On-line batching problems

M.F. Reijnhoudt
(fransr@win.tue.nl)

Department of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

We introduce on-line batching problems as a new class of on-line combinatorial problems. In an on-line batching problem the task is to chop batches from a growing queue of objects, such that the cost is minimized, where the cost consists of setup cost for each batch, processing cost for each batch, and holding cost for each object that is not processed immediately.

For on-line batching problems we propose a solution approach based on decomposing the chopping and the processing of the batches. The chopping problem is solved with statistical learning, whereas the processing of a batch is independent of other batches. In this article we only consider the case where processing a batch is equivalent to solving a combinatorial optimization problem.

In order to generate learning examples we need partial solutions to infinite off-line batching problems. We show that these solutions can be obtained with a forward algorithm that solves a sequence of finite off-line batching problems using a dynamic programming formulation.

On the basis of an empirical performance analysis, we conclude that our approach outperforms all classical approaches in terms of cost and applicability.

1 Introduction

In an on-line batching problem every time unit exactly one object arrives. After each arrival the following steps must be taken.

1. Add the new object to the end of the queue.

2. Determine the next chop, that is the number of objects in the next batch, say \( k \), where \( k \) is smaller than or equal to the number of objects in the queue.

3. If \( k = 0 \) (no batch is chopped) then do nothing until the arrival of the next object, otherwise continue.

4. Remove the first \( k \) objects from the queue and process the batch containing these objects.

Note that each batch consists of objects that arrived consecutively in time. The goal is to find chops that minimize the total cost, which consists of a constant setup cost for each batch, processing cost for each batch, and holding cost for each object that is not processed immediately, but has to wait in the queue.

Batching is a common part of solutions of real-world problems in production, scheduling, and transport. However, often batching is done implicitly, because a new
batch is started at regular time intervals. For example, a factory makes a new production schedule every week, based on the available orders. This is generally considered good practice, and the focus is on processing batches of orders.

But with more demanding customers and Just In Time (JIT) production the uncertainty factor becomes more important, because it might be inefficient to use batches of fixed duration. For example, demand is known for the next ten days, but nothing is known behind that horizon. What will be a good decision now? In this article the focus is on choosing good batch sizes.

1.1 Related literature

Batching (or clustering) is done in different domains of combinatorial optimization, but they all differ from our problem in some aspects.

In vehicle routing problems ([Christofides, 1985]), a given set of delivery addresses must be partitioned over vehicles. After that, each vehicle faces an instance of the traveling salesman problem. Here batching reduces an off-line problem to a set of smaller off-line problems, while in our problem batching reduces an on-line problem to a set of smaller off-line problems. Furthermore, the partitioning is not restricted (so there are super-exponentially many different partitionings), while in batching problems batches consist of objects that arrived consecutively in time (so there are exponentially many different choppings).

Job batching problems are production scheduling problems where a given set of jobs must be partitioned in batches, with the restriction that the jobs of a batch are available for further processing after the last job of the batch is finished. This problem has been studied with different objectives ([Naddef & Santos, 1988], [Webster & Baker, 1995], and [Cheng, Kovalyov & Lin, 1997]). Here batching solves the off-line problem, because processing the remaining batches is trivial, while in our problem processing a batch can be as hard as solving an instance of an \(NP\)-hard combinatorial problem. Furthermore, the partitioning is, again, not restricted, while in our problem each batch can only contain objects, that arrived consecutively in time.

Other examples are given by [Federgruen & Tzur, 1995], who, among other problems, discuss the above presented production scheduling problem, with the additional constraint that the jobs must be processed in a given order. The algorithm they developed to solve this problem strongly depends on the property that processing a batch is trivial, while for our problem processing a batch can be equivalent to solving an instance of a (possibly \(NP\)-hard) combinatorial problem.

Our problem is derived from the on-line single-item lot-sizing problem, where demands must be satisfied by production. When the production and holding cost are concave, it can be proven that each production satisfies an integer number of demands ([Zangwill, 1968]). The demands that are satisfied by the production in one period, are consecutive in time, and can be seen as a batch. Note that for lot-sizing a new batch must start whenever the demand for the current period is not yet satisfied, while in on-line batching problems there is (theoretically) never an obligation to chop.
1.2 Outline

Our solution approach for on-line batching problems follows a divide-and-conquer approach, which is based on decomposing the chopping and the processing of the batches. The chopping which we would like to propose is the one which appears to be optimal when all the objects are known: the optimal off-line solution. Optimal processing of a batch can be done independently of all other batches.

We exploit the fact that afterwards optimal decisions can be calculated. In this way learning examples may be drawn from reality or from simulation, which provide the basis for a statistical learning approach.

Although we are mainly interested in on-line batching problems, we will also present the off-line version, because learning examples are constructed from partial solutions to instances of the infinite off-line batching problem. We show that these solutions can be obtained with a forward algorithm that solves a sequence of finite off-line batching problems using a dynamic programming formulation.

The remaining of this paper is organized as follows. In section 2 we present some examples of on-line batching problems. Section 3 contains formal definitions of on-line and off-line batching problems.

Section 4 is devoted to solution approaches for batching problems. In section 4.1 and section 4.2 we show how solutions to the finite off-line and the infinite off-line version can be obtained. In section 4.3 we present our approach to the on-line batching problem and we also discuss some classical approaches.

An empirical performance comparison of our approach and some classical approaches is presented in section 5, and we conclude in section 6.

2 Some examples

This section presents examples, to give an impression of on-line batching problems.

2.1 Container filling

In a transshipment trade, boxes must be packed into containers of fixed size. At the end of each time unit a new box of a certain size arrives.

After each arrival of a box either all the boxes stay in the queue for one more time unit or the next chop is determined, a packing plan is made, the boxes in the batch are packed into containers, and the containers are released for further shipment. Costs are incurred whenever a packing plan is made, a container is used, and when a box has to wait in the queue.

As soon as a batch must be processed, an instance of the bin packing problem ([Garey & Johnson, 1979]) has to be solved, where given a bin capacity $B$ and a set $\{a_1, \ldots, a_n\}$ of items, where $a_i$ is the size of the $i$th item, the problem is to find the minimum number of bins, such that all the items can be packed.
2.2 Pallet piling

In a pallet warehouse, boxes must be piled on pallets. Each time unit a new box with a certain height arrives. On one pallet $m$ piles of boxes can be made. The height of a pallet is the height of the highest pile.

After each arrival of a box either all the boxes stay in the queue for one more time unit or the next chop is determined, a piling plan is made, the boxes in the batch are piled on a pallet, and the pallet is stored in the warehouse. Costs are incurred whenever a pallet is used, a pallet is stored in the warehouse (this part of the cost is proportional to the height of the pallet), and when a box has to wait in the queue.

As soon as a batch must be processed, an instance of the multiprocessor scheduling problem has to be solved, where given a number $m$ of machines and a set of tasks $\{p_1, \ldots, p_n\}$, where $p_i$ is the length of the $i$th task, the problem is to find an $m$-processor schedule $P_1, \ldots, P_m$, where $P_1, \ldots, P_m$ is a partition of $p_1, \ldots, p_n$, such that the makespan,

$$\max_{1 \leq i \leq m} \sum_{p \in P_i} p,$$

is minimized.

Note that in the bin packing problem the bin capacity is given and the number of bins has to be minimized, whereas in the multiprocessor scheduling problem the number of processors (bins) is given and the makespan (the size of the bins) is to be minimized. So, in a way, the pallet piling problem is the dual of the container filling problem.

2.3 Pizza delivery

A pizza delivery service gets orders for pizza’s, which have to be made and delivered.

After each arrival of an order either all the orders stay in the queue for one more time unit or the next chop is determined, the pizza’s related to the orders in the batch are made, a route visiting each delivery address is determined, and the pizza deliverer is sent on his route. Costs are incurred whenever pizza’s are made, a pizza deliverer travels, and when an order has to wait in the queue.

As soon as a batch must be processed, an instance of the traveling salesman problem ([Garey & Johnson, 1979]) has to be solved, where given a set of cities $\{c_1, \ldots, c_n\}$ and distances $d(c_i, c_j)$ for each pair of cities, the problem is to find the shortest tour visiting each city once, that is, to find a permutation $\pi$, such that

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)})$$

is minimized. If there is more than one pizza deliverer, a vehicle routing problem ([Christofides, 1985]) has to be solved.

2.4 Propeller assembling

In a propeller factory, propellers are produced, each consisting of $D$ blades. Blades are made separately, so all blades vary slightly in weight. In each propeller the $D$ blades
must be balanced.

After each production of a blade either all the blades stay in the queue for one more
time unit or the next chop is determined, as many propellers as possible are assembled
from the blades in the batch, and the unused blades are considered difficult to match
and thus recycled. Costs are incurred whenever propeller assembling starts, a blade is
recycled, and when a blade has to wait in the queue.

As soon as a batch must be processed, an instance of a circular assignment problem
has to be solved, where given a set of masses \( \{m_1, \ldots, m_n\} \) and a threshold \( T \), the prob-
lem is to find the maximal number of mutually disjoint subsets \( P_j = \{m_j, \ldots, m_{jD}\} \),
such that for each \( P_j \) there is a permutation \( \pi_j \) for which

\[
\left| \sum_{k=1}^{D} m_{j\pi_j(k)} e^{2\pi ik/D} \right| < T.
\]

In order to get a minimization problem, we take the number of recycled blades as the
cost of a solution. Observe that this is not a classical maximal matching problem.

3 Definition of batching problems

In section 3.1 we will give a definition of on-line batching problems and we will define
the corresponding off-line version in section 3.2.

Suppose we have an instance of a batching problem of length \( n \) and a chopping
(solution) for that instance. Then the chopping is characterized by the chops and by
the times a chop is made. A chopping is an integer step function \( c(\cdot) \), where the steps
\((c(t) > c(t - 1))\) are at times when a chop is made and the step size \((c(t) - c(t - 1))\)
is the number of objects in the new batch.

**Definition 3.1 (chopping).**
A chopping for a batching problem of length \( n \) is a function \( c : \{0, \ldots, n\} \to \{0, \ldots, n\} \),
such that \( 0 = c(0) \leq c(1) \leq \ldots \leq c(n) = n \) and \( c(i) \leq i \) for \( i = 1, \ldots, n \).

The value of \( c(t) \) indicates the number of objects that are processed right after time
\( t \), so there is a chop right after object \( c(t) \) for all \( t \). Recall that each time unit exactly one
object arrives, so object \( a_t \) arrives at time \( t \) and cannot be processed before that time.
Figure 1 gives an example of a chopping of an instance with 10 objects, where, for
example, the decision to chop after object 3 is made at time 4 \((0 = c(3) < c(4) = 3)\).

To measure the quality of a chopping, the cost of a chopping can be calculated.

**Definition 3.2 (cost of a chopping).**
Let \( S \) denote a constant startup cost for each batch, \( b \) denote a constant weight factor,
\( q(\cdot) \) denote a processing cost function, and \( w(\cdot) > 0 \) a weight function.
Then the cost of the chopping \( c \) up to time \( n \) is

\[
C(c, n) = \sum_{t=1}^{n} [P(c, t) + H(c, t)],
\]
where the processing cost $P$ is given by

$$P(c, t) = \begin{cases} 0 & \text{if } c(t - 1) = c(t) \\ S + b \sum_{i=c(t)-1}^{c(t)} w(a_i) & \text{if } c(t - 1) < c(t), \end{cases}$$

and the holding cost $H$ is given by

$$H(c, t) = \sum_{i=c(t)+1}^{t} w(a_i).$$

The two constituents of the cost influence the optimal chopping in different directions. Minimizing the holding cost $H$ will result in short batches, while minimizing the processing cost $P$ will result in long batches. The meaning of the processing cost depends on how the objects in the batch are processed.

1. For the container filling example based on the bin packing problem it represents the number of bins needed to pack all the boxes in a batch.

2. For the pallet piling example based on the multiprocessor scheduling problem it represents the height of the highest pile on the pallet containing all the boxes in a batch.

3. For the pizza delivery example based on the traveling salesman problem it represents the length of the route that visits each delivery address in a batch.

4. For the propeller assembling based on a circular assignment problem it represents the number of recycled blades in a batch.

Note that processing a batch in the examples of section 2 is equivalent to solving a finite instance of an NP-hard problem, called the batch processing problem.
3.1 On-line batching problems

The variant that we are most interested in, is the infinite on-line variant of a batching problem, which has been introduced in section 1. So, the infinite on-line batching problem is the problem of deciding, after each arrival of an object, whether to wait for the next arrival or where to chop, in such a way that the total cost is minimized. Note that since the decision at time $t$ is only based on the objects $a_1, \ldots, a_t$ and the chopping already determined ($c(0), \ldots, c(t-1)$), decisions are based on partial knowledge.

Here batching reduces an infinite on-line batching problem to an infinite number of finite off-line batch processing problems, which can be solved independent of each other. Once the chopping is known, only relatively small instances of the batch processing problem remain. On the other hand, efficient chopping depends on solutions to possible instances of the batch processing problem. So the chopping and the resulting batch processing problems are mutually dependent.

When evaluating different approaches for obtaining a chopping for infinite on-line batching problems, it is often easier to compare their behaviour on finite on-line batching problems.

Definition 3.3 (finite on-line batching problem).
Let $n$ denote the number of periods, and $a_t$ denote the object that arrives at time $t = 1, \ldots, n$. Then the finite on-line batching problem is the problem of finding values

$$c(t) \in \{c(t-1), \ldots, t\}, \text{ for } t = 1, \ldots, n-1,$$

where $c(0) = 0$, $c(n) = n$, and the choice of $c(t)$ is only based on the objects $(a_1, \ldots, a_t)$ and the chopping up to time $t-1$, that is $c(0), \ldots, c(t-1)$, such that $C(c, n)$ is minimized.

3.2 Off-line batching problems

Our approach for solving on-line batching problems uses solutions to infinite off-line batching problems. These solutions can be obtained by solving sequences of finite off-line problems. In off-line problems there is complete knowledge about all the objects, so the main difference with on-line batching problems is that the choice of chop $c(i)$ possibly depends on all the objects.

Definition 3.4 (finite off-line batching problem).
Let $n$ denote the number of periods, and $a_t$ denote the object that arrives at time $t = 1, \ldots, n$. Then the finite off-line batching problem is the problem of finding values

$$0 = c(0) \leq c(1) \leq \ldots \leq c(n) = n,$$

where $c(t) \leq t$, such that $C(c, n)$ is minimized.

An infinite off-line problem is an unusual type of problem. It looks like an on-line problem because it is infinite, but it looks like an off-line problem since there is
complete knowledge about all the objects. The reason why we use it here is because partial solutions to this type of problem provide us with learning examples. So, we extend the problem formulation of the finite off-line batching problem along the lines of [Lundin & Morton, 1975] to get a formulation of an infinite off-line batching problem.

Definition 3.5 (infinite-horizon optimal chop).
Let $c$ and $n$ be integers with $c \leq n$. Suppose that for all $N \geq n$ and irrespective of the objects $a_{n+1}, a_{n+2}, \ldots$ there exists an optimal solution to the problem with $N$ objects with a chop right after time $c$.

Then chop $c$ is called an infinite-horizon optimal chop. 

An obvious formulation of an infinite off-line batching problem is to find infinite-horizon optimal chops $c_1, c_2, \ldots$. For practical reasons however, we concentrate on the determination of the first infinite-horizon optimal chop.

Definition 3.6 (infinite off-line batching problem).
An infinite off-line batching problem is a problem where given an infinite queue of objects $(a_1, a_2, \ldots)$, the task is to find an infinite-horizon optimal chop.

It is easy to see that this is without loss of generality, since by repeatedly solving instances of an infinite off-line batching problem, the infinite-horizon optimal chops $c_1, c_2, \ldots$ can be determined one by one.

The existence of infinite-horizon optimal chops can in general not be assured. In fact, for specific batch processing problems, it is possible to construct object sequences for which no infinite-optimal chop exist ([Bean, Smith & Yano, 1987]). Nevertheless, it is fairly safe to state that any reasonable batching problem is more likely to have infinite-horizon optimal chops than not ([Lundin & Morton, 1975] and [Morton, 1981]).

4 Solution approaches

For on-line batching problems we propose a solution approach based on statistical learning. Solutions to off-line batching problems guide the process of choosing efficient chops in an on-line batching problem. In order to generate learning examples for that solution approach we need partial solutions to infinite off-line batching problems.

We therefore first present a solution approach for finite off-line batching problems using a dynamic programming formulation. Secondly, we show that solutions to infinite off-line batching problems can be obtained with a forward algorithm that solves a sequence of finite off-line batching problems. Thirdly, we introduce our approach for on-line batching problems, and we end this section with a discussion of some classical approaches for the purpose of comparison.

4.1 Finite off-line batching problems

The cost of an optimal solution for the off-line batching problem with $n$ objects, $b(n)$, can be computed by

$$b(n) = \min_c C(c, n).$$
Note, however, that there are exponentially many valid choppings.

This problem can also be solved by dynamic programming. Then the focus is on
the batches, and not on the decisions that are made each time unit, so we are more
interested in the cost per batch than in the cost per time unit.

**Definition 4.1 (cost of a batch).**

Let \( a_u, \ldots, a_v \) be a batch in a chopping. Then

\[
B(a_u, \ldots, a_v) = S + b \cdot q(a_u, \ldots, a_v) + \sum_{t=0}^{v-u}(v-t) \cdot w(a_t)
\]

denotes the cost related to the objects in the batch, that is processed at time \( v \).

Then \( b(n) \) can be computed by the dynamic program given by

\[
b(n) = \begin{cases} 0 & \text{if } n = 0 \\ \min_{0 \leq l < n} \{ b(l) + B(a_{l+1}, \ldots, a_n) \} & \text{if } n > 0. \end{cases}
\]

So, for a specific value of \( n \), and given the values of \( b(l) \) for \( 0 \leq l < n \), the only thing
that remains is fixing the last chop such that the total cost is minimized.

The calculation of \( B(a_{l+1}, \ldots, a_n) \) involves an algorithm to find a solution for a
finite instance of the batch processing problem. All batch processing problems pre-
sented in section 2 are NP-hard combinatorial problems, so no deterministic algorithm
is known that solves the general problem in polynomial time. This makes it imprac-
tical to look for optimal solutions, so we have to use a heuristic for solving the batch
processing problem.

**Definition 4.2 (r-approximation heuristic).**

A heuristic \( H \) is called an \( r \)-approximation heuristic for a minimization problem if for
any instance \( A \) it holds that

\[
C_H(A) \leq r \cdot C_{opt}(A),
\]

where \( C_H(A) \) denotes the cost heuristic \( H \) produces for instance \( A \), and \( C_{opt}(A) \) de-
notes the minimal cost for instance \( A \).

**Theorem 4.1.** The algorithm which uses the dynamic programming formulation and an
\( r \)-approximation heuristic for the computation of \( B(a_u, \ldots, a_v) \) is an \( r \)-approximation
heuristic for the finite off-line batching problem.

**Proof.** Given an instance of a finite off-line batching problem. Look at the optimal
chopping. The cost of each batch is less than \( r \) times the optimal cost of the batch,
and so the total cost of the instance is less than \( r \) times the optimal cost. Because the
algorithm finds a solution with minimum cost with respect to the heuristic, \( r \) times the
optimal cost is an upper bound on the cost found.

Note that if the heuristic for the computation of \( B(a_u, \ldots, a_v) \) runs in polynomial
time, then the dynamic programming formulation can be computed in polynomial time,
for all \( n \).
4.2 Infinite off-line batching problems

Recall from section 3.2 that when an infinite-horizon optimal chop has been found, the infinite off-line batching problem decomposes into a finite off-line batching problem with the objects before the chop and a new infinite off-line batching problem with the objects after the chop. Because the problem regenerates itself after the chop, chops are equivalent to regeneration points in lot-sizing ([Wagner & Whitin, 1958], [Stehouwer, 1997]).

It appears that when an infinite-horizon optimal chop exists, it can be concluded from a finite number of objects. So, batching problems with an infinite number of objects can be solved by a forward algorithm that solves off-line batching problems with \( t \) objects for \( t = 1, 2, \ldots \), until a stopping rule indicates that an infinite-horizon optimal chop has been found, provided infinite-horizon optimal chops exist and there is an upper bound on the length of a batch in an optimal solution. Next we derive a stopping rule and an upper bound \( M \) on the length of a batch in an optimal solution.

Let \( \mathcal{L}_n \) denote the set of those \( l \) that minimize the right-hand side of the dynamic program presented in section 4.1. Then \( \mathcal{L}_n \) represents the set of last but one chops in an optimal solution for the instance with \( n \) objects.

Let \( \mathcal{R}_n \) denote the set of chops in an optimal solution for the instance with \( n \) objects. Then \( \mathcal{R}_n \) is defined by the forward recursion

\[
\mathcal{R}_n = \begin{cases} 
\emptyset & \text{if } n = 0 \\
\{n\} \cup \bigcup_{l \in \mathcal{L}_n} \mathcal{R}_l & \text{if } n > 0.
\end{cases}
\]

**Theorem 4.2.** Suppose a finite upper bound \( M \) exists on the length of a batch in an optimal solution for an instance of \( n \) objects that is independent of \( n \). Let

\[
\delta_n^k = \bigcap_{0 \leq j < k} \mathcal{R}_{n-j}.
\]

If \( t \in \delta_n^M \) then \( t \) is an infinite-horizon optimal chop.

**Proof.** We prove this by showing that if \( t \in \delta_n^M \), then \( t \in \mathcal{R}_N \) for all \( N \geq n \). Because \( M \) is an upper bound on the length of a batch in an optimal solution, it follows that for all \( N \geq n \) there exists a \( 0 \leq j < M \) such that \( n - j \in \mathcal{R}_N \).

Then \( t \in \delta_n^M \) implies that \( t \in \mathcal{R}_{n-j} \). From the definition of \( \mathcal{R}_n \) it is clear that \( \mathcal{R}_{n-j} \subseteq \mathcal{R}_N \), so \( t \in \mathcal{R}_N \), for all \( N \geq n \). \( \square \)

A forward algorithm corresponding with this stopping rule finds for each consecutive value of \( n \) the maximal value of \( k \) such that \( \delta_n^k \neq \emptyset \), and it stops when \( k = M \), because then Theorem 4.2 can be applied. In figure 2 the forward algorithm for the detection of infinite-horizon optimal chops corresponding with the above presented stopping rule is presented in pseudo-code. For reasons of convenience the code for the calculation of \( b(n), \mathcal{L}_n, \) and \( \mathcal{R}_n \) is left out.

Notice that Theorem 4.2 does not guarantee the existence of infinite-horizon optimal chops; therefore the termination of the forward algorithm cannot be assured. However, if infinite-horizon optimal chops exist, the minimal infinite-horizon optimal chop is a member of \( \delta \) when the algorithm terminates.
**procedure** ForwardAlgorithm

**var**

\( k, n : \text{int}; \)

\( \delta : \text{set of int}; \)

**begin**

\( \delta, k, n := \mathcal{R}_1, 1, 1; \)

**while** \( k < M \) **do** \{ \( \delta = \delta \cap \mathcal{R}_n \neq \emptyset \) \}

\( \delta, k, n := \delta \cap \mathcal{R}_{n+1}, k + 1, n + 1; \)

**if** \( \delta = \emptyset \) **then**

\( \delta, k := \mathcal{R}_n, 1; \)

**while** \( \delta \cap \mathcal{R}_{n-k} \neq \emptyset \) **do**

\( \delta, k := \delta \cap \mathcal{R}_{n-k}, k + 1 \)

**od**

**fi**

**od** \{ \( k = M \) \}

**end**

---

**Figure 2:** The forward algorithm in pseudo-code.

**Theorem 4.3.** Every partial solution with \( t \) objects found by the forward algorithm using an \( r \)-approximation algorithm for the batch processing problem, has a value that is less than or equal to \( r \) times the optimal value for the instance with \( t \) objects.

**Proof.** The instance with \( t \) objects is solved in an optimal way with respect to the heuristic for the batch processing problem. So according to Theorem 4.1, the cost of such an instance is less than or equal to \( r \) times the optimal cost for that instance.

For combinatorial problems an upper bound \( M \) on the length of a batch in an optimal solution can be derived.

**Theorem 4.4.** Let \((a_1, \ldots, a_v)\) be a batch in an optimal solution, with cost

\[
B(a_1, \ldots, a_v) = S + b \cdot q(a_1, \ldots, a_v) + \sum_{i=u}^{v} (v - t) \cdot w(a_i),
\]

as defined in section 4.

Then

\[
M = \left( \frac{S + b \cdot Q}{w_{\text{min}}} \right) + 1,
\]

is an upper bound on the length of a batch in an optimal solution, where \( w_{\text{min}} \) denotes a lower bound for \( w(a_1) \) and \( Q \) denotes a problem-specific upper bound on the extra cost incurred by splitting a batch into a batch containing the first object and another batch containing the other objects.
Proof. Because \((a_u, \ldots, a_v)\) is a batch in an optimal solution, splitting the batch into two smaller batches will result in cost at least \(B(a_u, \ldots, a_v)\), so it holds that

\[
B(a_u) + B(a_{u+1}, \ldots, a_v) - B(a_u, \ldots, a_v) =
S + b \left(q(a_u) + q(a_{u+1}, \ldots, a_v) - q(a_u, \ldots, a_v)\right) - (v - u) w(a_u) \geq 0.
\]

That means that the length of a batch in an optimal solution is bounded by

\[
 v - u + 1 \leq \left(\frac{S + b Q}{w_{min}}\right) + 1 = M,
\]

because \(q(a_u) + q(a_{u+1}, \ldots, a_v) - q(a_u, \ldots, a_v) \leq Q\), and \(w_{min} \leq w(a_u)\).

Note that the minimal value of \(Q\) depends on the algorithm to solve the batch processing problem. If the processing cost is calculated with an \(r\)-approximation heuristic, then \(Q = r \cdot Q_{opt}\), where \(Q_{opt}\) denotes the value of \(Q\) for an optimal algorithm. For the batch processing problems of the examples in section 2, we will give the minimal value for \(Q_{opt}\).

1. For the container filling example based on the bin packing problem \(Q_{opt} = 1\).
2. For the pallet piling example based on the multiprocessor scheduling problem \(Q_{opt} = p_{max}\), where \(p_{max}\) is the maximum height of a box.
3. For the pizza delivery example based on the traveling salesman problem \(Q_{opt} = 2c_{max}\), where \(c_{max}\) is the maximum distance to a customer.
4. For the propeller assembling based on an assignment problem \(Q_{opt} = D\), where \(D\) is the number of blades that is needed to assemble a propeller.

### 4.3 On-line batching problems

An approach for on-line batching problems is a procedure that decides at each integer moment \(t\) whether there will be a chop or not, based on the objects \(a_1, \ldots, a_t\) and the chopping up to time \(t - 1\). If there will be a chop, the next batch is processed, otherwise we wait for the arrival of the next object and apply the same function again.

In case of a finite on-line batching problem, the last batch is processed directly right after the last object arrived, irrespective of the on-line approach used. As with general on-line problems, there are some classical solution approaches, but first we will present statistical classification as a more recent solution approach.

#### 4.3.1 Solution approach based on statistical learning

Our solution approach for on-line batching problems is a divide-and-conquer approach, which is based on decomposing the chopping and the processing of the batches. The chopping which we want to approximate is the optimal off-line solution. In section 4.2 we have shown that partial solutions to infinite off-line problems can be calculated, if they exist. From such solutions to infinite instances from reality or from simulation,
learning examples can be constructed, which provide the basis for a statistical learning approach. Optimal solutions for the resulting batches are determined independently of all other batches, by solving an instance of the batch processing problem.

This approach outperformed all classical approaches for on-line lot-sizing in extensive experimental comparisons ([Aarts, Reijnhoudt, Stehouwer & Wessels, 2000]). Learning examples were obtained from partial solutions to the off-line lot-sizing problem. For on-line lot-sizing, two statistical learning approaches were used: one based on an artificial neural network (ANN) and one based on the K-nearest-neighbors method (KNN). See [Stehouwer, 1997] for more details. We propose statistical learning as a solution approach for on-line batching problems, because basically, lot-sizing can be seen as a batching problem with another type of holding cost and without processing cost. So, the same approaches can be applied to batching problems, implying the following two steps.

First, generate learning examples by calculating infinite-horizon optimal chops for instances of the infinite off-line batching problem with the forward algorithm of section 4.2 on instances with real-world data or on randomly generated sequences.

Secondly, train the statistical learning method with the learning examples and validate against a test instance to find (near-)optimal values for the parameters. In the empirical performance evaluation of section 5 we use the KNN method. The KNN method looks at the hyper-sphere around the vector with the known objects exactly enclosing K learning examples, and looks at the batch length with the highest frequency among the learning examples contained in that hyper-sphere. So, when using the K-nearest-neighbors method, training means choosing a value for K.

### 4.3.2 Classical solution approaches

For the purpose of comparison we also discuss some classical approaches, that can be easily applied to on-line batching problems. The first approach is a myopic approach, the other approaches make use of forecasting.

**CON** Use a constant length \( s_{\text{CON}} \) for each batch. Suppose every object that arrives has the same value \( \bar{a} \). Then there is no uncertainty and it is mathematically optimal to use the same batch length each time. Then \( s_{\text{CON}} \) is the batch length for which the total cost per unit time is minimized, that is \( s_{\text{CON}} \) is given by

\[
s_{\text{CON}} = \arg \min_{1 \leq s \leq M} \frac{B(\bar{a}_1, \ldots, \bar{a}_s)}{s},
\]

where \( \bar{a}_s = \bar{a} \) for all \( 1 \leq s \leq M \) and \( \bar{a} \) denotes the overall average object. This is done analogously to the economic order quantity method of [Harris, 1913]. It depends on the structure of an object whether \( \bar{a} \) can also be approximated from data drawn from reality. Note that the value of \( s_{\text{CON}} \) can also be approximated directly by simulation on data drawn from reality or on randomly generated sequences.

Based on the work of [Carlson, Beckman & Kropp, 1982] we can use a forecasting technique to extend the queue in combination with our forward algorithm to detect
an infinite-horizon optimal chop. If after $X$ iterations still no infinite-horizon optimal chop is found, the smallest element of $\delta$ is selected (see figure 2).

**AVG** Take the overall average object as a forecast for unknown future objects.

**MVG** Take the $x$-period moving average to forecast unknown future objects, that is, take the average of the last $x$ objects as forecast.

For the next three approaches we take $H$ the maximum of the number of objects that is known and a fixed $H_{\text{min}}$, and we use **AVG** or **MVG** to forecast unknown objects, if needed.

**PUP** Find the batch with the first local minimum price per unit object weight (based on [Gorham, 1968]), that is,

$$s_{\text{PUP}} = \arg\min_{1 \leq s \leq H} \frac{B(a_1, \ldots, a_s) \sum_{t=1}^{s} w(a_t)}{s}.$$

**PUT** Find the batch with the first local minimum price per unit time (based on [Silver & Meal, 1973]), that is,

$$s_{\text{PUT}} = \arg\min_{1 \leq s \leq H} \frac{B(a_1, \ldots, a_s)}{s}.$$

**FIX** Solve the finite off-line batching problem with $H$ objects and use the smallest first batch length among the optimal solutions.

## 5 Empirical performance comparison

We performed some experiments with the one-dimensional bin packing problem as batch processing problem in which we compared the presented solution approaches for on-line batching problems.

We choose the bin capacity $B$ equal to 100 and items were generated using the formula

$$a_t = \hat{a}_t + (\bar{a} - \frac{1}{2}R) \sin \left(\frac{2\pi t}{T}\right),$$

where $T = 4$ denotes the cycle length and $\hat{a}_t$ denotes a uniformly distributed random variable with mean $\bar{a} = 50$ and range $R = 60$.

As heuristic for the (NP-hard) bin packing problem we took the Best Fit heuristic, which puts items one by one in the order in which they arrived in the bin with the least remaining free space in which the item still fits. For the off-line case we also used the Best Fit Decreasing heuristic, which first sorts the items in non-increasing order and then applies the Best Fit heuristic. In the cost function we took the setup cost $S$ equal to 200, the weight factor $b$ for the processing cost equal to 500, and the weight of an item equal to the size of the item ($w(a_t) = a_t$).
With the forward algorithm of section 4.2 we generated 1000 learning examples using the Best Fit heuristic to solve the finite off-line instances of the bin packing problem. To train the \texttt{knn} algorithm we took a random test instance of 1000 items, calculated the cost for $1 \leq K \leq 120$, and took the value of $K$ for which the cost was minimal (in our case for $K = 54$). We used the same test instance to determine the optimal value of $1 \leq x \leq 20$ for the \texttt{mvg} algorithm and we found that $x = 6$ was optimal. We did not include the \texttt{ann} method in our experiments, because it is not trivial to choose good topologies and training the networks is rather time consuming.

Next, we generated 100 random instance of 100 items. For each instance we calculated the following costs.

- The optimal off-line cost with the dynamic programming formulation of section 4.1.
- The cost using the classical on-line solution approaches presented in section 4.3.2.
- The cost using the $K$-nearest-neighbors method (\texttt{knn}) discussed in section 4.3.1.
- The off-line cost (\texttt{ofl}) with the Best Fit heuristic (\texttt{bf}) and the Best Fit Decreasing (\texttt{bdf}) heuristic for the batch processing problem.

In figure 3 we present the approaches of the last three categories, the parameter values we used, the values found after training (between brackets), and the mean percentage deviation from the optimal off-line cost.

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Value</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUP BF, $H_{min} = 10$, AVG</td>
<td></td>
<td>25%</td>
<td></td>
</tr>
<tr>
<td>PUP BF, $H_{min} = 10$, MVG</td>
<td></td>
<td>22%</td>
<td></td>
</tr>
<tr>
<td>CON BF, ($S_{CON} = 2$)</td>
<td></td>
<td>19%</td>
<td></td>
</tr>
<tr>
<td>MVG BF, $X = 50$, ($x = 6$)</td>
<td></td>
<td>19%</td>
<td></td>
</tr>
<tr>
<td>FIJ BF, $H_{min} = 10$, MVG</td>
<td></td>
<td>19%</td>
<td></td>
</tr>
<tr>
<td>FIJ BF, $H_{min} = 10$, AVG</td>
<td></td>
<td>18%</td>
<td></td>
</tr>
<tr>
<td>AVG BF, $X = 50$</td>
<td></td>
<td>18%</td>
<td></td>
</tr>
<tr>
<td>PUT BF, $H_{min} = 10$, AVG</td>
<td></td>
<td>17%</td>
<td></td>
</tr>
<tr>
<td>PUT BF, $H_{min} = 10$, MVG</td>
<td></td>
<td>17%</td>
<td></td>
</tr>
<tr>
<td>I\texttt{JGHG} BF, ($K = 54$)</td>
<td></td>
<td>14%</td>
<td></td>
</tr>
<tr>
<td>\texttt{ofl} BF</td>
<td></td>
<td>1.9%</td>
<td></td>
</tr>
<tr>
<td>\texttt{ofl} \texttt{BFD}</td>
<td></td>
<td>1.2%</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: The mean percentage deviation from the optimal off-line cost for off-line heuristics and on-line approaches on 100 instances of 100 items
From the values in figure 3 we conclude that using a heuristic instead of solving the instances of the batch processing problem optimally increases the cost in general, but that for random instances the increase is much less than the upper bound on the increase. For example, the Best Fit heuristic is a \text{17}\text{10}-heuristic, but the mean extra cost for a random instance is less than 1.9% of the optimal cost, and the Best Fit Decreasing heuristic is an \text{14}\text{11}-heuristic, but for the same random instances the mean extra cost is even less than 1.2%.

Observe that the approaches CON, MGV, FIX, AVG, and PUT perform relatively good and they give similar results in terms of cost, PUT performs considerably worse, and all approaches are outperformed by KNN.

For many other combinatorial problems forecasting is not as easy as it is for the one-dimensional bin packing problem. For example, in the case of the pizza delivery (see section 2.3), assume that the delivery addresses are evenly spread over a circle of a certain size with the pizza delivery service as the center. Then forecasting by averaging produces delivery addresses that are the same or near to each other, which is much more unrealistic than forecasting in the container filling problem. When forecasting does not produce useful results CON and KNN are the only reasonable approaches.

6 Conclusions

We introduced and defined on-line and off-line batching problems as a new class of combinatorial problems.

Our solution approach for on-line batching problems based on statistical learning outperforms all classical approaches in terms of cost and applicability.

Learning examples for our solution approach are constructed from partial solutions to infinite off-line batching problems. We showed that these solutions can be obtained with a forward algorithm that solves a sequence of finite off-line batching problems using a dynamic programming formulation.

In an empirical performance analysis with the one-dimensional bin packing problem as the batch processing problem, our approach outperformed all classical approaches in terms of cost. Results obtained by [Stehouwer, 1997] suggest that artificial neural networks will even outperform the K-nearest-neighbors method.

When averaging does not produce useful results for the batch processing problem, the only classical competitor is the algorithm that uses the same batch length for every batch, which is obviously fairly rigid. So, our approach outperforms all classical approaches presented here in terms of applicability.

References

BEEAN, J.C., R.L. SMITH, AND C.A. YANO [1987], Forecast horizons for the discounted


