Ann Oper Res (2008) 159: 107–123
DOI 10.1007/s10479-007-0270-5

Scheduling orders on either dedicated or flexible machines in parallel to minimize total weighted completion time

Joseph Y.-T. Leung · Haibing Li · Michael Pinedo

Published online: 1 December 2007

Abstract We are interested in the problem of scheduling orders for different product types in a facility with a number of machines in parallel. Each order asks for certain amounts of various different product types which can be produced concurrently. Each product type can be produced on a subset of the machines. Two extreme cases of machine environments are of interest. In the first case, each product type can be produced on one and only one machine which is dedicated to that product type. In the second case, all machines are identical and flexible; each product type can be produced by any one of the machines. Moreover, when a machine in this case switches over from one product type to another, no setup is required. Each order has a release date and a weight. Preemptions are not allowed. The objective is minimizing the total weighted completion time of the orders. Even when all orders are available at time 0, both types of machine environments have been shown to be NP-hard for any fixed number (≥ 2) of machines. This paper focuses on the design and analysis of approximation algorithms for these two machine environments. We also present empirical comparisons of the various algorithms. The conclusions from the empirical analyses provide insights into the trade-offs with regard to solution quality, speed, and memory space.

This research is supported by the National Science Foundation through grants DMI-0300156 and DMI-0245603.

Electronic supplementary material The online version of this article (http://dx.doi.org/10.1007/s10479-007-0270-5) contains supplementary material, which is available to authorized users.

J.Y.-T. Leung (✉)
Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA
e-mail: leung@cis.njit.edu

H. Li
Lehman Brothers Inc., 1301 The Avenue of Americas, New York, NY 10019, USA
e-mail: hl27@njit.edu

M. Pinedo
Stern School of Business, New York University, 40 West Fourth Street, New York, NY 10012, USA
e-mail: mpinedo@stern.nyu.edu

Keywords Order scheduling · Total weighted completion time · Approximation algorithms · NP-hard

1 Introduction

The problems under consideration in this paper are typically referred to as *order scheduling* (Leung et al. 2005b). In such a problem, we consider a facility with m machines in parallel, which can produce k different product types. In particular, each product type $l = 1, 2, \ldots, k$ can be produced on a subset of the m machines, namely $M_l \subseteq \{1, 2, \ldots, m\}$. Two extreme cases of machine environments are of interest. In the first case, the so-called *fully dedicated* case, each product type can be produced by one and only one machine which is dedicated to that particular product type. In the second case, referred to as the *fully flexible* case, all machines are identical and flexible so that each product type can be produced by any one of the machines. In this case, when a machine switches over from one product type to another, no setup is required.

Assume there are n orders from n different clients. Each order $j = 1, 2, \ldots, n$ requests a quantity of product type $l = 1, 2, \ldots, k$ which requires $p_{lj} \geq 0$ units of processing. If $p_{lj} = 0$, it implies that order j does not request any amount of product type l. Each order j may also have a *release date* r_j which denotes the time when the order arrives, and a weight w_j which ranks the importance of order j. The *completion time* of order j, denoted by C_j, is the time when the last product type for this order has been finished on one of the machines. Let C_{lj} denote the individual completion time of product type l for order j on one machine, it is obvious that $C_j = \max_l\{C_{lj}\}$.

In this paper, we focus on the objective of minimizing the total weighted completion time of the orders, i.e., $\sum w_j C_j$. For due-date related objectives concerning the fully dedicated case, the reader is referred to Ng et al. (2003) and Leung et al. (2006a). For a more general description of order scheduling models and their many application examples, the reader is referred to Li (2005).

Even when all orders are available at time 0, the two environments have been shown to be NP-hard for any fixed number ($\geq$2) of machines (Sung and Yoon 1998; Blocher and Chhajed 1996). We focus on the design and analysis of approximation algorithms for these two machine environments. Because of space constraints, we have to omit the proofs for our results. The interested reader may want to go through the references.

This paper is organized as follows. In the next section, we consider the dedicated case. The flexible case is considered in Sect. 3. In the last section we present our conclusions.

2 The dedicated machine environment

As mentioned before, in the dedicated machine environment, each machine can produce one and only one product type. Without loss of generality, we assume that machine i is the only machine that can produce type i and type i is the only type that can be produced on machine i (we can always relabel the machines and product types to achieve this), so that the subscript i refers to a machine as well as to a product type. We refer to p_{ij} as the time required to produce the product type i requirement for order j on machine i. For convenience, we denote the problem of minimizing $\sum w_j C_j$ as $PD \mid r_j \mid \sum w_j C_j$. The problem is strongly NP-hard even if all $r_j = 0$ (Sung and Yoon 1998). An overview of the past work on some special cases of this problem can be found in Leung et al. (2005b).

Due to the strong NP-hardness of the problem, it is of interest to develop good heuristics. The algorithms we considered are of two types: priority rules (either static or dynamic) and

LP-based algorithms. The priority rules are only applicable to $PD \mid\mid \sum w_j C_j$, i.e., when $r_j = 0$ for all j.

2.1 Priority rules for $PD \mid\mid \sum w_j C_j$

Let Ω denote the set of orders that have not yet been scheduled. Assuming a partial schedule π, we consider five greedy ways of selecting the next order $j^* \in \Omega$ to be added to the partial schedule. The first two methods described below are basically static priority rules, i.e., the entire sequence can be determined at time $t = 0$ based only on information pertaining to the orders. The third method is a two-pass rule, which schedules the orders in two passes (the schedule information obtained in the first pass provides the data necessary for doing the second pass). The fourth and fifth method are single pass dynamic priority rules. That is, the schedule can be developed in a single pass; however, it cannot be done using only information pertaining to the orders. In order to add an additional order to a partial schedule, information pertaining to the existing partial schedule has to be taken into account as well.

- The Weighted Shortest Total Processing Time first (*WSTP*) rule schedules the orders in increasing order of $\sum_{i=1}^{m} p_{ij}/w_j$. Ties are broken arbitrarily.
- The Weighted Shortest Maximum Processing Time first (*WSMP*) rule schedules the orders in increasing order of $\max_i\{p_{ij}\}/w_j$. Ties are broken arbitrarily.
- The Weighted Smallest Maximum Completion Time first (*WSMC*) rule first sequences the orders on each machine, $i = 1, 2, \ldots, m$, in increasing order of p_{ij}/w_j. (Note that the sequences of the orders on the various machines may be different.) The rule then computes the completion time for order j as $C_j^{'} = \max_{i=1}^{m}\{C_{ij}\}$. In a second pass, the rule schedules the orders in increasing order of $C_j^{'}$. Ties are broken arbitrarily.
- The rule that applies Weighted Shortest Processing Time first to the machine with the largest current load (*WSPL*); it functions as a dynamic priority rule that generates a sequence of orders one at a time, each time selecting as the next order the order $j^* \in \Omega$ such that

$$j^* = \arg\min_{j\in\Omega}\left\{\frac{p_{i^*j}}{w_j}\right\},$$

  where i^* is the machine with the largest workload under the partial schedule π. Ties are broken arbitrarily.
- The Weighted Earliest Completion Time first (*WECT*) rule selects as the next order j^* which satisfies

$$j^* = \arg\min_{j\in\Omega}\left\{\frac{C_j - C_k}{w_j}\right\},$$

  where C_k is the finish time of the order that was scheduled immediately before order j^*. Ties may be broken arbitrarily.

For each heuristic described above, after a sequence of orders, say S, has been generated, a postprocessing procedure can be applied. Let $[j]$ be the order scheduled in position j of S. The postprocessing procedure works as follows: For each position $j = 2, 3, \ldots, n$ in S, interchange order $[j]$ (for convenience, we denote this order as j^*) with order $[j-1]$ if $C_{j^*} \le C_{[j-1]}$. Such an interchange generates a solution that is at least as good as the original sequence. Note that the case $C_{j^*} \le C_{[j-1]}$ occurs only when $p_{i^*j^*} = 0$, where i^* refers to the machine that determines the completion time of order $[j-1]$ (i.e., machine i^* has, among all machines, the latest finishing time for order $[j-1]$). If, after the swap of orders, $C_{j^*} \le C_{[j-2]}$, then we follow up with an additional interchange of order j^* with order

$[j-2]$. We continue with this postprocessing until C_{j^*} is larger than the completion time of the order that immediately precedes it. Note that after each swap, the finish time of j^* either decreases or remains unchanged, while the finish time of each order that is swapped with j^* remains unchanged. This is due to the fact that order j^* has zero processing time on the machine on which the swapped order has its largest finish time. Thus, the postprocessing, if any, produces a solution that is no worse than the one prior to the postprocessing.

Note that in each heuristic, there may at times be ties. Since ties may be broken arbitrarily, each heuristic could lead to various different schedules with different values of objective functions.

Through a sorting algorithm, both *WSTP* and *WSMP* can be implemented to run in $O(mn + n \lg n)$ time, and *WSMC* can be implemented to run in $O(mn \lg n)$ time. Both *WSPL* and *WECT* can be implemented in a rather straightforward manner to run in $O(mn^2)$ time.

When $m = 2$, Sung and Yoon (1998) showed that both *WSTP* and *WSMC* have an approximation ratio of 2. Actually, Wang and Cheng (2003) showed that *WSTP*, *WSMP* and *WSMC* are all m-approximation algorithms when applied to $PDm \mid\mid \sum w_j C_j$. As for *WSPL*, Leung et al. (2005a) showed that the algorithm is unbounded even when all $w_j = 1$. However, an empirical analysis showed that it performs well in practice for $w_j = 1$. The *WECT* algorithm is a generalization of the m-approximation *ECT* algorithm introduced in Leung et al. (2005a). We can show the following result whose proof can be found in Leung et al. (2006b).

Theorem 2.1 *For $PD \mid\mid \sum w_j C_j$, the worst-case bound of WECT is m.*

It would not be surprising that the priority rules may perform better when the processing times of each order are subject to constraints that ensure some form of regularity in the processing times. Sung and Yoon (1998) showed that for $m = 2$ the performance ratio of *WSTP* can be reduced to 3/2 when the processing times satisfy $(p_{1j} + p_{2j})/2 \geq |p_{1j} - p_{2j}|$ for each $j = 1, 2, \ldots, n$. In the following two theorems we obtain tighter bounds when the priority rules are applied to $PD \mid\mid \sum w_j C_j$ with the processing times subject to additional constraints; their proofs can be found in Leung et al. (2006b).

Theorem 2.2 *If*

$$\sum_{i=1}^{m} p_{ij}/m \geq \max_{1 \leq i \leq m} \{p_{ij}\} - \min_{1 \leq i \leq m} \{p_{ij}\}$$

for each order $j = 1, 2, \ldots, n$, then the worst-case bound of WSTP is $2 - \frac{1}{m}$, while both WSMP and WECT have a worst-case bound of

$$\frac{1}{1 + \mathcal{H}(m) - \mathcal{H}(2m-1)},$$

where $\mathcal{H}(k) \equiv 1 + \frac{1}{2} + \cdots + \frac{1}{k}$ is the harmonic series.

Note that

$$\lim_{m \to \infty} \frac{1}{1 + \mathcal{H}(m) - \mathcal{H}(2m-1)} = \frac{1}{1 - \ln 2} \approx 3.259.$$

Theorem 2.3 *If $\max_{1 \leq i \leq m}\{p_{ij}\} \leq 3 \min_{1 \leq i \leq m}\{p_{ij}\}$ for each order $j = 1, 2, \ldots, n$, then WSTP, WSMP and WECT all have a worst-case bound of $3 - \frac{6}{m+2}$.*

2.2 LP-based algorithms for $PD \mid r_j \mid \sum w_j C_j$

In this subsection we allow orders to have different release dates. Preemptions are not allowed. However, unforced idleness of the machines is allowed. We present two approximation algorithms based on different LP relaxations.

2.2.1 An algorithm based on a completion time formulation

Hall et al. (1997) presented a 3-approximation LP-based algorithm for $1 \mid r_j \mid \sum w_j C_j$. We shall extend this algorithm in order to solve $PD \mid r_j \mid \sum w_j C_j$.

Let $\mathcal{O} = \{1, 2, \ldots, n\}$ denote the set of all orders. For any subset $\mathcal{S} \subseteq \mathcal{O}$, let $p_i(\mathcal{S}) = \sum_{j \in \mathcal{S}} p_{ij}$, and $p_i^2(\mathcal{S}) = \sum_{j \in \mathcal{S}} p_{ij}^2$, $i = 1, 2, \ldots, m$. The $PD \mid r_j \mid \sum w_j C_j$ problem can be relaxed to the following linear program:

$$LP_1 = \text{minimize} \sum_{j=1}^{n} w_j C_j$$

subject to

$$C_j \geq r_j + p_{ij}, \quad i = 1, \ldots, m, \ j = 1, \ldots, n; \tag{1}$$

$$C_j \geq C_{ij}, \quad i = 1, \ldots, m, \ j = 1, \ldots, n; \tag{2}$$

$$\sum_{j \in \mathcal{S}} p_{ij} C_{ij} \geq \frac{p_i^2(\mathcal{S}) + (p_i(\mathcal{S}))^2}{2}, \quad i = 1, \ldots, m, \text{ for each } \mathcal{S} \subseteq \mathcal{O}. \tag{3}$$

Constraint sets (1) and (2) are trivial. However, constraint set (3) needs some justification. Assume that $\mathcal{S} = \{1, 2, \ldots, |\mathcal{S}|\}$. It follows that for $j \in \mathcal{S}$,

$$C_{ij} \geq \sum_{k \leq j} p_{ik}, \quad i = 1, \ldots, m.$$

The inequality is due to the fact that there may be some idle time in the schedule because of the release dates. Thus,

$$p_{ij} C_{ij} \geq p_{ij} \sum_{k \leq j} p_{ik}.$$

Summing $p_{ij} C_{ij}$ over all $j \in \mathcal{S}$ and simple algebra results in (3).

It is clear that (3) generates an exponential number of constraints. For the one-machine case, Queyranne (1993) has shown that such constraints can be separated polynomially so that the above linear program can be solved in polynomial time by a variant of ellipsoid method (Grötschel et al. 1993). This is the key observation that the above linear program can be used as a relaxation for approximation algorithms. Now consider the following algorithm:

An LP-based Algorithm Using Completion Times (LP_1)

Step 1: Solve LP_1 by the ellipsoid method; let the optimal solution be $\overline{C}_1, \overline{C}_2, \ldots, \overline{C}_n$.
Step 2: Schedule the orders in nondecreasing order of $\overline{C}_j$. Break ties arbitrarily. Insert an idle time when r_j is greater than the completion time of the $(j-1)$th order.

We can show the following results for the above LP-based algorithm; its proof can be found in Leung et al. (2006b):

Theorem 2.4 *The worst-case bounds of LP_1 applied to $PD \mid\mid \sum w_j C_j$ and $PD \mid r_j \mid \sum w_j C_j$ are 2 and 3, respectively.*

2.2.2 An algorithm based on a time interval formulation

Inspired by the time interval indexed linear programming formulation for $P \mid r_j \mid \sum w_j C_j$ due to Hall et al. (1997), Wang and Cheng (2003) presented a 16/3-approximation algorithm for $PD \mid\mid \sum w_j C_j$. In what follows, we present an extension of Wang and Cheng's algorithm for $PD \mid r_j \mid \sum w_j C_j$.

Given $\lambda > 1$, we divide the time horizon of potential completion times into the intervals: $[1, 1], (1, \lambda], (\lambda, \lambda^2], \ldots, (\lambda^{L-1}, \lambda^L]$, where L is the smallest integer so that $\lambda^L \geq \max_{1\leq j\leq n}\{r_j\} + \max_{1\leq i\leq m}\{\sum_{j=1}^{n} p_{ij}\}$. For convenience, let $t_0 = 1$, and $t_l = \lambda^{l-1}, l = 1, \ldots, L$. Thus, the lth interval runs from t_{l-1} to t_l, $l = 1, 2, \ldots, L$. Let the decision variable x_{jl} be:

$$x_{jl} = \begin{cases} 1, & \text{if order } j \text{ is scheduled to finish within the interval } (t_{l-1}, t_l]; \\ 0, & \text{otherwise.} \end{cases}$$

Consider the following linear programming relaxation:

$$LP_2 = \text{minimize} \sum_{j=1}^{n} w_j \sum_{l=1}^{L} t_{l-1} x_{jl}$$

subject to

$$\sum_{l=1}^{L} x_{jl} = 1, \quad j = 1, \ldots, n; \tag{4}$$

$$\sum_{k=1}^{l}\sum_{j=1}^{n} p_{ij} x_{jk} = \sum_{j=1}^{n} p_{ij} \sum_{k=1}^{l} x_{jk} \leq t_l, \quad i = 1, \ldots, m;\ l = 1, \ldots, L; \tag{5}$$

$$x_{jl} = 0, \quad \text{if } t_l < r_j + p_{ij},\ j = 1, \ldots, n,\ l = 1, \ldots, L; \tag{6}$$

$$x_{jl} \geq 0, \quad j = 1, \ldots, n;\ l = 1, \ldots, L. \tag{7}$$

The following algorithm is based on the above LP relaxation:

An LP-based Algorithm Using Time Intervals (LP_2)

Step 1: Given λ, solve LP_2 and let the optimal solution be $\bar{x}_{jl}$, $j = 1, \ldots, n, l = 1, \ldots, L$.
Step 2: Let $\overline{C}_j = \sum_{l=1}^{L} t_{l-1}\bar{x}_{jl}$, $j = 1, \ldots, n$.
Step 3: Schedule the jobs in nondecreasing order of $\overline{C}_j$. Ties are broken arbitrarily. Insert idle time when r_j is greater than the completion time of the $(j-1)$th order.

Wang and Cheng (2003) showed that, given $\lambda = 2$, LP_2 is a 16/3-approximation algorithm for $PD \mid\mid \sum w_j C_j$. When the orders have different release dates, we can show the following result; its proof can be found in Leung et al. (2006b):

Theorem 2.5 *Given $\lambda = 2$, LP_2 is a $\frac{19}{3}$-approximation algorithm for $PD \mid r_j \mid \sum w_j C_j$.*

For LP_2, it would be of interest to investigate if a smaller λ leads to a better performance. In what follows, we present an empirical analysis.

2.3 Empirical analysis of the algorithms

Since the priority rules apply only to $PD \mid\mid \sum w_j C_j$, we focus our experiments on the problem with all release dates equal to zero.

For each problem size with $n = 20, 50, 100, 200$ orders and $m = 2, 5, 10, 20$ machines, 100 instances are randomly generated using a factor called *order diversity*. The order diversity k is used to characterize the number of different product types each order requires. The following three cases of order diversity are considered:

$k = 2$: In problem instances 1 to 20 each order requests 2 different product types. We denote this group as G_1.

$k = m$: In problem instances 21 to 80 each order requests the maximum number of different product types, namely m, i.e. the number of machines. However, these 18 instances are grouped into the following 3 subgroups:

For instances 21 to 40 in group G_2, the processing times of each order have no additional constraints.

For instances 41 to 60 in group G_3, the processing times of each order j are subject to the constraint

$$\max_{1\le i\le m}\{p_{ij}\} \le 3 \min_{1\le i\le m}\{p_{ij}\}.$$

For instances 61 to 80 in group G_4, the processing times of each order j are subject to the constraints

$$\sum_{i=1}^{m} p_{ij}/m \ge \max_i\{p_{ij}\} - \min_i\{p_{ij}\}.$$

$k = r$: In problem instances 81 to 100 in group G_5 each order requests a random number (r) of different product types; r is randomly generated from the uniform distribution $[1, m]$.

When the number of product types, l, for each order j is determined, l machines are chosen randomly. For each machine i that is selected, an integer processing time p_{ij} is generated from the uniform distribution [1, 100]. Note that for instances 41 to 80, the processing times of each order are generated in such a way that they satisfy the additional requirements. In addition to the generation of processing times, for each order j, a weight is randomly generated from the uniform distribution [1, 10]. In total, $4 \times 4 \times 100 = 1600$ instances are generated.

The algorithms are implemented in C++. We used the GLPK 4.4 (Makhorin 2004) callable library to solve the linear programs in the LP_2 algorithm. The running environment is based on the Windows 2000 operating system; the PC used was a desktop computer (CPU 3.0 GHz plus 512 MB RAM). It should be noted that the time-interval LP-based algorithm needs a significant amount of virtual memory. For example, for a problem instance with $n = 200$ and $m = 20$, when we let $\lambda = 2^{1/4}$, the total memory usage for the time-interval LP-based algorithm could reach 1 GB. Because of this we set the total file paging size for the hard disk equal to 1272 MB.

We study the performance of the algorithms in terms of two aspects: the number of times they turn out to be the best and their average costs. We also compare the average

running times of the algorithms. In what follows, we use the following notation to refer to the different versions of LP_2.

$$LP_2'\text{: } LP_2 \text{ with } \lambda = 2^{1/4} \qquad LP_2''\text{: } LP_2 \text{ with } \lambda = 2^{1/2} \qquad LP_2'''\text{: } LP_2 \text{ with } \lambda = 2$$

Due to space constraints, we omit the tables with the experimental results; these tables are available at http://web.njit.edu/~leung/aor/tablesPD.pdf, see also electronic supplementary material.

We first compare the performance of the algorithms in terms of the number of times at which they generate the best solution. From the results obtained, we can draw the following conclusions:

- In group G_1 with $k = 2$, LP_1, LP_2', and *WECT* have the best performance. A close study of the results reveals that for instances with small n, LP_1 is better than both LP_2' and *WECT*, and the performance of LP_2' is close to that of *WECT*. In some occasional cases, LP_2'' also produces best solutions. When n is large, LP_2' is significantly better than all other algorithms. The results also reveal a counter-intuitive finding that LP_1 does not perform better than the other algorithms all the time, even though it has the best known worst-case performance bound (note that the performance ratios of LP_2' and LP_2'' are unknown yet).
- For the instances in group G_2, for which $k = m$ but with no additional constraints on the processing times, LP_2', LP_1, and *WECT* exhibit the best performance. For small n, LP_1 and LP_2' perform closely, and are slightly better than *WECT*. However, for large n, LP_2' performs better than LP_1, which in turn performs better than *WECT*. Again, LP_2'' also produces best solutions for some occasional cases.
- *WECT* is the best algorithm in groups G_3 and G_4. LP_2' is second best, and LP_1 third best. *WSTP*, *WSMP*, *WSMC*, and *WSPL* may occasionally beat the other algorithms, especially when n is small. However, when n becomes large, the tendency is that *WECT* beats all other algorithms. These results are consistent with our theoretical analysis that the priority rules exhibit a better performance with additional constraints on the processing times, except that the results for LP_1 are somehow counter-intuitive.
- For the instances in group G_5, for which $k = r$, the best algorithms are LP_2' and LP_1. For small n, LP_1 performs closely to LP_2'. In addition, for occasional cases of small n and m, *WECT* may beat the other algorithms. However, for large n, LP_2' becomes significantly better than all other algorithms.

We now compare the performance of the algorithms in terms of the average costs. We also compare the average running times of the various algorithms. For convenience, we use the notation $\prec$ to indicate that algorithm A outperforms algorithm B if $A \prec B$. Furthermore, if algorithm A almost ties with algorithm B, we denote it as $A \sim B$.

- For each n, the percentage of each priority rule (except for *WECT*) tends to increase when m increases. This is consistent with our previous theoretical analysis which indicates that the performance of each priority rule becomes worse when m becomes larger. In contrast, neither LP_1 nor LP_2 exhibit such a relationship between their performance and the value of m. Secondly, the results also show that a smaller λ leads to a better performance of LP_2.
- For the problem instances in group G_1, when $n \geq 100$, it turns out that LP_2' is significantly better than all other algorithms, and we find that

$$LP_2' \prec LP_1 \prec LP_2'' \prec LP_2''' \sim WECT \sim WSMC \prec WSTP \prec WSMP \prec WSPL.$$

When n is small, we find that

$$LP_1 \prec LP_2' \prec LP_2'' \sim WECT \prec LP_2''' \sim WSMC \prec WSTP \prec WSMP \prec WSPL.$$

– In group G_2, *WECT*, LP_1, and LP_2' can beat one another when $n = 20$. However, when $n \geq 50$, LP_2' is much better than all other algorithms, and we find that

$$LP_2' \prec LP_1 \prec WECT \prec LP_2'' \prec LP_2''' \prec WSTP \prec WSMC \prec WSMP \prec WSPL.$$

Note that the difference between *WECT* and LP_2' is less than 2%. Thus, the performance of *WECT* is actually very close to that of LP_2'. However, the results show that LP_2' requires hours of running time for large instances, while *WECT* requires only milliseconds. As stated before, LP_2' requires virtual memory up to 1.2 GB. By contrast, *WECT* only requires several kilobytes of memory.

– In group G_3, the two best algorithms are *WECT* and LP_2'. *WECT* tends to perform better than the others when n is small, while LP_2' performs well when n is large. The results show that

$$LP_2' \prec LP_2'' \prec LP_1 \prec WECT \prec LP_2''' \prec WSTP \prec WSMC \prec WSMP \sim WSPL.$$

Note that the difference between *WECT* and LP_2' is less than 0.5%. Thus, the performance of *WECT* is almost the same as that of LP_2'. However, to achieve such performance, LP_2' requires more computational resources than *WECT*. It is also interesting to see that, *WSTP*, *WSMC*, and *WSMP* perform better than LP_2'''.

– In group G_4, *WECT* is the best. In details, the algorithms are ranked as follows:

$$WECT \prec LP_2' \prec LP_1 \prec LP_2'' \prec WSTP \prec WSMC \sim WSMP \prec LP_2''' \sim WSPL.$$

Again, *WSTP* , *WSMC*, and *WSMP* perform better than LP_2'''.

– For group G_5, LP_2' is the best. The algorithms are ranked as:

$$LP_2' \prec LP_2'' \prec LP_1 \prec WECT \prec LP_2''' \prec WSTP \prec WSMC \prec WSMP \prec WSPL.$$

Again, the results produced by *WECT* are quite close to those of the LP-based algorithms.

To compare the performance versus the running times of the various versions of LP_2, we compare for each group of instances the percentages that the average costs of LP_2'' and LP_2''' are larger than that of LP_2'. From the results, we can see that the gap between the average cost of LP_2'' and that of LP_2' is actually very small. For most cases, it is less than 1.0%; and the largest one is 5.9%. Therefore, the performance of LP_2'' is actually very close to that of LP_2'. However, from the results, the average running time of LP_2' is much more than that of LP_2'' (in some cases more than 10 times). In addition, in the experiments, we noticed that the memory requirement of LP_2' is twice that of LP_2''. Thus, in practice, if the use of LP_2 is considered, it is recommended to choose $\lambda = \sqrt{2}$ in order to strike a balance between performance and the use of computational resources.

Summarizing the empirical analysis, we recommend the use of *WECT* and LP_2 with $\lambda = \sqrt{2}$. It should be noted that, even though LP_1 performs better than LP_2 with $\lambda = \sqrt{2}$, and it does not take too much memory space, it runs very slowly. This is the main reason why we do not recommend the use of LP_1 in practice. In an environment which requires a solution to be generated quickly with limited memory spaces, *WECT* is the most preferable. It is simple to implement, requires a small amount of memory, runs fast, and produces good results.

3 The flexible machine environment

In the flexible case with all identical machines, each product type can be produced by any one of the machines. We refer to p_{lj} as the time required to process product type l of order j on any machine $i = 1, 2, \ldots, m$. When a machine switches over from one product type to another, we assume that no setup is required. For convenience, if k is fixed, the problem under consideration is denoted as $PF \mid \Pi k \mid \sum w_j C_j$ when the number of machines m is arbitrary, and it is denoted as $PFm \mid \Pi k \mid \sum w_j C_j$ when m is fixed. If k is arbitrary, then the problem is denoted as either $PF \mid \Pi \mid \sum w_j C_j$ or $PFm \mid \Pi \mid \sum w_j C_j$, depending on whether the number of machines is arbitrary or fixed.

Since $PF \mid \Pi \mid \sum C_j$ is strongly NP-hard (Blocher and Chhajed 1996), it follows that the more general version $PF \mid \Pi \mid \sum w_j C_j$ is NP-hard as well. We are interested in designing heuristics for $PF \mid \Pi \mid \sum w_j C_j$ that consist of two phases. The first phase determines the sequence of orders, while the second phase assigns the individual jobs within each order to the specific machines. Based on this idea, we consider two classes of heuristics: *sequential two-phase heuristics* and *dynamic two-phase heuristics*.

3.1 Sequential two-phase heuristics

The first phase of the *sequential two-phase heuristics* sequences the orders; the second phase assigns the individual jobs of each order to the specific machines. Rules for sequencing the orders include:

– The *Weighted Shortest Total Processing time first* (*WSTP*) rule which sequences the orders in increasing order of

$$\frac{\sum_{l=1}^{n_j} p_{lj}}{w_j}.$$

– The *Weighted Shortest LPT Makespan first* (*WSLM*) rule which sequences the orders in increasing order of

$$\frac{C_{LPT}^{(j)}}{w_j}, \quad j = 1, 2, \ldots, n,$$

where $C_{LPT}^{(j)}$ is the makespan of the schedule obtained by scheduling the jobs of order j on all m parallel machines according to the *longest processing time first* (*LPT*) rule, assuming each machine is available from time zero on.

– The *Weighted Shortest MULTIFIT Makespan first* (*WSMM*) rule which sequences the orders in increasing order of

$$\frac{C_{MF}^{(j)}}{w_j}, \quad j = 1, 2, \ldots, n,$$

where $C_{MF}^{(j)}$ is the makespan of the schedule obtained by scheduling the jobs of order j on all m parallel machines according to the *MF* assignment rule which is described below, assuming each machine is available from time zero on.

After the sequence of orders has been determined by one of the above rules, the individual jobs within each order are assigned to the specific machines according to one of the assignment rules listed below:

- The *List Scheduling* rule (*LS*) assigns in each iteration an unassigned (arbitrary) job to a machine (or one of the machines) with the smallest workload, until all jobs are assigned.
- The *Longest Processing Time first* rule (*LPT*) assigns in each iteration an unassigned job with the longest processing time to a machine (or one of the machines) with the smallest workload, until all jobs are assigned.
- The *Bin Packing* rule (*BIN*) first determines a target completion time for an order using the *LPT* assignment rule (just as a trial, not as a real assignment). This completion time is used as a target completion time (bin size). At each iteration, the *BIN* rule assigns an unassigned job with the longest processing time to a machine with the largest workload. If the workload of the machine exceeds the target completion time after the assignment, then undo this assignment and try the assignment on the machine with the second largest workload. This try-and-check procedure is repeated until the job can be assigned to a machine without exceeding the target completion time. If assigning the job to the machine with the smallest workload still exceeds the target completion time, then assign it to this machine, and reset the target completion time as the completion time of the job on this machine. The whole procedure is repeated until all jobs are assigned to the machines.
- The *MULTIFIT* rule (*MF*) assigns the jobs of an order to the machines following an idea that is similar to (but not exactly the same as) the *MULTIFIT* algorithm for $Pm \mid\mid C_{\max}$ (Coffman et al. 1978). The original *MULTIFIT* algorithm uses the *First Fit Decreasing* (*FFD*) rule for the bin packing problem. In contrast, we use the *Best Fit Decreasing* (*BFD*) rule. Let j be the order whose jobs are to be assigned. In the *BFD* procedure, we treat the machines as bins that are partially filled, and treat the jobs of order j as items whose sizes are exactly equal to their processing times. The jobs of order j are pre-sorted in nonincreasing order of their processing times. Given the partial schedule generated for the orders scheduled before order j, and given a target completion time t (bin size), the pre-sorted jobs of order j are assigned sequentially, each going into the bin (machine) with the largest workload into which it still fits. If all the jobs can be assigned to the machines without exceeding t, then *BFD* is considered "successful".

  If we specify for t a lower bound $LB_t(j)$, and an upper bound $UB_t(j)$, by trying *BFD* with different values of t in between $LB_t(j)$ and $UB_t(j)$, schedules of different length can be generated. If the processing times are integers, a binary search procedure would make the algorithm run faster. Using a binary search procedure, we can initially try *BFD* with $t = (UB_t(j) + LB_t(j))/2$; whenever *BFD* succeeds, we let $UB_t(j) = t$; otherwise, let $LB_t(j) = t$. This procedure is repeated until t cannot be updated any more, or until after a specified number of iterations, say $\mathcal{I}$. The schedule obtained by trying *BFD* with the latest $UB_t(j)$ as t is chosen.

  Now we fix an initial setting for $LB_t(j)$ and $UB_t(j)$. Before the jobs of order j are assigned, we denote the smallest workload of the m machines as $C_{\min}$. It is easy to see that, an initial lower bound $LB_t(j)$ can be set as

$$LB_t(j) = C_{\min} + \max\left\{\sum_{l=1}^{n_j} \frac{p_{lj}}{m}, \max_{1\le l\le n_j}\{p_{lj}\}\right\},$$

  since $LB_t(j)$ is no larger than the completion time of an optimal assignment. As for the initial upper bound $UB_t(j)$, we can set it as the completion time of order j obtained by a trial assignment using the *BIN* rule. In case the *BFD* would not be successful even with the initial upper bound as its target completion time, we accept the assignment by the *BIN* rule.

The various ways of combining sequencing rules with assignment rules lead to twelve different heuristics. However, we have focused only on those that appear the most promising:

- Four heuristics based on *WSTP*, namely, *WSTP-LS*, *WSTP-LPT*, *WSTP-BIN*, *WSTP-MF*.
- One heuristic based on *WSLM*, namely, *WSLM-LPT*.
- One heuristic based on *WSMM*, namely, *WSMM-MF*.

The unweighted version of *WSTP-LS* has been studied in Yang and Posner (2005). The *WSTP-LPT*, *WSTP-BIN* and *WSLM-LPT* rules are generalizations of the unweighted versions which are described in Blocher and Chhajed (1996). The *WSTP-MF* and *WSMM-MF* rules are new.

In order to determine the time complexity of these algorithms, consider first the running time of the sequencing rules:

- *WSTP* needs to compute $\sum_{l=1}^{n_j} p_{lj}/w_j$ for all orders, which takes $O(kn)$ time. Then, applying a sort procedure on $\sum_{l=1}^{n_j} p_{lj}/w_j$ takes $O(n \lg n)$ time. Thus, *WSTP* runs in $O(kn + n \lg n)$ time.
- *WSLM* needs to compute the makespan of the jobs of each order according to the *LPT* rule. Since applying *LPT* on the jobs in each order takes $O(k \lg k + k \lg m)$, n orders need $O(kn \lg km)$ time. In addition, after computing the makespans of the *LPT* schedules for all orders, the orders have to be sorted in terms of these makespans. The sorting procedure takes $O(n \lg n)$ time. Thus, the total running time of *WSLM* is $O(kn \lg km + n \lg n)$.
- *WSMM* needs to compute the makespan of the jobs of each order by using the *MF* rule. As we can see later, this takes $O(kn \lg k + \mathcal{I}knm)$ time. After computing the makespans, sorting the orders in terms of their makespans takes $O(n \lg n)$. Thus, in total, *WSMM* takes $O(kn \lg k + \mathcal{I}knm + n \lg n)$ time.

Now we consider the running time of the assignment rules:

- For *LS*, we can use a min-heap data structure to maintain the machines with different workloads, it costs $O(\lg m)$ time to retrieve from the min-heap the machine with the smallest workload, and costs another $O(\lg m)$ time to update the workload of this machine in the heap after a job is assigned. Since *LS* sequentially assigns the jobs of each order in arbitrary order, and there are $O(kn)$ jobs, it follows that *LS* takes $O(kn \lg m)$ time.
- For *LPT*, additional time is required to sort the jobs of each order nonincreasingly in terms of their processing times. This takes $O(nk \lg k)$ time. The subsequent assignment procedure requires the same time as *LS*. Thus, *LPT* runs in $O(nk \lg k + kn \lg m) = O(kn \lg km)$.
- *BIN* uses *LPT* to obtain a trial assignment, which already takes $O(kn \lg km)$ time. Note that in the worst case the assignment of a job needs to be tried on all m machines from the largest-workload one to the smallest-workload one. This worst-case takes $O(km)$ time to assign the jobs of each order. Therefore, n orders need $O(knm)$ time. Thus, *BIN* takes $O(kn \lg km + knm)$ time in total.
- *MF* first uses *BIN* to determine the upper bounds of target completion times before assigning the jobs of the n orders. This takes $O(kn \lg km + knm)$ time. Then, for each order, *MF* uses *BFD* to assign its jobs. Note that in the worst case the assignment of a job according to *BFD* also needs to try all m machines from the one with the largest-workload to the one with the smallest-workload. As in *BIN*, *BFD* takes $O(knm)$ time, the number of runs of *BFD* for each order is $\mathcal{I}$. Thus, *MF* takes $O(kn \lg km + \mathcal{I}knm)$ time in total. Note that $\mathcal{I}$ is usually a small integer. For example, when $\mathcal{I} = 20$, the gap between the upper bound and the lower bound is $2^{20} = 1048576$, which is a very wide range for a binary search procedure already.

The running time of each heuristic is as follows:

Heuristic	Time complexity	Heuristic	Time complexity
WSTP-LS	$O(kn\lg m + n\lg n)$	*WSTP-MF*	$O(kn\lg km + n\lg n + \mathcal{I}knm)$
WSTP-LPT	$O(kn\lg km + n\lg n)$	*WSLM-LPT*	$O(kn\lg km + n\lg n)$
WSTP-BIN	$O(kn\lg km + n\lg n + knm)$	*WSMM-MF*	$O(kn\lg km + n\lg n + \mathcal{I}knm)$

3.2 Dynamic two-phase heuristics

The second class of heuristics are referred to as *dynamic two-phase heuristics*. In these heuristics, the sequence of orders is not fixed prior to the assignment of the various jobs to the machines, i.e., the sequence is determined dynamically. The heuristics use the *LPT* rule, the *BIN* rule or the *MF* rule to assign the jobs to the machines. However, to determine the next order to be sequenced, a greedy approach is applied to make a trial assignment of the jobs of all remaining orders by using one of three rules, and the next selected order j^* satisfies

$$j^* = \arg\min_{j\in\Omega}\left\{\frac{C_j - C_{j'}}{w_j}\right\},$$

where Ω is the set of unscheduled orders, and $C_{j'}$ is the finish time of the order that was scheduled immediately before order j^*. Ties may be broken arbitrarily. If $C_{j^*} < C_{j'}$, then we can shift forward all jobs belonging to j^* assigned to each machine, and put them before all jobs belonging to j' on that machine. Now after this shift operation, if there exists another order j'' such that $C_{j^*} < C_{j''}$, we carry out the same shift operation between j^* and j''. This procedure is repeated until such a case does not occur any more. Clearly, with such a shift operation, the finish time of an order such as order j' remains unchanged, whereas the finish time of j^* decreases. This post-processing procedure helps reduce the objective cost of the schedule. The three heuristics are referred to as *Weighted Earliest Completion Time by LPT* (*WECT-LPT*), *Weighted Earliest Completion Time by BIN* (*WECT-BIN*) and *Weighted Earliest Completion Time by MF* (*WECT-MF*), respectively. The first two heuristics are generalizations of the unweighted versions presented in (Blocher and Chhajed 1996), while *WECT-MF* is new. Natural implementation of each heuristic requires n^2 runs of the respective assignment rule. Thus, the running times of these three algorithms are $O(kn^2\lg km)$, $O(kn^2\lg km + kn^2m)$ and $O(kn^2\lg km + \mathcal{I}kn^2m)$, respectively.

3.3 Worst-case analyses of the heuristics

In what follows, we analyze the performance bounds of the above heuristics. Blocher and Chhajed (1996) made an empirical study for the unweighted version of *WSTP-LPT*, *WSTP-BIN*, *WECT-LPT*, and *WECT-BIN*. Yang and Posner (2005) did a worst-case analysis for the unweighted version of *WSTP-LPT*, and showed that it has a tight bound of 6/5 for the unweighted problem with only two machines.

When $n = 1$, it is clear that the problem becomes $P \,||\, C_{\max}$. The results for $P \,||\, C_{\max}$ imply the following:

Lemma 3.1 *For $PF \mid \Pi \mid \sum w_jC_j$, the worst-case ratio of algorithms that use LS and LPT cannot be less than $2 - \frac{1}{m}$ and $\frac{4}{3} - \frac{1}{3m}$, respectively.*

We can show the following result for the four heuristics based on the *WSTP* sequencing rule; its proof can be found in Leung et al. (2007):

Theorem 3.2 *For* $PF \mid \Pi \mid \sum w_j C_j$, *each of WSTP-LS*, *WSTP-LPT*, *WSTP-BIN*, *and WSTP-MF has a worst-case bound of* $2 - \frac{1}{m}$.

It turns out that the remaining five algorithms, namely, *WSLM-LPT*, *WSMM-MF*, *WECT-LPT*, *WECT-BIN* and *WECT-MF*, can perform very badly. To see this, consider the following example:

- Let $w_j = 1,\ j = 1, 2, \ldots, n$.
- Let $x = \rho \cdot m$, where $0 < \rho < 1$; let $\epsilon > 0$.
- Each order $j = 1, 2, \ldots, x$ requests m product types, each of which requires 1 unit of processing.
- Each order $j = x + 1, x + 2, \ldots, x + m(m - x)$ requests only 1 product type requiring $1 + \epsilon$ units of processing.

It is easy to see that each heuristic

$$H \in \{\textit{WSLM-LPT}, \textit{WSMM-MF}, \textit{WECT-LPT}, \textit{WECT-BIN}, \textit{WECT-MF}\}$$

produces the same schedule, with the objective value being

$$\sum w_j C_j(H) = \sum_{j=1}^{x} j + m \cdot \sum_{j=1}^{m-x} (x + j \cdot (1 + \epsilon)).$$

On the other hand, it also easy to see that an optimal schedule has an objective value of

$$\sum w_j C_j(OPT) = m \cdot \sum_{j=1}^{m-x} j \cdot (1 + \epsilon) + \sum_{j=1}^{x} (j + (m - x)(1 + \epsilon)).$$

It can be determined that

$$\lim_{m \to \infty} \left(\lim_{\epsilon \to 0} \frac{\sum w_j C_j(H)}{\sum w_j C_j(OPT)} \right) = \frac{1 + \rho}{1 - \rho}.$$

Thus, when ρ is close to 1, the above ratio can be arbitrarily large. This implies that the performance ratio of these heuristics is not bounded by any constant. Actually, we can show that the upper bound is m; its proof can be found in Leung et al. (2007).

Theorem 3.3 *For* $PF \mid \Pi \mid \sum w_j C_j$, *each one of the WSLM-LPT*, *WSMM-MF*, *WECT-LP*, *WECT-BIN*, *and WECT-MF algorithms has a worst-case bound of* m.

However, it is not clear whether this bound is tight or not. Note that the above example is not a tight example.

3.4 Empirical analysis of the heuristics

We generate problem instances with different sizes that are determined by n, m and k, where $n \in \{20, 50, 100, 200\}$, $m \in \{2, 5, 10, 20\}$ and $k \in \{2, 5, 10, 20, 50, 100\}$. For each combination of n, m and k, 10 problem instances are randomly generated. These 10 problem instances have a similar structure and are treated as a group. To produce an instance for a

combination of n, m and k, n orders are generated. For each order j, the number of product types k_j is generated from the uniform distribution $[1, k]$. Then, for each product type $l = 1, 2, \ldots, k_j$, an integer processing time p_{lj} is generated from the uniform distribution $[1, 100]$. In addition, a weight for order j is randomly generated from the uniform distribution $[1, 10]$. In total, 960 instances are generated.

The algorithms have been implemented in C++. The running environment is based on the Windows 2000 operating system; the PC used was a notebook computer (Pentium III 900 MHz plus 384 MB RAM). In what follows, we study the performance of the algorithms in terms of two aspects: the frequencies at which they are the best and a comparison of their average costs. We also compare their average running times. Again, due to space constraints, we omit the tables with the experimental results, which are available at http://web.njit.edu/~leung/aor/tablesPF.pdf.

We first compare the heuristics in terms of the frequencies at which they are the best. The results show that *WSTP-MF* is the best. In addition, the tendency is that a larger k leads to a higher percentage of *WSTP-MF* that are the best. The same tendency also applies to the other two heuristics based on the *MF* assignment rule, i.e., *WSMM-MF* and *WECT-MF*, even though they are inferior to *WSTP-MF*. Thus, it would be advantageous to apply *MF*-based heuristics to instances in which each order requests many different product types. When k is small, *WSMM-BIN* is comparable to *WSMM-MF*. This enforces our observation that *MF* does not have much of an advantage when each order has only a small number of jobs. Even though the analysis in the previous section showed that the worst-case performance bounds of the heuristics based on *WSTP* are much better than the other heuristics, we could not find a clear indication from the experimental results. Thus, we need to compare the heuristics in terms of the average costs. From the average cost point of view, we can draw the following conclusions:

- *WSTP-MF* performs the best, *WSTP-BIN* performs second best.
- The four heuristics based on *WSTP* are better than the others. This agrees with the worst-case bounds described earlier.
- When k and n are fixed, the performances of the heuristics worsen as m increases. This is also consistent with the corresponding worst-case bounds obtained previously.
- When n and m are fixed, the performances of the heuristics improve as k increases. In addition, as k increases, the differences in the performances of the various heuristics become smaller.
- When m and k are fixed, the differences between the *WSTP*-based heuristics become smaller as n increases. On the other hand, as k decreases, the differences between those heuristics which are not based on *WSTP* and the *WSTP*-based heuristics become larger.
- The performances of the heuristics are sensitive to k/m. When k/m is small, the gaps between the heuristics that are not based on the *WSTP* rule and the *WSTP*-based heuristics are large. When k/m is large, these gaps are small.

As for the average running times of the heuristics, the results show that the *WSTP*-based heuristics run faster than other heuristics. Based on the observations made above, we would recommend for practical applications either *WSTP-MF* or *WSTP-BIN*.

4 Concluding remarks

In this paper, we presented an overview of the design and analysis of approximation algorithms for two extreme cases of the customer order scheduling problem, with the minimization of the total weighted completion time as objective.

For the dedicated case, the algorithms include several priority rules as well as two LP-based algorithms. Although priority rules are easy to implement, our analysis showed that their performance guarantees vary according to the distributional properties of the processing times. In contrast, various linear programming relaxations uniformly provide tight lower bounds for approximation algorithms. However, different relaxations may result in approximation algorithm with very different performance guarantees. Fortunately, both LP-based algorithms we presented in this paper have a fixed ratio performance guarantee. According to our empirical analysis, we would recommend the use of *WECT* and LP_2 with $\lambda = \sqrt{2}$. However, with regard to solution quality, speed, memory space, and implementation complexity, *WECT* is more preferable.

For the flexible case, the heuristics considered fall into two categories, namely the sequential two-phase heuristics and the dynamic two-phase heuristics. We performed a worst case as well as an empirical analysis of nine heuristics. The analyses reveal that the four *WSTP*-based heuristics perform better than the five other heuristics, in spite of the fact that the four *WSTP*-based heuristics are static whereas three of the other heuristics are dynamic. This may, at first sight, appear to be an anomaly, since observations in classical scheduling problems indicate that dynamic heuristics usually perform better than static heuristics. One reason why the static *WSTP*-based heuristics perform better than the dynamic heuristics may be based on the fact that the dynamic selection criteria may put some orders with many jobs ahead of a large number of orders with a few jobs; the cumulative cost of these orders with few jobs becomes very large, just as in the example we presented in order to show that the static *WSTP*-based heuristics are better than the dynamic heuristics. Our worst-case and empirical analyses also validate this observation. With regard to solution quality and speed, we recommend the *WSTP*-based heuristics.

References

Blocher, J., & Chhajed, D. (1996). The customer order lead-time problem on parallel machines. *Naval Research Logistics*, *43*, 629–654.

Coffman, E. G., Garey, M. R., & Johnson, D. S. (1978). An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, *7*, 1–17.

Grötschel, M., Lovasz, L., & Schrijver, A. (1993). *Geometric algorithms and combinatorial optimization*. Berlin: Springer.

Hall, L. A., Schulz, A. S., Shmoys, D. B., & Wein, J. (1997). Scheduling to minimize average completion time: off-line and on-line approximation algorithms. *Mathematics of Operations Research*, *22*, 513–544.

Leung, J. Y.-T., Li, H., & Pinedo, M. (2005a). Order scheduling in an environment with dedicated resources in parallel. *Journal of Scheduling*, *8*, 355–386.

Leung, J. Y.-T., Li, H., & Pinedo, M. (2005b). Order scheduling models: an overview. In G. Kendall, E. Burke, S. Petrovic & M. Gendreau (Eds.), *Multidisciplinary scheduling: theory and applications* (pp. 37–53). Berlin: Springer.

Leung, J. Y.-T., Li, H., & Pinedo, M. (2006a). Scheduling orders for multiple product types with due date related objectives. *European Journal of Operational Research*, *168*, 370–389.

Leung, J. Y.-T., Li, H., & Pinedo, M. (2006b). Approximation algorithms for minimizing total weighted completion time of orders on identical machines in parallel. *Naval Research Logistics*, *53*, 243–260.

Leung, J. Y.-T., Li, H., & Pinedo, M. (2007). Scheduling orders for multiple product types to minimize total weighted completion time. *Discrete Applied Mathematics*, *155*, 945–970.

Li, H. (2005). *Order scheduling in dedicated and flexible machine environments*. Ph.D. Thesis, Department of Computer Science, New Jersey Institute of Technology, Newark, New Jersey.

Makhorin, A. (2004). *GNU linear programming kit: reference manual (version 4.4)*.

Ng, C., Cheng, T., & Yuan, J. (2003). Concurrent open shop scheduling to minimize the weighted number of tardy jobs. *Journal of Scheduling*, *6*, 405–412.

Queranne, M. (1993). Structure of a simple scheduling polyhedron. *Mathematical Programming*, *58*, 263–285.

Sung, C., & Yoon, S. (1998). Minimizing total weighted completion time at a pre-assembly stage composed of two feeding machines. *International Journal of Production Economics*, *54*, 247–255.

Wang, G., & Cheng, T. (2003). Customer order scheduling to minimize total weighted completion time. In *Proceedings of the 1st multidisciplinary international conference on scheduling: theory and applications* (pp. 409–416). Nottingham, United Kingdom.

Yang, J., & Posner, M. E. (2005). Scheduling parallel machines for the customer order problem. *Journal of Scheduling*, *8*, 49–74.