Journal of Scheduling

Manuscript Draft

Manuscript Number: JOSH109

Title: Job Shop Scheduling with Setup Times, Deadlines and Precedence Constraints

Article Type: Paper

Keywords: Job Shop Scheduling; Setup Times; Deadlines; Precedence Constraints; Shifting Bottleneck

Abstract

In the last 15 years several procedures have been developed that can find solutions of acceptable quality in reasonable computing time to Job Shop Scheduling problems in environments that do not involve sequence-dependent setup times of the machines. The presence of the latter, however, changes the picture dramatically. In this paper we adapt one of the best known heuristics, the Shifting Bottleneck Procedure, to the case when sequence dependent setup times play an important role. This is done by treating the single machine scheduling problems that arise in the process as Traveling Salesman Problems with time windows, and solving the latter by an efficient dynamic programming algorithm. The model treated here also incorporates precedence constraints, release times and deadlines. Computational experience on a vast array of instances, mainly from the semiconductor industry, shows our procedure to advance substantially the state of the art.

Job Shop Scheduling with Setup Times, Deadlines and Precedence Constraints[1]

Egon Balas[2]
Carnegie Mellon University, Pittsburgh PA

Neil Simonetti
Bryn Athyn College of the New Church, Bryn Athyn PA

Alkis Vazacopoulos[3]
Dash Optimization, Englewood Cliffs, NJ

February 2007

[1]paper presented in New York at MISTA 2005

[2]supported by the National Science Foundation through grant DMI-9802773 and by the Office of Naval Research through contract #N00014-97-1-0133

[3]corresponding author

1 Introduction and Problem Description

In several manufacturing environments, scheduling is considered to be one of their most crucial tasks. By efficiently allocating specific resources to activities over time, organizations can maximize utilization of these resources, improve productivity, and reduce overall cost. Numerous scheduling applications can be found in the automobile, chemical, computer, semiconductor, printing, and pharmaceutical industries. Because of this, research in scheduling has grown dramatically in the last fifty years.

In this paper, we deal with a variant of the Job Shop Scheduling Problem. Release dates, deadlines, and sequence-dependent setup times are introduced. This latter feature significantly changes the nature of the problem and requires a new approach. The variant of the Job Shop Scheduling problem discussed here is a very common problem in the semiconductor industry [15].

In the job shop scheduling problem, jobs (items) are to be processed on machines (resources) while minimizing some function of completion times of the jobs, subject to constraints: (i) the sequence of machines for each job is prescribed, and (ii) each machine can process at most one job at a time. The processing of a job on a machine is called an operation, and its duration is a given constant. A time may be needed to adjust a machine between two consecutive operations (jobs), which is called a setup time, and which may or may not be sequence-dependent. Jobs may also have a release time and/or a deadline. The objective here is to minimize the time needed to complete all the jobs, also known as the makespan.

Let $N = \{0, 1, \dots, n\}$ denote the set of operations, with 0 and n the dummy "start" and "finish" operations, respectively, 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 c_{ij} denote the setup time required between every pair of operations (define $c_{ij} := 0$ if $(i,j) \notin E_k$ for any k), let d_j denote the (fixed) duration or processing time, and let t_j be the (variable) start time of operation j. Finally, let L_i be the release time and U_i the deadline of the job associated with operation i, and let R and D be the sets of operations for which release times and deadlines are defined. The problem can then be stated as:

$$\begin{array}{rcllcl}
\text{minimize} & t_n & & & & \\
\text{subject to} & & & & & \\
t_j - t_i & \geq & d_i & (i,j) & \in & A \\
t_i & \geq & 0 & i & \in & N \\
t_j - t_i \geq d_i + c_{ij} & \vee & t_i - t_j \geq d_j + c_{ji} & (i,j) & \in & E_k,\ k \in M \\
t_i & \geq & L_i, & i & \in & R \\
t_i & \leq & U_i - d_i, & i & \in & D
\end{array} \qquad \text{(P)}$$

Any feasible solution to (P) is called a schedule.

It is useful to represent this problem on a *disjunctive graph* $G := (N, A, E)$ with node set N, (directed) arc set A, and (undirected, but orientable) edge set E. The length of an arc $(i,j) \in A$ is d_i, whereas the length of an edge $(i,j) \in E$ is either $d_i + c_{ij}$ or $d_j + c_{ji}$ depending on its orientation. Each machine k corresponds to a set N_k of nodes (operations) and a set E_k of edges which together form a disjunctive clique.

For the last fifteen years, job shop scheduling problems without sequence dependent setup times have been successfully treated with the Shifting Bottleneck Procedure described in [1], improved upon in [4, 6], and extended to the case of jobs with deadlines in [3]. This is a heuristic procedure without a theoretical performance guarantee but with a solid track record of empirical performance. Its driving engine is an efficient procedure used repeatedly, for solving a certain type of one-machine problem. However, the introduction of sequence-dependent setup times changes the nature of the one-machine problems and thereby invalidates the earlier a pproach. The current paper's main contribution is to treat the one-machine problem with release times, deadlines and sequence-dependent setup times as a Traveling Salesman Problem (TSP) with time windows, and solve it by a dynamic programming algorithm designed for this purpose. The efficiency of the procedure is demonstrated on a large array of instances from the literature, mainly from the semiconductor industry.

Our paper is organized as follows: Section 2 describes the Shifting Bottleneck Procedure and explains the need for a new approach to solve its one-machine problems in the presence of sequence-dependent setup times. It then formulates this problem as an asymmetric TSP with time windows determined by the release times and due dates resulting from the dynamics of the evolving overall schedule, along with the fixed deadlines. Section 3 reduces this problem to an asymmetric TSP with a special type of precedence constraints solvable by a dynamic programming algorithm whose complexity is linear in the number of operations. Finally, section 4 discusses our computational experience and comparisons with other approaches.

2 The Shifting Bottleneck and One-Machine Problem

The Shifting Bottleneck Procedure [1] sequences the machines consecutively, one at a time, with the remaining unsequenced machines ignored (i.e. the corresponding edge sets removed) and the machines already sequenced held fixed (i.e. the corresponding edge sets replaced by directed arcs). At each step a bottleneck machine is determined from among those not yet sequenced, by solving a one machine scheduling problem for each unsequenced machine and choosing the one with maximum makespan. This bottleneck machine is then sequenced optimally by using the solution to the corresponding one-machine problem. After a machine has been sequenced, each machine in turn is freed up and resequenced, with the sequences on the remaining machines held fixed. Thus a concise statement of the procedure is as follows:

Let M_0 be the set of machines already sequenced ($M_0 = \emptyset$ at the start).

Step 1. Identify a 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 current partial schedule. If $M_0 = M$, stop; otherwise go to 1.

An improved version of the SB procedure, developed in [6], replaces Step 2 by a guided local search based on an interchange scheme and using the concept of neighborhood trees. Structural properties of the neighborhood are used to guide the search in promising directions. While the resulting procedure, SB-GLS, is computationally more expensive, the increase in cost is compensated by a significant improvement in the quality of solutions. Further improvements in solution quality can be obtained at a steeper computational cost by randomizing the choice of machines to be resequenced in step 2, and repeating this step h times for some positive integer h. This variant is called SB-RGLSh (see [6] for details).

Obviously, the success of the Shifting Bottleneck (SB) approach depends primarily on our ability to solve efficiently the one-machine problems that arise during the procedure. These problems can be described as follows. At a given stage of the SB procedure, let M_0 be the set of machines already sequenced, and let $G(M_0)$ be the disjunctive graph obtained from G by replacing all disjunctive arc sets corresponding to the machines in M_0 by the conjunctive arc sets representing the sequences chosen for those machines, and deleting all disjunctive arc sets corresponding to the machines in $M \setminus (M_0 \cup \{k\})$ for some fixed $k \in M \setminus M_0$. Then the one-machine problem to be solved for machine k can be stated as

$$\begin{array}{rcll}
\text{minimize} & t_m & & \\
\text{subject to} & & & \\
t_m - t_i & \geq & d_i + q_i, & i \in I \\
t_i & \geq & r_i, & i \in I \\
t_i & \leq & U_i - d_i - q_i & i \in I \cap D \\
t_j - t_i \geq d_i + c_{ij} & \vee & t_i - t_j \geq d_j + c_{ji} & (i,j) \in E_k,
\end{array} \qquad P(k, M_0)$$

where I is the set of jobs to be processed on machine k, $m = |I|$, r_i is the release time (or head) of job i, equal to the length $L(0,i)$ of the longest path from node 0 to node i in $G(M_0)$, and q_i is the delivery time (or tail) of job i, defined as $L(i,n) - d_i$, where $L(i,n)$ is the length of a longest path from node i to node n in $G(M_0)$.

Although this problem is $\mathcal{NP}$-complete in the strong sense, in the absence of sequence-dependent setup times and deadlines it has a very nice feature that makes it efficiently solvable in most cases. This feature consists in the fact that if $r_i = r_j$ for all $i, j \in I$, then the Longest Tail rule, which schedules the jobs in order of decreasing tail length, yields an optimal schedule. Furthermore, when the schedule produced by this rule is not optimal, then a set of jobs $J \subset I$ and a job $p \in I \setminus J$ are readily determined, such that in any feasible schedule p comes either before, or after all $j \in J$. This leads to a highly efficient branching rule (see Carlier [10]) that permits the solution of $P(k, M_0)$ with search trees typically not exceeding $O(|I|)$ nodes.

Unfortunately, in the presence of sequence-dependent setup times, this nice feature of the problem vanishes, and therefore a different approach is needed if $P(k, M_0)$ is to be solved efficiently. Notice that $P(k, M_0)$ is essentially the problem of ordering I into an optimal sequence, given upper and lower bounds on the position of each item in the sequence, as well as a minimum time-distance between each pair of items. In other words, $P(k, M_0)$ can be viewed as a TSP with time windows. This, of course, is also true in the absence of sequence-dependent setup times; but in that case the TSP feature is not significant. To see this, it suffices to point out that if the cost of traveling from city i to any other city is d_i, then the cost of a TSP tour is $\sum(d_i : i \in I)$, irrespective of the order in which the cities are visited. If there are setup times but they are not sequence-dependent, more precisely they depend only on one job instead of a pair, then they can be incorporated into the processing times d_i and the problem remains essentially unchanged. It is the presence of sequence-dependent setup costs that makes the TSP feature significant, and makes it worth stating the problem as an asymmetric TSP with time windows.

3 The TSP with Precedence Constraints

In the variant of the TSP with time windows (TSPTW) relevant to our case, we are given a set I of cities, a travel time t_{ij} between cities i and j for all pairs $i, j \in I$, and a time window $[a_i, b_i]$ for each city i, with the interpretation that city i has to be visited not earlier than a_i and not later than b_i. If city i is reached before a_i, there is a waiting time w_i until a_i; but if city i is visited later than b_i, the tour is infeasible. The objective is to find a feasible tour of minimum total duration, including the waiting times. In the case of the one-machine problem $P(k, M_0)$ stated as a TSPTW, $t_{ij} = d_i + c_{ij}$, $i, j \in I$, $a_i = r_i$, and $b_i = \min\{t^*_m, U_i\} - d_i - q_i$ for all $i \in I$, where t^*_m is the length of a feasible schedule. Since the upper bounds of all the time windows depend on the value of t^*_m, the problem in fact asks for the smallest value t_m^{**} of t_m^*, for which a feasible tour exists, as well as for an optimal tour given that $t_m^* = t_m^{**}$. Here t_m^{**} can be determined by binary search or some other procedure that solves the problem for different values of t_m^*.

The TSPTW in turn can be reformulated as a TSP with precedence constraints of a special type, for which an efficient dynamic programming algorithm has been proposed in [2] and implemented in [5]. This algorithm has the property of searching an exponentially large neighborhood in time linear in the number of cities.

To be more specific, let I be a set of cities and t_{ij} the travel time between i and j, for $i, j \in I$. Given a positive integer k and an initial ordering $\sigma = \{1, \dots, m\}$ of I, we are asked to find an optimal tour, i.e. permutation π of σ, satisfying $\pi(1) = 1$ and

$$\pi(i) < \pi(j) \text{ for all } i, j \text{ such that } i + k \leq j. \tag{1}$$

In other words, if city i precedes city j by k or more places in the initial ordering, then city i must be visited before city j in an optimal tour.

TSP's that satisfy condition (1) can be solved by dynamic programming in time linear in $m = |I|$ (though exponential in k). Furthermore, this property still holds if k is replaced by city-specific constants $k(i) > 0$, $i \in I$. The algorithm in question constructs an auxiliary network G^* with a source, a sink, and a layer of nodes for each position in the tour, such that there is a 1-1 correspondence between tours satisfying condition (1) and source-sink paths in G^*. An optimal tour satisfying (1) is then found by constructing a shortest source-sink path in G^*. Since the size of the layers is exponential in k, memory is a limiting factor in this approach. At the current state of the art, values of k beyond 20 are not practical.

One of the nice features of this approach is that when a TSP does *not* satisfy condition (1) because the constant k is not large enough (or we do not wish to invest the time required to run the algorithm for a large value of k), then the dynamic programming algorithm, no longer guaranteed to find an optimal solution, can still be used as a linear-time heuristic that examines an exponential-sized neighborhood.[5]

Next we discuss how the TSPTW can be formulated as a TSP with condition (1). Given an initial ordering of the cities, the time windows can be used to derive precedence constraints of the type desired. Indeed, in any feasible tour, city i has to precede any city j such that $a_j + t_{ji} > b_i$. If j_0 is the smallest index such that $a_j + t_{ji} > b_i$ for all $j \geq j_0$, then we can define $k(i) = j_0 - i$. Doing this for every i, we obtain a TSP with condition (1) whose feasible solutions include all feasible tours for the TSP with the time windows $[a_i, b_i]$. Since the converse is not necessarily true, the dynamic programming routine has to be amended with a feasibility check: while traversing G^*, paths that violate the upper limit of some time window must be weeded out, whereas paths that violate a lower limit of some time window must have their lengths (costs) increased by the waiting time. This check is a simple comparison that does not affect the complexity of the algorithm.

Clearly, the size of the constants $k(i)$, $i \in I$, derived in this way depends on the tightness of the time windows as well as on the quality ("distance" from the optimum) of the initial ordering. One way of choosing the latter is to sort the time windows by their midpoint. Typically this gives a reasonable-sized largest $k(i)$, which can often be improved (reduced) by heuristically rearranging (interchanging) some of the time windows.

As the TSPTW formulation of the one-machine problem has to be solved repeatedly for different values of t^*_m, the time windows themselves have to be redefined for each new t^*_m: to be more specific, as in practice we solve the problem for a sequence of diminishing values of t^*_m (see below), the time windows get tightened after each such iteration.

If the time-windows of the TSPTW problem imply precedence constraints of type (1) with a value of $k := max_i k(i)$ larger than is practical to use, then the exponential neighborhood search over the neighborhood defined by (1) with the largest affordable k serves as a heuristic, which still performs very well on TSPTW instances [5].

As a TSP tour is built by dynamic programming, the states in the dynamic program for the TSP [13] corresponding to paths specify the last node in the path and the set of nodes already visited in the path. A simple test at each node can enforce additional precedence constraints generated by the scheduling problem. Furthermore, because of the nature of constraint (1) we can be certain that any path of length l corresponding to a state in the above dynamic program will have visited nodes 1 through $l - k$, and will not have visited nodes $l + k$ through n. This allows for a constant-time test for precedence constraints, preserving the linear-time complexity of the algorithm.

The presence of delayed precedence constraints (see [5]) creates a hurdle that the dynamic program cannot handle entirely. Given a required delay of w_{ij} between the completion of operation i and the beginning of operation j, the cost of the arc from node i to node j in the corresponding TSP can be changed from c_{ij} to w_{ij} if $w_{ij} > c_{ij}$, but this cannot enforce the delayed constraint if additional jobs are scheduled between operations i and j. Still, the algorithm performs well as a heuristic to find the minimum target makespan M, which will be seen in section 4.

4 Computational Results

We have implemented our procedure in C and have run all our tests on a Sun Ultra 60 using a 360 MHz UltraSPARC-II processor. Complete results can be viewed at
`http://www.andrew.cmu.edu/~neils/tsp`.

4.1 Description of Test Data

We have tested our algorithms primarily on the rich data sets generated by Ovacik and Uzsoy [14, 15]. These instances were inspired by the actual environment in the semiconductor industry. We briefly summarize their description by the authors.

The 960 standard instances of [14] are artificially constructed classical job shop scheduling instances with the addition of setup times. Each job has to be processed on each machine exactly once. The jobs visit the machines in a predetermined sequence and all machines have sequence-dependent setup times. The machine sequence for each job is a random permutation of the machines, while the processing times and setup times are uniformly distributed random numbers. Jobs also have deadlines, uniformly distributed on an interval determined by the expected workload of the system and several other parameters.

The 1920 instances of [15] come from a semiconductor testing facility consisting of a number of testing workcenters and a brand workcenter. This corresponds to the post-burn-in portion of an actual large testing facility. Processing and setup times were derived from data gathered in the actual facility, otherwise the structure of the instances follows the pattern described in the previous paragraph.

The 1800 reentrant instances of [15] are motivated by the wafer fabrication phase of semiconductor manufacturing. Jobs pass through a basic sequence of operations which is repeated a number of times. Processing and setup times are uniformly distributed between 1 and 200.

In all three cases, the instances are grouped into six classes, which differ by certain properties of the data. Within each class, the instances are grouped into sets of 20, where each set involves the same number of jobs and the same number of machines. For more details, see [14, 15]. Data for these problems can be found at:
`http://cobweb.ecn.purdue.edu/~uzsoy2/ResearchGroup/`

4.2 Adding a Guided Local Search

As mentioned in section 2, combining the Shifting Bottleneck with a Guided Local Search on standard job shop scheduling problems improves the quality of solutions at a computational cost. While in the standard case this extra cost is usually amply justified, it was open to question whether this is still true under the changed circumstances produced by the setup times, where the one machine problems are processed by the dynamic programming routine. Therefore our first experiments were directed at settling this issue. Table 1 compares the algorithm based on the standard Shifting Bottleneck procedure with one that uses the combined algorithm on the first of the six classes of instances in [14], the data set i305. As it can be seen, the combined algorithm obtains significantly better results, at the cost of what seems a tolerable increase in computing time.

Table 1: Results with and without local search.

Average of Maximum Lateness (avg. CPU seconds)						
Problems	Jobs	Machines	with local search		without local search	
1-20	10	5	361.65	(1.58)	421.9	(0.33)
21-40	20	5	93.85	(61.36)	125.95	(58.26)
41-60	10	10	1087.10	(5.75)	1250.30	(0.42)
61-80	20	10	815.20	(103.85)	1058.25	(83.67)
81-100	10	15	1650.65	(13.88)	1899.45	(0.58)
101-120	20	15	1521.00	(111.36)	1914.50	(65.54)
121-140	10	20	2158.85	(22.26)	2508.25	(1.01)
141-160	20	20	2097.00	(105.48)	2674.25	(39.70)
Dataset i305, $k = 15$						

4.3 Step-Down versus Binary Search

As mentioned in section 3, when finding the optimal target makespan M for the one-machine scheduling problem, a binary search can be used. In practice, a “step-down” approach requires fewer than $log(M)$ steps to find this value of M, and the earlier steps can be taken more quickly by using smaller values for the parameter k used in the dynamic programming model outlined in section 3. Starting with a target makespan large enough to ensure that any sequence is feasible, the dynamic program will find a tour with a reasonable quality makespan, say, M_1. The algorithm can now be run with a target makespan of $M_1 - 1$ in an effort to improve on the makespan. If an improvement is found, it will have a makespan of $M_2 = M_1 - m$ for some positive integer m, and the next target makespan would be $M_2 - 1$. The process is repeated until no improvement is found. Because M_1 is usually not far from the optimal M, this approach takes fewer than $log(M)$ steps in practice.

Another advantage of this “step-down” approach is that a smaller value of k can be used for the first few steps of the process, when the windows are bigger and feasible solutions are easier to find. Table 2 shows this advantage on a subset of the test problems used. The first column shows the results when $k = 15$ is used for the “step-down” process, and the second column shows the results when $k = 10$ is used until no feasible solution is found, and then $k = 15$ is used to continue the “step-down” process. The running times for the second column are significantly lower, while the results are similar. The reason the results may differ is that the procedure is still only a heuristic, and a different sequence of “step-downs” may cause the algorithm to search different neighborhoods. Also, one different solution to a one-machine scheduling problem may create a different path of optimizations by the shifting bottleneck.

Table 2: Illustrating the benefit of the “step-down” search

Problem Set i305			Maximum Lateness (CPU seconds)	
Problem Number	Jobs	Machines	step-down $k = 15$	step-down $k = 10, 15$
21	20	5	515 (214.77)	493 (55.31)
22	20	5	-40 (168.70)	-158 (50.29)
23	20	5	149 (134.36)	156 (47.14)
24	20	5	-81 (162.16)	-81 (67.50)
25	20	5	4 (213.00)	-64 (53.89)
61	20	10	548 (290.49)	548 (102.89)
62	20	10	707 (408.86)	714 (154.35)
63	20	10	1035 (278.83)	1035 (111.78)
64	20	10	988 (342.44)	988 (108.47)
65	20	10	622 (376.45)	553 (108.74)
101	20	15	1511 (355.47)	1511 (101.38)
102	20	15	1519 (385.88)	1459 (99.32)
103	20	15	1487 (336.39)	1615 (109.95)
104	20	15	1934 (332.80)	1934 (128.27)
105	20	15	1756 (451.96)	1766 (140.30)
141	20	20	1871 (277.58)	1871 (108.36)
142	20	20	1902 (290.16)	1902 (94.63)
143	20	20	2223 (252.22)	2223 (89.38)
144	20	20	2136 (263.69)	2092 (104.78)
145	20	20	1987 (296.07)	2042 (97.44)

4.4 Comparisons with Ovacik-Uzsoy for different values of k

In tables 3, 4, and 5, we compare the performance of our procedure to that of Ovacik and Uzsoy [14, 15], based on a variety of dispatching rules. We also compare the performance of our procedure with two different values of the parameter k used in the dynamic programming routine to solve the one-machine problems. In both cases, $k = 10$ is used for the first part of the "step-down". The "step-down" is then continued with either $k = 12$, or $k = 15$. A summary of the results on three of the 18 data sets is shown in these tables, which is typical of all the results. For full results, see `http://www.andrew.cmu.edu/~neils/tsp`.

Table 3: Summary of results on the standard problems

Average percent improvement of total makespan vs. Ovacik and Uzsoy (avg. CPU seconds)						
Problems	Jobs	Machines	$k = 10/12$		$k = 10/15$	
Dataset i305 [14]						
1-20	10	5	16.15	(1.86)	16.15	(1.58)
21-40	20	5	17.40	(16.02)	17.25	(61.36)
41-60	10	10	24.85	(5.68)	24.85	(5.75)
61-80	20	10	21.45	(41.67)	21.90	(103.85)
81-100	10	15	28.05	(13.64)	28.05	(13.88)
101-120	20	15	24.10	(73.15)	24.50	(111.36)
121-140	10	20	31.90	(21.72)	31.90	(22.26)
141-160	20	20	26.80	(93.22)	26.75	(105.48)
Results for all 18 data sets can be viewed at: `http://www.andrew.cmu.edu/~neils/tsp`						

Of the 960 standard problems introduced in [14], 88 (9.2%) had better solutions with $k = 12$, 96 (10.0%) had better solutions with $k = 15$, and 776 (80.8%) had the same solution with both k values. Comparing to Ovacik and Uzsoy, both $k = 12$ and $k = 15$ outperformed [14] in all but two of the 960 instances (99.8%).

Of the 1920 semiconductor problems introduced in [15], 328 (17.1%) had better solutions with $k = 12$, 412 (21.5%) had better solutions with $k = 15$, and 1180 (61.5%) had the same solution with both k values. In the larger problems from this set, running times for $k = 15$

Table 4: Summary of results on semiconductor problems.

Average percent improvement of total makespan vs. Ovacik and Uzsoy (avg. CPU seconds)						
Problems	Jobs	Machines	$k = 10/12$		$k = 10/15$	
Dataset i305 [15]						
1-20	10	7	-0.05	(0.60)	0.05	(1.45)
21-40	10	12	1.10	(0.89)	1.10	(1.39)
41-60	10	17	1.10	(0.78)	1.10	(0.86)
61-80	10	22	1.25	(0.50)	1.25	(0.52)
81-100	25	7	2.55	(6.60)	2.45	(36.65)
101-120	25	12	1.10	(5.70)	1.20	(26.24)
121-140	25	17	1.20	(6.01)	1.25	(32.49)
141-160	25	22	0.80	(9.03)	1.05	(44.55)
161-180	50	7	4.10	(39.17)	4.85	(277.61)
181-200	50	12	3.15	(37.40)	3.05	(308.02)
201-220	50	17	1.00	(40.21)	0.90	(287.91)
221-240	50	22	4.10	(57.25)	3.90	(433.65)
241-260	75	7	3.75	(74.38)	4.70	(616.19)
261-280	75	12	3.95	(91.53)	3.95	(842.90)
281-300	75	17	4.95	(166.74)	4.75	(1342.48)
301-320	75	22	4.30	(124.76)	4.55	(1163.31)
Results for all 18 data sets can be viewed at: `http://www.andrew.cmu.edu/~neils/tsp`						

Table 5: Summary of results on the reentrant problems.

Average percent improvement of total makespan vs. Ovacik and Uzsoy (avg. CPU seconds)						
Problems	Jobs	Machines	$k = 10/12$		$k = 10/15$	
Dataset re305 [15]						
1-20	10	6	9.10	(0.82)	9.10	(0.81)
21-40	20	6	9.00	(3.43)	9.00	(6.35)
41-60	10	6	14.00	(2.17)	14.00	(1.94)
61-80	20	6	16.00	(18.18)	15.85	(57.50)
81-100	10	6	13.05	(1.74)	13.05	(1.68)
101-120	20	6	16.05	(10.87)	16.75	(26.27)
121-140	10	6	18.30	(7.26)	17.85	(9.92)
141-160	20	6	19.15	(51.49)	18.85	(314.54)
161-180	10	6	19.15	(6.05)	19.15	(9.74)
181-200	20	6	19.25	(38.89)	19.00	(195.10)
201-220	10	6	19.40	(24.62)	19.55	(63.03)
221-240	20	6	19.10	(134.14)	19.30	(883.97)
241-260	10	6	18.25	(12.62)	18.00	(29.06)
261-280	20	6	19.05	(69.27)	18.80	(399.02)
281-300	10	6	18.65	(42.41)	18.50	(137.18)
Results for all 18 data sets can be viewed at: `http://www.andrew.cmu.edu/~neils/tsp`						

were around 10 times longer than $k = 12$ in some cases. Comparing to Ovacik and Uzsoy, $k = 12$ outperformed [15] in 1311 instances (68.3%) and $k = 15$ outperformed [15] in 1316 instances (68.5%).

Of the 1800 reentrant problems introduced in [15], 482 (26.8%) had better solutions with $k = 12$, 461 (25.6%) had better solutions with $k = 15$, and 857 (47.6%) had the same solution with both k values. Comparing to Ovacik and Uzsoy, $k = 12$ outperformed [15] in all but 15 instances (99.2%) and $k = 15$ outperformed [15] in all but 17 instances (99.1%).

4.5 Other Comparisons

We have also tested our algorithms on a set of problems from Brucker and Thiele [8]. These are jobshop scheduling problems with setup times derived from the corresponding benchmark problems tested in [1]. Brucker and Thiele have divided the operations into groups such that each operation on the same job belongs to the same group, and there are sequence dependent setup times between operations belonging to different groups.

Instances ps01–ps05, ps06–ps10, and ps11–ps15 are 10×5 (ten jobs, five machines), 15×5, and 20×5, respectively; and instances pss12 and pss13 are also 20×5. For more details, see [8]. The last two instances were added to the set of 15 because they were the only two problems from their set that had been solved by both Focacci et al. [12] and by Buscaylet and Artigues [9].

Table 6 compares our solutions with the results from Brucker and Thiele [8], Focacci, Laborie and Nuijten [12], and Buscaylet and Artigues [9]. Brucker and Thiele's algorithm is a specialized branch and bound procedure. Focacci et al. use constraint programming with lower bounds obtained by linear programming. For the larger problems, their procedure has two phases, each allowed to run for 60 seconds. The first is aimed at minimizing the makespan, the second at minimizing total setup time with the makespan bounded by the value found in the first phase. For the smaller problems, ps01–ps05, Focacci et al. would allow the first phase to run until the optimal makespan was found (to a limit of 30 minutes).

Table 6: Comparison with a Randomized Shifting Bottleneck

CPU times in parentheses				Single-Run	Multi-Run
Instance	[8]	[12]	[9]	SB-GLS	SB-RGLS10
t2-ps01	798[1](502)	798(1800)	800(14.0)	815(1.92)	798(384)
t2-ps02	784[1](158)	784[1](88)	784(85.1)	807(1.74)	784(588)
t2-ps03	749[1](1892)	749[1](144)	764(103)	771(1.79)	749(882)
t2-ps04	730[1](189)	730[1](388)	745(209)	738(1.46)	730(554)
t2-ps05	691[1](770)	691[1](30)	703(216)	693(1.22)	693(264)
t2-ps06	1056(7200)	-	-	1027(10.8)	1018(1211)
t2-ps07	1087(7200)	-	-	1022(11.0)	1003(1374)
t2-ps08	1096(7200)	-	-	1013(9.95)	975(846)
t2-ps09	1119(7200)	-	-	1101(8.22)	1060(430)
t2-ps10	1058(7200)	-	-	1051(9.04)	1018(542)
t2-ps11	1658(7200)	-	-	1572(48.9)	1470(3033)
t2-ps12	1528(7200)	1450(60)/1448(120)[2]	1464(178)	1369(40.0)	1305(2186)
t2-ps13	1549(7200)	1669(60)/1658(120)[2]	1575(149)	1439(66.3)	1439(2506)
t2-ps14	1592(7200)	-	-	1602(42.5)	1485(2115)
t2-ps15	1744(7200)	-	-	1542(50.6)	1527(3029)
t2-pss12	1384(7200)	1367(60)/1362(120)[2]	1359(85.3)	1305(47.9)	1290(2079)
t2-pss13	1463(7200)	1522(60)/1518(120)[2]	1470(126)	1409(53.5)	1398(1864)

$k = 10/15$ for the dynamic programming subroutine.

Processors:

[8] used a Sun 4/20 workstation (20MHz). [12] used a Pentium II (300MHz).

[9] used a PC (2.8MHz). We used a Sun Ultra 60 (360MHz).

[1]These results have a guarantee of optimality.

[2]Results shown here are from each phase of a two phase procedure.

The algorithm of Buscaylet and Artigues is a tabu search performed on the best schedule from 1000 schedules generated from a strict order schedule generation scheme with a random priority rule.

As to our approach, besides the algorithm SB-GLS described in Section 2, we also used a more computation-intensive version shown in the last column of Table 6 as Multi-Run SB-RGLS10. This is a two-phase procedure whose first phase is the algorithm SB-RGLS10 outlined in Section 2, and whose second phase is a modified version of SB-RGLS10 in which the bottleneck machine is chosen at random from a uniform distribution; the second phase is repeated 40 times. This procedure is available at:
`http://www.andrew.cmu.edu/~neils/tsp`.

Comparing the data of Table 6 for the different procedures does not easily lend itself to conclusions, partly because of the incompleteness of some of the runs, partly because of the differences in the computers used. It is clear, however, that our approach compares much less favorably on the smaller 10×5 instances (ps01–ps05), than on the harder 15×5 and 20×5 instances. On the hard instances, our procedures are way ahead of all the others in terms of solution quality, and also better in terms of computing times. One possible reason for this is that the role of setup times is increased in the larger instances, since the single-machine problems become larger. The number of distinct setup times per machine grows with the square of the number of jobs, and the time needed to solve the associated TSPTW increases exponentially for most algorithms. Since the running time of the dynamic programming algorithm for the TSPTW is linear in the number of jobs (for a given k), it is not surprizing that a larger number of jobs per machine favors our procedure. Thus the results testify to the suitability of our approach to dealing with scheduling problems involving sequence-dependent setup times.

References

[1] Adams, J., Balas, E. and Zawack, D., 1988. The Shifting Bottleneck Procedure for Job-Shop Scheduling. *Management Science, 34,* 391-401.

[2] Balas, E., 1999. New Classes of Efficiently Solvable Generalized Traveling Salesman Problems, *Annals of Operations Research, 84,* 529-558.

[3] Balas, E., Lancia, G., Serafini, P. and Vazacopoulos, A., 1998. Job Shop Scheduling with Deadlines. *Journal of Combinatorial Optimization, 1,* 329-353.

[4] Balas, E., Lenstra, J.K. and Vazacopoulos, A., 1995. The One-Machine Problem with Delayed Precedence Constraints and its Use in Job Shop Scheduling. *Management Science, 41,* 94-109.

[5] Balas, E. and Simonetti, N., 2001. Linear Time Dynamic Programming Algorithms for New Classes of Restricted TSPs. *INFORMS Journal on Computing, 13,* 56-75.

[6] Balas, E. and Vazacopoulos, A., 1998. Guided Local Search with Shifting Bottleneck for Job Shop Scheduling. *Management Science, 44,* 262-275.

[7] Baptiste, P., C. Le Pape, W. Nuijten. 2001. *Constrained-based Scheduling: Applying Constraint Programming to Scheduling Problems.* Kluwer Academic Publishers.

[8] Brucker, P. and Thiele, O., 1996. A Branch and Bound Method for the General-Shop Scheduling Problem with Sequence Dependent Setup Times, *Operations Research Spektrum, 18,* 145-161.

[9] Buscaylet, F. and Artigues, C., 2003. A Fast Tabu Search Method for the Job-Shop Problem with Sequence-Dependent Setup Times, MIC 2003, The Fifth Metaheuristics International Conference, Kyoto, Japan.

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

[11] Carlier, J. and Pinson, E., 1989. An Algorithm for Solving the Job Shop Problem. *Management Science, 35,* 164-176.

[12] Focacci, F., Laborie, P., and Nuijten, W., 2000. Solving Scheduling Problems with Setup Times and Alternative Resources. In *Fifth International Conference on Artificial Intelligence Planning and Scheduling,* Breckenbridge, Colorado, 92-101.

[13] Held, M. and Karp, R.M., 1962. A Dynamic Programming Approach to Sequencing Problems. *SIAM Journal of Applied Mathematics, 10,* 196-210.

[14] Ovacik, I. and Uzsoy, R., 1994. Exploiting Shop Floor Status Information to Schedule Complex Job Shops. *Journal of Manufacturing Systems, 13(2),* 73-84.

[15] Ovacik, I. and Uzsoy, R., 1997. *Decomposition Methods for Complex Factory Scheduling Problems,* Kluwer Academic Publishers, 1997.