Open Shop Scheduling to Minimize Finish Time

TEOFILO GONZALEZ AND SARTAJ SAHNI

University of Minnesota, Minneapolis, Minnesota

ABSTRACT A linear time algorithm to obtain a minimum finish time schedule for the two-processor open shop together with a polynomial time algorithm to obtain a minimum finish time preemptive schedule for open shops with more than two processors are obtained It is also shown that the problem of obtaining minimum finish time nonpreemptive schedules when the open shop has more than two processors is NP-complete.

KEY WORDS AND PHRASES. open shop, preemptive and nonpreemptive schedules, finish time, polynomial complexity, NP-complete

CR CATEGORIES 4 32, 5 39

1. *Introduction*

A *shop* consists of $m \geq 1$ processors (or machines). Each of these processors performs a different task. There are $n \geq 1$ jobs. Each job i has m tasks. The processing time for task j of job i is $t_{j,i}$. Task j of job i is to be processed on processor j, $1 \leq j \leq m$. A *schedule for a processor* j is a sequence of tuples (l_i, s_{li}, f_{li}), $1 \leq i \leq r$. The l_i are job indexes, s_{li} is the start time of job l_i, and f_{li} is its finish time. Job l_i is processed continuously on processor j from s_{l_i} to f_{l_i}. The tuples in the schedule are ordered such that $s_{l_i} < f_{l_i} \leq s_{l_{i+1}}$, $1 \leq i < r$. There may be more than one tuple per job and it is assumed that $l_i \neq l_{i+1}$, $1 \leq i < r$. It is also required that each job i spend exactly $t_{j,i}$ total time on processor j. A *schedule for an m-shop* is a set of m processor schedules, one for each processor in the shop. In addition these m processor schedules must be such that no job is to be processed simultaneously on two or more processors. A shop schedule will be abbreviated to schedule in future references. The *finish time* of a schedule is the latest completion time of the individual processor schedules and represents the time at which all tasks have been completed. An *optimal finish time* (OFT) schedule is one which has the least finish time among all schedules. A *nonpreemptive* schedule is one in which the individual processor schedule has at most one tuple (i, s_i, f_i) for each job i to be scheduled. For any processor j this allows for $t_{j,i} = 0$ and also requires that $f_i - s_i = t_{j,i}$. A schedule in which no restriction is placed on the number of tuples per job per processor is *preemptive*. Note that all nonpreemptive schedules are also preemptive, while the reverse is not true.

Open shop schedules differ from flow shop and job shop schedules [2, 3] in that in an open shop no restrictions are placed on the order in which the tasks for any job are to be processed. It is easy to conceive of situations where the tasks making up a job can be performed in any order, even though it is not possible to carry out more than one task at any particular time. For example, consider a large automotive garage with specialized

This research was supported in part by the National Science Foundation under Grant DCR 74-10081 and by University of Minnesota Graduate School Research Grant 468-0100-4909-06

Authors' present addresses: T Gonzalez, Computer Science Department, The Pennsylvania State University, University Park, PA 16802, S. Sahni, Department of Computer Science, 114 Lind Hall, University of Minnesota, Minneapolis, MN 55455

shops. A car may require the following work: replace exhaust pipes and muffler, align wheels, and tune up. These three tasks may be carried out in any order. However, since the exhaust system, alignment, and tune-up shops are in different buildings, it is not possible to perform two tasks simultaneously. In this particular example preemption may not be desirable. Open shop scheduling is also interesting from the theoretical standpoint. It is well known that determining optimal preemptive and nonpreemptive schedules for the flow shop and job shop is NP-complete. Removing the ordering constraint from these two shop problems yields the seemingly simpler open shop. As our results will show, removing the ordering constraints allows us to efficiently solve the preemptive scheduling problem but not the nonpreemptive one.

In this paper we shall investigate OFT schedules for the open shop. It is clear that when $m = 1$, OFT schedules can be trivially obtained. We shall therefore restrict ourselves to the case $m > 1$. First, in Section 2 we show that preemptive and nonpreemptive OFT schedules can be obtained in linear time when $m = 2$. This contrasts with Johnson's $O(n \log n)$ algorithm [3, p. 89] for the 2-processor flow shop. When $m > 2$, OFT preemptive schedules can still be obtained in polynomial time (Section 3).

For nonpreemptive scheduling, however, the problem of finding OFT schedules when $m > 2$ is NP-complete. These results may be compared to similar results obtained for flow shop and job shop OFT scheduling. In [4, 5] it is shown that the problem of finding nonpreemptive OFT schedules for the flow shop when $m > 2$ and for the job shop when $m > 1$ are NP-complete. In [5] it is also shown that the problem of finding preemptive OFT schedules for the 3-processor flow shop and 2-processor job shop are NP-complete. Thus, as far as the complexity of finish time scheduling is concerned, open shops are easier to schedule when a preemptive schedule is desired.

2. *OFT Scheduling for* $m = 2$

In this section a linear time algorithm to obtain nonpreemptive and preemptive OFT schedules for the case of two processors is presented. For notational simplicity we denote $t_{1,i}$, the task time on processor 1, by a_i and $t_{2,i}$ by b_i, $1 \le i \le n$. Informally the algorithm proceeds by dividing the jobs into two groups A and B. The jobs in A have $a_i \ge b_i$, while those in B have $a_i < b_i$. The schedule is built from the "middle," with jobs from A added on at the right and those from B at the left. The schedule from the jobs in A is such that there is no idle time on processor 1 (except at the end), and for each job in A it is possible to start its execution on processor 2 immediately following its completion on processor 1. The part of the schedule made up with jobs in B is such that the only idle time on processor 2 is at the beginning. In addition the processing of a job on processor 1 can be started such that its processing on processor 2 can be carried out immediately after completion on processor 1. Finally some finishing touches involving only the first and last jobs in the schedule are made. This guarantees an optimal schedule.

Line

```
1  Algorithm OPEN_SHOP
   // this algorithm finds a minimum finish time nonpreemptive schedule for the open shop problem
      with task times (a_i; b_i), 1 ≤ i ≤ n //
   // initialize variables:  a_0, b_0 represent a dummy job
                             T_i = sum of task times assigned to processor i, 1 ≤ i ≤ 2
                             l = index of leftmost job in schedule
                             r = index of rightmost job in schedule
                             S_i = sequence for processor i, 1 ≤ i ≤ 2//
2  T_1 ← T_2 ← a_0 ← b_0 ← l ← r ← 0, S ← null
   // schedule the n jobs //
3  for i ← 1 to n do
4  T_1 ← T_1 + a_i, T_2 ← T_2 + b_i
5  if a_i ≥ b_i then [if a_i ≥ b_r then
                          [// put r on right, || means string concatenation //
```

```
6                                   S ← S || r, r ← i]
                                      else
                                    [// put i on right //
7                                   S ← S || i]]
8                 else [if b_i ≥ a_i then
                                    [// put l on left //
9                                   S ← l || S; l ← i]
                                      else
                                    [// put i on left //
10                                  S ← i || S]]
11  end
    // finishing touch //
12  if T_1 − a_l < T_2 − b_r then [S_1 ← S || r || l; S_2 ← l || S || r]
13                       else [S_1 ← l || S || r; S_2 ← r || l ||S]
14  delete all occurrences of job 0 from S_1 and S_2
    // an optimal schedule is obtained by processing jobs on processor i in the order specified by S_i, 1 ≤ i ≤ 2.
       The exact schedule may be determined using Theorem 2.1 //
15  return
16  end OPEN_SHOP
```

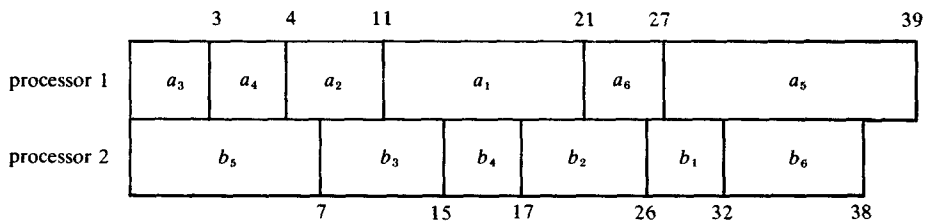
Example 2.1. Consider the open shop problem with six jobs having task times as below:

Processor	Job					
	1	2	3	4	5	6
1	10	7	3	1	12	6
2	6	9	8	2	7	6

Initially $l = r = 0$ and $S = \varnothing$. The following table gives the values of S, r, l at the end of each iteration of the for loop 3–12.

End of iteration	S	r	l
1	0	1	0
2	00	1	2
3	200	1	3
4	4200	1	3
5	42001	5	3
6	420016	5	3

We have $S = 420016$, $r = 5$, $l = 3$, $T_1 = 39$, and $T_2 = 38$. Since $T_1 - a_3 > T_2 - b_5$, we get $S_1 = 34200165$ and $S_2 = 53420016$. Deleting all occurrences of the 0's, we get $S_1 = 342165$ and $S_2 = 534216$. Processing by these permutations gives the Gantt chart:



The following two lemmas will be useful in proving the correctness of algorithm OPEN_SHOP.

LEMMA 2.1. *Let the set of jobs being scheduled be such that* $a_i \geq b_i$, $1 \leq i \leq n$, *and let* D *be the permutation obtained after deleting the* 0*'s from* S *in line* 14 *of algorithm*

OPEN_SHOP and concatenating r to the right. The jobs $1 - n$ *may be scheduled in the order such that*:

(*i*) *there is no idle time on processor* 1 *except following the completion of the last task on this processor*;

(*ii*) *for every job i, its processor* 1 *task is completed before the start of its processor* 2 *task*;

(*iii*) *for the last job r, the difference* Δ *between the completion time of task* 1 *and the start time of task* 2 *is zero.*

PROOF. The proof is by induction on n. The lemma is clearly true for $n = 1$. Assume that the lemma is true for $1 \leq n < k$. We shall show that it is also true for $n = k$. Let the k jobs be $J_1, \cdots, J_k$ and let r' be the value of r at the beginning of the iteration of the for loop of lines 3–11 when $i = n$. From the algorithm it is clear that the permutation D' obtained at line 14 when the $k - 1$ jobs $J_1, \cdots, J_{k-1}$ are to be scheduled is of the form $D''r'$. Moreover $D = D''r'k$ or $D = D''kr'$. From the induction hypothesis it follows that the jobs $J_1, \cdots, J_{k-1}$ can be scheduled according to the permutation $D''r'$ so as to satisfy (i)-(iii) of the lemma, i.e. these $k - 1$ jobs may be scheduled as in Figure 1. Let i be the job immediately preceding r' in D'. In case $k = 2$, let $i = 0$ with $a_0 = b_0 = 0$.

If $A_k \geq b_{r'}$ then $D = D'k$ and it is clear that the job k can be added on to the schedule of Figure 1, at the right end, so that (i)-(iii) of the lemma hold.

If $A_k < b_{r'}$ then $D = D''kr'$. Now job r' is moved a_k units to the right so that a_k can be accommodated between i and r', satisfying (i). Let f_1 be the finish time of a_i and $f_2 \geq f_1$ be the finish time of b_i. The finish time of a_k is then $f_1 + a_k < f_1 + a_{r'}$ as $a_{r'} \geq b_{r'}$. By (iii) the start time of $b_{r'}$ has to be $f_1 + a_k + a_{r'}$. Also we know, from the induction hypothesis, that $f_1 + a_{r'} - f_2 = \Delta' \geq 0$, i.e. $f_1 + a_{r'} \geq f_2$. The earliest that b_k may be scheduled is $\max\{f_1 + a_k, f_2\} < f_1 + a_{r'}$. This implies that there is enough time between the start time of $b_{r'}$ and the earliest start time of b_k to complete the processing of b_k □

LEMMA 2.2. *Let the set of jobs being scheduled be such that* $a_i < b_i$, $1 \leq i \leq n$, *and let C be the permutation obtained after deleting the* 0*'s from S in line* 12 *of algorithm OPEN_SHOP and concatenating l to the left. The jobs may be scheduled in the order C such that*:

(*i*) *there is no idle time on processor* 2 *except at the beginning*;

(*ii*) *for every task i, its processor* 1 *task is completed before the start of its processor* 2 *task*;

(*iii*) *for the first job l, the difference* Δ *between the completion time of task* 1 *and the start time of task* 2 *is zero.*

PROOF. The proof is similar to that of Lemma 2.1. □

LEMMA 2.3. *Let* (a_i, b_i) *be the processing times for job i on processors* 1 *and* 2 *respectively,* $1 \leq i \leq n$. *Let* f^* *be the finish time of an optimal finish time preemptive schedule. Then,* $f^* \geq \max\{\max_i\{a_i + b_i\}, T_1, T_2\}$ *where* $T_1 = \sum_1^n a_i$ *and* $T_2 = \sum_1^n b_i$.

PROOF. Obvious. □

We are now ready to prove the correctness of Algorithm OPEN_SHOP.

THEOREM 2.1. *Algorithm OPEN_SHOP generates optimal finish time schedules.*

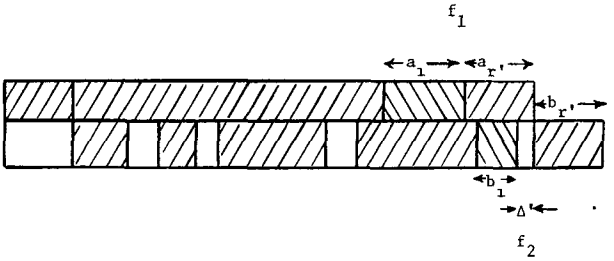


FIG 1 Scheduling by $D' = D''r'$ Shaded region indicate task processing Last job is r' $\Delta' \geq 0$

PROOF. Let $J_1, \cdots, J_n$ be the set of jobs being scheduled. Let A be the subset with $a_i \geq b_i$ and let B be the remaining jobs. It is easy to verify that the theorem is true when either A or B is empty. So assume A and B to be nonempty sets. Let S be as defined by the algorithm after completing the loop of lines 5–11. Let E be the permutation obtained after deleting the 0's from $l \parallel S \parallel r$. Then $E = CD$ where C consists solely of jobs in B, and D consists solely of jobs in A. From Lemmas 2.1 and 2.2 it follows that the jobs in A and B may be scheduled in the orders D and C to obtain schedules as in Figure 2. In the schedules of Figure 2 the processor 1 tasks for C and the processor 2 tasks for D have to be scheduled such that all the idle time appears either at the end or at the beginning. It is easy to see that this can be done. For example, the schedule of Figure 2(b) is simply obtained from Figure 1 by shifting processor 2 tasks to the right to eliminate interior idle time on P2.

Let $T_1 = \sum_1^n a_i$ and $T_2 = \sum_1^n b_i$. The schedule for the entire set of jobs is obtained by merging the two schedules of Figure 2 together so that either (a) the blocks on P1 meet first—this happens when $(\alpha_2 - \alpha_1) \leq \beta_1$; or (b) the blocks on P2 meet first—this happens when $(\alpha_2 - \alpha_1) > \beta_1$.

Let us consider these two cases separately.

Case (a) $\alpha_2 - \alpha_1 \leq \beta_1$. This happens when $T_1 - a_l \geq T_2 - b_r$. In this case line 13 of the algorithm results in the tasks on P1 being processed in the order CD while those on P2 are processed in the order rCD', where D' is D with r deleted. The section α_0–α_2 of Figure 2(a) is now shifted right until it meets with β_1–β_2 of Figure 2(b). Task b_r is moved to the leftmost point. The finish time of the schedule obtained becomes $\max\{a_r + b_r, T_1, T_2\}$, which by Lemma 2.3 is optimal.

Case (b) $\alpha_2 - \alpha_1 > \beta_1$. This happens when $T_1 - \alpha_l < T_2 - b_r$. In this case line 12 of the algorithm results in the tasks on P1 being processed in the order $C'Dl$, where C' is C with l deleted. Tasks on P2 are processed in the order CD. The schedule is obtained by processing tasks on P2 with no idle time starting at time 0. Tasks on P1 are processed with no idle time (except at the end) in the order $C'D$. Task a_l is started as early as possible following $C'D$. The finish time is seen to be $\max\{a_l + b_l, T_1, T_2\}$, which by Lemma 2.3 is optimal.

This completes the proof. □

COROLLARY 2.1. *Algorithm OPEN_SHOP generates optimal preemptive schedules for* $m = 2$.

PROOF. By Lemma 2.3 the finish time is the same for both preemptive and non-preemptive optimal schedules when $m = 2$. □

LEMMA 2.4. *The time complexity of algorithm OPEN_SHOP is* $O(n)$.

PROOF. The for loop of lines 3–11 is iterated n times. Each iteration takes a fixed amount of time. The remainder of the algorithm takes a constant amount of time Hence the complexity is $O(n)$. □

3 *Preemptive OFT Scheduling* $m > 2$

In this section we present two algorithms for optimal preemptive scheduling. The first of these is intuitively simple and so only an informal description of it is given. This algorithm makes use of basic concepts from the theory of maximal matchings in bipartite

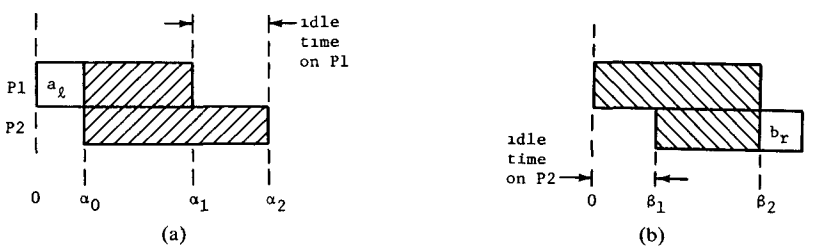


FIG 2 Partial schedules obtained for sets B and A, respectively.

graphs [6] and has a time complexity of $O(r^2)$ where r is the number of nonzero tasks. The second algorithm is a refinement over the first and has a slightly better computing time, i.e. $O(r(\min\{r, m^2\} + m \log n))$ where m is the number of processors, n the number of jobs, and r the number of nonzero tasks. It is assumed that $r \geq n$ and $r \geq m$. Hence when $r > m^2$ and $m > \log n$, the computing time of the second algorithm becomes $O(rm^2)$, which is better than $O(r^2)$. However, when $m > r/\log n$ the first algorithm has a better asymptotic time than the second (this happens, for instance, when each job has at most k nonzero tasks and $m > kn/\log n$).

Before describing these algorithms, we review some terminology and fundamental results concerning bipartite graphs. The following definitions and Proposition 3.1 are reproduced from [6].

Definition 3.1. Let $G = (X \cup Y, E)$ be a bipartite graph with vertex set $X \cup Y$ and edge set E. (If (i,j) is an edge in E then either $i \in X$ and $j \in Y$ or $i \in Y$ and $j \in X$.) A set $I \subseteq E$ is a *matching* if no vertex $v \in X \cup Y$ is incident with more than one edge in I. A matching of maximum cardinality is called a *maximum matching*. A matching is called a complete matching of X into Y if the cardinality (size) of I equals the number of vertices in X.

Definition 3.2. Let I be a matching. A vertex v is *free* relative to I if it is incident with no edge in I. A path (without repeated vertices) $p = (v_1, v_2)(v_2, v_3) \cdots (v_{2k-1}, v_{2k})$ is called an *augmenting path* if its endpoints v_1 and v_{2k} are both free and its edges are alternately in $E - I$ and in I.

PROPOSITION 3.1. *I is a maximum matching iff there is no augmenting path relative to I.*

When a matching I is *augmented* by an augmenting path P the resulting matching I' is $(I \cup P) - (I \cap P)$. The cardinality of I' is 1 + cardinality (I). Note that the matching I' still matches all vertices that were in the matching I (however, two new vertices v_1 and v_{2k} are added on).

PROPOSITION 3.2. *If $G = (\{X \cup Y\}, E)$ is a bipartite graph, $|E| = e$, $|X| = n$, and $|Y| = m$, $n \geq m$, then an augmenting path relative to I starting at some free vertex i can be found in time $O(\min\{m^2, e\})$.*

PROOF. See Appendix.

Given a set of n jobs with task times $t_{j,i}$, $1 \leq i \leq n$ and $1 \leq j \leq m$, for an m-processor open shop, we define the following quantities:

$$T_j = \sum_{1 \leq i \leq n} t_{j,i} = \text{total time needed on processor } j,\ 1 \leq j \leq m,$$

$$L_i = \sum_{1 \leq j \leq m} t_{j,i} = \text{length of job } i,\ 1 \leq i \leq n, \text{ and}$$

r = number of nonzero tasks.

From a simple extension of Lemma 2.3 to m processors, we know that every preemptive schedule must have a finish time that is at least

$$\alpha = \max_{i,j}\{T_j, L_i\}. \tag{3.1}$$

We will in fact show that the optimal preemptive schedule always has a finish time of α.

In the first algorithm starting from the given open shop problem we construct a bipartite graph with $2(n + m)$ vertices. $n + m$ of these are labeled $J_1, \cdots, J_{n+m}$ to represent the n jobs together with m fictitious jobs that we shall introduce. The remaining vertices are labeled $M_1, \cdots, M_{n+m}$ to represent the m processors together with n fictitious processors. The bipartite graph G will contain undirected weighted edges between J and M type vertices. The weight $w(J_i, M_j)$ of an edge (J_i, M_j) will represent the amount of processing time job i requires on processor j. The weight of a node $p(J_i)$ or $p(M_j)$ is the sum of the weights of the edges incident to this node. To begin with, the following edges with nonzero weight are included in G:

$$E_1 = \{(J_i, M_j) \mid t_{j,i} \neq 0,\ 1 \leq i \leq n,\ 1 \leq j \leq m\}. \tag{3.2}$$

For all edges $(J_i, M_j) \in E_1$ we define $w(J_i, M_j) = t_{j,i}$. Now a set of edges E_2 connecting $J_1, \cdots, J_n$ to $M_{m+1}, \cdots, M_{m+n}$ are added in such a way that $p(J_i) = \alpha$, $1 \le i \le n$.

$$E_2 = \{(J_i, M_{m+i}) \mid \alpha - L_i \neq 0, 1 \le i \le n\}. \tag{3.3}$$

For all edges $(J_i, M_{m+i}) \in E_2$ we define $w(J_i, M_{m+i}) = \alpha - L_i$. A set of edges E_3 is included to connect $M_1, \cdots, M_m$ to $J_{n+1}, \cdots, J_{n+m}$ in such a way that $p(M_j) = \alpha$, $1 \le j \le m$.

$$E_3 = \{(J_{n+j}, M_j) \mid \alpha - T_j \neq 0, 1 \le j \le m\}. \tag{3.4}$$

For all edges $(J_{n+j}, M_j) \in E_3$ we define $w(J_{n+j}, M_j) = \alpha - T_j$. Finally edges connecting $J_{n+1}, \cdots, J_{n+m}$ to $M_{m+1}, \cdots, M_{m+n}$ are added to make the weight of each of these vertices α. This set of edges E_4 is of size at most $n + m$ as each (J_i, M_j) edge introduced brings the weight of either J_i or M_j to α. One may easily verify that E_4 can be so constructed.

The bipartite graph $G(X, Y, E)$ is then $(\{J_1, \cdots, J_{n+m}\}, \{M_1, \cdots, M_{n+m}\}, E_1 \cup E_2 \cup E_3 \cup E_4)$. X is the set of vertices representing jobs, while Y is the set representing processors.

We illustrate this construction with an example.

Example 3.1 Let $m = 3$ and $n = 4$. The task times are defined by the matrix:

	job →	1	2	3	4	T
processor	1	10	20	0	0	30
	2	10	0	20	0	30
	3	10	0	0	20	30
	L	30	20	20	20	

Therefore $\alpha = 30$ The bipartite graph obtained using the above construction is shown in Figure 3 The edge set E_3 is empty as $T_j = p(M_j) = \alpha$, $1 \le j \le m$. □

Having constructed the bipartite graph G from the open shop problem as described earlier, we obtain a complete matching of $X = \{J_1, \cdots, J_{n+m}\}$ into $Y = \{M_1, \cdots M_{n+m}\}$. Let this matching be $e_1, e_2, \cdots, e_{n+m}$. Let $\mu = \min_{1 \le i \le n+m} \{w(e_i)\}$. The jobs incident to

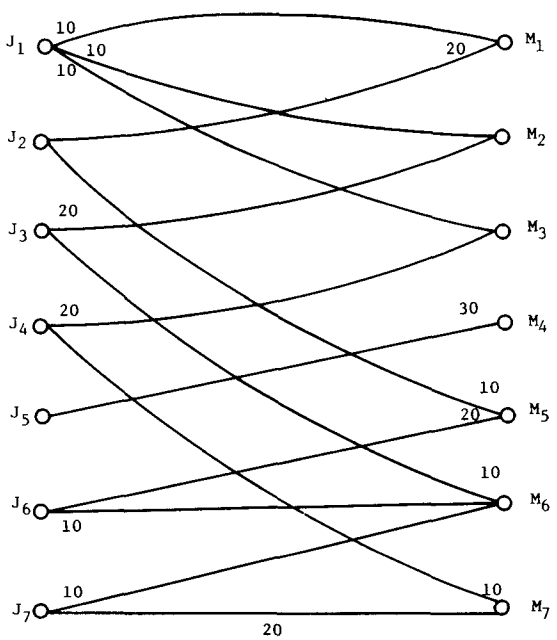


FIG. 3 Bipartite graph for Example 3 1

the edges $e_1, \cdots, e_{n+m}$ are scheduled on their respective processors for a time period of μ, and the weight of the edges $e_1, \cdots, e_{n+m}$ is decreased by μ. This results in the deletion of at least one edge (i.e. the weight of at least one edge becomes zero). By scheduling a job on its respective processor we mean that if (J_i, M_j) is one of the edges in the match, then job i is processed on processor j for μ units of time. If $j > m$, then job i is not processed in that interval. If $i > n$, then processor j is idle in that time interval. This process is repeated until all edges are deleted. Assuming that at each iteration a matching of size $n + m$ can be found, all $n + m$ processors are kept busy at all times (either processing real or fictitious jobs). The total processing time needed is $\sum_1^{n+m} p(M_i) = (n + m)\alpha$. Hence the finish time of the schedule is $(n + m)\alpha/(n + m) = \alpha$ and the schedule is optimal. Since each time a complete match is found one edge is deleted, complete matchings have to be found at most $O(r)$ times (recall that r is the number of nonzero tasks and that $r \geq n$ and $r \geq m$). Hence the maximum number of preemptions per processor is $O(r)$. The first complete matching can be found in time $O(r(n + m)^{.5})$ [6]. Subsequent matches require finding augmenting paths, each of which can be determined in time $O(r)$ (Proposition 3.2 with $e \sim O(r)$). Since a total of $O(r)$ such paths may be needed, the total computing time for the process becomes $O(r^2)$.

Example 3.2. Let us try out the informal computational process described above on the bipartite graph of Example 3.1. The following complete matchings are obtained (this is not a unique set of matchings):

(a) $\{(J_1, M_2),(J_2, M_1),(J_3, M_6),(J_4, M_3),(J_5, M_4),(J_6, M_5),(J_7, M_7)\}$, $r = 10$;
(b) $\{(J_1, M_1),(J_2, M_5),(J_3, M_2),(J_4, M_3),(J_5, M_4),(J_6, M_6),(J_7, M_7)\}$, $r = 10$;
(c) $\{(J_1, M_3),(J_2, M_1),(J_3, M_2),(J_4, M_7),(J_5, M_4),(J_6, M_5),(J_7, M_6)\}$, $r = 10$.

This yields the following schedule:

	10	10	10
M1	J2	J1	J2
M2	J1	J3	J3
M3	J4	J4	J1
M4	J5	J5	J5
M5	J6	J2	J6
M6	J3	J6	J7
M7	J7	J7	J4

Deleting the fictitious jobs and processors, the following preemptive schedule is obtained:

	10	10	10
M1	J2	J1	J2
M2	J1	J3	J3
M3	J4	J4	J1

The schedule requires only one preemption, i.e. on $M1$. Since the edge set E_3 was empty, there is no idle time on any of the processors. In general, however, this will not be the case, and the deletion of the fictitious jobs will leave some idle time on the processors. □

The success of the algorithm rests in the existence of a complete matching at each iteration. The next three lemmas prove that a complete match always exists. The vertices of the graph are divided into two disjoint sets $X = \{J_1, \cdots, J_{n+m}\}$ and $Y = \{M_1, \cdots, M_{n+m}\}$.

LEMMA 3.1. *At each iteration the weight of every vertex in the bipartite graph is equal.*

PROOF. By construction, this is certainly true for the first iteration, i.e. $p(M_i) = p(J_i) = \alpha$, $1 \leq i \leq n + m$. After a complete match is found the weight of $n + m$ edges decreases by r. The $2(n + m)$ vertices of G are each incident to exactly one edge in the matching. Hence the weight of each vertex decreases by r. Consequently all vertices have the same weight at all times. □

LEMMA 3.2 (Hall's theorem). *In a bipartite graph a complete matching of vertex set Y into vertex set X exists if and only if $|A| \leq |R(A)|$ for every subset A of Y, where $R(A)$ denotes the set of vertices in X that are adjacent to the vertices in A.*

PROOF. See Liu [9, p. 282, Th. 11.1] or Berge [1, p. 134].

LEMMA 3.3. *The conditions of Lemma* 3.2 *are valid for every bipartite graph with vertices of equal weight.*

PROOF. Let α be the weight of a vertex. Let A be any subset of Y. Then the sum of the weights of vertices in A is $\alpha \mid A \mid$. The corresponding sum for $R(A)$ is $\alpha \mid R(A) \mid$. Since this sum includes all edges incident to A, we have $\alpha \mid A \mid \leq \alpha \mid R(A) \mid$ and so $\mid A \mid \leq \mid R(A) \mid$, as $\alpha > 0$. □

The second algorithm is based upon a computational refinement of the algorithm described above. Once again a bipartite graph is constructed. This graph consists of the two vertex sets $X = \{J_1, \cdots, J_{n+m}\}$ and $Y = \{M_1, \cdots, M_m\}$. The edge set is $E_1 \cup E_3$ (cf. eqs. (3.2) and (3.4)), i.e. the fictitious processors of the earlier construction are dispensed with. Now, we look for complete matchings of Y into X. While before any complete match of Y into X was acceptable, now we have to be careful about the matching that is chosen. To see this, note that if initially the matching $\{(J_2, M_1),(J_3, M_2),(J_4, M_3)\}$ is chosen for the job set of Example 3.1, then there is no complete matching at the next iteration and consequently no schedule with finish time α can be obtained following this choice of a matching. To assist in proper choice of a complete matching we make use of an additional vector S called the slack vector. For every job i, its slack time is defined to be the difference between the amount of time remaining in the schedule and the amount of processing left for that job. If the slack time for a job becomes zero, then it is essential that the job be processed continuously up to the completion of the schedule at α as otherwise the schedule length will be greater than α. When the slack time for a job becomes zero, the job is said to have become *critical.*

Example 3.3. Consider the three-processor open shop problem with four jobs and the following task times:

Processor	Job				T
	1	2	3	4	
1	10	8	5	3	26
2	6	7	9	9	31
3	7	8	3	3	21
L	23	23	17	15	$\alpha = \max_{i,j}\{T_i, L_j\} = 31$

Addition of the jobs J_5, J_6, and J_7 introduces three more columns into the above table:

$$\begin{bmatrix} 5 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 10 \end{bmatrix}$$

Initially, the slack times are $SLACK(i) = \alpha - L_i$ and we have $SLACK = (8, 8, 4, 6, 26, 31, 21)$. No job is critical.

We first state the algorithm and then prove its correctness. For convenience, the array S in Algorithm P will represent the latest time a job may start so that its processing may be complete by α. Thus $SLACK(i) = S_i$ − current time. A job therefore becomes critical when S_i = current time. Algorithm P does not require that weights be assigned to the edges in eqs. (3.2) and (3.4). This weight assignment will, however, be used later to show that the algorithm works.

Algorithm P

// obtain an optimal preemptive schedule for the m processor open shop with n jobs and processing time $t_{j,i}$, $1 \leq i \leq n$, $1 \leq j \leq m$//
// compute length, α, of optimal schedule//
1 $T_j \leftarrow \sum_{i=1}^{n} t_{j,i}$, $1 \leq j \leq m$
2 $L_i \leftarrow \sum_{j=1}^{m} t_{j,i}$, $1 \leq i \leq n$
3 $\alpha \leftarrow \max_{i,j}\{T_j, L_i\}$
//create fictitious jobs and compute slack vector//
4 $t_{j,n+j} \leftarrow \alpha - T_j$, $1 \leq j \leq m$

```
5  S_i ← α − L_i, 1 ≤ i ≤ n
6  S_{n+j} ← T_j, 1 ≤ j ≤ m
7  n ← n + m
   //compute initial complete matching of Y = {M_1, ··· , M_m} into X = {J_1, ··· , J_{n+m}}. This match is
     obtained as a set, I, of edges (j, i) matching M_j to J_i//
8  I ← INITIAL_MATCHING, TIME ← 0  //current time//
9  loop
10   l ← index of job not in matching having least slack time
11   (p, q) ← task and job in matching with least remaining processing time
12   Δ ← min{t_{p,q}, S_l − TIME} //max time for which I can be used//
     //schedule I for Δ time units//
13   if Δ > 0 then [print (Δ, I),
                    t_{j,i} ← t_{j,i} − Δ for (j, i)∈I
                    S_i ← S_i + Δ for all jobs i∈I
                    TIME ← TIME + Δ
                    if TIME = α then stop]
14   delete from I all pairs (i, j) such that t_{j,i} = 0
     //complete matching I including all critical jobs//
15   if there is a critical job not in I then
       [delete from I all pairs (j, i) such that i is noncritical
16     repeat
17       let J_l be a critical job not in I
18       augment I using an augmenting path starting at J_l
19     until there is no critical job not in I
20     reintroduce into I all pairs (j, i) that were deleted in line 15 and such that M_j is still free]
     //complete the match//
21   while size of I ≠ m do
22     let M_j be a processor not in the matching I
23     augment I using an augmenting path starting at M_j
24   end
25 forever
26 end of Algorithm P
```

In order to prove the correctness of Algorithm P we have to show the following:

(i) There exists an initial complete matching in line 8.

(ii) The matching I can be augmented so as to include the critical job J_l in line 18.

(iii) Augmenting to a complete match including all critical jobs can always be carried out as required in lines 21–24.

The following three lemmas show that these three requirements can always be met. α is as defined in line 3 of the algorithm.

LEMMA 3.4. *There exists a complete matching of Y into X in line* 8

PROOF. Let A be any subset of vertices in Y. The weight of each vertex in A is α. The weight of any vertex in X is less than or equal to α by definition of α. Since the weight of $R(A)$ is greater than or equal to the weight of A, it follows that $\alpha \mid A \mid \leq \alpha \mid R(A) \mid$ and so $\mid A \mid \leq \mid R(A) \mid$. The result now follows from Lemma 3.2. □

LEMMA 3.5. *In line* 18 *there exists an augmenting path relative to I starting at* J_l.

PROOF. Consider the bipartite graph G' formed by the vertices X' and Y, where X' consists of all vertices representing jobs in the matching I and the vertex J_l. All edges connecting X' and Y in the original graph are included in G'. By the deletion of line 15 it follows that all vertices in X' are critical. Hence their weight is $\alpha - t$ if t is the value of *TIME* when the loop of lines 16–19 is being executed. Since $\alpha - t$ is the total remaining time on all the processors, the weight of vertices in Y in the graph G' is less than or equal to $\alpha - t$. Using the same argument as in Lemma 3.4, it follows that there is a complete match of X' into Y Hence I is not a maximum matching in G' Hence there is an augmenting path relative to I beginning at J_l. □

LEMMA 3.6 *There is always an augmenting path relative to I beginning at* M_j *in line* 23.

PROOF. At any time t the bipartite graph formed by vertices $X = \{J_1, \cdots, J_{n+m}\}$ and $Y' = \{M_i \mid M_i \text{ is in the matching } I\} \{M_j\}$ have the following properties: (a) the weight of

vertices in Y' is $\alpha - t$, and (b) the weight of vertices in X is less than or equal to $\alpha - t$ (as no vertex can have a slack time less than 0, see lines 11–13). Hence the conditions of the proof of Lemma 3.4 hold and there is a complete matching of Y' into X. By Proposition 3.1 there must be an augmenting path relative to I beginning at the free vertex M_j.

Note that the complete matching obtained at the end of the while loop 21–24 must contain all the critical jobs, as the initial matching I contained all of them and augmenting paths only add on vertices to an existing matching.

Since all processors are kept busy at all times and the total amount of processing is $m\alpha$, the finish time of the schedule generated by Algorithm P is α. This schedule is therefore optimal. □

All that remains now is to analyze the complexity of Algorithm P. In carrying out this analysis we shall need a bound on the number of jobs that can become critical. Lemma 3.7 provides this bound, and Lemma 3.8 analyzes the algorithm

LEMMA 3.7. *The number of critical jobs at any time is less than or equal to m.*

PROOF. Since all processors are kept busy at all times, it follows that at any time t the total amount of processing remaining is $m(\alpha - t)$. If at time t there are more than m critical jobs, then the processing remaining for all these critical jobs is greater than or equal to $(m + 1)(\alpha - t) > m(\alpha - t)$, which is a contradiction. Since once a job becomes critical, it stays critical until the end of the schedule, the total number of jobs that can become critical is also less than or equal to m.

LEMMA 3.8. *The asymptotic time complexity of Algorithm P is $O(r(\min\{r, m^2\} + m \log n))$, where n is the number of jobs, m the number of processors, and r the number of nonzero tasks. r is assumed to be greater than or equal to $\max\{n, m\}$.*

PROOF. Lines 1–7 take time $O(r)$ if the task times are maintained using linked lists (see [7]). Line 8 can be carried out in time $O(rm^5)$ (see [6]). If the slack times are set up as a balanced search tree or heap [7], then each execution of line 10 takes time $O(m \log n)$. At each iteration of the "loop forever" loop (lines 9–25), either a critical job is created or a task is completed (see lines 10–13). Hence by Lemma 3.7, the maximum number of iterations of this loop is $r + m = O(r)$. The total contribution of line 10 is therefore $O(rm \log n)$. The contribution from lines 11–12 and 14 is $O(rm)$. In line 13 the change in S_i requires deletion and insertion of m values from the balanced search tree. This requires a time of $O(m \log n)$. The total contribution of line 13 is therefore $O(rm \log n)$. Line 15 has the same contribution. The total computing time for Algorithm P is therefore $O(rm \log n +$ total time from lines 16–24). Over the entire algorithm the loop of lines 16–19 is iterated at most m times. By Proposition 3.2 an augmenting path can be found in time $O(\min\{r, m^2\})$. The total time for this loop is therefore $O\{\min\{r, m^2\}m + m \log n\}$. The maximum number of augmenting paths needed in the loop of lines 21–24 is $m + r$ (as one path is needed each time a critical job is found). The computing time of Algorithm P then becomes $O(\min\{r, m^2\}(m + r) + rm \log n) = O(r(\min\{r, m^2\} + m \log n))$. □

4. *Complexity of Nonpreemptive Scheduling for $m > 2$*

Having presented a very efficient algorithm to obtain an OFT schedule for $m = 2$ (preemptive and nonpreemptive) and a reasonably efficient algorithm to obtain an OFT preemptive schedule for all $m > 2$, the next question that arises is: Is there a similarly efficient algorithm for the case of nonpreemptive schedules when $m > 2$? We answer this question by showing that this problem is NP-complete [8] even when we restrict ourselves to the case when the job set consists of only one job with three nonzero task times while all other jobs have only one nonzero task time. This, then, implies that obtaining a polynomial time algorithm for $m > 2$ is as difficult as doing the same for all the other NP-complete problems. An even stronger result can be obtained when $m > 3$. Since NP-complete problems are normally stated as language recognition problems, we restate the OFT problem as such a problem.

LOFT. Given an open shop with $m > 2$ processors, a deadline τ, and a set of n jobs with processing times $t_{j,i}$, $1 \le j \le m$, $1 \le i \le n$, is there a nonpreemptive schedule with finish time less than or equal to τ?

In proving LOFT NP-complete, we shall make use of the following NP-complete problem [8].

PARTITION. A multiset $S = \{a_1, \cdots, a_n\}$ is said to have a partition iff there exists a subset, u, of the indices $1 - n$ such that $\sum_{i \in u} a_i = (\sum_{i=1}^{n} a_i)/2$. The partition problem is that of determining for an arbitrary multiset S whether it has a partition. The a_i may be assumed integer.

THEOREM 4.1. *LOFT, for any fixed $m \ge 3$, is NP-complete.*

PROOF. It is easy to show that LOFT, for any fixed $m \ge 3$, can be recognized in nondeterministic polynomial time by a Turing machine. The Turing machine just guesses the optimal permutation on each of the processors and verifies that the finish time is less than or equal to τ. The remainder of the proof is presented in Lemma 4.1. It is sufficient to prove this part for the case $m = 3$.

LEMMA 4.1. *If LOFT with $m = 3$ is polynomial sovable, then so is PARTITION.*

PROOF. From the partition problem $S = \{a_1, a_2, \cdots, a_n\}$ construct the following open shop problem, OS, with $3n + 1$ jobs, $m = 3$ machines, and all jobs with one nonzero task except for one with three tasks:

$$t_{1,i} = a_i, \quad t_{2,i} = t_{3,i} = 0, \quad \text{for } 1 \le i \le n,$$
$$t_{2,i} = a_i, \quad t_{1,i} = t_{3,i} = 0, \quad \text{for } n + 1 \le i \le 2n,$$
$$t_{3,i} = a_i, \quad t_{1,i} = t_{2,i} = 0, \quad \text{for } 2n + 1 \le i \le 3n,$$
$$t_{1,3n+1} = t_{2,3n+1} = t_{3,3n+1} = T/2,$$

where $T = \sum_1^n a_i$ and $\tau = 3T/2$.

We now show that the above open shop problem has a schedule with finish time less than or equal to $3T/2$ iff S has a partition.

(a) If S has a partition u then there is a schedule with finish time $3T/2$. One such schedule is shown in Figure 4.

(b) If S has no partition, then all schedules for OS must have a finish time greater than $3T/2$.

This is shown by contradiction. Assume that there is a schedule for OS with finish time less than or equal to $3T/2$. Since $t_{1,3n+1} = t_{2,3n+1} = t_{3,3n+1} = T/2$, it follows that in this schedule job $3n + 1$ must be being processed at all times. Since the schedule is nonpreemptive, there must be a processor j such that $t_{j,3n+1}$ begins at time $T/2$ and finishes at T. For this processor there is a set of jobs with $t_{j,i}$, $(j - 1)n + 1 \le i \le jn$ and $\sum_{(j-1)n+1}^{jn} t_{j,i} = T$. Since S has no partition, it follows that all the $T/2$ units of time preceding $t_{j,3n+1}$ on processor j cannot be used. Hence more than $T/2$ are needed after time T to complete the remaining tasks. Hence the finish time must be greater than $3T/2$. This contradicts our assumption regarding the schedule. There is therefore no schedule with finish time less than or equal to $\tau = 3T/2$ when S has no partition. □

Note that the proof of Lemma 4.1 actually shows that a very simple subcase of LOFT,

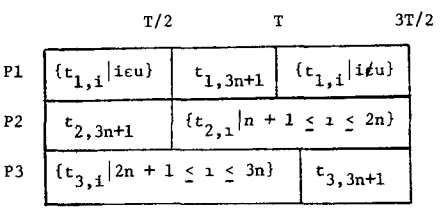


FIG 4. Optimal schedule when S has a partition

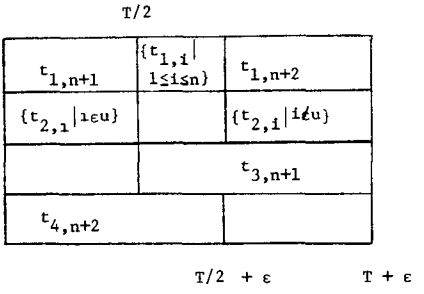


FIG 5 Optimal schedule when S has a partition

i.e. when only one job has three nonzero tasks and the remaining have at most one nonzero task, is NP-complete. When $m > 3$ the proof of Lemma 4.1 can be strengthened to the case when each job has at most two tasks.

LEMMA 4.2. *If LOFT is polynomial solvable for $m > 3$ (using only two tasks per job), then so is PARTITION*

PROOF From the partition problem $S = \{a_1, a_2, \cdots, a_n\}$ the following open shop problem, OS, with $n + 2$ jobs, $m = 4$ machines, and all jobs having at most two nonzero tasks is constructed.

$$t_{1,i} = \epsilon/n, \quad t_{2,i} = a_i, \quad t_{3,i} = t_{4,i} = 0 \quad \text{for } 1 \le i \le n,$$
$$t_{1,n+1} = T/2, \quad t_{2,n+1} = t_{4,n+1} = 0, \quad t_{3,n+1} = T/2 + \epsilon,$$
$$t_{1,n+2} = T/2, \quad t_{2,n+2} = t_{3,n+2} = 0, \quad t_{4,n+2} = T/2 + \epsilon,$$

where $T = \sum_1^n a_i$, $\tau = T + \epsilon$, and $0 < \epsilon < 1$.

We show that the above open shop problem has a schedule with finish time $< T + \epsilon$ iff S has a partition.

(a) If S has a partition u, then there is a schedule with finish time $T + \epsilon$. Figure 5 presents such a schedule.

(b) If S has no partition, then all schedules for OS must have a finish time greater than $T + \epsilon$.

This is shown by contradiction. Assume that there is a schedule for OS with finish time less than or equal to $T + \epsilon$. Since jobs $n + 1$ and $n + 2$ need a total time $T + \epsilon$ they must be scheduled all the time, and this will leave processor 1 free in the time interval $[T/2, T/2 + \epsilon]$. This is just enough time to process the n tasks $t_{1,i}$, $1 \le i \le n$. This means that all tasks t_2, that start their processing before time $T/2$ must terminate before time $T/2 + \epsilon$, as otherwise for some job j, $t_{1,j}$ and $t_{2,j}$ would be processed at the same time. Let u be the set of jobs that complete processing on processor 2 before time $T/2 + \epsilon$. Then $\sum_{i \in u} t_{2,i} \le T/2 < T/2 + \epsilon$ as the a_i are integer. This implies that tasks with total length greater than or equal to $T/2$ is left for processing after time $T/2$. If the schedule is to finish at time $T + \epsilon$ it must be the case that $\sum_{i \in u} t_{2,i} = T/2$, i.e. S has a partition. This contradicts the assumption. Hence when S has no partition there is no schedule with finish time less than or equal to $\tau = T + \epsilon$.

Lemma 4.2 leaves open the status of three-processor scheduling with two tasks per job.

Appendix

PROPOSITION 3.2. *If $G = (\{X \cup Y\}, E)$ is a bipartite graph, $|E| = e$, $|X| = n$, and $|Y| = m$, $n \ge m$, then an augmenting path relative to I starting at some free vertex i can be found in time $O(\min\{m^2, e\})$.*

PROOF. We prove this by exhibiting an augmenting path algorithm with a computing time of $O(\min\{m^2, e\})$. This algorithm assumes that the bipartite graph G is represented by its adjacency lists. (It is also assumed that the vertex set is indexed 1 through $n + m$ with $X = \{v_1, v_2, \cdots, v_n\}$ and $Y = \{v_{n+1}, \cdots, v_{n+m}\}$). Three one-dimensional arrays $FREE(1{:}n + m)$, $MARK(1{:}n + m)$, and $MATCH(1{:}n + m)$ are made use of. At entry to the augmenting path algorithm we have $FREE(i) = MARK(i) = 0$, $1 \le i \le n + m$. The initial values of $MATCH(i)$, $1 \le i \le n + m$, are not important. In addition, a FIFO queue, QUEUE, is made use of. The statement QUEUE $\Leftarrow p$ adds p to the end of the queue while $p \Leftarrow$ QUEUE deletes an element from the front of the queue and assigns it to p. Algorithm AUG works by generating an augmenting tree with the free vertex i as root and at level 1. The tree is generated level by level. Edges connecting levels q and $q + 1$ for q odd are edges not in I. The remaining edges are in I. Thus, the path from the root to any node is a valid initial segment for an augmenting path. Lines 7–22 generate the next level when the next level is even. Lines 24–28 do this for the case when the next level is odd. We use the same strategy as in [6] and look for a shortest augmenting path.

Hence if a node has already been added to the tree it is not reconsidered at a later time. Once a node is included in the tree its *MARK* bit is set to 1. Lines 14–18 reset all *MARK* and *FREE* bits changed by the algorithm. Hence *MARK* and *FREE* have to be initialized to zero only for the first use of this algorithm.

Let us analyze the computing time of this algorithm. Since $m = |Y| \leq |X|$, the number of edges in I is at most m. The time for lines 1–4 is therefore $O(m)$. Each iteration of the loop of lines 10–21 takes $O(1)$ amount of time except when a free vertex is reached. At this time $O(m)$ time is spent in lines 12–17. This happens at most once for the whole algorithm. For any vertex j the maximum number of iterations of this loop is m. This is so as at most m of the vertices adjacent to j may be in the matching I and hence not free. Let r_i be the number of nodes on the ith odd level of the augmenting tree. Then the overall contribution of lines 7–22 is at most $O(\sum mr_i + m) = O(m^2)$. $\sum r_i \leq m$ as there are at most m vertices in I and no vertex gets into the tree more than once. Since each edge in G is examined at most once, another bound is $O(e)$. Hence the time for lines 7–22 is $O(\min\{m^2, e\})$. The number of nodes on an odd level is equal to the number on the preceding even level as the connecting edges are taken from I. The total contribution from lines 24–28 is therefore $O(\sum r_i) = O(m)$. From this we conclude that an augmenting path (if it exists) may be found in time $O(\min\{m^2, e\})$ when $e \geq m$. □

The loops of lines 1–4 and 13–16 may be speeded slightly by realizing that it is sufficient to initialize *FREE*(j) to 1 only if j is in I and j will be on an even level. Similarly, *MATCH*(j) need be initialized only for those j in I that can be on odd levels of the augmenting tree.

```
Line Algorithm AUG(I,J)
 1   for each edge (j, k) in I do
 2     FREE(j) ← FREE(k) ← 1   //not free//
 3     MATCH(j) ← k; MATCH(k) ← j
 4   end
 5   QUEUE ⇐ 'i∞' //∞ is end of level marker//, MATCH(i) ← 0
 6   loop
 7     loop
 8       j ⇐ QUEUE //take off a vertex from front of queue//
 9       if j = ∞ then exit //end of level//
10       for each vertex p adjacent to j do
11         if FREE(p) = 0 then [//augmenting path found//
12                              trace path from root to p
                                this is the augmenting path
                                //reset FREE and MARK//
13                              for each edge (j, k) in I do
14                                FREE(j) ← FREE(k) ← 0
15                                MARK(j) ← MARK(k) ← 0
16                              end
17                              return]
18         if MATCH(j) ≠ p and MARK(p) = 0 then [//p not in tree//
19                                               QUEUE ⇐ p//add p to tree//
20                                               MARK(p) ← j]
21         end
22     forever
23     QUEUE ⇐ ∞ //end of level//
       //next level edges must be from I//
24     loop
25       j ⇐ QUEUE
26       if j = ∞ then exit
27       QUEUE ⇐ MATCH(j)
28     forever
29     if QUEUE empty then [stop //no augmenting path//]
30     QUEUE ⇐ ∞ //end of level//
31   forever
32 end AUG
```

ACKNOWLEDGMENT. Several organizational changes suggested by the referee have improved the readability of this paper. We wish to thank the referee for this.

REFERENCES

1 BERGE, C *Graphs and Hypergraphs* American Elsevier, New York, 1973, p 134
2 COFFMAN, E G JR *Computer and Job Shop Scheduling Theory* Wiley, New York, 1976.
3. CONWAY, R W , MAXWELL, W L , AND MILLER, L W *Theory of Scheduling* Addison-Wesley, Reading, Mass , 1968
4 GAREY, M R , JOHNSON, D , AND SETHI, R Complexity of flow shop and job shop scheduling. Tech Rep 168, Pennsylvania State U , University Park, Pa , June 1975
5 GONZALEZ, T , AND SAHNI, S Flow shop and jop shop schedules Tech Rep. TR 75-14, U of Minnesota, Minneapolis, Minn , July 1975
6. HOPCROFT, J.E , AND KARP, R.M. A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SICOMP 2* (1973), 225-231
7 HOROWITZ, E., AND SAHNI, S *Fundamentals of Data Structures* Computer Science Press, Los Angeles, Calif , 1976
8 KARP, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R E Miller and J W Thatcher, Eds , Plenum Press, New York, 1972, pp 85-104
9. LIU, C.L. *Introduction to Combinatorial Mathematics*. McGraw-Hill, New York, 1968.

RECEIVED JUNE 1975; REVISED MARCH 1976