ELSEVIER

Operations Research Letters 27 (2000) 109–118

operations research letters

www.elsevier.com/locate/dsw

Improved dynamic programs for some batching problems involving the maximum lateness criterion

A.P.M. Wagelmans[a,*], A.E. Gerodimos[b]

[a] *Econometric Institute, Erasmus University Rotterdam, P.O. Box 1738, 3000 DR Rotterdam, Netherlands*
[b] *Centre for Quantitative Finance, Imperial College, Exhibition Road, London SW7 2BX, UK*

Received 1 May 1999; received in revised form 1 November 1999

Abstract

We study four scheduling problems involving the maximum lateness criterion and an element of batching. For all the problems that we examine, algorithms appear in the literature that consist of a sorting step to determine an optimal job sequence, followed by a dynamic programming step that determines the optimal batches. In each case, the dynamic program is based on a backward recursion of which a straightforward implementation requires $O(n^2)$ time, where n is the number of jobs. We present improved implementations of these dynamic programs that are based on monotonicity properties of the objective expressed as a function of the total processing time of the first batch. These properties and the use of efficient data structures enable optimal solutions to be found for each of the four problems in $O(n \log n)$ time; in two cases, the batching step is actually performed in linear time and the overall complexity is determined by the sorting step.

Keywords: Batching; Scheduling; Dynamic programming; Computational complexity; Lateness

1. Introduction

The early 1990s saw the emergence of powerful techniques that reduced the time requirement of dynamic programming algorithms for the classic economic lot sizing (ELS) problem [8,14,1]. Subsequently, it was realized that certain scheduling problems involving the sum of completion times objective and an element of batching exhibited structural properties that made them amenable to more efficient dynamic programming solutions. In some cases [7,3], the improved schemes were problem-specific; in other cases [6,10], the dynamic programming recursion could be written in a form that allowed the application of the *geometric techniques* of Van Hoesel et al. [13], which are a generalization of the technique used in [14]. The typical complexity improvement was from $O(n^2)$ to $O(n \log n)$, where n is the number of jobs. A question that arises naturally

* Corresponding author. Fax: +31-10-408-9162.
E-mail address: wagelmans@few.eur.nl (A.P.M. Wagelmans).

0167-6377/00/$ - see front matter
PII: S0167-6377(00)00040-7

is whether similar improvements can be achieved in solving the maximum lateness counterparts of these batching problems since, in a standard implementation, the respective dynamic programs have also quadratic time requirements. This paper provides an affirmative answer to this question. We study four such batching problems and provide implementations of dynamic programming with a time requirement that is either linear or $O(n \log n)$. Since the batching problems are solved after an initial sorting step, our results imply $O(n \log n)$ algorithms for the four maximum lateness problems.

The remainder of this paper is organized as follows. In Section 2 we sketch our approach with particular focus on a subproblem that we encounter frequently when solving the four batching problems. Subsequently, we list the problems in order of relative complexity, both in terms of the improved running time and the difficulty of obtaining this improvement. Specifically, Section 3 deals with the problem of batching jobs of a single type under batch availability. A problem in which jobs are processed by a batching machine is the subject of Section 4. In Section 5 we give an improved algorithm for batching customized two-operation jobs on a single machine under batch availability and we indicate how a similar approach can be adopted in the case of item availability.

2. Preliminaries

In general, solving scheduling problems with a batching element involves taking the appropriate batching and sequencing decisions. For the problems that we examine in this paper, these two aspects can be decoupled. In fact, for three of our problems there is an optimal schedule in which jobs complete according to the earliest due date (EDD) rule, whereas for the problem studied in Section 4 the shortest processing time (SPT) rule is optimal. In any case, the sorting step imposes a lower bound of $O(n \log n)$ on the overall complexity of any algorithm. For two of the problems examined here, improving the efficiency of the dynamic programming step results in the sorting step being the overall bottleneck.

Although it is difficult to provide a description of a procedure that would be general enough to be applied to all the problems tackled in this paper, we now sketch some common elements of our approach; the implementation details and some special data structures deployed are covered in the subsequent sections.

Our starting point is always a backward recursion dynamic program with *batch insertion* [11]: for a predetermined job sequence, the optimal schedule is built by inserting entire batches of jobs (or operations) at the start of previously obtained schedules. The recursions are of the general form:

$$G_k = \min_{k<l\leqslant n+1} \{\max\{P_{k,l} + G_l, L_{k,l}\}\}, \tag{1}$$

where G_k is the minimum lateness of schedules including jobs $k, k+1, \ldots, n$, whereas $P_{k,l}$ and $L_{k,l}$ denote the total processing time and overall lateness of the inserted batch, which consists of the jobs $k, k+1, \ldots, l-1$. In other words, l is the index of the first job in the second batch. For a given choice of l, the overall lateness of the schedule either occurs in the first batch, in which case it is equal to $L_{k,l}$, or it occurs in the part of the schedule starting with job l. The minimum lateness in the latter part of the schedule is simply the value G_l corrected for the fact that job l starts at time $P_{k,l}$ instead of time 0. This explains the term $P_{k,l} + G_l$ in (1). The overall lateness of the schedule is given by the maximum of the lateness in the first batch and the lateness in the remainder of the schedule. The minimum operation in (1) corresponds to the fact that the best possible choice for the start of the second batch is made.

The first step in our approach is to observe that the index set $\{k+1, \ldots, n+1\}$ can be partitioned into two mutually exclusive index sets I_k^1 and I_k^2 so that the first set contains exactly those indices l for which $P_{k,l} + G_l \geqslant L_{k,l}$ and the second set contains the remaining indices. In view of that, (1) can be re-written as

$$G_k = \min\left\{\min_{l\in I_k^1}\{P_{k,l} + G_l\}, \min_{l\in I_k^2} L_{k,l}\right\}, \tag{2}$$

For reasons that will become clearer in the subsequent sections, the solution to the second minimization problem in (2) is obtained by retrieving the minimal index l^* from I_k^2. This leaves us with the following tasks:

(a) maintain/update the index sets I_k^1 and I_k^2 efficiently;
(b) solve the first minimization problem and, where applicable, a second optimization problem that arises when calculating L_{k,l^*}.

With respect to the first task, observe that I_k^1 and I_k^2 are not necessarily contiguous. In fact, as we show in the subsequent sections, the satisfaction of this additional condition by some problems leads to linear-time implementations; where this is not the case, updating these index sets is "costly" and the complexity of the dynamic program becomes $O(n \log n)$.

As for the second task, both the first minimization problem and the non-trivial variants of the second optimization problem possess a key property that enables a solution to be found in time which is overall linear. Specifically, the idea is to transform all such problems into a problem of the following type:

Problem P. *Determine the minima* m_k, $k = 1, 2, \ldots, n$, *defined as*

$$m_k = \min_{k < l \leqslant u_k} f_l,$$

where $u_k \in \{k, k+1, \ldots, n\}$ *for all* $k = 1, 2, \ldots, n$ ($m_k = \infty$ *if* $u_k = k$) *and the following conditions hold*:

(a) $u_k \leqslant u_{k+1}$ *for all* $k = 1, 2, \ldots, n-1$;
(b) $u_n = n$;
(c) u_k *is known once* m_{k+1} *is known*, $k = 1, 2, \ldots, n-1$;
(d) f_k *is known once* m_k *is known*, $k = 1, 2, \ldots, n$.

Conditions (c) and (d) suggest that the values m_k can only be calculated in order of decreasing index k. A straightforward way to solve problem P requires $O(n^2)$ time. We show, however, that a linear time bound is possible.

Consider, for an arbitrary $k \in \{1, 2 \ldots, n\}$ with $u_k > k$, the values f_l, $l = k+1, k+2, \ldots, u_k$. Let $t(1)$, $t(2), \ldots, t(r)$ be the unique subsequence of $k+1, k+2, \ldots, u_k$ which has the following properties:

1. $t(1) = k+1$;
2. $t(i+1)$ is the smallest index in $\{t(i)+1, t(i)+2, \ldots, u_k\}$ such that $f_{t(i+1)} < f_{t(i)}$, $i = 1, \ldots, r-1$.

Clearly, this subsequence has the properties $t(1) < t(2) < \cdots < t(r)$ and $f_{t(1)} > f_{t(2)} > \cdots > f_{t(r)}$. Moreover, $f_{t(r)} = m_k$. Hence, given the subsequence, the desired minimum is immediately available.

We keep track of the subsequence by storing its elements in decreasing order in a doubly linked list (i.e, f_{k+1} is the element at the top). This particular data structure has the property that elements can be deleted from or added to the top or bottom of the list in constant time (see [2]).

To see why this data structure is convenient, first observe the following: if for a given $k \geqslant 2$, a value $l \in \{k+1, k+2, \ldots, u_k\}$ is not selected in the subsequence, then l will not be selected for $k-1$. This means that when for a certain $k \geqslant 2$ the elements $t(1), t(2), \ldots, t(r)$ of the subsequence are given, then – once u_{k-1} is known – the corresponding subsequence for $k-1$ can be constructed as follows. Because $u_{k-1} \leqslant u_k$, we first delete from the bottom of the list any element larger than u_{k-1}. Now, suppose $u_{k-1} > k-1$. Then, because k will be added at the top of the list, we delete from the top all remaining elements $t(i)$ for which $f_k \leqslant f_{t(i)}$. Finally, we add k at the top of the list. In case $u_{k-1} = k-1$ the list is empty after the deletion operations and no element is added.

The above updating process is carried out $n-1$ times in total. Each time at most one element is added, which requires constant time per addition. Furthermore, several elements may be deleted. Note that, because

the list elements are already ordered, each deletion requires indeed constant time. The number of deleted elements cannot be bounded nicely for each separate execution of the updating process. However, the overall number of deletions is not larger than n. The reason for this is simple: in the updating process, each of the elements $1,2,\ldots,n$ is added at most once to the list and therefore it can be deleted at most once.

To summarize the above discussion: we have shown that problem P can be solved in $O(n)$ time.

In the following sections, we will repeatedly have to calculate partial sums such as $\sum_{h=k}^{n} p_h$ for all $k = 1,2,\ldots,n$. Note that this can be done in overall linear time in a preprocessing step. When needed, partial sums such as $\sum_{h=k}^{l-1} p_h$ can subsequently be calculated in constant time as $\sum_{h=k}^{n} p_h - \sum_{h=l}^{n} p_h$.

3. Scheduling jobs of a single type under batch availability

The problem we are addressing in this section may be stated formally as follows. There are n jobs to be scheduled on a single machine. Each job j ($j=1,\ldots,n$) has a processing time p_j and a due date d_j by which it should ideally complete. Jobs can be processed consecutively in *batches*. At the start of the schedule and prior to each batch, a set-up time s is incurred, which motivates the formation of longer batches so as to reduce the completion time of later jobs. However, *batch availability* applies, which means that all the jobs that belong to the same batch complete only when the last job in the batch completes. As a consequence, extending a batch by including additional jobs increases the completion time of the jobs previously in the batch.

The above problem setting is introduced in [12]. For the sum of completion times objective, an efficient algorithm is given by Coffman et al. [7]: the batching step is performed in linear time to give an overall time requirement of $O(n \log n)$. An extension of this algorithm for a slightly more general cost function is proposed by Albers and Brucker [3] (see also [5]). It is worth pointing out that the approach in [7,3], like ours, relies on the notion of a queue. However, in the problem examined here, the presence of a maximum operation within the dynamic programming recursion is an additional complication that does not arise in the sum of completion times variant. (This is also true for the problems addressed in later sections.)

It is shown in [15] that there is an optimal schedule in which jobs complete according to the earliest due date (EDD) rule. Thus, the jobs can be re-indexed according to this rule in $O(n \log n)$ time and the problem reduces to one of batching that can be solved using a backward dynamic program with batch insertion. Let G_k denote the minimum overall lateness of a schedule containing jobs $k,k+1,\ldots,n$ when starting at time 0. The initialization is $G_{n+1} = -\infty$ and the recursion for $k = n,n-1,\ldots,1$ is

$$G_k = \min_{k<l\leqslant n+1} \left\{ \max \left\{ \left(s + \sum_{h=k}^{l-1} p_h \right) + G_l, \left(s + \sum_{h=k}^{l-1} p_h \right) - d_k \right\} \right\}, \tag{3}$$

Here l denotes the first job in the second batch of the schedule. Since this batch starts at time $s + \sum_{h=k}^{l-1} p_h$, the minimum overall lateness from this batch onward is given by the first term between brackets, while the lateness of the first batch is given by the other term (since job k has the smallest due date).

As pointed out in [15], a straightforward implementation of the above algorithm requires $O(n^2)$ time. However, we now show that the dynamic programming part can be implemented in linear time.

From (3) we have, for every $l \leqslant n-1$,

$$G_{l+1} = \min_{l+1<i\leqslant n+1} \left\{ \max \left\{ \left(s + \sum_{h=l+1}^{i-1} p_h \right) + G_i, \left(s + \sum_{h=l+1}^{i-1} p_h \right) - d_{l+1} \right\} \right\},$$

which, because $d_l \leqslant d_{l+1}$, implies

$$G_{l+1} \leqslant \min_{l+1<i\leqslant n+1} \left\{ \max \left\{ \left(s + \sum_{h=l}^{i-1} p_h \right) + G_i, \left(s + \sum_{h=l}^{i-1} p_h \right) - d_l \right\} \right\}. \tag{4}$$

Furthermore, it clearly holds that

$$G_{l+1} \leqslant \max\{(s + p_l) + G_{l+1},\ (s + p_l) - d_l\}. \tag{5}$$

Combining (4) and (5) yields

$$G_{l+1} \leqslant \min_{l<i\leqslant n+1} \left\{ \max \left\{ \left(s + \sum_{h=l}^{i-1} p_h \right) + G_i, \left(s + \sum_{h=l}^{i-1} p_i \right) - d_l \right\} \right\},$$

which, because of (3), is simply $G_{l+1} \leqslant G_l$. Hence, if $G_{l+1} \geqslant -d_k$ for some $k \in \{1,\ldots,n\}$, then also $G_l \geqslant -d_k$. We now define q_k as the largest job index l in $\{k+1,\ldots,n\}$ such that $G_l \geqslant -d_k$; if no such index exists, we define $q_k = k$. (Note that, because of the EDD order, $q_{k+1} \geqslant q_k$ holds for every $k \leqslant n-1$.)

From the above observations, it follows that for all indices $l \in I_k^1 = \{k+1,\ldots,q_k\}$ the maximum in (3) is given by the first term, whereas for $l \in I_k^2 = \{q_k+1,\ldots,n+1\}$, the maximum is given by the second term. Now (3) can be rewritten as

$$G_k = \min \left\{ \min_{k<l\leqslant q_k} \left\{ s + \sum_{h=k}^{l-1} p_h + G_l \right\}, \quad \min_{q_k<l\leqslant n+1} \left\{ s + \sum_{h=k}^{l-1} p_h - d_k \right\} \right\}.$$

Note that, for this problem, each of I_k^1 and I_k^2 is contiguous. Further, the second minimum is always attained for $l = q_k + 1$: owing to the batch availability assumption and the EDD indexing of the jobs, the overall lateness of a batch is always determined by the first job in the batch. Consequently, the remaining task is to compute

$$s + \sum_{h=k}^{n} p_h + \min_{k<l\leqslant q_k} \left\{ -\sum_{h=l}^{n} p_h + G_l \right\} \tag{6}$$

efficiently. However, since this has to be done for every value of k, we actually need to solve an instance of problem P with $u_k = q_k$ and $f_l = -\sum_{h=l}^{n} p_h + G_l$. As shown in Section 2, this problem can be solved in $O(n)$ time. Hence, it takes overall $O(n)$ time to calculate the minima given by (6).

Since the values $\sum_{h=l}^{n} p_h$, $l = 1,2,\ldots,n$, and, because of monotonicity, the values q_k, $k = 1,2,\ldots,n$, can be computed in $O(n)$ time, we have now shown that the time requirement of our algorithm to solve the batching problem is linear. Hence, because of the sorting step, the overall time requirement is $O(n \log n)$. This constitutes an improvement over the algorithm in [15].

4. Scheduling jobs on a batching machine

The problem we are addressing in this section may be stated formally as follows. There are n jobs to be processed on a single *batching machine*. This machine is capable of processing up to b jobs simultaneously in batches. Each job j ($j = 1,\ldots,n$) has a processing time p_j and a due date d_j by which it should ideally complete. Whenever a batch is formed, its completion time is equal to the largest processing time of any job in the batch.

The model is analyzed extensively in a recent paper by Brucker et al. [6]. They distinguish between the *unbounded* case where $b \geqslant n$ and the *bounded* case whereby $b < n$. For the unbounded problem of minimizing the maximum lateness, it is shown in [6] that there is an SPT-batch optimal schedule. Thus, the jobs can be re-indexed according to this rule in $O(n \log n)$ time and the problem reduces to one of batching that can

be solved using the following backward dynamic program with batch insertion of Brucker et al. [6]. Let G_k denote the minimum overall lateness of a schedule containing jobs $k, k+1, \ldots, n$ when starting at time 0. The initialization is $G_{n+1} = -\infty$ and the recursion for $k = n, n-1, \ldots, 1$ is

$$G_k = \min_{k<l\leqslant n+1} \left\{ \max \left\{ p_{l-1} + G_l,\ p_{l-1} + \max_{k\leqslant j\leqslant l-1} \{-d_j\} \right\} \right\}, \tag{7}$$

where l should again be interpreted as the first job of the second batch, which starts when the first batch completes. By definition, this happens when the longest job $(l-1)$ of the first batch completes.

A standard implementation of the above algorithm, as proposed in [6], requires $O(n^2)$ time. We now show that the dynamic programming part can be implemented in linear time, thus yielding an overall time requirement of $O(n \log n)$.

Our approach is somewhat similar to the one in the previous section. As before it can be verified that $G_{l+1} \leqslant G_l$ for every $l \leqslant n-1$. Hence, if $G_{l+1} > \max_{k<j\leqslant l}\{-d_j\}$ for some $k \in \{1, \ldots, n\}$, then

$$G_l \geqslant G_{l+1} > \max_{k<j\leqslant l} \{-d_j\} \geqslant \max_{k<j\leqslant l-1} \{-d_j\}.$$

It follows that, if I_k^1 and I_k^2 are defined as in Section 2 (that is: $I_k^1 = \{l \in \{k+1, \ldots, n\} | G_l \geqslant \max_{k<j\leqslant l-1}\{-d_j\}\}$ and $I_k^2 = \{k+1, \ldots, n+1\} \setminus I_k^1$), then, $I_k^1 = \{k+1, \ldots, q_k\}$ and $I_k^2 = \{q_k+1, \ldots, n+1\}$, where q_k is the largest index with the required property. For convenience, we define $q_k = k$ if the inequality is not satisfied by any job in $\{k+1, k+2, \ldots, n\}$. Note that q_k is non-decreasing in k. Recursion formula (7) can now be rewritten as

$$G_k = \min \left\{ \min_{k<l\leqslant q_k} \{p_{l-1} + G_l\},\quad \min_{q_k<l\leqslant n+1} \left\{ p_{l-1} + \max_{k\leqslant j\leqslant l-1} \{-d_j\} \right\} \right\}.$$

The first minimization problem between brackets can again be viewed as an instance of problem P with $u_k = q_k$ and $f_l = p_{l-1} + G_l$. With respect to the second minimization problem, we observe that, for a fixed arbitrary k, the minimum is attained for l as small as possible, i.e. $l = q_k + 1$, since this minimizes both the term p_{l-1}, because of the SPT order, as well as the range over which the maximum is computed. Hence, we are left with calculating

$$p_{q_k} + \max_{k\leqslant j\leqslant q_k} \{-d_j\} = p_{q_k} + \max \left\{ -d_k, \max_{k<j\leqslant q_k} \{-d_j\} \right\}.$$

This boils down to solving the problem

$$\min_{k<j\leqslant q_k} \{d_j\},$$

which is an instance of problem P with $u_k = q_k$ and $f_j = d_j$. From these observations and the fact that, because of monotonicity, the values q_k, $k = 1, 2, \ldots, n$, can be computed in $O(n)$ time, it follows that the time requirement of our algorithm to solve the batching problem is linear. Hence, taking into account the SPT-sorting step, the overall time requirement is again $O(n \log n)$. This constitutes an improvement over the algorithm in [6].

Finally, we note that Brucker et al. [6] use their algorithm for minimizing the maximum lateness as a subroutine in a polynomial procedure for minimizing the maximum cost. Therefore, the $O(\log n)$ improvement obtained here applies to that procedure too.

5. Scheduling customized two-operation jobs

The problem we are addressing in this section may be stated formally as follows. There are n jobs which have to be scheduled on a single machine. Each job j ($j = 1, 2, \ldots, n$) has two operations, namely a *standard*

Table 1
Job data 1

Job i	1	2	3
$p_i^{(1)}$	1	1	1
$p_i^{(2)}$	1	1	$c+2$
d_j	$c+1$	$2c+3$	$2c+3$

operation followed – not necessarily immediately – by a *specific* operation. These operations have processing times $p_j^{(1)}$ and $p_j^{(2)}$, respectively. A set-up time is required before the first standard operation and whenever there is a switch in production from specific to standard operations; two standard operations may be processed consecutively to form a *batch* without a set-up in between. With respect to the way in which standard operations are released (become available) after processing, two schemes are possible: *batch availability*, defined in Section 3, and the alternative *item availability* whereby an operation becomes available immediately after it has been processed. We only analyze the batch availability variant explicitly and give comments as to how the result can be extended to the item availability case.

The model is introduced in [4] (for batch availability) and then analyzed for due-date related criteria in [10]. We note that the problem discussed in [4] for the sum of completion times objective was shown to be equivalent to the, seemingly simpler, problem studied in [7]. In particular, it was shown that the specific (unique) operations can essentially be removed from the problem. If this were also the case for the maximum lateness variants of these problems, then the results of Section 3 could be used directly to solve the problem discussed in this section. Before we proceed with our analysis, it is worthwhile to show that this is not the case. Consider the instance of the two-operation variant in which the set up time is c ($c>0$) and there are three jobs with due dates and operation processing times as shown in Table 1.

It can be easily verified that the problem of Section 3 obtained by omitting the specific operations, has as the unique optimal solution job 1 in the first batch and jobs 2 and 3 in the second batch. The value of this solution is $L_{\max}=0$. However, inserting the specific operations into this schedule (immediately after the corresponding batch) yields a schedule for the two-component problem with lateness equal to $c+4$. It is easy to see that scheduling all the standard operations in one batch, followed by all the specific operations in EDD order, yields a schedule with lateness of 4. Thus, our example suggests that there is no obvious way to translate optimal solutions to the problem in Section 3 into optimal solutions for the problem in this section. This observation and the analysis below seem to lead to the conclusion that the problem in this section is genuinely more complex.

Returning to the two-operation problem, it is shown in [10] that there is an optimal schedule in which jobs complete according to the EDD rule. Thus, the jobs can be re-indexed according to this rule in $O(n\log n)$ time and the problem reduces to one of batching that can be solved using a backward dynamic program with batch insertion [10]. Let G_k denote the minimum overall lateness of a schedule containing jobs $k,k+1,\dots,n$. The initialization is $G_{n+1}=-\infty$ and the recursion for $k=n,n-1,\dots,1$ is

$$G_k=\min_{k<l\leqslant n+1}\left\{\max\left\{\begin{array}{l} s+\displaystyle\sum_{h=k}^{l-1}p_h^{(1)}+\sum_{h=k}^{l-1}p_h^{(2)}+G_l,\\ s+\displaystyle\sum_{h=k}^{l-1}p_h^{(1)}+\max_{k\leqslant j\leqslant l-1}\left\{\sum_{h=k}^{j}p_h^{(2)}-d_j\right\}\end{array}\right\}\right\}. \tag{8}$$

In a similar way as before, it can be verified that $G_{k+1}\leqslant G_k$ for $k=1,2,\dots,n-1$ holds. A standard implementation of the above algorithm requires $O(n^2)$ time, if some preprocessing is used. We now show that the

dynamic programming part can be implemented in $O(n \log n)$ time thus yielding an overall time requirement of $O(n \log n)$.

For the maximum in (8) to be given by the first term, the following needs to hold:

$$G_l \geqslant \max_{k<j\leqslant l-1} \left\{ -\sum_{h=j+1}^{l-1} p_h^{(2)} - d_j \right\},$$

or equivalently

$$G_l - \sum_{h=l}^{n} p_h^{(2)} \geqslant \max_{k\leqslant j\leqslant l-1} \left\{ -\sum_{h=j+1}^{n} p_h^{(2)} - d_j \right\}. \tag{9}$$

Consider an arbitrary index $k \in \{2,3,\ldots,n\}$. Let the subset $I_1^k \subseteq \{k+1, k+2, \ldots, n\}$ contain the indices for which (9) holds. We first explain how we determine I_1^{k-1}. Since the left-hand side value of (9) does not depend on k and

$$\max_{k\leqslant j\leqslant l-1} \left\{ -\sum_{h=j+1}^{n} p_h^{(2)} - d_j \right\} \leqslant \max_{k-1\leqslant j\leqslant l-1} \left\{ -\sum_{h=j+1}^{n} p_h^{(2)} - d_j \right\}$$

it holds that $(I_1^{k-1} \cap \{k+1, k+2, \ldots, n\}) \subseteq I_1^k$. Moreover, the elements of I_1^k which are not in I_1^{k-1} are exactly those $l \in I_1^k$ for which

$$G_l - \sum_{h=l}^{n} p_h^{(2)} < -\sum_{h=k}^{n} p_h^{(2)} - d_{k-1}. \tag{10}$$

Note that the right-hand side of (10) is a constant for fixed k. Hence, if the inequality is satisfied for one or more indices in I_1^k, then these correspond to the *smallest* elements of the set $\{G_l - \sum_{h=l}^{n} p_h^{(2)} \mid l \in I_1^k\}$. This fact can be used to efficiently determine I_1^{k-1}. In our implementation, we make use of a *heap*, which we denote by H_1. Recall that this data structure has the following properties [2]:

(i) the minimum of all values stored in the heap can be retrieved in constant time;
(ii) adding a value to the heap takes $O(\log m)$ time, where m is the number of stored values;
(iii) deleting the minimum value from the heap takes $O(\log m)$ time.

Suppose that heap H_1 contains the values $G_l - \sum_{h=l}^{n} p_h^{(2)}$ for all $l \in I_1^k$. After G_k has been calculated (how this is done efficiently will be shown below), we would like H_1 to contain the values $G_l - \sum_{h=l}^{n} p_h^{(2)}$ for all $l \in I_1^{k-1}$. To achieve this, we first check whether the minimum value is less than the right-hand side of (10). If this is the case, then we delete the minimum from H_1 and we repeat the comparison with the new minimum value. We keep deleting the current minimum value from H_1 until this value becomes at least as large as the right-hand side of (10) or until H_1 is empty. Then we check whether $G_k - \sum_{h=k}^{n} p_h^{(2)}$ is at least as large as the right-hand side of (10). Only if this is the case, do we add $G_k - \sum_{h=k}^{n} p_h^{(2)}$ to H_1. At this point, H_1 contains the values $G_l - \sum_{h=l}^{n} p_h^{(2)}$ for all $l \in I_1^{k-1}$. In parallel to updating H_1, we can keep track of the indices that correspond to its elements.

Let us now turn to the issue of the efficient calculation of G_k. From the definition of I_1^k it follows that we would like to calculate

$$s + \min_{l\in I_1^k} \left\{ \sum_{h=k}^{l-1} p_h^{(1)} + \sum_{h=k}^{l-1} p_h^{(2)} + G_l \right\} \tag{11}$$

and

$$s + \min_{l \in I_2^k} \max_{k \leqslant j \leqslant l-1} \left\{ \sum_{h=k}^{l-1} p_h^{(1)} + \sum_{h=k}^{j} p_h^{(2)} - d_j \right\}, \tag{12}$$

where $I_2^k = \{k+1, k+2, \ldots, n+1\} \setminus I_1^k$.

First consider (12). Suppose $l, i \in I_2^k$ and $l < i$, then

$$\begin{aligned} &\max_{k \leqslant j \leqslant l-1} \left\{ \sum_{h=k}^{l-1} p_h^{(1)} + \sum_{h=k}^{j} p_h^{(2)} - d_j \right\} \\ &\quad \leqslant \max_{k \leqslant j \leqslant l-1} \left\{ \sum_{h=k}^{i-1} p_h^{(1)} + \sum_{h=k}^{j} p_h^{(2)} - d_j \right\} \\ &\quad \leqslant \max_{k \leqslant j \leqslant i-1} \left\{ \sum_{h=k}^{i-1} p_h^{(1)} + \sum_{h=k}^{j} p_h^{(2)} - d_j \right\}. \end{aligned}$$

It follows that the minimum in (12) is attained for the smallest element of I_2^k, which we denote by q_k; we define $q_k = k$ if $I_2 = \emptyset$. Hence, (12) is equivalent to

$$s + \max_{k \leqslant j \leqslant q_k - 1} \left\{ \sum_{h=k}^{q_k-1} p_h^{(1)} + \sum_{h=k}^{j} p_h^{(2)} - d_j \right\}$$

or

$$s - \sum_{h=k}^{q_k-1} p_h^{(1)} - \sum_{h=1}^{k-1} p_h^{(2)} + \max \left\{ \sum_{h=1}^{k} p_h^{(2)} + d_k, - \min_{k < j \leqslant q_k - 1} \left\{ - \sum_{h=1}^{j} p_h^{(2)} + d_j \right\} \right\}.$$

From the discussion about the updating process of heap H_1, it follows that the values q_k are non-decreasing in k. (Also note that keeping track of the values q_k, $k = 1, 2, \ldots, n$, requires overall $O(n)$ time.) Hence, the minimization is an instance of Problem P with $u_k = q_k - 1$ and $f_j = -\sum_{h=1}^{j} p_h^{(2)} + d_j$. It follows that (12) can be calculated for all values of $k = 1, 2, \ldots, n$ together in linear time.

For the efficient calculation of (11), we use a heap H_2 which contains the values $\sum_{h=1}^{l-1} p_h^{(1)} + \sum_{h=1}^{l-1} p_h^{(2)} + G_l$ for all $l \in I_1^k$ and possibly for some $l \in I_2^k$. Note that these values are independent of k. To calculate (11), we simply retrieve the minimum from the heap. If the minimum corresponds to an element of I_2^k, we delete this value from H_2 and retrieve the new minimum. This is repeated until the minimum corresponds to an element of I_1^k or until H_2 is empty. In the latter case the value of (11) is ∞, while in the former case we get the value of (11) by adding s and subtracting $\sum_{h=1}^{k-1} p_h^{(1)} + \sum_{h=1}^{k-1} p_h^{(2)}$.

The time complexity of the above algorithm depends on the number of additions to and deletions from the heaps. For every $l = 1, 2, \ldots, n$, the value $G_l - \sum_{h=l}^{n} p_h^{(2)}$ is added at most once to H_1 and the value $\sum_{h=1}^{l-1} p_h^{(1)} + \sum_{h=1}^{l-1} p_h^{(2)} + G_l$ is added at most once to H_2. (These additions actually occur at the same point in time.) Furthermore, deletion from H_1 and H_2 also occurs at most once for every index. Since the heaps never contain more than n elements, it follows that the total computational effort involving heap operations is $O(n \log n)$.

We have now arrived at the required result: our algorithm solves the batching problem in $O(n \log n)$ time thus yielding an overall time requirement of $O(n \log n)$ time. This constitutes an improvement over the algorithm in [10].

With respect to the item availability case, we note that the problem can be solved using a double recursion dynamic program with block insertion; such a scheme is proposed in [9] and enables us to deploy the approach developed in this section "twice" (in parallel, even) to reduce the overall complexity to $O(n \log n)$.

Acknowledgements

Comments of an Associate Editor and an anonymous referee have improved the presentation of this paper. The authors wish to thank Chris Potts who suggested this research topic. Financial support by the Tinbergen Institute is gratefully acknowledged.

References

[1] A. Aggarwal, J.K. Park, Improved algorithms for economic lot size problems, Oper. Res. 41 (3) (1993) 549–571.
[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman, Data Structures and Algorithms, Addison-Wesley, Reading, MA, 1987.
[3] S. Albers, P. Brucker, The complexity of one-machine batching problems, Disc. Appl. Math. 47 (1993) 87–107.
[4] K.R. Baker, Scheduling the production of components at a common facility, IIE Trans. 20 (1) (1988) 32–35.
[5] P. Brucker, Scheduling Algorithms, Springer, Berlin, 1995.
[6] P. Brucker, A. Gladky, H. Hoogeveen, M.Y. Kovalyov, C.N. Potts, T. Tautenhahn, S. van de Velde, Scheduling a batching machine, J. Schedul. 1 (1998) 31–54.
[7] E.G. Coffman, M. Yannakakis, M.J. Magazine, C. Santos, Batch sizing and job sequencing on a single machine, Ann. Oper. Res. 26 (1990) 135–147.
[8] A. Federgruen, M. Tzur, A simple forward algorithm to solve general dynamic lot sizing models with n periods in $O(n \log n)$ or $O(n)$ time, Manage. Sci. 37 (1991) 909–925.
[9] A.E. Gerodimos, C.A. Glass, C.N. Potts, Scheduling customised jobs on a single machine under item availability, unpublished report, Faculty of Mathematical Studies, University of Southampton, UK (1999).
[10] A.E. Gerodimos, C.A. Glass, C.N. Potts, Scheduling the production of two-operation jobs on a single machine, Euro. J. Oper. Res. 120 (2000) 250–259.
[11] M.Y. Kovalyov, C.N. Potts, Scheduling with batching: a review, Euro. J. Oper. Res. 120 (2000) 228–249.
[12] C.A. Santos, M.J. Magazine, Batching in single operation manufacturing systems, Oper. Res. Lett. 4 (3) (1985) 99–103.
[13] S. van Hoesel, A. Wagelmans, B. Moerman, Using geometric techniques to improve dynamic programming algorithms for the economic lot-sizing problem and extensions, Euro. J. Oper. Res. 75 (1994) 312–331.
[14] A. Wagelmans, S. van Hoesel, A. Kolen, Economic lot sizing: an $O(n \log n)$ algorithm that runs in linear time in the Wagner–Whitin case, Oper. Res. 40 (Supp. 1) (1992) 145–156.
[15] S. Webster, K.R. Baker, Scheduling groups of jobs on a single machine, Oper. Res. 43 (1995) 692–703.