8.0 : Intro

Model: essentially is a representation of how the environment will behave.

TD, MC ⟶ DP

model-free: computational efficiency ⟶ sample efficiency

model-based: works when you already know the model. You can think of it as the agent deciding what to do without having to act: planning.

- Model-based learning (DP, heuristic search): rely on planning.
- Model-free learning (MC, TD): rely on learning.

Similarities:

- Both revolve around the computation of value functions
- Both look ahead of future events, compute a backed-up value and use it as an update target for an value function.

In this chapter we explore how model-free and model-based approaches can be combined.

8.1: Models and Planning

Models:

A model is anything the agent can use to predict the environment's behavior.

- Given a state and an action, a model predicts the next state and the next reward.
- If the model is stochastic, we have several next possible states and rewards. The model can:
  - Output the whole probability distribution over the next states and rewards: distribution models.
  - Produce only one of the possibility (the one w/ highest probability): sample models.
- A model can be used to simulate the environment and produce simulated experience.

Planning: refers to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment.
Planning uses model to generate simulated experience for additional learning.

model $\xrightarrow{\text{planning}}$ policy.

We can distinguish 2 types of planning:

- state-space planning: search thru state space for an optimal policy. Action cause transitions.
- plan-space planning: search thru the space of plans. Operators transform one plan into another (ecolutionary methods, partial-order planning).

we focus on state-space planning, that share a common structure:

model → simulated experience → (backups) → values → policy

1. All state-space planning methods involve computing value functions as key intermediate towards improving the policy.
2. They compute value functions by updates of backup operations from simulated experience.

EX: DP make sweeps thru the space of states, generating for each the distribution of possible transitions. Each distribution is then used to compute a backup value (update target) and update the state's estimated value.

- planning uses simulated experience generated by the model.
- learning uses real experience generated by the environment.

There are differences but in many cases a learning algorithm can be substituted for the key update step of a planning method, because they apply just as well to simulated experience. Let's see an example with an algorithm we know: Q-learning.

random-sample one-step tabular Q-learning:

Planning method based on Q-learning and random samples from a sample model:

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

In practice, we observe planning with very small steps seems to be the most efficient approach.

8.2: Dyna: Integrated planning, acting, and learning

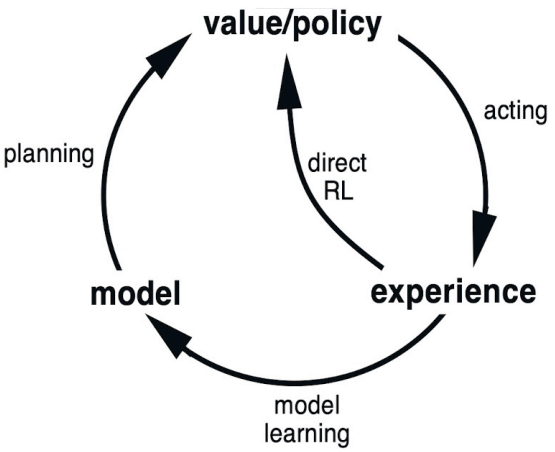
When planning is done online, while interacting w/ the environment, a number of issues arise.

- new information may change the model (and thus planning).
- how to divide the computing resources btwn decision making and model learning?

Dyna-Q: Simple architecture integrating the major functions of an online planning agent.

What do we do with real experience;

- model-learning: improve the model using real experience (to make it more accurately match the environment).
- direct RL: improve the value functions and policy using real experiences from interacting w/ the environment.
- Indirect RL/Planning: improve the value functions and policy via the model



Dyna-Q interactions

Both direct and indirect method have advantages and disadvantages.

- Indirect methods often make fuller use of limited amount of experience, - how? using planning.

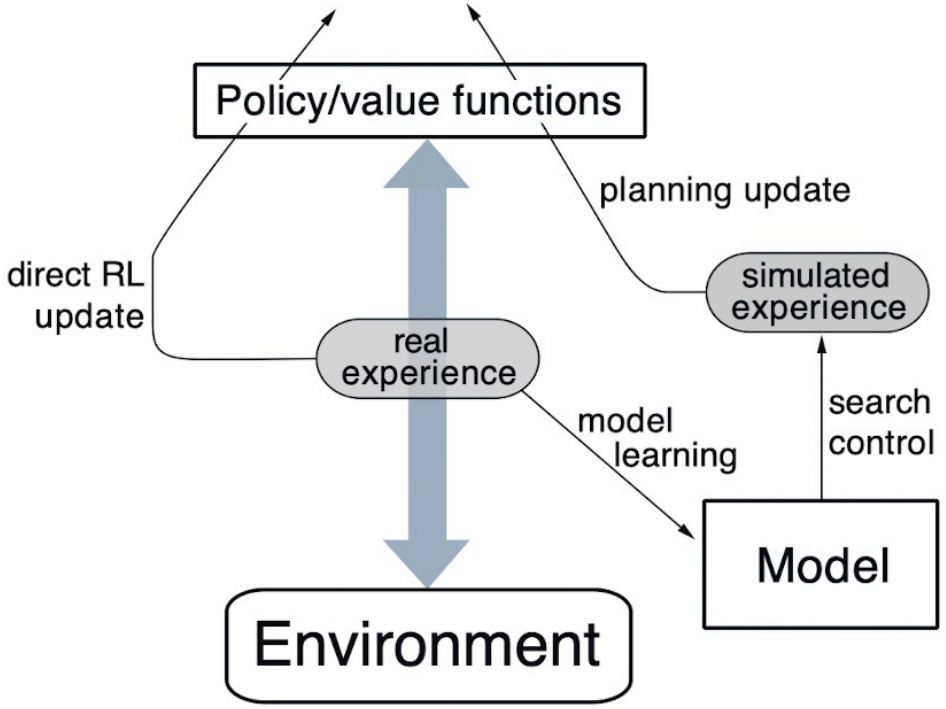
- Direct methods are often simpler and not affected by bias in the design of the model.

Dyna-Q:

Dyna-Q includes all of the processes shown in the diagram: planning, acting, direct RL, and model learning.

- the planning method is the random sample one-step tabular Q-learning shown above.
- The direct RL method is one-step tabular Q-learning.
- The model learning method is table-based and assumes the environment is deterministic.
  - After each transition $S_t, A_t \rightarrow S_{t+1}, R_{t+1}$, the model records in its table entry for S_t, A_t, the prediction that S_{t+1}, R_{t+1} will follow.
  - If the model is queries with a state-action pair it has seen before, it simply returns the last S_{t+1}, R_{t+1} experienced.
  - During planning, the Q-learning algorithm samples only from state-actions the model has seen.

Overall architecture of Dyna-Q agents:



- Search control: process that selects starting states and actions from the simulated experiences.
- planning: is achieved by applying RL methods to simulated experience.

Typically, as in Dyna-Q, the same algorithm is used for direct learning and for planning. Learning and planning are deeply integrated, differing only in the source of their experience.

Conceptually, planning, acting, direct RL and model-learning all occur in parallel. But for concreteness and implementation on a computer, we specify the order:

- Acting, model-learning, and direct RL require little computation.
- The remaining time in each step is devoted to planning which is inherently computationally intensive.

Algorithm:

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ // $Q(s,a)$ = Q-table.

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state // start at a current state

(b) $A \leftarrow \varepsilon$-greedy(S,Q) // take ε-greedy action

(c) Take action A; observe resultant reward, R, and state, S'

(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$ // direct learning

(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment) // model learning

(f) Loop repeat n times: // planning

$S \leftarrow$ random previously observed state // randomly sample previously visited S, A

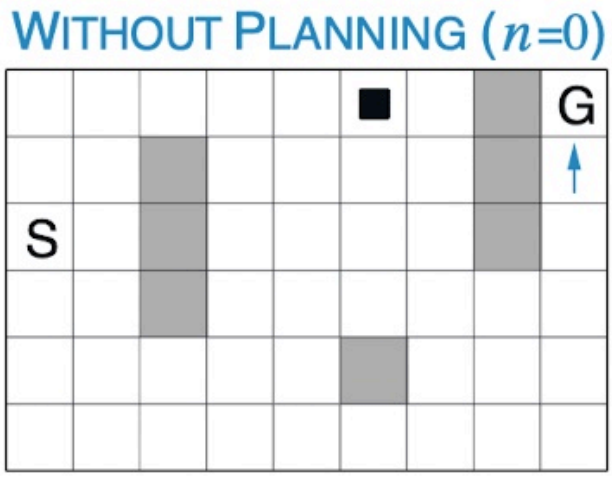
$A \leftarrow$ random action previously taken in S

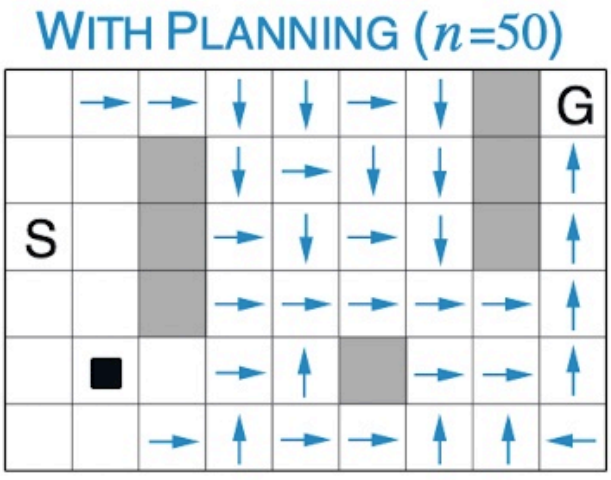
$R, S' \leftarrow Model(S,A)$ // use model to get simulated next state and reward

$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$ // update Q-table.

- Model(S, a) denotes the content of the predicted S_{t+1} and R_{t+1} for (S, a).
- Direct RL, model-learning, planning are steps d, e, f respectively.
- If e and f were omitted, the algorithm would be one-step tabular Q-learning.

A simple maze example shows that adding the planning $n > 0$ element dramatically improves the agent behavior.





Intermixing planning and acting: make them both proceed as fast as they can;

- The agent is always reactive, responding with the latest sensory information.
- planning is always done in the background.
- model learning is always done in the background too. As new information is gained, the model is updated.

As the model changes, the planning process will gradually compute a different way of behaving to match the newer model.

8.3: When the model is wrong

Model may be incorrect because:

- The environment is stochastic and only a limited amount of experience is available. Stochastic error.
- The model has learned using a function approximation that failed to generalize.
- The environment has changed and new behavior has not been observed yet: Distribution shift.

In some cases, following the suboptimal policy computed over a wrong model quickly leads to the discovery and correction of the modelling error, for example, we will try to reach an expected reward that does not exist.

Exploration/Exploitation tradeoff: we want the agent to explore to find changes in the environment but not so much that the performance is greatly degraded. Simple heuristics are often effective.

Dyna-Q+ : Exploration bonus.

Dyna-Q+ uses one such heuristics. This agent keeps track for each state-action pair of how many times steps have elapsed since it was last tried in a real interaction with the environment. A special bonus reward is added to simulated experience involving these actions. If the modeled reward for a transition is r, and the transition has not been tried for over τ time steps, then planning updates are done as if the transition produced a reward of $r + k\sqrt{\tau}$ for some small k. This encourages the agent to make some tests, which has a cost, but often is worth it.

8.4: Prioritized Sweeping

So far for the dyna agents, simulated transitions are started in state action pairs selected uniformly at random from all previously experienced state-action pairs. Uniform selection is usually not the best, and we'd better focus on particular state action pairs.

Backward-focusing of planning computations: in general, we want to work backward not just from goal states but from any state whose value have changed.

As the frontier of useful updates propagate backward, it grows rapidly, producing many state-action pairs that could be usefully updated. But not all of them are equally useful. The value of some states have changed a lot, and the value of their predecessors are also likely to change a lot. It is thus natural to prioritize according to a measure of urgency, and perform them in order of priority.

This is the idea behind prioritized sweeping. A queue is maintained for every state-action pair whose value would change if updated, prioritized by the size of their change.

Prioritized sweeping has been found to drastically increase the speed of convergence.

Algorithm:

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

What about stochastic environments?
The model is maintained by keeping counts of the number of times each state-action pair has been experienced and what their next states were. We can update each state-action pair not with a sample update but with an expected update, taking into account the possibilities of the next states and their probability of occurring.

using expected updates can waste a lot of computation on low probability transitions. Often using samples help converge faster despite the added variance. Sample updates can be better because break the overall backing-up computation into smaller pieces (individual transitions) which enables them to be more focused on the pieces that would have the greater impact.

Summary:

we have suggested in this chapter that all kinds of state-space planning can be viewed as sequences of value updates, varying only in:

- the type of update, expected or sample, large or small,
- the order in which the updates are done.

we have focused on backward focusing, but this is only one strategy. Another could be to focus on states according to how easy they can be reached from states that are frequently visited under current policy, which can be called forward focusing.

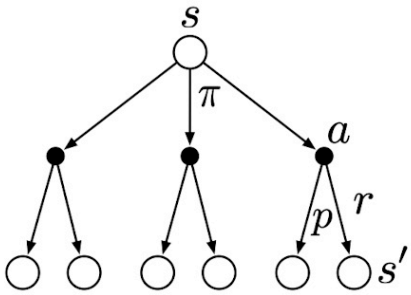
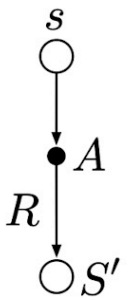
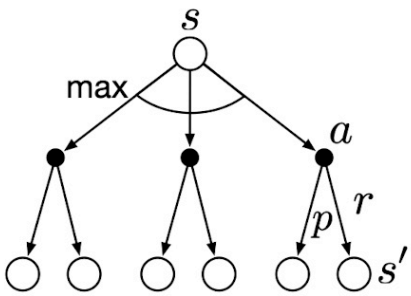
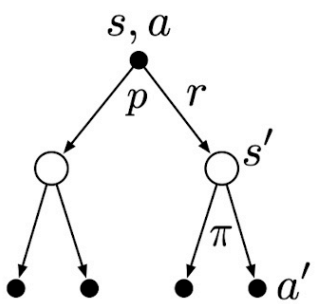
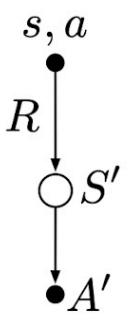
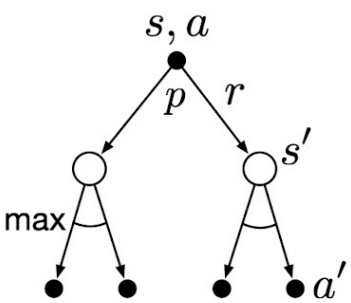
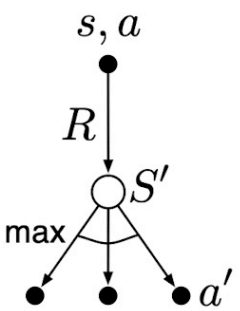
8.5: Expected vs Sample updates

The examples of the previous sections gives an idea of how planning and learning can be combined. Now we'll focus on each of the components, starting with expected vs sample update.

we have considered many value-function updates. If we focus on one-step updates, they vary along 3 dimensions.

- update state value or action values.
- Estimate the value of the optimal policy or any given policy.
- The updates are expected updates (consider all possible events) or sample updates (consider a sample of what might happen).

These 3 binary dimensions give rise to 8 cases, 7 of which are specific algorithms. The 8th case does not seem to give rise to any specific algorithm.

Value estimated	Expected updates (DP)	Sample updates (one-step TD)
$v_\pi(s)$ (prediction)	 policy evaluation (DP prediction)	 TD(0) (TD control)
$v_*(s)$	 value iteration (DP control)	
$q_\pi(s,a)$	 q-policy evaluation (DP prediction)	 Sarsa (TD control, on-policy)
$q_*(s,a)$	 q-value iteration (DP control)	 Q-learning (TD control, off-policy)

Any of these one-step method can be used in planning methods.

- The Dyna-Q discussed before uses q_* sample updates but could use q_* expected update or q_π updates.
- The Dyna-AC system uses V_π sample updates w/ a learning policy structure.
- For stochastic problems, prioritized sweeping is always done using one of the expected updates.

When we don't have the environmental model, we can't do an expected model so we use a sample.

Expected update:

- yield a better estimate bc they are uncorrupted by sampling error.
- but require more computation.

computational requirements

For q_* approximation in this set up:

- discrete states and actions,
- table-lookup representation of the approximate value function Q.
- a model in the form of estimated dynamics $\hat{p}(s', r \mid s, a)$.

The expected update is:

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

The sample update is the Q-learning like update:

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

The difference btwn these expected and sample updates is significant to the extent that the environment is stochastic. Many possible next states = many computation for the expected update.

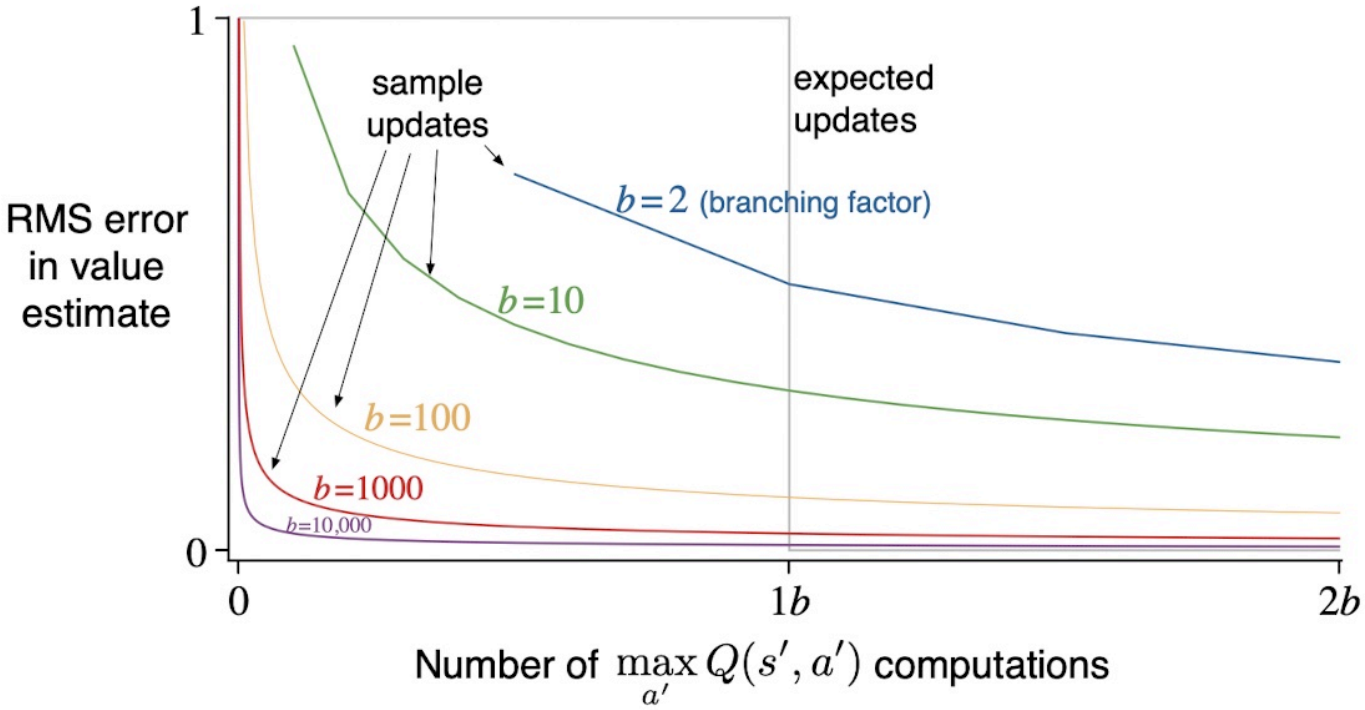
For a particular set (s,a), let b be the branching factor, aka the # of possible next states. Then an expected update of this pair requires roughly b times as much computation as a sample update.

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

For moderately large b the error falls dramatically with a tiny fraction of b updates. For these cases, many state-action pairs could have their values improved dramatically in the same time that a single action-pair could undergo an expected update.

By causing estimates to be more accurate sooner, sample updates will have an advantage that the values backed up from successor states will be more accurate.

TLDR; Sample updates are better than expected updates on problems with a large branching factor and many states.



8.6: Trajectory Sampling

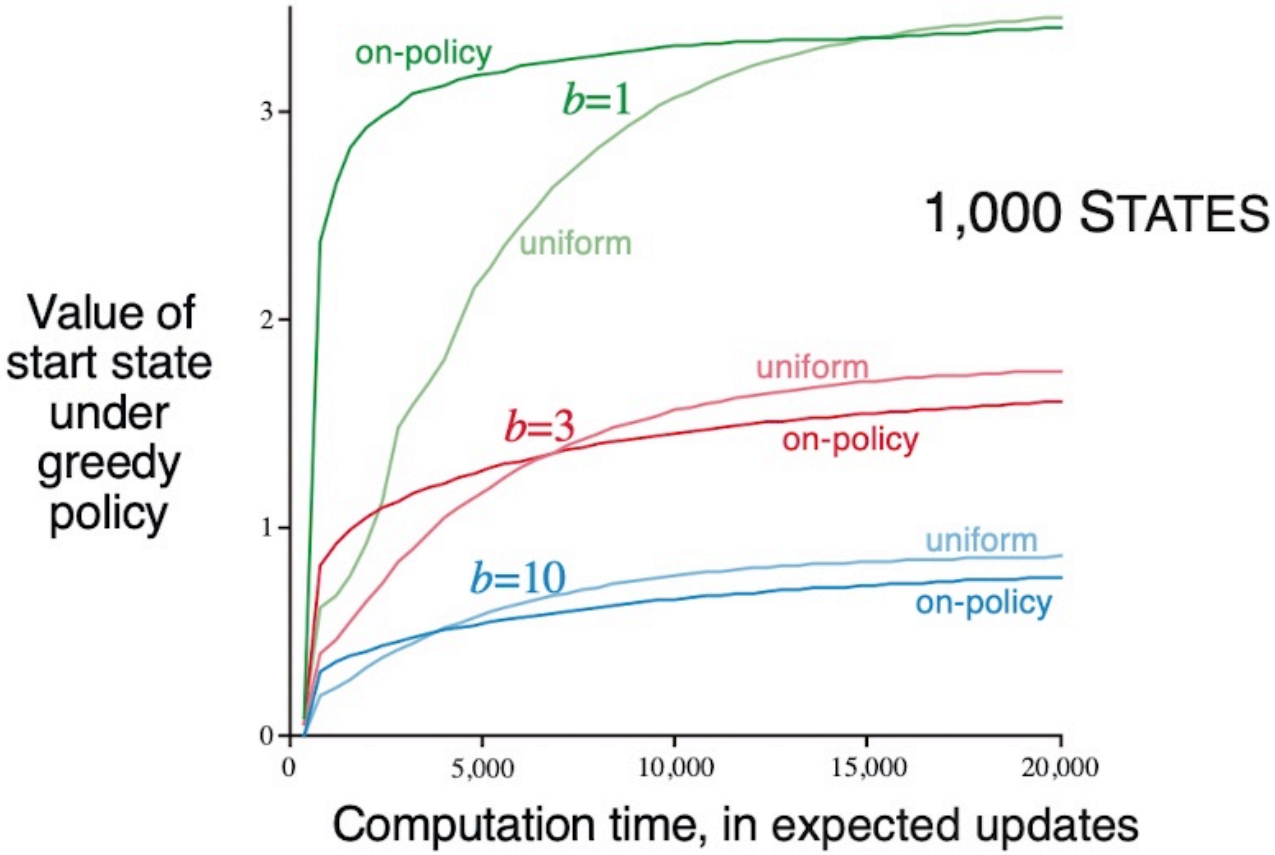
Here we compare 2 ways of distributing updates:

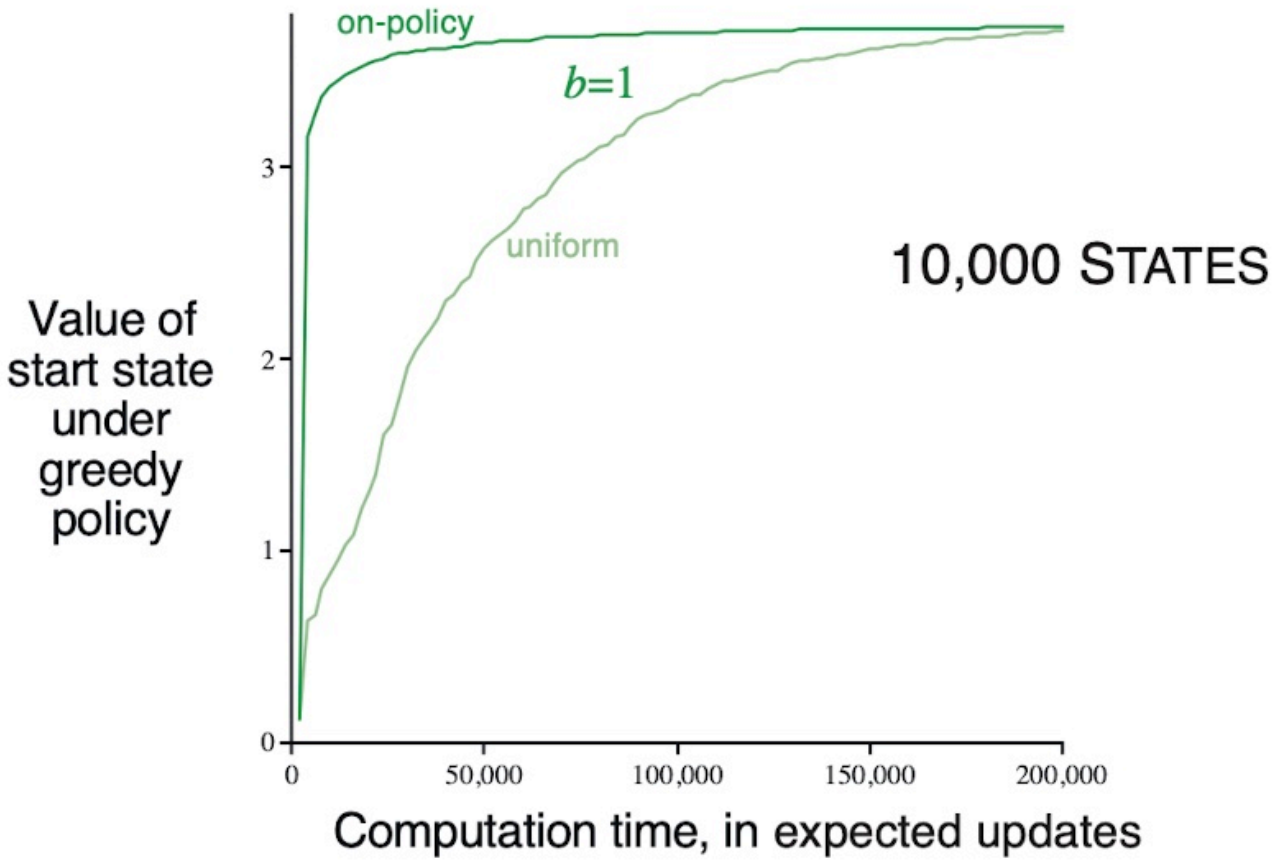
- Exhaustive sweep: classical approach from DP: perform sweeps thru the entire state-space, updating each state once per sweep.
- Sample from the state space according to some distribution.
  - uniform sample: Dyna-Q.
  - on-policy distribution (interact with the model following the current policy). Sample state transitions and rewards are given by the model, and sample actions are given by the current policy. We simulate explicit individual trajectories and perform update at the state encountered along the way
  → trajectory sampling.

Is the on-policy distribution a good one?

- It ignores vast, uninteresting parts of the space
- could be bad because we update the same parts over and over again.

Focusing on the on-policy distribution may hurt in the long run bc the commonly occuring states all already have their correct value. Sampling them is useless. This is why exhaustive sampling is better in the long run, at least for small problems. For large problems, the on-policy sampling is more efficient;


1,000 STATES
Value of start state under greedy policy
on-policy
b=1
uniform
uniform
b=3
on-policy
uniform
b=10
on-policy
0
1
2
3
0
5,000
10,000
15,000
20,000
Computation time, in expected updates


10,000 STATES
Value of start state under greedy policy
on-policy
b=1
uniform
0
1
2
3
0
50,000
100,000
150,000
200,000
Computation time, in expected updates

8.7 : Real-time Dynamic Programming (RTDP)

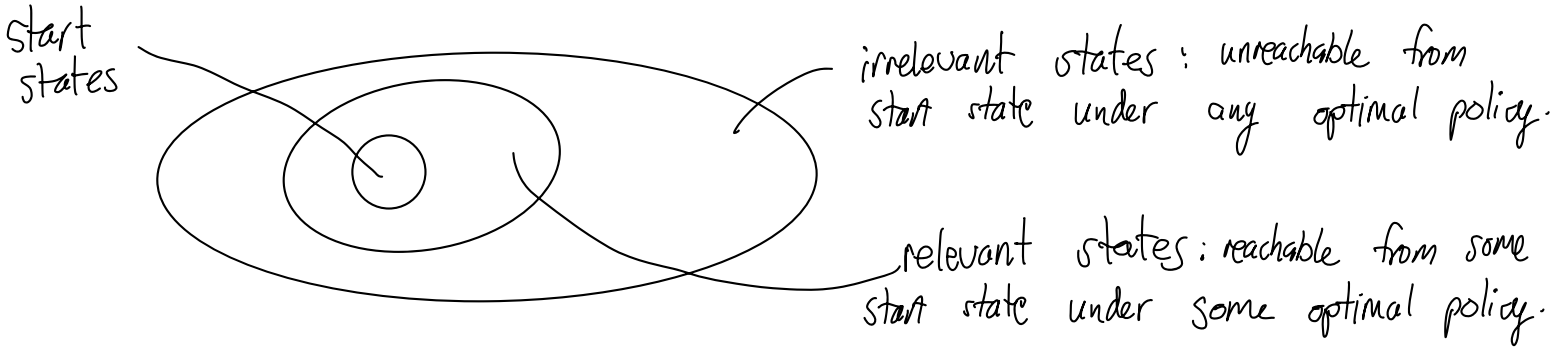
RTDP is an on-policy trajectory-sampling version of the value iteration algorithm of DP. It is closely related to full-sweep policy iteration so we illustrate some advantages that on-policy trajectory sampling can have.

RTDP updates the values of states visited in actual or simulated trajectories by means of expected tabular value-iteration updates.

RTDP is an example of an asynchronous DP algorithm. Asynchronous algorithms do not do full sweeps, they update state values in any order, 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.

Evaluation:

- trajectories start from a designated set of start states.
- we have a prediction problem for a given policy.
- on-policy trajectory sampling allows to focus on useful states.



Improvement:

For control, the goal is to find an optimal policy instead of evaluating a given policy, there might be states that cannot be reached by any of the optimal policies from any of the start states, so there is no need to specify optimal actions for those irrelevant states. we only need an optimal partial policy.

Finding an optimal partial policy can require visiting all state-action pairs, even those turning out to be irrelevant in the end. It is generally not possible to stop updating any state if convergence to an optimal policy is important.

For certain types of problems under certain 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.

- The task must be an undisclosed episodic task for MDP with absorbing goal state that generates 0 reward.

At every step of real or simulated trajectory, RTDP selects a greedy action and applies the expected value-iteration update operation to the current state. It can also update other states, like those visited in a limited-horizon look-ahead search from the current state.

More conditions for guarantee of optimal convergence:

- the initial value of each goal state is 0.
- there exists at least one policy that guarantees a goal state will be reached from any start state.
- all rewards from transitions from non-goal states are strictly negative.
- the initial values are $\geq$ to optimal values.

Tasks having these properties are examples of stochastic optimal path problems, which are usually stated in terms of cost minimization instead of reward maximization, but we can just maximize the negative returns.

As the value function approaches V^*, the policy used by the agent to generate trajectories approach an optimal policy be it is always greedy wrt the current value function. This is unlike traditional value iteration, where it terminates when the value function changes by a tiny amount in a sweep. It is possible that policies that are greedy wrt the latest value function were

optimal long before value iteration terminates! Checking the emergence of an optimal policy before value iteration converges is not a part of conventional DP and requires way more computation.

8.8: Planning at Decision Time

Planning can be used in 2 ways:

1) Background planning: so far (Dyna, DP), we use planning to gradually improve a policy or value function on the basis of simulated experience from a model. We then select actions by comparing the current state's action values. Well before an action is selected for any current state S_t, planning has played a role in improving the way to select the action for many states, including S_t. Planning here is not focused on the current state.

2) Decision-time planning: Begin and complete planning after encountering each state S_t, to produce a single action A_t.
Dummy example: planning when only state-values are available, so at each state we select an action based on the values of model-predicted next states for each action. We can also look much deeper than 1 step ahead. Here, planning focuses on a particular state.

Decision-time planning:

Even when planning is done at decision time, we can still view it as proceeding from simulated experience to updates and values, and ultimately to a policy. It's just that now the values and policy are specific to the current state. Those values and policies are typically discarded right after being used to select the current action.

In general, we may want to do a bit of both:

- focus planning on the current state.
- and store the results of planning so as to help when we return to the same state after.

Decision-time planning is best when fast responses are not required. If low latency action selection is the priority, we're better off doing planning in the background to compute a policy that can be rapidly used.

8.9: Heuristic Search

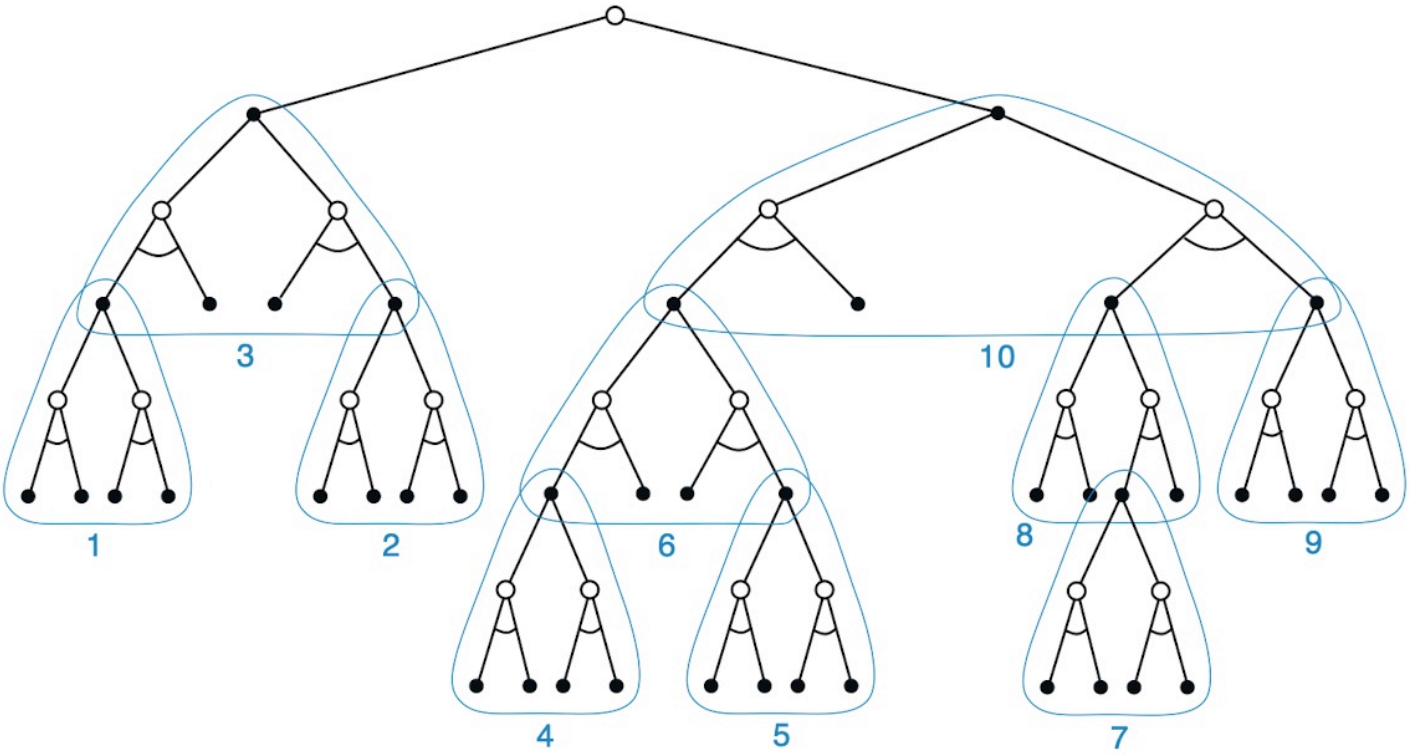
The classical state-space planning methods are decision-time planning methods collectively known as heuristic search. In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state (at the root), then the best is chosen as the current action, and all other backed-up values are discarded.

☆ Heuristic search can be seen as the extension of the idea of a greedy policy beyond a single step, to obtain better action selections. If the search is of sufficient depth k s.t. γ^k is very small, then the actions will be correspondingly near optimal. The trade off here is that deeper search leads to more computation.

Much of the effectiveness of heuristic search is due to its search tree being tightly focused on the states and actions that might immediately follows the current state. This great focusing of memory and computational resources on the current decision is presumably why heuristic search can be so effective.

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:



8.10: Rollout Algorithms

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environmental state. They estimate action-values for a given policy by averaging the returns of many simulated trajectories that start w/ each possible action and then follow the given policy. When the action-value estimates are accurate enough, the best one is executed and the process starts anew from the resulting next state.

Unlike MC algorithm, the goal here is not to estimate a complete optimal action-value function. Instead, we only care about the current state and one given policy, called the rollout policy. The aim of a rollout algorithm is to improve upon the rollout policy, not to find an optimal policy.

The computation time needed by a rollout algorithm depends on many things including the # of actions that have to be evaluated for each decision, and the # of time steps in the simulated trajectories. It can be demanding but:

- it is possible to run many trials in parallel on separate processes bc MC trials are independent of one another.

- we can truncate the trajectories short of complete episodes and correct the truncated returns by the means of a stored evaluation function.
- Monitor the MC simulations and prune away actions that are unlikely to be the best.

Rollout algorithms are not considered learning algorithms since they do not maintain long-term memory of values or policies, but they still use RL techniques (MC) and take advantage of the policy improvement property by acting greedily wrt the estimated action-values.

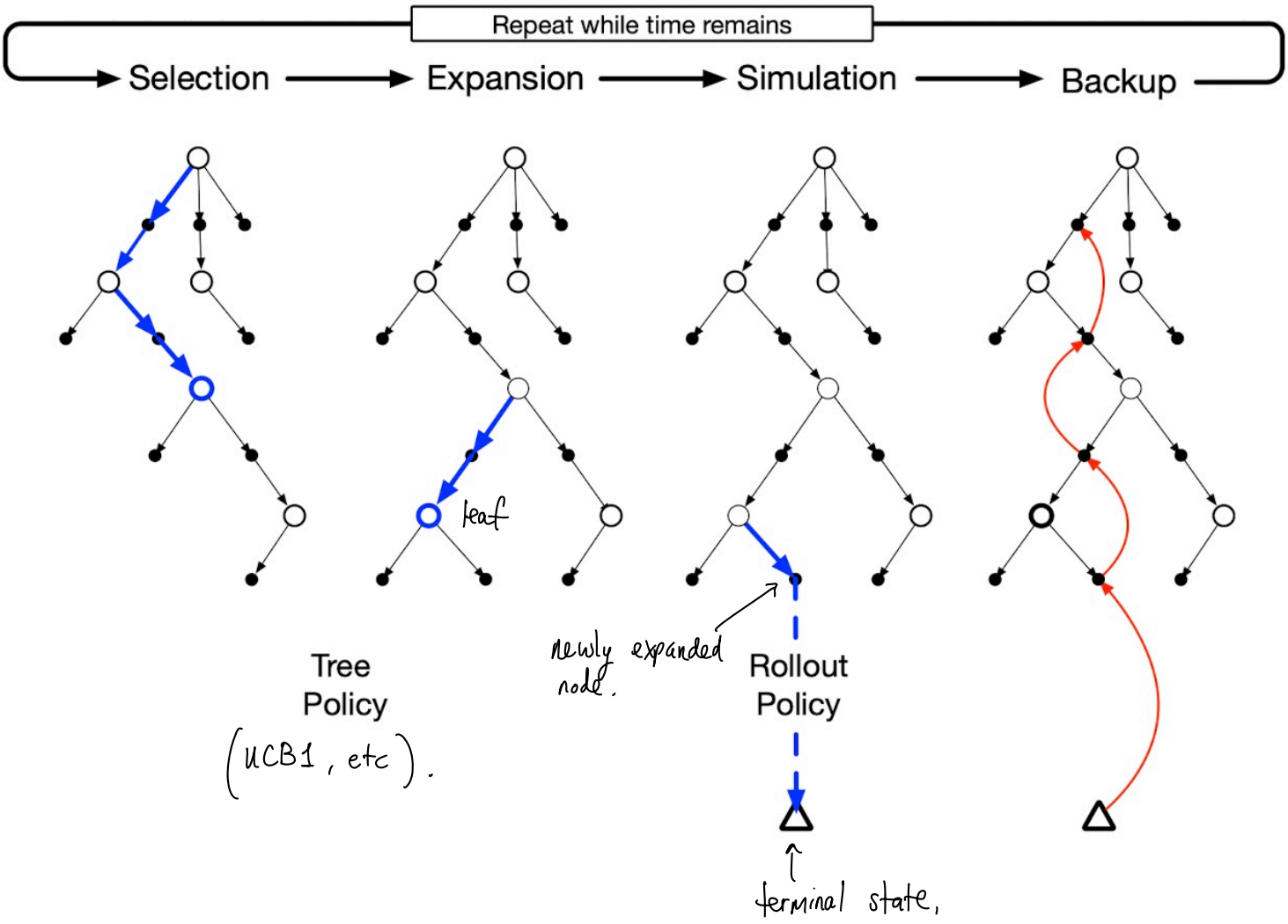
8.11 : Monte Carlo Tree Search (MCTS)

An example of decision-time planning. It is basically a rollout algorithm, but enhanced with a means of accumulating value estimates from the MC simulations, in order to successively direct simulations towards more highly-rewarding trajectories.

MCTS is executed after encountering each new state to select the action. 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). The core idea of MCTS is to successively focus multiple simulations starting at the current state by extending the initial portions of trajectories that have received high evaluations from earlier simulations.

The actions in the simulated trajectories are generated using a simple policy called a rollout policy.

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:



MCTS incrementally extends the tree by adding promising nodes based on the result of simulated trajectories. Any simulated trajectory will pass thru the tree and exit at some leaf node. Outside the tree and at leaf nodes the rollout policy is used to pick actions, but inside the tree we use the tree policy that balances exploration and exploitation (like an ε-greedy policy).

In more detail, each iteration of a basic MCTS consists of the following 4 steps:

1) Selection: starting at the root node, a tree policy based on the action value attached to the edges of the tree traverses the tree to select a leaf node.

2) Expansion: When you reach a leaf node, add one or more new child nodes representing unexplored actions from the state.

3) Simulation: From the selected node or its newly added children, run a complete simulation to the end of episode. Actions are selected first by the tree policy and beyond the tree by the rollout policy.

4) Backup: Take the result from the simulation and propagate it back up the tree, updating the value estimates for all the nodes that were visited during the selection phase.

Key Insight: 2 different policies:

- Tree policy: used inside the built tree, carefully balances exploration and exploitation.
- Rollout policy: used outside the tree, typically simple/random for speed.

MCTS continues executing these 4 steps, starting each time at the tree's root node, until no more time is left. Then, finally, an action from the root node (which still represents the current state of the environment) is selected (maybe the action having the largest action value, or highest visit count). After the action is selected, the environment transitions into new state and MCTS is run again, often starting w/ a tree that contains the relevant nodes from the tree constructed by the previous execution. All the remaining nodes are discarded along with their action values.

Summary:

- MCTS is a decision-time planning algorithm based on Monte Carlo control applied to simulations that start from the root state.
- It saves action-value estimates attached to the tree edges and updates them using sample updates (this focuses trajectories).
- MCTS expands its tree, so it effectively grows a look-up table to store a partial action-value function.

MCTS thus avoids the problem of globally approximating an action-value function while it retains the benefit of using past experience to guide exploration.

8.12 : Summary of the Chapter

Planning requires a model of the environment,

- A distribution model consists of the probabilities of next states and rewards for possible actions. DP requires a distribution model to be able to use expected updates, which involves computing an expectation over all possible states and rewards,
- A sample model is needed to simulate environmental interaction and use sample updates,

There is a close relationship btwn planning optimal behavior and learning optimal behavior:

- Both involve estimating the same value functions.
- Both naturally update the estimates incrementally, in a long series of backing-up operations.
- Any of the learning methods can be converted into planning methods simply by running them over simulated data.

Dimensions:

- Size of the update: the smaller the update, the more incremental the planning methods can be.
  small updates: one-step sample updates like Dyna.
- Distribution of updates (of the focus of the search):
  - prioritized sweeping focuses backward on the

predecessors of interesting states.

- on-policy trajectory sampling focuses on states that are likely.

Planning can also focus forward from pertinent states, and the most important form of this is when planning is done at decision time. Rollout algorithms and MCTS benefit from online, incremental, sample-based value estimation and policy improvement.

8.13 : Summary of Part I: Dimensions

Each idea presented in this part can be viewed as a dimension along which methods vary. The set of the dimension spans a large space of possible methods.

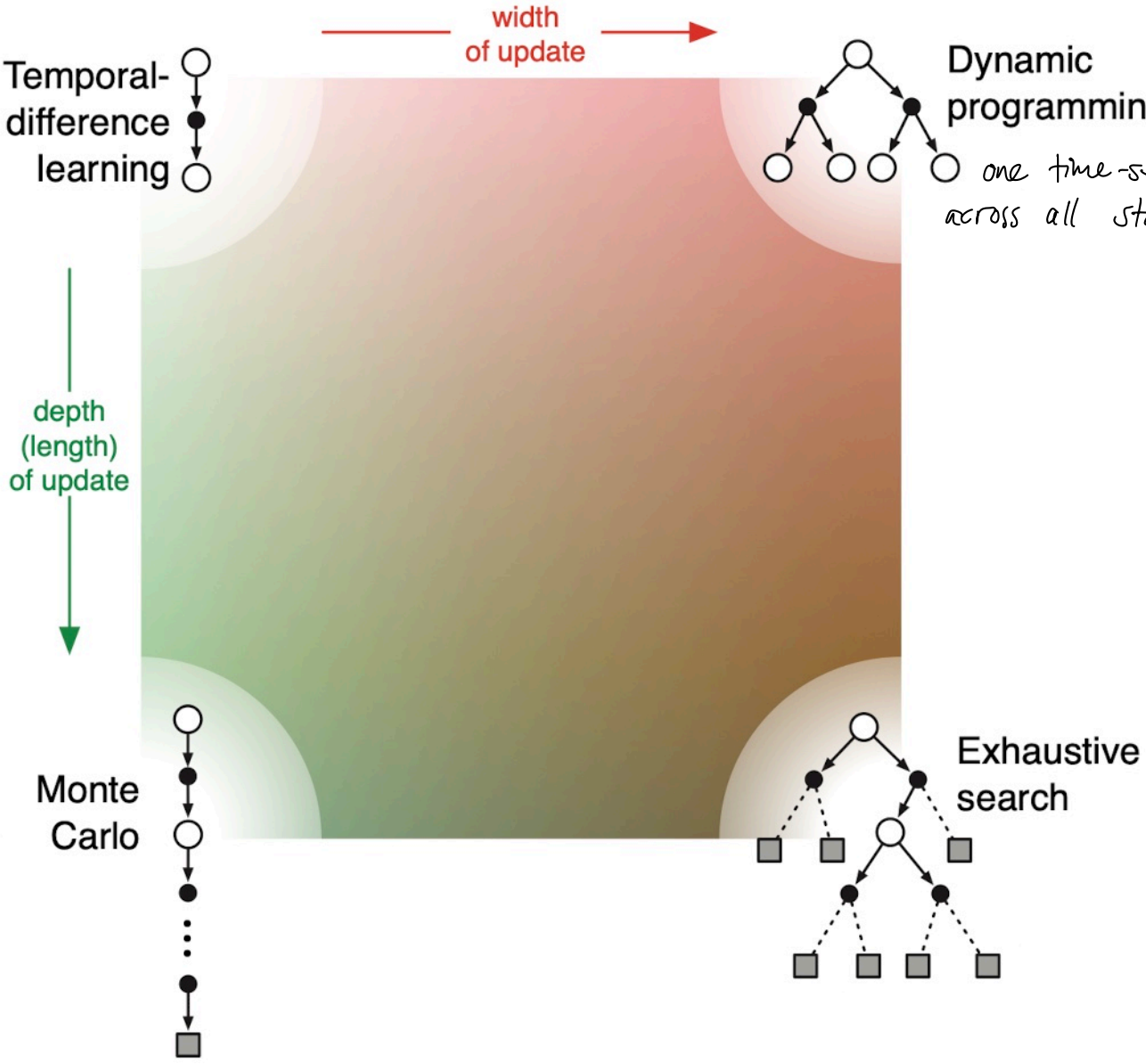
All the methods so far have 3 key ideas in common:

- they all seek to estimate value functions,
- they all operate by backing up values along actual or possible state trajectories
- they all follow GPI, meaning they maintain an approximate value function and an approximate policy, and they continually improve each based on the other.

Main dimensions:

Two of the main dimensions are shown (Fig 8.15),

- whether we use sample updates based on a sample trajectory or an expected update based on a distribution of possible trajectories.
- Depth of updates.


width
of update
Temporal-
difference
learning
Dynamic
programming
one time-step ahead,
across all states.
depth
(length)
of update
Monte
Carlo
Exhaustive
search

Other Dimensions:

- Binary distinction btwn off-policy and on-policy methods
- Definition of return: is the task episodic, discounted or undiscounted?
- Action-values vs state-values vs afterstate-values.
- Action selection / exploration: we have considered simple ways to do this: ε-greedy, optimistic init, softmax, and upper confidence bound (UCB).
- Synchronous vs asynchronous: are the updates for all states performed simultaneously or one-by-one in some order?
- Real vs simulated.
- Location of the updates: what states should be updated? Model free can only choose among encountered state but model-based methods can choose arbitrarily.
- Timing of updates: should updates be done as part of selecting actions or only afterwards?
- Memory of updates: how long should updated values be obtained?