Optimally Competitive List Batching

Wolfgang W. Bein[1], Leah Epstein[2], Lawrence L. Larmore[1], and John Noga[3]

[1] School of Computer Science, University of Nevada
Las Vegas, Nevada 89154, USA. ***
bein@cs.unlv.edu larmore@cs.unlv.edu
[2] School of Computer Science, The Interdisciplinary Center
Herzliya, Israel. †
lea@idc.ac.il
[3] Department of Computer Science, California State University Northridge
Northridge, CA 91330, USA
jnoga@csun.edu

Abstract. Batching has been studied extensively in the offline case, but many applications such as manufacturing or TCP acknowledgement require online solutions.
We consider online batching problems, where the order of jobs to be batched is fixed and where we seek to minimize average flow time. We present optimally competitive algorithms for s-batch (competitive ratio 2) and p-batch problems (competitive ratio of 4.) We also derive results for naturally occurring special cases. In particular, we consider the case of unit processing times.

Keywords: Design of Algorithms; Online Algorithms; Batching, TCP acknowledgement.

1 Introduction

Batching problems are machine scheduling problems, where a set of jobs $\mathcal{J} = \{1, \ldots, n\}$ with processing times p_i, $i \in \mathcal{J}$, has to be scheduled on a single machine. The set of jobs $\mathcal{J}$ has to be partitioned into $\mathcal{J} = \bigcup_{k=1}^{r} \mathcal{J}_k$, to form a sequence of *batches* $\mathcal{J}_1, \ldots, \mathcal{J}_r$ for some integer r. A batch combines jobs to run jointly, and each job's completion time is taken to be the completion time of the entire batch. We assume that when each batch is scheduled it requires a setup time s. In an *s-batch* problem the length of a batch is the sum of the processing times of the jobs in the batch, whereas in a *p-batch* problem the length is the maximum of the processing times in the batch. We seek to find a schedule that minimizes the *total flow time* $\sum t_i$, where t_i denotes the completion time of job i in a given schedule, and consider the versions the problems, where the order of the jobs is given and fixed. We respectively refer to these problems as the *list s-batch* problem and *list p-batch* problem. We remark that the use of the term

*** Research of these authors supported by NSF grant CCR-9821009.
† Research of this author supported by Israel Science Foundation grant 250/01.

"s-batch" intuitively has to do with the fact that in an s-batch problem jobs are to be processed sequentially, whereas for "p-batch" problems the jobs in each individual batch are to processed on the machine in some parallel manner.

We say that the batching problem is *online*, if jobs arrive one by one, and each job has to be scheduled before a new job is seen. By "a job has to be scheduled" we mean that the job has to either (a) be included in the current batch or (b) it has to be scheduled as the first job of a new batch. An algorithm that follows this protocol is called an *online algorithm* for the batching problem. We say the online algorithm *batches* a job if it follows the action described in (b), otherwise, if it follows the action in (a) we say the algorithm does not batch.

The quality of an online algorithm $\mathcal{A}$ is measured by the *competitive ratio*, which is the worst case ratio of the cost of $\mathcal{A}$ to the cost of an optimal offline algorithm *opt* that knows the entire sequence of jobs in advance. We note that the offline list s-batch problem is a special case of the $1|s-batch|\sum C_i$ problem which has been well studied and has a linear time algorithm [3]. Many related offline problems have been studied as well, see e.g. [4, 2].

An application of the problem is the following. Jobs (or processes) are to be run on either a single processor or on a large number of multiple processors. Jobs are partitioned into batches that use a joint resource. The resources of each batch have to be set-up before it can start. The successful processing of a batch acknowledged when it terminates. A job may be seen as completed when an acknowledgement is sent (and not necessarily at its completion time), which is done after all jobs of the batch are completed. The goal is to minimize the sum of flow times of all jobs. An s-batch simulates a single processor, in that case at each time one job is run, and a batch is completed when all its jobs are completed, i.e. the time to run a batch is the sum of processing times of its jobs. A p-batch simulates a multiprocessor system where each job may run on a different processor and therefore the time to run a batch is the maximum processing time of any job in the batch.

Our problem is related to the TCP acknowledgement problem. With TCP there exists a possibility of using a single acknowledgement packet to simultaneously acknowledge multiple arriving packets, thereby reducing the overhead of the acknowledgments. Dooly, Goldman, and Scott [5] introduced the dynamic TCP acknowledgement problem in which the goal is to minimize the number of acknowledgments sent plus the sum of the delays of all data packets which are the time gaps between the packets arrival time and the time at which the acknowledgement is sent. The above paper gave an optimally competitive algorithm (with competitive ratio 2) for this problem. Albers and Bals [1] derived tight bounds for a variation of the problem in which the goal is to minimize the number of acknowledgments sent plus the maximal delay incurred for any of the packets. A more generalized problem where a constant number of clients is to be served by a single server was recently studied in [6].

As mentioned above, the s-batch problem is an online one-machine scheduling problem. The p-batch can be seen as such a problem as well, where the single machine is capable of processing several jobs in parallel. We can see the batching

problems both as scheduling to minimize the sum of completion times, and as scheduling to minimize the sum of flow times, as no release times are present (the flow time of a job is its completion time minus its release time). However, both the classical "sum of completion times" problem and the "sum of flow times" are very different from the s-batch problem. For completion times, the optimal competitive ratio is 2 [9–11], whereas for the flow problem the best competitive ratio can be easily shown to be linear in the number of jobs. In these two problems there are release times and no set-up times, so there are no batches. Each job is run separately and the jobs do not need to be assigned in the order they arrived. There are very few one-machine papers where immediate decision on assignments is required. Fiat and Woeginger [8] studied one such model where the goal is minimization of total completion time. A single machine is available to be used starting time zero. Each job has to be assigned (immediately upon arrival) to a slot of time. The length of this slot should be identical to the processing requirement of the job. However, idle times may be introduced, and the jobs can be run in any order. It was shown that the competitive ratio is strictly larger than logarithmic in the number of jobs n, but for any $\varepsilon > 0$, an algorithm of $(\log n)^{1+\varepsilon}$ competitive ratio exists. Another immediate dispatching problem to minimize the sum of completion times (plus a penalty function) is studied in [7]. Jobs arrive one by one, where a job can be either accepted or rejected by paying some penalty (which depends on the job). The penalty function is simply the sum of penalties of rejected jobs.

2 Optimally Competitive List s-Batching

Throughout this paper we will assume for the setup time that $s = 1$ since processing times can be scaled appropriately. Also, we will make use of very short *null jobs*. We denote a null job by the symbol $\mathcal{O}$, and a sequence of null jobs by $\mathcal{O}_1, \mathcal{O}_2, \ldots$. The length of a null job is $\varepsilon > 0$, where ε is arbitrarily small, and therefore we will simplify our exposition by appropriately ignoring this quantity for the length of a schedule. We note that in all of our proofs the ε quantities could be introduced explicitly and then a limit taken at the end, with no change to the results.

Theorem 2.1. *The competitive ratio of any deterministic online algorithm for the list s-batch problem is no better than* 2.

Proof. We first show that for any deterministic online algorithm $\mathcal{A}$ one can construct a request sequence such that the cost for $\mathcal{A}$ cannot be can be better than twice the cost of *opt* on that sequence. Such a sequence is made up of a number of phases. Each phase consists of a number L of null jobs, followed by a single job π of length 1; we write $\sigma^L = \mathcal{O}_1, \ldots, \mathcal{O}_L, \pi$. An entire sequence is of the form:

$$\rho_k = \sigma^N, \sigma^{N^2}, \ldots, \sigma^{N^k}$$

where N is a large integer. Thus a sequence is always made up of a number phases, where the length of the phase is increasing from phase to phase. A sequence of this form is sufficient to prove our result, for any $\mathcal{A}$.

Certainly $\mathcal{A}$ must either

a) batch during every phase σ^i for $i \leq m$, for some $m \ll N$, or
b) have an earliest phase $i \leq m$ in which it does not batch.

Case a: In this case the sequence is chosen to be ρ_m. We have

$$cost_{opt} = N^m(1 + (m-1)) + O(N^{m-1}).$$

We note that we may w.l.o.g. assume $\mathcal{A}$ always batches towards the end of the phase right before the job of length 1. Thus,

$$cost_{\mathcal{A}} = N^m(m + (m-1)) + O(N^{m-1}).$$

The lower bound follows for $\mathcal{A}$ since

$$\frac{m + (m-1)}{1 + (m-1)} \to 2$$

as m increases.

Case b: Whenever $\mathcal{A}$ batches we may as before assume that $\mathcal{A}$ always batches towards the end of the phase right before the job of length 1. Unlike before, in this case there exists a smallest ℓ such that $\mathcal{A}$ does not batch in phase ℓ. To obtain our result the sequence is chosen to be ρ_ℓ. We have

$$cost_{opt} = N^\ell(1 + (\ell - 1)) + O(N^{\ell-1}),$$

whereas

$$cost_{\mathcal{A}} = N^\ell(\ell + \ell) + O(N^{\ell-1}),$$

yielding again the lower bound of 2.

We now present a class of algorithms, one of which achieves the competitive ratio of 2 and thus matches the lower bound. For each $B > 0$, we define algorithm Pseudobatch(B) and, in fact, we call the algorithm with parameter value $B = 1$ simply "Pseudobatch", *i.e.* without any parameter. Algorithm Pseudobatch(B) keeps a tally of processing thus far; we call the set of jobs associated with this tally the "pseudo batch". Once a job α causes the processing requirements in the pseudo batch to exceed B, algorithm Pseudobatch(B) batches, and the pseudo batch is cleared. Note that this means that (a) α is the first job in a new batch and (b) the old tally, which contains p_α, is cleared and thus p_α is not part of the upcoming tally during the batch just opened. We note that therefore in general the pseudo batches and the actual batches created by the algorithm are shifted by one job, and the very first job does not belong to any pseudo batch.

Theorem 2.2. *The competitive ratio of algorithm* Pseudobatch *is no worse than* 2.

Proof. We note that for the optimal schedule with completion times $t_1^*, t_2^*, \ldots$. we have

$$t_i^* \geq 1 + S_i$$

where $S_i = \sum_{j=1}^{i} p_j$.

For the algorithm we have

$$t_i \leq m_i + S_i + 1$$

where m_i is the number of batches up to job i including the current batch. We also have

$$m_i \leq 1 + S_i$$

Thus $t_i \leq 2 + 2S_i$, which implies the result.

The next result shows that the exact competitiveness of 2 relies on the fact that the jobs may be arbitrarily small. Indeed, it is easy to show that if there is a lower bound on the size of the jobs then it is possible to construct an algorithm with competitiveness better than two.

Theorem 2.3. *If the processing time of every job is at least p, then there is a C-competitive online algorithm, where $C = \frac{1+\sqrt{p+1}}{\sqrt{p+1}}$.*

Proof. We consider PSEUDOBATCH(B), with $B = \sqrt{p+1}$. Note that $C = 1 + \frac{1}{B}$. We will prove that PSEUDOBATCH(B) is C-competitive, given that $p_i \geq p$ for all i.

As before, let $S_i = \sum_{j=1}^{i} p_j$, t_i^* the completion time of the i^{th} job in the optimal schedule, t_i the completion time of the i^{th} job in the schedule created by PSEUDOBATCH(B), and m_i the number of batches up to and including the batch which contains the i^{th} job in the schedule created by PSEUDOBATCH(B). We shall prove that

$$\frac{t_i}{t_i^*} \leq C \tag{1}$$

for all i.

The i^{th} job is in the m_i^{th} batch. Let us say that the ℓ^{th} job is the last job in that batch. Then $S_\ell - S_i \leq B$, since $i+1, \ldots, \ell$ must be in the same pseudo batch.

As before, $t_i^* \geq S_i + 1$, and $t_i = S_\ell + m_i$. By the definition of the algorithm PSEUDOBATCH(B), we have $m_i \leq \frac{S_i - p_1}{B} + 1$. Since $p_1 \geq p$, we have that

$$\frac{p_1 + 1 + B}{p_1 + 1} \leq \frac{p + 1 + B}{p + 1} = C$$

Recall that $1 + \frac{1}{B} = C$. We have:

$$\frac{t_i}{t_i^*} \leq \frac{S_\ell + m_i}{S_i + 1}$$

$$
\begin{aligned}
&\leq \frac{S_i + m_i + B}{S_i + 1} \\
&= \frac{S_i - p_1 + m_i + p_1 + B}{S_i - p_1 + 1 + p_1} \\
&\leq \frac{\left(1 + \frac{1}{B}\right)(S_i - p_1) + 1 + p_1 + B}{S_i - p_1 + 1 + p_1} \\
&\leq \frac{C(S_i - p_1) + C(1 + p_1)}{S_i - p_1 + 1 + p_1} = C
\end{aligned}
$$

This verifies (1) for each i, and we are done.

3 Identical Job Sizes

For machine scheduling problems it is typical that restricting to unit jobs makes the problem easier to analyze. However, this is not the case for list batching. In this section we give results for the case $s = 1$ and $p_j = 1$, for all $j \in \mathcal{J}$. In this case we can give an exact description of the optimal offline solution. To this end define

$$
\begin{aligned}
\mathit{optcost}[n] &= \text{optimal cost of n jobs} \\
\mathit{firstbatch}[n] &= \text{size of first batch for n jobs.}
\end{aligned}
$$

We have the following recursive definition of $\mathit{optcost}[n]$:

$$
\mathit{optcost}[n] = \begin{cases} 0 & \text{for } n = 0 \\ \min_{0 \leq p < n} \mathit{optcost}[p] + n(n - p + 1) & \text{for all } n > 0. \end{cases}
$$

To see this let $n - p$ be the number of items in the first batch. Then p is the number of items in the remaining batches. We assume that they are processed optimally. The cost of processing the first batch is $(n - p)(n - p + 1)$. The cost of processing all remaining batches is $\mathit{optcost}[p] + (n - p + 1)p$.

We define a function $F[n]$, for $n >= 0$, as follows. If $n = m(m + 1)/2 + k$ for some $m \geq 0$ and some $0 \leq k \leq m + 1$, then

$$
F[n] = \frac{m(m+1)(m+2)(3m+5)}{24} + k(n + m - k + 1) + \frac{k(k+1)}{2}. \qquad (2)
$$

In the special case that $n = \frac{m(m+1)}{2}$ for some $m > 0$, then the rule gives two different formulae for $F[n]$. Routinely, we verify that the values are equal, in fact they are both equal to $m(m+1)\frac{3m^2+11m+10}{24}$.

The following facts will be useful in describing the offline solution in closed form:

Lemma 3.1. *a) If* $n = \frac{m(m+1)}{2} + k$ *where* $0 \leq k < m+1$, *then* $F[n+1] - F[n] = n + m + 2$.

b) If $n = \frac{m(m+1)}{2}$, *then* $F[n] - F[n-1] = n + m$.
c) If $n \geq 1$, *then* $F[n+1] + F[n-1] > 2F[n]$.

Proof. We first prove part a). To that end, let

$$F[n] = \frac{m(m+1)(m+2)(3m+5)}{24} + k(n+m-k+1) + \frac{k(k+1)}{2}$$

and

$$F[n+1] = \frac{m(m+1)(m+2)(3m+5)}{24} + (k+1)(n+m-k+1) + \frac{(k+1)(k+2)}{2}.$$

Then

$$F[n+1] - F[n] = (n+m-k+1) + k + 1 = n + m + 2,$$

which proves part a).

For part b) simply write $n = (m-1)m/2 + m$, then apply part a).

In the proof of part c) we consider two cases:

Case 1: $n = m(m+1)/2 + k$ where $0 < k < m+1$. Applying part a) twice,

$$F[n+1] + F[n-1] - 2F[n] = (n+m+2) - (n+m+1) = 1.$$

Case 2: $n = m(m+1)/2$ for some $m > 0$. Then $F[n+1] - F[n] = n + m + 2$ by part a), and $F[n] - F[n-1] = n + m$ by part b). Thus

$$F[n+1] + F[n-1] - 2F[n] = 2.$$

We are now ready to give the closed form:

Theorem 3.2. *For* $optcost[n]$, $optcost = F[n]$ *for all* $n \geq 0$. *Furthermore, if* $n = \frac{m(m+1)}{2} + k$ *for some* $m \geq 0$ *and some* $0 \leq k \leq m+1$, *then the optimal size of the first batch is* m *if* $k = 0$, *is* $m+1$ *if* $k = m+1$, *and is either* m *or* $m+1$ *if* $0 < k < m+1$.

Proof. We first show $optcost[n] \leq F[n]$ for all n by strong induction on n.

If $n = 0$ we are done. If $n > 0$, select $m > 0$ and $0 \leq k < m+1$ such that $n = \frac{m(m+1)}{2}A + k$.

We show

$$F[n] = F[n-m] + n(m+1) \tag{3}$$

To show equation 3, note

$$F[n] = \frac{m(m+1)(m+2)(3m+5)}{24} + k(n+m-k+1) + \frac{k(k+1)}{2}.$$

Then, since $0 \leq k \leq m$

$$F[n-m] = \frac{(m-1)m(m+1)(3m+2)}{24} + k(n-k) + \frac{k(k+1)}{2}.$$

Then

$$F[n] - F[n-m] - n(m+1) =$$
$$\frac{m(m+1)(12m+12)}{24} + k(m+1) - \frac{m(m+1)}{2} + k(m+1) = 0.$$

This establishes equation 3.

By the inductive hypothesis, $F[n-m] \geq \mathit{optcost}[n-m]$. By definition, $\mathit{optcost}[n] \leq \mathit{optcost}[n-m] + n(m+1)$. Then $F[n] = F[n-m] + n(m+1) \geq \mathit{optcost}[n-m] + n(m+1) \geq \mathit{optcost}[n]$. This completes the proof of the fact $\mathit{optcost}[n] \leq F[n]$ for all n.

It now follows the proof that indeed $\mathit{optcost}[n] = F[n]$ holds. This is also by strong induction. The case $n = 0$ is trivial. For fixed $n > 0$, we now define

$$G[p] = F[p] + n(n-p+1) \text{ for all } p < n.$$

It is easy to see that

$$G[p+1] + G[p-1] > 2G[p] \text{ for all } p > 0. \tag{4}$$

since the second term in the definition of G is linear and by Lemma 3.1, part c).

Let $n > 0$. Write $n = \frac{m(m+1)}{2} + k$, where $m > 0$ and $0 <= k < m+1$.

Case 1: $k = 0$. Then let $p = n - m = \frac{(m-1)m}{2}$. By definition of G and by Lemma 3.1, part a), we have

$$G[p+1] - G[p] = 1.$$

By definition of G and by Lemma 3.1, part b), we have

$$G[p] - G[p-1] = 1.$$

We can easily check that $F[n] = G[p]$. It follows, by equation 4 and by the inductive hypothesis that $\mathit{optcost}[n] = F[n]$ and that $\mathit{firstbatch}[n] = m$.

Case 2: $k > 0$. Then let $p = n - m - 1 = \frac{(m-1)m}{2} + k - 1$. By definition of G and by Lemma 3.1, part a), we have

$$G[p+1] - G[p] = 0.$$

By definition of G and by Lemma 3.1, part a), we have

$$G[p+2] - G[p+1] = 1.$$

If $k = 1$, by definition of G and by Lemma 3.1, part b), we have

$$G[p] - G[p-1] = 1.$$

If $k > 1$, by definition of G and by Lemma 3.1, part a), we have

$$G[p] - G[p-1] = 1.$$

We can easily check that

$$F[n] = G[p] = G[p+1].$$

It follows the inductive hypothesis and equation 4, that $optcost[n] = F[n]$ and that $firstbatch[n] = m$ or $m+1$.

Our next goal is to find a lower bound on the competitive ratio of any algorithm for the unit jobs case and give an algorithm which achieves this ratio.

Define $\mathcal{D}$ to be the algorithm which batches after jobs: 2, 5, 9, 13, 18, 23, 29, 35, 41, 48, 54, 61, 68, 76, 84, 91, 100, 108, 117, 126, 135, 145, 156, 167, 179, 192, 206, 221, 238, 257, 278, 302, 329, 361, 397, 439, 488, 545, 612, 690, 781, 888, 1013, 1159, 1329, 1528, 1760, and 2000+40i for all $i \geq 0$.

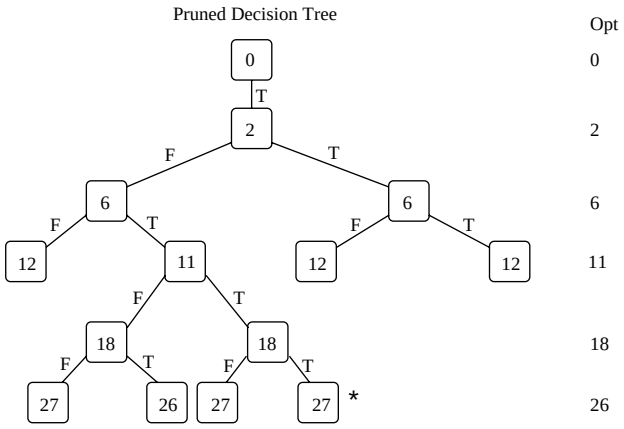

Fig. 1. The Decision Tree used in the Pruning Procedure

Theorem 3.3. *For the list batching problem restricted to unit job sizes no online algorithm can have a competitive ratio smaller than 619/583 and the algorithm described above achieves this ratio.*

Proof. Any online algorithm for list batching restricted to unit jobs is described by a sequence of decisions: should the i^{th} job be the first job in a new batch? In other words, every online algorithm is a path in a decision tree where a node at level i has two children: one representing the choice not to batch prior to job i and one representing making job i the first job in a new batch. However, it can be noted that having an empty batch only increases an algorithm's cost and therefore the first job should begin the first batch (i.e. we should not close the first batch prior to the first job).

If we can show that any path from the root to a node with depth d in the decision tree must encounter a node at which the ratio of online cost to offline

cost is at least 619/583 then we have established our lower bound. Utilizing a small computer program it is easy to verify that this fact holds for $d = 100$. What is unusual is that considering less than 100 jobs does not yield the bound.

Consider the algorithm $\mathcal{D}$ described above. Verifying that $\mathcal{D}$ maintains a cost ratio of at most 619/583 for all job sequences with less than 2000 jobs is tedious but trivial for a computer program. For sequences with more than 2000 jobs we note that: 1) the contribution of the first 2000 jobs to the optimal cost will only increase because the size of the optimal batches increases with the number of jobs and 2) the contribution of job $i > 2000$ to the optimal cost is at least $i+1$ while the contribution to the online cost is at most $i + 48 + (i - 2000)/40 < 619/583i$. Therefore $\mathcal{D}$ is 619/583-competitive.

Given that there are exponentially many paths from the root to a node at depth d, two notes on efficiency are appropriate here. First, if a node is encountered where the ratio of costs is greater than or equal to 619/583 then no further descendents need to be checked. This alone brings the calculation described above to manageable levels. Second, given two nodes n_1 and n_2 which have not been pruned by the previous procedure, if the online cost at n_1 is less or equal to the online cost at n_2 and both have done their most recent batching at the same point then descendants of n_2 need not be considered. This follows because the cost on any sequence of choices leading from n_2 is greater or equal to the the same cost on n_1. We illustrate the preceding ideas with the diagram of Figure 1. Level i corresponds to all possible decisions after i jobs have arrived. We can prune at level 3 because $12/11 > 619/583$ and descendants of the starred node need not be considered.

4 The List p-Batch Problem

We now turn to the list p-batch problem and define a class of algorithms, Threshold, one of which has an optimal competitive ratio. For a sequence $A =< a_1, a_2, \ldots >$ we define Threshold(A) to be the algorithm that batches for the ℓ^{th} time whenever the processing requirement is larger of equal threshold value a_ℓ. Specifically, we consider the sequnce $A^\star = \ < (i+1)2^i - 1,\ i = 1, 2, \ldots >$, and we will write "Threshold",*i.e.* without any parameter, to mean Threshold$(A^\star)$.

We have:

Theorem 4.1. *The competitive ratio of algorithm* Threshold *is no worse than* 4.

Proof. Consider a job j which is in the ℓ^{th} batch of the online algorithm. This single job will contribute at most $\ell + \sum_{i=1}^{\ell} a_i^\star$ to the online cost, because there have been at most ℓ set up times and the length of batch i is at most $a_i^\star$. On the other hand, it will contribute at least $1 + a_{l-1}^\star$ to the offline cost. The calculation below shows that the ratio is 4.

$$\frac{\ell + \sum_{i=1}^{\ell} a_i^\star}{1 + a_{l-1}^\star} = \frac{\ell + \sum_{i=1}^{\ell} \left[(i+1)2^i - 1\right]}{1 + l2^{l-1} - 1}$$

$$= \frac{\left[(l-1)2^{l+1}+2\right]+\left[2^{l+1}-2\right]}{l2^{l-1}}$$
$$= \frac{l2^{l+1}}{l2^{l-1}}$$
$$= 4.$$

We now show that algorithms THRESHOLD is optimally comeptitive:

Theorem 4.2. *No deterministic online algorithm for the list p-batch problem can have competitiveness less than 4.*

Proof. We prove that for any $\delta > 0$ there is no $(4-\delta)$-competitive algorithm. Fix $\gamma > 0$ and let N be a very large integer such that $1/\gamma \ll N$. To this end, we show that for any deterministic online algorithm $\mathcal{A}$ one can construct a request sequence such that the cost for $\mathcal{A}$ cannot be can be better than $(4-\delta)$ times the cost of *opt* on that sequence. Define now $\tau^L = \mathcal{O}_1, \ldots, \mathcal{O}_L$. Let $\mathcal{C}^k$ denote a job with processing requirement $k\gamma$. Then we will construct a sequence of jobs $\mathcal{C}^1, \mathcal{C}^2, \mathcal{C}^3, \ldots$ punctuated by various τ^L. More precisely we define the sequence to be

$$\tau^N, \mathcal{C}^1, \mathcal{C}^2, \ldots, \mathcal{C}^{k_1}, \tau^{N^2}, \mathcal{C}^{(k_1+1)}, \ldots, \mathcal{C}^{k_2}, \tau^{N^3}, \ldots$$

where $\mathcal{C}^{k_\ell}$ is the first job in batch $\ell+1$ for the online algorithm and it is stipulated that the sequence may terminate after job $\mathcal{C}^{k_\ell}$ for any ℓ.

Then *opt* can serve all jobs with two batches (one ending with τ^{N^ℓ} and one ending with $\mathcal{C}^{k_\ell}$). Therefore to be $(4-\delta)$-competitive the following inequality must hold for all ℓ

$$N^\ell(\ell + \sum_{i=1}^{\ell}(f_i - \gamma)) + O(N^{\ell-1}) \leq (4-\delta)N^\ell(1+f_{\ell-1}) + O(N^{\ell-1}),$$

where for simplicity, define $f_i = k_i\gamma$. If γ is chosen sufficiently small and N is chosen sufficiently large these inequalities require that

$$\ell + \sum_{i=1}^{\ell} f_i \leq (4-\delta/2)(1+f_{l-1})$$

hold for all ℓ. Further the sequence $f_i, i = 1, 2, \ldots$ must be increasing by definition. A simple variational argument can show that these inequalities have a solution iff there is a solution with all inequalities tight. These resulting equalities can be solved using recurrences. We find that the unique solution yields values of f_i which are not monotone increasing. We conclude that therefore there can be no $(4-\delta)$-competitive algorithm. We mention that if the multiplicative factor $(4-\delta/2)$ is replaced by 4 then there is a solution which is monotone increasing. In fact, in this case, the values of the f_is are the $a_i^\star$s which define THRESHOLD.

5 Conclusion

For the s-batch problem we showed tight bounds of 2. Both the upper bound and the lower bound follow from a ratio of two in the completion times of the algorithm compared to the optimal offline schedule, not only for the total cost, but for each job separately. Therefore those bounds hold for a larger class of goal functions, including weighted total flow time and ℓ_p-norm of flow times. We note that our results for identical job sizes are obtained for the case the the job size equals the setup time. Those techniques can be easily applied to cases where the two values are different.

References

1. S. Albers and H. Bals. Dynamic TCP acknowledgement: Penalizing long delays. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'2003)*, pages 47–55, 2003.
2. P. Baptiste. Batching identical jobs. *Mathematical Methods of Operation Research*, 52:355–367, 2000.
3. P. Brucker. *Scheduling Algorithms*. Springer Verlag, 2001.
4. P. Brucker, A. Gladky, H. Hoogeveen, M. Kovalyov, C. Potts, T. Tautenhahn, and S. van de Velde. Scheduling a batch processing machine. *Journal of Scheduling*, 1(1):31–54, 1998.
5. D. R. Dooly, S. A. Goldman, and S. D. Scott. On-line analysis of the TCP acknowledgement delay problem. *Journal of the ACM*, 48(2):243–273, 2001.
6. L. Epstein and A. Kesselman. On the remote server problem or more about TCP acknowledgments. manuscript, 2003.
7. L. Epstein, J. Noga, and G. J. Woeginger. On-line scheduling of unit time jobs with rejection: Minimizing the total completion time. *Operations Research Letters*, 30(6):415–420, 2002.
8. Amos Fiat and Gerhard J. Woeginger. On-line scheduling on a single machine: Minimizing the total completion time. *Acta Informatica*, 36:287–293, 1999.
9. J. A. Hoogeveen and A. P. A. Vestjens. Optimal on-line algorithms for single-machine scheduling. In *Proc. 5th Conf. Integer Programming and Combinatorial Optimization (IPCO)*, pages 404–414, 1996.
10. X. Lu, R. Sitters, and L. Stougie. A class of on-line scheduling algorithms to minimize total completion time. *Operations Research Letters*, 2003. to appear.
11. C. A. Phillips, C. Stein, and J. Wein. Scheduling jobs that arrive over time. In *Algorithms and Data Structures, 4th International Workshop (WADS'95)*, pages 86–97, 1995.