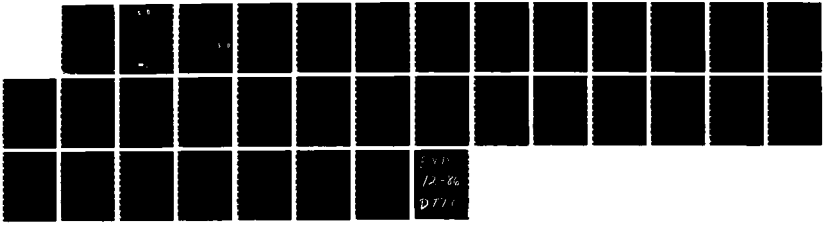
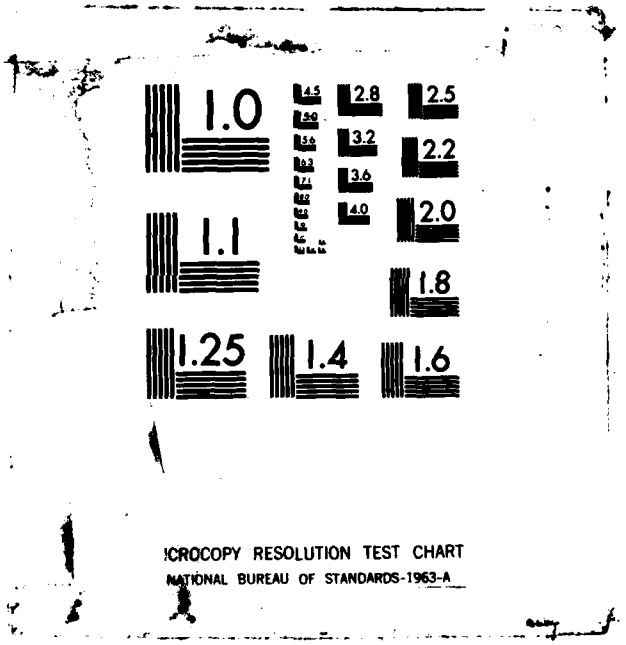
AD-A173 110 THE SHIFTING BOTTLENECK PROCEDURE FOR JOB SHOP SCHEDULING(U) CARNEGIE-MELLON UNIV PITTSBURGH PA MANAGEMENT SCIENCES RESEARCH GROUP J ADAMS ET AL 1/1

UNCLASSIFIED JUL 86 MSRR-525 N00014-85-K-0198 F/G 5/1 NL



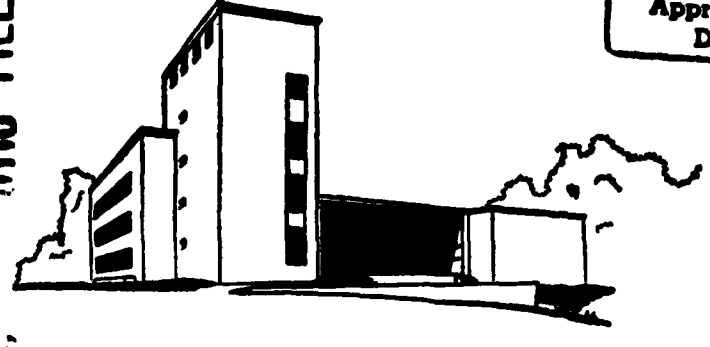


Carnegie-Mellon University

PITTSBURGH, PENNSYLVANIA 15213

GRADUATE SCHOOL OF INDUSTRIAL ADMINISTRATION

WILLIAM LARIMER MELLON, FOUNDER



W. P. #4-86-87

Management Science Research Report No. MSRR-525

THE SHIFTING BOTTLENECK PROCEDURE FOR JOB SHOP SCHEDULING

by

Joseph Adams, Egon Balas
Carnegie Mellon University

and

Daniel Zawack
American Airlines

July 1986

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.

Management Science Research Group
Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

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 truncated 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.

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. The problem can then be stated as

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

where d_i is the (fixed) duration (processing time) and t_i the (variable) start time of operation i; N is the set of operations, M the set of machines, A the set of pairs of operations constrained by precedence relations representing condition (i) above, while E_k is the set of pairs of operations to be performed on machine k and which therefore cannot overlap in time, as specified in (ii). Any feasible solution to (P) is called a *schedule*. For literature on this subject, see [1, 4, 5, 8, 11].

It is useful to represent this problem on a *disjunctive graph* [12, 2] 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 14 operations (on five jobs) and four machines. The nodes of G correspond to operations (the source 0 and the sink n are the dummy "start" and "finish" 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.

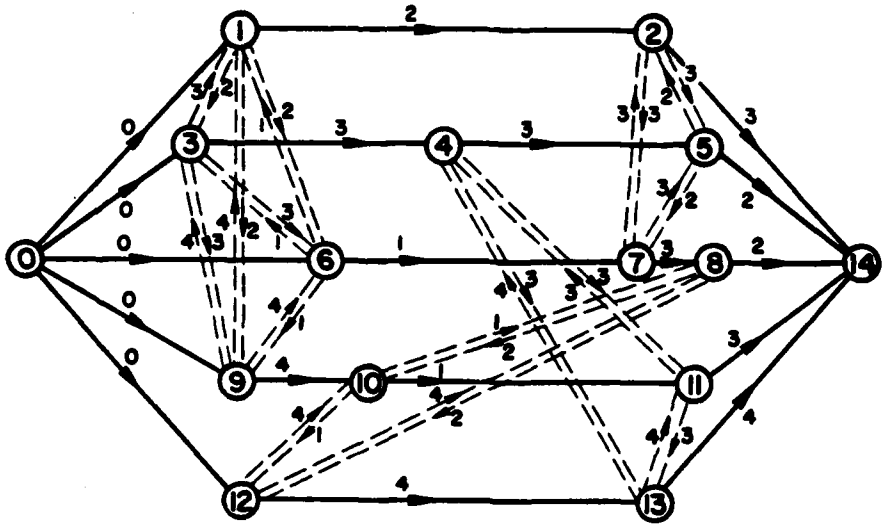


Fig. 1.

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 consists of 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, $k \in M$. 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 [6], but even among members of the latter class it belongs to the worst: 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 either an "active" schedule (i.e. such that no operation can be started earlier without delaying some other operation) or 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 by an order of magnitude 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 selection $S = \cup (S_k : k \in M)$ and the corresponding digraph 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 [2] 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 of criticality. 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 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, $p \in M \setminus M_0$, $p \neq k$. 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 $M \setminus M_0$ if $v(m,M_0) = \max\{v(k,M_0) : k \in M \setminus M_0\}$, 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 ($M_0 = \phi$ at the start).

Step 1. Identify the bottleneck machine m among the machines $k \in M \setminus M_0$ 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. 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 $k \in M \setminus M_0$ we solve the problem

$$(P(k,M_0)) \qquad \begin{aligned} \min\ & t_n \\ & t_j - t_i \geq d_i && (i,j) \in \cup\,(S_p : p \in M_0) \cup A \\ & t_i \geq 0 && i \in N \\ & t_j - t_i \geq d_i \vee t_i - t_j \geq d_j && (i,j) \in E_k \end{aligned}$$

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; where r_i is the length of a longest path from source to node i in D_T, and f_i is the difference between the length of a longest path from source to sink and that of a longest path from node i to the sink, in D_T, with $T := \cup\,(S_p : p \in M_0)$. 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, M_1, M_2 and M_3, of which M_1 and M_3 have infinite capacity while M_2 processes one job at a time, and where the processing time of job i is r_i on M_1, d_i on M_2, and $q_i := L - f_i$ on M_3, with L equal to the length of a longest source-sink path in D_T. 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

$$
P^*(k,M_0)\qquad
\begin{aligned}
&\min\ t_n\\
&t_n - t_i \geq d_i + q_i && i \in N^*\\
&t_i \geq r_i\\
&t_j - t_i \geq d_i \ \vee\ t_i - t_j \geq d_j && (i,j) \in E_k
\end{aligned}
$$

where the r_i and q_i are defined as above, and N^* is the set of jobs to be processed on machine k (corresponding to M_2 in this model).

In order to set up problem $P^*(k,M_0)$ we have to solve $2|N^*|$ longest path problems in D_T to calculate the numbers r_i and q_i. Solving a longest path problem in an acyclic network on $|N^*|$ nodes by standard methods takes $O(|N^*|^2)$ 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 $p \in M_0$. However, it is well known that an acyclic complete directed graph is the transitive closure of its unique directed Hamilton path. Therefore, of the $|S_p|(|S_p|-1) / 2$ arcs of each such subgraph, only the $|S_p|$ 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 (successors). The labeling algorithm based on Bellman's equations can then be modified as follows.

A node i can be labeled when its predecessors have been labeled. We keep all labeled nodes in a queue. To start, we label the source node and place it into the queue. Then we repeatedly apply the following

Iterative step. Pick the next node from the head of the queue, say i, remove it from the queue, and for each successor j of i in D_T^*,

— check whether j can be labeled

— if so, label j and append it to the tail of the queue.

Stop when the queue is empty.

Each node i has at most two successors in D_T^*, and a successor j of i has at most one predecessor other than i; hence the Iterative Step takes constant time, and it is applied $|N^*|$ times. Thus the algorithm takes $O(|N^*|)$ time.

In our implementation, the graph D_T^* is not constructed explicitly. Rather than delete the redundent 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 exactly 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, speeding up by an order of magnitude the time required for the longest path calculations, which is the most time-consuming part of our procedure, has had a major effect on the overall efficiency of the latter.

4. Solving the One-Machine Problem

Having generated a problem $P^*(k,M_0)$, we then solve it by the algorithm of Carlier [3], which is closely related to the one by McMahon and Florian [9]. Although this problem is NP-complete in the strong sense [6], 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 [8] and Rinnooy Kan [11] report favorable results with the algorithm of [9] on problems with up to 80 jobs; Carlier [3] 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 [3].

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 $t = \min\{r_j : j \in N^*\}$, $S = N^*$, and then repeatedly execute the following

Iterative Step. Among the unscheduled jobs ready to be scheduled at time t (i.e., those $j \in S$ 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 $t_j := t$, $S := S \setminus \{j\}$. Then if $S = \phi$, stop; otherwise set $t := \max\{t_j + d_j, \min\{r_j : j \in S\}\}$ and return.

Along with the schedule generated by the above heuristic, we obtain a critical path in the disjunctive graph associated with the problem. Let j(1), ..., j(p) be the nodes on this critical path other than the source and sink (in case of multiple critical paths, any one can serve). Let k be the largest integer in $\{1, \ldots, p\}$ such that $q_{j(k)} < q_{j(p)}$, 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 they rarely exceed 2n.

5. The Local Reoptimization Procedure

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

The problems $(P^*(k(i), M_0 \setminus \{k(i)\}))$ encountered during local reoptimization are generated and solved by the same techniques as the problems $(P^*(k, M_0))$, discussed in sections 3 and 4. The ordering k(1), ..., 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 $(P^*(k(i),M_0\backslash\{k(i)\}))$.

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. deleting the corresponding selections $S_{k(i)}$ from the associated graph (we take α to be the minimum of $|M_0|^{1/2}$ and the number of noncritical machines in M_0). At the end of the cycle, the machines that had been removed are reintroduced one by one, successively, and the cycle is completed. This second, modified procedure typically finds additional improvements.

6. Truncated 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 truncated enumeration tree.

The nodes and arcs of our search tree can be described as follows. Every node corresponds to a set M_0 of machines that have been sequenced. In particular, the node corresponding to a given M_0 represents the problem $(P(M_0))$ obtained from (P) by replacing the disjunctive arc sets E_p, $p \in M_0$, by the selections S_p, $p \in M$. Every arc corresponds to a pair of sets M_0, M_k, where $M_k = M_0 \cup \{k\}$ for some $k \in M_0\backslash M$. At a typical node of the

search tree corresponding to some set M_0, we apply to $(P(M_0))$ 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 corresponding to a given M_0, we store as successor nodes in the search tree the $f(\ell)$ highest ranking problems $(P^*(k,M_0))$, $k \in M \setminus M_0$. Here ℓ is the level of 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 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 $f(\ell)$ for the levels $\ell = 1, \ldots, \ell^*$ (we actually use $\ell^* = \lceil m^{1/2} \rceil$). At the end of this phase, all the active nodes of the search tree are on level ℓ^*. Further, they form groups of $f(\ell^*)$ nodes, each group containing the successors of a node on level $\ell^* - 1$. 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. Naturally, 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 Tables 1 and 2 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 [8], problems 2, 3 and 4 are from [10], problems 10-15 are from [7], 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 Tables give 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. The degree of difficulty of solving a problem by SBI of course

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	80.49	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

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	Number of			SBI			SBII			
	Machines	Jobs	Operations	Value	CPU Sec	Micro-runs	Value	CPU Sec	Macro-runs	LB
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, micro-runs, macro-runs, LB: see Table 1
* optimal value, proved to be optimal

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 $\max\{v(k,\phi) : k \in M\}$, was not exceeded 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 [10, 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 [8a]. 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 [7] 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]. (Problems 1, 2 of Tables 7, 8, 9 are the same as Problems 10-15 of Table 2.)

The ten priority dispatching rules (p.d.r.'s in the sequel) used by Lawrence are as follows:

1. FCFS (First Come First Served). Select the operation that becomes available at the earliest time.
2. LST (Late Start Time). Select the operation with earliest late start time (same as MWKR).
3. EFT (Early Finish Time). Select the operation that can be finished earliest.
4. LFT (Late Finish Time). Select the operation with the earliest late finish time.
5. MINSLK (Minimum Slack). Select the operation with minimum slack time.
6. SPT (Shortest Processing Time). Select the operation with the shortest processing time.
7. LPT (Longest Processing Time). Select the operation with the longest processing time.
8. MIS (Most Immediate Successors). Select the operation with the largest number of successors.
9. FA (First Available). Select the first available operation.
10. RANDOM. Select randomly among the available operations.

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 ten times, 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). Tables 3 - 10 show the results, alongside with those obtained for the straight and the enumerative versions of the Shifting Bottleneck Procedure (SBI and SBII, respectively).

As the tables show, 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

Table 3

5 machines, 10 jobs

Problem	Priority Dispatching Rule						SBI		SBII				Improvement	
	Straight			Randomized			Value	CPU Sec	Value	CPU Sec	Macro-Runs	LB	SBI %	SBII %
	Value	CPU Sec	Best Rule	Value	CPU Sec	Best Rule								
1	679	4.11	8	679	157	8	666*	1.26	——	——	——	666	1.91	——
2	792	4.03	4	727	125	6	720	1.69	669	12.5	30	655	1.0	7.98
3	673	4.22	6	634	113	3	623	2.46	605	31.8	51	588	1.74	4.57
4	670	4.33	4	621	139	6	597	2.79	593	45.4	51	567	3.86	4.51
5	594	3.58	2,4,5	594	100	1-6,8,10	593*	0.52	——	——	——	593	0.2	0.2

Value, Macro-runs, LB: see Table 1
Best rule: priority dispatching rule that gave the best value
Improvement: Percent improvement in solution value over that found by the randomized p. d. r.-based procedure
* Solution proved to be optimal

Table 4

5 machines, 15 jobs

Problem	Priority Dispatching Rule						SBI		SBII				Improvement	
	Straight			Randomized			Value	CPU Sec	Value	CPU Sec	Macro-Runs	LB	SBI %	SBII %
	Value	CPU Sec	Best Rule	Value	CPU Sec	Best Rule								
1	927	8.20	1	927	233	1,8	926*	1.28	----	---	----	926	0	----
2	947	8.57	4	920	194	4	890*	1.51	----	---	----	890	3.26	----
3	880	8.28	3	866	280	10	868	2.41	863*	4.52	2	863	- 0.2	0.35
4	952	8.31	3	952	260	3,6,8	951*	0.85	----	----	----	951	0.11	----
5	959*	8.40	8	959*	217	1,4,8	959*	0.81	----	----	----	958	0	----

For legend see Table 3

Table 5
5 machines, 20 jobs

Problem	Priority Dispatching Rule						SBI		SBII				Improvement	
	Straight			Randomized			Value	CPU Sec	Value	CPU Sec	Macro-Runs	LB	SBI %	SBII %
	Value	CPU Sec	Best Rule	Value	CPU Sec	Best Rule								
1	1223	15.24	8	1223	364	3,4,8	1222*	2.03	——	——	——	——	0	——
2	1041	12.68	1,4	1040	291	1,4	1039*	0.87	——	——	——	——	0	——
3	1151	14.17	8	1151	409	1,2,8	1150*	1.23	——	——	——	——	0	——
4	1293	14.77	1,4	1293	379	1,4,8	1292*	0.94	——	——	——	——	0	——
5	1320	15.74	3	1314	327	6	1207*	3.09	——	——	——	——	8.14	——

For legend see Table 3

Table 6
10 machines, 10 jobs

Problem	Priority Dispatching Rule						SBI		SBII				Improvement	
	Straight			Randomized			Value	CPU Sec	Value	CPU Sec**	Macro-Runs	LB	SBI %	SBII %
	Value	CPU Sec	Best Rule	Value	CPU Sec	Best Rule								
1	1036	7.66	5	1036	240	3,5	1021	6.48	978	240	93	875	1.45	5.60
2	857	6.85	4	857	192	4	796	4.58	787	192	47	737	7.12	8.17
3	897	6.55	4	897	225	4	891	10.2	859	225	64	770	0.67	4.24
4	926	7.45	3	898	240	4	875	7.40	860	240	45	709	2.56	4.24
5	1001	7.89	6	942	289	8	924	10.2	914	289	46	807	1.91	2.97

** Time limit set to time required by randomized priority dispatching rule
For rest of legend see Table 3

Table 7
10 machines, 15 jobs

Problem	Priority Dispatching Rule						SBI		SBII				Improvement	
	Straight			Randomized			Value	CPU Sec	Value	CPU Sec**	Macro-Runs	LB	SBI %	SBII %
	Value	CPU Sec	Best Rule	Value	CPU Sec	Best Rule								
1	1208	14.71	8	1198	362	5	1172	21.9	1084	362	25	995	2.17	9.52
2	1085	13.93	4	1038	414	10	1040	19.2	944	419	44	913	- 0.2	9.06
3	1163	14.22	6	1108	417	8	1061	24.6	1032*	225	30	1032	4.24	6.86
4	1142	14.33	1	1048	435	8	1000	25.5	976	434	36	881	4.58	6.87
5	1259	14.70	8	1160	430	4,8	1048	27.9	1017	430	42	894	1.03	3.71

For legend see Table 6

Table 8
10 machines, 20 jobs

Problem	Priority Dispatching Rule						SBI		SBII				Improvement	
	Straight			Randomized			Value	CPU Sec	Value	CPU Sec**	Macro-Runs	LB	SBI %	SBII %
	Value	CPU Sec	Best Rule	Value	CPU Sec	Best Rule								
1	1373	24.62	1	1373	744	1	1304	48.5	1224	744	62	1218	5.03	10.85
2	1472	25.79	3	1417	837	6	1325	45.5	1291	837	64	1235	6.49	8.89
3	1475	25.5	1	1402	901	5	1256	28.5	1250	901	68	1216	10.41	10.84
4	1539	25.38	10	1382	892	1	1294	48.0	1239	892	41	1114	6.37	10.35
5	1604	26.7	4	1508	816	6	1403	37.8	1355*	551	41	1355	6.96	10.15

For legend see Table 6

Table 9
10 machines, 30 jobs

Problem	Priority Dispatching Rule						SBI		SBII				Improvement	
	Straight			Randomized			Value	CPU Sec	Value	CPU Sec	Macro-Runs	LB	SBI %	SBII %
	Value	CPU Sec	Best Rule	Value	CPU Sec	Best Rule								
1	1935	55.42	1	1852	1786	1	1784*	38.3	——	——	——	——	3.67	——
2	1969	57.48	10	1916	1889	8	1850*	29.1	——	——	——	——	3.44	——
3	1871	54.13	4	1806	1313	3	1719*	25.6	——	——	——	——	4.82	——
4	1926	55.65	1	1844	1559	1	1721*	27.6	——	——	——	——	6.67	——
5	2097	56.61	4	1987	1537	10	1888*	21.3	——	——	——	——	4.98	——

For legend see Table 6

Table 10
15 machines, 15 jobs

Problem	Priority Dispatching Rule						SBI		SBII				Improvement	
	Straight			Randomized			Value	CPU Sec	Value	CPU Sec**	Macro-Runs	LB	SBI %	SBII %
	Value	CPU Sec	Best Rule	Value	CPU Sec	Best Rule								
1	1517	26.20	1	1385	735	2	1351	46.9	1305	735	30	1224	2.45	5.78
2	1670	26.95	6	1551	837	5	1485	61.4	1423	837	33	1355	4.26	8.25
3	1405	24.43	6	1388	1079	5	1280	57.7	1255	1079	51	1077	7.78	9.58
4	1436	24.40	4	1341	669	6	1321	71.8	1273	669	24	1221	1.49	5.07
5	1477	24.71	6	1383	899	3	1326	76.7	1269	899	39	1170	4.12	8.24

For legend see Table 6

p.d.r.-based procedure on each of the problems. The result, as shown in the Tables, 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 per cent and 10 per cent.

Acknowledgment

Thanks are due to Steve Lawrence for making his problem set and his computational results available to us.

References

[1] K. R. Baker, *Introduction to Sequencing and Scheduling.* Wiley, 1974.

[2] E. Balas, "Machine Sequencing via Disjunctive Graphs: An Implicit Enumeration Algorithm." *Operations Research, 17,* 1969, p. 941-957.

[3] J. Carlier, "The One-Machine Sequencing Problem." *European Journal of Operational Research, 11,* 1982, p. 42-47.

[4] R. N. Conway, W. L. Maxwell and L. W. Miller, *Theory of Scheduling.* Addison-Wesley, 1967.

[5] S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job Shop.* Wiley, 1982.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability.* W. H. Freeman and Co., 1979.

[7] S. Lawrence, Supplement to "Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques." GSIA, Carnegie Mellon University, October 1984.

[8] J. K. Lenstra, *Sequencing by Enumerative Methods.* Mathematical Centre Tract 69, Mathematisch Centrum, Amsterdam.

[8a] J. K. Lenstra, Personal communication, May 1986.

[9] G. McMahon and M. Florian, "On Scheduling with Ready Times and Due Dates to Minimize Maximum Lateness." *Operations Research, 23,* 1975, p. 475-482.

[10] J. F. Muth and G. L. Thompson, *Industrial Scheduling.* Prentice-Hall, 1963.

[11] A. H. G. Rinnooy Kan, *Machine Scheduling Problems: Classification, Complexity and Computations*. Nijhoff, The Hague, 1976.

[12] R. Roy and B. Sussman, "Les problemes d'ordonnancement avec constraintes disjunctives." Note DS No. 9 bis, SEMA, Paris, 1964.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MSRR 525	2. GOVT ACCESSION NO.: AD-A173110
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): The Shifting Bottleneck Procedure for Job Shop Scheduling	5. TYPE OF REPORT & PERIOD COVERED: Technical Report, July 1986
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Joseph Adams, Egon Balas, and Daniel Zawack	8. CONTRACT OR GRANT NUMBER(s): N00014 ~~00-K-0000~~ 85K 0198
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Personnel and Training Research Programs, Office of Naval Research (Code 434), Arlington VA 22217	12. REPORT DATE: July 1986 13. NUMBER OF PAGES: 27
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

scheduling
sequencing
heuristics
bottleneck

this report

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

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 pro-

(cont.)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20 (continued)

cedure to the nodes of a truncated 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. Keywords:

→ Heuristics.

END

12-86

DTIC