Heuristics for the Flow Line Problem with Setup Costs[1]

Roger Z. Ríos-Mercado
Graduate Program in Operations Research
University of Texas at Austin
Austin, TX 78712–1063
roger@bajor.me.utexas.edu

Jonathan F. Bard
Graduate Program in Operations Research
University of Texas at Austin
Austin, TX 78712–1063
jbard@mail.utexas.edu

June 1996
Revised April 1997
Accepted May 1997

[1]Submitted to *European Journal of Operational Research*

Abstract

This paper presents two new heuristics for the flowshop scheduling problem with sequence-dependent setup times and makespan minimization objective. The first is an extension of a procedure that has been very successful for the general flowshop scheduling problem. The other is a greedy randomized adaptive search procedure (GRASP) which is a technique that has achieved good results on a variety of combinatorial optimization problems. Both heuristics are compared to a previously proposed algorithm based on the traveling salesman problem (TSP). In addition, local search procedures are developed and adapted to each of the heuristics. A two-phase lower bounding scheme is presented as well. The first phase finds a lower bound based on the assignment relaxation for the asymmetric TSP. In phase two, attempts are made to improve the bound by inserting idle time. All procedures are compared for two different classes of randomly generated instances. In the first case where setup times are an order of magnitude smaller than the processing times, the new approaches prove superior to the TSP-based heuristic; for the case where both processing and setup times are identically distributed, the TSP-based heuristic outperforms the proposed procedures.

Keywords: Heuristics, flowshop scheduling, setup times, makespan

1 Introduction

In this paper, we address the problem of finding a permutation schedule of n jobs in an m-machine flowshop environment that minimizes the maximum completion time $C_{\max}$ of all jobs, also known as the makespan. The jobs are available at time zero and have sequence-dependent setup times on each machine. All problem parameters, such as processing times and setup times, are assumed to be known with certainty. This problem is regarded in the scheduling literature as the sequence-dependent setup time flowshop (SDST flowshop). Another way to represent scheduling problems is by using the standard $\alpha|\beta|\gamma$ notation (Pinedo [17]). In this regard, our problem is written as $F|s_{ijk},\ prmu|C_{\max}$, where the first field describes the machine environment (F stands for an m-machine flowshop), the second field provides details of processing characteristics and constraints (s_{ijk} stands for sequence-dependent setup times and $prmu$ means that the order or permutation in which the jobs go through the first machine is maintained throughout the system; that is, the queues in front of each machine operate according to the FIFO discipline), and the third field contains the objective to be minimized. The SDST flowshop is $\mathcal{NP}$-hard. We can see this by noting that the one machine version of the problem with zero processing times corresponds to an instance of the well-known asymmetric traveling salesman problem (ATSP).

The SDST flowshop is encountered in many manufacturing environments such as those arising in the chemical and pharmaceutical industries. For example, the use of a single system to produce different chemical compounds may require some cleansing between process runs, while the time to set up a facility for the next task may be strongly dependent on its immediate predecessor. Thus it is not always acceptable to assume that the time required to perform any task is independent of its position in the sequence.

Sequence-dependent properties are relevant in other fields as well. For example, the scheduling of aircraft approaching or leaving a terminal area can be modeled as a single-machine scheduling problem. Because the time separations between successive aircraft belonging to different fleets vary according to their respective position, sequence-dependent processing times must be allowed for a more realistic description of the problem.

Our work includes the development of two new heuristics and a local search phase. One of the proposed heuristics is based in an idea due to Nawaz et al. [15] that has been very successful for the general flowshop scheduling problem with no setup times. We extend their approach to handle this feature. The other algorithm we develop is called a greedy randomized adaptive search procedure (GRASP), which is a heuristic approach to combinatorial optimization problems that combines greedy heuristics, randomization, and local search techniques. GRASP has been applied successfully to set covering problems (Feo and Resende [6]), airline flight scheduling and maintenance base planning (Feo and Bard [5]), scheduling on parallel machines (Laguna and González-Velarde [13]), and vehicle routing problems with time windows (Kontoravdis and Bard [12]). The proposed

procedures are compared to a previously developed algorithm due to Simons [22]. His algorithm attempts to exploit the strong relationship between the SDST flowshop and the ATSP.

Another contribution of this work is the development of a lower bounding scheme for the SDST flowshop. The proposed scheme consists of two phases: in phase one, a lower bound based on the assignment (AP) relaxation of the ATSP is computed. In phase two, we attempt to improve this bound by inserting idle time. All the procedures are evaluated for two different classes of randomly generated instances. For the case where the setup times are an order of magnitude smaller that the processing times, the proposed algorithms prove superior to Simons' heuristic (`SETUP()`). For the case where both processing and setup times are identically distributed, `SETUP()` outperforms the proposed heuristics. We also found that the latter type of instances were more "difficult" to solve in the sense that the relative gap between the heuristic solution and the lower bound is significantly larger than the gap found for the former type of instances. In many of those cases near-optimal solutions were obtained.

The rest of the paper is organized as follows. A brief literature review is presented in Section 2. In Section 3 we formally describe and formulate the problem as a mixed-integer program. Heuristics and local search procedures are described in Sections 4 and 5, respectively. The lower bounding scheme is presented in Section 6. We then highlight our computational experience in Section 7 and conclude with a discussion of the results.

2 Related Work

For an excellent review of flowshop scheduling in general, including computational complexity results, see [20]. For a more general overview on complexity results and optimization and approximation algorithms involving single-machine, parallel machines, open shops, job shops, and flowshop scheduling problems, the reader is referred to Lawler et al. [14].

2.1 Minimizing Makespan on Regular Flowshops

The flowshop scheduling problem (with no setups) has been an intense subject of study over the past 25 years. Several exact optimization schemes, mostly based on branch-and-bound, have been proposed for $F||C_{\max}$ including those of Potts [18] and Carlier and Rebai [3].

Heuristic approaches for $F||C_{\max}$ can be divided into (a) quick procedures [15, 21] and (b) extensive search procedures [26, 16] (including techniques such as tabu search). Several studies have shown (e.g., [25]) that the most effective quick procedure is the heuristic due to Nawaz et al. [15]. In our work, we attempt to take advantage of this result and extend their algorithm to the case where setup times are included. Our implementation, `NEHT-RB()`, is further described in Section 4.2.

2.2 Sequence-Dependent Setup Times

Heuristics: The most relevant work on heuristics for $F|s_{ijk}, prmu|C_{\max}$ is due to Simons [22]. He describes four heuristics and compares them with three benchmarks that represent generally practiced approaches to scheduling in this environment. Experimental results for problems with up to 15 machines and 15 jobs are presented. His findings indicate that two of the proposed heuristics (`SETUP()` and `TOTAL()`) produce substantially better results than the other methods tested. This is the procedure we use as a benchmark to test our algorithms.

Exact optimization: To the best of our knowledge, no exact methods have been proposed for the SDST flowshop. However, Gupta [11] presents a branch-and-bound algorithm for the case where the objective is to minimize the total machine setup time. No computational results are reported. All other work is restricted to the 1- and 2-machine case.

2-machine case: Work on $F2|s_{ijk}, prmu|C_{\max}$ includes Corwin and Esogbue [4], who consider a subclass of this problem that arises when one of the machines has no setup times. After establishing the optimality of permutation schedules, they develop an efficient dynamic programming formulation which they show is comparable, from a computational standpoint, to the corresponding formulation of the traveling salesman problem. No algorithm is developed.

Gupta and Darrow [10] establish the $\mathcal{NP}$-hardness of the problem and show that permutation schedules do not always minimize makespan. They derive sufficient conditions for a permutation schedule to be optimal, and propose and evaluate empirically four heuristics. They observe that the procedures perform quite well for problems where setup times are an order of magnitude smaller than the processing times. However, when the magnitude of the setup times was in the same range as the processing times, the performance of the first two proposed algorithms decreased sharply.

Szwarc and Gupta [23] develop a polynomially bounded approximate method for the special case where the sequence-dependent setup times are additive. Their computational experiments show optimal results for the 2-machine case. Work on the 1-machine case is reviewed in [20].

3 Mathematical Formulation

In the flowshop environment, a set of n jobs must be scheduled through a set of m machines, where each job has the same routing. Therefore, without loss of generality, we assume that the machines are ordered according to how they are visited by each job. Although for a general flowshop the job sequence may not be the same for every machine, here we assume a *permutation schedule*; i.e., a subset of the feasible schedules that requires the same job sequence on every machine. We suppose that each job is available at time zero and has no due date (i.e., for job j ready time $r_j = 0$ and due date $d_j = \infty$). We also assume that there is a setup time which is sequence-dependent so that for every machine i there is a setup time that must precede the start of a given task that

depends on both the job to be processed (k) and the job that immediately precedes it (j). The setup time on machine i is denoted by s_{ijk} and is assumed to be *asymmetric*; i.e., $s_{ijk} \neq s_{ikj}$. After the last job has been processed on a given machine, the machine is brought back to an acceptable "ending" state. We assume that this last operation takes zero time because we are interested in job completion time rather than machine completion time. Our objective is to minimize the time at which the last job in the sequence finishes processing on the last machine, also known as *makespan*. As pointed out in Section 1, this problem is denoted by $F|s_{ijk},\ prmu|C_{\max}$ or SDST flowshop.

Example 3.1 Consider the following instance of $F2|s_{ijk},\ prmu|C_{\max}$ with four jobs.

p_{ij}	1	2	3	4
1	6	3	2	1
2	2	2	4	2

s_{1jk}	1	2	3	4
0	3	4	1	7
1	-	5	3	2
2	5	-	3	1
3	2	1	-	5
4	3	2	5	-

s_{2jk}	1	2	3	4
0	2	3	1	6
1	-	1	3	5
2	4	-	3	1
3	3	4	-	1
4	7	8	4	-

A schedule $S = (3, 1, 2, 4)$ is shown in Figure 1. The corresponding makespan is 24, which is optimal. □

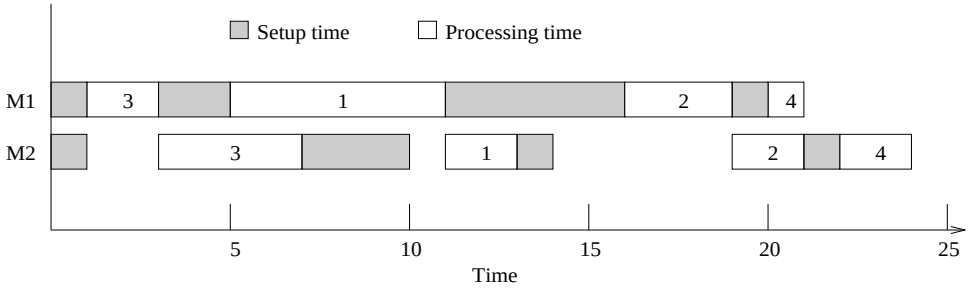

Figure 1: *Example of a* 2×4 *SDST flowshop*

3.1 Notation

In the development of the mathematical model, we make use of the following notation.

- *Indices and sets*

  m number of machines

  n number of jobs

  i machine index; $i \in I = \{1, 2, \ldots, m\}$

  j, k job indices; $j \in J = \{1, 2, \ldots, n\}$

  $J_0 = J \cup \{0\}$ extended set of jobs, including a dummy job denoted by 0

- *Input data*

  p_{ij} processing time of job j on machine i; $i \in I$, $j \in J$

  s_{ijk} setup time on machine i when job j is scheduled right before job k; $i \in I$, $j \in J_0$, $k \in J$

- *Computed parameters*

  A_i upper bound on the time at which machine i finishes processing its last job; $i \in I$,

$$A_i \quad = \quad A_{i-1} + \sum_{j \in J} p_{ij} + \min \left\{ \sum_{j \in J_0} \max_{k \in J} \{s_{ijk}\}, \sum_{k \in J} \max_{j \in J_0} \{s_{ijk}\} \right\}$$

  where $A_0 = 0$

  B_i upper bound on the initial setup time for machine i; $i \in I$,

$$B_i \quad = \quad \max_{j \in J} \{s_{i0j}\}$$

3.2 Formulation

We define the decision variables as follows:

$$\begin{aligned}
\mathbf{x}_{jk} &= \begin{cases} 1 & \text{if job } j \text{ is the immediate predecessor of job } k;\ j,k \in J_0 \\ 0 & \text{otherwise} \end{cases} \\
\mathbf{y}_{ij} &= \text{starting time of job } j \text{ on machine } i;\ i \in I, j \in J \\
C_{\max} &= \text{completion times of all jobs (makespan)}
\end{aligned}$$

In the definition of $\mathbf{x}_{jk}$, notice that $\mathbf{x}_{0j} = 1$ ($\mathbf{x}_{j0} = 1$) implies that job j is the first (last) job in the sequence for $j \in J$. Also notice that s_{i0k} denotes the initial setup time on machine i when job k has no predecessor; that is, when job k is scheduled first, for all $k \in J$. This variable definition yields what we call a TSP-based formulation.

(FS) Minimize $C_{\max}$ (1.1)

subject to

$$\sum_{\substack{j \in J_0 \\ j \neq k}} \mathbf{x}_{jk} = 1 \qquad k \in J_0 \tag{1.2}$$

$$\sum_{\substack{k \in J_0 \\ k \neq j}} \mathbf{x}_{jk} = 1 \qquad j \in J_0 \tag{1.3}$$

$$\mathbf{y}_{ij} + p_{ij} + s_{ijk} \leq \mathbf{y}_{ik} + A_i(1 - \mathbf{x}_{jk}) \qquad i \in I,\ j,k \in J,\ j \neq k \tag{1.4}$$

$$s_{i0k} \leq \mathbf{y}_{ik} + B_i(1 - \mathbf{x}_{0k}) \qquad i \in I,\ k \in J \tag{1.5}$$

$$\mathbf{y}_{mj} + p_{mj} \leq C_{\max} \qquad j \in J \tag{1.6}$$

$$\mathbf{y}_{ij} + p_{ij} \leq \mathbf{y}_{i+1,j} \qquad i \in I \setminus \{m\},\ j \in J \tag{1.7}$$

$$\mathbf{x}_{jk} \in \{0,1\} \qquad j,k \in J_0,\ j \neq k \tag{1.8}$$

$$\mathbf{y}_{ij} \geq 0 \qquad i \in I,\ j \in J \qquad (1.9)$$

Equations (1.2) and (1.3) state that every job must have a predecessor and successor, respectively. Subtour elimination constraints are given by eqs. (1.4) and (1.5). The former establishes that if job j precedes job k, then the starting time of job k on machine i must not exceed the completion time of job j on machine i ($\mathbf{y}_{ij} + p_{ij}$) plus the corresponding setup time. The latter says that if job k is the first job scheduled on machine i, then it must start after the initial setup time s_{i0k}. Constraint (1.6) assures that the makespan is greater than or equal to the completion time of the last machine, while (1.7) states that a job cannot start processing on one machine if it has not finished processing on the previous one.

In formulation (1.1)-(1.9), we assume that s_{ij0}, the time required to bring machine i to an acceptable end state when job j is processed last, is zero for all $i \in I$. Thus the makespan is governed by the completion times of the jobs only. Note that it is possible to combine $p_{ij} + s_{ijk}$ in (1.4) into a single term $t_{ijk} = p_{ij} + s_{ijk}$, but that we still need to handle the processing times p_{ij} separately in constraints (1.6) and (1.7).

4 Heuristics

We study the following heuristics for $F|s_{ijk},\ prmu|C_{\max}$.

- `SETUP()`: This is the only previously existing procedure of which we are aware for the SDST flowshop [22].
- `NEHT-RB()`: This is a modified version of a heuristic (NEH) proposed by Nawaz, Enscore and Ham [15] for $F||C_{\max}$. We extend the NEH heuristic to handle setup times.
- `GRASP()`: Our proposed greedy randomized adaptive search procedure.

4.1 Simons' `SETUP()` Heuristic

In the first of two phases of Simons' heuristics, an instance of the ATSP is built as follows. Every job is identified with a "city." Procedure `TOTAL()` computes the entries in the distance (cost) matrix as the sum of both the processing and setup times over all the machines. Procedure `SETUP()` considers the sum of setup times only. In the second phase, a feasible tour is obtained by invoking a heuristic for the ATSP. This heuristic uses the well-known Vogel's approximation method (VAM) for obtaining good initial solutions to transportation problems with a slight modification to eliminate the possibility of subtours.

It should be noted that Simons does not include a setup time for the first job to be processed. However, when initial setups are present and sequence-dependent, these must be handled explicitly.

In our formulation, this initial setup is considered so modifications were necessary to account for it. In addition, we also improved `SETUP()` by adding a local search phase. This is discussed in Section 5.

Procedure `TOTAL()`
Input: Instance of the SDST flowshop.
Output: Feasible schedule S.

Step 1. Compute $(n+1) \times (n+1)$ cost matrix as $a_{jk} = \sum_i s_{ijk} + \sum_i p_{ik}$
Step 2. Apply VAM to (a_{jk}) to obtain a tour S
Step 3. Output S
Step 4. Stop

Figure 2: *Pseudocode of Simons'* `TOTAL()` *heuristic*

Figure 2 shows the pseudo-code for the `TOTAL()` heuristic. The `SETUP()` heuristic is given by the same pseudo-code, except for a modification in Step 1 that excludes the sum of processing times, $\sum_i p_{ik}$.

Computational complexity: The computation of the cost matrix performed in Step 1 takes $O(mn^2)$ time. The application of Voguel's method to a $(n+1)$-city problem is $O(n^2)$ and hence the overall procedures `TOTAL()` and `SETUP()` have worst-case complexity of $O(mn^2)$.

4.2 `NEHT-RB()` Heuristic

The best known heuristic for the general flowshop scheduling problem with makespan minimization is NEH, due to Nawaz et al. [15]. This procedure consists of inserting a job into the best available position of a set of partially scheduled jobs; that is, in the position that would cause the smallest increment to the value of the makespan. The original worst-case complexity of the heuristic was $O(mn^3)$. Taillard [24] subsequently proposed a better way to perform the computations and came up with a complexity of $O(mn^2)$. Here we extend the NEH heuristic to handle setup times as well while maintaining the same complexity of $O(mn^2)$. We call this procedure `NEHT-RB()` (Nawaz-Enscore-Ham, modified by Taillard, extended by Ríos-Mercado and Bard).

The `NEHT-RB()` idea of building a feasible schedule is very simple. At each iteration of the algorithm there is a partial schedule S. A job h is selected from a priority list P of unscheduled jobs. Nawaz et al. ssuggest an LPT (largest processing time) rule; that is, a list where the jobs are ordered from largest to smallest total processing time. The partial schedule S and the job h define a unique greedy function $\psi(j) : \{1, 2, \ldots, |S+1|\} \rightarrow R$, where $\psi(j)$ is the makespan of the new schedule S' resulting from inserting job h at the j-th position (right before the j-th job) in S.

Here, position $|S+1|$ means an insertion at the end of the schedule. Job h is inserted into position

$$k = \text{argmin}_{j=1,\ldots,|S+1|}\{\psi(j)\};$$

that is, the position in S that has the lowest makespan value.

Procedure `NEHT-RB()`
Input: Set P of unscheduled jobs.
Output: Feasible schedule S.

Step 0. Set $S = \emptyset$
Step 1. Sort the jobs in P to form an LPT priority list
Step 2. **while** $|P| > 0$ **do**
Step 2a. Remove h, the first job from P
Step 2b. Compute $\psi(j)$ for every position $j = 1, \ldots, |S+1|$
Step 2c. Find $k = \text{argmin}_j\{\psi(j)\}$
Step 2d. Insert job h at position k in S
Step 3. Output S
Step 4. Stop

Figure 3: *Pseudocode of procedure* `NEHT-RB()`

Figure 3 shows the pseudo-code for the procedure. In Step 1 of `NEHT-RB()`, we form an LPT list with respect to the sum of the processing times of each job over all machines. In Step 2b, we use Taillard's modification. Our modification incorporates sequence-dependent setup times.

Computing the partial makespans: We now describe how to efficiently compute the greedy function $\psi(j)$ given in Step 2b of procedure `NEHT-RB()` (Figure 3). Typically, a job within brackets $[j]$ denotes the job in position j. Here, for simplicity, we drop the brackets and assume that a current schedule is given by $S = (1, 2, \ldots, k-1)$. Let h denote the job to be inserted. Define the following parameters:

- e_{ij} = the earliest completion time of job j on machine i; $(i = 1, \ldots, m)$ and $(j = 1, \ldots, k-1)$. These parameters are recursively computed as

$$\begin{aligned} e_{i0} &= 0 \\ e_{0j} &= r_j \\ e_{ij} &= \max\{e_{i-1,j},\ e_{i,j-1} + s_{i,j-1,j}\} + p_{ij} \end{aligned}$$

  where r_j denotes the release time of job j. Here r_j is assumed to be zero.

- q_{ij} = the duration between the starting time of the job j on machine i and the end of operations; $(i = m, m-1, \ldots, 1)$ and $(j = k-1, k-2, \ldots, 1)$.

$$
\begin{aligned}
q_{ik} &= 0 \\
q_{m+1,j} &= 0 \\
q_{ij} &= \max\{q_{i+1,j},\ q_{i,j+1} + s_{i,j,j+1}\} + p_{ij}
\end{aligned}
$$

- f_{ij} = the earliest relative completion time on machine i of candidate job h if inserted at the j-th position; $(i = 1, 2, \ldots, m)$ and $(j = 1, 2, \ldots, k)$.

$$
\begin{aligned}
f_{i0} &= 0 \\
f_{0j} &= r_h \\
f_{ij} &= \max\{f_{i-1,j},\ e_{i,j-1} + s_{i,j-1,h}\} + p_{ih}
\end{aligned}
$$

- $\psi(j)$ = the value of the partial makespan when adding job k at the j-th position; $(j = 1, \ldots, k)$.

$$
\psi(j) = \max_{i=1,\ldots,m} \{f_{ij} + s_{ihj} + q_{ij}\} \tag{2}
$$

where $s_{ihj} = q_{ij} = 0$ for $j = k$.

Procedure `Makespans()`
Input: Partial schedule $S = (1, 2, \ldots, k-1)$ *and job* h *to be inserted.*
Output: Vector $\psi(j)$ *with the value of the makespan when job* h *is inserted in the* j*-th position of schedule* S*.*

Step 1. Compute the earliest completion times e_{ij}
Step 2. Compute the tails q_{ij}
Step 3. Compute the relative completion times f_{ij}
Step 4. Compute values of partial makespan $\psi(j)$
Step 5. Output vector $\psi(j)$
Step 6. Stop

Figure 4: *Pseudocode of procedure for computing partial makespans*

Figure 4 shows how these computations are performed in procedure `Makespans()`. Steps 1, 2, and 3 of take $O(km)$ time each. Step 4 is $O(k \log m)$. Therefore, this procedure is executed in $O(km)$ time. Figure 5 illustrates the procedure when job h is inserted at position 3 (between jobs 2 and 3) in a partial 4-job schedule.

Computational complexity: The complexity of Step 1 of `NEHT-RB()` (Figure 3) is $O(n \log n)$. At the k-th iteration of Step 2; that is, k jobs already scheduled, Step 2a takes $O(1)$, Step 2b takes $O(km)$, complexity of Step 2c is $O(k \log k)$, and Step 2d takes $O(km)$ time. Thus, the complexity

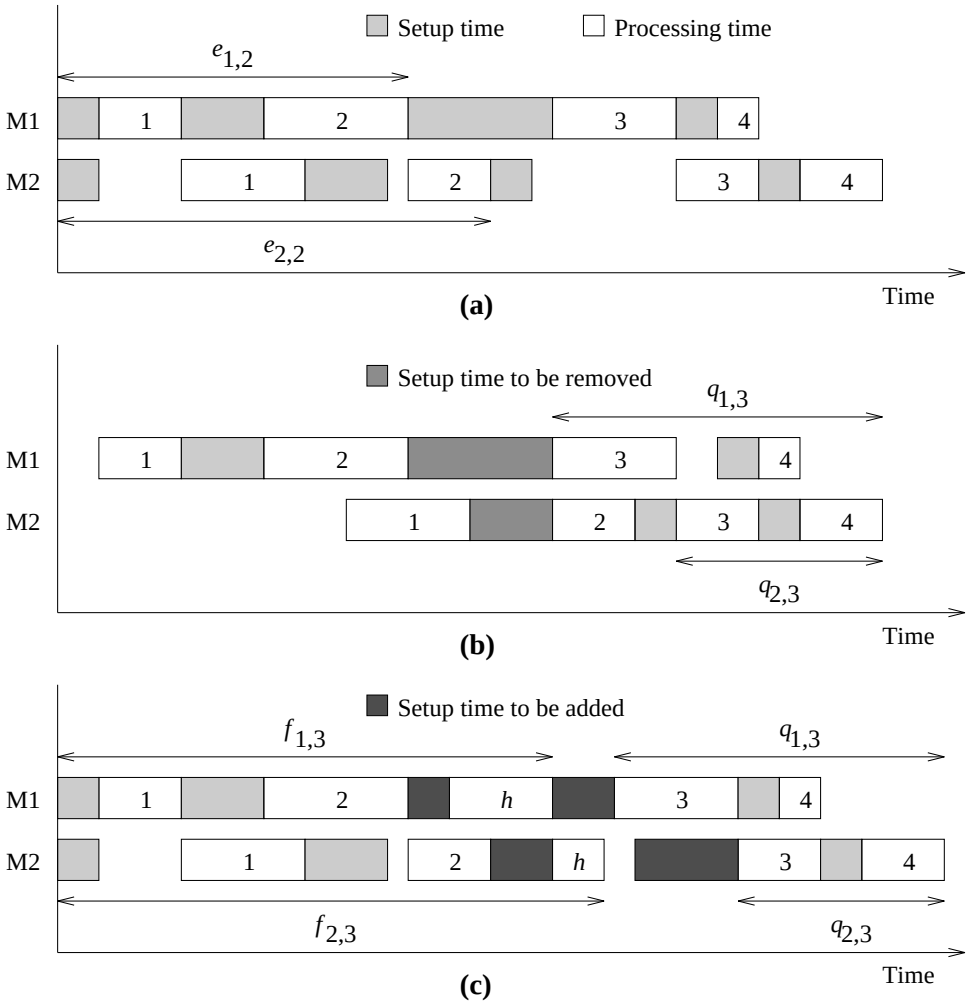

Figure 5: *Illustration of partial makespan computation*

of Step 2 at the k-th iteration is $O(km)$. This yields an overall time complexity of $O(mn^2)$ for one execution of `NEHT-RB()`.

Example 4.1 (Example 3.1 continued)
We will now illustrate how algorithm `NEHT-RB()` proceeds.

Step 0: Initialize the set of scheduled jobs $S = \emptyset$.

Step 1: Given the total processing time for each job

j	1	2	3	4
$\sum_i p_{ij}$	8	5	6	3

form the LPT priority list as follows: $P = (1, 3, 2, 4)$.

Step 2: (Iteration 1) Job 1 is selected (and removed) from P. Now $P = (3, 2, 4)$. Because there are no scheduled jobs, insert job 1 into $S = (1)$ and go to the next iteration.

(Iteration 2) Job 3 is selected (and removed) from P. Now $P = (2, 4)$, $|S| = 1$, and $\psi(k)$ (makespan value when job 3 is inserted in position k in S) is computed as follows

k	1	2
$\psi(k)$	13	18

Thus job 3 is inserted in position $k = 1$ (at the beginning of S). $S = (3, 1)$.

(Iteration 3) Job 2 is selected (and removed) from P. Now $P = (4)$, $|S| = 2$, and $\psi(k)$ is computed as follows

k	1	2	3
$\psi(k)$	22	20	23

Thus job 2 is inserted in position $k = 2$ (immediately preceding job 1). $S = (3, 2, 1)$.

(Iteration 4) Job 4 is selected (and removed) from P. Now $P = \emptyset$, $|S| = 3$, and $\psi(k)$ is computed as follows

k	1	2	3	4
$\psi(k)$	32	27	25	27

Thus job 4 is inserted in position $k = 3$ (immediately preceding job 1). $S = (3, 2, 4, 1)$.

Step 3: Output schedule $S = (3, 2, 4, 1)$ with corresponding $C_{\max}(S) = 25$.

Note that the optimal schedule is $S^* = (3, 1, 2, 4)$ with $C_{\max}(S^*) = 24$. □

4.3 GRASP

GRASP consists of two phases: a construction phase and an improving phase. During the construction phase, a feasible solution is built, one element (job) at a time. At each iteration, all feasible moves are ranked and one is randomly selected from a restricted candidate list (RCL). The ranking is done according to a greedy function that adaptively takes into account changes in the current state.

One way to limit the RCL is by its cardinality where only the top λ elements are included. A different approach is by considering only those elements whose greedy function value is within a fixed percentage of the best move. Sometimes both approaches are applied simultaneously; i.e., only the top λ elements whose greedy function value is within a given percentage ρ of the value of the best move are considered. The choice of the parameters λ and ρ requires insight into the problem. A compromise has to be made between being too restrictive or being too inclusive. If the criterion used to form the list is too restrictive, only a few candidates will be available. The

extreme case is when only one element is allowed. This corresponds to a pure greedy approach so the same solution will be obtained every time GRASP is executed. The advantage of being restrictive in forming the candidate list is that the greedy objective is not overly compromised; the disadvantage is that the optimum and many very good solutions may be overlooked. If we allow for large values of λ, in the other hand, the value of the greedy function may be compromised. This implies that the solutions found during the construction phase might not be as good in terms of their objective function value. GRASP phase 1 is applied N times, using different initial seed values to generate a solution (schedule) to the problem. In general, a solution delivered in phase 1 is not guaranteed to be locally optimal with respect to simple neighborhood definitions. Hence it is often beneficial to apply an improving phase (phase 2) where a local search technique is used to improve the current solution. Since doing the local search is expensive (in terms of its computational effort) as compared to building a feasible solution, in our implementation we apply the local search every K iterations to the best phase 1 solution in that subset. The procedure outputs the best of the N/K local optimal solutions. Figure 6 shows a flow chart of our implementation.

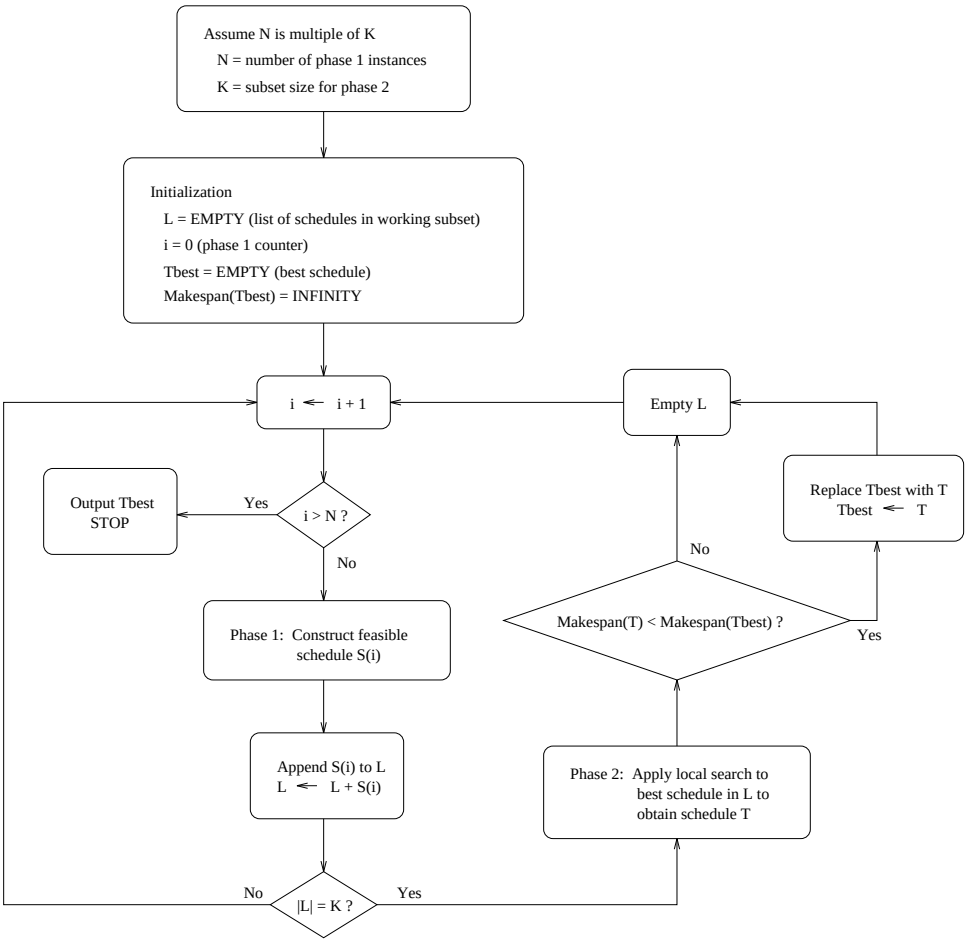

Figure 6: *Flow chart of complete GRASP algorithm*

The fundamental difference between GRASP and other meta-heuristics such as tabu search and

simulated annealing is that GRASP relies on high quality phase 1 solutions (due to the inherent worst-case complexity of the local search) whereas the other methods do not require good feasible solutions. They spend practically all of their time improving the incumbent solution and attempting to overcome local optimality. For a GRASP tutorial, the reader is referred to [7].

Below we present a GRASP for $F|s_{ijk},\ prmu|C_{\max}$ based on job insertion. This approach was found to be significantly more successful than a GRASP based on appending jobs to the partial schedule.

GRASP for the SDST Flowshop: The GRASP construction phase follows the same insertion idea as algorithm `NEHT-RB()` discussed in Section 4.2. The difference between them is the selection strategy for inserting the next unscheduled job into the partial schedule. Recall that `NEHT-RB()` always inserts the job in the best available position.

Procedure `GRASP()`
Input: Set P of unscheduled jobs and size λ of the restricted candidate list.
Output: Feasible schedule S.

Step 0. Set $S = \emptyset$
Step 1. Sort the jobs in P to form an LPT priority list
Step 2. **while** $|P| > 0$ **do**
Step 2a. Remove h, the first job from P
Step 2b. Compute $\psi(j)$ for every position $j = 1, \ldots, |S+1|$
Step 2c. Construct the RCL with the best λ positions
Step 2d. Choose randomly a position k from RCL
Step 2e. Insert job h at position k in S
Step 3. Output S
Step 4. Stop

Figure 7: *Pseudocode of* `GRASP()` *phase 1*

In `GRASP()`, the positions available for insertion are sorted by nondecreasing values of $\psi(j)$ and a restricted candidate list is formed with the best λ positions. The probabilistic strategy of `GRASP()` selects one of the positions in the RCL randomly with equal probability. The job h is inserted at the selected position into the current partial schedule S and the completion times C_{ij} for all jobs in the schedule are updated. Figure 7 shows the pseudo-code of the procedure (phase 1). Notice that `GRASP()` reduces to `NEHT-RB()` for the extreme case $\lambda = 1$.

In Step 1 of `GRASP()`, we form an LPT (largest processing time) priority list with respect to the sum of the processing times of each job over all the machines. In Step 2b, we use procedure `Makespans()`, which was seen in Section 4.2 to require $O(km)$ time.

Computational complexity: The complexity of Step 1 is $O(n \log n)$. At the k-th iteration of Step 2 (k jobs already scheduled), Step 2a takes $O(1)$, Step 2b takes $O(km)$, complexity of Step 2c is $O(k \log \lambda)$, Step 2d can be done in $O(\log \lambda)$ time, and Step 2e in $O(km)$. Thus, the complexity of Step 2 at the k-th iteration is $O(km)$. This yields a time complexity of $O(mn^2)$ for one execution of `GRASP()` phase 1. Therefore, the overall phase 1 time complexity is $O(Nmn^2)$.

Example 4.2 (Example 3.1 continued)
We now illustrate the GRASP construction phase with RCL cardinality limitation $\lambda = 2$.

Step 0: Initialize the set of scheduled jobs $S = \emptyset$.

Step 1: Given the total processing time for each job

j	1	2	3	4
$\sum_i p_{ij}$	8	5	6	3

form the LPT priority list as follows: $P = (1, 3, 2, 4)$.

Step 2: (Iteration 1) Job 1 is selected (and removed) from P. Now $P = (3, 2, 4)$. Since there are no scheduled jobs, insert job 1 into $S = (1)$ and go to the next iteration.

(Iteration 2) Job 3 is selected (and removed) from P. Now $P = (2, 4)$, $|S| = 1$, and $\psi(k)$ (makespan value when job 3 is inserted in position k in S) is computed as.

k	1	2
$\psi(k)$	13	18

Because $\lambda = 2$, RCL $= \{1, 2\}$. One is selected at random, say $k = 1$. Thus, job 3 is inserted in position $k = 1$ (at the beginning of S). $S = (3, 1)$.

(Iteration 3) Job 2 is selected (and removed) from P. Now $P = (4)$, $|S| = 2$, and $\psi(k)$ is computed as follows

k	1	2	3
$\psi(k)$	22	20	23

Form RCL=$\{1, 2\}$ and select one at random, say $k = 1$. Job 2 is inserted in position $k = 1$ (at the beginning of S). $S = (2, 3, 1)$.

(Iteration 4) Job 4 is selected (and removed) from P. Now $P = \emptyset$. For $|S| = 3$, $\psi(k)$ is computed as follows

k	1	2	3	4
$\psi(k)$	30	26	29	30

Form RCL $= \{2, 3\}$ and select one at random, say $k = 3$. Job 4 is inserted in position $k = 3$ (immediately succeeding job 3). $S = (2, 3, 4, 1)$.

Step 3: Output schedule $S = (2, 3, 4, 1)$ with corresponding $C_{\max}(S) = 29$.

Recall that the optimal schedule is $S^* = (3, 1, 2, 4)$ with $C_{\max}(S^*) = 24$. □

5 Local Search Procedures

Neighborhoods can be defined in a number of different ways, which have different computational implications. Consider, for instance, a 2-opt neighborhood definition that consists of exchanging two edges in a given tour or sequence of jobs. For this neighborhood, a move in a TSP takes $O(1)$ time to evaluate whereas a move in the SDST flowshop takes $O(mn^2)$. One of the most common neighborhoods for scheduling problems is the 2-job exchange which has been used by Widmer and Hertz [26] and by Taillard [24] for $F||C_{\max}$. We considered the 2-job exchange as well. In addition, we generalized the 1-job reinsertion neighborhood proposed by Taillard [24] for $F||C_{\max}$ to develop an L-job string reinsertion procedure. This was motivated by the presence of the sequence-dependent setup times, which suggest that subsets (or strings) of consecutive jobs might fit together in a given schedule. We tried both procedures for our problem and found that the string reinsertion uniformly outperformed the 2-job exchange, just as Taillard found the 1-job reinsertion performed better than the 2-job exchange for the regular flowshop.

5.1 L-Job String Reinsertion

Given a feasible schedule S, let $N_S^L(j, k)$ be the schedule formed from S by removing a string of L jobs starting at the j-th position and reinserting the string at position k. The neighborhood of S is given by

$$N(S) \quad = \quad \left\{ N_S^L(j, k) \ : \ 1 \leq j, k \leq n + 1 - L \right\}$$

For a given value of L, $N(S)$ is entirely defined by j and k. The size of $N(S)$ is

$$|N(S)| \quad = \quad (n - L)^2$$

An example of a 2-job string reinsertion neighbor is shown in Figure 8. The sequence on the right $S' = N_S^2(3, 1)$ is formed from S by removing the 2-job string starting at the 3-rd position (jobs 5 and 4) and reinserting it at the position 1 (immediately preceding job 2). The evaluation of all makespans can be executed in $O(n^2 m)$, using the `Makespans()` algorithm described in Section 4.2.

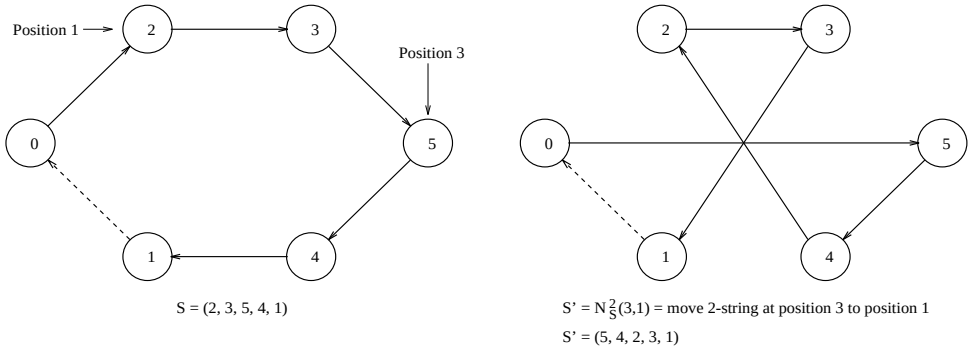

Figure 8: *Illustration of 2-job string reinsertion neighborhood*

5.2 Implementation Considerations

A primary concern in the implementation of local search procedures is how to "move" from the current feasible solution to a neighbor solution with a better objective function value. There are three fundamental ways of doing this. The first is to examine the whole neighborhood and then make a move to the "best" neighbor. The second is to examine one neighbor at a time and make a move as soon as a better solution is found. The trade-off is that in the first case we expect the incremental improvement in the objective value to be greater; however, the computational effort is higher. The third option is to examine a smaller neighborhood at the expense of the solution quality. This idea was used by Reeves in [19] for the 1-job reinsertion local search on the flowshop context. Here, we use this idea in the following way. Given a string L of jobs, there are $(n - L)$ possible sites where L can be reinserted. We observe that the evaluation of all these possible moves can be cleverly done in $O(mn^2)$, which is the same complexity of evaluating just one move. Therefore, after making this evaluation, we make the move by reinserting L in the best of these $(n - L)$ positions.

Heuristic	String size	NSC
`SETUP()`	3	Lexicographic (last)
`NEHT-RB()`	1	Lexicographic (last)
`GRASP()`	1	Lexicographic (first)

Table 1: *Parameter selection for string reinsertion procedure*

When the choice is to examine a smaller neighborhood (or subregion) as described above, we must have a criterion for selecting the "next" subregion, or in our case, how to select the next string L. The neighbor selection criteria (NSC) defines a way of choosing the next subregion to be examined. Typical examples of NSC are a lexicographic strategy and a random strategy. In the former, one sorts all unexamined subregions according to a given lexicographic rule. A lexicographic first (last) rule selects the first (last) string of the sorted list and removes it from the list of unexamined strings. In a random strategy, the next string is chosen randomly among all

unexamined candidates. We did a preliminary computation designed to fine-tune the local search procedure as a function of both the NSC and string size. This included the evaluation of these settings on over 360 instances, ranging from 2 to 6 machines and 20 to 100 jobs. The best choices of these parameters for a particular heuristic in terms of the quality of the solution found are shown in Table 1. As can be seen, a string size of 1 did better in the insertion-based heuristics, as opposed to `SETUP()`. An explanation of this is that both `NEHT-RB()` and `GRASP()` are heuristics that find a feasible solution by inserting one job at a time. This produces a feasible schedule where the interrelationship among strings of jobs may not be as strong as a feasible solution delivered by `SETUP()` which is a TSP-based heuristic. Thus, `SETUP()` benefits better from a 3-job string reinsertion.

6 Lower Bounds

Recall the MIP formulation (1.1)-(1.9) presented in Section 3. Constraint (1.7) implies that

$$\mathbf{y}_{ij} + p_{ij} \quad \geq \quad \mathbf{y}_{i-1,j} + p_{i-1,j} \quad i \in I \setminus \{m\},\ j \in J.$$

Therefore, the makespan constraint (1.6) can also be written as

$$\mathbf{y}_{ij} + p_{ij} \quad \leq \quad C_{\max} \quad i \in I,\ j \in J.$$

By relaxing the machine link constraints (1.7), the starting time for a job j on a given machine i is no longer linked to the finishing time on the previous machine. We call this new problem SFS (separable flow shop), with optimal objective function value $v(\text{SFS})$. It is clear that $v(\text{SFS}) \leq v(\text{FS})$, where $v(\text{FS})$ is the optimal value of problem FS.

Let SFS(i) be the SFS problem where all the subtour elimination and makespan constraints not related to machine i are removed. Let $S = (1, \ldots, n)$ be a feasible schedule for SFS(i). Here we assume for simplicity that the jobs in S are sequenced in order so the makespan of S is given by

$$\begin{aligned} C_{\max}(S) &= s_{i01} + p_{i1} + s_{i12} + p_{i2} + \ldots + s_{i,n-1,n} + p_{in} + s_{in0} \\ &= \sum_{j=1}^{n} p_{ij} + \sum_{j=0}^{n} s_{ij,j+1}\mathbf{x}_{j,j+1} \end{aligned}$$

where index $n+1$ corresponds to index 0 and $s_{in0} = 0$. Thus SFS(i) can be expressed as

$$\text{(SFS}(i)\text{)} \qquad \text{Minimize} \quad \sum_{j \in J_0} p_{ij} + \sum_{j \in J_0} \sum_{\substack{k \in J_0 \\ k \neq j}} s_{ijk}\mathbf{x}_{jk} \tag{3.1}$$

subject to

$$\sum_{\substack{j \in J_0 \\ j \neq k}} \mathbf{x}_{jk} = 1 \qquad k \in J_0 \tag{3.2}$$

$$\sum_{\substack{k \in J_0 \\ k \neq j}} \mathbf{x}_{jk} = 1 \qquad j \in J_0 \tag{3.3}$$

$$\mathbf{y}_{ij} - \mathbf{y}_{ik} + p_{ij} + s_{ijk} \leq A_i(1 - \mathbf{x}_{jk}) \quad j,k \in J,\ j \neq k \quad (3.4)$$

$$-\mathbf{y}_{ik} + s_{i0k} \leq B_i(1 - \mathbf{x}_{0k}) \quad k \in J \quad (3.5)$$

$$\mathbf{x}_{jk} \in \{0,1\} \quad j,k \in J_0,\ j \neq k \quad (3.6)$$

$$\mathbf{y}_{ij} \geq 0 \quad j \in J \quad (3.7)$$

for all $i \in I$.

6.1 A Lower Bounding Scheme for the SDST Flowshop

For a fixed machine i, $\sum_j p_{ij}$ in (3.1) is constant so problem SFS(i) reduces to an instance of the ATSP, where J_0 is the set of vertices and s_{ijk} is the distance between vertices j and k. Equations (3.2) and (3.3) correspond to the assignment constraints. Time-based subtour elimination constraints are given by (3.4) and (3.5). From the imposed relaxations we have

$$v(\text{SFS}(i)) \leq v(\text{SFS}) \leq v(\text{FS})$$

for all $i \in I$. Because any valid lower bound for SFS(i), call it L_i, is a valid lower bound for FS, we then proceed to compute a lower bound for every subproblem SFS(i) and obtain a lower bound on v(FS) by

$$C_{\max}^{LB} = \max_{i \in I}\{L_i\}$$

The suggested lower bounding procedure for FS is outlined in Figure 9, where procedure `lower_bound_ATSP(`c_{jk}`)` in Step 1c is any valid lower bound for SFS(i) (ATSP with cost matrix (c_{jk})).

Procedure `lower_bound_FS()` **(Phase 1)**
Input: An instance of the SDST flowshop with corresponding setup time matrix (s_{ijk}) *and processing time matrix* (p_{ij}).
Output: Lower bound $C_{\max}^{\text{LB}}$ *for the value of the makespan* $C_{\max}$.

Step 1. **for** $i = 1$ **to** m **do**
Step 1a. Let $P_i = \sum_j p_{ij}$
Step 1b. Let $c_{jk} = s_{ijk}$ be the input cost matrix for the ATSP SFS(i)
Step 1c. $L_i = P_i +$ `lower_bound_ATSP(`c_{jk}`)`
Step 2. Output $C_{\max}^{\text{LB}} = \max_i\{L_i\}$
Step 3. Stop

Figure 9: *Pseudocode of lower bounding procedure for SDST flowshop (phase 1)*

We have observed that in all of the randomly generated instances this lower bound $C_{\max}^{LB}$ is considerably better than the value $v(LP)$ of the linear programming (LP) relaxation of problem FS. However, the following example shows that this is not always the case.

Example 6.1 Consider the following 2×3 instance of the SDST flowshop.

p_{ij}	1	2	3
1	1	1	1
2	1	1	1

s_{1jk}	1	2	3
0	1	20	20
1	–	1	20
2	20	–	1
3	20	20	–

s_{2jk}	1	2	3
0	20	20	1
1	–	20	20
2	1	–	20
3	20	1	–

An optimal solution is given by $S^* = (1, 3, 2)$ with $C_{\max}(S^*) = 45$. The lower bound delivered by `lower_bound_FS()` is 6 when an exact procedure is used at Step 1c in every call to `lower_bound_ATSP()`. The LP relaxation lower bound is 8.333. □

6.2 Lower Bounds for the ATSP

Several lower bounding schemes have been proposed for ATSP. Approaches based on the assignment problem (AP) (obtained when subtour elimination constraints are relaxed), r-arborescence problem (r-ARB) (obtained when the outdegree assignment constraints are relaxed) as well as on Lagrangian relaxation are extensively discussed in [2].

It has been observed that for randomly generated instances, the AP relaxation provides a very tight bound [2]. The improvement obtained by any other scheme is very slim compared to the related computational effort. This makes AP an attractive approach when strong asymmetry is present. However, for symmetric problems ($c_{jk} \approx c_{kj}$) the results are not as good. Computational experience shows that the loss of effectiveness of exact algorithms for the symmetric case is mainly due to the weakness of the available lower bounds.

To deal with harder cases, schemes based on additive approaches have been developed. Balas and Christofides [1] proposed an additive approach based on Lagrangian relaxation. Most recently, Fischetti and Toth [8] have implemented an additive scheme that outperformed the restricted Lagrangian approach of Balas and Christofides. Their procedure yields a sequence of increasing lower bounds within a general framework that exploits several substructures of the ATSP including AP and r-ARB. We compared two lower bounding schemes for the SDST flowshop. One is based on the AP relaxation and the other on the additive approach of Fischetti and Toth. In our experiments, we observed that the improvement obtained by the latter was very small. This is attributed to the fact that for the instances having setup times that are completely asymmetric, the AP bound is very tight. This phenomenon was also observed by Fischetti and Toth for the ATSP. As the problem becomes less asymmetric the results yielded by the additive approach improve considerably. Since the data sets we are working with are assumed to have asymmetric setup times, we use the lower bounding approach based on the AP relaxation.

6.3 Improving the Lower Bound for SDST Flowshop

Procedure `lower_bound_FS()`
Input: An instance of the SDST flowshop with corresponding setup time matrix (s_{ijk}) *and processing time matrix* (p_{ij}).
Output: Lower bound $C_{\max}^{\mathrm{LB}}$ *for the value of the makespan* $C_{\max}$.

Step 1. **for** $i = 1$ **to** m **do**
Step 1a. Let $P_i = \sum_j p_{ij}$
Step 1b. Let $c_{jk} = s_{ijk}$ be the input cost matrix for the ATSP SFS(i)
Step 1c. $L_i = P_i +$ `lower_bound_ATSP`(c_{jk})
Step 2. **for** $i = 2$ **to** m **do**
Step 2a. **if** $\Delta = L_{i-1} - (L_i - p_i^{min}) > 0$
Step 2b **then** $L_i \leftarrow L_i + \Delta$
Step 2. Output $C_{\max}^{\mathrm{LB}} = L_m$
Step 3. Stop

Figure 10: *Pseudocode of lower bounding procedure for SDST flowshop*

Let C_{ij} be the completion time of job j on machine i. In particular, let T_i be the completion of time of last job on machine i; that is, T_i is the time at which machine i finishes processing. Then we have the following relation

$$C_{ij} \quad = \quad \max\{C_{i-1,j}, C_{i,j-1} + s_{i,j-1,j}\} + p_{ij}$$

In particular, if n represents the last job in the sequence, we have

$$C_{in} = \max\{C_{i-1,n}, C_{i,n-1} + s_{i,n-1,n}\} + p_{in}$$

Because $T_i = C_{in}$ we have that $T_i - p_{in} \geq T_{i-1}$. This is valid for job n, and certainly it is also valid for $p_i^{\min} = \min_{j \in J}\{p_{ij}\}$; $i \in I$. This suggests the following recursive improvement for a set $\{L_i\}$, where L_i is a valid lower bound on the completion time on machine i; $i \in I$. If $\Delta = L_{i-1} - (L_i - p_i^{\min}) > 0$, then L_i can be improved by Δ; that is, $L_i \leftarrow L_i + \Delta$. Hence $C_{\max}^{\mathrm{LB}} = L_m$ is a valid lower bound for $C_{\max}$.

We have observed that this improvement step has achieved up to a 5% reduction on the relative gap for most of the instances examined. The modified procedure is shown in Figure 10.

7 Experimental Work

All procedures were coded in C++ and compiled with the Sun C++ compiler `CC` version 2.0.1 and optimization flag set to `-O`. CPU times were obtained by calling the `clock()` function on a SPARCStation 10. To evaluate the various schemes, 20 instances of the SDST flowshop were

randomly generated for every combination

$$m \times n \quad \in \quad \{(2,4,6) \times (20,50,100)\}$$

for two different classes of data sets (available from authors).

- Data set A: $p_{ij} \in [1,99]$ and $s_{ijk} \in [1,10]$
- Data set B: $p_{ij} \in [1,99]$ and $s_{ijk} \in [1,99]$

It has been reported that many real-world instances match data set A (e.g., [10]). Data set B is included to allow us to investigate the effect on the algorithms when the setup times assume a wider range.

For each set of instances we performed several comparisons:

- *Summary statistics.* To identify dominating characteristics we compiled the following objective function value statistics
  - Number of times heuristic is best or tied for best
  - Average percentage above lower bound

  and time related statistics
  - Average CPU time
  - Worst CPU time
- *Friedman test.* This is a non-parametric test, analogous to the classical ANOVA test of homogeneity, which we apply to the null hypothesis:

  $$H_0 : \; E[S] = E[N] = E[G]$$

  under the assumption of normal distributions with a common variance, where S, N, and G are random variables corresponding to percentages above the lower bound generated by heuristics `SETUP()`, `NEHT-RB`, and `GRASP()`, respectively. The test statistic is given by

  $$T_F \quad = \quad \frac{(r-1)\{B_F - rq(q+1)^2/4\}}{A_F - B_F}$$

  $(r = 20, \; q = 3)$ where

  $$A_F \quad = \quad \sum_{i=1}^{r} \sum_{j=1}^{q} (R_{ij})^2 \quad \text{and} \quad B_F \quad = \quad \frac{1}{r} \sum_{j=1}^{q} \left(\sum_{i=1}^{r} R_{ij} \right)^2$$

  with R_{ij} being the rank (from 1 to q) assigned to heuristic j ($j =$ `SETUP()`, `NEHT-RB()`, and `GRASP()`) on problem i (lowest value gets rank of 1). In the case of ties, average ranks are used. The null hypothesis is rejected at level α if the test statistic exceeds the $1 - \alpha$ quantile of the F-distribution with $q - 1$ and $(r-1)(q-1)$ degrees of freedom.

- *Wilcoxon test.* If Friedman test is significant, that is, the null hypothesis is rejected, we attempt to identify the "best" heuristic by performing a pairwise test among all candidates. We apply the Wilcoxon signed rank test, a well-known non-parametric statistical test, to compare any two of the three heuristics. For the two heuristics `NEHT-RB()` and `GRASP()`, for instance, the null hypothesis is $E[N] = E[G]$; and the alternate hypothesis is either $E[N] > E[G]$ or $E[N] < E[G]$. The Wilcoxon test uses signed ranks of differences to assess the difference in location of two populations. The Wilcoxon statistic W is computed in the following way. First, rank the absolute differences of the original measurements, $|d_i| = |N_i - G_i|$. If any $d_i = 0$, drop it from consideration and decrease r by one. If ties occur, average the ranks of the items involved in the tie and use the average as the rank of each tied item. Second, attach the sign of $N_i - G_i$ to the rank on the i-th absolute difference, and denote this signed rank by R_i. Finally, obtain the sum W of the signed ranks:

$$W = R_1 + \ldots + R_r$$

  The null hypothesis should be rejected at the α significance level if $W > W_{1-\alpha}$ ($W < W_{1-\alpha}$) if the alternate hypothesis is $E[N] > E[G]$ ($E[N] < E[G]$). For $r \geq 10$, the critical value W_α can be approximated by

$$W_\alpha = Z(\alpha)\sqrt{r(r+1)(2r+1)/6}$$

  where $Z(\alpha)$ is the standard normal fractile such that the proportion α of the area is to the left of $Z(\alpha)$.

- *Expected utility.* This approach for comparing two or more heuristics is based on the notion that we seek a heuristic that performs well on the average and that very rarely performs poorly; that is, it is concerned with downside risk as well as expected accuracy. The procedure incorporates this attitude towards risk in a *risk-averse* utility function. As suggested by Golden and Stewart, we calculate the expected utility for each heuristic as

$$\alpha - \beta(1 - \hat{b}t)^{-\hat{c}}$$

  where $\hat{b} = s^2/\bar{x}, \hat{c} = (\bar{x}/s)^2$ are estimated parameters of a gamma distribution; $\alpha = 600$, $\beta = 100$ are arbitrarily chosen parameters and $t = 0.05$ gives a measure of risk aversion for the utility function. It should be pointed out that t must be less than $1/\hat{b}$ for each heuristic.

The application of the Friedman test, Wilcoxon test, and the expected utility approach to evaluate heuristics is proposed by Golden and Stewart [9] for the TSP.

Initial testing was done on over 360 instances for a fixed number of iterations $N = 100$, and different values for the partial search strategy subset K and size of restricted candidate list λ

(ranging from 5 to 15 and 2 to 4, respectively). The choices $K = 10$ and $\lambda = 2$ provided the best balance between the quality of the solution and the computational requirements. These settings were used to conduct the experiments; that is, we applied the construction phase $N = 100$ times and then we did the local search once every $K = 10$ iterations on the most promising solution in that subset (see Section 4.3). To evaluate the quality of the heuristics we compared the results with those obtained from our AP-based two-phase lower bounding procedure discussed in Section 6.

7.1 Experiment 1: Data Set A

		$n = 20$				$n = 50$				$n = 100$			
m		LB	UB	RG	H	LB	UB	RG	H	LB	UB	RG	H
2	Best	1193	1197	0.3	G	2495	2505	0.4	G	5554	5573	0.3	N
	Average	1088	1103	1.4	G	2706	2736	1.1	G	5274	5316	0.8	S
	Worst	1041	1073	3.1	G	2539	2593	2.1	S	4686	4754	1.5	S
4	Best	1196	1214	5.5	G	3136	3172	1.1	G	5349	5417	1.3	G
	Average	1180	1252	6.1	G	2766	2855	3.2	G	5378	5523	2.7	G
	Worst	1056	1188	12.5	N	2542	2700	6.2	N	5223	5481	4.9	N
6	Best	1293	1402	8.4	G	3138	3249	3.5	S	5629	5781	2.7	G
	Average	1243	1407	13.2	G	2879	3054	6.1	G	5448	5704	4.7	G
	Worst	1168	1391	19.1	G	2710	2990	10.3	N	5230	5621	7.5	G

Table 2: *Lower bound computations for data set A*

Table 2 shows the lower bound (LB), upper bound (UB), relative gap percentage (RG) between upper and lower bound. Also indicated is the heuristic (H) that found the upper bound for both the best and worst instances (out of 20) in terms of their relative gap. Average values are shown as well. Values are computed for each combination of m and n. Heuristics are identified by their initials (S, N, and G). We observe that most of the 2-machine instances were solved within a 1% relative gap. As the number of machines grow, the relative gap increases too.

		$n = 20$			$n = 50$			$n = 100$		
m	Statistic	S	N	G	S	N	G	S	N	G
2	Best	0	3	17	5	2	14	14	3	5
	Average % deviation	2.6	2.1	1.4	1.2	1.4	1.1	0.8	1.2	1.0
4	Best	0	2	18	1	3	16	1	1	18
	Average % deviation	9.1	7.0	6.1	4.3	3.7	3.2	3.5	3.2	2.7
6	Best	1	4	15	0	2	18	0	2	18
	Average % deviation	17.7	14.1	13.2	8.4	6.8	6.1	6.2	5.1	4.7

Table 3: *Heuristic comparison for data set A*

Summary statistics on the makespan are shown in Table 3. For each cell, entries in the first (Best) row indicate the number of times each heuristic found the best (or tied for best) solution.

Entries in the second row show the average percentage above the lower bound. We first point out that the difference between the makespans delivered by the algorithms is very small, although `GRASP()` dominates in practically all instances, the only exception being the 2×100 data sets.

CPU time (sec)										
		$n = 20$			$n = 50$			$n = 100$		
m	Statistic	S	N	G	S	N	G	S	N	G
2	Average	0.12	0.11	2.45	1.87	1.21	18.40	12.62	8.86	114.29
	Worst	0.38	0.26	2.80	3.13	2.13	22.71	20.49	14.24	149.26
4	Average	0.40	0.22	4.12	4.21	2.56	33.26	23.62	18.92	219.00
	Worst	0.81	0.39	4.92	8.46	5.07	37.45	45.94	36.61	265.53
6	Average	0.54	0.31	5.95	6.99	3.56	49.48	43.07	30.21	328.76
	Worst	0.85	0.60	6.69	13.55	6.65	60.15	67.67	52.15	437.94

Table 4: *Time statistics for data set A*

CPU time statistics are presented in Table 4. For these data sets, `NEHT-RB()` is on average 30% to 70% faster than `SETUP()` and considerably faster that `GRASP()`. `NEHT-RB()` also provides the best results regarding worst-case CPU time. We observe in general that the randomization effect introduced by `GRASP()` over `NEHT-RB()` produces an improvement of up to 3% on the quality of the solution at a cost of up to 10 times as much CPU time.

m	$n = 20$	$n = 50$	$n = 100$
2	`GRASP()` best	`GRASP()` best	`SETUP()` best
	($p < 0.0004$)	($p < 0.0444$)	($p < 0.0149$)
4	`GRASP()` best	`GRASP()` best	`GRASP()` best
	($p < 0.0011$)	($p < 0.0011$)	($p < 0.0006$)
6	`GRASP()` best	`GRASP()` best	`GRASP()` best
	($p < 0.0005$)	($p < 0.0011$)	($p < 0.0004$)

Table 5: *Wilcoxon test results for data set A*

The Friedman test was significant (at $\alpha = 0.01$) for each $m \times n$ combination. We then performed a pairwise Wilcoxon test on each combination with results displayed in Table 5. The p-value shown in the second row in every cell is the probability that the sample outcome could have been more extreme than the observed one when the null hypothesis hold. Large p-values support the null hypothesis while small p-values support the alternate hypothesis. As can be seen, all the tests are significant at $\alpha = 0.05$. Procedure `SETUP()` is found to be statistically best for the 2×100 data set, whereas in all other cases `GRASP()` dominates.

Comparisons between heuristics using the expected utility approach are given in Table 6, which indicates that expected utility values are nearly identical. This supports the hypothesis that no significant difference exists among the heuristics.

Expected utility									
	$n = 20$			$n = 50$			$n = 100$		
m	S	N	G	S	N	G	S	N	G
2	493.4	494.6	496.4	496.8	496.5	497.2	498.1	497.0	497.5
4	473.9	480.6	483.1	488.6	490.2	491.5	490.7	491.8	492.9
6	443.6	457.2	460.5	476.4	481.3	483.3	483.3	486.3	487.5

Table 6: *Expected utility comparison of heuristics for data set A*

7.2 Experiment 2: Data Set B

		$n = 20$				$n = 50$				$n = 100$			
m		LB	UB	RG	H	LB	UB	RG	H	LB	UB	RG	H
2	Best	1269	1392	9.7	G	3155	3529	11.9	S	5458	6139	12.5	S
	Average	1214	1468	20.9	G	2837	3328	17.3	S	5386	6167	14.5	S
	Worst	1057	1375	30.1	S	2668	3281	23.0	S	4792	5705	19.1	S
4	Best	1283	1613	25.7	G	3167	4109	29.7	S	5706	7350	28.8	S
	Average	1314	1823	38.7	G	2945	4079	38.5	S	5488	7431	35.4	S
	Worst	1208	1852	53.3	S	2840	4187	47.4	S	5235	7373	40.8	S
6	Best	1505	2132	41.7	N	3254	4614	41.8	G	5679	8186	44.1	S
	Average	1374	2095	52.5	G	3004	4557	51.7	G	5558	8248	48.4	S
	Worst	1261	2114	67.6	G	2700	4379	62.2	G	5348	8173	52.8	S

Table 7: *Lower bound computations for data set B*

Table 7 shows the lower bound, upper bound, relative gap percentage between upper and lower bound, and the heuristic that found the upper bound for both the best and worst instances (out of 20) in terms of their relative gap. The average relative gap percentage is shown as well. Values are computed for each combination of m and n. We observe larger relative gaps; however, the quality of the lower bound remains to be further investigated.

		$n = 20$			$n = 50$			$n = 100$		
m	Statistic	S	N	G	S	N	G	S	N	G
2	Best	7	0	13	20	0	1	20	0	0
	Average % deviation	22.4	24.8	20.9	17.3	24.5	21.3	14.5	23.8	22.0
4	Best	2	0	18	15	1	4	20	0	0
	Average % deviation	43.7	43.4	38.7	38.5	43.0	40.2	35.4	44.1	42.2
6	Best	1	2	17	4	1	15	20	0	0
	Average % deviation	58.1	56.7	52.5	52.7	54.3	51.7	48.4	55.4	53.8

Table 8: *Heuristic comparison for data set B*

Summary statistics on the makespan are shown in Table 8. Entries have the same meaning as

those described in the previous section. As can be seen, SETUP() clearly dominates the other two for the 100-job data sets. This tendency is observed in 50-job instances as well. However, as the number of machines gets large, GRASP() tends to do better, which can be observed in the 6×50 data set. For the smallest sized instances (20-job data sets) GRASP() delivers better solutions than the other two.

CPU time (sec)										
		$n = 20$			$n = 50$			$n = 100$		
m	Statistic	S	N	G	S	N	G	S	N	G
2	Average	0.11	0.12	2.50	1.19	1.53	19.95	6.75	9.58	130.79
	Worst	0.17	0.21	2.76	1.80	3.60	23.70	9.56	14.91	146.75
4	Average	0.23	0.20	3.96	1.85	2.12	29.52	10.43	13.71	178.83
	Worst	0.37	0.46	4.34	2.95	4.99	33.44	18.16	30.24	205.56
6	Average	0.28	0.27	5.20	2.57	2.44	37.42	15.92	16.86	219.23
	Worst	0.50	0.78	5.90	4.31	4.42	46.78	24.28	38.83	259.28

Table 9: *Time statistics for data set B*

CPU time statistics are presented in Table 9. We observe that, on average, NEHT-RB() and SETUP() take the same amount of time, both of them being considerably faster than GRASP(). It can also be learned from the table that SETUP() has a better empirical worst-case time behavior than NEHT-RB(). We observe in general that the randomization effect introduced by GRASP() over NEHT-RB() produces an improvement of up to 12% on the quality of the solution at a cost of up to 15 times as much CPU time.

m	$n = 20$	$n = 50$	$n = 100$
2	GRASP() best ($p < 0.0071$)	SETUP() best ($p < 0.0005$)	SETUP() best ($p < 0.0004$)
4	GRASP() best ($p < 0.0006$)	SETUP() best ($p < 0.0085$)	SETUP() best ($p < 0.0004$)
6	GRASP() best ($p < 0.0004$)	GRASP() best ($p < 0.0242$)	SETUP() best ($p < 0.0004$)

Table 10: *Wilcoxon test results for data set B*

The Friedman test was significant at the $\alpha = 0.01$ level for each combination of m and n. Wilcoxon test was then performed between each pair of heuristics (for every combination). These results are shown in Table 10. It is found that SETUP() outperforms the other two heuristics in all the 100-job instances. This is also true for the 2×50 and 4×50 instances. For the 6×50, and all the 20-job data sets, GRASP() is superior.

Comparisons between heuristics using the expected utility approach are given in Table 11. From this table, we observe that SETUP() is the most accurate (in the 2×50, 4×50, and all the 100-job

Expected utility									
	$n = 20$			$n = 50$			$n = 100$		
m	S	N	G	S	N	G	S	N	G
2	423.7	411.9	429.6	445.4	413.5	428.5	456.0	418.1	426.3
4	298.7	299.6	331.5	334.4	303.1	323.4	357.0	297.2	311.3
6	162.1	174.8	220.9	221.6	205.9	231.9	263.6	198.5	214.9

Table 11: *Expected utility comparison of heuristics for data set B*

instances) and that the rankings coincide with those determined from the previous results.

8 Conclusions

We have proposed two new insertion-based heuristics for $F|s_{ijk}, prmu|C_{\max}$. Both procedures, `NEHT-RB()` and `GRASP()`, were extensively evaluated and compared with the only existing heuristic, TSP-based `SETUP()`, for this class of scheduling problem.

As part of the study a local search procedure based on string reinsertion was presented and included in the heuristic computations. Another contribution of this work centered on the development of a lower bounding scheme derived from the additive approach for the ATSP. An improvement phase based on idle time insertion was included as well. The lower bound obtained by the enhanced scheme was found to be marginally better than the LP relaxation lower bound.

For data set A, the TSP-based heuristic worked better on the larger 2-machine instances; however, when the number of machines grows, the insertion-based heuristics `NEHT-RB()` and `GRASP()` dominated. This stems from the fact that the fewer the number of machines, the more the problem resembles an ATSP so a TSP-based procedure should do well. Recall that in `SETUP()` the distance between jobs is computed as the sum of the setup times between jobs over all the machines. In the extreme case where there is only one machine, the problem reduces entirely to an instance of the ATSP. As more machines are added, the sum of setup times becomes less representative of the "distance" between the jobs. How small does the number of machines have to be for `SETUP()` to do better than the insertion-based heuristics depends not only on the number of jobs, but on the magnitude of the setup times as well. In data set A, we observe a threshold value of $m = 2$ or 3. However, for data set B, `SETUP()` was found to outperform the others with respect to both makespan (especially for the 50- and 100-job data sets) and CPU time. This implies a threshold value of $m > 6$. One explanation of the better performance of `SETUP()` on the larger instances of data set B is as follows. Both insertion-based heuristics include a makespan estimation routine that has the setup costs as part of its performance measure; there is no other explicit treatment to the setups in the heuristics. Since the job insertion decision is made one job at a time, while the sequence-dependent setup time is dictated by the interrelationships of an entire sequence of

jobs, a TSP-based heuristic tends to do better than this insertion-style methods, specially when the number of machines is small.

`SETUP()` and `NEHT-RB()` run considerably faster that `GRASP()`. This is to be expected because they are deterministic algorithms and will deliver a unique solution for each instance. By increasing the iteration counter in `GRASP()`, more and perhaps better solutions can be found.

Our computational study also revealed that data set B instances appeared to be harder to solve. We observed that while our heuristics delivered near-optimal solutions for several of the data set A instances, the best solution (for data set B) had a relative gap on the average of 15-22%, 35-42%, and 48-55% for the 2-, 4-, and 6-machine instances, respectively. Nevertheless, further work remains to be done to determine the quality of the lower bound.

9 Acknowledgments

The research of Roger Ríos-Mercado was partly supported by the Mexican National Council of Science and Technology (CONACyT) and by a Continuing Fellowship from The University of Texas at Austin. Jonathan Bard was supported by a grant from the Texas Higher Education Coordinating Boards' Advanced Research Program. We also thank Matthew Saltzman for allowing us to use his C implementation of the dense shortest augmenting path algorithm to solve AP, and Mateo Fischetti and Paolo Toth for providing their FORTRAN code to solve r-SAP.

We also thank an anonymous referee whose suggestions helped improve the presentation of this paper.

References

[1] Balas, E., and Christofides, N., "A restricted Lagrangean approach to the traveling salesman problem", *Mathematical Programming* 21/1 (1981) 19–46.

[2] Balas, E., and Toth, P., "Branch and bound methods", in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnoy Kan and D.B. Shmoys (eds.), *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons, Chichester, 1990, 361–401.

[3] Carlier, J., and Rebai, I., "Two branch and bound algorithms for the permutation flow shop problem", *European Journal of Operational Research*, 90/2 (1996) 238–251.

[4] Corwin, B.D., and Esogbue, A.O., "Two machine flow shop scheduling problems with sequence dependent setup times: A dynamic programming approach", *Naval Research Logistics Quarterly* 21/3 (1974) 515–524.

[5] Feo, T.A., and Bard, J.F., "Flight scheduling and maintenance base planning", *Management Science* 35/12 (1989) 1415–1432.

[6] Feo, T.A., and Resende, M.G.C., "A probabilistic heuristic for a computationally difficult set covering problem", *Operations Research Letters* 8/2 (1989) 67–71.

[7] Feo, T.A., and Resende, M.G.C., "Greedy randomized adaptive search procedures", *Journal of Global Optimization* 6 (1995) 109–133.

[8] Fischetti, M., and Toth, P., "An additive bounding procedure for the asymmetric traveling salesman problem", *Mathematical Programming* 53/2 (1992) 173–197.

[9] Golden, B.L., and Stewart, W.R., "Empirical analysis of heuristics", in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnoy Kan and D.B. Shmoys (eds.), *The Traveling Salesman Problem*, John Wiley & Sons, New York, 1990, 207–249.

[10] Gupta, J.N.D., and Darrow, W.P., "The two-machine sequence dependent flowshop scheduling problem", *European Journal of Operational Research* 24/3 (1986) 439–446.

[11] Gupta, S.K., "n jobs and m machines job-shop problems with sequence-dependent set-up times", *International Journal of Production Research* 20/5 (1982) 643–656.

[12] Kontoravdis, G., and Bard, J.F., "A randomized adaptive search procedure for the vehicle routing problem with time windows", *ORSA Journal on Computing* 7/1 (1995) 10–23.

[13] Laguna, M., and González-Velarde, J.L., "A search heuristic for just-in-time scheduling in parallel machines", *Journal of Intelligent Manufacturing* 2 (1991) 253–260.

[14] Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D., "Sequencing and scheduling: Algorithms and complexity", in: S.S. Graves, A.H.G. Rinnooy Kan and P. Zipkin (eds.), *Handbook in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory*, North-Holland, New York, 1993, 445-522.

[15] Nawaz, M., Enscore, E.E., Jr., and Ham, I., "A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem", *OMEGA The International Journal of Management Science* 11/1 (1983) 91–95.

[16] Nowicki, E., and Smutnicki, C., "A fast tabu search algorithm for the flow shop problem", Report 8/94, Institute of Engineering Cybernetics, Technical University of Wrocław, 1994.

[17] Pinedo, M., *Scheduling: Theory, Algorithms, and Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[18] Potts, C.N., "An adaptive branching rule for the permutation flow-shop problem", *European Journal of Operational Research* 5/1 (1980) 19–25.

[19] Reeves, C.R., "Improving the efficiency of tabu search for machine sequencing problems", *Journal of the Operational Research Society* 44/4 (1993) 375–382.

[20] Ríos-Mercado, R.Z., "Optimization of the flow shop scheduling problem with setup times", PhD thesis, University of Texas at Austin, Austin, TX, 1997.

[21] Sarin, S., and Lefoka, M., "Scheduling heuristics for the n-job m-machine flow shop", *OMEGA The International Journal of Management Science* 21/2 (1993) 229–234.

[22] Simons, J.V., Jr., "Heuristics in flow shop scheduling with sequence dependent setup times", *OMEGA The International Journal of Management Science* 20/2 (1992) 215–225.

[23] Szwarc, W., and Gupta, J.N.D., "A flow-shop with sequence-dependent additive setup times", *Naval Research Logistics Quarterly* 34/5 (1987) 619–627.

[24] Taillard, E., "Some efficient heuristic methods for the flow shop sequencing problem", *European Journal of Operational Research* 47/1 (1990) 65–74.

[25] Turner, S., and Booth, D., "Comparison of heuristics for flow shop sequencing", *OMEGA The International Journal of Management Science* 15/1 (1987) 75–85.

[26] Widmer, M., and Hertz, A., "A new heuristic method for the flow shop sequencing problem", *European Journal of Operational Research* 41/2 (1989) 186–193.