Finding leximin-optimal solutions using constraint programming: new algorithms and their application to combinatorial auctions

Sylvain Bouveret and Michel Lemaître

Abstract

We study the problem of computing a leximin-optimal solution of a constraint network. This problem is highly motivated by fairness and efficiency requirements in many real-world applications implying human agents. We compare several generic algorithms which solve this problem in a constraint programming framework. The second one is entirely original, and the other ones are partially based on existing works adapted to fit with this problem. These algorithms are tested on combinatorial auctions instances.

1 Introduction

Many advances have been done in recent years in modeling and solving combinatorial problems with constraint programming (CP). These advances concern, among others, the ability of this framework to deal with human reasoning schemes, such as, for example, the expression of preferences with soft constraints. However, one aspect of importance has only received little attention in the constraints community to date: the way to handle fairness requirements in multiagent combinatorial problems.

The seek for fairness stands as a subjective but strong requirement in a wide set of real-world problems implying human agents. It is particularly relevant in crew or worker timetabling and rostering problems, or the optimization of long and short-term planning for firemen and emergency services. Fairness is also ubiquitous in multiagent resource allocation problems, like, among others, bandwidth allocation among network users, fair share of airspace and airport resources among several airlines or Earth observing satellite scheduling and sharing problems [11].

In spite of the wide range of problems concerned by fairness issues, it often lacks a theoretical and generic approach. In many Constraint Programming and Operational Research works, fairness is only enforced by specific heuristic local choices guiding the search towards supposed equitable solutions. However, a few works may be cited for their approach of this fairness requirement. [11] make use of an Earth observation satellite scheduling and sharing problem to investigate three ways of handling fairness among agents in the context of constraint satisfaction. More recently [18] proposed a new constraint based on statistics, which enforces the relative balance of a given set of variables, and can possibly be used to ensure a kind of equity among a set of agents. Equity is also studied in Operational Research, with for example [17], who investigate a way of solving linear programs by aggregating multiple criteria using an Ordered Weighted Average Operator (OWA) [22]. Depending on the weights used in the OWA, this kind of aggregators can provide equitable compromises.

Microeconomy and Social Choice theory provide an important literature on fairness in collective decision making. From this theoretical background we borrow the idea of representing the agents preferences by *utility* levels, and we adopt the *leximin* preorder on utility profiles for conveying the fairness and efficiency requirements.

Apart from the fact that it conveys and formalizes the concept of equity in multiagent contexts, the leximin preorder is also a subject of interest in other contexts, such as fuzzy CSP [6], and symmetry-breaking in constraint satisfaction problems [7].

This contribution is organized as follows. Section 2 gives a minimal background in social choice theory and justifies the interest of the leximin preorder as a fairness criterion. Section 3 defines the search for leximin-optimality in a constraint programming framework. The main contribution of this paper is Section 4, which presents three algorithms for computing leximin-optimal solutions, the first one being entirely original, and the other ones adapted from existing works. The proposed algorithms have been implemented and tested within a constraint programming system. Section 5 presents an experimental comparison of these algorithms[1].

2 Background on social choice theory

We first introduce some notations. Calligraphic letters (*e.g.* $\mathcal{X}$) will stand for sets. Vectors will be written with an arrow (*e.g.* $\overrightarrow{x}$), or between brackets (*e.g.* $\langle x_1, \dots, x_n \rangle$). $f(\overrightarrow{x})$ will be used as a shortcut for $\langle f(x_1), \dots, f(x_n) \rangle$. Vector $\overrightarrow{x}^{\uparrow}$ will stand for the vector composed by each element of $\overrightarrow{x}$ rearranged in increasing order. We will write $x_i^{\uparrow}$ for the i^{th} component of vector $\overrightarrow{x}^{\uparrow}$. Finally, the interval of integers between k and l will be written $[\![k, l]\!]$.

2.1 Collective decision making and welfarism

Let $\mathcal{N}$ be a set of n agents, and $\mathcal{S}$ be a set of admissible alternatives concerning all of them, among which a benevolent arbitrator has to choose one. The most classical model describing this situation is *welfarism* (see *e.g.* [9, 15]): the choice of the arbitrator is made on the basis of the utility levels enjoyed by the individual agents and on those levels only. Each agent $i \in \mathcal{N}$ has an individual utility function u_i that maps each admissible alternative $s \in \mathcal{S}$ to a numerical

[1] A similar paper is going to appear in the proceedings of IJCAI'07 with the section 5 based on a different application.

index $u_i(s)$. We make here the classical assumption that the individual utilities are comparable between the agents[2]. Therefore each alternative s can be attached to a single *utility profile* $\langle u_1(s), \ldots, u_n(s)\rangle$. According to welfarism, comparing two alternatives is performed by comparing their respective utility profiles.

A standard 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 $uc(s) = g(u_1(s), \ldots, u_n(s))$. An optimal alternative is one of those maximizing the collective utility.

2.2 The leximin order as a fairness and efficiency criterion

The main difficulty of equitable decision problems is that we have to reconcile the contradictory wishes of the agents. Since generally no solution fully satisfies everyone, the aggregation function g must lead to fair and Pareto-efficient[3] compromises.

The problem of choosing the right aggregation function g is far beyond the scope of this paper. We only describe the two classical ones corresponding to two opposite points of view on social welfare[4]: classical utilitarianism and egalitarianism. The rule advocated by the defenders of classical utilitarianism is that the best decision is the one that maximizes the sum of individual utilities (thus corresponding to $g = +$). However this kind of aggregation function can lead to huge differences of utility levels among the agents, thus ruling out this aggregator in the context of equitable decisions. From the egalitarian point of view, the best decision is the one that maximizes the happiness of the least satisfied agent (thus corresponding to $g = \min$). Whereas this kind of aggregation function is particularly well-suited for problems in which fairness is essential, it has a major drawback, due to the idempotency of the min operator, and known as "drowning effect" in the community of fuzzy CSP (see *e.g.*[4]). Indeed, it leaves many alternatives indistinguishable, such as for example the ones with utility profiles $\langle 0, \ldots, 0\rangle$ and $\langle 1000, \ldots, 1000, 0\rangle$, even if the second one appears to be much better than the first one. In other words, the min aggregation function can lead to non Pareto-optimal decisions, which is not desirable.

The leximin preorder is a well-known refinement of the order induced by the min function that overcomes this drawback. It is classically introduced in the social choice literature (see [15]) as the social welfare ordering that reconcile egalitarianism and Pareto-efficiency, and also in fuzzy CSP [6]. It is defined as follows:

[2]In other words, they are expressed using a common utility scale.

[3]A decision is Pareto-efficient if and only if we cannot strictly increase the satisfaction of an agent unless we strictly decrease the satisfaction of another agent. Pareto-efficiency is generally taken as a basic postulate in collective decision making.

[4]Compromises between these two extremes are possible. See *e.g.* [16, page 68] or [22] (*OWA aggregators*).

Definition 1 (leximin preorder [15]) *Let $\overrightarrow{x}$ and $\overrightarrow{y}$ be two vectors of $\mathbb{N}^n$. $\overrightarrow{x}$ and $\overrightarrow{y}$ are said leximin-indifferent (written $\overrightarrow{x} \sim_{leximin} \overrightarrow{y}$) if and only if $\overrightarrow{x}^{\uparrow} = \overrightarrow{y}^{\uparrow}$. The vector $\overrightarrow{y}$ is 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 other words, the leximin preorder is the lexicographic preorder over ordered utility vectors. For example, we have $\langle 4, 1, 5, 1\rangle \prec_{leximin} \langle 2, 2, 1, 2\rangle$.

A known result is that no collective utility function can represent the leximin preorder[5], unless the set of possible utility profiles is finite. In this latter case, it can be represented by the following non-linear functions: $g_1 : \overrightarrow{x} \mapsto -\sum_{i=i}^{n} n^{-x_i}$ (adapted for leximin from a remark in [7]) and $g_2 : \overrightarrow{x} \mapsto -\sum_{i=1}^{n} x_i^{-q}$, where $q > 0$ is large enough [15]. The major drawback of using this kind of representation is that it rapidly becomes unreasonable to use it when the upper bound of the possible values of $\overrightarrow{x}$ increases. Moreover, it hides the semantics of the leximin preorder, and hinders the computational benefits we could possibly take advantage of.

In the following, we will use the leximin preorder as a criterion for ensuring fairness and Pareto-efficiency, and we will seek the non-dominated solutions in the sense of the leximin preorder. Those solutions will be called leximin-optimal. This problem will be expressed in the next section in a CP framework.

3 Leximin and Constraint programming

The constraint programming framework is an effective and flexible tool for modeling and solving many different combinatorial problems such as planning and scheduling problems, resource allocation problems, or configuration problems. This paradigm is based on the notion of *constraint network* [14]. A constraint network consists of a set of variables $\mathcal{X} = \{x_1, \dots, x_p\}$, a set of associated domains $\mathcal{D} = \{d_{x_1}, \dots, d_{x_p}\}$, d_{x_i} being the set of possible values for x_i, and a set of constraints $\mathcal{C}$, where each $C \in \mathcal{C}$ specifies a set of allowed tuples $R(C)$ over a set of variables $X(C)$. We will also suppose that all the domains are in $\mathbb{N}$, and use the following notations: $\underline{x} = \min(d_x)$ and $\overline{x} = \max(d_x)$.

An instantiation v of a set $\mathcal{S}$ 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 $\mathcal{S}' \subsetneq \mathcal{S}$, the projection of an instantiation of $\mathcal{S}$ over $\mathcal{S}'$ is the restriction of this instantiation to $\mathcal{S}'$ and is written $v_{\downarrow \mathcal{S}'}$. An instantiation is said to be consistent if and only if it satisfies all the constraints. A complete consistent instantiation of a constraint network is called a solution. The set of solutions of $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is written $sol(\mathcal{X}, \mathcal{D}, \mathcal{C})$.

Given a constraint network, the problem of determining whether it has a solution is called a Constraint Satisfaction Problem (CSP) and is NP-complete.

[5]In other words there is no g such that $\overrightarrow{x} \preceq_{leximin} \overrightarrow{y} \Leftrightarrow g(\overrightarrow{x}) \leq g(\overrightarrow{y})$. See [15].

The CSP can be classically adapted to become an optimization problem in the following way. Given a constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ and an *objective* variable $o \in \mathcal{X}$, find the value M of d_o such that $M = \max\{v(o) \mid v \in sol(\mathcal{X}, \mathcal{D}, \mathcal{C})\}$. We will write $max(\mathcal{X}, \mathcal{D}, \mathcal{C}, o)$ for the subset of those solutions that maximize the objective variable o.

Expressing a collective decision making problem with a numerical collective utility criterion as a CSP with objective variable is straightforward: consider the collective utility as the objective variable, and link it to the variables representing individual utilities with a constraint. However this cannot directly encode our problem of computing a leximin-optimal solution, which is a kind of multicriteria optimization problem. We introduce formally the MaxLeximinCSP problem as follows :

Definition 2 (Problem MaxLeximinCSP)
Input: *a constraint network* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$*; a vector of variables* $\overrightarrow{u} = \langle u_1, \ldots, u_n \rangle \in \mathcal{X}^n$*, called the* objective vector.
Output: *"Inconsistent" if* $sol(\mathcal{X}, \mathcal{D}, \mathcal{C}) = \emptyset$*. Otherwise a solution* $\widehat{v}$ *such that* $\forall v \in sol(\mathcal{X}, \mathcal{D}, \mathcal{C})$, $v(\overrightarrow{u}) \preceq_{leximin} \widehat{v}(\overrightarrow{u})$.

We describe in the next section several generic constraint programming algorithms that solve this problem. The second one is entirely original, and the other ones are based on existing works that are adapted to fit with our problem.

4 Proposed algorithms

4.1 Using a sorting constraint

Our first algorithm is directly based on the definition 1 of the leximin preorder, which requires to sort the vectors to be compared before performing a lexicographic comparison. We can therefore introduce, using additional variables, the sorted version of the objective vector. This can be done naturally in the CP paradigm by introducing a vector of variables $\overrightarrow{y}$ and enforcing the constraint **Sort**$(\overrightarrow{u}, \overrightarrow{y})$ which is defined as follows:

Definition 3 (Constraint Sort) *Let* $\overrightarrow{x}$ *and* $\overrightarrow{x}'$ *be two vectors of variables of the same length, and* v *be an instantiation. The constraint* **Sort**$(\overrightarrow{x}, \overrightarrow{x}')$ *holds on the set of variables being either in* $\overrightarrow{x}$ *or in* $\overrightarrow{x}'$*, and is satisfied by* v *if and only if* $v(\overrightarrow{x}')$ *is the sorted version of* $v(\overrightarrow{x})$ *in increasing order.*

This constraint has been particularly studied in two works, which both introduce a filtering algorithm for enforcing bound consistency on this constraint. The first algorithm comes from [1] and runs in $O(n \log n)$ (n being the size of $\overrightarrow{x}$). [13] designed a simpler algorithm that runs in $O(n)$ plus the time required to sort the interval endpoints of $\overrightarrow{x}$, which can asymptotically be faster than $O(n \log n)$.

The algorithm 1 intuitively works as follows : having introduced the sorted version $\overrightarrow{y}$ of the objective vector $\overrightarrow{u}$, it successively maximizes the components of this vector, provided that the leximin-optimal solution is the solution that maximizes y_1, and, given this maximal value, maximizes y_2, and so on until y_n.

Algorithm 1: Solving the MaxLeximinCSP using a sorting constraint.

```
input  : A const. network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; $\langle u_1, \ldots, u_n \rangle \in \mathcal{X}^n$
output: A solution to the MaxLeximinCSP problem

if solve$(\mathcal{X}, \mathcal{D}, \mathcal{C})$ = "Inconsistent" return "Inconsistent";
$\mathcal{X}' \leftarrow \mathcal{X} \cup \{y_1, \ldots, y_n\}$;
$\mathcal{D}' \leftarrow \mathcal{D} \cup \{d_{y_1}, \ldots, d_{y_n}\}$ with $d_{y_i} = [\![\min_j(\underline{u_j}), \max_j(\overline{u_j})]\!]$;
$\mathcal{C}' \leftarrow \mathcal{C} \cup \{\mathbf{Sort}(\overrightarrow{u}, \overrightarrow{y})\}$;
for $i \leftarrow 1$ to $n$ do
    $\widehat{v}_{(i)} \leftarrow \mathbf{maximize}(\mathcal{X}', \mathcal{D}', \mathcal{C}', y_i)$;
    $d_{y_i} \leftarrow \{\widehat{v}_{(i)}(y_i)\}$;
return $\widehat{v}_{(n)\downarrow\mathcal{X}}$;
```

In the algorithm 1 (and in the following ones also), the functions **solve** and **maximize** (the detail of which is the concern of solving techniques for constraints satisfaction problems) respectively return one solution $v \in sol(\mathcal{X}, \mathcal{D}, \mathcal{C})$ (or "Inconsistent" if such a solution does not exist), and an optimal solution $\widehat{v} \in max(\mathcal{X}_i, \mathcal{D}_i, \mathcal{C}_i, y_i)$ (or "Inconsistent" if $sol(\mathcal{X}_i, \mathcal{D}_i, \mathcal{C}_i) = \emptyset$). We assume – contrary to usual constraint solvers – that these two functions do not modify the input constraint network.

Proposition 1 *If the two functions* **maximize** *and* **solve** *are both correct and both halt, then algorithm 1 halts and solves the* MaxLeximinCSP *problem.*

Proof: If $sol(\mathcal{X}, \mathcal{D}, \mathcal{C}) = \emptyset$ and if **solve** is correct, then algorithm 1 obviously returns "Inconsistent". We will suppose in the following that $sol(\mathcal{X}, \mathcal{D}, \mathcal{C}) \neq \emptyset$ and we will use the following notations: $\mathcal{S}_i$ and $\mathcal{S}'_i$ are the sets of solutions of $(\mathcal{X}', \mathcal{D}', \mathcal{C}')$ respectively at the beginning and at the end of iteration i.

We have obviously $\forall i \in [\![1, n-1]\!]$ $\mathcal{S}_{i+1} = \mathcal{S}'_i$, which proves that if $\mathcal{S}_i \neq \emptyset$, then the call to **maximize** at line 1 does not return "Inconsistent", and $\mathcal{S}_{i+1} \neq \emptyset$. Thus $\widehat{v}_{(n)}$ is well-defined, and obviously $(\widehat{v}_{(n)})_{\downarrow\mathcal{X}}$ is a solution of $(\mathcal{X}, \mathcal{D}, \mathcal{C})$.

We note $\widehat{v} = \widehat{v}_{(n)}$ the instantiation computed by the last **maximize** in algorithm 1. Suppose that there is an instantiation $v \in sol(\mathcal{X}, \mathcal{D}, \mathcal{C})$ such that $\widehat{v}(\overrightarrow{u}) \prec_{leximin} v(\overrightarrow{u})$. We define v^+ the extension of v that instantiates each y_i to $v(\overrightarrow{u})^{\uparrow}_i$. Then, due to constraint **Sort**, $\widehat{v}(\overrightarrow{y})$ and $v^+(y)$ are the respective sorted version of $\widehat{v}(\overrightarrow{u})$ and $v^+(u)$. Following definition 1, there is an $i \in [\![0, n-1]\!]$ such that $\forall j \in [\![1, i]\!]$, $\widehat{v}(y_j) = v^+(y_j)$ and $\widehat{v}(y_{i+1}) < v^+(y_{i+1})$. Due to line 1, we have $\widehat{v}(y_{i+1}) = \widehat{v}_{(n)}(y_{i+1}) = \widehat{v}_{(i+1)}(y_{i+1})$. Thus v^+ is a solution in $max(\mathcal{X}', \mathcal{D}', \mathcal{C}', y_{i+1})$ with objective value $v^+_{(i+1)}(y_{i+1})$ strictly greater than $\widehat{v}_{(i+1)}(y_{i+1})$, which contradicts the hypothesis about **maximize**. ■

4.2 Using a cardinality combinator

Our second algorithm is based on an alternative definition of the sorting of the objective vector. In fact, it can be noticed that, given two vectors of numbers $\overrightarrow{x}$ and $\overrightarrow{x}'$, $\overrightarrow{x}'$ is the sorted version of $\overrightarrow{x}$ in increasing order if and only if for all i, x'_i is the maximal value such that at least $n-i+1$ values from vector $\overrightarrow{x}$ are greater than or equal to x'_i.

Like the first algorithm, this algorithm works by successively computing the sorted components of the leximin-optimal objective vector, but contrary to the first one, this new algorithm does not expliciteely introduce the sorted version of the objective vector. This new algorithm informally works as follows. It first computes the maximal value y_1 such that there is a solution v with $\forall i$, $y_1 \leq v(u_i)$, or in other words $\sum_i (y_1 \leq v(u_i)) = n$, where by convention the value of $(y_1 \leq v(u_i))$ is 1 if the inequality is satisfied and 0 otherwise[6]. Then, after having fixed this value for y_1, it computes the maximal value y_2 such that there is a solution v with $\sum_i (y_2 \leq v(u_i)) \geq n-1$, and so on until the maximal value y_n such that there is a solution v with $\sum_i (y_n \leq v(u_i)) \geq 1$.

To enforce the constraint on the y_i, we make use of the meta-constraint **AtLeast**, derived from a *cardinality combinator* introduced by [21], and present in most CP systems:

Definition 4 (Meta-constraint AtLeast) *Let Γ be a set of p constraints, and $k \in [\![1,p]\!]$ be an integer. The meta-constraint* $\mathbf{AtLeast}(\Gamma, k)$ *holds on the union of the scopes of the constraints in Γ, and allows a tuple if and only if at least k constraints from Γ are satisfied.*

Due to its genericity, this meta-constraint cannot provide very efficient filtering procedures. Fortunately, in our case where each constraint in Γ is of the form $x_i \geq y$, bound-consistency can be enforced using algorithm 2.

Algorithm 2: Enforcing bound-consistency on the **AtLeast** meta-constraint with linear constraints.

```
input : A vector of variables $\langle x_1, \ldots, x_n \rangle$, a variable $y$, an integer $k \leq n$.
output: The domain reductions of $\langle x_1, \ldots, x_n \rangle$ and $y$ to enforce bound
        consistency on $\mathbf{AtLeast}(\{x_1 \geq y, \ldots, x_n \geq y\}, k)$, or "Inconsistent"
1 $\overline{y} \leftarrow (\sup(\overrightarrow{x}))^{\uparrow}_{n-k+1}$ ;          /* where $\sup(\overrightarrow{x}) = \langle \overline{x_1}, \ldots, \overline{x_n} \rangle$ */
2 if $\sum_i (\overline{x_i} < \underline{y}) > n-k$ return "Inconsistent";
3 if $\sum_i (\overline{x_i} < \underline{y}) = n-k$
4     forall $i$ such that $\overline{x_i} \geq \underline{y}$ do $\underline{x_i} \leftarrow \max(\underline{y}, \underline{x_i})$
```

This algorithm runs in $O(n)$, since the selection of the $n-k+1^{\text{st}}$ lowest value of $\sup(\overrightarrow{x})$ can be done in $O(n)$ [2]. We can notice that this algorithm is well-suited for event-based implementation of constraint programming: in case of an update of one of the $\overline{x_i}$, only line 1 needs to be run ; in case of an update

[6]This convention is inspired by the constraint modeling language OPL [20].

of $\underline{y}$, only lines 2 and 3 need to be run ; any other update do not need the algorithm to be run. The procedure can also benefit from storing the ordered vector $(\sup(\overrightarrow{x}))^{\uparrow}$ and updating it when one of the $\overline{x_i}$ changes. By doing so, we can access $(\sup(\overrightarrow{x}))^{\uparrow}_{n-k+1}$ in $O(1)$.

It can also be noticed that since all of the constraints of Γ are linear, the meta-constraint **AtLeast** can be expressed using a set of linear constraints, therefore allowing our algorithm to be processed with a linear solver. The classical idea [8, p.11] is to express our constraint **AtLeast** by introducing n 0–1 variables $\{\delta_1, \ldots, \delta_n\}$, and a set of linear constraints $\{x_1+\delta_1\overline{y} \geq y, \ldots, x_n+\delta_n\overline{y} \geq y, \sum_{i=1}^{n} \delta_i \leq n-k\}$.

This second approach is presented in algorithm 3.

Algorithm 3: Solving the MaxLeximinCSP using a cardinality constraint.

```
input  : A const. network $(\mathcal{X},\mathcal{D},\mathcal{C})$; $\langle u_1,\ldots,u_n\rangle \in \mathcal{X}^n$
output: A solution to the MaxLeximinCSP problem
1 if solve$(\mathcal{X},\mathcal{D},\mathcal{C})$ = "Inconsistent" return "Inconsistent";
2 $(\mathcal{X}_0,\mathcal{D}'_0,\mathcal{C}_0) \leftarrow (\mathcal{X},\mathcal{D},\mathcal{C})$;
3 for $i \leftarrow 1$ to $n$ do
4     $\mathcal{X}_i \leftarrow \mathcal{X}_{i-1} \cup \{y_i\}$;
5     $\mathcal{D}_i \leftarrow \mathcal{D}'_{i-1} \cup \{d_{y_i}\}$ with $d_{y_i} = [\![\min_j(\underline{u_j}), \max_j(\overline{u_j})]\!]$;
6     $\mathcal{C}_i \leftarrow \mathcal{C}_{i-1} \cup \{$AtLeast$(\{y_i \leq u_1,\ldots,y_i \leq u_n\}, n-i+1)\}$;
7     $\widehat{v}_{(i)} \leftarrow$ maximize$(\mathcal{X}_i,\mathcal{D}_i,\mathcal{C}_i,y_i)$;
8     $\mathcal{D}'_i \leftarrow \mathcal{D}_i$ with $d_{y_i} \leftarrow \{\widehat{v}_{(i)}(y_i)\}$;
9 return $\widehat{v}_{(n)\downarrow\mathcal{X}}$;
```

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

The following example illustrates the behavior of the algorithm. It is a simple resource allocation problem, where 3 objects must be allocated to 3 agents, with the following constraints: each agent must get one and only one object, and one object cannot be allocated to more than one agent (i.e. a perfect matching agent/objects). A utility is associated with each pair (agent,object) with respect to the array above.

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

	p_1	p_2	p_3	p_4	p_5	p_6
u_1	3	3	5	5	7	7
u_2	9	8	3	8	3	9
u_3	1	7	1	3	7	3

The algorithm runs in 3 steps: **Step 1:** After having introduced one variable y_1, 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 $\widehat{y_1} = 3$. The variable y_1 is fixed to this value, implicitly removing profiles p_1 and p_3. **Step 2:** After having introduced one variable y_2, we look for the maximal value $\widehat{y_2}$ of y_2 such that **at least** 2 agents get at least y_2. We find $\widehat{y_2} = 7$. The variable y_2 is fixed to this value, implicitly removing profile p_4. **Step 3:** After having introduced one variable y_3, we look for the maximal value $\widehat{y_3}$ of y_3 such that **at least** 1 agent gets at least y_3. We find $\widehat{y_3} = 9$. Only one instantiation maximizes y_3: p_6. Finally, the returned leximin-optimal allocation is: $a_1 \leftarrow o_3$, $a_2 \leftarrow o_2$ and $a_3 \leftarrow o_1$.

Proposition 2 *If the two functions* **maximize** *and* **solve** *are both correct and both halt, then algorithm 3 halts and solves the* MaxLeximinCSP *problem.*

The complete proof of this proposition can be found in the article published in the proceedings of IJCAI'07. We just give here a proof sketch.

Proof sketch: The proposition can be proved using the following steps.

- We first prove the initial remark : if $\overrightarrow{x}$ is a vector of size n, then at least $n-i+1$ components of $\overrightarrow{x}$ are greater than or equal to $x_i^{\uparrow}$.
- Then we must prove that if the initial constraint network has a solution then $\widehat{v}_{(n)}$ is well-defined and not equal to "Inconsistent".
- We then prove that $\widehat{v}_{(n)}(\overrightarrow{y})$ is equal to $\widehat{v}_{(n)}(\overrightarrow{u})^{\uparrow}$ if $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ has a solution.
- By putting things together, we can finally prove that $\widehat{v}_{(n)\downarrow\mathcal{X}}$ is really the leximin-optimal solution, using the fact that if there was a better solution (in the sense of the leximin preorder), the call to **maximize** at some iteration would have eliminated the solution actually returned by the algorithm. ■

4.3 Using a multiset ordering constraint

Our third algorithm computing a leximin-optimal solution is probably the most intuitive one. This algorithm proceeds in a pseudo branch and bound manner: it computes a first solution, then it tries to improve it by specifying that the next solution has to be better (in the sense of the leximin preorder) than the current one, and so on until the constraint network becomes inconsistent. This approach is based on the following constraint:

Definition 5 (Constraint Leximin) *Let $\overrightarrow{x}$ be a vector of variables, $\overrightarrow{\lambda}$ be a vector of integers, and v be an instantiation. The constraint* **Leximin**$(\overrightarrow{\lambda}, \overrightarrow{x})$ *holds on the set of variables belonging to $\overrightarrow{x}$, and is satisfied by v if and only if $\overrightarrow{\lambda} \prec_{leximin} \overrightarrow{v(x)}$.*

Although this constraint does not exist in the literature, the work of [7] introduces an algorithm for enforcing generalized arc-consistency on a quite similar constraint: the multiset ordering constraint, which is, in the context of multisets, the equivalent of a leximax[7] constraint on vectors of variables. At the price of some slight modifications, the algorithm they introduce can easily be used to enforce the latter constraint **Leximin**.

Proposition 3 *If the function* **solve** *is correct and halts, then algorithm 4 halts and solves the* MaxLeximinCSP *problem.*

The proof is rather straightforward, so we omit it.

[7]The leximax is based on an increasing reordering of the values, instead of a decreasing one for leximin.

Algorithm 4: Solving the MaxLeximinCSP using a constraint **Leximin**.

```
input  : A const. network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; $\langle u_1, \ldots, u_n \rangle \in \mathcal{X}^n$
output: A solution to the MaxLeximinCSP problem
1 $\widehat{v} \leftarrow$ null; $v \leftarrow$ solve$(\mathcal{X}, \mathcal{D}, \mathcal{C})$;
2 while $v \neq$ "Inconsistent" do
3     $\widehat{v} \leftarrow v$;
4     $\mathcal{C} \leftarrow \mathcal{C} \cup \{$Leximin$(\widehat{v}(\overrightarrow{u}), \overrightarrow{u})\}$;
5     $v \leftarrow$ solve$(\mathcal{X}, \mathcal{D}, \mathcal{C})$;
6 if $\widehat{v} \neq$ null then return $\widehat{v}$ else return "Inconsistent";
```

4.4 Other approaches

In the context of fuzzy constraints, two algorithms dedicated to the computation of leximin-optimal solutions have been published by [4]. These algorithms work by enumerating, at each step, all the subsets of fuzzy constraints (corresponding to our agents) having a property connected to the notion of consistency degree.

[5, p. 162] describes two very simple algorithms for solving the closely related "Lexicographic Max-Ordering" problem (in our terms, finding the "leximax-optimal"). They however do not seem realistic in the context of combinatorial problems, since they are based on an enumeration of all utility profiles.

5 Experimental results

Combinatorial auctions[3, 19] – auctions in which bidders place unrestricted bids for bundles of goods – are subject of increasing study in the recent years. Their central problem is the *Winner Determination Problem* (WDP), which has been extensively studied. It definitely corresponds to an utilitarian point of view, namely maximizing the revenue of the auctioneer, which is the sum of the selected bids, whoever receive them. Even if fairness does not seem to be a relevant issue in combinatorial auctions, the WDP can however inspire us a fair resource allocation problem with indivisible goods, where the agents express their preferences over bundles of items:

Definition 6 (Fair CA instance) *Given a set of agents $\mathcal{N}$ and a set of objects $\mathcal{O}$, a* bid *b is a triple $\langle s(b), p(b), a(b) \rangle \in 2^{\mathcal{O}} \times \mathbb{N} \times \mathcal{N}$ (a bundle of objects, a price and an agent). Given a set of non-intersecting bids $\mathcal{W}$ and an agent i, the utility of i regarding $\mathcal{W}$ is $u_i(\mathcal{W}) = \sum\{p(b) \mid b \in \mathcal{W}$ and $a(b) = i\}$. A* fair combinatorial auctions instance *is defined as follows:*
Input: *A set of n agents $\mathcal{N}$, a set of objects $\mathcal{O}$ and a set of bids $\mathcal{B}$.*
Output: *A set of non-intersecting bids $\mathcal{W} \subseteq \mathcal{B}$ such that there is no $\mathcal{W}' \subseteq \mathcal{B}$ with $\langle u_1(\mathcal{W}'), \ldots, u_n(\mathcal{W}') \rangle \succ_{\text{leximin}} \langle u_1(\mathcal{W}), \ldots, u_n(\mathcal{W}) \rangle$.*

The algorithms 3, 1, 4 and the first algorithm from [4] have been implemented and tested on CA instances using the constraint programming tool

kind	Algorithm 1 (**Sort**)				Algorithm 3 (**AtLeast**)			
	avg	min	max	N%	avg	min	max	N%
1	122.6	4.5	482.7	100%	**121.2**	5.1	470.1	100%
2	394.8	162.5	600	80%	**158.6**	82.8	350.6	100%
3	**480**	66	600	30%	480.8	64	600	30%
4	600	600	600	0%	**506.6**	196.2	600	30%
5	12.1	5.6	23	100%	**4.8**	2.6	7.9	100%
6	78.8	47.9	156.4	100%	**68.5**	44.1	131.6	100%

Algorithm 4 (**Leximin**)				Algorithm from [4]				Sum-optimal			
avg	min	max	N%	avg	min	max	N%	avg	min	max	N%
380	42.4	600	60%	488	32.6	600	20%	485	158	600	40%
479	161	600	50%	600	600	600	0%	485	158	600	40%
600	600	600	0%	600	600	600	0%	600	600	600	0%
600	600	600	0%	600	600	600	0%	600	600	600	0%
62.4	26.4	128	100%	600	600	600	0%	19.4	2.1	49.1	100%
94.7	26.4	203	100%	600	600	600	0%	18.8	3.7	45.7	100%

Table 1: CPU times (in sec.) and percentage of instances solved within 10 minutes (each algorithm tested on 10 instances of each kind).

Choco [10]. The test instances have been generated using CATS [12], which aims at making realistic and economically motivated bids for combinatorial auctions, *e.g* by simulating some kind of relations such as substitutabilities and complementarities between the goods. We used six different kind of instances (see [12] for the definitions of the different kinds of relationships between the goods): (1) 5 agents, 200 objects, 200 bids, arbitrary relationships, (2) 30 agents, 200 objects, 200 bids, arbitrary relationships, (3) 5 agents, 200 objects, 200 bids, regions-based relationships, (4) 30 agents, 200 objects, 200 bids, regions-based relationships, (5) 20 agents, 200 objects, 100 bids, arbitrary relationships, (6) 20 agents, 50 objects, 200 bids, arbitrary relationships.

The running times of the tests are shown in table 1. They show that the most efficient algorithm on these kinds of instances is algorithm 3, followed by algorithm 1. Conversely, algorithm 4 and the algorithm from [4] are inefficient. It is interesting to notice that, whereas the algorithms 1 and 4 are affected by the increasing of the number of agents (see *e.g* kinds 1 and 2), the running time of algorithm 3 only slightly increases (in spite of the fact that the number of calls to **maximize** is exactly the number of agents). For each instance, we also solved the WDP using our contraint programming model, which is – due to the genericity of the CP framework – far less efficient than the dedicated algorithms. It is surprising to see that solving the WDP using our CP model requires much more time than solving the MaxLeximinCSP with algorithm 3. This is rather counterintuitive since, all other parameters being equal, the running time tends to decrease with the number of agents, and solving the WDP in our constraint programming framework comes down to solve the MaxLeximinCSP on a one-agent instance.

These results must however be considered with care, since they are subject to our implementation of the algorithms. For example, not every optimizations given in [13] for the constraint **Sort** have been implemented yet. The also depend on our modeling of the combinatorial auctions problem: we used a bid-centered modeling (that is, the decision variables are the bid allocations), with binary exclusion constraint to model the incompatibilities between the bids.

Anyway, it is interesting to notice that the performances of the algorithms have been dramatically increased by using the following variable choice heuristics. Choose as the next bid to allocate the first among the non-instantiated ones, according to the lexicographic increasing order on the two following criteria: 1) the current utility of the bid's owner, 2) the price of the bid. In other words, the next bid that the algorithm will try to select is the one with the highest price among those of the currently unhappiest agent.

It is also of interest to compare the quality of the leximin-optimal solution and the sum-optimal solution in term of fairness. One visual indicator of the fairness level of a solution is its Lorenz curve [15]. Formally, given a vector $\langle u_1, \ldots, u_n \rangle$, its Lorenz curve is the following vector: $\langle u_1^{\uparrow}, u_1^{\uparrow} + u_2^{\uparrow}, \cdots \sum_{i=1}^{n} u_i^{\uparrow} \rangle$. For a perfectly equitable utility vector, the Lorenz curve is a regular staircase line from the origin $(0, 0)$ to the point $(n, \sum_i u_i)$. On the opposite, a perfectly unfair utility vector (all agents having $u_i = 0$ except one) is very far from the regular staircase line. So the unfairness of a utility vector can be appreciated by the "distance" of the Lorenz curve to the regular staircase[8]. The Lorenz curve of a vector is always convex[9], and the less convex a Lorenz curve is, the fairer the vector is. Figure 1 shows the Lorenz curves of the utility vectors of the sum- and leximin-optimal solutions in a CA instance with 20 agents.

6 Conclusion

The leximin preorder cannot be ignored when dealing with optimization problems in which some kind of fairness must be enforced between utilities of agents or equally important criteria. This paper brings a contribution to the computation of leximin-optimal solutions of combinatorial problems. It describes, within a constraint programming framework, three generic algorithms solving this problem, the second one being entirely new. These algorithms have been tested on combinatorial auctions instances. The experimental results show that our algorithm is better than the others in all of the tested cases.

References

[1] N. Bleuzen-Guernalec and A. Colmerauer. Narrowing a block of sortings in quadratic time. In *Proc. of CP'97*, pages 2–16, Linz, Austria, 1997.

[8] In other words, its relative "convexity".

[9] That is: $u_{i+1}^{\uparrow} - u_i^{\uparrow} \geq u_{j+1}^{\uparrow} - u_j^{\uparrow} \Leftrightarrow i \geq j$.

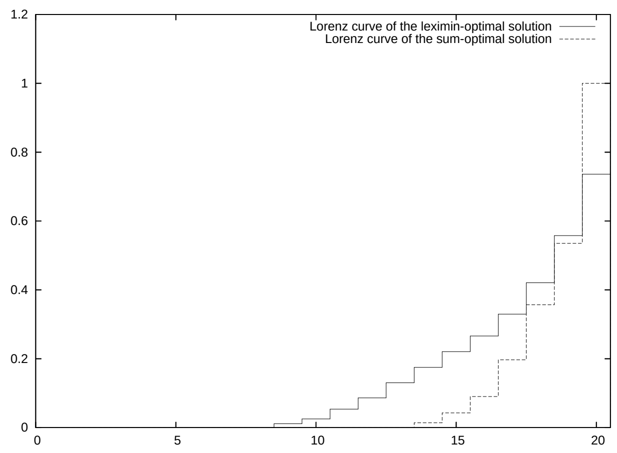

Figure 1: Lorenz curves of the utility vectors of the sum-optimal and leximin-optimal solutions in an combinatorial instance with 20 agents.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms, second edition.* MIT Press, 2001.

[3] Peter Cramton, Yoav Shoham, and Richard Steinberg. *Combinatorial Auctions.* MIT Press, 2006.

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

[5] M. Ehrgott. *Multicriteria Optimization.* Number 491 in Lecture Notes in Economics and Mathematical Systems. Springer, 2000.

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

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

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

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

[10] F. Laburthe. CHOCO: implementing a CP kernel. In *Proc. of TRICS'2000, Workshop on techniques for implementing CP systems*, Singapore, 2000. `http://sourceforge.net/projects/choco`.

[11] M. Lemaître, G. Verfaillie, and N. Bataille. Exploiting a Common Property Resource under a Fairness Constraint: a Case Study. In *Proc. of IJCAI-99*, Stockholm, 1999.

[12] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of ACM Conference on Electronic Commerce (EC-00)*, 2000.

[13] K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *Proc. of CP'00*, 2000.

[14] U. Montanari. Network of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

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

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

[17] W. Ogryczak and T. Śliwiński. On solving linear programs with the ordered weighted averaging objective. *Europen Journal of O.R.*, 148:80–91, 2003.

[18] G. Pesant and J-C. Régin. sPREAD: A balancing constraint based on statistics. In *Proc. of CP'05*, Sitges, Spain, 2005.

[19] T. Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, 134:1–54, 2002.

[20] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

[21] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *A.I.*, 58(1-3):113–159, 1992.

[22] R. Yager. On ordered weighted averaging aggregation operators in multicriteria decision making. *IEEE Trans. on Systems, Man, and Cybernetics*, 18:183–190, 1988.

Sylvain Bouveret
ONERA-DCSD. 2, avenue Édouard Belin. BP4025. 31055 Toulouse cedex 4.
IRIT, Université Paul Sabatier. 31062 Toulouse cedex.
Email: `sylvain.bouveret@cert.fr`

Michel Lemaître
ONERA-DCSD. 2, avenue Édouard Belin. BP4025. 31055 Toulouse cedex 4.
Email: `michel.lemaitre@cert.fr`