Forward-Checking Filtering for Nested Cardinality Constraints: Application to an Energy Cost-Aware Production Planning Problem for Tissue Manufacturing

Cyrille Dejemeppe[1], Olivier Devolder[2], Victor Lecomte[1], and Pierre Schaus[1]

[1]ICTEAM, UCLouvain, Belgium
[2]N-SIDE, Belgium
{cyrille.dejemeppe, pierre.schaus}@uclouvain.be,
victor.lecomte@student.uclouvain.be, ode@n-side.com

Abstract. Response to electricity price fluctuations becomes increasingly important for industries with high energy demands. Consumer tissue manufacturing (toilet paper, kitchen rolls, facial tissues) is such an industry. Its production process is flexible enough to leverage partial planning reorganization allowing to reduce electricity consumption. The idea is to shift the production of the tissues (rolls) requiring more energy when electricity prices (forecasts) are lower. As production plans are subject to many constraints, not every reorganization is possible. An important constraint is the order book that translates into hard production deadlines. A Constraint Programming (CP) model to enforce the due dates can be encoded with p Global Cardinality Constraints (GCC); one for each of the p prefixes of the production variable array. This decomposition into separate GCC's hinders propagation and should rather be modeled using the global `nested_gcc` constraint introduced by Zanarini and Pesant. Unfortunately it is well known that the GAC propagation does not always pay off in practice for cardinality constraints when compared to lighter Forward-Checking (FWC) algorithms. We introduce a preprocessing step to tighten the cardinality bounds of the GCC's potentially strengthening the pruning of the individual FWC filterings. We further improve the FWC propagation procedure with a global algorithm reducing the amortized computation cost to $\mathcal{O}(log(p))$ instead of $\mathcal{O}(p)$. We describe an energy cost-aware CP model for tissue manufacturing production planning including the `nested_gcc`. Our experiments on real historical data illustrates the scalability of the approach using a Large Neighborhood Search (LNS).

Introduction

The share of renewable energy production, such as wind or solar power is growing fast in several countries of the EU [19]. While the production of nuclear and fossil energy tends to be stable, renewable energy production is highly dependent of both climatic conditions and time of the day considered. Renewable resources add a huge variability on the offer and demand, and thus on the price of electricity. As an example, Figure 1 shows the historical electricity prices in Europe on March 3rd, 2014. In this example, the electricity prices fluctuate with a multiplicative factor higher than 3.5. Performing activities requiring more energy when electricity price is low represents both an economical and ecological advantage (the energy produced is not ”wasted”).

In [14], Simonis and Hadzic propose a cumulative constraint that links the energy consumption of activities with evolving electricity prices. We believe this kind of energy-aware optimization will become increasingly present in the industries with order-driven production planning that can be easily split into different steps. It generally offers enough flexibility to reduce the energy costs by scheduling tasks requiring more energy when the electricity price is lower. This paper addresses the problem of

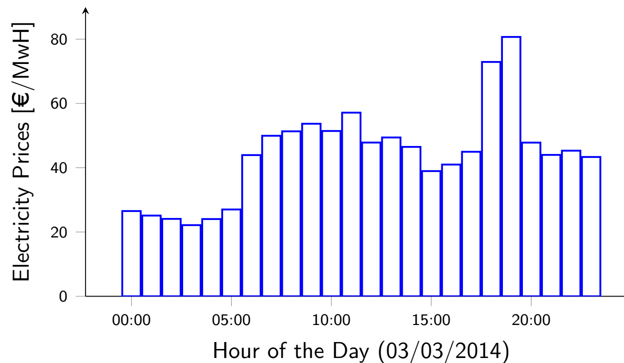


Fig. 1: Historical evolution of electricity prices on the EU market on March 3rd, 2014.

energy-efficient scheduling in consumer tissue production planning. Consumer tissue production planning offers several levers of flexibility, allowing to drastically reduce the energy costs for a given set of orders. Indeed, the paper machine receiving paper pulp as input and producing paper rolls consumes an amount of energy that depends on the tissue properties (quality, density of fibers, thickness, etc). Therefore, our CP model attempts to schedule the production of paper rolls requiring more energy when electricity price forecasts are lower. The order book limits the flexibility and is modeled using a `nested_gcc` [20]. A flow based GAC filtering for this constraint is proposed in [20]. This paper introduces a light filtering algorithm particularly well suited to tackle large instances in a Large Neighborhood Search (LNS) requiring fast restarts. A preprocessing step to tighten the initial cardinality bounds allows to obtain additional pruning compared to a naive decomposition with Forward Checking (FWC) GCCs. Furthermore, we propose a general refined FWC propagation procedure allowing to reduce its amortized time complexity from $\mathcal{O}(p)$ with the decomposition into multiple GCCs to $\mathcal{O}(\log(p))$.

In Section 1, we describe the consumer tissue manufacturing problem. Then, in Section 2, we propose a CP model to solve this industrial problem. Section 3 describes the preprocessing step to obtain tighter cardinality bounds as well as our own FWC propagation algorithm for the `nested_gcc`. Finally, Section 4 explains the results obtained on real historical data with our model.

1 Paper Production Planning

An important industrial site in Belgium manufactures hygienic paper (toilet paper and facial tissues are examples of refined paper they produce). Paper rolls are produced before being converted into different products (e.g. toilet papers or kitchen rolls). The production is a two step process: paper roll production, then conversion of paper rolls into final products. In Figure 2, we give a schematic overview of the different steps in the production of paper on the industrial site considered. The energy consumption

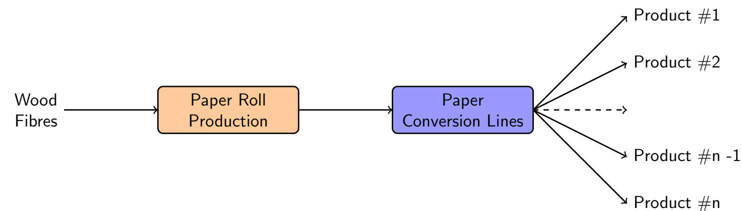


Fig. 2: Production steps in paper industry.

can vary up to 15% depending on the type of roll produced. Therefore the company is looking for the less expensive production planning given the electricity price forecasts.

The potential savings depends on the flexibility of the production site. For example, a factory continuously producing a same product does not have much potential to reduce its energy bill. On the contrary, a manufacturer producing many different products on a production line, each requiring a significantly different amount of energy has probably more flexibility to reduce its energy bill. The paper roll production can be split in two main successive steps: paper pulp production and transformation of paper pulp into paper rolls. The potential energy gain on the first step is negligible compared to the the second step. Indeed, as reported during our visits made on site, the pulp production part does not contain much flexibility and is significantly less energy-intensive than the paper machines producing paper rolls. Therefore this work focuses on the roll production part on the paper machine of the production line.

1.1 Paper Machine Scheduling

The paper machine transforms paper pulp into paper rolls. This consists in a continuous process in which the paper pulp is spread out on a treadmill passing through several presses and in front of a succession of heating devices or ventilation systems in order to dry the pulp and obtain a sheet of paper that will then be rolled up to form paper rolls.

As this process is continuous, the biggest factor that can impact the consumption of electricity is the kind of paper that has to go through the process. Indeed, depending on the type of paper pulp on the treadmill, the treadmill speed, the temperature of the heater, the speed of ventilation systems and other parameters will vary. The flexibility comes from the possible permutations of paper types according to electricity prices.

A new calibration (of the treadmill and the other components) is necessary for any change of paper type on the machine. This calibration is time consuming and the quality of the paper cannot be ensured during a transition between two different paper types. A minimal duration between any change of paper type is imposed in order to reduce their frequency. The duration for calibration and the loss of paper quality incurred depends on the transition of paper types. Some transitions are more desirable than others. A transition cost can thus be associated for every transition type (i.e. every pair of paper types that will be produced successively).

2 A Planning Model for Paper Roll Production

In this section, we describe a production planning model to represent the transformation of paper pulp into paper rolls. The constraints of this problem are:

– For every demand of paper rolls of a given type at a specified due date, a larger or equivalent amount of paper rolls of the same type has to be produced before the respective due date.
– When a paper type is produced, it has to be produced for a minimum duration before another paper type can be produced

The objective quantities should be optimized:

– The total energy cost of the production planning has to be minimized.
– The cost (and thus also the number) of transitions between different successive paper types has to be minimized.

A formal definition of the problem is given next. Item indices i, j are ranging on the set $\{1, \ldots, I\}$. Time index t is ranging over $\{1, \ldots, T\}$ where T is the horizon of the planning at an hour basis (since electricity price is changing every hour). The deadline indices are a subset of the time indices: $\{l_1, \ldots, l_L\} \subseteq \{1, \ldots, T\}$.

$$\text{minimize} \quad w_1 \sum_{i,t} (p_t \cdot c_i \cdot x_{t,i}) + w_2 \sum_{i,j,t} (s_{i,j} \cdot y_{i,j,t}) \tag{1}$$

$$\text{subject to} \quad y_{i,j,t} + 1 \geq x_{t,i} + x_{t+1,j} \qquad \forall i, j, t \tag{2}$$

$$\sum_{i} x_{t,i} = 1 \qquad \forall t \tag{3}$$

$$\text{lower}_l^i \leq \sum_{t=1}^{l} x_{t,i} \leq \text{upper}_l^i \qquad \forall i, l \tag{4}$$

$$\texttt{contiguous sequence length} \geq k \tag{5}$$

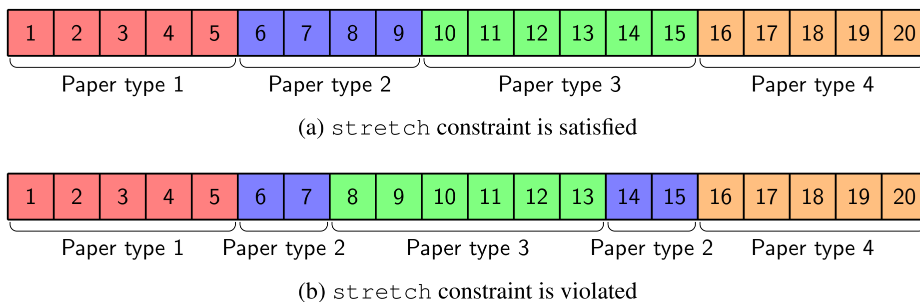
$$x_{t,i} \in \{0, 1\} \qquad \forall i, t \tag{6}$$

$$y_{i,j,t} \in \{0, 1\} \qquad \forall i, j, t \tag{7}$$

The variable $x_{t,i}$ is a binary variable equal to 1 if paper type i is produced at period t. The variable $y_{i,j,t}$ is true only if there is a transition from paper type i to paper

type j occurring at time t. Equation (1) is the objective function composed of two terms weighted by w_1 and w_2. The first term is the energy cost with p_t the price of electricity at time t and c_i is the energy consumption per period for paper type i. The second term is a penalty to pay for the transitions with $s_{i,j}$ the cost associated to the transition between paper type i and paper type j. Equation (2) ensures that $y_{i,j,t} = 1$ only if a product of type i is scheduled at time t and a product of type j at time $t+1$. Equation (3) ensures that only a single product is scheduled at any time. The constraints at Equation (4) enforce that the number of products of type i scheduled during the first $l \in \{l_1, \ldots, l_L\}$ periods is within the interval $[\text{lower}_l^i, \text{upper}_l^i]$. The constraint (5) is more difficult to express concisely in a mathematical form. It asks that contiguous sequences of successive variables of a same type should be of length at least k.

CP Model The problem described above is modeled into Constraint Programming (CP). For every hour t of the planning, a variable x_t with domain $\{1, \ldots, I\}$ is introduced: the paper type to be produced at the hour t. We can compute the electricity consumption c_{x_t} at time t with element constraints[1]. The electricity price to pay is then $\sum_t c_{x_t} \cdot p_t$. The transition cost $s_{x_t, x_{t+1}}$ at every time-point t is also computed with element constraints. The overall transition costs is $\sum_t s_{x_t, x_{t+1}}$. The order book constraint of Equation (4) can be enforced with a Global Cardinality Constraint (GCC) [11] at every deadline $l \in L$. However, the pruning of this formulation can be improved by using `nested_gcc` [20]. An efficient FWC algorithm for this constraint is introduced in Section 3. The constraint (5) asks that contiguous sequences of a same paper type should be at least of length k. This can easily be expressed in CP with a `stretch` [4] or a `regular` [9] constraint. In Figure 3a, we see a schedule where the constraint is satisfied for $k = 4$ (i.e. there is no succession of rectangles of the same color with length inferior to 4). On the other hand, Figure 3b shows an unfeasible schedule for the same set of paper types produced since there are two successions of 2 periods where the paper type is blue. The two objectives, minimization of electricity costs and minimization of transition costs between paper types, are aggregated in a sum that is minimized.



(a) `stretch` constraint is satisfied

(b) `stretch` constraint is violated

Fig. 3: Example of feasible and unfeasible schedule for `stretch` constraint.

[1] The element constraint [18] allows to access the value of an array where the index is a variable.

3 A Nested GCC Forward Consistent Propagator

The Global Cardinality Constraint (GCC) [11] on a vector of variables restricts the number of occurrences for each values to be within a specified interval. On our problem, the book order constrains the production on the first $l \in \{l_1, \dots, l_L\}$ variables to contain at least lower_l^i times the value i. As an example, let us consider a schedule with 20 periods. A first deadline could impose that we produce at least 4 paper rolls of type 1 for period 11 and at least 6 paper rolls of type 1 for period 18. A feasible schedule for this example is shown in Figure 4.

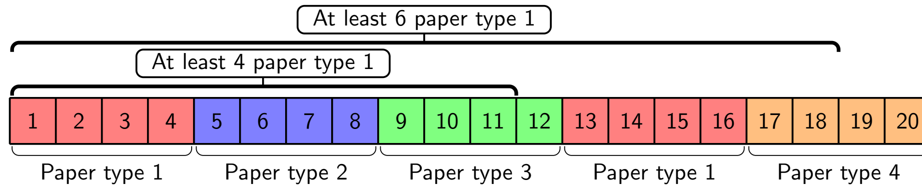


Fig. 4: Feasible schedules with nested GCC (2 deadlines on the paper type 1: at least 4 units at the end of period 11 and at least 6 units at the end of period 18).

Similarly, stock constraints impose a maximum number of times upper_l^i a value i can appear in the first l variables. As deadlines and stock constraints are nested on overlapping variable sets, we are in the special case of a GCC: the `nested_gcc` [20]. More formally

$$\texttt{nested_gcc}([x_1, \dots, x_n], [\text{lower}_{l_1}^1, \dots, \text{lower}_{l_L}^I], \dots, [\text{upper}_{l_1}^1, \dots, \text{upper}_{l_L}^I]$$

enforces the following constraints

$$\text{lower}_{l_k}^i \leq \sum_{t=1}^{l_k} (x_t = i) \leq \text{upper}_{l_k}^i \quad \forall i \in \{1, \dots, I\}, k \in \{1, \dots, L\}.$$

The nested GCC constraint can be expressed with a decomposition of several standard GCCs: one for every deadline l_k. However, this GCCs decomposition hinders propagation as shown in [20]. An example in which the decomposition misses pruning opportunity is displayed in Figure 5. In this example, there are already 4 variables set to value red in the range $[1, 16]$ constrained to contain at most 7 variables assigned to red. This imposes that the range before the first variable set to value red (range from 1 to 12) should contain at most 3 variables set to value red. This kind of unfeasible assignment would be detected by the flow-based GAC propagator of the `nested_gcc` from [20].

In practice, the strongest filtering algorithms are not always the winners on every problem. As explained in [15]: Maintaining a higher level of consistency takes more time; on the other hand, if more values can be removed from the domains of the variables, the search effort will be reduced and this will save time. Whether or not the time saved outweighs the time spent depends on the problem. In practice, many solvers (such

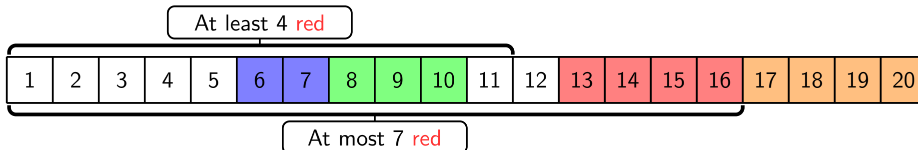


Fig. 5: Example of pruning missed by a classic GCC decomposition. The white cells represent unbounded variables while other colors represent value assignations.

as the very efficient OR-Tools [7]) use a default forward checking filtering (FWC) for the GCC. Our application problem contains large instances that will be solved with a Large Neighborhood Search (LNS). In an LNS setting, the strength of the filtering is also less important since the time spent at each node becomes the most critical to allow fast restarts and a good diversification. Our experience suggests that it is rarely the case that heavier propagation pays off when using LNS. Therefore we are interested to design an efficient forward checking propagation procedure for the `nested_gcc`.

In the following we design a FWC propagator achieving both a potentially stronger and faster pruning when compared to a naive decomposition of L FWC-GCCs. The improvement in pruning is obtained by a preprocessing step that strengthens the bounds of the cardinalities $\text{lower}^i_{l_k}$ and $\text{upper}^i_{l_k}$. The improvement in terms of running time is obtained by maintaining incremental counters avoiding the need to propagate every sub-GCC on each domain update. We present first the pre-computation step, then the FWC filtering procedure.

3.1 Bounds Pre-Computation

This step aims at tightening the bounds $\text{lower}^i_{l_k}$ and $\text{upper}^i_{l_k}$ specified by the user and minimizing the number of these to a minimal set. Two reasonings can be done:

1. between different ranges for the same value (e.g. the occurrences of red in range $[1, 4]$ and range $[1, 5]$).
2. between the bounds for the different values specified at a same date t (e.g. the occurrences of red versus blue in range $[1, 6]$).

Per-value deductions The following forward and backward deductions can be made:

– Lower bounds: if there are at least *two* red in range $[1, 4]$, then there are at least *two* red in range $[1, 5]$ (forward), and at least *one* red in range $[1, 3]$ (backward).
– Upper bounds: if there are at most *two* red in range $[1, 4]$, then there are at most *three* red in range $[1, 5]$ (forward), and at most *two* red in range $[1, 3]$ (backward).

We can make those deductions based on the quantities lower^i_t and upper^i_t containing respectively the best-known lower and upper bounds on the occurrences of i for range $[1, t]$. This is done by traversing these values for each range once forward and once backward. The forward update of these values is defined as follows, t increasing from 1 to $n - 1$:

$$\text{lower}^i_t = \max \begin{cases} \text{lower}^i_t \\ \text{lower}^i_{t-1} \end{cases} \qquad \text{upper}^i_t = \min \begin{cases} \text{upper}^i_t \\ \text{upper}^i_{t-1} + 1 \end{cases}$$

Similarly, the backward update is defined as follows, i decreasing from n to 2:

$$\text{lower}^i_t = \max \begin{cases} \text{lower}^i_t \\ \text{lower}^i_{t+1} - 1 \end{cases} \qquad \text{upper}^i_t = \min \begin{cases} \text{upper}^i_t \\ \text{upper}^i_{t+1} \end{cases}$$

Inter-value deductions Intuitively, there are two types:

– For a given time t and for some paper type i, if the value lower^i_t is large, then the production of other types of paper before t is limited.
– For a given time t and for some paper type i, if the value upper^i_t is small, then the production of other types of paper before t must be compensated.

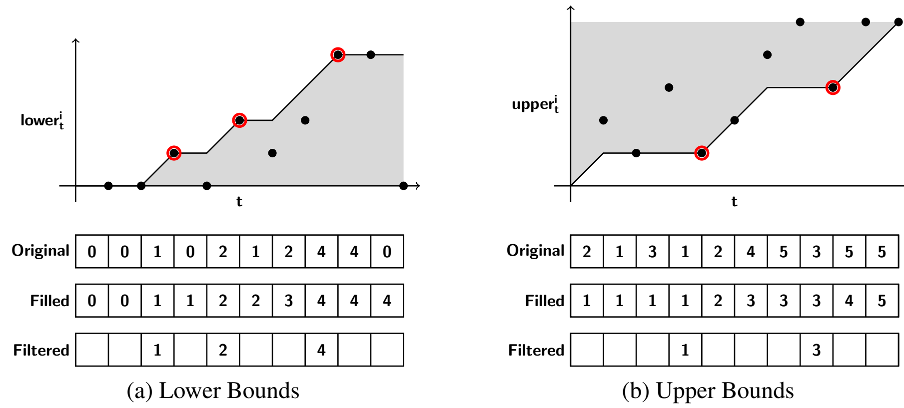
For example, for a period of length 5, if the sum of the deadlines for the other types is 3 ($\sum_{j \neq i} \text{lower}^j_5 = 3$), then at most 2 units of red (type 1) can be produced, and similarly if the sum of the storage space ($\sum_{j \neq i} \text{upper}^j_5 = 3$) for the other types is 3, then *at least* 2 units of red must be produced.

For every possible value i, and every possible index t, we define two quantities: lower^i_t and upper^i_t. These values are initially set to respectively, deadlines and stock constraints applying on range $[1, t]$ for value i (or respectively 0 and $n = t$ if not defined). We aim at setting these values with the best-known respectively lower and upper bounds on the number of occurrences of i on range $[1, t]$. For every value i and every index t defining range $[1, t]$, entries in arrays are defined as follows:

$$\text{lower}^i_t = \max \begin{cases} \text{lower}^i_t \\ t - \displaystyle\sum_{j \neq i} \text{upper}^j_t \end{cases} \qquad \text{upper}^i_t = \min \begin{cases} \text{upper}^i_t \\ t - \displaystyle\sum_{j \neq i} \text{lower}^j_t \end{cases}$$

Example An example of per-value pre-computation of lower bounds for a given value is shown in Figure 6a. Initial lower bounds in the gray zone are updated since dominated by the other specified bounds. The arrays displayed in this example represent the quantities lower^i_t at the different steps of the bound tightening. *Original* represents the original bounds specified by the user, *Filled* represents the bounds after application of the forward (left to right in the array) and backward (right to left in the array) updates described earlier.

After the tightening step of the bounds lower^i_t and upper^i_t, the number of these can be minimized to only keep the useful bounds in a decomposition of the `nested_gcc`. On the example of Figure 6a, the minimal set of useful bounds is indicated with a $\otimes$. Those are given in the *Filtered* array. A similar example to deduce the upper bounds for a given value is shown in Figure 6b. The pre-computation is done only once, at the initialization of the constraint. As such, the equalities defining lower^i_t and upper^i_t are assignment statements (not constraints). It can be shown that the final minimal set of bounds obtained after 1) the per-value deductions, 2) inter-value deduction and 3) minimization of the set of bounds, is the unique smallest set of bounds that contains all the useful information initially specified in the quantities lower^i_t and upper^i_t. Furthermore, the set of the times on which those final bounds apply is always a subset of the times at which a lower or upper bound was originally given.



(a) Lower Bounds

(b) Upper Bounds

Fig. 6: Example of deduction of best-known lower and upper bounds for a value.

3.2 Updating locally

With the improved bounds we have pre-computed in the previous step, we could very well use a standard FWC-GCC constraint for every range that is involved in the bounds, and obtain an improved pruning. However, if there are p such ranges, it would result in $\mathcal{O}(p)$ amortized time complexity per domain update. We present here a propagator that performs updates in $\mathcal{O}(\log(p))$ amortized time and offers the same pruning. As a reference point, the pruning given by forward-checking GCCs is such that

- when the number of variables whose domain still contains a given value decreases to the lower bound associated to it, these variables are assigned to the value.
- when the number of variables bound to a given value increases to the upper bound associated to it, this value is removed from all other variables.

The main challenge of this algorithm is to avoid checking those variable counts on every lower bound or upper bound when an update is received. In order to do that, for every value that we track and for both lower and upper bounds, we divide the variables into the segments that are formed by the bounds, and only count variables inside those segments. For example, if we have a maximum of 2 red in range $[1, 3]$ and a maximum of 5 red in range $[1, 8]$, we will separate the variables into the segments $[1, 3]$ (the first 3 variables) and $[4, 8]$ (the next 5 variables). We justify in the next paragraphs why local checks inside those segments are enough to detect and trigger the required pruning. This example is shown in Figure 7.

Let us examine the differences between the bounds in our example: $5 - 2 = 3$. We will call this difference the *critical point* of the segment $[3, 7]$. If the number of variables bound to red in that segment reaches 3, then there will be at least 3 occurrences of red in that segment. As a consequence, if the pruning condition in range $[1, 3]$ is met, so that we have 2 variables bound to red in $[1, 3]$, then in total there will be at least 5 variables bound to red in the range $[1, 8]$, so we have to prune there as well. In other words, pruning in $[1, 3]$ can only happen if pruning in $[1, 8]$ also happens; and since in both

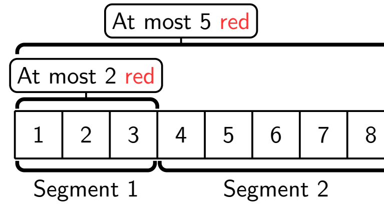


Fig. 7: Example of segment decomposition.

cases pruning means removing the value red from all unbound variables, it becomes useless to track the upper bound on $[1, 3]$.

Conversely, if there are *less than* 3 variables bound to red in the segment $[4, 8]$, then pruning for the upper bound of range $[1, 8]$ will happen strictly after pruning happens in $[1, 3]$ (if ever). Indeed, pruning in $[1, 3]$ happens when 2 variables in that segment are bound to red, and at that point less than 5 variables would be bound to red in $[1, 8]$.

For the leftmost segment, since there is no bound on the left, we simply define the critical point as the bound on the right, in this case 2 for segment $[1, 3]$. In this segment, reaching the critical point by having 2 variables bound to red means reaching a pruning cases, so we have to remove the red value from the last variable. If the number of variables bound to red is strictly under the critical point, however, no pruning can be performed.

From these remarks we can notice that no pruning will happen in a segment until it reaches its critical point. All that is left is to precisely determine what to do when it is reached. Note that we have taken upper bounds as an example, but the critical point also makes sense for lower bounds: instead of counting the number of variables bound to the value, count the number of variables that have the value in their domain.

We can also observe a useful property of critical points: if we combine two consecutive segments, the distance to the critical point in the merged segment will be the sum of the distances to the critical points in the small segments. Indeed, when summing the critical points, the middle bound will cancel itself out; and the number of variables that are bound to a value or that have a value in their domain is clearly the sum of those numbers in the segments that are being merged.

3.3 Pruning cases and segment merging

Let us now develop an updating strategy based on critical points. We split the variables into contiguous disjoint segments as described above. In the leftmost segment, pruning can happen only when its corresponding critical point has been reached. For other segments, if their respective critical point have not been reached, then no pruning can occur before some pruning happens on the left bound. When the critical point of a segment is reached, we can consider two different actions to perform, depending on whether the considered segment is the leftmost one or not.

First, if the segment is the leftmost segment, we have to trigger pruning in it. As none of the segment on its right has reached its critical point, no pruning should occur

on those. Once the pruning has been applied to the leftmost segment, it is removed and its neighboring right segment, if it exists, is marked as the leftmost segment. To achieve fast pruning, we propose to maintain a list of unbound variables still containing a particular value in an array based reversible double linked list. This allows value removal in constant time (as there is one list per possible value). We refer to this list as the *unbound list*. When a critical point is reached, the pruning on a segment will only be applied on variables in the unbound list.

Second, if the segment is *not* the leftmost segment, then reaching the critical point makes the bound on the left of the segment completely redundant in terms of pruning with the bound on the right of the segment. Therefore, the bound on the left can be forgotten, and this segment can be merged with its left neighboring segment. Since distance to the critical point is additive, the larger segment will not have reached its critical point either. To keep the propagator efficient in terms of time complexity, we have to determine efficiently to which segment a variable belongs. We also have to determine an efficient way to merge segments. This problem can be solved easily using a union-find data structure [16].

Here is a description of the steps to perform when a variable x has been bound to a value v and it is inside an upper bound segment:

1. Remove the variable from the unbound list of v.
2. Find the segment containing the variable (*find* operation in our union-find data-structure).
3. Increase the counter of assigned variables bound to v in the segment.
4. If the critical point of the segment has been reached and the segment is the leftmost segment, remove v from all the variables in the segment. Then, if there exists a right neighbor segment, define it as the leftmost segment.
5. If the critical point of the segment has been reached and the segment is not the leftmost segment, merge the segment with its left neighbor segment (*union* operation in our union-find data-structure).

Similarly, the following steps are performed when a value v has been removed from a variable x and it is inside a lower bound segment:

1. Remove the variable from the unbound list of v.
2. Find the segment containing the variable (*find* operation in our union-find data-structure).
3. Decrease the counter of variables which domain contains v in the segment.
4. If the counter has reached the critical point of the segment and it is the leftmost segment, assign v to all the unbound variables in the segment. Then, if there exists a right neighbor segment, define it as the leftmost segment.
5. If the counter has reached the critical point of the segment and it is not the leftmost segment, merge the segment with its left neighbor segment (*union* operation in our union-find data-structure).

3.4 Complexity

The complexity analysis assumes one has access to the Δ change of the variables as for instance proposed in [1] for the OscaR solver also available in OR-Tools [7], or the advisors of Gecode [6].

Let us define u as the number of updates, that is, the sum of the number of value removals over the whole search. Note that when the constraint itself removes a value from a variable, it counts in u as well. We will also use n, the number of variables, and p, which as earlier is the number of distinct ranges involved in the bounds. Looking at the steps performed when a value has been removed/assigned, we can deduce the time complexity for a particular update. Note that even though step 4 can take $\mathcal{O}(n)$ for one particular update to be processed, the variables pruned also count as updates, so it remains amortized constant time per update.

When we combine all of this, we discover that the total complexity is the number of updates multiplied by the cost of a union-find operation. One would think that would give $\mathcal{O}(u \cdot \alpha(p))$ since there will be at most p segments in each union-find structure. However, as this is implemented in a CP framework, we are working with a reversible union-find structure. As such, a particular update could be repeated arbitrarily many times in different places in the search tree. This means we cannot use the amortized $\mathcal{O}(\alpha(p))$ complexity of union-find operations, but rather the $\mathcal{O}(\log(p))$ worst case. As a result, we obtain a time complexity in $\mathcal{O}(u \log(p))$ for the whole search, or an amortized complexity of $\mathcal{O}(\log(p))$ per update.

4 Results

We experiment the CP model on historical data from a tissue manufacturing site in Belgium. This historical data contains the amount and type of paper rolls produced from paper pulp over a couple of years. The historical electricity prices on the EU market over the same period are also available. Combining those two sources of data, we were able to produce instances as follows:

1. Randomly select two dates separated from a specified amount of days. This defines the time window tw representing the instance.
2. Collect over tw the historical type of paper roll produced every hour.
3. Collect over tw the historical European electricity prices every hour.
4. Collect over tw, for every paper type i the contiguous periods at which i is produced. Let $[t_1, t_2]$ be such an interval where product type i is produced continuously. A deadline is imposed to produce additionally at least $t_2 - t_1 + 1$ items for date $t_2 + \delta$.

The shifting of deadlines by δ gives some flexibility to the model for optimization. As we don't have the historical stock constraints, we only impose over the whole time window tw to produce the exact same type and numbers of rolls. We have generated 4 sets of 10 instances for planning of respectively 4, 6, 8 and 11 days (96, 144, 192 and 264 time periods).

In order to evaluate the efficiency of the new FWC procedure for `nested_gcc`, we compare several models including different propagation procedures. All these models are based on the one described in Section 2 and only differ by the propagation procedure for the `nested_gcc` constraints. We propose to compare three forward checking propagation procedures:

GCC-FWC A decomposition of classic FWC-GCCs; one FWC-GCC for every range $[1, t]$ on which deadlines and stock constraints occur.

PreGCC-FWC After a pre-computation of optimal bounds (as described in Section 3), a decomposition of classic FWC-GCCs; one FWC-GCC for every range $[1, t]$ on which optimal bounds occur.

NestedGCC-FWC After a pre-computation of optimal bounds, the new FWC propagator described in Section 3.

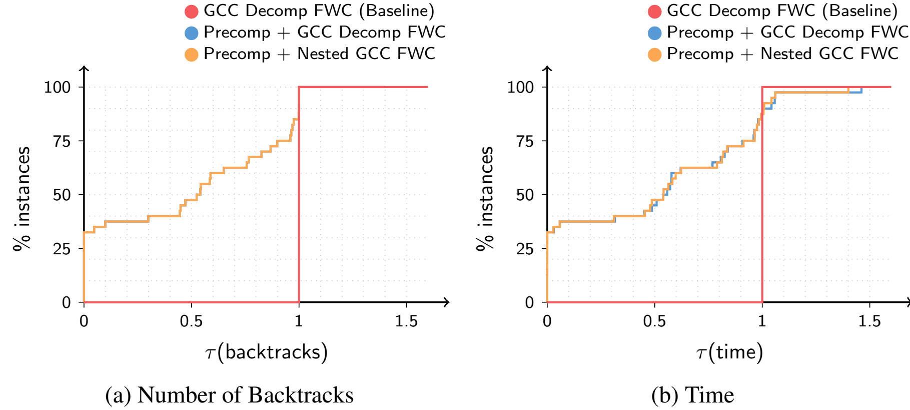
These models and propagation procedures have all been implemented in the open-source solver OscaR [8]. The propagation procedures are compared with *performance profiles* as described in [17] to compare filtering algorithms using GCC-FWC as baseline. Our measures are obtained by replaying a search tree generated with the baseline approach. Performance profiles [2] are cumulative distribution functions of a performance metric τ. In this paper, τ is the ratio between the solution time (or number of backtracks) of a target approach (i.e. PreGCC-FWC or NestedGCC-FWC) and the one the baseline (i.e. GCC-FWC). For the resolution time metric, the function is defined as:

$$F_{\phi_i}(\tau) = \frac{1}{|\mathcal{M}|} \left| \left\{ M \in \mathcal{M} : \frac{t(replay(\text{st}), M_i \cup \phi_i)}{t(replay(\text{st}), M)} \leq \tau \right\} \right|$$

where $\mathcal{M}$ is the set of considered instances while $t(replay(\text{st}), M \cup \phi_i)$ and $t(replay(\text{st}), M)$ are the time (backtracks) required to replay the generated search tree respectively with our different models and the baseline. The function for the number of backtracks is similar. For this paper, the original search trees have been generated with the baseline model using a binary first-fail heuristic.

Figures 8a and 8b respectively provide the profiles for number of backtracks and resolution times for all 40 instances. In Figure 8a, we can see that both approaches using the pre-computation step have a much smaller number of backtracks. Note that, as expected, once the new bounds have been computed, both PreGCC-FWC and NestedGCC-FWC offer the same pruning. We can also see that for about 35% of the instances, these propagators were able to almost completely cut the search tree explored by GCC-FWC. Finally, we can observe that there are only a bit less than 15% of the instances for which the propagators using pre-computed bounds are not able to achieve more pruning than GCC-FWC.

In Figure 8b, we can see the profiles of resolution times for the different propagators. We can see that both PreGCC-FWC and NestedGCC-FWC are faster than GCC-FWC for about 90% of the instances. The reason is the stronger filtering that is induced by the bounds-strengthening procedure. The 10% of instances for which both these variants are slower than GCC-FWC are those on which they offer no additional pruning; and even in this case, they are at worst less than 1.5 time slower than GCC-FWC. We can see however that resolution times are similar for PreGCC-FWC and NestedGCC-FWC. After profiling the application, we have seen that the GCC constraints only take a small fraction of the resolution time on this problem (less than 2%). Also the density of the number deadlines is not very large. This problem is thus not a good candidate to observe speedups with the more advanced FWC algorithm. We have tested artificial problems (not reported for space reason) with a larger number of deadlines. We observed a speedup between 2 to 3 times for the PreGCC-FWC .



(a) Number of Backtracks

(b) Time

Fig. 8: Performance profiles of `nested_gcc` variants

Energy consumption minimization with LNS

This section aims at showing the potential improvement brought by our model over historical production plans. An LNS is used with our CP model from Section 2 over the historical data and we compare the reductions in terms of cost. The search strategy used is Conflict Ordering Search [3]. The LNS setting is the following: at each iteration, we select 80% of possible values (e.g. paper types). Variables associated to these 80% values are then relaxed. This is done to relax the production plan except some *blocks of production* over some specific paper types. The search is stopped if one of these two conditions is met:

1. 180 seconds have elapsed since the beginning of the restart.
2. 200 relaxations have been performed (with a maximum of 1000 backtracks).

Table 1 shows the ratio of objectives (initial/optimized value) obtained. We can see that the cost of transitions is on average significantly reduced. However, the variance over this objective ratio is high: the reduction of transition cost is really important on some instances but it decreases less on other instances. The ratio of the energy cost however has a small variance. On most of the instances, LNS is able to reduce energy costs by around 22.5%. These results are promising but somewhat optimistic since it relies on a perfect knowledge of electricity future prices. Since forecasts can by definition be wrong, the gain could be reduced in practice.

	Global	Energy	Transition
Average	6.40	1.29	56.14
Variance	69.46	0.10	84,211.22

Table 1: Ratio of historical and optimized objective values (historical/optimized).

Future work

It would be interesting to test the benefits of the bound tightening for a decomposition of nested_gcc with Bound Consistent GCC [10]. As future work we plan to use variable objective large neighborhood search [12] to obtain a better pruning from our two terms composing the objective or to compute a Pareto front using a multi-objective large neighborhood search [13]. The CP model could also be extended with stocking costs computations [5] since it is not desirable to produce too early before the deadlines. We also plan to couple the paper machine scheduling problem studied in this paper with the batch scheduling problem happening just before in the production process. This would allow an integrated optimization of the whole production. Finally we would like to test the electricity price forecasts of the Enertop module of N-SIDE[2] to obtain a better estimate of the real energy gain. It was not possible to do it in this work. It would require to feed the forecast module with external features (weather forecast, etc.) that we don't have for the historical data.

Conclusion

In this paper we described the problem of reducing energy costs in paper tissue production. To tackle this problem, we propose to reorganize a large part of the manufacturing process: the production of paper rolls from paper pulp. According to forecasts of electricity prices, paper rolls whose production require a larger amount of energy will be produced when prices are low. On the opposite, paper rolls whose production require a smaller amount of energy will be produced when prices are high. The problem is subject to many constraints; an important one is the order book that translates into hard production deadlines. To represent the problem, we propose a CP model including all the constraints. This model will be linked with other CP models corresponding to other steps of the production workflow. The deadline and stock constraints of the problem are expressed with nested_gccs. As the model will be solved with an LNS framework, it has to be scalable. We propose a new FWC propagation procedure for the nested_gcc. This new propagation procedure comports two main step. First, an optimal and minimal set of bounds is computed. This new set of bounds allow us to achieve additional pruning that wouldn't be achieved with initial bounds. Then, we propose a global FWC propagation procedure based on these bounds which has an amortized time complexity in $\mathcal{O}(\log(p))$ (where p is the number of ranges considered). The performances of our new propagation procedure was evaluated on instances generated from historical data. The preprocessing step tightening the cardinality bounds brought significant pruning for many instances.

[2] http://energy.n-side.com/enertop-energy-flexibility-optimization/

References

1. V. l. C. de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre. Sparse-sets for domain implementation. In *Techniques for implementing constraint programming systems (TRICS) workshop at CP 2013*, 2013.
2. E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.
3. S. Gay, R. Hartert, C. Lecoutre, and P. Schaus. Conflict ordering search for scheduling problems. In *Principles and Practice of Constraint Programming*, pages 140–148. Springer, 2015.
4. L. Hellsten, G. Pesant, and P. Van Beek. A domain consistency algorithm for the stretch constraint. In *Principles and Practice of Constraint Programming–CP 2004*, pages 290–304. Springer, 2004.
5. V. R. Houndji, P. Schaus, L. Wolsey, and Y. Deville. The stockingcost constraint. In *Principles and Practice of Constraint Programming*, pages 382–397. Springer, 2014.
6. M. Z. Lagerkvist and C. Schulte. Advisors for incremental propagation. In *Principles and Practice of Constraint Programming–CP 2007*, pages 409–422. Springer, 2007.
7. OR-Tools Team, Laurent Perron. OR-TOOLS, 2010. Available from `https://developers.google.com/optimization/`.
8. OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.
9. G. Pesant. A regular language membership constraint for finite sequences of variables. In *Principles and Practice of Constraint Programming–CP 2004*, pages 482–495. Springer, 2004.
10. C.-G. Quimper, P. Van Beek, A. López-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Principles and Practice of Constraint Programming–CP 2003*, pages 600–614. Springer, 2003.
11. J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, AAAI'96, pages 209–215. AAAI Press, 1996.
12. P. Schaus. Variable objective large neighborhood search: A practical approach to solve over-constrained problems. In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, pages 971–978. IEEE, 2013.
13. P. Schaus and R. Hartert. Multi-objective large neighborhood search. In *Principles and Practice of Constraint Programming*, pages 611–627. Springer, 2013.
14. H. Simonis and T. Hadzic. A resource cost aware cumulative. In *Recent Advances in Constraints*, pages 76–89. Springer, 2011.
15. B. M. Smith. Modelling for constraint programming. *Lecture Notes for the First International Summer School on Constraint Programming*, 2005.
16. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.
17. S. Van Cauwelaert, M. Lombardi, and P. Schaus. Understanding the potential of propagators. In *Integration of AI and OR Techniques in Constraint Programming*, pages 427–436. Springer, 2015.
18. P. Van Hentenryck and J.-P. Carillon. Generality versus specificity: An experience with ai and or techniques. In *AAAI*, pages 660–664, 1988.
19. R. Wstenhagen and M. Bilharz. Green energy market development in germany: effective public policy and emerging customer demand. *Energy Policy*, 34(13):1681 – 1696, 2006.
20. A. Zanarini and G. Pesant. Generalizations of the global cardinality constraint for hierarchical resources. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 361–375. Springer, 2007.