9.5.3 Value Iteration

Value iteration is a method of computing an optimal MDP policy and its value.

Value iteration starts at the "end" and then works backward, refining an estimate of either Q^* or V^*. There is really no end, so it uses an arbitrary end point. Let V_k be the value function assuming there are k stages to go, and let Q_k be the Q-function assuming there are k stages to go. These can be defined recursively. Value iteration starts with an arbitrary function V_0 and uses the following equations to get the functions for $k+1$ stages to go from the functions for k stages to go:

$$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma V_k(s')) \quad \text{for } k \geq 0$$

$$V_k(s) = \max_a Q_k(s,a) \quad \text{for } k>0.$$

It can either save the $V[S]$ array or the $Q[S,A]$ array. Saving the V array results in less storage, but it is more difficult to determine an optimal action, and one more iteration is needed to determine which action results in the greatest value.

```
1: Procedure Value_Iteration(S,A,P,R,θ)
2:         Inputs
3:                 S is the set of all states
4:                 A is the set of all actions
5:                 P is state transition function specifying P(s'|s,a)
6:                 R is a reward function R(s,a,s')
7:                 θ a threshold, θ>0
8:         Output
9:                 π[S] approximately optimal policy
10:                V[S] value function
11:        Local
12:                real array V_k[S] is a sequence of value functions
13:                action array π[S]
14:        assign V_0[S] arbitrarily
15:        k ←0
16:        repeat
17:                k ←k+1
18:                for each state s do
```

19: $V_k[s] = max_a \sum_{s'} P(s'|s,a) (R(s,a,s') + \gamma V_{k-1}[s'])$
20: **until** $\forall s \; |V_k[s] - V_{k-1}[s]| < \theta$
21: **for each** state s **do**
22: $\pi[s] = argmax_a \sum_{s'} P(s'|s,a) (R(s,a,s') + \gamma V_k[s'])$
23: **return** π, V_k

Figure 9.14: Value iteration for MDPs, storing V

Figure 9.14 shows the value iteration algorithm when the V array is stored. This procedure converges no matter what is the initial value function V_0. An initial value function that approximates V^* converges quicker than one that does not. The basis for many abstraction techniques for MDPs is to use some heuristic method to approximate V^* and to use this as an initial seed for value iteration.

Example 9.26: Consider the 9 squares around the *+10* reward of Example 9.25. The discount is $\gamma=0.9$. Suppose the algorithm starts with $V_0[s]=0$ for all states s.

The values of V_1, V_2, and V_3 (to one decimal point) for these nine cells is

0	0	*-0.1*
0	10	*-0.1*
0	0	*-0.1*

0	*6.3*	*-0.1*
6.3	*9.8*	*6.2*
0	*6.3*	*-0.1*

4.5	*6.2*	*4.4*
6.2	*9.7*	*6.6*
4.5	*6.1*	*4.4*

After the first step of value iteration, the nodes get their immediate expected reward. The center node in this figure is the *+10* reward state. The right nodes have a value of *-0.1*, with the optimal actions being up, left, and down; each of these has a *0.1* chance of crashing into the wall for a reward of *-1*.

The middle grid shows V_2, the values after the second step of value iteration. Consider the node that is immediately to the left of the *+10* rewarding state. Its optimal value is to go to the right; it has a 0.7 chance of getting a reward of 10 in the following state, so that is worth 9 (10 times the discount of *0.9*) to it now. The expected reward for the other possible resulting states is *0*. Thus, the value of this state is $0.7 \times 9 = 6.3$.

Consider the node immediately to the right of the *+10* rewarding state after the second step of value iteration. The agent's optimal action in this state is to go left. The value of this state is

	Prob	*Reward*		*Future Value*	
	0.7×(	*0*	+	*0.9×10)*	*Agent goes left*
+	*0.1×(*	*0*	+	*0.9×-0.1)*	*Agent goes up*

$+ \quad 0.1\times(\quad -1 \quad + \quad 0.9\times-0.1) \quad$ *Agent goes right*

$+ \quad 0.1\times(\quad 0 \quad + \quad 0.9\times-0.1) \quad$ *Agent goes down*

which evaluates to 6.173.

Notice also how the *+10* reward state now has a value less than 10. This is because the agent gets flung to one of the corners and these corners look bad at this stage.

After the next step of value iteration, shown on the right-hand side of the figure, the effect of the +10 reward has progressed one more step. In particular, the corners shown get values that indicate a reward in 3 steps.

An applet is available on the book web site showing the details of value iteration for this example.

The value iteration algorithm of Figure 9.14 has an array for each stage, but it really only must store the current and the previous arrays. It can update one array based on values from the other.

A common refinement of this algorithm is **asynchronous value iteration**. Rather than sweeping through the states to create a new value function, asynchronous value iteration updates the states one at a time, in any order, and store the values in a single array. Asynchronous value iteration can store either the *Q[s,a]* array or the *V[s]* array. Figure 9.15 shows asynchronous value iteration when the *Q* array is stored. It converges faster and uses less space than value iteration and is the basis of some of the algorithms for reinforcement learning. Termination can be difficult to determine if the agent must guarantee a particular error, unless it is careful about how the actions and states are selected. Often, this procedure is run indefinitely and is always prepared to give its best estimate of the optimal action in a state when asked.

```
1: Procedure Asynchronous_Value_Iteration(S,A,P,R)
2:         Inputs
3:                 S is the set of all states
4:                 A is the set of all actions
5:                 P is state transition function specifying P(s'|s,a)
6:                 R is a reward function R(s,a,s')
7:         Output
8:                 π[s] approximately optimal policy
9:                 Q[S,A] value function
10:        Local
11:                real array Q[S,A]
12:                action array π[S]
13:        assign Q[S,A] arbitrarily
14:        repeat
15:                select a state s
16:                select an action a
17:                Q[s,a] = ∑_s' P(s'|s,a) (R(s,a,s')+ γmax_a' Q[s',a'])
18:        until termination
19:        for each state s do
20:                π[s] = argmax_a Q[s,a]
21:        return π,Q
```

Figure 9.15: Asynchronous value iteration for MDPs

Asynchronous value iteration could also be implemented by storing just the *V[s]* array. In that case, the algorithm selects a state *s* and carries out the update:

$$V[s]=max_a \sum_{s'} P(s'|s,a) \ (R(s,a,s')+ \gamma V[s']).$$

Although this variant stores less information, it is more difficult to extract the policy. It requires one extra

backup to determine which action a results in the maximum value. This can be done using

$$\pi[s] = \text{argmax}_a \sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma V[s']).$$

Example 9.27: In Example 9.26, the state one step up and one step to the left of the +10 reward state only had its value updated after three value iterations, in which each iteration involved a sweep through all of the states.

In asynchronous value iteration, the *+10* reward state can be chosen first. Then the node to its left can be chosen, and its value will be $0.7\times0.9\times10=6.3$. Then the node above that node could be chosen, and its value would become $0.7\times0.9\times6.3=3.969$. Note that it has a value that reflects that it is close to a *+10* reward after considering 3 states, not 300 states, as does value iteration.