Chapter 12

APPROXIMATION ALGORITHMS FOR NP-HARD PROBLEMS

12.1 INTRODUCTION

In the preceding chapter we saw strong evidence to support the claim that no NP-hard problem can be solved in polynomial time. Yet, many NP-hard optimization problems have great practical importance and it is desirable to solve large instances of these problems in a "reasonable" amount of time. The best known algorithms for NP-hard problems have a worst case complexity that is exponential in the number of inputs. While the results of the last chapter may favor abandoning the quest for polynomial time algorithms, there is still plenty of room for improvement in an exponential algorithm. We may look for algorithms with subexponential complexity, say $2^{n/c}$ (for $c > 1$), $2^{\sqrt{n}}$ or $n^{\log n}$. In the exercises of Chapter 5 an $O(2^{n/2})$ algorithm for the knapsack problem was developed. This algorithm can also be used for the partition, sum of subsets and exact cover problem. Tarjan and Trojanowski ("Finding a maximum independent set," *SIAM Computing*, 6(3), pp. 537–546, 1977.) have obtained an $O(2^{n/3})$ algorithm for the max-clique, max-independent set and minimum node cover problems. The discovery of a subexponential algorithm for an NP-hard problem increases the maximum problem size that can actually be solved. However, for large problem instances, even an $O(n^4)$ algorithm requires too much computational effort. Clearly, what is needed is an algorithm of low polynomial complexity (say $O(n)$ or $O(n^2)$).

The use of heuristics in an existing algorithm may enable it to quickly solve a large instance of a problem provided the heuristic "works" on that instance. This was clearly demonstrated in the chapters on backtracking and branch-and-bound. A heuristic, however, does not "work" equally effectively on all problem instances. Exponential time algorithms, even coupled with heuristics will still show exponential behavior on some set of inputs.

If we are to produce an algorithm of low polynomial complexity to solve an NP-hard optimization problem, then it will be necessary to relax the meaning of solve. In this chapter we shall discuss two relaxations of the meaning of solve. In the first we shall remove the requirement that the algorithm that solves the optimization problem P must always generate an optimal solution. This requirement will be replaced by the requirement that the algorithm for P must always generate a feasible solution with value "close" to the value of an optimal solution. A feasible solution with value close to the value of an optimal solution is called an *approximate solution.* An *approximation algorithm* for P is an algorithm that generates approximate solutions for P.

While at first one may discount the virtue of an approximate solution, one should bear in mind that often, the data for the problem instance being solved is only known approximately. Hence, an approximate solution (provided its value is "sufficiently" close to that of an exact solution) may be no less meaningful than an exact solution. In the case of NP-hard problems approximate solutions have added importance as it may be true that exact solutions (i.e. optimal solutions) cannot be obtained in a feasible amount of computing time. An approximate solution may be all one can get using a reasonable amount of computing time.

In the second relaxation we shall look for an algorithm for P that *almost always* generates optimal solutions. Algorithms with this property are called *probabilistically good* algorithms. These are considered in Section 12.6. In the remainder of this section we develop the terminology to be used in discussing approximation algorithms.

Let P be a problem such as the knapsack or the traveling salesperson problem. Let I be an instance of problem P and let $F^*(I)$ be the value of an optimal solution to I. An approximation algorithm will in general produce a feasible solution to I whose value $\hat{F}(I)$ is less than (greater than) $F^*(I)$ in case P is a maximization (minimization) problem. Several categories of approximation algorithms may be defined.

Let $\mathcal{A}$ be an algorithm which generates a feasible solution to every instance I of a problem P. Let $F^*(I)$ be the value of an optimal solution to I and let $\hat{F}(I)$ be the value of the feasible solution generated by $\mathcal{A}$.

Definition $\mathcal{A}$ is an *absolute approximation* algorithm for problem P if and only if for every instance I of P, $|F^*(I) - \hat{F}(I)| \leq k$ for some constant k.

Definition $\mathcal{A}$ is an *f(n)-approximate* algorithm if and only if for every

instance I of size n, $|F^*(I) - \hat{F}(I)|/F^*(I) \leq f(n)$. It is assumed that $F^*(I) > 0$.

Definition An *ϵ-approximate* algorithm is an $f(n)$-approximate algorithm for which $f(n) \leq \epsilon$ for some constant ϵ.

Note that for a maximization problem, $|F^*(I) - \hat{F}(I)|/F^*(I) \leq 1$ for every feasible solution to I. Hence, for maximization problems we will normally require $\epsilon < 1$ for an algorithm to be judged ϵ-approximate. In the next few definitions we consider algorithms $\mathcal{A}(\epsilon)$ with ϵ an input to $\mathcal{A}$.

Definition $\mathcal{A}(\epsilon)$ is an *approximation scheme* iff for every given $\epsilon > 0$ and problem instance I, $\mathcal{A}(\epsilon)$ generates a feasible solution such that $|F^*(I) - \hat{F}(I)|/F^*(I) \leq \epsilon$. Again, we assume $F^*(I) > 0$.

Definition An approximation scheme is a *polynomial time approximation scheme* iff for every fixed $\epsilon > 0$ it has a computing time that is polynomial in the problem size.

Definition An approximation scheme whose computing time is a polynomial both in the problem size and in $1/\epsilon$ is a *fully polynomial time approximation scheme.*

Clearly, the most desirable kind of approximation algorithm is an absolute approximation algorithm. Unfortunately, for most NP-hard problems it can be shown that fast algorithms of this type exist only if $P =$ NP. Surprisingly, this statement is true even for the existence of $f(n)$-approximate algorithms for certain NP-hard problems.

Example 12.1 Consider the knapsack instance $n = 3$, $M = 100$, $\{p_1, p_2, p_3\} = \{20, 10, 19\}$ and $\{w_1, w_2, w_3\} = \{65, 20, 35\}$. $(x_1, x_2, x_3) = (1, 1, 1)$ is not a feasible solution as $\Sigma\, w_i x_i > M$. The solution $(x_1, x_2, x_3) = (1, 0, 1)$ is an optimal solution. Its value $\Sigma\, p_i x_i$ is 39. Hence, $F^*(I) = 39$ for this instance. The solution $(x_1, x_2, x_3) = (1, 1, 0)$ is suboptimal. Its value is $\Sigma\, p_i x_i = 30$. This is a candidate for a possible output from an approximation algorithm. In fact, every feasible solution (in this case all three element 0/1 vectors other than (1, 1, 1) are feasible) is a candidate for output by an approximation algorithm. If the solution (1, 1, 0) is generated by an approximation algorithm on this instance then $\hat{F}(I) = 30$. $|F^*(I) - \hat{F}(I)| = 9$ and $|F^*(I) - \hat{F}(I)|/F^*(I) = 0.3$. □

Example 12.2 Consider the following approximation algorithm for the 0/1 knapsack problem: consider the objects in nonincreasing order of p_i/w_i. If object i fits then set $x_i = 1$ otherwise set $x_i = 0$. When this algorithm is used on the instance of Example 12.1, the objects are considered in the order 1, 3, 2. The result is $(x_1, x_2, x_3) = (1, 0, 1)$. The optimal solution is obtained. Now, consider the following instance: $n = 2$, $(p_1, p_2) = (2, r)$, $(w_1, w_2) = (1, r)$ and $M = r$. When $r > 1$, the optimal solution is $(x_1, x_2) = (0, 1)$. Its value, $F^*(I)$, is r. The solution generated by the approximation algorithm is $(x_1, x_2) = (1, 0)$. Its value, $\hat{F}(I)$, is 2. Hence, $|F^*(I) - \hat{F}(I)| = r - 2$. Our approximation algorithm is not an absolute approximation algorithm as there exists no constant k such that $|F^*(I) - \hat{F}(I)| \leq k$ for all instances I. Furthermore, note that $|F^*(I) - \hat{F}(I)|/F^*(I) = 1 - 2/r$. This approaches 1 as r becomes large. $|F^*(I) - \hat{F}(I)|/F^*(I) \leq 1$ for every feasible solution to every knapsack instance. Since the above algorithm always generates a feasible solution it is a 1-approximate algorithm. It is, however, not an ϵ-approximate algorithm for any ϵ, $\epsilon < 1$. □

Corresponding to the notions of absolute approximation algorithm and $f(n)$-approximate algorithm, we may define approximation problems in the obvious way. So, we can speak of k-absolute approximate problems and $f(n)$-approximate problems. The .5-approximate knapsack problem is to find any 0/1 feasible solution with $|F^*(I) - \hat{F}(I)|/F^*(I) \leq .5$.

As we shall see, approximation algorithms are usually just heuristics or rules that on the surface look like they might solve the optimization problem exactly. However, they do not. Instead, they only guarantee to generate feasible solutions with value within some constant or some factor of the optimal value. Being heuristic in nature, these algorithms are very much dependent on the individual problem being solved.

12.2 ABSOLUTE APPROXIMATIONS

Planar Graph Coloring

There are very few NP-hard optimization problems for which polynomial time absolute approximation algorithms are known. One problem is that of determining the minimum number of colors needed to color a planar graph $G = (V, E)$. It is known that every planar graph is four colorable. One may easily determine if a graph is 0, 1 or 2 colorable. It is zero colorable iff $V = \phi$. It is 1 colorable iff $E = \phi$. G is two colorable iff it is bi partite (see Exercise 6.41). Determining if a planar graph is three colorable

is NP-hard. However, all planar graphs are four colorable. An absolute approximation algorithm with $|F^*(I) - \hat{F}(I)| \leq 1$ is easy to obtain. Algorithm 12.1 is such an algorithm. It finds an exact answer when the graph can be colored using at most two colors. Since we can determine whether or not a graph is bipartite in time $O(|V| + |E|)$, the complexity of the algorithm is $O(|V| + |E|)$.

```
procedure ACOLOR(V, E)
  //determine an approximation to the minimum number of colors//
  //needed to color the planar graph G = (V, E)//

    case
      : V = φ: return (0)
      : E = φ: return (1)
      : G is bipartite: return (2)
      : else: return (4)
    endcase
end ACOLOR
```

Algorithm 12.1 Approximate coloring

Maximum Programs Stored Problem

Assume that we have n programs and two storage devices (say disks or tapes). We shall assume the devices are disks. Our discussion applies to any kind of storage device. Let l_i be the amount of storage needed to store the ith program. Let L be the storage capacity of each disk. Determining the maximum number of these n programs that can be stored on the two disks (without splitting a program over the disks) is NP-hard.

Theorem 12.1 Partition α Maximum Programs Stored.

Proof: Let $\{a_1, a_2, \ldots, a_n\}$ define an instance of the partition problem. We may assume $\Sigma\, a_i = 2T$. Define an instance of the maximum programs stored problem as follows: $L = T$ and $l_i = a_i$, $1 \leq i \leq n$. Clearly, $\{a_i, \ldots, a_n\}$ has a partition iff all n programs can be stored on the two disks. □

By considering programs in order of nondecreasing storage requirement l_i, we can obtain a polynomial time absolute approximation algorithm. Procedure PSTORE assumes $l_1 \leq l_2 \leq \cdots \leq l_n$ and assigns programs

to disk 1 so long as enough space remains on this tape. Then it begins assigning programs to disk 2. In addition to the time needed to initially sort the programs into nondecreasing order of l_i, $O(n)$ time is needed to obtain the storage assignment.

```
procedure PSTORE(l, n, L)
  //assume l_i ≤ l_{i+1}, 1 ≤ i < n//
    i ← 1
    for j ← 1 to 2 do
      sum ← 0  //amount of disk j already assigned//
      while sum + l_i ≤ L do
        print ('store program', i, 'on disk', j)
        sum ← sum + l_i
        i ← i + 1
        if i > n then return endif
      repeat
    repeat
end PSTORE
```

Algorithm 12.2 Approximation algorithm to store programs

Example 12.3 Let $L = 10$, $n = 4$ and $(l_1, l_2, l_3, l_4) = (2, 4, 5, 6)$. Procedure PSTORE will store programs 1 and 2 on disk 1 and only program 3 on disk 2. An optimal storage scheme stores all four programs. One way to do this is to store programs 1 and 4 on disk 1 and the other two on disk 2. □

Theorem 12.2 Let I be any instance of the maximum programs stored problem. Let $F^*(I)$ be the maximum number of programs that can be stored on two disks of length L each. Let $\hat{F}(I)$ be the number of programs stored using procedure PSTORE. Then, $|F^*(I) - \hat{F}(I)| \leq 1$.

Proof: Assume that k programs are stored when Algorithm 12.2 is used. Then, $\hat{F}(I) = k$. Consider the program storage problem when only one disk of capacity $2L$ is available. In this case, considering programs in order of nondecreasing storage requirement maximizes the number of programs stored. Assume that p programs get stored when this strategy is used on a single disk of length $2L$. Clearly, $p \geq F^*(I)$ and $\sum_1^p l_i \leq 2L$. Let j be the largest index such that $\sum_1^j l_i \leq L$. It is easy to verify that $j \leq p$ and that PSTORE assigns the first j programs to disk 1. Also,

$$\sum_{i=j+1}^{p-1} l_i \leq \sum_{i=j+2}^{p} l_i \leq L.$$

Hence, PSTORE assigns at least programs $j + 1, j + 2, \ldots, p - 1$ to disk 2. So, $\hat{F}(I) \geq p - 1$ and $|F^*(I) - \hat{F}(I)| \leq 1$. □

Algorithm PSTORE may be extended in the obvious way to obtain a $k - 1$ absolute approximation algorithm for the case of k disks.

NP-hard Absolute Approximations

The absolute approximation algorithms for the planar graph coloring and the maximum program storage problems are very simple and straightforward. Thus, one may expect that polynomial time absolute approximation algorithms exist for most other NP-hard problems. Unfortunately, for the majority of NP-hard problems one can provide very simple proofs to show that a polynomial time absolute approximation algorithm exists iff a polynomial time exact algorithm does. Let us look at some sample proofs.

Theorem 12.3 The absolute approximate knapsack problem is NP-hard.

Proof: We shall show that the 0/1 knapsack problem with integer profits reduces to the absolute approximate knapsack problem. The theorem then follows from the observation that the knapsack problem with integer profits is NP-hard. Assume there is a polynomial time algorithm $\mathcal{A}$ that guarantees feasible solutions such that $|F^*(I) - \hat{F}(I)| \leq k$ for every instance I and a fixed k. Let (p_i, w_i), $1 \leq i \leq n$ and M define an instance of the knapsack problem. Assume the p_i are integer. Let I' be the instance defined by $((k + 1)p_i, w_i)$, $1 \leq i \leq n$ and M. Clearly, I and I' have the same set of feasible solutions. Further, $F^*(I') = (k + 1)F^*(I)$ and I and I' have the same optimal solutions. Also, since all the p_i are integer, it follows that all feasible solutions to I' either have value $F^*(I')$ or have value at most $F^*(I') - (k + 1)$. If $\hat{F}(I')$ is the value of the solution generated by $\mathcal{A}$ for instance I' then $F^*(I') - \hat{F}(I')$ is either 0 or at least k + 1. Hence if $F^*(I') - \hat{F}(I') \leq$ k then $F^*(I') = \hat{F}(I')$. So, $\mathcal{A}$ can be used to obtain an optimal solution for I' and hence I. Since the length of I' is at most (log k)*(length of I), it follows that using the above construction we can obtain a polynomial time algorithm for the knapsack problem with integer profits. □

Example 12.4 Consider the knapsack instance $n = 3$, $M = 100$, $(p_1, p_2,$

$p_3) = (1, 2, 3)$ and $(w_1, w_2, w_3) = (50, 60, 30)$. The feasible solutions are (1, 0, 0), (0, 1, 0), (0, 0, 1), (1, 0, 1) and (0, 1, 1). The values of these solutions are 1, 2, 3, 4 and 5 respectively. If we multiply the p's by 5 then $(\hat{p}_1, \hat{p}_2, \hat{p}_3) = (5, 10, 15)$. The feasible solutions are unchanged. Their values are now 5, 10, 15, 20 and 25 respectively. If we had an absolute approximation algorithm for $k = 4$ then, this algorithm will have to output the solution (0, 1, 1) as no other solution is within 4 of the optimal solution value. □

Now, consider the problem of obtaining a maximum clique of an undirected graph. The following theorem shows that obtaining a polynomial time absolute approximation algorithm for this problem is as hard as obtaining a polynomial time algorithm for the exact problem.

Theorem 12.4 Max clique $\propto$ absolute approximation max clique.

Proof: Assume that the algorithm for the absolute approximation problem finds solutions such that $|F^*(I) - \hat{F}(I)| \leq k$. From any given graph $G = (V, E)$, we construct another graph $G' = (V', E')$ such that G' consists of $k + 1$ copies of G connected together such that there is an edge between every two vertices in distinct copies of G. I.e., if $V = \{v_1, v_2, \ldots, v_n\}$ then

$$V' = \bigcup_{i=1}^{k+1} \{v_1^i, v_2^i, \ldots, v_n^i\}$$

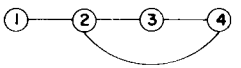
and

$$E' = \left(\bigcup_{i=1}^{k+1} \{(v_p^i, v_r^i) | (v_p, v_r) \in E\} \right) \cup \{(v_p^i, v_r^j) | i \neq j\}.$$

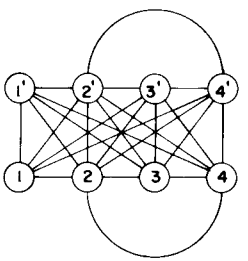
Clearly, the maximum clique size in G is q iff the maximum clique size in G' is $(k + 1)q$. Further, any clique in G' which is within k of the optimal clique size in G' must contain a sub-clique of size q which is a clique of size q in G. Hence, we can obtain a maximum clique for G from a k-absolute approximate maximum clique for G'. □

Example 12.5 Figure 12.1(b) shows the graph G' that results when the construction of Theorem 12.4 is applied to the graph of Figure 12.1(a). We have assumed $k = 1$. The graph of Figure 12.1(a) has two cliques.

One consists of the vertex set $\{1, 2\}$ and the other $\{2, 3, 4\}$. Thus, an absolute approximation algorithm for $k = 1$ could output either of the two as solution cliques. In the graph of Figure 12.1(b), however, the two cliques are $\{1, 2, 1', 2'\}$ and $\{2, 3, 4, 2', 3', 4'\}$. Only the latter may be output. Hence, an absolute approximation algorithm with $k = 1$ will output the maximum clique. □



(a)



(b)

Figure 12.1 Graphs for Example 12.5

12.3 ϵ-APPROXIMATIONS

Scheduling Independent Tasks

Obtaining minimum finish time schedules on m, $m \geq 2$ identical processors is NP-hard. There exists a very simple scheduling rule that generates schedules with a finish time very close to that of an optimal schedule. An instance I of the scheduling problem is defined by a set of n task times, t_i, $1 \leq i \leq n$, and m, the number of processors. The scheduling rule we are about to describe is known as the LPT (longest processing time) rule. An LPT schedule is a schedule that results from this rule.

Definition An *LPT schedule* is one that is the result of an algorithm which, whenever a processor becomes free, assigns to that processor a task whose time is the largest of those tasks not yet assigned. Ties are broken in an arbitrary manner.

Example 12.6 Let $m = 3$, $n = 6$ and $(t_1, t_2, t_3, t_4, t_5, t_6) = (8, 7, 6, 5, 4, 3)$. In an LPT schedule tasks 1, 2 and 3 are assigned to processors 1, 2 and 3 respectively. Tasks 4, 5 and 6 are respectively assigned to processors 3, 2 and 1. Figure 12.2 shows this LPT schedule. The finish time is 11. Since, $\Sigma\, t_i/3 = 11$, the schedule is also optimal.

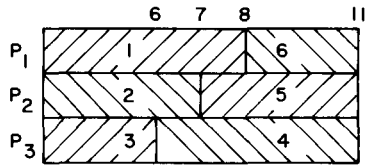


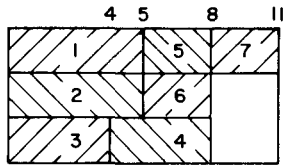
Figure 12.2 LPT schedule for Example 12.6

Example 12.7 Let $m = 3$, $n = 7$ and $(t_1, t_2, t_3, t_4, t_5, t_6, t_7) = (5, 5, 4, 4, 3, 3, 3)$. Figure 12.3(a) shows the LPT schedule. This has a finish time of 11. Figure 12.3(b) shows an optimal schedule. Its finish time is 9. Hence, for this instance $|F^*(I) - \hat{F}(I)|/F^*(I) = (11 - 9)/9 = 2/9$. □

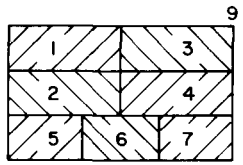
It is possible to implement the LPT rule so that at most 0(n log n) time is needed to generate an LPT schedule for n tasks on m processors. An exercise examines this. The preceding examples show that while the LPT rule may generate optimal schedules for some problem instances, it does not do so for all instances. How bad can LPT schedules be relative to optimal schedules? This question is answered by the following theorem.

Theorem 12.5 [Graham] Let $F^*(I)$ be the finish time of an optimal m processor schedule for instance I of the task scheduling problem. Let $\hat{F}(I)$ be the finish time of an LPT schedule for the same instance. Then,

$$\frac{|F^*(I) - \hat{F}(I)|}{F^*(I)} \leq \frac{1}{3} - \frac{1}{3m}$$



(a) LPT Schedule



(b) Optimal Schedule

Figure 12.3 LPT and optimal schedules for Example 12.7

Proof: The theorem is clearly true for $m = 1$. So, assume $m \geq 2$. Assume that for some m, $m > 1$, there exists a set of tasks for which the theorem is not true. Then, let $(t_1, t_2, \ldots, t_n)$ define an instance I with the fewest number of tasks for which the theorem is violated. We may assume $t_1 \geq t_2 \geq \cdots \geq t_n$ and that an LPT schedule is obtained by assigning tasks in the order $1, 2, 3, \ldots, n$.

Let S be the LPT schedule obtained by assigning these n tasks in this order. Let $\hat{F}(I)$ be its finish time. Let k be the index of a task with latest completion time. Then, $k = n$. To see this, suppose $k < n$. Then, the finish time $\hat{f}$ of the LPT schedule for tasks $1, 2, \ldots, k$ is also $\hat{F}(I)$. The finish time, f^*, of an optimal schedule for these k tasks is no more than $F^*(I)$. Hence, $|f^* - \hat{f}|/f^* \geq |F^*(I) - \hat{F}(I)|/F^*(I) > 1/3 - 1/(3m)$. (The latter inequality follows from the assumption on I.) $|f^* - \hat{f}|/f^* > 1/3 - 1/(3m)$ contradicts the assumption that I is the smallest m processor instance for which the theorem does not hold. Hence, $k = n$.

Now, we show that in no optimal schedule for I can more than two tasks be assigned to any processor. Hence, $n \leq 2m$. Since task n has the latest completion time in the LPT schedule for I, it follows that this task is started

at time $\hat{F}(I) - t_n$ in this schedule. Further, no processor can have any idle time until this time. Hence, we obtain:

$$\hat{F}(I) - t_n \leq \frac{1}{m} \sum_{1}^{n-1} t_i$$

So,

$$\hat{F}(I) \leq \frac{1}{m} \sum_{1}^{n} t_i + \frac{m-1}{m} t_n.$$

Since,

$$F^*(I) \geq \frac{1}{m} \sum_{1}^{n} t_i,$$

we can conclude that

$$\hat{F}(I) - F^*(I) \leq \frac{m-1}{m} t_n$$

or

$$\frac{|F^*(I) - \hat{F}(I)|}{F^*(I)} \leq \frac{m-1}{m} \frac{t_n}{F^*(I)}$$

But, from the assumption on I, the left hand side of the above inequality is greater than $1/3 - 1/(3m)$. So,

$$\frac{1}{3} - \frac{1}{3m} < \frac{m-1}{m} \frac{t_n}{F^*(I)}$$

or

$$m - 1 < 3(m-1) t_n / F^*(I)$$

or

$$F^*(I) < 3t_n.$$

Hence, in an optimal schedule for I, no more than two tasks can be assigned to any processor. When the optimal schedule contains at most two tasks on any processor then it may be shown that the LPT schedule is also optimal. We leave this part of the proof as an exercise. Hence, $|F^*(I) - \hat{F}(I)|/F^*(I) = 0$ for this case. This contradicts the assumption on I. So, there can be no I that violates the theorem. □

Theorem 12.5 establishes the LPT rule as a $(1/3 - 1/(3m))$-approximate rule for task scheduling. As remarked earlier, this rule can be implemented to have complexity $O(n \log n)$. The following example shows that $1/3 - 1/(3m)$ is a tight bound on the worst case performance of the LPT rule.

Example 12.8 Let $n = 2m + 1$, $t_i = 2m - \lfloor (i + 1)/2 \rfloor$, $i = 1, 2, \ldots, 2m$ and $t_{2m+1} = m$. Figure 12.4(a) shows the LPT schedule. This has a finish time of $4m - 1$. Figure 12.4(b) shows an optimal schedule. Its finish time is $3m$. Hence, $|F^*(I) - \hat{F}(I)|/F^*(I) = 1/3 - 1/(3m)$. □

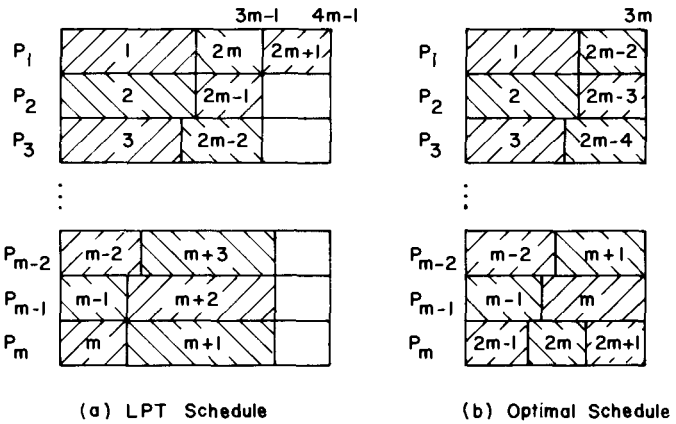


Figure 12.4 Schedules for Example 12.8

For LPT schedules, the worst case error bound of $1/3 - 1/(3m)$ is not very indicative of the expected closeness of LPT finish times to optimal finish times. When $m = 10$, the worst case error bound is .3. Two experiments were conducted ("An application of bin-packing to multiprocessor scheduling," by E. Coffman, M. Garey and D. Johnson, *SIAM Computing*,

7(1), pp. 1-17, 1978.) to see what kind of error one might expect on a random problem for $m = 10$. In the first experiment, 30 tasks with task times chosen according to a uniform distribution between 0 and 1 were generated. $F^*(I)$ was estimated to be $\Sigma_1^{30} t_i/10$ and $\hat{F}(I)$ was the length of the LPT schedule generated. The experiment was repeated ten times and the average value of $|F^*(I) - \hat{F}(I)|/F^*(I)$ computed. This value was 0.074. In the second experiment task times were chosen according to a normal distribution. The average $|F^*(I) - \hat{F}(I)|/F^*(I)$ was 0.023 this time. These figures are probably a little inflated as $\Sigma_1^{30} t_i/10$ is probably an underestimation of the true $F^*(I)$.

Efficient ϵ-approximate algorithms exist for many scheduling problems. The references at the end of this chapter point to some of the better known ϵ-approximate scheduling algorithms. Some of these algorithms are also discussed in the exercises.

Bin Packing

In this problem we are given n objects which have to be placed in bins of equal capacity L. Object i requires l_i units of bin capacity. The objective is to determine the minimum number of bins needed to accommodate all n objects. No object may be placed partly in one bin and partly in another.

Example 12.9 Let $L = 10$, $n = 6$ and $(l_1, l_2, l_3, l_4, l_5, l_6) = (5, 6, 3, 7, 5, 4)$. Figure 12.5 shows a packing of the 6 objects using only three bins. Numbers in bins are object indices. It is easy to see that at least 3 bins are needed.

Figure 12.5 Optimal packing for Example 12.9

The bin packing problem may be regarded as a variation of the scheduling problem considered earlier. The bins represent processors and L is the time by which all tasks must be completed. l_1 is the processing requirement of task i. The problem is to determine the minimum number of processors needed to accomplish this. An alternative interpretation is to regard the bins as tapes. L is the length of a tape and l_i the tape length needed to store program i. The problem is to determine the minimum

number of tapes needed to store all n programs. Clearly, many interpretations exist for this problem.

Theorem 12.6 The bin packing problem is NP-hard.

Proof: To see this consider the partition problem. Let $\{a_1, a_2, \ldots, a_n\}$ be an instance of the partition problem. Define an instance of the bin packing problem as follows: $l_i = a_i$, $1 \leq i \leq n$ and $L = \Sigma\, a_i/2$. Clearly, the minimum number of bins needed is 2 iff there is a partition for $\{a_1, a_2, \ldots, a_n\}$. □

One can devise many simple heuristics for the bin packing problem. These will not, in general, obtain optimal packings. They will, however, obtain packings that use only a "small" fraction of bins more than an optimal packing. Four simple heuristics are:

I. *First Fit* (FF)

Index the bins 1, 2, 3, ... All bins are initially filled to level zero. Objects are considered for packing in the order 1, 2, ..., n. To pack object i, find the least index j such that bin j is filled to a level r, $r \leq L - l_i$. Pack i into bin j. Bin j is now filled to level $r + l_i$.

II. *Best Fit* (BF)

The initial conditions on the bins and objects are the same as for FF. When object i is being considered, find the least j such that bin j is filled to a level r, $r \leq L - l_i$ and r is as large as possible. Pack i into bin j. Bin j is now filled to level $r + l_i$.

III. *First Fit Decreasing* (FFD)

Reorder the objects so that $l_i \geq l_{i+1}$, $1 \leq i < n$. Now use First Fit to pack the objects.

IV. *Best Fit Decreasing* (BFD)

Reorder the objects so that $l_i \geq l_{i+1}$, $1 \leq i < n$. Now use Best Fit to pack the objects.

Example 12.10 Consider the problem instance of Example 12.9. Figure 12.6 shows the packings resulting when each of the above four packing rules is used. For FFD and BFD the six objects are considered in the order

(4, 2, 1, 5, 6, 3). As is evident from the figure, FFD and BFD do better than either FF or BF on this instance. While FFD and BFD obtain optimal packings on this instance, they do not in general obtain such a packing. □

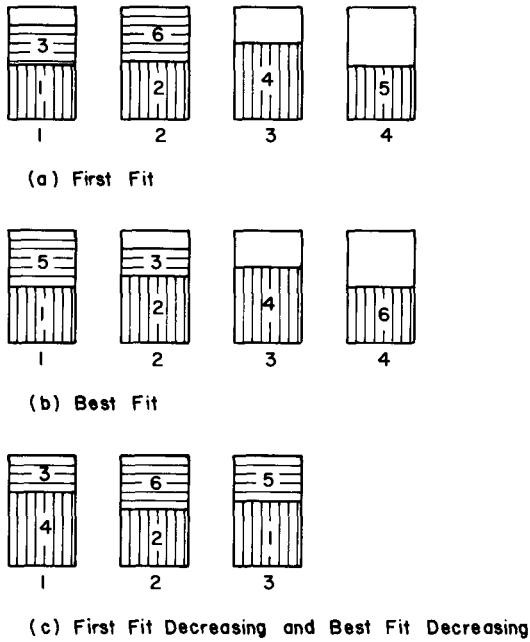


Figure 12.6 Packings resulting from the four heuristics

Theorem 12.7 Let I be an instance of the bin packing problem and let $F^*(I)$ be the minimum number of bins needed for this instance. The packing generated by either FF or BF uses no more than $(17/10)\ F^*(I) + 2$ bins. The packing generated by either FFD or BFD uses no more than $(11/9)\ F^*(I) + 4$ bins. These bounds are the best possible bounds for the respective algorithms.

Proof: The proof of this theorem is rather long and complex. It may be found in the paper: "Worst-Case Performance Bounds For Simple One-Dimensional Packing Algorithms," by Johnson, Demers, Ullman, Garey and Graham, *SIAM Jr. On Computing*, 3(4), pp. 299-325 (1974). □

NP-hard ϵ-Approximation Problems

As in the case of absolute approximations, there exist many NP-hard optimization problems for which the corresponding ϵ-approximation problems are also NP-hard. Let us look at some of these. To begin, consider the traveling salesperson problem.

Theorem 12.8 Hamiltonian cycle $\propto$ ϵ-approximate traveling salesperson.

Proof: Let $G(N,A)$ be any graph. Construct the complete graph $G_1(V, E)$ such that $V = N$ and $E = \{(u, v) | u, v \in V \text{ and } u \neq v\}$. Define the edge weighting function w to be

$$w(u, v) = \begin{cases} 1 & \text{if } (u, v) \in A \\ k & \text{otherwise} \end{cases}$$

Let $n = |N|$. For $k > 1$, the traveling salesperson problem on G_1 has a solution of length n if and only if G has a Hamiltonian cycle. Otherwise, all solutions to G_1 have length $\geq k + n - 1$. If we choose $k \geq (1 + \epsilon)n$, then the only solutions approximating a solution with value n (if there was a Hamiltonian cycle in G_1) also have length n. Consequently, if the ϵ-approximate solution has length $\leq (1 + \epsilon)n$ then it must be of length n. If it has length $>(1 + \epsilon)n$ then G has no Hamiltonian cycle. □

Another NP-hard ϵ-approximation problem is the 0/1 integer programming problem. In the optimization version of this problem we are provided with a linear optimization function $f(x) = \sum p_i x_i + p_0$. We are required to find a 0/1 vector $(x_1, x_2, \ldots, x_n)$ such that $f(x)$ is optimized (either maximized or minimized) subject to the constraints that $\sum a_{ij}x_j \leq b_i$, $1 \leq i \leq k$. k is the number of constraints. Note that the 0/1-knapsack problem is a special case of the 0/1 integer programming problem just described. Hence, the integer programming problem is also NP-hard. We shall now show that the corresponding ϵ-approximation problem is NP-hard for all ϵ, $\epsilon > 0$. This is true even when there is only one constraint (i.e., $k = 1$).

Theorem 12.9 Partition $\propto$ ϵ-approximate integer programming.

Proof: Let $(a_1, a_2, \ldots, a_n)$ be an instance of the partition problem. Construct the following 0/1 integer program:

$$\text{minimize } 1 + k(m - \Sigma a_i x_i)$$
$$\text{subject to } \Sigma a_i x_i \leq m$$
$$x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$
$$m = \Sigma a_i/2$$

The value of an optimal solution is 1 iff the a_is have a partition. If they don't then every optimal solution has a value at least $1 + k$. Suppose there is a polynomial time ϵ-approximate algorithm for the 0/1 integer programming problem for some ϵ, $\epsilon > 0$. Then, by choosing $k > \epsilon$ and using the above construction, this approximation algorithm can be used to solve, in polynomial time, the partition problem. The given partition instance has a partition iff the ϵ-approximate algorithm generates a solution with value 1. All other solutions have value $\hat{F}(I)$ such that $|F^*(I) - \hat{F}(I)|/F^*(I) \geq k > \epsilon$. □

As a final example of an ϵ-approximation problem that is NP-Hard for all ϵ, $\epsilon > 0$, consider the quadratic assignment problem. In one interpretation this problem is concerned with optimally locating m plants. There are n possible sites for these plants, $n \geq m$. At most one plant may be located in any of these n sites. We shall use $x_{i,k}$, $1 \leq i \leq n$, $1 \leq k \leq m$ as mn 0/1 variables. $x_{i,k} = 1$ iff plant k is to be located at site i. The location of the plants is to be chosen so as to minimize the total cost of transporting goods between plants. Let $d_{k,l}$ be the amount of goods to be transported from plant k to plant l. $d_{k,k} = 0$, $1 \leq k \leq m$. Let $c_{i,j}$ be the cost of transporting one unit of the goods from site i to site j. $c_{i,i} = 0$, $1 \leq i \leq n$. The *quadratic assignment problem* has the following mathematical formulation:

$$\text{minimize } f(x) = \sum_{i,j=1}^{n} \sum_{k,l=1}^{m} c_{i,j} d_{k,l} x_{i,k} x_{j,l}$$

$$\text{subject to (a)} \quad \sum_{k=1}^{m} x_{i,k} \leq 1, \quad 1 \leq i \leq n$$

$$\text{(b)} \quad \sum_{i=1}^{n} x_{i,k} = 1, \quad 1 \leq k \leq m$$

$$\text{(c)} \quad x_{i,k} = 0, 1 \text{ for all } i, k$$
$$c_{i,j},\ d_{k,l} \geq 0, \quad 1 \leq i, j \leq n, \quad 1 \leq k, l \leq m$$

Condition (a) ensures that at most one plant is located at any site. Condition (b) ensures that every plant is located at exactly one site. $f(x)$ is the total transportation cost.

Example 12.11 Assume two plants are to be located ($m = 2$) and there are three possible sites ($n = 3$). Assume

$$\begin{pmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{pmatrix} = \begin{pmatrix} 0 & 4 \\ 10 & 0 \end{pmatrix}$$

and

$$\begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} 0 & 9 & 3 \\ 5 & 0 & 10 \\ 2 & 6 & 0 \end{pmatrix}$$

If plant 1 is located at site 1 and plant 2 at site 2 then the transportation cost $f(x)$ is 9*4 + 5*10 = 86. If plant 1 is located at site 3 and plant 2 at site 1 then the cost $f(x)$ is 2*4 + 3*10 = 38. The optimal locations are plant 1 at site 1 and plant 2 at site 3. The cost $f(x)$ is 3*4 + 2*10 = 32. □

Theorem 12.10 Hamiltonian cycle $\propto$ ϵ-approximate quadratic assignment.

Proof: Let $G(N,A)$ be an undirected graph with $m = |N|$. The following quadratic assignment instance is constructed from G:

$$n = m$$

$$c_{j,i} = \begin{cases} 1 & i = (j \bmod m) + 1,\ 1 \leq i,\ j \leq m. \\ 0 & \text{otherwise} \end{cases}$$

$$d_{k,l} = \begin{cases} 1 & \text{if } (k, l) \in A,\ 1 \leq k, l \leq m. \\ \omega & \text{otherwise} \end{cases}$$

The total cost, $f(\gamma)$, of an assignment, γ, of plants to locations is

$\sum_{i=1}^{n} c_{i,j} d_{\gamma(i)\gamma(j)}$ where $j = (i \bmod m) + 1$ and $\gamma(i)$ is the index of the plant assigned to location i. If G has a Hamiltonian cycle $i_1, i_2, \ldots, i_n, i_1$ then the assignment $\gamma(j) = i_j$ has a cost $f(\gamma) = m$. In case G has no Hamiltonian cycle then at least one of the values $d_{\gamma(i).\,\gamma(i \bmod m+1)}$ must be ω and so the cost becomes $\geq m + \omega - 1$. Choosing $\omega > (1 + \epsilon)m$ results in optimal solutions with a value of m if G has a Hamiltonian cycle and value $>(1 + \epsilon)m$ if G has no Hamiltonian cycle. Thus, from an ϵ-approximate solution, it can be determined whether or not G has a Hamiltonian cycle. □

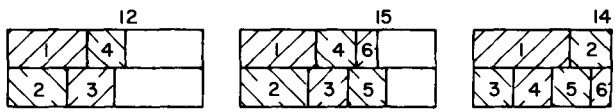
Many other ϵ-approximation problems are known to be NP-hard. Some of these are examined in the exercises. While the three problems just discussed were NP-hard for ϵ, $\epsilon > 0$, it is quite possible for an ϵ-approximation problem to be NP-hard only for ϵ in some range, say, $0 < \epsilon \leq r$. For $\epsilon > r$ there may exist simple polynomial time approximation algorithms.

12.4 POLYNOMIAL TIME APPROXIMATION SCHEMES

Scheduling Independent Tasks

We have seen that the LPT rule leads to a $(1/3 - 1/(3m))$-approximate algorithm for the problem of obtaining an m processor schedule for n tasks. A polynomial time approximation scheme is also known for this problem. This scheme relies on the following scheduling rule: (i) Let k be some specified and fixed integer. (ii) Obtain an optimal schedule for the k longest tasks. (iii) Schedule the remaining $n - k$ tasks using the LPT rule.

Example 12.12 Let $m = 2$; $n = 6$; $(t_1, t_2, t_3, t_4, t_5, t_6) = (8, 6, 5, 4, 4, 1)$ and $k = 4$. The four longest tasks have task times 8, 6, 5 and 4 respectively. An optimal schedule for these has finish time 12 (Figure 12.7(a)). When the remaining two tasks are scheduled using the LPT rule, the schedule of Figure 12.7(b) results. This has finish time 15. Figure 12.7(c) shows an optimal schedule. This has finish time 14. □



(a) optimal for 4 tasks (b) completed schedule (c) overall optimal

Figure 12.7 Using the approximation scheme with $k = 4$

Theorem 12.11 [Graham] Let I be an m processor instance of the scheduling problem. Let $F^*(I)$ be the finish time of an optimal schedule for I and let $\hat{F}(I)$ be the length of the schedule generated by the above scheduling rule. Then,

$$\frac{|F^*(I) - \hat{F}(I)|}{F^*(I)} \leq \frac{1 - 1/m}{1 + \lfloor k/m \rfloor}$$

Proof: Let r be the finish time of an optimal schedule for the k longest tasks. If $\hat{F}(I) = r$ then, $F^*(I) = \hat{F}(I)$ and the theorem is proved. So, assume $\hat{F}(I) > r$. Let t_i, $1 \leq i \leq n$ be the task times of the n tasks of I. Without loss of generality, we may assume $t_i \geq t_{i+1}$, $1 \leq i < n$ and $n > k$. Also, we may assume $n > m$. Let j, $j > k$ be such that task j has finish time $\hat{F}(I)$. Then, no processor may be idle in the interval $[0, \hat{F}(I) - t_j]$. Since $t_{k+1} \geq t_j$, it follows that no processor is idle in the interval $[0, \hat{F}(I) - t_{k+1}]$. Hence,

$$\sum_{i=1}^{n} t_i \geq m(\hat{F}(I) - t_{k+1}) + t_{k+1}$$

and so,

$$F^*(I) \geq \frac{1}{m} \sum_{1}^{n} t_i \geq \hat{F}(I) - \frac{m-1}{m} t_{k+1}$$

or

$$|F^*(I) - \hat{F}(I)| \leq \frac{m-1}{m} t_{k+1}.$$

Since $t_i \geq t_{k+1}$, $1 \leq i \leq k+1$ and at least one processor must execute at least $1 + \lfloor k/m \rfloor$ of these $k+1$ tasks, it follows that:

$$F^*(I) \geq (1 + \lfloor k/m \rfloor) t_{k+1}.$$

Combining these two inequalities, we obtain

$$\frac{|F^*(I) - \hat{F}(I)|}{F^*(I)} \leq ((m-1)/m)/(1 + \lfloor k/m \rfloor) = \frac{1 - 1/m}{1 + \lfloor k/m \rfloor}. \quad \square$$

Using the result of Theorem 12.11, we can construct a polynomial time ϵ-approximation scheme for the scheduling problem. This scheme has ϵ as

an input variable. For any input ϵ it computes an integer k such that $\epsilon \leq (1 - 1/m)/(1 + \lfloor k/m \rfloor)$. This defines the k to be used in the scheduling rule described above. Solving for k, we obtain that any integer k, $k > (m - 1)/\epsilon - m$ will guarantee ϵ-approximate schedules. The time required to obtain such schedules, however, depends mainly on the time needed to obtain an optimal schedule for k tasks on m machines. Using a branch-and-bound algorithm, this time is $O(m^k)$. The time needed to arrange the tasks such that $t_i \geq t_{i+1}$ and also to obtain the LPT schedule for the remaining $n - k$ tasks is $O(n \log n)$. Hence the total time needed by the ϵ-approximate scheme is $O(n \log n + m^k) = O(n \log n + m^{((m-1)/\epsilon - m)})$. Since this time is not polynomial in $1/\epsilon$ (it is exponential in $1/\epsilon$), this approximation scheme is not a fully polynomial time approximation scheme. It is a polynomial time approximation scheme (for any fixed m) as the computing time is polynomial in the number of tasks n.

0/1 Knapsack

The 0/1 knapsack heuristic proposed in Example 12.2 does not result in an ϵ-approximate algorithm for any ϵ, $0 < \epsilon < 1$. Suppose we try out the heuristic described by procedure ϵ-APPROX (Algorithm 12.3). In this procedure P and W are the sets of profits and weights respectively. It is assumed that $p_i/w_i \geq p_{i+1}/w_{i+1}$, $1 \leq i < n$. M is the knapsack capacity and k a nonnegative integer. In the loop of lines 2-5, all $\sum_{i=0}^{k} \binom{n}{i}$ different subsets, I, consisting of at most k of the n objects are generated. If the currently generated subset I is such that $\sum_{i \in I} w_i > M$ it is discarded (as it is infeasible). Otherwise, the space remaining in the knapsack (i.e., $M - \sum_{i \in I} w_i$) is filled using the heuristic described in Example 12.2. This heuristic is stated more formally as procedure L (Algorithm 12.4).

```
line  procedure ε-APPROX(P, W, M, n, k)
          // (i)   the size of a combination is the number of objects in it;//
          // (ii)  the weight of a combination is the sum of the weights of//
                   //the objects in that combination;//
          //(iii)  k is a nonnegative integer which defines the order of the//
                   //algorithm//
  1       PMAX ← 0;
  2       for all combinations I of size ≤ k and weight ≤ M do
  3          P_I ← Σ_{i∈I} p_i
  4          PMAX ← max(PMAX, P_I + L(I, P, W, M, n))
  5       repeat
  6     end ε-APPROX
```

Algorithm 12.3 Heuristic algorithm for knapsack problem

```
procedure L(I, P, W, M, n)
  S ← 0; i ← 1; T ← M − Σ_{i∈I} w_i   //initialize//
  for i ← 1 to n do
    if i ∉ I and w_i ≤ T then S ← S + p_i
                              T ← T − w_i
    endif
  repeat
  return (S)
end L
```

Algorithm 12.4 Subalgorithm for procedure ϵ-APPROX

Example 12.13 Consider the knapsack problem instance with $n = 8$ objects, size of knapsack $= M = 110$, $P = \{11, 21, 31, 33, 43, 53, 55, 65\}$ and $W = \{1, 11, 21, 23, 33, 43, 45, 55\}$.

The optimal solution is obtained by putting objects 1, 2, 3, 5 and 6 into the knapsack. This results in an optimal profit, P^*, of 159 and a weight of 109.

We obtain the following approximations for different k:

a) $k = 0$, PMAX is just the lower bound solution $L(\phi, P, W, M, n)$; PMAX $= 139$; $x = (1, 1, 1, 1, 1, 0, 0, 0)$; $W = \Sigma_i x_i w_i = 89$; $(P^* - \text{PMAX})/P^* = 20/159 = .126$.

b) $k = 1$, PMAX $= 151$; $x = (1, 1, 1, 1, 0, 0, 1, 0)$; $W = 101$; $(P^* - \text{PMAX})/P^* = 8/159 = .05$.

c) $k = 2$, PMAX $= P^* = 159$; $x = (1, 1, 1, 0, 1, 1, 0, 0)$; $W = 109$.

The table of Figure 12.8 gives the details for $k = 1$. It is interesting to note that the combinations $I = \{1\}, \{2\}, \{3\}, \{4\}, \{5\}$ need not be tried since for $I = \{\phi\}$ x_6 is the first x_i which is 0 and so these combinations will yield the same PMAX as $I = \{\phi\}$. This will be true for all combinations I that include only objects for which x_i was 1 in the solution for $I = \{\phi\}$. □

Theorem 12.12 Let J be an instance of the knapsack problem. Let n, M, P and W be as defined for procedure ϵ-APPROX. Let P^* be the value of an optimal solution for J. Let PMAX be as defined by procedure ϵ-APPROX on termination. Then,

$$|P^* - \text{PMAX}|/P^* < 1/(k + 1).$$

Proof: Let R be the set of objects included in the knapsack in some optimal solution. So, $\Sigma_{i \in R} p_i = P^*$ and $\Sigma_{i \in R} w_i \leq M$. If the number of objects in R, $|R|$, is such that $|R| \leq k$ then at some time in the execution of pro-

I	PMAX	P_I	R_I	L	PMAX = max {PMAX, $P_I + L$}	x_{optimal}
ϕ	0	11	1	128	139	(1,1,1,1,1,0,0,0)
6	139	53	43	96	149	(1,1,1,1,0,1,0,0)
7	149	55	45	9	151	(1,1,1,1,0,0,1,0)
8	151	65	55	63	151	(1,1,1,1,0,0,1,0)

*Note that rather than update x_{optimal} it is easier to update the optimal I and recompute x_{optimal} at the end

Figure 12.8 Expansion of Example 12.13 for $k = 1$

cedure ϵ-APPROX, $I = R$ and so PMAX $= P^*$. So, assume $|R| > k$. Let $(\hat{p}_i, \hat{w}_i)$, $1 \le i \le |R|$ be the profits and weights of the objects in R. Assume these have been indexed such that $\hat{p}_1, \ldots, \hat{p}_k$ are the k largest profits in R and that $\hat{p}_i / \hat{w}_i \ge \hat{p}_{i+1} / \hat{w}_{i+1}$, $k < i < |R|$. From the first of these assumptions, it follows that $\hat{p}_{k+t} \le P^*/(k+1)$, $1 \le t \le |R| - k$. Since the loop of lines 2–5 tries out all combinations of size at most k, it follows that in some iteration, I corresponds to the set of k largest profits in R. Hence, $P_I = \sum_{i \in I} p_i = \sum_{i=1}^{k} \hat{p}_i$. Consider the computation of line 4 in this iteration. In the computation of $L(I, P, W, M, n)$ let j be the least index such that $j \notin I$, $w_j > T$ and $j \in R$. Thus, object j corresponds to one of the objects $(\hat{p}_r, \hat{w}_r)$, $k < r \le |R|$ and j is not included in the knapsack by algorithm L. Let object j correspond to $(\hat{p}_m, \hat{w}_m)$.

At the time object j is considered, $T < w_j = \hat{w}_m$. The amount of space filled by procedure L is $M - \sum_{i \in I} w_i - T$ and this is larger than $\sum_{i=k+1}^{m-1} \hat{w}_i$ (as $\sum_1^m \hat{w}_i \le M$). Since this amount of space is filled by considering objects in nondecreasing order of p_i/w_i, it follows that the profit S added by L is no less than

$$\sum_{i=k+1}^{m-1} \hat{p}_i + \frac{\hat{p}_m}{\hat{w}_m} \Delta$$

where

$$\Delta = M - T - \sum_{1}^{m-1} \hat{w}_i.$$

Also,

$$\sum_{i=m}^{|R|} \hat{p}_i \leq \frac{\hat{p}_m}{\hat{w}_m} \left(M - \sum_{1}^{m-1} \hat{w}_i \right).$$

From these two inequalities, we obtain:

$$\begin{aligned} P^* &= P_I + \sum_{k+1}^{|R|} \hat{p}_i \\ &\leq P_I + S - \frac{\hat{p}_m}{\hat{w}_m}\Delta + \frac{\hat{p}_m}{\hat{w}_m}\left(M - \sum_{1}^{m-1} \hat{w}_i \right) \\ &= P_I + S + \hat{p}_m(T/\hat{w}_m) \\ &< P_I + S + \hat{p}_m \end{aligned}$$

Since, PMAX $\geq P_I + S$ and $\hat{p}_m \leq P^*/(k+1)$, it follows that:

$$\frac{|P^* - \text{PMAX}|}{P^*} < \frac{\hat{p}_m}{P^*} \leq \frac{1}{k+1}$$

This completes the proof. □

The time required by Algorithm 12.3 is $O(n^{k+1})$. To see this, note that the total number of subsets tried is

$$\sum_{i=0}^{k} \binom{n}{i} \quad \text{and} \quad \sum_{i=0}^{k} \binom{n}{i} \leq \sum_{i=0}^{k} n^i = \frac{n^{k+1} - 1}{n - 1} = O(n^k).$$

Subalgorithm L has complexity $O(n)$. So, the total time is $O(n^{k+1})$.

Algorithm ϵ-APPROX may be used as a polynomial time approximation scheme. For any given ϵ, $0 < \epsilon < 1$ we may choose k to be the least integer greater than or equal to $(1/\epsilon) - 1$. This will guarantee a fractional error in the solution vaue of at most ϵ. The computing time is $O(n^{1/\epsilon})$.

While Theorem 12.12 provides an upper bound on $|P^* - \text{PMAX}|/P^*$, it does not say anything about how good this bound is. Nor does it say anything about the kind of performance we may expect in practice. Let us now address these two problems.

Theorem 12.13 For every k there exist knapsack instances for which $|(P^* - \text{PMAX})/P^*|$ gets as close to $1/(k+1)$ as desired.

Proof: For any k, the simplest examples approaching the lower bound are obtained by setting: $n = k + 2$; $w_1 = 1$; $p_1 = 2$; p_i, $w_i = q$, $2 \leq i \leq k + 2$, $q > 2$, $M = (k + 1)q$. Then, $P^* = (k + 1)q$. The PMAX given by ϵ-APROX for this k is $kq + 2$ and so $|(P^* - \text{PMAX})/P^*| = (1 - 2/q)/(k + 1)$. By choosing q increasingly large one can get as close to $1/(k + 1)$ as desired. □

Another upper bound on the value of $|P^* - \text{PMAX})/P^*|$ can be obtained from the proof of Theorem 12.12. We know that $P^* - \text{PMAX} < \hat{p}_m$ and that $P^* \geq \text{PMAX}$. Also since $\hat{p}_m$ is one of $\hat{p}_{k+1}, \ldots, \hat{p}_{|R|}$, it follows that $\hat{p}_m \leq \tilde{p}$ where $\tilde{p}$ is the $(k + 1)$-st largest p. Hence $|(P^* - \text{PMAX})/P^*| < \min\{1/(k + 1), \tilde{p}/\text{PMAX}\}$. In most cases $\tilde{p}/\text{PMAX}$ will be smaller than $1/(k + 1)$ and so will give a better estimate of closeness in cases where the optimal is not known. We note that $\tilde{p}$ is easy to compute.

The preceding discussion leads to the following theorem:

Theorem 12.14 The deviation of the solution PMAX obtained from the ϵ-approximate algorithm, from the true optimal P^* is bounded by $|(P^* - \text{PMAX})/P^*| < \min\{1/(k + 1), \tilde{p}/\text{PMAX}\}$.

In order to get a feel for how the approximation scheme might perform in practice, a simulation was conducted. A sample of 600 knapsack instances was used. This sample included problems with $n = 15, 20, 25, 30, \ldots, 60$. For each problem size, 60 instances were generated. These 60 instances included five from each of the following six distributions:

I. random weights w_i and random profits p_i, $1 \leq w_i, p_i \leq 100$.
II. random weights w_i and random profits p_i, $1 \leq w_i, p_i \leq 1000$.
III. random weights w_i, $1 \leq w_i \leq 100$, $p_i = w_i + 10$.
IV. random weights w_i, $1 \leq w_i \leq 1000$, $p_i = w_i + 100$.
V. random profits p_i, $1 \leq p_i \leq 100$, $w_i = p_i + 10$.
VI. random profits p_i, $1 \leq p_i \leq 1000$, $w_i = p_i + 100$.

Random profits and weights were chosen from a uniform distribution over the given range. For each set of p's and w's, two M's were used; $M = 2^* \max\{w_i\}$ and $M = \Sigma\, w_i/2$. This makes for a total of 600 problem instances. Figure 12.9 summarizes the results. The figure gives the number of problems for which $(P^* - \text{PMAX})/P^*$ was in a particular range. .5-APPROX is ϵ-APPROX with $k = 1$ and .33-APPROX is ϵ-APPROX with $k = 2$. As is evident, the observed $|P^* - \text{PMAX}|/P^*$ values are much less than indicated by the worst case bound of Theorem 12.12. Figure

12.10, gives the result of a simulation for large n. Computing times are for a FORTRAN program run on an IBM 360/65 computer.

	$(P^* - PMAX)/P^*$ *100									
Method	0 (Optimal value)	.1%	.5%	1%	2%	3%	4%	5%	10%	25%
L(ϕ,P,S,M,n)	239	267	341	390	443	484	511	528	583	600
.5-APPROX	360	404	477	527	567	585	593	598	600	
.33-APPROX	483	527	564	581	596	600				

Figures give number of solutions that were within r percent of the true optimal solution value; r is the figure in the column head.

Figure 12.9 Results of simulation for set of 600 problems

Problem size n	100	200	500	1000	2000	3000	4000	5000
Computing Time	.25	.9	3.5	14.6	60.4	98.3	180.	350.
Estimated % difference min{ p/PMAX, .5}*100	2.5%	1.3%	.5%	.25%	.12%	.08%	.06%	.04%

$M = \Sigma w_i/2$; $w_i, p_i \in [1,1000]$; times in seconds

Figure 12.10 Computing times using the .5-approximate algorithm

12.5 FULLY POLYNOMIAL TIME APPROXIMATION SCHEMES

The approximation algorithms and schemes we have seen so far are particular to the problem considered. There is no set of well defined techniques that one may use to obtain such algorithms. The heuristics used depended very much on the particular problem being solved. For the case of fully polynomial time approximation schemes, we can identify three underlying techniques. These techniques apply to a variety of optimization problems. We shall discuss these three techniques in terms of maximization problems.

We shall assume the maximization problem to be of the form:

$$\max \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} a_{ij} x_i \leq b_j, \qquad 1 \leq j \leq m$$

$$x_i = 0 \text{ or } 1 \qquad 1 \leq i \leq n \tag{12.1}$$

$$p_i, a_{ij} \geq 0$$

Without loss of generality, we will assume that $a_{ij} \leq b_j$, $1 \leq i \leq n$ and $1 \leq j \leq m$.

If $1 \leq k \leq n$, then the assignment $x_i = y_i$, will be said to be a *feasible assignment* iff there exists at least one feasible solution to (12.1) with $x_i = y_i$, $1 \leq i \leq k$. A *completion* of a feasible assignment $x_i = y_i$ is any feasible solution to (12.1) with $x_i = y_i$, $1 \leq i \leq k$. Let $x_i = y_i$ and $x_i = z_i$, $1 \leq i \leq k$ be two feasible assignments such that for at least one j, $1 \leq j \leq k$, $y_j \neq z_j$. Let $\sum p_i y_i = \sum p_i z_i$. We shall say that $y_1, \ldots, y_k$ *dominates* $z_1, \ldots, z_k$ iff there exists a completion $y_1, \ldots, y_k, y_{k+1}, \ldots, y_n$ such that $\sum_{i=1}^{n} p_i y_i$ is greater than or equal to $\sum_{1 \leq i \leq n} p_i z_i$ for all completions $z_1, \ldots, z_n$ of $z_1, \ldots, z_k$. The approximation techniques to be discussed will apply to those problems that can be formulated as (12.1) and for which simple rules can be found to determine when one feasible assignment dominates another. Such rules exist for example for problems solvable by the dynamic programming technique. Some such problems are 0/1-knapsack; job sequencing with deadlines; job sequencing to minimize finish time and job sequencing to minimize weighted mean finish time.

One way to solve problems stated as above is to systematically generate all feasible assignments starting from the null assignment. Let $S^{(i)}$ represent the set of all feasible assignments for $x_1, x_2, \ldots, x_i$. Then $S^{(0)}$ represents the null assignment and $S^{(n)}$ the set of all completions. The answer to our problem is an assignment in $S^{(n)}$ that maximizes the objective function. The solution approach is then to generate $S^{(i+1)}$ from $S^{(i)}$, $1 \leq i < n$. If an $S^{(i)}$ contains two feasible assignments $y_1, \ldots, y_i$ and $z_1, \ldots, z_i$ such that $\sum p_i y_i = \sum p_i z_i$ then use of the dominance rules enables us to discard or kill that assignment which is dominated. (In some cases the dominance rules may

permit the discarding or killing of a feasible assignment even when $\Sigma\ p_j y_j \neq \Sigma\ p_j z_j$. This happens, for instance, in the knapsack problem (see Section 5.5). Following the use of the dominance rules, it is the case that for each feasible assignment in $S^{(i)}$ $\Sigma_{j=1}^{i}\ p_j x_j$ is distinct. However, despite this, it is possible for each $S^{(i)}$ to contain twice as many feasible assignments as in $S^{(i-1)}$. This results in a worst case computing time that is exponential in n. Note that this solution approach is identical to the dynamic programming solution methodology for the knapsack problem (Section 5.5) and also to the branch-and-bound algorithm later developed for this problem (Section 8.2).

The approximation methods we are about to discuss are called rounding, interval partitioning and separation. These methods will restrict the number of distinct $\Sigma_{j=1}^{i}\ p_j x_j$ to be only a polynomial function of n. The error introduced will be within some prespecified bound.

Rounding

The aim of rounding is to start from a problem instance, I, formulated as in (12.1) and to transform it to another problem instance I' that is easier to solve. This transformation is carried out in such a way that the optimal solution value of I' is "close" to the optimal solution value of I. In particular, if we are provided with a bound, ϵ, on the fractional difference between the exact and approximate solution values then we require that $|F^*(I) - F^*(I')/F^*(I)| \leq \epsilon$, where $F^*(I)$ and $F^*(I')$ represent the optimal solution values of I and I' respectively.

I' is obtained from I by changing the objective function to max $\Sigma\ q_i x_i$. Since I and I' have the same constraints, they have the same feasible solutions. Hence, if the p_i's and q_i's differ by only a "small" amount, the value of an optimal solution to I' will be close to the value of an optimal solution to I.

For example, if the p_i in I have the values: $(p_1, p_2, p_3, p_4) = (1.1, 2.1, 1001.6, 1002.3)$ then if we construct I' with $(q_1, q_2, q_3, q_4) = (0, 0, 1000, 1000)$ it is easy to see that the value of any solution in I is at most 7.1 more than the value of the same solution in I'. This worst case difference is achieved only when $x_i = 1$, $1 \leq i \leq 4$ is a feasible solution for I (and hence also for I'). Since, $a_{ij} \leq b_j$, $1 \leq i \leq n$ and $1 \leq j \leq m$, it follows that $F^*(I) \geq 1002.3$ (as one feasible solution is $x_1 = x_2 = x_3 = 0$ and $x_4 = 1$). But $F^*(I) - F^*(I') \leq 7.1$ and so $(F^*(I) - F^*(I'))/F^*(I)$

≤ 0.007. Solving I using the procedure outlined above, the feasible assignments in $S^{(i)}$ could have the following distinct profit values:

$S^{(0)}$ $\{0\}$
$S^{(1)}$ $\{0, 1.1\}$
$S^{(2)}$ $\{0, 1.1, 2.1, 3.2\}$
$S^{(3)}$ $\{0, 1.1, 2.1, 3.2, 1001.6, 1002.7, 1003.7, 1004.8\}$
$S^{(4)}$ $\{0, 1.1, 2.1, 3.2, 1001.6, 1002.3, 1002.7, 1003.4, 1003.7, 1004.4, 1004.8, 1005.5, 2003.9, 2005, 2006, 2007.1\}$

Thus, barring any elimination of feasible assignments resulting from the dominance rules or from any heuristic, the solution of I using the procedure outlined above would require the computation of $\sum_{0 \leq i \leq n} |S^{(i)}| = 31$ feasible assignments.

The feasible assignments for I' have the following values:

$S^{(0)}$ $\{0\}$
$S^{(1)}$ $\{0\}$
$S^{(2)}$ $\{0\}$
$S^{(3)}$ $\{0, 1000\}$
$S^{(4)}$ $\{0, 1000, 2000\}$

Note that $\sum_{i=0}^{n} |S^{(i)}|$ is only 8. Hence I' can be solved in about one fourth the time needed for I. An inaccuracy of at most .7% is introduced.

Given the p_i's and an ϵ, what should the q_i's be so that

$$(F^*(I) - F^*(I'))/F^*(I) \leq \epsilon \quad \text{and} \quad \sum_{i=0}^{n} |S^{(i)}| \leq u(n, 1/\epsilon)$$

where u is a polynomial in n and $1/\epsilon$? Once we can figure this out we will have a fully polynomial approximation scheme for our problem since it is possible to go from $S^{(i-1)}$ to $S^{(i)}$ in time proportional to $O(S^{(i-1)})$. (See the knapsack algorithm of Section 5.5.)

Let LB be an estimate for $F^*(I)$ such that $F^*(I) \geq$ LB. Clearly, we may assume LB $\geq \max_i \{p_i\}$. If

$$\sum_{i=1}^{n} |p_i - q_i| \leq \epsilon F^*(I)$$

then, it is clear that, $(F^*(I) - F^*(I'))/F^*(I) \leq \epsilon$. Define $q_i = p_i - \text{rem}(p_i, (\text{LB} \cdot \epsilon)/n)$ where rem(a, b) is the remainder of a/b, i.e., $a -$

$\lfloor a/b \rfloor\ b$ (e.g., rem(7, 6) = 1/6 and rem(2.2, 1.3) = .9). Since rem(p_i, LB·ϵ/n) < LB·ϵ/n, it follows that $\sum |p_i - q_i| <$ LB·$\epsilon \leq F^*\cdot\epsilon$. Hence, if an optimal solution to I' is used as an optimal solution for I then the fractional error is less than ϵ.

In order to determine the time required to solve I' exactly, it is useful to introduce another problem I'' with s_i, $1 \leq i \leq n$ as its objective function coefficients. Define $s_i = \lfloor (p_i \cdot n)/(\text{LB}\cdot\epsilon) \rfloor$, $1 \leq i \leq n$. It is easy to see that $s_i = (q_i \cdot n)/(\text{LB}\cdot\epsilon)$. Clearly, the $S^{(i)}$'s corresponding to the solutions of I' and I'' will have the same number of tuples. (r, t) is a tuple in an $S^{(i)}$ for I' iff $((r \cdot n)/(\text{LB}\cdot\epsilon), t)$, is a tuple in the $S^{(i)}$ for I''. Hence, the time needed to solve I' is the same as that needed to solve I''. Since $p_i \leq$ LB, it follows that $s_i \leq \lfloor n/\epsilon \rfloor$. Hence

$$|S^{(i)}| \leq 1 + \sum_{j=1}^{i} s_j \leq 1 + i\lfloor n/\epsilon \rfloor$$

and so

$$\sum_{i=0}^{n-1} |S^{(i)}| \leq n + \sum_{i=0}^{n-1} i\lfloor n/\epsilon \rfloor = O(n^3/\epsilon).$$

Thus, if we can go from $S^{(i-1)}$ to $S^{(i)}$ in $O(|S^{(i-1)}|)$ time then I'' and hence I' can be solved in $O(n^3/\epsilon)$ time. Moreover, the solution for I' will be an ϵ-approximate solution for I and we would thus have a fully polynomial time approximation scheme. When using rounding, we will actually solve I'' and use the resulting optimal solution as the solution to I.

Example 12.14 Consider the 0/1 knapsack problem of Section 5.5. While solving this problem by successively generating $S^{(0)}$, $S^{(1)}$, ..., $S^{(n)}$ the feasible assignments for $S^{(i)}$ may be represented by tuples of the form (r, t) where

$$r = \sum_{j=1}^{i} p_j x_j \quad \text{and} \quad t = \sum_{j=1}^{i} w_j x_j.$$

The dominance rule developed in Section 5.5 for this problem is: (r_1, t_1) dominates (r_2, t_2) iff $t_1 \leq t_2$ and $r_1 \geq r_2$.

Let us solve the following instance of the 0/1 knapsack problem: $n = 5$, $M = 1112$ and $(p_1, p_2, p_3, p_4, p_5) = (w_1, w_2, w_3, w_4, w_5) = \{1, 2, 10, 100, 1000\}$. Since $p_i = w_i$, $1 \leq i \leq 5$, the tuples (r, t) in $S^{(i)}$, $0 \leq i \leq 5$

will have $r = t$. Consequently, it is necessary to retain only one of the two coordinates r, t. The $S^{(i)}$ obtained for this instance are: $S^{(0)} = \{0\}$; $S^{(1)} = \{0, 1\}$; $S^{(2)} = \{0, 1, 2, 3\}$; $S^{(3)} = \{0, 1, 2, 3, 10, 11, 12, 13\}$; $S^{(4)} = \{0, 1, 2, 3, 10, 11, 12, 13, 100, 101, 102, 103, 110, 111, 112, 113\}$; $S^{(5)} = \{0, 1, 2, 3, 10, 11, 12, 13, 100, 101, 102, 103, 110, 111, 112, 113, 1000, 1001, 1002, 1003, 1010, 1011, 1012, 1013, 1100, 1101, 1102, 1103, 1110, 1111, 1112\}$.

The optimal solution has value $\Sigma\, p_i x_i = 1112$.

Now, let us use rounding on the above problem instance to find an approximate solution with value at most 10% less than the optimal value. We thus have $\epsilon = 1/10$. Also, we know that $F^*(I) \geq \text{LB} \geq \max\{p_i\} = 1000$. The problem I'' to be solved is: $n = 5$, $M = 1112$, $(s_1, s_2, s_3, s_4, s_5) = (0, 0, 0, 5, 50)$ and $(w_1, w_2, w_3, w_4, w_5) = (1, 2, 10, 100, 1000)$. Hence, $S^{(0)} = S^{(1)} = S^{(2)} = S^{(3)} = \{(0, 0)\}$; $S^{(4)} = \{(0, 0), (5, 100)\}$; $S^{(5)} = \{(0, 0), (5, 100), (50, 1000), (55, 1100)\}$.

The optimal solution is $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 1, 1)$. Its value in I'' is 55 and in the original problem 1100. The error $(F^*(I) - \hat{F}(I))/F^*(I)$ is therefore $12/1112 < 0.011 < \epsilon$. At this time we see that the solution may be improved by setting either $x_1 = 1$ or $x_2 = 1$ or $x_3 = 1$. □

Rounding as described in its full generality results in $O(n^3/\epsilon)$ time approximation schemes. It is possible to specialize this technique to the specific problem being solved. In particular, we can obtain specialized and asymptotically faster polynomial time approximation schemes for the knapsack problem as well as for the problem of scheduling tasks on two processors to minimize finish time. The complexity of the resulting algorithms is $O(n(\log n + 1/\epsilon^2))$.

Let us investigate the specialized rounding scheme for the 0/1 knapsack problem. Let I be an instance of this problem and let ϵ be the desired accuracy. Let $P^*(I)$ be the value of an optimal solution. First, a good estimate UB for $P^*(I)$ is obtained. This is done by ordering the n objects in I such that $p_i/w_i \geq p_{i+1}/w_{i+1}$, $1 \leq i < n$. Next, we find the largest j such that $\Sigma_1^j w_i \leq M$. If $j = n$, then the optimal solution is $x_i = 1$, $1 \leq i \leq n$ and $P^*(I) = \Sigma\, p_i$. So, assume $j < n$. Define $\text{UB} = \Sigma_1^{j+1} p_i$. We can show $\frac{1}{2}\,\text{UB} \leq P^*(I) < \text{UB}$. The inequality $P^*(I) < \text{UB}$ follows from the ordering on p_i/w_i. The inequality $\frac{1}{2}\text{UB} \leq P^*(I)$ follows from the observation that

$$P^*(I) \geq \sum_{i=1}^{J} p_i \quad \text{and} \quad P^*(I) \geq \max\left\{\sum_1^j p_i, p_{j+1}\right\}.$$

Hence, $2P^*(I) \geq \Sigma_1^{j+1} p_i = \text{UB}$.

Now, let $\delta = UB^*\epsilon^2/9$. Divide the n objects into 2 classes BIG and SMALL. BIG includes all objects with $p_i > \epsilon\ UB/3$. SMALL includes all other objects. Let the number of objects in BIG be r. Replace each p_i in BIG by q_i such that $q_i = \lfloor p_i/\delta \rfloor$. (This is the rounding step.) The knapsack problem is solved exactly using these r objects and the q_i's.

Let $S^{(r)}$ be the set up tuples resulting from the dynamic programming algorithm. For each tuple $(x, y) \in S^{(r)}$ fill the remaining space $M - y$ by considering the objects in SMALL in nondecreasing order of p_i/w_i. Use the filling that has maximum value as the answer.

Example 12.15 Consider the problem instance of Example 12.14. $n = 5$, $(p_1, p_2, p_3, p_4, p_5) = (w_1, w_2, w_3, w_4, w_5,) = (1, 2, 10, 100, 1000)$, $M = 1112$ and $\epsilon = 1/10$. The objects are already in nonincreasing order of p_i/w_i. For this instance, $UB = \Sigma_1^5 p_i = 1113$. Hence, $\delta = 3.71/3$ and $\epsilon UB/3 = 37.1$. SMALL, therefore, includes objects 1, 2 and 3. BIG $= \{4, 5\}$. $q_4 = \lfloor p_4/\delta \rfloor = 94$ and $q_5 = \lfloor p_5/\delta \rfloor = 946$. Solving the knapsack instance $n = 2$, $M = 1112$, $(q_4, w_4) = (94, 100)$ and $(q_5, w_5) = (946, 1000)$, we obtain: $S^{(0)} = \{(0, 0)\}$; $S^{(1)} = \{(0, 0), (94, 100)\}$ and $S^{(2)} = \{(0, 0), (94, 100), (946, 1000), (1040, 1100)\}$. Filling (0,0) from SMALL, we get the tuple (13, 13). Filling (94, 100), (946, 1000) and (1040, 1100) yields the tuples (107, 113), (959, 1013) and (1043, 1100) respectively. The answer is given by the tuple (1043, 1100). This corresponds to $(x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 1, 1)$ and $\Sigma p_i x_i = 1103$. □

An exercise explores a modification to the basic rounding scheme illustrated in the above example. This modification results in "better" solutions.

Theorem 12.15 [Ibarra and Kim] The algorithm just described is an ϵ-approximate algorithm for the 0/1-knapsack problem.

Proof: The proof may be found in the paper by Ibarra and Kim which is cited at end of this chapter. □

The time needed to initially sort according to p_i/w_i is $O(n \log n)$. UB can be computed in $O(n)$ time. Since $P^*(I) \leq UB$, there are at most $UB/\delta = 9/\epsilon^2$ tuples in any $S^{(i)}$ in the solution of BIG. The time to obtain $S^{(r)}$ is therefore $O(r/\epsilon^2) \leq O(n/\epsilon^2)$. Filling each tuple in $S^{(r)}$ with objects from SMALL takes $O(|\,SMALL\,|)$ time. $|S^{(r)}| \leq 9/\epsilon^2$ and so the total time for this step is at most $O(n/\epsilon^2)$. The total time for the algorithm is therefore $O(n(\log n + 1/\epsilon^2))$. A faster approximation scheme for the knapsack problem has been obtained by Lawler (see the references). His scheme also uses rounding.

Interval Partitioning

Unlike rounding, interval partitioning does not transform the original problem instance into one that is easier to solve. Instead, an attempt is made to solve the problem instance I by generating a restricted class of the feasible assignments for $S^{(0)}$, $S^{(1)}$, ..., $S^{(n)}$. Let P_i be the maximum $\sum_{j=1}^{i} p_j x_j$ amongst all feasible assignments generated for $S^{(i)}$. Then the profit interval $[0, P_i]$ is divided into subintervals each of size $P_i\epsilon/(n-1)$ (except possibly the last interval which may be a little smaller). All feasible assignments in $S^{(i)}$ with $\sum_{j=1}^{i} p_j x_j$ in the same subinterval are regarded as having the same $\sum_{j=1}^{i} p_j x_j$ and the dominance rules are used to discard all but one of them. The $S^{(i)}$ resulting from this elimination is used in the generation of $S^{(i+1)}$. Since the number of subintervals for each $S^{(i)}$ is at most $\lceil n/\epsilon \rceil + 1$, $|S^{(i)}| \leq \lceil n/\epsilon \rceil + 1$. Hence, $\sum_{i}^{n} |S^{(i)}| = O(n^2/\epsilon)$.

The error introduced in each feasible assignment due to this elimination in $S^{(i)}$ is less than the subinterval length. This error may however propagate from $S^{(1)}$ up through $S^{(n)}$. However, the error is additive. Let $\hat{F}(I)$ be the value of the optimal generated using interval partitioning, and $F^*(I)$ the value of a true optimal. It follows that

$$F^*(I) - \hat{F}(I) \leq \left(\epsilon \sum_{i=1}^{n-1} P_i\right)/(n-1).$$

Since $P_i \leq F^*(I)$, it follows that $(F^*(I) - \hat{F}(I))/F^*(I) \leq \epsilon$, as desired.

In many cases the algorithm may be speeded by starting with a good estimate, LB for $F^*(I)$ such that $F^*(I) \geq$ LB. The subinterval size is then $\text{LB} \cdot \epsilon/(n-1)$ rather than $P_i\epsilon/(n-1)$. When a feasible assignment with value greater than LB is discovered, the subinterval size can be chosen as described above.

Example 12.16 Consider the same instance of the 0/1 knapsack problem as in Example 12.14. $\epsilon = 1/10$ and $F^*(I) \geq \text{LB} \geq 1000$. We can start with a subinterval size of $\text{LB} \cdot \epsilon/(n-1) = 1000/40 = 25$. Since all tuples (p, t) in $S^{(i)}$ have $p = t$, only p will be explicitly retained. The intervals are $[0, 25)$, $[25, 50)$,... etc. Using interval partitioning we obtain: $S^{(0)} = S^{(1)} = S^{(2)} = S^{(3)} = \{0\}$; $S^{(4)} = \{0, 100\}$; $S^{(5)} = \{0, 100, 1000, 1100\}$.

The best solution generated using interval partitioning is $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 1, 1)$ and its value $\hat{F}(I)$ is 1100. $(F^*(I) - \hat{F}(I))/F^*(I) = 12/1112 < 0.011 < \epsilon$. Again, the solution value may be improved by using a heuristic to change some of the x_i's from 0 to 1. □

Separation

Assume that in solving a problem instance I, we have obtained an $S^{(i)}$ with feasible solutions having the following $\Sigma_{i \le j \le i} p_j x_j$: 0, 3.9, 4.1, 7.8, 8.2, 11.9, 12.1. Further assume that the interval size $P_i\epsilon/(n-1)$ is 2. Then the subintervals are [0, 2), [2, 4), [4, 6), [6, 8), [8, 10), [10, 12) and [12, 14). Each value above falls in a different subinterval and so no feasible assignments are eliminated. However, there are three pairs of assignments with values within $P_i\epsilon/(n-1)$. If the dominance rules are used for each pair, only 4 assignments will remain. The error introduced is at most $P_i\epsilon/(n-1)$. More formally, let $a_0, a_1, a_2, \ldots, a_r$ be the distinct values of $\Sigma_{j=1}^{i} p_j x_j$ in $S^{(i)}$. Let us assume $a_0 < a_1 < a_2 \cdots < a_r$. We will construct a new set J from $S^{(i)}$ by making a left to right scan and retaining a tuple only if its value exceeds the value of the last tuple in J by more than $P_i\epsilon/(n-1)$. This is described by the following algorithm:

```
J ← assignment corresponding to a_0; XP ← a_0
for j ← 1 to r do
  if a_j > XP + P_iε/(n − 1)
                then put assignment corresponding to a_j into J
                XP ← a_j
  endif
repeat
```

The preceding algorithm assumes that the assignment with less profit will dominate the one with more profit in case we regard both assignments as yielding the same profit $\Sigma\, p_j x_j$. In case the reverse is true the algorithm can start with a_r and work downwards. The analysis for this strategy is the same as that for interval partitioning. The same comments regarding the use of a good estimate for $F^*(I)$ hold here too.

Intuitively one may expect separation to always work better than interval partitioning. The following example illustrates that this need not be the case. However, empirical studies with one problem indicate interval partitioning to be inferior in practice.

Example 12.17 Using separation on the data of Example 12.14 yields the same $S^{(i)}$ as obtained using interval partitioning. We have already seen an instance where separation performs better than interval partitioning. Now, we shall see an example where interval partitioning does better than separation. Assume that the subinterval size $\mathrm{LB}\cdot\epsilon/(n-1)$ is 2. Then the

intervals are [0, 2), [2, 4), [4, 6) $\cdots$ etc. Assume further that $(p_1, p_2, p_3, p_4, p_5) = (3, 1, 5.1, 5.1, 5.1)$. Then, following the use of interval partitioning we have: $S^{(0)} = \{0\}$; $S^{(1)} = \{0, 3\}$; $S^{(2)} = \{0, 3, 4\}$; $S^{(3)} = \{0, 3, 4, 8.1\}$; $S^{(4)} = \{0, 3, 4, 8.1, 13.2\}$; $S^{(5)} = \{0, 3, 4, 8.1, 13.2, 18.3\}$.

Using separation with $\text{LB} \cdot \epsilon/(n-1) = 2$ we have: $S^{(0)} = \{0\}$; $S^{(1)} = \{0, 3\}$; $S^2 = \{0, 3\}$; $S^{(3)} = \{0, 3, 5.1, 8.1\}$; $S^{(4)} = \{0, 3, 5.1, 8.1, 10.2, 13.2\}$; $S^{(5)} = \{0, 3, 5.1, 8.1, 10.2, 13.2, 15.3, 18.3\}$. □

In order to compare the relative performance of interval partitioning (I) and separation (S), a simulation was carried out. We used the job sequencing with deadlines problem as the test problem. Algorithms for I and S were programmed in FORTRAN and run on a CDC CYBER 74 computer. Both algorithms were tested with $\epsilon = 0.1$. Three data sets were used: (p_i = profit; t_i = processing time needed; d_i = deadline).

Data Set A: random profits $p_i \in [1, 100]$, $t_i = p_i$ and $d_i = \sum_1^n t_i/2$.

Data Set B: random $p_i \in [1, 100]$; $t_i = p_i$ and random $d_i \in [t_i, t_i + 25n]$

Data Set C: random $p_i \in [1, 100]$; random $t_i \in [1, 100]$ and random $d_i \in [t_i, t_i + 25n]$.

The program had a capacity to solve all problems generating no more than 9000 tuples (i.e., $\sum_0^n |S^{(i)}| \leq 9000$). For each data set an attempt was made to run 10 problems of size 5, 15, 25, 35, 45, ... Figure 12.11 summarizes the results.

The exercises examine some of the other problems to which these techniques apply. It is interesting to note that one may couple existing heuristics to the approximation schemes that result from the above three techniques. This is because of the similarity in solution procedures for the exact and approximate problems. In the approximation algorithms of Sections 12.2–12.4 it is usually not possible to use existing heuristics.

At this point, one might well ask the question: What kind of NP-hard problems can have fully polynomial time approximation schemes? Clearly, no NP-hard ϵ-approximation problem can have such a scheme unless $P = \text{NP}$. A stronger result may be proven. This stronger result is that the only NP-hard problems that can have fully polynomial time approximation schemes (unless $P = \text{NP}$) are those which are polynomially solvable if restricted to problem instances in which all numbers are bounded by a fixed polynomial in n. Examples of such problems are the knapsack and job sequencing with deadlines problems.

Data Set	A	B	C
Total number of problems solved	80	30	30
Number of optimal solutions generated by I	54	20	16
Number of optimal solutions generated by S	53	18	14
Average fracitonal error in nonoptimal solutions by I	.0025	.0047	.0040
Average fractional error in nonoptimal solutions by S	.0024	.0047	.0040
Number of I solutions better than S	3	7	9
Number of S solutions better than I	1	7	6

Figure 12.11 Relative performance of I and S

Definition [Garey and Johnson] Let L be some problem. Let I be an instance of L and let LENGTH(I) be the number of bits in the representation of I. Let MAX(I) be the magnitude of the largest number in I. Without loss of generality, we may assume that all numbers in I are integer. For some fixed polynomial p let L_p be problem L restricted to those instances I for which MAX(I) $\leq$ p(LENGTH(I)). Problem L is *strongly NP-hard* iff there exists a polynomial p such that L_p is NP-hard.

Examples of problems that are strongly NP-hard are: Hamiltonian cycle; node cover; feedback arc set; traveling salesperson, max-clique, etc. The 0/1 knapsack problem is probably not strongly NP-hard (note that there is no known way to show that a problem is not strongly NP-hard) as when MAX(I) $\leq$ p(LENGTH(I)) then I can be solved in time O(LENGTH(I)^2* p(LENGTH(I))) using the dynamic programming algorithm of Section 5.5.

Theorem 12.16 [Garey and Johnson] Let L be an optimization problem such that all feasible solutions to all possible instances have a value that is a positive integer. Further, assume that for all instances I of L, the optimal value $F^*(I)$ is bounded by a polynomial function p in the variables LENGTH(I) and MAX(I), i.e., $0 < F^*(I) < p$(LENGTH(I), MAX(I)) and $F^*(I)$ is an integer. If L has a fully polynomial time approximation scheme, then L has an exact algorithm of complexity a polynomial in LENGTH(I) and MAX(I).

Proof: Suppose L has a fully polynomial time approximation scheme. We shall show how to obtain optimal solutions to L in polynomial time. Let I be any instance of L. Define $\epsilon = 1/p(\text{LENGTH}(I), \text{MAX}(I))$. With this ϵ, the approximation scheme is forced to generate an optimal solution. To see this, let $\hat{F}(I)$ be the value of the solution generated. Then,

$$|F^*(I) - \hat{F}(I)| \leq \epsilon F^*(I) \leq F^*(I)/p(\text{LENGTH}(I), \text{MAX}(I)) < 1$$

Since, by assumption all feasible solutions are integer valued, $F^*(I) = \hat{F}(I)$. Hence, with this ϵ, the approximation scheme becomes an exact algorithm.

The complexity of the resulting exact algorithm is easy to obtain. Let $q(\text{LENGTH}(I), 1/\epsilon)$ be a polynomial such that the complexity of the approximation scheme is $O(q(\text{LENGTH}(I), 1/\epsilon))$. The complexity of this scheme when ϵ is chosen as above is $O(q(\text{LENGTH}(I), p(\text{LENGTH}(I), \text{MAX}(I)))$ which is $O(q'(\text{LENGTH}(I), \text{MAX}(I)))$ for some polynomial q'. □

When Theorem 12.16 is applied to integer valued problems that are NP-hard in the strong sense, we see that no such problem can have a fully polynomial time approximation scheme unless $P = \text{NP}$. The above theorem also tells us something about the kind of exact algorithms obtainable for strongly NP-hard problems. A *pseudo-polynomial* time algorithm is one whose complexity is a polynomial in LENGTH(I) and MAX(I). The dynamic programming algorithm for the knapsack problem (Section 5.5) is a pseudo-polynomial time algorithm. No strongly NP-hard problem can have a pseudo polynomial time algorithm unless $P = \text{NP}$.

12.6 PROBABILISTICALLY GOOD ALGORITHMS

The approximation algorithms of the preceding sections had the nice property that their worst case performance could be bounded by some constants (k in the case of an absolute approximation and ϵ in the case of an ϵ-approximation). The requirement of bounded performance tends to categorize other algorithms that "usually work well" as being bad. Some algorithms with unbounded performance may in fact "almost always" either solve the problem exactly or generate a solution that is "exceedingly close" in value to the value of an optimal solution. Such algorithms are "good" in a probabilistic sense. If we pick a problem instance I at random then there is a very high probability that the algorithm will generate a very good approximate solution. In this section we shall consider two algorithms with this property. Both algorithms are for NP-hard problems.

First, since we shall be carrying out a probabilistic analysis of the algorithms we need to define a sample space of inputs. The sample space is set up by first defining a sample space S_n for each problem size n. Problem instances of size n are drawn from S_n. Then, the overall sample space is the infinite cartesian product $S_1 \times S_2 \times S_3 \times \ldots \times S_n \ldots$. An element of the sample space is a sequence $X = x_1, x_2, \ldots, x_n, \ldots$ such that x_i is drawn from S_i.

Definition [Karp] An algorithm $\mathcal{A}$ solves a problem L *almost everywhere* (abbreviated a. e.) if, when $X = x_1, x_2, \ldots, x_n, \ldots$ is drawn from the sample space $S_1 \times S_2 \times S_3 \times \ldots \times S_n, \ldots$, the number of x_i on which the algorithm fails to solve L is finite with probability 1.

Since both the algorithms we shall be discussing are for NP-hard graph problems, we shall first describe the sample space for which the probabilistic analysis will be carried out. Let $p(n)$ be a function such that $0 \leq p(n) \leq 1$ for all $n \geq 0$. A random n vertex graph is constructed by including edge (i, j), $i \neq j$, with probability $p(n)$.

The first algorithm we shall consider is due to Posa. This is an algorithm to find a Hamiltonian cycle in an undirected graph. Informally, Posa's algorithm proceeds as follows. First, an arbitrary vertex (say vertex 1) is chosen as the start vertex. The algorithm maintains a simple path P starting from vertex 1 and ending at vertex k. Initially P is a trivial path with $k = 1$, i.e., there are no edges in P. At each iteration of the algorithm an attempt is made to increase the length of P. This is done by considering an edge (k, j) incident to the end point k of P. When edge (k, j) is being considered, one of three possibilities exist:

(i) [$j = 1$ and path P includes all the vertices of the graph]
In this case a Hamiltonian cycle has been found and the algorithm terminates.

(ii) [j is not on the path P]
In this case the length of path P is increased by adding (k, j) to it. j becomes the new end point of P.

(iii) [j is already on path P]
Now there is a unique edge $e = (j, m)$ in P such that deletion of e and the inclusion of (k, j) to P results in a simple path. e is deleted and (k, j) added to P. P is now a simple path with endpoint m.

The algorithm is constrained so that case (iii) does not generate two paths of the same length having the same end point. With a proper choice

of data representations, this algorithm can be implemented to run in time $O(n^2)$ where n is the number of vertices in the graph G. It is easy to see that this algorithm does not always find a Hamiltonian cycle in a graph that contains such a cycle. However, Posa has shown the following:

Theorem 12.17 [Posa] If $p(n) \approx (\alpha \ln n/n)$, $\alpha > 1$ then the preceding algorithm finds a Hamiltonian cycle (a. e.).

Proof: See the paper by Posa. □

Example 12.18 Let us try out the above algorithm on the five vertex graph of Figure 12.12. The path P initially consists of vertex 1 only. Assume edge (1, 4) is chosen. This represents case (ii) and P is expanded to $\{1, 4\}$. Assume edge (4, 5) is chosen next. Path P now becomes $\{1, 4, 5\}$. Edge (1, 5) is the only possibility for the next edge. This results in case (iii) and P becomes $\{1, 5, 4\}$. Now assume edges (4, 3) and (3, 2) are considered. P becomes $\{1, 5, 4, 3, 2\}$. If edge (1, 2) is next considered, a Hamiltonian cycle is found and the algorithm terminates. □

The next probabilistically good algorithm we shall look at is for the maximum independent set problem. A subset of vertices N of graph $G(V, E)$ is said to be independent iff no two vertices in N are adjacent in G. Algorithm 12.5 is a greedy algorithm to construct a maximum independent set.

```
procedure INDEP(V, E)
  N ← φ
  while there is a v ∈(V-N) and v not adjacent to any vertex in N do
      N ← N ∪ {v}
  repeat
  return (N)
end INDEP
```

Algorithm 12.5 Finding an independent set

One can easily construct examples of n vertex graphs for which INDEP generates independent sets of size 1 when in fact a maximum independent set contains $n - 1$ vertices. However, for certain probability distributions it can be shown that INDEP generates good approximations almost everywhere. If $F^*(I)$ and $\hat{F}(I)$ represent the size of a maximum independent set and one generated by algorithm INDEP, respectively, then the following theorem is obtained:

Theorem 12.18 [Karp] If $p(n) = c$, for some constanct c, then for every $\epsilon > 0$ we have:

$$(F^*(I) - \hat{F}(I))/F^*(I) \leq .5 + \epsilon \quad \text{(a.e.)}.$$

Proof: See the paper by Karp. □

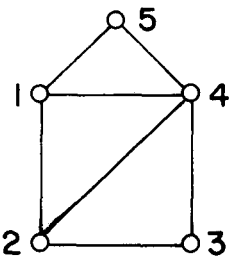


Figure 12.12 Graph for Example 12.18

Algorithm INDEP can easily be implemented to have polynomial complexity. Some other NP-hard problems for which probabilistically good algorithms are known are: Euclidean traveling salesperson, minimal colorings of graphs, set covering, maximum weighted clique and partition.

REFERENCES AND SELECTED READINGS

Note: Exercise numbers at the end of a reference indicate that these exercises are based on work reported in this reference. The reference, however, contains more results than covered by the cited exercises.

Our terminology for absolute, $f(n)$ and ϵ-approximation algorithms is taken from the paper:

"Combinatorial problems: reducibility and approximation," by S. Sahni and E. Horowitz, *Op. Res.*, 26(4), 1978.

The terms approximation scheme, polynomial time approximation scheme and fully polynomial time approximation scheme were coined by Garey and Johnson and used in their lecture on approximation algorithms which was presented at the Symposium on Algorithms and Complexity, Carnegie Mellon Institute, Pittsburgh, 1976. Sahni pointed out that for the 0/1 knapsack problem the corresponding

absolute approximation problem is also NP-hard. The following paper, contains the remark on the knapsack problem:

"Approximate algorithms for the 0/1-knapsack problem," by S. Sahni, *JACM*, 22, pp. 115–124, 1975. [exercise 18].

This paper also contains the polynomial time approximation scheme for the 0/1-knapsack problem discussed in §12.4. Several other absolute approximation problems are shown NP-hard in:

"A computer scientist looks at reliability computations," by A. Rosenthal, in *Reliability and Fault Tree Analysis*, edited by J. Fussel and N. Singpurwalla, *SIAM*, 1975.

The analysis of the LPT rule of Section 12.3 is due to R. Graham and appears in:

"Bounds on multiprocessor timing anomalies," by R. Graham, *SIAM Jr. on Appl. Math.*, 17(2), pp. 416–429, 1969.

This paper also contains the polynomial time approximation scheme for scheduling independent tasks that was discussed in §12.4. ϵ-approximate bin packing algorithms may be found in:

"Performance bounds for simple one dimensional bin packing algorithms," by D. Johnson, A. Demers, J. Ullman, M. Garey and R. Graham, *SIAM Jr. on Comput.*, 3(4), pp. 299–325, 1974.

An excellent bibliography on approximation algorithms is:

"Approximation algorithms for combinatorial problems: an annotated bibliography," by M. Garey and D. Johnson, in *Algorithms and Complexity: Recent Results and New Directions*, J. Traub, ed., Academic Press, 1976.

Polynomial time ϵ-approximate algorithms for many scheduling problems may be found in the following papers:

"Scheduling independent tasks to reduce mean finishing time," by J. Bruno, E. Coffman and R. Sethi, *CACM*, 17(7), 382–387, 1974.

"Algorithms for minimizing mean flow time," by J. Bruno, E. Coffman and R. Sethi, *Proc. IFIP Congr. 74*, North Holland Pub. Co., Amsterdam, 1974, pp. 504–510.

"A level algorithm for preemptive scheduling," by E. Horvath, S. Lam and R. Sethi, *JACM*, 24(1), pp. 32–43, 1977.

"Bounds on LPT schedules on uniform processors," by T. Gonzalez, O. Ibarra and S. Sahni, *SIAM Jr. on Computing*, 6(1), pp. 155–166, 1977 [exercises 4–6].

"Job shop and flow shop schedules: complexity and approximation," by T. Gonzalez and S. Sahni, *Oper. Res.*, 26(1), pp. 36–52, 1978.

"Heuristic algorithms for scheduling independent tasks on nonidentical processors," by O. Ibarra and C. Kim, *JACM*, 24(2), pp. 280–289, 1977 [exercises 8–10].

A 0.5-approximate algorithm for the Euclidean traveling salesperson problem appears in:

"Worst-case analysis of a new heuristic for the traveling salesman problem," by N. Christofedes, Manag. Sci. Res. Report #388, Carnegie Mellon University, 1976.

ϵ-approximate algorithms for other NP-hard problems appear in:

"Approximation algorithms for some routing problems," by G. Fredrickson, M. Hecht and C. Kim, *Proc. 17th An. Symp. on Found. of Comp. Sci.*, Houston, Texas, pp. 216–227, 1976.

"Location of bank accounts to optimize float: an analytic study of exact and approximate algorithms," by G. Cornuejols, M. Fisher and G. Nemhauser, *Manag. Sci.* 23(8), pp. 789–810, 1977.

"An analysis of approximations for maximizing submodular set functions—II," by M. Fisher, G. Nemhauser and L. Wolsey, CORE discussion paper #7629, Universite Catholique De Louvain, Belgium, 1976.

"An analysis of approximations for finding a maximum weight hamiltonian circuit," by M. Fisher, G. Nemhauser and L. Wolsey, CORE, University of Louvain, Belgium, 1977.

"Code generation for expressions with common subexpressions," by A. Aho, S. Johnson and J. Ullman, *JACM*, 24(1), pp. 146–160, 1977.

Some $f(n)$-approximate algorithms appear in:

"Approximation algorithms for combinatorial problems," by D. Johnson, *JCSS*, 9, pp. 256–278, 1974. [exercises 11–16].

The approximation algorithm MSAT2 (exercise 12) for the maximum satisfiability problem has also been studied by K. Lieberherr. The *weight* of a CNF formula F is defined to be $w(F) = \Sigma_i 2^{-|C_i|}$ where $|C_i|$ is the number of literals in the ith clause of F. He shows that MSAT2 leaves at most $\lfloor w(F) \rfloor$ clauses unsatisfied. This result together with a generalization of MSAT2 appears in the paper:

"Interpretations of 2-satisfiable conjunctive normal forms," by K. Lieberherr, Florida State Univ., Tallahassee, to appear in *JACM.*

Lieberherr has also considered the notion of an optimal polynomial time approximation algorithm. A polynomial time approximation algorithm is optimal iff the problem of guaranteeing better solutions is NP-hard. MSAT2 (exercise 12) as well as some heuristics for other NP-hard problems are shown optimal in the following paper:

"Optimal heuristics for combinatorial optimization problems," by K. Lieberherr, Florida State University, Tallahassee, 1978.

Sahni and Gonzalez were the first to show the existence of NP-hard ϵ-approximate problems. Their results appear in the paper:

"*P*-complete approximation problems," by S. Sahni and T. Gonzalez, *JACM,* 23, pp. 555–565, 1976. [exercises 20–26, 29]

Garey and Johnson have shown that the ϵ-approximate graph coloring problem is NP-Hard for $\epsilon < 1$. Their result appears in the paper:

"The complexity of near optimal graph coloring," by M. Garey and D. Johnson, *JACM,* 23, pp. 43–49, 1976.

Some other NP-hard ϵ-approximate problems appear in:

"Traversal marker placement problems are NP-complete," by S. Maheshwari, University of Colorado, Technical report #CU-CS-092-76, 1976.

"Code generation for short/long address machines," by E. Robertson, Univ. of Wisconsin, MRC Report #1779, 1977.

A polynomial time approximation scheme for submodular set functions appears in:

"Best algorithms for approximating the maximum of a submodular set function," by G. Nemhauser and L. Wolsey, CORE discussion paper #7636, Universite Catholique De Louvain, Belgium, 1976.

An approximation scheme for scheduling tasks with precedence constraints appears in:

"Scheduling for maximum profits/minimum time," by O. Ibarra and C. Kim, *Math. of Oper. Res.,* to appear.

Ibarra and Kim were the first to discover the existence of fully polynomial time approximation schemes for NP-hard problems. Their work appears in the paper:

"Fast approximation algorithms for the knapsack and sum of subsets problems," by O. Ibarra and C. Kim, *JACM*, 22, pp. 463–468, 1975. [exercises 27–28]

This paper develops the $O(n(\log n + 1/\epsilon^2))$ algorithm for the 0/1 knapsack problem. An approximation scheme for the integer knapsack problem is also developed. E. Lawler has improved upon these schemes. He has obtained $O(n \log(1/\epsilon) + 1/\epsilon^4)$ and $O(n + 1/\epsilon^3)$ schemes for the knapsack and sum of subset problems. Lawler's work appears in the paper:

"Fast approximation algorithms for knapsack problems," by E. Lawler, *Proc. 18th Ann. Symp. on Foundations of Computer Science*, Rhode Island, pp. 206–213, 1977.

Fully polynomial time approximation schemes for many scheduling problems appear in the papers:

"Algorithms for scheduling independent tasks," by S. Sahni, *JACM*, 23, pp. 114–127, 1976. [exercises 30, 31, 33–37, 39–41].

"Exact and approximate algorithms for scheduling nonidentical processors," by E. Horowitz and S. Sahni, *JACM*, 23, pp. 317–327, 1976. [exercises 31 and 38]

Our discussion of the general techniques: rounding, interval partitioning and separation is from the paper:

"General techniques for combinatorial approximation," by S. Sahni, *Oper. Res.*, 25(6), pp. 920–936, 1977.

The notion of strongly NP-hard is due to Garey and Johnson. Theorem 12.16 is also due to them and appears in:

"'"Strong" NP-Completeness results: motivation, examples and implications," by M. Garey and D. Johnson, Bell Laboratories Report, Murray Hill, 1976.

The discussion on probabilistically good algorithms is based on the following papers:

"The probabilistic analysis of some combinatorial search algorithms," by R. Karp, University of California, Berkeley, Memo No. ERL-M581, April 1976.

"The fast approximate solution of hard combinatorial problems," by R. Karp,

Proc. Sixth Southeastern Conf. on Combinatorics, Graph Theory, and Computing, Winnipeg, 1975.

"Hamiltonian circuits in random graphs," by L. Posa, *Discrete Mathematics*, 14, pp. 359-364, 1976.

"Probabilistic analysis of partitioning algorithms for the traveling salesman problem in the plane," by R. Karp, *Math. of Oper. Res.*, 2(3), pp. 209–224, 1977.

The following paper contains a "good" algorithm for the general traveling salesperson problem. The algorithms have worked well on all problem instances tested. However, no statistical or probabilistic analysis has been made.

"An effective heuristic algorithm for the traveling salesman problem," by S. Lin and P. Kernighan, *Operations Research*, 21(2), 1973, 498–516.

Analysis of other probabilistically good algorithms appear in:

"Fast Probabilistic algorithms for hamiltonian circuits and matchings," by D. Angluin and L. Valiant, *Proc. 9th Annual Symp. on Theo. of Computing,* pp. 30–41, 1977.

"Analysis of the expected performance of algorithms for the partition problem," by C. Kim, Technical Report, University of Maryland, 1976.

"Maximization problems on graphs with edge weights chosen from a normal distribution," by G. Lueker *Proc. 10th Annual Symp. on Theo. of Computing*, pp. 13–18, 1978.

EXERCISES

1. The following NP-hard problems were defined in either Chapter 11 or 12. For each of these defined in the exercises, the exercise number appears in parenthesis. For each of these problems, clearly state the corresponding absolute approximation problem. (Some of the problems listed below were defined as decision problems. For these, there correspond obvious optimization problems that are also NP-hard. The absolute approximation problem is to be defined relative to the corresponding optimization problem.) Also, show that the corresponding absolute approximation problem is NP-hard.

   i) Node Cover
   ii) Set Cover (ex. 11.20)
   iii) Set Packing (ex. 11.43)
   iv) Feedback Node Set
   v) Feedback Arc Set (ex. 11.11)

vi) Chromatic Number
vii) Clique Cover (ex. 11.19)
viii) Max-Independent Set (see Section 12.6)
ix) Nonpreemptive scheduling of independent tasks to minimize finish time on $m > 1$ processors (Section 12.3)
x) Flow shop scheduling to minimize finish time ($m > 2$)
xi) Job shop scheduling to minimize finish time ($m > 1$)

2. Obtain an $O(n \log n)$ algorithm that implements the LPT scheduling rule.

3. Show that LPT schedules are optimal for all task sets that have an optimal schedule in which no more than two tasks are assigned to any processor.

4. A uniform processor system is a set of $m \geq 1$ processors. Processor i operates at a speed s_i, $s_i > 0$. If task i requires t_i units of processing then, it may be completed in t_i/s_i units of real time on processor p_i. When $s_i = 1$, $1 \leq i \leq m$ we have a system of identical processors (section 12.3). An MLPT schedule is defined to be any schedule obtained by assigning tasks to processors in order of nonincreasing processing times. When a task is being considered for assignment to a processor, it is assigned to that processor on which its finishing time will be earliest. Ties are broken by assigning the task to the processor with least index.

a) Let $m = 3$, $s_1 = 1$, $s_2 = 2$ and $s_3 = 3$. Let the number of tasks n be 6. $(t_1, t_2, t_3, t_4, t_5, t_6) = (9, 6, 3, 3, 2, 2)$. Obtain the MLPT schedule for this set of tasks. Is this an optimal schedule? If not obtain an optimal schedule.
b) Show that there exists a two processor system and a set I for which $|F^*(I) - \hat{F}(I)|/F^*(I) > 1/3 - 1/(3m)$. $\hat{F}(I)$ is the finish time of the MLPT schedule. Note that $1/3 - 1/(3m)$ is the bound for LPT schedules on identical processors.
c) Write an algorithm to obtain MLPT schedules. What is the time complexity of your algorithm?

5. Let I be any instance of the uniform processor scheduling problem. Let $\hat{F}(I)$ and $F^*(I)$ respectively be the finish times of MLPT and optimal schedules. Show that $\hat{F}(I)/F^*(I) \leq 2m/(m + 1)$ (see exercise 4).

6. For a uniform processor system (see exercises 4 and 5) show that when $m = 2$, $\hat{F}(I)/F^*(I) \leq (1 + \sqrt{17})/4$. Show that this is the best possible bound for $m = 2$.

7. Let $P_1, \ldots, P_m$ be a set of processors. Let $t_{i,j}$, $t_{i,j} > 0$ be the time needed to process task i if its processing is carried out on processor P_j, $1 \leq i \leq n$, $1 \leq j \leq m$. For a uniform processor system, $t_{i,j}/t_{i,k} = s_k/s_j$ where s_k and s_j are

the speeds of P_k and P_j respectively. In a system of *nonidentical processors*, such a relation need not exist. As an example, consider $n = 2$, $m = 2$ and

$$\begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 2 \end{pmatrix}.$$

If task 1 is processed on P_2 and task 2 on P_1, then the finish time is 3. If task 1 is processed on P_1 and task 2 on P_2, the finish time is 2. Show that if a schedule is constructed by assigning task i to processor j such that $t_{i,j} \leq t_{i,k}$, $1 \leq k \leq m$ then $\hat{F}(I)/F^*(I) \leq m$. $\hat{F}(I)$ and $F^*(I)$ are respectively the finish times of the schedule constructed and of an optimal schedule. Show that this bound is best possible for this algorithm.

8. For the scheduling problem of Exercise 7, define procedure A as:

```
procedure A
  f_j ← 0, 1 ≤ j ≤ m
  for i ← 1 to n do
    k ← least j such that f_j + t_{i,j} ≤ f_l + t_{i,l}, 1 ≤ l ≤ m
    f_k ← f_k + t_{i,k}
    print ('schedule task', i , 'on processor', k)
  repeat
end A
```

Algorithm 12.6 Scheduling

f_j is the current finish time on processor j. So, $\hat{F}(I) = \max_j \{f_j\}$. Show that $\hat{F}(I)/F^*(I) \leq m$ and this bound is best possible.

9. In the above exercise, first order the tasks so that $\min_j \{t_{i,j}\} \geq \min_j \{t_{i+1,j}\}$, $1 \leq i < n$. Then use algorithm A. Show that $\hat{F}(I)/F^*(I) \leq m$ and this bound is best possible.

10. Show that the results of exercise 8 hold even if the initial ordering is such that $\max_j \{t_{i,j}\} \geq \max_j \{t_{i+1,j}\}$, $1 \leq i < n$.

11. The satisfiability problem was introduced in chapter 11. Define maximum satisfiability to be the problem of determining a maximum subset of clauses that can be satisifed simultaneously. If a formula has p clauses, then all p clauses can be simultaneously satisfied iff the formula is satisfiable. For procedure MSAT, show that for every instance I, $|F^*(I) - \hat{F}(I)|/F^*(I) \leq 1/(k+1)$. k is the minimum number of literals in any clause of I. Show that this bound is best possible for this algorithm.

```
procedure MSAT(I)
  //approximation algorithm for maximum satisfiability. I is a formula.//
  //Let x_i, 1 ≤ i ≤ n be the variables in I and let C_i, 1 ≤ i ≤ p be the//
  //clauses.//
  CL ← φ  //set of clauses simultaneously satisfiable//
  LEFT ← {C_i | 1 ≤ i ≤ p}  //remaining clauses//
  LIT ← {x_i, x̂_i | 1 ≤ i ≤ n}  //set of all literals//
  while LIT contains a literal occurring in a clause in LEFT do
    let y be a literal in LIT that is in the most clauses of LEFT.
    let R be the subset of clauses in LEFT that contain y
    CL ← CL ∪ R; LEFT ← LEFT − R
    LIT ← LIT − {y, ŷ}
  repeat
  return (CL)
end MSAT
```

Algorithm 12.7 Procedure for Exercise 11

12. Show that if procedure MSAT2 is used for the maximum satisfiability problem of Exercise 11 then, $|F^*(I) - \hat{F}(I)|/F^*(I) \leq 1/2^k$ where k, $\hat{F}$ and F^* are as in Exercise 11.

```
procedure MSAT2 (I)
  //same function as MSAT//
  w(i) ← 2^{-|C_i|}, 1 ≤ i ≤ p  //weighting function |C_k| = number of//
                                //literals in C_i//
  CL ← φ; LEFT ← {C_i | 1 ≤ i ≤ p}
  LIT ← {x_i, x̂_i | 1 ≤ i ≤ n}
  while LIT contains a literal occurring in a clause in LEFT do
    let y ∈ LIT be such that y occurs in a clause in LEFT
    let R be the subset of clauses in LEFT containing y
    let S be the subset of clauses in LEFT containing ŷ
    if Σ_{C_i∈R} w(i) ≥ Σ_{C_i∈S} w(i) then CL ← CL ∪ R
                                            LEFT ← LEFT − R
                                            w(i) ← 2*w(i) for each C_i ∈ S
                                       else CL ← CL ∪ S
                                            LEFT ← LEFT − S
                                            w(i) ← 2*w(i) for each C_i ∈ R
    endif
    LIT ← LIT − {y, ŷ}
  repeat
  return (CL)
end MSAT2
```

Algorithm 12.8 Procedure for Exercise 12

13. Consider the set cover problem of Exercise 11.20. Show that if procedure SET_COVER is used for the optimization version of this problem then

$$\hat{F}(I)/F^*(I) \leq \sum_{1}^{k} (1/j)$$

where k is the maximum number of elements in any set. Show that this bound is best possible.

```
procedure SET_COVER(F)
  //S_i, 1 ≤ i ≤ m are the sets in F. |S_i| is the number of elements in S_i.//
  //|∪ S_i| = n//
  G ← ∪S_i; R_i ← S_i, 1 ≤ i ≤ m
  COV ← φ   //elements covered//
  T ← φ   //cover being constructed//
  while COV ≠ G do
    let R_j be such that |R_j| ≥ |R_q|, 1 ≤ q ≤ m
    COV ← COV ∪ R_j; T ← T ∪ S_j
    R_i ← R_i − R_j, 1 ≤ i ≤ m
  repeat
  return (T)
end SET_COVER
```

Algorithm 12.9 Procedure for Exercise 13

14. Consider a modified set cover problem (MSC) in which we are required to find a cover T such that $\Sigma_{S \in T} |S|$ is minimum.

(a) Show that exact cover αMSC (see Exercise 11.21)
(b) Show that procedure MSC is not an ϵ-approximate algorithm for this problem for any ϵ, $\epsilon > 0$.

```
procedure MSC (F)
  //same variables as in SET_COVER//
    T ← φ; LEFT ← {S_i | 1 ≤ i ≤ m}; G ← ∪S_i
    while G ≠ φ do
      let S_j be a set in LEFT such that
        |S_j − G|/|S_j ∩ G| ≤ |S_q − G|/|S_q ∩ G| for all S_q ∈ LEFT
      T ← T ∪ S_j; G ← G − S_j; LEFT ← LEFT − S_j
    repeat
    return (T)
end MSC
```

Algorithm 12.10 Procedure for Exercise 14

15. Consider the following heuristic for the max clique problem: i) delete from G a vertex that is not connected to every other vertex ii) repeat (i) until the remaining graph is a clique. Show that this heuristic does not result in an ϵ-approximate algorithm for the max clique problem for any ϵ, $0 < \epsilon < 1$.

16. For the max-clique problem, consider the following heuristic: (i) $S \leftarrow \phi$, (ii) add to S a vertex not in S that is connected to all vertices in S. If there is no such vertex then stop with S the approximate max clique, otherwise repeat (ii). Show that the algorithm resulting from this heuristic is not an ϵ-approximate algorithm for the max-clique problem for any ϵ, $\epsilon < 1$.

17. Show that procedure COLOR is not an ϵ-approximate coloring algorithm for the minimum colorability problem for any ϵ, $\epsilon > 0$.

```
procedure COLOR (G)
  //G = (V, E) is a graph with |V| = n vertices. COL(i) is the color to use//
  //for vertex i, 1 ≤ i ≤ n//
  i ← 1  //next color to use//
  j ← 0  //number of vertices colored//
  while j ≠ n do
    S ← φ  //vertices colored with color i//
    while there is an uncolored vertex, v, not adjacent to a vertex in S do
      COL(v) ← i; S ← S ∪ {v}; j ← j + 1
    repeat
    i ← i + 1
  repeat
  return (COL)
end COLOR
```

Algorithm 12.11 Procedure for Exercise 17

18. Show that if line 4 of Algorithm 12.3 is changed to PMAX $\leftarrow$ max {PMAX, $L(I, P, W, M, n)$} and line 1 of procedure L replaced by the line

$$S \leftarrow 0; \qquad i \leftarrow 1; \qquad T \leftarrow M$$

then the resulting algorithm is not ϵ-approximate for any ϵ, $0 < \epsilon < 1$. Note that the new heuristic constrains I to be outside the knapsack. The original heuristic constrains I to be inside the knapsack.

19. Show that procedure INDEP of Section 12.6 is not an ϵ-approximate algorithm for the maximum independent set problem for any ϵ, $0 < \epsilon < 1$.

20. Consider any tour for the traveling salesperson problem. Let city i_1 be the starting point. Assume the n cities appear in the tour in the order $i_1, i_2, i_3, \ldots, i_n, i_{n+1} = i_1$. Let $l(i_j, i_{j+1})$ be the length of edge $\langle i_j, i_{j+1} \rangle$. The arrival time Y_k at city i_k is

$$Y_k = \sum_{j=1}^{k-1} l(i_j, i_{j+1}), \; 1 < k \leq n+1$$

The mean arrival time $\overline{Y}$ is

$$\overline{Y} = \frac{1}{n} \sum_{k=2}^{n+1} Y_k$$

Show that the ϵ-approximate minimum mean arrival time problem is NP-hard for all ϵ, $\epsilon > 0$.

21. Let Y_k and $\overline{Y}$ be as in Exercise 20. The variance, σ, in arrival times is

$$\frac{1}{n} \sum_{2}^{n+1} (Y_k - \overline{Y})^2.$$

Show that the ϵ-approximate minimum variance time problem is NP-Hard for all ϵ, $\epsilon > 0$.

22. An edge disjoint cycle cover of an undirected graph G is a set of edge disjoint cycles such that every vertex is included in at least one cycle. The size of such a cycle cover is the number of cycles in it.

(a) Show that finding a minimum cycle cover of this type is NP-hard.

(b) Show that the ϵ-approximation version of this problem is NP-hard for all ϵ, $\epsilon > 0$.

23. Show that if the cycles in Exercise 22 are constrained to be vertex disjoint then the problem remains NP-Hard. Show that the ϵ-approximate version is NP-hard for all ϵ, $\epsilon > 0$.

24. Consider the partitioning problem:

Let $G = (V, E)$ be an undirected graph. Let f: $E \rightarrow Z$ be an edge weighting function and let w: $V \rightarrow Z$ be a vertex weighting function. Let k be a fixed integer, $k \geq 2$. The problem is to obtain k disjoint sets $S_1, \ldots, S_k$ such that:

(a) $\cup S_i = V$

(b) $S_i \cap S_j = \phi$ for $i \neq j$

(c) $\Sigma_{j \in S_i} w(j) \leq W; \quad 1 \leq i \leq k$

(d) $\sum_{i=1}^{k} \sum_{\substack{(u,v) \in E \\ u,v \in S_i}} f(u, v)$ is maximized

W is a number which may vary from instance to instance. This partitioning problem finds application in the minimization of the cost of interpage references between subroutines of a program. Show that the ϵ-approximate version of this problem is NP-hard for all ϵ, $0 < \epsilon < 1$.

25. Let $G = (V, E)$ be an undirected graph. Assume that the vertices represent documents. The edges are weighted such that $w(i, j)$ is the dissimilarity between documents i and j. It is desired to partition the vertices into $k \geq 3$ disjoint clusters such that

$$\sum_{i=1}^{k} \sum_{\substack{(u,v) \in E \\ u,v \in C_i}} w(u, v)$$

is minimized. C_i is the set of documents in cluster i. Show that the ϵ-approximate version of this problem is NP-hard for all ϵ, $\epsilon > 0$. Note that k is a fixed integer provided with each problem instance and may be different for different instances.

26. In one interpretation of the generalized assignment problem, we have m agents who have to perform n tasks. If agent i is assigned to perform task j then a cost c_{ij} is incurred. When agent i performs task j, r_{ij} units of his resources are used. Agent i has a total of b_i units of resource. The objective is to find an assignment of agents to tasks such that the total cost of the assignment is minimized and such that no agent requires more than his total available resource to complete the tasks he is assigned to. Only one agent may be assigned to a task.

Using x_{ij} to be a 0/1 variable such that $x_{ij} = 1$ if agent i is assigned to task j and $x_{ij} = 0$ otherwise, the generalized assignment problem may be formulated mathematically as:

$$\text{minimize} \quad \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$$

$$\text{subject to} \quad \sum_{j=1}^{n} r_{ij} x_{ij} \leq b_i, \quad 1 \leq i \leq m$$

$$\sum_{i=1}^{m} x_{ij} = 1, \quad 1 \leq j \leq n$$

$$x_{ij} = 0 \text{ or } 1, \text{ for all } i \text{ and } j$$

The constraints $\Sigma\, x_{ij} = 1$ ensure that exactly one agent is assigned to each task. Many other interpretations are possible for this problem.

Show that the corresponding ϵ-approximation problem is NP-hard for all ϵ, $\epsilon > 0$.

27. Consider the $O(n(\log n + 1/\epsilon^2))$ rounding algorithm for the 0/1 knapsack problem. Let $S^{(r)}$ be the final set of tuples in the solution of BIG. Show that no more than $(9/\epsilon^2)/q_i$ objects with rounded profit value q_i can contribute to any tuple in $S^{(r)}$. From this, conclude that BIG can have at most $(9/\epsilon^2)/q_i$ objects with rounded profit value q_i. Hence, $r \leq \Sigma\, (9/\epsilon^2)/q_i$ where q_i is in the range $[3/\epsilon, 9/\epsilon^2]$. Now, show that the time needed to obtain $S^{(r)}$ is $O(81/\epsilon^4 \ln (3/\epsilon))$. Use the relation

$$\sum_{3/\epsilon}^{9/\epsilon^2} (9/\epsilon^2)/q_i \simeq \int_{3/\epsilon}^{9/\epsilon^2} (9/\epsilon^2)\, dq_i/q_i = \frac{9}{\epsilon^2} \ln (3/\epsilon).$$

28. Write a SPARKS algorithm for the $O(n(\log n + 1/\epsilon^2))$ rounding scheme discussed in § 12.5. When solving BIG use three tuples (P, Q, W) such that $P = \Sigma\, p_i x_i$, $Q = \Sigma\, q_i x_i$ and $W = \Sigma\, w_i x_i$. Tuple (P_1, Q_1, W_1) dominates (P_2, Q_2, W_2) iff $Q_1 \geq Q_2$ and $W_1 \leq W_2$. In case $Q_1 = Q_2$ and $W_1 = W_2$ then an additional dominance criteria may be used. In this case the tuple (P_1, Q_1, W_1) dominates (P_2, Q_2, W_2) iff $P_1 > P_2$. Otherwise, (P_2, Q_2, W_2) dominates (P_1, Q_1, W_1). Show that your algorithm is of time complexity $O(n(\log n + 1/\epsilon^2))$.

29. Show that if we change the optimization function of Exercise 25 to maximize

$$\sum_{\substack{u \in C_i \\ v \notin C_i \\ (u,v) \in E}} w(u, v)$$

then there is a polynomial time ϵ-approximation algorithm for some ϵ, $0 < \epsilon < 1$.

30. Use separation to obtain a fully polynomial time approximation scheme for the independent task scheduling problem when $m = 2$ (see Section 12.4).

31. Do Exercise 30 for the case when the two processors operate at speeds s_1 and s_2, $s_1 \neq s_2$. See Exercise 4.

32. Do Exercise 30 for the case when the two processors are nonidentical (see Exercise 5).

33. Use separation to obtain a fully polynomial time approximation algorithm for the job sequencing with deadlines problem.

34. Use separation to obtain a fully polynomial time approximation scheme for the problem of obtaining two processor schedules with minimum mean weighted finish time (see Section 11.4). Assume that the two processors are identical.

35. Do Exercise 34 for the case when a minimum mean finish time schedule that has minimum finish time amongst all minimum mean finish time schedules is desired. Again, assume two identical processors.

36. Do Exercise 30 using rounding.

37. Do Exercise 31 using rounding.

38. Do Exercise 32 using rounding.

39. Do Exercise 33 using rounding.

40. Do Exercise 34 using rounding.

41. Do Exercise 35 using rounding.

42. Show that the following problems are strongly NP-hard

i) Max Clique
ii) Set Cover
iii) Node Cover
iv) Set Packing
v) Feedback Node Set
vi) Feedback Arc Set
vii) Chromatic Number
viii) Clique Cover