Heuristic Procedures for Scheduling Job Families with Setups and Due Dates

Kenneth R. Baker

The Amos Tuck School of Business Administration, Dartmouth College, Hanover, New Hampshire 03755

Received June 1998; revised April 1999; accepted 13 July 1999

Abstract: This paper examines heuristic solution procedures for scheduling jobs on a single machine to minimize the maximum lateness in the presence of setup times between different job families. It reviews the state of knowledge about the solution of this problem, which is known to be difficult to solve in general, and examines natural solution approaches derived from some of the underlying theory. The emphasis is on the design and computational evaluation of new heuristic procedures. © 1999 John Wiley & Sons, Inc. Naval Research Logistics 46: 978–991, 1999

Keywords: production scheduling; batch manufacturing; heuristic procedures

1. INTRODUCTION

A basic trade-off in batch manufacturing involves efficiency criteria and due-date criteria. When we pursue high levels of efficiency, we tend to make batch sizes large, so that many items can be run on the same setup. However, when resources are committed to long production runs for one item, other items may get delayed past their due dates. On the other hand, when we seek good due-date performance, we tend to make batch sizes small, so that priorities can be shifted to jobs that face the most urgent due-date pressures. However, this shifting around may require numerous setups and lead to a loss of productive efficiency. In the long run, this loss of effective capacity may lead to diminished ability to meet due dates. Thus, there is an inherent conflict between efficiency and due-date performance, a conflict that presents a challenge to any scheduling procedure that is used for short-term scheduling of production batches.

In this paper, we examine a scheduling problem which contains this trade-off in one of its most elementary forms. The problem is quite difficult to optimize, so there is good reason to look for heuristic procedures that will reliably produce good solutions. We describe two heuristic procedures that are suggested by the underlying theory and that perform well under certain conditions. Then we develop new heuristic procedures that enhance our ability to find good solutions quickly. Our computational tests suggest that these procedures perform quite well.

The computational work itself provides some useful lessons. In order for computational tests to be meaningful, they should avoid redundant test conditions, and they should attempt to find conditions under which the solution procedure is most severly challenged. In our case, the design of a useful set of experiments takes special care and illustrates broader principles of testing that should be applicable in other scheduling problems.

 CCC 0894-069X/99/080978-14

The specific model for this problem is the classical single machine model, with n jobs simultaneously available for processing. This model is relevant when there is effectively one stage of production or when the production process is dominated by a single bottleneck machine. But even if we are ultimately interested in more complex situations, the single machine model provides us with an opportunity to discover heuristic rules, to test conjectures, and to develop working principles that generalize to more complicated models.

We assume that the jobs being scheduled are identified with distinct families and that setups are required when the machine changes from one family to another. This feature gives rise to the so-called *family scheduling model*. For this model we adopt the objective of minimizing the maximum job lateness. In other words, we want to minimize the worst case of a missed due date in the schedule. The maximum lateness criterion, together with the presence of family setups, gives rise to the fundamental trade-off described earlier.

To specify the problem more formally, we let f denote the number of families, and we let n_i denote the number of jobs in family i, where $n_1 + n_2 + \cdots + n_f = n$. Additionally, we use the subscript pair (i, j) to identify the jth job from family i, so that we can denote the processing time and the due date of job (i, j) as p_{ij} and d_{ij}, respectively. The setup time, common to all families, is denoted s.

In a particular schedule, job (i, j) will have completion time C_{ij} and lateness defined by $L_{ij} = C_{ij} - d_{ij}$. The performance measure for the schedule is $L_{\max} = \max\{L_{ij}\}$. For this criterion Bruno and Downey [2] have shown that the problem of minimizing $L_{\max}$ is NP-hard in general—that is, an efficient solution algorithm is unlikely to exist. Computationally, problems containing up to 50 jobs are within reach of optimization algorithms, using modest amounts of computer time, as demonstrated by Hariri and Potts [5]. However, for problems containing more than 50 jobs, optimization is not a reliable approach. Such problem sizes warrant the use of intelligent heuristics in order to find solutions.

In the next section we review what research has revealed about this problem and about the solution approaches that we already know. The following sections introduce new heuristic procedures, and thereafter we report on computational tests that describe the effectiveness of these procedures. A final section summarizes the main findings and conclusions.

2. EXISTING THEORY

2. 1. Earliest Due Date Sequencing

If there were no setup times in the problem, then the solution would be simple: We could minimize $L_{\max}$ by sequencing jobs in due date order. This result, due to Jackson [6], is usually called the Earliest Due Date (EDD) Rule, as given below.

PROPERTY 1 (EDD Rule): Given a set of n jobs, with known processing times and due dates, the minimum value of $L_{\max}$ is achieved by sequencing the jobs in nondecreasing order of their due dates.

The implication of this result for the family scheduling model is that when setup times are quite small, the EDD sequence should provide a good heuristic solution.

A related feature of the EDD Rule is that it holds for jobs within a given family, as shown by Monma and Potts [8]. We restate their result below.

PROPERTY 2 (EDD within families): There exists an optimal schedule such that the jobs within each family are sequenced in nondecreasing order of their due dates.

This result tells us that if we form batches of jobs, with each batch composed of jobs from a common family, then the jobs in a batch should appear in EDD order. Property 2 does not provide enough information to build an entire schedule, but we can imbed it in other schemes (such as we discuss later) for heuristic scheduling. The practical implication of Property 2 is to focus scheduling decisions on choosing a family rather than choosing a job. For example, a scheduler relying on dispatching will need a rule for selecting a job whenever the machine becomes free. In light of Property 2, the rule should aim at determining which family is most critical; within that family the job with the earliest due date should come next. In light of Property 2, we can also assume that the jobs in each family are initially numbered in EDD order, so that $d_{ij} \leq d_{i,j+1}$.

2. 2. Family Sequencing and Batch Sequencing

When setup times are negligible, the EDD sequence will be optimal, as noted earlier. At the other extreme, when setup times are quite large, it makes sense to schedule each family as a single batch. We refer to a solution in which each family is scheduled in one batch as a *Group Technology (GT) sequence*. The optimal GT sequence is not difficult to construct. Define the family due date d_i as follows:

$$d_i = \min_j \{d_{ij} + q_{ij}\},$$

where q_{ij} represents the processing time in family i that occurs after job (i, j), sometimes called the "tail" of job (i, j).

$$q_{ij} = \sum_{k>j} p_{ik}.$$

As noted by Potts and Van Wassenhove [9], we can construct the optimal GT sequence by first sequencing the jobs in nondecreasing order of their due dates within each family and then sequencing the families in nondecreasing order of their family due dates. (For convenience, we refer to the optimal GT sequence simply as the GT sequence in the material that follows.)

The concept of sequencing by family due dates has broader applicability in schedules where families are split among two or more batches. Let k denote the batch number in sequence, and let ρ_k and δ_k denote batch processing times and due dates, respectively. That is, ρ_k is simply the sum of the processing times in batch k, and

$$\delta_k = \min_j \{d_{kj} + q_{kj}\},$$

where j is the index of a job in batch k and

$$q_{kj} = \rho_k - (p_{k1} + p_{k2} + \cdots + p_{kj}).$$

(Here, in the notation d_{kj}, we use the first subscript to denote batch number rather than family number. This adaptation of the original notation will sometimes be convenient, and should not be confusing in context.) Using these batch-related parameters, Webster and Baker [11] observe that the EDD Rule adapts to sequencing batches in the following way.

PROPERTY 3 (EDD Rule for batches): There exists an optimal schedule such that the batches are sequenced in nondecreasing order of their batch due dates.

This result tells us that whenever we schedule batches of jobs, the batches should be ordered according to the EDD Rule for batches.

Although families may be split among several batches, the GT sequence may still produce an optimal solution to the general problem. Webster and Baker [11] give sufficient conditions for the GT sequence to solve the general problem, and two cases are of special interest: identical due dates or large setup times. Below we state these results more formally.

When all the due dates are the same, the maximum lateness occurs for the last job in the schedule, and, obviously, it is desirable to have the minimum number of batches. The GT sequence is therefore optimal, although there are several other sequences that will minimize the number of batches.

PROPERTY 4 (GT for identical due dates): If all due dates are identical, then the GT sequence achieves the minimum value of $L_{\max}$.

The implication of Property 4 is that when the variability among due dates is small, the GT sequence should provide a good heuristic solution.

When the setup time is large, there is also an incentive to minimize the number of batches. Webster and Baker [11] derive a critical value S^*, which is sufficient to make the GT sequence optimal. The critical value can be defined as follows:

$$S^* = \max_i [\max_j \{d_{ij} + q_{ij}\} - d_i],$$

where it is relevant to note that the value S^* depends on the problem data.

PROPERTY 5 (GT for large setups): If $s \geq S^*$, then the GT sequence achieves the minimum value of $L_{\max}$.

The implication of Property 5 is that when the setup time is relatively large (close to S^*), then the GT sequence should provide a good heuristic solution.

In summary, existing theory tells us that the $L_{\max}$ problem is quite difficult to solve, but there are two obvious heuristic solutions to the problem, namely, the EDD sequence and the GT sequence. Furthermore, these two heuristic methods actually produce optimal solutions in some identifiable special cases—when setups are negligible, when setups are large, and when due dates are identical.

No standard terminology has existed for this scheduling model. For example, Mason and Anderson [7] and Williams and Wirth [12], who study a different criterion for the same model, refer to job *classes* (rather than families) and schedules made up of *runs* (rather than batches.) Potts and Van Wassenhove [9] refer to families and batches in their survey paper, which has influenced much of the research that has followed, and in recent articles a consensus seems to be emerging. Schutten et al. [10], who study a variation of this same model with dynamic job arrivals, refer to families and batches. Ghosh and Gupta [3] and Hariri and Potts [5], who deal with different aspects of the same problem we address here, also refer to families and batches. In light of the fact that each of these papers has an author who once used different terminology, it seems that families and batches are becoming standard terms, and that is the terminology we adopt.

3. TWO HEURISTIC PROCEDURES

The EDD sequence and the GT sequence are easy to construct, and, as heuristic procedures, they provide near-optimal performance in certain special cases. With respect to the size of the setup times in a given problem, the GT sequence should provide a good solution when the setup time approaches S^*, whereas the EDD sequence should provide a good solution when the setup

time approaches zero. Suppose we define the *setup factor* as $u = s/S^*$. Then we know that the GT sequence is optimal for $u \geq 1$ and that the EDD sequence is optimal for $u = 0$.

With respect to the due dates in a given problem, the GT sequence should provide a good solution when the due dates are nearly identical. Suppose we use the range of the due dates to quantify their variability. Define the relative *due date range* as $r = (d_{\max} - d_{\min})/\sum p_{ij}$. Then we know that the GT sequence is optimal for $r = 0$. In addition, when r grows very large, the GT sequence and the EDD sequence both deliver good solutions.

In order to portray the performance of heuristic procedures, imagine a graph with coordinate axes for u and r, as shown in Figure 1. The horizontal axis is the u-axis, covering the interval (0, 1). The vertical axis is the r-axis, with no specific upper limit. From the discussion above, we know that the EDD sequence is optimal along the vertical axis, and the GT sequence is optimal along the horizontal axis, as well as along the right-hand vertical boundary at $u = 1$.

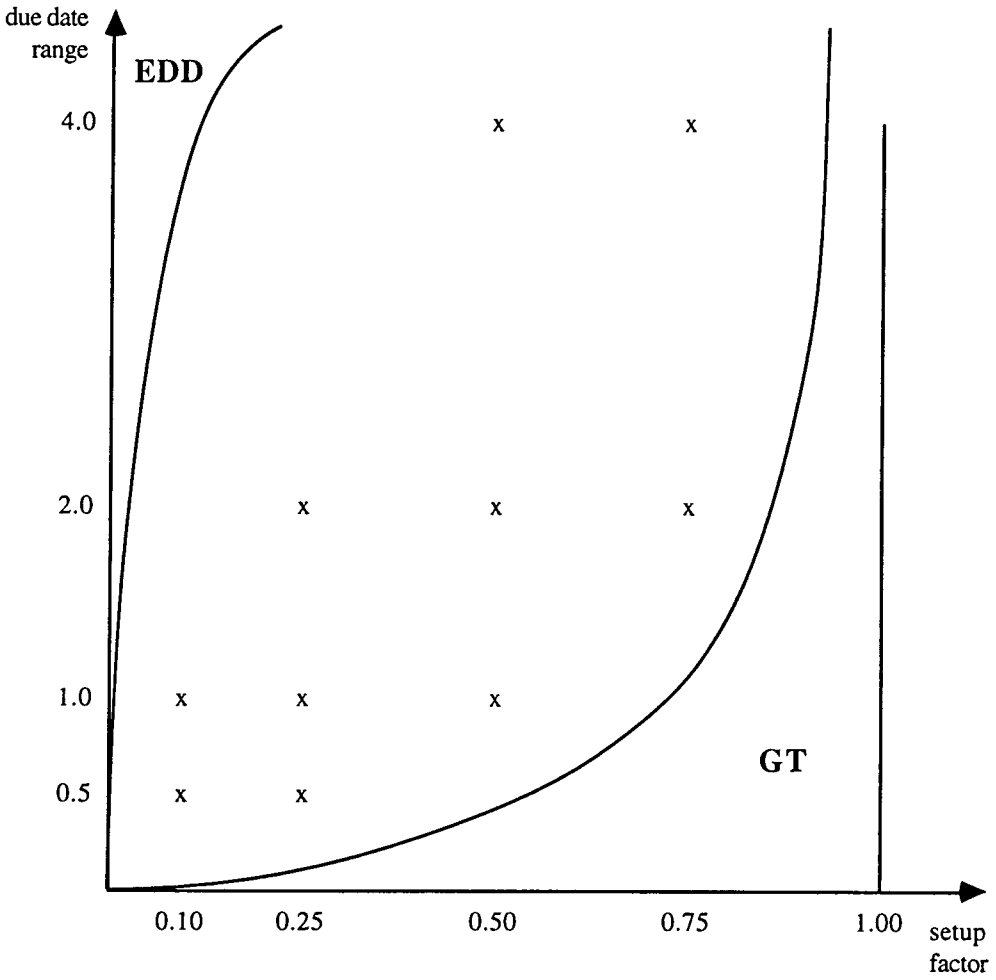

Figure 1. Map of due date and setup parameters.

A set of exploratory runs with randomly-generated problem data suggests that the EDD sequence is likely to produce optimal solutions in a region close to the r-axis. Similarly, the GT sequence is likely to be optimal in a region close to the lower and right-hand side boundaries. These regions of ''likely'' optimality are sketched in Figure 1. They are not shown as symmetric because it appears that the performance of the EDD sequence deteriorates rapidly as the setup factor increases, but the performance of the GT sequence is somewhat more robust for small due date ranges or large setup factors. Conceptually, optimal solutions, or very good suboptimal solutions, can be obtained in these two regions by constructing the EDD sequence and the GT sequence and selecting the better of the two solutions. The region in between is more problematic. Here, both heuristic procedures deteriorate in performance.

4. THE GAP HEURISTIC

The EDD Rule looks only at due dates and ignores the implications of family membership for setup times. It works poorly at times because it allows too many setups to occur. Intuitively, if the due dates of two jobs from the same family are close enough, then they should be scheduled in the same batch. Separating them only means scheduling an additional setup, which causes one of the two jobs to be later than if they had stayed in the same batch. Only if consecutive jobs from the same family have due dates sufficiently far apart should we consider placing them in separate batches. A rigorous statement of this intuitive property was derived by Webster and Baker [11]. Let i and k denote two separate batches from the same family.

PROPERTY 6 (Gap condition): There exists an optimal schedule such that if batch i precedes batch k, and the two batches belong to the same family, then

$$\delta_i \leq \Delta_{i+1,k-1} + s < d_{k1},$$

where Δ_{ik} represents the due date of batches i through k, defined as

$$\Delta_{ik} = \min_{i \leq u \leq k} \left\{ \delta_u + \sum_{t=u+1}^{k} (s + \rho_t) \right\}.$$

The implication of Property 6 is that two batches, i and k, belonging to the same family, can be separated if the due date of batch i and the due date of the first job in batch k are sufficiently far apart, i.e., if they are separated by enough of a gap. In effect, Property 6 tells us when a job is not ''splittable'' from the batch of the same family that precedes it. Of course, the difficulty is that we do not know at the outset which batches will be scheduled between i and k.

A simple heuristic can be constructed around the Gap condition. Start with the jobs in EDD order and place them, one at a time, into the sequence. Each time a job is placed in the sequence, it is added to the last batch if its family matches the family of the last batch. If not, the next alternative is to consider adding the job to the latest batch already scheduled for its family. If such a batch exists, then the job is added if the Gap condition fails. Otherwise, the job is added to the end of the sequence, initiating a new batch. Whenever a job is placed in some batch other than the last, the batches in the partial schedule are resequenced to conform to Property 3 (EDD for batches).

This is a quick procedure for constructing a schedule, in that once a job is placed into the schedule it is never removed (although it may be resequenced.) In addition, batches are expanded, but never split, in the process of executing the algorithm. Because the procedure considers the

jobs one by one in EDD order, and, because it uses a criterion for combining jobs into batches, it offers the opportunity to combine some of the positive features of both the EDD sequence and the GT sequence. Finally, the procedure guarantees that the schedule it ultimately produces will satisfy Properties 2 and 3 (EDD within families and EDD for batches), although it may not satisfy Property 6 (Gap condition).

5. IMPROVEMENT ROUTINES

The Gap Heuristic tends to schedule too many batches. Early in the procedure, separate batches may be created for a family based on optimistic information about the jobs that will ultimately occur between those batches. Later on, it might be desirable to combine some of those batches. We might guess that performance could be improved by adding a step that combines batches. Conversely, the GT sequence tends to schedule too few batches, because it does not split families at all. We might guess that performance could be improved by adding a step that splits batches. Splitting and combining are not difficult to implement, once a complete schedule is available.

First we describe a neighborhood search approach to combining batches of the same family. The idea is that combining two batches into one will save a setup, and, if the saved setup occurs early enough in the schedule, we can reduce the maximum lateness. Specifically, we want to accelerate the batch in which the maximum lateness occurs, a batch we refer to as the *critical* batch, denoted batch c. (If there are ties, c denotes the latest batch that achieves $L_{\max}$.)

For any complete schedule, called a *seed*, we define a *C-neighborhood* as the set of schedules that can be obtained from the seed by choosing a batch and combining it with the next earlier batch of the same family. In Figure 2, suppose that we identify batch k, and suppose that batch i is the next earlier batch in the same family as k. As the figure shows, the hypothetical change is to combine batch k with i to determine whether $L_{\max}$ improves. If necessary, the combining step may involve some resequencing to conform to the EDD rule for batches. If there is an improvement, we proceed with the improved schedule as a seed.

We start this process with the critical batch (i.e., with $k = c$ in Fig. 2), looking for a previous batch of the same family. If we cannot find an improvement, we shift k to the batch that directly precedes the critical batch and look for a previous batch of the corresponding family. Eventually, we will find no combinations that improve the original schedule, or else we will locate a desirable combination. In that case, we implement the combination, thus generating a new seed. We can then determine the new critical batch and repeat the C-neighborhood search until no further improvement can be made.

To gain some efficiency, we need consider only a limited portion of the C-neighborhood. In particular, in order to make an improvement, we must be able to accelerate batch c. Thus, there

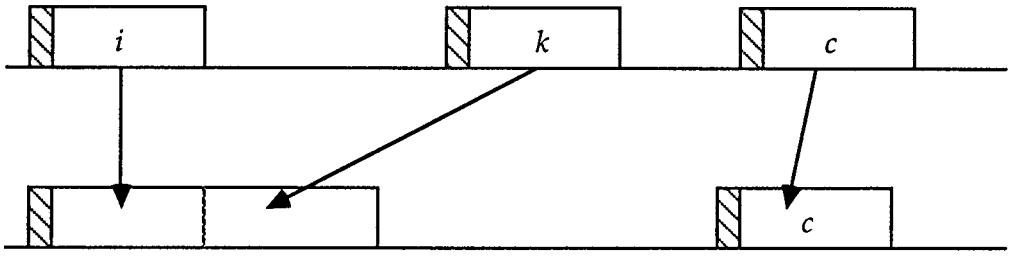

Figure 2. Combining batches i and k.

is no advantage in trying to carry out the combining step with a batch k that occurs after batch c. In addition, once we specify batch k, we need only try combining it with its next earlier batch within the family. If that does not work, there is no advantage to combining batch k with an even earlier batch.

The opposite strategy involves splitting an existing batch into two smaller batches. In this case, we define an *S-neighborhood* as the set of schedules that can be obtained from a seed by splitting off the last job from a batch and inserting it later in the schedule, possibly as a separate batch. If necessary, the splitting step may require resequencing according to the EDD rule for batches. After the split, the first part of the split batch (i') will precede batch c, but the second part (j) must follow it (see Fig. 3). For convenience, we take j to be a single job. Batches that follow j will be delayed by the length of a setup time.

Again, we start with an initial seed and examine its S-neighborhood to see whether a schedule with an improved value of $L_{\max}$ can be found there. If not, we stop; but if an improvement exists, then the improved schedule becomes the seed, and we invoke the neighborhood search again.

To gain efficiency, we need examine only a portion of the S-neighborhood. Consider where in the schedule job j should appear, assuming it remains a separate batch. It cannot appear prior to any batch with a lateness at least as large as $L_c - s$ if we are to achieve an improvement. This result merely reflects that fact that splitting will delay batches after job j by a setup time. Therefore, we first identify a set B of consecutive batches at the end of the schedule with the property $L_b < L_c - s$. (This set may be empty.) As shown in Figure 3, let a denote the last batch in the original schedule prior to the set B, and let C_a denote its original completion time. By construction, the lateness of batches in B will be lower than L_c after the splitting is done. The value of $L_{\max}$ for the new schedule will be lower only if the lateness of j is lower than L_c after the split. In other words, we require that

$$C_a + s - d_j < L_c$$

or

$$d_j > C_a + s - L_c.$$

Thus, we can implement the splitting procedure by identifying batch c, constructing set B, computing C_a, and then searching for a qualifying job j among the batches preceding batch c.

In the foregoing, we assumed that splitting job j would result in a new batch and an incremental setup. However, we can first look for the possibility of moving job j into a later batch k of the same family. In this case job j is the only job made later by the split, and for an improvement, we

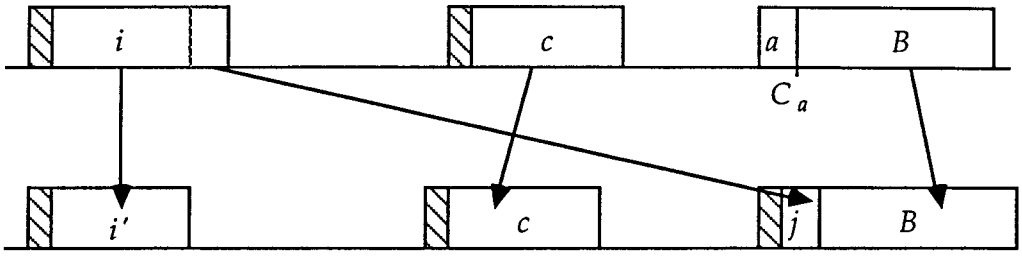

Figure 3. Splitting the last job from batch i.

require

$$T - d_j < L_c,$$

where T denotes the start time of processing for batch k in the original schedule. Thus, we need consider only those batches that start processing at

$$T < d_j + L_c.$$

If we find such a batch, then the split will lead to an improvement. If no such batch is available, we resort to the more general case above, in which batch j remains separate.

The C-neighborhood search and the S-neighborhood search can be initiated with any sequence. However, there are certain combinations that appear fruitful. Since the GT sequence may have too few batches, it is logical to use it as a seed for the S-neighborhood search, and since the Gap Heuristic tends to have too many setups, it is logical to use the schedule it produces as a seed for the C-neighborhood search. We refer to GT/S or to Gap/C to indicate both the nature of the initial seed and the choice of neighborhood.

Finally, some iteration between the two types of neighborhood searches is conceivable. In particular, Gap/CS (in which the C-neighborhood search is followed by the S-neighborhood search) represents a plausible heuristic scheme, but following it with another C-neighborhood search is hardly ever worthwhile. Similarly, GT/SC (in which the S-neighborhood search is followed by the C-neighborhood search) makes a marginal improvement over GT/S.

6. COMPUTATIONAL TESTS

A computational study is appropriate in order to provide some perspective on the effectiveness of the various heuristic procedures. An experimental approach of this sort relies on two elements: a set of test problems and a measure of effectiveness. (See Hall and Posner [4] for an in-depth discussion of generating experimental data.) As we shall see, it sometimes makes sense to think about performance measures before generating test data.

6.1. Measure of Effectiveness

Perhaps the most intuitive choice of a performance measure for comparing heuristic procedures is the ratio

$$V_0 = L/L^*,$$

where L denotes the value of $L_{\max}$ produced by a given heuristic, L^* denotes the optimal value, and V_0 measures the performance of the heuristic. In an experimental approach, values of V_0 would be averaged over a set of randomly generated test problems in order to summarize the effectiveness of a particular procedure. Unfortunately, there is an implicit premise in this choice: a notion that average values of V_0 near one indicate good performance. However, it is possible to have L^* turn out negative, or even zero. In such cases the value of V_0 tells us more about the optimal value of $L_{\max}$ than about the suboptimality in the heuristic procedure.

One way to alter the summary measure is to use a difference between heuristic and optimal values in the numerator:

$$V_1 = (L - L^*)/L^*.$$

Although the numerator will always be nonnegative, V_1 still approaches infinity when the optimal solution is near zero. In such cases V_1 still tells us little about suboptimality. We could force the

denominator to be nonzero by shifting the due dates sufficiently, but there is another problem. A shift in all the due dates by a constant leaves the numerator unchanged but alters the denominator. Thus, even for a problem instance where L^* is positive, V_1 decreases when all the due dates are shifted upward, even though the shift does not change the essential problem to be solved. In an attempt to avoid this type of instability, we used the following alternative:

$$V = (L - L^*)/p,$$

where p denotes the average processing time in a schedule. The metric V measures the amount of suboptimality in proportion to the average processing time. The value $V = 0$ occurs when a heuristic produces an optimal solution, and, obviously, small values of V are desirable. The value $V = 1$ may also represent a useful target, indicating that no job is later than the average run time. Compared to V_1 this measure may be a little less intuitive, but V is not unstable for L^* values near zero, nor does it change when all due dates are shifted by a constant.

6. 2. Test Problems

In selecting test problems, one important goal is to create problem instances that are representative of the general problem class. Hall and Posner [4] refer to this property as *correspondence*. The trap to avoid is the inadvertent creation of ''easy'' problem instances that do not really require the ingenuity of a solution procedure. Fortunately, the existing theory for the $L_{\max}$ problem provides some guidance. In particular, we know that when setup times are very large or very small, or when due dates are tightly clustered, an optimum will be produced by either the EDD sequence or the GT sequence. For problem instances to be ''interesting'' they must lie, in some sense, between these extremes.

The processing times in the test problems were randomly sampled from a uniform distribution on the integers from 1 to 99. This convention is commonly used in computational experiments for scheduling and other combinatorial problems. Next, the test problems required the systematic selection of due dates and setup times.

If we add a constant to all the due dates in a given problem, then there will be no difference in the sequences produced by the various procedures, but the value of $L_{\max}$ will change by an amount equal to the constant. In other words, it is not the absolute value of the due dates that is of intrinsic interest, but only their relative distances from each other. For the purposes of the experiments, therefore, due dates were uniform samples drawn from the integers between zero and rnp. After the due dates in a given problem were drawn, $d_{\min}$ was subtracted from each, thus anchoring the minimum due date in each test problem at zero and assuring that the optimal value of $L_{\max}$ would be positive. (This translation achieves a degree of what Hall and Posner [4] refer to as *parsimony* in the generated data.)

After obtaining the processing times and due dates, the next step was to generate a setup time, using the setup factor and the parameter S^*, which is calculated from the processing times and due dates. The setup time was calculated as $s = uS^*$.

The choice of parameters u and r was dictated by a desire to generate problem instances that are unlikely to be easy in the sense described earlier. The specific combinations chosen are shown in Figure 1: 10 pairs (u, r) lying between the regions where the EDD and GT sequences are known to produce optima.

The computational experiments included test problems of various sizes, and for each parametric choice of u and r, 20 test problems were created for each problem size. Specifically, in the first phase, the problem sizes were 2×12, 3×8, 4×6, and 5×5, where the size is given as $f \times n_i$.

Thus, the testbed covered 40 different experimental conditions, with 20 replications at each, for a total of 800 ''smaller'' test problems. In the second phase, the problem sizes were 2×18, 3×12, 4×9, and 6×6, accounting for another 800 ''larger'' test problems. Optimal solutions were obtained using the branch-and-bound algorithm described by Baker and Magazine [1]).

7. EXPERIMENTAL RESULTS

Some representative observations are shown in Table 1. At each (u, r) combination that was tested, the table shows two triplets. The top triplet corresponds to the GT solution and its variants (GT/S and GT/SC), and the bottom triplet corresponds to the Gap Heuristic and its variants (Gap/C and Gap/CS), as noted in the table. Each triplet shows the number of optimal solutions (out of 20 replications) produced by the three variations: the basic version, one neighborhood search, and two complementary neighborhood searches, in that order. The tabulated data are drawn from the 3×8 problem sets.

Two patterns are suggested by the data in Table 1. First, the heuristic procedures are somewhat complementary, in that Gap/CS tends to perform well when u-values are high, while GT/SC tends to perform well when r-values are low. Second, we obtain diminishing returns from invoking neighborhood searches, although the first use is effective.

The EDD sequence produced no optima in this region. Evidently, its performance degrades rapidly as the setup factor increases from zero, so that, even by $u = 0.10$, there were no optimal solutions encountered. Overall, the GT sequence was optimal in 13% of the 800 smaller test problems, and the Gap Heuristic was optimal in 29%. The other variations produced optima more frequently: 47% in the case of Gap/CS and 37% in the case of GT/SC.

Another summary of the outcomes is shown in Table 2, where the summaries cover all 800 smaller test problems. Looking at the average values of V, we see values near 1.0 for both Gap/CS and GT/SC. The last column shows the summary for a hybrid solution procedure that consists of constructing both the Gap/CS solution and the GT/SC solution, and choosing the better of the two. This procedure was able to produce an optimal solution about 56% of the time and achieved an average value of V equal to about 0.58. In other words, the maximum lateness is less than the mean processing time, on average.

Table 3 provides the results for the larger test problems, which tend to be more difficult to solve than the smaller problems of Table 2. Nevertheless, very similar patterns can be observed. Again, the EDD sequence produced no optima, while the GT sequence was optimal in 9% of the larger

Table 1. Number of times optimal: a comparison of the GT sequence (top) and the Gap Heuristic (bottom), in 20 test problems for each combination (u, r).

r-Value	0.10	0.25	0.50	0.75	
4.0			0 1 1	6 11 11	[GT]
			8 8 8	14 14 14	[GAP]
2.0		0 0 0	0 4 4	8 13 13	
		4 4 4	5 5 6	11 13 14	
1.0	0 4 4	0 1 1	2 8 8		
	0 2 2	3 6 7	7 7 11		
0.5	0 12 13	5 14 14			
	0 8 8	1 7 11			
		u-Value			

Table 2. A comparison of the heuristic procedures on the smaller test problems, showing the percentage of problems in which optima were generated, along with the average V.

Measure	Size	EDD	GT	Gap/CS	GT/SC	Hybrid
Percent optima	(2×12)	0	8	58	33	66
Percent optima	(3×8)	0	10	42	34	50
Percent optima	(4×6)	0	12	39	39	48
Percent optima	(5×5)	0	20	48	42	59
Average V	(2×12)	237	3.86	0.82	1.88	0.60
Average V	(3×8)	305	3.99	1.15	1.73	0.90
Average V	(4×6)	344	3.84	1.26	1.41	0.92
Average V	(5×5)	393	2.96	0.78	0.87	0.52

problems and the Gap Heuristic was optimal in 22%. The two main heuristics produced optimal solutions more frequently: 33% in the case of Gap/CS and 21% in the case of GT/SC. Here, the Hybrid procedure produced the optimal solution about 38% of the time, with an average value of V equal to about 1.08. This means that the maximum lateness was nearly equal to the mean processing time, on average.

Our first concern is identifying an effective heuristic procedure. Overall, Gap/CS produced the best average performance for each problem size. The GT/SC procedure is not quite as effective on average, but on individual problems it is sometimes the best procedure. The Hybrid procedure is an effective way to harness the effectiveness of each. Recalling Figure 1, and the fact that the test data were consciously structured to avoid easy cases, these heuristic procedures seem to provide good solutions quickly. (Computation times are negligible on a desktop computer using a Pentium 200 MHz processor.)

A secondary consideration is the identification of parameters that make it difficult to produce optimal solutions with the heuristic methods. Tables 4 and 5 provide a breakdown of the average performance achieved by the Hybrid procedure according to due-date range and setup factor. To the extent that these results indicate problem difficulty, it appears that the optimal solution is most difficult to reach when the due date range is between 1.0 and 2.0 and when the setup factor is between 0.25 and 0.50. Average performance appears to improve on either side of those values.

Table 3. A comparison of the heuristic procedures on the larger test problems, showing the percentage of problems in which optima were generated, along with the average V.

Measure	Size	EDD	GT	Gap/CS	GT/SC	Hybrid
Percent optima	(2×18)	0	8	39	16	46
Percent optima	(3×12)	0	4	32	13	35
Percent optima	(4×9)	0	7	25	20	30
Percent optima	(6×6)	0	19	36	32	41
Average V	(2×18)	397	4.66	1.30	2.89	1.20
Average V	(3×12)	548	4.92	1.58	2.27	1.33
Average V	(4×9)	624	4.28	1.40	1.61	1.03
Average V	(6×6)	638	3.29	1.04	1.27	0.77

Table 4. Summary of average performance on the smaller problems for the Hybrid procedure, for each parametric case in the study.

Setup factor (u)	0.10	0.25	0.50	0.75
Average V	0.36	0.71	0.76	0.31
Due date range (r)	0.5	1.0	2.0	4.0
Average V	0.15	0.64	0.78	0.60

8. CONCLUSIONS

In this research, we have examined a basic scheduling model in which there is a trade-off between efficiency and due-date performance, and we have explored the effectiveness of various heuristic approaches for constructing schedules. Each of these approaches exploits some known theoretical property of the optimal solution.

The EDD sequence simply arranges the jobs from earliest to latest due date, which would be the optimal sequence if there were no setup times. Its computational requirement is $O(n \log n)$, since it requires the sorting of n jobs. Although it is likely to be optimal when setup times are close to zero, its performance degrades rapidly as setup times increase, and over the interval of parametric cases that was tested, its average performance was rather poor.

The GT sequence places the jobs in batches, one batch per family, and then sequences the batches from earliest to latest batch due date. Its computational requirement is no worse than $O(n \log n)$, since it requires the sorting of jobs within batches and then the sorting of batches. Sequencing batches by their batch due dates is known to be optimal, so the only source of suboptimality in the GT sequence is the restriction of one batch per family.

The Gap Heuristic is a newly developed procedure that exploits a splitting condition while adding jobs one at a time to a schedule. Its computational requirement is $O(n^2 \log n)$ since each iteration inserts one job into the schedule, and there could be a resorting of the batches with each insertion so that their batch due dates are in order. Nevertheless, the procedure may make suboptimal batching decisions early on, when information is available about only part of the job set.

Neighborhood search routines can improve heuristic performance. The GT sequence can be improved by allowing batches to be split, and the result can be improved a little more by allowing batches to be combined. The Gap Heuristic can be improved by combining and then splitting batches. The best overall results from a single algorithm were achieved by using the Gap Heuristic as an initial seed. However, the best results were obtained by running both GT/SC and Gap/CS and then choosing the better of the two schedules. As with any neighborhood search regime, there is no way to know how many improvements might be made, and so the computational complexity of the procedure is unknown.

The computational study gives us some insight into the parametric cases of the problem that are easy or hard to solve. As indicated by Properties 1, 4, and 5, some special cases lead to

Table 5. Summary of average performance on the larger problems for the Hybrid procedure, for each parametric case in the study.

Setup factor (u)	0.10	0.25	0.50	0.75
Average V	0.64	1.32	1.58	0.42
Due date range (r)	0.5	1.0	2.0	4.0
Average V	0.30	1.25	1.46	1.06

straightforward optimal solutions: when setup times are zero, when setup times are larger than a problem-dependent critical value, or when due dates are identical. Problems that nearly satisfy one of these criteria also tend to be solved effectively by the heuristics. Tables 4 and 5 suggests which parametric cases may be hardest to solve in general.

The pursuit of good due-date performance in the presence of family setup times is a challenging scheduling problem, although it represents a reasonably common one in batch manufacturing. Not only are simplified static versions of this problem difficult to solve, but it is also hard to glean insights from simple cases that might be helpful in a situation where jobs arrive over time and where scheduling decisions are made dynamically. One useful insight involves the EDD priority rule as a sorting mechanisms. Although it can be applied at the level of jobs (resulting in the EDD sequence) and at the level of families (resulting in the GT sequence), it seems best to focus on applying EDD at the level of batches. What remains is to find a good mechanism for allocating jobs to batches. The Gap Heuristic provides one guidepost in this context, in that it represents a systematic basis for deciding when to allow jobs of a given family to be split among two or more batches. A second insight comes from the effectiveness of an S-neighborhood search. This observation suggests that it might be helpful, in dynamic settings, to reevaluate the last portion of a batch as conditions change, to determine whether it should be separated and postponed. Hopefully, these types of insights can be adapted to develop good dispatching rules for more complicated situations involving batch scheduling.

ACKNOWLEDGMENTS

Acknowledgement is due to Roger Trussell, Anne Poirer, and John Harris for coding assistance. This paper reflects helpful comments on an earlier draft made by Marc Posner, Gerald Campbell, and Scott Webster.

REFERENCES

[1] K.R. Baker and M.J. Magazine, Minimizing maximum lateness with job families, Working Paper 314, Amos Tuck School, Dartmouth College, 1998.

[2] J. Bruno and P. Downey, Complexity of task sequencing with deadlines, set-up times and changeover costs, SIAM J Comput 7, (1978), 393–404.

[3] J.B. Ghosh and J.N.D. Gupta, Batch scheduling to minimize maximum lateness, Oper Res Lett 21, (1997), 77–80.

[4] N.G. Hall and M.E. Posner, Generating experimental data for scheduling problems, manuscript, Ohio State University, 1996.

[5] A.M.A. Hariri and C.N. Potts, Sincle machine scheduling with batch setup times to minimize maximum lateness, Ann Oper Res 70, (1997), 75–92.

[6] J.R. Jackson, Scheduling a production line to minimize maximum tardiness, Research Report 43, Management Science Research Project, UCLA, Los Angeles, 1955.

[7] A.J. Mason and E.J. Anderson, Minimizing flow time on a single machine with job classes and setup times, Nav Res Logistics 38, (1991), 333–350.

[8] C.L. Monma and C.N. Potts, On the complexity of scheduling with batch set-up times, Oper Res 37, (1989), 798–804.

[9] C.N. Potts and L.N. Van Wassenhove, Integrating scheduling with batching and lot-sizing: A review of algorithms and complexity, J Oper Res Soc 43, (1992), 395–406.

[10] J.M.J. Schutten, S.L. van de Velde, and W.H.M. Zijm, Single-machine scheduling with release dates, due dates and family setup times, Manage Sci 42, (1996), 1165–1174.

[11] S.W. Webster and K.R. Baker, Scheduling groups of jobs on a single machine, Oper Res 43, (1995), 692–703.

[12] D. Williams and A. Wirth, A new heuristic for a single machine scheduling problem with set-up times, J Oper Res Soc 47, (1996), 175–180.