4.3 Policy Iteration

Once a policy, π, has been improved using vπ to yield a better policy, π′, we can then compute vπ′ and improve it again to yield an even better π″. We can thus obtain a sequence of monotonically improving policies and value functions: π0 E −→vπ0 I −→π1 E −→vπ1 I −→π2 E −→· · · I −→π∗ E −→v∗, where E −→denotes a policy evaluation and I −→denotes a policy improvement. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations. This way of finding an optimal policy is called policy iteration. A complete algorithm is given in the box below. Note that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation (presumably because the value function changes little from one policy to the next). Policy Iteration (using iterative policy evaluation) for estimating π ≈π∗ 1. Initialization V (s) ∈R and π(s) ∈A(s) arbitrarily for all s ∈S 2. Policy Evaluation Loop: ∆←0 Loop for each s ∈S: v ←V (s) V (s) ←P s′,r p(s′, r|s, π(s)) r + γV (s′) ∆←max(∆, |v −V (s)|) until ∆< θ (a small positive number determining the accuracy of estimation) 3. Policy Improvement policy-stable ←true For each s ∈S: old-action ←π(s) π(s) ←argmaxa P s′,r p(s′, r|s, a) r + γV (s′) If old-action ̸= π(s), then policy-stable ←false If policy-stable, then stop and return V ≈v∗and π ≈π∗; else go to 2 Policy iteration often converges in surprisingly few iterations. This is illustrated by the example in Figure 4.1. The bottom-left diagram shows the value function for the equiprobable random policy, and the bottom-right diagram shows a greedy policy for this value function. The policy improvement theorem assures us that these policies are better than the original random policy. In this case, however, these policies are not just better, but optimal, proceeding to the terminal states in the minimum number of steps. In this example, policy iteration would find the optimal policy after just one iteration. Exercise 4.4 The policy iteration algorithm on the previous page has a subtle bug in that it may never terminate if the policy continually switches between two or more policies that are equally good. This is ok for pedagogy, but not for actual use. Modifiy the pseudocode so that convergence is guaranteed. □ Exercise 4.5 How would policy iteration be defined for action values? Give a complete algorithm for computing q∗, analogous to that on page 80 for computing v∗. Please pay special attention to this exercise, because the ideas involved will be used throughout the rest of the book. □ Exercise 4.6 Suppose you are restricted to considering only policies that are ε-soft, meaning that the probability of selecting each action in each state, s, is at least ε/|A(s)|. Describe qualitatively the changes that would be required in each of the steps 3, 2, and 1, in that order, of the policy iteration algorithm for v∗(page 80). □ Example 4.2: Jack's Car Rental Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited $10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of $2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is n is λn n! e−λ, where λ is the expected number. Suppose λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be γ = 0.9 and formulate this as a continuing finite MDP, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations

overnight. Figure 4.2 shows the sequence of policies found by policy iteration starting from the policy that never moves any cars. Exercise 4.7 (programming) Write a program for policy iteration and re-solve Jack's car rental problem with the following changes. One of Jack's employees at the first location rides a bus home each night and lives near the second location. She is happy to shuttle one car to the second location for free. Each additional car still costs $2, as do all cars moved in the other direction. In addition, Jack has limited parking space at each location. If more than 10 cars are kept overnight at a location (after any moving of cars), then an 4V 612 #Cars at second location 0 420 20 0 20 #Cars at first location 1 1 5 !1 !2 -4 4 3 2 4 3 2 !3 0 0 5 !1 !2!3 !4 1 2 3 4 0 "1 "0 "2 !3 !4 !2 0 1 2 3 4 !1 "3 2 !4 !3 !2 0 1 3 4 5 !1 "4 #Cars at second location #Cars at first location 5 20 0 0 20 vk Figure 4.2: The sequence of policies found by policy iteration on Jack's car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative numbers indicate transfers from the second location to the first). Each successive policy is a strict improvement over the previous policy, and the last policy is optimal. additional cost of $4 must be incurred to use a second parking lot (independent of how many cars are kept there). These sorts of nonlinearities and arbitrary dynamics often occur in real problems and cannot easily be handled by optimization methods other than dynamic programming. To check your program, first replicate the results given for the original problem. If your computer is too slow for the full problem, cut all the numbers of cars in half. □

4.4 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to vπ occurs only in the limit. Must we wait for exact convergence, or can we stop short of that? The example in Figure 4.1 certainly suggests that it may be possible to truncate policy evaluation. In that example, policy evaluation iterations beyond the first three have no effect on the corresponding greedy policy. In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called value iteration. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps: vk+1(s) .= max a E[Rt+1 + γvk(St+1) | St =s, At =a] = max a X s′,r p(s′, r|s, a) h r + γvk(s′) i , (4.10) for all s ∈S. For arbitrary v0, the sequence {vk} can be shown to converge to v∗under the same conditions that guarantee the existence of v∗. Another way of understanding value iteration is by reference to the Bellman optimality equation (4.1). Note that value iteration is obtained simply by turning the Bellman optimality equation into an update rule. Also note how the value iteration update is identical to the policy evaluation update (4.5) except that it requires the maximum to be taken over all actions. Another way of seeing this close relationship is to compare the backup diagrams for these algorithms on page 59 (policy evaluation) and on the left of Figure 3.4 (value iteration). These two are the natural backup operations for computing vπ and v∗. Finally, let us consider how value iteration terminates. Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to v∗. In practice, we stop once the value function changes by only a small amount in a sweep. The box below shows a complete algorithm with this kind of termination condition. Value Iteration, for estimating π ≈π∗ Algorithm parameter: a small threshold θ > 0 determining accuracy of estimation Initialize V (s), for all s ∈S+, arbitrarily except that V (terminal) = 0 Loop: | ∆←0 | Loop for each s ∈S: | v ←V (s) | V (s) ←maxa P s′,r p(s′, r|s, a) r + γV (s′) | ∆←max(∆, |v −V (s)|) until ∆< θ Output a deterministic policy, π ≈π∗, such that π(s) =

argmaxa P s′,r p(s′, r|s, a) r + γV (s′) Value iteration effectively combines, in each of its sweeps, one sweep of policy eval- uation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep. In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation updates and some of which use value iteration updates. Because the max operation in (4.10) is the only difference between these updates, this just means that the max operation is added to some sweeps of policy evaluation. All of these algorithms converge to an optimal policy for discounted finite MDPs. Example 4.3: Gambler's Problem A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake. The game ends when the gambler wins by reaching his goal of $100, or loses by running out of money. On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars. This problem can be formulated as an undiscounted, episodic, finite MDP. The state is the gam- 99 75 50 25 1 1 10 20 30 40 50 1 0 0.2 0.4 0.6 0.8 1 25 50 75 99 Capital Capital Value estimates Final policy (stake) sweep 1 sweep 2 sweep 3 sweep 32 Figure 4.3: The solution to the gambler's problem for ph = 0.4. The upper graph shows the value func- tion found by successive sweeps of value iteration. The lower graph shows the final policy. bler's capital, s ∈ {1, 2, . . . , 99} and the actions are stakes, a ∈ {0, 1, . . . , min(s, 100 −s)}. The re- ward is zero on all transitions except those on which the gambler reaches his goal, when it is +1. The state- value function then gives the prob- ability of winning from each state. A policy is a mapping from levels of capital to stakes. The optimal policy maximizes the probability of reaching the goal. Let ph denote the probability of the coin coming up heads. If ph is known, then the entire problem is known and it can be solved, for instance, by value iter- ation. Figure 4.3 shows the change in the value function over successive sweeps of value iteration, and the final policy found, for the case of ph = 0.4. This policy is optimal, but not unique. In fact, there is a whole family of optimal policies, all corre- sponding to ties for the argmax ac- tion selection with respect to the op- timal value function. Can you guess what the entire family looks like? Exercise 4.8 Why does the optimal policy for the gambler's problem have such a curious form? In particular, for capital of 50 it bets it all on one flip, but for capital of 51 it does not. Why is this a good policy? □ Exercise 4.9 (programming) Implement value iteration for the gambler's problem and solve it for ph = 0.25 and ph = 0.55. In programming, you may find it convenient 4.5. Asynchronous Dynamic Programming 85 to introduce two dummy states corresponding to termination with capital of 0 and 100, giving them values of 0 and 1 respectively. Show your results graphically, as in Figure 4.3. Are your results stable as θ →0? □ Exercise 4.10 What is the analog of the value iteration update (4.10) for action values, qk+1(s, a)?an infinite number of times (the sequence could even be stochastic). (In the undiscounted episodic case, it is possible that there are some orderings of updates that do not result in convergence, but it is relatively easy to avoid these.) Similarly, it is possible to intermix policy evaluation and value iteration updates to produce a kind of asynchronous truncated policy iteration. Although the details of this and other more unusual DP algorithms are beyond the scope of this book, it is clear that a few different updates form building blocks that can be used flexibly in a wide variety of sweepless DP algorithms. Of course, avoiding sweeps does not necessarily mean that we can get away with less computation. It just means that an algorithm does not need to get locked into any hopelessly long sweep before it can make progress improving a policy. We can try to take advantage of this flexibility by selecting the states to which we apply updates so as to improve the algorithm's rate of progress. We can try to order the updates to let value information propagate from state to state in an efficient way. Some states may not need their values updated as often as others. We might even try to skip updating some states entirely if they are not relevant to optimal behavior. Some ideas for doing this are

discussed in Chapter 8. Asynchronous algorithms also make it easier to intermix computation with real-time