Noname manuscript No.
(will be inserted by the editor)

Efficient Filtering for the Resource-Cost AllDifferent Constraint

Sascha Van Cauwelaert · Pierre Schaus

Received: date / Accepted: date

Abstract This paper studies a family of optimization problems where a set of items, each requiring a possibly different amount of resource, must be assigned to different slots for which the price of the resource can vary. The objective is then to assign items such that the overall resource cost is minimized. Such problems arise commonly in domains such as production scheduling in the presence of fluctuating renewable energy costs or variants of the Travelling Salesman Problem. In Constraint Programming, this can be naturally modeled in two ways: (a) with a sum of ELEMENT constraints; (b) with a MINIMUMASSIGNMENT constraint. Unfortunately the sum of ELEMENT constraints obtains a weak filtering and the MINIMUMASSIGNMENT constraint does not scale well on large instances. This work proposes a third approach by introducing the RESOURCECOSTALLDIFFERENT constraint and an associated incremental and scalable filtering algorithm, running in $\mathcal{O}(n \cdot m)$, where n is the number of unbound variables and m is the maximum domain size of unbound variables. Its goal is to compute the total cost in a scalable manner by dealing with the fact that all assignments must be different. We first evaluate the efficiency of the new filtering on a real industrial problem and then on the Product Matrix Travelling Salesman Problem, a special case of the Asymmetric Travelling Salesman Problem. The study shows experimentally that our approach generally outperforms the decomposition and the MINIMUMASSIGNMENT ones for the problems we considered.

Keywords AllDifferent · Assignment Cost · Filtering · Scalability · Scheduling · Energy · Resource · Product Matrix Travelling Salesman Problem

Sascha Van Cauwelaert
Place Sainte Barbe 2 bte L5.02.01 1348 Louvain-la-Neuve
E-mail: sascha.vancauwelaert@uclouvain.be

Pierre Schaus
Place Sainte Barbe 2 bte L5.02.01 1348 Louvain-la-Neuve
E-mail: pierre.schaus@uclouvain.be

1 Introduction

In this work, we consider a family of optimization problems where a set of items, each requiring a possibly different amount of resource, must be assigned to different slots for which the price of the resource can vary. In particular, we first consider the domain of production scheduling in the presence of fluctuating renewable energy costs. Indeed, some countries tend to decrease the use of fossil and nuclear energies and to replace them with renewable energies [33]. For instance in Europe, Germany has started a nuclear phase-out that should be completed by 2022. A consequence of the adoption of renewable energy is the larger fluctuation of energy prices. At the same time, the Electricity Price Forecast (EPF) at the daily base becomes more and more accurate [32]. This high volatility of the prices combined with EPF tools opens new perspectives and challenges for production scheduling optimization. A producer can get a competitive advantage if he is able to schedule his most energy consuming processes when the prices are low. The production planning should therefore leverage as much as possible the flexibility in the production process to translate it into energy cost savings.

In the setting we consider, the energy cost is assumed to be known at each future time slot (provided by an EPF module). The goal is then to schedule each item over the time slots such that the overall energy bill is minimized. For each item, its contribution to the cost is the energy required to produce it multiplied by the energy cost forecast at the time slot it is produced.

Example 1. Let us illustrate this situation with the example given in Figure 1. 3 items a, b and c have to be produced and their consumption are $C(a) = 2$, $C(b) = 4$ and $C(c) = 3$ (see left plot). There are 9 time slots where these items can be produced, each one being associated with an energy price $P(s)$. For instance, the slot 3 has a price of $P(3) = 15$. In the given schedule, a, b and c are produced at slot 1, 3 and 5, respectively. Hence, the total cost is $T = 2 \cdot 20 + 4 \cdot 15 + 3 \cdot 15 = 145$.

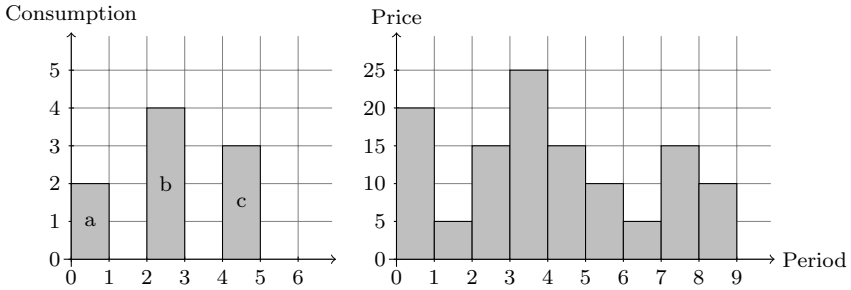

Fig. 1 Energy Consumption of items (left) and energy prices (right).

For a given set of items $\mathcal{I}$, we can formally define the total cost as:

$$T = \sum_{i=1}^{|\mathcal{I}|} C(\mathcal{I}_i) \cdot P(s(\mathcal{I}_i)) \tag{1}$$

where $s(\mathcal{I}_i)$ is the time slot assigned to the item $\mathcal{I}_i$.

In Constraint Programming (CP), one can model this cost in two ways:

1. with a sum of ELEMENT [27] constraints.
2. with a MINIMUMASSIGNMENT constraint that has a filtering based on reduced costs (referred to ILCALLDIFFCOST in [9] and MINWEIGHTALLDIFF in [22]). In this case, the cost of an edge linking a variable (i.e., an item) to a value (i.e., a time slot) is the product of the item consumption with the energy price of the time slot.

Unfortunately, both approaches have limitations. Modeling with a sum of ELEMENT constraints does not take into account the fact that the items must all be assigned to different time slots. For instance, in Figure 1, if all items could be assigned to any time slot, this approach would consider that all items can be assigned to the slots 2 or 7. The computed minimum cost would therefore be $5 \cdot (2 + 4 + 3) = 45$, which is clearly impossible since only 2 items can be produced at an energy price of 5.

The MINIMUMASSIGNMENT incorporates the all-different constraint, but the algorithm has a quite high time complexity of $\mathcal{O}(n^3)$, where n is the number of items[1].

Contribution This paper proposes a third approach by introducing the RESOURCECOSTALLDIFFERENT constraint and an associated scalable filtering algorithm. In the particular case where the total cost is computed with Equation 1, the constraint fills a gap between using a sum of individual ELEMENT constraints and solving the general matching problem with cost computed by MINIMUMASSIGNMENT. Its goal is to compute the total cost in a scalable and incremental manner, while considering the fact that all assignments must be different. Furthermore, efficient domain filtering can be performed in $\mathcal{O}(n \cdot m)$, where n is the number of unbound variables and m is the maximum domain size of unbound variables. While this is a better time complexity than $\mathcal{O}(n^3)$ required by MINIMUMASSIGNMENT, it comes at the price of a weaker inference. However, this trade-off pays off in practice for the two problems we considered, namely the *Continuous Casting Steel Production with Electricity Bill Minimization* and the *Product Matrix Travelling Salesman Problem*. The latter problem illustrates that the RESOURCECOSTALLDIFFERENT constraint can be used in other domains than production scheduling.

The results on the first problem demonstrate that MINIMUMASSIGNMENT is always slower than our approach. In addition, RESOURCECOSTALLDIFFERENT is often faster than the decomposition, sometimes with an important

[1] Although in practice the computation in a search tree is incremental.

speed-up. This is especially true for large instances, for which an order of magnitude is gained in $\sim 20\%$ of the cases, while $\sim 75\%$ of them are solved faster. Moreover, our algorithm is robust, in the sense that when it is slower, it is by a small factor (3.2 at the very most). Results also illustrate that for a non-negligible number of the large instances, we can get an important gain in terms of number of backtracks as compared with a decomposition: a reduction of at least one order of magnitude is obtained for 30% of the instances.

For the second problem, the results show that our approach is the fastest in CP. Moreover, it outperforms the Concorde solver[2,3], a custom state-of-the-art Branch-and-Cut Mixed Integer Programming solver, for 90% of the instances, sometimes by an important factor (some instances could not be solved in a factor 32 as compared with our constraint).

Related Work The need to compute the total cost related to some assignments in CP is not rare. It is for instance used to solve the Travelling Salesman Problem and the Travelling Salesman Problem with Time Windows. Focacci et al. proposed a global optimization constraint [9,10] (they call it IlcAllDiffCost) to compute a lower bound on the total cost when all assignments must be different and to filter the domains based on reduced costs. However, they use a linear formulation to compute the reduced costs, and therefore do not obtain arc-consistency. In [22], Sellmann calls this constraint MinWeightAllDiff and proposes an arc-consistent filtering algorithm in which exact reduced costs are used. More recently [8], Ducomman et al. came up with a different computation of the exact reduced costs, by making use of an all pairs of shortest paths algorithm. In this paper, we denote by MinimumAssignment these equivalent constraints and use the algorithm from [10] in particular for our experiments.[2] Finally, Régin introduced the more general GCCCost global constraint [20] for which several variables can be assigned to the same value as long as cardinality constraints are respected.

For all those constraints, there is no assumptions on the costs associated to the different variable-assignment pairs. The ResourceCostAllDifferent constraint we introduce is actually a particular case of a MinimumAssignment constraint for which the cost of assigning a variable X_i to a value s amounts to the product of a given *consumption* of X_i with the fixed *price* of assigning the value s to a variable. This particularity allows getting a more efficient filtering, as we shall see in the results.

For Scheduling problems, global constraints that consider (electricity) costs have been defined recently [23,24]. The particular case of the disjunctive-cost constraint was then further studied when activities have variable durations [30]. Finally, a recent application of CP was successfully used to optimize a tissue manufacturing planning problem from an energy viewpoint [6].

[2] We also experimented with the version from [8] to get exact reduced costs, but it appeared to be very slow for the instances we considered, so we do not report any results regarding that implementation.

Paper Outline We first formally define the constraint and provide related terminologies. We then describe an algorithm that checks feasibility, based on the computation of a lower bound for the total cost. Next, we present our filtering algorithm to prune unfeasible values of all variables. Before we conclude, we evaluate our work on two problems: the *Continuous Casting Steel Production with Electricity Bill Minimization* that is an industrial use case, and the *Product Matrix Travelling Salesman Problem* that is more academic.

2 Constraint Definition

Notation We write $D(X_i)$ to denote the domain of a variable X_i of a sequence of variables X. The minimum and maximum of the domain of a variable X_i are written $\underline{X_i}$ and $\overline{X_i}$, respectively.

Definition 1. *Let*

- *X be a sequence of integer variables that are the production time slots for each item to produce,*
- *C be a sequence of integer constants of length $|X|$ that are the amount of resource required to produce each item,*
- *H be an integer constant, that is the number of time slots (horizon),*
- *P be a sequence of H integer constants giving the price of the resource at each time slot, and*
- *T be an integer variable that is the total resource cost of the scheduling,*

the constraint ResourceCostAllDifferent*(X, C, P, T) ensures that*

$$\wedge \begin{cases} \text{allDifferent}(X) \\ T = \sum_{i=1}^{|X|} C(X_i) \cdot P(X_i) \end{cases} \tag{2}$$

with $D(X_i) \subseteq [0..H[, \forall X_i \in X$.

Intuitively, all variables X_i must be assigned to a different value. Moreover, they all have a fixed consumption given by $C(X_i)$. Any variable X_i assigned to a value s then implies a resource price of $P(s)$ multiplied by the consumption $C(X_i)$. The total cost T amounts to the sum, over all the variables, of the product of their consumption with the price of their assignment.

2.1 Lower Bound Computation and Feasibility Check

The constraint can be separated in two parts: all assignments must be different, and the assignments must respect the total cost constraint. For the first part, we rely on well-known propagators (see [28,19]) devised for the allDifferent constraint. For the second part, we propose to compute a lower bound for the minimum total cost T, written T_{lb}. In the following, for a given sequence of variables X and a sequence of assignments A, we denote the total production cost of mapping the i^{th} element of X to the i^{th} element of A, by $Prod_{cost}(X, A) = \sum_{i=1}^{\min(|X|,|A|)} C(X_i) \cdot P(A_i)$.

Lower-Bound for the cost There is an inherent matching problem involved in the filtering of allDifferent, as the domains of the variables can be different. To compute our lower bound efficiently, we relax the domain of the variables by assuming that all variables can be assigned to any value of any current domain, that is, any $s \in \bigcup_{X_i \in X} D(X_i)$. But we do not relax the all-different constraint. The matching problem can then be solved greedily on the relaxed problem.

Example 2. Let us reconsider the example of Figure 1, and let us assume there are 6 more items d to i to be scheduled for production, with a consumption of $C(d) = 2$, $C(e) = 3$, $C(f) = 4$, $C(g) = 3$, $C(h) = 5$ and $C(i) = 6$. Moreover, $D(d) = D(f) = D(g) = \{2, 4, 6, 7, 8, 9\}$, $D(e) = \{2, 4, 8\}$ and $D(h) = D(i) = \{2, 7\}$. To compute our lower bound, we first compute the exact cost $C_{assigned}$ that has to be paid due to already assigned items. In our example a, b and c are assigned and $C_{assigned} = 145$. Secondly, one must compute the cost due to the set UV of unbound variables, $C_{unbound}$. We ease the matching of those items by assuming that they can be assigned to any slot $s \in \bigcup_{X_i \in UV} D(X_i) = \{2, 4, 6, 7, 8, 9\}$. To compute $C_{unbound}$, we map the item with the largest consumption with the lowest resource price slot, so i is matched with slot 2. We then proceed similarly for all remaining unbound items: h, f, e, g and d are respectively mapped with slot 7, 6, 9, 8 and 4. This computation can be done by sorting unbound variables and time slots by non-increasing order of consumption and non-decreasing order of price, respectively. In the end, $C_{unbound} = 6 \cdot 5 + 5 \cdot 5 + 4 \cdot 10 + 3 \cdot 10 + 3 \cdot 15 + 2 \cdot 25 = 220$. This is a lower bound since item e is matched with slot 9 so as to minimize the total cost, but this value is actually not in its domain. A lower bound for the total cost is then $T_{lb} = 145 + 220 = 365$.

In the general case, T_{lb} can be computed as follows:

1. Compute the exact cost $C_{assigned}$ due to the set of assigned variables $X_{assigned}$.
2. Sort the unbound variables $UV = X \backslash X_{assigned}$ by non-increasing order of C, UV^{sorted}.
3. Compute the set of unmapped values that are still part of the domains of the unbound variables, i.e., $UA = \bigcup_{X_i \in UV} D(X_i)$.
4. Sort UA by non-decreasing order of P, UA^{sorted}.
5. Compute the minimal cost mapping of variables of UV^{sorted} to the values of UA^{sorted}, i.e., $C_{unbound} = Prod_{cost}(UV^{sorted}, UA^{sorted})$. Notice that $C_{unbound}$ is a lower bound for the cost due to the unbound variables, as a variable might be mapped to a value that is not in its domain.
6. The lower bound is then $T_{lb} = C_{assigned} + C_{unbound}$.

As described in Section 3, one can achieve this computation efficiently. First, the sorts of variable and value sequences can be done once and for all, as they remain correct for the whole search process. Incremental and reversible data structures allow keeping the use of those sorts at any time. Moreover, the cost $C_{assigned}$ due to bound variables and the set $\bigcup_{X_i \in UV} D(X_i)$ can

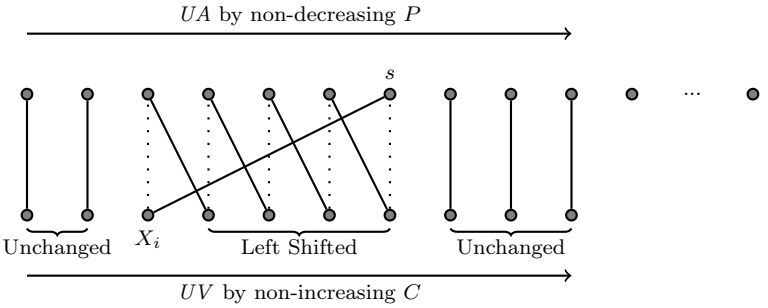

Fig. 2 Optimal mapping of variables if the variable $X_i = s$.

be computed incrementally. Once we have computed T_{lb}, a first constraint inference is the feasibility check:

$$T_{lb} > \overline{T} \implies \textbf{Fail} \tag{FC}$$

2.2 Domain Filtering

In order to filter a value s of the domain of a variable X_i, one needs to compute the reduced cost (value of T_{lb} *if* $X_i = s$), written $T_{lb}^{X_i=s}$. One can then use the inference rule:

$$\forall X_i \in X \, \forall s \in D(X_i) : T_{lb}^{X_i=s} > \overline{T} \implies X_i \neq s \tag{DF}$$

To compute $T_{lb}^{X_i=s}$, one has to compute the value $C_{unbound}$ under the constraint $X_i = s$, i.e.:

$$C_{unbound}^{X_i=s} = C(X_i) \cdot P(s) + Prod_{cost}(UV^{sorted} \backslash \{X_i\}, UA^{sorted} \backslash \{s\}) \tag{3}$$

$T_{lb}^{X_i=s}$ is then defined as $C_{assigned} + C_{unbound}^{X_i=s}$.

We are interested in computing Equation 3 for all variables $X_i \in UV$ and all values $s \in D(X_i)$. An inefficient way would be to recompute the second term for each pair (X_i, s). However, one can notice that when a variable is assigned to a given value of its domain, to compute the optimal mapping for the other variables by means of UV^{sorted} and UA^{sorted}, some variables are mapped to the same assignment as in the original optimal assignment (see Figure 2). Moreover, the variables that must be mapped to other assignments must all be mapped to the predecessor/successor of the value they were mapped with in the original matching (see Figure 2).

This observation allows us to compute the second term of Equation 3 in $\mathcal{O}(1)$, provided we have precomputed the 3 arrays CS, CS^{left} and CS^{right}, for which the i^{th} element is defined as:

$$\begin{aligned} CS_i &= Prod_{cost}(UV_{1..i}^{sorted}, UA^{sorted}) \\ CS_i^{left} &= Prod_{cost}(UV_{2..i}^{sorted}, UA^{sorted}) \\ CS_i^{right} &= Prod_{cost}(UV_{1..i}^{sorted}, UA_{2..|UA|}^{sorted}) \end{aligned}$$

Let us call ${X_i}^{pos}$ and s^{pos} the positions of X_i in UV^{sorted} and s in UA^{sorted}, respectively. Let us assume $s^{pos} > {X_i}^{pos}$ as in Figure 2 (there is symmetric reasoning for the case $s^{pos} < {X_i}^{pos}$, and the case $s^{pos} = {X_i}^{pos}$ is already managed by the feasibility checker). Then, computing the optimal mapping can simply be done with:

$$\underbrace{CS_{{X_i}^{pos}-1}}_{Unchanged} + \overbrace{C(X_i)\cdot P(s)}^{Assignment} + \underbrace{CS^{left}_{s^{pos}-1} - CS^{left}_{{X_i}^{pos}-1}}_{LeftShifted} + \overbrace{CS_{|UV|} - CS_{s^{pos}}}^{Unchanged}$$

Example 3. Let us reconsider values from Example 2 and let us assume $\overline{T} = 380$. Prior to filtering, we can precompute $CS = (30, 55, 95, 125, 170, 220)$, $CS^{left} = (25, 45, 75, 105, 135)$ and $CS^{right} = (30, 80, 120, 165, 240)$. We now wish to compute $C^{f=4}_{unbound}$. Since $f^{pos} = 3$ and $4^{pos} = 6$, we have, in $\mathcal{O}(1)$, $C^{f=4}_{unbound} = CS_2 + C(f)\cdot P(4) + CS^{left}_5 - CS^{left}_2 + CS_6 - CS_6 = 55+100+90+0 = 245$. If we add $C_{assigned}$, we finally have a cost of $390 > \overline{T}$ and therefore $4 \notin D(f)$.

Time Complexity Computing CS, CS^{left} and CS^{right} is $\mathcal{O}(n)$ in time, where $n = |UV|$. Then one must compute $C^{X_i=s}_{unbound}$ for each pair $(X_i, s) : X_i \in UV, s \in D(X_i)$ and perform a feasibility check in $\mathcal{O}(1)$. The time complexity for checking all pairs variable-value is therefore $\mathcal{O}(n \cdot m)$, where $m = \max_{X_i \in UV} |D(X_i)|$.

3 Algorithms

The implementation relies on several incremental and reversible data structures. First, we use reversible doubly-linked lists to maintain the sequences UV and UA, ordered by non-increasing C and non-decreasing P, respectively. They are maintained during search and their state is retrieved upon backtracking thanks to trailing (see [31,1]). Second, we use an array of reversible sparse sets [5], $X_{withValue}$, keeping track of the variables whose domain contain a given value s. This is useful to maintain UA efficiently. We also make use of *delta* sets. For a variable X_i, the set Δ_{X_i} contains all the values that were removed from the domain of X_i since the last call to the algorithm. This is done in an efficient manner, described in [5]. For instance, retrieving the set Δ_{X_i} or its size is $\mathcal{O}(1)$. Finally, we maintain the total cost due to assigned variables $C_{assigned}$ as a reversible integer.

Feasibility Checker Algorithm 1 allows computing T_{lb} and to perform the feasibility check. Lines 1-11 update the state by means of the freshly bound variables (i.e., not bound in the previous call): the variable is removed from the sequence of unbound variables UV, its exact contribution is added to $C_{assigned}$, and its assignment is removed from the possible assignments UA. Moreover, for each value removed from its domain, the variable is removed

from the set of variables that could be assigned to this value. If this set gets empty, no variable can be assigned to the value and this time slot is therefore removed from the sequence of possible assignments. A first check is done with $C_{assigned}$. Lines 15-22 update the set of remaining available assignments, i.e., they remove from UA the values that are no more contained in any domain. To do so, the sequence UV is traversed and for each variable, if its delta set is not empty, we update the corresponding sparse sets of values present in the delta set. Line 23 computes a lower bound for the cost due to unbound variables, by traversing UV and UA. The feasibility check is done in lines 24-26 and the total cost is lower bounded in line 27.

Algorithm 1: Incremental Computation of T_{lb}. The algorithm also serves as a feasibility checker with inference rule FC.

```
1  forall the X_i ∈ UV : |D(X_i)| = 1 do
2  |  UV ← UV\{X_i}
3  |  C_assigned ← C_assigned + C(X_i) · P(X_i)
4  |  UA ← UA\{X_i.value()}
5  |  forall the s ∈ Δ_{X_i} do
6  |  |  X_withValue(s) ← X_withValue(s)\{X_i}
7  |  |  if X_withValue(s) = ∅ then
8  |  |  |  UA ← UA\{s}
9  |  |  end
10 |  end
11 end
12 if C_assigned > T̄ then
13 |  return Fail
14 end
15 forall the X_i ∈ UV do
16 |  forall the s ∈ Δ_{X_i} do
17 |  |  X_withValue(s) ← X_withValue(s)\{X_i}
18 |  |  if X_withValue(s) = ∅ then
19 |  |  |  UA ← UA\{s}
20 |  |  end
21 |  end
22 end
23 C_unbound ← Prod_cost(UV, UA)
24 if C_assigned + C_unbound > T̄ then
25 |  return Fail
26 end
27 post(T ≥ C_assigned + C_unbound)
```

Domain Filtering Algorithm 2 achieves domain filtering. We assume CS, CS^{left} and CS^{right} are computed according to current state. It can be done by simply traversing the UV and UA sequences. Line 1 computes $P_{assignment}$ that maps unbound assignments to their position in UA. The loops in lines 2-18 apply the domain filtering rule for each pair $(X_i \in UV, s \in D(X_i))$. Depending on the relative positions of X_i and s, $C_{unbound}^{X_i=s}$ is computed. At line 10,

a feasibility check for this particular assignment is performed. If it is unfeasible, we remove the time slot of the domain of the variable and update the corresponding sparse set (and possibly UA if it is emptied).

Algorithm 2: Filter the domains by means of the inference rule DF.

```
1  P_assignment ← map_{a→p} /* Map from assignments to position in UA */
2  forall the X_i ∈ UV do
3  |  forall the s ∈ D(X_i) do
4  |  |  s^pos ← P_assignment(s)
5  |  |  if X_i^pos < s^pos then
6  |  |  |  C_unbound^{X_i=s} ← CS_{X_i^pos − 1} + C(X_i) · P(s) + CS^left_{s^pos − 1} − CS^left_{X_i^pos − 1}
   |  |  |                      +CS_{|UV|} − CS_{s^pos}
7  |  |  else if X_i^pos > s^pos then
8  |  |  |  C_unbound^{X_i=s} ← CS_{s^pos − 1} + C(X_i) · P(s) + CS^right_{X_i^pos − 1} − CS^right_{s^pos − 1}
   |  |  |                      +CS_{|UV|} − CS_{X_i^pos}
9  |  |  end
10 |  |  if C_unbound^{X_i=s} + C_assigned > T̄ then
11 |  |  |  D(X_i) ← D(X_i)\s
12 |  |  |  X_withValue(s) ← X_withValue(s)\{X_i}
13 |  |  |  if X_withValue(s) = ∅ then
14 |  |  |  |  UA ← UA\{s}
15 |  |  |  end
16 |  |  end
17 |  end
18 end
```

4 Continuous Casting Steel Production with Electricity Bill Minimization

To evaluate the ResourceCostAllDifferent (RCAD) constraint in practice, we first considered a real-life industrial problem, the Continuous Casting Steel Production with Electricity Bill Minimization (CCSPEBM). We compared the performances of our filtering algorithm with those of existing approaches. All experiments were performed with the *OscaR* solver [16], AMD Opteron processors (2.7 GHz), the Java Runtime Environment 8, and a memory consumption limit of 4Gb.

We first experimented with small-scale instances: RCAD provides generally additional pruning as compared with a decomposition in a sum of element constraints, sometimes by one order of magnitude. This is reflected from a time perspective: one order of magnitude can be gained and RCAD is at the very most 3 times slower. MinimumAssignment is always slower than RCAD and almost only brings overhead if they are used together. Moreover, since the exact reduced costs are not computed in MinimumAssignment as it is too

expensive[3], it appears RCAD prunes generally more than MinimumAssignment.

To challenge the scalability of our approach, we also considered larger instances. We discovered that RCAD is faster than the decomposition in a sum of element constraints for $\sim 75\%$ of the instances and that at least one order of magnitude is gained for $\sim 20\%$ of the instances. From a number of backtracks point of view, an order of magnitude is gained for $\sim 30\%$ of the instances. Finally, RCAD is slower by a maximum factor of 3.2, showing our approach is not only efficient but also robust.

Before providing the different results in detail, this section gives the definition of the problem, a CP model to solve it, and describes the comparison methodology we used.

4.1 Problem Definition

A *Continuous Casting Steel Production Problem* is a scheduling problem, for which a CP approach has been proposed in [12]. The main difference in our problem is that steel is produced by melting scrap in an Electric Arc Furnace (EAF) instead of a blast furnace. The total electricity cost minimization becomes the objective of the problem.

The CCSPEBM consists in scheduling a set of programs $\mathcal{P}$. A program p is made of a sequence $\mathcal{B}_p$ of batches. Each batch must be processed by a given sequence of machines: the EAF, the Ladle Furnace (LF), the Argon Oxygen Decarburization (AOD), and finally the Caster (CAS). For a given program, the AOD might actually not be used (this is known a priori). A machine can process at most one batch at any time. An illustration of a CCSPEBM schedule is given in Figure 3.

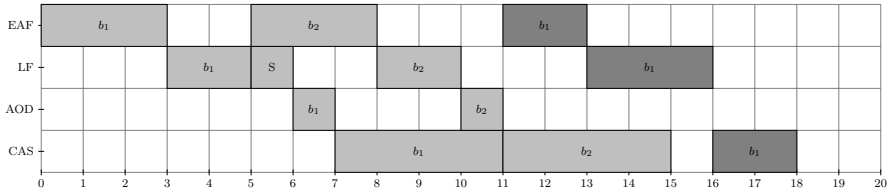

Fig. 3 Example of a CCSPEBM schedule with two programs, respectively with two batches and one batch. Activities of the first/second program are in light/dark gray. Only the first program makes use of the AOD and its first batch is stocked by the LF during one time slot (represented by the S square).

In the process, a batch is first melt using the EAF. Because the steel must be kept at a high temperature, the subsequent machines must process the batch as soon as the previous machine has finished. In addition, a batch

[3] According to preliminary experiments we conducted. We do not report any results when exact reduced costs are used.

cannot stay too long in the whole process, that is, there is a maximum span between the EAF and the CAS. The CAS must process all batches of a given program without interruption (see for instance the first program in Figure 3). The processing time of a batch by a machine is known a priori and cannot be modified. However, the LF/AOD might stock a batch during a small amount of time (see Figure 3).

The two electric machines requiring a significant amount of electrical energy are the EAF and the LF. A *consumption profile* is associated to the processing of a batch by a machine, that is, a function that provides the (non-null) consumption of each time period of the processing time. The profiles follow bounded ramp functions: the first slot requires a given amount and then stays constant with another higher consumption for the remaining duration. The profile can be different for each batch when processed by the EAF but it is the same for all batches processed by the LF. It is assumed that the LF does not consume energy while it stocks a batch since the consumption is negligible, i.e., the consumption required to stock a batch is 0. Finally, the electricity prices can vary at each time slot of the whole horizon.

4.2 CP Model

A standard CP scheduling approach is used for solving this problem. The processing of a batch by a machine is represented by an activity. Each activity is modeled with three integer variables used to represent its start, duration and end: s, d and e, such that $s + d = e$. For a machine m, s_m denotes the array of starting times of all activities executed on m, and s_m^b the starting time of the processing of the batch b by machine m (similar notations are used for duration and ending time). We allow d_{lf} and d_{aod} to vary since the LF/AOD may stock a batch for a moment (see the S square in Figure 3). For every batch b, the different activities must be scheduled continuously:

$$\forall p \in \mathcal{P}, \forall b \in \mathcal{B}_p : e_{eaf}^b = s_{lf}^b \wedge e_{lf}^b = s_{aod}^b \wedge e_{aod}^b = s_{cas}^b$$

All batches of a given program must be processed in sequence by the caster and without any interruption:

$$\forall p \in \mathcal{P}, \forall i \in 1..|\mathcal{B}_p| - 1 : e_{cas}^{b_i} = s_{cas}^{b_{i+1}}$$

A machine can only process one batch at a time. This is modeled with unary resource constraints [29]:

$$\begin{aligned}&\textsc{unary}(s_{eaf}, d_{eaf}, e_{eaf}) \wedge \textsc{unary}(s_{lf}, d_{lf}, e_{lf}) \wedge \\ &\textsc{unary}(s_{aod}, d_{aod}, e_{aod}) \wedge \textsc{unary}(s_{cas}, d_{cas}, e_{cas})\end{aligned}$$

Finally, a batch can not stay too long in the whole process:

$$\forall b \in \mathcal{B} : s_{cas}^b - e_{eaf}^b \leq max_{span}$$

where max_{span} is the maximum duration a batch can stay after the EAF and before the CAS.

Electricity Bill Minimization To model the objective, we split a given activity on the EAF/LF that must be processed during d time slots into d consecutive chunks with a duration of 1 time slot. Each of these chunks is then an item to be produced and has a consumption given by the batch consumption profile on the machine. The objective is modeled with Equation 1 and is to be minimized. We compare the following modelings of Equation 1:

- *SumElements*, using a sum of ELEMENT constraints. In this case, we do not split activities into chunks. We precompute the total cost of starting an activity at a given time and use this cost in the ELEMENT constraint.
- *SumElements* $\cup$ *MinAssignment*, using a MINIMUMASSIGNMENT constraint [10] additionally for each consuming machine. The cost of assigning a chunk at a time slot s is simply its consumption in the profile of the activity multiplied by the price at slot s. We do not use the exact reduced costs [8] that appeared to require a prohibitive computation time for the size of the considered instances.
- *SumElements* $\cup$ *RCAD*, using our constraint additionally, since *RCAD* and *SumElements* actually do not subsume each other. *RCAD* alone can miss some filtering as compared to *SumElements* because of the union of domains relaxation in our procedure.
- *SumElements* $\cup$ *RCAD* $\cup$ *MinAssignment*, using all constraints altogether.

4.3 Comparison Methodology

Replay Evaluation To compare the different models, we use the *Replay Evaluation* framework introduced in [25]. It allows assessing the benefit of using stronger propagation while removing the unpredictable effects of dynamic search heuristics. The general idea is to generate a search tree using the *baseline* model, i.e., the one with the weakest pruning (in our case, *SumElements*) and save it in memory. The saved tree is then traversed with the different models in order to get metrics such as solving time and number of backtracks. This evaluation allows ensuring that any gain in time/number of backtracks can be attributed to additional propagation solely, and not to the dynamic search heuristic.

To generate the search tree, we use a simple first-fail search heuristic, the focus of this work being on propagation: we first branch on the variable with the smallest domain. For the value heuristic, prior to search and for each consuming activity, we order the time slots by non-decreasing order of total processing cost of starting the activity at the time slot. Let s be the available slot minimizing cost of batch b on machine m, we then branch in a ternary fashion in this order: $s_m^b = s$, $s_m^b < s$ and $s_m^b > s$. For non-consuming activities, we assign the activity to its minimum starting time on the left branch, and remove this assignment on the right branch.

Instance Generation This work being part of an industrial project, we discussed with consultants of the *N-side* company[4] in order to generate realistic instances, and we use real historical electricity prices on the EU market. Our instances have between 2 and 15 batches per program, and their duration is between 3 and 4 slots at the EAF, exactly 4 slots at the AOD/LF, and between 4 and 5 slots at the CAS. 80% of the programs do use an AOD and possible consumptions vary between 1 and 100. We ensure that there are enough programs so that the caster is used between 60% and 80% of the horizon. Finally, max_{span} amounts to 150% of the sum of processing times of a batch by the LF and the AOD.

4.4 Small Instances

We first consider 158 small instances (96 time slots). Search trees were generated with a time limit of 30 s. and a backtrack limit of 500000. The results are given in Figures 4, 5, 6.

In Figure 4, one can see that *SumElements* $\cup$ *RCAD* is sometimes faster than *SumElements*, but not always. However, the factor is generally not large when it is slower (at most ~ 3). The number of backtracks is almost always reduced, sometimes significantly. For only a few instances there is no additional pruning, which explains the slowest execution times due to the overhead induced by our constraint. We perform an extended comparison of those approaches for larger instances in the next section.

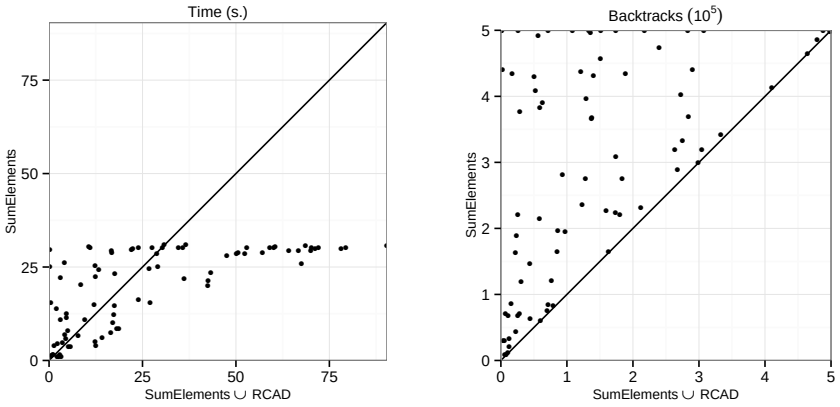

Fig. 4 Comparison of *SumElements* and *SumElements* $\cup$ *RCAD* on small instances.

Comparisons of *SumElements*$\cup$*MinAssignment* and *SumElements*$\cup$*RCAD* are reported in Figure 5. An important observation is that *SumElements* $\cup$

[4] `http://www.n-side.com/`

MinAssignment is always slower than *SumElements*∪*RCAD*. Interestingly, one can see that *SumElements*∪*RCAD* prunes generally more than *SumElements*∪ *MinAssignment* : the reason is that *RCAD* is not subsumed by *MinAssignment* since we use the version from Focacci et al. [9] that uses the linear programming reduced costs and not the exact ones. It would be subsumed using the exact reduced costs as in [8] but unfortunately computing them was too costly according to first preliminary experiments we conducted.

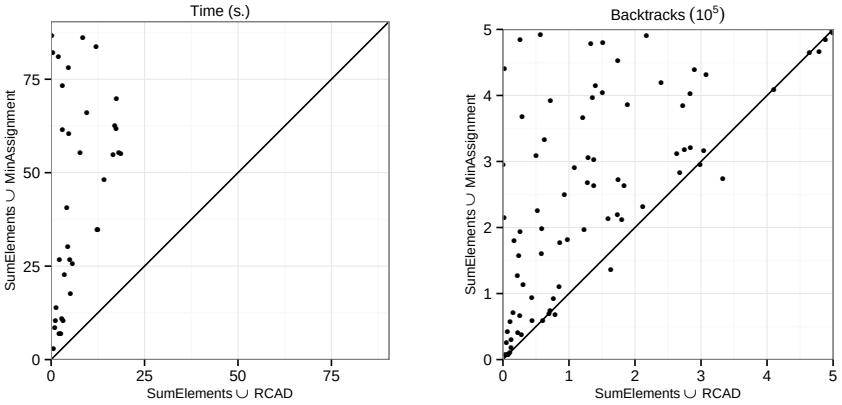

Fig. 5 Comparison of *SumElements*∪*MinAssignment* and *SumElements*∪*RCAD* on small instances.

We therefore make a last comparison, between *SumElements*∪*MinAssignment* ∪*RCAD* and *SumElements*∪*RCAD* (see Figure 6). One can observe that MinimumAssignment often only brings overhead: the data points are generally close to the equality line for the number of backtracks, while the search times are slower. As a conclusion, MinimumAssignment does not scale well and we will not use it for our comparison on larger instances.

4.5 Large Instances

To challenge the scalability of our approach, we generated 227 larger instances with 480 time slots. We generated search trees using a time limit of 300 s., in order to grasp more information about the performances of both approaches during search. Results are given in Figure 7. As for small instances, one can see that there is often an important gain in terms of number of backtracks. There is no additional pruning only for a few instances. From an execution time perspective, one can observe the replays for *SumElements* all takes around 300 s., which is expected since it is the model used to generate the search trees. Moreover, *SumElements* ∪ *RCAD* is faster for most of the instances.

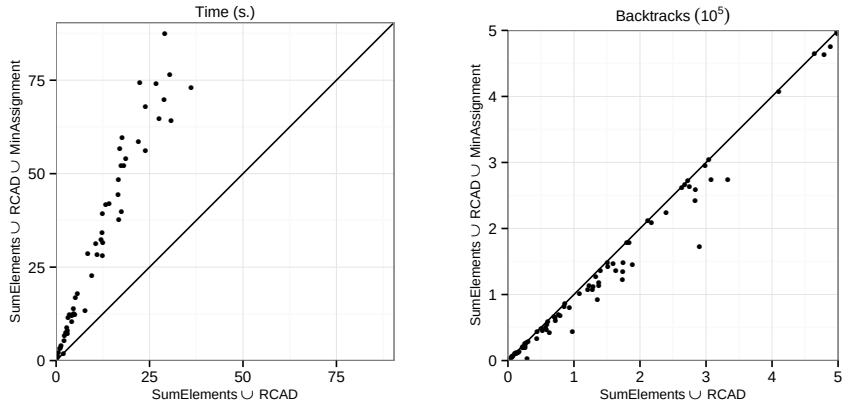

Fig. 6 Comparison of $SumElements \cup MinAssignment \cup RCAD$ and $SumElements \cup RCAD$ on small instances.

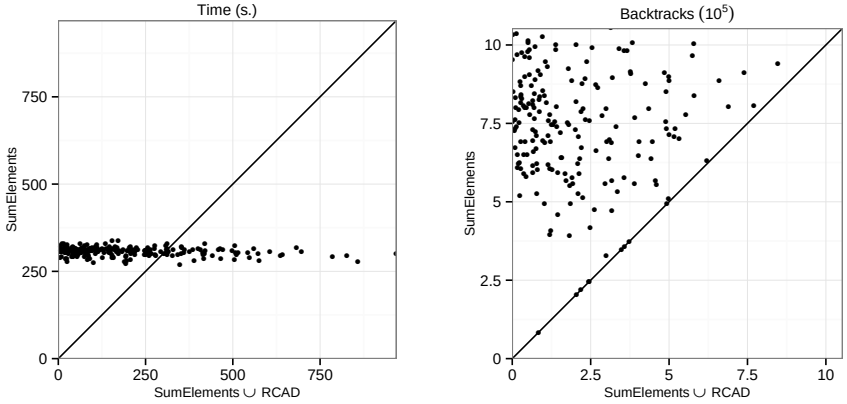

Fig. 7 Comparison of $SumElements$ and $SumElements \cup RCAD$ on large instances.

In order to quantify and better visualize the gain obtained by our approach, we constructed so-called *performance profiles* [7], that we built with a public web tool [26] made available to the community.[5] Performance profiles are cumulative distribution functions of a performance metric ratio τ. In our case, τ is a ratio of time or of number of backtracks. For time, the function is defined as in [25] with (it is similar for backtracks):

$$F_m(\tau) = \frac{1}{|\mathcal{I}|} \left| \left\{ i \in \mathcal{I} : \frac{time_{replay}(m,i)}{time_{replay}(b,i)} \leq \tau \right\} \right|$$

where $\mathcal{I}$ is the set of considered instances, m is a model and b is the baseline model (in our case $SumElements$).

[5] Accessible at `http://performance-profile.info.ucl.ac.be/`.

Figures 8 and 9 respectively provide the profiles for the number of backtracks and the time metrics. Let us first read the profiles for the number backtracks: $F_{SumElements \cup RCAD}(1) \simeq 95\%$, so we achieve more pruning for $\sim 95\%$ of the instances. Furthermore, $F_{SumElements \cup RCAD}(0.1) \simeq 30\%$, which means that we gain at least one order of magnitude on the number of backtracks for $\sim 30\%$ of the instances.

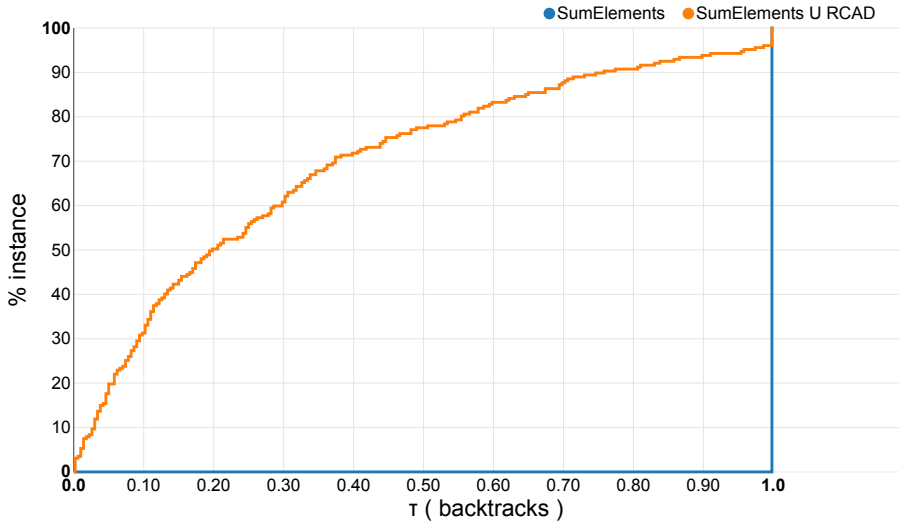

Fig. 8 Backtracks performance profile on large instances of the CCSPEBM.

From an execution time perspective, $F_{SumElements \cup RCAD}(1) \simeq 75\%$, so our approach is faster for $\sim 75\%$ of the instances. One can also observe that at least one order of magnitude is gained for $\sim 20\%$ of the instances. Finally, notice that $F_{SumElements \cup RCAD}$ reaches 100% at $\tau = 3.2$ This means that if $SumElements \cup RCAD$ is slower than $SumElements$, it is at most by a factor of 3.2. This illustrates that our approach is also robust as it does not deteriorate much the execution time in case it does not bring any additional filtering.

5 The Product Matrix Travelling Salesman Problem

In order to evaluate our constraint on a second problem, let us consider a particular case of the well-known Asymmetric Travelling Salesman Problem (ATSP), the Product Matrix Travelling Salesman Problem (PMTSP), that was first formally described in [18]. We recently added this problem to the CSPLib [4]. We consider this problem because it is more academic than the previous one. The experiments show that our approach is the fastest for $\sim 90\%$ of the considered instances, even if we also experimented with the

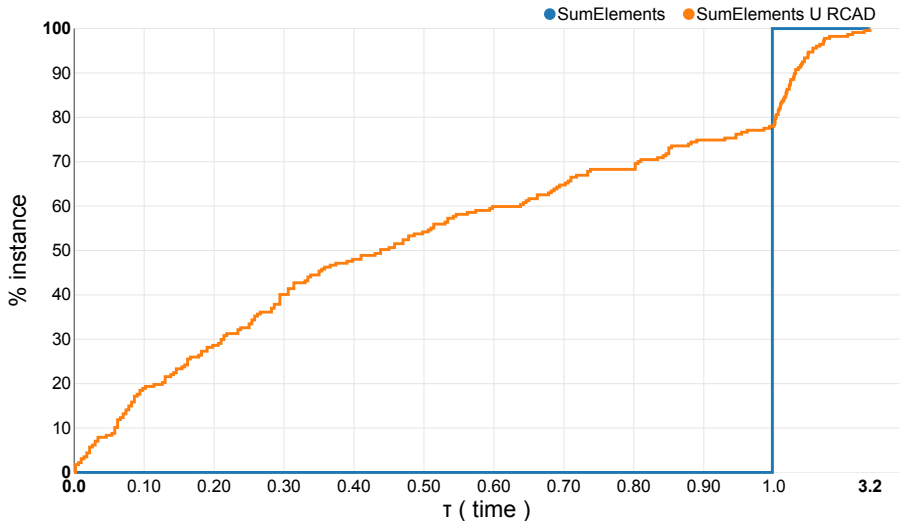

Fig. 9 Time performance profile on large instances of the CCSPEBM.

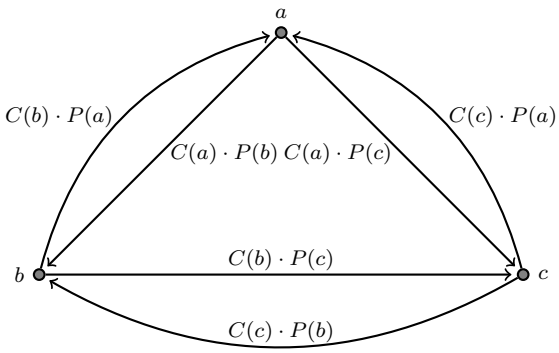

Fig. 10 Instance of the Product Matrix Travelling Salesman Problem.

Concorde solver[2,3], a custom state-of-the-art Branch-and-Cut Mixed Integer Programming solver for the Travelling Salesman Problem (TSP).

Definition Given two vectors of n elements C and P, one can construct a simple graph G with n vertices, such that the directed edge from the vertex i to the vertex $j \neq i$ has a cost equal to $C(i) \cdot P(j)$. The distance matrix is therefore the matrix product between the vectors C and P, hence the name of the problem. An instance of the PMTSP is illustrated in Figure 10. The problem consists in finding an Hamiltonian circuit of minimum total cost in the graph G. The problem was proven to be NP-hard in [21,13].

Solving In CP, the ATSP is usually solved with a successor model (see [17]): an array of n variables *succ* is used to represent the successors for each vertex.

One then impose that the array is a Hamiltonian circuit (filtering from [17]) :

$$\text{CIRCUIT}(succ)$$

One can model the objective with either: 1) a sum of ELEMENT constraints, 2) a MINIMUMASSIGNMENT constraint, or 3) the RESOURCECOSTALLDIFFERENT constraint. The sum of ELEMENT constraints misses a lot of pruning in this case, so we will not consider it here. Let us respectively call *RCAD*, *MinAssignment* and *MinAssignment* $\cup$ *RCAD* the models with RESOURCECOSTALLDIFFERENT, MINIMUMASSIGNMENT, and both algorithms.

Results We generated 50 instances with 200 vertices. We compared the performances of the different CP models and those of the Concorde solver[2,3], a custom state-of-the-art Branch-and-Cut Mixed Integer Programming solver for the TSP. Since Concorde does not solve ATSPs, we used the standard reduction from ATSPs to TSP given in [15] to solve our instances with this solver. We used the *Conflict Ordering Search* heuristic [11] to solve the problems in CP. For all approaches, the instances were solved to optimality. All experiments were performed on a machine with an Intel Core i7 (2,2 GHz) processor. The results are given in Figure 11, as a standard performance profile as defined in [7].

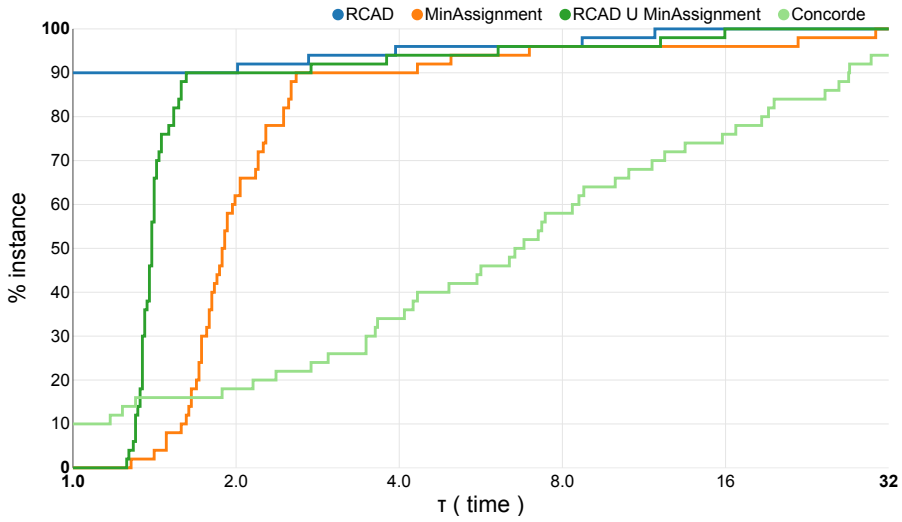

Fig. 11 Performance profile for the different CP models and the Concorde solver on the 50 generated instances of the PMTSP.

First, one can observe that *MinAssignment* is the slowest of the CP approaches, which is a conclusion already done in the last section. Using our propagator *additionnaly* provides some speed-up since it can provide more pruning or detect a failure faster at a given node since it has a higher priority

in the propagation queue. Yet, only using our algorithm is the best option, since it is always faster than the other CP models. The overhead of using MinimumAssignment to possibly get more pruning simply does not pay off in this case.

An interesting observation is that Concorde is only the fastest solver for $\sim 10\%$ of the instances, while *RCAD* is the fastest for the remaining $\sim 90\%$ instances. Moreover, the solving time ratio for Concorde, as compared with *RCAD*, is often large: 5% of the instances are even not solved in a factor 32 of the time required by *RCAD*. At the same time, when *RCAD* is not the fastest, it requires at the very most a factor of 12 as compared with Concorde. This indicates that even when *RCAD* is not the fastest, it is more robust. From this experiment, one can conclude that the CP technology should be the preferred one to solve this problem.

6 Conclusion

In this work, we considered problems where a set of items, each requiring a different amount of resource, must be assigned to different slots, and where the price for a unit of resource can vary at each slot. In this class of problems, the objective is to assign items such that the overall resource cost is minimized. We showed two ways of modeling such an objective in CP and their limitations (limited inference or scalability). To cope with those limitations, we introduced a new filtering algorithm, that we evaluate on a real and large-scale industrial problem, the CCSPEBM. The results demonstrate that, especially for large instances, our approach is often faster, sometimes with an important speed-up: an order of magnitude is gained for $\sim 20\%$ of the large instances and $\sim 75\%$ of them are solved faster. Moreover, our algorithm is robust, in the sense that when it is slower, it is by a small factor (3.2 at the very most). Results also illustrate that for a non-negligible number of the large instances, we can get an important gain in terms of number of backtracks as compared with a decomposition: a reduction of at least one order of magnitude is obtained for 30% of the instances. We also considered a more academic problem, the PMTSP. We compared our approach with existing ones in CP, as well as with Concorde, a custom TSP solver. The results show that our approach is the fastest in CP, and that it outperforms Concorde for 90% of the instances, sometimes by an important factor (some instances could not be solved in a factor 32 as compared with our approach).

Future Work We would like to extend our algorithm to handle the production of several items at the same time slot. This would therefore be a particular case of GccCost for cardinalities larger than 1. Moreover, we would like to study the performances of our algorithm on other kinds of problems such as discrete lot sizing problems [14]. Finally, in the case of the CCSPEBM, an important future work is the evaluation of the robustness of the approach in

function of errors in prices prediction, as there is a certain level of uncertainty in prices forecasts.

Acknowledgements This Research has been supported by the "Service Publique de Wallonie Direction générale opérationnelle de l'Economie, de l'Emploi & de la Recherche" under the scope of the InduStore project.

References

1. Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
2. David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. Concorde TSP solver, 2006.
3. David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2011.
4. Sascha Van Cauwelaert and Pierre Schaus. CSPLib problem 075: Product matrix travelling salesman problem. `http://www.csplib.org/Problems/prob075`.
5. Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-sets for domain implementation. In *International workshop on Techniques foR Implementing Constraint programming Systems*, pages 1–10, 2013.
6. Cyrille Dejemeppe, Olivier Devolder, Victor Lecomte, and Pierre Schaus. Forward-checking filtering for nested cardinality constraints: Application to an energy cost-aware production planning problem for tissue manufacturing. In *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, pages 108–124. Springer, 2016.
7. Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.
8. Sylvain Ducomman, Hadrien Cambazard, and Bernard Penz. Alternative filtering for the weighted circuit constraint: Comparing lower bounds for the TSP and solving TSPTW. In *AAAI Conference on Artificial Intelligence*, 2016.
9. Filippo Focacci, Andrea Lodi, and Michela Milano. Integration of CP and OR methods for matching problems. In *International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 1999.
10. Filippo Focacci, Andrea Lodi, Michela Milano, and Daniele Vigo. Solving TSP through the integration of OR and CP techniques. *Electronic notes in discrete mathematics*, 1:13–25, 1999.
11. Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 140–148. Springer, 2015.
12. Steven Gay, Pierre Schaus, and Vivian De Smedt. Continuous casting scheduling with constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 831–845. Springer, 2014.
13. Paul C. Gilmore, Eugene L. Lawler, and David Shmoys. Well-solved special cases of the traveling salesman problem. In John Wiley & Sons, editor, *The Traveling Salesman Problem*. 1985.
14. Vinasétan Ratheil Houndji, Pierre Schaus, Laurence Wolsey, and Yves Deville. The stockingcost constraint. In *International Conference on Principles and Practice of Constraint Programming*, pages 382–397. Springer, 2014.
15. Roy Jonker and Ton Volgenant. Transforming asymmetric into symmetric traveling salesman problems. *Operations Research Letters*, 2(4):161–163, 1983.
16. OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.
17. Gilles Pesant, Michel Gendreau, Jean-Yves Potvin, and Jean-Marc Rousseau. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998.

18. Robert D. Plante, Timothy J. Lowe, and R. Chandrasekaran. The product matrix traveling salesman problem: an application and solution heuristic. *Operations Research*, 35(5):772–783, 1987.
19. Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI Conference on Artificial Intelligence*, volume 94, pages 362–367, 1994.
20. Jean-Charles Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, 7(3-4):387–405, 2002.
21. V. I. Sarvanov. On the complexity of minimizing a linear form on a set of cyclic permutations. In *Dokl. Akad. Nauk SSSR*, volume 253, pages 533–535, 1980.
22. Meinolf Sellmann. An arc-consistency algorithm for the minimum weight all different constraint. In *International Conference on Principles and Practice of Constraint Programming*, pages 744–749. Springer, 2002.
23. Helmut Simonis and Tarik Hadzic. A family of resource constraints for energy cost aware scheduling. In *Third International Workshop on Constraint Reasoning and Optimization for Computational Sustainability, St. Andrews, Scotland, UK*, 2010.
24. Helmut Simonis and Tarik Hadzic. A resource cost aware cumulative. In *Recent Advances in Constraints: 14th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2009, Barcelona, Spain, June 15-17, 2009, Revised Selected Papers*, pages 76–89. Springer, 2011.
25. Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. Understanding the potential of propagators. In *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, pages 427–436. Springer, 2015.
26. Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. A visual web tool to perform what-if analysis of optimization approaches. *arXiv preprint arXiv:1703.06042*, 2017.
27. Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity: An experience with AI and OR techniques. In *AAAI Conference on Artificial Intelligence*, pages 660–664, 1988.
28. Willem-Jan van Hoeve. The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015, 2001.
29. Petr Vilím. $\mathcal{O}(n.log(n))$ filtering algorithms for unary resource constraint. In Jean-Charles Régin and Michel Rueher, editors, *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, pages 335–347. Springer, 2004.
30. Gilles Madi Wamba and Nicolas Beldiceanu. The taskintersection constraint. In *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, pages 246–261. Springer, 2016.
31. David H. D. Warren. *An abstract Prolog instruction set*, volume 309. Artificial Intelligence Center, SRI International Menlo Park, California, 1983.
32. Rafał Weron. Electricity price forecasting: A review of the state-of-the-art with a look into the future. *International Journal of Forecasting*, 30(4):1030–1081, 2014.
33. Rolf Wüstenhagen and Michael Bilharz. Green energy market development in germany: effective public policy and emerging customer demand. *Energy policy*, 34(13):1681–1696, 2006.