Approximation Algorithms for the Job Interval Selection Problem and Related Scheduling Problems

Julia Chuzhoy *
Computer Science Department
Technion — IIT
Haifa 32000, Israel
E-mail: cjulia@cs.technion.ac.il

Rafail Ostrovsky †
Telcordia Technologies
445 South Street, Morristown
New Jersey 07960
E-mail: rafail@research.telcordia.com

Yuval Rabani ‡
Computer Science Department
Technion — IIT
Haifa 32000, Israel
E-mail: rabani@cs.technion.ac.il

Abstract

In this paper we consider the job interval selection problem (JISP), a simple scheduling model with a rich history and numerous applications. Special cases of this problem include the so-called real-time scheduling problem (also known as the throughput maximization problem) in single and multiple machine environments. In these special cases we have to maximize the number of jobs scheduled between their release date and deadline (preemption is not allowed). Even the single machine case is NP-hard. The unrelated machines case, as well as other special cases of JISP, are MAX SNP-hard. A simple greedy algorithm gives a 2-approximation for JISP. Despite many efforts, this was the best approximation guarantee known, even for throughput maximization on a single machine. In this paper, we break this barrier and show an approximation guarantee of less than 1.582 for arbitrary instances of JISP. For some special cases, we show better results. Our methods can be used to give improved bounds for some related resource allocation problems that were considered recently in the literature.

1 Introduction

Problem statement and motivation. The job interval selection problem (JISP) is a simple yet powerful model of scheduling problems. In this model, the input is a set of n jobs. Each job is a set of intervals of the real line. The intervals may be listed explicitly or implied by other parameters defining the job. To schedule a job, one of the intervals defining it must be selected. To schedule several jobs, the intervals selected for the jobs must not overlap. The objective is to schedule as many jobs as possible under these constraints. For example, one popular special case of JISP has each job j specified by a release date r_j, a deadline d_j, and a processing time p_j. To schedule job j, an interval of length p_j must be selected within the interval $[r_j, d_j]$. Using the notation convention of [21], this problem is equivalent to $1|r_j|\sum \overline{U}_j$. The generalizations of this problem to multiple machine environments (for example, the unrelated machines case $R|r_j|\sum \overline{U}_j$) are also common in applications. These can be modeled as JISP by concatenating the schedules for the machines along the real line, and specifying the possible intervals for each job accordingly. Due to some of their applications, these special cases of JISP are often called the *throughput maximization problem* or the *real-time scheduling problem*.

Instances of JISP are used to model scheduling problems in numerous applications. Some examples include: selection of projects to be performed during a space mis-

*Work supported in part by Yuval Rabani's grant from the CONSIST consortium.

†Part of this work was done while visiting the Computer Science Department at the Technion.

‡Part of this work was done while visiting Telcordia Research. Work at the Technion supported by Israel Science Foundation grant number 386/99, by BSF grants 96-00402 and 99-00217, by Ministry of Science contract number 9480198, by EU contract number 14084 (APPOL), by the CONSIST consortium (through the MAGNET program of the Ministry of Trade and Industry), and by the Fund for the Promotion of Research at the Technion.

sion [16],[1] placement of feeders in feeder racks of an assembly line for printed circuit boards [9, 27], time-constrained communication scheduling [1], and adaptive rate-controlled scheduling for multimedia applications [28, 23, 25]. These applications and others inspired the development of many heuristics for JISP or special cases of JISP, most of them lacking theoretical analysis.

Our results. In this paper we give several exact and approximation algorithms for JISP or special cases of JISP. In particular, our main result is a polynomial time approximation algorithm for JISP with guarantee arbitrarily close to $e/(e-1) < 1.582$. Our algorithm gives better guarantees for JISPk, the special case of JISP where each job has at most k possible intervals. For example, our bound for JISP2 is arbitrarily close to $\frac{4}{3}$. We consider the special case of $1|r_j|\sum \overline{U}_j$ and give a pseudo-polynomial time [2] algorithm to solve the problem optimally for the special case of constant relative window sizes (i.e., when there is a constant k such that for every job j, $d_j - r_j \leq k \cdot p_j$), which occurs in adaptive rate-controlled scheduling applications. For the same problem, we give a polynomial time approximation scheme for the special case that the job processing times are taken from a constant sized set with a constant ratio of largest to smallest value. In fact, the latter two results hold even in the case that jobs have weights and the goal is to maximize the total weight of scheduled jobs (i.e., for special cases of $1|r_j|\sum w_j\overline{U}_j$). Our results can be extended to handle more general cases. For example, we get a ratio arbitrarily close to $(2e-1)/(e-1) < 2.582$ for a generalization of JISP where intervals have heights and can overlap in time, as long as the total height at any time does not exceed 1. Our polynomial time approximation scheme works even for a fixed number of identical machines $Pm|r_j|\sum w_j\overline{U}_j$. The details will appear in the full version of the paper.

Previous work. Work on (special cases of) JISP dates back to the 1950s. Jackson [17] proved that the earliest due date (EDD) greedy rule is an optimal algorithm for $1||L_{\max}$. This implies that if all jobs can be scheduled in an instance of $1||\sum \overline{U}_j$, then EDD finds such a schedule. Moore [24] gave a greedy $O(n \log n)$ time optimal algorithm for $1||\sum \overline{U}_j$. On the other hand, the weighted version $1||\sum w_j\overline{U}_j$ is NP-hard (KNAPSACK is a special case when all deadlines are equal). Sahni [26] presented a fully polynomial time approximation scheme for this problem. When release dates are introduced, then already $1|r_j|\sum \overline{U}_j$ is NP-hard in the strong sense [13]. The following simple greedy rule gives a 2-approximation algorithm: Whenever the machine becomes idle, schedule a job that finishes first among all available jobs (see Adler et al. [1] or Spieksma [27]). Much of the recent work on JISP variants extends this bound to more general settings. Indeed, Spieksma [27] showed that the greedy algorithm gives a 2-approximation for arbitrary instances of JISP. Bar-Noy, Guha, Naor, and Schieber [5] gave a 2-approximation for $1|r_j|\sum w_j\overline{U}_j$ and a 3-approximation for $R|r_j|\sum w_j\overline{U}_j$ using a natural time-indexed linear programming formulation of fractional schedules. Bar-Noy, Bar-Yehuda, Freund, Naor, and Scheiber [4] and independently Berman and DasGupta [7] gave combinatorial 2-approximation algorithms for $R|r_j|\sum w_j\overline{U}_j$, based on the local ratio/primal-dual schema.[3] Though these and other papers contain better bounds for some special cases of JISP (see below), no technique for improving upon the factor of 2 approximation was known prior to this paper, even for the special case of $1|r_j|\sum \overline{U}_j$. The integrality ratio of the natural LP formulation, even for this special case, is 2 [27, 5], and attempts to strengthen the relaxation have failed [12]. As for hardness results, JISP2 is MAX SNP-hard [27]. Also, $R|r_j|\sum \overline{U}_j$ is MAX SNP-hard [5]. In both cases, the constant bounds for which the problem is known to be hard are very close to 1.

Some other special cases of JISP are known to be in P. Interval scheduling, where every job has a single choice, is equivalent to maximum independent set in interval graphs, and therefore has a polynomial time algorithm, even for the weighted case (see [14]). In fact, Arkin and Silverberg [2] gave a flow-based algorithm for weighted interval scheduling on identical machines. The problem becomes NP-hard on unrelated machines, even without weights. Baptiste [3], generalizing a result of Carlier [8], showed that $1|p_j = p, r_j|\sum w_j\overline{U}_j$ (i.e., when all job processing times are equal) is in P. His dynamic programming algorithm can be generalized easily to handle the case of a fixed number of related machines $Qm|p_j = p, r_j|\sum w_j\overline{U}_j$.

There are also NP-hard special cases of JISP that were known to have better than 2 approximations. Spieksma [27] proved that the natural LP formulation has a better than 2 integrality ratio in the case of JISP2. Berman and DasGupta [7] gave a better than 2 approximation algorithm for the special case of $1|r_j|\sum w_j\overline{U}_j$ with constant relative window sizes (the ratio approaches 2 as the relative window sizes grow). Our optimal algorithm improves their result. Bar-Noy et al. [5] showed that the greedy algorithm's approximation guarantee for the identical machines case $P|r_j|\sum \overline{U}_j$ approaches $e/(e-1)$ as the number of machines grows. The same holds in the weighted case for their LP-based algorithm and for the combinatorial algorithms

[1] According to the reference, the decision process may take up to 25% of the budget of a mission.

[2] This means that the time parameters r_j, d_j, p_j for each job are given in unary notation.

[3] Using time-indexed formulations requires the time parameters r_j, d_j, p_j to be written in unary. For time parameters in binary notation, slightly weaker bounds hold. In all cases where job weights are mentioned, we assume that they are given in binary notation.

of [4, 7]. They pointed out this improvement as a possible scheduling anomaly. Our results refute this possibility, as they give guarantees approaching $e/(e-1)$ for all cases of JISP.

We note that some of the above-mentioned problems were investigated also in the context of on-line computing, where jobs have to be scheduled or discarded as they arrive (see, for example, [6, 22, 10, 19]).

Our methods. As mentioned above, it seems hopeless to improve the previously known factor 2 approximation using the natural LP relaxation or some simple modification of it, other than in some special cases. Our algorithms rely instead on proving special structural properties of optimal or near-optimal solutions. The idea underlying the approximation algorithm for JISP is a partition of the time line into blocks, such that most of the jobs can be scheduled in blocks that do not contain many jobs. The computation of the partition is not trivial. The partition allows us to generate an LP relaxation that leads to the improved approximation guarantee. A partition into blocks also underlies the PTAS for the case of a constant number of job sizes. There, the partition is simple, and most of the difficulty is in setting the dynamic program that exploits the partition. Finally, the pseudo-polynomial time algorithm for bounded relative window sizes uses a dynamic program that is motivated by Baptiste's algorithm for uniform job sizes [3]. Our case is more complicated, and the result is based on a bound on the number of small jobs that can "overtake" a larger job in an optimal schedule.

Throughout this paper we assume without loss of generality that all the time parameters are integer.

2 A 1.582 Approximation Algorithm for JISP

In this section we present a polynomial time $(e/(e-1)+\epsilon)$-approximation algorithm for JISP, where $\epsilon > 0$ is an arbitrary constant. The main idea of the algorithm is a partition of the time line into blocks. We use several iterations of the greedy algorithm to compute a partition that allows us to discard job intervals that cross block boundaries without losing too many jobs. Moreover, we are able to estimate the number of jobs in each block. We deal separately with blocks that contain a large number of jobs in an optimal solution. For the other blocks, we generate an LP relaxation to the scheduling problem by enumerating over all feasible schedules in each block. The advantage of this LP is that the fractional schedules for the blocks can be combined without overlap, unlike the fractional schedules for individual jobs.

Put $k = \lceil \frac{6}{\epsilon} \rceil$, let S be the input set of jobs, and let T be the maximum finish time of a job in S (the time horizon). The algorithm works in two phases. In the first phase, we divide the time line $[0, T]$ into blocks and schedule jobs in some of the blocks. In the second phase, we schedule at most $4k^{k \ln k+3}$ jobs in each of the remaining blocks. Every scheduled job (in both phases) is completely contained in a single block. The analysis of the algorithm depends on the fact that these added constraints do not reduce the optimal solution by much. Therefore, we must perform the partition into blocks carefully. Throughout the analysis of the algorithm, we fix an arbitrary optimal solution OPT. Abusing notation, we us OPT to denote both the optimal schedule and the set of jobs used in this schedule.

We begin with the description of the first phase. At the end of the phase, we have a partition of the time line into blocks. In some of the blocks, we determine the schedule in the first phase. We also compute a set S_{pass} of jobs to be scheduled in the second phase. Let S^I denote the set of jobs scheduled in the first phase, and let B^I denote the set of blocks where the jobs from S^I are scheduled. Let B^{II} be the set of the remaining blocks. In the first phase, we perform at most $k \ln k + 1$ iterations. The first iteration is slightly different from the others. Its purpose is to compute an initial partition into blocks. In each of the following iterations we refine the partition into blocks from the previous iteration. In the second phase, we schedule a set of jobs $S^{II} \subset S_{pass} \subset S \setminus S^I$ in the blocks from B^{II}. Given a partition B of the time line into blocks, let OPT_B be an optimal schedule under the constraint that no job may cross the boundary of a block in B.

The first iteration: In the first iteration we run GREEDY. Denote by S_1 the set of jobs that are scheduled by GREEDY. Using the schedule produced by GREEDY, we partition the time line into blocks, each containing k^3 jobs that GREEDY scheduled. (Notice that the last block might have fewer jobs, and its endpoint is the time horizon T.) We denote this partition into blocks by B_1.

Lemma 1. $|\mathrm{OPT}_{B_1}| \geq (1 - 1/k^3)|\mathrm{OPT}|$.

Proof: In each block OPT might schedule at most one job that crosses the right boundary of the block. In fact, this cannot happen in the last block, as it extends to the time horizon. Thus, the number of jobs eliminated from OPT by the partition into blocks is at most $\lceil |S_1|/k^3 \rceil - 1$. However, $|\mathrm{OPT}| \geq |S_1|$. □

Lemma 2. In each block computed by the above iteration, OPT schedules at most k^3 jobs from $R_1 = S \setminus S_1$.

Proof: The lemma follows from the existence of a one-to-one mapping of unscheduled optimal jobs to GREEDY-scheduled jobs. Each unscheduled optimal job is mapped to a unique overlaping GREEDY-scheduled job that prevented it from being scheduled, because it has an earlier finish time. □

The partition after the first iteration does not harm the optimal solution too much, as Lemma 1 states. However, by Lemma 2, OPT may schedule as many as twice the number of jobs that were scheduled by GREEDY. To do that, OPT might schedule a very large number of jobs from S_1 in some blocks. We must identify these blocks and further partition them. This is the purpose of the following iterations. In the following iterations we only refine the existing partition into blocks. Thus, Lemma 2 holds for the block partition throughout the first phase.

The ith iteration: The input to the ith iteration is the set of jobs S_{i-1} that was scheduled in the previous iteration, and the previous iteration's partition B_{i-1} into blocks. The output is a schedule for a subset of jobs $S_i \subset S_{i-1}$, and a new partition into blocks B_i that refines the input partition. Implicitly, a set $R_i = S_{i-1} \setminus S_i$ of unscheduled jobs is defined and used in the analysis. To compute the new schedule, we run GREEDY on S_{i-1}, disallowing job intervals that cross block boundaries. Whenever we complete the schedule of a block, we check how many jobs were scheduled in the block. If more than k^{i+2} jobs are scheduled, we partition the block into smaller blocks, each containing k^{i+2} scheduled jobs (except, perhaps, the last) and then proceed with GREEDY. Otherwise, we empty the block and proceed with GREEDY. (Notice that jobs from the emptied block can now be scheduled in a later block.) Let S_i denote the set of jobs that get scheduled eventually by this process and B_i the new partition into blocks.

Lemma 3. $|\text{OPT}_{B_i}| \geq (1 - i/k^3)|\text{OPT}|$.

Proof: In every iteration j, for $1 \leq j \leq i$, the number of new blocks increases by at most $\frac{|S_j|}{k^{j+2}} \leq \frac{|S_j|}{k^3}$. Each block eliminates at most one job from OPT (the job that crosses the block's right boundary, if such a job exists). Since there is a feasible solution containing all the jobs from S_j (the one computed in iteration j), $|\text{OPT}| \geq |S_j|$. In total, the number of jobs eliminated from the optimal solution by the iterations $1, \ldots, i$ is at most $\Sigma_{j=1}^{i} \frac{|S_j|}{k^3} \leq \frac{i}{k^3}|\text{OPT}|$. So there is a feasible solution of jobs inside the blocks from B_i, containing at least $\left(1 - \frac{i}{k^3}\right)|\text{OPT}|$ jobs. □

Lemma 4. In each block computed by the above iteration, OPT schedules at most $2k^{i+2}$ jobs from $R_i = S_{i-1} \setminus S_i$.

Proof: Consider a block b from B_i. All the jobs from R_i were available when GREEDY tried to schedule jobs in this block, as none of these jobs are scheduled in any other block. In both cases, whether the block b was emptied by GREEDY, or it was created by partitioning some block from the previous iteration, GREEDY can schedule at most k^{i+2} jobs in this block. As there is a one-to-one correspondence between the unscheduled jobs from R_i and the jobs scheduled by GREEDY in block b, at most $2k^{i+2}$ jobs from R_i can be scheduled in block b. □

Stopping condition: We terminate the first phase if $|S_i| \geq \left(1 - \frac{1}{k}\right)|S_{i-1}|$ or after the completion of iteration $k \ln k + 1$. In the former case, if $|S_i| \geq \left(1 - \frac{1}{k}\right)|S_{i-1}|$, we discard the block refinement of the last iteration, i.e., we set $B_i = B_{i-1}$. We set $S^I = S_i$, B^I is the set of blocks where the jobs from S^I are scheduled, $S_{pass} = S \setminus S_{i-1}$, and $B^{II} = B_i \setminus B^I$. In the latter case, we set $S^I = \emptyset$, $B^I = \emptyset$, $S_{pass} = S \setminus S_{(k \ln k+1)}$, $B^{II} = B_{(k \ln k+1)}$ and we remove the jobs in $S_{(k \ln k+1)}$ from the schedule. We sometimes refer to the blocks from B^{II} as empty blocks.

Lemma 5. Let r denote the number of iterations in the first phase. Then $|\text{OPT}_{B_r}| \geq \left(1 - \frac{1}{k}\right)|\text{OPT}|$.

Proof: Since $r \leq k \ln k + 1$, by Lemma 3, $|\text{OPT}_{B_r}| \geq \left(1 - \frac{k \ln k+1}{k^3}\right)|\text{OPT}| \geq \left(1 - \frac{1}{k}\right)|\text{OPT}|$. □

Let J be any set of jobs. We denote by $\text{OPT}_{B^{II}}(J)$ the best schedule of jobs from J in blocks from B^{II}.

Lemma 6. $|S_1| + |\text{OPT}_{B^{II}}(S_{pass})| \geq (1 - \epsilon)|\text{OPT}|$.

Proof: Consider two cases.

Case 1: If the first phase terminates after $(k \ln k + 1)$ iterations, then for all $1 < i \leq k \ln k$, $|S_i| \leq \left(1 - \frac{1}{k}\right)|S_{i-1}|$, and $|S_{(k \ln k+1)}| \leq \left(1 - \frac{1}{k}\right)^{k \ln k}|S_1| \leq \frac{|S_1|}{k} \leq \frac{|\text{OPT}|}{k}$. As $S_{pass} = S \setminus S_{(k \ln k+1)}$, $B^{II} = B_{(k \ln k+1)}$, and using the fact that, by Lemma 5, $\text{OPT}_{(B_{k \ln k+1})} \geq \left(1 - \frac{1}{k}\right)|\text{OPT}|$, we get that $|\text{OPT}_{B^{II}}(S_{pass})| = |\text{OPT}_{B^{II}}(S \setminus S_{(k \ln k+1)})| \geq |\text{OPT}_{B^{II}}(S)| - |S_{(k \ln k+1)}| \geq \left(1 - \frac{1}{k}\right)|\text{OPT}| - \frac{1}{k}|\text{OPT}| = \left(1 - \frac{2}{k}\right)|\text{OPT}| > (1 - \epsilon)|\text{OPT}|$.

Case 2: If the algorithm terminated after iteration $r < k \ln k + 1$, then $S^I = S_r$, and $S_{pass} = S \setminus S_{r-1}$. We have $|\text{OPT}_{B^{II}}(S_{pass})| \geq |\text{OPT}_{B_r}(S_{pass})| - |\text{OPT}_{B^I}(S_{pass})| = |\text{OPT}_{B_r}(S \setminus S_{r-1})| - |\text{OPT}_{B^I}(S_{pass})| \geq |\text{OPT}_{B_r}(S)| - |S_{r-1}| - |\text{OPT}_{B^I}(S_{pass})|$. Thus,

$$|S^I| + |\text{OPT}_{B^{II}}(S_{pass})| \geq$$

$$\geq |\text{OPT}_{B_r}(S)| - (|S_{r-1}| - |S_r|) - |\text{OPT}_{B^I}(S_{pass})|. \quad (1)$$

We bound each of the terms separately. By Lemma 5, $|\text{OPT}_{B_r}(S)| \geq \left(1 - \frac{1}{k}\right)|\text{OPT}|$. As the algorithm finished before the iteration $k \ln k + 1$, $|S_r| \geq \left(1 - \frac{1}{k}\right)|S_{r-1}|$, so $|S_{r-1}| - |S_r| \leq \frac{1}{k}|S_{r-1}| \leq \frac{1}{k}|\text{OPT}|$. Finally, observe that $S_{pass} = S \setminus S_{r-1} = R_1 \cup R_2 \cup \ldots \cup R_{r-1}$. Let b be some block in B^I. By Lemma 4 and the fact that B_r is a refinement of the block partitions of the previous iterations, at most $2k^{i+2}$ jobs from set R_i can be scheduled in block b, for all $1 \leq i \leq r - 1$. Thus, at most $\Sigma_{i=1}^{r-1} 2k^{i+2} \leq 4k^{r+1}$ jobs from S_{pass} can be scheduled in b. On the other hand, we know that at least k^{r+2} jobs from S^I are scheduled in b in iteration r. Thus, $|\text{OPT}_{B^I}(S_{pass})| \le \frac{4}{k}|S^I| \le \frac{4}{k}|\text{OPT}|$. Substituting these bounds into (1), we get

$$|S^I| + |\text{OPT}_{B^{II}}(S_{pass})| \ge$$
$$\ge \left(1 - \frac{1}{k}\right)|\text{OPT}| - \frac{1}{k}|\text{OPT}| - \frac{4}{k}|\text{OPT}| =$$
$$= \left(1 - \frac{6}{k}\right)|\text{OPT}| = (1-\epsilon)|\text{OPT}|. \qquad \square$$

Lemma 7. In every empty block OPT schedules less than $4k^{k\ln k+3}$ jobs from S_{pass}.

Proof: The lemma follows from Lemmas 2 and 4. Every empty block is completely contained in a single block in each of the previous iterations. The jobs from S_{pass} that OPT schedules in an empty block are contained in the sets $R_1, R_2, \ldots, R_j$, where j is the last iteration. Thus, the number of jobs OPT schedules in an empty block is less than $2\sum_{i=1}^{j} k^{i+2} < 4k^{j+2} \le 4k^{k\ln k+3}$. $\square$

Notice that the number of blocks at the end of the first phase is polynomial in n. In each iteration we create at most n new blocks, and there are at most $k\ln k + 1$ iterations.

We now proceed with the description of the second phase of the algorithm. The input to this phase is the final partition into blocks that was computed in the previous phase (where each block is marked as empty or not), and the set S_{pass} of jobs yet to be scheduled. Let S_{optp} denote the set of jobs from S_{pass} that OPT schedules in empty blocks. We define an integer program that computes the best schedule of jobs from S_{pass} in empty blocks. The number of jobs scheduled in the integer program is clearly an upper bound on $|S_{optp}|$. We then use the integer program's linear programming relaxation to compute an approximate solution. Let B denote the set of empty blocks. By Lemma 7, for every block $b \in B$, OPT schedules at most $4k^{k\ln k+3}$ jobs from S_{pass} in b. Given an ordered set of at most $4k^{k\ln k+3}$ jobs, it is easy to schedule the jobs in b in that order, if such a schedule exists: Scan the jobs from first to last, and place each job in its turn as early as possible inside the block b. Thus, the number of possible schedules in b for jobs from S_{pass} is at most the number of ordered sets of jobs of size at most $4k^{k\ln k+3}$, which is

$$\sum_{s=0}^{4k^{k\ln k+3}} s!\binom{n}{s} = n^{\exp(k\ln^2 k)}.$$

Let $M(b)$ denote the set of all such schedules for block $b \in B$. The integer program contains, for every block $b \in B$, and for every schedule $m \in M(b)$, a variable $y^b_m \in \{0,1\}$. Setting $y^b_m = 1$ means that the schedule m is chosen for the block b. The integer program that computes an upper bound on S_{optp} is the following:

$$\text{maximize} \quad \sum_{b\in B}\sum_{m\in M(b)}\sum_{j\in m} y^b_m \quad \text{subject to}$$

$$\begin{array}{ll} \sum_{b\in B}\sum_{m\in M(b)|j\in m} y^b_m \le 1 & \forall j \in S_{pass} \\ \sum_{m\in M(b)} y^b_m = 1 & \forall b \in B \\ y^b_m \in \{0,1\} & \forall b \in B,\ \forall m \in M(b). \end{array}$$

The first set of constraints makes sure that each job is scheduled at most once, and the second set of constraints makes sure that a unique feasible schedule is chosen for every empty block. The linear programming relaxation is derived by replacing the last set of constraints with the constraints $y \ge 0$. Denote the resulting linear program by LP. Let y be a feasible solution to LP. We round y to an integer solution y_{int} using the following two steps algorithm:

1. In every block $b \in B$, choose at random, independently of the choice in other blocks, a schedule $m \in M(b)$ with distribution y^b (i.e., schedule $m \in M(b)$ is chosen with probability y^b_m).

2. For every job $j \in S_{pass}$, if more than one block has a schedule containing j as a result of the previous step, remove j from all schedules containing it except one, chosen arbitrarily.

For every job $j \in S_{pass}$ and for every block $b \in B$, put $x^b_j = \sum_{m\in M(b)|j\in m} y^b_m$, and put $x_j = \sum_{b\in B} x^b_j$. Clearly, the value of the solution y is $z = \sum_{j\in S_{pass}} x_j$. Let p_j be the probability that j is scheduled in y_{int}, and let z_{int} be the value of the solution y_{int}. (Both y_{int} and z_{int} are random variables.)

Lemma 8. For every job $j \in S_{pass}$, $p_j \ge \left(1 - \frac{1}{e}\right)x_j$.

Proof: The probability that we do not schedule j is the probability that in no block a schedule containing j was chosen, which is $\prod_b \left(1 - x^b_j\right)$. Let t be the number of blocks where a schedule containing j appears with positive probability in y. The product is maximized when in each such block b, $x^b_j = x_j/t$. Thus, $p_j = 1 - \prod_b \left(1 - x^b_j\right) \ge 1 - (1 - x_j/t)^t$. Therefore, $p_j/x_j \ge \left(1 - (1 - x_j/t)^t\right)/x_j$. The right-hand side is monotonically decreasing in x_j, and thus the minimum is achieved at $x_j = 1$. We conclude that $p_j/x_j \ge 1 - (1 - 1/t)^t \ge 1 - \frac{1}{e}$. This completes the proof of the lemma. $\square$

Corollary 9. $E[z_{int}] \ge \left(1 - \frac{1}{e}\right)z$.

Proof: $E[z_{int}] = \sum_{j\in S_{pass}} p_j \ge \sum_{j\in S_{pass}} \left(1 - \frac{1}{e}\right)x_j = \left(1 - \frac{1}{e}\right)z$. $\square$

We can now state and prove the main theorem in this section:

Theorem 10. For every $\epsilon > 0$, the above two-phase algorithm runs in polynomial time and guarantees, in expectation, an $e/(e-1)+\epsilon$ approximation to JISP.

Proof: Let S^{II} be the set of jobs scheduled by rounding the optimal solution y^* to LP. Let z^* be the value of y^*. The expected value of the solution produced by the algorithm is $E[|S^I|+|S^{II}|] = |S^I|+E[|S^{II}|]$. By Corollary 9, $E[|S^{II}|] \geq \left(1-\frac{1}{e}\right)z^* \geq \left(1-\frac{1}{e}\right)|\text{OPT}_{B^{II}}(S_{pass})|$.

As $|S^I| + |\text{OPT}_{B^{II}}(S_{pass})| \geq (1-\epsilon)|\text{OPT}|$ (by Lemma 6), the expected value of the solution is $|S^I| + E[|S^{II}|] \geq |S^I| + \left(1-\frac{1}{e}\right)|\text{OPT}_{B^{II}}(S_{pass})| \geq \left(1-\frac{1}{e}\right)(|S^I|+|\text{OPT}_{B^{II}}(S_{pass})|) \geq \left(1-\frac{1}{e}\right)(1-\epsilon)|\text{OPT}| \geq \left(1-\frac{1}{e}-\epsilon\right)|\text{OPT}|$. □

3 Jobs with Small Windows

In this section we give a dynamic programming algorithm that computes an optimal solution for instances of $1|r_j|\sum w_j\overline{U_j}$. Let $T = \max_{j\in S} d_j$ denote the time horizon. The running time of our algorithm is polynomial in $n = |S|$ and in T, and is exponential in poly(k), where $k = \max_{j\in S}(d_j - r_j)/p_j$. Thus, if for every job $j \in S$, its window size $d_j - r_j$ is at most a constant factor times its processing time p_j, we get a pseudo-polynomial time algorithm.

Let $S = \{1, 2, \ldots, n\}$ be the set of jobs, sorted in non-decreasing order of processing times, ties broken arbitrarily. Let Release$_j(s,e) = \{i \leq j \mid r_i \in [s,e)\}$. The dynamic program computes the entries $D(s,x,e,j,\text{IN},\text{OUT})$, where $s \leq x < e$ are integers (points on the time line), $j \in S$, and IN, OUT are subsets of S of size at most k^2. We require the following conditions on the sets of jobs IN and OUT:

- $\text{IN} \bigcap \text{OUT} = \emptyset$.
- $\text{OUT} \subseteq \text{Release}_j(s,e)$, and all the jobs in OUT can be scheduled after time e (as their release dates are before e, this condition can be checked by using the EDD rule).
- $\text{IN} \subseteq \text{Release}_j(0,s)$, and all the jobs in IN can be scheduled after time x (this also can be checked using EDD).

The value stored in $D(s,x,e,j,\text{IN},\text{OUT})$ is an optimal schedule of jobs from the set $\text{Release}_j(s,e)\bigcup \text{IN} \setminus \text{OUT}$ in the time interval $[x,e)$. The output of the algorithm is the entry $D(0,0,T,n,\emptyset,\emptyset)$.

We compute the entries of D in increasing order of j. For $j = 0$, for all $s,x,e,\text{IN},\text{OUT}$, $D(s,x,e,0,\text{IN},\text{OUT})$ is the empty schedule. Inductively, the algorithm computes $D(s,x,e,j,\text{IN},\text{OUT})$ as follows: If $j \notin \text{Release}_j(s,e)\bigcup \text{IN} \setminus \text{OUT}$, set $D(s,x,e,j,\text{IN},\text{OUT}) = D(s,x,e,j-1,\text{IN}\setminus\{j\},\text{OUT}\setminus\{j\})$. Otherwise, enumerate over all feasible placements of j in the interval $[x, e-p_j]$. For each such placement t, compute an optimal schedule S_t as explained below. Finally, set $D(s,x,e,j,\text{IN},\text{OUT})$ to be the best schedule among $D(s,x,e,j-1,\text{IN}\setminus\{j\},\text{OUT}\setminus\{j\})$ and S_t, for all t.

It remains to show how to compute S_t. If we schedule job j starting at time t, then the scheduling problem of $D(s,x,e,j,\text{IN},\text{OUT})$ is split into two subproblems on the intervals $[s,t)$ and $[t,e)$. Thus, S_t is the union of the schedules $D(s,x,t,j-1,E,F)$, J, and $D(t,t+p_j,e,j-1,G,H)$, for some sets E,F,G,H, where J is the schedule containing just the job j placed starting at t. To enumerate over the relevant choices for E,F,G,H all we have to do is to decide which jobs with release date before t are scheduled after j. We partition the set OUT into two sets of jobs, those with release dates before t and those with release dates after t. Let $B_1 = \text{OUT}\bigcap \text{Release}_{j-1}(s,t)$ and let $B_2 = \text{OUT}\bigcap \text{Release}_{j-1}(t,e)$. For every partition of $\text{IN} \setminus \{j\}$ into A_1 and A_2, and for every $B \subseteq \text{Release}_{j-1}(s,t) \setminus B_1$, such that $A_2 \bigcup B$ can be scheduled after time $t+p_j$ and $B_1 \bigcup B$ can be scheduled after time $t+p_j$, set $E = A_1$, $F = B_1\bigcup B$, $G = A_2 \bigcup B$, and $H = B_2$. (Below, we prove that these settings satisfy the conditions on the indices of the table D.) We set S_t to be the schedule for the best such partition of IN and choice of B. This completes the description of the dynamic program.

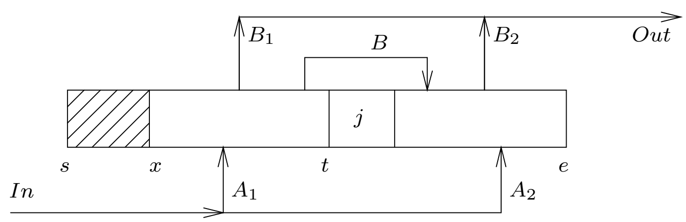

Figure 1. Computation of S_t

We now proceed with the analysis of the algorithm. We begin the analysis with an observation on the structure of optimal solutions. Consider an optimal solution OPT. For every job j scheduled in OPT, let t_j denote the starting time of j in OPT. Let $B(j) = \{i < j \mid r_i < t_j \text{ and } t_i > t_j\}$.

Lemma 11. For every $j \in S$, $|B(j)| \leq k^2$.

Proof: Let $i \in B(j)$. As jobs are sorted by their processing times, $p_i \leq p_j$. On the other hand, $p_i > p_j/k$, otherwise the job j is longer than the window of i, in contradiction with the assumption that $r_i < t_j$ whereas $t_i > t_j$. Consider the job $i \in B(j)$ with maximum t_i. All the jobs in $B(j)$ are scheduled by OPT inside $[r_i, d_i]$. By our discussion, $d_i - r_i \leq kp_j$. By the lower bound on the processing time of the jobs in $B(j)$, at most k^2 such jobs fit in this interval. □

Remark: A tighter analysis gives a bound of $O(k \log k)$.

Lemma 12. Every choice for the sets E, F, G, H considered by the above algorithm satisfies the following conditions: Each of the sets contains at most k^2 jobs, and $D(s, x, t, j-1, E, F)$, $D(t, t+p_j, e, j-1, G, H)$ are valid entries of D.

The proof of this lemma is easy following the above discussion, and is omitted from this extended abstract.

Lemma 13. The schedule $D(s, x, e, j, \text{IN}, \text{OUT})$ computed by the algorithm is a feasible schedule of jobs from $\text{Release}_j(s, e) \bigcup \text{IN} \setminus \text{OUT}$ in the time interval $[x, e)$.

Proof: The proof is by induction on j. The empty schedule $D(s, x, e, 0, \text{IN}, \text{OUT})$ is clearly feasible. Consider the schedule $D(s, x, e, j, \text{IN}, \text{OUT})$. Job j is scheduled only if it belongs to $\text{Release}_j(s, e) \bigcup \text{IN} \setminus \text{OUT}$. If j is scheduled, it starts at some time $t \in [x, e - p_j)$ inside its time window, so its own schedule is feasible. If j is not scheduled, the schedule is $D(s, x, e, j-1, \text{IN} \setminus \{j\}, \text{OUT} \setminus \{j\})$, which is feasible by the induction hypothesis. If j is scheduled at time t, the schedule we return is the union of j's schedule, $D(s, x, t, j-1, E, F)$, and $D(t, t+p_j, e, j-1, G, H)$. We argue that the sets of jobs used in these schedules are distinct, so no job is scheduled twice. (Clearly, the schedules do not overlap.) This follows from the fact that the sets $\text{Release}_{j-1}(s, t)$, $\text{Release}_{j-1}(t, e)$, A_1, and A_2 are all distinct, and the jobs in B are considered only in the computation of $D(t, t+p_j, e, j-1, G, H)$. $\square$

Lemma 14. The schedule $D(s, x, e, j, \text{IN}, \text{OUT})$ is computed correctly.

Proof: The proof is by induction on j. Clearly, the lemma is true for $j = 0$. Now consider an optimal schedule $\text{OPT}(s, x, e, j, \text{IN}, \text{OUT})$ of jobs from $\text{Release}_j(s, e) \bigcup \text{IN} \setminus \text{OUT}$ in the time interval $[x, e)$. If j is not scheduled in this solution, then by induction this optimal schedule has the same profit as $D(s, x, e, j-1, \text{IN} \setminus \{j\}, \text{OUT} \setminus \{j\})$, which is one of the schedules checked by the algorithm in the computation of $D(s, x, e, j, \text{IN}, \text{OUT})$. So assume that j is scheduled in $\text{OPT}(s, x, e, j, \text{IN}, \text{OUT})$ starting at time t. Let $B_1 = \text{OUT} \bigcap \text{Release}_{j-1}(s, t)$ and let $B_2 = \text{OUT} \bigcap \text{Release}_{j-1}(t, e)$. Let A_2 be the subset of IN scheduled in $\text{OPT}(s, x, e, j, \text{IN}, \text{OUT})$ after time t, and let $A_1 = \text{IN} \setminus (A_2 \bigcap \{j\})$. Let B be the subset of jobs from $\text{Release}_{j-1}(s, t) \setminus B_2$ scheduled in $\text{OPT}(s, x, e, j, \text{IN}, \text{OUT})$ after job j. Then, by the induction hypothesis, the schedule considered by the algorithm for $E = A_1$, $F = B_1 \bigcup B$, $G = A_2 \bigcup B$, and $H = B_2$ is as good as $\text{OPT}(s, x, e, j, \text{IN}, \text{OUT})$. $\square$

We conclude

Theorem 15. The dynamic programming algorithm computes an optimal schedule in time $O\left(n^{poly(k)} T^4\right)$.

Proof: The correctness of the algorithm follows from Lemmas 13 and 14. The number of entries in the dynamic programming table D is $O\left(T^3 n \binom{n}{k^2}^2\right)$. To compute an entry $D(s, x, e, j, \text{IN}, \text{OUT})$, we have to check at most T possible placements of job j. For each such placement, there are at most 2^{k^2} possible partitions of IN, and $\binom{n}{k^2}$ choices of B. For each such partition of IN and choice of B, we have to run EDD several times. This takes at most $O(n \log n)$ time. So the time complexity of the algorithm is $O\left(T^4 n^2 \log n \binom{n}{k^2}^3 2^{k^2}\right) \le O\left(T^4 2^{k^2} n^{3k^2+2} \log n\right)$. $\square$

4 A PTAS for Instances with Restrictions on Job Processing Times

In this section we present a polynomial time approximation scheme for $1|r_j|\sum w_j \overline{U}_j$ under the following assumptions: Job processing times can take one of a constant number c of possible values, and the ratio r between the maximum possible value P and the minimum possible value p is upper bounded by a constant.

Set $\epsilon > 0$, and put $k = \left\lceil \frac{8r(2c+1)}{\epsilon} \right\rceil + 2$. Let x be an integer, to be specified later. Let $f_1, f_2, \ldots, f_l$ be independent, uniformly distributed, random variables with $f_1 \in \{1, \ldots, Pk^{2k+2x}\}$, and for every $i = 2, 3, \ldots, l$, $f_i \in \{1, \ldots, k^{2k}\}$. We divide the time line $\{0, 1, 2, \ldots, T\}$ (where $T = \max_j d_j$ is the time horizon) into blocks in l levels. Level-1 blocks are special. The first level-1 block ends at time f_1, and the other blocks are identical, each of length Pk^{2k+2x}. For $i = 2, 3, \ldots, l$, the first level-i block ends at the end of the f_i level-$(i-1)$ block, and the other level-i blocks are identical, each of length k^{2k} level-$(i-1)$ blocks. We set $l = O(\log T)$.

We place some of the jobs in sets S_i, $i = 1, 2, \ldots, l$ according to their windows sizes. Let $t_j = d_j - r_j$ be the window size of job j. We set $S_1(x) = \{j \in S \mid 1 \le t_j \le Pk^{2k+2x-1}\}$, and, for $i = 2, 3, \ldots, l$, $S_i(x) = \{j \in S \mid Pk^{2(i-1)k+2x+1} \le t_j \le Pk^{2ik+2x-1}\}$. Let $U(x) = \bigcup_i S_i(x)$. Let OPT denote an optimal schedule, and let $\text{OPT}_B(U(x))$ denote an optimal schedule of jobs in $U(x)$ under the following constraints: No job is scheduled so that it crosses the boundary of a level-1 block, and no job in $S_i(x)$ is scheduled unless its window is contained in a level-i block. (A schedule that satisfies these constraints is called a schedule within blocks.)

Lemma 16. There is a choice of $x \in \{0, 1, \ldots, k-1, k\}$ such that $E[w(\text{OPT}_B(U(x)))] \ge \left(1 - \frac{3}{k}\right) w(\text{OPT}(S))$. (The expectation is over the choice of $f_1, f_2, \ldots, f_l$.)

Proof: Consider the sets $\overline{U}(x)$ for $x = 0, 1, \ldots, k-1$, defined as $\overline{U}(x) = \{j \in S \mid \exists i \ge 1 \text{ s.t. } Pk^{2ik+2x-1} <$

$t_j < Pk^{2ik+2x+1}\}$. All these sets are disjoint, and $U(x) = S \setminus \overline{U}(x)$. Therefore, there exists x such that $w(U(x) \bigcap \text{OPT}(S)) = w(\text{OPT}(S)) - w(\overline{U}(x) \bigcap \text{OPT}(S)) \geq \left(1 - \frac{1}{k}\right) w(\text{OPT}(S))$. So consider the schedule $\text{OPT}(U(x))$. For every scheduled job, the probability that it crosses the boundary of a level-1 block is at most $P/Pk^{2k+2x} = 1/k^{2k+2x} \leq 1/k$. Finally, consider a job $j \in S_i$, for some $i = 1, 2, \ldots, l$. The probability that j's window crosses the boundary of a level-i block is at most $Pk^{2ik+2x-1}/Pk^{2ik+2x} = 1/k$. □

Notice that we can find x by enumerating over all possible values. From now on, let x be the correct choice. Next, we modify the jobs in $U(x)$ as follows: For every $i = 2, 3, \ldots, l$, for every $j \in S_i(x)$, we shrink the window of j so that its release date and deadline are aligned with level-$(i-1)$ block boundaries. Let $\tilde{U}(x)$ denote the resulting instance, and let $\text{OPT}_B(\tilde{U}(x))$ denote an optimal schedule within blocks (as defined above) for $\tilde{U}(x)$.

Lemma 17.
$$w(\text{OPT}_B(\tilde{U}(x))) \geq \left(1 - \tfrac{8r}{k-2}\right)^{2c} \cdot w(\text{OPT}_B(U(x))).$$

Proof: We convert the schedule $\text{OPT}_B(U(x))$ into a feasible schedule within blocks for $\tilde{U}(x)$ as follows: For each of the c different job processing times, we process jobs j with that processing time in order from right to left. If $t_j \leq Pk^{2k+2x}$, we do nothing. Otherwise, if j is scheduled in the first t_j/k time slots in its window, we try to move it forward in its window beyond that point, but not to the last t_j/k time slots in its window. If there is no room for j, we remove it from the schedule. After completing this process, we execute a symmetric process, scanning jobs from left to right and removing them from the last t_j/k time slots in their schedule. Clearly, the resulting schedule is feasible within blocks, and the only question is how many jobs were removed. We analyze a pass in a single direction for a single job processing time value. The lemma follows by combining the bounds for all $2c$ passes. We proceed by induction on the length of the schedule. Consider the last job j that is removed from the schedule. Let I be the time interval between j's position in the schedule and $d_j - \frac{t_j}{k}$. The jobs with processing time p_j that were scheduled in I and previously removed must have been scheduled in the first $\frac{t_j}{k}$ time slots of j's window (otherwise we could move j forward). Moreover, the rest of j's window must be full enough to prevent us from moving j there. Thus, there are at most $\lceil \frac{t_j}{kp} \rceil \leq \frac{t_j}{kp} + 1$ jobs of size p_j that were removed from I and at least $\lceil \frac{t_j(1-\frac{2}{k})}{2P} \rceil \geq \frac{t_j(1-\frac{2}{k})}{2P} - 1$ jobs that remain scheduled in I. So we removed from I at most a fraction of

$$\frac{\frac{t_j}{kp} + 1}{\frac{t_j(1-\frac{2}{k})}{2P} - 1} \leq \frac{2\frac{t_j}{kp}}{\frac{1}{2} t_j \frac{k-2}{2kP}} = \frac{8r}{k-2}$$

jobs. All the removed jobs in I are accounted for, so we can use the induction hypothesis on the remaining schedule beyond I. □

Next we describe a dynamic programming algorithm that computes an optimal solution within blocks for $\tilde{U}(x)$. We compute the entries $D(i, b, v)$, where i is a level number, b is a block number within level i, and v is a vector of c integers, each entry corresponding to one of the allowed job processing times. The value stored in $D(i, b, v)$ is an optimal solution within blocks for the level-i block number b, using the jobs from $S_1(x), \ldots, S_i(x)$ whose windows are completely contained in block b , plus v_q additional jobs with processing time q for all q, that can be placed anywhere in block b. We compute the entries $D(i, b, v)$ by induction on i. For $i = 1$, we have blocks of length Pk^{2k+2x}, so a block may contain at most rk^{2k+2x} jobs (real ones or "virtual" ones from v). We can enumerate over all possible schedules in polynomial time. The inductive step is done as follows. Consider a level-i block and the jobs from $S_i(x)$ whose window is contained in that block. In $\tilde{U}(x)$ these windows are aligned with level-$(i-1)$ block boundaries, so there are at most ck^{4k} job types, where the type of a job is its processing time and window. We compute $D(i, b, v)$ using another dynamic programming procedure that computes the entries $E(l, u)$. The value of $E(l, u)$ is the best schedule for the first l level-$(i-1)$ blocks inside block b, using u_t jobs of type t, for every possible type t (u has to match v at the end). We compute E by induction on l. The initial values are $E(1, u) = D(i-1, b', v')$, where b' is the first level-$(i-1)$ block inside block b, and v' matches u. The inductive step sets $E(l, u)$ as the best choice, for all integer vectors u', $0 \leq u' \leq u$, of the union of $E(l-1, u') + D(i-1, b+l-1, v')$, where v' matches $u - u'$. We have

Lemma 18. The above algorithm runs in polynomial time and computes an optimal schedule within blocks for $\tilde{U}(x)$.

Proof sketch: There are $O(\log T)$ levels, at most n blocks containing a job in each level, and at most n^{c+1} different counter vectors v, so the number of entries of D is bounded by a polynomial. By the above discussion, the computation of each entry takes polynomial time. □

Let SCHED be the schedule computed by the above algorithm. Lemmas 16, 17, and 18, and the choice of k, trivially imply

Theorem 19. $w(\text{SCHED}) \geq (1-\epsilon) w(\text{OPT})$.

References

[1] M. Adler, A.L. Rosenberg, R.K. Sitaraman, and W. Unger. Scheduling time-constrained communication in linear networks. In *Proc. 10th Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 269–278, 1998.

[2] E.M. Arkin and E.B. Silverberg. Scheduling jobs with fixed start and end times *Discrete Applied Mathematics*, 18:1–8, 1987.

[3] P. Baptiste. Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times. *Journal of Scheduling*, 2:245–252, 1999.

[4] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber. A unified approach to approximating resource allocation and scheduling. In *Proc. 32nd Ann. ACM Symp. on Theory of Computing*, May 2000.

[5] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber. Approximating the throughput of multiple machines in real-time scheduling. In *Proc. 31st Ann. ACM Symp. on Theory of Computing*, pages 622–631, May 1999.

[6] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. *Real-Time Systems*, 4:125–144, 1992. Preliminary version appeared in *FOCS '91*.

[7] P. Berman and B. DasGupta. Improvements in throughput maximiztion for real-time scheduling. In *Proc. 32nd Ann. ACM Symp. on Theory of Computing*, May 2000.

[8] J. Carlier. Probléme à une machine et algorithmes polynômiaux. *Questio*, 5(4):219–228, 1981.

[9] Y. Crama, O.E. Flippo, J.J. van de Klundert, and F.C.R. Spieksma. The assembly of printed circuit boards: a case with multiple machines and multiple board types. *European Journal of Operations Research*, 98:457–472, 1997.

[10] U. Faigle and W.M. Nawijn. Note on scheduling intervals on-line. *Discrete Applied Mathematics*, 58:13–17, 1995.

[11] M. Fischetti, S. Martello, and P. Toth. The fixed job schedule problem with spread-time constraints. *Operations Research*, 35:849–858, 1987.

[12] A. Freund. Private Communication.

[13] M.R. Garey and D.S. Johnson. Two processor scheduling with start times and deadlines. *SIAM Journal on Computing*, 6:416–426, 1977.

[14] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, 1980.

[15] S.C. Graves, A.H.G. Rinnooy Kan, and P.H. Zipkin, editors. *Handbooks of Operations Research, Volume 4: Logistics for Production and Inventory*. North-Holland, 1993.

[16] N.G. Hall and M.J. Magazine. Maximizing the value of a space mission. *European Journal of Operations Research*, 78:224–241, 1994.

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

[18] H. Kise, T. Ibaraki, and H. Mine. A solvable case of one machine scheduling problem with ready and due dates. *Operations Research*, 26:121–126, 1978.

[19] G. Koren and D. Shasha. An optimal on-line scheduling algorithm for overloaded real-time systems. *SIAM Journal on Computing*, 24:318–339, 1995.

[20] E.L. Lawler. Sequencing to minimize the weighted number of tardy jobs. *Recherche Operationnel*, 10:27–33, 1976.

[21] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. Sequencing and Scheduling: Algorithms and Complexity. In [15].

[22] R.J. Lipton and A. Tomkins. Online interval scheduling. In *Proc. 5th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 302–311, 1994.

[23] H. Liu and M.E. Zarki, Adaptive source rate control for real-time wireless video transmission. *Mobile Networks and Applications*, 3:49–60, 1998.

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

[25] G.R. Rajugopal and R.H.M. Hafez. Adaptive rate controlled, robust video communication over packet wireless networks. *Mobile Networks and Applications*, 3:33–47, 1998.

[26] S. Sahni. Algorithms for scheduling independent tasks. *Journal of the ACM*, 23:116–127, 1976.

[27] F.C.R. Spieksma. On the approximability of an interval scheduling problem. *Journal of Scheduling*, 2:215–227, 1999.

[28] D.K.Y. Yau and S.S. Lam. Adaptive rate-controlled scheduling for multimedia applications. In *Proc. IS&T/SPIE Multimedia Computing and Networking Conf.*, January 1996.