Constraints (2023) 28:320–361
https://doi.org/10.1007/s10601-023-09347-2

Exact methods for the Oven Scheduling Problem

Marie-Louise Lackner[1] · Christoph Mrkvicka[2] · Nysret Musliu[1] · Daniel Walkiewicz[2] · Felix Winter[1]

Accepted: 24 March 2023 / Published online: 4 July 2023
© The Author(s) 2023

Abstract

The Oven Scheduling Problem (OSP) is a new parallel batch scheduling problem that arises in the area of electronic component manufacturing. Jobs need to be scheduled to one of several ovens and may be processed simultaneously in one batch if they have compatible requirements. The scheduling of jobs must respect several constraints concerning eligibility and availability of ovens, release dates of jobs, setup times between batches as well as oven capacities. Running the ovens is highly energy-intensive and thus the main objective, besides finishing jobs on time, is to minimize the cumulative batch processing time across all ovens. This objective distinguishes the OSP from other batch processing problems which typically minimize objectives related to makespan, tardiness or lateness. We propose to solve this NP-hard scheduling problem using exact techniques and present two different modelling approaches, one based on batch positions and another on representative jobs for batches. These models are formulated as constraint programming (CP) and integer linear programming (ILP) models and implemented both in the solver-independent modeling language MiniZinc and using interval variables in CP Optimizer. An extensive experimental evaluation of our solution methods is performed on a diverse set of problem instances. We evaluate the performance of several state-of-the-art solvers on the different models and on three variants of the objective function that reflect different real-life scenarios. We show that our models can find feasible solutions for instances of realistic size, many of those being provably optimal or nearly optimal solutions.

✉ Marie-Louise Lackner
marie-louise.lackner@tuwien.ac.at

Christoph Mrkvicka
christoph.mrkvicka@mcp-alfa.com

Nysret Musliu
musliu@dbai.tuwien.ac.at

Daniel Walkiewicz
daniel.walkiewicz@mcp-alfa.com

Felix Winter
winter@dbai.tuwien.ac.at

[1] Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, DBAI, TU Wien, Favoritenstraße 9, 1040 Vienna, Austria

[2] MCP GmbH, Canovagasse 7, 1010 Vienna, Austria

Keywords Oven Scheduling Problem · Parallel batch processing · Constraint programming · Integer linear programming

1 Introduction

In the electronics industry, many components need to undergo a hardening process which is performed in specialised heat treatment ovens. As running these ovens is a highly energy-intensive task, it is advantageous to group multiple jobs that produce compatible components into batches for simultaneous processing. However, creating an efficient oven schedule is a complex task as several cost objectives related to oven processing time, job tardiness and setup costs need to be minimized. Furthermore, a multitude of constraints that impose restrictions on the availability, capacity, and eligibility of ovens have to be considered. Due to the inherent complexity of the problem and the large number of jobs that usually have to be batched in real-life scheduling scenarios, efficient automated solution methods are thus needed to find optimized schedules.

Over the last three decades, a wealth of scientific papers investigated batch scheduling problems. Several early problem variants using single machine and parallel machine settings were categorized and shown to be NP-hard [1]; further literature reviews on different problem variants have been provided by Mathirajan et al. [2] as well as Fowler et al. [3]. Batch scheduling problems share the common goal that jobs are processed simultaneously in batches in order to increase efficiency. Besides this common goal, a variety of different problems with unique constraints and solution objectives arise from different applications in the chemical, aeronautical, electronic and steel-producing industry where batch processing machines can appear in the form of autoclaves [4], ovens [5] or kilns [6].

For example, a just-in-time batch scheduling problem that aims to minimize tardiness and earliness objectives has been recently investigated in [7]. Another recent study [6] introduced a batch scheduling problem from the steel industry that includes setup times, release times, as well as due date constraints. Furthermore, a complex two-phase batch scheduling problem from the composites manufacturing industry has been solved with the use of CP and hybrid techniques [8].

Exact methods used for finding optimal schedules on batch processing machines involve dynamic programming [9] for the simplest variants as well as CP and mixed integer programming (MIP) models. CP models have e.g. been proposed by Malapert et al. [4] and by Kosch et al. [10], where both publications consider batch scheduling on a single machine with non-identical job sizes and due dates but without release dates. A novel arc-flow based CP model for minimizing makespan on parallel batch processing machines was recently proposed [11]. Branch and Bound [12] and Branch and Price [13] methods have been investigated as well. As the majority of batch scheduling problems are $\mathcal{NP}$-hard, exact methods are often not capable of solving large instances within a reasonable time limit and thus (meta-)heuristic techniques are designed in addition. These range from GRASP approaches [14] and variable neighbourhood search [15], over genetic algorithms [16, 17], ant colony optimization [18] and particle swarm optimization [19] to simulated annealing [20].

In this paper, we introduce the Oven Scheduling Problem (OSP), which is a new real-life batch scheduling problem from the area of electronic component manufacturing. The OSP defines a unique combination of cumulative batch processing time, tardiness and setup cost objectives that need to be minimized. To the best of our knowledge, this objective has not been studied previously in batch scheduling problems. Furthermore, we take special

requirements of the electronic component manufacturing industry into account. Thus, the problem considers specialized constraints concerning the availability of ovens as well as constraints regarding oven capacity, oven eligibility and job compatibility.

The main contributions of this paper are:

- We introduce and formally specify a new real-life batch scheduling problem.
- Based on two different modelling approaches, we propose solver independent CP and ILP models. These models are implemented using the high-level modeling language MiniZinc, which allows us to use a palette of state-of-the-art solvers and to find the best-suited one for the Oven Scheduling Problem. In addition, these two modelling approaches are formulated using interval variables, which are decision variables tailored for the use in scheduling problems. The interval variable models are implemented using the OPL modeling language and solved with CP Optimizer.
- We provide a construction heuristic that can be used to quickly obtain feasible solutions.
- All our solution methods are extensively evaluated through a series of benchmark experiments, including the evaluation of several search strategies and a warm-start approach. Results are compared for three different variants of the objective function that arise from practical use cases. For a sample of 80 benchmark instances, we obtain optimal results for 37 instances, and provide upper and lower bounds on the objective for all instances.

We make the benchmark instances as well as the models described in this paper publicly available on Zenodo [21]. The current paper is a significant extension of our CP 2021 conference paper [22]. The major addition to our conference paper is an entirely new modeling approach based on representative jobs for batches, which produces the best results on our benchmark set (see Sections 4 and 5.2).

The rest of this paper is organized as follows: We first provide a description of the OSP (Section 2) before we formally define the problem and formulate an ILP model (Section 3). Then we present a second modelling approach using representative jobs (Section 4). Alternative model implementations as well as search strategies and the warm-start approach are described in Section 5. Finally, we present and discuss experimental results (Section 6).

2 Description of the OSP

The OSP consists in creating a feasible assignment of jobs to batches and in finding an optimal schedule of these batches on a set of ovens, which we refer to as machines in the remainder of the paper.

Jobs that are assigned to the same batch need to have the same *attribute*;[1] in the context of heat treatment this can be thought of as the temperature at which components need to be processed. Moreover, a batch cannot start before the *release date* of any job assigned to this batch. The *batch processing time* may not be shorter than the minimal processing time of any assigned job and must not be longer then any job's maximal processing time, as this could damage the produced components. Every job can only be assigned to a set of *eligible machines*. Moreover, machines have a maximal *capacity*, which may not be exceeded by the cumulative size of jobs in a single batch.

Setup times between consecutive batches must also be taken into account. Setup times depend on the ordered pair of attributes of the jobs in the respective batches and are independent of the machine assignments. In the context of heat treatment, this can be thought

[1] In the literature, the concept of attribute compatibility is often treated under the term of *incompatible job families*, see, e.g. [15].

of as the time required to switch from one temperature to another. Moreover, setup times before the first batch on every machine need to be taken into account. Setup times before first batches depend on the *initial state* of machines. Setup operations are scheduled immediately before every batch.

Furthermore, machines are only available during machine-dependent *availability intervals*, both batch processing times and setup times between batches need to be scheduled within these machine availability intervals. In practice, an interval for which a machine is unavailable can correspond to a period during which a machine is turned off or occupied with some other task, or to a period for which the personnel required for the setup and running of ovens is unavailable. For all described cases, the setup of machines for a given batch cannot be performed during an interval for which the machine is unavailable and must precede the batch immediately. Therefore, setup times also need to fall completely within the associated availability interval.

The objective of the OSP is to minimize the cumulative batch processing time, total setup costs, as well as the number of tardy jobs. These three objective components are combined in a single objective function using a linear combination, as is formalized in Section 3.2. As the minimization of job tardiness usually has the highest priority in practice, the tardiness objective has a higher weight than the other objectives.

In practice, the cumulative batch processing time should be minimized as the cost of running an oven depends merely on the processing time of the entire batch and not on the number of jobs within a batch. Therefore, running an oven containing a single small job incurs the same costs as running the oven filled to its maximal capacity.

Furthermore, we note that setup costs and setup times are not necessarily correlated. In fact, cooling down an oven from a high to a low temperature might not incur any (energy) costs, but still might require a certain amount of time. Setup costs are used to capture all costs that are related to the setup required between batches; e.g. costs related to personnel involved in the setup operation can also be captured by setup costs. Therefore, setup times are not included in the objective function, only setup costs are. However, up to a certain extent, setup times are implicitly minimized since they can have an impact on the tardiness of jobs.

Using the three-field notation introduced by Graham et al. [23], the OSP can be classified as $\tilde{P}|r_j, \bar{d}_j, maxt_j, b_i, ST_{sd,b}, SC_{sd,b}, \mathcal{E}_j, Av_m|$obj, where $\mathcal{E}_j$ stands for eligible machines and Av_m for availability of machines. A more formal description of the problem constraints and the objective function obj is given in Section 3.

As shown by Uzsoy [24], minimizing makespan on a single batch processing machine is an $\mathcal{NP}$-hard problem. This can be seen as follows: For the special case that all jobs have identical processing times, minimizing makespan on a single batch processing machine is equivalent to a bin packing problem where the bin capacity is equal to the machine capacity and item sizes are given by the job sizes. It follows that the OSP is $\mathcal{NP}$-hard as well, since it generalizes this problem. Indeed, minimizing makespan on a single batch processing machine is equivalent to minimizing batch processing times in an oven scheduling problem with a single machine, a single attribute, no setup times, and for which all jobs are available at the start of the scheduling horizon.

2.1 Instance parameters of the OSP

An instance of the OSP consists of a set $\mathcal{M} = \{1, \ldots, k\}$ of machines, a set $\mathcal{J} = \{1, \ldots, n\}$ of jobs and a set $\mathcal{A} = \{1, \ldots, a\}$ of attributes as well as the length $l \in \mathbb{N}$ of the scheduling horizon. Every machine $m \in \mathcal{M}$ has a maximum capacity c_m and an initial state $s_m \in \mathcal{A}$. The

machine availability times are specified in the form of time intervals $[as(m,i), ae(m,i)] \subseteq [0,l]$ where $as(m,i)$ denotes the start and $ae(m,i)$ the end time of the i-th interval on machine m. W.l.o.g. we assume that every machine has the same number of availability intervals and denote this number by I where some of these intervals might be empty (i.e. $as(m,i) = ae(m,i)$). Moreover, availability intervals have to be sorted in increasing order (i.e. $as(m,i) \leq ae(m,i) \leq as(m,i+1)]$ for all $i \leq I-1$).

Every job $j \in \mathcal{J}$ is specified by the following list of properties:

- A set of eligible machines $\mathcal{E}_j \subseteq \mathcal{M}$.
- An earliest start time (or release time) $et_j \in \mathbb{N}$ with $0 \leq et_j < l$.
- A latest end time (or due date) $lt_j \in \mathbb{N}$ with $et_j < lt_j \leq l$.
- A minimal processing time $mint_j \in \mathbb{N}$ with $min_T \leq mint_j \leq max_T$, where $min_T > 0$ is the overall minimum and $max_T \leq l$ is the overall maximum processing time.
- A maximal processing time $maxt_j \in \mathbb{N}$ with $mint_j \leq maxt_j \leq max_T$.
- A size $s_j \in \mathbb{N}$.
- An attribute $a_j \in \mathcal{A}$.

Moreover, an $(a \times a)$-matrix of setup times $st = (st(a_i, a_j))_{1 \leq a_i, a_j \leq a}$ and an $(a \times a)$-matrix of setup costs $sc = (sc(a_i, a_j))_{1 \leq a_i, a_j \leq a}$ are given to denote the setup times (resp. costs) incurred between a batch with attribute a_i and a subsequent batch with attribute a_j. Setup times (resp. costs) are integers in the range $[0, max_{ST}]$ (resp. $[0, max_{SC}]$), where $max_{ST} \leq l$ (resp. $max_{SC} \in \mathbb{N}$) denotes the maximal setup time (resp. maximal setup cost). Note that these matrices are not necessarily symmetric.

2.2 Objective function

The objective of the OSP is to minimize the following threefunctions:

- the *cumulative batch processing time* across all machines p
- the *number of tardy jobs* t
- and the *cumulative setup costs* sc.

The cumulative batch processing time is the sum of processing times of all batches, not of all jobs. Minimizing the cumulative batch processing time thus leads to batches being formed in the most efficient way. Setup costs are composed of the setup costs before the first batch on every machine, which are determined by the initial state of the machine and the attribute of the first batch, and the setup costs between consecutive batches, which are determined by the attributes of the batches.

Depending on the application scenario, each of these three objective components are given integer weights, and these weighted components are then combined in a single function by means of a linear combination. Using such weights also allows us to implement lexicographic optimization, as is detailed at the end of Section 3.2.

2.3 Example instance with six jobs

Consider the following example for an OSP instance consisting of six jobs, two attributes, two machines and a scheduling horizon of length $l = 15$. The instance parameters are summarized in the following tables and matrices:

Machine	M_1	M_2
c_m	100	150
s_m	1	2
Availability intervals $[as, ae]$	[0,6] [8,14]	[2,10] [11, 14]

$$st = \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$$

$$sc = \begin{pmatrix} 0 & 20 \\ 10 & 0 \end{pmatrix}$$

Job j	1	2	3	4	5	6
$\mathcal{E}_j$	M_1	M_1 M_2	M_1	M_1 M_2	M_2	M_2
et_j	2	0	0	3	0	2
lt_j	10	10	20	20	20	20
$mint_j$	3	3	3	5	5	5
$maxt_j$	3	5	5	8	8	10
s_j	40	60	30	50	50	50
a_j	2	2	1	1	1	1

Figure 1 visualises a solution of this instance consisting of three batches. In this visualisation, the dark grey areas correspond to time intervals for which the machine is not available, light gray rectangles before batches are setup times and the dashed lines represent the machine capacities. This solution is optimal with respect to the cumulative batch processing time ($p = 11$) and the tardiness ($t = 0$). It is however not optimal with respect to the setup costs: for this solution, the setup costs are $sc = 40$, but solutions with $sc = 30$ exist as well. However, lower setup costs can only be achieved to the detriment of job tardiness.

2.4 Construction heuristic

We designed a construction heuristic that can find initial solutions and can also serve as upper bound on the optimal solution cost. The heuristic is a *dispatching rule* that prioritizes jobs first by their earliest start date and then by their latest end date. Similar rules, such as the Earliest Due Date (EDD) dispatching rule presented by Uzsoy [25] for batch scheduling on a single machine with incompatible job families, have been proposed for other batch scheduling problems.

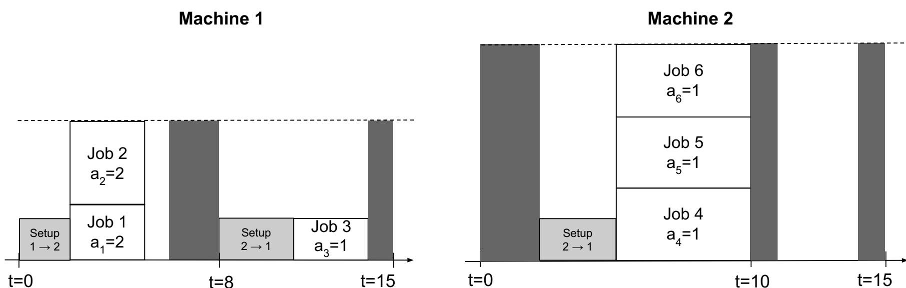

Fig. 1 An optimal solution to the OSP for a small example problem

The heuristic starts at time $t = 0$. At every time step t, the list of currently available machines is generated: these are all machines on which no batch is running at time t and for which t lies within a machine availability interval. For these available machines, the list of available jobs is generated: these are jobs that have not yet been scheduled, have already been released and can be processed on one of the available machines. Among these jobs, the one with the earliest due date is chosen. If there are several jobs that fulfill these conditions, the largest one is chosen first. In case of ties, the job with smallest index is selected. Let the selected job be $j \in \mathcal{J}$. Moreover, if there are several machines for which j fits into the current availability interval and that are eligible for j, the heuristic chooses the machine for which the setup time from the previous batch (or from the initial state of the machine) is minimal.

Once a job j is scheduled, the algorithm tries to add other jobs that are currently available to the same batch: For this, jobs are only considered if their attribute as well as minimal and maximal processing times are compatible with job j and the combined batch size does not exceed the machine capacity. Moreover, unless job j finishes too late anyway, other jobs are only considered if their processing time does not force j to finish late. If job j finishes late in the current schedule, all compatible jobs that are currently available are considered for addition to the batch. In case there are several jobs that meet these requirements, we sort them in decreasing order of their latest end dates and keep adding jobs to the batch as long as the machine capacity and compatibility requirements are fulfilled. If there are no jobs (left) that can be added to the batch and the maximum machine capacity is not reached yet, we look ahead and also consider jobs with an earliest start time $> t$. However, we only consider compatible jobs that do not force job j to finish late (unless this was already the case) and that can still be processed within the current machine availability interval. Once all jobs for the batch have been chosen, the batch is scheduled at the earliest possible time.

If there are other jobs that can be processed at time t, we schedule them and fill up their batches in the same way. If no job can be scheduled, the time is increased to $t + 1$ and the above procedure is repeated until the end of the scheduling horizon is reached or all jobs have been scheduled.

For the example instance presented in the previous section (Section 2.3), the construction heuristic delivers the optimal solution depicted in Fig. 1. It proceeds as follows: At time $t = 0$, machine 1 is available and jobs 2 and 3 are available for this machine. Since job 2 has an earlier latest end date, it is scheduled first. In order to respect the necessary setup times from the initial state of machine 1, job 2 is scheduled at time $t = 2$. To this batch, job 1 can be added. No other jobs can be scheduled at the time is increased to $t = 2$. Now, job 5 is selected and scheduled at time $t = 5$ on machine 2. To this batch, jobs 4 and 6 can be added. Finally, the time is increased until $t = 8$ and job 3 can be scheduled.

Note that this construction heuristic is not guaranteed to find a feasible solution for every instance of the OSP. sIndeed, due to the ordering of the jobs and the way that jobs are chosen to fill up batches, it is possible that the heuristic is not capable of scheduling all jobs within the scheduling horizon. In order to see this, consider the following small example with a single machine and two jobs: The scheduling horizon is $l = 3$ and the machine is available for the entire interval $[0, 3]$. The two jobs 1 and 2 have the same attribute and sizes are such that they can be processed together in a batch. The earliest start, latest end and processing times are given as follows: $et_1 = 0$ and $et_2 = 1$, $lt_1 = 2$ and $lt_2 = 3$, $mint_1 = mint_2 = maxt_1 = maxt_2 = 2$. The only feasible solution is to schedule both jobs at time $t = 1$, which leads to job 1 finishing late. The construction heuristic however schedules job 1 at time $t = 0$ and then fails to schedule job 2. Even though we cannot guarantee that the described construction heuristic finds a feasible solution for all instances

(where such a solution exists), we can guarantee that the partial solution provided by the heuristic, i.e., the solution restricted to those jobs that were scheduled, is be feasible. This is due to the fact that all constraints are fulfilled when jobs are scheduled or added to existing batches by the heuristic.

3 Formal problem definition and direct model for the OSP

In this section we provide a formal definition of the OSP that will also serve as an ILP model for the problem that can be directly implemented using the modelling language MiniZinc [26]. The full MiniZinc model is also available on Zenodo [21].

The advantage of the free and open-source constraint modeling language MiniZinc is that it can be used to model constraint optimization problems in a high-level, solver-independent way. This model is then compiled into FlatZinc, which is a solver input language that can be processed by a multitude of state-of-the-art solvers.

We first explain how batches are modeled and afterwards define decision variables, objective function, and the set of constraints.

The core idea is to model batches by their positions on machines. In the worst case we need as many batches as there are jobs; we thus define the set of potential batches as $\mathcal{B} = \{B_{1,1}, \ldots, B_{1,n}, \ldots, B_{k,1}, \ldots, B_{k,n}\}$ to model up to n batches for machines 1 to k. In order to break symmetries in the model, we further enforce that batches are sorted in ascending order of their start times and empty batches are scheduled at the end. That is, $B_{m,b+1}$ is the batch following immediately after batch $B_{m,b}$ on machine m for $b \leq n-1$. Clearly, at most n of the $k \cdot n$ potential batches will actually be used and the rest will remain empty.

3.1 Variables

We define the following decision variables:

- Machine and batch number assigned to job: $X_{m,b,j} \in \{0,1\} \quad \forall m \in \mathcal{M} \quad \forall b \in [1,n] \quad \forall j \in \mathcal{J}$
  The binary variables $X_{m,b,j}$ encode whether job j is assigned to batch $B_{m,b}$, i.e., $X_{m,b,j} = 1$ if and only if job j is assigned to batch b on machine m.
- Start times of batches: $S_{m,b} \in [0,l] \subset \mathbb{N} \quad \forall m \in \mathcal{M} \quad \forall b \in [1,n]$
- Processing times of batches: $P_{m,b} \in [0, max_T] \subset \mathbb{N} \quad \forall m \in \mathcal{M} \quad \forall b \in [1,n]$

To handle empty batches, we define an additional attribute with value 0 and extend the matrices of setup times $\overline{st}$ and setup costs $\overline{sc}$ so that no costs occur when transitioning from an arbitrary batch to an empty batch: $\overline{st}(a_i, a_j) = \overline{sc}(a_i, a_j) = 0$ if $a_i = 0$ or a_j=0. Moreover, we add a machine availability interval $[l,l]$ of length 0 to the list of availability intervals so that empty batches can be scheduled for this interval (the maximum number of intervals per machine therefore becomes $I+1$).

We then define the following auxiliary variables:

- Availability interval for batch: $I_{m,b,i} \in \{0,1\} \quad \forall m \in \mathcal{M} \quad \forall b \in [1,n] \quad \forall i \in [1, I+1]$
  The binary variables $I_{m,b,i}$ encode whether batch b on machine m is scheduled within the machine's i-th availability interval.
- Attribute of batch: $A_{m,b} \in [0,a] \quad \forall m \in \mathcal{M} \quad \forall b \in [1,n]$

- Tardy jobs: $T_{m,b,j} \in \{0, 1\} \quad \forall m \in \mathcal{M} \forall b \in [1, n] \forall j \in \mathcal{J}$
  The binary auxiliary variables $T_{m,b,j}$ encode whether job j in batch $B_{m,b}$ finishes after its latest end date.
- Setup times and costs: $st_{m,b} \in [0, max_{ST}]$ and $sc_{m,b} \in [0, max_{SC}] \forall m \in \mathcal{M} \quad \forall b \in [1, n]$ for setup times and costs between batch b on machine m and the following batch.
- Empty batches: $E_{m,b} \in \{0, 1\} \quad \forall m \in \mathcal{M} \quad \forall b \in [1, n]$
  The binary variables $E_{m,b}$ encode whether batch b on machine m is empty.

3.2 Objective function

Recall that the three components of the objective function are the cumulative batch processing time across all machines p, the number of tardy jobs t and the cumulative setup costs sc. They are formally defined as follows:

$$
\begin{aligned}
p &= \sum_{m \in \mathcal{M}} \sum_{b=1}^{n} P_{m,b} \\
sc &= \sum_{m \in \mathcal{M}} \overline{sc}(s_m, A_{m,1}) + \sum_{m \in \mathcal{M}} \sum_{b=1}^{n-1} sc_{m,b} \\
t &= \sum_{j \in \mathcal{J}} \sum_{m \in \mathcal{M}} \sum_{b=1}^{n} T_{m,b,j}
\end{aligned} \tag{1}
$$

Note that in the definition of the setup costs, the first sum refers to the setup costs from the initial state of a machine to its first batch and the second sum refers to the costs incurred between batches.

Normalization of cost components Finding a trade-off among several objectives is the topic of multiobjective optimization [27, 28]. In practice, lexicographic optimization is often chosen. For our specific application, we wanted an approach that allows high flexibility and lets users decide on the importance of each objective. We thus decided in consultation with our industrial partner to define the objective function as a linear combination of the three components. This approach also enables lexicographic optimization, by setting the weights accordingly as will be explained at the end of this subsection. Also note that all solutions that are optimal with respect to a weighted sum of the objective components are Pareto-optimal [29] with respect to the three objective components. However we cannot guarantee to find all Pareto-optimal solutions using this type of objective function.

In order to combine the three objective components using a linear combination, p, sc and t need to be normalized to $\tilde{p}$, $\tilde{sc}$ and $\tilde{t} \in [0, 1]$ so that we can guarantee that all three components take values in the same range:

$$
\begin{aligned}
\tilde{p} &= \frac{p}{avg_t \cdot n} \text{with } avg_t = \lceil \frac{\sum_{j \in \mathcal{J}} mint_j}{n} \rceil \\
\tilde{sc} &= \frac{sc}{\max(max_{SC}, 1) \cdot n} \\
\tilde{t} &= \frac{t}{n}
\end{aligned} \tag{2}
$$

In the worst case, every batch processes a single job. In this case the total batch processing time is n times the average job processing time avg_t. The setup costs are bounded by the maximum setup cost multiplied with the number of jobs. We take the maximum of 1 and

max_{SC} since it is possible that $max_{SC} = 0$. The number of tardy jobs is clearly bounded by the total number of jobs.

Finally, the objective function obj is a linear combination of the three normalized components:

$$\text{obj} = (w_p \cdot \tilde{p} + w_{sc} \cdot \tilde{sc} + w_t \cdot \tilde{t})/(w_p + w_{sc} + w_t) \quad \in [0, 1] \subset \mathbb{R} \tag{3}$$

where the weights w_p, w_{sc} and w_t take integer values.

Integer-valued objective As some state-of-the-art CP solvers can only handle integer domains, we propose an alternative objective function obj′, where we additionally multiply $\tilde{p}$, $\tilde{sc}$, and $\tilde{t}$ by the number of jobs and the least common multiple of avg_t and max_{SC}:

$$\begin{aligned} \text{obj}' &= C \cdot n \cdot (w_p + w_{sc} + w_t) \cdot \text{obj} \quad \in \mathbb{N} \\ &= \frac{w_p \cdot C}{avg_t} \cdot p + \frac{w_{sc} \cdot C}{\max(max_{SC}, 1)} \cdot sc + w_t \cdot C \cdot t, \\ \text{where } C &= \texttt{lcm}(avg_t, \max(max_{SC}, 1)). \end{aligned} \tag{4}$$

Preliminary experiments using the MIP solver Gurobi showed that using obj and obj′ both lead to similar results. We therefore used only obj′ in our final experimental evaluation.

Lexicographic optimization The use of integer weights for each of the objective components also allows us to express the OSP as a lexicographic minimization problem. This can be done using so-called "big M"-constants. In order to explain this in more detail, let us denote by $\text{obj}_1 \in \{p, sc, t\}$ the objective component with highest lexicographic importance, obj_2 the component with second-highest importance and with obj_3 the component with lowest importance. Lexicographic optimality is defined using the lexicographic order $<_{lex}$ as follows:

$$\begin{aligned} & s_1 <_{lex} s_2 \\ \iff \quad & \text{obj}_j(s_1) < \text{obj}_j(s_2) \\ & \text{with } j \in \{1, 2, 3\} \text{ the smallest index such that } \text{obj}_j(s_1) \neq \text{obj}_j(s_2) \end{aligned}$$

and where s_1 and s_2 are two feasible solutions to the OSP. Lexicographic minimization can be achieved by minimizing the integer-valued objective function obj' with weights set as follows:

$$\begin{aligned} \text{obj'} &= w_1 \cdot \text{obj}_1 + w_2 \cdot \text{obj}_2 + w_3 \cdot \text{obj}_3 \\ w_3 &= 1 \\ w_2 &= ub(\text{obj}_3) + 1 \\ w_1 &= w_2 \cdot (ub(\text{obj}_2) + 1), \end{aligned}$$

where the upper bounds of the objective components are calculated as in the normalization of components (see (2)), i.e.,

$$\begin{aligned} ub(p) &= avg_t \cdot n \\ ub(sc) &= \max(max_{SC}, 1) \cdot n \\ ub(t) &= n. \end{aligned}$$

Note that the calculation of the least common multiple as in (4) is not necessary here. The corresponding normalized objective function obj is then given as follows:

$$\text{obj} = \frac{\text{obj}'}{ub(\text{obj})} = \frac{\text{obj}'}{w_1 \cdot ub(\text{obj}_1) + w_2 \cdot ub(\text{obj}_2) + ub(\text{obj}_3)} \in [0, 1].$$

For a concrete example of lexicographic minimization, see the description of use case UC2 in Section 6.1.2.

3.3 Constraints

In what follows, we formally define the constraints of the OSP.[2]

- Each job is assigned to exactly one batch:

$$\sum_{m \in \mathcal{M}} \sum_{b=1}^{n} X_{m,b,j} = 1 \quad \forall j$$

- Each job is assigned to exactly one eligible machine:

$$\sum_{m \in \mathcal{E}_j} \sum_{b=1}^{n} X_{m,b,j} = 1 \quad \forall j$$

- Batches may not start before the earliest start of any job in the batch:

$$S_{m,b} \geq et_j \cdot X_{m,b,j} \quad \forall m, \forall b, \forall j$$

- Batch processing times must lie between the minimal and maximal processing time of all assigned jobs:

$$\begin{aligned} &mint_j \cdot X_{m,b,j} \leq P_{m,b} \quad \wedge \\ &P_{m,b} \leq maxt_j \cdot X_{m,b,j} + max_T \cdot (1 - X_{m,b,j}) \qquad \forall m, \forall b, \forall j \end{aligned}$$

- Batches on the same machine may not overlap and setup times must be considered between consecutive batches:

$$S_{m,b} + P_{m,b} + st_{m,b} \leq S_{m,b+1} \quad \forall m, \forall b \leq n-1,$$

- Batches and the preceding setup times must lie entirely within one machine availability interval:

$$as(m,i) \cdot I_{m,b,i} \leq S_{m,b} \quad \wedge$$

$$S_{m,b} \leq ae(m,i) \cdot I_{m,b,i} + l \cdot (1 - I_{m,b,i}) \qquad \forall m, \forall b, \forall i \tag{5}$$

$$\sum_{1 \leq i \leq I+1} I_{m,b,i} = 1 \qquad \forall m, \forall b \tag{6}$$

$$as(m,i) \cdot I_{m,b,i} \leq S_{m,b} - st_{m,b-1} \qquad \forall m, \forall b \geq 2, \forall i \tag{7}$$

$$as(m,i) \cdot I_{m,1,i} \leq S_{m,1} - st(s_m, A_{m,1}) \qquad \forall m, \forall i \tag{8}$$

$$S_{m,b} + P_{m,b} \leq ae(m,i) \cdot I_{m,b,i} + l \cdot (1 - I_{m,b,i}) \ \forall m, \forall b, \forall i \tag{9}$$

The binary auxiliary variables $I_{m,b,i}$ in constraint (5) encode whether batch b on machine m is scheduled within the i-th availability interval $[as(m,i), ae(m,i)]$ of machine m. Therefore, if $I_{m,b,i} = 1$, it must hold that $as(m,i) \leq S_{m,b} \leq ae(m,i)$. The redundant constraint (6) ensures that every batch is scheduled within exactly one availability interval. Constraints (7)– (9) ensure that the entire processing time of batch $B_{m,b}$ as well as the preceding setup times $st_{m,b-1}$ lie within a single availability interval. Constraint (8) takes care of the special case of the first batch on a machine.

[2] If not stated otherwise, $\forall m$ is short for $\forall m \in \mathcal{M}$, $\forall b$ for $\forall b \in [1,n]$, $\forall i$ for $\forall i \in [1, I+1]$ and $\forall j$ for $\forall j \in \mathcal{J}$.

- Total batch size must be less than machine capacity:

$$\sum_{j \in \mathcal{J}} s_j \cdot X_{m,b,j} \leq c_m \quad \forall m, \forall b$$

- Jobs in one batch must have the same attribute, which we model with auxiliary variables $A_{m,b}$ to set the attribute of a batch:

$$\begin{aligned} & a_j \cdot X_{m,b,j} \leq A_{m,b} \quad \wedge \\ & A_{m,b} \leq a_j \cdot X_{m,b,j} + a \cdot (1 - X_{m,b,j}) \qquad \forall m, \forall b, \forall j, \end{aligned}$$

where a is the total number of attributes. It thus has to hold that $a_j = A_{m,b}$ if job j is scheduled in batch $B_{m,b}$ $0 \leq A_{m,b} \leq a$ if job j is not scheduled in batch $B_{m,b}$.

- Helper variables for tardy jobs:

$$T_{m,b,j} \leq X_{m,b,j} \qquad \forall j, \forall m, \forall b \tag{10}$$

$$S_{m,b} + P_{m,b} \leq (X_{m,b,j} - T_{m,b,j}) \cdot (lt_j - l) + l \quad \forall j, \forall m, \forall b \tag{11}$$

$$S_{m,b} + P_{m,b} + (1 - T_{m,b,j}) \cdot (l + 1) > lt_j \quad \forall j, \forall m, \forall b \tag{12}$$

The binary auxiliary variables $T_{m,b,j}$ encode whether job j in batch b on machine m finishes after its latest end date and is used to calculate the number of tardy jobs t. Constraint (10) ensures that $T_{j,m,b} = 1$ is only possible if job j is assigned to batch b on machine m. If $T_{j,m,b} = 0$, job j must finish before lt_j (Constraint (11)) and if $T_{j,m,b} = 1$, it must hold that $S_{m,b} + P_{m,b} > lt_j$ (Constraint (12)).

- Setup times and costs between consecutive batches on the same machine:

$$\begin{aligned} st_{m,b} &= \bar{st}(A_{m,b}, A_{m,b+1}) \ \forall m, \forall b \leq n-1 \\ sc_{m,b} &= \bar{sc}(A_{m,b}, A_{m,b+1}) \ \forall m, \forall b \leq n-1 \end{aligned}$$

The helper variables $st_{m,b}$ (resp. $sc_{m,b}$) encode the setup time (resp. setup cost) between batch b on machine m and the following batch on the same machine. Note that setup times and costs are also defined between empty batches and set to 0 in this case, as the setup costs and times from attribute 0 to attribute 0 are equal to 0.

- Set decision variables for empty batches:

$$\textstyle\sum_{j \in \mathcal{J}} X_{m,b,j} \geq 1 - E_{m,b} \quad \forall m, \forall b \tag{13}$$

$$X_{m,b,j} \leq 1 - E_{m,b} \quad \forall m, \forall b, \forall j \tag{14}$$

$$S_{m,b} \geq l \cdot E_{m,b} \quad \forall m, \forall b \tag{15}$$

$$P_{m,b} <= max_T \cdot (1 - E_{m,b}) \ \forall m, \forall b \tag{16}$$

$$E_{m,b} <= I_{m,b,I+1} \quad \forall m, \forall b \tag{17}$$

$$A_{m,b} <= a \cdot (1 - E_{m,b}) \quad \forall m, \forall b \tag{18}$$

$$E_{m,b} <= E_{m,b+1} \quad \forall m, \forall b \leq n-1 \tag{19}$$

The binary helper variables $E_{m,b}$ encode whether batch b on machine m is empty or not. Constraints (13) and (14) ensure that $E_{m,b} = 1$ iff no job is scheduled for batch $B_{m,b}$. The constraints (15) to (17) set the start times, processing times, availability intervals, and attributes for empty batches: $S_{m,b} = l$, $P_{m,b} = 0$, $A_{m,b} = 0$ and $I_{m,b} = I + 1$. Moreover, in order to break symmetries, the list of batches $(B_{m,b})_{1 \leq b \leq n}$ per machine $m \in \mathcal{M}$ is sorted so that all non-empty batches appear first (constraint (19)).

4 Interval variable model using representative jobs for batches

For the model presented in Section 3, we decided to create decision variables for start times and processing times of every batch that could possibly be present ($n \cdot k$ batches in total). Which batch a job is assigned to was then encoded using additional decision variables. The scheduling constraints were formulated for the batches.

An alternative approach is to schedule jobs instead of batches. However, only one *representative job* for every batch is scheduled; all other jobs in the same batch point to their representative job. In order to break symmetries, the job with lowest index is chosen to be the representative job for its batch. Whereas the previous approach models the OSP primarily as an assignment problem (jobs are assigned to batches that have a predefined order), this approach focuses more on the scheduling aspect of the OSP: a subset of representative jobs are chosen for which an optimal schedule needs to be found. Such a modelling approach has been used previously in the literature for batch scheduling problems, e.g. by Kosch et al. [10] who speak of *host jobs* instead of representative jobs.

Since CP Optimizer is particularly well suited for scheduling problems as demonstrated by Laborie et al. [30], we formulated this modelling approach based on representative jobs as a CP model for IBM ILOG CPLEX Studio. This model is written using the Optimization Programming Language (OPL) [31] and makes use of *interval variables*, as well as other concepts that are specific to CP Optimizer. The full OPL model is also available on Zenodo [21]. In addition, we modelled this approach using the MiniZinc modelling language in order to be able to evaluate the performance of different solvers. This alternative solver-independent CP formulation is briefly described in Section 5.2.

Before we describe the model, we give a brief explanation of the use of interval variables in OPL. An interval variable int is a decision variable that can either be present or absent in a solution. In case it is present, it is characterized by a start time $Start(int)$, an end time $End(int)$ and a size $size(int)$. An interval variable is defined by a tuple ($\texttt{optional}$, $[StartMin, EndMax]$, $[SZMin, SZMax]$, F) or ($\texttt{optional}$, $[StartMin, EndMax]$, $[SZMin, SZMax]$), where the parameters can take the following values:

- $\texttt{optional} \in \{\top, \bot\}$ indicates whether the interval variable is optional or not. Optional interval variables are used to model tasks that can be left unperformed in the schedule,
- the *time window* $[StartMin, EndMax] \subseteq \mathbb{N}$ restricts the start and end time of the interval variable, i.e., $StartMin \leq Start(int)$ and $End(int) \leq EndMax$.
- the *size range* $[SZMin, SZMax] \subseteq \mathbb{N}$ restricts the size of the interval variable, i.e., $SZMin \leq size(int) \leq SZMax$,
- the *intensity function* F, is a step-function taking integer values. Intensity functions are used to model time periods during which tasks cannot be (fully) performed. For the OSP, intensity functions can be used to model the machine availability times.

For a more formal definition of the used CP Optimizer concepts, see [30], and for more information on how they can be used, see the tutorial [32].

We define the following decision variables for the OSP:

- Optional interval variables for jobs:

$$\text{interval job}_j : (\top, [et_j, l], [mint_j, maxt_j]) \quad \forall j \in \mathcal{J}.$$

  The variable job_j is only present if job j is the representative job of its batch. Note that the time window $[et_j, l]$ ensures that every representative job starts after its earliest start time et_j.

- Optional interval variables for jobs on machines:

$$\text{interval jobM}_{j,m} : (\top, [et_j, l], [mint_j, maxt_j], av_m) \quad \forall j \in \mathcal{J} \quad \forall m \in \mathcal{M}.$$

  The variable $\text{jobM}_{j,m}$ is only present if job j is assigned to machine m and is representative for its batch. The intensity function av_m encodes the machine availability times and is modeled using *intensity step functions*; $av_m(t) = 100\%$ if machine m is available at time t (the job can be fully performed by the machine) and $av_m(t) = 0\%$ otherwise (the job cannot be performed by the machine).
- Pointers to representative jobs in the same batch:

$$\text{inBatchWith}_j \in [0, n] \quad \forall j \in \mathcal{J}.$$

  If $\text{inBatchWith}_j = i$ for some job $i \in \mathcal{J}$, job j is scheduled in the same batch as job i and job i is representative for this batch. If $\text{inBatchWith}_j = 0$, job j is representative for its batch.
- In order to establish the order of jobs, respectively batches, on machines, we make use of a type of decision variable called *interval sequence variable* in OPL. An interval sequence variable seq is defined by a tuple (S, T) where S is a set of interval variables and $T \subseteq \mathbb{N}$ with $|T| = |S|$ is a set of so-called integer *types*. The value of an interval sequence variable seq is a permutation of the interval variables in S (any absent interval variables are ignored). We define the following interval sequence variables:

$$\text{sequence mach}_m : (\{\text{jobM}_{j,m} : j \in \mathcal{J}\}, \{a_j : j \in \mathcal{J}\}) \quad \forall m \in \mathcal{M}.$$

  The value of mach_m is thus a permutation of the interval variables $\text{jobM}_{j,m}$. The types are used to model the attribute of jobs (the attribute of a batch containing job j is equal to a_j). This allows us to formulate scheduling constraints such as `noOverlap` and to use functions such as `endOfPrev` and `typeOfPrev` to formulate constraints on the start times of batches and setup times between batches.
- Setup times between batches are also modelled using optional interval variables:

$$\text{interval setup}_{j,m} : (\top, [0, l], [0, max_{ST}], av_m) \quad \forall j \in \mathcal{J} \quad \forall m \in \mathcal{M},$$

  where max_{ST} denotes the overall maximum setup time. The variable $\text{setup}_{j,m}$ is used to model the setup operation before $\text{jobM}_{j,m}$. Setup times need to be modelled in addition to jobs themselves in order to ensure that setup times fall within the same machine availability interval as the following jobs.
- A binary helper variable is used to model tardy jobs:

$$\text{tardy}_j \in \{0, 1\} \quad \forall j \in \mathcal{J}.$$

For the formulation of the objective function as well as the constraints, we make use of the OPL function `typeOfPrev`. For a sequence interval variable s, an interval variable i, and two integers b and c, `typeOfPrev`(s, i, b, c) returns the type (=attribute) of the interval variable that is previous to i in the sequence s. When i is present and is the first interval in s, the function returns the constant integer value b. When i is absent, the function returns the

constant integer value c. We then model the OSP as follows:

$$\text{Min.obj'} = \tilde{w_p} \cdot p + \tilde{w_{sc}} \cdot sc + \tilde{w_t} \cdot t, \text{ with}$$
$$p = \sum_{\substack{m \in \mathcal{M} \\ j \in \mathcal{J}}} \texttt{lengthOf}(\text{jobM}_{j,m}), \qquad t = \sum_{j \in \mathcal{J}} \text{tardy}_j,$$
$$sc = \sum_{\substack{m \in \mathcal{M} \\ j \in \mathcal{J}}} sc_{prev(j,m),a_j}, \tag{20}$$

where $prev(j, m) = \texttt{typeOfPrev}(\text{mach}_m, \text{jobM}_{j,m}, s_m, 0)$.

$$\text{s.t.} \quad \texttt{alternative}(\text{job}_j, \{\text{jobM}_{j,m} : m \in \mathcal{M}\}) \quad \forall j \tag{21}$$
$$\texttt{alternative}(\text{job}_j, \{\text{jobM}_{j,m} : m \in \mathcal{E}_m\}) \quad \forall j \tag{22}$$
$$(\texttt{presenceOf}(\text{job}_j) \wedge \text{inBatchWith}_j = 0) \vee (\neg\texttt{presenceOf}(\text{job}_j) \wedge \text{inBatchWith}_j > 0) \quad \forall j \tag{23}$$
$$\text{inBatchWith}_j = i \Rightarrow \texttt{presenceOf}(\text{job}_i) \wedge i < j \quad \forall i, j \in \mathcal{J} \tag{24}$$
$$\text{inBatchWith}_j = i \Rightarrow a_i = a_j \wedge mint_j \leq \texttt{lengthOf}(\text{job}_i) \leq maxt_j \wedge \texttt{startOf}(\text{job}_i) \geq et_j \wedge \sum_{m \in \mathcal{E}_j} \texttt{presenceOf}(\text{jobM}_{i,m}) = 1 \quad \forall i, j \in \mathcal{J} \tag{25}$$
$$\texttt{presenceOf}(\text{jobM}_{j,m}) \cdot (s_j + \sum_{\substack{i \in \mathcal{J} \text{ with} \\ \text{inBatchWith}_i = j}} s_i) \leq c_m \quad \forall m, \forall j \tag{26}$$
$$\texttt{noOverlap}(\text{mach}_m, st) \quad \forall m \tag{27}$$
$$\texttt{presenceOf}(\text{jobM}_{j,m}) = \texttt{presenceOf}(\text{setup}_{j,m}) \quad \forall m, \forall j \tag{28}$$
$$\texttt{endAtStart}(\text{setup}_{j,m}, \text{jobM}_{j,m}) \quad \forall m, \forall j \tag{29}$$
$$\texttt{endOfPrev}(\text{mach}_m, \text{jobM}_{j,m}, 0) \leq \texttt{startOf}(\text{setup}_{j,m}) \quad \forall m, \forall j \tag{30}$$
$$\texttt{lengthOf}(\text{setup}_{j,m}) = st_{a_i,a_j}, \text{ where } a_i = \texttt{typeOfPrev}(\text{mach}_m, \text{jobM}_{j,m}, s_m, 0) \quad \forall m, \forall j \tag{31}$$
$$\texttt{forbidExtent}(\text{jobM}_{j,m}, av_m) \quad \forall m, \forall j \tag{32}$$
$$\texttt{forbidExtent}(\text{setup}_{j,m}, av_m) \quad \forall m, \forall j \tag{33}$$
$$(\texttt{presenceOf}(\text{job}_j) \wedge \texttt{endOf}(\text{job}_j) > lt_j) \vee (\text{inBatchWith}_j = i \wedge \texttt{endOf}(\text{job}_i) > lt_j) \Rightarrow \text{tardy}_j = 1 \quad \forall i, j \in \mathcal{J} \tag{34}$$
$$\sum_{m \in \mathcal{M}, j \in \mathcal{J}} \texttt{presenceOf}(\text{jobM}_{j,m}) \leq n \tag{35}$$

The alternative constraint in (21) ensures that every job j, whenever present, is scheduled on exactly one machine and that the interval variables job_j and $\text{jobM}_{j,m}$ are synchronised, where m is the machine j is assigned to. Constraint (22) ensures that every job j, whenever present, is scheduled on an eligible machine. Constraints (23) and (24) ensure

that every job is either scheduled or points at some other job that is scheduled and has a lower index.

Constraint (25) concerns the conditions that need to be fulfilled for all non-representative jobs in a batch: every job needs to have the same attribute as the representative job for the batch, the length of the interval variable corresponding to the representative job must be greater than or equal to the minimum processing time and smaller than or equal to than the maximum processing time of every job in the batch and the start of the representative job may not before the earliest start time of any other job in the batch. Note that for representative jobs, the conditions on the processing times and the earliest start time are fulfilled by definition of the job interval variables. However, for all non-representative jobs in a batch, these conditions need to be stated explicitly in the form of constraints. Moreover, it is ensured that jobs can only be processed together with another job if the assigned machine is eligible. Constraint (26) guarantees that machine capacities are not exceeded.

The `noOverlap`-constraint in equation (27) ensures that representative jobs, i.e. batches, on the same machine do not overlap. Passing the setup time matrix st as additional parameter to the `noOverlap`-constraint enforces a minimal distance between consecutive jobs based on the types (=attributes) of the jobs. This constraint is sufficient to ensure that enough time is left between batches to allow for the setup operation to be performed. However, we also need to guarantee that the setup operation falls within the same machine availability interval as the following batch.

Therefore, setup times are also modelled as interval variables and have to fulfill constraints (28)–(31): the setup time interval before job j on machine m is present if and only if job j is present on machine m (constraint (28)), setup times end exactly at the start of the following job (constraint (29)) and start after the preceding job on the same machine, or at a time ≥ 0 if the following job is the first on this machine (constraint (30)). The length of the setup time interval is given by the entry $st(a_i, a_j)$ in the setup time-matrix, where a_i is the attribute of the preceding and a_j of the following job on the machine. Constraints (32) and (33) finally force jobs and setup times to fall entirely within a machine availability interval of the assigned machine: whenever $\text{jobM}_{j,m}$ or $\text{setup}_{j,m}$ is present, it cannot overlap a point t where $av_m(t) = 0$.

The helper variable encoding tardy jobs is set via constraint (34). Finally, (35) is a redundant constraint that states that the total number of batches is smaller or equal to the total number of jobs.

5 Alternative model implementations, search strategies and warm-start

To make a thorough comparison of our models possible, we also implemented the direct model presented in Section 3 using OPL and created a solver-independent MiniZinc model with representative jobs per batch based on the OPL model presented in Section 4. We briefly describe these two alternative implementations in the following. The full models are available on Zenodo [21].

5.1 Alternative interval variable model using batch positions

This model is based on the direct model using batch positions as described in Section 3.

We use optional interval variables for all possible batches:

$$B_{m,b} : (\top, [0, l], [min_T, max_T], av_m) \text{ for all } m \in \mathcal{M}, b \in [1, n],$$

where min_T is the overall minimum and max_T is the overall maximum processing time). Batches are optional since not all $k \cdot n$ batches will actually be used. Setup times between batches are also modelled using optional interval variables: $st_{m,b}$: $(\top, [0, l], [0, max_{ST}], av_m)$ for all $m \in \mathcal{M}$ and all possible batch positions $b \in [1, n]$ (recall that max_{ST} is the overall maximum setup time).

Besides these interval variables, we use the same decision variables as in Section 3: $X_{m,b,j} \in \{0, 1\}$ to encode whether job j is assigned to batch $B_{m,b}$, $A_{m,b} \in [0, a]$ for the attribute of batch $B_{m,b}$ and $sc_{m,b}$ for setup costs.

The following constraints are used for the batch and setup time interval variables:

- `presenceOf` ensures that a batch is present iff some job is assigned to it. Similarly, setup times are present iff the preceding and following batch are present.
- `endBeforeStart` is used to enforce the order of batches and setup times on the same machine and `endAtStart` is used to schedule setup times exactly before the following batch.
- `startOf` restricts the start time of batches to be after the earliest start date of any assigned job and `lengthOf` is used to enforce that batch processing times lie between the minimal and maximal processing time for every assigned job
- `noOverlap` is used as a redundant constraint to ensure that batches on the same machine do not overlap
- `forbidExtent` is used to guarantee that batches and setup times are scheduled entirely within one machine availability interval: whenever $B_{m,b}$ or $st_{m,b}$ is present, it cannot overlap a point t where $av_m(t) = 0$.

5.2 Alternative direct model with representative jobs per batch

This model is based on the OPL model presented in detail in Section 4 and is implemented using the high-level constraint modeling language MiniZinc [26].

In MiniZinc, interval variables are not available. Instead, we model (optional) start times of jobs and processing times of jobs in separate decision variables. Optional integer variables are available in MiniZinc as well and can either take an integer value or the value <>, indicating that the variable is absent. We define the following decision variables:

- Optional start time of jobs and optional start times of jobs on machines: $\text{start}_j \in [0, l]$ and $\text{startM}_{j,m} \in [0, l] \quad \forall j \in \mathcal{J} \quad \forall m \in \mathcal{M}$. (this variable is absent for jobs which are not representative or not assigned to the given machine).
- Processing times of jobs and processing times of jobs on machines: $\text{proc}_j \in [0, max_T]$ and $\text{procM}_{j,m} \in [0, max_T] \quad \forall j \in \mathcal{J} \quad \forall m \in \mathcal{M}$.
- Optional pointers to representative jobs in the same batch: $\text{inBatchWith}_j \in \mathcal{J} \quad \forall j \in \mathcal{J}$
- Optional variables for the length of setup times before batches: $\text{setup}_{j,m} \in [0, max_{ST}] \quad \forall j \in \mathcal{J} \quad \forall m \in \mathcal{M}$.

Moreover, we use the following auxiliary variables to handle machine availability intervals and to define setup times between batches:

- Optional variables for the availability interval a job is scheduled to an a given machine: $I_{j,m} \in [I] \quad \forall j \in \mathcal{J} \quad \forall m \in \mathcal{M}$.

- Attribute of previous job on the same machine: $\text{attPrev}_{j,m} \in [0, a] \quad \forall j \in \mathcal{J} \quad \forall m \in \mathcal{M}$ (is equal to 0 if there is no previous job).
- End time of previous job on the same machine: $\text{endPrev}_{j,m} \in [0, l] \quad \forall j \in \mathcal{J} \quad \forall m \in \mathcal{M}$ (is equal to 0 if there is no previous job).
- Binary variables encoding whether a job is the first one on its assigned machine: $\text{first}_{j,m} \in \{0, 1\} \quad \forall j \in \mathcal{J} \quad \forall m \in \mathcal{M}$.

We make use of the following global constraints:

- `alternative` to ensure that every job that is present, i.e., every representative job, is scheduled to exactly one machine,
- `disjunctive` to enforce that representative jobs on the same machine do not overlap.

5.3 Alternative interval variable model using state functions in OPL

Another way of modeling batches in CP Optimizer is with the help of so-called *state function variables* [30], as done, e.g., by Ham et al. [33] for a flexible job shop scheduling problem with parallel batch processing machines. These functions model the time evolution of a value that can be required by interval variables; for the OSP, state functions could model the attribute of jobs. Two interval variables requiring incompatible states cannot overlap and interval variables requiring compatible states can be batched together. That is, state functions can be used to model that jobs can only be processed together in a batch if they have the same attribute. We developed an alternative OPL model for the OSP using state functions; it is available on Zenodo [21]. However, preliminary experiments showed that this model is not competitive with the one using representative jobs presented in Section 4. We therefore did not include it in the experimental evaluation presented in this paper.

5.4 Alternative direct model using MiniZinc's high-level CP modeling notation and batch positions

We also created a direct model using the modeling approach based on batch positions as described in Section 3 and MiniZinc's high-level CP modeling notation to formulate the constraints. This has the advantage that constraints do not need to be manually linearised and the model is somewhat easier to read for non-experts. In the formulation of the constraints, we implicitly make use of constraint reification to express conditional sums and additionally use the maximum global constraint. Furthermore, we implicitly utilize the element constraint to use variables as array indices.

As an example, let us present the constraints for batch start and processing times. We use the decision variables $M_j \in \mathcal{M}$ for the machine job j is assigned to and $B_j \in [1, n]$ for the position of the assigned batch. Then the constraint stating that jobs may not start before their earliest start time can be formulated as follows:

$$S_{M_j,B_j} \geq et_j \quad \forall j \in \mathcal{J}.$$

The constraint on batch processing times can be stated as:

$$P_{m,b} = \max(mint_j : j \in \mathcal{J} \text{ with } B_j = b \wedge M_j = m) \quad \forall m \in \mathcal{M}, b \in [b_m]$$
$$P_{M_j,B_j} \leq maxt_j \quad \forall j \in \mathcal{J}$$

The full model can again be found on Zenodo [21]. and is referred to as `direct_cp_model_simpler.mzn`. We decided not to include this model in our exper-

imental evaluation, since the results in our previous publication [22] showed that–in terms of the number of solutions found and the quality of solutions–this model cannot achieve better results than the ILP model from Section 3 for any of the evaluated solvers. Since both models are based on the same modelling approach, we decided to include only the stronger one of the two in this paper.

5.5 Programmed search strategies

For our solver-independent MiniZinc models, we implemented several programmed search strategies, which are based on variable- and value selection heuristics. Such heuristics determine the order of the explored variable and value assignments for a CP solver and can play a critical role in reducing the search space that needs to be enumerated by the solver. For our experiments, we implemented the search strategies directly in the MiniZinc language using search annotations.

Variable ordering In our implemented search strategies we first select an auxiliary variable that captures the total number of batches. For this variable we always use a minimum value first heuristic to encourage the solver to look for low cost solutions early in the search. Afterwards, we sequentially select decision variables related to a job by assigning the associated batch, machine, batch start time, and batch duration for the job (i.e., $B_1, M_1, S_{(M_1,B_1)}, P_{(M_1,B_1)}, \ldots, B_{|\mathcal{J}|}, M_{|\mathcal{J}|}, S_{(M_{|\mathcal{J}|},B_{|\mathcal{J}|})}, P_{(M_{|\mathcal{J}|},B_{|\mathcal{J}|})}$). For the CP model using representative jobs briefly presented in Section 5.2, we select the jobs' start times and processing times (i.e., $\text{start}_1, \text{proc}_1, \ldots, \text{start}_{|\mathcal{J}|}, \text{proc}_{|\mathcal{J}|}$).

Variable selection heuristics We use three different variable selection strategies on the set of decision variables that are related to job assignments: *input order* (select variables based on the specified order), *smallest* (select variables that have the smallest values in their domain first, break ties by the specified order), and *first fail* (select variables that have the smallest domains first, break ties by the specified order).

Value selection heuristics We experimented with two different value selection heuristics for the set of variables which is related to job assignments: *min* (the smallest value from a variable domain is assigned first), and *split* (the variable domain is bisected to first exclude the upper half of the domain).

Evaluated search strategies Using the previously defined heuristics we evaluated 8 different programmed search strategies:

1. *default*: Use the solver's default search strategy.
2. *search1*: Assign number of batches first, then continue with the solver's default strategy.
3. *search2*: Assign number of batches first, then continue with *input order* and *min* value selection on the job variables.
4. *search3*: Assign number of batches first, then continue with *smallest* and *min* value selection on the job variables.
5. *search4*: Assign number of batches first, then continue with *first fail* and *min* value selection on the job variables.
6. *search5*: Assign number of batches first, then continue with *input order* and *split* value selection on the job variables.
7. *search6*: Assign number of batches first, then continue with *smallest* and *split* value selection on the job variables.
8. *search7*: Assign number of batches first, then continue with *first fail* and *split* value selection on the job variables.

5.6 Warm-start approach

For those solvers that support a warm-start option, we implemented such an approach. The construction heuristic described in Section 2.4 was used to find an initial solution for a given OSP instance. The solution is then provided to the model by fixing the values of certain decision variables; based on these values the solver then needs to complete this partial solution. We decided to provide values of the following variables:

- For the models using batch positions: the machines and batches jobs are assigned to, the start times of batches. More precisely:
  - For the direct model implemented in MiniZinc as described in Section 3: the values of the variables $X_{m,b,j}$ and $S_{m,b}$ for all $m \in \mathcal{M}$, $b \in [1, n]$, $j \in \mathcal{J}$.
    Note that MiniZinc only supports providing warm-start variables as one-dimensional arrays and therefore a slight modification of the model was necessary. We defined two sets of additional variables encoding the machine a job is assigned to and the batch position a job is assigned to. These variables were then used to set the values of the binary decision variables $X_{m,b,j}$.
  - For the interval variable model described in Section 5.1: The values of the binary variables $X_{m,b,j}$ and the start times of the interval variables $B_{m,b}$ for all $m \in \mathcal{M}$, $b \in [1, n]$, $j \in \mathcal{J}$.
- For the models using representative jobs: the pointers to representative jobs in the same batch and the start times or processing times of representative jobs. More precisely:
  - For the interval variable model described in Section 4: the pointers to representative jobs inBatchWith_j, the start times of interval variables for jobs job_j and for jobs on machines $\text{jobM}_{j,m}$ for all $j \in \mathcal{J}$, $m \in \mathcal{M}$.
  - For the direct model implemented in MiniZinc as described in Section 5.2: the pointers to representative jobs inBatchWith_j, the processing times of jobs proc_j and of jobs on machines $\text{procM}_{j,m}$ for all $j \in \mathcal{J}$, $m \in \mathcal{M}$. We decided to give the values for processing times rather than start times because MiniZinc does not support providing arrays of optional variables for warm-start.

6 Experimental evaluation

In this section we give an overview on the results of an extensive set of experiments that we conducted to evaluate all proposed models. First, we describe our experimental setup in Section 6.1: This consists of the description of the benchmark set, of three different sets of weights that stem from three practical use cases, of the solution methods chosen for comparison and of the experimental environment. Moreover, we describe the method chosen to compare the overall performance of solutions methods against each other. Then, in Section 6.2 we configure our methods: where the chosen solvers allow to do so, we choose the best search strategy per model and per use case and evaluate the impact of the warm-start approach. Finally, in Section 6.3, we present the results of our extensive analysis: We first give an overall performance evaluation of our methods, compare the sizes of our models and then investigate the performance with respect to several criteria in more detail.

6.1 Experimental setup

6.1.1 Benchmark instances

Using a random instance generator specifically designed for the OSP, we created a large set of benchmark instances to evaluate the performance of our proposed models. Our random instance generator is described in detail in Section 1 of the Appendix. First, we executed the random generator once for every possible configuration of the 15 parameter values specified in Table 13. Thereby we produced 1024 instances for each of the 16 combinations of the parameters n, k and a. Then we randomly selected 5 instances from every set of 1024 instances, creating a set of 80 instances which we used throughout our experiments. This set thus consists of 20 instances each with 10 (instances 1-20), 25 (21-40), 50 (41-60) and 100 jobs (61-80). All benchmark instances turn out to be satisfiable and solvable by the construction heuristic described in Section 2.4. This reflects our real-life industrial application, for which feasible solutions usually can be found heuristically and the main aim is to find cost-minimal schedules. The set of benchmark instances is publicly available [21].

Note that this set of benchmark instances is slightly different to the one presented in the authors' conference paper [22], since we have added initial states of machines. Note also that the objective function as defined in equation (4) no longer includes setup times; we can therefore not expect that the optimal solutions for this benchmark set have the same solution cost as in the conference paper [22].

6.1.2 Three use cases for the choice of weights

Together with our industrial partner, we formulated weight settings for three different use cases that we used throughout our experiments.

- **UC1:** Reducing job tardiness has the highest priority for this use case. Tardy jobs are only accepted if absolutely necessary and if scheduling all jobs on time would lead to much higher runtime of ovens. Alternatively, finishing jobs before their latest end date could have been modelled as hard constraints, but this would potentially lead to infeasible instances which was not desired by our industrial partner. Reducing the runtime of ovens has a higher priority than reducing setup costs. The weights are set as follows:
  - $w_p = 4$
  - $w_{sc} = 1$
  - $w_t = 100$

  This choice of weights implies that reducing the number of tardy jobs by one reduces the solution cost by the same amount as reducing the cumulative batch processing time by $25 \cdot avg_t$, i.e., 25 times the average processing time of a job. In other words, grouping 26 jobs with average processing time in a single batch instead of processing them individually has the same impact as scheduling one more job so that it finishes on time. Note that for our benchmark instances that have 100 jobs or less, it will hardly ever be possible to schedule 26 or more jobs in one batch. Thus, setting the tardiness-weight w_t to $25 \cdot w_p$ basically corresponds to modelling lexicographic minimization with tardiness being lexicographically more important than cumulative batch processing time.
- **UC2:** This use case models lexicographic minimization with minimizing cumulative batch processing time being lexicographically more important than the number of tardy jobs and minimizing tardiness being lexicographically more important than setup costs.

Using the notation of Section 3.2, we thus have: $p = \text{obj}_1$, $t = \text{obj}_2$ and $sc = \text{obj}_3$. In this case, weights need to be set individually for each instance. This is done as follows (as described in Section 3.2):

- $w_p = w_t \cdot (n + 1)$
- $w_{sc} = 1$
- $w_t = n \cdot max_{SC} + 1$

- **UC3:** In this use case, the same importance given to reducing tardiness and oven runtime. Reducing setup costs is less important by a factor of 2.

  - $w_p = 2$
  - $w_{sc} = 1$
  - $w_t = 2$

  This choice of weights implies that processing two jobs with average processing time in a single batch instead of processing them individually has the same impact on the solution cost as scheduling one more job so that it finishes on time. Reducing the number of tardy jobs by one (or the cumulative batch processing time by avg_t), has the same impact as reducing the setup costs by $2 \cdot \max_{SC}$.

6.1.3 Evaluated methods and computing environment

An overview of the models and solvers that we evaluated on our benchmark set can be found in Table 1: We implemented the models presented in Sections 3 and 5.2 using the high-level constraint modeling language MiniZinc [26]. We used the MiniZinc IDE version 2.5.3 and recent versions of Chuffed (version 0.10.4), OR-Tools (version 8.0), CP Optimizer (version 20.1.0.0) and Gurobi (version 9.1.1) to evaluate the models. Note that MiniZinc automatically provides an ILP encoding for any constraints that are not already stated in a linear form when an ILP solver such as Gurobi is used to solve the high-level constraint model [34]. For Chuffed and OR-Tools, we used all 7 search strategies described in Section 5.5 and compared them with the solver's default search strategy. For Chuffed, we activated the free search parameter which allows the solver to interleave between the given search strategy and its default search. Furthermore, we investigated a warm-start approach with Gurobi (for the other solvers, warm-start is currently not supported by MiniZinc), as described in Section 5.6. Finally, the OPL models presented in Sections 4 and 5.1 were run using CP Optimizer in IBM ILOG CPLEX Optimization Studio version 20.1.0.0. The warm-start approach was evaluated for these two models as well.

This results in a total of 42 different combinations per instance, when considering all possible combinations of models, solvers, search strategies and of the warm-start option. The time limit for every one of these combinations was set to one hour per instance; this includes the flattening time in MiniZinc and the model extraction time for CP Optimizer. In order to refer to one of these methods in our evaluation, we use the following abbreviations throughout the remainder of this section: "solver-model", where the model names introduced in Table 1 are used. The solvers are referred to as "chuffed", "gurobi", "cpopt", "ortools" and "oplrun".[3]

Experiments were run on single cores, using a computing cluster with 10 identical nodes, each having 24 cores, an Intel(R) Xeon(R) CPU E5–2650 v4 @ 2.20GHz and 252 GB RAM.

[3] "oplrun" is used as an abbreviation for CP Optimizer run directly using the `oplrun` executable in order to distinguish it from CP Optimizer run via MiniZinc.

Table 1 Overview of the experimentally evaluated models and of the solvers used

	solver independent MiniZinc model = direct model	OPL model for CP Optimizer = interval variable model
model with batch positions	"direct-batch-pos" (Section 3)	"interval-batch-pos" (Section 5.1)
model with representative jobs	"direct-repr-jobs" (Section 5.2)	"interval-repr-jobs" (Section 4)
solvers used		
warm-start supported	Gurobi	CP Optimizer (oplrun)
not supported	Chuffed, OR-Tools, CP Optimizer	

6.1.4 Borda scores for the assessment of solution methods

We want to compare the performance of our solution methods across the whole benchmark set and assign a score to every solution method. Moreover, when assessing the performance of a solution method on a particular instance, we would like to take into account the solution status, i.e., whether a solution has been found and whether the found solution was proven to be optimal within the runtime limit, as well as the solution quality for non-optimal solutions. The assessment method used by the MiniZinc challenges [35] fulfills both these goals in a particularly concise way and we therefore decided to implement this method for our experimental evaluation.

The following explanation of the assessment method follows the description of the rules of the MiniZinc challenge 2022 as presented on the challenge's website.[4] The scoring procedure is based on the *Borda count* [36] voting system. Each benchmark instance is treated like a voter who gives each solution method a score that reflects how many solution methods it beats on this instance. The overall *Borda score* of a solution method is then given by summing up the scores across all instances.

More precisely, a method m is better than method m' on instance i if and only if:

- m solved instance i (found a feasible solution for instance i) within the runtime limit and m' did not solve the instance, or
- both m and m' solved the instance i within the runtime limit, but m provided an optimality proof and m' did not, or
- both m and m' did not provide an optimality proof for instance i within the runtime limit, but m found a solution with higher quality than m', i.e., the solution cost of the solution found by m is smaller than the solution cost of the one one found by m'.

A method m then scores points on instance i by comparing its performance with each other method m' on instance i. Points are scored as follows:

- If m gives a better answer than m' (in the above sense), it scores 1 point.
- If m gives a worse answer than m', it scores 0 points.
- If m and m' give indistinguishable answers, the scoring is based on execution time comparison:

[4] https://www.minizinc.org/challenge2022/rules2022.html

Table 2 Best variants of solution methods per use case

solution method	UC1	UC2	UC3
chuffed-direct-batch-pos	search4	search4	search5
chuffed-direct-repr-jobs	search6	search3	search6
ortools-direct-batch-pos	search7	search3	search3
ortools-direct-repr-jobs	search6	search6	search6
gurobi-direct-batch-pos	warm-start	warm-start	warm-start
gurobi-direct-repr-jobs	warm-start	warm-start	warm-start
oplrun-interval-batch-pos	warm-start	warm-start	warm-start
oplrun-interval-repr-jobs	warm-start	–	warm-start

- Let $t(i, m)$ ($t(i, m')$) denote the total time required by method m (method m') on instance i. Then method m scores

$$\frac{t(i, m')}{t(i, m') + t(i, m)}$$

points.
- For the special case that both m and m' do not solve instance i, both score 0 points.

6.2 Configuration of solution methods for each use case

Based on the results of our experiments, we select the best search strategy for every eligible solution method and evaluate whether using the warm-start option is beneficial. This selection per use case is summarised in Table 2. The selected variants are then used in the overall comparison of solution methods in Section 6.3.

6.2.1 Search strategies

To evaluate the performance of the 7 search strategies described in Section 5.5, we ran experiments on all benchmark instances using Chuffed and OR-Tools together with the batch position model as well as the representative jobs model.

We calculated Borda scores independently for both MiniZinc models, the two solvers Chuffed and OR-Tools and the three use cases. The Borda scores and ranks for all three use cases with Chuffed and the batch position based model are displayed in Table 3.

We can see in the results that the solver's default search scored the lowest rank for all three use cases. Further, we clearly see an improvement with the custom search strategies in the Borda score compared to default search. Search4, which uses a first fail based variable selection strategy together with a minimum value selection heuristic, performed best on UC1 and UC2, however, for UC3 Search5, which relies on an input order based variable selection, performed better.

Table 4 shows the Borda scores and results for Chuffed with the representative jobs model.

Again the solver's default search strategy is not competitive with any custom search, although the difference in Borda scores is smaller than it was with the batch position model. Search6, which relies on a smallest domain variable selection strategy and a split value

selection heuristic, performs best on UC1 and UC3, while Search3, which uses a minimum domain value selection instead, produces the best results on UC2.

Results achieved by the different search strategies with OR-Tools and the batch position model are shown in Table 5.

Similar as with the results produced by Chuffed there is a clear improvement over the solver's default search when using the custom search strategies. In this case Search3 per-

Table 3 Comparison of Borda Scores achieved by different search strategies with Chuffed and the model using batch positions

search	UC1 Score	UC1 Rank	UC2 Score	UC2 Rank	UC3 Score	UC3 Rank
default	41.13	8	50.30	8	48.04	8
search1	214.20	7	223.59	7	215.69	7
search2	247.18	6	267.71	5	258.69	5
search3	261.28	5	271.25	4	246.01	6
search4	312.22	1	283.66	1	264.31	4
search5	282.93	3	281.70	2	311.05	1
search6	262.69	4	272.66	3	281.74	3
search7	300.36	2	262.13	6	296.48	2

Table 4 Comparison of Borda Scores achieved by different search strategies with Chuffed and the model using representative jobs

search	UC1 Score	UC1 Rank	UC2 Score	UC2 Rank	UC3 Score	UC3 Rank
default	17.98	8	45.44	8	15.24	8
search1	64.82	7	48.49	7	60.25	7
search2	106.88	3	115.23	4	116.34	5
search3	123.20	2	159.52	1	143.87	2
search4	101.66	6	112.20	6	111.68	6
search5	105.20	4	124.54	3	121.79	3
search6	124.46	1	159.26	2	152.43	1
search7	104.79	5	112.31	5	120.39	4

Table 5 Comparison of Borda Scores achieved by different search strategies with OR-Tools and the model using batch positions

search	UC1 Score	UC1 Rank	UC2 Score	UC2 Rank	UC3 Score	UC3 Rank
default	172.84	8	197.04	8	201.17	8
search1	210.03	7	252.80	7	243.87	7
search2	293.23	3	279.64	5	290.62	3
search3	296.54	2	288.95	1	309.78	1
search4	293.13	4	282.28	4	294.03	2
search5	292.92	5	283.45	3	276.61	4
search6	284.31	6	279.51	6	261.31	6
search7	305.00	1	284.33	2	272.61	5

Table 6 Comparison of Borda Scores achieved by different search strategies with OR-Tools and the model using representative jobs

search	UC1 Score	UC1 Rank	UC2 Score	UC2 Rank	UC3 Score	UC3 Rank
default	4.96	8	2.77	8	6.47	8
search1	145.57	4	139.17	4	146.93	4
search2	90.27	7	84.13	7	81.89	7
search3	308.29	2	340.07	2	315.06	2
search4	133.66	5	128.05	5	136.81	5
search5	120.33	6	108.78	6	89.39	6
search6	336.29	1	342.27	1	345.63	1
search7	169.63	3	162.76	3	172.82	3

formed best on UC2 and UC3, whereas Search7, which uses a first fail variable selection together with a split value selection strategy, scores rank 1 on UC1.

Finally, Table 6 displays the scores for OR-Tools and the representative jobs model.

Again the default search cannot compete with the other search strategies. Further, Search6 is ranked first on all three use cases, which is similar to the results with Chuffed and the representative job model where Search6 also performed high scores for all use cases.

To summarize, the evaluated custom search strategies could clearly improve the results of the default search strategies in all considered cases. Further, which search strategy performed best differs depending on the evaluated model and solver. However, the results indicate that Search6 delivers high scores with the representative job model with both Chuffed and OR-Tools.

Search strategies can also be used for Gurobi and are implemented with branching priorities by the MiniZinc interface. However, initial experiments with Gurobi and all search strategies did not reveal significant differences between the search strategies for this solver. For CP Optimizer run via MiniZinc, search strategies are currently not supported.

Based on the results presented in this section, we selected the search strategy that led to best results per solver, model, and use case for our final experiments which are presented in later sections (see Table 2 for details).

6.2.2 Warm-start option

As described in Section 5.6, we implemented a warm-start approach. This approach was tested with Gurobi run via MiniZinc and the models direct-batch-pos and direct-repr-jobs as well as with CP Optimizer run directly and the models interval-batch-pos and interval-repr-jobs. Note that warm-starts are referred to as *starting points* in CP Optimizer. Currently, MiniZinc does not support a warm-start option for the other evaluated solvers. An overview of the results, when comparing the chosen method with and without the warm-start option, can be found in Table 7.

The construction heuristic (see Section 2.4) could find feasible solutions for all 80 instances within a few seconds. On average, the construction heuristic required 0.271 seconds per instance, 57 out of 80 instances could be solved in less than 0.1 seconds and the maximum time required per instance was 5.6 seconds. As can be seen in the left part of Table 7, warm-starting the methods gurobi-direct-batch-pos, oplrun-interval-batch-pos and oplrun-interval-repr-jobs with these initial solutions led to solutions for all 80 instances, i.e, 20

Table 7 Impact of the warm-start option on the number of solved instances and on the solution quality

model	warm-start	# solved instances			# better solutions		
		UC1	UC2	UC3	UC1	UC2	UC3
gurobi-direct-batch-pos	X	64	61	63	7	10	7
	✓	**80**	**80**	**80**	**24**	**31**	**26**
gurobi-direct-repr-jobs	X	33	31	36	3	0	1
	✓	**39**	**36**	**38**	**10**	**5**	**8**
oplrun-interval-batch-pos	X	49	50	49	8	10	10
	✓	**80**	**80**	**80**	**49**	**49**	**48**
oplrun-interval-repr-jobs	X	**80**	**80**	**80**	13	**12**	16
	✓	**80**	**80**	**80**	**19**	10	**20**

The best results per method and use case are highlighted in bold font

to 30 more instances than without the warm-start option, for gurobi-direct-batch-pos and oplrun-interval-batch-pos. For gurobi-direct-repr-jobs however, only 39 instances could be solved for UC1 (36 for UC2 and 38 for UC3), increasing the number of solved instances only slightly.[5]

The heuristic solution could be improved by the warm-start approach for almost all of the 80 benchmark instances for the methods gurobi-direct-batch-pos and oplrun-interval-repr-jobs. Interestingly, for one of the benchmark instances with 10 jobs, the construction heuristic already finds the optimal solution. However, for both gurobi-direct-repr-jobs and oplrun-interval-batch-pos, this is not the case: the initial solution can only be improved for slightly less than half of the benchmark instances (between 34 and 37 instances for all three use cases).

Regarding the solution quality, using the warm-starting approach leads to an improvement for some instances, but not for all. Detailed numbers can be found in the right part of Table 7. These should be read as follows, as exemplified for the method gurobi-direct-batch-pos and UC1: For 7 of the 80 benchmark instances, the solution found without the warm-start approach was strictly better than with warm-start; for 24 instances the solution was strictly better when using the warm-start approach. Note that we consider found solutions to be better if the other method could not find a solution. As one can see by the values in bold font (indicating whether it was advantageous to use the warm-start option), using warm-start on average helped to improve the solution quality within the runtime limit of one hour for almost all methods and use cases. The number of better solutions with the warm-start approach are largest for gurobi-direct-batch-pos and oplrun-interval-batch-pos. For oplrun-interval-repr-jobs however, the picture is less clear. For UC1 and UC3, using the warm-start approach led to lower solutions costs for more instances than it did not, but the number of better solutions are quite close with and without the warm-start approach. For UC2, the numbers are also close but on average it was better not to use the warm-start approach. A possible explanation is that oplrun-interval-repr-jobs is already good a solving the benchmark instances even without the additional warm-start data. The time spent on loading the warm-start data and completing it to a full solution is thus better spent on improving the solution that can be found without warm-start.

[5] The warmstart data provided to Gurobi only contains values for a subset of the decision variables. The solver thus needs to complete the partial solution and, for gurobi-cp-repr-ws, fails to do so for 41 instances.

As for the number of optimality proofs, these could be increased in more than half of the cases, mostly when using Gurobi as a solver. When comparing proof times, it did - on average - however not make much difference whether warm-start was used or not.

To conclude, the major advantage of using the warm-start approach is that it can help find solutions for more instances. Moreover, for the modelling approach using batch positions and both Gurobi and CP Optimizer, the solutions found were in the great majority of the cases better than (or equally good as) those found without warm-start. Also for gurobi-direct-repr-jobs, the quality of solutions was improved more often than deteriorated by using the warm-start approach. For oplrun-interval-repr-jobs, that could already find solutions for all 80 benchmark instances without warm-start the advantage is not so clear and the use of a warm-start approach cannot be recommended in general.

6.3 Experimental results

In this section, we summarize the results of all performed experiments. For every solution method and every use case, we picked the best configuration regarding search strategies and warm-starting (see the results displayed in Table 2).

6.3.1 Evaluation of overall performance of our solution methods

In order to compare the overall performance of our solution methods, we computed the Borda scores (as defined in Section 6.1.4) independently for all three use cases for all configured evaluated methods, including the greedy construction heuristic. The exact techniques are configured using the best search strategy per use case and, if advantageous, the warm-start option as reported in Table 2. This also allows us to evaluate the robustness of our solution methods regarding different sets of weights of the objective components. The results are displayed in Fig. 2, where one line-graph is drawn per use case. The solution methods are sorted from best to worse when considering the sum of Borda scores obtained for all three use cases. As one can see, the Borda scores for the two other use cases UC2 and UC3 follow the same general tendency and are close in value to those for use case UC1. This means that the performance of the evaluated solution methods is almost the same across all three evaluated use cases and that the performance of our solution methods are robust with respect to changes in the chosen weights.

In particular, for all three use cases the first four places are taken by the same solvers: The overall best solution method is the interval model with representative jobs run directly with CP Optimizer (oplrun-interval-repr-jobs) and the second best is the direct model with batch positions run with Gurobi (gurobi-direct-batch-pos). The third place is taken by cpopt-direct-batch-pos and the fourth by oplrun-interval-batch-pos. The values of the Borda scores of the three best solution methods for all three use cases UC1, UC2 and UC3 are reported in Table 8. Note that for UC1 and UC3, the scores obtained by oplrun-interval-repr-jobs and gurobi-direct-batch-pos are very close in value; for UC2 however, the difference is larger. One possible explanation is that UC2 models lexicographic minimization and involves large multiplicative constants for the three components of the objective function; it could be that CP Optimizer is better at dealing with these large numerical values in the objective function than Gurobi. Moreover, note that there is a significant gap between the second and third place; oplrun-interval-repr-jobs and gurobi-direct-batch-pos are by far the best two solutions methods for all three use cases.

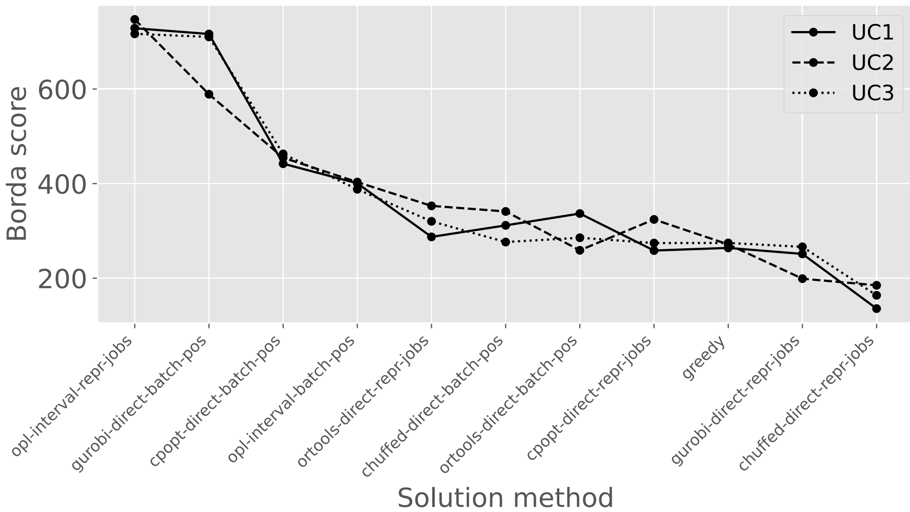

Fig. 2 Borda scores of compared solution methods for the three use cases UC1, UC2 and UC3

Table 8 Borda scores of the three best solution methods for the three use cases UC1, UC2 and UC3

	1st place oplrun-interval-repr-jobs	2nd place gurobi-direct-batch-pos	3rd place cpopt-direct-batch-pos
UC1	728.53	716.39	442.14
UC2	747.16	589.02	454.91
UC3	716.77	710.25	462.80

Moreover, note that the maximum possible score for any solution method is 800, since every solution method is compared to 10 other solution methods on a total of 80 instances. The scores of the overall best solution method opl-interval-repr-jobs thus lie between 89% (for UC3) and 93% (for UC2) of the possible maximum. Also for the second-best solution method gurobi-direct-batch-pos, the scores lie between 73% (UC2) and 89% (UC1). A method that always achieves better results than all other methods would achieve a score of 100%. The very high scores of the two best solution methods lead us to the conclusion that when solving a benchmark instance for the OSP with one of these methods, the likelihood is very large that the result is better than when using any of the other methods.

It is also interesting to observe that when using a Gurobi, Chuffed or CP Optimizer run via MiniZinc, the modelling approach using batch positions is better than the one with representative jobs, whereas the contrary is the case for the interval models run directly with CP Optimizer. A possible explanation can be found by investigating the model sizes of these two modelling approaches, as we will see in the following section.

Also note that the greedy construction heuristic ranks in 9th place out of 12, when summing up the scores across all three use cases and was capable of achieving scores that are comparable with those achieved by the exact techniques ortools-direct-repr-jobs, ortools-direct-batch-pos,

Table 9 Overview on the number of variables and constraints for selected instances and the two direct modelling approaches solved with Gurobi

inst	n	k	a	gurobi-direct-batch-pos		gurobi-direct-repr-jobs	
				var	cons	var	cons
1	10	2	2	810	2,207	9,100	21,688
6	10	2	5	1,342	2,354	9,677	22,455
11	10	5	2	2,039	5,688	21,642	51,660
16	10	5	5	3,319	5,978	21,765	51,620
21	25	2	2	3,590	11,693	86,893	236,384
26	25	2	5	4,842	11,010	85,726	234,026
31	25	5	2	9,019	29,839	208,741	573,510
36	25	5	5	12,269	27,556	207,027	570,146
41	50	2	2	11,988	43,321	580,177	1,669,031
46	50	2	5	14,942	46,034	581,916	1,669,249
51	50	5	2	30,566	111,036	1,426,954	4,117,751
56	50	5	5	36,699	104,336	1,425,095	4,113,469
61	100	2	2	44,590	156,446	-	-
66	100	2	5	49,142	172,382	-	-
71	100	5	2	110,664	428,584	-	-
76	100	5	5	124,549	444,589	-	-

chuffed-direct-batch-pos, cpopt-direct-repr-jobs and gurobi-direct-repr-jobs (for some of the use cases). This relatively good result is mainly due to the fact that the construction heuristic was capable of solving all 80 benchmark instances, which is not the case for all exact methods (details for UC1 can be found later on in Table 11).

To sum up: Across all three use cases, the overall best solution method is the interval model with representative jobs run with CP Optimizer directly and the second best is the direct model with batch positions run with Gurobi. Also for the other solution methods, the choice of weights does not have much influence on the overall performance of our methods and we can conclude that our methods are robust with respect to different weight settings.

6.3.2 Comparison of model sizes

Tables 9 and 10 summarize the numbers of variables and constraints for selected instances for use case UC1 (we selected one instance for each of the 16 different size categories). These numbers are retrieved from the solvers' log files.

The tables display in each row from left to right: The id of the instance, the parameters n (number of jobs), k (number of machines), a (number of attributes) that were used for generating the instance, the number of constraints (cons) and variables (vars) for all four modelling approaches. Table 9 contains information for the two direct models implemented in MiniZinc and solved with the overall best solver Gurobi; Table 10 contains information on the same instances for the two interval variable models solved directly with CP Optimizer.

The numbers displayed in Table 9 show that the direct-batch-pos model uses much fewer variables and constraints compared to direct-repr-jobs for instances of all sizes with Gurobi. Actually, the direct-repr-jobs model becomes so large with $n = 100$ that the model compilation with MiniZinc could not finish within the given runtime, and thus no numbers on

Table 10 Overview on the number of variables and constraints for selected instances and the two interval variable modelling approaches solved with CP Optimizer

inst	n	k	a	oplrun-interval-batch-pos		oplrun-interval-repr-jobs	
				var	cons	var	cons
1	10	2	2	282	1,652	112	998
6	10	2	5	282	1,652	112	998
11	10	5	2	705	4,082	235	1,691
16	10	5	5	705	4,082	235	1,691
21	25	2	2	1,452	7,877	277	4,343
26	25	2	5	1,452	7,877	277	4,343
31	25	5	2	3,630	19,622	580	6,071
36	25	5	5	3,630	19,622	580	6,071
41	50	2	2	5,402	28,252	552	14,918
46	50	2	5	5,402	28,252	552	14,918
51	50	5	2	13,505	70,522	1,155	18,371
56	50	5	5	13,505	70,522	1,155	18,371
61	100	2	2	20,802	106,502	1,102	54,818
66	100	2	5	20,802	106,502	1,102	54,818
71	100	5	2	52,005	266,072	2,305	61,721
76	100	5	5	52,005	266,072	2,305	61,721

the variables and constraints can be shown in the table (marked with -). On the other hand, when comparing the interval-batch-pos and interval-repr-jobs models in Table 10, we can see that the model using representative jobs uses a lower number of variables and constraints. This is an expected result as the representative job model can be captured more efficiently when using the scheduling global constraints available with CP Optimizer. We further see that the number of attributes does not affect the number of variables and constraints with CP Optimizer. Again this is expected as attributes can be implicitly handled with the scheduling global constraints.

These numerical findings are in line with the overall performance results presented in the previous section: when using a direct modelling approach in MiniZinc the smaller model with batch positions leads to better results; when using interval variables with CP Optimizer the smaller model with representative jobs beats the batch position model.

6.3.3 Detailed results for several performance measures (for UC1)

The rankings reported in Section 6.3.1 allowed us to compare the overall performance of our solution methods. Now, we want to take a closer look at several performance measures and answer the following questions: Which methods are best at …

- …finding solutions?
- …providing optimality proofs?
- …finding good solutions?
- …providing lower bounds?
- …closing the optimality gap?

Finally, we are also interested in assessing the quality of solutions found by the best solution methods. In the following, we answer these questions for the first use case UC1 since the results for the other two use cases are similar.

Table 11 provides an overview of the final results produced for UC1 on the 80 benchmark instances with all configured evaluated methods, including the greedy construction heuristic. For the exact techniques, we used the beat search strategy as reported in Table 2 and the warm-start option. The first column in each row denotes the evaluated solver and model. The solution methods are sorted by their Borda scores for UC1. From left to right, the columns display: the number of solved instances, the number of instances where overall best, i.e. minimal, solution cost results could be achieved, the number of obtained optimal solutions, the number of optimality proofs, the number of instances where the fastest optimality proof could be found, and the number of instances where the best, i.e. maximal, lower bound could be found.

Finding solutions The OPL model with representative jobs (oplrun-interval-repr-jobs) and with batch positions (oplrun-interval-batch-pos) as well as the direct model with batch positions run with Gurobi (gurobi-direct-batch-pos) finds solutions for all 80 instances. This is not surprising since these three solution methods all use a warm-start approach (for a comparison of the results with and without the warm-start approach, see Section 6.2.2) and the construction heuristic is capable of finding feasible solutions for all 80 instances. Even without a warm-start approach, ortools-direct-batch-pos was capable of finding solutions for 78 instances, followed by cpopt-direct-batch-pos (71 instances) and chuffed-direct-batch-pos (70 instances). For all solvers run via MiniZinc, the model with batch positions led to much better results than the one with representative batches; this is in line with the overall performance results.

To conclude, we can note that besides the best-performing solution methods OR-Tools was also good at finding solutions with the model using batch positions.

Finding optimal solutions and delivering optimality proofs Using all evaluated methods, optimal solutions could be found for 38 instances: all instances with 10 jobs, 16 instances with 25 jobs, one with 50 jobs and one with 100 jobs.

Most optimality proofs were provided by Gurobi and the batch position model (34 proofs), followed by CP Optimizer run directly with the representative jobs model (28 proofs) and Gurobi with the representative jobs model (26 proofs). Moreover, some of the solutions found by Gurobi were optimal, but not proven to be optimal (3 solutions each for gurobi-direct-batch-pos and gurobi-direct-repr-jobs). With oplrun-interval-repr-job, 7 additional optimal solutions could be found but were not proven to be optimal.

Far less optimality proofs were delivered by the other solution methods, but optimal solutions could still be found. Even though cpopt-direct-batch-pos (cpopt-direct-repr-jobs) could only provide 13 (18) optimality proofs, it did find 31 (29) optimal solutions; similarly chuffed-direct-batch-pos provided only 14 optimality proofs but could find 24 optimal solutions. For ortools-direct-batch-pos, optimality proofs could be delivered for all 20 instances where the optimal solution was found. It is noteworthy that for Chuffed and CP Optimizer run via MiniZinc significantly more optimality proofs could be found when using the model with representative jobs.

Overall, the greatest number of fastest proofs were delivered by oplrun-interval-repr-jobs (15 fastest proofs in total). For the subset of instances with 10 jobs (20 instances in total) for which many solution methods could deliver optimality proofs, Table 12 contains further

Table 11 Overview of the detailed computational results for UC1 based on 80 benchmark instances

method	#solved	#best	#optimal	#proven optimal	#fastest proof	#best lower bound
oplrun-interval-repr-jobs	**80**	**65**	35	28	**15**	28
gurobi-direct-batch-pos	**80**	54	**37**	**34**	9	44
cpopt-direct-batch-pos	71	38	31	13	0	22
oplrun-interval-batch-pos	**80**	23	22	17	0	17
ortools-direct-batch-pos	78	20	20	20	0	20
chuffed-direct-batch-pos	70	24	24	14	0	0
ortools-direct-repr-jobs	56	21	21	18	3	18
cpopt-direct-repr-jobs	48	30	29	18	0	**50**
greedy	**80**	1	1	0	0	0
gurobi-direct-repr-jobs	39	29	29	26	0	28
chuffed-direct-repr-jobs	27	23	23	21	0	0

Best results per column are printed in bold font

Table 12 Average runtimes until optimality is proven and number of visited nodes for the 20 small instances with ten jobs

method	avg rt	avg nodes	std rt	std nodes
oplrun-interval-repr-jobs	4.06	1.04E+05	7.81	2.27E+05
gurobi-direct-batch-pos	**2.23**	**2.15E+02**	**1.92**	**3.29E+02**
ortools-direct-batch-pos	153.07	n/a	353.96	n/a
gurobi-direct-repr-jobs	34.58	5.62E+02	43.92	7.47E+02
chuffed-direct-repr-jobs	105.46	2.59E+06	126.20	4.08E+06

Numbers are reported only for those methods that could prove optimality for all small instances
The best results are highlighted in bold font

information regarding optimality proofs: the average total runtime (which includes the time for finding the optimal solution and for proving optimality), the average number of nodes visited in the search process, the standard deviation of the runtime and the standard deviation of the number of visited nodes. We can see that fastest proofs were delivered by gurobi-direct-batch-pos: on average, 2 seconds were required per instance and all proofs could be delivered within less than 10 seconds. Second best results were achieved by oplrun-interval-repr-jobs; the slightly higher average and standard deviation are mainly due to a single instance for which more than 10 seconds were required for the optimality proof. The other three solution methods that could provide optimality proofs for all 20 small instances require more time with averages above 30 seconds. However, for all methods compared in Table 12, the proof times are on average less than three minutes and exceed ten minutes only for a single instance: the maximum proof time for a small instance is 183 seconds for gurobi-direct-repr-jobs, 515 seconds for chuffed-direct-repr-jobs and 1473 seconds for ortools-direct-batch-pos.

For the solution methods run via MiniZinc, the average number and the standard deviation of nodes visited during the search process are in line with the distribution of proof times: the fewer nodes were visited, the faster proofs were delivered. For oplrun-interval-repr-jobs, the number of nodes visited were higher, even though proofs could be delivered very quickly.

To sum up, the two best performing methods oplrun-interval-repr-jobs and gurobi-direct-batch-pos also found most optimal solutions and provided most optimality proofs. However, in terms of delivering optimality proofs, gurobi-direct-batch-pos clearly outperforms the overall best method oplrun-interval-repr-jobs. For small instances with ten jobs, the two best performing methods were capable of providing proofs much faster than other methods.

Best solutions Regarding the quality of found solutions, the results are totally in line with the overall ranking of solution methods: best results were achieved by oplrun-interval-repr-jobs (65 best results), followed by gurobi-direct-batch-pos (54 best results) and cpopt-direct-batch-pos (38 best results). For all other solution methods, best results could only be achieved for those instances that were solved optimally or for one additional instance.

Best lower bounds Lower bounds (also referred to as dual bounds) on the solution cost of an optimal solution are provided by Gurobi, OR-Tools and CP Optimizer (both when called directly and via MiniZinc). Interestingly, best lower bounds were found by CP Optimizer run via MiniZinc and the model using representative jobs, a solution method that did not perform particularly well with respect to other criteria (best bounds for 50 instances). Second-best results were provided by gurobi-direct-batch-pos, the solution method that could also find most optimality proofs (best bounds for 44 instances).

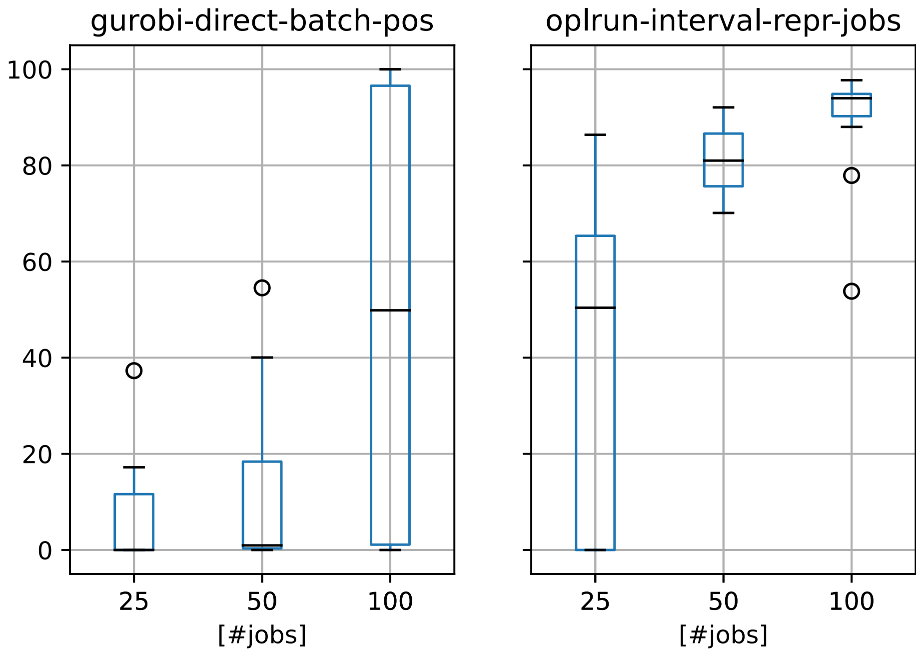

Fig. 3 Optimality gap in % for the solutions found by the two best solution methods. Results are grouped by the size of the instances

Optimality gap for solutions found by best methods Finally, we want to evaluate the ability of the two best performing methods oplrun-interval-repr-jobs and gurobi-direct-batch-pos in closing the optimality gap. That is, how good are these two methods and finding primal and dual bounds that are close to each other? For small instances with 10 jobs, optimal solutions and optimality proofs could be found very quickly. For larger instances, for which no optimality proof could be delivered within the runtime limit, we compute the optimality gap which is defined as follows: For a given solver, if $s(i)$ is the objective value of the solution found and $b(i)$ is the best (i.e. maximal) lower bound found for instance i, the optimality gap is given by $g(i) = (s(i) - b(i))/s(i)$. The boxplots in Fig. 3 visualize the distribution of the optimality gap (in %) per instance for the two best solution methods oplrun-interval-repr-jobs and gurobi-direct-batch-pos. The results are grouped by the number of jobs in the instance.

For both compared solution methods, the optimality gap increases with the number of jobs per instance. However, for all three job sizes the median optimality gaps are significantly smaller for gurobi-direct-batch-pos than for oplrun-interval-repr-jobs. For instances with 25 jobs and gurobi-direct-batch-pos, the optimality gap is smaller than 1% for 14 instances compared to 8 instances for oplrun-interval-repr-jobs. Moreover, the gap is never above 40% for gurobi-direct-batch-pos, whereas it is above 40% for 11 instances for oplrun-interval-repr-jobs. For instance with 50 jobs and gurobi-direct-batch-pos, the gap is smaller than 1% for 10 instances and never above 60%; for oplrun-interval-repr-jobs it is always above 60%. For the largest jobs with 100 jobs, the gap reaches values close to 100% for both solvers. But again, the median value is much lower for gurobi-direct-batch-pos (50%) than for oplrun-interval-repr-jobs (94%).

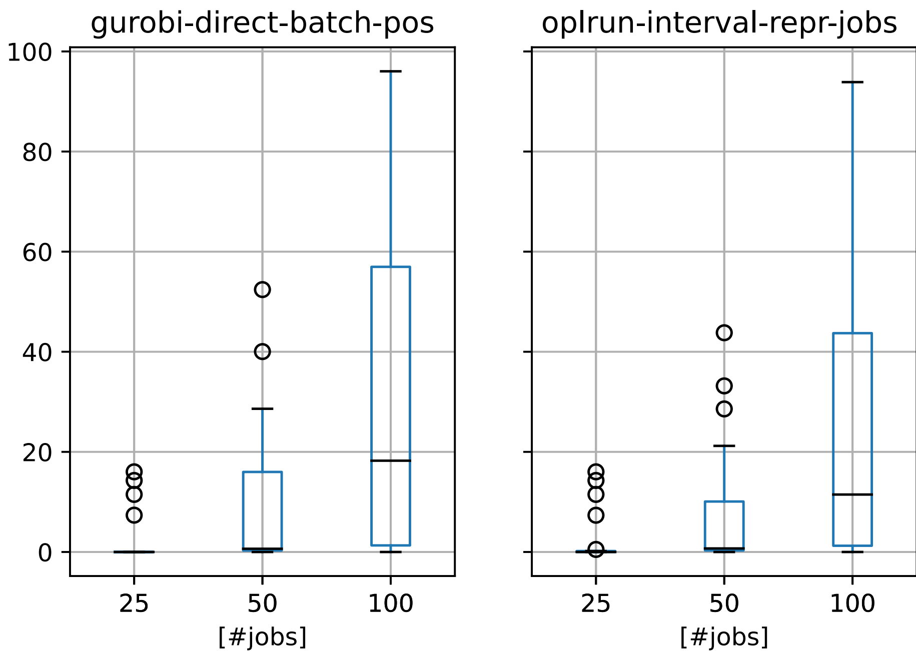

Fig. 4 Solution quality for the solutions found by the two best solution methods measured in terms of the relative distance to the overall best lower bound found. Results are grouped by the size of the instances

We can conclude that for those instances where no optimality proof could be delivered within the runtime limit, gurobi-direct-batch-pos is much stronger than oplrun-interval-repr-jobs in closing the gap between upper and lower bounds.

Quality of solutions found by best methods We previously saw that gurobi-direct-batch-pos is a lot better than oplrun-interval-repr-jobs at closing the optimality gap for instances with 25 jobs or more. This does however not imply that the solutions found by oplrun-interval-repr-jobs are necessarily worse than those found by gurobi-direct-batch-pos. Indeed, as indicated in Table 11, more best solutions were found by oplrun-interval-repr-jobs than by gurobi-direct-batch-pos. In order to evaluate the quality of solutions found by the best two solution methods, we therefore compute the relative distance to the overall best lower bound found as follows. For a given instance i, let $ob(i)$ be the overall best lower bound found by any solver and let $s(i)$ be the objective value of the solution found by some solver. Then the relative distance for this solver is given by $r(i) = (s(i) - ob(i))/s(i)$. The boxplots in Fig. 4 visualize the distribution of this relative distance (in %) per instance for the two best solution methods oplrun-interval-repr-jobs and gurobi-direct-batch-pos. The results are grouped by the number of jobs in the instance.

The first observation that we can make is that—in contrast to the results concerning the optimality gap displayed in Fig. 3—the results look very much alike for both best-performing methods. For both methods, the relative distance increases with the number of jobs per instance: while the solutions are optimal or very close to the optimum for most instances with 25 jobs (oplrun-interval-repr-jobs found optimal solutions for 14 out of 20 instances, gurobi-direct-batch-pos did so for 16 instances) and never above 20 %. This is no longer the case for instances with 50 or 100 jobs. Even though the relative distance generally increases

with the number of jobs, there are also quite a few larger instances that are close to the optimum: For oplrun-interval-repr-jobs there are 15 instances with 50 jobs or more for which the optimality gap is less than 1% (14 instances for gurobi-direct-batch-pos). In total, the optimality gap is less than 1% for 51 instances for oplrun-interval-repr-jobs (50 instances for gurobi-direct-batch-pos) and smaller than 10% for 61 instances out of 80 (for both methods).

Looking closer at the data, we can see that oplrun-interval-repr-jobs achieves results with slightly smaller optimality gap for the larger instances with 50 or 100 jobs than gurobi-direct-batch-pos. Nonetheless, the large instances still leave room for improvement for both methods, regarding finding better solutions (or improved lower bounds).

6.4 Conclusions of the experimental evaluation

The overall scoring of solution methods provided in Section 6.3.1 produced two clear winners: the interval model with representative jobs solved by CP Optimizer directly (oplrun-interval-repr-jobs) and the direct model with batch positions solved by Gurobi (gurobi-direct-batch-pos). The rankings of solution methods created for three different practical use cases shows the robustness of our methods with respect to changes in the weights of the objective components. The analysis of our solution methods with respect to several performance criteria in Section 6.3.3 showed that the two winning methods also produced best results in terms of the number of solved instances, of best solutions, of optimality proofs and of proof speed. Only with respect to finding good lower bounds best results were achieved by a different method, namely cpopt-direct-repr-jobs.

The evaluation of different search strategies showed that the selection of search strategy has a major impact on the solvers Chuffed and Gurobi. However, the solution methods using these solvers could still not compete with the best two solution methods. We saw that including an initial solution with a warm-start approach can help the solvers CP Optimizer and Gurobi produce more solutions. The positive impact was however less obvious for the best method oplrun-interval-repr-jobs, which could already produce very good results without a warm-start approach.

Finally, let us attempt to answer the question which solver and which model are best at solving the OSP. This is difficult to answer in general, since the combination of solver and model is crucial for the performance of a solution method.

Using the results presented in Section 6.3.3 we can nonetheless discern CP Optimizer (run directly) as the overall best method when it comes to solving all instances and finding good solutions. The second best solver in this respect is Gurobi; moreover Gurobi is the best solver when it comes to providing optimality proofs.

Regarding the performance of the different models, the experimental results show that the model using representative jobs produced the largest number of best results when interval variables can be utilized (which is the case with the OPL models and CP Optimizer). However, if interval variables are not supported by the solver (which is the case with Gurobi, Chuffed, OR-tools, and CP Optimizer via the MiniZinc interface), in general the number of best found solutions is improved when using the model with batch positions instead of representative jobs. We conclude that using representative jobs for batches can be considered the overall better model if efficient interval variables can be utilized. However, if no solver that supports interval variables is available or proving optimality is the main aim, the model based on batch positions can leads to better results.

7 Summary and future work

In this paper, we introduced and formally defined the Oven Scheduling Problem and provided instances for this problem, a new batch scheduling problem that appears in the electronic component industry. We proposed two different modelling approaches, presented corresponding CP and ILP models in MiniZinc and OPL and investigated various search strategies.

Using our models as well as a warm-start approach, we were able to find feasible solutions for all 80 benchmark instances. Provably optimal solutions could be found for nearly half of the instance set and for 51 of the 80 instances, the optimality gap is less than 1%. Overall, the best results could be achieved with the interval variable model using representative jobs (see Section 4) and running CP Optimizer directly, followed by the direct model with batch positions (see Section 3) solved by Gurobi using warm-start. We evaluated all our solution methods for three different use cases that arise in practice. The results showed that the same methods achieved best results in all three cases, which indicates the robustness of our solution methods with respect to changes in the weights of the objective components.

The proposed exact approach was able to produce solutions for instances up to 100 jobs. However, to further improve the solution quality for large instances and efficiently tackle large-scale realistic instances with more than 100 jobs, we plan to develop metaheuristic search strategies based on local search or large neighborhood search. First results using a local search approach with simulated annealing already look promising and we plan to continue our research in this direction [37]. Within a runtime limit of five minutes, the simulated annealing approach could reach all optimal solutions found by exact methods. The solution quality could be improved for approximately half the instances for which no provably optimal solutions were found by the exact methods.

Moreover, in order to explain which parameters cause instances to be hard, an in-depth instance space analysis could be conducted. This could also shed light onto which methods perform best on different parts of the instance space.

Acknowledgements The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

Funding Open access funding provided by TU Wien (TUW).

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Appendix

Random instance generator

The random instance generator we propose is based on random instance generation procedures for related problems from the literature [16, 38]. However, as the existing variants were designed for batch scheduling problems which neither include machine eligibility con-

straints, machine availability times nor setup costs and times, the random generation of the associated instance parameters is a novel contribution of this paper. The list of parameters for this instance generator is given in Table 13.

Jobs. The list of n jobs is generated as follows. First, for every job j, the minimal processing time $mint_j$ is chosen using a discrete uniform distribution $U(1, max_T)$. For the maximum processing time, there are two options: either there is no upper limit on the processing time of jobs (`max_time` = `false`), in which case the maximum processing time is set to max_T for all jobs. Or, if `max_time` = `true`, the maximum processing time for a job is chosen using a discrete uniform distribution $U(mint_j, max_T)$. Next, the earliest start and latest end times are determined for every job. The earliest start time et_j is chosen similarly as by Velez Gallego [38] according to a discrete uniform distribution $U(0, \lceil \rho \cdot Z \rceil)$ where $\rho \in [0, 1]$ and $Z = \sum mint_j$ is the total processing time of all jobs. If $\rho = 0$, all jobs are available right at the beginning and as ρ grows, the jobs are released over a longer interval. The latest end time lt_j is chosen as in [16] according to $lt_j = et_j + \lfloor U(1, \phi) \cdot mint_j \rfloor$ where $\phi \geq 1$. If $\phi = 1$, the latest end time is equal to the sum of the earliest start time and the minimum processing time, meaning that all jobs must be processed immediately in order to finish on time. As ϕ grows, more time is given for every job to be completed and tardy jobs are less likely. Regarding the set of eligible machines for a job, one machine is chosen at random among all machines. Additional machines are then added to this set with probability σ each. The size s_j and attribute a_j of a job are both chosen at random between 1 and the maximum job size s or number of attributes respectively.

Attributes. The setup times and setup costs matrices can be of four different types: Constant, arbitrary, realistic and symmetric. For the type "constant", setup times/costs are all equal to a randomly chosen constant between 0 and $\lceil max_T/4 \rceil$. For the type "arbitrary", every entry is chosen independently at random between 1 and $\lceil max_T/4 \rceil$. For the type "realistic", setup times/costs between two batches of the same attribute are lower and are chosen independently at random between 0 and $\lceil max_T/8 \rceil$, whereas setup times/costs between different attributes are higher and are chosen between $\lceil max_T/8 \rceil + 1$ and $\lceil max_T/4 \rceil$. For the type "symmetric" a symmetric matrix is generated with random entries between 0 and $\lceil max_T/4 \rceil$.

Machines. The maximum machine capacity c_m is randomly chosen between min_C and max_C where the lower bound min_C is set to the maximum job size s to ensure that every job fits into every machine. The initial state s_m is chosen at random among the set of attributes $\{1, \ldots, a\}$. For the machine availability times, we first fix the length of the scheduling horizon l. If we assume that every job is processed in a batch of its own and that all jobs are processed on the same machine, the total runtime is at most equal to the sum of all processing times Z plus n times the maximal setup time max_{st}. The parameter $\tau \in (0, 1]$ is a lower bound for the fraction of time that every machine is available. Thus, if max_{et} is the latest earliest start time, all jobs should–on average–be finished at time

$$l = max_{et} + \lceil (Z + n \cdot max_{st})/(\tau) \rceil$$

which we use to set the length of the scheduling horizon. Note that if the latest end date of a job is greater than this upper bound we simply use it instead. Now, for every one of the k machines, we pick the number of availability intervals I randomly between 1 and max_I. Every interval $[start_i, end_i]$ should be long enough to accommodate at least a single job with minimal processing time $min_T = \min(mint_j : j \in \mathcal{J})$ (plus the necessary setup times). Thus, the minimum distance between two interval start times $start_i$ and $start_{i+1}$ is $d = min_T + max_{st}$. We first pick the start time $start_1$ of the first interval: In order to guarantee that every machine is available at least a fraction τ of the time, $0 \leq start_1 \leq \lfloor l \cdot (1 - \tau) \rfloor$ must hold and in order to leave enough time for all availability intervals, it has to hold

Table 13 List of parameters of the random instance generator

Name	Description	Values
Parameters relating to jobs		
n	number of jobs	10, 25, 50, 100
max_T	overall maximum processing time	10, 100
`max_time`	`true` if jobs have a max. processing time	`true`, `false`
ρ	determines spread of earliest start times	0.1, 0.5
ϕ	determines time from earliest start to latest end of job	2, 5
σ	determines number of eligible machines per job	0.2, 0.5
s	maximum job size	5, 20
Parameters relating to attributes		
a	number of attributes	2, 5
`s_time` = `s_cost`	type of setup-time matrix type of setup-cost matrix	`constant`, `arbitrary`, `realistic`, `symmetric`
Parameters relating to machines		
k	number of machines	2, 5
$min_C = s$	lower bound for max. machine capacity (=max. job size)	5, 20
max_C	upper bound for maximum machine capacity	20, 100
τ	lower bound for the fraction of time machines are available	0.25, 0.75
max_I	max. number of availability intervals	5

$start_1 \leq l - I \cdot d$. Next, for the start times of the remaining intervals, we pick $I - 1$ random integers between $(start_1 + d)$ and $(l - d)$ that are at least d apart. Finally, we determine the end time end_i of the i-th interval:

$$end_i = start_i + \max(d, \lceil U(\tau, 1) \cdot (start_{i+1} - start_i) \rceil)$$

References

1. Potts, C. N., & Kovalyov, M. Y. (2000). Scheduling with batching: A review. *European Journal of Operational Research, 120*(2), 228–249.
2. Mathirajan, M., & Sivakumar, A. I. (2006). A literature review, classification and simple meta-analysis on scheduling of batch processors in semiconductor. *The International Journal of Advanced Manufacturing Technology, 29*(9–10), 990–1001.
3. Fowler, J. W., & Mönch, L. (2022). A survey of scheduling with parallel batch (p-batch) processing. *European Journal of Operational Research, 298*(1), 1–24.
4. Malapert, A., Guéret, C., & Rousseau, L.-M. (2012). A constraint programming approach for a batch processing problem with non-identical job sizes. *European Journal of Operational Research, 221*(3), 533–545.
5. Lee, C.-Y., Uzsoy, R., & Martin-Vega, L. A. (1992). Efficient algorithms for scheduling semiconductor burn-in operations. *Operations Research, 40*(4), 764–775.
6. Zhao, Z., Liu, S., Zhou, M., Guo, X., & Qi, L. (2020). Decomposition Method for New Single-Machine Scheduling Problems From Steel Production Systems. *IEEE Transactions on Automation Science and Engineering, 17*(3), 1376–1387.
7. Polyakovskiy, S., Thiruvady, D., & M'Hallah, R. (2020). Just-in-time batch scheduling subject to batch size. In *Proceedings of the 2020 genetic and evolutionary computation conference* (GECCO '20, pp. 228–235). New York, NY: Association for Computing Machinery
8. Tang, T. Y. & Beck, J. C. (2020). CP and Hybrid Models for Two-Stage Batching and Scheduling. In *Integration of constraint programming, artificial intelligence, and operations research* (Lecture Notes in Computer Science, pp 431–446)
9. Brucker, P., Gladky, A., Hoogeveen, H., Kovalyov, M. Y., Potts, C. N., Tautenhahn, T., & Van De Velde, S. L. (1998). Scheduling a batching machine. *Journal of Scheduling, 1*(1), 31–54.
10. Kosch, S. & Beck, J. C. (2014). A new mip model for parallel-batch scheduling with non-identical job sizes. In *International Conference on AI and OR techniques in constriant programming for combinatorial optimization problems* (pp. 55–70). Springer
11. Trindade, R. S., de Araújo, O. C., & Fampa, M. (2020). Arc-flow approach for parallel batch processing machine scheduling with non-identical job sizes. In *International Symposium on Combinatorial Optimization* (pp. 179–190). Springer
12. Azizoglu, M., & Webster, S. (2001). Scheduling a batch processing machine with incompatible job families. *Computers & Industrial Engineering, 39*(3–4), 325–335.
13. Parsa, N. R., Karimi, B., & Kashan, A. H. (2010). A branch and price algorithm to minimize makespan on a single batch processing machine with non-identical job sizes. *Computers & Operations Research, 37*(10), 1720–1730.
14. Damodaran, P., Vélez-Gallego, M. C., & Maya, J. (2011). A grasp approach for makespan minimization on parallel batch processing machines. *Journal of Intelligent Manufacturing, 22*(5), 767–777.
15. Cakici, E., Mason, S. J., Fowler, J. W., & Geismar, H. N. (2013). Batch scheduling on parallel machines with dynamic job arrivals and incompatible job families. *International Journal of Production Research, 51*(8), 2462–2477.
16. 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 & Operations Research, 34*(10), 3016–3028.
17. Costa, A., Cappadonna, F. A., & Fichera, S. (2014). A novel genetic algorithm for the hybrid flow shop scheduling with parallel batching and eligibility constraints. *The International Journal of Advanced Manufacturing Technology, 75*(5–8), 833–847.
18. Cheng, B., Wang, Q., Yang, S., & Hu, X. (2013). An improved ant colony optimization for scheduling identical parallel batching machines with arbitrary job sizes. *Applied Soft Computing, 13*(2), 765–772.
19. Zhou, H., Pang, J., Chen, P.-K., & Chou, F.-D. (2018). A modified particle swarm optimization algorithm for a batch-processing machine scheduling problem with arbitrary release times and non-identical job sizes. *Computers & Industrial Engineering, 123*, 67–81.

20. Damodaran, P., & Vélez-Gallego, M. C. (2012). A simulated annealing algorithm to minimize makespan of parallel batch processing machines with unequal job ready times. *Expert Systems with Applications, 39*(1), 1451–1458.
21. Lackner, M.-L., Mrkvicka, C., Musliu, N., Walkiewicz, D., & Winter, F. (2022a). Benchmark instances and models for the Oven Scheduling Problem [Data Set]. Zenodo. https://doi.org/10.5281/zenodo.7456938
22. Lackner, M.-L., Mrkvicka, C., Musliu, N., Walkiewicz, D., & Winter, F. (2021). Minimizing Cumulative Batch Processing Time for an Industrial Oven Scheduling Problem. In *27th International conference on principles and practice of constraint programming (CP 2021)*, volume 210 of *leibniz international proceedings in informatics (LIPIcs)* (pp. 37:1–37:18). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik
23. Graham, R. L., Lawler, E. L., Lenstra, J. K., & Kan, A. R. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics, 5*, 287–326.
24. Uzsoy, R. (1994). Scheduling a single batch processing machine with non-identical job sizes. *The International Journal of Production Research, 32*(7), 1615–1635.
25. Uzsoy, R. (1995). Scheduling batch processing machines with incompatible job families. *International Journal of Production Research, 33*(10), 2685–2708.
26. Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., & Tack, G. (2007). MiniZinc: Towards a Standard CP Modelling Language. In C. Bessière (Ed.), *Principles and practice of constraint programming - CP 2007* (Lecture Notes in Computer Science, pp. 529–543). Berlin, Heidelberg: Springer.
27. Deb, K. (2014). Multi-objective optimization. *Search methodologies* (pp. 403–449). Boston, MA: Springer.
28. Miettinen, K. (2012). *Nonlinear multiobjective optimization* (vol. 12). New York: Springer Science & Business Media.
29. Chiandussi, G., Codegone, M., Ferrero, S., & Varesio, F. (2012). Comparison of multi-objective optimization methodologies for engineering applications. *Computers & Mathematics with Applications, 63*(5), 912–942.
30. Laborie, P., Rogerie, J., Shaw, P., & Vilím, P. (2018). IBM ILOG CP optimizer for scheduling. *Constraints, 23*(2), 210–250.
31. Hentenryck, P. V. (2002). Constraint and integer programming in OPL. *INFORMS Journal on Computing, 14*(4), 345–372.
32. (2017). *IBM ILOG CPLEX Optimization studio, Getting Started with Scheduling in CPLEX Studio*. IBM. https://www.ibm.com/docs/en/SSSA5P_12.8.0/ilog.odms.studio.help/pdf/sched_gs.pdf
33. Ham, A. M., & Cakici, E. (2016). Flexible job shop scheduling problem with parallel batch processing machines: Mip and cp approaches. *Computers & Industrial Engineering, 102*, 160–165.
34. Belov, G., Stuckey, P. J., Tack, G., & Wallace, M. (2016). Improved linearization of constraint programming models. In M. Rueher (Ed.), *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science* (pp 49–65). Toulouse, France: Springer.
35. Stuckey, P. J., Feydy, T., Schutt, A., Tack, G., & Fischer, J. (2014). The minizinc challenge 2008–2013. *AI Magazine, 35*(2), 55–60.
36. Zwicker, W. S. (2016). Introduction to the theory of voting. In F. Brandt, V. Conitzer, U. Endriss, J. Lang, & A. D. Procaccia (Eds.), *Handbook of computational social choice* (chapter 2, pp. 23–56). Cambridge: Cambridge University Press.
37. Lackner, M.-L., Musliu, N., & Winter, F. (2022b). Solving an industrial oven scheduling problem with a simulated annealing approach. In *Proceedings of the 13th international conference on the practice and theory of automated timetabling - PATAT 2022 - Volume III* (pp 115–120)
38. Velez Gallego, M. C. (2009). *Algorithms for scheduling parallel batch processing machines with non-identical job ready times*. PhD thesis, Florida: International University

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.