A Decision Support System for Food Recycling based on Constraint Logic Programming and Ontological Reasoning

Federico Chesani[1][0000−0003−1664−9632] and Giuseppe Cota[2][0000−0002−3780−6265] and Evelina Lamma[2][0000−0003−2747−4292] and Paola Mello[1][0000−0002−5929−8193] and Fabrizio Riguzzi[3][0000−0003−1654−9703]

[1] Dipartimento di Informatica Scienza e Ingegneria – University of Bologna
Viale Risorgimento 2, 40136, Bologna, Italy
name.surname@unibo.it
[2] Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[3] Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
name.surname@unife.it

Abstract. In 2011, FAO estimated that about one third of the total production of edible food for human consumption, gets lost or wasted, globally. In industrialized countries, more than 40% of the food losses occur at retail and consumer levels. A considerable part of the wasted food can be reused. The SORT project aims at "recycling" food in excess by converting it into animal feed or fuel for biogas/biomass power plants. During this process of reconditioning it is necessary to make choices in order to minimize the costs and maximize the earnings. However, due to the extremely complex nature of the process, it is not possible for a human being to make these choices at runtime and in an optimal manner. In these cases, Decision Support Systems (DSS) can be of help.
In this paper we propose a DSS based on the Constraint Logic Programming (CLP) paradigm and ontology reasoning for finding the optimal solution. The information about feed processing and feed categories were extracted from Regulation (EU) 2017/1017 and stored into an ontology. Finally, we provide an evaluation of our system on several synthetic datasets with different search settings.

1 Introduction

In 2011, FAO (Food and Agriculture Organization of the United Nations) estimated that about one third of the total production of food for human consumption, 1.3 billion tonnes per year, gets lost or wasted worldwide [3].

Food waste has strong impact on deforestation, ecosystem degradation and on natural and human resource consumption, including water, land, energy, labour and capital. Moreover it produces greenhouse gas emissions, contributing to global warming and climate change.

Food loss refers to losses of edible food mass throughout the early stages of the supply chain. Food losses take place during the stages of production, harvesting, treatment, storage and processing. Food losses occurring at the end of the food chain (retail distribution and final consumption) are called *food waste*, which depend on the behaviour of retailers and consumers. For instance, at retail level, large quantities of food are wasted due to appearance standards (e.g. dented but unbroken cans, fruit with imperfections, etc.).

In industrialized countries, more than 40% of the food losses are food waste. In particular, the quantity of food wasted at consumer level in industrialised countries amounts to 222 million tonnes, roughly equivalent to the food production available in sub-Saharan Africa (230 million tonnes) [3]. The European Commission estimated that in 2012 about 88 million tonnes of food loss and waste were produced, with a related cost of 143 billion euros [9].

A considerable part of the food waste (currently destined for landfill) can be reused. The SORT project[4] aims at "recycling" food items in excess or no longer consumable by humans. The main goals of the SORT project are:

– Recovering the packaging for subsequent recycling.
– Reconditioning the to-be-wasted food into a sellable product that can be used as feed material for animals or as fuel for biogas/biomass power plants.

The process envisioned by SORT is complex and involves several parties and actors (retail companies, vehicles, machinery for food processing, warehouses, animal feed factories, biogas/biomass power plants, etc.). In order to maximize the earnings and minimize the costs, several decisions must be made during the process, which must take into account several variables and purposes (food items used for reconditioning, maximizing profit, the cost of using machinery for processing, storage costs, etc.). Given the extremely complex nature of the SORT process, a Decision Support Systems (DSS) can be used to aid the process manager to make choices.

In this paper we present a DSS for the SORT process developed using the Constraint Logic Programming (CLP) paradigm for finding the optimal solution. The knowledge about feed processes and categories was extracted from Regulation (EU) 2017/1017 and stored into an OWL ontology. Information represented in a logical formalism, such as OWL, enables the CLP engine to perform reasoning for finding the best solution.

The paper is organized as follows. Section 2 illustrates the processing of food articles inside a SORT plant. Section 3 provides some background knowledge of the technologies used for this DSS. In particular, it recalls the main concepts of description logics and Constraint Logic Programming. Section 4 presents the SORT Ontology, used to represent the feed materials and processes. Section 5 illustrates the CLP(FD) model of the DSS. Section 6 shows experimental results on synthetic datasets. Section 7 concludes the paper.

[4] The SORT project is promoted by the Italian Ministry of Education, Universities and Research. Code: SCN_00367, Ministerial Act n. 2427 `http://attiministeriali.miur.it/anno-2015/ottobre/dd-28102015-(1).aspx`.

2 SORT Process Chain

The food articles to be reused are gathered at large retailers and collection centers, packaged and sent to the so-called SORT plants. Every box of food articles has an RFID chip for identification and tracking. The customers of a SORT plant are feed factories and biogas/biomass power plants, which can make different orders of feed materials.

Figure 1 shows the process chain of a SORT plant. When the boxes reach the SORT plant, they are opened and their contents are poured into an hopper. Then the Sorter machinery sorts the food articles into other boxes in order to create, if possible, homogeneous boxes, i.e. boxes that contain similar food articles. It is worth noting that, during this phase, the Sorter is not managed by the DSS. The boxes are then stored in an automated warehouse. This phase of the process chain is called *input sorting*.

In the warehouse we can have both homogeneous content boxes and heterogeneous content boxes. The presence of boxes with heterogeneous content certainly represents a complication, but also an interesting and stimulating challenge from a computational point of view.

If the articles contained in a box are selected by the DSS to satisfy an order, that box will be opened and its content will be poured into Sorter's hopper again and each article will be sent to one of the N regular output destinations or sent back, according to the guidelines specified by the DSS. At the end of each output destination there is a box that gathers all the sorted articles. This phase of the process chain is called *output sorting*. Finally, the selected articles are sent to machinery for unpacking and food processing or sent back to the warehouse.

Example 1. Consider the case where the SORT plant received two orders, o_1 and o_2, that request two different types of feed materials and consider the situation depicted in Fig. 1. Moreover, let us suppose that both o_1 and o_2 must be satisfied. The DSS then assigns the first output destination (out 1) to o_1 and the second output destination (out 2) to o_2 and, at the same time, assigns some of the food articles contained in the boxes with identifiers 5 and 17 to o_1 and some of the articles contained in boxes 17 and 131 to o_2. Boxes 5, 17 and 131 are then opened and their content is poured into the hopper. The Sorter then sends the articles chosen to satisfy o_1 to out 1 and the articles chosen to satisfy o_2 to out 2.

It should be noted that in the SORT plant, there is only one sorting line (for the time being). That means that the Sorter machinery depicted in Figure 1 on the left side of the warehouse (input sorting phase) is the same Sorter machinery on the right of the warehouse (output sorting phase). Consequently, during the output sorting phase, the Sorter will not be able to perform the input sorting step, i.e. it cannot process the boxes coming from large retailers and collection centers.

2.1 Use cases of the DSS

The envisaged user of the DSS is the manager of the SORT plant. Its job is to choose, day by day, which orders must be satisfied and which articles must be

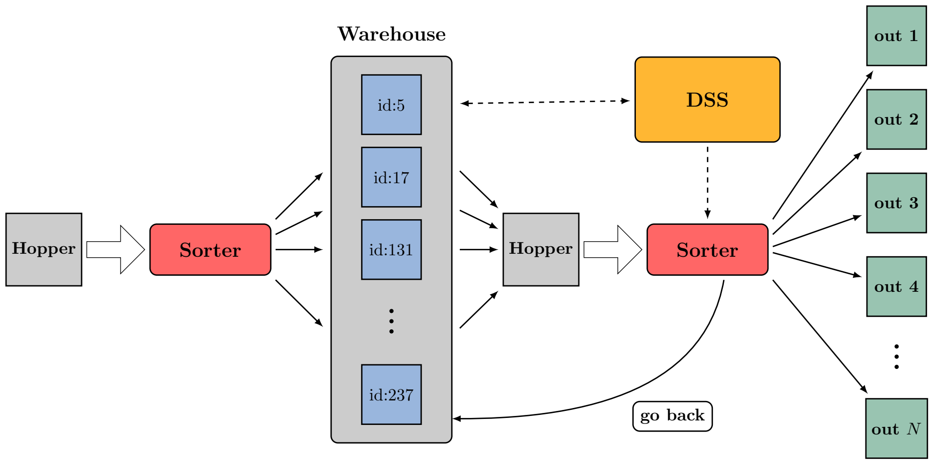

Fig. 1. SORT chain. Every box in the warehouse has RFID chip with a unique identifier.

used to satisfy them in order to maximize the earnings, according to the available articles in the warehouse and to the received orders. These decisions are made with the help of the DSS.

One of the components of the DSS is a web-based application (still under development) which allows the user to perform the necessary operations to achieve the optimal solutions. We planned four use cases of the DSS:

1. **Manual selection of orders and boxes** The user selects which orders must be satisfied that day and which boxes must be used. The DSS assigns each user-selected order to an output destination and uses *only* the food articles contained in the selected boxes to satisfy them.
2. **Manual selection of orders** The user selects only the orders that must be satisfied that day. The DSS assigns each selected order to an output destination and uses the articles available in the warehouse to satisfy them.
3. **Manual selection of boxes** The user selects the boxes and the DSS, according to the optimization criterion, calculates which orders must be satisfied by using *only* the food articles contained in the selected boxes.
4. **Automatic selection** The DSS automatically selects the orders to satisfy, i.e. it automatically assigns an output destination to each chosen order, and chooses which articles must be used to satisfy them, according to the optimization criterion.

3 Background

3.1 Description Logics

An ontology describes the concepts of the domain of interest and their relations with a formalism that allows information to be processable by machines. The

Web Ontology Language (OWL) is a family of knowledge representation languages, based on description logics, for authoring ontologies or knowledge bases. OWL 2 [11] is the last version of this language and is a W3C recommendation since 2012. We used this logical formalism to represent the knowledge about feed processes and categories.

Description Logics (DLs) are fragments of FOL languages used for modeling knowledge bases (KBs) that exhibit nice computational properties such as decidability and/or low complexity [1].

There are many different DL languages that differ in the constructs that are allowed for defining concepts (sets of individuals of the domain) and roles (sets of pairs of individuals). Below we illustrate the $\mathcal{ALC}$ DL, which is one of the most common fragments and it is the basis for many other DLs.

Let us consider a set of *atomic concepts* $\mathbf{C}$, a set of *atomic roles* $\mathbf{R}$ and a set of individuals $\mathbf{I}$. In $\mathcal{ALC}$ a *concept* C is either $C_1 \in \mathbf{C}$, $\bot$, $\top$, $(C_2 \sqcap C_3)$, $(C_2 \sqcup C_3)$, $\neg C$, $\exists R.C$ or $\forall R.C$, where C_2 and C_3 are concepts and $R \in \mathbf{R}$.

A *TBox* $\mathcal{T}$ is a finite set of *concept inclusion axioms* $C \sqsubseteq D$, where C and D are concepts. An *ABox* $\mathcal{A}$ is a finite set of *concept membership axioms* $a : C$, *role membership axioms* $(a, b) : R$, *equality axioms* $a = b$ and *inequality axioms* $a \neq b$, where $C \in \mathbf{C}$, $R \in \mathbf{R}$ and $a, b \in \mathbf{I}$. A $\mathcal{ALC}$ KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ consists of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$. It is usually assigned a semantics in terms of interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty *domain* and $\cdot^{\mathcal{I}}$ is the *interpretation function*, which assigns an element in $\Delta^{\mathcal{I}}$ to each $a \in \mathbf{I}$, a subset of $\Delta^{\mathcal{I}}$ to each concept and a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to each role.

3.2 Constraint Logic Programming

A Constraint Satisfaction Problem (CSP) is defined by a set of *variables* $\mathbf{X} = \{X_1, \ldots, X_n\}$ and a set of *constraints* $\mathbf{C} = \{C_1, \ldots, C_m\}$ [8]. Each variable X_i has a domain D_i of possible values. A constraint defines a restriction over the possible combinations of values of a subset of variables. A solution of a CSP is an assignment of values to all variables that satisfies all the imposed constraints. Constraint solving means finding a solution to a given CSP.

Constraint Programming (CP) is a programming paradigm where the relations among the variables are expressed in the form of constraints. In CP, the user states the constraints and a general purpose constraint solver tries to solve them. Constraint Logic Programming (CLP) is a paradigm of programming languages that merges two declarative paradigms: Logic Programming (LP) and CP.

A CLP language is defined by the class (*sort*) of allowable variable domains and constraints. A CLP language over a constraint class X is denoted as CLP(X) [4]. For instance, CLP($\mathcal{R}$) is a CLP language in which variables can range on the reals [5], whereas in CLP(FD) variables range on finite domains.

In CLP(X) the constraints of class X can be added to the body of the usual logic programming clauses. During Selective Linear Definite (SLD) resolution, the constraint are not resolved, but they are added to a special data structure, called the *constraint store*. The store is then processed by an external solver, called *constraint solver*. The constraint solver checks whether the conjunction of constraints is satisfiable and, in case, modifies the store in order to obtain a simplified state of the variables. The act of modifying the constraint store is known under the term of *propagation*.

Besides variables and constraints, the user can also define an *objective function*: a function that the solver tries to maximize or minimize without violating any constraint. If an objective function is defined, like in our case, then the problem becomes an optimization problem.

We modelled our problem in Constraint Logic Programming on finite domains (CLP(FD)). Finite domain constraints are usually intended to be arithmetic constraints over finite integer domain variables. Therefore a CLP(FD) language needs a constraint system that is able to handle (i.e. store and propagate) this type of constraints.

There exist several implementations of logic programming and CLP(FD). For this project we decided to adopt ECL^{i}PS^e [7][5], which features a library for finite domains called `gfd`. This library interfaces ECL^{i}PS^e with Gecode's constraint solver [10]. To find the optimal solution we use ECL^{i}PS^e's branch-and-bound approach.

4 Knowledge Representation of Feed Materials

Communion Regulation (EU) 2017/1017 is the legislative text in the European Union regarding the production of animal feed. It contains the list of all processes and the list of all categories of feed materials. The list of the processes that feed materials may undergo is illustrated in Part B of the regulatory text, whereas the catalogue of feed materials is given in Part C.

As already mentioned in Section 3.1, we used the logical formalism OWL 2, based on DLs, to represent the knowledge about feed processes and categories. In particular, we used this regulation to generate an OWL 2 KB, called *SORT Ontology*, which can be used by the DSS to check the compatibility between an order and a food article.

4.1 SORT Ontology

Given the large number of processes and categories, we developed a simple program written in Java that parses the XML version of the aforementioned European regulation, and automatically outputs the SORT Ontology.

The tool allows the SORT system to be easily updated to future regulatory and legal developments. Moreover the ontology supports several languages[6].

Figure 2 shows an excerpt from the SORT Ontology in OWL 2 RDF/XML.

Representing the knowledge about feed materials and processes by means of an ontology allows to exploit a reasoner to check if an article can be used to

[5] `http://eclipseclp.org`

[6] Currently supported: Italian, English, Spanish, French and German.

```
<owl:Class rdf:about="#1">
    <rdfs:subClassOf rdf:resource="#Feed"/>
    <rdfs:label xml:lang="en">Cereal grains and products
       derived thereof</rdfs:label>
    <rdfs:label xml:lang="it">Cereali e prodotti derivati</
       rdfs:label>
    ...
</owl:Class>

<owl:Class rdf:about="#1.1.1">
    <rdfs:subClassOf rdf:resource="#1"/>
    <rdfs:comment xml:lang="en">Grains of Hordeum vulgare L.<
       /rdfs:comment>
    <rdfs:comment xml:lang="it">Grani di Hordeum vulgare L.</
       rdfs:comment>
    <rdfs:label xml:lang="en">Barley</rdfs:label>
    <rdfs:label xml:lang="it">Orzo</rdfs:label>
    ...
</owl:Class>
```

Fig. 2. Excerpt from the SORT Ontology in OWL 2 RDF/XML. Here we can see that subcategory 1.1.1 (Barley) is a subclass of category 1 (Cereal grains and products derived thereof). Moreover for each feed category and subcategory there are labels and comments in different languages.

satisfy an order (see *Compatible* function in Eq. 1 Section 5). Moreover, it allows to obtain new information about food articles by exploiting other ontologies (see the Open Food Facts project (`https://world.openfoodfacts.org/`).

5 CLP Model

In this section we illustrate the CLP model of the DSS and the Prolog implementation of some constraints.

5.1 Variables

The variables of our model are the list of orders $\mathbf{O} = \{o_1, \ldots, o_K\}$ and the list of food articles $\mathbf{A} = \{a_1, \ldots, a_M\}$, where K and M are the number of orders and articles, respectively. As we mentioned in Section 2, the sorting line, during the output sorting phase, can have N regular outputs (plus a special output for articles that should go back to the warehouse), therefore the domain of an order variable is

$$\forall i \in \{1, \ldots, K\}\ o_i \in \{0, \ldots, N\}$$

where 0 means that the order will not be satisfied.

Each article, instead, can be used to satisfy one and only one order. Therefore the domain of an article variable contains the order indices plus the values -1 and 0:

$$\forall j \in \{1, \ldots, M\}\ a_j \in \{-1, \ldots, K\}$$

where 0 means "stay", i.e. the article is not associated with any order and stays in the warehouse without moving, whereas -1 represents "go back", i.e. the article goes through the Sorter but it will be sent back to the warehouse.

For instance, $a_j = i$ with $i > 0$ means that the j-th food article will be used to satisfy the i-th order. $a_j = 0$ means that the j-th food article will not be used, whereas $a_j = -1$ means that the box which contains a_j will be opened, some of its articles will be used to satisfy some orders, but the j-th article will not be used and it should go back in the warehouse.

5.2 Constraints

We can split the constraints of our model into three categories: article constraints, order constraints and constraints for symmetry breaking.

Article Constraints Here we illustrate the constraints relating food articles to the boxes that contain them. In particular, these constraints are used to maintain the consistency of the warehouse.

The first article constraint simply states that articles can't be associated with incompatible orders

$$\forall j \in \{1, \ldots, M\}\ \forall i \in \{1, \ldots, K\}\ \neg \mathit{Compatible}(a_j, o_i) \Rightarrow a_j \neq o_i \qquad (1)$$

where $\mathit{Compatible}(a_j, o_i)$ is a function which checks if article belongs to a feed category compatible with the feed category requested by an order. To check the compatibility between an order and a food article a DL reasoner and the SORT Ontology described in Section 4 can be exploited. In order to save time, these compatibility checks can be performed before running the CLP engine. For instance, we could build a dictionary where each order is associated with a list of compatible articles and then search the best solution.

In the SORT environment, each order consists of a requested quantity of only one feed material and each article belongs to only one feed category. For this scenario, our DSS does not perform any reasoning task, at least for now. Function *Compatible* checks if feed category of the order matches with the feed category of the article. However, in the future, for more complex scenarios, ontological reasoning might be necessary.

The following constraint is fundamental to preserve the consistency of the boxes contained in the warehouse. In fact, once a box is opened, all of its contents are poured into the hopper before the Sorter and therefore all the items contained in that box must be sorted by the Sorter (one of the possible destinations could be back to stock). It is not possible to have situations in which an article present in a box goes through the Sorter while the other articles contained in the same box remain in storage.

$$\forall j \in \{1, \ldots, M\}\ a_j > 0 \Rightarrow (\forall k \in \{1, \ldots, M\}\ \mathit{Box}(a_j) = \mathit{Box}(a_k) \Rightarrow a_k \neq 0) \qquad (2)$$

where $Box(a_j)$ is a function which returns the identifier of the box which contains the article associated with the variable a_j.

The following constraint, instead, states that if an article is associated with an order, then that order must have a destination greater than 0

$$\forall j \in \{1,\ldots,M\}\ (a_j = i, i > 0) \Rightarrow o_i > 0 \tag{3}$$

Order Constraints The following constraint states that is not possible for two distinct order to share the same destination if they will be both satisfied

$$\forall i \in \{1,\ldots,K\}\ o_i > 0 \Rightarrow \forall k \in \{1,\ldots,K\}\ o_i \neq o_k \tag{4}$$

This constraint was implemented by imposing an `atmost/3` constraint to the order variables for each possible destination

```
for(I, 1, NDest),
   param(OrderDestinations)
 do
   atmost(1, OrderDestinations, I)
```

where `NDest` is the number of possible destinations N, i.e. the possible outputs of the sorter, and `OrderDestinations` is the list of the order variables.

If the i-th order must be satisfied ($o_i > 0$), then enough food product must be available

$$\forall i \in \{1,\ldots,K\}\ o_i > 0 \Rightarrow \sum_{a_j = i} Quantity(a_j) \geq Quantity(o_i) \tag{5}$$

where $Quantity(\cdot)$ is a function which returns the quantity of food product contained (requested) in an article (by an order).

If there is not enough food product to satisfy an order, then the order is unsatisfiable and for sure its destination will be "stay"

$$\forall i \in \{1,\ldots,K\} \sum_{Compatible(a_j,o_i)} Quantity(a_j) < Quantity(o_i) \Rightarrow o_i = 0 \tag{6}$$

The following constraint fixes an upper limit of food product that must be used to satisfy an order. It states that the sum of the quantities of food product contained in the articles used to satisfy an order must not exceed the quantity requested by an order plus a surplus quantity of food.

$$\forall i \in \{1,\ldots,K\}\ o_i > 0 \Rightarrow \sum_{a_j = i} Quantity(a_j) \leq Quantity(o_i) + Surplus \tag{7}$$

In our model we fixed the surplus quantity to 5 kg.

Symmetry Breaking Constraints In order to improve the search, we prune the search space by imposing the following "symmetry breaking" constraints which remove equivalent or symmetric solutions.

Example 2 (Equivalent solutions). Consider the case where we have three order variables: o_1, o_2 and o_3 and the number of regular outputs of the Sorter is N. Suppose there exists a feasible solution

$$\{o_1 = 1, o_2 = 0, o_3 = 0\}$$

Since we can associate an order with any output of the Sorter, other solutions could be

$$\{o_1 = 2, o_2 = 0, o_3 = 0\} \ \ldots \ \{o_1 = N, o_2 = 0, o_3 = 0\}$$

We have N equivalent solutions.

To prune the search space and avoid situations like the one described in Example 2, we impose that the maximum destination value that an order variable can have is the number of orders that are going to be satisfied (i.e. the number of order variables greater than 0)

$$\sum_{o_i > 0} 1 = S = \max_i o_i \tag{8}$$

Example 3 (Symmetric solutions). Consider the case where we have three order variables: o_1, o_2 and o_3 and the number of regular outputs of the Sorter is N and constraint 8 has been added in the model. Suppose there exists a feasible solution

$$\{o_1 = 1, o_2 = 2, o_3 = 0\}$$

A solution symmetric w.r.t. the one above is

$$\{o_1 = 2, o_2 = 1, o_3 = 0\}$$

Symmetries like the one described in Example 3 can be broken by imposing a total order to the order variables

$$\forall i, j \in 1, \ldots, K \ i < j, o_i > 0, o_j > 0 \Rightarrow o_i < o_j \tag{9}$$

This constraint can be simply imposed by exploiting the global constraint `precede/3`

```
numlist(1, NDest, DestinationList),
precede(DestinationList, OrderDestinations).
```

where `NDest` is the number of possible destinations.

5.3 Objective function

The aim of the DSS user is maximize the earnings and minimize the costs. In particular the user wants to maximize the incomes obtained by satisfying the orders, while minimizing the storage costs, the penalties that must be paid if the orders are not delivered on time and the processing cost, i.e. the cost for using machinery for sorting, unpacking and food processing. So far, to compute the processing cost, only the sorting cost is taken into account.

In order to obtain an optimal solution, we define in our model the following objective function that should be maximized

$$\textsc{Profit} = \sum_{o_i > 0} \mathit{Income}(o_i) - \textsc{UnusedArticlesCost} \\ - \textsc{LatePenalty} - \textsc{SortingPenalty} \tag{10}$$

where the function $\mathit{Income}(o_i)$ returns the income obtainable by satisfying the i-th order. $\textsc{UnusedArticlesCost}$ is the cost of keeping the unused articles one more day in the storage and it is defined as

$$\textsc{UnusedArticlesCost} = \sum_{a_j \leq 0} \mathit{StorageCost}(a_j) \tag{11}$$

where $\mathit{StorageCost}(a_j)$ is a function which returns the daily storage cost of the j-th article.

$\textsc{LatePenalty}$ is the cost of not satisfying not even today those order whose deadlines were missed and it is defined as

$$\textsc{LatePenalty} = \sum_{o_i \in \mathit{LateOrders}} \mathit{LateOrderPenalty}(o_i) \cdot \mathit{DaysLate}(o_i) \tag{12}$$

where $\mathit{LateOrders}$ is the set of orders with expired deadline, $\mathit{LateOrderPenalty}(o_i)$ is a function which returns the penalty for being one day late to satisfy the i-th order and $\mathit{DaysLate}(o_i)$ returns the number of delay days.

$\textsc{SortingPenalty}$ is a penalty for all those articles that must "go back" and therefore should go through the sorter machine once again. So far, this is the only cost taken into account. It is defined as

$$\sum_{a_j = -1} \mathit{ArticleSortingPenalty}(a_j) \tag{13}$$

where $\mathit{ArticleSortingPenalty}(a_j)$ returns the penalty of sorting once more the j-th article.

5.4 Search

To find the optimal solution we use the predicate `bb_min/3`, which performs a branch-and-bound algorithm. We separated the search for article and order variables, in order to allow the use of different heuristics for the two lists of variables.

```
bb_min(
  (gfd_search:search(OrderDestinations, 0,
      SelectOrderVars, ChoiceOrderVars, MethodOrderVars
      , []),
   gfd_search:search(ArticleDestinations, 0,
       SelectArticleVars, ChoiceArticleVars,
       MethodArticleVars, []) ),
  Cost,
  bb_options{timeout:Timeout, delta:Delta, strategy:
      Strategy} )
```

In the two predicates `gfd_search:search/6`, `SelectOrderVars` and `SelectArticleVars` are the variable selection methods for orders and articles respectively; `ChoiceOrderVars` and `ChoiceArticleVars` are the choice methods; `MethodOrderVars` and `MethodArticleVars` are the search methods. Among the available options of `bb_min` we have `Timeout`, that is the timeout of the optimization process (its default value is 0, i.e. infinite time), `Delta`, which is the minimal absolute improvement of `Cost` required for each step (default value is 10000), and `Strategy`, which is the optimization strategy used to find the best solution.

In order to improve the performances of the search, we developed a custom variable selection method, called `select_income`. Considering that the system sorts the list of orders in descending order by profit, if this heuristics is chosen for order variables, the order with the least value greater than zero is selected, which corresponds to the order with the greatest income. If this heuristics is chosen for article variables, the article that can be used for the most profitable order is selected.

```
select_income(Var, Elem) :-
    get_domain_as_list(Var, Domain),
    first_gt_zero(Domain, Elem).
```

`first_gt_zero/2` unifies `Elem` with the first value in the domain greater than 0, if there is not such values it unifies `Elem` with 0.

6 Experiments

A prototypical plant is under development in the SORT project. In the meanwhile, since real data about orders and article is still not available, to test the system we generated synthetic random datasets with 30 orders whose incomes vary from 10,000 to 500,000 and an increasing number of articles. The size of the datasets is expressed in number of articles. The quantity of food product of each article randomly varies from 500 g to 5 kg. Whereas the quantity of requested feed material for each order varies from 50 kg to 100 kg.

We performed two types of experiments. In the first a complete search in the space of solutions is performed, by taking into account different configurations and strategies. In the second we set a timeout of 15 seconds and the obtained solutions are compared with the optimal solutions found with complete search. For all the experiments we set the sorting penalty to 0 and in the predicate `bb_min/3` we set the option `delta` to 10,000.

In these tests we evaluated our system by combining several heuristics and strategies for search and optimization. However, due space limitations it was impossible to show the results for each heuristics combination. Table 1 shows some of the heuristics combinations used in our tests whose results are reported in this paper[7].

Table 1. Subset of the possible combinations of heuristics and strategies for search and optimization.

Name	Article Select	Article Choice	Order Select	Order Choice	Opt. strategy
Comb. 1	`select_income`	`indomain_max`	`select_income`	`indomain_max`	`continue`
Comb. 2	`select_income`	`indomain_max`	`input_order`	`indomain_max`	`continue`
Comb. 3	`select_income`	`indomain_max`	`select_income`	`indomain_max`	`dichotomic`
Comb. 4	`select_income`	`indomain_max`	`input_order`	`indomain_max`	`dichotomic`
Comb. 5	`input_order`	`indomain(0)`	`most_constrained`	`indomain_min`	`continue`

All the tests were performed on the HPC System Marconi[8] equipped with Intel Xeon E5-2697 v4 (Broadwell) @ 2.30 GHz, using 1 core for each test.

Test 1: complete search We consider a number of articles from 100 to 400 in steps of 100 and, for each number of articles, we generated 5 datasets.

Table 2 reports the average CPU time in seconds for finding the optimal solution for 5 different random datasets with the combinations of heuristics illustrated in Table 1. We set a timeout of 1 hour, so the cells with "–" indicate that the timeout occurred at least in one dataset.

Table 2. Average CPU time (in seconds) for the search of the optimal solution with different configurations (Test 1) for each dataset size. "–" means that the execution timed out (1 h) in one dataset at least.

Heuristics	100	200	300	400
Comb. 1	0.154	0.38	1.024	1.726
Comb. 2	0.146	0.372	1.046	1.862
Comb. 3	0.126	0.328	0.980	–
Comb. 4	0.132	0.416	0.994	–
Comb. 5	0.160	–	–	–

The results showed that the adopted heuristics affect the computational time. In particular, among all the heuristics combinations used for Test 1 only Comb. 1 and 2 were able to scale up to 500 articles and both combinations use the custom variable selection method `select_income` for the article variables.

[7] The results for other combinations are shown here: `https://goo.gl/J4DWiu`.
[8] `http://www.hpc.cineca.it/hardware/marconi`

Test 2: search with timeout It might be infeasible for the user to wait too long for the best solution. Sometimes a solution which is suboptimal but obtained quickly is the best thing for rapid decision making. To evaluate the behaviour of our system in these scenarios, for each dataset we run the CLP engine twice. In the first execution the optimal solution is found by performing a complete search by using Comb. 1 (Table 1) and a timeout of 24 hours. In the second one, instead, we set a timeout of 15 seconds and we bounded the backtracking search steps to 1000. Then the (suboptimal) solutions obtained in the second run are compared with the optimal solutions found with complete search in the first run.

We consider a number of articles from 500 to 1000 in steps of 100 and, for each number of articles, we generated 5 datasets.

The suboptimal solutions are evaluated using the Mean Absolute Percentage Error (MAPE) as metrics. The MAPE between the optimal solution obtained with a complete search ($S_i^{optimal}$) and the solution obtained with a search with timeout ($S_i^{timeout}$) is given by

$$MAPE = \frac{100}{N}\sum_{i=1}^{N}\left|\frac{S_i^{optimal} - S_i^{timeout}}{S_i^{optimal}}\right|$$

where N is the number of datasets in which the complete optimization search did not reach the timeout.

Table 3. MAPE between the optimal solutions and the suboptimal solutions obtained with limited search by adopting different configurations for each dataset size (Test 2). "–" means that the complete optimization search timed out (24 h) at least in one of the 5 datasets and it was impossible to perform comparisons.

Heuristics	**500**	**600**	**700**	**800**	**900**	**1000**
Comb. 1	0.59	0	–	–	–	–
Comb. 2	0.59	0	–	–	–	–
Comb. 3	1.03	0.76	–	–	–	–
Comb. 4	1.03	0.76	–	–	–	–

Table 3 shows that with datasets containing more than 600 articles the complete optimization search becomes infeasible, whereas with an approximate approach for each dataset we obtained at least a suboptimal solution.

7 Conclusions

In this paper we presented a DSS which exploits the Constraint Logic Programming paradigm and supports ontology reasoning for finding the optimal solution, given the availability of articles in the warehouse of the SORT plant.

The experiments showed that CLP is a valid approach for modelling and solving complex problems such as the SORT process.

In the future, we plan to integrate a DL reasoner in our DSS in order to tackle more complex scenarios (e.g. scenarios where orders can request mixtures of feed materials), complete the development of its web interface and test our DSS with real data on a prototype of the SORT plant. So far, in our model, every article has the same storage cost, however there are different type of warehouses (refrigerated, room temperature, etc.); in the future we plan to use different storage costs depending on the warehouse in which the article was stored. Moreover we will take into account costs for using machinery for unpacking and food processing. We would also like to extend our model by taking into account not only articles already in stock, but also those currently being transported to the SORT plant.

In our approach we used ECL^{i}PS^e for representing the constraints and Gecode to solve them, it could be interesting to encode our problem with MiniZinc [6] and compare the two approaches. A completely different approach could be to exploit (Mixed) Integer Programming methods: encode the problem as a linear problem and solve it with a linear optimiser, such as Gurobi [2]. We reserve the implementations of these different approaches as future work.

Acknowledgement This work was supported by the SORT project, promoted by the Italian Ministry of Education, Universities and Research. Code: SCN_00367, Ministerial Act n. 2427.

References

1. Baader, F., Horrocks, I., Sattler, U.: Description Logics, chap. 3, pp. 135–179. Elsevier, Amsterdam (2008)
2. Gurobi Optimization, L.: Gurobi optimizer reference manual (2018), `http://www.gurobi.com`
3. Gustavsson, J., Cederberg, C., Sonesson, U., van Otterdijk, R., Meybeck, A.: Global food losses and food waste: extent, causes and prevention (2011), `http://www.fao.org/docrep/014/mb060e/mb060e00.pdf`
4. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 111–119. ACM Press (1987)
5. Jaffar, J., Michaylov, S., Stuckey, P.J., Yap, R.H.: The clp (r) language and system. ACM T. Prog. Lang. Sys. **14**(3), 339–395 (1992)
6. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard cp modelling language. In: International Conference on Principles and Practice of Constraint Programming. pp. 529–543. Springer (2007)
7. Niederliński, A.: A Gentle Guide to Constraint Logic Programming via ECL^{i}PS^e. Jacek Skalmierski Computer Studio (2015)
8. Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Pearson (2016)
9. Stenmarck, Å., Jensen, C., Quested, T., Moates, G.: Estimates of european food waste levels. Tech. rep., IVL Swedish Environmental Research Institute (2016)
10. Team, G.: Gecode: Generic constraint development environment (2006), `http://www.gecode.org`
11. W3C: OWL 2 Web Ontology Language (12 2012), `http://www.w3.org/TR/2012/REC-owl2-overview-20121211/`