Chapter 6

Fully Polynomial Time Approximation Schemes

Recall that the approximation ratio for an approximation algorithm is a measure to evaluate the approximation performance of the algorithm. The closer the ratio to 1 the better the approximation performance of the algorithm. It is notable that there is a class of NP-hard optimization problems, most originating from scheduling problems, for which there are polynomial time approximation algorithms whose approximation ratio $1 + \epsilon$ can be as close to 1 as desired. Of course, because of the NP-hardness of the problems, the running time of such an algorithm increases when the error bound ϵ decreases, but in a very reasonable way: it is bounded by a polynomial of $1/\epsilon$. Such an approximation algorithm is called a *fully polynomial time approximation scheme* (or shortly FPTAS) for the NP-hard optimization problem. A fully polynomial time approximation scheme seems the best possible we can expect for approximating an NP-hard optimization problem.

In this chapter, we introduce the main techniques for constructing fully polynomial time approximation schemes for NP-hard optimization problems. These techniques include pseudo-polynomial time algorithms and approximation by scaling. Two NP-hard optimization problems, the KNAPSACK problem and the c-MAKESPAN problem, are used as examples to illustrate the techniques. In the last section of this chapter, we also give a detailed discussion on what NP-hard optimization problems may not have a fully polynomial time approximation scheme. An important concept, the strong NP-hardness, is introduced, and we prove that in most cases, a strongly NP-hard optimization problem has no fully polynomial time approximation scheme unless our working conjecture $\mathsf{P} \neq \mathsf{NP}$ fails.

6.1 Pseudo-polynomial time algorithms

We first consider algorithms that solve certain NP-hard optimization problems *precisely*. Of course, we cannot expect that these algorithms run in polynomial time. However, these algorithms run in *pseudo-polynomial time* in the sense that the running time of these algorithms is bounded by a two-variable polynomial whose variables are the length of the input instance and the largest number appearing in the input instance. These pseudo-polynomial time algorithms will play a crucial role in our later development of approximation algorithms for the NP-hard optimization problems.

We start with the KNAPSACK problem, which is defined as follows.

$$\text{KNAPSACK} = (I_Q, S_Q, f_Q, opt_Q)$$
$$I_Q = \{\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle \mid s_i, v_j, B : \text{positive integers}\}$$
$$S_Q(\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle) = \{S \subseteq \{1, \ldots, n\} \mid \textstyle\sum_{i\in S} s_i \leq B\}$$
$$f_Q(\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle, S) = \textstyle\sum_{i\in S} v_i$$
$$opt_Q = \max$$

It is an easy exercise to prove, via a polynomial-time reduction from the NP-hard problem PARTITION, that the KNAPSACK problem is NP-hard.

The KNAPSACK problem is to take the maximum value with a knapsack of size B, given n items of size s_i and value v_i, $i = 1, \ldots, n$. To simplify our description, for a subset S of $\{1, \ldots, n\}$, we will call $\sum_{i\in S} s_i$ the *size* of S and $\sum_{i\in S} v_i$ the *value* of S. Let V_0 be a value not smaller than the value of optimal solutions to the instance $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$. For each index i, $1 \leq i \leq n$ and for each value $v \leq V_0$, we consider the following question

> **Question** $K(i, v)$
>
> Is there a subset S of $\{1, \ldots, i\}$ such that the size of S is not larger than B and the value of S is equal to v?

The answer to Question $K(i, v)$ is "yes" *if and only if* at least one of the following two cases is true: (1) there is a subset S' of $\{1, \ldots, i-1\}$ such that the size of S' is not larger than B and the value of S' is equal to v (in this case, simply let S be S'), and (2) there is a subset S'' of $\{1, \ldots, i-1\}$ such that the size of S'' is not larger than $B - s_i$ and the value of S'' is equal to $v - v_i$ (in this case, let $S = S'' \cup \{i\}$). Therefore, the solution to Question $K(i, v)$ is implied by solutions to the questions of the form $K(i-1, *)$.

For small values of i and v, the solution to Question $K(i, v)$ can be decided directly. In particular, the answer to $K(0, v)$ is always "no" for $v > 0$; and the answer to $K(0, 0)$ is "yes".

The above discussion motivates the following dynamic programming algorithm for solving the KNAPSACK problem. We first compute $K(0, v)$ for all v, $0 \leq v \leq V_0$. Then, inductively we compute each $K(i, v)$ based on the solutions to $K(i-1, v')$ for all $0 \leq v' \leq V_0$. For each item $K(i, v)$, we associate it with a subset S in $\{1, \ldots, i\}$ such that the size of S is not larger than B and the value of S is equal to v, if such a subset exists at all.

Now a potential problem arises. How do we handle two different witnesses for a "yes" answer to the Question $K(i, v)$? More specifically, suppose that we find two subsets S_1 and S_2 of $\{1, \ldots, i\}$ such that both of S_1 and S_2 have size bounded by B and value equal to v, should we keep both of them with $K(i, v)$, or ignore one of them? Keeping both can make $K(i, v)$ exponentially grow as i increases, which will significantly slow down our algorithm. Thus, we intend to ignore one of S_1 and S_2. Which one do we want to ignore? Intuitively, the one with larger size should be ignored (recall that S_1 and S_2 have the same value). However, we must make sure that ignoring the set of larger size will not cause a loss of the track of optimal solutions to the original instance of the KNAPSACK problem. This is ensured by the following lemma.

Lemma 6.1.1 *Let S_1 and S_2 be two subsets of $\{1, \ldots, i\}$ such that S_1 and S_2 have the same value, and the size of S_1 is at least as large as the size of S_2. If S_1 leads to an optimal solution $S = S_1 \cup S_3$ for the* KNAPSACK *problem, where $S_3 \subseteq \{i+1, \ldots, n\}$, then $S' = S_2 \cup S_3$ is also an optimal solution for the* KNAPSACK *problem.*

PROOF. Let $size(S)$ and $value(S)$ denote the size and value of a subset S of $\{1, \ldots, n\}$, respectively. We have

$$size(S') = size(S_2) + size(S_3) \quad \text{and} \quad size(S) = size(S_1) + size(S_3)$$

By the assumption that $size(S_1) \geq size(S_2)$, we have $size(S) \geq size(S')$. Since S is an optimal solution, we have $size(S) \leq B$, which implies $size(S') \leq B$. Thus S' is also a solution to the KNAPSACK problem. Moreover, since $value(S_1) = value(S_2)$, we have

$$\begin{aligned} value(S') &= value(S_2) + value(S_3) \\ &= value(S_1) + value(S_3) \\ &= value(S) \end{aligned}$$

Thus, S' is also an optimal solution. □

By Lemma 6.1.1, for two subsets S_1 and S_2 of $\{1, \ldots, i\}$ that both witness the "yes" answer to Question $K(i, v)$, if the one of larger size leads to an optimal solution, then the one with smaller size also leads to an optimal solution. Therefore, ignoring the set of larger size will not lead to a loss of the track of optimal solutions. That is, if we can derive an optimal solution based on the set of larger size, then we can also derive an optimal solution based on the set of smaller size using exactly the same procedure.

Now a dynamic programming algorithm based on the above discussion can be described as in Figure 6.1. Here the order of computation is slightly different from the one described above: instead of computing $K(i, v)$ based on $K(i-1, v)$ and $K(i-1, v-v_i)$, we start from each $K(i-1, v')$ and try to "extend" it to $K(i, v')$ and $K(i, v'+v_i)$.

Subroutine. Put$(S_0, K[i, v])$
1. **if** $(K[i, v] = *)$ $K[i, v] = S_0$
2. **else if** $(size(S_0) < size(K[i, v]))$ $K[i, v] = S_0$.

Algorithm. Knapsack-Dyn(n, V_0)
INPUT: $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$, all positive integers
OUTPUT: a subset $S \subseteq \{1, ..., n\}$ of size $\leq B$ and value maximized

1. **for** (all $0 \leq i \leq n$ and $0 \leq v \leq V_0$) $K[i, v] = *$;
2. $K[0, 0] = \phi$; \\ ϕ is the empty set
3. **for** (all $0 \leq i \leq n-1$ and $0 \leq v \leq V_0$)
3.1 **if** $(K[i, v] \neq *)$
3.1.1 **Put**$(K[i, v], K[i+1, v])$;
3.1.2 **if** $(size(K[i, v]) + s_{i+1} \leq B)$
Put$(K[i, v] \cup \{i+1\}, K[i+1, v+v_{i+1}])$;
4. return the item $K[n, v] \neq *$ with v maximized.

Figure 6.1: Dynamic programming for KNAPSACK

The subroutine **Put**$(S_0, K[i, v])$ is used to solve the multiple witness problem, where S_0 is a subset of $\{1, \ldots, i\}$ such that S_0 has value v.

Step 4 of the algorithm **Knapsack-Dyn**(n, V_0) finds the largest value $v \leq V_0$ such that $K[n, v] \neq *$. Obviously, if V_0 is not smaller than the value of optimal solutions to the input instance, then step 4 of the algorithm will find the subset S of $\{1, 2, \ldots, n\}$ with the largest value under the constraint that S has size bounded by B.

According to our discussion, it should be clear that the algorithm **Knapsack-Dyn**(n, V_0) solves the KNAPSACK problem for any value V_0 not smaller than the value of optimal solutions to the input instance.

Lemma 6.1.2 *The algorithm* **Knapsack-Dyn**(n, V_0) *runs in time* $O(nV_0)$.

PROOF. We show data structures on which the **if** statement in Step 3.1 can be executed in constant time. The theorem follows directly from this discussion.

For each item $K[i, v]$, which is for a subset S_{iv} of $\{1, \ldots, i\}$, we attach three parameters: (1) the size of S_{iv}, (2) a marker m_{iv} indicating whether i is contained in S_{iv}, and (3) a pointer p_{iv} to an item $K[i-1, v']$ in the previous row such that the set S_{iv} is derived from the set $K[i-1, v']$. Note that the actual set S_{iv} is *not* stored in $K[i, v]$.

With these parameters, the size of the set S_{iv} can be directly read from $K[i, v]$ in constant time. Moreover, it is also easy to verify that the subroutine calls **Put**$(K[i, v], K[i+1, v])$ and **Put**$(K[i, v] \cup \{i+1\}, K[i+1, v+v_{i+1}])$ can be performed in constant time by updating the parameters in $K[i+1, v]$ and $K[i+1, v+v_{i+1}]$.

Thus, steps 1-3 of algorithm **Knapsack-Dyn**(n, V_0) take time $O(nV_0)$.

We show how the actual optimal solution $K[n, v]$ is returned in step 4. After deciding the item $K[n, v]$ in step 4, which corresponds to an optimal solution S_{nv} that is a subset of $\{1, \ldots, n\}$, we first check the marker m_{nv} to see if S_{nv} contains n, then follow the point p_{nv} to an item $K[n-1, v']$, where we can check whether the set S_{nv} contains $n-1$ and a pointer to an item in the $(n-2)$nd row, and so on. In time $O(n)$, we will be able to "backtrack and collect" all elements in S_{nv} and return the actual set S_{nv}. □

A direct implementation of the algorithm **Knapsack-Dyn**(n, V_0) uses a 2-dimensional array $K[0..n, 0..V_0]$, which takes $O(nV_0)$ space (i.e., the amount of computer memory). We may reduce the space complexity of the algorithm from $O(nV_0)$ to $O(V_0)$, as follows. Observe that at any moment, only two rows $K[i, *]$ and $K[i+1, *]$ of the array $K[0..n, 0..V_0]$ need to be kept: when the values of the i-th row become available, all values for rows before the i-th row will not be used further so they can be ignored. Therefore, in the algorithm **Knapsack-Dyn**(n, V_0), we can use two arrays of size V_0 to keep the current two rows, which take only $O(V_0)$ space. However, we should remark that this implementation will only give you the value of the optimal solution. In order to construct the actual subset for the optimal solution, as we explained above for step 4, we still need to keep the entire 2-dimensional array $K[0..n, 0..V_0]$.

In general, we can conveniently let the bound V_0 be $\sum_{i=1}^{n} v_i$, which is an obvious upper bound for the optimal solution value. With this bound V_0, the algorithm **Knapsack-Dyn**(n, V_0) runs in time polynomial in both n and

V_0, and solves the KNAPSACK problem precisely. Unfortunately, since the value V_0 can be larger than any polynomial of n, the algorithm **Knapsack-Dyn**(n, V_0) is not a polynomial time algorithm in terms of the input length n. On the other hand, the algorithm **Knapsack-Dyn**(n, V_0) does provide very important information about the KNAPSACK problem, in particular from the following points of views:

1. If values of all items in the input instance are not very large, i.e., bounded by a polynomial of n, then the value V_0 is also bounded by a polynomial of n. In this case, the algorithm **Knapsack-Dyn**(n, V_0) runs in time polynomial in n and constructs an optimal solution for the given input instance; and

2. The algorithm has laid an important foundation for approximation algorithms for the KNAPSACK problem. This will be discussed in detail in the next section.

The algorithm **Knapsack-Dyn**(n, V_0) is a typical method for solving a class of optimization problems, in particular many scheduling problems. To study this method in general setting, we start with some terminologies.

Definition 6.1.1 Suppose $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ is an optimization problem. For each input instance $x \in I_Q$ we define:

- length(x) = the length of a binary representation of x; and
- max(x) = the largest number that appears in the input x.

If no number appears in the input instance x, we define max$(x) = 1$.

The definitions of length(x) and max(x) can vary by some degree without loss of the generality of our discussion. For example, length(x) can also denote the length of the representation of the input x based on any fixed alphabet, and max(x) can be defined to be the sum of all numbers appearing in the input x. Our discussion below will be valid for any of these variations. The point is that for two different definition systems (length(x), max(x)) and (length$'(x)$, max$'(x)$), we require that length(x) and length$'(x)$ are polynomially related and that max(x) and max$'(x)$ are polynomially related for all input instances x.

Definition 6.1.2 Let Q be an optimization problem. A algorithm $\mathcal{A}$ solving Q runs in *pseudo-polynomial time* if there is a two-variable polynomial

p such that on any input instance x of Q, the running time of the algorithm $\mathcal{A}$ is bounded by $p(\text{length}(x), \max(x))$. In this case, we also say that the problem Q is solvable in pseudo-polynomial time.

As an example, consider the KNAPSACK problem. For any instance of the problem $\alpha = \langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$, if we let $V_0 = \sum_{i=1}^{n} v_i \leq n \cdot \max(\alpha)$, where $\max(\alpha) = \max_{i=1}^{n}\{v_i\}$, then the algorithm **Knapsack-Dyn**(n, V_0) constructs an optimal solution for the instance α in time $O(nV_0)$, which is bounded by a polynomial of length(α) and $\max(\alpha)$. Thus,

Theorem 6.1.3 KNAPSACK *is solvable in pseudo-polynomial time.*

KNAPSACK is a maximization problem. As another example, we present a pseudo-polynomial time algorithm for the minimization problem c-MAKESPAN, where c is a fixed integer. Recall the defiition of the problem:

c-MAKESPAN $= (I_Q, S_Q, f_Q, opt_Q)$

I_Q: the set of tuples $T = \langle t_1, \ldots, t_n\rangle$, where each t_i is an integer denoting the processing time for the ith job

S_Q: $S_Q(T)$ is the set of partitions $P = (S_1, \ldots, S_c)$ of the numbers $\langle t_1, \ldots, t_n\rangle$ into c parts (P is called a *schedule* of $\langle t_1, \ldots, t_n\rangle$)

f_Q: $f_Q(T, P) = \max_{1\leq d\leq c}\{\sum_{t_j\in S_d} t_j\}$

opt_Q: min

Again, a simple polynomial-time reduction from the NP-hard problem PARTITION shows that for any integer $c > 1$, the c-MAKESPAN problem is NP-hard. Thus, there is no polynomial-time algorithm for the c-MAKESPAN problem for any integer $c > 1$ unless P = NP.

Let T_0 be a value not smaller than the value of optimal solutions to the instance $\langle t_1, \ldots, t_n\rangle$. Note that every schedule $(S_1, \ldots, S_c)$ of the n jobs $\langle t_1, \ldots, t_n\rangle$, where S_d is the subset of $\{1, \ldots, n\}$ that corresponds to the jobs assigned to the d-th processor, can be written as a c-tuple $(T_1, \ldots, T_c)$ with $0 \leq T_d \leq T_0$ for all $1 \leq d \leq c$, where $T_d = \sum_{h\in S_d} t_h$ is the total execution time assigned to the d-th processor. The c-tuple $(T_1, \ldots, T_c)$ will be called the *time configuration* for the schedule $(S_1, \ldots, S_c)$.

Now as for the KNAPSACK problem, for each i, $0 \leq i \leq n$, and for each time configuration $(T_1, \ldots, T_c)$, $1 \leq T_d \leq T_0$, $1 \leq d \leq c$, we ask the question

> Is there a schedule of the first i jobs $\{t_1, \ldots, t_i\}$ that gives the time configuration $(T_1, \ldots, T_c)$?

This question is equivalent to the following question

> Is there an index d, where $1 \leq d \leq c$, such that the first $i-1$ jobs $\{t_1, \ldots, t_{i-1}\}$ can be scheduled with the time configuration $(T_1, \ldots, T_{d-1}, T_d - t_i, T_{d+1}, \ldots, T_c)$?

This observation suggests the dynamic programming algorithm given in Figure 6.2. A $c+1$ dimensional array $H[0..n, 0..T_0, \ldots, 0..T_0]$ is used such that the item $H[i, T_1, \ldots, T_c]$ records the existence of a schedule on the first i jobs with the time configuration $(T_1, \ldots, T_c)$. Again, instead of recording the whole schedule corresponding to $H[i, T_1, \ldots, T_c]$, we can simply record the processor index to which the i-th job is assigned. A pointer is used in $H[i, T_1, \ldots, T_c]$ that points to an item of form $H[i-1, *, \ldots, *]$ so that the machine assignment of each of the first $i-1$ jobs can be found following the pointers.

Algorithm. c-**Makespan-Dyn**(n, T_0)
INPUT: n jobs with processing times $t_1, \ldots, t_n$
OUTPUT: an optimal schedule of the jobs on c processors

1. **for** each i, $0 \leq i \leq n$, and each $(T_1, \ldots, T_c)$, $0 \leq T_d \leq T_0$, $1 \leq d \leq c$ **do**
   $H[i, T_1, \ldots, T_c] = *$;
2. $H[0, 0, \ldots, 0] = 0$;
3. **for** $i = 0$ **to** $n-1$ **do**
   for each $(T_1, \ldots, T_c)$, $0 \leq T_d \leq T_0$, $1 \leq d \leq c$ **do**
   if $H[i, T_1, \ldots, T_c] \neq *$ **then**
   for $d = 1$ **to** c **do** \\recording job t_{i+1} is assigned to processor d
   $H[i+1, T_1, \ldots, T_{d-1}, T_d + t_{i+1}, T_{d+1}, \ldots, T_c] = d$;
4. return the $H[n, T_1, \ldots, T_c] \neq *$ with $\max_d\{T_d\}$ minimized.

Figure 6.2: Dynamic programming for c-MAKESPAN

An obvious upper bound T_0 on the value of optimal solutions is $\sum_{i=1}^{n} t_i$. The following theorem follows directly from the algorithm c-**Makespan-Dyn**(n, T_0), with $T_0 = \sum_{i=1}^{n} t_i$.

Theorem 6.1.4 *The algorithm* c-**Makespan-Dyn**(n, T_0) *solves the problem* c-MAKESPAN *in time* $O(nT_0^c)$. *In consequence, the* c-MAKESPAN *problem is solvable in pseudo-polynomial time.*

In many practical applications, developing a pseudo-polynomial time algorithm for an NP-hard optimization problem may have significant impact.

First, in most practical applications, numbers appearing in an input instance are in general not extremely large. For example, numbers appearing in scheduling problems in general represent processing resource (e.g., computational time and storage) requirements for tasks, which are unlikely to be very large because we will actually process the tasks after the schedule and we could not afford to do so if any task requires an inordinately large amount of resource. For this kind of applications, a pseudo-polynomial time algorithm will become a polynomial-time algorithm and solve the problem, even if the original problem is NP-hard in its general form.

Furthermore, a pseudo-polynomial time algorithm can be useful even when there is no natural bound on the numbers appearing in input instances. In general, input instances that are of practical interests and contain very large numbers might be very rare. If this is the case, then a pseudo-polynomial time algorithm will work efficiently for most input instances, and only "slow down" in very rare situations.

6.2 Approximation by scaling

In the last section, we presented an algorithm **Knapsack-Dyn**(n, V_0) that, on an input instance $\alpha = \langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$ of KNAPSACK, constructs an optimal solution for α in time $O(nV_0)$, where $V_0 = \sum_{i=1}^{n} v_i$. If V_0 is not bounded by any polynomial function of n, then the running time of the algorithm is not polynomial. Is there a way to lower the value of V_0? Well, an obvious way is to divide each value v_i by a sufficiently large number K so that V_0 is replaced by a smaller value $V_0' = V_0/K$. In order to let the algorithm **Knapsack-Dyn**(n, V_0') to run in polynomial time, we must have $V_0' \leq an^b$ for some constants a and b, or equivalently, $K \geq V_0/(an^b)$. Another problem is that the value v_i/K may no longer be an integer while by our definition, all input values in an instance of the KNAPSACK problem are integers. Thus, we will take $v_i' = \lfloor v_i/K \rfloor$. This gives a new instance α' for the KNAPSACK problem

$$\alpha' = \langle s_1, \ldots, s_n; v_1', \ldots, v_n'; B\rangle$$

where $v_i' = \lfloor v_i/K \rfloor$, for $i = 1, \ldots, n$, and $V_0' = \lceil V_0/K \rceil$ is obviously an upper bound on the value of optimal solutions to the instance α'. For $K \geq V_0/(an^b)$ for some constants a and b, the algorithm **Knapsack-Dyn**(n, V_0') finds an optimal solution for α' in polynomial time. Note that a solution to α' is also a solution to α and we intend to "approximate" the optimal solution to α by an optimal solution to α'. Since the application of the floor function $\lfloor \cdot \rfloor$,

we lose precisions thus an optimal solution for α' may not be an optimal solution for α. How much precision have we lost? Intuitively, the larger the value K, the more precision we would lose. Thus, we want K to be as small as possible. On the other hand, we want K to be as large as possible so that the running time of the algorithm **Knapsack-Dyn**(n, V_0') can be bounded by a polynomial. Now a natural question is whether there is a value K that makes the algorithm **Knapsack-Dyn**(n, V_0') run in polynomial time and cause not much precision loss so that the optimal solution to the instance α' is "close" to the optimal solution to the instance α. For this, we need the following formal analysis.

Let $S \subseteq \{1, \ldots, n\}$ be an optimal solution to the instance α, and let $S' \subseteq \{1, \ldots, n\}$ be the optimal solution to the instance α' produced by the algorithm **Knapsack-Dyn**(n, V_0'). Note that S is also a solution to the instance α' and that S' is also a solution to the instance α. Let $Opt(\alpha) = \sum_{i \in S} v_i$ and $Apx(\alpha) = \sum_{j \in S'} v_i$ be the objective function values of the solutions S and S', respectively. Therefore, $Opt(\alpha)/Apx(\alpha)$ is the approximation ratio for the algorithm we proposed. In order to bound the approximation ratio by a given constant ϵ, we consider

$$\begin{aligned} Opt(\alpha) &= \sum_{i \in S} v_i = K \sum_{i \in S} v_i/K \leq K \sum_{i \in S} (\lfloor v_i/K \rfloor + 1) \\ &\leq Kn + K \sum_{i \in S} \lfloor v_i/K \rfloor = Kn + K \sum_{i \in S} v_i' \end{aligned}$$

The last inequality is because the cardinality of the set S is bounded by n.

Since S' is an optimal solution to $\alpha' = \langle s_1, \ldots, s_n; v_1', \ldots, v_n'; B \rangle$, while S is also a solution to α', we must have

$$\sum_{i \in S} v_i' \leq \sum_{i \in S'} v_i'$$

Thus,

$$\begin{aligned} Opt(\alpha) &\leq Kn + K \sum_{i \in S'} v_i' = Kn + K \sum_{i \in S'} \lfloor v_i/K \rfloor \\ &\leq Kn + K \sum_{i \in S'} v_i/K = Kn + Apx(\alpha) \end{aligned} \tag{6.1}$$

This gives us the approximation ratio.

$$\frac{Opt(\alpha)}{Apx(\alpha)} \leq 1 + \frac{Kn}{Apx(\alpha)}$$

Without loss of generality, we can assume that $s_i \leq B$ for all $i = 1, \ldots, n$ (otherwise, the index i can be simply deleted from the input instance since it can never make contribution to a feasible solution to α). Thus, $Opt(\alpha)$ is at least as large as $\max_{1 \leq i \leq n}\{v_i\} \geq V_0/n$, where $V_0 = \sum_{i=1}^{n} v_i$. From inequality (6.1), we have

$$Apx(\alpha) \geq Opt(\alpha) - Kn \geq (V_0/n) - Kn$$

It follows that

$$\frac{Opt(\alpha)}{Apx(\alpha)} \leq 1 + \frac{Kn}{(V_0/n) - Kn} = 1 + \frac{Kn^2}{V_0 - Kn^2}$$

Thus, in order to bound the approximation ratio by $1 + \epsilon$, one should have

$$\frac{Kn^2}{V_0 - Kn^2} \leq \epsilon$$

This leads to $K \leq (\epsilon V_0)/(n^2(1 + \epsilon))$.

Recall that in order to make the algorithm **Knapsack-Dyn**(n, V_0') run in polynomial time on the input instance α', we must have $K \geq V_0/(an^b)$ for some constants a and b. Combining these two relations, we get $a = 1 + 1/\epsilon$, and $b = 2$, and the value

$$K = V_0/(an^b) = \frac{V_0}{(1 + 1/\epsilon)n^2}$$

makes the algorithm **Knapsack-Dyn**(n, V_0') run in time $O(n^3(1 + 1/\epsilon)) = O(n^3/\epsilon)$ and produces a solution S' to the instance α with approximation ratio bounded by $1 + \epsilon$.

We summarize the above discussion in the algorithm given in Figure 6.3.

Theorem 6.2.1 *For an input instance α of* KNAPSACK *and for any real number $\epsilon > 0$, the algorithm* **Knapsack-Apx** *runs in time $O(n^3/\epsilon)$ and produces a solution to α with approximation ratio bounded by $1 + \epsilon$.*

According to Theorem 6.2.1, the running time of the approximation algorithm **Knapsack-Apx** increases when the input size n increases and the error bound ϵ decreases. This seems reasonable and necessary. Moreover, the running time increases "slowly" with n and $1/\epsilon$ – which is bounded by a polynomial of n and $1/\epsilon$. This motivates the following definition.

Algorithm. Knapsack-Apx
INPUT: $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$, and $\epsilon > 0$
OUTPUT: a subset $S' \subseteq \{1, ..., n\}$, such that $\sum_{i \in S'} s_i \leq B$

1. $V_0 = \sum_{i=1}^{n} v_i$; $\quad K = \frac{V_0}{(1+1/\epsilon)n^2}$;
2. **for** $i = 1$ **to** n **do** $\quad v'_i = \lfloor v_i/K \rfloor$;
3. apply algorithm **Knapsack-Dyn**$(n, \lceil V_0/K \rceil)$ on $\langle s_1, \ldots, s_n; v'_1, \ldots, v'_n; B\rangle$;
4. return the subset $S' \subseteq \{1, \ldots, n\}$ obtained in step 3.

Figure 6.3: FPTAS for the KNAPSACK problem

Definition 6.2.1 An optimization problem Q has a *fully polynomial time approximation scheme* (FPTAS) if it has an approximation algorithm A such that given $\langle x, \epsilon\rangle$, where x is an instance of Q and ϵ is a positive constant, A finds a solution for x with approximation ratio bounded by $1 + \epsilon$ in time polynomial in both n and $1/\epsilon$.

By the definition, the KNAPSACK problem has a fully polynomial time approximation scheme. In the following, we present a fully polynomial time approximation scheme for the c-MAKESPAN problem.

The approach for developing a fully polynomial time approximation scheme for the c-MAKESPAN problem is similar to that for the KNAPSACK problem. For an input instance $\alpha = \langle t_1, \ldots, t_n\rangle$ of the c-MAKESPAN problem, we have the dynamic programming algorithm **c-Makespan-Dyn**(n, T_0), which constructs an optimal solution to the instance α in time $O(nT_0^c)$, where $T_0 = \sum_{i=1}^{n} t_i$. We reduce the running time of the algorithm by scaling the value T_0 by dividing all t_i in the input instance α by a large number K. By properly choosing the scaling factor K, we can make the algorithm **c-Makespan-Dyn** to run on the new instance in polynomial time and keep the approximation ratio bounded by $1 + \epsilon$. Because of the similarity, some details in the algorithms and in the analysis are omitted. The reader is advised to refer to corresponding discussion on the KNAPSACK problem and complete the omitted parts for a better understanding.

The approximation algorithm for c-MAKESPAN is given in Figure 6.4.

Theorem 6.2.2 *The algorithm* **c-Makespan-Apx** *on input* $\langle t_1, \ldots, t_n; \epsilon\rangle$ *produces a schedule* $(S'_1, \ldots, S'_c)$ *with approximation ratio bounded by* $1 + \epsilon$ *and runs in time* $O(n^{c+1}/\epsilon^c)$.

PROOF. The time complexity of the algorithm **c-Makespan-Apx** is dom-

Algorithm. c**-Makespan-Apx**
INPUT: $\langle t_1, \ldots, t_n; \epsilon \rangle$, all t_i's are integers
OUTPUT: a schedule of the n jobs on c processors

1. $T_0 = \sum_{i=1}^{n} t_i$; $K = \epsilon T_0/(cn)$; $T_0' = \lceil T_0/K \rceil + n$;
2. **for** $i = 1$ **to** n **do** $t_i' = \lceil t_i/K \rceil$;
3. apply algorithm c-**Makespan-Dyn**(n, T_0') on input $\langle t_1', \ldots, t_n' \rangle$;
4. return the schedule obtained in step 3.

Figure 6.4: FPTAS for the c-MAKESPAN problem

inated by step 3. Since $T_0' = \lceil T_0/K \rceil + n = cn/\epsilon + n = O(n/\epsilon)$, by Theorem 6.1.4, the algorithm c-**Makespan-Dyn**(n, T_0') in step 3, thus the algorithm c-**Makespan-Apx**, runs in time $O(n(T_0')^c) = O(n^{c+1}/\epsilon^c)$.

Let $(S_1, \ldots, S_c)$ be an optimal solution to the instance $\alpha = \langle t_1, \ldots, t_n \rangle$ of the c-MAKESPAN problem, and let $(S_1', \ldots, S_c')$ be the optimal solution to the instance $\alpha' = \langle t_1', \ldots, t_n' \rangle$ obtained by the algorithm c-**Makespan-Dyn**. Note that $(S_1, \ldots, S_c)$ is also a solution to the instance $\alpha' = \langle t_1', \ldots, t_n' \rangle$ and $(S_1', \ldots, S_c')$ is also a solution to the instance $\alpha = \langle t_1, \ldots, t_n \rangle$.

For all d, $1 \le d \le c$, let

$$T_d = \sum_{h \in S_d} t_h, \quad V_d = \sum_{h \in S_d} t_h', \quad T_d' = \sum_{h \in S_d'} t_h, \quad V_d' = \sum_{h \in S_d'} t_h'.$$

Without loss of generality, suppose

$$T_1 = \max_{1 \le d \le c} \{T_d\}, \quad V_2 = \max_{1 \le d \le c} \{V_d\}, \quad T_3' = \max_{1 \le d \le c} \{T_d'\}, \quad V_4' = \max_{1 \le d \le c} \{V_d'\}.$$

Therefore, on instance $\langle t_1, \ldots, t_n \rangle$, the schedule $(S_1, \ldots, S_c)$ has makespan T_1 and the schedule $(S_1', \ldots, S_c')$ has makespan T_3'; and on instance $\langle t_1', \ldots, t_n' \rangle$, the schedule $(S_1, \ldots, S_c)$ has makespan V_2 and the schedule $(S_1', \ldots, S_c')$ has makespan V_4'. The approximation ratio of the algorithm c-**Processor-Apx** is T_3'/T_1. We have

$$T_3' = \sum_{h \in S_3'} t_h = K \sum_{h \in S_3'} (t_h/K) \le K \sum_{h \in S_3'} t_h' = KV_3' \le KV_4'.$$

The last inequality is by the assumption $V_4' = \max_{1 \le d \le c} \{V_d'\}$.

Since $(S_1', \ldots, S_c')$ is an optimal schedule on instance $\langle t_1', \ldots, t_n' \rangle$, we have $V_4' \le V_2$. Thus,

$$T_3' \le KV_2 = K \sum_{h \in S_2} t_h' = K \sum_{h \in S_2} \lceil t_h/K \rceil$$

$$\leq K \sum_{h \in S_2} ((t_h/K) + 1) \leq T_2 + Kn \leq T_1 + Kn.$$

The last inequality is by the assumption $T_1 = \max_{1 \leq d \leq c}\{T_d\}$.

This gives us immediately

$$T_3'/T_1 \leq 1 + Kn/T_1$$

It is easy to see that $T_1 \geq \sum_{i=1}^{n} t_i/c = T_0/c$, and recall that $K = \epsilon T_0/(cn)$, we obtain $Kn/T_1 \leq \epsilon$. That is, the schedule $(S_1', \ldots, S_c')$ produced by the algorithm c-**Makespan-Apx** has approximation ratio bounded by $1+\epsilon$. □

Corollary 6.2.3 *For a fixed constant c, the c-*Makespan *problem has a fully polynomial time approximation scheme.*

Theorem 6.2.1 and Theorem 6.2.2 present fully polynomial time approximation schemes for the Knapsack problem and the c-Makespan problem, respectively, using the pseudo-polynomial time algorithms for the problems by properly scaling and rounding input instances. Most known fully polynomial time approximation schemes for optimization problems are derived using this method. In fact, as shown in [20], under a very general constraint, this is essentially the only way to derive fully polynomial time approximation schemes for optimization problems. Therefore, pseudo-polynomial time algorithms are closely related to fully polynomial time approximation schemes for optimization problems. The following theorem shows that under a very general condition, having a pseudo-polynomial time algorithm is a necessary condition for the existence of a fully polynomial time approximation scheme for an optimization problem.

Theorem 6.2.4 *Let $Q = \langle I, S, f, opt\rangle$ be an optimization problem such that for all input instance $x \in I$ we have $Opt(x) \leq p(length(x), \max(x))$, where p is a two variable polynomial. If Q has a fully polynomial time approximation scheme, then Q can be solved in pseudo-polynomial time.*

Proof. Suppose that Q is a minimization problem, i.e., $opt = \min$. Since Q has a fully polynomial time approximation scheme, there is an approximation algorithm A for Q such that for any input instance $x \in I$, the algorithm A produces a solution $y \in S(x)$ in time $p_1(|x|, 1/\epsilon)$ satisfying

$$\frac{f(x,y)}{Opt(x)} \leq 1 + \epsilon,$$

where p_1 is a two variable polynomial.

Let $\epsilon = 1/(p(\text{length}(x), \max(x)) + 1)$, then the solution y satisfies

$$f(x,y) \leq Opt(x) + \frac{Opt(x)}{p(\text{length}(x), \max(x)) + 1}. < Opt(x) + 1$$

Since both $f(x,y)$ and $Opt(x)$ are integers and $f(x,y) \geq Opt(x)$, we get immediately $f(x,y) = Opt(x)$. That is, the solution produced by the algorithm A is actually an optimal solution. Moreover, the running time of the algorithm A for producing the solution y is bounded by

$$p_1(|x|, p(\text{length}(x), \max(x)) + 1)$$

which is a polynomial of length(x) and $\max(x)$. We conclude that the optimization problem Q can be solved in pseudo-polynomial time. □

6.3 Improving time complexity

We have shown that the KNAPSACK problem and the c-MAKESPAN problem can be approximated within a ratio $1 + \epsilon$ in polynomial time for any given $\epsilon > 0$. On the other hand, one should observe that the running time of the approximation algorithms is very significant. For the KNAPSACK problem, the running time of the approximation algorithm **Knapsack-Apx** is $O(n^3/\epsilon)$; and for the c-MAKESPAN problem, the running time of the approximation algorithm c-**Makespan-Apx** is $O(n^{c+1}/\epsilon^c)$. When the input size n is reasonably large and the required error bound ϵ is very small, these algorithms may become impractical.

In this section, we discuss several techniques that have been used extensively in developing efficient approximation algorithms for scheduling problems. We should point out that these techniques are not only useful for improving the algorithm running time, but also often important for achieving better approximation ratios.

Reducing the number of parameters

Consider the approximation algorithm c-**Makespan-Apx** for the problem c-MAKESPAN (Figure 6.4). The running time of the algorithm is dominated by step 3, which is a call to the dynamic programming algorithm c-**Makespan-Dyn**(n, T_0'). Therefore, if we can improve the time complexity of the algorithm c-**Makespan-Dyn**(n, T_0'), we improve the running time of the approximation algorithm c-**Makespan-Apx**.

The algorithm c-**Makespan-Dyn**(n, T_0') (see Figure 6.2) works on a $(c+1)$-dimensional array $H[0..n, 0..T_0', \ldots, 0..T_0']$, where $T_0' = O(n/\epsilon)$ (see the proof of Theorem 6.2.2). The item $H[i, T_1, \ldots, T_c] = d$ records a schedule for the first i jobs that assigns the job i to the processor d with a time configuration $(T_1, \ldots, T_c)$. The running time of the algorithm c-**Makespan-Dyn**(n, T_0') is necessarily at least $O(n(T_0')^c) = O(n^{c+1}/\epsilon^c)$.

To reduce the running time of the algorithm, we reduce the dimension of the array $H[*, \ldots, *]$ from $c+1$ to c, as follows. We let the item $H[i, T_1, \ldots, T_{c-1}]$ record the processing time of the cth processor. More precisely, $H[i, T_1, \ldots, T_{c-1}] = T_c$ if there is a schedule for the first i jobs whose time configuration is $(T_1, \ldots, T_{c-1}, T_c)$. The modification of the algorithm c-**Makespan-Dyn**(n, T_0') based on this change is straightforward, for which we present the part for step 3 in Figure 6.5. Of course, we still need to keep another two pieces of information related to each item $H[i, T_1, \ldots, T_{c-1}]$: a processor index d indicating that the job i is assigned to processor d, and a pointer to an item $H[i-1, T_1', \ldots, T_{c-1}']$ for constructing the actual schedule corresponding to the time configuration $(T_1, \ldots T_{c-1}, H[i, T_1, \ldots, T_{c-1}])$.

```
.........
4.  for i = 0 to n − 1 do
      for each (T_1, ..., T_{c−1}), 0 ≤ T_d ≤ T_0, 1 ≤ d ≤ c − 1 do
        if H[i, T_1, ..., T_{c−1}] ≠ ∗ then
          H[i + 1, T_1, ..., T_{c−1}] = H[i, T_1, ..., T_{c−1}] + t_{i+1};
          \\assign job t_{i+1} to processor c
          for d = 1 to c − 1 do \\assign job t_i to processors 1, 2, ..., c − 1
          H[i + 1, T_1, ..., T_{d−1}, T_d + t_{i+1}, T_{d+1}, ..., T_{c−1}] = H[i, T_1, ..., T_{c−1}];
.........
```

Figure 6.5: Modified algorithm c-**Makespan-Dyn**

The running time of the algorithm c-**Makespan-Dyn**(n, T_0) now is obviously bounded by $O(nT_0^{c-1})$. Therefore, if step 3 in the algorithm c-**Makespan-Apx** (Figure 6.4) calls the modified algorithm c-**Makespan-Dyn**(n, T_0'), where $T_0' = O(n/\epsilon)$, the running time of the algorithm c-**Makespan-Apx** is reduced from $O(n(T_0')^c) = O(n^{c+1}/\epsilon^c)$ to $O(n(T_0')^{c-1}) = O(n^c/\epsilon^{c-1})$. We summarize the discussion in the following theorem.

Theorem 6.3.1 *The algorithm* c-**Makespan-Apx** *on input* $\langle t_1, \ldots, t_n; \epsilon \rangle$ *produces a schedule* $(S_1', \ldots, S_c')$ *with approximation ratio bounded by* $1+\epsilon$ *and runs in time* $O(n^c/\epsilon^{c-1})$.

Reducing search space

Consider the dynamic programming algorithm **Knapsack-Dyn**(n, V_0) (Figure 6.1). For an instance $\alpha = \langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$ of KNAPSACK, in order to let the algorithm **Knapsack-Dyn**(n, V_0) construct an optimal solution, V_0 must be not smaller than the value of optimal solutions to the instance. In particular, we can let $V_0 = \sum_{i=1}^{n} v_i$. We used a 2-dimensional array $K[0..n, 0..V_0]$. The item $K[i, v]$ records a subset of $\{1, \ldots, i\}$ whose value is v and size is bounded by B. Note that the value of an optimal solution to α can be as small as V_0/n. Therefore, if we can derive a closer upper bound V^* on the value of optimal solutions to α, we may speed up our dynamic programming algorithm by calling **Knapsack-Dyn**(n, V^*) instead of **Knapsack-Dyn**(n, V_0).

To derive a better bound on the optimal solution value, we can perform a "pre-approximation algorithm" that provides a bound V^* not much larger than the optimal solution value, then use this value V^* as an upper bound for the optimal solution value in the dynamic programming algorithm.

Let S be a set of items whose size and value are s_i and v_i, respectively, for $i = 1, \ldots, n$. Let B be an integer. A *B-partition of S* is a triple (S', S'', r), where $r \in S''$, such that

(1) $S' \cup S'' = S$ and $S' \cap S'' = \emptyset$;

(2) $v_j/s_j \geq v_r/s_r \geq v_k/s_k$ for all $j \in S'$ and all $k \in S''$; and

(3) $\sum_{j \in S'} s_j \leq B$ but $\sum_{j \in S'} s_j + s_r > B$.

Now consider the algorithm **Pre-Apx** given in Figure 6.6.

Algorithm. Pre-Apx
INPUT: $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$, all positive integers
OUTPUT: a subset S of $\{1, \ldots, n\}$ of size bounded by B

1. construct a B-partition (S', S'', r) for the set $\{1, 2, \ldots, n\}$;
2. let $v_k = \max_i\{v_i\}$;
3. **if** $(v_k > \sum_{j \in S'} v_j)$ **then** return $\{k\}$ **else** return S'.

Figure 6.6: Finding an upper bound on optimal solution value

We first analyze the complexity of the algorithm.

Lemma 6.3.2 *The algorithm* **Pre-Apx** *runs in linear time.*

PROOF. It is sufficient to show that the B-partition (S', S'', r) of the set $\{1, 2, \ldots, n\}$ can be constructed in linear time.

If we sort the items by the ratios v_i/s_i, then the B-partition can be trivially constructed. However, sorting the items takes time $\Omega(n \log n)$. Therefore, we should avoid sorting.

We use the linear time algorithm that, given a set S of n numbers, returns the median of S (i.e., the $(\lfloor n/2 \rfloor)$-th largest number in S) (the readers are referred to [14] for more detailed discussion of this algorithm).

We perform a binary search procedure as follows. First we find, in linear time, an item h in S such that the ratio v_h/s_h is the median over all ratios $v_1/s_1, \ldots, v_n/s_n$. The item h partitions the set S into two subsets S_1 and S_2 of equal size, where for each item j in S_1, $v_j/s_j \geq v_h/s_h$, and for each item k in S_2, $v_k/s_k \leq v_h/s_h$. Assume $h \in S_2$. The subsets S_1 and S_2 can be constructed in linear time. Let $size(S_1) = \sum_{j \in S_1} s_j$. There are two possible cases: (1) $size(S_1) \leq B$. In this case we recursively construct a B'-partition (S_2', S_2'', r) of the set S_2, where $B' = B - size(S_1)$. Now $(S_1 \cup S_2', S_2'', r)$ is a B-partition of the set S; and (2) $size(S_1) > B$. In this case we construct a B-partition (S_1', S_1'', r) of the set S_1. Then $(S_1', S_1'' \cup S_2, r)$ is a B-partition of the set S. Note that each of the subsets S_1 and S_2 has at most $n/2$ items. Thus, if we let $t(n)$ be the running time of this recursive procedure, we have the recurrence relation $t(n) = O(n) + t(n/2)$, from which we can easily derive $t(n) = O(n)$. That is, the B-partition (S', S'', r) can be constructed in linear time. This completes the proof of the lemma. □

Note that the algorithm **Pre-Apx** is an approximation algorithm for KNAPSACK, whose approximation ratio is given by the following lemma.

Lemma 6.3.3 *The approximation algorithm* **Pre-Apx** *for the* KNAPSACK *problem has an approximation ratio bounded by* 2.

PROOF. Note that when v_j, v_k, s_j, and s_k are positive integers, we have

$$\frac{v_j}{s_j} \geq \frac{v_k}{s_k} \quad \text{implies} \quad \frac{v_j}{s_j} \geq \frac{v_j + v_k}{s_j + s_k} \geq \frac{v_k}{s_k} \tag{6.2}$$

Let (S', S'', r) be the B-partition of $\{1, 2, \ldots, n\}$ constructed by the algorithm **Pre-Apx**. The algorithm **Pre-Apx** constructs a solution $S_{apx} \subseteq \{1, 2, \ldots, n\}$ whose value is

$$\max\{\sum_{j \in S'} v_j, v_1, v_2, \ldots, v_n\}$$

Let $\bar{S} = S' \cup \{r\}$ and let S_{opt} be an optimal solution.

Let $S_0 = \bar{S} \cap S_{opt}$. Thus, $\bar{S} = S_0 \cup T_1$ and $S_{opt} = S_0 \cup T_2$, where $T_1 \cap T_2 = \emptyset$. Note that for any $j \in T_1$ and any $k \in T_2$, we have

$$\frac{v_j}{s_j} \geq \frac{v_r}{s_r} \geq \frac{v_k}{s_k}$$

By repeatedly using the relation (6.2), we have

$$\frac{\sum_{j \in T_1} v_j}{\sum_{j \in T_1} s_j} \geq \frac{v_r}{s_r} \geq \frac{\sum_{k \in T_2} v_k}{\sum_{k \in T_2} s_k} \tag{6.3}$$

Sine the size of $\bar{S}$ is larger than B while the size of S_{opt} is bounded by B, we have $\sum_{j \in T_1} s_j > \sum_{k \in T_2} s_k$, which combined with (6.3) gives

$$\sum_{j \in T_1} v_j \geq \sum_{j \in T_2} v_k$$

This shows that the value of the set $\bar{S}$ is not smaller than the value of the optimal solution S_{opt}. Since $\bar{S} = S' \cup \{r\}$, according to the algorithm **Pre-Apx**, the value of $\bar{S}$ is bounded by twice of the value of the solution S_{apx} constructed by the algorithm **Pre-Apx**. This proves that the approximation ratio of the algorithm **Pre-Apx** is bounded by 2. □

Therefore, for an instance α of KNAPSACK, we can first apply the algorithm **Pre-Apx** to construct a solution. Suppose that this solution has value V^*, then we have $V^* \leq Opt(\alpha) \leq 2V^*$, where $Opt(\alpha)$ is the optimal solution value of α. Thus, the value $2V^*$ can be used as an upper bound for the optimal solution value for α.

We show how this refinement improves the running time of our fully polynomial time approximation scheme for the KNAPSACK problem. For this, we modify our scaling factor K in the algorithm **Knapsack-Apx** (Figure 6.3). The modified algorithm is given in Figure 6.7.

The following theorem shows that the modified algorithm **Knapsack-Apx (Revision I)** for the KNAPSACK problem has the same approximation ratio but the running time improved by a fact n.

Theorem 6.3.4 *The algorithm* **Knapsack-Apx (Revision I)** *for the* KNAPSACK *problem has approximation ratio* $1+\epsilon$ *and runs in time* $O(n^2/\epsilon)$.

PROOF. Again the time complexity of the algorithm is dominated by step 4, which calls the dynamic programming algorithm **Knapsack-Dyn**(n, V_0'). By Lemma 6.1.2, step 4 of the algorithm takes time $O(nV_0')$. Since $V_0' =$

Algorithm. Knapsack-Apx (Revision I)
INPUT: $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$, and $\epsilon > 0$
OUTPUT: a subset $S' \subseteq \{1, ..., n\}$, such that $\sum_{i \in S'} s_i \leq B$

1. call algorithm **Pre-Apx** to obtain a solution of value V^*;
2. $K = \frac{V^*}{n(1+1/\epsilon)}$; $V_0' = \lfloor 2V^*/K \rfloor$;
3. **for** $i = 1$ **to** n **do** $v_i' = \lfloor v_i/K \rfloor$;
4. apply algorithm **Knapsack-Dyn**(n, V_0') on $\langle s_1, \ldots, s_n; v_1', \ldots, v_n'; B\rangle$;
5. return the subset $S' \subseteq \{1, \ldots, n\}$ obtained in step 4.

Figure 6.7: Revision I for the FPTAS for KNAPSACK

$\lfloor 2V^*/K \rfloor = 2n(1+1/\epsilon) = O(n/\epsilon)$, we conclude that the running time of the algorithm **Knapsack-Apx (Revision I)** is bounded by $O(n^2/\epsilon)$.

We must ensure that the value V_0' is a large enough upper bound for the optimal value $Opt(\alpha')$ of the instance $\alpha' = \langle s_1, \ldots, s_n; v_1', \ldots, v_n'; B\rangle$. For this, let S be an optimal solution to the instance α'. Then

$$Opt(\alpha') = \sum_{i \in S} v_i' = \sum_{i \in S} \lfloor v_i/K \rfloor \leq (\sum_{i \in S} v_i)/K.$$

Observing that the subset S is also a solution to the original instance $\alpha = \langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$ and by Lemma 6.3.3, $Opt(\alpha) \leq 2V^*$, we have

$$Opt(\alpha') \leq (\sum_{i \in S} v_i)/K \leq Opt(\alpha)/K \leq 2V^*/K.$$

Since $Opt(\alpha')$ is an integer, we get $Opt(\alpha') \leq \lfloor 2V^*/K \rfloor = V_0'$. Therefore, the value V_0' is a valid upper bound for the value $Opt(\alpha')$.

Now we analyze the approximation ratio for the algorithm **Knapsack-Apx (Revision I)**. Using exactly the same derivation as we did in Section 5.2, we get (see the relation in (6.1) in Section 5.2)

$$Opt(\alpha) \leq Kn + Apx(\alpha),$$

where $Apx(\alpha)$ is the value of the solution constructed by the algorithm **Knapsack-Apx (Revision I)**. Dividing both sides by $Apx(\alpha)$, we get

$$\frac{Opt(\alpha)}{Apx(\alpha)} \leq 1 + \frac{Kn}{Apx(\alpha)}.$$

Moreover, since $Apx(\alpha) \geq Opt(\alpha) - Kn \geq V^* - Kn$ (note here we have used a better estimation $Opt(\alpha) \geq V^*$ than the one in Section 5.2, in which the estimation $Opt(\alpha) \geq \sum_{i=1}^{n} v_i/n$ was used), we get

$$\frac{Opt(\alpha)}{Apx(\alpha)} \leq 1 + \frac{Kn}{V^* - Kn} = 1 + \epsilon.$$

This completes the proof of the theorem. □

Separating large items and small items

Another popular technique for improving the running time (and sometimes also the approximation ratio) is to treat large items and small items in an instance differently. The basic idea of this technique can be described as follows: we first set a threshold value T. The items whose value is larger than or equal to T are *large items* and the items whose value is smaller than T are *small items*. We use common methods, such as the dynamic programming method and the scaling method, to construct a solution for the large items. Then we add the small items by greedy method. This approach is based on the following observations: (1) the number of large items is relatively small so that the running time of the dynamic programming can be reduced; (2) applying the floor or ceiling function on the scaled values (such as $\lfloor v_i/K \rfloor$ for the KNAPSACK problem and $\lceil t_i/K \rceil$ for the c-MAKESPAN problem) only for large items in general loses less precision; and (3) greedy method for adding small items in general introduces only small approximation errors.

We illustrate this technique by re-considering the c-MAKESPAN problem.

Let $\alpha = \langle t_1, t_2, \ldots, t_n \rangle$ be an instance of the c-MAKESPAN problem. Let $T_0 = \sum_{i=1}^{n} t_i$. A job t_i is a large job if $t_i \geq \epsilon T_0/c$, and a job t_j is a small job if $t_i < \epsilon T_0/c$. Let α_l be the set of all large jobs and let α_s be the set of all small jobs. Note that the number n_l of large jobs is bounded by

$$n_l \leq \frac{T_0}{\epsilon T_0/c} = \frac{c}{\epsilon}. \tag{6.4}$$

Without loss of generality, we suppose that the first n_l jobs t_1, t_2, …, t_{n_l} are large jobs and the rest of the jobs are small jobs.

Apply the algorithm c-**Makespan-Apx** (see Figure 6.4) to the n_l large jobs $\langle t_1, t_2, \ldots, t_{n_l} \rangle$ with the following modifications:

1. let $T_0' = \sum_{j=1}^{n_l} t_j$, and set $K = \epsilon^2 T_0'/c^2$; and
2. use $T_0'' = \lceil T_0'/K \rceil + n_l$ in the call to the dynamic programming algorithm c-**Makespan-Dyn**(n_l, T_0'') on the scaled instance $\alpha' = \langle t_1', t_2', \ldots, t_{n_l}' \rangle$, where $t_j' = \lceil t_j/K \rceil$, $j = 1, \ldots, n_l$.

The value T_0'' is a valid upper bound on the makespan for the scaled instance $\alpha' = \langle t_1', t_2', \ldots, t_{n_l}' \rangle$ because we have (we use the inequality (6.4) here)

$$\sum_{j=1}^{n_l} t_j' = \sum_{j=1}^{n_l} \lceil t_j/K \rceil \leq \frac{\sum_{j=1}^{n_l} t_j}{K} + n_l = \frac{T_0'}{K} + n_l \leq T_0''.$$

By the analysis in Theorem 6.3.1, the running time of the algorithm c-**Makespan-Apx** on the large jobs $\alpha' = \langle t_1, \ldots, t_{n_l} \rangle$ is bounded by $O(n_l (T_0'')^{c-1})$. Replacing T_0'' by $\lceil T_0'/K \rceil + n_l$, n_l by c/ϵ, and K by $\epsilon^2 T_0'/c^2$, we conclude that the running time of the algorithm c-**Makespan-Apx** on the large jobs $\alpha_l = \langle t_1, \ldots, t_{n_l} \rangle$ is bounded by $O(1/\epsilon^{2c-1})$.

To analyze the approximation ratio for the algorithm c-**Makespan-Apx** on the large jobs $\alpha_l = \langle t_1, \ldots, t_{n_l} \rangle$, we follow exactly the same analysis given in the proof of Theorem 6.2.2 except that we replace n, the total number of jobs in the input, by n_l, the total number of large jobs. This analysis gives

$$\frac{Apx(\alpha_l)}{Opt(\alpha_l)} \leq 1 + \frac{K n_l}{Opt(\alpha_l)},$$

where $Apx(\alpha_l)$ is the makespan of the schedule constructed by the algorithm c-**Makespan-Apx** for the large jobs α_l, while $Opt(\alpha_l)$ is the makespan of an optimal schedule for the large jobs α_l. By the inequalities $n_l \leq c/\epsilon$ and $Opt(\alpha_l) \geq T_0'/c$, we obtain

$$\frac{Apx(\alpha_l)}{Opt(\alpha_l)} \leq 1 + \epsilon.$$

Note that the optimal makespan for the large job set α_l cannot be larger than the optimal makespan for the original set $\alpha = \langle t_1, \ldots, t_n \rangle$ of jobs. Therefore, if we let $Opt(\alpha)$ be the optimal makespan for the original job set α, then we have

$$Apx(\alpha_l) \leq Opt(\alpha_l)(1+\epsilon) \leq Opt(\alpha)(1+\epsilon) = Opt(\alpha) + \epsilon \cdot Opt(\alpha).$$

Now we are ready to describe an approximation algorithm for the c-Makespan problem: given an instance $\alpha = \langle t_1, \ldots, t_n \rangle$ for c-Makespan, (1) construct, in time $O(1/\epsilon^{2c-1})$, a schedule $\mathcal{S}_l$ of makespan bounded by $Opt(\alpha) + \epsilon \cdot Opt(\alpha)$ for the set α_l of large jobs; (2) assign the small jobs by a greedy method, i.e., we assign each small job (in arbitrary order) to the most lightly loaded processor. The assignment of the small jobs can be easily done in time $O(n)$ (note that the number c of processors is a fixed constant and that the number of small jobs is bounded by n). Thus, the running

time of this approximation algorithm is $O(n + 1/\epsilon^{2c-1})$. We claim that this algorithm has an approximation ratio bounded by $1 + \epsilon$. Let $Apx(\alpha)$ be the makespan for the schedule constructed by this algorithm.

Suppose that processor d has the longest running time $T_d = Apx(\alpha)$. Consider the last job t_i assigned to processor d. If t_i is a large job, then the processor d is assigned no small jobs. Thus, T_d is the makespan of the scheduling $\mathcal{S}_l$ for the large jobs α_l. By the above analysis,

$$Apx(\alpha) = T_d = Apx(\alpha_l) \leq Opt(\alpha)(1 + \epsilon).$$

On the other hand, if t_i is a small job, then $t_i < \epsilon T_0/c$ and by the greedy method, all c processors have running time at least $T_d - t_i$. Therefore, $\sum_{i=1}^{n} t_i \geq c(T_d - t_i)$ and $Opt(\alpha) \geq T_d - t_i$. This gives (note $T_0/c \leq Opt(\alpha)$)

$$Apx(\alpha) = T_d \leq Opt(\alpha) + t_i \leq Opt(\alpha) + \epsilon T_0/c \leq Opt(\alpha)(1 + \epsilon).$$

Therefore, in any case, the ratio $Apx(\alpha)/Opt(\alpha)$ is bounded by $1 + \epsilon$.

We summarize the above discussion into the following theorem.

Theorem 6.3.5 *There is a fully polynomial time approximation scheme for* c-MAKESPAN *that, given an instance* α *and an* $\epsilon > 0$*, constructs, in time* $O(n + 1/\epsilon^{2c-1})$*, a schedule for* α *of approximation ratio bounded by* $1 + \epsilon$*.*

Note that the fully polynomial time approximation scheme for the c-MAKESPAN problem in Theorem 6.3.5 runs in linear time when the error bound $\epsilon > 0$ is a fixed constant.

The technique can also be applied to the KNAPSACK problem. With a more complex analysis, it can be shown that there is a fully polynomial time approximation scheme for the KNAPSACK problem that, given an instance α of KNAPSACK and an $\epsilon > 0$, constructs, in time $O(n/\epsilon^2)$, a subset of α whose size is bounded by B and value is at least $Opt(\alpha)/(1 + \epsilon)$.

Further improvements on the KNAPSACK problem are possible. For example, with a more careful treatment of the large and small items, one can develop a fully polynomial time approximation scheme for the KNAPSACK problem of running time $O(n/\log(1/\epsilon) + 1/\epsilon^4)$. Interested readers are referred to [92] for detailed discussions.

6.4 Which problems have no FPTAS?

Fully polynomial time approximation schemes seem the best we can expect for NP-hard optimization problems. An NP-hard optimization problem with

a fully polynomial time approximation scheme can be approximated to a ratio $1+\epsilon$ for any $\epsilon > 0$ within a reasonable computational time, which is bounded by a polynomial of the input length and $1/\epsilon$. We have seen that several NP-hard optimization problems, such as KNAPSACK and c-MAKESPAN, have fully polynomial time approximation schemes.

A natural question is whether every NP-hard optimization problem has a fully polynomial time approximation scheme. If not, how do we determine if or not a given NP-hard optimization problem has a fully polynomial time approximation scheme. We discuss this isse in the current section.

Definition 6.4.1 Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem. For each instance $x \in I_Q$, let $Opt_Q(x) = opt_Q\{f_Q(x,y)|y \in S_Q(x)\}$, i.e., $Opt_Q(x)$ is the value of the objective function f_Q on instance x and an optimal solution to x.

The following theorem provides a very convenient and sufficient condition for an NP-hard optimization problem to have no fully polynomial time approximation schemes.

Theorem 6.4.1 *Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem. If there is a fixed polynomial p such that for all instances $x \in I_Q$, $Opt_Q(x)$ is bounded by $p(|x|)$, then Q does not have a fully polynomial time approximation scheme unless Q can be precisely solved in polynomial-time.*

PROOF. Let A be an approximation algorithm that is a fully polynomial time approximation scheme for the optimization problem Q. We show that Q can be precisely solved in polynomial time.

By the definition, we assume that the running time of A is bounded by $O(n^c/\epsilon^d)$, where c and d are constants. Moreover, by the condition given in the theorem, we assume that $Opt_Q(x) \leq n^h$, where h is also a constant.

First consider the case $opt_Q = \min$. For an instance $x \in I_Q$, let $A(x)$ be the objective function value on the instance x and the solution to x constructed by the algorithm A. Thus, we know that for any $\epsilon > 0$, the algorithm A constructs in time $O(n^c/\epsilon^d)$ a solution with approximation ratio $A(x)/Opt(x) \leq 1+\epsilon$. Also note that $A(x)/Opt(x) \geq 1$.

Now, let $\epsilon = 1/n^{h+1}$, then the algorithm A constructs a solution with approximation ratio bounded by

$$1 \leq \frac{A(x)}{Opt(x)} \leq 1 + \frac{1}{n^{h+1}},$$

which gives

$$Opt(x) \leq A(x) \leq Opt(x) + Opt(x)/n^{h+1}$$

Since both $Opt(x)$ and $A(x)$ are integers, and $Opt(x) \leq n^h$ implies that $Opt(x)/n^{h+1}$ is a number strictly less than 1, we conclude that

$$Opt(x) = A(x).$$

That is, the algorithm A actually constructs an optimal solution to the instance x. Moreover, the running time of A is bounded by $O(n^c/(1/n^{h+1})^d) = O(n^{c+hd+d})$, which is a polynomial of n.

The case that $opt_Q = \max$ can be proved similarly. Note that in this case, we should also have $A(x) \leq n^h$. Thus, in time $O(n^c/(1/n^{h+1})^d) = O(n^{c+hd+d})$, the algorithm A constructs a solution to x with the value $A(x)$ such that

$$1 \leq \frac{Opt(x)}{A(x)} \leq 1 + \frac{1}{n^{h+1}},$$

which gives

$$A(x) \leq Opt(x) \leq A(x) + A(x)/n^{h+1}.$$

Now since $A(x)/n^{h+1} < 1$, we conclude $Opt(x) = A(x)$. □

In particular, Theorem 6.4.1 says that if $Opt_Q(x)$ is bounded by a polynomial of the input length $|x|$ and Q is known to be NP-hard, then Q does not have a fully polynomial time approximation scheme unless P = NP.

Theorem 6.4.1 is actually very powerful. Most NP-hard optimization problems satisfy the condition of the theorem, thus we can derive directly that these problems have no fully polynomial time approximation schemes. We will give a few examples below to illustrate the power of Theorem 6.4.1.

Consider the following problem:

INDEPENDENT SET

I_Q: the set of undirected graphs $G = (V, E)$

S_Q: $S_Q(G)$ is the set of subsets S of V such that no two vertices in S are adjacent

f_Q: $f_Q(G, S)$ is equal to the number of vertices in S

opt_Q: max

It is easy to apply Theorem 6.4.1 to show that INDEPENDENT SET has no fully polynomial time approximation scheme. In fact, the value of the

objective function is bounded by the number of vertices in the input graph G, which is certainly bounded by a polynomial of the input length $|G|$.

There are many other graph problems (actually, most graph problems) like the INDEPENDENT SET problem that ask to optimize the size of a subset of vertices or edges of the input graph satisfying certain given properties. For all these problems, we can conclude directly from Theorem 6.4.1 that they do not have a fully polynomial time approximation scheme unless they can be solved precisely in polynomial time.

Let us consider another example of a problem for which no fully polynomial time approximation scheme exists.

BIN PACKING

INPUT: $\langle t_1, t_2, \ldots, t_n; B \rangle$, all integers and $t_i \leq B$ for all i

OUTPUT: a packing of the n objects of size t_1, ..., t_n into the minimum number of bins of size B

It is pretty easy to prove that the NP-complete problem PARTITION is polynomial time reducible to the BIN PACKING problem. Thus, the BIN PACKING problem is NP-hard. The BIN PACKING problem can be interpreted as a scheduling problem in which n jobs of processing time t_1, ..., t_n and the makespan B (i.e., the deadline) are given, we are looking for a schedule of the jobs so that the number of processors used in the schedule is minimized. Since $t_i \leq B$ for all i, we know that at most n bins are needed to pack the n objects. Thus, $Opt(x) \leq n$ for all input instances x of n objects. By Theorem 6.4.1, we conclude directly that the BIN PACKING problem has no fully polynomial time approximation scheme unless P = NP.

What if the condition of Theorem 6.4.1 does not hold? Can we still derive a claim of nonexistence of a fully polynomial time approximation scheme for an optimization problem? We study this question with the famous TRAVELING SALESMAN problem, and will derive general rules for this kind of optimization problems.

TRAVELING SALESMAN

INPUT: a weighted complete graph G

OUTPUT: a simple cycle containing all vertices of G (such a simple cycle is called a *traveling salesman tour*) and the weight of the cycle is minimized

The TRAVELING SALESMAN problem obviously does not satisfy the conditions of Theorem 6.4.1. For example, if all edges of the input graph G of n

vertices have weight of order $\Theta(2^n)$, then the weight of the minimum traveling salesman tour is $\Omega(n2^n)$ while a binary representation of the input graph G has length bounded by $O(n^3)$ (note: the length of the binary representation of a number of order $\Theta(2^n)$ is $O(n)$ and G has $O(n^2)$ edges). Therefore, Theorem 6.4.1 does not apply to the TRAVELING SALESMAN problem.

To show the non-approximability of the TRAVELING SALESMAN problem, we consider a simpler version of the TRAVELING SALESMAN problem, which is defined as follows.

TRAVELING SALESMAN 1-2

INPUT: a weighted complete graph G such that the weight of each edge of G is either 1 or 2

OUTPUT: a traveling salesman tour of minimum weight

Lemma 6.4.2 *The* TRAVELING SALESMAN 1-2 *problem is* NP*-hard.*

PROOF. We show that the well-known NP-complete problem HAMILTONIAN CIRCUIT is polynomial time reducible to the TRAVELING SALESMAN 1-2 problem.

By the definition, for each undirected unweighted graph G of n vertices, the HAMILTONIAN CIRCUIT problem asks if G contains a Hamiltonian circuit, i.e., a simple cycle of length n (for more discussion of the problem, the reader is referred to [53]).

Given an instance $G = (V, E)$ of the HAMILTONIAN CIRCUIT problem, we add edges to G to make a weighted complete graph $G' = (V, E \cup E')$ such that for each edge $e \in E$ of G' that is in the original graph G, we assign a weight 1 and for each edge $e' \in E'$ of G' that is not in the original graph G, we assign a weight 2. The graph G' is certainly an input instance of the TRAVELING SALESMAN 1-2 problem. Now, let T be a minimum weighted traveling salesman tour in G'. It is easy to verify that the weight of T is equal to n if and only if the original graph G contains a Hamiltonian circuit.

This completes the proof. □

Theorem 6.4.1 can apply to the TRAVELING SALESMAN 1-2 problem.

Lemma 6.4.3 *The* TRAVELING SALESMAN 1-2 *problem has no fully polynomial time approximation scheme unless* P = NP*.*

PROOF. Since the weight of a traveling salesman tour for an instance G of TRAVELING SALESMAN 1-2 is at most $2n$, assuming that G has n vertices,

the condition of Theorem 6.4.1 is satisfied by TRAVELING SALESMAN 1-2. Now the theorem follows from Theorem 6.4.1 and Lemma 6.4.2. □

Now we are ready for a conclusion on the approximability of the TRAVELING SALESMAN problem in its general form.

Theorem 6.4.4 *The* TRAVELING SALESMAN *problem has no fully polynomial time approximation scheme unless* P = NP.

PROOF. Since each instance for the TRAVELING SALESMAN 1-2 problem is also an instance for the TRAVELING SALESMAN problem, a fully polynomial time approximation scheme for the TRAVELING SALESMAN problem should also be a fully polynomial time approximation scheme for the TRAVELING SALESMAN 1-2 problem. Now the theorem follows from Lemma 6.4.3. □

Theorem 6.4.4 illustrates a general technique for proving the nonexistence of fully polynomial time approximation schemes for NP-hard optimization problems when Theorem 6.4.1 is not applicable. We formulate it as follows.

Let $Q = (I_Q, S_Q, f_Q, opt_Q)$ be an optimization problem. Recall that for each instance x of Q, length(x) is the length of a binary representation of x and $\max(x)$ is the largest number that appears in x.

Definition 6.4.2 Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem and let q be any function. A subproblem Q' of Q is a Q_q-*subproblem* if $Q' = \langle I'_Q, S_Q, f_Q, opt_Q \rangle$ such that $I'_Q \subseteq I_Q$ and for all $x \in I'_Q$, $\max(x) \leq q(\text{length}(x))$.

The following definition was first introduced and studied by Garey and Johnson [53].

Definition 6.4.3 An optimization problem $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ is NP-*hard in the strong sense* if a Q_q-subproblem of Q is NP-hard for some polynomial q.

The concept of the strong NP-hardness can be naturally extended to decision problems.

The TRAVELING SALESMAN problem is an example of problems that are NP-hard in the strong sense, as shown by the following theorem.

Theorem 6.4.5 TRAVELING SALESMAN *is* NP*-hard in the strong sense.*

PROOF. If we denote by Q the TRAVELING SALESMAN problem, then TRAVELING SALESMAN 1-2 is a Q_2-subproblem of Q. By Lemma 6.4.2, TRAVELING SALESMAN 1-2 is NP-hard. Now by the above definition, TRAVELING SALESMAN is NP-hard in the strong sense. □

If the condition $\max(x) \leq p(\text{lengh}(x))$ for some fixed polynomial p is satisfied for all instances x of an NP-hard optimization problem Q, then Q is NP-hard in the strong sense. Note that for many NP-hard optimization problems, in particular for many NP-hard optimization problems for which the condition of Theorem 6.4.1 is satisfied, the condition $\max(x) \leq p(\text{lengh}(x))$ is satisfied trivially. Thus, these NP-hard optimization problems are also NP-hard in the strong sense. On the other hand, there are many other NP-hard optimization problems that are not NP-hard in the strong sense. In particular, we have the following theorem.

Theorem 6.4.6 *If an optimization problem Q has a pseudo-polynomial time algorithm, then Q is not* NP*-hard in the strong sense unless* P = NP*.*

PROOF. Suppose that Q has a pseudo-polynomial time algorithm A that on an instance x of Q, constructs an optimal solution to x in time $O((\text{length}(x))^c(\max(x))^d)$ for some constants c and d.

If Q is NP-hard in the strong sense, then there is a Q_p-subproblem Q' of Q for some fixed polynomial p such that Q' is also NP-hard. However, for all instances x of Q', $\max(x) \leq p((length)(x))$. Thus, the algorithm A constructs an optimal solution for each instance x of Q' in time

$$O((\text{length}(x))^c(\max(x))^d) = O((\text{length}(x))^c(p(\text{length}(x)))^d),$$

which is bounded by a polynomial of length(x). Thus, the NP-hard optimization problem Q' can be solved by the polynomial time algorithm A, which implies P = NP. □

Theorem 6.4.6 combined with Theorems 6.1.3-6.1.4 gives the following:

Corollary 6.4.7 *The* KNAPSACK *problem and the* c*-*MAKESPAN *problem are not* NP*-hard in the strong sense unless* P = NP*.*

The following theorem serves as a fundamental theorem for proving the nonexistence of fully polynomial time approximation schemes for an NP-hard

optimization problem, in particular when Theorem 6.4.1 is not applicable. A two-parameter function $f(x, y)$ is a *polynomial of x and y* if $f(x, y)$ is a finite sum of the terms of form $x^c y^d$, where c and d are non-negative integers.

Theorem 6.4.8 *Let Q be an optimization problem that is* NP*-hard in the strong sense. Suppose that for all instances x of Q, $Opt_Q(x)$ is bounded by a polynomial of* length(x) *and* max(x)*. Then Q has no fully polynomial time approximation scheme unless* P = NP*.*

Proof. The proof of this theorem is very similar to the discussion we have given for the Traveling Salesman problem.

Since Q is NP-hard in the strong sense, a Q_q-subproblem Q' is NP-hard for a polynomial q. Let $Q' = \langle I'_Q, S_Q, f_Q, opt_Q \rangle$. Then for each instance $x \in I'_Q$, we have $\max(x) \leq q(\text{length}(x))$. Combining this condition with the condition stated in the theorem that $Opt_Q(x)$ is bounded by a polynomial of length(x) and max(x), we derive that $Opt_Q(x)$ is bounded by a polynomial of length(x) for all instances $x \in I'_Q$. Now by Theorem 6.4.1, the problem Q' has no fully polynomial time approximation scheme unless P = NP. Since each instance of Q' is also an input instance of Q, a fully polynomial time approximation scheme for Q is also a fully polynomial time approximation scheme for Q'. Now the theorem follows. □

Remark. How common can $Opt_Q(x)$ be bounded by a polynomial of length(x) and max(x)? In fact, this situation is fairly common because for most optimization problems, the objective function value is defined through additions or constant number of multiplications on the numbers appearing in the instance x, which is certainly bounded by a polynomial of length(x) and max(x). Of course, the condition is not universely true for general optimization problems. For example, an objective function can be simply defined to be the exponentiation of the sum of a subset of input values, which cannot be bounded by any polynomial of length(x) and max(x).

In general, it is easy to verify the condition that $Opt_Q(x)$ is bounded by a polynomial of length(x) and max(x). Therefore, in order to apply Theorem 6.4.8, we need to prove the strong NP-hardness for a given optimization problem Q. There are two general techniques serving for this purpose. The first one is to pick an NP-complete problem L and show that L is polynomial time reducible to a Q_q-subproblem of Q for some polynomial q. Our polynomial time reduction from Hamiltonian Circuit to Traveling Salesman 1-2, which leads to the strong NP-hardness of the general Traveling Salesman problem (Theorem 6.4.5), well illustrates this technique.

The second technique is to develop a polynomial time reduction from a known strongly NP-hard optimization problem R to the given optimization problem Q. For this, we also require that for each polynomial p, the reduction transforms a R_p-subproblem of R into a Q_q-subproblem of Q for some polynomial q. We explain this technique by showing that the following familiar optimization problem is NP-hard in the strong sense.

MAKESPAN

I_Q: the set of tuples $T = \{t_1, \ldots, t_n; m\}$, where t_i is the processing time for the ith job and m is the number of identical processors

S_Q: $S_Q(T)$ is the set of partitions $P = (T_1, \ldots, T_m)$ of the numbers $\{t_1, \ldots, t_n\}$ into m parts

f_Q: $f_Q(T, P)$ is equal to the processing time of the largest subset in the partition P: $f_Q(T, P) = \max_i\{\sum_{t_j \in T_i} t_j\}$

opt_Q: min

To show that MAKESPAN is NP-hard in the strong sense, we reduce the following strongly NP-hard (decision) problem to MAKESPAN.

THREE-PARTITION

INPUT: $\{t_1, t_2, \ldots, t_{3m}; B\}$, all integers, where $B/4 \leq t_i \leq B/2$ for all i, and $\sum_{i=1}^{3m} t_i = mB$

QUESTION: Can $\{t_1, \ldots, t_{3m}\}$ be partitioned into m sets, each of size B?

The THREE-PARTITION problem has played a fundamental role in proving strong NP-hardness for many scheduling problems. For a proof that the THREE-PARTITION problem is NP-hard in the strong sense, the reader is referred to Garey and Johnson's authoritative book [53], Section 4.2.2.

The reduction $\mathcal{R}$ from the THREE-PARTITION problem to the MAKESPAN problem is straightforward: given an instance $\alpha = \{t_1, t_2, \ldots, t_{3m}; B\}$ of THREE-PARTITION, we construct an instance $\beta = \{t_1, t_2, \ldots, t_{3m}; m\}$ for MAKESPAN. It is clear that the optimal makespan for the instance β for MAKESPAN is B if and only if α is a yes-instance for THREE-PARTITION. Note that the input length length(β) is at least 1/2 times the input length length(α) (in fact, length(α) and length(β) are roughly equal), and that $\max(\beta)$ is bounded by $\max(\alpha) + \text{length}(\alpha)$.

Since the THREE-PARTITION problem is NP-hard in the strong sense, there is a polynomial q such that a (THREE-PARTITION)$_q$-subproblem R'

of THREE-PARTITION is NP-hard. Let Q' be a subproblem of MAKESPAN such that Q' consists of instances of the form $\{t_1, t_2, \ldots, t_{3m}; m\}$, where $\sum_{i=1}^{3m} t_i = mB$ and $\{t_1, t_2, \ldots, t_{3m}; B\}$ is an instance of R'. Therefore, the polynomial time reduction $\mathcal{R}$ reduces the problem R' to the problem Q'. Therefore, the problem Q' is NP-hard. Moreover, for each instance α of R', we have $\max(\alpha) \leq q(\text{length}(\alpha))$. Now for each instance β of Q' that is the image of an instance α of R' under the reduction $\mathcal{R}$, we have

$$\begin{aligned} \max(\beta) &\leq \max(\alpha) + \text{length}(\alpha) \leq q(\text{length}(\alpha)) + \text{length}(\alpha) \\ &\leq q(2 \cdot \text{length}(\beta)) + 2 \cdot \text{length}(\beta). \end{aligned}$$

Therefore, the NP-hard problem Q' is a $(\text{MAKESPAN})_p$-subproblem of the MAKESPAN problem, where p is a polynomial. This proves that the MAKESPAN problem is NP-hard in the strong sense.

It is trivial to verify that the other conditions of Theorem 6.4.8 are satisfied by the MAKESPAN problem. Thus,

Theorem 6.4.9 *The* MAKESPAN *problem is* NP*-hard in the strong sense. Moreover, the* MAKESPAN *problem has no fully polynomial time approximation scheme unless* P = NP.

We should point out that the c-MAKESPAN problem, i.e., the MAKESPAN problem with the number of processors being fixed by a constant c, has a fully polynomial time approximation scheme (Corollary 6.2.3). However, if the number m of processors is given as a variable in the input, then the problem becomes NP-hard in the strong sense and has no fully polynomial time approximation scheme.