Minimizing the weighted number of tardy task units†

Joseph Y.-T. Leung*, Vincent K. M. Yu*, W.-D. Wei†

*Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588-0115, USA
†Department of Mathematics, Sichuan University, Chengdu, 610064, China

(Received 18 June 1991; revised 18 August 1992)

Abstract

The problem of minimizing the weighted number of tardy task units on a single processor is considered. We give an $O(n \log n + kn)$-time algorithm for a set of $n$ tasks with $k$ distinct weights. The relation of this problem with that of minimizing the total weighted error in the imprecise computation model is also discussed.

Key words: Preemptive schedule; Tardy task units; On-time task; Late task; Feasible schedule; Imprecise computation; Single processor

1. Introduction

We consider the problem of preemptively scheduling a set \{\(T_1, T_2, \ldots, T_n\)\} of $n$ independent tasks on a single processor with the objective of minimizing the weighted number of tardy task units. For each task $T_i$, we denote by $r_i, d_i, p_i$ and $w_i$ its release time, deadline, processing time and weight, respectively. (In this paper we assume that all parameters are positive integers.) Each task must start at or later than its release time, and it must be processed for a total duration equal to its processing time. With respect to a schedule $S$, a task is on-time if it is completed by its deadline; otherwise, it is tardy. A feasible schedule is one in which there is no tardy task. The time units during which a task is processed beyond its deadline are called the tardy task units; let $t_i$ denote the number of tardy units of $T_i$. Our problem can be stated as
follows: Given a task system \( TS = (\{T_i\}, \{r_i\}, \{d_i\}, \{p_i\}, \{w_i\}) \) with \( n \) independent tasks, find a schedule \( S \) for \( TS \) on a single processor such that the weighted number of tardy task units, \( \sum_{i=1}^{n} w_i t_i \), is minimized.

The above problem was first studied by Blazewicz [1] who gave a linear programming solution for multiprocessor systems. Subsequently, Blazewicz and Finke [2] showed that it can also be solved by a network flow technique; their algorithm runs in \( O(n^6) \) time if the task weights are all integers and \( O(n^2 \log n) \) time otherwise. For a single processor, Hochbaum and Shamir [4] gave an \( O(n^2) \)-time algorithm for the weighted case and an \( O(n \log n) \)-time algorithm for the unweighted case. Potts and van Wassenhove [8] have also considered similar problems in production scheduling framework.

The problem of minimizing the weighted number of tardy task units is closely related to that of minimizing the total weighted error in the imprecise computation model [7, 9, 10]. The imprecise computation model, introduced by Lin et al. [5, 6], was designed to tradeoff the accuracy of task computation for meeting deadline constraints of real-time tasks. In this model, each task \( T_i \) consists of two subtasks, the mandatory subtask \( M_i \) and the optional subtask \( O_i \), with \( m_i \) and \( o_i \) denoting their processing times, respectively. In scheduling tasks of this kind, it is stipulated that all mandatory subtasks be completed by their deadlines, while the optional subtasks can be left unfinished. If an optional subtask is not completed by its deadline, it incurs an error equal to the product of its weight and the length of its unfinished portion. The goal is to find a schedule such that the total weighted error is minimized. (For this problem, we assume that there is a feasible schedule for all mandatory subtasks.)

Minimizing the weighted number of tardy task units can be regarded as a special case of minimizing the total weighted error in the imprecise computation model: Simply let \( m_i = 0 \) and \( o_i = p_i \) for \( 1 \leq i \leq n \). On the other hand, minimizing the total weighted error in the imprecise computation model can be solved by any algorithm for minimizing the weighted number of tardy task units. This can be done by treating \( M_i \) and \( O_i \) as two different tasks: \( O_i \) has weight \( w_i \) and \( M_i \) is assigned a new weight \( W > \max \{w_i \mid 1 \leq i \leq n \} \). This will ensure that all mandatory subtasks be scheduled on-time. Thus, the complexities of the two problems are equivalent.

For the problem of minimizing the total weighted error in the imprecise computation model, Shih et al. [10] gave a network flow approach similar to the one by Blazewicz and Finke [2]; their algorithm runs in \( O(n^2 \log^2 n) \) time for the unweighted case and \( O(n^6) \) time for the weighted case. For a single processor, Shih et al. [9] gave an \( O(n \log n) \)-time algorithm for the unweighted case and an \( O(n^3 \log n) \)-time algorithm for the weighted case.

In this article we give a new algorithm for minimizing the weighted number of tardy task units on a single processor. Our algorithm runs in \( O(n \log n + kn) \) time, where \( k \) is the number of distinct weights. Since \( k \) is between 1 and \( n \), the running time of our algorithm lies between \( O(n \log n) \) and \( O(n^2) \). As will be seen later, our algorithm is based on the techniques used in [4, 9].

In the next section, we will review the algorithm by Hochbaum and Shamir [4] and that by Shih et al. [9]. In Section 3, we give the new algorithm and show that it solves the problem of minimizing the weighted number of tardy task units. Finally, we draw some concluding remarks in the last section.
2. Review of known algorithms

We begin by describing the algorithm of Hochbaum and Shamir [4] which solves the unweighted case. For brevity, we will denote their algorithm as Algorithm HS. We will give a slight variation of their algorithm; it schedules only the nontardy units of a task rather than the whole task. A full schedule can be obtained from the partial one by scheduling the tardy units after all the nontardy units in an arbitrary manner.

Algorithm HS assumes that the tasks have been indexed in nonincreasing order of release times. Let \(0 = u_0 < u_1 < \cdots < u_p = \max_{i \in n} \{d_i\}\) be the \(p + 1\) distinct integers obtained from the multiset \(\{r_1, r_2, \ldots, r_n, d_1, \ldots, d_n\}\). These \(p + 1\) integers divided the time frame into \(p\) segments: \([u_0, u_1], [u_1, u_2], \ldots, [u_p, u_{p+1}]\). The output of Algorithm HS is an \(n \times p\) matrix \(S\), where \(S_{ij}\) is the number of time units task \(T_i\) is scheduled in segment \(j\) ([\(u_{i-1}, u_i\)]). Below is a formal description of the algorithm.

**Algorithm HS**

1. For \(i = 1, \ldots, p\) do: \(l_i \leftarrow u_i - u_{i-1}\).
2. For \(i = 1, \ldots, n\) do:
   - Find \(a\) satisfying \(u_a = d_i\) and \(b\) satisfying \(u_b = r_i\).
   - For \(j = a, a - 1, \ldots, b + 1\) do:
     - \(\delta \leftarrow \min \{l_j, p_i\}\).
     - \(S_{ij} \leftarrow \delta, l_j \leftarrow l_j - \delta, p_i \leftarrow p_i - \delta\).
   - repeat
   - repeat

Algorithm HS schedules tasks in nonincreasing order of release times. When a task is scheduled, it is assigned from the latest segment \([u_{a-1}, u_a]\) in which it can be nontardy until the earliest segment \([u_b, u_{b+1}]\), with the maximum number of time units assigned in each segment.

Let us examine the complexity of Algorithm HS. The time it takes to index the tasks in nonincreasing order of release times as well as obtaining the set \(\{u_0, u_1, \ldots, u_p\}\) is \(O(n \log n)\). Step 1 of the algorithm takes linear time and a straightforward implementation of Step 2 takes \(O(n^2)\) time. Thus, it appears that the running time of Algorithm HS is \(O(n^2)\). However, observe that whenever a value of some \(S_{ij}\) is increased, either all the units of a task have been scheduled or a segment has been saturated (or both). Hence, at most \(n + p - 1 = O(n)\) values of \(S_{ij}\)'s will be positive in the solution. If we can avoid scanning all those pairs \((i, j)\) for which \(S_{ij} = 0\), then Step 2 will only take linear time and hence the overall running time of the algorithm is \(O(n \log n)\). As it turns out, this can be done by the special UNION-FIND algorithm due to Gabow and Tarjan [3]. We will omit the description here; the reader is referred to [4] for details.

A schedule produced by Algorithm HS will be denoted as a HS-schedule. Define a block as a maximal time interval in which there is only one task assigned (task block) or the processor is idle (idle block). Without any increase in time complexity, Algorithm HS can be modified to produce a schedule represented by a doubly linked list of blocks. In the following we will show that the number of blocks in a
HS-schedule is no more than $2n + 1$. This fact will be used in Section 3 when we analyze the time complexity of our algorithm.

**Lemma 2.1.** The number of blocks in a HS-schedule is no more than $2n + 1$.

**Proof.** Let $a_i$ denote the number of task blocks of which $T_i$ consists and let $b_i$ denote the number of idle blocks just before $T_i$ is scheduled, $1 \leq i \leq n$. Let $b_{n+1}$ denote the total number of idle blocks in a HS-schedule. By definitions, the number of blocks in a HS-schedule is $\sum_{i=1}^{n} a_i + b_{n+1}$. In the following we will show that this quantity is no more than $2n + 1$.

It is obvious that $b_1 = 1$. When $T_i$ is scheduled, it is always assigned the maximum number of time units in a segment. Furthermore, it is assigned from time $d_i$ until $r_i$. Thus, the $a_i$ pieces of $T_i$ will eliminate at least $a_i - 2$ idle blocks from the schedule. Therefore, we have

$$b_{i+1} \leq b_i - (a_i - 2) = b_i - a_i + 2$$

for each $1 \leq i \leq n$. Consequently, we have

$$\sum_{i=1}^{n} b_{i+1} \leq \sum_{i=1}^{n} b_i - \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} 2.$$

Simplifying, we obtain

$$\sum_{i=1}^{n} a_i + b_{n+1} \leq b_1 + 2n = 2n + 1. \quad \square$$

The algorithm for the weighted case uses Algorithm HS as a subroutine. It assumes that the tasks have been indexed in nonincreasing order of weights. Let $c_i$, $1 \leq i \leq n$, denote the number of tardy units of $T_i$ in an optimal schedule. Using the earliest due date (EDD) rule, an optimal schedule can easily be obtained in $O(n \log n)$ time once the $r_i$'s are known. The algorithm determines these values in phases as follows. After $j$ phases, it would have already determined the values $r_1, r_2, \ldots, r_j$. In the $(j+1)$st phase, it uses Algorithm HS to solve the unweighted subproblem for $T_1, T_2, \ldots, T_{j+1}$, where the processing times of the first $j$ tasks are $p_1, \ldots, p_j$, and the processing time of the $(j + 1)$st task is $p_{j+1}$. Let $x$ be the number of tardy units in the HS-schedule obtained. $r_{j+1}$ is then given by $p_{j+1} - x$, and the algorithm proceeds to the next phase.

The algorithm for the weighted case uses Algorithm HS as a subroutine. Since Algorithm HS takes linear time after the initial sorting, the running time of the algorithm becomes $O(n^2)$.

We now turn our attention to the problem of minimizing the total weighted error in the imprecise computation model. As noted before, Shih et al. [9] gave an $O(n \log n)$-time algorithm for the unweighted case. Their algorithm is based on a slight variation of the EDD rule which operates exactly like the EDD rule, except that a task will be terminated and the remaining units discarded when its deadline is reached. Let us denote this variation of the EDD rule as the MEDD rule.

The algorithm of Shih et al. [9] consists of three parts. In the first part, it uses the MEDD rule to obtain a schedule $S_m$ for $M$, where $M$ is the set of all mandatory
subtasks. In the second part, it uses the MEDD rule to obtain a schedule $S_1$ for $M \cup O$, where $O$ is the set of all optional subtasks. In the last part, which they called the “adjustment step”, it transforms $S_1$ into an optimal schedule $S_m$, using $S_m$ as a template. The adjustment step is needed to ensure that all of the mandatory subtasks are on-time.

The adjustment step proceeds as follows. Let there be $q$ blocks in $S_m$: $V_i = [v_{i-1}, v_i], 1 \leq i \leq q$. By definition, each block either has one task scheduled or is idle. $S_1$ is transformed block by block, from $V_q$ down to $V_1$. No adjustment is needed for those blocks which correspond to idle blocks in $S_m$. Let $V_i$ be a task block in $S_m$, and let $M_i$ be the task scheduled within $V_i$ in $S_m$. If the number of time units assigned to $M_i$ in $S_m$ at time $v_{i-1}$ or after is larger than that in $S_1$, then more of $M_i$ will be assigned within $V_i$ in $S_1$ to make up the difference, by removing any task, expect $M_i$, that was originally assigned within $V_i$. (As pointed out in [9], the reassignment can always be done.) Otherwise, the block needs no adjustment. When the transformation is completed, the final schedule would have the property that all of the mandatory subtasks are on-time.

The first two steps of the above algorithm takes $O(n \log n)$ time, since the MEDD rule takes $O(n \log n)$ time. The adjustment step can be implemented to run in linear time; see also Section 3 for a description of implementation. Thus, the overall running time of the algorithm is $O(n \log n)$.

The algorithm of Shih et al. [9] for the weighted case is very similar to the one by Hochbaum and Shamir [4], except that the unweighted algorithm of Shih et al. [9] is used to solve the unweighted subproblems instead. Assume that the tasks have been indexed in nonincreasing order of weights. It first applies the unweighted algorithm to $M \cup \{O_1\}$ (Note that $O_1$ has the largest weight among all the optional subtasks.) Let $x_1$ be the number of tardy units in the schedule obtained. It creates a mandatory subtask $M_1$ with processing time $\pi_1 = o_1 - x_1$. The algorithm then proceeds to the next iteration, applying the unweighted algorithm to $M \cup \{M_1\} \cup \{O_2\}$. Again, it creates a mandatory subtask $M_2$ with processing time $\pi_2 = o_2 - x_2$, where $x_2$ is the number of tardy units in the schedule just obtained. This process is repeated until $\pi_1, \pi_2, \ldots, \pi_n$ are all determined; $\pi_i$ will be the number of nontardy units of $O_i$ in an optimal schedule.

The above algorithm invokes $n$ calls to the unweighted algorithm. Since the unweighted algorithm needs $O(n \log n)$ time (dictated by the $O(n \log n)$ time of the MEDD rule), the overall running time of the algorithm becomes $O(n^2 \log n)$.

3. The new algorithm

Our algorithm is based on the ideas of Hochbaum and Shamir [4] and Shih et al. [9]. Like the algorithm of Hochbaum and Shamir, tasks are considered in decreasing order of weights. Let $w_1 > w_2 > \cdots > w_k$ be the $k$ distinct weights of the tasks in $TS$, and let $TS$ be partitioned into $k$ sets, $TS_1, TS_2, \ldots, TS_k$, such that $TS_j$ consists of all the tasks with weight $w_j, 1 \leq j \leq k$. The algorithm constructs $k$ sets of tasks, $\bar{TS}_1, \bar{TS}_2, \ldots, \bar{TS}_k$, where $\bar{TS}_j$ consists of all the tasks in $TS_j$ with processing times equal to the numbers of nontardy units of the tasks in an optimal schedule, $1 \leq j \leq k$. 


Using the EDD rule to schedule these $k$ sets of tasks, an optimal schedule can be obtained in $O(n \log n)$ time.

The algorithm proceeds in $k$ phases. At the end of the $j$th phase, it would have already determined the sets $\bar{T}S_1, \bar{T}S_2, \ldots, \bar{T}S_j$. The union of these sets is stored in $\bar{T}S$ (which initially is the empty set), and a schedule obtained by Algorithm HS for $\bar{T}S$ is stored in $\bar{S}$ (which initially is the empty schedule). In the $(j + 1)$st phase, it uses Algorithm HS to construct a schedule $S_{j+1}$ for $\bar{T}S \cup TS_{j+1}$. Then it goes through an adjustment step (described below), transforming $S_{j+1}$ into $\bar{S}_{j+1}$ with $\bar{S}$ as a template.

The algorithm then repeats the above process in the next phase.

The adjustment step mentioned above proceeds as follows. Let there be $q$ blocks in $\bar{S}$: $V_i = [v_{i-1}, v_i]$, $1 \leq i \leq q$. $S_{j+1}$ is transformed block by block, from $V_1$ to $V_q$. (Unlike the transformation by Shih et al. [9], our adjustment proceeds from earlier time to later time. This is because Algorithm HS schedules tasks from later time to earlier time, rather than from earlier time to later time as the MEDD rule does.) Adjustment of $S_{j+1}$ is necessary only for those blocks which correspond to task blocks in $\bar{S}$. Let $V_i$ be a task block in $\bar{S}$, and let $T_i$ be the task scheduled within $V_i$ in $\bar{S}$. Let $\bar{N}(i)$ (resp. $N_{j+1}(i)$) denote the number of time units $T_i$ has executed in $\bar{S}$ (resp. $S_{j+1}$) from the beginning until time $v_i$. If $\bar{N}(i) > N_{j+1}(i)$, then assign $(\bar{N}(i) - N_{j+1}(i))$ more time units to $T_i$ within $V_i$ in $S_{j+1}$, by removing any task, except $T_i$, that was originally assigned within $V_i$. (Note that the reassignment can always be done.) Otherwise, no adjustment is needed.

Fig. 1 gives a set of tasks with two distinct weights. The schedule $\bar{S}$ after the first phase is shown in Fig. 1(a). $S_2$ and $\bar{S}_2$ are shown in Figs. 1(b) and (c), respectively. Finally, the schedule $\bar{S}$ after the second phase is shown in Fig. 1(d); this is an optimal schedule for the set of tasks.

A formal description of our algorithm, to be called Algorithm A, is given below.

Algorithm A

**Input:** A single processor and a task system $TS$ with $k$ distinct weights, $w_1 > w_2 > \cdots > w_k$. Assume that $TS = TS_1 \cup TS_2 \cup \cdots \cup TS_k$, where $TS_j$, $1 \leq j \leq k$, consists of all the tasks with weight $w_j$.

**Output:** An optimal schedule $\bar{S}$ for $TS$.

**Method:**

1. $\bar{T}S \leftarrow \emptyset$ and $\bar{S} \leftarrow$ empty schedule.
2. For $j = 1, \ldots, k$ do:
   
   2.1 $S_j \leftarrow$ schedule obtained by Algorithm HS for $\bar{T}S \cup TS_j$.
   
   Begin (Adjustment Step)
   
   Let there be $q$ blocks in $\bar{S}$: $V_i = [v_{i-1}, v_i]$, $1 \leq i \leq q$.
   
   For $i = 1, \ldots, q$ do:
   
   If $V_i$ is a task block in $\bar{S}$ then
Let $T_i$ be the task executed within $V_i$ in $\hat{S}$. Let $\hat{N}(i)$ (resp. $N_j(i)$) be the number of time units $T_i$ has executed in $\hat{S}$ (resp. $S_j$) from the beginning until time $v_i$.

If $\hat{N}(i) > N_j(i)$ then
assign \((\hat{N}(i) - N_j(i))\) more time units to \(T_i\) within \(V_i\) in \(S_j\), by replacing any task, except \(T_k\), that was originally assigned within \(V_i\).

```endif
endif
```

repeat

```End
\hat{S}_j \leftarrow S_j.
\hat{T}S_j \leftarrow \text{the tasks in } TS_j \text{ with the processing time of each task equal to the number of nontardy units of the task in } \hat{S}_j.
\hat{T}S \leftarrow \hat{T}S \cup \hat{T}S_j.
\hat{S} \leftarrow \text{schedule obtained by Algorithm HS for } \hat{T}S.
```

Let us examine the time complexity of Algorithm A. Observed that Algorithm A utilizes Algorithm HS to construct schedules for various subsets of \(TS\). As noted in Section 2, Algorithm HS requires that the release times and deadlines of the tasks be ordered. With an initial sort of the release times and deadlines of all the tasks in \(TS\), we can obtain in linear time an ordering of the release times and deadlines of the tasks for any subset of \(TS\). Furthermore, once the ordering is obtained, Algorithm HS only needs linear time to construct a schedule.

Step 1 of Algorithm A takes constant time and Step 2 is iterated \(k\) times. If we can show that each iteration of Step 2 takes \(O(n)\) time (after the initial sorting), then the overall running time of Algorithm A becomes \(O(n \log n + kn)\). From the above discussions, it is clear that every substep in Step 2, except possibly the adjustment step, takes linear time. In the following we will show that the adjustment step can be done in linear time. As mentioned in Section 2, Algorithm HS can be modified, with no increase in time complexity, to produce a schedule represented by a doubly linked list of blocks. Thus, we may assume that \(\hat{S}\) and \(S_j\) are in this representation. The adjustment process is performed by traversing the two linked lists, modifying \(S_j\), if necessary, as the list is traversed. By Lemma 2.1, the number of blocks in a HS-schedule is linear to the number of tasks. The values \(\hat{N}(i)\) and \(N_j(i)\) can be obtained with the help of two one-dimensional arrays \(L^i\) and \(L\): \(L^i(l)\) (resp. \(L(l)\)) contains the number of time units \(T_i\) has executed in \(\hat{S}\) (resp. \(S_j\)) since the beginning. \(L\) and \(L^i\) initially have zero in every entry, and they are updated as the linked lists are traversed. Thus, the adjustment process takes linear time.

From the above discussions, we have the following theorem.

**Theorem 3.1.** Algorithm A has a worst-case time complexity of \(O(n \log n + kn)\).

We now turn our attention to the correctness proof of the algorithm. The next theorem shows the Algorithm A always generates an optimal schedule.

**Theorem 3.2.** Algorithm A always generates a schedule with the minimum weighted number of tardy task units.
Proof. We shall prove, by induction on $j$, that the following two properties hold at the end of the $j$th iteration in Step 2 of the algorithm: (1) The total processing time of the task in $\tilde{S}_1 \cup \tilde{S}_2 \cup \cdots \cup \tilde{S}_j$ is the maximum number of nontardy units that can be scheduled for the tasks in $TS_1 \cup TS_2 \cup \cdots \cup TS_j$, and (2) $\tilde{S}$ is a schedule for $TS_1 \cup TS_2 \cup \cdots \cup TS_j$ with the maximum weighted number of nontardy units. The theorem immediately follows from (2) when $j = k$.

The basis case, $j = 1$, is obvious. Assume that the inductive hypothesis holds for $j - 1$, we wish to prove that it also holds for $j$. Observe that $S_j$ is the schedule obtained by Algorithm HS for $TS_1 \cup \cdots \cup TS_{j-1} \cup TS_j$. Since Algorithm HS always generates a schedule with the maximum number of nontardy units, it follows that $S_j$ has the maximum number of nontardy units for the tasks in $\tilde{S}_1 \cup \cdots \cup \tilde{S}_{j-1} \cup TS_j$. By the inductive hypothesis, the total processing time of the tasks in $\tilde{S}_1 \cup \tilde{S}_2 \cup \cdots \cup \tilde{S}_{j-1}$ is the maximum number of nontardy units that can be scheduled for the tasks in $TS_1 \cup TS_2 \cup \cdots \cup TS_{j-1}$. Thus, $S_j$ has the maximum number of nontardy units for the tasks in $TS_1 \cup TS_2 \cup \cdots \cup TS_j$.

The adjustment step will not change the number of nontardy units. Thus, $\tilde{S}_j$ has the same number of nontardy units as $S_j$. Since $\tilde{S}_j$ is obtained from $\tilde{S}_1$, it follows that (1) holds.

Observe that $\tilde{S}$ is a feasible schedule for $\tilde{S}_1 \cup \tilde{S}_2 \cup \cdots \cup \tilde{S}_j$, i.e. every task is fully scheduled. By the inductive hypothesis, a feasible schedule for $\tilde{S}_1 \cup \cdots \cup \tilde{S}_{j-1}$ is an optimal schedule for $TS_1 \cup \cdots \cup TS_{j-1}$. Since (1) holds and since the tasks in $\tilde{S}_j$ have a smaller weight than any task in $\tilde{S}_1 \cup \cdots \cup \tilde{S}_{j-1}$, it follows that (2) holds. □

4. Conclusions

In this article we have studied the problem of minimizing the weighted number of tardy task units on a single processor. We showed that this problem is related to that of minimizing the total weighted error in the imprecise computation model. We gave an $O(n \log n + kn)$-time algorithm for this problem, where $n$ is the number of tasks and $k$ is the number of distinct weights. The time complexity of our algorithm is an improvement over those of Hochbaum and Shamir [4] and Shih et al. [9] when $k$ is small.

Acknowledgement

We would like to thank the anonymous referees for their helpful suggestions.

References