One-Processor Scheduling of Tasks with Preferred Starting Times

Michael R. Garey
Robert E. Tarjan
Gordon T. Wilfong

CS-TR-49 -86

One-Processor Scheduling of Tasks with Preferred Starting Times

Michael R. Garey

Robert E. Tarjan [†]

Gordon T. Wilfong

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

We consider a new class of one-processor scheduling problems having the following form: Tasks $T_1, T_2, \ldots, T_N$ are given, with each T_i having a specified length l_i and a preferred starting time p_i. The tasks are to be scheduled nonpreemptively (i.e., a task cannot be split) on a single processor as close to their preferred starting times as possible. We examine two different cost measures for such schedules, the sum of the individual discrepancies from the preferred starting times and the maximum such discrepancy. For the first of these, we show that the problem of finding minimum cost schedules is NP-complete; however, we give an efficient algorithm that finds minimum cost schedules whenever the tasks either all have the same length or are required to be executed in a given fixed sequence. For the second cost measure, we give an efficient algorithm that finds minimum cost schedules in general, with no constraints on the ordering or lengths of the tasks.

[†] Also Computer Science Department, Princeton University, Princeton, NJ 08544.

One-Processor Scheduling of Tasks with Preferred Starting Times

Michael R. Garey

Robert E. Tarjan †

Gordon T. Wilfong

AT&T Bell Laboratories
Murray Hill, NJ 07974

1. Introduction

Combinatorial scheduling theory has been of interest for almost three decades, and an impressive body of literature, encompassing a wide variety of problems and methods, now confronts any neophyte to the field. It is hard to imagine anyone posing a new type of scheduling problem that does more than merely combine aspects of different problems that were previously studied independently of one another. However, in this paper, we shall study exactly such a new class of problems, involving tasks that have "preferred" times at which one would like them to be executed.

The problems we shall be discussing all have the following general form: We are given N *tasks*, $T_1, T_2, \ldots, T_N$, each with a *length* $l_i \geq 0$ and a *preferred starting time* $p_i \geq 0$. It is desired that these tasks be scheduled on a single processor, without preemption (i.e., once started, a task is always executed to its completion l_i time units later), so that each is begun as close to its preferred starting time as possible. A *schedule* assigns to each task T_i a *starting time* s_i such that no two tasks overlap in their execution; i.e., the execution intervals $[s_i, s_i + l_i]$ and $[s_j, s_j + l_j]$ for any two tasks T_i and T_j can intersect only at their endpoints.

We shall consider two different ways of measuring how well the schedule meets the objective of starting tasks close to their preferred starting times. We define the *discrepancy* of a task T_i from its preferred starting time to be $|s_i - p_i|$. The first such measure is the *total discrepancy*, $\sum_{i=1}^{N} |s_i - p_i|$. The second measure is the *maximum discrepancy*, $\max_{1 \leq i \leq N} |s_i - p_i|$. We shall consider these two cost measures separately, in each case seeking schedules that make the cost as small as possible.

† Also Computer Science Department, Princeton University, Princeton, NJ 08544.

Observe that these problems could just as well be stated using "preferred completion times" or "preferred midpoint times," since it is easy to transform any of these variants into any of the others, so there will be no loss of generality in restricting our attention to preferred starting times.

Notice also that, although these problems bear superficial similarity to previously studied problems having specified latest starting or finishing times (deadlines), all the cost measures that have been considered for these older problems treat tasks that finish early more favorably than tasks that finish late (and often more favorably than those that finish on time). In contrast, the problems we consider regard finishing early to be just as undesirable as finishing late. This is natural, for example, in the scheduling of a sequence of experiments that depend on predetermined external events (such as the position of the sun) and in scheduling manufacturing process steps to coordinate with deliveries (seeking to achieve something similar to "just-in-time" inventory control, in cases where delivery times are less adjustable than process steps).

The only prior work of which we are aware that considers preferred starting times is that of Vere [5]. Indeed, Vere's work motivated ours. Vere's paper considers a more general class of multiprocessor scheduling problems and presents a two-part exponential time algorithm for solving them. The first part produces for each task an interval of time, called a *window*, during which it can be started, with the property that for any fixed choice of a starting time for the task within its window there exists a legal schedule for the remaining tasks. The window always contains the preferred starting time for the task. The second part of Vere's algorithm then fixes a schedule of starting times for each processor using the windows previously produced. However, the algorithm does not attempt to minimize distance from the preferred starting times to the actual starting times.

The remainder of our paper, all of which deals with scheduling on just a single processor (often called "sequencing"), is organized as follows. In Section 2, we examine the problem of minimizing the total discrepancy from preferred starting times. We first show that the corresponding decision problem is NP-complete and hence the optimization problem is unlikely to be solvable by a polynomial time algorithm. We then give an $O(N \log N)$-time algorithm that minimizes the total discrepancy when the tasks must be executed in a given fixed order, and we show that, when all the tasks have the same length, executing the tasks in order of their preferred starting times is always optimum. We also show that the algorithm generalizes easily to handle several types of additional constraints, including window constraints like those obtained from the first part of Vere's algorithm. In Section 3, we consider the problem of minimizing the maximum discrepancy of any task from its preferred starting time and give an efficient algorithm for solving this problem in general. The algorithm can also be viewed as solving a special case of the NP-complete problem [1] of sequencing arbitrary length tasks under specified release time and deadline constraints. We conclude in Section 4 with some brief final remarks.

2. Minimizing Total Discrepancy

In this section we shall show that the general problem of minimizing the total discrepancy is NP-complete, and we shall present an $O(N \log N)$-time algorithm that minimizes the total discrepancy whenever the ordering of the tasks is fixed in advance or the task lengths are all the same. Section 2.1 contains the NP-completeness proof for the general problem. Section 2.2 contains a description of our algorithm for the case of a fixed task ordering. Section 2.3 gives a proof that this algorithm produces a minimum cost schedule for a fixed task ordering. Section 2.4 describes an $O(N \log N)$-time implementation. Section 2.5 contains a proof that in the case of equal length tasks, scheduling the tasks in order of their preferred starting times is always optimum. Section 2.6 covers several generalizations of the algorithm.

2.1. The Total Discrepancy Problem is NP-Complete

In this section we show that the general problem of minimizing the total discrepancy is NP-complete. More precisely, we show that the decision problem, "Is there a schedule with total discrepancy no more than k?," is NP-complete (and hence the optimization problem is at least that "hard").

In order to simplify the presentation, it will be convenient to work with preferred midtimes for the tasks, rather than preferred starting times. (Following this section, we shall return to the use of preferred starting times for the remainder of the paper.) Thus for each task T_i we assume we are given a length l_i and a preferred midtime M_i. For any schedule S, let $m_i(S)$ denote the actual midtime for T_i in S, i.e., the time exactly half way between the starting and finishing times for T_i in S.

The decision problem that will be shown to be NP-complete is as follows.

Total Discrepancy: Given $N, k \in Z^+$ and M_i, $l_i \in Z^+$ $(1 \leq i \leq N)$, is there a schedule S such that $cost(S) = \sum_{i=1}^{N} | m_i(S) - M_i | \leq k$?

The total discrepancy problem will be shown to be NP-hard by reducing the following NP-complete problem, which is a version of partition [1], to it.

Even-Odd Partition: Given $n \in Z^+$ and a set $B = \{b_1, b_2, \cdots, b_{2n}\}$ of positive integers, where $b_i < b_{i+1}$ for $1 \leq i < 2n$, does there exist a partition of B into subsets B_1 and B_2 such that $\sum_{b \in B_1} b = \sum_{b \in B_2} b$ and such that for each i, $1 \leq i \leq n$, B_1 (and hence B_2) contains exactly one of $\{ b_{2i-1}, b_{2i}\}$?

We first show that the even-odd partition problem is NP-complete. Although this problem is mentioned in [1] and is generally known to be NP-complete, as far as we know no proof has appeared in the literature. We shall reduce the following standard version of the partition problem to it:

Partition: Given $n \in Z^+$ and a set $A = \{a_1, a_2, \cdots, a_n\}$ of positive integers, does there exist a partition of A into subsets A_1 and A_2 such that $\sum_{a \in A_1} a = \sum_{a \in A_2} a$?

Lemma 1. The even-odd partition problem is NP-complete.

Proof. Showing that the problem is in NP is straightforward. To show that it is NP-hard we reduce partition to it. That is, given an instance P of the partition problem we show how to construct an instance EO of the even-odd partition problem that has a solution if and only if P does. The construction of EO (and the size of EO) will be polynomial in the size of P.

Let P be an instance of the partition problem. That is, we are given n and the set $A = \{a_1, a_2, \cdots, a_n\}$, and we wish to determine the existence of the desired partition of A.

Consider the following instance EO of the even-odd partition problem. Without loss of generality we assume for $1 \leq i \leq n$, $a_i > 1$. We define the elements of the set B of EO recursively as follows:

$$b_1 = 1$$
$$b_{2i} = b_{2i-1} + a_i,\ 1 \leq i \leq n$$
$$b_{2i+1} = b_{2i} + 1,\ 1 \leq i < n$$

Clearly $b_i < b_{i+1}$ for $1 \leq i < 2n$. Suppose B_1 and B_2 form a partition of B such that for each i exactly one of b_{2i} and b_{2i-1} is in B_1 (and the other is in B_2). Let $\beta_j = \sum_{b_i \in B_j} b_i = \sum_{i=1}^{n} b_{2i-1} + \sum_{b_{2i} \in B_j} a_i$ for $j = 1,2$. Then $\beta_1 = \beta_2$ if and only if $\sum_{b_{2i} \in B_1} a_i = \sum_{b_{2i} \in B_2} a_i$. Therefore setting $A_j = \{a_i : b_{2i} \in B_j\}$ for $j = 1,2$ provides a partition of A that satisfies P if and only if B_j for $j = 1,2$ is an even-odd partition of B that satisfies EO. □

We wish to show that the total discrepancy problem is NP-complete by reducing the even-odd partition problem to it. Towards this end we first consider a special case of the total discrepancy problem in which we are given tasks $T_0, T_1, \cdots, T_{2n}$, a length l_i for each T_i, and a *single* preferred midtime M that applies for each of the tasks. We assume that M is large (e.g $M > \sum_{i=0}^{2n} l_i$) and that the tasks are ordered so that $0 < l_0 < l_1 < \cdots < l_{2n}$. The cost of a schedule S of these tasks will be $cost(S) = \sum_{i=0}^{2n} |M - m_i(S)|$.

We shall show that for problems of this type minimum cost schedules take on a very special form.

The first observation we make is that a minimum cost schedule S for such a set of tasks cannot have any gaps between tasks. Clearly if there were a gap in the schedule then the cost of the schedule could be reduced by closing the gap by rescheduling some of the tasks closer to M.

We define the sets $A(S) = \{T_i : m_i(S) < M\}$ and $B(S) = \{T_i : m_i(S) > M\}$. Notice that if $m_i(S) = M$ then T_i belongs neither to $A(S)$ nor to $B(S)$. The notation $|X|$ will be used to denote the cardinality of a set X.

It is easy to show that for any minimum cost schedule S if T_i and T_j are both in $A(S)$ and $l_i < l_j$ then $m_i(S) > m_j(S)$. Similarly, if T_i and T_j are both in $B(S)$ and $l_i < l_j$ then $m_i(S) < m_j(S)$. We say that a schedule is *ordered* if it has these properties. These observations follow by a simple swapping argument. Intuitively, the reason that minimum cost schedules are ordered is as follows. Suppose $T \in B(S)$. Then the length of T occurs as an additive term in the discrepancy for each task in $B(S)$ that is further from M than T. Thus the tasks that are closer to M contribute their lengths to the cost of the schedule the largest number of times, and hence it is beneficial to schedule the shorter tasks nearer to M.

We now show that another property of a minimum cost schedule S is that there is some task not in $A(S)$ or $B(S)$. That is, there is a task whose scheduled midtime is at M.

Lemma 2. If S is a minimum cost schedule then $m_i(S) = M$ for some i, $0 \leq i \leq 2n$.

Proof. Suppose $m_i(S) \neq M$ for all i, $0 \leq i \leq 2n$. Then $|A(S)| + |(B(S)| = 2n + 1$ and so $|A(S)| \neq |B(S)|$. Consider the case $|A(S)| < |B(S)|$.

Let $m_j(S) = \min\{m_i(S) : T_i \in B(S)\}$ and $k = m_j(S) - M$. Define R to be the schedule such that $m_i(R) = m_i(S) - k$. That is, R is obtained from S by rescheduling all tasks k units earlier. By the definition of k, it follows that $m_j(R) = M$. Then $cost(R) = cost(S) - |B(S)|k + |A(S)|k$, but since $|B(S)| > |A(S)|$ we can conclude that $cost(R) < cost(S)$. This contradicts the minimality of the cost of S.

The case where $|A(S)| > |B(S)|$ leads to a similar contradiction. It follows that there must be some i such that $m_i(S) = M$ whenever S is a minimum cost schedule. □

The next property we show for minimum cost schedules is that there must always be the same number of tasks with midtimes scheduled after M as with midtimes scheduled before M.

Lemma 3. If S is a minimum cost schedule then $|A(S)| = |B(S)|$.

Proof. By way of contradiction assume that $|A(S)| < |B(S)|$. The case where $|A(S)| > |B(S)|$ is similar.

By the previous result we know there is some i such that $m_i(S) = M$ (and hence T_i is in neither $A(S)$ nor $B(S)$). Therefore it must be that $|A(S)| + |B(S)| = 2n$ and so $|B(S)| \geq |A(S)| + 2$.

Define j, k and R as in the previous proof. Then $cost(R) = cost(S) - |B(S)|k + (|A(S)| + 1)k$ because the midpoints of the tasks in $B(S)$ are k units closer to M, the midpoints of the tasks in $A(S)$ are k units farther from M, and the midpoint of the task T_i that has its midpoint at M in S is also k units farther from M in R than in S. Thus

$$cost(R) \leq cost(S) + (|A(S)| + 1 - |A(S)| - 2)k = cost(S) - k < cost(S).$$

This contradicts the minimality of the cost of S and hence we conclude that $|A(S)| = |B(S)|$. □

We now know that every minimum cost schedule is ordered and has no gaps, that it has some task T_j scheduled with its midpoint at M, and that the remaining tasks are divided equally in number on either side of M. Therefore we can completely describe a minimum cost schedule by giving the order of the tasks in the schedule and indicating which task has its midtime at M. We use the notation

$$[A_n, A_{n-1}, \cdots A_1, T @ M, B_1, B_2, \cdots, B_n]$$

to denote the minimum cost schedule S that has the tasks scheduled in the order indicated and that has task T scheduled with its midtime at M.

We now show that the task with scheduled midpoint M must be T_0, the task with the shortest length.

Lemma 4. If S is a minimum cost schedule then $m_0(S) = M$.

Proof. By way of contradiction assume that $m_j(S) = M$ and $j \neq 0$. That is,

$$S = [A_n, A_{n-1}, \cdots A_1, T_j @ M, B_1, B_2, \cdots, B_n].$$

Since S is a minimum cost schedule we know that it is ordered. Thus either $A_1 = T_0$ or $B_1 = T_0$.

Suppose $B_1 = T_0$. Let S' be the schedule

$$[A_n, A_{n-1}, \cdots A_1, B_1 @ M, T_j, B_2, \cdots, B_n].$$

Then the cost due to B_i $(2 \leq i \leq n)$ increases by $(l_j - l_0)/2$, the cost due to A_i $(1 \leq i \leq n)$ decreases by $(l_j - l_0)/2$, the cost due to T_j increases by $(l_0 + l_j)/2$, and the cost due to T_0 decreases by $(l_0 + l_j)/2$. Thus $cost(S') = cost(S) - (l_j - l_0)/2$. Since $l_j > l_0$ we conclude that $cost(S') < cost(S)$. This contradicts the fact that S is a minimum cost schedule.

The case where A_1 is T_0 is handled similarly. Thus $m_0(S) = M$ if S is a minimum cost schedule. □

Thus far we know that a minimum cost schedule must be an ordered schedule without gaps such that the shortest task has its midtime at M and the numbers of tasks on the two sides of M are equal. We would like to further characterize the form of any minimum cost schedule by showing how to choose the side of M on which a task should be scheduled. Towards this end we make the following definition. If

$$S = [A_n, A_{n-1}, \cdots A_1, T_0 @ M, B_1, B_2, \cdots, B_n]$$

then define

$$S_i = [A_n, \cdots A_{i+1}, B_i, A_{i-1}, \cdots A_1, T_0 @ M, B_1, \cdots, B_{i-1}, A_i, B_{i+1}, \cdots, B_n].$$

That is, S_i is the schedule obtained from S by swapping the order of A_i and B_i and leaving the order of the remaining tasks the same. We now show that swapping such tasks does not change the cost of the schedule. Intuitively, this follows because the number of tasks scheduled before A_i is the same as the number of tasks scheduled after B_i and hence the number of times each contributes its length to the cost is the same in either case.

Lemma 5. For S_i as defined above, $cost(S_i) = cost(S)$.

Proof. Let a_i be the length of A_i and let b_i be the length of B_i. Without loss of generality assume $a_i < b_i$.

The cost due to A_j and B_j $(1 \leq j \leq i-1)$ remains the same in S_i as in S. Also the cost due to T_0 remains 0. The costs due to the $n - i$ tasks $B_{i+1}, B_{i+2}, \cdots, B_n$ are $b_i - a_i$ less in S_i than in S and the costs due to the $n - i$ tasks $A_{i+1}, A_{i+2}, \cdots, A_n$ are $b_i - a_i$ more in S_i than in S. Therefore the changes in the costs due to all of these tasks cancel each other out.

The sum of the costs due to A_i and B_i remains the same in S_i as in S. Thus swapping A_i and B_i causes no change in the cost of the schedule. □

As a consequence of the previous lemma we know that S is a minimum cost schedule if and only if S_i $(1 \leq i \leq n)$ is a minimum cost schedule.

Lemma 6. If $S = [A_n, A_{n-1}, \ldots, A_1, T_0 @ M, B_1, B_2, \ldots, B_n]$ is a minimum cost schedule, then $\{A_i, B_i\} = \{T_{2i}, T_{2i-1}\}$.

Proof. Consider S_i where $1 \leq i \leq n$. By the above remarks we know that S_i is also a minimum cost schedule. Therefore S_i must also be an ordered schedule. Thus, not including T_0, there must be exactly $2(i-1)$ tasks with lengths less than both the length of A_i and the length of B_i (and hence $2(n-i)$ with greater lengths). The only tasks with this property are T_{2i} and T_{2i-1}, as required. Hence A_i is one of T_{2i} and T_{2i-1} and B_i must be the remaining one. □

Given the above characterization of a minimum cost schedule it is easy to show that the cost of any such schedule is $\sum_{i=1}^{n}(l_{2i} + l_{2i-1})(n - i + 1/2) + (l_0)n$.

We are now in a position to show that the total discrepancy problem is NP-complete. Again, we show that the problem is NP-hard by reducing a known NP-complete problem (the even-odd partition problem) to it, leaving the proof that it is in NP as a simple exercise.

Theorem 1. The total discrepancy problem is NP-complete.

Proof. Let EO be an instance of the even-odd partition problem. That is, we are given a set $B = \{b_1, b_2, \cdots, b_{2n}\}$ of positive integers, with $b_i < b_{i+1}$ for $1 \leq i < n$, and we wish to find a partition of B into sets B_1 and B_2 such that for each i, $1 \leq i \leq n$, exactly one of b_{2i} and b_{2i-1} is in B_1 (and the other is in B_2) and such that $\sum_{b \in B_1} b = \sum_{b \in B_2} b$. Without loss of generality we assume that $b_1 > 1$.

From this instance we define the following instance D of the total discrepancy problem. We take N, the number of tasks to be $2n+2$. The tasks are $T_0, T_1, \cdots, T_{2n+1}$ such that $l_0 = b_1 - 1$, $l_{2n+1} = 2$ and $l_i = b_i$ for $1 \leq i \leq 2n$. We set $M_j = \sum_{i=0}^{2n} l_i/2 = M$ for $0 \leq j \leq 2n$ and $M_{2n+1} = 2M + l_{2n+1}/2$. We wish to know if there is a schedule of the tasks such that the cost of the schedule is at most $k = \sum_{i=1}^{n} (l_{2i} + l_{2i-1})(n - i + 1/2) + (l_0)n$.

By our previous results we know that the cost of any schedule of the tasks $T_0, T_1, \cdots, T_{2n}$ must be at least k. Therefore there is a schedule S of all the tasks that costs at most k if and only if the cost due to T_{2n+1} in S is 0 and the total cost due to the other tasks is k.

Let S be any schedule of the $2n+2$ tasks and let S' be the schedule that results by removing T_{2n+1} from S. Then S' has cost k if and only if S' is a minimum cost schedule for $T_0, T_1, \cdots, T_{2n}$, and we know from our previous results that any such S' must be an ordered schedule without gaps, must have $m_0(S') = M$, and must have exactly one of T_{2i} and T_{2i-1} in $A(S')$ (and the remaining one in $B(S')$).

The additional cost due to T_{2n+1} in S is 0 if and only if

$$M + l_0/2 + \sum_{T_i \in B(S')} l_i \leq M_{2n+1} - l_{2n+1}/2 = 2M.$$

Since

$$M = \sum_{i=0}^{2n} l_i/2,$$

we can conclude that the additional cost due to T_{2n+1} is 0 if and only if

$$\sum_{T_i \in B(S')} l_i \leq \sum_{i=1}^{2n} l_i/2$$

and

$$\sum_{T_i \in A(S')} l_i \leq \sum_{i=1}^{2n} l_i/2$$

because the tasks in $A(S')$ must be scheduled between 0 and $M - l_0/2$.

Therefore the schedule S has cost at most k if and only if S' is an ordered schedule without gaps, exactly one of T_{2i} and T_{2i-1} is in $A(S')$, and $\sum_{T_i \in B(S')} l_i = \sum_{T_i \in A(S')} l_i = \sum_{i=1}^{2n} l_i/2$.

Let $B_1 = \{b_i : T_i \in A(S')\}$ and $B_2 = \{b_i : T_i \in B(S')\}$. Then S is a solution to D if and only if B_1 and B_2 form an even-odd partition of B that satisfies EO. □

2.2. A Scheduling Algorithm for a Fixed Task Ordering

We now turn to the special case in which the tasks are to be executed in a given fixed sequence. Here, and in the remainder of the paper, we shall assume the preferred times for the tasks are given as preferred starting times, and throughout the rest of this section we shall refer to the sum of the task discrepancies as the *cost* of the schedule.

Assume that for $1 \leq i < N$, task T_i must be completed by the time task T_{i+1} is started, i.e. the tasks must be executed in the order given by their indices (which is arbitrary). Our scheduling algorithm schedules tasks one at a time in increasing order by index, sometimes shifting previously scheduled tasks earlier when scheduling a new task.

Let S_n be the schedule computed for the first n tasks. The *blocks* of the schedule are the maximal sets of tasks $T_{i_0}, \ldots, T_{i_1}$ that are scheduled contiguously, i.e., $s_i + l_i = s_{i+1}$ for $i_0 \leq i < i_1$, $s_{i_0 - 1} + l_{i_0 - 1} < s_0$ (or $i_0 = 1$), and $s_{i_1} + l_{i_1} < s_{i_1 + 1}$ (or $i_1 = n$). Assume that there are t blocks $B_1, \ldots, B_t$ in S_n.

We define a partition of each block B_j into two subsets, *Decrease*(j) and *Increase*(j), as follows: task $T_i \in B_j$ is in *Decrease*(j) if $s_i > p_i$ and is in *Increase*(j) if $s_i \leq p_i$. The idea is that if $T_i \in$ *Decrease*(j), reducing s_i decreases the discrepancy of T_i, whereas if $T_i \in$ *Increase*(j), reducing s_i increases the discrepancy of T_i. We define *Inc*(j) to be the number of tasks in *Increase*(j) and *Dec*(j) to be the number of tasks in *Decrease*(j). As a way of representing the blocks, we denote by *first*(j) and *last*(j) the smallest and largest indices, respectively, of tasks in B_j.

Our initial schedule S_1 simply schedules T_1 to start at time p_1. In general, given S_n, we schedule T_{n+1} as follows. There are two cases. If $s_n + l_n \leq p_{n+1}$, then we schedule T_{n+1} to start at p_{n+1}. In this case T_{n+1} has no discrepancy, and S_n and S_{n+1} have the same cost. If on the other hand $s_n + l_n > p_{n+1}$, we begin by scheduling T_{n+1} to start at $s_n + l_n$. Now T_{n+1} has positive discrepancy, and both the block B_t and the set *Decrease*(t) have gained T_{n+1} as a member. A key property that our algorithm will maintain is that, for each block B_j in S_n, either $Dec(j) < Inc(j)$ or $s_{first(j)} = 0$ (in which case $j = 1$). Hence, after scheduling T_{n+1}, we have either $Dec(t) \leq Inc(t)$ or

$s_{first(t)} = 0$. If $s_{first(t)} = 0$ or $Dec(t) < Inc(t)$, we take no further action; the current schedule is S_{n+1}. On the other hand, if now $Dec(t) = Inc(t)$ and $s_{first(t)} \neq 0$, we can shift the entire block B_t earlier without affecting the cost of the schedule. We shift block B_t earlier until one of three things happens:

(i) $s_{first(t)}$ becomes zero;

(ii) for some $T_i \in B_t$, s_i becomes equal to p_i; or

(iii) $s_{first(t)}$ becomes equal to $s_{last(t-1)} + l_{last(t-1)}$.

The resulting schedule is S_{n+1}.

Observe that case (i) above can only occur if $t = 1$; if it occurs, block B_t cannot be moved earlier because it now starts at time zero. In case (ii), task T_i is transferred from set *Decrease*(t) to *Increase*(t), and further shifting of B_t will only increase the cost of the schedule. In case (iii), block B_t is merged into block B_{t-1}. By the property mentioned earlier, in S_n it will be true that either $Dec(t-1) < Inc(t-1)$ or $s_{first(t-1)} = 0$, which implies that after B_t is merged into B_{t-1}, it is still true that either $Dec(t-1) < Inc(t-1)$ or $s_{first(t-1)} = 0$, i.e., the combined block cannot be shifted earlier without increasing the cost (or illegally starting a task before time 0).

The scheduling algorithm consists of beginning with the schedule S_1 and applying the above construction to form schedules $S_2, S_3, \ldots, S_N$. The observations above suggest why the algorithm should work. We prove correctness in the next section.

2.3. Proof of Correctness

For any schedule S, we denote the cost of S by $cost(S)$. Our correctness proof begins with a lemma.

Lemma 7. For any schedule S_n computed by the algorithm, each block B_j in S_n satisfies $Dec(j) < Inc(j)$ or $s_{first(j)} = 0$.

Proof. By induction on n, using the observations at the end of Section 2.2. □

Theorem 2. For any n, the schedule S_n computed by the algorithm has minimum cost among all schedules of the first n tasks.

Proof. By induction on n. The theorem is certainly true for the empty schedule S_0. Assuming that it is true for values 0 through n, we shall prove it for $n + 1$. Let S be any schedule of the first $n + 1$ tasks, and let s_i and s_i' be the starting times of task T_i in S_n and S, respectively. We consider two cases.

Case 1. If $s_n + l_n \leq p_{n+1}$, then in S_{n+1} task T_{n+1} is scheduled to start at p_{n+1}, i.e., it has a discrepancy of zero. Thus $cost(S_{n+1}) = cost(S_n)$. But $cost(S) \geq cost(S_n)$ by the induction hypothesis, since S with T_{n+1} deleted is a schedule for the first n tasks, having cost at least $cost(S_n)$ by the induction hypothesis, and the discrepancy of

T_{n+1} in S (and indeed in any schedule) is non-negative. It follows that $cost(S) \geq cost(S_{n+1})$, i.e., the theorem is true.

Case 2. On the other hand, suppose $s_n + l_n > p_{n+1}$. Let R be the schedule formed from S_n by starting T_{n+1} at time $s_n + l_n$. Then $cost(S_{n+1}) = cost(R) = cost(S_n) + s_n + l_n - p_{n+1}$, since the shift that produces S_{n+1} from R does not change the cost of the schedule. (This follows from Lemma 7 by the discussion in the description of the algorithm.) Let us compare $cost(S)$ with $cost(R)$. We consider two subcases.

Case 2.1. If we have $s_{n+1}' \geq s_n + l_n$, then $cost(S) \geq cost(S_n) + s_{n+1}' - p_{n+1} \geq cost(S_n) + s_n + l_n - p_{n+1} = cost(R)$, which means the theorem is true.

Case 2.2. The remaining case is $s_{n+1}' < s_n + l_n$. Let B_t be the last block in R, which consists of the last block in S_n augmented by T_{n+1}. By Lemma 7 and the discussion in the algorithm description, $Dec(t) \leq Inc(t)$, where these values are defined with respect to R. Since every task in B_t is scheduled no later in S than in R, it follows that the total discrepancy of the tasks in B_t is no less in S than in R. Let S' denote the schedule of the tasks in $T_1, T_2, \ldots, T_{first(t)-1}$ formed from S by removing the tasks in B_t. Removing the tasks in B_t from R leaves exactly $S_{first(t)-1}$. By the induction hypothesis, $cost(S') \geq cost(S_{first(t)-1})$, which implies by the observation above that $cost(S) \geq cost(R)$, and again the theorem is true. □

Note that the minimum cost schedule is not necessarily unique. The algorithm computes an "earliest starting time" minimum schedule; that is, among minimum cost schedules, the one computed by the algorithm schedules every task as early as possible. Note also that by changing the task ordering it may be possible to reduce the cost of the schedule.

2.4. Efficient Implementation

It is straightforward to implement the scheduling algorithm to run in $O(N^2)$ time. To achieve a smaller time bound, we need a data structure that will allow us to determine quickly the amounts by which blocks should shift. For this purpose we use *heaps* [4] (sometimes called *priority queues* [2]). For each block B_j in the current schedule, we maintain a heap $h(j)$ containing all indices i such that $T_i \in Decrease(j)$. Each index i in $h(j)$ has a *key* equal to $s_i - p_i$, i.e., the maximum amount by which s_i can be decreased while decreasing the discrepancy of T_i. We need the following operations on heaps:

make_heap (h):	create a new, empty heap, named h.
find_min (h):	find an item of minimum key in heap h and return its key, without removing the item from h.

delete_min (h):	find an item of minimum key in h, delete it from h, and return it.
insert (i,x,h):	insert item i with key x in heap h.
meld (h_1,h_2):	combine item-disjoint heaps h_1 and h_2 into a new heap, which becomes h_1; h_2 becomes empty.
add_to_all_keys (x,h):	add x to the keys of all items in h.

These operations can be implemented to run in $O(\log N)$ time per operation (each heap will contain at most N items). Indeed, the time for *make_heap* and *add_to_all_keys* is $O(1)$ [4].

The implementation of the algorithm as the procedure *minimize_sum* is given in Figure 1. It uses the auxiliary procedure *shift* given in Figure 2.

```
procedure minimize_sum;
    $t := first(1) := last(1) := Inc(1) := 1; Dec(1) := 0; s_1 := p_1;$
    $make\_heap(h(1));$
    for $n := 1$ to $N$ do
        if $s_n + l_n < p_{n+1}$ then
            {$t := t + 1; first(t) := last(t) := n + 1;$
            $Inc(t) := 1; Dec(t) := 0; s_{n+1} := p_{n+1};$
            $make\_heap(h(t))$}
        else if $s_n + l_n = p_{n+1}$ then
            {$last(t) := n + 1; Inc(t) := Inc(t) + 1; s_{n+1} := s_n + l_n$}
        else if $s_n + l_n > p_{n+1}$ then
            {$last(t) := n + 1; Dec(t) := Dec(t) + 1; s_{n+1} := s_n + l_n;$
            $insert(n + 1, s_{n+1} - p_{n+1}, h(t));$
            if $Dec(t) = Inc(t)$ then shift}
    end for
end minimize_sum;
```

Figure 1. The procedure *minimize_sum*.

```
procedure $shift$;
    $\Delta_1 := find\_min(h(t))$;
    $\Delta_2 :=$ if $t = 1$ then $s_1$ else $s_{first(t)} - s_{last(t-1)} - l_{last(t-1)}$;
    $\Delta := \min\{\Delta_1, \Delta_2\}$;
    $add\_to\_all\_keys(-\Delta, h(t))$;
    $s_{first(t)} := s_{first(t)} - \Delta$; $s_{last(t)} := s_{last(t)} - \Delta$;
    while $find\_min(h(t)) = 0$ do
        $i := delete\_min(h(t))$;
        $Dec(t) := Dec(t) - 1$; $Inc(t) := Inc(t) + 1$;
    end while;
    if $s_{first(t)} = s_{last(t-1)} + l_{last(t-1)}$ then
        $\{meld(h(t-1), h(t))$;
        $last(t-1) := last(t)$;
        $Inc(t-1) := Inc(t-1) + Inc(t)$;
        $Dec(t-1) := Dec(t-1) + Dec(t)$;
        $t := t - 1\}$
end $shift$;
```

Figure 2. The procedure *shift*.

When *minimize_sum* finishes its computation, the values of $first(j)$, $last(j)$, $s_{first(j)}$, and $s_{last(j)}$ for each block B_j are correct; starting times for tasks not beginning or ending a block may not be correct because they do not reflect shifts. The correct starting times of the tasks can be computed in $O(N)$ time using the following recurrence:

$$s_i = \begin{cases} s_{first(j)} & \text{if } i = first(j) \text{ for some } j \le t \\ s_{i-1} + l_{i-1} & \text{otherwise.} \end{cases}$$

It is straightforward to show that the implementation is correct and runs in $O(N \log N)$ time.

2.5. Equal Task Lengths

Suppose that all tasks have the same length. Then, as we show below, among schedules having no restriction on the task ordering, there is always a minimum cost one that executes the tasks in order of increasing (nondecreasing) preferred starting times. Thus in this case we can find a globally minimum cost schedule by first sorting the tasks by preferred starting time and then applying the previously described algorithm. As a notational convenience, we shall assume in this section that the tasks are indexed so that $p_i \leq p_{i+1}$.

Theorem 3. If all tasks have the same length, then there is a minimum cost schedule in which $s_i \leq s_{i+1}$ for $1 \leq i < N$.

Proof. We use a standard interchange argument. Assume S is any schedule containing two tasks T_i and T_j with $p_i < p_j$ but $s_i > s_j$. We shall show that the tasks can be interchanged in S without increasing the cost of S. The theorem then follows by induction on the number of interchanges needed to put the tasks in order by preferred starting time.

Since T_i and T_j have equal lengths, interchanging them results in a feasible schedule, i.e. it does not produce overlapping tasks. There are six cases, depending on the relative ordering of p_i, p_j, s_i, and s_j. In the two cases $s_j < s_i \leq p_i < p_j$ and $p_i < p_j \leq s_j < s_i$, interchanging T_i and T_j results in no change in the cost of the schedule. In the four cases $s_j \leq p_i \leq s_i \leq p_j$, $s_j \leq p_i < p_j < s_i$, $p_i \leq s_j < s_i \leq p_j$, and $p_i \leq s_j \leq p_j \leq s_i$, the interchange always reduces the cost. □

2.6. Generalizations of the Algorithm

The following generalizations, motivated primarily by the additional constraints considered by Vere [5], can be made to the procedure *minimize_sum* of Section 2.4. It should be noted, however, that the ordering result for equal length tasks (Theorem 3) need not continue to hold in these versions of the problem.

2.6.1. Window Constraints

We assume in this section that for each T_i, we are given a window of time $[a_i, b_i]$ in which T_i must be started. We call these constraints *window constraints.* Following Vere [5], we assume that each window $[a_i, b_i]$ is such that $a_i + l_i \leq a_{i+1}$ and $b_i + l_i \leq b_{i+1}$. Suppose that several of the tasks have their starting times fixed in a consistent manner (i.e., the fixed tasks do not overlap one another in time). Then the above property ensures that there exists a schedule in which the fixed tasks start at their fixed times and the starting times of the remaining tasks fall within their given windows.

It is easy to extend the algorithm given previously to handle such constraints. A block B_j is said to be *fixed* if $s_i = a_i$ for some T_i in B_j. That is, no task in B_j can be rescheduled earlier without splitting the block or violating a window constraint. When shifting the rightmost block B_t to the left in modifying the schedule for the first n tasks to incorporate T_{n+1}, we now have an additional stopping criterion: we stop shifting if B_t becomes fixed. No shifting is possible if B_t is already fixed, or if T_{n+1} is merged into

B_t and causes B_t to become fixed because $s_{n+1} = a_{n+1}$.

To keep track of when blocks become fixed, we need only maintain a single variable for each block B_j, equal to the least distance of any task in the block to the left end of its window. When two blocks are merged (or a task is merged into a block), the new value of this variable is the smaller of the corresponding values for the two original blocks. In determining how far we can shift B_t, we simply take the smaller of this value and the shift distance computed in the previous version of *shift*. Clearly this can be done without increasing the time complexity of the algorithm.

2.6.2. Consecutive Task Constraints

We next consider the case in which we allow *consecutive task constraints*, which are of the form $s_{i+1} = s_i + l_i$. We shall show how to extend the scheduling algorithm to handle such constraints. We call a set of consecutive tasks $T_j, T_{j+1}, \ldots, T_k$ a *string* if T_{i+1} is constrained to start when T_i finishes for $j \leq i < k$, but not for $i = j - 1$ or $i = k$. (That is, $T_j, T_{j+1}, \ldots, T_k$ is a maximal set of tasks whose starting times are fixed with respect to each other.)

We modify the algorithm so that at each iteration it attempts to schedule the next string instead of just the next task. For any string it is easy to find a starting time for the string so as to maximize the number of tasks in the string with initial starting times greater than their preferred starting times. Call this time q_k for the k^{th} string. The first step in the algorithm is very similar to that of the original algorithm, except that the relationship between $s_n + l_n$ and q_n is tested. Similar actions are taken as before, except that instead of just adding one task to $h(t)$, each task in the string must be added.

The major difference in the algorithm is that whereas before we only had to shift at most once in each iteration of the main loop, now it may be the case that after the string is placed at the end of the schedule (i.e. just before *shift* is called for the first time) we have $Dec(t) > Inc(t)$. Instead of just shifting once we continue to shift until $Dec(t) < Inc(t)$ or the left end of the block reaches 0. Notice that once a task joins $Increase(t)$ it never leaves. Also, every time *shift* is called, the difference between $Dec(t)$ and $Inc(t)$ is reduced by at least one or else the left end of the block reaches 0, in which case *shift* is no longer called. Thus, in each iteration of the main loop the number of times *shift* can be called is at most the number of tasks in the string being added. The modified algorithm takes $O(N \log N)$ time in the worst case, since *shift* is called at most N times and takes $O(\log N)$ time per call.

2.6.3. Weighted Sum of Discrepancies

Our final generalization allows each task T_i to have an associated *weight* $w_i > 0$, with the objective now being to minimize the weighted sum of the discrepancies from the preferred starting times.

The key change to the algorithm necessary to handle this generalization is to change the definitions of *Inc*(i) and *Dec*(i). They are redefined to be the sum of the weights in the corresponding sets *Increase*(i) and *Decrease*(i), rather than the number of tasks in these sets. It is not hard to see that the updating of the corresponding variables in the algorithm is easily changed to correspond to these new definitions. However, as in the previous section, when adding a new task to the schedule, we may again have to shift more than once, since merged blocks can still have $Inc(i) \leq Dec(i)$. The details of the modifications to the algorithm are straightforward. An argument similar to that of the previous section shows that the worst-case time complexity remains $O(N \log N)$. Also notice that for any combination of the generalizations we have presented, the worst-case time complexity is $O(N \log N)$.

3. Sequencing Tasks to Minimize Maximum Discrepancy

In this section we consider the problem of scheduling N tasks so as to minimize the maximum discrepancy of any task from its preferred starting time.

One convenient way to address this problem is to seek an algorithm for the simpler variant in which we are also given a bound $\gamma \geq 0$ on the desired maximum discrepancy and asked whether there exists a schedule with maximum discrepancy no more than γ, with the requirement that we construct such a schedule whenever one exists. Any algorithm for solving this version can be used as a subroutine to solve the original problem, simply by performing a binary search on γ to find its minimum achievable value. (The sum of the task lengths is an obvious upper bound on γ, although in practice a better bound will usually be obtained by taking the maximum discrepancy actually achieved by some heuristically obtained schedule. The number of bits of accuracy needed to describe the optimal value of γ is certainly no more than the number of bits in the sum of the task lengths plus the sum of the preferred starting times.) Hence we shall concentrate on this variant.

We can reformulate this version of the problem in terms of individual task "release times" and "deadlines" by observing that our goal is equivalent to finding a schedule that begins each task T_i no sooner than $\max\{0, p_i - \gamma\}$ and that finishes T_i no later than $p_i + l_i + \gamma$. Thus, instead of thinking of each task T_i as having a preferred starting time p_i, we can think of T_i as having a given *release time* r_i $(= \max\{0, p_i - \gamma\})$ and a given *deadline* d_i $(= p_i + l_i + \gamma)$, with our goal being a schedule that begins each task no earlier than its release time and finishes it no later than its deadline. Although the general problem of determining whether there exists a one-processor schedule for such a set of tasks is NP-complete [1], in our case the task release times and deadlines have a special form. Namely, there exists a constant $\alpha = 2\gamma$ such that, for each task T_i, either $d_i - r_i = l_i + \alpha$ or $r_i = 0$ and $d_i < l_i + \alpha$. We shall see that this makes the problem significantly easier.

We shall work with the problem in this form. A schedule will be called *feasible* if it begins each task no sooner than its release time and finishes it no later than its deadline. A schedule is *optimum* if it is feasible and, among all feasible schedules, has minimum makespan (i.e., minimizes the completion time of the latest finishing task). This additional optimization criterion will be useful for building up new schedules from those for smaller sets of tasks. Indeed, our algorithm will actually produce an optimum schedule for our problem rather than just an arbitrary feasible schedule.

3.1. Normal Forms for Optimum Schedules

The key result we shall use for solving this problem is the following:

Theorem 4. Consider any instance of the above scheduling problem, and let T_N be a task having the largest deadline in this instance. Then, if there exists a feasible schedule, there exists an optimum schedule in which the tasks that follow T_N all have release times strictly later than the time at which T_N starts. (We say that such a schedule satisfies the *release time property.*)

Proof. Consider any such instance for which a feasible schedule exists, and let T_N be a maximum deadline task. For any feasible schedule, let us say that a task T_i is a *straggler* in that schedule if it is executed after T_N but has a release time no greater than the time at which T_N starts. Suppose, contrary to the theorem statement, that every feasible schedule for this instance has at least one straggler.

Consider any optimum schedule with the following properties:

(a) Among all optimum schedules, it has the fewest tasks executed after T_N.

(b) Among all optimum schedules satisfying (a), it has the fewest tasks executed between T_N and the first straggler that follows T_N.

Let $T_1, T_2, \ldots, T_j$ denote the sequence of tasks executed after T_N in this schedule up to and including the earliest straggler T_j. For $1 \leq i \leq j$, let t_i denote the starting time for T_i in the schedule, let t_0 denote the starting time for T_N, and let t_{j+1} denote the finishing time for T_j. We shall alter the portion of the schedule between t_0 and t_{j+1} in such a way as to show that the original schedule could not have satisfied the stated properties, a contradiction to the assumption that every feasible schedule includes a straggler.

First, suppose that some task among $T_1, T_2, \ldots, T_{j-1}$ has its deadline at or after

t_{j+1}. Let T_k be the rightmost such task. We construct a new schedule by pulling out T_k, moving $T_{k+1}, \ldots, T_j$ left to t_k, and reinserting T_k into the gap created, so it now finishes at time t_{j+1}. Obviously, T_k is still legally scheduled, since it moved right to a point no greater than its deadline. Also, T_j is still legally scheduled, since it moved left, but still begins after $t_0 \geq r_j$. Since each task $T_i \in \{T_{k+1}, \ldots, T_{j-1}\}$ satisfies $d_i < t_{j+1}$, it must be the case that $r_i = d_i - \alpha - l_i < t_{j+1} - \alpha - l_i \leq t_1 - l_i < t_1$, because $t_{j+1} \leq d_k \leq d_N = r_N + l_N + \alpha \leq t_0 + l_N + \alpha \leq t_1 + \alpha$. (The release time r_i must differ from d_i by exactly $\alpha + l_i$, because $r_i > t_0 \geq 0$). Thus, since T_i does not start before t_1 in the new schedule, it too is still legally scheduled. Hence the new schedule is feasible, has the same number of tasks executed after T_N as the original schedule, and has fewer tasks executed between T_N and T_j than before, a contradiction to our choice of original schedule. Therefore, we know that the deadlines for the tasks $T_1, T_2, \ldots, T_{j-1}$ must all precede t_{j+1}.

Now suppose we interchange the tasks T_N and T_j, shifting the block of tasks between them left or right depending on which of the two is longer. We want to show that the new schedule is still feasible, which will give us our final contradiction, since the new schedule is still an optimum schedule and has fewer tasks executed after T_N than the original.

Obviously T_N and T_j are legally scheduled in the new schedule, since their release times are both at or before t_0 and their deadlines are both at or after t_{j+1}. Consider any task T_i, $1 \leq i \leq j-1$. Since $d_i < t_{j+1}$, the release time r_i for T_i is less than $t_{j+1} - \alpha - l_i$ (again, r_i must differ from d_i by the full amount $\alpha + l_i$, because $r_i > t_0 \geq 0$). However, T_i now starts after T_j finishes, which is at most α to the left of where T_j finished in the original schedule, i.e., at least $t_{j+1} - \alpha$, which is already greater than the release time for T_i. Thus T_i still starts after its release time. On the other hand, since $r_i > t_0$ (by assumption), we must have $d_i > t_0 + l_i + \alpha > t_0 + \alpha$. Since T_i now finishes before the start of T_N, and T_N starts at most α later than it started in the original schedule, i.e., no later than $t_0 + \alpha$, we also have that T_i still finishes before its deadline. Thus the new schedule remains a feasible schedule and is still optimum because we have not changed the makespan. Since it has fewer tasks executed after T_N than the schedule with which we started, we have our final contradiction, proving the theorem. □

Our algorithm will not be based directly on Theorem 4, but, rather, on the following corollary of Theorem 4:

Corollary 1. Consider any problem instance for which a feasible schedule exists, and let T_N be a maximum deadline task in this instance. Then there always exists an optimum schedule in which, for some real number $\beta \geq 0$, the set of tasks executed before T_N is exactly the set of all tasks with deadline β or less.

Proof. For any optimum schedule of the form given by Theorem 4, let t be the time at which T_N starts in the schedule. Every task T_i executed before T_N has $d_i \leq r_i + l_i + \alpha$. Since T_i finishes no later than time t in the current schedule, we have $r_i + l_i \leq t$ and hence $d_i \leq t + \alpha$. On the other hand, every task executed after T_N has release time greater than t and hence deadline greater than $t + \alpha$. Thus the set of tasks executed before T_N is exactly the set of tasks with deadline $t + \alpha$ or less. □

Consider any optimum schedule of the form given by Corollary 1. Since T_N starts no sooner than r_N and since $d_N = r_N + l_N + \alpha$, the finishing time for T_N must be at least $d_N - \alpha$ (or 0, if this is negative). Any other task T_i satisfies $d_i \leq d_N$ and hence $r_i \leq \max\{0, d_N - \alpha\}$. In particular, this says that all tasks executed after T_N must have release times no greater than the time at which T_N finishes. This immediately implies that there can be no idle time in the schedule from the time that T_N starts to the time that the last task finishes, for otherwise we could shorten the schedule by shifting to the left any tasks that follow such an idle period. Moreover, it also implies that we can without loss of generality execute the tasks following T_N in order of increasing (nondecreasing) deadlines, since, if any ordering of those tasks finishes all of them by their deadlines, this ordering necessarily will.

Let the tasks in our given problem instance be indexed so that $d_1 \leq d_2 \leq \cdots \leq d_N$, and, for convenience, let us introduce an additional dummy task T_0 having $r_0 = d_0 = l_0 = 0$. Then Corollary 1 and the preceding paragraph tell us that, whenever there exists a feasible schedule for this instance, there necessarily exists an optimal schedule of the following *standard form*: For some index j, $0 \leq j \leq N-1$, it begins with a schedule (which can be assumed optimum) for the tasks $T_0, T_1, \ldots, T_j$. This is followed by the remaining tasks $T_N, T_{j+1}, T_{j+2}, \ldots, T_{N-1}$, executed in this order and without any intervening idle time, beginning as soon as possible after the completion of the first part of the schedule, i.e., beginning at the maximum of r_N and the completion time for the prior portion of the schedule. We will call the index j the *split index* corresponding to this standard form schedule.

We also observe that the release time property given in Theorem 4 remains relevant, although it is not the case that every optimum schedule of the above standard form must have this property. The proof of Corollary 1 actually showed that any optimum schedule with the release time property must have the form given in Corollary 1, and hence we know that there always exists an optimum schedule satisfying Corollary 1 and having the release time property. The necessary transformations, as discussed above, performed on such a schedule to obtain a schedule in our standard form neither changed the set of tasks executed after T_N nor increased the starting time for T_N. Thus we have:

Corollary 2. Whenever there exists a feasible schedule for the given problem instance, there always exists an optimum schedule that is both in standard form and has the release time property.

3.2. The Optimization Algorithm

These normalization results lead directly to an algorithm for solving our problem. We first sort the given tasks so that $d_0 \leq d_1 \leq \cdots \leq d_N$. We then successively find optimum schedules for the subproblems consisting of tasks $T_0, T_1, \ldots, T_n$, for n increasing from 0 to N. For each such subproblem, we can restrict our search to solutions in standard form, with T_n playing the role of the maximum deadline task in that subproblem.

To solve the subproblem consisting of T_0 alone, we simply schedule T_0 at time 0. In general, to optimally schedule the task set $T_0, T_1, \ldots, T_n$, given optimal schedules for the smaller sets, we merely need to consider each of the indices $0, 1, \ldots, n-1$ as a possible split index for this problem. For each such index j, we can determine the corresponding standard form schedule using the already-computed optimum schedule for the tasks $T_0, T_1, \ldots, T_j$. We then have to make sure that the resulting schedule is feasible, which requires only that we verify that T_n and all tasks that follow it in the schedule are completed by their respective deadlines. This is because the release times of all the tasks following T_n in the schedule are no greater than the finishing time of T_n (see the discussion following Corollary 1). In checking the various candidate schedules, we can safely ignore any that do not have the release time property (although it does not hurt to consider them), since Corollary 2 assures us that we will still find an optimum schedule. Of the schedules that meet these criteria, the one with least finishing time is an optimum schedule for $T_0, T_1, \ldots, T_n$. If there are several with the least finishing time, we can choose among them arbitrarily. If there are none, then this subproblem is infeasible, which implies that the entire problem is infeasible, so we can halt. Unless the subproblem for this value of n is infeasible, we then increment n by one and repeat the process, continuing until we have completed the case with $n = N$, which solves the problem with which we started.

It is not difficult to implement this approach to run in $O(N^2)$ time; however, with a bit more care the time can be reduced to $O(N)$ plus an initial $O(N \log N)$ for sorting the tasks by deadline. Notice that, when we apply the algorithm repeatedly with different values of α to solve the problem of minimizing the maximum discrepancy of any task from its preferred starting time, the sorting need not be redone each time, since the same ordering by deadline holds for all α.

Hence let us assume that the tasks have been given to us presorted by deadline and indexed so that $d_0 \leq d_1 \leq \cdots \leq d_N$. For $0 \leq n \leq N$, let $F(n)$ denote the finishing time of an optimal schedule for the tasks $T_0, T_1, \ldots, T_n$, and let $S(n)$ denote the split index corresponding to some such schedule that is in standard form. We shall concentrate on computing the values of all the $F(n)$ and $S(n)$, from which an optimal schedule can be constructed easily in linear time.

The key to computing these values in $O(N)$ time is to observe that many of the possible values $0,1, \ldots, n-1$ for the split index j do not actually need to be considered when computing $F(n)$ and $S(n)$, either because the corresponding standard form schedule will not have the release time property or because the length of that schedule cannot possibly be shorter than the length of some other schedule already considered for this value of n. To see how such "irrelevant" possibilities for j arise and are avoided by the algorithm, we first describe the general approach, ignoring the details of the computation.

We shall keep the relevant possibilities for the split index j on a pushdown stack, which will always be in sorted order with the largest index on top. Initially, we set $F(0) = S(0) = 0$ and place the index 0 on the stack as its first element. For each n, $1 \leq n \leq N$, in turn, we compute $F(n)$ and $S(n)$ by working down the stack, considering each of the possible values for the split index j when we encounter it on the stack. If j is the topmost stack element, we compute the length of the standard form schedule corresponding to this split index and determine whether this schedule is feasible. If it is feasible and shorter than the best such schedule seen so far for this value of n, we save it as our new current best.

Suppose this schedule (feasible or not) starts T_n at time $F(j) > r_n$. It follows immediately that the index j will be an irrelevant split index for all larger values of n. For any $k > n$, the schedule for $T_0, T_1, \ldots, T_k$ corresponding to split index j cannot have the release time property, because it executes T_n after T_k even though the release time for T_n is smaller than the time ($F(j)$ or later) at which this schedule begins executing T_k. Therefore, we can now pop j off the stack, because we shall never need to consider j as a split index value again. We then continue by examining as our next possibility the index that has newly risen to the top of the stack.

On the other hand, suppose the schedule starts T_n at time r_n. That is, $F(j) \leq r_n$. Then we know that we need look no deeper in the stack for this value of n. All indices that occur deeper in the stack are less than j and will have corresponding F-values no greater than $F(j)$. Hence the standard form schedule corresponding to each such split index will also start T_n at time r_n, will follow T_n with a superset of the tasks following it in the standard form schedule for split index j, and hence can be no shorter than our current schedule (and cannot be feasible unless the schedule for split index j also is). Therefore we need consider no further split index possibilities for this value of n. We simply set $F(n)$ and $S(n)$ according to the best schedule found in the above process, halting if no feasible schedule was found for this value of n. If $n < N$, we then prepare for the computation of $F(n+1)$ and $S(n+1)$ by pushing the index n onto the top of the stack as a future possibility for the split index j. Notice that in this last case we did not pop from the stack the last split index value considered, since it may still be relevant for later values of n. (We could pop the stack in this case if $F(j) = r_n$ because, as in the earlier case, schedules of the first k tasks (where $k > n$) with j as the split index will not have the release time property.)

It is easy to see that the approach outlined above will consider only $O(N)$ split indices (and corresponding standard form schedules) overall, since each possible split index is placed on the stack exactly once and, except for a single index left on the stack for each value of n, we always pop from the stack any split index that we consider. In order to complete the description of the algorithm, we need to show how in constant time we can compute the length and check the feasibility of each of the relevant schedules.

We maintain for each index j on the stack two additional values, $\lambda(j)$ and $\Delta(j)$, which depend on both j and the next lower index k on the stack. The value of $\lambda(j)$ is simply the sum of the lengths of the tasks $T_{k+1}, T_{k+2}, \ldots, T_j$. The value of $\Delta(j)$ is the latest time at which the block of tasks $T_{k+1}, T_{k+2}, \ldots, T_j$, executed in this order and with no idle time between them, could be completed and still finish each individual task in the block by its deadline (independent of the other tasks in the problem). It is convenient to think of the block of tasks formed from $T_{k+1}, T_{k+2}, \ldots, T_j$ as if it were simply a single task having length $\lambda(j)$ and deadline $\Delta(j)$.

Now let us see how these values are updated and used during the computation of the various $F(n)$ and $S(n)$. For $n = 0$, we simply set $\lambda(0) = \Delta(0) = 0$ when we place the index 0 on the stack. In general, for a particular value of n, $1 \leq n \leq N$, we begin by initializing *bestF* $= \infty$ and *bestS* $= \infty$ as the best F-value and corresponding split index seen so far for this n. We then initialize two new variables, λ and Δ, to the values l_n and d_n, respectively. In general, if j is the index currently on top of the stack, we maintain these variables so that $\lambda = l_{j+1} + l_{j+2} + \cdots + l_n$ and so that Δ is the latest time at which the block of tasks $T_n, T_{j+1}, T_{j+2}, \ldots, T_{n-1}$, executed in this order and with no idle time between them, could be completed and still finish all its constituent tasks by their deadlines. We can think of pushing the block of tasks to later and later times until the finish time of one of the tasks, say T_h, reaches its deadline. The task T_h is said to *determine* Δ. We then begin considering the various possibilities for the split index j on the stack (the first will always be $n-1$).

Let j denote the index currently on top of the stack. The standard form schedule corresponding to this split index starts T_n at time $s = \max\{F(j), r_n\}$ and has makespan $f = s + \lambda$. It is feasible if $f \leq \Delta$. If it is feasible and $f <$ *bestF*, we then set *bestF* $= f$ and *bestS* $= j$, establishing this as our new current best solution.

Consider the case where $s > r_n$. Let k be the index on the stack just below j. In this case we must pop j off the stack and update λ and Δ as follows before going on to consider the next stack element k. The new value of λ is easily seen to be $\lambda = \lambda + \lambda(j)$. The new value of Δ is the latest time at which the block of tasks $T_n, T_{k+1}, T_{k+2}, \ldots, T_j, T_{j+1}, \ldots, T_{n-1}$ could be executed consecutively and still finish in time to satisfy their deadlines. Let T_h be the task that determines this new value for Δ. If T_h is one of $T_{k+1}, \ldots, T_j$, then the new value of Δ is $\Delta(j) + l_{j+1} + l_{j+2} + \cdots + l_{n-1}$. But this can be rewritten as $\Delta(j) + \lambda - l_n$. However, if T_h is one of $T_n, T_{j+1}, T_{j+2}, \ldots, T_{n-1}$ then the value of Δ remains unchanged. Thus we set $\Delta = \min\{\Delta, \Delta(j) + \lambda - l_n\}$.

In the case where $s = r_n$ we know that we can stop the search for a better split index for the the tasks $T_0, T_1, \ldots, T_n$.

At this point we check to see if any feasible split index for n was found. If no feasible split index was found (i.e. *bestF* $= \infty$), we halt since this subproblem is infeasible and so the entire problem is infeasible. Otherwise, we set $F(n) =$ *bestF* and $S(n) =$ *bestS*, push the index n on top of the stack, set $\lambda(n) = \lambda$ and $\Delta(n)$ as described next, and then go on to compute $F(n+1)$ and $S(n+1)$. To see how $\Delta(n)$ should be set, let j denote the index on the stack just below n. If the task that determines $\Delta(n)$ is T_n, then $\Delta(n) = d_n$. On the other hand, if the task that determines $\Delta(n)$ is one of $T_{j+1}, T_{j+2}, \ldots, T_{n-1}$ then $\Delta(n) = \Delta + l_n$. Thus we set $\Delta(n) = \min\{d_n, \Delta + l_n\}$.

It is not hard to check that the values of all variables are maintained as claimed and hence that the procedure correctly computes all $F(n)$ and $S(n)$ in linear time.

The procedure *minimize_max* given in Figure 3 implements this algorithm. It uses operations *push*, *pop*, and *top* (with the obvious meanings) to manipulate the stack, which is assumed to be initially empty.

```
procedure minimize_max;
    $F(0) := S(0) := \lambda(0) := \Delta(0) := 0$;
    push(0);
    for $n := 1$ to $N$ do ;
        $\lambda := l_n$; $\Delta := d_n$;
        $bestF := bestS := \infty$;
        while true do
            $s := \max\{F(top), r_n\}$; $f := s + \lambda$;
            if $f \leq \Delta$ and $f < bestF$ then
                {$bestS := j$; $bestF := f$}
            if $s > r_n$ then
                {$\lambda := \lambda + \lambda(n)$; $\Delta := \min\{\Delta, \Delta(j) + \lambda - l_n\}$; pop}
            else break
        end while;
        if $bestF = \infty$ then halt (infeasible subproblem)
        else
            {$F(n) := bestF$; $S(n) := bestS$;
            push(n); $\lambda(n) := \lambda$; $\Delta(n) := \min\{d_n, \Delta + l_n\}$}
    end for
end minimize_max;
```

Figure 3. The procedure *minimize_max*.

4. Remarks

It should be clear from the results presented in this paper that preferred starting times add an interesting and nontrivial new aspect to combinatorial scheduling problems. We have explored only the simplest variants of such problems, all with only a single processor. Problems with more than one processor are certainly also of interest, although the NP-completeness of most multiprocessor scheduling problems with arbitrary task lengths suggests that the most fruitful areas of exploration will involve identical length tasks.

The main single processor scheduling problem left open by our results is that of finding a more direct approach to minimizing the maximum discrepancy. We have given a way to solve this problem that combines an algorithm for the corresponding decision problem with a binary search on the possible values for the maximum discrepancy. If all task lengths are integers of magnitude at most S, then our overall algorithm runs in time $O(N(\log N + \log S))$. It would be preferable to have an algorithm whose running time is independent of S. Although such a method can be obtained from our basic algorithm by applying a general technique described in [3], the time bound for the resulting method becomes quadratic in N. It remains an open question whether there exists some way to minimize the maximum discrepancy that avoids the binary search and has time complexity only $O(N \log N)$.

References

[1] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., New York, 1979.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[3] N. Megiddo, Combinatorial Optimization with Rational Objective Functions, *Mathematics of Operations Research,* Vol. 4, 1974, pp. 414-424.

[4] R. E. Tarjan, *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[5] S. Vere, Planning in Time: Windows and Durations for Activities and Goals, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. PAMI-5, No. 3, May 1983, pp. 246-267.