Chapter 3

Rounding data and dynamic programming

Dynamic programming is a standard technique in algorithm design in which an optimal solution for a problem is built up from optimal solutions for a number of subproblems, normally stored in a table or multidimensional array. Approximation algorithms can be designed using dynamic programming in a variety of ways, many of which involve rounding the input data in some way.

For instance, sometimes weakly NP-hard problems have dynamic programming algorithms that run in time polynomial in the input size if the input is represented in unary rather than in binary (so, for example, the number 7 would be encoded as 1111111). If so, we say that the algorithm is *pseudopolynomial*. Then by rounding the input values so that the number of distinct values is polynomial in the input size and an error parameter $\epsilon > 0$, this pseudopolynomial algorithm can be made to run in time polynomial in the size of the original instance. We can often show that the rounding does not sacrifice too much in the quality of the solution produced. We will use this technique in discussing the knapsack problem in Section 3.1.

For other problems, such as scheduling problems, we can often make distinctions between "large" and "small" parts of the input instance; for instance, in scheduling problems, we distinguish between jobs that have large and small processing times. We can then show that by rounding the sizes of the large inputs so that, again, the number of distinct, large input values is polynomial in the input size and an error parameter, we can use dynamic programming to find an optimal solution on just the large inputs. Then this solution must be augmented to a solution for the whole input by dealing with the small inputs in some way. Using these ideas, we will devise polynomial-time approximation schemes for the problem of scheduling parallel machines introduced in the last chapter, and for a new problem of packing bins.

3.1 The knapsack problem

A traveler with a knapsack comes across a treasure hoard. Unfortunately, his knapsack can hold only so much. What items should he place in his knapsack in order to maximize the value of the items he takes away? This unrealistic scenario gives the name to the *knapsack problem*. In the knapsack problem, we are given a set of n items $I = \{1, \ldots, n\}$, where each item i has a value v_i and a size s_i. All sizes and values are positive integers. The knapsack has capacity B, where B

```
A(1) ← {(0, 0), (s_1, w_1)}
for j ← 2 to n do
    A(j) ← A(j − 1)
    for each (t, w) ∈ A(j − 1) do
        if t + s_j ≤ B then
            Add (t + s_j, w + v_j) to A(j)
    Remove dominated pairs from A(j)
return max_{(t,w)∈A(n)} w
```

Algorithm 3.1: A dynamic programming algorithm for the knapsack problem.

is also a positive integer. The goal is to find a subset of items $S \subseteq I$ that maximizes the value $\sum_{i \in S} v_i$ of items in the knapsack subject to the constraint that the total size of these items is no more than the capacity; that is, $\sum_{i \in S} s_i \leq B$. We assume that we consider only items that could actually fit in the knapsack (by themselves), so that $s_i \leq B$ for each $i \in I$. Although the application stated above is unlikely to be useful in real life, the knapsack problem is well studied because it is a simplified model of a problem that arises in many realistic scenarios.

We now argue that we can use dynamic programming to find the optimal solution to the knapsack problem. We maintain an array entry $A(j)$ for $j = 1, \ldots, n$. Each entry $A(j)$ is a list of pairs (t, w). A pair (t, w) in the list of entry $A(j)$ indicates that there is a set S from the first j items that uses space exactly $t \leq B$ and has value exactly w; that is, there exists a set $S \subseteq \{1, \ldots, j\}$, such that $\sum_{i \in S} s_i = t \leq B$ and $\sum_{i \in S} v_i = w$. Each list does not contain all possible such pairs, but instead keeps track of only the most efficient ones. To do this, we introduce the notion of one pair *dominating* another one; a pair (t, w) dominates another pair (t', w') if $t \leq t'$ and $w \geq w'$; that is, the solution indicated by the pair (t, w) uses no more space than (t', w'), but has at least as much value. Note that domination is a transitive property; that is, if (t, w) dominates (t', w') and (t', w') dominates (t'', w''), then (t, w) also dominates (t'', w''). We will ensure that in any list, no pair dominates another one; this means that we can assume each list $A(j)$ is of the form $(t_1, w_1), \ldots, (t_k, w_k)$ with $t_1 < t_2 < \cdots < t_k$ and $w_1 < w_2 < \cdots < w_k$. Since the sizes of the items are integers, this implies that there are at most $B + 1$ pairs in each list. Furthermore, if we let $V = \sum_{i=1}^{n} v_i$ be the maximum possible value for the knapsack, then there can be at most $V + 1$ pairs in the list. Finally, we ensure that for each feasible set $S \subseteq \{1, \ldots, j\}$ (with $\sum_{i \in S} s_i \leq B$), the list $A(j)$ contains some pair (t, w) that dominates $(\sum_{i \in S} s_i, \sum_{i \in S} v_i)$.

In Algorithm 3.1, we give the dynamic program that constructs the lists $A(j)$ and solves the knapsack problem. We start out with $A(1) = \{(0, 0), (s_1, w_1)\}$. For each $j = 2, \ldots, n$, we do the following. We first set $A(j) \leftarrow A(j-1)$, and for each $(t, w) \in A(j-1)$, we also add the pair $(t + s_j, w + v_j)$ to the list $A(j)$ if $t + s_j \leq B$. We finally remove from $A(j)$ all dominated pairs by sorting the list with respect to their space component, retaining the best value for each space total possible, and removing any larger space total that does not have a corresponding larger value. One way to view this process is to generate two lists, $A(j-1)$ and the one augmented by (s_j, w_j), and then perform a type of merging of these two lists. We return the pair (t, w) from $A(n)$ of maximum value as our solution. Next we argue that this algorithm is correct.

Theorem 3.1: *Algorithm 3.1 correctly computes the optimal value of the knapsack problem.*

Proof. By induction on j we prove that $A(j)$ contains all non-dominated pairs corresponding to feasible sets $S \subseteq \{1, \ldots, j\}$. Certainly this is true in the base case by setting $A(1)$ to

$\{(0,0),(s_1,w_1)\}$. Now suppose it is true for $A(j-1)$. Let $S \subseteq \{1,\ldots,j\}$, and let $t = \sum_{i\in S} s_i \leq B$ and $w = \sum_{i\in S} v_i$. We claim that there is some pair $(t',w') \in A(j)$ such that $t' \leq t$ and $w' \geq w$. First, suppose that $j \notin S$. Then the claim follows by the induction hypothesis and by the fact that we initially set $A(j)$ to $A(j-1)$ and removed dominated pairs. Now suppose $j \in S$. Then for $S' = S - \{j\}$, by the induction hypothesis, there is some $(\hat{t},\hat{w}) \in A(j-1)$ that dominates $(\sum_{i\in S'} s_i, \sum_{i\in S'} v_i)$, so that $\hat{t} \leq \sum_{i\in S'} s_i$ and $\hat{w} \geq \sum_{i\in S'} v_i$. Then the algorithm will add the pair $(\hat{t}+s_j, \hat{w}+v_j)$ to $A(j)$, where $\hat{t}+s_j \leq t \leq B$ and $\hat{w}+v_j \geq w$. Thus, there will be some pair $(t',w') \in A(j)$ that dominates (t,w). □

Algorithm 3.1 takes $O(n\min(B,V))$ time. This is not a polynomial-time algorithm, since we assume that all input numbers are encoded in binary; thus, the size of the input number B is essentially $\log_2 B$, and so the running time $O(nB)$ is exponential in the size of the input number B, not polynomial. If we were to assume that the input is given in unary, then $O(nB)$ would be a polynomial in the size of the input. It is sometimes useful to make this distinction between problems.

Definition 3.2: *An algorithm for a problem* Π *is said to be* pseudopolynomial *if its running time is polynomial in the size of the input when the numeric part of the input is encoded in unary.*

If the maximum possible value V were some polynomial in n, then the running time would indeed be a polynomial in the input size. We now show how to get a polynomial-time approximation scheme for the knapsack problem by rounding the values of the items so that V is indeed a polynomial in n. The rounding induces some loss of precision in the value of a solution, but we will show that this does not affect the final value by too much. Recall the definition of an approximation scheme from Chapter 1.

Definition 3.3: *A* polynomial-time approximation scheme (PTAS) *is a family of algorithms* $\{A_\epsilon\}$*, where there is an algorithm for each* $\epsilon > 0$*, such that* A_ϵ *is a* $(1+\epsilon)$*-approximation algorithm (for minimization problems) or a* $(1-\epsilon)$*-approximation algorithm (for maximization problems).*

Note that the running time of the algorithm A_ϵ is allowed to depend arbitrarily on $1/\epsilon$: this dependence could be exponential in $1/\epsilon$, or worse. We often focus attention on algorithms for which we can give a good bound of the dependence of the running time of A_ϵ on $1/\epsilon$. This motivates the following definition.

Definition 3.4: *A* fully polynomial-time approximation scheme (FPAS, FPTAS) *is an approximation scheme such that the running time of* A_ϵ *is bounded by a polynomial in* $1/\epsilon$.

We can now give a fully polynomial-time approximation scheme for the knapsack problem. Suppose that we measure value in (integer) multiples of μ (where we shall set μ below), and convert each value v_i by rounding down to the nearest integer multiple of μ; more precisely, we set v_i' to be $\lfloor v_i/\mu \rfloor$ for each item i. We can then run the dynamic programming algorithm of Figure 3.1 on the items with sizes s_i and values v_i', and output the optimal solution for the rounded data as a near-optimal solution for the true data. The main idea here is that we wish to show that the accuracy we lose in rounding is not so great, and yet the rounding enables us to have the algorithm run in polynomial time. Let us first do a rough estimate; if we used values $\tilde{v}_i = v_i'\mu$ instead of v_i, then each value is inaccurate by at most μ, and so each feasible solution has its value changed by at most $n\mu$. We want the error introduced to be at most ϵ times a lower bound on the optimal value (and so be sure that the true relative error is at most ϵ). Let M be the maximum value of an item; that is, $M = \max_{i\in I} v_i$. Then M is a lower bound

$M \leftarrow \max_{i \in I} v_i$
$\mu \leftarrow \epsilon M/n$
$v'_i \leftarrow \lfloor v_i/\mu \rfloor$ for all $i \in I$
Run Algorithm 3.1 for knapsack instance with values v'_i

Algorithm 3.2: An approximation scheme for the knapsack problem.

on OPT, since one possible solution is to pack the most valuable item in the knapsack by itself. Thus, it makes sense to set μ so that $n\mu = \epsilon M$ or, in other words, to set $\mu = \epsilon M/n$.

Note that with the modified values, $V' = \sum_{i=1}^{n} v'_i = \sum_{i=1}^{n} \lfloor \frac{v_i}{\epsilon M/n} \rfloor = O(n^2/\epsilon)$. Thus, the running time of the algorithm is $O(n \min(B, V')) = O(n^3/\epsilon)$ and is bounded by a polynomial in $1/\epsilon$. We can now prove that the algorithm returns a solution whose value is at least $(1 - \epsilon)$ times the value of an optimal solution.

Theorem 3.5: *Algorithm 3.2 is a fully polynomial-time approximation scheme for the knapsack problem.*

Proof. We need to show that the algorithm returns a solution whose value is at least $(1 - \epsilon)$ times the value of an optimal solution. Let S be the set of items returned by the algorithm. Let O be an optimal set of items. Certainly $M \leq \mathrm{OPT}$, since one possible solution is to put the most valuable item in a knapsack by itself. Furthermore, by the definition of v'_i, $\mu v'_i \leq v_i \leq \mu(v'_i + 1)$, so that $\mu v'_i \geq v_i - \mu$. Applying the definitions of the rounded data, along with the fact that S is an optimal solution for the values v'_i, we can derive the following chain of inequalities:

$$\begin{aligned}
\sum_{i \in S} v_i &\geq \mu \sum_{i \in S} v'_i \\
&\geq \mu \sum_{i \in O} v'_i \\
&\geq \sum_{i \in O} v_i - |O|\mu \\
&\geq \sum_{i \in O} v_i - n\mu \\
&= \sum_{i \in O} v_i - \epsilon M \\
&\geq \mathrm{OPT} - \epsilon\,\mathrm{OPT} = (1 - \epsilon)\,\mathrm{OPT}\,.
\end{aligned}$$

□

3.2 Scheduling jobs on identical parallel machines

We return to the problem of scheduling a collection of n jobs on m identical parallel machines; in Section 2.3 we presented a result that by first sorting the jobs in order of non-increasing processing requirement, and then using a list scheduling rule, we find a schedule of length guaranteed to be at most 4/3 times the optimum. In this section, we will show that this result contains the seeds of a polynomial-time approximation scheme: for any given value of $\rho > 1$, we give an algorithm that runs in polynomial time and finds a solution of objective function value at most ρ times the optimal value.