RL Ch8 Planning and Learning with Tabular Methods

Le-Rong HSU

July 2023

1 Introduction

Why do we need a model? For example, we need a model when our "real experience", or training data, is not enough for satisfying prediction. We then use a model to simulate experience.

- model-based: DP, heuristic search
- model-free: MC, TD methods

Our goal in this chapter is to integrate the model-based methods with model-free methods. Let's go.

2 Models and Planning

A model of an environment is anything that an agent can use to predict how the environment will respond to its actions.

- distribution model: produces all probabilities of all possibilities
- sample model: produces just one one of the possibilities according to the probabilities

Models can be used to mimic or simulate experience, or produce *simulated experiences*. Given a starting state and policy/action,

- distribution model $\rightarrow$ generates all possible transitions weighted by their probabilities of occurring
- sample model $\rightarrow$ produces a possible transition

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

Planning refers to any process that takes a model as input and produces or improves a policy.

- state-space planning → search through the state space for an optimal policy or an optimal path to a goal

- plan-space planning → search through the space of plans.

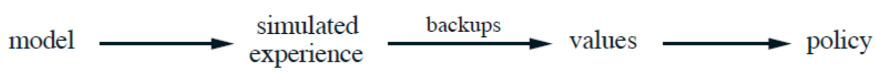
Plan-space methods are difficult to apply efficiently to the stochastic sequential decision problems that are the focus in reinforcement learning, and we do not consider them further.

All state-space planning methods share a common structure:

- compute value functions as a key intermediate step toward improving the policy

- compute value functions by updates (or backup operations) applied to simulated experience.

This common structure can be diagrammed as follows:



- planning methods → simulated experience generated by a model

- learning methods → use real experience generated by the environment.

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.

Algorithm 1 Random-sample one-step tabular Q-planning

---

while True **do**
  Select S, A randomly
  Send S, A to a sample model, and obtain sample next reward R and a sample new state S'
  Apply one-step tabular Q-learning to S, A, R, S'

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

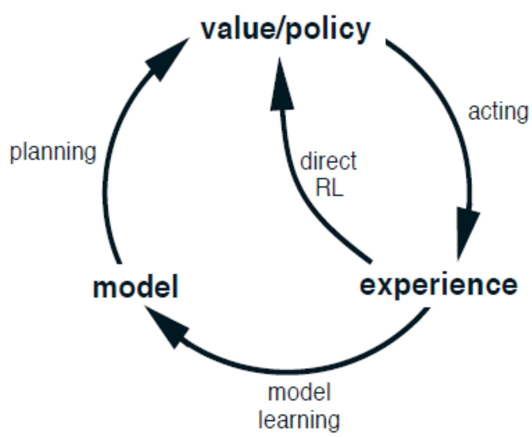
end while

Remark. To ensure the convergence of Q, each state-action pair must be selected an infinite number of times in Step 1, and α must decrease appropriately over time.

3 Dyna: Integrated Planning, Acting, and Learning

When we have a planning agent in our model, there are two roles for real experience:

- be used to improve the model → model learning
- be used to directly improve the value function and policy using RL methods → direct RL



Dyna-Q

Assume the environment is deterministic.

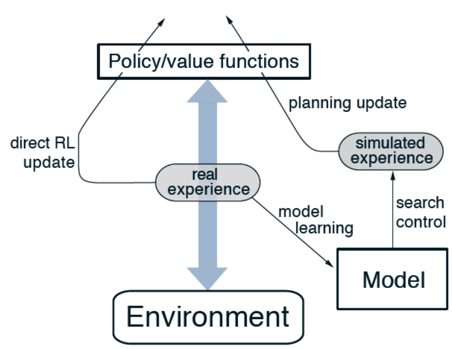


Figure 8.1

- direct RL update: improve the value function and the policy

- learning: learned from real experiences
- planning: achieved by applying reinforcement learning methods to the simulated experiences
- search control: selects the starting states and actions for the simulated experiences generated by the model

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

- If (e) and (f) are omitted, then the remaining algorithm is one-step Q-learning.
- Model(s,a) predicts the next state, reward

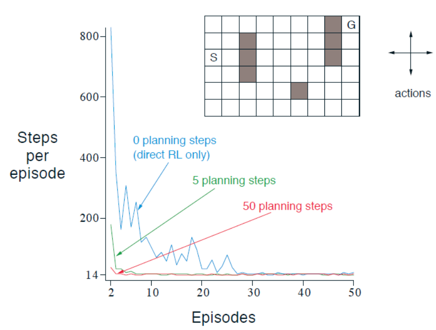


Figure 8.2 A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps (n) per real step. The task is to travel from **S** to **G** as quickly as possible.

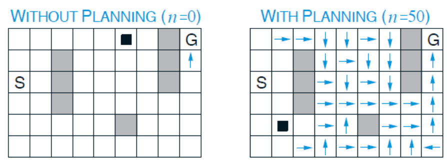


Figure 8.3 The arrows indicate the greedy action in each state; if no arrow is shown for a state, then all of its action values were equal. The black square indicates the location of the agent.

4 When the Model Is Wrong

In general, we cannot expect that the model is filled, or updated, with correct information. Models may be incorrect because

- the environment is stochastic and only a limited number of samples have been observed
- the environment has changed and its new behavior has not yet been observed
- the model was learned using function approximation that has generalized imperfectly.

In some cases, policies with exploration leads to the discovery and correction of the modeling error.

Bonus Reward

- Dyna-Q
- Dyna-Q+: Dyna-Q with an exploration bonus that encourages exploration

The reward mechanism: 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 $r + \kappa\sqrt{\tau}$, for some small κ.

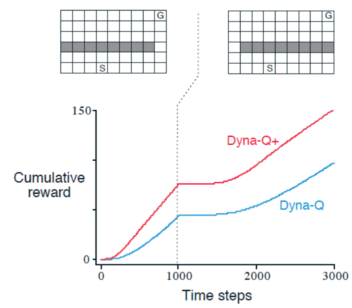


Figure 8.4 Average performance of Dyna agents on a blocking task. The left environment was used for the first 1000 steps, the right environment for the rest.

Greater difficulties arise when the environment changes to become *better* or *easier* 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.

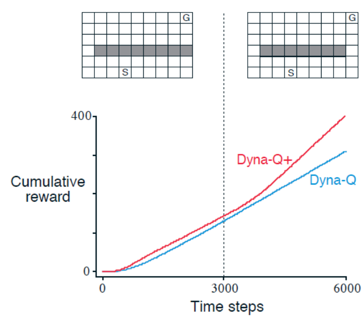


Figure 8.5 Average performance of Dyna agents on a blocking task. The left environment was used for the first 3000 steps, the right environment for the rest.

The graph shows that the regular Dyna-Q agent never switched to the shortcut. In fact, it never realized that it existed. Even with an ε-greedy policy, it is very unlikely that an agent will take so many exploratory actions as to discover the shortcut.

Remark. Exploration of Dyna-Q+ costs the leading gap in figure 8.5.

5 Prioritized Sweeping

Recall that in the step (f) of Dyna-Q, we (uniformly) randomly select a state and a random action from observed states and actions. However, a uniform selection is usually not the best; planning can be much more efficient if simulated transitions and updates are focused on particular state–action pairs.

For example, consider the maze game in figure 8.3. 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. This means that it is pointless to perform updates along almost all transitions, because they take the agent from one zero-valued state to another.

Backward Focusing

Suppose now that the agent discovers a change in the environment and changes its estimated value of one state, either up or down.

- the values of many other states may be changed
- useful updates: actions that lead directly into the one state whose value has been changed
- values of these actions changed $\rightarrow$ values of the predecessor states may change in turn

Prioritized Sweeping

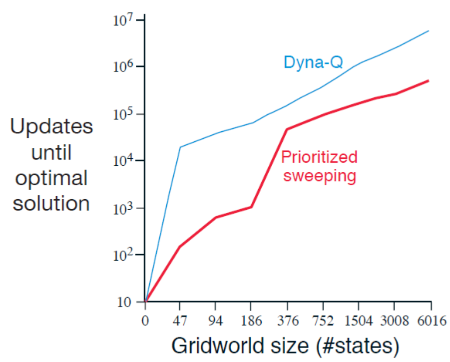
Not all updates are equally important $\rightarrow$ prioritize.

- initialize a queue to store the state-action pairs whose estimated value would change nontrivially if updated
- prioritize the pairs by the size of change
- 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

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

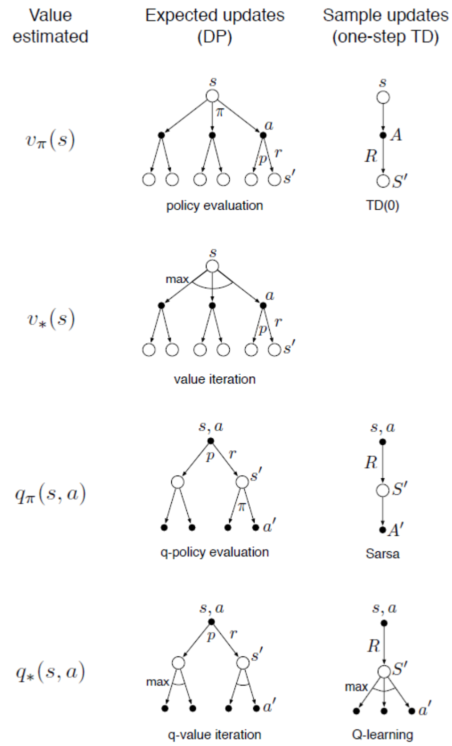
Example 8.4: Prioritized Sweeping on Mazes Prioritized sweeping has been found to dramatically increase the speed at which optimal solutions are found in maze tasks, often by a factor of $n = 5$ to $n = 10$.



■

6 Expected vs. Sample Updates

The difference between these expected and sample updates is significant to the extent that the environment is stochastic.



Notice that **expected updates are not available in the absence of a distribution model**, while sample updates can be done with sampling from the environment or a sample model.

Expected updates certainly yield a better estimate because they are uncorrupted by sampling error, but they also require more computation.

Consider the expected and sample updates for approximating q_*. The expected update for s, a is

$$Q(s,a) \leftarrow \sum_{s',r} \hat{p}(s',r|s,a)[r + \gamma \max_{a'} Q(s',a')]. \tag{8.1}$$

The corresponding sample update given the reward R and the next state S' is

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max_{a'} Q(S',a') - Q(s,a)]. \tag{8.2}$$

- sample update
  - cheaper computation
  - sample error
- expected update
  - exact computation, $Q(s,a)$'s correctness is limited only by the correctness of $Q(s',a')$
  - require a lot computation
- if enough time $\rightarrow$ expected update is better due to the absence of sampling error
- lack of time and computational resources $\rightarrow$ sample update

Let b be the branching factor, i.e., the number of possible next states such that $\hat{p}(s'|s,a) > 0$. Then an expected update of a pair requires roughly b times as much computation as a sample update.

Sample Updates are Better

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

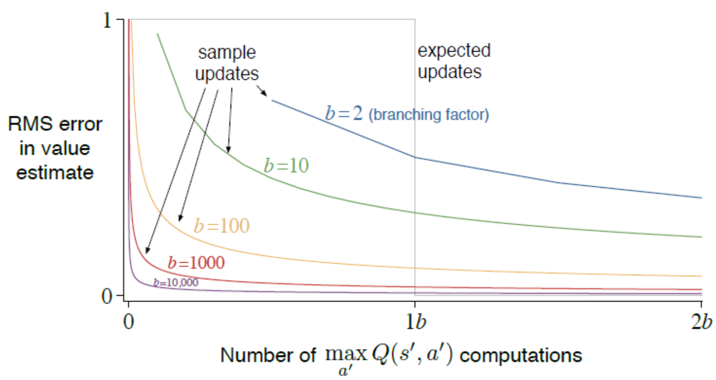


Figure 8.7: Comparison of efficiency of expected and sample updates.

Figure 8.7 Comparison of efficiency of expected and sample updates.

All b successor states are equally likely and in which the error in the initial estimate is 1. The values at the next states are assumed correct, so the expected update reduces the error to zero upon its completion. Sample updates reduce the error $\propto \sqrt{\frac{b-1}{bt}}$ where t is the number of sample updates having been performed (assuming sample average, i.e. $\alpha = \frac{1}{t}$).

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.

7 Trajectory Sampling

- DP: perform sweeps through the entire state (or state–action) space, updating each state (or state–action pair) once per sweep
  - not necessary, waste of computational resources

Trajectory Sampling

Sample trajectories according the distribution observed while following the policy.

- episodic task: one starts in a start state and simulates until the terminal state
- continuing task: one starts anywhere and just keeps simulating

We will also see in Part II that the on-policy distribution has significant advantages when function approximation is used. Whether or not function approximation is used, one might expect on-policy focusing to significantly improve the speed of planning.

Uniform vs. Simulated Trajectories (better)

Experiment:

- start in the same state, undiscounted
- uniform: cycled through all state–action pairs, updating each in place
- on-policy: updates using simulated episodes, updating each state–action pair that occurred under the current ε-greedy policy (ε=0.1)
- From each of $|\mathcal{S}|$ states, two actions are 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.
- $\mathbb{P}(s \to \text{terminal}) = 0.1$ for all s
- $b_s = b$ for all s
- expected reward $\sim \mathcal{N}(0, 1)$

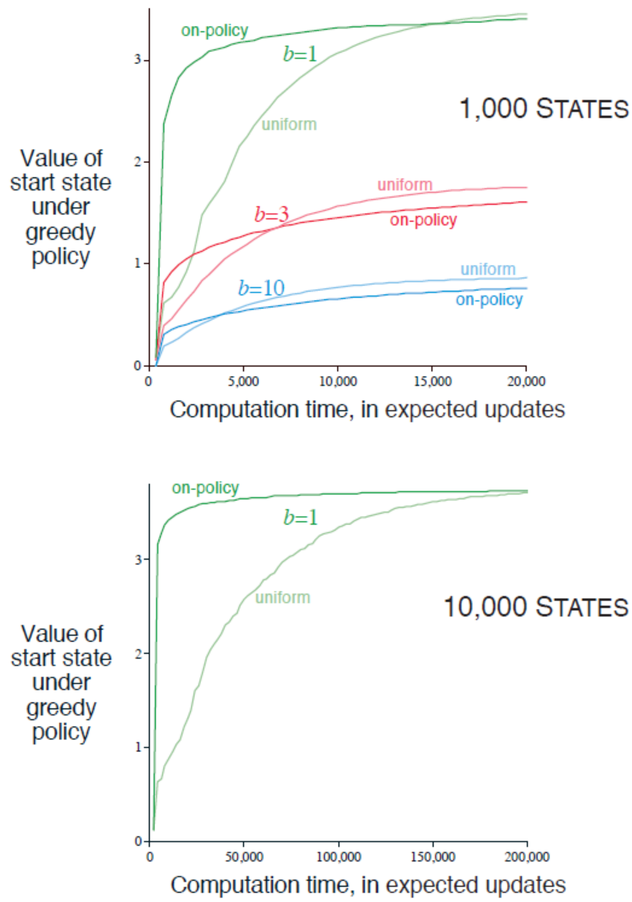


Figure 8.8 Results are for randomly generated tasks of two sizes and various branching factors b.

In all cases, sampling according to the on-policy distribution resulted in faster planning initially and retarded planning in the long run. In other experiments, we found that these effects also became stronger as the number of states increased.

The results are reasonable since:

- in the short term:
  - on-policy distribution $\rightarrow$ focusing on states that are near descendants of the start state
- in the long run:
  - Commonly occurring states all already have their correct values.

8 Real-time Dynamic Programming

Real-time dynamic programming, or RTDP, is an on-policy trajectory-sampling version of the value-iteration algorithm of dynamic programming (DP). RTDP updates the values of states visited in actual or simulated trajectories by means of expected tabular value-iteration updates as defined by (4.10).

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r | s, a)[r + \gamma v_k(s')] \end{aligned} \tag{4.10}$$

Some key points about RTDP:

- RTDP is an example of an asynchronous DP algorithm
- the update order is dictated by the order states are visited in real or simulated trajectories
- trajectories can start only from a designated set of start states
- prediction problem:
  - skip *irrelevant* states
- control problem:
  - find *optimal partial policy*: a policy that is optimal for the relevant states
  - on-policy trajectory-sampling control method but with **exploring starts**

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.

Suppose each episode starts in a state randomly chosen from the set of start states and ending at a goal state. RTDP converges with probability one to a policy that is optimal for all the relevant states provided:

- the initial value of every goal state is zero
- there exists at least one policy that guarantees that a goal state will be reached with probability one from any start state
- all rewards for transitions from non-goal states are strictly negative
- 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 zero)

The result is proved in Learning to act using real-time dynamic programming by Barto, Bradtke, and Singh (1995).

Remark. The tasks for which this result holds are undiscounted episodic tasks for MDPs with absorbing goal states that generate zero rewards.

Remark. Tasks having these properties are examples of stochastic optimal path problems, which are usually stated in terms of cost minimization instead of as reward maximization as we do here.

Example 8.6: RTDP on the Racetrack

Figure 5.5: A couple of right turns for the racetrack task.

Figure 5.5 Results are for randomly generated tasks of two sizes and various branching factors b.

Game setting:

- start state: zero-speed state
- velocity: no limit
- thus, state set is infinite
- reward $= -1$ for each step until the car cross the finish line
- hit boundary $\rightarrow$ moved back to a random position on the starting line and the episode continues
- the set of states that can be reached from the set of start states via any policy is finite
- the set of policy can be viewed as the state set of this problem
- DP: accuracy $< 10^{-4}$
- RTDP: cross the finish line over 20 episodes

	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 $\leq$ 100 times	—	98.45
% of states updated $\leq$ 10 times	—	80.51
% of states updated 0 times	—	3.18

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 in an average run.

Another advantage of RTDP is that 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.

Conclusion:

- DP: update all states, need massive computation
- RTDP: focus on subsets of the states that were relevant to the problem's objective. Because the convergence theorem for RTDP applies to the simulations, we know that RTDP eventually would have focused only on relevant states.
- RTDP achieved nearly optimal control with about 50% of the computation required by value iteration

9 Planning at Decision Time

Background Planning

- improve a policy or value function on the basis of simulated experience obtained from a model
- compare the current state's action values obtained from a table
- planning is not focused on the current state

Decision-time Planning

Say we are at state S_t.

- begin and complete the planning before S_{t+1}, i.e., we plan every time we visit a state
- output an action A_t

Some key points:

- Decision-time planning is most useful in applications in which fast responses are not required. e.g. chess game
- Otherwise, apply background planning to a policy that can then be rapidly applied to each newly encountered state.

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 environment state.

averages the returns of many simulated trajectories	$\rightarrow$	update action values
	$\rightarrow$	if some action-value estimate is accurate enough
	$\rightarrow$	the action with highest action value will be selected

A *rollout policy* produce Monte Carlo estimates of action values only for each current state and for a given policy. **It does not aim to estimate a complete optimal action-value function q_* or complete action-value function q_π. It utilizes current action values to simulate complete episodes** and choose an action by the above procedure. The **policy improvement theorem** ensures better policy after each update.

11 Monte Carlo Tree Search

Core idea: simulate the game

Monte Carlo Tree Search (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 single-agent sequential decision problems if there is an environment model simple enough for fast multistep simulation. For example, **Alpha Go**.

MCTS consists of 3 steps: select, rollout/expand, backup (or backpropagate). We use a node to refer to a state.

1. **Select**: Keep selecting nodes until the leaf node is reached. If there's no leaf nodes then select the root node. The policy of selection can be for example, ε-epsilon or UCB.
2. **Rollout/Expand**: If the node hasn't been chosen (updated), we simulate a complete episode by a rollout policy. If the node has been updated, we add child nodes the this leaf node and then select one of them (again, ε-epsilon or UCB).
3. **Backup**: Update the current leaf node and propagate to its parent nodes.

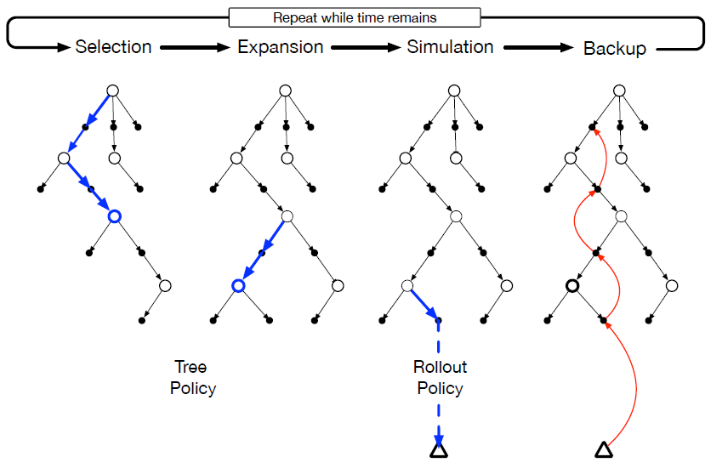


Figure 8.10 Selection, **Expansion** (though possibly skipped on some iterations), **Simulation**, and **Backup**

Conclusion:

- decision-time algorithm
- online, incremental, sample-based value estimation
- policy improvement
- focusing the Monte Carlo trials on trajectories whose initial segments are common to high-return trajectories previously simulated