ORIE 5530: Modeling Under Uncertainty

Lecture 12 (Dynamic Programming: Finite Horizon)

Professor Mark S. Squillante,
Adapted from Professor Itai Gurvich's original notes

In this lecture we embark on a short journey into dynamic optimization. In dynamic optimization (often referred to as dynamic programming) we change our decisions dynamically from one period to the next. This should be contrasted with static optimization where we make one decision, say how much money to put in a closed bank account. Many decisions in practice allow for dynamic adjustment. All airlines and many retailers adjust their prices dynamically in response to changes in the inventory of available seats on a plane or remaining inventory, etc.

12.1 Finite horizon discrete time Dynamic Programming

We start with an example. An "investment-consumption" problem. It is taken from very nice notes by Peter Glynn.

You have a finite number of periods $0, 1, \ldots, n$. At period i, you can consume some of your capital, the rest you can invest in the bank. The money that you put in the bank (that part that you do not consume) makes a return of R_{i+1} between periods i and $i+1$ where $R_1, \ldots, R_n$ are independent random variables. They are not necessarily identically distributed random variables. It could be, for example, that as you get closer to the date in which your account "expires" the returns are larger.

Thus, if you had X_i money at the end of period i and you consumed $\$C_i$, the amount you have available at period $i+1$ is

$$X_{i+1} = (X_i - C_i)R_{i+1}.$$

Of course, there is a reason we consume. We get some utility from it. Let's assume that if you consume $\$C$ you get an immediate utility that is equal to that value. You also have utility from having some money left at the end of the horizon (say, you give a gift to your kids). Let us assume that this terminal utility is just equal to the money you have remaining at the end.

You are trying to maximize your total utility over the horizon.

$$\mathbb{E}\left[\sum_{i=0}^{n-1} C_i + X_n\right]$$

What we are trying to do is **choose the optimal consumption level at each period**.

Let us define

$V_i(x) =$ maximal expected utility with i periods passed (so $n-i$ periods to go) and x available dollars.

The key is to express the value as an optimization problem where we maximize the expected *immediate reward* plus the best we can do in the future given the action we take now (and the state to which it takes us). Here this means we are going to have a recursion of the form

$$V_i(x) = \max_{0 \le c \le x} \{c + \mathbb{E}_x[V_{i+1}(X_{i+1})]\}.$$

Notice that given $X_i = x$ and we take an action c we can replace X_{i+1} with $(x-c)R_{i+1}$, so this recursion reduces here to

$$V_i(x) = \max_{0\le c\le x}\{c + \mathbb{E}[V_{i+1}((x-c)R_{i+1})]\}.$$

If we know what $V_n(x)$ is we could solve for all other periods i and dollars x *backward*. $V_n(x)$ is then the utility if we have 0 steps to go and x dollars so $V_n(x) = x$. There is no choice here. Let's now go one step backward – to step $n-1$. If we have x dollars in the beginning of step $n-1$ and we consume $c \le x$, then we get immediately a utility of c. We will have $(x-c)R_n$ in the last period at which point our utility is just the money we have and hence $(x-c)R_n$ or, in expectation, $\mathbb{E}[(x-c)R_n]$. Thus,

$$V_{n-1}(x) = \max_{0\le c\le x}\{c + \mathbb{E}[(x-c)R_n]\} = \max_{0\le c\le x}\{c + (x-c)\mathbb{E}[R_n]\}.$$

In general, in period i we obtain what we consume and then the expected utility with $i+1$ period passes (or $n-(i+1)$ to go) assuming we take the best possible actions given that we have $(x-c)R_{i+1}$ dollars left. That is,

$$V_i(x) = \max_{0\le c\le x}\{c + \mathbb{E}[V_{i+1}((x-c)R_{i+1})]\}.$$

This then gives us a solution via backward induction. In period $n-1$ we will put everything we have in investment if $\mathbb{E}[R_n] < 1$ in which case $V_{n-1}(x) = x$. Otherwise, we consume $c = 0$ and have $V_{n-1}(x) = x\mathbb{E}[R_n]$:

$$V_{n-1}(x) = \max\{x, x\mathbb{E}[R_n]\} = x\max\{1, \mathbb{E}[R_n]\}.$$

Define the notation $\gamma_n = \max\{1, \mathbb{E}[R_n]\}$. Then, $V_{n-1}(x) = \gamma_n x$. Proceeding by backward induction,

$$V_{n-2}(x) = \max_{0\le c\le x}\{c + \mathbb{E}[V_{n-1}((x-c)R_{n-1})]\} = \max_{0\le x\le c}\{c + \gamma_n\mathbb{E}[(x-c)R_{n-1}]\}.$$

By the same argument we would consume everything if $\gamma_n\mathbb{E}[R_{n-1}] < 1$ and nothing otherwise. Thus,

$$V_{n-2}(x) = \max\{x, \gamma_n\mathbb{E}[xR_{n-1}]\} = x\max\{1, \gamma_n\mathbb{E}[R_{n-1}]\}.$$

Letting $\gamma_{n-1} = \max\{1, \gamma_n\mathbb{E}[R_{n-1}]\}$ we can proceed in this way to get with $\gamma_i = \max\{1, \gamma_{i+1}\mathbb{E}[R_i]\}$ that we will consume all if $\gamma_i < 1$ and invest all otherwise.

Let's pause now to reflect on what we did and generalize this. We have the following elements:

- An expected immediate reward that depends on the action (let's call it δ). In general, it may depend also on the state. So we can write $r(\delta, x)$ for the reward if you take the action δ when in state x. In our example above $r(\delta, x) = \delta$. It is independent of x and the action is the consumption c and that is your reward.
- Constraints on acceptable actions. In the example above the control c could take only values between 0 and x, that is we require that $c \in \Delta(x) = \{y : 0 \le y \le x\}$. In general, we will write this as $\delta \in \Delta(x)$.
- Dynamics that govern the transition between the current state and the next, *given the action we take*. Above we could compute $X_{i+1} = (X_i - C_i)R_{i+1}$. This means for example that, given consumption c,

$$\mathbb{P}_c^i(x,y) = \mathbb{P}_c\{X_{i+1} = y | X_i = x\} = \begin{cases} 1 & \text{if } c = x, y = 0 \\ \mathbb{P}\{R_{i+1} = y/(x-c)\} & \text{otherwise.} \end{cases}$$

  The first row follows since if we invest all our money ($c = x$) the next state will be 0 with probability 1 (does not matter what interest the bank offers, we have no money remaining to

put there). Also, the only relevant transitions are from x to y of the form $(x-c)r$. Then, $\mathbb{E}_x[V_{i+1}(X_{i+1})] = \sum_y \mathbb{P}_c^i(x,y)V_{i+1}(y)$

Notice that here, because we allow R_i to have a different distribution per i, the transition probability depends on i.

- A recursion that says that the optimal decision in this period is to maximize the combination of what we get in this period plus the optimal utility starting at the state we move to:

$$V_i(x) = \max_{0\le c\le x}\{c + \sum_y \mathbb{P}_c^i(x,y)V_{i+1}(y)\}$$

This is the finite horizon *dynamic programming equation.*

This is clearly generalizable. Take any setting where there is

$$\text{a per-period reward function } r(A_i, X_i)$$

(that can depend on the action A_i in period i and on the state X_i). In the example we had $A_i = C_i$ as the consumption and $(X_i - C_i)$ as the investment.

We seek to choose actions $A_1, \ldots, A_n$ to maximize utility. We can change are actions dynamically – we can choose A_{n-1} after we have seen the states up to that time evolving. The utility we seek to maximize is

$$\mathbb{E}[\sum_{i=0}^{n-1} r(A_i, X_i) + u(X_n)],$$

where we allow for a terminal utility $u(X_n)$ that depends on the *terminal* state. We can write the dynamic programming equation

$$V_i(x) = \max_{\delta\in\Delta(x)} \left\{ r(\delta, x) + \sum_y \mathbb{P}_\delta^i(x,y)V_{i+1}(y) \right\}.$$

This might still be more general than needed in some cases. In the example we had, $R_1, R_2, \ldots, R_n$ were independent but did not have to be identically distributed. This is why we allow $\mathbb{P}_\delta$ to depend on the time i. If they were all identically distributed we could have $\mathbb{P}_\delta$ instead of $\mathbb{P}_\delta^i$. Also, we do not necessarily have to have a terminal value for the remaining X_n, that is, $u(X_n) = 0$ can also be the case. We have freedom within this framework.

Finally, notice that for each x in the state space and time i we are going to get from solving this equation a recommended action $\delta_i^*(x)$.

The fundamental result here is that

- This dynamic programming equation has a unique finite-valued solution.
- The action $A_i = \delta^*(X_i)$ is an optimal action at state X_i.

While the result is somewhat intuitive, there is depth to it. It says that we can find optimal actions that do not depend on all the history. We only need the current state and the time.

Example 1: Pricing inventory In this setting we have an initial inventory of x units. Think of this as, for example, the number of seats on a flight. One customer arrives per period. An arriving customer will buy the product (say, a ticket) at price p with probability $\lambda(p)$ (and not buy with probability $1-\lambda(p)$). If we had only one seat to sell we would choose the price that maximizes the expected immediate reward = price*probability of sale = $p * \lambda(p)$. If the arriving customers buys a unit of the product at the offered price then the inventory goes down by 1. Let $B_i(p)$ be a random variable that equals 1 if the i^{th} customer

buys the product when offered a price p. That is, $\mathbb{P}\{B_i(p) = 1\} = \lambda(p)$. We will assume customers are statistically identical (they all have the same probability of buying under a price p). Then, we seek to choose a price strategy (saying what price p_i to offer to the i^{th} customer given the inventory states) to maximize

$$\mathbb{E}_x \left[\sum_{i=1}^{n} p_i B_i(p_i) \right].$$

Here are the ingredients:

- **Immediate expected reward**: Under the action δ, the expected reward is given by
$$r(\delta, x) = \delta \mathbb{E}[B_i(\delta)] = \delta \lambda(\delta).$$
- **Transition Probability:** If we offer the price δ, the transition is
$$\mathbb{P}_\delta(x, x-1) = \mathbb{P}_\delta\{X_{i+1} = x-1 | X_i = x\} = \lambda(\delta),$$
and
$$\mathbb{P}_\delta(x, x) = \mathbb{P}_\delta\{X_{i+1} = x | X_i = x\} = 1 - \lambda(\delta).$$
All other transition have 0 probability.
- **Terminal value:** In the last period no customer arrives and the inventory we have remaining is useless. Thus, $V_{n+1}(x) = 0$ for all x and that serves for our recursion.

The dynamic programming equation is given by

$$V_i(x) = \max_\delta \{\delta\lambda(\delta) + \lambda(\delta)V_{i+1}(x-1) + (1-\lambda(\delta))V_{i+1}(x)\}.$$

To make things concrete, let us assume the following structure. Let us normalize the price to 1 (if you are counting in thousands of dollars, 1 will stand for one thousdand). If the price is 1 nobody buys, and everybody buys at \$0. Things are linear in between. Namely,

$$\lambda(\delta) = (1 - \delta).$$

Then, replacing $\lambda(\delta)$ with $(1-\delta)$, the equation is, **for** $x \geq 1$

$$V_i(x) = \max_\delta \{\delta(1-\delta) + (1-\delta)V_{i+1}(x-1) + \delta V_{i+1}(x)\}.$$

Notice that, given V_{i+1}, we know how to find the optimal δ by differentiating and comparing to 0. Differentiating $\delta(1-\delta) + (1-\delta)V_{i+1}(x-1) + \delta V_{i+1}(x)$ with respect to δ gives $1 - 2\delta + V_{i+1}(x) - V_{i+1}(x-1)$. Comparing to 0 gives

$$\delta^*(i, x) = \frac{1}{2}\left(1 + V_{i+1}(x) - V_{i+1}(x-1)\right).$$

There is a nice interpretation of this as an opportunity-cost correction to the base price of $1/2$. Now if we plug this back into our recursion equation we get the equation, **for** $x \geq 1$,

$$\begin{aligned} V_i(x) &= \delta^*(i,x)(1-\delta^*(i,x)) + (1-\delta^*(i,x))V_{i+1}(x-1) + \delta^*(i,x)V_{i+1}(x) \\ &= \left(\frac{1}{2}(1 + V_{i+1}(x) - V_{i+1}(x-1))\right)^2 + V_{i+1}(x-1). \end{aligned}$$

(Check the last transition yourselves.) So now we have a recursion that we can easily code starting from $V_{n+1}(x) = 0$. **Notice, also, that for** $x = 0$**,** $V_i(0) = 0$ **for all** i**.** See the worksheet in the excel file $FiniteHorizonPricing.xlsx$.

A final comment. What would you do if prices could only take on one of three values, say $1/4, 1/2$ or $3/4$. Then, we cannot solve for the optimal price via differentiation. In the code we will have to compare. Namely, when solving for $V_i(x)$, given $V_{i+1}(x)$ that we have from the backward induction, we will compare the different prices

$$V_i(x) = \max_{\delta \in \{1/4, 1/2, 3/4\}} \{\delta(1-\delta) + (1-\delta)V_{i+1}(x-1) + \delta V_{i+1}(x)\}.$$

We will compute the right hand side for each price. Do a comparison and pick the highest as $\delta^*(i, x)$.

Example 2: The multi-secretary problem. n items are presented one at a time. They have values $Z_1, Z_2, \ldots, Z_n$ that are independent and identically distributed from a discrete distribution over $\{1, 2, 3, 4\}$ with probabilities $\mathbb{P}\{Z_i = k\} = p_k$ for each value k.

Once a value is presented you have to decide whether to take it or not. You cannot go back and correct your actions. You have a finite budget $c < n$. So the total number of items you can take is c out of the n presented to you (if $c \geq n$ there would be no question to solve here — you would just take everything).

Let the remaining budget be c and the value we see in step i be a_i. At each step the policy is $\delta \in \{0, 1\}^4$. Meaning $\delta = (\delta_1, \delta_2, \delta_3, \delta_4)$. Having $\delta_1 = 1$ means that we will take the secretary if his value turns out to be 1, $\delta_2 = 1$ means we take the secretary if his value is 2. If both δ_1 and $\delta_2 = 1$ we will take the secretary if his values are either 1 or 2. You see where this is going. So the control specifies these vectors δ.

The state to track is the amount of budget we have left. The other elements of this are

- Expected immediate reward:
$$r(x, \delta) = \sum_{k=1}^{4} k p_k \delta_k$$
(why is that?)
- Transition: if $\delta_k = 1$, and the secretary turns out to be of type k, then our budget goes down by one. That is,
$$\mathbb{P}_\delta(x, x-1) = \sum_{k=1}^{4} p_k \delta_k.$$

The dynamic programming equation is given by

$$V_i(x) = \max_{\delta \in \{0,1\}^4} \left\{ \sum_{k=1}^{4} k p_k \delta_k + (\sum_{k=1}^{4} p_k \delta_k) V_{i+1}(x-1) + (1 - \sum_{k=1}^{4} p_k \delta_k) V_{i+1}(x) \right\}, \text{ for } x = 1, \ldots, n.$$

And we have the terminal condition, for all budget $c \geq 1$, $V_{n+1}(c) = 0$ (the expected value). This problem is very difficult to solve by hand but relatively easy to code. This is part of your last homework.