Dynamic Programming

Dynamic programming (DP) is the most important technique to solve many optimization problems. In most applications, dynamic programming obtains solutions by working backwards from the end of a problem to the beginning, in such a way that a large, complicated problem is broken up into a series of smaller, more tractable problems.

1 Simple examples of dynamic programming

1.1 Which is the heaviest coin

We have 21 coins and are told that all the coins are of the same weight except one is heavier than any of the other coins. How many weighings on a balance will it take to find the heaviest coin?

Analysis: The answer of the question is not so clear at first glance. However, let us think backwards. What is the maximum number of coin we should have left in order for the last weighing to successfully tell the heaviest coin. This is an easy question to answer: 3 coins. We only need to pick two of three coins and weigh them on the balance. If they are of the same weight, then the one coin left is the heaviest coin. If they are not, the heavier one is the heaviest coin in the bunch. However, if we have 4 or more coins left, we cannot always tell which one is the heaviest if only one weighing is left.

Now let us go back one step and ask what is the maximum number of coin we should have left in order for the last two weighing to successfully tell the heaviest coin. Suppose we have $m \leq 9$ coins left. What we can do is to divide the coin into three groups $(a, a, b)$ such that $a + a + b = m$, and $a \leq 3, b \leq 3$. Then we weight the first two groups on the balance. If they have the same weight, then the heaviest coin is in the group of $b$ coins. If they are not, then the heaviest coin is in the heavier group of $a$ coins. Either way, since $a \leq 3, b \leq 3$, we can always tell the heaviest coin in the last weighing. Now assume that we have $m \geq 10$ coins, and we divide into three groups $(a, a, b)$. A weighing on balance will tell which group the heaviest coin belongs to. No matter how we do it, there is at least a group with at least 4 coins. If it turns out that the heavier coin is in that group, we are in trouble, because the last weighing cannot tell the heaviest coin. Therefore, we can tell the heaviest coin from at most 9 coins, and cannot not tell if there are at least 10 coins.

Go on step further, it is easy to see that if we have at most $3 \cdot 9 = 27$ coins, we can tell the heaviest one in three weighings. Therefore, for 21 coins, we need at most 3 weighings. Not only does the above procedure gives the number of weighings need, it also yields how should the weighings go: We will divide the coins into three groups $(a, a, b)$ such that $a \leq 9, b \leq 9$. Say we take $a = b = 7$. The first weighing will tell us the group the heaviest coin belongs to. Then we divide that group (of 7 coins) into three groups $(2, 2, 3)$ or $(3, 3, 1)$. The second weighing will tell which group the heaviest coin belong to, and the last weighing will only need to tell the heaviest coin from at most 3 coins.
**Remark:** The above procedure indeed solves a more general problem: suppose we have \( n \geq 2 \) coins to start with, then the number of weighings needed to tell the heaviest coin is \( k \) with \( 3^{k-1} < n < 3^k \). In other words, \( k \) weighings will tell the heaviest coin from at most \( 3^k \) coins. A mathematical description for the above process is as follows. Let

\[
v_k = \max \{ n : \text{the heaviest coin can be distinguished from at most } k \text{ weighings} \}.
\]

We start with \( v_1 = 3 \) and we have \( v_{k+1} = 3v_k \). Therefore, \( v_k = 3^k \).

The analysis also tells how to weigh these \( n \) coins in \( k \) weighting: each time divide the coins into three groups \((a, a, b)\) such that \( a \leq 3^{k-1}, \ b \leq 3^{k-1} \) if there are \( l \) weighings left.

**Remark:** The coin problem can be viewed as an optimization problem: minimize the number of weighings to tell the heaviest coin. The above analysis, though simple, gives some basic characteristics of the dynamic programming: not only does DP gives a systematic way to solve an optimization problem, it also tells you more information on the optimal policy. In the coin problem, the policy corresponds to the process of weighing.

1.2 **Shortest path problem**

The following example is a simple shortest path problem, which we will discuss later in great generality. Consider the following map, and one wants to find the shortest path from NY to LA.

\[
\text{(1)} \quad \text{Cleveland} \quad \text{1800} \quad \text{Phoenix} \quad \text{400} \\
\text{(2)} \quad \text{St. Louis} \quad \text{600} \quad \text{Dallas} \quad \text{1500} \quad \text{LA} \\
\text{(3)} \quad \text{Nashville} \quad \text{1200} \quad \text{Salt Lake City} \quad \text{600} \\
\text{NY} \quad 950 \quad \text{900} \\
\]

This shortest path problem has a very distinctive feature in that the problem can be divided into several stages.

- Stage 0 = \{NY\}
- Stage 1 = \{Cleveland, St. Louis, Nashville\}
- Stage 2 = \{Phoenix, Dallas, Salt Lake City\}
- Stage 3 = \{LA\}

Each stage consists of one or several cities, and one always moves from a city in one stage to a city in the next stage. A direct approach to the problem is to enumerate all the possible paths and choose the one with the minimal distance. Such an approach will work in such a small-scale example. But imagine doing this for a complicated map!

The idea here is again working backwards. Let

\[
V_k(x) = \text{shortest path to LA when you are at city } x \text{ of stage } k.
\]
Clearly, \( V_3(\text{LA}) = 0 \)

\[
V_2(\text{Phoenix}) = 400, \quad V_2(\text{Dallas}) = 1300, \quad V_2(\text{Salt Lake City}) = 600.
\]

And it is easy to see that

\[
V_1(\text{Cleveland}) = \min \{ \text{dist}(\text{Cleveland, Phoenix}) + V_2(\text{Phoenix}), \text{dist}(\text{Cleveland, Dallas}) + V_2(\text{Dallas}) \} = 1800 + 400 = 2200,
\]

\[
V_1(\text{St. Louis}) = \min \{ \text{dist}(\text{St. Louis, Phoenix}) + V_2(\text{Phoenix}), \text{dist}(\text{St. Louis, Dallas}) + V_2(\text{Dallas}) \} = 1100 + 400 + 600 + 1300 = 1500.
\]

\[
V_1(\text{Nashville}) = \min \{ \text{dist}(\text{Nashville, Dallas}) + V_2(\text{Dallas}), \\
\text{dist}(\text{Nashville, Salt Lake City}) + V_2(\text{Salt Lake City}) \} = \min \{ 600 + 1300, 1200 + 600 \} = 1800.
\]

It is easy now to have

\[
V_0(\text{NY}) = \min \{ \text{dist}(\text{NY, Cleveland}) + V_1(\text{Cleveland}), \text{dist}(\text{NY, St. Louis}) + V_1(\text{St. Louis}), \\
\text{dist}(\text{NY, Nashville}) + V_1(\text{Nashville}) \} = \min \{ 400 + 2200, 950 + 1500, 600 + 1500 \} = 2450.
\]

As before, the DP also tells us the shortest path is “NY→St. Louis→Salt Lake City→LA”.

**Remark:** The above iterations can be written in a more compact form.

\[
V_k(x) = \min \{ \text{dist}(x, y) + V_{k+1}(y) : y \in \text{Stage}(k+1) \},
\]

with the convention that \( d(x, y) = \infty \) if there is no path from \( x \) to \( y \). This is called the **dynamic programming equation** (DPE).

### 1.3 American option pricing

The next example is an American option pricing problem and can be viewed as a simple example of **Decision Analysis**. A bit of intuitive probabilities is involved. Even though the notation becomes more messy, the basic idea of dynamic programming remains same and simple.

Suppose you own a share of American put option on IBM with strike price 60 dollars. In other words, if you liquidate the option when the IBM stock price is \( S \), you get a payoff \( (60 - S)^+ = \max\{0, 60 - S\} \). Everyday the stock price has probability 0.6 to go up 10 dollars and probability 0.4 to go south 10 dollars. Figure 1 is a tree for the stock price. The option can be liquidated at any time \( t = 0, 1, \) or \( t = 2 \). You want to maximize your average payoff from the liquidation. Question: should you liquidate the option now, or should you wait? What is the maximal average payoff you can achieve?
Solution: The solution is not at all clear. If you liquidate now, you will get \((60 - 50)^+ = 10\) dollars. If you choose to wait, it is hard to tell that whether you could get more or less.

Now let us work backwards. Suppose we are at \(t = 2\), what should we do? This is obvious since we will liquidate if the price is below the strike to get a payoff \((60 - S)^+\), and the option is toilet paper is the stock price is above the strike. The payoff is calculated in Figure 2.

Go back one step, say we are at \(t = 1\), and the stock price become 60. The liquidation of the option will yield a payoff of 0. If we choose to wait, we will get an average payoff \(0.6 \cdot 10 + 0.4 \cdot 10 = 10\). Therefore, if we are at \(t = 1\) and the stock price is 60, we should wait and expect an average payoff

\[
\max\{(60 - 60)^+, 0.6 \cdot 10 + 0.4 \cdot 10\} = \max\{0, 4\} - 4.
\]

What if at \(t = 1\) the stock price is 40 dollars. If you liquidate, you will get a payoff 20 dollars. If not, you will on average get a payoff \(0.6 \cdot 10 + 0.4 \cdot 30 = 18\) dollars. Therefore, if we are at \(t = 1\) and the stock price is 40, we should immediately liquidate and get a payoff

\[
\max\{(60 - 40)^+, 0.6 \cdot 10 + 0.4 \cdot 30\} = \max\{20, 18\} = 20.
\]

Go back one more step, and we are \(t = 0\). If we liquidate, we get a payoff 10 dollars. But if we wait, we will expect to get a payoff \(0.6 \cdot 4 + 0.4 \cdot 20 = 10.4\). Therefore at \(t = 0\), we should wait and expect an average payoff

\[
\max\{(60 - 50)^+, 0.6 \cdot 4 + 0.4 \cdot 20\} = \max\{10, 10.4\} = 10.4
\]

At \(t = 1\), if the stock price is 60 dollars, keep waiting; if the stock is 40 dollars, liquidate.

Remark: Again, the DP works backwards, and tell us the information on the optimal strategy.

Remark: The above DP procedure can be written in a more compact form. Let

\[
V_k(x) = \text{The maximal average payoff if you are at } t = k \text{ and the stock price is } x.
\]

We are interested in \(V_0(x)\) at \(x = 50\). However, we have \(V_2(x) = (60 - x)^+\), and

\[
V_k(x) = \max \left\{ (60 - x)^+, p_u V_{k+1}(x_u) + p_d V_{k+1}(x_d) \right\}
\]

where \(x_u = x + 10\), \(x_d = x - 10\), \(p_u - 0.6\), \(p_d - 0.4\). This equation is again said to be the dynamic programming equation (DPE). The first term in the maximization corresponds to the payoff if you liquidate at \(t = k\), while the second term corresponds to the average payoff if you choose to wait. Note that at any time, we have two strategies, either liquidate or wait.

2 A general framework of deterministic dynamic programming

Consider a system whose state is determined by

\[
x_{n+1} = f_n(x_n, u_n), \quad n = 0, 1, \ldots, N.
\]

The parameters in the above equation has interpretation as

\[
n = \text{the label for the stage (usually it is naturally defined)},
\]

\[
x_n = \text{the state at stage } n
\]

\[
u_n = \text{the control (or, decision) variable at stage } n.
\]
In other words, suppose at stage \( n \), the system is in state \( x_n \). If a decision \( u_n \) is applied, the system will move to state \( s_{n+1} \) at stage \( n+1 \).

In the previous example of shortest path, the stage is artificially defined, and the state is the city, and the decision variable is which city to travel to. In the example of American option pricing, the stage is naturally defined as the time, and the state is the price of the IBM stock, and the decision variable is binary “liquidate” or “wait”.

Let us consider the following cost criteria. The goal is to select a decision (control) sequence \( U = \{u_0, u_1, \ldots, u_N\} \) so as to

\[
g(x_{N+1}) + \sum_{n=0}^{N} c_n(x_n, u_n)
\]

We can view

\[
g(x) = \begin{cases} 
\text{terminal cost} & \text{if the terminal state } x_{N+1} \text{ is } x \\
\text{running cost} & \text{if at stage } n, \text{ a decision } u_n = u \text{ is made at state } x_n = x.
\end{cases}
\]

In the example of shortest path, the running cost \( c_n(x, u) \) is nothing but the distance from \( x \) to destination determined by the control \( u \). There is no terminal cost involved.

**Remark:** For American option pricing, the situation is a bit subtle since if a decision of “liquidation” is made at any stage, the stock price (the state) after that stage does not matter a bit. Therefore the problem cannot directly fit into the cost structure we mentioned above. However, the idea to solve the problem using DPE (which we will describe below) remains the same. We will come back to this later in the section of Probabilistic Dynamic Programming.

**Remark:** Sometimes the above cost criteria is called additive. Of course one can consider a multiplicative (or even mixed) cost criteria. A general multiplicative cost reads

\[
g(x_{N+1}) \cdot \prod_{n=0}^{N} c_n(x_n, u_n).
\]

The analysis of this cost is exactly the same as that of the additive cost criteria.

The **value function** is defined as

\[
V_0(x) = \min_U \left[ g(x_{N+1}) + \sum_{n=0}^{N} c_n(x_n, u_n) \right] \quad \text{given } x_0 = x.
\]

To solve for \( V_0 \), we will expand the problem and work backwards. For notation convenience, we will denote by

\[
V_j(x) = \min_U \left[ g(x_{N+1}) + \sum_{n=j}^{N} c_n(x_n, u_n) \right] \quad \text{given } x_j = x,
\]

for \( j = 1, 2, \ldots, N + 1 \), with convention that \( \sum_{N+1}^{N} 0 \).
Dynamic Programming Equation (DPE)

It is not difficult to see that $V_{N+1}(x) \equiv g(x)$ for any $x$ by definition. Furthermore, it is intuitive that the functions $V_j$ should satisfy the following equation

\[
V_j(x) = \min_u \left[ c_j(x, u) + V_{j+1}(f_j(x, u)) \right], \quad \text{for every } x.
\]

This equation is called the dynamic programming equation (DPE). Knowing $V_{N+1}$, one can recursively solve all $V_j$. The interpretation of DPE is also clear. Suppose at stage $j$, the system is at state $x_j = x$. Choosing a control $u_j = u$, at stage $j + 1$, the state will become

\[
x_{j+1} = f_j(x_j, u_j) = f_j(x, u).
\]

Therefore the DPE says "The minimum of the cost from stage $j$ to the end of the problem must be attained by choosing at stage $j$ a decision that minimize the sum of the costs incurred during the current stage plus the minimum cost that can be incurred from stage $j + 1$ to the end of the problem." We give a short (not so rigorous) proof below.

**Proof:** Given $x_j = x$, fix an arbitrary control $u_j = u$. For any control $U = \{u_{j+1}, \ldots, u_N\}$ afterwards, by definition,

\[
V_j(x) \leq c_j(x, u) + \sum_{n=j+1}^{N} c_n(x_n, u_n) + g(x_{N+1}).
\]

Now minimize the right-hand-side over $U = \{u_{j+1}, \ldots, u_N\}$, we have

\[
V_j(x) \leq c_j(x, u) + V_{j+1}(f_j(x, u)).
\]

But remember that $u$ is arbitrary, we have

\[
V_j(x) \leq \min_u \left[ c_j(x, u) + V_{j+1}(f_j(x, u)) \right].
\]

On the other hand, if we use an optimal control $u_j^* = u^*$ at stage $j$ and afterwards optimal control $U^* = \{u_{j+1}^*, \ldots, u_N^*\}$, then we have

\[
V_j(x) = c_j(x, u^*) + \sum_{n=j+1}^{N} c_n(x_n, u_n^*) + g(x_{N+1}) = c_j(x, u^*) + V_{j+1}(f_j(x, u^*))
\]

\[
\geq \min_u \left[ c_j(x, u) + V_{j+1}(f_j(x, u)) \right].
\]

These two inequalities yield the (DPE). Furthermore, from the proof we verify that "the optimal control at stage $j$ is the control $u^*$ that achieving the minimum in the RHS of the DPE."

**Remark:** Note that the DPE actually solve $V_0(x)$ for any $x$. In practice $x_0 = x$ is a specific value, but which is not important since one only need to plug this specific initial state into function $V_0$ to obtain the value of optimization problem associated with this specific $x_0$. 


Remark: It is not hard to believe that for the multiplicative cost criteria, one can similarly define $V_j$ (with convention $\prod_{N+1} = 1$), and the DPE will become

\[
V_j(x) = \min_u \left[ r_j(x, u) \cdot V_{j+1}(f_j(x, u)) \right], \quad \text{for every } x,
\]

with $V_{N+1}(x) \equiv g(x)$. Again, the optimal control at stage $j$ is the minimizing $u^*$ in the RHS of the DPE.

Remark: The above procedure remains true in a maximization problem – just replace all the “min” above by “max”.

2.1 Examples of DP

Example: The owner of a lake must decide how many bass to catch and sell at the beginning of each year. Assume that the market demand for bass is unlimited. If $x$ is the bass sold in year $n$, a revenue of $r(x)$ is earned. The cost to catch $x$ bass is $c(x, b)$, where $b$ is the number of bass in the lake at the beginning of the year. The number of bass in the lake at the beginning of a year is 20% greater than the number in the lake at the end of the previous year. The owner is interested in maximizing the overall profit over the next $N$ years.

**DP Formulation:** In this case, the stage is naturally defined as the time $n$, which varies from 1 to $N$. The state variable is the number of bass at the beginning of year $n$, denoted by $s_n$. The decision variable at year $n$ (control) is the number of bass caught, denoted by $x_n$.

The dynamic for the system is

\[
x_{n+1} = 1.2(x_n - s_n), \quad n = 1, 2, \ldots, N - 1.
\]

The objective is to maximize the overall profit

\[
v(s) \equiv \max_{\{x_n\}} \sum_{n=1}^{N} [r(x_n) - c(x_n, s_n)], \quad \text{given } s_1 = s.
\]

The dynamic programming will work backwards: define

\[
v_j(s) \equiv \max_{\{x_n\}} \sum_{n=j}^{N} [r(x_n) - c(x_n, s_n)], \quad \text{given } s_j = s,
\]

for $j = 1, \ldots, N$, and define $v_{N+1}(s) = 0$ for any $s$ (note the terminal cost in the optimization problem is 0).

Clearly the quantity of interest is $v \equiv v_1$. The dynamic programming equation (DPE) can be written as

\[
v_j(s) = \max_{0 \leq x \leq s} [r(x) - c(x, s) + v_{j+1}(1.2(s - x))], \quad j = 1, 2, \ldots, N.
\]

Since $v_{N+1} \equiv 0$, one can work backwards to solve for all $v_j$. At year $j$, the optimal amount of bass to sell $x^*_j$ is the maximizing $x$ in the above DPE, given that $s_j = s$ (i.e., at the beginning of year $j$, the number of bass in the lake is $s$). \hfill \Box
Remark: A weakness of the previous formulation is that the profits received during later years are weighted the same as profits received during earlier years. Now we consider the following discounting factor $0 < \beta < 1$: $\$1$ received in year $i + 1$ is equivalent to $\beta$ dollar received in year $j$. Then the optimization becomes (abusing the notation a bit)

$$ v(s) \triangleq \max_{\{x_n\}} \sum_{n=1}^{N} \beta^{n-1} [r(x_n) - c(x_n, s_n)], \quad \text{given } s_1 = s. $$

Similarly we define

$$ v_j(s) \triangleq \max_{\{x_n\}} \sum_{n=j}^{N} \beta^{n-j} [r(x_n) - c(x_n, s_n)], \quad \text{given } s_j = s. $$

$v_j$ can be interpreted as the optimal profit from year $j$ to year $N$ (valued by the dollar in year $j$). Clearly $v_1 \equiv v$. The DPE becomes

$$ v_j(s) = \max_{0 \leq x \leq s} [r(x) - c(x, s) + \beta v_{j+1}(1.2(s - x))]. $$

and $v_{N+1}(s) \equiv 0$. $\Box$

Example: Farmer Jones now possesses $5000$ and $10$ tons of wheat. During month $j$, the price of wheat is $p_j$ (assumed known). During each month, he must decide how much wheat to buy or to sell. There are three restrictions on each month's wheat transaction: (1) During any month, the amount of money spent on wheat cannot exceed the cash on hand at the beginning of the month; (2) during any month, he cannot sell more wheat than he has at the beginning of the month; (3) because of limited warehouse capacity, the ending inventory of wheat for each month cannot exceed 10 ton. Show how dynamic programming can be utilized to maximize the amount of cash farmer Jones has on hand at the end of three months.

Formulation: Again, time is the stage. At the beginning of month $n$ (the present is the beginning of month 1), farmer Jones must decide how much wheat to buy or sell. We will denote by $u_n$ the change (i.e., the control) in Jones' wheat position during month $n$: $u_n \geq 0$ corresponds to a month of wheat purchase, and $u_n \leq 0$ to a month of wheat sale. The state at the beginning of month $n$ is the amount of wheat on hand, denoted by $w_n$, and the cash on hand, denoted by $c_n$. To ease notation, write $s_n = (w_n, c_n)$. The optimization is to maximize the cash gain during the 3 month period.

$$ v(s) = \max_{\{u_n\}} \sum_{n=1}^{3} -p_n u_n, \quad \text{given } s_1 = s, $$

under the constraints (1)-(3).

As before, define

$$ v_j(s) = \max_{\{u_n\}} \sum_{n=j}^{3} -p_n u_n, \quad \text{given } s_j = s, $$

with $v_3(s) \equiv 0$. Note that $v \equiv v$. The DPE takes the form:

$$ v_j(s) = \max_{-w \leq u \leq \min(10 - w, c/p_j)} [-p_j u + v_{j+1}(s)], \quad \text{for all } s = (w, c); $$

8
here \( \bar{s} = (\bar{w}, \bar{c}) \) with 
\[
\bar{w} = w + u, \quad \bar{c} = c - p_j u.
\]

That the control \( u \) is constrained in the interval \(-w \leq u \leq \min\{10 - w, c/p_j\}\) is due to the constraints: (1) cannot buy more than the cash on hand gives \( u \leq c/p_j \); (2) cannot sell more wheat than he has gives \( u \geq -w \); (2) cannot store more than 10 tons of wheat gives \( u + w \leq 10 \) or \( u \leq 10 - w \).

The DPE can recursively determine all \( v_j \). For example,
\[
v_3(s) = \max_{-w \leq u \leq \min\{10 - w, c/p_j\}} \{-p_3 u + v_4(\bar{s})\} = p_3 w, \quad \text{with maximizing } u^* = -w.
\]
That is in month 3 Jones should sell all the wheat. \( v_2 \) and \( v_1 \) can also be determined in the same manner.

A special case is that \( p_1 \geq p_2 \geq p_3 \), in which case it is obviously optimal to sell all the wheat in the beginning of month 1. This is verified by the DPE. Under this circumstance, we have
\[
v_2(s) = \max_{-w \leq u \leq \min\{10 - w, c/p_2\}} \{-p_2 u + v_3(s)\} = \max_{-w \leq u \leq \min\{10 - w, c/p_2\}} \{-p_2 u + p_3(w + u)\} = p_2 w,
\]
with maximizing \( u^* = -w \); and
\[
v_1(s) = \max_{-w \leq u \leq \min\{10 - w, c/p_1\}} \{-p_1 u + v_2(s)\} = \max_{-w \leq u \leq \min\{10 - w, c/p_1\}} \{-p_1 u + p_2(w + u)\} = p_1 w,
\]
with maximizing \( u^* = -w \). For the specific initial state \( s_1 = s_0 = (10, 5000) \), the optimal cash gain is \( v_1(s) = p_1 w - 10p_1 \), and Jones should sell the wheat immediately. His overall cash on hand at the end of month 3 is 5000 + 10p_1.

Another special case is \( p_1 \leq p_2 \leq p_3 \). The obviously optimal policy is to buy as much wheat as the cash and warehouse capacity will allow, and sell them in month 3. This is also confirmed by the DPE. In this case,
\[
v_2(s) = \max_{-w \leq u \leq \min\{10 - w, c/p_2\}} \{-p_2 u + v_3(\bar{s})\} = \max_{-w \leq u \leq \min\{10 - w, c/p_2\}} \{-p_2 u + p_3(w + u)\}
\]
\[
= p_3 w + (p_3 - p_2) \min\{10 - w, c/p_2\};
\]
with maximizing \( u^* = \min\{10 - w, c/p_2\} \), and
\[
v_1(s) = \max_{-w \leq u \leq \min\{10 - w, c/p_1\}} \{-p_1 u + v_2(s)\}
\]
\[
= \max_{-w \leq u \leq \min\{10 - w, c/p_1\}} \{-p_1 u + p_3(w + u) + (p_3 - p_2) \min\{10 - w - u, c - p_1 u\}/p_2\}
\]
\[
\text{However, the term to be maximized equals}
\]
\[
F(u) = -p_1 u + p_3(w + u) + (p_3 - p_2) \min\{10 - w - u, c - p_1 u\}/p_2 + p_3 w + p_3 c + p_3 p_2 + p_3 p_1 - p_3 p_2 - p_3 p_1,
\]
\[
\text{which is an increasing function of } u. \text{ Therefore,}
\]
\[
v_1(s) = p_3 w + \min\{10 - w\}(p_3 - p_2) + (p_2 - p_1)u^*, \quad c(p_3 - p_2)/p_2 + u^*(p_2 - p_1)p_3/p_2
\]
\[
\text{with } u^* = \min\{10 - w, c/p_1\}.
\]

\( \square \)
Exercise: Can you formulate an LP to solve this maximization problem?

Example: It’s the last weekend of the 1996 campaign, and candidate Blaa Blaa is in NY. Before election day, he must visit Miami, Dallas, and Chicago and then return to his NY headquarters. Blaa Blaa wants to minimize the total distance he must travel. In what order he should visit the cities?

<table>
<thead>
<tr>
<th></th>
<th>NY</th>
<th>Miami</th>
<th>Dallas</th>
<th>Chicago</th>
</tr>
</thead>
<tbody>
<tr>
<td>NY</td>
<td></td>
<td>1334</td>
<td>1559</td>
<td>809</td>
</tr>
<tr>
<td>Miami</td>
<td>1334</td>
<td></td>
<td>1343</td>
<td>1397</td>
</tr>
<tr>
<td>Dallas</td>
<td>1559</td>
<td>1343</td>
<td></td>
<td>921</td>
</tr>
<tr>
<td>Chicago</td>
<td>809</td>
<td>1397</td>
<td>921</td>
<td></td>
</tr>
</tbody>
</table>

Solution: It is not obvious how to work this problem into the structure of dynamic programming. The stage $j$ is easily defined as the $j$-th stop of the trip, with $j = 0$ as the starting point and $j = 4$ as the ending point (both are NY in this case). The definition of the state, however, is a bit more subtle: The state is $(I, S)$ where $I = \{\text{last city visited}\}$ and $S = \{\text{cities visited}\}$.

Let us define

\[
f_j(I, S) = \text{the minimal distance that must be travelled to complete the tour if Blaa is at the } j\text{-th stop with } I \text{ being the last city visited and } S \text{ being all the } j \text{ cities visited.}
\]

To ease notation, we will denote by $\{N, M, D, C\}$ the four cities, and the distance between city $I$ and city $J$ is denoted by $d_{IJ}$. The DPE for this problem can be written as

\[
f_j(I, S) = \min_{J \in S} [d_{IJ} + f_{j+1}(J, S \cup \{J\})], \quad \forall 0 \leq j \leq 2.
\]

with

\[
f_3(I, \{M, D, C\}) = d_{IN} = \text{distance from city } I \text{ to NY}.
\]

Recursively, we have

\[
f_2(I, S) = \min_{J \in S} [d_{IJ} + f_3(J, S \cup \{J\})] = \min_{J \in S} [d_{IJ} + d_{JN}];
\]

or

\[
\begin{align*}
f_2(M, \{M, D\}) &= d_{MC} + d_{CN} = 1397 + 809 = 2206; \\
f_2(D, \{M, D\}) &= d_{DC} + d_{CN} = 921 + 809 = 1730; \\
f_2(M, \{M, C\}) &= d_{MD} + d_{DN} = 1343 + 1559 = 2902; \\
f_2(C, \{M, C\}) &= d_{CD} + d_{DN} = 921 + 1559 = 2480; \\
f_2(D, \{D, C\}) &= d_{DM} + d_{MN} = 1343 + 1334 = 2677; \\
f_2(C, \{D, C\}) &= d_{CM} + d_{MN} = 1397 + 1334 = 2731;
\end{align*}
\]
and

\[ f_1(I, S) = \min_{J \in S} [d_{IJ} + f_2(J, S \cup \{J\})] \]

or

\[
f_1(M, \{M\}) = \min_{J \in \{D, C\}} [d_{MJ} + f_2(J, \{M, J\})]
= \min\{d_{MD} + f_2(D, \{M, D\}), d_{MC} + f_2(C, \{M, C\})\}
= \min\{1343 + 1730, 1397 + 2480\}
= 3073 \quad \text{(with the minimum achieved at } J^* = D) \]

\[
f_1(D, \{D\}) = \min_{J \in \{M, C\}} [d_{DJ} + f_2(J, \{D, J\})]
= \min\{d_{DM} + f_2(M, \{D, M\}), d_{DC} + f_2(C, \{D, C\})\}
= \min\{1343 + 2206, 921 + 2731\}
= 3549 \quad \text{(with the minimum achieved at } J^* = M) \]

\[
f_1(C, \{C\}) = \min_{J \in \{M, D\}} [d_{CJ} + f_2(J, \{C, J\})]
= \min\{d_{CM} + f_2(M, \{C, M\}), d_{CD} + f_2(D, \{C, D\})\}
= \min\{1397 + 2902, 921 + 2677\}
= 3598 \quad \text{(with the minimum achieved at } J^* = D) \]

and finally

\[
f_0(N, \{N\}) = \min_{J \in \{M, D, C\}} [d_{NJ} + f_1(J, \{J\})]
= \min\{d_{NM} + f_1(M, \{M\}), d_{ND} + f_1(D, \{D\}), d_{NC} + f_1(C, \{C\})\}
= \min\{1334 + 3073, 1559 + 3549, 809 + 3598\}
= 4407 \quad \text{(with the minimum achieved at } J^* = M \text{ or } J^* = C) \]

Therefore, the shortest tour will be either

\[ N \rightarrow M \rightarrow D \rightarrow C \rightarrow N \quad \text{or} \quad N \rightarrow C \rightarrow D \rightarrow M \rightarrow N. \]

Both has total distance 4407 miles. Note these two tours are reverse to each other. \qed
2.2 Shortest path problems

To consider a shortest path problem in general, we need to introduce some new concepts.

Definition: A graph, or network, is defined by two sets of symbols: nodes and arcs. An arc consists of a pair of nodes, and represents a possible direction of motion between the two nodes. The length of an arc from node $i$ to node $j$ is denoted by $d_{ij}$. In general, $d_{ij}$ could be negative. Note that the length of an arc does not necessarily mean the physical length, it could stand for some very general quantity associated with the arc, say for example, cost.

Definition: An arc is said to be directed if it is only allowed in one direction. Usually the direction is indicated by an arrowhead at the end of the arc. An arc is said to be undirected if both directions are allowed. In this case, there is no arrowhead on the arc. An undirected arc can be equivalently represented by two directed arcs.

Definition: A path is a sequence of arcs such that the terminal node of each arc is identical to the initial node of the next arc.

Below we will consider the shortest path problem for a network. Suppose the nodes of a network are denoted by $\{1, 2, \cdots, N\}$, and one wants to find a path from node $1$ to node $N$ with the shortest total length.

2.2.1 Shortest path for simple networks

As we have seen in Section 1.2, we can use dynamic programming to solve the shortest path problem if the nodes can be divided into groups, which we call "stage", and one always travels from a node in one stage to a node in the next stage.

Assume node 1 is in stage 0 and node $N$ is in stage $K$. The minimal distance from node $1$ to node $N$ can be recursively determined by the DPE that for any node $x$ in stage $k$,

$$V_k(x) = \min \{d_{xy} + V_{k+1}(y) : y \in \text{Stage}(k+1)\}; \quad k = K - 1, K - 2, \cdots, 1.$$ 

and $V_K(N) = 0$. With convention $d_{xy} = \infty$ if there is no arc from $x$ to $y$.

Exercise: Consider the following network. Find the shortest path from node 1 to node 10. Also find the shortest path from node 3 to node 10.

![Diagram of network](image_url)

Solution: There are three shortest paths: $1 \to 3 \to 5 \to 8 \to 10$, $1 \to 4 \to 6 \to 9 \to 10$, and $1 \to 4 \to 5 \to 8 \to 10$. Each of these path has total length 11. The shortest path from node 3 to node 10 is $3 \to 5 \to 8 \to 10$ with total length 7.
2.2.2 Shortest path by Dijkstra’s algorithm

An assumption here is the length of each arc is non-negative. Even though the Dijkstra’s algorithm is implicitly defined by a Dijkstra, it is not necessary to specify the stages.

The idea is as follows. Instead of searching the shortest path from node 1 to node N, we search the shortest path from node 1 to node n for any node $n \in \{1, 2, \cdots, N\}$. Define

$$v(j) = \text{the shortest distance from node 1 to the } j\text{-th nearest node},$$

and with convention, the 1st nearest node is node 1 itself. Suppose $N$ is the $k$th nearest node to node 1, then the shortest path from node 1 to node $N$ is $v(k)$. The key observation is that

The shortest path from node 1 to the $(j + 1)$-th nearest node only passes through nodes contained in the $j$ nearest nodes to node 1. In other words, there exists an $1 \leq i \leq j$ such that the shortest path from node 1 to the $(j + 1)$-th nearest node will consist of a shortest path to the $i$-th nearest node and an arc connecting the $i$-th nearest node and the $(j + 1)$-th nearest node.

Let $m_i$ denote the $i$-th nearest node to node 1 and $A_j = \{m_1, \cdots, m_j\}$. The equation to determine the $(j + 1)$-th nearest node and $v(j + 1)$ is

$$v(j + 1) = \min_{1 \leq i \leq j} \min_{m \notin A_j} [v(i) + d_{m,n}] = \min_{1 \leq i \leq j} \left[ v(i) + \min_{m \notin A_j} d_{m,n} \right].$$

with convention $d_{mn} = \infty$ if no arc from node $m$ to node $n$. The minimizing node $n^*$ in the RHS will be the $(j + 1)$-th nearest node to node 1. The shortest path to from node 1 to node $n^*$ is the shortest path to $m_{n^*}$ (the minimizing node $m_{n^*}$) and then the arc from $m_{n^*}$ to $n^*$.

We will illustrate the algorithm by the following examples. Figure 1 is a undirected network, i.e. every arc is undirected. The arcs in Figure 2 is mixed. In both network, the goal is to determine the shortest path from node 1 to node 8.

\[\text{Figure 1.}\]

\[\text{Figure 2.}\]

For Figure 1, there are two shortest path $1 \to 3 \to 6 \to 7 \to 8$ and $1 \to 4 \to 6 \to 7 \to 8$. Each has total length 17. For Figure 2, there are also two shortest path $1 \to 4 \to 6 \to 8$ and $1 \to 2 \to 5 \to 8$, each has total length 20.
Figure 1.

<table>
<thead>
<tr>
<th>$n$</th>
<th>Solved Nodes</th>
<th>Closest Connected Unsolved Nodes</th>
<th>Total Distance</th>
<th>$n$-th nearest Node</th>
<th>Minimum Distance</th>
<th>Last Arc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1 → 1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1 → 2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>1 → 4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>1 → 3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>5</td>
<td>11</td>
<td>6</td>
<td>10</td>
<td>4 → 6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>10</td>
<td></td>
<td></td>
<td>3 → 6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>5</td>
<td>11</td>
<td>5</td>
<td>11</td>
<td>2 → 5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td>3 → 5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>7</td>
<td>16</td>
<td>7</td>
<td>14</td>
<td>6 → 7</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>8</td>
<td>20</td>
<td>8</td>
<td>17</td>
<td>7 → 8</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.

<table>
<thead>
<tr>
<th>$n$</th>
<th>Solved Nodes</th>
<th>Closest Connected Unsolved Nodes</th>
<th>Total Distance</th>
<th>$n$-th nearest Node</th>
<th>Minimum Distance</th>
<th>Last Arc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1 → 1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1 → 2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>1 → 4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>11</td>
<td>6</td>
<td>10</td>
<td>4 → 6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>5</td>
<td>11</td>
<td>5</td>
<td>11</td>
<td>2 → 5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>3</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>3</td>
<td>14</td>
<td>3</td>
<td>14</td>
<td>6 → 3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>14</td>
<td>7</td>
<td>14</td>
<td>6 → 7</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>3</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>7</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>8</td>
<td>20</td>
<td>8</td>
<td>20</td>
<td>6 → 8</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>20</td>
<td></td>
<td></td>
<td>5 → 8</td>
</tr>
</tbody>
</table>

*Exercise:* In Figure 2, what is the shortest path from node 1 to node 3?
2.2.3 Shortest path in general network

Below we present another algorithm which is explicitly defined by a recursive DPE. This algorithm is very general and does not require the length of each arc to be non-negative. Define function

\[ v_j(n) \doteq \begin{cases} \text{the length of the shortest path from node 1 to node } n & \\
\text{when } j \text{ or fewer arcs must be used.} & 
\end{cases} \]

When there is no path from node 1 to node \( n \) with \( j \) or fewer arcs, then \( v_j(n) = \infty \).

With this formulation, one gets the shortest path from node 1 to any other nodes \( n \); indeed, the shortest path from node 1 to node \( n \) has a total length \( v_j(n) \) when \( j = N - 1 \) since the path can contain at most \( N - 1 \) arcs.

The DPE of \( v_j \) is given by

\[ v_{j+1}(n) = \min_{m \neq n} \left[ v_j(m) + d_{mn} \right], \quad \forall \ j = 0, 1, \cdots, N - 1, \ n = 1, 2, \cdots, N \]

(except for node \( n = 1 \), include \( m = n = 1 \) in the minimization). Note in this formulation, \( j \) represents the stage, and \( m, n \) represent the nodes. The initial conditions are given by

\[ v_0(n) = \begin{cases} \infty & \text{if } n = 1 \\
0 & \text{if } n = 2, 3, \cdots, N \end{cases} \]

Consider the following example.

![Diagram](image)

The calculation are as follows.

\[ v_0(1) = 0 \]
\[ v_0(2) = \infty \]
\[ v_0(3) = \infty \]
\[ v_0(4) = \infty. \]

\[ v_1(1) = \min[0 + 0, \infty + \infty, \infty + \infty, \infty + \infty] = 0, \quad (m^* = 1) \]
\[ v_1(2) = \min[0 + 2, \infty - 4, \infty + \infty] - 2, \quad (m^* = 1) \]
\[ v_1(3) = \min[0 + 5, \infty + 6, \infty + \infty] = 5, \quad (m^* = 1) \]
\[ v_1(4) = \min[0 + \infty, \infty + 3, \infty + 1] = \infty, \quad (m^* = 1, 2, 3). \]

\[ v_2(1) = \min[0 + 0, 2 + \infty, 5 + \infty, \infty + \infty] = 0, \quad (m^* = 1) \]
\[ v_2(2) = \min[0 + 2, 5 - 4, \infty + \infty] - 1, \quad (m^* = 3) \]
\[ v_2(3) = \min[0 + 5, 2 + 6, \infty + \infty] = 5, \quad (m^* = 1) \]
\[ v_2(4) = \min[0 + \infty, 2 + 3, 5 + 1] = 5, \quad (m^* = 2). \]
\[ v_3(1) = \min \{ 0 + 0, 1 + \infty, 5 + \infty, 5 + \infty \} = 0, \quad (m^* = 1) \]
\[ v_3(2) = \min \{ 0 + 2, 5 - 4, 5 + \infty \} = 1, \quad (m^* = 3) \]
\[ v_3(3) = \min \{ 0 + 5, 1 + 6, 5 + \infty \} = 5, \quad (m^* = 1) \]
\[ v_3(4) = \min \{ 0 + \infty, 1 + 3, 5 + 1 \} = 4, \quad (m^* = 2) \]

**Exercise:** Compute \( v_4 \) for each node, and verify that \( v_4 = v_3 \).

Therefore, the shortest path from node 1 to node 4 is 1 \( \rightarrow \) 3 \( \rightarrow \) 2 \( \rightarrow \) 4, which has total length 4.

### 2.3 Resource allocation

A special case of resource allocation is the so called *knapsack problem*. Consider the following example.

**Example:** Suppose a 10-lb knapsack is to be filled with the items listed in the table below. Assume there are unlimited number of items of each type. To maximize the total benefit, how should the knapsack be filled?

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight (lb)</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item 1</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>Item 2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Item 3</td>
<td>5</td>
<td>12</td>
</tr>
</tbody>
</table>

**Solution:** This optimization can indeed be represented by an LP.

Maximize \[ Z = 11x_1 + 7x_2 + 12x_3 \]

such that \[ 4x_1 + 3x_2 + 5x_3 \leq 10, \]

and \( x_1, x_2, x_3 \geq 0 \) are non-negative integers.

The LP is different from before is that the \( x_i \) is required to be an integer.

There are several ways to solve this problem using DP. In the following, we set \[ g(w) = \text{the maximum benefit that can be gained from a } w\text{-lb knapsack.} \]

The recursive equation to solve \( g(w) \) is

\[ g(w) = \max_j \{ b_j + g(w - w_j) \}; \]

here \( j \) denote the \( j \)-th item, with \( w_j \) its weight and \( b_j \) its benefit, and \( j \) must satisfy \( w_j \leq w \). The intuition behind this equation is self-obvious enough.
By definition, clearly \( g(0) = g(1) = g(2) = 0 \). Furthermore,

\[
\begin{align*}
  g(3) &= \max_j \{ b_j + g(w - w_j) \} = 7 + a(0) = 7, \quad (j^* = 2) \\
  g(4) &= \max \{ 11 + g(0), 7 + g(1) \} = 11, \quad (j^* = 1) \\
  g(5) &= \max \{ 11 + g(1), 7 + g(2), 12 + g(0) \} = 12, \quad (j^* = 3) \\
  g(6) &= \max \{ 11 + g(2), 7 + g(3), 12 + g(1) \} = 14, \quad (j^* = 2) \\
  g(7) &= \max \{ 11 + g(3), 7 + g(4), 12 + g(2) \} = 18, \quad (j^* = 1, 2) \\
  g(8) &= \max \{ 11 + g(4), 7 + g(5), 12 + g(3) \} = 22, \quad (j^* = 1) \\
  g(9) &= \max \{ 11 + g(5), 7 + g(6), 12 + g(4) \} = 23, \quad (j^* = 1, 3) \\
  g(10) &= \max \{ 11 + g(6), 7 + g(7), 12 + g(5) \} = 25, \quad (j^* = 1, 2)
\end{align*}
\]

Therefore the maximal benefit is 25, and the way to fill in the knapsack is that one item of type 1, and two items of type 2.

2.4 A Turnpike theorem

A drawback of the above dynamic programming is that when \( w \) is large, too many iterations are involved to solve for \( g(w) \). We will introduce a Turnpike theorem to reduce the computational effort. Observe that the “best item” is the item with the largest value of \( b_j/w_j \) (benefit per unit weight). Assume that \( n \) type of items have been ordered, so that

\[
\frac{b_1}{w_1} \geq \frac{b_2}{w_2} \geq \cdots \geq \frac{b_n}{w_n}.
\]

The ordering could be different from the original labeling.

**Turnpike Theorem**: Consider a knapsack problem for which

\[
\frac{b_1}{w_1} > \frac{b_2}{w_2}.
\]

Set

\[
w^* = b_1 \left( \frac{b_1}{w_1} - \frac{b_2}{w_2} \right).
\]

Suppose the knapsack can hold \( w \) pounds, with \( w \geq w^* \). Then the optimal solution to the knapsack problem must use at least one item of type 1.

**Proof**: Consider the same knapsack problem except that item of type 1 are not going to be used. The corresponding LP becomes

\[
\text{Maximize } Z = b_2 x_2 + b_3 x_3 + \cdots + b_n x_n
\]

such that

\[
w_2 x_2 + w_3 x_3 + \cdots + w_n x_n \leq w,
\]

and \( x_2, \cdots, x_n \geq 0 \) are non-negative integers.
But clearly, the value of this knapsack problem is at most \( b_2 w / w_2 \) (why?).

Now consider the following solution to the original knapsack problem: put as many items of type 1 as possible into the knapsack. This way, we can put in

\[
\left\lfloor \frac{w}{w_1} \right\rfloor
\]

items of type 1. Here \( \lfloor \cdot \rfloor \) denotes the integer part of a real number. Therefore, the benefit from this solution is at least

\[
b_1 \cdot \left\lfloor \frac{w}{w_1} \right\rfloor > b_1 \cdot \left( \frac{w}{w_1} - 1 \right) = b_1 \cdot \left( \frac{w}{w_1} - 1 \right)
\]

In case \( w \geq w^* \), we have

\[
b_1 \cdot \left\lfloor \frac{w}{w_1} \right\rfloor > b_1 \cdot \left( \frac{w}{w_1} - 1 \right) = w \cdot \left( \frac{b_1}{w_1} - b_1 \right) \geq w \cdot \left( \frac{b_2}{w_2} \right).
\]

Therefore, it turns out that an solution with no items of type 1 is never optimal. \( \square \)

**Corollary:** The optimal solution of the knapsack problem will indeed include at least \( 1 + \left\lfloor (w - w^*) / w_1 \right\rfloor \) items of type 1.

**Example:** Reconsider the preceding example but with \( w = 4000 \text{ lb} \).

**Solution.** We have

\[
\frac{11}{4} > \frac{12}{5} > 7 / 3.
\]

Therefore

\[
w^* = 11 \left( 11 - \frac{12}{5} \right) = \frac{220}{7}.
\]

Therefore, the optimal solution will contain at least

\[
1 + \left\lfloor \frac{w - w^*}{w_1} \right\rfloor = 1 + \left\lfloor \frac{4000 - 220/7}{4} \right\rfloor = 903
\]

items of type 1. Therefore, all we need to solve is a knapsack problem with \( w = 4000 - 993.4 = 28 \text{ lb} \) knapsack, which greatly reduces the computational effort.

**Exercise:** Complete the above example by solving the knapsack problem with \( w = 28 \text{ lb} \).

### 2.5 General resource allocation problem, formulation, DPE

A general resource allocation problem can be expressed as follows. Suppose we have \( w \) units of resource available, and \( N \) activities to which the resource can be allocated. If the activity \( n \) is implemented at the level \( x_n \) units (assumed to be a non-negative integer), then \( g_n(x_n) \) units of the resources are used and a benefit \( r_n(x_n) \) is obtained. The problem to maximize the total benefit subject to the limited resource availability may be written as
Maximize \[ \sum_{n=1}^{N} r_n(x_n) \]
such that
\[ \sum_{n=1}^{N} g_n(x_n) \leq w \]
and \( x_1, \ldots, x_N \) are non-negative integers.

In the knapsack problem, the total unit of resource is the weight the knapsack can hold; \( x_n \) stand for the number of type \( n \) items put in, and \( g_n \) and \( r_n \) are the weight and benefit of type \( n \) items, respectively.

One approach to formulate this problem as dynamic programming is as follows: Consider the functions
\[ v_j(w) = \max_{\{{\cdots}\}} \sum_{n=j}^{N} r_n(x_n), \quad \text{such that} \quad \sum_{n=j}^{N} g_n(x_n) \leq w, \]
with \( v_{N+1}(w) = 0 \). In this case \( v_j(w) \) can be interpreted as the maximal benefit one can receive if there are \( w \) unit of resources available to be allocated to resources \( j, \cdots, N \).

The DPE is
\[ v_j(w) = \max_x \{ r_j(x) + v_{j+1}(w - g_j(x)) \}, \quad \forall \ w, \]
where \( x \) must be a non-negative integer satisfying \( g_j(x) \leq w \).

**Exercise:** Solve the previous knapsack example using the dynamic programming equation above.

**Exercise:** The number of crimes on each of a city’s three police precincts depends on the number of patrol cars assigned to each precinct. Three patrol cars are available. Use dynamic programming to determine how many patrol cars should be assigned to each precinct so as to minimize the total number of crimes.

<table>
<thead>
<tr>
<th></th>
<th>No. of patrol cars</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Precinct 1</td>
<td>14</td>
</tr>
<tr>
<td>Precinct 2</td>
<td>25</td>
</tr>
<tr>
<td>Precinct 3</td>
<td>20</td>
</tr>
</tbody>
</table>

**Solution:** Let \( x_i \) = number of patrol cars assigned to Precinct \( i \). Then the optimization problem is

Minimize \[ \sum_{n=1}^{3} r_n(x_n) \]
such that
\[ \sum_{n=1}^{3} x_n \leq 5 \]
and \( x_1, x_2, x_3 \) are non-negative integers.
Define
\[ v_1(w) = \sum_{n=1}^{3} r_n(x_n) \]
such that
\[ \sum_{n=1}^{3} x_n \leq w, \]
and \( x_n \) are all non-negative integers.

- \( v_4(w) \equiv 0 \).
- \( v_3(w) = \min_{x} [v_3(x) + v_4(w-x)] - \min_{x} v_3(x) \) over all non-negative integer \( w \) such that \( x \leq w \). We have
  \[ \begin{align*}
  v_3(3) &= 8, \quad (x^* = 3) \\
  v_3(2) &= 11, \quad (x^* = 2) \\
  v_3(1) &= 14, \quad (x^* = 1) \\
  v_3(0) &= 20, \quad (x^* = 0). 
  \end{align*} \]

- \( v_2(w) = \min_{x} [r_2(x) + v_3(w-x)] \) over all non-negative integer \( x \) such that \( x \leq w \). We have
  \[ \begin{align*}
  v_2(3) &= \min [14 + 20, 16 + 14, 19 + 11, 25 + 8] = 30, \quad (x^* = 1, 2) \\
  v_2(2) &= \min [16 + 20, 19 + 14, 25 + 11] = 33, \quad (x^* = 1) \\
  v_2(1) &= \min [19 + 20, 25 + 14] = 39, \quad (x^* = 1, 0) \\
  v_2(0) &= \min [25 + 20] = 45, \quad (x^* = 0). 
  \end{align*} \]

- \( v_1(w) = \min_{x} [r_1(x) + v_2(w-x)] \) over all non-negative integer \( x \) such that \( x \leq w \). We have
  \[ v_1(3) = \min [4 + 45, 7 + 39, 10 + 33, 14 + 30] = 43, \quad (x^* = 1) \]

Therefore, the optimal solution is to assign one patrol car to each precinct, which will yield a minimal number of crimes 43.

\[ \Box \]

Exercise: What if there are only two patrol cars available?
3 Preliminary probabilistic dynamic programming

The deterministic DPE (for minimization of an additive cost criteria) can be loosely written as

$$V_j(\text{current state}) = \min_{\text{all feasible decisions}} \{ \text{cost during the current stage} + V_{j+1}(\text{new state}) \}.$$  

The idea for probabilistic dynamic programming is the same, the only difference is that now the new state is a random outcome and the goal is to minimize the average cost. The DPE in this case is very similar, with "$V_{j+1}(\text{new state})$" replaced by "Average of $V_{j+1}(\text{new state})$".

**Example:** A gambler has $2. She is allowed to play a game of chance two times, and her goal is to maximize her probability of ending up with at least $4. If she gambles $b$ dollars on a play of the game, with probability 0.4 she wins the game and increases her capital by $b$ dollars; with probability 0.6 she loses the game and decrease her capital by $b$ dollars. On any play of the game, the gambler cannot bet more than she has available. Design a betting strategy for the gambler.

**Solution:** For $j = 1, 2, 3$, define

$$V_j(x) = \text{Maximal probability of having at least $4$ at the end of game two}$$

$$\text{given that at the beginning of j-th play the gambler has capital x dollars.}$$

Also define $V_3(x) = 1$ if $x \geq 4$ and $V_3(x) = 0$ if $x < 4$. We are interested in $V_1(2)$.

One can solve the problem recursively. Indeed, we have the DPE

$$V_j(x) = \max_{b \in \{0, 1, \ldots, x\}} \text{Average of } V_{j+1}(\text{new state}) = \max_{b \in \{0, 1, \ldots, x\}} [0.4V_{j+1}(x + b) + 0.6V_{j+1}(x - b)]$$

for $j = 1, 2$ (why?) From $V_0$ we can recursively determine all $V_j$.

- $V_2(x) = \max_{b \in \{0, 1, \ldots, x\}} [0.4V_3(x + b) + 0.6V_3(x - b)]$. Clearly, we have

$$V_2(x) = \begin{cases} 1 & \text{if } x = 4, 5, \ldots \\ 0.4 & \text{if } x = 2, 3 \\ 0 & \text{if } x = 0, 1 \end{cases}$$

- $V_1(x) = \max_{b \in \{0, 1, \ldots, x\}} [0.4V_2(x + b) + 0.6V_2(x - b)]$. In particular,

$$V_1(2) = \max_{b \in \{0, 1, 2\}} [0.4V_2(2 + b) + 0.6V_2(2 - b)] = \begin{cases} 0.4 & b^* = 0 \\ 0.16 & b = 1 \\ 0.4 & b^* = 2 \end{cases}$$

Therefore, the maximal probability is 0.4 and one best policy is to bet 2 in the first play; if it is lost, so be it; if the gambler wins the first play, she should sit out in the second play. Another best strategy is to sit out in the first play, and bet 2 in the second play. \qed
Exercise: Suppose the gambler wants to maximize her probability of having at least $6 at the end of the 4th play. How should she play? (Solution: The optimal probability is 0.1984, and one of the optimal betting strategies is

Example: Tennis player Tom has two types of serves: a hard serve (H) and a soft serve (S). The probability that Tom’s hard serve will land in bounds is \( p_H \), and the probability of his soft serve will land in bounds is \( p_S \). If Tom’s hard serve lands in bounds, there is a probability \( w_H \) that Tom will win the point. If Tom’s soft serve lands in bounds, there is a probability \( w_S \) that Tom will win the point. We assume \( p_H < p_S \) and \( w_H > w_S \). Tom’s goal is to maximize the probability of winning a point on which he serves. Remember that if both serves are out of bounds, Tom loses the point.

Solution: Define \( f_i \), \( i = 1, 2 \) as the probability that Tom wins a point if he plays optimally and is about to take his \( i \)-th serve. To determine the optimal strategy, we will work backward. What is \( f_2 \)? If Tom serves hard on his second serve, he has a probability \( p_{HWH} \) to win the point, while he has probability \( p_{SWS} \) to win the point if he serves soft. Therefore, we have

\[
f_2 = \max\{p_{HWH}, p_{SWS}\}.
\]

To determine \( f_1 \), observe the following equation:

\[
f_1 = \max\{p_{HWH} + (1 - p_H)f_2, p_{SWS} + (1 - p_S)f_2\}
\]

There are three possibilities,

1. \( p_{HWH} \geq p_{SWS} \): In this case, \( f_2 = p_{HWH} \) and

\[
f_1 = \max\{p_{HWH} + (1 - p_H)p_{HWH}, p_{SWS} + (1 - p_S)p_{HWH}\} = p_{HWH} + (1 - p_H)p_{HWH},
\]

since

\[
(p_{HWH} + (1 - p_H)p_{HWH}) - (p_{SWS} + (1 - p_S)p_{HWH})
= p_{HWH}(1 + p_S - p_H) - p_{SWS}
> p_{HWH} - p_{SWS} > 0.
\]

Therefore, Tom should serve hard on both serves.
2. \( p_{SW}(1 + PH - PS) \leq PH\text{WH} < p_{SW} \): In this case, \( f_2 = p_{SW} \), and
\[
f_1 = \max \{ PH\text{WH} + (1 - PH)\text{PS} \text{WH} + (1 - PS)\text{PS} \text{WH} \} = PH\text{WH} + (1 - PH)\text{PS} \text{WH}
\]
So Tom should serve hard on the first serve, and soft on the second.

3. \( PH\text{WH} < p_{SW}(1 + PH - PS) \). In this case, \( f_2 = p_{SW} \), and
\[
f_1 = \max \{ PH\text{WH} + (1 - PH)p_{SW}, p_{SW} + (1 - PS)p_{SW} \} = p_{SW} + (1 - PS)p_{SW},
\]
and Tom should serve soft on both serves.

**Example:** We will return to the bass problem: Every year the owner of a lake must decide how many bass to capture and sell. During year \( n \) the price is \( p_n \), for the bass. If the lake contains \( b \) bass in the beginning of the year \( n \), the cost of catching \( x \) bass is \( c_n(x|b) \). But now the bass population grows by a random factor \( D \), where \( \text{Prob}(D = d) = q(d) \). Can you formulate a dynamic programming recursion if the owner wants to maximize the average net profit over the next five years?

**Solution:** Define for \( n = 1, 2, \cdots, 5 \),
\[
v_n(b) = \text{the maximal average net profit during the years } n, n + 1, \cdots, 5
\]
if the lake contains \( b \) bass at the beginning of year \( n \).

and \( v_0(b) \equiv 0 \). The recursive DPE is
\[
v_n(b) = \max_{0 \leq x \leq b} \left[ xp_n - c_n(x|b) + \sum_d q(d) v_{n+1}(d(b - x)) \right].
\]