Declarative and Mathematical Programming approaches to Decision Support Systems for food recycling

(Article begins on next page)

24 April 2024

Declarative and Mathematical Programming Approaches to Decision Support Systems for Food Recycling

Federico Chesani[a], Giuseppe Cota[b,*], Marco Gavanelli[d], Evelina Lamma[d],
Paola Mello[a], Fabrizio Riguzzi[c]

[a]*Dipartimento di Informatica Scienza e Ingegneria – Università di Bologna*
Viale Risorgimento 2, 40136, Bologna, Italy
[b]*Dipartimento di Scienze Matematiche, Fisiche e Informatiche – Università di Parma*
Parco Area delle Scienze, 53/A, 43124, Parma, Italy
[c]*Dipartimento di Matematica e Informatica – Università di Ferrara*
Via Saragat 1, 44122, Ferrara, Italy
[d]*Dipartimento di Ingegneria – Università di Ferrara*
Via Saragat 1, 44122, Ferrara, Italy

Abstract

Every year about one third of the food production intended for humans gets lost or wasted. This wastefulness of resources leads to the emission of unnecessary greenhouse gas, contributing to global warming and climate change.

The solution proposed by the SORT project is to "recycle" the surplus of food by reconditioning it into animal feed or fuel for biogas/biomass power plants. In order to maximize the earnings and minimize the costs, several choices must be made during the reconditioning process. Given the extremely complex nature of the process, Decision Support Systems (DSSs) could be helpful to reduce the human effort in decision making.

In this paper, we present a DSS for food recycling developed using two approaches for finding the optimal solution: one based on Binary Linear Programming (BLP) and the other based on Answer Set Programming (ASP), which outperform our previous approach based on Constraint Logic Programming (CLP) on Finite Domains (CLP(FD)).

In particular, the BLP and the CLP(FD) approaches are developed in ECL^{i}PS^e, a Prolog system that interfaces with various state-of-the-art Mathematical and Constraint Programming solvers. The ASP approach, instead, is developed in `clingo`. The three approaches are compared on several synthetic datasets that simulate the operative conditions of the DSS.

---

*Corresponding author

Email addresses: `federico.chesani@unibo.it` (Federico Chesani),
`giuseppe.cota@unife.it` (Giuseppe Cota), `marco.gavanelli@unife.it` (Marco Gavanelli),
`evelina.lamma@unife.it` (Evelina Lamma), `paola.mello@unibo.it` (Paola Mello),
`fabrizio.riguzzi@unife.it` (Fabrizio Riguzzi)

1. Introduction

According to a study published by the Food and Agriculture Organization of the United Nations (FAO) in 2011, every year about one third of the global production of food for humans gets lost or wasted (Gustavsson et al., 2011).

Food loss refers to losses of edible food mass throughout all stages of the food chain and have heavy repercussions on ecosystem degradation and on human resource consumption.

The term *food waste* refers to food losses occurring at the last stages of the food chain (retail distribution and final consumption). These losses depend on the behaviour of retailers and final consumers. For instance, at retail level, large quantities of food are wasted due to appearance standards (e.g. dented but unbroken cans, vegetables with imperfections, etc.).

The solution envisioned by the SORT project[1] aims at "recycling" a significant part of food in excess or no longer consumable by humans, in order to reduce as much as possible the food waste destined for landfill. The project involved experts in production plants, veterinary, computer science, logistics, and it identified, as main possible uses of the wasted food, reconditioning into feed material for animals and production of biogas/biomass for sustainable energy production. Moreover, the packaging is also recovered for recycling.

The process envisioned by SORT is complex and involves several parties and actors, which can be split into four main categories:

- *Retail companies*, which gather the food in excess or no longer consumable by humans.

- *SORT plants*, which are responsible of the reconditioning process of the food in excess.

- *Customers*, which request orders to the SORT plants (animal feed factories and biogas/biomass power plants).

- *SORT logistics*, that part of the SORT project which is responsible of the delivery of orders from SORT plants to customers and of food articles from retail companies to SORT plants.

Figure 1 shows the entire SORT process and the relations between the different actors. The food articles to recycle are gathered at large retailers and collection centers and packaged into boxes with an RFID chip for identification and tracking. These boxes are temporarily stored in the warehouses of

---

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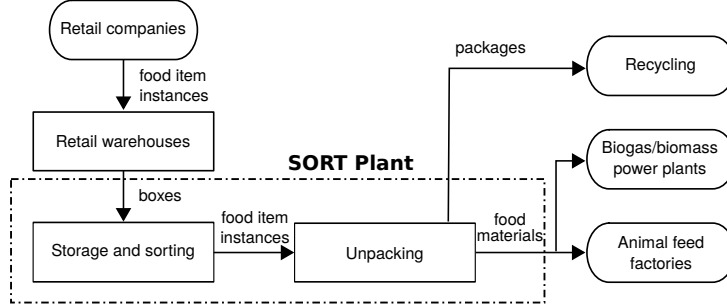
Figure 1: SORT process.

the retail companies, then the SORT logistics delivers the boxes from the retail warehouses to the so-called SORT plants.

A SORT plant stores the incoming boxes and sorts the food articles according to the received orders. The food articles are then unpacked and the feed material is sent to customers, while the gathered packages are recovered for recycling.

Feed factories and biogas/biomass power plants are the main customers of a SORT plant, each customer can make different orders of feed materials. The satisfied orders are the main source of income for a SORT plant. On the contrary, if an order was not satisfied and its deadline expired, the delay constitutes a cost in terms of penalties to pay.

The feed materials can be organized into an ontology, as we did in our previous work (Chesani et al., 2018). However, in this work this approach was abandoned. We reserve the definition of a complete ontology when it will be clear what kind of mixtures of feed materials can be requested by customers.

In order to be self-sustainable, the process must also produce economic profit, which is attainable if a careful choice is made for each of the available freedom degrees, and considering all the involved parameters (income of the orders, expiry dates, food articles to be reconditioned, penalties, storage costs, etc.). Making decisions in this complex environment can be very hard for a human.

In this paper we propose a Decision Support System (DSS) that can be used as aid by the process manager to make the best choices to maximize the profit and reduce the human effort. This paper is an upgrade and an extension of our previous work (Chesani et al., 2018) where we developed a DSS for the SORT process based on Constraint Logic Programming of Finite Domains (CLP(FD)) for finding the optimal solution. We developed two new approaches, both implemented in declarative programming languages: one is based on a mathematical programming model (in particular, Binary Linear Programming (BLP)), implemented in the Constraint Logic Programming language ECL^iPS^e (Schimpf and Shen, 2012), the other based on Answer Set Programming (ASP), a declarative language with solvers based on artificial intelligence techniques, and we used `clingo` (Gebser et al., 2012) as solver.

In order to find the most suitable of the proposed approaches, we experimen-

tally compare the three approaches on several synthetic datasets of increasing size that simulate the operative conditions of the DSS. The experimental results show that the new proposed approaches outperform the previous one based on CLP(FD). Thanks to the new formulations, we are now able to address problems of significant size. In particular, we could find provably optimal solutions even when considering a number of elements as large as half of the supposed size of the warehouse. In case larger sizes will be necessary, sub-optimal solutions are nevertheless provided[2].

Note also that in this work we only used open-source solvers, however there exist also commercial solvers, that usually provide solutions is shorter times.

The rest of the paper is structured as follows. The process chain of the SORT plant is described in Section 2. Section 3 illustrates the paradigms used for this DSS. In particular, it recalls the main concepts of CLP, BLP and ASP. Section 4 recaps our previous approach based on CLP(FD). Section 5 and Section 6 illustrate, respectively, the BLP approach and the ASP approach of the DSS. Section 7 shows experimental results on synthetic datasets. Section 8 concludes the paper.

2. SORT Plant Process Chain

Before illustrating this process, we must make a terminological distinction between *food item*, *instance of a food item* and *food article*. A *food item* is a good identified by a European Article Number (EAN), i.e. a barcode, that has a specific volume and weight, while an *instance of a food item* also has its own production and expiry dates. For example, a coffee product of a specific brand is a food item, while a specific bag of coffee of that brand produced on 13/05/2019 that expires on 20/06/2020 is an instance of a food item.

A set of instances of a food item that have (almost) the same expiry date can be considered as a single *food article*. For instance, a set of ten cans of beans with the same expiry date can be seen as a unique food article that has ten times the weight and the volume of a single can of beans. Using this approach, we reduce the number of variables in our declarative models.

Figure 2 shows the process chain of a SORT plant. The first phase of this process is called *input sorting*. The boxes that reach the SORT plant are opened and their contents are poured into a hopper. Then the instances of food items are sorted into other boxes by the Sorter machinery, in order to create food articles and, if possible, homogeneous boxes, i.e. boxes that contain similar food articles with close expiry dates. When a box is full or the maximum weight is reached, it is closed and then stored in an automated warehouse, which may contain up to 1000 boxes. The size of the stored boxes is $400 \times 600 \times 420$ mm and a single box cannot be heavier than 15 kg in order to be lifted by a human

---

[2]We call in this context *sub-optimal* all solutions for which it was impossible to prove optimality within the given timeout. Usually the proof of optimality takes more time than finding the optimal solution; for this reason, a *sub-optimal* solution might actually be optimal.

operator. These are not limits of our DSS but they are imposed by the design of the SORT plant and by Italian laws. It is worth noting that, during *input sorting*, the Sorter does not receive any directive by the DSS. The policies of the warehouse, such as efficiently allocating space, or filling as much as possible the boxes, are not the subject of this work, and are completely independent of the DSS. Indeed, the optimal allocation of warehouse space could be the subject of another work.

In the following phase of the process chain, called *output sorting*, the DSS assigns an output destination to each food article. The Sorter machinery has N regular output destinations plus two non-regular output destinations: *go back*, i.e. the article must go back to the warehouse, and *stay*, i.e. the article must stay in the warehouse without moving. The DSS can associate a regular output with an order requested by a customer. If some of the articles contained in a box are chosen by the DSS to satisfy an order, then that box will be opened and its items will be poured into the hopper again and, in accordance with the guidelines defined by the DSS, each article will be routed by the Sorter to one of the N regular output destinations or sent back to the warehouse (*go back* destination). At the end of each output destination (except for *stay*) there is a box that gathers all the sorted articles. When a box is full or the maximum weight is reached, it is substituted automatically with an empty one. The DSS does not need to take into account the composition of the actual boxes having a *go back* destination, but only decides if each item must *go back* or if it is used for some order. Also, the box substitution is automatic, and not handled by the DSS. The articles contained in closed boxes remain in the warehouse without moving (*stay* destination). The boxes that contain articles that must go back to the warehouse, instead, can be extremely heterogeneous; therefore their content must be poured again into the hopper, in order to repeat, for these articles, the input sorting phase. Finally, the boxes containing the articles that will be utilised to satisfy an order are sent to machinery for unpacking and food processing.

Example 1. *Consider the case where two orders, o_1 and o_2, were commissioned and that both of them are satisfiable. The DSS finds the best solution and associates order o_1 with the first output destination (out 1) and order o_2 with the second output destination (out 2) and, simultaneously, assigns some of the food articles contained in the boxes 5 and 17 to o_1 and some of the articles contained in boxes 17 and 131 to o_2. The boxes with identifiers 5, 17 and 131 are then picked from the warehouse, opened and their content is poured into the hopper. The Sorter then sends to out 1 the articles chosen to satisfy o_1 and to out 2 the ones chosen to satisfy o_2. The unused articles, which are the ones assigned to* go back, *should be sent back to the warehouse after repeating the input sorting phase.*

It is worth noting that, in the current SORT plant's design, there is only one sorting line. That means that the Sorter machinery depicted on the left of Figure 2 (input sorting phase) is exactly the same Sorter machinery on the right

(output sorting phase). Consequently, the Sorter is not able to execute input sorting and output sorting at the same time.
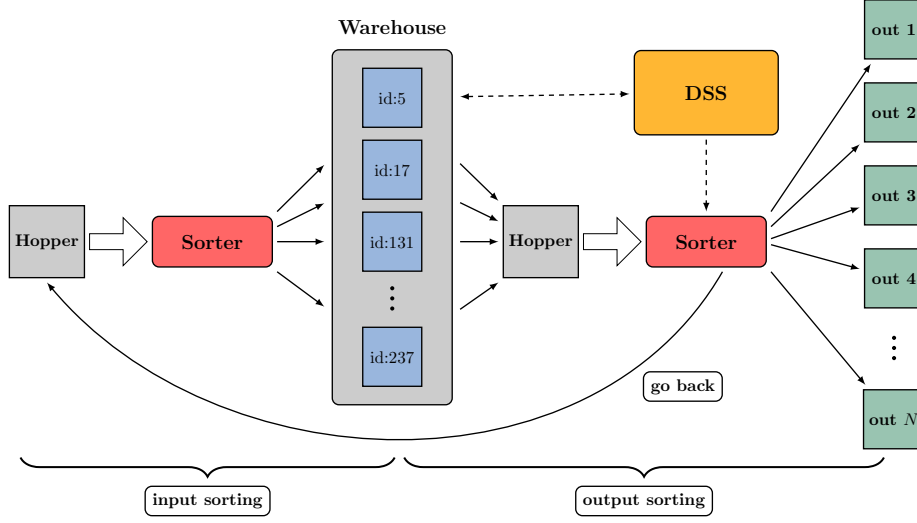
Figure 2: SORT chain. Each box in the warehouse has an RFID chip used for identification and tracking.

3. Background

The three techniques used in this work are not only state of the art techniques to solve the class of problems we are addressing, but they are also declarative approaches. This means that the most important part is modelling the problem in the devised language, and once the model of the problem is devised, the user is encouraged to use the solver as a black-box, since other issues (like tuning the parameters of the solver) are of secondary importance. For ASP and BLP, the producers of the solver provide very good default parameters, that have been carefully selected in order to provide, on average, the best possible performance, making the solver competitive with other solvers. For CLP(FD), in a previous work (Chesani et al., 2018) we did a systematic experimentation to find the combination of parameters providing the best performance.

3.1. Constraint Logic Programming

CLP is a set of logic programming languages that can deal with particular predicates named *constraints*. Constraints are not defined as all other predicates in the knowledge base of the program, but they are handled by an external solution algorithm, named constraint solver, that can simplify them and check for their consistency during the resolution of the program. There exist various CLP languages, each dealing with a specific domain, and featuring a constraint

solver. Notable examples of CLP languages are CLP($\mathcal{R}$), to deal with continuous variables and using the simplex algorithm to deal with linear constraints, and CLP(FD), that deals with variables ranging on finite domains.

In this work, we are mainly concerned with CLP(FD), with a solver aimed at solving the so-called Constraint Satisfaction Problems by using constraint propagation algorithms.

A Constraint Satisfaction Problem (CSP) involves a set of variables, that are considered as unknowns, each ranging on a finite set, called *domain*. A set of constraints are imposed on the variables, stating that not all combinations of values for the variables are feasible. For example, the constraint $A < B$ is satisfied by all pairs of values such that the value taken by variable A is less than the value of variable B. The aim is assigning to each variable one value taken from its domain in such a way that all constraints are satisfied: this is called a *solution* of the CSP.

In some cases, amongst all the solutions of a CSP one needs to find the best one, according to some objective function. This is called a *Constraint Optimization Problem*.

The solution algorithms employed in CLP(FD) are based on notions of constraint propagation with different levels of consistency of the constraints. Each constraint has a propagation algorithm, that is tailored to solving efficiently such constraint; such algorithms remove values inconsistent with the specific constraints, since they cannot be extended to solutions of the whole problem. The technologies used in CLP(FD) were later implemented also in non-logic languages, giving rise to the area of Constraint Programming. CLP(FD) is usually particularly effective in solving problems that employ *global constraints*, i.e., constraints that involve many variables (possibly, all variables in the problem) and for which there exist strong propagation algorithms.

3.2. Binary Linear Programming

Mathematical Programming (also known as mathematical/numerical optimisation) is a branch of operations research concerning the study of optimisation using mathematical techniques. A Mathematical Programming problem is defined by a set of variables, an objective function that must be minimised or maximised and a set of constraints, expressed as equalities and inequalities, on the variables.

Mathematical programming is divided in several problem classes, such as Linear Programming (LP), quadratic programming, convex programming, etc. In particular, in LP the objective function and the constraints are linear and the variables are continuous. If the variables are required to be integers, then the problem is in the Integer Linear Programming (ILP) class. Binary Linear Programming (BLP) is a subclass of ILP where all the variables are binary-valued.

A BLP problem is expressed as:

$$\begin{aligned} \text{Maximize} \quad & \mathbf{c}^T \mathbf{x} \\ \text{such that} \quad & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \\ \text{and} \quad & \mathbf{x} \in \{0, 1\} \end{aligned} \tag{1}$$

The usual solution technique is based on the linear relaxation of the BLP problem, in which the integrality constraint $\mathbf{x} \in \{0, 1\}$ is relaxed, requiring each variable to belong to the interval $[0, 1]$: $\mathbf{x} \in [0, 1]$; in fact, if all variables are allowed to take real values, the relaxed problem can be solved to optimality in polynomial time. If in the optimal solution of the linear relaxation all variables have integer values, the problem is solved. Otherwise, the value of the objective function of the optimal linear relaxation provides a valid upper bound to the optimal solution of the BLP: no integer solution can have a better objective function value.

Two main techniques can be used to start a search for the optimal integer solution. One is named *cutting planes*, and consists in adding further linear constraints that eliminate fractional solutions without removing any integer solution. The other, named *branch-and-bound*, consists in selecting one variable x_i with non-integer value and creating two sub-problems: one in which $x_i = 0$ and the other in which $x_i = 1$. Again, the two sub-problems are solved recursively. In particular, their linear relaxation provides an upper bound to the optimal solution of the two sub-problems, and if one of these is worse than any previously obtained solution, the corresponding sub-problem can be discarded without the need to fully solve it.

These techniques are usually very effective if the problem is naturally modelled with linear constraints, in particular when the number of linear constraints in the linear formulation is not exceedingly high. Also, since it relies heavily on the linear relaxation, BLP is able to focus directly on the best solutions, so it is a very effective technique for finding the optimum.

As in our previous CLP(FD) program (Chesani et al., 2018) (described briefly in Section 4), we adopted ECL^{i}PS^e for modelling the BLP problem. ECL^{i}PS^e provides a library, called `eplex` (Shen and Schimpf, 2005), which allows the user to model a Mathematical Programming problem in ECL^{i}PS^e, and then solve the problem by using external Mathematical Programming solvers.

In our tests we used COIN-OR CBC (Saltzman, 2002) as Mathematical Programming solver because it is open source, but other (probably faster) commercial solvers, such as Gurobi (Gurobi Optimization, LLC, 2018), could be used.

3.3. Answer Set Programming

Amongst the logic languages, a recent proposal is ASP. While the most widely used logic programming language, Prolog, includes language structures that are not declarative (such as the *cut* to commit to some branching decisions,

or assert/retract that can be used to create a form of destructive assignment), ASP rejects all possible sources of non-declarativity and is, therefore, a pure logic language. There exist very efficient ASP solvers (Simons et al., 2002; Lin and Zhao, 2004; Giunchiglia et al., 2006; Leone et al., 2006; Gebser et al., 2011), some capable of obtaining efficiency comparable with that of SAT solvers (as they won some SAT competitions); on the other hand the logic program is usually *grounded* before the solving activity starts, and the ground program may be very large. An ASP solver is able to find the stable models (Gelfond and Lifschitz, 1988) of a logic program, also known as *answer sets*.

In ASP, a program is a set of clauses in the form

$$h_1, h_2, \ldots h_n \text{ :- } p_1, p_2, \ldots p_n.$$

where the head $h_1, h_2, \ldots h_n$ is a disjunction of atoms and the body $p_1, \ldots, p_n$ is a conjunction of literals. Each literal is either an atom or the negation of an atom. A clause with empty body is called a *fact*, while a clause with empty head is named *Integrity Constraint (IC)*, and its body must be false in each answer set.

Modern ASP solvers extend the language of programs with a number of features; we list the most relevant ones for this article.

Choice rules. A choice rule has the syntax

$$\{h\} \text{ :- } Body.$$

and the meaning is that if the *Body* is true, then the atom in curly brackets h can be chosen to be true, i.e., the solver can choose whether it is true or false.

The syntax of choice rules can be extended to cardinality rules of the form

$$l \{h\} u \text{ :- } Body.$$

which state that the number of true atoms matching h is between a lower bound l and an upper bound u.

Conditions. A condition has syntax $a(X) : b(X)$ where a and b are atoms. It is expanded to the set of atoms $\{a(X)|b(X)\}$, i.e., the set of atoms $a(X)$ such that $b(X)$ is true. If a condition occurs in the body of a clause, the atoms in the set are considered in conjunction; if it is in the head, they are considered in disjunction.

Aggregates. ASP features aggregates of sum `#sum`, maximum `#max`, minimum `#min` etc. An aggregate `#agg` can only occur in an atom

$$\texttt{\#agg}\{t_1, t_2, \ldots, t_n : Goal\} \, Op \, Expr,$$

where $t_1, t_2, \ldots, t_n$ is a tuple of terms, *Goal* is a conjunction of literals, *Op* is a relational operator ($=, <, \leq, \ldots$), and *Expr* is an expression. The meaning is that the set of tuples $\{t_1, t_2, \ldots, t_n : Goal\}$ making *Goal* true is considered; on such

set, the first term t_1 is the term subject of aggregation and the value obtained is compared through Op with $Expr$. For example, $\texttt{\#sum}\{A, B : p(A, B)\} \geq 3$ means

$$\sum_{(A,B) \in \{A,B:p(A,B)\}} A \geq 3.$$

A number of ASP solvers have been developed (Simons et al., 2002; Lin and Zhao, 2004; Giunchiglia et al., 2006; Leone et al., 2006; Gebser et al., 2011); the usual solution technique is to ground the program, i.e., substitute each variable in all possible ways (with all its possible instantiations). For this reason, syntactic restrictions are imposed on the language in order to ensure that the ground program is always finite. Still, the ground program is typically exponentially larger than the original program. ASP solvers employ very advanced search strategies to find quickly a feasible solution, i.e., a solution satisfying all constraints, and rely often on technologies similar to those used in SAT solvers, such as unit propagation, conflict directed clause learning, restarts. ASP solvers are typically very effective at dealing with problems in which the satisfaction component is very hard, i.e., when the hardest part is finding a feasible solution, while they are usually less effective than mathematical programming techniques at advancing directly to the optimal solutions.

3.3.1. Mapping Binary Linear Programming to Answer Set Programming

The BLP problem illustrated in Eq. 1 can be translated in the following ASP program.

```
:- #sum{ Val_ij, J : x(J), a(I, J, Val_ij) } > Bi,
   b(I, Bi),
   row(I).
{ x(J) } :- column(J).
column(1..N) :- nvar(N).
row(1..M) :- nrows(M).
```

where predicate `x(J)` represents the j-th binary variable in vector $\mathbf{x}$, predicates `nvar/1` and `nrows/1` return, respectively, the number of variables, i.e. the number of elements in $\mathbf{x}$, and the number of rows in matrix $\mathbf{A}$. Predicate `b(I, Bi)` unifies `Bi` with the i-th element of vector $\mathbf{b}$, while `a(I, J, Val_ij)` unifies `Val_ij` with the element a_{ij} of $\mathbf{A}$.

This program shows that ASP as expressive as BLP. However, it is worth noticing that this direct transformation could lead to inefficient ASP programs, therefore, a reformulation of the problem in ASP is usually more advisable.

3.4. Comparison of solvers

CLP(FD), ASP and BLP can be considered three modelling and solution techniques for constrained optimization problems.

From a computational complexity point of view, CLP(FD) is a Turing-complete programming language, while ASP typically addresses problems up to the second level of the polynomial hierarchy (although there exist extensions

that make it Turing-complete) and BLP addresses NP-hard problems whose satisfaction component is NP-complete.

This implies that any NP-complete problem can be formulated into these three formalisms, and that it is always possible to automatically reformulate an NP-complete problem given in one formulation to the other two; in Section 3.3.1 we showed one example.

On the other hand, automatic reformulation techniques in general cannot capture many of the practical aspects. A same problem can be modelled with different CLP(FD) formulations (and with different ASP or BLP formulations), and some of them will be solved efficiently, as they exploit the strengths of the chosen solver, while others may slow down the solution.

It is often impossible to select a-priori the best solver for a given application, however some general rules of thumb can be given.

CLP(FD) leverages on strong constraint propagation of global constraints, so it will be effective when the problem is formulated with many efficient global constraints. Global constraints reduce, during search, the domains of the variables, effectively pruning the search tree. These are look-forward techniques, tailored for many different global constraints, that strive to reduce as much as possible the search space just after an assignment is made. Often, the user is provided with many different search strategies, and (s)he has to select one or implement a new one tailored for the problem to be addressed.

ASP solvers are more focussed on look-back techniques, that try to understand the reasons of a failure and learn new constraints from them. They exploit very efficient search strategies, based on clause learning and restarts. The idea in ASP is that the user should not delve into selecting a search strategy or define tailored search strategies (although (s)he can), but use the default search strategies of the solver, that work well, on average, in a variety of problems. One drawback is that the grounding phase expands exponentially the program written by the user, and it might reach the memory limits of the machine, or become so large that the solution of such large programs might be overwhelming.

BLP, and, in general, mathematical programming formulations, are based on the vast literacy on mathematical optimization, that is extremely efficient to guide the search toward the optimum, in particular when most of the constraints are naturally formalized as linear constraints. On the other hand, if the variables take only integer values and the linear formulation is cumbersome or the linear relaxation of the problem provides a bound that is uninformative, then such techniques are less effective, and tend to enumerate large parts of the search space, since they cannot use non-linear constraints to efficiently prune the search space.

In general, it is very hard to know a-priori which technique will perform best, and one possible strategy is to perform experimental evaluation.

It is worth noting that all these techniques are *complete*, meaning that they can find the optimal solution in finite time and prove that it is the real optimum, i.e., that no better solution can exist. Thus, all these techniques provide the same optimal solution, or solutions of exactly the same quality (unless they are terminated after some timeout). There exist other techniques, such as local

search or population-based methods, that are used for very large problems, that usually can find quickly a reasonable solution but cannot prove that such solution is optimal, and will loop forever if requested to find the real optimum.

4. CLP(FD) Model

In this section we recap the CLP(FD) model of the DSS introduced by Chesani et al. (2018).

4.1. Variables

In the CLP(FD) constraint model, we have a list of decision variables associated to orders $\mathbf{o} = \{o_1, \ldots, o_K\}$ and a list of variables associated to food articles $\mathbf{a} = \{a_1, \ldots, a_M\}$. The domain of an order variable is

$$\forall j \in \{1, \ldots, K\} \ o_j \in \{0, 1\}$$

where 1 means that the order will be satisfied and 0 that it will be not. A posteriori, i.e. after a solution is found, the DSS assigns each order to an output destination. Order variables are sorted in descending order by income.

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

$$\forall i \in \{1, \ldots, M\} \ a_i \in \{-1, \ldots, K\}$$

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

4.2. Constraints

We split the constraints into two categories: article constraints and order constraints.

4.2.1. Article Constraints

The first article constraint simply states that articles cannot be associated with incompatible orders

$$\neg Compatible(i, j) \Rightarrow a_i \neq j \qquad \forall i \in \{1, \ldots, M\}, \forall j \in \{1, \ldots, K\} \qquad (2)$$

where $Compatible(i, j)$ is true if the i-th article belongs to a feed category compatible with the feed category requested by the j-th order.

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. Predicate *Compatible* checks if the feed category of the order matches with the feed category of the article. However, in the future, for more complex scenarios, we plan to use ontological reasoning for compatibility checking.

The following constraint preserves the consistency of the boxes contained in the warehouse. Once a box is opened, all of its contents are poured into the

hopper and all the items contained in that box must be sorted by the Sorter (note that one possible destination is to go back to stock). It is not possible to have that an article in a box goes through the Sorter while the other articles contained in the same box remain in storage:

$$a_i \neq 0 \Rightarrow a_k \neq 0 \quad \forall i \in \{1, \ldots, M\}, \forall k \in ArticlesBox(Box(i)) \smallsetminus \{i\} \quad (3)$$

where function $Box(i)$ returns the index of the box containing the i-th article and function $ArticlesBox(l)$ returns the indexes of the articles that are contained in the l-th box.

Suppose that we are using L boxes. The following constraint states that if none of the articles contained in a box is used to satisfy an order, then the box must not be opened and all the articles contained in that box must stay in the warehouse without moving. This constraint avoids situations in which the box is opened and all of its content goes back.

$$\nexists i \in ArticlesBox(l) \ s.t. \ a_i > 0 \Rightarrow \forall k \in ArticlesBox(l) \ a_k = 0 \quad \forall l \in \{1, \ldots, L\} \quad (4)$$

If there is an article associated to an order, then that order must be completely satisfied:

$$a_i = j, j > 0 \Rightarrow o_j = 1 \qquad \forall i \in \{1, \ldots, M\} \qquad (5)$$

4.2.2. Order Constraints

At most N orders can be satisfied, where N is the number of regular outputs:

$$\sum_{j=1}^{K} o_j \leq N \qquad (6)$$

This constraint was implemented by simply imposing an `atmost/3` constraint to the order variables.

If the j-th order must be satisfied ($o_j = 1$), then enough food product must be available

$$o_j = 1 \Rightarrow \sum_{a_i=j} Quantity^a(i) \geq Quantity^o(j) \qquad \forall j \in \{1, \ldots, K\} \quad (7)$$

where function $Quantity^a(\cdot)$ (respectively, $Quantity^o(\cdot)$) returns the quantity of food product contained (requested) in an article (by an order). In our implementation, to iterate the sum over the list of article variables that have value equal to j, we execute the following steps. First, we extract the list of compatible articles, then this list is associated with a list of boolean variables. Each boolean variable has value 1 if the corresponding compatible article is used to satisfy the order and 0 otherwise. The summation above is implemented as a scalar product of the compatible article quantities and the list of the corresponding boolean variables.

13

In order to avoid using more food product than necessary for satisfying an order, 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 that order plus a surplus quantity of food.

$$o_j = 1 \Rightarrow \sum_{a_i = j} Quantity^a(i) \leq Quantity^o(j) + Surplus \quad \forall j \in \{1, \ldots, K\} \quad (8)$$

In the experiments, we fixed the *Surplus* to 5 kg.

If there is not enough food product to satisfy an order, then the order is unsatisfiable

$$\sum_{Compatible(i,j)} Quantity^a(i) < Quantity^o(j) \Rightarrow o_j = 0 \quad \forall j \in \{1, \ldots, K\} \quad (9)$$

4.3. 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.

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

$$\text{PROFIT} = \sum_{j=1}^{K} o_j \cdot Income(j) - \text{UNUSEDARTICLESCOST}$$
$$- \text{LATEPENALTY} - \text{SORTINGPENALTY} \quad (10)$$

where the function *Income*(*j*) returns the income obtainable by satisfying the *j*-th order. UNUSEDARTICLESCOST is the cost of keeping the unused articles one more day in the storage and it is defined as

$$\text{UNUSEDARTICLESCOST} = \sum_{a_i \leq 0} StorageCost(i) \quad (11)$$

where *StorageCost*(*i*) is a function which returns the daily storage cost of the *i*-th article, which is computed by multiplying the number of kilograms of the *i*-th article with a constant value of storage cost.

LATEPENALTY is the cost of not satisfying not even today those orders whose deadlines were missed, plus the cost of the accumulated delay:

$$\text{LATEPENALTY} = \sum_{j=1}^{K} LatePenalty(j) \cdot DaysLate(j) \cdot (1 - o_j) \quad (12)$$

where *LatePenalty*(*j*) is the penalty for being one day late in satisfying the *j*-th order and *DaysLate*(*j*) is the number of days already passed since the deadline expired, including the current day. The later an order is, the higher the cost.

SORTINGPENALTY is a penalty for all those articles that must "go back" and therefore should go through the sorter machine once again. It is defined as

$$\text{SORTINGPENALTY} = \sum_{a_i=-1} ArticleSortingPenalty(i) \qquad (13)$$

where $ArticleSortingPenalty(i)$ is the penalty of sorting once more the i-th article, which is computed by multiplying the number of aggregated food item instances represented by the i-th article with a constant value of sorting cost.

5. Binary Linear Programming approach

In this section we illustrate the BLP formulation of the DSS constraints.

5.1. Input data

The input data is provided by means of the following predicates:

```
order(Order_id, Feed_category, Deadline, Quantity, Income).
```

provides the data about the orders, predicate

```
item(Ean, Quantity, Volume, Feed_category).
```

identifies the food items that entered the SORT system, while predicate

```
article(Article_id, Ean, Box_id, Expiry, Storage_date,
↪   Feed_category, Quantity).
```

is about the food articles, where each article represent a set of instances of a particular food item that have the same expiry date.

5.2. Variables

The variables of the BLP model are:

- The list of order variables $\mathbf{o} = \{o_1 \ldots, o_K\}$, where K is the number of orders that the SORT plant received from clients.

- The matrix $\mathbf{A}$ (of size $M \times C$) of article variables, where M (number of rows) is the number of articles, and the number of columns C is the number of orders K plus the number of non-regular outputs (*stay* and *go back*), i.e. $C = K + 2$. The index of the columns of $\mathbf{A}$ varies from -1 to K.

- The list of box variables $\mathbf{b} = \{b_1, \ldots, b_L\}$, where L is the number of boxes.

15

The domain of an order variable is

$$\forall j \in \{1, \ldots, K\} \; o_j \in \{0, 1\}$$

where 1 means that the order will be satisfied and 0 that it will be not.

The elements of the matrix of article variables are boolean variables

$$\forall i \in \{1, \ldots, M\} \; \forall j \in \{-1, \ldots, K\} \; a_{ij} \in \{0, 1\}.$$

In particular, $a_{ij} = 1$ means that the i-th article

- will be used to satisfy the j-th order if $j \geq 1$,

- must stay in the warehouse (without opening its box) if $j = 0$,

- will go back to the warehouse (after its box was opened) if $j = -1$.

There is also a boolean variable associated with each box

$$\forall l \in \{1, \ldots, L\} \; b_l \in \{0, 1\}$$

where $b_l = 1$ means that the l-th box will be opened.

Each box b_l contains a set of articles; we define a function $Box(i)$ that returns the index of the box that contains the i-th article.

5.3. Constraints

We can split the constraints into two categories: article constraints and order constraints.

5.3.1. Article Constraints

The first article constraint states that an article can be associated only with one destination

$$\sum_{j=-1}^{K} a_{ij} = 1 \qquad\qquad \forall i \in \{1, \ldots, M\} \qquad (14)$$

The second article constraint simply states that articles cannot be associated with incompatible orders

$$\neg Compatible(i, j) \Rightarrow a_{ij} = 0 \qquad \forall i \in \{1, \ldots, M\}, \forall j \in \{1, \ldots, K\} \qquad (15)$$

where $Compatible(i, j)$ is true if the i-th article belongs to a feed category compatible with the feed category requested by the j-th order.

To preserve the consistency of the boxes contained in the warehouse, the following constraints are imposed

$$a_{ij} \leq b_l \qquad l = Box(i), \forall i \in \{1, \ldots, M\}, \forall j \in \{-1, 1, \ldots, C\} \qquad (16)$$
$$a_{i0} + b_l \leq 1 \qquad l = Box(i), \forall i \in \{1, \ldots, M\} \qquad (17)$$

16

Constraint 16 states that if the i-th article is associated with an order or must go back, then the box that contains it must be opened. Instead, 17 states that it is not possible that the i-th article must *stay* and that the box that contains it is opened.

Moreover, we want to avoid situations where the box is opened and all its content goes back. This can be done by imposing the following constraint

$$\sum_{i \in ArticlesBox(l)} a_{i,-1} \leq |ArticlesBox(l)| - 1 \qquad \forall l \in \{1, \ldots, L\} \qquad (18)$$

where $ArticlesBox(l)$ is the set of the indices of the articles that are contained into the l-th box.

The following constraint states that if an article i is used to satisfy an order j, then that order must be satisfied

$$a_{ij} \leq o_j \qquad \forall i \in \{1, \ldots, M\}, \forall j \in \{1, \ldots, K\}. \qquad (19)$$

5.3.2. Order Constraints

The following constraint states that at most N orders can be satisfied, where N is the number of regular outputs

$$\sum_{j=1}^{K} o_j \leq N \qquad (20)$$

The following constraint states that an order cannot be satisfied if not enough food is assigned to such order

$$\sum_{\substack{i=1 \\ Compatible(i,j)}}^{M} Quantity^a(i) \cdot a_{ij} \geq o_j \cdot Quantity^o(j) \quad \forall j \in \{1, \ldots, K\} \qquad (21)$$

The following constraint fixes an upper limit to the quantity of food product that can be used to satisfy an order

$$\sum_{\substack{i=1 \\ Compatible(i,j)}}^{M} Quantity^a(i) \cdot a_{ij} \leq o_j \cdot Quantity^o(j) + \text{SURPLUS} \quad \forall j \in \{1, \ldots, K\}.$$

$$(22)$$

If the amout of compatible food requested by an order is not available in the warehouse, then that order is unsatisfiable

$$\sum_{Compatible(i,j)} Quantity^a(i) < Quantity^o(j) \Rightarrow o_j = 0 \quad \forall j \in \{1, \ldots, K\} \qquad (23)$$

5.4. Objective Function

The objective function is the same as for the CLP(FD) approach:

$$\text{PROFIT} = \sum_{j=1}^{K} o_j \cdot Income(j) - \text{UNUSEDARTICLESCOST}$$
$$- \text{LATEPENALTY} - \text{SORTINGPENALTY} \tag{24}$$

In this case, the cost UNUSEDARTICLESCOST of keeping the unused articles one more day in the storage is defined as

$$\text{UNUSEDARTICLESCOST} = \sum_{i=1}^{M} (a_{i1} + a_{i2}) \cdot StorageCost(i) \tag{25}$$

where $StorageCost(i)$ returns the daily storage cost of the i-th article.

LATEPENALTY is defined as in the CLP(FD) case.

SORTINGCOST is a cost for all those articles that must "go back" and therefore should go through the sorter machine once again. It is defined as

$$\text{SORTINGCOST} = \sum_{i=1}^{M} a_{i1} \cdot ArticleSortingCost(i) \tag{26}$$

where $ArticleSortingCost(i)$ returns the cost of sorting once more the i-th article.

6. Answer Set Programming approach

In the ASP formulation of a problem, the solution is encoded in the model of the logic program. Below, we develop a program providing the solution as a model. The encoding of the input data is the same as in subsection 5.1.

6.1. Model Generation and Integrity Constraints

One of the main decisions is to select the orders that will be satisfied; we define a predicate `satisfied_order(Order_id,Income)` that is true for each order that must be satisfied. Such predicate is in the generation part of the program, so that the solver is allowed *"to guess"* its truth value. More precisely, for each order such that `order(Order_id,_,_,_,Income)` is true, the solver can decide whether the corresponding atom `satisfied_order(Order_id, Income)` is true:

```
0 { satisfied_order(Order_id,Income):
↪   order(Order_id,_,_,_,Income) } N
        :- noutput(N).
```

18

The previous clause also includes a cardinality constraint, stating that the number of atoms for which satisfied_order is true should be between 0 and N (the number of outputs). This clause corresponds to the constraint defined in Eq. (20).

The other decision to take is the destination of each article. An article identified by Article_id can be used to satisfy an order identified by an Order_id; in such a case the atom article_used(Article_id,Order_id) will be true. Otherwise, the article can *stay* (making true atom article_stay(Article_id)) or be sent back (making true atom article_back(Article_id)).

```
{ article_used(Article_id,Order_id) } :-
    article(Article_id, _, _, _, _, Feed_category, _),
    order(Order_id, Feed_category, _, _, _).
{ article_back(Article_id) } :-
    article(Article_id, _, _, _, _, _, _).
{ article_stay(Article_id) } :-
    article(Article_id, _, _, _, _, _, _).
```

Note that the article can be used only to satisfy orders of the same feed category. Clearly, for a given article the three destinations are mutually exclusive; this can be imposed through the following integrity constraints:

```
:- article_stay(ArticleId), article_back(ArticleId).
:- article_stay(ArticleId), article_used(ArticleId, _).
:- article_back(ArticleId), article_used(ArticleId, _).
```

We still have to impose that at least one destination must be selected for each article. We first define that if an article is sent back or has to stay, then that the article is unused:

```
article_unused(Article_id) :- article_back(Article_id).
article_unused(Article_id) :- article_stay(Article_id).
```

Then we state that, for a given article, it is impossible that it is neither used for an order nor unused:

```
 :- not article_used(Article_id, _),
    not article_unused(Article_id),
    article(Article_id, _, _, _, _, _, _).
```

It is not possible for an article to be used for different orders.

```
:- article_used(ArticleId,OrderId1),
→   article_used(ArticleId,OrderId2),
    OrderId1 != OrderId2.
```

All the clauses defined above involving the atoms article_used, article_back and article_stay correspond to the constraints defined in Equations 14 and 15.

We can now define the concept of open boxes; a box is opened if it contains an article that is either used to satisfy an order or sent back to the warehouse. The following clauses correspond to the constraints in Eq. (16):

```
open_box(Box_id) :-
  article_used(Article_id,Order_id),
  article(Article_id, _, Box_id, _, _, _, _).
open_box(Box_id) :-
  article_back(Article_id),
  article(Article_id, _, Box_id, _, _, _, _).
```

If a box was opened, all of its content must be extracted, so articles contained in it cannot be in status *stay*. This clause corresponds to the constraints in Eq. (17):

```
:- article_stay(Article_id),
   article(Article_id, _, Box_id, _, _, _, _),
   open_box(Box_id).
```

To avoid that a box is opened just to send back all its content, we first define that a box is used for some order (predicate **box_used_in_order**) if there is some order using one of its articles:

```
box_used_in_order(Box_id) :-
        article_used(Article_id,Order_id),
        article(Article_id, _, Box_id, _, _, _, _).
```

and then impose that one box cannot be opened and not used for some order:

```
:- open_box(Box_id), not box_used_in_order(Box_id).
```

The two clauses above correspond to the constraints in Eq. (18).

Also, it should be forbidden for an article to be used to satisfy an order if that order is not (fully) satisfied (it corresponds to Eq. (19)):

```
:- not satisfied_order(Order_id),
↪   article_used(Article_id,Order_id).
```

and an order cannot be satisfied if the sum of the quantities of the articles used to satisfy such order is not enough (see Eq. (21) and (23))

```
:- satisfied_order(Order_id),
   order(Order_id, Feed_category, Deadline, OrderQuantity,
   ↪   Income),
   #sum { ArtQuantity, Article_id :
          article_used(Article_id, Order_id),
          article(Article_id, _, Box_id, _, _, Feed_category,
          ↪   ArtQuantity)
        } < OrderQuantity.
```

On the other hand, the sum of the quantities of the articles used to satisfy an order should not exceed the request by more than a given surplus (see Eq. (22)):

```
:- order(Order_id, Feed_category, Deadline, OrderQuantity,
↪  Income),
   max_order_surplus(Surplus),
   satisfied_order(Order_id),
   #sum { ArtQuantity, Article_id :
         article_used(Article_id, Order_id),
         article(Article_id, _, Box_id, _, _, Feed_category,
         ↪  ArtQuantity)
       } > OrderQuantity + Surplus.
```

6.2. Optimization

The objective is maximizing the profit, as defined in Equation 10; it is the algebraic sum of four terms, taking into consideration the income for satisfied orders, the storage costs for unused articles, the sorting costs for articles sent back, and the late penalty:

```
#maximize
   { Income,Order_id : satisfied_order(Order_id, Income);
   -ArticleStorageCost,Article_id : storage_cost(StorageCost),
       article_unused(Article_id),
       article(Article_id, Ean, _, _, _, _, ArtQuantity),
       item(Ean, ItemQuantity, _),
       ArticleStorageCost = ArtQuantity / 1000 * StorageCost;
   -ArticleSortingCost,Article_id : sorting_cost(SortingCost),
       article_back(Article_id),
       article(Article_id, Ean, _, _, _, _, ArtQuantity),
       item(Ean, ItemQuantity, _),
       ArticleSortingCost = (ArtQuantity / ItemQuantity) *
       ↪  SortingCost;
   -LateDaysCost, Order_id :  late_order_penalty(LateCost),
       not satisfied_order(Order_id),
       order(Order_id, _, Deadline, _, _),
       LateDaysCost = @late_days(Deadline) * LateCost
   }.
```

@late_days/1 is a predicate that calls an external Python function that returns the number of delay days including the current one.

7. Experiments

A prototypical SORT plant is still under development, therefore real data about orders and articles is still not available. To test the scalability of the

proposed approaches we generated synthetic random datasets with 30 orders and an increasing number of articles (and hence an increasing number of boxes). In our experiments, each order has an income varying from 0.14 to 0.23 euros per kg (the income of each order was randomly chosen), we used the price of corn which can be variable and is used as economic point of reference for the price of animal feed materials based on grain. The price of other feed materials such as meat can be different but we do not have enough information to define competitive prices for these materials (the economic model of the SORT project is still under development). However, these differences of prices do not affect the timings required for finding the optimal solutions by our models. The quantity of requested feed material for each order, instead, varies[3] from 100 kg to 500 kg and the surplus was set to 5 kg. Moreover, the maximum weight that a box can carry was set to 15 kg (limit imposed by law) and its max capacity is 0.1008 m^3 ($400 \times 600 \times 420$ mm).

The size of the datasets is expressed in number of articles. As mentioned before, a food article is an aggregation of instances of a food item. In our experiments, each food article was generated by aggregating a number of instances between 1 and N, where N is the maximum number of instances of a given food item that a box can contain according to the maximum weight and capacity. In our synthetic dataset generator, every time a food article is created, it is assigned to the current box if there is enough space to contain it and if adding it does not exceed the weight limit of the box. Otherwise, a new box identifier is generated and the newly created article is assigned to it.

Each food item has a volume that can vary from 100 (small yogurt cup) to 4'000 cm^3 (limits imposed by the Sorter machinery) and a density that varies from 85 (roughly the density of a box of cereals) to 1100 kg/m^3 (roughly the density of a small yogurt cup). The quantity, i.e. the weight, of an item depends, obviously, on the volume and the density chosen.

The quantity and the income of an order, the volume and the density of a food item and the number of food item instances for an article are values that are randomly chosen using the discrete uniform distribution. Finally, we set the storage cost to 0.0005 euros per kg, which is the estimated electricity cost in Italy for the designed refrigerated warehouse, and the sorting cost to 0.10 euros, which is the estimated cost of one minute of time spent by a human worker (it might be necessary to pour the content of the go back boxes into the hopper or to check for accidental damages).

In our experiments we compare the performance of the three methods previously detailed: the BLP and ASP, and CLP(FD) approaches.

We performed two types of experiments. In the first, we compared the three approaches on datasets containing up to 300 articles. In the second experiment, we tested the scalability of the best performing approach on larger datasets. In

---

[3]Feed factories usually process tons of material, therefore, given that they can take feed material from different sources, we expect to receive orders of the magnitude of hundreds of kilograms.

a previous work (Chesani et al., 2018), we evaluated the CLP(FD) program, implemented in ECL^{i}PS^e, by combining several heuristics and strategies for search and optimization. The experimental results in (Chesani et al., 2018) showed that the best combination is the one where, for all the variables, `select_income` is the variable selection method and `indomain_max` is the value choice method, i.e. the values of a variable are tried in decreasing order, and `continue` is the optimization strategy[4]. Therefore, in this work, we report the performances of the CLP(FD) approach using only this combination of heuristics.

All the tests were performed on GNU/Linux machines equipped with Intel Xeon E5-2697 v4 (Broadwell) @ 2.30 GHz, using 1 core for each test. In all the experiments, we set an execution timeout of 4 hours.

7.1. Test 1: comparison of different approaches

We consider a number of articles from 100 to 300 in steps of 10 and, for each number of articles, we generated 10 datasets. We performed optimization by using the CLP(FD) approach with the best configuration for the search, the ASP and the BLP approaches. Then we compared the three approaches.

Figure 3 reports the geometric mean of the execution time in seconds for finding the optimal solution for 10 different random datasets of increasing size. The star symbol means that in one dataset the timeout occurred without finding the best solution, therefore the execution time was averaged over 9 datasets. If the timeout occurred at least in two datasets the timing is not reported.

The results show that the ASP program has the best performances up to datasets with 230 articles and it is able to scale up to 240 articles. The CLP(FD) approach starts having troubles already with 130 articles[5]. Note also that the parameters of the search in ECL^{i}PS^e were tuned in (Chesani et al., 2018) to the values that gave best performance in the given application, while we left the default parameters of `clingo`. This further shows the efficacy of the default search values used in ASP solvers.

However, when the datasets become larger the ASP approach is no longer able to scale, while the BLP approach does not have particular issues with datasets containing up to 300 articles.

7.2. Test 2: evaluation of the BLP approach with bigger datasets

In these experiments we evaluate the BLP approach on datasets that have a greater number of articles. We consider a number of articles from 400 to 1,000 in steps of 100 and, for each number of articles, we generated 10 datasets. Table 1 reports the geometric mean of the execution time in seconds for finding the optimal solution with the BLP approach for 10 different random datasets

---

[4]With the exception of `select_income`, the other heuristics are pre-defined options of the `search` predicate of ECL^{i}PS^e (see `http://eclipseclp.org/doc/bips/lib/gfd_search/search-6.html`).

[5]In (Chesani et al., 2018) the CLP(FD) program was able to scale up to 400 articles because in that paper we had set the option `delta` to 10,000 and the sorting penalty to 0, while in the current work we set `delta` to 1 and article sorting penalty to 10.
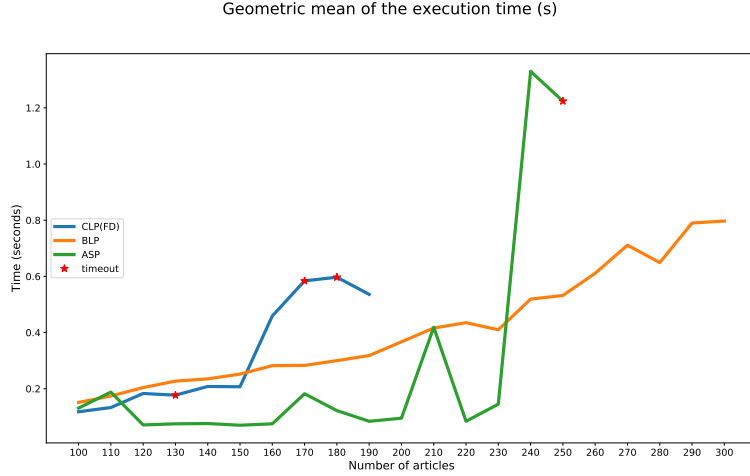
Figure 3: Geometric mean of the execution time (in seconds) for the search of the optimal solutions using the three approaches for each dataset size. The star symbol in the graph means that in one dataset the timeout occurred without finding the best solution, therefore geometric mean of the execution time was computed over 9 datasets.

of increasing size. The results show that the BLP approach is able to scale up to 800 articles. With datasets containing 900 or 1000 articles, the timeout was reached in 4 of them.

Table 1: Geometric mean of the execution time (in seconds) for the search of the optimal solution using the BLP approach for each dataset size. The symbol '*' ('**') means that in one dataset (two datasets) the timeout occurred without finding the best solution, therefore the execution time was averaged over 9 (8) datasets. The cells with '–' indicate that the timeout occurred at least in three datasets.

Approach	400	500	600	700	800	900	1000
BLP	1.542	6.429*	8.560**	38.243*	123.261*	–	–

The results show that the BLP approach is able to find the optimal solution up to datasets containing 800 articles. With more than 800 articles the SORT plant manager will have to settle for suboptimal solutions. It is worth pointing out that, in our experimental setting, dealing with 800 food articles means dealing with more than 500 hundred boxes (the average in our datasets is 546 boxes) which is more than half of the maximum capacity of the designed warehouse of the SORT plant.

7.3. Discussion

In this particular application the BLP approach showed best performance amongst the three devised technologies. This is probably due to the fact that, in

the given formulation, all the constraints were cast as linear constraints without resorting to complex linearization schemes, so the linear relaxation provides a good bound to the real value of objective function, which strongly empowers the branch-and-bound algorithm employed in BLP. Also, the set of constraints are not too tight, leaving many combinations as feasible solutions, so the main difficulty is finding the optimum and proving optimality, two strong points of BLP with respect to its competitors.

The CLP(FD) formulation was the worst performing, probably due to the fact that it does not have strong global constraints guiding the search, and that the satisfaction part of the problem was not the most challenging part.

ASP provided the best results for small instances, probably due to the advanced search strategies implemented in `clingo`. Still, it could not compete with BLP in larger instances.

The source code of the proposed approaches and the synthetic datasets used in our experiments are contained in the following git repository: `https://bitbucket.org/machinelearningunife/dss_models`.

8. Conclusions

Using logic programming instead of imperative-based solutions provides several well-known advantages such as declarativity, rapid prototyping and ease of modifiability in case of changing requirements.

In this paper we presented a DSS that considers two paradigms for finding the optimal solution given the content available in the warehouse of the SORT plant. One approach is based on BLP and developed in ECL^iPS^e, while the other one is based on ASP and developed in `clingo`.

The experiments showed that both BLP and ASP are valid approaches for modelling and solving complex problems such as the SORT process. However, for this process, the BLP approach outperforms the one based on ASP and the previous one based on CLP(FD) (Chesani et al., 2018). This certainly does not mean that the BLP approach is, in general, better than the other approaches, but, for this particular problem posed by the SORT process, the BLP approach has a much better performance.

The devised DSS has still some limitations. So far, in our model, every article has the same storage cost, however there exist several kinds of warehouses (refrigerated, room temperature, etc.); in the future we plan to use different storage costs depending on the type of warehouse in which the article was stored. Moreover, we will consider costs for using machinery for unpacking and food processing. These last costs will depend on the type of packaging and on the food type and consistency (opening a carton of milk can have a different cost than opening a plastic can of powdered milk).

In addition, once the prototype of SORT plant has been built, we plan to test our DSS with real data and to tackle more complex scenarios where orders can request mixtures of feed materials. We would also like to improve our model by taking into account not only articles already in the warehouse, but also those currently being transported to the SORT plant.

In our approach, we used ECL^{i}PS^e for representing the BLP constraints, it could be interesting to encode our problem with MiniZinc (Nethercote et al., 2007) and compare all the approaches. A completely different direction of research could be to combine BLP and CLP(FD) solvers. Various combinations could be considered; one of the possibility is to use a portfolio approach (Amadini et al., 2016). However it makes sense to do all of this after we have real data. We reserve the implementations of these different approaches as future work.

References

Amadini, R., Gabbrielli, M., Mauro, J., 2016. Portfolio approaches for constraint optimization problems. Ann. Math. Artif. Intell. 76, 229–246. URL: `https://doi.org/10.1007/s10472-015-9459-5`, doi:10.1007/s10472-015-9459-5.

Chesani, F., Cota, G., Lamma, E., Mello, P., Riguzzi, F., 2018. A decision support system for food recycling based on constraint logic programming and ontological reasoning, in: Proceedings of the 33rd Italian Conference on Computational Logic, Bolzano, Italy, September 20-22, 2018, CEUR-WS.org, Aachen. pp. 117–131. URL: `http://ceur-ws.org/Vol-2214/paper13.pdf`.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M., 2011. Potassco: The Potsdam answer set solving collection. AI Commun. 24, 105–124.

Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., 2012. Answer set solving in practice. Synthesis lectures on artificial intelligence and machine learning 6, 1–238.

Gelfond, M., Lifschitz, V., 1988. The stable model semantics for logic programming., in: 5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988), MIT Press. pp. 1070–1080.

Giunchiglia, E., Lierler, Y., Maratea, M., 2006. Answer set programming based on propositional satisfiability. J. Autom. Reasoning 36, 345–377.

Gurobi Optimization, LLC, 2018. Gurobi optimizer reference manual. Available from `http://www.gurobi.com`.

Gustavsson, J., Cederberg, C., Sonesson, U., van Otterdijk, R., Meybeck, A., 2011. Global food losses and food waste: extent, causes and prevention. URL: `http://www.fao.org/docrep/014/mb060e/mb060e00.pdf`.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F., 2006. The DLV system for knowledge representation and reasoning. ACM T. Comput. Log. 7, 499–562.

Lin, F., Zhao, Y., 2004. ASSAT: computing answer sets of a logic program by SAT solvers. Artif. Intell. 157, 115–137.

Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G., 2007. MiniZinc: Towards a standard CP modelling language, in: International Conference on Principles and Practice of Constraint Programming, Springer. pp. 529–543.

Saltzman, M.J., 2002. COIN-OR: An open-source library for optimization, in: Nielsen, S. (Ed.), Programming Languages and Systems in Computational Economics and Finance, Springer, Boston, MA.

Schimpf, J., Shen, K., 2012. ECL^{i}PS^e - from LP to CLP. Theor. Pract. Log. Prog. 12, 127–156. URL: `https://doi.org/10.1017/S1471068411000469`, doi:`10.1017/S1471068411000469`.

Shen, K., Schimpf, J., 2005. Eplex: Harnessing mathematical programming solvers for constraint logic programming, in: van Beek, P. (Ed.), Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings, Springer. pp. 622–636. URL: `https://doi.org/10.1007/11564751_46`, doi:`10.1007/11564751\_46`.

Simons, P., Niemelä, I., Soininen, T., 2002. Extending and implementing the stable model semantics. Artif. Intell. 138, 181–234. doi:`10.1016/S0004-3702(02)00187-X`.