versão impressa ISSN 0101-7438 / versão online ISSN 1678-5142

EFFICIENT POLYNOMIAL ALGORITHMS FOR SPECIAL CASES OF WEIGHTED EARLY/TARDY SCHEDULING WITH RELEASE DATES AND A COMMON DUE DATE

Jorge M. S. Valente *
Rui A. F. S. Alves
Faculdade de Economia
Universidade do Porto
Porto – Portugal
jvalente@fep.up.pt

* *Corresponding author*/autor para quem as correspondências devem ser encaminhadas

Recebido em 02/2003; aceito em 11/2003 após 1 revisão
Received February 2003; accepted November 2003 after one revision

Abstract

In this paper we consider a single machine scheduling problem with integer release dates and a common due date. The objective is to minimise the weighted sum of the jobs' earliness and tardiness costs. We present an efficient polynomial algorithm for the unit processing time case. We also show how to calculate, for the general case, the minimum non-restrictive due date.

Keywords: scheduling; early/tardy; release dates.

Resumo

Neste artigo consideramos um problema de sequenciamento com um único processador no qual existem datas de disponibilidade inteiras e uma data de entrega comum. O objectivo consiste em minimizar a soma ponderada dos custos de posse e de atraso. Um algoritmo polinomial é apresentado para o caso no qual os tempos de processamento são iguais a um. É também desenvolvido um algoritmo que permite determinar, para o caso geral, o menor valor não restritivo da data de entrega.

Palavras-chave: sequenciamento; *early/tardy*; datas de disponibilidade.

1. Introduction

In this paper we consider the following scheduling problem. A set of *n* independent jobs $\{J_1, J_2, ..., J_n\}$, each with a possibly different integer release date r_j, processing time p_j, and a common integer due date *d*, has to be scheduled without preemptions on a single machine that can handle at most one job at a time. The objective is to minimise $\sum_{j=1}^{n}\{h(d-C_j)^+ + w(C_j-d)^+\}$, where C_j is the completion time of J_j and *h* and *w* are, respectively, the earliness and tardiness cost per unit time. The earliness and tardiness costs are therefore different, although they are identical for all jobs. In the classification scheme proposed by Lawler, Lenstra, Rinnooy Kan & Shmoys (1993), this problem can be represented as $1|d_j = d,\ r_j|\sum\{h(d-C_j)^+ + w(C_j-d)^+\}$. Scheduling models with both earliness and tardiness costs are particularly appealing, since they are compatible with the philosophy of just-in-time production. The earliness and tardiness costs are allowed to differ (even though they are assumed to be identical for all jobs), and the model is also made more realistic by the existence of different release dates, since in most production settings the orders are released to the shop floor over time (and not all simultaneously). Therefore, the problem considered has several potential practical applications.

To the best of our knowledge, this specific problem has not yet been analysed in the literature, although models with identical release dates, as well as some related problems with different release dates, have been previously considered. The problem with identical release dates and *h*=*w*=1, i.e., $1|d_j = d|\sum|C_j - d|$, has been considered by several authors. Kanet (1981) and Bagchi, Sullivan & Chang (1986) presented O(*n*log *n*) algorithms for solving the non-restrictive due date version of this problem. The common due date is non-restrictive when it does not constrain the optimal schedule cost. The restrictive case, however, has been proved NP-hard by Hall, Kubiak & Sethi (1991). Sundararaghavan & Ahmed (1984) present a branch-and-bound (B&B) algorithm and a heuristic procedure for the special case of the restrictive version in which all sequences must start at time 0. This special case was also considered by Bagchi, Sullivan & Chang (1986) and Szwarc (1989). Bagchi, Sullivan & Chang (1986) propose a branching procedure, while Szwarc (1989) develops several dominance conditions that are used in a B&B algorithm. Szwarc (1989) also presents a sufficient condition for an optimal sequence to start at time 0. The general restrictive version has also been analysed by several authors. Baker & Chadowitz (1989) presented a modified version of the heuristic proposed by Sundararaghavan & Ahmed (1984). Hall, Kubiak & Sethi (1991) develop several dominance conditions and a dynamic programming algorithm that obtains an optimal solution in pseudopolynomial time. Ventura & Weng (1995) improve the efficiency of this algorithm by remarking that some of its subroutines are unnecessary and can therefore be eliminated. Hoogeveen, Oosterhout & Van de Velde (1994) present a B&B algorithm that uses lagrangean relaxation to calculate both lower and upper bounds.

The identical release dates version of our problem, i.e., $1|d_j = d|\sum\{h(d-C_j)^+ + w(C_j-d)^+\}$, has also been previously considered. Bagchi, Chang & Sullivan (1987) present an O(*n*log *n*) algorithm for solving the non-restrictive case. The restrictive case is necessarily NP-hard, since even the problem with $h=w=1$ is NP-hard, as noted above. The special case of the restrictive version in which all sequences must start at time 0 was considered by Bagchi, Chang & Sullivan (1987) and Baker & Chadowitz (1989). Bagchi, Chang & Sullivan (1987)

present some dominance properties and a branching algorithm, while Baker & Chadowitz (1989) propose a heuristic method. The general case of the restrictive version was analysed by Mondal & Sen (2001), who present an algorithm that obtains an optimal solution through the use of a graph search space. The computational results show that this approach is faster than both dynamic programming or B&B algorithms that use a depth-first search strategy.

Models with unequal release dates have been addressed in Nandkeolyar, Ahmed & Sundararaghavan (1993), Sridharan & Zhou (1996), Bank & Werner (2001) and Cheng, Chen & Shakhlevich (2002). Nandkeolyar, Ahmed & Sundararaghavan (1993) consider the single machine problem with the objective of minimising $\sum_{j=1}^{n} c_j \left| d_j - C_j \right|$, where c_j and d_j are, respectively, the cost per unit time and the due date of J_j. They present several heuristics developed using a novel approach. These heuristics were developed in a modular fashion, with different modules providing lookahead features, sequence generation and the determination of the specific schedule. The performance of the heuristics, as well as that of the different modules, was tested on several classes of problems. Sridharan & Zhou (1996) present a heuristic based on a decision theory approach for a more general problem where the cost per unit time is allowed to be different according to whether the job is early or tardy. Bank & Werner (2001) consider a model with unrelated parallel machines, a common due date and earliness and tardiness costs that may differ between jobs. They derive several structural properties that are useful when searching for an approximate solution and present several constructive and iterative heuristics. Cheng, Chen & Shakhlevich (2002) analyse a model where all jobs have a common due date that needs to be determined. The problem is to determine both a due date and a schedule in order to minimize a total penalty that depends on the earliness, the tardiness and the due date. They show that the problem is strongly NP-hard and give an efficient algorithm that finds an optimal due date and schedule when either the job sequence is predetermined or all jobs have identical processing times. They also present three approximation algorithms for both the general and some special cases. It should be pointed out that there are a large number of papers considering earliness and tardiness penalties. For more information on problems with earliness and tardiness costs, the interested reader is referred to Baker & Scudder (1990), who present a comprehensive survey of early/tardy scheduling.

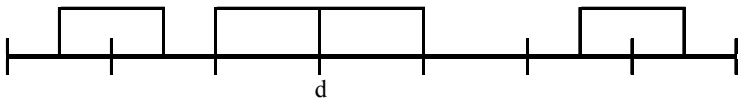
The remainder of this paper is organized as follows. In section 2 we consider the special case of our problem where all jobs have unit processing times, i.e., $1|d_j = d,\ p_j = 1,\ r_j|\sum\{h(d-C_j)^+ + w(C_j-d)^+\}$. We present an algorithm that obtains an optimal solution in O($n$log n) time, therefore establishing the polynomial solvability of this particular case. In section 3 we consider the general problem $1|d_j = d,\ r_j|\sum\{h(d-C_j)^+ + w(C_j-d)^+\}$ and develop an algorithm, again with complexity O($n$log n), for determining the minimum non-restrictive value of the common due date d. In this section we remark that the due date is non-restrictive when the optimal schedule for the non-restrictive version of the problem with equal release dates is feasible. This implies that the non-restrictive version of our problem can be solved in polynomial time using the algorithm proposed by Bagchi, Chang & Sullivan (1987) for the problem with identical release dates. The restrictive version, however, is then NP-hard, since even the restrictive version of $1|d_j = d|\sum\left|C_j - d\right|$ is NP-hard, as noted above. Finally, we provide some concluding remarks in section 4.

2. An algorithm for the problem with unit processing times

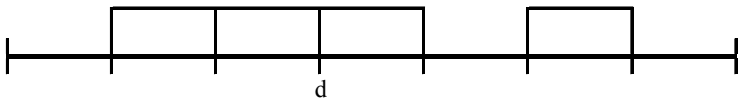
In this section we propose an O($n$log n) algorithm for the problem with unit processing times. Several lemmas and theorems that characterize the structure of an optimal solution are first developed. The algorithm then simply schedules the jobs in such a way that the lemmas and theorems are satisfied (hence optimally).

Lemma 1 There exists an optimal sequence where each C_j is integer.

Proof. Any feasible schedule with non integer C_j's can be transformed into a feasible sequence, of lower or equal cost, where all C_j's are integer. Starting from d and scanning left, take the first job, if any, with a non integer $C_j < d$ such that there exists idle time to the right of that job. Move that job to the right until it is blocked by another job or it completes at an integer time, whichever occurs first. Any such movement will decrease the cost of the schedule. Repeat until no such jobs exist. Perform a similar scan to the right of d, this time moving jobs to the left (such a move is always feasible because each job has an integer r_j and we stop as soon as an integer start time is reached, if the job is not blocked before). Again, any such movement decreases the schedule cost (see Figure 1). At this time, at most one group of jobs performed consecutively with no idle time in between (or possibly just a single job) starts and completes at non integer times that encompass d. Let $\boldsymbol{A}$ and $\boldsymbol{B}$ denote the sets of jobs in that group that complete after and before d, respectively, and let $|\boldsymbol{X}|$ denote the cardinality of set $\boldsymbol{X}$. If $w|\boldsymbol{A}| > h|\boldsymbol{B}|$, move the block backwards until an integer start time is reached. The earliness of each job in $\boldsymbol{B}$ will increase by the amount of backwards movement but the tardiness of each job in $\boldsymbol{A}$ will decrease by that same amount. Since $w|\boldsymbol{A}| > h|\boldsymbol{B}|$ the decrease in the weighted tardiness exceeds the increase in weighted earliness, so the schedule cost will decrease. If $w|\boldsymbol{A}| < h|\boldsymbol{B}|$, move the block forward instead. If $w|\boldsymbol{A}| = h|\boldsymbol{B}|$, move either forward or backward. The new schedule has all integer C_j's and its cost is lower or the same.■



a) schedule with non-integer Cj's



b) lower cost schedule with all integer Cj's

Figure 1 – Lemma 1 example

This lemma is useful because it allows us to focus on unit time slots that begin at integer times (even though all parameters are integers, jobs are allowed to start at non integer times), since there exists an optimal sequence where jobs are scheduled in *n* of those slots. This result also indicates that the problem could be formulated as a weighted bipartite matching problem, and therefore solved in $O(n^3)$ time, but a more efficient approach is possible. The next lemma identifies the best possible time slots.

Lemma 2 Any feasible sequence in which the jobs are scheduled in the *n* consecutive time slots in the time range $[\, d - \left\lceil n\frac{w}{h+w} \right\rceil, d + \left\lfloor n\frac{h}{h+w} \right\rfloor\,]$ is also optimal.

Proof. Any slot not in this range has a cost that is at least as high as the cost of the most expensive slot in the range. The cost of any slot inside the time range is not higher than $n\frac{hw}{h+w}$, since the most expensive early and tardy slots have a cost of $h(\left\lceil n\frac{w}{h+w} - 1 \right\rceil)$ and $w(\left\lfloor n\frac{h}{h+w} \right\rfloor)$, respectively. Any slot not in the time range has a cost of at least $n\frac{hw}{h+w}$, since the least expensive early and tardy slots have a cost of $h(\left\lceil n\frac{w}{h+w} \right\rceil)$ and $w(\left\lfloor n\frac{h}{h+w} \right\rfloor + 1)$, respectively.■

Assume, for the remainder of this section, that jobs have been renumbered in non-decreasing order of their release dates. Let EC_j be the earliest possible completion time of job *j* when jobs are considered for processing in increasing order of their index *j*. Let $EC = EC_n$ denote the earliest possible completion time of the job with the largest index. The next lemma indicates when it is feasible to schedule the jobs in the best possible slots.

Lemma 3 It is possible to schedule the jobs in the *n* consecutive time slots in the time range $[\, d - \left\lceil n\frac{w}{h+w} \right\rceil, d + \left\lfloor n\frac{h}{h+w} \right\rfloor\,]$ if and only if $EC \le d + \left\lfloor n\frac{h}{h+w} \right\rfloor$.

Proof. If $EC > d + \left\lfloor n\frac{h}{h+w} \right\rfloor$ then obviously at least one job cannot be completed up to $d + \left\lfloor n\frac{h}{h+w} \right\rfloor$. If $EC \le d + \left\lfloor n\frac{h}{h+w} \right\rfloor$, any job with $EC_j > d - \left\lceil n\frac{w}{h+w} \right\rceil$ can be scheduled to complete at its EC_j, while the remaining jobs can be arbitrarily assigned to the still empty time slots in the optimal range. That assignment is clearly feasible, since we are delaying the start time of each of those jobs.■

The next theorem provides a condition for classifying a due date as restrictive or non-restrictive.

Theorem 4 The due date d is non-restrictive when $d \geq EC - \left\lfloor n\frac{h}{h+w} \right\rfloor$.

Proof. Lemma 2 identifies the best possible schedule. From lemma 3, that schedule is only feasible when $d \geq EC - \left\lfloor n\frac{h}{h+w} \right\rfloor$.■

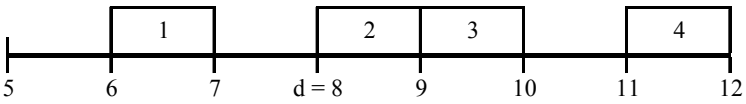
The next lemmas provide further characteristics of an optimal solution that will be used in the algorithm.

Lemma 5 If $EC > d + \left\lfloor n\frac{h}{h+w} \right\rfloor$, all jobs will be scheduled in the time range $[d - \left\lceil n\frac{w}{h+w} \right\rceil$, $EC]$ in an optimal sequence.

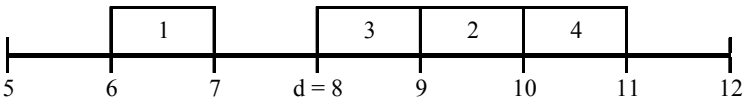
Proof. It is clear that all slots in the range have a lower cost than any slot that starts at or later than EC. Also we can once again schedule jobs with $EC_j > d - \left\lceil n\frac{w}{h+w} \right\rceil$ to complete at their EC_j, while the remaining jobs can be arbitrarily assigned to the still empty time slots in the range $[\, d - \left\lceil n\frac{w}{h+w} \right\rceil, d + \left\lfloor n\frac{h}{h+w} \right\rfloor]$, thereby decreasing their cost.■

Lemma 6 There exists an optimal schedule in which all jobs with $EC_j \geq d$ are scheduled to complete at their EC_j.

Proof. Consider the slots with a completion time equal to EC_j, for $EC_j \geq d$. If a job with $EC_j \geq d$ is completed later than its EC_j, and the slot with a completion time equal to EC_j is free, one can simply move that job into this slot (thereby decreasing its cost, since it will be completed earlier). Also, any feasible schedule in which any slot with a completion time equal to EC_j, for $EC_j \geq d$, is occupied by a job with $EC_j < d$ (an offending job) can be converted into an equal cost sequence in which all such slots are filled with jobs with $EC_j \geq d$. For each of those slots, we simply swap any offending job with the job whose EC_j is equal to that slot's completion time. These swaps do not change the schedule cost and feasibility is maintained (the release date of the offending job in not larger than that of the job with which it is swapped, since its EC_j is lower, so it can also be feasibly scheduled in its destination slot). After all such swaps are performed, only jobs with $EC_j \geq d$ use those slots, though they are not necessarily in EC_j order (jobs are considered in increasing order of their indexes when calculating the EC_j's, so some jobs can feasibly be scheduled before their EC_j). If that is the case, reordering the jobs according to their EC_j will not alter cost nor feasibility. Therefore, all jobs with $EC_j \geq d$ can be optimally scheduled to complete at their EC_j. An example is presented in Figure 2; assume $d = 8$, $EC_1 = 4$, $EC_2 = 5$, $EC_3 = 9$ and $EC_4 = 11$.■



a) initial schedule



b) schedule with all jobs with ECj = d finishing at their ECj

Figure 2 – Lemma 6 example

Lemma 6 assigns optimal slots for jobs with $EC_j \geq d$, so all that remains is to optimally schedule the remaining jobs in the available slots. The next lemma considers those jobs.

Lemma 7 Given that jobs with $EC_j \geq d$ are scheduled according to lemma 6, the remaining jobs should be scheduled in the available slots that are closest to d (i.e., the lowest cost slots).

Proof. We simply need to prove that such a schedule is feasible, since it's clearly optimal. Let $|\boldsymbol{B}|$ be the number of jobs with $EC_j < d$ (and therefore of necessary slots). The earliest possible completion times of the available slots that are closest to d are $d-1$, $d-2$,…, $d-|\boldsymbol{B}|$. The latest possible EC_j's of the remaining jobs are also $d-1$, $d-2$,…, $d-|\boldsymbol{B}|$, so they can be feasibly assigned to the least cost slots. Since any other case involves later slots and/or earlier EC_j's, it's always feasible to schedule the remaining jobs in the least cost available slots. An easy way for an algorithm to ensure feasibility is to simply consider jobs in decreasing order of their EC_j.■

We now present an algorithm that solves the problem with unit processing times. The algorithm schedules the jobs in such a way that the previous lemmas are satisfied (hence optimally). The algorithm uses a min heap of free time slot ranges and their associated minimum cost (the cost of the best slot in that range), which serves as the key for pushing and popping elements from the heap.

Algorithm 1

Step 1: Sort and renumber jobs in non-decreasing order of r_j.

Step 2: Calculate EC_j for all jobs.

Step 3: If $EC < d + \left\lfloor n\frac{h}{h+w} \right\rfloor$, push range $[EC, d + \left\lfloor n\frac{h}{h+w} \right\rfloor]$ on heap.

Step 4: For each job, in decreasing order of EC_j, do:

If $EC_j \geq d$

schedule j to complete at its EC_j;

if $j > 1$, push range $[EC_{j-1}, EC_j - 1]$ on heap;

Else

schedule j to complete at its EC_j or at the best available free time slot on the heap (whichever has a lower cost; ties can be broken arbitrarily); in the latter case update the range that included that slot and re-insert it on the heap;

if $j > 1$, push on heap:

range $[EC_{j-1}, EC_j - 1]$ if j completes at its EC_j;

range $[EC_{j-1}, EC_j]$ if j was scheduled at the best available free time slot on the heap.

In the previous algorithm ranges are obviously only pushed on the heap when the upper limit is higher than the lower limit. Updating a time range that ends at or before d simply involves decreasing its finish time by one time unit (thereby eliminating its previously best slot) and increasing its cost by h. Similarly, updating a time range that starts at or after d consists of increasing its start time by one time unit and increasing its cost by w. Only one range that contains d as an interior point can be generated. When such a range is updated, it's divided into the two separate ranges that result from eliminating the time slot which finishes at d. Step 1 takes O($n$log n) time and Step 2 O(n) time. Step 3 can be done in constant time. In Step 4, the For loop is executed n times. At each iteration pushing or popping the heap takes O(log n) time and scheduling the job and updating time ranges (when necessary) takes O(1) time. Therefore, the complexity of the algorithm is O($n$log n).

Theorem 8 Algorithm 1 generates an optimal schedule.

Proof. The theorem follows from the previous lemmas. Jobs with $EC_j \geq d$ are scheduled as in lemma 6. The algorithm pushes all available time slots with completion time not later than $\max(EC, d + \left\lfloor n\frac{h}{h+w} \right\rfloor)$ on the heap and jobs with $EC_j < d$ are scheduled on the best of those slots, as established in lemma 7. Note that when $EC \leq d + \left\lfloor n\frac{h}{h+w} \right\rfloor$, jobs with $EC_j \geq d$ are scheduled to finish inside the optimal range $[d - \left\lceil n\frac{w}{h+w} \right\rceil, d + \left\lfloor n\frac{h}{h+w} \right\rfloor]$. The remaining jobs will also be scheduled inside this range, since the algorithm will push its slots into the heap (note that range $[EC, d + \left\lfloor n\frac{h}{h+w} \right\rfloor]$ is pushed on the heap).■

Table 1 – Algorithm 1 example

Index	1	2	3	4	5
r_j	0	3	3	6	8

In Table 1 we present an example for Algorithm 1; assume $d = 7$, $h = 2$ and $w = 1$. The jobs are already renumbered in non-decreasing order of r_j. In step 2 the following EC_j's are calculated: $EC_1 = 1$; $EC_2 = 4$; $EC_3 = 5$; $EC_4 = 7$ and $EC_5 = EC = 9$. The due date is in this case non-restrictive, since $EC \le d + \left\lfloor n\frac{h}{h+w} \right\rfloor$ ($9 \le 7 + 3$). In step 3 the range [9, 10] (cost: 3) is pushed on the heap. In step 4, the jobs are scheduled in decreasing EC_j order (which, given the way the EC_j's are calculated, is also decreasing index order). Since $EC_5 \ge d$, job 5 is scheduled in the slot [8, 9] and range [7, 8] (cost: 1) is pushed on the heap. Job 4 is then considered, and we have $EC_4 = d$, so job 4 is scheduled in the slot [6, 7] and range [5, 6] (cost: 2) is pushed on the heap. Job 3 is the next job to be scheduled, and $EC_3 < d$. If job 3 is scheduled to complete at $EC_3 = 5$ its cost will be 4; if it is scheduled in the best slot available on the heap (slot [7, 8]) its cost is 1. Therefore, job 3 is scheduled in the slot [7, 8] and range [4, 5] (cost: 4) is pushed on the heap. Jobs 2 and 1 will then be scheduled in the slots [5, 6] and [9, 10], respectively. The algorithm schedules the jobs in the optimal range $[\, d - \left\lceil n\frac{w}{h+w} \right\rceil, d + \left\lfloor n\frac{h}{h+w} \right\rfloor]$, in this case [5, 10].

3. Calculating the minimum non-restrictive common due date

With different release dates, the due date d will be non-restrictive when the optimal schedule for the non-restrictive version of the problem with equal release dates is feasible, since clearly no better schedule can be generated. Therefore, we must find the minimum value of the common due date d for which that schedule is feasible. Throughout this section assume that the jobs have been renumbered in shortest processing time (SPT) order, i.e., in non-decreasing order of p_j. An optimal schedule for the non-restrictive version of the problem with equal release dates can be determined by the following procedure presented by Bagchi, Chang & Sullivan (1987). Let $\boldsymbol{B}$ be a sequence of jobs to be scheduled without idle time such that the last job in $\boldsymbol{B}$ is completed at d. Let $\boldsymbol{A}$ be a sequence of jobs to be scheduled without idle time such that the first job in $\boldsymbol{A}$ starts at d. An optimal schedule for the problem with identical release dates consists of $\boldsymbol{B}$ followed by $\boldsymbol{A}$, given that those sets are generated by the following rule: assign jobs, in their index order, to the beginning of $\boldsymbol{B}$ if $h|\boldsymbol{B}| < w(|\boldsymbol{A}| + 1)$, and to the end of $\boldsymbol{A}$ otherwise. The minimum non-restrictive common due date for the problem with equal release dates, which will be denoted as $\Delta_{r=0}$, is then $\Delta_{r=0} = \sum_{j \in B} p_j$.

We will now consider the minimum non-restrictive due date when different release dates are allowed. If all jobs shared a common release date r, the smallest non-restrictive common due date would simply be $\Delta_{r=0} + r$. If jobs have different release dates, and also different

processing times, then we could determine the start time of each job in the schedule generated by Bagchi, Chang & Sullivan's procedure (assuming all release dates equal to 0) and calculate the maximum violation of a release date (i.e., the maximum positive difference between a job's release date and its start time in Bagchi, Chang & Sullivan's schedule). The minimum non-restrictive common due date could then be obtained by adding that maximum violation to $\Delta_{r=0}$. When processing times occur more than once, the situation is more complicated. Since processing times are not unique, ties occur when renumbering the jobs in non decreasing order of p_j, and several different Bagchi, Chang & Sullivan schedules may be generated, each leading to a possibly different maximum violation of a release date. Therefore, when renumbering jobs, we need to break ties in such a way that the resulting Bagchi, Chang & Sullivan schedule minimizes the maximum violation of a release date. The following algorithm generates the minimum non-restrictive value of the common due date *d* (denoted by Δ) when release dates are allowed to be different, and processing times may occur more than once. Let p_A and p_B, respectively, be the sum of the processing times of the jobs currently assigned to ***A*** and ***B***. Jobs are added to the beginning of ***B*** and to the end of ***A***.

Algorithm 2

Step 1: Sort and renumber jobs in non-decreasing order of p_j; break ties by choosing job with lower r_j.

Step 2: Set p_A, p_B and Δ to 0, set ***A*** and ***B*** to $\varnothing$.

Step 3: Consider jobs in increasing index order:

If p_j is unique

assign *j* to ***B*** if $h|\boldsymbol{B}| < w(|\boldsymbol{A}| + 1)$ and to ***A*** otherwise.

If *j* is added to ***B***

let $\Delta_j = r_j + p_j + p_B$;

If $\Delta_j > \Delta$, set $\Delta = \Delta_j$;

$p_B = p_B + p_j$;

Else

let $\Delta_j = r_j - p_A$;

If $\Delta_j > \Delta$, set $\Delta = \Delta_j$;

$p_A = p_A + p_j$;

Else

use the rule above to determine the number of jobs, from all those that share the same p_j, that are to be assigned to ***B*** and to ***A***;

the jobs assigned to ***B*** are those with lower r_j, and are assigned in non-increasing order of r_j;

the jobs assigned to ***A*** are those with higher r_j, and are assigned in non-decreasing order of r_j;

update Δ_j, Δ, p_A and p_B as above.

The complexity of the algorithm is O($n$log n), since Step 1 takes O($n$log n) time, Step 2 takes constant time and Step 3 requires O(n) time.

Theorem 9 Algorithm 2 generates the minimum non-restrictive value for d.

Proof. The algorithm clearly generates a sequence that is optimal for the problem with equal r_j, and any $d < \Delta$ leads to an infeasible schedule, since at least one job will not be available at its optimal start time. However, when several jobs have identical p_j, several optimum sequences exist for the problem with equal r_j (those jobs will have to go into certain positions, but several assignments are possible). When this happens, the algorithm assigns the jobs with lower r_j to the earlier slots. Therefore, we need to show that any other assignment does not lead to a lower Δ. Take any pair of jobs i and j such that $r_i < r_j$ but j is scheduled before i. Let Δ_k^1 be the value of Δ_k when j is scheduled before i and Δ_k^2 be the value of Δ_k after those jobs are swapped, so that i then precedes j. Assume that both j and i are in set $\boldsymbol{B}$. We have $\Delta_i^1 = r_i + p_i + p_B$ and $\Delta_j^1 = r_j + p_j + p_C + p_i + p_B$, where C is the (possibly empty) set of jobs scheduled between j and i. Δ_j^1 is the larger of the two values. After swapping i and j we have $\Delta_j^2 = r_j + p_j + p_B$ and $\Delta_i^2 = r_i + p_i + p_C + p_j + p_B$. Since $\Delta_i^2 < \Delta_j^1$ and $\Delta_j^2 < \Delta_j^1$, the value of Δ cannot be higher after the swap. When both j and i are in set $\boldsymbol{A}$, a similar conclusion can be reached. Finally, assume that j is in set $\boldsymbol{B}$ and i is in set $\boldsymbol{A}$. When Δ is being calculated, we have $\Delta_j^1 = r_j + p_j + p_B$ and $\Delta_i^1 = r_i - p_A$, and $\Delta_j^1 > \Delta_i^1$. After swapping those jobs we would have $\Delta_i^2 = r_i + p_i + p_B$ and $\Delta_j^2 = r_j - p_A$. Since $\Delta_i^2 < \Delta_j^1$ and $\Delta_j^2 < \Delta_j^1$, the value of Δ once again cannot be higher after the swap. Therefore, when several jobs have the same p_j, assigning the jobs with lower r_j to the earlier slots leads to a value of Δ that cannot be higher than the value generated by any other assignment.■

Table 2 – Algorithm 2 example

Index	1	2	3	4	5
p_j	6	8	8	9	11
r_j	1	7	9	8	6

In Table 2 we present an example for Algorithm 2; assume $h = 2$ and $w = 1$. Jobs have already been renumbered in non-decreasing order of p_j, with ties broken by lower r_j. In step 3 the jobs are considered in increasing index order. The first job's p_j is unique and job 1 is assigned to $\boldsymbol{B}$, since $h|\boldsymbol{B}| < w(|\boldsymbol{A}| + 1)$ (2 * 0 < 1 * (0 + 1)). The algorithm then

calculates $\Delta_1 = 1 + 6 + 0 = 7$. Since $\Delta_1 > \Delta = 0$, the algorithm sets $\Delta = \Delta_1 = 7$ and then updates $p_B = 6$. Jobs 2 and 3 have identical processing times. We must then determine the number of jobs assigned to $\boldsymbol{A}$ and $\boldsymbol{B}$. Since we have $h|\boldsymbol{B}| > w(|\boldsymbol{A}| + 1)$ $(2 * 1 > 1 * (0 + 1))$ for the first job to be assigned and $h|\boldsymbol{B}| = w(|\boldsymbol{A}| + 1)$ $(2 * 1 = 1 * (1 + 1))$ for the second job, both jobs are assigned to $\boldsymbol{A}$ in non-decreasing order of r_j. The algorithm then calculates $\Delta_2 = 7 - 0 = 7$. Since $\Delta_2 = \Delta$, Δ is not changed, while p_A is set to 8. When job 3 is assigned to $\boldsymbol{A}$, we have $\Delta_3 = 9 - 8 = 1$. The value of Δ once again does not change, and p_A is updated to $8 + 8 = 16$. The processing time of job 4 is unique and this job is assigned to $\boldsymbol{B}$, since $h|\boldsymbol{B}| < w(|\boldsymbol{A}| + 1)$ $(2 * 1 < 1 * (2 + 1))$. The algorithm calculates $\Delta_4 = 8 + 9 + 6 = 23$. Since $\Delta_4 > \Delta$, the algorithm sets $\Delta = \Delta_4 = 23$ and then updates $p_B = 6 + 9 = 15$. Finally, job 5 is assigned to set $\boldsymbol{A}$ and $\Delta_5 = 6 - 16 = -10 < \Delta$. Therefore, Δ is not changed and the minimum non-restrictive due date is equal to 23.

4. Conclusion

In this paper we considered a single machine weighted earliness/tardiness scheduling problem with different release dates and a common due date. We presented a $O(n \log n)$ algorithm for optimally solving the special case where all jobs have unit processing times, therefore establishing its polynomial solvability. We also developed an algorithm that determines, in $O(n \log n)$ time, the minimum non-restrictive value of the common due date. We remarked that the due date is non-restrictive when the optimal schedule for the non-restrictive version of the problem with equal release dates is feasible. This implies that the non-restrictive version of our problem can be solved in polynomial time using the algorithm proposed for the problem with identical release dates. The restrictive version, however, is NP-hard, since even the restrictive case of the problem with identical release dates is NP-hard. The restrictive version offers ample opportunities for future research, since no exact or heuristic algorithms have yet been proposed. The algorithm given for the problem with unit processing times provides some insights that might be useful in developing heuristic algorithms for the general case. Even though it cannot be directly applied, without changes, to the general problem, since jobs have different processing times and may require more than one time unit to process, some of its ideas can be incorporated in a heuristic algorithm for the general case. For instance, jobs whose earliest possible completion time is greater than or equal to the common due date might be scheduled to complete at their earliest completion times, with empty time ranges still being pushed into a heap of available time periods. A schedule could also be constructed by scheduling all jobs to complete at their earliest completion times. Jobs that finish their processing before the common due date, as well as possibly the whole schedule, could then be moved forward in time to try to reduce the total cost.

Acknowledgements

The authors thank the anonymous referees for several helpful comments that were used to improve this paper.

References

(1) Bagchi, U.; Chang, Y. & Sullivan, R. (1987). Minimizing absolute and squared deviations of completion times with different earliness and tardiness penalties and common due date. *Naval Research Logistics Quarterly*, **34**, 739-751.

(2) Bagchi, U.; Sullivan, R. & Chang, Y. (1986). Minimizing mean absolute deviation of completion times about a common due date. *Naval Research Logistics Quarterly*, **33**, 227-240.

(3) Baker, K. & Chadowitz, A. (1989). Algorithms for minimizing earliness and tardiness penalties with a common due date. Working Paper 240, Amos Tuck School of Business Administration, Dartmouth College, Hanover, N.H.

(4) Baker, K. & Scudder, G.D. (1990). Sequencing with earliness and tardiness penalties: A review. *Operations Research*, **38**, 22-36.

(5) Bank, J. & Werner, F. (2001). Heuristic algorithms for unrelated parallel machine scheduling with a common due date, release dates, and linear earliness and tardiness penalties. *Mathematical and Computer Modelling*, **33**, 363-383.

(6) Cheng, T.C.E.; Chen, Z.-L. & Shakhlevich, N.V. (2002). Common due date assignment and scheduling with ready times. *Computers & Operations Research*, **29**, 1957-1967.

(7) Hall, N.; Kubiak, W. & Sethi, S. (1991). Earliness-tardiness scheduling problems, II: Deviation of completion times about a restrictive common due date. *Operations Research*, **39**, 847-856.

(8) Hoogeveen, J.; Oosterhout, H. & Van de Velde, S.L. (1994). New lower and upper bounds for scheduling around a small common due date. *Operations Research*, **42**, 102-110.

(9) Kanet, J. (1981). Minimizing the average deviation of job completion times about a common due date. *Naval Research Logistics Quarterly*, **28**, 643-651.

(10) Lawler, E.L.; Lenstra, J.K.; Rinnooy Kan, A.H.G. & Shmoys, D.B. (1993). Sequencing and scheduling: Algorithms and complexity. **In:** *Logistics of Production and Inventory, Handbooks in Operations Research and Management Science* [edited by S.C. Graves, A.H.G. Rinnooy Kan and P.H. Zipkin], North-Holland, 445-522.

(11) Mondal, S.A. & Sen, A.K. (2001). Single machine weighted earliness-tardiness penalty problem with a common due date. *Computers & Operations Research*, **28**, 649-669.

(12) Nandkeolyar, U.; Ahmed, M.U. & Sundararaghavan, P.S. (1993). Dynamic single-machine weighted absolute deviation problem: Predictive heuristics and evaluation. *International Journal of Production Research*, **31**, 1453-1466.

(13) Sridharan, V. & Zhou, Z. (1996). A decision theory based scheduling procedure for single machine weighted earliness and tardiness problem. *European Journal of Operational Research*, **94**, 292-301.

(14) Sundararaghavan, P. & Ahmed, M. (1984). Minimizing the sum of absolute lateness in single machine and multimachine scheduling. *Naval Research Logistics Quarterly*, **31**, 325-333.

(15) Szwarc, W. (1989). Single machine scheduling to minimize absolute deviation of completion times from a common due date. *Naval Research Logistics*, **36**, 663-673.

(16) Ventura, J.A. & Weng, M.X. (1995). An improved dynamic programming algorithm for the single machine mean absolute deviation problem with a restrictive common due date. *Operations Research Letters*, **17**, 149-152.