Noname manuscript No.
(will be inserted by the editor)

Scheduling with Gaps: New Models and Algorithms

Marek Chrobak · Mordecai Golin · Tak-Wah Lam · Dorian Nogneng

Abstract We consider scheduling problems for unit jobs with release times, where the number or size of the gaps in the schedule is taken into consideration, either in the objective function or as a constraint. Except for several papers on minimum-energy scheduling, there is no work in the scheduling literature that uses performance metrics depending on the gap structure of a schedule. One of our objectives is to initiate the study of such scheduling problems.

We focus on the model with unit-length jobs. First we examine scheduling problems with deadlines, where we consider two variants of minimum-gap scheduling: maximizing throughput with a budget for the number of gaps and minimizing the number of gaps with a throughput requirement. We then turn to other objective functions. For example, in some scenarios gaps in a schedule may be actually desirable, leading to the problem of maximizing the number of gaps. Other versions we study include minimizing maximum gap or maximizing minimum gap. The second part of the paper examines the model without deadlines, where we focus on the tradeoff between the number of gaps and the total or maximum flow time.

For all these problems we provide polynomial time algorithms, with running times ranging from $O(n \log n)$ for some problems to $O(n^7)$ for other. The solutions involve a spectrum of algorithmic techniques, including different dynamic programming formulations, speed-up techniques based on searching Monge arrays, searching $X+Y$ matrices, or implicit binary search.

Throughout the paper, we also draw a connection between gap scheduling problems and their continuous analogues, namely hitting set problems for intervals of real numbers. As it turns out, for some problems, the continuous variants provide insights leading to more efficient algorithms for the corresponding discrete versions, while for other problems completely new techniques are needed to solve the discrete version.

1 Introduction

We consider scheduling of unit-length jobs with release times, where the number or size of the gaps in the schedule is taken into consideration, either in the objective function or as a constraint.

This research was inspired by the work on scheduling problems where the objective is to minimize the number of gaps in a schedule. Such problems arise in minimum-energy scheduling in the power-down model, where a schedule specifies not only execution times of jobs but also at what times the processor can be turned off. The processor uses energy at rate L per time unit when the power is on, and it does not consume any energy when it is off. If the energy required to power-up the system is less than L then energy minimization is equivalent to minimizing the number of gaps in the schedule. The problem was introduced

M.Chrobak was supported by NSF grants CCF-0729071, CCF-1217314 and CCF-1536026. M. Golin was supported by grant FSGRF14EG28.

M. Chrobak
Department of Computer Science, University of California at Riverside, USA.

M. Golin
Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong.

T-W. Lam
Department of Computer Science, University of Hong Kong, Hong Kong.

D. Nogneng
LIX, École Polytechnique, France.

in 2005 by Irani and Pruhs [24], and its complexity remained open for a few years. The first progress was achieved by Baptiste [6], who gave a polynomial time algorithm for unit jobs that achieves running time $O(n^7)$. This time complexity was subsequently reduced to $O(n^4)$ in [7,8]. (In that paper a generalization to arbitrary processing times with job preemption is also considered.) A greedy algorithm was analyzed in [15,14] and shown to have approximation ratio 2. Other variants of this problem have been studied, for example the multiprocessor case [18] or the case when jobs have agreeable deadlines [2,3]. (See the survey in [5] for more information.)

To our knowledge, the above gap-minimization model is the only scheduling model in the literature that considers gaps in the schedule as a performance measure. As we show, however, one can formulate a number of other natural, but not yet studied variants of gap scheduling problems. Some of these problems can be solved using dynamic-programming techniques resembling those used for minimizing the number of gaps. Other require new approaches, giving rise to new and interesting algorithmic problems.

Throughout the paper, we focus exclusively on the model with unit-length jobs. The first type of scheduling problems we study involve jobs with release times and deadlines. In this category, we address the following problems:

– In Section 3, we study maximizing throughput (the number or total weight of scheduled jobs) with a budget γ for the number of gaps. We give an $O(\gamma n^6)$-time algorithm for this problem.
– In Section 4 we study the variant where we need to minimize the number of gaps under a throughput requirement, namely where either the number of jobs or their total weight must meet a specified threshold. We show that this problem can be solved in time $O(n^7)$.
– In the two problems above, the underlying assumption was that it is desirable to have as few gaps as possible. However, in certain applications gaps in a schedule may be actually desirable. This motivates the gap scheduling model where we wish to *maximize* the number of gaps while scheduling all jobs (providing that the instance is feasible). We study this problem in Section 5, and we provide an algorithm that computes an optimal schedule in time $O(n^5)$.
– Instead of the total number of gaps, the *size* of gaps may be a useful attribute of a schedule. In Section 6 we study the problem where, assuming that the given instance is feasible, we want to compute a schedule for which the maximum gap size is minimized. We give an $O(n^2 \log n)$-time algorithm for this problem.

We also consider scheduling problems where jobs have no deadlines. Now all jobs need to be scheduled. In this model we can of course schedule all jobs in one block, without gaps, but then some jobs may need to wait a long time for execution. To avoid this, we will also take into account the flow time measure, where the flow of a job is the time elapsed between its release and completion times, and we will attempt to minimize either the maximum flow or the total flow of jobs. We address three problems in this category:

– Minimizing total flow time with a budget γ for the number of gaps (Section 7). As we show, this problem can be solved in time $O(n \log n + \gamma n)$, by exploiting the Monge property of the dynamic programming arrays. The running time is in fact $O(\gamma n)$ if the jobs are given in sorted order of release times.
– Minimizing the number of gaps with a budget for total flow (Section 8). The algorithm from Section 7 can be adapted to solve this problem in time $O(n \log n + g^* n)$, where g^* is the optimum value. If the jobs are given in sorted order of release times, the running time is $O(g^* n)$.
– Minimizing the number of gaps with a bound on the maximum flow time (Section 9). We show that this problem can be solved in time $O(n \log n)$, or even $O(n)$ if the jobs are already sorted in order of increasing release times.
– Minimizing maximum flow time with a budget γ for the number of gaps (Section 10). For this problem we give an algorithm with running time $O(n \log n)$.

Summarizing, for all these problems we provide polynomial-time algorithms, with running times ranging from $O(n \log n)$ for some problems, to $O(n^7)$ for other. Interestingly, the solutions involve a wide spectrum of algorithmic techniques, including different dynamic programming formulations and speed-up techniques based on searching Monge arrays, searching $X + Y$ matrices, and implicit binary search.

As another theme throughout the paper, we draw a connection between gap scheduling problems that we study and their continuous analogues, which are variants of hitting set problems for intervals of real numbers. In this continuous model, each job is represented by an interval between its release time and deadline, and a "schedule" assigns it to a point in this interval. For example, the continuous version of the minimum-gap scheduling problem is equivalent to computing a hitting set of minimum cardinality. As it turns out, for some problems, the continuous variants provide insights leading to more efficient algorithms for the corresponding discrete versions, while in other problems completely new techniques are needed to solve the discrete version.

2 Preliminaries

The time is assumed to be discrete, divided into unit time intervals $[t, t+1)$, for $t = 1, 2, ...$, that we call *slots*. We will number these consecutive slots $0, 1, ...$, and we will refer to $[t, t+1)$ simply as *time slot* t, or occasionally even as *time* t. By $\mathcal{J}$ we will denote the instance, consisting of a set of unit-length jobs numbered $1, 2, ..., n$, each job j with a given integer release time r_j. This r_j denotes the first slot where j can be executed.

A *schedule* S of $\mathcal{J}$ is defined by an assignment of jobs to time slots such that (i) if a job j is assigned to a slot t then $t \geq r_j$, and (ii) no two jobs are assigned to the same slot. If j is assigned to slot t in a schedule S then we say that it is *scheduled* or *executed* at t. In most scheduling problems we assume that all jobs can be scheduled. In problems that involve throughput we will also consider partial schedules, where only a subset of the jobs is scheduled (for jobs outside this subset the schedule is undefined).

For a given schedule S, time slots where jobs are scheduled are called *busy*, while all other slots are called *idle*. An inclusion-wise maximal time interval of busy slots is called a *block* of S. An interval between two consecutive blocks in S is called a *gap* of S. Of course, the number of blocks in S is always one more than the number of gaps.

Instances with deadlines. In some of the scheduling problems we consider the jobs in $\mathcal{J}$ will also have specified deadlines. The *deadline* of job j is denoted d_j, is assumed to be integer, and it is the last slot where j can be scheduled. (Thus it may happen that $d_j = r_j$, in which case j can only be executed in one slot.)

For instances with deadlines, we can restrict our attention to schedules S that satisfy the *earliest-deadline-first property (EDF)*: at any time t, either S is idle at t or it schedules a pending job with the earliest deadline. Using the standard exchange argument, any schedule can be converted into one that satisfies the EDF property and has the same set of busy slots.

Without loss of generality, we can make the following assumptions about $\mathcal{J}$:

(i) $r_j \leq d_j$ for each j,
(ii) all jobs are ordered according to deadlines, that is $d_1 \leq \ldots \leq d_n$,
(iii) all release times are distinct and all deadlines are distinct, and
(iv) $\mathcal{J}$ is feasible (that is, all jobs can be scheduled).

The validity of assumptions (i) and (ii) is trivial. Assumption (iv) follows immediately from (iii), as we can simply schedule each job at its release time. Therefore we only need to justify (iii).

To show that assumption (iii) is valid, we modify the original instance as follows: If two release times are equal, say when $r_i = r_j$ and $d_j \leq d_i$ for $i \neq j$, then we let $r_i = r_i + 1$. Symmetrically, if $d_i = d_j$ and $r_i \leq r_j$ then we let $d_i = d_i - 1$. If this change produces a job i with $d_i < r_i$, then job i cannot of course be scheduled. For problems where the feasibility is a requirement, we can then report that the instance is not feasible. For other problems, we can remove this job i from the instance altogether. The correctness of assumption (iii) can be justified using a standard exchange argument that we formalize by proving the lemma below. (Schedules considered in this lemma are allowed to be partial.)

Lemma 1 *Let $\mathcal{J}'$ be the instance obtained by modifying a given instance $\mathcal{J}$ as explained above, and let X be some set of time slots. Then $\mathcal{J}$ has a schedule S whose set of busy slots is X if and only if $\mathcal{J}'$ has a schedule S' whose set of busy slots is X.*

Proof We now justify Lemma 1. It is sufficient to consider only the case when $\mathcal{J}'$ is obtained from $\mathcal{J}$ by modifying just one job, as then we can apply the lemma repeatedly. So suppose that we have two different jobs i, j in $\mathcal{J}$ with $r_i = r_j$ and $d_j \leq d_i$, and that $\mathcal{J}'$ is obtained from $\mathcal{J}$ by replacing i by i' such that $r_{i'} = r_i + 1$ and $d_{i'} = d_i$.

($\Leftarrow$) This implication is trivial, because any schedule S' of $\mathcal{J}'$ gives us a schedule S of $\mathcal{J}$ with the same set of busy slots by simply replacing i' by i (if i' is used at all).

($\Rightarrow$) Consider a schedule S of some subset of $\mathcal{J}$ in which X is the set of busy slots. If i is not scheduled in S then we can simply use $S' = S$. If i is scheduled in S at slot other than r_i, then we can obtain S' by replacing i by i'. The last case is when i is scheduled at a slot r_i in S. If j is scheduled in S as well then we obtain S' by swapping i and j in S and then replacing i by i', with i' scheduled where j was scheduled in S. On the other hand, if j is not scheduled in S, then we obtain S' by replacing i by j which is scheduled at $r_j = r_i$. □

To implement the modification of the instance outlined before Lemma 1, when we adjust the release times we can process them in increasing order to facilitate finding equal release times. Each job's release time can be incremented at most n times, and maintaining the ordering will introduce a logarithmic overhead. Deadlines can be processed in the symmetric way. Then the overall running time to modify the instance will be $O(n^2 \log n)$. Thus this preprocessing does not affect the overall running time of our algorithms for instances with deadlines (that all have running time at least this large).

Instances without deadlines. For schedules involving objective functions other than throughput, we can assume that the jobs are ordered according to non-decreasing release times. For the total-flow objective function we can also assume that all release times are different. The reason is that, although modifying the release times may change the total flow value (see the definition of the flow time in Section 7, paragraph 1), this change will be uniform for all schedules, so the schedule's optimality will not be affected. The appropriate modification of release times can be achieved in time $O(n \log n)$ as follows: First, sort all jobs in order of release times, so that $r_1 \leq r_2 \leq ... \leq r_n$. Process them in this order. Providing that the new release times $r'_1 < r'_2 < ... < r'_{j-1}$ of jobs $1, 2, ..., j-1$ are already computed, let the new release time of job j be $r'_j = \max(r'_{j-1} + 1, r_j)$. If the jobs are already given in the sorted order, this process will in fact take time $O(n)$. Thus the running times of our algorithms are not affected by this preprocessing.

We remark that modifying release times may affect the maximum flow values non-uniformly (that is, differently for different schedules), so we will not be using the assumption about different release times in Sections 9 and 10, where maximum flow of jobs is considered.

Shifting blocks. To improve the running time, some of our algorithms use assumptions about possible locations of the blocks in an optimal schedule. The general idea is that each block can be shifted, without affecting the objective function, to a location where it will contain either a deadline or a release time. The following lemma (that is implicit in [7]) is useful for this purpose. We formulate the lemma for leftward shifts; an analogous lemma can be formulated for rightward shifts and for deadlines instead of release times.

Lemma 2 *Assume that all jobs in the instance have different release times. Let $B = [u, v]$ be a block in a schedule such that the job scheduled at v has release time strictly before v. Then B can be shifted leftward by one slot, in the sense that the jobs in B can be scheduled in the interval $[u-1, v-1]$.*

Proof We construct a sequence of job indices $i_1, i_2, ..., i_q$ such that i_1 is the job scheduled at v, each job i_b, for $b = 2, 3, ..., q$, is scheduled in B at the release time $r_{i_{b-1}}$ of the previous job in the sequence, and $r_{i_q} < u$. This is quite simple: As mentioned earlier, we start by letting i_1 be the job scheduled at v. Suppose that for some $c \geq 1$ we have already chosen jobs $i_1, i_2, ..., i_c$ such that i_c is scheduled in B and each i_b, for $b = 2, 3, ..., c$, is scheduled at $r_{i_{b-1}}$. The choice of this sequence implies that $r_{i_c} < r_{i_{c-1}} < ... < r_1 = v$. If $r_{i_c} < u$, we let $q = c$ and we are done. So suppose that $r_{i_c} \geq u$. Since all release times are different, we have $r_{i_c} < r_{i_{c-1}}$. We then take i_{c+1} to be the job scheduled at r_{i_c}. By repeating this process, we obtain the desired sequence.

Given the jobs $i_1, i_2, ..., i_q$ from the previous paragraph, we can modify the schedule by scheduling i_q at time $u-1$, and scheduling each i_b, $b = 1, 2, ..., q-1$ at r_{i_b}. This will result in shifting B to the left by one slot, proving the lemma. $\square$

Interval hitting. For some of our scheduling problems it is useful to consider their "continuous" analogues obtained by assuming that all release times and deadlines are spread very far apart; thus in the limit we can think of jobs as having length 0. Each r_j and d_j (if deadlines are in the instance) is a point in time, and to "schedule" j we assign it to a point in the interval $[r_j, d_j]$. Two jobs that would be assigned to consecutive slots in a discrete schedule will then end up being on the same point. This continuous problem is then equivalent to computing a hitting set for a given collection of intervals on the real line, with some conditions involving gaps in-between its consecutive points.

More formally, in the hitting-set problem we are given a collection of intervals $I_j = [r_j, d_j]$, where r_j, d_j are real numbers. Our objective is to compute a set H of points such that $H \cap I_j \neq \emptyset$ for all j. This set H is called a *hitting set* of the intervals $I_1, I_2, ..., I_n$. (This formalism corresponds to scheduling problems with deadlines and where all jobs need to be scheduled; it can be easily adapted in a natural way to other variants that we study, when jobs may not have deadlines, or when some jobs do not need to be scheduled.)

If H is a hitting set of intervals $I_1, I_2, ..., I_n$, then for each j we can pick a *representative* $h_j \in H \cap I_j$. Sorting these representatives from left to right, $h_{i_1} \leq h_{i_2} \leq ... \leq h_{i_n}$, the intervals between consecutive representatives are called *gaps* of H. The length of the gap between h_{i_b} and $h_{i_{b+1}}$ is $h_{i_{b+1}} - h_{i_b}$.

For each gap scheduling problem we can then consider the corresponding hitting-set problem. For example, minimizing the number of gaps in a schedule translates into a minimum-cardinality hitting set for a collection of intervals. It is well known (folklore) that this problem can be solved with a greedy algorithm in time $O(n \log n)$: Initialize $H = \emptyset$. Then, going from left to right, at each step locate the earliest-ending interval I_j not yet hit by the points in H and add d_j to H.

These interval-hitting problems are conceptually easier to deal with than their discrete counterparts. As we show, some algorithms for interval-hitting problems extend to their corresponding gap scheduling problems, while for other these discrete variants require different techniques.

3 Maximizing Throughput with Budget for Gaps

In this section we consider a variant of gap scheduling where we want to maximize throughput (that is, the number of scheduled jobs), given a budget γ for the number of gaps. We first show that the continuous version of this problem can be solved in time $O(\gamma n^2)$. For the discrete case we give an algorithm with running time $O(\gamma n^6)$.

Continuous case. Formally, the continuous variant of the problem is defined as follows. We are given a collection of intervals $I_j = [r_j, d_j]$, $j = 1, 2, ..., n$ and a positive integer $\xi \leq n$. The objective is to compute a set H of at most ξ points that hits the maximum number of intervals, where a point is said to *hit* a set if it belongs to this set. (Here, ξ corresponds to the number of blocks, so its value is one more than the number of gaps.) Without loss of generality we only need to consider sets $H \subseteq \{d_1, d_2, ..., d_n\}$ and we can assume that all release times and deadlines are different.

There is a simple dynamic-programming algorithm for this problem that works as follows. Order the intervals according to deadlines, that is $d_1 < d_2 < ... < d_n$. For $h = 1, 2, ..., \xi$ and $b = 1, 2, ..., n$, let $\mathsf{T}_{b,h}$ be the maximum number of input intervals that can be hit by a subset $H \subseteq \{d_1, d_2, ..., d_b\}$ such that $|H| \leq h$ and $d_b \in H$. For all b, we first initialize $\mathsf{T}_{b,1}$ to be the number of intervals that contain d_b. Similarly, for all h, we let $\mathsf{T}_{1,h}$ to be the number of intervals that contain d_1. Then, for all $h = 2, 3, ..., \xi$ and $b = 2, 3, ..., n$, we can compute $T_{b,h}$ using the recurrence:

$$\mathsf{T}_{b,h} = \max_{a<b} \{\mathsf{T}_{a,h-1} + \Delta_{a,b}\},$$

where $\Delta_{a,b}$ is the number of intervals I_i such that $d_a < r_i \leq d_b \leq d_i$, namely the intervals that are hit by d_b but not by d_a. The output value is $\max_b \mathsf{T}_{b,\xi}$.

With a bit of care, all values $\Delta_{a,b}$ can be pre-computed in time $O(n^2)$: First sort all release times and deadlines. For each a, consider only intervals I_i to the right of d_a, namely those with $r_i > d_a$. We will make a sweep through release times and deadlines, starting at d_a, and for each visited point counting the number of intervals hit by this point. We start with $x = d_a$ and with a counter q initialized to 0. Then iteratively increment x to the next release time or deadline, whichever is earliest. At each step update q, by increasing it if the new point is a release time and decreasing it if the current point is a deadline. If the new point is $x = d_b$, record the value of q as $\Delta_{a,b}$. This sweep costs time $O(n)$.

This gives us an algorithm with running time $O(\xi n^2)$, because we have $O(\xi n)$ values $\mathsf{T}_{b,h}$ to compute, each computation taking time $O(n)$.

Note: As we found out after completing the initial version of this manuscript, an algorithm with the same complexity was given earlier in [25]. We have decided to retain the above solution in the paper as it provides useful context for the discrete case considered next, accentuating the contrast between the continuous and discrete variants. Also, recently Damaschke [17] gave a more efficient algorithm for the special case when the interval graph induced by intervals $I_1, I_2, ..., I_n$ is sparse.

Discrete case. For the discrete case, when we schedule unit jobs, a more intricate dynamic programming approach is needed. The fundamental idea of our approach is similar to that in [6,7,8].

A rough intuition here is that scheduling some jobs with short spans, which are more restricted, may create a lot of gaps. (A span of job j is $d_j - r_j + 1$, the length of the interval where it can be scheduled.) We would like to distribute jobs with longer spans, as many as possible, to fill many of these gaps. The remaining gaps may be then filled with jobs that have even longer spans, and so on. Figure 1 shows an example of an instance and a schedule that maximizes throughput for the budget of 2 gaps.

Denote by $\mathcal{J}$ the set of jobs on input, ordered by deadlines, that is $d_1 < d_2 < ... < d_n$. (In Section 2 we showed that we can assume all deadlines to be different.) For each job k and times $u \leq v$, let $\mathcal{J}_{k,u,v}$ denote

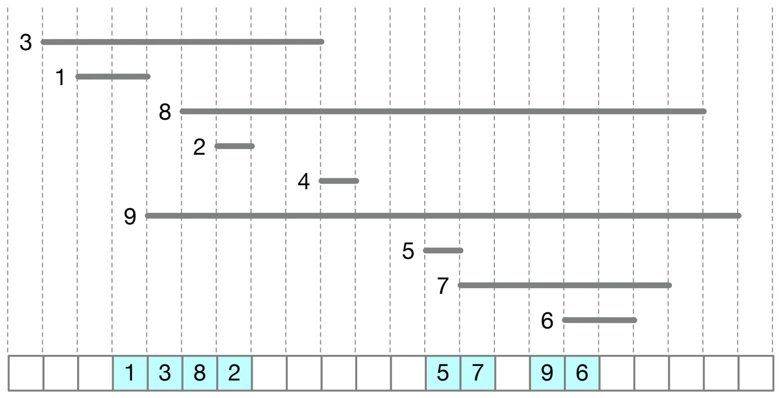
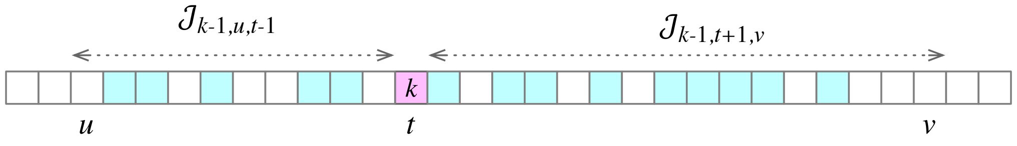


Fig. 1 An example of an instance with $n = 9$ jobs and its schedule with maximum throughput of 8 for the budget of 2 gaps. (There are other optimal schedules.) Each job j is represented by a horizontal line segment starting at slot r_j and ending at slot d_j.

the sub-instance of $\mathcal{J}$ that consists of all jobs $j \in \{1, 2, ..., k\}$ that satisfy $u \le r_j \le v$. Define $\mathsf{T}_{k,u,v,g}$ to be the maximum number of jobs from $\mathcal{J}_{k,u,v}$ that can be scheduled in the interval $[u, v]$ with the number of gaps not exceeding g. Here, the initial and final gap (between u and the first job, and between the last job and v) are also counted, if present.

To derive a recurrence for $\mathsf{T}_{k,u,v,g}$ we reason as follows. If $\mathcal{J}_{k,u,v} = \emptyset$ then $\mathsf{T}_{k,u,v,g} = 0$. If $\mathcal{J}_{k,u,v} \neq \emptyset$ and $k \notin \mathcal{J}_{k,u,v}$ then $\mathsf{T}_{k,u,v,g} = \mathsf{T}_{k-1,u,v,g}$. So for the rest of the derivation assume that $k \in \mathcal{J}_{k,u,v}$.

Consider an optimal schedule S for $\mathcal{J}_{k,u,v}$, that is the one that realizes $\mathsf{T}_{k,u,v,g}$. If k is not scheduled by S, then $\mathsf{T}_{k,u,v,g} = \mathsf{T}_{k-1,u,v,g}$. In the remaining cases we assume that k is scheduled by S, say at time t, where $u \le r_k \le t \le \min(v, d_k)$.



$\mathcal{J}_{k\text{-}1,u,t\text{-}1}$
$\mathcal{J}_{k\text{-}1,t+1,v}$
k
u
t
v

Fig. 2 An illustration of the recurrence for $\mathsf{T}_{k,u,v,g}$.

Naturally, all jobs from $\mathcal{J}_{k-1,t+1,v}$ that are scheduled by S are scheduled in $[t + 1, v]$. As explained in Section 2, we can assume that S has the EDF property. Thus no job from $\mathcal{J}_{k-1,u,t-1}$ can be scheduled in $[t + 1, v]$ because such a job has an earlier deadline than k and so it cannot be pending in S at time t. So all jobs from $\mathcal{J}_{k-1,u,t-1}$ that are scheduled by S are scheduled in $[u, t - 1]$. Further, for the same reason, if there is a job in $\mathcal{J}_{k,u,v} \setminus \{k\}$ released at time t then it cannot be scheduled by S. (In fact, we can assume that such job does not exist, because otherwise we could swap it with k, as k's deadline is larger. But we do not use this observation in the algorithm.)

The above paragraph gives us the optimal substructure property needed for a dynamic-programming formulation. Specifically, using the optimality of S and letting h be the number of gaps in $[u, t - 1]$ in S, we have that the portion of S in $[u, t - 1]$ is a schedule of $\mathcal{J}_{k-1,u,t-1}$ with at most h gaps and maximum throughput, and the portion of S in $[t+1, v]$ is a schedule of $\mathcal{J}_{k-1,t+1,v}$ with at most $g-h$ gaps and maximum throughput. (See Figure 2 for illustration.) Therefore $\mathsf{T}_{k,u,v,g} = \mathsf{T}_{k-1,u,t-1,h} + \mathsf{T}_{k-1,t+1,v,g-h} + 1$.

Overall, for $k \in \mathcal{J}_{k,u,v}$, the argument above gives us the following formula for $\mathsf{T}_{k,u,v,g}$:

$$\mathsf{T}_{k,u,v,g} \;=\; \max \left\{ \begin{array}{c} \mathsf{T}_{k-1,u,v,g} \\ \max\limits_{\substack{r_k \le t \le \min(d_k, v) \\ 0 \le h \le g}} \{\mathsf{T}_{k-1,u,t-1,h} + \mathsf{T}_{k-1,t+1,v,g-h}\} + 1 \end{array} \right\} \qquad (1)$$

The solution of the original instance $\mathcal{J}$ is $\mathsf{T}_{n,r_{\min}-1,d_n+1,\gamma} - 2$, where $r_{\min}$ is the minimum release time. (Recall that d_n is the maximum deadline, by the deadline ordering.) We subtract 2 to account for the initial and final gap which will always be present in the overall solution for $\mathcal{J}_{k,u,v}$ as we start with the interval $[u, v] = [r_{\min} - 1, d_n + 1]$.

To achieve polynomial time we still need to somehow limit the ranges of u, v and t in (1) to some polynomial-size domain. This can be achieved using Lemma 2 which implies that we only need to consider schedules in which every block ends at some release time.

Define $R = \{r_i : 1 \le i \le n\}$ to be the set of all release times, and for any interval $[x, y]$ of integers define $R + [x, y] = \{r + z : r \in R \ \&\ z \in [x, y]\}$. (For $y = x$ we will simplify this notation and write $R + [x]$ instead of $R + [x, x]$.) Then, by the above paragraph, we can assume that all busy slots are in the set $R + [-n+1, 0]$. The slot t in the bottom option on the right-hand side of recurrence (1) is always busy, and the slots u and v are equal, respectively, $t + 1$ and $t - 1$, so they are either busy or are adjacent to a busy slot. Therefore we can restrict the ranges of u, v and t to the set $R + [-n, 1] \cup \{d_n + 1\}$, which has cardinality $O(n^2)$. (We need to also include $d_n + 1$, which is the value of v in the solution for the whole instance $\mathcal{J}$.) This gives us a bound of $O(\gamma n^5)$ on the number of values $\mathsf{T}_{k,u,v,g}$ to be computed, each requiring time $O(\gamma n^2)$. Thus the overall running time is $O(\gamma^2 n^7)$.

A faster algorithm. We now show how to improve this running time by two orders of magnitude. To this end, we further restrict the range of the left endpoint u to the set R. This will involve a slight modification of the recurrence and the instance (adding an artificial "dummy" tight job). The second improvement is obtained by distinguishing two cases, depending on whether or not k is the last job in the optimal schedule. If k is not last, we can reduce the range of t to $R + [-1]$, and if k is last then we can eliminate the maximization over h. The details follow.

As a first step, we claim that we can assume that in the original instance $\mathcal{J}$ the first job is a tight job separated from the rest of the instance, that is $r_1 = d_1 \le \min_{j \ne 1} r_j - 2$. If the first job does not satisfy this property, we can simply add such a job, without affecting the asymptotic running time. The optimal value for the whole instance $\mathcal{J}$ will be computed as $\mathsf{T}_{n,r_{\min},d_n+1,\gamma} - 1$, with 1 subtracted to account for the final gap. So in this initial recursive call the second parameter is in R. (If job 1 was artificially added to $\mathcal{J}$, the optimal solution for $\mathcal{J} \setminus \{1\}$ can be recovered from the optimal solution of $\mathcal{J}$ by subtracting 1, to account for the gap between job 1 and the rest of the schedule.)

Then we proceed by induction. Consider a sub-instance $\mathcal{J}_{k,u,v}$, with $u \in R$, for which we want to compute $\mathsf{T}_{k,u,v,g}$. We can assume that $k \in \mathcal{J}_{k,u,v}$, as otherwise $\mathsf{T}_{k,u,v,g} = \mathsf{T}_{k-1,u,v,g}$. We have two cases, depending on whether k is last or not.

Suppose that k is not last. In this case we can assume that there is a job scheduled right after k, at time $t+1$, for otherwise we could reschedule k as the first job in the next block, without increasing the number of gaps. (Here we use the fact that k has maximum deadline in $\mathcal{J}_{k,u,v}$.) By the EDF property, no scheduled jobs in $\mathcal{J}_{k,u,v} \setminus \{k\}$ are pending at time t. Thus the job scheduled at time $t + 1$, say c, is scheduled at its release time $r_c = t + 1$. Therefore in this case we have $\mathsf{T}_{k,u,v,g} = \mathsf{T}_{k-1,u,t-1,h} + \mathsf{T}_{k-1,t+1,v,g-h} + 1$ for some h (as in recurrence (1)), where $u, t + 1 \in R$, and $t \in R + [-1]$.

Next, assume that k is scheduled last. In this case we do not know how to further restrict the range of t, but instead we can avoid maximization over h. The optimal substructure property holds here as well, that is the portion of S in the interval $[u, t-1]$ must be an optimal schedule for the corresponding sub-instance. Thus the recurrence has two sub-cases: If $t = v$ then there is no final gap and $\mathsf{T}_{k,u,v,g} = \mathsf{T}_{k-1,u,v-1,g} + 1$. Otherwise, there is a final gap and $\mathsf{T}_{k,u,v,g} = \mathsf{T}_{k-1,u,t-1,g-1} + 1$. In both cases the second parameter in the recursive call is $u \in R$ and there is no maximization with respect to h.

Algorithm MaxThrpt. As explained above, we assume in the algorithm that $r_1 = d_1 \le \min_{j \ne 1} r_j - 2$. For all $k = 0, 1, ..., n$ and time slots u, v, where $u \in R$, $v \in R + [-n, 1] \cup \{d_n\}$ and $u \le v$, we process all instances $\mathcal{J}_{k,u,v}$ in order of increasing k, and for each k in order of increasing interval length, $v - u$. For each instance $\mathcal{J}_{k,u,v}$ and each gap budget $g = 0, 1, ..., \gamma$ we compute the corresponding value $\mathsf{T}_{k,u,v,g}$. If some value of $\mathsf{T}_{k,u,v,g}$ appears on the right-hand side of the recurrence with v outside its range (that is when $v \notin R + [-n, 1] \cup \{d_n\}$), then we assume that $\mathsf{T}_{k,u,v,g} = -\infty$.

First, if $\mathcal{J}_{k,u,v} = \emptyset$, we let $\mathsf{T}_{k,u,v,g} = 0$. Assume now that $\mathcal{J}_{k,u,v} \ne \emptyset$. If $k \notin \mathcal{J}_{k,u,v}$ (which means that $r_k \notin [u, v]$) then $\mathsf{T}_{k,u,v,g} = \mathsf{T}_{k-1,u,v,g}$. Otherwise, we compute $\mathsf{T}_{k,u,v,g}$ using the following recurrence:

$$\mathsf{T}_{k,u,v,g} = \max \left\{ \begin{array}{l} \mathsf{T}_{k-1,u,v,g} \\ \max\limits_{\substack{t \in R' \\ 0 \le h \le g}} \{\mathsf{T}_{k-1,u,t-1,h} + \mathsf{T}_{k-1,t+1,v,g-h}\} + 1 \\ \max\limits_{t \in R''} \{\mathsf{T}_{k-1,u,t-1,g-1}\} + 1 \\ \mathsf{T}_{k-1,u,v-1,g} + 1 \quad \text{if } d_k \ge v \end{array} \right\} \tag{2}$$

where the ranges of t above are

$$R' = (R + [-1]) \cap [r_k, \min(d_k, v)]$$
$$R'' = (R + [-n+1, 0]) \cap [r_k, \min(d_k, v)]$$

The algorithm outputs $\mathsf{T}_{n,r_{\min},d_n+1,\gamma} - 1$ as the solution to the whole instance $\mathcal{J}$. (This formula is explained before the statement of the algorithm.)

As discussed earlier, with the above restrictions on u, v and t, we have n choices for u and $O(n^2)$ choices for v. With $n+1$ choices for k and $\gamma+1$ choices for g, the size of the $\mathsf{T}_{k,u,v,g}$ table is $O(\gamma n^4)$. In the above recurrence, in the second option we iterate over up to n choices for t and $\gamma+1$ choices for h, and in the third option we iterate over up to n^2 choices for t. So the overall running time is $O(\gamma n^6)$.

Summarizing, we obtain the following theorem:

Theorem 1 *For any instance $\mathcal{J}$ and a gap budget $\gamma \leq n$, Algorithm* MaxThrpt *in time $O(\gamma n^6)$ computes a schedule of $\mathcal{J}$ that has maximum throughput among all schedules with at most γ gaps.*

Weighted throughput. Theorem 1 easily extends to the model where jobs have non-negative weights and the objective is to maximize the weighted throughput. The only change is that instead of adding 1 in the recurrence for $\mathsf{T}_{k,u,v,g}$ we would add the weight of k. The running time remains the same.

4 Minimizing the Number of Gaps with Throughput Requirement

Suppose now that we want to minimize the number of gaps under a throughput requirement, that is we want to find a schedule that schedules at least a given number $m \in \{0, 1, ..., n\}$ of jobs while minimizing the number of gaps. Without loss of generality we can assume that there exists a schedule with throughput at least m; in fact, as explained in Section 2, we can even assume that the whole instance is feasible.

We can solve this problem, both the continuous and discrete version, by leveraging the algorithms from the previous section. We explain the solution for the continuous variant; the solution of the discrete case can be obtained in an analogous manner.

Recall that $\mathsf{T}_{b,h}$ was defined to be the maximum number of intervals that can be hit with a subset of $\{d_1, d_2, ..., d_b\}$ that includes d_b and has cardinality at most h. We can use all these values to compute T_h, which is the maximum number of intervals that can be hit with a set of cardinality at most h (without any additional restrictions). Computing these for all h will take time $O(n^3)$. By definition, we have $\mathsf{T}_1 \leq \mathsf{T}_2 \leq ... \leq \mathsf{T}_n$. Then, given our requirement m on the throughput, we compute the smallest h for which $\mathsf{T}_h \geq m$. This h is the output of the algorithm. The total running time will be $O(n^3)$.

An essentially identical scheme will produce an algorithm for the discrete case with running time $O(n^7)$, giving us the following result.

Theorem 2 *For any instance $\mathcal{J}$ and $m \leq n$, the above-described algorithm in time $O(n^7)$ computes a schedule of $\mathcal{J}$ that has the minimum number of gaps among all schedules with throughput at least m.*

Comment: The time bounds for the continuous and discrete versions can be refined by expressing them in terms of the optimum number g^* of gaps. This can be achieved by stopping the computation of the recurrence formulas for the smallest h for which the throughput is reached. For the discrete case, the running time will then be $O(g^* n^6)$.

Weighted throughput. Similar to the previous section, Theorem 2 also holds for the weighted throughput case, as the algorithm does not depend on the threshold value m being bounded by n.

5 Maximizing the Number of Gaps

In the preceding sections we studied problems where we were interested in schedules with as few gaps as possible. However, in some applications, gaps in the schedule may actually be desirable. This can arise, for example, when the input stream consists of two types of jobs, some with high priority and other with low priority. High-priority jobs are allowed to reserve their slots in advance, while low-priority jobs are executed only if there are slots available. We can then schedule high-priority jobs first, and maximizing the number of gaps in their schedule would help to improve throughput and latency for low-priority jobs. One such specific scenario appears in QoS networks when coordination of access to a Wi-Fi channel is implemented using so-called point coordination function (PCF) mechanism [28]. One of the features of PCF is that it inserts gaps (in our terminology) into the schedule of high-priority traffic in order to allow low-priority traffic to access the channel.

Thus in this section we will examine the variant of gap scheduling where the objective is to create as many gaps as possible in the schedule. The continuous version of this problem is trivial: for any interval $I_j = [r_j, d_j]$ with $r_j = d_j$, we must of course choose $h_j = r_j$. Each interval $I_j = [r_j, d_j]$ with $r_j < d_j$ can be assigned a unique point $h_j \in I_j$. Thus in this section we will focus only on the discrete model.

Specifically, we are again given an instance $\mathcal{J}$ with n unit jobs with release times and deadlines, and we assume that the instance is feasible, that is all jobs can be scheduled. The objective is to find a schedule for $\mathcal{J}$ (with all jobs scheduled) that maximizes the number of gaps. As before, we will assume that all jobs have different deadlines and different release times, and that they are ordered according to increasing deadlines, $d_1 < d_2 < ... < d_n$. We can also assume that jobs 1 and n satisfy $r_1 = d_1 = \min_{j>1} r_j - 2$ and $d_n = r_n = \max_{j<n} d_j + 2$, that is, they are tight jobs executed at the beginning and end of the schedule, separated by gaps from other jobs. Such jobs can be added to the instance, increasing the number of gaps uniformly by 2 for all schedules; thus the choice of the optimum schedule is not affected, only its value increases by 2. (This is a technical assumption that allows us to fix the range of the dynamic program below.) Figure 3 shows an example of an instance $\mathcal{J}$ with $n = 10$ jobs and its schedule with 7 gaps.

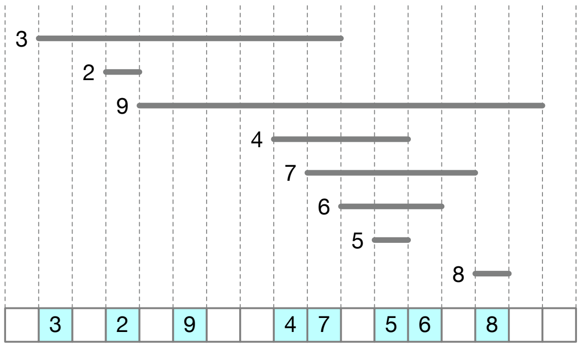


Fig. 3 An example of an instance and its schedule with maximum number of gaps, for $n = 10$. Each job j is represented by a horizontal line segment starting at slot r_j and ending at slot d_j. The special jobs 1 and $n = 10$ are not shown. In this schedule we have 7 gaps, which includes the gap between jobs $1, 3$ and the gap between job $8, 10$.

As in Section 3, for any job $k = 1, 2..., n$ and two time steps $u \le v$ define $\mathcal{J}_{k,u,v}$ to be the sub-instance of $\mathcal{J}$ that consists of all jobs $j \in \{1, 2, ..., k\}$ that satisfy $u \le r_j \le v$. By the assumption about different deadlines, each sub-instance $\mathcal{J}_{k,u,v}$ is feasible. Define $\mathsf{D}_{k,u,v}$ to be the maximum number of gaps in a schedule of $\mathcal{J}_{k,u,v}$ in the interval $[u, v]$. In $\mathsf{D}_{k,u,v}$ we include the extremal gaps in the schedule (if any), namely the initial gap between u and the first job and the final gap between the last job and v.

Lemma 3 *For any sub-instance $\mathcal{J}_{k,u,v}$ there is a schedule S with $\mathsf{D}_{k,u,v}$ gaps in the interval $[u, v]$ that satisfies the following two conditions:*

(i) For any job $j \in \mathcal{J}_{k,u,v}$, if j is scheduled at time S_j then all gaps in the interval $[r_j, S_j]$ have length at most 2 (including the gap between r_j and the first job).

(ii) For each block B of S, either all jobs in B are scheduled at their release times or, assuming that B does not start at u, the gap immediately to the left of B has length 1.

Proof We show that we can modify any schedule S with $\mathsf{D}_{k,u,v}$ gaps to have properties (i) and (ii).

First, suppose that some job j violates property (i), that is S has a gap $[x, x']$ such that $r_j \le x < x + 2 \le x' \le S_j - 1$. We can then move j to time slot $x + 1$. Removing j from time slot S_j can decrease the number of gaps at most by 1 (if j was in a block by itself). Rescheduling j at time $x + 1$ will increase the number of gaps by 1. Thus overall the number of gaps cannot decrease.

If S has a block $B = [y, y']$ that violates property (ii), choose j to be the first job in B with $S_j > r_j$. Since all release times are different, we must have $r_j < y$. We can then move j to slot $y - 1$ and, since the gap that precedes B has length at least 2, the number of gaps will not decrease.

The two operations above convert the current schedule S into a new schedule S' whose set of busy slots is lexicographically smaller than that of S. The number of gaps in S' is the same or larger than the number of gaps in S. Thus this process must eventually end, producing a schedule that satisfies conditions (i) and (ii). $\square$

At the very fundamental level, the idea behind our algorithm is similar to that in Section 3. We use dynamic programming to compute all values $\mathsf{D}_{k,u,v}$. Assume that $k \in \mathcal{J}_{k,u,v}$, for otherwise $\mathsf{D}_{k,u,v} = \mathsf{D}_{k-1,u,v}$. Suppose that, in some optimal schedule S for $\mathcal{J}_{k,u,v}$, k is scheduled at some time $t \in [u,v]$. Obviously, we have $t \geq r_k \in [u,v]$. By the EDF property, t itself cannot be a release time of any job in $\mathcal{J}_{k,u,v}$ other than k. This property is important for the correctness of our recurrence, as it implies that $\mathcal{J}_{k,u,v}$ can be partitioned into three disjoint sets: $\mathcal{J}_{k,u,v} = \mathcal{J}_{k-1,u,t-1} \cup \{k\} \cup \mathcal{J}_{k-1,t+1,v}$. Naturally, all jobs in $\mathcal{J}_{k-1,t+1,v}$ are scheduled by S in $[t+1,v]$. Further, using the EDF property again, all jobs in $\mathcal{J}_{k-1,u,t-1}$ cannot be scheduled after t, so they are all scheduled in $[u,t-1]$. This implies the following optimal substructure property: the portion of S in $[u,t-1]$ is an optimal schedule of $\mathcal{J}_{k-1,u,t-1}$, and the portion of S in $[t+1,v]$ is an optimal schedule of $\mathcal{J}_{k-1,t+1,v}$. We thus conclude that $\mathsf{D}_{k,u,v} = \mathsf{D}_{k-1,u,t-1} + \mathsf{D}_{k-1,t+1,v}$.

Since we do not know t a priori, we can maximize the expression on the right-hand side over all choices of t, giving us the recurrence for $\mathsf{D}_{k,u,v}$ (in the case when $k \in \mathcal{J}_{k,u,v}$):

$$\mathsf{D}_{k,u,v} = \max_{\substack{r_k \leq t \leq \min(v,d_k) \\ t \notin R_{k-1,u,v}}} \{\mathsf{D}_{k-1,u,t-1} + \mathsf{D}_{k-1,t+1,v}\} \tag{3}$$

where we use notation $R_{k-1,u,v}$ for the set of release times of the jobs in $\mathcal{J}_{k-1,u,v}$. Note that the range of the maximum above is not empty, because $r_k \leq \min(v,d_k)$ and $r_k \notin R_{k-1,u,v}$, so r_k is a candidate for t. We still need to show that we can reduce the ranges of u, v and t in (3) to some polynomial-size domain.

We claim that we only need to consider instances $\mathcal{J}_{k,u,v}$ where $u,v \in R + [-1,3n+1]$. (See Section 3 for the definition of sets $R + [x,y]$.) Indeed, this follows from Lemma 3(i), which implies that in the recurrence (3) for $\mathsf{D}_{k,u,v}$ we only need to consider slots t between r_k and $r_k + 3n$, inclusive. Thus, in the sub-instances $\mathcal{J}_{k-1,u,t-1}$ and $\mathcal{J}_{k-1,t+1,v}$ the new arguments $v' = t-1$ and $u' = t+1$ will satisfy $v', u' \in \{r_k - 1, r_k, ..., r_k + 3n + 1\} \subseteq R + [-1, 3n+1]$. The initial arguments are r_1 and $d_n = r_n$, both in $R + [-1,3n+1]$, completing the proof of our claim. As $|R + [-1,3n+1]| = O(n^2)$, this gives us $O(n^5)$ instances $\mathcal{J}_{k,u,v}$ to consider. For each $\mathcal{J}_{k,u,v}$, using Lemma 3(i), to compute $\mathsf{D}_{k,u,v}$ it is sufficient to iterate only over $t = r_k, r_k+1, ..., \min(v, d_k, r_k+3n)$. This would give us the overall running time $O(n^6)$.

Next, we argue that this running time can be further improved to $O(n^5)$. The general idea is to show that, in essence, the recurrence (3) needs to be applied only to $O(n)$ values of u. To this end, we modify recurrence (3) as follows:

$$\mathsf{D}_{k,u,v} = \max_{\substack{r_k \leq t \leq \min(v,d_k) \\ t \notin R_{k-1,u,v}}} \left\{\mathsf{D}_{k-1,u,t-1} + \mathsf{D}_{k-1,\mu(t),v}\right\} \tag{4}$$

where $\mu(t)$ is determined based on three cases: If $\mathcal{J}_{k-1,t+1,v} = \emptyset$, let $\mu(t) = v+1$. Otherwise, let $\mu' = \min\{r_j : j \in \mathcal{J}_{k-1,t+1,v}\}$. If $\mu' = t+1$, let $\mu(t) = t+1$, otherwise let $\mu(t) = \mu' - 1$. (Note that $\mu(t)$ depends also on v and k, but we omit these in our notation to reduce clutter.)

We claim that (4) is a correct recurrence for $\mathsf{D}_{k,u,v}$, providing that $k \in \mathcal{J}_{k,u,v}$. Indeed, from the definition of $\mu(t)$ we have $\mathcal{J}_{k-1,t+1,v} = \mathcal{J}_{k-1,\mu(t),v}$, and sub-instance $\mathcal{J}_{k-1,\mu(t),v}$ is scheduled inside the interval $[\mu(t), v]$. Finally, the optimal schedules of $\mathcal{J}_{k-1,t+1,v}$ and $\mathcal{J}_{k-1,\mu(t),v}$ have the same number of gaps. (The reason for distinguishing between the cases when $\mu' = t+1$ and $\mu' \neq t+1$ was to take into account the initial gap.)

Using (4), the recurrence will remain correct if we restrict the range of u's to the set $R + [-1,0]$, whose cardinality is $O(n)$. Then the total number of instances $\mathcal{J}_{k,u,v}$ to consider is $O(n^4)$, implying the running time of $O(n^5)$. The complete algorithm is described below.

Algorithm MaxGaps**.** We consider all instances $\mathcal{J}_{k,u,v}$, where u and v are time slots such that $u,v \in [r_1, d_n]$ and $u \leq v+1$, and k is either a job, that is $k \in \{1,2,...,n\}$, or $k = 0$. We process these instances in order of increasing k and increasing difference $v - u$. For each instance $\mathcal{J}_{k,u,v}$, the value of $\mathsf{D}_{k,u,v}$ is computed as follows.

We first deal with the base case, when $\mathcal{J}_{k,u,v} = \emptyset$. Then, if $u = v+1$ we let $\mathsf{D}_{k,u,v} = 0$, and if $u \leq v$ we let $\mathsf{D}_{k,u,v} = 1$.

So assume now that $\mathcal{J}_{k,u,v} \neq \emptyset$, which implies that $u \leq v$ and $k \geq 1$. Then, if $k \notin \mathcal{J}_{k,u,v}$ we let $\mathsf{D}_{k,u,v} = \mathsf{D}_{k-1,u,v}$. Otherwise we have $k \in \mathcal{J}_{k,u,v}$, in which case we compute $\mathsf{D}_{k,u,v}$ using the following recurrence:

$$\mathsf{D}_{k,u,v} = \max_{\substack{r_k \leq t \leq \min(v,d_k,r_k+3n) \\ t \notin R_{k-1,u,v}}} \left\{\mathsf{D}_{k-1,u,t-1} + \mathsf{D}_{k-1,\mu(t),v}\right\}$$

After all values are computed, the algorithm outputs D_{n,r_1,d_n}. By the analysis above, we obtain the following theorem.

Theorem 3 *For any instance $\mathcal{J}$, Algorithm* MaxGaps *in time $O(n^5)$ computes a schedule of $\mathcal{J}$ with maximum number of gaps.*

6 Minimizing Maximum Gap

In the earlier sections we focussed on the number of gaps in the schedule. For certain applications, the *size* of the gaps is also of interest. In this section we will study the problem where the objective is to minimize the maximum gap in the schedule. Such schedules tend to spread the jobs more uniformly over the time range and produce many gaps, which may be useful in applications discussed in Section 5, where a good schedule should leave some gaps between high-priority jobs, to allow other jobs to access the processor. This could also be useful in temperature control of the processor (see the discussion at the end of Section 11).

The general setting is as before. We have an instance $\mathcal{J}$ consisting of n unit jobs, where job j has release time r_j and deadline $d_j \geq r_j$. As explained in Section 2, we can assume that $\mathcal{J}$ is feasible. The objective is to compute a schedule of all jobs that minimizes the maximum gap size.

Interestingly, this problem is structurally different from these in the previous sections, because now, intuitively, a good schedule should spread the jobs more-or-less evenly in time. For example, if we have $n-2$ jobs released at 0, all with deadline $D \gg n$, plus two more tight jobs 1 and n in time slots 0 and D, respectively, then we should schedule the non-tight jobs $j = 2, 3, ..., n-1$ at time slots $\approx (j-1)\frac{D}{n-1}$. In contrast, the algorithms in Sections 3 and 4 attempted to group the jobs into a small number of blocks. Similar to the objective in Section 5, a schedule that minimizes the maximum gap size will typically create many gaps, but, as can be seen in Figure 4, these two objective functions will in general produce different schedules.

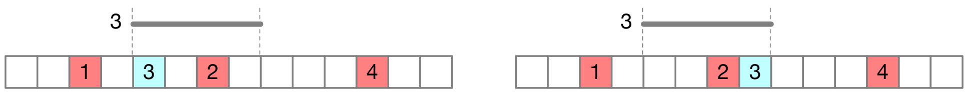


Fig. 4 An instance with two schedules. Red/dark shaded slots represent tight jobs. The range of job 3 is represented by a horizontal segment. The schedule on the left maximizes the number of gaps. The schedule on the right minimizes the maximum gap. Both schedules are unique optimal solutions for their respective objective functions.

In this section we give an $O(n^2 \log n)$-time algorithm for computing schedules that minimize the maximum gap. We first give an algorithm for the continuous model, and then extend it to the discrete model.

6.1 The Continuous Case

The continuous analogue of our scheduling problem can be formulated as follows. The input consists of n intervals $I_1, I_2, ..., I_n$. As before, $I_j = [r_j, d_j]$ for each j. The objective is to compute a hitting set H for these intervals that minimizes the maximum gap between its consecutive points. Another way to think about this problem is as computing a *representative* $h_j \in H \cap I_j$ for each interval I_j. Except for degenerate situations (two equal intervals of length 0), we can assume that all representatives are different, although we will not be using this property in our algorithm, and we treat H as a multiset.

We order the intervals so that $d_1 \leq d_2 \leq ... \leq d_n$. (In this continuous version we cannot assume all d_j's are different without losing generality.) Further, we only need to be concerned with sets H that contain d_1, because if H contains any points before d_1 then we can replace them all by d_1 without increasing the maximum gap in H. Also, if $\max_i r_i \leq d_1$ then there is a singleton hitting set, $H = \{d_1\}$, whose maximum gap is equal to 0. Thus we can also assume that $\max_i r_i > d_1$, so that we need at least two points in H.

Consider first the decision version: *"Given $\lambda > 0$, is there a hitting set H for $I_1, I_2, ..., I_n$ in which all gaps are at most λ?"* If λ has this property, we will call it *viable*. We first give a greedy algorithm for this decision version and then later we show how to use it to obtain an efficient algorithm for the minimization version.

Algorithm Viable(λ). We initialize $h^\circ_1 = d_1$ and $U = \{2, 3, ..., n\}$. U represents the set containing the indices of intervals that do not have yet representatives selected. We move from left to right, at each step assigning a representative to one interval in U, placing this representative as far to the right as possible, and we remove this interval from U.

Specifically, at the beginning of a step $j \geq 2$ the current set of representatives is $h^\circ_1, h^\circ_2, ..., h^\circ_{j-1}$, listed in non-decreasing order. In this step we proceed as follows. Let $z = h^\circ_{j-1} + \lambda$. If all $j \in U$ satisfy $r_j > z$, declare failure and return false. Otherwise, choose $j \in U$ with $r_j \leq z$ that minimizes d_j and remove j from U. We now have two cases. If $d_j \leq z$, let $h^\circ_j = d_j$, and otherwise (that is, when $r_j \leq z < d_j$) let $h^\circ_j = z$. Then increment j and continue. If the process completes with $U = \emptyset$ (and thus also $j = n$), return true and the computed solution $H^\circ = \{h^\circ_1, h^\circ_2, ..., h^\circ_n\}$.

To show correctness of Algorithm Viable(λ), let $H = \{h_1, h_2, ..., h_n\}$ be some solution in increasing order and with all gaps at most λ. We show that this solution can be converted into the one computed by our algorithm. For $j = 1$, as we explained earlier, we can assume that $h_1 = d_1$, so $h_1 = h^\circ_1$. Consider the first step when Algorithm Viable(λ) chooses some $h^\circ_j \neq h_j$. (If there is no such step, we are done.) By the choice of h°_j in the algorithm, we have that $h_j < h^\circ_j$. (Otherwise, either the gap between h_{j-1} and h_j would exceed λ or H would not hit I_j.) We can then replace h_j by h°_j in H, without increasing the gap size to above λ. This way, we increase the number of steps of Algorithm Viable(λ) that produce the same representatives as those in H. So repeating this process sufficiently many times eventually converts H into the set H°.

We claim that Algorithm Viable(λ) can be implemented in time $O(n \log n)$. Instead of U, the algorithm maintains a set $U' \subseteq U$ that, when a step $j \geq 2$ starts, consists of indices i for which $r_i \leq h^\circ_{j-1} + \lambda$ and I_i does not yet have a representative. Store U' in a priority queue with priority values equal to the deadlines. Then choosing the new interval I_j in the algorithm and removing j from U' takes time $O(\log n)$. When j is incremented (after adding h°_j to the solution), the indices of new intervals are inserted into U' in order of release times (which can be sorted in the pre-processing stage), with each insertion taking time $O(\log n)$.

Now, the idea is to use Algorithm Viable(λ) as an oracle in binary search on λ's. For this to work, we need to be able to efficiently identify a small set of candidate values for the optimal λ. Let

$$\Lambda \;=\; \left\{ \frac{r_i - d_j}{k} : k \in \{1, 2, ..., n-1\}, i, j \in \{1, 2, ..., n\}, r_i > d_j \right\}.$$

Observe that $|\Lambda| = O(n^3)$ and, by our assumption that $\max_i r_i > d_1$, also $\Lambda \neq \emptyset$.

We claim that Λ contains the optimal gap length λ^*. The argument is this. Consider some hitting set $H^* = \{h^*_1, h^*_2, ..., h^*_n\}$ whose maximum gap is λ^*, sorted in non-decreasing order. Choose some maximal (w.r.t. inclusion) consecutive sub-sequence $h^*_a < h^*_{a+1} < ... < h^*_b$ with all gaps equal to λ^*, and suppose that h^*_a is not a deadline. Then we can move h^*_a by a little bit to the right without creating a gap longer than λ^*. Similarly, if h^*_b is not a release time then we can apply a similar procedure to h^*_b and shift it to the left. Each such operation reduces the number of gaps of length λ^*. Since λ^* is optimal, eventually we must get stuck, meaning that we will find a sub-sequence like the one above with the first and last indices a and b that satisfy $h^*_a = d_j$ and $h^*_b = r_i$, for some i and j. Then we will have $\lambda^* = \frac{r_i - d_j}{b-a} \in \Lambda$.

The idea above immediately yields an $O(n^3 \log n)$-time algorithm. This algorithm first computes the set Λ, sorts it, and then finds the optimal λ through binary search in Λ. Note that the running time is dominated by sorting Λ.

We now show that this running time can be improved to $O(n^2 \log n)$, by conducting a more careful search in Λ that avoids constructing Λ explicitly. The basic idea is to use a smaller set Δ that consists of all values $r_i - d_j$ where $r_i > d_j$. This set Δ implicitly represents Λ, in the sense that it consists of all numerator values of the fractions in Λ. More precisely, each value in Λ can be expressed as x/k, for some $x \in \Delta$ and $1 \leq k \leq n-1$. One can visualize Λ by representing such values x/k as points in 2D, with the two coordinates representing the values of x and k, and point (x, k) representing x/k (see Figure 5). Roughly, the algorithm then finds two consecutive values v, w in Δ such that $w/(n-1)$ is viable but $v/(n-1)$ is not. It then finds an index κ such that v/κ is viable but $v/(\kappa+1)$ is not. Then the optimum value of λ must be between v/κ and $v/(\kappa+1)$. We then show that there are only $O(n^2)$ such values in Λ, so by doing a binary search among these values we can find the optimum λ in time $O(n^2 \log n)$. A detailed algorithm with complete analysis follows.

Algorithm MinMaxGap. The algorithm is described below in Pseudocode 1. In this pseudo-code, to avoid multi-level nesting, we assume that the algorithm terminates if the **return** statement is reached.

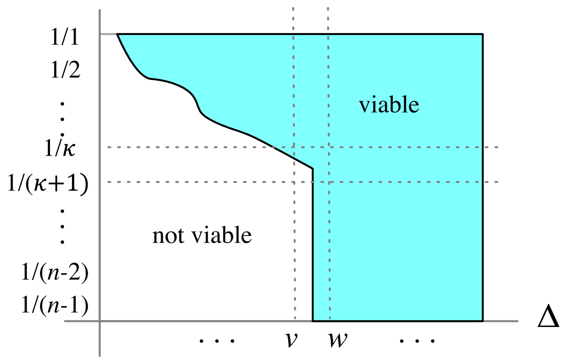


Fig. 5 An illustration of the the idea behind Algorithm MinMaxGap. Viable fractions in Λ are represented by the shaded region.

Pseudocode 1 Algorithm MinMaxGap

```
1: if $\max_i r_i \le d_1$ then return 0
2: $\Delta \leftarrow \{r_i - d_j : r_i > d_j, i, j \in \{1, 2, ..., n\}\}$
3: sort $\Delta$ in non-decreasing order
4: if Viable($\frac{\min(\Delta)}{n-1}$) then return $\frac{\min(\Delta)}{n-1}$
5: $v \leftarrow \max\left\{x \in \Delta : \text{Viable}(\frac{x}{n-1}) = \mathsf{false}\right\}$
6: $w \leftarrow \min\{x \in \Delta : x > v\}$
7: if Viable($v$) = false then return $\frac{w}{n-1}$
8: $\kappa \leftarrow \max\{k \in \{1, 2, ..., n-1\} : \text{Viable}(\frac{v}{k}) = \mathsf{true}\}$
9: $\Lambda' \leftarrow \left\{\frac{x}{\lceil \kappa x/v \rceil} : x \in \Delta \text{ and } \frac{v}{\kappa+1} < \frac{x}{\lceil \kappa x \rceil / v} \le \frac{v}{\kappa}\right\} \cup \left\{\frac{w}{n-1}\right\}$
10: sort $\Lambda'$ in non-decreasing order
11: return $\min\{\lambda \in \Lambda' : \text{Viable}(\lambda) = \mathsf{true}\}$
```

We now explain the steps in the algorithm and justify correctness and the running time. First, if $\max_i r_i \le d_1$ then there is a hitting set with all representatives on one point, and we return 0 as the optimum value (Line 1).

Otherwise we have $\max_i r_i > d_1$, that is any hitting set needs at least two points and the optimal gap is strictly positive. We then compute all positive values $r_i - d_j$, store them in a set Δ and sort them (Lines 2-3). This will take time $O(n^2 \log n)$.

If $\frac{\min(\Delta)}{n-1}$ is viable (which we check in Line 4), then this is the optimal value, since no hitting set can have all gaps smaller than $\frac{\min(\Delta)}{n-1} = \min(\Lambda)$. We can thus now assume that $\frac{\min(\Delta)}{n-1}$ is not viable.

Next, we compute the largest $v \in \Delta$ for which $\frac{v}{n-1}$ is not viable. By the previous paragraph, such v exists. To this end, we can do binary search in the set $\left\{\frac{x}{n-1} : x \in \Delta\right\}$, at each step making calls to Viable() to determine whether the current split value is viable or not. The binary search will take time $O(n^2 \log n)$. We also let w to be the next value in Δ after v. (If there is no such value, let $w = \infty$.)

At this point we check whether v is viable. If it is not, it means that for all $x \in \Delta$ with $x \le v$, all fractions x/k, for $k = 1, 2, ..., n-1$, are not viable as well. Then the smallest viable value in Λ must be $\frac{w}{n-1}$, so we output $\frac{w}{n-1}$ in Line 7. (Note that in this case w must exist, because if v were the largest value in Δ then v would be viable.)

If v is viable, we compute the largest κ for which v/κ is viable (Line 8). By the choice of v we have $\kappa < n-1$. We now also know that the optimal value for λ has the form $\frac{x}{k} \in \Lambda$ where $x \in \Delta$, $x \le v$, and

$$\frac{v}{\kappa+1} < \frac{x}{k} \le \frac{v}{\kappa}. \tag{5}$$

So we only need to search for λ among such values.

Next, we define a small set Λ' that contains all candidate values from the previous paragraph. To this end, we claim that for any $x \in \Delta$, if $x \le v$ then there is at most one integer $k_x \in \{1, ..., n-1\}$ for which condition (5) holds. This follows from simple calculation, as (5) implies that

$$\frac{x}{v} \cdot \kappa \;\le\; k < \frac{x}{v} \cdot \kappa + \frac{x}{v} \;\le\; \frac{x}{v} \cdot \kappa + 1.$$

Thus the only candidate for k_x is $k_x = \lceil \frac{x}{v} \cdot \kappa \rceil$.

The above argument gives us that the only candidates for the optimal gap size we need to consider are all values x/k_x, for $x \in \Delta$ and $x \le v$, plus the value $\frac{w}{n-1}$ that we identified before as another candidate. In Lines 9-10 we let Λ' be the set of these candidates and we sort them in non-decreasing order. Finally, we find the smallest viable value in Λ'. As $|\Lambda'| = O(n^2)$, this can be done in time $O(n^2 \log n)$ with binary search that calls VIABLE() for each split value.

6.2 The Discrete Case

We now show that Algorithm MINMAXGAP from the previous section can be adapted to the discrete case, namely to scheduling unit jobs.

Let $\mathcal{J}$ be an instance of unit job scheduling with release times and deadlines. As explained in Section 2, we can now assume without loss of generality (and in contrast to the continuous case) that all deadlines are different and sorted in increasing order, $d_1 < d_2 < ... < d_n$.

We treat $\mathcal{J}$ as a collection of intervals $I_j = [r_j, d_j]$, $j = 1, 2, ..., n$, and run Algorithm MINMAXGAP. This will produce a set of (real-valued) representatives $H = \{h_1, h_2, ..., h_n\}$ for the intervals in $\mathcal{J}$. (Here h_j denotes the representative of interval I_j, so the elements in H may not be in increasing order.) Let λ be the maximum gap between these representatives. Since λ is an optimal gap for the continuous variant, $\bar{\lambda} = \lceil \lambda \rceil - 1$ is a lower bound on the optimal gap length for the discrete variant. (We need to subtract 1 to account for unit length of jobs.) It is thus enough to construct a schedule with all gaps of length at most $\bar{\lambda}$.

Recall that Algorithm VIABLE(λ) either assigns jobs to their deadlines or it spaces consecutive jobs at intervals of λ between some deadline and some release time. As explained before, without loss of generality we can assume that job 1 is scheduled at d_1, and Algorithm VIABLE(λ) will in fact produce $h_1 = d_1$. If all other h_i's are also deadlines, we are done. Otherwise, the rough idea is to tentatively assign each job j to h_j (which may not be integral), and then, going from left to right, gradually shift each job to the first available slot after h_j. This does not quite work, because when several intervals have their representatives in the same slot, this could force some jobs past their deadlines. So the correct process needs to be more subtle and allow for some job reordering, as described below.

Procedure ADJUST(λ)**.** We describe how to convert H into a schedule S of $\mathcal{J}$. Start by initializing $S_1 = d_1$ and $P = \emptyset$. (Set P represents pending jobs that are "delayed", namely those whose representatives' values in H are before or at the current slot.) Then consider slots $t = d_1 + 1, d_1 + 2, ...,$ one by one. For each such t, first add to P all jobs j with $\lceil h_j \rceil = t$. If $P \ne \emptyset$, choose j to be the job in P with minimum d_j, let $S_j = t$, and remove j from P. Then increment t to $t + 1$ and continue.

We claim that $S = (S_1, S_2, ..., S_n)$ is a feasible schedule. By the way we add jobs to P, if $j \in P$ when we consider slot t then $r_j \le h_j \le t$. Since also $h_j \le d_j$, each job will be added to P not later than when processing slot $t = d_j$. Also, the assumption about different deadlines implies (by simple induction) that when we consider a slot t then all jobs in P have deadlines at least t; in particular this gives us that no job will miss its deadline. Thus $S_j \in [r_j, d_j]$ for all $j \in \mathcal{J}$.

Next, we show that the maximum gap size in S is equal to $\bar{\lambda}$. Obviously (see above), it cannot be smaller. To show that it is not larger, consider a tentative assignment $Q = \{Q_1, Q_2, ..., Q_n\}$ of jobs to slots defined by $Q_j = \lceil h_j \rceil$, for all $j \in \mathcal{J}$. (This is not a feasible schedule because it may assign different jobs to the same slot.) We first show that the maximum gap in this assignment is at most $\bar{\lambda}$. Consider two jobs j and j' that are consecutive in Q; that is, $Q_j < Q_{j'}$ and there is no job ℓ with $Q_j < Q_\ell < Q_{j'}$. We can assume that $h_j = \max\{h_\ell : Q_\ell = Q_j\}$ and $h_{j'} = \min\{h_\ell : Q_\ell = Q_{j'}\}$. Then j and j' are also consecutive in H and the length of the gap between them is $h_{j'} - h_j \le \lambda$. We then have

$$Q_{j'} \;=\; \lceil h_{j'} \rceil \le \lceil h_j + \lambda \rceil \;\le\; \lceil h_j \rceil + \lceil \lambda \rceil \;\le\; Q_j + 1 + \bar{\lambda}.$$

Thus all gaps in Q are at most $\bar{\lambda}$. But all slots of Q are also used by S because, in Procedure ADJUST(λ), when we consider slot $t \in Q$ set P is not empty. This implies that the gaps in S are bounded from above by $\bar{\lambda}$. We can thus conclude that S is optimal.

The way we described Procedure ADJUST(λ), its running time would not be bounded by a function of n. This is easy to fix by skipping all the slots t for which the current set P is empty. Specifically, we do this: Suppose that when we process a slot t we have $P \neq \emptyset$. If $|P| \geq 2$ then P remains non-empty after scheduling a job in slot t, so in this case we increment t by 1. Otherwise, we increment it to the first value $\lceil h_j \rceil$ after t. This way we will only examine n slots. With routine data structures, this approach will give us running time $O(n \log n)$.

The discussion above focussed only on computing the optimum gap size. Given this value and using Algorithm VIABLE(), one can also compute an actual optimum schedule. Summarizing, we obtain the following theorem.

Theorem 4 *For any instance $\mathcal{J}$, Algorithm* MINMAXGAP *(adapted for the discrete case, as explained above) in time $O(n^2 \log n)$ computes a schedule of $\mathcal{J}$ whose maximum gap value is minimized.*

7 Minimizing Total Flow Time with a Budget for Gaps

Unlike in earlier sections, we now consider jobs without deadlines and focus on the tradeoff between the number of gaps and the delay of jobs. Formally, an instance $\mathcal{J}$ is given by a collection of n unit length jobs. For each job $j = 1, 2, ..., n$ we are given its release time r_j. If, in some schedule S, job j is executed at time S_j then $F_j = S_j - r_j$ is called the *flow time of j in S*. We are also given a budget value γ for the number of gaps. The objective is to compute a schedule S for $\mathcal{J}$ that minimizes the total flow time $F_\Sigma(S) = \sum_j F_j$ among all schedules with at most γ gaps. Figure 6 shows an example of an instance and a schedule with two gaps.

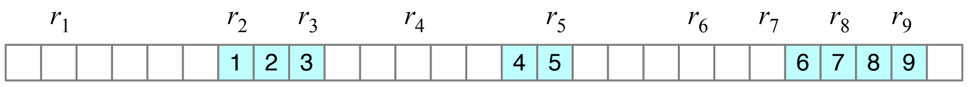


Fig. 6 An instance and its schedule with two gaps and total flow value $F_1 + ... + F_9 = 5+1+0+3+0+3+2+1+0 = 15$.

Continuous case. The continuous variant of this problem is equivalent to the *k-medians problem* on a directed line: Given points $r_1, r_2, ..., r_n$, find a set H of k points that minimizes the sum

$$\sum_{i=1}^{n} \min_{\substack{h \in H \\ h \geq r_i}} (h - r_i),$$

where the ith term of the sum represents the distance between r_i and the first point in H after r_i. (Here, the value of k corresponds to $\gamma - 1$, the number of blocks in the discrete schedule.) This is a well-studied problem and it can be solved in time $O(kn)$ if the points are given in a sorted order [29]. Prior to the work in [29], the undirected case on the line was addressed in [23,4], and extension to trees have also been studied – see [16], for example, and references therein.

Discrete case. The discrete case differs from its continuous analogue because the jobs executed in the same block do not occupy a single point. Nevertheless, we show that the techniques for computing k-medians can be adapted to minimum-flow scheduling with gaps, resulting in an algorithm with running time $O(n \log n + \gamma n)$.

Without loss of generality, we assume that all release times are different and ordered in increasing order, that is $r_1 < r_2 < ... < r_n$. Any instance can be modified to have this property in time $O(n \log n)$. As explained in Section 2, this modification changes the flow of all schedules uniformly, so the optimality is not affected. Sorting the release times is the only part of the algorithm that requires time $O(n \log n)$; the remaining part will run in time $O(\gamma n)$.

We first give a simple dynamic programming formulation with running time $O(\gamma n^2)$, and then show how to improve it to $O(\gamma n)$. Any schedule with at most γ gaps consists of at most $\gamma + 1$ blocks. To reduce the running time, we need to show that these blocks can only be located at a small number of possible places. For this, we will need the following lemma, that follows directly from Lemma 2 and an exchange argument.

Lemma 4 *There is an optimal schedule with the following properties: (i) all jobs are scheduled in order of their release times, and (ii) the last job of each block is scheduled at its release time.*

Based on this lemma, each block consists of consecutive jobs, say $i, i+1, ..., j$, with the last job j scheduled at time r_j. The hth job of the block is scheduled at time $r_j - j + h$. So the contribution of this block to the total flow is

$$\begin{aligned} W_{i,j} &= \sum_{h=i}^{j} F_h = \sum_{h=i}^{j-1} (r_j - j + h - r_h) \\ &= (j-i)r_j - \binom{j-i+1}{2} - R_{j-1} + R_{i-1}, \end{aligned}$$

where $R_b = \sum_{a=1}^{b} r_a$, for each job b.

A simple $O(\gamma n^2)$-time algorithm. For each $j = 0, 1, ..., n$, define $\mathcal{J}_j$ to be the sub-instance of $\mathcal{J}$ consisting of jobs $1, 2, ..., j$. Let $\mathsf{F}_{j,g}$ denote the minimum total flow of a schedule for $\mathcal{J}_j$ with at most g gaps, where $g \leq \gamma$. We initialize $\mathsf{F}_{0,g} = 0$ for all $g = 0, 1, ..., \gamma$ and $\mathsf{F}_{j,0} = W_{1,j}$ for $j = 1, 2, ..., n$. Then, for $j = 1, 2, ..., n$ and $g = 1, 2, ..., \gamma$, we compute

$$\mathsf{F}_{j,g} = \min_{1 \leq i \leq j} \{\mathsf{F}_{i-1,g-1} + W_{i,j}\}.$$

The algorithm returns $\mathsf{F}_{n,\gamma}$ as the optimum value for the whole instance $\mathcal{J}$.

To justify correctness, we need to explain why the above recurrence holds. Consider a schedule that realizes $\mathsf{F}_{j,g}$. From Lemma 4, since we are minimizing the total flow, we can assume that job j is scheduled at r_j. Let i be the first job of the last block. As we calculated earlier, the contribution of this block to the total flow is $W_{i,j}$. The schedule for the remaining jobs, $1, 2, ..., i-1$, has at most $g-1$ gaps and must have optimum total flow time, so (inductively) its total flow time is equal $\mathsf{F}_{i-1,g-1}$.

We now consider the running time. All values $W_{i,j}$ can be precomputed in time $O(n^2)$. We have $\gamma + 1$ choices for g and $n+1$ choices for j, so there are $O(\gamma n)$ values $\mathsf{F}_{j,g}$ to compute. Computing each value takes time $O(n)$, for the total running time $O(\gamma n^2)$.

An $O(\gamma n)$-time algorithm. To improve the running time to $O(\gamma n)$, we show that the values $W_{i,j}$ satisfy the Monge property[1] (see, for example [29,10,9]).

Lemma 5 *For all $1 \leq i \leq i' \leq j \leq j' \leq n$, we have*

$$W_{i,j} + W_{i',j'} \leq W_{i,j'} + W_{i',j}.$$

Proof It is well known (see [10,9], for example), and easy to prove, that it is sufficient to prove the inequality in the lemma for $i' = i+1$ and $j' = j+1$, that is

$$W_{i,j} + W_{i+1,j+1} \leq W_{i,j+1} + W_{i+1,j}. \tag{6}$$

To show (6), we compute $W_{i,j} - W_{i+1,j}$ and $W_{i+1,j+1} - W_{i,j+1}$ separately:

$$\begin{aligned} W_{i,j} - W_{i+1,j} &= \left[(j-i)r_j - \binom{j-i+1}{2} - R_{j-1} + R_{i-1}\right] \\ &\quad - \left[(j-i-1)r_j - \binom{j-i}{2} - R_{j-1} + R_i\right] \\ &= r_j - j + i - r_i, \end{aligned}$$

and

$$\begin{aligned} W_{i+1,j+1} - W_{i,j+1} &= \left[(j-i)r_{j+1} - \binom{j-i+1}{2} - R_j + R_i\right] \\ &\quad - \left[(j+1-i)r_{j+1} - \binom{j-i+2}{2} - R_j + R_{i-1}\right] \\ &= -r_{j+1} + j + 1 - i + r_i. \end{aligned}$$

[1] For upper triangular matrices this property is often referred to as the quadrangle inequality. This distinction is only cosmetic, as we can also think of $[W_{i,j}]$ as a full square matrix by filling the lower triangle of the matrix with ∞ values.

Adding these equations, we get

$$W_{i,j} + W_{i+1,j+1} - W_{i,j+1} - W_{i+1,j} \; = \; r_j - r_{j+1} + 1 \; \le \; 0,$$

because $r_j < r_{j+1}$, due to our assumption that all release times are different. This completes the proof of (6) and the lemma. □

Algorithm MinTotFlow. With Lemma 5, the improved algorithm follows the standard method of speeding-up dynamic programming by leveraging the Monge property, and is essentially the same as in [29]. We briefly outline it here for the sake of readers unfamiliar with this approach. First, we sort the jobs in order of release times. This will cost time $O(n \log n)$. Unlike in the $O(\gamma n^2)$-time algorithm above, now we will *not* pre-compute all values $W_{i,j}$, as this would cost time $O(n^2)$. Instead, in time $O(n)$ we precompute only all values $R_b = \sum_{a=1}^{b} r_a$, for $b = 1, 2, ..., n$. With these values precomputed, we can compute each value $W_{i,j}$ in time $O(1)$ whenever it's needed. The algorithm then loops on $g = 1, 2, ..., \gamma$, and for any given g it computes all n values $\mathsf{F}_{j,g}$, for $j = 1, 2, ..., n$. To this end, consider iteration g, when the values $\mathsf{F}_{j,g-1}$ are already computed for all j. Define an auxiliary function $V_{i,j} = \mathsf{F}_{i-1,g-1} + W_{i,j}$. We think of $[V_{i,j}]$ as an implicit matrix whose values can be each computed in time $O(1)$, when needed. Further, using Lemma 5 it is easy to show that this matrix $[V_{i,j}]$ also satisfies the Monge property. (The extra F-terms in the Monge property for $[V_{i,j}]$ cancel out, reducing the inequality to Lemma 5.) By exploiting this property, in iteration g all minima $\mathsf{F}_{j,g} \; = \; \min_{1 \le i \le j} V_{i,j}$, for $j = 1, 2, ..., n$, can be computed in time $O(n)$ using the classical algorithm from [1]. With the whole matrix $[\mathsf{F}_{j,g}]$ computed, the algorithm returns $\mathsf{F}_{n,\gamma}$. The overall running time is $O(n \log n + \gamma n)$.

Theorem 5 *For any instance $\mathcal{J}$, Algorithm* MinTotFlow *(as outlined above) in time $O(n \log n + \gamma n)$ computes a schedule of $\mathcal{J}$ that has minimum total flow among all schedules with at most γ gaps.*

8 Minimizing Number of Gaps with a Bound on Total Flow

An alternative way to formulate the tradeoff in the previous section would be to find a schedule that minimizes the number of gaps, given a budget f for the total flow F_Σ. This can be reduced to the previous problem by finding the smallest g for which there is a schedule with at most g gaps and total flow at most f. Our solution is the same for both the continuous and discrete versions, so we focus only on the discrete variant.

Using the notation from the previous section, $\mathsf{F}_{n,g}$ represents the minimum total flow of a schedule with at most g gaps. Then $\mathsf{F}_{n,g} = W_{1,n}$, $\mathsf{F}_{n,n-1} = 0$, and $\mathsf{F}_{n,g}$ is non-increasing as g increases. Algorithm MinTotFlow computes the values of matrix $[\mathsf{F}_{j,g}]$ column by column, that is in order of increasing g. We can then adapt this algorithm to stop as soon as it finds g for which $\mathsf{F}_{n,g} \le f$. Then the minimum number of gaps is $g^* = g$. This gives us the following result.

Theorem 6 *For any instance $\mathcal{J}$ and a flow bound f, the above modification of Algorithm* MinTotFlow *in time $O(n \log n + g^* n)$ computes a schedule of $\mathcal{J}$ that minimizes the number of gaps among all schedules with total flow at most f. (Here, $g^* \le n$ denotes the number of gaps in the optimum solution.)*

9 Minimizing Number of Gaps with a Bound on Maximum Flow

Now, instead of total flow time, we consider the objective function equal to the *maximum* flow time, $F_{\max} = \max_j (S_j - r_j)$, that we wish to minimize. At the same time, we would also like to minimize the number of gaps. This leads to two optimization problems, by placing a bound on one value and minimizing the other. In this section we consider the problem of minimizing the number of gaps when an upper bound on the flow of each job is given. For this problem, we give $O(n \log n)$-time algorithm.

Formally, we are given an instance $\mathcal{J}$ consisting of n unit jobs with release times and a threshold value f. The objective is to compute a schedule of $\mathcal{J}$ that minimizes the number of gaps among all schedules with maximum flow time bounded by f. (If there is no schedule with maximum flow at most f, the algorithm should report failure.) As before, without loss of generality, we can assume that the jobs are sorted according to their release times, that is $r_1 \le r_2 \le ... \le r_n$. (As we remarked earlier in Section 2, we cannot now assume that all jobs have different release times. In fact, the presence of jobs with equal release times causes the algorithm for the discrete case to be more involved than for the continuous case.)

Continuous case. We start by giving an $O(n \log n)$-time algorithm for the continuous case. Here we are given a collection of n real numbers $r_1, r_2, ..., r_n$, and a number f, and we want to compute a set H of minimum cardinality such that $\min\{h \in H : h \geq r_i\} \leq r_i + f$ for all $i = 1, 2, ..., n$.

We show that this can be solved in time $O(n)$, assuming the release times are sorted, $r_1 \leq r_2 \leq ... \leq r_n$. Indeed, this is very simple, using a greedy algorithm that computes H in a single pass through the input. Specifically, initialize $H = \{r_1 + f\}$. Then in each step choose i to be smallest index for which $r_i > \max(H)$ and add $r_i + f$ to H. A routine inductive argument shows that the computed set H has indeed minimum cardinality. The algorithm is essentially a linear scan through the sorted sequence of release times, so its running time is $O(n)$. With sorting, the time will be $O(n \log n)$.

Discrete case. Next, we want to show that we can achieve the same running time for the discrete variant, where we schedule unit jobs. The greedy single-pass algorithm above does not directly apply because each point in H corresponds now to a (possibly long) block of jobs, affecting the maximum flow value.

The basic idea of our approach is to think about the problem as the gap minimization problem with "virtual" deadlines, where the virtual deadline of each job j is defined by $r_j + f$. We now need to solve the gap minimization problem for jobs with deadlines which, as discussed in the introduction, can be solved in time $O(n^4)$ [6, 7, 8]. However, we can do better than this. The instance with deadlines we created satisfies the "agreeable deadline" property, which means that the ordering of the deadlines is the same as the ordering of release times. For such instances a minimum-gap schedule can be computed in time $O(n \log n)$ (see [2], for example). This will thus give us an $O(n \log n)$-time algorithm for gap minimization with a bound on maximum flow.

In the remainder of this section we present an alternative $O(n \log n)$-time algorithm for this problem, which has the advantage that its running time is actually $O(n)$ if the jobs are already sorted in non-decreasing order of release times. Besides being of its own interest, such an algorithm will be useful in the next section.

Let $\mathcal{J}$ be the given instance of n unit jobs numbered $1, 2, ..., n$, whose release times are ordered in non-decreasing order: $r_1 \leq r_2 \leq ... \leq r_n$. In this ordering the ties are broken arbitrarily. It is easy to see (by a simple exchange argument) that there is an optimal schedule in which all jobs are scheduled in order $1, 2, ..., n$, and we will only consider such schedules from now on.

Algorithm MinGapMaxFlow**.** The algorithm has two stages. In the first stage we produce a tentative schedule Q by greedily scheduling the jobs from left to right: Start with $t = r_1$. We have n steps, and in each step we schedule one job. When step j starts, jobs $1, 2, ..., j-1$ will already be scheduled before the current slot t. We then schedule j as follows: If $r_j \leq t$, schedule j in slot t and let $t = t + 1$; otherwise schedule j in slot r_j and let $t = r_j + 1$. After we schedule all jobs, we check their flow values. If there is a job in Q with flow larger than f, declare failure (meaning that there is no schedule with maximum flow at most f) and stop. Otherwise, continue to the next stage.

We now explain the second stage, in which we convert Q into the final schedule S. This is accomplished by shifting some jobs to the right to reduce the number of gaps, without exceeding the maximum flow restriction. The computation consists of phases numbered $0, 1, ..., g$, where g is the final number of gaps. In each phase we construct one block of S. Let $Q^0 = Q$. In general, let Q^l denote the schedule at the start of phase l. With Q^l we associate a time slot v_{l-1} which represents the last time slot processed in phases $0, 1, ..., l-1$. We (artificially) initialize $v_{-1} = r_1 - 1$. The intuition is that, in Q^l, the jobs from Q in the time segment $[r_1, v_{l-1}]$ will be rearranged into l blocks, while in the time segment $[v_{l-1} + 1, \infty]$ the tentative schedule Q will be still unchanged. Formally, Q^l will satisfy the following invariants (see Figure 7):

(i) All jobs are scheduled in order of their release times.
(ii) The jobs scheduled in interval $[r_1, v_{l-1}]$ are exactly the jobs in $\mathcal{J}$ released in time segment $[r_1, v_{l-1}]$ and scheduled by Q in this time segment.
(iii) The jobs in $[r_1, v_{l-1}]$ are scheduled in l blocks $B_0, B_1, ..., B_{l-1}$, listed from left to right, where $B_h = [u_h, v_h]$ for $h = 0, 1, ..., l-1$. In each block B_h, at least one job has flow time equal f and all other jobs have flow time at most f.
(iv) In interval $[v_{l-1} + 1, \infty]$ schedule Q^l is identical to Q.
(v) Slot $v_{l-1} + 1$ is idle in Q^l and is not a release time of any job.

Let i be the first job in Q^l after v_{l-1}, scheduled at slot Q^l_i. From properties (iv) and (v), and from the way the first stage works, we have $Q^l_i = r_i \geq v_{l-1} + 2$. We start with block B_l initialized as $B_l = [u_l, v_l] = [Q^l_i, Q^l_i]$; that is, it consists only of job i. With B_l we associate its maximum flow time value $F(B_l)$ that is initialized to $Q^l_i - r_i = 0$. Then, in each step of this phase we will either shift B_l to the right or add another job to it. Specifically, we do this. If there is a job j scheduled by Q^l in time slot $v_l + 1$, we add this job to B_l without changing its schedule, which means that we increment v_l, and we update the maximum flow value, $F(B_l) \leftarrow \max\{F(B_l), Q^l_j - r_j\}$. Suppose now that there is no job scheduled in slot $v_l + 1$. If $F(B_l) < f$ then we shift B_l by 1 to the right, that is we increment each u_l, v_l, and $F(B_l)$ by 1. Otherwise, we end the phase. If the last job in B_l is not n, we go to phase $l + 1$. If this job is n, we are done (and $g = l$), and we return $S = Q^l$.

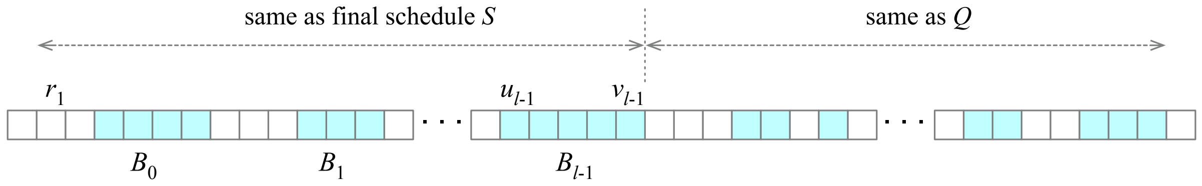


Fig. 7 Illustration of the invariant for Algorithm MinGapMaxFlow, showing the structure of schedule Q^l when phase l is about to start.

We now argue that Algorithm MinGapMaxFlow is correct. To this end, we start with the observation that the tentative schedule Q dominates each other schedule S', in the sense that $Q_j \le S'_j$ for all jobs j in $\mathcal{J}$. (As explained earlier, we consider only schedules, including S', where jobs are scheduled in order $1, 2, ..., n$.) This follows directly from how Q is constructed in the first stage, namely that in Q each job j is scheduled at the first idle slot which is not before r_j and is after the slots of jobs $1, 2, ..., j-1$. This observation implies that schedule Q minimizes the maximum flow. Therefore if Algorithm MinGapMaxFlow proceeds to the second stage, we know that there is a schedule with maximum flow at most f. Further, any such schedule can be obtained from Q by shifting some jobs to the right, preserving the order of jobs.

Consider now S. That the flow of all jobs in S is at most f should also be clear, as when shifting jobs in the second phase of Algorithm MinGapMaxFlow we explicitly ensure that this condition is preserved. Finally, we argue that S minimizes the number of gaps among all schedules with maximum flow at most f. To show this, it is enough to prove that for any two consecutive blocks B_l and B_{l+1} there are two jobs, one in each, that must be separated by a gap in any schedule with maximum flow at most f. To this end, let p be a job in B_l whose flow in S is exactly f, that is $S_p = r_p + f$. (Such p exists, by property (iii).) Let i be the first job in B_{l+1}. Thus job $i-1$ is the last job in B_l and it is scheduled at slot $S_{i-1} = v_l = S_p + i - 1 - p = r_p + f + i - 1 - p$. By property (v), we have $r_i \ge v_l + 2$. Thus

$$r_i - (r_p + f) \;\ge\; v_l + 2 - r_p - f \;=\; i - p + 1.$$

The latest slot when we can schedule job p is $r_p + f$ and the earliest slot when we can schedule job i is r_i. The time segment $[r_p + f, r_i]$ has $r_i - (r_p + f) + 1$ slots, which is strictly greater (by the above inequality) than the number of jobs $i - p + 1$ between p and i (inclusive) that we need to schedule in this segment, so there has to be an idle slot between p and i, as claimed.

We now claim that Algorithm MinGapMaxFlow can be implemented in time $O(n)$. The first stage clearly runs in time $O(n)$, so we focus on the second stage. In our implementation, for each block B_l we have a list of jobs scheduled in this block, in order of release times. (However, for the jobs in B_l we do *not* keep track of which slot they are scheduled in during the second stage. Updating these values after each shift would be too time consuming.) Instead of repeatedly shifting B_l, we compute the smallest shift value δ such that, after shifting B_l by δ, either $F(B_l)$ will become equal f or there will be a job scheduled by Q^l right after B_l. Specifically, if i is the first job scheduled after B_l, at time Q^l_i, then the shift value is $\delta = \min(Q^l_i - v_l - 1, f - F(B_l))$. Thus all three values of u_l, v_l and $F(B_l)$ are increased by δ. After this computation is complete, the slot S_j of each job j can be computed by adding its index within its block B_l to this block's start time u_l. With these modifications, the running time of the second stage will be $O(n)$.

Summarizing this section, we obtain the following theorem:

Theorem 7 *For any instance $\mathcal{J}$ and a flow bound f, Algorithm* MinGapMaxFlow *in time $O(n \log n)$ computes a schedule of $\mathcal{J}$ that minimizes the number of gaps among all schedules with maximum flow at most f. If the release times are already sorted, the running time of Algorithm* MinGapMaxFlow *is $O(n)$.*

10 Minimizing Maximum Flow with a Budget for Gaps

We now consider an alternative variant of the tradeoff between minimizing the maximum flow and the number of gaps. This time, for a given collection of n unit jobs with release times $r_1, r_2, ..., r_n$ and a budget γ, we want to compute a schedule that minimizes the maximum flow time value $F_{\max}$ and has at most γ gaps. (Recall that $F_{\max} = \max_j F_j$, where F_j is the flow time of job j, that is $F_j = S_j - r_j$.) We can again assume that $r_1 \leq r_2 \leq ... \leq r_n$ and restrict our attention to schedules where jobs are scheduled in order $1, 2, ..., n$.

Continuous case. In the continuous case, $r_1, r_2, ..., r_n$ are points on the real line, and we want to compute a set H of at most k points that minimizes $F_{\max}(H) = \max_i \min_{x \in H, x \geq r_i}(x - r_i)$. This is a special case of the k-center problem, when the underlying space is the directed line, which can be solved in time $O(n \log^* n)$ if the points are already sorted [13]. (The undirected version of this problem has been extensively studied since early 1980's, even for the more general cases of trees [11,26,20,19,22], culminating in an $O(n)$-time algorithm [20].) As we do not assume the inputs to be sorted, a simpler $O(n \log n)$-time algorithm that we outline below will be sufficient for our purpose. The ingredients for this algorithm are present in various forms in the above cited work on the k-center problem, but we include it here for the sake of completeness, and as a stepping stone to our algorithm for the discrete case.

Similar to our algorithm in Section 8, the high-level idea is based on *parametric search* (see [19,20,22], for example). It involves binary search for the optimal value f^* of $F_{\max}(H)$, where at each step of the binary search we use the algorithm from the previous section as an oracle.

For binary search, however, we need a small set of candidate values for f^*. If H is an optimal solution, then, without loss of generality, we can assume that H contains only release times, since any other point in H can be shifted left until it reaches a release time. Thus we only need to consider the set Φ of all values of the form $r_j - r_i$ for $j > i$. (We will tacitly assume that $F_{\max}(H) > 0$, because verifying whether $F_{\max}(H) = 0$ is easy: just check if the number of different r_i's is at most k.) Since $|\Phi| = O(n^2)$ and we need to sort Φ before doing binary search, we would obtain an $O(n^2 \log n)$-time algorithm.

Fortunately, we do not need to construct Φ explicitly. Observe that the elements of Φ can be thought of as forming an implicit $X + Y$ matrix with sorted rows and columns, where X is the vector of release times and $Y = -X$. We can thus use the $O(n)$-time selection algorithm for $X + Y$ matrices [21,27] to speed up computation. Specifically, at each step we will have two indices p, q, with $1 \leq p \leq q \leq n(n-1)/2$, such that the optimal value of f^* is between the pth and qth smallest values in Φ, inclusive. If $p = q$, we are done, so assume that $p < q$. We let $l = \lfloor (p+q)/2 \rfloor$ and we use the algorithm from [21,27] to find the lth smallest element in Φ, say f. We now determine whether $f^* \leq f$ by applying the $O(n)$ algorithm from the previous section to answer the query "is there a set H with $|H| \leq k$ and $F_{\max}(H) \leq f$?". If the answer is "yes", we let $q = l$, otherwise we let $p = l + 1$. This will give us an algorithm with running time $O(n \log n)$.

Discrete case. We now show that we can solve the scheduling variant in time $O(n \log n)$ as well. The solution is similar to the one for the continuous case, with two modifications. The first modification concerns the set Φ of candidate values for the maximum flow. We show that Φ can be still expressed as an $X + Y$ set, for some sets X and Y of small cardinality. The second modification involves using Algorithm MinGapMaxFlow to answer decision queries in the binary search, instead of the algorithm for the continuous model.

Without loss of generality, we can restrict our attention to schedules S that have the following structure:

(i) Jobs in S appear in order $1, 2, ..., n$ from left to right. (This assumption was already justified earlier).
(ii) Any block in S contains a job scheduled at its release time. (Otherwise we can shift this block to the left.)
(iii) If a job i is scheduled by S in some block B, then r_i is either in B or in the gap preceding B. (Otherwise, by the ordering of release times and (i), we can assume that i is the first job in B. We could then move i to the end of the previous block, and repeat this process.)
(iv) Any two jobs released at the same time are scheduled in the same block. (This follows from (iii).)

To apply search in $X + Y$ matrices, we would like to restrict X and Y to have size $O(n)$. Assumption (ii) gives us immediately that there is an optimal schedule where each job is scheduled in a slot in $R + [-n, n]$ (see Section 3), but this set has quadratic size, so it's too large for our purpose.

To construct smaller sets X, Y, we reason as follows. Consider some optimal schedule S. Choose i to be a job with maximum flow time in S, and suppose that i is scheduled by S in some block B. By (ii), B

has a job j scheduled at time r_j. Then the flow time of i can be written as

$$\begin{aligned} F_i \;=\; S_i - r_i \;&=\; (r_j + i - j) - r_i \\ &=\; (r_j - j) - (r_i - i). \end{aligned} \tag{7}$$

This equation holds no matter whether $j \leq i$ or $j > i$. Now, take X to be the set of all values $r_j - j$ for $j = 1, 2, ..., n$ and $Y = -X$. We can sort X and Y in time $O(n \log n)$. By (7), we only need to search for the optimal flow value in $\varPhi = X + Y$. Analogously to the continuous case, we perform binary search in $\varPhi$, using the $O(n)$-time algorithm from [21,27] for selection in $X + Y$ matrices and Algorithm MinGapMaxFlow as the decision oracle at each step, and since the release times can be pre-sorted, each invocation of this oracle will take time $O(n)$. Thus the running time will be $O(n \log n)$.

The complete algorithm in pseudo-code is given below. In the algorithm we assume that $n \geq 3$ and $0 \leq \gamma \leq n - 2$, as for $\gamma \geq n - 2$ we have $F_{\max} = 0$. In this pseudo-code, MatrixSelect(X, Y, l) is a call to an $O(n)$-time algorithm in [21,27] that finds the lth smallest value in the (implicit) matrix $X + Y$.

Pseudocode 2 Algorithm MinMaxflowGap

```
1: Sort release times so that $r_1 \leq r_2 \leq ... \leq r_n$
2: $X \leftarrow \{r_j - j : j \in \{1, 2, ..., n\}\}$
3: $Y \leftarrow -X$
4: $p \leftarrow 1$ and $q \leftarrow n$
5: while $p < q$ do
6:     $l \leftarrow \lfloor (p + q)/2 \rfloor$
7:     $f \leftarrow$ MatrixSelect$(X, Y, l)$
8:     if MinGapMaxFlow$(f) \leq \gamma$ then
9:         $q \leftarrow l$
10:    else
11:        $p \leftarrow l$
12: return MatrixSelect$(X, Y, p)$
```

Summarizing this section, we obtain the following theorem:

Theorem 8 *For any instance $\mathcal{J}$ and a gap budget γ, Algorithm* MinMaxflowGap *in time $O(n \log n)$ computes a schedule of $\mathcal{J}$ that minimizes the maximum flow value among all schedules with at most γ gaps.*

11 Final Comments

We studied several scheduling problems for unit-length jobs where the gap structure of the computed schedule is taken into consideration. For all problems we considered we provided polynomial-time algorithms, with running times ranging from $O(n \log n)$ to $O(n^7)$.

Many open problems remain. The most intriguing question is whether the running time for minimizing the number of gaps for unit jobs can be improved to below $O(n^4)$. As discussed in Section 1, this problem is closely related to energy minimization in the power-down model, and faster algorithms for this problem would likely also apply to computing minimum-energy schedules. Speeding up the algorithms in Sections 3, 4, 5, and 6 would also be of considerable interest.

There is a number of other variants of gap scheduling, even for unit jobs, that we have not addressed in our paper. Here are some examples:

– The problem of maximizing the minimum gap. This is somewhat similar to the problem we studied in Section 6, but we are not sure whether our method can be extended to this model. (We remark here that, according to our definition, the minimum gap size cannot be 0. For the purpose of maximizing the minimum gap, one can also consider an alternative model where “gaps” of size 0 are taken into account.)
– The tradeoff between throughput and gap size. Here, one can consider either the lower or upper bound on the gap size.

– The tradeoff between flow time (total or maximum) and gap size. For example, one may wish to minimize the total flow time with all gaps not exceeding a specified threshold.
– The problems of maximizing the number of gaps or minimizing the maximum gap, studied in Sections 5 and 6, were motivated by applications where the schedule for high-priority jobs needs to contain gaps where low-priority jobs can be inserted. A more accurate model for such applications would be to require that each block is of length at most b, for some given parameter b. Testing feasibility, with this requirement, can be achieved in high-degree polynomial time by extending the techniques from [6,7,8] and Sections 3 and 6, but it would be interesting to see whether more efficient solutions exist.

A natural extension of our work would be to study variants of gap scheduling for jobs of arbitrary length, for models with preemptive or non-preemptive jobs. The algorithm for minimizing the number of gaps, for example, can be extended to jobs of arbitrary length [7,8] if preemptions are allowed, although its running time increases from $O(n^4)$ to $O(n^5)$.

Another related direction of research would be to focus on the sizes of blocks in the schedule, or even consider them together with gap sizes. For example, schedules with low density (maximum ratio of the number of jobs in an interval to its length) would be helpful in controlling the processor's temperature during the execution [12], as they include idle time slots that allow the processor to cool down between executing consecutive blocks.

Acknowledgements We would like to thank Nael Abu-Ghazaleh for pointing out the connection between gap scheduling and wireless channel access scheduling for high- and low-priority traffic streams [28].

References

1. Aggarwal, A., Klawe, M., Moran, S., Shor, P., Wilber, R.: Geometric applications of a matrix-searching algorithm. Algorithmica **2**, 195–208 (1987). DOI 10.1007/BF01840359
2. Angel, E., Bampis, E., Chau, V.: Low complexity scheduling algorithm minimizing the energy for tasks with agreeable deadlines. In: 10th Latin American Theoretical Informatics Symposium (LATIN'12), pp. 13–24 (2012)
3. Angel, E., Bampis, E., Chau, V.: Low complexity scheduling algorithms minimizing the energy for tasks with agreeable deadlines. Discrete Applied Mathematics **175**, 1 – 10 (2014)
4. Auletta, V., Parente, D., Persiano, G.: Placing resources on a growing line. J. Algorithms **26**(1), 87–100 (1998)
5. Bampis, E.: Algorithmic issues in energy-efficient computation. In: Discrete Optimization and Operations Research, pp. 3–14. Springer International Publishing (2016)
6. Baptiste, P.: Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'06), pp. 364–367 (2006)
7. Baptiste, P., Chrobak, M., Dürr, C.: Polynomial time algorithms for minimum energy scheduling. In: Proceedings of the 15th Annual European Symposium on Algorithms (ESA'07), pp. 136–150 (2007)
8. Baptiste, P., Chrobak, M., Dürr, C.: Polynomial-time algorithms for minimum energy scheduling. ACM Trans. Algorithms **8**(3), 26:1–26:29 (2012)
9. Bein, W.W., Golin, M.J., Larmore, L.L., Zhang, Y.: The Knuth-Yao quadrangle-inequality speedup is a consequence of total monotonicity. ACM Trans. Algorithms **6**(1), 17:1–17:22 (2009)
10. Burkard, R.E., Klinz, B., Rudolf, R.: Perspectives of Monge properties in optimization. Discrete Applied Mathematics **70**(2), 95 – 161 (1996)
11. Chen, D.Z., Li, J., Wang, H.: Efficient algorithms for the one-dimensional k-center problem. Theoretical Computer Science **592**, 135 – 142 (2015)
12. Chrobak, M., Dürr, C., Hurand, M., Robert, J.: Algorithms for temperature-aware task scheduling in microprocessor systems. SUSCOM **1**(3), 241–247 (2011)
13. Chrobak, M., Eppstein, D., Italiano, G.F., Yung, M.: Efficient sequential and parallel algorithms for computing recovery points in trees and paths. In: 2nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'91), p. 158167 (1991)
14. Chrobak, M., Feige, U., Hajiaghayi, M.T., Khanna, S., Li, F., Naor, S.: A greedy approximation algorithm for minimum-gap scheduling. J. Sched. **20**(3), 279–292 (2017)
15. Chrobak, M., Feige, U., Taghi Hajiaghayi, M., Khanna, S., Li, F., Naor, S.: A greedy approximation algorithm for minimum-gap scheduling. In: Proc. 8th International Conference on Algorithms and Complexity (CIAC'13), pp. 97–109 (2013)
16. Chrobak, M., Larmore, L.L., Rytter, W.: The k-median problem for directed trees. In: Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001 Marianske Lazne, Czech Republic, August 27-31, 2001, Proceedings, pp. 260–271 (2001)
17. Damaschke, P.: Refined algorithms for hitting many intervals. Information Processing Letters **118**, 117 – 122 (2017)
18. Demaine, E.D., Ghodsi, M., Hajiaghayi, M.T., Sayedi-Roshkhar, A.S., Zadimoghaddam, M.: Scheduling to minimize gaps and power consumption. In: Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'07), pp. 46–54 (2007)
19. Frederickson, G.N.: Optimal algorithms for tree partitioning. In: 2nd Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms (SODA'91), pp. 168–177 (1991)
20. Frederickson, G.N.: Parametric search and locating supply centers in trees. In: Workshop on Algorithms and Data Structures (WADS'91), pp. 299–319 (1991)

21. Frederickson, G.N., Johnson, D.B.: The complexity of selection and ranking in X+Y and matrices with sorted columns. J. Comput. Syst. Sci. **24**(2), 197–208 (1982)
22. Frederickson, G.N., Zhou, S.: Optimal parametric search for path and tree partitioning. CoRR **abs/1711.00599** (2017). URL `http://arxiv.org/abs/1711.00599`
23. Hassin, R., Tamir, A.: Improved complexity bounds for location problems on the real line. Oper. Res. Lett. **10**(7), 395–402 (1991)
24. Irani, S., Pruhs, K.R.: Algorithmic problems in power management. SIGACT News **36**(2), 63–76 (2005). DOI 10.1145/1067309.1067324. URL `http://doi.acm.org/10.1145/1067309.1067324`
25. Jansen, K., Scheffler, P., Woeginger, G.: The disjoint cliques problem. RAIRO Recherche Opérationnelle **31**, 45–66 (1997)
26. Megiddo, N., Tamir, A.: New results on the complexity of p-center problems. SIAM J. Comput. **12**, 751–758 (1983)
27. Mirzaian, A., Arjomandi, E.: Selection in X+Y and matrices with sorted rows and columns. Inf. Process. Lett. **20**(1), 13–17 (1985)
28. Wikipedia: Point coordination function. `http://en.wikipedia.org/wiki/Point_coordination_function`
29. Woeginger, G.J.: Monge strikes again: optimal placement of web proxies in the Internet. Oper. Res. Lett. **27**(3), 93–96 (2000)