Automated Production Scheduling for Artificial Teeth Manufacturing

Abstract

In industrial artificial teeth manufacturing, nowadays a high level of automation is required to produce a large quantity of teeth in short production cycles. As a large variety of different product shapes and colors have to be processed on a single machine, the creation of efficient production schedules becomes a very challenging task. Due to the complexity of the problem and several cost minimization objectives that need to be considered, there is a large potential to improve the currently manually created schedules with automated solution methods.

In this paper, we formally specify and solve a novel challenging real-life machine batch scheduling problem from the area of artificial teeth manufacturing. Additionally, we provide a collection of real-life benchmark instances that can be used to evaluate solution methods for the problem.

To efficiently solve the problem, we propose an innovative construction heuristic and metaheuristic approach as well as an exact method using constraint programming. An extensive experimental evaluation shows that exact techniques can efficiently solve small scheduling scenarios and provide optimal solutions for three instances. Furthermore, we show that the proposed metaheuristic approach is able to reach optimal results for several small instances and can find high quality solutions also for large real-life benchmark instances.

Introduction

Modern-day production sites for artificial teeth manufacturing use an automated production process to produce large quantities of teeth in a variety of different shapes and colors. To efficiently handle large-scale production, batches of product moulds are usually simultaneously processed on a single machine. As resource constraints have to be respected and different machine programs need to be utilized for varying product families, creating cost efficient batches becomes a very challenging task in practice.

Currently, schedules are usually manually created to minimize job tardiness as well as costs caused by setup times and overproduction. However, due to the complexity of the problem and the multiple cost objectives that should be minimized, there is a large potential to save costs by using automated scheduling solutions.

A variety of batch scheduling problems, which share the goal to efficiently schedule batches of jobs onto machines, have been the subject of intensive study in the past. (Potts and Kovalyov 2000) provided an overview and categorization of earlier NP-hard batch scheduling variants for several single machine and parallel machine environments. Since then researchers studied further problem variants and investigated innovative solution methods for many real-life scheduling problems that arise from different application domains.

For example, a recent publication by (Polyakovskiy, Thiruvady, and M'Hallah 2020) studied a just-in-time batch scheduling problem that aims to minimize tardiness as well as earliness objectives and was shown to be NP-hard in (Hazır and Kedad-Sidhoum 2014). Furthermore, novel practical single machine scheduling problems from the industry have been investigated recently. (Zhao et al. 2020) for example investigate a batch scheduling problem from the steel industry which considers sequence-dependent setup times, release times and due time constraints where batches of jobs are predetermined in advance. Recently, (Tang and Beck 2020) further investigated a complex two-phase batch scheduling problem from the composites manufacturing industry which they approach using constraint programming, heuristics, and hybrid techniques. Further NP-hard single machine scheduling problem variants that do not include batching decisions but aim to minimize tardiness and setup time similarly as it is done in artificial teeth manufacturing, have been recently investigated for example by (Niu et al. 2019) and by (de Weerdt, Baart, and He 2020).

In this work we introduce for the first time the artificial teeth scheduling problem (ATSP), which is a novel single machine batch scheduling variant that appears in real-life production plants of the artificial teeth manufacturing area. While the single machine batch scheduling problem variants that have been investigated in the recent literature are given a predetermined set of jobs as part of the input, instances of the ATSP include customer demands but do not specify any job information. Therefore, novel solution methods are required that do not only design efficient batches, but also create jobs that efficiently fulfill all customer demands. Additionally, approaches to the ATSP have to consider several constraints which impose restrictions on feasible schedules and need to further optimize multiple cost objectives including an objective that aims to minimize waste caused by overproduction.

The following list summarizes the main contributions of this paper:

- We introduce and formally specify a novel challenging real-life machine scheduling problem from the area of artificial teeth manufacturing.
- We provide a collection of 12 instances that contain challenging real-life scheduling scenarios. We make all instances publicly available so that they can be used as benchmarks by the scientific community.
- We propose an innovative construction heuristic that can quickly produce feasible schedules to real-life problems. As this solution method was developed in collaboration with expert practitioners to automatize the manual planning process we use it as a baseline approach in this work.
- We propose a constraint programming model that can be utilized with state-of-the-art solvers as an exact method. Using this approach we provide optimality results for three of the benchmark instances.
- We develop a metaheuristic approach using local search which includes 7 innovative neighborhood operators and is able to provide high-quality solutions for large real-life scheduling scenarios.
- We evaluate all of our methods empirically, and show that the metaheuristic and exact approach can successfully improve the baseline results.

In the following section, we provide a description and formal specification of the ATSP. Afterwards, we first describe the construction heuristic before we later introduce a constraint model for the problem and propose a metaheuristic approach to solve the ATSP. At the end of the paper we present and discuss computational results and finally give concluding remarks.

Problem Definition

To efficiently produce a large number of artificial teeth, many teeth are usually processed simultaneously in batches. Therefore, each job in a production schedule uses a number of different product moulds to produce teeth. Such a mould essentially produces a certain tooth shape and is associated to a product line so that all moulds that belong to the same line form a family of related shapes. However, a job in the schedule needs to additionally decide which color should be applied to each of the produced teeth and therefore the final tooth product type is determined not only by its product line but also by the applied color.

Each job is further configured by a length- and production program parameter. The length parameter sets the number of production cycles of the job that determines the total number of produced teeth. Note that each cycle produces the same teeth using the moulds which are assigned to the job. The production program parameter determines how many moulds are simultaneously processed by the job, which mould types are compatible, and the processing time of a single production cycle. As every production program requires a fixed amount of moulds to be processed per cycle, it might be necessary to produce more teeth than necessary in some job cycles. Usually this cannot be completely avoided, therefore one of the problem's goals is to minimize the amount of waste caused by excessively produced teeth.

Consecutively scheduled jobs may either use different production programs or share the same program with a different set of mould and or color assignments. In any case a setup time is required between jobs, however if different production programs are used a longer setup time is required.

Finally, the main goal of the ATSP is to create a schedule that fulfills all given customer demands by creating jobs in a way that the makespan, total tardiness, and produced waste is minimized.

Figure 1 further illustrates the problem, by visualizing a schedule with three jobs for a small example instance.

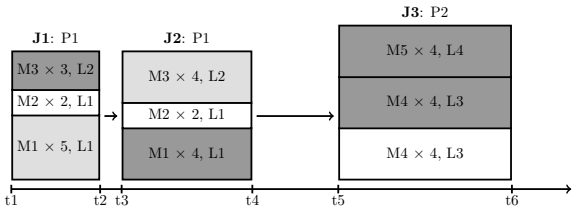

Figure 1: A small example schedule for the ATSP.

The figure shows three jobs J1, J2, and J3 being scheduled on the horizontal time line. Time points t1, t3, and t5 indicate the starting times of each job, whereas timepoints t2, t4, and t6 denote the corresponding job end times (in this case the total makespan is t6-t1). Jobs J1 and J2 both use the production program P1, whereas job J3 uses a different program P2. Note that the setup time between jobs J1 and J2 (visualized by the lengths of the horizontal arrows between jobs) is much smaller than it is between J2 and J3, as J1 and J2 both use program P1, but J2 and J3 use different programs. Furthermore, the the horizontal length of the jobs indicates the number of assigned production cycles. Therefore, J2 uses more cycles than J1.

As the production program defines the total number of assigned moulds, we can see in Figure 1 that J1 and J2 both use a total of 10 moulds, whereas J3 uses a total of 12 moulds. Mould types M1, M2, and M3 are in this case compatible only with program P1, and mould types M4 and M5 are associated to P2. We can further see in the figure, that each mould type is associated to a certain product line (e.g. M3 corresponds to line L2), and that the same mould type may be used with different colorings within the same job (e.g. in J3 mould type M4 is used in white color and gray color). Note that any two colors may only be used within the same job if they are compatible. Which pairs of colors are compatible is specified as part of the problem's input.

We now provide the full formal specification of the ATSP in the following sections. For simplicity we make use of the Iverson bracket notation[1].

Input parameters

The following parameters describe instances of the problem:

[1] $[P] = 1$, if $P = true$ and $[P] = 0$ if $P = false$

Description	Parameter
Set of colors	C
Set of programs	P
Set of mould types	M
Set of product lines	L
Set of demands	D
Setup time between identical programs	$sj \in \mathbb{N}$
Setup time between different programs	$sp \in \mathbb{N}$
Max product types per job	$w \in \mathbb{N}_{>0}$
Min cycles per job	$c_{\min} \in \mathbb{N}_{>0}$
Max cycles per job	$c_{\max} \in \mathbb{N}_{>0}$
Number of available moulds per type	$a_m \in \mathbb{N}\, \forall m \in M$
Number of mould slots per program	$am_p \in \mathbb{N}\, \forall p \in P$
Cycle time per program	$t_p \in \mathbb{N}_{>0}\, \forall p \in P$
Admissible program per mould type	$p_m \in P\, \forall m \in M$
Product line of each mould type	$l_m \in L\, \forall m \in M$
Requested mould type per demand	$dm_d \in M\, \forall d \in D$
Requested mould quantity per demand	$dq_d \in \mathbb{N}_{>0}\, \forall d \in D$
Due date of each demand	$dd_d \in \mathbb{N}\, \forall d \in D$
Requested color for each demand	$dc_d \in C\, \forall d \in D$
Set of compatible colors per color	$comp_c \in 2^C\, \forall c \in C$

Table 1: Input parameters of the ATSP

Variables

We define the following variables for the ATSP:

- Number of assigned jobs: $j \in \mathbb{N} \quad J = \{1, \ldots, j\}$
- Program assigned to each job: $jp_i \in P \quad \forall i \in J$
- Length of each job (i.e. the number of cycles):
$$jl_i \in \mathbb{N}_{>0} \quad \forall i \in J$$
- The mould types (with color) assigned to each job:
$$jm_{i,m,c} \in \mathbb{N} \quad \forall i \in J, m \in M, c \in C$$
- The total mould types (with color) produced by each job:
$$totaljm_{i,m,c} = jm_{i,m,c} \cdot jl_i \quad \forall i \in J, m \in M, c \in C$$

Constraints

Several constraints impose restrictions on feasible schedules:

- The number of assigned moulds to each job must be equal to the number of mould slots of the job's program:
$$\sum_{m \in M} \sum_{c \in C} jm_{i,m,c} = am_{(jp_i)} \quad \forall i \in J$$
- The number of scheduled moulds per job must not exceed mould availability:
$$\sum_{c \in C} jm_{i,m,c} \leq a_m \quad \forall i \in J, m \in M$$
- The number of different product types within a single job must be less than or equal to the allowed maximum:
$$\sum_{c \in C} \sum_{l \in L} \sum_{m \in M} ([l_m = l] \cdot [jm_{i,m,c} > 0]) \leq w \quad \forall i \in J$$
- All demands need to be fulfilled:
$$\sum_{d \in D} [dm_d = m \wedge dc_d = c] \cdot dq_d \leq \sum_{i \in J} totaljm_{i,m,c}\, \forall m \in M, c \in C$$
- Job moulds must be compatible with the job's program:
$$\sum_{c \in C} jm_{i,m,c} \cdot [jp_i \neq p_m] = 0 \quad \forall i \in J, m \in M$$
- A single job must not use incompatible colors:
$$\left[\sum_{m \in M} jm_{i,m,c_1} > 0\right] \leq \left[\sum_{m \in M} jm_{i,m,c_2} = 0\right]$$
$$\forall i \in J, c_1 \in C, c_2 \in (C \setminus comp_{c_1})$$

Objective Function

For the formal definition of the objective function we introduce the following auxiliary variables:

- The processing time for each job: $jt_i \in \mathbb{N}_{>0} \quad \forall i \in J$
- The finishing time for each job: $je_i \in \mathbb{N}_{>0} \quad \forall i \in J$
- The finishing job for each demand (after completion the demand is fulfilled): $dj_d \in J \quad \forall d \in D$

Several constraints set the values of the auxiliary variables:

- Set the job times: $jt_i = jl_i \cdot t_{(jp_i)} \quad \forall i \in J$
- Set job finishing times:
$$je_i = jt_1 + \sum_{k=2}^{i} (jt_k + sj + [jp_k \neq jp_{k-1}] \cdot (sp - sj))\, \forall i \in J$$
- Set demand finishing jobs:
$$\sum_{i=1}^{dj_d} totaljm_{i,m,c} \geq \sum_{d' \in D'} dq_{d'} \quad \forall d \in D \text{ where } m = dm_d,$$
$$c = dc_d, D' = \{d' \in D | dd_{d'} \leq dd_d \wedge dm_{d'} = m \wedge dc_{d'} = c\}$$

Using these auxiliary variables, the objective function aims to minimize three solution objectives:

1. The last job should be finished early to minimize the total **makespan** of the schedule: $ms = je_j$
2. The number of excessively produced moulds which are not consumed by any demand are considered to be **waste** and should be minimized:
$$waste = \sum_{i \in J} \sum_{m \in M} \sum_{c \in C} totaljm_{i,m,c} - \sum_{d \in D} dq_d$$
3. The total **tardiness** of all demands in the schedule should be mimized:
$$tard = \sum_{d \in D} \max(0, je_{(dj_d)} - dd_d)$$

Finally, we aggregate all three objectives in a normalized weighted sum where the objectives marked with * denote the costs of a given reference solution and w_{1-3} are weight parameters:

$$minimize \quad \frac{w_1 \cdot ms}{ms*} + \frac{w_2 \cdot waste}{waste*} + \frac{w_3 \cdot tard}{tard*}$$

Note that the reference solution costs have to be provided individually for each benchmark instance. Parameters w_{1-3} are then used to configure the relative importance of the three individual objectives. In practice, the weight parameters and reference solution costs can be configured according to the practical use case.

To evaluate the methods proposed in this paper with the set of real-life benchmark instances, we decided together with expert practitioners to set $w_{1-3} = 1$ as in the particular practical scheduling scenarios all three objectives are considered to be of similar importance. Furthermore, we use the construction heuristic approach that we propose in the next section, to generate all reference solution costs. As this method was developed in collaboration with domain experts to automatize the manual planning process, it represents a good baseline for the quality of current practical results.

Construction Heuristic Approach

In this section we propose a construction heuristic to quickly build feasible schedules for instances of the ATSP.

The main idea is to consecutively create jobs by greedily fulfilling the demands which are ordered by their due dates. In other words, the next job is configured to fulfill the next most urgent demand as quickly as possible, using feasible mould type and color assignments.

Algorithm 1 presents the detailed procedure of the construction heuristic.

Algorithm 1: A construction heuristic for the ATSP

```
fn CreateSchedule
    schedule = Initialize empty schedule
    sorted_demands = sort demands by due date
    while sorted_demands.Count() > 0 do
        d = sorted_demands.GetNext()
        program = p_(dm_d)
        length = ⌈dq_d / a_m⌉
        length = min{c_max, max{length, c_min}}
        j = Create new job j
        Update used moulds of j to fulfill d
        Remove d from sorted_demands
        for d' ∈ sorted_demands do
            if j has no more unused moulds left then
                exit loop
            if d' is compatible with job j then
                Update used moulds of j to fulfill d'
                Update sorted_demands
        while j has free remaining mould slots do
            Update j to use any available mould
        schedule.AppendJob(j)
return schedule
```

The algorithm first initializes an empty schedule and sorts the list of demands by their due date. Afterwards, the procedure creates jobs to fulfill demands in a loop until the list of sorted demands is empty. Within the outer while loop, the algorithm selects the next most urgent demand, and determines which program the job needs to use to fulfill the demand. Furthermore, the number of job cycles that are required to fulfill the demand is calculated based on the number of available moulds per cycle, as well as the minimum and maximum job length. The job is then created, and the number of used moulds within the job is updated accordingly. Additionally, the algorithm removes the processed demand from the list of remaining demands.

The newly created job is at this point likely to be only partially filled with moulds, and the selected program may require further moulds to be attached to this job. The inner for loop therefore goes over the complete list of sorted demands to look for other demands that could be fulfilled by this job. Thereby, for each demand it has to be checked if the required mould is still available in the job and the demanded product is compatible to the other already scheduled products so that no hard constraint would get violated. If the demand is compatible, the heuristic updates the remaining demand quantity as well as the assigned job moulds accordingly.

After the for loop, it can still be the case that free mould slots are left in the job, as no more compatible demands exist. In this case the algorithm simply fills any remaining unused mould slots by using any available mould types. The outer while loop ends by appending the newly created job at the end of the schedule. The overall job creating procedure continues until no more demands are left and afterwards the schedule is returned.

Constraint Modeling Approach

In this section we propose a constraint programming formulation for the ATSP using the input parameters from Table 1.

Model Variables

The model we propose uses several arrays of decision variables, where the length of many arrays is dependent on the maximum number of jobs that can be scheduled. The problem instances do not set any restrictions on the number of jobs, however an arbitrary number of jobs can lead to an unnecessary blow up of the variables in the model. Therefore, we set the maximum number of possible jobs based on a user defined model parameter max_jobs and in the following refer to the set of possible job ids as $J = \{1, \ldots, max\_jobs\}$.

We use the following variable arrays in our model:

- $jp_i \in \{0, \ldots, |P|\} \quad \forall i \in J$
- $jl_i \in \{c_{min}, \ldots, c_{max}\} \quad \forall i \in J$
- $jm_{i,m,c} \in \{0, \ldots, \max\{am_p | p \in P\}\} \quad \forall i \in J, m \in M, c \in C$
- $tjm_{i,m,c} \in \{0, \ldots, \max\{am_p \cdot c_{max} | p \in P\}\}$
  $\forall i \in J, m \in M, c \in C$

Variable arrays jp and jl determine the selected program, as well as the number of selected cycles for each job. Note that the domain of jp includes 0 to indicate that a job variable should be ignored, allowing the formulation to use less then the maximum number of jobs.

Variable arrays jm and tjm determine which moulds and colors are assigned to each of the jobs. The upper bound of these variable domains is calculated by the maximum number of possible program slots.

In addition to the mentioned variables, the model we propose uses a set of auxiliary variable arrays that are used in the formulation of the cost objectives. To efficiently set the domains of these auxiliary variables, we calculate several lower and upper bounds based on the input parameters:

- $lb\_end = c_{min} \cdot \min\{t_p | p \in P\}$
- $ub\_end = max\_jobs \cdot c_{max} \cdot \max\{t_p | p \in P\} + max\_jobs \cdot sp$
- $ub\_time = \max\{t_p | p \in P\} \cdot c_{max}$
- $ub\_waste = \max\{am_p | p \in P\} \cdot c_{max} \cdot max\_jobs$
- $ub\_tardiness = \sum_{d \in D} \max\{0, ub\_end - dd_d\}$

lb_end and ub_end define lower and upper bounds on the job end times in the schedule based on the minimum and maximum values regarding the number of cycles and cycle processing times. ub_time defines a bound on the maximum job processing time, whereas ub_waste and $ub\_tardiness$ provide upper bounds on the total waste and tardiness costs based on the maximum number of scheduled moulds and the maximum job end time.

Using these bounds we define auxiliary variables:

- $je_i \in \{lb\_end, \ldots, ub\_end\} \quad \forall i \in J$
- $jt_i \in \{0, \ldots, ub\_time\} \quad \forall i \in J$
- $de_d \in J \quad \forall d \in D$
- $ms \in \{lb\_end, \ldots, ub\_end\} \quad \forall d \in D$
- $waste \in \{0, \ldots, ub\_waste\} \quad \forall d \in D$
- $tard \in \{0, \ldots, ub\_tardiness\} \quad \forall d \in D$

The je, jt, and de variable arrays capture the job end times, job processing times, and demand end jobs. The ms, $waste$, and $tard$ variables capture the individual cost objectives.

Model Constraints

We use a high-level CP modeling notation to declare the constraints of the problem. Most parts of the model are directly solvable with constraint programming solvers, however at some points we implicitly make use of constraint reification to express conditional sums and logical implications. Furthermore, we implicitly utilize the element constraint to use variables as indices for array access, and make use of the maximum global constraint.

The following constraints are used in our formulation:

- We break symmetrical job assignments by aligning unused jobs at the end of the schedule and setting the length of unused jobs to the minimum domain value:

$$(jp_i = 0) \Rightarrow (jp_{i+1} = 0) \quad \forall i \in \{1, \ldots, max\_jobs - 1\}$$
$$(jp_i = 0) \Rightarrow (jl_i = c_{min}) \quad \forall i \in J$$

- Check that the amount of assigned job moulds is compatible with the program (we set $am_0 = 0$):

$$\sum_{m \in M} \sum_{c \in C} jm_{i,m,c} = am_{(jp_i)} \quad \forall i \in J$$

- The amount of available moulds must not be exceeded:

$$\sum_{c \in C} jm_{i,m,c} \leq a_m \quad \forall i \in J, m \in M$$

- The number of product types must not be larger than the allowed maximum:

$$\sum_{c \in C} \sum_{l \in L} \left[\sum_{m \in M} ([l_m = l] jm_{i,m,c}) > 0 \right] \leq w \; \forall i \in J$$

- Channel the tjm and jm variables:

$$tjm_{i,m,c} = jm_{i,m,c} \cdot jl_i \quad \forall i \in J, m \in M, c \in C$$

As these constraints are bilinear, we additionally used an alternative linearized version in our implementation. To achieve the linearization we applied a binary encoding of bilinear constraints as described by (Gupte et al. 2013).

- Ensure that all demands are fulfilled:

$$\sum_{d \in D} [dm_d = m \wedge dc_d = c] dq_d \leq \sum_{i \in J} tjm_{i,m,c} \; \forall m \in M, c \in \bigcup_{d \in D} dc_d$$

- Moulds have to be compatible with the job's program:

$$(jp_i \neq p_m) \Rightarrow (jm_{i,m,c} = 0) \quad \forall i \in J, m \in M, c \in C$$

- Only compatible colors may be assigned to the same job:

$$\left(\sum_{m' \in M} jm_{i,m',c_1} > 0 \right) \Rightarrow (jm_{i,m,c_2} = 0)$$
$$\forall i \in J, m \in M, c_1 \in C, c_2 \in C \setminus comp_{c_1}$$

- Set the job time variables:

$$(jp_i = p) \Rightarrow (jt_i = jl_i \cdot t_p) \quad \forall i \in J, p \in P$$
$$(jp_i = 0) \Rightarrow (jt_i = 0) \quad \forall i \in J$$

- Set the job end time variables:

$$(jp_i > 0) \Rightarrow$$
$$\left(je_i = jt_1 + \sum_{k=2}^{i} (jt_k + sj + [jp_{k-1} \neq jp_k] \cdot (sp - sj)) \right) \quad \forall i \in J$$
$$(jp_i = 0) \Rightarrow (je_i = 0) \quad \forall i \in J$$

- Set demand end job variables:

$$jp_{(de_d)} > 0 \quad \forall d \in D$$
$$\sum_{i=1}^{de_d} tjm_{i,(dm_d),(dc_d)} \geq \sum_{d' \in D'} dq_{d'} > \sum_{i=1}^{de_d - 1} tjm_{i,(dm_d),(dc_d)}$$
$$\forall d \in D, D' = \{d' \in D | dd_{d'} \leq dd_d \wedge dm_{d'} = dm_d \wedge dc_{d'} = dc_d\}$$

- Set the makespan: $ms = maximum(je)$
- Set the total waste:

$$waste = \sum_{i \in J} \sum_{m \in M} \sum_{c \in C} tjm_{i,m,c,} - \sum_{d \in D} dq_d$$

- Set total tardiness: $tard = \sum_{d \in D} maximum(\{0, je_{(de_d)} - dd_d\})$

Model Objective Function

The objective function aggregates the ms, $waste$, and $tard$ variables in a normalized weighted sum the same way as we have described it in the problem specification section.

Metaheuristic Approach

In this section, we propose a local search based metaheuristic approach for the ATSP. We first describe the solution representation, cost function, and the generation of initial solution. Afterwards, we propose several search neighborhoods for the problem, and describe how random neighborhood moves are generated in each search iteration. Finally, we present our neighborhood move acceptance criteria that is used to escape local optima.

Solution Representation and Cost Function

In our metaheuristic approach we represent solutions similarly as we did in the constraint model by using three variable arrays to store the assigned programs for each job, the length of each job, as well as the mould and color assignments assigned for each job. Therefore, we need to provide a parameter max_jobs that determines the length of these arrays and thereby limits the maximum number of jobs.

To determine the costs of candidate solutions, we use the previously defined normalized objective function but extend it in a way that it additionally captures potential hard constraint violations as follows:

$$cost(S) = \frac{ms}{ms*} + \frac{waste}{waste*} + \frac{tard}{tard*} + HC \cdot M$$

The function $cost(S)$ calculates the costs of a candidate solution S by adding the number of total hard constraints

violations HC multiplied by a big constant M to the normalized objectives, where M should ideally be larger than the worst case normalized objective costs. As we normalize our objectives using a reference solution it suffices to set M to a very large integer in practice.

To determine HC we further need to define for each hard constraint how we actually count the number of violations. Regarding mould availability, we can simply count the number of assigned moulds that are unavailable. For unfulfilled demands, we count the number of missing moulds. If any incompatible colorings are assigned to a job, we count the total mould quantities that use any of the incompatible colors. We further count any mould quantities that are incompatible with the selected program. If the number of allowed product types is exceeded in a job, we first calculate the mould quantities for each product type assigned to the job. Afterwards, we count the n lowest product type quantities as violations, where n is the difference between the allowed maximum number of product types and the actually number of assigned product types. Finally, in case too many moulds are assigned to a job we simply count the excessive mould quantities.

To generate an initial candidate solution for our metaheuristic approach we consider two options: We can either start search from an empty schedule or use our construction heuristic to produce an initial schedule.

Search Neighborhoods

In the following we propose seven search neighborhoods for our local search approach:

1. **Swap two jobs:** Swaps the positions of two existing jobs
2. **Increment length:** Increments the cycles of a job by 1
3. **Decrement length:** Decrements the cycles of a job by 1
4. **Change single mould assignment:** Changes a single assigned mould type and/or color to a different mould type and/or color within the same job.
5. **Delete last job:** Deletes the last job in the schedule
6. **Append new job:** Appends a new job at the end of the current schedule. Move parameters define the job program, as well as the mould quantities that should be used in the newly created job.
7. **Swap mould assignments between two jobs:** Swaps a single mould type and/or color assignment from a job with a single mould and/or color assignment from another job.

Note that neighborhoods 2-6 would suffice to reach all possible solution. However, the additional swap neighborhoods (1 and 7) have the advantage that they can swap mould assignments and reposition jobs without violating any demand constraints in intermediate solutions.

We only allow the insertion and deletion of jobs at the end of the schedule mainly for the purpose of an efficient move generation. Note that the insertion of jobs is mainly motivated to handle unfulfilled demand violations, while the deletion of jobs is mainly motivated to lower the makespan and waste objective. Therefore, the purpose of these neighborhoods does not directly rely on the job position.

Neighborhood Exploration

Exploring the complete neighborhood easily becomes computationally expensive, especially when dealing with large real-life instances. Therefore, we do not explore the full neighborhood in our approach, but instead randomly select only a single move out of the complete neighborhood in each iteration.

Which move is generated, is determined based on a random selection procedure that is configured by parameters N_1-N_7. Each parameter N_1-N_7 defines a real value between 0 and 1 that determines the probability to consider each of the seven neighborhoods in a iteration. We determine in each iteration a single random move in 3 steps: First, for each neighborhood we randomly decide based on the associated parameter whether or not it is considered for move generation. Afterwards, we randomly select one of the neighborhoods that have been selected in the previous step. Finally, we uniformly sample a single move from the chosen neighborhood.

Move Acceptance

Once a single random move has been generated, we evaluate the change of the current solution's quality that would be caused by the move. Based on the result we then decide whether or not the move should be applied to the current solution.

In our approach we use a move acceptance function that implements simulated annealing (Kirkpatrick, Gelatt, and Vecchi 1983). The function ensures that a cost improving move is always accepted, whereas a non-cost-improving move is only accepted with a certain probability that depends on the change in solution quality as well as a temperature value. We set the temperature value at the beginning of local search to a user defined parameter. Afterwards, we use a geometrical cooling scheme that decreases the temperature value after each search iteration by multiplication with a user defined cooling rate parameter.

Algorithm 2 presents the full acceptance function, where $cost(S)$ is the cost of the current solution, $cost(S*)$ is the cost after the application of the randomly generated move, T is the current temperature value, and $random()$ is a uniformly sampled real value between 0 and 1.

Algorithm 2: Move acceptance function

```
fn AcceptMove(cost(S), cost(S*), T)
    result = True
    if cost(S) ≤ cost(S*) then
        p = e^(−(cost(S*)−cost(S)) / T)
        if random() > p then
            result = False
return result
```

Computational Results

In this section we first describe the experimental environment and parameter configuration before we later present and discuss computational results. All of our experiments as well as the parameter tuning were conducted on 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.

To experimentally evaluate our methods we received 6 problem instances from our industry partners that represent real-life scheduling scenarios as they appeared at production sites of artificial teeth manufacturing. Early experiments with these instances showed that all of them cause a very large search space, which makes it hard for exact methods to reach results within reasonable runtime. Therefore, we additionally generated 6 smaller instances by randomly selecting 25% of the colors and mould types together with associated demands for each of the realistic instances. Table 2 displays size parameters of all 12 benchmark instances, where instances I1 - I6 form the small instance set and instances I7 - I12 are the large real-life scheduling scenarios.

Inst.	C	M	D	L	P	Vars	CS
I 1	5	38	20	4	2	12649	14243
I 2	4	28	24	3	1	15475	17442
I 3	4	16	7	1	1	3242	4045
I 4	5	38	4	4	2	5849	6048
I 5	4	28	9	3	1	6840	7646
I 6	3	16	1	1	1	1247	1447
I 7	22	153	799	4	2	621749	583209
I 8	18	114	390	3	1	372409	337268
I 9	18	64	285	1	1	135664	149647
I 10	22	153	190	4	2	373599	302936
I 11	18	114	224	3	1	294675	253199
I 12	13	64	36	1	1	50463	45298

Table 2: Size parameters of the used benchmark instances.

Columns 2-6 from table 2 provide information about the number of colors, mould types, demands, production lines and programs. Furthermore, Columns 7 and 8 display the number of generated variables using the bilinear model.

In a first series of experiments, we used our construction heuristic to produce reference solutions for all of the instances (i.e. the aggregated normalized solution cost is always exactly 3 for the construction heuristic). The max_job parameter was set for the constraint model to 50 and for the metaheuristic approach to 500 (50 should clearly suffice for all benchmark instances, however for the metaheuristic we could even use a higher value without risking memory leaks).

To evaluate the metaheuristic approach we further have to configure several parameters. Based on some manual tuning attempts we selected a T value of 0.001 and a α value of 0.999 and set all neighborhood probabilities to 1 as the default. Starting from the default values, we further used the state-of-the-art parameter tuning software SMAC (Lindauer et al. 2017) to automatically tune all of the parameters (Parameter value ranges were restricted to $T \in [0.0001, 2]$, $\alpha \in [0.9, 0.9999]$, and $N1 - N7 \in [0, 1]$). The tuning process was then started with the metaheuristic that starts from an initial reference solution and all 12 instances as the training set. We restricted the runtime limit for each individual run to 10 minutes and set the overall wallclock time limit to 4 days. The resulting parameter configuration which we used for our final experiments is as follows: $T = 0.4735, \alpha = 0.9274, N1 = 0.2042, N2 = 0.0407, N3 = 0.8522, N4 = 0.0632, N5 = 0.8630, N6 = 0.6250$, and $N7 = 0.3972$.

To evaluate exact approaches that utilize the proposed bilinear and linearized constraint programming model we implemented both models using the modeling language MiniZinc (Nethercote et al. 2007), which provides interfaces to state-of-the-art constraint programming (CP) and mixed integer programming (MIP) solvers (for the latter MiniZinc automatically converts the constraint model into a MIP model).

We then performed experiments with the MIP solvers gurobi (Gurobi Optimization 2020) and cplex (Corporation 2019), as well as the CP solvers gecode (Gecode Team 2020) and chuffed (Chu 2011). As chuffed is not able to handle floating point objectives, we simply used non normalized values in the objective for chuffed and normalized the final objective in a post processing step.

For gecode and chuffed we further used a programmed search strategy that first selects all jm variables based on a smallest domain first heuristic where minimum values are assigned first. For the remaining variables, we use the solvers default search and further activated the free search parameter for chuffed which allows the solver to alternate between the given search strategy and its default one on each restart. For all evaluated exact and metaheuristic methods we set a time limit of 1 hour for each run. Numerical values have been rounded to two decimal places in all final results. Table 3 summarizes the final results produced by exact methods.

Inst.	Cpx L	Cpx B	Grb L	Grb B
I 1	3.37	2.96	**2.53**	**2.53**
I 2	2.23	3.67	**1.98**	1.99
I 3	**2.23**	**2.23**	**2.23**	**2.23**
I 4	**2.54**	**2.54**	**2.54**	**2.54**
I 5	2.12	2.11	2.11	**2.1**
I 6	**3**	**3**	**3**	**3**
Inst.	**Gce L**	**Gce B**	**Chu L**	**Chu B**
I 1	-	37.53	579.59	-
I 2	-	-	19.31	272.87
I 3	69.69	3	3	1123.83
I 4	37.87	37.87	162.23	619.5
I 5	43.85	45.11	37.96	-
I 6	**3**	**3**	**3**	**3**

Table 3: Summarized results for exact methods.

Note that Table 3 only displays results for instances 1-6 (I 1-6), as none of the exact approaches were able to reach feasible solutions within the time limit for instances 7-12. Each row shows the final normalized objective value reached for the corresponding instance with solvers cplex (Cpx), gurobi (Grb), gecode (Gce), and chuffed (Chu) using the bilinear channeling constraints (B) and the linearized channeling constraints (L). Best results for each row are formatted in bold face and a - denotes that no solution could be found within the time limit.

The results presented in Table 3 show that gurobi produces overall the best results for all instances. All approaches are able to reach the best objective value of 3 for instance 6

which is equal to the reference solution cost. As instance 6 contains only a single demand (see Table 2) this is an expected result. The results further show that the CP solvers gecode and chuffed seem to be not competitive compared to the MIP solvers for instances 1-5. We see that there are only small differences between the bilinear and linearized models, especially for gurobi. We assume this is due to the fact that gurobi recently introduced improved techniques for bilinear constraints.

Table 4 provides an overview on the best dual bounds (DB) by the evaluated MIP solvers. The best dual bounds per row are formatted in bold face.

Inst.	Cpx L DB	Cpx B DB	Grb L DB	Grb B DB
I 1	1.45	1.24	1.92	**2.11**
I 2	0.99	0.73	0.92	**1.08**
I 3	**2.23**	**2.23**	**2.23**	**2.23**
I 4	**2.54**	**2.54**	**2.54**	**2.54**
I 5	1.34	1.06	1.38	**1.65**
I 6	**3**	**3**	**3**	**3**
I 7	0.47	0.47	**0.49**	**0.49**
I 8	**0.14**	**0.14**	**0.14**	0.13
I 9	**0.58**	0.55	**0.58**	**0.58**
I 10	**0.53**	0.51	**0.53**	0.51
I 11	**0.6**	**0.6**	**0.6**	0.57
I 12	1.02	0.96	**1.14**	1

Table 4: Final dual bounds achieved by MIP methods.

We can see that for the large instances, the best dual bounds can be obtained with gurobi using the linearized model. For small instances on the other hand the bilinear model with gurobi produced the best results. This indicates that for large problems linearizing the constraints can be helpful to quickly obtain good dual bounds.

Table 5 summarizes results produced by the metaheuristic approach starting from an empty schedule (LS) and starting from an initial reference solution (LS/C). Note that the proposed method is not deterministic, as neighborhood moves are randomly generated in each iteration. Therefore, these results were obtained by 10 repetitive runs on each instance and the table displays in addition to the overall best cost (B) also the mean costs (M) and the standard deviation (S). The best mean costs per instance are formatted in bold face and a - denotes that no feasible solution was reached.

The results presented in Table 5 show that starting from an initial schedule produces the best mean costs for all instances. Furthermore, we can see that for the majority of the instances starting from an empty solution cannot reach any feasible solution within the runtime limit. This indicates that starting from a construction heuristic is very effective to deliver robust and good results especially for large instances.

Finally, Table 6 summarizes the overall best results produced with exact and metaheuristic methods.

We can see in the results that the exact methods could prove optimality for instances 3,4, and 6 and that metaheuristics could also reach optimal results in these cases. It seems that overall the exact methods produce results of similar quality compared to the metaheuristic approach on the smaller instances. However, we can clearly see that the metaheuristic performed better for the large instances.

Inst.	LS B	LS M	LS S	LS/C B	LS/C M	LS/C S
I 1	2.53	**2.53**	0	2.53	**2.53**	0
I 2	1.94	**1.95**	0	1.94	**1.95**	0
I 3	2.23	**2.23**	0	2.23	**2.23**	0
I 4	2.54	**2.54**	0	2.54	**2.54**	0
I 5	2.15	2.28	0.16	2.13	**2.16**	0.02
I 6	3	**3**	0	3	**3**	0
I 7	-	-	-	2.93	**2.97**	0.02
I 8	-	-	-	2.26	**2.37**	0.04
I 9	-	-	-	2.83	**2.92**	0.05
I 10	4.53	5.83	0.72	2.65	**2.83**	0.1
I 11	-	-	-	2.64	**2.66**	0.01
I 12	5.21	6.59	0.64	2.97	**2.99**	0.01

Table 5: Overview on computational results for local search.

Inst.	LB	Exact	LS
I 1	2.11	**2.53**	**2.53**
I 2	1.08	1.98	**1.94**
I 3	2.23	**2.23**	**2.23**
I 4	2.54	**2.54**	**2.54**
I 5	1.65	**2.1**	2.13
I 6	3	**3**	**3**
I 7	0.49	-	**2.93**
I 8	0.14	-	**2.26**
I 9	0.58	-	**2.83**
I 10	0.53	-	**2.65**
I 11	0.6	-	**2.64**
I 12	1.14	-	**2.97**

Table 6: Overview on overall results.

Conclusion

In this paper we have introduced a novel real-life single machine batch scheduling problem that appears in artificial teeth manufacturing. We further formally specified the problem and provided a collection of benchmark instances which contain real-life scheduling scenarios from the industry.

In collaboration with expert practitioners from the field we developed a construction heuristic that automatizes the existing manual planning process and used it to generate baseline solutions to the benchmark instances. Additionally, we proposed a constraint programming model that we utilized as an exact approach and as well as a metaheuristic approach that we use to solve large practical instances.

Computational results showed that the exact approach is able to prove three optimal results and produced the best results for five small instances, whereas the metaheuristic approach performed similarly on the small instances and produced the best results for the large benchmark instances.

The investigation of a large neighborhood search approach that hybridizes exact and metaheuristic results to further improve the results for large instances could be interesting for future work.

References

Chu, G. G. 2011. Improving combinatorial optimization .

Corporation, I. 2019. *IBM ILOG CPLEX 12.10 User's Manual*.

de Weerdt, M.; Baart, R.; and He, L. 2020. Single-machine scheduling with release times, deadlines, setup times, and rejection. *European Journal of Operational Research* .

Gecode Team. 2020. *Gecode: Generic Constraint Development Environment*. URL http://www.gecode.org.

Gupte, A.; Ahmed, S.; Cheon, M. S.; and Dey, S. 2013. Solving Mixed Integer Bilinear Problems Using MILP Formulations. *SIAM Journal on Optimization* 23(2): 721–744.

Gurobi Optimization, L. 2020. *Gurobi Optimizer Reference Manual*. URL http://www.gurobi.com.

Hazır, Ö.; and Kedad-Sidhoum, S. 2014. Batch sizing and just-in-time scheduling with common due date. *Annals of Operations Research* 213(1): 187–202.

Kirkpatrick, S.; Gelatt, C. D.; and Vecchi, M. P. 1983. Optimization by Simulated Annealing. *Science* 220(4598): 671–680.

Lindauer, M.; Eggensperger, K.; Feurer, M.; Falkner, S.; Biedenkapp, A.; and Hutter, F. 2017. *SMAC v3: Algorithm Configuration in Python*. GitHub.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards a Standard CP Modelling Language. In Bessière, C., ed., *Principles and Practice of Constraint Programming – CP 2007*, Lecture Notes in Computer Science, 529–543. Berlin, Heidelberg: Springer. ISBN 978-3-540-74970-7.

Niu, S.; Song, S.; Ding, J.-Y.; Zhang, Y.; and Chiong, R. 2019. Distributionally robust single machine scheduling with the total tardiness criterion. *Computers & Operations Research* 101: 13–28.

Polyakovskiy, S.; Thiruvady, D.; and 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, 228–235. New York, NY, USA: Association for Computing Machinery.

Potts, C. N.; and Kovalyov, M. Y. 2000. Scheduling with batching: A review. *European Journal of Operational Research* 120(2): 228–249.

Tang, T. Y.; and 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, 431–446.

Zhao, Z.; Liu, S.; Zhou, M.; Guo, X.; and 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.