MATHEMATICS OF OPERATIONS RESEARCH
Vol. 00, No. 0, Xxxxxx 20xx, pp. xxx–xxx
ISSN 0364-765X | EISSN 1526-5471 |xx|0000|0xxx
DOI 10.1287/moor.xxxx.xxxx

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

Julia Chuzhoy
CSAIL MIT, Cambridge, MA and Dept. of CIS, University of Pennsylvania, Philadelphia, PA.
email: cjulia@csail.mit.edu

Rafail Ostrovsky
3731 Boelter Hall, UCLA Computer Science Department, University of California, Los Angeles, CA 90095-1596, USA.
email: rafail@cs.ucla.edu

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

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.

Key words: scheduling; throughput; approximation algorithms; PTAS

MSC2000 Subject Classification: Primary: 68W25, 68Q25, 68W40; Secondary: 68W05

OR/MS subject classification: Primary: Analysis of algorithms; Secondary: suboptimal algorithms

History: Received: Xxxx xx, xxxx; revised: Yyyyyy yy, yyyy and Zzzzzz zz, zzzz.

Acknowledgments. A preliminary version of this paper appeared in Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science, October 2001, pages 348–356. Work done while the first author was a graduate student at the Computer Science Department at the Technion, and while the second author was affiliated with Telcordia Research. Part of this work was done while the second author was visiting the Technion, and while the third author was visiting Telcordia Research. The first author was supported in part by Yuval Rabani's grant from the CONSIST consortium. The third author was supported in part by ISF grant 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, and by the Fund for the Promotion of Research at the Technion.

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 [19], this

problem is equivalent to $1|r_j|\sum \overline{U}_j$. The generalizations of this special case of JISP 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*.

Special cases of JISP are used to model scheduling problems in numerous applications. Some examples include: selection of projects to be performed during a space mission [16],[1] placement of feeders in feeder racks of an assembly line for printed circuit boards [10, 25], time-constrained communication scheduling [1], and adaptive rate-controlled scheduling for multimedia applications [26, 21, 23]. These applications and others inspired the development of many heuristics for JISP or special cases of JISP, most of them lacking theoretical analysis (see the above-mentioned references).

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. In fact, the latter result holds 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 $1|r_j|\sum w_j\overline{U}_j$). The ideas we use in our 1.582-approximation algorithm for JISP can also be applied to the resource allocation problem, 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. We obtain a ratio arbitrarily close to $(2e-1)/(e-1) < 2.582$, improving on the best previously known approximation factor of 5 [4].

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 [22] 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 [24] 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 [25]). Much of the recent work on JISP variants extends this bound to more general settings. Indeed, Spieksma [25] 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 [25, 5]. As for hardness of approximation results, JISP2 is MAX SNP-hard [25]. Also, $R|r_j|\sum \overline{U}_j$ is MAX SNP-hard [5]. In both cases, the constant lower 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 [9], showed that

[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.

$1|p_j = p, r_j| \sum w_j \overline{U}_j$ (i.e., when all job processing times are equal) is in P.

There are also NP-hard special cases of JISP that were known to have better than 2 approximations. Spieksma [25] 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 of [4, 7]. They pointed out this improvement as a possible scheduling anomaly. Our results refute this possibility (at least in the unweighted case), 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, 20, 11, 18]).

Our methods. Our algorithms rely on proving special structural properties of optimal or near-optimal solutions. The main idea behind the approximation algorithm for JISP is computing a division of the time line into blocks, such that there exists a near-optimal solution, in which no job crosses block boundaries, and only a constant number of jobs is scheduled inside each block. We then use the above partition to write a better linear program, which enumerates over all the possible schedules inside each block. In fact, the partition produced by our algorithm has slightly weaker properties, but they are still sufficient for producing an improved linear program. The algorithm for JISP consists of two phases. In the first phase, we compute a partition of time line into blocks, and divide the blocks into two subsets B^I and B^{II}. The algorithm also produces a schedule of a subset S^I of jobs in blocks B^I. The heart of the algorithm analysis is showing that there is a near-optimal schedule with the following properties: (1) no job is crossing block boundaries, (2) the subset of jobs scheduled inside the blocks of B^I is exactly S^I, and (3) in each block $b \in B^{II}$, only a constant number of jobs is scheduled. The goal of the second phase is to find a near-optimal solution of the remaining jobs in subset B^{II} of blocks. The constant bound on the number of jobs to be scheduled inside each such block allows us to generate a new LP relaxation that leads to the improved approximation guarantee.

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 integral.

2. A 1.582 Approximation Algorithm for JISP The JISP is defined as follows. The input is a set J of n jobs, where for each job $j \in J$, a set $\mathcal{I}(j)$ of time intervals is specified. The sets $\mathcal{I}(j)$ of time intervals can be either given explicitly, or in other ways (for example, by listing the release date, the deadline and the processing time of a job). In order to schedule job $j \in J$, one of the intervals $I \in \mathcal{I}(j)$ must be chosen, and we say that job j is scheduled on interval I in this case. The goal is to schedule maximum number of jobs, while all the intervals on which the jobs are scheduled must be non-overlapping.

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. For the other blocks, we generate an LP relaxation to the scheduling problem by enumerating over all feasible schedules in each block. We then randomly round the optimal LP-solution, to obtain the improved approximation guarantee.

Let $k = \lceil \frac{6}{\epsilon} \rceil$. We denote the input set of jobs by S, and the maximum finish time of a job in S (the time horizon) by T. The algorithm works in two phases. In the first phase, we divide the time line $[0, T]$ into blocks and partitions the blocks into two subsets B^I and B^{II}. We also schedule a subset $S^I \subseteq S$ of jobs in blocks B^I. In the second phase, we schedule at most $4k^{k \ln k + 3}$ jobs in each block of B^{II}. 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 use OPT to denote both some fixed optimal schedule and the set of jobs scheduled in this schedule. 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.

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 empty 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}.

The first iteration: In the first iteration we run algorithm GREEDY, which is defined as follows. Start at time 0. Whenever the machine becomes idle, schedule a job that finishes first among all the available jobs.

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. For any feasible schedule SCHED, let |SCHED| denote the number of jobs scheduled in SCHED.

LEMMA 2.1 $|\text{OPT}_{B_1}| \geq (1 - 1/k^3)|\text{OPT}|$.

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, $|\text{OPT}| \geq |S_1|$. □

LEMMA 2.2 *In each block computed by the first 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 jobs in OPT to scheduled jobs in S_1. Each unscheduled job in OPT is mapped to the unique overlapping job in S_1 that prevented it from being scheduled, because it had an earlier finish time. □

The partition after the first iteration does not harm the optimal solution too much, as Lemma 2.1 states. However, by Lemma 2.2, 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 later iterations. In later iterations we only refine the existing partition into blocks. Thus, Lemma 2.2 holds for the block partition throughout the first phase.

The *i*th 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 to the next block. 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 2.3 $|\text{OPT}_{B_i}| \geq (1 - i/k^3)|\text{OPT}|$.

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 $\sum_{j=1}^{i} \frac{|S_j|}{k^3} \leq \frac{i}{k^3}|\text{OPT}|$. Thus, there is a feasible solution of jobs that do not cross the boundaries of the blocks from B_i, containing at least $\left(1 - \frac{i}{k^3}\right)|\text{OPT}|$ jobs. □

LEMMA 2.4 *In each block computed by the ith 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 block b, 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: For each integer $i : 1 \leq i \leq k \ln k$, let A_i be the following event: i is the first iteration in which $|S_i| \geq \left(1 - \frac{1}{k}\right)|S_{i-1}|$ holds. Let E be the event that A_i does not happen for any $i : 1 \leq i \leq k \ln k$.

If event A_i happens for some $i : 1 \leq i \leq k \ln k$, then the first phase is terminated after the completion of iteration i. The output of the first phase in this case is determined as follows. We discard the block refinement of the last iteration, i.e., the final block partition is 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 $B^{II} = B_{i-1} \setminus B^I$.

If event E happens, we terminate the algorithm after the completion of phase $k \ln k + 1$. We set $S^I = \emptyset$, $B^I = \emptyset$, $S_{pass} = S \setminus S_{(k \ln k+1)}$, and $B^{II} = B_{(k \ln k+1)}$. We denote by r the number of iterations in the first phase. We sometimes refer to blocks in B^{II} as *empty* blocks.

LEMMA 2.5 $|\text{OPT}_{B_r}| \geq \left(1 - \frac{1}{k}\right)|\text{OPT}|$.

PROOF. Since $r \leq k \ln k + 1$, by Lemma 2.3, $|\text{OPT}_{B_r}| \geq \left(1 - \frac{k \ln k+1}{k^3}\right)|\text{OPT}| \geq \left(1 - \frac{1}{k}\right)|\text{OPT}|$. □

DEFINITION 2.1 *Let* $B^I, B^{II}, S^I, S_{pass}$ *be the output of the first phase of the algorithm, and let* B *be the final partition of the time line into blocks. Given any schedule* SCHED*, we say that it is a* restricted feasible schedule*, if the following three conditions hold: (1) no job crosses a block boundary, (2) the set of jobs scheduled in blocks* B^I *is exactly* S^I*, and (3) all the jobs scheduled in blocks of* B^{II} *belong to the subset* S_{pass} *of jobs.*

Let OPT′ be the optimal restricted feasible schedule. The heart of the algorithm analysis is proving that the number of jobs scheduled in OPT′ is close to |OPT|.

LEMMA 2.6 $|\text{OPT}'| \geq (1 - \epsilon)|\text{OPT}|$.

PROOF. Consider two cases.

Case 1: Event E happens. Recall that in this case, for each $i : 1 < i \leq k \ln k$, $|S_i| \leq \left(1 - \frac{1}{k}\right)|S_{i-1}|$, and thus $|S_{(k \ln k+1)}| \leq \left(1 - \frac{1}{k}\right)^{k \ln k}|S_1| \leq \frac{|S_1|}{k} \leq \frac{|\text{OPT}|}{k}$ holds.

We now show how to convert OPT into a restricted feasible schedule in two steps, without losing too many jobs. Recall that $S_{pass} = S \setminus S_{(k \ln k+1)}$, $B^{II} = B_{(k \ln k+1)}$, $B^I = \emptyset$, $S^I = \emptyset$. The first step is removing from OPT all the jobs that cross block boundaries. From Lemma 2.5, at least $|\text{OPT}_{B_{(k \ln k+1)}}| \geq \left(1 - \frac{1}{k}\right)|\text{OPT}|$ jobs remain after this step. The second step is removing all the jobs in $S \setminus S_{pass}$ from the schedule. Since $S \setminus S_{pass} = S_{(k \ln k+1)}$ the number of jobs we remove from OPT in this step is

$|S_{(k\ln k+1)}| \leq \frac{|\text{OPT}|}{k}$. It is easy to see that the resulting schedule is restricted feasible, and that it contains at least $\left(1-\frac{2}{k}\right)|\text{OPT}| \geq (1-\epsilon)|\text{OPT}|$ jobs.

Case 2: Assume now event A_i happens for some $i: 1 \leq i \leq k\ln k$. Recall that in this case $S^I = S_i$, B^I is the subset of blocks where jobs in S^I are scheduled in iteration i, B^{II} contains the remaining blocks, and $S_{pass} = S \setminus S_{i-1}$. We start with the optimal solution OPT, and convert it into a restricted feasible solution in three steps, while bounding the number of jobs we remove from OPT in each step. In the first step, we remove all the jobs that cross block boundaries. Again, following Lemma 2.5, the number of jobs removed at this step is at most $\frac{|\text{OPT}|}{k}$. The second step is removing all the jobs in $S_{i-1} \setminus S_i$ from the schedule. Since $|S_i| \geq \left(1-\frac{1}{k}\right)|S_{i-1}|$ and $S_i \subseteq S_{i-1}$, we have that $|S_{i-1} \setminus S_i| = |S_{i-1}| - |S_i| \leq \frac{|S_i|}{k} \leq \frac{|\text{OPT}|}{k}$. Thus, we lose at most $\frac{|\text{OPT}|}{k}$ jobs in this step. In our final third step, we discard all the jobs that are currently scheduled in blocks B^I but do not belong to S^I, and schedule the job set S^I in blocks B^I similarly to the schedule produced by the first phase of the algorithm. Observe that all the jobs discarded in this step belong to $S_{pass} = S \setminus S_{i-1} = R_1 \cup R_2 \cup \ldots \cup R_{i-1}$. Let b be some block in B^I. By Lemma 2.4 and the fact that B_{i-1} is a refinement of the block partitions of the previous iterations, for all $1 \leq j \leq i-1$, at most $2k^{j+2}$ jobs from set R_j can be scheduled in block b. Thus, at most $\Sigma_{j=1}^{i-1} 2k^{j+2} \leq 4k^{i+1}$ jobs from S_{pass} can be scheduled in b. On the other hand, we know that at least k^{i+2} jobs from S^I are scheduled in b in iteration i. Thus, the number of jobs removed from OPT on this step is bounded by $\frac{4}{k}|S^I| \leq \frac{4}{k}|\text{OPT}|$.

The schedule obtained after performing the above steps is clearly restricted feasible, and the number of jobs it contains is at least:

$$|\text{OPT}| - \frac{|\text{OPT}|}{k} - \frac{|\text{OPT}|}{k} - \frac{4|\text{OPT}|}{k} = \left(1-\frac{6}{k}\right)|\text{OPT}| = (1-\epsilon)|\text{OPT}|$$

□

LEMMA 2.7 *In every empty block* OPT′ *schedules less than* $4k^{k\ln k+3}$ *jobs from* S_{pass}.

PROOF. The lemma follows from Lemmas 2.2 and 2.4. 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 $2\sum_{i=1}^{j} k^{i+2} < 4k^{j+2} \leq 4k^{k\ln k+3}$. □

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'_{pass} 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 $|S'_{pass}|$. 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 2.7, 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 $\sum_{s=0}^{4k^{k\ln k+3}} s!\binom{n}{s} = n^{2^{O(k\ln^2 k)}}$. Let $\mathcal{M}(b)$ 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 $M \in \mathcal{M}(b)$, a variable $y_M^b \in \{0,1\}$. Setting $y_M^b = 1$ means that the schedule M is chosen for the block b. The integer program that computes an upper bound on $|S'_{pass}|$ is the following:

$$\text{maximize} \quad \sum_{b\in B}\sum_{M\in\mathcal{M}(b)}\sum_{j\in M} y_M^b \quad \text{subject to}$$

$$\begin{aligned} \textstyle\sum_{b\in B}\sum_{M\in\mathcal{M}(b)|j\in M} y_M^b \le 1 \quad & \forall j \in S_{pass} \\ \textstyle\sum_{M\in\mathcal{M}(b)} y_M^b = 1 \quad & \forall b \in B \\ y_M^b \in \{0,1\} \quad & \forall b \in B,\ \forall M \in \mathcal{M}(b). \end{aligned}$$

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-step algorithm:

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

(ii) 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 $x_j^b = \sum_{M\in\mathcal{M}(b)|j\in M} y_M^b$, and put $x_j = \sum_{b\in B} x_j^b$. 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 2.8 *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 every block no schedule containing j was chosen, which is $\prod_b \left(1-x_j^b\right)$. 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, $x_j^b = x_j/t$. Thus, $p_j = 1-\prod_b \left(1-x_j^b\right) \ge 1-\left(1-x_j/t\right)^t$. Therefore, $p_j/x_j \ge \left(1-\left(1-x_j/t\right)^t\right)/x_j$. The right-hand side is monotonically decreasing in x_j, and thus the minimum is achieved at $x_j = 1$. We conclude that $p_j/x_j \ge 1-\left(1-1/t\right)^t \ge 1-\frac{1}{e}$. This completes the proof of the lemma. □

COROLLARY 2.1 $E[z_{int}] \ge \left(1-\frac{1}{e}\right) z$.

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

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

THEOREM 2.1 *For every $\epsilon > 0$, the two-phase algorithm runs in polynomial time and guarantees, in expectation, an $e/(e-1)+\epsilon$ approximation to JISP.*

PROOF. Let $S^{I\!I}$ 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 $E[|S^I|+|S^{I\!I}|] = |S^I|+E[|S^{I\!I}|]$. By Corollary 2.1, $E[|S^{I\!I}|] \ge \left(1-\frac{1}{e}\right) z^* \ge \left(1-\frac{1}{e}\right)|S'_{pass}|$.

As $|S^I| + |S'_{pass}| = |\text{OPT}'| \ge (1-\epsilon)|\text{OPT}|$ (by Lemma 2.6), the expected value of the solution is $|S^I|+E[|S^{I\!I}|] \ge |S^I|+\left(1-\frac{1}{e}\right)|S'_{pass}| \ge \left(1-\frac{1}{e}\right)\left(|S^I|+|S'_{pass}|\right) \ge \left(1-\frac{1}{e}\right)(1-\epsilon)|\text{OPT}| \ge \left(1-\frac{1}{e}-\epsilon\right)|\text{OPT}|$. □

Recall that JISPk is a special case of JISP, where each job has at most k possible intervals. It is easy to see, from the proof of Lemma 2.8 and from Corollary 2.1 and Theorem 2.1, that the approximation factor our algorithm achieves for JISPk is $\frac{k^k}{k^k-(k-1)^k}+\epsilon$. In particular, for $k=2$, the approximation factor is $4/3+\epsilon$.

Resource Allocation The resource allocation problem is defined similarly to throughput maximization on one machine ($1|r_j|\sum \overline{U}_j$), except that now each job j has a height h_j. Several jobs can be executed simultaneously, as long as the total sum of heights of jobs executed at the same time never exceeds 1.

Calinescu et al [8] consider a special case where each job has exactly one interval in which it can be executed, i.e., $d_j - r_j = p_j$ for all j. They show a $(2+\epsilon)$-approximation algorithm as follows. The jobs

are divided into a set of large jobs (with heights $\geq \delta$ for some constant δ) and a set of small jobs (all the other jobs). For the set of large jobs, the problem can be solved optimally by dynamic programming. For the set of small jobs, they show an LP-based $(1+\epsilon)$-approximation algorithm, where ϵ is a function of δ. Since either more than half of jobs in the optimal solution are large, or more than half of jobs in the optimal solution are small, this gives a $(2+\epsilon)$-approximation.

We use these ideas combined with the ideas we used in the approximation algorithm for JISP to improve the approximation factor of resource allocation problem (in the unweighted case).

We also start by fixing some small constant δ, and dividing the jobs into the set of large and small jobs, as described above. It turns out that the algorithm of [8] for small jobs easily extends to the general resource allocation problem. The following lemma easily follows from [8].

LEMMA 2.9 *There is an approximation algorithm for resource allocation problem, where the heights of jobs are at most δ, that achieves a $(1+\epsilon(\delta))$-approximation. For $0 < \delta \leq 0.01$, $0 < \epsilon(\delta) < 1$ and is monotonically increasing in δ.*

Below we show that our approximation algorithm for JISP can be extended to handle resource allocation for large jobs, for any constant δ. The algorithm gives $(\frac{e}{e-1}+\epsilon)$-approximation for large jobs.

Finally, we combine both results as follows. If the optimal solution contains more than a fraction $\frac{e}{2e-1}$ of large jobs, our algorithm for large jobs will schedule at least $\frac{e-1}{2e-1}(1-\epsilon')|\text{OPT}|$ jobs. Otherwise, the optimal solution contains at least a fraction $\frac{e-1}{2e-1}$ of small jobs. Then the algorithm for small jobs will schedule at least $\frac{e-1}{2e-1}(1-\epsilon'')|\text{OPT}|$ jobs. Thus, the approximation factor of our algorithm is $\frac{2e-1}{e-1}+\epsilon$, where ϵ is an arbitrarily small constant. This improves the best previously known approximation factor of 5, due to [4].

It now remains to show how to extend our approximation algorithm for JISP to resource allocation with large jobs. Let z be the maximum number of large jobs that can be scheduled simultaneously, i.e., $z = \lfloor \frac{1}{\delta} \rfloor$ where δ is the minimum height of a large job. The first phase of the algorithm is performed almost similarly to the original algorithm, with the following changes.

First Iteration: We run GREEDY again, in the same manner as before (i.e., all the jobs are scheduled on non-overlapping intervals; we do not attempt to schedule several jobs to be executed simultaneously). The only difference from the first iteration in the original algorithm is that now the jobs are divided into blocks containing $k^3 z$ jobs each. It is easy to see that Lemma 2.1 still holds: indeed, for each block boundary there are at most z jobs in the optimal schedule crossing it, while the block contains at least $k^3 z$ jobs scheduled in the current iteration. Also, similarly to Lemma 2.2, OPT schedules at most $k^3 z^2$ jobs from R_1 in each block. The reasoning is the same as in the proof of Lemma 2.2, except that now, for each job j scheduled in some block b by the algorithm, there can be up to z jobs in the optimal solution, which are prevented from being scheduled by j.

Iteration *i*: Iteration i is performed similarly as in the original algorithm (again, we only schedule jobs on non-overlapping intervals). The only difference is that now a block is emptied if it contains less than $k^{i+2}z^i$ jobs, and it is subdivided into smaller blocks containing $k^{i+2}z^i$ jobs otherwise. Since each new block boundary is crossed by at most z jobs in the optimal solution, Lemma 2.3 still holds. The statement of Lemma 2.4 changes as follows: in each block computed by the ith iteration, OPT schedules at most $2k^{i+2}z^{i+1}$ jobs from R_i. Again, the proof is almost unchanged: we only need to notice that each job scheduled in some block in iteration i prevents at most z jobs from OPT from being scheduled in the current iteration.

The stopping condition: remains exactly the same as in the original algorithm. Since Lemmas 2.1 and 2.3 hold for the new algorithm, Lemma 2.5 is also true. It now only remains to analyze Lemma 2.6. The analysis of the first case remains unchanged: the only jobs discarded in this case are the jobs that cross block boundaries or the jobs from $S_{(k \ln k+1)}$. Consider now the second case, when event A_i happens. The jobs discarded in the first step are the jobs that cross block boundaries, and their number is bounded by $\frac{|\text{OPT}|}{k}$ as before. The jobs discarded at the second step are the jobs in $S_{i-1} \setminus S_i$, whose number is also bounded by $\frac{|\text{OPT}|}{k}$ as before. For the third step, we remove the jobs in $S_{pass} = R_1 \cup R_2 \cup \cdots \cup R_{i-1}$, which

are scheduled in blocks B^I. Consider one such block b. Then for each $j : 1 \leq j \leq i-1$, at most $2k^{j+2}z^{j+1}$ jobs from R_j are scheduled in b. Summing up for all $j : 1 \leq j \leq i-1$, we get that at most $4k^{i+1}z^i$ jobs from S_{pass} are scheduled in b. Since there are $k^{i+2}z^i$ jobs from S^I scheduled in b in iteration i, we get that the number of jobs removed from the optimal schedule in the third step is at most $\frac{4}{k}|S^I| \leq \frac{4}{k}|\text{OPT}|$. The rest of the proof remains unchanged.

Finally, the second phase of the algorithm remains almost unchanged: the only difference is that the number of jobs that can be scheduled in each empty block now grows by a factor $z^{O(k \ln k)}$, and the possible schedules considered by the algorithm allow jobs to be executed simultaneously, as long as the sum of their heights does not exceed 1. Since both z and k are constant, the running time of the algorithm remains polynomial.

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 $\text{Release}_j(s, e) = \{i \mid i \leq j,\ r_i \in [s, e)\}$. 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 3.1 *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 3.2 *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.

LEMMA 3.3 *The schedule $D(s, x, e, j, \text{IN}, \text{OUT})$ computed by the algorithm is a feasible schedule of jobs from $Release_j(s, e) \bigcup \text{IN} \setminus \text{OUT}$ 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 $D(t, t + p_j, e, j - 1, G, H)$. □

LEMMA 3.4 *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 job j, and let $A_1 = \text{IN} \setminus (A_2 \bigcup \{j\})$. Let B be the subset of jobs from $\text{Release}_{j-1}(s, t) \setminus B_1$ 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 $\text{OPT}(s, x, e, j, \text{IN}, \text{OUT})$. □

We conclude

THEOREM 3.1 *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 3.3 and 3.4. 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 $O\left(T^4 n^2 \log n \binom{n}{k^2}^3 2^{k^2}\right) \leq O\left(T^4 2^{k^2} n^{3k^2+2} \log n\right)$. □

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. *Journal of the ACM (JACM)* volume 48, issue 5, pages 1069 - 1090, 2001.

[5] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber. Approximating the throughput of multiple machines in real-time scheduling. *SIAM Journal on Computing*, volume 31, number 2, pp. 331-352, 2001.

[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.

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

[8] G. Calinescu, A. Chakrabarti, H.J. Karloff, and Y. Rabani. Improved Approximation Algorithms for Resource Allocation. IPCO 2002, the 9th Conference on Integer Programming and Combinatorial Optimization, Lecture Notes in Computer Science 2337, Springer-Verlag, 2002, pp. 401-414

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

[10] 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.

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

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

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

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

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

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

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

[23] 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.

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

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

[26] D.K.Y. Yau and S.S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, Vol. 5, No. 4, pp. 475–488, August 1997.