An Almost-Linear Algorithm for Two-Processor Scheduling

HAROLD N. GABOW

University of Colorado at Boulder, Boulder, Colorado

Abstract. A well-known problem in scheduling theory is to execute n unit-length jobs subject to precedence constraints on two processors in minimum finish time Previous algorithms begin by finding the transitive closure of the precedence dag and so use time $O(\min(en, n^{2.61}))$. An $O(e + n\alpha(n))$ algorithm is presented which is based on the idea of a "highest-level-first" (HLF) schedule Such a schedule always executes nodes on the longest paths of the precedence dag An HLF schedule is guaranteed to be optimum and can be constructed efficiently

Categories and Subject Descriptors: D.4 1 [**Operating Systems**] Process Management—*Scheduling*; F 2.2 [**Analysis of Algorithms and Problem Complexity**] Nonnumerical Algorithms and Problems—*computations on discrete structures, sequencing and scheduling*, G 2 2 [**Discrete Mathematics**] Graph Theory—*graph algorithms*

General Terms. Algorithms, Theory

Additional Key Words and Phrases Precedence constraints, critical path scheduling, highest-level-first scheduling, directed acyclic graphs

1. *Introduction*

A classic problem in scheduling theory is to find a minimum makespan, nonpreemptive schedule for a collection of unit-length jobs subject to precedence constraints [3]. In other words, we are given n jobs to be executed on m processors. Each job requires exactly one unit of execution time and can run on any processor. A directed acyclic graph *(dag)* specifies the precedence constraints: an edge from x to y means job x must be completed before job y begins. A schedule with the smallest overall finish time is sought. (In a more succinct notation, this problem is $P/prec, p_j = 1/C_{max}$ [10].)

When the number of processors m is arbitrary, this problem is NP-complete [19]. For any fixed $m \geq 3$, the complexity is open [9]. Here we study the tractable case $m = 2$.

For two processors a number of efficient algorithms have been given. A common aspect of these algorithms is that they require the dag to be transitively closed (or, in one case, transitively reduced). Otherwise the dag must be put into transitively closed form (or transitively reduced form). The best known algorithms for this use time $O(\min(en, n^{2.61}))$, where e is the number of edges of the dag. (The transitive closure can be computed in time $O(en)$ by n depth-first searches. Alternatively, it can be reduced to matrix multiplication [2], which is $O(n^{2.61})$ [16]. The transitive reduction

This research was supported in part by the National Science Foundation under Grant MCS 78-18909

Author's address. Department of Computer Science, University of Colorado, Boulder, CO 80309

requires basically the same time [1].) As we see below, this first step of putting the dag into a specialized form dominates the run time.

The two-processor algorithm of Fujii et al. [5] is based on matching techniques. Excluding transitive closure time, it requires the time to find a maximum matching, which in this case is $O(n^{2.5})$ [12, 15]. Coffman and Graham [4] give another algorithm based on a lexicographic numbering scheme. It works on a transitively closed or transitively reduced dag. Sethi [17] shows that the lexicographic numbering can be done in time $O(e + n\alpha(n))$.[1] Garey and Johnson [8] give a third algorithm, which allows individual job deadlines in addition to precedence constraints. It can be used to solve our problem. The algorithm uses time $O(n^2)$ to compute "modified deadlines." It works on a transitively closed dag.

Time bounds for these algorithms often assume that the transitive closure of the dag (or transitive reduction) is given. In practice this is unlikely. On general dags the transitive closure step dominates, and the algorithms use time $O(\min(en, n^{2.61}))$ ($O(n^{2.61})$ for [5]).

We present an algorithm that does not use the transitive closure and runs in time $O(e + n\alpha(n))$ on an arbitrary dag. It is based on the idea of a "highest-level-first" (HLF) schedule. Such a schedule always executes nodes on the longest paths of the dag. An HLF schedule is guaranteed to be optimum and can be constructed efficiently.

Section 2 gives a precise definition of an HLF schedule. Section 3 gives the algorithm and its analysis. Section 4 briefly discusses applications of the algorithm to other scheduling problems.

2. *HLF Schedules*

This section gives some basic terminology and introduces the notion of HLF schedules.

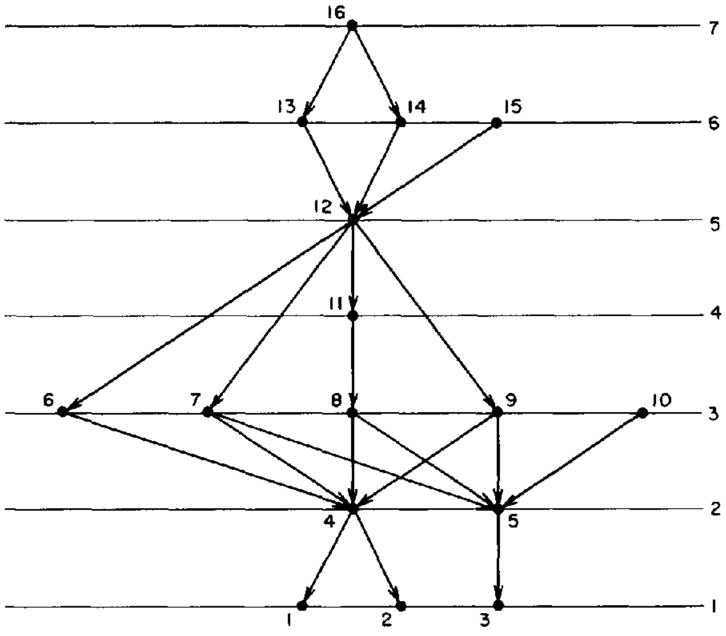
A scheduling problem is defined by a *dag* (directed acyclic graph). The number of nodes and edges are denoted by n and e, respectively. If there is an edge from node x to node y, then x is an *immediate predecessor* of y and we write $x \rightarrow y$; if there is a directed path (of zero or more edges) from x to y, then x is a *predecessor* of y, y is a *successor* of x, and we write $x \xrightarrow{*} y$. A dag can be partitioned into *levels* i, $L \geq i \geq 1$: level i consists of all nodes x that start paths with i nodes but not paths with $i + 1$ nodes; we write $level(x) = i$. L denotes the highest level of the dag. Figure 1a gives an example dag.

A (*two-processor*) *schedule* is an assignment of the nodes of the dag to time units, $1, 2, \ldots, \omega$, so that each node is assigned to exactly one time unit, at most two nodes are assigned to the same time unit, and $x \rightarrow y$ implies that x is assigned to a lower time unit than y. The schedule *executes* (or *schedules*) node x during time unit i if x is assigned to i. ω is the *finish time* or *makespan* of the schedule. An *optimum schedule* minimizes ω. Such a schedule has the fewest possible number of *idle* time units, that is, time units assigned only one node. Figure 1b gives a schedule as a Gantt chart (the ith column shows the nodes executed in time unit i).

We define HLF schedules in two steps. First, a *level schedule* "executes levels" in the order $L, L - 1, \ldots, 1$ (recall that L is the highest level). More precisely, suppose levels $L, \ldots, i + 1$ have already been executed, and level i contains u unexecuted nodes. Then level i is executed in the next $\lceil u/2 \rceil$ time units, as follows. The first $\lfloor u/2 \rfloor$ units each execute two nodes of i. If u is even, this completes the execution of level i. Otherwise, u is odd, and the $\lceil u/2 \rceil$nd unit executes the last node x of i and

[1] $\alpha(n)$ is an inverse of Ackermann's function and is very slow-growing [18]

(a)



(b)

16	14	12	11	8	6	4	2
15	13	10	9	7	5	3	1

FIG 1 (a) Example dag (b) Schedule

possibly (but not necessarily) a node y of a lower level. This completes the execution of level i. Figure 1b shows a level schedule.

It is convenient to introduce some auxiliary terms for level schedules. If for a level i the above quantity u is odd, then i is a *1-level*. In this case (x, y), the ordered pair formed from the two nodes executed in the last time unit for i, is a *jump*. (Note that level(x) $= i$ and level(y) $< i$.) We say the jump goes *from* x *to* y. (Alternatively, the jump goes *from* level(x) *to* level(y), or level(x) *jumps* y.) Note that since there can be jumps to level i, the number u can be less than the number of nodes originally on level i.

In case i is a 1-level and the node y does not exist (i.e., the $\lceil u/2 \rceil$nd time unit executes only a node x on level i), then y is taken to be a dummy node 0. So $(x, 0)$ is the jump from i. Node 0 is on a fictitious level 0. As such, it is below all other nodes. By convention a level schedule can make an arbitrary number of jumps to node 0.

An optimum schedule that is level always exists. This is a consequence of the fact that the Coffman–Graham algorithm produces such a schedule [4]. Alternatively, one can prove this directly by transforming an arbitrary optimum schedule to be level. For example, suppose in some optimum schedule, for $i = 1, 2$, x_i is scheduled with y_i, where x_i is a node on the highest level L but y_i is not. Then x_1 and x_2 can be scheduled together. y_1 and y_2, and other nodes, can each be scheduled with a successor of x_1 or x_2. The basic principle is that if level(x) $>$ level(y) and $x \not\to y$, then x has a successor z on level(y), $z \neq y$; z and y can be scheduled together. Proceeding this way, we get an optimum schedule that begins by executing x_1 and x_2. Repeating this transformation gives an optimum, level schedule. Details of this proof are in [6].

Loosely speaking, we wish to define an HLF schedule as a level schedule that always jumps to the highest level possible. This seems like a desirable characteristic, since it is consistent with the critical path heuristic, that is, always execute nodes on the longest path of the dag. This heuristic guarantees an optimum schedule when there are m processors and the dag is a tree [11].

To give a rigorous definition of HLF, first consider an arbitrary level schedule. Let the 1-levels be $f_1 > f_2 > \cdots > f_k$, and let level f_i jump to level t_i (recall that $t_i = 0$ if no real node is jumped from f_i). The *jump sequence* of the schedule is the ordered k-tuple $(t_1, t_2, \ldots, t_k)$. The jump sequence for Figure 1 is (6, 3, 3, 2, 1). Note that the levels f_i can be deduced from the jump sequence and the dag (a level f is a 1-level if the number of nodes on f, minus the number of occurrences of f in the jump sequence, is odd.) Note also that the jump sequence determines ω. (The number of 0's in the jump sequence is the number of idle time units.)

Jump sequences are compared using lexicographic order. Thus $(t_1, \ldots, t_k) > (s_1, \ldots, s_r)$ if for some j, $1 \le j \le \min(k, r)$, $t_i = s_i$ for $1 \le i < j$ and $t_j > s_j$. (Note that lexicographic order allows the possibility that $(t_1, \ldots, t_k) > (s_1, \ldots, s_r)$ if $t_i = s_i$ for $1 \le i \le r$ and $k > r$. However, this cannot occur with jump sequences: if $t_i = s_i$ for $1 \le i \le r$, then $k = r$ and $(t_1, \ldots, t_k) = (s_1, \ldots, s_r)$.)

Now define a *highest-level-first* (*HLF*) schedule as a level schedule whose jump sequence is as large as possible. Such a schedule always jumps to the highest level possible; when there is a choice of nodes to jump on that level, it jumps the node that allows subsequent jumps to be highest. Figure 1 shows an HLF schedule.

Any HLF schedule is optimum. This is proved in Section 3 in the analysis of the algorithm. However, to motivate the algorithm, we indicate here why HLF schedules are optimum.

First, this fact can be proved directly by transforming an arbitrary optimum level schedule to be HLF. This elaborates on the transformation, mentioned above, of an optimum schedule to a level schedule. Details are in [6].

Second, we can make a simple plausibility argument. Suppose f is a 1-level that can conceivably jump nodes $y_1, \ldots, y_k$. Then each y_i can be jumped from any 1-level g, $f \ge g > \text{level}(y_i)$. If f jumps the highest y_i, it preserves the greatest number of other nodes y_j for lower 1-levels g. For example, suppose f can jump y_1 and y_2, where $\text{level}(y_1) > \text{level}(y_2)$. Jumping y_1 from f allows a level g, $g > \text{level}(y_2)$, to jump y_2. On the other hand, jumping y_2 from f destroys this option. Since a 1-level g with $\text{level}(y_1) > g$ cannot jump y_1, this choice may lead to a suboptimum schedule. Figure 1 illustrates this: Level 7 can jump nodes 15 and 10. Jumping node 10, which is not the highest, forces levels 6 and 5 to have idle jumps, giving a suboptimum schedule.

This reasoning also shows that when there is a choice of nodes to jump on the highest level, the choice should be made so subsequent jumps can be highest. In other words, the schedule should be HLF.

Before moving on to the algorithm, note that our remarks on level and HLF schedules apply only to two-processor scheduling. For three or more processors there are dags that admit no optimum, level schedule. Consequently, it is not clear how the notion of an HLF schedule can be generalized. We return to this issue in Section 4.

3. *The Algorithm*

This section presents an algorithm that finds an optimum, HLF schedule in time $O(e + n\alpha(n))$.

We begin with some informal reasoning. The difficulty in constructing an HLF schedule arises when there is a choice of nodes to jump. For instance, in Figure 1, level 4 can jump a number of nodes on level 3: nodes 6, 7, 9, and 10. In general terms, suppose that the highest level to which some 1-level f can jump is t. If a number of nodes on t can be jumped, which one should be chosen? The HLF definition implies that the choice should be made to allow subsequent jumps to be highest. So consider a subsequent jump, from a 1-level g, $g < f$. We first claim that, surprisingly, the choice of node for f has *no* affect on the jump from g if $g > t$.

To see this, suppose that level f can jump any of the nodes $y_1, \ldots, y_k$ on level t; further, choosing y_1 makes some level s the highest level to which g can jump. If $s > t$, it is clear that g can jump to s when any arbitrary y_i is chosen. If $s = t$, choosing y_i instead of y_1 makes y_1 available, and again g can jump to s. Finally, suppose $s < t$. g cannot jump to any level above s, so g cannot jump to t. This implies that node y_i must have been jumped from a level above g, for $1 \leq i \leq k$. Clearly the levels above g will jump all nodes y_i, *regardless* of the choice y_1 for f. So in all cases the choice of node to jump for f has no effect on g, for $g > t$.

Thus the highest level affected by f's jump to t is t itself. The HLF definition implies that f should jump a node on t that allows t's jump to be as high as possible.

To understand what this means, observe that the nodes on t (or on any level) are of two types: those that must be jumped, called "nonfree," and those that need not be jumped, called "free." In Figure 1, node 10 is nonfree; node 9 is free, since level 4 can jump node 6 (or 7) instead of 9. (A rigorous definition of "free" is given below.)

The jump from level t can conceivably be made *from* any free node of t. So to ensure that the jump from t is as high as possible, there is a simple choice rule: level f should *not* jump the free node on t that is in the highest jump from t. For example, in Figure 1, for $f = 4$, $t = 3$, node 6 should not be jumped: it can jump to (node 5 of) level 2, while the other free nodes of level 3 cannot jump to level 2. (Recall that node 10 is nonfree.) This single rule guarantees an HLF schedule!

Now we can describe a two-pass procedure that finds an HLF schedule. Pass I computes the jump *from* each 1-level f, for $f = L, \ldots, 1$: It finds the highest level t to which f can jump. If level t has several nodes that can be jumped, it guesses one arbitrarily. The guesses may be incorrect (i.e., they may violate the above choice rule.) However, they allow Pass I to keep track of the nonfree and free nodes. Pass I always finds the best jump from a free node x of f. Pass II alters the schedule so that free nodes x that were incorrectly jumped do not get jumped.

This approach has a (very slight) drawback in terms of efficiency. The difficulty is in finding the highest level t to jump to. t changes arbitrarily with successive 1-levels. The priority queues of [20] can be used to find t. This gives an $O(e + n \log \log n)$ algorithm. This algorithm finds use in some related problems [7].

For greater efficiency the computation can be restructured. Pass I computes the jumps *to* level t, for $t = L, \ldots, 1$: For each node y on level t, it finds the highest 1-level f that has not been assigned a jump but can jump to y. It guesses that f jumps to y. As above, Pass I keeps track of the nonfree and free nodes and always computes jumps from free nodes. Pass II fixes bad guesses.

This second approach has the advantage of a simpler "highest level" computation. The first approach computes the highest level t to jump to; a given t may be highest at various, arbitrary times. The second approach computes the highest 1-level f to jump from; a given f is highest only once. (After its jump has been found, level f is no longer a candidate.) This allows the use of set merging techniques, giving an $O(e + n\alpha(n))$ algorithm.

It remains to give some details about how Pass I computes free nodes and how Pass II fixes bad guesses. In Pass I, consider a level t. Let the 1-levels that jump to t be $f_1 > f_2 > \cdots > f_k$. Choose r maximum, $0 \le r \le k$, so that the set of nodes on level t that are jumped from $f_1, \ldots, f_r$ cannot change (i.e., although it may be possible to vary the node jumped from a particular f_i, $i \le r$, the levels $f_1, \ldots, f_r$ must jump a fixed set of r nodes on level t.) If $r < k$, then it is not hard to see that there is a choice of node to jump for each f_i, $i > r$. More precisely, the nodes on t jumped from $f_{r+1}, \ldots, f_k$ are chosen from a set of more than $k - r$ nodes; furthermore, any node in that set need not be jumped. Thus the nodes jumped from $f_1, \ldots, f_r$ are the nonfree nodes of level t, and the remaining nodes are free.

Pass I computes the nonfree nodes by finding the levels $f_1, \ldots, f_r$. Note that the jumps that Pass I guesses for these levels do not change in Pass II. Pass I guesses the jumps for levels $f_{r+1}, \ldots, f_k$ in a way that makes it particularly easy for Pass II to fix bad guesses: it chooses a node z on level t that can be jumped from f_{r+1} and guesses jumps so z is *not* jumped. (This can be done by the definition of f_r.) Note that any level f_i, $r < i \le k$, can jump z. Now suppose it turns out that Pass I makes a bad guess. In other words, the best jump from t is (x, y), and Pass I guesses that some level f_i, $r < i \le k$, jumps node x. Pass II fixes this by rerouting the jump from f_i so it goes to z instead of to x.

Node z is called a "substitute node." For each level, Pass I computes the free and nonfree nodes and a substitute node. Pass II fixes the bad guesses of Pass I by changing jumps to go to substitute nodes. These changes may in turn cause further substitutions in Pass II.

Now we give a detailed description of the algorithm, beginning with the data structures. The schedule is specified in arrays FROM and TO. For $L \ge f \ge 1$, (FROM(f), TO(f)) is the jump from level f. (So FROM(f) and TO(f) are nodes, with level (FROM(f)) $= f$, level (TO(f)) $< f$.) There are two special cases: if TO(f) $= -1$, f is not a 1-level, and there is no jump from f; if TO(f) $= 0$, node FROM(f) is scheduled with an idle processor. Clearly these arrays give enough information to deduce the entire schedule (in linear time), if desired.

The FROM and TO arrays can be used to store both the jumps that Pass I guesses and the final jumps that Pass II computes. In an actual implementation this should be done. However, in the proof of correctness it is desirable to distinguish between guesses and final values. For this reason an array T is used to hold guessed TO-values. Pass I guesses the *to* nodes of jumps and stores them in T. Pass II copies T to TO, and then modifies TO to the final jumps.

Pass I partitions the levels of the dag into sets, as follows. A level f is called "open" if its jump has not been found. More precisely, when level t is being processed, level f is *open* if $f \le t$ or if $f > t$ and the jumps to f make it a 1-level but $T(f) = 0$ (Pass I initializes T-values to 0). Each open level f has a set of levels,

$$\text{LSET}(f) = \{g \mid L \ge g \ge f \text{ and } f \text{ is the highest open level with } g \ge f\}.$$

LSETs are manipulated by the operations FIND(g) (which returns the open level f with $g \in$ LSET(f)) and UNION(f, g) (which does a destructive merge of LSET(f) into LSET(g)) [2].

In processing level t, Pass I finds when each node y on level t is ready to be jumped; that is, it computes

$$R(y) = \text{the highest open level that can jump to } y.$$

It also finds which nodes may be jumped from a given open level f; that is, it computes the list

$$\text{RLIST}(f) = \{y \mid y \text{ is on level } t \text{ and } R(y) = f\}.$$

(These interpretations for R and RLIST are valid immediately before line 6 of the algorithm.)

Pass I computes the substitute node on level t, SUB(t). SUB(t) is not jumped in Pass I but is ready as early as possible (i.e., $R(\text{SUB}(t))$ is as large as possible). Any level $f \le R(\text{SUB}(t))$ can jump SUB(t) instead of $T(f)$. So Pass II can use SUB(t) to ensure that FROM(t) is not jumped, thus fixing bad guesses. This motivates the following definition.

Definition 1. A node y on level t is *free* if $y = T(f)$ implies that $f \le R(\text{SUB}(t))$, that is, either y is not jumped in Pass I, or y is jumped from $R(\text{SUB}(t))$ or below.

This definition is consistent with the earlier intuitive description of "free."

Note the special case in which SUB(t) = 0. Since the algorithm sets $R(0) = 0$, the only free nodes on such a level t are those that are not jumped. Also note that, at least intuitively, any level has free nodes, because not every node of a level can be jumped.

The algorithm works as follows. Pass I processes levels t in decreasing order, $t = L, \ldots, 1$. For each t, R and RLIST values are computed (lines 2–5). Then RLISTS are used to guess jumps, that is, T-values (lines 6–9). The node with highest R-value that need not be jumped does not get jumped; instead it is made SUB(t) (line 10). The method for finding SUB(t) relies on merging RLISTs so nodes with higher R-values are at the end (line 9).

Pass II processes levels f in increasing order, $f = 1, \ldots, L$. For each f, a correct node FROM(f) is found. If FROM(f) happens to be jumped by Pass I, the jump is switched to go to SUB(f) instead of FROM(f).

Now we give the algorithm in pseudo-ALGOL.

```
    procedure H, comment this procedure finds the jumps of an HLF schedule for a given dag,
    begin
    Initialization:
0.      partition the nodes of the dag into levels $L, \ldots, 1$, set SUB($t$) = 0, $T(t) = 0$, LSET($t$) = $\{t\}$,
           RLIST($t$) = $\varnothing$, for $L \ge t \ge 1$; set LSET(0) = $\{0\}$, $R(0) = 0$,
    Pass I:
1       for $t \leftarrow L$ to 1 by $-1$ do begin
2           for each node $y$ on level $t$ do begin
3               $r \leftarrow \min\{L + 1, l \mid$ an immediate predecessor $x$ of $y$ is "executed at level $l$," that is, $x =$
                   $T(l)$ or $x$ is on level $l$ and is not a $T$-value$\}$,
4               if $r \le L$, $T(r) = 0$, and some free node on level $r$ does not immediately precede $y$
                   comment the test for "free" is in Definition 1,
                     then $R(y) \leftarrow r$
                     else $R(y) \leftarrow$ FIND($r - 1$),
5               add $y$ to RLIST($R(y)$),
                end,
6           while RLIST($f$) $\ne \varnothing$ for some $f > t$ do begin
7               remove the first node $y$ from RLIST($f$), $T(f) \leftarrow y$,
8               $g \leftarrow$ FIND($f - 1$), UNION($f, g$),
9               add RLIST($f$) to the end of RLIST($g$), comment now RLIST($f$) = $\varnothing$,
                end,
10          $z \leftarrow$ the last node of RLIST($t$), RLIST($t$) $\leftarrow \varnothing$,
            if $R(z) > t$ then SUB($t$) $\leftarrow z$,
11          if level $t$ is not a 1-level (i.e., the number of nodes that are not $T$-values is even) then begin
                $T(t) \leftarrow -1$, UNION($t, t - 1$) end,
        end Pass I,
```

(a)

y	16	(15)	14	13	12	11	(10)	9*	8	7	6	(5)	4	(3)	2	1
$R(y)$	7	7	6	6	5	4	5	4	3	4	4	3	2	2	1	1

f	7	6	5	4	3	2	1
$T(f)$	15	−1	10	6	5	3	−1

(b)

f	7	6	5	4	3	2	1
FROM(f)	16	—	12	11	6	4	—
TO(f)	15	−1	10	9	5	3	−1

FIG. 2. (a) Pass-I values, ( ) = nonfree, * = SUB (b) Pass-II values.

```
Pass II
    let TO($g$) = $T(g)$ for $1 \le g \le L$ comment TO and $T$ can be the same array,
12      for $f \leftarrow 1$ to $L$ do begin
13.         if TO($f$) $\ge 0$ then begin
14              let FROM($f$) be a free node on level $f$ that does not immediately precede TO($f$) if
                  TO($f$) $> 0$,
15.             if FROM($f$) = TO($g$) for some $g$ then TO($g$) $\leftarrow$ SUB($f$),
    end end end $H$
```

Figure 2 gives the values calculated by the algorithm for Figure 1. In Figure 2a, substitute nodes have asterisks, that is, y^* means y = SUB(level(y)), and nonfree nodes are parenthesized, (y). The jumps of Figure 2b correspond to the schedule of Figure 1.

Now we prove that the algorithm is correct. Let the *H schedule* be the one computed by the algorithm, that is, the level schedule with jumps (FROM(f), TO(f)), $L \ge f \ge 1$. The proof is organized as follows. Lemmas 1–4 give the basic properties of Pass I. Lemma 5 shows how Pass II modifies jumps to get the H schedule. Corollaries 1–4 give properties of the H schedule that are analogous to Lemmas 1–4. These properties include the facts that H is a valid schedule (Corollary 3) and H has an HLF-like property (Corollary 4). The latter is used to prove that H is optimum (Lemmas 6–8). (Note that Lemma 10 actually shows that H is an HLF schedule.)

The proof assumes in its organization that the algorithm runs to the end of Pass II. Inspection reveals two places where the algorithm could conceivably halt prematurely: in line 10, node z might not exist if RLIST(t) is empty; in line 14, a node FROM(f) with the desired properties might not exist. We assume at the outset of the proof that in both cases if a node does not exist, the algorithm skips to the next line and continues execution. We will see that actually the nodes always exist: Corollary 2 shows that FROM(f) exists, and a remark following Corollary 3 shows that z exists.

The proof treats 0 as a dummy node on a fictitious level 0. Thus a level l with TO(l) = 0 jumps to node 0. Similarly an assertion like "level(TO(l)) $> f$" means TO(l) is a real node, above level f.

To start, note that the LSETs are maintained (by lines 0, 8, and 11) in accordance with their definition above.

The first property of Pass I says that if a level f jumps a free node of a level t, then no subsequent jump from above t goes below t.

LEMMA 1. *Let f be a 1-level where $T(f)$ is free; let l be a 1-level where $f \geq l >$ level$(T(f))$. Then level$(T(l)) \geq$ level$(T(f))$; if equality holds, then node $T(l)$ is free.*

PROOF. In Pass I, let y be a node that is in RLIST(t) in line 10. It is easy to see that after the loop of lines 6–9, no level l with $R(y) \geq l > t$ is open.

Now consider Pass I when $t =$ level$(T(f))$. The hypotheses of the lemma imply that $R(\mathrm{SUB}(t)) \geq f \geq l > t$. So the above remark shows that l is not open after t is processed. This means that level$(T(l)) \geq$ level$(T(f))$, as desired. Further, if equality holds, then $T(l)$ is a node on level t, and $R(\mathrm{SUB}(t)) \geq l$ shows that $T(l)$ is free. □

The next lemma will be used to show that FROM nodes exist.

LEMMA 2. *Let y be a node with $R(y) \geq f$ for some 1-level f. Then f contains a free node x, $x \not\rightarrow y$.*

PROOF. Let r be the value computed in line 3 for y. So $r \geq R(y) \geq f$. If $r = f$, then $R(y) = r$, and the Lemma holds by line 4. Otherwise $r > f$. Since f is a 1-level, it contains a node x that is not a T-value. x is clearly free; $x \not\rightarrow y$, since $r > f$. □

The next lemma will be used to show that H respects precedence. It says that Pass I executes any immediate predecessor of a node y at level $R(y)$ or earlier.

LEMMA 3. *If $x \rightarrow y$, then either level$(x) \geq R(y)$ or $x = T(f)$ for some level $f > R(y)$.*

PROOF. Line 3 sets r so that any immediate predecessor x of y is executed at level r or earlier, that is, level$(x) \geq r$ or $x = T(f)$ for some $f \geq r$. Further, $r \geq R(y)$. So the lemma holds unless $f = r = R(y)$ and $x = T(r)$. But this cannot be, since line 4 shows that $r = R(y)$ implies that $T(r)$ is not a predecessor of y. □

The next lemma essentially shows the HLF property for Pass I. To motivate its statement, let l be a 1-level, and let $z = T(l)$. The HLF property implies that any node y above level(z) cannot be jumped from l. Thus if level$(y) >$ level(z) and y is scheduled after l, then all free nodes of l precede y. This is Lemma 4(a). Lemma 4(b) shows the related fact, that all nonfree nodes must indeed be jumped, or, equivalently, that a free node cannot be substituted for a nonfree node.

LEMMA 4. *Let l be a 1-level. Let y be a node executed after l by Pass I, that is, $l >$ level(y) and $y \neq T(f)$ for any $f \geq l$. Let $z = T(l)$, and suppose that either*

(a) level$(y) >$ level(z), or
(b) level$(y) =$ level(z) and y is free but z is not.

Then all free nodes of l precede y.

PROOF. First note that without loss of generality, y has no predecessors executed after l. For let x be such a predecessor. It is easy to see that x satisfies the hypotheses of the lemma (in particular, alternative (a)), and the conclusion for x gives the conclusion for y.

So all predecessors of y are executed before or at level l. This implies line 3 for y sets $r \geq l$. Now it suffices to show that $l > R(y)$. For alternatives (a) and (b) both imply that level l is open when $R(y)$ is computed. To get $l > R(y)$, line 4 must set $R(y) = \mathrm{FIND}(r - 1)$. Thus $l > \mathrm{FIND}(r - 1)$, whence $r = l$. And since the else branch of line 4 is taken (and $T(r) = T(l) = 0$), all free nodes on level l precede y. This is the lemma's conclusion.

We show $l > R(y)$ by contradiction. Recall that $y \neq T(f)$ for $f \geq l$. So lines 5–9 with $R(y) \geq l$ show that l is assigned a jump. Thus level(y) = level(z) and alternative (b) holds. If y is free, then $R(\text{SUB}(t)) \geq R(y)$, by lines 5–10. Thus $R(\text{SUB}(t)) \geq l$, whence z is free. But this contradicts (b). □

Now we examine how Pass II computes TO-values.

LEMMA 5. *For any* 1*-level* g, *TO*(g) *is either* $T(g)$ *or* *SUB*(*level*($T(g)$)). *In the latter case,* $T(g)$ *is free. In both cases,* $R(TO(g)) \geq g$.

PROOF. At the start of Pass II any value TO(g) is $T(g)$. Line 15 may change TO(g) from $T(g)$ to SUB(f), where f = level($T(g)$). This is done only if $T(g)$ is free (by line 14). Further, TO(g) is not changed again, since the new value is still on level ($T(g)$).

It remains only to show that $R(\text{TO}(g)) \geq g$. Lines 5–9 show that $R(T(g)) \geq g$. And if $T(g)$ is free, Definition 1 shows that $R(\text{SUB}(\text{level}(T(g)))) \geq g$. □

COROLLARY 1. *For any* 1*-level* g, *level*($T(g)$) = *level*($TO(g)$). $T(g)$ *is free iff* $TO(g)$ *is free. If* $T(g)$ *is nonfree,* $T(g) = TO(g)$.

Now we can show that the H schedule is well defined, that is, the FROM and TO arrays specify the jumps of a level schedule. This means first that the FROM nodes, calculated in line 14, actually exist. Second, no FROM node is itself jumped.

COROLLARY 2. *For any* 1*-level* f, *node* *FROM*(f) *exists and is not jumped* (*i.e.,* $FROM(f) \neq TO(g)$ *for any* g).

PROOF. When lines 13–15 are executed for level f, TO(f) has its final value. By Lemma 5, $R(\text{TO}(f)) \geq f$. So in line 14 node FROM(f) exists, by Lemma 2. Line 15 ensures that FROM(f) is not jumped. □

The next result shows that the H schedule is a valid schedule, that is, it respects the precedence constraints.

COROLLARY 3. *If* $x \rightarrow y$, *then the* H *schedule executes node* x *before node* y.

PROOF. Since H is a level schedule, the conclusion is obvious if y is not jumped. So suppose y = TO(f) for some 1-level f.

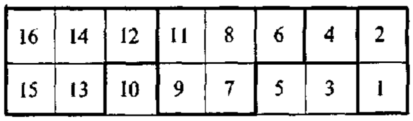
Lemma 5 shows that $R(y) \geq f$. So from Lemma 3, either level(x) $\geq f$ or $x = T(g)$ for some level $g > f$.

First suppose that level(x) $\geq f$. Then x is executed before y, unless x = FROM(f). But the latter is impossible by line 14.

So suppose that level (x) $< f$ and $x = T(g)$ for $g > f$. It suffices to show that x = TO(g), since level g jumps before f. To do this, assume the contrary. Thus $T(g) \neq$ TO(g). So $T(g)$ is free (Corollary 1). Now Lemma 1 applied to g shows level($T(f)$) $\geq$ level ($T(g)$). In other words, level(y) $\geq$ level(x). But this contradicts $x \rightarrow y$. □

Corollary 3 shows that H is a valid level schedule. One consequence of this fact is that in line 10, node z always exists, that is, RLIST(t) is not empty. For if RLIST(t) is empty, all nodes of t are jumped in Pass I. This implies that all nodes of t are jumped in the H schedule. But this is impossible, since in a level schedule, any level t has a node that is not jumped. (The last node above t to be executed precedes a node on t that is not jumped.)

Finally we show a version of the HLF property for H. This version, analogous to Lemma 4, is not the HLF property itself. Rather it is tailored to prove that the H



16	14	12	11	8	6	4	2
15	13	10	9	7	5	3	1

FIG 3 Schedule with blocks

schedule is optimum. We return to this point below in Lemma 10, which shows that this version actually implies the HLF property.

COROLLARY 4. *Let l be a* 1*-level. Let y be a node executed after level l in the H schedule. Let $z = TO(l)$ be the node jumped by l, where either*

(*a*) *$level(y) > level(z)$, or*
(*b*) *$level(y) = level(z)$ and y is free but z is not.*

Then all free nodes of l precede y.

PROOF. It suffices to show that the hypotheses of Lemma 4 hold for y, since Lemma 4 has the desired conclusion.

We first show that y is executed after l by Pass I. Since this holds for the H schedule, $l >$ level(y) and $y \neq$ TO(f) for $f \geq l$. So it suffices to show $y \neq T(f)$ for any $f \geq l$. Suppose on the contrary that $y = T(f)$. This means Pass II changes TO(f). So y is free. Lemma 1 (and Corollary 1) show that level(z) $\geq$ level(y). So alternative (b) holds. Now Lemma 1 (and Corollary 1) show that z is free. But this contradicts (b).

It remains to show that alternatives (a) or (b) of Lemma 4 hold. Each is implied by its counterpart in Corollary 4, by Corollary 1. □

The next two lemmas show that the H schedule has the same structure that guarantees optimality as the Coffman–Graham algorithm [4]: The H schedule partitions into "blocks." Any schedule executes blocks in the same order as H. Further, H is optimum on individual blocks. This implies that H is optimum.

The blocks X_i are defined by boundary levels l_i:

Definition 2. The levels l_i, $1 \leq i \leq B + 1$, are defined as follows: $l_1 = 1$. For $i > 1$, l_i is the lowest 1-level such that $l_i > l_{i-1}$ and either

(a) l_i jumps below l_{i-1}, that is, level(TO(l_i)) $< l_{i-1}$, or
(b) l_i jumps to a nonfree node on l_{i-1}, that is, level(TO(l_i)) $= l_{i-1}$ and R(SUB(l_{i-1})) $< l_i$.

Let l_B be the last value defined using the above criteria, and set $l_{B+1} = L + 1$.

For $1 \leq i \leq B$, *block* X_i consists of all nodes scheduled after level l_{i+1}, up to and including l_i, except for the node jumped from l_i. Equivalently, $X_i = \{x \mid l_{i+1} > \text{level}(x) \geq l_i$ and x is not jumped from l_{i+1} or above$\}$.

Note that any level l with an idle jump (TO(l) = 0) is a boundary level l_i. (This follows from the convention that 0 is a dummy node on level 0.) Also, any node is in exactly one block, except for a node jumped from a boundary level (which is in no block). Figure 3 shows the blocks for the schedule of Figure 1. The nodes of each block are enclosed in heavy lines.

The next two lemmas show that any schedule processes blocks in order.

LEMMA 6. *For a block X_i, $1 < i \leq B$, any node $x \in X_i$ on level l_i precedes all nodes of X_{i-1}, that is, $x \xrightarrow{*} X_{i-1}$.*

PROOF. First note that for any block X_i, $1 \le i \le B$, any node $x \in X_i$ on level l_i is free. For suppose on the contrary that x is nonfree. So x is jumped from some 1-level l. Since x is on level l_i, Definition 2 implies that $l \ge l_{i+1}$. But then $x \notin X_i$, a contradiction.

To show the lemma, take any $x \in X_i$ on level l_i and any $y \in X_{i-1}$. x is free by the above remark; similarly, if y is on l_{i-1}, it too is free. This shows that the hypotheses of Corollary 4 are satisfied for level l_i and node y. Thus $x \xrightarrow{*} y$. □

LEMMA 7. *For a block X_i, $1 < i \le B$, $X_i \xrightarrow{*} X_{i-1}$.*

PROOF. Consider any node $x \in X_i$. By Lemma 6 it suffices to show that x has a successor z on level l_i, with $z \in X_i$.

By Definition 2, level(x) $\ge l_i$. Clearly we can assume that level(x) $> l_i$. So x has a successor z on level l_i. z must be executed after x, whence after level l_{i+1}. So $z \in X_i$, as desired. □

Now we prove that H has minimum length.

LEMMA 8. *The H schedule is optimum.*

PROOF. Let $\omega(X_i)$ (respectively $\omega^*(X_i)$) denote the number of time units in the H schedule (optimum schedule) in which some node of block X_i is executed. First note that

$$\sum_{i=1}^{B} \omega^*(X_i) \ge \sum_{i=1}^{B} \omega(X_i). \tag{1}$$

This is true because in H, every time unit counted in $\omega(X_i)$, except the last, executes two nodes of X_i. (Note that any 1-level l of a block X_i, $l > l_i$, jumps a node of X_i.) The last time unit executes at least one node of X_i. (It can execute two nodes of X_i if $i = 1$.) So X_i has at least $2\omega(X_i) - 1$ nodes. This implies that $\omega^*(X_i) \ge \omega(X_i)$. Inequality (1) follows.

Now observe that the length of the optimum schedule is at least the left-hand side of (1). (Lemma 7 implies that any time unit is counted in at most one term $\omega^*(X_i)$.) The length of the H schedule is the right-hand side of (1), by Definition 2. So (1) implies that H is optimum. □

We turn our attention to the efficiency of the algorithm. We will show that the set merging operations use time $O(n\alpha(n))$, while the remainder of the algorithm is $O(e + n)$.

First we describe some additional data structures. The dag is stored in an adjacency structure: each node has a list of its immediate predecessors. Level information is stored in two ways. An array LEVEL gives the level of each node, that is, node x is on LEVEL(x). Also, each level has a list of the nodes on that level. This data structure for level information is constructed in line 0 when levels are found, in $O(n)$ additional time.

Another array T' indicates when each node is jumped in Pass I. More precisely, for each node x, $T'(x) = l$ iff $x = T(l)$; if x is not a T-value, $T'(x) = -1$. T' is initialized to -1 in line 0, and values are assigned to T' when T is assigned, in line 7. Clearly the total time spent computing T' is $O(n)$. Note that T' allows us to check if a given node x is free in time $O(1)$, since x is free iff $T'(x) \le R(\text{SUB}(\text{LEVEL}(x)))$.

With these data structures it is easy to see that lines 0–3 and 11–15 are $O(e + n)$, because $O(1)$ time is spent on each edge, node, or level. Line 0 finds the levels of the dag by using predecessor lists in a modified topological sort [13]. Line 2 loops through the nodes y on level t using the list of nodes on level t. Line 3 calculates r using the T' and LEVEL arrays. Line 11 checks if t is a 1-level using the list of nodes on level t and T'. Line 14 finds node FROM(f) by flagging the immediate predecessors of TO(f) that are on level f and finding a free, unflagged node on the list of nodes on level f. Finally, for line 15, note that FROM(f) = TO(g) iff FROM(f) = $T(g)$. So line 15 uses T' to find level g.

Now we discuss the remaining lines, 4–10. The only nontrivial part of line 4 is the test that some free node on level r does not immediately precede y. To do this, the algorithm stores, for each level r, a count of the free nodes on r. This count is computed after level r is processed (line 11); a total of $O(n)$ time for all levels is used for this computation. Line 4 computes the number of free immediate predecessors of y on level r. This number is less than the count for r iff the test has an affirmative answer. So the total time spent in the test in line 4 is $O(e + n)$.

For lines 5–10, the following data structure is used for RLISTs. For each level f, RLIST(f) is a singly linked list with pointers to the first and last elements. There is also a linked list of levels f that have RLIST(f) nonempty. With this data structure, each operation involving RLISTs in lines 5–10 is $O(1)$. (This includes finding level f in line 6.) Further, observe that line 5 is executed once for each node y; lines 7–9 are executed at most once for each level f (a value $T(f)$ is assigned only once); line 10 is executed once for each level t. So the total time in lines 5–10 is $O(n)$.

LEMMA 9. *Algorithm H uses time $O(e + n\alpha(n))$ and space $O(e + n)$.*

PROOF. The above discussion shows that aside from the set merging operations UNION and FIND, the algorithm uses time $O(e + n)$. Line 4 does at most one FIND for each node y, and line 8 does at most one FIND for each level f. So there are at most $2n$ FINDs. Lines 8 and 11 do at most one UNION for each level, so there are at most n UNIONs. Hence the total time for set merging operations is $O(n\alpha(n))$ [18]. The time bound follows.

For the space bound, note that all data structures use $O(1)$ space for each node, edge, or level. □

We summarize Lemmas 8 and 9 in our first main result.

THEOREM 1. *Algorithm H finds an optimum schedule in time $O(e + n\alpha(n))$ and space $O(e + n)$.*

We conclude the analysis by showing that the HLF property guarantees optimality. This justifies the intuitive discussion of Section 2. The main step is to prove that algorithm H finds an HLF schedule.

LEMMA 10. *The H schedule is an HLF schedule.*

PROOF. Let the H schedule have jump sequence $(t_1, \ldots, t_k)$. Let S be an arbitrary level schedule with jump sequence $(s_1, \ldots, s_r)$. We wish to show that $(t_1, \ldots, t_k) \geq (s_1, \ldots, s_r)$, where $\geq$ denotes lexicographic order. We do this by proving inductively that for all i, $1 \leq i \leq \min(k, r)$,

(i) $(t_1, \ldots, t_i) \geq (s_1, \ldots, s_i)$;
(ii) if equality holds in (i), then in each of the first i jumps of H and S, H jumps a free node iff S does.

Note that for any index i, if inequality holds in (i), then the induction is completed trivially and the desired conclusion follows. On the other hand, if (i) holds with equality for $i = \min(k, r)$, then it is easy to see that $k = r$, $(t_1, \ldots, t_k) = (s_1, \ldots, s_r)$, and again the desired conclusion follows.

So assume that (i) and (ii) hold for indices strictly less than i. We prove (i) and (ii) for i as follows. As mentioned above, we can assume that $(t_1, \ldots, t_{i-1}) = (s_1, \ldots, s_{i-1})$ if $i > 1$. This imples that the ith 1-level is the same in both schedules, call it l. Let the jump from l be *from* node x in schedule S and *to* node z in schedule H. (Thus $z = \mathrm{TO}(l)$ and $\mathrm{level}(z) = t_i$.) We will show that (i) and (ii) are both consequences of Corollary 4.

First observe two properties that hold for both S and H:

(1) All nonfree nodes of l are jumped from above l.
(2) All nonfree nodes of level(z) are jumped from above l, if z is free.

(1) is obvious for H. (2) holds for H because of Lemma 1 and Corollary 1. Furthermore, (1) and (2) for H imply their counterparts for S, because of (ii).

Next observe that node x is free. For x is not jumped in S, and so it is free by (1).

To prove (i), we must show that in S, l jumps to level(z) or below. Equivalently, if g is a level with $l > g > \mathrm{level}(z)$, S does not jump to g. To see this, suppose H executes b nodes of g before l, and a nodes of g after l. (Of course H does not execute any nodes of g at level l). S executes b nodes of g before l, since (i) holds with equality. Further, if y is a node on level g that H executes after l, then by Corollary 4(a), x precedes y. (Recall that x is a free node of l.) So S executes y after l. Thus S executes a nodes of g after l. No nodes of g remain for S to jump from l. This proves (i).

For (ii) we must show that z is free iff S jumps a free node of level(z). If z is free, (2) implies that S can only jump a free node of level(z), as desired.

On the other hand suppose z is nonfree. Let y be a free node of level(z). H executes y after l (by Lemma 1 and Corollary 1). So x precedes y by Corollary 4(b). Thus S can only jump a nonfree node of level(z), as desired. □

This completes the formal justification for the HLF definition:

THEOREM 2. *Any HLF schedule is optimum.*

PROOF. Any HLF schedule has the same jump sequence as H. Thus it has the same length as H. So Lemma 10 implies the Theorem. □

4. *Conclusions*

We have shown that for two-processor systems, HLF schedules are optimum and can be constructed efficiently. It is natural to ask how these schedules fare on various extensions of the model.

For example, consider the case of $m > 2$ processors. If the dag is a tree, Hu's algorithm [11] finds an optimum schedule for arbitrary m; further, the schedule is HLF. Unfortunately, this is not true in general: for any $m > 2$ there are dags that admit no optimum, level schedule. In fact, there are dags where *any* level schedule is a factor $2 - 2/m$ greater than optimum [14]. Among level schedules, however, the HLF strategy is best: $2 - 2/m$ is an upper bound on the accuracy, and the time to find an HLF schedule is almost linear.

Other extensions of the basic model include tasks with arbitrary integer lengths, uniform processors (i.e., processors whose speeds differ by a constant factor), and scheduling with resources other than processors. In each case the results are similar: the HLF strategy achieves the best possible accuracy bound for a level schedule, and the time is $O(e + n\alpha(n))$ or $O(e + n \log \log n)$. These results are presented in detail in [7]. These problems and others illustrate the usefulness of the highest-level-first scheduling method.

ACKNOWLEDGMENTS. The author thanks the anonymous referees for helpful advice on clarifying the presentation and Dr. Michael Garey for valuable editorial assistance.

REFERENCES

1 AHO, A V , GAREY, M R , AND ULLMAN, J D The transitive reduction of a directed graph. *SIAM J Comput 1* (1972), 131–137

2 AHO, A V , HOPCROFT, J E , AND ULLMAN, J D *The Design and Analysis of Computer Algorithms* Addison-Wesley, Reading, Mass , 1974

3 COFFMAN, E G JR , ED *Computer and Job-Shop Scheduling Theory* Wiley, New York, 1976

4 COFFMAN, E G JR , AND GRAHAM, R L Optimal scheduling for two-processor systems *Acta Inf 1*, 3 (1972), 200–213

5 FUJII, M , KASAMI, T , AND NINOMIYA, K Optimal sequencing of two equivalent processors *SIAM J Appl Math 17*, 4 (1969), 784–789 Erratum, *SIAM J Appl Math 20* (1971), 141

6 GABOW, H N An almost-linear algorithm for two-processor scheduling Tech Rep CU-CS-169-80, Dep of Computer Science, Univ of Colorado, Boulder, Colo , Jan 1980

7 GABOW, H.N Highest-level-first algorithms for approximate scheduling In preparation

8 GAREY, M R , AND JOHNSON, D S Scheduling tasks with nonuniform deadlines on two processors *J ACM 23*, 3 (July 1976), 461–467

9 GAREY, M R , AND JOHNSON, D S *Computers and Intractability A Guide to the Theory of NP-Completeness* Freeman, San Francisco, 1979

10 GRAHAM, R L , LAWLER, E L , LENSTRA, J K , AND RINNOOY KAN, A H G Optimization and approximation in deterministic sequencing and scheduling A survey *Ann Discrete Math 5* (1979), 287–326

11. HU, T C Parallel sequencing and assembly line problems. *Oper Res 9*, 6 (1961), 841–848

12 KARIV, O An $O(n^{2.5})$ algorithm for finding a maximum matching in a general graph Ph D Dissertation, Weizmann Institute of Science, Rehovot, Israel, 1976

13 KNUTH, D E *The Art of Computer Programming, Vol 1 Fundamental Algorithms* Addison-Wesley, Reading, Mass , 1973

14 LAM, S , AND SETHI, R Worst case analysis of two scheduling algorithms *SIAM J Comput 6* (1977), 518–536

15 MICALI, S , AND VAZIRANI, V V An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs Proc 21st Ann IEEE Symp on Foundations of Computer Science, Syracuse, N Y , Oct 1980, pp 17–27

16 PAN, V Y Field extension and trilinear aggregating, uniting and cancelling for the acceleration of matrix multiplications Proc 20th Ann IEEE Symp on Foundations of Computer Science, San Juan, Puerto Rico, Oct 1979, pp 28–38

17 SETHI, R Scheduling graphs on two processors *SIAM J. Comput 5*, 1 (1976), 73–82

18 TARJAN, R E Efficiency of a good but not linear set union algorithm *J ACM 22*, 2 (Apr 1975), 215–225

19 ULLMAN, J D NP-complete scheduling problems *J Comput Syst Sci 10* (1975), 384–393

20 VAN EMDE BOAS, P , KAAS, R , AND ZIJLSTRA, E Design and implementation of an efficient priority queue *Math Syst Theory 10* (1977), 99–127

RECEIVED FEBRUARY 1980, REVISED MAY 1981, ACCEPTED MAY 1981