A Note on Scheduling Jobs with Equal Processing Times and Inclusive Processing Set Restrictions

Abstract

We consider the problem of scheduling n jobs on m parallel machines with inclusive processing set restrictions. Each job has a given release date, and all jobs have equal processing times. The objective is to minimize the makespan of the schedule. Li and Li (2015) have developed an $O(n^2 + mn \log n)$ time algorithm for this problem. In this note, we present a modified algorithm with an improved time complexity of $O(\min\{m, \log n\} \cdot n \log n)$.

Keywords: Scheduling; parallel machines; equal processing time jobs; inclusive processing sets

1 Introduction

Consider the following scheduling problem studied by Li and Li (2015): Given n jobs $J_1, J_2, \ldots, J_n$ and m parallel machines $M_1, M_2, \ldots, M_m$, we would like to schedule the jobs on the machines so as to minimize the makespan of the schedule. All jobs have the same processing time $p > 0$, and job preemption is not allowed. Associated with each job J_j is a release date $r_j \geq 0$ and a grade $a_j \in \{1, 2, \ldots, m\}$. Job J_j can be processed by any one of the machines $M_{a_j}, M_{a_j+1}, \ldots, M_m$ but not by any of the machines $M_1, M_2, \ldots, M_{a_j-1}$.

This problem is denoted as $P|\mathcal{M}_j(\text{inclusive}), r_j, p_j = p|C_{\max}$, and $\mathcal{M}_j = \{M_{a_j}, M_{a_j+1}, \ldots, M_m\}$ is called the *processing set* of J_j. The processing sets are referred to as "inclusive" because for every pair $\mathcal{M}_j$ and $\mathcal{M}_k$, either $\mathcal{M}_j \subseteq \mathcal{M}_k$ or $\mathcal{M}_k \subseteq \mathcal{M}_j$ (see Leung and Li 2008). Li and Li (2015) have developed a polynomial time algorithm for this problem. In this note, we provide a modified algorithm with an improved time complexity. Without loss of generality, we assume that $m \leq n$ (if $m > n$, then machines $M_1, M_2, \ldots, M_{m-n}$ are not needed and can be ignored).

As stated in Li and Li (2015), the set

$$\Lambda = \big\{T \;\big|\; T = r_j + kp;\; j, k \in \{1, 2, \ldots, n\}\big\},$$

contains all candidates for the optimal makespan of $P|\mathcal{M}_j(\text{inclusive}), r_j, p_j = p|C_{\max}$. Note that $|\Lambda| = O(n^2)$. Li and Li (2015) have shown that the elements of Λ can be generated and sorted in an ascending order in $O(n^2)$ time. Once the elements of Λ are sorted, it takes $O(\log|\Lambda|) = O(\log n)$ iterations of a binary search to determine the optimal makespan T^*. In each iteration, there is a target makespan $T \in \Lambda$, and we would like to either assign all n jobs to the machines while maintaining a makespan no greater than T, or determine that it is impossible to do so. They have developed an $O(mn)$ time procedure to perform such a job assignment for any given T. Thus, the overall time complexity of their algorithm is $O(n^2 + mn \log n)$.

In our improved algorithm, we use two binary searches, but each binary search involves a smaller set of candidates. In addition, we provide an alternative implementation of Li and Li's (2015) job assignment procedure. This alternative implementation has a running time of $O(n \log n)$

instead of $O(mn)$. Combining these two techniques, we obtain an overall time complexity of $O(\min\{m, \log n\} \cdot n \log n)$ for problem $P|\mathcal{M}_j(\text{inclusive}), r_j, p_j = p|C_{\max}$.

2 The Job Assignment Procedure

We first consider the following problem: Given any target makespan $T \in \Lambda$, we would like to either assign all n jobs to the machines so that all jobs finish processing no later than T, or determine that it is impossible to do so. Li and Li (2015) have presented a procedure for solving this problem. The procedure attempts to assign jobs to disjoint time slots of size p in the form of $[T-kp, T-(k-1)p]$, where $k = 1, 2, \ldots, \min\{n, \lfloor T/p \rfloor\}$. A job J_j can be assigned to a time slot $[T-kp, T-(k-1)p]$ on machine M_i only if $r_j \le T - kp$ and $i \ge a_j$. The details of Li and Li's (2015) procedure are given as follows.

Procedure $\mathcal{P}(T)$:

```
Step 1. Let $\Phi_0 \leftarrow \emptyset$.
Step 2. For $i = 1, 2, \ldots, m$,
    (a) let $\Phi_i \leftarrow \Phi_{i-1} \cup \{J_j \mid a_j = i\}$;
    (b) let $S \leftarrow T - p$;
    (c) let $\Psi \leftarrow \{j \mid J_j \in \Phi_i \text{ and } r_j \le S\}$;
    (d) if $\Psi \ne \emptyset$ then let $\ell \leftarrow \arg\max_{j\in\Psi}\{r_j\}$, schedule $J_\ell$ to start at time $S$ on machine $M_i$, let
        $\Phi_i \leftarrow \Phi_i \setminus \{J_\ell\}$, let $S \leftarrow S - p$, and go to Step 2(c); otherwise, move on to the next $i$.
```

In this procedure, S keeps track of the start time of the slot, Φ_i contains the unassigned jobs that are eligible for machine M_i, and Ψ contains the indices of those unassigned jobs that are feasible for the time interval $[S, S+p]$ on the current machine. In Li and Li (2015), the elements of Φ_i, Ψ, and $\{J_j \mid a_j = i\}$ are maintained in a descending order of r_j, and Step 2(a) of procedure $\mathcal{P}(T)$ is a merging operation of two sorted arrays. Thus, each iteration of Step 2 requires $O(n)$ time, and the time complexity of the procedure is $O(mn)$.

We now present an alternative implementation of procedure $\mathcal{P}(T)$. We use a balanced binary tree to store the elements of Φ_i using r_j as the key, so that a job can be inserted to and deleted from Φ_i in $O(\log n)$ time (see Knuth 1998, Sec. 6.2.3 for a discussion of balanced trees). Let $h_i = |\{J_j \mid a_j = i\}|$, which is the number of jobs inserted into Φ_{i-1} in Step 2(a). Then, each execution of Step 2(a) can be done in $O(h_i \log n)$ time. The total running time of Step 2(a) in all m iterations combined is $O(\sum_{i=1}^{m} h_i \log n) = O(n \log n)$. In Steps 2(c)–(d), instead of creating set Ψ, we search for the job J_ℓ; that is, among the jobs of Φ_i stored in the balanced binary tree, we search for the job with the largest r_j such that $r_j \leq S$. This can be accomplished in $O(\log n)$ time. In Step 2(d), removing J_ℓ from Φ_i requires $O(\log n)$ time. Thus, each execution of Steps 2(c)–(d) requires $O(\log n)$ time. Note that Steps 2(c)–(d) are executed $m+n$ times in the procedure. Hence, the total running time of Steps 2(c)–(d) in all the iterations combined is $O(n \log n)$. Therefore, the complexity of this alternative implementation of procedure $\mathcal{P}(T)$ is $O(n \log n)$. This leads to the following lemma.

Lemma 1 *Procedure $\mathcal{P}(T)$ can be implemented in $O(\min\{m, \log n\} \cdot n)$ time.*

Proof. If $m \leq O(\log n)$, then we use Li and Li's (2015) original implementation, which has a running time of $O(mn)$. Otherwise, we use our alternative implementation, which has a running time of $O(n \log n)$. Thus, procedure $\mathcal{P}(T)$ can be implemented in $O(\min\{m, \log n\} \cdot n)$ time. ∎

3 Searching for the Optimal Makespan

We now present an improved method for determining the optimal makespan T^*. Suppose that we know an upper bound T_U on the optimal makespan such that

$$T_U - p < T^* \leq T_U.$$

Then, it suffices to consider a subset of candidates $\Lambda' \subseteq \Lambda$ for the optimal makespan, where

$$\begin{aligned}\Lambda' &= \{T \mid T = r_j + kp \text{ and } T_U - p < T \leq T_U;\ j,k \in \{1,2,\ldots,n\}\} \\ &= \left\{T \;\middle|\; T = r_j + \left\lfloor \frac{T_U - r_j}{p} \right\rfloor p;\ j \in \{1,2,\ldots,n\}\right\}.\end{aligned}$$

After constructing set Λ', we sort its elements in an ascending order and then use a binary search to determine the optimal makespan T^*. In each iteration of the binary search, there is a target makespan T, and we execute procedure $\mathcal{P}(T)$ to determine if there exists a feasible solution with a makespan no greater than T.

To determine T_U, we consider the range of the optimal makespan. First, $T^* \geq r_{\max} + p$ because the completion time of the job with the largest release date is at least $r_{\max} + p$ in any feasible schedule, where $r_{\max} = \max\{r_1, r_2, \ldots, r_n\}$. Second, $T^* \leq r_{\max} + np$ because the schedule where all n jobs are scheduled on machine M_m after time $r_{\max}$ with no idle time between jobs is feasible. Thus, there must exist $k \in \{1, 2, \ldots, n\}$ such that

$$r_{\max} + (k-1)p < T^* \leq r_{\max} + kp.$$

Hence, it suffices to consider a subset of candidates $\Lambda'' \subseteq \Lambda$ for T_U, where

$$\Lambda'' = \big\{T \;\big|\; T = r_{\max} + kp;\; k \in \{1, 2, \ldots, n\}\big\}.$$

After constructing set Λ'', we use a binary search to determine $T_U \in \Lambda''$ such that $\mathcal{P}(T_U)$ returns a feasible schedule but $\mathcal{P}(T_U - p)$ returns an infeasible solution. So, $T_U - p < T^* \leq T_U$. In each iteration of the binary search, there is a target makespan T, and we execute procedure $\mathcal{P}(T)$ to determine if there exists a feasible solution with a makespan no greater than T.

The following theorem states the main result of this note.

Theorem 2 *Problem $P \mid \mathcal{M}_j(inclusive), r_j, p_j = p \mid C_{\max}$ is solvable in $O(\min\{m, \log n\} \cdot n \log n)$ time.*

Proof. Note that $|\Lambda''| = n$. Constructing set Λ'' with its elements arranged in an ascending order can be achieved in $O(n)$ time. The binary search on set Λ'' has $O(\log n)$ iterations. By Lemma 1, each iteration requires $O(\min\{m, \log n\} \cdot n)$ time. Thus, T_U can be determined in $O(\min\{m, \log n\} \cdot n \log n)$ time.

Once T_U is determined, we construct set Λ' for the main problem. Note that $|\Lambda'| \leq n$, and sorting the elements of Λ' requires $O(n \log n)$ time. The binary search on set Λ' has $O(\log n)$

iterations. By Lemma 1, each iteration requires $O(\min\{m, \log n\} \cdot n)$ time. Hence, T^* can be determined in $O(\min\{m, \log n\} \cdot n \log n)$ time. ∎

Table 1 provides a detailed breakdown of the running time of each component of the algorithm.

Table 1: Comparison of the two algorithms.

	Li and Li's (2015) algorithm	Our algorithm	
		Determining T_U	Main problem
Constructing/sorting set Λ, Λ', or Λ''	$O(n^2)$	$O(n)$	$O(n \log n)$
Binary search	$O(\log n)$	$O(\log n)$	$O(\log n)$
Procedure $\mathcal{P}(T)$	$O(mn)$	$O(\min\{m, \log n\} \cdot n)$	$O(\min\{m, \log n\} \cdot n)$
Overall complexity	$O(n^2 + mn \log n)$	$O(\min\{m, \log n\} \cdot n \log n)$	

4 Computational Experiments

To compare the performance of our algorithm with Li and Li's (2015) algorithm, a set of computational experiments is conducted. In these experiments, we use randomly generated problems, and the data of each test instance are generated in the same way as in Li and Li (2015). Specifically, we set $p = 1$. For each job j, a_j is randomly generated according to a discrete uniform distribution within $\{1, 2, \ldots, m\}$, and r_j is randomly generated according to a continuous uniform distribution within the interval $[0, \lambda]$, where λ is a parameter which measures the diversity of job release dates. We run the experiments with $n = 300, 900, 2700, 8100$, $m = 2, 6, 18, 54$, and $\lambda = n/2m, n/m, 2n/m$. Thus, there are $4 \times 4 \times 3 = 48$ different parameter settings. For each combination of n, m, and λ, we generate 10 test instances.

From the proof of Lemma 1, procedure $\mathcal{P}(T)$ can be implemented more efficiently if we use Li and Li's (2015) original implementation when $m \leq O(\log n)$ and use our alternative implementation when $m > O(\log n)$. However, this saving in computational time is much less significant than the saving obtained by the improved method for searching the optimal makespan T^*. Hence, for simplicity, in this computational study we use Li and Li's (2015) original implementation of procedure $\mathcal{P}(T)$ only. We coded our algorithm and Li and Li's (2015) algorithm in Excel VBA and ran the experiments on a PC with a 2.83 GHz processor and 4 GB RAM. Table 2 summarizes the computational results, where the average computational time of the 10 test instances in each parameter setting is reported.

Table 2: Average computational time of the 10 test instances (in seconds).

			$n = 300$	$n = 900$	$n = 2700$	$n = 8100$
$m = 2$	$\lambda = n/2m$	Our algorithm	< 0.01	< 0.01	0.02	0.08
		Li and Li's (2015) algorithm	0.03	0.23	1.97	16.13
	$\lambda = n/m$	Our algorithm	< 0.01	< 0.01	0.03	0.08
		Li and Li's (2015) algorithm	0.03	0.25	2.11	17.34
	$\lambda = 2n/m$	Our algorithm	< 0.01	< 0.01	0.02	0.08
		Li and Li's (2015) algorithm	0.03	0.28	2.39	19.92
$m = 6$	$\lambda = n/2m$	Our algorithm	< 0.01	0.01	0.04	0.11
		Li and Li's (2015) algorithm	0.03	0.22	1.89	15.42
	$\lambda = n/m$	Our algorithm	< 0.01	0.01	0.04	0.12
		Li and Li's (2015) algorithm	0.03	0.23	1.94	15.80
	$\lambda = 2n/m$	Our algorithm	< 0.01	< 0.01	0.03	0.11
		Li and Li's (2015) algorithm	0.03	0.24	2.03	16.57
$m = 18$	$\lambda = n/2m$	Our algorithm	< 0.01	0.02	0.06	0.20
		Li and Li's (2015) algorithm	0.03	0.23	1.89	15.26
	$\lambda = n/m$	Our algorithm	< 0.01	0.02	0.06	0.20
		Li and Li's (2015) algorithm	0.03	0.23	1.90	15.42
	$\lambda = 2n/m$	Our algorithm	< 0.01	0.02	0.06	0.19
		Li and Li's (2015) algorithm	0.03	0.23	1.93	15.63
$m = 54$	$\lambda = n/2m$	Our algorithm	0.01	0.04	0.14	0.45
		Li and Li's (2015) algorithm	0.03	0.25	1.94	15.37
	$\lambda = n/m$	Our algorithm	0.01	0.04	0.13	0.44
		Li and Li's (2015) algorithm	0.04	0.25	1.95	15.46
	$\lambda = 2n/m$	Our algorithm	0.01	0.04	0.12	0.41
		Li and Li's (2015) algorithm	0.04	0.25	1.97	15.56

From Table 2, we observe that parameter λ has little impact on the running time of our algorithm. Parameter m has a small impact, while parameter n has a relatively large impact. As n increases, the running time of our algorithm increases but its rate is much lower than that of Li and Li's (2015) algorithm. The reason is that Li and Li's (2015) algorithm needs to construct and sort the elements of an array Λ of $O(n^2)$ size, whereas both arrays Λ' and Λ'' in our algorithm are of $O(n)$ size. Overall, our algorithm is highly efficient and could solve the largest test instance in less than 0.5 seconds.

5 Concluding Remarks

The technique presented in Section 3 can be applied to problems with a more general setting. Li and Li (2015) have shown that their algorithm can be extended to solve the more general problem with tree-hierarchical processing sets (i.e., problem $P \mid \mathcal{M}_j(tree), r_j, p_j = p \mid C_{\max}$), and their extended algorithm has an $O(n^2 + mn \log n)$ running time. It is easy to check that sets Λ' and Λ'', as well as the binary search routines, can be applied to problem $P \mid \mathcal{M}_j(tree), r_j, p_j = p \mid C_{\max}$. The execution of procedure $\mathcal{P}(T)$ requires $O(mn)$ time (note: the alternative implementation presented in Section 2 does not apply to tree-hierarchical processing sets). Hence, the running time of Li and Li's (2015) extended algorithm can be reduced to $O(mn \log n)$.

We would like to make a final remark that the technique presented in Section 2 (i.e., using a balanced tree for merging job subsets iteratively) can be applied to other scheduling problems to attain lower computational complexities. Ou *et al.* (2015) have used such a technique for solving a single-machine scheduling problem with job rejection.

Acknowledgments

The first author was supported in part by The Hong Kong Polytechnic University under grant 1-BBZL. The second author was supported in part by PSC CUNY (The City University of New York) Grant TRADA-46-477.

References

Knuth, D.E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd edition, Addison-Wesley, Reading, MA.

Leung, J.Y.-T. and Li, C.-L. (2008). Scheduling with processing set restrictions: A survey. *International Journal of Production Economics* **116**(2), 251–262.

Li, C.-L. and Li, Q. (2015). Scheduling jobs with release dates, equal processing times, and inclusive processing set restrictions. *Journal of the Operational Research Society* **66**(3), 516–523.

Ou, J., Zhong, X. and Li, C.-L. (2015). Improved algorithms for single machine scheduling with release dates and rejection. Submitted for publication.