MANAGEMENT SCIENCE
Vol. 34, No. 3, March 1988
Printed in U.S.A.

THE SHIFTING BOTTLENECK PROCEDURE FOR JOB SHOP SCHEDULING*

JOSEPH ADAMS, EGON BALAS AND DANIEL ZAWACK
Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213
American Airlines

We describe an approximation method for solving the minimum makespan problem of job shop scheduling. It sequences the machines one by one, successively, taking each time the machine identified as a bottleneck among the machines not yet sequenced. Every time after a new machine is sequenced, all previously established sequences are locally reoptimized. Both the bottleneck identification and the local reoptimization procedures are based on repeatedly solving certain one-machine scheduling problems. Besides this straight version of the Shifting Bottleneck Procedure, we have also implemented a version that applies the procedure to the nodes of a partial search tree. Computational testing shows that our approach yields consistently better results than other procedures discussed in the literature. A high point of our computational testing occurred when the enumerative version of the Shifting Bottleneck Procedure found in a little over five minutes an optimal schedule to a notorious ten machines/ten jobs problem on which many algorithms have been run for hours without finding an optimal solution.
(PRODUCTION SCHEDULING—JOB SHOP; DETERMINISTIC)

1. The Problem

The *job shop scheduling* or *machine sequencing* problem is as follows. Jobs (items) are to be processed on machines with the objective of minimizing some function of the completion times of the jobs, subject to the constraints that (i) the sequence of machines for each job is prescribed; and (ii) each machine can process only one job at a time. The processing of a job on a machine is called an operation; its time (duration) is fixed, and it cannot be interrupted. Here we choose the objective of minimizing the makespan, i.e. the time needed for processing all jobs.

Let $N = \{0, 1, \ldots, n\}$ denote the set of operations (with 0 and n the dummy operations "start" and "finish"), M the set of machines, A the set of pairs of operations constrained by precedence relations representing condition (i) above, and E_k the set of pairs of operations to be performed on machine k and which therefore cannot overlap in time, as specified in (ii). Further, let d_i denote the (fixed) duration (processing time) and t_i the (variable) start time of operation i. The problem can then be stated as

$$\min t_n$$

$$\begin{aligned} t_j - t_i &\geq d_i, && (i,j) \in A, \\ t_i &\geq 0, && i \in N, \\ t_j - t_i \geq d_i \vee t_i - t_j &\geq d_j, && (i,j) \in E_k,\ k \in M. \end{aligned} \tag{P}$$

Any feasible solution to (*P*) is called a *schedule*. For literature on this subject, see Baker (1974), Conway et al. (1967), French (1982), Lenstra (1976), Rinnooy Kan (1976). It is useful to represent this problem on a *disjunctive graph* (Roy and Sussman 1964, Balas 1969) $G = (N, A, E)$, with node set N, ordinary (conjunctive) arc set A, and disjunctive arc set E. Figure 1 illustrates this graph for a problem with 15 operations (on five jobs)

* Accepted by Marshall L. Fisher and Alexander H. G. Rinnooy Kan, acting as Special Editors; received August 6, 1986. This paper has been with the authors 2 months for 1 revision.

0025-1909/88/3403/0391$01.25

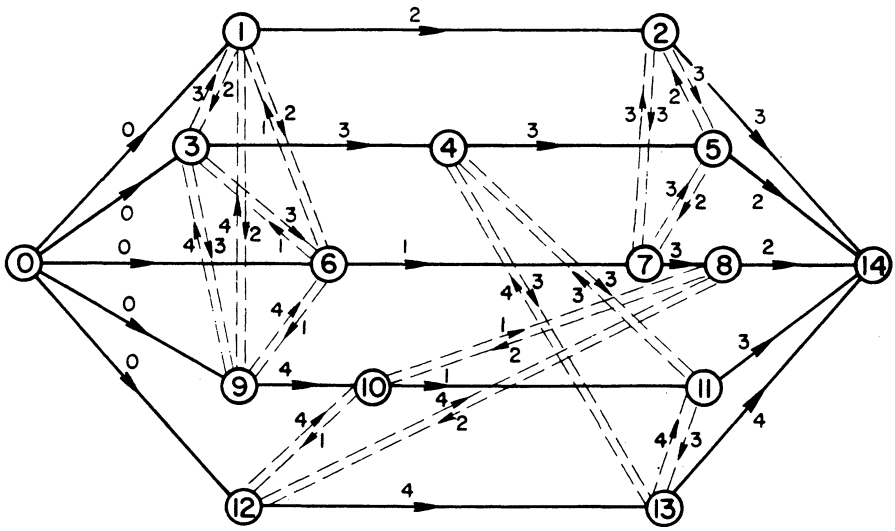

FIGURE 1

and four machines. The nodes of G correspond to operations, the directed arcs to precedence relations, and the pairs of disjunctive arcs to pairs of operations to be performed on the same machine. The numbers on the arcs are the processing times. The set of disjunctive arcs E decomposes into cliques E_k, $E = \cup(E_k : k \in M)$, one for each machine.

We will denote by $D = (N, A)$ the directed graph obtained from G by removing all the disjunctive arcs. A *selection* S_k in E_k contains exactly one member of each disjunctive arc pair of E_k. A selection is *acyclic* if it contains no directed cycle. Each acyclic selection S_k corresponds to a unique sequence of the operations pertaining to machine k, and vice-versa. Thus *sequencing a machine k* means choosing an acyclic selection in E_k. A *complete selection* S consists of the union of selections S_k, one in each E_k, $k \in M$. A *partial selection* is defined similarly, with the union taken over some subset M_0 of M. Picking a complete selection S, i.e. replacing the disjunctive arc set E by the ordinary (conjunctive) arc set S, gives rise to the (ordinary) directed graph $D_S = (N, A \cup S)$. A complete selection S is *acyclic* if the digraph D_S is acyclic (notice that if S is acyclic then each S_k, $k \in M$, is acyclic, but the converse is not true). Every acyclic complete selection S defines a family of schedules, and every schedule belongs to exactly one such family. Further, the makespan of a schedule that is optimal for S is equal to the length of a longest path in D_S. Thus in the language of disjunctive graphs, our problem is that of finding an acyclic complete selection $S \subset E$ that minimizes the length of a longest path in the directed graph D_S.

2. The Approach

Job shop scheduling is among the hardest combinatorial optimization problems. Not only is it *NP*-complete (Garey and Johnson 1979), but even among members of the latter class it belongs to the worst in practice: we can solve exactly randomly generated traveling salesman problems with 300–400 cities (over 100,000 variables) or set covering problems with hundreds of constraints and thousands of variables, but we are typically unable to schedule optimally ten jobs on ten machines. Since job shop scheduling is a very important everyday practical problem, it is therefore natural to look for approximation methods that produce an acceptable schedule in useful time.

Most of the heuristic job shop scheduling procedures described in the literature are

based on "priority dispatching" rules. These are rules for choosing an operation from a specified subset to be scheduled next. They include such criteria as SPT (shortest processing time), MWKR (most work remaining), FCFS (first come, first served), etc. The subset of eligible operations is designed to produce an "active" schedule (i.e. such that no operation can be started earlier without delaying some other operation) which is usually also required to be a "nondelay" schedule (i.e. such that no machine is idle at a time when it could begin executing some operation). These are one-pass procedures of the greedy type, in that they construct a solution through a sequence of decisions based on what seems locally best, and the decisions once made are final. Like most procedures of this type in other areas of optimization, these heuristics are fast, and they usually find solutions that are not too bad. In many situations this is all that is needed, and so their use is justified. However, with the rapid increase in the speed of computing and the growing need for efficiency in scheduling, it becomes increasingly important to explore ways of obtaining better schedules at some extra computational cost, short of going all the way towards the usually futile attempt of finding a guaranteed optimal schedule. Our paper describes an approach meant to accomplish this goal.

We sequence the machines one at a time, consecutively. In order to do this, for each machine not yet sequenced we solve to optimality a one-machine scheduling problem that is a relaxation of the original problem, and use the outcome both to rank the machines and to sequence the machine with highest rank. Every time a new machine has been sequenced, we reoptimize the sequence of each previously sequenced machine that is susceptible to improvement by again solving a one-machine problem.

Our method of solving the one-machine problems is not new; although we have speeded up considerably the time required for generating these problems. Instead, the main contribution of our approach is the way we use this relaxation to decide upon the order in which the machines should be sequenced. This is based on the classic idea of giving priority to bottleneck machines.

There is more than one way in which a machine can be viewed as a bottleneck. A first concept of this type is that of criticality. Given a partial selection $S = \cup(S_k : k \in M_0)$ and the corresponding diagraph D_S, we say that machine k is *critical* with respect to S (or the schedule associated with S) if S_k has some arc on a longest path in D_S. This definition certainly makes sense in view of the known fact (Balas 1969) that any schedule better than the one associated with S uses a selection in which at least one arc of every longest path in D_S is reversed. While appealing and theoretically justified, this notion is however not sufficiently operational for our purposes: it simply partitions the set of machines into critical and noncritical ones without offering means of distinguishing between degrees to which a machine constitutes a bottleneck. In order to prioritize the machines, we need a concept that expresses the bottleneck quality as a matter of degree rather than a yes or no property. This quality could be measured, for instance, by the marginal utility of the machine in reducing the makespan, were it not for the practical difficulty of assessing the latter. Instead, we use as a measure of the bottleneck quality of machine k the value of an optimal solution to a certain one-machine scheduling problem on machine k. To be more specific, let $M_0 \subset M$ be the set of machines that have already been sequenced by choosing selections S_p, $p \in M_0$, and for any $k \in M \backslash M_0$ let $(P(k, M_0))$ be the problem obtained from (P) by (i) replacing each disjunctive arc set E_p, $p \in M_0$, by the corresponding selection S_p, and (ii) deleting each disjunctive arc set E_p, $p \in M \backslash M_0$, $p \neq k$. This problem is equivalent to minimizing maximum lateness in a one-machine scheduling problem (for machine k) with due dates. Machine m is then called the *bottleneck* among the machines indexed by $M \backslash M_0$ if $v(m, M_0) = \max \{v(k, M_0) : k \in M \backslash M_0\}$, where $v(k, M_0)$ is the value of an optimal solution to $(P(k, M_0))$.

A brief statement of the Shifting Bottleneck Procedure is as follows. Let M_0 be the set of machines already sequenced ($M_0 = \varnothing$ at the start).

Step 1. Identify a bottleneck machine m among the machines $k \in M \backslash M_0$ and sequence it optimally. Set $M_0 \leftarrow M_0 \cup \{m\}$ and go to 2.

Step 2. Reoptimize the sequence of each critical machine $k \in M_0$ in turn, while keeping the other sequences fixed; i.e., set $M_0' := M_0 - \{k\}$ and solve $P(k, M_0')$.Then if $M_0 = M$, stop; otherwise go to 1.

The details are discussed in the next three sections.

3. An $O(n)$ Longest Path Algorithm

To identify in Step 1 the next bottleneck machine to be sequenced, for each $k \in M \backslash M_0$ we solve the problem

$$\min t_n$$

$$t_j - t_i \geq d_i, \quad (i, j) \in \cup(S_p : p \in M_0) \cup A,$$

$$t_i \geq 0, \quad i \in N,$$

$$t_j - t_i \geq d_i \vee t_i - t_j \geq d_j, \quad (i, j) \in E_k. \qquad (P(k, M_0))$$

Also, to reoptimize in Step 2 the sequence of each critical machine $k \in M_0$, for each such machine we solve a problem of the form $(P(k, M_0'))$ for some subset $M_0' \subset M_0$.

Problem $(P(k, M_0))$ is equivalent to that of finding a schedule for machine k that minimizes the maximum lateness, given that each operation i to be performed on machine k has, besides the processing time d_i, also a release time r_i and a due date f_i. Here $r_i = L(0, i)$ and $f_i = L(0, n) - L(i, n) + d_i$, with $L(i, j)$ the length of a longest path from i to j in D_T, and $T := \cup(S_p : p \in M_0)$. This latter problem in turn can be viewed as a minimum makespan problem where each job has to be processed in order by three machines, of which the first and the third have infinite capacity, while the second one (corresponding to machine k in the above model) processes one job at a time, and where the processing time of job i is r_i on the first machine, d_i on the second, and $q_i := L(0, n) - f_i$ on the third machine. The numbers r_i and q_i are sometimes referred to as the "head" and the "tail" of job i.

Thus the one-machine problems that we solve during the algorithm are of the form

$$\min t_n$$

$$t_n - t_i \geq d_i + q_i,$$

$$t_i \geq r_i, \quad i \in N^*,$$

$$t_j - t_i \geq d_i \vee t_i - t_j \geq d_j, \quad (i, j) \in E_k, \qquad (P^*(k, M_0))$$

where the r_i and q_i are defined as above, and N^* is the set of jobs to be processed on machine k.

In order to set up problem $(P^*(k, M_0))$ we have to solve two longest path problems in D_T to calculate the numbers r_i and q_i. Solving a longest path problem in an acyclic network with α arcs by standard methods takes $O(\alpha)$ time. We use instead an $O(n)$ algorithm that takes advantage of the special structure of the digraphs D_T on which our problems are defined.

Typically, the digraphs D_T are quite dense: they contain a complete subgraph for every $p \in M_0$. Thus in general $\alpha = O(n^2)$. However, it is well known that an acyclic complete directed graph is the transitive closure of its unique directed Hamilton path. Therefore, although the number of arcs in each S_p is $\frac{1}{2}v(v-1)$, where v is the number of nodes in the subgraph generated by S_p, only the $v - 1$ arcs that form the unique Hamilton path in the subgraph are of consequence for the longest path calculation; the rest can be deleted or simply ignored. In the resulting digraph, say D_T^*, every node except for the source and sink has at least one and at most two predecessors (succes-

sors). The labeling algorithm for the longest path calculations can then be modified so as to use only the relevant arcs, and its complexity thus becomes $O(n)$.

In our implementation, the graph $D_{\bar{J}}^{*}$ is not constructed explicitly. Rather than delete the redundant arcs, we keep two sets of lists: a "job list" for each job, containing the sequence of operations pertaining to that job, and a "machine list" for each machine already sequenced, containing the sequence of operations pertaining to that machine. Every node then appears on exactly one job list and at most one machine list; and its predecessors and/or successors are its neighbors on the two lists.

While the central idea of the shifting bottleneck procedure does not depend on the way one solves the longest path problems encountered, nevertheless this is the most time-consuming part of our procedure. Thus being able to solve the longest path problems in $O(n)$ time is important for the overall efficiency of the procedure.

4. Solving the One-Machine Problem

Having generated a problem $(P^*(k, M_0))$, we then solve it by the algorithm of Carlier (1982), which is closely related to the one by McMahon and Florian (1975). Although this problem is *NP*-complete in the strong sense (Garey and Johnson 1979), both of the above algorithms, which are of the branch and bound type, are known to be able to solve in a matter of seconds fairly large problems with data drawn from a realistic range: Lenstra (1976) and Rinnooy Kan (1976) report favorable results with the algorithm of McMahon and Florian (1975) on problems with up to 80 jobs; Carlier (1982) reports excellent results with his version of the algorithm on problems with up to 1,000 jobs.

For the sake of completeness, we outline here the version of Carlier's algorithm that we implemented. For details the reader is referred to Carlier (1982).

At every node of the branch and bound tree, a heuristic based on the MWKR (most work remaining) priority dispatching rule is applied to the current one-machine problem. To be specific, we start by setting $t = \min \{r_j : j \in N^*\}$, $Q = N^*$, and then repeatedly execute the following

Iterative Step. Among the unscheduled jobs ready to be scheduled at time t (i.e., those $j \in Q$ such that $r_j \leq t$), choose one, say j, with the greatest q_i (if there are ties, break them by giving preference to the greatest d_i), and schedule it by setting $t_j := t$, $Q := Q \backslash \{j\}$. Then if $Q = \varnothing$, stop; otherwise set $t := \max \{t_j + d_j, \min \{r_i : i \in Q\}\}$ and return.

Along with the schedule generated by the above heuristic, we obtain a critical path in the digraph associated with this schedule. If there are several critical paths, we choose the one with the most arcs. Let $j(1), \ldots, j(p)$ be the nodes on this critical path other than the source and sink. Let k be the largest integer in $\{1, \ldots, p\}$ such that $q_{j(k)} < q_{j(p)}$ (if there is no such k, the schedule is optimal), and let $J = \{j(k+1), \ldots, j(p)\}$. The set J plays two roles. First,

$$h(J) := \min \{r_i : i \in J\} + \sum (d_i : i \in J) + \min \{q_i : i \in J\}$$

is easily seen to be a lower bound on the minimum makespan. Second, it can be shown that in any optimal schedule job $j(k)$ comes either before or after all jobs $i \in J$.

While the lower bound $h(J)$ can be used to discard nodes of the branch and bound tree for which the upper bound is attained, the dichotomy defined by the position of $j(k)$ relative to J can serve as the basis of a branching rule. Namely, if the current node of the search tree cannot be discarded by comparing the lower and upper bounds, it can be replaced by two successor nodes, one in which job $j(k)$ has a new tail $q'_{j(k)}$, large enough to force the heuristic to schedule job $j(k)$ before all $i \in J$, and a second one in which job $j(k)$ has a new head $r'_{j(k)}$, large enough to have job $j(k)$ scheduled after all $i \in J$.

The branch and bound trees generated by the procedure are typically much smaller than n, and their size (number of nodes) rarely exceeds $2n$.

Occasionally, the selection associated with the optimal solution to $(P(k, M_0))$ may create a cycle in the resulting digraph. This can be avoided by a slight modification of the algorithm to take into account precedence relations between jobs when solving the one-machine problem. While this change is trivial and does not affect running time, its implementation would also require a systematic recording of precedence relations created by each selection. Since the occurrence of cycles is very rare (we encountered none during any of our runs), our procedure instead handles it by a special step. Namely, whenever a cycle is discovered during a longest path calculation, the particular selection and precedence relation that gave rise to it is identified and the one-machine problem that produced the culprit is solved again, this time subject to the precedence constraint.

5. The Local Reoptimization Procedure

Let M_0 be the set of machines already sequenced, and let $k(1), \ldots, k(p)$ be an arbitrary ordering of M_0 (here $p = |M_0|$). By a *local reoptimization cycle* we mean the following procedure. For $i = 1, \ldots, p$, solve the problem $(P^*(k(i), M_0\backslash\{k(i)\}))$ and substitute the optimal selection $S_{k(i)}$ for the old selection. As long as $|M_0| < |M|$, we go through at most three local reoptimization cycles for each set M_0. At the last step, when $|M_0| = |M|$, we continue the local reoptimization to the point where there is no improvement for a full cycle.

The problems $(P^*(k(i), M_0\backslash\{k(i)\}))$ encountered during local reoptimization are generated and solved by the same techniques as the problems $(P^*(k, M_0))$, discussed in §§3 and 4. The ordering $k(1), \ldots, k(p)$ of M_0 is at first given by the order in which the machines indexed by M_0 were sequenced. Every time a full cycle is completed, the elements of M_0 are reordered according to decreasing values of the solutions to the problem $(P^*(k(i), M_0\backslash\{k(i)\}))$.

It follows from the above description that the computational effort required by the local reoptimization procedure is that of setting up and solving $O(\max\{|M|^2, \gamma|M|\})$ one-machine problems, where γ is the gap between the value of the shifting bottleneck schedule and the optimum.

Finally, upon completion of the local reoptimization procedure for a given M_0, we found it useful to repeat the procedure after temporarily removing from the problem the last (according to the current ordering) α noncritical machines, i.e. machines $k(i)$ such that $S_{k(i)}$ has no arc on a critical path in D_T (where $T := \cup(S_P : p \in M_0)$), by deleting the corresponding selections $S_{k(i)}$ from the associated digraph (we take α to be the minimum of $|M_0|^{1/2}$ and the number of noncritical machines in M_0). At the end the machines that had been removed are reintroduced one by one, successively, and the procedure for M_0 is completed. This second, modified procedure typically finds additional improvements.

6. Selective Enumeration

As the computational results of the next section show, the shifting bottleneck procedure almost always obtains considerably better schedules than the best among the priority dispatch rule heuristics, and it frequently finds an optimal schedule. Nevertheless, for situations when the quality of the schedule is sufficiently important to justify a more intensive computational effort, we have developed a second version of our approach, which applies the shifting bottleneck procedure as described above to the nodes of a partial enumeration tree.

The nodes and arcs of our search tree can be described as follows. A node corresponds to a set M_0 of machines that have been sequenced in a particular way, given by the selections S_p, $p \in M_0$ (the root of the search tree corresponds to $M_0 = \varnothing$). An arc

corresponds to a pair of sets (M_0, $M_0 \cup \{k\}$) and a selection S_k for some $k \in M \backslash M_0$, generated by solving ($P^*(k, M_0)$). At a typical node of the search tree corresponding to some set M_0, we apply to the corresponding problem the shifting bottleneck procedure as described in the previous sections, with the difference that whenever we rank the machines according to decreasing $v(k, M_0)$ in order to identify the current bottleneck, we store a certain number of the one-machine problems generated for further exploration later in the process. To be specific, for a node $w(M_0)$ corresponding to a given M_0, we generate as successors of $w(M_0)$ in the search tree the nodes corresponding to the $f(l)$ highest-ranking problems ($P^*(k, M_0)$), $k \in M \backslash M_0$. Here l is the level of $w(M_0)$ in the tree, equal to $|M_0|$, and f is a decreasing function of l whose parameters are chosen to reflect considerations based on problem size, available storage space and limits on computing time.

A second instrument for limiting the size of the search tree is a penalty function, defined for every node, that penalizes the choices made at different levels in generating the node in question, in proportion to their deviation from the bottleneck, and with a weighting that is heavier for the higher than for the lower levels of the tree. Whenever the value of the penalty function for a node exceeds a predetermined limit, the node is discarded.

Whenever a node of the search tree is chosen to be processed, in keeping with the bottleneck principle it is the highest-ranking (in terms of its value $v(k, M_0)$) unexplored node among the successors of its parent node. As to our search strategy, we use a combination of breadth first with depth first. In a first phase, we generate all the nodes provided for by the successor function $f(l)$ for the levels $l = 1, \ldots, l^*$ (we actually use $l^* = \lceil |M|^{1/2} \rceil$). At the end of this phase, all the active nodes of the search tree are on level l^*. Further, they form groups of $f(l^*)$ nodes, each group containing the successors of a node on level $l^* - 1$. Next we switch to a procedure that selects the highest-ranking member of one of the groups, based on an evaluation defined for every group, and explores the associated branch straight to the bottom of the search tree, or as far as the penalty function permits. The current best solution value is always stored as an upper bound, and branches on which the upper bound is attained are abandoned. When the bottom of the tree is reached or further advance along a branch is foregone because of the penalty function or the bound, we select the highest-ranking member of another group of nodes and continue.

7. Computational Experience

A FORTRAN implementation of the Shifting Bottleneck procedure was tested on a VAX 780/11, on problems taken from the literature or generated for the purposes of this experiment. The problems range from small ones, for which an optimal solution was known, to problems involving up to 500 operations.

The problems in Table 1 have the following characteristics. All jobs have to be processed on all machines (except for Problem 1, which has a more special structure). The sequence of machines for each job is randomly generated from a uniform distribution. Problem 1 is from Lenstra (1976), problems 2, 3 and 4 are from Muth and Thompson (1963), problems 10–15 are from Lawrence (1984), while the remaining problems were generated by the authors, with processing times randomly drawn from a uniform distribution on the interval [50, 100] for problem 5, [25, 100] for problem 6, [11, 40] for problems 7, 8, 9, and [5, 99] for problems 16–19. The table gives the dimensions of the problems and the results obtained by solving them with the Shifting Bottleneck procedure in its straight version (SBI), as well as in its enumerative version (SBII).

As the results show, SBI took on the order of one to two minutes for the larger problems, although it involved hundreds of micro-runs, i.e. one-machine problems.

TABLE 1

Problem	Number of			SBI			SBII			
	Machines	Jobs	Operations	Value	CPU Sec	Micro-runs	Value	CPU Sec	Macro-Runs	LB
1	5	4	20	13*	0.50	21	—	—	—	13
2	6	6	36	55*	1.50	82	—	—	—	52
3	10	10	100	1015	10.10	249	930*#	851	270	808
4	5	20	100	1290	3.50	71	1178	80	32	1164
5	10	10	100	1306	5.70	181	1239	1503	352	1028
6	10	10	100	962	12.67	235	943	1101	343	835
7	15	20	300	730	118.87	1057	710	1269	30	650
8	15	20	300	774	125.02	1105	716	1775	35	597
9	15	20	300	751	94.32	845	735	1312	35	616
10	10	15	150	1172	21.89	343	1084	362	25	995
11	10	15	150	1040	19.24	293	944	414	44	913
12	10	20	200	1304	48.54	525	1224	744	62	1218
13	10	20	200	1325	45.54	434	1291	837	64	1235
14	10	30	300	1784*	38.26	212	—	—	—	1784
15	10	30	300	1850*	29.06	164	—	—	—	1850
16	10	40	400	2553*	11.05	61	—	—	—	2553
17	10	40	400	2228*	75.03	226	—	—	—	2228
18	10	50	500	2864*	53.42	98	—	—	—	2864
19	10	50	500	2985*	27.47	75	—	—	—	2985

Value: makespan of the best schedule obtained.
Micro-runs: number of one-machine problems solved.
Macro-runs: number of times SBI was run.
LB: lower bound given by solution value for the first level bottleneck problem.
*: value known to be optimal.
#: optimal value found after 320 seconds.

TABLE 2

Problem	Priority Dispatching Rule				SBI		SBII			Improvement	
	Straight		Randomized								
	Value	CPU Sec	Value	CPU Sec	Value	CPU Sec	Value	CPU Sec	LB	SBI %	SBII %
5 machines, 10 jobs											
1	679	4.11	679	157	666*	1.26	—	—	666	1.91	—
2	792	4.03	727	125	720	1.69	669	12.5	655	1.0	7.98
3	673	4.22	634	113	623	2.46	605	31.8	588	1.74	4.57
4	670	4.33	621	139	597	2.79	593	45.4	567	3.86	4.51
5	594	3.58	594	100	593*	0.52	—	—	593	0.2	0.2
5 machines, 15 jobs											
6	927	8.20	927	233	926*	1.28	—	—	926	0	—
7	947	8.57	920	194	890*	1.51	—	—	890	3.26	—
8	880	8.28	866	280	868	2.41	863*	4.52	863	−0.2	0.35
9	952	8.31	952	260	951*	0.85	—	—	951	0.11	—
10	959*	8.40	959*	217	959*	0.81	—	—	958	0	—
5 machines, 20 jobs											
11	1223	15.24	1223	364	1222*	2.03	—	—	—	0	—
12	1041	12.68	1040	291	1039*	0.87	—	—	—	0	—
13	1151	14.17	1151	409	1150*	1.23	—	—	—	0	—
14	1293	14.77	1293	379	1292*	0.94	—	—	—	0	—
15	1320	15.74	1314	327	1207*	3.09	—	—	—	8.14	—

TABLE 2 (*cont'd*)

Problem	Priority Dispatching Rule				SBI		SBII			Improvement	
	Straight		Randomized								
	Value	CPU Sec	Value	CPU Sec	Value	CPU Sec	Value	CPU Sec	LB	SBI %	SBII %
10 machines, 10 jobs											
16	1036	7.66	1036	240	1021	6.48	978	240**	875	1.45	5.60
17	857	6.85	857	192	796	4.58	787	192**	737	7.12	8.17
18	897	6.55	897	225	891	10.2	859	225**	770	0.67	4.24
19	926	7.45	898	240	875	7.40	860	240**	709	2.56	4.24
20	1001	7.89	942	289	924	10.2	914	289**	807	1.91	2.97
10 machines, 15 jobs											
21	1208	14.71	1198	362	1172	21.9	1084	362**	995	2.17	9.52
22	1085	13.93	1038	414	1040	19.2	944	419**	913	−0.2	9.06
23	1163	14.22	1108	417	1061	24.6	1032*	225**	1032	4.24	6.86
24	1142	14.33	1048	435	1000	25.5	976	434**	881	4.58	6.87
25	1259	14.70	1160	430	1048	27.9	1017	430**	894	1.03	3.71
10 machines, 20 jobs											
26	1373	24.62	1373	744	1304	48.5	1224	744**	1218	5.03	10.85
27	1472	25.79	1417	837	1325	45.5	1291	837**	1235	6.49	8.89
28	1475	25.5	1402	901	1256	28.5	1250	901**	1216	10.41	10.84
29	1539	25.38	1382	892	1294	48.0	1239	892**	1114	6.37	10.35
30	1604	26.7	1508	816	1403	37.8	1355*	551**	1355	6.96	10.15
10 machines, 30 jobs											
31	1935	55.42	1852	1786	1784*	38.3	—	—	—	3.67	—
32	1969	57.48	1916	1889	1850*	29.1	—	—	—	3.44	—
33	1871	54.13	1806	1313	1719*	25.6	—	—	—	4.82	—
34	1926	55.65	1844	1559	1721*	27.6	—	—	—	6.67	—
35	2097	56.61	1987	1537	1888*	21.3	—	—	—	4.98	—
15 machines, 15 jobs											
36	1517	26.20	1385	735	1351	46.9	1305	735**	1224	2.45	5.78
37	1670	26.95	1551	837	1485	61.4	1423	837**	1355	4.26	8.25
38	1405	24.43	1388	1079	1280	57.7	1255	1079**	1077	7.78	9.58
39	1436	24.40	1341	669	1321	71.8	1273	669**	1221	1.49	5.07
40	1477	24.71	1383	899	1326	76.7	1269	899**	1170	4.12	8.24

Value: makespan of the best schedule obtained.

LB: lower bound given by solution value for the first level bottleneck problem.

Improvement: Percent improvement in solution value over that found by the randomized p.d.r.-based procedure.

* Solution proved to be optimal.

** Time limit set to time required by randomized priority dispatching rule.

The degree of difficulty of solving a problem by SBI of course sharply increases with the number of machines. However, for a given number of machines, an increase in the number of jobs does not seem to make the problem more difficult; on the contrary, while the computational effort shows a moderate increase, the quality of the solutions found seems to improve. In all the problems with ten machines and 30 or more jobs,

without exception, the optimal solution was found by SBI and was proved to be optimal, because the lower bound provided by the bottleneck problem on the first level, i.e. the value max $\{v(k, \varnothing) : k \in M\}$, was matched by the makespan of the schedule found by the procedure. Naturally, in these cases the proven optimality of the solution has eliminated the need to apply SBII. This seems to be quite a remarkable property of the Shifting Bottleneck Procedure.

Among the problems of Table 1, Problem 3 is the notorious ten jobs/ten machines problem from Muth and Thompson (1963, p. 236) that has defied solution for more than 20 years in spite of the fact that every available algorithm was tried on it. Over the years, better and better solutions were discovered, usually in computer runs that took several hours and generated tens of thousands of search tree nodes. A couple of years ago, a solution with a value of 930 was found (a new record at the time) at the end of just such a long run. More recently J. Carlier and E. Pinson have announced that after another long run that generated 22,000 nodes in five hours on a Prime 2655 computer this solution was proved to be optimal (Lenstra 1986). The enumerative version of the Shifting Bottleneck procedure found this solution in just over five minutes of VAX 780/11 time (without, however, proving optimality).

In order to compare our procedure with other methods, we solved 40 test problems generated by Lawrence (1984) that were also solved by him with ten different procedures based on priority dispatching rules, both straight and randomized. In these 40 problems, each job is to be processed on every machine, the sequence of machines for each job is random, and the processing times are randomly drawn integers from the interval [5, 99] (the first two 10-machine problems with 15, 20 and 30 jobs in Table 2 are the same as Problems 10–15 of Table 1).

The ten priority dispatching rules (p.d.r.'s in the sequel) used by Lawrence are as follows: FCFS (First Come First Served), LST (Late Start Time), EFT (Early Finish Time), LFT (Late Finish Time), MINSLK (Minimum Slack), SPT (Shortest Processing Time), LPT (Longest Processing Time), MIS (Most Immediate Successors), FA (First Available), RANDOM.

These p.d.r.'s were first applied in a straightforward fashion, then each of them was randomized. The randomized rule is to select one of the available operations at random from a probability distribution which makes the odds of being selected proportional to the priority assigned to each operation by the given dispatching rule. The run is then repeated until ten consecutive runs produce no improvement, and the best result obtained is reported.

On the 40 test problems, none of the ten priority dispatching rules dominated all the others. Eight of the ten rules gave the best result on at least one problem; the remaining two, LPT and FA, were never best. Since the computing times required by any of the p.d.r.-based procedures are modest (although much less so in the randomized than in the straight case), we chose the best of the ten results for each problem and recorded as computing time the sum of the CPU times for the runs with the eight rules that were effective in at least one case (in other words, we simply ignored the time for the runs with the two rules that proved ineffective). Table 2 shows the results, alongside with those obtained for the straight and the enumerative versions of the Shifting Bottleneck Procedure (SBI and SBII, respectively). As the table shows, the straight version of the Shifting Bottleneck Procedure (SBI) finds solutions that are most of the time (in 38 out of the 40 cases) better than the solutions found by the p.d.r.-based procedures, whether in their straight or randomized version, at a computational cost that is usually comparable to that of the straight p.d.r.-based procedure, but at least an order of magnitude lower than that of the randomized version of the procedure.

Further, the enumerative version of the Shifting Bottleneck Procedure (SBII) most of the time finds substantially improved solutions over those found by the straight version.

In order to make the comparison between SBII and the p.d.r.-based procedures conclusive, SBII was run on each of the 40 problems with a time limit set to the CPU time required by the randomized p.d.r.-based procedure on each of the problems. The result, as shown in Table 2, is that the SBII solution is always, without exception, at least as good as that found by the randomized p.d.r.-based procedure in the same amount of time, and in the vast majority of the cases it is considerably better. The typical improvement is somewhere between 4 percent and 10 percent.[1]

[1] Thanks are due to Steve Lawrence for making his problem set and his computational results available to us, and to the referees for their constructive, helpful comments.

The research underlying this report was supported by Grant ECS 8503192 of the National Science Foundation and Contract N00014-85-K-0198 with the U.S. Office of Naval Research. Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

References

BAKER, K. R., *Introduction to Sequencing and Scheduling*, Wiley, New York, 1974.

BALAS, E., "Machine Sequencing via Disjunctive Graphs: An Implicit Enumeration Algorithm," *Oper. Res.*, 17 (1969), 941–957.

CARLIER, J., "The One-Machine Sequencing Problem," *European J. Oper. Res.*, 11 (1982), 42–47.

CONWAY, R. N., W. L. MAXWELL AND L. W. MILLER, *Theory of Scheduling*. Addison-Wesley, Reading, MA, 1967.

FRENCH, S., *Sequencing and Scheduling: An Introduction to the Mathematics of the Job Shop*, Wiley, New York, 1982.

GAREY, M. R. AND D. S. JOHNSON, *Computers and Intractability*, W. H. Freeman and Co., San Francisco, 1979.

LAWRENCE, S., Supplement to "Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques," GSIA, Carnegie-Mellon University, October 1984.

LENSTRA, J. K., *Sequencing by Enumerative Methods*, Mathematical Centre Tract 69, Mathematisch Centrum, Amsterdam, 1976.

———, Personal communication, May 1986.

MCMAHON, G. AND M. FLORIAN, "On Scheduling with Ready Times and Due Dates to Minimize Maximum Lateness," *Oper. Res.*, 23 (1975), 475–482.

MUTH, J. F., AND G. L. THOMPSON, *Industrial Scheduling*, Prentice-Hall, Englewood Cliffs, N.J., 1963.

RINNOOY KAN, A. H. G., *Machine Scheduling Problems: Classification, Complexity and Computations*, Nijhoff, The Hague, 1976.

ROY, B., AND B. SUSSMAN, "Les problèmes d'ordonnancement avec contraintes disjonctives," Note DS No. 9 bis, SEMA, Paris, 1964.