A Constraint Programming Approach for a Batch Processing Problem with Non-identical Job Sizes

Technical report 11/6/AUTO, École des Mines de Nantes

Arnaud Malapert[a,c,*], Christelle Guéret[b], Louis-Martin Rousseau[c,**]

[a] *École des Mines de Nantes, LINA UMR CNRS 6241, Nantes, France*
[b] *École des Mines de Nantes, IRCCyN UMR CNRS 6597, Nantes, France*
[c] *École Polytechnique de Montréal, CIRRELT, Montréal, Québec, Canada*

Abstract

This paper presents a constraint programming approach for a batch processing machine on which a finite number of jobs of non-identical sizes must be scheduled. A batch processing machine can process several jobs simultaneously and the objective is to minimize the maximal lateness. The constraint programming formulation proposed relies on the decomposition of the problem into finding an assignment of the jobs to the batches, and then minimizing the lateness of the batches on a single machine. This formulation is enhanced by a new optimization constraint which is based on a relaxed problem and applies cost-based domain filtering techniques. Experimental results demonstrate the efficiency of cost-based domain filtering techniques. Comparisons to other exact approaches clearly show the benefits of the proposed approach: our optimization constraint can optimally solve problems that are one order of magnitude greater than those solved by a mathematical formulation or by a branch-and-price.

Keywords: Combinatorial optimization, Artificial intelligence, Constraints satisfaction, Scheduling, Packing

1. Introduction

This paper presents a constraint programming approach for a batch processing machine on which a finite number of jobs of non-identical sizes must be scheduled. A parallel batch processing machine can process several jobs simultaneously. Such machines are encountered in chemical, pharmaceutical, aeronautical, and semiconductor wafer industries where an oven, a drier, or an autoclave is used during the process. These machines are often bottleneck because of their long processing times.

*Corresponding author
**Principal corresponding author
Email addresses: arnaud.malapert@mines-nantes.fr (Arnaud Malapert), christelle.gueret@mines-nantes.fr (Christelle Guéret), louis-martin.rousseau@polymtl.ca (Louis-Martin Rousseau)

Early work on batch processor models can be traced to Ikura & Gimple (1986) who propose an optimal algorithm for a problem with identical job processing times, identical job sizes, dynamic job arrivals and the objective of minimizing the makespan. Since then, heuristics (Perez et al., 2000; Wang & Uzsoy, 2002; Uzsoy, 1995), genetic algorithm (Wang & Uzsoy, 2002) and exact methods (Webster & Baker, 1995; Mehta & Uzsoy, 1998; Liu et al., 2007) have been proposed for identical job sizes and due date related performance measures. Several papers describe approaches for batch processing machine problems with non-identical job sizes and concern mostly completion time related performance measures (C_{max}, $\sum C_j$, $\sum w_j C_j$): heuristics (Azizoglu & Webster, 2000); genetic algorithm (Damodaran et al., 2006); simulated annealing (Damodaran et al., 2007); exact methods (Azizoglu & Webster, 2000, 2001; Dupont & Dhaenens-Flipo, 2002; Parsa et al., 2009; Kashan et al., 2009; Sabouni & Jolai, 2010). For an extensive review on scheduling with batching, we refer the reader to Potts & Kovalyov (2000).
On the other hand, constraint programming (CP) is an appealing technology in a variety of combinatorial problems which has grown steadily since the last three decades. According to Boucher et al. (1997), the main reason for success of CP on scheduling problems is the use of constraint propagation, which aims at removing from variable domains combinations of values which cannot appear in any consistent solution. Surprisingly, although they have been successfully used to solve various scheduling problems, only one paper proposes a constraint programming approach for this kind of problem: the filtering algorithms of Vilím (2007) for serial batch processing with sequence dependent setup times and job families. These algorithms are based on the well-known unary resource constraint, which deals mostly with completion time related performance measures. For an extensive review on constraint based scheduling, we refer the reader to Baptiste et al. (2001).

In this paper, we propose a new constraint programming approach for a problem derived from a real application in aeronautical industry. Composite components used in this kind of industry are made up of two categories of constituent materials: matrix (epoxy resin) and reinforcement (carbon fiber). They are fabricated according to the following process: the matrix is covered by carbon fiber, then the resulting part is consolidated into a solid structure at elevated temperature and pressure in an autoclave (batch processing machine). Several parts (jobs) of different size can be processed in a same autoclave at the same time. To our knowledge, only two papers (Daste et al., 2008b,a) concern the resolution of this problem which consists in scheduling the jobs on a batch processing machine in order to minimize the maximum lateness of a batch processing machine with non-identical job sizes. In these papers, the authors propose a mathematical formulation and a branch-and-price.
More formally, the problem can be described as follows. A set J of n jobs and one single batching machine with capacity b are given (*bounded model* – $b < n$). Each job j is characterized by an integer triplet (p_j, d_j, s_j), where p_j is its processing time, d_j is its due date, and s_j is its size. The sizes of the jobs are non-identical. The batch processing machine can process several jobs simultaneously as a batch as long as the sum of the sizes of the jobs that are in the batch does not exceed the capacity b of the machine. The processing time of a batch is equal to the longest processing time among the jobs in the batch (*parallel-batch* or *p-batch*). The completion time C_j of a job j is the completion time of the batch to which it belongs. The machine and jobs are assumed to be continuously available from time zero onwards, or equivalently, they have equal releases dates. Once

the processing of a batch has been initiated, no job can be removed from or added to the batch. The objective is to minimize the maximum lateness $L_{max} = \max_{1 \leq j \leq n}(C_j - d_j)$. This problem, denoted by $1|p\text{-}batch; b < n; non\text{-}identical|L_{max}$, is unary NP-hard because Brucker et al. (1998) proved that the same problem with identical job sizes is unary NP-hard.

This paper is organized as follows. Section 2 introduces our constraint programming formulation. Section 3 describes a new optimization constraint based on the resolution of a relaxed problem enhanced by cost-based domain filtering techniques. The search strategy inspired from well-known bin packing approaches and a new value selection heuristic are described in Section 4. Finally, Section 5 evaluates the performance of filtering rules and of the new value selection heuristics, then compares our results to those of a mathematical formulation and to a branch-and-price approach. Without loss of generality, we assume in the remainder of the paper that the jobs are sorted according to decreasing size ($s_j \geq s_{j+1}$), and that the size of each job is lower than the batch capacity ($s_j \leq b$). Note that the number m of batches is lower than or equal to n.

2. Constraint programming formulation

Constraint programming techniques have been widely used to solve scheduling problems. A *constraint satisfaction problem* (CSP) consists of a set $\mathcal{V}$ of variables defined by a corresponding set of possible values (the domains $\mathcal{D}$) and a set $\mathcal{C}$ of constraints. A solution of the problem is an assignment of a value to each variable such that all constraints are simultaneously satisfied. Constraints are handled through a propagation mechanism which allows the reduction of the domains of variables and the pruning of the search tree. The propagation mechanism coupled with a backtracking scheme allows the search space to be explored in a complete way. Scheduling is probably one of the most successful areas for CP thanks to specialized global constraints, which allow modelling an expressive and concise condition involving a non-fixed number of variables, as for instance, resource limitations. For an extensive review on constraint programming, we refer the reader to Rossi et al. (2006).

Our constraint programming formulation relies on the decomposition of the problem into finding an assignment of the jobs to the batches, and then minimizing the maximal lateness of the batches on a single machine. The problem of assigning the jobs to the batches is equivalent to the *one-dimensional bin packing problem*. Indeed, the definition of this problem is the following: given n indivisible items (jobs), each of a known non-negative size s_j, and m bins (batches), each of capacity b, can we pack the n items into the m bins such that the sum of the sizes of the items in any bin is not greater than b? This problem has been shown NP-complete (Garey & Johnson, 1979). Then, once the jobs are packed into the batches, the problem of scheduling the batches is equivalent to minimizing the maximal lateness of a set of jobs (batches) on a single machine. This problem, denoted as $1||L_{max}$, is polynomially solvable (Lawler, 1973): an optimal schedule is obtained by applying Jackson's scheduling rule, also known as the earliest due date (EDD-)rule which schedules the tasks in order of non decreasing due dates (Brucker, 2001).

We now present the CP model of the studied problem. Let $J = [1, n]$ denote the set of job's indices and $K = [1, m]$ denote the set of batch's indices. Let $d_{max} = \max_J\{d_j\}$ and $p_{max} = \max_J\{p_j\}$ be respectively the greatest due date and processing time among

the jobs. Let $B_j \in K$ denote the batch where the job j is packed, and $J_k \subseteq J$ denote the set of jobs which are packed into the batch k. These variables satisfy the relation: $\forall j \in J,\ B_j = k \Leftrightarrow j \in J_k$. The positive integer variables $P_k \in [0, p_{max}]$, $D_k \in [0, d_{max}]$ and $S_k \in [0, b]$ represent the processing time, the due date, and the load or total size of the batch k respectively. Lastly, let $M \in [0, m]$ be the number of non-empty batches and let L_{max} be the objective variable (unbounded). The constraint programming formulation is given below:

$$\texttt{maxOfASet}\left(P_k,\ J_k,\ [p_j]_J,\ 0\right) \qquad \forall k \in K \qquad (1)$$

$$\texttt{minOfASet}\left(D_k,\ J_k,\ [d_j]_J,\ d_{max}\right) \qquad \forall k \in K \qquad (2)$$

$$\texttt{pack}\left([J_k]_K,\ [B_j]_J,\ [S_k]_K,\ M,\ [s_j]_J\right) \qquad (3)$$

$$\texttt{sequenceEDD}\left([B_j]_J,\ [D_k]_K,\ [P_k]_K,\ M,\ L_{max}\right) \qquad (4)$$

Constraints (1), where $[p_j]_J$ is the associative array of processing times, enforce that the duration P_k of batch k is the maximal duration of its jobs (set J_k) if the batch is not empty, and equals 0 otherwise. Similarly, Constraints (2) enforce that the due date D_k of a batch k is the minimal due date of its jobs if the batch is not empty, and equals d_{max} otherwise. Indeed, the lateness of batch k defined as $\max_{j \in J_k}(C_k - d_j)$ is equal to $C_k - \min_{j \in J_k}(d_j)$ where C_k is the completion time of the batch k. Note that the lateness of a batch may be negative.

Constraint (3) is inspired from the global constraint of Shaw (2004) for the bin-packing problem. This constraint uses propagation rules incorporating knapsack-based reasoning, as well as a dynamic lower bound on the number of bins required. The constraint replaces the channeling constraints between assignment variables ($\forall j \in J,\ B_j = k \Leftrightarrow j \in J_k$) and enforces the consistency between assignments and loads ($\forall k \in K,\ \sum_{j \in J_k} s_j = S_k$). Furthermore, `pack` propagates the redundant constraint specifying that the sum of the bin loads is equal to the sum of the item sizes ($\sum_J s_j = \sum_K S_k$). Note that the limited capacity of the batches is enforced by the initial domain of the load variables S_k. AppendixA page 17 describes briefly our implementation of `pack`.

Lastly, Constraint (4) enforces that the objective value L_{max} is equal to the maximum lateness of the batches scheduled according to the EDD-rule. This constraint applies several filtering rules that are explained in details in Section 3.

Finally, a solution to the constraint programming formulation is composed of a feasible assignment of the jobs to the batches, and of the maximal lateness of the instance of the problem $1||L_{max}$ associated with the batches. Note that this model is also valid in presence of additional constraints like heterogeneous capacities, job incompatibilities, and load-balancing. Indeed, taking these constraints into account only implies to adapt the domains of the associated variables.

Korf (2003) mentions that dominance conditions on feasible assignments, which allows to consider a small subset of them, is a key property to solve bin packing problems. Efficient bin packing algorithms (Martello & Toth, 1990; Scholl et al., 1997; Korf, 2003; Shaw, 2004; Fukunaga & Korf, 2007) state equivalence and dominance rules to reduce the search space. But these procedures often consider that equal-sized items and equal-loaded bins can be swapped without sacrificing solution quality. In a batching machine context, swapping equal-sized items can change solution quality because of the modifications on the durations and due dates of the batches. Thus, these rules cannot be applied here.

Similarly, bin completion methods (Scholl et al., 1997; Korf, 2003; Fukunaga & Korf, 2007) can not be applied either since they consider dominance between assignments regarding only the loads of the batches.
However, it is well-known that the search space can be reduced by making sure that two jobs i and j such that $s_i + s_j > b$ belong to different batches. Since the jobs are sorted according to decreasing size, let j_0 denote the largest index such that any pair of jobs with indices lower than j_0 are in different batches: $j_0 = \max\{j \mid \forall\ 1 \leq i < j,\ s_{i+1} + s_i > b\}$. Then, Constraints (5) which pack the largest jobs into the first consecutive batches, can be added to the formulation.

$$B_j = j \qquad 1 \leq j \leq j_0 \tag{5}$$

3. Description of the `sequenceEDD` constraint

Pruning generally derives from feasibility reasoning. When coping with optimization problems, pruning can be done also on the basis of costs, *i.e.* optimality reasoning. Propagation can be aimed at removing combination of values which cannot lead to solutions whose cost is better then the best one found so far. Focacci et al. (1999) proposed to embed in global constraints optimization components representing suitable relaxations of the constraint itself. These components provide efficient operations research algorithms computing the optimal solution of the relaxed problem and a gradient function representing the estimated cost of each variable-value assignment. They show the benefit of using this information for pruning and for guiding the search on a variety of combinatorial optimization problems.

As explained in Section 2, the global constraint `sequenceEDD` enforces that the objective value L_{max} is equal to the maximum lateness of an EDD-sequence of batches. This constraint uses a relaxation of the problem that yields a lower bound for the objective function to prune portions of the search space. The general idea is to infer primitive constraints on the basis of information on costs. We use optimization components within a global constraint representing a proper relaxation of the problem, which consists in minimizing the maximal lateness of batches on a single machine. The optimization components provide the optimal solution of the relaxed problem, its value and a gradient function computing the cost to be added to the optimal solution for some variable-value assignments. The optimal value of this solution improves the lower bound of the objective function and prunes portions of the search space for which their lower bound is bigger than the best solution found so far. The relaxed problem is presented in Section 3.1 followed by four filtering rules presented in Sections 3.2, 3.3 and 3.4.

3.1. Relaxed problem

In this section, we describe the relaxed instance $I(\mathcal{A})$ of the problem $1||L_{max}$ built upon a partial assignment $\mathcal{A}$ of the variables, as well as an algorithm to solve it. At each point in the resolution, we have a partial assignment $\mathcal{A}$ which we define as the set of current domains of all variables. The current domain $\mathcal{D}(x)$ of a variable x is always a (non-strict) subset of its initial domain. An assignment $\mathcal{A}'$ extends a partial assignment $\mathcal{A}$, denoted $\mathcal{A}' \subseteq \mathcal{A}$, if the domain of any variable x in $\mathcal{A}'$ is a subset of its domain in $\mathcal{A}$. Let $\min(x)$ and $\max(x)$ be the minimum and maximum value of the domain $\mathcal{D}(x)$ of x in the current partial assignment. Let $x \leftarrow a$ denote the restriction

of the domain of x to a single value a Let $\delta_1, \delta_2, \ldots, \delta_l$ be the distinct increasing values of the due dates d_j among jobs $j \in J$. Note that l can be smaller than n if some jobs have identical due dates. However, for sake of clarity, we will consider that $l = n$. Let $K_{\mathcal{R}p}(\mathcal{A}) = \{k \in K \mid \max(D_k)\ \mathcal{R}\ \delta_p\}$ be the set of batches related to the due date δ_p by the arithmetic relation $\mathcal{R} \in \{<, \leq, =, \geq, >\}$. Similarly, Let $J_{\mathcal{R}p}(\mathcal{A}) = \{j \in J \mid d_j\ \mathcal{R}\ \delta_p\}$ be the set of jobs related to the due date δ_p. Finally, let $P(\mathcal{A}, \tilde{K}) = \sum_{k \in \tilde{K}} \min(P_k)$ be the minimal total duration of a set of batches $\tilde{K} \subseteq K$ in the partial assignment $\mathcal{A}$.
An instance $I(\mathcal{A})$ of the relaxed problem consists of n buckets where a bucket $p \in J$ is the set of batches $K_{=p}(\mathcal{A})$. Each bucket p has a due date δ_p and a processing time $\pi_p(\mathcal{A}) = P(\mathcal{A}, K_{=p}(\mathcal{A}))$. As the buckets are, by definition, numbered according to a strictly increasing order of their due date ($\delta_p < \delta_{p+1}$), minimizing the maximum lateness of $I(\mathcal{A})$ is equivalent to computing the maximal lateness of the sequence of buckets in this order. Let $C_p(\mathcal{A}) = \sum_{q=1}^{p} \pi_q(\mathcal{A})$ and $L_p(\mathcal{A}) = C_p(\mathcal{A}) - \delta_p$ be respectively the completion time and lateness of bucket p in the sequence. Therefore, the optimal objective value of instance $I(\mathcal{A})$ is given by: $L(\mathcal{A}) = \max_J(L_p(\mathcal{A}))$.

Figure 1 illustrates the solutions of two relaxed problems for a batching machine of capacity $b = 10$. Table 1(a) contains the jobs to schedule in the original problem $1|p\text{-}batch; b < n; non\text{-}identical|L_{max}$. Table 1(b) gives the buckets of instance $I(\mathcal{A}_1)$ where $\mathcal{A}_1$ is a solution of the problem, *i.e.* a total assignment. In this example, note that there are less buckets ($l = 3$) than jobs ($n = 4$). Figure 1(d) shows an optimal solution of the relaxed instance $I(\mathcal{A}_1)$ which is also an optimal solution of the original problem. The batching machine is represented as a drawing where the horizontal and vertical axes correspond respectively to the time and the load of the resource. A job is represented as a rectangle for which the length and the height respectively match the duration and the size of the task. Jobs with an identical starting time belong to the same batch. The solution contains three batches: the first one contains only job 1; the second one contains the jobs 2 and 4, and has a load equal to $s_2 + s_4 = 9$, a minimal duration equal to $\max\{p_2, p_4\} = 9$ and a maximal due date equal to $\min\{d_2, d_4\} = 2$; the third one contains only job 3. A bucket is represented as a dashed rectangle which encapsulates its batches. The lateness of each bucket is represented on the right part of the figure as a line starting from its due date and ending at its completion time. At this point, the bucket 1 contains batches 1 and 2 because $\max(D_1) = \max(D_2) = d_1$. The bucket 2 is empty because there is no batch k such that $\max(D_k) = \delta_2$ and its lateness, drawn as a dashed line, is therefore dominated by its first non-empty predecessor (bucket 1). The bucket 3 contains batch 3 with job 3.
Table 1(c) gives the buckets of instance $I(\mathcal{A}_2)$ where $\mathcal{A}_2$ is a partial assignment where the job 4 is not yet assigned. Figure 1(e) shows a solution of the relaxed instance $I(\mathcal{A}_2)$ in which the job 4 does not appear. At this point, each bucket contains a unique batch: the first, second and third buckets contain respectively the batches 1, 2 and 3. In fact, $\mathcal{A}_1$ is inferred at the root node by the propagation of Constraints (5) which enforces that $B_1 = 1$, $B_2 = 2$, and $B_3 = 3$.

At each point of the resolution, the construction of instance $I(\mathcal{A})$ can be done in $O(n)$. Indeed, the due dates of the jobs need to be sorted in increasing order which takes $O(n \log n)$, but this can be done once and for all before starting the search process. Then, building instance $I(\mathcal{A})$ consists in assigning each batch k to a bucket p. This can be done in $O(1)$ as the bucket of a batch k is the bucket p such that $\delta_p = \max(D_k)$.

Job	1	2	3	4
s_j	8	7	5	2
p_j	5	8	7	9
d_j	2	7	10	2

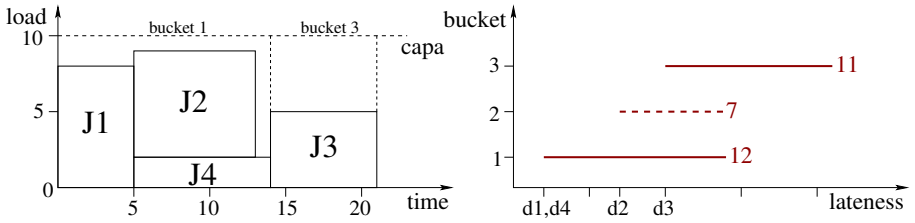
(a) Initial instance.

Bucket	1	2	3
δ_p	2	7	10
$\pi_p(\mathcal{A})$	14	0	7

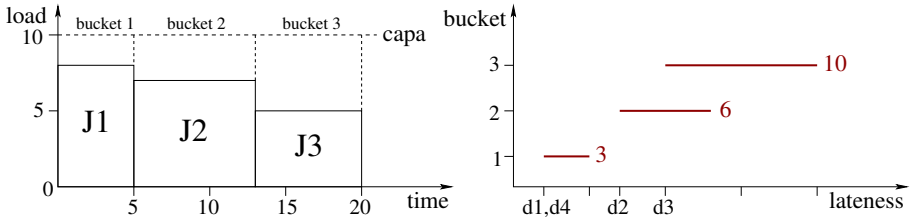
(b) Relaxed instance $I(\mathcal{A}_1)$.

Bucket	1	2	3
δ_p	2	7	10
$\pi_p(\mathcal{A})$	5	8	7

(c) Relaxed instance $I(\mathcal{A}_2)$.



(d) Solution of the relaxed problem for the solution (total assignment) $\mathcal{A}_1$.
$\mathcal{A}_1 = \{B_1 \leftarrow 1, B_2 \leftarrow 2, B_3 \leftarrow 3, B_4 \leftarrow 2\}$



(e) Solution of the relaxed problem for the partial assignment $\mathcal{A}_2$.
Job 4 is not present since it is not yet assigned.
$\mathcal{A}_2 = \{B_1 \leftarrow 1, B_2 \leftarrow 2, B_3 \leftarrow 3, B_4 \in [1, 4]\}$.

Figure 1: Two illustrative examples of the construction of $I(\mathcal{A})$.

Inserting batch k in its bucket, and updating the duration of the bucket is also done in constant time. The overall complexity is $O(n)$, because the number of batches m is lower than n. Finally, the resolution of $I(\mathcal{A})$ simply consists in sequencing the buckets in their numbering order, which can be done in $O(n)$. The following sections present filtering rules based on the resolution of this relaxed problem.

3.2. Filtering the lateness variable

The filtering rules on the lateness variable are based on the two following propositions.

Proposition 1. *L is a monotonic function from the partially ordered set of partial assignments onto the integers:*

$$\mathcal{A}' \subseteq \mathcal{A} \Rightarrow L(\mathcal{A}') \geq L(\mathcal{A}).$$

Proof. Since Constraints (1) enforce that the minimal duration of a batch in $\mathcal{A}'$ is greater than or equal to its minimal duration in $\mathcal{A}$, the total duration of a set of batches $\tilde{K}$ does not decrease from $\mathcal{A}$ to $\mathcal{A}'$: $\forall \tilde{K} \subseteq K,\ P(\mathcal{A}, \tilde{K}) \leq P(\mathcal{A}', \tilde{K})$.
Furthermore, since Constraints (2) enforce that assigning a new job in a batch belonging to bucket p can only imply its transfer to a bucket p' such that $p' \leq p$, the set of batches

before bucket p can only increase from $\mathcal{A}$ to $\mathcal{A}'$: $\forall p \in J,\ K_{\leq p}(\mathcal{A}) \subseteq K_{\leq p}(\mathcal{A}')$. Besides, since P_k is a positive integer variable, the function $P(\mathcal{A}, \tilde{K})$ is an increasing function from the sets of batches onto the integers: $\tilde{K} \subseteq \tilde{K}' \Rightarrow P(\mathcal{A}, \tilde{K}) \leq P(\mathcal{A}, \tilde{K}')$. Therefore, the following inequalities hold:

$$\begin{aligned} C_p(\mathcal{A}) &= \sum_{1 \leq q \leq p} \pi_q(\mathcal{A}) = P(\mathcal{A}, K_{\leq p}(\mathcal{A})) \\ &\leq P(\mathcal{A}', K_{\leq p}(\mathcal{A})) \leq P(\mathcal{A}', K_{\leq p}(\mathcal{A}')) \\ &\leq C_p(\mathcal{A}'). \end{aligned}$$

Since the completion time of each bucket p can only increase from $\mathcal{A}$ to $\mathcal{A}'$, the monotony of the function $I(\mathcal{A})$ is proven as follows:

$$(1) \wedge (2) \quad \Rightarrow \quad \forall p \in J,\ C_p(\mathcal{A}') \geq C_p(\mathcal{A}) \quad \Rightarrow \quad L(\mathcal{A}) \leq L(\mathcal{A}')$$

□

Proposition 2. *Once all durations and due dates of the batches are instantiated, the optimal schedule of the relaxed problem $I(\mathcal{A})$ corresponds also to an optimal schedule of any feasible assignment which extends $\mathcal{A}$.*

Proof. Once all durations and due dates of the batches are instantiated, the durations and latenesses of buckets stay unchanged until a feasible solution is found. Therefore, the proposition is satisfied if and only if the maximal lateness of the relaxed problem is equal to the maximal lateness of any solution which extends $\mathcal{A}$. As all the batches of a bucket p have the same due date δ_p, all the orderings of these batches are equivalent regarding the EDD-rule. Thus, the optimal schedule of the buckets for the relaxed problem corresponds to an optimal schedule of the batches (respecting the EDD-rule) for the initial problem. Note that it is often better to schedule the batches of a bucket according to non-decreasing processing time because it improves the average lateness of the batches. □

Let recall that $x \leftarrow v$ denotes the restriction of the domain of x to a single value a. Furthermore, let $x \not\leftarrow v$ denote the elimination of a single value v, $\min(x) \leftarrow v$ denote the restriction to values greater than v, and $\max(x) \leftarrow v$ denote the restriction to values lower than v.
From Proposition 2, we deduce the *final lateness filtering rule* (FF) which solves the relaxation problem to instantiate the objective variable once all durations and due dates are instantiated:

$$\forall k \in K, P_k \text{ and } D_k \text{ are instantiated} \Rightarrow L_{max} \leftarrow L(\mathcal{A}). \qquad \text{(FF)}$$

Note that this case occurs sometimes before a total assignment of the jobs to the batches is reached, since propagation can reduce all domains to singloton. A corollary of Propositions 1 and 2 is that, at each point of the search, the maximal lateness $L(\mathcal{A})$ of $I(\mathcal{A})$ is a lower bound on any feasible schedule which extends $\mathcal{A}$. Thus, the minimal objective value can be updated at each relevant domain change by the *lateness filtering rule* (LF):

$$\exists k \in K, \min(P_k) \text{ or } \max(D_k) \text{ have changed} \Rightarrow \min(L_{max}) \leftarrow L(\mathcal{A}) \qquad \text{(LF)}$$

3.3. Cost-based domain filtering of assignments

The *cost-based domain filtering rule of assignments* (AF) reduces the search space based on the marginal cost of the assignment of job j to batch k ($B_j \leftarrow k$). The idea is to eliminate, at each relevant domain change, every job j, as a candidate for packing in a batch k, if the marginal cost associated with the assignment of the job j to the batch k ($B_j \leftarrow k$) exceeds the best upper bound found so far, or more formally:

$$\begin{aligned} &\exists k \in K, \min(P_k) \text{ or } \max(D_k) \text{ have changed} \Rightarrow \\ &\qquad \forall j \in J, \text{ such that } |\mathcal{D}(B_j)| > 1 \text{ and } \forall k \in \mathcal{D}(B_j), \\ &\qquad\qquad L(\mathcal{A} \cap \{B_j \leftarrow k\}) > \max(L_{max}) \quad \Rightarrow \quad B_j \not\leftarrow k \end{aligned} \tag{AF}$$

where $\mathcal{A} \cap \{B_j \leftarrow k\}$ stands for $\mathcal{A} \cap \{B_j \leftarrow k, \min(P_k) \leftarrow p_j, \max(D_k) \leftarrow d_j\}$. Indeed, the propagation of Constraints (1) and (2) after the assignment $B_j \leftarrow k$ implies respectively that $\min(P_k) \leftarrow p_j$ and $\max(D_k) \leftarrow d_j$.
A simple filtering algorithm can compute the marginal cost from scratch for each possible assignment with an overall complexity of $O(n^3)$. We propose, in AppendixB, an $O(nm)$ version of this algorithm based on the incremental computation of marginal costs.

3.4. Cost-based domain filtering based on bin packing

In this section, we introduce a cost-based domain filtering rule based on the computation of marginal cost associated with a number of non-empty batches M. Let $K^\star = \{k \in K \mid \exists j \in J,\ B_j = k\}$ be the set of non-empty batches. Note that we assume that $|K^\star| = \max\{k \in K^\star\}$, *i.e.* the jobs are packed into consecutive batches. Let $J^\star = \{j \in J \mid \mathcal{D}(B_j) > 1 \wedge \max(B_j) > |K^\star|\}$ denote the set of *open* jobs, *i.e.* unpacked jobs that can be used to create new batches. The rule (PF1) updates the possible number of non-empty batches by considering the current number of non-empty batches and the number of open jobs.

$$\min(M) \leftarrow |K^\star| \qquad \max(M) \leftarrow |K^\star| + |J^\star| \tag{PF1}$$

This rule (also applied by `pack`) are required to ensure correctness of the reasoning presented below. Let $\mathcal{A}' = \mathcal{A} \cap \left(\cup_{j \in \tilde{J}} \{B_j \leftarrow k_j\}\right)$ denote the assignment of a subset $\tilde{J} \subseteq J^\star$ of open jobs to new batches, *i.e.* pairwise different empty batches ($\forall j \in \tilde{J},\ k_j > |K^\star|$ and $\forall i \neq j \in \tilde{J},\ k_i \neq k_j$). Therefore, the completion time and lateness of each bucket p have to be updated according to the total duration increase of its predecessors. Indeed, the new completion time of bucket p is equal to its completion time before the assignments plus the duration increase of its predecessors: $C_p(\mathcal{A}') = C_p(\mathcal{A}) + \sum_{j \in \tilde{J}_{\leq p}} p_j$. Unfortunately, the combinatorial of such assignments grows exponentially, and filtering them is difficult and costly.
Therefore, we compute a lower bound of $C_p(\mathcal{A}')$ by computing a lower bound of the completion time $C_p(\mathcal{A} \cap \{M \leftarrow q\})$ of a bucket p for a given number q of non-empty batches as follow. Let $\Pi(q)$ be the sum of q open jobs of smallest processing times. If $q - |K^\star|$ new batches are created with the open jobs, at least $q - |K^\star| - |J^\star_{>p}|$ new batches are scheduled before bucket p. Then, the completion time of any bucket p after the creation of $q - |K^\star *|$ batches is greater than its completion time before the creation

of the new batches plus the minimal sum $\Pi(q - |K^\star| - |J^\star_{>p}|)$ of the processing times of $q - |K^\star| - |J^\star_{>p}|$ open jobs:

$$\begin{aligned} C_p(\mathcal{A} \cap \{M \leftarrow q\}) &= C_p(\mathcal{A}) + \Pi(q - |K^\star| - |J^\star_{>p}|) \\ &\leq C_p\left(\mathcal{A} \cap \left(\cup_{j \in \tilde{J}} \{B_j \leftarrow k_j\}\right)\right) \qquad \forall \tilde{J} \subseteq J^\star,\ |\tilde{J}| = q - |K^\star| \end{aligned}$$

As a consequence, the maximal lateness of any solution extending $\mathcal{A}$ with exactly q non-empty batches is greater than $L(\mathcal{A} \cap \{M \leftarrow q\})$. The rule (PF2) updates the minimal objective value according to the marginal cost associated with the current minimum number of batches. The rule (PF3) decrements the maximum number of batches if its marginal cost exceeds the best upper bound found so far.

$$\min(L_{max}) \leftarrow L(\mathcal{A} \cap \{M \leftarrow \min(M)\}) \tag{PF2}$$

$$L(\mathcal{A} \cap \{M \leftarrow \max(M)\}) > \max(L_{max}) \Rightarrow \max(M) \leftarrow \max(M) - 1 \tag{PF3}$$

At each relevant domain change, the *cost-based domain filtering rule based on bin packing* (PF) applies the three filtering rules described above.

$$(\exists j \in J, \mathcal{D}(B_j) \text{ has changed}) \vee (\exists k \in K, \min(P_k) \text{ or } \max(D_k) \text{ have changed}) \Rightarrow \text{Apply rules (PF1), (PF2) and (PF3)} \tag{PF}$$

The computation of function Π is $O(n \log n)$ since it can be performed during initialization by sorting, and then summing the processing times of open jobs. The rules (PF1) and (PF2) are applied within linear time, since $L(\mathcal{A})$ can be computed in $O(n)$. The rule (PF3) is applied until the marginal cost associated with the maximum number of batches becomes feasible, *i.e.* at most $|J^\star|$ times. Therefore, the overall complexity of the cost-based domain filtering based on bin packing is $O(n^2)$.

Figure 2 illustrates the computation of marginal costs associated to the presence of exactly $q = 4$ non empty batches for the instance introduced in Table 1(a) and an empty assignment $\mathcal{A}_3$ (before the initial propagation). The set of open jobs $J^\star$ is equal to the set of jobs J whereas the set of non-empty batches $K^\star$ is empty. Therefore, rules (PF1) and (PF2) does not modify the domain of M. Since the completion time $C_p(\mathcal{A}_3)$ of each bucket p is equal to 0, the new completion time $C_p(\mathcal{A}_3 \cap \{M \leftarrow 4\})$ is equal to $\Pi(q - |J^\star_{>p}|)$. As a consequence, the bucket 1 ends at $\Pi(4 - |\{2,3\}|) = p_1 + p_3 = 12$, the bucket 2 ends at $\Pi(4 - |\{3\}|) = p_1 + p_3 + p_2 = 20$, and the bucket 3 ends at $\Pi(4 - |\emptyset|) = p_1 + p_3 + p_2 + p_4 = 29$. If we suppose that the best upper bound found so far is equal to 15, then the maximum number of non-empty batches is reduced from 4 to 3 and the rule (PF2) is applied for $q = 3$.

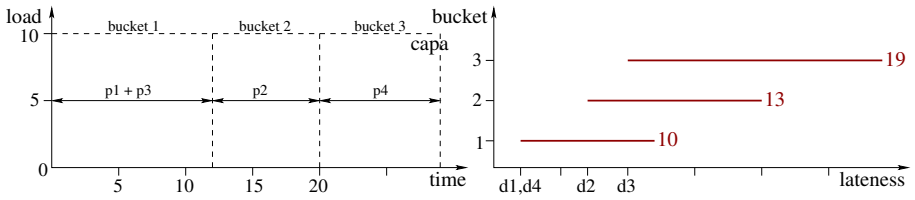


Figure 2: Example of the computation of $L(\mathcal{A}_3 \cap \{M \leftarrow 4\})$ for the (empty) assignment $\mathcal{A}_3$. $\mathcal{A}_3 = \{B_1 \in [1,4], B_2 \in [1,4], B_3 \in [1,4], B_4 \in [1,4]\}$.

4. Search strategy

At each node in the search tree, the branching selects the variable B_j of a job j using a variable selection heuristic, and assigns job j to a batch k chosen with a value selection heuristic, *i.e.* takes the decision $B_j \leftarrow k$. On backtracking, the search states that the chosen job cannot be placed in the selected batch. If this batch becomes empty, a symmetry breaking rule is applied which eliminates "equivalent" batches, *i.e.* other empty batches, from the list of candidates for the current job. Note that this dynamic symmetry breaking rule generalizes the idea behind Constraints (5) because, on backtracking, we would forbid packing a large jobs into another batch.
Several variable selection heuristics based on durations, sizes and due dates can be used. We chose the heuristic called *complete decreasing* (Gent & Walsh, 1997) which packs jobs in order of non-increasing size. Indeed, preliminary experiment showed that packing large jobs first improves filtering of `pack` and `sequenceEDD` constraints more than variants using also processing times and due dates.
Concerning the value selection heuristics, we investigated two classical heuristics in bin packing problems, namely *first fit* that selects the first available batch, and *best fit* which selects an available batch with the least free space. We also propose a new value selection heuristic named *batch fit* which selects an available batch having the least value of $\gamma(B_j \leftarrow k)$ where $\gamma(B_j \leftarrow k)$ measures roughly the fitness of the assignment within the instance $I(\mathcal{A})$ as follows:

$$\gamma(B_j \leftarrow k) = \frac{|\min(P_k) - p_j|}{\max_J(p_j) - \min_J(p_j)} + \frac{|\max(D_k) - d_j|}{\max_J(d_j) - \min_J(d_j)}$$

The idea of this heuristic is that a perfect solution would be composed of batches containing jobs having all identical durations and due dates.

5. Experimental results

This section presents computational experiments conducted to evaluate our approach. Section 5.1 describes the set of instances on which the experiments have been done. Section 5.2 evaluates the performance of the different filtering rules. The performance of the new value selection heuristic is studied in Section 5.3. Finally, our approach is compared to a mathematical formulation in Section 5.4 and to a branch-and-price in Section 5.5.
All the experiments were conducted on a cluster of Linux machines, each node with 48 GB of RAM and two quad core 2.4 GHz processor. Our implementation is based on `choco` (`http://choco.mines-nantes.fr`) which is an open source java library for constraint programming built on an event-based propagation mechanism with backtrackable structures. The constraint `pack` has been integrated in the latest releases ($\geq$ 2.0.0) and the solver has been extended with our new optimization constraint and search heuristics. The constraint `sequenceEDD` always applies the algorithm for the rule (AF) presented in AppendixB. The time limit has been fixed to 3600 seconds (1h).

5.1. Instances

Our algorithm has been tested on randomly generated instances proposed by Daste et al. (2008b) ranging from 10 to 100 jobs. The processing times are determined using the

uniform distribution ($p_j = U[1, 99]$) based on semiconductor manufacturing applications. Job sizes are generated from discrete uniform distribution between 1 and 10, and the machine capacity is $b = 10$ (inspired by works of Ghazvini & Dupont (1998)). For a given instance, once the processing times and sizes of all jobs are computed, the due dates are generated using the following formula: $d_j = U[0, \alpha] \times \tilde{C}_{max} + U[1, \beta] \times p_j$ (inspired by works of Malve & Uzsoy (2007)) where $\tilde{C}_{max} = (\sum_{j=1}^{n} s_j \times \sum_{j=1}^{n} p_j) \div (b \times n)$ is an approximation of the time required to process all the jobs on the batch processing machine. For each number of jobs $n \in \{10, 20, 50, 75, 100\}$, 40 benchmark instances were generated with $\alpha = 0.1$ and $\beta = 3$.

5.2. Performance of the filtering rules

Improving the filtering of a constraint could benefit the resolution but it happens that simple and short algorithms outperform more complicated ones. Therefore, it is reasonable to evaluate their filtering power (see Section 5.2.1) and to ask whether cost-based domain filtering techniques are useful during search (see Section 5.2.2). In this section, the name of a filtering rule is an abbreviation which reflects its nested structure, *i.e.* (FF) $\subset$ (LF) $\subset$ (AF) $\subset$ (PF). For instance, (PF) refers to the use of all filtering rules.

5.2.1. Destructive lower bound

We measured the performance of the different filtering rules by computing destructive lower bounds. Destructive lower bounds are obtained by setting a maximal objective function value F and trying to contradict (destruct) the feasibility of this reduced problem. In case of success, F is a valid lower bound. The lower bound F is incremented by one unit until the infeasibility of the reduced problem can not be proven without searching. The computation of such lower bound has the advantage of being independent from the branching heuristics. Three different destructive lower bounds were computed using (LF), (AF), and (PF) respectively. Note that we ignore (FF) since it is unlikely that it performs any filtering in this case. Because destructive lower bound is computed very quickly, another three (better) lower bounds where computed using shaving technique as suggested in Rossi et al. (2006). Shaving is similar to proof by contradiction. We propagate the assignment of a job j to a batch k. If an infeasibility is found, then the assignment is invalid and so we can eliminate job j as candidate for packing in batch k. To limit CPU time, shaving was used for each assignment $B_j \leftarrow k$ only once.

Let ub denote the best upper bound found for a given instance reported in AppendixC. The quality of the computed lower bound lb for a given instance is estimated by $(100 \times (lb + d_{max})) \div (ub + d_{max})$ (inspired by Malve & Uzsoy (2007)). This measure is equal to the optimality gap until the problem's size $n = 20$, since these instances have been solved optimally. Besides, it is almost equal to the optimality gap for problem's size $n = 50$, since only two instances are not solved optimally. Table 1 gives the average quality of destructive lower bounds as a function of size (columns 5–7 and 8–10). The efficiency of destructive lower bounds is also compared to those givens by the initial propagation (at the root node) (columns 2–4). Computation times are only reported for destructive lower bounds with shaving (columns 11–13) because they are not significant for the others ($\ll 1$ second).
Note also that Constraints (5) greatly contribute to the quality of these lower bounds.

	Initial Propag.			Destr. LB			Destr. LB + Shaving					
n	LF	AF	PF	LF	AF	PF	LF	AF	PF	$\bar{t}_{\mathrm{LF}}$	$\bar{t}_{\mathrm{AF}}$	$\bar{t}_{\mathrm{PF}}$
10	89.4	89.4	89.7	89.4	97.9	98.1	93.4	99.0	**99.2**	0.07	**0.05**	**0.05**
20	90.1	90.1	90.4	90.1	95.6	95.7	93.6	96.8	**97.0**	0.17	0.15	**0.1**
50	89.7	89.7	90.2	89.7	93.6	93.6	91.8	**94.1**	**94.1**	2.71	2.71	**1.43**
75	88.5	88.5	89.1	88.5	91.4	91.5	89.8	91.7	**91.8**	8.8	12.01	**5.25**
100	87.2	87.2	87.6	87.2	89.4	89.4	88.3	89.6	**89.7**	23.20	29.48	**14.04**

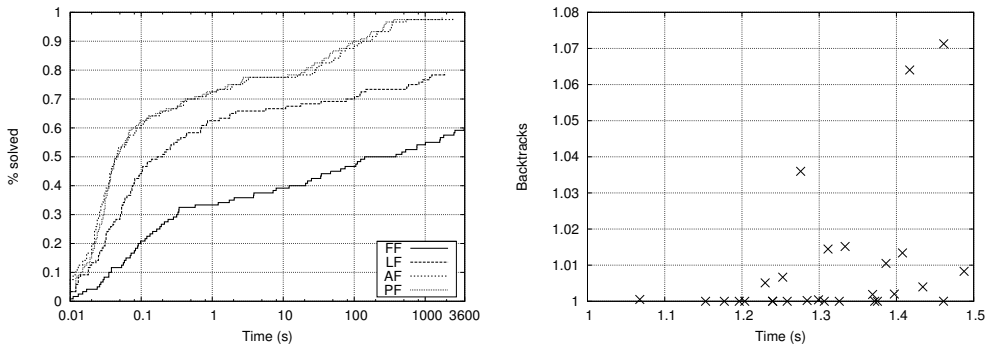
Table 1: Quality of destructive lower bounds.

The rule (AF) does not strengthen the lower bound during the initial propagation, whereas it greatly improves the two destructive lower bounds. The rule (PF) slightly improves on (AF) in any case. Besides, destructive lower bounds without shaving are efficient and their computation times stay negligible. However, using shaving slightly improves the quality of destructive lower bounds, but leads to a significant increase of computation time. Note that the rule (PF) improves the destructive lower bound with shaving in comparison to the rule (AF) while reducing its computation time by half.

5.2.2. Complete decreasing first fit

The procedure *complete decreasing first fit* (Gent & Walsh, 1997) packs jobs in order of non-increasing size, packing each job in the first available batch that will accommodate it. Since the destructive lower bound mechanism is deactivated, the impact of the filtering on the decision process is lessened. We chose instances that could be solved optimally by most of the filtering rules, from (FF) to (PF), so that comparisons could be made in reasonable CPU times. In this regard, we examined all instances with less than 50 jobs (120 instances).

Figure 3(a) shows the percentage of instances solved optimally as a function of the time in seconds. First, the performance of the rule (FF) show that the model is able to capture some solutions even when filtering happens only once all durations and due dates have been instantiated. We believe that the constraint programming approach, within this



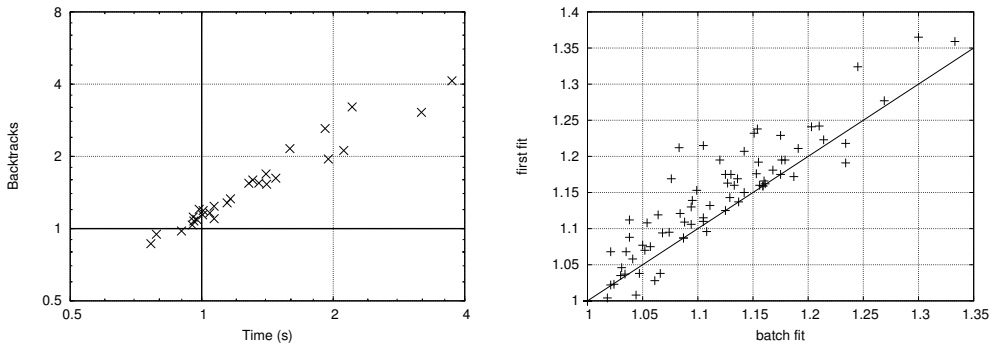
(a) Time comparison ($n \leq 50$). (b) (AF) ÷ (PF) ($n = 50$).

Figure 3: Comparison of filtering rules using *complete decreasing first fit*.

model, is extremely effective at identifying assignments that lead to equivalent EDD-schedule. Secondly, the rule (LF) based on the relaxed problem improves both solution quality and time over the final lateness rule (FF) but a number of instances remains unsolved optimally. Finally, only two instances with 50 jobs remain unsolved optimally within the time limit when using rule (AF) or (PF). However, this graph does not show clearly the gain offered by the rule (PF) over the rule (AF).
To this end, Figure 3(b) compares the resolution of rules (AF) and (PF). Each point represents one instance and its x coordinate is the ratio of the solving time with (AF) over the solving time with (PF), whereas its y coordinate is the ratio of the number of backtracks with (AF) over the number of backtracks with (PF). We only report the results of instances with solving times greater than 2 seconds, which happens only for problem of size $n = 50$. All points are located above and on the right of the point (1,1), because all instances are improved by the use of (PF). In fact, the rule (PF) lead to a similar behaviour than during the computation of destructive lower bounds. It always reduces the computation time and some instances are solved approximately 30 % times faster using (PF). However, the number of backtracks are roughly identical using (AF) or (PF).

5.3. Performance of value selection heuristics

We evaluate the impact of value selection heuristics on the resolution. In this regard, we examine all instances using all filtering rules without applying any destructive lower bound. We only compare *batch fit* to *first fit*, because *first fit* and *best fit* gives equivalent results within our approach. Indeed, the difference between the solution times of *first fit* and *best fit* is always less than 5 seconds on instances with less than 50 jobs, and even in this case, the difference represents at most 2 % of the solution time. Furthermore, both heuristics give the same objective values for all instances with strictly more than 50 jobs.
Figure 4(a) analyses the effect of the value selection heuristic on instances with 50 jobs and solving times greater than 2 seconds. Each point represents one instance and its x coordinate is the ratio of the solving time using *first fit* over the solving time using *batch fit*, whereas its y coordinate is the ratio of the number of backtracks with *first fit* over the



(a) *first fit* ÷ *batch fit* ($n = 50$). (b) Gap comparison ($n > 50$).

Figure 4: Comparison of value selection heuristics.

number of backtracks with *batch fit*. Notice also that the scale is logarithmic, and that all points are around the diagonal as the number of backtracks is roughly proportional to the time (top-left and bottom-right quadrants are empty). All points located above or on the right of the point (1,1) are instances improved by the use of *batch fit* (top-right quadrant). The heuristic *batch fit* globally improves the solution time and even by a factor 4 on some instances. However, the performances over a very few instances located below (1,1) are slightly degraded (bottom-left quadrant).
Figure 4(b) analyses the effect of the value selection heuristic on instances with strictly more than 50 jobs. Let lb denote the best lower bound found for a given instance reported in AppendixC. The quality gap of a solution ub for a given instance is estimated by $(ub + d_{max}) \div (lb + d_{max})$ (inspired by Malve & Uzsoy (2007)). This quality measure is often greater than the optimality gap since these instances have been infrequently solved optimally. Each point represents one instance and its x coordinate is the quality gap with *batch fit*, whereas its y coordinate is the quality gap with *first fit*. All points located above the line $(x = y)$ are upper bounds improved by *batch fit*. Using *batch fit* globally improves the solution, but the performances over a few instances are slightly degraded.

5.4. Comparison to a mathematical formulation

Our approach is now compared to a mathematical formulation of the studied problem inspired by Daste et al. (2008b). Let x_{jk} be a boolean variable equal to 1 if the job j is assigned to the k-th batch of the sequence. The positive integer variables P_k, D_k, and C_k represent the processing time, due date, and completion time of the k-th batch respectively.

$$\min L_{max} \tag{6}$$

Subject to:

$$\sum_{k \in K} x_{jk} = 1 \qquad \forall j \in J \tag{7}$$

$$\sum_{j \in J} s_j x_{jk} \leq b \qquad \forall k \in K \tag{8}$$

$$p_j x_{jk} \leq P_k \qquad \forall j \in J,\ \forall k \in K \tag{9}$$

$$C_{k-1} + P_k = C_k \qquad \forall j \in J,\ \forall k \in K \tag{10}$$

$$(d_{max} - d_j)(1 - x_{jk}) + d_j \geq D_k \qquad \forall j \in J,\ \forall k \in K \tag{11}$$

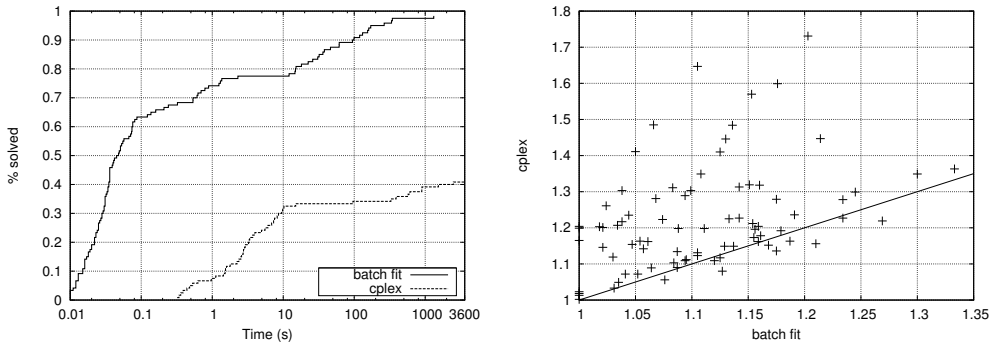
$$D_{k-1} \leq D_k \qquad \forall k \in K \tag{12}$$

$$C_k - D_k \leq L_{max} \qquad \forall k \in K \tag{13}$$

$$\forall j \in J,\ \forall k \in K,\ x_{jk} \in \{0, 1\}$$

$$\forall k \in K,\ C_k \geq 0,\ P_k \geq 0,\ D_k \geq 0$$

Constraints (6) represent the objective function which aims to minimize the maximum lateness. Constraints (7) state that each job must be assigned to exactly one batch, and Constraints (8) ensure that no batch exceeds the machine capacity b. These constraints define the bin packing model proposed by Martello & Toth (1990). Constraints (9) state that the duration of each batch k is equal to the maximum duration of its jobs. Constraints (10) ensure that the completion time of the k-th batch is equal to the completion time of the $(k-1)$-th batch plus the processing time of the k-th batch, *i.e.* the solution is a sequence of batches in their numbering order without idle time. Note that the addition of a first fictitious batch ending at the initial time ($C_0 = 0$) is required.



(a) Time comparison ($n \leq 50$). (b) Gap comparison ($n > 50$).

Figure 5: Comparison to the mathematical formulation.

Constraints (11) state that the due date D_k of the k-th batch is equal to the earliest due date of the jobs that are in the batch. Constraints (12) ensure that the sequence of batches satisfies the EDD-rule. These constraints help the resolution since most permutations of batches which lead to the same EDD-schedule are forbidden but they are not mandatory for model's correctness. Constraints (13) are the definition of batch latenesses and force L_{max} to be equal to the maximum lateness. Finally, a solution to the mathematical formulation is a feasible assignment of the jobs to an EDD-sequence of batches. As opposed to the constraint programming formulation, the solution encoding does not consider permutations of batches as equivalent solutions.

Our approach applies the value selection heuristic *batch fit* and all filtering rules. Comparison have been done with the resolution of the mathematical formulation stated as an Ilog OPL 6.1.1 model and solved by `Ilog cplex` 11.2.1. `Ilog cplex` provides a performance tuning tool which tries to find the best combination of its parameters. The tuning has been done on instances with 10 jobs since it is required to solve them optimally several times. First, the tuning changed the kind of *cuts* to reduce the number of branches explored. A *cut* is a constraint added to a model to restrict non-integer solutions that would otherwise be solutions of the continuous relaxation, Second, the tuning favoured feasibility by packing an job j into a batch k in the first branch ($x_{jk} = 1$) instead of the default behavior which forbids the packing ($x_{jk} = 0$).
Figure 5(a) shows the percentage of instances solved optimally within one hour as a function of the time in seconds. Our constraint programming approach outperforms the mathematical formulation: it solves more instances with solution times that are orders of magnitude lower.
Figure 5(b) compares the quality gap on instances with strictly more than 50 jobs. The time limit of `Ilog cplex` has been increased until 43200 seconds (12h). Each point represents one instance and its x coordinate is the quality gap that we obtained, whereas its y coordinate is the quality gap obtained with `Ilog cplex`. All points located above the line ($x = y$) are upper bounds improved by our approach. Despite a smaller time limit, the constraint programming approach provides better solution than the mathematical formulation on the vast majority of instances. Furthermore, the difference between the

two upper bounds is very tight when the mathematical formulation found the best, whereas it can be significant when the constraint approach did.

5.5. Comparison to a branch-and-price

Branch-and-price integrates branch-and-bound and column generation methods for solving large scale integer programs (Barnhart et al., 1998). Daste et al. (2008a) proposed a branch-and-price algorithm with a master problem where each column is a batch. A solution of the master problem is a feasible sequence of batches. At each iteration, the objective of the subproblem is to find one column (batch) which improves the current solution of the master problem. This pricing subproblem is solved by a greedy algorithm, then, if needed, by an exact enumeration method.
Their algorithm has been tested on instances presented in Section 5.1 with less than 50 jobs and a time limit of 3600 seconds (1h). All their experiments have been conducted on a Pentium 2.66 GHz with 1 GB of RAM. Detailed results on benchmark instances are not mentioned. 55% and 8% of instances of size $n = 20$ and $n = 50$ are solved by the branch-and-price procedure within one hour. Although branch-and-price is more efficient than the mathematical formulation, it is less efficient than our constraint programming which solves optimally all instances of size $n = 20$ with an average solution time lower than 1 second, and 95% of instances of size $n = 50$ with an average solution time lower than 100 seconds.

6. Conclusion

We have presented a constraint programming approach to minimize the maximal lateness for a batch processing machine on which a finite number of jobs of non-identical sizes must be scheduled. This approach exploits a new optimization constraint based on a relaxed problem which applies cost-based domain filtering rules and the search is enhanced with a dedicated branching heuristics. Computational results demonstrate the positive effect of each component and give better solution with computation times that are orders of magnitude lower than a mathematical formulation or a branch-and-price based on bin packing and sequencing models.
In further research, we will apply our approach to problems with completion time related measures (C_{max}, $\sum C_j$, $\sum w_j C_j$). In addition, subsequent research topics include the study of parallel batching machines and additional constraints, for instance job release dates that remain incompatible with our approach.

AppendixA. Short description of the `pack` constraint

In this section, we describe briefly our implementation of the global constraint `pack` originally proposed by Shaw (2004) for the bin-packing problem. Let recall that this constraint replaces the channeling constraints between assignment variables ($\forall j \in J,\ B_j = k \Leftrightarrow j \in J_k$) and enforces the consistency between assignments and loads ($\forall k \in K,\ \sum_{j \in E_k} s_j = S_k$). Furthermore, `pack` propagates the redundant constraint specifying that the sum of the bin loads is equal to the sum of the item sizes ($\sum_J s_j = \sum_K S_k$).
First the constraint performs the propagation of typical model which corresponds to Constraints (7) and (8) of the mathematical formulation (see Section 5.4). In addition,

it uses propagation rules incorporating knapsack-based reasoning, as well as a dynamic lower bound on the number of bins required.
The additional constraint propagation rules are based upon treating the simpler problem of packing items into a single bin. That is, given a bin, can we find a subset of items that when packed in the bin would bring the load in a given interval? This problem is a type of knapsack or subset sum problem (Kellerer et al., 2004). Proving that there is no solution to this problem for any bin would mean that search could be pruned. If an item appears in every set of items that can be placed in the bin, we can commit it to the bin. Conversely, if the item never appears in such a set, we can eliminate it as a candidate item. So, if no legal packing can attain a given load, it cannot be a legal load for the bin. Shaw (2004) proposed efficient algorithms which do not depend on item sizes or bin capacity, but which do not in general achieve generalized arc consistency on the subset sum problem as opposed to Trick (2003). Next, they adapt bounding procedures to partial assignments. The idea is to transform a partial assignment into a new bin packing instance and to apply a bounding procedure to this new instance. Our implementation uses the lower bound L^{1D}_{CCM} (Carlier et al., 2007), which dominates the lower bound L^{1D}_{MV} originally used.
Our implementation differs also from the one of Shaw (2004) by the addition of variables S_k and M. Indeed, variables S_k helps to express additional constraints and were maintained internally by Shaw (2004). The variable M represents the objective of the bin packing problem and its value can be restricted by other constraints.
Shaw (2004) demonstrated that this new constraint can cut search by orders of magnitude. Additional comparisons showed that the new constraint coupled with a standard packing algorithm based on *complete decreasing* can significantly outperform Martello & Toth (1990)'s procedure.

AppendixB. Algorithm for cost-based domain filtering of assignments

We recall that a simple algorithm can compute marginal costs from scratch for each possible assignment with an overall complexity of $O(n^3)$. In this section, we describe an $O(n^2)$ version of this algorithm (cf. Algorithm 1) based on the incremental computation of marginal costs. The principle consists in exploiting formula (B.1) for quick computation of marginal costs by distinguishing several cases. For instance, the assignment of a job to any empty batch leads to the same marginal cost. In fact, the completion time of a bucket p after the assignment of a job j to a batch $k \in \mathcal{D}(B_j)$ is given by:

$$C_p(\mathcal{A} \cup \{B_j \leftarrow k\}) = \begin{cases} C_p(\mathcal{A}) & \text{if } \delta_p < \min(d_j, \max(D_k)) \quad \text{(B.1a)} \\ C_p(\mathcal{A}) + \max(p_j, \min(P_k)) & \text{if } d_j \leq \delta_p < \max(D_k) \quad \text{(B.1b)} \\ C_p(\mathcal{A}) + \max(p_j - \min(P_k), 0) & \text{if } \delta_p \geq \max(D_k) \quad \text{(B.1c)} \end{cases}$$

Figure B.6 illustrates the idea behind the formula (B.1): the assignment of a job j to a batch k leads to transfer the batch k from a bucket p to one of its predecessors or itself, *i.e.* the bucket $p' \leq p$ such that $\delta_{p'} = \min(d_j, \max(D_k))$. First, the sequence of buckets $1, \ldots, p'-1$ stays unchanged and so does the completion times (B.1a) and its maximal lateness which is dominated by $L(\mathcal{A})$. Indeed, the rule (LF) is applied after the initialization of buckets and before the rule (AF). Then, completion times of buckets $p', \ldots, p-1$ are postponed for the updated batch duration $\max(\min(P_k), p_j)$ and

Algorithm 1: Cost-based domain filtering of assignments.

$\mathcal{B} = \emptyset$; // Set of batches to prune
$\Lambda_{\geq p} = -\infty$; // Lateness of sub-sequence $p, \ldots, n$
foreach $k \in K$ **do** $\Lambda^k_{\geq p} = -\infty$; // Lateness without batch k of sub-sequence $p, \ldots, n$
/* Try assignments in decreasing order of buckets */
L1 **for** $p \leftarrow n$ **to** 1 **do**
  $\Lambda_{\geq p} = \max(L_p(\mathcal{A}), \Lambda_{\geq p})$; // Maximal lateness of sub-sequence $p, \ldots, n$
  $\Delta = \max(L_{max}) - \Lambda_{\geq p}$; // Maximal duration increase allowed at bucket p
  $\mathcal{B}_{=p} = \{k \in K_{=p} \mid \min(P_k) > 0 \wedge (|\mathcal{D}(P_k)| > 1 \vee |\mathcal{D}(D_k)| > 1)\}$; // New batches to prune
  /* Update durations of new batches at bucket p */
L2 **foreach** $k \in \mathcal{B}_{=p}$ **do** $\max(P_k) \leftarrow \min(P_k) + \Delta$;
  /* assignments of an available job j to other batches */
L3 **foreach** $k \in \mathcal{B}$ **do** $\Lambda^k_{\geq p} = \max(L_p(\mathcal{A}), \Lambda^k_{\geq p})$;
L4 **forall** $j \in J_{=p}$ **such that** $|\mathcal{D}(B_j)| > 1$ **do**
    /* Assignment to non-empty batches */
L5 **foreach** $k \in \mathcal{B}$ **do**
      if $\delta_p < \min(D_k)$ **then** $\mathcal{B} = \mathcal{B} \backslash \{k\}$;
      else if $\Lambda^k_{\geq p} + \max(\min(P_k), p_j) > \max(L_{max})$ **then** $B_j \not\leftarrow k$;
    /* Assignment to empty batches */
    if $p_j > \Delta$ **then** $\max(B_j) \leftarrow |K^\star|$;
  /* Update data structure */
L6 **foreach** $k \in \mathcal{B}_{=p}$ **do** $\Lambda^k_{\geq p} = \Lambda_{\geq p} - \min(P_k)$;
  $\mathcal{B} = \mathcal{B} \cup \mathcal{B}_{=p}$;

therefore, they are given by (B.1b). Last, the duration increase $\max(p_j - \min(P_k), 0)$ of batch k is added to the completion time of buckets $p, \ldots, n$ (B.1c). If the batch duration increase is zero, then the maximal lateness of buckets $p, \ldots, n$ is also dominated by $L(\mathcal{A})$

The backward algorithm 1 prunes variables B_j according to the rule (AF). The pruning is illustrated in Figure B.7 by considering the assignment of job 4 within the partial assignment $\mathcal{A}_2$ (see Figure 1 page 7). After initialization, Loop L1 iterates over buckets in descending order to detect infeasible assignments. The maximal lateness $\Lambda_{\geq p}$ and the maximal duration increase Δ of the sub-sequence of buckets $p, \ldots, n$ are updated using recursive formulas derived from Section 3.1.

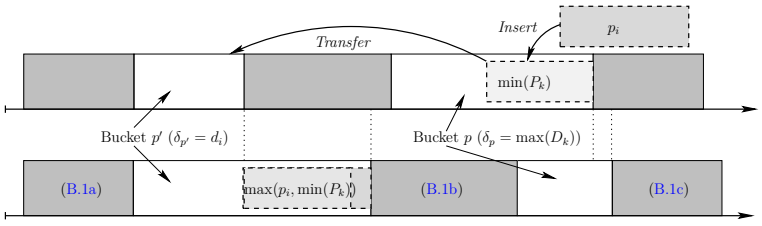
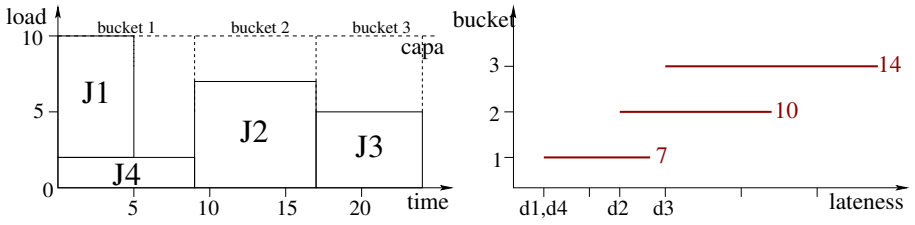
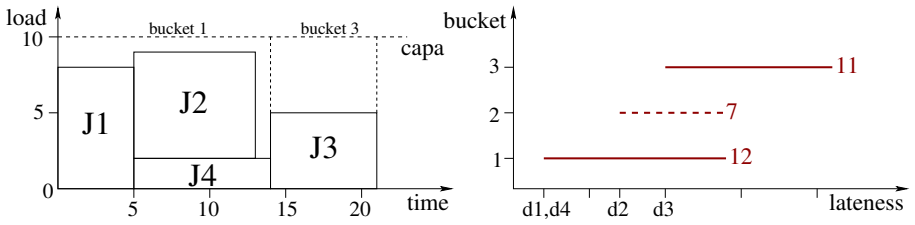


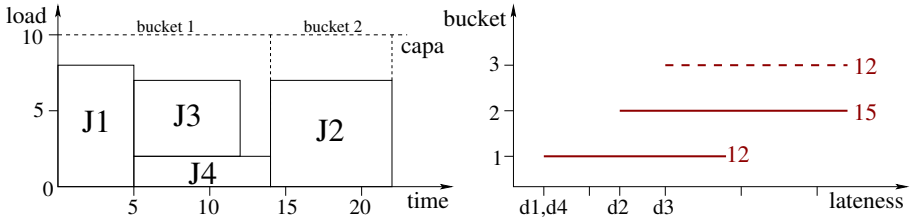
Figure B.6: Anticipating the impact of an assignment on the buckets.



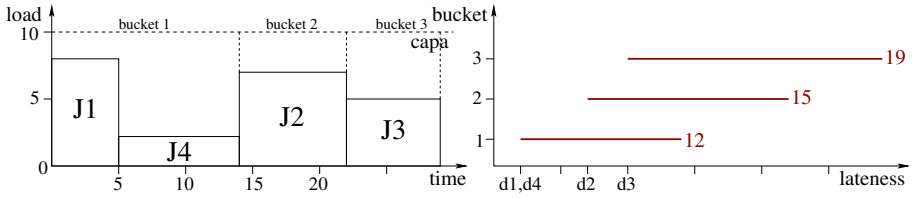
(a) Assignment: $B_4 \leftarrow 1$



(b) Assignment: $B_4 \leftarrow 2$



(c) Assignment: $B_4 \leftarrow 3$



(d) Assignment: $B_4 \leftarrow 4$

Figure B.7: Impact on the solution of the relaxed problem of the assignment of job 4 to a batch for the partial assignment $\mathcal{A}_2 = \{B_1 \leftarrow 1, B_2 \leftarrow 2, B_3 \leftarrow 3, B_4 \in [1, 4]\}$.

New batches of $\mathcal{B}_{=p}$ scheduled at the latest at bucket p that are neither empty nor full are now considered for pruning by the algorithm. Loop L2 updates the maximal duration of these batches, but the removal of infeasible assignments is delegated to Constraints (1). This loop detects simultaneously all infeasible assignments of jobs such that $\max(D_k) \leq d_j$, since their marginal costs depend only on the duration increase of the batch when (B.1b) is not defined as illustrated in Figure 7(a).
Loop L3 updates incrementally the maximal lateness $\Lambda^k_{\geq p}$ of the sequence $p, \ldots, n$ without the batch k. Then, Loop L4 inspects only assignments of jobs that would effectively

lead to the transfer of the target batch to the bucket p. The internal Loop L5 considers batches to prune discovered in previously visited buckets. If the transfer is infeasible because of due date restrictions, then the batch is eliminated from the set $\mathcal{B}$ of batches to prune. Otherwise, the assignment of job j to the batch k is eliminated if the maximal lateness after its transfer into the bucket p exceeds the current upper bound. In fact, the incremental computation of $\Lambda^k_{\geq p}$ reduces the complexity of (B.1b) from linear time to constant time. In Figure 7(b), the transfer of batch 2 from the bucket 2 to 1 does not modify the sequence of batches, whereas the transfer of batch 3 from the bucket 3 to 1 leads to swap batches 2 and 3 in Figure 7(c).
Then, the algorithm simultaneously considers all assignments of a job to an empty batches for which (B.1b) and (B.1c) are equals, since the duration increase is equal to the updated duration ($\min(P_k) = 0$) as illustrated in Figure 7(d). In this case, B_j is restricted to values lower than the index $|K^\star|$ of the last non-empty batch.
Lastly, Loop L6 initializes the value $\Lambda^k_{\geq p}$ of new batches to prune by subtracting their contribution to the lateness of the sub-sequence. Then, the set $\mathcal{B}$ of batches to prune is updated.
The complexity of Algorithm 1 depends only on its nested loops because instructions are done in constant time. A simple calculation shows that the complexity depends only on nested Loops L1, L4, and L5. In the worst case, nested Loops L1 and L4 iterates over a partition of the jobs J which is done in $O(n)$. Loop L5 is done in $O(m)$, because $\mathcal{B} \subseteq K$ is a subset of batches. Therefore, the complexity of Algorithm 1 is $O(nm)$.

AppendixC. Results summary

Table C.2 page 23 summarizes the best known lower and upper bounds. The first column (#) reports the index of the instance. Columns 2–3 report optimum values (L_{max}) for problem sizes $n = 10, 20$. Columns 4–9 report lower (lb) and upper (ub) bounds for problem sizes $n = 50, 75, 100$. Note that the lower bound is not given when the upper bound has been proven optimal.

Azizoglu, M., & Webster, S. (2000). Scheduling a batch processing machine with non-identical job sizes. *International Journal of Production Research*, *38*, 2173–2184.

Azizoglu, M., & Webster, S. (2001). Scheduling a batch processing machine with incompatible job families. *Computers and Industrial Engineering*, *39*, 325–335.

Baptiste, P., Le Pape, C., & Nuijten, W. (2001). *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer.

Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., & Vance, P. H. (1998). Branch-and-price: column generation for solving huge integer programs. *Operations Research*, *46*, 316–329.

Boucher, E., Bachelu, A., Varnier, C., Baptiste, P., & Legeard, B. (1997). Multi-criteria comparison between algorithmic, constraint logic and specific constraint programming on a real scheduling problem. In *Proceedings of the 3rd International Conference on the Practical Application of Constraint Technology* (pp. 47–64). Blackpool: The Practical Application Ltd.

Brucker, P. (2001). *Scheduling Algorithms — Third Edition*. Berlin-Göttingen-Heidelberg-New York: Springer-Verlag.

Brucker, P., Gladky, A., Hoogeveen, H., Koyalyov, M., Potts, C., Tautenham, T., & van de Velde, S. (1998). Scheduling a batching machine. *Journal of Scheduling*, *1*, 31–54.

Carlier, J., Clautiaux, F., & Moukrim, A. (2007). New reduction procedures and lower bounds for the two-dimensional bin packing problem with fixed orientation. *Comput. Oper. Res.*, *34*, 2223–2250.

Choco Team (2011). Choco: an open source java constraint programming library. `http://choco.mines-nantes.fr`.

Damodaran, P., Manjeshwar, P. K., & Srihari, K. (2006). Minimizing makespan on a batch-processing machine with non-identical job sizes using genetic algorithms. *International Journal of Production Economics*, *103*, 882–891.

Damodaran, P., Srihari, K., & Lam, S. S. (2007). Scheduling a capacitated batch-processing machine to minimize makespan. *Robotics and Computer-Integrated Manufacturing*, *23*, 208–216.

Daste, D., Gueret, C., & Lahlou, C. (2008a). A Branch-and-Price algorithm to minimize the maximum lateness on a batch processing machine. In *Proceedings of the 11th international workshop on Project Management and Scheduling PMS'08* (pp. 64–69). IStanbul Turquie.

Daste, D., Gueret, C., & Lahlou, C. (2008b). Génération de colonnes pour l'ordonnancement d'une machine à traitement par fournées. In *7ième conférence internationale de modélisation et simulation MOSIM'08* (pp. 1783–1790). Paris France volume 3.

Dupont, L., & Dhaenens-Flipo, C. (2002). Minimizing the makespan on a batch machine with non-identical job sizes: An exact procedure. *Computers and Operations Research*, *29*, 807–819.

Focacci, F., Lodi, A., & Milano, M. (1999). Cost-based domain filtering. In *CP '99: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming* (pp. 189–203). London, UK: Springer-Verlag.

Fukunaga, A. S., & Korf, R. E. (2007). Bin completion algorithms for multicontainer packing, knapsack, and covering problems. *J. Artif. Intell. Res. (JAIR)*, *28*, 393–429.

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, New York: W. H. Freeman and Company.

Gent, I. P., & Walsh, T. (1997). From approximate to optimal solutions: Constructing pruning and propagation rules. In *IJCAI* (pp. 1396–1401).

Ghazvini, F. J., & Dupont, L. (1998). Minimizing mean flow times criteria on a single batch processing machine with non-identical jobs sizes. *International Journal of Production Economics*, *55*, 273–280.

IBM (2011). IBM Ilog Cplex Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

Ikura, Y., & Gimple, M. (1986). Scheduling algorithms for a single batch processing machine. *Operations Research Letters*, *5*, 61–65.

Kashan, A. H., Karimi, B., & Ghomi, S. F. (2009). A note on minimizing makespan on a single batch processing machine with nonidentical job sizes. *Theoretical Computer Science*, *410*, 2754–2758.

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). *Knapsack Problems*. Springer, Berlin, Germany.

Korf, R. E. (2003). An improved algorithm for optimal bin packing. In G. Gottlob, & T. Walsh (Eds.), *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003* (pp. 1252–1258). Morgan Kaufmann.

Lawler, E. (1973). Optimal sequencing of a single machine subject to precedence constraints. *Management science*, *19*, 544–546.

Liu, L., Ng, C., & Cheng, T. (2007). Scheduling jobs with agreeable processing times and due dates on a single batch processing machine. *Theoretical Computer Science*, *374*, 159–169.

Malve, S., & Uzsoy, R. (2007). A genetic algorithm for minimizing maximum lateness on parallel identical batch processing machines with dynamic job arrivals and incompatible job families. *Computers & OR*, *34*, 3016–3028.

Martello, S., & Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. New York: Wiley.

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

Parsa, N. R., Karimi, B., & Kashan, A. H. (2009). A branch and price algorithm to minimize makespan on a single batch processing machine with non-identical job sizes. *Computers & Operations Research*, *In Press, Accepted Manuscript*, –.

Perez, I., Fowler, J., & Carlyle, W. (2000). Minimizing total weighted tardiness on a single batch process machine with incompatible job families. *Computers and Operations Research*, *32*, 327–341.

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

Rossi, F., Beek, P. v., & Walsh, T. (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. New York, NY, USA: Elsevier Science Inc.

Sabouni, M. Y., & Jolai, F. (2010). Optimal methods for batch processing problem with makespan and maximum lateness objectives. *Applied Mathematical Modelling*, *34*, 314–324.

Scholl, A., Klein, R., & Jürgens, C. (1997). Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers & OR*, *24*, 627–645.

Shaw, P. (2004). A constraint for bin packing. In M. Wallace (Ed.), *Principles and Practice of Constraint*

Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings (pp. 648–662). Springer volume 3258 of *Lecture Notes in Computer Science*.

Trick, M. A. (2003). A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, *118*, 73–84.

Uzsoy, R. (1995). Scheduling batch processing machines with incompatible job families. *International Journal of Production Research*, *33*, 2685–2708. IP.

Vilím, P. (2007). *Global Constraints in Scheduling*. Ph.D. thesis Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic KTIML MFF, Universita Karlova, Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic.

Wang, C., & Uzsoy, R. (2002). A genetic algorithm to minimize maximum lateness on a batch processing machine. *Computers and Operations Research*, *29*, 1621–1640. IP.

Webster, S., & Baker, K. (1995). Scheduling groups of jobs on a single machine. *Operations Research*, *43*, 692–703. IP.

	$n = 10$	$n = 20$	$n = 50$		$n = 75$		$n = 100$	
#	L_{max}	L_{max}	lb	ub	lb	ub	lb	ub
1	71	389		1349	1548	1712	2647	2939
2	59	282		893	1316	1600	1964	2430
3	37	63		132	-27	263	58	427
4	-103	-147		20	372	453	72	379
5	132	337		1502		1481	1866	2218
6	21	254		1139	1409	1725	1851	2246
7	-90	33		74	59	149	386	400
8	-17	-61		11	9	58	66	190
9	163	309		1043	1807	2164	1863	2575
10	7	313		1114	1327	1435	2509	2863
11	50	-39		285		201	98	310
12	-15	-119		91	314	396	-62	316
13	58	278		948	1743	2033	2421	2702
14	4	209		1322	1450	1830	1770	2024
15	-21	127		178	-31	302	158	633
16	-49	-20		252	-35	192	280	575
17	150	303		802	1598	1919		2840
18	203	256		1022	1052	1423	2485	2595
19	-101	49		367	-41	59	-9	420
20	-49	-16		202	185	288	-120	416
21	59	240		1156	1754	2054	2440	2592
22	44	291		1128		2105	1468	2167
23	-66	17		181	159	181	62	130
24	-43	203		-102	-64	139	473	609
25	136	620		1574	1405	1820		2609
26	84	274		1354	1347	1592	1965	2460
27	58	46		162	10	125	-73	334
28	-105	-34		102	192	273	125	482
29	140	257		1255	1531	1687	2420	2607
30	49	353		1010	1528	1714	1829	2438
31	-13	106		269	67	200	-40	338
32	-64	-33		324	-72	238		449
33	197	423		1117	1776	2032	2134	2528
34	44	293		936	1132	1421	2156	2605
35	27	53		292	204	301	-18	496
36	-32	-55	-41	30	-19	47	-32	259
37	81	227		1328	1535	1880	2516	2780
38	10	216		1025	1260	1452	2027	2323
39	11	58		114	354	355	96	414
40	-24	14	15	102	204	261	175	393

Table C.2: Summary of best known lower and upper bounds for the benchmark instances of Daste et al..