Discrete Optimization 6 (2009) 292–298

Contents lists available at ScienceDirect

Discrete Optimization

journal homepage: www.elsevier.com/locate/disopt

A fast preemptive scheduling algorithm with release times and inclusive processing set restrictions

Yumei Huo [a], Joseph Y.-T. Leung [b,*], Xin Wang [b]

[a] Department of Computer Science, CUNY at Staten Island, Staten Island, NY 10314, United States

[b] Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, United States

ARTICLE INFO

Article history:
Received 10 December 2007
Received in revised form 16 February 2009
Accepted 18 February 2009
Available online 19 March 2009

Keywords:
Preemptive scheduling
Release time
Inclusive processing set
Makespan minimization
Polynomial-time algorithms

ABSTRACT

We consider the problem of preemptively scheduling n independent jobs on m parallel machines so as to minimize the makespan. Each job J_j has a release time r_j and it can only be processed on a subset of machines $\mathcal{M}_j$. The machines are linearly ordered. Each job J_j has a machine index a_j such that $\mathcal{M}_j = \{M_{a_j}, M_{a_j+1}, \ldots, M_m\}$. We first show that there is no 1-competitive online algorithm for this problem. We then give an offline algorithm with a running time of $O(nk \log P + mnk^2 + m^3k)$, where k is the number of distinct release times and P is the total processing time of all jobs.

1. Introduction

We consider the problem of preemptively scheduling n independent jobs on m parallel machines so as to minimize the makespan. Each job J_j has a release time r_j and it can only be processed on a subset of machines $\mathcal{M}_j$. The machines are linearly ordered. Each job J_j has a machine index a_j such that $\mathcal{M}_j = \{M_{a_j}, M_{a_j+1}, \ldots, M_m\}$. We denote this problem as $P \mid \mathcal{M}_j(inclusive), r_j, pmtn \mid C_{\max}$.

Scheduling problems with inclusive processing set restrictions occur quite often in practice. Hwang et al. [1] give a scenario occurring in the service industry in which a service provider has customers categorized as platinum, gold, silver, and regular members, where "higher-level customers" receive better services. One method of providing such a differentiated service is to label servers and customers with prespecified grade of service (GoS) levels and allow a customer to be served only by a server with GoS level less than or equal to that of the customer. Glass and Kellerer [2] describe a situation of assigning jobs to computers with memory capacity. Each job has a memory requirement and each computer has a memory capacity. A job can only be assigned to a computer with enough memory capacity. Ou et al. [3] consider the process of loading and unloading cargoes of a vessel, where there are multiple nonidentical loading/unloading cranes operating in parallel. The cranes have the same operating speed but different weight capacity limits. Each piece of cargo can be handled by any crane with a weight capacity limit no less than the weight of the cargo.

We consider the case where jobs are released at different times. We first consider online scheduling algorithms; i.e., algorithms that schedule jobs without knowledge of future job arrivals. We show that there does not exist a 1-competitive online algorithm. An online algorithm is said to be 1-competitive if it performs as well as an optimal offline algorithm. We then give a fast offline algorithm with a running time of $O(nk \log P + mnk^2 + m^3k)$, where k is the number of distinct release times and P is the total processing time of all jobs.

* Corresponding address: Department of Computer Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102, United States. Tel.: +1 973 596 3387; fax: +1 973 596 5777.

E-mail addresses: huo@mail.csi.cuny.edu (Y. Huo), joseph.y.leung@njit.edu (J.Y.-T. Leung), xw37@njit.edu (X. Wang).

1572-5286/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.disopt.2009.02.002

1.1. Background and related work

The problem $P \mid \mathcal{M}_j(inclusive), r_j, pmtn \mid C_{\max}$ is a natural generalization of the classical problem $P \mid pmtn \mid C_{\max}$ which can be solved by McNaughton's rule [4]. McNaughton's rule first computes a deadline D which is the larger of two quantities. The first quantity is the maximum processing time of all the jobs, while the second quantity is the sum of the processing times of all the jobs divided by the number of machines. The jobs are then scheduled, one by one, from time 0 until time D. If a job finishes beyond time D, then it will be preempted at time D and continues on another machine from time 0 until it finishes. Hong and Leung [5] have given an optimal algorithm to solve $P \mid pmtn, r_j \mid C_{\max}$. Their algorithm is based on the idea of McNaughton's rule.

The problem we study is a special case of the general problem where $\mathcal{M}_j$ can be any arbitrary subset, denoted by $P \mid \mathcal{M}_j, r_j, pmtn \mid C_{\max}$, which in turn is a special case of $R \mid r_j, pmtn \mid C_{\max}$. (The processing time of job j on machine M_i is p_j if M_i is in $\mathcal{M}_j$; otherwise, it is ∞.) Lawler and Labetoulle [6] have given a linear programming algorithm to solve $R \mid r_j, pmtn \mid C_{\max}$ and hence $P \mid \mathcal{M}_j, r_j, pmtn \mid C_{\max}$. Brucker et al. [7] give a network flow algorithm to solve $P \mid \mathcal{M}_j, r_j, pmtn \mid C_{\max}$ with a running time of $O(n^3(n+m)^3 \log P)$, where P is the total processing time of all jobs. The running time of their algorithm can be improved to $O(mn^2k^2 \log n \log P)$ if we use the fast network flow algorithm of Goldberg and Tarjan [8]. In this paper we give a faster algorithm for $P \mid \mathcal{M}_j(inclusive), r_j, pmtn \mid C_{\max}$ with a running time of $O(nk \log P + mnk^2 + m^3k)$, where k is the number of distinct release times and P is the total processing time of all jobs.

For nonpreemptive scheduling, $P \mid \mathcal{M}_j(inclusive), r_j \mid C_{\max}$ is NP-hard in the strong sense, since $P \mid\mid C_{\max}$ is. If all jobs are released at time 0, there are several approximation algorithms for $P \mid \mathcal{M}_j(inclusive) \mid C_{\max}$. The first approximation algorithm is the Modified Largest Processing Time first algorithm given by Hwang et al. [1]. They show that the algorithm obeys a bound of $\frac{5}{4}$ for $m = 2$ and $2 - \frac{1}{m-1}$ for $m \geq 3$. Subsequently, Glass and Kellerer [2] give a $\frac{3}{2}$-approximation algorithm for this problem. Recently, Ou et al. [3] give an improved algorithm with a worst-case bound of $\frac{4}{3} + \epsilon$, where ϵ is any given positive number. They also give a polynomial time approximation scheme (PTAS) for this problem. Most recently, Li and Wang [9] give a PTAS for $P \mid \mathcal{M}_j(inclusive), r_j \mid C_{\max}$.

1.2. The model and main results

Suppose the machines are labeled from 1 to m. We are given a set of machine intervals $MI = \{MI_1, \ldots, MI_z\}$, where $MI_i = \{M_{i_1}, M_{i_1+1}, \ldots, M_m\}$ is the set of machines with consecutive labels. Each job J_j, $1 \leq j \leq n$, can be represented by a triple (p_j, S_j, r_j), where p_j is the processing time of the job, S_j is the machine interval such that $S_j \in MI$, and r_j is the release time of the job. Job J_j can only be scheduled on the machines in S_j. The objective is to minimize the makespan $C_{\max}$. For each machine interval MI_i, we define $J(MI_i)$ to be the set of jobs whose machine interval is exactly MI_i. That is,

$$J(MI_i) = \{J_j | S_j = MI_i\}.$$

The organization of the paper is as follows. In the next section, we show that there does not exist a 1-competitive online algorithm for $P \mid \mathcal{M}_j(inclusive), r_j, pmtn \mid C_{\max}$. In Section 3, we give the offline algorithm for $P \mid \mathcal{M}_j(inclusive), r_j, pmtn \mid C_{\max}$. Finally, we draw some concluding remarks in Section 4.

2. Online algorithm

In this section we will show that there does not exist a 1-competitive online algorithm for $P \mid \mathcal{M}_j(inclusive), r_j, pmtn \mid C_{\max}$. This is in sharp contrast with $P \mid r_j, pmtn \mid C_{\max}$ which admits a 1-competitive online algorithm due to Hong and Leung [5].

We consider the following example. Let $m = 4$ and let there be two machine intervals $MI_1 = \{M_1, M_2, M_3, M_4\}$ and $MI_2 = \{M_2, M_3, M_4\}$. At time $t = 0$, eight jobs are released: $J_1 = J_2 = J_3 = J_4 = J_5 = J_6 = (3, MI_2, 0)$ and $J_7 = J_8 = (5, MI_1, 0)$. Suppose that there is a 1-competitive online algorithm A. An adversary will observe the schedule produced by A until time $t = 6$. There are two cases to consider.

Case 1: If the first six jobs are all completed by time $t = 6$, then machines M_2, M_3 and M_4 are fully occupied by the six jobs in the time interval [0, 6]. In this case the makespan of the schedule produced by algorithm A is at least 8. (Algorithm A can schedule three time units each for jobs J_7 and J_8 in the time interval [0, 6], and the remaining two units of time in the time interval [6, 8].) However, the optimal schedule has makespan 7. Therefore, algorithm A cannot be 1-competitive.

Case 2: If some job(s) among the first six jobs are not completed by time $t = 6$, then the adversary releases 12 jobs with processing time one unit and these 12 jobs can only be scheduled on the machines in MI_2. It is clear that the makespan of the schedule produced by algorithm A is strictly greater than 10. However, the optimal schedule has makespan 10. Again, algorithm A cannot be 1-competitive.

3. Offline algorithm

In this section, we give a fast offline algorithm for inclusive processing set when jobs have different release times. Let $r_0 < r_1 < \cdots < r_{k-1}$ be the k distinct release times. Without loss of generality, we may assume that $r_0 = 0$. An upper bound for the optimal makespan is $U = r_{k-1} + \sum_{j=1}^{n} p_j$ and an obvious lower bound is $L = r_{k-1}$. We conduct a binary search in the interval $[L, U]$, searching for the optimal makespan C^*. For each value C obtained in the binary search, we test if there is a feasible schedule with makespan C.

For convenience of presentation, we let $r_k = C$. For each $0 \le i \le k-1$, let TS_i denote the time segment $[r_i, r_{i+1}]$. We first do some preprocessing of the jobs. For any job J_l released at time r_i, if p_l is greater than $r_{i+1} - r_i$, we cut it into two pieces: a job J_l' released at r_i with processing time $r_{i+1} - r_i$, and a job J_l'' released at time r_{i+1} with processing time $p_l - (r_{i+1} - r_i)$. If the processing time of J_l'' is still greater than $r_{i+2} - r_{i+1}$, we again cut it into two pieces, and so on. After the preprocessing step, we may assume that for any job released at time r_i, its processing time is less than or equal to $r_{i+1} - r_i$. We use the array $T(l, i)$, $1 \le l \le n$ and $0 \le i \le k-1$, to store the processing time of job l in the time segment TS_i. Clearly, $T(l, i) = 0$ if the release time of job l is larger than r_i. When we construct the optimal schedule, we will update $T(l, i)$ so that it reflects the amount of processing of job l done in the time segment TS_i in the optimal schedule. Once we get the updated $T(l, i)$, the optimal schedule can be obtained by McNaughton's [4] wraparound rule segment by segment, and within each segment the jobs are scheduled from the outermost machine interval to the innermost one.

We assume that there are z machine intervals in the job set: $MI_1 \supset MI_2 \supset \cdots \supset MI_z$. For each machine interval MI_j, $1 \le j \le z-1$, let Y_j denote the machines in $MI_j - MI_{j+1}$, and let Y_z denote all the machines in MI_z. We use the array $IDLE(j, i)$, $1 \le j \le z$ and $0 \le i \le k-1$, to store the total idle time of the machines in Y_j during the time segment TS_i. With a slight abuse of notation, we say that we schedule jobs in the idle time in $IDLE(j, i)$ to mean that jobs are scheduled on the machines in Y_j during the idle time in the time segment TS_i. From now on, we assume that for every time segment, we have *exactly* z machine intervals. We use the symbol $J(MI_j, i)$ to denote the set of jobs with machine interval MI_j released at time r_i. Clearly, $J(MI_j, i) = \emptyset$ if there is no job with machine interval MI_j released at time r_i.

In the next subsection, we will give a test to see if there is a feasible schedule with makespan C. In the following subsection, we will describe an algorithm to obtain an optimal schedule once the optimal makespan C^* is determined.

3.1. Feasibility test

The algorithm works backwards in time. The time segment $TS_{k-1} = [r_{k-1}, r_k]$ can be easily determined. We schedule the jobs by McNaughton's rule, starting with the jobs in the outermost interval and ending with the jobs in the innermost interval. If some jobs cannot be feasibly scheduled, we declare that it is impossible to have a schedule with makespan C. Otherwise, we compute $IDLE(j, k-1)$ for all $1 \le j \le z$. We let $\beta_j = IDLE(j, k-1)$ for all $1 \le j \le z$.

Suppose we have determined that the jobs in the time segment $[r_{i+1}, r_{i+2}]$ can be feasibly scheduled. We now consider the jobs released at time r_i. We will consider the jobs from the outermost interval to the innermost interval. For each MI_j, $1 \le j \le z$, let $extra_j$ be the total idle time of the machines in Y_j (i.e., the total idle time in the interval $[r_i, r_{i+1}]$ plus the total idle time in the interval $[r_{i+1}, r_k]$, minus the total processing time of the jobs in $J(MI_j, i)$). If $extra_j < 0$, then some jobs in this machine interval have to be scheduled on some other machines of an inner interval.

After we computed $extra_j$, the following procedure (Algorithm *Compute-Idle-and-Test-Feasibility*) can be used to compute the amount of processing times needed to be moved between any pair of Y_j's. The variable $move(p, q)$ is used to store the amount of processing times needed to be moved from the machines in Y_p to the machines in Y_q. We also update $extra_j$ for $1 \le j \le z$. Finally, we let $\beta_j = extra_j$ for $1 \le j \le z$.

```
1  Algorithm:Compute-Idle-and-Test-Feasibility
   Input: r_i: release time; β_1, ..., β_z computed at r_{i+1}.
2  for j = 1 to z do
3  |  extra_j = |Y_j| · (r_i − r_{i+1}) + β_j − (total processing time of the jobs in J(MI_j, i));
4  q = 2;
5  for j = 1 to z do
6  |  while extra_j < 0 do
7  |  |  if q > z then stop and output "Not Feasible";
8  |  |  if extra_q > 0 then
9  |  |  |  move(j, q) = min(extra_q, −extra_j);
10 |  |  |  extra_q = extra_q − move(j, q);
11 |  |  |  extra_j = extra_j + move(j, q);
12 |  |  |  if extra_q = 0 then q = q + 1;
13 |  |  else
14 |  |  |  q = q + 1;
15 |  |
16 |
17 for j = 1 to z do
18 |  β_j = extra_j;
19 output "Feasible";
```

The running time of Algorithm *Compute-Idle-and-Test-Feasibility* for one time segment is $O(m+n_i)$, where n_i is the number of jobs released at r_i. The for-loop in Steps 2 to 3 takes $O(m+n_i)$ time. The for-loop in Steps 5 to 16 takes $O(m)$ time because

in each iteration, we either increase the value of p, or the value of q, and the algorithm will terminate when either p or q reaches z. The for-loop in Steps 17 to 18 takes $O(m)$ time. Thus, the overall running time for testing k time segments is $O((m+n)k)$. Since we may assume that $n > m$, the running time becomes $O(nk)$. To find the optimal makespan, the time needed is $O(nk \log P)$, where $P = \sum_{j=1}^{n} p_j$.

The above algorithm determines if there is enough room to schedule the jobs released at r_i. But there is one issue that needs to be resolved. It is possible that a job released at r_i is the same job as the jobs released at $r_{i+1}, \ldots, r_{k-1}$, due to the preprocessing step. Therefore, the jobs released at r_i must be scheduled carefully so that there is no overlap with the same jobs in subsequent time segments. Fortunately, we can show that the jobs can indeed be feasibly scheduled if the jobs released at r_i pass the test of Algorithm *Compute-Idle-and-Test-Feasibility*. In the next subsection, we will give an algorithm to construct an optimal schedule.

3.2. Obtaining an optimal schedule

As was mentioned in Section 3.1, to obtain an optimal schedule, all we need to do is to compute $T(l, i)$ which stores the amount of processing of job l done in the time segment TS_i in an optimal schedule. Initially, $T(l, i)$ is the processing time of job l in the time segment TS_i. We compute $T(l, i)$ backwards in time, starting with the time segment TS_{k-1}. $T(l, k-1)$ is exactly the initial value. We then schedule the jobs in the time segment TS_{k-1} by McNaughton's rule, starting with the jobs in the outermost interval and ending with the jobs in the innermost interval. From the schedule, we set $IDLE(j, k-1)$, $1 \le j \le z$, to be the total idle time of the machines in Y_j during the time segment TS_{k-1}. We let $\alpha_j = IDLE(j, k-1)$ for $1 \le j \le z$.

Suppose we have computed $T(l, i+1)$ for the jobs in the time segment TS_{i+1}. We now consider the jobs released at time r_i. For each machine interval MI_j, let $\alpha_j = \sum_{x=i+1}^{k-1} IDLE(j, x)$; i.e., α_j is the total idle time of the machines in Y_j after time r_{i+1}. We then invoke Algorithm *Compute-Idle-and-Test-Feasibility* to test feasibility for this time segment. Algorithm *Compute-Idle-and-Test-Feasibility* will compute $extra_j$ and $move(p, q)$, in addition to testing feasibility; see the algorithm in Section 3.1.

After Algorithm *Compute-Idle-and-Test-Feasibility* is called, if $extra_j \ge \alpha_j$, then we do not need to fill any job in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$. The jobs in $J(MI_j, i)$ will be scheduled exclusively on the machines of Y_j in the time segment $[r_i, r_{i+1}]$. On the other hand, if $extra_j < \alpha_j$, then besides scheduling all the machines of Y_j in the time segment $[r_i, r_{i+1}]$, we need to fill in an amount of processing time equal to $fill_j = \alpha_j - extra_j$ in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$. Let $fill_j = \max\{\alpha_j - extra_j, 0\}$ for all $1 \le j \le z$.

If $fill_j = 0$ for each machine interval MI_j, then we will schedule all the jobs released at time r_i in the time segment TS_i. Thus, $T(l, i)$ will be the same as the initial value. We use McNaughton's rule to construct a schedule in this segment. From the schedule, we can update $IDLE(j, i)$ for all $1 \le j \le z$.

If $fill_j > 0$ for some machine interval Y_j, then we need to fill some jobs released at time r_i in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$. It is possible that a job released at r_i is the same job as the jobs released at $r_{i+1}, \ldots, r_{k-1}$, due to the preprocessing step. Therefore, the jobs released at r_i must be scheduled carefully so as to avoid any overlap with the same job in subsequent time segments. In the following we will describe how to schedule these jobs. The basic idea is that we first schedule jobs with total processing time equal to $fill_j$ in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$. After we finish this step for all the machine intervals, we then schedule the remaining jobs in the time segment TS_i by McNaughton's rule. Finally, we update $IDLE(j, i)$ for all $1 \le j \le z$.

Suppose $fill_1 > 0$. Then we schedule the jobs of $J(MI_1, i)$ with total processing time equal to $fill_1$ in the idle time in $IDLE(1, x)$, $i+1 \le x \le k-1$, on the machines of Y_1. We schedule the jobs in the following manner. For any job $l \in J(MI_1, i)$, we try to schedule this job in the idle time in $IDLE(1, i+1)$ if $IDLE(1, i+1) > 0$. We can schedule job l for an amount equal to $\min\{fill_1, IDLE(1, i+1), T(l, i), r_{i+2} - r_{i+1} - T(l, i+1)\}$ in the time segment TS_{i+1}. Then we update $fill_1$, $IDLE(1, i+1)$, $T(l, i)$ and $T(l, i+1)$. If job l is completely scheduled and $IDLE(1, i+1)$ is still greater than 0, we try to schedule another job of $J(MI_1, i)$ in the idle time in $IDLE(1, i+1)$ using the same method. If job l still has some processing time not scheduled, we try to schedule it in $IDLE(1, i+2)$ and so on. We will show later in Lemma 1 that we can always schedule the jobs of $J(MI_1, i)$ with total processing time equal to $fill_1$ in the idle time in $IDLE(1, x)$, $i+1 \le x \le k-1$, without producing any overlap.

Now, consider the next machine interval Y_j with $fill_j > 0$ and $j > 1$. We first find the maximum machine interval index p such that no processing time need to be moved from Y_a to Y_b for any $1 \le a \le p-1$ and $p \le b \le j$; i.e., $move(a, b) = 0$ for all $1 \le a \le p-1$ and $p \le b \le j$. Then, for each pair of indexes c and d such that $p \le c < d-1 \le j-1$ and $move(c, d) > 0$, we add the value $move(c, d)$ to each item $move(c, c+1)$, $move(c+1, c+2)$, ..., and $move(d-1, d)$. Finally, we set $move(c, d)$ to be 0.

Then, from $g = p$ to $j-1$, we try all jobs of $J(MI_g, i)$ to fill in a total amount up to at most $\min\{fill_j, move(g, g+1), move(g+1, g+2), \ldots, move(j-1, j)\}$ in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$, on the machines of Y_j. Each job is filled in the order of $IDLE(j, i+1)$, $IDLE(j, i+2)$, ..., $IDLE(j, k-1)$. Let ρ_g be the total processing time among all jobs of $J(MI_g, i)$ that are filled in. We update the sequence $move(g, g+1)$, $move(g+1, g+2)$, ..., $move(j-1, j)$ by reducing ρ_g from them, and we update the array $T(\cdot, \cdot)$ and $IDLE(j, x)$, $i+1 \le x \le k-1$. We stop when either $fill_j = 0$ or we have tried all jobs of $J(MI_p, i)$, $J(MI_{p+1}, i)$, ..., and $J(MI_{j-1}, i)$. In the latter case, we then try all jobs of $J(MI_j, i)$ to fill in an amount

equal to $fill_j$ in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$, on the machines of Y_j, and update the array $T(\cdot,\cdot)$ and $IDLE(j, x)$. (Again, the jobs are filled in the order of $IDLE(j, i+1)$, $IDLE(j, i+2)$, ..., $IDLE(j, k-1)$.) We will show later in Lemma 1 that we can always schedule jobs of $J(MI_j, i)$ with total processing time $fill_j$ in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$, without producing any overlap.

After we consider all the machine intervals MI_j with $fill_j > 0$ by the above procedure, every machine interval MI_j must have $fill_j = 0$. Then we schedule the remaining jobs $T(l, i)$, $1 \le l \le n$, in the time segment TS_i by McNaughton's rule. From the schedule, we can update $IDLE(j, i)$, $1 \le j \le z$. The process described above is shown in Algorithm *Schedule-Inclusive-Intervals-with-Release-Time*.

Input: $T(l, y)$ and $IDLE(j, y) 1 \le l \le n$, $i+1 \le y \le k-1$ and $1 \le j \le z$
Output: $T(l, i)$: the amount of processing of job J_l done in the time segment TS_i

$IDLE(j, i) = |Y_j| \cdot (r_{i+1} - r_i)$ for $1 \le j \le z$;
$\alpha_j = \sum_{x=i+1}^{k-1} IDLE(j, x)$ for $1 \le j \le z$;
Call Algorithm *Compute-Idle-and-Test-Feasibility* to compute $move(\cdot,\cdot)$ and $extra_j$ for all $1 \le j \le z$;
For each $1 \le j \le z$, $fill_j = \max\{\alpha_j - extra_j, 0\}$;
Let $J(MI_j, i)$ be the set of jobs with machine interval MI_j released at r_i, $1 \le j \le z$;
$j = 1$;
repeat
  if $fill_j > 0$ **then**
    Let p be the maximum index such that $move(a, b) = 0$ for all $1 \le a \le p-1$ and $p \le b \le j$;
    foreach *pair of c and d such that* $p \le c < d-1 < j$ *and* $move(c, d) > 0$ **do**
      for $x = c$ **to** $d-1$ **do**
        $move(x, x+1) = move(x, x+1) + move(c, d)$
      $move(c, d) = 0$;
    $g = p$;
    while $fill_j > 0$ *and* $g < j$ **do**
      For each job l of $J(MI_g, i)$, fill in an amount up to $\min\{move(g, g+1), \ldots, move(j-1, j), fill_j\}$ in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$.
      Update $fill_j$, $T(\cdot,\cdot)$, and $IDLE(j, x)$, $i+1 \le x \le k-1$;
      Let ρ be the total processing time filled in;
      for $x = g$ **to** $j-1$ **do** $move(x, x+1) = move(x, x+1) - \rho$;
      $g = g+1$;
    if $fill_j > 0$ **then**
      For each job l of $J(MI_j, i)$, fill in an amount up to $fill_j$ in the idle time in $IDLE(j, x)$ for $i+1 \le x \le k-1$.
      Update $fill_j$, $T(l, i)$, $T(l, x)$ and $IDLE(j, x)$;
  $j = j+1$;
until $j > z$;
Schedule the remaining jobs of $J(MI_j, i)$ in the time segment TS_i by McNaughton's rule and update $IDLE(j, i)$ for $1 \le j \le z$.

Algorithm 1: Schedule-Inclusive-Intervals-with-Release-Time

We repeat the procedure for all time segments. After we finish all time segments, we get $T(l, i)$ for any job l and any $0 \le i \le k-1$. Then we can obtain a feasible schedule by McNaughton's rule, segment by segment.

Lemma 1. *Suppose the jobs released at time* r_i *passed the test of Algorithm* Compute-Idle-and-Test-Feasibility. *For each machine interval* MI_j, *we can fill in exactly* $fill_j$ *amount of processing time in the idle time in* $IDLE(j, x)$ *for* $i+1 \le x \le k-1$ *by Algorithm* Schedule-Inclusive-Intervals-with-Release-Times *without any overlap.*

Proof. First, it is easy to see that no overlap can be produced by the above procedure and for each machine interval MI_j, no more than $fill_j$ amount of processing time is filled in the idle time in $IDLE(j, x)$ for $i+1 \le x \le k-1$. We now prove that for each machine interval MI_j, we can always fill in at least $fill_j$ amount of processing time in the idle time in $IDLE(j, x)$ for $i+1 \le x \le k-1$ without producing any overlap.

For any set JS of jobs, let $\rho(JS)$ denote the total processing time of all the jobs in JS. We will prove, by induction on j, the following claim.

Claim 1. *For any machine interval* MI_j, $1 \le j \le z$, *with* $extra_j \le \alpha_j$, *after Algorithm* Compute-Idle-and-Test-Feasibility *is called, the following equation always holds:*

$$\rho(J(MI_j, i)) + \sum_{x=1}^{j-1} move(x, j) - \sum_{x=j+1}^{z} move(j, x) = |Y_j| * (r_{i+1} - r_i) + fill_j.$$

Recall that α_j is the total idle time in $IDLE(j, x)$ for $i+1 \le x \le k-1$ before Algorithm *Compute-Idle-and-Test-Feasibility* is called in this iteration, and $extra_j$ is the total idle time in $IDLE(j, x)$ for $i \le x \le k-1$ after Algorithm *Compute-Idle-and-Test-Feasibility* is called in this iteration. So the claim means that after Algorithm *Compute-Idle-and-Test-Feasibility* is called, for any machine interval MI_j in which at least $|Y_j| * (r_{i+1} - r_i)$ of processing time can be filled, the total processing time of $J(MI_j, i)$ plus the total processing time that needs to be moved into Y_j minus the total processing time that needs to be moved out of Y_j is exactly equal to the total idle time in Y_j in the time segment TS_i plus $fill_j$.

We will prove the claim by induction on j. For the base case $j = 1$, if $fill_1 = 0$, the claim clearly holds. On the other hand, if $fill_1 > 0$, we must have

$$\rho(J(MI_1, i)) - \sum_{x=2}^{z} move(1, x) = |Y_1| * (r_{i+1} - r_i) + fill_1.$$

We now show that we can always schedule the jobs of $J(MI_1, i)$ with total processing time $fill_1$ in the idle time in $IDLE(1, x)$ for $i+1 \le x \le k-1$ on the machines of Y_1. Suppose not. Then there must be a time segment b, $i+1 \le b \le k-1$, such that some idle time of $IDLE(1, b)$ cannot be filled. Since $\rho(J(MI_1, i)) \ge |Y_1| * (r_{i+1} - r_i)$, we must have at least $|Y_1|$ jobs in $J(MI_1, i)$. Since all these jobs cannot be filled in $IDLE(1, b)$, they must have been scheduled in the time segment TS_b for a duration equal to $r_{b+1} - r_b$. But this means that there is no idle time on the machines of Y_1 in the time segment TS_b, contradicting the fact that $IDLE(1, b) > 0$.

After $fill_1$ amount of processing time is filled in the idle time in $IDLE(1, x)$, $i+1 \le x \le k-1$, $J(MI_1, i)$ is updated by removing the scheduled jobs or parts of the jobs, and $fill_1$ is updated to 0. So $\rho(J(MI_1, i)) - \sum_{x=2}^{z} move(1, x) = |Y_1| * (r_{i+1} - r_i) + fill_1$ still holds.

Assume that the claim holds for the first $j-1$ machine intervals. We want to prove that it also holds for the jth interval. If $fill_j = 0$, the claim obviously holds. If $fill_j > 0$, then we must have $extra_j < \alpha_j$. So $\rho(J(MI_j, i)) + \sum_{x=1}^{j-1} move(x, j) - \sum_{x=j+1}^{z} move(j, x) = |Y_j| * (r_{i+1} - r_i) + fill_j$. Let p be the maximum index such that no processing time need to be moved from Y_a to Y_b for any $1 \le a \le p-1$ and $p \le b \le j$. Then for each pair of indexes c and d, $p \le c < d \le j$, such that $move(c, d) > 0$, we must have for all $c \le u \le d-1$, $extra_u = 0$ and $\rho(J(MI_u, i)) + \sum_{x=1}^{u-1} move(x, u) - \sum_{x=u+1}^{z} move(u, x) = |Y_u| * (r_{i+1} - r_i) + fill_u$. So, after we consider all such pairs of c and d, $p \le c < d \le j$, such that $move(c, d) > 0$, we must have for all $p \le u \le j-1$, the following equation holds: $\rho(J(MI_u, i)) + \sum_{x=1}^{u-1} move(x, u) - \sum_{x=u+1}^{z} move(u, x) = |Y_u| * (r_{i+1} - r_i) + fill_u$.

For each pair of indexes c and d, $p \le c < d-1 \le u-1$, such that $move(c, d) > 0$, we add $move(c, d)$ to each of the items $move(c, c+1), move(c+1, c+2), \ldots, move(d-1, d)$, and set $move(c, d) = 0$. So, after we consider all such pairs, we must have for all $p \le u < j$, $move(u, u+1) > 0$ and $\rho(J(MI_u, i)) + move(u-1, u) - \sum_{x=u+1}^{z} move(u, x) = |Y_u| * (r_{i+1} - r_i) + fill_u$.

In our algorithm, from $g = p$ to $j-1$, we try all jobs of $J(MI_g, i)$ to schedule up to $\min\{move(g, g+1), move(g+1, g+2), \ldots, move(j-1, j), fill_j\}$ amount of processing time in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$, on the machines of Y_j. Let ρ be the total processing time that is filled. We update the sequence $move(g, g+1), move(g+1, g+2), \ldots, move(j-1, j)$ by reducing ρ from each of them. Furthermore, $fill_j$ is also updated in the course of scheduling the jobs. Finally, we update $J(MI_g, i)$ by removing all the scheduled jobs or parts of the jobs. Since both $move(j-1, j)$ and $fill_j$ are reduced by ρ and other items are not changed, the following equation still holds: $\rho(J(MI_j, i)) + move(j-1, j) - \sum_{x=j+1}^{z} move(j, x) = |Y_j| * (r_{i+1} - r_i) + fill_j$. Similarly, since both $move(g, g+1)$ and $\rho(J(MI_g, i))$ are reduced by ρ and other items are not changed, the following equation still holds: $\rho(J(MI_g, i)) + move(g-1, g) - \sum_{y=x+1}^{z} move(g, y) = |Y_g| * (r_{i+1} - r_i) + fill_g$. Notice that $fill_g = 0$ now. There is no change in other machine intervals, so all the equations still hold.

We stop when either $fill_j = 0$ or we have tried all jobs of $J(MI_p, i), J(MI_{p+1}, i), \ldots$, and $J(MI_{j-1}, i)$. If, after we have tried all jobs of $J(MI_p, i), J(MI_{p+1}, i), \ldots$, and $J(MI_{j-1}, i)$, $fill_j$ is still greater than 0, we then try all jobs of $J(MI_j, i)$ to schedule $fill_j$ amount of processing time in the idle time in $IDLE(j, x)$ for $i+1 \le x \le k-1$ on the machines of Y_j, and we update $J(MI_j, i)$ by removing all the scheduled jobs or parts of the jobs. Since both $\rho(J(MI_j, i))$ and $fill_j$ are reduced by the same amount and the other items are not changed, the following equation still holds: $\rho(J(MI_j, i)) + move(j-1, j) - \sum_{x=j+1}^{z} move(j, x) = |Y_j| * (r_{i+1} - r_i) + fill_j$.

We now show that after we tried all the jobs of $J(MI_j, i)$, $fill_j$ must be 0. Suppose not. Let w be the last machine index such that $move(w, w+1) = 0$. That is, from $w+1$ on, each time we try to fill $\min\{fill_j, move(x, x+1), \ldots, move(j-1, j)\}$, $x > w$, of processing time in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$, on the machines of Y_j, we were able to fill only $\delta < \min\{fill_j, move(x, x+1), \ldots, move(j-1, j)\}$ of processing time in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$, on the machines of Y_j. That means we have tried all the jobs in $J(MI_x, i)$, $w+1 \le x \le j$, to fill in the idle time in $IDLE(j, x)$, $i+1 \le x \le k-1$. Since we still have $fill_j > 0$, there must be a time segment b, $i+1 \le b \le k-1$, such that some idle time in $IDLE(j, b)$ cannot be filled.

Since for each $w+1 \le u \le j$, we have

$$\rho(J(MI_u, i)) + move(u-1, u) - \sum_{x=u+1}^{z} move(u, x) = |Y_u| * (r_{i+1} - r_i) + fill_u$$

and

$$move(w, w+1) = 0,$$

we must have

$$\rho(J(MI_{w+1}, i)) - \sum_{x=w+2}^{z} move(w+1, x) = |Y_{w+1}| * (r_{i+1} - r_i) + fill_{w+1}$$

and

$$\begin{aligned}
&\rho(J(MI_{w+1}, i)) + \rho(J(MI_{w+2}, i)) + \cdots + \rho(J(MI_j, i)) \\
&\quad + move(w+1, w+2) + move(w+2, w+3) + \cdots + move(j-1, j) \\
&\quad - \left(\sum_{x=w+2}^{z} move(w+1, x) + \sum_{x=w+3}^{z} move(w+2, x) + \cdots + \sum_{x=j+1}^{z} move(j, x) \right) \\
&\geq (|Y_{w+1}| + |Y_{w+2}| + \cdots + |Y_j|) * (r_{i+1} - r_i) + fill_{w+1} + \cdots + fill_j.
\end{aligned}$$

Notice that $fill_u = 0$ for all $w+1 \leq u \leq j-1$. Thus, we have $\rho(J(MI_{w+1}, i)) + \rho(J(MI_{w+2}, i)) + \cdots + \rho(J(MI_j, i)) \geq (|Y_{w+1}| + |Y_{w+2}| + \cdots + |Y_j|) * (r_{i+1} - r_i) + fill_j$.

Since each of these jobs has length at most $(r_{i+1} - r_i)$, there must be at least $|Y_{w+1}| + |Y_{w+2}| + \cdots + |Y_j|$ jobs in $J(MI_{w+1}, i)$, $J(MI_{w+2}, i)$, ..., and $J(MI_j, i)$. Since all these jobs cannot be filled in the idle time in $IDLE(j, b)$, these jobs must have been scheduled in the time segment TS_b for a duration equal to $(r_{b+1} - r_b)$. Therefore, there should be no idle time in time segment TS_b on the machines of Y_j, contradicting the fact that $IDLE(j, b) > 0$.

So, for machine interval MI_j, the claim still holds and we can fill exactly $fill_j$ of processing time in the idle time in $IDLE(j, x)$, $i+1 \leq x \leq k-1$, on the machines of Y_j. □

Algorithm 1 constructs the schedule for one time segment. Steps 1 to 6 take $O(n+m)$ time. Step 9 takes $O(m)$ time for each machine interval MI_j. Since there are at most m machine intervals, Step 9 takes a total of $O(m^2)$ time. Steps 10 to 13 take $O(m^3)$ time since there are $O(m^2)$ pairs of c and d. Steps 15 to 22 take $O(mnk)$ time. Therefore, the total time needed for one time segment is $O(mnk + m^3)$. Hence the total time for k time segment is $O(mnk^2 + m^3k)$. After we update the array $T(\cdot, \cdot)$, we can obtain a schedule by McNaughton's rule which takes $O(nk)$ time. Thus, we have the following theorem.

Theorem 1. *For the inclusive processing set restrictions, we can obtain an optimal schedule in time* $O(nk \log P + mnk^2 + m^3k)$.

4. Conclusion

In this paper we show that there is no 1-competitive online algorithm for the preemptive scheduling problem with different release times and inclusive processing set restrictions. For future research, it will be interesting to develop an online algorithm with a small competitive ratio. For the offline case, we give a fast algorithm for the inclusive processing set restriction. For future research, it will be interesting to see if there are even more efficient algorithms than the one presented in this paper.

Acknowledgements

The first author's work was supported in part by the PSC-CUNY research fund and second author's work was supported in part by the NSF grant DMI-0556010.

References

[1] H.-C. Hwang, S.Y. Chang, K. Lee, Parallel machine scheduling under a grade of service provisions, Computers & Operations Research 31 (2004) 2055–2061.
[2] C.A. Glass, H. Kellerer, Parallel machine scheduling with job assignment restrictions, Naval Research Logistics 54 (2007) 250–257.
[3] J. Ou, J.Y.-T. Leung, C.-L. Li, Scheduling parallel machines with inclusive processing set restrictions, Naval Research Logistics 55 (2008) 328–338.
[4] R. McNaughton, Scheduling with deadlines and loss functions, Management Science 6 (1959) 1–12.
[5] K.S. Hong, J.Y.-T. Leung, On-line scheduling of real-time tasks, IEEE Transactions on Computers 41 (1992) 1326–1331.
[6] E.L. Lawler, J. Labetoulle, On preemptive scheduling of unrelated parallel processors by linear programming, Journal of ACM 25 (1978) 612–619.
[7] P. Brucker, B. Jurisch, A. Kramer, Complexity of scheduling problems with multi-purpose machines, Annals of Operations Research 70 (1997) 57–73.
[8] A.V. Goldberg, R.E. Tarjan, A new approach to the maximum flow problem, Journal of ACM 35 (1988) 921–940.
[9] C.-L. Li, X. Wang, Scheduling parallel machines with inclusive processing set restrictions and job release times, Working Paper, Department of Logistics, The Hong Kong Polytechnic University, Kowloon, Hong Kong.