single transitions and rewards generated according to these probabilities
- DP requires a distribution model because it uses expected updates, which involve computing expectations over all the possible next states and rewards
- A sample model, on the other hand, is what is needed to simulate interacting with the environment during which sample updates, like those used by many RL algorithms, can be used
- Sample models are generally much easier to obtain than distribution models



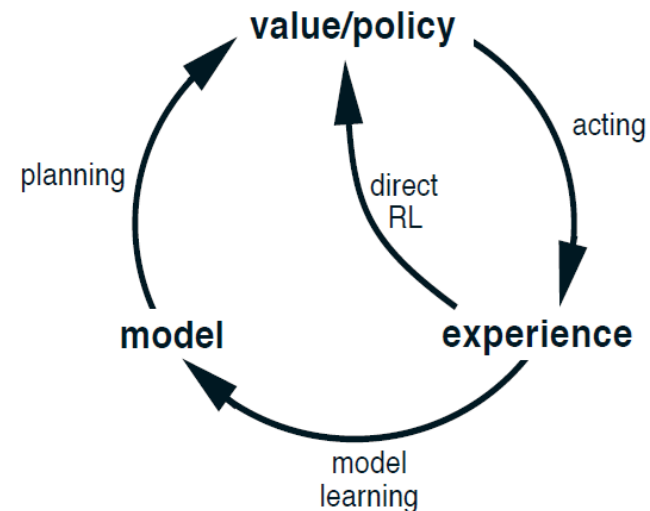
Conclusion

- Close relationships between planning optimal behavior and learning optimal behavior
 - Both involve estimating the same value functions, and in both cases it is natural to update the estimates incrementally, in a long series of small backing-up operations
 - This makes it straightforward to integrate learning and planning processes simply by allowing both to update the same estimated value function
 - In addition, any of the learning methods can be converted into planning methods simply by applying them to simulated (model- generated) experience rather than to real experience
 - In this case learning and planning become even more similar; they are possibly identical algorithms operating on two different sources of experience



Conclusion

- It is straightforward to integrate incremental planning methods with acting and model learning
- Planning, acting, and model-learning interact in a circular fashion, each producing what the other needs to improve
- The most natural approach is for all processes to proceed asynchronously and in parallel
- If the processes must share computational resources, then the division can be handled almost arbitrarily—by whatever organization is most convenient and efficient for the task at hand



Sutton and Barto,
Reinforcement
Learning, 2018



Conclusion

- Dimensions of variation among state-space planning methods
- Dim 1: variation in the size of updates
 - The smaller the updates, the more incremental the planning methods can be
 - Among the smallest updates are one-step sample updates, as in Dyna
- Dim 2: distribution of updates, that is, of the focus of search
 - Prioritized sweeping focuses backward on the predecessors of states whose values have recently changed
 - On-policy trajectory sampling focuses on states or state–action pairs that the agent is likely to encounter when controlling its environment. This can allow computation to skip over parts of the state space that are irrelevant to the prediction or control problem
 - Realtime DP, an on-policy trajectory sampling version of value iteration, illustrates some of the advantages this strategy has over conventional sweep-based policy iteration



Conclusion

- Planning can also focus forward from pertinent states, such as states actually encountered during an agent-environment interaction
- The most important form of this is when planning is done at decision time, that is, as part of the action-selection process
- Classical heuristic search as studied in AI is an example of this
- Other examples are rollout algorithms and MCTS that benefit from online, incremental, sample-based value estimation and policy improvement



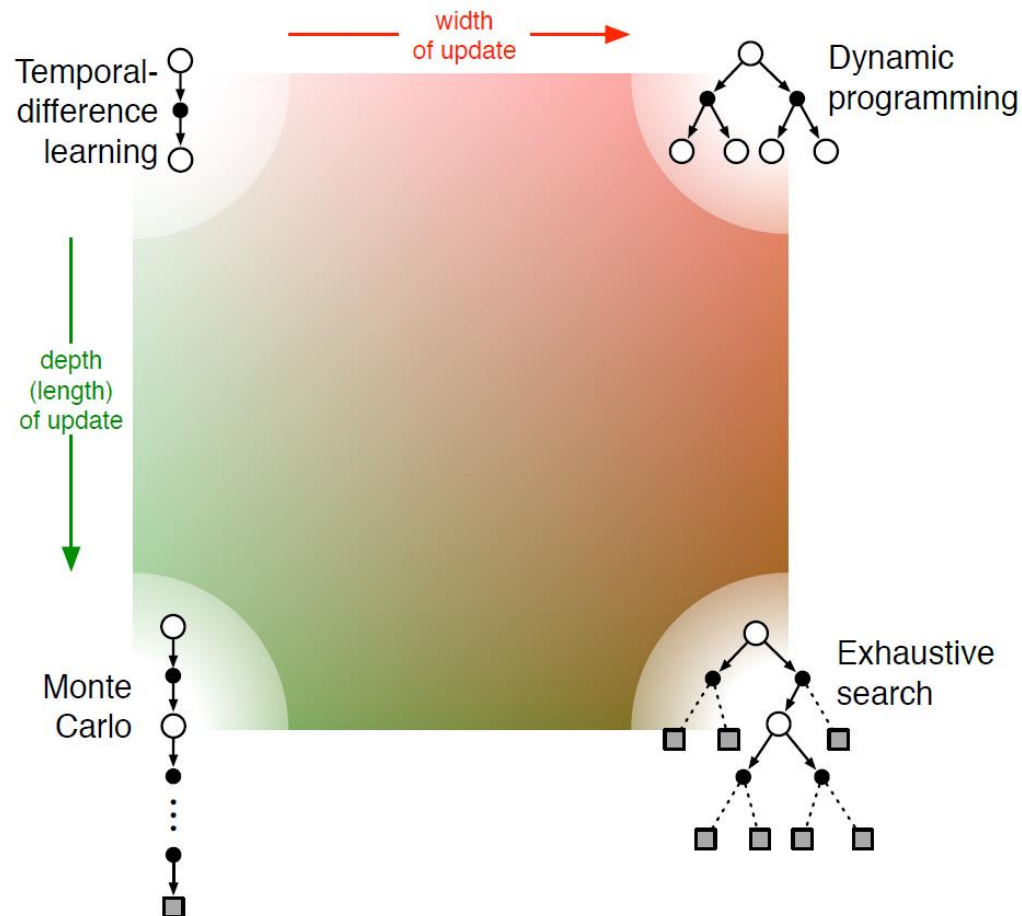
Conclusion: Part I

- Key ideas of many RL methods
 - 1) They all seek to estimate value functions
 - 2) They all operate by backing up values along actual or possible state trajectories
 - 3) They all follow the general strategy of generalized policy iteration (GPI), meaning that they maintain an approximate value function and an approximate policy, and they continually try to improve each on the basis of the other



Conclusion: Part I

- Various dimensions of RL methods
 - Width and depth of update





Conclusion: Part I

- Various dimensions of RL methods
 - On-policy and off-policy methods
 - Definition of return: episodic or continuing, discounted or undiscounted?
 - Action values vs. state values vs. afterstate values
 - Action selection/exploration
 - Synchronous vs. asynchronous
 - Real vs. simulated
 - Location of updates: what states or state–action pairs should be updated?
 - Timing of updates: should updates be done as part of selecting actions, or only afterward?
 - Memory for updates: how long should updated values be retained? Should they be retained permanently, or only while computing an action selection, as in heuristic search?



Questions?