Open Shop Scheduling to Minimize Makespan

Kevin Jurcik
Department of Mathematical Sciences
Lakehead University
Thunder Bay, Ontario

Abstract
We study various classical Open Shop scheduling problems to minimize the makespan. For problems which are polynomially solveable, we present a polynomial time algorithm to solve the problem. For those problems which are NP-hard, we give a proof of NP-hardness. We also present two modern Open Shop scheduling problems, Controllable Open Shop and Flexible Open Shop. For each of these problems, we present an algorithm to solve the problem.

Contents

1 Introduction

In this paper we study various classical Open Shop scheduling problems to minimize the makespan, as well as two modern Open Shop problems. In an open shop, we are presented with m machines and n jobs. Each job is composed of m tasks. The ith task of the jth job requires p_{ij} units of processing to be completed on the ith machine. When $m = 2$, we let a_j be the processing time of the first task of Job j and let b_j be the processing time of the second task of Job j. No two tasks of the same job may be processed concurrently, nor may any single machine process two jobs simultaneously. The makespan $C_{\max}$ of a schedule is the time at which no more processing is required among all the jobs or, equivalently, the time at which the latest machine finishes processing.

Any particular scheduling problem can be denoted in the form $\alpha|\beta|\gamma$, where α specifies the machine environment, β specifies the processing characteristics and constraints, and γ specifies the optimization criteria. For the classical problems discussed in this paper, the fields will take on very few values. We have that α will be either $O2$, $O3$, or Om corresponding to 2, 3 or an arbitrary number m of machines. We have that β will be any combination of the constraints r_j and $prmp$. The constraint r_j specifies that not every job is available for processing at time 0, but rather Job j will become available at some time r_j. Consequently, no task of Job j can be processed until after time r_j. The constraint $prmp$ specifies that the processing of a task may be preempted at any time to be continued at a later time. If $prmp$ is not present, then each task must be processed continuously until completion once it has begun processing on some machine. Finally, we have that γ will take on only the function $C_{\max}$ as defined above. That is, the objective goal of each of the classical problems studied in this paper is to find a schedule with minimal makespan.

Of great interest during the study of scheduling problems is the complexity of the problem. We say that a problem is polynomially solvable if there exists an algorithm to find an optimal schedule which runs in an amount of time which is polynomial in the size of the input n, the number of jobs. Otherwise, if we are able to prove that finding an optimal schedule for a problem is at least as difficult as finding a solution for a different suitably difficult problem (that is, a problem which is NP-complete), we say that the problem is NP-hard.

For those problems which are polynomially solvable, we present a polynomial time algorithm to solve the problem. For those problems which are NP-hard, we present a proof of NP-hardness.

1.1 Table of Results

Here we present a table where each row contains a problem, a simple summary of the complexity of the problem, and a reference to the section of this paper where the problem is studied in detail.

Problem	Results	Reference
$O2\|\|C_{\max}$	Solvable by $O(n)$-time algorithm	Section 2.1
$Om\|\|C_{\max}$	NP-hard for $m > 2$	Section 2.2
$O2\|r_i\|C_{\max}$	NP-Hard	Section 2.3
$Om\|prmp\|C_{\max}$	Solvable by $O(r^2)$ algorithm, where r is the number of nonzero tasks	Section 2.4
$Om\|prmp, r_i\|C_{\max}$	Solvable by $O(pn)$ algorithm, where p is the number of distinct release times	Section 2.5
$O2\|contr, C_{\max} \leq C\|K$	Solvable by $O(n)$ algorithm	Section 3.1
$O2\|contr\|(C_{\max}, K)$	Solvable by $O(n\log n)$ algorithm	Section 3.2
$FO2\|prmp\|C_{\max}$	Solvable by $O(n)$ algorithm	Section 4

2 Classical Problems

2.1 $O2||C_{\max}$

First, we look at $O2||C_{\max}$, minimizing the makespan in an open shop with two machines without preemption. The problem is polynomially solvable, and in fact the makespan of an optimal scehdule can be stated explicitly in a closed form. We have that the makespan of an optimal schedule is given by

$$C_{\max} = \max\{\sum_j a_j, \sum_j b_j, \max_j\{a_j + b_j\}\}.$$

An algorithm is presented by T. Gonzalez and S. Sahni [4]. In the following algorithm, let $a||b$ denote the concatenation of strings a and b.

Step 1: Let $a_o = b_0 = 0$. Let $r = l = 0$. Let S, S_1, S_2 be empty strings. Let $A = \{j|a_j \geq b_j\}, B = \{j|a_j < b_j\}$.

Step 2: For each j in A, if $a_j \leq b_r$, then append j to the right of S. Otherwise, append r to the right of S and let $r := j$.

Step 3: For each j in B, if $b_j \leq a_l$, then append j to the left of S. Otherwise, append l to the left of S and let $l := j$.

Step 4: Remove all occurrences of the digit 0 in the string S.

Step 5: If $\sum(a_j) - a_l < \sum(b_j) - b_r$, then let $S_1 := S||r||l$ and $S_2 := l||S||r$. Otherwise, let $S_1 := l||S||r$ and $S_2 := r||l||S$.

Step 6: Schedule the jobs on Machine 1 back-to-back in the order dictated by the string S_1, and schedule the jobs on Machine 2 back-to-back in the order dictated by the string S_2.

A useful feature of the schedule generated by this algorithm is that the jobs may be scheduled in such a way that each machine has at most one idle period.

Pinedo [8] gives a rule which also will find an optimal schedule. The rule *LAPT* (longest alternate processing time) selects for processing on a machine the job which has the longest processing time on the other machine. This algorithm also runs in O(n) time and also has the property that each machine has at most one idle period.

Since this algorithm achieves the lower bound of the optimal makespan, and since the problem $O2|prmp|C_{\max}$ has the same lower bound for an optimal makespan, then it follows that this algorithm also solves the problem $O2|prmp|C_{\max}$.

2.2 $Om||C_{\max}$, for $m > 2$

The most natural progression in complexity from the previous problem is to extend the number of machines from 2 to an arbitrary number m. We show that the problem $O3||C_{\max}$ is NP-hard, and so it follows that $Om||C_{\max}$ is NP-hard for any $m > 2$. The reduction is given by Gonzalez and Sahni [4]:

First, let us define the following problem, which we will call *LOFT*: Given an open shop with $m > 2$ processors, a deadline τ, and a set of n jobs with processing times p_{ij}, is there a nonpreemptive schedule with finish time less than or equal to τ?

In proving *LOFT* NP-complete, we will use the NP-complete problem *PARTITION*: A multiset $S = \{a_1, a_2, ..., a_n\}$ is said to have a partition iff there exists a subset U of the indices 1 through n such that $\sum_{i \in U} a_i = (\sum_{i=1}^{n} a_i)/2$. We assume that each a_i is integral. The *PARTITION* problem is that of determining, for an arbitrary multiset S, whether a partition exists or not.

We now show that if *LOFT* with $m = 3$ is solvable, then so is *PARTITION*. Suppose we have a multiset $S = \{a_1, a_2, ..., a_n\}$. Construct the following open shop problem with $3n + 1$ jobs, $m = 3$ machines, and all jobs but the last having one nonzero task. Job $3n + 1$ has three nonzero tasks:

$$\begin{array}{llll} p_{1j} = a_j & p_{2j} = 0 & p_{3j} = 0, & 1 \le j \le n \\ p_{1k} = 0 & p_{2k} = a_{k-n} & p_{3k} = 0, & n+1 \le k \le 2n \\ p_{1l} = 0 & p_{2l} = 0 & p_{3l} = a_{l-2n}, & 2n+1 \le l \le 3n \\ p_{1,3n+1} = T/2 & p_{2,3n+1} = T/2 & p_{3,3n+1} = T/2 & \end{array}$$

where $T = \sum_{i=1}^{n} a_i$, and let $\tau = 3T/2$.

We now show that the above problem has a schedule with makespan less than or equal to τ if and only if S has a partition. Suppose S has a partition, and let U, V be sets so that $\sum_{i \in U} a_i = (\sum_{i=1}^{n} a_i)/2$, and $V = S \backslash U$. Then a feasible schedule with makespan $3T/2$ can be found as follows: schedule the tasks of Job $3n+1$ on Machines 1, 2 and 3 in order, so that the tasks finish at time $T/2, T$, and $3T/2$ respectively. Schedule Jobs $1, 2, ..., n$ on Machine 1 on the interval $[T/2, 3T/2]$, and Jobs $2n + 1, 2n + 2, ..., 3n$ on Machine 3 on the interval $[0, T]$. Finally, schedule on Machine 2 Jobs $n + 1, n + 2, ..., 2n$ which are

associated with the indices in set U on the interval $[0, T/2]$ and the jobs associated with indices in V on the interval $[T, 3T/2]$.

Since $\sum_{i \in U} a_i = (\sum_{i=1}^{n} a_i)/2$, we have that each machine will be busy on the interval $[0, 3T/2]$, and the makespan of the schedule is $\tau = 3T/2$.

Suppose now that a schedule of makespan τ exists. Since the total processing time of Job $3n + 1$ is $3T/2$, then its tasks must start at times $0, T/2$ and T to finish by time $3T/2$. Then the remaining jobs on some machine must be processed on the intervals $[0, T/2]$ and $[T, 3T/2]$. Since there is no preemption, and the total processing time of the respective tasks of the remaining jobs is $\sum_{i \in S} a_i = T$, then it follows that there is no idle time on the given intervals. Hence, a partition of S can be found by taking the indices of the jobs processed on the interval $[0, T/2]$ (with the job indices adjusted accordingly to lie within the range 1 through n).

Thus, we have shown that the problem $O3||C_{\max}$ is NP-hard, and it follows that the more general problem $Om||C_{\max}$ is also NP-hard.

2.3 $O2|r_j|C_{\max}$

Next we consider what happens when not all jobs are available for processing at time 0. Each job j now has a release time r_j. That is, Job j may not be scheduled on any machine before time r_j. It is shown by Lawler et al. [5] that the problem $O2||L_{\max}$ is NP-hard. We show here that $O2|r_j|C_{\max}$ is also NP-hard by reducing the former problem to it.

Suppose we have an instance of $O2||L_{\max}$. Let Job j have processing time a_j on Machine 1 and b_j on Machine 2 as usual, and have due date d_j. Define an instance of $O2|r_j|C_{\max}$ as follows. Let $a'_j = a_j$ and $b'_j = b_j$ be the processing times of Job j on Machines 1 and 2 respectively, and let $r'_j = \max\{d_j | j = 1, 2, ..., n\} - d_j$ be the release time of Job j.

Suppose that $O2|r_j|C_{\max}$ can be solved in polynomial time; let $C'_{\max}$ be the makespan of an optimal schedule S' for the above problem formulation. Construct a schedule for $O2||L_{\max}$ by mapping the interval $[t_1, t_2]$ of S' to the interval $[C'_{\max} - t_2, C'_{\max} - t_1]$ in the new schedule S. Since the the processing done in S' falls in the interval $[0, C'_{\max}]$, then it follows that the processing done in S also falls in the interval $[0, C'_{\max}]$, and so $C_{\max} = C'_{\max}$.

We have that Job j will be finished at time $C'_{\max} - r_j = C_{\max} - (\max\{d_j | j = 1, 2, ..., n\} - d_j) = L_{\max} + d_j$. Since $C'_{\max}$ is minimal by the assumption that S' is optimal, then we have that $L_{\max}$ is also minimal, and so S is an optimal schedule for $O2||L_{\max}$. However, since we have that $O2||L_{\max}$ is NP-hard, it must follow that our original assumption was wrong and so $O2|r_j|C_{\max}$ is also NP-hard.

2.4 $Om|prmp|C_{\max}$

We have shown that $Om||C_{\max}$ is NP-Hard for all $m > 2$. Next we look at the case with m machines where preemption is allowed. Again, we find that the optimal makespan for an instance of this problem can be stated in a closed form of the processing times. We have

$$C_{\max} = \max_{i,j}\{W_i, P_j\}$$

where $W_i = \sum_{j=1}^{n} p_{ij}$ is the workload of Machine i, and $P_j = \sum_{i=1}^{m} p_{ij}$ is the total processing time of Job j.

An algorithm is presented by Gonzalez and Sahni [4], which utilizes maximum edge matchings in a bipartite graph. A similar algorithm using matrices and decrementing sets is also explained in Pinedo's textbook [8]. The latter is presented here.

Let P be an $m \times n$ matrix of the processing times p_{ij}. Row i or Column j is called *tight* if the sum of its entries is equal to $C = \max_{i,j}\{W_i, P_j\}$, otherwise it is called *slack*. We start by finding a *decrementing set* for P. A decrementing set is a set of entries containing exactly one entry from each tight row and column, and at most one entry from each other row or column. Such a decrementing set can always be found for a nonnegative matrix, though the proof is omitted.

When the decrementing set is found, we find Δ so that if Δ is subtracted from all the entries in the decrementing set, the resulting matrix P' has one more zero entry or one more tight row or column than P. The schedule corresponding to the decrementing set and the value Δ includes the processing of the corresponding tasks in the decrementing set on the interval $[0, \Delta]$, and then recursively running the algorithm on the newly obtained matrix P' of the remaining processing time of all tasks.

Since Δ is integral, we have that this process terminates, and it will do so in polynomial time.

2.5 $Om|r_j, prmp|C_{\max}$

The next problem we look at takes the previous problem, m machines with preemption, and adds release times. Let the release times of the jobs be given by $r_j, j = 1, 2, ..., n$. Of these, let $a_1 < a_2 < ... < a_p$ be all the distinct release times. The algorithm finds, for a given value $a_{p+1} > a_p$, whether a feasible schedule exists so that all jobs are completed by time a_{p+1}. Thus, we may adjust the value of a_{p+1} until we find a value C for which there is a feasible schedule for $a_{p+1} = C$, but no schedule for any value of a_{p+1} less than C. We assume that the release times are integral, and so the makespan of the optimal schedule is also optimal.

Let $I_k = a_{k+1} - a_k$ for $k = 1, 2, ..., p$. Let x_{ijk} represent the amount of processing of job j on machine i during the interval $[a_k, a_{k+1}]$. Then consider the following linear program:

$$
\begin{array}{lllll}
\sum_{i=1}^{m} x_{ijk} \leq I_k & 1 \leq j \leq n & 1 \leq k \leq p & (1) \\
\sum_{j=1}^{n} x_{ijk} \leq I_k & 1 \leq i \leq m & 1 \leq k \leq p & (2) \\
\sum_{k=1}^{p} x_{ijk} = p_{ij} & 1 \leq j \leq n & 1 \leq i \leq m & (3) \\
x_{ijk} \geq 0 & \text{if } r_j \leq a_k & & (4) \\
x_{ijk} = 0 & \text{if } r_j > a_k & & (5)
\end{array}
$$

Constraint (1) requires that each machine is assigned no more processing time than the interval length for any given interval. Constraint (2) requires that each job is scheduled for no more than the interval length for any given interval. Constraint (3) requires that each task is finished before time a_{p+1}. Constraints (4) and (5) require that a job is not processed before its release time.

The method to find the makespan is as follows. Find a value C so that a feasible solution for the above linear program exists for a_{p+1}, but no feasible solution exists for any value less than C (assuming integral values, such a value can be found in time logarithmic in the job lengths). Given a feasible solution, use the algorithm for $Om|prmp|C_{\max}$ on each of the p intervals $[a_k, a_{k+1}]$, and combine the solutions to create a feasible schedule. Since each use of the algorithm is polynomial time, then the total running time of this part of the algorithm is linear in p, the number of intervals (which is bounded above by n, the number of jobs). Thus the above algorithm is polynomial.

3 Open Shop Scheduling with Controllable Processing Times

3.1 $O2|contr, C_{\max} \leq C|K$

Now we consider the problem of finding an optimal schedule on two machines when we are able to control the processing times of each task. However, compressing a task incurs some additional cost. Hence the problem is to minimize the makespan while not exceeding a given cost, or minimizing the cost while not exceeding a given makespan. Cheng and Shakhlevich [1] give an algorithm for the latter problem which runs in $O(n)$ time, and also give an algorithm which finds all the breakpoints of the related bicriteria problem which runs in $O(n\log n)$ time. The algorithms are described here.

Let a_j and b_j be the processing times of job j on Machines 1 and 2 respectively. Let $\overline{a}_j$ and $\underline{a}_j$ be the upper and lower bound for the possible values of processing time for Job j on Machine 1, and similarly define $\overline{b}_j$ and $\underline{b}_j$ for Machine 2. Let the cost of compressing Job j on Machine 1 by one unit be given by α_j, and on Machine 2 by β_j. For given processing times a_j and b_j of Job j, let $u_j = \overline{a}_j - a_j$ and $v_j = \overline{b}_j - b_j$ be the compression amounts of Job j on Machines 1 and 2 respectively. Let $K = \sum_{j=1}^{n} (\alpha_j u_j + \beta_j v_j)$ be the total compression cost. Thus, the problem of minimizing K while obtaining a makespan no larger than some fixed value C can be represented as $O2|contr, C_{\max} \leq C|K$.

We know from an algorithm presented earlier in this paper that if we are given the

processing times for each job, then the minimum makespan can be phrased as

$$C_{\max} = \max\{\sum_{j\in N} a_j, \sum_{j\in N} b_j, \max_j\{a_j + b_j\}\}.$$

Futhermore, we can find a schedule with such a makespan in which each machine has at most one idle period (including time between the end of a machine's workload and the makespan). As such, we may create an artificial job $n+1$ whose minimum processing time is 0 on both machines, the maximum processing time is arbitrarily large (say equal to the target makespan value C), and the cost to compress a task of the job is 0. This simplifies the problem slightly by allowing us to consider schedules with $\sum_{j=1}^{n} a_j = \sum_{j=1}^{n} b_j = C$.

The problem of minimizing the compression cost then becomes equivalent to the following linear programming problem:

$$\begin{array}{lll} LP(AB): \text{ maximize} & \sum_{j\in N} (\alpha_j u_j + \beta_j v_j) & \\ \text{subject to} & \sum_{j\in N} \underline{a}_j + u_j) = C & \text{(I)} \\ & \sum_{j\in N} (\underline{b}_j + v_j) = C & \text{(II)} \\ & (\underline{a}_j + u_j) + (\underline{b}_j + v_j) \leq C, j \in N & \text{(III)} \\ & 0 \leq u_j \leq \overline{a}_j - \underline{a}_j, j \in N & \text{(IV)} \\ & 0 \leq v_j \leq \overline{b}_j - \underline{b}_j, j \in N & \text{(V)} \end{array}$$

Ignoring the constraint (III), we can separate the above LP into two continuous knapsack problems, one for each machine. If the solution induced by the two knapsack problems does not violate constraint (III), then it follows that such a solution will be optimal to the LP (since we can only hope to improve on an optimal solution by removing constraints).

If the combined solutions do not satisfy constraint (III), then there is exactly one Job k such that $(\underline{a}_k + u_k) + (\underline{b}_k + v_k) > C$. We call such a job *critical*. Cheng and Shakhlevich prove that if a critical job exists, then there is a solution $(\mathbf{u}^*, \mathbf{v}^*)$ to *LP(AB)* such that $(\underline{a}_k + u_k^*) + (\underline{b}_k + v_k^*) = C$, and also state as a corollary that such a solution must not satisfy $u_k^* = v_k^* = 0$ nor satisfy $u_k^* = \overline{a}_k - \underline{a}_k$ and $v_k^* = \overline{b}_k - \underline{b}_k$ (that is, it is neither the case that both tasks of Job k are both fully compressed or both fully decompressed).

The paper goes on to describe a new linear programming problem, $LP'(AB)$, that does not include Job k, and proves that it is equivalent to $LP(AB)$, determining the values of u_k and v_k based on the values of the remaining variables. The new linear problem is an instance of the *generalized upper bound resource allocation problem*, which is solvable in $O(n)$ time.

Thus, the algorithm is as follows:

Step 1: Find solutions $\mathbf{u}^A, \mathbf{v}^B$ to the two continuous knapsack problems associated with $LP(AB)$.

Step 2: If $\mathbf{u}^A$ and $\mathbf{v}^B$ satisfy constraint (III), then return $\mathbf{u}^* = \mathbf{u}^A, \mathbf{v}^* = \mathbf{v}^B$.

Step 3: Else, determine a critical job k, formulate $LP'(AB)$, and find its solution $u'_j, v'_j, j \in N\backslash\{k\}$. Let $u'_k - \sum_{j\in N\backslash\{k\}} (\underline{b}_j + v_j) - \underline{a}_k$, and $v'_k - \sum_{j\in N\backslash\{k\}} (\underline{a}_j + u_j) - \underline{b}_k$. Return $\mathbf{u}^* = \mathbf{u}', \mathbf{v}^* = \mathbf{v}'$.

The continuous knapsack problems can each be solved in $O(n)$ time, as can the determination of a critical job and the problem $LP'(AB)$. Thus, the running time of the algorithm is $O(n)$.

3.2 $O2|contr|(C_{\max}, K)$

The previous algorithm will provide us with an optimal solution only when we know what value we wish our makespan to take. We may be more interested in finding a variety of combinations of the objective values $C_{\max}$ and K. In this case, we would want to find all the *Pareto optimal* solutions. We say that a schedule S is Pareto optimal if there does not exist a schedule S' such that $C_{\max}(S') \leq C_{\max}(S)$ and $K(S') \leq K(S)$ where one of these two inequalities is strict.

Recall that the algorithm for the problem in the previous section will attempt to find an optimal schedule by solving a continuous knapsack problem for each machine. As we solve these problems for varying values of C, the algorithm will always choose to decompress the jobs which yield the best change in the objective value, and will not decompress the jobs which improve the objective function the least. If we sort the tasks on Machines 1 and 2 respectively so that $\alpha_{i_1} \geq \alpha_{i_2} \geq ...\alpha_{i_{n+1}} = 0$ and $\beta_{j_1} \geq \beta_{j_2} \geq .... \geq \beta_{j_{n+1}} = 0$, then we can decompress the tasks in the order $i_1, i_2, ..., i_{n+1}$ and $j_1, j_2, ..., j_{n+1}$ until decompressing a job any further will not satisfy $\sum_{j=1}^{n} a_j = \sum_{j=1}^{n} b_j = C$. That is, in any optimal schedule, there will be two tasks i_l and j_m that are "next in line" to be decompressed if some slack became available. In the case that the optimal schedule for a given value of C has a critical job, then it is the case that if the tasks are listed in the above order, excluding the tasks of the critical job, then the same result will follow, with the compression values of the critical job being determined by the compression rates of the other jobs.

Using the above fact, we see that the structure of the tasks ordered in such a way will take on one of four different types. If there is no critical job, then the tasks will be decompressed in order, and we will have on each machine a group of fully decompressed jobs, followed by i_l (or j_m respectively), followed by a group of fully compressed jobs. We will call this schedule Type 1. If there exists a critical job, then one ofthe following three structures will exist. Either both tasks of job k will reside in the group of fully decompressed jobs, in which case we would like to further compress the tasks but cannot without violating constraint (III) (Type 2a), or we would like to further compress the task on Machine 2 and a different task $i_l \neq k$ on Machine 1 (Type 2b), or we would like to further compress the task on Machine 1 and a different task $j_m \neq k$ on Machine 2 (Type 2c). In Types 2b and 2c, if we would like to compress a task other than the respective task of Job k, then this implies that either the task of Job k is fully decompressed or still fully compressed.

We can find the first Pareto optimal point by taking the minimum makespan value $C^0 = C_{\max} = \max\{\sum_{j\in N} \underline{a}_j, \sum_{j\in N} \underline{b}_j, \max_j\{\underline{a}_j + \underline{b}_j\}\}$ and running the algorithm for the problem $O2|contr, C_{\max} \leq C^0|K$.

From this first point, we can move to the next breakpoint by decompressing one task on each machine at the same rate; in particular, we will decompress the task that offers the

best improvement in the objective function. For each type of schedule, this pair of jobs is given by

$$(i,j) = \begin{cases} (i_l, j_m) & \text{if } S^0 \text{ is of Type 1} \\ (i_l, k) & \text{if } S^0 \text{ is of Type 2b} \\ (k, j_m) & \text{if } S^0 \text{ is of Type 2c} \\ (i_l, k) & \text{if } S^0 \text{ is of Type 2a and } \alpha_{i_l} + \beta_k \geq \beta_{j_m} + \alpha_k \\ (k, j_m) & \text{if } S^0 \text{ is of Type 2a and } \alpha_{i_l} + \beta_k < \beta_{j_m} + \alpha_k \end{cases}$$

We decompress Tasks i and j by the same amount z, given by

$$z = \begin{cases} \min\{\overline{a}_i - a_i, \overline{b}_j - b_j\}, & \text{if } i \neq j, \\ \min\{\overline{a}_i - a_i, \overline{b}_j - b_j, C^0 - (a_i + b_j)\}, & \text{otherwise.} \end{cases}$$

The makespan of the next Pareto optimal point is $C^1 = C^0 + z$. Solving the problem $O2|contr, C_{\max} \leq C^1|K$, we can find the corresponding compression cost K^1 and Schedule S^1. By repeating the steps to go from the first Pareto optimal point to the second, we can find each Pareto optimal point, until all tasks have been fully decompressed.

Thus, the algorithm can be described as follows:

Order the tasks on each machine by decreasing values of α_j and β_j respectively. Find the solution to the problem $O2|contr, C_{\max} \leq C^0|K$ and find the compression values a_j, b_j for each task using the algorithm in the previous section, and determine the type of the corresponding schedule. Let $k = 1$. While there are still compressed tasks, find the pair of tasks (i, j) to be decompressed, and determine the compression amount z. Update the values as follows:

$$\begin{array}{ll} a_i := a_i + z, & C^k := C^{k-1} + z \\ b_j := b_j + z, & K^k := K^{k-1} - (\alpha_i + \beta_j)z \\ k := k + 1. & \end{array}$$

Sorting the jobs takes $O(n\log n)$ time. The number of iterations is limited to the number of times a task becomes fully decompressed, or a job becomes critical, so does not exceed $3n + 1$. Solving the initial problem takes $O(n)$ time, and each other operation in the algorithm takes constant time. Hence the algorithm for finding the breakpoints of the Pareto optimal solutions is $O(n\log n)$.

4 Flexible Open Shop Scheduling

4.1 Introduction

Next we consider the flexible open shop problem $FO2|prmp|C_{\max}$, as well as the related problem $Pm|prmp|C_{\max}$. Recall that, in a two machine open shop, we have n jobs to be

processed and each job j requires a_j units of processing on the first machine and b_j units of processing on the second machine. In the flexible open shop, there are $m+1$ machines comprising two stages. Machine 0 comprises Stage 1, while machines $1, 2, ..., m$ comprise Stage 2. Job j has two tasks which must be completed. Task 1 of Job j requires a_j units of processing on Stage 1, and Task 2 of Job j requires b_j units of processing on Stage 2 which may be completed by any number of the machines $1, 2, ..., m$ (that is, Stage 2 models an m-machine parallel machine environment). We retain the restriction that a job may not be processed on both stages at the same time, nor may it be processed by more than one machine in a stage at the same time. Our goal is to find a schedule minimizing the makespan $C_{\max}$, defined as the time at which no job requires further processing.

We show that the makespan of an optimal schedule is given by

$$C^*_{\max} = \max\{\sum_j a_j, \frac{1}{m}\sum_j b_j, \max_j\{a_j + b_j\}\}$$

and we also give an O(mn)-time algorithm for finding a schedule with such a makespan.

4.2 Solving $Pm|prmp|C_{\max}$

It is helpful to study the problem $Pm|prmp|C_{\max}$ before looking at the Flexible Open Shop problem. We do so because we have noted that in $FO2|prmp|C_{\max}$, Stage 2 models the same environment as $Pm|prmp|C_{\max}$. In the problem $Pm|prmp|C_{\max}$, there are again n jobs, but each job has only one task which requires p_j units of processing on any of the m machines. A job may not be processed on two machines at the same time. The problem has been implicitly solved in that $Pm|prmp|C_{\max}$ reduces from $P|tree, prmp|C_{\max}$ which has ben shown to be polynomially solvable ([3],[7]).

Lemma 1: The makespan of a feasible schedule for $P2|prmp|C_{\max}$ is no less than

$$C_2 = \max\{\frac{1}{m}\sum_j p_j, \max_j\{p_j\}\}.$$

Proof. The total processing time required is $\sum_j p_j$, and if this workload is divided evenly among the m machines, then each machine will finish at time $\frac{1}{m}\sum_j p_j$. If one machine does less than this amount of processing, then some other machine will do more as a consequence, and at least one machine will end later than time $\frac{1}{m}\sum_j p_j$. Since a job may not be processed on two machines at once, then no job j may finish before time p_j, and so one machine may not finish before time $\max_j\{p_j\}$. Hence, the makespan of any feasible schedule is no less than C_2.

□

We now present a straightforward algorithm to solve $Pm|prmp|C_{\max}$.

Algorithm 2:

Step 1: Determine the value C_2 as in Lemma 2. Let $i := 1$.

Step 2: For $j := 1$ to n, while Job j is not yet finished,

(a) Schedule Job j on Machine i until it is completed or until time C_2, whichever occurs first.

(b) If Job j is not yet completed, preempt it. If time C_2 is reached, let $i := i+1$.

Theorem 3: Algorithm 2 produces an optimal schedule for $Pm|prmp|C_{\max}$.

Proof. Since there are m machines, then there are $mC_2 \geq \sum p_j$ units of processing available. If Job j first begins processing on some machine at time t, then it will continue being processed until it is finished or until time C_2. If time C_2 is reached, then Job j will be processed on the entire interval $[t, C_2]$ and will be completed on the interval $[0, t]$ on the next machine. If the job is not completed on this interval, then p_j would exceed C_2, a contradiction. Thus, no job is scheduled on two machines concurrently. Since no feasible schedule can have makespan less than C_2 and Algorithm 2 produces a feasible schedule with makespan C_2, then the scheduled produced by Algorithm 2 is optimal. □

4.3 Solving $FO2|prmp|C_{\max}$

By combining ideas found in the above algorithm, we can now solve the problem $FO2|prmp|C_{\max}$. We start again by finding a lower bound for the makespan of a feasible schedule.

Lemma 4: The makespan of a feasible schedule for $FO2|prmp|C_{\max}$ is no less than

$$C = \max\{\sum_j a_j, \frac{1}{m}\sum_j b_j, \max_j\{a_j + b_j\}\}.$$

Proof. The earliest completion time of the first machine is no less than $\sum_j a_j$. The completion time of any one job is no less than $\max_j\{a_j + b_j\}$. The completion time of the latest machine among $1, 2, ..., m$ is no less than $\frac{1}{m}\sum_j b_j$ (with equality only if the workload is split evenly among the m machines). Hence, any feasible schedule may not finish before time C. Thus, the makespan of a feasible schedule is no less than C. □

We now present an algorithm which finds an optimal schedule.

Algorithm 5:

Step 1: Determine the value C as in Lemma 4.

Step 2: Schedule jobs in the order $1, 2, ..., n$ on Machine 0. Let f_j be the time at which Job j finishes on Machine 0 and let $f_0 = 0$, so that Job j is scheduled on Machine 0 on the interval $[f_{j-1}, f_j]$.

Step 3: Let $t := 0$, and let $i := 1$.

Step 4: For l from 1 to n, while Job l is not completed,

(a) If $i \leq m$, then

- Skipping over the interval $[f_{l-1}, f_l]$, schedule Job l on Machine i from time t onward until the task is completed or time C is reached, whichever occurs first.
- If the task is completed, then let t be the time of completion the task.
- If time C is reached, preempt the task if it is not yet finished. Let $t := 0$ and $i := i + 1$. If $i > m$, let $k := l$.

(b) Else, if $i > m$,

- Find an interval $[t_1, t_2]$ upon which a machine is idle, and schedule job l on the idle machine starting from time t_1 until the job is completed or until time t_2, whichever occurs first.

Step 5: Let A be the set of distinct intervals upon which job k is processed on two machines at once. Let B be the set of distinct intervals upon which job k is not processed.

Step 6: While A is not empty,

(a) Choose an interval $[t_1, t_2]$ from A. Let $m_1 \in \{1, 2, ..., m\}$ be a machine upon which job k is processed on $[t_1, t_2]$. Choose an interval $[s_1, s_2]$ from B.

(b) Let $d = \min\{t_2 - t_1, s_2 - s_1\}$.

(c) If $t_2 - t_1 > d$, then split $[t_1, t_2]$ into two smaller intervals $[t_1, t_1 + d]$ and $[t_1 + d, t_2]$. Remove $[t_1, t_2]$ from A and add $[t_1, t_1 + d]$ and $[t_1 + d, t_2]$ to A.

(d) Else if $s_2 - s_1 > d$, then split $[s_1, s_2]$ into two smaller intervals $[s_1, s_1 + d]$ and $[s_1 + d, s_2]$. Remove $[s_1, s_2]$ from B and add $[s_1, s_1 + d]$ and $[s_1 + d, s_2]$ to B.

(e) Choose a machine $m_2 \in \{1, 2, ..., m\}$ which, on the interval $[s_1, s_1 + d]$, is either idle or processing a job which is not processed on $[t_1, t_1 + d]$.

(f) Swap the operations of Machine m_1 on $[t_1, t_1+d]$ and Machine m_2 on $[s_1, s_1+d]$ with one another.

(g) Remove $[t_1, t_1 + d]$ from A and remove $[s_1, s_1 + d]$ from B.

Lemma 6: There are at most k idle intervals created in Step 4(a) of Algorithm 5, and they are all pairwise disjoint.

Proof. In Step 4(a) of the algorithm, jobs are scheduled back to back, with the exception that Job j is not scheduled on the interval $[f_{j-1}, f_j]$. Since jobs are preempted only at time C, in which case they are started again on the next machine at time 0, then we have that no job will be preempted at time f_{j-1} more than once (otherwise, the job will require processing in excess of $|[f_{j+1}, f_j] \cup [f_j, C] \cup [0, f_{j-1}]| = C$, a contradiction to our choice of C). Thus, there will be at most one idle period for each of the k jobs scheduled by Step 4(a), and each such interval is contained in one of the pairwise disjoint intervals $[f_{j-1}, f_j]$. □

Lemma 7: At any point in Algorithm 5, at most one job k will be scheduled on any two machines at the same time.

Proof. Any job which is scheduled entirely by Step 4(a) is scheduled in such a way to avoid scheduling the job on two machines at the same time. Any job which is scheduled entirely by Step 4(b) is scheduled only on the idle intervals created within the previous loop, which are pairwise disjoint by Lemma 6. When two operations are swapped by Step 6 of the algorithm, the swapped jobs are chosen in such a way to avoid scheduling any job on two machines at the same time. Thus, the only job which may be scheduled on two machines at once is the job which is partially scheduled by each of the methods in Step 4; namely, the job denoted k in the algorithm. □

Theorem 8: Algorithm 5 generates an optimal schedule.

Proof. So long as Algorithm 5 returns a feasible schedule, then the resulting schedule is optimal by Lemma 7. By choice of C, we know that the workload of Stage 1 will be finished in the interval $[0, C]$. Similarly, the workload of Stage 2 will be finished in the interval $[0, C]$, since the algorithm will not leave any idle periods on the Stage 2 machines. The steps of the algorithm ensure that no job is scheduled both on Machine 0 and one of the machines $1, 2, ..., m$ concurrently. Thus, we need only prove there exists an interval $[s_1, s_2]$ and a machine m_2 as described in Steps 6(a) and 6(e) of the algorithm.

Let S be the (possibly infeasible) schedule produced by Step 5 of the algorithm, and suppose job k is processed on two machines at once, say on the interval $[t_1, t_2]$. If it were the case that there does not exist an interval $[s_1, s_2]$ upon which k is not processed, then k is processed on the entire interval $[0, C]$. Since k is processed by two machines on the interval $[t_1, t_2]$, then we have that the required processing time of Job k is at least $C + (t_2 - t_1) > C$, a contradiction by choice of C. Thus, there must exist some interval $[s_1, s_2]$ upon which k is not processed.

If one of the machines among $1, 2, .., m$ is idle on the interval $[s_1, s_2]$, then we are done. Otherwise, since we assume k is not processed on one of the machines $1, 2, ..., m$ on the interval $[s_1, s_2]$, then by Lemma 5 we know that the machines $1, 2, ..., m$ are processing m distinct jobs. On the interval $[t_1, t_2]$, there are 2 machines processing k, and so there are at most $m - 1$ distinct jobs besides k processed on this interval, and so there must exist a machine among $1, 2, ..., m$ which processes some job not processed on the interval $[t_1, t_2]$. □

Theorem 9: Algorithm 5 finishes in O(mn) time.

Proof. First, note that the end of an idle period created in the loop in Step 4(a) will correspond to the completion of a task on the first machine. This is because the idle period occurs only when a job skips past the period due to the fact that the same job is processed on Stage 1. Once the job is completed on Stage 1, the job will be processed on Stage 2. Thus, the times at which jobs are preempted in Step 4(b) correspond to the time at which a task on Stage 1 finishes.

The structure of the schedule changes only when a machine stops processing a job, and this occurs precisely when a task is finished or a job is preempted. It follows that the number of maximal intervals upon which the structure of the schedule does not change is at most the sum of the number of times at which a task is completed (1 for each of the $2n$ tasks) and the number of distinct times a job is preempted (only 1, since jobs are preempted only at time C in Step 4(a) or at a time corresponding to the completion of a task in Step 4(b)). That is, the number of maximal intervals upon which the structure of the schedule does not change is given by $2n + 1$.

Letting A be the set of intervals upon which Job k is processed on two machines concurrently, and letting B be the set of intervals upon which Job k is not processed on any machine, we note that $|A| + |B|$ can not exceed the number of intervals, and so $|A| + |B| \leq 2n + 1$. During each iteration of Step 6, a swap occurs whereby either an interval is fully removed from A (when the inequality in Step 6(d) is satisfied), or an interval is fully removed from B (when the inequality in Step 6(c) is satisfied). Thus, the number of iterations of the loop does not exceed $2n + 1$.

Thus the work required is as follows. There are no more than m preemptions in Step 4(a), resulting in no more than m additional iterations. At most k idle periods are created in Step 4(a), and so there are no more than k preemptions caused by Step 4(b). Thus, the loop iterates no more than $n + m + k \leq 2n + m$ times, and each iteration does constant work. Creating the sets A and B in Step 5 can be done in $O(nm)$ time. Step 6 iterates no more than $2n + 1$ times, and constant work is done in each iteration. Thus, the workload is done in $O(2n + m + nm + 2n + 1) = O(nm)$ time. □

4.4 Example

We will use Algorithm 5 to find an optimal schedule for the following instance of $FO2|prmp|C_{\max}$ with 4 jobs and 4 machines, where Machines 2, 3, and 4 form Stage 2.

job	1	2	3	4	5	6
a_j	0	3	3	2	1	1
b_j	5	5	7	3	8	2

We find that $C = 10 = \sum a_j = \frac{1}{3} \sum b_j = a_3 + b_3$, so our schedule will have a makespan of 10. We first schedule the jobs back-to-back on Machine 1. Using the algorithm, we have no problem scheduling Jobs 1 and 2 on Machine 2. While processing Job 3 on Machine 3, we must preempt the job at time 3 and start it again at time 6, due to the fact that Job 3 is processed on Stage 1 during the interval [3,6]. Similarly, we preempt Job 5 during its processing on Machine 4. While processing Job 5 on Stage 2, we reach time 10 on the last machine. We let $k = 5$, and progress to step 4(b) of Algorithm 5.

We are still processing Job 5 when step 4(b) begins. We find an idle interval which was created in step 4(a); Machine 3 is idle on the interval [3,6]. Upon completion of Job 5, we do the same for Job 6, using the intervals [5,6] and [8,9] on Machines 3 and 4 respectively. Figure 1 shows the state of the partial schedule at this point (processing assigned in step

4(b) is shown in grey).

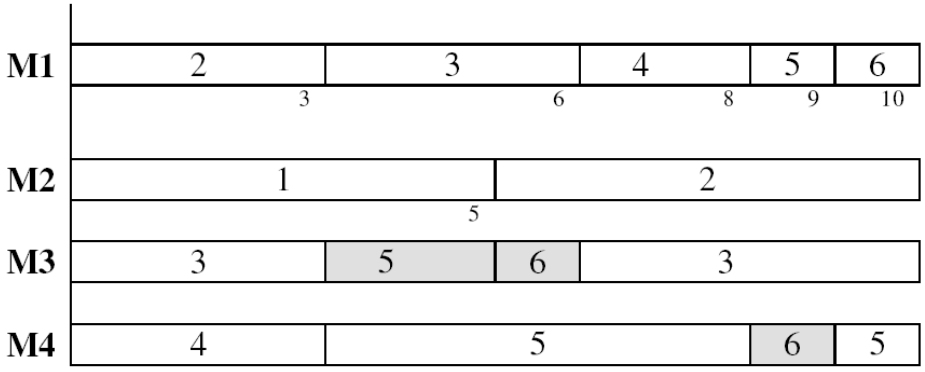

Figure 1.

In step 5, we create the sets $A = \{[3, 5]\}$ and $B = \{[0, 3]\}$, corresponding to the intervals where Job 5 is processed on more than one machine and no machines respectively. We choose the single interval in each set, and find the value $d = \min\{5-3, 3-0\} = 2$. Thus, we break the interval [0,3] into the two smaller intervals [0,2] and [2,3], and update the set $B = \{[0, 2], [2, 3]\}$.

On the interval [3,5], Jobs 1, 3, 5 are processed. On [0,2], Jobs 1,3,4 are processed on Stage 2. Since Job 4 is not processed on [3,5], we can safely swap the processing of Job 4 on the interval [0,2] with the processing of Job 5 on [3,5]. After doing so, we remove the intervals from the sets A and B. We now have that set A is empty, and so we rest assured knowing that we are left with a feasible schedule with minimal makespan, as depicted in Figure 2.

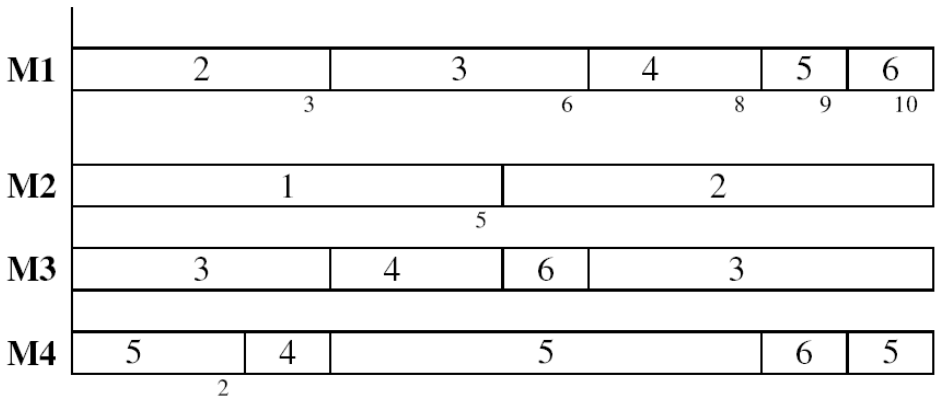

Figure 2.

4.5 Optimal Number of Stage 2 Machines

In a related problem, we may have control over the number of machines available for use in Stage 2. In general, adding machines will decrease the completion time of the schedule,

so some cost is usually associated with adding machines. The simplest cost function in such a case would be

$$K(m) = \alpha C^*_{\max}(m) + \beta m.$$

In this cost function, $C^*_{\max}(m)$ represents the makespan of an optimal schedule in which Stage 2 has m machines. The values α and β represent the incurred cost of delaying the schedule by one unit of time and the cost of acquiring an additional machine respectively. Given an instance of $FO2|prmp|C_{\max}$ and values α and β, we wish to find the value m^* which minimizes the cost function $K(m)$.

Clearly we must have that $m \geq 1$, and an upper bound for m presents itself; we know that the makespan of an optimal schedule for $FO2|prmp|C_{\max}$ is given by the formula

$$C^*_{\max} = \max\{\sum_j a_j, \frac{1}{m}\sum_j b_j, \max_j\{a_j + b_j\}\}.$$

As m goes off to infinity, we have that the Stage 2 component of the formula goes to zero, and the other two components determine the makespan. That is, once we have that

$$\max\{\sum_j a_j, \max_j\{a_j + b_j\}\} \geq \frac{1}{m}\sum_j b_j,$$

then adding more machines will not help to decrease the makespan. Solving for m, we get that adding more machines will not decrease the makespan of an optimal schedule once m reaches the value

$$m' = \left\lceil \min\left\{ \frac{\sum_j b_j}{\sum_j a_j}, \min\left\{ \frac{\sum_j b_j}{a_j + b_j} \right\} \right\} \right\rceil.$$

So long as $m < m'$, we must have that $C^*_{\max}(m)$ is equal to $\frac{1}{m}\sum_j b_j$. Thus, our problem is to find the value m^* which minimizes the function

$$K(m) = \alpha \frac{1}{m}\sum_j b_j + \beta m.$$

Using elementary calculus, we use the derivative of the function to determine when the function is minimized or maximized,

$$K'(m) = -\alpha \frac{1}{m^2}\sum_j b_j + \beta$$

$$K''(m) = 2\alpha \frac{1}{m^3}\sum_j b_j$$

We have that $K'(m) = 0$ when $m = \pm\sqrt{\frac{\alpha \sum_j b_j}{\beta}}$. Clearly only the positive value of m interests us. When m takes on this value, the second derivative is positive, indicating

that the point represents a local minimum. Since we require that m^* be an integer, we simply check the value of the cost function $K(m)$ at the values $\left\lfloor \frac{\alpha \sum_j b_j}{\beta} \right\rfloor$ and $\left\lceil \frac{\alpha \sum_j b_j}{\beta} \right\rceil$ and compare the two values to find the minimum; since the second derivative is positive for all positive values of m, one of these two points is guaranteed to be the absolute minimum for the function.

Upon finding the values m' and m^*, we compare the two; if $m' \leq m^*$, then the optimal number of machines for the problem is m'. Otherwise, the optimal number of machines for the problem is m^*.

4.6 Example

We will now consider the example in section 4.4, now attempting to minimize the operation costs. Let $\alpha = 4$ and $\beta = 25$, to represent that each machine required in Stage 2 incurs a cost of 25, and each unit of time that is required in processing incurs a cost of 4. Our cost function is $K(m) = 4C^*_{\max}(m) + 25m$.

We first find the value m':

$$\begin{aligned} m' &= \left\lceil \min\left\{ \frac{\sum_j b_j}{\sum_j a_j}, \min\left\{ \frac{\sum_j b_j}{a_j + b_j} \right\} \right\} \right\rceil \\ &= \left\lceil \min\left\{ \frac{30}{10}, \frac{30}{10} \right\} \right\rceil \\ &= 3 \end{aligned}$$

This lets us know that having more than 3 machines will not yield a schedule with smaller makespan than an optimal schedule using 3 machines. Next, we find the value m^* which minimizes $K(m)$:

$$\sqrt{\frac{\alpha \sum_j b_j}{\beta}} = \sqrt{\frac{(4)(30)}{(25)}} \approx 2.191$$

Thus, either $m^* = 2$ or $m^* = 3$. We find that $K(2) = 4(\frac{1}{2}(30)) + 25(2) = 110$ and $K(3) = 4(\frac{1}{3}(30)) + 25(3) = 115$. Thus, $m^* = 2$. Since $2 \leq m' = 3$, then we have that the cost is minimized when Stage 2 contains 2 machines.

5 Conclusion

We have shown results about some classical problems in open shop scheduling. We have seen how changing the number of machines affects the complexity of a problem, as well as how adding the processing constraints r_j and $prmp$ affect the complexity of problems. Beyond these changes, there are many more ways to change the processing constraints of the system that render existing algorithms obsolete. Similarly, we have only considered

minimization of the makespan as the objective goal. Many other objective functions can be considered in open shop scheduling to create even more probems.

We have also studied two modern open shop scheduling problems. With the ability to control processing times, we now have a multi-objective problem where we must specify a bound on one of the objectives, or otherwise find all pareto optimal pairs to solve the problem. Despite the increase in variables and decisions, the problem remains to be O(n) when we set a maximum value for the makespan, and O($n$log$n$) if we prefer to find all the notable Pareto optimal solutions.

The other modern problem we have studied is the flexible open shop problem. We have shown that the problem can be solved by an O(n)-time algorithm and have proved the correctness of the algorithm. Many related problems can be considered by altering the machine structure of each stage or by changing the processing constraints. In doing so, many flexible open shop problems may be declared as NP-hard when a related subproblem is already NP-hard. For instance, we may easily conclude that $FO2||C_{\max}$ is NP-hard due to the fact that $Pm||C_{\max}$ is NP-hard [6].

6 References

[1] T.C.E. Cheng and N. Shakhlevich, Two-machine open shop problem with controllable processing times, 2006.

[2] Y. Cho and S. Sahni, Preemptive Scheduling of Independent Jobs with Release and Due Times on Open, Flow and Job Shops, Operations Research Vol. 29, 1981, pp. 511-522.

[3] T. Gonzalez and D.B. Johnson. A new algorithm for preemptive scheduling of trees. J. Assoc. Comput. Mach., 27(2):287-312, 1980.

[4] T. Gonzalez and S. Sahni, Open shop scheduling to minimize finish time, J. Assoc. Comput. Mach. 23 (1976), pp. 665-679.

[5] E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan, Minimizing maximum lateness in a two-machine open shop, Math. Oper. Res., 6(1):153-158, 1981.

[6] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. Ann. of Discrete Math., 1:343-362, 1977.

[7] R.R. Muntz and E.G. Coffman, Jr. Preemptive scheduling of real-time tasks on multiprocessor systems. J. Assoc. Comput. Mach., 17:324-338, 1970.

[8] M.L. Pinedo, Scheduling: Theory, Algorithms, and Systems, Third Edition, Springer, 2008