Improved dynamic programs for batching problems with maximum lateness criterion

A P M Wagelmans
Econometric Institute
Erasmus University Rotterdam
PO Box 1738, 3000 DR Rotterdam
The Netherlands

A E Gerodimos

Econometric Institute Report EI–9857/A
November 1998

Abstract

We study a class of scheduling problems involving the maximum lateness criterion and an element of batching. For all the problems that we examine, algorithms appear in the literature which consist of a sorting step to determine an optimal job sequence, followed by a dynamic programming step which determines the optimal batches. In each case, the dynamic program is based on a backward recursion of which a straightfoward 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 length of the first batch. These properties and the use of efficient data structures enable us to exclude partial schedules that cannot lead to an overall optimum early on in the enumeration process. The four problems that we consider are solved in $O(n \log n)$ time; in two occasions, the batching step is actually performed in linear time and the overall complexity is determined by the sorting step.

1 Introduction

The early 1990's saw the emergence of powerful techniques that reduced the time requirement of dynamic programming algorithms for the classic economic lot sizing (ELS) problem [8, 15, 1]. It was subsequently 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.* [14], which are a generalization of the technique used in [15]. 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 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 progamming whose time requirement 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 will encounter frequently when solving the four batching problems. Subsequently, we list these 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 customised two-operation jobs on a single machine under batch availability and we indicate how a similar approach can be applied in case of item availability. Finally, some concluding remarks are given in Section 6.

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 the problems there is an optimal schedule in which jobs complete according to the earliest due date (EDD) rule, while 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, after the improvement of the time requirement of the dynamic programming step, the sorting step actually becomes the overall bottleneck.

Our approach makes implicit use of the – realistic – assumption that the processing time of a batch does not decrease when we include an additional job. The latter definition is general enough to contain so-called sum-batch problems where the total processing time of a batch equals the sum of processing times of all the jobs in the batch, as well as max-batch problems in which the job with the largest processing time determines the total batch processing time. In either case, the maximum lateness of a schedule does not decrease (and may increase) if an additional job is included in it.

In our approach to solving the dynamic programs, we will encounter problems of the following type:

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

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

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

a. $u_k \leq u_{k+1}$ *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,* $k = 1, 2, \ldots, n$

Clearly, because of conditions (c) and (d), 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 will show, however, that a linear time bound is possible. To this end, we will use a special data structure. 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 $t(1) < t(2) < \ldots < t(r)$ and $f_{t(1)} > f_{t(2)} > \ldots > f_{t(r)}$. Moreover, $f_{t(r)} = m_k$. Hence, given the subsequence, the desired minimum is immediately available.

We will keep track of the subsequence by storing its elements in decreasing order in a list (i.e, f_{k+1} is the element at the top). This list has the property that elements at the bottom can only be deleted, while at the top elements can be deleted and added. Hence, it could be viewed as a combination of a stack and a queue, and it can easily be implemented such that each deletion and each addition requires constant time (see [2]).

To see why this data structure is convenient, first notice the following: if for a given $k \geq 2$, 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 follows from the fact that there exists some value in the subsequence, say $t(i)$, such that $k+1 \leq t(i) < l$ and $f_{t(i)} < f_l$. Now if $l \in \{k, k+1, \ldots, u_{k-1}\}$, then also $t(i) \in \{k, k+1, \ldots, u_{k-1}\}$ and this implies that l should still be excluded from the subsequence. This means that when for a certain $k \geq 2$ 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 $u_{k-1} \leq u_k$, 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 $f_k \leq f_{t(i)}$. Finally, we add k at the top of the list. In case $u_{k-1} = k-1$ the list will be empty after the deletion operations and no element will be 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 can not be bounded nicely for each individual time the updating process is carried out. 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.

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*. 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 [13]. 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 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 [16] 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

$$G_k = \min_{k<l\leq n+1} \{ \max\{(s + a_k - a_l) + G_l,\ (s + a_k - a_l) - d_k\}\}, \qquad (1)$$

where $a_k = \sum_{h=k}^{n} p_h$ for $k = 1, \ldots, n$. Here l denotes the first job in the second batch of the schedule. Since this batch starts at time $s + a_k - a_l$, 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 [16], 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 (1), or common sense reasoning, it follows that $G_{l+1} \leq G_l$ for every $l \leq n-1$. Hence, if $G_{l+1} \geq -d_k$ for some $k \in \{1, \dots, n\}$, then also $G_l \geq -d_k$. We now define q_k as the largest job index l in $\{k+1, \dots, n\}$ such that $G_l \geq -d_k$; if no such index exists, we define $q_k = k$. (Note that, because of the EDD order, $q_{k+1} \geq q_k$ holds for every $k \leq n-1$.)

From the above observations, it follows that for all indices l satisfying $k < l \leq q_k$ the maximum in (1) is given by the first term, whereas when $q_k < l \leq n+1$, then the maximum is given by the second term. Now (1) can be rewritten as

$$G_k = \min\{\min_{k<l\leq q_k}\{s + a_k - a_l + G_l\}, \min_{q_k<l\leq n+1}\{s + a_k - a_l - d_k\}\}.$$

Note that 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.

The remaining task is to compute

$$s + a_k + \min_{k<l\leq q_k}\{-a_l + G_l\} \tag{2}$$

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 $f_l = -a_l + G_l$. Hence, it takes overall $O(n)$ time to calculate the minima given by (2).

Since the parameters a_l, $l = 1, 2, \dots, , n$, and, because of monotonicity, the values q_k, $k = 1, 2, \dots, n$, 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 [16].

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 ($j = 1, \dots, n$) 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 $b \geq n$ 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

$$G_k = \min_{k<l\leq n+1} \{\max\{p_{l-1} + G_l,\ p_{l-1} + \max_{k\leq j\leq l-1} \{-d_j\}\}\}, \tag{3}$$

where l should again be interpreted as the first job of the second batch, which starts when the first batch completes. By assumption, this happens when the longest job $(l-1)$ of the first batch completes. The recursion formula reflects the fact that the overall lateness either occurs in the second batch or later, or in the first batch.

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. Again it can easily be verified that $G_{l+1} \leq G_l$ for every $l \leq n-1$. Hence, if $G_{l+1} > \max_{k<j\leq l}\{-d_j\}$ for some $k \in \{1, \ldots, n\}$, then

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

It follows that if there exist jobs l in $\{k+1, k+2, \ldots, n\}$ satisfying $G_l \geq \max_{k<j\leq l-1}\{-d_j\}$, then these jobs are exactly the jobs numbered $k+1$ up to some index q_k. 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 (3) can now be rewritten as

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

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 every k the minimum is attained for l as small as possible, i.e., $l = q_k + 1$, because this minimizes both the term p_{l-1}, because of the SPT order, as well as the range over which the maximum is taken. Hence, we are left with calculating

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

This boils down to solving the problem

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

which is an instance of (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, because of the 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 improvement obtained here affects the overall time requirement of this procedure.

5 Scheduling customised 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* 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 contiguously 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 a paper by [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 would apply. Before we proceed with our analysis, it seems therefore appropriate 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.

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	$c+1$	$2c+3$	$2c+3$

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. This establishes that the inclusion of the problem in Section 3 in the corresponding two-operation problem is proper.

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 $k, k+1, \ldots, n$. The initialization is $G_{n+1} = -\infty$ and the recursion for $k = n, n-1, \ldots, 1$ is

$$G(k) = \min_{k<l\leq n+1} \{\max \left\{ \begin{array}{l} s + \sum_{h=k}^{l-1} p_h^{(1)} + \sum_{h=k}^{l-1} p_h^{(2)} + G(l), \\ \\ s + \sum_{h=k}^{l-1} p_h^{(1)} + \max_{k\leq j\leq l-1}\{\sum_{h=k}^{j} p_h^{(2)} - d_j\} \end{array} \right\}. \tag{4}$$

Again, $G_{k+1} \leq G_k$ for $k = 1, 2, \ldots, n-1$ 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 (4) to be given by the first term, the following needs to hold:

$$G(l) \geq \max_{k<j\leq l-1}\{-\sum_{h=j+1}^{l-1} p_h^{(2)} - d_j\},$$

or equivalently

$$G(l) - \sum_{h=l}^{n} p_h^{(2)} \geq \max_{k\leq j\leq l-1}\{-\sum_{h=j+1}^{n} p_h^{(2)} - d_j\}. \qquad (5)$$

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 (5) holds. We will first explain how we determine I_1^{k-1}. Since the left-hand-side value of (5) does not depend on k and

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

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

$$G(l) - \sum_{h=l}^{n} p_h^{(2)} < -\sum_{h=k}^{n} p_h^{(2)} - d_{k-1}. \qquad (6)$$

Note that the right-hand-side of (6) 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 a this data structure has the following properties ([2]):

1. (i) the minimum of all values stored in the heap can be retrieved in constant time,

2. (ii) adding a value to the heap takes $O(\log m)$ time, where m is the number of stored values,

3. (iii) deleting a 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 (6). 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 the as large as the right-hand-side of (6) 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 (6). Only if this is the case, 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

$$s+\min_{l \in I_1^k}\{\sum_{h=k}^{l-1} p_h^{(1)}+\sum_{h=k}^{l-1} p_h^{(2)}+G(l)\} \tag{7}$$

and

$$s+\min_{l \in I_2^k} \max_{k \leq j \leq l-1}\{\sum_{h=k}^{l-1} p_h^{(1)}+\sum_{h=k}^{j} p_h^{(2)}-d_j\}, \tag{8}$$

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

First consider (8). Suppose $l, i \in I_2^k$ and $l<i$, then

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

$$\max_{k \leq j \leq l-1}\{\sum_{h=k}^{i-1} p_h^{(1)}+\sum_{h=k}^{j} p_h^{(2)}-d_j\} \leq$$

$$\max_{k \leq j \leq i-1}\{\sum_{h=k}^{i-1} p_h^{(1)}+\sum_{h=k}^{j} p_h^{(2)}-d_j\}.$$

It follows that the minimum in (8) 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, (8) is equivalent to

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

or

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

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 (P) with $u_k = q_k - 1$ and $f_j = -\sum_{h=1}^{j} p_h^{(2)} + d_j$. It follows that (8) can be calculated for all values of $k = 1, 2, \ldots, n$ together in linear time.

For the efficient calculation of (7), 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 (7), we simply retrieve the minimum of 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 (7) is ∞, while in the former case we get the value of (7) 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, also deletion from H_1 and H_2 occurs at most once for every index. Since the heaps never contain more than n elements, it follows that the computational effort involving heap operations is $O(n \log n)$.

Finally, we note that partial sums such as $\sum_{h=k}^{l-1} p_h^{(1)}$ can be replaced by $\sum_{h=k}^{n} p_h^{(1)} - \sum_{h=l}^{n} p_h^{(1)}$. Partial sums of the latter type can be calculated in linear time in a preprocessing step.

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 $O(n \log n)$.

6 Concluding Remarks

We have presented improved dynamic programming algorithms for a class of scheduling problems involving the maximum lateness criterion and an element of batching. A question that arises is whether insights gained from this study can help to reduce the time requirement of algorithms for other more complicated models involving the batching of jobs that belong to different *families*. Our analysis seems to indicate that the answer is negative because, in those problems, the tasks of sequencing and batching can only be separated within each family but not at the overall level.

Acknowledgements

The authors would like to thank Chris Potts who suggested this research topic. Financial support by the Tinbergen Institute is gratefully acknowledged.

References

[1] A Aggarwal and J K Park. Improved algorithms for economic lot size problems. *Operations Research*, 41(3):549–571, 1993.

[2] A V Aho, J E Hopcroft, and J D Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1987.

[3] S Albers and P Brucker. The complexity of one-machine batching problems. *Discrete Applied Mathematics*, 47:87–107, 1993.

[4] K R Baker. Scheduling the production of components at a common facility. *IIE Transactions*, 20(1):32–35, 1988.

[5] P Brucker. *Scheduling Algorithms*. Springer Verlag, Berlin, 1995.

[6] P Brucker, A Gladky, H Hoogeveen, M Y Kovalyov, C N Potts, T Tautenhahn, and S van de Velde. Scheduling a batching machine. *Journal of Scheduling*, 1:31–54, 1998.

[7] E G Coffman, M Yannakakis, M J Magazine, and C Santos. Batch sizing and job sequencing on a single machine. *Annals of Operations Research*, 26:135–147, 1990.

[8] A Federgruen and 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. *Management Science*, 37:909–925, 1991.

[9] A E Gerodimos, C A Glass, and C N Potts. Scheduling customised jobs on a single machine under item availability. submitted to *IIE Transactions in Operations Research*, 1998.

[10] A E Gerodimos, C A Glass, and C N Potts. Scheduling the production of two-operation jobs on a single machine. to appear in a Feature Issue of *European Journal of Operational Research* dedicated to the 15th EURO Summer Institute, ESI XV, on Production Scheduling, St Vincent, Aosta Valley, Italy, September 12-26, 1997, 1998.

[11] M Y Kovalyov and C N Potts. Scheduling with batching: A review. to appear in a Feature Issue of *European Journal of Operational Research* dedicated to the 15th EURO Summer Institute, ESI XV, on Production Scheduling, St Vincent, Aosta Valley, Italy, September 12-26, 1997, 1998.

[12] C N Potts and L N Van Wassenhove. Integrating scheduling with batching and lot sizing: A review of algorithms and complexity. *Journal of the Operational Research Society*, 43(5):395–406, 1992.

[13] C A Santos and M J Magazine. Batching in single operation manufacturing systems. *Operations Research Letters*, 4(3):99–103, 1985.

[14] S van Hoesel, A Wagelmans, and B Moerman. Using geometric techniques to improve dynamic programming algorithms for the economic lot-sizing problem and extensions. *European Journal of Operational Research*, 75:312–331, 1994.

[15] A Wagelmans, S van Hoesel, and A Kolen. Economic lot sizing: An $O(n \log n)$ algorithm that runs in linear time in the Wagner-Whitin case. *Operations Research*, 40(Supp. 1):145–156, 1992.

[16] S Webster and K R Baker. Scheduling groups of jobs on a single machine. *Operations Research*, 43:692–703, 1995.