CARNEGIE-MELLON UNIVERSITY

PITTSBURGH, PENNSYLVANIA 15213

GRADUATE SCHOOL OF INDUSTRIAL ADMINISTRATION
WILLIAM LARIMER MELLON, FOUNDER

DISTRIBUTION STATEMENT A
Approved for public release: Distribution Unlimited

86 10 01 199
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

DISTRIBUTION STATEMENT A
Approved for public release; Distribution Unlimited
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

\[ \min t_n \]

\[ t_j - t_i \geq d_i \quad (i,j) \in A \]

\[ t_i \geq 0, \quad i \in N \]

\[ t_j - t_i \geq d_j \quad (i,j) \in E_k, \quad k \in M \]

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] \( G = (N, A, B) \), with node set \( N \), ordinary (conjunctive) arc set \( A \), and disjunctive arc set \( B \). 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.

![Graph](image)

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 $S \subseteq E$ 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 $S = \bigcup (S_k : k \in M)$ 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 ($M_0 = \emptyset$ at the start).

Step 1. Identify the bottleneck machine $m$ among the machines $k \in M \setminus M_0$ and sequence it optimally. Set $M_0 = M_0 \cup \{m\}$ 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

$$\min_{t_n} \quad \begin{cases} t_j - t_i \geq d_i & (i,j) \in \bigcup (S_p : p \leq M_0) \cup A \\ t_i \geq 0 & i \in N \\ t_j - t_i \geq d_i \lor t_i - t_j \geq d_j & (i,j) \in E_k \end{cases}$$

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 $T := \bigcup (S_p : p \leq M_0)$. 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

$$\min_{t_n} \quad \begin{cases} t_j - t_i \geq d_i & (i,j) \in \bigcup (S_p : p \leq M_0) \cup A \\ t_i \geq 0 & i \in N \\ t_j - t_i \geq d_i \lor t_i - t_j \geq d_j & (i,j) \in E_k \end{cases}$$
\begin{align*}
\min \ t_n \\
\ \ t_n - t_i \geq d_i + q_i \quad i \in N^* \\
\ \ t_i \geq r_i \\
\ \ t_j - t_i \geq d_i \vee t_i - t_j \geq d_j \quad (i,j) \in E_k
\end{align*}

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 \( p : M_0 \). 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 \( D_T^p \), 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 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 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
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
\[ t = \min\{r_j : j \in N^*\}, \quad S = N^* \]
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 \( t_j := t, \quad S := S \setminus \{j\} \). Then if
\( S = \emptyset \), 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
\( j(1), \ldots, j(p) \) 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 lar-
gest 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 $k(1), \ldots, k(p)$ 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 $|M_0| < |M| - 1$, 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 $(P^*(k(i), M_0 \backslash \{k(i)\}))$ 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 $k(1), \ldots, k(p)$ 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
\((P^*(k(i),M_0\backslash\{k(i)\}))\).

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) \(a\)
noncritical machines, i.e. deleting the corresponding selections \(S_{k(i)}\) from
the associated graph (we take \(a\) 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 \(k \in M_0\). 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 $f(t)$ highest ranking problems $(P^*(k,M_0))$, $k \in M \setminus M_0$. Here $t$ is the level of the tree, equal to $|M_0|$, and $f$ is a decreasing function of $t$ 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 $f(t)$ for the levels $t = 1, \ldots, t^*$ (we actually use $t^* = \lceil m^{1/2} \rceil$). At the end of this phase, all the active nodes of the search tree are on level $t^*$. Further, they form groups of $f(t^*)$ nodes, each group containing the successors of a node on level $t^* - 1$. 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>
<thead>
<tr>
<th>Problem</th>
<th>Number of Operations</th>
<th>Value</th>
<th>CPU Sec</th>
<th>Micro-runs</th>
<th>Value</th>
<th>CPU Sec</th>
<th>Macro-runs</th>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>4</td>
<td>20</td>
<td>13*</td>
<td>0.50</td>
<td>21</td>
<td>---</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>36</td>
<td>55*</td>
<td>1.50</td>
<td>82</td>
<td>---</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>10</td>
<td>100</td>
<td>1015</td>
<td>10.10</td>
<td>249</td>
<td>930*</td>
<td>808</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>20</td>
<td>100</td>
<td>1290</td>
<td>3.50</td>
<td>71</td>
<td>1178</td>
<td>1164</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>10</td>
<td>100</td>
<td>1306</td>
<td>5.70</td>
<td>181</td>
<td>1239</td>
<td>1503</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>10</td>
<td>100</td>
<td>962</td>
<td>12.67</td>
<td>235</td>
<td>943</td>
<td>343</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>20</td>
<td>300</td>
<td>730</td>
<td>118.87</td>
<td>1057</td>
<td>710</td>
<td>650</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>20</td>
<td>300</td>
<td>774</td>
<td>125.02</td>
<td>1105</td>
<td>716</td>
<td>35</td>
</tr>
<tr>
<td>9</td>
<td>15</td>
<td>20</td>
<td>300</td>
<td>751</td>
<td>94.32</td>
<td>845</td>
<td>735</td>
<td>616</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Number of Machines</th>
<th>Number of Jobs</th>
<th>Number of Operations</th>
<th>SBI Value</th>
<th>CPU Sec</th>
<th>Micro-runs</th>
<th>SBII Value</th>
<th>CPU Sec</th>
<th>Macro-runs</th>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>15</td>
<td>150</td>
<td>1172</td>
<td>21.89</td>
<td>343</td>
<td>1084</td>
<td>362</td>
<td>25</td>
<td>995</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>15</td>
<td>150</td>
<td>1040</td>
<td>19.24</td>
<td>293</td>
<td>944</td>
<td>414</td>
<td>44</td>
<td>913</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>20</td>
<td>200</td>
<td>1304</td>
<td>48.54</td>
<td>525</td>
<td>1224</td>
<td>744</td>
<td>62</td>
<td>1218</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>20</td>
<td>200</td>
<td>1325</td>
<td>45.54</td>
<td>434</td>
<td>1291</td>
<td>837</td>
<td>64</td>
<td>1235</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>30</td>
<td>300</td>
<td>1794*</td>
<td>38.26</td>
<td>212</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>1784</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>30</td>
<td>300</td>
<td>1850*</td>
<td>29.06</td>
<td>164</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>1850</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>40</td>
<td>400</td>
<td>2553*</td>
<td>11.05</td>
<td>61</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>2553</td>
</tr>
<tr>
<td>17</td>
<td>10</td>
<td>40</td>
<td>400</td>
<td>2228*</td>
<td>75.03</td>
<td>226</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>2228</td>
</tr>
<tr>
<td>18</td>
<td>10</td>
<td>50</td>
<td>500</td>
<td>2864*</td>
<td>53.42</td>
<td>98</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>2864</td>
</tr>
<tr>
<td>19</td>
<td>10</td>
<td>50</td>
<td>500</td>
<td>2985*</td>
<td>27.47</td>
<td>75</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>2985</td>
</tr>
</tbody>
</table>

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 $\max\{v(k,\star) : k = M\}$, 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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Priority Dispatching Rule</th>
<th>SBI</th>
<th>SBII</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Straight</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>CPU Sec</td>
<td>Best Rule</td>
</tr>
<tr>
<td>1</td>
<td>679</td>
<td>4.11</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>792</td>
<td>4.03</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>673</td>
<td>4.22</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>670</td>
<td>4.33</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>594</td>
<td>3.58</td>
<td>2,4,5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Priority Dispatching Rule</th>
<th>SBI</th>
<th>SBII</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Straight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>CPU Sec</td>
<td>Best Rule</td>
<td>Value</td>
</tr>
<tr>
<td>1</td>
<td>927</td>
<td>8.20</td>
<td>1</td>
<td>927</td>
</tr>
<tr>
<td>2</td>
<td>947</td>
<td>8.57</td>
<td>4</td>
<td>920</td>
</tr>
<tr>
<td>3</td>
<td>880</td>
<td>8.28</td>
<td>3</td>
<td>866</td>
</tr>
<tr>
<td>4</td>
<td>952</td>
<td>8.31</td>
<td>3</td>
<td>952</td>
</tr>
<tr>
<td>5</td>
<td>959*</td>
<td>8.40</td>
<td>8</td>
<td>959*</td>
</tr>
</tbody>
</table>

For legend see Table 3
<table>
<thead>
<tr>
<th>Problem</th>
<th>Priority Dispatching Rule</th>
<th>SBI</th>
<th>SBIi</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Straight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>CPU Sec</td>
<td>Best Rule</td>
<td>Value</td>
</tr>
<tr>
<td>1</td>
<td>1223</td>
<td>15.24</td>
<td>8</td>
<td>1223</td>
</tr>
<tr>
<td>2</td>
<td>1041</td>
<td>12.68</td>
<td>1,4</td>
<td>1040.</td>
</tr>
<tr>
<td>3</td>
<td>1151</td>
<td>14.17</td>
<td>8</td>
<td>1151</td>
</tr>
<tr>
<td>4</td>
<td>1293</td>
<td>14.77</td>
<td>1,4</td>
<td>1293</td>
</tr>
<tr>
<td>5</td>
<td>1320</td>
<td>15.74</td>
<td>3</td>
<td>1314</td>
</tr>
</tbody>
</table>

For legend see Table 3
<table>
<thead>
<tr>
<th>Problem</th>
<th>Priority Dispatching Rule</th>
<th>SBI</th>
<th>SBI II</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Straight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>CPU Sec</td>
<td>Best Rule</td>
<td>Value</td>
</tr>
<tr>
<td>1</td>
<td>1036</td>
<td>7.66</td>
<td>5</td>
<td>1036</td>
</tr>
<tr>
<td>2</td>
<td>857</td>
<td>6.85</td>
<td>4</td>
<td>857</td>
</tr>
<tr>
<td>3</td>
<td>897</td>
<td>6.55</td>
<td>4</td>
<td>897</td>
</tr>
<tr>
<td>4</td>
<td>926</td>
<td>7.45</td>
<td>3</td>
<td>998</td>
</tr>
<tr>
<td>5</td>
<td>1001</td>
<td>7.89</td>
<td>6</td>
<td>942</td>
</tr>
</tbody>
</table>

** Time limit set to time required by randomized priority dispatching rule
For rest of legend see Table 3
<table>
<thead>
<tr>
<th>Problem</th>
<th>Priority Dispatching Rule</th>
<th>SBI</th>
<th>SBII</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Value</td>
<td>CPU Sec</td>
<td>Best Rule</td>
<td>Value</td>
</tr>
<tr>
<td>Straight</td>
<td>Randomized</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1208</td>
<td>14.71</td>
<td>8</td>
<td>1198</td>
</tr>
<tr>
<td>2</td>
<td>1085</td>
<td>13.93</td>
<td>4</td>
<td>1038</td>
</tr>
<tr>
<td>3</td>
<td>1163</td>
<td>14.22</td>
<td>6</td>
<td>1108</td>
</tr>
<tr>
<td>4</td>
<td>1142</td>
<td>14.33</td>
<td>1</td>
<td>1048</td>
</tr>
<tr>
<td>5</td>
<td>1259</td>
<td>14.70</td>
<td>8</td>
<td>1160</td>
</tr>
</tbody>
</table>

For legend see Table 6
<table>
<thead>
<tr>
<th>Problem</th>
<th>Priority Dispatching Rule</th>
<th>SBI</th>
<th>SBII</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Straight</td>
<td>Randomized</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>CPU</td>
<td>Best</td>
<td>Value</td>
</tr>
<tr>
<td>1</td>
<td>1373</td>
<td>24.62</td>
<td>1</td>
<td>1373</td>
</tr>
<tr>
<td>2</td>
<td>1472</td>
<td>25.79</td>
<td>3</td>
<td>1417</td>
</tr>
<tr>
<td>3</td>
<td>1475</td>
<td>25.5</td>
<td>1</td>
<td>1402</td>
</tr>
<tr>
<td>4</td>
<td>1539</td>
<td>25.38</td>
<td>10</td>
<td>1382</td>
</tr>
<tr>
<td>5</td>
<td>1604</td>
<td>26.7</td>
<td>4</td>
<td>1508</td>
</tr>
</tbody>
</table>

For legend see Table 6
<table>
<thead>
<tr>
<th>Problem</th>
<th>Priority Dispatching Rule</th>
<th>SBI</th>
<th>SBII</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Straight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>CPU Sec</td>
<td>Best Rule</td>
<td>Value</td>
</tr>
<tr>
<td>1</td>
<td>1935</td>
<td>55.42</td>
<td>1</td>
<td>1852</td>
</tr>
<tr>
<td>2</td>
<td>1969</td>
<td>57.48</td>
<td>10</td>
<td>1916</td>
</tr>
<tr>
<td>3</td>
<td>1871</td>
<td>54.13</td>
<td>4</td>
<td>1806</td>
</tr>
<tr>
<td>4</td>
<td>1926</td>
<td>55.65</td>
<td>1</td>
<td>1844</td>
</tr>
<tr>
<td>5</td>
<td>2097</td>
<td>56.61</td>
<td>4</td>
<td>1987</td>
</tr>
</tbody>
</table>

For legend see Table 6
**Table 10**

15 machines, 15 jobs

<table>
<thead>
<tr>
<th>Problem</th>
<th>Priority Dispatching Rule</th>
<th>SBI</th>
<th>SBII</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Straight</td>
<td>SBI</td>
<td>SBII</td>
<td>LB</td>
</tr>
<tr>
<td>Value</td>
<td>CPU Sec</td>
<td>Best Rule</td>
<td>Value</td>
<td>CPU Sec</td>
</tr>
<tr>
<td>1</td>
<td>1517</td>
<td>26.20</td>
<td>1</td>
<td>1385</td>
</tr>
<tr>
<td>2</td>
<td>1670</td>
<td>26.95</td>
<td>6</td>
<td>1551</td>
</tr>
<tr>
<td>3</td>
<td>1405</td>
<td>24.43</td>
<td>6</td>
<td>1388</td>
</tr>
<tr>
<td>4</td>
<td>1436</td>
<td>24.40</td>
<td>4</td>
<td>1341</td>
</tr>
<tr>
<td>5</td>
<td>1477</td>
<td>24.71</td>
<td>6</td>
<td>1383</td>
</tr>
</tbody>
</table>

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


Addison-Wesley, 1967.

[5] S. French, Sequencing and Scheduling: An Introduction to the


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

Tract 69, Mathematisch Centrum, Amsterdam.


475-482.

1963.

The Shifting Bottleneck Procedure for Job Shop Scheduling

Joseph Adams, Egon Balas, and Daniel Zawack

Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh PA 15213

Personnel and Training Research Programs
Office of Naval Research (Code 434)
Arlington VA 22217

Approved for public release; Distribution Unlimited

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-
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.

Keywords:
END
12-86
DTIC