COURSE NOTES*

Planning and Learning

Richard S. Sutton and Andrew G. Barto

In this chapter we develop a unified view of 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 Monte Carlo and temporal-difference methods. We think of the former as *planning* methods and of the latter as *learning* methods. Although there are real differences between these two kinds of methods, there are also great similarities. In particular, 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 to update an approximate value function. Earlier in this book we presented Monte Carlo and temporal-difference methods as distinct alternatives, then showed how they can be seamlessly integrated by using eligibility traces such as in TD(λ). Our goal in this chapter is a similar integration of planning and learning methods. Having established these as distinct in earlier chapters, we now explore the extent to which they can be intermixed.

1 Models and Planning

By a *model* of the environment we mean 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. If the model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities; these we call *distribution models*. Other models produce just one of the possibilities, sampled according to the probabilities; these we call *sample models*. For example, consider modeling the sum of a dozen dice. A distribution model would produce all possible sums and their

*These course notes are chapters from a textbook, *Reinforcement Learning: An Introduction*, by Richard S. Sutton and Andrew G. Barto, to be published by MIT Press in January, 1998.

probabilities of occurring, whereas a sample model would produce an individual sum drawn according to this probability distribution. The kind of model assumed in dynamic programming—estimates of the state transition probabilities and expected rewards, $P^a_{ss'}$ and $R^a_{ss'}$—is a distribution model. The kind of model used in the blackjack example in Chapter 5 is a sample model. Distribution models are stronger than sample models in that they can always be used to produce samples. However, in surprisingly many applications it is much easier to obtain sample models than distribution models.

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*.

The word *planning* is used in several different ways in different fields. We use the term to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment:

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

Within artificial intelligence, there are two distinct approaches to planning according to our definition. In *state-space planning*, which includes the approach we take in this book, planning is viewed primarily as a search through the state space for an optimal policy or path to a goal. Actions cause transitions from state to state, and value functions are computed over states. In what we call *plan-space planning*, planning is instead viewed as a search through the space of plans. Operators transform one plan into another, and value functions, if any, are defined over the space of plans. Plan-space planning includes evolutionary methods and *partial-order planning*, a popular kind of planning in artificial intelligence in which the ordering of steps is not completely determined at all stages of planning. Plan-space methods are difficult to apply efficiently to the stochastic optimal control problems that are the focus in reinforcement learning, and we do not consider them further (but see Section 11.6 for one application of reinforcement learning within plan-space planning).

The unified view we present in this chapter is that all state-space planning methods share a common structure, a structure that is also present in the learning methods presented in this book. It takes the rest of the chapter to develop this view, but there are two basic ideas: (1) all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy, and (2) they compute their value functions by backup operations applied to simulated experience. This common structure can be diagrammed as

Do forever:

1. Select a state, $s \in \mathcal{S}$, and an action, $a \in \mathcal{A}(s)$, at random
2. Send s, a to a sample model, and obtain
   a sample next state, s', and a sample next reward, r
3. Apply one-step tabular Q-learning to s, a, s', r:
   $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

Figure 1: Random-sample one-step tabular Q-planning

follows:

Dynamic programming methods clearly fit this structure: they 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 and update the state's estimated value. In this chapter we argue that various other state-space planning methods also fit this structure, with individual methods differing only in the kinds of backups they do, the order in which they do them, and in how long the backed-up information is retained.

Viewing planning methods in this way emphasizes their relationship to the learning methods that we have described in this book. The heart of both learning and planning methods is the estimation of value functions by backup operations. The difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment. Of course this difference leads to a number of other differences, for example, in how performance is assessed and in how flexibly experience can be generated. But the common structure means that many ideas and algorithms can be transferred between planning and learning. In particular, in many cases a learning algorithm can be substituted for the key backup 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. Figure 1 shows a simple example of a planning method based on one-step tabular Q-learning and on random samples from a sample model. This method, which we call *random-sample one-step tabular Q-planning*, 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).

In addition to the unified view of planning and learning methods, a second theme in this chapter is the benefits of planning in small, incremental steps. This

enables planning to be interrupted or redirected at any time with little wasted computation, which appears to be a key requirement for efficiently intermixing planning with acting and with learning of the model. More surprisingly, later in this chapter we present evidence that 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.

2 Integrating Planning, Acting, and Learning

When planning is done on-line, 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 in some way to the states or decisions currently under consideration, or expected in the near future. If decision-making and model-learning are both computation-intensive processes, then the available computational resources may need to be divided between them. To begin exploring these issues, in this section we present Dyna-Q, a simple architecture integrating the major functions needed in an on-line planning agent. Each function appears in Dyna-Q in a simple, almost trivial, form. In subsequent sections we elaborate some of the alternate ways of achieving each function and the trade-offs between them. For now, we seek merely to illustrate the ideas and stimulate your intuition.

Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (to make it more accurately match the real environment) and it can be used to directly improve the value function and policy using the kinds of reinforcement learning methods we have discussed in previous chapters. The former we call *model-learning*, and the latter we call *direct reinforcement learning* (direct RL). The possible relationships between experience, model, values, and policy are summarized in Figure 2. Each arrow shows a relationship of influence and presumed improvement. Note how experience can improve value and policy functions either directly or indirectly via the model. It is the latter, which is sometimes called *indirect reinforcement learning*, that is involved in planning.

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. On the other hand, direct methods are 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. Related debates in psychology and AI concern the relative importance of cognition as opposed to trial-and-error learning, and of deliberative planning as opposed to reactive decision-making. Our view is that

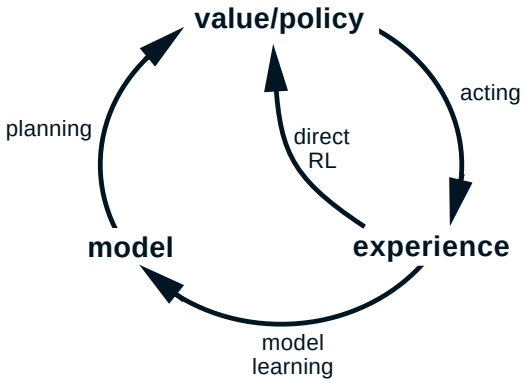

Figure 2: Relationships among learning, planning, and acting.

the contrast between the alternatives in all these debates has been exaggerated, that more insight can be gained by recognizing the similarities between these two sides than by opposing them. For example, in this book we have emphasized the deep similarities between dynamic programming and temporal-difference methods, even though one was designed for planning and the other for modelfree learning.

Dyna-Q includes all of the processes shown in Figure 2—planning, acting, model-learning, and direct RL—all occurring continually. The planning method is the random-sample one-step tabular Q-planning method given in Figure 1. The direct RL method is one-step tabular Q-learning. The model -learning method is also table-based and assumes the world is deterministic. After each transition $s_t, a_t \rightsquigarrow s_{t+1}, r_{t+1}$, the model records in its table entry for s_t, a_t the prediction that s_{t+1}, r_{t+1} will 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 (in Step 1), so the model is never queried with a pair about which it has no information.

The overall architecture of Dyna agents, of which the Dyna-Q algorithm is one example, is shown in Figure 3. The central column represents the basic interaction between agent and environment, giving rise to a trajectory of real experience. The arrow on the left of the figure represents direct reinforcement learning operating on real experience to improve the value function and the policy. On the right are model-based processes. The model is learned from real experience and gives rise to simulated experience. We use the term *search control* to refer to the process that selects the starting states and actions for the simulated experiences generated by the model. Finally, planning is achieved by applying reinforcement learning methods to the simulated experiences just as if they had really happened. Typically, as in Dyna-Q, the same reinforcement learning method is used both for

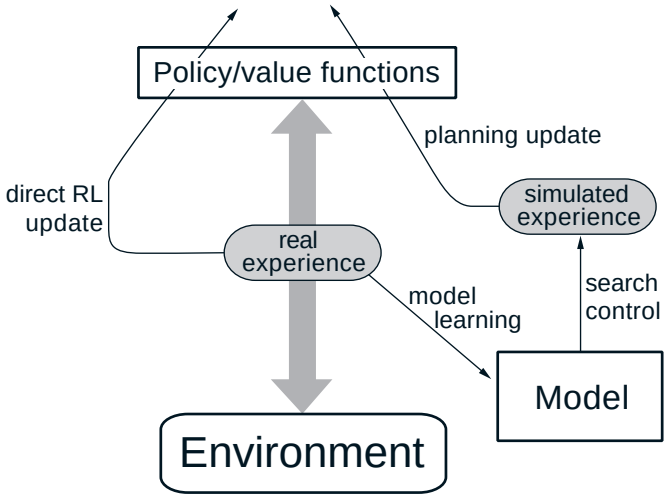

Figure 3: The general Dyna Architecture

learning from real experience and for planning from simulated experience. The reinforcement learning method is thus the "final common path" for both learning and planning. Learning and planning are deeply integrated in the sense that they share almost all the same machinery, differing only in the source of their experience.

Conceptually, planning, acting, model-learning, and direct RL occur simultaneously and in parallel in Dyna agents. For concreteness and implementation on a serial computer, however, we fully specify the order in which they occur within a time step. In Dyna-Q, the acting, model-learning, and direct RL processes require little computation, and we assume they consume just a fraction of the time. The remaining time in each step can be devoted to the planning process, which is inherently computation-intensive. Let us assume that there is time in each step, after acting, model-learning, and direct RL, to complete N iterations (Steps 1–3) of the Q-planning algorithm. Figure 4 shows the complete algorithm for Dyna-Q.

Example 1: Dyna Maze Consider the simple maze shown inset in Figure 5. In each of the 47 states 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. This is a discounted, episodic task with $\gamma = 0.95$.

The main part of Figure 5 shows average learning curves from an experiment in which Dyna-Q agents were applied to the maze task. The initial action values

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $s \leftarrow$ current (nonterminal) state
(b) $a \leftarrow \varepsilon$-greedy(s,Q)
(c) Execute action a; observe resultant state, s', and reward, r
(d) $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
(e) $Model(s,a) \leftarrow s', r$ (assuming deterministic environment)
(f) Repeat N times:
$s \leftarrow$ random previously observed state
$a \leftarrow$ random action previously taken in s
$s', r \leftarrow Model(s,a)$
$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

Figure 4: Dyna-Q Algorithm. $Model(s,a)$ denotes the contents of the model (predicted next state and reward) for state–action pair s,a. Direct reinforcement learning, model-learning, and planning are implemented by steps (d), (e), and (f), respectively. If (e) and (f) were omitted, the remaining algorithm would be one-step tabular Q-learning.

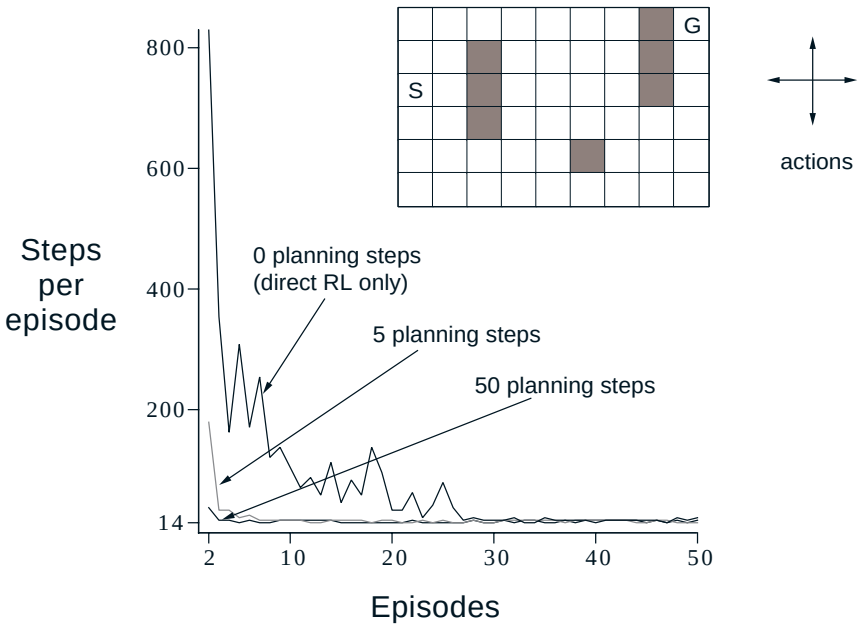

Figure 5: A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps per real step. The task is to travel from S to S as quickly as possible.

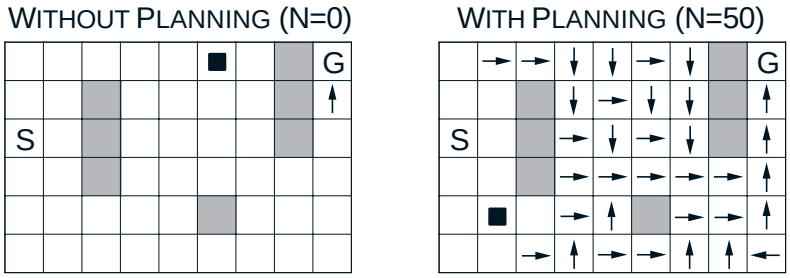

Figure 6: Policies found by planning and nonplanning Dyna-Q agents halfway through the second episode. The arrows indicate the greedy action in each state; no arrow is shown for a state if all of its action values are equal. The black square indicates the location of the agent.

were zero, the step-size parameter was $\alpha = 0.1$, and the exploration parameter was $\varepsilon = 0.1$. When selecting greedily among actions, ties were broken randomly. The agents varied in the number of planning steps, N, they performed per real step. For each N, the curves show the number of steps taken by the agent in each episode, averaged over 30 repetitions of the experiment. In each repetition, the initial seed for the random number generator was held constant across algorithms. Because of this, the first episode was exactly the same (about 1700 steps) for all values of N, and its data are not shown in the figure. 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, utilizing only direct reinforcement learning (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.

Figure 6 shows why the planning agents found the solution so much faster than the nonplanning agent. Shown are the policies found by the $N = 0$ and $N = 50$ agents halfway through the second episode. 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, again only one step is learned during the first episode, but here during the second episode an extensive policy has been developed that by the episode's end will reach almost back to the start state. This policy is built by the planning process while the agent is still wandering near the start state. By the end of the third episode a complete optimal policy will have been found and perfect performance attained. •

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. Because planning proceeds incrementally, it is trivial to intermix plan-

ning and acting. 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.

Exercise 1 The nonplanning method looks particularly poor in Figure 6 because it is a one-step method; a method using eligibility traces would do better. Do you think an eligibility trace method could do as well as the Dyna method? Explain why or why not.

3 When the Model Is Wrong

In the maze example presented in the previous section, the changes in the model were relatively modest. The model started out empty, and was then filled only with exactly correct information. In general, we cannot expect to be so fortunate. Models may be incorrect because the environment is stochastic and only a limited number of samples have been observed, because the model was learned using function approximation that has generalized imperfectly, or simply because the environment has changed and its new behavior has not yet been observed. When the model is incorrect, the planning process will compute a suboptimal policy.

In some cases, the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error. This tends to happen when the model is optimistic in the sense of predicting greater reward or better state transitions than are actually possible. The planned policy attempts to exploit these opportunities and in doing so discovers that they do not exist.

Example 2: Blocking Maze A maze example illustrating this relatively minor kind of modeling error and recovery from it is shown in Figure 7. Initially, there is a short path from start to goal, to the right of the barrier, as shown in the upper left of the figure. After 1000 time steps, the short path is "blocked," and a longer path is opened up along the left-hand side of the barrier, as shown in upper right of the figure. The graph shows average cumulative reward for Dyna-Q and two other Dyna agents. The first part of the graph shows that all three 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 were able to find 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.

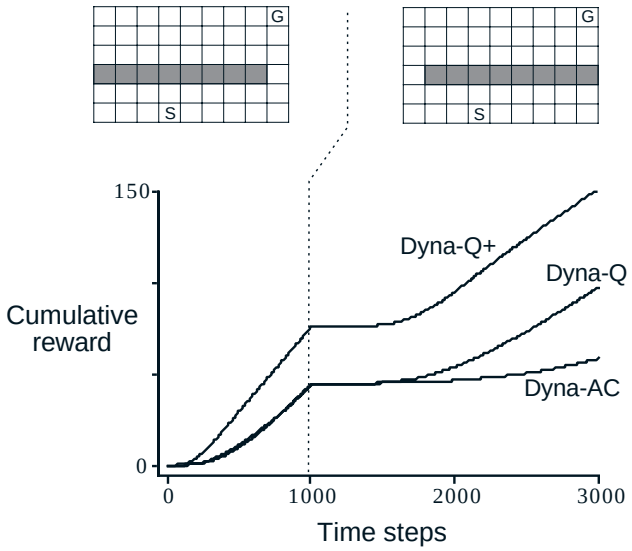

Figure 7: Average performance of Dyna agents on a blocking task. The left environment was used for the first 1000 steps, the right environment for the rest. Dyna-Q+ is Dyna-Q with an exploration bonus that encourages exploration. Dyna-AC is a Dyna agent that uses an actor–critic learning method instead of Q-learning.

In these cases the modeling error may not be detected for a long time, if ever, as we see in the next example.

Example 3: Shortcut Maze The problem caused by this kind of environmental change is illustrated by the maze example shown in Figure 8. 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). The graph shows that two of the three Dyna agents never switched to the shortcut. In fact, they never realized that it existed. Their models said that there was no shortcut, so the more they planned, the less likely they were 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. •

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. As in the earlier exploration/exploitation conflict, there probably is no solution that is both perfect and practical, but simple heuristics are often effective.

The Dyna-Q+ agent that did solve the shortcut maze uses one such heuristic.

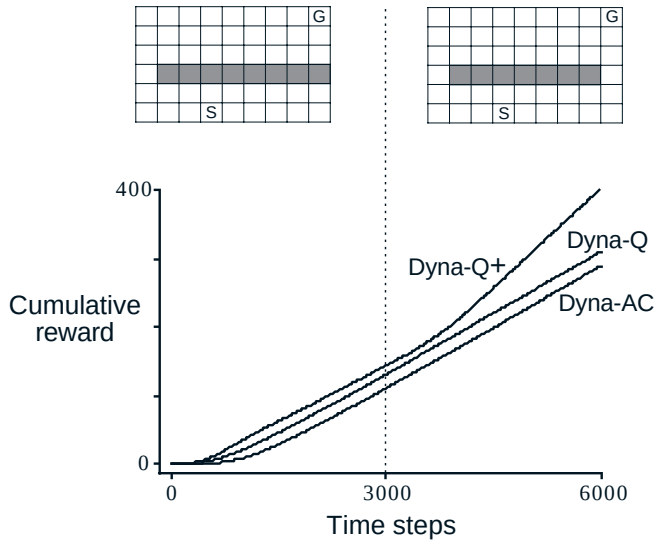

Figure 8: Average performance of Dyna agents on a shortcut task. The left environment was used for the first 3000 steps, the right environment for the rest.

This 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 n time steps, then planning backups are done as if that transition produced a reward of $r + \kappa\sqrt{n}$, for some small κ. This encourages the agent to keep testing all accessible state transitions and even to plan long sequences of actions in order to carry out such tests. Of course all this testing has its cost, but in many cases, as in the shortcut maze, this kind of computational curiosity is well worth the extra exploration.

Exercise 2 Why did the Dyna agent with exploration bonus, Dyna-Q+, perform better in the first phase as well as in the second phase of the blocking and shortcut experiments?

Exercise 3 Careful inspection of Figure 8 reveals that the difference between Dyna-Q+ and Dyna-Q narrowed slightly over the first part of the experiment. What is the reason for this?

Exercise 4 (programming) The exploration bonus described above actually changes the estimated values of states and actions. Is this necessary? Suppose the bonus $\kappa\sqrt{n}$ was used not in backups, but solely in action selection. That is, suppose the action selected was always that for which $Q(s,a) + \kappa\sqrt{n_{sa}}$ was maximal.

Carry out a gridworld experiment that tests and illustrates the strengths and weaknesses of this alternate approach.

4 Prioritized Sweeping

In the Dyna agents presented in the preceding sections, simulated transitions are started in state–action pairs selected uniformly at random from all previously experienced pairs. But a uniform selection is usually not the best; planning can be much more efficient if simulated transitions and backups are focused on particular state–action pairs. For example, consider what happens during the second episode of the first maze task (Figure 6). 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 back up along almost all transitions, because they take the agent from one zero-valued state to another, and thus the backups would have no effect. Only a backup along a transition into the state just prior to the goal, or from it into the goal, will change any values. If simulated transitions are generated uniformly, then many wasteful backups will be made before stumbling onto one of the two useful ones. As planning progresses, the region of useful backups 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.

This example suggests that 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. Goal states are just a special case, convenient for stimulating intuition. In general, we want to work back not just from goal states but from any state whose value has changed. Assume that the values are initially correct given the model, as they were in the maze example prior to discovering the goal. Suppose now that the agent discovers a change in the environment and changes its estimated value of one state. Typically, this will imply that the values of many other states should also be changed, but the only useful one-step backups are those of actions that lead directly into the one state whose value has already 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 backed up, 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 backups or terminating the propagation.

As the frontier of useful backups propagates backward, it often grows rapidly, producing many state–action pairs that could usefully be backed up. But not all of these will be equally useful. The values of some states may have changed a

Initialize $Q(s,a)$, $Model(s,a)$, for all s, a, and $PQueue$ to empty
Do forever:
- (a) $s \leftarrow$ current (nonterminal) state
- (b) $a \leftarrow policy(s,Q)$
- (c) Execute action a; observe resultant state, s', and reward, r
- (d) $Model(s,a) \leftarrow s', r$
- (e) $p \leftarrow |r + \gamma \max_{a'} Q(s',a') - Q(s,a)|$.
- (f) if $p > \theta$, then insert s, a into $PQueue$ with priority p
- (g) Repeat N times, while $PQueue$ is not empty:
  - $s, a \leftarrow first(PQueue)$
  - $s', r \leftarrow Model(s,a)$
  - $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  - Repeat, for all $\bar{s}, \bar{a}$ predicted to lead to s:
    - $\bar{r} \leftarrow$ predicted reward
    - $p \leftarrow |\bar{r} + \gamma \max_a Q(s,a) - Q(\bar{s},\bar{a})|$.
    - if $p > \theta$ then insert $\bar{s}, \bar{a}$ into $PQueue$ with priority p

Figure 9: The prioritized sweeping algorithm for a deterministic environment.

lot, whereas others 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 backed up. It is natural to prioritize the backups according to a measure of their urgency, and perform them in order of priority. This is the idea behind *prioritized sweeping*. A queue is maintained of every state–action pair whose estimated value would change nontrivially if backed up, prioritized by the size of the change. When the top pair in the queue is backed up, 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 (if there is a previous entry of the pair in the queue, then insertion results in only the higher priority entry's remaining in the queue). In this way the effects of changes are efficiently propagated backward until quiescence. The full algorithm for the case of deterministic environments is given in Figure 9.

Example 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 5 to 10. A typical example is shown in Figure 10. These data are for a sequence of maze tasks of exactly the same structure as the one shown in Figure 5, except that they vary in the grid resolution. Prioritized sweeping maintained a decisive advantage over unprioritized Dyna-Q. Both systems made at most $N = 5$ backups per environmental interaction. •

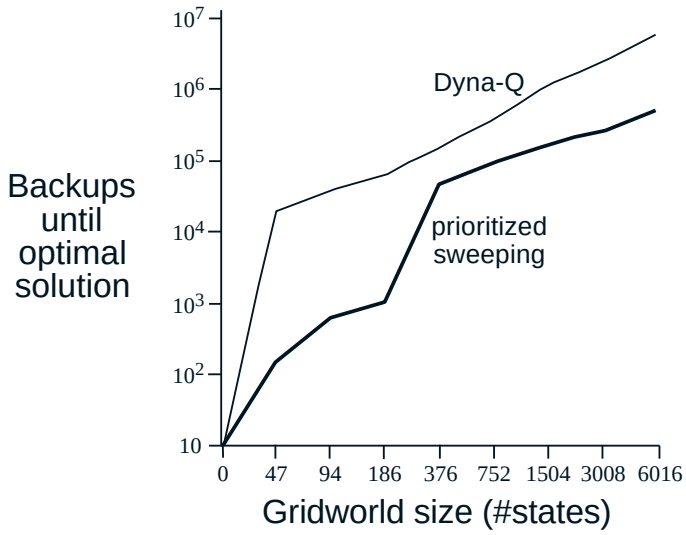

Figure 10: Prioritized sweeping significantly shortens learning time on the Dyna maze task for a wide range of grid resolutions. Reprinted from Peng and Williams (1993).

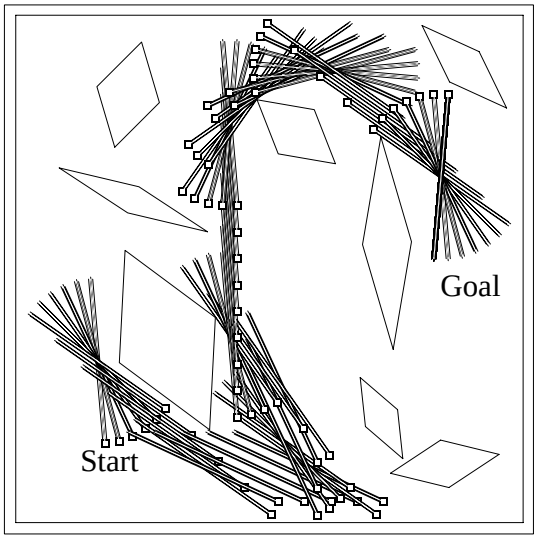

Figure 11: A rod-maneuvering task and its solution by prioritized sweeping. Reprinted from Moore and Atkeson (1993).

Example 5: Rod Maneuvering The objective in this task is to maneuver a rod around some awkwardly placed obstacles to a goal position in the fewest number of steps (Figure 11). 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. The distance of each movement is approximately 1/20 of the work space, and the rotation increment is 10 degrees. Translations are deterministic and quantized to one of 20×20 positions. The figure shows the obstacles and the shortest solution from start to goal, found by prioritized sweeping. This problem is still deterministic, but 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. •

Prioritized sweeping is clearly a powerful idea, but the algorithms that have been developed so far appear not to extend easily to more interesting cases. The greatest problem is that the algorithms appear to rely on the assumption of discrete states. When a change occurs at one state, these methods perform a computation on all the predecessor states that may have been affected. If function approximation is used to learn the model or the value function, then a single backup could influence a great many other states. It is not apparent how these states could be identified or processed efficiently. On the other hand, the general idea of focusing search on the states believed to have changed in value, and then on their predecessors, seems intuitively to be valid in general. Additional research may produce more general versions of prioritized sweeping.

Extensions of prioritized sweeping to stochastic environments are relatively straightforward. The model is maintained by keeping counts of the number of times each state–action pair has been experienced and of what the next states were. It is natural then to backup each pair not with a sample backup, as we have been using so far, but with a full backup, taking into account all possible next states and their probabilities of occurring.

5 Full vs. Sample Backups

The examples in the previous sections give some idea of the range of possibilities for combining methods of learning and planning. In the rest of this chapter, we analyze some of the component ideas involved, starting with the relative advantages of full and sample backups.

Much of this book has been about different kinds of backups, and we have considered a great many varieties. Focusing for the moment on one-step backups, they vary primarily along three binary dimensions. The first two dimensions are whether they back up state values or action values and whether they estimate the value for the optimal policy or for an arbitrary given policy. These two dimensions give rise to four classes of backups for approximating the four value functions,

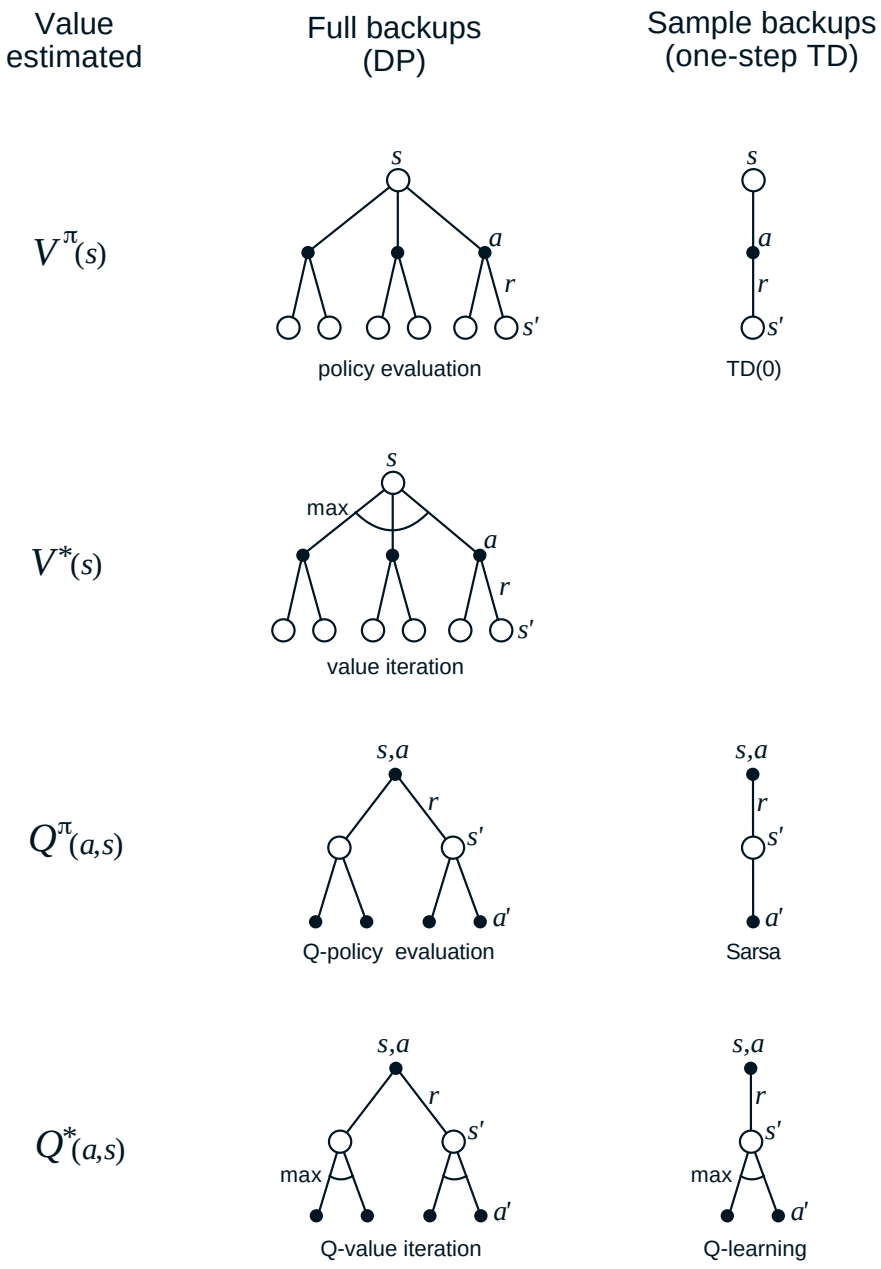

Figure 12: The one-step backups.

Q^*, V^*, Q^π, and V^π. The other binary dimension is whether the backups are *full* backups, considering all possible events that might happen, or *sample* backups, considering a single sample of what might happen. These three binary dimensions give rise to eight cases, seven of which correspond to specific algorithms, as shown in Figure 12. (The eighth case does not seem to correspond to any useful backup.) Any of these one-step backups can be used in planning methods. The Dyna-Q agents discussed earlier use Q^* sample backups, but they could just as well use Q^* full backups, or either full or sample Q^π backups. The Dyna-AC system uses V^π sample backups together with a learning policy structure. For stochastic problems, prioritized sweeping is always done using one of the full backups.

When we introduced one-step sample backups in Chapter 6, we presented them as substitutes for full backups. In the absence of a distribution model, full backups are not possible, but sample backups can be done using sample transitions from the environment or a sample model. Implicit in that point of view is that full backups, if possible, are preferable to sample backups. But are they? Full backups 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 full and sample backups for planning we must control for their different computational requirements.

For concreteness, consider the full and sample backups 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 state-transition probabilities, $\hat{P}^a_{ss'}$, and expected rewards, $\hat{R}^a_{ss'}$. The full backup for a state–action pair, s, a, is:

$$Q(s,a) \leftarrow \sum_{s'} \hat{P}^a_{ss'} \Big[\hat{R}^a_{ss'} + \gamma \max_{a'} Q(s',a') \Big]. \tag{1}$$

The corresponding sample backup for s, a, given a sample next state, s', is the Q-learning-like update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \Big[\hat{R}^a_{ss'} + \gamma \max_{a'} Q(s',a') - Q(s,a) \Big], \tag{2}$$

where α is the usual positive step-size parameter and the model's expected value of the reward, $\hat{R}^a_{ss'}$ is used in place of the sample reward that is used in applying Q-learning without a model.

The difference between these full and sample backups is significant to the extent that the environment is stochastic, specifically, to the extent that, given a state and action, many possible next states may occur with various probabilities. If only one next state is possible, then the full and sample backups given above are identical (taking $\alpha = 1$). If there are many possible next states, then there may be significant differences. In favor of the full backup 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 backup is in addition affected by sampling error. On the other hand, the sample backup is cheaper computationally because it considers only one next state, not all possible next states. In practice, the computation required by backup operations is usually dominated by the number of state–action pairs at which Q is evaluated. For a particular starting pair, s,a, let b be the *branching factor*, the number of possible next states, s', for which $\hat{P}^a_{ss'} > 0$. Then a full backup of this pair requires roughly b times as much computation as a sample backup.

If there is enough time to complete a full backup, then the resulting estimate is generally better than that of b sample backups because of the absence of sampling error. But if there is insufficient time to complete a full backup, then sample backups are always preferable because they at least make some improvement in the value estimate with fewer than b backups. In a large problem with many state–action pairs, we are often in the latter situation. With so many state–action pairs, full backups of all of them would take a very long time. Before that we may be much better off with a few sample backups at many state–action pairs than with full backups at a few pairs. Given a unit of computational effort, is it better devoted to a few full backups or to b-times as many sample backups?

Figure 13 shows the results of an analysis that suggests an answer to this question. It shows the estimation error as a function of computation time for full and sample backups for a variety of branching factors, b. The case considered is that in which 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 full backup reduces the error to zero upon its completion. In this case, sample backups reduce the error according to $\frac{1}{\sqrt{t}}\frac{b-1}{b}$ where t is the number of sample backups that have been performed (assuming sample averages, i.e., $\alpha = 1/t$). The key observation is that for moderately large b the error falls dramatically with a tiny fraction of b backups. For these cases, many state–action pairs could have their values improved dramatically, to within a few percent of the effect of a full backup, in the same time that one state–action pair could be backed up fully.

The advantage of sample backups shown in Figure 13 is probably an underestimate of the real effect. In a real problem, the values of the successor states would themselves be estimates updated by backups. By causing estimates to be more accurate sooner, sample backups will have a second advantage in that the values backed up from the successor states will be more accurate. These results suggest that sample backups are likely to be superior to full backups on problems with large stochastic branching factors and too many states to be solved exactly.

Exercise 5 The analysis above assumed that all of the b possible next states were equally likely to occur. Suppose instead that the distribution was highly skewed,

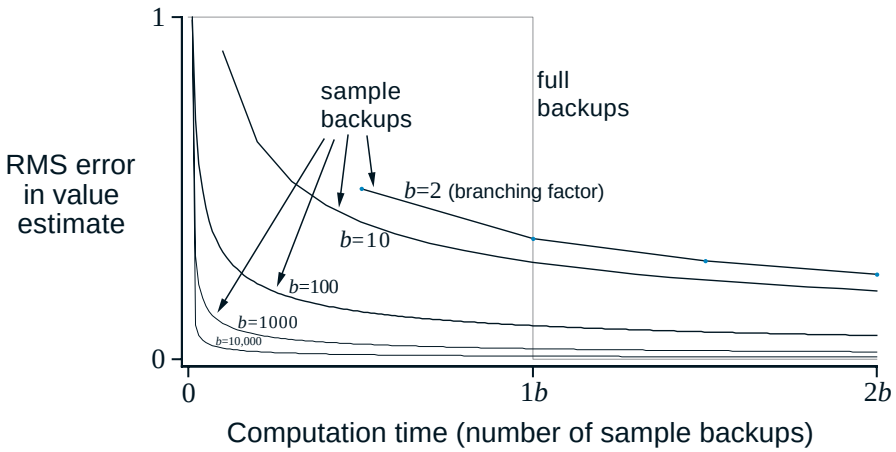

Figure 13: Comparison of efficiency of full and sample backups.

that some of the b states were much more likely to occur than most. Would this strengthen or weaken the case for sample backups over full backups? Support your answer.

6 Trajectory Sampling

In this section we compare two ways of distributing backups. The classical approach, from dynamic programming, is to perform sweeps through the entire state (or state–action) space, backing up 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. As we discussed in Chapter 4, exhaustive sweeps and the equal treatment of all states that they imply are not necessary properties of dynamic programming. In principle, backups can be distributed any way one likes (to assure convergence, all states or state–action pairs must be visited in the limit an infinite number of times), but in practice exhaustive sweeps are often used.

The second approach is to 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 backups 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

the 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. In other words, one simulates explicit individual trajectories and performs backups at the state or state–action pairs encountered along the way. We call this way of generating experience and backups *trajectory sampling*.

It is hard to imagine any efficient way of distributing backups according to the on-policy distribution other than by trajectory sampling. If one had an explicit representation of the on-policy distribution, then one could sweep through all states, weighting the backup of each according to the on-policy distribution, but this leaves us again with all the computational costs of exhaustive sweeps. Possibly one could sample and update individual state–action pairs from the distribution, but even if this could be done efficiently, what benefit would this provide over simulating trajectories? Even knowing the on-policy distribution in an explicit form is unlikely. The distribution changes whenever the policy changes, and computing the distribution requires computation comparable to a complete policy evaluation. Consideration of such other possibilities makes trajectory sampling seem both efficient and elegant.

Is the on-policy distribution of backups a good one? Intuitively it seems like a good choice, at least better than the uniform distribution. For example, if you are learning to play chess, you study positions that might arise in real games, not random positions of chess pieces. The latter may be valid states, but to be able to accurately value them is a different skill from evaluating positions in real games. We also know from the Chapter 8 that the on-policy distribution has significant advantages when function approximation is used. At the current time this is the only distribution for which we can guarantee convergence with general linear function approximation. Whether or not function approximation is used, one might expect on-policy focusing to significantly improve the speed of planning.

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 backed up over and over. We conducted a small experiment to assess the effect empirically. To isolate the effect of the backup distribution, we used entirely one-step full tabular backups, as defined by (1). In the *uniform* case, we cycled through all state–action pairs, backing up each in place, and in the *on-policy* case we simulated episodes, backing up each state–action pair that occurred under the current ε-greedy policy ($\varepsilon = 0.1$). The tasks were undiscounted episodic tasks, generated randomly as follows. From each of the $|\mathcal{S}|$ 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,

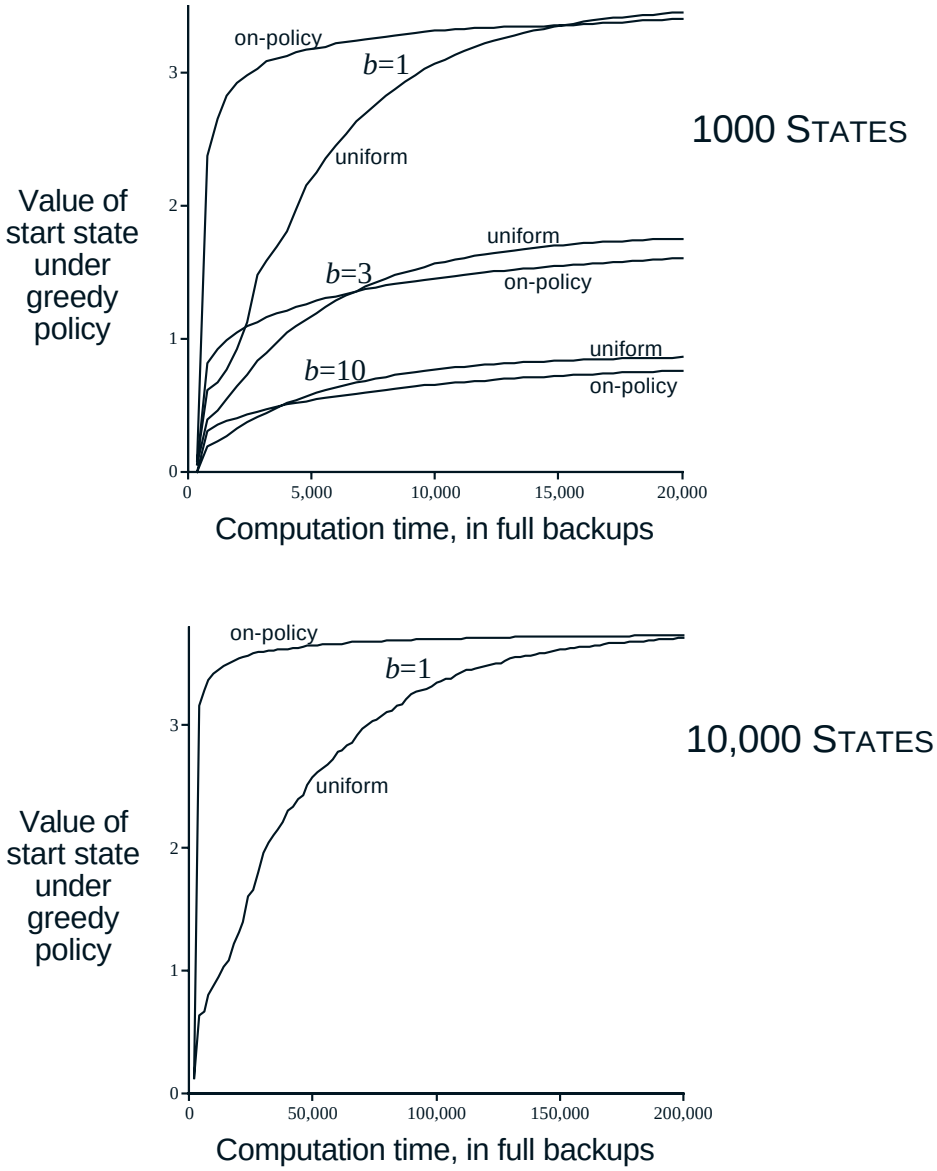

Figure 14: Relative efficiency of backups distributed uniformly across the state space versus focused on simulated on-policy trajectories. Results are for randomly generated tasks of two sizes and various branching factors, b.

b, was the same for all state–action pairs. In addition, on all transitions there was a 0.1 probability of transition to the terminal state, ending the episode. We used episodic tasks to get a clear measure of the quality of the current policy. At any point in the planning process one can stop and exhaustively compute $V^{\tilde{\pi}}(s_0)$, the true value of the start state under the greedy policy, $\tilde{\pi}$, given the current action-value function, Q, as an indication of how well the agent would do on a new episode on which it acted greedily (all the while assuming the model is correct).

The upper part of Figure 14 shows results averaged over 200 sample tasks with 1000 states and branching factors of 1, 3, and 10. The quality of the policies found is plotted as a function of the number of full backups completed. 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. In other experiments, we found that these effects also became stronger as the number of states increased. For example, the lower part of Figure 14 shows results for a branching factor of 1 for tasks with 10,000 states. In this case the advantage of on-policy focusing is large and long-lasting.

All of these results make sense. 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`directly` for problems in which a small subset of the state–action space is visited under the on-policy distribution.

Exercise 6 Some of the graphs in Figure 14 seem to be scalloped in their early portions, particularly the upper graph for $b = 1$ and the uniform distribution. Why do you think this is? What aspects of the data shown support your hypothesis?

Exercise 7 (programming) If you have access to a moderately large computer, try replicating the experiment whose results are shown in the lower part of Figure 14. Then try the same experiment but with $b = 3$. Discuss the meaning of your results.

7 Heuristic Search

The predominant state-space planning methods in artificial intelligence are collectively known as *heuristic search*. Although superficially different from the planning methods we have discussed so far in this chapter, heuristic search and some of its component ideas can be combined with these methods in useful ways. Unlike these methods, heuristic search is not concerned with changing the approximate, or "heuristic," value function, but only with making improved action selections given the current value function. In other words, heuristic search is planning as part of a policy computation.

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 max-backups (those for V^* and Q^*) discussed throughout this book. 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.

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 presented throughout this book. In a sense we have taken this approach all along. Our greedy and ε-greedy action-selection methods are not unlike heuristic search, albeit on a smaller scale. For example, to compute the greedy action given a model and a state-value function, we must look ahead from each possible action to each possible next state, backup 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.

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.[1] 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.

[1] There are interesting exceptions to this. See, e.g., Pearl (1984).

A good example is provided by Tesauro's grandmaster-level backgammon player, TD-Gammon (Section 11.1). This system used TD(λ) 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.

So far we have emphasized heuristic search as an action-selection technique, but this may not be its most important aspect. Heuristic search also suggests ways of selectively distributing backups that may lead to better and faster approximation of the optimal value function. A great deal of research on heuristic search has been devoted to making the search as efficient as possible. The search tree is grown selectively, deeper along some lines and shallower along others. For example, the search tree is often deeper for the actions that seem most likely to be best, and shallower for those that the agent will probably not want to take anyway. Can we use a similar idea to improve the distribution of backups? Perhaps it can be done by preferentially updating state–action pairs whose values appear to be close to the maximum available from the state. To our knowledge, this and other possibilities for distributing backups based on ideas borrowed from heuristic search have not yet been explored.

We should not overlook the most obvious way in which heuristic search focuses backups: on the current state. 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. You may spend more of your life playing chess than checkers, but when you play checkers, it pays to think about checkers and about your particular checkers position, your likely next moves, and successor positions. However you select actions, it is these states and actions that are of highest priority for backups and where you most urgently want your approximate value function to be accurate. Not only should your computation be preferentially devoted to imminent events, but so should your limited memory resources. In chess, for example, 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.

The distribution of backups 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

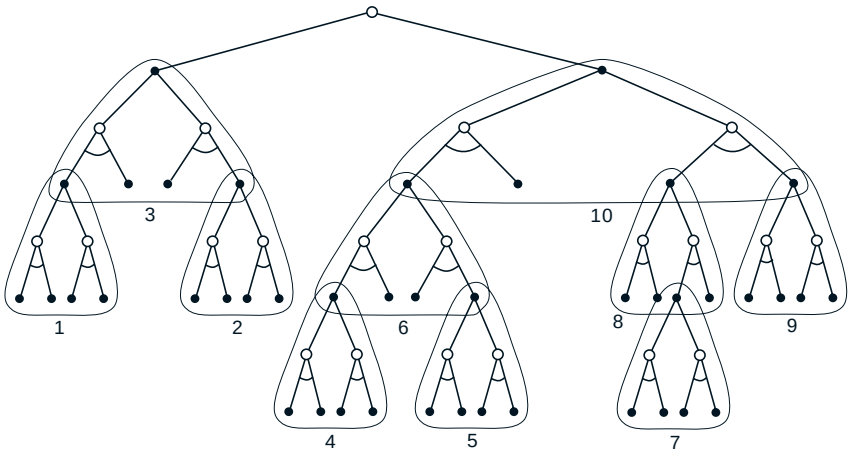

Figure 15: The deep backups of heuristic search can be implemented as a sequence of one-step backups (shown here outlined). The ordering shown is for a selective depth-first search.

individual, one-step backups from bottom up, as suggested by Figure 15. If the backups are ordered in this way and a table-lookup representation is used, then exactly the same backup would be achieved as in heuristic search. Any state-space search can be viewed in this way as the piecing together of a large number of individual one-step backups. Thus, the performance improvement observed with deeper searches is not due to the use of multistep backups as such. Instead, it is due to the focus and concentration of backups on states and actions immediately downstream from the current state. By devoting a large amount of computation specifically relevant to the candidate actions, a much better decision can be made than by relying on unfocused backups.

8 Summary

We have presented a perspective emphasizing the surprisingly 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 backup 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.

It is straightforward to integrate incremental planning methods with acting

and model-learning. Planning, acting, and model-learning interact in a circular fashion (Figure 2), each producing what the other needs to improve; no other interaction among them is either required or prohibited. 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.

In this chapter we have touched upon a number of dimensions of variation among state-space planning methods. One of the most important of these is the distribution of backups, that is, of the focus of search. Prioritized sweeping focuses on the predecessors of states whose values have recently changed. Heuristic search applied to reinforcement learning focuses, inter alia, on the successors of the current state. Trajectory sampling is a convenient way of focusing on the on-policy distribution. All of these approaches can significantly speed planning and are current topics of research.

Another interesting dimension of variation is the size of backups. The smaller the backups, the more incremental the planning methods can be. Among the smallest backups are one-step sample backups. We presented one study suggesting that one-step sample backups may be preferable on very large problems. A related issue is the depth of backups. In many cases deep backups can be implemented as sequences of shallow backups.

9 Bibliographical and Historical Remarks

1 The overall view of planning and learning presented here has developed gradually over a number of years, in part by the authors (Sutton, 1990, 1991a, 1991b; Barto, Bradtke, and Singh, 1991, 1995; Sutton and Pinette, 1985; Sutton and Barto, 1981b); it has been strongly influenced by Agre and Chapman (1990; Agre 1988), Bertsekas and Tsitsiklis (1989), Singh (1993), and others. The authors were also strongly influenced by psychological studies of latent learning (Tolman, 1932) and by psychological views of the nature of thought (e.g., Galanter and Gerstenhaber, 1956; Craik, 1943; Campbell, 1960; Dennett, 1978).

2–3 The terms *direct* and *indirect*, which we use to describe different kinds of reinforcement learning, are from the adaptive control literature (e.g., Goodwin and Sin, 1984), where they are used to make the same kind of distinction. The term *system identification* is used in adaptive control for what we call *model-learning* (e.g., Goodwin and Sin, 1984; Ljung and Söderstrom, 1983; Young, 1984). The Dyna architecture is due to Sutton (1990), and the results in these sections are based on results reported there.

4 Prioritized sweeping was developed simultaneously and independently by Moore and Atkeson (1993) and Peng and Williams (1993). The results in Figure 10 are due to Peng and Williams (1993). The results in Figure 11 are due to Moore and Atkeson.

5 This section was strongly influenced by the experiments of Singh (1993).

7 For further reading on heuristic search, the reader is encouraged to consult texts and surveys such as those by Russell and Norvig (1995) and Korf (1988). Peng and Williams (1993) explored a forward focusing of backups much as is suggested in this section.

References

Agre, PE (1988). *The Dynamic Structure of Everyday Life.* PhD thesis Massachusetts Institute of Technology, Cambridge, MA. AI-TR 1085, MIT Artificial Intelligence Laboratory.

Agre, PE and Chapman, D (1990). What are plans for? *Robotics and Autonomous Systems,* **6**, 17–34.

Barto, AG, Bradtke, SJ, and Singh, SP (1991). Real-time learning and control using asynchronous dynamic programming. Technical Report 91-57 Department of Computer and Information Science, University of Massachusetts Amherst, MA.

Barto, AG, Bradtke, SJ, and Singh, SP (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence,* **72**, 81–138.

Bertsekas, DP and Tsitsiklis, JN (1989). *Parallel and Distributed Computation: Numerical Methods.* Englewood Cliffs, NJ: Prentice-Hall.

Campbell, DT (1959). Blind variation and selective survival as a general strategy in knowledge-processes. In: *Self-Organizing Systems*, (MC Yovits and S Cameron, eds) pp. 205–231. New York: Pergamon.

Craik, KJW (1943). *The Nature of Explanation.* Cambridge: Cambridge University Press.

Dennett, DC (1978). *Brainstorms* chapter Why the Law-of-Effect Will Not Go Away, pp. 71–89. Cambridge, MA: Bradford/MIT Press.

Galanter, E and Gerstenhaber, M (1956). On thought: The extrinsic theory. *Psychological Review,* , **63**, 218–227.

Goodwin, GC and Sin, KS (1984). *Adaptive Filtering Prediction and Control.* Englewood Cliffs, N.J.: Prentice-Hall.

Korf, RE (1988). Optimal path finding algorithms. In: *Search in Artificial Intelligence*, (LN Kanal and V Kumar, eds) pp. 223–267. Berlin: Springer Verlag.

Ljung, L and Söderstrom, T (1983). *Theory and Practice of Recursive Identification.* Cambridge, MA: MIT Press.

Moore, AW and Atkeson, CG (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning,* **13**, 103–130.

Pearl, J (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley.

Peng, J and Williams, RJ (1993). Efficient learning and planning within the Dyna framework. *Adaptive Behavior,* **1** (4).

Russell, S and Norvig, P (1995). *Artificial Intelligence: A Modern Approach.* Englewood Cliffs, NJ: Prentice Hall.

Singh, SP (1993). *Learning to Solve Markovian Decision Processes.* PhD thesis University of Massachusetts, Amherst. Appeared as CMPSCI Technical Report 93-77.

Sutton, RS (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In: *Proceedings of the Seventh International Conference on Machine Learning* pp. 216–224, San Mateo, CA: Morgan Kaufmann.

Sutton, RS (1991a). Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bulletin,* **2**, 160–163. Also appeared in *Working Notes of the 1991 AAAI Spring Symposium*, pages 151–155.

Sutton, RS (1991b). Planning by incremental dynamic programming. In: *Proceedings of the Eighth International Workshop on Machine Learning*, (LA Birnbaum and GC Collins, eds) pp. 353–357, San Mateo, CA: Morgan Kaufmann.

Sutton, RS and Barto, AG (1981). An adaptive network that constructs and uses an internal model of its world. *Cognition and Brain Theory,* **3**, 217–246.

Sutton, RS and Pinette, B (1985). The learning of world models by connectionist networks. In: *Proceedings of the Seventh Annual Conference of the Cognitive Science Society* , Irvine, CA:.

Tolman, EC (1932). *Purposive Behavior in Animals and Men.* New York: Century.

Young, P (1984). *Recursive Estimation and Time-Series Analysis.* Springer-Verlag.