I L S 2020
INTERNATIONAL CONFERENCE ON INFORMATION
SYSTEMS, LOGISTICS & SUPPLY CHAIN
Austin, Texas April 22-24, 2020

Disjunctive Scheduling with Setup Times: Optimizing a Food Factory

Nicolas Blais [1], Alexis Remartini [1], Claude-Guy Quimper [1], Nadia Lehoux [1], Jonathan Gaudreault [1]

[1] Université Laval, Québec, Canada
{nicolas.blais.7@ulaval.ca, alexis.remartini.1@ulaval.ca, claude-guy.quimper@ift.ulaval.ca, nadia.lehoux@gmc.ulaval.ca, jonathan.gaudreault@ift.ulaval.ca}

Abstract. We propose a new approach to solve scheduling problems applied to a food factory. The goal is to schedule the recipes in a way that minimizes the total setup time. Our model allows splitting a recipe in two batches if necessary. It also considers other specific constraints like avoiding too long setup on a day shift, having a maximum of tasks done during days shift, etc. A good heuristic based on the idea of scheduling the most difficult tasks as soon as possible is presented.

Keywords: Scheduling, Constraint Programming, Minimizing Setup Times.

1. Introduction

The food industry is an environment with many standards to respect that can quickly become complex if multiple products are made on the same production line. In some factories, planners must manage the availability of the ingredients and of the production line in addition to the allergens of each product. They have to take into account the setup time between all the products which depends largely on the allergens of the products. The setup time, that includes the cleaning efforts, typically takes much longer between a recipe that contains allergens and a recipe that does not because workers must conduct an allergen clean-up for the entire production line which can involve many hours. This clean-up does not occur when passing from an allergen-free product to a product with allergens.
The aim of this work is to schedule a set of tasks for a planning horizon over the weeks 5 to 8 with the objective of minimizing the setup times. The first 4 weeks are not considered (frozen horizon) because changes at this time generate conflicts for the schedule and the procurement of raw materials. The schedule is recomputed weekly. The tasks are sequentially performed, i.e. we schedule on a single machine.
The following methodology was followed. First, an analysis of the problem was carried out to clearly identify the problem. Then, the collaboration with the company made it possible to obtain data and constitute a benchmark. We designed a model and improve the search strategy of a constraint solver to obtain the best possible results on the instances. The solutions were presented to the company (validation of the model) and modifications were made to satisfy the set of constraints that must be taken into account by the planners. We reduce by 26%, in average, the setup time compared to what the human planners did.
In the literature, there are a few papers that use constraint programming applied to the food industry. Our model contributes to the field by creating a set of constraints adapted to the branching strategy we propose.
This paper is organized as follows. Section 2 states the problem. Section 3 presents the literature surrounding our problem. The model and the branching strategy are presented in Section 4. The experimental results are presented in Section 5. Section 6 concludes the paper.

2. Problem Description

The problem was observed in a world recognized granola bar and cookie factory that offers a wide range of products, some of them being allergen-free. To help the planners manage such a complex product portfolio, we develop an application which takes their production system's dynamics into account.
In our notation, we use capital letters to denote decision variables and sets and small letters for parameters or constants. We consider a set of tasks T^* that correspond to cookie recipes. Each task $i \in T^*$ has a due date lct_i called Latest Completion Times, release date est_i called Earliest Starting Time which is computed according to the availability and the preservability of the ingredients, and a processing time p_i that is proportional to the number of cookies to produce. A task i has a starting time S_i which is unknown and needs to be computed. The transition time (or setup time) between a task i to a task j is given by $t_{i,j}$. This time takes into account the cleaning time and the machine reconfiguration. All production lines are stopped on the weekend and some are idle during the nights. These downtimes can be used to conduct a setup, but the production itself is stopped. Let J be set of days in the scheduling horizon. For each day d $\in J$, the working shift starts at time $shiftBegin_d$ and ends at time $shiftEnd_d$. There is only one shift per day.
There are two types of tasks: *production orders* and *planned orders*. The *production orders* are already scheduled and cannot be moved. We therefore have $est_i + p_i = lct_i$ which leaves no freedom on the time the task is scheduled: $S_i = est_i$. The *planned orders* need to be scheduled so the time window in which the task must be scheduled is not tight: $est_i + p_i < lct_i$. Moreover, planned orders can be executed in two segments in order to have a better use of the production line i.e. avoiding having unused time before the end of the shift. For instance, a task i can start on Monday, be stopped during the night and be resumed on Friday. At the end of its execution on Friday, the task must be completed, i.e., the time spent on Monday and Friday must sum up to the processing time p_i. Nothing forces the two tasks to be adjacent in the schedule i.e., there may be other tasks running between them. When a task is executed in two parts, the duration of each of its part must be greater or equal to a threshold *minDuration*. Tasks cannot be separated into more than two parts, because starting a new production can decline the productivity. Indeed, the first minutes of production are often unstable. So, separating a task into too many parts increases the chances of not getting the right amount of product in the allotted time.
There is a limit of *maxTaskShift* tasks that can be achieved during a workday (or shift) to avoid too many setups during a day. Only setups with duration below a threshold *maxSetup* are permitted during a working day in order to avoid idle workers and to maximize the use of daytime. A subset of tasks $T_M \subset T^*$ requires setups that can only be done on the weekend. Tasks in T_M can be scheduled on Mondays ($J_M \subset J$) or right after another task in T_M. All parameters and sets are summarized in Tables 1 and 2.

Table 1: List of Sets.

Sets	Definition
T^*	Set of all fictive tasks
T_M	Set of tasks that must be done on a Monday
J	Set of working days
J_M	Set of working days that are Mondays
T	Set of task parts
T_1	Set of the first parts of the tasks
T_2	Set of the second parts of the tasks

Table 2: List of Parameters.

Parameters	Definition
$t_{i,j}$	Transition time from a task i to a task j
est_i	Earliest starting time of the task i
lct_i	Latest completion time of the task i
p_i	Processing time needed for the task i
$shiftBegin_d$	Starting time of the working day d
$shiftEnd_d$	Ending time of the working day d
minDuration	Minimum duration of separated task
maxTaskShift	Number maximum of tasks during a shift
maxSetup	Maximum setup time allowed during day

3. Literature Review

Scheduling problems are numerous and varied. *Allahverdi* [12] presents a survey of scheduling problems with setup times. The notation used ($\alpha|\beta|\gamma$) classifies our problem in the category $1|ST_{sd}|TST$: 1 because there is a single machine, ST_{sd} for sequence-dependent setups and TST for minimizing the sum of the setup times. Few papers explore this problem [12].
The core of the studied problem has the same form as the Traveling Salesman Problem with Time Windows (TSPTW) which was proved to be NP-Hard [8]. Indeed, tasks can be represented as cities and setup times between tasks as distances between cities. The shortest Hamiltonian path provides an ordering of the tasks

that minimizes the sum of the setup times. Several methods have been developed to solve this problem. *Ángel-Bello et al.* [6] used Mixed Integer Programming MIP. *Abdallah et al.* [7] used a heuristic approach (Family Splitting Algorithm) to find quality solutions. *Fagerholt et al.* [5] use dynamic programming. *Claassen et al.* [3] uses a relax-and-fix heuristic. *Hebrard and Grimes* [4] use Constraint Programming (CP) to solve job shop problems with setup times. Given the range of possible solving methods, it becomes difficult to know which one is the best. *Ku and Beck* [1] compare MIP and CP methods. The results showed that CP outperforms MIP on larger instances. As a result, using CP for our case becomes a natural choice.
Constraint programming is a technology originating from artificial intelligence that solves combinatorial and optimization problems. It offers a large collection of constraints to model the problems: from simple constraints such as linear constraints to more complex ones that entirely model the usage of a resource in a scheduling problem. A *constraint satisfaction problem* is defined by a set of variables. Each variable X_i has a domain denoted dom(X_i). Each constraint is posted on a subset of variables and restricts their possible assignments. A solution is an assignment where each variable is given a value from its domain and each constraint is satisfied. In a *constraint optimization problem*, one also aims at minimizing/maximizing an objective function. Constraint solvers usually explore a search tree and perform a branch and bound to solve optimization problems under constraints. At each node of the tree, filtering algorithms associated with each constraint of the problem prune the branches that cannot lead to a solution.
Some constraints can be simple. For example, the ELEMENT constraint ensures that a variable X takes the value at index I in an array of variables Y, i.e. $X = Y[I]$. However, what makes constraint programming so efficient is the use of global constraints posted on many variables whose filtering algorithms significantly prune the search space. For example, the DISJUNCTIVE($[S_1, \ldots, S_n], [p_1, \ldots, p_n]$) constraint introduced in [9] prevents any pair of tasks *i* and *j* with starting time variables S_i and S_j and processing times p_i and p_j from executing simultaneously: $S_i + p_i \leq S_j \vee S_j + p_j \leq S_i$. This constraint uses filtering rules (e.g. time-tabling) to remove inconsistent values from the domain of the starting time variables. *Dejemeppe et al.* [2] created a variation of DISJUNCTIVE that takes into account setup times. The global constraint WEIGHTEDCIRCUIT($[N_1, \ldots, N_n]$, W) [15] accepts assignments that encodes a Hamiltonian cycle in a graph with a total weight smaller than or equal to W. The nodes are labeled with integers from 1 to *n*. The node next to node *i* in the cycle is N_i. The Global Cardinality Constraint [10] GLOBALCARDINALITY($[X_1, \ldots, X_n]$, $[l_1, \ldots, l_m]$, $[u_1, \ldots, u_m]$) ensures that the value v occurs between l_v and u_v times in the vector $\vec{X}$. Finally, meta constraints allow enforcing relations between constraints. For instance, in the constraints $C_1 \Longrightarrow C_2$ and IFTHENELSE(C_1, C_2, C_3), constraint C_2 is satisfied whenever constraint C_1 is satisfied and constraint C_3 is satisfied whenever constraint C_1 is violated.

4. Methods and model

4.1. Variables

We present a model that encodes our problem. The main decision variables are the starting time of a task $\in T$ denoted S_i and the task that follows a task i in the schedule denoted N_i. All other variables are auxiliary, which means their values are function of S_i and N_i. Table 3 summarizes the variables and their domains.

Table 3: List of variables.

Variable	Domain	Definition
W	$[0, \infty)$	Sum of all transition times
N_i	$T \setminus \{i\}$	Task next to task i
PR_i	$T \setminus \{i\}$	Task preceding task i
S_i	$[est_i, lct_i]$	Starting time of the second part of task i
P_{i1}	$[minDuration, p_i]$	Processing time of the first part of task i
P_{i2}	$[0, p_i - minDuration]$	Processing time of the second part of task i
$\tilde{P}_i$	$[0, p_i + \max_{j \in T^*} t_{i,j}]$	Processing time, including the transition to task next to *i*
D_i	J	Working day of the task i
D_i'	$J \cup \{-1\}$	Working day of the task i if $\tilde{p}_i > 0$ and -1 otherwise
TND_i	*shiftBegin*	Starting time of the upcoming morning of the task i
POS_i	[1, maxTaskShift]	Position of the task i in the shift

M_i	$shiftBegin \cup \{0\}$	Starting time of the morning following task *i* if the maximum number of tasks is reached and 0 otherwise.

Since the tasks are preemptive and can be executed in two parts, we declare a set of tasks *T* that contains two tasks i_1 and i_2 for each task *i* in T*. These tasks represent the first part and the second part of task *i*. We also have in *T* two virtual tasks, *sentinel-begin* and *sentinel-end,* that mark the beginning and the end of the schedule. Each task i_1, i_2 in *T* has starting time variables S_{i1} and S_{i2} with domain dom(S_{i1}) = dom(S_{i2}) = [est_i, lct_i]. They also have processing time variables $P_{i1}$and $P_{i2}$with domains dom(P_{i1}) = [*minDuration*, p_i] and dom(P_{i2}) = [0, p_i – *minDuration*]. When P_{i2} = 0, the task entirely executes in the first part. The *next* variable N_i indicates which task follows task *i* and has for domain dom(N_i) = $T \setminus \{i\}$. The variable PR_i represents the task that precedes task *i* and has the same domain as N_i. The variable $\tilde{P}_i$, for *i* in *T*, is duration of the task part *i* plus the setup time that follows the task. The *working day* variable D_i tells the day which a task part is done with *J* as domain. D_i' is used to represent the working day of task *i* if $P_i > 0$ and -1 otherwise. The value -1 occurs when *i* is the second part of a task that is entirely executed in its first part. The starting time of the upcoming morning of a task *i* TND_i can take any value in *shiftBegin*. The position of a task *i* in a shift is denoted with POS_i and has for domain dom(POS_i) = [1, *maxTaskShift*]. Finally, the variable M_i takes the value of the next morning of task *i* if the maximum number of tasks is reached and 0 otherwise. So, its domain is dom(M_i) = *shiftBegin* $\cup$ {0}.

4.2. Main Constraints

We present the constraints of the model that is summarized below. The objective function (1) minimizes the total setup time (*W*) i.e. the transition time t_{i,N_i} of a task *i* with its successor N_i for all task *i*.

$$\text{Minimize } W = \sum_{i \in T} t_{i,N_i}. \tag{1}$$

$$P_{i_1} + P_{i_2} = p_i. \qquad \forall\, i \in T^* \tag{2}$$

$$S_{i_1} + P_{i_1} \leq S_{i_2}. \qquad \forall\, i \in T^* \tag{3}$$

$$N_{i_2} \neq i_1. \qquad \forall\, i \in T^* \tag{4}$$

$$P_{i_1} = p_i \Longrightarrow N_{i_1} = i_2 \wedge D_{i_1} = D_{i_2}. \qquad \forall\, i \in T^* \tag{5}$$

$$P_{i_1} = \min\left(p_i,\ shiftEnd_{D_{i_1}} - S_{i_1}\right). \qquad \forall\, i \in T^* \tag{6}$$

$$S_{i_1} + p_i > shiftEnd_{D_{i_1}} \Longrightarrow S_{i_1} + p_i > shiftEnd_{D_{i_1}} + minDuration\ . \qquad \forall\, i \in T_1 \tag{7}$$

$$\tilde{P}_i = P_i + t_{i,N_i}. \qquad \forall\, i \in T \tag{8}$$

$$\textsc{Disjunctive}(S, \tilde{P}). \tag{9}$$

$$\textsc{WeightedCircuit}(N, t, W). \tag{10}$$

$$N_{\text{sentinel-end}} = \text{sentinel-begin}. \qquad \forall\, i \in T \tag{11}$$

$$D_i \geq d \Rightarrow S_i \geq shiftBegin_d. \qquad \forall\, i \in T, \forall\, d \in J \tag{12}$$

$$D_i \leq d \Rightarrow S_i \leq shiftEnd_d. \qquad \forall\, i \in T, \forall\, d \in J \tag{13}$$

$$\textsc{IfThenElse}(\ P_i > 0,\ D_i' = D_i,\ D_i' = -1). \qquad \forall\, i \in T \tag{14}$$

$$\textsc{GlobalCardinality}(D', maxTaskShift). \tag{15}$$

$$t_{i,j} > maxSetup \wedge D_i = D_j \Longrightarrow N_i \neq j. \qquad \forall\, i, j \in T \tag{16}$$

$$t_{i,j} > (maxTaskShift - 1)maxSetup \wedge t_{j,i} > (maxTaskShift - 1)maxSetup \Longrightarrow D_i \neq D_j. \qquad \forall\, i, j \in T \tag{17}$$

$$i = N_{PR_i} \qquad \forall\, i \in T \tag{18}$$

$$D_i \notin J_M \Longrightarrow PR_i \in T_M. \qquad \forall\, i \in T_M \tag{19}$$

$$TND_i = ShiftBegin_{D_i + 1} \qquad \forall\, i \in T \tag{20}$$

$$D_{N_i} = D_i \wedge P_{N_i} \neq 0 \Longrightarrow POS_{N_i} = POS_i + 1. \qquad \forall\, i \in T \tag{21}$$

$$D_{N_i} = D_i \wedge P_{N_i} = 0 \Longrightarrow POS_{N_i} = POS_i. \qquad \forall\, i \in T \tag{22}$$

$$D_{N_i} \neq D_i \Longrightarrow POS_{N_i} = 1. \qquad \forall\, i \in T \tag{23}$$

$$\text{IfThenElse}(POS_i = maxTaskShift \wedge p_{N_i} \neq 0,\ M_i = TND_i,\ M_i = 0). \qquad \forall\, i \in T \tag{24}$$

$$\begin{aligned} & N_i \in \{T_1\} \wedge [(p_{N_i} \geq 2\ minDuration \wedge shiftEnd_{D_i} - S_i - P_i \geq minDuration) \vee \\ & (p_{N_i} < 2minDuration \wedge shiftEnd_{D_i} - S_i - P_i \geq p_{N_i})] \qquad \forall\, i \in T \\ & \Longrightarrow S_{N_i} = \max(S_i + \tilde{P}_i,\ M_i,\ est_{N_i}). \end{aligned} \tag{25}$$

$$N_i \in \{T_1\} \wedge [(p_{N_i} \geq 2\text{minDuration} \wedge \text{shiftEnd}_{D_i} - S_i - P_i < \text{minDuration}) \vee (p_{N_i} < 2\text{minDuration} \wedge \text{shiftEnd}_{D_i} - S_i - P_i < p_{N_i})] \Rightarrow S_{N_i} = \max(TND_i, est_{N_i}). \quad \forall i \in T \quad \textbf{(26)}$$

$$N_i \in \{T_2\} \wedge [(p_{N_i} = 0) \vee (p_{N_i} \geq \text{minDuration} \wedge \text{shiftEnd}_{D_i} - S_i - P_i \geq p_{N_i})] \Rightarrow S_{N_i} = \max(S_i + \tilde{P}_i, M_i). \quad \forall i \in T \quad \textbf{(27)}$$

$$N_i \in \{T_2\} \wedge (p_{N_i} \geq \text{minDuration} \wedge \text{shiftEnd}_{D_i} - S_i - P_i < p_{N_i}) \Rightarrow S_{N_i} = TND_i. \quad \forall i \in T \quad \textbf{(28)}$$

A task *i* in *T** can be executed in two parts that are represented by two tasks i_1 and i_2 in *T*. Constraint (2) makes sure that the sum of the durations of the parts in *T* is equal to the duration of the original task in *T**. Constraint (3) forces the first part to execute before the second. Constraint (4) is redundant and prevents the first part to follow the second. In the situation where a task is executed in one part, the duration of the first part is the duration of the original task: $P_i = p_i$. In this case, we force the second part, with a null duration, to succeed to the first part (5). If a task executes in two parts, (6) forces the first part to complete at the end of the shift. In other words, only the end of a shift can justify not to fully execute a task. Constraint (7) ensures that the time left to execute in the second part is longer than minDuration.
The variable $\tilde{P}_i$ gets the duration to execute a task part i $\in$ T and to complete the setup after the task (8). These durations are passed to the DISJUNCTIVE constraint (9) to ensure that starting time variables S_i are set in a way that no two tasks nor setups overlap. The constraint WEIGHTEDCIRCUIT (10) takes as input the next variables N_i and the setup time matrix *t* and maps them to the variable, the sum of the transition times. In order for the next variables to form a cycle, we force the sentinel task *sentinel-begin* to follow the task *sentinel-end* (11). Constraints (12) and (13) force the execution of a task during a work shift. This encoding with two constraints allows a stronger filtering between the variables S_i and D_i.
A specificity of the production lines limits the number of tasks during a day (*maxTaskShift*). Tasks with a null processing time are not included in the total number of tasks. Constraint (14) sets the variable D_i' to the working day if the processing time of a task *i* is greater than 0 and -1 otherwise. Constraint (15) prevents any day (except -1) to occur more than *maxTaskShift* times in vector *D'*.
Setups longer than the parameter *maxSetup* during a working day are not permitted. Constraint (16) ensures that a transition beyond that threshold leads to a task that is performed another day. Exploiting the triangle inequality, Constraint (17) states the condition in which the transition time is simply too large to allow two tasks executing on the same day, even if these two tasks execute first and last on that day.
Tasks in T_M are either executed on a Monday or preceded by a task in T_M. The variable PR_i is the tasks that precedes *i*. Constraint (18) maps the predecessor variables to the next variables. Using the predecessor variable, Constraint (19) enforces that a task in T_M is either done a Monday or after a task in T_M.

4.3. Additional Constraints

The constraints that we presented so far fully encode the problem. However, constraint programming takes its efficiency from its filtering algorithm and it is sometimes necessary to add additional constraints in order to speed up the search process. These constraints are presented in this section. Note that these constraints might eliminate some solutions that are known to be suboptimal.
The solver constructs a solution by choosing variables and assigning them a value. We add to the model constraints whose filtering algorithm sets the value of the starting time variable S_{N_i} whenever the next variable N_i is affected to a value. Before explaining these constraints, we declare two variables. The variable TND_i (time of next day) is the time at which the upcoming shift starts. It is set by Constraint (20). The variable POS_i is the position of a task *i* in its shift.
The position of a task is determined by three cases. The first case occurs when a task *i* and its next task are executed on the same day and the next task has a non-null processing time. In this situation, the Constraint (21) is applied and the positions of the tasks are one apart. The second case occurs when a task *i* and its next task are executed on the same day, but the next task has a null processing time. In such a case, Constraint (22) makes both tasks have the same position. The third case occurs when a task and its next task occurs on different days. In that case, Constraint (23) resets the position of the next task to one. When the variable POS_i is given the value *maxTaskShift*, the following tasks must be executed on the next day. If this happens, Constraint (24) sets the variable M_i to the value of the next morning and 0 otherwise.

Now we have all what is needed in order to be able to set the starting time variable S_{N_i} when the next variable N_i is set. There are four possible cases. Constraint (25) handles the first case when the task is a first part. This constraint checks whether a task can be immediately executed after another task before the end of the day shift. If so, the task starts after the preceding task, the next morning if the day is full, or at its earliest starting time (whichever comes first). For a task with a processing time greater than or equal to twice *minDuration*, we need to have a minimum of *minDuration* time before the night to start it. For the task with duration less than twice *minDuration*, we need to have enough time to execute the task entirely to be in this condition.
The second case is applied only for the first part of a task and is handled by (26). This constraint manages the case when the next task must start on the upcoming morning (TND_i) or at its earliest starting time. Except for the fact that it is only applied to the first part of the task, this constraint manages the opposite case of the Constraint (25). If there is not enough time before the end of the day shift, the task must be done in the upcoming morning. If the processing time of the next task (p_{N_i}) is greater than twice the *minDuration* and there is less than *minDuration* time before the night, or if a task has less than twice *minDuration* and there is less than the processing time p_{N_i} before the end of the working day, this case is applied.
The third case is only applied to the second part of a task and is handled by (27). This constraint manages the case when a second part of a task must start directly after the previous task or start the next day if the current day is full. It happens when the first part executes completely, and the second part has a null processing time. The task must start directly after the first part. It should be noted that in that case, M_i will always be zero. It also happens there is enough time before the end of the working day to finish the task.
Finally, the last case is also applied to the second part of the task and is modelled with Constraint (28). It manages the case when there is not enough time to complete the task before the end of the working day. The task must therefore start in the beginning of the upcoming morning TND_i.

4.4. Search Strategy

Constraint Programming is an exact method that guarantees an optimal solution, given sufficient time. A branching heuristic can help speeding up the search process. We design search heuristic called *opportunity cost heuristic* that analyses the domains of the next variables N, the starting time variables S, and the transition matrix t, and that branches on the next variables N. We recall that filtering algorithms keep removing values from the domains during the search process and that the heuristic makes a choice according to the current domains rather than the initial domains stated in Table 3. The heuristic's principle relies on the preference to schedule now the tasks that will be harder to schedule later. Branching on the variables N is a strategic choice. If branching on $N_i = v_1$ leads to a failure, the solver branches on $N_i = v_2$ which is a significantly different solution. If the heuristic was branching on the starting time variables, say $S_i = 1$, upon a failure, the solver would try another value such as $S_i = 2$. However, starting the task i one minute later is not significantly different. Let i be the current task (initially the task *sentinel-begin)*. The heuristic finds which task v should be assigned to N_i. After the branching $N_i = v$ is performed, v becomes the new current task. The heuristic chooses the task v with the smallest value in the domain of its starting time S and breaks ties by selecting the task that is more difficult to schedule later. This is conducted based on these rules:

1. Choose v that contains the smallest value in dom(S_v). If v is unique, choose v, else go to Step 2.
2. For each task v computed in Step 1, calculate the sum of the transition times of a task not yet scheduled toward v. Go to Step 3.
3. Randomly choose the next task with probability proportional to the sums computed in Step 2.

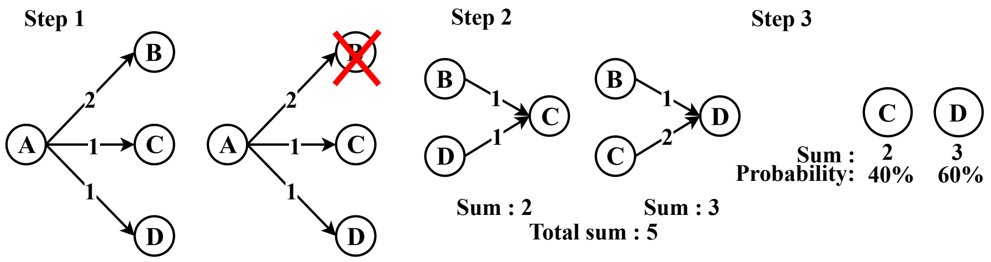

Figure 1: Example of the use of the opportunity cost heuristic

We conduct a search with restarts, i.e. after exploring a given number of nodes in the search tree, the solver restarts the search from the root node. Such a search strategy requires the heuristic to make some randomized decisions in order not to re-explore the same portion of the search, hence the randomization of Step 3. Figure 1 shows an example of the use of the heuristic. In the first step, task A is already scheduled, and we want to know what task should be executed afterwards. B is automatically eliminated because it can start one unit of time after the others. In step 2, the setup of tasks not yet scheduled towards tasks C and D is calculated. This result is useful in step 3 in order to make task D more likely to be chosen. Task D is therefore more difficult to schedule because it has a chance to end up after task C causing a setup of 2 units.

5. Experimentation

The experiments were carried out in collaboration with a cookie factory. During the development of the model, the company sent data that was used by their planners. Instances of the problem were solved with the model and the solutions were returned to the planners. They analysed them and gave feedback to improve the model. This has been repeated until the results were of good quality. For some of these instances, we had the actual schedules used by the planners which allowed us to make a comparison between the schedules made manually and those automatically generated by our model.
Branching heuristics: Prior to designing the *opportunity cost heuristic* described in Section 4.4, we tried standard branching heuristics available in most constraint solvers. We used *Smallest,* a heuristic that selects the starting time variable S_i with the smallest value in its domain and assign it to its smallest value. *Smallest* can be used in conjunction with two meta branching heuristics: Last Conflict [12] and Conflict Ordering [13]. These meta branching heuristics apply the *Smallest* branching heuristic, but as the search explores the search tree, the meta heuristics learn which variables are conflictual and start altering the original *Smallest* heuristics in order to give more importance to these variables. These meta branching heuristics were specially designed for scheduling problems. All these heuristics are already implemented in the solver Choco 4.0.6. We also implemented and integrated the *opportunity cost heuristic* to that solver.
Search strategy: When the exploration of the search reaches a dead end, i.e. when the choices made so far by the branching heuristics cannot lead to any solution, the search needs to reconsider the choices that were made. Generally, a solver reconsiders the last choice that was made until all choices are exhausted. It then reconsiders the previous last choice, and so on. This search strategy is called Depth First Search (DFS). Another search strategy called Limited Discrepancy Search (LDS) has been effective in many industrial cases [14]. This strategy explores the solution with zero reconsidered choice, then all solutions with exactly one reconsidered choice, then all solutions with exactly two reconsidered choices, and so on. LDS requires the branching heuristics to be deterministic, for that purpose, we replace Step 3 of the opportunity cost heuristic with a deterministic choice: we select the task with the largest sum computed at Step 2.

Table 1: Comparison between the planners and computed schedule

Instances	Human Planners	Smallest time	Last conflict	Conflict ordering	Opportunity cost heuristic	Opportunity cost heuristic with LDS
1	1665	1135	1175	1145	**1070**	**1070**
2	2965	2160	2160	2185	2085	**2045**
3	1645	1390	-	-	**1305**	-
4	1225	1260	1260	1290	**1110**	-
5	1655	1470	-	1555	1185	**1105**

The experiments were done on a MacBook Pro with a 2.6 GHz Intel Core i7 processor. We solve 5 industrial instances counting from 45 to 65 tasks. Table 1 reports the best objective value (cumulative setup time in minutes) obtained for each solving technique after a 15-minute cut-off as well as the objective value obtained by the human planners. Our solutions have shorter setup times than what the human planners obtained. The Last Conflict and Conflict Ordering fail to always return a solution within 15 minutes. With that respect, the opportunity cost heuristic is a better solution than Smallest, Last Conflict, and Conflict Ordering. Breaking the ties with the sum of the entering setups gives better results on the studied instances and a solution is found for each instance. LDS sometimes gives better results than any branching heuristics used with DFS. However, sometimes, it does not find a single solution which makes this heuristic unusable for the case study. The reason for this performance is that the number of side constraints can regularly leads to failures. Depth First Search (DFS) can quickly correct this error by making another branching, but LDS tries to explore all solutions with fewer reconsidered choices rather than quickly fixing the solution.

Overall, our model with the opportunity cost branching heuristic and a DFS search strategy offers the best performances. It makes it possible to find better solutions than the planners with a reduction of setup time of 26%. Only the use of LDS allows better solutions, but it could not solve all instances. The difference in the objective values between opportunity cost heuristic with DFS and with LDS was not significant.

6. Conclusion

In the food industry, setup times can affect the efficiency of a production line since it is not a value-added activity. In addition, planners must take into account several constraints such as the availability of ingredients and the production line. The goal of this work is to schedule a set of tasks on a single machine for the weeks 5 to 8 of the upcoming horizon. This one is done weekly using constraint programming. The model has been developed iteratively with the company.
The scientific contribution of this work is to use constraint programming to solve an industrial problem in scheduling that minimizes setup times. Indeed, there is very few papers that use this method to solve applied cases. Also, the model's originality is to have created a set of constraints that aim to help the heuristic in its branching. For the industrial contribution, we managed to solve complex cases by obtaining good results. We obtain an improvement of 26% on average compared to what was done with the planners.

7. Acknowledgement

We thank Vincent Gingras who laid the basis of the model in the early stage of this project.

8. References

1. Ku, W.Y., Christopher Beck, J.: Mixed Integer Programming Models for Job Shop Scheduling: A Computational Analysis. In : Computers & Operations 73, pp. 165--173 (2016)
2. Dejemeppe, C., Van Cauwelaert, S., Schaus, P.: The Unary Resource with Transition Times. In : Principles and Practice of Constraint Programming, pp. 89--104 (2015)
3. Claassen, G.D.H., Gerdessen, J.C., Hendrix, E.M. and van der Vorst, J.G.: On production planning and scheduling in food processing industry: Modelling non-triangular setups and product decay. In : Computers & Operations 76, pp. 147--154 (2016)
4. Grimes, D., Hebrard, E.: Job shop scheduling with setup times and maximal time-lags: A simple constraint programming approach. International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, pp. 147--161 (2010)
5. Fagerholt K, Christiansen M.: A travelling salesman problem with allocation, time window and precedence constraints - an application to ship scheduling. In: Int. Transactions in Operational Research, pp. 231-244 (2000)
6. Ángel-Bello F, Álvarez A, Pacheco J, Martínez I.: A single machine scheduling problem with availability constraints and sequence-dependent setup costs. In: Applied Mathematical Modelling 35, pp. 2041--2050 (2011)
7. Abdallah KS, Jang J.: Scheduling a single machine with job family setup times to minimize total tardiness. In: 2017 International Conference on Engineering, Technology and Innovation (ICE/ITMC), pp. 665--672 (2017)
8. Carlier J.: The one-machine sequencing problem. In: European Journal of Operational Research, pp. 42--47 (1982)
9. Benchimol P, Van Hoeve WJ, Régin JC, Rousseau LM, Rueher M. Improved filtering for weighted circuit constraints. In: Constraints 17, no. 3, pp. 205--233 (2012)
10. Oplobedu A, Marcovitch J, Tourbier Y.: CHARME: Un langage industriel de programmation par contraintes, illustré par une application chez Renault. In: Ninth International Workshop on Expert Systems and their Applications: General Conference, pp. 55--70 (1989)
11. Allahverdi A.: The third comprehensive survey on scheduling problems with setup times/costs. In: European Journal of Operational Research, pp. 345--378 (2015)
12. Lecoutre C, Saïs L, Tabary S, Vidal V.: Reasoning from last conflict(s) in constraint programming.In: Artificial Intelligence, pp. 1592--1614 (2009)
13. Gay S, Hartert R, Lecoutre C, Schaus P.: Conflict ordering search for scheduling problems. In: International conference on principles and practice of constraint programming, pp. 140--148 (2015)
14. Harvey WD, Ginsberg ML.: Limited discrepancy search. In: International Joint Conferences on Artificial Intelligence, pp. 607--615 (1995)
15. Focacci F, Lodi A, Milano M.: Cost-based domain filtering. In: International conference on principles and practice of constraint programming, pp. 189--203 (1999)