Preemptive scheduling of jobs with tied parameters on a single processor to minimize the number of late jobs

NODARI VAKHANIA
State University Morelos, Science Faculty
Av. Universidad 1001 62210 Cuernavaca, Morelos, Mexico.
Institute of Computational Mathematics, Georgian Academy of Sciences, Tbilisi, Georgia
nodari@uaem.mx

Abstract: We have \( n \) jobs with release times and due-dates to be scheduled on a single-machine that can handle at most one job at a time. Our objective is to minimize the number of late jobs, ones completed after their due-dates. This problem is known to be solvable in time \( O(n^4) \). We show that the special case of the problem when job processing times and due-dates are tied is solved by an on-line algorithm. Both preemptive and non-preemptive versions can be solved relatively easily in time \( O(n \log n) \) if either the release times or due-dates of all jobs are equal (Moore [4]).

In this paper we deal with another special case of the problem \( 1/pmttn, r_j/\sum U_j \) when job processing times and due-dates are tied in the following way. For each pair of jobs \( i, j \) with \( d_i > d_j, p_i \geq p_j \), can be solved in time \( O(n^2) \). See also Lawler for an off-line dynamic programming algorithm with the same time complexity. This algorithm solves a more general problem with job weights, a special case of \( 1/pmttn, r_j/\sum w_j U_j \) in which the jobs can be ordered so that \( r_1 \leq r_2 \leq \ldots \leq r_n, p_1 \leq p_2 \leq \ldots \leq p_n \) and \( w_1 \geq w_2 \geq \ldots \geq w_n \). Both preemptive and non-preemptive versions can be solved relatively easily in time \( O(n \log n) \) if either the release times or due-dates of all jobs are equal (Moore [4]).

In this section we present the algorithm and its correctness proof. First, we give the following general observation. Due to the nature of our objective function, if a job is late then it can obviously be scheduled arbitrarily late without affecting our objective function. Suppose \( S \) is a feasible schedule with all its

1 Introduction

Jobs from \( J = \{1, 2, \ldots, n\} \) have to be assigned to or scheduled on a single machine when each \( j \in J \) becomes available at an integer release time \( r_j \) and has an integer due date \( d_j \) which is the desired time to complete \( j \). Job \( j \) needs an integer processing time \( p_j \) on the machine. A schedule assigns each job \( j \) time intervals with the total length of \( p_j \) starting no earlier than at time \( r_j \) so that there is no intersection between intervals of different jobs, i.e., the machine may handle at most one job at a time. In this way we allow job preemptions splitting jobs into parts. A job is late (on time, respectively) if it (its latest scheduled part) is completed after (at or before, respectively) its due date. Our objective is to minimize the number of late jobs. Due to this objective function, we may assume that every job may potentially be completed by its due date, i.e., \( r_j + p \leq d_j \), for each \( j \) (then we say that job release times and due dates are agreeable).

The general problem described above is commonly abbreviated as \( 1/pmttn, r_j/\sum U_j \) (\( U_j \) is a 0-1 function taking value 1 iff job \( j \) is late). It is known to be solvable in polynomial time. In particular, Lawler [3] and Baptiste [4] have suggested dynamic programming algorithms with the time complexity of \( O(n^3) \) and \( O(n^4) \), respectively. Recently Vakhania [5] has improved the time complexity to \( O(n^3 \log n) \). The non-preemptive version of the above problem \( 1/r_j/\sum U_j \) is known to be strongly NP-hard. The version with equal-length jobs \( 1/pmttn, p_j = p, r_j/\sum U_j \) can be solved on-line in time \( O(n \log n) \) Vakhania [5] and [6]. See also Lawler [3] for an off-line dynamic programming algorithm with the same time complexity. This algorithm solves a more general problem with job weights, a special case of \( 1/pmttn, r_j/\sum w_j U_j \) in which the jobs can be ordered so that \( r_1 \leq r_2 \leq \ldots \leq r_n, p_1 \leq p_2 \leq \ldots \leq p_n \) and \( w_1 \geq w_2 \geq \ldots \geq w_n \). Both preemptive and non-preemptive versions can be solved relatively easily in time \( O(n \log n) \) if either the release times or due-dates of all jobs are equal (Moore [4]).

In this paper we deal with another special case of the problem \( 1/pmttn, r_j/\sum U_j \) when job processing times and due-dates are tied in the following way. For each pair of jobs \( i, j \) with \( d_i > d_j, p_i \geq p_j \); for \( d_i = d_j, p_i \) and \( p_j \) have no further restriction. Our model is motivated by some practical applications when the manufacturer prefers to finish shorter jobs ahead longer ones in order to provide the customer with the maximal amount of the completed jobs ASAP. Then the manufacturer sets the due-dates so that shorter jobs have smaller due-dates. The algorithm we suggest has a superb time complexity of \( O(n^2) \) compared to the above mentioned algorithms, which makes it more appropriate for the above type of applications.

2 The algorithm

In this section we present the algorithm and its correctness proof. First, we give the following general observation. Due to the nature of our objective function, if a job is late then it can obviously be scheduled arbitrarily late without affecting our objective function. Suppose \( S \) is a feasible schedule with all its
jobs being included on-time, and we can assert that we have included the maximal possible number of jobs in it. Then we can append all the omitted jobs in an arbitrary feasible fashion at the end of \( S \), in linear time. Because of this, we shall take care only on on-time scheduling of jobs. Thus we shall exclusively deal with the schedules in which all the jobs are scheduled on time; among all such schedules, we shall look for one containing the maximal possible number of jobs, i.e., an optimal one.

**The description**

First we give a brief description of our algorithm. Our first step is to renumber jobs in \( J \) in a non-decreasing order of their due-dates. After this preprocessing with the cost of \( O(n \log n) \) the jobs in \( J \) are ordered so that \( d_1 \leq d_2 \leq \ldots \leq d_n \). According to our assumption, this also yields \( p_1 \leq p_2 \leq \ldots \leq p_n \). Iteratively, we shall process the jobs in this order trying to schedule each next incoming job \( i, i = 1, 2, \ldots, n \), at its release time. If the conflict with some already scheduled job(s) occurs then it might be split and scheduled within the available idle time intervals in case it can be completed by time \( d_i \). Otherwise \( i \) is omitted. We give some more details below.

We will have \( n \) basic iterations, \( i = 1, 2, \ldots, n \), so that we try to schedule job \( i \) at iteration \( i \). We use a doubly linked list \( L \) to keep the track of the already occupied time intervals on the machine. Each element of this list keeps two parameters which are left and right limits of the corresponding interval. The elements will be organized so that the right limit of the interval represented by an element of the list is strictly less than the left time interval represented by the next element(s) of the list.

Now we describe procedure \( \text{SCAN}(i) \) which is the main body of our algorithm used at each iteration \( i \). \( \text{SCAN}(i) \) either includes job \( i \) updating the current list \( L \) or establishes that \( i \) cannot be included, in which case \( i \) is omitted. Initially on iteration 1 \( \text{SCAN}(1) \) includes job 1 within the interval \([r_1, r_1 + p_1]\) and adds our first element to \( L \) representing the time (execution) interval of job 1.

Suppose for \( i \geq 2 \) the time interval \([r_i, r_i + p_i]\) is idle by iteration \( i \). Then job \( i \) is scheduled within that interval on iteration \( i \). Either a new element is added to \( L \) or one or two elements from \( L \) might be converted into one element with the modified limits representing the time intervals or at least two jobs including job \( i \) (so the corresponding time intervals are merged). A new element will be added to \( L \) if either \( r_i + p_i \) is strictly less than the left-limit of the first element of \( L \) (in this case the new element representing time interval of \( i \) will be inserted as the first one in \( L \)) or \( r_i \) is strictly more than the right limit of the last element of \( L \) (in this case the new element will be inserted as the last one in \( L \)). Otherwise, if there are two successive elements in \( L \) such that \( r_i \) is strictly more than the left limit of the first of these elements, and \( r_i + p_i \) is strictly less than the left-limit of the second element, then the new element representing time interval of \( i \) will be inserted in between these two elements. Otherwise, if the equality is reached in one or both of the above strict inequalities then the corresponding (two or three) time intervals are merged. The resultant time interval will be represented by a single new element of \( L \) substituting the above one or two elements. The left and right limits of the new interval are determined in the obvious way.

If none of the above cases occur then the time interval \([r_i, r_i + p_i]\) is not idle by iteration \( i \), i.e., it intersects with some time interval(s) already represented in \( L \) in more than one point. In this case job \( i \) might be included or may not. \( \text{SCAN}(i) \) scans all occupied time intervals from time moment \( r_i \) searching for the earliest idle time moment \( t \) such that \( t + p_i \leq d_i \). If there exists no such \( t \) then job \( i \) is omitted; otherwise, if \( i \) can be completely scheduled from the moment \( t \) (no intersection with any other occupied interval occurs) then it is scheduled within the interval \([t, t + p_i]\). Otherwise, the above verification is repeated; i.e., the next earliest idle time moment \( t \) is looked for such that \( t + p_i \leq d_i \), where \( p_i^* \) is the length of the yet unscheduled portion of job \( i \). Similar steps are carried out till either job \( i \) is feasibly included or it cannot be feasibly included as above and hence it is omitted. In case of success \( (i \) is feasibly included) all the corresponding time intervals are merged in the straightforward way into a single time interval now also representing the execution interval of job \( i \), the corresponding elements of \( L \) being replaced by a new single element with the above determined left and right limits.

This completes the description of the procedure \( \text{SCAN}(i) \). The overall algorithm is now as follows:

**ALGORITHM MAIN**

Step 1.  
Reorder jobs in \( J \) so that \( d_1 \leq d_2 \leq \ldots \leq d_n \)

Step 2.  
FOR \( i = 1 \) to \( n \) DO \( \text{SCAN}(i) \).

**The correctness proof**

Theorem 1. The algorithm \( \text{MAIN} \) produces an optimal schedule in time \( O(n^2) \).
Proof. As to the time complexity, Step 1 which is sorting, takes time $O(n \log n)$. We have $n$ iterations on Step 2, on each of which $SCAN(i)$ is called. It remains to see the time complexity of this procedure. If the next incoming job $i$ can be included without intersecting with any of the intervals in $L$ then it is included in a constant time and the update of $L$ with the new execution interval (that of job $i$) will take time $O(n)$ as clearly, there are no more than $n$ intervals in $L$. In general, while inserting the next incoming job, we may need to skip at most $n-1$ time intervals from $L$. All the skipped time intervals are to be merged in a single time interval (unified by the portions of the newly included job $i$). Hence we will spend time $O(n)$ for the inclusion of each $i$ and update of $L$ on each iteration. Clearly, we will spend the same amount of time whenever $i$ cannot be included. Hence the overall time complexity is $O(n \log n) + O(n)O(n) = O(n^2)$.

We now switch to the soundness part. According to our renumbering of jobs on Step 1, the shortest jobs will be included ahead longer ones on Step 2. The algorithm is clearly optimal if $SCAN(i)$ has succeeded to include every next incoming job $i$. Otherwise, let $i$ be the first incoming job that could not have been included. Then due to our assumption that job release times and due-dates are agreeable, the feasible interval $(r_i, d_i)$ of $i$ must intersect with some interval(s) from $L$ representing the execution intervals of one or more already assigned jobs. Let us denote this set of jobs by $I(i)$. From our construction, the due-date of every $j \in I(i)$ is no more than $d_i$. Hence, $j$ cannot be moved (feasibly) out of the feasible interval of $i$, and it follows that either job $i$ or one of the $j$-s is to be omitted in any feasible schedule. Suppose we omit any subset of jobs from $I(i)$. Then we claim that we may include no more than $|I(i)|$ (yet unscheduled) jobs instead. Indeed, if we remove some $j \in I(i)$ including instead some yet unscheduled job $l$, then $l$ will again be scheduled within the feasible interval of $i$, whereas it will take no less space than was taken by job $j$, as $p_l \geq p_j$. Jobs $j$ and $l$ might be replaced by a corresponding job sets and a similar reasoning can be applied to these job sets. It follows that there is no benefit in moving or omitting one of the already assigned jobs. And since the corresponding set together with job $i$ is not feasible, job $i$ is to be omitted. We reiterate the whole reasoning for every next incoming job that cannot be feasibly included completing in this way the proof. \qed

Acknowledgements

This work was Partially supported by CONACyT grant 48433.

References:


