A column generation algorithm for common due date scheduling
van den Akker, J.M.; Hoogeveen, J.A.; Velde, van de, S.L.

Published: 01/01/1997

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
A COLUMN GENERATION ALGORITHM
FOR COMMON DUE DATE SCHEDULING

Marjan van den Akker
National Aerospace Laboratory NLR
P.O. Box 90502, 1006 BM Amsterdam, The Netherlands
email address: vdakker@nlr.nl

Han Hoogeveen
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
email address: slam@win.tue.nl

Steef van de Velde
Faculty of Mechanical Engineering
University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
email address: s.l.vandevalde@wb.utwente.nl

January 1997
ABSTRACT

We consider the single-machine problem of scheduling \( n \) jobs to minimize the total weighted deviation from a given common due date, where the weights for early and tardy completion are asymmetric. First, we assume that the common due date is large. We formulate this problem as an integer linear program with an exponential number of variables and present a column generation algorithm to solve efficiently the linear programming relaxation. Our comprehensive computational study shows that this lower bounding approach performs exceptionally well on randomly generated instances: the solution to the linear program was integral for all randomly generated instances. In this fashion, we were able to solve instances with up to 60 jobs.

Our computational results suggest that the integrality of the optimal solution of the linear programming relaxation is a structural property. We show by example that the integrality gap can be positive, however.

To start up the column generation algorithm, we need a heuristic to generate reasonably good solutions. A simple multi-start iterative improvement algorithm turned out to have a compelling empirical performance: it found an optimal solution for each of our randomly generated instances.

We conclude with showing how the column generation approach can be adapted to deal with a small common due date.

1980 Mathematics Subject Classification (Revision 1991): 90B35.

Keywords and Phrases: asymmetric earliness-tardiness scheduling, common due date, linear programming, column generation, iterative improvement heuristic.
1 Introduction

We consider the following problem. A set $J = \{J_1, \ldots, J_n\}$ of $n$ independent jobs has to be scheduled on a single machine that is continuously available from time zero onwards. The machine can handle at most one job at a time. Job $J_j$ ($j = 1, \ldots, n$) requires a positive integral uninterrupted processing time $p_j$ and should ideally be completed exactly on its due date $d$, which is common to all jobs. We say that this common due date is large, if $d \geq \sum_{j=1}^{n} p_j$; otherwise, we call it small. In case of a large common due date, the constraint that the machine is not available before time zero is redundant. A schedule specifies for each job $J_j$ a completion time $C_j$ such that the jobs do not overlap in their execution. The order in which the machine processes the jobs is called the job sequence. For a given schedule, the earliness of $J_j$ is defined as $E_j = \max\{0, d - C_j\}$ and its tardiness as $T_j = \max\{0, C_j - d\}$. Accordingly, $J_j$ is called early, just-in-time, or tardy if $C_j < d$, $C_j = d$, or $C_j > d$, respectively.

The cost of a schedule $\sigma$ is the sum of weighted job earliness and tardiness, that is,

$$f(\sigma) = \sum_{j=1}^{n} [\alpha_j E_j + \beta_j T_j],$$

where $\alpha_j$ and $\beta_j$ are given positive weights. The problem is to find a schedule with minimum cost.

For the time being, we assume that the common due date is large; we drop this assumption later. We can then take advantage of three well-known properties that characterize a class of optimal solutions (Quaddus, 1987; Baker and Scudder, 1989):

---

- There is no idle time between the execution of the jobs;

- One of the jobs completes exactly on time $d$;

- The jobs completed at or before $d$ are in order of nondecreasing $\alpha_j/p_j$ ratio, and the jobs started at or after $d$ are in nonincreasing $\beta_j/p_j$ ratio.

---

Schedules that possess these three properties are called V-shaped. Note that any V-shaped schedule consists of two parts: the early schedule, consisting of the jobs completed before
or on time $d$; and the tardy schedule, consisting of the jobs completed after time $d$. We call these two schedules complimentary. The characterization implies that the problem is a partitioning problem: we need to select the jobs that are completed at or before $d$, and the jobs that are completed after $d$. Furthermore, due to these three properties, the value of the common due date is irrelevant.

The problem is arguably the most vexing earliness-tardiness scheduling problem remaining; for an overview of such problems, we refer to Baker and Scudder (1990). First of all, the problem is NP-hard in the ordinary sense, but it defies the type of pseudopolynomial algorithm that is so common in earliness-tardiness scheduling — it is therefore still an open question whether the problem is solvable in pseudopolynomial time or NP-hard in the strong sense. The NP-hardness of the problem follows from the NP-hardness of its symmetric counterpart where $\alpha_j = \beta_j = w_j$ for each job $J_j$ ($j = 1, \ldots, n$) (Hall and Posner, 1991). Hall and Posner also present an $O(n \sum_{j=1}^{n} p_j)$ time and space dynamic programming algorithm, thereby establishing the computational complexity of the symmetric problem. This algorithm proceeds by adding the jobs in order of nondecreasing $w_j/p_j$ ratio; hence, the dynamic programming algorithm can be applied to the asymmetric case only if the ratios $\alpha_j/p_j$ and $\beta_j/p_j$ induce the same job sequence. Second, the problem has proved to be very hard in practice as well, since it is quite difficult to compute strong lower bounds (Hall and Sriskandarajah, 1990; De, Ghosh, and Wells, 1994). De et al. formulate the problem as a quadratic 0-1 integer programming problem, which they solve by the branch-and-bound algorithm proposed by Pardalos and Rodgers (1990). Their algorithm solves randomly generated instances with up to 30 jobs without much effort, but it may take more than 300 seconds to solve an instance with $n = 40$ on a VAX 4000-300 machine. De et al. present also a specific randomized local search algorithm, a so-called GRASP algorithm, which has empirically only a small erroneous behavior.

Recently, the combination of linear programming and column generation has been shown to be successful for solving parallel-machine scheduling problems that are actually partitioning problems, such as minimizing total weighted completion time; see Van den Akker,

In this paper, we present a column generation algorithm for computing an exceptionally strong linear programming lower bound for the asymmetric earliness-tardiness problem. We proceed from a set-covering-like formulation of the problem with an exponential number of variables; this formulation is given in Section 2. Then, in Section 3, we present a column generation algorithm to solve the linear programming relaxation. Our computational experience in Section 4 shows a remarkable phenomenon: the solution to the linear programming relaxation was always integral, for all of our randomly generated instances. In this way, we managed to solve instances with up to 60 jobs provably to optimality, without any branch-and-bound algorithm, by ‘just’ solving the linear programming relaxation of the problem. Furthermore, we show that the multi-start iterative improvement algorithm we use as a heuristic to generate the initial columns has a compelling empirical performance as well: it gave an optimal solution for each of our randomly generated instances. Note that the column generation algorithm may be necessary even though the initial columns constitute an optimal solution, if optimality of the solution cannot be established immediately because of degeneracy.

Our computational results suggested that the solution to the linear programming relaxation can be guaranteed to be integral. This is not true, however. In Section 5 we present a counterexample to this conjecture put forward by our computational results. In Section 6, we show that the column generation approach is easily adapted to deal with the case of a small common due date. We present some conclusions in Section 7.

2 Mathematical formulation

We formulate the large common due date problem as a set covering problem with an exponential number of binary variables, $n$ covering constraints, and two additional side-constraints. Our formulation reflects that we search for an optimal V-shaped schedule.

Let $S$ be the set of all early and tardy schedules, including the empty tardy schedule.
Each feasible schedule $s \in S$ is characterized by a 0-1 vector $a_s = (a_{1s}, \ldots, a_{n+2,s})$, where $a_{js} = 1$ if $J_j$ is included in $s$ ($j = 1, \ldots, n$), $a_{n+1,s} = 1$ only if $s$ is an early schedule, and $a_{n+2,s} = 1$ only if $s$ is a tardy schedule. Given $a_s$, we can recover the corresponding schedule $s$ in a straightforward fashion.

Let now $c_s$ be the cost of schedule $s$, and let $x_s$ be a 0-1 variable that takes the value 1 if schedule $s$ is selected and the value 0, otherwise. The problem is then to minimize

$$
\sum_{s \in S} c_s x_s
$$

subject to

$$
\sum_{s \in S} a_{js} x_s \geq 1, \text{ for } j = 1, \ldots, n, \quad (1)
$$

$$
\sum_{s \in S} a_{n+1,s} x_s \leq 1, \quad (2)
$$

$$
\sum_{s \in S} a_{n+2,s} x_s \leq 1, \quad (3)
$$

$$
x_s \in \{0, 1\}, \text{ for each } s \in S. \quad (4)
$$

Conditions (1) enforce that each job is executed at least once. Condition (2) makes sure that no more than one early schedule is selected, while condition (3) makes sure that no more than one tardy schedule is selected. Of course, conditions (1)-(3) will hold with equality in any optimal solution, since all weights are positive.

We cannot hope that this problem is solvable in time polynomial in $n$, since the underlying problem is NP-hard. Furthermore, an explicit formulation of even a modest problem instance is impossible because of the huge number of schedules involved. We are therefore interested in solving the linear programming relaxation, which must proceed by column generation due to the exponential number of variables involved.

### 3 Column generation

The linear programming relaxation of the integer linear programming problem is obtained by replacing conditions (4) by the conditions

$$
0 \leq x_s \leq 1, \text{ for each } s \in S, \quad (5)
$$
which in turn can be replaced by the nonnegativity constraints

\[ x_s \geq 0, \text{ for each } s \in S, \]  

as conditions (2) and (3) prohibit values greater than 1.

In each iteration, we take only a subset of the schedules, say \( S \), into consideration, solve the linear programming relaxation, and add new schedules if needed. From the theory of linear programming, we know that adding a schedule \( s \) with corresponding variable \( x_s \) can decrease the value of the linear programming solution only if \( s \) has negative reduced cost. The reduced cost \( c_s' \) of \( s \) with vector \( a_s \) is defined as

\[ c_s' = c_s - \sum_{j=1}^{n+2} a_{js} \lambda_j, \]

where \( \lambda_j (j = 1, \ldots, n) \) is the value of the dual variable corresponding to the \( j \)th of the constraints (1), and \( \lambda_{n+1} \) and \( \lambda_{n+2} \) are the values of the dual variables corresponding to the conditions (2) and (3); these values follow from the linear programming solution.

We want to solve the pricing problem of finding a schedule \( s \) with minimal \( c_s' \) value; if this minimum is nonnegative, then we know that the value of the linear programming solution will not decrease by taking the remaining schedules into consideration, which implies that we have found the optimal solution of the linear programming relaxation. We solve the pricing problem by finding the early and tardy schedule with minimum reduced cost among all early and tardy schedules, respectively. To that end, we use two pricing algorithms: one to find an early schedule with minimum reduced cost; and one to find a tardy schedule with minimum reduced cost. The latter is essentially the same as the pricing algorithm that we used for the problem of minimizing total weighted completion time on a set of identical parallel machines; see Van den Akker et al. (1995). The pricing algorithm to find an early schedule with minimum reduced cost is very similar; we work out the details of this algorithm below.

In case of an early schedule, we know that \( a_{n+1,s} = 1 \) and \( a_{n+2,s} = 0 \); the pricing problem then reduces to minimizing \( c_s - \sum_{j=1}^{n} a_{js} \lambda_j \) over all binary vectors \( a_s \). Suppose that the jobs
have been reindexed in order of nonincreasing $\alpha_j/p_j$ ratios, that is,

$$\frac{\alpha_1}{p_1} \geq \cdots \geq \frac{\alpha_n}{p_n}.$$

Then for any job $J_j$ in the early schedule $s$ with vector $a_s$, we have that

$$E_j = d - C_j = \sum_{i=1}^{j-1} a_{is}p_i,$$

which implies that

$$c'_s = \sum_{j=1}^{n} \alpha_j a_{js} \sum_{i=1}^{j-1} a_{is}p_i - \sum_{j=1}^{n} a_{js} \lambda_j - \lambda_{n+1} = \sum_{j=1}^{n} \left[ \alpha_j \sum_{i=1}^{j-1} a_{is}p_i - \lambda_j \right] a_{js} - \lambda_{n+1}.$$  

Hence, if $\sum_{i=1}^{j-1} a_{is}p_i = t$, then including $J_j$ in the early schedule affects $c'_s$ by $\alpha_j t - \lambda_j$.

We present a pseudo-polynomial dynamic programming algorithm to solve the pricing problem. Let $F_j(t)$ denote the minimum reduced cost for all early schedules that consist of jobs from the set $\{J_1, \ldots, J_j\}$ in which the first job in the schedule starts at time $d - t$. The initialization is

$$F_j(t) = \begin{cases} -\lambda_{n+1}, & \text{if } j = 0 \text{ and } t = 0, \\ \infty, & \text{otherwise}. \end{cases}$$

The recursion is then, for $j = 1, \ldots, n, t = 0, \ldots, \sum_{i=1}^{j} p_i$

$$F_j(t) = \min\{F_{j-1}(t), F_{j-1}(t - p_j) + \alpha_j(t - p_j) - \lambda_j\},$$

where the first and second term reflect the decision of leaving $J_j$ out of $s$ and adding $J_j$ to $s$, respectively. The early schedule with minimum reduced cost is the one corresponding to

$$\min_{\alpha \leq t \leq P} F_n(t),$$

where $P = \sum_{i=1}^{n} p_i$. Note that any value $F_n(t) < 0$ induces an early schedule with negative reduced cost. This raises the issue whether just the early schedule with minimum negative reduced cost, a small number of early schedules with most negative reduced cost, or all early schedules with negative reduced cost should be added to the set $\tilde{S}$. This implementation issue is discussed in Section 4.1.
We construct the pricing algorithm for generating tardy schedules in a similar fashion. Let 
\( G_j(t) \) denote the minimum reduced cost for all tardy schedules that consist of jobs from 
the set \( \{J_1, \ldots, J_j\} \) in which the last job completes at time \( t \). As our initialization, we have 
\[
G_j(t) = \begin{cases} 
-\lambda_{n+2}, & \text{if } j = 0 \text{ and } t = 0, \\
\infty, & \text{otherwise.}
\end{cases}
\]

The values \( G_j(t) \) (\( j = 1, \ldots, n; t = 0, \ldots, \sum_{i=1}^j p_i \)), are computed through the recurrence 
relation 
\[
G_j(t) = \min\{G_{j-1}(t), G_{j-1}(t - p_j) + \beta_j t - \lambda_j\},
\]
and we determine 
\[
\min_{0 \leq t \leq P} G_n(t),
\]
to find the tardy schedule with minimum reduced cost from among all tardy schedules. Again, each value \( G_n(t) < 0 \) induces a tardy schedule with negative reduced cost. The issue 
of which tardy schedules with negative reduced cost to add to \( S \) is addressed in Section 4.1.

Note that both pricing algorithms run in \( O(n \sum_{j=1}^n p_j) \) time and space.

4 Computational results

In this section, we report on our computational experience with our column generation 
algorithm for randomly generated instances. We first discuss the implementation issues and 
then our computational results.

4.1 Implementation issues

The algorithms were coded in the computer language C, and the experiments were conducted 
on an HP9000/710 Unix machine. We used the package CPLEX to solve the linear programs.

The implementation issues involved are:

— The design of a heuristic to generate the initial set \( \tilde{S} \) of early and tardy schedules;
— The size of the initial set \( \tilde{S} \), that is, the number of columns to be generated by the 
  heuristic;
— The columns to add to the linear program per iteration.

The first implementation issue is the design of a heuristic for generating initial columns to compute the initial dual variables with which we start the column generation method. We use a simple iterative improvement procedure for this purpose, which works as follows. First, we generate a feasible solution by deciding randomly whether a job is scheduled early or tardy. Then, we compute the corresponding V-shaped schedule and we search the neighborhood of the current schedule for a better V-shaped schedule. The neighborhood of a V-shaped schedule consists of all V-shaped schedules that can be obtained by three types of changing operations: moving a tardy job to the early schedule; moving an early job to the tardy schedule; and swapping an early and a tardy job. As soon as we find a better schedule in the current neighborhood, we adopt it as the new schedule. This process is repeated and terminates when no further improvement can be found.

The second issue is the number of initial solutions to be generated by the heuristic. Note that multiple initial solutions can be obtained by repetitive use of the heuristic described above. As expected, we found that the speed of convergence of the column generation algorithm is approximately a parabolic function of this number. In case of a small number, the initial dual variables may be a long shot away from the optimal dual variables; in case of a large number, the size of the linear programs may outweigh the benefit of having better initial dual variables. Our computational experiments indicated that running the heuristic 50 times, each time with a different starting solution, resulting in at most 100 columns, was a fairly robust choice. We have not tried to finetune this number to size or any other characteristic of an instance.

The third issue is how many and which schedules to add to the set \( \tilde{S} \) in each iteration. The dilemma we are facing is that the more schedules we add per iteration, the fewer linear programs we (probably) need to solve — which is good; but the more schedules we add per iteration, the bigger the linear programs become — which is bad. In the previous section, we noted that each value \( F_n(t) < 0 \) and \( G_n(t) < 0 \) corresponds to a column with negative reduced cost. Hence, using the pricing algorithm, we can determine as many early and tardy
schedules with negative reduced cost as there are values \( t \) for which \( F_n(t) < 0 \) or \( G_n(t) < 0 \), at the expense of a little extra effort. In our computation results, however, this turned out to be not worthwhile. Accordingly, per iteration we add no more than one tardy and one early schedule. There is one exception to this rule, however: If a schedule with minimum reduced cost together with its complementary schedule constitute a better primal solution than the incumbent upper bound, then both schedules are added to \( \tilde{S} \).

### 4.2 Column generation algorithm

In this subsection, we give a sketchy description of our implementation of the column generation algorithm — we found this implementation the most robust.

**Column generation algorithm**

**Step 1.** Use the iterative improvement heuristic with 50 different starting solutions to generate the initial set \( \tilde{S} \) of early and tardy schedules.

**Step 2.** Solve the linear programming relaxation.

**Step 3.** Run both pricing algorithms to determine the early schedule with minimum negative reduced cost and the tardy schedule with minimum negative reduced cost.

**Step 4.** If neither an early, nor a tardy schedule with negative reduced cost exists, then go to **Step 6**. If such an early schedule exists, then determine its complementary tardy schedule. If together they constitute a better feasible solution than we have right now, then add both the early schedule and its complementary tardy schedule to the set \( \tilde{S} \). If they do not form a better schedule, then just add the early schedule. The same procedure applies to the tardy schedule with minimum negative reduced costs, if it exists.

**Step 5.** Return to **Step 2**.

**Step 6.** Stop. We have solved the linear programming relaxation to optimality.

### 4.3 Performance of the column generation algorithm

We tested our algorithm on two classes of randomly generated instances:
(i) instances with processing times and weights drawn from the uniform distribution \([1, 100]\). This concurs with the procedure used by De et al. (1991) to generate instances.

(ii) instances with processing times and weights drawn from the uniform distribution \([1, 10]\).

We tested our algorithm on instances with \(n = 10, 20, 30, 40, 50, 60\) jobs, and for each combination of \(n\) and instance class we generated 100 instances.

For each value of \(n\) and for either instance class, we report on the number of times (out of 100) that the optimal linear programming solution was integral, the average computation time, the maximum computation time, the average number of columns generated, the maximum number of columns generated, the average number of linear programming problems solved, and the maximum number of linear programming problems solved.

Tables 1 and 2 summarize our computational results for the column generation algorithm. The headers of the columns are:

<table>
<thead>
<tr>
<th>(n)</th>
<th>(OPT)</th>
<th>(ACT)</th>
<th>(MCT)</th>
<th>(ACOL)</th>
<th>(MCOL)</th>
<th>(ALP)</th>
<th>(MLP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>0.28</td>
<td>0.49</td>
<td>55</td>
<td>74</td>
<td>22</td>
<td>38</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>3.59</td>
<td>4.35</td>
<td>191</td>
<td>248</td>
<td>86</td>
<td>122</td>
</tr>
<tr>
<td>30</td>
<td>100</td>
<td>14.21</td>
<td>26.21</td>
<td>397</td>
<td>483</td>
<td>227</td>
<td>316</td>
</tr>
<tr>
<td>40</td>
<td>100</td>
<td>80.28</td>
<td>112.74</td>
<td>730</td>
<td>875</td>
<td>500</td>
<td>642</td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>283.11</td>
<td>595.84</td>
<td>1278</td>
<td>1592</td>
<td>991</td>
<td>1364</td>
</tr>
<tr>
<td>60</td>
<td>100</td>
<td>1734.56</td>
<td>2346.18</td>
<td>4143</td>
<td>4722</td>
<td>1705</td>
<td>2013</td>
</tr>
</tbody>
</table>

Table 1: Results for the class (i) instances.
Table 1 shows the results for the class (i) instances, and Table 2 shows the results for the class (ii) instances.

An astonishing but very convenient phenomenon was that the linear programming solution turned out to be integral for each instance: \( OPT = 100 \) for each \( n \) and either instance class. What is more, the solution of each intermediate linear programming problem was always integral as well — and we have solved millions of these. This raises the question whether integrality of the optimal solution of the linear programming relaxation is a structural property. It is not — as we will show in Section 5.

Since the pricing algorithm requires pseudo-polynomial time, we can expect beforehand that the performance of our algorithm deteriorates with the size of the processing times of the jobs. Indeed, the class (ii) instances, with smaller processing times, are easier to solve than the class (i) instances.

As a whole, our computational results show that using our algorithm we can solve larger problems to optimality than before: we solve instances with up to 60 jobs, while De et al. (1994) went no further than 40 jobs. Solving larger instances was impossible due to memory problems: the size of the linear program became too big.

Having the optimal solution value of each instance we generated, we are able to report on the empirical performance of the iterative improvement heuristic we proposed in Section 4.1. Table 3 summarizes our findings. We report the maximum optimality gap \( (MOG) \) between the heuristic and the optimal solution value, expressed as a percentage of the optimal solution value. We also give the average computation time \( (ACT) \) — the variance of the

<table>
<thead>
<tr>
<th>( n )</th>
<th>( OPT )</th>
<th>( ACT )</th>
<th>( MCT )</th>
<th>( ACOL )</th>
<th>( MCOL )</th>
<th>( ALP )</th>
<th>( MLP )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>0.24</td>
<td>0.30</td>
<td>44</td>
<td>53</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>1.72</td>
<td>2.40</td>
<td>146</td>
<td>179</td>
<td>92</td>
<td>124</td>
</tr>
<tr>
<td>30</td>
<td>100</td>
<td>10.06</td>
<td>12.41</td>
<td>317</td>
<td>397</td>
<td>223</td>
<td>397</td>
</tr>
<tr>
<td>40</td>
<td>100</td>
<td>46.88</td>
<td>64.09</td>
<td>541</td>
<td>725</td>
<td>403</td>
<td>572</td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>61.01</td>
<td>493.91</td>
<td>870</td>
<td>1778</td>
<td>669</td>
<td>1591</td>
</tr>
<tr>
<td>60</td>
<td>100</td>
<td>387.63</td>
<td>1174.48</td>
<td>1344</td>
<td>2161</td>
<td>904</td>
<td>1707</td>
</tr>
</tbody>
</table>

Table 2: Results for the class (ii) instances.
computation time is very little. Again, we encountered an unexpected phenomenon: the iterative improvement heuristic with multiple starts gave an optimal solution for each randomly generated instance. Being slower, it outperforms De et al.’s randomized local search algorithm in terms of quality.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$MOG$</th>
<th>$ACT$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000</td>
<td>0.04</td>
</tr>
<tr>
<td>20</td>
<td>0.000</td>
<td>0.30</td>
</tr>
<tr>
<td>30</td>
<td>0.000</td>
<td>1.05</td>
</tr>
<tr>
<td>40</td>
<td>0.000</td>
<td>2.62</td>
</tr>
<tr>
<td>50</td>
<td>0.000</td>
<td>5.15</td>
</tr>
<tr>
<td>60</td>
<td>0.000</td>
<td>9.45</td>
</tr>
</tbody>
</table>

Table 3: Performance of the heuristic for class (i) and (ii) instances.

5 An instance with integrality gap

The computational results presented in the previous section show that column generation is a very suitable approach for solving the asymmetric earliness-tardiness problem with a large common due date. Our computational results made us almost believe that there was no integrality gap. We managed to construct a five-job instance for which the column generation approach failed, however. The data are shown in Table 4. There are several optimum schedules, for instance the combination of the early schedule $J_1, J_3, J_4$ and the tardy schedule $J_2, J_5$; they all have cost $808$. The optimum solution to the LP-problem is to take half of the early schedules $J_1, J_2, J_4$ and $J_1, J_3, J_5$ and half of the tardy schedules $J_3, J_4$ and $J_2, J_5$; its value is $807.5$.

<table>
<thead>
<tr>
<th>$J_j$</th>
<th>$p_j$</th>
<th>$\alpha_j$</th>
<th>$\beta_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_1$</td>
<td>1000</td>
<td>22</td>
<td>1000</td>
</tr>
<tr>
<td>$J_2$</td>
<td>2</td>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>$J_3$</td>
<td>3</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>$J_4$</td>
<td>18</td>
<td>1000</td>
<td>12</td>
</tr>
<tr>
<td>$J_5$</td>
<td>20</td>
<td>1000</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 4: Instance with positive integrality gap.
Note that if we multiply all penalty weights $\alpha_j$ and $\beta_j$ with a factor $M > 0$, then the integrality gap becomes $\frac{1}{2}M$. Hence, the integrality gap is unbounded.

6 Adaptation to a small common due date

If $d < P$, where $P = \sum_{j=1}^{n} p_j$, then the nonavailability of the machine before time zero may cause infeasibility of the optimal solution for the large common due date problem. As shown in an earlier paper (Hoogeveen and Van de Velde, 1991), the optimum schedule then takes one of the following two forms:

1. V-shaped with one job completing at time $d$;

2. Almost V-shaped with all jobs processed in the interval $[0, P]$,

where almost V-shaped means that the schedule V-shaped with exception of the job that is started before and completed after time $d$; this job is usually referred to as the pivot. We find the optimum schedule by determining the best schedule in either class.

Optimizing over the first class of schedules requires a straightforward adaptation of the column generation approach only. We have to take care that the total processing time of the jobs in the early schedule amounts to no more than $d$, which is easily incorporated in the dynamic programming algorithm.

Optimizing over the second class requires more effort. Suppose that we know the identity of the pivot; say it is job $J_j$. We do not know the start and completion time of $J_j$, however. Hence, we do not know the completion and start time of the last early and the first tardy job, respectively. We do know, however, that the first early and the last tardy job start at time 0 and complete at time $P$, respectively. Therefore, we construct an early schedule from the outside, that is, we build the early schedule from left to right starting at time 0. Similarly, we build the tardy schedule from time $P$ backwards. Note that the total processing time $t$ of the jobs in the early and tardy schedule should satisfy $d-p_j < t < d$ and $P-d-p_j < t < P-d$, respectively; we can use smaller than signs instead of smaller than or equal signs, because each schedule with a job completed at time $d$ falls into the first class. If we introduce a linear
programming problem for each choice of the pivot, then we solve the linear programming relaxation by taking the minimum of all \((n + 1)\) outcome values.

It is possible to combine these \(n\) linear programming problems, however. The first issue we have to decide on is whether we include the pivot in the early or tardy schedule. This decision affects the running time of the algorithm only; the approach is the same for both possibilities. Suppose that we add the pivot to the early schedule; hence, we can use the standard dynamic programming algorithm to generate tardy schedules, where the total processing time \(t\) of the jobs in the tardy schedule should satisfy \(P - d - p_{\text{max}} < t < P - d\), where \(p_{\text{max}}\) denotes the maximum processing time. As to generating early schedules, we let the dynamic programming algorithm run \(n\) times, once for every choice of the pivot. We then compare the solutions obtained in these \(n\) runs and add the best one to the linear program, if its reduced cost is negative. The times needed for finding the early and tardy schedule with minimum reduced cost are \(O(n^2d)\) and \(O(n(P - d))\), respectively. If we decide to include the pivot in the tardy schedule, then we find the early and tardy schedule with minimum reduced cost in a similar fashion; this takes \(O(nd)\) and \(O(n^2(P - d))\) time, respectively. Hence, the running time is smaller if we include the pivot in the early schedule, if \(d < P/2\), and in the tardy schedule, otherwise.

It is not clear beforehand if it pays off to combine the \(n\) linear programming problems. We may have to add many columns, which may lead to memory problems. On the other hand, we do not have to go through all \(n\) subproblems, many of which are irrelevant.

7 Conclusions

We have presented an effective column generation method for solving the problem of scheduling jobs around a large common due date with asymmetric weights, and we indicated how to adapt this method to deal with a small common due date. Using this method, we were able to solve instances with up to 60 jobs to optimality by solving the linear programming relaxation of a set covering formulation of the problem — branch-and-bound was never required for our randomly generated instances. We have shown that the integrality gap can be
positive, unbounded even, however. Furthermore, we have shown that a simple multi-start iterative improvement local search algorithm performs very well.

Hoogeveen, Oosterhout, and Van de Velde (1994) showed why randomly generated instances of the symmetric earliness-tardiness problem with unit penalty weights can be expected to be computationally easy for large instances. For minimizing total weighted completion time on identical parallel machines, Chan et al. (1995) proved that the linear programming solution value of a set covering formulation of the problem is, under mild conditions, asymptotically optimal. But it is still an open question why randomly generated instances of the earliness-tardiness problem under study are easy in practice in the sense that the linear programming solution always seems to give an integral solution.

References


