# 25 How to solve difficult problems?

*I think computer science, by and large, is still stuck in the Modern age.*
– Larry Wall

*Progress is possible only if we train ourselves to think about programs without thinking of them as pieces of executable code.*
– Edsger Wybe Dijkstra

Thanks to the concept of NP-completeness and the Cook-Levin theorem, we can identify certain computational problems as NP-complete. Unless P = NP, these problems turn out to be difficult to solve in feasible time, and may be termed intractable. In reality, however, we may actually encounter such intractable problems. Trying to prove P = NP (or otherwise) does not seem to be a practical approach to *solve* the intractable problems. Fortunately, several algorithmic paradigms are available to deal with computationally difficult problems. In this chapter, we deal with three such paradigms.

*Approximation algorithms* typically apply to optimization problems and compute sub-optimal solutions with optimality achieved within guaranteed factors. The basic idea is that if computing the best solution is infeasible, we have to be satisfied with a solution that is nearly as good as the best, but that can be computed in feasible (that is, polynomial) time.

*Randomized algorithms* make random guesses at certain steps, and constitute a natural way of *implementing* non-determinism. However, unlike a non-deterministic algorithm, a randomized algorithm takes a unique random path of computation during each execution. The random choices are now governed by pure chance (instead of certificates). If a significant fraction of the computation paths end up in the answer *Yes*, then with a high probability we can actually reach that answer within a few (polynomial in the input size) iterations of the algorithm. On the other hand, if every such attempt arrives at the answer *No*, we conclude (with some probability of error) that the answer is *No*. In this case, the algorithm takes a polynomial running time, but may occasionally produce a wrong answer. There is another variant of randomized algorithms, in which the answer is always correct, but the running time is *expected* to be polynomial-time. In rare situations (that is, situations with very low probability), the algorithm may take as bad as exponential running time.

*Backtracking* is yet another way to traverse the non-deterministic computation tree for a problem. In the first place, this strategy removes the need for explicitly computing the entire tree (which may be exponentially large in size). Second, using some heuristic ideas, we may often abort exploration along certain paths. This way, we make the search more efficient than an exhaustive search in the tree. *Branch-and-bound algorithms* are backtracking algorithms suited to optimization problems.

Although our basic motivation for learning these classes of algorithms is to solve computationally difficult problems, there are other situations too where these algorithms prove to be useful. For example, the best-known exact or deterministic algorithm for a problem may be polynomial-time, but can be replaced, in practice, by much faster approximate or randomized algorithms. On the other extreme, there are certain problems for which none of the above strategies seem to offer

efficient solutions. As we have already seen throughout this book, everything cannot fit in only a handful of ready-made formulas.

## 25.1 Approximation algorithms

Approximation algorithms pertain to optimization problems, whereas we have, so far, categorized only decision problems as easy or difficult. In order that optimization problems too make sense in this setting, let me first formalize the notion of an optimization problem.

Let $P$ be an optimization problem, and $\mathscr{O}_I$ the set of possible output instances on an input $I$. To every output instance $O \in \mathscr{O}_I$, we associate a real number $f(O)$. We call $f$ the *objective function*. Our goal is to identify that particular output $O^*$, for which the value $f(O^*)$ is as small (if $P$ is a minimization problem) or large (if $P$ is a maximization problem) as possible. If several output instances correspond to the optimum value, identifying any one of these output instances suffices. In order that $P$ qualifies as an optimization problem in NP, the following requirements must be met. Here, the term *easy* refers to polynomial running times in the size $n$ of the input $I$.

- It is easy to test the membership $O \in \mathscr{O}_I$.
- It is easy to compute $f(O)$ for every $O \in \mathscr{O}_I$.

The optimum value $f(O^*)$ (a real number) is denoted by $\mathrm{OPT}_I$ (or simply by OPT if $I$ is understood from the context).

An optimization problem is often rephrased in terms of a decision problem by specifying an input $I$ and a real bound $B$. We then ask the question whether there exists an $O \in \mathscr{O}_I$ such that $f(O) \leqslant B$ (minimization problem) or $f(O) \geqslant B$ (maximization problem). It is easy to argue that for appropriate choices of the bound $B$, the decision problem is solvable in polynomial time if and only if the optimization problem is solvable in polynomial time. Moreover, the decision problem is in the class NP if and only if the optimization problem is in NP under the modified definition.

An approximation algorithm $A$ for an optimization problem $P$ in NP is an algorithm that, given an input instance $I$, produces an output $O \in \mathscr{O}_I$ with the guarantee that $f(O) \leqslant \rho \times \mathrm{OPT}_I$ (if $P$ is a minimization problem) or $f(O) \geqslant \rho \times \mathrm{OPT}_I$ (if $P$ is a maximization problem), where $\rho$ is a real number ($\rho > 1$ for minimization problems, and $0 < \rho < 1$ for maximization problems). For the algorithm $A$ to be useful, it is required that the running time of $A$ be polynomial in the size $n$ of $I$. The factor $\rho$ indicates how close the approximation is to the optimum value. Values of $\rho$ closer to 1 imply better approximation. We call $A$ a *$\rho$-approximation algorithm*.

It would be worthwhile to investigate how we can actually conclude anything about the approximation factor $\rho$ if it is infeasible to compute the optimum value $\mathrm{OPT}_I$ itself. Of course, it makes little sense to compute $\rho$ as the ratio of $f(O)$ and $\mathrm{OPT}_I$. We instead resort to indirect (non-constructive) mathematical proofs in order to claim about approximation factors. An approximation factor $\rho$ is called *tight* if there exist input instances on which $A$ actually produces outputs $O$ with $f(O) = \rho \times \mathrm{OPT}_I$.

Approximation algorithms are often classified in accordance with the approximation factor $\rho$ and the running time (which, in addition to the input size $n$, may depend on $\rho$). Before proceeding to such classifications, let me supply some examples.

### 25.1.1 The minimum vertex cover problem

Let $G = (V, E)$ be an undirected graph. A *vertex cover* for $G$ is a subset $U \subseteq V$ such that every edge $e \in E$ has at least one endpoint in $U$. The problem of minimizing the size $|U|$ (over all vertex covers $U$ of $G$) is called the *minimum vertex cover problem*, denoted MIN-VERTEX-COVER. It is easy to verify that MIN-VERTEX-COVER is a minimization problem in NP. Two approximation algorithms to solve this problem are now described. The first one, based upon a simple greedy strategy, yields logarithmic approximation factors. The second one, based upon matching in graphs, is a 2-approximation algorithm for MIN-VERTEX-COVER. For both these algorithms, the approximation factors turn out to be tight.

#### 25.1.1.1 A logarithmic approximation algorithm for MIN-VERTEX-COVER

An obvious greedy strategy is to select vertices covering the maximum numbers of (remaining) edges, as described now.

```
Initialize U = ∅.
while (E is not empty) {
    Find a vertex u ∈ V of largest (remaining) degree.
    Add u to U.
    Delete from E all the (remaining) edges with u as one endpoint.
}
Return U.
```

This greedy algorithm computes a vertex cover of $G$ in time polynomial in the size $|V| + |E|$ of the graph $G$. Let $k$ be the size of the vertex cover $U$ output by this algorithm. Suppose that the vertices $u_1, u_2, \ldots, u_k$ are selected (in that order) for inclusion in $U$. On the other hand, suppose that $U^*$ is a minimum vertex cover for $G$. We need to compute an upper bound on the ratio $|U|/|U^*|$.

Let $t = |U^*|$. We have $|U| = k$, so $\rho = k/t$. Let $G_0 = G$, and for $1 \leqslant i \leqslant k$, $G_i = (V, E_i)$ the graph after the edges incident upon $u_1, u_2, \ldots, u_i$ are removed. Finally, let $m_i = |E_i|$. We analyze the removal of $u_{i+1}$ from $G_i$ for $0 \leqslant i < k$. Suppose that $u_1, u_2, \ldots, u_i$ contain $t_i$ of the $t$ vertices of $U^*$. The remaining $t - t_i$ vertices of $U^*$ constitute a vertex cover of $G_i$. Therefore there must exist a vertex $v_{i+1} \in U^* \setminus \{u_1, u_2, \ldots, u_i\}$ such that the degree of $v_{i+1}$ in $G_i$ is at least $m_i/(t - t_i)$. The greedy algorithm chooses a vertex $u_{i+1}$ whose degree in $G_i$ is no smaller than the degree of $v_{i+1}$. After the edges incident upon $u_{i+1}$ are removed, the number of edges of the resulting graph $G_{i+1}$ must therefore satisfy

$$m_{i+1} \leqslant m_i \left(1 - \frac{1}{t - t_i}\right) \leqslant m_i \left(1 - \frac{1}{t}\right).$$

From this observation, it follows that

$$m_i \leqslant m \left(1 - \frac{1}{t}\right)^i$$

for all $i$, where $m = m_0 = |E|$. For $i = t \ln m$, we have

$$m_i \leqslant m \left(1 - \frac{1}{t}\right)^{t \ln m} < m \left(e^{-1}\right)^{\ln m} = 1.$$

Therefore $k \leqslant t \ln m$, that is, $\rho = k/t \leqslant \ln m$. Finally, since $m = \mathrm{O}(n^2)$, $\rho$ is at most $\Theta(\log n)$.

Figure 116: The logarithmic approximation factor for the greedy vertex cover algorithm is optimal

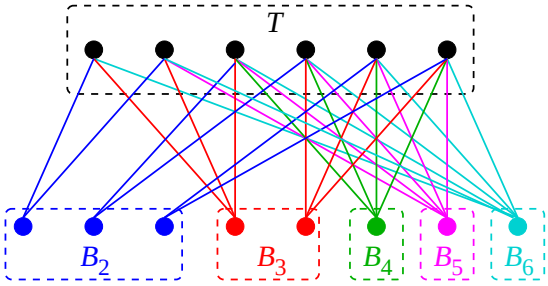



Figure 116 illustrates that this logarithmic bound is tight (up to constant multiplication factors). The figure shows a bipartite graph $G = (V, E)$ with the vertex set $V$ partitioned into $T$ (the top vertices) and $B$ (the bottom vertices). $T$ consists of exactly $t$ vertices ($t = 6$ in the figure), whereas $B$ is partitioned into $t-1$ groups $B_2, B_3, \ldots, B_t$, where the $i$-th group $B_i$ contains exactly $\lfloor t/i \rfloor$ vertices. For any $i$, each of the vertices in $B_i$ is adjacent to exactly $i$ vertices in $T$, and no two vertices of $B_i$ share a neighbor in $T$.

Let us see how the greedy algorithm works on this graph. Initially, the degree of each vertex in $B_i$ is $i$, whereas the degree of each vertex in $T$ is $\leqslant t-1$. In other words, the vertex with the largest degree is the only vertex in $B_t$. So this vertex is added to the greedy cover, and all the edges incident on this vertex are removed. This leaves a graph in which every vertex of $T$ has remaining degree $\leqslant t-2$. But any vertex in $B_{t-1}$ already has degree $t-1$. ($B_{t-1}$ contains only one vertex if $t \geqslant 3$.) This vertex is then added to the greedy cover, and all the edges incident on it are removed. Proceeding in this fashion, the algorithm outputs the set $B$ of all bottom vertices as the greedy vertex cover. Let us now estimate the size of $B$. First, I derive an upper bound:

$$|B| = \sum_{i=2}^{t} \left\lfloor \frac{t}{i} \right\rfloor \leqslant \sum_{i=2}^{t} \frac{t}{i} = t(H_t - 1) \leqslant t \ln t.$$

Then, I compute a lower bound:

$$\begin{aligned} |B| &= \sum_{i=2}^{t} \left\lfloor \frac{t}{i} \right\rfloor \geqslant \sum_{i=2}^{t} \frac{t-(i-1)}{i} = (t+1)\left(\sum_{i=2}^{t} \frac{1}{i}\right) - (t-1) \\ &\geqslant (t-1)(H_t - 2) \geqslant (t-1)(\ln(t+1) - 2). \end{aligned}$$

It follows that the greedy vertex cover has size $|U| = |B| = \Theta(t \log t)$. On the other hand, the set $T$ itself is a vertex cover for $G$, and $|T| = t$. Let $U^*$ be a minimum vertex cover of $G$. It follows that

$$|U^*| \leqslant |T| = \frac{1}{\Theta(\log t)} |U|.$$

Since $n = |V| = |B| + |T| = \Theta(t \log t)$, we have $\log t = \Theta(\log n)$, so

$$\rho = \frac{|U|}{|U^*|} \geqslant \Theta(\log n).$$

### 25.1.1.2 A 2-approximation algorithm for MIN-VERTEX-COVER

The greedy vertex cover algorithm discussed in the last section gives an approximation guarantee which degrades logarithmically with the increase in the input size. This is not a good achievement. We are now going to study another approximation algorithm for MIN-VERTEX-COVER, which guarantees an approximation factor of 2 irrespective of the input size.

A *matching* in an undirected graph $G = (V,E)$ is a subset $D \subseteq E$ such that no two edges of $D$ share an endpoint. Any matching $D$ in $G$ is related to any vertex cover $U$ of $G$ as

$$|D| \leqslant |U|.$$

For proving this inequality, notice that $U$ must contain at least one endpoint of every edge in $D$. The following approximation algorithm is based upon this observation.

```
Initialize U = ∅.
while (E is not empty) {
    Pick any edge e = (u,v) from E.
    Add u and v to U.
    Remove u and v from V.
    Remove from E all edges incident on u or v.
}
Return U.
```

Clearly, this algorithm runs in time polynomial in the size of $G$. The set $U$ returned by this algorithm is a vertex cover for $G$. Let $D$ denote the set of edges picked in all the iterations of the `while` loop. The algorithm guarantees that $D$ is a matching in $G$. Moreover, $U$ consists precisely of both the endpoints of all the edges in $D$, that is,

$$|U| = 2|D|.$$

If $U^*$ is a minimum vertex cover of $G$, we have

$$|D| \leqslant |U^*|.$$

Combining these two relations yields

$$|U| \leqslant 2|U^*|.$$

Figure 117: Illustration of the matching-based minimum vertex cover algorithm

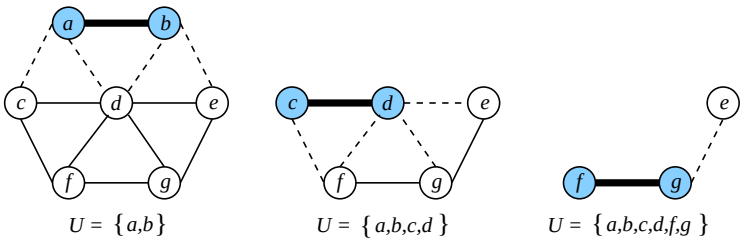



Figure 117 illustrates the working of the above 2-approximation algorithm. The bold edges are chosen in three iterations to form the matching $D$. The shaded vertices are removed along with the bold edges. The dotted edges too get removed in the process. The resulting vertex cover $U$ consists

of the six vertices $a,b,c,d,f,g$. It is easy to argue that an optimal vertex cover for this graph is $U^* = \{a,d,e,f\}$. For this example, we have $\rho = |U|/|U^*| = 3/2$.

It turns out that the approximation factor 2 is tight for this algorithm. For an illustration, consider the complete bipartite graph $K_{n,n}$ which consists of two independent vertex sets $V_1, V_2$ of size $n$ each, with an edge connecting every vertex of $V_1$ with every vertex of $V_2$. It is easy to argue that in this case, the approximation algorithm outputs the entire vertex set $V_1 \cup V_2$ of size $2n$, whereas a minimum vertex cover (like $V_1$) has size $n$.

### 25.1.2 The traveling salesperson problem

Let $G = (V,E)$ be the complete undirected graph on a vertex set $V$. Each member of $V$ represents a city. For each pair $(u,v)$ of cities, there is a *positive* cost $c(u,v)$ of travel between $u$ and $v$. Since $G$ is undirected, we assume that the cost function is symmetric about its two arguments, that is, $c(u,v) = c(v,u)$ for every two cities $u,v$. The traveling salesperson problem (TSP) is the determination of a Hamiltonian cycle $Z$ in $G$ for which the sum $c(Z)$ of all the edge costs on $Z$ is as small as possible. This is an optimization problem in NP. In Section 24.3.3, we have studied a decision version of this problem.

#### 25.1.2.1 The Euclidean traveling salesperson problem

As a special variant of the TSP, let us consider the situation where all the cities are assumed to lie on a plane and the inter-city cost $c(u,v)$ is taken as the Euclidean distance $d(u,v)$ between $u$ and $v$. This variant, denoted as EUCLIDEAN-TSP, has a 2-approximation algorithm described now.

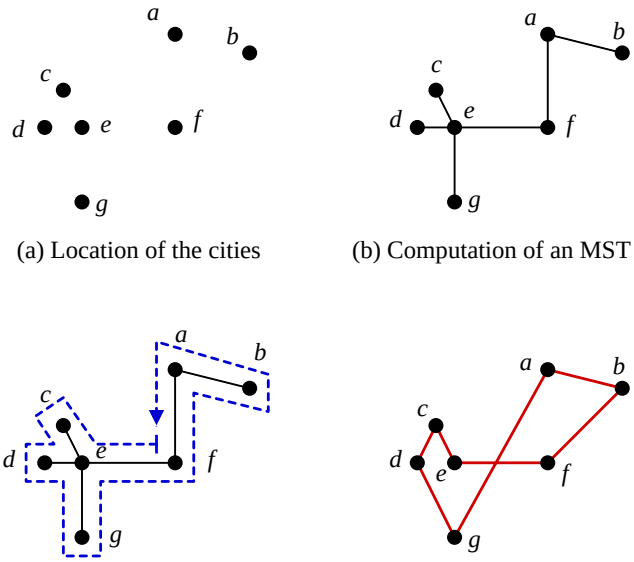
```
Compute a minimum spanning tree T of G under the given cost function.
Choose an arbitrary vertex u_0 of T.
Treat T as a tree rooted at u_0.
Impose an arbitrary ordering on the children of each node.
Make a pre-order traversal of T (starting at the root u_0).
Suppose that the traversal returns the list u_0,u_1,u_2,...,u_{n-1} of visited nodes.
Return the Hamiltonian cycle Z = (u_0,u_1,u_2,...,u_{n-1},u_0).
```

The working of this algorithm is demonstrated in Figure 118. Start with seven cities $a$–$g$. An MST $T$ for these cities is computed in Part (b) of the figure. Let us choose the vertex $f$ as the root, and order the children of $f$ as $e,a$, and those of $e$ as $c,d,g$. The resulting tree is now traversed in a pre-order fashion, as illustrated in Part (c). This traversal actually makes a closed walk $W = f,e,c,e,d,e,g,e,f,a,b,a,f$ in the graph, in which only the edges of $T$ are used, with every edge of $T$ visited exactly twice, once in each direction. If we remove the duplicate listing of visited vertices from this walk, we get the list $f,e,c,d,g,a,b$. The output Hamiltonian cycle is, therefore, $Z = (f,e,c,d,g,a,b,f)$, as shown in Part (d).

The proof that this is a 2-approximation algorithm for EUCLIDEAN-TSP is straightforward. Let $Z^*$ be an optimal Hamiltonian cycle in $G$. The removal of any edge from $Z^*$ leaves a Hamiltonian path in $G$, which is also a spanning tree of $G$. Since $T$ is a minimum spanning tree, and since every edge cost is positive, we have

$$c(T) \leqslant c(Z^*).$$

Figure 118: Illustration of the approximate EUCLIDEAN-TSP algorithm




(a) Location of the cities (b) Computation of an MST

(c) Preorder traversal of MST (d) The TSP cycle

Now, consider the closed walk $W$ in $G$ made by the pre-order traversal (as illustrated in Part (c) of Figure 118). We have

$$c(W) = 2c(T).$$

Combining these two relations gives

$$c(W) \leqslant 2c(Z^*).$$

Now, we make use of the fact that the cost function on the edge set of $G$ is a distance function in the two-dimensional plane. The output Hamiltonian cycle $Z$ is obtained by removing duplicate vertices from $W$. When a vertex $v$ is removed from $W$, we change the listing from $\ldots, u, v, w, \ldots$ to $\ldots, u, w, \ldots$. This stands for replacing the travel from $u$ to $v$ followed by the travel from $v$ to $w$ by the direct travel from $u$ to $w$. By the triangle inequality of distances, we have $c(u,w) \leqslant c(u,v) + c(v,w)$, that is, the removal of duplicate vertices cannot increase the cost of the walk. To sum up, we have the desired relation

$$c(Z) \leqslant c(W) \leqslant 2c(Z^*).$$

It is trivial to argue that the approximation algorithm runs in polynomial time in the number $n$ of cities. Even if the input size includes the representation complexity of the inter-city costs, the running time remains polynomial.

Better approximation algorithms exist for EUCLIDEAN-TSP. Indeed, for any real constant $\varepsilon > 0$, there exist $(1+\varepsilon)$-approximation algorithms with running times polynomial in $n$. We will not discuss these algorithms in this book.

#### 25.1.2.2 The general traveling salesperson problem

It is clear from the analysis of the 2-approximation algorithm for EUCLIDEAN-TSP that the only property we need for proving its approximation guarantee is the triangle inequality. Any cost function (not necessarily arising out of a distance metric) which satisfies this inequality enjoys the same 2-approximation algorithm.

The general version of the TSP, on the other hand, turns out to be an optimization problem in NP, which is very resistant to approximation possibilities. More precisely, I now show that, for any constant $\rho > 1$, the existence of a $\rho$-approximation algorithm for TSP implies P = NP (which is popularly believed to be false). To that effect, let us assume that there does exist such an algorithm $A$. Using this (hypothetical) algorithm, we can solve the NP-Complete problem HAM-CYCLE in polynomial time, as follows.

Let $G = (V, E)$ be an input instance for HAM-CYCLE. We convert this to the complete graph $G' = (V, E')$ on the same vertex set $V$. We assign the following cost to each pair $(u, v)$ of vertices in $V$, where $n = |V|$, and $\rho'$ is any real number larger than $\rho$.

$$c(u,v) = \begin{cases} 1/n & \text{if } (u,v) \in E, \\ \rho' & \text{otherwise.} \end{cases}$$

We feed the converted graph $G'$ to the hypothetical algorithm $A$. Since $A$ is a polynomial-time algorithm, it outputs a suboptimal Hamiltonian cycle $Z$ in $G'$. I claim that the original graph $G$ contains a Hamiltonian cycle if and only if $c(Z) = 1$.

Suppose that $G$ contains a Hamiltonian cycle. Then, any optimal Hamiltonian cycle $Z^*$ in $G'$ has a cost $c(Z^*) = n \times \frac{1}{n} = 1$. On the contrary, any non-optimal Hamiltonian cycle $Z$ in $G'$ uses at least one edge of cost $\rho'$, implying that $c(Z) \geqslant \rho' > \rho$. Since $A$ is a $\rho$-approximation algorithm, it must return a Hamiltonian cycle of cost $\leqslant \rho \times c(Z^*) = \rho$. This indicates that $A$ must return an optimal solution with cost 1.

Conversely, if $G$ does not contain a Hamiltonian cycle, then any Hamiltonian cycle $Z$ in $G'$ has a cost $c(Z) \geqslant \rho' > \rho > 1$. In particular, the cycle returned by $A$ will be of cost larger than 1.

### 25.1.3 Polynomial-time approximation schemes

It is interesting to ask whether we can make the approximation ratio $\rho$ as close to 1 as possible. In other words, we attempt to achieve an approximation ratio $\rho = 1 \pm \varepsilon$ (where the sign distinguishes minimization problems from maximization problems) with $\varepsilon$ as small as we desire (but always keeping $\varepsilon > 0$). It turns out that such approximation ratios are achievable for certain optimization problems in NP.

Usually, such good approximation ratios do not come for free. As we reduce $\varepsilon$, the running time of the approximation algorithm is expected to increase. Consequently, it is worthwhile to express the running time of the approximation algorithm as a function of both $n$ (the input size) and $1/\varepsilon$ (the goodness of approximation). So long as $\varepsilon$ is restricted to constant values only, the dependence of the running time on $1/\varepsilon$ may be of little concern. On the other hand, if we allow polynomial expressions in $n$ as $1/\varepsilon$, the dependence of the running time on $\varepsilon$ becomes crucial. For example, an approximation algorithm of running time $\Theta(n^{1/\varepsilon})$ is a polynomial-time algorithm for any positive constant $\varepsilon$ (only the degree of the polynomial varies with $\varepsilon$). However, if we plan to take $1/\varepsilon = n$

(or $\log n$), then the running time no longer remains polynomial. On the other hand, a running time like $O(n^2/\varepsilon)$ continues to remain polynomial for any polynomial expression of $n$ used as $1/\varepsilon$.

This motivates us to define two particular types of approximation algorithms.

**Definition** Let $A$ be an $(1 \pm \varepsilon)$-approximation algorithm for an optimization problem. We call $A$ a *polynomial-time approximation scheme* (*PTAS*) if the running time of $A$ is a polynomial in $n$ (the input size). We call $A$ a *fully polynomial-time approximation scheme* (*FPTAS*) if the running time of $A$ is a polynomial in both $n$ and $1/\varepsilon$.

In practice, one uses a PTAS/FPTAS as a tradeoff between the running time and the approximation quality. Stated differently, one fixes an approximation ratio $\rho = 1 \pm \varepsilon$ depending upon the maximum time one can spend for solving the problem on the input sizes $n$ of practical concern.

### 25.1.4 An FPTAS for the knapsack problem

Suppose that we have $n$ objects $O_1, O_2, \ldots, O_n$. The $i$-th object $O_i$ has weight $w_i$ and value (profit) $p_i$. In order to simplify matters, let us restrict our attention to the case that each $w_i$ and each $p_i$ are positive integers. Finally, suppose that there is a knapsack of (weight) capacity $C$. Our task is to pack a subcollection $O_{i_1}, O_{i_2}, \ldots, O_{i_m}$ of the given objects in the knapsack so as to maximize the profit $p_{i_1} + p_{i_2} + \cdots + p_{i_m}$ of packed objects. The constraint is, of course, that the sum of the weights of the chosen objects must not exceed the knapsack capacity, that is, $w_{i_1} + w_{i_2} + \cdots + w_{i_m} \leqslant C$.

An obvious greedy strategy to solve the knapsack problem is to keep on packing objects in the order of decreasing profit values until the knapsack capacity is exceeded. This algorithm, however, produces solutions that can be made as bad as one desires. For example, take $C + 1$ objects. The first one is of weight $C$ and profit 2, whereas each of the remaining objects is of weight 1 and profit 1. The greedy algorithm chooses the first object that alone fills the knapsack to its capacity, so the resulting profit is 2. On the other hand, we can pack all the given objects except the first in the knapsack achieving a profit of $C$. Therefore, the greedy algorithm produces a solution which is $C/2$ times as bad as the optimal solution.

A (presumably) better greedy strategy is to pack the objects in the decreasing order of their profit/weight ratios. Although this new strategy solves the knapsack problem optimally for the instance described in the last paragraph, it can also produce solutions as bad as one desires. For example, suppose that there are only two objects. The first one is of weight 1 and profit 2 (so the profit/weight ratio is 2), whereas the second one is of weight $C$ and profit $C$ (the profit/weight ratio is 1). Our second greedy algorithm packs the first object in the knapsack leaving a capacity not large enough to accommodate the second. The total profit is thus 2. On the other hand, we could put only the second object in the knapsack, achieving a profit of $C$. In this case too, the greedy solution is $C/2$ times poorer than the optimal solution. (Note that, for this input instance, the first greedy algorithm outputs the optimal solution if $C \geqslant 2$.)

I will now describe a dynamic-programming algorithm to compute the exact solution for the knapsack problem. Let $P = p_1 + p_2 + \cdots + p_n$ be the sum of profits of all the given objects. The dynamic-programming algorithm builds a two-dimensional table $T(i, p)$ in the row-major order for all integer values $1 \leqslant i \leqslant n$ and $1 \leqslant p \leqslant P$. Let $S(i, p)$ stand for a lightest (with respect to weight) subset $S$ of the set of the first $i$ objects $O_1, O_2, \ldots, O_i$, for which the sum of the profits of the objects in $S$ is exactly $p$. The table entry $T(i, p)$ stores the sum of the weights of the elements of $S(i, p)$. If no subset of $\{O_1, O_2, \ldots, O_i\}$ has the profit $p$, we take $T(i, p) = \infty$.

We initialize the first row of the table as follows:

$$T(1,p) = \begin{cases} w_1 & \text{if } p = p_1, \\ \infty & \text{otherwise.} \end{cases}$$

Subsequently, for $i = 2,3,\ldots,n$, we populate the $i$-th row of the table, by bringing into consideration the $i$-th object $O_i$. For computing $T(i,p)$, we consider three cases. If $p_i > p$, then no subset of $\{O_1,O_2,\ldots,O_i\}$, which contains $O_i$, can achieve a profit of exactly $p$, that is, we cannot include $O_i$ in $S(i,p)$. If $p_i = p$, we must compare the weights of $\{O_i\}$ and $S(i-1,p)$. Finally, if $p_i < p$, we consider two possibilities. If we plan to take $O_i$, we must account for a profit of exactly $p-p_i$ from objects taken from $\{O_1,O_2,\ldots,O_{i-1}\}$. On the other hand, if we do not take $O_i$, then a lightest subset of $\{O_1,O_2,\ldots,O_{i-1}\}$ of total profit $p$ should be considered. To sum up, we have the following cases:

$$T(i,p) = \begin{cases} T(i-1,p) & \text{if } p < p_i, \\ \min\Big(w_i,\ T(i-1,p)\Big) & \text{if } p = p_i, \\ \min\Big(w_i + T(i-1,p-p_i),\ T(i-1,p)\Big) & \text{if } p > p_i. \end{cases}$$

After the $n$-th row of $T$ is populated, the maximum profit that can be packed in the knapsack is computed as

$$\max_{1 \leqslant p \leqslant P} \big\{ p \mid T(n,p) \leqslant C \big\}.$$

The correctness of this algorithm is evident. Let me now deduce its running time. There are exactly $nP$ entries in the table $T$. First, assume that each weight $w_i$ fits in a single-precision (like 32-bit) integer, and so also does each profit $p_i$. However, as $n$ grows indefinitely, we cannot store the table entries $T(i,p)$ as single-precision integers. But since each $T(i,p)$ is a sum of $\leqslant i \leqslant n$ integer weights, each cell of $T$ does not require more than $\Theta(\log n)$ single-precision words. Therefore, each single $T(i,p)$ entry can be computed in $\mathrm{O}(\log n)$ time. So the total running time of the algorithm is $\mathrm{O}(nP\log n)$. Here, $P$ is not necessarily a single-precision value. However, we can say $P \leqslant np_{max}$, where $p_{max}$ denotes the maximum of $p_1,p_2,\ldots,p_n$. Therefore, the running time of the dynamic-programming algorithm is $\mathrm{O}(n^2 p_{max}\log n)$. Since $p_{max}$ is a single-precision integer value, this is already a polynomial-time algorithm for solving the knapsack problem exactly. Note that, in this case, the input is of size $\Theta(n)$, so we can take $n$ itself as the input-size parameter.

A problem arises when we remove the restriction that $p_i$ are single-precision values. We will continue to assume that the weights are single-precision values (although, frankly speaking, this does not matter). If $l$ is the bit size of $p_{max}$, then the input size is $\mathrm{O}(nl)$, whereas the running time, although polynomial in $n$, becomes exponential in $l$, since $2^{l-1} \leqslant p_{max} < 2^l$. (This situation is identical to the case of multiple-precision integers, as discussed in Chapter 22.)

In order to reduce the running time of this dynamic-programming algorithm, we scale down all the profit values by a constant factor, that is, we define the new profit of the $i$-th object as

$$p'_i = \left\lfloor \frac{p_i}{\sigma} \right\rfloor.$$

The scaling factor $\sigma$ will be chosen later in order to achieve a desired approximation factor $1-\varepsilon$. Notice that the scaled profit values are integer *approximations* of the original profit values, so a solution which is optimal with respect to the profit values $p_i$ may no longer remain optimal with respect to the approximate profit values $p'_i$, and conversely.

We run the above dynamic-programming algorithm on the scaled profit values $p'_i$ (and the original weights $w_i$). The algorithm gives an optimal solution with respect to the scaled values;

call it SOPT$'$. When this recommendation of packing in the knapsack is carried back to the original profit values, we obtain a solution SOPT of the original optimization problem. We require

$$\text{SOPT} \geqslant (1-\varepsilon)\text{OPT},$$

where OPT is the optimal profit for the original problem. Denote, by OPT$'$, the profit of this optimal collection with respect to the scaled profit values.

We have $p'_i = \lfloor \frac{p_i}{\sigma} \rfloor \geqslant \frac{p_i}{\sigma} - 1$, so that $\sigma p'_i \geqslant p_i - \sigma$, that is, $p_i - \sigma p'_i \leqslant \sigma$. Since the collection corresponding to OPT contains at most $n$ objects, it follows that

$$\text{OPT} - \sigma\text{OPT}' \leqslant n\sigma.$$

Moreover, $p'_i = \lfloor \frac{p_i}{\sigma} \rfloor \leqslant \frac{p_i}{\sigma}$. Applying this inequality to the collection corresponding to SOPT$'$ gives

$$\sigma\text{SOPT}' \leqslant \text{SOPT}.$$

Finally, note that SOPT$'$ is optimal with respect to the scaled profit values, that is,

$$\text{SOPT}' \geqslant \text{OPT}'.$$

Combining all these observations yields

$$\text{SOPT} \geqslant \sigma\text{SOPT}' \geqslant \sigma\text{OPT}' \geqslant \text{OPT} - n\sigma.$$

This indicates that the requirement SOPT $\geqslant (1-\varepsilon)$OPT is fulfilled by any $\sigma$ satisfying

$$\sigma \leqslant \frac{\varepsilon \times \text{OPT}}{n}.$$

We may assume that each of the given objects individually fits in the knapsack (because, at the very beginning, we may throw away all the objects heavier than the knapsack capacity). This, in particular, implies that OPT $\geqslant p_{max}$. Therefore, we may take

$$\sigma = \frac{\varepsilon \times p_{max}}{n}.$$

Having established the desired approximation guarantee, I now need to investigate what effect this approximate algorithm has on the running time of solving the knapsack problem. We may use the earlier expression for the running time with $p_{max}$ replaced by $p'_{max}$, that is, the new running time is $\mathrm{O}(n^2 p'_{max} \log n)$. But $p'_{max} = \lfloor p_{max}/\sigma \rfloor \leqslant p_{max}/\sigma = n/\varepsilon$, that is, the running time of the approximate algorithms is $\mathrm{O}\left(\frac{n^3 \log n}{\varepsilon}\right)$, an expression polynomial in both $n$ and $1/\varepsilon$. To sum up, the above approximate dynamic-programming algorithm is a fully polynomial-time approximation scheme (FPTAS) for the knapsack problem.

## 25.2 Randomized algorithms

Most of the algorithms that have appeared in this book so far are *deterministic* in the sense that there is a unique branch of computation for each given input. We have also studied an abstract class of algorithms known as *non-deterministic* algorithms, in which each step is allowed to make a choice from a finite set of possibilities. Such a choice is called a *guess*. If the input is accepted by the algorithm, a proper sequence of guesses leads to a *Yes* answer, whereas an improper sequence leads to a *No* answer. A hypothetical computer with infinite resources (like infinite number of processors) may emulate such a non-deterministic algorithm by a parallel algorithm. A practical computer can at best emulate a non-deterministic algorithm by a sequential exhaustive traversal of the computation tree, often leading to exponential blowup in the running time.