5

Dynamic Programming

5.1 SOME MOTIVATION: SHORTEST PATHS

Here we look at another approach to solving certain combinatorial optimization problems. To see the basic idea, consider the shortest path problem again. Given a directed graph $D = (V, A)$, nonnegative arc distances c_e for $e \in A$, and an initial node $s \in V$, the problem is to find the shortest path from s to every other node $v \in V \setminus \{s\}$. See Figure 5.1, in which a shortest path from s to t is shown, as well as one intermediate node p on the path.

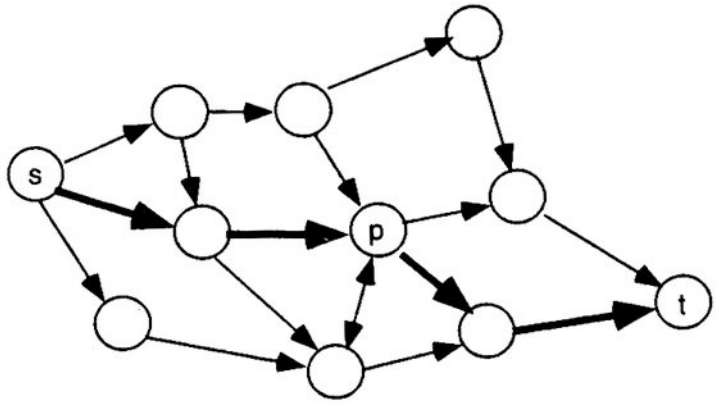

Fig. 5.1 Shortest s-t path

Observation 5.1 If the shortest path from s to t passes by node p, the subpaths (s, p) and (p, t) are shortest paths from s to p, and p to t respectively.

If this were not true, the shorter subpath would allow us to construct a shorter path from s to t, leading to a contradiction.

The question is how to use this idea to find shortest paths.

Observation 5.2 Let $d(v)$ denote the length of a shortest path from s to v. Then

$$d(v) = \min_{i \in V^-(v)} \{d(i) + c_{iv}\}. \tag{5.1}$$

In other words, if we know the lengths of the shortest paths from s to every neighbor (predecessor) of v, then we can find the length of the shortest path from s to v.

This still does not lead to an algorithm for general digraphs, because we may need to know $d(i)$ to calculate $d(j)$, and $d(j)$ to calculate $d(i)$. However, for certain digraphs, it does provide a simple algorithm.

Observation 5.3 Given an acyclic digraph $D = (V, A)$ with $n = |V|, m = |A|$, where the nodes are ordered so that $i < j$ for all arcs $(i, j) \in A$, then for the problem of finding shortest paths from node 1 to all other nodes, the recurrence (5.1) for $v = 2, \ldots, n$ leads to an $O(m)$ algorithm.

For arbitrary directed graphs with nonnegative weights $c \in R_+^{|A|}$, we need to somehow impose an ordering. One way to do this is to define a more general function $D_k(i)$ as the length of a shortest path from s to i containing at most k arcs. Then we have the recurrence:

$$D_k(j) = \min\{D_{k-1}(j), \min_{i \in V^-(j)} [D_{k-1}(i) + c_{ij}]\}.$$

Now by increasing k from 1 to $n-1$, and each time calculating $D_k(j)$ for all $j \in V$ by the recursion, we end up with an $O(mn)$ algorithm and $d(j) = D_{n-1}(j)$.

This approach whereby an optimal solution value for one problem is calculated recursively from the optimal values of slightly different problems is called *Dynamic Programming (DP)*. Below we will see how it is possible to apply similar ideas to derive a recursion for several interesting problems. The standard terminology used is the *Principle of Optimality* for the property that pieces of optimal solutions are themselves optimal, *states* that correspond to the nodes for which values need to be calculated, and *stages* for the steps which define the ordering.

5.2 UNCAPACITATED LOT-SIZING

The uncapacitated lot-sizing problem (ULS) was introduced in Chapter 1 where two different mixed integer programming formulations were presented. The problem again is to find a minimum cost production plan that satisfies

all the nonnegative demands $\{d_t\}_{t=1}^n$, given the costs of production $\{p_t\}_{t=1}^n$, storage $\{h_t\}_{t=1}^n$, and set-up $\{f_t\}_{t=1}^n$. We assume $f_t \geq 0$ for all t.

To obtain an efficient dynamic programming algorithm, it is necessary to understand the structure of the optimal solutions. For this it is useful to view the problem as a network design problem. Repeating the MIP formulation of Section 1.4, we have:

$$\min \sum_{t=1}^{n} p_t x_t + \sum_{t=1}^{n} h_t s_t + \sum_{t=1}^{n} f_t y_t \tag{5.2}$$

$$s_{t-1} + x_t = d_t + s_t \text{ for } t = 1, \ldots, n \tag{5.3}$$

$$x_t \leq M y_t \text{ for } t = 1, \ldots, n \tag{5.4}$$

$$s_0 = s_n = 0, s \in R_+^{n+1}, x \in R_+^n, y \in B^n \tag{5.5}$$

where x_t denotes the production in period t, and s_t the stock at the end of period t. We see that every feasible solution corresponds to a flow in the network shown in Figure 5.2,

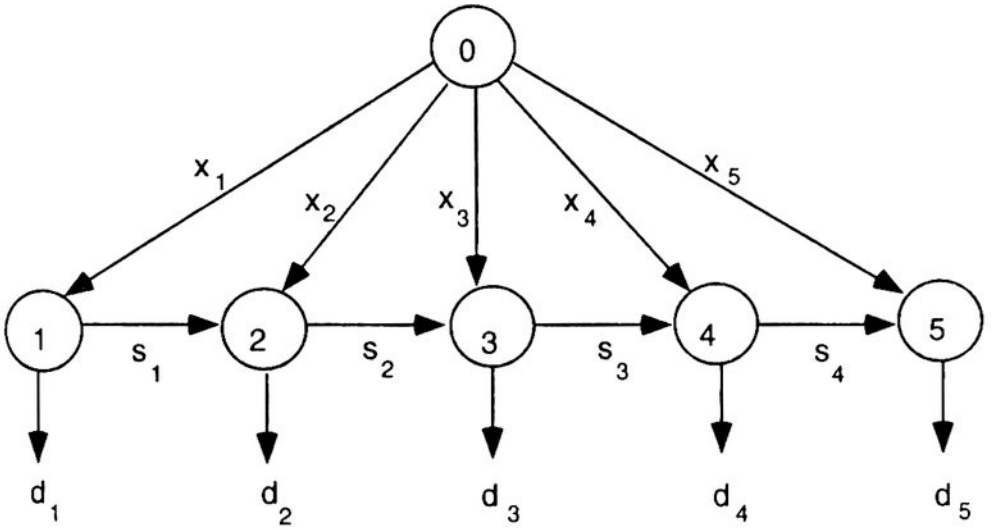

Fig. 5.2 Network for lot-sizing problem

where p_t is the flow cost on arc $(0,t)$, h_t is the flow cost on arc $(t,t+1)$, and the fixed costs f_t are incurred if $x_t > 0$ on arc $(0,t)$.

Thus ULS is a fixed charge network flow problem in which one must choose which arcs $(0,t)$ are open (which are the production periods), and then find a minimum cost flow through the network. Optimal solutions to ULS have two important structural properties that follow from this network viewpoint.

Proposition 5.1 *(i) There exists an optimal solution with $s_{t-1}x_t = 0$ for all t. (Production takes place only when the stock is zero.)*
(ii) There exists an optimal solution such that if $x_t > 0, x_t = \sum_{i=t}^{t+k} d_i$ for some $k > 0$. (If production takes place in t, the amount produced exactly satisfies demand for periods t to $t+k$.)

Proof. Suppose that the production periods have been chosen optimally (certain arcs $(0,t)$ are open). Now as an optimal extreme flow uses a set of

arcs forming a tree, the set of arcs with positive flow contains no cycle, and it follows that only one of the arcs arriving at node t can have a positive flow (i.e., $s_{t-1}x_t = 0$). The second statement then follows immediately. ∎

Property (ii) is the important property we need to derive a DP algorithm for ULS.

For the rest of this section we let d_{it} denote the sum of demands for periods i up to t (i.e., $d_{it} = \sum_{j=i}^{t} d_j$). The next observation is repeated from Section 1.4 to simplify the calculations.

Observation 5.4 As $s_t = \sum_{i=1}^{t} x_i - d_{1t}$, the stock variables can be eliminated from the objective function giving $\sum_{t=1}^{n} p_t x_t + \sum_{t=1}^{n} h_t s_t = \sum_{t=1}^{n} p_t x_t + \sum_{t=1}^{n} h_t(\sum_{i=1}^{t} x_i - d_{1t}) = \sum_{t=1}^{n} c_t x_t - \sum_{t=1}^{n} h_t d_{1t}$ where $c_t = p_t + \sum_{i=t}^{n} h_i$. This allows us to work with the modified cost function $\sum_{t=1}^{n} c_t x_t + 0 \sum_{t=1}^{n} s_t + \sum_{t=1}^{n} f_t y_t$, and the constant term $\sum_{t=1}^{n} h_t d_{1t}$ must be subtracted at the end of the calculations.

Let $H(k)$ be the minimum cost of a solution for periods $1, \ldots, k$. If $t \leq k$ is the last period in which production occurs (namely $x_t = d_{tk}$), what happens in periods $1, \ldots, t-1$? Clearly the least cost solution must be optimal for periods $1, \ldots, t-1$, and thus has cost $H(t-1)$. This gives the recursion.

Forward Recursion

$$H(k) = \min_{1 \leq t \leq k} \{H(t-1) + f_t + c_t d_{tk}\}$$

with $H(0) = 0$.

Calculating $H(k)$ for $k = 1, \ldots, n$ leads to the value $H(n)$ of an optimal solution of ULS. Working back gives a corresponding optimal solution. It is also easy to see that $O(n^2)$ calculations suffice to obtain $H(n)$ and an optimal solution.

Example 5.1 Consider an instance of ULS with $n = 4, d = (2, 4, 5, 1), p = (3, 3, 3, 3), h = (1, 2, 1, 1)$ and $f = (12, 20, 16, 8)$. We start by calculating $c = (8, 7, 5, 4), (d_{11}, d_{12}, d_{13}, d_{14}) = (2, 6, 11, 12)$ and the constant $\sum_{t=1}^{4} h_t d_{1t} = 37$. Now we successively calculate the values of $H(k)$ using the recursion.

$H(1) = f_1 + c_1 d_1 = 28.$

$H(2) = \min[28 + c_1 d_2, H(1) + f_2 + c_2 d_2] = \min[60, 76] = 60.$

$H(3) = \min[60 + c_1 d_3, 76 + c_2 d_3, H(2) + f_3 + c_3 d_3] = \min[100, 111, 101] = 100.$

$H(4) = \min[100 + c_1 d_4, 111 + c_2 d_4, 101 + c_3 d_4, H(3) + f_4 + c_4 d_4]$
$= \min[108, 118, 106, 112] = 106.$

Working backwards, we see that $H(4) = 106 = H(2) + f_3 + c_3 d_{34}$, so $y_3 = 1, x_3 = 6, y_4 = x_4 = 0$. Also $H(2) = f_1 + c_1 d_{12}$, so $y_1 = 1, x_1 = 6, y_2 = x_2 = 0$. Thus we have found an optimal solution $x = (6, 0, 6, 0), y = (1, 0, 1, 0), s =$

$(4,0,1,0)$ whose value in the original costs is $106 - 37 = 69$. Checking we have $6p_1 + f_1 + 6p_3 + f_3 + 4h_1 + 1h_3 = 69$. ∎

Another possibility is to solve ULS directly as a shortest path problem. Consider a directed graph with nodes $\{0, 1, \ldots, n\}$ and arcs (i, j) for all $i < j$. The cost $f_{i+1} + c_{i+1}d_{i+1,j}$ of arc (i, j) is the cost of starting production in $i+1$ and satisfying the demand for periods $i + 1$ up to j. Figure 5.3 shows the shortest path instance arising from the data of Example 5.1. Now a least cost path from node 0 to node n provides a minimum cost set of production intervals and solves ULS.

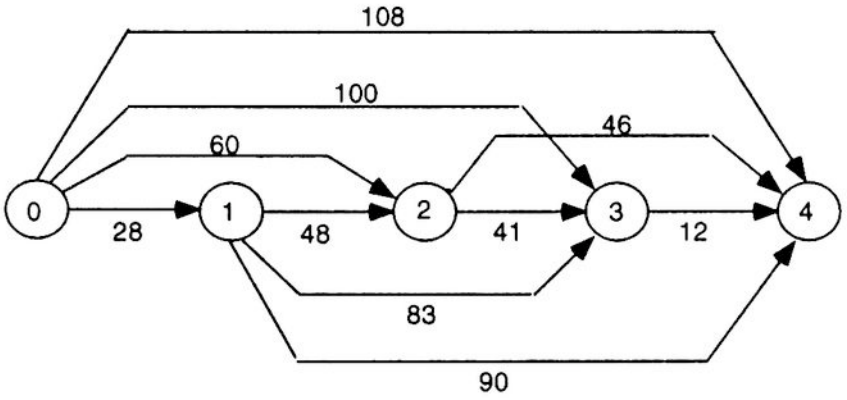

Fig. 5.3 Shortest path for ULS

We observe that $H(k)$ is the cost of a cheapest path from nodes 0 to k, and as the directed graph is acyclic, we know from Observation 5.3 that the corresponding shortest path algorithm is $O(m) = O(n^2)$.

5.3 AN OPTIMAL SUBTREE OF A TREE

Here we consider another problem that can be tackled by dynamic programming. However, the recursion here is not related at all to shortest paths. The *Optimal Subtree of a Tree Problem* involves a tree $T = (V, E)$ with a root $r \in V$ and weights c_v for $v \in V$. The problem is to choose a subtree of T rooted at r of maximum weight, or the empty tree if there is no positive weight rooted subtree.

To describe a dynamic programming recursion we need some notation. For a rooted tree, each node v has a well-defined *predecessor* $p(v)$ on the unique path from the root r to v, and, for $v \neq r$, a set of *immediate successors* $S(v) = \{w \in V : p(w) = v\}$. Also we let $T(v)$ be the *subtree of* T *rooted at* v containing all nodes w for which the path from r to w contains v.

For any node v of T, let $H(v)$ denote the optimal solution value of the rooted subtree problem defined on the tree $T(v)$ with node v as the root. If the optimal subtree is empty, clearly $H(v) = 0$. Otherwise the optimal subtree contains v. It may also contain subtrees of $T(w)$ rooted at w for $w \in S(v)$. By

the principle of optimality, these subtrees must themselves be optimal rooted subtrees. Hence we obtain the recursion:

$$H(v) = \max\{0, c_v + \sum_{w \in S(v)} H(w)\}.$$

To initialize the recursion, we start with the leaves (nodes having no successors) of the tree. For a leaf $v \in V$, $H(v) = \max[c_v, 0]$. The calculations are then carried out by working in from the leaves to the root, until the optimal value $H(r)$ is obtained. As before, an optimal solution is then easily found by working backwards out from the root, eliminating every subtree $T(v)$ encountered with $H(v) = 0$. Finally note that each of the terms c_v and $H(v)$ occurs just once on the right-hand side during the recursive calculations, and so the algorithm is $O(n)$.

Example 5.2 For the instance of the optimal subtree of a tree problem shown in Figure 5.4 with root $r = 1$, we start with the leaf nodes $H(4) = H(6) = H(7) = H(11) = 0, H(9) = 5, H(10) = 3, H(12) = 3$, and $H(13) = 3$. Working in, $H(5) = \max[0, -6 + 5 + 3] = 2$ and $H(8) = \max[0, 2 + 0 + 3 + 3] = 8$. Now the values of $H(v)$ for all successors of nodes 2 and 3 are known, and so $H(2) = 4$ and $H(3) = 0$ can be calculated. Finally $H(1) = \max[0, -2+4+0] = 2$. Cutting off subtrees $T(3), T(4)$, and $T(6)$ leaves an optimal subtree with nodes 1,2,5,9,10 of value $H(1) = 2$. ∎

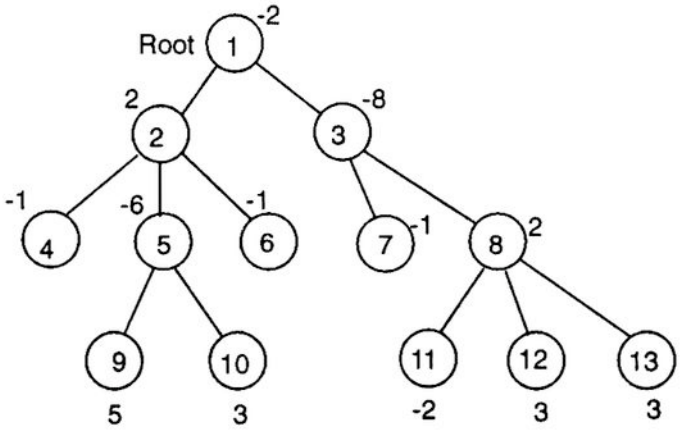

Fig. 5.4 Rooted tree with node weights c_v

5.4 KNAPSACK PROBLEMS

Here we examine various knapsack problems. Whereas ULS and the optimal subtree problems have the Efficient Optimization Property, knapsack problems in general are more difficult. This is made more precise in the next chapter. Dynamic programming provides an effective approach for such problems if the size of the data is restricted.

5.4.1 0–1 Knapsack

First we consider the 0–1 knapsack problem:

$$\begin{aligned} z = \max & \textstyle\sum_{j=1}^{n} c_j x_j \\ & \textstyle\sum_{j=1}^{n} a_j x_j \leq b \\ & x \in B^n \end{aligned}$$

where the coefficients $\{a_j\}_{j=1}^{n}$ and b are positive integers.

Thinking of the right-hand side λ taking values from $0, 1, \ldots, b$ as the state, and the subset of variables $x_1, \ldots, x_k$ represented by k as the stage, leads us to define the problem $P_r(\lambda)$ and the optimal value function $f_r(\lambda)$ as follows:

$$(P_r(\lambda)) \qquad \begin{aligned} f_r(\lambda) = \max & \textstyle\sum_{j=1}^{r} c_j x_j \\ & \textstyle\sum_{j=1}^{r} a_j x_j \leq \lambda \\ & x \in B^r. \end{aligned}$$

Then $z = f_n(b)$ gives us the optimal value of the knapsack problem. Thus we need to define a recursion that allows us to calculate $f_r(\lambda)$ in terms of values of $f_s(\mu)$ for $s \leq r$ and $\mu < \lambda$.

What can we say about an optimal solution x^* for problem $P_r(\lambda)$ with value $f_r(\lambda)$? Clearly either $x_r^* = 0$ or $x_r^* = 1$.

(i) If $x_r^* = 0$, then by the same optimality argument we used for shortest paths, $f_r(\lambda) = f_{r-1}(\lambda)$.

(ii)If $x_r^* = 1$, then $f_r(\lambda) = c_r + f_{r-1}(\lambda - a_r)$.

Thus we arrive at the recursion:

$$f_r(\lambda) = \max\{f_{r-1}(\lambda), c_r + f_{r-1}(\lambda - a_r)\}.$$

Now starting the recursion with $f_0(\lambda) = 0$ for $\lambda \geq 0$, or alternatively with $f_1(\lambda) = 0$ for $0 \leq \lambda < a_1$ and $f_1(\lambda) = \max[c_1, 0]$ for $\lambda \geq a_1$, we then use the recursion to successively calculate $f_2, f_3, \ldots, f_n$ for all integral values of λ from 0 to b.

The question that then remains is how to find an associated optimal solution. For this we have two related options. In both cases we iterate back from the optimal value $f_n(b)$. Either we must keep all the $f_r(\lambda)$ values, or an indicator $p_r(\lambda)$ which is 0 if $f_r(\lambda) = f_{r-1}(\lambda)$, and 1 otherwise.

If $p_n(b) = 0$, then as $f_n(b) = f_{n-1}(b)$, we set $x_n^* = 0$ and continue by looking for an optimal solution of value $f_{n-1}(b)$.

If $p_n(b) = 1$, then as $f_n(b) = c_n + f_{n-1}(b - a_n)$, we set $x_n^* = 1$ and then look for an optimal solution of value $f_{n-1}(b - a_n)$.

Iterating n times allows us to obtain an optimal solution.

Counting the number of calculations required to arrive at $z = f_n(b)$, we see that for each calculation $f_r(\lambda)$ for $\lambda = 0, 1, \ldots, b$ and $r = 1, \ldots, n$ there are a constant number of additions, subtractions, and comparisons. Calculating the optimal solution requires at most the same amount of work. Thus the DP algorithm is $O(nb)$.

Example 5.3 Consider the 0–1 knapsack instance:

$$\begin{aligned} z = \max\ & 10x_1 + 7x_2 + 25x_3 + 24x_4 \\ & 2x_1 + 1x_2 + 6x_3 + 5x_4 \leq 7 \\ & x \in B^4. \end{aligned}$$

The values of $f_r(\lambda)$ and $p_r(\lambda)$ are shown in Table 5.1. The values of $f_1(\lambda)$ are calculated by the formula described above. The next column is then calculated from top to bottom using the recursion. For example, $f_2(7) = \max\{f_1(7), 7 + f_1(7-1)\} = \max\{10, 7+10\} = 17$, and as the second term of the maximization gives the value of $f_2(7)$, we set $p_2(7) = 1$. The optimal value $z = f_4(7) = 34$.

	f_1	f_2	f_3	f_4	p_1	p_2	p_3	p_4
λ =0	0	0	0	0	0	0	0	0
1	0	7	7	7	0	1	0	0
2	10	10	10	10	1	0	0	0
3	10	17	17	17	1	1	0	0
4	10	17	17	17	1	1	0	0
5	10	17	17	24	1	1	0	1
6	10	17	25	31	1	1	1	1
7	10	17	32	34	1	1	1	1

Table 5.1 $f_r(\lambda)$ for a 0–1 knapsack problem

Working backwards, $p_4(7) = 1$ and hence $x_4^* = 1$. $p_3(7-5) = p_3(2) = p_2(2) = 0$ and hence $x_3^* = x_2^* = 0$. $p_1(2) = 1$ and hence $x_1^* = 1$. Thus $x^* = (1, 0, 0, 1)$ is an optimal solution. ∎

5.4.2 Integer Knapsack Problems

Now we consider the integer knapsack problem:

$$\begin{aligned} z = \max\ & \textstyle\sum_{j=1}^{n} c_j x_j \\ & \textstyle\sum_{j=1}^{n} a_j x_j \leq b \\ & x \in Z_+^n \end{aligned}$$

where again the coefficients $\{a_j\}_{j=1}^n$ and b are positive integers. Copying from the 0–1 case, we define $P_r(\lambda)$ and the value function $g_r(\lambda)$ as follows:

$$(P_r(\lambda)) \qquad \begin{aligned} g_r(\lambda) = \max & \textstyle\sum_{j=1}^r c_j x_j \\ & \textstyle\sum_{j=1}^r a_j x_j \leq \lambda \\ & x \in Z_+^r. \end{aligned}$$

Then $z = g_n(b)$ gives us the optimal value of the integer knapsack problem.

To build a recursion, a first idea is to again copy from the 0–1 case. If x^* is an optimal solution to $P_r(\lambda)$ giving value $g_r(\lambda)$, then we consider the value of x_r^*. If $x_r^* = t$, then using the principle of optimality $g_r(\lambda) = c_r t + g_{r-1}(\lambda - t a_r)$ for some $t = 0, 1, \ldots, \lfloor \frac{\lambda}{a_r} \rfloor$, and we obtain the recursion:

$$g_r(\lambda) = \max_{t=0,1,\ldots,\lfloor \lambda/a_r \rfloor} \{c_r t + g_{r-1}(\lambda - t a_r)\}.$$

As $\lfloor \frac{\lambda}{a_r} \rfloor = b$ in the worst case, this gives an algorithm of complexity $O(nb^2)$.

Can one do better? Is it possible to reduce the calculation of $g_r(\lambda)$ to a comparison of only two cases?

(i) Taking $x_r^* = 0$, we again have $g_r(\lambda) = g_{r-1}(\lambda)$.

(ii) Otherwise, we must have $x_r^* \geq 1$, and we can no longer copy from above.

However, as $x_r^* = 1 + t$ with t a nonnegative integer, we claim, again by the principle of optimality, that if we reduce the value of x_r^* by 1, the remaining vector $(x_1^*, \ldots, x_{r-1}^*, t)$ must be optimal for the problem $P_r(\lambda - a_r)$. Thus we have $g_r(\lambda) = c_r + g_r(\lambda - a_r)$, and we arrive at the recursion:

$$g_r(\lambda) = \max\{g_{r-1}(\lambda), c_r + g_r(\lambda - a_r)\}.$$

This now gives an algorithm of complexity $O(nb)$, which is the same as that of the 0–1 problem. We again set $p_r(\lambda) = 0$ if $g_r(\lambda) = g_{r-1}(\lambda)$ and $p_r(\lambda) = 1$ otherwise.

Example 5.4 Consider the knapsack instance:

$$\begin{aligned} z = \max & \; 7x_1 + 9x_2 + 2x_3 + 15x_4 \\ & 3x_1 + 4x_2 + 1x_3 + 7x_4 \leq 10 \\ & x \in Z_+^4. \end{aligned}$$

The values of $g_r(\lambda)$ are shown in Table 5.2. With $c_1 \geq 0$, the values of $g_1(\lambda)$ are easily calculated to be $c_1 \lfloor \frac{\lambda}{a_1} \rfloor$. The next column is then calculated from top to bottom using the recursion. For example, $g_2(8) = \max\{g_1(8), 9 + g_2(8-4)\} = \max\{14, 9+9\} = 18$.

Working back, we see that $p_4(10) = p_3(10) = 0$ and thus $x_4^* = x_3^* = 0$. $p_2(10) = 1$ and $p_2(6) = 0$ and so $x_2^* = 1$. $p_1(6) = p_1(3) = 1$ and thus $x_1^* = 2$. Hence we obtain an optimal solution $x^* = (2, 1, 0, 0)$. ∎

	g_1	g_2	g_3	g_4	p_1	p_2	p_3	p_4
λ =0	0	0	0	0	0	0	0	0
1	0	0	2	2	0	0	1	0
2	0	0	4	4	0	0	1	0
3	7	7	7	7	1	0	0	0
4	7	9	9	9	1	1	0	0
5	7	9	11	11	1	1	1	0
6	14	14	14	14	1	0	0	0
7	14	16	18	18	1	1	1	0
8	14	18	18	18	1	1	0	0
9	21	21	21	21	1	0	0	0
10	21	23	23	23	1	1	0	0

Table 5.2 $g_r(\lambda)$ for an integer knapsack problem

Another recursion can be used for the integer knapsack problem. Looking at the example above, we see that in fact all the important information is contained in the n^{th} column containing the values of $g_n(\lambda)$. Writing h in place of g_n, can we directly write a recursion for $h(\lambda)$?

Again the principle of optimality tells us that if x^* is an optimal solution of $P_n(\lambda)$ of value

$$h(\lambda) = \max\{\sum_{j=1}^{n} c_j x_j : \sum_{j=1}^{n} a_j x_j \leq \lambda, x \in Z_+^n\}$$

with $x_j^* \geq 1$, then $h(\lambda) = c_j + h(\lambda - a_j)$.

Thus we obtain the recursion:

$$h(\lambda) = \max[0, \max_{j:a_j \leq \lambda} \{c_j + h(\lambda - a_j)\}].$$

This also leads to an $O(nb)$ algorithm. Applied to the instance of Example 5.4, it gives precisely the values in the g_4 column of Table 5.2.

As a final observation, the dynamic programming approach for knapsack problems can also be viewed as a longest path problem. Construct an acyclic digraph $D = (V, A)$ with nodes $0, 1, \ldots, b$, arcs $(\lambda, \lambda + a_j)$ for $\lambda \in Z_+^1, \lambda \leq b - a_j$ with weight c_j for $j = 1, \ldots, n$, and 0-weight arcs $(\lambda, \lambda + 1)$ for $\lambda \in Z_+^1, \lambda \leq b - 1$. $h(\lambda)$ is precisely the value of a longest path from node 0 to node λ. Figure 5.5 shows the digraph arising from the instance:

$$z = \max 10x_1 + 7x_2 + 25x_3 + 24x_4$$
$$2x_1 + 1x_2 + 6x_3 + 5x_4 \leq 7$$
$$x \in Z_+^4,$$

except that the 0-weight arcs $(\lambda, \lambda + 1)$ are omitted by dominance.

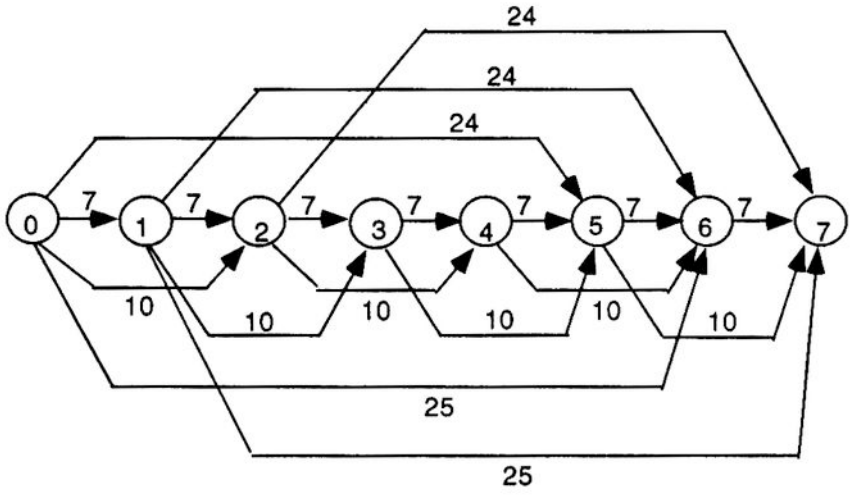

Fig. 5.5 Knapsack longest path problem

5.5 NOTES

5.1 The principle of optimality and dynamic programming originated with Bellman [Bell57]. One of the more recent books on dynamic programming is [Den82]. Shortest path problems with additional restrictions arise as subproblems in many routing problems, and are solved by dynamic programming, see [Desetal95].

5.2 The uncapacitated lot-sizing model and the dynamic programming algorithm for it are from [WagWhi58]. Dynamic programming recursions have been developed for many generalizations including lot-sizing with backlogging [Zan66], with constant production capacities [FloKle71], with start-up costs [Fle90], and with lower bounds on production [Con98]. Recently [WagvanHKoe92] among others have shown how the running time of the DP algorithm for the basic model can be significantly improved.

5.3 Various generalizations of the subtree problem are of interest. The more general problem of finding an optimal upper/lower set in a partially ordered set is examined in [GroLie81], and shown to be solvable as a network flow problem.

The problem of partitioning a tree into subtrees is tackled using dynamic programming in [BarEdmWol86], and this model is further studied in [AghMagWol95]. A telecommunications problem with such structure is studied in [Balaketal95].

Many other combinatorial optimization problems become easy when the underlying graph is a tree [MagWol95], or more generally a series-parallel graph [Taketal82].

5.4 The classical paper on the solution of knapsack problems by dynamic programming is [GilGom66]. The longest path or dynamic programming viewpoint was later extended by Gomory leading to the group relaxation of an

integer program [Gom65], and later to the superadditive duality theory for integer programs; see Notes of Section 2.2. The idea of reversing the roles of the objective and constraint rows (Exercise 5.7) is from [IbaKim75].

The dynamic programming appoach to TSP was first described by [HelKar62]. Relaxations of this approach, known as *state space relaxation*, have been used for a variety of constrained path and routing problems; see [ChrMinTot81]. Recently [Psa80] and [Balas95] have shown how such a recursion is of practical interest when certain restrictions on the tours or arrival sequences are imposed.

5.6 EXERCISES

1. Solve the uncapacitated lot-sizing problem with $n = 4$ periods, unit production costs $p = (1, 1, 1, 2)$, unit storage costs $h = (1, 1, 1, 1)$, set-up costs $f = (20, 10, 45, 15)$, and demands $d = (8, 5, 13, 4)$.

2. Consider the uncapacitated lot-sizing problem with backlogging ($ULSB$). *Backlogging* means that demand in a given period can be satisfied from production in a later period. If $r_t \geq 0$ denotes the amount backlogged in period t, the flow conservation constraints (5.3) become

$$s_{t-1} - r_{t-1} + x_t = d_t + s_t - r_t.$$

Show that there always exists an optimal solution to $ULSB$ with

(i) $s_{t-1}x_t = x_t r_t = s_{t-1}r_t = 0$.
(ii) $x_t > 0$ implies $x_t = \sum_{i=p}^{q} d_i$ with $p \leq t \leq q$.

Use this to derive a dynamic programming recursion for $ULSB$, or to reformulate $ULSB$ as a shortest path problem.

3. Find a maximum weight rooted subtree for the rooted tree shown in Figure 5.6.

4. Formulate the optimal subtree of a tree problem as an integer program. Is this IP easy to solve?

5. Given a digraph $D = (V, A)$, travel times c_{ij} for $(i, j) \in A$ for traversing the arcs, and earliest passage times r_j for $j \in V$, consider the problem of minimizing the time required to go from node 1 to node n.

(i) Describe a dynamic programming solution.
(ii) Formulate as a mixed integer program. Is this mixed integer program easy to solve?

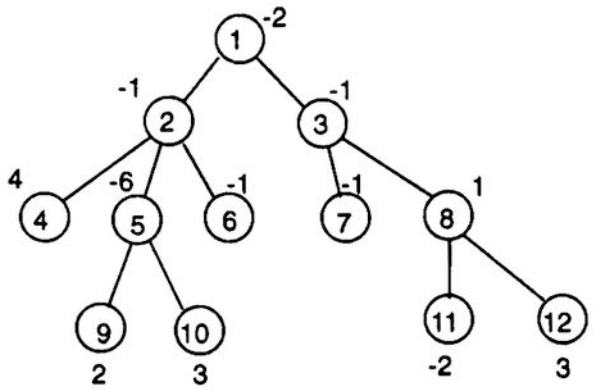

Fig. 5.6 Rooted tree with node weights c_v

6. Solve the knapsack problem

$$\min 5x_1 - 3x_2 + 7x_3 + 10x_4$$
$$2x_1 - x_2 + 4x_3 + 5x_4 \geq \lambda$$
$$x_1, x_4 \in Z_+^1, x_2, x_3 \in B^1$$

for all values of λ between 7 and 10.

7. Let $f(\lambda) = \max\{\sum_{j=1}^n c_j x_j : \sum_{j=1}^n a_j x_j \leq \lambda, x \in X \subset R^n\}$ and $h(t) = \min\{\sum_{j=1}^n a_j x_j : \sum_{j=1}^n c_j x_j \geq t, x \in X\}$.
Show that

(i) $f(\lambda) \geq t$ if and only if $h(t) \leq \lambda$.
(ii) $f(b) = \max\{t : h(t) \leq b\}$.

Use these observations to solve the knapsack problem

$$\max 3x_1 + 4x_2 + 6x_3 + 5x_4 + 8x_5$$
$$412x_1 + 507x_2 + 714x_3 + 671x_4 + 920x_5 \leq 1794$$
$$x \in B^5$$

by dynamic programming.

8. Solve the problem

$$\max x_1^3 + 2x_2^{\frac{1}{2}} + 4x_3 + 4x_4$$
$$2x_1^2 + 4x_2 + 6x_3 + 5x_4 \leq t$$
$$x_1 \leq 3, x_2 \leq 2, x_4 \leq 1$$
$$x \in Z_+^4$$

for $t \leq 12$. (Hint. One of the recursions in the chapter is appropriate.)

9. Formulate the 0–1 knapsack problem as a longest path problem.

10. Derive a dynamic programming recursion for the $STSP$ using $f(S,j)$ with $1, j \in S$ where $f(S,j)$ denotes the length of a shortest Hamiltonian path starting at node 1, passing through the nodes of $S \setminus \{1,j\}$ and terminating at node j.

11. Given the weighted rooted subtree of Figure 5.6, devise a dynamic programming algorithm to find optimal weighted subtrees with $k = 2, 3, \ldots, n-1$ nodes.

12. Given a tree T and a list of $T_1, \ldots, T_m$ of subtrees of T with weights $c(T_i)$ for $i = 1, \ldots, m$, describe an algorithm to find a maximum weight packing of node disjoint subtrees.

13.* Given a rooted tree T on n nodes, and an n by n matrix C, describe an algorithm to find a maximum weight packing of rooted subtrees of T, where the value of a subtree on node set S with root $i \in S$ is $\sum_{j \in S} c_{ij}$.