

type *lisgraph* of Section 5.4). This allows us to save time in the inner **for** loop, since we only have to consider those nodes w adjacent to v ; but how are we to avoid taking a time in $\Omega(n^2)$ to determine in succession the $n - 2$ values taken by v ?

The answer is to use an inverted heap containing one node for each element v of C , ordered by the value of $D[v]$. Thus the element v of C that minimizes $D[v]$ will always be found at the root. Initialization of the heap takes a time in $\Theta(n)$. The instruction " $C \leftarrow C \setminus \{v\}$ " consists of eliminating the root from the heap, which takes a time in $O(\log n)$. As for the inner **for** loop, it now consists of looking, for each element w of C adjacent to v , to see whether $D[v] + L[v, w]$ is less than $D[w]$. If so, we must modify $D[w]$ and percolate w up the heap, which again takes a time in $O(\log n)$. This does not happen more than once for each edge of the graph.

To sum up, we have to remove the root of the heap exactly $n - 2$ times and to percolate at most a nodes, giving a total time in $\Theta((a + n)\log n)$. If the graph is connected, $a \geq n - 1$, and the time is in $\Theta(a \log n)$. The straightforward implementation is therefore preferable if the graph is dense, whereas it is preferable to use a heap if the graph is sparse. If $a \in \Theta(n^2 / \log n)$, the choice of representation may depend on the specific implementation. Problem 6.16 suggests a way to speed up the algorithm by using a k -ary heap with a well-chosen value of k ; other, still faster algorithms are known; see Problem 6.17 and Section 6.8.

6.5 The knapsack problem (1)

This problem arises in various forms. In this chapter we look at the simplest version; a more difficult variant will be introduced in Section 8.4.

We are given n objects and a knapsack. For $i = 1, 2, \dots, n$, object i has a positive weight w_i and a positive value v_i . The knapsack can carry a weight not exceeding W . Our aim is to fill the knapsack in a way that maximizes the value of the included objects, while respecting the capacity constraint. In this first version of the problem we assume that the objects can be broken into smaller pieces, so we may decide to carry only a fraction x_i of object i , where $0 \leq x_i \leq 1$. (If we are not allowed to break objects, the problem is much harder.) In this case, object i contributes $x_i w_i$ to the total weight in the knapsack, and $x_i v_i$ to the value of the load. In symbols, the problem can be stated as follows:

$$\text{maximize } \sum_{i=1}^n x_i v_i \quad \text{subject to } \sum_{i=1}^n x_i w_i \leq W$$

where $v_i > 0$, $w_i > 0$ and $0 \leq x_i \leq 1$ for $1 \leq i \leq n$. Here the conditions on v_i and w_i are constraints on the instance; those on x_i are constraints on the solution. We shall use a greedy algorithm to solve the problem. In terms of our general schema, the candidates are the different objects, and a solution is a vector (x_1, \dots, x_n) telling us what fraction of each object to include. A feasible solution is one that respects the constraints given above, and the objective function is the total value of the objects in the knapsack. What we should take as the selection function remains to be seen.

If $\sum_{i=1}^n w_i \leq W$, it is clearly optimal to pack all the objects in the knapsack. We can therefore assume that in any interesting instance of the problem $\sum_{i=1}^n w_i > W$. It is also clear that an optimal solution must fill the knapsack exactly, for otherwise

we could add a fraction of one of the remaining objects and increase the value of the load. Thus in an optimal solution $\sum_{i=1}^n x_i w_i = W$. Since we are hoping to find a greedy algorithm that works, our general strategy will be to select each object in turn in some suitable order, to put as large a fraction as possible of the selected object into the knapsack, and to stop when the knapsack is full. Here is the algorithm.

```

✓ function knapsack( $w[1..n], v[1..n], W$ ): array  $[1..n]$ 
    {initialization}
    for  $i = 1$  to  $n$  do  $x[i] \leftarrow 0$ 
     $weight \leftarrow 0$ 
    {greedy loop}
    while  $weight < W$  do
         $i \leftarrow$  the best remaining object {see below}
        if  $weight + w[i] \leq W$  then  $x[i] \leftarrow 1$ 
             $weight \leftarrow weight + w[i]$ 
        else  $x[i] \leftarrow (W - weight) / w[i]$ 
             $weight \leftarrow W$ 

    return  $x$ 

```


There are at least three plausible selection functions for this problem: at each stage we might choose the most valuable remaining object, arguing that this increases the value of the load as quickly as possible; we might choose the lightest remaining object, on the grounds that this uses up capacity as slowly as possible; or we might avoid these extremes by choosing the object whose value per unit weight is as high as possible. Figures 6.5 and 6.6 show how these three different tactics work in one particular instance. Here we have five objects, and $W = 100$. If we select the objects in order of decreasing value, then we choose first object 3, then object 5, and finally we fill the knapsack with half of object 4. The value of the solution obtained in this way is $66 + 60 + 40/2 = 146$. If we select the objects in order of increasing weight, then we choose objects 1, 2, 3 and 4 in that order, and now the knapsack is full. The value of this solution is $20 + 30 + 66 + 40 = 156$. Finally if we select the objects in order of decreasing v_i/w_i , we choose first object 3, then object 1, next object 2, and finally we fill the knapsack with four-fifths of object 5. Using this tactic, the value of the solution is $20 + 30 + 66 + 0.8 \times 60 = 164$.

✓

	$n = 5, W = 100$				
w	10	20	30	40	50
v	20	30	66	40	60
v/w	2.0	1.5	2.2	1.0	1.2

Figure 6.5. An instance of the knapsack problem

This example shows that the solution obtained by a greedy algorithm that maximizes the value of each object it selects is not necessarily optimal, nor is the solution obtained by minimizing the weight of each object that is chosen. Fortunately the



Select:	x_i					Value
Max v_i	0	0	1	0.5	1	146
Min w_i	1	1	1	1	0	156
Max v_i/w_i	1	1	1	0	0.8	164

Figure 6.6. Three greedy approaches to the instance in Figure 6.5

following proof shows that the third possibility, selecting the object that maximizes the value per unit weight, does lead to an optimal solution.

✓ **Theorem 6.5.1** *If objects are selected in order of decreasing v_i/w_i , then algorithm knapsack finds an optimal solution.*

Proof Suppose without loss of generality that the available objects are numbered in order of decreasing value per unit weight, that is, that

$$v_1/w_1 \geq v_2/w_2 \geq \dots \geq v_n/w_n.$$

Let $X = (x_1, \dots, x_n)$ be the solution found by the greedy algorithm. If all the x_i are equal to 1, this solution is clearly optimal. Otherwise, let j be the smallest index such that $x_j < 1$. Looking at the way the algorithm works, it is clear that $x_i = 1$ when $i < j$, that $x_i = 0$ when $i > j$, and that $\sum_{i=1}^n x_i w_i = W$. Let the value of the solution X be $V(X) = \sum_{i=1}^n x_i v_i$.

Now let $Y = (y_1, \dots, y_n)$ be any feasible solution. Since Y is feasible, $\sum_{i=1}^n y_i w_i \leq W$, and hence $\sum_{i=1}^n (x_i - y_i) w_i \geq 0$. Let the value of the solution Y be $V(Y) = \sum_{i=1}^n y_i v_i$. Now

$$V(X) - V(Y) = \sum_{i=1}^n (x_i - y_i) v_i = \sum_{i=1}^n (x_i - y_i) w_i \frac{v_i}{w_i}$$

When $i < j$, $x_i = 1$ and so $x_i - y_i$ is positive or zero, while $v_i/w_i \geq v_j/w_j$; when $i > j$, $x_i = 0$ and so $x_i - y_i$ is negative or zero, while $v_i/w_i \leq v_j/w_j$; and of course when $i = j$, $v_i/w_i = v_j/w_j$. Thus in every case $(x_i - y_i)(v_i/w_i) \geq (x_i - y_i)(v_j/w_j)$. Hence

$$V(X) - V(Y) \geq (v_j/w_j) \sum_{i=1}^n (x_i - y_i) w_i \geq 0.$$

We have thus proved that no feasible solution can have a value greater than $V(X)$, so the solution X is optimal. ■

Implementation of the algorithm is straightforward. If the objects are already sorted into decreasing order of v_i/w_i , then the greedy loop clearly takes a time in $O(n)$; the total time including the sort is therefore in $O(n \log n)$. As in Section 6.3.1, it may be worthwhile to keep the objects in a heap with the largest value of v_i/w_i at the root. Creating the heap takes a time in $O(n)$, while each trip round the greedy loop now takes a time in $O(\log n)$ since the heap property must be restored after the root is removed. Although this does not alter the worst-case analysis, it may be faster if only a few objects are needed to fill the knapsack.