CS181 Lecture 3 — Markov Decision Processes

Avi Pfeffer

Last time we discussed the decision-theoretic basis for AI. The maximum expected utility principle (MEU) provides the foundation for what it means to be a rational agent: a rational agent is one that tries to maximize its expected utility, given its uncertainty about the world.

We focused on single-shot decisions, where the agent gets some evidence, chooses an action, and receives a reward. Today we begin to focus on the more realistic situation of repeated interactions. The agent gets some information about the world, chooses an action, receives a reward, gets some more information, chooses another action, receives another reward, and so on. This type of agent needs to consider not only its immediate reward when making its decision, but also its utility in the long run.

1 Markov Decision Processes

The basic framework for studying utility-maximizing agents that have repeated interactions with the world is called a Markov Decision Process, abbreviated MDP. The framework consists of the following elements:

- A set S of possible states of the world. We will assume that S is a finite set $\{1,...,n\}$.
- A set A of possible actions $\{a_1,...,a_m\}$.

- A reward model R, assigning a reward R(i, a) to each state i and action a. This is the most general formulation. Sometimes it will be simpler to associate rewards with states, rather than with states and actions. When we do that, we will mean that the reward for taking any action in the state is the reward associated with the state.

- A transition model T_a for each action a. The transition model tells us how actions affect the state of the world, and also how the world state changes exogenously. T_a is a matrix: $T_{a\,ij}$ specifies the probability of getting to state j if the agent takes action a in state i. We will also use the notation $T_{a\,i}$, to denote the probability distribution over the next state, given that action a is taken in state i.

This framework describes an agent with repeated interactions with the world. The agent starts in some state i in S. The agent then gets to choose an action

a. The agent receives reward R(i, a), while the world transitions to a new state j, according to the probability distribution $T_{a\,i}$. Then the cycle continues: the agent chooses another action b, and the world transitions to k according to the distribution $T_{b\,j}$, the agent receiving R(j, b). And so on.

The framework makes a fundamental assumption: that the reward model and transition model depend only on the current state and current action, and not on previous history. This assumption is known as the Markov assumption (hence the name Markov Decision Process), which is a basic assumption used in reasoning about temporal models. The Markov assumption is often stated as "The future is independent of the past given the present".

The MDP framework describes how the agent interacts with the world, but we haven't yet described what the goals of the agent actually are, i.e., what problem we are trying to solve. Here are some variants of the problem:

Finite horizon We are given a number N, called the horizon, and assume that the agent is only interested in maximizing the reward accumulated in the first N steps. In a sense, N represents the degree to which the agent looks ahead. N = 1 corresponds to a greedy agent that is only trying to maximize its next reward. Formally, if the agent passes through states $s_{(0)}, s_{(1)}, \ldots, s_{(N)}$, taking actions $a_{(0)}, \ldots, a_{(N-1)}$, the agent receives utility $\sum_{t=0}^{N-1} R(s_{(t)}, a_{(t)})$.[1]

Total reward The agent is interested in maximizing its reward from now until eternity. This makes most sense if we know the agent will die at some point, but we don't have an upper bound on how long it will take for it to die. If it is possible for the agent to go on living forever, its total reward might be unbounded, so the problem of maximizing the total reward is ill-posed. Formally, if the agent passes through states $s^{(0)}, s^{(1)}, \ldots$, taking actions $a^{(0)}, a^{(1)}, \ldots$, its utility is $\sum_{t=0}^{\infty} R(s^{(t)}, a^{(t)})$.

Total discounted reward A standard way to deal with agents that might live forever is to say that they are interested in maximizing their total reward, but future rewards are discounted. We are given a discount factor γ. The reward received at time t is discounted by γ^t, so the total discounted reward is $\sum_{t=0}^{\infty} \gamma^t R(s^{(t)}, a^{(t)})$. Provided $0 \leq \gamma < 1$, this sum is guaranteed to converge.[2]

Long-run average reward Another way to deal with the problems with the total reward formulation, while still considering an agent's reward arbitrarily far into the future, is to look at the average reward per unit time over the (possibly infinite) lifespan of the agent. This is defined as $\lim_{N \to \infty} \frac{1}{N} \sum_{t=0}^{N-1} R(s^{(t)}, a^{(t)})$. While this formulation is theoretically appealing, the limit in the definition makes it mathematically intractable in many cases.

[1] Note the use of superscript in parentheses for time indexes. This is a common notation.

[2] This fact also assumes that the reward function is bounded, which is obviously true if there is only a finite set of states. Strange things start to happen with unbounded reward functions.

In this course, we will focus on the finite horizon and total discounted reward cases (which we will simply call the infinite horizon case).

In order to complete the specification of the problem, we need to ask two more questions:

1. Does the agent actually know the current state of the world when it gets to choose its action? For now, we will assume that it does (even though this assumption is not true in many real-world situations). It turns out that the problem is significantly more difficult if the agent does not know the state of the world precisely, but only gets evidence about the state.

2. Does the agent actually know the transition and reward models? We

will start out assuming that it does. However, we will soon relax this assumption, when we discuss reinforcement learning.

2 Examples

Let's look at some example MDPs. For the first example, suppose you are participating in an Internet auction. We'll use a somewhat artificial model to keep things simple. The bidding starts at $100, and in each round you get a chance to bid. If your bid is successful, you will be recorded as the current bidder of record, and the asking price will go up $100. Someone else might also bid, but only one successful bid is accepted on each round. The bidding closes if a bid of $200 is received, or no bid is received for two successive rounds. You value the item being sold at $150. If you do not buy the item, you receive 0 reward. If you buy it for $x, you receive a reward of $150 - x$.

Formally, we specify the problem as follows:

- The state space consists of tuples $< x,y,z >$, where x is the current highest bid, y specifies whether or not you are the current bidder, and z is the number of rounds since the last bid was received. The possible values for x are $0, $100, and $200. Whether or not you are the current bidder is a Boolean variable. The number of rounds since the last bid was received is 0, 1 or 2. So the total number of states is $3 * 2 * 3 = 18$.

- There are two possible actions: to bid or not to bid, denote b and $\neg$b.

- Since you only get a non-zero reward when the auction terminates and you have bought the item, the reward function is defined as follows. While the action is normally an argument to the reward function, it has no effect on the reward in this case, so we drop it.

$$R(< x,y,z >) = \begin{cases} 150 - x & \text{if } y = \text{true and } z = 2 \text{ or } x = 200 \\ 0 & \text{otherwise} \end{cases}$$

- The transition model is specified as follows. States where x = $200 or z = 2 are terminal states — the auction ends at those states, and there is

no further action. We only need to specify what happens if x < $200 and

$z < 2$. Let's say that if you do not bid, there is a 50% probability that someone else will succesfully bid. If you do bid, your bid will be successful 70% of the time, but 30% of the time, someone else will beat you to the punch. Formally, we define the transition model as follows.

$$T_{<x,y,z>} = \begin{cases} < x + 100, \text{false}, 0 > \text{ with probability } 0.5 \\ < x, y, z + 1 > \quad \text{with probability } 0.5 \end{cases}$$

$$T_{<x,y,z>} = \begin{cases} < x + 100, \text{true}, 0 > \text{ with probability } 0.7 \\ < x + 100, \text{false}, 0 > \text{ with probability } 0.3 \end{cases}$$

Another common example of an MDP is robot navigation. Suppose that your robot is navigating on a grid. It has a goal to get to, and perhaps some dangerous spots like stairwells that it needs to avoid. Unfortunately, because of slippage problems, its operators are not deterministic, and it needs to take that into account when planning a path to the goal. Here, the state space consists of possible locations of the robot, and the direction in which it is currently facing. The total number of states is 4 times the number of locations, which is quite manageable. The actions might be to move forward, turn left or turn right. The reward model gives the agent a reward if it gets to the goal, and a punishment if it falls down a stairwell. The transition model states that in most cases actions have their expected effect, e.g. moving forward will normally successfully move the robot forward one space, but with some probability the robot will stick in the same place, and with some probability it will move forward two spaces.

We can extend this example to our plant-eating robot. In addition to navigating around the planet, the robot also may eat plants. Let us assume that the robot has explored the planet, and located and analyzed all the plants. In fact, let us assume that for each plant, the robot has determined the probability that the plant is nutritious. It is at a particular location, facing a particular direction, with a particular energy level, and its goal is to survive as long as possible.

The state space needs to be much richer for this problem than for the simple navigation task. Specifically, in addition to the location and direction of the robot, the state needs to include a variable indicating how much energy the robot currently has, and a variable for each plant, indicating whether or not it has been eaten. The set of actions is expanded to include an eating action. The transition model for the eating action is specified as follows: If the robot eats plant w, and the probability that w is nutritious is p, then it gains 10 units of energy with probability p, and loses 20 energy with probability $1 - p$. The reward model is very simple. Since the goal of the robot is to stay alive as long as possible, we can simply assign a reward of 1 to every state in which the robot is alive, i.e. it has positive energy.

3 Expectimax Search

Given an MDP, the task is to find a policy that maximizes the agent's utility. A policy specifies what the agent should do at every state it might encounter. We usually notate policies by the letter π. How do we find an optimal policy? Let's concentrate on the finite horizon case first.

One approach is to view an MDP as a game against nature. I.e., it is just like a game, except that instead of having an adversarial opponent, your opponent is "nature", which behaves in a specific probabilistic manner.

Those who have taken CS182 will recall that we solve games by building a game tree. There are two kinds of nodes in the tree: those where you get to move, and those where nature moves. Nodes where you get to move are associated with the current state i; nodes where nature moves are associated with the pair <i,a>, where i is the current state and a is your chosen action. The root of the game tree is the initial state s_0. If you choose action a at node i, the game transitions to node $< i,a >$. From node $< i,a >$, the game can transition to any state j such that $T^a_{ij} > 0$. The edge from $< i,a >$ to j is annotated with T^a_{ij}. This corresponds to the fact that nature might "choose" to go from <i,a> to j, and the probability that nature will make that choice is the transition probability T^a_{ij}.

How do we solve this game? We use a simple idea: every node of the tree has a value. The value of a node is simply the expected total reward that you can expect to get, if the game begins at that node. Now we can state three simple equations that will allow us to compute the value of every node in the tree. We will define a function, called the Q function, taking a state i and an action a, whose value is the value at nature's node $< i,a >$. (The Q function will turn out to be useful for reinforcement learning.)

$$\pi^*(i) = \arg\max_a Q(i, a) \tag{1}$$

$$V(i) = Q(i, \pi^*(i)) \tag{2}$$

$$Q(i, a) = R(i, a) + \sum_j T^a_{ij} V(j) \tag{3}$$

The first equation states that at any state i, the optimal action $\pi^*(i)$ is the action a that maximizes your expected value amongst the child nodes <i,a>.[3] The second equation simply states that assuming you are playing the optimal strategy, the total value you can expect from a game beginning at state i is equal to the value of the child reached by taking the optimal action $\pi^*(i)$. The third equation simply states that the value at nature's node $< i,a >$ is the immediate reward received for taking action a in state i, plus the expectation over the values of the children j of $< i,a >$, where the expectation is taken according to the transition model T^a_i.

We can solve these equations from the leaves of the tree upwards. The algorithm for doing this is called expectimax search. The name corresponds to the

[3] The superscript $*$ is often used to denote optimality.

Page 6

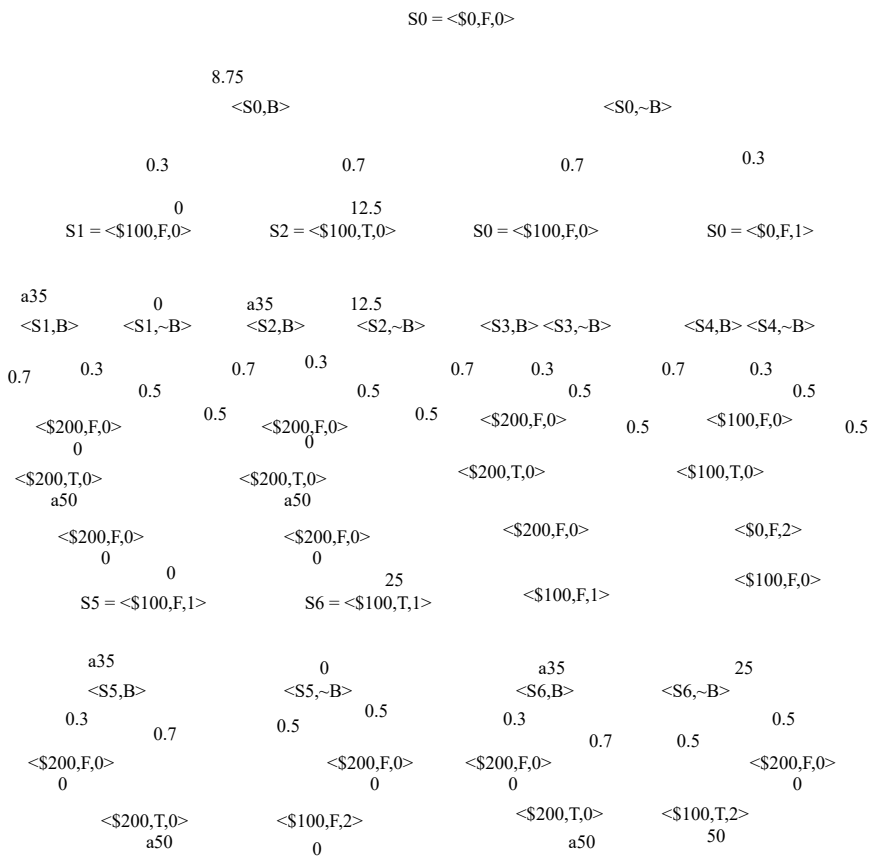

Figure 1: Expectimax search for internet auction example

fact that the algorithm alternates between taking expectations and maximizations.[4] The algorithm is as follows:

```
Expectimax(i) =
    // Takes an initial state i
    // Returns the value of i
    // Stores the optimal action at i
    If i is terminal
        Then return 0
    Else
        For each action a
            Q(i, a) = R(i, a) + ∑j Ta_ij · Expectimax(j)
        π*(i) = arg maxa Q(i, a)
        Return Q(i, π*(i)).
```

Figure 1 shows part of the expectimax tree for the internet auction example. The left subtree, corresponding to the initial action of bid, has been expanded completely and analyzed. The right subtree has only been expanded to the second level. Nature's edges are annotated with probabilities, while nodes whose values have been computed are annotated with the value. From the analysis, we see that the auction is worth at least $8.75 to the agent. The agent can use the strategy of bidding right away, hoping that the bid succeeds, and hoping that noone bids for the next two rounds. We do not know yet whether the agent can do better by passing in the first round. As it happens, the agent cannot do as well by passing. If the agent passes, it will hope that noone else bids, and then have to bid immediately in the next round, hoping its bid will be successful. It will then have to hope that noone bids for the next two rounds. This strategy has smaller chance of succeeding than bidding in the first round.

How expensive is this algorithm? It is proportional to the size of the tree, which is exponential in the horizon. What is the base of the exponent? The number of actions times the number of possible transitions for nature. To be precise, let the horizon be N, the number of actions be A, and the maximum number of transitions for any state and action be M. Then the cost of the algorithm is $O((AM)^N)$. How large is M? It depends on the model. M is the maximum number of non-zero entries in any row of the transition matrix for any action. In general, if the matrices are sparse, the branching factor will be small and the algorithm will be more efficient.[5]

There are algorithms that search the game tree cleverly in order to reduce the amount of search. An example is branch-and-bound (similar to alpha-beta pruning). But search algorithms is a topic of CS182, so we won't go into them.

[4]Those of you who took CS182 will recall that the algorithm for solving two-player games is called minimax search. The two algorithms are very similar. The difference is that in expectimax search, an expectation operator replaces the minimization operator.

[5]A tighter bound on the cost of the algorithm is $O(\sum_a M_a)^N$, where M_a is the maximum number of entries in a row in the transition matrix T_a.

4 Dynamic Programming

Instead, we'll look at another idea. Notice that the same state may be reached by many different paths. This is true even in deterministic settings, but is especially true in probabilistic settings - nature tends to act as an equalizer, negating the results of your actions, (especially actions in the distant past).

If there are states that can be reached in multiple ways, there's a tremendous amount of wasted work in the expectimax algorithm, because the entire tree beneath the repeated states is searched multiple times.

An alternative approach is as follows: let us denote by $V_k(i)$ the k-step-to-go value of state i. That is, the value that you can expect to get if you start in state i and proceed for k steps. So, $V_0(i)$ is just 0. For $k > 0$, we can compute $V_k(i)$ if we already know the values of $V_{k-1}(j)$ for all possible successor states j. We can modify the formulas above slightly, to take into account the number of steps to go.

$$V_0(i) = 0 \tag{4}$$

$$\pi_k^*(i) = \arg\max_a Q_k(i, a) \quad (5)$$

$$V_k(i) = Q_k(i, \pi^*(i)) \quad (6)$$

$$Q_k(i, a) = R(i, a) + \sum_j T_{ij}^a V_{k-1}(j) \quad (7)$$

This suggests an algorithm, where we solve this series of equations layer by layer, from the bottom up. The algorithm, called value iteration, is as follows:

```
ValueIteration(H) = // Finite horizon version
    // Takes the horizon H
    // Stores the k-step-to-go value and policy
    // for each state and each k from 1 to N
    For each state i
        V0(i)=0
    For k = 1 to H
        For each state i
            For each action a
                Qk(i, a) = R(i, a) + ∑j Ta ij · Vk−1(j)
            π*k(i) = arg maxa Qk(i, a)
            Vk(i) = Qk(i, π*k(i)).
```

What is the meaning of π^*_k and V_k? π^*_k is the optimal policy for the k-th step before the end, assuming you have only k steps to go. V_k is the value you get in the last k steps of the game, assuming you play optimally. As k grows larger, the nature of the optimal strategy changes. For k = 1, the strategy will be greedy, considering only the reward in the next step. For small k, only the next few rewards will be considered, and there will be no long-term planning.

For large k, there is incentive to sacrifice short-term gain for long run benefit, because there is plenty of time to reap the reward of your investment.

What is the complexity of finite horizon value iteration? If the number of states is n, the number of actions is m, and the horizon is N, the complexity is $O(Nmn^2)$. This is linear in the horizon, instead of exponential as we found for expectimax.

However, expectimax still has one advantage over dynamic programming: it only has to compute values for states that are actually reached. If states

can generally not be reached in multiple ways, and there are many states that are not reached at all, expectimax may be better. There is a simple algorithm that combines the advantages of both, using an idea called reachability analysis. The first stage is simply to determine which of the possible states are reachable. Then, value iteration is performed over this reduced set of states.

5 Infinite Horizon Value Iteration

So far, we've focused on the finite horizon case. What about the infinite horizon, total discounted reward case? It turns out that the value iteration idea generalizes quite easily to this case. First, we need to modify the above equations, defining the k-step-to-go rewards and policies, to take into account the discount factor.

$$V_0(i) = 0 \tag{8}$$

$$\pi^*_k(i) = \arg\max_a Q_k(i, a) \tag{9}$$

$$V_k(i) = Q_k(i, \pi^*(i)) \tag{10}$$

$$Q_k(i, a) = R(i, a) + \gamma \cdot \sum_j T^a_{ij} V_{k-1}(j) \tag{11}$$

The discussion above said that π^*_k is the optimal policy for the k-th step before the end, if you assume that the game will end after k steps. Similarly, V_k is the value obtained in k steps, by a policy maker who plays optimally for a k step game. This suggests that if we take k to infinity, then perhaps in the limit we will get a policy π^* that is optimal for the infinite horizon case, and a value V that is the total expected discounted value of each node, under the optimal policy. This idea is implemented in the infinite horizon version of value iteration:

```
ValueIteration(γ) = // Infinite horizon version
    // Takes the discount factor γ
    // Returns the optimal value and policy for each state
    For each state i
        V (i)=0
    Repeat
        For each state i
```

V (i) = V (i) // store the old value
For each state i
For each action a
$Q(i, a) = R(i, a) + \gamma \cdot \sum_j T_a$ $_{ij} \cdot V(j)$
$\pi_*(i) = \arg\max_a Q(i, a)$
$V(i) = Q(i, \pi_*(i))$
Until convergence // $\forall_i V(i) - V(i) < \epsilon$
Return V, π_*.

6 Analysis

Does the infinite horizon value iteration algorithm work? Does the value function converge? If it does converge, does it converge to the optimal value function? Intuitively the algorithm makes sense, as it takes the finite horizon case to the limit, but sometimes our intuitions break down going from finity to infinity. In order to answer these questions, we need some analysis. In the process, we will develop some useful tools for thinking about MDPs, which will eventually lead to a better algorithm for solving them.

The first concept to define is that of a stationary policy. A stationary policy is one that only depends on the current state, and not on time or history. A stationary policy π will always specify the same action $\pi(i)$ at a state i.

It is a fact that in the infinite horizon case, there is an optimal stationary policy. Intuitively this is clear. No matter what has happened in the past, when we are at a state i we need to choose the action that maximizes our expected utility from here on out. And since the horizon is infinite, we always have to look infinitely far ahead.

This is in contrast to the finite horizon case. With a finite horizon, the policy depends on the horizon. If the horizon is small, a greedy policy that optimizes immediate reward will be better. If the horizon is large, a long-term view makes more sense.

For the infinite horizon case, since we know that there exists an optimal stationary policy, we can restrict our attention to such policies. The next question to ask is, suppose we play a stationary policy π. What is the expected utility of beginning at a state i, and playing the policy π? This is called the value of i under π, and is denoted by $V_\pi(i)$.

We can answer this question by the following observation. Since π is stationary, we know that if we reach a state j after one step, the expected future reward starting from j will be $V_\pi(j)$. Furthermore, π specifies what action to take at i, so we know the distribution over the successor states j. Therefore, the expected reward from i is given by

$$V_\pi(i) = R(i, \pi(i)) + \gamma \sum_j T_{ij}^{\pi(i)} V_\pi(j) \tag{12}$$

We have an equation like this for each state i. These equations define a set of n linear equations with n variables. It can be shown that they in fact have a

unique solution.

So, where are we? We know that there is a stationary optimal policy π_*. We also know that for any stationary policy, the value function under that policy satisfies Equation 12. So we can plug in π_* to get

$$V_{\pi^*}(i) = R(i, \pi_*(i)) + \gamma \sum_j T_{ij}^{\pi^*(i)} V_{\pi^*}(j) \quad (13)$$

But in fact we know something more. We know that π_* is optimal. This implies that π_* specifies the best possible immediate action to take, given that you know you will get V_{π^*} starting at the next state. It follows that

$$\pi_*(i) = \arg\max_a [R(i, a) + \gamma \sum_j T_{ij}^a V_{\pi^*}(j)] \quad (14)$$

and therefore that

$$V_{\pi^*}(i) = \max_a [R(i, a) + \gamma \sum_j T_{ij}^a V_{\pi^*}(j)] \quad (15)$$

Equation 15 specifies a set of equations, known as the optimality equations or Bellman equations.[6] We know that a stationary optimal policy must satisfy these equations. The equations are non-linear (they include a maximization operator), so it is natural to ask whether or not they have a solution, and whether or not the solution is unique.

Theorem 6.1: For an MDP with a finite state and action space

1. The optimality equations have a unique solution.
2. The values produced by value iteration converge to the solution of the optimality equations.

A formal proof of this theorem is beyond the scope of this course. It rests on a very useful tool from real analysis called the Banach Fixed Point Theorem, also known as the Contraction Mapping Theorem. I will try to give you an intuition as to why Theorem 6.1 is true.

Equations of the form of Equation 15 are called fixpoint equations. The reason is as follows. Suppose we start with some arbitrary value function V . We can apply the right hand side of Equation 15 to V to get a new value function V . Thus the right hand side of Equation 15 defines an operator F on the space of value functions. Equation 15 says that for the optimal value function V_{π^*} ,

$$V_{\pi^*} = F(V_{\pi^*}) \quad (16)$$

We call V_{π^*} a fixpoint of the function F, because F fixes V_{π^*} in its place.

[6] Named after Richard Bellman, who did pioneering work on MDPs in the 1960s.

Now, a natural process to find a fixpoint of F is as follows. Begin with any value function V_0. Apply F to it to get V_1. Apply F again to get V_2, and so on. This is in fact exactly what value iteration does!

The questions we asked at the beginning of this section can now be refined: Does this process of iterating F converge to a limit? Is the limit unique? The answer is yes, because F has a special property: it is a contraction. That means that given any two initial value functions V_1 and V_2, F brings them closer to each other. Formally, the following property holds:

$$F(V_1) - F(V_2) \leq \gamma V_1 - V_2 \tag{17}$$

Intuitively, if I have a process that keeps bringing points closer to each other, and I iterate the process, I will eventually reach some central point. The Contraction Mapping Theorem says that under appropriate conditions, which are satisfied here, this is in fact the case: the process of iterating F has a limit, which is the unique fixpoint of F.

Incidentally, notice that γ appears in Equation 17. This is the same γ that we have used all along as the discount factor. The smaller γ is, the faster the iteration process converges. So in fact, γ has a crucial impact on the complexity of infinite horizon value iteration. For γ close to 1, convergence will be slow. In fact, if we think about it, this is not surprising. When γ is close to 1, it means that future rewards are only slightly discounted, so the optimal policy needs to take a long-term view — just like a large horizon in the finite horizon case.