

Chapter 8: Planning and Learning with Tabular Methods

Seungjae Ryan Lee

Major Themes of Chapter 8

1. Unifying *planning* and *learning* methods
 - Model-based methods rely on planning
 - Model-free methods rely on learning
2. Benefits of planning in small, incremental steps
 - Key to efficient combination of planning and learning



Model of the Environment

- Anything the agent can use to predict the environment
- Used to mimic or *simulate* experience

1. *Distribution models*

- Describe all possibilities and probabilities
- Stronger but difficult to obtain

2. *Sample models*

- Produce one possibility sampled according to the probabilities
- Weaker but easier to obtain

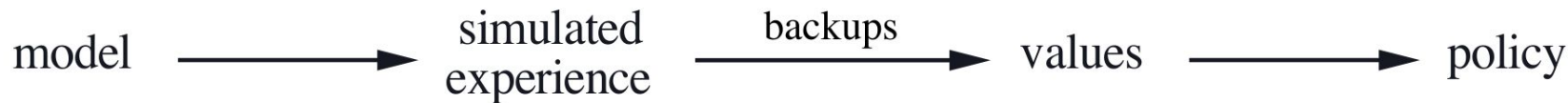
Planning

- Process of using a model to produce / improve policy
- *State-space Planning*
 - Search through the state space for optimal policy
 - Includes RL approaches introduced until now
- *Plan-space Planning*
 - Search through space of plans
 - Not efficient in RL
 - ex) Evolutionary methods



State-space Planning

- All state-space planning methods share common structure
 - Involves **computing value functions**
 - Compute value functions by **updates or backup operations**

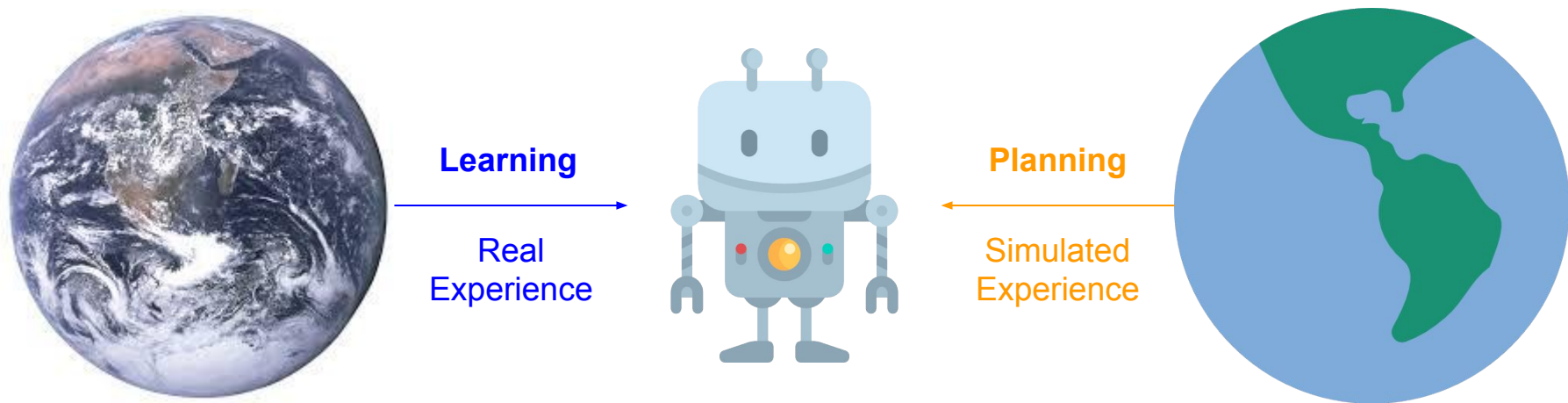


- ex) Dynamic Programming



Relationship between Planning and Learning

- Both estimate value function by backup update operations
- *Planning* uses simulated experience from model
- *Learning* uses real experience from environment



Random-sample one-step tabular Q-planning

- Planning and Learning is similar enough to transfer algorithms
- Same convergence guarantees as one-step tabular Q-learning
 - All states and actions selected infinitely many times
 - Step size decrease over time
- Need just the *sample model*

Random-sample one-step tabular Q-planning

Random-sample one-step tabular Q-planning

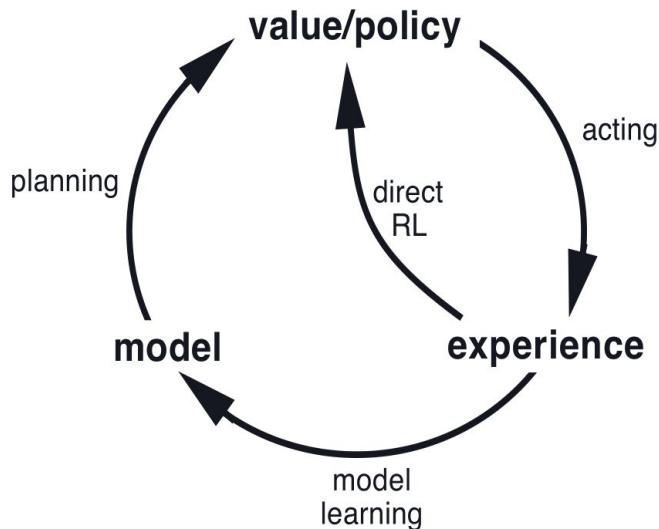
Loop forever:

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' :

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

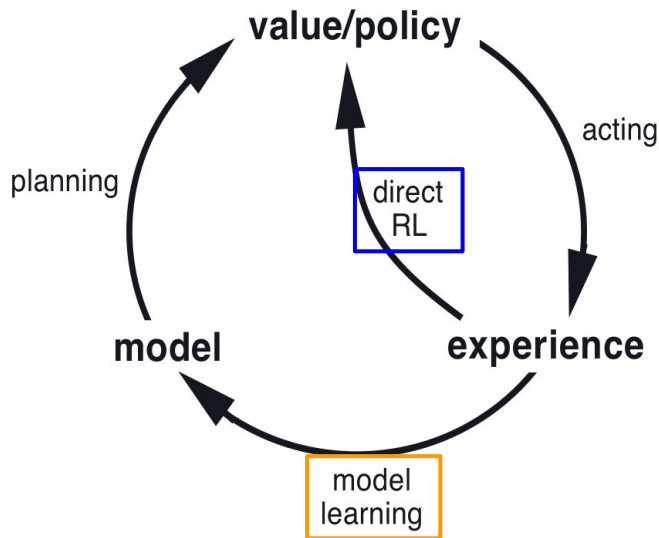
On-line Planning

- Need to incorporate new experience with planning
- Divide computational resources between decision making and model learning



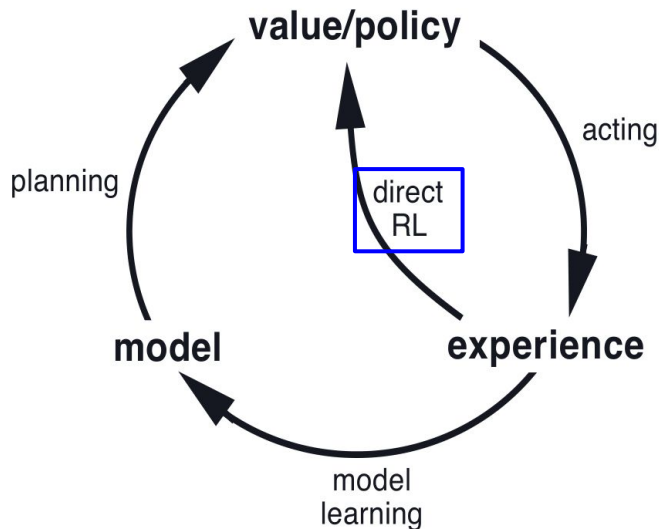
Roles of Real Experience

1. *Model learning*: Improve the model to increase accuracy
2. *Direct RL*: Directly improve the value function and policy



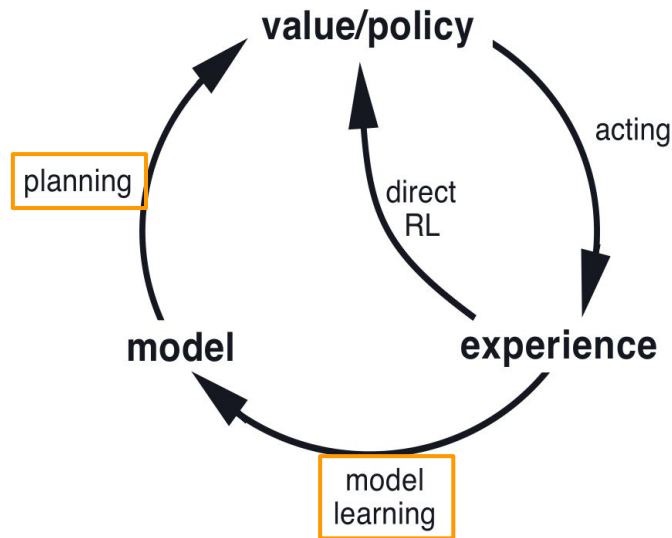
Direct Reinforcement Learning

- Improve value functions and policy with real experience
- Simpler and not affected by bias from model design



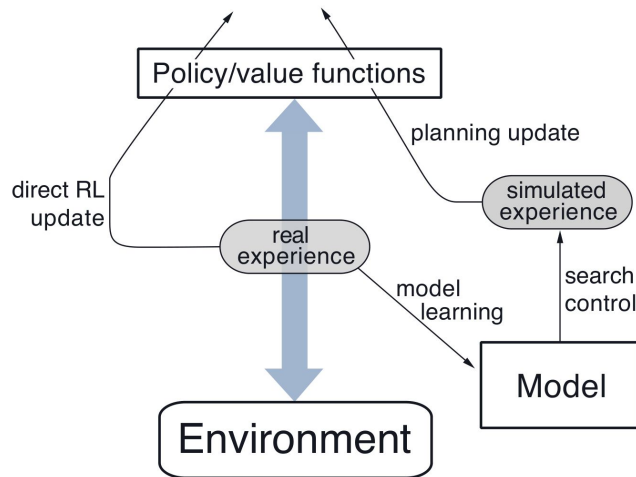
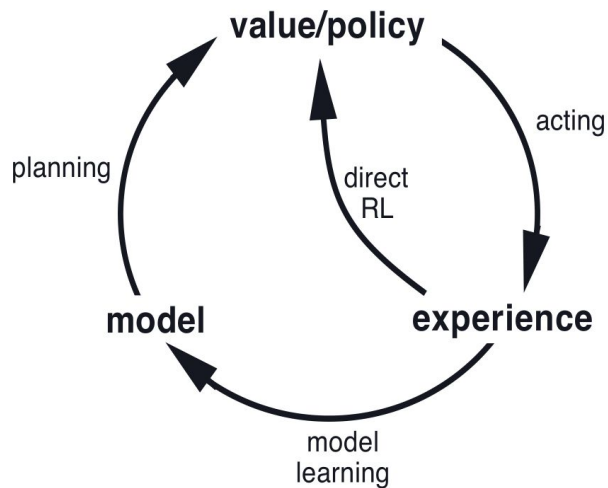
Indirect Reinforcement Learning

- Improve value functions and policy by improving the model
- Achieve better policy with fewer environmental interactions



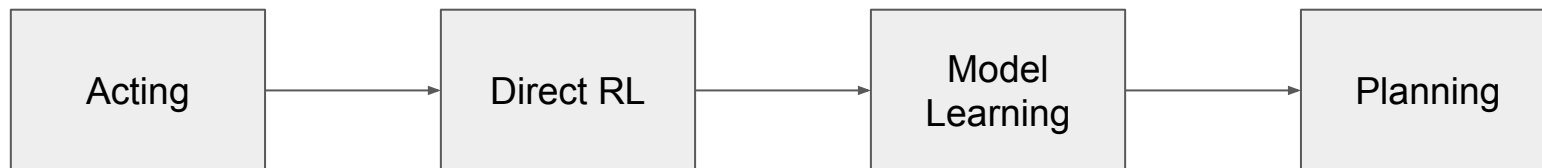
Dyna-Q

- Simple on-line planning agent with all processes
 - Planning: random-sample one-step tabular Q-planning
 - Direct RL: one-step tabular Q-learning
 - Model learning: Return last observed next state and reward as prediction



Dyna-Q: Implementation

- Order of process on a serial computer



- Can be parallelized to a *reactive, deliberative* agent
 - *reactive*: responding instantly to latest information
 - *deliberative*: always planning in the background
- Planning is most computationally intensive
 - Complete n iteration of Q-planning algorithm on each timestep

Dyna-Q: Pseudocode

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) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
- (e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
 - $S \leftarrow$ random previously observed state
 - $A \leftarrow$ random action previously taken in S
 - $R, S' \leftarrow Model(S, A)$
 - $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

Acting

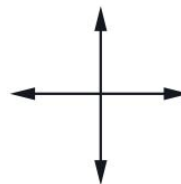
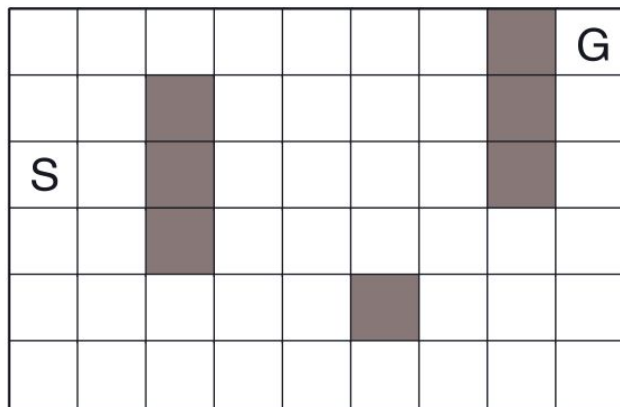
Direct RL

Model Learning

Planning

Dyna Maze Example

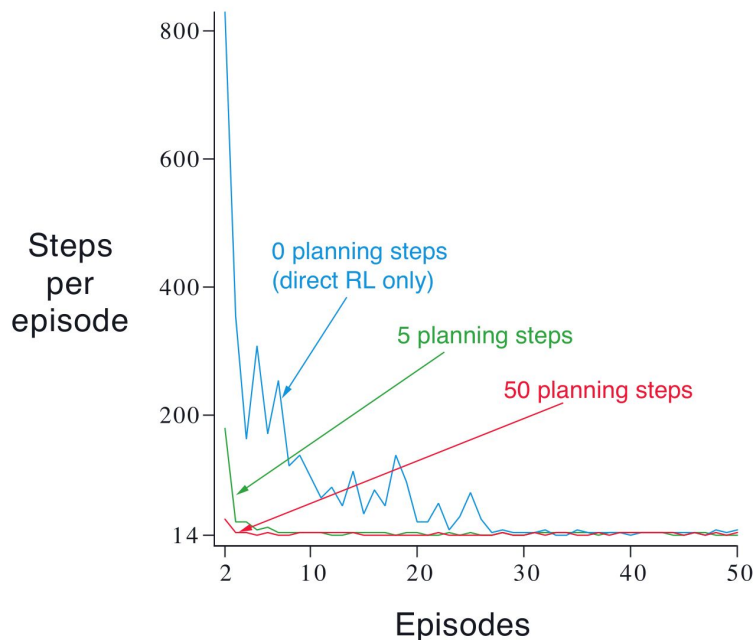
- Only reward is on reaching goal state (+1)
 - Takes long time for reward propagate



actions

Dyna Maze Example: Result

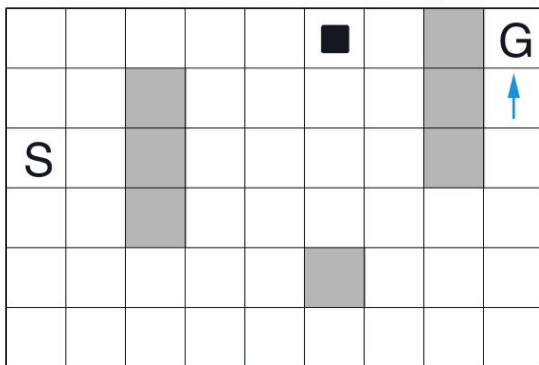
- Much faster convergence to optimal policy



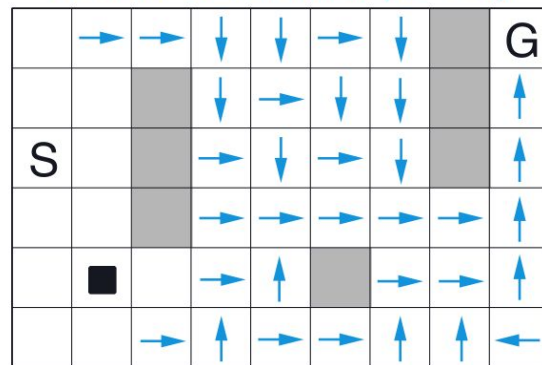
Dyna Maze Example: Intuition

- Without planning, each episode adds only one step to the policy
- With planning, extensive policy is developed by the end of episode

WITHOUT PLANNING ($n=0$)



WITH PLANNING ($n=50$)



Halfway through second episode

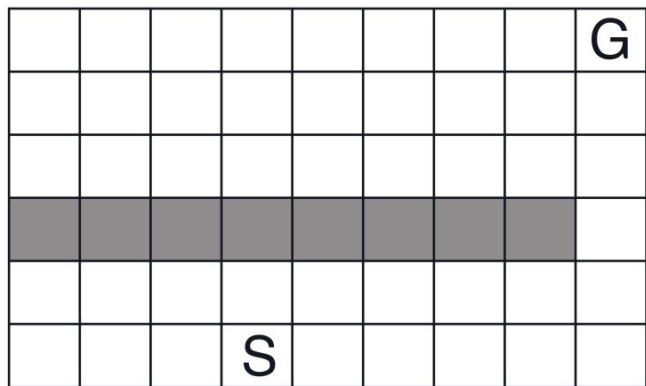
Black square ■ : location of the agent

Possibility of a Wrong Model

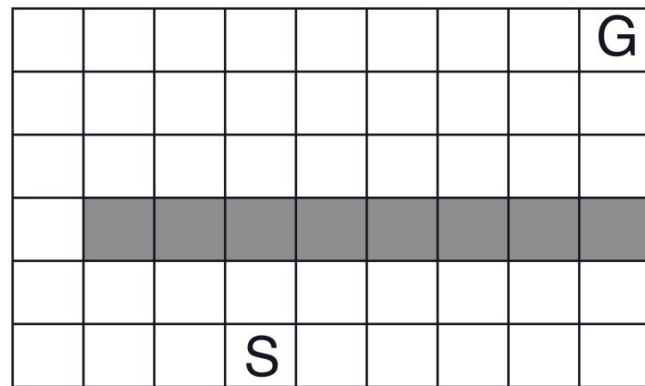
- Model can be wrong for various reasons:
 - Stochastic environment & limited number of samples
 - Function approximation
 - Environment has changed
- Can lead to suboptimal policy

Optimistic Wrong Model: Blocking Maze Example

- Agent can correct modelling error for optimistic models
 - Agent attempts to “exploit” false opportunities
 - Agent discovers they do not exist

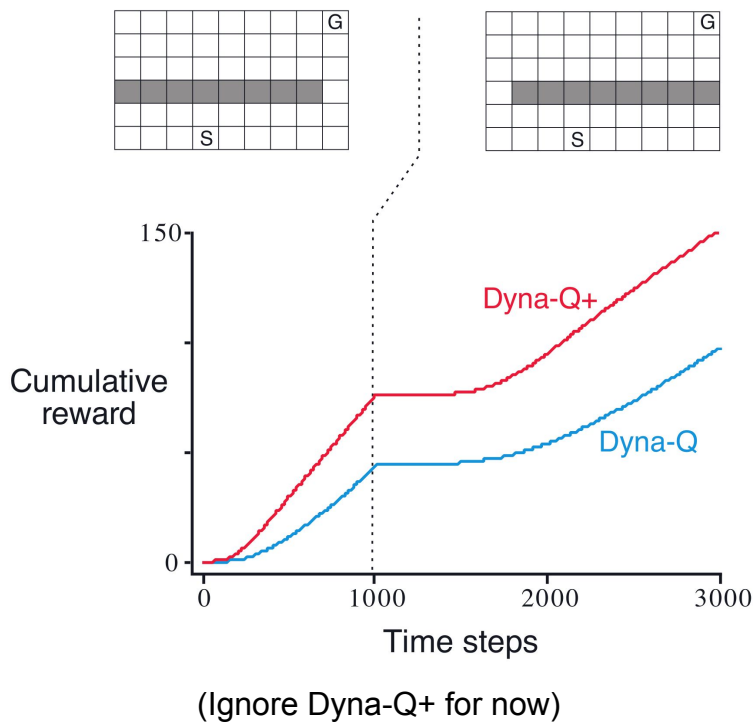


⋮



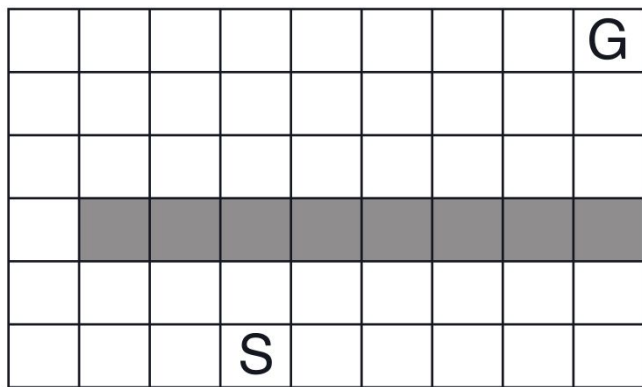
Environment changes after 1000 timesteps

Optimistic Wrong Model: Blocking Maze Example

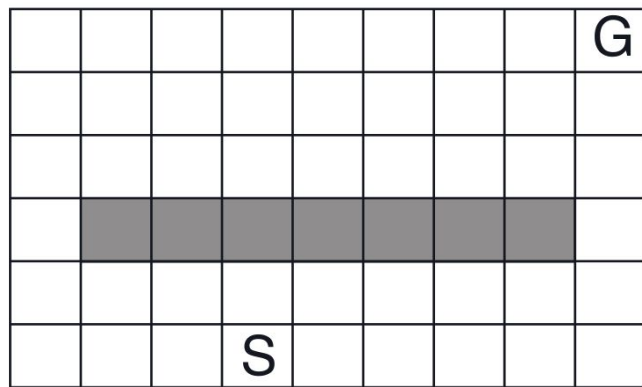


Pessimistic Wrong Model: Shortcut Maze Example

- Agent might never learn modelling error for optimistic models
 - Agent never realizes a better path exists
 - Unlikely even with an ϵ -greedy policy

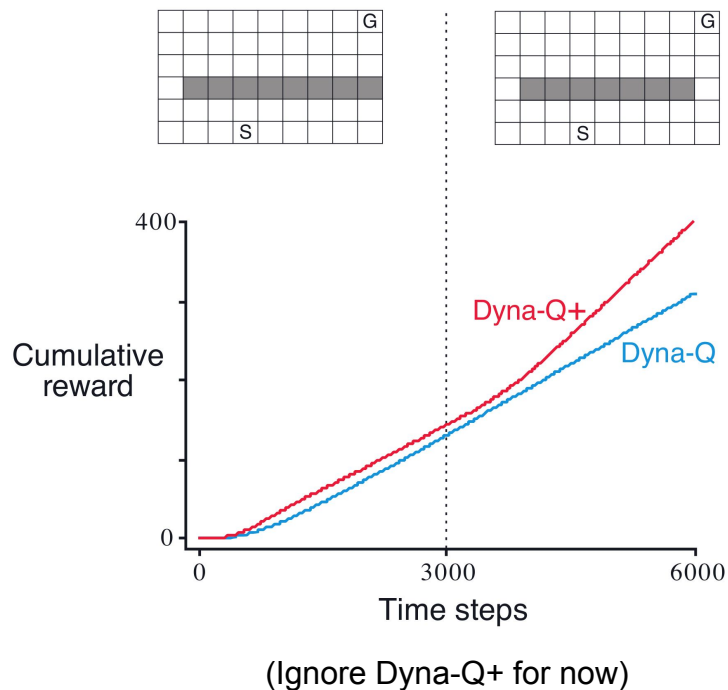


⋮



Environment changes after 1000 timesteps

Pessimistic Wrong Model: Shortcut Maze Example

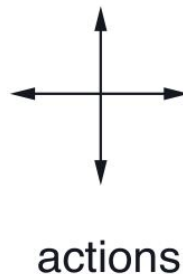
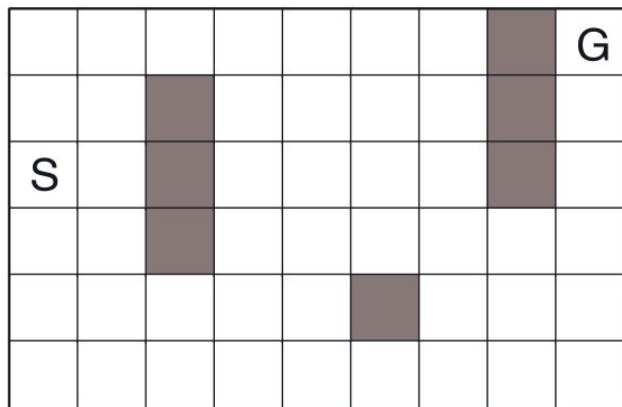


Dyna-Q+

- Need to balance exploration and exploitation
 - Keep track of elapsed time τ since last visit for each state-action pair
 - Add bonus reward $\kappa\sqrt{\tau}$ during *planning*
 - Allow untried state-action pair to be visited in *planning*
-
- Costly, but the computational curiosity is worth the cost

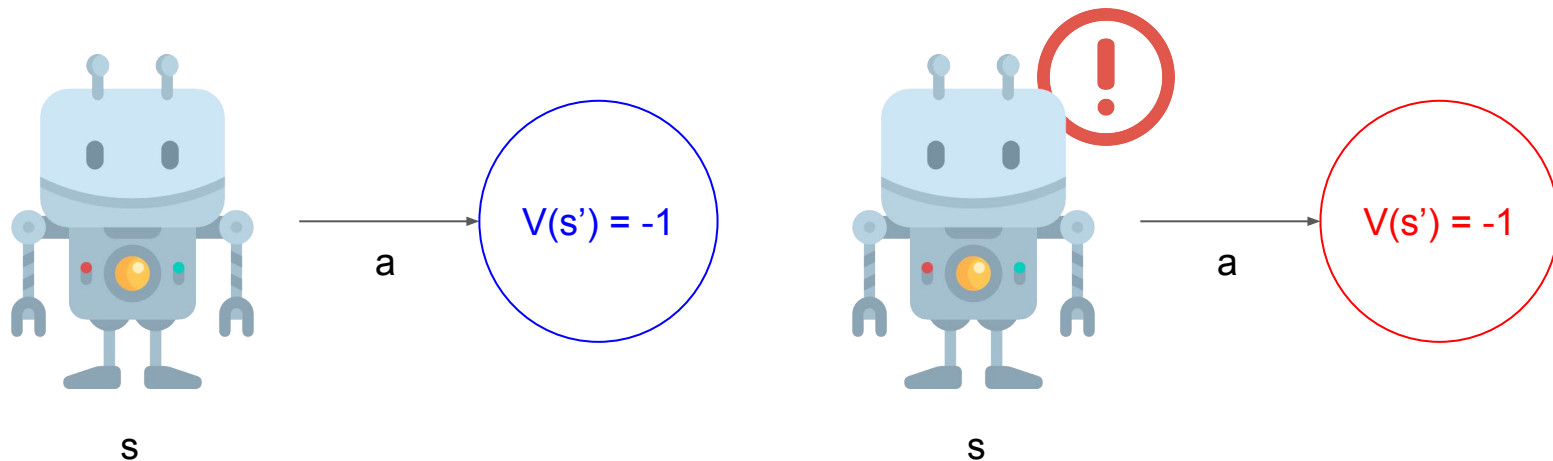
Prioritized Sweeping: Intuition

- *Focused* simulated transitions and updates can make planning more efficient
 - At the start of second episode, only the penultimate state has nonzero value estimate
 - Almost updates do nothing



Prioritized Sweeping: Backward Focusing

- Intuition: work backward from “goal states”
- Work back from any state whose value changed
 - Typically implies other states' values also changed
- Update predecessor states of changed state



Prioritized Sweeping

- Not all changes are equally useful
 - magnitude of change in value
 - transition probabilities
- Prioritize updates via *priority queue*
 - Pop max-priority pair and update
 - Insert all predecessor pairs with effect above some small threshold
 - (Only keep higher priority if already exists)
 - Repeat until quiescence

Prioritized Sweeping: Pseudocode

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 :

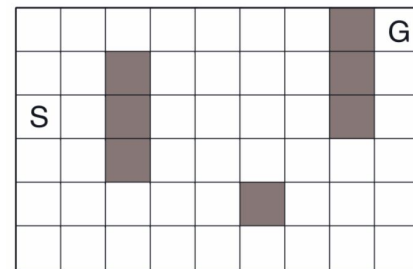
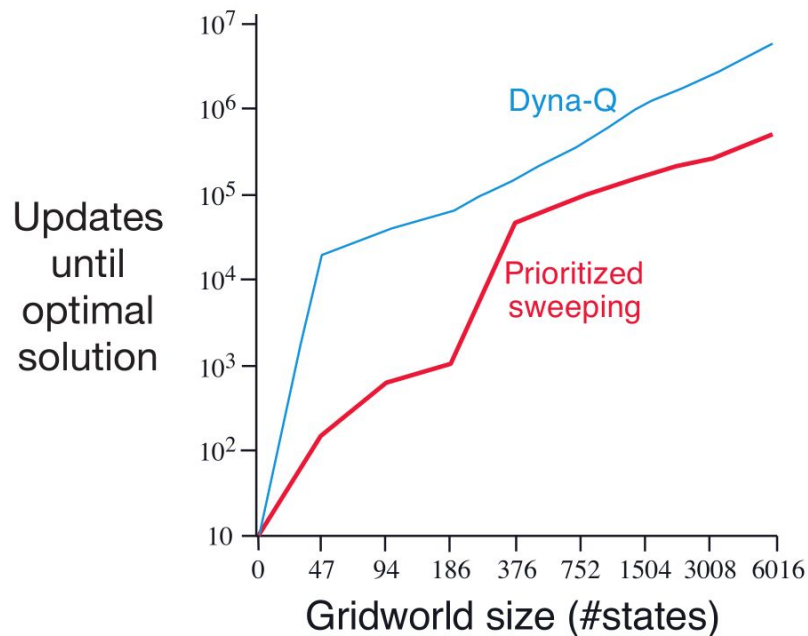
$\bar{R} \leftarrow$ predicted reward for \bar{S}, \bar{A}, S

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

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

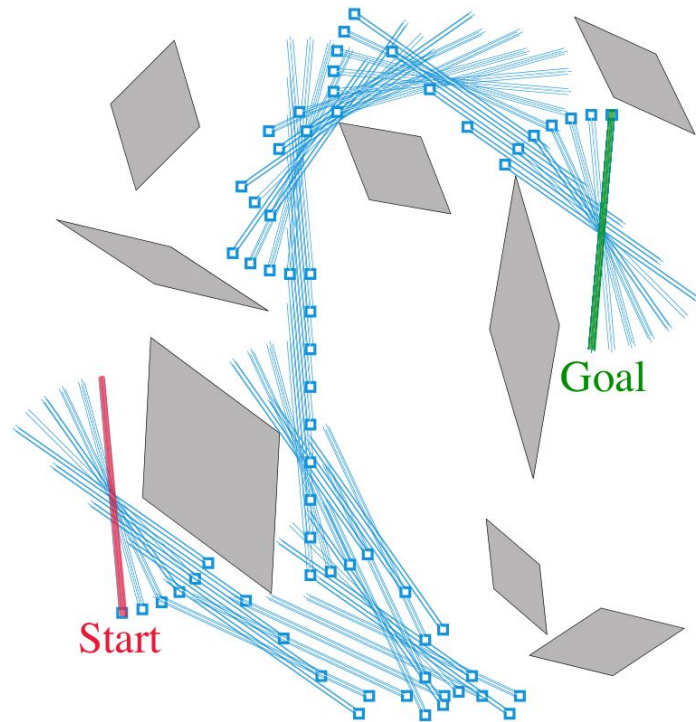
Prioritized Sweeping: Maze Example

- Decisive advantage over unprioritized Dyna-Q



Prioritized Sweeping: Rod Maneuvering Example

- Maneuver rod around obstacles
 - 14400 potential states, 4 actions (translate, rotate)
- Too large to be solved without prioritization



Prioritized Sweeping: Limitations

- Stochastic environment
 - Use expected update instead of sample updates
 - Can waste computation on low-probability transitions
- How about sample updates?

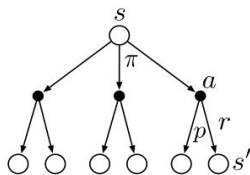
Expected vs Sample Updates

- Recurring theme throughout RL
- Any can be used for planning

Value
estimated

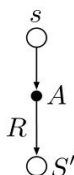
$v_{\pi}(s)$

Expected updates
(DP)



policy evaluation

Sample updates
(one-step TD)

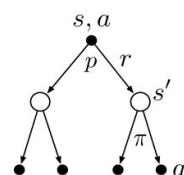


TD(0)

Value
estimated

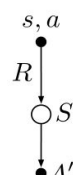
$q_{\pi}(s, a)$

Expected updates
(DP)



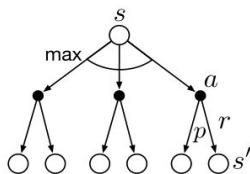
q-policy evaluation

Sample updates
(one-step TD)



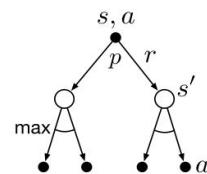
Sarsa

$v_{*}(s)$

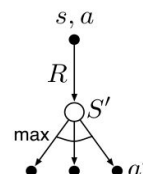


value iteration

$q_{*}(s, a)$



q-value iteration



Q-learning

Expected vs Sample Updates

- Expected updates yields better estimate

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

- Sample updates requires less computation

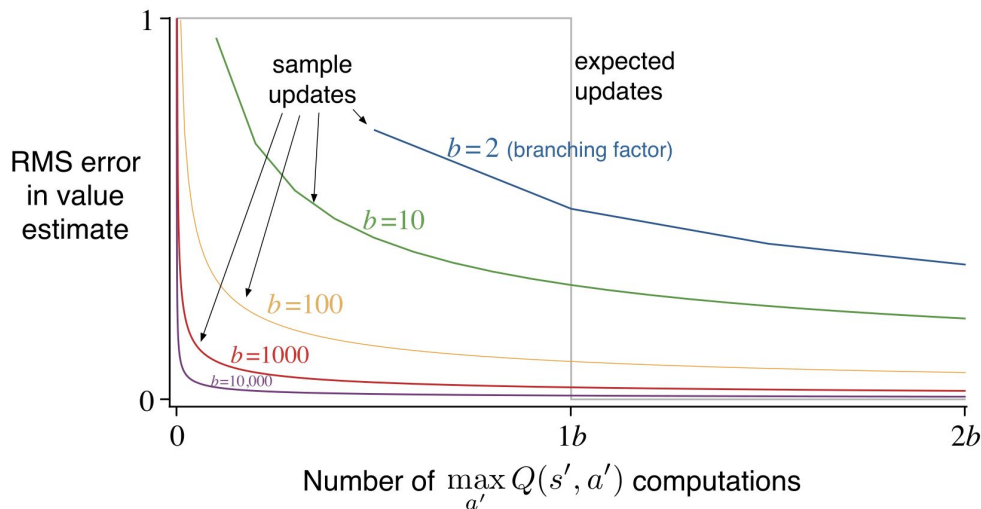
$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R + \gamma \max_{a'} Q(S', a') - Q(s, a) \right],$$

Expected vs Sample Updates: Computation

- *Branching factor d* : number of possible next states
 - Determines computation needed for expected update
 - $T(\text{Expected update}) \approx d * T(\text{Sample update})$
- If enough time for expected update, resulting estimate is usually better
 - No sampling error
- If not enough time, sample update can at least somewhat improve estimate
 - Smaller steps

Expected vs Sample Updates: Analysis

- Sample updates reduce error according to $\sqrt{\frac{b-1}{bt}}$
- Does not account sample update having better estimate of successor states



Distributing Updates

1. Dynamic Programming

- Sweep through entire state space (or state-action space)

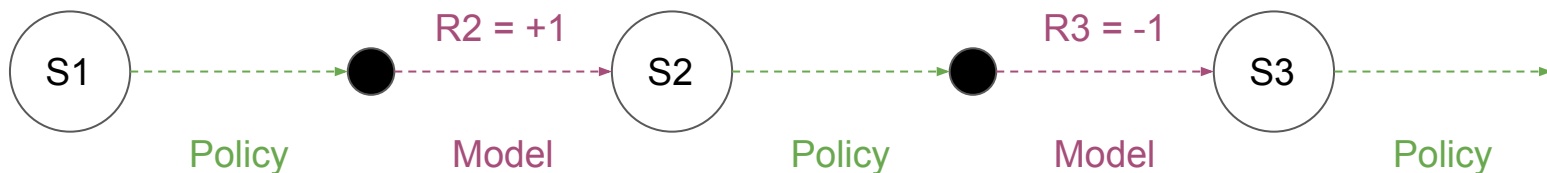
2. Dyna-Q

- Sample uniformly

- Both suffers from updating irrelevant states most of the time

Trajectory Sampling

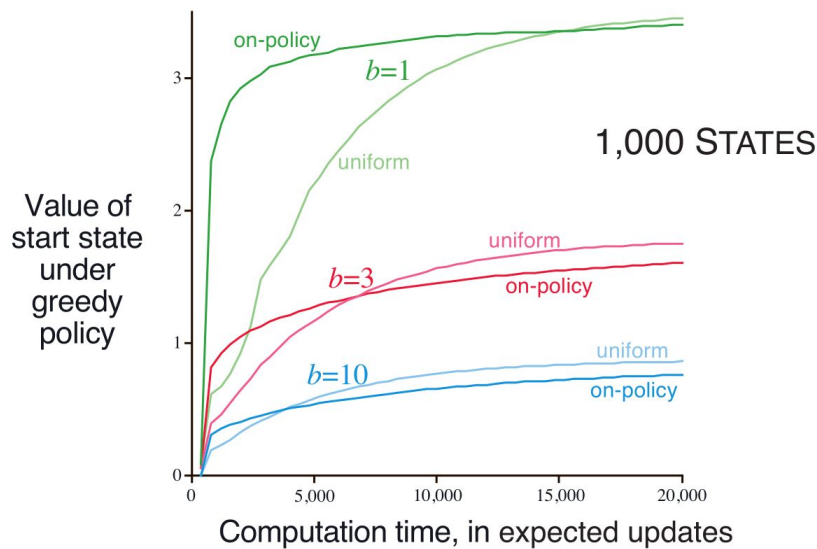
- Gather experience by sampling explicit individual trajectories
 - Sample state transitions and rewards from the model
 - Sample actions from a **distribution**



- Effects of **on-policy distribution**
 - Can ignore vast, uninteresting parts of the space
 - Significantly advantageous when function approximation is used

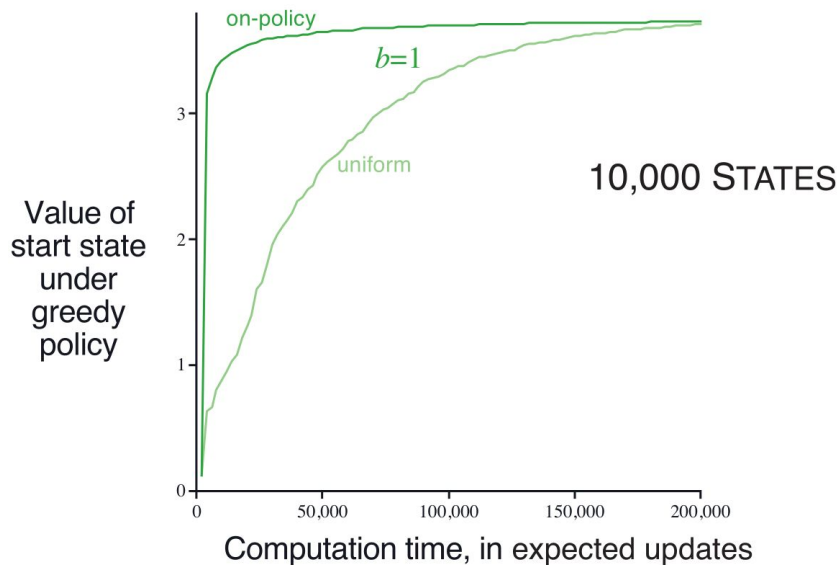
Trajectory Sampling: On-policy Distribution Example

- Faster planning initially, but retarded in the long run
- Better when branching factor b is small (can focus on just few states)



Trajectory Sampling: On-policy Distribution Example

- Long-lasting advantage when state space is large
 - Focusing on states have bigger impact when state space is large



Real-time Dynamic Programming (RTDP)

- On-policy trajectory-sampling value iteration algorithm
- Gather real or simulated trajectories
 - *Asynchronous DP*: nonsystematic sweeps

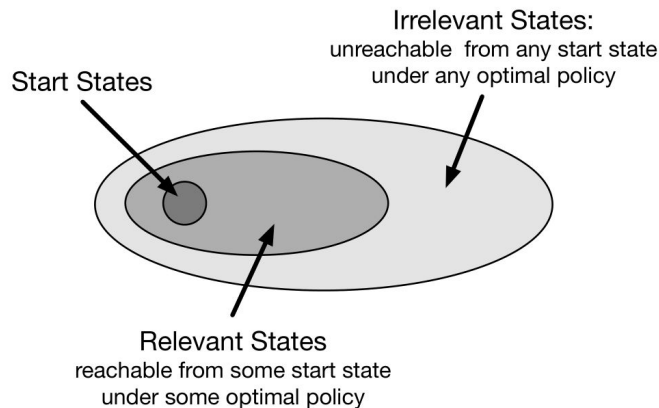


- Update with value iteration

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

RTDP: Prediction and Control

- Prediction problem: skip any state not reachable by policy
- Control problem: find the *optimal partial policy*
 - A policy that is optimal for relevant states but arbitrary for other states
 - Finding such policy requires visiting all (s, a) infinitely many times



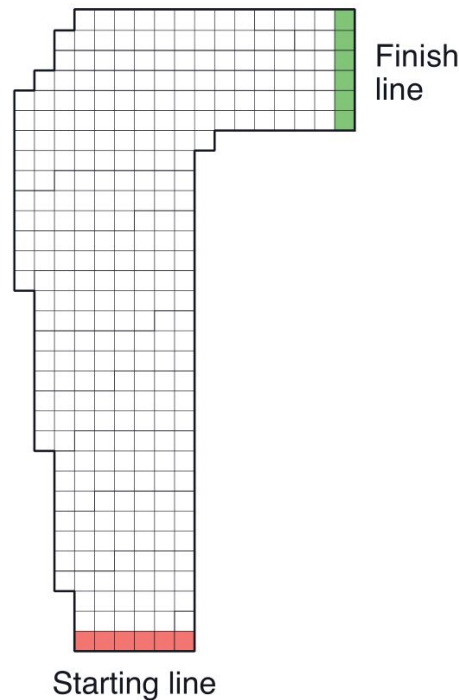
Stochastic Optimal Path Problems

- Conditions
 - Undiscounted episodic task with absorbing goal states
 - Initial value of every goal state is 0
 - At least one policy can definitively reach a goal state from any starting state
 - All rewards from non-goal states are negative
 - All initial values are optimistic
- Guarantees
 - RTDP does not need to visit all (s, a) infinite times to find *optimal partial policy*

Stochastic Optimal Path Problem: Racetrack

- 9115 reachable states, 599 relevant
- RTDP needs half the update of DP
 - Visits almost all states at least once
 - Focuses to relevant states quickly

	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



DP vs. RTDP: Checking Convergence

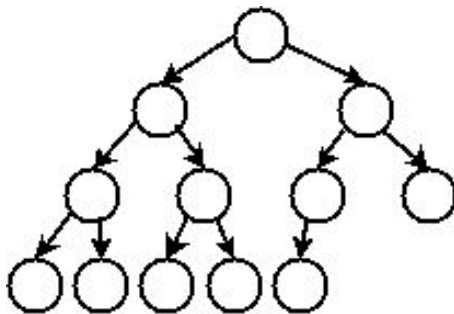
- DP: Update with exhaustive sweeps until Δv is sufficiently small
 - Unaware of policy performance until value function has converged
 - Could lead to overcomputation
- RTDP: Update with trajectories
 - Check policy performance via trajectories
 - Detect convergence earlier than DP

Background Planning vs. Decision-Time Planning

- *Background Planning*
 - Gradually improve policy or value function
 - Not focused on the current state
 - Better when low-latency action selection is required
- *Decision-Time Planning*
 - Select single action through planning
 - Focused on the current state
 - Typically discard value / policy used in planning after each action selection
 - Most useful when fast responses are not required

Heuristic Search

- Generate tree of possible continuations for each encountered states
 - Compute best action with the search tree
 - More computation is needed
 - Slower response time
- Value function can be held constant or updated
- Works best with perfect model and imperfect Q



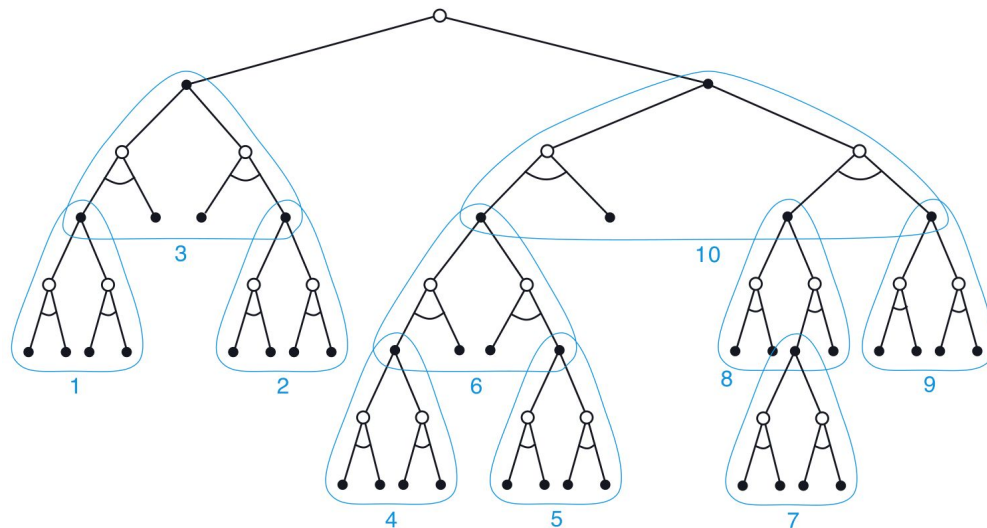
Heuristic Search: Focusing States

- Focus on states that might immediately follow the current state
 - Computations: Generate tree with current state as head
 - Memory: Store estimates only for relevant states
- Particularly efficient when state space is large (ex. chess)



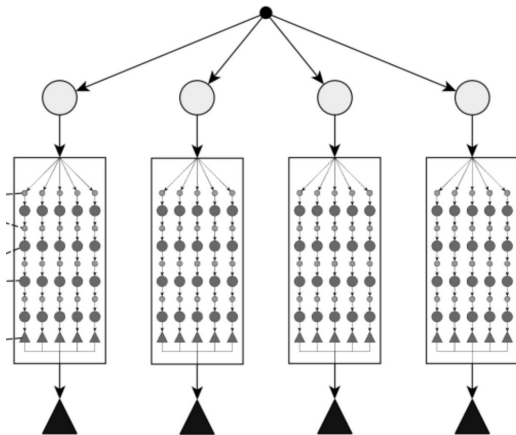
Heuristic Search: Focusing Updates

- Focus distribution of *updates* on the current state
 - Construct search tree
 - Perform one-step updates from bottom-up



Rollout Algorithms

- Estimate Q by averaging simulated trajectories from a *rollout policy*
- Choose action with highest Q
- Does not compute Q for all states / actions (unlike MC control)
- Not a *learning* algorithm since values and policies are not stored



Rollout Algorithms: Policy

- Satisfies the *policy improvement theorem* for the policy π
 - Same as one step of the policy iteration algorithm

$$v_{\pi'}(s) = q_{\pi}(s, a) \geq v_{\pi}(s)$$

- Rollout seeks to improve upon the default policy, not to find the optimal policy
 - Better default policy \rightarrow Better estimates \rightarrow Better policy from rollout algorithm

Rollout Algorithms: Time Constraints

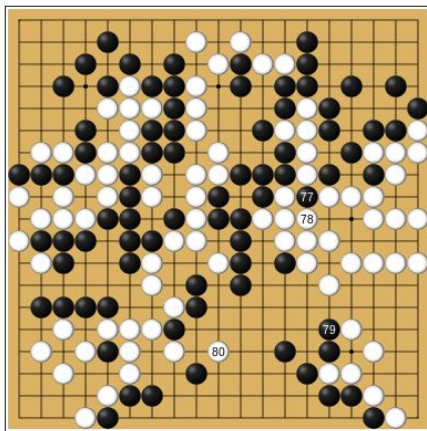
- Good rollout policies need a lot of trajectories
- Rollout algorithms often have strict time constraints

Possible Mitigations

1. Run trajectories on separate processors
2. Truncate simulated trajectories before termination
 - Use stored evaluation function
3. Prune unlikely candidate actions

Monte Carlo Tree Search (MCTS)

- Rollout algorithm with *directed simulations*
 - Accumulate value estimates in a tree structure
 - Direct simulations toward more high-rewarding trajectories
- Behind successes in Go

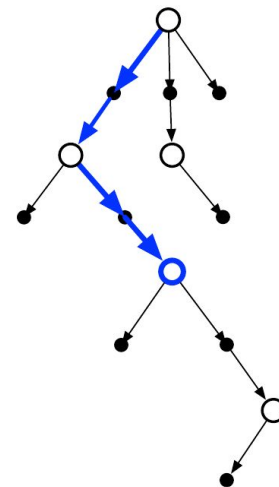


Monte Carlo Tree Search: Algorithm

- Repeat until termination:
 - a. **Selection:** Select beginning of trajectory
 - b. **Expansion:** Expand tree
 - c. **Simulation:** Simulate an episode
 - d. **Backup:** Update values
- Select action with some criteria
 - a. Largest action value
 - b. Largest visit count

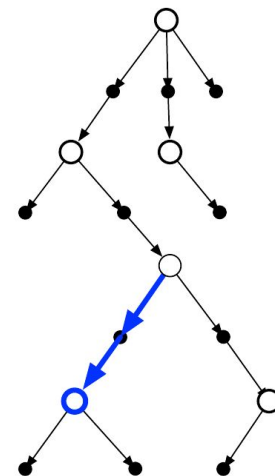
Monte Carlo Tree Search: Selection

- Start at the root node
- Traverse down the tree to select a leaf node



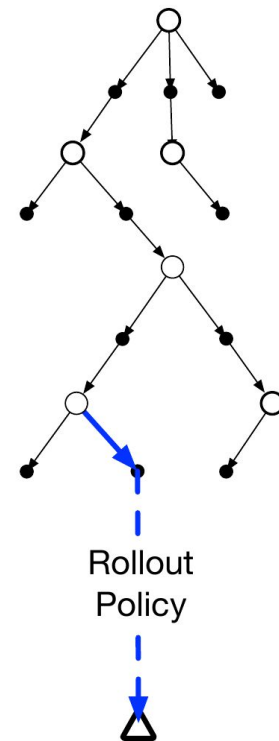
Monte Carlo Tree Search: Expansion

- Expand the selected leaf node
 - Add one or more child nodes via unexplored actions



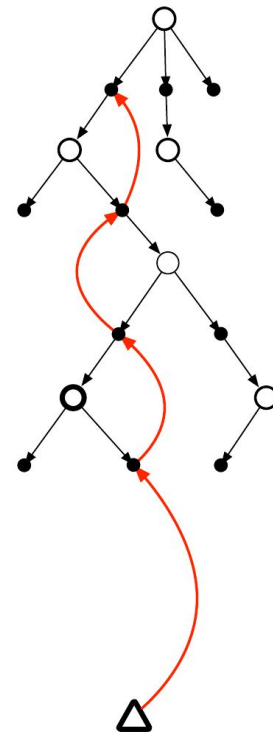
Monte Carlo Tree Search: Simulation

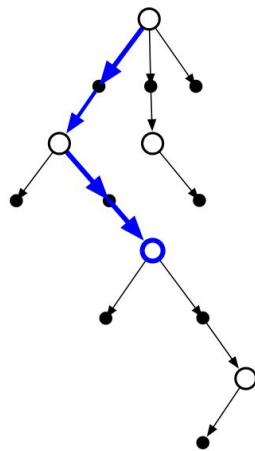
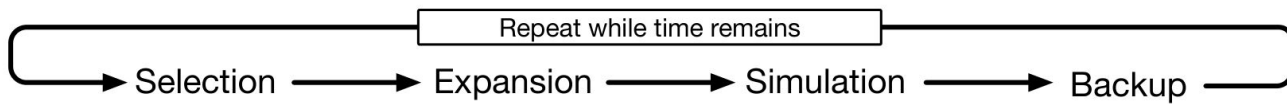
- From leaf node or new child node, simulate a complete episode
- Generates a Monte Carlo trial
 - Selected first by the tree policy
 - Selected beyond the tree by the rollout policy



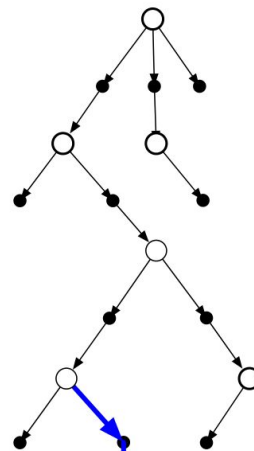
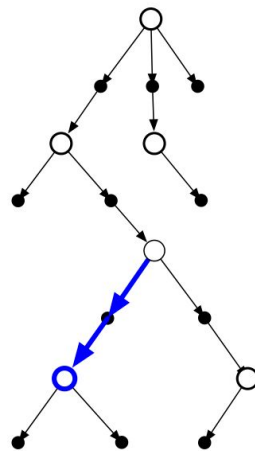
Monte Carlo Tree Search: Backup

- Update or Initialize values of nodes traversed in *tree policy*
 - No values saved for the rollout policy beyond the tree

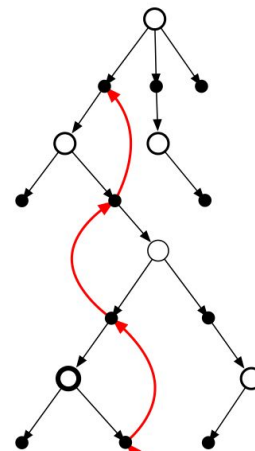




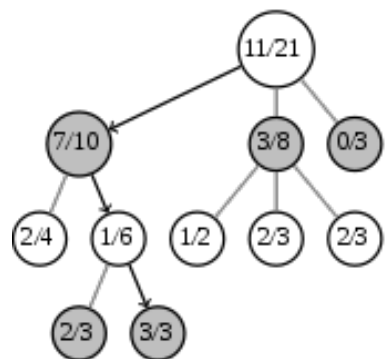
Tree
Policy



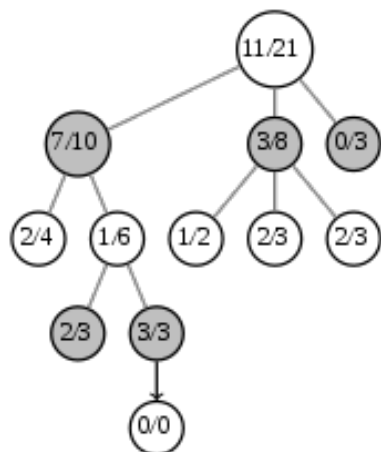
Rollout
Policy



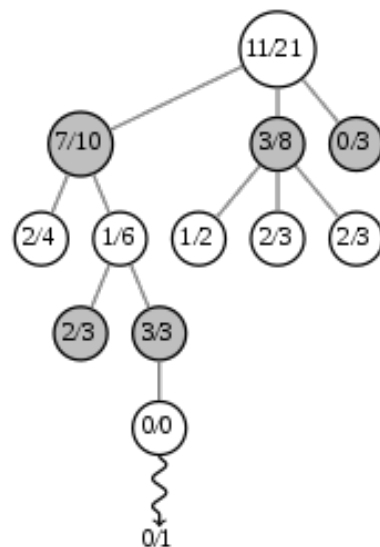
Selection



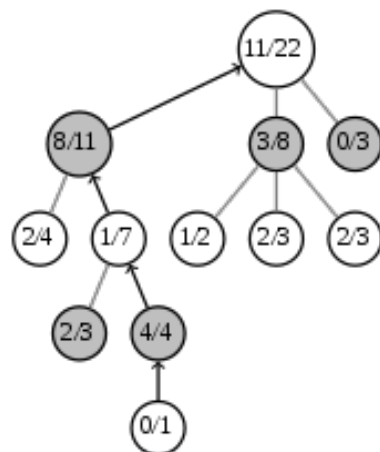
Expansion



Simulation



Backpropagation



Monte Carlo Tree Search: Insight

- Can use online, incremental, sample-based methods
- Can focus MC trials on segments with high-return trajectories
- Can efficiently grow a partial value table
 - Does not need to save all values
 - Does not need function approximation

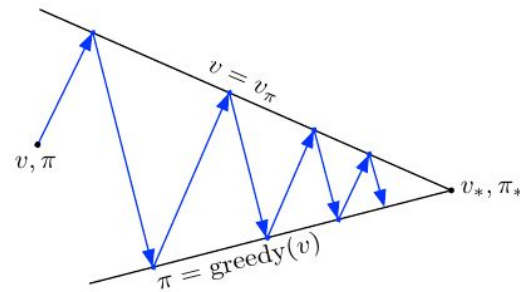
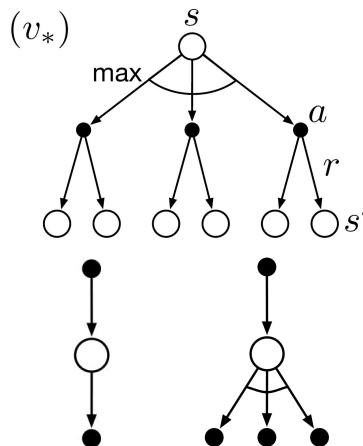
Summary of Chapter 8

- Planning requires a *model* of the environment
 - *Distribution model* consists of transition probabilities
 - *Sample model* produces single transitions and rewards
- Planning and Learning share many similarities
 - Any learning method can be converted to planning method
- Planning can vary in *size* of updates
 - ex) 1-step sample updates
- Planning can vary in *distribution* of updates
 - ex) Prioritized sweeping, On-policy trajectory sampling, RTDP
- Planning can *focus forward* from pertinent states
 - Decision-time planning
 - ex) Heuristic search, Rollout algorithms, MCTS

Summary of Part I

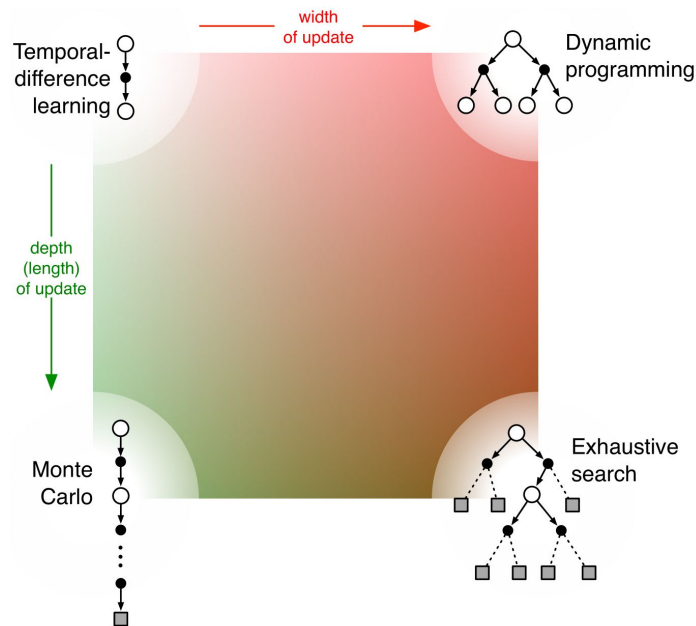
- Three underlying ideas:
 - a. Estimate value functions
 - b. Back up values along actual / possible state trajectories
 - c. Use Generalized Policy Iteration (GPI)
 - Keep approximate value function and policy
 - Use one to improve another

$$V_{\pi} \longrightarrow v_{\pi}$$



Summary of Part I: Dimensions

- Three important dimensions:
 - a. Sample update vs Expected update
 - Sample update: Using a sample trajectory
 - Expected update: Using the distribution model
 - b. Depth of updates: degree of bootstrapping
 - c. On-policy vs Off-policy methods
- One undiscussed important dimension:
Function Approximation



Summary of Part I: Other Dimensions

- Episodic vs. Continuing returns
- Discounted vs. Undiscounted returns
- Action values vs. State values vs. Afterstate values
- Exploration methods
 - ϵ -greedy, optimistic initialization, softmax, UCB
- Synchronous vs. Asynchronous updates
- Real vs. Simulated experience
- Location, Timing, and Memory of updates
 - Which state or state-action pair to update in model-based methods?
 - Should updates be part of selected actions, or only afterward?
 - How long should the updated values be retained?

Thank you!

Original content from

- [Reinforcement Learning: An Introduction by Sutton and Barto](#)

You can find more content in

- [github.com/seungjaeryanlee](#)
- [www.endtoend.ai](#)