JOURNAL OF ALGORITHMS 14, 45–68 (1993)

Minimizing Mean Flow Time with Release Time and Deadline Constraints*

JIANZHONG DU AND JOSEPH Y-T. LEUNG

Computer Science Program, University of Texas at Dallas, Richardson, Texas 75083

Received February 23, 1988; accepted September 26, 1991

The problem of preemptively scheduling a set of n independent tasks on a single processor so as to minimize the mean flow time is considered. Each task has associated with it a processing time, a release time, and a deadline. A feasible schedule is one in which each task is completely processed within its interval of release time and deadline. The problem is to find a feasible schedule with the minimum mean flow time. It is known that two special cases of the problem can be solved in $O(n \log n)$ time: the case of identical deadlines by the *smallest remaining processing time* rule as described by Baker and the case of identical release times by the *deadline* rule as described by Smith. In this paper we generalize both algorithms and show that they solve a much broader class of problem instances, namely those which contain no triple of tasks satisfying certain conditions. The generalized algorithms, and an algorithm that recognizes problem instances satisfying the condition, can all be implemented in $O(n^2)$ time. Finally, we show that the general problem is NP-hard, answering an open question posed by Lawler.

1. INTRODUCTION

Suppose we are given a set $\{T_1, T_2, \ldots, T_n\}$ of n independent tasks, where task j has a processing time p_j, a release time r_j, and a deadline d_j. The n tasks are to be preemptively scheduled on a single processor so that each task is completely processed within its interval of release time and deadline; such a schedule is called a *feasible* schedule. Our problem is to find a feasible schedule such that ΣC_j is minimized, where C_j is the completion time of task j in the schedule. In the notation of [6], the

*Research supported in part by the ONR Grant N00014-87-K-0833 and in part by a grant from Texas Instruments, Inc.

0196-6774/93 $5.00

scheduling problem studied in this paper is $1|pmtn, r_j, d_j|\Sigma C_j$. From now on, we shall be using the notation of [6].

It is known that two special cases of this problem can be solved in $O(n \log n)$ time: $1|pmtn, r_j|\Sigma C_j$ by the *smallest remaining processing time* rule of Baker [1], and $1|pmtn, d_j|\Sigma C_j$ by the *deadline* rule of Smith [11]. In fact, Smith's deadline rule solves the problem $1|d_j|\Sigma C_j$. Since preemption cannot reduce ΣC_j for a set of tasks with identical release times, Smith's *deadline* rule solves $1|pmtn, d_j|\Sigma C_j$ as well.

We show that Baker's algorithm and Smith's algorithm can each be generalized to solve a much broader class of problem instances, namely those which contain no ordered triple of tasks (i, j, k) with $r_i, r_k < r_j < d_i < d_j, d_k$ and $p_j < p_i, p_k$. Such a triple will be called an *obstruction* in this paper. The generalized Baker's algorithm, the generalized Smith's algorithm, and an algorithm recognizing problem instances with no obstruction, can all be implemented to run in $O(n^2)$ time.

Note that problem instances solved by the generalized Baker's algorithm and the generalized Smith's algorithm include those for which:

(1) The intervals $[r_j, d_j]$, $j = 1, 2, \ldots, n$, are nested; i.e., there is no pair of indexes i and j such that $r_i < r_j < d_i < d_j$.

(2) Release times and deadlines are oppositely ordered; i.e., there is a numbering of tasks such that

$$r_1 \leq r_2 \leq \cdots \leq r_n \text{ and } d_1 \geq d_2 \geq \cdots \geq d_n.$$

(3) Processing times and deadlines are similarly ordered; i.e., there is a numbering of tasks such that

$$p_1 \leq p_2 \leq \cdots \leq p_n \text{ and } d_1 \leq d_2 \leq \cdots \leq d_n.$$

(4) Processing times and release times are similarly ordered.

(5) Processing times are identical.

We show that the problem $1|pmtn, r_j, d_j|\Sigma C_j$ is NP-hard, answering an open question posed by Lawler [9]. It follows that, although there may be other classes of problem instances that can be solved in polynomial time, there is no polynomial-time algorithm for the general problem, unless P = NP.

Nonpreemptive versions of the general problem are known also to be intractable: $1|r_j|\Sigma C_j$ is NP-hard in the "ordinary" sense [10] and $1|r_j, d_j|\Sigma C_j$ is NP-hard in the "strong" sense [4]. As noted before, the problem $1|d_j|\Sigma C_j$ can be solved by Smith's algorithm [11]. Minimizing mean weighted flow time is even harder: $1|pmtn, r_j|\Sigma w_j C_j$, $1|pmtn, d_j|\Sigma w_j C_j$, and $1|d_j|\Sigma w_j C_j$ are all NP-hard [8, 10]. The only poly-

nomially solvable case is $1||\Sigma w_j C_j$ (and hence $1|pmtn|\Sigma w_j C_j$), which is solved by the ratio rule [11].

The paper is organized as follows. In the next section we present the EDD (earliest due date) rule which enables us to determine if a feasible schedule exists for a given instance of the $1|pmtn, r_j, d_j|\Sigma C_j$ problem [7]. The generalized Baker's algorithm and the generalized Smith's algorithm are given in Sections 3 and 4, respectively. Section 5 is devoted to showing that both of the generalized algorithms solve all problem instances with no obstruction. An $O(n^2)$ time algorithm for recognizing problem instances with no obstruction is given in Section 6. Section 7 shows that both of the generalized algorithms can be implemented to run in $O(n^2)$ time. An NP-hardness proof of the general problem is given in Section 8. Finally, we draw some conclusions in the last section.

2. The EDD Rule and Its Implications

The well-known EDD (earliest due date) rule [7] enables us to determine whether there is a feasible schedule for a given instance of the $1|pmtn, r_j, d_j|\Sigma C_j$ problem: Schedule the tasks over time, with a decision point coinciding with the time a task is completed or a new task is released. At each decision point, from among all the tasks that are available for processing (i.e., tasks that have been released, but not yet completed), choose to process a task with the earliest deadline. If the resulting schedule is not feasible, i.e., some task does not meet its deadline, then no feasible schedule exists.

With an initial sort of the tasks by release times and with the use of a priority queue to keep track of the available tasks, an EDD schedule can be constructed in $O(n \log n)$ time.

Suppose we know that there is an optimal schedule in which the tasks are completed at times $C_1, C_2, \ldots, C_n$. Then we can construct such an optimal schedule by applying the EDD rule with the completion times in the role of deadlines. In fact, we can construct an optimal schedule, even if all we know is the order in which tasks are completed. (This fact implies that the decision version of the problem $1|pmtn, r_j, d_j|\Sigma C_j$ is in the class of NP, since a nondeterministic Turing machine can guess an optimal ordering of the completion times and the EDD rule can be used to construct an optimal schedule from the optimal ordering.)

From the preceding observation, it follows that if $C_i < C_j$ in an optimal schedule, then there is an optimal schedule in which task j is not processed while task i is available for processing, i.e., there is no processing of task j in the interval $[r_i, C_i]$. Also, there is an optimal schedule in

which no task is preempted, except at the release time of another task. Hence there is an optimal schedule with no more than $n - 1$ preemptions.

It is never advantageous to leave the processor idle when there are tasks available for processing. Hence, without knowing the order of the completion times, one can predict the periods of idle time that will occur in an optimal schedule. In fact, they are exactly the same as those in the EDD schedule. It is clear that in minimizing $\sum C_j$ we can consider separately the subsets of tasks that are released between successive periods of idle time.

Actually, a more refined decomposition of the set of tasks may be possible. The following definitions and observations are adapted from [2]. Define a *block* B to be a minimal subset of tasks processed without idle time from $r(B) = \min_{j \in B}\{r_j\}$ until $t(B) = r(B) + \sum_{j \in B} p_j$, and each task j not in B is either completed not later than $r(B)$ (i.e., $C_j \leq r(B)$ when a schedule is constructed by the EDD rule) or not released before $t(B)$ (i.e., $r_j \geq t(B)$). The block structure of a set of tasks can be determined in $O(n \log n)$ time in the course of constructing an EDD schedule: The end of a block is found when no tasks are available at a decision point, other than newly released tasks. It is clear that in minimizing $\sum C_j$ we can deal with each block as a separate subproblem.

In the remainder of this paper we assume that the tasks form a block, unless stated otherwise. Furthermore, we assume that the earliest release time of the tasks is 0.

3. Generalized Baker's Rule

$1|pmtn, r_j|\sum C_j$ can be solved in $O(n \log n)$ time by applying "Baker's rule": Schedule the tasks over time, with a decision point coinciding with the time a task is completed or a new task is released. At each decision point, from among all the tasks that are available for processing choose to process a task with the smallest remaining processing time.

Baker's rule is clearly the same as the EDD rule, except for the way in which tasks are chosen for processing at each decision point. It also admits an $O(n \log n)$ implementation.

Suppose Baker's rule is applied to an arbitrary instance of $1|pmtn, r_j, d_j|\sum C_j$. (In doing so, break ties between tasks with the smallest remaining processing time by choosing a task with the earliest possible deadline.) We assert that if the resulting schedule is feasible, then it is optimal. This follows from the observation that, in effect, we have solved a relaxation of the original problem instance in which all deadlines are made equal and very large. If an optimal schedule for this relaxation happens to be feasible with respect to the original deadlines, then it must be optimal with respect to those deadlines.

Note that feasibility of the schedule produced by Baker's rule can, for example, be guaranteed in the case that processing times, release times, and deadlines are similarly ordered, i.e., there is a numbering of tasks such that

$$p_1 \leq p_2 \leq \cdots \leq p_n,$$
$$r_1 \leq r_2 \leq \cdots \leq r_n,$$
$$d_1 \leq d_2 \leq \cdots \leq d_n.$$

In this case, Baker's rule and the EDD rule produce identical schedules.

But quite typically, Baker's rule will not produce a feasible schedule. One way to modify the rule so that it will always yield a feasible schedule is as follows. Call an available task j *eligible* if it is feasible to complete j prior to any of the other available tasks. Now at each decision point choose to process an eligible task with the smallest remaining processing time; such a task will be called a *smallest eligible task*. (Break ties between smallest eligible tasks by choosing one with the earliest possible deadline.)

One way to determine whether a given task is eligible is to modify its deadline by setting it to the smallest deadline of any of the available tasks, leaving the other deadlines as they are. Then apply the EDD rule to the available and unreleased tasks. If the resulting schedule is feasible, the task is eligible. This suggests an $O(n^3 \log n)$ implementation of the generalized Baker's rule: There are $O(n)$ applications of the EDD rule at $O(n)$

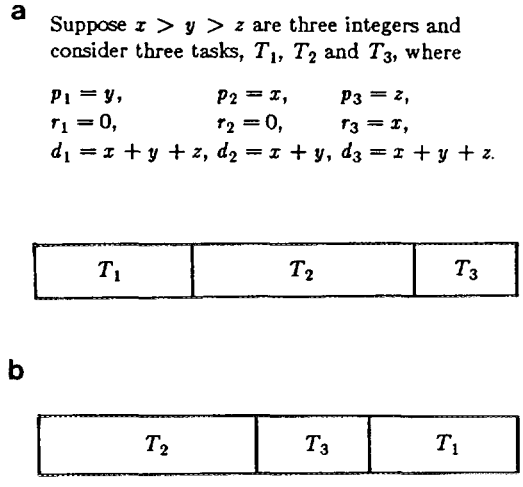

FIG. 1. An example illustrating the generalized Baker's rule and the generalized Smith's rule: (a) schedule produced by the generalized Baker's rule, $\Sigma C_j = 2x + 3y + z$; (b) schedule produced by the generalized Smith's rule, $\Sigma C_j = 3x + y + 2z$.

decision points, with each application requiring $O(n \log n)$ time. In Section 7 we shall describe an $O(n^2)$ implementation.

Unfortunately, the generalized Baker's rule does not necessarily produce an optimal schedule. Consider the three tasks shown in Fig. 1. Figure 1a shows the schedule produced by the generalized Baker's rule, while Fig. 1b gives an alternate schedule. If $y > (x + z)/2$, the second schedule gives a smaller ΣC_j than the first one. In Section 5 we shall derive general conditions under which the generalized Baker's rule is valid.

4. Generalized Smith's Rule

Smith's rule yields optimal schedules for $1|d_j|\Sigma C_j$. Since preemptions cannot reduce ΣC_j when all release times are the same, Smith's rule solves $1|pmtn, d_j|\Sigma C_j$ as well.

Smith's rule is as follows: Find the time t at which the last task will be completed. (If all the release times are 0, then $t = \Sigma p_j$.) From among all the tasks j with $d_j \geq t$, choose a task with the largest processing time. This leaves a problem instance with $n - 1$ tasks, to which the rule is again applied.

Let us now consider how to generalize Smith's rule so that it can be made to apply to tasks with arbitrary release times and deadlines. Recall that we assume the tasks form a block and the earliest release time is 0. For simplicity, we assume that all $d_j \leq t$, where $t = \Sigma p_j$ is the completion time of the last task.

It is not necessarily optimal to choose to complete last a task with the largest processing time, among all the tasks which are feasible to complete last. To see this, consider the two tasks T_1 and T_2, with $p_1 = 4$, $r_1 = 0$, $d_1 = 7$, $p_2 = 3$, $r_2 = 2$, and $d_2 = 7$. Both tasks are feasible to complete last, with $\Sigma C_j = 12$ if T_1 is completed last and $\Sigma C_j = 11$ if T_2 is completed last. If the proposed criterion were applied to the above two tasks, we would have obtained the suboptimal schedule.

In order to obtain a better generalization of Smith's rule, we propose to choose the task that is completed last in an EDD schedule that is constructed with the following refinement: When a choice must be made between tasks with deadline t, choose to process any task with deadline t, other than the one with the largest remaining processing time. (Break ties between tasks with the largest processing time by choosing to process a task with the earliest possible release time.)

If l is the last task completed in this EDD schedule and if there is an optimal schedule S in which l is also the last task completed, then the periods in which l is processed in the EDD schedule are precisely the periods in which l is processed in S. The other $n - 1$ tasks then decom-

pose into one or more blocks. By repeated application of the rule for choosing the last task, a feasible schedule is produced.

The generalized Smith's rule can clearly be implemented to run in $O(n^2 \log n)$ time: There are $O(n)$ applications of the refined EDD rule, with each application requiring $O(n \log n)$ time. In Section 7, we provide an $O(n^2)$ implementation.

Interestingly, the generalized Smith's rule yields an optimal schedule for the example for which the generalized Baker's rule fails. However, the opposite may also be the case. As an example, consider again the three tasks shown in Fig. 1. Figure 1a shows the schedule produced by the generalized Baker's rule, while Fig. 1b gives the schedule produced by the generalized Smith's rule. The first schedule has a smaller ΣC_j than the second one if and only if $y < (x + z)/2$. Thus, by appropriately choosing the values of x, y, and z, we can obtain examples for which the generalized Smith's rule yields an optimal schedule while the generalized Baker's rule fails, and conversely.

5. Validity of the Generalized Rules

We now wish to consider conditions under which we can be assured that the generalized Baker's rule and the generalized Smith's rule yield optimal solutions. It is clear that the generalized Baker's rule yields the same schedules as Baker's rule for $1|pmtn, r_j|\Sigma C_j$. We assert that the generalized Baker's rule yields optimal schedules for $1|pmtn, d_j|\Sigma C_j$, the problem for which Smith's rule was designed. (It is not difficult to derive an inductive proof for this assertion. However, we omit the proof here since Theorem 1 given below also proves this.) It is also clear that the generalized Smith's rule yields the same schedules as Smith's rule for $1|pmtn, d_j|\Sigma C_j$. We assert that the generalized Smith's rule yields optimal schedules for $1|pmtn, r_j|\Sigma C_j$, the problem for which Baker's rule was designed. (Again, it is not difficult to derive an inductive proof for this assertion. We omit the proof here since Theorem 2 given below also proves this.) We wish to show that both of the generalized algorithms yield optimal schedules for a much broader class of problem instances, namely those instances with no obstruction.

Before we show these results, we need to introduce the following definitions and notation. An ordered pair of tasks (i, j) is *skewed* if $r_i < r_j < d_i < d_j$. Task k is said to *cover* the skewed pair (i, j) if $r_k < r_j < d_i < d_k$. The ordered triple (i, j, k) denotes the covered skewed pair with k covering the skewed pair (i, j). A covered skewed pair (i, j, k) is an obstruction if $p_j < p_i, p_k$. A sequence of tasks $(i_1, i_2, \ldots, i_k)$ is called a

skewed sequence from i_1 to i_k if $(i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k)$ are skewed pairs.

For a given set of tasks, let $r^{(1)} < r^{(2)} < \cdots < r^{(l)}$ be the distinct release times of the tasks, and let $R^{(1)}, R^{(2)}, \ldots, R^{(l)}$ be the subsets of tasks with those release times. Let $d^{(1)} < d^{(2)} < \cdots < d^{(m)}$ be the distinct deadlines of the tasks, and let $D^{(1)}, D^{(2)}, \ldots, D^{(m)}$ be the subsets of tasks with those deadlines.

We first consider the generalized Baker's rule. The next lemma gives a condition under which the partial schedule obtained by the generalized Baker's rule at each decision point can lead to an optimal schedule.

LEMMA 1. *There is an optimal schedule in which the task i^* is completed first among all the tasks in $R^{(1)}$ if (1) i^* is a smallest eligible task among all the tasks in $R^{(1)}$ and (2) $p_{i^*} \le p_{i_j}$ whenever there is a skewed sequence $(i_1, i_2, \ldots, i_j)$ from a task $i_1 \in R^{(1)}$ to the task i_j such that $d_{i_{j-1}} < d_{i^*}$.*

Proof. Let S be an optimal schedule in which i^* is not completed first among all the tasks in $R^{(1)}$. We assert that there is always a task k in S with k completed prior to i^* such that interchanging the order of completion times of k with i^* yields another optimal schedule. With this assertion a simple induction will show that S can be transformed into another optimal schedule in which i^* is completed first among all the tasks in $R^{(1)}$.

We will prove this by showing that there is a task k in S such that $C_k < C_{i^*} \le d_k$ and $p_{i^*} \le p_k$. It is clear that if k satisfies this condition, interchanging the order of completion times of k with i^* yields another optimal schedule. In the following we shall show that such a task exists.

First, consider the case $C_{i^*} \le \min\{d_i | i \in R^{(1)}\}$. In this case the task $k \in R^{(1)}$ completing first in S is the desired task. (Since k is completed first in S, k is also an eligible task and hence $p_{i^*} \le p_k$. Furthermore, $C_k < C_{i^*} \le \min\{d_i | i \in R^{(1)}\} \le d_k$.)

Now assume that $C_{i^*} > \min\{d_i | i \in R^{(1)}\}$. With this assumption we consider the following two cases separately: (1) There is a task $i \in R^{(1)}$ with $C_{i^*} \le d_i$ such that i is completed prior to i^* and (2) every task $i \in R^{(1)}$ with $C_{i^*} \le d_i$ is such that i is completed after i^*.

We assert that if there is a task $k \in R^{(1)}$ such that $C_k < C_{i^*} \le d_k$, then $p_{i^*} \le p_k$, and hence k is the desired task. Otherwise, let k be a task in $R^{(1)}$ such that $C_k < C_{i^*} \le d_k$ and $p_k < p_{i^*}$. We shall show that k is also an eligible task, contradicting the fact that i^* is a smallest eligible task.

To show k is eligible, we may assume that $C_k > \min\{d_i | i \in R^{(1)}\}$; as otherwise, k is readily seen to be eligible by considering the feasible schedule obtained from S by interchanging k with the task in $R^{(1)}$ completed first in S. Define E to be the set of all tasks completed prior to

k in S. Since S is an optimal schedule and since k is released at the earliest release time, all the tasks started before C_k are completed by C_k in S. Thus, the tasks processed by C_k in S are exactly those in the set $E \cup \{k\}$. We claim that there is a task $j \in E$ such that $d_j \geq C_k$. Otherwise, every task in E would have deadline earlier than C_k, and hence it would be impossible to complete k prior to C_k in any schedule whatsoever. (This follows from the observation that the total processing time of all the tasks in the set $E \cup \{k\}$ is exactly C_k. Since the deadlines of all the tasks in E are earlier than C_k, it would be impossible to complete k prior to C_k without missing the deadline of some task in E.) Since $p_k < p_{i^*}$, this implies that it would also be impossible to complete i^* prior to C_k in any schedule whatsoever, contradicting the fact that i^* is an eligible task. Thus, there must be a task j in E with $d_j \geq C_k$. Now, if we interchange the order of completion times of j with k, we obtain another feasible schedule in which k is completed earlier than in S. We can repeat the above process until we obtain a feasible schedule in which k is completed by $\min\{d_i | i \in R^{(1)}\}$. This shows that k is also an eligible task.

From the above discussions, we are only left with the case that $C_{i^*} > \min\{d_i | i \in R^{(1)}\}$ and that every task $i \in R^{(1)}$ with $C_{i^*} \leq d_i$ is such that i is completed after i^*. Let $\hat{F}$ be the set of all tasks completed prior to i^* in S with deadlines earlier than C_{i^*} and F be the set of all tasks completed prior to i^* in S with deadlines no earlier than C_{i^*}. Clearly, the set $\hat{F} \cup F$ contains all the tasks completed prior to C_{i^*} in S. Since S is an optimal schedule and since i^* is released at the earliest release time, all the tasks started before C_{i^*} are completed by C_{i^*} in S. Thus, the tasks processed by C_{i^*} in S are exactly those in the set $\hat{F} \cup F \cup \{i^*\}$. Note that $\hat{F}$ is nonempty since $\hat{F} \supseteq \{i | i \in R^{(1)} \text{ and } d_i < C_{i^*}\} \neq \emptyset$. Also, F is nonempty; as otherwise, it would be impossible to complete i^* prior to C_{i^*} in any schedule whatsoever, contradicting the fact that i^* is an eligible task. (Again, this follows from the observation that if F were empty, then the total processing time of all the tasks in the set $\hat{F} \cup \{i^*\}$ would be exactly C_{i^*}. Since the deadlines of all the tasks in $\hat{F}$ are earlier than C_{i^*}, it would be impossible to complete i^* prior to C_{i^*} without missing the deadline of some task in $\hat{F}$.) From the definition of F and the condition stated in the beginning of the paragraph, none of the tasks in F can be in $R^{(1)}$. Hence every task in F is released after the earliest release time.

We claim that there is a task k in F with $p_{i^*} \leq p_k$, and hence k is the desired task to be interchanged with i^*. This can be proved by showing the existence of a skewed sequence $(i_1, i_2, \ldots, i_j)$ from a task $i_1 \in R^{(1)} \cap \hat{F}$ to a task $i_j \in F$ such that $i_1, i_2, \ldots, i_{j-1}$ are all in $\hat{F}$ (and hence $d_{i_{j-1}} < C_{i^*} \leq d_{i^*}$). If such a skewed sequence exists, then, by the second condition of the lemma, we have $p_{i^*} \leq p_{i_j}$, and hence i_j is the desired k. Thus, our proof is reduced to showing that such a skewed sequence exists.

Suppose to the contrary that no such skewed sequence exists. Then, there must be a time t, $\max\{d_i | i \in R^{(1)} \text{ and } d_i < C_{i^*}\} \leq t \leq \min\{r_i | i \in F\}$, such that no task in the set $\hat{F} \cup F$ has release time earlier than t and deadline later than t. The set $\hat{F} \cup F$ can be partitioned into two sets G_1 and G_2 such that G_1 contains all those tasks with deadlines no later than t and G_2 contains all those tasks with release time no earlier than t. (Note that $G_1 \supseteq \{i | i \in R^{(1)} \text{ and } d_i < C_{i^*}\}$ and $G_2 \supseteq F$.) Since i^* is completed after t, it is clear that $\sum_{i \in G_1} p_i + p_{i^*} > t \geq \max\{d_i | i \in G_1\}$. Thus, it is impossible to complete i^* by $\max\{d_i | i \in G_1\}$ in any schedule whatsoever, contradicting the fact that i^* is an eligible task. □

We note that the proof of the lemma remains valid for any set of *remaining* tasks at any decision point t, where the set of remaining tasks at t is defined to be all the available tasks at t and all the unreleased tasks. The processing time of an available task is the remaining processing time of the task at t and its release time is t. Thus, $r^{(1)}$ is t and $R^{(1)}$ consists of all the available tasks at t. With the use of Lemma 1, we can prove that the generalized Baker's rule yields an optimal schedule for a set of tasks with no obstruction, by showing that the tasks chosen by the algorithm at each decision point satisfy the conditions of the lemma.

THEOREM 1. *The generalized Baker's rule yields an optimal schedule for a set of tasks with no obstruction.*

Proof. An easy induction on the number of decision points shows that if a set of tasks has no obstruction, then every set of remaining tasks at every decision point of the generalized Baker's rule also has no obstruction. We assert that the tasks chosen by the generalized Baker's rule at each decision point satisfy the conditions of Lemma 1. With this assertion the theorem is proved by a simple induction on the number of decision points.

Let i^* be the task chosen by the generalized Baker's rule at a decision point. It is clear that i^* is a smallest eligible task among all the tasks in $R^{(1)}$. Thus, the first condition of Lemma 1 is satisfied. Because of the tie-breaking rule used by the generalized Baker's rule, we have $p_{i^*} < p_i$ for any $i \in R^{(1)}$ with $d_i < d_{i^*}$. We claim that $p_{i^*} \leq p_{i_j}$ whenever there is a skewed sequence $(i_1, i_2, \ldots, i_j)$ from a task $i_1 \in R^{(1)}$ to a task i_j such that $d_{i_{j-1}} < d_{i^*}$. Otherwise, let $(i_1, i_2, \ldots, i_j)$ be the shortest skewed sequence violating the claim; i.e., $p_{i^*} \leq p_{i_k}$ for $k = 1, 2, \ldots, j-1$, but $p_{i^*} > p_{i_j}$. Then we have $p_{i_{j-1}} \geq p_{i^*} > p_{i_j}$, and hence the ordered triple (i_{j-1}, i_j, i^*) constitutes an obstruction, contradicting the fact that the set of tasks has no obstruction. This proves that the second condition of Lemma 1 is also satisfied. □

We now consider the generalized Smith's rule. The next lemma is the counterpart of Lemma 1 for proving the optimality of the generalized Smith's rule. Before we give the lemma, we need to introduce the following notation. For a given schedule S, let $\rho_i(t)$ denote the remaining processing time of task i at time t in S; if $t < r_i$, $\rho_i(t)$ is defined to be p_i.

LEMMA 2. *There is an optimal schedule in which the task i^* is completed last among all the tasks in $D^{(m)}$ if* (1) $p_{i^*} \geq \rho_j(r_{i^*})$ *in any feasible schedule for any* $j \in D^{(m)}$ *with* $r_j \leq r_{i^*}$, (2) *there is a feasible schedule in which* $\rho_{i^*}(r_j) \geq p_j$ *for any* $j \in D^{(m)}$ *with* $r_j > r_{i^*}$, *and* (3) $p_{i_1} \leq p_{i_j}$ *whenever there is a skewed sequence* $(i_1, i_2, \ldots, i_j)$ *from a task* i_1 *to a task* $i_j \in D^{(m)}$ *such that* $r_{i^*} < r_{i_2}$.

Proof. Let S be an optimal schedule in which i^* is not completed last among all the tasks in $D^{(m)}$. We assert that there is always a task k in S with k completed after i^* such that interchanging the order of completion times of k with i^* yields another optimal schedule. With the above assertion a simple induction will show that S can be transformed into another optimal schedule in which i^* is completed last among all the tasks in $D^{(m)}$.

We will prove this by showing that there is a task k in S with k completed after i^*, and a time $t \geq r_k, r_{i^*}$, such that $\rho_k(t) \leq \rho_{i^*}(t)$. It is clear that if k satisfies the above condition, interchanging the order of completion times of k with i^* yields another optimal schedule. In the following we shall show that such a task exists.

Before we show that task k exists, we note that S satisfies the following property:

(P_1) If task i is completed prior to task j, then j is not processed in the interval $[r_i, C_i]$ in S.

It follows from the first two conditions of the lemma that $\rho_{i^*}(r_{i^*}) \geq \rho_j(r_{i^*})$ for any $j \in D^{(m)}$. Suppose the inequality holds until the time $t \geq r_{i^*}$, but not after t. That is, $\rho_{i^*}(t) \geq \rho_j(t)$ for any $j \in D^{(m)}$, but $\rho_{i^*}(t + \delta) < \rho_j(t + \delta)$ for some $j \in D^{(m)}$ and some $\delta > 0$. Note that t must exist since i^* is not completed last in S; in fact, $r_{i^*} \leq t < C_{i^*}$. Moreover, from the definition of t, i^* must be processed in S in the interval $[t, t + \delta]$.

Consider the partial schedule of S after t. Let E be the set of all tasks completed after t, except i^*; i.e., the tasks completed after t are exactly those in the set $E \cup \{i^*\}$. Partition E into two sets $\hat{F}$ and F such that $\hat{F}$ is the set of all tasks with deadlines earlier than $d^{(m)}$ and F is the set of all tasks with deadlines exactly $d^{(m)}$. Clearly, F is nonempty since the task completing last in S is in F.

First, consider the case $t \geq \min\{r_i | i \in F\}$. We claim that the task $k \in F$ with $r_k = \min\{r_i | i \in F\}$ is the desired task to be interchanged with i^*. By definition of t, $\rho_k(t) \leq \rho_{i^*}(t)$. Since i^* is processed in the interval $[t, t + \delta]$ and since S satisfies property P_1, we must have k completed after i^*. Thus, k is the desired task.

Now consider the case $t < \min\{r_i | i \in F\}$. In this case we claim that $\hat{F}$ is nonempty and that there is some task i in $\hat{F}$ with $r_i \leq t$. If $\hat{F}$ were empty, then i^* would be the only available task at t. This implies that the largest remaining processing time of i^* at $t + \delta$ is at most $\rho_{i^*}(t + \delta)$ in any schedule whatsoever. Since $\rho_{i^*}(t + \delta) < \rho_j(t + \delta) = p_j$ for some $j \in D^{(m)}$ (in fact, j must be a task in F), the second condition of the lemma would be violated. Thus, $\hat{F}$ must be nonempty. If $\min\{r_i | i \in \hat{F}\} > t$, then i^* would be the only available task at t, and by the same reasoning as before, the second condition of the lemma would be violated. Thus, there must be a task i in $\hat{F}$ with $r_i \leq t$.

We assert that there is a skewed sequence $(i_1, i_2, \ldots, i_j)$ from a task $i_1 \in \hat{F}$ with $r_{i_1} \leq t$ to a task $i_j \in F$ such that $r_{i_2} > t \geq r_{i^*}$. Suppose to the contrary that no such skewed sequence exists. Then, there must be a time t', $t + \delta < \max\{d_i | i \in \hat{F} \text{ and } r_i \leq t\} \leq t' \leq \min\{r_i | i \in F\}$, such that no task in the set $\hat{F} \cup F$ has release time earlier than t' and deadline later than t'. The set $\hat{F} \cup F$ can be partitioned into two sets G_1 and G_2 such that G_1 contains all those tasks with deadlines no later than t' and G_2 contains all those tasks with release times no earlier than t'. (Note that $G_1 \supseteq \{i | i \in \hat{F} \text{ and } r_i \leq t\}$ and $G_2 \supseteq F$.) Since the tasks in G_2 cannot start prior to t' and since $t + \delta < t' \leq \min\{r_i | i \in F\}$, it is clear that the largest remaining processing time of i^* at $\min\{r_i | i \in F\}$ must be no larger than $\rho_{i^*}(t + \delta)$ in any schedule whatsoever. Since $\rho_{i^*}(t + \delta) < \rho_j(t + \delta) = p_j$ for some $j \in D^{(m)}$ (in fact, j must be a task in F), the second condition of the lemma must be violated. Thus, there must be a skewed sequence $(i_1, i_2, \ldots, i_j)$ from a task $i_1 \in \hat{F}$ with $r_{i_1} \leq t$ to a task $i_j \in F$ such that $r_{i_2} > t$.

Since there is a skewed sequence from i_1 to i_j, by the third condition of the lemma, we have $p_{i_1} \leq p_{i_j}$. Thus, $\rho_{i_1}(t) \leq p_{i_1} \leq p_{i_j} = \rho_{i_1}(t) \leq \rho_{i^*}(t)$. Since i^* is processed in the interval $[t, t + \delta]$ and since S satisfies property P_1, we must have i_1 completed after i^*. Hence i_1 is the desired k to be interchanged with i^*. □

With the use of Lemma 2, we can prove that the generalized Smith's rule yields an optimal schedule for a set of tasks with no obstruction, by showing that the tasks chosen to complete last by the algorithm at each iteration satisfy the conditions of the lemma.

THEOREM 2. *The generalized Smith's rule yields an optimal schedule for a set of tasks with no obstruction.*

Proof. An easy induction on the number of tasks in a block shows that if a set of tasks has no obstruction, then the blocks obtained at each iteration of the generalized Smith's rule also have no obstruction. We assert that the tasks chosen to complete last by the generalized Smith's rule at each iteration satisfy the conditions of Lemma 2. With this assertion the theorem is proved by a simple induction on the number of tasks in a block.

Let i^* be the task chosen to complete last by the generalized Smith's rule at some iteration. It is not difficult to verify that i^* satisfies the first two conditions of Lemma 2. Because of the tie-breaking rule used by the generalized Smith's rule, we have $p_{i^*} > p_j$ for any $j \in D^{(m)}$ with $r_j > r_{i^*}$. We claim that $p_{i_1} \leq p_{i_j}$ whenever there is a skewed sequence $(i_1, i_2, \ldots, i_j)$ from a task i_1 to a task $i_j \in D^{(m)}$ such that $r_{i^*} < r_{i_2}$. Otherwise, let $(i_1, i_2, \ldots, i_j)$ be the shortest skewed sequence violating the claim; i.e., $p_{i_j} \geq p_{i_k}$ for $k = j-1, j-2, \ldots, 2$, but $p_{i_j} < p_{i_1}$. Then we have $p_{i_1} > p_{i_j} \geq p_{i_2}$ and $p_{i^*} > p_{i_j} \geq p_{i_2}$. Hence the ordered triple (i_1, i_2, i^*) constitutes an obstruction, contradicting the fact that the set of tasks has no obstruction. This shows that the third condition of Lemma 2 is also satisfied. □

6. Recognition of Obstructions

We now consider how to recognize whether a set of tasks contains skewed pairs, covered skewed pairs, or obstructions. We shall give a sequence of algorithms to recognize whether a set of tasks contains skewed pairs, covered skewed pairs, or obstructions.

Let $r^{(1)} < r^{(2)} < \cdots < r^{(l)}$ be the distinct release times of the tasks, and let $R^{(1)}, R^{(2)}, \ldots, R^{(l)}$ be the subsets of tasks with those release times.

Recognition of Skewed Pairs.
$A := R^{(1)}$;
for $h = 2, \ldots, l$
    $A := A - \{i | d_i \leq r^{(h)}\}$;
    if $\min\{d_i | i \in A\} < \max\{d_j | j \in R^{(h)}\}$
        then stop; (a skewed pair has been found)
    $A := A \cup R^{(h)}$;
endfor;

We assert that if any skewed pairs exist, the algorithm will find at least one such pair. In the *for-loop* of the algorithm, the set A contains all those tasks with release time earlier than $r^{(h)}$ and deadline later than $r^{(h)}$. Thus, if there is a task i in A and a task j in $R^{(h)}$ such that $d_i < d_j$, then (i, j) constitutes a skewed pair. Such a pair exists if and only if $\min\{d_i | i \in A\} < \max\{d_j | j \in R^{(h)}\}$.

The above algorithm can be implemented to run in $O(n^2)$ time. With an initial sort of the tasks by release times, the sets $R^{(1)}, R^{(2)}, \ldots, R^{(l)}$ can be obtained in $O(n \log n)$ time. The tasks in A and $R^{(h)}$, $h = 1, 2, \ldots, l$, will be maintained in nondescending order of deadlines. With this setup each iteration of the *for-loop* can be implemented to run in linear time. Since there are at most n release times, the overall running time of the algorithm is $O(n^2)$. Note that recognition of skewed pairs can actually be done in $O(n \log n)$ time; we give a slower algorithm here for the purpose of illustrating later algorithms.

RECOGNITION OF COVERED SKEWED PAIRS.
$A := R^{(1)}$;
for $h = 2, \ldots, l$
  $A := A - \{i | d_i \leq r^{(h)}\}$;
  if $\min\{d_i | i \in A\} < \max\{d_j | j \in R^{(h)}\}$ and
      $\min\{d_i | i \in A\} < \max\{d_k | k \in A\}$
    then stop; (a covered skewed pair has been found)
  $A := A \cup R^{(h)}$;
endfor;

We assert that if any covered skewed pairs exist, the algorithm will find at least one such pair. Note that the algorithm is identical to the first one, except that one more condition is added to the *if statement*, namely $\min\{d_i | i \in A\} < \max\{d_k | k \in A\}$. The added condition is to ensure that there is a task k in A covering the skewed pair (i, j).

Using the same data structure as the first one, the above algorithm also admits an $O(n^2)$ implementation.

RECOGNITION OF OBSTRUCTIONS.
$A := R^{(1)}$;
for $h = 2, \ldots, l$
  $A := A - \{i | d_i \leq r^{(h)}\}$;
  for each $i \in A$
    if there is $j \in R^{(h)}$ and $k \in A$ such that
        $d_i < d_k, d_j$ and $p_j < p_i, p_k$
      then stop; (an obstruction has been found)
  endfor;
  $A := A \cup R^{(h)}$;
endfor;

The validity of the algorithm follows directly from the definition of obstructions. We wish to show that the algorithm can be implemented to run in $O(n^2)$ time. As before the tasks in A and $R^{(h)}$, $h = 1, 2, \ldots, l$, are maintained in nondescending order of deadlines. To each task i in A we

attach a positive number $b_i = \max\{p_k | k = i \text{ or } k \text{ follows } i \text{ in } A\}$. Similarly, we attach a positive number $s_j = \min\{p_k | k = j \text{ or } k \text{ follows } j \text{ in } R^{(h)}\}$ to each task j in $R^{(h)}$, $h = 1, 2, \ldots, l$. It is clear that these numbers can be computed in linear time for any given A or $R^{(h)}$.

To show that the algorithm admits an $O(n^2)$ implementation, all we need show is that the innermost *for loop* can be implemented to run in linear time. They are implemented as follows: The sets A and $R^{(h)}$ will be scanned in nondescending order of deadlines. For each i in A, we locate the first j in $R^{(h)}$ such that $d_i < d_j$. Then we find the first k that follows i with $d_i < d_k$. If $s_j < p_i, b_k$, we have found an obstruction. Otherwise, there can be no obstruction that includes i; in this case we try the next i in A. Note that for the next i, we do not need to start from the beginning of $R^{(h)}$ to locate the j nor do we need to start from the beginning of A to locate the k; we can simply start from where they were left off previously. (Thus, the sets A and $R^{(h)}$ will be scanned only once.) The above process is repeated until either A or $R^{(h)}$ is exhausted, or an obstruction has been found. Clearly, this operation takes time that is linear to the total number of elements in A and $R^{(h)}$.

7. Implementations of the Generalized Rules

We have noted that there is a straightforward $O(n^3 \log n)$ implementation of the generalized Baker's rule. The bottleneck in this implementation is the $O(n \log n)$ EDD computation that is performed; each time one must test whether it is feasible to complete a given available task prior to the completion of the other available tasks. In order to improve on the time bound we need a more efficient method for testing feasibility. In the following we shall show that the smallest eligible task can be determined in linear time at each decision point, yielding an $O(n^2)$ implementation of the generalized Baker's rule. We remind the reader that we assume that the tasks form a block and that the earliest release time of the tasks is 0.

Let $d^{(1)} < d^{(2)} < \cdots < d^{(l)}$ be the distinct deadlines of the tasks, and let $P[\ ]$ be an array, where $P[i] = \sum_{j:\ d_j \le d^{(i)}} p_j$, $i = 1, 2, \ldots, l$. Since the tasks are feasible, we have $P[i] \le d^{(i)}$ for $i = 1, 2, \ldots, l$.

Consider how the array $P[\ ]$ can be used for the generalized Baker's rule. At any given decision point let A denote the set of available tasks, and let $ad^{(1)} < ad^{(2)} < \cdots < ad^{(m)}$ be the distinct deadlines of the tasks in A. Let $i_{(j)}$, $j = 1, 2, \ldots, m$, be the smallest task in A with deadline $ad^{(j)}$. It is clear that the smallest eligible task must be one of the tasks in $\{i_{(1)}, i_{(2)}, \ldots, i_{(m)}\}$. Furthermore, $i_{(j)}$, $j = 1, 2, \ldots, m$, is eligible if and only if there is a feasible schedule in which $i_{(j)}$ is completed by $ad^{(1)}$. We assert

that $i_{(j)}$ can be completed by $ad^{(1)}$ if and only if

$$P[k] + p_{i_{(j)}} \leq d^{(k)}, \tag{7.1}$$

for each k such that $ad^{(1)} \leq d^{(k)} < ad^{(j)}$.

We begin with the deadline $ad^{(m)}$ and the task $i_{(m)}$ as a candidate of the smallest eligible task. With the use of (7.1) we can efficiently test whether $i_{(m)}$ can be completed by $ad^{(m-1)}$; it can if and only if condition (7.1) is satisfied for each k such that $ad^{(m-1)} \leq d^{(k)} < ad^{(m)}$. We assert that $i_{(m)}$ should be chosen over $i_{(m-1)}$ as a candidate of the smallest eligible task if and only if $p_{i_{(m)}} < p_{i_{(m-1)}}$ and $i_{(m)}$ can be completed by $ad^{(m-1)}$. This follows from the observation that if $p_{i_{(m)}} < p_{i_{(m-1)}}$ and $i_{(m)}$ can be completed by $ad^{(m-1)}$, then $i_{(m-1)}$ can be an eligible task only if $i_{(m)}$ can.

The preceding discussions suggest a linear search for the smallest eligible task. Starting with the deadline $ad^{(m)}$ and the task $i_{(m)}$ as a candidate, we repeatedly screen out the candidates $i_{(m)}, i_{(m-1)}, \ldots, i_{(1)}$, until we reach the deadline $ad^{(1)}$. The winner found at $ad^{(1)}$ is the desired task.

The following fragment of code shows how the smallest eligible task can be chosen in linear time at each decision point. (Note. "$\arg\min\{a_i | i \in S\}$" denotes the index $j \in S$ such that $a_j = \min\{a_i | i \in S\}$.) The variable j keeps track of the candidate. The variable *flag* is used to signal that the current candidate cannot meet the next deadline $ad^{(g)}$; i.e., *flag* is *true* if the current candidate cannot meet some intermediate deadline between $ad^{(g)}$ and $ad^{(g+1)}$:

```
x := index i such that d^(i) = ad^(1);
y := index i such that d^(i) = ad^(m);
j := arg min{p_i | i ∈ A and d_i = d^(y)};
flag := false;
for h = y − 1, y − 2, ..., x
    if P[h] + p_j > d^(h)
        then flag := true;
    if d^(h) = ad^(g) for some g
        then {k := arg min{p_i | i ∈ A and d_i = d^(h)};
              if flag or p_k ≤ p_j then j := k;
              flag := false;
             };
endfor;
```

After j is selected, the array $P[\ ]$ is revised by adding β to each $P[i]$ such that $d^{(i)} < d_j$, where β is the amount of j processed between the current and the next decision points.

We now consider efficient implementation of the generalized Smith's rule. We have noted that there is an $O(n^2 \log n)$ implementation of the generalized Smith's rule. The bottleneck in this implementation is the $O(n \log n)$ EDD computation that is performed, each time a task is chosen to complete last. In the following we shall show that this choice can be done in linear time, without constructing the EDD schedule. Furthermore, the remaining tasks can be decomposed into blocks in linear time also, yielding an $O(n^2)$ implementation of the generalized Smith's rule. Let $r^{(1)} < r^{(2)} < \cdots < r^{(l)}$ be the distinct release times of the tasks, and let $Q[\ ]$ be an array where $Q[i] = \sum_{j:\, r_j \le r^{(i)}} p_j$, $i = 1, 2, \ldots, l$.

Consider how the array $Q[\ ]$ can be used for the generalized Smith's rule. Let B be the set of tasks with the latest deadline, and let $br^{(1)} < br^{(2)} < \cdots < br^{(m)}$ be the distinct release times of the tasks in B. Let $k_{(j)}$, $j = 1, 2, \ldots, m$, be the largest task in B with release time $br^{(j)}$. It is clear that the last task to complete is one of the tasks in $\{k_{(1)}, k_{(2)}, \ldots, k_{(m)}\}$.

A linear search of $k_{(1)}, k_{(2)}, \ldots, k_{(m)}$, is performed in order to search for the last task to complete. We begin with the release time $br^{(1)}$ and the task $k_{(1)}$ as a candidate. With the help of the array $Q[\ ]$, we can compute the largest possible remaining processing time of $k_{(1)}$ at the release time $br^{(2)}$, say β. Task $k_{(1)}$ will be chosen over $k_{(2)}$ if and only if β is larger than $p_{k_{(2)}}$. The above process is repeated until we reach $br^{(m)}$, at which time the winner will be chosen to complete last.

The following fragment of code shows how the last task to complete can be chosen in linear time. (Note. "$\arg\max\{a_i | i \in S\}$" denotes the index $j \in S$ such that $a_j = \max\{a_i | i \in S\}$.) The variable j keeps track of the candidate and β keeps track of the remaining processing time of j:

```
x := index i such that r^(i) = br^(1);
y := index i such that r^(i) = br^(m);
j := arg max{p_i | i ∈ B and r_i = r^(x)};
β := p_j;
for h = x + 1, x + 2, ..., y
    β := β − max{0, r^(h) − (Q[h − 1] − β)};
    if r^(h) = br^(g) for some g
        then {k := arg max{p_i | i ∈ B and r_i = r^(h)};
              if β ≤ p_k
                  then {j := k;
                        β := p_k;
                       };
              };
endfor;
```

After j is selected, we need to decompose the remaining tasks into blocks. First, the array $Q[\ ]$ is revised by subtracting p_j from each $Q[i]$

such that $r^{(i)} \geq r_j$. From the revised array, we can determine the periods of idle time in the EDD schedule for the remaining tasks as follows. Let c be a variable that keeps track of the total idle time, with c initialized to 0 at the beginning. We scan the (distinct) release times in ascending order until we find a pair of consecutive release times $r^{(i)}$ and $r^{(i+1)}$ such that $r^{(i)} \leq Q[i] + c < r^{(i+1)}$. It is clear that there is a period of idle time between $Q[i] + c$ and $r^{(i+1)}$. We update c by adding $r^{(i+1)} - (Q[i] + c)$ to it, and the above process is repeated until the latest release time $r^{(l)}$ is reached.

8. NP-Hardness Proof

We demonstrate NP-hardness of the $1|pmtn, r_j, d_j|\Sigma C_j$ problem by transformation of the known NP-complete ODD–EVEN PARTITION problem [4, 5] to the decision version of the scheduling problem.

ODD–EVEN PARTITION. Given $2n$ positive integers $a_1, a_2, \ldots, a_{2n}$, where $a_j > a_{j+1}$ for $j = 1, 2, \ldots, 2n - 1$, is there a partition of the integers into two sets, A_1 and A_2, such that

$$\sum_{j \in A_1} a_j = \sum_{j \in A_2} a_j,$$

where A_1 and A_2 each contain exactly one element of each odd–even pair $\{a_{2i-1}, a_{2i}\}$, $i = 1, 2, \ldots, n$?

Without loss of generality, we shall assume that the integers a_j are such that

$$a_{2i} > \sum_{j=2i+1}^{2n} a_j, \qquad i = 1, 2, \ldots, n. \tag{8.1}$$

If a given instance of ODD–EVEN PARTITION does not satisfy this condition, we can easily modify it, by adding constants to the odd–even pairs, so that it does:

for $i = n - 1, n - 2, \ldots, 1$

$$a_{2i} := a_{2i} + \sum_{j=2i+1}^{2n} a_j;$$

$$a_{2i-1} := a_{2i-1} + \sum_{j=2i+1}^{2n} a_j;$$

endfor;

It is easily seen that this transformation can be performed in polynomial time and does not affect the problem in any significant way.

The decision version of the $1|pmtn, r_j, d_j|\Sigma C_j$ scheduling problem is as follows:

$1|pmtn, r_j, d_j|\Sigma C_j$. Given a single positive integer W and n triples of nonnegative integers (p_j, r_j, d_j), $j = 1, 2, \ldots, n$, where each triple represents the processing time, release time, deadline of a task, is there a feasible preemptive schedule such that ΣC_j does not exceed W?

For a given instance of ODD–EVEN PARTITION, let $A = \frac{1}{2}\Sigma_{j=1}^{2n} a_j$, and let E and L be suitably large integers; $E \geq nA$ and $L \geq (n+1)E$ will do. We create an instance of the scheduling problem with $3n + 1$ tasks: $2n$ *partition* tasks, n *enforcer* tasks, and a single *slack* task:

Partition task P_j, $j = 1, 2, \ldots, 2n$, has processing time a_j and a very large deadline, say $2L$. Each "odd" task P_{2i-1} has release time $(i-1)E$ and each "even" task P_{2i} has release time $(i-1)E + (n-i+4)(a_{2i-1} - a_{2i})$.

Enforcer task Q_i, $i = 1, 2, \ldots, n$, has processing time $E - a_{2i-1}$, release time $(i-1)E$ and deadline iE.

The slack task S has processing time L, release time 0, and deadline

$$L + nE - (1/2)\sum_{i=1}^{n}(a_{2i-1} - a_{2i}).$$

The task intervals for the $3n + 1$ tasks are indicated schematically in Fig. 2. Note that the set of tasks consists of the union of two sets, each of which is nested.

It is possible to complete either one (but not both) of the partition tasks P_{2i-1} and P_{2i} in the interval $[(i-1)E, iE]$. Because of condition (8.1), it is impossible to complete both of the partition tasks P_{2i-1} and P_{2i} prior to the completion of the slack task S.

With respect to a given feasible schedule call those partition tasks that are finished prior to the completion of the slack task "initial" tasks and the other partition tasks "final" tasks. We have seen that there is no feasible schedule with more than n initial tasks, with at least one from each odd–even pair. Let I be a subset of the partition tasks, with exactly one task from each odd–even pair. We assert that there is a feasible schedule in which all the tasks in I are initial, if and only if the total processing time of the tasks in I does not exceed A. This follows from the choice of the processing time and deadline of the slack task.

Because L has been chosen to be such a large number, and the slack task S must therefore finish at such a late time, a schedule cannot be optimal unless it has exactly n initial tasks. Furthermore, a schedule

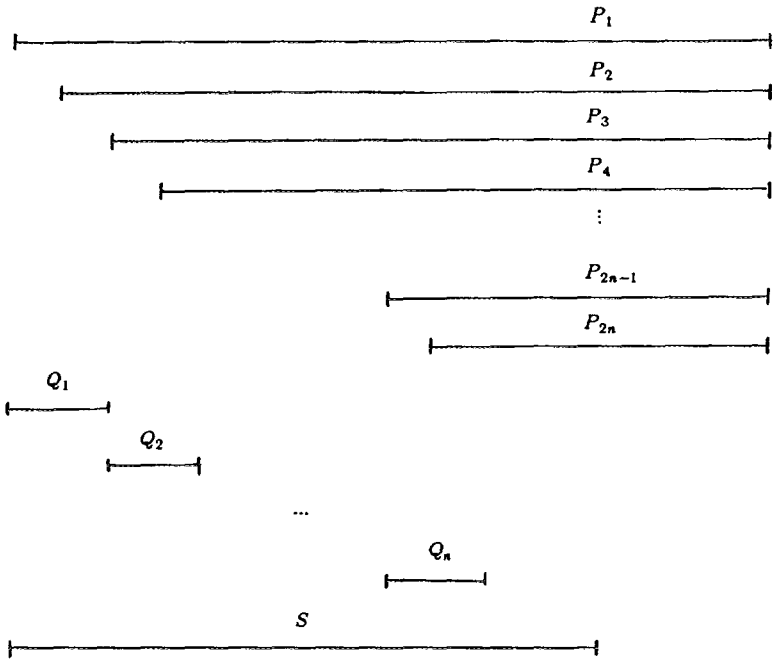

FIG. 2. Release time and deadline pattern of the tasks in the reduction.

cannot be optimal unless its n final tasks are completed in increasing order of their processing times.

It follows from the above observations that an optimal schedule can be found by optimizing over all possible choice of n initial tasks, exactly one from each odd–even pair, and with total processing time not exceeding A. Given any such choice of initial tasks, the completion times of all $3n + 1$ tasks are completely determined: Each initial task is completed prior to its corresponding enforcer task, all initial tasks and all enforcer tasks are completed prior to the slack task, and all final tasks are completed after the slack task in increasing order of processing time. It follows that the EDD rule can be applied with respect to this ordering of completion times to find a schedule that minimizes ΣC_j with respect to the given choice of initial tasks.

As an example, suppose $n = 4$, P_1, P_4, P_5, and P_8 are initial tasks, and P_2, P_3, P_6, and P_7 are final tasks. Then an optimal schedule for this choice of initial tasks takes the form shown in Fig. 3.

Note that in an optimal schedule a final task is not started until after the slack task is completed. All partition tasks are scheduled without preemption; only the enforcer tasks and the slack task are preempted.

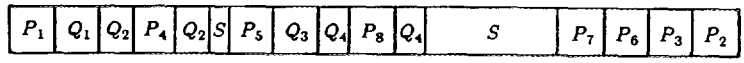

FIG. 3. An example schedule.

We now want to show that ΣC_j is minimized by maximizing the total processing time of the initial tasks. We have chosen the release time of P_{2i} to be larger than the release time of P_{2i-1} in order to penalize the choice of the (smaller) even task of each odd–even pair. In fact, as we shall show, the release times have been chosen in such a way that the net effect of choosing P_{2i} instead of P_{2i-1} is to make ΣC_j larger by exactly $a_{2i-1} - a_{2i}$.

Preliminary to a proof of the preceding assertion, first consider how to compute the total completion time of the final tasks. Let $P^{(i)}$ be the final task chosen from the pair P_{2i-1} and P_{2i}, and let $a^{(i)}$ be its processing time. Let C_S denote the completion time of the slack task. Then the completion time of $P^{(i)}$ is $C_S + a^{(n)} + a^{(n-1)} + \cdots + a^{(i)}$. Hence the total completion time of the final tasks is

$$\begin{aligned}&\left(C_S + a^{(n)}\right) + \left(C_S + a^{(n)} + a^{(n-1)}\right) + \cdots \\ &\qquad + \left(C_S + a^{(n)} + a^{(n-1)} + \cdots + a^{(1)}\right) \\ &= nC_S + \sum_{i=1}^{n} ia^{(i)}. \end{aligned} \tag{8.2}$$

Now let us analyze the effect of choosing P_{2i} as an initial task, instead of P_{2i-1}:

(1) P_{2i} is completed at time $(i-1)E + (n-i+4)(a_{2i-1} - a_{2i}) + a_{2i}$, whereas if P_{2i-1} had been chosen as an initial task it would have been completed at time $(i-1)E + a_{2i-1}$. This has the net effect of increasing ΣC_j by $(n-i+3)(a_{2i-1} - a_{2i})$.

(2) Enforcer task Q_i is completed at $iE - (a_{2i-1} - a_{2i})$, instead of iE. This has the net effect of decreasing ΣC_j by $a_{2i-1} - a_{2i}$.

(3) The completion times of the other initial tasks and enforcer tasks are independent of the choice of P_{2i-1} or P_{2i} as an initial task.

(4) The slack task S is completed earlier, by an amount equal to $a_{2i-1} - a_{2i}$. Hence ΣC_j is decreased by $a_{2i-1} - a_{2i}$.

(5) Taking into account the term nC_S in (8.2), the net effect on ΣC_j is to decrease this sum by $(n-i)(a_{2i-1} - a_{2i})$.

A bit of algebra shows that the net effect of choosing P_{2i} is to increase ΣC_j by exactly $a_{2i-1} - a_{2i}$.

We have thus shown that an optimal schedule results from a choice of n initial tasks, exactly one from each odd–even pair, such that the total processing time is maximized, subject to the constraint that it does not exceed A. To complete our transformation from ODD–EVEN PARTITION, we only need to compute ΣC_j for an initial set of tasks such that its total processing time is exactly A. This will give us the threshold W for the decision version of the $1|pmtn, r_j, d_j|\Sigma C_j$ problem.

In order to make our task easier, we suppose that the deadline of the slack task is increased, so that it is feasible to choose all the odd tasks as an initial set. We will then compute ΣC_j for this choice of initial tasks and add $\Sigma_{i=1}^{n} a_{2i-1} - A$ to this result, in order to obtain W. For such a schedule:

(1) The total completion time of the initial tasks is $n(n-1)E/2 + \Sigma_{i=1}^{n} a_{2i-1}$.

(2) The total completion time of the enforcer tasks is $n(n+1)E/2$.

(3) The slack task is completed at time $L + nE$.

(4) The total completion time of the final tasks is $n(L+nE) + \Sigma_{i=1}^{n} i a_{2i}$.

Adding these completion times, we obtain

$$n(2n+1)E + (n+1)L + \sum_{i=1}^{n} a_{2i-1} + \sum_{i=1}^{n} i a_{2i}.$$

Adding $\Sigma_{i=1}^{n} a_{2i-1} - A$ to this result, we obtain

$$W = n(2n+1)E + (n+1)L + 2\sum_{i=1}^{n} a_{2i-1} + \sum_{i=1}^{n} i a_{2i} - A.$$

This concludes the proof of NP-hardness. □

9. Conclusion

In this paper we have shown that $1|pmtn, r_j, d_j|\Sigma C_j$ is NP-hard in the ordinary sense. Can $1|pmtn, r_j, d_j|\Sigma C_j$ be shown to be strongly NP-hard, or does it admit a pseudopolynomial time algorithm? For multiprocessor systems, Du, Leung, and Young [3] have recently shown that $P2|pmtn, r_j|\Sigma C_j$ is NP-hard in the ordinary sense. In spite of extensive research effort, we have not been able to determine the complexity of $P2|pmtn, d_j|\Sigma C_j$. For future research, we think it is worthwhile to settle this open question.

We have generalized Baker's algorithm and Smith's algorithm and show that they can solve the class of problem instances with no obstruction. Is there any other interesting class of problem instances that can be solved by either of the generalized algorithms? Better yet, is it possible to characterize the *exact* class solved by either of the generalized algorithms? Note that in practice, we should use a composite algorithm that consists of running both algorithms and chooses the better of the two schedules. Since the two classes are incomparable (i.e., neither is a subset of the other), the composite algorithm solves a broader class than each of them alone.

A problem instance with no obstruction has the following interesting property: There is an optimal schedule in which the number of tasks completed by each instant of time is no less than that in any other feasible schedule. In fact, both of the generalized algorithms produce such an optimal schedule. Note that not all problem instances have this property; the one given in Fig. 1 does not. We conjecture that the class of problem instances with the above stated property can be solved in polynomial time, possibly by the generalized Baker's rule or the generalized Smith's rule. It will be interesting to characterize this class and give an optimal algorithm for it.

ACKNOWLEDGMENT

We gratefully acknowledge the extensive suggestions offered by one of the referees, which have led to a great improvement in presentation over our earlier version.

REFERENCES

1. K. R. BAKER, "Introduction to Sequencing and Scheduling," Wiley, New York, 1974.
2. K. R. BAKER, E. L. LAWLER, J. K. LENSTRA, AND A. H. G. RINNOOY KAN, Preemptive scheduling of a single machine to minimize maximum cost subject to release dates and precedence constraints, *Oper. Res.* **31** (1983), 381–386.
3. J. DU, J. Y-T. LEUNG, AND G. H. YOUNG, Minimizing mean flow time with release time constraint, *Theor. Comput. Sci.* **75** (1990), 347–355.
4. M. R. GAREY AND D. S. JOHNSON, "Computers and Intractability: A Guide to the Theory of NP-Completeness," Freeman, San Francisco, 1979.
5. M. R. GAREY, R. E. TARJAN, AND G. T. WILFONG, One-processor scheduling with symmetric earliness and tardiness, *Math. Oper. Res.* **13** (1988), 330–348.
6. R. L. GRAHAM, E. L. LAWLER, J. K. LENSTRA, AND A. H. G. RINNOOY KAN, Optimization and approximation in deterministic sequencing and scheduling: A survey, *Ann. Discrete Math.* **5** (1979), 287–326.
7. W. A. HORN, Some simple scheduling algorithms, *Naval Res. Logist. Quart.* **21** (1974), 177–185.

8. J. LABETOULLE, E. L. LAWLER, J. K. LENSTRA, AND A. H. G. RINNOOY KAN, "Preemptive Scheduling of Uniform Machines Subject to Release Dates," Report BW 99, Mathematisch Centrum, Amsterdam, 1979.
9. E. L. LAWLER, Recent results in the theory of machine scheduling, in "Mathematical Programming: The State of the Art" (A. Bachem, M. Groschel, and B. Korte, Eds.), Springer-Verlag, New York/Berlin, 1982.
10. J. K. LENSTRA, "Sequencing by Enumerative Methods," Mathematical Centre Tracts 69, Mathematisch Centrum, Amsterdam, 1977.
11. W. E. SMITH, Various optimizers for single-stage production, *Naval Res. Logist. Quart.* **3** (1956), 59–66.