

11

Approximation Algorithms for Minimizing Average Weighted Completion Time

Chandra Chekuri

Bell Labs

Sanjeev Khanna

University of Pennsylvania

11.1	Introduction	11-1
11.2	Algorithms for Problems with Precedence Constraints	11-3
	LP-Based Algorithms • Combinatorial Algorithms	
11.3	Unrelated Machines	11-16
11.4	Approximation Schemes for Problems with Release Dates	11-19
	Input Transformations • Dynamic Programming • Nonpreemptive Scheduling on Parallel Machines • Preemptive Scheduling on Parallel Machines	
11.5	Conclusions and Open Problems	11-26

11.1 Introduction

In this chapter we survey approximation algorithms for scheduling to minimize average (weighted) completion time (equivalently sum of (weighted) completion times). We are given n jobs J_1, \dots, J_n , where each job J_j has a positive weight w_j . We denote by C_j the completion time of job j in a given schedule. The objective in the problems we consider is to find a schedule to minimize $\sum_j w_j C_j$ (average weighted completion time). The most basic problem in this context is to minimize $\sum_j C_j$ on a single machine with job j having a processing time p_j , and all jobs are available at time 0, in other words, the problem $1 \parallel \sum_j C_j$. It is easy to see that ordering jobs by the SPT rule (shortest processing time first) gives an optimal schedule. A slight generalization with jobs having weights, $1 \parallel \sum_j w_j C_j$, also has a simple optimality rule first stated by Smith [1] (known as Smith's rule): schedule jobs in nondecreasing order of the ratio p_j/w_j .

Earlier work on minimizing weighted completion times focussed on identifying polynomial time solvable cases by generalizing Smith's rule. Algorithms were found for solving the problem on a single machine when jobs are allowed to have some restricted class of precedence constraints such as in- and out-tree precedences and series-parallel precedence constraints [2–4]. Sydney [4] unified these results by showing the applicability of Smith's rule to *decomposable* classes of precedence constraints. We are interested in a more

general setting with release dates, arbitrary precedence constraints, and multiple machines, any of which make the problem NP-hard [5]. Thus, we will consider approximation algorithms. Online algorithms for some of these problems will be considered elsewhere in the book.

The problem of minimizing makespan ($\max_j C_j$) is an extremely well studied measure in scheduling and starting with Graham's seminal paper on multiprocessor scheduling [6], approximation algorithms have been studied for several variants. However, even though weighted completion time is closely related to makespan (generalizes makespan when precedence constraints are present), it is only recently that approximation algorithms have been designed for this objective. The work of Phillips, Stein, and Wein [7] was the first to explore weighted completion time in detail from the approximation algorithms point of view, in particular for problems with release dates. A variety of algorithms and techniques were developed in the same work; in particular simple ordering rules based on preemptive schedules and LP relaxations were shown to be effective in obtaining near-optimal schedules. Following [7], scheduling to minimize weighted completion time received substantial attention. Polyhedral formulations for single machine problems have been an active area of research and a variety of results are known from the 80s and early 90s. Building on some of the ideas in [7] and insight from polyhedral formulations, Hall, Shmoys, and Wein [8], and independently Schulz [9], developed the first constant factor approximation algorithms for minimizing weighted completion time with precedence constraints. Subsequently, many improvements, generalizations, and combinatorial methods were developed in a number of papers [10–21]. Hoogeveen, Schuurman, and Woeginger [22] systematically studied in-approximability results and established APX-hardness for a number of problems. More recently, polynomial time approximation schemes (PTASes) were obtained for several variants [20,23] based on enumeration and dynamic programming based ideas. Together, these results give us a fairly comprehensive picture of the approximability of scheduling to minimize weighted completion time. Figure 11.1 tabulates the known results and in Section 11.5 we mention several as yet unresolved questions.

This survey aims to describe in detail selected algorithms that illuminate some of the useful and interesting ideas for scheduling to minimize weighted completion time. It is infeasible to do justice to all the known literature given the nature of the survey and the space constraints. We hope that we have captured several central ideas and algorithms that would enable the reader to get both an overview of the known results and some of the more advanced technical ideas. In Section 11.5 we give pointers to material that we would have liked to include but could not.

The rest of the survey is organized as follows. We have broadly partitioned the algorithms into three categories. In Section 11.2 we describe approximation algorithms when jobs have precedence constraints.

Problem	Approx. Ratio	Ref.	Inapproximability
$1 \sum_j w_j C_j$	1	[1]	
$1 r_j, pmtn \sum_j C_j$	1		
$P pmtn \sum_j C_j$	1	[25, 26]	
$P r_j \sum_j w_j C_j$	PTAS	[20]	NP-hard
$Q r_j \sum_j w_j C_j$	PTAS	[27]	NP-hard
$Rm r_j \sum_j w_j C_j$	PTAS	[20]	NP-hard
$R \sum_j w_j C_j$	3/2	[28]	APX-hard [22]
$R r_j \sum_j w_j C_j$	2	[28]	APX-hard [22]
$1 prec, spl \sum_j w_j C_j$	1	[3]	
$1 prec \sum_j w_j C_j$	2	[29]	NP-hard
$1 prec, r_j \sum_j w_j C_j$	$(\epsilon + \epsilon)$	[30]	NP-hard
$P prec, r_j \sum_j w_j C_j$	4	[16]	4/3 [31]
$Q prec, r_j \sum_j w_j C_j$	$O(\log m)$	[15]	4/3 [31]
$O r_j \sum_j w_j C_j$	5.83	[21]	APX-hard [22]
$J r_{hj} \sum_j w_j C_j$	$O\left(\left(\frac{\log m_{\mu}}{\log \log m_{\mu}}\right)^2\right)$	[21]	APX-hard [22]

FIGURE 11.1 Complexity of minimizing average completion time.

Linear programming formulations have played an important role in algorithms for this case and we describe several formulations and how they have been used to develop constant factor approximation algorithms. We also show how simpler combinatorial methods can be used to obtain similar ratios. The latter algorithms have the advantage of being more efficient than the LP based methods as well as providing useful structural insight into the problem. In Section 11.3 we describe an algorithm for unrelated machines. The algorithm illustrates the need for a time indexed formulation as well as the usefulness of randomized rounding. Finally, in Section 11.4 we describe a framework that has led to polynomial time approximation schemes for a variety of problems with release dates. In particular we present a PTAS for scheduling on identical parallel machines. Section 11.5 concludes with discussion, pointers to related work, and open problems that remain.

Applications and Motivation: Makespan of a schedule measures the time by which all the jobs in the schedule finish. However, when jobs are independent and competing for the same resource, a more natural measure of performance is the *average* completion time of jobs. This motivated early work for this measure including the work of Smith [1]. More recently, an application to instruction scheduling in VLIW processors is presented in the work of Chekuri et al. [24]. In this application the weight of a job (instruction) is derived from profile information and indicates the likelihood of the program block terminating at that job. This is a compelling application for scheduling jobs with precedence constraints on a complex machine environment to minimize sum of weighted completion times.

In addition to applications, minimizing weighted completion time presents interesting theoretical problems in finding orderings of posets and has spurred work on polyhedral formulations as well as combinatorial methods. There is an intimate relation between minimizing makespan and minimizing weighted completion time and exploring this has improved our understanding of both.

Techniques: The algorithms we present, except for the approximation schemes in Section 11.4 have an underlying theme. They obtain an ordering of the jobs in one form or the other: via LP relaxations or preemptive schedules or based on decomposition methods. The ordering is converted into a feasible schedule by one of several variants of list scheduling, sometimes based on randomization. Typically, the bounds are shown by a job by job analysis. Although the above paradigm broadly unifies many of the algorithms, subtle and nontrivial problem specific ideas are necessary to obtain the best results. Finally, approximation schemes require a different approach, that of dynamic programming based enumeration over a structured space of schedules.

11.2 Algorithms for Problems with Precedence Constraints

In this section we describe algorithms when jobs have precedence constraints. We note that in this case minimizing makespan is a special case of minimizing weighted completion time. Add a dummy job that is preceded by all the other jobs, the dummy job has weight 1 and the rest have weight 0. The first constant factor approximation algorithms for minimizing completion time when jobs have precedence constraints were obtained via linear programming relaxations. For single machine and identical parallel machines, combinatorial algorithms have been developed although the ratios obtained are some what worse than the ones obtainable by LP methods. However, the combinatorial methods, in addition to yielding efficient algorithms, have provided structural insight, have been useful in extending algorithms to some stochastic settings [32], and in obtaining improved algorithms for special classes of precedence constraints [33,34]. However, for complex machine environments (related or unrelated machines) the only nontrivial algorithms we know are based on LP methods.

11.2.1 LP-Based Algorithms

A variety of LP formulations have been explored for minimizing weighted completion time, in particular for the single machine problem. We mention some of the work here: Balas [35], Wolsey [36], Dyer and Wolsey [37], Queyranne [38], Queyranne and Wang [39], Lasserre and Queyranne [40], Sousa and Wolsey [41],

von Arnim and Schulz [42], Crama and Spieksma [43], Van den Akker, Van Hoesel, and Savelsbergh [44], Van den Akker [45], Van den Akker, Hurkens, and Savelsbergh [46], von Arnim, Schrader, and Wang [47]. The goal of this line of work was to examine the strength of various formulations in obtaining exact solutions via branch and bound and branch and cut methods. More recently these formulations have been used to obtain provably good approximation algorithms, starting with the work of Phillips, Stein, and Wein [7] and subsequently Hall, Shmoys, and Wein [8] and Schulz [9], and many others. Below we describe algorithms for the single machine and identical parallel machines.

11.2.1.1 Single Machine Scheduling

We start by describing and analyzing the completion time formulation for the problem $1 | r_j, prec | \sum_j w_j C_j$. We then describe two other formulations, the time indexed formulation and the linear ordering formulation.

Completion Time Formulation: The completion time formulation is based on the work of Queyranne. The formulation has variables $C_j, j = 1, \dots, n$. C_j indicates the completion time of job j in the schedule. The formulation is the following.

$$\min \sum_{j=1}^n w_j C_j$$

subject to

$$C_j \geq r_j + p_j \quad j = 1, \dots, n \quad (11.1)$$

$$C_k \geq C_j + p_k \quad \text{if } j < k \quad (11.2)$$

$$\sum_{j \in S} p_j C_j \geq \frac{1}{2} \left[\left(\sum_{j \in S} p_j \right)^2 + \sum_{j \in S} p_j^2 \right] \quad S \subseteq N \quad (11.3)$$

The nontrivial constraint in the above formulation is Equation (11.3). We justify the constraint as follows. Consider any valid schedule for the jobs. Without loss of generality, for ease of notation, assume that the jobs in S are $\{1, 2, \dots, |S|\}$ and that they are scheduled in this order in the schedule. Then it follows that for $j \in S$, $C_j \geq \sum_{k \leq j} p_k$, hence $p_j C_j \geq p_j \sum_{k \leq j} p_k$. Summing over all $j \in S$ and simple algebra results in the inequality (11.3). It is easy to see that the inequality is agnostic to the ordering of S in the schedule and hence holds for all orderings. Note that Equation (11.3) generates an exponential number of constraints. Queyranne [38] has shown a polynomial time separation oracle for these constraints and hence the above formulation can be solved in polynomial time by the ellipsoid method.

We now state an important lemma regarding the formulation.

Lemma 11.1

Let \tilde{C} be a feasible solution to the completion time formulation and without loss of generality let $\tilde{C}_1 \leq \tilde{C}_2 \leq \dots \leq \tilde{C}_n$. Then, the following is true for $j = 1, \dots, n$.

$$\tilde{C}_j \geq \frac{1}{2} \sum_{k=1}^j p_k$$

Proof

We have that $\bar{C}_k \leq \bar{C}_j$ for $k = 1, \dots, j$, hence $\sum_{k=1}^j p_k \bar{C}_k \leq (\sum_{k=1}^j p_k) \bar{C}_j$. Consider the set $S = \{1, 2, \dots, j\}$ in Equation (11.3). It follows that

$$\begin{aligned} \left(\sum_{k=1}^j p_k \right) \bar{C}_j &\geq \sum_{k=1}^j p_k \bar{C}_k \\ &\geq \frac{1}{2} \left[\left(\sum_{k=1}^j p_k \right)^2 + \sum_{k=1}^j p_k^2 \right] \\ &\geq \frac{1}{2} \left(\sum_{k=1}^j p_k \right)^2 \end{aligned}$$

The lemma follows. \square

We now analyze the performance of the algorithm Schedule-by- \bar{C} which obtains a valid schedule as follows. First, the above LP is solved to obtain an optimal solution \bar{C} . Without loss of generality assume that $\bar{C}_1 \leq \bar{C}_2 \leq \dots \leq \bar{C}_n$. The algorithm then schedules jobs in order of nondecreasing \bar{C}_j inserting idle time as necessary if the release date r_j is greater than the completion time of the $(j - 1)$ th job.

Lemma 11.2

Let \tilde{C}_j denote the completion time of j in the schedule produced by Schedule-by- \bar{C} . Then, for $j = 1, \dots, n$,

$$\tilde{C}_j \leq \max_{k=1}^j r_k + 2\bar{C}_j.$$

Hence $\tilde{C}_j \leq 3\bar{C}_j$.

Proof

Fix a job j and let $S = \{1, 2, \dots, j\}$. In the schedule produced by Schedule-by- \bar{C}_j , it is clear that there is no idle time in the interval $[\max_{k=1}^j r_k, \tilde{C}_j]$ since all jobs in S are released by $\max_{k=1}^j r_k$. From the ordering rule of the algorithm it follows that $\tilde{C}_j \leq \max_{k=1}^j r_k + p(S)$, and using Lemma 11.1, we conclude that $\tilde{C}_j \leq \max_{k=1}^j r_k + 2\bar{C}_j$. Finally, we observe that Equation (11.1) implies that $\bar{C}_k \geq r_k$ for $k = 1, \dots, n$, and since $\bar{C}_j \geq \bar{C}_k$, $k = 1, \dots, j$, we have that $\tilde{C}_j \geq \max_{k=1}^j r_k$. Thus $\tilde{C}_j \leq 3\bar{C}_j$. \square

The following theorem follows from the lemma above.

Theorem 11.1

Schedule-by- \bar{C} gives a 2-approximation for $1 \mid prec \mid \sum_j w_j C_j$ and a 3-approximation for $1 \mid prec, r_j \mid \sum_j w_j C_j$.

Proof

In the problem $1 \mid prec \mid \sum_j w_j C_j$, $r_j = 0$ for all j . Hence, from Lemma 11.2 we obtain that $\tilde{C}_j \leq 2\bar{C}_j$, hence $\sum_j w_j \tilde{C}_j \leq 2 \sum_j w_j \bar{C}_j$. The quantity $\sum_j w_j \bar{C}_j$ is the optimum value of the LP relaxation and hence is a lower bound on the integral optimum solution. This proves the 2-approximation.

For the problem $1 \mid prec, r_j \mid \sum_j w_j C_j$, we use the fact that $\tilde{C}_j \leq 3\bar{C}_j$ from Lemma 11.2 to obtain the 3-approximation. \square

The analysis can be refined to obtain a $(2 - \frac{2}{n+1})$ -approximation for $1 \mid prec \mid \sum_j w_j C_j$, see Schulz [9] for details.

Time-Indexed Formulation: The time-indexed formulation for $1 | prec | \sum_j w_j C_j$ was introduced by Dyer and Wolsey [37]. The size of the formulation is pseudo-polynomial; however, as Hall et al. [29] have shown, an approximate polynomial size formulation can be derived from it. In the time-indexed formulation an upper bound on the schedule length, T , is used. In the case of $1 | prec | \sum_j w_j C_j$ it is easy to see that $T = \sum_j p_j$ since in absence of release dates, idle time can be eliminated from any schedule. For each job j in $1, 2, \dots, n$ and time t in $1, \dots, T$ there is a variable x_{jt} that is 1 if j completes processing at time t . The formulation below is for $1 | prec | \sum_j w_j C_j$.

$$\min \sum_{j=1}^n w_j \sum_{t=1}^T t x_{jt}$$

subject to

$$\sum_{t=1}^T x_{jt} = 1 \quad j = 1, \dots, n \quad (11.4)$$

$$\sum_{s=1}^t x_{js} \geq \sum_{s=1}^{t+p_k} x_{ks} \quad \text{if } j < k, t = p_j, \dots, T - p_k \quad (11.5)$$

$$\sum_{j=1}^n \sum_{s=t}^{\min\{t+p_j-1, T\}} x_{js} \leq 1 \quad t = 1, \dots, T \quad (11.6)$$

$$x_{jt} \geq 0 \quad j = 1, \dots, n, t = 1, \dots, T \quad (11.7)$$

$$x_{jt} = 0 \quad t = 1, \dots, r_j + p_j - 1 \quad (11.8)$$

Linear Ordering Formulation of Potts: We describe the formulation of Potts [48] that uses linear ordering variables δ_{ij} for $i \neq j$. The variable δ_{ij} is 1 if i is completed before j in the schedule, and is 0 otherwise. The formulation below is for $1 | prec | \sum_j w_j C_j$.

$$\min \sum_{j=1}^n w_j C_j$$

subject to

$$C_j = p_j + \sum_{i=1}^n p_i \delta_{ij} \quad j = 1, \dots, n \quad (11.9)$$

$$\delta_{ij} + \delta_{ji} = 1 \quad i \neq j \quad (11.10)$$

$$\delta_{ij} + \delta_{jk} + \delta_{ki} \leq 2 \quad i < j < k \text{ or } k < j < i \quad (11.11)$$

$$\delta_{ij} = 1 \quad i < j \quad (11.12)$$

$$\delta_{ij} \geq 0 \quad i \neq j \quad (11.13)$$

Both of the above formulations can be easily extended to handle release dates, that is for $1 | r_j, prec | \sum_j w_j C_j$. Schulz [9] has shown that the completion time formulation is no stronger than both the time-indexed formulation and the linear ordering formulation. Thus, it follows that both these formulations can also be used to obtain Theorem 11.1. The advantage of the linear ordering formulation is that it is polynomial sized. Chudak and Hochbaum [19] have shown that the linear ordering formulation can be rewritten with only two variables per inequality, this allows it to be solved by using a minimum cut computation in an associated graph. For the problem $1 | r_j, prec | \sum_j w_j C_j$, Schulz and Skutella [30] obtain a ratio of $(e + \epsilon)$, where e is the base of the natural logarithm, this improves upon the 3-approximation that we presented. For this, they use the idea of ordering jobs by their α -completion times in a solution to the time-indexed formulation for the problem, where α is picked according to a probability distribution. See discussion in Section 11.5.

All the formulations for the single machine problem $1 | prec | \sum_j w_j C_j$ have an integrality gap of 2. While it is easy to construct gaps for the completion time and time-indexed formulations [29], it is somewhat nontrivial to do so for the linear ordering formulation. In [13] an example using bipartite strong expander graphs is used to show a gap of 2 for all the known formulations.

11.2.1.2 Identical Parallel Machines

The completion time formulation has been extended to the problem $P | prec, r_j | \sum_j w_j C_j$ by Hall et al. [29] as follows. The constraint (11.3) is replaced by the following where m is the number of machines.

$$\sum_{j \in S} p_j C_j \geq \frac{1}{2m} \left(\sum_{j \in S} p_j \right)^2 + \frac{1}{2} \sum_{j \in S} p_j^2 \quad S \subseteq N \quad (11.14)$$

For the validity of the above constraint, see Ref. [29]. Once again we can order jobs by their completion times \bar{C} in the LP relaxation. The following lemma can be shown with an analysis similar to that of Lemma 11.1.

Lemma 11.3

Let \bar{C} be a feasible solution to the completion time formulation for parallel machines and without loss of generality let $\bar{C}_1 \leq \bar{C}_2 \leq \dots \leq \bar{C}_n$. Then, the following is true for $j = 1, \dots, n$.

$$\bar{C}_j \geq \frac{1}{2m} \sum_{k=1}^j p_k$$

The following is also an easy lemma to prove from the constraints.

Lemma 11.4

For any sequence of jobs j_1, j_2, \dots, j_k such that $j_1 < j_2 < \dots < j_k$, the following holds:

$$\bar{C}_{j_k} \geq \max_{i=1}^k \left(r_{j_i} + \sum_{\ell=i}^k p_{j_\ell} \right)$$

It is however not straight forward to generalize Schedule-by- \bar{C} to the case of parallel machines. There is a tradeoff between utilizing the resources of all machines and giving priority to the ordering of the jobs. Two list scheduling algorithms are natural. The first is the greedy algorithm of Graham [6] for minimizing makespan on parallel machines. In Graham's list scheduling, we are given an ordering of the jobs that is consistent with precedence constraints. A job j is *ready* at time t if all its predecessors are completed by t and $t \geq r_j$. If a machine is free, the algorithm schedules the earliest job in the given ordering that is ready. Using Lemmas 11.3 and 11.4 the following theorem can be shown for jobs with unit processing times. The analysis is very similar to Graham's analysis for minimizing makespan. The advantage of equal processing times is that jobs that are scheduled out of order do not delay earlier jobs in the order. For the same reason the theorem below also holds if jobs can be preempted. For more details see Hall et al. [29].

Theorem 11.2

For the problems $P | prec, r_j, p_j = 1 | \sum_j w_j C_j$ and $P | prec, r_j, pmtn | \sum_j w_j C_j$, Graham's list scheduling with the ordering according to \bar{C} yields a 3-approximation algorithm.

Even though Graham's list scheduling is effective for the problems in the above theorem, it can produce schedules as much as an $\Omega(m)$ factor away from the optimum for the problem $P | prec | \sum_j w_j C_j$ [16]. The second list scheduling is one that strictly schedules in order, in spirit akin to Schedule-by- \bar{C} . It requires

some care to define the precise generalization to multiple machines. Below we describe the variant from Munier, Queyranne, and Schulz [16].

List Scheduling:

1. The list $L = (\ell(1), \ell(2), \dots, \ell(n))$ is given.
2. Initially all machines are empty. For $h = 1, \dots, m$, set machine completion time $\gamma_h = 0$.
3. For $k = 1, \dots, n$ do:
 - a. Let job $j = \ell(k)$, its start time $S_j = \max(\{C_i : i \prec j\}, \min\{\gamma_h : h = 1, \dots, m\})$ and its completion time $C_j = S_j + p_j$.
 - b. Assign job j to a machine h such that $\gamma_h \leq S_j$. Update $\gamma_h = C_j$.

Notice that the above algorithm schedules strictly in order of the list. It can be shown that using the above algorithm with the ordering provided by \bar{C} also leads to a poor schedule [16]. Using a deceptively simple modification, Munier et al. show that if the list is obtained by ordering jobs in nondecreasing order of their LP *midpoints* (the LP midpoint of job j , \bar{M}_j , is defined as $(\bar{C}_j - p_j/2)$), then the list schedule results in a 4-approximation! We now prove this result. For job j we denote by \tilde{S}_j the start time of j in the schedule constructed. The main result from [16] is the following.

Theorem 11.3

Let \bar{C} and \bar{M} denote the LP completion time and midpoint vectors of the jobs. Let \tilde{S} denote the vector of start times in the feasible schedule constructed by List Scheduling. Then for $j = 1, \dots, n$, $\tilde{S}_j \leq 4\bar{M}_j$ and hence $\tilde{C}_j \leq 4\bar{C}_j$.

The following lemmas follow easily from the modified constraint (11.14).

Lemma 11.5

Let \bar{C} be a feasible solution to the completion time formulation for parallel machines and without loss of generality let $\bar{M}_1 \leq \bar{M}_2 \leq \dots \leq \bar{M}_n$. Then, the following is true for $j = 1, \dots, n$.

$$\bar{M}_j \geq \frac{1}{2m} \sum_{k=1}^{j-1} p_k$$

Lemma 11.6

For any sequence of jobs j_1, j_2, \dots, j_k such that $j_1 \prec j_2 \prec \dots \prec j_k$, the following holds.

$$\bar{M}_{j_k} - \bar{M}_{j_1} \geq \frac{1}{2} \sum_{\ell=1}^k p_{j_\ell}$$

We will use $[j]$ to refer to the set $\{1, \dots, j\}$. Precedence constraints between jobs induce a directed acyclic graph $G = ([n], A)$ where $(i, j) \in A$ if and only if $i \prec j$. Given the schedule produced by the list scheduling algorithm we define for each j the graph $G^j = ([j], A^j)$ where

$$A^j = \{(k, \ell) \in A : k, \ell \in [j] \text{ and } \tilde{C}_\ell = \tilde{C}_k + p_\ell\}$$

Thus A^j is a subset of the precedence constraints in $[j]$ that are *tight* in the schedule. Fix a job j and consider the state of the schedule when j is scheduled by the algorithm. The time interval $[0, \tilde{S}_j)$ can be partitioned into those intervals in which all machines are busy processing jobs and those in which at least one machine is idle. Let μ be the total time in which all machines are busy and λ the rest. To prove Theorem 11.3 it is sufficient to show that $\mu \leq 2\bar{M}_j$ and $\lambda \leq 2\bar{M}_j$. Since no job $i > j$ is considered for scheduling by the time j is scheduled, $\mu \leq \frac{1}{m} \sum_{k \leq j-1} p_k$; by Lemma 11.5 it follows that $\mu \leq 2\bar{M}_j$.

Now we show that $\lambda \leq 2\bar{M}_j$. The nonbusy intervals in $[0, \bar{S}_j]$ can be partitioned into maximal intervals $(a_1, b_1), (a_2, b_2), \dots, (a_q, b_q)$ such that $0 \leq a_1$, and for $h = 2, \dots, q$, $b_{h-1} < a_h < b_h$, and $b_q \leq \bar{S}_j$. For ease of notation define $b_0 = 0$ and $a_{q+1} = \infty$. We now prove the following crucial lemma.

Lemma 11.7

Let $k \in [j]$ be a job such that $\bar{S}_k \in [b_h, a_{h+1}]$ for some $h \in 1, \dots, q$. Let $v_1, v_2, \dots, v_s = k$ be a maximal path in A^j that ends in k . Then, there is an index $g \leq h$ such that $\bar{S}_{v_1} \in [b_g, a_{g+1}]$ and if $g > 0$ then there exists a job $\ell \neq k$ such that $\bar{M}_\ell \leq \bar{M}_k$ and $\bar{S}_\ell = b_g$. Further

$$\bar{M}_k - \bar{M}_{v_1} \geq \max \left[\frac{1}{2} (b_h - a_{g+1}), 0 \right] \quad (11.15)$$

Proof

Since the path is maximal, the job v_1 has no precedence constraints preventing it from being scheduled earlier than \bar{S}_{v_1} . Hence, it must be that \bar{S}_{v_1} belongs to some busy interval $[b_g, a_{g+1}]$. Suppose $g > 0$. Since $[a_g, b_g]$ is a maximal busy interval some job ℓ starts at b_g . We claim that $\bar{M}_\ell \leq \bar{M}_{v_1}$. If not, v_1 would have been considered earlier than ℓ for scheduling and again by maximality of the path, v_1 would have started strictly before b_g , a contradiction.

Now we prove Equation (11.15). If $g = h$, then $b_h - a_{g+1} < 0$ and the equation is trivial. Suppose $g < h$, then because from the definition of edges in A^j , we have that

$$b_h - a_{g+1} \leq \bar{S}_{v_s} - \bar{S}_{v_1} = \sum_{i=1}^{s-1} \bar{S}_{v_{i+1}} - \bar{S}_{v_i} = \sum_{i=1}^{s-1} p_{v_i} \quad (11.16)$$

Since $(v_i, v_{i+1}) \in A^j$, from the LP constraints (11.2) we have the following for $i = 1, \dots, s-1$.

$$\bar{M}_{v_{i+1}} \geq \bar{M}_{v_i} + \frac{1}{2}(p_{v_i} + p_{v_{i+1}}) \quad (11.17)$$

Putting together the above two equations,

$$\bar{M}_{v_s} - \bar{M}_{v_1} = \bar{M}_k - \bar{M}_{v_1} \geq \frac{1}{2} \sum_{i=1}^{s-1} p_{v_i} \geq \frac{1}{2}(b_h - a_{g+1}) \quad (11.18)$$

Now we are ready to prove Theorem 11.3. Using Lemma 11.7 we create a sequence of indices $q = i_1 > i_2 > \dots > i_r = 0$. With each index i_ℓ we associate a job $x(i_\ell)$. The sequence is constructed as follows. We let $x(i_1) = x(q)$ be a job such that $\bar{S}_{x(q)} = b_q$, such a job must exist. Let $v_1, v_2, \dots, v_s = x(q)$ be a maximal path that ends in $x(q)$. If $v_1 \in [0, a_1]$ we stop and set $i_2 = i_r = 0$. Otherwise, from Lemma 11.7 there is a job ℓ such that $\bar{S}_\ell = b_g$ for some $g \leq q$. We claim that $g < q$ since otherwise i_1 would have been scheduled earlier than b_q . We set $i_2 = g$ and $x(i_2) = \ell$. From Equation (11.15) $\bar{M}_{x(q)} - \bar{M}_{v_1} \geq \frac{1}{2}(b_q - a_{g+1})$. Since $\bar{M}_{v_1} \geq \bar{M}_\ell$, we also have that $\bar{M}_{x(q)} - \bar{M}_{x(g)} \geq \frac{1}{2}(b_q - a_{g+1})$, in other words $\bar{M}_{x(i_1)} - \bar{M}_{x(i_2)} \geq \frac{1}{2}(b_{i_1} - a_{i_2+1})$. We continue the process with i_2 to obtain i_3 and so on until $\bar{S}_{i_r} \in [0, a_1]$. It is clear that the process terminates since the maximal path length at each step is of length at least 2. Using the same reasoning as above we have the following equation for $k = 1, \dots, r-1$.

$$\bar{M}_{x(i_k)} - \bar{M}_{x(i_{k+1})} \geq \frac{1}{2}(b_{i_k} - a_{i_{k+1}+1}) \quad (11.19)$$

Adding up all the above equations for $k = 1, \dots, r-1$ yields

$$\bar{M}_{x(i_1)} - \bar{M}_{x(i_r)} \geq \sum_{k=1}^{r-1} \frac{1}{2}(b_{i_k} - a_{i_{k+1}+1}) \quad (11.20)$$

It is easy to see from the construction that every idle interval $[a_h, b_h]$ is contained in one of the intervals $[a_{i_k+1}, b_{i_k}]$ for some k . Hence, it follows that

$$\lambda = \sum_{h=1}^q (b_h - a_h) \leq \sum_{k=1}^{r-1} (b_{i_k} - a_{i_k+1}) \leq 2(\bar{M}_{x(i_1)} - \bar{M}_{x(i_r)}) \leq 2\bar{M}_j \quad (11.21)$$

This finishes the proof of Theorem 11.3. \square

It is instructive for the reader to understand why the above proof fails if we try to use an ordering based on either the completion times or the start times ($\bar{S}_j = \bar{C}_j - p_j$). In [16] examples are given that show that the approximation ratio provided by the algorithm is tight (factor of 4) and that the LP integrality gap is at least 3.

The algorithm and the bound generalize in a straight forward fashion to the case where there are delays associated with the precedence constraints, that is, for all $i \prec j$ there is a delay d_{ij} which indicates the time that j has to wait to start after i completes.

11.2.2 Combinatorial Algorithms

We have seen LP based methods to minimize weighted completion time when jobs have precedence constraints and/or release dates. Now we describe combinatorial methods based on structural insights into the problem. Our first algorithm is for the single machine problem $1 | prec | \sum_j w_j C_j$. For identical parallel machines we give an algorithm that takes an approximately good single machine schedule and converts into an approximate schedule on parallel machines.

11.2.2.1 One-Machine Scheduling with Precedence Constraints

Polynomial time solvable cases of $1 | prec | \sum_j w_j C_j$ include precedence constraints induced by forests (in and out forests) and generalized series-parallel graphs [2–4]. These special cases have been solved by combinatorial methods (in fact by $O(n \log n)$ time algorithms).

We now describe a simple combinatorial 2-approximation algorithm for the single machine problem with general precedence constraints. This algorithm was obtained independently by Chekuri and Motwani [13], and Margot, Queyranne, and Wang [14]. Chudak and Hochbaum [19] derived a linear programming relaxation for $1 | prec | \sum_j w_j C_j$ that uses only two variables per inequality, using the linear ordering formulation of Potts [48] described earlier. Such formulations can be solved by a minimum cut computation and this also leads to a combinatorial approximation algorithm. However, the running time obtained is worse than that of the algorithm we present below [13,14] by a factor of n .

Let $G = (V, E)$ denote the precedence graph where V is the set of jobs. We will use jobs and vertices interchangeably. We say that i precedes j , denoted by $i \prec j$, if and only if there is a path from i to j in G . For any vertex $i \in V$, let G_i denote the subgraph of G induced by the set of vertices preceding i .

Definition 11.1

The rank of a job J_i , denoted by q_i , is defined as $q_i = p_i/w_i$. Similarly, the rank of a set of jobs A denoted by $q(A)$ is defined as $q(A) = p(A)/w(A)$, where $p(A) = \sum_{J_i \in A} p_i$ and $w(A) = \sum_{J_i \in A} w_i$.

Definition 11.2

A subdag G' of G is said to be precedence closed if for every job $J_i \in G'$, G_i is a subgraph of G' .

The rank of a graph is simply the rank of its node set.

Definition 11.3

We define G^* to be a precedence-closed subgraph of G of minimum rank, i.e., among all precedence-closed subgraphs of G , G^* is of minimum rank.

Note that G^* could be the entire graph G .

A Characterization of the Optimal Schedule: Smith's rule for a set of independent jobs states that there is an optimal schedule that schedules jobs in nondecreasing order of their ranks. We generalize this rule for the case of precedence constraints in a natural way. A version of the following theorem was proved by Sydney [4]. The proof relies on a careful exchange argument ([13] also provides a proof).

Definition 11.4

A segment in a schedule S is any set of jobs that are scheduled consecutively in S .

Theorem 11.4 (Sydney)

There exists an optimal sequential schedule where the optimal schedule for G^* occurs as a segment that starts at time zero.

Note that when G^* is the same as G this theorem does not help in reducing the problem.

A 2-approximation Theorem 11.4 suggests the following natural algorithm. Given G , compute G^* and schedule G^* and $G - G^*$ recursively. It is not *a priori* clear that G^* can be computed in polynomial time, however this is indeed the case. The other important issue is to handle the case when G cannot be decomposed because G^* is the same as G . We have to settle for an approximation in this case, for otherwise we would have a polynomial time algorithm to compute the optimal schedule.

The following lemma establishes a strong lower bound on the optimal schedule value when $G^* = G$.

Lemma 11.8

If G^* is the same as G , $\text{OPT} \geq w(G)p(G)/2$.

Proof

Let $\alpha = q(G)$. Let S be an optimal schedule for G . Without loss of generality assume that the ordering of the jobs in S is J_1, J_2, \dots, J_n . For any j , $1 \leq j \leq n$, observe that $C_j = \sum_{1 \leq i \leq j} p_i \geq \alpha \sum_{1 \leq i \leq j} w_i$. This is because the set of jobs J_1, J_2, \dots, J_j form a precedence closed subdag, and from our assumption on G^* it follows that $\sum_{i \leq j} p_i / \sum_{i \leq j} w_i \geq \alpha$. We bound the value of the optimal schedule as follows.

$$\begin{aligned} \text{OPT} &= \sum_{j=1}^n w_j C_j \geq \sum_{j=1}^n w_j \sum_{i=1}^j \alpha w_i \\ &= \alpha \left(\sum_{j=1}^n w_j^2 + \sum_{1 \leq i < j \leq n} w_i w_j \right) = \alpha \left(\left(\sum_{j=1}^n w_j \right)^2 - \sum_{1 \leq i < j \leq n} w_i w_j \right) \\ &\geq \alpha (w(G)^2 - w(G)^2/2) \\ &= \alpha w(G)^2/2 = w(G)p(G)/2 \end{aligned}$$

The last equality is true because $q(G^*) = q(G) = p(G)/w(G) = \alpha$. □

The following lemma is straight forward.

Lemma 11.9

Any feasible schedule with no idle time has a weighted completion time of at most $w(G)p(G)$.

Theorem 11.5

If G^ for a graph can be computed in time $O(T(n))$, then there is a 2-approximation algorithm for computing the minimum weighted completion time schedule that runs in time $O(nT(n))$.*

Proof

Given G , we compute G^* in time $O(T(n))$. If G^* is the same as G we schedule G arbitrarily and Lemmas 11.8 and 11.9 guarantee that we have a 2-approximation. If G^* is a proper subdag we recurse on G^* and $G - G^*$. From Theorem 11.4 we have $\text{OPT}(G) = \text{OPT}(G^*) + p(G^*)w(G - G^*) + \text{OPT}(G - G^*)$. Inductively if we have 2-approximate solutions for G^* and $G - G^*$ it is clear that we can combine them to get a 2-approximation of the overall schedule. Now we establish the running time bound. We observe that $(G^*)^* = G^*$, therefore it suffices to recurse only on $G - G^*$. It follows that we make at most n calls to the routine to compute G^* and the bound on the running time of the procedure follows. \square

An algorithm to compute G^* using a parametric minimum cut computation in an associated graph is presented in Lawler's book [49]. The associated graph is dense and has $\Omega(n^2)$ edges. Applying known algorithms for parametric minimum cut, G^* can be computed in strongly polynomial time of $O(n^3)$ [50] or in $O(n^{8/3} \log nU)$ time [51], where $U = \max_{i=1}^n (p_i + w_i)$.

11.2.2.2 A General Conversion Algorithm

In this section we describe a technique of Chekuri et al. [11] to obtain parallel machine schedules from one-machine schedules that works even when jobs have precedence constraints and release dates. Given an average weighted completion time scheduling problem, we show that if we can approximate the one-machine preemptive variant, then we can also approximate the m -machine nonpreemptive variant, with a slight degradation in the quality of approximation.

We use the superscript m to denote the number of machines; thus, X^m denotes a schedule for m machines, C^m denotes the sum of weighted completion time of X^m , and C_j^m denotes the completion time of job j under schedule X^m . The subscript OPT refers to an optimal schedule; thus, an optimal schedule is denoted by X_{OPT}^m , and its weighted completion time is denoted by C_{OPT}^m . For a set of jobs A , $p(A)$ denotes the sum of processing times of jobs in A .

Definition 11.5

For any vertex j , recursively define the quantity κ_j as follows. For a vertex j with no predecessors $\kappa_j = p_j + r_j$. Otherwise, define $\kappa_j = p_j + \max\{\max_{i \prec j} \kappa_i, r_j\}$. Any path P_{ij} from i to j where $p(P_{ij}) = \kappa_j$ is referred to as a critical path to j .

We now describe the Delay-List algorithm. Given a one-machine schedule which is a ρ -approximation, Delay-List produces a schedule for $m \geq 2$ machines whose value is within a factor $(k_1\rho + k_2)$ of the optimal m -machine schedule, where k_1 and k_2 are small constants. We will describe a variant of this scheduling algorithm which yields $k_1 = (1 + \beta)$ and $k_2 = (1 + 1/\beta)$ for any $\beta > 0$.

The main idea is as follows. The one-machine schedule taken as a list (jobs in order of their completion times in the schedule) provides some priority information on which jobs to schedule earlier. However, when trying to convert the one-machine schedule into an m -machine one, precedence constraints

prevent complete parallelization. Thus, we may have to execute jobs out-of-order from the list to benefit from parallelism. If all p_i are identical (say 1), we can afford to use Graham's list scheduling. If there is an idle machine and we schedule some available job on it, it is not going to delay jobs which become available soon, since it completes in one time unit. On the other hand, if not all p_i 's are the same, a job could keep a machine busy, delaying more profitable jobs that become available soon. At the same time, we cannot afford to keep machines idle. We strike a balance between the two extremes: schedule a job out-of-order only if there has been enough idle time already to justify scheduling it. To measure whether there has been enough idle time, we introduce a charging scheme.

Assume, for ease of exposition, that all processing times are integers and that time is discrete. This restriction can be removed without much difficulty and we use it only in the interests of clarity and intuition. A job is *ready* if it has been released and all its predecessors are done. The time at which job j is ready in a schedule X is denoted by q_j^X and time at which it starts is denoted by S_j^X .

We use X^m to denote the m -machine schedule that our algorithm constructs and for ease of notation the superscript m will be used in place of X^m to refer to quantities of interest in this schedule. Let $\beta > 0$ be some constant. At each discrete time step t , the algorithm applies one of the following three cases:

1. *There is an idle machine M and the first job j on the list is ready at time t — schedule j on M and charge all uncharged idle time in the interval (q_j^m, S_j^m) to j .*
2. *There is an idle machine — and the first job j in the list is not ready at time t but there is another ready job on the list — focusing on the job k which is the first in the list among the ready jobs, schedule it if there is at least βp_k uncharged idle time among all machines, and charge βp_k idle time to k .*
3. *There is no idle time or the above two cases do not apply — do not schedule any job, merely increment t .*

Definition 11.6

A job is said to be scheduled in order if it is scheduled when it is at the head of the list. Otherwise it is said to be scheduled out of order. The set of jobs which are scheduled before a job j but which come later in the list than j is denoted by O_j . The set of jobs which come after j in the list is denoted by A_j and those which come before j by B_j (includes j).

Definition 11.7

For each job i , define a path $P'_i = j_1, j_2, \dots, j_\ell$, with $j_\ell = i$ with respect to the schedule X^m as follows. The job j_k is the predecessor of j_{k+1} with the largest completion time (in X^m) among all the predecessors of j_{k+1} such that $C_{j_k}^m \geq r_{j_{k+1}}$; ties are broken arbitrarily. j_1 is the job where this process terminates when there are no predecessors which satisfy the above condition. The jobs in P'_i define a disjoint set of time intervals $(0, r_{j_1}]$, $(S_{j_1}^m, C_{j_1}^m]$, \dots , $(S_{j_\ell}^m, C_{j_\ell}^m]$ in the schedule. Let κ'_i denote the sum of the lengths of the intervals.

Fact 11.1 $\kappa'_i \leq \kappa_i$.

Fact 11.2 *The idle time charged to each job i is less than or equal to βp_i .*

Proof

The fact is clear if idle time is charged to i according to case 2 in the description of our algorithm. Suppose case 1 applies to i . Since i was ready at q_i^m and was not scheduled according to case 2 earlier, the idle time

in the interval (q_i^m, S_i^m) that is charged to i is less than βp_i . We remark that the algorithm with discrete time units might charge more idle time due to integrality of the time unit. However, that is easily fixed in the continuous case where we schedule i at the first time instant when at least βp_i units of uncharged idle time have accumulated. \square

A crucial feature of the algorithm is that when it schedules jobs, it considers only the first job in the list that is ready, even if there is enough idle time for other ready jobs that are later in the list. The proof of the following lemma makes use of this feature.

Lemma 11.10

For every job i , there is no uncharged idle time in the time interval (q_i^m, S_i^m) , and furthermore all the idle time is charged only to jobs in B_i .

Proof

By the preceding remarks, it is clear that no job in A_i is started in the time interval (q_i^m, S_i^m) since i was ready at q_i^m . From this we can conclude that there is no idle time charged to jobs in A_i in that time interval. Since i is ready at q_i^m and was not scheduled before S_i^m , from cases 1 and 2 in the description of our algorithm there cannot be any uncharged idle time. \square

The following lemma shows that for any job i , the algorithm does not schedule too many jobs from A_i before scheduling i itself.

Lemma 11.11

For every job i , the total idle time charged to jobs in A_i , in the interval $(0, S_i^m)$, is bounded by $m(\kappa_i' - p_i)$. It follows that $p(O_i) \leq m(\kappa_i' - p_i)/\beta \leq m(\kappa_i - p_i)/\beta$.

Proof

Consider a job j_k in P_i' . The job j_{k+1} is ready to be scheduled at the completion of j_k , that is $q_{j_{k+1}}^m = C_{j_k}^m$. From Lemma 11.10, it follows that in the time interval between $(C_{j_k}^m, S_{j_{k+1}}^m)$ there is no idle time charged to jobs in $A_{j_{k+1}}$. Since $A_{j_{k+1}} \supset A_i$ it follows that all the idle time for jobs in A_i has to be accumulated in the intersection between $(0, S_i^m)$ and the time intervals defined by P_i' . This quantity is clearly bounded by $m(\kappa_i' - p_i)$. The second part follows since the total processing time of the jobs in O_i is bounded by $1/\beta$ times the total idle time that can be charged to jobs in A_i (recall that $O_i \subseteq A_i$). \square

Theorem 11.6

Let X^m be the schedule produced by the algorithm DELAY LIST using a list X^1 . Then for each job i , $C_i^m \leq (1 + \beta)p(B_i)/m + (1 + 1/\beta)\kappa_i' - p_i/\beta$.

Proof

Consider a job i . We can split the time interval $(0, C_i^m)$ into two disjoint sets of time intervals T_1 and T_2 as follows. The set T_1 consists of all the disjoint time intervals defined by P_i' . The set T_2 consists of the time intervals obtained by removing the intervals in T_1 from $(0, C_i^m)$. Let t_1 and t_2 be the sum of the times of the intervals in T_1 and T_2 , respectively. From the definition of T_1 , it follows that $t_1 = \kappa_i' \leq \kappa_i$. From Lemma 11.10, in the time intervals of T_2 , all the idle time is either charged to jobs in B_i and, the only jobs which run are from $B_i \cup O_i$. From Fact 11.2, the idle time charged to jobs in B_i is bounded by $\beta p(B_i)$. Therefore, the time t_2 is bounded by $(\beta p(B_i) + p(B_i) + p(O_i))/m$. Using Lemma 11.11 we see that $t_1 + t_2$ is bounded by $(1 + \beta)p(B_i)/m + (1 + 1/\beta)\kappa_i' - p_i/\beta$. \square

One-Machine Relaxation: In order to use Delay List, we will need to start with a one machine schedule. The following two lemmas provide lower bounds on the optimal m -machine schedule in terms of the optimal one-machine schedule. This one-machine schedule can be either preemptive or nonpreemptive, the bounds hold in either case.

Lemma 11.12

$$C_{\text{OPT}}^m \geq C_{\text{OPT}}^1/m.$$

Proof

Given a schedule X^m on m machines with total weighted completion time C^m , we will construct a one-machine schedule X^1 with total weighted completion time at most mC^m as follows. Order the jobs according to their completion times in X^m with the jobs completing early coming earlier in the ordering. This ordering is our schedule X^1 . Note that there could be idle time in the schedule due to release dates. If $i \prec j$ then $C_i^m \leq S_j^m \leq C_j^m$ which implies that there will be no precedence violations in X^1 . We claim that $C_i^1 \leq mC_i^m$ for every job i . Let P be the sum of the processing times of all the jobs which finish before i (including i) in X^m . Let I be the total idle time in the schedule X^m before C_i^m . It is easy to see that $mC_i^m \geq P + I$. We claim that $C_i^1 \leq P + I$. The idle time in the schedule X^1 can be charged to idle time in the schedule X^m and P is the sum of all jobs which come before i in X^1 . This implies the desired result. □

Lemma 11.13

$$C_{\text{OPT}}^m \geq \sum_i w_i \kappa_i = C_{\text{OPT}}^\infty.$$

Proof

The length of the critical path κ_i , is an obvious lower bound on the completion time C_i^m of job i . Summing up over all jobs gives the first inequality. It is also easy to see that if the number of machines is unbounded that every job i can be scheduled at the earliest time it is available and will finish by κ_i yielding the equality. □

Obtaining Generic m -Machine Schedules: In this section we derive our main theorem relating m -machine schedules to one-machine schedules.

We begin with a corollary to Theorem 11.6.

Corollary 11.1

Let X^m be the schedule produced by the algorithm Delay-List using a one-machine schedule X^1 as the list. Then for each job i , $C_i^m \leq (1 + \beta)C_i^1/m + (1 + 1/\beta)\kappa_i$.

Proof

Since all jobs in B_i come before i in the one-machine schedule, it follows that $p(B_i) \leq C_i^1$. Plugging this and Fact 11.1 into the bound in Theorem 11.6, we conclude that $C_i^m \leq (1 + \beta)C_i^1/m + (1 + 1/\beta)\kappa_i$. □

Theorem 11.7

Given an instance I of scheduling to minimize sum of weighted completion times and a one-machine schedule for I that is within a factor ρ of an optimal one-machine schedule, Delay-List gives a m -machine schedule for I that is within a factor $(1 + \beta)\rho + (1 + 1/\beta)$ of an optimal m -machine schedule. Further, Delay-List can be implemented in $O(n \log n)$ time.

Proof

Let X^1 be a schedule which is within a factor ρ of the optimal one-machine schedule. Then $C^1 = \sum_i w_i C_i^1 \leq \rho C_{\text{OPT}}^1$. By Corollary 11.1, the schedule created by the algorithm Delay-List satisfies,

$$\begin{aligned} C^m &= \sum_i w_i C_i^m \\ &\leq \sum_i w_i \left[(1 + \beta) \frac{C_i^1}{m} + \left(1 + \frac{1}{\beta}\right) \kappa_i \right] \\ &= \frac{1 + \beta}{m} \sum_i w_i C_i^1 + \left(1 + \frac{1}{\beta}\right) \sum_i w_i \kappa_i \end{aligned}$$

From Lemmas 11.12 and 11.13 it follows that

$$\begin{aligned} C^m &\leq \frac{(1 + \beta)\rho C_{\text{OPT}}^1}{m} + \left(1 + \frac{1}{\beta}\right) C_{\text{OPT}}^\infty \\ &\leq \left[(1 + \beta)\rho + \left(1 + \frac{1}{\beta}\right) \right] C_{\text{OPT}}^m \end{aligned}$$

The running time bound on Delay-List can be easily seen from the description of the algorithm. □

There is an interesting property of the conversion algorithm that is useful in its applications and worth pointing out explicitly. We explain it via an example. Suppose we want to compute an m -machine schedule with release dates and precedence constraints. From Theorem 11.7 it would appear that we need to compute a one-machine schedule for the problem that has both precedence constraints and release dates. However, we can completely ignore the release dates in computing the one-machine schedule X^1 . This follows from a careful examination of the upper bound proved in Theorem 11.6 and the proof of Theorem 11.7. This is useful since the approximation ratio for the problem $1 | prec | \sum_j w_j C_j$ is 2, while it is $(e + \epsilon)$ for $1 | prec, r_j | \sum_j w_j C_j$ [30]. In another example, the problem $1 || \sum_j w_j C_j$ has a very simple polynomial time algorithm using Smith's ratio rule while $1 | r_j | \sum_j w_j C_j$ is NP-hard. Thus release dates play a role only in the conversion algorithm and not in the single machine schedule. A similar claim can be made when there are delays between jobs. In this setting a positive delay d_{ij} between jobs i and j indicates that i is a predecessor of j and that j cannot start until d_{ij} time units after i completes. Delay-List and its analysis can be generalized to handle delays, and obtain the same results as those in Theorem 11.6 and Theorem 11.7. The only change required is in the definition of ready time of a job which now depends also on the delay after a predecessor finishes. As with release dates we can ignore the delay values (not the precedence constraints implied by them though) in computing the single machine schedule.

11.3 Unrelated Machines

In this section we present LP-based approximation algorithms for $R || \sum_j w_j C_j$ and $R | r_j | \sum_j w_j C_j$ from [17]. For each job j and machine i , p_{ij} denotes the processing time of j on i . Let $T = \max_{j \in J} r_j + \sum_{j \in J} \max_i p_{ij}$ denote an upper bound on the schedule length. The algorithms are based on the following time indexed relaxation which is pseudo-polynomial in the input size. The variable y_{ijt} indicates the fraction of the time interval $[t, t + 1)$ that j is processed on i .

$$\min \sum_j w_j C_j$$

subject to

$$\sum_{i=1}^m \sum_{t=r_j}^T \frac{y_{ijt}}{p_{ij}} = 1 \quad \text{for all } j \quad (11.22)$$

$$\sum_{j \in J} y_{ijt} \leq 1 \quad \text{for all } i \text{ and } t \quad (11.23)$$

$$C_j \geq \sum_{i=1}^m \sum_{t=r_j}^T \left[\frac{y_{ijt}}{p_{ij}} \left(t + \frac{1}{2} \right) + \frac{1}{2} y_{ijt} \right] \quad \text{for all } j \quad (11.24)$$

$$y_{ijt} \geq 0 \quad \text{for all } i, j, \text{ and } t \quad (11.25)$$

Equation (11.22) states that job j is processed completely. Equation (11.23) ensures that machine i can process one unit of work in each unit time interval. To understand Equation (11.24) consider the ideal situation, when a job j is processed on a single machine i continuously from time h to $h + p_{ij}$. Then, it can be verified easily that Equation (11.24) gives the exact completion time for j . In other circumstances, it can be easily checked that Equation (11.24) provides a lower bound on C_j . Note that the formulation remains a relaxation even if y_{ijt} are constrained to be binary.

We now describe a randomized rounding algorithm, Rand-Round, from [17] that uses an optimal solution to the above LP.

1. Let \tilde{C} be an optimal solution to the LP.
2. Assign each job j , *independently*, to machine-time pair (i, t) , where the probability of j being assigned to (i, t) is exactly y_{ijt}/p_{ij} . Let (i_j, t_j) be the chosen pair for j .
3. On each machine i , schedule the jobs assigned to i nonpreemptively in order of t_j (ties broken *randomly*).

Let \tilde{C}_j be the completion time of j produced by the above algorithm. Note that \tilde{C}_j is a random variable. The analysis rests on the following lemma.

Lemma 11.14

The expected completion time of j , $E[\tilde{C}_j]$ is upper bounded by the following:

$$E[\tilde{C}_j] \leq 2 \sum_{i=1}^m \sum_{t=r_j}^T \frac{y_{ijt}}{p_{ij}} \left(t + \frac{1}{2} \right) + \sum_{i=1}^m \sum_{t=r_j}^T y_{ijt}$$

Proof

Consider a fixed job j and let t' be the earliest time such that there is no idle time in the interval $[t', \tilde{C}_j)$ on machine i_j . Let A_j be the set of jobs that are scheduled on i_j in this interval. It follows from the definition of t' and A_j that

$$\tilde{C}_j = t' + \sum_{k \in A_j} p_{ik}$$

It is easy to see that $t' \leq t_j$ for, otherwise we would have scheduled j earlier. Since i_j was busy in $[t', \tilde{C}_j)$, we can upper bound \tilde{C}_j as

$$\tilde{C}_j \leq t_j + \sum_{k \in A_j} p_{ik}$$

To bound $E[\tilde{C}_j]$ we analyze the conditional expectation $E_{i,t}[\tilde{C}_j]$ under the event that j is assigned to the machine–time pair (i, t) . It follows from the previous equation that

$$\begin{aligned} E_{i,t}[\tilde{C}_j] &\leq t + E_{i,t} \left[\sum_{k \in A_j} p_{ik} \right] \\ &\leq t + p_{ij} + \sum_{k \neq j} p_{ik} \Pr_{i,t}[k \text{ scheduled on } i \text{ before } j] \\ &= t + p_{ij} + \sum_{k \neq j} p_{ik} \left(\sum_{s=r_k}^{t-1} \frac{y_{iks}}{p_{ik}} + \frac{1}{2} \frac{y_{ikt}}{p_{ik}} \right) \\ &\leq t + p_{ij} + \left(t + \frac{1}{2} \right) \leq 2 \left(t + \frac{1}{2} \right) + p_{ij} \end{aligned}$$

In the penultimate line above the factor of $\frac{1}{2}$ comes from the fact that the algorithm breaks ties randomly. In the last inequality above we use the LP constraints given in Equation (11.24). Now we can obtain the unconditional expectation as

$$\begin{aligned} E[\tilde{C}_j] &= \sum_{i=1}^m \sum_{t=r_j}^T \Pr[j \text{ assigned to } (i, t)] \cdot E_{i,t}[\tilde{C}_j] \\ &= \sum_{i=1}^m \sum_{t=r_j}^T \frac{y_{ijt}}{p_{ij}} E_{i,t}[\tilde{C}_j] \leq 2 \sum_{i=1}^m \sum_{t=r_j}^T \frac{y_{ijt}}{p_{ij}} \left(t + \frac{1}{2} \right) + \sum_{i=1}^m \sum_{t=r_j}^T y_{ijt} \end{aligned}$$

This gives us the desired equation. \square

Combining Lemma 11.14 and the constraint (11.24) it follows that $E[\tilde{C}_j] \leq 2\tilde{C}_j$. By linearity of expectation we obtain the following.

Theorem 11.8

For instances of $R \mid r_j \mid \sum_j w_j C_j$, *Rand-Round* produces a solution whose expected value is within twice the value of the optimum solution the LP.

If there are no release dates, that is for the problem $R \mid \sum_j w_j C_j$ the integrality gap can be improved to $3/2$ as follows. First, we can strengthen the LP by adding the following additional constraints:

$$C_j \geq \sum_{i=1}^m \sum_{t=r_j}^T y_{ijt} \quad \text{for all } j \quad (11.26)$$

When all release dates are 0, we can also strengthen Lemma 11.14 to show that

$$E[\tilde{C}_j] \leq \sum_{i=1}^m \sum_{t=r_j}^T \frac{y_{ijt}}{p_{ij}} \left(t + \frac{1}{2} \right) + \sum_{i=1}^m \sum_{t=r_j}^T y_{ijt}$$

The above equation comes about by observing in the proof of Lemma 11.14 that $t' = 0$ and $\tilde{C}_j = \sum_{k \in A_j} p_{ik}$ which in turn leads to $E_{i,t}[\tilde{C}_j] \leq p_{ij} + (t + \frac{1}{2})$.

We leave it as an exercise to the reader to prove the validity of (11.26) and put the above facts together to show the following theorem.

Theorem 11.9

For instances of $R \mid \sum_j w_j C_j$, *Rand-Round* produces a solution whose expected value is within $3/2$ times the value of an optimum solution to the LP with the strengthened inequalities (11.26).

In [17], the integrality gaps in the above two theorems are shown to be tight. However, as we point out in Section 11.5, the integrality gaps of the LP when strengthened with an additional inequality is unknown. Rand-Round can be derandomized using the method of conditional probabilities. We refer the reader to [17] for more details. As we mentioned earlier, the LP is pseudo-polynomial in the input size. The formulation can be modified to obtain a polynomial sized LP while losing only a factor of $(1 + \epsilon)$ in the objective function value. This idea was originally used by Hall, Shmoys, and Wein [8] and details in the context of the above results can be found in [17]. Using the modified LP and the Rand-Round algorithm results in approximation ratios of $(2 + \epsilon)$ and $(3/2 + \epsilon)$ for $R | r_j | \sum_j w_j C_j$ and $R || \sum_j w_j C_j$, respectively. The latter result for $R || \sum_j w_j C_j$ was independently observed by Chudak [18]. Using convex quadratic formulations, Skutella [28] improved the ratios to 2 and $3/2$, respectively.

11.4 Approximation Schemes for Problems with Release Dates

The algorithms considered thus far share the basic paradigm of solving a relaxation of the completion time problem and use information from the relaxation to obtain an ordering on the jobs and/or an assignment of the jobs to machines. Even though these ideas yield constant factor approximation bounds, there are fundamental barriers to turning these approaches into approximation schemes. In particular, there are provable constant factor gaps between the objective value of the relaxation and the average completion time of the optimal schedule. Thus these approaches cannot directly yield approximation schemes.

In this section we develop a framework to obtain approximation schemes when jobs have release dates. In particular we present an approximation scheme for the problem $P | r_j | \sum_j w_j C_j$. The framework essentially consists of a dynamic programming-based search in a carefully chosen subset of all possible schedules. Given any $\epsilon > 0$, we will show that we can identify a subset of schedules such that each schedule in the subset has a nice structure and the subset contains a schedule that is $(1 + \epsilon)$ -approximate. We then use dynamic programming to explore this structured space and find the best possible schedule. There are two key components for implementing this framework: (i) input transformations, and (ii) dynamic programming. Input transformations add structure to a given arbitrary instance while dynamic programming helps us efficiently enumerate over the space of near-optimal schedules for a given structured instance. Each transformation on the input instance as well as optimality relaxation in the dynamic programming potentially increases the objective function value by a $1 + O(\epsilon)$ factor, so we can perform a constant number of them while still staying within a $1 + O(\epsilon)$ factor of the original optimum. When we describe such a transformation, we shall say it produces $1 + O(\epsilon)$ loss.

We next develop each of these components in detail. The ideas described below apply to both $P | r_j | \sum_j w_j C_j$ and $P | r_j, pmtn | \sum_j w_j C_j$. In describing the dynamic programming, we will primarily focus on the nonpreemptive case, and toward the end mention the modifications needed for the preemptive case. To simplify notation we will assume throughout that $1/\epsilon$ is an integer and that $\epsilon \leq 1/4$. We use C_j and S_j to denote the completion and start time respectively of job j , and OPT to denote the objective value of some fixed optimal schedule.

We note that the framework that we describe can be extended with further ideas to obtain approximation schemes for the problems $Rm | r_j | \sum_j w_j C_j$ [20] and $Q | r_j | \sum_j w_j C_j$ [27].

11.4.1 Input Transformations

The goal of the input transformations is to impose a certain structure on the input instance that will facilitate efficient search by dynamic programming. In order to analyze the effect of each transformation, we show that an optimal schedule can be modified with a small increase in objective value so as to conform to the modified input structure. We will apply two transformations. The first transformation is a standard discretization of the input instance by *geometric rounding*. The second transformation, referred to as *job shifting*, is a new transformation that ensures that at each release date, only a small number of jobs arrive.

11.4.1.1 Geometric Rounding

Our first transformation reduces the number of distinct processing times and release dates in a given input instance.

Lemma 11.15

With $1 + \epsilon$ loss, we can assume that all processing times and release dates are integer powers of $1 + \epsilon$.

Proof

We transform the given instance in two steps. First, we multiply every release date and processing time by $1 + \epsilon$; this increases the objective by the same amount (we are simply changing time units). Then we decrease each date and time to the next lower integer power of $1 + \epsilon$ (which is still greater than the original value). This can only decrease the objective function value. \square

Notation: For an arbitrary integer x , we define $R_x := (1 + \epsilon)^x$. As a result of Lemma 11.15, we can assume that all release dates are of the form R_x for some integer x . We partition the time interval $[0, \infty)$ into disjoint intervals of the form $I_x := [R_x, R_{x+1})$ (Lemma 11.16 below ensures that no jobs are released at time 0). We will use I_x to refer to both the interval and the size $(R_{x+1} - R_x)$ of the interval. We will often use the fact that $I_x = \epsilon R_x$, i. e., the length of an interval is ϵ times its start time.

11.4.1.2 Job Shifting

The goal of job shifting is to ensure that for every job, the difference between its release time and completion time is only a small number of intervals, say $f(1/\epsilon)$, for some function f . It is easy to see intuitively that this property can be helpful in efficiently implementing a dynamic programming approach. An input instance can violate this property in two ways. First, a job released at some time R_x can be arbitrarily larger than $(1 + \epsilon)^x$ and hence must be alive for an arbitrarily large number of intervals. Second, many jobs can be simultaneously released at some time R_x , no schedule can finish all of the jobs in a fixed number of intervals following R_x . The first of these two problems is easily fixed by suitably delaying the release date of each job. The second problem is more difficult to fix and it requires statically pruning and delaying jobs at each release date. This shifting of jobs is a critical ingredient of our approach.

We start by classifying each job as either small or large, and handle them differently. We say that a job is *small* if its size is less than ϵ^2 times the size of the interval in which it arrives (i. e., $p_x < \epsilon^2 I_x$), and *large* otherwise. The lemma below shows that we can always ensure that no job is arbitrarily large compared to the size of the interval in which it is released.

Lemma 11.16

With $1 + \epsilon$ loss, we can enforce $r_j \geq \epsilon p_j$ for all jobs j .

Proof

Increase all processing times by a factor of $(1 + \epsilon)$; thus a job j with original processing time p_j , now has processing time $(1 + \epsilon)p_j$. As noted in Lemma 11.15, this increases the optimal objective value by a factor of at most $(1 + \epsilon)$. Now consider an optimal schedule σ for this modified instance. For each job j , ignore the first ϵp_j units of its processing in the schedule σ . All remaining processing of job j now occurs at time ϵp_j or later, and moreover, p_j units of job j still get processed. The lemma follows. \square

An immediate corollary is that in nonpreemptive schedules, a job never crosses many intervals.

Corollary 11.2

In any nonpreemptive schedule, each job crosses at most $s := \lceil \log_{1+\epsilon}(1 + \frac{1}{\epsilon}) \rceil$ intervals.

Proof

Suppose job j starts in interval $I_x = [R_x, R_{x+1})$. Since $R_x \geq r_j \geq \epsilon p_j$ (Lemma 11.16), we have $I_x = \epsilon R_x \geq \epsilon^2 p_j$. The s intervals following x sum in size to $I_x/\epsilon^2 \geq p_j$. \square

Lemma 11.17

There exists a $(1 + O(\epsilon))$ -approximate schedule such that the following hold:

1. for any two small jobs j and k with $r_j \leq r_k$, $\frac{p_j}{w_j} \leq \frac{p_k}{w_k}$ and $j < k$, $x(j) \leq x(k)$ holds, where $x(j)$ and $x(k)$ are the intervals in which j and k are scheduled
2. each small job finishes within the interval that it is started

Proof

Fix an optimal schedule. Let s_x be the time allocated by this schedule to executing small jobs within interval I_x . We can assume that the small jobs are executed contiguously within each interval — this increases the schedule value by at most a $(1 + \epsilon)$ -factor. We now create a new schedule to satisfy the first property in the statement of the lemma as follows. We first describe an assignment of small jobs to intervals without specifying the precise schedule of the jobs within an interval. The assignment is done as follows. Suppose we have assigned small jobs to intervals I_1 to I_{x-1} . Now we describe the assignment of jobs to I_x . Let A_x be set of all small jobs that have been released by I_x but have not been assigned to any of the intervals I_1 to I_{x-1} . For job j in A_x consider the tuple $(p_j/w_j, r_j, j)$. We order jobs in A_x in nondecreasing order of their tuples, using the dictionary ordering. We assign jobs in this order to I_x until the total processing time of the jobs assigned to I_x just exceeds s_x . Since each job in A_x is small, the total processing assigned to I_x is no more than $s_x + \epsilon I_x$. We claim that the total weight of jobs assigned to intervals I_1 to I_x by the above procedure dominates the weight of small jobs completed by the optimal schedule in intervals I_1 to I_x . This is relatively easy to verify and we leave it as an exercise to the reader. Also, observe that for each x , small jobs assigned to I_x by the above procedure can be scheduled in I_x using the space left by the optimal schedule, provided we stretch I_x by a $(1 + \epsilon)$ -factor. Stretching the intervals by a $(1 + \epsilon)$ -factor increases the objective function value by at most a $(1 + \epsilon)$ -factor. The assignment procedure satisfies the first desired property. The second property also can be easily accomplished by expanding each interval by a $(1 + \epsilon)$ -factor. \square

As a result of Lemma 11.17, we can order all small jobs released at R_x according to their ratio $\frac{p_j}{w_j}$ and consider them for scheduling only in that order. Let T_x and H_x denote the small and large jobs released at R_x . Note that in this section small means an ϵ^2 fraction of the interval. Let $p(S)$ denote the sum of the processing times of the jobs in set S . The next lemma says that any input instance I can be modified with $1 + \epsilon$ loss to an instance I' so that the total size of the small and large jobs released at any release date R_x is $O(mI_x)$. This lemma plays an important role in our implementation of the dynamic programming framework since it allows us to represent compactly information about unfinished jobs as we move from one block to the next.

Lemma 11.18

(Job Shifting) An instance can be modified with $1 + O(\epsilon)$ loss to an instance I' such that the following conditions hold.

- $p(T'_x) \leq (1 + \epsilon)mI_x$ for all x
- The number of distinct job sizes in H'_x is at most $\lfloor 1 + 4 \log_{1+\epsilon} \frac{1}{\epsilon} \rfloor$
- The number of jobs of each distinct size in H'_x is at most $\frac{m}{\epsilon^2}$

Proof

Consider the input instance I . The total processing time available in interval I_x is mI_x . Order the small jobs in T_x by nondecreasing ratios $\frac{p_j}{w_j}$ and pick jobs according to this order until the processing time of

jobs picked would exceed $(1 + \epsilon)mI_x$ if one more job is added. Picking jobs according to this order is justified by Lemma 11.17. The remaining jobs, which are released at R_x but cannot be processed in I_x , can safely be moved to the next release date R_{x+1} .

For each job j in H_x , Lemma 11.16 yields $R_x \geq \epsilon p_j$. On the other hand, since j is large we get $p_j \geq \epsilon^2 I_x = \epsilon^3 R_x$. Since all job sizes are powers of $1 + \epsilon$, the number of distinct job sizes in H_x is as claimed. For a particular size that is large for I_x , we can order jobs by nonincreasing weights. The number of jobs of each size class that can be executed in the current interval is limited to $\frac{mI_x}{\epsilon^3 R_x} = \frac{m}{\epsilon^2}$. \square

11.4.2 Dynamic Programming

The basic idea of the dynamic programming is to decompose the time horizon into a sequence of *blocks*. A block is a set of $s = \lceil \log_{1+\epsilon}(1 + \frac{1}{\epsilon}) \rceil$ consecutive intervals. Note that $s \leq \frac{1}{\epsilon^2}$. Let $\mathcal{B}_0, \mathcal{B}_1, \dots, \mathcal{B}_\ell$ be the partition of the time interval $[\min_j r_j, D]$ into blocks where D is an upper bound on the schedule makespan say, $(\sum_j p_j + \max_j r_j)$. Our goal is to do dynamic programming with blocks as units. There is interaction between blocks since jobs from an earlier block can cross into the current block. However, by the choice of the block size and Corollary 11.2, no job crosses an entire block in any nonpreemptive schedule. In other words, jobs that start in \mathcal{B}_i finish either in \mathcal{B}_i or \mathcal{B}_{i+1} . A *frontier* describes the potential ways that jobs in one block finish in the next. An incoming frontier for a block \mathcal{B}_i specifies for each machine the time at which the crossing job from \mathcal{B}_{i-1} finishes on that machine.

Lemma 11.19

There exists a $(1 + \epsilon)$ -approximate schedule which considers only $(m + 1)^{s/\epsilon}$ feasible frontiers between any two blocks.

Proof

By Lemma 11.18 we can restrict attention to schedules in which small jobs never cross an interval. Each block consists of a fixed number s of intervals. Fix an optimal schedule and consider any machine in a block \mathcal{B}_i . A large job j continuing from the preceding block finishes in one of the s intervals of block \mathcal{B}_i which we denote by $I_{x(j)}$. We can round up C_j to C'_j where $C'_j = R_{x(j)} + i \in I_{x(j)}$ for some integer $0 \leq i \leq \frac{1}{\epsilon} - 1$. This will increase the schedule value by only a $1 + \epsilon$ factor. Thus, we can restrict the completion times of crossing jobs to $\frac{s}{\epsilon}$ discrete time instants. Each machine realizes one of these possibilities. A frontier can thus be described as a tuple $(m_1, \dots, m_{s/\epsilon})$ where m_i is the number of machines with crossing jobs finishing at the i th discrete time instant. Therefore, there are at most $(m + 1)^{s/\epsilon}$ frontiers to consider. \square

Let \mathcal{F} denote the possible set of frontiers between blocks. The high level idea behind the dynamic programming is now easy to describe. The dynamic programming table entry $O(i, F, U)$ stores the minimum weighted completion time achievable by starting the set U of jobs before the end of block \mathcal{B}_i while leaving a frontier of $F \in \mathcal{F}$ for block \mathcal{B}_{i+1} . Given all the table entries for some i , the values for $i + 1$ can be computed as follows. Let $W(i, F_1, F_2, V)$ be the minimum weighted completion time achievable by scheduling the set of jobs V in block \mathcal{B}_i , with F_1 as the incoming frontier from block \mathcal{B}_{i-1} and F_2 the outgoing frontier to block \mathcal{B}_{i+1} . We obtain the following equation:

$$O(i + 1, F, U) = \min_{F' \in \mathcal{F}, V \subset U} [O(i, F', V) + W(i + 1, F', F, U - V)]$$

There are two difficulties in implementing the dynamic programming. First, we cannot maintain the table entries for each possible subset of jobs in polynomial time. Therefore, we need to show the existence of approximate schedules that have compact representations for the set of subsets of jobs remaining after each block. Second, we need a procedure that computes the quantity $W(i, F_1, F_2, V)$. In what follows, we

describe how these elements can be efficiently implemented for the parallel machine case. We focus on the nonpreemptive case and later describe the necessary modifications for handling the preemptive case. Below, we give a lemma that bounds the duration for which a job can remain unprocessed after its release.

Lemma 11.20

There exists a $[1 + O(\epsilon)]$ -approximate schedule in which every job finishes within $O(\frac{1}{\epsilon^2})$ blocks after it is released.

Proof

Consider some fixed optimal schedule. For each block \mathcal{B}_i , let A_i denote the set of jobs released in \mathcal{B}_i which are not finished in the optimal schedule by the end of \mathcal{B}_i , where $\ell_i = i + \frac{1}{\epsilon^2}$. For each i , we alter the schedule by scheduling all the jobs in A_i in \mathcal{B}_i . Let I_x be the smallest interval in block \mathcal{B}_i . Using Lemma 11.18, the total volume of jobs released in \mathcal{B}_i can be verified to be at most $\epsilon m I_x$, in other words $p(A_i) \leq \epsilon m I_x$. Further, for every job $j \in A$, $p_j \leq \epsilon^4 I_x$ holds. To schedule jobs of A_i in \mathcal{B}_i , we group jobs of A_i into units each with volume between ϵI_x and $(\epsilon + \epsilon^4) I_x$. From the bound on $p(A_i)$ the number of units is at most m . We assign each unit to an exclusive machine and schedule the unit on the machine as soon as the crossing job on that machine from \mathcal{B}_{ℓ_i-1} finishes. We shift the jobs on the machine to accommodate the extra volume. Clearly, the completion times of jobs in A_i have only decreased. It is easy to see that expanding intervals by a $(1 + O(\epsilon))$ -factor can accommodate the increase in the completion times of jobs in \mathcal{B}_i for all i . The new schedule satisfies the desired property. \square

11.4.3 Nonpreemptive Scheduling on Parallel Machines

We now describe how to implement the two main components of the dynamic program: Compact representation of job subsets and scheduling within a block. We use time-stretching to show existence of schedules for which both tasks can be efficiently performed.

11.4.3.1 Compact Representation of Job Subsets

Recall that H_x and T_x denote the large and small jobs released at R_x . Let A_{xi} and B_{xi} denote the set of large and small jobs released at R_x that are scheduled in block \mathcal{B}_i . Let U_{xi} and V_{xi} denote the set of large and small jobs among jobs released at R_x that remain *after* block \mathcal{B}_i . Our goal is to show that there exist $(1 + \epsilon)$ — approximate schedules with compact representations for these sets. Let $b(x)$ denote the block containing the interval I_x .

Large Jobs: Consider large jobs released in interval I_x . By Lemma 11.18, there are $O(\frac{1}{\epsilon} \log \frac{1}{\epsilon})$ distinct size classes in H_x . Within each size class, we assume that the jobs are ordered in nonincreasing order of weight. For each block \mathcal{B}_i , we specify the set U_{xi} by simply indicating for each size class, the smallest indexed job that remains to be scheduled. Since within each size class, any optimal schedule executes the jobs in nonincreasing weight order, this completely specifies the set A_{xi} . The total number of choices for U_{xi} are $(m/\epsilon^2)^{O(\log(1/\epsilon)/\epsilon)}$ by Lemma 11.18. Moreover, for any block \mathcal{B}_i , by Lemma 11.20, there are $O(s/\epsilon^2)$ release dates before \mathcal{B}_i whose jobs can be still alive. Thus, there are only $(m/\epsilon^2)^{O((\log(1/\epsilon)/\epsilon)(s/\epsilon^2))}$ possible sets of large jobs that can be alive at the end of any block \mathcal{B}_i .

Small Jobs: The approach described above is not efficient for maintaining a description of unprocessed small jobs since the number of small jobs arriving at any release date can be arbitrarily large. We will instead focus on maintaining information about the unprocessed volume of small jobs.

Recall that for each release date R_x , we order the set T_x using Smith's ratio rule. Without any loss of generality, assume that the jobs in T_x are $\{1, 2, \dots, \alpha\}$, indexed in nondecreasing order of p/w ratio. Let j_1 be the least index such that the total processing time of the first j_1 jobs exceeds $\epsilon^2 I_x$, denote this set of jobs by $J_1(x)$. Let j_2 be the least index such that the total processing time of the next $j_2 - j_1$ jobs exceeds

$\epsilon^2 I_x$, denote this set of jobs by $J_2(x)$. We continue in this manner to construct sets $J_1(x), J_2(x), \dots, J_\ell(x)$, where each set $J_i(x)$, $1 \leq i < \ell$ contains a processing volume of at least $\epsilon^2 I_x$, and at most $2\epsilon^2 I_x$. The last set $J_\ell(x)$ contains at most $2\epsilon^2 I_x$ processing volume. By Lemma 11.18, we know that $\ell \leq 2m$. Our main observation is as follows.

Lemma 11.21

There exists a $(1 + O(\epsilon))$ -approximate schedule in which for every release date R_x , all jobs in a set $J_i(x)$ are scheduled in the same interval.

Proof

By Lemma 11.17, we know that there exists a $(1 + \epsilon)$ -approximate schedule in which all small jobs arriving at any release date are executed in accordance with Smith's ratio rule. We start with such a schedule, say σ , and show that by stretching each interval by a $(1 + 2\epsilon)$ factor, we can modify σ to satisfy the property indicated in the Lemma. In the modified schedule, no job is scheduled in a later interval than it is scheduled in σ . Thus the modification only increases the objective function value by $(1 + 2\epsilon)$.

Fix a set $J_i(x)$ and let I_y be the first interval in which a job from $J_i(x)$ is scheduled in σ . Let M be a machine on which this event occurs. If all jobs in $J_i(x)$ are scheduled in I_y , the property above is satisfied. Otherwise, we know that no jobs from any set $J_{i'}(x)$ for $i' > i$ were scheduled in this interval. We now schedule all jobs in $J_i(x)$ on machine M in this interval. The total additional processing load added is at most $2\epsilon^2 I_x$. Adding over all intervals I_x with $x < y$, we can bound the additional processing overhead on machine M in interval I_y to be at most $2\epsilon I_y$. Thus, this additional load is accommodated by the stretching described above. \square

Lemma 11.21 above says that for each block \mathcal{B}_i , we can specify the set V_{xi} by simply indicating the smallest index i such that $J_i(x)$ remains to be scheduled. This completely specifies the set B_{xi} . The total number of choices for V_{xi} are (m/ϵ^2) by Lemma 11.18. Moreover, for any block \mathcal{B}_i , by Lemma 11.20, there are $O(s/\epsilon^2)$ release dates before \mathcal{B}_i whose jobs can be still alive. Thus, there are only $(m/\epsilon^2)^{O(s/\epsilon^2)}$ possible sets of small jobs that can be alive at the end of any block \mathcal{B}_i .

We summarize our considerations and results of this subsection in the following lemma:

Lemma 11.22

There is a $(1 + O(\epsilon))$ -approximate schedule \mathcal{S} such that for each block \mathcal{B}_i the following is true:

- There are $k = (\frac{m}{\epsilon^2})^{O(1/\epsilon^9)}$ sets G_1^i, \dots, G_k^i that can be constructed in polynomial time
- G_i , the set of jobs remaining in \mathcal{S} after block \mathcal{B}_i , is one of $\{G_1^i, \dots, G_k^i\}$

11.4.3.2 Scheduling Jobs Within a Block

We now describe how to compute $W(i, F_1, F_2, V)$. Since this is itself an NP-hard problem we settle for a relaxation. A $1 + \epsilon$ decision procedure for computing $W(i, F_1, F_2, V)$ outputs a schedule that is within $1 + \epsilon$ of $W(i, F_1, F_2, V)$ and shifts the frontier F_2 by at most a $1 + \epsilon$ factor. Clearly such a procedure suffices in order to compute a $(1 + O(\epsilon))$ -optimal solution to the dynamic program given above. We now describe a $1 + \epsilon$ decision procedure that runs in polynomial time for each fixed ϵ .

For the purposes of scheduling jobs in the block, we partition the job set V into *big* and *tiny* as follows. Let I_x be the smallest interval in \mathcal{B}_i . We call a job $j \in V$ big if $p_j \geq \epsilon I_x$, otherwise we call it tiny. Note that this definition differs from the earlier definition of large and small. Since the block consists of s intervals, for any big job $j \in V$ and interval I_y in \mathcal{B}_i we have the property that $p_j \geq \epsilon^2 I_y$.

Our objective is to enumerate over all potential schedules of big jobs. In particular, we restrict ourselves to schedules where, in each interval I_x , a big job starts only at one of the $\frac{1}{\epsilon^3}$ times specified by $R_x + i\epsilon^3 I_x$, for $i = 0, \dots, \frac{1}{\epsilon^3} - 1$. Furthermore, in our enumeration of big job schedules we will only specify the size

and the start times of the big jobs scheduled. This is sufficient information to reconstruct their schedule: whenever we have two jobs of same size available, we always schedule the one with the larger weight first. With these restrictions, the schedule of big jobs on a machine within a block is completely determined by three things: its incoming frontier, its outgoing frontier, and the sizes of the big jobs started at each of the discrete time units in each of the s intervals. We estimate the number of different possibilities for each machine. The number of discrete times where a big job can be scheduled is $\frac{s}{\epsilon^3} \leq 1/\epsilon^3$. The number of big job sizes in a block can be seen to be $O(\log \frac{1}{\epsilon})$. Hence, the total number of possibilities is $(O(\log \frac{1}{\epsilon}))^{1/\epsilon^3}$ which is upper bounded by $k = 2^{O(1/\epsilon^6)}$. Thus, the configurations of all machines are from one of $(m+1)^k$ possibilities. Out of these we consider only those that are compatible with the incoming and outgoing frontiers F_1 and F_2 and have a feasible schedule for the big jobs in V . Both conditions can be checked in a straightforward way.

We schedule the tiny jobs in a greedy fashion in the spaces left by the big jobs. For each interval I_x , all the big jobs that start and finish in I_x can be assumed to execute contiguously at the earliest available time in the interval. We increase each of the spaces by a $1 + \epsilon$ factor to accommodate all the tiny jobs. The scheduling of tiny jobs is similar to that described in the proof of Lemma 11.17. In each interval I_x in \mathcal{B}_i , we order the tiny jobs in V that are released by I_x and not yet scheduled, in nondecreasing order of p_j/w_j . The jobs are scheduled in this order within the spaces left by the big jobs until there is no more space left in interval I_x . Then we proceed to I_{x+1} . This procedure can be accomplished in polynomial time in the number of tiny jobs in V , but can also be done in polynomial in m by grouping tiny jobs into units of size ϵI_x . We omit details. The scheduling of tiny jobs has to be repeated with each possibility of scheduling large jobs. We thus obtain the following.

Lemma 11.23

There is a $1 + \epsilon$ decision procedure to compute $W(i, F_1, F_2, V)$ that runs in time $(m+k)^k$ where $k = 2^{O(1/\epsilon^6)}$.

We remark that the running time of the procedure can be improved by doing dynamic programming between intervals of the block instead of brute force enumeration of all big job schedules. The improved running time will be $m^{\text{poly}(1/\epsilon)}$. We omit details here and state the result.

Theorem 11.10

There is a PTAS for $P \mid r_j \mid \sum w_j C_j$ that constructs a $(1 + \epsilon)$ -approximation in time $O(n \log n + n(m+1)^{\text{poly}(1/\epsilon)})$.

The number of potential blocks for the dynamic programming is $O(\log D)$, where D is an upper bound on the schedule makespan. However, there are only $O(n/\epsilon^3)$ interesting blocks since each job j finishes by r_j/ϵ^4 .

11.4.4 Preemptive Scheduling on Parallel Machines

In the preemptive case, several computational aspects of the preceding algorithm can be simplified, leading to an approximation scheme with a better running time. Specifically, since large jobs can be executed fractionally, we do not need to keep track of the frontier formed by the crossing jobs. Moreover, we can do dynamic programming directly with intervals instead of blocks and an approximate schedule can be specified by the fractions of jobs that are processed in any interval. This significantly reduces the amount of enumeration needed in the dynamic programming. For instance, since there are no release dates within an interval, we can use McNaughton's wrap around rule [25] to compute a preemptive schedule with optimal makespan in $O(n)$ time. Thus if we knew the job fragments that execute within an interval, they can be efficiently scheduled. We omit here the various technical details involved and summarize below the running time of our approximation scheme.

Theorem 11.11

There is a PTAS for $P | r_j, pmtn | \sum w_j C_j$ that constructs a $(1 + \epsilon)$ -approximation in time $O(n \log n + n(m + 1)^{\text{poly}(1/\epsilon)})$.

11.5 Conclusions and Open Problems

We have surveyed approximation algorithms for minimizing sum of weighted completion times under a variety of constraints and machine environments. We have chosen to present a few algorithms and ideas in detail rather than giving an overview of all the known results. Below we mention several related ideas and results that we would have liked to cover but had to omit due to space constraints.

- Before [20] obtained PTASes for problems with release dates, constant factor approximation algorithms were known that had ratios better than the ones we presented in Section 11.2. They are still the algorithms of choice for simplicity and efficiency. We refer the reader to [11] for a $(\frac{e}{e-1})$ -approximation for $1 | r_j | \sum_j C_j$, to [52] for a 1.6853-approximation for $1 | r_j | \sum_j w_j C_j$, and to [17] for a 2-approximation to $P | r_j | \sum_j w_j C_j$. These algorithms have the additional advantage of being online. The underlying technical idea is to order the jobs by their α_j -completion times in a time-indexed relaxation for the problem where the α_j are chosen from an appropriately chosen probability distribution. The notion of α -completion times was used in [8] and the power of randomization to improve the ratios was first demonstrated independently in [11] and [12]. This technique also improves the approximation ratio for $1 | r_j, prec | \sum_j w_j C_j$ from 3 to $(e + \epsilon)$ [30].
- As we mentioned in a few places, makespan and weighted completion time are related in several ways. With precedence constraints, makespan is a special case of weighted completion time. However, (approximation) algorithms for minimizing makespan can be used to obtain approximation algorithms for minimizing weighted completion time. We mention two techniques in this regard. The first is dual-approximation based approach for online algorithms developed in [8]. The second approach which is more general and suited for offline algorithms is the one of Queyranne and Sviridenko [21] where approximation algorithms for makespan that are based on simple lower bounds are translated into approximation algorithms for weighted completion time. They use this approach to obtain algorithms for shop scheduling problem with minsum criteria.
- As indicated earlier, time-indexed formulations are pseudo-polynomial in size of the input. To obtain polynomial time algorithms, it is necessary to reduce their size. Hall et al. [8,29] accomplish this by partitioning time into geometrically increasing intervals: for parameter $\epsilon > 0$, the intervals are of the form $[(1 + \epsilon)^\ell, (1 + \epsilon)^{\ell+1})$ for $0 \leq \ell \leq \lceil \log_{1+\epsilon} T \rceil$. The formulation has variables $x_{j\ell}$ to indicate if job j finishes in interval ℓ . The interval indexed formulation is particularly useful in obtaining an $O(\log m)$ -approximation for the problem $Q | r_j, prec | \sum_j w_j C_j$ [15]. Interval indexed formulations allow a more direct way to use a makespan algorithm to obtain an algorithm for minimizing weighted completion times. For example, the algorithms in [21] which are based on completion time variables do not explicitly use interval indexed formulations. However, the notion of geometrically increasing intervals is used in the rounding phase. We also mention that this relation between makespan and weighted completion time shows the existence of schedules that are simultaneously good for both measures. For a more formal treatment of the models where such results are possible and for the best known tradeoffs, see [53–55].
- We did not detail in-approximability results in this survey except to mention the known results in the table in Figure 11.1. Hoogeveen, Schuurman, and Woeginger [22] derived most of the known nontrivial results, establishing APX-hardness for several variants. The reductions are, in spirit, similar to the earlier known in-approximability results for minimizing makespan, although there are technical differences.

This survey has focussed primarily on recent work for minimizing weighted completion time. Some prior surveys on scheduling provide pointers to earlier work. These include the surveys of Graham et al. [56] from 1979 and that of Lawler et al. [5] from 1993. More recent surveys of interest are by Hall [57] and by Karger, Stein, and Wein [58]. We list below several interesting open problems in the context of scheduling to minimize weighted completion time.

1. Approximability of the single machine problem $1 | prec | \sum_j w_j C_j$. The approximation ratio we have is 2 via both LP based methods and combinatorial methods. On the other hand we do not know if the problem is APX-hard. Closing this gap is perhaps the most interesting open problem in this area. All the known LP formulations have an integrality gap of 2 as shown in [13]. Woeginger [33] has given an approximation preserving reduction from arbitrary instances to a restricted class of instances used in [13] to establish integrality gaps.
2. Improved approximation ratio for $P | prec | \sum_j w_j C_j$. Munier et al. [16], as described in Section 11.2.1.2, obtain a 4-approximation. However, even for the LP relaxation they use, the best lower bound on the integrality gap is 3. In terms of in-approximability, there is no evidence that the problem is harder to approximate than the makespan problem $P | prec | C_{max}$ for which we have a 2-approximation and an in-approximability ratio of $4/3$.
3. Improved approximation for $R || \sum_j w_j C_j$ and $R | r_j | \sum_j w_j C_j$. The integrality gap of the LP for $R | r_j | \sum_j w_j C_j$, as presented in Section 11.3, is 2, and even with the additional constraint (11.26), the gap is $3/2$ for $R || \sum_j w_j C_j$. We can further strengthen the LP by adding the following set of constraints.

$$\sum_{i=1}^m y_{ijt} \leq 1 \quad \text{for all jobs } j, \text{ and } t = 0, \dots, T \quad (11.27)$$

The integrality gap of the LP strengthened with the above inequality is unknown for both the problems and it would be interesting to see if improved approximation ratios can be obtained by using it. This problem is suggested in [17].

There are several open problems for minimizing makespan when jobs have precedence constraints that translate into open problem for minimizing weighted completion time. We refer the reader to the list of open problems compiled by Schuurman and Woeginger [59] for some of them.

References

- [1] W. E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.
- [2] D. Adolphson. Single machine job sequencing with precedence constraints. *SIAM Journal on Computing*, 6:40–54, 1977.
- [3] E. L. Lawler. Sequencing jobs to minimize total weighted completion time. *Annals of Discrete Mathematics*, 2:75–90, 1978.
- [4] J. Sydney. Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs. *Operations Research*, 23(2):283–298, 1975.
- [5] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: algorithms and complexity. In S. C. Graves et al. (eds.), *Handbooks in OR & MS*, Elsevier Science Publishers, 4:445–522, 1993.
- [6] R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–81, 1966.
- [7] C. Phillips, C. Stein, and J. Wein. Minimizing average completion time in the presence of release dates. *Mathematical Programming B*, 82:199–223, 1998.

- [8] L. A. Hall, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: offline and online algorithms. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, 142–151, 1996.
- [9] A. S. Schulz. Scheduling to minimize total weighted completion time: performance guarantees of lp based heuristics and lower bounds. In *Proceedings of IPCO*, 301–315, 1996.
- [10] S. Chakrabarti, C. A. Phillips, A. S. Schulz, D. B. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. In *Proceedings of the 23rd ICALP*, 1996.
- [11] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. In *SIAM Journal on Computing*, 31(1): 146–166, 2000.
- [12] M. X. Goemans. Improved approximation algorithms for scheduling with release dates. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, 591–598, 1997.
- [13] C. Chekuri and R. Motwani. Precedence constrained scheduling to minimize weighted completion time on a single machine. *Discrete Applied Mathematics*, 98(1-2): 29–38, 1999.
- [14] F. Margot, M. Queyranne, and Y. Wang. Decompositions, network flows, and a precedence constrained single machine scheduling problem. *Operations Research*, 2000.
- [15] F. Chudak and D. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. *Journal of Algorithms*, 30:323–343, 1999.
- [16] A. Munier, M. Queyranne, and A. S. Schulz. Approximation bounds for a general class of precedence constrained parallel machine scheduling problems. In *Proceedings of IPCO*, 367–382, 1998.
- [17] A. Schulz and M. Skutella. Scheduling unrelated machines by randomized rounding. In *SIAM Journal on Discrete Mathematics*, 15(4): 450–469, 2002.
- [18] F. Chudak. A min-sum $3/2$ -approximation algorithm for scheduling unrelated parallel machines. *Journal of Scheduling*, 2:73–77, 1999.
- [19] F. Chudak and D. Hochbaum. A half-integral linear programming relaxation for scheduling precedence-constrained jobs on a single machine. *Operations Research Letters*, 25: 199–204, 1999.
- [20] F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, and M. Sviridenko. Approximation schemes for minimizing average weighted completion time with release dates. In *Proceedings of FOCS*, 1999.
- [21] M. Queyranne and M. Sviridenko. Approximation algorithms for shop scheduling problems with minsum objective. *Journal of Scheduling*, 5: 287–305, 2002.
- [22] J. A. Hoogeveen, P. Schuurman, and G. J. Woeginger. Non-approximability results for scheduling problems with minsum criteria. In *Proceedings of IPCO*, 353–366, 1998.
- [23] M. Skutella and G. J. Woeginger. A PTAS for minimizing the weighted sum of job completion times on parallel machines. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, 400–407, 1999.
- [24] C. Chekuri, R. Johnson, R. Motwani, B. K. Natarajan, B. R. Rau, and M. Schlansker. Profile-driven instruction level parallel scheduling with applications to super blocks. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, 58–67, 1996.
- [25] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6:1–12, 1959.
- [26] W. Horn. Minimizing average flowtime with parallel machines. *Operations Research*, 21:846–847, 1973.
- [27] C. Chekuri and S. Khanna. A PTAS for minimizing weighted completion time on uniformly related machines. In *Proceedings of ICALP*, 2001.
- [28] M. Skutella. Convex quadratic and semidefinite programming relaxations in scheduling. *Journal of the ACM*, 48(2):206–242, 2001.
- [29] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: offline and online algorithms. *Mathematics of Operations Research*, 22:513–544, 1997.
- [30] A. Schulz and M. Skutella. Random-based scheduling: new approximations and LP lower bounds. In *Proceedings of RANDOM*, 1997.
- [31] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26:22–35, 1978.

- [32] M. Skutella and M. Uetz. Scheduling precedence-constrained jobs with stochastic processing times on parallel machines. In *Proceedings of SODA*, 589–590, 2001.
- [33] G. Woeginger. On the approximability of average completion time scheduling under precedence constraints. In *Proceedings of ICALP*, 887–897, 2001.
- [34] S. Kolliopoulos and G. Steiner. Partially-ordered knapsack and applications to scheduling. In *Proceedings of ESA*, 2002.
- [35] E. Balas. On the facial structure of scheduling polyhedra. *Mathematical Programming Studies*, 24: 179–218, 1985.
- [36] L. Wolsey. Mixed integer programming formulations for production planning and scheduling problems. *Invited Talk at the 12th International Symposium on Mathematical Programming*, MIT, Cambridge.
- [37] M. Dyer and L. Wolsey. Formulating the single machine sequencing problem with release dates as a mixed integer program. *Discrete Applied Mathematics*, 26:255–270, 1990.
- [38] M. Queyranne. Structure of a simple scheduling polyhedron. *Mathematical Programming*, 58: 263–285, 1993.
- [39] M. Queyranne and Y. Wang. Single-machine scheduling polyhedra with precedence constraints. *Mathematics of Operations Research*, 16(1):1–20, 1991.
- [40] J.-B. Lasserre and M. Queyranne. Generic scheduling polyhedra and a new mixed-integer formulation for single-machine scheduling. In *Proceedings of IPCO*, 136–149, 1992.
- [41] J. Sousa and L. Wolsey. A time-indexed formulation of nonpreemptive single-machine scheduling problems. *Mathematical Programming*, 54:353–367, 1992.
- [42] A. von Arnim and A. Schulz. Facets of the generalized permutahedron of a poset. *Discrete Applied Mathematics*, 72: 179–192, 1997.
- [43] Y. Crama and C. Spiessma. Scheduling jobs of equal length: complexity, facets, and computational results. In *Proceedings of IPCO*, 277–291, 1995.
- [44] J. Van den Akker, C. Van Hoesel, and M. Savelsbergh. Facet-inducing inequalities for single-machine scheduling problems. Memorandum COSOR 93-27, Eindhoven University of Technology, Eindhoven, The Netherlands, 1993.
- [45] J. Van den Akker. LP-based solution methods for single-machine scheduling problems. PhD Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1994.
- [46] J. Van den Akker, C. Hurkens, and M. Savelsbergh. A time-indexed formulation for single-machine scheduling problems: branch and cut. Preprint, 1995.
- [47] A. von Arnim, R. Schrader, and Y. Wang. The permutahedron of N-sparse posets. Preprint, 1996.
- [48] C. N. Potts. An algorithm for the single machine sequencing problem with precedence constraints. *Mathematical Programming Studies*, 13:78–87, 1980.
- [49] E. L. Lawler. *Combinatorial Optimization*. Holt, Rinehart, and Winston 1976.
- [50] G. Gallo, M. D. Grigoriadis, and R. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18:30–55, 1989.
- [51] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, 2–11, 1997.
- [52] M. Goemans, M. Queyranne, A. Schulz, M. Skutella, and Y. Wang. Single machine scheduling with release dates. *SIAM Journal on Discrete Mathematics*, 15:165–192, 2002.
- [53] C. Stein and J. Wein. On the existence of schedules that are near-optimal for both makespan and total weighted completion time. *OR Letters*, 21: 1997.
- [54] J. Aslam, A. Rasala, C. Stein, and N. Young. Improved bicriteria existence theorems for scheduling problems. In *Proceedings of SODA*, 846–847, 1999.
- [55] A. Rasala, C. Stein, E. Torng, and P. Uthaisombut. Existence theorems, lower bounds and algorithms for scheduling to meet two objectives. In *Proceedings of SODA*, 723–731, 2002.
- [56] 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.

- [57] L. Hall. Approximation algorithms for scheduling. In D. Hochbaum (ed.), *Approximation Algorithms for NP-Hard Problems*, PWS Publishing, 1995.
- [58] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. In M. Atallah (ed.), *Algorithms and Theory of Computation Handbook*, CRC Press, 1998.
- [59] P. Schuurman and G. Woeginger. Polynomial time approximation algorithms for machine scheduling: ten open problems. *Journal of Scheduling*, 2(5):203–213, 2000.
- [60] N. Alon, Y. Azar, G. J. Woeginger, and T. Yadid. Approximation schemes for scheduling on parallel machines. *Journal of Scheduling*, 1:55–66, 1998.
- [61] K. R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, 1974.
- [62] 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.
- [63] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the Association for Computing Machinery*, 23:317–327, 1976.
- [64] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.
- [65] M. W. P. Savelsbergh, R. N. Uma, and J. Wein. An experimental study of LP-based scheduling heuristics. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, 453–461, 1998.