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

A Greedy Approximation Algorithm for Minimum-Gap Scheduling

Marek Chrobak · Uriel Feige · Mohammad Taghi Hajiaghayi · Sanjeev Khanna · Fei Li · Seffi Naor

Abstract We consider scheduling of unit-length jobs with release times and deadlines, where the objective is to minimize the number of gaps in the schedule. Polynomial-time algorithms for this problem are known, yet they are rather inefficient, with the best algorithm running in time $O(n^4)$ and requiring $O(n^3)$ memory. We present a greedy algorithm that approximates the optimum solution within a factor of 2 and show that our analysis is tight. Our algorithm runs in time $O(n^2 \log n)$ and needs only $O(n)$ memory. In fact, the running time is $O(n(g^* + 1) \log n)$, where g^* is the minimum number of gaps.

1 Introduction

Research on approximation algorithms up to date has focussed mostly on optimization problems that are $\mathbb{NP}$-hard. From the purely practical point of view, however, there is little difference between exponential and high-degree polynomial running times. Memory requirements could also be a critical factor, because high-degree polynomial algorithms typically involve computing entries in a high-dimensional table via dynamic programming. An algorithm requiring $O(n^4)$ or more memory would be impractical even for relatively modest values of n because when the main memory fills up, disk paging will considerably slow down the (already slow) execution. With this in mind, for such problems it is natural to ask whether there are faster algorithms that use little memory and produce near-optimal solutions. This direction of research is not entirely new. For example, in recent years, approximate streaming algorithms have been extensively studied for problems that are polynomially solvable, but where massive amounts of data need to be processed in nearly linear time.

In this paper we focus on the problem of minimum-gap job scheduling, where the objective is to schedule a collection of unit-length jobs with given release times and deadlines, in such a way that the number of gaps (idle intervals) in the schedule is minimized. This scheduling paradigm was originally proposed, in a somewhat more general form, by Irani and Pruhs [7]. The first polynomial-time algorithm for this problem, with running time $O(n^7)$, was given by Baptiste [3]. This was subsequently improved by Baptiste *et al.* [4], who gave an algorithm with running time $O(n^4)$ and space complexity $O(n^3)$. All these algorithms are based on dynamic programming.

Marek Chrobak
Department of Computer Science, University of California, Riverside, CA 92521, USA.

Uriel Feige
Department of Computer Science and Applied Mathematics, the Weizmann Institute, Rehovot 76100, Israel.

Mohammad Taghi Hajiaghayi
Computer Science Department, University of Maryland, College Park, MD 20742, USA. The author is also with AT&T Labs–Research.

Sanjeev Khanna
Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, USA.

Fei Li
Department of Computer Science, George Mason University, Fairfax, VA 22030 USA.

Seffi Naor
Computer Science Department, Technion, Haifa 32000, Israel.

Our results. The main contribution of this paper is an efficient algorithm for minimum-gap scheduling of unit-length jobs that computes a near-optimal solution. The algorithm runs in time $O(n^2 \log n)$, uses only $O(n)$ space, and it approximates the optimum within a factor of 2. More precisely, if the optimal schedule has g^* gaps, our algorithm will find a schedule with at most $2g^* - 1$ gaps (assuming $g^* \geq 1$). The running time can in fact be expressed as $O(n(g^* + 1) \log n)$; thus, since $g^* \leq n$, the algorithm is considerably faster if the optimum is small. (To be fair, so is the algorithm in [3], whose running time can be reduced to $O(n^3(g^* + 1))$.) The algorithm itself is a simple greedy algorithm: it adds gaps one by one, at each step adding the longest gap for which there exists a feasible schedule. The analysis of the approximation ratio and an efficient implementation, however, require good insights into the gap structure of feasible schedules.

Related work. Prior to the paper by Baptiste [3], Chretienne [5] studied various versions of scheduling where only schedules without gaps are allowed. The algorithm in [3] can be extended to handle jobs of arbitrary length, assuming that preemptions are allowed, although then the time complexity increases to $O(n^5)$. Working in another direction, Demaine *et al.* [6] showed that for p processors the gap minimization problem can be solved in time $O(n^7 p^5)$ if jobs have unit lengths.

The generalization of minimum-gap scheduling introduced by Irani and Pruhs [7], that we alluded to earlier, is concerned with computing minimum-energy schedules in the power-down model. In their model, the processor uses energy at constant rate when processing jobs and it can be turned off during the idle periods with some additive energy penalty representing an overhead for turning the power back on. If this penalty is at most 1 then the problem is equivalent to minimizing the number of gaps. The algorithms from [3] can be extended to this power-down model without increasing their running times. Note that our approximation ratio is even better if we express it in terms of the energy function: since both the optimum and the algorithm pay n for job processing, the ratio can be bounded by $1 + g^*/(n+g^*)$. Thus the ratio is at most 1.5, and it is only $1 + o(1)$ if $g^* = o(n)$.

In an even more general energy-consumption model, the processor may also have the speed-scaling capability, in addition to the power-down mechanism. The complexity of this problem had been open for quite some time, until, just recently, Albers and Antoniadis [2] showed it to be $\mathbb{NP}$-hard. The reader is referred to that paper, as well as the surveys in [1, 7], for more information on the models involving speed-scaling.

2 Preliminaries

Basic definitions and properties. We assume that the time axis is partitioned into unit-length time slots numbered $0, 1, \ldots$. By $\mathcal{J}$ we will denote the instance, consisting of a set of unit-length jobs numbered $1, 2, \ldots, n$, each job j with a given release time r_j and deadline d_j, both integers. Without loss of generality, $r_j \leq d_j$ for each j. By $r_{\min} = \min_j r_j$ and $d_{\max} = \max_j d_j$ we denote the earliest release time and the latest deadline, respectively.

Throughout the paper, by a *(feasible) schedule* S of $\mathcal{J}$ we mean a function that assigns jobs to time slots such that each job j is assigned to a slot $t \in [r_j, d_j]$ and different jobs are assigned to different slots. If j is assigned by S to a slot t then we say that j is *scheduled* in S at time t. If S schedules a job at time t then we say that slot t is *busy*; otherwise we call it *idle*. The *support* of a schedule S, denoted $\mathsf{Supp}(S)$, is the set of all busy slots in S. An inclusion-maximal interval consisting of busy slots is called a *block*. A block starting at $r_{\min}$ or ending at $d_{\max}$ is called *exterior* and all other blocks are called *interior*. Any inclusion-maximal interval of idle slots between $r_{\min}$ and $d_{\max}$ is called a *gap*.

Note that, with the above definitions, if there are idle slots between $r_{\min}$ and the first job then they also form a gap and there is no left exterior block, and a similar property holds for the idle slots right before $d_{\max}$. To avoid this, we will assume that jobs 1 and n are tight jobs with $r_1 = d_1 = r_{\min}$ and $r_n = d_n = d_{\max}$, so these jobs must be scheduled at $r_{\min}$ and $d_{\max}$, respectively, and each schedule must have both exterior blocks. We can modify any instance to have this property by adding two such jobs to it (one right before the minimum release time and the other right after the maximum deadline), without changing the number of gaps in the optimum solution.

We call an instance $\mathcal{J}$ *feasible* if it has a schedule. We are only interested in feasible instances, so in the paper we will be assuming that $\mathcal{J}$ is feasible. Checking feasibility is very simple. One way to do that is to run the greedy earliest-deadline-first algorithm (EDF): process the time slots from left to right and at each step schedule the earliest-deadline job that has already been released but not yet scheduled. It is easy to see that $\mathcal{J}$ is feasible if and only if no job misses its deadline in EDF. Another way is to use the theory of bipartite matchings: a schedule can be thought of as a matching between jobs and time slots. We can then use Hall's theorem to characterize feasible instances. In fact, in this application of Hall's theorem it is sufficient to consider only time intervals instead of arbitrary sets of time slots. This implies that $\mathcal{J}$ is feasible if and only if for any time interval $[t,u]$ we have

$$|Load(t,u)| \leq u - t + 1, \tag{1}$$

where $Load(t,u) = \{j : t \leq r_j \leq d_j \leq u\}$ is the set of jobs that must be scheduled in $[t,u]$.

Without loss of generality, we can assume that all release times are distinct and that all deadlines are distinct. Indeed, if $r_i = r_j$ and $d_i \leq d_j$ for two jobs i, j, since these jobs cannot both be scheduled at time r_i, we may as well increase by 1 the release time of j. A similar argument applies to deadlines. This modification can be obtained in time $O(n \log n)$ and it does not affect the optimum number of gaps of $\mathcal{J}$.

Scheduling with forbidden slots. It will be convenient to consider a more general version of the above problem, where some slots in $[r_{\min}, d_{\max}]$ are designated as *forbidden*, namely no job is allowed to be scheduled in them. We will typically use letters X, Y or Z to denote the set of forbidden slots. A schedule of $\mathcal{J}$ that does not schedule any jobs in a set Z of forbidden slots is said to *obey* Z. A set Z of forbidden slots will be called *viable* if there is a schedule that obeys Z.

Formally, we can think of a schedule with forbidden slots as a pair (S, Y), where Y is a set of forbidden slots and S is a schedule that obeys Y. In the rest of the paper, however, we will avoid this formalism as the set Y of forbidden slots associated with S will be always understood from context.

All definitions and properties above extend naturally to scheduling with forbidden slots. Now, for any schedule S we have three types of slots: *busy*, *idle* and *forbidden*. The definition of gaps does not change. The *support* is now defined as the set of slots that are either busy or forbidden, and a block is a maximal interval consisting of slots in the support. The support uniquely determines our objective function (the number of gaps), and thus we will be mainly interested in the support of the schedules we consider, rather than in the exact mapping from jobs to slots.

Inequality (1) generalizes naturally to scheduling with forbidden slots, as follows: a forbidden set Z is viable if and only if

$$|Load(t,u)| \leq |[t,u] - Z|, \tag{2}$$

holds for all $t \leq u$, where $[t,u] - Z$ is the set of non-forbidden slots between t and u (inclusive). This characterization is, however, too general for the purpose of our analysis, so in the next section we establish additional properties of viable forbidden regions.

3 Transfer Paths

Let Q be a feasible schedule. Consider a sequence $\boldsymbol{t} = (t_0, t_1, ..., t_k)$ of different time slots such that $t_0, ..., t_{k-1}$ are busy and t_k is idle in Q. Let j_a be the job scheduled by Q in slot t_a, for $a = 0, ..., k-1$. We will say that $\boldsymbol{t}$ is a *transfer path for* Q (or simply a *transfer path* if Q is understood from context) if $t_{a+1} \in [r_{j_a}, d_{j_a}]$ for all $a = 0, ..., k-1$. Given such a transfer path $\boldsymbol{t}$, the *shift operation along* $\boldsymbol{t}$ moves each j_a from slot t_a to slot t_{a+1}. From the definition, this shift operation produces another feasible schedule. For technical reasons we allow $k = 0$ in the definition of transfer paths, in which case t_0 itself is idle, $\boldsymbol{t} = (t_0)$, and no jobs will be moved by the shift.

Note that if $Z = \{t_0\}$ is a forbidden set that consists of only one slot t_0, then the shift operation will convert Q into a new schedule that obeys Z. To generalize this idea to arbitrary forbidden sets, we prove the lemma below.

Lemma 1 *Let Q be a feasible schedule. Then a set Z of forbidden slots is viable if and only if there are $|Z|$ disjoint transfer paths for Q starting in Z.*

Proof (⇐) This implication is simple: For each $x \in Z$ perform the shift operation along the path starting in x, as defined before the lemma. The resulting schedule Q' is feasible and it does not schedule any jobs in Z, so Z is viable.

(⇒) Let S be an arbitrary schedule that obeys Z. Consider a bipartite graph $\mathcal{G}$ whose vertex set consists of jobs and time slots, with job j connected to slot t if $t \in [r_j, d_j]$. Then both Q and S can be thought of as perfect matchings in $\mathcal{G}$, in the sense that all jobs are matched to some slots. In S, all jobs will be matched to non-forbidden slots. By the theory of bipartite matchings, there is a set of disjoint alternating paths in $\mathcal{G}$ (paths that alternate between the edges of Q and S) connecting slots that are not matched in S to those that are not matched in Q. Slots that are not matched in both schedules form trivial paths, that consist of just one vertex.

Consider a slot x that is not matched in S. In other words, x is either idle or forbidden in schedule S. The alternating path in $\mathcal{G}$ starting at x, expressed as a list of vertices, has the form: $x = t_0 - j_0 - t_1 - j_1 - ... - j_{k-1} - t_k$, where, for each $a = 0, ..., k-1$, j_a is the job scheduled at t_a in Q and at t_{a+1} in S, and t_k is idle in Q. Therefore this path defines uniquely a transfer path $\boldsymbol{t} = (t_0, t_1, ..., t_k)$ starting at $t_0 = x$. Note that if x is idle in Q then this path is trivial – it ends at x. This way we obtain $|Z|$ disjoint transfer paths for all slots $x \in Z$, as claimed.

Any set $\mathcal{P}$ of transfer paths that satisfies Lemma 1 will be called a *Z-transfer multi-path for Q*. We will omit the attributes Z and/or Q if they are understood from context. By performing the shifts along the paths in $\mathcal{P}$ we can convert Q into a new schedule S that obeys Z. For brevity, we will write

$$S = \mathsf{Shift}(Q, \mathcal{P}).$$

Next, we would like to strengthen the → implication in Lemma 1 by showing that Q has a Z-transfer multi-path with a regular structure, where each path proceeds in one direction (either left or right) and where different paths do not "cross" (in the sense formalized below). As it turns out, a general claim like this is not true – sometimes these paths may be unique but not possess all these properties. However, we show that in such a case the original schedule can be replaced by a schedule with the same support and with transfer paths satisfying the desired properties.

To formalize the above intuition we need a few more definitions. If $\boldsymbol{t} = (t_0, ..., t_k)$ is a transfer path then any pair of slots (t_a, t_{a+1}) in $\boldsymbol{t}$ is called a *hop* of $\boldsymbol{t}$. The length of hop (t_a, t_{a+1}) is $|t_a - t_{a+1}|$. The *hop length* of $\boldsymbol{t}$ is the sum of the lengths of its hops, that is $\sum_{a=0}^{k-1} |t_a - t_{a+1}|$.

A hop (t_a, t_{a+1}) of $\boldsymbol{t}$ is *leftward* if $t_a > t_{a+1}$ and *rightward* otherwise. We say that $\boldsymbol{t}$ is *leftward* (resp.*rightward*) if all its hops are leftward (resp. *rightward*). A path that is either leftward or rightward will be called *straight*. Trivial transfer paths are considered both leftward and rightward.

For two non-trivial disjoint transfer paths $\boldsymbol{t} = (t_0, ..., t_k)$ and $\boldsymbol{u} = (u_0, ..., u_l)$, we say that $\boldsymbol{t}$ and $\boldsymbol{u}$ *cross* if there are indices a, b for which one of the following four-conditions holds:

$$\begin{aligned} &t_a < u_{b+1} < t_{a+1} < u_b, \text{ or} \\ &u_b < t_{a+1} < u_{b+1} < t_a, \text{ or} \\ &t_{a+1} < u_b < t_a < u_{b+1}, \text{ or} \\ &u_{b+1} < t_a < u_b < t_{a+1}. \end{aligned}$$

If such a, b exist, we will also refer to the pair of hops (t_a, t_{a+1}) and (u_b, u_{b+1}) as a *crossing*. One can think of the first two cases as "inward" crossings, with the two hops directed towards each other, and the last two cases as "outward" crossings, with the two hops directed away from each other.

Suppose that paths $\boldsymbol{t}$ and $\boldsymbol{u}$ are disjoint, non-trivial, and straight. It is easy to verify that if $\boldsymbol{t}$ and $\boldsymbol{u}$ also satisfy either $t_0 < u_l < t_k < u_0$ or $u_l < t_0 < u_0 < t_k$, then these paths must cross. Interestingly, this claim is not true if we drop the assumption that $\boldsymbol{t}$ and $\boldsymbol{u}$ are straight.

Lemma 2 *As before, let Q be a feasible schedule and let Z be a viable forbidden set. Then there is a schedule Q' such that*

(i) *$\mathit{Supp}(Q') = \mathit{Supp}(Q)$, and*
(ii) *Q' has a Z-transfer multi-path $\mathcal{P}$ in which all paths are straight and do not cross.*

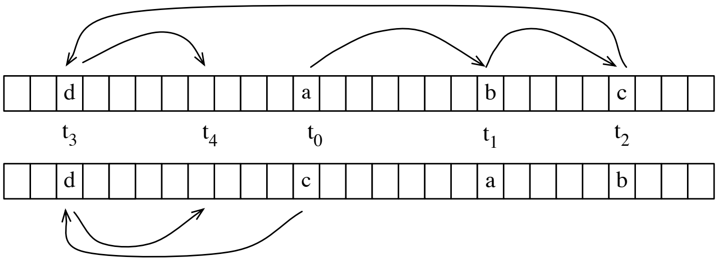

Fig. 1 Converting a path into a straight path in the proof of Lemma 2. Path $\boldsymbol{t} = (t_0, t_1, t_2, t_3, t_4)$ and its modified version (t_0, t_3, t_4) are marked with arrows. Jobs a, b, c are cyclically shifted.

Proof Let $\mathcal{R}$ be a Z-transfer multi-path for Q. Lemma 1 states that $\mathcal{R}$ exists. The idea of the proof is to define a number of operations that alter Q and the paths in $\mathcal{R}$, and to argue that by applying these operations repeatedly we eventually must obtain a schedule Q' and a set of transfer paths $\mathcal{P}$ that satisfy the above conditions. Define the *total hop length* of $\mathcal{R}$ to be the sum of hop lengths of all paths in $\mathcal{R}$.

Suppose that $\mathcal{R}$ has a path $\boldsymbol{t} = (t_0, ..., t_k)$ that is not straight. Without loss of generality, $t_0 < t_1$. Let a be the index such that $t_0 < t_1 < ... < t_a$ and $t_{a+1} < t_a$. We have two cases. If $t_{a+1} > t_0$, let b be the index for which $t_b < t_{a+1} < t_{b+1}$. In this case we replace $\boldsymbol{t}$ by the transfer path $(t_0, ..., t_b, t_{a+1}, t_{a+2}, ..., t_k)$. The other case is when $t_{a+1} < t_0$. In this case we do two modifications. First, in Q we shift $(t_0, ..., t_{a-1})$, that is for each $c = 0, ..., a-1$ we reschedule the job from t_c in t_{c+1}. Then we reschedule the job from t_a in t_0. This does not change the support of the schedule. Next, we replace $\boldsymbol{t}$ in $\mathcal{R}$ by the path $(t_0, t_{a+1}, ..., t_k)$. (See Figure 1.) Note that in both cases the new path starts at t_0, ends at t_k, and satisfies the definition of transfer paths. Further, this modification reduces the total hop length of $\mathcal{R}$.

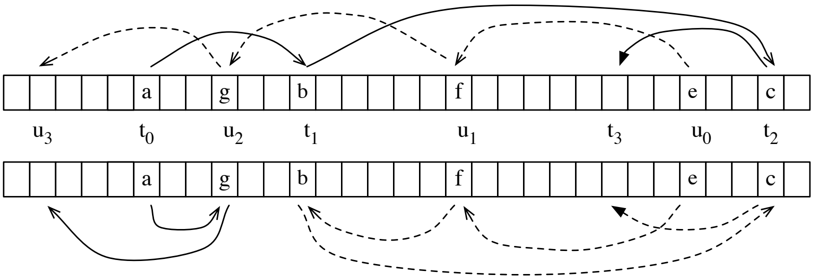

Fig. 2 Removing path crossings in the proof of Lemma 2. Path $\boldsymbol{t} = (t_0, t_1, t_2, t_3)$ and its modified version (t_0, u_2, u_3) are marked with solid arrows; path $\boldsymbol{u} = (u_0, u_1, u_2, u_3)$ and its modified version $(u_0, u_1, t_1, t_2, t_3)$ are marked with dashed arrows. The crossing that is being removed is between hops (t_0, t_1) and (u_1, u_2).

Consider now two paths in $\mathcal{R}$ that cross, $\boldsymbol{t} = (t_0, ..., t_k)$ and $\boldsymbol{u} = (u_0, ..., u_l)$. Without loss of generality, we can assume that the hops that cross are (t_a, t_{a+1}) and (u_b, u_{b+1}), where $t_a < t_{a+1}$. We have two cases, depending on the type of crossing. If $t_a < u_{b+1} < t_{a+1} < u_b$ (that is, an inward crossing), then we replace $\boldsymbol{t}$ and $\boldsymbol{u}$ in $\mathcal{R}$ by paths $(t_0, ..., t_a, u_{b+1}, ..., u_l)$ and $(u_0, ..., u_b, t_{a+1}, ..., t_k)$. (See Figure 2 for illustration.) It is easy to check that these two paths are indeed correct transfer paths starting at t_0 and u_0 and ending at u_l and t_k, respectively. The second case is that of an outward crossing, when $u_{b+1} < t_a < u_b < t_{a+1}$. In this case we also need to modify the schedule by swapping the jobs in slots t_a and u_b. Then we replace $\boldsymbol{t}$ and $\boldsymbol{u}$ in $\mathcal{R}$ by $(t_0, ..., t_a, u_{b+1}, ..., u_l)$ and $(u_0, ..., u_b, t_{a+1}, ..., t_k)$. This modification reduces the total hop length of $\mathcal{R}$.

Each of the operations above reduces the total hop length of $\mathcal{R}$; thus, after a sufficient number of repetitions we must obtain a set $\mathcal{R}$ of transfer paths to which none of the above operations will apply. Also, these operations do not change the support of the schedule. Let Q' be the schedule Q after the steps above and let $\mathcal{P}$ be the final set $\mathcal{R}$ of the transfer paths. Then Q' and $\mathcal{P}$ satisfy the properties in the lemma, completing the proof.

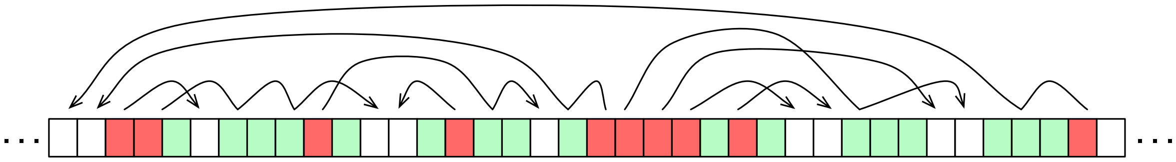

Fig. 3 An illustration of the structure of transfer paths that satisfy Lemma 2. Busy slots are lightly shaded and forbidden slots are dark shaded.

Note that even if $\mathcal{P}$ satisfies Lemma 2, it is still possible that opposite-oriented paths traverse over the same slots. If this happens, however, then one of the paths must be completely "covered" by a hop of the other path, as summarized in the corollary below. (See also Figure 3.)

Corollary 1 *Assume that $\mathcal{P}$ is a Z-transfer multi-path for Q that satisfies Lemma 2, and let $\boldsymbol{t} = (t_0, ..., t_k)$ and $\boldsymbol{u} = (u_0, ..., u_l)$ be two paths in $\mathcal{P}$, where $\boldsymbol{t}$ is leftward and $\boldsymbol{u}$ is rightward. If there are any indices a, b such that $t_{a+1} < u_b < t_a$ then $t_{a+1} < u_0 < u_l < t_a$, that is the whole path $\boldsymbol{u}$ is between t_{a+1} and t_a. An analogous statement holds if $\boldsymbol{t}$ is rightward and $\boldsymbol{u}$ is leftward.*

We would like to make here an observation that, although not used in our analysis later, may be of its own interest. If two paths $\boldsymbol{t}, \boldsymbol{u} \in \mathcal{P}$ satisfy the condition in Corollary 1 then we will say that $\boldsymbol{t}$ *eclipses* $\boldsymbol{u}$. If $\boldsymbol{t}$ *eclipses* $\boldsymbol{u}$ and $\boldsymbol{u}$ is straight then the total hop length of $\boldsymbol{t}$ must be greater than the total hop length of $\boldsymbol{u}$. This implies that this eclipse relation is a partial order on $\mathcal{P}$, as long as $\mathcal{P}$ satisfies Lemma 2.

4 The Greedy Algorithm

Our greedy algorithm LVG (for Longest-Viable-Gap) is very simple: at each step it creates a maximum-length gap that can be feasibly added to the schedule. (Recall that we assume the instance $\mathcal{J}$ to be feasible.) More formally, we describe this algorithm using the terminology of forbidden slots.

Algorithm LVG*:* Initialize $Z_0 = \emptyset$. The algorithm works in stages. In stage $s = 1, 2, ...$, we do this: If Z_{s-1} is an inclusion-maximal forbidden set that is viable for $\mathcal{J}$ then schedule $\mathcal{J}$ in the set $[r_{\min}, d_{\max}] - Z_{s-1}$ of time slots and output the computed schedule S_{LVG}. (The forbidden regions then become the gaps of S_{LVG}.) Otherwise, find the longest interval $X_s \subseteq [r_{\min}, d_{\max}] - Z_{s-1}$ for which $Z_{s-1} \cup X_s$ is viable and add X_s to Z_{s-1}, that is $Z_s \leftarrow Z_{s-1} \cup X_s$.

Note that after each stage the set Z_s of forbidden slots is a disjoint union of the forbidden intervals added at stages $1, 2, ..., s$. In fact, by the algorithm, any two consecutive forbidden intervals in Z_s must be separated by at least one busy time slot.

4.1 Approximation Ratio Analysis

We now show that the number of gaps in schedule S_{LVG} is within a factor of two from the optimum. More specifically, we will show that the number of gaps is at most $2g^* - 1$, where g^* is the minimum number of gaps in any schedule of $\mathcal{J}$. (We assume that $g^* \geq 1$, since for $g^* = 0$ it is easy to see that S_{LVG} will not contain any gaps.)

Proof outline. The outline of the proof is as follows. We start with an optimal schedule Q_0, namely the one with g^* gaps, and we will gradually modify it by introducing forbidden regions computed by Algorithm LVG. The resulting schedule, as it evolves, will be called the *reference schedule* and denoted Q_s. The construction of Q_s will ensure that it obeys Z_s, that the blocks of Q_{s-1} will be contained in blocks of Q_s, and that each block of Q_s contains some block of Q_{s-1}. As a result, each gap in the reference schedule shrinks over time and will eventually disappear.

The idea of the analysis is to charge forbidden regions X_s either to the blocks or to the gaps of Q_0. We will show that there are two types of forbidden regions, called *oriented* and *disoriented*, that each interior block of Q_0 can intersect at most one disoriented region (while exterior blocks cannot intersect any), and that introducing each oriented region causes at least one gap in the reference schedule to disappear. Further, each disoriented region intersects at least one block of Q_0. Therefore the total number of forbidden regions is bounded by the number of interior blocks plus the number of gaps in Q_0, which add up to $2g^* - 1$.

Construction of reference schedules. The first thing we need to do is to specify how we compute the reference schedule for each stage s of the algorithm. Let m be the number of stages of Algorithm LVG and $Z = Z_m$. For the rest of the proof we fix a Z-transfer multi-path $\mathcal{P}$ for Q_0 that satisfies Lemma 2, that is all paths in $\mathcal{P}$ are straight and they do not cross.

For any s, define $\mathcal{P}_s$ to be the set of those paths in $\mathcal{P}$ that start in the slots of Z_s. Thus $\mathcal{P}_1 \subset \mathcal{P}_2 \subset ... \subset \mathcal{P}_m = \mathcal{P}$. Note that $\mathcal{P}_s$ is a Z_s-transfer multi-path for Q_0, so shifting along the paths in $\mathcal{P}_s$ would give us a schedule that obeys Z_s. However, this construction would not give us reference schedules with the desired properties, because it can create busy slots in the middle of some gaps of the reference schedule, instead of "growing" the existing blocks.

To formalize the desired relation between consecutive reference schedules, we introduce another definition. Consider two schedules Q, Q', where Q obeys a forbidden set Y and Q' obeys a forbidden set Y' such that $Y \subseteq Y'$. We will say that Q' is an *augmentation* of Q if

(a1) $\mathsf{Supp}(Q) \subseteq \mathsf{Supp}(Q')$, and
(a2) each block of Q' contains a block of Q.

Recall that, by definition, forbidden slots are included in the support. Immediately from (a1), (a2) we obtain that if Q' is an augmentation of Q then the number of gaps in Q' does not exceed the number of gaps in Q.

Our objective is now to convert each $\mathcal{P}_s$ into another Z_s-transfer multi-path $\widehat{\mathcal{P}}_s$ such that if we take $Q_s = \mathsf{Shift}(Q_0, \widehat{\mathcal{P}}_s)$ then each Q_s will satisfy Lemma 2 and will be an augmentation of Q_{s-1}. For each path $\boldsymbol{t} = (t_0, ..., t_k) \in \mathcal{P}_s$, $\widehat{\mathcal{P}}_s$ will contain a *truncation* of $\boldsymbol{t}$, defined as a path $\hat{\boldsymbol{t}} = (t_0, ..., t_a, \tau)$, for some index a and slot $\tau \in (t_a, t_{a+1}]$.

We now describe the *truncation process*, an iterative procedure that constructs such reference schedules. The construction runs parallel to the algorithm. Fix some arbitrary stage s, suppose that we already have computed $\widehat{\mathcal{P}}_{s-1}$ and Q_{s-1}, and now we show how to construct $\widehat{\mathcal{P}}_s$ and Q_s. We first introduce some concepts and properties:

- We will maintain a set $\mathcal{R}$ of transfer paths, $\mathcal{R} \subseteq \mathcal{P}_s$. $\mathcal{R}$ is initialized to $\mathcal{P}_{s-1}$ and at the end of the stage we will have $\mathcal{R} = \mathcal{P}_s$. The cardinality of $\mathcal{R}$ is monotonically non-decreasing, but not the set $\mathcal{R}$ itself; that is, some paths may get removed from $\mathcal{R}$ and replaced by other paths. Naturally, being a subset of $\mathcal{P}_s$, $\mathcal{R}$ is a Y-transfer multi-path for Q_0, where Y is the set of starting slots of the paths in $\mathcal{R}$. Y is considered the current forbidden set; it will be initially equal to Z_{s-1} and at the end of the stage it will become Z_s. Since Y is implicitly defined by $\mathcal{R}$, we will not specify how it is updated.
- An any iteration, for each path $\boldsymbol{t} \in \mathcal{R}$ we maintain its unique truncation $\hat{\boldsymbol{t}}$. Let $\widehat{\mathcal{R}} = \{\hat{\boldsymbol{t}} : \boldsymbol{t} \in \mathcal{R}\}$. Similar to R, at each step $\widehat{\mathcal{R}}$ is a Y-transfer multi-path for Q_0, for Y defined above. Initially $\widehat{\mathcal{R}} = \widehat{\mathcal{P}}_{s-1}$ and when the stage ends we will set $\widehat{\mathcal{P}}_s = \widehat{\mathcal{R}}$.
- W is a schedule initialized to Q_{s-1}. We will maintain the invariant that W obeys Y and $W = \mathsf{Shift}(Q_0, \widehat{\mathcal{R}})$. At the end of the stage we will set $Q_s = W$.

We now describe one step of the truncation process. If $\mathcal{R} = \mathcal{P}_s$, we take $Q_s = W$, $\widehat{\mathcal{P}}_s = \widehat{\mathcal{R}}$, and we are done. Otherwise, choose arbitrarily a path $\boldsymbol{t} = (t_0, ..., t_k) \in \mathcal{P}_s - \mathcal{R}$. Without loss of generality, assume that $\boldsymbol{t}$ is rightward. We now have two cases.

(t1) If there is an idle slot τ in W with $t_0 < \tau \leq t_k$, then choose τ to be such a slot that is nearest to t_0. Let a be the largest index for which $t_a < \tau$. Then do this: add $\boldsymbol{t}$ to $\mathcal{R}$, set $\hat{\boldsymbol{t}} = (t_0, ..., t_a, \tau)$, and modify W by performing the shift along $\hat{\boldsymbol{t}}$, that is move the job from t_a to τ and from each t_c, $c = a-1, ..., 0$ to t_{c+1}. In this case τ will become a busy slot in W.

(t2) If no such idle slot exists, it means that there is some path $\boldsymbol{u} \in \mathcal{R}$ whose current truncation $\hat{\boldsymbol{u}} = (u_1, ..., u_b, \tau')$ ends at $\tau' = t_k$. In this case, we do this: modify W by undoing the shift along $\hat{\boldsymbol{u}}$ (that is, by shifting backwards: the job from each u_c, $c = 1, ..., b$, is moved to u_{c-1} and the job from τ' is moved to u_b), remove $\boldsymbol{u}$ from $\mathcal{R}$, add $\boldsymbol{t}$ to $\mathcal{R}$, and modify W by performing the shift along $\boldsymbol{t}$.

Note that any path $\boldsymbol{t}$ may enter and leave $\mathcal{R}$ several times, and each time $\boldsymbol{t}$ is truncated the endpoint τ of $\hat{\boldsymbol{t}}$ gets farther and farther from t_0. It is possible that the process will terminate with $\hat{\boldsymbol{t}} \neq \boldsymbol{t}$. However, if at some step case (t2) applied to $\boldsymbol{t}$, then this truncation step is vacuous, in the sense that after the step we have $\hat{\boldsymbol{t}} = \boldsymbol{t}$, and from now on $\boldsymbol{t}$ will never be removed from $\mathcal{R}$. These observations imply that the above truncation process always ends.

Lemma 3 *Fix some stage $s \geq 1$. Then*

(i) *Q_s is an augmentation of Q_{s-1}.*
(ii) $|\mathit{Supp}(Q_s) - \mathit{Supp}(Q_{s-1})| = |X_s|$.
(iii) *Furthermore, denoting by ξ^0 the number of idle slots of Q_{s-1} in X_s, we can write $|X_s| = \xi^- + \xi^0 + \xi^+$, such that $\mathit{Supp}(Q_s) - \mathit{Supp}(Q_{s-1})$ consists of the ξ^0 idle slots in X_s (which become forbidden in Q_s), the ξ^- nearest idle slots of Q_{s-1} to the left of X_s, and the ξ^+ nearest idle slots of Q_{s-1} to the right of X_s (which become busy in Q_s).*

Proof In the truncation process, at the beginning of stage s we have $W = Q_{s-1}$. During the process, we never change a status of a slot from busy or forbidden to idle. Specifically, in steps (t1), for non-trivial paths the first slot t_0 of $\hat{\boldsymbol{t}}$ was busy and will become forbidden (that is, added to Y) and the last slot τ was idle and will become busy. For trivial paths, $t_0 = t_k$ was idle and will become forbidden. In steps (t2), if $\boldsymbol{t}$ is non-trivial then t_0 was busy and will become forbidden, while t_k was and stays busy. If $\boldsymbol{t}$ is trivial, the status of $t_0 = t_k$ will change from busy to forbidden. In regard to path $\boldsymbol{u}$, observe that $\boldsymbol{u}$ must be a non-trivial path, since otherwise $\hat{\boldsymbol{u}}$ could not end at t_k. So undoing the shift along $\hat{\boldsymbol{u}}$ will cause u_0 to change from forbidden to busy. This shows that a busy or forbidden slot never becomes idle, so $\mathsf{Supp}(Q_{s-1}) \subseteq \mathsf{Supp}(Q_s)$.

New busy slots are only added in steps (t1), in which case τ is either in X_s, or is a nearest idle slot to X_s, in the sense that all slots between τ and X_s are in the support of W. This implies that Q_s is an augmentation of Q_{s-1}.

To justify (ii), it is sufficient to show the invariant that $|\mathcal{R} - \mathcal{P}_{s-1}| = |\mathsf{Supp}(W) - \mathsf{Supp}(Q_{s-1})|$. This is easy to justify: the invariant holds at the beginning (both sides are 0), in each step (t1) both sides increase by 1, and in each step (t2) both sides remain unchanged.

Finally, part (iii) follows from (i), (ii), using the fact that in steps of type (t1) the newly created busy slot τ is the first available idle slot between t_0 and t_k.

Using Lemma 3, we can make the relation between Q_{s-1} and Q_s more specific (see Figure 4). Let h be the number of gaps in Q_{s-1} and let $C_0, ..., C_h$ be the blocks of Q_{s-1} ordered from left to right. Thus C_0 and C_h are exterior blocks and all other are interior blocks. Then, for some indices $a \leq b$, the blocks of Q_s are $C_0, ..., C_{a-1}, D, C_{b+1}, ..., C_h$, where the new block D contains X_s as well as all blocks $C_a, ..., C_b$. As a result of adding X_s, in stage s the $b-a$ gaps of Q_{s-1} between C_a and C_b disappear from the reference schedule. For $b = a$, no gap disappears and $C_a \subset D$. In this case adding X_s causes C_a to expand.

Two types of regions. We now define two types of forbidden regions, as we mentioned earlier. Consider some forbidden region X_p. If all paths of $\mathcal{P}$ starting at X_p are leftward (resp. rightward) then we say that X_p is *left-oriented* (resp. *right-oriented*). A region X_p that is either left-oriented or right-oriented will be called *oriented*, and if it is neither, it will be called *disoriented*. Recall that trivial paths (consisting only of the start vertex) are considered both leftward and rightward. An oriented region may contain a number of trivial paths, but all non-trivial paths starting in this region must have the same orientation. A disoriented region must contain starting slots of at least one non-trivial leftward path and one non-trivial rightward path.

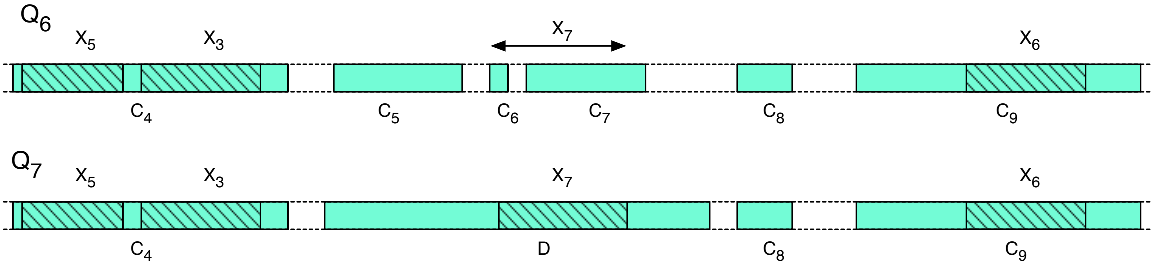

Fig. 4 An illustration for updating reference schedules. Blocks are shaded, with forbidden regions being pattern-shaded. The figure shows stage $s = 7$, when the new forbidden region X_7 is added. As a result, a new block D of Q_7 is created that contains X_7, as well as blocks C_5, C_6 and C_7 of Q_6. Other blocks of Q_6 remain unchanged.

Charging disoriented regions. Let $B_0, ..., B_{g^*}$ be the blocks of Q_0, ordered from left to right. The lemma below establishes some relations between disoriented forbidden regions X_s and the blocks and gaps of Q_0.

Lemma 4 (i) *If B_q is an exterior block then B_q does not intersect any disoriented forbidden regions.* (ii) *If B_q is an interior block then B_q intersects at most one disoriented forbidden region.* (iii) *If X_s is a disoriented forbidden region then X_s intersects at least one block of Q_0.*

Proof Part (i) is simple. Without loss of generality, suppose B_q is the leftmost block, that is $q = 0$, and let $x \in B_0 \cap X_s$. If $\boldsymbol{t} \in \mathcal{P}$ starts at x and is non-trivial then $\boldsymbol{t}$ cannot be leftward, because $\boldsymbol{t}$ ends in an idle slot and there are no idle slots to the left of x. So all paths from $\mathcal{P}$ starting in $B_0 \cap X_s$ are rightward. Thus X_s is right-oriented.

Now we prove part (ii). Fix some interior block B_q and, towards contradiction, suppose that there are two disoriented forbidden regions that intersect B_q, say X_s and $X_{s'}$, where X_s is before $X_{s'}$. Then there are two non-trivial transfer paths in $\mathcal{P}$, a rightward path $\boldsymbol{t} = (t_0, ..., t_k)$ starting in $X_s \cap B_q$ and a leftward path $\boldsymbol{u} = (u_0, ..., u_l)$ starting in $X_{s'} \cap B_q$. Both paths must end in idle slots of Q_0 that are not in Z and there are no such slots in $B_q \cup X_s \cup X_{s'}$. Therefore $\boldsymbol{t}$ ends to the right of $X_{s'}$ and $\boldsymbol{u}$ ends to the left of X_s. Thus we have $u_l < t_0 < u_0 < t_k$, which means that paths $\boldsymbol{t}$ and $\boldsymbol{u}$ cross, contradicting Lemma 2, which $\mathcal{P}$ was assumed to satisfy.

The last part, (iii), follows directly from the definition of disoriented regions, since if X_s were contained in a gap of Q_0 then all transfer paths starting in X_s would be trivial.

Charging oriented regions. This is the most nuanced part of our analysis. We want to show that at each stage when an oriented forbidden region is added, at least one gap in the reference schedule disappears.

To illustrate the principle of the proof, consider the first stage, when introducing the first forbidden interval X_1, and suppose that X_1 is left-oriented. Assume also, for simplicity, that all slots in X_1 are busy in Q_0, that is X_1 is included in some block $B = [f_B, l_B]$ of Q_0. In this case, $\mathcal{P}_1$ has $|X_1|$ leftward paths, all starting in X_1 and ending strictly to the left of B, each in a different idle slot.

We now examine the truncation process that defines Q_1. Imagine first that in stage 1 we always choose the path $\boldsymbol{t} \in \mathcal{P}_1 - \mathcal{R}$ whose endpoint is nearest to X_1. Let G be the gap immediately to the left of B. Since G itself is a candidate for a forbidden region, we have $|G| \le |X_1|$. The path truncated in the first iteration will end in the slot $f_B - 1$, adjacent to B on the left, and this slot will now become busy, extending B to a larger block. By the choice of this path, the remaining pending paths end to the left of $f_B - 1$. Thus the next path will get truncated at the slot $f_B - 2$, and so on. During this process, no truncated path will be removed from $\mathcal{R}$, that is the case (t2) will never apply. This implies that after $|G|$ iterations all slots in G will become busy and G will disappear. If $|X_1| > |G|$, the remaining iterations will reduce the number of idle slots further, without increasing the number of gaps.

Next, we observe that it does not matter how the paths from $\mathcal{P}_1 - \mathcal{R}$ are chosen in this process – we claim that G will disappear even if these choices are arbitrary. In this case it may happen that when we choose some path $\boldsymbol{t} \in \mathcal{P}_s - \mathcal{R}$, its endpoint t_k may be busy, as in Case (t2). Then $\boldsymbol{t}$ will be added to $\mathcal{R}$, while the path $\boldsymbol{u}$ whose truncation $\hat{\boldsymbol{u}}$ ends in t_k will be removed from $\mathcal{R}$. If this happens though we have that the endpoint of $\boldsymbol{u}$ is to the left of t_k. Thus eventually we will have to make $|G|$ iterations of type (t1), where the nearest still idle slot of G becomes busy, and after the last of these iterations G will disappear.

Let us now consider the same scenario but at some stage $s \geq 2$. In this case the argument is more subtle. The difficulty that arises here is that during this process some paths from $\mathcal{P}_{s-1}$ may be "reincarnated", that is removed from $\mathcal{R}$ in (t2). What's worse, these paths could even be rightward, so the reasoning for $s = 1$ in the previous paragraph does not apply. To handle this difficulty, our argument relies critically on the structure of $\mathcal{P}_s$ (as described in Lemma 2), and it shows that if the gap to the left of X_s does not disappear in Q_s then the gap to the right of X_s will have to disappear.

Lemma 5 *If* $X_s = [f_{X_s}, l_{X_s}]$ *is an oriented region then at least one gap of* Q_{s-1} *disappears in* Q_s.

Proof We start by making a few simple observations. If X_s contains a gap of Q_{s-1}, then this gap will disappear when stage s ends. Note also that X_s cannot be strictly contained in a gap of Q_{s-1}, since otherwise we could increase X_s, contradicting the algorithm. Thus for the rest of the proof we can assume that X_s has a non-empty intersection with exactly one block $B = [f_B, l_B]$ of Q_{s-1}. If B is an exterior block then Lemma 3 immediately implies that the gap adjacent to B will disappear, because X_s is at least as long as this gap. Therefore we can assume that B is an interior block. Denote by G and H, respectively, the gaps immediately to the left and to the right of B. By symmetry, we can assume that X_s is left-oriented, so all paths in $\mathcal{P}_s - \mathcal{P}_{s-1}$ are leftward.

Summarizing, we have $X_s \subset G \cup B \cup H$ and all sets $G - X_s$, $B \cap X_s$, $H - X_s$ are not empty. We will show that at least one of the gaps G, H will disappear in Q_s. The proof is by contradiction; we assume that both G and H have some idle slots after stage s and show that this assumption leads to a contradiction with Lemma 2, which $\mathcal{P}$ was assumed to satisfy.

We first give the proof for the case when $X_s \subseteq B$. From the algorithm, $|X_s| \geq \max(|G|, |H|)$. By Lemma 3, $|X_s|$ idle slots immediately to the left or right of X_s will become busy. It is not possible that all these slots are on one side of X_s, because then the gap on this side would disappear, contradicting the assumption from the paragraph above. Therefore both gaps shrink; in particular, the rightmost slot of G and the leftmost slot of H become busy in Q_s.

At any step of the truncation process (including previous stages), when some path $\boldsymbol{t} = (t_0, ..., t_k) \in \mathcal{R}$ is truncated to $\hat{\boldsymbol{t}} = (t_0, ..., t_a, \tau)$, all slots between t_0 and τ are either forbidden or busy, so all these slots are in the same block of W. This and the assumption that G and H do not disappear in Q_s implies that, in stage s, when we truncate $\boldsymbol{t} = (t_0, ..., t_k)$ and Case (t2) occurs, then the path $\boldsymbol{u}$ added to $\mathcal{R}$ must start in B. Therefore at all steps of stage s the paths in $\mathcal{P}_s - \mathcal{R}$ start in B.

Let $\boldsymbol{u} \in \mathcal{P}_s$ be the path whose truncation $\hat{\boldsymbol{u}}$ ends in $f_B - 1$ (the rightmost slot of G) right after stage s. There could be now some busy slots immediately to the left of $f_B - 1$ introduced in stage s, but no transfer paths start at these slots, because they were idle in Q_0. Together with the previous paragraph, this implies that $\boldsymbol{u}$ must be leftward and that it starts in B. If $\boldsymbol{u}$ does not start in X_s, it means that, during the truncation process in stage s, it was added to $\mathcal{R}$ replacing some other path $\boldsymbol{u}'$ which, by the same argument, must also start in B. Proceeding in this manner, we can define a sequence $\boldsymbol{u}^1, ..., \boldsymbol{u}^p = \boldsymbol{u}$ of transfer paths from $\mathcal{P}_s$, all starting in B, such that $\boldsymbol{u}^1$ is a leftward path starting in X_s (so $\boldsymbol{u}^1$ was in $\mathcal{P}_s - \mathcal{P}_{s-1}$ when stage s started) and, for $i = 1, ..., p-1$, $\boldsymbol{u}^{i+1}$ is the path replaced by $\boldsymbol{u}^i$ in $\mathcal{R}$ at some step of type (t2) during stage s. Similarly, define $\boldsymbol{v}$ to be the rightward path whose truncation ends in the leftmost slot of H and let $\boldsymbol{v}^1, ..., \boldsymbol{v}^q = \boldsymbol{v}$ be the similarly defined sequence for $\boldsymbol{v}$, namely $\boldsymbol{v}^1$ is a leftward path starting in X_s and, for $i = 1, ..., q-1$, $\boldsymbol{v}^{i+1}$ is the path replaced by $\boldsymbol{v}^i$ in $\mathcal{R}$. Our goal is to show that there are paths $\boldsymbol{u}^i$ and $\boldsymbol{v}^j$ that cross, which would give us a contradiction.

Several steps in our argument will rely on the following simple observation which follows directly from the definition of the truncation process. Note that this observation holds even if $\boldsymbol{t}$ is trivial.

Observation 1 *Suppose that at some iteration of type (t2) in the truncation process we choose a path* $\boldsymbol{t} = (t_0, ..., t_k) \in \mathcal{P}_s - \mathcal{R}$ *and it replaces a path* $\boldsymbol{t}' = (t'_0, ..., t'_l)$ *in* $\mathcal{R}$ *(because* $\hat{\boldsymbol{t}}'$ *ended at* t_k*). Then* $\min(t'_0, t'_l) < t_k < \max(t'_0, t'_l)$.

Let $\boldsymbol{u}^g$ be the leftward path among $\boldsymbol{u}^1, ..., \boldsymbol{u}^p$ whose start point u^g_0 is rightmost. Note that $\boldsymbol{u}^g$ exists, because $\boldsymbol{u}^p$ is a candidate for $\boldsymbol{u}^g$. Similarly, let $\boldsymbol{v}^h$ be the rightward path among $\boldsymbol{v}^1, ..., \boldsymbol{v}^q$ whose start point v^h_0 is leftmost.

Claim We have (i) $u^g_0 \geq f_{X_s}$ and (ii) the leftward paths in $\{\boldsymbol{u}^1, ..., \boldsymbol{u}^p\}$ cover the interval $[f_B, u^g_0]$, in the following sense: for each $z \in [f_B, u^g_0]$ there is a leftward path $\boldsymbol{u}^i = (u^i_0, ..., u^i_{k_i})$ such that $u^i_{k_i} \leq z \leq u^i_0$.

The proof of Claim 4.1 is very simple. Part (i) holds because $\boldsymbol{u}^1$ is leftward and $u_0^1 \geq f_{X_s}$. Property (ii) then follows by applying Observation 1 iteratively to show that the leftward paths among $\boldsymbol{u}^g, ..., \boldsymbol{u}^p$ cover the interval $[f_B, u_0^g]$. More specifically, for $i = g, ..., p-1$, we have that the endpoint $u_{k_i}^i$ of $\boldsymbol{u}^i$ is between u_0^{i+1} and $u_{k_{i+1}}^{i+1}$, the start and endpoints of $\boldsymbol{u}^{i+1}$. As i increases, $u_{k_i}^i$ may move left or right, depending on whether $\boldsymbol{u}$ is leftward or rightward, but it satisfies the invariant that the interval $[u_{k_i}^i, u_0^g]$ is covered by the leftward paths among $\boldsymbol{u}^g, ..., \boldsymbol{u}^i$, and the last value of $u_{k_i}^i$, namely $u_{k_p}^p$, is before f_B. This implies Claim 4.1.

Claim We have (i) $v_0^h < f_{X_s}$ and (ii) the rightward paths in $\{\boldsymbol{v}^1, ..., \boldsymbol{v}^q\}$ cover the interval $[v_0^h, l_B]$, that is for each $z \in [v_0^h, l_B]$ there is a rightward path $\boldsymbol{v}^j = (v_0^j, ..., v_{l_j}^j)$ such that $v_0^j \leq z \leq v_{l_j}^j$.

The argument for part (ii) is analogous to that in Claim 4.1, so we only show part (i). Note that part (i) is not symmetric to part (i) of Claim 4.1, and proving (i) requires a bit of work. We show that if $\boldsymbol{v}^e$ is the first non-trivial rightward path among $\boldsymbol{v}^1, ..., \boldsymbol{v}^q$ then $v_0^e < f_{X_s}$. This $\boldsymbol{v}^e$ exists because $\boldsymbol{v}^q$ is a candidate. The key fact here is that $e \neq 1$, because X_s is left-oriented. Since $\boldsymbol{v}^0, ..., \boldsymbol{v}^{e-1}$ are leftward, Observation 1 implies that $v_{l_0}^0 > v_{l_1}^1 > ... > v_{l_{e-1}}^{e-1}$, that is the sequence of their endpoints is decreasing. But $v_{l_0}^0 \leq v_0^0 \in X_s$, so we get that $v_{l_{e-1}}^{e-1} \leq l_{X_s}$. Applying Observation 1 again, we obtain $v_0^e < v_{l_{e-1}}^{e-1}$, because $\boldsymbol{v}^e$ is rightward. But X_s is left-oriented, so $\boldsymbol{v}^e$ cannot start in X_s, which gives us $v_0^e < f_{X_s}$, as claimed. This completes the proof of Claim 4.1.

Continuing the proof of the lemma, we now focus on v_0^h. The two claims above imply that $v_0^h < u_0^g$. Since the paths $\boldsymbol{u}^1, ..., \boldsymbol{u}^p$ cover $[f_B, u_0^g]$ and $v_0^h \in [f_B, u_0^g]$, there is a leftward path $\boldsymbol{u}^i$ such that $u_{a+1}^i < v_0^h < u_a^i$, for some index a. Since the rightward paths among $\boldsymbol{v}^1, ..., \boldsymbol{v}^q$ cover the interval $[v_0^h, l_B]$ and $u_a^i \in [v_0^h, l_B]$, there is a rightward path $\boldsymbol{v}^j$ such that $v_b^j < u_a^i < v_{b+1}^j$, for some index b. By these inequalities and our choice of $\boldsymbol{v}^h$, we have

$$u_{a+1}^i < v_0^h \leq v_0^j \leq v_b^j < u_a^i < v_{b+1}^j.$$

This means that $\boldsymbol{u}^i$ and $\boldsymbol{v}^j$ cross, giving us a contradiction.

We have thus completed the proof of the lemma when $X_s \subseteq B$. We now extend it to the general case, when X_s may overlap G or H or both. Recall that both $G - X_s$ and $H - X_s$ are not empty. All we need to do is to show that the idle slots adjacent to $X_s \cup B$ will become busy in Q_s, since then we can choose paths $\boldsymbol{u}$, $\boldsymbol{v}$ and the corresponding sequences as before, and the construction above applies.

Suppose that $X_s \cap G \neq \emptyset$. We claim that the slot $l_{X_s} - 1$, namely the slot of G adjacent to X_s, must become busy in Q_s. Indeed, if this slot remained idle in Q_s then $X_s \cup \{l_{X_s} - 1\}$ would be a viable forbidden region in stage s, contradicting the maximality of X_s. By the same argument, if $X_s \cap H \neq \emptyset$ then the slot of H adjacent to X_s will become busy in Q_s. This immediately takes care of the case when X_s overlaps both G and H.

It remains to examine the case when X_s overlaps only one of G, H. By symmetry, we can assume that $X_s \cap G \neq \emptyset$ but $X_s \cap H = \emptyset$. By the paragraph above, slot $f_{X_s} - 1$ will be busy in Q_s, so we only need to show that slot $l_B + 1$ will be also busy. Towards contradiction, if $l_B + 1$ is not busy in Q_s, Lemma 3 implies that the nearest $|X_s \cap B|$ idle slots to the left of X_s will become busy. By the choice of X_s we have $|X_s| \geq |G|$, so $|X_s \cap B| \geq |G - X_s|$, which would imply that G will disappear in Q_s, giving us a contradiction. Thus $l_B + 1$ must be busy in Q_s.

Putting everything together now, Lemma 4 implies that the number of disoriented forbidden regions among $X_1, ..., X_m$ is at most $g^* - 1$, the number of interior blocks in Q_0. Lemma 5, in turn, implies that the number of oriented forbidden regions among $X_1, ..., X_m$ is at most g^*, the number of gaps in Q_0. Thus $m \leq 2g^* - 1$. This gives us the main result of this paper.

Theorem 1 *Suppose that the minimum number of gaps in a schedule of $\mathcal{J}$ is $g^* \geq 1$. Then the schedule S_{LVG} computed by Algorithm* LVG *has at most $2g^* - 1$ gaps.*

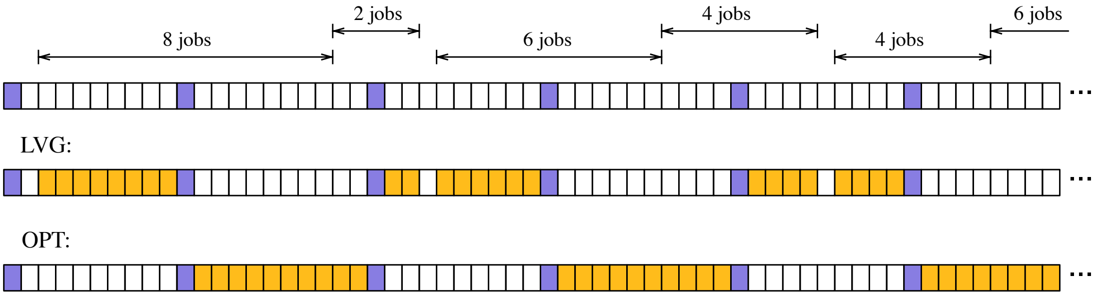

Fig. 5 The lower-bound construction for $k = 5$. Only the leftmost portion of the instance is shown. Tight jobs are dark-shaded and loose jobs are light-shaded in the schedules. Bundles are represented by bi-directional arrows spanning the interval between the release time and deadline for this bundle and labelled by the number of jobs in the bundle.

4.2 A Lower Bound for Algorithm LVG

We now show that our analysis of Algorithm LVG in the previous section is tight. For any $k \geq 2$ we show that there is an instance $\mathcal{J}_k$ on which Algorithm LVG constructs a schedule with $2k - 1$ gaps, while the optimum schedule has $g^* = k$ gaps.

$\mathcal{J}_k$ has $2k$ tight jobs, where, for $a = 0, 1, ..., k-1$, the $2a$'th tight job has release time and deadline equal $\tau_{2a} = (4k+1)a$, and the $(2a+1)$'st tight job has release time and deadline equal $\tau_{2a+1} = (4k+1)a + 2k$. For each tight job, except first and last, there is also an associated bundle of loose jobs. For $a = 1, ..., k-1$, the bundle associated with the $(2a)$'th tight job has $2a$ identical jobs with release times $\tau_{2a} - 2a$ and deadlines $\tau_{2a} + 2a$, and the bundle associated with the $(2a-1)$'th tight job has $2k - 2a$ identical jobs with release times $\tau_{2a-1} - 2k + 2a$ and deadlines $\tau_{2a-1} + 2k - 2a$. Algorithm LVG will first create $k - 1$ forbidden regions $[\tau_{2a-1} + 1, \tau_{2a} - 1]$ for $a = 1, ..., k-1$, each of length $2k$, and then another k regions of length 1 inside the intervals $[\tau_{2a} + 1, \tau_{2a+1} - 1]$ for $a = 0, 1, ..., k-1$, resulting in $2k - 1$ gaps. For $a = 1, ..., k-1$, the optimum will schedule the jobs from batch $2a$ before τ_{2a}, and the jobs from batch $2a - 1$ after τ_{2a-1}. This produces only k gaps $[\tau_{2a} + 1, \tau_{2a+1} - 1]$, for $a = 0, 1, ..., k-1$. The construction is illustrated in Figure 5, which shows $\mathcal{J}_5$, the schedule produced by Algorithm LVG, and the optimal schedule.

5 Implementation in Time $O(n(g^* + 1) \log n)$

We now show how to implement Algorithm LVG in time $O(n(g^* + 1) \log n)$ and memory $O(n)$, where g^* is the optimum number of gaps. As mentioned in Section 2, we assume that initially all release times are different and all deadlines are different. Any instance can be modified to satisfy this condition, without affecting feasibility, in time $O(n \log n)$. We also assume that jobs 1 and n (with minimum and maximum deadlines) are tight, that is $r_1 = d_1 = r_{\min}$ and $r_n = d_n = d_{\max}$. By Theorem 1, the number of stages in the algorithm is at most $2g^* - 1$, so it is sufficient to show that each stage s can be implemented in time $O(n \log n)$.

Rather than maintaining forbidden regions explicitly, our algorithm will remove these regions from the timeline altogether. In addition, we will maintain the invariant that after each stage all release times are different and all deadlines are different, without affecting the computed solution. Having all release times different and all deadlines different will allow us to efficiently locate the next viable forbidden region of maximum length. In order to maintain this invariant, at stage s, all deadlines that fall into X_s, and possibly some earlier deadlines as well, will be shifted to the left. This needs to be done with care; in particular, some deadlines may need to be reordered. Roughly, this will be accomplished by assigning the deadlines moving back in time, starting from time $f_{X_s} - 1$, and breaking ties in favor of the jobs with later release times. An analogous procedure will be applied to release times, starting from $l_{X_s} + 1$ and moving forward in time. After this, all release times and deadlines after l_X can be reduced by $l_{X_s} - f_{X_s} + 1$, which has the effect of removing X_s from the timeline.

We now provide the details. We will use notation x_j and y_j for the release times and deadlines, as they vary over time. Initially, $(x_j, y_j) = (r_j, d_j)$, for all j. As explained before, we maintain the invariant that all $x_1, ..., x_n$ are different and all $y_1, ..., y_n$ are different. We assume that at the beginning of each stage the algorithm computes a permutation of each sequence $(x_j)_j$ and $(y_j)_j$ in increasing order. This will take time $O(n \log n)$.

Finding a maximum viable forbidden region. We claim that an interval $[u, v]$ is a viable forbidden region if and only if it does not contain the whole range of any job, where the *range* of j is defined to be $[x_j, y_j]$, the interval where j can be scheduled. In other words, for any j, either $x_j \notin [u, v]$ or $y_j \notin [u, v]$. Indeed, if $[u, v]$ contains the range of j then $[u, v]$ is, trivially, not a viable forbidden region. On the other hand, if $[u, v]$ does not contain the range of any job, then we can schedule all jobs outside $[u, v]$ as follows: first schedule each job released before u at its release time, and then schedule all remaining jobs at their deadlines. By the invariant described above, this is a feasible schedule.

The above paragraph implies that the maximum viable forbidden region has the form $[x_a + 1, y_b - 1]$ for two different jobs a, b such that $y_a \in [x_a, y_b - 1]$ and $x_b \in [x_a + 1, y_b]$. We now show how to find such a pair a, b in linear time.

We use variables a and b to store the indices of the largest viable forbidden interval $[x_a + 1, y_b - 1]$ found so far, and $\theta = y_b - x_a - 1$ will be its length. Initially, we can set $a = b = 1$ and $\theta = -1$. (Recall that 1 is a special job with $r_1 = d_1 = r_{\min}$. The values of x_1 and y_1 will not change during the algorithm.) At the end, $[x_a + 1, y_b - 1]$ will be the maximum forbidden interval. We use also two other indices α and β to iterate over intervals $[x_\alpha + 1, y_\beta - 1]$ that are viable forbidden regions and potential candidates for a maximum one. For any choice of α and β, if $y_\beta - x_\alpha - 1 > \theta$, we update $(a, b, \theta) \leftarrow (\alpha, \beta, y_\beta - x_\alpha - 1)$.

It remains to show how we list all candidate intervals $[x_\alpha + 1, y_\beta - 1]$. We start with $\alpha = \beta = 1$. Then, at any step we do this. If $x_\beta \leq x_\alpha$ then we choose the minimum $y_\gamma > y_\beta$. Note that $[x_\alpha + 1, y_\gamma - 1]$ is also a viable forbidden region because it does not contain x_β. So we set $\beta \leftarrow \gamma$ and continue. Otherwise, we know that $[x_\alpha + 1, y_\beta - 1]$ is the maximum viable forbidden region starting at $x_\alpha + 1$. In this case we find the smallest $x_\gamma > x_\alpha$, we update $\alpha \leftarrow \gamma$ and proceed. Note that such γ exists and satisfies $x_\gamma \leq y_\beta$, because β itself is a candidate for γ.

At each iteration, we either increment α or β, so the total number of iterations will not exceed $2n$. Since we store all x_i's and all y_i's in a sorted order, each step will take constant time, so the overall running time to find a maximum forbidden region is $O(n)$.

Compressing forbidden regions. Let $X = [f_X, l_X]$ denote the maximum forbidden region X_s found in the current stage. We now want to compress X, that is remove it from the timeline. All deadlines inside X are changed to $f_X - 1$ and all release times inside X are changed to $l_X + 1$. Next, we decrement all release times and deadlines after l_X by $l_X - f_X + 1$. Note that this operation does not change the ordering of the x_i's or the y_i's, so their values can trivially be updated in linear time. Thus the compression stage can be done in linear time.

The compression phase does not affect feasibility, that is a schedule that obeys X for the instance before the compression can be converted into a schedule for the instance after the compression, and vice versa. It is also important to observe, at this point, that in the modified instance both jobs a and b (for which X was equal $[x_a + 1, y_b - 1]$) are now tight, that is $x_a = y_a$ and $x_b = y_b$. Thus the forbidden intervals found in the subsequent stages will not contain these two slots, guaranteeing that all found intervals are disjoint.

As a result of compressing the schedule, different jobs may end up having equal deadlines or release times, violating our invariant. It thus remains to show how to modify the instance to restore this invariant.

Updating release times and deadlines. We show how to update the deadlines y_i; the computation for the release times is similar. Recall that after compressing the forbidden interval X all deadlines from this interval have been reset to $f_X - 1$. Let K be the set of jobs whose deadlines were in X before the compression. We set $t = f_X - 1$ and decrement it repeatedly while updating K and the deadlines of jobs removed from K. Specifically, this works as follows: If $K = \emptyset$, we are done. Otherwise, if there is a job $j \notin K$ with $y_j = t$, we add j to K (there can only be one such job). We then identify job $k \in K$ with maximum release time x_k, remove it from K, set $y_k \leftarrow t$, and decrement t.

Here, again, we need to argue that this modification does not affect feasibility. It is sufficient to show that for a collection K of jobs with the same deadline t, if $k \in K$ has the maximum release time, then reducing the deadlines of all jobs in $K - \{k\}$ by 1 does not destroy feasibility. Indeed, this follows from a simple exchange argument: if some other job $c \in K$ is scheduled at time t then, since k is released after c, we can exchange k and c in this schedule and thus have k scheduled at time t. This new schedule is feasible even after all deadlines of jobs in $K - \{k\}$ are reduced by 1.

This stage can be implemented in time $O(n \log n)$ using a priority queue to store K. Then each insertion of a new job j (for which $y_j = t$) and deletion of $k \in K$ with maximum x_k will take time $O(\log n)$.

Output. For each $s = 1, 2, ..., m$, do this: let $X = [f_X, l_X]$ be the forbidden region X_s found in this stage, and let δ be the total size of the forbidden regions found by the algorithm in stages $1, 2, ..., s-1$ that are located before f_X. Then the algorithm outputs $[f_X + \delta, l_X + \delta]$ as the forbidden region X_s. This computation can be performed in time $O(n \log n)$.

6 Final Comments

A number of interesting questions remain open, the most intriguing one being whether it is possible to efficiently approximate the optimum solution within a factor of $1 + \epsilon$, for arbitrary $\epsilon > 0$. Ideally, such an algorithm should run in near-linear time. We hope that our results in Section 2, that elucidate the structure of the set of transfer paths, will be helpful in making progress towards designing such an algorithm.

Our 2-approximation result for Algorithm LVG remains valid for the more general scheduling problem where jobs have arbitrary processing times and preemptions are allowed, because then a job with processing time p can be thought of as p identical unit-length jobs. For this case, although Algorithm LVG can be still easily implemented in polynomial time, we do not have an implementation that would significantly improve on the $O(n^5)$ running time from [4].

Acknowledgements. Marek Chrobak has been supported by National Science Foundation grants CCF-0729071 and CCF-1217314. Mohammad Taghi Hajiaghayi has been supported in part by by the National Science Foundation CAREER award 1053605, Office of Naval Research YIP award N000141110662, and a University of Maryland Research and Scholarship Award (RASA). Fei Li has been supported by National Science Foundation grants CCF-0915681 and CCF-1146578.

References

1. Susanne Albers. Energy-efficient algorithms. *Communications of the ACM*, 53(5):86–96, May 2010.
2. Susanne Albers and Antonios Antoniadis. Race to idle: new algorithms for speed scaling with a sleep state. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1266–1285, 2012.
3. Philippe Baptiste. 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)*, pages 364–367, 2006.
4. Philippe Baptiste, Marek Chrobak, and Christoph Dürr. Polynomial time algorithms for minimum energy scheduling. In *Proceedings of the 15th Annual European Symposium on Algorithms (ESA)*, pages 136–150, 2007.
5. Philippe Chretienne. On single-machine scheduling without intermediate delays. *Discrete Applied Mathematics*, 156(13):2543 – 2550, 2008.
6. Erik D. Demaine, Mohammad Ghodsi, Mohammad Taghi Hajiaghayi, Amin S. Sayedi-Roshkhar, and Morteza Zadimoghaddam. Scheduling to minimize gaps and power consumption. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 46–54, 2007.
7. Sandy Irani and Kirk R. Pruhs. Algorithmic problems in power management. *SIGACT News*, 36(2):63–76, June 2005.