ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Journal of Discrete Algorithms 4 (2006) 339–352

JOURNAL OF DISCRETE ALGORITHMS

www.elsevier.com/locate/jda

Online weighted flow time and deadline scheduling

Luca Becchetti [a,*,1], Stefano Leonardi [a,1], Alberto Marchetti-Spaccamela [a,1], Kirk Pruhs [b,2]

[a] *Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Italy*
[b] *Computer Science Department, University of Pittsburgh, USA*

Available online 18 January 2006

Abstract

In this paper we study some aspects of weighted flow time. We first show that the online algorithm Highest Density First is an O(1)-speed O(1)-approximation algorithm for $P|r_i, pmtn|\sum w_i F_i$. We then consider a related Deadline Scheduling Problem that involves minimizing the weight of the jobs unfinished by some unknown deadline D on a uniprocessor. We show that any c-competitive online algorithm for weighted flow time must also be c-competitive for deadline scheduling. We then give an O(1)-competitive algorithm for deadline scheduling.

Keywords: Scheduling; Weighted flow time; On-line

1. Introduction

We consider several aspects of online scheduling to minimize total weighted flow time. In this problem a sequence of jobs has to be processed on a set of m identical machines. The ith job has a release time r_i, a processing time or length x_i and a non-negative weight w_i.

* Corresponding author.
E-mail addresses: becchett@dis.uniroma1.it (L. Becchetti), leon@dis.uniroma1.it (S. Leonardi), alberto@dis.uniroma1.it (A. Marchetti-Spaccamela), kirk@cs.pitt.edu (K. Pruhs).

[1] Partially supported by EU Integrated Project DELIS and by EU IST Project AEOLUS.
[2] Supported in part by NSF grants CCR-0098752, ANI-0123705, CNS-0325353, and CCF-0448196, and by a grant from the US Airforce.

1570-8667/$ – see front matter
doi:10.1016/j.jda.2005.12.001

At any time, only one processor may be running any particular job. Processors may preempt one job to run another. The ith job is completed after it has been run for x_i time units. The flow time F_i of the ith job is the difference between its completion time and its release date. The objective function to minimize is the weighted flow time $\sum w_i \cdot F_i$. Following the standard three field notation for scheduling problems, we denote this problem by $P|r_i, pmtn|\sum w_i \cdot F_i$.

By far the most commonly used measure of system performance is average flow time, or equivalently average user perceived latency. Weighted flow time is an obvious generalization that models the situation where different jobs may have different priorities. Another motivation for considering weighted flow time is that it is a special case of broadcast scheduling (see for example [1]).

A reasonable amount is known about minimizing average flow time online [8–10], however, minimizing weighted flow time online (or offline for that matter) is not yet well understood. The uniprocessor offline problem $1|r_i, pmtn|\sum w_i \cdot F_i$ is known to be NP-hard [6]. But there is no known offline polynomial-time constant approximation algorithm. The one previous positive result for weighted flow time used resource augmentation analysis. Resource augmentation analysis was proposed [9] as a method for analyzing scheduling problems that are hard to approximate. Using the notation and terminology of [11], an *s-speed c-approximation* algorithm A has the property that the value of the objective function of the schedule that A produces with processors of speed $s \geqslant 1$ is at most c times the optimal value of the objective function for speed 1 processors. An algorithm is c-competitive if it is a 1-speed c-approximation algorithm. It was shown in [11] that an LP-based online algorithm, Preemptively-Schedule-Halves-by-$\bar{M}_j$, is an O(1)-speed O(1)-approximation algorithm.

The first contribution of this paper, covered in Section 2, is that we simplify the resource augmentation analysis of $1|r_i, pmtn|\sum w_i F_i$ given in [11], while also extending the analysis to multiprocessors. Namely, we show that the polynomial time, online algorithm Highest Density First (HDF), which always runs the job that maximizes its weight divided by its length, is an online O(1)-speed O(1)-approximation algorithm. While this analysis of HDF is not stated in [11], upon reflection one can see that all the insights necessary to assert that HDF is an O(1)-speed O(1)-approximation algorithm on a single processor are inherent in their analysis of their proposed algorithm Preemptively-Schedule-Halves-by-$\bar{M}_j$. Besides making this result explicit, our proof has the advantages that (1) our analysis is from first principles and does not require understanding of a rather complicated LP lower bound, and (2) our analysis also easily extends to the multiprocessor problem $P|r_i, pmtn|\sum w_i F_i$, while the result in [11] apparently does not.

In Section 3 we consider competitive analysis of $1|r_i, pmtn|\sum w_i F_i$. We first show that every c-competitive online algorithm A has to be *locally c-competitive*, that is, at every time t, the overall weight of the jobs that A has not finished by time t can be at most c times the minimum possible weight of the unfinished jobs at time t. The requirement of local competitiveness suggests that we consider what we call the Deadline Scheduling Problem (DSP).

The input to DSP consists of n jobs, with each job i having a length x_i and a nonnegative weight w_i. Both the length of a job and its weight are revealed to the algorithm when the jobs are released at time 0. The goal is to construct a schedule (linear order)

where the overall weight of the jobs unfinished by some initially unknown deadline D is minimized. In the standard three field scheduling notation, one might denote this problem as $1|d_i = D|\sum w_i \cdot U_i$. The requirement of local competitiveness means that any c-competitive online algorithm for $1|r_i, pmtn|\sum w_i F_i$ must be c-competitive for DSP. The DSP problem can be seen as a dual to the deadline scheduling problem considered in [7]. The setting considered in [7] was the same as DSP, except that the goal was to maximize the jobs completed before the deadline D on multiple machines. In [7] it is shown that no constant competitive algorithm for the maximization problem exists. The second main contribution of this paper is an online O(1)-competitive algorithm for DSP on one machine.

2. Resource augmentation analysis of $P|r_i, pmtn|\sum w_i F_i$

In this section, we adopt the following notation. We use $A(s)$ to denote the schedule produced by algorithm A with a speed s processor, and abuse notation slightly by using A to denote $A(1)$. Let $U^A(t)$ be the jobs that have been released before time t, but not finished by algorithm A by time t. We define the *density* of a job j as the ratio $\mu_j = w_j/p_j$. We use $y_j^A(t)$ to denote the remaining unprocessed length of job j at time t according to the schedule produced by algorithm A. Similarly, we use $p_j^A(t) = p_j - y_j^A(t)$ to denote the processed length of job j before time t. So from time t, algorithm A, with a speed s processor, could finish j in $y_j^A(t)/s$ time units. We let $w_j^A(t) = y_j^A(t)\frac{w_j}{p_j}$ be the fractional remaining weight of job j at time t in A's schedule. We denote by $W^A(t) = \sum_{j\in U^A(t)} w_j$ and by $F^A(t) = \sum_{j\in U^A(t)} w_j^A(t)$ respectively the overall weight and the overall fractional weight of jobs that A has not completed by time t.

It is well known that the total weighted flow time of a schedule A is equal to $\int W^A(t)\,\mathrm{d}t$; to see this note that during the infinitesimal time $\mathrm{d}t$, the weighted flow time increases by $W^A(t)$. Hence to prove that an algorithm A is a c-approximation algorithm for weighted flow time it is sufficient to show that A is locally c-competitive, this meaning that, at any time t, $W^A(t) \leqslant cW^{\mathrm{OPT}}(t)$.

Recall that, given m processors, the algorithm Highest Density First (HDF) always runs the up to m densest available jobs. We now prove, via a simple exchange argument, that giving HDF a faster processor doesn't decrease the time that HDF runs any unfinished job by any time.

Lemma 1. *For any number of processors, for any job instance, for any time t, and for any $j \in U^{\mathrm{HDF}(1+\varepsilon)}(t)$, it is the case that $p_j^{\mathrm{HDF}(1+\varepsilon)}(t) \geqslant (1+\varepsilon)p_j^{\mathrm{HDF}(1)}(t)$, or equivalently, before time t it is the case that* HDF$(1+\varepsilon)$ *has run job j for more time than* HDF(1).

Proof. Assume to reach a contradiction that there is a time t and a job j where this does not hold. Further, assume that t is the first such time where this fails to hold. Note that obviously $t > 0$. HDF(1) must be running j at time t by the definition of t. Also by our assumptions, HDF$(1+\varepsilon)$ can not have finished j before time t, and does not run j at time t. Hence, HDF$(1+\varepsilon)$ must be running some job h at time t, that HDF(1) is not running at time t. HDF$(1+\varepsilon)$ could not be idling any processor at time t since j was available to be

run, but was not selected. Since HDF$(1+\varepsilon)$ is running job h instead of job j, job h must be denser than job j. Then the only reason that HDF(1) would not be running h is if it finished h before time t. But this contradicts our assumption of the minimality of t. □

We now prove that $F^{\text{HDF}}(t)$ is a lower bound to $W^{\text{OPT}}(t)$ in the uniprocessor setting.

Lemma 2. *For any instance of the problem* $1|r_i, pmtn|\sum w_i F_i$*, and any time* t*, it is the case that* $F^{\text{HDF}}(t) \leqslant F^{\text{OPT}}(t) \leqslant W^{\text{OPT}}(t)$.

Proof. The second inequality is clearly true. To prove $F^{\text{HDF}}(t) \leqslant F^{\text{OPT}}(t)$ we use a simple exchange argument. To reach a contradiction, let OPT be an optimal schedule that agrees with the schedule HDF(1) the longest. That is, assume that OPT and HDF(1) schedule the same job at every time strictly before some time t, and that no other optimal schedule agrees with HDF(1) for a longer initial period. Let i be the job that HDF(1) is running at time t and let j be the job that OPT is running at time t. The portion of job i that HDF(1) is running at time t must be run later by OPT at some time, say s. Now create a new schedule OPT′ from OPT by swapping the jobs run at time s and time t, that is OPT′ runs job i at time t, and job j at time s. Note that job i has been released by time t since it is run at that time in the schedule HDF(1). By the definition of HDF, it must be the case that $\mu_i \geqslant \mu_j$. Hence, since we are considering the fractional remaining weight of the jobs, it follows that for all times u, $F^{\text{OPT}'}(u) \leqslant F^{\text{OPT}}(u)$, and OPT′ is also an optimal solution. This is a contradiction to the assumption that OPT was the optimal solution that agreed with HDF(1) the longest. □

Observe that the above lemma is not true for parallel machines since HDF may underutilize some of the machines with respect to the optimum. We now establish that HDF$(1+\varepsilon)$ is locally $(1+\frac{1}{\varepsilon})$-competitive against HDF in the uniprocessor setting.

Lemma 3. *For any* $\varepsilon > 0$*, for any job instance, for any time* t*, and for any job* j*, it is the case that* $W^{\text{HDF}(1+\varepsilon)}(t) \leqslant (1+\frac{1}{\varepsilon})F^{\text{HDF}}(t)$.

Proof. It follows from Lemma 1 that the only way that $W^{\text{HDF}(1+\varepsilon)}(t)$ can be larger than $F^{\text{HDF}}(t)$ is due to the contributions of jobs that were run but not completed by both HDF$(1+\varepsilon)$ and HDF. Let j be such a job. Since HDF cannot have processed j for more than $p_j^{\text{HDF}(1+\varepsilon)}(t)/(1+\varepsilon)$ time units before time t,

$$w_j^{\text{HDF}(1)}(t) \geqslant \frac{1}{p_j}\left(p_j - \frac{p_j^{\text{HDF}(1+\varepsilon)}(t)}{1+\varepsilon}\right)w_j.$$

The ratio $w_j/w_j^{\text{HDF}(1)}(t)$ is then at most

$$p_j \Big/ \left(p_j - \frac{p_j^{\text{HDF}(1+\varepsilon)}(t)}{1+\varepsilon}\right).$$

When $p_j^{\mathrm{HDF}(1+\varepsilon)}(t) = p_j$, which is where this ratio is maximized, this ratio is $1 + 1/\varepsilon$. Thus the result follows. □

Theorem 4 then follows immediately from Lemmas 2 and 3.

Theorem 4. *Highest Density First* (HDF) *is a* $(1+\varepsilon)$*-speed* $(1+\frac{1}{\varepsilon})$*-approximation algorithm for the problem* $1|r_i, pmtn|\sum w_i F_i$.

Theorem 5. *Highest Density First* (HDF) *is a* $(2+2\varepsilon)$*-speed* $(1+\varepsilon)$*-approximation online algorithm for* $P|r_i, pmtn|\sum w_i F_i$.

Proof. The proof follows the same lines as the proof of Theorem 4, and uses some techniques from the analysis of SRPT in [11]. We show that each $F^{\mathrm{FHDF}(2)}(t)$ is a lower bound to $W^{\mathrm{OPT}(1)}(t)$. The following property of busy algorithms A (algorithms that don't unnecessary idle a processor) is shown in [11]:

$$\forall t \sum_j p_j^{A(2)}(t) \geqslant \sum_j p_j^{\mathrm{OPT}(1)}(t).$$

That is, at all times it is the case that A, with a 2 speed processor, has done at least as much work as any unit speed schedule. Let I_t be the set of jobs finished by OPT by time t. Hence, if the input was only I_t, HDF(2) would have finished all of I_t by time t since HDF is a busy algorithm. One can show, via a simple exchange argument, that HDF has the property that adding more jobs to the instance never decreases the fractional weight that it has processed by time t, that is $\sum_j w_j \frac{p_j^{\mathrm{HDF}}(t)}{p_j}$ is monotone non-decreasing as more jobs are added to the job set. Therefore, $F^{\mathrm{FHDF}(2)}(t) \leqslant W^{\mathrm{OPT}(1)}(t)$. We then conclude by noting that HDF$(2+2\varepsilon)$ is locally $(1+\frac{1}{\varepsilon})$-competitive against FHDF(2) by Lemma 2. □

3. Deadline scheduling problem

In this section we first show that a c-competitive algorithm for minimizing weighted flow time must be also locally *c-competitive*. We then introduce the Deadline Scheduling Problem (DSP), and give an O(1)-competitive algorithm for DSP.

Theorem 6. *Every c-competitive deterministic online algorithm A for* $1|r_i, pmtn|\sum w_i F_i$ *must be locally c-competitive.*

Proof. Let A be a c-competitive algorithm. Assume to reach a contradiction that there is an instance I, and a rational time t such that $W^A(t) > c \cdot W^{\mathrm{OPT}}(t)$. Intuitively, we construct a new instance I' on which A is not c-competitive by bringing in a sequence of arbitrarily short and dense jobs starting from time t. First remove any jobs from I that are released after time t. Let ℓ be least common multiple of $\{y_j^A(t) \mid j \in U^A(t)\} \cup \{y_j^{\mathrm{OPT}}(t) \mid j \in U^{\mathrm{OPT}}(t)\}$, that is, the unexecuted portions of the jobs unfinished by A at time t union the unexecuted portions of the jobs in OPT at time t. Let w be sufficiently small when compared with

$W^{\mathrm{OPT}}(t)$. Let ε_w be an arbitrarily small length such that ℓ/ε_w is an integer, and for all $j \in U^A(t)$ it is the case that $\frac{w_j}{y_j(t)} \leqslant \frac{w}{\varepsilon_w}$, that is, $\frac{w}{\varepsilon_w}$ is greater than the density of any jobs portions in $U^A(t)$. Between time t, and some large time T, we bring in one job of length ε_w, and weight w, every ε_w time units.

It is easy to see the optimal strategy for A is to always run these new jobs after time t. By making T sufficiently large, the relative contribution to the weighted flow time of jobs waiting before time t, becomes negligible. The contribution to the weighted flow time for the jobs released after time t is $w \cdot \varepsilon_w \frac{(T-t)}{\varepsilon_w} = w \cdot (T-t)$ since each of these $\frac{T-t}{\varepsilon_w}$ jobs have flow time ε_w. Thus the competitive ratio can be approximated arbitrarily well by $\frac{W^A(t)\cdot T + w\cdot T}{w^{\mathrm{OPT}}(t)\cdot T + w\cdot T}$. Since w is small when compared with $W^{\mathrm{OPT}}(t)$, The competitive ratio is well approximated by $\frac{W^A(t)}{w^{\mathrm{OPT}}(t)}$, which is bigger than c by assumption. This contradicts the assumption that A was a c-competitive algorithm. □

Theorem 6 motivates our consideration of the Deadline Scheduling Problem (DSP). Recall that in DSP n jobs are released at time 0, and the goal is to minimize the weight of the jobs not completed by some a priori unknown deadline. We now give an example to show that DSP is more subtle than one might at first think. Consider the instance of DSP shown in Fig. 1.

Two natural schedules are: (1) first run the low density job followed by the k^3 high density jobs, and (2) first run the k^3 high density jobs followed by the low density job. It is easy to see that the first algorithm is not constant competitive at time $t = k^3$, and that the second algorithm is not constant competitive at time $t = k^3 + k^2 - 1$. In fact, what the online algorithm should do in this instance is to first run $k^3 - \Theta(k)$ of the high density jobs, then run the low density job, and then finish with the remaining $\Theta(k)$ high density jobs. A little reflection reveals that this schedule is O(1)-competitive for this instance of DSP. This instance demonstrates that the scheduler has to balance between delaying low density jobs, and delaying low weight jobs.

An alternative way to look at DSP is by reversing the time axis. This makes DSP equivalent to the problem minimizing the overall weight of jobs that have been started by some unknown deadline. This problem is more formally stated below.

NDSP problem statement. The input consists of n jobs released at time 0, with known processing times and weights, plus an a priori unknown deadline t. The goal is to find a schedule that minimizes the total weight of jobs that have been *started* before time t, including the job running at time t. Intuitively, time t in this schedule corresponds to time $\sum p_i - t$ in the original schedule.

Number of jobs	Weight	Length	Density = weight/length
1	k	k^2	$1/k$
k^3	1	1	1

Fig. 1. A non-trivial input for DSP. $k \gg 1$.

In this section we adopt following notation. We use $A(t)$ to refer to the schedule created by algorithm A up through time t. We use $W^A(t)$ to refer to the weight of the jobs started by algorithm A before time t. We use $\textsc{Opt}(t)$ to denote the schedule that the minimum weight collection of jobs with length at least t. Note that in general there is not a single schedule that optimizes all t's simultaneously, that is, the schedules $\textsc{Opt}(t)$ and $\textsc{Opt}(t')$ may be very different. For a fixed instance of NDSP, the competitive ratio of an algorithm A is the maximum over all $t \in [0, \sum p_j]$ of $W^A(t)/W^{\textsc{Opt}}(t)$. Since the deadline t is fixed throughout this section, we will generally drop it from our notation.

We first describe a polynomial time offline algorithm Off for NDSP, and then show that Off is 3-competitive with respect to Opt. We then propose a polynomial time online algorithm called R for NDSP, and show that R is 8-competitive with respect to Off. Hence, we can conclude that R is a 24-competitive algorithm. Observe that NDSP is simply the minimum knapsack problem, for which a FPTAS exists [12]. Our purpose for defining the algorithm Off is that its solutions will be structurally similar to the solutions produced by our online algorithm R. Both R and Off at various points in their execution must select the lowest density job from some collection; Ties may be broken arbitrarily, but in order to simplify our analysis, we assume that Off and R break ties similarly (say by the original numbering of the jobs).

Algorithm Off intuitively tries to schedule the least dense jobs until time t. The algorithm Off must then be concerned with the possibility that the job that it has scheduled at time t, say j, is of too high a weight. So Off recursively tries to construct a lower aggregate weight schedule to replace j, from those unscheduled jobs with weight at most $w_j/2$. Observe that algorithm Off runs in polynomial time.

Description of Algorithm Off. Algorithm Off computes a set of busy schedules $\textsc{Off}_1, \ldots, \textsc{Off}_u$. The schedule produced by Off is then the schedule among $\textsc{Off}_1, \ldots, \textsc{Off}_u$ with minimum weight. When Off is started, $\mathcal{J}$ is the collection of all jobs, and in general $\mathcal{J}$ contains the jobs that might be scheduled in future. The variable s is initialized to $s = 0$, and in general indicates is the time that the last job in the current schedule ends. Variable h is initially set to 1, and in general is the index for the current schedule $\textsc{Off}_h$ that Off is constructing.

1. Let j be the least dense job in $\mathcal{J}$.
2. Remove j from $\mathcal{J}$.
3. If $s + p_j < t$ then
   (a) Append j to the end of the schedule $\textsc{Off}_h$.
   (b) Let $s = s + p_j$.
   (c) Go to step 1.
4. Else if $s + p_j \geqslant t$ then
   (a) Let $\textsc{Off}_{h+1} = \textsc{Off}_h$. **Comment**: Initialize $\textsc{Off}_{h+1}$.
   (b) Append j to the end of the schedule $\textsc{Off}_h$. **Comment**: Schedule $\textsc{Off}_h$ is now completed.
   (c) Let $h = h + 1$.
   (d) Remove from $\mathcal{J}$ any job with weight greater than $w_j/2$.
   (e) If the total length of the jobs in $\mathcal{J}$ is at least $t - s$ then go to step 1.

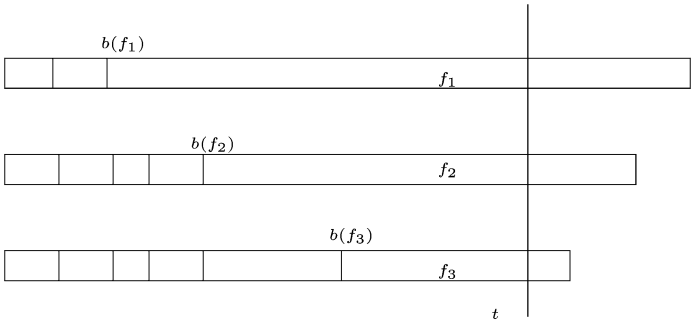

Fig. 2. A depiction of our notation with $k = 3$.

If a job j is scheduled in step 3a then we say that j is a *non-crossing job*. If a job j is scheduled in step 4b then we say that j is a *crossing job*, and that afterwards a Schedule OFF_h *was closed on* j.

Theorem 7. $W^{\text{OFF}} \leqslant 3W^{\text{OPT}}$.

Proof. The jobs in the schedule OFF are ordered in increasing order of density. Assume that the jobs in OPT are ordered by increasing density as well. Let us introduce some notation. Let $f_1, \ldots, f_u$ be the sequence of jobs that close schedules $\text{OFF}_1, \ldots, \text{OFF}_u$. Let $b(f_h)$ be the time at which j_h is started in OFF_h. This notation is illustrated in Fig. 2.

We break the proof into two cases. In the first case we assume that there is some schedule OFF_h that runs at some time instant a job of density higher than the job run at the same time by OPT. Denote by x the earliest such time instant. Since jobs scheduled by OPT are ordered by increasing density, we assume w.l.o.g. that OFF_h starts such a job at time x. Denote by j the job started by OFF_h at time x and by i the job run by OPT at time x. Clearly, $\mu_j > \mu_i$. It is easy to see that OFF_1 is always running a job with density at most the density of the job that OPT is running at that time. Hence, it must be the case that $h > 1$. Since $i \neq j$, there must be some job g run by OPT before i, that OFF_h does not run before j. Note that it may be the case that $g = i$. Then the only possible reason that OFF didn't select g instead of j, is that g was eliminated in step 4d, before OFF closed a schedule on some f_d, $d \leqslant h - 1$. Hence, $w_g \geqslant w_{f_{h-1}}/2$, and $w_{f_{h-1}} \leqslant 2\text{OPT}$ since w_g is scheduled in OPT. By the minimality of x, it must be the case that, at all times before time $b(f_{h-1})$, the density of the job that OFF_{h-1} is running is at most the density of the job that OPT is running. Hence, the aggregate weight of the jobs in OFF_{h-1}, exclusive of f_{h-1}, is at most OPT. We can then conclude that $\text{OFF}_{h-1} \leqslant 3\text{OPT}$, and hence $\text{OFF} \leqslant 3\text{OPT}$ by the definition of OFF.

In the second case assume that, at all times in all the schedules OFF_h, it is the case that the job that OFF_h is running at this time is at most the density of the job that OPT is running at this time. Hence, the weight of the jobs in the last schedule OFF_u, exclusive of f_u, is at most OPT. Consider the time that OFF adds job f_u to schedule OFF_u. The fact that OFF_u is the last schedule produced by OFF means that the total lengths of jobs in OFF_u, union the jobs that are $\mathcal{J}$ after the subsequent step 4d is strictly less than t. The

jobs in $\mathcal{J}$ at this time, are those jobs with weight at most $w_{f_u}/2$ that were not previously scheduled in OFF_u. Hence, the total length of the jobs in the original input with weight at most $w_{f_u}/2$ is strictly less than t. Therefore at least one job in OPT has weight at least $w_{f_u}/2$. Hence, $w_{f_u} \leqslant 2\text{OPT}$. Therefore, once again we can conclude that $\text{OFF}_u \leqslant 3\text{OPT}$, and that $\text{OFF} \leqslant 3\text{OPT}$. □

We now turn to describing the algorithm R. R is an on-line algorithm that produces a complete schedule of $\mathcal{J}$ without knowing the deadline t. The cost incurred by R will be the total weight of jobs started by time t. Intuitively, the algorithm R first considers the lowest density job, say j. However, R cannot immediately schedule j because it may have high weight, which might mean that R wouldn't be competitive if the deadline occurred while R was executing j. To protect against this possibility, R tries to recursively schedule jobs with weight at most $w_j/2$, before it schedules j, until the aggregate weight of those jobs scheduled before j exceeds w_j. We now give a more formal description of the algorithm R.

Description of Algorithm R. The algorithm R takes as input a collection $\mathcal{J}$ of jobs. Initially, $\mathcal{J}$ is the collection of all jobs, and in general, $\mathcal{J}$ will be those jobs not yet scheduled by previous calls to R. The algorithm R is described below:
While $\mathcal{J} \neq \emptyset$

1. Select a $j \in \mathcal{J}$ of minimum density. Remove j from $\mathcal{J}$.
2. (a) Initialize a collection $\mathcal{I}_j$ to the empty set. $\mathcal{I}_j$ will be the jobs scheduled during step 2.
   (b) Let $\mathcal{J}_j$ be those jobs in $\mathcal{J}$ with weight at most $w_j/2$.
   (c) If $\mathcal{J}_j$ is empty then go to step 3.
   (d) Recursively apply R to $\mathcal{J}_j$. Add the jobs scheduled in this recursive call to $\mathcal{I}_j$. Remove the jobs scheduled in this recursive call from $\mathcal{J}$.
   (e) If the aggregate weight of the jobs in $\mathcal{I}_j$ exceeds w_j then go to step 3, else go to step 2c.
3. Schedule j after all jobs of $\mathcal{I}_j$.

End While

We say a job j was *picked* if it was selected in step 1. If, after j is picked, a recursive call to R is made with input $\mathcal{J}_j$, we say that R *recurses* on j. Notice that the algorithm R also runs in polynomial time.

We now explain how to think of the schedule R as a forest, in the graph theoretic sense. The forest has a vertex for every job in $\mathcal{J}$. For this reason we will henceforth use the terms job and vertex interchangeably. The jobs in the subtree $T(j)$ rooted at job j are j and those jobs scheduled in the loop on step 2 after j was selected in step 1. The ordering of the children is by non-decreasing density, that is, the order in which R selects and schedules these jobs. The roots of the forest are those vertices selected in step 1 at the top level of the execution of the algorithm. Similarly, the ordering of the roots of the forest is also by non-decreasing density, and in the order in which R selects and schedules these jobs. We

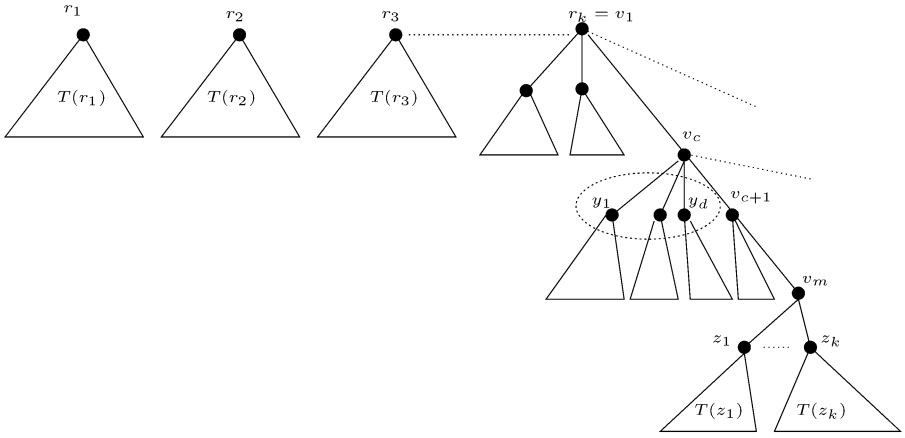

Fig. 3. Forest structure of R's schedule.

adopt the convention that the orderings of trees in the forest, and children of a vertex, are from left to right.

Before proceeding we need to introduce some notation. Given a descendant y of a vertex x in some tree, we denote by $P(x, y)$ the path from x to y in T, x and y inclusive. If y is not a descendant of x, then $P(x, y)$ is the empty path. We denote by $r_1, \ldots, r_k$ the roots of the trees in R that contain at least one job that R started before time t. Let $v_1, \ldots, v_m$ be the vertices in the path $P(r_k, v_m)$, where $v_1 = r_k$, and where v_m is the job that R is running at time t. Finally, denote by $W(S)$ the total weight of the jobs in a set S. See Fig. 3 for an illustration of the forest structure of R and some of our definitions.

The following facts can easily be seen to follow from the definition of R. The post-order of the jobs within each tree is also the order that these jobs are scheduled by R. All jobs in a tree $T(r_h)$, $h < k$, are completed by R before time t. All jobs in a tree $T(r_h)$, $h > k$, are not started by R before time t. The jobs of $T(r_k)$ that are started by R before time t are: (i) the jobs in $T(v_m)$, and (ii) the jobs in $T(\ell)$, for each left sibling ℓ of a job on $P(v_2, v_m)$. All the jobs in the path $P(r_k, v_{m-1})$ are not started by R before time t. All the jobs in a $T(\ell)$, ℓ being a right sibling of a job on the path $P(v_2, v_m)$, are not started by R before time t.

The following lemma shows that the aggregate weight of the vertices in a subtree is not too large.

Lemma 8. *For any job x in* R, $W(T(x)) \leqslant 4w_x$.

Proof. The proof is by induction on the height of $T(x)$. The base case (i.e. x is a leaf) is straightforward. Let y be x's rightmost child. The aggregate weight of the subtrees rooted at the left siblings of y is at most w_x, or R would not have picked y in tree $T(x)$. Since R removes jobs of weight greater than $w_x/2$ in step 2b, we know that $w_y \leqslant w_x/2$. By induction $W(T(y)) \leqslant 4w_y$, and therefore $W(T(y)) \leqslant 2w_x$. Hence, by adding the weight of the subtrees rooted at x's children to the weight of x, we get that $W(T(x)) \leqslant w_x + w_x + 2w_x = 4w_x$. □

The following lemmas illuminate the structural similarities between the schedules R and OFF. The proof of Lemma 9 should be obvious from the description of OFF. We will slightly abuse terminology by referring to $r_1, \ldots, r_{k-1}$ as the left siblings of r_k.

Lemma 9. *If* OFF $\neq$ OFF$_c$*, for all* $c = 1, \ldots, b$*, then all non-crossing jobs in each* OFF$_c$*, for all* $c = 1, \ldots, b$*, are in* OFF.

Lemma 10. *Assume* OFF *schedules no job on* $P(v_1, v_b)$*,* $b \in \{1, \ldots, m\}$*. Then for every* $c = 1, \ldots, b$ *it is the case that*:

- v_c *is the crossing job that closes* OFF$_c$*,*
- OFF$_c$ *contains all the left siblings of each* v_l*,* $l = 1, \ldots, c$*, and*
- *all the jobs in* OFF$_c$*, other than* v_c*, are also in subtrees of* R *that are rooted at some left sibling of a* v_l*,* $l = 1, \ldots, c$*.*

Moreover, if $b = m$ *then* OFF *schedules all the children of* v_m*.*

Proof. The proof is by induction on c. First consider the base case $c = 1$. The jobs with density less than μ_{v_1} are contained in trees $T(r_1), \ldots, T(r_{k-1})$ in R. Since time t occurred when R was scheduling a job in $T(v_1)$, the overall length of jobs of density strictly less than μ_{v_1} is less than t. Hence, OFF scheduled v_1 in OFF$_1$. By Lemma 9, the reason that OFF does not contain v_1 must be because v_1 is the crossing job that closes OFF$_1$. Then OFF$_1$ must also contain all the root vertices $r_1, \ldots, r_{k-1}$ as non-crossing jobs. Further all jobs in OFF$_1$, other than v_1, must be contained in $T(r_1), \ldots, T(r_{k-1})$, since all other jobs have densities higher than μ_{v_1}.

For the inductive hypothesis, we assume that the claim is true up to vertex v_c that closes solution OFF$_c$. We then prove the claim for vertex v_{c+1}. Consider the time right before OFF is going to schedule a job in OFF$_{c+1}$ at time $b(f_{c+1})$, and let $\mathcal{K}$ be the jobs in $\mathcal{J}$ at this time that have density less than $\mu_{v_{c+1}}$. The jobs in $\mathcal{K}$ are those jobs with density in the range $(\mu_{v_c}, \mu_{v_{c+1}})$, and weight at most $w_{v_c}/2$. Note that by the properties of R, all the jobs in $\mathcal{K}$ are contained in subtrees of R rooted at some left sibling of a v_l, $l \in \{1, \ldots, c+1\}$. Since t did not occur before v_{c+1} began execution in R, $b(f_c)$ plus the total length of the jobs in $\mathcal{K}$ is less than t. Therefore v_{c+1} will eventually be selected to be included in OFF$_{c+1}$, after all of v_{c+1} left siblings were selected to be in OFF$_{c+1}$. But by Lemma 9, since jobs $v_1, \ldots, v_{c+1}$ are not scheduled by OFF, v_{c+1} must be the crossing job that closes OFF$_{c+1}$.

Now assume that $b = m$, and that OFF scheduled none of the jobs in $P(v_1, v_m)$. We wish to establish that OFF scheduled all of the children of v_m in R. Let $z_1, \ldots, z_k = v_{m+1}$ be the children of v_m in R, ordered from left to right. Consider the time right before OFF is going to schedule a job in OFF$_{m+1}$ at time $b(f_{m+1})$, and let $\mathcal{K}$ be the jobs in $\mathcal{J}$ at this time that have density less than or equal $\mu_{v_{m+1}}$. The jobs in $\mathcal{K}$ are those jobs with density in the range $(\mu_{m_c}, \mu_{v_{m+1}}]$, and weight at most $w_{v_m}/2$. Note that by the properties of R, all the jobs in $\mathcal{K}$ are contained in subtrees of R rooted at some left sibling of a v_l, $l \in \{1, \ldots, m+1\}$. Since t did not occur while v_{m+1} was executing in R, $b(f_m)$ plus the total length of the jobs in $\mathcal{K}$ is less than t. Therefore v_{m+1} will eventually be selected to be a non-crossing job in OFF$_{m+1}$, after all of v_{m+1} left siblings were selected to be non-crossing jobs in OFF$_{m+1}$.

Then by Lemma 9, and the assumption that OFF $\neq$ OFF$_c$, for any $c = 1, \ldots, m$, it must be the case that $z_1, \ldots, z_k$ are scheduled by OFF. □

Let us discuss the consequences of the previous two lemmas. Assume that OFF schedules no job on the path $P(v_1, v_b)$. Then OFF $\neq$ OFF$_c$ for any $c = 1, \ldots, b$. Hence, all non-crossing jobs in each OFF$_c$, $c = 1, \ldots, b$, must be in OFF. Further, each left sibling of a v_l in $P(v_1, v_b)$ is in OFF.

Theorem 11. $R(t) \leqslant 8\text{OFF}$.

Proof. We will charge all the weight of the jobs started by R before time t to the jobs scheduled by OFF in a way that no job in Off is charged for more than 8 times its weight.

Note that by Lemma 10 OFF completes the roots $r_1, \ldots, r_{k-1}$ by time t. We then charge $W(T(r_i))$ to each r_i scheduled by OFF, $1 \leqslant i \leqslant k-1$, and therefore by Lemma 8 we charge each r_i at most four times its weight.

We are left to charge the weight of the jobs in $T(r_k)$ started by R before time t. We consider three cases:

(i) OFF schedules job $r_k = v_1$;
(ii) OFF does not schedule job $r_k = v_1$, but OFF schedules a job on path $P(v_1, v_m)$;
(iii) OFF schedules no job on path $P(v_1, v_m)$.

In case (i), we charge $W(T(r_k))$ to r_k. This way r_k is charged at most four times its weight. Now consider case (ii). Let v_b be the job on $P(v_1, v_m)$ closest to v_1 scheduled by OFF. We charge to vertex v_b the weight of all the jobs in $T(v_b)$ started by R before t. By Lemma 8, v_b is charged no more than four times its own weight. By Lemma 10, OFF also scheduled all of the left siblings ℓ of any vertex $v_c \in P(v_1, v_b)$; We charge $w(T(\ell))$ to each such ℓ. By Lemma 8, each such ℓ is charged no more than four times its own weight. We have now accounted for all jobs started by R before time t.

For the remainder of the proof we consider case (iii). By Lemma 10, OFF schedules all of the left siblings ℓ of the jobs on $P(v_1, v_m)$ in R. We charge $w(T(\ell))$ to each such ℓ. By Lemma 8, each such ℓ is charged no more than four times its own weight. By Lemma 10, OFF schedules all of the children of v_m in R.

We now argue that it must be the case that $w(T(v_m)) - w_{v_m} \geqslant w_{v_m}$. Assume to reach a contradiction that $w(T(v_m)) - w_m < w_{v_m}$; That is, assume that the recursion of R on v_m was completed because $w(\mathcal{J}_t)$ was less than w_{v_m} at step 2e. By Lemma 10, v_m was the crossing job in OFF$_m$. Consider the time right before OFF is going to schedule a job in OFF$_{m+1}$ at time $b(f_{m+1})$, and let $\mathcal{K}$ be the jobs in $\mathcal{J}$ at this time. The jobs in $\mathcal{K}$ when are the jobs with density greater than μ_{v_m}, and weight at most $w_{v_m}/2$. The jobs in $\mathcal{J}_{v_m}$ when R recursed on v_m were those jobs with density greater than μ_{v_m} and weight at most $w_{v_m}/2$, that R had not scheduled earlier. Hence, the fact that R had not yet reached time t after scheduling the rightmost child of v_m, implies that $b(f_m)$ plus the aggregate length of the jobs in $\mathcal{K}$ is less than t. Hence, OFF would not have constructed a schedule OFF$_{m+1}$. Hence by Lemma 10, OFF would have to schedule some job on the path $P(v_1, v_m)$, which is a contradiction.

So we may now assume that the recursion of R on v_m completed because $w(T(v_m)) - w_{v_m} \geqslant w_{v_m}$. Denote by $z_1, \ldots, z_k$ all the children of v_m. By Lemma 10 the jobs $z_1, \ldots, z_k$ are all scheduled OFF. We charge $w(T(z_i))$, $1 \leqslant i \leqslant k$, to each such z_i for the weight of the jobs in z_i's subtree. By Lemma 8, each such z_i is charged no more than four times its own weight for this. We charge $w(T(z_i))$, $1 \leqslant i \leqslant k$, to each such z_i for the weight of v_m. By the assumption of this subcase, this is sufficient to fully charge away the weight of v_m. Hence, each such z_i is charged at most eight times its weight. We have now accounted for all jobs started by R before time t. □

Theorem 12 immediately follows from Theorems 7 and 11 by reversing the schedule given by R.

Theorem 12. *There exists a* 24*-competitive algorithm for DSP.*

4. Conclusions

Subsequent to our investigations, there have been several related results. A polynomial time offline algorithm for $1|r_i, pmtn| \sum w_i \cdot F_i$ was proposed, and shown to be polylog-competitive, in [5]. The algorithm R can be generalized to an online polynomial-time algorithm for the weight flow time problem $1|r_i, pmtn| \sum w_i \cdot F_i$ by recomputing the remaining schedule at each release date. The algorithm given in [5] and the generalization of R are quite similar, though not identical. A $O(\log W)$-competitive offline polynomial time algorithm was given in [2]. In [4], a quasi-polynomial time approximation scheme is given. The resource augmentation analysis of HDF was extended to weighted sum of flow times squared in [3].

References

[1] S. Aacharya, S. Muthukrishnan, Scheduling on-demand broadcasts: New metrics and algorithms, in: 4th ACM/IEEE International Conference on Mobile Computing and Networking, 1998, pp. 43–54.
[2] N. Bansal, K. Dhamdhere, Minimizing weighted response time, in: ACM/SIAM Symposium on Discrete Algorithms, 2003, pp. 508–516.
[3] N. Bansal, K. Pruhs, Server scheduling in the weighted l_p norm, in: Latin American Theoretical INformatics, 2004, pp. 434–443.
[4] C. Chekuri, S. Khanna, Approximation schemes for preemptive weighted flow time, in: ACM Symposium on Theory of Computing, 2001.
[5] C. Chekuri, S. Khanna, A. Zhu, Algorithms for minimizing weighted flow time, in: ACM/SIAM Symposium on Theory of Computing, 2001.
[6] J. Labetoulle, E. Lawler, J.K. Lenstra, A. Rinnooy Kan, Preemptive scheduling of uniform machines subject to release dates, in: W.R. Pulleyblank (Ed.), Progress in Combinatorial Optimization, Academic Press, New York, 1984, pp. 245–261.
[7] V. Liberatore, Scheduling jobs before shut-down, Nordic J. Comput. 7 (3) (2000) 204–226.
[8] S. Leonardi, D. Raz, Approximating total flow time on parallel machines, in: ACM Symposium on Theory of Computing, 1997, pp. 110–119.
[9] B. Kalyanasundaram, K. Pruhs, Speed is as powerful as clairvoyance, J. ACM 47 (4) (2000) 617–643.

[10] B. Kalyanasundaram, K. Pruhs, Minimizing flow time nonclairvoyantly, in: IEEE Symposium on Foundations of Computer Science, 1997, pp. 345–352.
[11] C. Phillips, C. Stein, E. Torng, J. Wein, Optimal time-critical scheduling via resource augmentation, in: ACM Symposium on Theory of Computing, 1997, pp. 140–149.
[12] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, Complexity and Approximation, Springer, Berlin, 1999.