International Journal of Production Research

ISSN: 0020-7543 (Print) 1366-588X (Online) Journal homepage: https://www.tandfonline.com/loi/tprs20

Minimizing total weighted tardiness on a batch-processing machine with incompatible job families and job ready times

Mary E. Kurz & Scott J. Mason

To cite this article: Mary E. Kurz & Scott J. Mason (2008) Minimizing total weighted tardiness on a batch-processing machine with incompatible job families and job ready times, International Journal of Production Research, 46:1, 131-151, DOI: 10.1080/00207540600786665

To link to this article: https://doi.org/10.1080/00207540600786665

International Journal of Production Research,
Vol. 46, No. 1, 1 January 2008, 131–151

Taylor & Francis
Taylor & Francis Group

Minimizing total weighted tardiness on a batch-processing machine with incompatible job families and job ready times

MARY E. KURZ*† and SCOTT J. MASON‡

†Department of Industrial Engineering, Clemson University, 110 Freeman Hall, Clemson, SC 29634-0920, USA
‡Department of Industrial Engineering, University of Arkansas, 4207 Bell Engineering Center, Fayetteville, AR 72701, USA

(*Revision received December 2005*)

Semiconductor wafer fabrication facilities (''wafer fabs'') strive to maximize on-time delivery performance for customer orders. Effectively scheduling jobs on critical or bottleneck equipment in the wafer fab can promote on-time deliveries. One type of critical fab equipment is a diffusion oven which processes multiple wafer lots (jobs) simultaneously in batches. We present a new polynomial time Batch Improvement Algorithm for scheduling a batch-processing machine to maximize on-time delivery performance (minimize total weighted tardiness) when job arrivals are dynamic. The proposed algorithm's performance is compared to previous research efforts under varying problem conditions. Experimental studies demonstrate the effectiveness of the Batch Improvement Algorithm.

Keywords: Heuristics; Batch-processing; Scheduling; Semiconductor manufacturing

1. Introduction

The integrated circuits (ICs) at the heart of today's technology age require a considerable amount of time and money to manufacture. These ICs are manufactured in ultra clean wafer fabrication facilities (''wafer fabs''). Wafer fabs often contain 70+ different types of processing tools, each of which can cost anywhere from U. S. $50,000 to $10,000,000. In order to mitigate the effects of both planned and unplanned tool downtime, multiple tools of each type are operated in parallel in the wafer fab (''parallel machines''). In total, current generation 200-mm wafer fabs usually require capital investments totaling US $2 billion in order to become a reality. After making this size of an investment in manufacturing capacity, IC manufacturers clearly strive to sell large volumes of their products. However, many companies compete in the same product space, affording customers the

*Corresponding author. Email: mkurz@clemson.edu

International Journal of Production Research
ISSN 0020–7543 print/ISSN 1366–588X online © 2008 Taylor & Francis
http://www.tandf.co.uk/journals
DOI: 10.1080/00207540600786665

opportunity to buy elsewhere if a particular manufacturer cannot meet its quoted product shipment or due dates. One way to measure a company's delivery performance is to calculate the total weighted tardiness $\mathrm{TWT} = \sum w_j T_j$.

A considerable amount of research has been conducted on efficiently scheduling jobs on production machines to maximize on-time delivery. Even though appearing to be quite simple, some single machine scheduling problems cannot be solved to optimality within any reasonable amount of time. For example, the single machine scheduling problem in which the maximum job lateness is to be minimized, commonly denoted as $1\|L_{\max}$ in the notation of Lawler *et al*. (1982), is easily solved. However, simply adding job release times (i.e., not all jobs are ready for processing at time $t=0$) results in the $1|r_j|L_{\max}$ problem. This problem is in the class NP-hard, as it is believed that no method exists to obtain the optimal solution in any practical amount of time (Lenstra *et al*. 1977). The ability of some tools to process multiple production jobs simultaneously in batches, such as diffusion ovens in semiconductor manufacturing, further complicates the already NP-hard scheduling problem. This further complication results from the large number of potential batches that must be evaluated for scheduling.

While recognizing the importance of the parallel machine environment, we have focused our initial research efforts on the single batch-processing machine environment, where each production job belongs to one of several product families. Several jobs in the same family may be processed in the same batch, while jobs from different families may not, reflecting the existence of incompatible families. This is typical of processing steps late in the semiconductor manufacturing process flow, such as metal deposition steps, that are visited by wafers containing differentiable product types. Further, each job has its own unique priority or weight, release time, and due date. As our objective is to maximize the wafer fab's delivery performance of customer orders, the scheduling problem under consideration can be denoted as $1|r_j, p-batch, incompatible|\sum w_j T_j$.

As the optimal solution to this or any other NP-hard scheduling problem cannot be obtained directly in a reasonable amount of time, researchers typically employ heuristic approaches to obtain "good" solutions to these problems. In this paper, we present an adaptation of Mehta and Uzsoy's (1998) batch apparent tardiness cost (BATC) algorithm, an adaptation of Kanet and Li's (2004) WMDD algorithm, and a new Batch Improvement Algorithm (BIA). Through extensive experimental test cases, we demonstrate the viability of the BIA for minimizing TWT for several families of jobs being processed on a single batch-processing machine under dynamic job arrivals. We distinguish between *static* job arrivals, in which all jobs are available for processing at time $t=0$, and *dynamic* job arrivals, in which jobs arrive at times distributed through time. Note that in both cases, job arrival times are known *a priori*, distinguishing the static and dynamic cases from the *on-line* case.

The remaining sections of this paper are organized as follows. We discuss the batch machine scheduling problem in section 2, providing a review of previous research efforts that focus on this problem. Next, section 3 presents the notation used in the paper, while section 4 reviews the BATC algorithm of Mehta and Uzsoy (1998) and the WMDD algorithm of Kanet and Li (2004), and presents several extensions of each. Section 5 contains the proposed Batch Improvement algorithm (BIA). Section 6 presents the experimental plan and results. Finally, we present our research conclusions in section 7, along with our directions for future research.

2. Previous Research

The scheduling of batch-processing machines is a complex and difficult problem. A batch-processing machine may process from one up to B jobs at a time, where B is the maximum batch size allowed by the machine. Given n jobs to be scheduled, Chandru *et al.* (1993) point out that at most n batches and at least $[n/B]$ batches will be required to process all jobs. The number of potential batches of size B that can be formed from n jobs is $\binom{n}{B}$, the number of potential batches of size $B-1$ that can be formed from n jobs is $\binom{n}{B-1}$, and so on (Mason *et al.* 2002). We refer the reader to Brucker *et al.* (1998) for a comprehensive discussion of the complexity of scheduling a single parallel batch-processing machine under regular scheduling criteria, noting that neither non-zero ready times nor incompatible families are considered. Given the complexity of this problem, considerable research effort has been dedicated to the batch-processing scheduling problem.

Early work involving batch-processing machines focused on the single machine problem. Coffman *et al.* (1990) discovered that to achieve an optimal schedule, with regard to minimized overall flow time, the processing times of the batched jobs should be non-decreasing throughout the schedule. Motivated by the long processing times of the burn-in operations of semiconductor testing, Chandru *et al.* (1993) developed the "full batch shortest processing time" (FBSPT) and "greedy ratio" (GR) heuristics to solve the single machine and parallel machines batching problems where all jobs are present at initial scheduling (i.e., all jobs have the same ready times). Both heuristics function based on the jobs being previously ordered by least processing time to greatest processing time.

In both cases full batches are formed and the machine is kept busy as long as any jobs remain unprocessed. However, FBSPT does not account for the case when widely varying processing times exist between two or more jobs. To account for this, the GR heuristic tests the ratio of the processing time of the last job in a potential batch divided by the number of jobs in that batch. The minimum ratio is selected and those jobs are batched together while all other jobs wait for a later batch. They determined that the GR heuristic performed better in minimizing the total completion time for both the single machine and parallel machine cases. Therefore, if two jobs have processing times which are very different, these are not likely to be batched together.

Uzsoy (1995) extended his previous research to a single batching machine where jobs from different job families cannot be batched together. He presents several heuristics to address these types of problems aimed at minimizing maximum lateness and maximum completion time. Hochbaum and Landy (1997) investigated both the single and parallel machine cases and developed the Fixed Sequence heuristic for the single machine case, which was proven to have a worst case performance of two times the optimal schedule. Lee and Uzsoy (1999) discussed minimizing makespan on a single, batch-processing machine with dynamic job arrivals, solving this problem with several additional heuristics.

Sung and Choung (2000) discuss minimizing makespan under the dynamic problem of different release times. Potts *et al.* (2001) look at two-machine batching problems in an open shop, job shop, and flow shop, showing that the complexity

involved in solving these scenarios is typically NP-hard. Van der Zee *et al.* (2001) investigate batching situations which arise in the aircraft industry and developed a forward looking strategy which looked at all machines in the near future, not simply idle machines. Perez *et al.* (2005) address $1|r_j, p-batch,incompatible|\sum w_jT_j$ by first forming batches and then scheduling those batches on the single machine. Various dispatching rules are used for both steps. Mönch *et al.* (2005) consider $P_m|r_j, p-batch,incompatible|\sum w_jT_j$ and address $|r_j, p-batch,incompatible|\sum w_jT_j$ as a subproblem by forming batches and selecting batches according to a priority index which considers both the potential weighted tardiness of the current batch and the jobs in the unscheduled batches. In both Perez *et al.* (2005) and Mönch *et al.* (2005), the single machine schedule is created using a constructive rather than an improvement algorithm.

The literature to date has addressed batch scheduling problems of both the single and multiple machine cases, but has indicated few instances when an optimal schedule is obtainable in a reasonable computational time. This is especially true in the multiple machine case. Therefore, research continues to seek faster running algorithms delivering better solutions to solve these problems. In most of the research, prior to batch formation, all jobs are ordered from shortest to longest processing times. This typically is done to minimize the total completion time metric. In more recent research, the goal has shifted from minimizing total completion time to minimizing functions based around meeting due dates of jobs. Specifically, Brucker *et al.* (1998) show that $1|p-batch|\sum w_jT_j$ is NP-hard when $2 \leq B < n$. In short, because of the computational complexity, no improvement method currently exists for single or multiple server batch-processing machines in an acceptable time frame.

3. Notation

Our notation closely follows that of Mehta and Uzsoy (1998). There are n jobs belonging to m families, with each family j having n_j jobs, $j=1,\ldots,m$. Job i in family j ("job ij") has ready time r_{ij}, due date d_{ij}, and weight w_{ij}, $i=1,\ldots,n_i$ All jobs in family j require P_j units of processing time, $j=1,\ldots,m$. We assume that all jobs in a family are indexed in non-decreasing order of ready time and non-decreasing order of weighted due date for jobs with the same ready times, so that $r_{ij} \leq r_{i+1},j$ and $d_{ij}/w_{ij} \leq d_{i+1}/w_{i+1,j}$ $\forall j$, $1 \leq i < n_j$.

Jobs are processed on machines in parallel batches, meaning the jobs are processed simultaneously. Let B denote the maximum batch size of each machine and B_{kj} represent the k-th formed batch of family j. Let $R_{kj} = \max_{ij \in B_{kj}}\{r_{ij}\}$ be the ready time of batch B_{kj} and C_{kj} represent the completion time of B_{kj}. No job or batch may begin processing before its associated ready time. All jobs ij in batch B_{kj} share the same completion time C_{kj}. If job $ij \in B_{kj}$ and $C_{ki} > d_{ij}$, job ij is *tardy*, and accrues a penalty of $w_{ij}(C_{ki}-d_{ij})$ units. If job ij is not tardy, no penalty is accrued. Penalty values are summed over all n jobs to compute the total weighted tardiness of the resulting schedule.

4. BATC and WMDD based Priority Indexes

In this section we discuss two priority indexes from the literature. The BATC based priority indexes are based in work by Mehta and Uzsoy (1998), who considered the single batch-processing machine problem to minimize total tardiness. The WMDD based priority indexes are based in work by Kanet and Li (2004) who considered the single unit-processing machine problem to minimize total weighted tardiness. In this work, we adapt the BATC index to account for weights and non-zero ready times and the WMDD index to account for batch processing and non-zero ready times.

Mehta and Uzsoy (1998) developed the batch apparent tardiness cost (BATC) algorithm, adapting the apparent tardiness cost (ATC) heuristic of Vepsalainen and Morton (1987). Mehta and Uzsoy proved that for the $1|p\text{-batch, incompatible}|\sum T_i$ problem, there exists an optimal solution which will contain full batches, except for possibly the last batch in each family, and there exists an optimal solution in which the jobs in each batch are consecutively indexed, when indexed in non-decreasing order of their due dates. They use these two properties in order to define a greedy batch formation method. In order to refer to it later, we define it here with some modifications to suit our purposes which do not change the original results. Because the Greedy Batch Formation algorithm can be completed in $O(n \log n)$ steps, Algorithm PI's complexity is $O(n \log n)$.

Greedy batch formation

(1) Index all jobs in each family in non-decreasing order of ready time and non-decreasing order of weighted due date for jobs with the same ready times, so that $r_{ij} \leq r_{i+1,j}$ and $d_{ij}/w_{ij} \leq d_{i+1,j}/w_{i+1,j}$ $\forall j$, $1 \leq i < n_j$.
(2) For each family j, form batches so that jobs $1j$ to Bj are in batch B_{1j}, jobs $B+1, j$ to $2B, j$ are in batch B_{2j} and so on, until the last batch of each family may have less than B jobs. Let $k_j = [n_j/B]$ be the number of batches of family j formed and $K = \sum_{j=1}^{m} k_j$.

We also provide the shell of the BATC algorithm as Algorithm PI, where we replace the priority index BATC(B_{lj}) with the notation $PI(B_{lj})$.

Algorithm PI

(1) For each family j, form batches using the Greedy Batch Formation Algorithm.
(2) Let $i = 0$ be the number of batches in the partial schedule.
(3) Let $S(i)$ be the set of scheduled batches in the current partial schedule. Initialize $S(0) = \emptyset$. Let t be the latest completion time of any batch scheduled; $t = \max_{B_{lj} \in S(i)}\{C_{lj}\}$. Initialize $t = 0$.
(4) Let $U(i)$ be the set of unscheduled batches. (Note that this does not include the concept of "eligible" batches). For each batch B_{lj} in $U(i)$, calculate a

priority index $PI(B_{lj})$. Let $st = \arg\max_{B_{lj} \in U(i)}\{PI(B_{lj})\}$. Schedule B_{st} on the single machine. Let $i = i + 1$; update $S(i)$, $U(i)$ and t.

(5) If $i = K$, stop. Otherwise, go to step 4.

Algorithm DWBATC

The full problem of interest includes both jobs with weights and non-zero ready times. In order to accommodate weights in the priority index, we must remember that the weights may be different for each job in each family. One approach could be to create a composite batch weight that applies to each job in the batch. However, we have chosen to incorporate the weights in the part of the priority index that considers each job individually. If a batch has one job, we see that the priority index increases as the job's weight increases, reflecting our desire to schedule the batches with more important jobs earlier.

When a problem has ready times, we want to discourage scheduling batches that are not ready for processing. DWBATC will result in a lower priority index when the batch's ready time is higher, reflecting our desire to postpone batches for which we will have machine idle time. However, we still consider all batches in $U(i)$ as opposed to making some batches ineligible.

Based on this discussion, we introduce the DWBATC priority index, as shown in (1).

$$PI(B_{lj}) = DWBATC(B_{lj}, t) = \frac{1}{p_j}\exp\left(\frac{-\sum_{ij \in B_{lj}} \max\{(d_{ij} - p_j - t)/(w_{ij}), 0\} - R_{lj}}{k\bar{p}}\right) \quad (1)$$

where k is a scaling parameter and $\bar{p}$ is the average processing time of the remaining batches (batches in $U(i) \backslash B_{lj}$). While equation (1) is somewhat similar to the BATCS computation in Mason *et al*. (2002), DWBATC performs its calculations at the individual job level, and then sums up these results in the negative exponential's numerator. This is different than BATCS, which performs its calculations at the aggregated batch level.

Our construction of DWBATC also lends itself very well to additional experimental evaluation of problem instances with static job arrivals and/or unit job weights. Clearly, when jobs are all available at time 0 and are equally weighted (i.e., $r_{ij} = 0$ and $w_{ij} = \forall ij$), DWBATC reduces to the BATC rule of Mehta and Uzsoy (1998). This reduction is obviously quite convenient (and in our opinion, required) for comparison purposes with the original BATC rule. When jobs are available at time zero but have non-identical weights, we will call the index "WBATC" while when jobs have non-zero ready times and have unit weights, we will call the index "DBATC". In this manner, we have four indexes available for further investigation.

Algorithm DBWMDD

Baker and Bertrand (1982) considered the single machine total tardiness scheduling problem, developing the MDD (modified due date) algorithm. The overall idea is to sequence the jobs in earliest due date order until such time as a job is late; at that point, its new due date is considered to be the earliest time by which it can

be completed. They calculated $MDD_i = \max\{t + p_i, d_i\}$ for each job i, where p_i is the processing time and d_i is the due date of job i. The next job to be scheduled has the smallest value of MDD_i; the time t is then updated to the completion time of the most recently scheduled job. Kanet and Li (2004) extended the MDD algorithm to attack the single machine total weighted tardiness scheduling problem, creating the WMDD index:

$$WMDD_i = \frac{1}{w_i}\max\{p_i, d_i - t\} \tag{2}$$

The WMDD index is used in the same manner as the MDD index in the MDD algorithm. The definition of $WMDD_i$ incorporates the observation that the index should reduce to the Weighted Shortest Processing Time rule (WSPT) when all unscheduled jobs are tardy (as may occur when t is larger) in order to minimize the total weighted tardiness.

In this research, we propose an extension of WMDD and MDD to the problem with parallel-batch scheduling and non-zero ready times. We propose the DBWMDD priority index in (3), which is used with Algorithm PI in the same manner as the DWBATC priority index. We use the negative, so that we can still select the batch with the largest index (as opposed to the smallest valued batch). In general, we aggregate the values of the WMDD index for each job in the batch, in order to accommodate the parallel-batch processing, and we incorporate the ready time into the priority index, in order to incorporate non-zero ready times.

$$PI(B_{lj}) = DBWMDD(B_{lj}, t) = -\sum_{ij \in B_{lj}} \left(\frac{\max\{p_j, d_{ij} - t\}}{w_{ij}} \right) - R_{lj} \tag{3}$$

Clearly, when all jobs are available at time 0 and batches are of size 1 (i.e., $r_{ij} = 0$ $\forall ij$ and $B = 1$), DBWMDD reduces to the WMDD rule of Kanet and Li (2004) When jobs are available at time zero but have non-identical weights, we will call the index "BWMDD" while when jobs have non-zero ready times and have unit weights, we will call the index "DB_MDD". In this manner, we have four more indexes available for further investigation. When all jobs are available at time 0 and batches are of size 1 (i.e., $r_{ij} = 0$ $\forall ij$ and $B = 1$), DBWMDD reduces to the WMDD rule of Kanet and Li (2004).

5. Batch Improvement Algorithm (BIA)

Algorithms PI's use of the Greedy Batch Formation Algorithm may not result in an optimal batch schedule in the cases that jobs have ready times. This was discussed in Mehta and Uzsoy (1998) and demonstrated by counterexample in Kurz (2003). Clearly, Algorithms DWBATC and DBWMDD, and their variants, require full batches which will unnecessarily and with potential negative consequence restrict the solution. One heuristic method by which $1|r_j, p - batch, incompatible| = \sum w_j T_j$ may be attacked could involve the creation of partial batches in a forward manner. We could consider the jobs in some order and ask whether the given job should be added to the current batch or be the first job in a new batch. However, another method could involve creating n batches of one job each and then considering when to

combine them. Because our objective is total weighted tardiness, we know that such a schedule will provide an upper bound on the objective function.

Kurz (2003) provided two simple conditions for the $P_m1|r_j, p-batch, p_j=p|\sum w_jT_j$ (one single family) problem that demonstrated when it was advantageous to move a job from a batch to the immediately previous batch, as long as that batch has room. Theorem 1 says that if job $1i$ is in the k-th batch for family 1, B_{lk}, and $r_{1i} > R_{1,k-1}$, then moving job $1i$ to $B_{1,k-1}$ will certainly not increase the total weighted tardiness. This means that the total weighted tardiness will not increase if a job that does not increase the batch's ready time is added to the batch. Theorem 2 says that if $1i$ is in B_{lk}, $r_{1i} > R_{1,k-1}$ and $C_{1,k-1} > r_{1i} + P_1 > R_{1,k-1} + P_1$, then moving job $1i$ to $B_{1,k-1}$ will certainly not increase the total weighted tardiness. This means that the total weighted tardiness will not increase if a job that does not increase the batch's *start* time is added to the batch. These two conditions are clearly true even when considering moving a job from B_{1k} to B_{1s} where $s<k$, meaning the new batch for the job may be any batch "before" the original batch. Moreover, while the problem of interest in this paper is concerned with jobs in multiple incompatible families, the same conditions hold true. These concepts motivate the development of the following improvement algorithm, BIA. BIA operates by trying to smartly move jobs from later batches to the current batch without increasing the starting time of the current batch.

Initially, batches are formed greedily by considering the jobs sorted in non-decreasing order of ready times, with ties being broken by non-decreasing order of weighted due date. This initial job sequencing is made without regard to job family designations. Batches of jobs in the same family are then formed by adding jobs one at a time according to the sorted list until either (1) the current batch is full (i.e., B same-family jobs have been grouped together) or (2) a different job family is encountered, thereby forcing the formation of a new batch, given our incompatible job families assumption. Assume that K batches have been formed and scheduled on a single machine. The batches will be indexed by their position in the initial sequence in this algorithm. Assume the k-th batch B_k starts its processing at time S_k and finishes at time C_k.

If batches are initially created as suggested here, we know that we will never add jobs to batch $K-1$ from batch K in the initial step. Even if batches $K-1$ and K were of the same family, they would only exist as two different batches in the initial solution if batch $K-1$ were full. Therefore we start with batch $K-2$, which will be appropriate even if there are more than two families.

In step 1, we determine whether the current batch needs any jobs added to it. If the current batch is the "0-th" batch, we have iterated through all the batched and can stop. If the current batch is full, it clearly does not need any more jobs. If the current batch is the last batch, there simply are no jobs after it which can be added to it. Therefore, if the last batch is empty, delete it. In all other cases, we will begin the search for jobs to add to the current batch.

Step 2 begins the search for jobs to add to the current batch by first determining to which family the additional jobs should belong. If the current batch has jobs already in it, it is clear in this case that the potential jobs to be added should belong to this family, as described in step 2.1. If however, the current batch is empty, we first need to determine if there are jobs in later batches which can be moved into the currently empty batch without increasing the starting time of the immediately

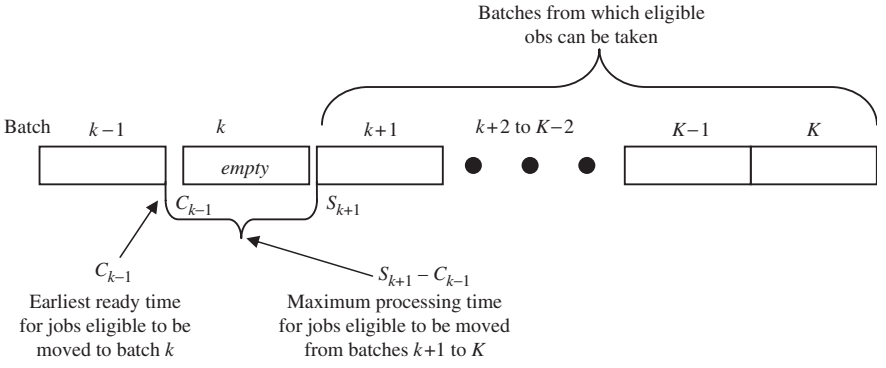

Figure 1. Finding jobs to add to the empty batch in BIA step 2.2

succeeding batch. In step 2.2, we form E, the set of jobs in later batches which are ready before the preceding batch completes and would finish before the succeeding batch starts. Figure 1 illustrates the process of determining the jobs in E. If E is empty, we move all later batches up by one and begin BIA on the previous batch. If E is not empty (step 2.2.2), we perform the sub-procedure *AddToCurrentBatch*(batch k, set E), which finds the job in E with the highest total weighted tardiness, adds it to the current batch, removes it from its previous batch and calls BIA from the moved job's previous batch. We will return to the current step upon completion of BIA. Once we complete step 2, we have a batch that has space for at least one job and we know the family to which the additional jobs should belong.

In step 3, we determine which jobs can be added to the current batch. This set formation is much simpler than the analogous set formation in step 2.2; the eligible jobs are those in later batches which have ready times no later than the current batches start time.

In step 4, we add jobs to the current batch from E, formed in step 3, as long as the current batch has room and E is not empty. Each batch which has jobs removed from it must also have BIA performed upon it. Once step 4 is complete, we consider the previous batch and begin BIA on that batch.

The complete algorithm is presented below. We first present the sub-procedure *AddToCurrentBatch*, which requires the current batch number and a set of jobs eligible to be added to it, and returns the batch number from which the added job was removed. A complete example is included in the appendix.

Sub-procedure sourcebatch = AddToCurrentBatch(batch k, set E)

$il^* = \arg\max_{ij \in E}\{w_{ij}T_{ij}\}$

Pick the job with the highest weighted tardiness and move it to B_k

Let f be the batch to which il^* belongs.
$B_k = B_k \cup il^*$
$E = E \backslash il^*$
$B_f = B_f \backslash il^*$
Return $sourcebatch = f$
(Step 0: Let $k = K - 2$.)

Algorithm BIA(k)

1. Update the batch ready, starting and completion times and job weighted tardiness values if needed.
   1.1 If $k=0$, STOP.
   1.2 If $|B_k|=B$, BIA $(k-1)$.
   1.3 If $k=K$ and $|B_k|=0$, $K=K-1$ and return.
   1.4 If $k=K$ and $|B_k|>0$, return.
   1.5 Else continue.
2. Let l be the family of the jobs in B_k.
   2.1. If $|B_k|\geq 0$, l is the family of any job in B_k.
   All jobs in B_k are in the same family
   2.2. If$|B_k|=0$, form $E=\{ij\colon r_{ij}\leq C_{k-1}, ij\in B_g, g>k, p_j\leq S_{k+1}-C_{k-1}\}$.
   These jobs will "fit" in the spot formerly filled by B_k
   2.2.1 If$|E|=0$ then B_k can be deleted, moving all later batches up
   for $g=k+1,\ldots,K$
   let $B_{g-1}=B_g$
   $K=K-1$
   BIA$(k-1)$
   2.2.2 Otherwise, $f=sourcebatch=AddToCurrentBatch$
   (batch k, set E) BIA(f) Fill the open spot in B_f
3. Form $E=\{il\colon r_{il}\leq S_k, il\in B_g, g>k\}$.
4. While $|B_k|<B$ and $|E|>0$
   $f=sourcebatch=AddToCurrentBatch$(batch k, set E)
   BIA(f) Fill the open spot in B_f
5. BIA$(k-1)$

6. Experimental plan

6.1 *Competing heuristic approaches*

The proposed Batch Improvement Algorithm (BIA) will be tested in a variety of single machine, batch processing environments to determine its efficacy in producing schedules that minimize TWT under varying job conditions. First, we consider the static $1|p-\text{batch}|\sum T_j$ problem, comparing BIA with the BATC algorithm of Mehta and Uzsoy (1998) and B_MDD. In this case, job j's weight $w_i=1, \forall j$. Next, we consider the static $1|p-\text{batch}|\sum w_jT_j$ problem where in $wj\sim DU[1,10]$, $\forall j$. In this case, BIA will be compared with WBATC, a variant of BATC that incorporates job weights into the index calculation and BWMDD.

We then investigate the dynamic $1|r_j,p-\text{batch}|\sum T_j$ problem that is characterized by non-zero job ready times and unit job weights. Here, a variant of BATC is used that incorporates non-zero job ready times into the index calculation (DBATC) as well as DB_MDD. Finally, we combine non-zero job ready times and non-unit job weights to compare the performance of BIA to DWBATC and DBWMDD for the dynamic $1|r_j,p-\text{batch}|\sum w_jT_j$ problem.

Table 1 summarizes the problems considered in this computational study as well as the approaches used to attack each of the problems.

Table 1. Problem Approaches Investigated.

Problem Class	Methods
$1\|p-\text{batch}\|\sum T_j$	BATC, B_MDD, BIA
$1\|p-\text{batch}\|\sum w_jT_j$	WBATC, BWMDD, BIA
$1\|r_j,p-\text{batch}\|\sum T_j$	DBATC, DB_MDD, BIA
$1\|r_j,p-\text{batch}\|\sum w_jT_j$	DWBATC, DBWMDD, BIA

Table 2. Experimental design used in random problem generation.

Problem factor	Values used	Total values
Jobs per family n_j	30, 40, 50, 60	4
Number of families m	3, 4, 5, 6	4
Maximum batch size B	4, 6, 8	3
Family processing time (hrs) P_j	$P(p_j=2)=P(p_j=4)=P(p_j=16)=0.2$ $P(p_j=10)=0.3, P(p_j=20)=0.1$	1
Job ready time r_j	*Static cases* (*S*): $r_j=0,\ \forall j$	1 (each S)
	Dynamic cases (*D*): $rj\sim DU$ $[0, \lfloor\alpha C_{\max}\rfloor]$ $\alpha\in\{0.5, 1, 1.5\}\quad C_{\max}=\frac{\sum_{j=1}^{m}\sum_{i=1}^{n_j}p_j}{B}$	3 (each D)
Job due date d_{ij}	$\sim DU\left[\lceil\mu-\frac{\mu R}{2}\rceil, \lfloor\mu+\frac{\mu R}{2}\rfloor\right]$ $\mu=C_{\max}(1-T);\ R\in\{0.5, 2.5\};\ T\in\{0.3, 0.6\}$	4
	Total factor combinations	192 (each S) 576 (each D)
	Replications per combination	10
	Total problems	1920 (each S) 5760 (each D)

6.2 *Experimental design*

We pattern our experimental design after the one presented by Mehta and Uzsoy (1998) to test the proposed algorithms in each of the cases mentioned in the previous section (table 2). A total of n_j jobs are created for each job family $j=1,\ldots,m$. All jobs in a given family require the same amount of processing time which is sampled from the discrete probability distribution given in table 2. Three different values for maximum batch size B are investigated, along with four different discrete uniform distributions for the due date of job i in family j ($i=1,\ldots,n_i$). Job due dates are characterized both by due date range factor R and by due date tightness factor T; job due dates are not family dependent. Finally, four different discrete uniform distributions are evaluated for both the Dynamic Total Tardiness and Dynamic TWT cases.

In total, the two static cases each contain $4\times4\times3\times1\times4=192$ design points, while the addition of non-zero job ready times results in $4\times4\times4\times3\times3\times4=576$ design points for each of the two dynamic cases. As 10 replicates are generated for each design point, a total of 3.840 static and 11 520 dynamic problem instances

will be generated and examined. For each instance, the resulting schedule's total (weighted) tardiness will be recorded. When employing BATC, WBATC, DBATC, and DWBATC, we employ a grid search to determine the value of the smoothing parameter k that best minimizes TWT. In each grid search, we examine values of k over the range 0.1 to 10 in increments (steps) of 0.1. To assess whether this grid search and the proposed BIA are computationally feasible, we also measure the computation time required to produce the schedule for each problem instance.

6.3 *Computational results*

Let $G(h, i)$ denote the objective function value resulting from the schedule produced by heuristic h on problem instance i. Further, heuristic $h \in H$, where H denotes the set of all competing heuristics for a given scheduling problem class (e.g., BATC, B_MDD, and BIA for the $1|p-\text{batch}|\sum T_j$ problem class). In order to assess the performance of the competing heuristics, we compute the performance ratio $PR(h) = G(h, i)/(\min_{h \in H} G(h, i))$. For each problem instance i, the best value of $PR(h) = 1.000$ signifies that heuristic h produces the schedule with the lowest objective function value in comparison to the other competing heuristic approaches.

Table 3 presents a high level overview of the experimental results for all four problem classes under study. In the table, the bolded $PR(h)$ value indicates the best performing heuristic approach. While the BATC family outperforms the B_MDD family and BIA for the zero ready time cases with and without job weights (i.e., the first two results columns), BIA is able to produce higher quality solutions for the more practical $1|r_j, p-\text{batch}|\sum T_j$ and $1|r_j, p-\text{batch}|\sum w_j T_j$ problem classes and compared to the competition (see the last two results columns). The following subsections examine the experimental results and required computation time for each of the four problem classes under study. All experiments summarized below were conducted on a SunFire V480 with a 900 MHz processor and 8 GB of RAM.

6.3.1 $1|p-\text{batch}|\sum T_j$ results. BIA outperforms both BATC and B_MDD in only 48 of the 1920 static total tardiness problem instances (i.e., in 2.5% of the instances). Even though 100 different k values were examined to determine the "best" smoothing parameter value, BATC required only 0.36 seconds on average to conduct the entire grid search and produce its superior results. Further, BATC produces the best overall schedule in 89.7% of the static problem instances under study. The minimum time required for BATC to produce a schedule was 0.07 seconds,

Table 3. Average $PR(h)$ values for each problem class by heuristic family.

Problem Class	BATC Family	B_MDD Family	BIA
$1\|p-\text{batch}\|\sum T_j$	**1.032 (BATC)**	1.133 (B_MDD)	3.342
$1\|p-\text{batch}\|\sum w_j T_j$	**1.452 (WBATC)**	1.517 (BWMDD)	1.488
$1\|r_j, p-\text{batch}\|\sum T_j$	1.891 (DBATC)	2.245 (DB_MDD)	**1.042**
$1\|r_j, p-\text{batch}\|\sum w_j T_j$	1.923 (DWBATC)	2.108 (DBWMDD)	**1.018**

while the maximum time was 1.09 seconds. B_MDD requires 0.01 seconds in the worst case to produce a schedule for a given problem instance and produces the best overall schedule for 25.9% of the problem instances, tying BATC in some cases.

Both competing heuristics are, on average, faster than BIA's 0.46 seconds of required computation time. BIA's minimum and maximum solution times were 0.01 seconds and 3.25 seconds, respectively. The average performance ratio values for the three competing heuristics are as follows: $PR(BATC) = 1.032$, $PR(B\_MDD) = 1.133$, and $PR(BIA) = 3.342$ (table 3). These results are not surprising, given that the competing heuristics were developed specifically for this class of single machine batch scheduling problem.

6.3.2 $1|p-\text{batch}|\sum w_jT_j$ results. Although BIA's performance improves considerably when non-unit job weights are introduced, it only outperforms WBATC and BWMDD in 23.5% of the static TWT problem instances. WBATC (BWMDD) produces the best overall schedule in 34.1% (42.7%) of the problem instances, with some ties. WBATC requires only 0.38 seconds on average (minimum of 0.08 seconds, maximum of 1.13 seconds) to produce its best schedule, while BIA requires an average of 0.79 seconds per problem instance (minimum of 0.04 seconds, maximum of 3.67 seconds). Comparable to B_MDD, BWMDD requires less than 0.01 seconds in the worst case to produce a schedule.

The average performance ratio values for the three competing heuristics are as follows: $PR(WBATC) = 1.452$, $PR(BWMDD) = 1.517$, and $PR(BIA) = 1.488$ (table 3). Under the existence of a wide job due date range ($R = 2.5$) and tightened job due dates ($T = 0.6$), the worst BIA performance of $PR(BIA) = 1.619$ is realized. When $R = 0.5$, BIA produces schedules with an average $PR(BIA)$ value of 1.467 for $T = 0.3$ and 1.560 when $T = 0.6$. While the introduction of non-unit job weights improved the overall performance of BIA relative to the competing heuristics, BIA still requires two times as much computation time to produce its results, results that are 3% worse on average in terms of TWT for static $1|p-\text{batch}|\sum w_jT_j$ problem instances.

6.3.3 $1|r_j, p-\text{batch}|\sum T_j$ results. Experimental results suggest that BIA is very effective at producing low total tardiness schedules when non-zero job ready times are present. BIA outperforms both DBATC and DB_MDD in 76.6% of the 5 760 dynamic total tardiness problem instances. Comparable to the solution time required by the other BATC variants, DBATC required 0.37 seconds per problem instance on average to produce a schedule, with the minimum (maximum) DBATC solution time being 0.08 (1.14) seconds. For this dynamic, unit job weight case, BIA required 0.53 seconds on average to generate a schedule, with a minimum (maximum) time of 0.01 (3.49) seconds. As before, DB_MDD requires a negligible amount of computational time.

The average performance ratio values for the three competing heuristics are as follows: $PR(DBATC) = 1.891$, $PR(DB\_MDD) = 2.245$, and $PR(BIA) = 1.042$ and (table 3). Figure 2 illustrates the relative performance of BIA, DB_MDD, and DBATC when $\alpha = 0.5$, which is representative of BIA's overall performance for this problem class. The results in figure 2 are aggregated over all levels of jobs per family and batch size in order to promote visual clarity of the results presentation.

BIA performs poorest when $R = 0.5$, albeit while outperforming the competing heuristics. As the due date range increases to $R = 2.5$, the performance of BIA improved by 10% at the same time when the performance of both DBATC and DB_MDD degrade substantially.

6.3.4 $1|r_j, p-\text{batch}|\sum w_j T_j$ results. Even though an exhaustive 100 point grid search was conducted for DWBATC, the variant of BATC that accommodates non-zero job ready times and non-unit job weights, BIA produces schedules with lower TWT in 83.5% of the $1|r_j, p-\text{batch}|\sum w_j T_j$ problem instances investigated for both DWBATC and DBWMDD. Figure 3 summarizes the results of these experiments, providing the average $PR(h)$ aggregated over both the number of jobs per family (n_i) and ready-time factor α. On average, $PR(BIA) < PR(DWBATC)$ and $PR(BIA) < PR(DBWMDD)$and for all cases. The average performance ratio values for the three competing heuristics are as follows: $PR(DWBATC) = 1.923$, $PR(DBWMDD) = 2.108$, and $PR(BIA) = 1.018$ (table 3).

In the figure 3 problem instances with 50 jobs, there is a noticeable degradation in the performance of *BIA*'s competitors. By splitting jobs into full batches of 4, 6, or 8, as required in the BATC-based algorithms, we see different numbers of jobs in partial batches, summarized in table 4. Those jobs in partial batches get placed into the partial batches based entirely on their ready times, but by the time they get processed, their ready times are essentially irrelevant, because the completion times of the previous batches are much larger. The impact of this is greatest for the 50 job case because 2 jobs per family get this problem, no matter the maximum batch size. In the 30, 40 and 60 job cases, there are fewer cases where the batches are not all full.

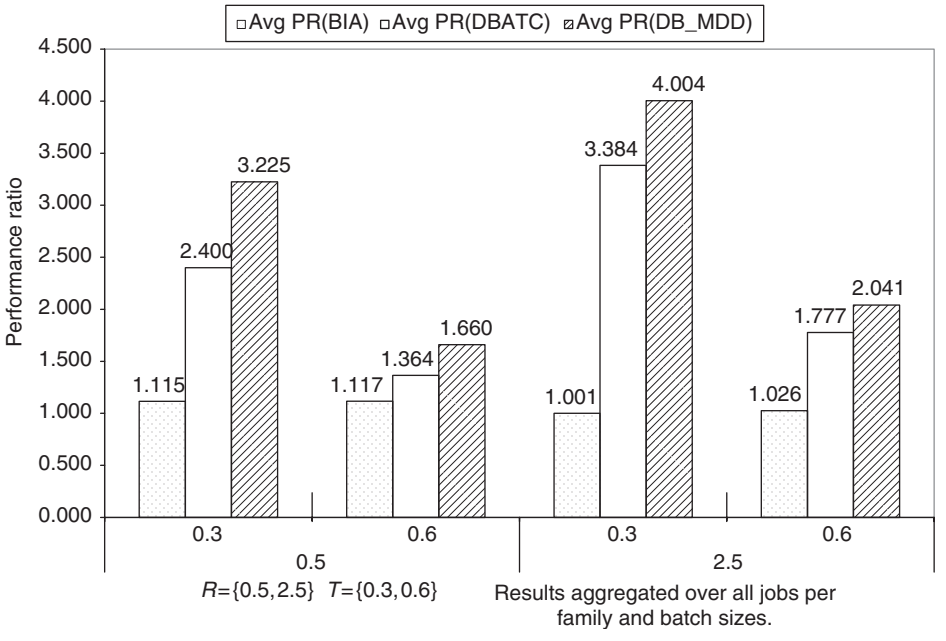

Figure 2. Average $PR(h)$ as a function of R and T when $\alpha = 0.5$

Again comparable to its previous computation performance, the BATC variant DWBATC required 0.38 seconds on average to compute a given job schedule, with a minimum computation time of 0.08 seconds and a maximum time of 1.15 seconds. BIA requires approximately twice as long to produce its schedules: 0.73 seconds on average, with a minimum time of 0.02 and a maximum time of 4.1 seconds. Finally, DBWMDD's required computation time was again negligible. Therefore, experimental results suggest the computational price paid for using the BIA heuristic is minimal compared to the significant TWT savings it produces.

7. Conclusions and future research

This paper has presented adaptations of BATC (Mehta and Uzsoy 1998) and WMDD (Kanet and Li 2004) for problems leading to the total weighted tardiness problem with non-zero job ready times, as well as a Batch Improvement

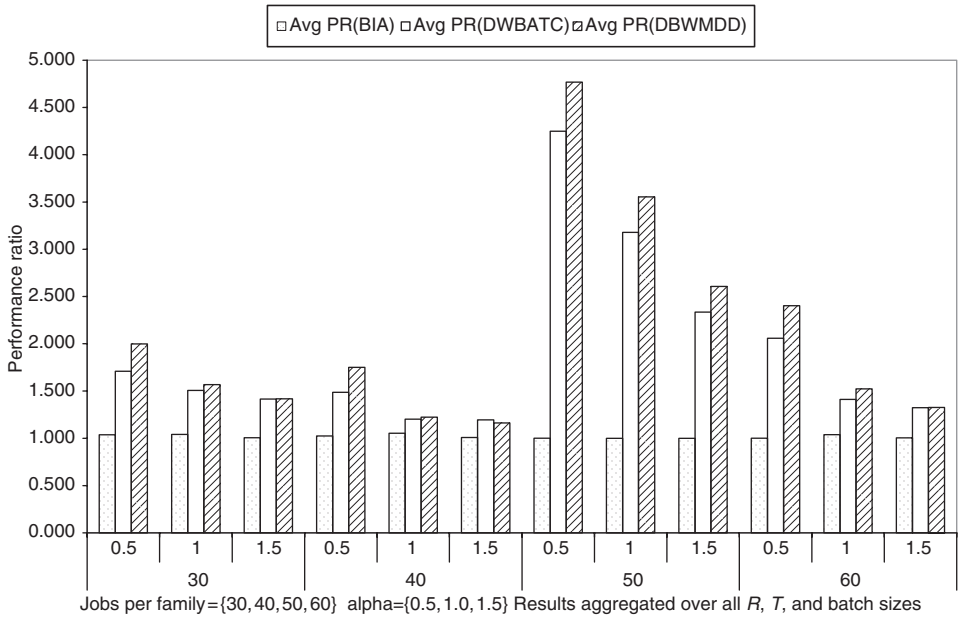

Figure 3. Average $PR(h)$ as a function of n_j and α

Table 4. Number of jobs in partial batches as a function of number of jobs per family and maximum batch size.

n_j	$B=4$	$B=6$	$B=8$
30	2	0	2
40	0	4	0
50	2	2	2
60	0	0	4

Algorithm (BIA). Through extensive computational experiments, BIA is shown to be well equipped to find low total weighted tardiness batch schedules, though the BATC and WMDD variants can occasionally find good solutions as well. The performance of BIA improves as the batch-scheduling problem moves from the initial conditions for which BATC was developed to address; namely, the introduction of ready times and weights for jobs favors the use of BIA over the other two competitors. In 83.5% of the problem instances with job weights and ready times, BIA finds lower TWT schedules.

BIA was developed to specifically allow the formation of partial batches in positions other-than-in the last position of the schedule, and this design feature is shown to be useful and effective. Due to the success in using BIA for the single batch-processing machine problem, we will adapt BIA for use in identical and non-identical parallel batch-processing machine environments where again jobs belong to incompatible families, comparing especially to the approach in Mönch *et al.* (2005). We hope to build upon the BIA's success so that effective, useful schedules can be generated quickly for this even more challenging and practical machine environment.

Appendix

Table 5 contains nine jobs belonging to two families, with weights, ready times, due dates and processing times as shown. Jobs 1–5 belong to family 1 and jobs 6–9 belong to family 2. The data has already been sorted by non-decreasing ready times followed by non-decreasing weighted due dates and broken into batches based on the family. If $B = 3$, $K = 7$ initial batches are formed. The ready times, starting times and completion times of each batch have been calculated, as well as the tardiness and weighted tardiness of each job. The total weighted tardiness is shown in the bottom of the table. A naïve lower bound on the total weighted tardiness can be found by calculating the sum of the weighted minimum tardiness for each job, $\sum_{\forall ij} w_{ij} \max(r_{ij} + p_j - d_{ij}, 0)$. In this case, the value is 74.

We begin with $k = K - 2 = 5$.

$k = 5$: The conditions in steps 1.1 to 1.4 do not apply, so we go to Step 2. Our goal is to find jobs in B_6 or B_7 that are ready no later than 95.

Table 5. Initial Schedule.

Job	w_{ij}	r_{ij}	d_{ij}	d_{ij}/w_{ij}	P_j	Family	Batch k	R_k	S_k	C_k	T_{ij}	$w_{ij}T_{ij}$
6	1	47	78	78	20	2	1	47	47	67	0	0
1	6	49	122	20.33	4	1	2	49	67	71	0	0
7	5	64	70	14	20	2	3	64	71	91	21	105
2	4	64	74	18.5	4	1	4	69	91	95	21	84
3	7	68	98	14	4	1		69	91	95	0	0
4	2	69	86	43	4	1		69	91	95	9	18
8	1	70	86	86	20	2	5	70	95	115	29	29
5	7	71	90	12.86	4	1	6	71	115	119	29	203
9	3	71	95	31.67	20	2	7	71	119	139	44	132
												571

B_5 contains jobs from family 2, so $l=2$. In step 3, we form E with jobs in family 2, ready by 95 and in B_6 or B_7. The only job that satisfies these conditions is job 9 in B_7, so $E=\{9\}$. In step 4, we add job 9 to B_5, so $B_5=\{8,9\}$, $E=\emptyset$, $B_7=\emptyset$ and $f=7$. We recurse to BIA(7).

$k=7$: In step 1, we update the batch ready, start and completion times, as well as the job completion, tardiness and weighted tardiness values. In step 1.3, we discover that we are at the last batch and it is empty. Therefore, we delete it and return to step 4 with $k=5$. The resulting schedule is shown in table 6.

$k=5$ con't: Step 4 is complete as $E=\emptyset$ Step 5 calls BIA(4).

$k=4$: There is no need to update the schedule at this point. The condition in step 1.2 holds since B_4 is full, so we call BIA(3)

$k=3$: There is no need to update the schedule at this point. The conditions in steps 1.1 to 1.4 do not apply, so we go to Step 2. In step 2.1, we set $l=2$. Step 3 sets $E=\{8,9\}$. In step 4, we add jobs from E to B_3 as long as E has jobs and B_3 has room, starting with the job with the largest weighted tardiness. Job 9 has the higher weighted tardiness, so $B_3=\{7,9\}$, $E=\{8\}$, $B_5=\{8\}$ and $f=5$. Fill the open space in B_5 by calling BIA(5).

$k=5$: In step 1, update the schedule times. Notice that several batch ready times have changed, which could have impacted the completion times of several batches in a different problem. In step 1.4, we discover that we are at the last batch and it is not empty. Therefore, we return to step 4 with $k=3$. The resulting schedule is shown in table 7.

$k=3$: The result of step 5 is $B_3=\{7,8,9\}$, $E=\emptyset$, $B_5=\emptyset$ and $f=5$. Call BIA(5).

$k=5$: In step 1, update the schedule times. The resulting schedule is shown in table 8. Notice that B_5 is empty but is still shown in the schedule.

The conditions in steps 1.1 to 1.4 do not apply, so we go to Step 2. We cannot go to step 2.1 because $B_5=\emptyset$. Therefore, we begin looking for a job that can "fit" in the space formerly occupied by the jobs in B_5. If we find one, we know that the jobs in batches after B_5 will not be delayed. If there are none, we simply move all the batches up,

Table 6. After job 9 moved to batch 5.

Job	w_{ij}	r_{ij}	d_{ij}	d_{ij}/w_{ij}	P_j	Family	Batch k	R_k	S_k	C_k	T_{ij}	$w_{ij}T_{ij}$
6	1	47	78	78	20	2	1	47	47	67	0	0
1	6	49	122	20.33	4	1	2	49	67	71	0	0
7	5	64	70	14	20	2	3	64	71	91	21	105
2	4	64	74	18.5	4	1	4	69	91	95	21	84
3	7	68	98	14	4	1		69	91	95	0	0
4	2	69	86	43	4	1		69	91	95	9	18
8	1	70	86	86	20	2	5	71	95	115	29	29
9	3	71	95	31.67	20	2		71	95	115	20	60
5	7	71	90	12.86	4	1	6	71	115	119	29	203
												499

as long as the batch ready times allow. In step 2.2, we form E as the set of jobs that are ready before 95, in a batch after B_5 and with processing time less than 115–95 = 20. $E = \{5\}$. We go to step 2.2.2 which instructs us to add the job with the largest weighted tardiness in E to B_5. The result is $B_5 = \{5\}$, $E = \emptyset$, $B_6 = \emptyset$ and $f = 6$. Call BIA(6).

$k = 6$: In step 1, update the schedule times. In step 1.3, we discover that we are at the last batch and it is empty. Therefore, we delete it by setting $K = 5$ and return to step 2.2.2 with $k = 5$. The resulting schedule is shown in table 9.

$k = 5$: In step 2.2.2, we simply continue on to step 3. Step 3 finds $E = \emptyset$, so skip step 4. Step 5 calls BIA(4).

$k = 4$: There is no need to update the schedule at this point. The condition in step 1.2 applies, so call BIA(3).

$k = 3$: There is no need to update the schedule at this point. The conditions in steps 1.1 to 1.4 do not apply, so we go to Step 2. In step 2.1, we set $l = 2$. Step 3 finds $E = \emptyset$ so skip step 4. Step 5 calls BIA(2).

Table 7. After job 9 moved to batch 3.

Job	w_{ij}	r_{ij}	d_{ij}	d_{ij}/w_{ij}	P_j	Family	Batch k	R_k	S_k	C_k	T_{ij}	$w_{ij}T_{ij}$
6	1	47	78	78	20	2	1	47	47	67	0	0
1	6	49	122	20.33	4	1	2	49	67	71	0	0
7	5	64	70	14	20	2	3	71	71	91	21	105
9	3	71	95	31.67	20	2		71	71	91	0	0
2	4	64	74	18.5	4	1	4	69	91	95	21	84
3	7	68	98	14	4	1		69	91	95	0	0
4	2	69	86	43	4	1		69	91	95	9	18
8	1	70	86	86	20	2	5	70	95	115	29	29
5	7	71	90	12.86	4	1	6	71	115	119	29	203
												439

Table 8. After job 8 moved to batch 3.

Job	w_{ij}	r_{ij}	d_{ij}	d_{ij}/w_{ij}	P_j	Family	Batch k	R_k	S_k	C_k	T_{ij}	$w_{ij}T_{ij}$
6	1	47	78	78	20	2	1	47	47	67	0	0
1	6	49	122	20.33	4	1	2	49	67	71	0	0
7	5	64	70	14	20	2	3	71	71	91	21	105
9	3	71	95	31.67	20	2		71	71	91	0	0
8	1	70	86	86	20	2		71	71	91	5	5
2	4	64	74	18.5	4	1	4	69	91	95	21	84
3	7	68	98	14	4	1		69	91	95	0	0
4	2	69	86	43	4	1		69	91	95	9	18
(empty)							5					
5	7	71	90	12.86	4	1	6	71	115	119	29	203
												415

$k=2$: There is no need to update the schedule at this point. The conditions in steps 1.1 to 1.4 do not apply, so we go to Step 2. In step 2.1, we set $l=1$. Step 3 sets $E=\{2\}$. In step 4, we find $B_2=\{1,2\}$, $E=\emptyset$, $B_4=\{3,4\}$ and $f=4$. Call BIA(4).

$k=4$: In step 1, update the schedule times. The resulting schedule is shown in table 10.

The conditions in steps 1.1 to 1.4 do not apply, so we go to Step 2. In step 2.1, we set $l=1$. Step 3 sets $E=\{5\}$ The result of step 4 is $B_4=\{3,4,5\}$, $E=\emptyset$, $B_5=\emptyset$ and $f=5$. Call BIA(5).

$k=5$: In step 1, update the schedule times. In step 1.3, we discover that we are at the last batch and it is empty, so we set $K=4$. We return to step 4 with $k=4$. The resulting schedule is shown in table 11.

$k=4$: In step 4, we see $E=\emptyset$, so step 5 calls BIA(3).

$k=3$: There is no need to update the schedule at this point. The conditions in steps 1.1 to 1.4 do not apply, so we go to Step 2. In step 2.1, we set $l=2$. Step 3 finds $E=\emptyset$ so skip step 4. Step 5 calls BIA(2).

Table 9. After job 5 moved to batch 5.

Job	w_{ij}	r_{ij}	d_{ij}	$d_{ij}\,w_{ij}$	P_j	Family	Batch k	R_k	S_k	C_k	T_{ij}	$w_{ij}T_{ij}$
6	1	47	78	78	20	2	1	47	47	67	0	0
1	6	49	122	20.33	4	1	2	49	67	71	0	0
7	5	64	70	14	20	2	3	71	71	91	21	105
9	3	71	95	31.67	20	2		71	71	91	0	0
8	1	70	86	86	20	2		71	71	91	5	5
2	4	64	74	18.5	4	1	4	69	91	95	21	84
3	7	68	98	14	4	1		69	91	95	0	0
4	2	69	86	43	4	1		69	91	95	9	18
5	7	71	90	12.86	4	1	5	71	95	99	9	63
												275

Table 10. After job 2 moved to batch 2.

Job	w_{ij}	r_{ij}	d_{ij}	$d_{ij}\,w_{ij}$	P_j	Family	Batch k	R_k	S_k	C_k	T_{ij}	$w_{ij}T_{ij}$
6	1	47	78	78	20	2	1	47	47	67	0	0
1	6	49	122	20.33	4	1	2	64	67	71	0	0
2	4	64	74	18.5	4	1		64	67	71	0	0
7	5	64	70	14	20	2	3	71	71	91	21	105
9	3	71	95	31.67	20	2		71	71	91	0	0
8	1	70	86	86	20	2		71	71	91	5	5
3	7	68	98	14	4	1	4	69	91	95	0	0
4	2	69	86	43	4	1		69	91	95	9	18
5	7	71	90	12.86	4	1	5	71	95	99	9	63
												191

Table 11. After job 5 moved to batch 4.

Job	w_{ij}	r_{ij}	d_{ij}	d_{ij}/w_{ij}	P_j	Family	Batch k	R_k	S_k	C_k	T_{ij}	$w_{ij}T_{ij}$
6	1	47	78	78	20	2	1	47	47	67	0	0
1	6	49	122	20.33	4	1	2	64	67	71	0	0
2	4	64	74	18.5	4	1		64	67	71	0	0
7	5	64	70	14	20	2	3	71	71	91	21	105
9	3	71	95	31.67	20	2		71	71	91	0	0
8	1	70	86	86	20	2		71	71	91	5	5
3	7	68	98	14	4	1	4	71	91	95	0	0
4	2	69	86	43	4	1		71	91	95	9	18
5	7	71	90	12.86	4	1		71	91	95	5	35
												163

$k=2$: There is no need to update the schedule at this point. The conditions in steps 1.1 to 1.4 do not apply, so we go to Step 2. In step 2.1, we set $l=1$. Step 3 finds $E=\emptyset$ so skip step 4. Step 5 calls BIA(1).

$k=1$: There is no need to update the schedule at this point. The conditions in steps 1.1 to 1.4 do not apply, so we go to Step 2. In step 2.1, we set $l=2$. Step 3 finds $E=\emptyset$ so skip step 4. Step 5 calls BIA(0).

$k=0$: Stop. BIA returns a total weighted tardiness value of 163.

In the application of BIA to the example problem, the total weighted tardiness was reduced from 571 to 163, which is much closer to the naïve lower bound of 74. In fact, BIA's solution is 21.6% above the true optimal solution of 134 as determined by a mixed-integer programming formulation of the single machine, batch scheduling problem with ready times.

References

Baker, K.R. and Bertrand, J.W.M., A dynamic priority rule for scheduling against due-dates. *J. Oper. Manag.*, 1982, **3**, 37–42.

Brucker, P., Gladky, A., Hoogeveen, H., Kovalyov, M.Y., Potts, C.N., Tautenhahn, T. and Van de Velde, S., Scheduling a batching machine. *J. Sched.*, 1998, **1**, 31–54.

Chandru, V., Lee, C.-Y. and Uzsoy, R., Minimizing total completion time on batch processing machines. *Int. J. Prod. Res.*, 1993, **31**(9), 2097–2121.

Coffman, E., Yannakakis, M., Magazine, M. and Santos, C., Batch sizing and job sequencing on a single machine. *Annals of Oper. Res.*, 1990, **26**, 135–147.

Hochbaum, D. and Landy, D., Scheduling semiconductor burn-in operations to minimize total flowtime. *Oper. Res.*, 1997, **45**(6), 874–885.

Kanet, J.J. and Li, X., A weighted modified due date rule for sequencing to minimize weighted tardiness. *J. Sched.*, 2004, **7**, 263–278.

Kurz, M.E. (2003). On the structure of optimal schedules for minimizing total weighted tardiness on parallel, batch-processing machines. *Industrial Engineering Research Conference Proceedings 2003*, Portland OR (CD-ROM).

Lawler, E.L., Lenstra, J.K. and Rinnooy Kan, AHG., Recent developments in deterministic sequencing and scheduling: A survey. In *Deterministic and Stochastic Scheduling*, edited by M.A.H. Dempster, J.K. Lenstra and AHG Rinnooy Kan, pp. 35–74, 1982 (D. Reidel Publishing Co.: The Netherlands).

Lee, C.Y. and Uzsoy, R., Minimizing makespan on a single batch processing machine with dynamic job arrivals. *Int. J. Prod. Res.*, 1999, **37**, 219–236.

Lenstra, J.K., Rinnooy Kan, A.H.G. and Brucker, P., Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1977, **1**, 343–362.

Mason, S.J., Fowler, J.W. and Carlyle, W.M., A modified shifting bottleneck heuristic for minimizing total weighted tardiness in complex job shops. *J. Sched.*, 2002, **5**(3), 247–262.

Mehta, S.V. and Uzsoy, R., Minimizing total tardiness on a batch processing machine with incompatible job families. *IIE Trans.*, 1998, **30**, 165–178.

Mönch, L., Balasubramanian, H., Fowler, J.W. and Pfund, M.E., Heuristic scheduling of jobs on parallel batch machines with incompatible job families and unequal ready times. *Computers and Oper. Res.*, 2005, **32**, 2731–2750.

Perez, I.C., Fowler, J.W. and Carlyle, W.M., Minimizing total weighted tardiness on a single batch process with incompatible job families. *Computers and Oper. Res.*, 2005, **32**, 327–341.

Potts, C., Strusevich, V. and Tautenhahn, T., Scheduling batches with simultaneous job processing for two-machine shop problems. *J. Sched.*, 2001, **4**(1), 25–51.

Sung, C. and Choung, Y., Minimizing makespan on a single burn-in oven in semiconductor manufacturing. *Eur. J. Oper. Res.*, 2000, **120**, 559–574.

Uzsoy, R., Scheduling batch processing machines with incompatible job families. *Int. J. Prod. Res.*, 1995, **33**(10), 2685–2708.

Van der Zee, D.-J., van Harten, A. and Schuur, P., On-line scheduling of multi-server batch operations. *IIE Trans.*, 2001, **33**, 569–586.

Vepsalainen, A. and Morton, T.E., Priority rules and lead time estimation for job shop scheduling with weighted tardiness costs. *Manage. Sci.*, 1987, **33**, 1036–1047.