9.1. Identical machines: classical performance guarantees

In a groundbreaking paper, Graham proposed studying the worst-case behavior of heuristic procedures for optimization problems. The problem studied in this paper is \( P \mid \mid C_{\text{max}} \), and the heuristic analyzed is the simplest list scheduling (LS) rule, where the jobs are listed in any fixed order, and whenever a machine becomes idle, the next job from the list is assigned to begin execution on that machine. In fact, this simple rule does surprisingly well.

**Theorem 9.1.** For any instance of \( P \mid \mid C_{\text{max}} \), \( C_{\text{max}}(LS)/C_{\text{max}}^* \leq 2 - \frac{1}{m} \).

**Proof.** Let \( J_i \) be the last job to be completed in a list schedule (see Figure 9.1). Note that no machine can be idle before the time \( t \) when \( J_i \) starts processing. Intuitively, we can see that the list schedule is no longer than twice the optimal one by considering the two parts of the schedule, before and after time \( t \). Each part by itself is no longer than the optimal schedule: the length of the part afterwards is \( p_i \), which is clearly at most \( C_{\text{max}}^* \), and since no machine is idle from time 0 to time \( t \), the optimal schedule has length at least \( t \).

More formally, the fact that no machine is idle prior to time \( t \) implies that \( \sum_{j \neq i} p_j \geq mt \).

Therefore,

\[
C_{\text{max}}(LS) = t + p_i \leq \frac{1}{m} \sum_{j \neq i} p_j + p_i = \frac{1}{m} \sum_j p_j + \frac{m-1}{m} p_i.
\]

By using the inequalities,

\[
C_{\text{max}}^* \geq \frac{1}{m} \sum_j p_j, \quad C_{\text{max}}^* \geq p_i;
\]

we obtain the bound claimed above. \(\square\)

This bound is tight for any value of \( m \), as is shown by the following class of instances. Let \( n = m(m-1)+1 \), \( p_1 = \cdots = p_{n-1} = 1 \), \( p_n = m \), and consider the list \( (J_1, J_2, \ldots, J_n) \). The optimal schedule assigns job \( J_n \) to a machine by itself, and then balances the remaining \( m(m-1) \) among the \( m-1 \) other machines, and so \( C_{\text{max}}^* = m \). However, the list scheduling heuristic first balances the unit-length jobs, \( m-1 \) to each of \( m \) machines, and then is left assigning \( J_n \), yielding that \( C_{\text{max}}(LS) = 2m-1 \).

As the previous example suggests, list scheduling performs poorly when the last job to finish is extremely long, and the inequality (9.1) implies that this is the only way that poor schedules are generated. One natural way to try to prevent this is to use the list in which the jobs are sorted in order of nonincreasing processing times. This is called the longest processing time (LPT) rule. As the next theorem shows, it performs significantly better than the arbitrary list scheduling rule.

**Theorem 9.2.** For any instance of \( P \mid \mid C_{\text{max}} \), \( C_{\text{max}}(LPT)/C_{\text{max}}^* \leq \frac{4}{3} - \frac{1}{3m} \).

**Proof.** Suppose that the theorem is false, and consider a counterexample with the minimum number of jobs. Let job \( J_l \) be the job that completes last. If \( l < n \), then the instance consisting of jobs \( \{J_1, \ldots, J_l\} \) is a smaller counterexample, since the completion time of the LPT schedule is unchanged, whereas the optimal schedule length can only decrease by considering only a subset of the jobs. Hence, job \( J_n \) is the last to finish. We will consider two cases
separately: (i) \( p_n \leq C_{\text{max}}^*/3 \); and (ii) \( p_n > C_{\text{max}}^*/3 \). In case (i) it is easy to see that (9.1) immediately implies the theorem, since

\[
C_{\text{max}}(LPT) \leq \frac{1}{m} \sum_j p_j + \frac{m-1}{m} p_i \leq C_{\text{max}}^* + \frac{m-1}{m} \frac{C_{\text{max}}^*}{3} = \left( \frac{4}{3} - \frac{1}{3m} \right) C_{\text{max}}^*.
\]

It remains only to show that if case (ii) holds, then the \textit{LPT} rule delivers an optimal solution. This will be left as an exercise. \( \square \)

Observe that inequality (9.1) holds whenever there is no idle time prior to the start of the job that completes last. We will exploit this observation in the design of the following algorithm \( A_k \) and its analysis. Let \( k \) be fixed positive integer. Suppose that we partition the job set into two parts: the \textit{long} jobs and the \textit{short} jobs, where a job \( J_s \) is considered short if \( p_s \leq \frac{1}{km} \sum_j p_j \). Note that this implies that there are at most \( km \) long jobs. Enumerate all possible \( m^{km} \) schedules for the long jobs, and choose the shortest one. Extend this schedule by using list scheduling for the short jobs. As in the analysis of the \textit{LPT} rule, we focus on the last job \( J_l \) to finish, and distinguish between two cases. If \( J_l \) is a short job, then inequality (9.1) can be applied to obtain the bound

\[
C_{\text{max}}(A_k) \leq C_{\text{max}}^* + p_l \leq C_{\text{max}}^* + \frac{1}{km} \sum_j p_j \leq (1 + \frac{1}{k}) C_{\text{max}}^*.
\]

If \( J_l \) is a long job, then the schedule delivered by the algorithm is optimal, since \( C_{\text{max}}(A_k) \) is equal to the length of the optimal schedule for just the long jobs, which is clearly no more than \( C_{\text{max}}^* \). Since algorithm \( A_k \) can easily be implemented to run in \( O(n \log m + (k+m) m^{km}) \) time, we have obtained the following theorem.

\textbf{Theorem 9.3.} For any fixed number of machines, \( m \), the family of algorithms \( \{ A_k \} \) forms a polynomial approximation scheme for \( Pm \mid \mid C_{\text{max}} \). \( \square \)

For the special case when \( m=2 \), there is another algorithm which has the same performance guarantee as the \textit{LPT} rule (see Exercise 9.3), but has been empirically observed to give much better solutions. The main idea of this \textit{differencing method} (D) is that two jobs are assigned at a time, one to each machine; the decision as to which job is assigned to which machine is made by scheduling a smaller instance, where both jobs are replaced by a single artificial job with processing requirement equal to the difference of their processing requirements. More precisely, if there is only one job, assign it to \( M_1 \); otherwise, if \( J_j \) and \( J_k \), respectively, are the jobs with the largest and second largest processing times, replace these by a new job \( J_l \) with \( p_l = p_j - p_k \), and call this procedure recursively; to convert the resulting schedule to one for the original instance, if \( J_l \) is assigned to \( M_1 \), then instead assign \( J_j \) to \( M_1 \), and \( J_k \) to the other machine.

\textbf{Exercises}

9.1. For each \( m>0 \), prove that the bound given in Theorem 9.2 is tight.

9.2. Use the following approach to yield a similar polynomial approximation scheme to the scheme \( \{ A_k \} \) analyzed in Theorem 9.3: schedule the \( l \) longest jobs optimally and complete the schedule using list scheduling.
9.3. Prove that for any instance of \( P2||C_{\text{max}} \), \( C_{\text{max}}(D)/C_{\text{max}}^* \leq 7/6 \), and show that this bound is tight.

9.2. Identical machines: using binary search for guarantees

The approximation algorithms discussed above are only the first step towards understanding how to efficiently obtain good solutions for \( P || C_{\text{max}} \). For the rest of this section, it will be useful to consider the decision version of \( P || C_{\text{max}} \): given a deadline \( d \), does there exist a schedule in which all jobs are completed by time \( d \)? An equivalent way in which to view this question is the following: given a collection of \( n \) pieces of specified sizes, is it possible to pack these pieces in \( m \) bins of size \( d \), so that each bin contains pieces of total size no more than \( d \)?

The problem of finding the minimum number of bins in which a given set of pieces can fit is called the bin-packing problem. For this problem, there is an approximation algorithm, called first fit decreasing (FFD), that is analogous to the LPT rule: list the pieces in order of nonincreasing size, and iteratively extend the current packing by placing the next piece in the first bin in which it fits (i.e., without exceeding the capacity \( d \)) or if there is no such bin, start a new one.

The FFD procedure can also be iteratively used within a heuristic for \( P || C_{\text{max}} \) as follows:

\[
S := \text{a list schedule;}
UB := C_{\text{max}}(LS);
LB := \max\{\max_j p_j, \frac{1}{m} \sum_j p_j\};
\]

repeat
\[
d := \left\lfloor \frac{LB + UB}{2} \right\rfloor;
\]
run the FFD algorithm to pack the jobs in bins of size \( d \); (*)
if more than \( m \) bins are used by the packing (**)
then
\[
LB := d + 1
\]
else
\[
UB := d;
S := \text{new schedule;}
\]
until \( LB = UB \).

In specifying this procedure, which is called the mult-fit (MF) algorithm, we are assuming that all data are integral. This algorithm has a better performance guarantee than is known for any variant of list scheduling.

**Theorem 9.4.** When mult-fit is run for only \( l \) iterations of binary search, then for any instance of \( P || C_{\text{max}} \), \( C_{\text{max}}(MF)/C_{\text{max}}^* \leq 1.2 + 2^{-l} \). □

A surprising aspect of the proof of this theorem is that it does not make use of the known analysis of the performance of FFD as a bin-packing algorithm; these two performance guarantees, while similar values, are unrelated.
In the binary search procedure given above, the statements indicated by (*) and (**) can be viewed as approximately answering the decision version of $P \mid \mid C_{\text{max}}$. Another way to produce such an approximation, which we will call a $\rho$-relaxed decision procedure, is as follows: for each instance and any deadline $d$, (i) the procedure either outputs 'no' or produces a schedule with $C_{\text{max}} \leq \rho d$, and (ii) if the output is 'no', then there is no schedule with $C_{\text{max}} \leq d$. This suggests a general framework for constructing approximation algorithms for $P \mid \mid C_{\text{max}}$, where statements (*) and (**) are replaced by:

run a $\rho$-relaxed decision procedure to schedule the jobs within the deadline $d$; (†)
if the output is ‘no’ (††)

Let $B_\rho$ denote the binary search procedure when a $\rho$-relaxed decision procedure is used in step indicated by (†). As we shall see, this framework can equally well be applied to more general problems, and the following lemma gives the main reason for its importance.

**Lemma 9.1.** For any instance of $P \mid \mid C_{\text{max}}$, $C_{\text{max}}(B_\rho)/C^*_{\text{max}} \leq \rho$. Furthermore, if the $\rho$-relaxed decision procedure runs in polynomial time, then $B_\rho$ runs in polynomial time.

**Proof.** To prove the lemma, we show that the algorithm maintains two invariants: at each iteration, (i) the schedule $S$ has $C_{\text{max}} \leq \rho UB$, and (ii) $C^*_{\text{max}} \geq LB$. Observe that these two conditions suffice, since $UB = LB$ at termination, and so the final schedule $S$ has $C_{\text{max}} \leq \rho UB = \rho LB \leq \rho C^*_{\text{max}}$.

It is easy to see that both (i) and (ii) hold initially. Suppose that in a particular iteration $UB$ is updated to $d$. In that case, the relaxed decision procedure has produced a schedule with $C_{\text{max}}$ at most $\rho d$, and so (i) still holds. Now suppose that $LB$ is updated to $d$. In this case, the relaxed decision procedure has output ‘no’ and so no feasible schedule completes by the deadline $d$. Since the data are integers, no schedule can complete earlier than $d+1$.

Furthermore, since the difference between $UB$ and $LB$ after $l$ iterations is an integer and is bounded from above by $2^{-l}C^*_{\text{max}}$, the algorithm $B_\rho$ must terminate after a polynomial number of iterations. Therefore, if the $\rho$-relaxed decision procedure runs in polynomial time, the entire binary search is completed in polynomial time. □

As a consequence of this theorem, we need only focus on constructing polynomial-time $\rho$-relaxed decision procedures in order to design efficient approximation algorithms for $P \mid \mid C_{\text{max}}$. We will show how to use this framework to construct a polynomial approximation scheme for $P \mid \mid C_{\text{max}}$ (where the number of machines is an input). By Theorem 7.7, this is the best we can hope for, since no fully polynomial approximation scheme exists unless $P=NP$.

Let $k$ be a fixed positive integer; we will show how to construct a $(1 + \frac{1}{k})$-relaxed decision procedure. Consider an instance of $P \mid \mid C_{\text{max}}$, along with a deadline $d$ that is at least $\max_j p_j$. Once again, we will partition the jobs into short jobs and long jobs: in this case, $J_j$ is called short if $p_j \leq d/k$.

For the time being, suppose that we are fortunate, and there are no short jobs. In the polynomial approximation scheme for a fixed number of machines, we solved such an instance to optimality by complete enumeration. In order to reduce the running time to be polynomial in $m$, we will produce only a near-optimal solution for the long jobs. To obtain such a
schedule, we will use make crucial use of a dynamic programming algorithm to solve a special case of the bin-packing problem. It is not hard to see that when the bin-packing problem is restricted to the class of instances where there are at most $c_1$ distinct piece sizes and at most $c_2$ pieces may be packed in one bin, it can be solved in $O((c_2 n)^{c_1})$ time (see Exercise 9.4). Observe that if $c_1$ is a constant, then this is a polynomial bound.

The relaxed decision procedure works as follows. Round down each processing time $p_j$ to the nearest multiple of $d/k^2$, that is, set $\bar{p}_j := \left\lfloor \frac{d}{k^2} \frac{p_j k^2}{d} \right\rfloor$. Use the claimed algorithm for the special case of the bin-packing problem to find the minimum number of machines that suffice to schedule all of the rounded jobs, at most $k-1$ per machine. If the optimal packing uses more than $m$ machines, output ‘no’; otherwise consider the packing as a schedule for the original instance.

We want to show that this procedure satisfies the two properties of a $(1+\frac{1}{k})$-relaxed decision procedure. Consider a schedule produced by this algorithm. For any job, the difference between the rounded processing time and the original processing time is at most $d/k^2$. Since each machine is assigned fewer than $k$ jobs, the cumulative effect of rounding the processing times is less than $k(d/k^2) = d/k$. Hence, all jobs are completed by time $(1+\frac{1}{k})d$. On the other hand, suppose that the original instance of $P_{\mid \mid C_{\text{max}}}$ has a feasible schedule that completes by the deadline $d$. Clearly, at most $k-1$ long jobs are scheduled on each machine. Hence, there must be a bin-packing for the rounded instance that uses at most $m$ machines, such that each machine is assigned at most $k-1$ jobs. Consequently, if the algorithm outputs ‘no’, there does not exist a schedule for the original instance that completes by the deadline $d$.

Note that the rounding ensures that there are at most $k^2+1$ distinct sizes of rounded jobs. Since $k$ is a fixed integer, the bin-packing algorithm runs in polynomial time, and so we have shown that this procedure is a polynomial-time $(1+\frac{1}{k})$-relaxed decision procedure.

Unfortunately, the previous discussion was based on the unfounded assumption that there were no short jobs. If there are short jobs, first check if $\sum_j p_j > md$, and if so, output ‘no’. Otherwise, temporarily delete the short jobs, and try to compute a schedule for the modified instance with the algorithm given above. Note that if the original instance has a schedule that completes by $d$, then a schedule that completes by $(1+\frac{1}{k})d$ will be produced for this subset of jobs. Once again, extend this schedule using list scheduling.

If this algorithm outputs ‘no’, then clearly no feasible solution exists. To complete the proof that this is a $(1+\frac{1}{k})$-relaxed decision procedure, we need only show that if a schedule is produced, then all short jobs are completed by $(1+\frac{1}{k})d$. Suppose that a short job $J_i$ finishes after $(1+\frac{1}{k})d$. Since $p_i \leq d/k$, this job must have started processing after time $d$, and thus in this schedule, all machines are processing jobs until after time $d$. But this is impossible, since then $\sum_j p_j > md$.

Summarizing, we have constructed the following $(1+\frac{1}{k})$-relaxed decision procedure $D_k$:
if \( \sum_j p_j > md \)

then

output ‘no’

else

temporarily focus on the subset of jobs \( J_L := \{ J_j | p_j > d/k \} \);

round down the processing time of each \( J_j \in J_L \) to the nearest multiple of \( d/k^2 \);

using the rounded data, find the optimal packing of \( J_L \) into bins of capacity \( d \) where each bin is assigned fewer than \( k \) jobs;

if the number of bins used is more than \( m \)

then

output ‘no’

else

interpret the packing as a schedule for the original data;

extend the schedule to include each \( J_j \notin J_L \) by using list scheduling.

Furthermore, we have already seen that for any fixed value of \( k \), the algorithm can be implemented to run in polynomial time, and so we have proved the following result.

**Theorem 9.5.** For any fixed integer \( k > 0 \), the algorithm \( D_k \) is a polynomial-time \( (1 + \frac{1}{k}) \)-relaxed decision procedure for \( P || C_{\text{max}} \). □

**Corollary 9.1.** If the algorithm \( B_{\alpha} \) uses the \( \alpha \)-relaxed decision procedure \( D_{\alpha} \), where \( \alpha = \frac{1}{\rho - 1} \), then the family of algorithms \( \{ B_{\rho} \} \) forms a polynomial approximation scheme for \( P || C_{\text{max}} \). □

**Exercises**

9.4. Give an \( O((c_2 n)^c) \) algorithm to find an optimal solution to the bin-packing problem where there are at most at \( c_1 \) distinct piece sizes and at most \( c_2 \) pieces may be packed in one bin.

9.5. Prove that the following algorithm is a \((5/4)\)-relaxed decision procedure for \( P || C_{\text{max}} \). The algorithm consists of six stages. In stage 1, check if \( \sum_j p_j > md \), and if so, output ‘no’ and halt. For stages 2 through 5, each \( J_j \) with \( p_j < d/4 \) is temporarily set aside, and the instance consisting of the remaining jobs is considered. In stage 2, while there exists a \( J_j \) with \( p_j \geq d/2 \), find the longest job \( J_k \) such that \( p_j + p_k < d \), and schedule \( J_j \) and \( J_k \) by themselves on one machine. Stages 3, 4 and 5 are executed \( m \) times: if none of these attempts succeeds in scheduling all \( J_j \) with \( p_j \geq d/4 \) on \( m \) machines, then output ‘no’ and halt. In the \( k \)th attempt, \( k = 1, \ldots, m \), for stage 3, find the \( 2k \) unscheduled jobs with the longest processing requirement, and schedule them two per machine in an arbitrary way. In stage 4, while there exists \( J_j \) with \( p_j > 5d/12 \), schedule \( J_j \) on a machine with the two longest unscheduled jobs with processing times at most \( 3d/8 \). In stage 5, schedule the remaining jobs three per machine in an arbitrary way. If a schedule has been constructed for all of the jobs with processing requirement at least \( d/4 \) using no more than \( m \) machines, stage 6 extends it for the remaining jobs by using list scheduling.
9.6. Give a \((6/5)\)-relaxed decision procedure for \(P \mid || C_{\text{max}}\) that runs in \(O(mn)\) time. Can you improve this to \(O(n \log n)\)?)

9.7. (a) Consider the following variant of \(P \mid || C_{\text{max}}\), which we denote by \(P \mid \text{mem} \mid C_{\text{max}}\). Each job \(J_j\) has both a processing requirement \(p_j\) and a memory requirement \(\rho_j\). Machine \(M_i\) has the capacity to process jobs with memory requirement at most \(\kappa_i\). The aim is to find a schedule that minimizes the maximum completion time, subject to the constraint that each job is assigned to a machine with sufficient memory capacity to process it. Consider the largest memory first (LMF) scheduling rule in which the jobs are listed in order of nondecreasing memory requirement: when a machine becomes idle it chooses the next job on the list for which it has sufficient capacity. Prove that for any instance of this problem, 
\[
C_{\text{max}(\text{LMF})/C_{\text{max}}^*} \leq 2 - \frac{1}{m}.
\]
(b) Suppose that there is a total of \(R\) units of memory. The memory can be partitioned among the \(m\) machines in any way, but it must be done only once, prior to scheduling the particular instance. The aim is to find the optimal partition of the memory so that the optimum for the resulting \(P \mid \text{mem} \mid C_{\text{max}}\) instance is minimized. Show how to compute a partition of the memory so that 
\[
C_{\text{max}(\text{LMF})/C_{\text{max}}^*} \leq 2 - \frac{1}{m},
\]
where \(C_{\text{max}}^*\) is with respect to the optimal memory partition. (Hint: Suppose that the jobs are indexed in order of nondecreasing memory requirement, and let \(LB\) be a lower bound on \(C_{\text{max}}^*\). For each \(i = 1, \ldots, m\), let \(k(i)\) be the smallest integer \(k\) such that \(\sum_{j=1}^{k} p_j > (i-1)LB\). Then, if the machines are indexed in order of nondecreasing memory capacity, \(\kappa_i\) must be at least \(\rho_{k(i)}\) in order to achieve the lower bound. If there is sufficient memory to assign to each \(M_i\), the corresponding \(\rho_{k(i)}\) units of memory, then the algorithm terminates; otherwise, this fact can be used to increase \(LB\), and a new iteration is begun.)

9.8. It is not hard to extend the performance guarantees for variants of list scheduling to \(P \mid r_j \mid L_{\text{max}}\).
(a) Show that if the list scheduling rule is used with the jobs given in EDD order, then for any instance of \(P \mid r_j \mid L_{\text{max}}\), \(L_{\text{max}}(\text{EDD}) - L_{\text{max}}^* \leq 2m - 1 - \max_j p_j\).
(b) Show that, in the delivery time model (as in Chapter 4), for any instance of \(P \mid r_j \mid L_{\text{max}}\), 
\[
L_{\text{max}}(\text{LS})/L_{\text{max}}^* < 2.
\]

9.3. Performance guarantees for uniform machines

Unfortunately, all machines are not created equal, and may work at different speeds. Order the machines so that the speeds are nonincreasing; that is, \(s_1 \geq s_2 \geq \cdots \geq s_m\). If \(p_j\) denotes the processing requirement of job \(J_j\), then processing \(J_j\) on \(M_i\) takes \(p_j/s_i\) time units.

In this more general model, the simplest algorithms do not work quite as well. For example, consider the list scheduling rule. We can analyze this procedure using the same techniques as we used for the case of identical machines. Consider the job \(J_i\) that completes last in some list schedule. Let \(t\) denote the time at which \(J_i\) begins processing; no machine is idle prior to time \(t\). Since machine \(M_i\) has the capacity to process \(ts_i\) units by time \(t\), it follows that 
\[
\sum_{j \neq i} p_j \geq t \sum_i s_i.
\]
Since there must be sufficient capacity to process all of the jobs by $C_{\text{max}}^*$,

$$C_{\text{max}}^* \sum_{j} s_j \geq \sum_{j} p_j.$$ 

The processing of $J_i$ takes at least $p_i/s_j$ time units, and so $C_{\text{max}}^* \geq p_i/s_j$. On the other hand, $J_i$ is certainly processed in the list schedule within $p_i/s_m$ time units. Combining these pieces, we see that

$$C_{\text{max}}(LS) \leq t + p_i/s_m \leq \frac{\sum_{j \neq i} p_j}{\sum_{j} s_j} + \frac{p_i}{s_m} = \frac{\sum_{j} p_j}{\sum_{j} s_j} + p_i \left( \frac{1}{s_m} - \frac{1}{\sum_{j} s_j} \right) \leq C_{\text{max}}^* + C_{\text{max}}^* \frac{s_j}{s_m} \left( \frac{1}{\sum_{j} s_j} \right).$$

Thus, we have shown the following theorem.

**Theorem 9.6.** For any instance of $Q || C_{\text{max}}$, $C_{\text{max}}(LS)/C_{\text{max}}^* \leq 1 + s_j/s_m - s_j/\sum_{j} s_j$. \( \square \)

In fact, this bound is tight, and by an appropriate choice of machine speeds may exceed any constant (see Exercise 9.9).

If we were to follow the approach adopted for identical machines, we would next focus on ways to construct a special list for which the list scheduling rule can then be run. However, unlike $P || C_{\text{max}}$, there need not be a list for which the list scheduling rule gives the optimal schedule for $Q || C_{\text{max}}$. The reason for this is that the list scheduling rule always delivers a schedule in which no machine is idle while there is still an unscheduled job. It is easy to construct instances for which every optimal schedule contains such unforced idle time (see Exercise 9.10).

The following simple variant of list scheduling may create unforced idle time: schedule the jobs in the order of the list, always assigning the next job to the machine on which it would complete earliest. While this strategy has a better performance guarantee, it still does not deliver solutions for $Q || C_{\text{max}}$ that are within a constant factor of the optimum. However, the LPT variant of this, which we shall call LPT', works quite well.

**Theorem 9.7.** For any instance of $Q || C_{\text{max}}$, $C_{\text{max}}(LPT')/C_{\text{max}}^* \leq 2 - \frac{1}{m}$.

**Proof.** Suppose that the theorem is false, and consider a counterexample in which the sum of the number of jobs and the number of machines is minimum. Let $m$ and $n$, respectively, denote the number of machines and the number of jobs in this counterexample. We shall assume that the jobs are indexed so that $p_1 \geq p_2 \geq \cdots \geq p_n$.

Consider the schedule generated by LPT'. Suppose that $J_l$ is the last job to finish, where $l \neq n$. In that case, we can obtain a smaller counterexample by deleting all jobs $J_j$, $j > l$, since that does not change the maximum completion time of the LPT' schedule and does not increase $C_{\text{max}}^*$. Thus, we may assume that $J_n$ is the last job to finish. Furthermore, suppose that no job is scheduled on machine $M_i$. Since $J_n$ is the last job to finish, and it is not scheduled on $M_i$, $p_n/s_n \geq C_{\text{max}}(LPT') > C_{\text{max}}^*$. Therefore, $M_i$ must be completely idle in any optimal schedule as well. But then, if we delete $M_i$, we have not changed either the LPT' schedule or the optimal schedule, and we obtain a smaller counterexample, which is a contradiction.

By the way in which $J_n$ is assigned to a machine, we see that for each $M_i$,
\[ C_{\text{max}}(LPT')s_i \leq p_n + \sum_{j \neq n} p_j. \]

Summing this inequality for all \( i = 1, \ldots, m \), we get
\[ C_{\text{max}}(LPT')\sum_i s_i \leq mp_n + \sum_{j \neq n} p_j = (m-1)p_n + \sum_j p_j. \]

Since no machine is idle, \( m \leq n \), and thus \( mp_n \leq \sum_j p_j \). From these inequalities, we see that
\[ C_{\text{max}}(LPT') \leq (2 - \frac{1}{m})\frac{\sum_j p_j}{\sum_i s_i} \leq (2 - \frac{1}{m})C^*, \]

which shows that our instance is not a counterexample. \( \Box \)

Unlike all of the analyses that we have seen thus far, the bound given in Theorem 9.7 is not tight. In fact, by using additional structural information, it is possible to show that the performance ratio is no more than 19/12. This is also not known to be tight; for the worst example known, \( C_{\text{max}}(LPT')/C^* = 1.52 \).

> From the previous section, recall the notion of a \( \rho \)-relaxed decision procedure and the binary search procedure \( B_\rho \) in which it was used. The definition of a \( \rho \)-relaxed decision procedure can be applied equally well to \( Q \mid \mid C_{\text{max}} \). Furthermore, we can adapt the initial lower and upper bounds to obtain the following analogue of \( B_\rho \).

\[ S := \text{the LPT' schedule}; \]
\[ UB := C_{\text{max}}(LPT'); \]
\[ LB := \max\{p_i/s_i, (\sum_j p_j)/(\sum_i s_i), C_{\text{max}}(LPT')/2\}; \]

repeat
\[ d := \frac{LB + UB}{2}; \]
run a \( \rho \)-relaxed decision procedure to schedule the jobs within the deadline \( d \);
if the output is 'no'
then
\[ LB := d + 1; \]
else
\[ UB := d; \]
\[ S := \text{new schedule}; \]
until \( LB = UB \).

It is easy to see that the analogue of Lemma 9.1 is also valid for \( Q \mid \mid C_{\text{max}} \). We will give a simple \( 3/2 \)-relaxed decision procedure for \( Q \mid \mid C_{\text{max}} \), which therefore yields a \( 3/2 \)-approximation algorithm for \( Q \mid \mid C_{\text{max}} \).

Our \( 3/2 \)-decision procedure is a recursive algorithm; that is, the procedure calls itself as a subroutine, but for a smaller instance. Let this procedure be called \( \text{Recurse}(J, m, d) \) where \( J \) denotes the set of jobs, \( m \) indicates that there are machines \( M_1, M_2, \ldots, M_m \) (where \( s_1 \geq s_2 \geq \cdots \geq s_m \)) and \( d \) is the deadline being considered. Recall that the output must either be a schedule that completes by \((3/2)d\) or 'no', where 'no' is output only if there is no schedule that completes within the deadline \( d \).

The procedure first checks if the capacity of the machines is sufficient for the given jobs: if \( \sum_{j \in J} p_j > d \sum_{i=1}^m s_i \), then output 'no' and halt. Otherwise, if \( m = 1 \), all jobs in \( J \) are assigned
to machine $M_1$, and the procedure ends. If $m > 1$, let $S$ be the set of all jobs $J_j \in J$ such that $p_j \leq ds_m/2$. If there are no jobs $J_j$ in $J-S$ with $p_j \leq ds_m$, call $Recursion(J-S,m-1,d)$. Otherwise, among all such jobs in $J-S$, let $J_r$ denote one of these with maximum processing requirement. Assign $J_r$ to be scheduled on $M_m$ and call $Recursion(J-S-\{J_r\},m-1,d)$. If the procedure has not output ‘no’ and halted, extend the schedule for $J-S$ on $M_1, \ldots, M_m$ to include $S$ by using list scheduling; that is, order the jobs of $S$ arbitrarily, and assign the next job to be scheduled on the machine that is currently finishing earliest.

**Theorem 9.8.** $Recursion(\{J_1, \ldots, J_n\},m,d)$ is a $3/2$-relaxed decision procedure for $Q || C_{\text{max}}$.

**Proof:** We first show that if there is a schedule that completes within time $d$, then the procedure outputs a schedule. To prove this, we will prove the following important claim: if the original instance has a schedule that completes by time $d$, then so must the smaller instance for which the recursive call is made. Given this, it is clear that no recursive call will ever output ‘no’ for an instance with a feasible deadline, and so a schedule will be output. If $d$ is a feasible deadline to schedule $J$ on the fastest $m$ machines, then $J-S$ can also be scheduled on these $m$ machines by time $d$. But for this subinstance, each job $J_j$ has $p_j > ds_m/2$, and so in any schedule that completes by time $d$, at most one of these jobs is scheduled on $M_m$. If all jobs have processing requirement more than $ds_m$, then clearly, $M_m$ must be completely idle. Otherwise, an interchange argument shows that there always exists a feasible schedule in which $J_r$ is scheduled on $M_m$. In either case, we have shown that the subinstance for which the recursive call is made has a schedule that completes within time $d$.

For the second half of the proof, we will show by induction on $m$ that if a schedule is produced, then it completes all jobs before $(3/2)d$. It is a simple exercise to verify the claim for $m=1$. Suppose that the algorithm outputs a schedule. Then the recursive call must also output a schedule, and by the inductive hypothesis, all jobs in $J-S$ are completed before $(3/2)d$. The job $J_r$, if it exists, clearly completes by time $d$. Since the machines have sufficient capacity to schedule all jobs within $d$ time units, then in any partial schedule, the machine that is currently finishing earliest must complete its jobs before time $d$. For each job $J_j \in S$, $p_j \leq ds_m/2$, and so it takes at most $d/2$ time units on any machine. But then each job in the list is assigned to a machine on which it completes before $(3/2)d$. $\square$

There is an analogue of the multifit (MF) procedure for $Q || C_{\text{max}}$ as well. This algorithm does still better.

**Theorem 9.9.** When multifit is run for only $l$ iterations of binary search, then for any instance of $Q || C_{\text{max}}$, $C_{\text{max}}(MF)/C_{\text{max}} \leq 1.4 + 2^{-l}$. $\square$

Finally, the approach that employs a $\rho$-relaxed decision procedure to construct a polynomial approximation scheme for $P || C_{\text{max}}$ can also be extended to the case of $Q || C_{\text{max}}$. The polynomial approximation scheme for $Q || C_{\text{max}}$ is substantially more complicated, and beyond the scope of this book.

**Exercises**

9.9. Give a family of instances of $Q || C_{\text{max}}$ that shows there does not exist a constant $c$ such that $C_{\text{max}}(LS)/C_{\text{max}} \leq c$. 


9.10. Give an instance of $Q | | C_{\text{max}}$ for which all optimal solutions have unforced idle time.

9.11. Show that the following algorithm is a 2-relaxed decision procedure for $Q | | C_{\text{max}}$. For each machine, we will construct a list of jobs so that these lists form a partition of the set of jobs. Assign each job $J_j$ to the list of the slowest machine $M_i$ such that $p_j \leq s_i d$. When a machine $M_i$ becomes idle, schedule the next job in its list on $M_i$; if its list is empty, find the next slowest machine with a non-empty list, and schedule the next job from that list on $M_i$. (When a job is scheduled, delete it from its list.) If the schedule constructed has $C_{\text{max}} > 2d$, instead output 'no'.

9.4. Unrelated machines: performance guarantees

When the machines are unrelated, it is possible that $J_1$ takes much longer on $M_2$ than on $M_1$, whereas $J_2$ takes much longer on $M_1$. This generalization makes it substantially more difficult to obtain good near-optimal solutions. In this section, we shall see that this statement can be made much more precise. Throughout the next two sections, the time that it takes to process job $J_j$ on machine $M_i$ will be denoted by $p_{ij} = p_j / s_{ij}$, which we assume to be integral.

The worst-case performance of list scheduling or any other known simple scheduling rule is pretty dismal. One particularly naive algorithm is to assign each job to be scheduled on the machine on which it takes the least time. Clearly, this schedule completes within $T = \sum_j \min_i \{ p_{ij} \}$. In addition, one can view $T$ as the minimum total requirement of the jobs, and since the best one could hope for is to balance this load evenly over all machines, $C_{\text{max}} \geq T/m$. Therefore, this greedy (G) algorithm satisfies $C_{\text{max}}(G)/C_{\text{max}}^* \leq m$, and this bound is tight (see Exercise 9.12).

A clever variant of list scheduling is significantly better. Maintain a separate list for each machine, and sort all $n$ jobs for $M_j$'s list in order of nondecreasing relative speed, $p_{ij} / \min_k p_{kj}$, $j = 1, \ldots, n$. Whenever a machine is idle, assign the next unscheduled job in its list, unless the smallest ratio of an unscheduled job is more than $\sqrt{m}$; in this case, no further jobs are scheduled on that machine. This relative speed (RS) rule can be shown to guarantee the following performance.

Theorem 9.10. For any instance of $R | | C_{\text{max}}$, $C_{\text{max}}(RS)/C_{\text{max}}^* \leq 2.5 \sqrt{m} + 1 + 1/(2 \sqrt{m})$. □

This bound is tight up to a constant factor.

Linear programming can be used to construct a much more effective procedure. One natural way to formulate $R | | C_{\text{max}}$ as an integer linear programming problem is as follows:

$$\min C_{\text{max}}$$

subject to

$$\sum_j p_{ij} x_{ij} \leq C_{\text{max}}, \quad \text{for } i = 1, \ldots, m,$$

$$\sum_i x_{ij} = 1, \quad \text{for } j = 1, \ldots, n,$$

$$x_{ij} \in \{0,1\}, \quad \text{for } i = 1, \ldots, m, j = 1, \ldots, n,$$

where $x_{ij}$ indicates if job $J_j$ is assigned to $M_i$. If each integer constraint (9.4) is relaxed to the linear constraint $x_{ij} \geq 0$, then we can solve the resulting linear program $LP_1$ to obtain a lower
bound on $C^*_{\text{max}}$.

In fact, $LP_1$ can also be used to obtain good integer solutions as well. The main idea is to obtain an optimal solution $x^*$ to $LP_1$; then, if $x_{ij}^*=1$, assign $J_j$ to machine $M_i$, and deal with the unassigned jobs in some other manner. We may assume that $x^*$ is an extreme point of $LP_1$, and we will use this in a critical way.

Consider any extreme point $\hat{x}$ of the feasible region of the linear program $LP_1$. There are $mn+1$ variables in $LP_1$, and so there must be $mn+1$ linearly independent constraints of $LP_1$ for which $\hat{x}$ is the unique feasible solution that satisfies these constraints with equality. However, other than those of the form $x_{ij}\geq 0$, there are only $m+n$ constraints. Therefore, all but $m+n-1$ of the components of $\hat{x}$ must equal 0. In order to satisfy the constraints (9.3), at least one component $\hat{x}_{ij}$ must be positive for each $j=1, \ldots, n$. There are at most $m-1$ other positive components, and so for all but at most $m-1$ of the constraints (9.3), exactly one variable is positive, and hence equal to 1. Therefore, the approach suggested above will be able to immediately assign all but at most $m-1$ jobs.

If the number of machines is small, then the schedule can be completed by enumerating all of the possible extensions and choosing the best one. This algorithm based on linear programming ($LP$) can be analyzed in the following way. Consider the schedule that is based in part on the integer assignments indicated by the optimal solution to $LP_1$, but is completed by assigning the remaining jobs to the machine on which they run in a particular optimal schedule. Of course, the schedule found by heuristic is at least as good as this new one. The new schedule can be split into two partial schedules in the obvious way. The partial schedule given by the linear programming solution has $C_{\text{max}}^*$ no more than $C^*_{\text{max}}$, as does the partial schedule for the remaining jobs. Since $C_{\text{max}}(LP)$ is no more than the sum of these two parts, $C_{\text{max}}(LP)\leq 2C^*_{\text{max}}$.

Unfortunately, there can be $m^{m-1}$ ways to assign each of the $m-1$ jobs to one of $m$ machines, and so there is no apparent way to find the optimal extension in polynomial time. However, this takes only constant time if $m$ is fixed, and so we get the following result.

**Theorem 9.11.** The algorithm $LP$ is a polynomial-time algorithm for $Rm || C_{\text{max}}$, and for any instance, $C_{\text{max}}(LP)/C^*_{\text{max}} \leq 2$.

Once again, there exist examples that prove that this analysis is tight. Later in this section, we will show that a fully polynomial approximation scheme can be derived for $Rm || C_{\text{max}}$.

This linear programming approach can be extended to yield a polynomial-time algorithm when $m$ is an input. Once again, we will switch to the perspective of finding a $\rho$-relaxed decision procedure. In this case, the binary search procedure must be modified to initialize $S$, $UB$ and $LB$ as follows:

\begin{align*}
S &:= \text{the greedy schedule;} \\
UB &:= C_{\text{max}}(G); \\
LB &:= \sum_j \min_i p_{ij}.
\end{align*}

Since $UB$ and $LB$ are only within a factor of $m$ initially, we might need to perform an additional $\log m$ iterations until the binary search terminates, but an analogue of Lemma 9.1 remains true.
We will show that the linear programming approach leads to a polynomial-time 2-relaxed decision procedure, which then yields a polynomial-time 2-approximation algorithm. Suppose that we wish to test if $C_{\text{max}}^* \leq d$. We can view this as testing the feasibility of the following system of constraints:

\begin{align}
\sum_{i} p_{ij} x_{ij} &\leq d, \quad \text{for } i=1, \ldots, m, \quad (9.5) \\
\sum_{j} x_{ij} &\leq 1, \quad \text{for } j=1, \ldots, n, \quad (9.6) \\
x_{ij} &\in \{0,1\}, \quad \text{for } i=1, \ldots, m, j=1, \ldots, n. \quad (9.7)
\end{align}

We will, as before, consider the linear relaxation of this, but in order to obtain a tighter relaxation, we add the constraints

\begin{equation}
x_{ij} = 0, \quad \text{if } p_{ij} > d. \quad (9.8)
\end{equation}

Let $LP_2$ denote the linear relaxation of the system of constraints (9.5)-(9.8). We will show how to round any extreme point of $LP_2$ into an integer solution that corresponds to a schedule with $C_{\text{max}} < 2d$. This rounding procedure can be done in polynomial time, and so we get the following polynomial-time 2-relaxed decision procedure: test if $LP_2$ is feasible; if not, output 'no'; otherwise, find an extreme point of $LP_2$ and round it to obtain the desired integer solution.

Suppose that $LP_2$ is feasible and we want to round the extreme point $\bar{x}$. One way to model the essential structure of this solution is to form the following bipartite graph: $G(\bar{x}) = (M \cup J, E)$, where $M = \{M_1, \ldots, M_m\}$ and $J = \{J_1, \ldots, J_n\}$ are the sets of machines and jobs, respectively, and $E = \{(M_i, J_j) | \bar{x}_{ij} > 0\}$. As in the proof of Theorem 9.11, by observing that $LP_2$ has $mn$ variables and only $m+n$ constraints not of the form $x_{ij} \geq 0$, we see that $G(\bar{x})$ has no more edges than nodes. We now show that each connected component of $G(\bar{x})$ has this property.

Let $C$ be a connected component of $G(\bar{x})$. If $M_C$ and $J_C$ are the sets of machine and job nodes contained in $C$, let $\bar{x}_C$ denote the restriction of $\bar{x}$ to those $\bar{x}_{ij}$ for which $i \in M_C$ and $j \in J_C$, and let $\bar{x}_C$ denote the remaining components of $\bar{x}$. For simplicity of notation, reorder the components so that $\bar{x} = (\bar{x}_C, \bar{x}_C')$. The connected component $C$ induces a smaller scheduling problem which restricts attention to the subset of machines $M_C$ and the subset of jobs $J_C$. We can formulate an analogous linear program to $LP_2$ for this subproblem, and denote it by $LP_C$.

We first prove that $\bar{x}_C$ is an extreme point of $LP_C$. Suppose not; then there exist distinct $y_1$ and $y_2$ such that $\bar{x}_C = (y_1 + y_2)/2$, where each $y_i$ is a feasible solution of $LP_C$. But now, $\bar{x} = ((y_1, \bar{x}_C') + (y_2, \bar{x}_C))/2$ where each $(y_i, \bar{x}_C)$ is a feasible solution of $LP_2$, which contradicts the fact that $\bar{x}$ was chosen to be an extreme point of $LP_2$. Since $\bar{x}_C$ is an extreme point of $LP_C$, it follows that $G(\bar{x}_C) = C$ has no more edges than nodes.

Since each component of $G(\bar{x}_C)$ has no more edges than nodes, and is, by definition, connected, each component must either be a tree or a tree plus one additional edge. We now use this fact to round the corresponding extreme point $\bar{x}$. As before, for each edge $(M_i, J_j)$ with $\bar{x}_{ij} = 1$, we assign job $J_j$ to machine $M_i$. These jobs correspond to job nodes of degree 1, so that by deleting all of these nodes we get a graph of the same type, $G'(\bar{x})$, with the additional property that each job node has degree at least 2.
We show that $G'(\hat{x})$ has a matching that covers all of the job nodes. For each component that is a tree, root the tree at any node, and match each job node with any one of its children. (Note that each job node must have at least one child and that, since each node has at most one parent, no machine is matched with more than one job.) For each component that contains a cycle, take alternate edges of the cycle in the matching. (Note that the cycle must be of even length.) If the edges of the cycle are deleted, we get a collection of trees which we think of as rooted at the node that had been contained in the cycle. For each job node that is not already matched, pair it with one of its children. This gives us the desired matching. If $(M_i,J_j)$ is in the matching, assign job $J_j$ to be processed on machine $M_i$.

It is straightforward to verify that the resulting schedule has $C_{\text{max}} \leq 2d$. For each machine $M_i$, at most one job $J_j$ is assigned to it based on the matching in $G'(\hat{x})$. Since the corresponding $\hat{x}_{ij}$ must be greater than 0, $p_{ij} \leq d$. For all of the remaining jobs assigned to $M_i$, the corresponding component of $\hat{x}$ is one, and so by (9.5), the total processing time of these jobs is at most $d$. Therefore, each machine is assigned jobs with total processing time at most $2d$. On the other hand, any schedule with $C_{\text{max}} \leq d$ corresponds to an (integer) feasible solution to $LP_2$, and so if $LP_2$ is infeasible, we are justified in answering ‘no’.

By using this relaxed decision procedure within the usual framework, we have obtained the following result for this approximation algorithm based on linear programming ($LP'$).

**Theorem 9.12.** The algorithm $LP'$ is a polynomial-time algorithm for $R || C_{\text{max}}$ and for any instance, $C_{\text{max}}(LP')/C_{\text{max}} \leq 2$. □

It is not hard to construct a family of instances that show that this analysis cannot be improved to yield a better worst-case bound (see Exercise 9.13).

It is significant to note that the structure of $G(\hat{x})$ that was used in rounding $\hat{x}$ can also be used in finding an extreme point of $LP_2$. From a practical point of view, this characterization leads to a particularly efficient implementation of the simplex method. Alternatively, the structure of this linear program can be used to derive a special-purpose combinatorial algorithm that runs in polynomial time.

Consider again the special case when the number of machines is a fixed integer $m$. We will show that in this case, there is a fully polynomial approximation scheme. Clearly, this implies that such a scheme exists for the cases of a fixed number of identical machines or uniform machines. The heart of the scheme is a pseudopolynomial algorithm to find an optimal solution to $Rm || C_{\text{max}}$. It is not hard to see that dynamic programming can be used to derive an algorithm that runs in $O(nmT^{m-1})$ time, where $T=\sum_j \min_i p_{ij}$ (see Exercise 9.18).

Given this pseudopolynomial-time algorithm, it is rather straightforward to complete the fully polynomial approximation scheme. For each positive integer $k$, we will construct a $(1+\frac{1}{k})$-approximation algorithm. Approximate each processing time $p_{ij}$ by $\lfloor p_{ij}/\delta \rfloor$, where $\delta=\frac{T}{kmn}$. Note that this rescales $T$ to at most $kmn$. Find the optimal solution to this rescaled and rounded problem in $O(nm(kmn)^{m-1})$ time, and consider this schedule for the original processing times. The schedule obtained is also optimal for the instance with processing times given by $\delta\lfloor p_{ij}/\delta \rfloor$, $i=1, \ldots, m$, $j=1, \ldots, n$, and its value of $C_{\text{max}}$ with these processing times is clearly a lower bound on the desired optimum. Since each $C_j$ is the sum of at most $n$ individual processing times, the difference between the value of $C_{\text{max}}$ for this schedule with the
original processing times and the value of $C_{\text{max}}$ for this schedule with the rounded processing times is at most

$$n \delta = nT/(kmn) = \frac{1}{k} \frac{T}{m} \leq \frac{1}{k} C^*_{\text{max}}.$$ 

Therefore, this algorithm, which we shall call $H_k$, has the worst-case guarantee, $C_{\text{max}}(H_k)/C^*_{\text{max}} \leq 1 + \frac{1}{k}$, and we have obtained the following result.

Theorem 9.13. For any fixed $m$, the family of algorithms \{$H_k$\} forms a fully polynomial approximation scheme for $Rm \mid \mid C_{\text{max}}$. □

Exercises

9.12. Give a family of examples that shows that for any number of machines, the analysis given for the greedy algorithm is tight.

9.13. Give a family of examples that shows that the analysis of $LP'$ given in Theorem 9.12 is tight. In addition, show that this bound is tight, even if $m=2$.

9.14. Suppose that when job $J_j$ is assigned to machine $M_i$, a cost of $c_{ij}$ is incurred. Let $c^*$ denote the minimum total cost of any schedule with maximum completion time $C^*_{\text{max}}$. Use the linear programming approach to give an algorithm that delivers a schedule with $C_{\text{max}} \leq 2C^*_{\text{max}}$ with total cost no more than $c^*$.

9.15. Suppose that when job $J_j$ is scheduled on machine $M_i$, it may take anywhere between $l_{ij}$ and $u_{ij}$ time units. If job $J_j$ is scheduled on machine $M_i$ to be processed in $u_{ij}$ time units, a cost of $c_{ij}$ is incurred; job $J_j$ may be processed faster on $M_i$ by incurring an additional cost of $s_{ij}$ per unit decrease. For example, the cost of scheduling $J_j$ on $M_i$ to take $l_{ij}$ units is $c_{ij} + s_{ij}(u_{ij} - l_{ij})$. The objective is to schedule all jobs with minimum total cost, subject to the constraint that all jobs are completed within a deadline $d$. Use the linear programming approach to devise an algorithm that delivers a schedule that costs no more than this optimum, and completes all jobs within $2d$. (Hint: When job $J_j$ is processed for any time in the range between $l_{ij}$ and $u_{ij}$, this can be viewed as processing some fraction of the job on machine $M_i$ at the speed needed to process the entire job in $l_{ij}$ time units, and the complementary fraction is processed at the speed at which it would be processed in $u_{ij}$ time units.)

9.16. Consider the variant of the decision version of $R \mid \mid C_{\text{max}}$ where each machine $M_i$ has its own deadline $d_i$. Modify the approach used in algorithm $LP'$ to give an algorithm that either outputs ‘no’, or else outputs a schedule in which each machine $M_i$ completes its assigned jobs by $d_i + \max_j P_{ij}$, where ‘no’ is output only when no schedule completes all jobs within the given deadlines.

9.17. The linear programming approach can also be used to design a polynomial approximation scheme for $Rm \mid \mid C_{\text{max}}$, where the space required does not depend exponentially on $m$. Recall that in designing a $(1+\frac{1}{k})$-approximation algorithm it is sufficient to find a $(1+\frac{1}{k})$-relaxed deci-
sion procedure. To use this approach, one can define a suitable notion of a bad assignment, where the job takes a constant fraction of the time prior to the deadline if assigned to that machine. As a result, there are only a constant number of ways to make bad assignments of the jobs. Combine complete enumeration of these possibilities with the algorithm of Exercise 9.15, to design a \((1+\frac{1}{k})\)-relaxed decision procedure.

9.18. Construct a dynamic programming algorithm to solve \(Rm \mid \mid C_{\text{max}}\) in \(O(nmT^{m-1})\) time, where \(T = \sum_{j} \min_{i} p_{ij}\). (Hint: recall Theorem 8.??.)

9.5. Unrelated machines: impossibilities

Unlike \(P \mid \mid C_{\text{max}}\) and \(Q \mid \mid C_{\text{max}}\), no polynomial approximation scheme is known for \(R \mid \mid C_{\text{max}}\). We shall see that it is highly unlikely that such a scheme exists, since this would imply that \(P=NP\). In order to prove such a result, we focus on the computational complexity of the decision version of \(R \mid \mid C_{\text{max}}\) with small integral deadlines.

Theorem 9.14. For \(R \mid \mid C_{\text{max}}\), the question of deciding if \(C_{\text{max}}^* \leq 3\) is \(NP\)-complete.

Proof. We prove this result by a reduction from the 3-dimensional matching problem. We are given an instance of this problem, consisting of a family of triples \(\{T_1, T_2, \ldots, T_m\}\) over the ground set \(A \cup B \cup C\), where \(A\), \(B\) and \(C\) are disjoint sets such that \(|A| = |B| = |C| = n\); each \(T_i\) satisfies \(|T_i \cap A| = |T_i \cap B| = |T_i \cap C| = 1\). We construct an instance of the scheduling problem with \(m\) machines and \(2n + m\) jobs. Machine \(M_i\) corresponds to the triple \(T_i\), for \(i = 1, \ldots, m\). There are \(3n\) 'element jobs' that correspond to the \(3n\) elements of \(A \cup B \cup C\) in the natural way. In addition, there are \(m-n\) 'dummy jobs'. (If \(m < n\), we construct some trivial 'no' instance of the scheduling problem.) Machine \(M_i\) corresponding to \(T_i = (a_j, b_k, c_l)\) can process each of the jobs corresponding to \(a_j\), \(b_k\) and \(c_l\) in one time unit and each other job in three time units. Note that the dummy jobs require three time units on each machine.

It is quite simple to show that \(C_{\text{max}}^* \leq 3\) if and only if there is a 3-dimensional matching. Suppose there is a matching. For each \(T_i = (a_j, b_k, c_l)\) in the matching, schedule the element jobs corresponding to \(a_j\), \(b_k\) and \(c_l\) on machine \(M_i\). Schedule the dummy jobs on the \(m-n\) machines corresponding to the triples that are not in the matching. This gives a schedule with \(C_{\text{max}}^* = 3\). Conversely, suppose that there is such a schedule. Each of the dummy jobs requires three time units on any machine and is thus scheduled by itself on some machine. Consider the set of \(n\) machines that are not processing dummy jobs. Since these are processing all of the \(3n\) element jobs, each of these jobs is processed in one time unit. Each three jobs that are assigned to one machine must therefore correspond to elements that form the triple corresponding to that machine. Since each element job is scheduled exactly once, the \(n\) triples corresponding to the machines that are not processing dummy jobs form a matching. \(\square\)

As an immediate corollary of this theorem, we get the following result.

Corollary 9.2. For every \(\rho < 4/3\), there does not exist a polynomial-time \(\rho\)-approximation algorithm for \(R \mid \mid C_{\text{max}}\) unless \(P=NP\).

Proof. Suppose there were such an algorithm. We will show that it yields a polynomial-time
algorithm for the 3-dimensional matching problem. Given an instance \( I \) of the 3-dimensional matching problem, map it into an instance of \( R || C_{\text{max}} \) using the reduction given above, and then apply the presumed approximation algorithm. We have just seen that \( I \) is a ‘yes’ instance if and only if the instance of \( R || C_{\text{max}} \) does have a schedule of length 3. Therefore, when \( I \) is a ‘yes’ instance, the approximation algorithm must output a schedule of length less than \((4/3)C^*_{\text{max}}\). But this length must be an integer, and so it must be at most 3. When \( I \) is a ‘no’ instance, the algorithm produces a schedule of length at least 4. Therefore, the algorithm outputs a schedule of length at most 3 if and only if \( I \) is a ‘yes’ instance. \( \square \)

The technique employed in Theorem 9.14 can be refined to yield a stronger result.

**Theorem 9.15.** For \( R || C_{\text{max}} \), the question of deciding if \( C^*_{\text{max}} \leq 2 \) at most 2 is \( NP \)-complete.

**Proof.** We again start from the 3-dimensional matching problem. We call the triples that contain \( a_j \) triples of type \( j \). Let \( t_j \) be the number of triples of type \( j \), for \( j = 1, \ldots, n \). As before, machine \( M_i \) corresponds to the triple \( T_i \), for \( i = 1, \ldots, m \). There are now only \( 2n \) element jobs, corresponding to the \( 2n \) elements of \( B \cup C \). We refine the construction of the dummy jobs: there are \( t_j - 1 \) dummy jobs of type \( j \), for \( j = 1, \ldots, n \). (Note that the total number of dummy jobs is \( m - n \), as before.) Machine \( M_i \) corresponding to a triple of type \( j \), say, \( T_i = (a_j, b_k, c_l) \), can process each of the element jobs corresponding to \( b_k \) and \( c_l \) in one time unit and each of the dummy jobs of type \( j \) in two time units; all other jobs require three time units on machine \( M_i \).

Suppose there is a matching. For each \( T_i = (a_j, b_k, c_l) \) in the matching, schedule the element jobs corresponding to \( b_k \) and \( c_l \) on machine \( M_i \). For each \( j \), this leaves \( t_j - 1 \) idle machines corresponding to triples of type \( j \) that are not in the matching; schedule the \( t_j - 1 \) dummy jobs of type \( j \) on these machines. This completes a schedule with \( C_{\text{max}} = 2 \). Conversely, suppose that there is such a schedule. Each dummy job of type \( j \) is scheduled on a machine corresponding to a triple of type \( j \). Therefore, there is exactly one machine corresponding to a triple of type \( j \) that is not processing dummy jobs, for \( j = 1, \ldots, n \). Each such machine is processing two element jobs in one time unit each. If the machine corresponds to a triple of type \( j \) and its two unit-time jobs correspond to \( b_k \) and \( c_l \), then \( (a_j, b_k, c_l) \) must be the triple corresponding to that machine. Since each element job is scheduled exactly once, the \( n \) triples corresponding to the machines that are not processing dummy jobs form a matching. \( \square \)

**Corollary 9.3.** For every \( p < 3/2 \), there does not exist a polynomial-time \( p \)-approximation algorithm for \( R || C_{\text{max}} \), unless \( P = NP \).

**Exercises**

9.19. Prove for any integers \( p < q \) such that \( 2p \neq q \), \( R || C_{\text{max}} \) is \( NP \)-hard even in the case that all \( p_{ij} \in \{ p, q \} \).

9.20. In contrast to Exercise 9.19, give a polynomial-time algorithm to solve the special case of \( R || C_{\text{max}} \) when each \( p_{ij} \in \{ 1, 2 \} \).

9.6. Probabilistic analysis of algorithms
The results presented in the previous sections provide ample illustration of the power of a worst-case approach to the analysis of heuristics. While necessarily pessimistic, the outcome of a worst-case analysis at least yields an ironclad performance guarantee that will always be valid. This safety belt often comes at the expense of realism, in that computational experiments might indicate that the worst-case behavior is rarely registered in practice. On the contrary, the average performance of an approximation algorithm is usually strikingly better than its worst-case behavior would suggest. In this section, we shall consider a mathematical framework in which these empirical results can be analyzed, and reconsider several of the approximation algorithms already discussed, but from this perspective. As a caveat, however, the reader should note that we might have little understanding of what an average instance really is in practice; the results presented in this section might be no more realistic than our assumptions about the instances considered.

We first must outline the precise mathematical definitions that will capture the notion of the average performance of a heuristic. Probability theory provides an appropriate setting for this approach. We shall assume that the reader is familiar with the basic terminology of this area. In the spirit of empirical computational work, a problem instance will be regarded as being generated by a random mechanism. For example, for the scheduling problem \( P\parallel C_{\text{max}} \), one would typically assume that the processing times \( p_j, j=1, \ldots, n \), are random variables whose joint distribution is given in advance. Given a particular realization of these random variables, the heuristic solution is computed; its value is obviously a random variable as well, whose distribution can be analyzed and whose expected value informs us about the average behavior of the heuristic in question, especially when compared to the expected value of the optimal solution.

The probabilistic analysis of algorithms, then, starts from a probability distribution over the class of all problem instances, and focuses on the random variables describing algorithmic behavior on a randomly generated instance. The analysis can be technically demanding; frequently, it is asymptotic in nature, in that precise statements are only possible if the problem size is allowed to go to infinity. Hence, it is appropriate at this point to introduce the three modes of stochastic convergence that typically arise in such a situation.

If \( y_1, y_2, \ldots \) is a sequence of random variables, then almost sure (a.s.) convergence of the sequence to a constant \( c \) means that

\[
Pr \{ \lim_{n \to \infty} y_n = c \} = 1.
\]

Sometimes this is referred to as convergence with probability 1. It implies the weaker condition convergence in probability, which requires that, for every \( \varepsilon > 0 \),

\[
\lim_{n \to \infty} Pr \{ |y_n - c| > \varepsilon \} = 0.
\]  \hspace{1cm} (9.9)

Finally, convergence in expectation means that

\[
\lim_{n \to \infty} |\text{Ex}[y_n] - c| = 0,
\]

and also implies (9.9).

We shall illustrate some of these notions on the problem \( P2\parallel C_{\text{max}} \). In this case, a problem instance corresponds to the processing times \( p_1, p_2, \ldots, p_n \); let us assume these to be independent random variables that are uniformly distributed over the interval \([0,1]\), which is a model favored by analysts and experimenters alike.
Let us first consider the performance of the list scheduling (LS) algorithm from a probabilistic perspective. From (9.1), we deduce that in the case of $P2||C_{\text{max}}$,

$$C_{\text{max}}(LS) \leq \sum_j p_j/2 + p_{\text{max}}/2.$$ 

Since the first term on the right-hand side is a lower bound on $C^*_{\text{max}}$, we see that

$$C_{\text{max}}(LS)/C^*_{\text{max}} \leq 1 + p_{\text{max}}/\sum_j p_j.$$  \hspace{1cm} (9.10)

Obviously, $p_{\text{max}} \leq 1$. In addition, the strong law of large numbers says that

$$(\sum_j p_j)/(n/2) \rightarrow 1 \text{ (a.s.)}.$$ 

Hence, the ratio $C_{\text{max}}(LS)/C^*_{\text{max}}$ itself also converges to 1 with probability 1. Put differently, the relative error of this simple heuristic almost surely vanishes for large problem sizes, and its depressing worst-case error of 50 percent occurs rather infrequently.

As we have seen above, almost sure convergence is just one of several ways to capture the asymptotic behavior of a sequence of random variables. Thus, there are other ways to capture the asymptotic optimality of these simple heuristics. They do, however, require a different probabilistic starting point.

By way of example, consider the inequality

$$Pr\left\{ \frac{\sum_j p_j}{n} - \frac{1}{2} \geq t \right\} \leq e^{-2nt^2},$$

which is a special case of Hoeffding's inequality. From (9.10), with $t = 1/4$, one can easily deduce that

$$Pr\left\{ \frac{C_{\text{max}}(LS) - C^*_{\text{max}}}{C^*_{\text{max}}} \leq \frac{4(m-1)}{3n} \right\} \geq 1 - e^{-n/8},$$

which, of course, confirms error convergence to 0 in probability, but which also provides precise information on the rate at which the relative error of list scheduling converges to 0. We shall later return to the third mode of convergence, convergence in expectation.

The entire analysis above has its implications for the optimal solution value $C^*_{\text{max}}$ itself. Indeed, a trivial byproduct is that

$$C^*_{\text{max}}/(n/4) \rightarrow 1 \text{ (a.s.)}.$$ 

This is a typical example of what is usually referred to as probabilistic value analysis; for large $n$, the optimal solution value can be guessed with increasing relative accuracy. The structure of the optimal solution itself does not lend itself to a similar probabilistic convergence analysis, and so it is much easier to predict the solution value that to say anything about the solution itself.

Now that asymptotic optimality in the relative sense turns out to be within easy reach, it is tempting to examine the asymptotic behavior of the absolute error for $C_{\text{max}}(A) - C^*_{\text{max}}$ for several algorithms $A$. In the case of list scheduling, it converges to a value strictly greater than 0. Can we do better? The insight gained from the worst-case analysis leads us to consider the LPT rule, with the hope that this might demonstrate superior probabilistic performance too. Let us investigate this possibility from a probabilistic perspective by focusing on the absolute difference $d_j(LPT)$ between the total processing time assigned to $M_1$ and to $M_2$ after $j$ jobs have been allocated. Clearly, $d_j(LPT)/2$ is an upper bound on the absolute error.
Let \( p(1) \leq p(2) \leq \ldots \leq p(n) \) denote the sorted list of processing times. The structure of the LPT rule immediately implies that
\[
d_n(LPT) \leq \max \{ d_{n-1}(LPT) - p(1), p(1) \}
\leq \max \{ d_{n-2}(LPT) - (p(1) + p(2)) - p(1), p(1) \}.
\]
Continuing recursively, we see that
\[
d_n(LPT) \leq \max_{k=1, \ldots, n} \left\{ p(k) - \sum_{j=1}^{k-1} p(j) \right\}.
\]
It is trivial to see that, for any fixed \( \varepsilon \in (0, 1) \),
\[
\max_{k=1, \ldots, \lfloor \varepsilon n \rfloor} \left\{ p(k) - \sum_{j=1}^{k-1} p(j) \right\} \leq p \left( \frac{\varepsilon n}{n+1} \right) \tag{9.11}
\]
and
\[
\max_{k=\lfloor \varepsilon n \rfloor + 1, \ldots, n} \left\{ p(k) - \sum_{j=1}^{k-1} p(j) \right\} \leq p \left( \frac{n}{n+1} \right) \sum_{j=1}^{\lfloor \varepsilon n \rfloor} p(j). \tag{9.12}
\]
For the uniform distribution, it is known that \( p \left( \frac{\varepsilon n}{n+1} \right) \varepsilon \to 1 \) (a.s.). In addition, it is not difficult to show that \( \sum_{j=1}^{\lfloor \varepsilon n \rfloor} p(j)/n \) almost surely converges to a positive constant (which depends on \( \varepsilon \)).

Since \( p(n) \) is obviously at most 1, this implies that the right-hand side of (9.12) tends to \(-\infty\), and hence the maximum of the right-hand sides of (9.11) and (9.12) tends to \( \varepsilon \). This establishes the following theorem.

**Theorem 9.16.** For \( P2||C_{\text{max}} \), \( d_n(LPT) \to 0 \) almost surely.

**Corollary 9.4.** For \( P2||C_{\text{max}} \), \( C_{\text{max}}(LPT) - C^*_{\text{max}} \to 0 \) almost surely.

We can use the same bounding technique to bound \( d_n(LPT) \) in expectation. From probability theory, we know that
\[
\mathbb{E}[p \left( \frac{\varepsilon n}{n+1} \right)] = \frac{\varepsilon n}{n+1}.
\]
and
\[
\mathbb{E} \left[ p(n) - \sum_{j=1}^{\lfloor \varepsilon n \rfloor} p(j) \right] = n - \sum_{j=1}^{\lfloor \varepsilon n \rfloor} j \left( \frac{n}{n+1} \right). \tag{9.13}
\]
For every fixed \( \varepsilon \in (0, 1) \), the right-hand side of (9.13) converges to \(-\infty\), and hence the absolute error of the LPT rule also converges to 0 in expectation.

**Theorem 9.17.** For \( P2||C_{\text{max}} \), \( \mathbb{E}[C_{\text{max}}(LPT) - C^*_{\text{max}}] \to 0 \).

This result can be extended for much more general models, such as \( Q||C_{\text{max}} \) (see Exercise 9.21).

Can we be still more ambitious? The above analysis for \( P2||C_{\text{max}} \) reveals that \( d_n(LPT) \) converges to 0 (a.s.), but does not provide any information on the rate at which this occurs. It
is possible to estimate this rate; one finds that $d_n(LPT)/(\log \log n/n)$ is almost surely bounded by a constant. At the same time, the smallest possible difference $d_n^*$ is known to converge much faster to 0; $d_n^*/(n^{2-\pi})$ is almost surely bounded by a constant, so that here convergence occurs at an exponential rate. From that perspective, the best possible heuristic would be one that achieves a similarly fast convergence. Does such a heuristic exist?

The answer to this question is unknown, but an impressive improvement on the rate of convergence of the LPT rule is obtained by a modified version of the differencing method. The differencing method was introduced in studying the probabilistic performance of algorithms for this problem, but there is no known theoretical analysis of this method. As is often the case in the probabilistic analysis of algorithms, it seems to be easier to modify this algorithm to obtain one that is similar, and yet more amenable to analysis. The typical problem is one of dependencies among the data used at various stages of the algorithm. Initially, we have assumed that the processing requirements of the jobs are independent random variables, and this assumption is extremely important to the analyses that we have given. However, after even one iteration of the differencing method, we have lost this essential property. The basic idea is to modify the algorithm so that iterations of the algorithm can be analyzed as essentially independent steps. The technical details of this result are far too complicated to present here; however, we shall give an outline of the modified algorithm and try motivate the modifications made.

The modified differencing method (MD) works in a series of phases, where in each phase most of the jobs are paired, and each pair is replaced by a new job with processing requirement equal to the difference of their processing requirements. Suppose that in the current phase, each $p_j \in (0,U]$; we can subdivide this interval into a number of equal-length subintervals. For each pair, both jobs will be selected from the same interval, and so the size of the new job created will be bounded by the length of the subinterval. The crucial modification is to introduce randomization into the algorithm itself. For each subinterval, the algorithm randomly pairs all jobs (assuming that there are an even number of jobs). As a result, the processing requirements of the jobs created in this way will be independent random variables, and will be distributed according to a triangular distribution. Of course, not all subintervals will have an even number of jobs, and something more complicated must be introduced to handle this situation. The algorithm terminates when there are only a specified constant number of jobs remaining, and this small instance can be solved to optimality.

The crux of the analysis is to calculate the way in which the length of the upper bound $U$ changes from iteration to iteration, as compared to the number of jobs that remain. The value of $U$ at any stage is an upper bound on the value of $d_n(MD)$; thus, if we can show that with high probability, the value of $U$ at the end is very small, then we obtain the corresponding bound for the solution given by algorithm. Roughly speaking, it can be shown that the number of jobs in the $i+1$th phase is at least $c^{-1}n$ and the upper bound $U$ is at most $c^{-1}n^{-1}$, where $c$ is a constant greater than 2. When the number of remaining jobs reaches the specified constant, the bound on the length of the interval is $n^{-O(\log n)}$.

Theorem 9.18. For $P_{2\|C_{max}}$, $C_{max}(MD)-C_{max}^* \leq d_n(MD)/2=n^{-O(\log n)}$.

In the probabilistic analysis of algorithms, it is often the case that the modified algorithm is justified by showing that, in some stochastic sense, the performance of the original algorithm dominates the modified one. It would be nice if this were true in this case, but this remains an
important open question.

We have dealt with $P2\|C_{\text{max}}$ at length, since this example exhibits many of the ingredients typically encountered in a probabilistic analysis:

- a combinatorial problem which is NP-hard and hence difficult to solve;
- a probability distribution over all problem instances to generate problem data as realizations of independent and identically distributed random variables;
- a probabilistic value analysis that yields an asymptotic characterization of the optimal solution value as a simple function of the problem data;
- a probabilistic error analysis of a fast heuristic to prove that its relative or absolute error tends to 0 with increasing problem size in some stochastic sense; and
- a rate of convergence analysis that yields some indication of how large the problem size must be in order to demonstrate asymptotic behavior in practice, and this, moreover, allows for further differentiation among the heuristics.

Bibliographic Notes

9.1. The seminal paper of Graham [1966] initiated the area of performance guarantees for approximation algorithms by considering the list scheduling rule. In a later paper, Graham [1969] also provided the analysis of the LPT rule, as well as the polynomial approximation schemes for $Pm \mid \mid C_{\text{max}}$ given in Theorem 9.3 and Exercise 9.2. The first fully polynomial approximation scheme for $Pm \mid C_{\text{max}}$ is due to Sahni [1976]. Karmarkar & Karp [1982] invented the differencing method, and its worst-case analysis is due to Fischetti & Martello [1987].

9.2. The multifit algorithm is due to Coffman, Garey & Johnson [1978], who proved that $C_{\text{max}}(MF)/C_{\text{max}}^* \leq 1.22 + 2^{-1}$; the improved bound given in Theorem 9.4 is due to Friesen [1984]. Although the bound of this theorem is not known to be tight, there are examples that achieve a ratio of 13/11. If the FFD bin-packing algorithm is replaced by the weaker first-fit algorithm (FF), where the jobs are not ordered by decreasing $p_j$, then all that can be guaranteed is

$$C_{\text{max}}(MF)/C_{\text{max}}^* \leq 2 - \frac{2}{m + 1}. \quad (\dagger)$$

Friesen & Langston [1986] refine the iterated approximation algorithm to provide algorithms $MF_k'$ with running time $O(n \log n + kn \log m)$ (where the constant embedded within the 'big Oh' notation is big indeed) that guarantee

$$C_{\text{max}}(MF_k')/C_{\text{max}}^* \leq \frac{72}{61} + 2^{-k}. \quad (\dagger)$$

The framework of using a relaxed decision procedure, as well as the first polynomial approximation scheme for $P \mid C_{\text{max}}$ is due to Hochbaum & Shmoys [1987]. Exercises 9.5 and 9.6 are also derived from Hochbaum & Shmoys [1987]. Hochbaum & Shmoys [1987] also give a (7/6)-relaxed decision procedure that runs in $O(n (m^4 + \log n))$ time. Exercises 9.7(a) and 9.7(b) are based on the work of Kafura & Shen [1977, 1978].

Several bounds are available which take into account the processing times of the jobs. Recall that the probabilistic analysis discussed in Section 9.0 relies on such a (worst-case) bound for list scheduling. Achugbue & Chin [1981] prove two results relating the performance
ratio of list scheduling to the value of π = max \( p_j / \min_j p_j \). If π ≤ 3, then

\[
C_{\text{max}}(LS)/C^*_{\text{max}} \leq \begin{cases} 
5/3 & \text{if } m = 3, 4, \\
17/10 & \text{if } m = 5, \\
2 - \frac{1}{3\lfloor m/3 \rfloor} & \text{if } m \geq 6,
\end{cases} \tag{†}
\]

and if π ≤ 2,

\[
C_{\text{max}}(LS)/C^*_{\text{max}} \leq \begin{cases} 
3/2 & \text{if } m = 2, 3, \\
5/3 - \frac{1}{3\lfloor m/2 \rfloor} & \text{if } m \geq 4.
\end{cases} \tag{†}
\]

For the case of LPT, Ibarra & Kim [1977] prove that

\[
C_{\text{max}}(LPT)/C^*_{\text{max}} \leq 1 + \frac{2(m-1)}{n} \quad \text{for } n \geq 2(m-1)\pi.
\]

Significantly less is known about the worst-case performance of approximation algorithms for other minmax criteria. For \( P | r_j | L_{\text{max}} \), Exercise 9.8(a) is due to Gusfield [1984], and 9.8(b) was observed by Hall & Shmoys [1989]. Hall & Shmoys [1989] also developed a polynomial approximation scheme for this problem. Extending Theorem 4.7, Simons [1983] showed that \( P | r_j, p_j = p | L_{\text{max}} \) can be solved in polynomial time. Simons & Warmuth [1989] gave an improved \( O(mn^2) \) algorithm based on a generalization of the approach of Garey, Johnson, Simons & Tarjan [1981].

9.3. Theorem 9.6 is due to Liu & Liu [1974], as are the observations given as Exercises 9.9 and 9.10. The variant of list scheduling \( (LS') \) in which the next job in this list is scheduled on the machine on which it will finish earliest is due to Cho & Sahni [1980]. They proved that

\[
C_{\text{max}}(LS')/C^*_{\text{max}} \leq \begin{cases} 
(1+\sqrt{5})/2 & \text{for } m = 2, \\
(1+\sqrt{2m-2})/2 & \text{for } m > 2.
\end{cases}
\]

The bound is tight for \( m \leq 6 \), but in general, the worst known examples have a performance ratio of \( \lceil (\log_2(3m-1)+1)/2 \rceil \). This approach followed the work of Gonzalez, Ibarra & Sahni [1977], who presented the LPT' algorithm and the analysis given in Theorem 9.7. The improved upper and lower bounds for LPT' have been obtained by Dobson [1984] and Friesen [1987]. Morrison [1988] showed that LPT is better than LS', in that

\[
C_{\text{max}}(LPT)/C^*_{\text{max}} \leq \max_i (s_i/(2\min_i s_i), 2). \tag{†}
\]

Friesen & Langston [1983] extended the multitfit approach to uniform processors. They proved that, if the bins are ordered in increasing size for each iteration of the binary search, then

\[
C_{\text{max}}(MF_k)/C^*_{\text{max}} \leq 1.4 + 2^{-k},
\]

and that there exists an example that has performance ratio 1.341. They also show that the decision to order the bins by increasing size is the correct one, since for decreasing bin sizes there exist examples with performance ratio 3/2.
Horowitz & Sahni [1976] gave a family of algorithms $A_k$ with running time $O(n^{2m} k^{m-1})$ such that

$$C_{\text{max}}(A_k)/C_{\text{max}}^* \leq 1 + \frac{1}{k},$$

so that for any fixed value of $m$, this is a fully polynomial approximation scheme. The algorithm $\text{Recurse}$ and the polynomial approximation scheme for $O \mid \mid C_{\text{max}}$ are due to Hochbaum & Shmoys [1988]. The algorithm and analysis given in Exercise 9.11 has been observed by Williamson (private communication).

9.4. The first work to consider approximation algorithms for unrelated machines is due to Ibarra & Kim [1977], who consider the greedy algorithm presented in Section 9.4, and other $m$-approximation algorithms. Davis & Jaffe [1981] analyze a number of algorithms, and obtain the bound for the $LE$ algorithm given in Theorem 9.10. The linear programming approach of Theorem 9.11 for $R \mid \mid C_{\text{max}}$ is due to Potts [1985]. The modified algorithm algorithm $LP'$ and its analysis are due to Lenstra, Shmoys & Tardos [1990]. The fully polynomial approximation scheme for $R_m \mid \mid C_{\text{max}}$ is due to Horowitz & Sahni [1976]. The model discussed in Exercise 9.15 is due to Trick [1990], and Tardos (private communication) observed the results given in Exercises 9.14 and 9.15. The polynomial approximation scheme derived in Exercises 9.16 and 9.17 is due to Lenstra, Shmoys & Tardos [1990].

9.5. The results in this section are due to Lenstra, Shmoys & Tardos [1990].