Scheduling Algorithms

David Karger, Massachusetts Institute of Technology
Cliff Stein, Dartmouth College
Joel Wein, Polytechnic University

1 Introduction

Scheduling theory is concerned with the *optimal allocation of scarce resources to activities over time.* The practice of this field dates to the first time two humans contended for a shared resource and developed a plan to share it without bloodshed. The theory of the design of algorithms for scheduling is younger, but still has a significant history – the earliest papers in the field were published more than forty years ago.

Researchers have studied literally thousands of variants of machine scheduling, and it would be impossible even just to enumerate all known variants in the space of this chapter. Our goal is more modest. We wish to make the reader familiar with an assortment of algorithmic techniques that have proved useful for a variety of scheduling problems. We will demonstrate these techniques on problems drawn from a collection of "basic problems," that model important issues arising in many scheduling environments, while at the same time remaining simple enough so as to permit elegant and useful analysis. The centrality of these models was reinforced by two influential surveys [GLLK79, LLKS93], and they have received much attention.

Algorithms for scheduling can be deployed in a variety of settings, as is illustrated by the following examples.

Example 1: Consider the central processing unit of a computer that must process a sequence of jobs that arrives over time; in what order should the jobs be processed in order to minimize, on average, the time that a job is in the system from arrival to completion?

Example 2: Consider a team of five astronauts preparing for the reentry of their space shuttle to the atmosphere. There is a set of tasks that must be accomplished by the team before reentry. Each task must be carried out by exactly one astronaut, and certain tasks can not be started until other tasks are completed. Which tasks should be performed by which astronaut, and in which order, to insure that the entire set of tasks is accomplished as quickly as possible?

Example 3: Consider a factory that produces different sorts of widgets. Each widget must first be processed by machine 1, then machine 2, and then machine 3, but different widgets require different amounts of processing time on different machines. The factory has orders for batches of widgets; each order has a date by which it must be completed. In what order should the machines work on different widgets in order to insure that the factory completes as many orders as possible on time?

Each of these are examples of scheduling problems that can be modeled directly by one of the basic problems. Even when we limit our attention to these basic problems, there are many variants, and therefore we will begin our study of algorithms for scheduling by laying out a classification of the different problems, and a notation for describing them.

A scheduling model is defined by three separate elements: the *machine environment*, the *optimality criterion*, and a set of *side constraints and characteristics*. We first discuss the simplest machine environment, and use that to introduce a variety of optimality criteria and side constraints. We then introduce and discuss more complex machine environments.

1.1 The One-Machine Environment

In all of our scheduling models we begin with a set $\mathcal{J}$ of n jobs, numbered $1, \ldots, n$. In the *one-machine* environment we have one machine that can process at most one job at a time. Each job j has a processing requirement p_j; namely, it requires processing for a total of p_j units of time on the machine. If each job must be processed in an uninterrupted fashion, we have a *nonpreemptive* scheduling model, whereas if a job may be processed for a period of time, interrupted and continued at a later point in time, we have a *preemptive* model. A schedule S for the set $\mathcal{J}$ specifies, for each job j, which p_j units of time the machine uses to process job j. Given a schedule S, we denote the *completion time* of job j in schedule S by C_j^S.

The goal of a scheduling algorithm is to produce a "good" schedule, but the definition of "good" will vary depending on the application. In Example 2 above, the goal is to process the entire batch of jobs as quickly as possible, or, in other words, to minimize the completion time of the last job finished in the schedule. In Example 1 we care less about the completion time of the last job in the batch as long as, on average, the jobs receive good service. Therefore, given a set of jobs and a machine environment, we must specify an *optimality criterion*; the goal of a scheduling algorithm will be to construct a schedule that optimizes this criterion. The two optimality criteria discussed above are among the most basic optimality criteria: the *average completion time* of a schedule and its *makespan*. We define the makespan $C^S_{\max} = \max_j C^S_j$ of a schedule to be the maximum completion time of any job in the schedule, and the average completion time of a schedule S to be $\frac{1}{n}\sum_{j=1}^n C^S_j$. Note that optimizing the average completion time is equivalent to optimizing the *sum* of completion times $\sum_{j=1}^n C^S_j$.

We next turn to *side constraints* that modify the one machine environment. A number of side constraints and characteristics are possible; for example, we must specify whether or not preemption is allowed. Two other possible constraints model the arrival of jobs over time or the possibility of logical dependence between jobs. In a scheduling environment with *release date* constraints, we associate with each job j a release date r_j; job j is only available for processing at time r_j or later. In a scheduling environment with *precedence constraints* we are given a partial order $\prec$ on the set $\mathcal{J}$ of jobs; if $i \prec j$ then we may not begin processing job j until job i is completed.

Although we are early in our discussion of scheduling models, we already have enough information to define a number of problems. We refer to various scheduling problems in the now-standard notation defined by Graham, Lawler, Lenstra, & Rinnooy Kan (1979) [GLLK79]. A problem is denoted by $\alpha|\beta|\gamma$, where (i) α denotes the machine environment, (ii) β denotes various side constraints and characteristics and (iii) γ denotes an optimality criterion.

For the one-machine environment α is 1. For the optimality criteria we have introduced so far, γ is either $\sum C_j$ or $C_{\max}$. At this point in our discussion, β is a subset of r_j, *prec*, and *pmtn*, where these denote respectively the presence of (non-trivial) release date constraints, precedence constraints and the ability to schedule preemptively. As an illustration, $1||\sum C_j$ denotes the problem of nonpreemptively scheduling independent jobs on one machine so as to minimize their average completion time, while $1|r_j|\sum C_j$ denotes the variant of the problem in which jobs have release dates. As another example, $1|r_j, pmtn, prec|C_{\max}$ denotes the problem of preemptively scheduling jobs with release dates and precedence constraints on one machine so as to minimize their makespan. Note that Example 1, given above, can be modeled by $1|r_j|\sum C_j$, or, if preemption is allowed, by $1|r_j, pmtn|\sum C_j$.

Two other possible elements of a scheduling application might be relevant in the one-machine environment. It is possible that not all jobs are of equal importance, and thus, when measuring average service provided to a job, one might wish to weight the average so as to give more importance to certain jobs. We model this by assigning a weight $w_j > 0$ to each job j, and generalize the $\sum C_j$ criterion to the *average weighted completion time* of a schedule, $\frac{1}{n}\sum_{j=1}^{n} w_j C_j$. In the scheduling notation this optimality criterion is denoted by $\sum w_j C_j$.

It is also possible that each job j may have an associated *due date* d_j by which it should be completed. This gives rise to two different optimality criteria. Given a schedule S, we define $L_j = C_j^S - d_j$ to be the *lateness* of job j, and we will be interested in constructing a schedule that minimizes $L_{\max} = \max_{j=1}^{n} L_j$, the *maximum lateness* of any job in the schedule. Alternatively, we concern ourselves with constructing a schedule that maximizes the number of jobs that complete by their due dates. To capture this, given a schedule S we define $U_j = 0$ if $C_j^S \leq d_j$ and $U_j = 1$ otherwise; we can thus describe our optimality criterion as the minimization of $\sum U_j$, or more generally, $\sum w_j U_j$. As illustrations, $1|r_j|L_{\max}$ denotes the problem of nonpreemptively scheduling, on one machine, jobs with release dates and due dates so as to minimize the maximum lateness of any job, and $1|prec|\sum w_j U_j$ denotes the problem of nonpreemptively scheduling precedence-constrained jobs on one machine so as to minimize the total (summed) weight of the late jobs.

Finally, we will consider one scheduling problem that deals with a more general optimality criterion. For each job j, we let $f_j(t)$ be any function that is nondecreasing with completion time of the job, and, with respect to a schedule S, define $f_{\max} = \max_{j=1}^{n} f_j(C_j^S)$. The specific problem that we will consider (in Section 3.1) is $1|prec|f_{\max}$ – the scheduling of precedence-constrained jobs on one machine so as to minimize the maximum value of $f_j(C_j)$ over all $j \in \mathcal{J}$.

1.2 More Complex Machine Environments: Parallel Machines and the Shop

Having introduced all of the optimality criteria, side characteristics and conditions that we will use in this survey, we now discuss more complex machine environments.

We first discuss *parallel machine* environments. In these environments we are given m machines. A job j with processing requirement p_j can be processed on any one of the machines, or, if preemption is allowed, started on one machine, and when preempted potentially continued on another machine. A machine can process at most one job at a time and a job can be processed by at most one machine at a time.

In the *identical parallel machine* environment the machines are identical, and job j requires p_j units of processing time when processed on any machine. In the *uniformly related machines* environment each machine i has a speed $s_i > 0$, and thus job j, if processed entirely on machine i, would take a total of p_j/s_i time to process. In the *unrelated parallel machines* environment we model machines that have different capabilities and thus their relative performance on a job is unrelated. In other words, the speed of machine i on job j, s_{ij}, depends on both the machine and the job; job j requires p_j/s_{ij} processing time on machine i. We define $p_{ij} = p_j/s_{ij}$.

In the *shop environment*, which primarily models various sorts of production environments, we again have m machines. In this setting a job j is made up of m *operations*, with each operation requiring processing on a specific one of the m machines. Different operations may take different amounts of time. In the *open shop* environment, the operations of a job can be processed in any order, as long as no two operations are processed on different machines simultaneously. In the *job shop environment*, there is a total order on the operations of a job, and one operation can not be started until its predecessor in the total order is completed. A special case of the job shop is the *flow shop*, in which the order of the operations is the same – each job requires processing on the same machines and in the same order, but different jobs may require different amounts of processing on the same machine.

In the scheduling notation, the identical, uniformly related and unrelated machine environments are denoted respectively by P, Q, and R. The open, flow and job shop environments are denoted by O, F and J. When the environment has a fixed number of machines the number is included in the environment specification; so, for example, P2 denotes the environment with two identical parallel machines. Note that Example 2 can be modeled by $\mathrm{P5}|prec|C_{\max}$, and Example 3 can be modeled by $\mathrm{F3}|r_j|\sum U_j$.

1.3 Contents

In this survey we focus exclusively on algorithms that provably run, in the worst case, in time polynomial in the size of the input. If the algorithm gives an optimal solution, we call it an *optimal* algorithm. Many of the problems that we consider, however, are $\mathcal{NP}$-hard, and thus it seems unlikely that polynomial-time algorithms exist to solve them. In these cases we will be interested in *approximation algorithms*; we define a ρ-approximation algorithm to be an algorithm that runs in polynomial time and delivers a solution of value at most ρ times optimal.

The rest of this chapter is organized as follows. In Section 2 we present a collection of simple heuristics that order the jobs based on a simple job-by-job priority and then schedule the jobs in priority order. These heuristics are useful both for solving certain problems optimally in polynomial time, and for giving simple yet high-quality approximations to certain $\mathcal{NP}$-hard scheduling problems. Many scheduling problems require a more complex approach than a simple priority rule; in Section 3 we study algorithms that are more sophisticated in their greedy choices. In Section 4 we discuss the application of some basic tools of combinatorial optimization, such as network optimization and linear programming, to the design of scheduling algorithms. We then turn exclusively to $\mathcal{NP}$-hard problems. In Section 5 we introduce the notion of a *relaxation* of a problem, and show how to use relaxations to design approximation algorithms. Finally, in Section 6 we discuss techniques by which certain other $\mathcal{NP}$-hard scheduling problems can be approximated arbitrarily closely in polynomial time.

2 Priority Rules

The most obvious approach to solving a scheduling problem is a greedy one: whenever a machine becomes available, assign some job to it. A more sophisticated variant of this approach is to give each job a *priority* derived from the particular optimality criterion, and then, whenever a machine becomes available, assign the available job of highest priority to it. In this section we discuss such scheduling strategies for one machine, parallel machine and shop problems. In all of our algorithms, the priority of a job can be determined without reference to other jobs. This typically gives a simple scheduling algorithm that runs in $O(n \log n)$ time—the bottleneck being the time needed to sort the jobs by priority. We also discuss the limitations of these approaches, giving examples of when they do not work well.

2.1 One Machine

We first focus on algorithms for single-machine problems in which we give each job a priority, sort by priorities, and schedule in this order. To establish the correctness of such a algorithms, it is often possible to apply an *interchange argument.* Suppose that there is an optimal schedule with jobs processed in non-priority order. It follows that some adjacent pair of jobs in the schedule has inverted priorities. We show that if we swap these two jobs, the scheduling objective function is improved, thus contradicting the claim that the original schedule was optimal.

2.1.1 $1||\sum w_j C_j$

In perhaps the simplest scheduling problem, our objective is to minimize the sum of completion times $\sum C_j$. Intuitively, it makes sense to schedule the largest job at the end of the schedule to ensure that it does not contribute to the delay on any other job. We formalize this by defining the `SPT` algorithm: order the jobs by nondecreasing processing time (breaking ties arbitrarily) and schedule in that order.

Theorem 2.1 `SPT` *is an optimal algorithm for* $1||\sum C_j$.

Proof: To establish the optimality of the schedule constructed by `SPT` we use an interchange argument. Suppose for the purpose of contradiction that the jobs in the optimal schedule are *not* scheduled in non-decreasing order of completion time. Then there is some pair of jobs j and k such that j immediately precedes k in the schedule but $p_j > p_k$.

Suppose we exchange jobs j and k. All jobs other than j and k still start, and thus complete, at the same time as they did before the swap. All that changes is the completion times of jobs j and k. Suppose that originally job j started at time t and ended at time $t + p_j$, so that job k started at time $t + p_j$ and finished at time $t + p_j + p_k$. It follows that the original contribution of these two jobs to the total completion time, namely $t + p_j + t + p_j + p_k = 2t + 2p_j + p_k$, is replaced by their new contribution of $2t + 2p_k + p_j$. This gives a net decrease of $p_j - p_k$ in $\sum C_j$, which is positive if $p_j > p_k$, implying that our original ordering was not optimal—a contradiction. □

This algorithm, and proof of optimality, generalizes to the optimization of average *weighted* completion time, $1||\sum w_j C_j$. Intuitively, we would like to schedule as much weight as possible with each unit of processing time. This suggests scheduling jobs in nonincreasing order of w_j/p_j; the optimality of this rule can be established by a simple generalization of the previous interchange argument.

Theorem 2.2 ([Smi56]) *Scheduling jobs in nonincreasing order of w_j/p_j gives an optimal schedule for $1||\sum w_j C_j$.*

2.1.2 $1||L_{\max}$

A simple greedy algorithm also solves $1||L_{\max}$, in which we seek to minimize the maximum job lateness. A natural strategy is to schedule the job that is closest to being late, which suggests the `EDD` algorithm: order the jobs by nondecreasing due dates (breaking ties arbitrarily) and schedule in that order.

Theorem 2.3 ([Jac55]) `EDD` *is an optimal algorithm for $1||L_{\max}$.*

Proof: We again use an interchange argument to prove that the the schedule constructed by `EDD` is optimal. Assume without loss of generality that all due dates are distinct, and number the jobs so that $d_1 < d_2 < \cdots < d_n$. Among all optimal schedules, we consider the one with the fewest *inversions*, where an inversion is a pair of jobs j, k such that $j < k$ but k is scheduled before j. Suppose the given optimal schedule is not the `EDD` schedule. Then there is a pair of jobs j and k such that $d_j < d_k$ but k immediately precedes j in the schedule.

Suppose we exchange jobs j and k. This does not change the completion time or lateness of any job other than j and k. We claim that we can only decrease $\max(L_j, L_k)$, so we do not increase the maximum lateness. Furthermore, since $j < k$, swapping jobs j and k decreases the number of inversions in the schedule. It follows that the new schedule has the same or better lateness than the original one but fewer inversions, a contradiction.

To prove the claim, note that originally $C_j > C_k$ but $d_j < d_k$. It follows that $\max(L_j, L_k) = C_j - d_j$. Under the exchange, job j's completion time, and thus lateness, decreases. Job k's completion time rises to (the old) C_j, but this gives it a lateness of $C_j - d_k < C_j - d_j$. Thus, the maximum of the two latenesses has decreased.

□

2.1.3 Preemption and Release Dates

We now consider the more complex one-machine environment in which jobs may arrive over time, as modeled by the introduction of release dates. The greedy heuristics of the previous sections are not immediately applicable, since jobs of high priority might be released relatively late and thus not be available for processing before jobs of lower priority. The most natural idea to cope with this complication is to always process the available (released) job of highest priority. In a preemptive setting, this would mean, upon the release of a job of higher priority, preempting the currently running job and switching to the "better" job. We will show that this idea in fact yields optimal scheduling algorithms.

We thus define the *Shortest Remaining Processing Time Algorithm* `SRPT` to, at each point in time, schedule the job with shortest remaining processing time, preempting when jobs of shorter processing time are released. We also generalize `EDD` to, upon the release of jobs with earlier dues dates than the job currently being processed, preempt the current job and process the job with the earliest due date.

Theorem 2.4 ([Bak74, Hor74]) `SRPT` *is an optimal algorithm for* $1|r_j, pmtn|\sum C_j$, *and* `EDD` *is an optimal algorithm for* $1|r_j, pmtn|L_{\max}$

Proof: As before, we argue by contradiction, using a similar greedy exchange argument. However, instead of exchanging entire jobs, we exchange pieces of jobs, which is now allowed in our preemptive environment.

We focus on $1|r_j, pmtn|\sum C_j$. Consider a schedule in which available job j with the shortest remaining processing time is not being processed at time t, and instead available job k is being processed. Let p'_j and p'_k denote the remaining processing times for jobs j and k after time t, so $p'_j < p'_k$. We now perform an exchange. Take the first p'_j units of time that were devoted to either of jobs j and k after time t, and use them instead to process job j to completion. Then, use the remaining p'_k units of time that were spent processing jobs j and k, and use them to schedule job j. This exchange preserves feasibility since both jobs were released by time t.

In the new schedule, all jobs other than j and k have the same completion times as before. Job k finishes when job j originally finished. But job j, which needed $p'_j < p'_k$ additional work, finishes *before* job k originally finished. Thus we have reduced $C_j + C_k$ without increasing any other completion time, meaning we have reduced $\sum C_j$, a contradiction.

The argument that EDD solved $1|r_j, pmtn|L_{\max}$ goes much the same way. If at time t, job j with the earliest remaining due date is not being processed and job k with a later due date is, we reallocate the time spent processing job k to job j. This makes job j finish earlier, and makes job k finish when job j did originally. This yields a better objective function value. □

By considering how SRPT and EDD function if all jobs are available at time 0, we conclude that on one machine, in the absence of release dates, preemption does not improve the quality of either the $\sum C_j$ or $L_{\max}$ optimality criteria. This is not the case when jobs have release dates; intuitively, a problem such as $1|r_j|\sum C_j$ seems more difficult, as one can not simply preempt the current job for a newly-arrived better one, but rather must decide whether to start a worse one or wait for the better one and incur idle time. This intuition about the additional difficulty of this setting is justified – $1|r_j|\sum C_j$ and $1|r_j|L_{\max}$ are in fact $\mathcal{NP}$-Complete problems. We discuss approximation algorithms for these problems in later sections.

We also note that these ideas have their limitations, and do not generalize to the $\sum w_j C_j$ criterion – $1|r_j, pmtn|\sum w_j C_j$ is $\mathcal{NP}$-hard. Finally, we note that SRPT and EDD are *on-line* algorithms – their decisions about which job to schedule currently do not require any information about which jobs are to be released in the future. See [Sga97] for a comprehensive survey of on-line scheduling.

2.2 The Two-Machine Flow Shop

We now consider a more complex machine environment in which we want to minimize the makespan in a flow shop. In general, this problem is $\mathcal{NP}$-hard, even in the case of three machines. However, in the special case of the two-machine flow shop, $F2||C_{\max}$ a priority-based ordering approach due to Johnson [Joh54] yields an optimal algorithm. We denote the operations of job j on the first and second machines as a pair (a_j, b_j). Intuitively, we want to get jobs done on the first machine as quickly as possible so as to minimize idleness on the second machine due to waiting for jobs from the first machine. This suggests using an SPT rule on the first machine. On the other hand, we want to process the jobs with large b_j as early as possible on the second machine, so they don't create a large tail of processing on machine 2 after machine 1 is finished. The obvious way to achieve this is to run the jobs with large b_j early on machine 2 while machine 1 is busy, so that machine 2 does not have much to process once machine 1 finishes. This suggests some kind of *longest processing time first* (`LPT`) rule for machine 2.

We now formalize this intuition. We partition our jobs into two sets. A is the set of jobs j for which $a_j \leq b_j$, while B is the set for which $a_j > b_j$. We construct a schedule by first ordering all the jobs in A by nondecreasing a_j value, and then all the jobs in B by nonincreasing b_j values. We process jobs in this order on both machines.

It may be surprising that we do not reorder jobs to process them on the second machine. It turns out that for two-machine flow shops, such reordering is not necessary. A schedule in which all jobs are processed in the same order is called a *permutation schedule.*

Lemma 2.5 *An instance of* $\mathrm{F2}||C_{\max}$ *always has an optimal schedule that is a permutation schedule.*

Proof: Consider any optimal schedule, and number the jobs according to the time at which they complete on machine 1. Suppose that job k immediately precedes job j in the order in which jobs are completed on machine 2, but $j < k$. Let t be the time at which job k is started on machine 2. It follows that job k has completed on machine 1 by time t. By our choice of ordering, it also follows that job j has completed on machine 1 by time t. Therefore, we can swap the order of jobs j and k on machine 2, and still have a legal schedule (since no other job's start time changes) with the same makespan. We can continue performing such swaps until there are none left to be done, implying that jobs on machine 2 are processed in the same order as those on machine 1. $\Box$

Having limited our search for optimal schedules to permutation schedules, we present a clever argument given by Lawler et. al. [LLKS93] to establish the optimality of the permutation schedule specified by Johnson's rule.

Renumber the jobs according to the Johnson ordering. Notice that in a permutation schedule for $\mathrm{F2}||C_{\max}$, there must be a job k that is started on machine 2 immediately after its completion on machine 1; for example, the job that starts immediately after the last idle time on machine 2. The makespan of the schedule is thus determined by the processing times of k jobs on machine 1 and $n-k+1$ jobs on machine 2, which is just a sum of $n+1$ processing times. If we reduce *all* the a_i and b_i by the same value p, then *every* sum of $n+1$ processing times decreases by $(n+1)p$, so the makespan of every permutation schedule is reduced by $(n+1)p$.

Now note that if a job has $a_i = 0$ it is scheduled first in some permutation schedule, since it delays no jobs on machine 1 and only "buys time" for jobs that are processed later than it on machine 2. Similarly, if a job has $b_i = 0$, it is scheduled last in some optimal schedule.

Therefore, we can construct an optimal permutation schedule by repeatedly finding the minimum value operation size amongst all the a_j and b_j values of the unscheduled jobs, subtracting that value from all of the operation sizes, and then scheduling the job with the new zero processing time according to the above rules. Now observe that the schedule constructed is exactly the schedule that orders the jobs by Johnson's rule. We have therefore proved the following.

Theorem 2.6 ([Joh54]) *Johnson's rule yields an optimal schedule for* $\mathrm{F}2||C_{\max}$.

2.3 Parallel machines

We now turn to the case of parallel machines. In the move to parallel machines, many problems that are easily solvable on one machine become $\mathcal{NP}$-hard; the focus therefore tends to be on approximation algorithms. In some cases, the simple priority-based rules we used for one machine generalize well. That is, we assign a priority to every job, and, whenever a machine becomes available, it starts processing the job that has the highest remaining priority. The schedules created by such algorithms, which immediately give work to any machine that becomes idle, will be referred to as *busy schedules.*

In this section, we also introduce a different method of analysis. Instead of arguing correctness based on interchange arguments, we give lower bounds on the quality of the optimal schedule. We then show that our algorithm produces a schedule whose quality is within some factor of the lower bound, thus demonstrating a fortiori that it is within this factor of the optimal schedule. This is a general technique for approximation, and it has the pleasing feature that we are able to guarantee that we are within a certain factor of the optimal value, *without* *knowing what that optimal value is.* Sometimes we can show that our greedy algorithm *achieves* the lower bound, thus demonstrating that the algorithm is actually optimal.

In this section, we devote most of our attention to the problem of minimizing the makespan (schedule length) on m parallel machines, and study the behavior of greedy algorithm for the problem. We remark that for the average-completion-time problem $\mathrm{P}||\sum C_j$, the greedy SPT algorithm also turns out to yield an optimal schedule. We discuss this further in Section 4.1.

As was mentioned in Section 2.1, $\mathrm{P}||C_{\max}$ is trivial when $m = 1$, as any schedule with no idle time will be optimal. Once we have more than one machine, things become more complicated. With preemption, it is possible to greedily construct an optimal schedule in polynomial time. In the non-preemptive setting, however, it is unlikely that there is an optimal polynomial time algorithm, since the problem is $\mathcal{NP}$-complete via a simple reduction from the $\mathcal{NP}$-complete Partition [GJ79]. We will thus focus on finding an approximately optimal solution. First, we will show that any busy schedule is a 2-approximation. We will then see how this can be improved with a slightly smarter algorithm, the *Longest Processing Time* (LPT) algorithm, which is a a 4/3-approximation algorithm. In Section 6 we will show that a more complicated algorithm can guarantee an even better quality of approximation.

Our analyses of these algorithms are all based on comparing their performance to certain lower bounds on the quality of the optimal schedule; their performance compared to the optimum can only be better. Our algorithms will make use of two simple lower bounds on $C^*_{\max}$, the makespan of the optimal schedule.

$$C^*_{\max} \geq \sum_{j=1}^{n} p_j/m \tag{1}$$

$$C^*_{\max} \geq p_j \text{ for all jobs } J_j \tag{2}$$

The first lower bound says that the schedule is at least as long as the average machine load, and the second says that the schedule is as least as long as the size of any job. To demonstrate the power of these lower bounds, we begin with the preemptive problem, $\mathrm{P}|pmtn|C_{\max}$. In this case, we show how to find a schedule that matches the maximum of the two lower bounds given above. We then use the lower bounds to establish approximation guarantees for the nonpreemptive case.

2.3.1 Minimizing $C_{\max}$ with preemptions

We give a simple algorithm, called McNaughton's[McN59] wrap-around rule, that creates an optimal schedule for $\mathrm{P}||C_{\max}$ with at most $m-1$ preemptions. This algorithm is different from many scheduling algorithms in that it creates the schedule machine by machine, rather than over time.

Observing that the lower bounds (1) and (2) still apply to preemptive schedules, we will give a schedule of length $D = \max\{\lceil \sum_j p_j/m \rceil, \max_j p_j\}$. We order the jobs arbitrarily. Then we begin placing jobs on the machines, in order, filling machine i up until time D before starting machine $i+1$. Thus, a job of length p_j may be split, assigned to the last t units of time of machine i and the first $p_j - t$ units of time on machine $i+1$, for some t. It is now easy to verify that since there are no more than mD units of processing, every job is scheduled, and because $D - t \geq p_j - t$ for any t, a job is scheduled on at most one machine at any time. Thus we have created an optimal preemptive schedule.

Theorem 2.7 ([McN59]) *McNaughton's wrap-around rule gives an optimal schedule for* $\mathrm{P}|pmtn|C_{\max}$ *time.*

2.3.2 $\mathrm{P}||C_{\max}$ – List Scheduling

In contrast to $\mathrm{P}||C_{\max}$, $\mathrm{P}||C_{\max}$ is $\mathcal{NP}$-hard. We consider the performance of the list scheduling (`LS`) algorithm, which is a greedy algorithm: whenever a machine becomes available, an unprocessed job will be processed. Then, we have the following result:

Theorem 2.8 ([Gra66]) `LS` *is a 2-approximation algorithm for* $\mathrm{P}||C_{\max}$.

Proof: Let j' be the last job to finish in the schedule constructed by `LS` and let $s_{j'}$ be the time that j' begins processing. $C_{\max}$ is therefore $s_{j'} + p_{j'}$. All machines must be busy up to time $s_{j'}$, since otherwise job j' could have been started earlier. The maximum amount of time that all machines can be busy is $\sum_{j=1}^{n} p_j/m$, and so we obtain that

$$\begin{aligned} C_{\max} &\leq s_{j'} + p_{j'} \\ &\leq \sum_{j=1}^{n} p_j/m + p_{j'} \\ &\leq C^*_{\max} + C^*_{\max} = 2C^*_{\max}. \end{aligned}$$

The last inequality comes from lower bounds (1) and (2) above. $\square$

This algorithm can easily be implemented in $O(n+m)$ time. By a similar analysis, the algorithm guarantees an approximation of the same quality even if the jobs have release dates.

2.3.3 Longest Processing Time First: $\mathrm{P}||C_{\max}$

It is useful to think of the analysis of `LS` in the following manner. Every job starts being processed before time $\sum_{j=1}^{n} p_j/m$, and hence the schedule length is no more than $\sum_{j=1}^{n} p_j/m$ plus the length of the longest job that is running at time $\sum_{j=1}^{n} p_j/m$.

This motivates the natural idea that it is good to run the longer jobs early in the schedule and the shorter jobs later. This is formalized in the `LPT` rule: sort jobs in nonincreasing order of processing time and list schedule in that order.

Theorem 2.9 ([Gra69]) `LPT` *is a 4/3-approximation algorithm for* $\mathrm{P}||C_{\max}$.

Proof: We first assume that j', the last job to finish in our schedule, is the last job to start. If it is not, remove all jobs that start after time $s_{j'}$. This does not affect the makespan of our schedule, since these jobs must have run on other machines. Furthermore, it can only decrease the optimal makespan for the modified instance. Thus, if we prove an approximation bound for this new instance, it applies a fortiori to our original instance. Therefore we can assume that the last job to finish is the smallest job and in this case, by the analysis of Theorem 2.8 above, `LPT` returns a schedule of length no more than $C^*_{\max} + p_{\min}$.

We consider two cases:

Case 1: $p_{\min} \le C^*_{\max}/3$ In this case $C^*_{\max} + p_{\min} \le C^*_{\max} + (1/3)C^*_{\max} \le (4/3)C^*_{\max}$.

Case 2: $p_{\min} > C^*_{\max}/3$ In this case, all jobs have $p_j > C^*_{\max}/3$, and hence in the optimal schedule there are at most 2 jobs per machine. Number the jobs in order of nonincreasing p_j. If $n \le m$, then the optimal schedule trivially puts one job per machines. We thus consider the remaining case with $m < n \le 2m$. In this case, we claim that, for each $j = 1, \ldots, m$ the optimal schedule pairs job j with job $2m+1-j$ if $2m+1-j \le n$ and places job j by itself otherwise. This can be shown to be optimal via a simple interchange argument. We finish the proof by observing that this is exactly the schedule that `LPT` would construct. $\square$

This algorithm needs to sort the jobs, and can be implemented in $O(m + n \log n)$ time. If we are willing to spend substantially more time, we can come arbitrarily close to optimal; see Section 6.

2.3.4 $\mathrm{P}|prec|C_{\max}$

Even when our input contains precedence constraints, list scheduling is still a 2-approximation algorithm. Given a precedence relation, we say that a job is available at time t if all its predecessors have completed processing by time t. Recall that in list scheduling, whenever a machine becomes idle, any available job is scheduled. Before giving the algorithm, we give one additional lower bound that is relevant to scheduling with precedence constraints. Let $j_{i_1}, j_{i_2}, \ldots, j_{i_k}$ be any set of jobs such that $j_{i_1} \prec j_{i_2} \prec, \ldots, \prec j_{i_k}$, then

$$C^*_{\max} \geq \sum_{\ell=1}^{k} p_{i_\ell}. \tag{3}$$

In other words the length of any chain of jobs is a lower bound on the makespan.

Theorem 2.10 ([Gra66]) `LS` *is a 2-approximation algorithm for* $\mathrm{P}|prec|C_{\max}$

Proof: Let j_1 be the last job to finish. Define j_2 to be the latest-finishing predecessor of j_1, and inductively define $j_{\ell+1}$ to be the latest-finishing predecessor of j_ℓ, continuing until reaching j_k, a job with no predecessors. Let $\mathcal{C} = \{j_1, \ldots, j_k\}$. We partition time into two sets, A, the points in time when a job in $\mathcal{C}$ is running, and B, the remaining time. Observe that during all times in B, all machines must be busy, for if they were not, there would be a job from $\mathcal{C}$ that had all its predecessors completed and would be ready to run. Hence, $C_{\max} \leq |A| + |B| \leq \sum_{j \in \mathcal{C}} p_j + \sum_{j=1}^{n} p_j/m \leq 2C^*_{\max}$, where the last inequality follows by applying lower bounds (3) and (1). $\square$

For the case when all processing times are exactly one, $\mathrm{P}|prec|C_{\max}$ is solvable in polynomial time if there are only 2 machines[Law76], and is $\mathcal{NP}$-complete if there are an arbitrary number of machines[Ull75]. The complexity of the problem in the case when there are a fixed constant number of machines, e.g. 3, is one of the more famous open problems in scheduling.

2.3.5 Open Shop Scheduling

List Scheduling can also be applied to $O||C_{\max}$. Recall that in this problem, each job must be processed for disjoint intervals of time on several different machines. By an analysis similar to that used for $P||C_{\max}$, we will show that any algorithm that constructs a busy schedule for $O||C_{\max}$ is a 2-approximation algorithm. Let $P_{\max}$ be the maximum total processing time, summed over all machines, for any one job, and let $\Pi_{\max}$ be the maximum total processing time, summed over all jobs, of any one machine. Clearly, both $P_{\max}$ and $\Pi_{\max}$ are lower bounds on the makespan of the optimal schedule. We show that the natural List Scheduling generalization of processing any available operation on a free machine constructs a schedule of makespan at most $P_{\max} + \Pi_{\max}$.

To see this, consider the machine M that finishes processing last, and consider the last job j to finish on machine M. At any time during the schedule, either M is processing a job or job j is being processed (if neither of these is true, then list scheduling would require that j be running on M, a contradiction). However, the total length of time during which j undergoes processing is at most $P_{\max}$. During all the remaining time in the schedule, machine M must be busy. But machine M is busy for at most $\Pi_{\max}$ time units. Thus the total length of the schedule is at most $P_{\max}+\Pi_{\max}$, as claimed. Since $P_{\max}+\Pi_{\max} \leq 2\max(P_{\max}, \Pi_{\max}) \leq 2C^*_{\max}$, we obtain

Theorem 2.11 (Racsmány, see [BF82]) *List Scheduling is a 2-approximation algorithm for* $O||C_{\max}$.

2.4 Limitations of Priority Rules

For many problems, simple scheduling rules do not yield good schedules, and thus given a scheduling problem, the algorithm designer should be careful about applying one of these rules without justification. In particular, for many problems, particularly those with precedence constraints and release dates, the optimal schedule has *unforced idle time*. That is, if one is constructing the schedule over time, there may be a time t when there is an idle machine m and an available job j , but scheduling job j on machine m at time t will yield a sub-optimal schedule.

Consider the problem Q$||C_{\max}$ and recall that for P$||C_{\max}$, list scheduling is a 2-approximation algorithm. Consider a 2-job 2-machine instance in which $s_1 = 1$, $s_2 = x$, $p_1 = 1$,$p_2 = 1$, and $x > 2$. Then `LS SPT` and `LPT` all schedule one job on machine 1 and one on machine 2, and the makespan is thus 1. However, the schedule that places both jobs on machine 2 has makespan $2/x < 1$. By making x arbitrarily large, we see that none of these simple algorithms, which all have approximation ratio $x/2$, have bounded approximation ratios.

For this problem there is actually a simple heuristic that comes within a factor of 2 of optimal, but for some problems, such as Q$|prec|C_{\max}$ and R$||C_{\max}$, there is no simple algorithm known that comes anywhere close to optimal. We also note that even though list scheduling is a 2-approximation for O$||C_{\max}$, for F$||C_{\max}$ it can be shown that busy schedules can be of makespan $\Omega(m)$ times optimal.

3 Sophisticated Greedy Approaches

For many problems, the priority algorithms that consider jobs in isolation, as in Section 2, are not sufficient. In this section, we consider algorithms that, rather than assigning a priority to a job and simply sorting by priorities, take other jobs into account when making a decision about where to schedule a job. The algorithms we study here are "incremental" in nature: they start with an empty solution and grow it, one job at a time, until the optimal solution is revealed. At each step, the decision about which job to add to the growing solution is made greedily, but is based on the current context of jobs which have already been scheduled. We present two examples which are classic examples of the *dynamic programming* paradigm, and others that are more specialized.

All the algorithms share an analysis based on the idea of *optimal substructure.* Namely, if we consider the optimal solution to a problem, we can often argue that its "subparts" (e.g., prefixes of the optimal schedule) are optimal solutions to "subproblems" (e.g., the problem of scheduling the set of jobs in that prefix). This lets us argue that as our algorithms build their solution incrementally, they are building optimal solutions to bigger and bigger subproblems of the original problem, until they reach an optimal solution to the entire problem.

3.1 Incremental Greed: $1||f_{\max}$

The first problem we consider is $1||f_{\max}$, which was defined in Section 1. In this problem, each job has some nondecreasing *penalty function* on its completion time C_j, and the goal is to find a schedule minimizing the maximum $f_j(C_j)$. As one example, $1||L_{\max}$ is captured by setting $f_j(t) = t - d_j$.

A greedy strategy still applies, when suitably modified. It is convenient, instead of talking about scheduling the "most penalizing" (e.g. earliest due date) job first, to talk about scheduling the "least penalizing" (e.g. latest due date) job last. For $S \subseteq \mathcal{J}$ let $p(S) = \sum_{j \in S} p_j$ be the total time to process jobs in S. We find the job j that minimizes $f_j(p(\mathcal{J}))$, and schedule this job last. We then (recursively) schedule all the remaining jobs before j so as to minimize their maximum penalty. We call this algorithm `Least-Cost-Last` .

Observe the difference between this and our previous scheduling rules. In this new scheme, we cannot determine the best job to schedule second-to-last until we know which job is scheduled last (we need to know the processing time of the last job). Thus, instead of a simple $O(n \log n)$-time sorting algorithm based on absolute priorities, we are faced with an algorithm that inspects k jobs in order to identify the job to be scheduled k^{th}, giving a total running time of $O(n + (n-1) + \cdots + 1) = O(n^2)$.

This change in algorithm is matched by a change in analysis. Since the notion of which job is worst can change as the schedule is constructed, there is no obvious fixed priority to which we can apply a local exchange argument. Instead, as with $\mathrm{P}|pmtn|C_{\max}$ in Section 2.3.1, we show that our algorithm's greedy decisions are in agreement with a provable lower bound on the quality of the optimal schedule. Our algorithm produces a schedule that matches the lower bound and must therefore be optimal.

Let $f^*_{\max}(S)$ denote the optimal value of the objective function if we are only scheduling the jobs in S. Consider the following two facts about $f^*_{\max}$:

$$\begin{aligned} f^*_{\max}(\mathcal{J}) &\geq \min_{j \in N} f_j(P(\mathcal{J})) \\ f^*_{\max}(\mathcal{J}) &\geq f^*_{\max}(\mathcal{J} - \{j\}) \end{aligned}$$

The first of these statements follows from the fact that some job must be scheduled last. The second follows from the fact that if we have an optimal schedule for $\mathcal{J}$ and remove a job from the schedule, then we do not increase the completion time of any job. Therefore, since the f_j are increasing functions, we do not increase any penalty.

We use these inequalities to prove by induction that our schedule is optimal. According to our scheduling rule, we schedule last the job j minimizing $f_j(P(\mathcal{J}))$. By induction, this gives us a schedule with objective $\max\{f_j(P(\mathcal{J})), f^*_{\max}(\mathcal{J} - \{j\})\}$. But since each of these quantities is (by the equations above) a lower bound on the optimal $f^*_{\max}(\mathcal{J})$, we see that in fact we obtain a schedule whose value is a lower bound on $f^*_{\max}(\mathcal{J})$, and thus must in fact equal $f^*_{\max}(\mathcal{J})$.

3.1.1 $1|prec|f_{\max}$

Our argument from the previous section continues to apply even if we introduce *precedence constraints*. In the $1|prec|f_{\max}$ problem, a partial order on jobs is given, and we must build a schedule that does not start a job until all jobs preceding it in the partial order have completed. Our above algorithm applies essentially unchanged to this case. Note that the last job in the schedule must be a job with no successors. We therefore build an optimal schedule by scheduling last the job j that, among jobs with no successors, minimizes $f_j(P(\mathcal{J}))$. We then recursively schedule all other jobs before it. The proof of optimality goes exactly as before, using the fact that if L is the set of all jobs without successors,

$$f^*_{max}(\mathcal{J}) \geq \min_{j \in L} f_j(P(\mathcal{J}))$$

This is the same as the first equation above, except that the minimum is taken only over jobs without successors. The remainder of the proof proceeds unchanged.

It should also be noted that, once again, the fact that our algorithm is greedy makes preemption a moot point. One job needs to finish last, and it immediately follows that we can do no better than executing all of that job last. Thus, our greedy algorithm continues to apply.

Theorem 3.1 ([Law73]) `Least-Cost-Last` *is an optimal algorithm for* $1|prec|f_{\max}$.

3.1.2 An alternative approach

Moore [Moo68] gave a different approach to $1||f_{\max}$ that may be faster in some cases. His scheme is based on a reduction to the maximum lateness problem and its solution by the `EDD` rule. To see how an algorithm for $L_{\max}$ can be applied to $1||f_{\max}$, suppose we want to know whether there is a schedule with $f_{\max} \leq B$. We can decide this as follows. Give each job j a deadline d_j equal to the maximum t for which $f_j(t) \leq B$. It is easy to see that a schedule has $f_{\max} \leq B$ precisely when every job finishes by its specified deadline. Thus, we have converted the feasibility problem for $f_{\max}$ into an instance of the lateness problem. The optimization problem may therefore be solved by a binary search for the correct value of B.

3.2 Dynamic Programming: $1||\sum w_j U_j$

We now consider $1||\sum w_j U_j$ problem, in which the goal is to minimize the total weight of late jobs. This problem is *weakly $\mathcal{NP}$-complete.* That is, although it is $\mathcal{NP}$-complete, for integral weights it is possible to solve the problem exactly in $O(n \sum w_j)$ time, which is polynomial if the w_j are bounded by a polynomial. The necessary algorithm is a classical dynamic program that builds the solution out of solutions to smaller problems (a detailed introduction to dynamic programming can be found in many algorithms textbooks, see, for example [CLR90]). This immediately yields an $O(n^2)$ algorithm for $1||\sum U_j$ problem–just take all weights to be 1. Furthermore, we will show in Section 6 that this algorithm can be used to derive an *fully polynomial approximation scheme* for general problem that finds a schedule with $\sum w_j U_j$ within $(1+\epsilon)$ of the optimum in time polynomial in ϵ and n.

The first observation to make is that under this objective, a schedule partitions the jobs into two types: those completed by their due dates, and those not completed. Clearly, we might as well process the jobs that meet their due date before processing any that do not. Furthermore, the order of processing of these jobs might as well be met using the Earliest Due Date (`EDD`) rule from Section 2.1.2: when all jobs can be completed by their due date (implying nonpositive maximum lateness) `EDD`, which minimizes maximum lateness, will clearly find a schedule that does so.

It is therefore convenient to discuss *feasible subsets* of jobs that can all be scheduled together to complete by their due dates. The question of finding a minimum weight set of late jobs can then be equivalently restated as finding a maximum weight feasible subset of the jobs.

To solve this problem, we aim to solve a harder one: namely, to identify the fastest-completing maximum weight feasible subset. We do so via dynamic programming. Order the jobs according to increasing due date. Let T_{wj} denote the minimum completion time of a weight w-or-greater feasible subset of $1, \ldots, j$, or ∞ if there is no such subset. Note that $T_{0j} = 0$, while $T_{w0} = \infty$ for all $w > 0$. We now give a dynamic program to compute the other values T_{wj}. Consider the fastest completing weight w-or-greater feasible subset of $\{1, \ldots, j+1\}$. Either $j+1$ is in this subset or it is not. If $j+1$ is not in the schedule, then the subset is a subset of $\{1, \ldots, j\}$ and clearly completes in time T_{wj}. If $j+1$ is in the feasible subset, then since we can schedule feasible subsets using EDD, $j+1$ can be scheduled last. The jobs preceding it have weight at least $w - w_{j+1}$, and clearly are the minimum-completion-time subset of $1, \ldots, j$ with this weight. Thus, the time of this feasible set is $T_{w-w_{j+1},j} + p_{j+1}$. It follows that

$$T_{w,j+1} = \begin{cases} \min(T_{w,j}, T_{w-w_{j+1}} + p_{j+1}) & \text{if } T_{w-w_{j+1},j} + p_j \leq d_{j+1} \\ T_{wj} & \text{otherwise} \end{cases}$$

Now observe that there is clearly no feasible subset of weight exceeding $\sum w_j$, so we can stop our dynamic program once we reach this value of w. This takes $O(n \sum w_j)$ time. Once we have all the values T_{wj}, we can find the maximum weight feasible subset by identifying the largest value of w for which some T_{wj} (and thus T_{wn}) is finite.

This gives a standard $O(n \sum_j w_j)$ time dynamic program for computing T_{wn} for every relevant value w; the maximum w for which T_{wn} is finite is the maximum total weight of jobs that can be completed by their due date.

Theorem 3.2 ([LM69]) *Dynamic programming yields an $O(n \sum w_j)$-time algorithm for exactly solving $1||\sum w_j U_j$.*

We remark that a similar dynamic program can be used to solve the problem in time $O(n \sum p_j)$, which is effective when the processing times are polynomially bounded integers. We also note that a quite simple greedy algorithm due to Moore [Moo68] can solve the unweighted $1||\sum U_j$ problem in $O(n \log n)$ time.

3.3 Another dynamic program: $\mathrm{P}||C_{\max}$

For a second example of the applicability of dynamic programming, we return to the $\mathcal{NP}$-hard problem $\mathrm{P}||C_{\max}$, and focus on a special case that is solvable in polynomial time—the case in which the number of different job processing times is bounded by a constant. While this special case might appear to be somewhat contrived, in Section 6 we will show that it can form the the core of a polynomial approximation scheme for $P||C_{\max}$.

Lemma 3.3 *Given an instance of* $P||C_{\max}$ *in which the* p_j *take on at most* s *distinct values, there exists an algorithm which finds an optimal solution in time* $n^{O(s)}$.

Proof: Assume for now that we are given a target schedule length T. We again use dynamic programming. Let the different processing times be $z_1, \ldots, z_s$. The key observation is that the set of jobs on a machine can be described by a length-s vector $V = (v_1, \ldots, v_s)$, where v_k is the number of jobs of length z_k. There are at most n^s such vectors since each entry has value at most n. Let V be the set of all such vectors whose total processing time (that is, $\sum v_i z_i$) is less than T. In the optimal schedule, every machine is assigned a set of jobs corresponding to a vector from this set. We now define $M(x_1, \ldots, x_k)$ to be the minimum number of machines needed to schedule a job set consisting of x_i jobs of size z_i, for $i = 1, \ldots, s$. We observe in the standard dynamic-programming fashion that

$$M((x_1, \ldots, x_s) = 1 + \min_{v \in V} M(x_1 - v_1, \ldots, x_s - v_s).$$

The minimization is over all possible vectors that could be processed by the "first" machine counted by the quantity 1, and the recursive expression denotes the best way to process the remaining work. Thus we need to compute an n^s entry table, where each entry depends on n^s other entries, and therefore the computation takes time $O(n^{2s})$.

It remains to handle the assumption that we know T. The easiest way to do this is to perform a binary search on all possible values of T. A slightly more sophisticated approach is to search only over the $O(n^s)$ makespans of vectors describing sets of jobs, as one of these clearly determines the makespan of the solution. $\square$

4 Matching and Linear Programming

Networks and linear programs are central themes in combinatorial optimization, and are useful tools in the solution of many problems. Therefore it is not surprising that these techniques can be applied profitably to scheduling problems as well. In this section, we discuss applications of *bipartite matching* and *linear programming* to the exact solution of certain scheduling problems;

in Section 5 we will revisit both techniques in the design of approximation algorithms for $\mathcal{NP}$-hard problems.

4.1 Applications of Matching

Given a bipartite graph on two sets of vertices A and B and an edge set $E \subseteq A \times B$, a *matching* is a set of edges, the set of whose endpoints contains each vertex of A and B at most once. A natural matching that is useful in scheduling problems is one that matches jobs to machines; the matching constraints constrain each job to be scheduled on at most one machine, and each machine to be processing at most one job. If A has no more vertices than B, we call a matching *perfect* if every vertex of A is in some matching edge. It is also possible to assign weights to the edges, and define the weight of a matching to be the sum of the weights of the matching edges. They key fact that we use in this section is that minimum weight perfect matchings can be computed in polynomial time.

4.1.1 Matching to Schedule Positions: $\mathrm{R}||\sum C_j$

In this section we give a polynomial-time algorithm for $\mathrm{R}||\sum C_j$ that matches jobs to *positions* in the schedule on each machine. For any schedule, let κ_{ik} be the k^{th}-from-last job to run on machine i, and let ℓ_i be the number of jobs that run on machine i. Then by observing that the completion time of a job is equal to the sum of the processing times of the jobs that run before it, we have that

$$\sum_j C_j = \sum_{i=1}^{m}\sum_{k=1}^{\ell_i} C_{\kappa_{ik}} = \sum_{i=1}^{m}\sum_{k=1}^{\ell_i}\sum_{x=k}^{\ell_i} p_{i,\kappa_{xi}} = \sum_{i=1}^{m}\sum_{k=1}^{\ell_i} k p_{i,\kappa_{ki}} \tag{4}$$

From this, we see that the k^{th} from last job to run on a machine contributes exactly k times its processing time to the sum of completion times. Based on this observation, Horn [Hor73] and Bruno, Coffman and Sethi [BCS74] proposed formulating the problem as a minimum-weight bipartite matching problem. We define a bipartite graph $G = (V, E)$ with $V = A \cup B$ as follows. A will contain n vertices v_j, one for each of the n jobs $j = 1, \ldots, n$. B will contain nm nodes w_{ik}, where vertex w_{ik} represents the kth-from-last position on machine i, for $i = 1, \ldots, m$ and $k = 1, \ldots, n$. We include in E an edge (v_j, w_{ik}) between every node in A and every node in B. Using (4) we define the weights on the edges from A to B as follows: edge (v_j, w_{ik}) is assigned weight kp_{ij}.

We now argue that a minimum-weight perfect matching in this graph corresponds to an optimal schedule. First, note that for each valid schedule there is a perfect matching in G. Not every perfect matching in G corresponds to a schedule, since a job might might be assigned to the kth from last position while less than k jobs are assigned to that machine; however, such a perfect matching is clearly not of minimal weight – a better matching can be obtained by pushing the $k' < k$ jobs assigned to that machine into the k' from last slots. Therefore, a schedule of minimum total completion time corresponds to a minimum-weight perfect matching in the bipartite graph.

Theorem 4.1 ([Hor73, BCS74]) *There is a polynomial-time algorithm for* $\mathrm{R}||\sum C_j$.

In the special case of parallel identical machines, it remains true that the k^{th} from last job to run on a machine contributes exactly k times its processing time to the sum of completion times. Since in this case the processing time of each job is the same on any machine, the algorithm is clear: schedule the m largest jobs last on each machine, schedule the next m largest jobs next to last, etc. The schedule constructed is exactly that constructed by the `SPT` algorithm.

Corollary 4.2 ([CMM67]) `SPT` *is an optimal algorithm for* $\mathrm{P}||\sum C_j$.

4.1.2 Matching Jobs to Machines: $\mathrm{O}|pmtn|C_{\max}$

For our second example of the utility of matching, we give an algorithm for $\mathrm{O}|pmtn|C_{\max}$ due to Gonzales and Sahni [GS76]. This algorithm will not find just one matching, but rather a sequence of matchings, each of which will correspond to a partial schedule, and then concatenate all of these partial schedules together. Recall from our discussion of $\mathrm{O}||C_{\max}$ in Section 2.3.5 that two lower bounds on the makespan of a nonpreemptive schedule are the *maximum machine load* $\Pi_{\max}$ and the *maximum job size* $P_{\max}$. Both of these remain lower bounds when preemption is allowed. In the nonpreemptive setting, a simple greedy algorithm gives a schedule with makespan bounded by $P_{\max} + \Pi_{\max}$. We now show that when preemption is allowed, matching can be used to achieve a makespan equal to $\max(P_{\max}, \Pi_{\max})$.

The intuition behind the algorithm is the following. Consider the schedule at any point in time. At this time, each machine is processing at most one job. In other words, the schedule at each point in time defines a matching between jobs and machines. We aim to find a matching that forms a part of the optimal schedule, and process jobs according to it for some time. Our goal is that processing the matched jobs on their matched machines for some amount of time t, and adjusting $P_{\max}$ and $\Pi_{\max}$ to reflect the decreased remaining processing requirements, should reduce $\max(P_{\max}, \Pi_{\max})$ by t. It follows that if we repeat this process for a total amount of time equal to $\max(P_{\max}, \Pi_{\max})$, we will reduce $\max(P_{\max}, \Pi_{\max})$ to 0, implying that there is no work remaining in the system.

What properties should our matching of jobs to machines have? Recall that our goal is to reduce our lower bound. Call a job *tight* if its total processing cost is $P_{\max}$. Call a machine *tight* if its total load is $\Pi_{\max}$. Clearly, it is necessary that every tight job undergo processing in our matching, since otherwise we will fail to subtract t from $P_{\max}$. Similarly, it is necessary that every tight machine be in the matching in order to ensure that we reduce $\Pi_{\max}$ by t. Lastly, we can only execute the matching for t time if every job-machine pair in the matching actually requires t units of processing. In other words, we are seeking a matching in which every tight machine and job is matched, and each matching edge requires positive processing time. Such a matching is referred to as a *decrementing set*. That it always exists is a nontrivial fact (about stochastic matrices) whose proof is beyond the scope of this survey; we refer the reader to Lawler and Labetoulle's presentation of this algorithm [LL78].

To find a decrementing set, we construct a (bipartite) graph with a node representing each job and machine, and include an edge between machine node i and job node j if job j requires a non-zero amount of processing on machine i. In this graph we require a matching that matches each tight machine or job node; this can easily be found with a slight variant of traditional matching algorithms.

Once we have found our decrementing set via matching, we have machines execute the jobs matched to them until one of the matched jobs completes its work on its machine, or until a new job or machine becomes tight (this can happen because some jobs and machines are not being processed in the matching). Whenever this happens, we find a new decrementing set. For simplicity, we assume that $P_{\max} = P_{\max}$; this can easily be arranged by adding dummy operations, which can only make our task harder. Since our decrementing set includes every tight job and machine, it follows that executing for time t will reduce both $P_{\max}$ and $\Pi_{\max}$ by

t. It follows that after $P_{\max} = \Pi_{\max}$ time, both quantities will be reduced to 0. Clearly this means that we are done in time equal to the lower bound.

One might worry that the number of decrementing set calculations we must perform could be non-polynomially bounded, making our approximation algorithm too slow. But this turns out not to happen. We only compute a new decrementing set when a job or machine finishes or when a new job or machine becomes tight. Each job-processor pair can finish only once, meaning that this occurs only nm times during our schedule. Also, each job or machine stays tight forever after it becomes tight; thus, new tight jobs and machines only occur $n + m$ times. Thus, constructing our schedule of optimal length requires only $mn + m + n$ matching computations.

Theorem 4.3 ([GS76]) *There is a polynomial time algorithm for* $\mathrm{O}|pmtn|C_{\max}$*, that finds an (optimal) schedule of makespan* $\max(P_{\max}, \Pi_{\max})$*.*

4.2 Linear Programming

We now discuss the application of *linear programming* to the design of scheduling algorithms. Given a vector of variables $x =< x_1, \ldots, x_n >$ a *linear program* is given by a set of m linear constraints of the form $a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n \leq b_i, 1 \leq i \leq m$, and a *cost vector* $c =< c_1, \ldots, c_n >$; the goal is to find an x that satisfies these constraints and that minimizes $cx = c_1x_1 + c_2x_2 + \ldots + c_nx_n$. Alternatively but equivalently, some of the inequality constraints might be given as equalities, and/or we may have no objective function and desire simply to find a feasible solution to the set of constraints. Many optimization problems can be formulated as linear programs, and thus solved efficiently, since a linear program can be solved in polynomial time [Kha79].

In this section we consider $\mathrm{R}|pmtn|C_{\max}$. To model this problem as a linear program, we use nm variables $x_{ij}, 1 \leq i \leq n, 1 \leq j \leq m$. Variable x_{ij} denotes the fraction of job j that is processed on machine i; for example, we would interpret a linear-programming solution with $x_{j1} = x_{j2} = x_{j3} = \frac{1}{3}$ as assigning $\frac{1}{3}$ of job j to machine 1, $\frac{1}{3}$ to machine 2 and $\frac{1}{3}$ to machine 3.

We now consider what sorts of linear constraints on the x_{ij} are necessary to ensure that they describe a valid solution to an instance of $\mathrm{R}|pmtn|C_{\max}$. Clearly the fraction of a job assigned to any machine must be positive, so we will create nm constraints

$$x_{ij} \geq 0.$$

In any schedule, we must fully process each job. We capture this requirement with the n constraints:

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

Note that, along with the previous constraints, these constraints imply that $x_{ij} \leq 1 \; \forall \; i, j$.

Our objective, of course, is to minimize the makespan D of the schedule. Recall that the amount of processing that job j would require, if run entirely on machine i, is p_{ij}. Therefore, for a set of fractional assignments x_{ij}, we can determine the amount of time machine i will work: it is just $\sum x_{ij} p_{ij}$, which must be at most D. We model this with the m constraints

$$\sum_{j=1}^{n} p_{ij} x_{ij} \leq D \text{ for } i = 1, \ldots, m.$$

Finally, we must ensure that no job is processed for more than D time; we model this with the n constraints

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

To summarize, we formulate the problem as the following linear program:

$$\begin{aligned}
\min \quad & D & & & (5)\\
\sum_{i=1}^{m} x_{ij} \quad &= \quad 1, & & \text{for } j = 1, \ldots, n, & (6)\\
\sum_{j=1}^{n} p_{ij} x_{ij} \quad &\leq \quad D, & & \text{for } i = 1, \ldots, m, & (7)\\
\sum_{i=1}^{n} p_{ij} x_{ij} \quad &\leq \quad D, & & \text{for} j = 1, \ldots, n, & (8)\\
x_{ij} \quad &\geq \quad 0 & & i = 1, \ldots, m, \; j = 1, \ldots, n. & (9)
\end{aligned}$$

It is clear that any feasible schedule for our problem yields an assignment of values to the x_{ij} that satisfies the constraints of our above linear program. However it is not completely clear that solving the linear program yields a solution to the scheduling problem; this linear program does not specify the ordering of the jobs on a specific machine, but simply assigns the jobs to machines while constraining the maximum load on any machine. It thus fails to explicitly require that a job not be processed simultaneously on each machine.

Interestingly enough, we can resolve this difficulty with an application of open-shop scheduling. We define an open shop problem by creating an operation o_{ij} for each positive variable x_{ij}, and define the size of o_{ij} to be $x_{ij}p_{ij}$. We then find an optimal preemptive schedule for this instance, using the matching-based algorithm discussed in Section 4.1.2. We know that both the maximum machine load and maximum job size of this open shop instance are bounded above by D, and therefore the makespan of the resulting open shop schedule is at most D. If we now reinterpret the operations of each job in the open-shop schedule as fragments of the original job in the unrelated machines instance, we see that we have given a preemptive schedule of length D in which no two fragments of a job are scheduled simultaneously.

We thus have established the following.

Theorem 4.4 ([LL78]) *There is an optimal algorithm for* $\mathrm{R}|pmtn|C_{\max}$

We will see further applications of linear programming to the development of approximation algorithms for $\mathcal{NP}$-hard scheduling problems in the next section.

5 Using Relaxations to Design Approximation Algorithms

We now turn to the design of approximation algorithms for $\mathcal{NP}$-hard scheduling algorithms. Many of the approximation algorithms in this area are based on a *relaxation* of the $\mathcal{NP}$-hard problem. A relaxation of a problem is a version of the problem with some of the requirements or constraints removed ("relaxed"). For example, we might consider $1|r_j, pmtn|\sum C_j$ to be a relaxation of $1|r_j|\sum C_j$ in which the "no preemption" constraint has been relaxed. A second example of a relaxation might be a version of the problem in which we relax the constraint that a machine processes at most one job at a time; a solution to this relaxation may have several jobs scheduled at one time on the same machine.

A solution to the original problem is a solution to the relaxation, but a solution to the relaxation is not necessarily a solution to the original problem. This is clearly illustrated by our nonpreemptive/preemptive example – a nonpreemptive schedule is a fine solution to a preemptive problem, although perhaps not an optimal one, but the converse is not true. It follows that in the case of a minimization problem, the value of the optimal solution to the relaxation is a not-necessarily-tight lower bound on the optimal solution to the original problem.

An idea that has proven quite useful is is first to define a relaxation of the problem which can be solved in polynomial time, and then to give an algorithm to convert the relaxation's solution into a valid solution to the original problem, with some degradation in the quality of solution. The key to making this work work well is to find a relaxation that preserves most of the structure of the original problem, so that the optimal relaxed solution is similar to the original optimum and thus does not degrade too much when converted to a valid solution.

In this section we discuss two sorts of relaxations of scheduling problems and their use in the design of approximation algorithms, namely the preemptive version of a nonpreemptive problem and a linear-programming relaxation of a problem. The algorithms discussed in this section are examples of more general categories of algorithms: algorithms that infer a *job ordering* from the relaxation and use that to construct a schedule, and algorithms that infer from the relaxation assignments of jobs to machines. We begin with the latter.

Before going any further, we introduce the notion of a relaxed decision procedure, which we will use both in Section 5.1 and later in Section 6. A *ρ-relaxed decision procedure* for a minimization problem accepts as input a target value T, and returns either **no**, asserting that no solution of value $\leq T$ exists, or returns a solution of value at most ρT. A polynomial-time *ρ-relaxed decision procedure* can easily be converted into a ρ-approximation algorithm for the problem via binary search for the optimum T; see [HS87, Hoc97] for more details. This simple idea is quite useful, as it essentially lets us pretend that we know the value T of an optimal solution to a problem.

5.1 Rounding a Fractional Assignment to Machines: $R||C_{\max}$

In this section we give a 2-relaxed decision procedure for $R||C_{\max}$. Recall the linear program that we used in giving an algorithm for $R|pmtn|C_{\max}$. If, instead of the constraints $x_{ij} \geq 0$ we could constrain the x_{ij} to be 0 or 1, the solution would constitute a valid nonpreemptive schedule. Furthermore, note that these integer constraints combined with the constraints (7) make the constraints (8) unnecessary. In other words, the following formulation has a feasible solution if and only if there is a nonpreemptive schedule of makespan D.

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad \text{for } j = 1, \ldots, n, \tag{10}$$

$$\sum_{j=1}^{n} p_{ij} x_{ij} \leq D, \qquad \text{for } i = 1, \ldots, m, \tag{11}$$

$$x_{ij} \in \{0, 1\} \qquad i = 1, \ldots, m, \; j = 1, \ldots, n. \tag{12}$$

This is an example of an *integer linear program*, in which the variables are constrained to be integers. Unfortunately, in contrast to linear programming, finding a solution to an integer linear program is $\mathcal{NP}$-complete. However, this integer programming formulation will still be useful. A very common method for obtaining a relaxation of an optimization problem is to formulate it as an integer linear program, and then to relax the integrality constraints. One obtains a fractional solution and then *rounds* the fractions to integers in a fashion that (hopefully) does not degrade the solution too dramatically.

In our setting, we relax the constraints (12) to $x_{ij} \geq 0$. We will also add an additional set of constraints that will ensure that the fractional solutions to this linear program have enough structure to be useful for approximation. Specifically, we disallow any part of a job j being processed on a machine i on which it could not complete in D time in a nonpreemptive schedule. Specifically, we include the following constraints

$$x_{ij} = 0, \qquad \text{if } p_{ij} \geq D. \tag{13}$$

(In fact, instead of adding constraints, we can simply remove such variables from the linear program.) Note that this constraint is actually implicit in the integer program given by the constraints (10) through (12), but was no longer guaranteed when we relaxed the integer constraints.

To recap, constraints (10), (11), and (13) along with $x_{ij} \geq 0$ consitute a *linear-programming relaxation* of $\mathrm{R}||C_{\max}$. Our relaxed decision procedure attempts solve this relaxation, obtaining a solution $\bar{x}_{ij}, 1 \leq i \leq m, 1 \leq j \leq n$. If there is no feasible solution, our RDP can output **no** – no nonpreemptive schedule has makespan D or less. If the linear program is feasible, we will give a way to derive an integral assignment of jobs to machines from the fractional solution.

Our job is made much easier by the fact, which we cite from the theory of linear programming, that we can find a so-called *basic* solution of this linear program that has at most $n+m$ positive variables. Since these $n+m$ positive variables must be distributed amongst n jobs, there are at most m jobs that are assigned in a fractional fashion to more than one machine.

We may now state our rounding procedure. For each (machine, job) pair (i, j) such that $\bar{x}_{ij} = 1$, we assign job j to machine i. We call the schedule of these jobs S_1. For the remaining at most m jobs, we simply construct a matching of the jobs to machines such that each job is matched to a machine it is already partially assigned to. We schedule each job on the machine to which it is matched, and call the schedule of these jobs S_2.

We defer momentarily the question of whether such a matching exists, and analyze the makespan of the resulting schedule, which is at most the sum of the makespans of S_1 and S_2. Since the x_{ij} form a feasible solution to the relaxed linear program, the makespan of S_1 is at most D. Since S_2 schedules at most one job per machine, and assigns j to i only if $x_{ij} > 0$, meaning $p_{ij} < D$, the makespan of S_2 is at most D (this argument is the reason we had to add constraint (13) to our linear program). Thus the overall schedule has length at most $2D$.

The argument that a matching always exists is somewhat complex and can only be sketched here. We create a graph G in which there is one node for each machine and one for each job, and an edge between each machine node i and job node j if $x_{ij} > 0$. We are again helped by the theory of linear programming, as the linear program we solved is a *generalized assignment problem*. As a result, for any basic solution, the structure of G is a forest of trees and 1-*trees*, which are trees with one additional edge added; for further details see [AMO93]. We need not consider jobs that are already integrally assigned, so for every pair (i, j) such that $x_{ij} = 1$, we remove from G the nodes representing machine i, job j and their mutual edge (note that the constraints imply that this machine and job is not connected to any other machine or job). In the forest that remains, the only leaves are machine nodes, since every remaining job node represents a job that is fractionally assigned by the linear program and thus has an edge to at least two machines.

It is now straightforward to find a matching in G. We first consider the 1-trees, and in particular consider the unique cycle in each 1-tree. The nodes in these cycles alternate between machine nodes and job nodes, with an equal number of each. We arbitrarily choose an orientation of the cycle and assign each job to the machine that follows it in the oriented cycle. We then remove all of the matched nodes from G. What remains is a forest of trees; furthermore, it is possible that for each of these trees we have created at most one new leaf that is a job node. We then root each of the trees in the forest, either at its leaf job node, or, if it does not have one, at an arbitrary vertex. Finally, we assign each job node to one of its children machine nodes in the rooted tree. Each machine node has at most one parent, and thus is assigned at most one job. We have thus succesfully matched all job nodes to machine nodes, as we required.

Thus, there exists a 2-relaxed decision procedure for $\mathrm{R}||C_{\max}$, and we have the following theorem.

Theorem 5.1 ([LST90]) *There is a 2-approximation algorithm for* $\mathrm{R}||C_{\max}$.

5.2 Inferring an Ordering from a Preemptive Schedule: $1|r_j|\sum C_j$

In this section and the next we discuss techniques for inferring an ordering of jobs from a relaxation. In this section we consider the problem $1|r_j|\sum C_j$. Recall, as mentioned in Section 2.1, that this problem is $\mathcal{NP}$-hard. However, we can find a good relaxation by the simple expedient of allowing preemption. Specifically, we use $1|r_j, pmtn|\sum C_j$ as a relaxation of $1|r_j|\sum C_j$. $1|r_j, pmtn|\sum C_j$ can be solved without linear programming, simply by using the `SRPT` rule. We will make use of this relaxation by extracting from it the order of completion of the jobs in the optimal preemptive schedule, and create a nonpreemptive schedule with the same order of completion.

Our algorithm, which we call `Convert-Preempt-Schedule`, is as follows. We first obtain an optimal preemptive schedule P for the instance in question. We then order the jobs in their order of completion in P; assume by renumbering that $C_1^P \leq \ldots C_n^P$. We schedule the jobs nonpreemptively by succesively scheduling the next job in the order unless it is not yet released, in which case we wait until its release date and then schedule it.

Theorem 5.2 ([PSW95]) `Convert-Preempt-Schedule` *is a 2-approximation algorithm for* $1|r_j|\sum C_j$.

Proof: The schedule constructed by `Convert-Preempt-Schedule`can be understood as follows.

For each job j, consider the point of completion of the last piece of j scheduled in P, insert p_j extra units of time into the schedule at the completion point of j in P (delaying by an additional p_j time the part of the schedule after C_j^P) and then schedule j nonpreemptively in the newly inserted block of length p_j. Then, remove from the schedule all of the time originally allocated to processing job j. Finally, cut out any idle time in the resulting schedule that can be removed without changing the scheduled order of the jobs or violating a release date constraint. The result is exactly the schedule computed by `Convert-Preempt-Schedule`.

Note that the completion of job j is only delayed by insertion of blocks for jobs that finish earlier in P and hence:

$$C_j^N \leq C_j^P + \sum_{k \leq j} p_k.$$

However, $\sum_{k \leq j} p_k \leq C_j^P$, since all of these jobs completed before j in P, and therefore

$$\sum_{j=1}^{n} C_j^N \leq 2 \sum_{j=1}^{n} C_j^P.$$

The theorem follows from the fact that the average completion time of the optimal preemptive schedule is a lower bound on the average completion time of the optimal nonpreemptive schedule. □

5.3 An Ordering from a Linear Programming Relaxation: $1|r_j, prec|\sum w_j C_j$

In this section we generalize the techniques of the previous section, applying them, not to a preemptive schedule, but instead to a linear programming relaxation of $1|r_j, prec|\sum w_j C_j$.

5.3.1 The Relaxation

We begin by describing the linear programming relaxation of our problem. Unlike our previous relaxation, this one does not arise from relaxing the integrality constraints of an integer linear program. Rather, we give several classes of inequalities that would be satisfied by feasible solutions to $1|r_j, prec|\sum w_j C_j$. These constraints are necessary but not sufficient to describe a valid solution to the problem.

The linear-programming formulation that we considered for $\mathrm{R}||C_{\max}$ assigned jobs to machines but captured no information about the ordering of jobs on a machine. For $1|r_j, prec|\sum w_j C_j$ the ordering of jobs on a machine is a critical element of a high-quality solution, so we seek a formulation that can model this. We do this by making time explicit in the formulation: we will have n variables C_j, one for each of the n jobs; C_j will represent the completion time of job j in a schedule.

Consider the following formulation in these C_j variables, solutions to which correspond to optimal solutions of $1|r_j, prec|\sum w_jC_j$.

$$\text{minimize } \sum_{j=1}^{n} w_jC_j \tag{14}$$

subject to

$$C_j \geq r_j + p_j, \quad j = 1, \ldots, n, \tag{15}$$

$$C_k \geq C_j + p_k, \quad \text{for each pair } j,k \text{ such that } j \prec k, \tag{16}$$

$$C_k \geq C_j + p_k \quad \text{or} \quad C_j \geq C_k + p_j, \quad \text{for each pair } j,k. \tag{17}$$

Unfortunately, the last set of constraints are not linear constraints. Instead, we use a class of valid inequalities, introduced by Wolsey [Wol85] and Queyranne [Que93]. Recall that we denote the entire set of jobs $\{1, \ldots, n\}$ as $\mathcal{J}$, and, for any subset $S \subseteq \mathcal{J}$, we define $p(S) = \sum_{j\in S} p_j$ and $p^2(S) = \sum_{j\in S} p_j^2$. We claim that for any feasible one-machine schedule (independent of constraints and objective)

$$\sum_{j\in S} p_jC_j \geq \frac{1}{2}(p^2(S) + p(S)^2), \quad \text{for each } S \subseteq \mathcal{J}. \tag{18}$$

We show that these inequalities are satisfied by the completion times of any valid schedule for one machine and thus in particular for $1|r_j, prec|\sum w_jC_j$.

Lemma 5.3 *Let $C_1, \ldots, C_n$ be the completion times of jobs in any feasible schedule on one machine. Then the C_j must satisfy the inequalities*

$$\sum_{j\in S} p_jC_j \geq \frac{1}{2}\left(p(S)^2 + p^2(S)\right) \quad \textit{for each } S \subseteq \mathcal{J}. \tag{19}$$

Proof: We assume that the jobs are indexed so that $C_1 \leq \cdots \leq C_n$. Consider first the case $S = \{1, \ldots, n\}$. Clearly for any job j, $C_j \geq \sum_{k=1}^{j} p_k$. Multiplying by p_j and summing over all j, we obtain

$$\sum_{j=1}^{n} p_jC_j \geq \sum_{j=1}^{n} p_j \sum_{k=1}^{j} p_k = \frac{1}{2}\sum_{j=1}^{n}\sum_{k=1}^{n} p_jp_k = \frac{1}{2}\left(p^2(S) + p(S)^2\right)$$

Thus (19) holds for $S = \{1, \ldots, n\}$. The general case follows from the fact that for any other set of jobs S, the jobs in S are feasibly scheduled by the schedule of $\{1, \ldots, n\}$—just ignore the other jobs. So we may view S as our entire set of jobs and apply the previous argument. $\square$

In the special case of $1||\sum w_j C_j$ the constraints (19) give an exact characterization of the problem; specifically, any set of C_j that satisfy these constraints must describe the completion times of a feasible schedule, and thus these linear constraints effectively replace the disjunctive constraints (17). When we extend the formulation to include constraints (15) and (16), we no longer have an exact formulation, but rather a linear-programming relaxation of $1|r_j, prec|\sum w_j C_j$.

We note that this formulation has an exponential number of constraints; however, it can be solved in polynomial time by the use of the ellipsoid algorithm for linear programming [Wol85, Que93]. We also note that in the special case in which we just have release dates, a slightly strengthened version can (remarkably) be solved optimally in $O(n \log n)$ time [Goe96].

5.3.2 Constructing a Schedule from a Solution to the Relaxation

We now show that a solution to this relaxation can be converted efficiently to an approximately-optimal schedule. For simplity, we ignore release dates and consider only $1|prec|\sum w_j C_j$. Our approximation algorithm, which we call SCHEDULE-BY-$\bar{C}_j$, is simple to state. We first solve the linear programming relaxation given by (14), (16) and (18) and call the solution $\bar{C}_1, \ldots, \bar{C}_n$; we assume without loss of generality that $\bar{C}_1 \leq \bar{C}_2 \leq \ldots \bar{C}_n$. We then schedule the jobs in the order $1, \ldots, n$. Since there are no release dates there is no idle time. Note that this ordering of the jobs respects the precedence constraints, because if the $\bar{C}_j$ satisfy (14) then $j \prec k$ implies that $\bar{C}_j < \bar{C}_k$.

To analyze SCHEDULE-BY-$\bar{C}_j$, we begin by understanding why it is not an optimal algorithm. Unfortunately, $\bar{C}_1 \leq \cdots \leq \bar{C}_n$ being a feasible solution to (18) does not guarantee that, in the schedule in which job j is designated to complete at time $\bar{C}_j$, that at most one job is scheduled at any point in time. More formally, the intervals $(C_j - p_j, C_j]$, $j = 1, \ldots, n$, are not constrained to be disjoint. If $\bar{C}_1 \leq \cdots \leq \bar{C}_n$ actually corresponded to a valid schedule, then $\bar{C}_j$ would be no less than $\sum_{k=1}^{j} p_k$ for all j. We will see that, although the formulation does not guarantee this property, it does yield a relaxation of this, which is sufficient for the purposes of approximation.

Theorem 5.4 ([HSSW97]) SCHEDULE-BY-$\bar{C}_j$ *is a 2-approximation algorithm for* $1|prec|\sum w_j C_j$.

Proof: Let $C_1^*, \ldots, C_n^*$ denote the completion times in some optimal schedule, and $\tilde{C}_1, \ldots, \tilde{C}_n$ denote the completion times in the schedule found by SCHEDULE-BY-$\bar{C}_j$.

Since the jobs have been renumbered so that $\bar{C}_1 \leq \cdots \leq \bar{C}_n$, taking $S = \{1, \ldots, j\}$ gives

$$\tilde{C}_j = p(S).$$

We now show that $\bar{C}_j$ is at least $\frac{1}{2}p(S)$. (Again, if the $\bar{C}_j$ were feasible completion times in an actual schedule, we would have $\bar{C}_j \geq p(S)$. This relaxed version of the property is the key to the approximation.)

We use inequality (18) for $S = \{1, 2, \ldots, j\}$.

$$\sum_{k=1}^{j} p_k \bar{C}_k \geq \frac{1}{2}(p^2(S) + p(S)^2) \geq \frac{1}{2}p(S)^2. \tag{20}$$

Since $\bar{C}_k \leq \bar{C}_j$, for each $k = 1, \ldots, j$, we have

$$\bar{C}_j p(S) = \bar{C}_j \sum_{k=1}^{j} p_k \geq \sum_{k=1}^{j} \bar{C}_k p_k \geq \frac{1}{2}p(S)^2.$$

Dividing by $p(S)$, we obtain that $\bar{C}_j$ is at least $\frac{1}{2}p(S)$. We therefore obtain $\tilde{C}_j \leq 2\bar{C}_j$, and since $\sum_j w_j \bar{C}_j \leq \sum_j w_j C_j^*$, the result follows. □

6 Polynomial approximation schemes

For certain $\mathcal{NP}$-hard scheduling problems there is a limit to our ability to approximate them in polynomial time; for example, Lenstra, Shmoys and Tardos proved that there is no ρ-approximation algorithm, with $\rho < 3/2$, for $R||C_{\max}$ unless $\mathcal{P} = \mathcal{NP}$ [LST90]. For certain problems, however, we can approximate their optimal solutions arbitrarily closely in polynomial time. In this section we present three *polynomial approximation schemes*; that is, polynomial time algorithms that, for any constant $\rho > 1$, deliver a solution whose objective value is at most ρ times optimal. The running time will depend on ρ – the smaller ρ is, the slower the algorithm will be.

We will present two approaches to the design of such algorithms. The first approach is based on rounding processing times or weights to small integers so that we can apply pseudopolynomial-time algorithms such as that for $1||\sum w_j U_j$. A second approach is based on identifying the "important" jobs that have the greatest impact on the solution, and processing them separately. In one version, illustrated for $\mathrm{P}||C_{\max}$, we round the large jobs so that there are only a constant number of large job sizes, schedule them using dynamic programming, and then schedule the small jobs arbitrarily. In a second version, illustrated for $1|r_j|L_{\max}$, we enumerate all possible schedules for the large jobs, and then fill in the small jobs around them.

6.1 From Pseudopolynomial to PTAS: $1||\sum w_j U_j$

In Section 3, we gave an $O(n \sum w_j)$ time algorithm for $1||\sum w_j U_j$. Note that although at optimality minimizing $\sum w_j U_j$ is equivalent to maximizing the total weight of on-time jobs, a ρ-approximation algorithm for one metric is *not* a ρ-approximation algorithm for the other metric. In this section, we explicitly focus on the optimality criterion $\sum w_j U_j$.

Since we already have an algorithm that runs in polynomial time when then weights are polynomial in n, a natural idea is to try to reduce any instance to such a special case. We will scale the weights so that the optimal solution is bounded by a polynomial in n; this will allow us to apply our dynamic programming algorithm to weights of polynomial size.

Suppose for now that we know W^*, the value of $\sum w_j U_j$ in the optimal schedule. Multiply every weight by $n/(\epsilon W^*)$; now the optimal $\sum w_j U_j$ becomes n/ϵ. Clearly, a schedule with $\sum w_j U_j$ within $(1+\epsilon)$ of optimum under these weights is also within $(1+\epsilon)$ of optimum under the original weights. Thus, it suffices to find a schedule with $\sum w_j U_j$ at most $(1+\epsilon)n/\epsilon = n/\epsilon + n$ under the new weights.

To do so, increase the weight of every job to the next larger integer. This increases the weight of each job by at most 1, and thus for any schedule, increases $\sum w_j U_j$ by at most n. Under these new weights, $\sum w_j U_j$ for the original optimal schedule is now at most $n/\epsilon + n$, so the optimal schedule under these integral weights has $\sum w_j U_j$ at most $n/\epsilon + n$. Since all weights are integers, we can apply the dynamic programming algorithm of Section 3 to find an optimal schedule for the rounded instance. Since we only rounded up, the same schedule under the original weights can only have a smaller $\sum w_j U_j$. Thus, we find a schedule with weight at most $n/\epsilon + n$ in the (scaled) original weights, i.e. a $(1+\epsilon)$ times optimum schedule.

The running time of our dynamic program is proportional to n times the sum of the (new) weights. This might be a problem, since the weights can be arbitrarily large. However, any job with new weight exceeding $n/\epsilon + n$ *must* be scheduled before its deadline. We therefore identify all such jobs, and modify our dynamic program: T_{wj} becomes the minimum time needed to complete all these jobs that must complete by their deadlines plus other jobs from $1, \ldots, j$ of total weight w. The dynamic programming argument goes through unchanged, but now we consider only jobs of weight $O(n/\epsilon)$. It follows that the largest value of w that we consider is $O(n^2/\epsilon)$, which means that the total running time is $O(n^3/\epsilon)$.

It remains to determine W^*. We clearly do not expect to determine it exactly, but note that an estimate will suffice. If we find a value W' with $W^*/n \leq W' \leq W^*$, the analysis above will go through with the running time increased by a factor of n. To compute W', we will solve the following problem: find a schedule that minimizes the weight of the maximum weight late job. The value of this schedule, W', is clearly a lower bound on W^*, as all schedules that minimize $\sum w_j U_j$ must have a late job of weight at least W'. Further, W^* is at most nW', since the schedule returned must have at most n late jobs each of which has weight at most W'. Hence our value W' is within a factor of n of optimal.

To compute W', we formulate a $1||f_{\max}$ problem. For each job j,

$$f_j(C_j) = \begin{cases} w_j & \text{if } C_j > d_j \\ 0 & \text{if } C_j \leq d_j \end{cases}$$

This will compute the schedule that minimizes the weight of the maximum weight late job. By the results of Section 2.1, we know we can compute this exactly in polynomial time.

Theorem 6.1 *There exists a $O(n^4/\epsilon)$-time $(1+\epsilon)$-approximation algorithm for $1||\sum w_j U_j$*

6.2 Rounding and Dynamic Programming: $\mathrm{P}||C_{\max}$

We now turn to the problem of $\mathrm{P}||C_{\max}$. Recall that in Lemma 3.3 we solved, in polynomial time, the special case when there are a constant number of different job sizes. We use this algorithm to handle the jobs that "matter" in an arbitrary instance. We will round and scale the input so that there is at most a constant number of sizes of big jobs, apply the dynamic programming algorithm of Section 3 to this special case, and then schedule the small jobs arbitrarily. By the definition of big and small, the overall contribution of the small jobs to the makespan will be negligible.

We will give a $(1+\epsilon)$-RDP for this problem that can be transformed as before into a $(1+\epsilon)$-approximation algorithm. We therefore assume that we have a target optimum schedule length T, We also assume for the rest of this section that $\epsilon T, \epsilon^2 T, \epsilon^{-1}$ and ϵ^{-2} are integers. The proofs can easily be modified to handle the case of arbitrary rational numbers.

We first show how to handle the large jobs.

Lemma 6.2 *Let I be an instance of $\mathrm{P}||C_{\max}$, let T be a target schedule length, and $\epsilon > 0$. Assume that $T \geq \max_j p_j$ and all $p_j \geq \epsilon T$. Then, for this case, there is a $(1+\epsilon)$-RDP for $\mathrm{P}||C_{\max}$.*

Proof: Form instance I' from I with processing times p'_j by rounding each p_j down to an integral multiple of $\epsilon^2 T$. This creates an instance in which:

1. $0 \leq p_j - p'_j \leq \epsilon^2 T$
2. there are at most $\frac{T}{\epsilon^2 T} \leq \frac{1}{\epsilon^2}$ different job sizes,
3. in any feasible schedule, each machine has at most $\frac{T}{\epsilon T} = \frac{1}{\epsilon}$ jobs.

Thus, we can apply Lemma 3.3 to instance I' and obtain an optimal solution to this scheduling problem; let its makespan be D. If $D > T$, then we know that there is no schedule of length $\leq T$ for I, since job sizes in I' are no greater than those in I. In this case we can answer "no schedule of length $\leq T$ exists". If $D \leq T$, then we will answer "there exists a schedule of length $\leq (1+\epsilon)T$. We now show that this answer will be correct. We simply take our schedule for I' and replace the rounded jobs with the original jobs from I. By (1) and (3) above, we add at most $\epsilon^2 T$ to the processing time of each job, and since there are at most $\frac{1}{\epsilon}$ jobs per machine, we add at most ϵT to the processing time per machine. Thus we can create a schedule with makespan at most $T + \epsilon T = (1+\epsilon)T$. $\square$

We now give the complete algorithm. The idea will be to remove the "small" jobs, use Lemma 6.2 to schedule the remaining jobs, and then add the small jobs back greedily. Given input I_0, target schedule length T, and $\rho = 1 + \epsilon > 1$, we execute the following algorithm.

Let R be the set of jobs with $p_j \leq \epsilon T$. Let $I = I_0 - R$
Apply Lemma 6.2 to I, T, and ρ.
If this algorithm returns no,
(†) then output "no schedule of length $\leq T$ exists".
    else
        for each job j in R
            if there is a machine i with load $\leq T$,
                then add job j to machine i
(*)   else return "no schedule of length $\leq T$ exists"
        return " yes a schedule of length $\leq \rho T$ exists"

Theorem 6.3 *The algorithm above is a ρ-relaxed decision procedure for* P$||C_{\max}$.

Proof: If the algorithm outputs "yes a schedule of length $\leq \rho T$ exists', then it has constructed such a schedule, and is clearly correct. If the algorithm outputs "no schedule of length $\leq T$ exists" on line (†), then it is because no schedule of length T exists for instance I, and the output is correct. But instance I is a subset of the original jobs and so if no schedule exists for I, then no schedule exists for I_0. If the algorithm outputs "no schedule of length $\leq T$ exists" on line (*), then at this point in the algorithm, every machine must have more than T units of processing on it. Thus, we have that $\sum_j p_j > mT$, which means that no schedule of length $\leq T$ exists. □

The running time is dominated by the dynamic programming in Lemma 3.3. It is polynomial in n, but the exponent is a polynomial in $1/\epsilon$. While for ρ very close to 1, the running time is prohibitively large, for larger, fixed values of ρ, a modified algorithm yields good schedules with near-linear running times; see [HS87] for details.

6.3 $1|r_j|L_{\max}$

We now turn to the problem of minimizing the maximum lateness in the presence of release dates. Recall from Section 2.1 that without release dates `EDD` is an exact algorithm for this problem. Once we add release dates the problem becomes $\mathcal{NP}$-hard. As we think about approximation algorithms, we come upon an immediate obstacle, namely that the objective function can be 0 or even negative, and hence a solution of value at most $\rho C^*_{\max}$ is clearly impossible. In order to get around this, we must guarantee that the objective value is positive. One simple way to do so is to decrease all the d_j's uniformly by some value δ. This decreases the objective value by exactly δ and does not change the structure of the optimal solution. In particular, if we pick δ large enough so that all the d_j's are negative, we are guaranteed that $L_{\max}$ is positive.

Alternatively, we can introduce an equivalent problem, which we call the *delivery time* problem. Each job, in addition to having a release date r_j and a processing time p_j has a *delivery time* q_j. A delivery time is an amount of time that must elapse between the completion time of a job and when it is finished. Our objective is now to minimize $\max_j\{C_j + q_j\}$. We see that by setting $q_j = -d_j$ (recall that d_j is now negative, and so q_j is positive), this problem is equivalent to minimizing maximum lateness, and in fact we will overload L_j and define it as $C_j + q_j$. We admit that forcing d_j to be negative is somewhat artificial and so we do not concentrate on this interpretation. We use instead the delivery time formulation, which in addition to modeling a number of applications, is a key subroutine in computational approaches to shop scheduling problems [LLKS93].

6.3.1 Jackson's rule is a 2 approximation

In the delivery-time model, EDD translates to Longest Delivery Time First. This is often referred to as *Jackson's rule*. [Jac55]. Let $L^*_{\max}$ be the optimum lateness. The following two lower bounds for this problem are easily derived:

$$L^*_{\max} \geq \sum_j p_j, \tag{21}$$

$$L^*_{\max} \geq r_j + p_j + q_j \text{ for all } j. \tag{22}$$

Lemma 6.4 *Jackson's Rule is a 2-approximation algorithm for the delivery time version of* $1|r_j|L_{\max}$.

Proof: Let c be a job for which $L_c = L_{\max}$. Since Jackson's rule creates a schedule with no unforced idle time, we know that there is no idle time between time r_c and C_c. Let J' be the set of jobs that run between r_c and C_c. Then

$$L_c = C_c + q_c \tag{23}$$

$$= r_c + \sum_{j \in J'} p_j + q_c \tag{24}$$

$$\leq \quad (r_c + q_c) + \sum_j p_j \tag{25}$$

$$= \quad 2L^*_{\max}, \tag{26}$$

where the last line follows by applying the two lower bounds (21) and (22). □

6.3.2 Enumeration of large jobs: $1|r_j|L_{\max}$

The presentation of this section follows that of Hall [Hal97]. The original approximation scheme for this problem is due to Hall and Shmoys [HS89].

To obtain better bounds, we need to look more carefully at when Jackson's rule can go wrong. Let s_j be the starting time of job j, $r_{\min}(S) = \min_{j \in S} r_j$, $q_{\min}(S) = \min_{j \in S} q_j$, recall that $p(S) = \sum_{j \in S} p_j$. Then clearly, for any $S \subseteq J$

$$L^*_{\max} \geq r_{\min}(S) + p(S) + q_{\min}(S). \tag{27}$$

Now consider a job c for which $L_c = L_{\max}$. Let t_i be the latest time before s_c at which the machine is idle, and let a be the job that runs immediately after this idle time. Let S be the set of jobs that run between s_a and C_c. We call S a *critical section.* Because of the idle time immediately before s_a, we know that for all $j \in S$, $r_j \geq r_a$. In other words we have a set of jobs, all of which were released after time r_a, and which end with the job that achieves $L_{\max}$. Now if for all $j \in S$ $q_j \geq q_c$, then we claim that $\mathrm{L}_c = L^*_{\max}$. This follows from the fact that

$$L_{\max} = L_c = r_a + p(S) + q_c = r_{\min}(S) + p(S) + q_{\min}(S),$$

and that the right hand side, by (27), is also a lower bound on $L^*_{\max}$. So, as long as, in a critical section, the job with the shortest delivery time is last, we have an optimal schedule. Thus, if Jackson's rule is not optimal, there must be a job b in the critical section which has $q_b < q_c$. We call the latest-schedule job in the critical section with $q_b < q_c$ an *interference job.* The following lemma shows the relationship between the interference job and its effect on $L_{\max}$.

Lemma 6.5 *Let b be an interference job in a schedule created by Jackson's rule. Then $L_{\max} < L^*_{\max} + p_b$.*

Thus, if interference jobs have small processing times, Jackson's rule does very well. To make sure that this is the case, we will handle the large jobs separately, to ensure that they are not interference jobs, and then use Jackson's rule on the remaining jobs.

Let us assume for now that we know the optimal schedule for instance I. Let s_j^* be the starting time of job j in the optimal schedule, and let $\delta > 0$ be a parameter to be chosen later. Partition the jobs into small jobs $S = \{j : p_j < \delta\}$ and big jobs $B = \{j : p_j \geq \delta\}$. We create instance $\tilde{I}$ as follows: if $j \in S$, then $\tilde{r}_j = r_j$, $\tilde{p}_j = p_j$, and $\tilde{q}_j = q_j$, otherwise, $\tilde{r}_j = s_j^*$, $\tilde{p}_j = p_j$, and $\tilde{q}_j = L^*_{\max}(I) - p_j - s_j^*$. Instance $\tilde{I}$ is no easier than instance I, since we have not decreased any release dates or delivery times. Yet, the optimal schedule for I remains an optimal schedule for $\tilde{I}$, by construction. In $\tilde{I}$ we have given the large jobs a release date equal to their optimal starting time, and a delivery time that is equal to the schedule length minus their completion time, and hence have constrained the large jobs to run exactly when they would run in the optimal schedule for instance I. Thus, in an optimal schedule for $\tilde{I}$, the big jobs run at exactly time $\tilde{r}_j$ and have $L_j = \tilde{r}_j + \tilde{p}_j + \tilde{q}_j = L^*_{\max}$.

Now we claim that if we run Jackson's rule on $\tilde{I}$, the big jobs will not be interference jobs.

Lemma 6.6 *If we run Jackson's rule on $\tilde{I}$, no job $b \in B$ will be an interference job.*

Proof: Assume that some job $b \in B$ is an interference job. As above, define the critical section, and jobs a and c. Since b is an interference job, we know that $\tilde{q}_c > \tilde{q}_b$ and $\tilde{r}_c > \tilde{r}_b$. We also know that $\tilde{r}_b = s_b^*$, and so c must run after b in the optimal schedule for I. Applying (27) to the set consisting of jobs b and c, we get that

$$L^*_{\max} \geq \tilde{r}_b + \tilde{p}_b + \tilde{p}_c + \tilde{q}_c \geq r_b + p_b + p_c + \tilde{q}_b = L^*_{\max} + p_c$$

which is a contradiction. $\Box$

So if we run Jackson's rule on $\tilde{I}$, we get a schedule whose length is at most $L^*_{\max}(I) + \delta$. Choosing $\delta = \epsilon \sum_j p_j$, and recalling that $L^*_{\max} \geq \sum p_j$, we get a schedule of length at most $(1+\epsilon)L^*_{\max}$. Further, there can be at most $\sum_j p_j / (\epsilon \sum_j p_j) = 1/\epsilon$ big jobs. The only problem is that we don't know $\tilde{I}$.

We now argue that it is not necessary to know $\tilde{I}$. First, observe that the set of big jobs is purely a function of the input, and ϵ. Now, if we knew the starting times of the big jobs in the optimal schedule for I , we would know $\tilde{I}$, and could run Jackson's rule on the job in S, inserting the big jobs at the appropriate time. This implies a numbering of the big jobs, i.e. each big job j_i is, for some k, the kth job in the schedule for $\tilde{I}$. Thus, we really only need to know k, and not the starting time for job j_i. Thus we just enumerate all possible numberings for the big jobs. There are $n^{1/\epsilon}$ such numberings. Given a numbering, we can run Jackson's rule on the small jobs, and insert the big jobs at the appropriate places in $O(n \log n)$ time, and thus we get an algorithm that in $O(n^{1+1/\epsilon} \log n)$ time finds a schedule with $L_{\max} \le (1+\epsilon)L^*_{\max}$.

7 Research Issues and Summary

In this chapter we have surveyed some of the basic techniques for deterministic scheduling. Scheduling is an old and therefore mature field, but important opportunities for research contributions remain. In addition to some of the outstanding open questions (see the survey by Lawler et al. [LLKS93]) it is our feeling that the most meaningful research contributions will be either new and innovative techniques for attacking old problems or new problem definitions that model more realistic applications.

There are other schools of approach to the design of algorithms for scheduling, such as those relying on techniques from artificial intelligence or from computational optimization. It will be quite valuable to forge stronger connections between these different approaches to solving scheduling problems.

8 Defining Terms

See the Introduction (Section 1).

Acknowledgements

We are greatful to Jan Karel Lenstra and David Shmoys for helpful comments.

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows : Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[Bak74] K. R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, 1974.

[BCS74] J.L. Bruno, E.G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17:382–387, 1974.

[BF82] I. Bárány and T. Fiala. Többgépes ütemezési problémák közel optimális megoldása. *Szigma–Mat.–Közgazdasági Folyóirat*, 15:177–191, 1982.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[CMM67] R.W. Conway, W.L. Maxwell, and L.W. Miller. *Theory of Scheduling*. Addison-Wesley, 1967.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[GLLK79] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[Goe96] M. Goemans. A supermodular relaxation for scheduling with release dates. In *Proceedings of the 5th Conference on Integer Programming and Combinatorial Optimization*, pages 288–300, June 1996. Published as Lecture Notes in Computer Science 1084, Springer-Verlag.

[Gra66] R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[Gra69] R.L. Graham. Bounds on multiprocessing anomalies. *SIAM Journal of Applied Mathematics*, 17:263–269, 1969.

[GS76] T. Gonzalez and S. Sahni. Open shop scheduling to minimize finish time. *Journal of the ACM*, 23:665–679, 1976.

[Hal97] L. A. Hall. *Approximation Algorithms for NP-hard problems*, chapter 1. PWS Publishing Company, 1997. D. Hochbaum, editor.

[HLvdV97] J.A. Hoogeveen, J.K. Lenstra, and S.L. van de Velde. Sequencing and scheduling. In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimization*. Wiley, Chichester, 1997. To appear.

[Hoc97] Dorit Hochbaum, editor. *Approximation Algorithms.* PWS, 1997.

[Hor73] W. Horn. Minimizing average flow time with parallel machines. *Operations Research*, 21:846–847, 1973.

[Hor74] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.

[HS87] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[HS89] L. Hall and D. B. Shmoys. Approximation schemes for constrained scheduling problems. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 134–141. IEEE, October 1989.

[HSSW97] L. A. Hall, A.S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line algorithms. To appear in Math of OR, 1997.

[Jac55] J.R̃. Jackson. Scheduling a production line to minimize maximum tardiness. Management Science Research Project Research Report 43, University of California, Los Angeles, 1955.

[Joh54] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, pages 61–68, 1954.

[Kha79] L./ G./ Khachiyan. A polynomial algorithm in linear programming (in russian). *Doklady Adademiia Nauk SSSR*, 224:1093–1096, 1979.

[Law73] E.L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19:544–546, 1973.

[Law76] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[LL78] E.L. Lawler and J. Labetoulle. On preeemptive scheduling of of unrelated parallel processors by linear programming. *Journal of the ACM*, 25:612–619, 1978.

[LLKS93] E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, and D.B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S.C. Graves, A.H.G. Rinnooy Kan, and P.H. Zipkin, editors, *Handbooks in Operations Research and Management Science, Vol 4., Logistics of Production and Inventory*, pages 445–522. North-Holland, 1993.

[LM69] E.L. Lawler and J. M. Moore. A functional equation and its application to resource allocation and sequencing problems. *Management Science*, pages 77–84, 1969.

[LST90] J.K. Lenstra, D.B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[McN59] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6:1–12, 1959.

[Moo68] J. M. Moore. An n-job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15:102–109, 1968.

[Pin95] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 1995.

[PSW95] C. Phillips, C. Stein, and J. Wein. Minimizing average completion time in the presence of release dates. To appear in Math Programming, 1995.

[QS94] M. Queyranne and A.S. Schulz. Polyhedral approaches to machine scheduling. Technical Report Technical Report 474/1995, Technical University of Berlin, 1994.

[Que93] M. Queyranne. Structure of a simple scheduling polyhedron. *Mathematical Programming*, 58:263–285, 1993.

[Sga97] J. Sgall. On-line scheduling – a survey. In A. Fiat and G. Woeginger, editors, *On-Line Algorithms, Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1997. To appear.

[Smi56] W.E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[Ull75] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[Wol85] L.A. Wolsey. Mixed integer programming formulations for production planning and scheduling problems. Invited talk at the 12th International Symposium on Mathematical Programming, MIT, Cambridge, 1985.

For Further Information

We conclude by reminding the reader what this chapter is not. In no way is this chapter a comprehensive survey of even the most basic and classical results in scheduling theory, and it is certainly not an up-to-date survey on the field. It also essentially entirely ignores "non-traditional" models, and does not touch on stochastic scheduling or on any of the other approaches to scheduling and resource allocation. The reader interested in a comprehensive survey of the field should consult the textbook by Pinedo [Pin95] and the survey by Lawler et. al. [LLKS93]. These sources provide pointers to a number of other references. In addition, we also recommend an annotated bibliography by Hoogeveen et. al. that contains information on recent results in scheduling theory [HLvdV97], the surveys by Queyranne and Schulz on polyhedral formulations[QS94], by Hall on approximation algorithms [Hal97],and by Sgall on online scheduling [Sga97]. Research on deterministic scheduling theory is published in many journals; for example see *Mathematics of Operations Research*, *Operations Research*, *SIAM Journal on Computing*, and *Journal of the ACM*.