WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT

On-Line Maintenance of Optimal Schedules for a Single Machine

Amril Aman
Anantaram Balakrishnan
Vijaya Chandru

SSM#3327-91-MSA August 1991

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139

On-Line Maintenance of Optimal Schedules for a Single Machine

Amril Aman
Anantaram Balakrishnan
Vijaya Chandru

SSM#3327-91-MSA August 1991

On–Line Maintenance of Optimal Schedules for a Single Machine

Amril Aman[*]

School of Industrial Engineering
Purdue University
West Lafayette, IN 47907

Anantaram Balakrishnan [†]

Sloan School of Management
M. I. T.
Cambridge, MA 02139

Vijaya Chandru[**]

School of Industrial Engineering
Purdue University
West Lafayette, IN 47907

August 1991

[*] Supported in part by the NSF Engineering Research Center for Intelligent Manufacturing Systems, Purdue University.

[†] Supported in part by M.I.T.'s *Leaders for Manufacturing* program.

[**] Supported in part by ONR Grant N00014-86-K-0689 and by NSF-ERC Intelligent Manufacturing Systems at Purdue University

Abstract

Effective and efficient scheduling in a dynamically changing environment is important for real-time control of manufacturing, computer, and telecommunication systems. This paper illustrates the algorithmic and analytical issues associated with developing efficient and effective methods to update schedules on–line. We consider the problem of dynamically scheduling precedence–constrained jobs on a single processor to minimize the maximum completion time penalty. We first develop an efficient technique to reoptimize a rolling schedule when new jobs arrive. The effectiveness of reoptimizing the current schedule as a long-term on–line strategy is measured by bounding its performance relative to oracles that have perfect information about future job arrivals.

Keywords: Scheduling, design and analysis of algorithms, heuristics

1. Introduction

Planning and scheduling dynamic systems with random job arrivals, failures, and preemption is a very challenging task. Typically, since future events cannot be forecast with enough detail and accuracy, planners often use on–line scheduling strategies. Consider, for instance, a production system with random job arrivals. On-line methods apply when detailed information regarding a job's processing requirement is revealed only at its release time. Thus, each release time represents an epoch at which the existing schedule is revised to reflect the new information. One on–line scheduling strategy consists of reoptimizing the current "rolling" schedule at each job arrival epoch using a deterministic scheduling algorithm that only uses information about the current system and workload status. We refer to this scheduling strategy as *On–line reoptimization*. This strategy of reacting to changes in system status (job arrival or completion, processor failure, etc.) by reoptimizing and updating the current schedule raises two issues.

First, given an optimal schedule of n tasks, can we devise an **efficient** method to revise this schedule, say, when a new task enters the system? Intuitively, since the existing n-job schedule contains useful information, exploiting this information to adjust the schedule is likely to be more efficient compared to reconstructing the optimal (n+1)-job schedule from scratch. We refer to the latter method as a **zero-base** algorithm, while a method that exploits current schedule information is an **updating** algorithm. Computer scientists have emphasized this issue of relative efficiency of updating methods in the context of certain geometric and graph problems by developing specialized data structures and updating algorithms (see, for example, Spira and Pan [1975], Chin and Houck [1978], Even and Shiloach [1981], Overmars and van Leeuwen [1981], Frederickson and Srinivas [1984], and Frederickson [1985]). In contrast, the research on deterministic resource scheduling (see, for example, Graham et al. [1979]) focuses primarily on **zero-base** algorithms. In this paper, we illustrate the algorithmic issues in dynamic reoptimization by developing an efficient **updating** method for one class of single-machine scheduling problems.

Efficient schedule updating methods are especially important for real-time planning and control. Consider, for instance, the following "bidding" scheme for assigning tasks in a distributed processing system (see, for example, Ramamritham and Stankovic [1984], Zhao and Ramamritham [1985], Malone et al. [1988]). Jobs with varying processing requirements and due dates arrive randomly at different processor locations. Each processor maintains and updates its own local schedule. When a new job enters the system (or when a processor fails), the source node (or a central coordinator) queries the other processors to determine their expected completion time before deciding where to dispatch the job. To formulate its response, each target processor must adjust its current schedule to accomodate the new job and determine its tentative completion time. Subsequently, when the job is awarded to a processor, the selected processor must again update its schedule. Given the possibly large volume of job announcements and reassignments, devising efficient updating algorithms to accomodate new jobs is clearly critical for this type of real-time control mechanism.

In addition to updating efficiency, we are also interested in the **effectiveness** of the on–line reoptimization strategy. In particular, what is the relative performance (i.e., closeness to optimality) of schedules obtained through on-line reoptimization compared to an "optimal" off-line decision procedure that has perfect information about the future? Recently, computer scientists have developed a standardized approach to study this performance characteristic of on-line methods. The approach seeks a worst-case measure called *competitiveness* to evaluate solution effectiveness. We illustrate this mode of analyzing effectiveness using our single machine scheduling example.

Studying efficiency and effectiveness issues for on–line reoptimization required a judicious choice of the scheduling context. In particular, both the structure and performance of on-line updating methods depend strongly on the scheduling objective. Consider, for instance, the problem of scheduling a single machine to minimize maximum tardiness for unrelated jobs with identical release times. The earliest due-date rule finds the optimal schedule for this problem (Jackson [1955]). Given an earliest due-date (EDD) schedule for n jobs, the updating problem consists of constructing a new EDD schedule

(by adjusting the current schedule) to accomodate a new job. If n denotes the number of currently scheduled jobs, re-sorting the (n+1) jobs to construct the new EDD schedule requires O(n log n) operations. However, updating can be performed much more efficiently if we use a heap structure (see, for example, Tarjan [1983], Aho, Hopcroft and Ullman [1974]) to store the current schedule. Updating the heap when a new job arrives merely involves inserting the new item and rebalancing the heap, which requires O(log n) effort. For other scheduling objectives, the updating method is not so obvious, and the n-fold computational improvement may not be possible. And, of course, for NP-hard scheduling problems (e.g., minimizing sum of completion times for jobs with arbitrary release dates) the updating problem is not likely to be polynomially solvable either.

In this paper we develop and analyze the worst-case performance of an on–line updating method for a single machine scheduling problem with precedence-constrained jobs, where the objective consists of minimizing the maximum completion time penalty over all jobs. In the classification scheme proposed by Graham et al. [1979], we consider the $1/prec/f_{max}$ problem. Lawler [1973] proposed an $O(n^2)$ zero-base algorithm to construct an optimal n-job schedule for this problem. Subsequently, Baker et al. [1982] generalized this algorithm to the case where jobs have arbitrary but known release dates, and preemption is permitted. For the $1/prec/f_{max}$ problem, we focus on a special class of penalty functions that satisfy a consistency property defined in Section 2. Several penalty functions such as linear completion time and tardiness penalties satisfy this property. For this class of scheduling problems, Section 3 first describes a new zero-base algorithm called the ***Forward*** algorithm (unlike Lawler's algorithm, this method schedules jobs from front to back) with O(m + n log n) worst-case time-complexity, where m denotes the number of arcs in the precedence graph. Subsequently, we develop an ***updating version*** of the Forward algorithm that uses information about the current n-job optimal schedule to optimally add a new job. If the new job has n' ($\leq$ n) ancestors, and the precedence subgraph induced by these ancestors has m' ($\leq$ m) arcs, the computational complexity of the Forward updating algorithm is O(m' + n'). Results of computer simulations reported in Section 4 confirm that, in practice, the Forward updating procedure requires significantly lower computational time than applying the zero-base algorithm

to construct the (n+1)-job optimal schedule. Section 5 analyzes the competitiveness of on–line reoptimization for selected penalty structures. We show that the method is 2-competitive (i.e., its worst-case performance ratio relative to the optimal, perfect information schedule is bounded above by a factor of 2) for a delivery-time version of the scheduling problem (without preemption), and when the penalty function is subadditive (with preemption).

This paper makes several specific contributions for the $1/prec/f_{max}$ problem with consistent penalty functions. In particular, we: (i) propose a new FORWARD algorithm; (ii) develop an updating (on-line) version of the FORWARD algorithm; (iii) empirically demonstrate the computational benefits of using updating algorithms (instead of zero-base algorithms) to perform schedule adjustments; and, (iv) analyze the competitiveness of on–line reoptimization for some special cases. However, our broader purpose is to use the $1/prec/f_{max}$ problem as an example to motivate the need for further work in the general area of efficient and effective schedule updating methods, and to illustrate the issues that arise in developing such methods.

2. Problem description and notation

The $1/prec/f_{max}$ problem consists of scheduling n jobs on a single machine, subject to precedence constraints on the jobs. Let p_j denote the processing time required for job j. The job precedence constraints are specified via a directed, acyclic ***precedence graph G*** whose nodes correspond to the jobs; the graph contains a directed arc (i,j) from node i to node j if job i is an immediate predecessor of job j. For convenience, assume that jobs are indexed from 1 to n, with $i < j$ if job i precedes job j. Let m denote the number of arcs in the precedence graph. We assume that the precedence graph is stored as a linked list requiring O(m) storage, with pointers from every job to each of its immediate predecessors. Let B_j denote the set of all ***immediate predecessors*** of job j. Job i is said to be an ***ancestor*** of job j if the precedence graph contains a directed path from i to j. Let $A_j \supseteq B_j$ denote the set of all ancestors of job j. Each job j carries a ***penalty function*** $f_j(t_j)$ that depends on its

completion time t_j. The scheduling objective is to ***minimize*** f_{max} = Max $\{f_j(t_j) : j = 1,2,...n\}$.

Lawler [1973] developed the following $O(n^2)$ zero-base algorithm to solve the $1/prec/f_{max}$ problem. The method iteratively builds an optimal sequence by scheduling jobs in reverse order, i.e., it first identifies the job to be processed last (i.e., in position n of the schedule), then the job in position (n–1), and so on. At stage k, let Q_k denote the set of k currently unscheduled jobs, and let T_k denote the cumulative processing time for all jobs in Q_k, i.e., $T_k = \Sigma \{p_j : j \in Q_k\}$. Also, let R_k be the subset of jobs in Q_k whose successors, if any, have all been already scheduled; we refer to jobs in R_k as the set of ***eligible*** jobs at stage k. During stage k, the algorithm assigns to position k the eligible job $j^* \in R_k$ with minimum penalty at time T_k, i.e., $f_{j^*}(T_k) =$ Min $\{ f_j(T_k) : j \in R_k \}$. The procedure terminates at the end of stage 1. Since each step requires O(n) operations to identify the eligible job with minimum penalty, the overall complexity of the algorithm is $O(n^2)$.

While Lawler's algorithm applies to arbitrary penalty functions, we will focus on a special class of penalty functions that satisfy the following *consistency* condition:

> A set of functions $f_1(.), f_2(.), ..., f_n(.)$ is said to be **consistent** if, for every pair of indices $i, j \in \{1, 2, ..., n\}$, either $f_i(t) \leq f_j(t)$ or $f_i(t) > f_j(t)$ for all values of completion time t.

As Figure 1(a) shows, consistent functions do not intersect; Figure 1(b) shows two penalty functions that are not consistent.

Several natural penalty functions satisfy the consistency property. Examples, shown in Figure 2, include (i) the *weighted* (linear and quadratic) *completion time* criteria $f_j(t) = w_j\, t$ or $f_j(t) = w_j\, t^2$ (ii) the *lateness penalty* $f_j(t) = t - d_j$, where d_j is the due date for job j, and (iii) the *tardiness penalty* $f_j(t) = \max \{0, t - d_j\}$. Jobs with consistent penalty functions have the same relative ranking (say, increasing order of penalties) for all completion time values. Hence, we will sometimes omit the completion time argument, and denote

as $f_i > f_j$ the fact that job i has a higher penalty than job j (for all completion time values).

For convenience, we will assume that the penalty functions for the different jobs are distinct, i.e., either $f_i > f_j$ or $f_i < f_j$. Thus, the job with the maximum penalty will always be unique. Our Forward algorithm requires only the relative order of jobs with respect to the penalty functions rather than the exact penalty values for different completion times. Hence, ranking the jobs in order of penalties is sufficient.

3. The Forward Algorithm for 1/prec/f_{max} problem with Consistent Penalty Functions

This section first describes a new zero-base procedure called the *FORWARD* algorithm to find the optimal n-job schedule for the 1/prec/f_{max} problem with consistent penalty functions. Unlike Lawler's algorithm, the new method schedules jobs from front to back (i.e., it assigns jobs to earlier positions first). We prove the correctness of this algorithm, and demonstrate how it facilitates updating the schedule when a new job enters the system.

3.1 The FORWARD Zero-Base Algorithm

The Forward algorithm is motivated by the following intuitive argument. Recall that, for consistent penalty functions, the relative ordering of jobs (in terms of their penalties) does not vary with time. Hence, if jobs are not constrained by precedence restrictions, we can minimize f_{max} by scheduling the jobs in decreasing order of penalty. However, this decreasing-penalty order may violate some precedence constraints. To satisfy the precedence constraints, consider the following 'natural' scheme to selectively (and parsimoniously) advance jobs: Start with the decreasing-penalty order as the candidate sequence; examine jobs from front to back in this sequence, and ensure precedence feasibility for each job j by advancing (i.e., scheduling immediately before job j) every ancestor that is currently scheduled after job j. This procedure effectively attempts to deviate as little as possible from the decreasing-penalty order by advancing only the essential low-penalty jobs that

must precede the high-penalty jobs. As we demonstrate next, this principle forms the basis for the Forward algorithm, and gives the optimal n-job schedule.

To describe and prove the validity of the Forward algorithm, we use some additional notation. For any subset of jobs S, let $B_j(S)$ be the set of all immediate predecessors of job j belonging to subset S, i.e., $B_j(S) = B_j \cap S$. Similarly, $A_j(S) = A_j \cap S$ denotes the set of ancestors of job j in subset S. The Forward algorithm relies on the following result. Recall that we have indexed jobs such that $i < j$ if job i precedes job j.

Proposition 1:

For any subset of jobs S, let j^* be the job in this subset with the maximum penalty. Then, subset S has an optimal schedule, denoted as $\Pi(S)$, that assigns job j^* to position $\{|A_{j^*}(S)| + 1\}$, and all its ancestors to the first $|A_{j^*}(S)|$ positions in increasing order of job indices (where $|A|$ denotes the number of elements of the set A).

Proof:

The first part of the proposition states that the subset S must have an optimal schedule that processes the maximum-penalty job j^* as soon as possible, i.e., this schedule first processes all ancestors of job j^*, followed immediately by j^*. We prove this result using an interchange argument. Consider an alternative optimal schedule Π'' that does not satisfy this property. Let job j^* be scheduled in position $k > |A_{j^*}(S)| + 1$, and let job $j' \notin A_{j^*}(S)$ be a non-ancestor that is scheduled closest to, but before, job j^*. Let k' denote the position of job j' in schedule Π'', $k' < k$. By our choice of k', all jobs in positions (k'+1) to (k–1) must be ancestors of j^*. Also, job j' is not an ancestor for any of these jobs; otherwise, j' would be j^*'s ancestor as well. Finally, since job j^* has the maximum penalty in the set S, $f_{j^*}(t) > f_{j'}(t)$, where t is the current completion time of job j^*. Consider now the new schedule obtained by postponing job j' to position k, and advancing all jobs in positions k'+1 to k by one position. By our previous observations, the new schedule must be feasible; furthermore, since job j^* has a higher penalty than job j', the

new schedule does not increase the maximum penalty of the schedule. By repeating this process until all non-ancestors of job j* are postponed beyond j*, we get an optimal schedule that satisfies the condition of the proposition.

Now, job j* has the maximum penalty among all jobs in S. Thus, the penalty incurred for j* must exceed the penalty for each of its ancestors (since these are completed earlier and have lower penalty functions), regardless of their relative order in positions 1 to $|A_{j^*}(S)|$. To be feasible, however, the assignment of these ancestors must satisfy the precedence constraints. Since jobs are numbered in order of their precedence, scheduling the ancestors in increasing index order gives a feasible schedule. ■

Proposition 1 suggests the following iterative scheduling procedure: first, identify the job j* with maximum penalty, schedule it in position $(|A_{j^*}(S)|+1)$, and assign all its predecessors to positions 1 through $|A_{j^*}(S)|$. Let $S' \subseteq S$ denote the remaining set of jobs (which are not ancestors of j*). In the optimal schedule $\Pi(S)$, these remaining jobs must be scheduled optimally in positions $(|A_{j^*}(S)|+2)$ to $|S|$. In effect, we can consider a new scheduling problem for the subset of jobs S', and apply Proposition 1 to this new subset, and so on. Our method implements this iterative procedure. We formally describe the algorithm next. In this description, r is the *iteration counter*, l_r is the pointer to the last position in the schedule that is filled in the r^{th} iteration, and S_r is the set of remaining unscheduled jobs at the beginning of iteration r.

The FORWARD Zero-Base Algorithm

Step 0: *Initialization*

Set	$r \leftarrow 1$;	iteration counter
	$S_r \leftarrow \{1,2,...,n\}$;	set of unscheduled jobs at iteration r
	$l_{r-1} = 0$.	last position scheduled in previous iteration

Step 1: *Iterative step*

(a) Find the job j*(r) with maximum penalty in set S_r;

$A_{j^*(r)}(S_r)$:= Set of all ancestors of $j^*(r)$ in S_r.

(b) Set $l_r \leftarrow l_{r-1} + |A_{j^*(r)}(S_r)| + 1$;
Assign $j^*(r)$ to position l_r

(c) Assign jobs of the set $A_{j^*(r)}(S_r)$ to positions $(l_{r-1}+1)$ through $l_r - 1$ in increasing order of job indices.

(d) Set $S_{r+1} \leftarrow S_r - A_{j^*(r)}(S_r) - \{j^*(r)\}$

(e) If S_{r+1} is empty, Stop. The current schedule is optimal;
Else, set $r \leftarrow r+1$, and return to Step 1(a).

Observe that the iterative step is performed at most n times. We refer to the job $j^*(r)$ with the maximum penalty at the r^{th} step as the ***r^{th} Bottleneck Job.*** As r increases, the corresponding bottleneck jobs have successively lower penalties.

3.1.1 *Example*

To illustrate the Forward algorithm, consider the precedence graph and the relative ordering of 6 jobs (in decreasing order of penalties) shown in Figure 3. Initially, all jobs are unscheduled, and the job with the largest penalty is job 5 (i.e., $j^*(1) = 5$). This job has two unscheduled ancestors, jobs 1 and 2, i.e., $A_5(S_1) = \{1,2\}$. The first iteration schedules the ancestors in positions 1 and 2, and schedules job 5 in position $l_1 = 3$. At the second iteration, job 6 has the largest penalty among all remaining jobs. Its unscheduled ancestor, job 3, is assigned to position 4, while job 6 is scheduled in position 5. In the final iteration, the only remaining job (job 4) is scheduled in the last position. Table 1 summarizes these computations.

3.1.2 *Data Structures and Computational Complexity*

To perform the Forward algorithm's computations efficiently, we maintain a special data structure, and make some minor algorithmic changes. Our implementation first sorts the jobs in decreasing order of penalties prior

to initiating the main algorithm. This sorting operation requires O(n log n) effort, and will facilitate the process of identifying the bottleneck job at each iteration of the main procedure. Also, our implementation does not determine the exact positions for the unscheduled ancestors in the set $A_{j^*(r)}(S_r)$ (i.e., it does not perform step 1(c)) immediately after at each iteration. Instead, we reindex these jobs temporarily, and determine the actual final schedule at the end of the main algorithm by performing an overall sorting operation. Initially, all jobs have a temporary index of 0. At iteration r, we assign the temporary index (nr + j) to each job $j \in A_{j^*(r)}(S_r)$ that is scheduled during that iteration, and job j*(r) is assigned the index (nr + j*(r)). Thus, all jobs that must be scheduled in the r^{th} iteration (between positions $(l_{r-1}+1)$ and (l_r-1)) have temporary indices in the range (nr+1) to n(r+1). After the main algorithm terminates, we sort the jobs in increasing order of their temporary indices (O(n log n) effort) to obtain the final optimal schedule. Observe that the largest possible value of a temporary index is n(n+1). Also, at intermediate iterations, all the jobs that have not yet been scheduled are easy to identify since they have temporary indices of 0.

Let us now analyze the computational complexity of the Forward algorithm. First, identifying the successive bottleneck jobs involves sequentially scanning the sorted list of jobs, which requires O(n) effort in total. (At step r, the r^{th} bottleneck job j*(r) is the first job following j*(r-1) in the sorted list with a temporary index of 0.) Now, consider the effort required to identify the unscheduled ancestors of job j*(r) (in step 1(a)). Starting with job j*(r), we trace back all unscheduled ancestors using the pointers to the immediate predecessors in the linked list representation of the precedence graph. If we encounter a previously scheduled job, we need not explore its ancestors since these ancestors must all be scheduled previously. Thus, the total effort required to identify the members of the set $A_{j^*(r)}(S_r)$ is O(m) over all iterations (since we examine each edge in the precedence graph exactly once). Combined with the initial and final sorting operations, we get an overall complexity of O(m + n log n). In general, the number of arcs m in the precedence graph is $O(n^2)$; hence, the Forward algorithm is no better than Lawler's original algorithm in the worst-case. However, for problems with sparse precedence graphs, we expect the Forward algorithm to perform better.

The Forward algorithm also extends to the more general $1/prec, r_j, pmtn/f_{max}$ problem where jobs have different release times r_j that are known in advance, and preemption is permitted. Appendix 1 describes this extension. Later (in Section 5), we use the schedule generated by this enhanced method as the benchmark to evaluate the effectiveness of on–line reoptimization when job release times are not known in advance.

3.1.3 Adapting Lawler's algorithm for consistent penalty functions

Note that when the penalty functions are consistent, we can also adapt Lawler's original $O(n^2)$ algorithm for $1/prec/f_{max}$ to run in $O(m + n \log n)$ time. Recall that, at each stage k, for k = n, n-1, ..., 1, Lawler's algorithm selects the eligible job $j \in R_k$ with the smallest penalty at the current completion time T_k. For general penalty functions, the order of eligible jobs (arranged in increasing order of penalty values at T_k) might change from stage to stage. However, with consistent penalty functions, the order is invariant. To exploit this property we use a heap structure to store the currently eligible jobs in sorted order (increasing penalties) at each stage. At stage k, we: (i) schedule the job that is currently at the root of the heap, (ii) delete this job from the heap, and (iii) insert in the heap all its immediate predecessors that just became eligible. Inserting and removing each job from the heap entails $O(n \log n)$ total effort; and, checking the eligibility of jobs at each stage requires O(m) effort (since each arc of the precedence graph must be examined once). Hence, the overall complexity of the heap implementation of Lawler's static algorithm is $O(m + n \log n)$, which is the same as the computational complexity of our Forward algorithm. However, as we show next, the Forward algorithm is more amenable to updating existing schedules.

3.2 The FORWARD Updating Algorithm

For the *updating* problem, we are given an optimal n-job schedule, and a new job, indexed as (n+1), with prespecified immediate predecessors B_{n+1}, arrives. We initially assume that job (n+1) does not have any successors in the current set of n jobs; later, we indicate how to apply the updating method when the new job also has successors among currently scheduled jobs. Let Π

= $\{j_1, j_2, j_3, \ldots, j_n\}$ denote the current optimal n-job schedule where j_k denotes the index of the job that is scheduled in the k^{th} position. The updating problem consists of constructing a new optimal schedule $\Pi' = \{j'_1, j'_2, \ldots, j'_n, j'_{n+1}\}$ that includes the new job (n+1).

The updating procedure uses information on the bottleneck jobs corresponding to the current schedule. As we mentioned in Section 3.1, the successive bottleneck jobs must have successively lower penalties; hence, the current sequence Π already lists the bottleneck jobs in order of decreasing penalties. Consider now the position for job (n+1) in the (n+1)-job optimal schedule Π', assuming we applied the Forward zero-base algorithm. Since job (n+1) does not precede any current jobs, its position in the schedule is determined solely by its penalty function relative to the current bottleneck jobs. In particular, suppose $f_{j^*(r-1)} < f_{n+1} < f_{j^*(r)}$, i.e., the new job's penalty lies between the penalties for the $(r-1)^{st}$ and r^{th} bottleneck jobs. Since the current sequence schedules existing bottleneck jobs in decreasing penalty order, we can identify the index r in linear time. And, the updating procedure must merely insert job (n+1) and all its previously unscheduled ancestors (i.e., ancestors that are not scheduled in positions 1 to l_{r-1}) immediately after the $(r-1)^{st}$ bottleneck job j*(r-1).

Assuming that we are initially given only the immediate predecessors B_{n+1} of job (n+1), finding the members of $A_{n+1}(S_r)$ (the set of unscheduled ancestors for job (n+1)) requires O(m') operations, where m' is the number of edges in the precedence subgraph induced by job (n+1) and its ancestors. We must then assign these ancestors to consecutive positions, starting with position $(l_{r-1}+1)$. Observe that the current schedule satisfies the precedence constraints among all ancestors of job (n+1). Hence, the jobs in $A_{n+1}(S_r)$ need not be re-sorted to satisfy precedence constraints; instead, we get a feasible schedule by merely scheduling these ancestors in the order in which they occur in the current schedule. Adjusting the current schedule in this manner requires at most O(n') effort, where n' is the number of ancestors of job (n+1). Thus, the overall complexity of the Forward updating procedure is O(m'+n') compared with O(m + n log n) for the Forward zero-base algorithm. Finally, note that the updating procedure can easily accommodate new jobs that must

precede existing jobs. Let j_s be the immediate successor of job (n+1) that is scheduled earliest in the current schedule, and let l_s denote its position in the current schedule. In the new schedule, the jobs that are currently scheduled in positions l_s and beyond will retain their relative order. Therefore, we need to apply the updating procedure only to jobs that are currently scheduled in positions 1 to l_s-1.

3.2.1 *Example*

For the example shown in Figure 3, Figure 4 illustrates the updating calculations when a new job (job 7) enters the system. This job has two immediate predecessors, jobs 4 and 5, and its penalty value lies between those of jobs 1 and 6. Job 7 must, therefore, be scheduled between the two consecutive bottleneck jobs 5 and 6. Job 7 has two ancestors, jobs 3 and 4, that are not scheduled prior to job 5. Hence, we assign these two jobs to positions 4 and 5, respectively (preserving their relative order in the current schedule); job 7 occupies position 6, followed by job 6. Figure 4 shows the updated schedule.

4. Computational results

Section 3 showed that the updating algorithm has better worst-case complexity than the zero-base algorithm. To verify this computational superiority in practice, we compared the computation times using the Forward zero-base algorithm and the updating procedure for an extensive set of random test problems ranging in size from 100 jobs to 400 jobs. We first describe the random problem generating procedure before presenting the computational results.

4.1 *Random Problem Generation*

Our random problem generator requires two user-specified parameters: the number of jobs (n), and the density δ of the precedence graph ($0 < \delta \leq 1$). Initially, we attempted to generate precedence graphs with random topologies by independently selecting arcs (i,j), for any pair of nodes i and j, with probability δ. We discovered, however, that the resulting precedence graphs contained many redundant arcs. For example, if the graph contains arcs (i,j),

(j,k), and (i,k), then arc (i,k) can be deleted since this precedence order (i.e., i preceding k) is implied by the other two arcs. Consequently, the reduced graph (with redundant arcs eliminated) was often much sparser than the desired density values. To overcome this problem and to avoid checking for redundancies, we decided to use layered precedence graphs that contain only arcs between successive layers; hence, none of the arcs are redundant.

To generate a random layered graph containing n nodes and with density parameter δ, the problem generator:

- randomly selects the number of layers (L) in the graph ($0 < L \leq n$);
- equally divides the number of nodes among the layers; and,
- for each pair of nodes i and j in successive layers, selects arc (i,j) with probability δ.

For the updating problem, the random generator assigns the new job (n+1) to a new layer, and connects node (n+1) to nodes in the previous layer with probability δ.

We implemented the zero-base and updating versions of the Forward algorithm in PASCAL on a Sun 4/390 workstation. For our computational tests, we considered four networks sizes, with number of nodes n = 100, 200, 300, and 400, and five values of the density parameter δ = 0.10, 0.25, 0.50, 0.75, and 0.90. For each combination (n,δ), we generated 100 random problem instances. Table 2 summarizes the mean (over 100 random instances) and standard deviation of CPU times (to add the $(n+1)^{st}$ job to the current schedule) for the zero-base and updating versions of the Forward algorithm for all the (n,δ) combinations. As Table 2 shows, the updating algorithm is faster than the zero-base version by a factor ranging from 2 to 5. Thus, for scheduling contexts that require numerous frequent updates, the magnitude of computational savings using the updating method can be substantial. For the 100-job problems, the CPU time for individual problem instances varies widely as indicated by the large standard deviation (relative to the mean). As the problem size increases, the CPU time for updating relative to zero-base scheduling appears to increase. The density parameter does not seem to have a significant or consistent effect on this ratio.

5. Competitiveness of On-line Reoptimization

Having demonstrated the relative *efficiency* of using tailored updating methods instead of zero-base algorithms to accomodate new jobs, we now examine the effectiveness of on–line reoptimization as a heuristic strategy to schedule dynamic systems. One approach to evaluate this effectiveness is to assume a tractable stochastic model for job arrivals, processing times, and precedence relationships, and analyze the expected performance of on–line reoptimization (or other dispatch rules) in this framework. However, this mode of analysis is often sensitive to the choice of the stochastic model governing the occurrence of random events.

Recently several researchers in theoretical computer science (e.g., Borodin et al. [1987], Chung et al. [1989], Manasse et al. [1988]) have developed an alternate approach to study on–line effectiveness using the notion of *competitiveness*. The approach involves characterizing the worst-case performance of the on–line method compared to an optimal off-line procedure that has perfect information about the future. In particular, for problems with a minimization objective, an on–line algorithm A is said to be *c-competitive* if the inequality

$$C_A \leq c\, C_0 + \alpha$$

holds for any instance of the on–line problem. Here, C_A denotes the "cost" incurred by the on-line algorithm A, and C_0 is the cost for the optimal off-line solution with clairvoyance. Thus, competitiveness is a useful measure for performance analysis of incremental algorithms. Researchers have started applying this measure to scheduling problems only recently; Shmoys, Wein and Williamson [1991] address competitiveness issues related to on–line scheduling of parallel machines to minimize makespan.

This section demonstrates the underlying principles and techniques of competitiveness analysis applied to our single machine scheduling problem. Developing bounds on the relative difference between the on-line and off-line optimal objective values for general penalty functions $f_j()$ is difficult since we cannot exploit any special properties of the solutions. We, therefore, need to separately study various specializations of $f_j()$. We consider two types of

penalty functions – subadditive functions and lateness. For subadditive penalty functions, we show that on-line reoptimization is 2–competitive when we permit preemptions, and (ρ+2)–competitive for non-preemptive scheduling, where ρ is the Aspect ratio (defined later). We then prove 2–competitiveness of on–line reoptimization for the lateness penalty case (delivery time version).

Before describing and proving the competitiveness results, let us clarify the context and the mechanics of on–line reoptimization. Jobs arrive randomly at various release times r_j. Assume that jobs are indexed in the order in which they arrive. We study effectiveness for both preemptive and non–preemptive scheduling problems. First, consider the case when preemptions are permitted, i.e., at each arrival epoch r_j, the job that is currently in process, say, job u can be interrupted and resumed later without any additional setup or reprocessing effort. In this case, applying the updating method involves: (i) determining the rank, say, r* (in decreasing penalty order) of the new job j relative to all the currently available jobs (including the current in–process job u), and (ii) inserting job j and its unscheduled ancestors immediately after the $(r^*-1)^{st}$ bottleneck job in the current schedule. Notice that the current job u is preempted only if job j has a higher penalty than all other available jobs. In the non-preemptive case, job u must necessarily be completed first, and only the remaining jobs can be rescheduled. Hence, job j is ranked only relative to these remaining jobs.

This on-line updating algorithm is a heuristic method that does not guarantee long-run optimality of the schedules. The benchmark for comparing the performance of the on-line method is an optimal off-line schedule that has prior knowledge (at time 0) about the exact arrival times r_j for all jobs j. In Graham et al.'s [1979] nomenclature, the off–line schedule is the optimal solution to either the $1/prec, r_j, pmtn/f_{max}$ or $1/prec, r_j/f_{max}$ problem depending on whether or not preemption is permitted. Note that the $1/prec/f_{max}$ problem (with preemption) is polynomially solvable using, say, the enhanced Forward algorithm described in Appendix 1, while the $1/prec, r_j/ f_{max}$ problem (without preemption) is known to be NP-hard. Indeed, the non-preemptive problem remains NP-hard even if we restrict the

penalty f_{max} to maximum lateness L_{max}, and relax the precedence constraints (i.e., for the $1/r_j/L_{max}$ problem).

5.1 Subadditive Penalty functions

A penalty function $f_j()$ is said to be *subadditive* if it satisfies the condition:

$$f_j(t_1 + t_2) \quad \leq \quad f_j(t_1) + f_j(t_2)$$

for all $t_1, t_2 \geq 0$. Interesting special cases of subadditive functions include *concave* penalty functions, and *linear* completion time penalties (i.e., $f_j(t) = \beta_j t$). We denote the maximum subadditive penalty as f^{SA}_{max}. This section studies the competitiveness of on-line reoptimization, with and without preemption, when all jobs have consistent, subadditive penalty functions. When preemptions are permitted, we show that the on-line reoptimization strategy is 2-competitive, i.e., the objective value of the on-line schedule is at most twice the optimal off-line value. When preemption is prohibited, the worst-case ratio increases to $(\rho+2)$, where ρ is a specified ratio of job processing times.

5.1.1 Scheduling with Preemptions

We now show that, for any k-job problem, the preemptive schedule obtained using on-line reoptimization (without anticipating future job arrivals) has a worst-case performance ratio of 2 relative to the optimal off-line schedule for the $1/r_j$, prec, pmtn/f^{SA}_{max} problem constructed by the enhanced Forward algorithm (Appendix 1).

Theorem 1:

For the 1/prec, r_j, pmtn/f^{SA}_{max} problem, on-line reoptimization is 2-competitive.

Proof:

Let Π be the preemptive schedule obtained using on-line reoptimization for a k-job problem. Let ϕ_k be the (maximum) completion time penalty for this schedule, and let j' denote the **critical** job, i.e.,

$$\phi_k = f_{j'}(t_{j'}) = \max \{f_j(t_j): 1 \le j \le k\},$$

where t_j is the completion time for job j in the schedule Π. First, we note several characteristics of the schedule Π. Since j' is the critical job, all jobs following job j' in schedule Π must have lower penalty functions (otherwise, a job with higher penalty function that is scheduled later than j' would be the critical job). Consider now the interval of time $[r_{j'}, t_{j'}]$ between the arrival of job j' and its completion. Let J' denote the set of all jobs j that are completed in this interval and <u>having equal or higher penalty functions</u> (J' also includes job j). Let A(J') denote the set of ancestors $j \notin J'$ of all jobs in the set J'. Clearly, every job that the on–line schedule completes in the interval $[r_{j'}, t_{j'}]$ must either be a member of the set J' or an ancestor of some job $j \in J'$. Also, the completion time $t_{j'}$ for job j' in schedule Π has the following upper bound:

$$t_{j'} \le r_{j'} + \sum_{j \in J'} p_j + \sum_{j \in A(J')} p_j \qquad (1)$$

Now consider the optimal off-line schedule Π^* which uses prior information on job release times. Let ϕ_k^* be the (maximum) completion time penalty for this schedule, and let T_j be the completion time for job j in Π^*. Among all the jobs belonging to the set J', let j" be the job that is scheduled last in Π^*. Observe that

$$\phi_k^* \ge f_{j"}(T_{j"}) \ge f_{j'}(T_{j"}), \qquad (2)$$

$$T_{j"} \ge \sum_{j \in J'} p_j + \sum_{j \in A(J')} p_j, \text{ and} \qquad (3)$$

$$T_{j"} \ge r_{j'}. \qquad (4)$$

Inequality (2) follows from the definition of ϕ_k^*, and because job $j" \in J'$ has an equal or higher penalty function than job j'. Inequalities (3) and (4) hold because both j' and j" belong to the set J', and job j" is completed last in Π^*. From (1), (3), and (4), we have

$$t_{j'} \le T_{j"} + T_{j"} = 2\,T_{j"}. \qquad (5)$$

Inequalities (2) and (5) imply that

$$\phi_k = f_{j'}(t_{j'})$$

$$\leq \quad f_{j'}(2T_{j''}) \qquad \text{from (5)}$$
$$\leq \quad 2\, f_{j'}(T_{j''}) \qquad \text{from subadditivity, and}$$
$$\leq \quad 2\, \phi_k^* \qquad \text{from (2).} \qquad (6)$$

Hence, the on-line reoptimization strategy is 2-competitive. ■

Claim: The worst-case bound of 2 (proved in Theorem 1) is tight.

The following example justifies this claim. Consider a 3-job problem instance with linear completion time penalties $f_j(t) = \beta_j\, t$. Job 1 is unrelated, while job 2 precedes job 3. Jobs 1 and 2 arrive at time 0, each requiring 10 units of processing time, and job 3 arrives at time 10 with a very small processing time. The jobs have the following ordering in terms of penalty functions: $f_3 > f_1 > f_2$. In our notation, the following parameters describe this problem instance: $r_1 = 0$, $r_2 = 0$, and $r_3 = 10$; $p_1 = 10$, $p_2 = 10$, and $p_3 = \varepsilon$; $B_3 = \{2\}$; and $\beta_1 = 1$, $\beta_2 = 0$, and $\beta_3 = 100$.

At time 0, both jobs 1 and 2 are available, but job 1 has higher penalty. Hence, the on-line method processes it first. The next epoch is at time 10, when job 3 arrives (and job 1 completes). At this time, the on-line method starts job 2 (to satisfy job 3's precedence constraint), and finally completes job 3 at time $(20 + \varepsilon)$. Job 3 is the critical job, with a completion time penalty of $100*(20+\varepsilon)$. On the other hand, the optimal off-line scheduling method anticipates job 3's higher penalty and delayed arrival at $t = 10$. Hence, the optimal off-line sequence is 2–3–1. Job 3, completed at $(10+\varepsilon)$, is again the critical job, with a penalty of $100*(10+\varepsilon)$. Thus, the ratio of on-line to off-line penalty values is $(20+\varepsilon)/(10+\varepsilon)$ which approaches 2 as ε approaches 0. Hence, the worst-case ratio of 2 is tight. Next, we show that, for the non-preemptive case, the worst-case ratio is higher.

5.1.2 Scheduling without preemptions

As we noted earlier, finding the optimal off-line schedule with no preemption is computationally intractable. Hence, unlike the preemptive case, we do not have a convenient characterization of the optimal off-line schedule. To evaluate the competitiveness of on-line reoptimization for the

non-preemptive case, we use the following strategy. We know that the optimal off-line preemptive schedule has a lower penalty than the optimal off-line non-preemptive schedule. Hence, if we can derive a worst-case ratio for the on-line, non-preemptive schedule with respect to the optimal, off-line preemptive schedule, this ratio should also hold for the off-line non-preemptive solution. Let ρ denote the maximum, over all job pairs i and j, of the ratio of processing time for job j to the sum of processing times for job i and all its ancestors, i.e.,

$$\rho \quad = \quad \max \{p_j/(p_i + \sum_{l \in A_i} p_l\} : 1 \le i,j \le k, i \ne j\}.$$

We refer to ρ as the *Aspect ratio*. Note that the denominator in the above expression is a lower bound on the earliest possible completion time of job i. Indeed, our competitiveness result applies even if replace the denominator of ρ with a tighter lower bound (on job i's completion time) involving, say, the release times of job i and its ancestors.

Theorem 2:

For the $1/prec, r_j/f^{SA}_{max}$ problem, the on-line reoptimization method is $(\rho+2)$-competitive.

Proof:

This proof is very similar to the proof for Theorem 1. Let Π_{np} be the on-line, non-preemptive schedule for a k-job problem. Let ϕ^{np}_k be the (maximum) penalty of this schedule, defined by the critical job j'. Consider the interval of time $[r_{j'}, t_{j'}]$ between the arrival of job j' and its completion in schedule Π_{np}. Let u be the job that is currently in process in Π_{np} when job j' arrives, and let J' be the set of all jobs with higher penalties than j' that are completed in the time interval $[r_{j'}, t_{j'}]$. Except for the in-process job u, all other jobs that are completed in this interval either belong to J' or are ancestors of one or more jobs in J'. Let A(J') be the set of all ancestors $j \notin J'$ for jobs in J'. As before, let Π^* be the optimal, off-line preemptive schedule; job $j'' \in J'$ is scheduled last in Π^* among all jobs of J'. T_j denotes the completion time for job j in Π^*, and ϕ^*_k is the shcedule's penalty value. The job completion times in the on-line and off-line schedules must satisfy the following inequalities:

$$t_{j'} \leq r_{j'} + p_u + \sum_{j \in A(J')} p_j + \sum_{j \in J'} p_j ; \quad (7)$$

$$T_{j''} \geq \max \{r_{j'}, \sum_{j \in A(J')} p_j + \sum_{j \in J'} p_j \}; \text{ and} \quad (8)$$

$$\phi_k^* \geq f_{j''}(T_{j''})$$

$$\geq f_{j''}(\max \{r_{j'}, \sum_{j \in A(J')} p_j + \sum_{j \in J'} p_j \}). \quad (9)$$

These inequalities imply that:

$$\phi_k^{np} = f_{j''}(t_{j''})$$

$$\leq f_{j''}(r_{j''}) + f_{j''}(\rho\, p_{j''}) + f_{j''}(\sum_{j \in A(J')} p_j + \sum_{j \in J'} p_j)$$

using subadditivity and (7)

$$\leq \phi_k^* + \rho\, \phi_k^* + \phi_k^* \quad \text{using subadditivity and (9)}$$

$$= (\rho+2)\, \phi_k^*.$$

Thus, the worst-case ratio for the on-line, non-preemptive schedule is at most $(\rho+2)$ relative to the optimal off-line, preemptive schedule. Hence, the on-line reoptimization strategy is at least $(\rho+2)$-competitive for the $1/r_j$, prec, pmtn/f_{max}^{SA} problem. ■

Claim: The worst-case bound of $(\rho+2)$ for non-preemptive schedules is tight.

To prove this claim, consider the following augmented version of the previous worst-case problem instance (described after Theorem 1). In addition to the 3 jobs in that example, we have a fourth job that arrives at time $r_4 = 0$, with penalty coefficient $\beta_4 = 1$, and a processing time of $p_4 = 20$. Also, job 3 arrives at $r_3 = 10 + \delta$, for some small $\delta > 0$. Note that the Aspect ratio ρ for this problem instance is $p_4/p_1 = 2$. Consider, first, the schedule obtained using on-line reoptimization. Job 1 (or job 4) is scheduled first and completes at time 10. Since job 3 is not yet available, the method schedules job 4, followed by job 2 and finally job 3. Job 3 completes at time $(40 + \varepsilon)$; it is the critical job with a penalty of $100*(40 + \varepsilon)$. Contrast this on–line schedule with the following optimal, non-preemptive schedule: Job 2 starts at time 0 and completes at time 10; the processor is idle from time 10 to time $(10 + \delta)$; Job 3 starts at time $(10+\delta)$, and completes at time $(10+\delta+\varepsilon)$, followed by jobs 1 and 4. Again, job 3 is the critical job, with a completion time penalty value of

$100*(10+\delta+\varepsilon)$. Thus, the ratio of the on–line penalty and the optimal, off–line (non–preemptive) penalty approaches $(\rho + 2) = 4$ as δ and ε tend to zero.

5.2 The Lateness Penalty function

We now consider the *Lateness* objective L_{max}, i.e., the penalty for job j is $f_j(t_j) = t_j - d_j$, where t_j and d_j are, respectively, the completion time and due date for job j, and $L_{max} = \text{Max}\ \{ f_j(t_j): j = 1,2,...,n\}$. Our discussions focus on non–preemptive, precedence-constrained scheduling for this problem. The best off–line approximation algorithm for the $1/r_j/L_{max}$ problem was developed by Hall and Shmoys [1991]. They show that, for the delivery-time formulation of this model (described later), a 4/3 approximation algorithm is possible using an enhanced version of Jackson's earliest due date rule. Although Jackson's rule can be applied on–line, the approximation techniques used by Hall and Shmoys sacrifice the on-line characteristic to achieve the tighter bounds.

To study on-line competitiveness, we cannot work directly with the maximum lateness objective L_{max} since some problem instances may have zero or negative optimal off-line L_{max} values. (If the off-line L_{max} is zero, the worst-case competitiveness becomes unbounded for any on-line algorithm that is even slightly suboptimal.) Instead of redefining competitiveness, we transform the L_{max} problem to the following equivalent delivery time version (Potts [1980]) which has a positive optimal value for all problem instances.

5.2.1 *The Delivery Time Formulation*

Let $\{d_j\}$ denote the job due dates, and let K be a value greater than the largest due date. Now define the *tail* q_j of job j as

$$q_j = K - d_j .$$

We interpret the tail q_j as the time to deliver the job j after it is completed. Thus, the *delivery time* of job j is $t_j + q_j$, where t_j is the completion time of job j. The objective of the scheduling problem now consists of minimizing the maximum delivery time over all jobs. The optimal schedule for this delivery

time version also minimizes maximum lateness. Observe that the penalty function $f(t_j) = t_j + K - d_j$ implied by the delivery time objective function is consistent (according to our definition in Section 2), with jobs that are due earlier having higher penalty functions. Thus, at each job arrival epoch, on-line reoptimization for the delivery time problem (without preemptions) involves adjusting the current schedule to process the available job with the earliest due date as soon as possible (subject to completing the current in-process job and all unfinished ancestors of the EDD job).

5.2.2 *2-competitiveness of On-line Reoptimization*

For any given k-job instance of the non-preemptive delivery time problem, let Π be the schedule obtained using on-line reoptimization. Let DT_j be the delivery time of job j in this schedule; ϕ_k is the maximum delivery time value over all jobs j. Denote the (maximum) delivery time of the optimal, off-line schedule Π^* as ϕ_k^*.

Theorem 3:

On-line reoptimization is 2-competitive for the non-preemptive delivery time problem, i.e., $\phi_k \leq 2\,\phi_k^*$ for all k.

Proof:

We prove this result by induction. Assume that jobs are indexed in the order in which they arrive, and first consider k = 2. Since delivery time for a job equals the sum of its start time, processing time, and tail, the optimal off-line value ϕ_k^* must be greater than or equal to the maximum processing time over the k jobs. In the two-job case, if both jobs arrive simultaneously, then the on-line updating method also constructs the optimal schedule. When the release times are different, say, job 1 arrives before job 2, the on-line method schedules job 1 before job 2. Suppose the optimal off-line solution consists of processing job 2 before job 1. Relative to this optimal schedule, the on-line schedule delays job 2 by at most p_1. Hence, job 2 incurs an incremental delivery time penalty of at most p_1 relative to its penalty in the optimal off-line schedule Π^*. Furthermore, job 2's penalty in Π^* is a lower bound on the optimal off-line objective function value ϕ_2^*. Hence,

$$\phi_2 \leq \phi_2^* + p_1 \leq \phi_2^* + \max\{p_1, p_2\}$$
$$\leq 2\,\phi_2^*.$$

Thus, the on-line method is 2-competitive for 2 jobs. Now suppose the method is 2-competitive for all $k' \leq (k-1)$ jobs. We show that it must also be 2-competitive for k jobs.

Let job $j' \leq k$ be the critical job in the on-line schedule, i.e.,

$$\phi_k = DT_{j'} = s_{j'} + p_{j'} + q_{j'},$$

where s_j is the start time for job j in the on-line schedule.

Case 1: $s_{j'} < s_k$.

In this case, job j' starts earlier than the new job k but also has the highest delivery time. Hence, job j' must have a higher penalty function (i.e., earlier due date) than job k. Since the Forward algorithm leaves all higher penalty jobs unaffected when job k arrives, job j' must start at $s_{j'}$ even in the (k–1) job on-line schedule. Hence,

$\phi_{k-1} \geq s_{j'} + p_{j'} + q_{j'} = \phi_k$.

But, $\phi_{k-1}^* \leq \phi_k^*$, and

$\phi_{k-1} \leq 2\,\phi_{k-1}^*$ by the induction hypothesis.

Hence, $\phi_k \leq 2\,\phi_k^*$.

Case 2: $s_{j'} \geq s_k$.

In this case,

$$\phi_k^* \geq r_{j'} + p_{j'} + q_{j'}$$
$$= (s_{j'} + p_{j'} + q_{j'}) - (s_{j'} - r_{j'})$$
$$= \phi_k - (s_{j'} - r_{j'}).$$

Note however that

$$s_j - r_j \leq \sum_{i=1}^{j} p_i \text{ for all jobs j.}$$

Otherwise, the interval of time between the arrival and start of job j contains some idle time which is impossible using the on-line method (since job j is waiting in the queue). Therefore,

$$\phi_k^* \geq \phi_k - \sum_{i=1}^{k} p_i,$$

which implies that $\phi_k \leq 2\,\phi_k^*$.

Thus, on-line reoptimization is 2-competitive. ■

As before, we can show that the worst-case ratio of 2 (proved in Theorem 3) for on-line reoptimization is tight. Indeed, for the non-preemptive delivery time problem, the following example (Kise and Uno [1978], Potts [1980]) shows that any on-line scheduling algorithm that does not introduce forced idleness (i.e., does not keep the machine idle when the job queue is not empty) must have a worst-case ratio of at least 2. Consider a problem instance with two jobs that are released respectively at $r_1 = 0$ and $r_2 = 1$, having processing times $p_1 = (P - 1)$ and $p_2 = 1$, and due dates $d_1 = P$ and $d_2 = 1$ (hence, the tails are $q_1 = 0$ and $q_2 = (P - 1)$). With no precedence constraints, the best strategy consists of keeping the machine idle for the first time period, and scheduling job 2 before job 1. The maximum delivery time for this solution is $(P + 1)$. On the other hand, any on-line method that does not anticipate future job arrivals or introduce forced idleness will schedule job 1 at time 0 (since it is the only available job). Since jobs cannot be preempted, job 2 begins processing only at time $(P - 1)$, and its delivery time is $(2P - 1)$. As P becomes arbitrarily large, the ratio of on-line to optimal off-line delivery time approaches 2. And, on–line reoptimization achieves this lowest possible worst-case ratio.

In retrospect, the 2-competitiveness of on–line reoptimization is not surprising in view of the worst-case bound of 2 for Schrage's heuristic (Potts [1980]). For the $1/r_j/DT_{max}$ delivery time problem (without preemption or precedence constraints), Schrage's heuristic consists of applying Jackson's earliest due date rule on–line (with a longest processing time tie-breaking rule), i.e., whenever a job completes, the method dispatches the currently available job with the earliest due date. Note that, without precedence constraints, on–line reoptimization also chooses this "current" EDD job sequence. Potts [1980] used a characterization of Schrage's heuristic schedule (in terms of an *interference* job) to prove that the method has a worst-case bound of 2. When jobs have precedence constraints, we can transform the delivery time problem $1/r_j, prec/DT_{max}$ to an equivalent unconstrained version by revising the job release times and tails as follows: if job i must

precede job j, set $r_j \leftarrow \max\{r_i, r_j\}$ and $q_i \leftarrow \max\{q_i, q_j+p_j\}$. Potts' result implies that Schrage's heuristic applied to this transformed problem is 2-competitive. Note that, when a new job arrives, updating the tails (to account for its precedence constraints) involves examining every currently available ancestor of the new job. In contrast, our on–line updating algorithm (Appendix 1) is more efficient since it first locates the new job relative to the current bottleneck jobs, and only examines ancestors that are scheduled later.

6. Conclusions

In this paper we have developed a new Forward algorithm and an updating version for one class of scheduling problems. The updating procedure reduces the computational effort to accomodate a new job into an existing schedule by using information from the current schedule. In contrast, applying a zero-base algorithm to reschedule all the jobs from scratch would entail significantly higher computational effort, as illustrated by our computational results of Section 4. Section 5 gives partial results characterizing the effectiveness of using deterministic reoptimization for on-line scheduling.

Our broader purpose in this paper is to demonstrate the scope, and efficiency and effectiveness issues in developing on-line updating algorithms for dynamic scheduling problems. In spite of their practical importance in contexts such as real-time control of distributed processors, updating algorithms have not been adequately studied in the scheduling literature. For illustrative purposes, we studied a single machine scheduling problem that can be solved efficiently. Exploring similar updating methods for other scheduling objectives and contexts is an important research direction that merits further investigation.

Appendix 1

FORWARD Algorithm for the 1/prec, r_j, pmtn/f_{max} problem with Consistent Penalties

The Forward algorithm described in Section 3 assumes that all jobs are simultaneously available at time 0. This Appendix describes an extension to handle arbitrary, but known, job release times; job preemption is permitted.

First, we review the notation. We are given n jobs, and a precedence graph G:(N,A) containing m arcs. B_j and A_j represent, respectively, the set of immediate predecessors and ancestors of job j. Each job has a "consistent" non-decreasing penalty function $f_j(t)$, i.e., either $f_j(t) < f_i(t)$ or $f_j(t) > f_i(t)$ for all completion times t, and $f_j(t) \geq f_j(t')$ if $t > t'$. Let p_j and r_j denote, respectively, the processing time and release time for job j.

Without loss of generality, we assume that all ancestors of any job j arrive at or before job j; otherwise, we can set $r_j = \text{Max}\{r_i + p_i: i \in B_j\}$. Also, for convenience, assume that jobs are indexed in order of release dates; consequently, $i < j$ if job i precedes job j. We require a preemptive schedule that minimizes $f_{max} = \text{Max}\{f_j(t_j): j = 1,2,...,n\}$ while satisfying the precedence constraints and release times, where t_j is the completion time for job j in the chosen schedule.

Scheduling Principle:

As before, the Forward algorithm identifies successive bottleneck jobs, and schedules each bottleneck job as early as possible. As before, we first sort all jobs in decreasing order of penalties. The method starts with an empty schedule, and progressively assigns jobs to appropriate "free" time intervals in the current schedule. At iteration r, the method has scheduled the first (r-1) bottleneck jobs and all their ancestors. Let I^r denote the set of available free time intervals in the current schedule at the start of iteration r. The following steps are performed at iteration r:

Step 1: Identify the r^{th} bottleneck job $j^*(r)$, i.e, job $j^*(r)$ has the largest penalty among all the currently unscheduled jobs;

Step 2: Consider each unscheduled ancestor j of job $j^*(r)$ in increasing index order: allocate to job j the first available p_j time units after the release time r_j. Update the available free intervals.

Step 3: Allocate to job $j^*(r)$ the first available $p_{j^*(r)}$ time units after the release time $r_{j^*(r)}$.

The algorithm iteratively repeats this process—finding the next bottleneck job, and scheduling this job and all its ancestors as early as possible in increasing index order. Note that as we identify and schedule more bottleneck jobs, the schedule becomes fragmented, i.e., it has "holes" due to delayed releases for certain jobs. These holes are filled whenever possible in subsequent steps; filling the holes might introduce preemptions, i.e., the total processing time of a job may be distributed over several intervals.

Let Π be the final schedule constructed by the Forward algorithm, and denote the set of bottleneck jobs as J_B.

Lemma:

In the final schedule constructed by the Forward algorithm,

$$f_{max} = \text{Max} \{ f_j(t_j) : j \in J_B \}.$$

Note that this property also holds for the original model when all jobs are simultaneously available at time 0.

Proof of correctness of the Forward Algorithm:

Suppose the schedule Π constructed by the Forward algorithm is not optimal. Let j^* be the *critical* job that determines the penalty of this schedule, i.e., $f_{max} = f_{j^*}(t_{j^*})$, and suppose j^* is the r^{*th} bottleneck job, i.e., $j^* = j^*(r^*)$. Let Π' be an optimal schedule; let f'_{max} and t'_j denote, respectively, the maximum penalty and the completion time for each job j in this schedule.

By the hypothesis, $f'_{max} < f_{max} = f_{j^*}(t_{j^*})$. Since penalty functions are non-decreasing, this inequality implies that job j^* must be scheduled earlier in Π', i.e., $t'_{j^*} < t_{j^*}$. Since the Forward algorithm schedules all bottleneck jobs as early as possible in order of decreasing penalties, t'_{j^*} can be less than t_{j^*} only if the schedule Π' completes some previous bottleneck job (i.e., a bottleneck job with a higher penalty than job j^* and scheduled before t_{j^*} in the Forward schedule Π) on or after t_{j^*}. Let $j^*(r)$ for some $r < r^*$ denote this bottleneck job. Since $t'_{j^*(r)} > t_{j^*}$, we must have $f'_{max} \geq f_{j^*(r)}(t_{j^*})$. However, since the job $j^*(r)$ has a higher penalty than j^*, $f_{j^*(r)}(t_{j^*}) > f_{j^*}(t_{j^*}) = f_{max}$ contradicting the optimality of schedule Π'. ■

References

Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1974) *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts.

Baker, K. R., E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan (1983) "Preemptive scheduling of a single machine to minimize maximum cost subject to release dates and precedence constraints", *Operations Research*, **31**, pp. 381-386.

Borodin, A., N. Linial, and M. Saks (1987) "An optimal on-line algoirthm for metrical task systems", *Proc. of 19th ACM Symposium on Theory of Computing*, pp. 373-382.

Chin, F., and D. Houck (1978) "Algorithms for updating minimum spanning trees", *Journal of Computing Systems Sciences*, **16**, pp. 333-344.

Chung, F. R. K., R. L. Graham, and M. E. Saks (1989) "A dynamic location problem for graphs", *Combinatorica*, **9**, pp. 111-131.

Even, S., and Y. Shiloach (1981) "An on-line edge deletion problem", *Journal of the Association of Computing Machinery*, **28**, pp. 1-4.

Frederickson, G. N. (1985) "Data structures for on-line updating of minimum spanning trees with applications",*SIAM Journal on Computing*, **14**, pp. 781-798.

Frederickson, G. N., and M. A. Srinivas (1984) "On-line updating of degree-constrained minimum spanning trees", *Proceedings of the 22nd Allerton Conference on Communication, Control, and Computing, October 1984.*

Hall, L. A., and D. Shmoys (1991) "Jackson's rule: Making a good heuristic better", to appear in *Mathematics of Operations Research.*

Jackson, J. R. (1955) "Scheduling a production line to minimize maximum tardiness", Research report 43, Management Science Research Project, University of California, Los Angeles.

Kise, H., and M. Uno (1978) "One-machine scheduling problems with earliest start and due time constraints", *Mem. Kyoto Tech. Univ. Sci. Tech.*, **27**, pp. 25-34.

Lawler, E. L. (1973) "Optimal sequencing of a single machine subject to precedence constraints", *Operations Research*, **26**, pp. 544-546.

Lawler, E. L., J. K. Lenstra, and A. H. G. Rinnooy Kan (1982) "Recent developments in deterministic sequencing and scheduling: A survey", in *Deterministic and Stochastic Scheduling*, M. A. H. Dempster, J. K. Lenstra, and A. H. G. Rinnooy Kan (eds.), Riedel, Dordrecht.

Malone, T. W., R. E. Fikes, K. R. Grant, and M. T. Howard (1988) "Enterprise: A market-like task scheduler for distributed computing environments", in *The Ecology of Computation*, B. A. Huberman (ed.), Elsevier Science Publishers B. V., Amsterdam, Holland, pp. 177-205.

Manasse, M. S., L. A. McGeoch, and D. D. Sleator (1988) "Competitive algorithms for on-line problems", *Proc. 20th ACM Symposium on Theory of Computing*, pp. 322-333.

Overmars, M. H., and J. van Leeuwen (1981) "Maintenance of configurations in the plane", *Journal of Computing System Sciences*, **23**, pp. 166-204

Potts, C. N. (1980) "Analysis of a heuristic for one machine sequencing with release dates and delivery times", *Operations Research*, pp. 1436-1441.

Ramamritham, K., and J. A. Stankovic (1984) "Dynamic task scheduling in distributed hard real-time systems", *IEEE Software*, **1**, 96-107.

Sahni, S., and Y. Cho (1979) "Nearly on line scheduling of a uniform processor system with release times", *SIAM Journal on Computing*, **8**, pp. 275-285.

Shmoys, D., J. Wein, and Williamson (1991) "On-line scheduling of parallel machines", preprint.

Spira, P. M., and A. Pan (1975) "On finding and updating spanning trees and shortest paths", *SIAM Journal on Computing*, **4**, pp. 215-225.

Tarjan, R. E. (1983) *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania.

Zhao, W., and K. Ramamritham (1985) "Distributed scheduling using bidding and focused addressing", *Proceedings of the Symposium on Reat-time Systems*, December 1985, pp. 103-111.

Figure 1
Consistent and Non-consistent Penalty Functions

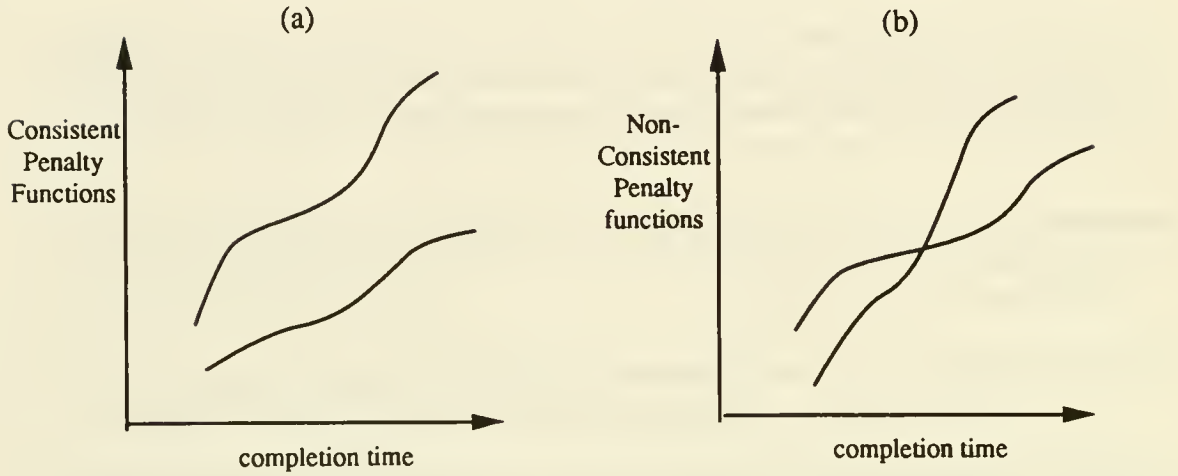
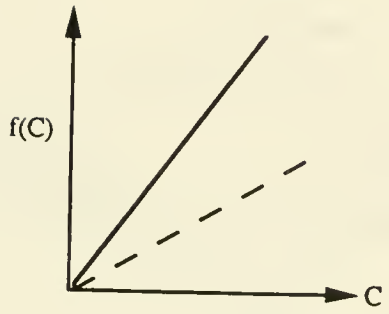
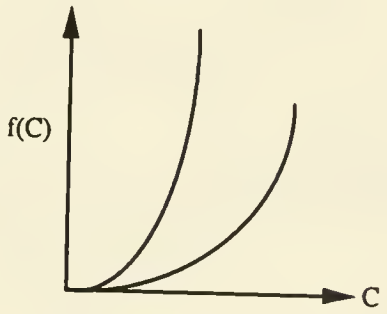


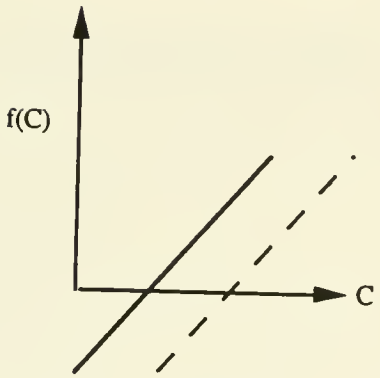
Figure 2
Special Consistent Penalty Functions



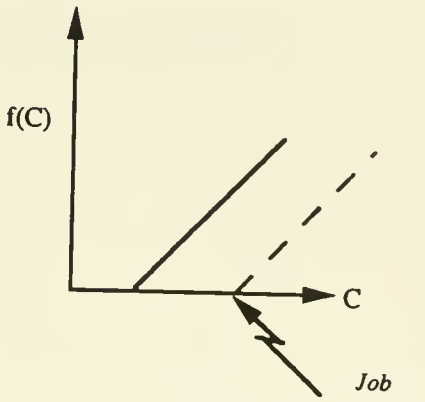
Weighted Linear Completion Time Penalty



Weighted Quadratic Completion Time Penalty

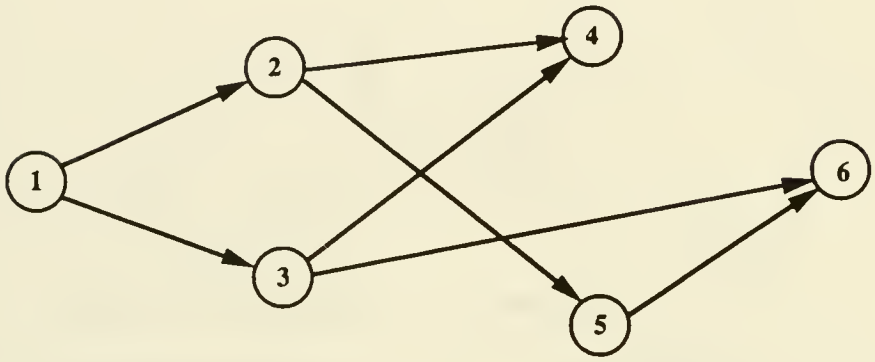


Lateness Penalty



Tardiness Penalty

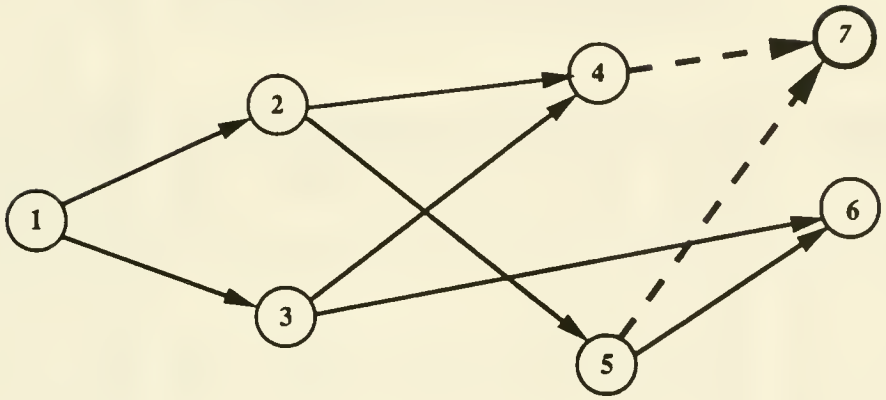
Figure 3
Example for Forward Zero-Base Algorithm



Ranking of jobs in order of decreasing penalties : 5, 1, 6, 3, 2, 4

n = number of jobs = 6
m = number of arcs in precedence graph = 7

Figure 4
Example for Forward Updating Algorithm



Ranking of jobs in order of decreasing penalties : 5, 1, 7, 6, 3, 2, 4

Updated optimal schedule: 1 - 2 - 5 - 3 - 4 - 7 - 6

Table 1
Forward Zero-Base Algorithm:
Iterations for 6-node example

Iteration k	Unscheduled Jobs S_k	Bottleneck Job $j^*(k)$	Unscheduled Ancestors $A_{j^*(k)}(S_k)$
1	1, 2, 3, 4, 5, 6	5	1, 2
2	3, 4, 6	6	3
3	4	4	—

Optimal schedule: 1 - 2 - 5 - 3 - 6 - 4

Table 2: **Computation times (Average & Standard Deviation) for Updating versus Zero-base Algorithm**

Arc Density δ	Number of Jobs (n)							
	n = 100		n = 200		n = 300		n = 400	
	Zero-base	Update	Zero-base	Update	Zero-base	Update	Zero-base	Update
0.90	24.9† (10.7)§	5.0 (7.7)	76.7 (13.0)	25.3 (8.9)	141.6 (19.6)	58.6 (9.0)	222.0 (27.8)	102.4 (12.6)
0.75	27.6 (7.8)	6.7 (8.3)	76.3 (12.1)	23.4 (9.5)	138.7 (19.6)	58.3 (9.7)	219.0 (24.3)	103.4 (8.9)
0.50	24.6 (8.3)	5.7 (8.0)	73.9 (14.3)	29.9 (11.2)	132.7 (17.5)	56.0 (8.7)	212.1 (23.8)	100.0 (7.5)
0.25	23.7 (9.5)	7.7 (8.4)	66.3 (10.4)	26.4 (8.9)	124.7 (11.7)	55.7 (9.4)	200.0 (22.6)	102.3 (10.1)
0.10	24.4 (9.0)	5.7 (8.1)	66.3 (11.8)	25.3 (8.2)	123.9 (12.1)	58.1 (10.2)	186.0 (15.5)	104.0 (13.8)

† Average CPU time (in milliseconds on a SUN 4/90) over 100 random problem instances;

§ Standard deviation of CPU times