A Greedy Approximation Algorithm for Minimum-Gap Scheduling

Marek Chrobak[1⋆] , Uriel Feige[2] , Mohammad Taghi Hajiaghayi[3⋆⋆], Sanjeev Khanna[4] , Fei Li[5⋆⋆⋆], and Seffi Naor[6]

[1] Department of Computer Science, Univ. of California at Riverside, USA.
[2] Department of Computer Science and Applied Mathematics, the Weizmann Institute, Israel.
[3] Computer Science Department, Univ. of Maryland, College Park, USA.
[4] Department of Computer and Information Science, Univ. of Pennsylvania, Philadelphia, USA.
[5] Department of Computer Science, George Mason University, USA.
[6] Computer Science Department, Technion, Israel.

Abstract. We consider scheduling of unit-length jobs with release times and deadlines to minimize the number of gaps in the schedule. The best algorithm for this problem runs in time $O(n^4)$ and requires $O(n^3)$ memory. We present a simple 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 $O(ng^* \log n)$, 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

⋆ Research supported by NSF grant CCF-1217314.
⋆⋆ Supported in part by NSF CAREER award 1053605, ONR YIP award N000141110662, and a University of Maryland Research and Scholarship Award (RASA). The author is also with AT&T Labs–Research.
⋆⋆⋆ Research supported by NSF grants CCF-0915681 and CCF-1146578.

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.

Our results. We give a simple, greedy algorithm for minimum-gap scheduling of unit-length jobs that computes a near-optimal solution. Our 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 $O(ng^* \log n)$; 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 $O(n^3 g^*)$.) The idea of the algorithm is to add gaps one by one, at each step adding the longest gap for which there exists a feasible schedule. Our analysis is based on new structural properties of schedules with gaps, that may be of independent interest.

Related work. Prior to the paper by Baptiste [3], Chretienne [5] studied versions of scheduling where only schedules without gaps are allowed. The algorithm in [3] can be extended to handle jobs of arbitrary length, with preemptions, 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 proposed by Irani and Pruhs [7] is concerned with computing minimum-energy schedules in the model where the processor uses energy at constant rate when processing jobs, but 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 $g^* = o(n)$.

The power-down model from [7] can be generalized further to include the speed-scaling capability. The reader is referred to surveys in [1, 7], for more information on the models involving speed-scaling.

2 Preliminaries

We assume that the time axis is partitioned into unit-length time slots numbered $0, 1, ...$ 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 release time r_j and deadline d_j, both integers. Without loss of generality, $r_j \leq d_j$ for each j. By a standard exchange argument, we can also assume that all release times are distinct and that all

deadlines are distinct. 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.

A *(feasible) schedule* S of $\mathcal{J}$ is 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 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, without changing the number of gaps in the optimum solution.

Throughout the paper we assume that the given instance $\mathcal{J}$ is *feasible*, that is, it has a schedule. Feasibility can be checked by running the greedy earliest-deadline-first algorithm (EDF): process the time slots from left to right and at each step schedule the earliest-deadline pending job, if there is any. Then $\mathcal{J}$ is feasible if and only if no job misses its deadline in EDF. Also, since a schedule can be thought of as a bipartite matching between jobs and time slots, by a simple adaptation of Hall's theorem we obtain that $\mathcal{J}$ is feasible if and only if for any time interval $[t, u]$ we have $|\mathit{Load}(t, u)| \leq u - t + 1$, where $\mathit{Load}(t, u) = \{j : t \leq r_j \leq d_j \leq u\}$ is the set of jobs that must be scheduled in $[t, u]$.

Scheduling with forbidden slots. We consider a more general model where some slots in $[r_{\min}, d_{\max}]$ are designated as *forbidden*, namely no job is allowed to be scheduled in them. 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. 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 *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. The criterion for feasibility generalizes naturally to scheduling with forbidden slots, as follows: a forbidden set Z is viable if and only if $|\mathit{Load}(t, u)| \leq |[t, u] - Z|$, holds for all $t \leq u$, where $[t, u] - Z$ is the set of non-forbidden slots between t and u (inclusive).

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}. 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 generalized 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 in Q starting in Z.*

Proof. ($\Leftarrow$) 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.

($\Rightarrow$) 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. There is a set of disjoint alternating paths in $\mathcal{G}$ (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 $\boldsymbol{t} = (t_0, t_1, ..., t_k)$ for slot x of Q. Note that if t_0 is idle in Q then this path is trivial – it ends at t_0. 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 show 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). This property is generally not true, but we show that in such a case Q can be replaced by a schedule with the same support that satisfies these 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 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: $t_a < u_{b+1} < t_{a+1} < u_b$ or $u_b < t_{a+1} < u_{b+1} < t_a$, or $t_{a+1} < u_b < t_a < u_{b+1}$, or $u_{b+1} < t_a < u_b < t_{a+1}$. 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.

Lemma 2. *Let Q be a feasible schedule and let Z be a viable forbidden set. Then there is a schedule Q' such that* (i) $\mathsf{Supp}(Q') = \mathsf{Supp}(Q)$*, and* (ii) Q' *has a Z-transfer multi-path $\mathcal{P}$ in which all paths in $\mathcal{P}$ are straight and do not cross.*

Proof. We only show here how to remove crossings. (The complete proof will appear in the full paper.) Consider now two paths in $\mathcal{R}$ that cross, $\boldsymbol{t} = (t_0, ..., t_k)$ and $\boldsymbol{u} = (u_0, ..., u_l)$. 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. If $t_a < u_{b+1} < t_{a+1} < u_b$, 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 1 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 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 $(u_0, ..., u_b, t_{a+1}, ..., t_k)$.

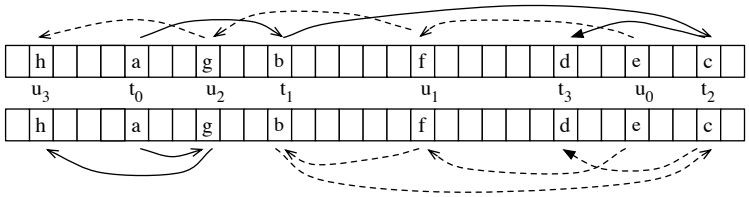

Fig. 1. Removing path crossings in the proof of Lemma 2.

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 without crossings. 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. □

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.

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.*

4 The Greedy Algorithm

Our greedy algorithm LVG (for Longest-Viable-Gap) is simple: at each step it creates a maximum-length gap that can be feasibly added to the schedule. 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 $Z_s \leftarrow Z_{s-1} \cup X_s$.

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, any two consecutive forbidden intervals in Z_s must be separated by at least one busy time slot.

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

Proof outline. In our proof, we start with an optimal schedule Q_0, namely the one with g^* gaps, and we 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 either to the blocks or to the gaps of Q_0. We 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, and that introducing each oriented region causes at least one gap in the reference schedule to disappear. Further, each disoriented region intersects a block of Q_0. As a result, the total number of forbidden regions is at most the number of interior blocks plus the number of gaps in Q_0, which add up to $2g^* - 1$.

Construction of reference schedules. 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. In particular, we have $\mathcal{P} = \mathcal{P}_m$.

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 region 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 the above definition 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 the 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:

- $\mathcal{R}$ is a set of transfer paths, $\mathcal{R} \subseteq \mathcal{P}_s$. It 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 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, $\mathcal{R}$ is a Y-transfer multi-path for Q_0, where Y is the set of starting slots of the paths in $\mathcal{R}$. Y 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 truncation $\hat{\boldsymbol{t}}$. Let $\widehat{\mathcal{R}} = \{\hat{\boldsymbol{t}} : \boldsymbol{t} \in \mathcal{R}\}$. 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 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 $Q_s = W$.

Consider now some step of this 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 $\hat{\boldsymbol{t}}$ (so τ will now 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 $\hat{\boldsymbol{u}}$ 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), 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 trivial, 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 truncation process always ends.

Lemma 3. *Fix some stage $s \geq 1$. Then* (i) *Q_s is an augmentation of Q_{s-1}.* (ii) *$|\mathsf{Supp}(Q_s) - \mathsf{Supp}(Q_{s-1})| = |X_s|$.* (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 $\mathsf{Supp}(Q_s) - \mathsf{Supp}(Q_{s-1})$ 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. 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 $\boldsymbol{t}$ was busy and will become forbidden 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 non-trivial, 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 (i) and (ii), note that the slots in $X_s - \mathsf{Supp}(Q_{s-1})$ will become forbidden in Q_s and that for each slot $x \in X_s \cap \mathsf{Supp}(Q_{s-1})$ there will be a step of type (t1) in stage s of the truncation process when we will chose a non-trivial path $\boldsymbol{t}$ starting at $t_0 = x$, so in this step a new busy slot will be created. This implies (ii), and together with (i) it also implies (iii). □

Using Lemma 3, we can make the relation between Q_{s-1} and Q_s more specific. 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.

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. 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, proving (i).

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.

Part (iii) follows from the definition of disoriented regions, since if X_s were contained in a gap 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 when an oriented forbidden region is added, at least one gap in the reference schedule disappears. The general idea is that if X_s is left-oriented and G is the nearest gap to the left of X_s, then by the maximality of X_s we have $|X_s| \geq |G|$. So when we process the leftward paths starting in X_s, the truncation process will eventually fill G. As it turns out, this is not actually true as stated, because these paths may end before G and their processing may activate other paths, that might be rightward. Nevertheless, using Lemma 2, we show that either G or the gap H to the right of X_s will be filled.

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. By symmetry, we can assume that X_s is left-oriented, so all paths in $\mathcal{P}_s - \mathcal{P}_{s-1}$ are leftward. 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. 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, $|X_s| \geq \max(|G|, |H|)$. This inequality, the assumption that G and H do not disappear in Q_s, and Lemma 3 imply together that 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. Thus, in stage s, the assumption that G and H do not disappear in Q_s implies that at all steps the paths in $\mathcal{P}_s - \mathcal{R}$ start in B.

Let $\boldsymbol{u}$ be the path whose truncation $\hat{\boldsymbol{u}}$ ends in $f_B - 1$ right after stage s. No transfer path can start in the slots immediately to the left of $f_B - 1$ because they were idle in Q_0. Together with the previous paragraph, this implies that $\boldsymbol{u}$ must be leftward. We can now choose a sequence $\boldsymbol{u}^1, ..., \boldsymbol{u}^p = \boldsymbol{u}$ of transfer paths from $\mathcal{P}_s$ such that $\boldsymbol{u}^1$ is a leftward path starting in X_s (so $\boldsymbol{u}^1$ was in $\mathcal{P}_s - \mathcal{R}$ 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). Similarly, define $\boldsymbol{v}$ to be the rightward path whose truncation ends in the leftmost slot of H and $\boldsymbol{v}^1, ..., \boldsymbol{v}^q = \boldsymbol{v}$ be the similarly defined sequence for $\boldsymbol{v}$. Our goal is to show that there are paths $\boldsymbol{u}^i$ and $\boldsymbol{v}^j$ that cross, which would give us a contradiction.

The following simple observation follows directly from the definition of the truncation process. Note that it 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 $\boldsymbol{t} = (t_0, ..., t_k)$ 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_0^g 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_0^h is leftmost.

Claim 1: We have (i) $u_0^g \geq f_{X_s}$ and (ii) the leftward paths in $\{\boldsymbol{u}^1, ..., \boldsymbol{u}^p\}$ cover the interval $[f_B, u_0^g]$, in the following sense: for each $z \in [f_B, u_0^g]$ there is a leftward path $\boldsymbol{u}^i = (u_0^i, ..., u_{k_i}^i)$ such that $u_{k_i}^i \leq z \leq u_0^i$.

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, 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 1.

Claim 2: 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 is similar to that in Claim 1. 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 is that $e \neq 1$, because X_s is left-oriented. We have two cases. If $e = 2$ then $\boldsymbol{v}^2$ is a rightward path whose truncation after stage $s-1$ ended in $\tau \in X_s$, and in stage s it was replaced in $\mathcal{R}$ by the trivial path $\boldsymbol{v}^1 = (\tau)$ in a step of type (t2). Then $\boldsymbol{v}_0^2 < f_{X_s}$ and (i) holds. If $e > 2$ then $\boldsymbol{v}^2$ is a non-trivial leftward path, so $v_{l_2}^2 < f_{X_s}$. Then (i) follows from Observation 1, by applying it iteratively to paths $\boldsymbol{v}^2, ..., \boldsymbol{v}^e$, all of which except $\boldsymbol{v}^e$ are leftward.

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 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 $X_s \cap H = \emptyset$. If $l_B + 1$, the slot of H adjacent to B, 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 $|X_s \cap B| \geq |G - X_s|$. Therefore G will disappear in Q_s, contradicting our assumption that it did not. □

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 computed by Algorithm* LVG *has at most $2g^* - 1$ gaps.*

Lower bound example. In the full paper we show that for any $k \geq 2$ there is an instance $\mathcal{J}_k$ on which Algorithm LVG finds a schedule with $2k-1$ gaps, while the optimum schedule has $g^* = k$ gaps. Thus our analysis is tight.

Implementation. In the full paper we show that Algorithm LVG can be implemented in time $O(ng^* \log n)$ and memory $O(n)$, where g^* is the optimum number of gaps. The idea is this: At each step we remove the forbidden regions from the timeline, maintaining the invariant that all release times are different and that all deadlines are different. This can be done in time $O(n \log n)$ per step. With this invariant, the maximum forbidden region has the form $[r_i + 1, d_j - 1]$ for some jobs i, j. We then show how to find such i, j in linear time. Since we have $O(g^*)$ steps, the overall running time will be $O(ng^* \log n)$.

5 Final Comments

Among the remaining open questions, 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 in this direction.

Our 2-approximation result for Algorithm LVG remains valid for scheduling jobs with arbitrary processing times when 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].

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.