Comparison of two constraint programming algorithms for computing leximin-optimal allocations

Sylvain Bouveret[12] and **Michel Lemaître**[1]

Abstract.

We propose two constraint programming algorithms for solving the following problem: fairly and efficiently allocate a finite set of objects to a set of agents, each one having their own utilities, under admissibility constraints. Our algorithms compute an allocation maximizing the $\mathrm{leximin}$ order on the utility profiles of the agents. Our main contribution is the use of a cardinality meta-constraint on the one hand, and an adaptation of an existing algorithm for enforcing a multiset ordering constraint on the other hand.

Moreover, we describe the application domain that motivated this work: sharing of satellite resources. We extract from this real-world application a simple and precise fair allocation problem that allows for testing and evaluating our algorithms, using a benchmark generator. The implementations of both algorithms have been tested using the constraints programming tool CHOCO [12], and a translation of the first algorithm to integer linear programming has been tested using CPLEX [10].

1 INTRODUCTION

Efficiently and fairly allocating a bounded set of resources to several agents having their own preferences is a rather general problem having a wide scope of applications. Some examples can easily be found: balanced timetables, fair share of communication networks, allocation of airport and airspace resources to several airlines, fair share of a constellation of Earth observation satellites.

Within this paper, this problem will be restricted using four additional hypotheses: 1) the set of resources is discrete and finite, and is thus equivalent to a finite set of indivisible and distinct objects; 2) the agents preferences over the set of admissible allocations are numerically expressed; 3) we seek *fair* and *efficient* allocations (the meaning of these words will be explained later); 4) the optimal allocation is computed in a *centralized* manner, by a "benevolent arbitrator", assumed to be fair, and obeying principles that are accepted by all the agents. In other words, we will not deal with distributed allocation procedures. We may notice that many real-world problems (like those cited before) still match these four hypotheses.

This problem has been studied in different active research communities: Operational Research (OR), Artificial Intelligence (AI), Microeconomy, Social Choice Theory. Our contribution is inspired by all of them: from the last two we borrow the idea of representing the agents preferences by *utility* levels, and we adopt the *leximin* preorder for conveying the fairness and efficiency requirements. OR and AI provide the CSP and constraint programming frameworks as effective and flexible computation and modeling tools.

Numerical preferences and utilities Let $\mathcal{S}$ be a finite set of admissible alternatives concerning n agents (each one having her own preferences), among which an arbitrator must choose one. The most classical model describing this situation *welfarism* (see *e.g.* [11, 15]). According to this model (which will be used in this paper), the decision of the arbitrator is based on the satisfaction levels enjoyed by the individual agents, and on those levels only. These levels are measured, in the cardinal version of this model, by a numerical index giving the *individual utility* $u_i(s)$ of agent i concerning alternative s. We admit here that the individual utilities are comparable between the agents (in other words, they are expressed in a common utility scale). Thus we can map each alternative s to a *utility profile* $\langle u_1(s), \ldots, u_n(s)\rangle$, and moreover this is the only relevant information we use to compare two alternatives.

An easy way to compare individual utility profiles is to aggregate each of them into a *collective utility* index, standing for the collective welfare of the agents community. If g is a well-chosen aggregation function, we thus have a collective utility function uc that maps each alternative s to a collective utility level $g(u_1(s), \ldots, u_n(s))$. An optimal alternative is one of those maximizing the collective utility.

Fairness and efficiency using the leximin preorder The main difficulty of our fair allocation problem lays on the fact that we have to reconcile the contradictory wishes of the agents. There is generally no allocation that fully satisfies everyone; we therefore request aggregation functions g leading to fair and efficient compromises. This notion of efficiency (simply meaning that resources have to be used as much as possible) is usually represented by Pareto-optimality[1].

The problem of choosing the right aggregation function g is far beyond the scope of this paper. We will only describe the two most usual ones, that stand for two rather extreme points of view on social welfare[2]. The first one is the sum function, that leads to the *classical utilitarianism*, and the second one is the $\min$ function, used by *egalitarianists*. The first function is not very relevant to our particular problem, since the resulting collective utility does not depend on the balancing of the utility profile. On the contrary, the minimum function is particularly well-suited for the kind of problems in which fairness plays an important role, because the optimal decisions are those maximizing the satisfaction of the less happy of the agents. However,

[1] Office National d'Études et de Recherches Aérospatiales – DCSD. 2, avenue Édouard Belin, B.P. 4025. F-31055 Toulouse cedex 4

[2] Institut de Recherche en Informatique de Toulouse. 118, route de Narbonne. F-31062 Toulouse cedex.

[1] A decision is Pareto-optimal if and only if we cannot strictly increase the satisfaction of an agent unless we strictly decrease the satisfaction of another agent.

[2] Some compromises between these two extremes exist. See *e.g.* [15, page 68] (sum of powers) or [19] (*Ordered Weighted Averaging aggregators*)

a major drawback of the minimum function, classically studied in the community of fuzzy CSP, and usually called "drowning effect" [3], is that this function leaves many alternatives indistinguishable. Thus for example, the utility profiles $\langle 0, \ldots, 0\rangle$ and $\langle 1000, \ldots, 1000, 0\rangle$ both produce the same collective utility 0, in spite of the fact that the second one appears to be far better than the first one. In other words, this aggregation function can lead to non Pareto-optimal decisions, which is not desirable.

Two classical refinements of the order induced by the $\min$ function overcoming this drawback can be proposed: the discrimin and $\mathrm{leximin}$ orders [5]. The first one is not a total preorder. The second one is classically used in Social Choice [14], and this is the refinement that we will use in this paper. Before introducing it formally in section 2, we describe it informally. Comparing two utility profiles using the leximin preorder is not based on an aggregation function, but on the profiles themselves. First, the two minimal values of the profiles are compared: if they are different, the biggest one wins; otherwise, these two minimal values are removed from the utility profiles and we repeat the same operation on the two new profiles. An equivalent procedure is to sort each profile in non-decreasing order and then to compare them using the lexicographic order.

This paper is organized as follows: section 2 formally defines our problem using the CSP framework. Section 3 presents our main contribution: it describes two algorithms for computing a $\mathrm{leximin}$-optimal alternative using constraint programming. These algorithms are based on existing constraints; our contribution is to use them to fit the needs of our problem for fairness and efficiency.

Section 4 deals with the motivating real-world application: sharing a constellation of Earth observation satellites. We extract from this real-world application a simple and precise fair allocation problem that allows for testing and evaluating our algorithms, using a benchmark generator. The implementations of the two algorithms have both been tested using the constraints programming tool CHOCO [12], and a translation of the first algorithm to integer linear programming has been tested using CPLEX [10]. Section 5 presents related work, just before conclusion and future work, section 6.

2 FRAMEWORK

The constraint programming framework is widely used for solving many different combinatorial problems such as timetable problems, scheduling problems, frequency allocation problems... This paradigm is based on the notion of *constraints network*. A constraints network is made of a set of variables $\mathcal{X} = \{x_1, \ldots, x_p\}$, where d_{x_i} is the finite set of possible values for x_i (we assume $d_{x_i} \subset \mathbb{N}$, and we use the following notations: $\underline{x_i} = \min(d_{x_i})$ and $\overline{x_i} = \max(d_{x_i})$), and of a set of constraints $\mathcal{C}$. Each constraint $C \in \mathcal{C}$ describes a set of allowed tuples $R(C)$ over a set of variables $X(C)$.

An instantiation v of a set S of variables is a function that maps each variable $x \in S$ to a value $v(x)$ of its domain d_x. If $S = \mathcal{X}$, this instantiation is said to be complete, otherwise, it is partial. If $S' \subsetneq S$, the projection of an instantiation of S over S' is the restriction of this instantiation to S', and it is written $v_{\downarrow S'}$. An instantiation is said to be consistent if and only if it satisfies every constraint. Given a constraints network, the problem of finding whether there exists a complete and consistent instantiation to this constraints networks is called Constraint Satisfaction Problem (CSP) and is NP-complete. Such an instantiation, if it exists, is called a solution of the CSP.

There is an optimization problem derived from the CSP (coming from the max-CSP extension of constraint satisfaction problems), where a special variable o plays the role of *objective variable*. A solution of an instance of this optimization problem is a complete consistent instantiation $\widehat{v}$ of the constraints network such that $\widehat{v}(o) = \max\{v'(o) | v' \text{ complete consistent instantiation}\}$.

Let $\overrightarrow{x} = \langle x_1, \ldots, x_n\rangle$ be a vector of numbers; we write $\overrightarrow{x}^{\uparrow} = \langle x_1^{\uparrow}, \ldots, x_n^{\uparrow}\rangle$ the vector made of every components of $\overrightarrow{x}$, sorted in non-decreasing order. We now define the leximin preorder over integer vectors:

Definition 1 (leximin preorder [14]) *Let $\overrightarrow{x}$ and $\overrightarrow{y}$ be two vectors of $\mathbb{N}^n$. $\overrightarrow{x}$ and $\overrightarrow{y}$ are said* $\mathrm{leximin}$*-indifferent (written $\overrightarrow{x} \sim_{leximin} \overrightarrow{y}$) if and only if $\overrightarrow{x}^{\uparrow} = \overrightarrow{y}^{\uparrow}$. The vector $\overrightarrow{y}$ is* $\mathrm{leximin}$*-preferred to $\overrightarrow{x}$ (written $\overrightarrow{x} \prec_{leximin} \overrightarrow{y}$) if and only if $\exists i \in [\![0, n-1]\!]$ such that $\forall j \in [\![1, i]\!]$, $x_j^{\uparrow} = y_j^{\uparrow}$ and $x_{i+1}^{\uparrow} < y_{i+1}^{\uparrow}$. We write $\overrightarrow{x} \preceq_{leximin} \overrightarrow{y}$ for $\overrightarrow{x} \prec_{leximin} \overrightarrow{y}$ or $\overrightarrow{x} \sim_{leximin} \overrightarrow{y}$.*

The binary relation $\preceq_{leximin}$ is a total preorder.

In a collective decision making problem, a $\mathrm{leximin}$-optimal solution is an alternative whose associated utility profile is maximal for the preorder $\preceq_{leximin}$. The main advantage of such a solution is that it is both $\min$-optimal and Pareto-efficient.

The CSP optimization variant allows for encoding the problem of maximizing the egalitarian collective utility (with $g = \min$) of n agents: we introduce n variables $\langle u_1, \ldots, u_n\rangle$ corresponding to the utilities of each agent, and one objective variable uc, linked with the other variables by the set of constraints $\{C_1, \ldots, C_n\}$, with $C_i = (uc \leq u_i)$.

However, expressing the problem of computing a $\mathrm{leximin}$-optimal decision in the CSP framework is not straightforward. It indeed needs a slight modification of the optimization version of the CSP. Let us consider the following problem:

LEXIMIN-OPTIMAL

Inputs:	a constraints network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; a vector of variables $\overrightarrow{u} = \langle u_1, \ldots, u_n\rangle$ ($\forall i,\ u_i \in \mathcal{X}$), called *objective vector*.
Output:	"Inconsistent" if there is no complete consistent instantiation. Otherwise a complete consistent instantiation $\widehat{v_s}$ such that for all complete consistent instantiation v, $v(\overrightarrow{u}) \preceq_{leximin} \widehat{v_s}(\overrightarrow{u})$.

We propose two generic algorithms for solving this problem. The first one is based on a cardinality meta-constraint, and the second one is based on a multiset ordering constraint.

3 TWO ALGORITHMS

3.1 Using cardinality constraints

The main idea of algorithm 1 is to iteratively compute each component of the sorted version of the optimal objective vector (*i.e.* the vector of values of the $\mathrm{leximin}$-optimal instantiation of $\overrightarrow{u}$). Thus we introduce at lines 3 and 4 a vector of optimization variables, the role of each variable y_i being to compute the value of the index i of the $\mathrm{leximin}$-optimal vector. During each iteration i of the loop 6..10, a cardinality constraint corresponding to the currently computed component is added (line 7), and then we compute (line 8) the maximal value of variable y_i such that the current constraints network (corresponding to the initial one, added with variables y_k and with cardinality constraints from previous iterations) has a solution. The variable y_i is fixed to this optimal value (line 9) for all the following iterations. In line 10 the domain of the next variable y_{i+1} is

Algorithm 1: Pseudo-code of an algorithm that computes a leximin-optimal solution of a constraints network using cardinality constraints.

inputs: a constraints network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; a vector $\langle u_1, \ldots, u_n \rangle$ of variables from $\mathcal{X}$

output: A solution to the problem [LEXIMIN-OPTIMAL] or "Inconsistent"

1 **if solve**$(\mathcal{X}, \mathcal{D}, \mathcal{C}) =$*"Inconsistent"* **then**
2 **return** *"Inconsistent"*
3 $\mathcal{X}' \leftarrow \mathcal{X} \cup \{y_1, \ldots, y_n\}$;
4 $\mathcal{D}' \leftarrow$
$\mathcal{D} \cup \{[\![\min_i(\underline{u_i}), \max_i(\overline{u_i})]\!], \ldots, [\![\min_i(\underline{u_i}), \max_i(\overline{u_i})]\!]\}$;
5 $\mathcal{C}' \leftarrow \mathcal{C}$;
6 **for** $i \leftarrow 1$ **to** n **do**
7 $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n-i+1)\}$;
8 $\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}', \mathcal{D}', \mathcal{C}'))$;
9 $d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;
10 $d_{y_{i+1}} \leftarrow [\![\widehat{v}(y_i), \max_i(\overline{u_i})]\!]$;
11 **return** $\widehat{v}_{\downarrow \mathcal{X}}$;

safely restricted, but this restriction does not seem to be essential (it does not decrease significantly the execution time).

The algorithm uses the cardinality meta-constraint **AtLeast**:

Definition 2 (Meta-constraint AtLeast) *Let Γ be a set of p constraints, and $k \in [\![1, p]\!]$ be an integer. The meta-constraint $\mathbf{AtLeast}(\Gamma, k)$ is the constraint that holds on the union of the scopes of the constraints in Γ, and that allows a tuple if and only if at least k constraints from Γ are satisfied.*

This meta-constraint[3] is introduced as *constraint combinator*, for example in [9]. The role of this constraint in algorithm 1 is to enforce a lexicographic order constraint (the set of cardinality constraints obliges the next solutions to be leximin-superior to the current partial leximin-optimal vector).

The two functions **solve** and **maximize** (the detail of which is the concern of solving techniques for constraints satisfaction problems) of lines 1 and 8 respectively return one solution of the constraints network $(\mathcal{X}', \mathcal{D}', \mathcal{C}')$ (or "Inconsistent" if such a solution does not exist), and an optimal solution of the constraints network $(\mathcal{X}', \mathcal{D}', \mathcal{C}')$, with objective variable y (or "Inconsistent" if such a solution does not exist). We assume – contrary to usual constraints solvers – that these two functions do not modify the input constraints network.

Let us illustrate how the algorithm works with an example:

We consider a basic resource allocation problem, where 3 objects have to be allocated to 3 agents, with the following constraints: (1) each agent must get one and only one object, and (2) one object cannot be allocated to more than one agent. A utility is associated with each pair $(agent, object)$, with respect to the array above.

	a_1	a_2	a_3
o_1	3	3	3
o_2	5	9	7
o_3	7	8	1

This problem has 6 feasible solutions (one for each permutation of $[\![1, 3]\!]$), producing the 6 utility profiles shown in the following array:

	p_1	p_2	p_3	p_4	p_5	p_6
utility for a_1	3	3	5	5	7	7
utility for a_2	9	8	3	8	3	9
utility for a_3	1	7	1	3	7	3

[3] The prefix "meta" indicates that this constraints has some other constraints as parameters.

The algorithm runs in 3 steps:

Step 1: We introduce a variable y_1 and we look for the maximal value $\widehat{y_1}$ of y_1 such that each (**at least** 3) agent gets at least y_1. We find $\widehat{y_1} = 3$.

Step 2: We add the constraint that each agent gets at least $\widehat{y_1} = 3$ (thus removing profiles p_1 and p_3). Then we introduce a variable y_2 and we look for the maximal value $\widehat{y_2}$ of y_2 such that **at least** two agents get at least y_2. We find $\widehat{y_2} = 7$.

Step 3: We add the constraint that at least 2 agents get at least $\widehat{y_2} = 7$ (thus removing profile p_4). Then we introduce a variable y_3 and we look for the maximal value $\widehat{y_3}$ of y_3 such that **at least** one agent gets at least y_3. We find $\widehat{y_3} = 9$. Only one instantiation maximizes y_3: p_6. The corresponding allocation is: $a_1 \leftarrow o_3$, $a_2 \leftarrow o_2$ and $a_3 \leftarrow o_1$.

Proposition 1 *If the two functions **maximize** and **solve** are both correct and both halt, then algorithm 1 halts and returns a solution to the problem [LEXIMIN-OPTIMAL].*

Proof: If functions **solve** and **maximize** both halt, then obviously algorithm 1 halts.

If the initial constraint network has no solution, and if function **solve** is correct, then algorithm 1 returns "Inconsistent". In the following, we will suppose that the initial constraint network has at least one solution.

We will write $\widehat{v_i}$ for the instantiation returned at iteration i by function **maximize**, and $\widehat{v_s}$ a solution to the problem [LEXIMIN-OPTIMAL].

Proof sketch : In order to prove that the algorithm is correct, we show that the call to **maximize** never returns "Inconsistent", and that $\widehat{v_n}(\overrightarrow{u})^{\uparrow} = \widehat{v_s}(\overrightarrow{u})^{\uparrow}$. To do that, we make use of induction on the following hypothesis: $(H_i) = \left(\widehat{v_i} \text{ exists and } \forall j \leq i,\ \widehat{v_i}(\overrightarrow{u})^{\uparrow}_j = \widehat{v_i}(y_j) = \widehat{v_s}(\overrightarrow{u})^{\uparrow}_j\right)$.

At iteration 1, the constraint $\mathbf{AtLeast}(\{u_1 \geq y_1, \ldots, u_n \geq y_1\}, n)$ is equivalent to the constraint $y_1 \leq \min_i(u_i)$. Therefore **maximize** returns an extension $\widehat{v_1}$ of a solution of the initial constraint network, such that $\widehat{v_1}(y_1) = \max\{\min_i(v_1(u_i)) | v_1 \text{ is a complete consistent instantiation}\}$. Thus we have $\widehat{v_1}(y_1) = \widehat{v_1}(\overrightarrow{u})^{\uparrow}_1$, Moreover, $\widehat{v_1}(y_1) \geq \widehat{v_s}(\overrightarrow{u})^{\uparrow}_1$ if **maximize** is correct (there is a consistent extension of $\widehat{v_s}$ on $(\mathcal{X}', \mathcal{D}', \mathcal{C}')$). We cannot have $\widehat{v_1}(y_1) > \widehat{v_s}(\overrightarrow{u})^{\uparrow}_1$, because it would mean that $\widehat{v_1}(\overrightarrow{u})^{\uparrow}_1 > \widehat{v_s}(\overrightarrow{u})^{\uparrow}_1$ and thus that $\widehat{v_1}$ would be a solution leximin-superior to $\widehat{v_s}$, which is not possible. This proves (H_1).

Let us show that $(H_i) \Rightarrow (H_{i+1})$, $1 \leq i \leq n-1$.

Let us first prove that $\widehat{v_{i+1}}$ exists (that is, the constraint network of iteration $i+1$ has at least one solution).

The instantiation $\widehat{v_s}$ is the projection over $\mathcal{X}$ of a solution of the constraint network at iteration $i+1$. Indeed:

— by definition, $\widehat{v_s}$ satisfies all the constraints of the initial network;

— $\forall j \leq i$, $\widehat{v_i}(y_j) = \widehat{v_s}(\overrightarrow{u})^{\uparrow}_j$ (following from (H_i)), thus $\widehat{v_s}(\overrightarrow{u})^{\uparrow}_j$ and all its following components in the sorted vector (that is $n-j+1$ components of $\widehat{v_s}(\overrightarrow{u})$) are greater or equal to $\widehat{v_j}(y_j)$, which satisfies constraint **AtLeast** at iteration j.

— $\widehat{v_s}(\overrightarrow{u})^{\uparrow}_{i+1} \geq \widehat{v_s}(\overrightarrow{u})^{\uparrow}_i$ by definition, thus there is at least one consistent value for y_{i+1} : $\widehat{v_i}(y_i)$.

Therefore the constraint network at iteration $i+1$ has at least one solution (thus $\widehat{v_{i+1}}$ exists).

Moreover, for all $j \leq i+1$, $\widehat{v_{i+1}}(\overrightarrow{u})^{\uparrow}_j \geq \widehat{v_{i+1}}(y_j)$ (otherwise at least one of the constraints **AtLeast** is violated). By noticing that an admissible allocation for $(\mathcal{X}', \mathcal{D}', \mathcal{C}')$ at iteration $i+1$ is also admissible at iteration i (because we only add a constraint and reduce the domain of variables between two iterations), we deduce that we cannot have $\widehat{v_{i+1}}(\overrightarrow{u})^{\uparrow}_j > \widehat{v_{i+1}}(y_j)$. Indeed, if it were the case, since $\widehat{v_{i+1}}(y_j) = \widehat{v_j}(y_j)$ (because for each $j < i+1$, the domain of y_j is a singleton), $\widehat{v_{i+1}}$ would have been stricly better than $\widehat{v_j}$ for

y_j at iteration j, which is not possible if **maximize** is correct. Thus $\forall j \leq i+1$, $\widehat{v_{i+1}}(\overrightarrow{u})^{\uparrow}_j = \widehat{v_{i+1}}(y_j)$, which proves the first equality.

The extension of $\widehat{v_s}$ that instantiates the value $\widehat{v_s}(\overrightarrow{u})^{\uparrow}_{i+1}$ to y_{i+1} is feasible at iteration $i+1$ (it satisfies, as well as the other constraints, the constraint **AtLeast** of iteration $i+1$). Therefore $\widehat{v_{i+1}}(y_{i+1}) \geq \widehat{v_s}(\overrightarrow{u})^{\uparrow}_{i+1}$. If we had $\widehat{v_{i+1}}(y_{i+1}) > \widehat{v_s}(\overrightarrow{u})^{\uparrow}_{i+1}$, then the projection of $\widehat{v_{i+1}}$ over $\mathcal{X}$ would be a solution of this constraint network such that $\forall j < i+1$, $\widehat{v_{i+1}}(\overrightarrow{u})^{\uparrow}_j = \widehat{v_s}(\overrightarrow{u})^{\uparrow}_j$ and $\widehat{v_{i+1}}(\overrightarrow{u})^{\uparrow}_{i+1} > \widehat{v_s}(\overrightarrow{u})^{\uparrow}_{i+1}$, thus a leximin-superior solution than $\widehat{v_s}$, which is not possible. We thus have $\widehat{v_{i+1}}(y_{i+1}) = \widehat{v_s}(\overrightarrow{u})^{\uparrow}_{i+1}$, which finally proves (H_{i+1}).

By induction, we have: (1) $\widehat{v_n}$ is a solution of the constraint network at iteration n, thus *a fortiori* its projection over $\mathcal{X}$ is a solution, and (2) for each i, $\widehat{v_n}(\overrightarrow{u})^{\uparrow}_i = \widehat{v_s}(\overrightarrow{u})^{\uparrow}_i$, thus $\widehat{v_n}(\overrightarrow{u})$ and $\widehat{v_s}(\overrightarrow{u})$ are leximin-indifferent.

Therefore the instantiation returned by algorithm 1 is a solution of the problem [LEXIMIN-OPTIMAL]. ■

Constraint programming suits particularly to the implementation of this algorithm; however, the cardinality meta-constraint is also expressible using the linear programming framework [8, p.11], by introducing n 0–1 variables $\{\delta_1, \ldots, \delta_n\}$. The cardinality constraint $\mathbf{AtLeast}(\{x_1 \geq y, \ldots, x_n \geq y\}, k)$ is then equivalent to the set of linear constraints $\{x_1 + \delta_1\overline{y} \geq y, \ldots, x_n + \delta_n\overline{y} \geq y, \sum_{i=1}^{n} \delta_i \leq n-k\}$.

3.2 Using a multiset ordering constraint

The second procedure we present for computing a leximin-optimal solution of a constraints network is based on the multiset ordering constraint introduced in [6]. In this paper, the authors describe an algorithm for enforcing generalized arc-consistency[4] on a multiset ordering constraint. Informally, such a constraint works on two multisets of variables M and N (unordered lists of variables where repetition is allowed), and ensures that $M \preceq_m N$, $\prec_m$ being the standard strict order on multisets: $M \prec_m N$ if and only if either M is empty and N is not, or the largest value in M is smaller than the largest one in N, or they are the same and, if we eliminate one occurrence of the largest value from both M and N, the resulting two multisets are ordered.

We can notice that this order also works for vectors of variables (that can be viewed as multisets), and that it is close to our definition of the leximin-ordering: the only difference is that at each step, we focus on the smallest value instead of the biggest one. The algorithm described in [6] is based on two vectors of length $u-l$ (with $u = \max(\{\overline{M_i} | i \in [\![0, |M|]\!]\} \cup \{\overline{N_j} | j \in [\![0, |N|]\!]\})$ and $l = \min(\{\underline{M_i} | i \in [\![0, |M|]\!]\} \cup \{\underline{N_j} | j \in [\![0, |N|]\!]\})$) called *occurrence vectors*. The basic algorithm explicitly computes these two vectors and runs in time $O(|N|+|M|+u-l)$. However, in our particular case, $u-l$ can be rather huge, as we will see in the description of the application, in section 4. We thus use a variant of this algorithm, suggested in [6], that does not compute the occurrence vectors, and runs in time $O(|N|\log(|N|) + |M|\log(|M|))$. The latter algorithm can also be easily translated to enforce the strict leximin ordering between one vector of variables and one vector of integers.

We define the constraint **Leximin**:

Definition 3 (Constraint Leximin) *Let $\overrightarrow{x}$ be a vector of variables and $\overrightarrow{\lambda}$ be a vector of integers. The constraint* $\mathbf{Leximin}(\overrightarrow{\lambda}, \overrightarrow{x})$ *concerns every variables belonging to $\overrightarrow{x}$, and allows a tuple $\overrightarrow{v(x)}$ if and only if $\overrightarrow{\lambda} \prec_{leximin} \overrightarrow{v(x)}$.*

[4] A constraint is generalized arc-consistent if and only if for each variable x in the constraint, for each value $v \in d_x$, if x is assigned to v, then compatible values exist for all the other variables in the constraint.

Algorithm 2: Pseudo-code of an algorithm that computes a leximin-optimal solution of a constraints network using a leximin constraint.

inputs : a constraints network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; a vector $\langle u_1, \ldots, u_n \rangle$ of variables from $\mathcal{X}$

output: A solution to the problem [LEXIMIN-OPTIMAL] or "Inconsistent"

```
1 v ← solve(X, D, C);
2 while v ≠ "Inconsistent" do
3 |   v̂ ← v;
4 |   C ← C ∪ {Leximin(v̂(u⃗), u⃗)};
5 |   v ← solve(X, D, C);
6 if v̂ ≠ null then return v̂;
7 else return "Inconsistent";
```

The principle of the algorithm is rather simple. At the beginning, it computes a solution to the constraints network, and at each step it tries to find a better solution, as regards the strict leximin order, until the constraints network becomes inconsistent. Of course, the algorithm can be implemented in a branch-and-bound manner, *i.e.* without restarting the search process after a solution has been found (this is what we actually implemented).

Proposition 2 *If the function **solve** is correct and halts, then algorithm 2 halts and returns a solution to the problem* LEXIMIN-OPTIMAL.

Proof: If $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is inconsistent, then $\widehat{v}$ is never initialized, and thus the algorithm returns "Inconsistent". Otherwise, the loop 2..4 is entered at least once, and $\widehat{v}$ is initialized (thus preventing the algorithm from returning "Inconsistent"). In this case, it returns a complete consistent instantiation $\widehat{v}$ of the initial constraints network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, because it is consistent with an sup-set of $\mathcal{C}$ (and therefore with $\mathcal{C}$). Furthermore, if **solve** is correct, there is no other complete consistent instantiation v' of $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ such that $v'(\overrightarrow{u}) \succ_{leximin} \widehat{v}(\overrightarrow{u})$ (otherwise the last call to **solve** would not have returned "Inconsistent"). ■

4 A REAL-WORLD APPLICATION: SHARING A CONSTELLATION OF SATELLITES

We now describe the real-world application that originally motivated this work, and that allowed for carrying out realistic experiments and evaluations.

4.1 Problem description

The application concerns the common exploitation by several agents (countries, companies, civil or military agencies, etc.), of a constellation of Earth Observation Satellites (EOS). The mission of an EOS is to acquire images (photographies) of specified areas on the Earth surface, in response to observation demands from users, as illustrated on figure 1. Such a satellite is operated by an Image Programming and Processing Center, the role of which is to collect each day observation demands from users, and to schedule consequently the acquisitions for the next day. Thus the Center selects, among all the observation demands concerning the same day, those that will be satisfied,

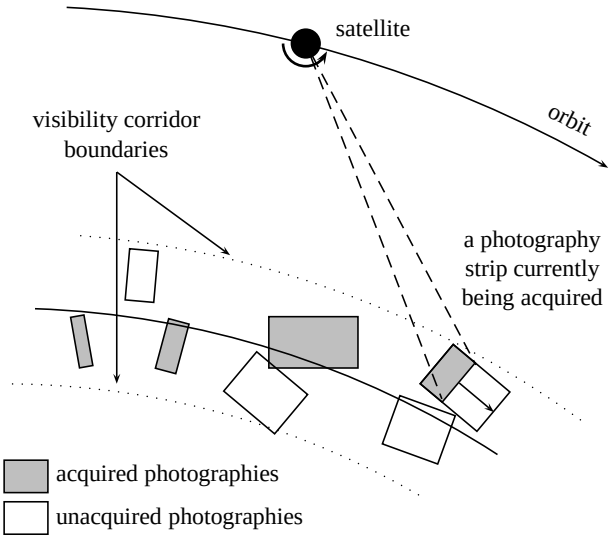


Figure 1. Acquisition of photography strips by an Earth Observation Satellite.

i.e. the set of photographies that will be acquired the next day by the constellation. This set of satisfied demands is therefore a day-to-day *allocation* of demands to agents.

The physical constraints and the huge number of demands concerning some specific areas generate some conflicts between demands. It is therefore generally impossible to satisfy simultaneously all the registered demands for the same day. This set of constraints defines the set of *admissible allocations*.

Here are some typical orders of magnitude concerning the real application: the number of agents generally lays between 3 and 6. Hundreds of demands are registered each day, among which about 100 to 200 will be satisfied.

The demands of an agent are of unequal importance. Each agent expresses the relative importance of its demands by giving them a *weight*, which is a positive or zero integer[5], and which implicitly formulates additive preferences: if two sets of demands have an equal sum of weights, then the concerned agent is indifferent between receiving the first set of demands or the second one. The *individual utility* of an allocation for an agent is the sum of the weights of the agent's demands that are satisfied by the allocation. A process of *normalization of utilities* – the detail of which is beyond the scope of this paper – is used, in order to make the individual utilities comparable. We only deal here with normalized utilities and weights.

All the agents did not equally fund the constellation, thus leading to unequal "rights to return on investments". There exist different ways to take unequal entitlements into account. One of the solutions, that we will use in this paper, is to translate this inequality by some consumption constraints: each agent has the right to a maximal amount of resource consumption, and this amount is different for each agent. These consumption constraints are added to the set of admissibility constraints.

Beyond these unequal entitlements, the allocation of demands to agents has to be fair. Several different solutions to this fair allocation problem has been proposed in [13] and in [4]. One of the proposed protocols to address fairness issues consists in choosing an allocation that maximizes the $\mathrm{leximin}$ order on individual utility profiles.

[5] A zero weight simply expresses the fact that an agent is not interested in the demand.

4.2 A fair allocation problem

We have extracted from this real-world application a simplified fair allocation problem. It can be viewed as an extension of problem [LEXIMIN-OPTIMAL], described in section 2.

A set of objects stands for the set of demands of our application. The conflicts between demands are approximately (but reasonably) represented by linear "generalized volume constraints". We have to notice that in this problem (1) all the objects will not necessarily be attributed, and (2) the same object can possibly be allocated to several agents[6].

Here is the formal description of this fair allocation problem. We describe first the inputs:

- $\mathcal{A}$ is a set of *agents*;
- $\mathcal{O}$ is a set of *objects* to attribute to agents;
- w_{io} is a positive or zero number representing the *weight* of object o according to agent i;
- r_o is the *resource consumption* of object o;
- $rmax_i$ is the *maximal amount of resource consumption* allowed for agent i;
- $\mathcal{C}$ is a set of *generalized volume constraints*;
- v_{co} is the *volume* of object o in the generalized volume constraint c;
- $vmax_c$ is the *maximal volume* in the generalized volume constraint c.

Then we define the following variables:

- $x_{io} = 1$ if object o is allocated to agent i, 0 otherwise, with $o \in \mathcal{O}$, $i \in \mathcal{A}$. The possible instantiations of the variables x_{io} stand for the set of possible allocations (among which the admissible ones are);
- $u_i = \sum_{o \in \mathcal{O}} x_{io} \cdot w_{io}$ is the individual utility of agent i, $i \in \mathcal{A}$;
- $s_o = \max_{i \in \mathcal{A}} x_{io} = 1$ is object o is allocated to at least one agent, 0 otherwise.

The problem is to find an allocation x maximizing the $\mathrm{leximin}$ order on the individual utility profiles $\langle u_i \rangle_{i \in \mathcal{A}}$, given the following admissibility constraints:

- $w_{io} = 0 \Rightarrow x_{io} = 0$ (if an agent gives weight 0 to an object, this object will not be allocated to her[7]);
- $\sum_{o \in \mathcal{O}} x_{io} \cdot r_o \leq rmax_i$, for all $i \in \mathcal{A}$ (constraints on the set of maximal consumptions of resources);
- $\sum_{o \in \mathcal{O}} s_o \cdot v_{co} \leq vmax_c$, for all $c \in C$ (generalized volume constraints).

Complexity of the problem Given an instance of the fair allocation problem, we write $\overrightarrow{u(x)}$ the individual utility profile resulting from the complete allocation x (*i.e.* in which all the x_{io} variables are instantiated). Let us consider the decision problem associated to the fair allocation problem, defined as follows: *Given an instance of the fair allocation problem and a utility profile $\overrightarrow{U}$, does an admissible allocation x (satisfying all the constraints) exist, such that $\overrightarrow{U} \preceq_{leximin} \overrightarrow{u(x)}$?*

When there is only one agent and only one (consumption or volume) constraint, we can easily recognize the [KNAPSACK] problem

[6] Which is indeed possible in our application.

[7] This constraint allows for selecting only the really relevant allocations.

[7, page 65], which is NP-complete. Our problem is of course therefore NP-complete.

We can also notice that it is a generalization of 0–1 multidimensional knapsack problem [18], but with a very particular optimization criterion.

4.3 Benchmark

We developed a random instance generator for the fair allocation problem described in subsection 4.2.

This generator[8] can be easily customized, using four groups of parameters:

- general parameters: number of agents, number of objects, random generator seed;
- objects weights parameters;
- consumption constraints parameters;
- generalized volume constraints parameters.

The weights of the objects are randomly generated. Two kinds of distributions are possible: a uniform distribution between 0 and w_{max}, and a distribution where the weights lay in different classes. The latter, corresponding approximately to the real case, is defined by two parameters: f_c (for example $f_c = 10$) which is the multiplicative factor between classes, and n_c (for example $n_c = 4$), the number of classes. The weights belonging to class $i \in [\![1, n_c]\!]$ are randomly chosen between $\frac{1}{2}(f_c)^i$ and $\frac{3}{2}(f_c)^i$. Moreover, we introduce a bias so that more demands from the low classes (less important) than those from the high classes will be generated.

As previously stated, consumption constraints allow for simulating unequal entitlements to the resource. In our generator, these unequal entitlements depend on two parameters: the entitlement of the agent having the lowest entitlement r_{min}, and the multiplicative factor between entitlements f_d. Agent i has the entitlement $r_{min} \times (f_d)^{i-1}$.

The following parameters concern the generalized volume constraints:

- constraints arity n_S (the same for each generalized volume constraint);
- maximum volume allowed for each constraint (fixed or random);
- volume of each object (fixed or random).

The generator also allows for instantiating these parameters by specifying the constraints tightness (which is in our cases the ratio between the number of forbidden objects and the arity of the constraint). The volume constraints hold on n_S consecutive objects.

4.4 Results

The two algorithms proposed in section 3 have been tested on our random instances. We used two different implementations of the first algorithm: the first one with the constraint programming library CHOCO [12], and the other one as an integer linear program using CPLEX [10]. The second algorithm has only been implemented in CHOCO. Our tests are based on an average instance with 4 agents having unequal entitlements, and 150 objects. Constraints are of average tightness: they forbid 10 objects out of 20 consecutive ones. The goals of the set of tests are to show the influence of the number of objects, the number of agents, and the distribution of the weights (uniformly distributed, or casted into different classes) on the solving time. For each value, the algorithms ran on 20 different random instances, with a time limit of 10 minutes per instance[9].

The first observation, looking at figure 2, is that CPLEX is much better than CHOCO on all the instances, which is not very surprising because all the constraints we have to deal with (even the cardinality combinator, if we make use of the trick presented in section 3) are linear and are therefore efficiently processed by CPLEX. Moreover, several parameters of CHOCO still need to be tuned, particularly those concerning heuristics[10]. However, CPLEX can potentially become incomplete if the weight distribution is to spread, certainly because CPLEX relaxes the problem and uses floating-point computations (this matter may probably be settled by customizing the optimization parameters of CPLEX). It can lead to two kinds of errors:

- the solution found by CPLEX is slightly different than the real one on some index of the leximin-optimal solution (which can induce a huge error on the next components);
- at some step, the call to function **maximize** returns "Inconsistent" (although it should not).

Figure 2(c) shows the number of errors from the second type (the number of errors from the first type is approximately the same).

Concerning the different weight distributions, the instances with uniformly distributed weights are clearly more difficult to solve than the other ones. The reason is certainly that when the weights are casted into different classes (say e.g. 4), solving such an instance is almost equivalent to solving 4 independent instances (one for each class), since having one object of some class is almost more important than having all the objects of its just inferior class.

Finally, one important issue is the comparison of the two constraint programming algorithms. As we can see in figures 2(b) and 2(d), the second algorithm (based on the multiset ordering constraint) is better for the instances with a non-uniform weight distribution; however there is a clear advantage to the other algorithm when the set of weights is uniformly distributed. Moreover, figure 2(e) shows that the second algorithm is far better than the first one when the number of agents increases. This is interesting because it can bring to light the areas of the parameter space where we should use one algorithm or the other one. More tests are required to entirely characterize these areas.

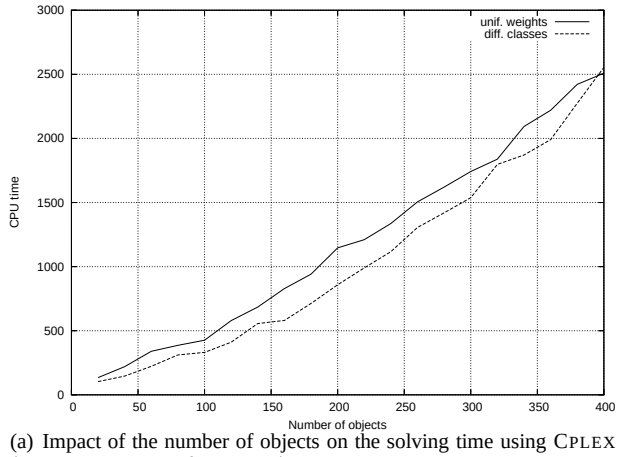
5 RELATED WORK

The article [1] describes an incomplete tabu search-like method for solving the multiagent problem of fairly allocating satellite resources. This version of the problem is actually very close to the real application introduced in section 4. It also takes into account most of the numerous operational constraints. In their version, the authors are also interested in finding a leximin-optimal allocation. However, the leximin order is not directly considered like we did in this paper, but is instead very closely represented by a collective utility function based on an Ordered Weighted Average operator [19]. Moreover, the solving techniques introduced in [1] are specifically dedicated to this precise problem, and to this kind of operational admissibility constraints.

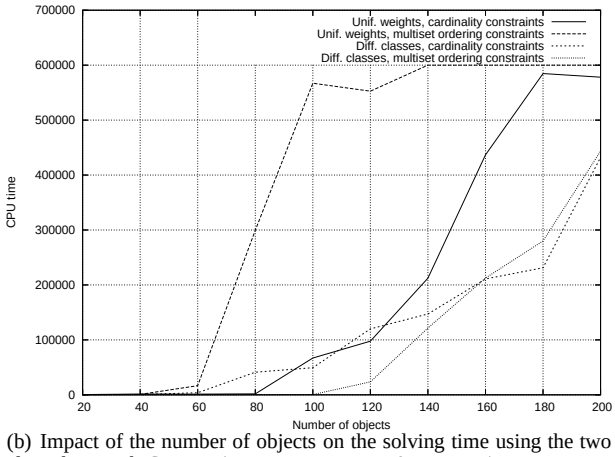
[8] Written in Java, and available at `http://www.cert.fr/dcsd/THESES/sbouveret/benchmark/index.html`

[9] Therefore the graphs showing solving times must be interpreted carefully, since the mean of the 20 solving times is biased if some instances have not been solved in 10 minutes.

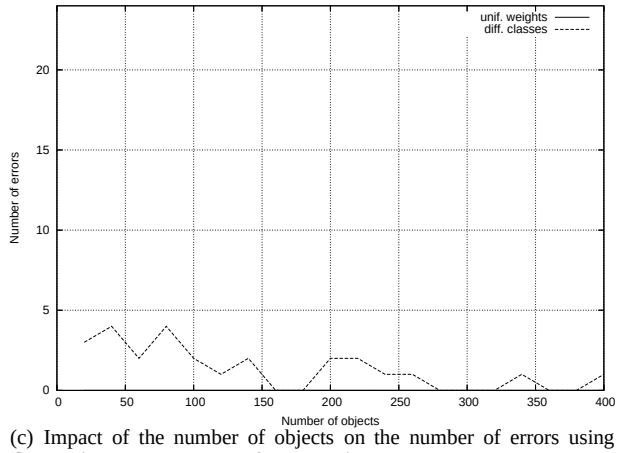
[10] We currently use a dynamic variable ordering trying to instantiate first the objects of higher weight of the current poorer agent. It seems to give better results than classical heuristics.



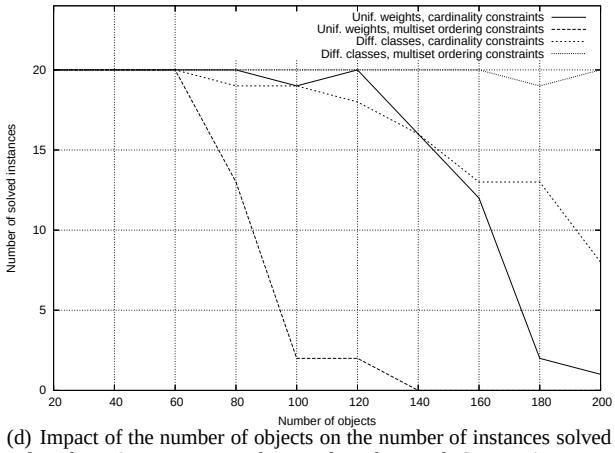
(a) Impact of the number of objects on the solving time using CPLEX (4 agents, mean on 20 instances)



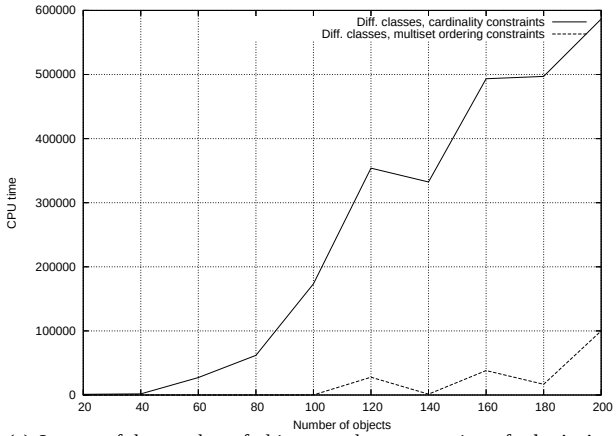
(b) Impact of the number of objects on the solving time using the two algorithms with CHOCO (4 agents, mean on 20 instances)



(c) Impact of the number of objects on the number of errors using CPLEX (4 agents, mean on 20 instances)



(d) Impact of the number of objects on the number of instances solved in less than 10 minutes using the two algorithms with CHOCO (4 agents, mean on 20 instances)



(e) Impact of the number of objects on the computation of a leximin-optimal allocation (10 agents, mean on 20 instances).

Figure 2. Impact of the number of objects on the computation of a leximin-optimal allocation (4 or 10 agents).

Computation of leximin-optimal solutions has some other application domains. We can find the leximin order, as indicated in section 1 as a refinement of the min operator in the study of fuzzy constraints [2, 5]. Two algorithms dedicated to the computation of leximin-optimal solutions has been published in [3]. These algorithms work by enumerating, at each step, all the subsets of fuzzy constraints (corresponding to our agents), so we think that their efficiency are questionable. However, these algorithms have to be experimented.

Even if fairness does not seem to be widely studied in combinatorial optimization, we can advert some work, among others, from Pesant and Régin [16], dealing with a global constraint dedicated to criteria balancing. The filtering procedures introduced in the latter paper are based on statistic rules. This approach is an appealing alternative to ours for dealing with equity. Applied to our problem, it should be linked with an optimization criterion that would also provide a kind of efficiency.

6 CONCLUSIONS AND FUTURE WORK

We studied in this paper the following *fair allocation* problem. A set of indivisible goods has to be allocated to a set of agents. The possible allocations are subject to some admissibility constraints. Each agent has its own utility function over the set of admissible allocations, and we have to select a fair admissible allocation. Fairness issues are expressed using the notion of leximin-*optimal* allocation over the agents utilities, which is a Pareto-optimal refinement of the maximin solution. This concept of leximin-optimal allocation potentially applies to each multiagent problem, where fairness is a key point.

The main contributions of this paper is the description of two algorithms dedicated to the computation of a leximin-optimal allocation, in a constraint programming framework. The first one is original, and is based on the cardinality meta-constraint **AtLeast**. The second one is a classical branch-and-bound based on the adaptation of an existing algorithm [6] that enforces global arc-consistency for a multiset ordering constraint. The constraint programming framework is particularly appealing here, as it allows for describing separately our particular algorithms on the one hand, and on the other hand the set of admissibility constraints, and the agents utility functions. This separation is all the more relevant since in real applications, most of the admissibility constraints are not fixed and do evolve with the application.

The two proposed algorithms are linked to a real-world application: fair sharing of satellite resources. This application has inspired a simplified multiagent allocation problem, for which we developed a random instance generator, whose generated instances are significantly close to real instances. Using this generator we have been able to test two distinct implementations of the first algorithm (one with the constraint programming tool CHOCO [12], and the other using the integer linear programming tool CPLEX 10.0 [10]), and one for the second algorithm, using CHOCO. The aim was to compare our two CP algorithms on a simple problem. The results of the implementations using CHOCO clearly show that none of the two algorithms is always better than the other one, and that it depends on the input parameters of the instance to be solved. The results also show that CPLEX is clearly better than CHOCO on this particular problem, but this is not very surprising because our problem particulary fits for ILP modelling.

This paper is an algorithmic approach, among other possible ones that may be more efficient. We have only studied here exact methods. More difficult instances may request the use of incomplete methods like those of [17], or [18, 1], mixing linear programming and tabu search. We hope that our random benchmark generator will allow people interested in the problem to propose and validate other approaches.

Among the possible extensions of this work, we can suggest:

- studying and using softer and more general modeling of the fairness requirement, replacing the leximin order by a parameterized collective utility function realizing some compromises between egalitarianism and classical utilitarianism;
- applying our algorithms to other practical fields, like computation of balanced timetables or sharing of airspace and airport resources.

REFERENCES

[1] N. Bianchessi, J.-F. Cordeau, J. Desrosiers, G. Laporte, and V. Raymond, 'A heuristic for the multi-satellite, multi-orbit and multi-user management of earth observation satellites', *European Journal of Operational Research*, (available online February 2006). doi:10.1016/j.ejor.2005.12.026.

[2] D. Dubois, H. Fargier, and H. Prade, 'Refinements of the maximin approach to decision-making in fuzzy environment', *Fuzzy Sets and Syst.*, **81**, 103–122, (1996).

[3] D. Dubois and P. Fortemps, 'Computing improved optimal solutions to max-min flexible constraint satisfaction problems', *European Journal of Operational Research*, (1999).

[4] H. Fargier, J. Lang, M. Lemaître, and G. Verfaillie, 'Partage équitable de ressources communes. (1) Un modèle général et son application au partage de ressources satellitaires. (2) éléments de complexité et d'algorithmique', *Technique et Science Informatiques*, **23**(9), 1187–1238, (2004).

[5] H. Fargier, J. Lang, and T. Schiex, 'Selecting preferred solutions in fuzzy constraint satisfaction problems', in *Proc. of EUFIT'93*, Aachen, (1993).

[6] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh, 'Multiset ordering constraints', in *Proc. of IJCAI'03*, (February 2003).

[7] M. R. Garey and D. S. Johnson, *Computers and Intractability, a guide to the theory of NP-completeness*, Freeman, 1979.

[8] R. S. Garfinkel and G. L. Nemhauser, *Integer Programming*, Wiley-Interscience, 1972.

[9] P. Van Hentenryck, H. Simonis, and M. Dincbas, 'Constraint satisfaction using constraint logic programming', *Artificial Intelligence*, **58**(1-3), 113–159, (1992).

[10] ILOG. Cplex 10.0. `http://www.ilog.com/products/cplex/`.

[11] R. L. Keeney and H. Raiffa, *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*, John Wiley and Sons, 1976.

[12] F. Laburthe, 'CHOCO : Implémentation du noyau d'un système de contraintes', in *Actes des JNPC-00*, Marseille, France, (2000). `http://sourceforge.net/projects/choco`.

[13] M. Lemaître, G. Verfaillie, and N. Bataille, 'Exploiting a Common Property Resource under a Fairness Constraint: a Case Study', in *Proc. of IJCAI-99*, pp. 206–211, Stockholm, (1999).

[14] H. Moulin, *Axioms of Cooperative Decision Making*, Cambridge University Press, 1988.

[15] H. Moulin, *Fair division and collective welfare*, MIT Press, 2003.

[16] G. Pesant and J-C. Régin, 'SPREAD: A balancing constraint based on statistics', in *Proc. of CP'05*, Sitges, Spain, (2005).

[17] M. Vasquez and Jin-Kao Hao, 'A logic-constrained knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite', *Journal of Computational Optimization and Applications*, **20**(2), 137–157, (2001).

[18] M. Vasquez and J.K. Hao, 'A Hybrid Approach for the 0–1 Multidimensional Knapsack Problem', in *Proc. of IJCAI-01*, volume 1, pp. 328–333, (August 2001).

[19] R. Yager, 'On ordered weighted averaging aggregation operators in multicriteria decision making', *IEEE Transactions on Systems, Man, and Cybernetics*, **18**, 183–190, (1988).