

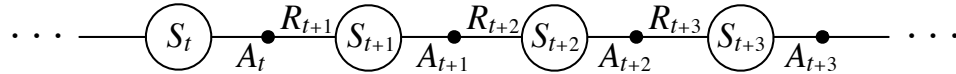
or the minimum squared-error estimates, they can be understood as moving roughly in these directions. Nonbatch TD(0) may be faster than constant- α MC because it is moving toward a better estimate, even though it is not getting all the way there. At the current time nothing more definite can be said about the relative efficiency of on-line TD and Monte Carlo methods.

Finally, it is worth noting that although the certainty-equivalence estimate is in some sense an optimal solution, it is almost never feasible to compute it directly. If N is the number of states, then just forming the maximum-likelihood estimate of the process may require N^2 memory, and computing the corresponding value function requires on the order of N^3 computational steps if done conventionally. In these terms it is indeed striking that TD methods can approximate the same solution using memory no more than N and repeated computations over the training set. On tasks with large state spaces, TD methods may be the only feasible way of approximating the certainty-equivalence solution.

6.4 Sarsa: On-Policy TD Control

We turn now to the use of TD prediction methods for the control problem. As usual, we follow the pattern of generalized policy iteration (GPI), only this time using TD methods for the evaluation or prediction part. As with Monte Carlo methods, we face the need to trade off exploration and exploitation, and again approaches fall into two main classes: on-policy and off-policy. In this section we present an on-policy TD control method.

The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $q_\pi(s, a)$ for the current behavior policy π and for all states s and actions a . This can be done using essentially the same TD method described above for learning v_π . Recall that an episode consists of an alternating sequence of states and state-action pairs:



In the previous section we considered transitions from state to state and learned the values of states. **Now we consider transitions from state-action pair to state-action pair, and learn the value of state-action pairs.** Formally these cases are identical: they are both Markov chains with a reward process. The theorems assuring the convergence of state values under TD(0) also apply

```

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
  Initialize  $S$ 
  Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
  Repeat (for each step of episode):
    Take action  $A$ , observe  $R, S'$ 
    Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
     $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$ 
     $S \leftarrow S'; A \leftarrow A'$ 
  until  $S$  is terminal

```

Figure 6.9: Sarsa: An on-policy TD control algorithm.

to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \quad (6.5)$$

This update is done after every transition from a nonterminal state S_t . If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state–action pair to the next. This quintuple gives rise to the name *Sarsa* for the algorithm.

It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate q_π for the behavior policy π , and at the same time change π toward greediness with respect to q_π . The general form of the Sarsa control algorithm is given in Figure 6.9.

The convergence properties of the Sarsa algorithm depend on the nature of the policy’s dependence on q . For example, one could use ϵ -greedy or ϵ -soft policies. According to Satinder Singh (personal communication), Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state–action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ϵ -greedy policies by setting $\epsilon = 1/t$), but this result has not yet been published in the literature.

Example 6.5: Windy Gridworld Figure 6.10 shows a standard gridworld, with start and goal states, but with one difference: there is a crosswind upward through the middle of the grid. The actions are the standard four—**up**, **down**, **right**, and **left**—but in the middle region the resultant next states are shifted upward by a “wind,” the strength of which varies from column to column. The strength of the wind is given below each column, in number of cells shifted

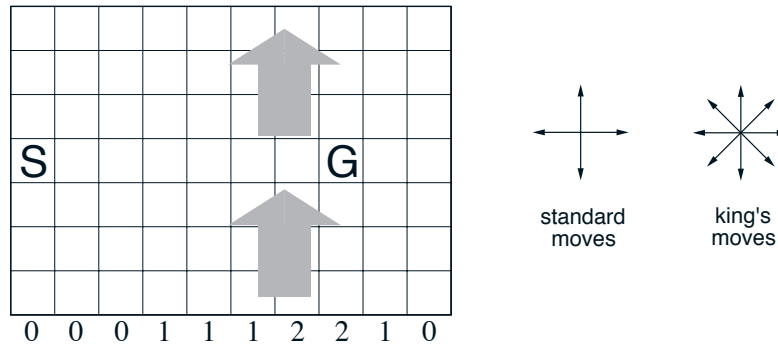


Figure 6.10: Gridworld in which movement is altered by a location-dependent, upward “wind.”

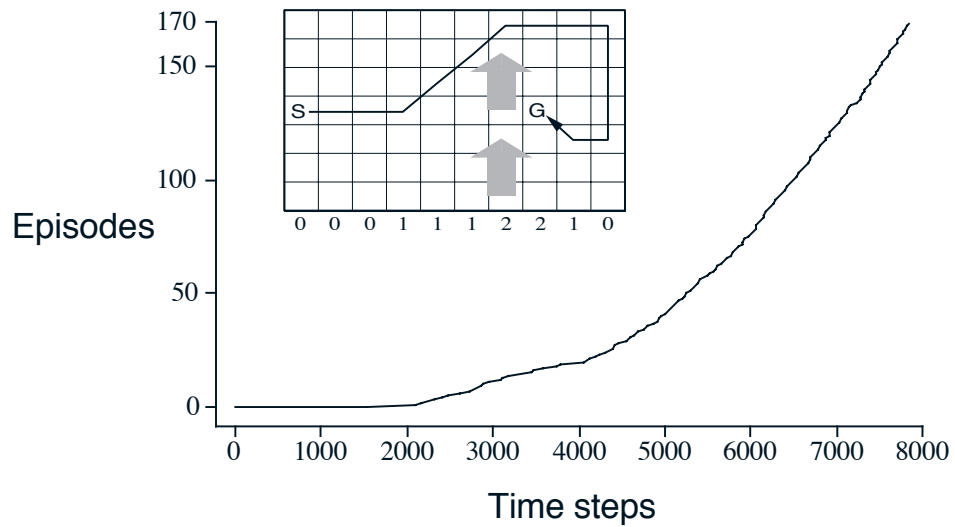


Figure 6.11: Results of Sarsa applied to the windy gridworld.

upward. For example, if you are one cell to the right of the goal, then the action `left` takes you to the cell just above the goal. Let us treat this as an undiscounted episodic task, with constant rewards of -1 until the goal state is reached. Figure 6.11 shows the result of applying ε -greedy Sarsa to this task, with $\varepsilon = 0.1$, $\alpha = 0.5$, and the initial values $Q(s, a) = 0$ for all s, a . The increasing slope of the graph shows that the goal is reached more and more quickly over time. By 8000 time steps, the greedy policy (shown inset) was long since optimal; continued ε -greedy exploration kept the average episode length at about 17 steps, two more than the minimum of 15. Note that Monte Carlo methods cannot easily be used on this task because termination is not guaranteed for all policies. If a policy was ever found that caused the agent to stay in the same state, then the next episode would never end. Step-by-step learning methods such as Sarsa do not have this problem because they quickly learn *during the episode* that such policies are poor, and switch to something else. ■

6.5 Q-Learning: Off-Policy TD Control

One of the most important breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as *Q-learning* (Watkins, 1989). Its simplest form, *one-step Q-learning*, is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \quad (6.6)$$

In this case, the learned action-value function, Q , directly approximates q_* , the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. As we observed in Chapter 5, this is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_* . The Q-learning algorithm is shown in procedural form in Figure 6.12.

What is the backup diagram for Q-learning? The rule (6.6) updates a state-action pair, so the top node, the root of the backup, must be a small, filled action node. The backup is also *from* action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate

```

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
  Initialize  $S$ 
  Repeat (for each step of episode):
    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
    Take action  $A$ , observe  $R, S'$ 
     $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
     $S \leftarrow S'$ ;
  until  $S$  is terminal

```

Figure 6.12: Q-learning: An off-policy TD control algorithm.

taking the maximum of these “next action” nodes with an arc across them (Figure 3.7). Can you guess now what the diagram is? If so, please do make a guess before turning to the answer in Figure 6.14.

Example 6.6: Cliff Walking This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Consider the gridworld shown in the upper part of Figure 6.13. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the the region marked “The Cliff.” Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start. The lower part of the figure shows the performance of the Sarsa and Q-learning methods with ϵ -greedy action selection, $\epsilon = 0.1$. After an initial transient, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ϵ -greedy action selection. Sarsa, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its on-line performance is worse than that of Sarsa, which learns the roundabout policy. Of course, if ϵ were gradually reduced, then both methods would asymptotically converge to the optimal policy. ■

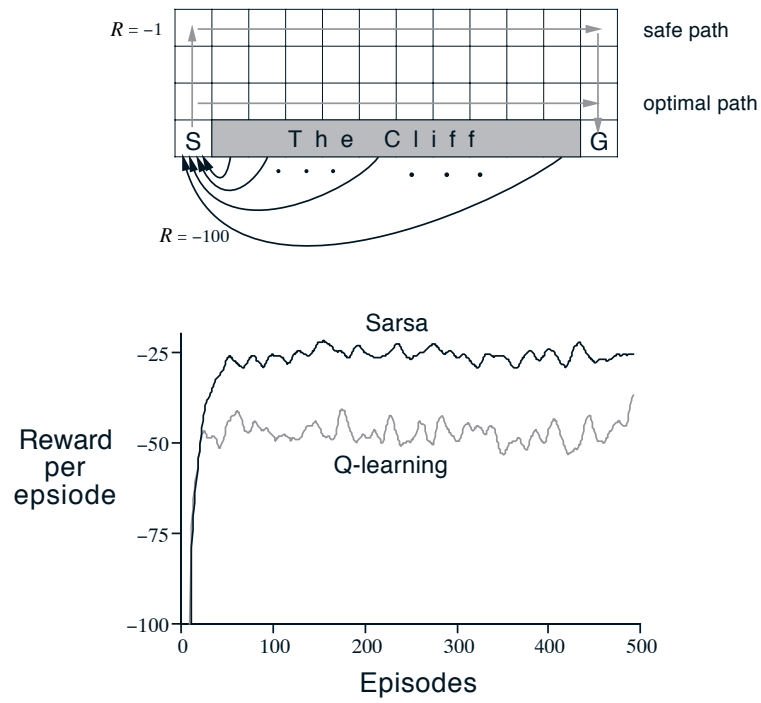


Figure 6.13: The cliff-walking task. The results are from a single run, but smoothed.



Figure 6.14: The backup diagram for Q-learning.