www.elsevier.com/locate/caor

Split-merge: Using exponential neighborhood search for scheduling a batching machine

Marta Cabo, Edgar Possani, Chris N. Potts, Xiang Song

PII: S0305-0548(15)00097-0
DOI: http://dx.doi.org/10.1016/j.cor.2015.04.017
Reference: CAOR3773

To appear in: *Computers & Operations Research*

Split-Merge: Using Exponential Neighborhood Search for Scheduling a Batching Machine

Marta Cabo[1a], Edgar Possani[1b*], Chris N. Potts[3], Xiang Song[2]

[1] *Department of Mathematics, Instituto Tecnológico Autónomo de México, Río Hondo No.1, Progreso Tizapán, Mexico City, D.F. 01080, Mexico. a) marta.cabo@itam.mx, b) epossani@itam.mx*

[2] *School of Mathematics, University of Southampton, Highfield, Southampton, SO17 1BJ, UK. C.N.Potts@soton.ac.uk*

[3] *Department of Mathematics, University of Portsmouth, Lion Gate Building, Lion Terrace, Portsmouth PO1 3HF, UK. xiang.song@port.ac.uk*

Abstract

We address the problem of scheduling a single batching machine to minimize the maximum lateness with a constraint restricting the batch size. A solution for this NP-hard problem is defined by a selection of jobs for each batch and an ordering of those batches. As an alternative, we choose to represent a solution as a sequence of jobs. This approach is justified by our development of a dynamic program to find a schedule that minimizes the maximum lateness while preserving the underlying job order. Given this solution representation, we are able to define and evaluate various job-insert and job-swap neighborhood searches. Furthermore we introduce a new neighborhood, named split-merge, that allows multiple job inserts in a single move. The split-merge neighborhood is of exponential size, but can be searched in polynomial time by dynamic programming. Computational results with an iterated descent algorithm that employs the split-merge neighborhood show that it compares favorably with corresponding iterated descent algorithms based on the job-insert and job-swap neighborhoods.

Keywords: batching machine, maximum lateness, local search, exponential neighborhoods, dynamic programming.

[*]Corresponding author. Email: epossani@itam.mx

1. Introduction

A batching machine is one that can process more than one job at the same time. Batching machines are common in the metalworking, chemical and microelectronics industries. We consider the case where the batching machine can process up to a certain number of jobs simultaneously (restricted batch size), and the processing time of a batch is equal to the largest processing time among all jobs within the batch. The scheduling problem we tackle in this paper is motivated by large-scale integrated circuit manufacturing as explained by Lee et al. [1]. Their model considers a burn-in oven, which operates as a batching machine to perform the final testing of printed circuits. The purpose of burn-in operations is to subject the chips to thermal stress in order to bring out latent defects. The burn-in time for each chip is specified by the type of product and it is known a priori. A number of products can be grouped in a batch to be processed together, where the processing time of the batch is the longest processing time among all jobs, due to the fact that it is possible to keep a circuit longer in the oven than its prescribed burn-in time but not taken out before. This batching machine process is critical for on-time delivery of the circuits since the processing times in burn-in operations are much longer than other operations in the testing areas. Thus, the efficient scheduling of these operations is of great concern to management.

A recent survey on scheduling semiconductor manufacturing operations can be found in [2]. Different batching machine settings are analyzed in the literature. Wang and Uzsoy [3] develop a genetic algorithm for the one batching machine problem to minimize the maximum lateness where jobs have release dates. Kashan et al. [4] focus on minimizing the makespan through a genetic algorithm that outperforms the simulated annealing approach suggested by Melouk et al. [5]. Recently Wang [6] considers the problem on minimizing total weighted tardiness with restricted batch sizes.

This paper considers a batching machine problem that has the following characteristics. Let $J = \{1, \ldots, n\}$ denote a set of n jobs that are to be processed. The maximum number of jobs the machine can process at a time is b, where $b < n$, so that the batch size is restricted. Associated with each job j, for $j = 1, \ldots, n$, is its non-negative processing time p_j and its due date d_j. We assume all jobs become available at time zero.

A schedule σ for this problem is a sequence of batches $\sigma = (\mathcal{B}_1, \ldots, \mathcal{B}_r)$, where r is the number of batches and $\mathcal{B}_k$ is a batch comprising a subset of jobs for $k = 1, \ldots, r$. There is a limit of b on the number of jobs in each batch, which implies that $|\mathcal{B}_k| \leq b$, for $k = 1, \ldots, r$. The processing time of each batch $\mathcal{B}_k$ is given by

$$p(\mathcal{B}_k) = \max_{j \in \mathcal{B}_k} \{p_j\},$$

and its completion time is

$$C(\mathcal{B}_k) = \sum_{j=1}^{k} p(\mathcal{B}_j).$$

Further, all jobs in a batch start and complete at the same time, which implies that the completion time of job j in schedule σ, for each $j \in \mathcal{B}_k$ is $C_j(\sigma) = C(\mathcal{B}_k)$.

Let $L_{\max}(\sigma)$ denote the maximum lateness of jobs in schedule σ, where the lateness of job j is given by $L_j(\sigma) = C_j(\sigma) - d_j$. We define $d(\mathcal{B}_k) = \min\limits_{j \in \mathcal{B}_k} d_j$ to be the due date of batch $\mathcal{B}_k$. Then there are two ways of computing the maximum lateness of a given schedule σ: either over the jobs using

$$L_{\max}(\sigma) = \max_{1 \leq j \leq n} \{C_j(\sigma) - d_j\},$$

or over the batches using

$$L_{\max}(\sigma) = \max_{1 \leq k \leq r} \{C(\mathcal{B}_k) - d(\mathcal{B}_k)\}.$$

The batching machine problem we study is to find an optimal schedule σ^* which minimizes the maximum lateness $L_{\max}(\sigma^*) = \min_\sigma L_{\max}(\sigma)$.

To illustrate this problem, we consider an instance with five jobs, a batch size $b = 3$, and processing times and due dates as follows:

Job j	1	2	3	4	5
p_j	1	4	8	7	7
d_j	5	10	16	16	7

A solution to this problem requires a selection of jobs for each batch and an ordering of those batches. As shown in Figure 1, a possible schedule with two batches is $\sigma_1 = (\{1, 2, 5\}, \{4, 3\})$ with $L_{\max} = 2$, an alternative schedule with 3 batches is $\sigma_2 = (\{1\}, \{2, 5, 4\}, \{3\})$ with $L_{\max} = 1$. Note that a schedule with fewer batches does not necessarily guarantee a better solution.

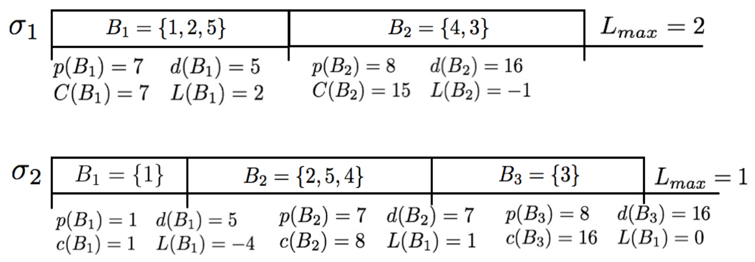
σ_1
$B_1 = \{1,2,5\}$
$B_2 = \{4,3\}$
$L_{max} = 2$
$p(B_1) = 7$ $d(B_1) = 5$
$C(B_1) = 7$ $L(B_1) = 2$
$p(B_2) = 8$ $d(B_2) = 16$
$C(B_2) = 15$ $L(B_2) = -1$
σ_2
$B_1 = \{1\}$
$B_2 = \{2,5,4\}$
$B_3 = \{3\}$
$L_{max} = 1$
$p(B_1) = 1$ $d(B_1) = 5$
$c(B_1) = 1$ $L(B_1) = -4$
$p(B_2) = 7$ $d(B_2) = 7$
$c(B_2) = 8$ $L(B_1) = 1$
$p(B_3) = 8$ $d(B_3) = 16$
$c(B_3) = 16$ $L(B_1) = 0$

Figure 1: Possible solutions for a 5-job instance

This problem is shown to be unary (or strongly) NP-hard by Brucker et al. [7]. Hence, there is an interest in developing local search heuristics for the problem.

A neighborhood search associates a neighborhood structure to any given solution s, and restricts the (local) search to the neighborhood of s. It is generally expected that a very large neighborhood will yield better solutions, especially if it can be searched efficiently. Over the last decade there has been much interest in such very large neighborhoods, especially to address difficult combinatorial optimization problems, like the multi-resource generalized assignment problem [8], the traveling salesman problem as in [9] or [10], the time-tabling problem [11] or the vehicle routing problem [12], as well as real-life applications as in the through-fleet-assignment problem [13] or a car sequencing application in [14]. Defining an exponential size neighborhood is not sufficient to guarantee an efficient local search algorithm. An example of this is highlighted in [9] when dealing with the quadratic assignment problem. Interesting reviews of very large-scaled neighborhoods can be found in [15] and [16].

Some recent research like the ones in [12], [17], [18], [19] and [10], develop polynomial or pseudo polynomial searches of the neighborhood space for NP-hard problems. A few exponential neighborhoods for scheduling problems have been constructed in the literature before. Angel and Bampis [20] consider a time-dependent version of the well known single-machine total weighted tardiness scheduling problem extending the work by Congram et al. [21], they develop a multi-start local search algorithm showing the superiority of dynasearch neighborhoods over traditional ones. Dynasearch, first introduced in [21], is equivalent to performing a series of 'independent' swap moves. Brueggemann and Hurink [18] consider a single-machine setting with release dates to minimize the total completion time, introducing a very large-scale neighborhood that is searched in polynomial time. Rios-Solis and Sourd [17] consider a parallel machine scheduling problem with earliness and tardiness penalties and proposed a pseudo-polynomial algorithm to explore an exponential neighborhood. Brueggemann and Hurink [22] also considers a parallel machine setting, but with the objective of minimizing the total weighted completion time, developing and evaluating very large-scale neighborhoods.

For batching machine scheduling, Hurink [23] analyzes an exponential neighborhood for a single batching machine to minimize the total weighted completion time. His model differs from ours in that the processing time of each batch is defined as the sum of the processing time of the jobs belonging to that batch. He uses a multiple transpose neighborhood within a tabu search heuristic. This is the only study, as far as we know, that develops an exponential neighborhood for a batching machine.

This paper proposes a new exponential-sized neighborhood for the problem of scheduling a batching machine, where a limit is imposed on the size of each batch, to minimize the maximum lateness of the jobs. A solution for this problem is defined by an assignment of jobs to each batch and an ordering of those batches. For example, in the previous instance with 5 jobs and a batch size limit of $b = 3$, the two schedules: $\sigma_1 = (\{1, 2, 5\}, \{4, 3\})$ and $\sigma_2 = (\{1\}, \{2, 5, 4\}, \{3\})$ are considered. An alternative representation of these solutions is to consider the underlying sequence of jobs, which in this case is the same for both schedules $\pi_{\sigma_1} = \pi_{\sigma_2} = (1, 2, 5, 4, 3)$. This approach is practical if, for a given sequence, there exist an efficient procedure to perform a batching that will minimize $L_{\max}$ while preserving the (underlying) ordering of jobs.

The main contributions of this paper are as follows. A dynamic programming formulation is derived that finds an optimal batching for a given underlying sequence of jobs (Section 3.1). Hence, we are able to define local search neighborhoods that are commonly used for sequencing problems such those given by job-insert and job-swap operators (Section 3.2). We propose an exponential neighborhood that can be viewed as a restricted version of a multiple-insert neighborhood. More precisely, a given sequence is split into two subsequences, which define the restricted multi-insert operations. We then propose an efficient dynamic programing algorithm (Section 4.4) that finds the best merging of these two subsequences into a single sequence. The set of sequences created by this procedure form the *split-merge* neighborhood.

Using descent, multi-start descent and iterated descent heuristics, a computational comparison of these neighborhoods (Section 5) shows the benefits of using the split-merge neighborhood.

The remaining sections of this paper is organized as follows. In Section 2, we present a mixed-integer linear programming formulation and discuss exact solutions for the problem. Section 3 contains a description of a job-swap and a job-insert sequence-preserving neighborhood for the problem that can be explored efficiently using dynamic programming. In Section 4, we introduce the split-merge neighborhood, analyzing its size and describing a method by which it can be explored. Section 5 reports on our computational experience, where we compare the performance of the split-merge neighborhood with the widely used job-swap and job-insert neighborhoods. We conclude in Section 6, pointing to possible further work and extensions.

2. Exact Solutions for the Batching Machine Problem

To evaluate the heuristics developed in this paper, it is useful to have a method of finding optimal solutions for a set of instances to serve as benchmarks for the problem. With this aim, we propose the following mixed-integer linear programming formulation. We use zero-one decision variables x_{jk}, for $j,\ k = 1, \ldots, n$, that are defined by

$$x_{jk} = \begin{cases} 1 & \text{if job } j \text{ is assigned to batch } \mathcal{B}_k, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, we are assuming that there are n batches, but allow some of these batches to be empty. Further, we define continuous decision variables as follows:

P_k: the processing time of batch $\mathcal{B}_k$, for $k = 1, \ldots, n$;
C_k: the completion time of batch $\mathcal{B}_k$, for $k = 1, \ldots, n$;
L: the maximum lateness of all jobs.

Also, we define the constant $M_k = \sum_{j=1}^{k} p_{\pi(j)}$, where $(\pi(1), \ldots, \pi(n))$ is a longest processing time (LPT) sequence of the jobs, i.e., $p_{\pi(1)} \geq \ldots \geq p_{\pi(n)}$. Then our MILP formulation for the batching machine problem is:

$$\begin{aligned} \text{Minimize} \quad & L \\ \text{subject to} \quad & L \geq C_k - d_j - (M_k - p_j)(1 - x_{jk}) \qquad j,\ k = 1, \ldots, n \qquad (1) \end{aligned}$$

$$\sum_{k=1}^{n} x_{jk} = 1 \qquad j = 1, \ldots, n \tag{2}$$
$$P_k \geq p_j x_{jk} \qquad j,\ k = 1, \ldots, n \tag{3}$$
$$C_1 = P_1 \tag{4}$$
$$C_k = C_{k-1} + P_k \qquad k = 2, \ldots, n \tag{5}$$
$$\sum_{j=1}^{n} x_{jk} \leq b \qquad k = 1, \ldots, n \tag{6}$$
$$x_{jk} \in \{0, 1\} \qquad j,\ k = 1, \ldots, n. \tag{7}$$

Note that when job j is contained in batch $\mathcal{B}_k$ so that $x_{jk} = 1$, then the right-hand side of constraint (1) becomes $C_k - d_j$, which is equal to the lateness of job j. Observing that the lateness L_j of job j can be bounded by

$$L_j \geq p_j - d_j \geq p_j - d_j + (C_k - M_k),$$

where the first inequality is valid because the completion time of job j in any schedule is at least p_j, and the second inequality hods because the definition of M_k ensures that it is an upper bound on the completion time of batch $\mathcal{B}_k$ for $k = 1, \ldots, n$. Thus when $x_{jk} = 0$, the right-hand side of (1) is a valid lower bound on the lateness of job j for every batch $\mathcal{B}_k$. Thus, constraints (1), together with the minimization of the objective function, ensure that L is in effect the maximum lateness. Constraint (2) requires that every job belongs to exactly one of the batches. The processing time of batch $\mathcal{B}_k$ is evaluated through constraint (3), while the batch completion times are calculated through constraints (4) and (5). Finally, constraint (6) imposes the restriction on batch size, while (7) restrict the values of the decision variables to take values consistent with their definitions.

Using the above formulation and Xpress-IVE 1.24.02, selecting the dual algorithm with parameters set as default with a two hour time limit, we were unable to find optimal solutions for some difficult instances of 20 jobs. Xpress found no noticeable improvement on the feasible solution after 20 minutes. The difference between the best solution obtained by Xpress and the optimal solution is around 26%. To obtain the optimal solution for those difficult instances we ran the branch and bound algorithm proposed in [24]. This branch and bound is not based on a LP model, but rather on bounds on partially constructed schedules, calculated through dynamic programming. Exact solutions for 25 or more jobs were difficult to find using the branch and bound algorithm proposed in [24]. It seems that even for instances with relative few jobs (i.e., around 25), finding exact solutions is computationally time consuming.

3. Sequence-Based Neighborhoods

A natural representation for a solution for our batching machine problem, is defined by a selection of jobs for each batch and an ordering of those batches. Recall the 5 job example presented in Section 1, with schedules $\sigma_1 = (\{1,2,5\},\{4,3\})$ and $\sigma_2 = (\{1\},\{2,5,4\},\{3\})$. Both schedules have the same underlying sequence of jobs $\pi = (1,2,5,4,3)$. On the other hand, since jobs within a batch are not ordered, schedule $\sigma_2 = (\{1\},\{2,5,4\},\{3\})$ is equal to $\sigma_3 = (\{1\},\{4,5,2\},\{3\})$ although the two schedules have different underlying sequences $\pi_{\sigma_2} = (1,2,5,4,3)$ and $\pi_{\sigma_3} = (1,4,5,2,3)$, respectively. Thus different sequences might lead to the same schedule. Rather than the direct approach of assigning jobs to batches and sequencing the batches, our solution approach is to search over possible underlying sequences. This approach is practical if, for a given sequence, there exists an efficient procedure to perform a batching that will minimize the maximum lateness, while preserving the (underlying) ordering of jobs. In this section, we develop a dynamic formulation that will find such an optimal batching for a given (fixed) sequence.

Potts & Kovalyov [25] show the usefulness of dynamic programming as a method for solving a variety of scheduling and batching problems. In particular, they give a characterization of optimal schedules for the case of unrestricted batch sizes (where $b \geq n$) for the batching machine model with regular scheduling criteria. If we assume that jobs are indexed according to the SPT (shortest processing time) rule so that $p_1 \leq \ldots \leq p_n$, an *SPT-batch schedule* is one in which adjacent jobs in the sequence $(1,\ldots,n)$ may be grouped to form batches. Brucker et al. [7] show that there exists a SPT-batch schedule that is optimal for the unrestricted batch sizes version of our problem of minimizing the maximum lateness. However, for the restricted version that we are aiming to solve, there is no such guarantee. Nevertheless, we can adapt their dynamic programming formulation to solve the unrestricted version for any given (fixed) sequence of jobs. Specifically, for a given sequence, we find a partition of the sequence into batches, with no more than b jobs in a batch, that will yield the minimum value for the maximum lateness among schedules having this underlying job sequence.

3.1. Optimal schedule for a given sequence

Consider a fixed sequence of jobs $\pi = (1, \dots, n)$ for which a batching is required. Let F_j be the minimum value of the maximum lateness of a schedule that contains jobs $j, \dots, n$ from π, where the processing of the first batch in the schedule starts at time zero. If a batch $\{j, \dots, k-1\}$, which has processing time $p' = \max_{j \le l \le k-1} p_l$, is inserted at the start of a schedule for jobs $k, \dots, n$, then the maximum lateness of jobs $k, \dots, n$ increases by p', while the maximum lateness for jobs $j, \dots, k-1$ is $p' - \min_{j \le l \le k-1} d_l$, which is the processing time of the batch minus its due date. Hence, we can state the dynamic programming recursion as follows:

Dynamic Programming Algorithm for Batching (DPB)

Initialize. $F_{n+1} = -\infty$.

Recursion. Compute for $j = n, n-1, \dots, 1$

$$F_j = \min_{j<k\le \min\{n+1,j+b\}} \{\max\{F_k + \max_{j\le l\le k-1} p_l, \max_{j \le l \le k-1} p_l - \min_{j\le l\le k-1} d_l\}\}. \tag{8}$$

Optimal solution. The optimal solution value is equal to F_1, and the corresponding batching can be found by backtracking.

We can rewrite the recursion equation (8) as

$$F_j = \min_{j<k\le \min\{n+1,j+b\}} \{\max_{j\le l\le k-1} p_l + \max\{F_k, -\min_{j\le l\le k-1} d_l\}\}. \tag{9}$$

Following the ideas of Wagelmans & Gerodimos [26] to improve computational efficency of the dynamic program of Brucker et al. [7], we define for any given j

$$G_k = \max_{j\le l\le k-1} p_l + \max\{F_k, -\min_{j\le l\le k-1} d_l\} \tag{10}$$

Then we obtain from (9) and (10) that

$$F_j = \min_{j<k\le \min\{n+1,j+b\}} G_k.$$

Note that G_k can be computed for $k = j+1, \dots, \min\{n+1, j+b\}$, in that order. Hence we can store the previous value of the $\max_{j\le l\le k-1} p_l$ and $\min_{j\le l\le k-1} d_l$ as k progresses through its range of values, thereby reducing the number of computations so that constant time is required for each value of k. Note that for each F_j there are at most b values of G_k to compare, and

n values of j. Hence, our dynamic programming can be implemented to run in $O(nb)$ time.

Algorithm DPB can either be used to explore job-swap and job-insert neighborhoods as defined in Section 3.2, or more generally as part of any metaheuristic that uses a sequence representation to explore the solution space of the problem.

3.2. Job-Swap and Job-Insert Neighborhoods

We now turn our attention to classical neighborhoods that can be used for solutions that are represented as sequences. One popular choices is the k-*exchange* neighborhood (where $k \geq 2$). This neighborhood is composed of all solutions that can be obtained by exchanging k elements of a given sequence. For $k = 2$, the neighborhood is usually referred to as *swap neighborhood.* For example, the sequence $(1, 4, 3, 2, 5, 6)$ is a neighbor of sequence $(1, 2, 3, 4, 5, 6)$, obtained by exchanging element 2 with element 4. Verifying local optimality for a k-exchange neighborhood requires $\Omega(n^k)$ time, where n is the total number of elements in the sequence. The more exchanges we allow, the more computationally expensive it becomes to search the neighborhood. The size of the swap neighborhood (where $k = 2$) is equal to $n(n - 1)/2$.

Another widely-used neighborhood is the *insert neighborhood.* This neighborhood comprises all solutions that can be obtained by selecting an element and inserting it in another position within the sequence. For example, sequence $(1, 5, 2, 3, 4, 6)$ is a neighbor of sequence $(1, 2, 3, 4, 5, 6)$, obtained by removing element 5 and inserting it before element 2. The size of this neighborhood is $(n - 1)^2$. A *block insert neighborhood* is when we select a set of adjacent elements (referred to as a block) and insert the block in another position within the sequence. For example, sequence $(5, 6, 1, 2, 3, 4)$ is a neighbor of sequence $(1, 2, 3, 4, 5, 6)$, obtained by removing the block $(5, 6)$ and inserting it before element 1. If we fix the block to be of size m (where $m \geq 2$), then the size of the neighborhood is $(n - m)^2 + 1$.

It is also common to define neighborhoods in which several simple moves are combined to form a compound move that is a single neighbor. For example, the *multi-insert neighborhood* is one where more than one insert move is allowed to form a neighbor. We might expect the multi-insert neighborhood to be more powerful than a simple insert neighborhood.

We now provide specific details of neighborhoods for our batching machine scheduling problem. We define a schedule σ' with underlying permutation π' to be in the *job-swap sequence-adjusting neighborhood* of schedule

σ with underlying sequence π if π' is a swap neighbor of π. The *job-insert sequence-adjusting neighborhood* is defined analogously. For brevity, we refer henceforth to job-swap and job-insert neighborhoods and neighbors.

The job-swap and job-insert neighborhoods can be searched efficiently using Algorithm DPB, which evaluates the various solutions within these neighborhoods. Recall that the swap and insert neighborhoods are of size $O(n^2)$. Hence, searching the job-swap or job-insert neighborhood of a schedule σ of our problem requires $O(n^3 b)$ time. For the particular case of the insert neighborhood the complexity can be improved to $O(n^2 b)$ using a modified dynamic programming formulation, as explained with more detail in Appendix A. Even though it might seem appealing to run this faster algorithm to search the insert neighborhood, we also show in the Appendix that its performance relative to other neighborhood does not justify its use here.

4. Split-Merge Neighborhood

In this section, we introduce our proposed split-merge neighborhood. We define the neighborhood in Section 4.1, derive its size in Section 4.2, suggest a splitting procedure in Section 4.3 and design a dynamic programming to explore it efficiently in Section 4.4.

4.1. Neighborhood definition

The general idea of the split-merge neighborhood is first to partition or split a sequence into two subsequences, and then merge these subsequences. More formally, for our batching machine problem, consider any feasible schedule σ, where π is the underlying sequence of jobs. Consider a partition of the jobs of π, into subsequences π_1 and π_2 (such that every job appears once in π_1 or π_2 and no jobs appears in both π_1 and π_2), and jobs in each of π_1 and π_2 appear in the same order as in π. We define a schedule σ' with underlying permutation π' to be in the *split-merge neighborhood* of schedule σ with underlying sequence π and split π_1 and π_2 if π' has the property that if job i precedes job j in either π_1 or π_2, then job i precedes job j in sequence π'.

To illustrate the definition of the split-merge neighborhood, consider a schedule with 6 jobs having $\pi = (1, 2, 3, 4, 5, 6)$ as its underlying job sequence. Let $\pi_1 = (2, 4, 5)$ and $\pi_2 = (1, 3, 6)$ be the split of π that is selected. Then, a feasible merging is $\pi' = (1, 3, 2, 4, 6, 5)$, which is equivalent to performing the following two insert moves in π: insert job 3 before job 2, and insert job 6 before job 5. Note that the order of jobs in π_1 and the order of jobs in π_2 is preserved in π', and that the initial sequence is a neighbor of itself. However, under this choice of π_1 and π_2, sequence $\pi'' = (1, 5, 6, 2, 3, 4)$ is not a neighbor of π as job 4 precedes job 5 in π_1, and job 3 precedes 6 in sequence π_2. On the other hand, if we had chosen $\pi_1 = (1, 2, 3, 4)$ and $\pi_2 = (5, 6)$, then this would be a valid merge operation, and π'' would be a neighbor of π. Note that π'' could be obtained from inserting block $(5, 6)$ in front of job 2. Hence, different splitting procedures for sequence π yield different restrictions on the possible resulting neighbors.

Summarizing, the split-merge neighborhood is a restricted multiple insert neighborhood, where the restrictions are given by the relative order of jobs in the split sequences and initial sequence, as explained above. The two determining factors for the effectiveness of the split-merge neighborhood are: (i) the splitting procedure; and (ii) the merging procedure. Both procedures are independent and the merge operation is well defined in the sense that for a given split of the original sequence, the merge procedure we suggest will always find the best neighbor.

In Section 4.4, we will show that given the split of the initial sequence a merged sequence and a feasible batching to minimize the maximum lateness of the resulting schedule can be obtained in polynomial time with a dynamic programming algorithm. The only remaining factor is to specify a good splitting procedure, in Section 4.3, we describe one that gave the best results.

4.2. Neighborhood Size

To calculate the size of the split-merge neighborhood, let the two subsequences created under splitting be π_1, and π_2, which contain n_1 and n_2 jobs, respectively. Any feasible merged sequence π' can be created by selecting n_1 positions in π' for the jobs of $\pi_1$to be placed, and then considering the jobs of π_2 sequentially, inserting the next job of π_2 in the first unfilled position in π'. This shows that the neighborhood size is $\binom{n}{n_1} = \binom{n}{n_2}$.

Note that the neighborhood size $\binom{n}{n_1}$ is increasing in n_1 for $n_1 < n/2$ and decreasing in n_1 for $n_1 > n/2$. Therefore, the maximum value of $\binom{n}{n_1}$ occurs when $n_1 = n_2 = n/2$ (for even n). Using Stirling's approximation (for example, see [27]), the corresponding neighborhood size is:

$$\binom{n}{n/2} \approx \frac{n^n\sqrt{2\pi n}\ e^{-n}}{\left(\frac{n}{2}\right)^n \pi n\ e^{-n}} = \frac{2^{n+1/2}}{\sqrt{\pi n}},$$

which shows that the size of the largest neighborhood under split merge is exponential in n. Neighborhoods of such a large size can be explored in polynomial time, as explained in Section 4.4. Furthermore, as larger neighborhoods tend to yield better quality solutions (better local optima), we are interested in using similarly sized split sequences to have as big a neighborhood as possible.

4.3. Splitting Procedure

Note that the split sequences can be of un-equal size; if we have a sequence, say π_1, with just one job, then the split-merge neighborhood is equivalent to a simple insert neighborhood where we only allow the job in sequence π_1 to be moved. In Section 4.2, we showed that sequences with (near) equal number of jobs are the best choice in terms of neighborhood size.

We have evaluated different splitting procedures, both deterministic and random. In initial experiments, the best results are obtained with a splitting that divides the sequence of jobs π randomly into τ parts, where τ is an input parameter that is found through computational testing. A formal statement of this procedure is given below.

Splitting Procedure (SP)

```
Initialize: $t_0 = 0$, $i = 0$ and $s = 1$.
Repeat:
    Generate $t \sim U[1, (\frac{2n}{\tau}) - 1]$.
    Let $i = i + 1$ and $t_i = \min\{t_{i-1} + t, n\}$.
    Add jobs $\pi(t_{i-1} + 1), \ldots, \pi(t_i)$ to the end of sequence $\pi_s$.
    If $s$ is even $s = 1$, else $s = 2$.
Until $t_i = n$
```

The aim of this splitting procedure is to divide sequence π into τ parts of similar size on average, where the i-th part is assigned to π_1 if i is odd, and to π_2 if it is even. Note that the expected value of t is n/τ, and hence we are assigning blocks of n/τ jobs, on average, to each of the subsequences π_1 and π_2, until all jobs of π are distributed between the two subsequences.

4.4. Merging Sequences

We propose the following backward dynamic programing recursion to find a best merged sequence from two subsequences, and form batches of jobs with the minimum value of the maximum lateness.

Let $\pi_1(1), \ldots, \pi_1(n_1)$ denote the n_1 jobs in the first subsequence, and $\pi_2(1), \ldots, \pi_2(n_2)$ the n_2 jobs in the second sequence. Let σ_{j_1,j_2} be a partial schedule with jobs $\pi_1(j_1), \ldots, \pi_1(n_1)$ from the first subsequence and jobs $\pi_2(j_2), \ldots, \pi_2(n_2)$ from the second subsequence, merged in such a way that the maximum lateness is minimized, and the relative order of the jobs in each subsequence is preserved, as explained above. Let L_{j_1,j_2} be the value of this maximum lateness. Then our dynamic program is as follows.

Dynamic Programming Algorithm for Merging and Batching (DPMB)

Initialize. $L_{n_1+1,n_2+1} = -\infty$.

Recursion. Compute for $j_1 = n_1, n_1 - 1, \ldots, 1$ and $j_2 = n_2, n_2 - 1, \ldots, 1$

$$L_{j_1,j_2} = \min_{k_1,k_2}\{\max\{L_{k_1,k_2} + \max_{k\in K} p_k, \max_{k\in K} p_k - \min_{k\in K} d_k\},$$

where the outer minimization over k_1 and k_2 is subject to: $j_1 \le k_1 \le n_1 + 1$, $j_2 \le k_2 \le n_2 + 1$ and $1 \le k_1 - j_1 + k_2 - j_2 \le b$; where $K = \{\pi_1(j_1), \ldots, \pi_1(k_1 - 1), \pi_2(j_2), \ldots, \pi_2(k_2 - 1)\}$.

Optimal solution. The optimal solution value is equal to $L_{1,1}$, and the corresponding batching and merged sequence can be found by backtracking.

In the recursion for Algorithm DPMB,jobs of set $K = \{\pi_1(j_1), \ldots, \pi_1(k_1 - 1), \pi_2(j_2), \ldots, \pi_2(k_2 - 1)\}$ form a batch containing the union of the jobs occupying positions $j_1, \ldots, k_1 - 1$ in π_1 and positions $j_2, \ldots, k_2 - 1$ in π_2. This batch is inserted at the beginning of the schedule σ_{j_1,j_2}. The minimum and maximum batch size restriction are implemented through the condition $1 \le k_1 - j_1 + k_2 - j_2 \le b$.

We conclude this section by discussing the time complexity of Algorithm DPMB. The recursion equation is applied for fewer than n^2 values of j_1 and j_2. Further, for each pair of values j_1 and j_2, there are fewer than b^2 sets K to produce the first batch. As in Section 3.1, the values $\max_{k\in K} p_k$ and $\min_{k\in K} d_k$ can be updated from previous values as k_1 and k_2 progress through their range. This implies that Algorithm DPMB requires $O(b^2n^2)$ time.

5. Computational Experience

In this section, we provide a computational evaluation of the usefulness of our new split-merge neighborhood. Section 5.1 describes how the test problems that we use are generated and gives details of the computing environment. Section 5.2 evaluates the performance of the split-merge versus job-insert and job-swap neighborhoods with a simple descent heuristic.

In Section 5.3, we evaluate the different neighborhoods within multi-start descent. Finally, Section 5.4 designs an iterated descent heuristic for the split-merge neighborhood and evaluates its performance on larger instances.

5.1. Test Problems and Experimental Design

We have used a set of data generated for instances with 20 jobs, similar to the ones used in [24], to compare the different local search heuristics we propose. The processing times of the jobs are integers uniformly distributed from 1 to 100. An estimate on the makespan of the schedule, given the processing times and the batch size b, is $T = \frac{1}{b}\sum_{j=1}^{n} p_j$. An obvious choice for generating the due dates is from a uniform distribution with values between 1 and T. However, to mimic different scenarios for the due dates, we have used a parameter λ that is assigned values $\lambda = 0.5, \lambda = 1.0$ and $\lambda = 1.5$, and then the due dates are integers uniformly distributed between 1 and λT. Hence, $\lambda = 0.5$ corresponds to a tight due date scenario, and $\lambda = 1.5$ is a slack scenario. For the 20-job set of instances, the maximum batch sizes are chosen as $b = 2,\ 3$ or 4, and we have 10 instances for each choice of b and λ.

We also generate new sets of data for 50 and 100 jobs to test an iterated descent algorithm incorporating the split-merge neighborhood that we propose in Section 4. The processing times and due dates of the jobs for this set of instances follow the same distributions as described above. For $n = 50$, the maximum maximum batch sizes are $b = 2$, $b = 5$, $b = 10$ and $b = 25$, and for $n = 100$, we select $b = 5$, $b = 10$, $b = 15$, $b = 25$ and $b = 50$. We generated 10 instances for each choice of b and λ. Thus, there are 120 and 150 instances with $n = 50$ and $n = 100$, respectively. All our heuristics were coded in C and the instances are run on a MacBook Pro with an Intel® Core$^{\text{TM}}$ i7 processor at 2.8 Ghz. The operating system is Mac OS X Lion 10.7.5.

For the various local search heuristics that are evaluated in this section, we use various performance measures. First, for $n = 20$, (global) optimal solutions are known from the branch and bound algorithm proposed by Possani [24]. Thus, we compute the following statistics.

- DV: average percentage deviation from the optimal solution value,
- NO: number of optimal solutions found,
- AT: average computation time in seconds over all instances considered.

The percentage of deviation from the optimal solution value is computed using

$$\Big(\frac{L_{\max}(\pi_{\text{LO}}) - L_{\max}(\pi_{\text{GO}})}{|L_{\max}(\pi_{\text{GO}})|}\Big)100,$$

where π_{LO} is the schedule corresponding to the local optimum obtained from the local search heuristic under consideration and π_{GO} is global optimum schedule. Note that this measure of deviation is not well defined when $L_{\max}(\pi_{\text{GO}})$ is zero, however, this is not the case in any of our instances.

5.2. Neighborhood comparison within descent

In this section, we compare the split-merge neighborhood with the swap and insert neighborhoods introduced in Section 3.2. These neighborhoods are used within a classical descent heuristic, either with a first improve or a best improve policy for selecting moves. A first-improve descent heuristic executes a move when the first improving neighbor is found, whereas a best-improve descent heuristic will only execute a move after the complete neighborhood has been searched and a best move has been found. Both descent heuristics stop once there is no improvement on the previous move. A local search in the split-merge neighborhood can be viewed as a best-improve descent, where the restricted moves are specified by the two subsequences and the merging operation finds the best neighbor. Based on these ideas we propose the following descent heuristic for the split-merge neighborhood:

```
Descent Heuristic in split-merge neighborhood
begin
    - Construct an initial sequence $\pi'$ where jobs are in EDD order.
    - Obtain an optimum batching for sequence $\pi'$ with Algorithm DPB of
      Section 3.1 and evaluate the maximum lateness $L_{\max}(\pi')$.
repeat
    - Set $\pi = \pi'$ and $L_{\max}(\pi) = L_{\max}(\pi')$
    - Split $\pi$ into two subsequences $\pi_1$ and $\pi_2$ using procedure SP as described
      in Section 4.3.
    - Merge subsequences $\pi_1$ and $\pi_2$ with Algorithm DPMB of Section 4.4
      to obtain a new sequence $\pi'$, and an optimal batching with maximum
      lateness $L_{\max}(\pi')$.
```

```
until Stopping criterion is met.
end
```

Recall that from Section 4 for every new sequence π' obtained by the DPMB, π' is a neighbor of π and thus $L_{\max}(\pi') \leq L_{\max}(\pi)$. The case $L_{\max}(\pi) = L_{\max}(\pi')$ corresponds to a *neutral* move where as $L_{\max}(\pi) < L_{\max}(\pi')$ yields a *descent* move. The standard stopping criterion would be $L_{\max}(\pi) = L_{\max}(\pi')$. Initial experiments with this criterion found that a move within the repeat-until loop was performed only once or twice. We interpret this as indicating that the split-merge neighborhood has a flat landscape, thus making it hard to escape from the first local optimum. To avoid this, we substitute the termination criterion with a criterion to stop after a certain amount of computation time l. This time is calculated as the average time it takes to perform a best-improve insert descent, since the split-merge neighborhood can be viewed as a restricted multi-insert best-improve neighborhood.

In this section, we compare five descent heuristics for the different neighborhoods. The first four have $L_{\max}(\pi) = L_{\max}(\pi')$ as stopping criterion, while the last one uses the time stopping criterion as explained above:

I-FI: Insert neighborhood within first-improve descent;
I-BI: Insert neighborhood within best-improve descent;
S-FI: Swap neighborhood within first-improve descent;
S-BI: Swap neighborhood within best-improve descent;
SM: Split-merge neighborhood.

The results for the computational experiments using the set of 20-job instances introduced in Section 5.1 are displayed in Table 1. These results are aggregated over the different maximum batch sizes (i.e., b = 2, 3 and 4). Thus, the main entries in Table 1 represent the performance over 30 instances. Further, the overall results combine the three different due date scenarios to give the performance over 90 instances. In this table, because of the small computation times, we have listed average computation times in microseconds in the columns labeled AT'.

A first-improve descent heuristic executes a move when the first improving neighbor is found, whereas a best-improve descent heuristic will only execute a move after the complete neighborhood has been searched. Thus, we might expect a first-improve descent heuristic to take less time than a best-improve descent heuristic. This behavior is clearly confirmed in Table 1,

Table 1: Comparison of descent heuristics for 20-job instances.

		I-FI			I-BI		
Neighborhood	λ	DV	NO	AT'	DV	NO	AT'
Insert	1.5	39.52	23	614	36.87	25	989
	1	13.90	3	1233	12.65	5	1817
	0.5	16.31	1	959	10.78	1	1952
	Overall	23.24	27	935	20.10	31	1600
		S-FI			S-BI		
	λ	DV	NO	AT'	DV	NO	AT'
Swap	1.5	53.26	24	1101	6.85	26	610
	1	10.96	7	749	7.55	4	1083
	0.5	6.34	4	924	5.34	7	1328
	Overall	23.18	35	925	6.58	37	1007
		SM					
	λ	DV	NO	AT'			
Split-Merge	1.5	4.74	26	1600			
	1	6.51	9	1600			
	0.5	6.13	5	1600			
	Overall	5.79	40	1600			

where we observe that the best-improve descent requires more computation time. However, solution quality is better under best-improve. Specifically average percentage deviations from the global optimum solution values are smaller under the best-improve strategy. This behavior is more apparent in the swap neighborhood, where the first-improve descent has an average deviation of 23.18%, whereas best-improve yields a 6.58% average deviation. We notice that, in most cases, it is harder to find global optima for those instances with more restrictive due dates.

The split-merge neighborhood yields the best performance overall. It finds more global optima, and it exhibits a smaller average deviation from the global optimum solution value than the other descent heuristics. Furthermore, the average deviations for heuristic SM from the global optimum do not depend significantly on whether or not the due dates are restrictive, as they do with the other heuristics.

In spite of the strong relative performance of heuristic SM, 40 out of the 90 global optima are found, which is only a 44% success rate in finding the global optima. Thus, it is of interest to explore other local search heuristics. An easy option is to re-start the descent from a new solution once the search has stopped. This can be done in two ways, one is through a multi-start descent scheme and the other is an iterated approach. In the following subsections, both are analyzed.

5.3. Neighborhood comparisons within multi-start

A multi-start descent heuristic is one in which a descent procedure is applied from different starting solutions, and the best of the local minima is selected as the heuristic solution. A recent survey on multi-start methods is provided by Martí et al. [28]. The idea is to diversify the search of the solution space in the hope of finding a better solution by starting in different points rather than just a simple descent from a single initial point. In this section, we compare the neighborhoods of interest within a multi-start setting.

Recall from Section 5.2 that best-improve descent heuristic yields better results than those for first-improve descent, both in terms of average percentage deviations from the global optima and the number of global optima found. Hence, we restrict our attention henceforth to best-improve when considering descent heuristics. Regarding the termination time l for the split-merge neighborhood search we use the same time as in Section 5.2. Thus, in this subsection, we compare the following three multi-start heuristics:

I-MBI: Insert neighborhood within multi-start best-improve descent;
S-MBI: Swap neighborhood within multi-start best-improve descent;
SM-M: Split-merge neighborhood within multi-start descent.

In our multi-start heuristics, we generate 10 random starting solutions in the form of sequences for use in I-MBI, S-MBI and SM-M. Each random starting solution yields a local optimum, and the best among the 10 local optima is then selected as the overall solution for the relevant multi-start heuristic. As in the previous subsections, we use the same set of 20-job instances, and aggregate the results over the different maximum batch sizes $b = 2$, 3 and 4. Thus, each of our presented results is the average over 30 different 20-job instances. The overall behavior over all 90 instances is also presented.

Table 2: Comparison of multi-start heuristics for 20-job instances

	I-MBI				S-MBI				SM-M			
	DV	NO	NTW	AT	DV	NO	NTW	AT	DV	NO	NTW	AT
$\lambda = 1.5$	18.03	17	18	0.048	11.58	17	17	0.023	2.36	28	29	0.016
$\lambda = 1$	8.66	5	5	0.047	7.79	5	6	0.023	3.59	12	21	0.016
$\lambda = 0.5$	5.27	2	3	0.046	2.74	7	9	0.022	2.84	9	18	0.016
Overal	10.65	24	26	0.047	7.37	29	32	0.023	2.93	49	68	0.016

The results for our multi-start descent heuristics are listed in Table 2. In addition to the performance statistics DV, NO and AT used previously, we also include:

- NTW: number of times the winning heuristic. In this measure, a count is made of the number of times the heuristic finds a solution no worse than that found with either of the other two heuristics.

For our multi-start descent heuristics, we use randomly constructed sequences to create starting solutions instead of the EDD ordered sequence used in Section 5.2. We first discuss the insert neighborhood, and specifically the performance of heuristic I-MBI. A good initial solution seems to be of importance for the insert neighborhood, as observed by comparing with the results given in Tables 1 and 2. The overall solution quality for the insert neighborhood is arguably better with EDD, even though in multi-start we are running descent for each of the 10 starting solutions. This conclusion is reached from the 31 global optima found using the EDD sequence compared with the 24 global optima resulting from the 10 randomly generated starting solutions. We also note the much larger total computation times with the multi-start descent heuristic (4701 microseconds compared with 1600 microseconds for heuristic I-BI). Further, heuristic I-MBI finds the ‘winning’ solution in only 26 of the 90 instances, compared to 32 for S-MBI and 68 for SM-M.

We now assess the performance of the swap neighborhood in multi-start best-improve descent through the results obtained for heuristic S-MBI. A comparison of results for heuristics I-MBI and S-MBI shows that, for each of the performance measures NO, DV, NTW and AT, S-MBI exhibits better performance than I-MBI. Our intuition for these results is that a swap move allows more batches to remain intact than an insert move. In this way, many more features of a solution can be retained within the swap neighborhood than is the case under insert. The starting solution also seems to play an important role for the swap neighborhood within multi-start descent, since S-BI with its EDD starting solution produces better overall performance measures than S-MBI with its random starting sequences. In spite the preference for swap over insert, the performance measures for S-MBI are significantly worse than for the multi-start heuristic with the split-merge neighborhood SM-M.

As implied above, the best multi-start descent heuristic is SM-M which uses the split-merge neighborhood. It finds about 54% of the global optima (49 out of 90), and found the ‘winning’ solutions in about 75% of the instances (68 out of 90). However, since it did not find the global optima for all the 20-job instances, it is of interest to see whether we can achieve better results by retaining some of the features of the previously obtained local optimum. We develop this idea by exploring an iterated descent heuristic in the next subsection.

5.4. Neighborhood comparisons within iterated descent

An iterated local search heuristic also uses multiple applications of a local search, but rather than starting from random solutions, it restarts from a modification of the previous local optimum. In the literature, this process of modification is frequently achieved by means of a *kick*. The key idea is to dislodge the local search from its current locality within the solution space with a view to exploring different areas.

A common method of performing a kick is to execute a series of random neighborhood moves using the same neighborhood as for the local search. As the split-merge neighborhood is a restricted version of the multi-insert neighborhood, we have chosen a kick to be a certain number of randomly selected insert moves. Equivalently, a kick for the swap neighborhood consists of a series of random swap moves. The number of moves is referred to as the *size of the kick*, κ. We want the kick to be sufficiently large to move to a solution that is not too close to the previous local optimum so that a return to this local optimum is avoided, but not so far away that the good characteristics of the previous local optimum are lost and we effectively have a multi-start procedure. We do not consider the insert neighborhood as it gave poor results as seen in previous sections, and the split-merge neighborhood is a restricted version of a multi-insert neighborhood.

We performed some initial experiments with the 20-job instances to determine appropriate values of κ; the values tried were: $2, 3, 4, 5$ and 6. To make a fair comparison with multi-start, our iterated descent algorithms each have 10 iterations, where each new iteration starts with a kick and ends with a local optima. The best choices for the parameter value κ in our initial experiments are obtained with $\kappa = 3$, for the swap neighborhood, and $\kappa = 2$ for the split-merge.

Table 3 shows the results obtained with the swap neighborhood. We compare multi-start with iterated descent starting from a randomly generated initial sequence (Iterated RS) and also starting with a sequence in EDD order (Iterated EDD).

Table 3: Comparison of heuristics using Swap Best Improve for 20-job instances

	MultiStart			Iterated RS			Iterated EDD		
	DV	NO	AT	DV	NO	AT	DV	NO	AT
$\lambda = 1.5$	11.58	17	0.0231	16.50	16	0.0098	1.97	27	0.0096
$\lambda = 1$	7.79	5	0.0226	9.54	10	0.0111	4.92	8	0.0098
$\lambda = 0.5$	2.74	7	0.0217	3.87	6	0.0099	2.50	13	0.0094
Overal	7.37	29	0.0225	9.97	32	0.0103	3.13	48	0.0096

Using an iterated local search seems to improve the total number of optimal solutions that are found. We also notice that the average time was reduced by more than half. As expected, the best results are obtained when the initial sequence has jobs in EDD order, since more optima were found in less time and with smaller deviations.

For the split-merge neighborhood we allow each descent to run for the same time as allowed for multi-start. Table 4 compares results for multi-start, iterated descent starting on a random initial sequence (Iterated RS) and iterated descent starting from a sequence containing jobs in EDD order (Iterated EDD)

Table 4: Comparison of heuristics using Split-Merge for 20-job instances

	MultiStart			Iterated RS			Iterated EDD		
	DV	NO	AT	DV	NO	AT	DV	NO	AT
$\lambda = 1.5$	2.36	28	0.0160	1.30	28	0.0160	1.03	28	0.0160
$\lambda = 1$	3.59	12	0.0160	2.36	18	0.0160	2.45	18	0.0160
$\lambda = 0.5$	2.84	9	0.0160	3.04	11	0.0160	1.89	14	0.0160
Overal	2.93	49	0.0160	2.24	57	0.0160	1.79	60	0.0160

AT: Average Time in seconds

Again, the iterated descent finds more global optima than multi-start, and produces solutions of superior quality compared to those obtained using the swap neighborhood. It is interesting to notice that the benefits of starting with a EDD sequence as compared to a random sequence are not as significant as when using the swap neighborhood. Hence, the split merge neighborhood is less dependent on the starting solution.

In an attempt to obtain the 90 global optima for this set of instances, we allow split merge to run for a longer period of time, using more kicks in the search. Using the same time for the local search as in Table 4, we obtained the 90 global optima when starting with a random initial sequence, with 350 kicks and a running time of 0.56 seconds for each instance. If the local search starts with an EDD sequence, then to find the 90 optima we need 300 kicks and only 0.48 seconds.

5.5. Neighborhood comparisons for bigger instances

We now present and discuss our results for the larger instances with 50 and 100 jobs, that are generated using the method explained in Section 5.1. For these instances, we do not have global optimal solutions, but instead can evaluate the heuristic solutions relative to a lower bound. The lower bound is that of the SPT-EDD-dynamic batch schedule as proposed in Possani [24]. This lower bound in based on a relaxation of the fact that each job is characterized by its processing time p_j and its due date d_j. In the

SPT-EDD-dynamic batch schedule the restriction on the batch size is not violated, but the p_j and d_j of job j may be disassociated, allowing consecutive batches one to contain the p_j and the other the d_j. Let π_{LO} be the schedule corresponding to the local optimum obtained as a result of the local search heuristic. Then, we consider specific performance measures as follows:

- DLB, which refers to the average percentage relative deviation of the heuristic solution value from a lower bound. For a single instance, the deviation is computed using

$$\text{DLB} = \frac{L_{\max}(\pi_{\text{LO}}) - \text{LB}}{\text{LB}} 100,$$

  where LB is the lower bound given by SPT-EDD-dynamic batch schedule.

- NO, number of optima found, based on the lower bound. Thus, if DLB $=$ 0, the local optimum is a global optimum.

- IEED, which refers to the average percentage improvement on the value of the optimal schedule for the EDD sequence over the instances considered. Let π_{EDD} denote the optimal schedule obtained from the EDD sequence. Then, the improvement on the EDD sequence value for a single instance is defined by

$$\text{IEED} = \frac{L_{\max}(\pi_{\text{EDD}}) - L_{\max}(\pi_{\text{LO}})}{L_{\max}(\pi_{\text{EDD}})} 100.$$

Our results of running an iterated descent heuristic on the split-merge neighborhood starting on a random sequence for the 50-job instances are summarized in Table 5, where entries IEDD, and DLB indicate the average over 10 instances, NO is the total number of optima found. The first parameter we need to fix is the running time for each descent move within the split-merge neighborhood. We set this time to be equal to the average computation time it takes a descent on the best-improve insert neighborhood starting in a random sequence (similar to how we set it in Section 5.2) which for 50 jobs is 0.24 seconds. The second parameter to set is the number of kicks, and finally the size of the kick. We did experiments for size of kick $\kappa = 6, 8, 10, 12$ and 16, and number of kicks: 50, 100 and 150. The best results were obtained with $\kappa = 10$ and 100 kicks. This means that after each descent move, 10 random insert moves are performed on the last best solution to obtain a new starting point for a new descent; 100 descent moves are performed in total.

Table 5: Results for the 50-job instances

	$\lambda = 1.5$			$\lambda = 1.0$			$\lambda = 0.5$		
b	IEDD	DLB	NO	IEDD	DLB	NO	IEDD	DLB	NO
2	23.4	63.8	4	53.7	502.7	0	26.8	854.5	0
5	44.4	22.1	3	52.2	120.3	0	39.6	304.0	0
10	32.7	134.8	1	41.4	60.0	0	35.6	143.2	0
25	9.6	12.6	3	9.6	30.3	0	12.9	49.9	0

With these parameters we were able to obtain the most optima, in this case 11, as seen in Table 5. For 100-job instances, we also ran the iterated split-merge heuristic, with $\kappa = 10$ and 100 kicks. We change the computation time for descent accordingly, so that it is run for 5.08 seconds. On average for each instance the heuristic took 8.4 minutes, and found 9 optima, as shown in Table 6.

Table 6: Results for 100-job instances

	$\lambda = 1.5$			$\lambda = 1.0$			$\lambda = 0.5$		
b	IEDD	DLB	NO	IEDD	DLB	NO	IEDD	DLB	NO
5	60.9	43.2	2	62.1	203.3	0	42.9	604.8	0
10	55.7	9.4	5	53.1	126.8	0	43.9	299.5	0
15	47.9	14.5	2	46.8	96.9	0	41.9	197.3	0
25	25.0	19.0	0	31.5	60.6	0	31.3	117.9	0
50	7.5	15.5	0	8.5	39.2	0	14.9	48.9	0

It is interesting to notice that even though we are starting on a random sequence the heuristic always improves on the value of the EDD sequence. Improvements range from 9.6 % to 53.7 % for the 50-job instances, and from 7.5% to 62.1 % for the 100-job instances. In general, the improvement was smaller as the batch size b increases. For both the 50- and 100-job instances, global optima are only found for some instances with slack due dates ($\lambda = 1.5$), which suggests that such instances might be easier. It is difficult to provide meaningful comments about solution quality relative to the optimum because the results listed under DLB and NO depend on the quality of the lower bound. However, we can notice that as the batch size increases the deviation from the lower bound decreases, as does the improvement on the EDD sequence, meaning that the gap between the value for an EDD sequence and the lower bound decreases.

6. Concluding remarks

This paper introduces a new neighborhood, split-merge, to solve the problem of scheduling jobs on a batching machine to minimize the maximum lateness of the jobs, where there is a limit on the size of a batch. For this problem, solutions can be represented as sequences of jobs since in this paper we develop a dynamic program (DPB), that optimally partitions the sequence into batches. A neighbor in split-merge is formed by splitting the sequence into two subsequences, and then using another dynamic program (DPMB) to merge the subsequences to form a new sequence. This is equivalent to exploring a restricted multi-insert neighborhood in one move. A key property is that the split-merge neighborhood is exponential in size, but can be searched in polynomial time.

Computational experiments with simple descent, multi-start descent and iterated descent heuristics that use the swap, insert and split-merge neighborhoods show that the proposed split-merge neighborhood is preferred. It appears that the ability of the split-merge neighborhood to explore diverse areas of the solution space is a major factor contributing to the high-quality solutions that it generates.

Future research will explore the potential of the split-merge neighborhood for producing competitive local search algorithms for other problems. Different batching machine problems could be investigated to include the number of late jobs on a single batching machine, or multi-objective functions.

Appendix A.

In this Appendix, we show that the best insert neighbor can be found by dynamic programming in $O(n^2 b)$ time. Let $\pi = (1, \dots, n)$ be the current sequence of jobs, and let $\pi^i = (\pi^i(1), \dots, \pi^i(n')) = (1, \dots, i-1, i+1, \dots, n)$ be the resulting sequence that is obtained when job i is removed from the current sequences, where $n' = n - 1$. Our aim is to provide a dynamic program that finds the best position to insert job i into π^i and produces an optimal batching that is associated with this new sequence.

Let F_j be the minimum value of the maximum lateness of a schedule that contains jobs $\pi^i(j), \ldots, \pi^i(n')$ where the processing of the first batch in the schedule starts at time zero. Further, let F'_j be the minimum value of the maximum lateness of a schedule that contains jobs $i, \pi^i(j), \ldots, \pi^i(n')$, where the processing of the first batch in the schedule starts at time zero, and job i is inserted in its optimal position in the sequence $(\pi^i(j), \ldots, \pi^i(n'))$ (which includes occupying the first or last position).

The values of F_j are computed recursively in the same way as for Algorithm DPB. For the computation of F'_j, either job i appears in the first batch of the schedule that contains jobs $\{i, j, \ldots, k-1\}$ for some $k \in \{j, \ldots, \min\{j+b-1, n'+1\}\}$, while the remaining part of the schedule is the batching implicit in the computation of F_k, or job i is not in the first batch of the schedule that contains jobs $\{j, \ldots, k-1\}$ for some $k \in \{j+1, \ldots, \min\{j+b, n'+1\}\}$, while the remaining part of the schedule including job i is the batching implicit in the computation of F'_k. Thus, we have the following dynamic programming algorithm.

Dynamic Programming Algorithm for Insertion and Batching (DPIn$_i$B)

Initialize. $F_{n'+1} = -\infty$ and $F'_{n'+1} = p_i - d_i$

Recursion. Compute for $j = n', n'-1, \ldots, 1$

$$F_j = \min_{j<k\le\min\{n'+1,j+b\}} \{\max\{F_k + \max_{l\in I_{jk}} p_l, \max_{l\in I_{jk}} p_l - \min_{l\in I_{jk}} d_l\}\}. \tag{A.1}$$

$$F'_j = \min\{\min_{j\le k\le\min\{n'+1,j+b-1\}} G_k, \min_{j<k\le\min\{n'+1,j+b\}} G'_k\}, \tag{A.2}$$

where

$$G_k = \max\{F_k + \max_{l\in I_{jk}\cup\{i\}} p_l, \max_{l\in I_{jk}\cup\{i\}} p_l - \min_{l\in I_{jk}\cup\{i\}} d_l\},$$

and

$$G'_k = \max\{F'_k + \max_{l\in I_{jk}} p_l, \max_{l\in I_{jk}} p_l - \min_{l\in I_{jk}} d_l\},$$

where $I_{jk} = \{\pi^i(j), \ldots, \pi^i(k-1)\}$.

Optimal solution. The optimal solution value is equal to F'_1, and the corresponding insertion position of job i and batching can be found by backtracking.

We can rewrite G_k and G'_k as

$$G_k = \max_{l \in I_{jk} \cup \{i\}} p_l + \max\{F_k, -\min_{l \in I_{jk} \cup \{i\}} d_l\}\}, \tag{A.3}$$

$$G'_k = \max_{l \in I_{jk}} p_l + \max\{F'_k, -\min_{l \in I_{jk}} d_l\}\}, \tag{A.4}$$

Based on the method described in Section 3.1, G_k and G'_k can be computed for $k = j, \dots, \min\{n' + 1, j + b - 1\}$ and $k = j + 1, \dots, \min\{n' + 1, j + b\}$ respectively in that order, in $O(b)$ time. This each F_j and F'_j are computed in $O(b)$ time, and the desired value of F'_1 is computed in $O(nb)$ time. Algorithm DPIn$_i$B is applied for $i = 1, \dots, n$ so that each job is considered for insertion, given an overall time complexity of $O(n^2 b)$ to find the best insert neighbor.

In Table A.7 we show a comparison between the standard $O(n^3 b)$ exploration of the insert neighborhood versus the $O(n^2 b)$ using DPIn$_i$B. The same test instances as explained in Section 5.1 are used. The first set of columns show the results for the standard implementation of first-improve (I-FI) and best-improve (I-BI) neighborhood, and the second set of columns for first and best improve using the above DPIn$_i$B algorithm, (I-FDPInB) and (I-BDPinB) respectively.

Table A.7: Comparison of Insert heuristics for 20-job instances

	I-FI			I-FDPInB		
	DV	NO	μ-sec	DV	NO	μ-sec
$\lambda = 1.5$	39.52	23	614	20.85	7	121
$\lambda = 1$	13.90	3	1233	63.09	5	96
$\lambda = 0.5$	16.31	1	959	20.73	7	109
Overall	23.24	27	935	34.89	19	109
	I-BI			I-BDPInB		
	DV	NO	μ-sec	DV	NO	μ-sec
$\lambda = 1.5$	36.81	25	989	10.00	11	265
$\lambda = 1$	12.65	5	1817	43.30	6	335
$\lambda = 0.5$	10.78	1	1952	8.92	10	455
Overall	20.10	31	1600	20.74	27	351

Figures under DV show the average percentage of deviation from the optimal solution value, under NO the number of optimal solutions found, and under μ-sec the average computational time in micro-seconds over all instances considered. As expected, the running times are reduced by employing DPIn$_i$B; first-improve by a factor of 9, and for the best-improve by a factor of 4.5. However, DPIn$_i$B found less number of optima in both cases. Further analysis shows that in some instances the DPIn$_i$B search found better solutions, but it was not consistently better overall. The differences are due to existence of multiple local optima, and the order in which the neighborhood is explored.

In Table A.8 we compare the DPIn$_i$B under multi-start descent scheme. The first set of columns (I-MBI) repeat the results of experiments done in Section 5.3 for the best improve descent with 10 randomly generated starting points. The second set of columns (I-MDPInB) show the results of exploring the neighborhood using DPIn$_i$B with 10 randomly generated starting points. Figures under NTW count the number of times the heuristic finds a solution no worse than that found with either of the other two, and the figures under sec the average time overall instances. Again the computation time is reduced, in this case by a factor of 5. Based on this, we increase the number of random start by the same factor (50 random starts), and the results are shown in the third set of columns.

As expected, starting from 50 different point improves the number of optima found in comparison with I-MBI using 10 starting points. However, when comparing with S-MBI from Table 2, we observe that the same numbers of optima are obtained but the swap neighborhood is twice as fast. Hence, it seems not to be worthwhile using DPIn$_i$B for further comparisons.

Table A.8: Comparison of multi-start heuristics for 20-job instances

	I-MBI 10 starts				I-MDPInB 10 starts				I-MDPInB 50 starts			
	DV	NO	NTW	sec	DV	NO	NTW	sec	DV	NO	NTW	sec
$\lambda = 1.5$	18.03	17	17	0.048	9.52	9	10	0.0060	2.942	12	22	0.030
$\lambda = 1$	8.66	5	14	0.047	18.9	6	9	0.0080	11.670	8	22	0.042
$\lambda = 0.5$	5.27	2	17	0.046	10.13	8	10	0.0104	5.439	9	18	0.051
Overall	10.65	24	47	0.047	12.85	23	29	0.0081	6.684	29	62	0.041

Acknowledgements

Comments of an Associate Editor and anonymous referees have improved the presentation of this paper. The ideas behind the dynamic program presented in the Appendix were suggested by one of the referees. Useful discussions with Jonathan Whitehead are gratefully acknowledged. The authors used Xpress-MP in this research, which was made available through FICO's Academic Partnership Program. Marta Cabo and Edgar Possani would like to thank Asociación Mexicana de Cultura A.C. for their support. Edgar Possani would like to acknowledge support by CONACyT through their program "Estancia Sabática en el Extranjero".

[1] C.-Y. Lee, R. Uzsoy, L. A. Martin-Vega, Efficient algorithms for scheduling semiconductor burn-in operations, Operations Research 40 (4) (1992) 764–775.

[2] L. Mönch, J. Fowler, S. Dauzère-Pérès, S.J.Mason, O. Rose, A survey of problems, solution techniques, and future challenges in scheduling semiconductor manufacturing operations, Journal of Scheduling 14 (2011) 583–599.

[3] C.-S. Wang, R. Uzsoy, A genetic algorithm to minimize lateness on a batch processing machine, Computers and Operations Research 29 (2002) 1621–1640.

[4] A. H. Kashan, B. Karimi, F. Jolai, Effective hybrid genetic algorithm for minimizing makespan on a single-batch-processing machine with non-identical job sizes, International Journal of Production Research 44 (12) (2006) 2337–2360.

[5] S. Melouk, P. Damodaran, P. Y. Chang, Minimizing makespan for single machine batch processing with non-identical job sizes using simulated annealing, International Journal of Production Economics 87 (2004) 141–147.

[6] H.-W. Wang, Solving single batch-processing machine problems using an iterated heuristic, International Journal of Production Research 49 (14) (2011) 4245–4261.

[7] P. Brucker, A. Gladky, H. Hoogeveen, M. Y. Kovalyov, C. N. Potts, T. Tautenhahn, S. L. V. de Velde, Scheduling an batching machine, Journal of Scheduling 1 (1998) 31–54.

[8] M. Yagiura, S. Iwasaki, T. Ibaraki, F. Glover, A very large-scale neighborhood search algorithm for the multi-resource generalized assignment problem, Discrete Optimization 1 (2004) 87–98.

[9] V. G. Deĭneko, G. J. Woeginger, A study of exponenential neighborhoods for the travelling salesman problem and for the quadratic assignment problem, Mathematical Programming 87 (3) (2000) 519–543.

[10] Ö. Ergun, J. B. Orlin, A dynamic programming methodology in very large scale neighborhood search applied to the traveling salesman problem, Discrete Optimization 3 (2006) 78–85.

[11] C. Meyers, J. Orlin, Very large-scale neighborhood search techniques in timetabling problems, in: Practice and Theory of Automated Timetabling VI, Vol. 3867 of Lecture Notes in Computer Science, 2007, pp. 24–39.

[12] E. Angel, E. Bampis, F. Pascual, An exponential (matching based) neighborhood for the vehicle routing problem, Journal of Combinatorial Optimization 15 (2008) 179–190.

[13] R. Ahuja, J. Goodstein, A. Mukherjee, J. Orlin, D. Sharma, A very large-scale neighborhood search algorithm for the combined through-fleet-assignment model, INFORMS Journal on Computing 19 (3) (2007) 416–428.

[14] B. Estellon, F. Gardi, K. Nouioua, Two local search approaches for solving real-life car sequencing problems, European Journal of Operational Research 191 (2008) 928–944.

[15] R. Ahuja, Ö. Ergun, J. B. Orlin, A. P. Punnen, A survey of very large scale neighborhoods search techniques, Discrete Applied Mathematics 123 (2002) 75–102.

[16] D. Pisinger, S. Ropke, Handbook of Metaheuristics, Vol. 146 of International Series in Operations Research & Management Science, Springer, 2010, Ch. Large Neighborhood Search, pp. 399–419.

[17] Y. Rios-Solis, F. Sourd, Exponential nieghborhood search for a parallel machine scheduling problem, Computers and Operations Research 35 (2008) 1697–1712.

[18] T. Brueggemann, J. Hurink, Two very large-scale neighborhoods for single machine scheduling, OR Spectrum 29 (2007) 513–533.

[19] T. Brueggemann, J. L. Hurink, T. Vredeveld, G. J. Woeginger, Performance of a very large-scale neighborhood for minimizing makespan on parallel machines, Electronic Notes in Discrete Mathematics 25 (2006) 29–33.

[20] E. Angel, E. Bampis, A multi-start dynasearch algorithm for the time dependent single-machine total weighted tardiness scheduling problem, European Journal of Operational Research 162 (2005) 281–289.

[21] R. K. Congram, C. N. Potts, S. L. V. de Velde, An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem, INFORMS Journal on Computing 14 (1) (2002) 52–67.

[22] T. Brueggemann, J. L. Hurink, Matching based very large-scale neighborhoods for parallel machine scheduling, Journal of Heuristics 17 (2011) 637–658.

[23] J. Hurink, An exponential neighbourhood for a one-machine batching problem, OR-Spektrum 21 (1999) 461–476.

[24] E. Possani, Lot streaming and batch scheduling: splitting and grouping jobs to improve production efficiency, Ph.D. thesis, University of Southampton (December 2001).
URL `http://eprints.soton.ac.uk/50621/`

[25] C. N. Potts, M. Y. Kovalyov, Scheduling with batching: A review, European Journal of Operational Research 120 (2000) 228–249.

[26] A. Wagelmans, A. E. Gerodimos, Improved dynamic programs for some batching problems involving the maximum lateness criterion, Operations Research Letters 27 (2000) 109–118.

[27] J. Stirling, I. Tweddle, James Stirling's Methodus Differentialis: An Annotated Translation of Stirling's Text, Sources and Studies in the History of Mathematics and Physical Sciences, Springer London, 2003.

[28] R. Martí, M. G. Resende, C. C. Ribeiro, Multi-start methods for combinatorial optimization, European Journal of Operational Research 226 (1) (2013) 1 – 8.