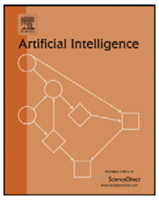
Artificial Intelligence 173 (2009) 343–364

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



Computing leximin-optimal solutions in constraint networks [☆]

Sylvain Bouveret [*], Michel Lemaître

ONERA – Centre de Toulouse, 2, avenue Édouard Belin, BP74025, 31055 Toulouse Cedex 4, France

ARTICLE INFO

Article history:
Received 31 October 2007
Received in revised form 23 September 2008
Accepted 31 October 2008
Available online 8 November 2008

Keywords:
Leximin
Fairness
Multiobjective optimization
Constraint programming

ABSTRACT

In many real-world multiobjective optimization problems one needs to find solutions or alternatives that provide a fair compromise between different conflicting objective functions—which could be criteria in a multicriteria context, or agent utilities in a multiagent context—while being efficient (i.e. informally, ensuring the greatest possible overall agents' satisfaction). This is typically the case in problems implying human agents, where fairness and efficiency requirements must be met. Preference handling, resource allocation problems are another examples of the need for balanced compromises between several conflicting objectives. A way to characterize good solutions in such problems is to use the leximin preorder to compare the vectors of objective values, and to select the solutions which maximize this preorder. In this article, we describe five algorithms for finding leximin-optimal solutions using constraint programming. Three of these algorithms are original. Other ones are adapted, in constraint programming settings, from existing works. The algorithms are compared experimentally on three benchmark problems.

1. Introduction

In many collective decision making problems implying human agents, one needs to find *efficient* and *fair* solutions. In a social choice context and informally, an efficient solution is a solution ensuring the greatest possible satisfaction to the society of agents. Efficiency[1] can have several formal definitions, as we will see later. Concerning fairness, this property refers to the need to make compromises between the agents' objectives, which are often conflicting or even antagonistic.

It is impossible to give a widely accepted and formal definition of the notion of fairness, just because it depends on the situation at stake and on the agents implied. The interested reader can refer to [33,48] for a deep investigation on fairness in many different contexts. One prominent definition of fairness, to which we will stick in this article, is the one given by egalitarianists: informally, in this context, a fair solution tries to balance the utilities of the agents, or to make the worst-off as well-off as possible. This definition will be formally defined later in the article.

Fairness is particularly relevant in areas such as 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 resource allocation problems, like, among others, bandwidth allocation among network users, fair share of airspace and airport resources among several airlines [1], or Earth observing satellite scheduling and sharing problems [24]. As we noticed earlier, the fairness

[☆] This article is an extended version of [S. Bouveret, M. Lemaître, New constraint programming approaches for the computation of leximin-optimal solutions in constraint networks, in: M.M. Veloso (Ed.), Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07), Hyderabad, India, AAAI Press, 2007].

[*] Corresponding author.

E-mail addresses: Sylvain.Bouveret@onera.fr (S. Bouveret), Michel.Lemaitre@onera.fr (M. Lemaître).

[1] Unfortunately, the words "efficiency" and "efficient" have two meanings. In Sections 1 and 2, we refer to the meaning they have in social choice and microeconomics contexts (e.g. Pareto-efficiency). From Section 3, we will refer to the computer science meaning (e.g. efficiency of an algorithm). In any case, this ambiguity can be easily cleared up considering the context in which the words appear.

0004-3702/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2008.10.010

requirement always comes with an efficiency requirement, at least implicitly: workers must be occupied, resources must be allocated, agents must be satisfied, … , "as most as possible".

In spite of the wide range of problems concerned by fairness issues, it often lacks a theoretical and generic approach. In many applications, fairness and efficiency are only enforced by specific heuristic local choices guiding the search towards supposed balanced alternatives or decisions. However, a few works may be cited for their approach of this fairness requirement. The article [24] investigates a multiagent decision problem which consists in sharing a common property resource in a fair and efficient way, three ways of handling this problem being proposed; the first one gives priority to fairness; the second one to efficiency; the third one computes a set of compromises based on a linear combination of efficiency and fairness indices. More recently and in a quite different direction, [36] proposes 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 fairness among a set of agents. Fairness is also studied in operational research, with for example [35], which proposes to model fairness requirements with an Ordered Weighted Average (OWA) aggregation [46] and investigates the way to solve this optimization problem using linear programming.

Microeconomics and social choice theory provide an important literature on fairness in collective decision making between agents. From this theoretical background we borrow the idea of representing preferences (or satisfaction levels) of agents by *utility* levels, and we adopt the *leximin* preorder on utility profiles[2] for conveying the fairness and efficiency requirements. Here is an informal definition of the leximin preorder. Consider two utility profiles $\overrightarrow{x} = (x_1, x_2, \dots, x_n)$ and $\overrightarrow{y} = (y_1, y_2, \dots, y_n)$. If $\min_i x_i < \min_i y_i$ then the leximin preorder strictly prefers $\overrightarrow{y}$ to $\overrightarrow{x}$. If $\min_i x_i = \min_i y_i$, then we eliminate one occurrence of the lowest value from both profiles, and we continue the process, comparing the second lowest values, and so on until either we find unequal minimal values, or the profiles are empty (in which case they are leximin-indifferent). Notice that the only case for which the profiles are leximin-indifferent is when they are equivalent up to a permutation of their elements. For example, the profile $(2, 2, 1, 2)$ is strictly leximin-preferred to the profile $(4, 1, 5, 1)$, while $(4, 1, 5, 1)$ and $(1, 1, 4, 5)$ are indifferent.

Before introducing it formally in Section 2, we will now give some reasons for which the leximin preorder conveys efficiency and fairness. First of all, leximin-optimal solutions are such that the worst-off agent is made as well-off as possible, thus perfectly matching the definition of fairness we introduced earlier. Secondly, leximin-optimal solutions are also Pareto-efficient, which means that we cannot increase the utility of one agent without decreasing the utility of another agent: it corresponds to a prominent notion of efficiency in collective decision making problems.

Computing a leximin-optimal solution is not a trivial problem, and cannot be easily translated into classical optimization frameworks. In this article, we will focus on this problem in the constraint programming (CP) framework, which is an effective and flexible tool for modeling and solving combinatorial problems. We will provide several generic algorithms for computing leximin-optimal solutions in this framework, the aim being to benefit from this powerful and expressive framework, and from existing state-of-the-art solvers, while adapting them to our particular problem.

Apart from the fact that it can convey and formalize the concepts of fairness and efficiency in multiagent contexts, the leximin preorder is also a subject of interest in other contexts. This preorder is of particular importance in the context of multiobjective or multicriteria decision making and optimization. In this context, it can be used to enforce a good balance between criteria or objectives, while ensuring the Pareto-efficiency of the solution. Of course, the algorithms given in this article are generic enough to be applied to multicriteria, multiobjective or multiagent problems. Moreover, we may notice that the leximin preorder is also of interest in other domains, such as fuzzy CSP [14], and symmetry-breaking in constraint satisfaction problems [15].

This contribution is organized as follows. Section 2 gives a minimal background in social choice theory and justifies formally the interest of the leximin preorder as a fairness criterion. Section 3 motivates the use of the constraint programming framework for dealing with the search for leximin-optimality in a generic way, and defines the search for leximin-optimality in this framework. The main contribution of this article is Section 4, which describes five algorithms for computing leximin-optimal alternatives, three of them being new, and the other ones adapted from existing works. We also introduce in this section a general method for designing good heuristics dedicated to the computation of leximin-optimal solutions. All the algorithms introduced have been implemented within a constraint programming system and their performance compared. Section 5 presents an experimental comparison of these algorithms on three different kinds of randomly generated instances.

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.* $(x_1, \dots, x_n)$). The notation $\{\overrightarrow{x}\}$ will be used as a shortcut for the set of elements of $\overrightarrow{x}$ (*i.e.* $\{x_1, \dots, x_n\}$), and, unless explicitly specified, $f(\overrightarrow{x})$ for $(f(x_1), \dots, f(x_n))$. Vector $\overrightarrow{x}^{\uparrow}$ (resp. $\overrightarrow{x}^{\downarrow}$) will stand for the vector composed by each element of $\overrightarrow{x}$ rearranged in increasing (resp. decreasing) order. We will write $x_i^{\uparrow}$ (resp. $x_i^{\downarrow}$) for the *i*th element of vector $\overrightarrow{x}^{\uparrow}$ (resp. $\overrightarrow{x}^{\downarrow}$). Finally, $\mathbb{R}$ denotes the set of real numbers; $\mathbb{N}$ is the set of non-negative integers; the interval of integers between k and l (included) is written $[\![k, l]\!]$.

[2] A utility profile is a vector of agent utilities, each profile corresponding to a given alternative or decision.

2.1. Collective decision making and welfarism

Let $\mathcal{N}$ be a set of n agents, and $\mathcal{A}$ be a set of admissible *alternatives* (or decisions) concerning all of them, among which a benevolent arbitrator—or the society of agents—has to choose one. The most prominent model describing this situation is *welfarism* (see for example [19,32]): 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 $i \in \mathcal{N}$ is associated with an individual utility function u_i that maps each admissible alternative $a \in \mathcal{A}$ to a numerical index $u_i(a) \in \mathbb{R}$. Therefore to each alternative a can be attached a single *utility profile* $(u_1(a), \dots, u_n(a))$. The model assumes that the individual utilities are comparable between the agents. In other words, they are expressed using a common utility scale. According to welfarism, comparing two alternatives is performed by comparing their respective utility profiles.

A standard way to compare utility profiles is to aggregate each of them into a *collective utility* index, standing for the collective welfare of the agents community. A collective utility function u_c maps each alternative a to a collective utility index $u_c(a) = g(u_1(a), \dots, u_n(a))$, where $g : \mathbb{R}^n \to \mathbb{R}$ is an aggregation function. The most used aggregation functions are the sum and the minimum, but a wide range of functions are possible, each one conveying different principles [28]. An optimal alternative, that is a most preferred one according to the collectivity, is an alternative maximizing the collective utility over the set of admissible alternatives.

2.2. The leximin preorder as a fairness and efficiency criterion

The main difficulty of fair decision problems is to reconcile the contradictory preferences of the agents. Since generally no solution fully satisfies everyone, the aggregation function g must lead to fair and efficient compromises. The fairness requirement will be discussed in depth in this section. Regarding efficiency requirements, the most widely accepted criterion is *Pareto-optimality*.

Definition 1 *(Pareto domination, Pareto-optimality [13]).* Let $\vec{x}$ and $\vec{y}$ be two vectors of $\mathbb{R}^n$. We say that $\vec{y}$ Pareto dominates $\vec{x}$ if and only if $\vec{x} \neq \vec{y}$ and $\forall i \in [\![1, n]\!]$: $x_i \leq y_i$. Let $\mathcal{V}$ be a set of vectors of $\mathbb{R}^n$. The vector $\vec{x} \in \mathcal{V}$ is Pareto-optimal in $\mathcal{V}$ if and only if no vector of $\mathcal{V}$ Pareto dominates $\vec{x}$. We extend these definitions to alternatives in the following way: let a and b two alternatives, we say that a dominates b if the utility profile of a dominates the utility profile of b. The alternative $a \in \mathcal{A}$ is Pareto-optimal in $\mathcal{A}$ if and only if no alternative of $\mathcal{A}$ dominates a.

The problem of choosing the right aggregation function g for computing a collective utility index corresponding to each profile is far beyond the scope of this article. We only describe the two standard ones corresponding to two opposite points of view on social welfare:[3] 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 aggregation in the context of fair decisions. As an example, consider two utility profiles: $\vec{u} = (10, 10, 10)$ and $\vec{v} = (1, 1, 29)$. An utilitarian decision maker will select profile $\vec{v}$, which is clearly unfair. Consider also that the sum aggregation cannot discriminate between $\vec{u} = (10, 10, 10)$ and $\vec{w} = (1, 1, 28)$, although $\vec{u}$ is clearly the fairest.

From the egalitarian point of view, the best decision is the one that maximizes the utility of the least satisfied agent (thus corresponding to $g = \min$).

Definition 2 *(Min preorder, min-optimality).* Let $\vec{x}$ and $\vec{y}$ be two vectors of $\mathbb{R}^n$. $\vec{x}$ and $\vec{y}$ are said min-indifferent (written $\vec{x} \sim_{min} \vec{y}$) if and only if $\min_{i=1}^n(x_i) = \min_{i=1}^n(y_i)$. The vector $\vec{y}$ is min-preferred to $\vec{x}$ (written $\vec{x} \prec_{min} \vec{y}$) if and only if $\min_{i=1}^n(x_i) < \min_{i=1}^n(y_i)$. We write $\vec{x} \preceq_{min} \vec{y}$ for $\vec{x} \prec_{min} \vec{y}$ or $\vec{x} \sim_{min} \vec{y}$. The binary relation $\preceq_{min}$ is a total preorder. Let $\mathcal{V}$ be a set of vectors of $\mathbb{R}^n$. A vector $\vec{y}$ of $\mathcal{V}$ is said min-optimal in $\mathcal{V}$ if and only if $\forall \vec{x} \in \mathcal{V}$: $\vec{x} \preceq_{min} \vec{y}$.

Whereas the min 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 [12]: it leaves many alternatives indistinguishable. As an example, consider the profiles $\vec{u} = (14, 20, 17)$ and $\vec{v} = (14, 15, 15)$. They cannot be distinguished by the min aggregation function, although $\vec{u}$ dominates $\vec{v}$ (so $\vec{v}$ is not Pareto-optimal). As a more striking example, the utility profiles $(0, \dots, 0)$ and $(1000, \dots, 1000, 0)$ cannot be discriminated, even if the second one appears to be much better than the first one. In other words, the min aggregation function can select non-Pareto-optimal alternatives, which is not desirable.

The leximin preorder is a known refinement of the order induced by the min aggregation function that overcomes this drawback. It has been introduced in the social choice literature (initially by [41] and discussed in depth in [10,22,32] among others) as the social welfare ordering that reconcile egalitarianism and Pareto-optimality, and also in fuzzy CSP [14]. It is defined as follows:

[3] Compromises between these two extremes are possible. See [33, page 68] or [46] (*OWA aggregation*).

Definition 3 *(Leximin preorder [41], leximin-optimality).* Let $\vec{x}$ and $\vec{y}$ be two vectors of $\mathbb{R}^n$. $\vec{x}$ and $\vec{y}$ are said leximin-indifferent (written $\vec{x} \sim_{leximin} \vec{y}$) if and only if $\vec{x}^{\uparrow} = \vec{y}^{\uparrow}$. The vector $\vec{y}$ is leximin-preferred to $\vec{x}$ (written $\vec{x} \prec_{leximin} \vec{y}$) 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 $\vec{x} \preceq_{leximin} \vec{y}$ for $\vec{x} \prec_{leximin} \vec{y}$ or $\vec{x} \sim_{leximin} \vec{y}$. The binary relation $\preceq_{leximin}$ is a total preorder. Let $\mathcal{V}$ be a set of vectors of $\mathbb{R}^n$. A vector $\vec{y}$ of $\mathcal{V}$ is said leximin-optimal in $\mathcal{V}$ if and only if $\forall \vec{x} \in \mathcal{V}$: $\vec{x} \preceq_{leximin} \vec{y}$.

In other words, the leximin preorder is the lexicographic preorder over ordered utility vectors. For example, we have $(4, 1, 5, 1) \prec_{leximin} (2, 2, 1, 2)$, because $(1, 2, 2, 2)$ is greater than $(1, 1, 4, 5)$, according to the lexicographic preorder.

From the previous definition, it is easy to see that a leximin-optimal vector is also a min-optimal one. It is known that a leximin-optimal vector is also a Pareto-optimal one [32]. This social welfare ordering has a noticeable characterization, which explains its central place and huge importance in the theory of cardinal welfarism: it is the only social welfare ordering which (1) is independent of the common utility pace and (2) which satisfies the Pigou–Dalton property at the same time (see for example [32, page 40] or [33, page 266]). The independence of the common utility pace property states that for any increasing bijection $\tau : \mathbb{R} \to \mathbb{R}$, we have $(u_1, \ldots, u_n) \preceq_{leximin} (v_1, \ldots, v_n) \Leftrightarrow (\tau(u_1), \ldots, \tau(u_n)) \preceq_{leximin} (\tau(v_1), \ldots, \tau(v_n))$; in other words, the individual utilities can be defined up to any increasing dilatation τ without modifying the leximin preorder. The Pigou-Dalton property asks that any sum-preserving transfer of utility from a more satisfied agent to a less satisfied one that narrows their two utilities leads to a collectively preferred utility profile. More formally: $\forall (\vec{u}, \vec{v})$ such that $\exists i \neq j$ such that (1) $u_i < \{v_i, v_j\} < u_j$, (2) $u_i + u_j = v_i + v_j$, and (3) $\forall k \notin \{i, j\}$, $u_k = v_k$, we have $\vec{u} \prec_{leximin} \vec{v}$. $\vec{v}$ is obviously a more equitable utility profile, since some amount of utility has been transferred from agent j (the former happier agent) to agent i. Satisfying the Pigou–Dalton property is desirable in a context where fairness is required: it reduces the inequalities between the agents when it is possible.

A known result is that no collective utility function can represent the leximin preorder,[4] unless the set of possible utility profiles is countable. This is not a major limitation, since in practice this set is often finite or it can be discretized and reduced to a finite one. In this latter case, the leximin preorder can be represented by the following non-linear functions:[5] $g_1 : \vec{x} \mapsto -\sum_{i=i}^{n} n^{-x_i}$ (adapted for leximin from one of the alternative approaches proposed in [15]), $g_2 : \vec{x} \mapsto -\sum_{i=1}^{n} x_i^{-q}$, where $q > 0$ is large enough [32], or by an Ordered Weighted Average operator [46,47] $g_3 : \vec{x} \mapsto \sum_{i=i}^{n} w_i \cdot u_i^{\uparrow}$, where $w_1 \gg w_2 \gg \cdots \gg w_n$ (where $\gg$ informally means "sufficiently bigger than"). 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 $\vec{x}$ increases. Moreover, it hides the semantics of the leximin preorder, and hinders the computational benefits we could possibly take advantage of. These points will be discussed further in Section 4.

In the following, we will use the leximin preorder as a criterion for ensuring fairness and efficiency, and we will seek the set of leximin-optimal alternatives. This problem will be expressed in the next section in a constraint programming framework.

3. Constraint programming and leximin-optimality

The constraint programming (CP) 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 [11,31,39]. Examples of actual CP frameworks and solvers are Ilog Solver, OPL Studio, Comet or Choco (see [16]).

The flexibility of the CP framework allows the user to model problems in a mathematical and incremental way, as a cooperation of separate constraints linked by shared variables. The CP user is provided with basic constraints such as "==", "≠", "≤", as well as so-called "global constraints" [45] such as "scalar product" and "allDifferent". A CP solver implements a kernel providing a basic inter-constraint propagation mechanism [3]. In this kernel, intra-constraint specific propagation algorithms are plugged.

The CP framework is a natural support of the search for leximin-optimality, for two complementary reasons. Firstly, constraints are a very convenient and flexible way to define the set of admissible alternatives, and to link decision variables to individual utilities. Secondly, the search for leximin-optimality can be considered separately in a generic way and casted as a kind of global constraint, hence providing highly re-usable algorithms in different contexts.

3.1. Constraint networks

The CP framework is based on the notion of *constraint network*. A constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ consists of a set of *variables* $\mathcal{X} = \{\mathbf{x_1}, \ldots, \mathbf{x_p}\}$, a set of associated *domains* $\mathcal{D} = \{\mathcal{D}_{\mathbf{x_1}}, \ldots, \mathcal{D}_{\mathbf{x_p}}\}$, $\mathcal{D}_{\mathbf{x_i}}$ being the set of possible *values* for $\mathbf{x_i}$, and a set of *constraints* $\mathcal{C}$, where each $C \in \mathcal{C}$ specifies a set of allowed instantiations (the concept of instantiation will be defined latter) $\mathcal{V}(C)$ over a set of variables $\mathcal{X}(C)$, called the scope of the constraint. From now on, we will suppose that all the domains are finite subsets of $\mathbb{N}$. A constraint whose scope is of size n will be called a n-ary constraint. In the following, we will use bold letters for variables (*e.g.* $\mathbf{x}$), and for a given variable $\mathbf{x}$, we will respectively write $\underline{\mathbf{x}}$ and $\bar{\mathbf{x}}$

[4] In other words there is no g such that $\vec{x} \preceq_{leximin} \vec{y} \Leftrightarrow g(\vec{x}) \leq g(\vec{y})$. See [32, page 34].

[5] For g_1 and g_2, the domain of utilities must be restricted to strictly positive real numbers.

for $\min(\mathcal{D}_{\mathbf{x}})$ and $\max(\mathcal{D}_{\mathbf{x}})$. In the algorithms, we will also use the following shortcuts for domain reductions: $\underline{\mathbf{x}} \leftarrow \alpha$ for $\mathcal{D}_{\mathbf{x}} \leftarrow \mathcal{D}_{\mathbf{x}} \cap [\![\alpha, +\infty[\![$ (all the values under α are removed from the domain of $\mathbf{x}$, notice that if $\alpha < \underline{\mathbf{x}}$, $\mathcal{D}_{\mathbf{x}}$ is not modified), $\bar{\mathbf{x}} \leftarrow \alpha$ for $\mathcal{D}_{\mathbf{x}} \leftarrow \mathcal{D}_{\mathbf{x}} \cap]\!]-\infty, \alpha]\!]$ (all the values over α are removed from the domain of $\mathbf{x}$, notice that if $\alpha > \bar{\mathbf{x}}$, $\mathcal{D}_{\mathbf{x}}$ is not modified), and $\mathbf{x} \leftarrow \alpha$ for $\mathcal{D}_{\mathbf{x}} \leftarrow \{\alpha\} \cap \mathcal{D}_{\mathbf{x}}$ (all the values different from α are removed from the domain of $\mathbf{x}$, notice that if $\alpha \notin \mathcal{D}_{\mathbf{x}}$, then $\mathcal{D}_{\mathbf{x}}$ becomes empty).

An *instantiation* v of a set $\mathcal{S}$ of variables is a function that maps each variable $\mathbf{x} \in \mathcal{S}$ to a value $v(\mathbf{x})$ of its domain $\mathcal{D}_{\mathbf{x}}$. If $\mathcal{S} = \mathcal{X}$, this instantiation is said to be *complete*, otherwise it is *partial*. If $\mathcal{S}' \subsetneq \mathcal{S}$, the *projection* of an instantiation v of $\mathcal{S}$ over $\mathcal{S}'$ is the restriction of this instantiation to $\mathcal{S}'$ and is written $v_{\downarrow \mathcal{S}'}$ (for any set of instantiations $\mathcal{V}$, we will also write $\mathcal{V}_{\downarrow \mathcal{S}'}$ the set $\{v_{\downarrow \mathcal{S}'} | v \in \mathcal{V}\}$). An instantiation v *satisfies* a constraint C if and only if it is defined on a superset of $\mathcal{X}(C)$ and $v_{\downarrow \mathcal{X}(C)} \in \mathcal{V}(C)$. 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 $sol(\mathcal{X}, \mathcal{D}, \mathcal{C})$.

3.2. Constraint Satisfaction Problems and constraint propagation

Given a constraint network, the problem of determining whether it has a solution is called a Constraint Satisfaction Problem (CSP) and is NP-complete. This problem can be solved by using backtracking search techniques [42] associated to constraint propagation [3,11], the latter aiming at detecting inconsistencies earlier in the searching process. Constraint propagation is generally based on algorithms for ensuring generalized arc consistency or bound consistency [3,11,45] that we now define, after [45]. The constraint C is said *generalized arc consistent* if and only if, for every $\mathbf{x} \in \mathcal{X}(C)$ and every value $\alpha \in \mathcal{D}_{\mathbf{x}}$ there exists an instantiation $v \in \mathcal{V}(C)$ such that $v(\mathbf{x}) = \alpha$. Now, let C be a constraint such that the domain $\mathcal{D}_{\mathbf{x}}$ of each variable $\mathbf{x} \in \mathcal{X}(C)$ is an interval domain, that is $\mathcal{D}_{\mathbf{x}} = [\![\underline{\mathbf{x}}, \bar{\mathbf{x}}]\!]$. Then, C is said *bound consistent* if and only if for every $\mathbf{x} \in \mathcal{X}(C)$ there exists an instantiation $v \in \mathcal{V}(C)$ such that $v(\mathbf{x}) = \underline{\mathbf{x}}$ and another instantiation $v' \in \mathcal{V}(C)$ such that $v'(\mathbf{x}) = \bar{\mathbf{x}}$.

3.3. The LeximinOptCSP problem

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

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 $\mathbf{o}$, and link it to the variables representing individual utilities $\mathbf{u_1}, \mathbf{u_2}, \dots, \mathbf{u_n}$ with the constraint $\mathbf{o} == g(\mathbf{u_1}, \mathbf{u_2}, \dots, \mathbf{u_n})$, where g is, as explained in Section 2, the chosen aggregation function. 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 LeximinOptCSP problem as follows:

Definition 4 *(LeximinOptCSP).*
Input: a constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; a vector of variables $\vec{\mathbf{u}}$ such that each element $\mathbf{u_i}$ is in $\mathcal{X}$, called the *objective vector*.
Output: "Inconsistent" if $sol(\mathcal{X}, \mathcal{D}, \mathcal{C}) = \emptyset$. Otherwise a solution $\widehat{v}$, called a leximin-optimal solution, such that $\forall v \in sol(\mathcal{X}, \mathcal{D}, \mathcal{C})$, $v(\vec{\mathbf{u}}) \preceq_{leximin} \widehat{v}(\vec{\mathbf{u}})$.

In the following, the set of leximin-optimal solutions of a constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ with objective vector $\vec{\mathbf{u}}$ will be written $leximinOpt(\mathcal{X}, \mathcal{D}, \mathcal{C}, \vec{\mathbf{u}})$.

We describe in the next section several generic constraint programming algorithms that solve this problem. The idea is to use the search-tree exploration algorithms and the constraint propagation mechanisms provided by the CP framework as tools for computing efficiently a leximin-optimal solution of a constraint network.

4. Constraint programming algorithms for leximin optimization

4.1. Overview of existing works and contribution of this article

Finding a leximin-optimal solution is algorithmically very easy when the input constraint network has only a few solutions: as a leximin-optimal solution is also a min-optimal one, we can compute all the min-optimal solutions (this problem can be solved as a CSP with a single objective variable) and then compare them to find a leximin-optimal one. It is the solution suggested in [13, page 162]. This rather naive approach can give good results on some class of problems and should not be completely ignored, as suggested by [34].

However, many instances have a huge number of min-optimal solutions, and thus need a cleverer approach to become tractable. The algorithmic aspects linked to the computation of a leximin-optimal solution have been treated in several works coming from several fields. Operational researchers have been interested in leximin-optimal solutions in the context of multicriteria optimization (see *e.g.* [13]), for example in the context of equitable resource allocation problems [27], location problems [34], or matrix games [37]. The leximin preorder is also an appealing criterion when dealing with fuzzy CSP, and its algorithmics has been naturally studied in this community [12]. However, the algorithms presented in these works are often either restricted to an easy case (*e.g.* continuous problems with linear or at least convex objective functions), or can become rapidly unreasonable in some cases, as we will see later in this section. An exception is the work from [34] citing [29] that briefly presents an efficient algorithm for computing a leximin-optimal solution in the discrete case. Section 4.7 revisits this latter work in a constraint programming setting.

In this section, we describe five different algorithms for solving the LeximinOptCSP, namely for computing leximin-optimal solutions in constraint networks:

- Algorithm 1 (leximin-based branch-and-bound, Section 4.3) is a new branch-and-bound algorithm, adapted to the leximin preorder. The main contribution here is the use, for lower bound filtering purposes, of a constraint introduced in [15] in a very different context.
- Algorithm 2 (branching on saturated subsets, Section 4.4) is an adaptation, in constraint programming settings, of a known algorithm due to [12].
- Algorithm 3 (based on the **Sort** constraint, Section 4.5) is a new algorithm, although simple and rather intuitive, based on the **Sort** constraint.
- Algorithm 4 (based on the **AtLeast** meta-constraint, Section 4.6) is also a new algorithm. It constitutes the main technical contribution of this article. It contains also a new and specific way to propagate the meta-constraint **AtLeast**.
- Algorithm 5 (using max-min transformations, Section 4.7), as said before, is revisiting a previous work by [29]. However, our presentation of this algorithm points out its interesting connection with *comparison networks*.

Note that all algorithms except the first one are multi-step optimization algorithms: they are based on iterative optimizations, where at each step we maximize the value of one element of the sorted version of the objective vector.

During the presentation of the algorithms, we will use the following example to illustrate how the algorithms work:

Example 1. Let $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network and let $(\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}) \in \mathcal{X}^3$ be an objective vector. We suppose that the set of solutions of the constraint network leads to the following set of admissible values for the objective vector: $(1,1,0)$, $(5,5,3)$, $(7,3,5)$, $(1,2,1)$, $(9,5,2)$, $(3,4,3)$, $(5,3,6)$ and $(10,3,4)$. Notice that this instance has 5 different min-optimal solutions, which are $(5,5,3)$, $(7,3,5)$, $(3,4,3)$, $(5,3,6)$ and $(10,3,4)$, and only one leximin-optimal solution, which is $(7,3,5)$. One can also notice that the latter leximin-optimal solution is different from the sum-optimal solution $(10,3,4)$, corresponding to the classical utilitarian point of view.

All the algorithms presented use two functions `solve` and `maximize` (the detail of which is the concern of solving techniques for constraints satisfaction problems), which respectively return one solution $v \in sol(\mathcal{X}, \mathcal{D}, \mathcal{C})$ (or "Inconsistent" if $sol(\mathcal{X}, \mathcal{D}, \mathcal{C}) = \emptyset$), and an optimal solution $\widehat{v} \in max(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathbf{o})$ (or "Inconsistent" if $sol(\mathcal{X}, \mathcal{D}, \mathcal{C}) = \emptyset$).

4.2. Difficulties with the leximin represented by a collective utility function

Before entering into the description of our algorithms, let us first explain why a straightforward approach, which consists in a standard optimization of a collective utility function representing the leximin preorder, is not effective.

As explained in Section 2, the leximin preorder can be represented by a collective utility function if the number of possible values for the objective vector is finite (which is our assumption). In that case, finding a leximin-optimal solution just comes down to a simple optimization problem with the adequate collective utility function. We here discuss the practical relevance of encoding the leximin preorder by a collective utility function in the CP optimization framework, that is, introducing an objective variable $\mathbf{u_c}$ that would stand for the collective utility and would be linked to the former objective vector $\vec{\mathbf{u}}$ by a constraint representing the collective utility function.

The first matter is the size of the domain of $\mathbf{u_c}$, due to the combinatorial nature of the set of admissible solutions. If we suppose that all the $\mathcal{D}_{\mathbf{u_i}}$ are identical,[6] and are of size m, then one can prove that the number of equivalent classes for the leximin preorder (corresponding to the minimal size of $\mathcal{D}_{\mathbf{u_c}}$) is:

$$\binom{m+n-1}{n} = \frac{(m+n-1)!}{(m-1)!\ n!}$$

This is actually the number of combinations with repetitions of n objects taken from a set of cardinality m (see for example [21, Exercise 1.2.6-60]). It is equivalent to m^n when $m \to +\infty$. This can become a problem, when m grows up, because it is difficult for some CP systems to handle very huge domains efficiently.

[6] This is a reasonable assumption, since for the leximin preorder to be meaningful, the elements of the objective vector must be expressed on a common scale.

The second issue is the way one can specify the collective utility function by a constraint between $\mathbf{u_c}$ and $\vec{\mathbf{u}}$ by using one of the aggregation functions introduced in Section 2.2. In the general case, neither the constraint $\mathbf{u_c} = -\sum_{i=1}^{n} n^{-\mathbf{u_i}}$, nor the constraint $\mathbf{u_c} = -\sum_{i=1}^{n} \mathbf{u_i}^{-q}$ are easy to propagate, which seems to dissuade to use this way for computing a leximin-optimal solution. However, for some particular cases of agents' utilities, this approach can be nevertheless efficient. For example, consider the case of a multiagent resource allocation problem where one must give one and only one object to each agent, an object j given to an agent i producing the utility $u_i = w_{ij}$. In that case, one can directly define the objective variables by $u_i = -w_{ij}^{-q}$ if object j is given to agent i, the $-w_{ij}^{-q}$ being precomputed.

Example 1a. In the example given at the end of Section 4.1, the collective utility function defined by $u_c : (u_1, u_2, u_3) \mapsto -(u_1+1)^{-9} - (u_2+1)^{-9} - (u_3+1)^{-9}$ is suitable for representing the leximin preorder. This needs a few explanations. The choice of $q = 9$ comes from the fact that it is the lowest integer (computed numerically) such that u_c represents the leximin preorder on the set of vectors of 3 elements taking their values between 0 and 10. Replacing the u_i by $(u_i + 1)$ prevents from being out of the domain of definition of the function u_c (defined for $u_i > 0$). The values for the admissible solutions are (approximately) the following: $u_c(1,1,0) = -1.00$, $u_c(5,5,3) = -4.01 \times 10^{-6}$, $u_c(7,3,5) = -3.92 \times 10^{-6}$, $u_c(1,2,1) = -3.96 \times 10^{-3}$, $u_c(9,5,2) = -5.09 \times 10^{-5}$, $u_c(3,4,3) = -8.14 \times 10^{-6}$, $u_c(5,3,6) = -3.94 \times 10^{-6}$ and $u_c(10,3,4) = -4.33 \times 10^{-6}$. One can check that the leximin-optimal vector $(7,3,5)$ has the greatest value.

Of course, even if the problem of propagating the constraint specifying the collective utility is solved, the aforementioned problem of the cardinality of the domain of the collective utility remains. In concrete terms, the constraint programming experiments we made showed us that this encoding of the leximin preorder is not very efficient, at least in the CP framework.

4.3. Algorithm 1: leximin-based branch-and-bound

A natural approach to the algorithmics of leximin is to adapt the branch-and-bound algorithm, which is the standard way of optimizing in CP, to the leximin-preorder. Informally speaking, it works as follows (see Algorithm 1): it computes a first solution, then tries to improve it by specifying that the next solution has to be strictly better (in the sense of the leximin preorder) than the current one, and so on until the constraint network becomes inconsistent. It works as if it was looking for all the solutions (like the naive approach evoked in Section 4.1), but instead of comparing them at the end of the search process, it prunes the branches that cannot lead to a better solution than the best one found so far. This approach is based on the following constraint:

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

Although this constraint does not exist in the literature, the works of [15] and [20] introduce 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 straightforward adaptations, the algorithm introduced in this work can be used to enforce the latter constraint **Leximin**. However, even if the Multiset Ordering constraint is very close to the constraint **Leximin**, it has never been used, to the best of our knowledge, in the context of computing a leximin-optimal (or leximax-optimal) solution. That is why we consider that this Algorithm 1 is new in this context.

Algorithm 1: leximin-based branch-and-bound.

```
   input  : A constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; an objective vector $(\mathbf{u_1}, \ldots, \mathbf{u_n}) \in \mathcal{X}^n$
   output: A solution to the LeximinOptCSP, or "Inconsistent"
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      $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{Leximin}(\widehat{v}(\vec{\mathbf{u}}), \vec{\mathbf{u}})\}$;
5      $v \leftarrow$ solve$(\mathcal{X}, \mathcal{D}, \mathcal{C})$;
6  if $\widehat{v} \neq$ null then return $\widehat{v}$ else return "Inconsistent";
```

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

Proposition 1. *If function* `solve` *is correct and halts, then Algorithm* 1 *halts and solves the* LeximinOptCSP.

The proof is rather straightforward, so we omit it.

4.4. Algorithm 2: branching on saturated subsets

Our next algorithm, which is our first multi-step algorithm, is based on a recursive solving of successive min-optimal sub-problems. This algorithm has been introduced, in the context of fuzzy CSP, in [12], and is also briefly suggested in [13, page 145]. The idea is to find all the possible sets of "worst" objective variables and to fix explicitly their value (defining what is called "strong α-cuts" in the context of fuzzy CSP). By "worst", we refer to *saturated subsets of objective variables*:

Definition 6 *(Saturated subset).* Let $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network and $\vec{\mathbf{u}}$ be a vector of objective variables. Let $\widehat{m}$ be the min-optimal value of $\vec{\mathbf{u}}$, that is, $\widehat{m} = \max_{v \in sol(\mathcal{X},\mathcal{D},\mathcal{C})}\{\min_i\{v(\mathbf{u_i})\}\}$. A *saturated subset of objective variables* is a subset $\mathcal{S}_{sat}$ of variables from $\vec{\mathbf{u}}$ such that $\exists v \in sol(\mathcal{X}, \mathcal{D}, \mathcal{C})$ with $\forall \mathbf{x} \in \mathcal{S}_{sat},\ v(\mathbf{x}) = \widehat{m}$ and $\forall \mathbf{y} \in \{\vec{\mathbf{u}}\} \setminus \mathcal{S}_{sat},\ v(\mathbf{y}) > \widehat{m}$.

In our Example 1, the saturated subsets are $\{\mathbf{u_2}\}$, $\{\mathbf{u_3}\}$ and $\{\mathbf{u_1}, \mathbf{u_3}\}$. Among these ones, the cardinality-minimal ones are $\{\mathbf{u_2}\}$ and $\{\mathbf{u_3}\}$.

Clearly, the only saturated subsets that can lead to a leximin-optimal solution are the cardinality-minimal ones. The idea of the algorithms introduced in [12] for computing leximin-optimal solutions in the context of fuzzy CSP is based on the computation of cardinality-minimal saturated subsets of objective variables. The algorithms informally work as follows. Firstly, the min-optimal value $\widehat{m}$ and the cardinality-minimal saturated subsets of objective variables are computed. Then for each such subset $\mathcal{S}_{sat}$, each variable from $\mathcal{S}_{sat}$ is removed from the objective vector, and its value is fixed to $\widehat{m}$, and the same is done on the new objective vector, until no variable remains.

In the general case, at each step there can be several cardinality-minimal saturated subsets of variables. The algorithm can therefore be seen as a branching procedure that chooses at each node on which saturated subset it branches. Algorithm 2 is the translation in the CP framework of the Depth-First-Search algorithm proposed in [12]. It is based on the function `explore`, which is recursively called to explore the search tree: at each node, it computes the min-optimal value (which corresponds to the function `findMinOptimal`, not developed here, but which makes one call to `maximize`), computes the cardinality-minimal saturated subsets (which corresponds to the function `findSaturatedSubsets` which makes several calls to `solve`), and branches on them. Before returning, the algorithm calls the function `leximinOptimal` which selects a leximin-optimal solution by simple leximin comparisons, because some branches of the search tree can lead to sub-optimal solutions.

Algorithm 2: branching on saturated subsets.

```
input  : A constraint network $(\mathcal{X},\mathcal{D},\mathcal{C})$; an objective vector $(\mathbf{u_1},\dots,\mathbf{u_n}) \in \mathcal{X}^n$
output: A solution to the LeximinOptCSP, or "Inconsistent"

1 $sol \leftarrow$ explore $((\mathcal{X},\mathcal{D},\mathcal{C}), (\mathbf{u_1},\dots,\mathbf{u_n}))$;
2 if $sol = \emptyset$ then return "Inconsistent";
3 return leximinOptimal($sol$);     /* Leximin comparison of the solutions found.  */
```

Function `explore`$((\mathcal{X},\mathcal{D},\mathcal{C}), (\mathbf{u_1},\dots,\mathbf{u_k}))$

```
input  : A constraint network $(\mathcal{X},\mathcal{D},\mathcal{C})$; an objective vector $(\mathbf{u_1},\dots,\mathbf{u_k}) \in \mathcal{X}^k$
output: A set of potential leximin-optimal solutions

1 if $k = 0$ then return solve$(\mathcal{X},\mathcal{D},\mathcal{C})$;
2 $\widehat{m} \leftarrow$ findMinOptimal$((\mathcal{X},\mathcal{D},\mathcal{C}), (\mathbf{u_1},\dots,\mathbf{u_k}))$;
3 $sat \leftarrow$ findMinimalSaturatedSubsets$((\mathcal{X},\mathcal{D},\mathcal{C}), (\mathbf{u_1},\dots,\mathbf{u_k}), \widehat{m})$;
4 $sol \leftarrow \emptyset$;
5 foreach $\mathcal{S} \in sat$ do
6     foreach $\mathbf{u_i} \in \mathcal{S}$ do  $\mathbf{u_i} \leftarrow \widehat{m}$;
7     foreach $\mathbf{u_i} \notin \mathcal{S}$ do  $\underline{\mathbf{u_i}} \leftarrow \widehat{m} + 1$;
8     $sol \leftarrow sol \cup$ explore$((\mathcal{X},\mathcal{D},\mathcal{C}), (\mathbf{u_1},\dots,\mathbf{u_k}) \setminus \mathcal{S})$
9 return $sol$;
```

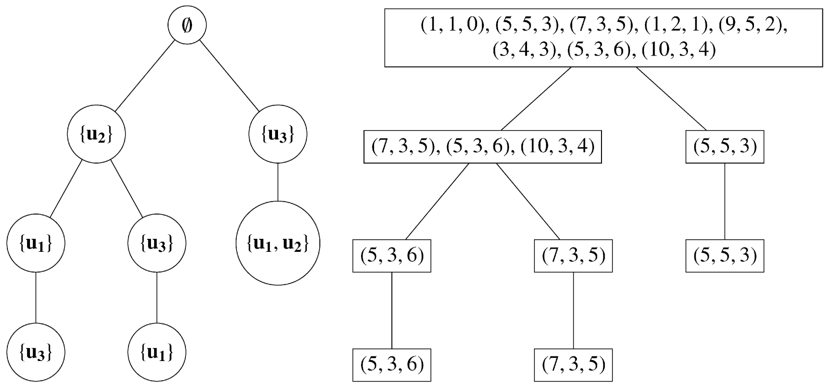


Fig. 1. The search tree developed by Algorithm 2 for Example 1.

```
Function findMinimalSaturatedSubsets($((\mathcal{X}, \mathcal{D}, \mathcal{C}), (\mathbf{u_1}, \ldots, \mathbf{u_k}), \widehat{m})$)
    input  : A constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ ; an objective vector $(\mathbf{u_1}, \ldots, \mathbf{u_k}) \in \mathcal{X}^k$ ; an integer $\widehat{m}$
    output: The set of cardinality-minimal saturated subsets of the objective vector for $\widehat{m}$
1   $sat \leftarrow \emptyset$; $i \leftarrow 1$;
2   while $i \leq k$ and $sat = \emptyset$ do
3       foreach $\mathcal{S} \subset \{\vec{\mathbf{u}}\}$ such that $|\mathcal{S}| = i$ do
4           if solve($\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \bigcup_{\mathbf{u_j} \in \mathcal{S}} \{\mathbf{u_j} = \widehat{m}\} \cup \bigcup_{\mathbf{u_j} \notin \mathcal{S}} \{\mathbf{u_j} > \widehat{m}\}$) $\neq$ "Inconsistent" then $sat \leftarrow sat \cup \mathcal{S}$ ;
5       $i \leftarrow i + 1$;
6   return $sat$;
```

Example 1b. The search tree developed by Algorithm 2 for Example 1 is shown in Fig. 1. On the left side of the figure, one can see the saturated subsets of objective variables, and on the right side the remaining solutions for each node of the search tree are shown. The set *sol* returned by the call to `explore` in Algorithm 2 is $\{(5, 3, 6), (7, 3, 5), (5, 5, 3)\}$.

The biggest problem with this algorithm is to find the saturated subsets, and, since in the general case there may be several ones, to branch on them. However, there are special known cases where there is, at each step, one saturated subset that is included in all the others, that is, only one cardinality-minimal saturated subset. In other words, `findMinimal-SaturatedSubsets` always returns only one saturated subset. In these cases, the previous algorithm does not produce any branching on the saturated subsets, and it suffices to choose at each step the cardinality-minimal saturated subset. This occurs for example in continuous linear problems, where the set of alternatives is convex (see *e.g.* [13,27,34,37]), which explains the success of this algorithm and why it performs well in this context.

*4.5. Algorithm 3: based on the **Sort** constraint*

However, in the context of discrete problems like leximin-CSP, there can be several cardinality-minimal saturated subsets at each step, and finding them might be very expensive, which renders this kind of algorithms unusable in practice. A solution to overcome this difficulty is to introduce new variables to replace the objective ones in a way such that (1) it preserves the leximin-optimal set of solutions, and (2) it guarantees the uniqueness of the cardinality-minimal saturated subset.

One way to do it is to introduce the sorted version of the objective vector. The leximin-optimal solutions relatively to the sorted objective vector are clearly the same as the leximin-optimal solutions relatively to the non-sorted objective vector. Moreover, the only saturated subsets are made of the first elements of the sorted objective vector, and thus the leximin-optimal solutions can be computed by successively maximizing the first elements of the latter sorted vector.

Introducing the entire sorted version $\vec{\mathbf{y}}$ of the objective vector $\vec{\mathbf{u}}$ can be naturally done in the CP framework by introducing a constraint **Sort**$(\vec{\mathbf{u}}, \vec{\mathbf{y}})$, which is defined as follows:

Definition 7 *(Constraint **Sort**).* Let $\vec{\mathbf{x}}$ and $\vec{\mathbf{x}}'$ be two vectors of variables of the same length, and v be an instantiation. The constraint **Sort**$(\vec{\mathbf{x}}, \vec{\mathbf{x}}')$ holds on $\{\vec{\mathbf{x}}\} \cup \{\vec{\mathbf{x}}'\}$, and is satisfied by v if and only if $v(\vec{\mathbf{x}}')$ is the sorted version of $v(\vec{\mathbf{x}})$ in increasing order.

This constraint has been particularly studied in two works, which both introduce a filtering algorithm for enforcing bound consistency on it. The first algorithm comes from [5] and runs in $O(n \log n)$ (n being the size of $\vec{\mathbf{x}}$). The authors of [30] designed a simpler algorithm that runs in $O(n)$ plus the time required to sort the interval endpoints of $\vec{\mathbf{x}}$, which can asymptotically be faster than $O(n \log n)$.

Our new algorithm (see Algorithm 3) intuitively works as follows: having introduced the sorted version $\vec{\mathbf{y}}$ of the objective vector $\vec{\mathbf{u}}$, it successively maximizes the elements of this vector, provided that the leximin-optimal solution is the solution that maximizes $\mathbf{y_1}$, and, given this maximal value, maximizes $\mathbf{y_2}$, and so on until $\mathbf{y_n}$.

Example 1c. Let us go back to our example. At the beginning of the algorithm, 3 new variables $(\mathbf{y_1}, \mathbf{y_2}, \mathbf{y_3})$ are introduced, to stand for the sorted version of the objective vector. The admissible instantiations for $(\vec{\mathbf{u}}, \vec{\mathbf{y}})$ are the following ones: $((1,1,0),(0,1,1))$, $((5,5,3),(3,5,5))$, $((7,3,5),(3,5,7))$, $((1,2,1),(1,1,2))$, $((9,5,2),(2,5,9))$, $((3,4,3),(3,3,4))$, $((5,3,6),(3,5,6))$, and $((10,3,4),(3,4,10))$.

- During the first step $\mathbf{y_1}$ is maximized (3 is its maximal value), and then it is fixed to its optimal value 3. The remaining admissible instantiations are then: $((5,5,3),(\mathbf{3},5,5))$, $((7,3,5),(\mathbf{3},5,7))$, $((3,4,3),(\mathbf{3},3,4))$, $((5,3,6),(\mathbf{3},5,6))$ and $((10,3,4),(\mathbf{3},4,10))$.
- During the second step $\mathbf{y_2}$ is maximized (5 is its maximal value), and then it is fixed to its optimal value 5. The remaining admissible instantiations are then: $((5,5,3),(3,\mathbf{5},5))$, $((7,3,5),(3,\mathbf{5},7))$ and $((5,3,6),(3,\mathbf{5},6))$.
- During the third step $\mathbf{y_3}$ is maximized (7 is its maximal value). The unique leximin-optimal solution is: $((7,3,5),(3,5,\mathbf{7}))$.

Algorithm 3: based on the **Sort** constraint.

```
input  : A constraint network $(\mathcal{X},\mathcal{D},\mathcal{C})$; an objective vector $(\mathbf{u_1},\dots,\mathbf{u_n}) \in \mathcal{X}^n$
output: A solution to the LeximinOptCSP, or "Inconsistent"
1 if solve($(\mathcal{X},\mathcal{D},\mathcal{C})$) ="Inconsistent" return "Inconsistent";
2 $\mathcal{X}' \leftarrow \mathcal{X} \cup \{\mathbf{y_1},\dots,\mathbf{y_n}\}$;
3 $\mathcal{D}' \leftarrow \mathcal{D} \cup \{\mathcal{D}_{\mathbf{y_1}},\dots,\mathcal{D}_{\mathbf{y_n}}\}$ with $\mathcal{D}_{\mathbf{y_i}} = [\![\min_j(\underline{\mathbf{u_j}}), \max_j(\overline{\mathbf{u_j}})]\!]$;
4 $\mathcal{C}' \leftarrow \mathcal{C} \cup \{\mathbf{Sort}(\vec{\mathbf{u}},\vec{\mathbf{y}})\}$;
5 for $i \leftarrow 1$ to $n$ do
6     $\widehat{v}_{(i)} \leftarrow$ maximize($\mathcal{X}',\mathcal{D}',\mathcal{C}',\mathbf{y_i}$);
7     $\mathbf{y_i} \leftarrow \widehat{v}_{(i)}(\mathbf{y_i})$;
8 return $(\widehat{v}_{(n)})_{\downarrow\mathcal{X}}$;
```

Proposition 2. *If functions* `maximize` *and* `solve` *are both correct and both halt, then Algorithm* 3 *halts and solves the* LeximinOptCSP.

Proof. If $sol(\mathcal{X},\mathcal{D},\mathcal{C}) = \emptyset$ and if `solve` is correct, then Algorithm 3 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 $\mathcal{S}_{i+1} = \mathcal{S}'_i$ for all $i \in [\![1, n-1]\!]$, which proves that if $\mathcal{S}_i \neq \emptyset$, then the call to `maximize` at line 6 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 3. Suppose that there is an instantiation $v \in sol(\mathcal{X},\mathcal{D},\mathcal{C})$ such that $\widehat{v}(\vec{\mathbf{u}}) \prec_{leximin} v(\vec{\mathbf{u}})$. We define v^+ the extension of v that instantiates each $\mathbf{y_i}$ to $v(\vec{\mathbf{u}})^{\uparrow}_i$. Then, due to constraint **Sort**, $\widehat{v}(\vec{\mathbf{y}})$ and $v^+(\vec{\mathbf{y}})$ are the respective sorted version of $\widehat{v}(\vec{\mathbf{u}})$ and $v^+(\vec{\mathbf{u}})$. Following Definition 3, there is an $i \in [\![0, n-1]\!]$ such that $\forall j \in [\![1, i]\!]$, $\widehat{v}(\mathbf{y_j}) = v^+(\mathbf{y_j})$ and $\widehat{v}(\mathbf{y_{i+1}}) < v^+(\mathbf{y_{i+1}})$. Due to line 7, we have $\widehat{v}(\mathbf{y_{i+1}}) = \widehat{v}_{(n)}(\mathbf{y_{i+1}}) = \widehat{v}_{(i+1)}(\mathbf{y_{i+1}})$. Thus v^+ is a solution in $max(\mathcal{X}',\mathcal{D}',\mathcal{C}',\mathbf{y_{i+1}})$ with objective value $v^+_{(i+1)}(\mathbf{y_{i+1}})$ strictly greater than $\widehat{v}_{(i+1)}(\mathbf{y_{i+1}})$, which contradicts the hypothesis about `maximize`. □

4.6. Algorithm 4: based on the *AtLeast* meta-constraint

Before presenting this new algorithm, we introduce the following notation: given a vector of numbers $\vec{x}$ and a number α, $\sum_i(\alpha \leq x_i)$ will be the cardinality of the set $\{i \mid \alpha \leq x_i\}$. This notation is inspired by the constraint modeling language OPL [43], where $(\alpha \leq x_i)$ is 1 if the inequality is satisfied and 0 otherwise.

The previous algorithm introduced explicitly the sorted version of the objective vector, which required the propagations to be performed on all its elements. However, it is possible to define the ith element of the sorted objective vector, without explicitly introducing the entire latter vector. The following proposition gives the trick:

Proposition 3. *Let $\vec{x}$ be a vector of numbers of size n. Then we have*:

$$x^{\uparrow}_i = \max\left\{\alpha \;\Big|\; \sum_i(\alpha \leq x_i) \geq n - i + 1\right\}.$$

In other words, the ith minimum of a vector of numbers of size n is the maximal number α such that at least $n-i+1$ elements of the vector are greater than or equal to α. This basic proposition is present, explicitly or not, in some other works involving sorting. It is explicitly used by [49] in the context of the job-shop problem, and implicitly used by [5] and [30] for propagating the constraint **Sort** (these two latter works are based on the former). Our Algorithm 4 will make a new usage of this proposition: it allows us to introduce each element of the sorted vector "lazily", one after another, contrary to the two other propagation algorithms dedicated to the constraint **Sort**.

The structure of Algorithm 4 is similar to the one of Algorithm 3. Informally it works as follows:

- it first computes the maximal value $\widehat{y_1}$ of $\mathbf{y_1}$ such that there is a solution v with $\sum_i(\widehat{y_1} \le v(\mathbf{u_i})) = n$ (or in other words $\forall i,\ \widehat{y_1} \le v(\mathbf{u_i})$),
- then it fixes $\mathbf{y_1}$ to $\widehat{y_1}$ and computes the maximal value $\widehat{y_2}$ of $\mathbf{y_2}$ such that there is a solution v with $\sum_i(\widehat{y_2} \le v(\mathbf{u_i})) \ge n-1$,
- and so on until, having fixed $\mathbf{y_{n-1}}$ to $\widehat{y_{n-1}}$, computing the maximal value $\widehat{y_n}$ of $\mathbf{y_n}$ such that there is a solution v with $\sum_i(\widehat{y_n} \le v(\mathbf{u_i})) \ge 1$.

To enforce the constraint on the $\mathbf{u_i}$, we make use of the meta-constraint **AtLeast**, derived from a *cardinality combinator* introduced by [44]:

Definition 8 *(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. It is satisfied by an instantiation v if and only if at least k constraints from Γ are satisfied by v.

This Algorithm 4 is now illustrated in our example.

Example 1d.

- During the first step a variable $\mathbf{y_1}$ is introduced, and all the objective variables must have a value which is higher than $\mathbf{y_1}$. It gives the following solutions for $(\vec{\mathbf{u}}, \mathbf{y_1})$: $((1,1,0),0)$, $((5,5,3),[\![0,\mathbf{3}]\!])$, $((7,3,5),[\![0,\mathbf{3}]\!])$, $((1,2,1),[\![0,1]\!])$, $((9,5,2),[\![0,2]\!])$, $((3,4,3),[\![0,\mathbf{3}]\!])$, $((5,3,6),[\![0,\mathbf{3}]\!])$ and $((10,3,4),[\![0,\mathbf{3}]\!])$. $\mathbf{y_1}$ is fixed to its maximal value 3 (written in bold), which restricts the set of admissible instantiations to the following ones: $((5,5,3),3)$, $((7,3,5),3)$, $((3,4,3),3)$, $((5,3,6),3)$, $((10,3,4),3)$.
- During the second step a variable $\mathbf{y_2}$ is introduced, and at least two of the objective variables must have a value which is higher than $\mathbf{y_2}$. The solutions for $(\vec{\mathbf{u}}, \mathbf{y_2})$ are thus the following ones: $((5,5,3),[\![0,\mathbf{5}]\!])$, $((7,3,5),[\![0,\mathbf{5}]\!])$, $((3,4,3),[\![0,4]\!])$, $((5,3,6),[\![0,\mathbf{5}]\!])$ and $((10,3,4),[\![0,4]\!])$. $\mathbf{y_2}$ is fixed to its maximal value 5 (written in bold), which restricts the set of admissible instantiations to $\{((5,5,3),5), ((7,3,5),5), ((5,3,6),5)\}$.
- During the third step a variable $\mathbf{y_3}$ is introduced, and at least one of the objective variables must have a value which is higher than $\mathbf{y_3}$. It gives the following solutions for $(\vec{\mathbf{u}}, \mathbf{y_3})$: $((5,5,3),[\![0,5]\!])$, $((7,3,5),[\![0,\mathbf{7}]\!])$ and $((5,3,6),[\![0,6]\!])$. The maximal value for $\mathbf{y_3}$ is 7 (written in bold), which leads to the unique leximin-optimal solution $(7,3,5)$.

Algorithm 4: based on the **AtLeast** meta-constraint.

```
input : A constraint network $(\mathcal{X},\mathcal{D},\mathcal{C})$; an objective vector $(\mathbf{u_1},\dots,\mathbf{u_n}) \in \mathcal{X}^n$
output: A solution to the LeximinOptCSP, or "Inconsistent"
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     $\mathcal{X}_i \leftarrow \mathcal{X}_{i-1} \cup \{\mathbf{y_i}\}$;
5     $\mathcal{D}_i \leftarrow \mathcal{D}'_{i-1} \cup \{\mathcal{D}_{\mathbf{y_i}}\}$ with $\mathcal{D}_{\mathbf{y_i}} = [\![\min_j(\underline{\mathbf{u_j}}), \max_j(\overline{\mathbf{u_j}})]\!]$;
6     $\mathcal{C}_i \leftarrow \mathcal{C}_{i-1} \cup \{\mathbf{AtLeast}(\{\mathbf{y_i} \le \mathbf{u_1},\dots,\mathbf{y_i} \le \mathbf{u_n}\}, n-i+1)\}$;
7     $\widehat{v_{(i)}} \leftarrow$ maximize$(\mathcal{X}_i,\mathcal{D}_i,\mathcal{C}_i,\mathbf{y_i})$;
8     $\mathcal{D}'_i \leftarrow \mathcal{D}_i$ with $\mathbf{y_i} \leftarrow \widehat{v_{(i)}}(\mathbf{y_i})$;
9 return $(\widehat{v_{(n)}})_{\downarrow\mathcal{X}}$;
```

Proposition 4. *If functions* `maximize` *and* `solve` *are both correct and both halt, then Algorithm* 4 *halts and solves the* LeximinOptCSP.

In the following proofs, we will write sol_i and sol'_i, respectively referring to $sol(\mathcal{X}_i, \mathcal{D}_i, \mathcal{C}_i)$ and $sol(\mathcal{X}_i, \mathcal{D}'_i, \mathcal{C}_i)$. We will also write $(sol_i)_{\downarrow\mathcal{X}_j}$ and $(sol'_i)_{\downarrow\mathcal{X}_j}$ for the same sets of solutions projected on $\mathcal{X}_j$ (with $j < i$). We can notice that $sol_0 = sol(\mathcal{X}, \mathcal{D}, \mathcal{C})$, and that $\forall i,\ sol'_i \subseteq sol_i$ (because of line 8 that restricts the domain of $\mathbf{y_i}$).

Lemma 1. *If $sol_0 \neq \emptyset$ then $\widehat{v}_{(n)}$ is well-defined and not equal to* "Inconsistent".

Proof. Let $i \in [\![1, n]\!]$, suppose that $sol'_{i-1} \neq \emptyset$, and let $v_{(i)} \in sol'_{i-1}$. Then extending $v_{(i)}$ by instantiating $\mathbf{y_i}$ to $\min_j(\underline{\mathbf{u_j}})$ leads to a solution of $(\mathcal{X}_i, \mathcal{D}_i, \mathcal{C}_i)$ (only one constraint has been added and it is satisfied by the latter instantiation). Therefore $sol_i \neq \emptyset$ and, if `maximize` is correct, $\widehat{v}_{(i)} \neq$ "Inconsistent" and $\widehat{v}_{(i)} \in sol'_i$. So, $sol'_i \neq \emptyset$. It proves Lemma 1 by induction. □

Lemma 2. *If $sol_0 \neq \emptyset$, then $(\widehat{v}_{(n)})_{\downarrow \mathcal{X}_i} \in sol_i, \forall i \in [\![0, n]\!]$.*

Proof. We have $sol'_i \subseteq sol_i$, and $(sol_{i+1})_{\downarrow \mathcal{X}_i} \subseteq sol'_i$ (since from $(\mathcal{X}_i, \mathcal{D}'_i, \mathcal{C}_i)$ to $(\mathcal{X}_{i+1}, \mathcal{D}_{i+1}, \mathcal{C}_{i+1})$ we just add a constraint). More generally, $(sol'_i)_{\downarrow \mathcal{X}_j} \subseteq (sol_i)_{\downarrow \mathcal{X}_j}$, and $(sol_{i+1})_{\downarrow \mathcal{X}_j} \subseteq (sol'_i)_{\downarrow \mathcal{X}_j}$, as soon as $j \leq i$. Hence, $(\widehat{v}_{(n)})_{\downarrow \mathcal{X}_i} \in (sol'_n)_{\downarrow \mathcal{X}_i} \subseteq (sol_n)_{\downarrow \mathcal{X}_i} \subseteq \cdots \subseteq (sol_{i+1})_{\downarrow \mathcal{X}_i} \subseteq sol'_i \subseteq sol_i$. □

Lemma 3. *If $sol_0 \neq \emptyset$, $\widehat{v}_{(n)}(\vec{\mathbf{y}})$ is equal to $\widehat{v}_{(n)}(\vec{\mathbf{u}})^{\uparrow}$.*

Proof. For all $i \in [\![1, n]\!]$, $(\widehat{v}_{(n)})_{\downarrow \mathcal{X}_i}$ is a solution of sol_i by Lemma 2. By Proposition 3, $(\widehat{v}_{(n)})_{\downarrow \mathcal{X}_i}[\mathbf{y_i} \leftarrow \widehat{v}_{(n)}(\vec{\mathbf{u}})^{\uparrow}_i]$ satisfies the cardinality constraint of iteration i, and is then a solution of sol_i. By definition of function `maximize`, we thus have $\widehat{v}_{(i)}(\mathbf{y_i}) \geq \widehat{v}_{(n)}(\vec{\mathbf{u}})^{\uparrow}_i$. Since $\widehat{v}_{(i)}(\mathbf{y_i}) = \widehat{v}_{(n)}(\mathbf{y_i})$, we have $\widehat{v}_{(n)}(\mathbf{y_i}) \geq \widehat{v}_{(n)}(\vec{\mathbf{u}})^{\uparrow}_i$.

Since $\widehat{v}_{(n)}$ is a solution of sol_n, at least $n - i + 1$ numbers from vector $\widehat{v}_{(n)}(\vec{\mathbf{u}})$ are greater than or equal to $\widehat{v}_{(n)}(\mathbf{y_i})$. At least the $n - i + 1$ greatest numbers from $\widehat{v}_{(n)}(\vec{\mathbf{u}})$ must then be greater than or equal to $\widehat{v}_{(n)}(\mathbf{y_i})$. These elements include $\widehat{v}_{(n)}(\vec{\mathbf{u}})^{\uparrow}_i$, which leads to $\widehat{v}_{(n)}(\mathbf{y_i}) \leq \widehat{v}_{(n)}(\vec{\mathbf{u}})^{\uparrow}_i$, proving the lemma. □

We can now put things together and prove Proposition 4.

Proof Proposition 4. If $sol(\mathcal{X}, \mathcal{D}, \mathcal{C}) = \emptyset$, and if `solve` is correct, then Algorithm 4 obviously returns "Inconsistent". Otherwise, following Lemma 1, it outputs an instantiation $(\widehat{v}_{(n)})_{\downarrow \mathcal{X}}$ which is, according to Lemma 2, a solution of $(\mathcal{X}_0, \mathcal{D}_0, \mathcal{C}_0) = (\mathcal{X}, \mathcal{D}, \mathcal{C})$.

Suppose that there is a $v \in sol(\mathcal{X}, \mathcal{D}, \mathcal{C})$ such that $\widehat{v}_{(n)}(\vec{\mathbf{u}}) \prec_{leximin} v(\vec{\mathbf{u}})$. Then following Definition 3 (up to a substitution of indices), $\exists i \in [\![1, n]\!]$ such that $\forall j < i$, $v(\vec{\mathbf{u}})^{\uparrow}_j = \widehat{v}_{(n)}(\vec{\mathbf{u}})^{\uparrow}_j$ and $\widehat{v}_{(n)}(\vec{\mathbf{u}})^{\uparrow}_i < v(\vec{\mathbf{u}})^{\uparrow}_i$. Let $v^{+}_{(i)}$ be the extension of v respectively instantiating $\mathbf{y_1}, \ldots \mathbf{y_{i-1}}$ to $\widehat{v}_{(n)}(\mathbf{y_1}), \ldots \widehat{v}_{(n)}(\mathbf{y_{i-1}})$ and $\mathbf{y_i}$ to $v(\vec{\mathbf{u}})^{\uparrow}_i$. Following Lemma 3, $\forall j$, $\widehat{v}_{(n)}(\mathbf{y_j}) = \widehat{v}_{(n)}(\vec{\mathbf{u}})^{\uparrow}_j$. By gathering all the previous equalities, we have $\forall j < i$ $v^{+}_{(i)}(\mathbf{y_j}) = \widehat{v}_{(n)}(\mathbf{y_j}) = v(\vec{\mathbf{u}})^{\uparrow}_j = (v^{+}_{(i)}(\vec{\mathbf{u}}))^{\uparrow}_j$. We also have $v^{+}_{(i)}(\mathbf{y_i}) = v(\vec{\mathbf{u}})^{\uparrow}_i = (v^{+}_{(i)}(\vec{\mathbf{u}}))^{\uparrow}_i$. By Proposition 3, $\forall j \leq i$ at least $n - j + 1$ numbers from $(v^{+}_{(i)}(\vec{\mathbf{u}}))$ are greater than or equal to $v^{+}_{(i)}(\mathbf{y_j})$, proving that $v^{+}_{(i)}$ satisfies all the cardinality constraints at iteration i. Since it also satisfies each constraint in $\mathcal{C}$ and maps each variable of $\mathcal{X}_i$ to one of its possible values, it is a solution of sol_i, and $v^{+}_{(i)}(\mathbf{y_i}) = v(\vec{\mathbf{u}})^{\uparrow}_i > \widehat{v}_{(n)}(\vec{\mathbf{u}})^{\uparrow}_i = \widehat{v}_{(i)}(\mathbf{y_i})$. It contradicts the definition of `maximize`, proving Proposition 4. □

There are some easy ways to encode the **AtLeast**(Γ, k) meta-constraint in a CP framework. A straightforward one is to "reify" each sub-constraint C_i in Γ, introducing a new variable $\mathbf{c_i}$ for each one with domain $\{0, 1\}$, then introducing new constraints holding on each $\mathbf{c_i}$, defined as $\mathbf{c_i} = 1$ if C_i is satisfied, $\mathbf{c_i} = 0$ otherwise, and finally posting $\sum_i \mathbf{c_i} \geq k$. Another possibility for our specific use would be to encode **AtLeast**$(\{\mathbf{y} \leq \mathbf{x_1}, \ldots, \mathbf{y} \leq \mathbf{x_n}\}, k)$ using the global constraint **CardPath** [2] or **Slide** [4]. **CardPath**$(\mathbf{m}, [\mathbf{x_1}, \ldots, \mathbf{x_n}], C)$, where C is a constraint of arity r, holds if and only if $C(\mathbf{x_i}, \ldots, \mathbf{x_{i+r-1}})$ holds $\mathbf{m}$ times, with $1 \leq i \leq n - r + 1$. Our specific use of **AtLeast** could be encoded by **CardPath** with $r = 1$ and C defined as $C(\mathbf{x_i}) = (\mathbf{y} \leq \mathbf{x_i})$ with the additional constraint $\mathbf{m} \geq k$.

However in our case where each constraint in the set Γ is of the form $\mathbf{y} \leq \mathbf{x_i}$, bound consistency can be enforced using the following specialized algorithm (recall that the notation $\overline{\mathbf{x}} \leftarrow \alpha$ means that all the values above α are removed from $\mathcal{D}_{\mathbf{x}}$).

Enforcing bound consistency on the **AtLeast** meta-constraint holding on simple comparison constraints.

```
input  : A vector of variables $(\mathbf{x_1}, \ldots, \mathbf{x_n})$, a variable $\mathbf{y}$, an integer $k \leq n$
output: AtLeast$(\{\mathbf{y} \leq \mathbf{x_1}, \ldots, \mathbf{y} \leq \mathbf{x_n}\}, k)$ is made bound consistent, or "Inconsistent" is returned
1 if $\sum_{i=1}^{n}(\underline{\mathbf{y}} \leq \overline{\mathbf{x_i}}) < k$ then return "Inconsistent";
2 if $\sum_{i=1}^{n}(\underline{\mathbf{y}} \leq \overline{\mathbf{x_i}}) = k$ then
3     forall $j \in [\![1, n]\!]$ such that $\underline{\mathbf{y}} \leq \overline{\mathbf{x_j}}$ do $\underline{\mathbf{x_j}} \leftarrow \underline{\mathbf{y}}$;
4 $\overline{\mathbf{y}} \leftarrow \overrightarrow{\overline{\mathbf{x}}}^{\downarrow}_{k}$;                                  /* where $\overrightarrow{\overline{\mathbf{x}}} = (\overline{\mathbf{x_1}}, \ldots, \overline{\mathbf{x_n}})$ */
5 return $\{\mathcal{D}_{\mathbf{x_1}}, \ldots, \mathcal{D}_{\mathbf{x_n}}, \mathcal{D}_{\mathbf{y}}\}$;
```

The algorithm informally works as follows. If the domains of the variables are such that the constraint cannot be satisfied anymore (line 1), the procedure returns “Inconsistent”. Otherwise, if exactly k variables among the $\mathbf{x_i}$ can still be greater than $\mathbf{y}$ then these variables must be greater than $\mathbf{y}$ (line 3). In any case the value of $\mathbf{y}$ cannot be higher than the kth highest value of the $\mathbf{x_i}$ (line 4).

This algorithm runs in $O(n)$, since the selection of the kth highest value of $\vec{\mathbf{x}}$ can be done in $O(n)$ [8, page 189]. We can notice that this algorithm is well-suited for event-based implementation of constraint propagation: in case of an update of one of the $\overline{\mathbf{x_i}}$, only lines 2–4 need to be run (because the update of $\bar{\mathbf{y}}$ will empty the domain of $\mathbf{y}$ if the condition on line 1 is not satisfied anymore); in case of an update of $\underline{\mathbf{y}}$, only lines 1–3 need to be run; any other update does not require to run the algorithm. The procedure can also benefit from storing the ordered vector $\vec{\bar{\mathbf{x}}}^{\downarrow}$ and updating it when one of the $\overline{\mathbf{x_i}}$ changes, taking $O(n)$ time. By doing so, we can access $\vec{\bar{\mathbf{x}}}^{\downarrow}_k$ in $O(1)$.

It can also be noticed that since all its arguments constraints are linear, the meta-constraint **AtLeast** can be expressed using a set of linear constraints, therefore allowing our algorithm to be implemented with a linear solver (provided that all other constraints are linear). The usual way [17, page 11] is to express our constraint **AtLeast** by introducing n 0–1 variables $\{\delta_\mathbf{1}, \ldots, \delta_\mathbf{n}\}$, and a set of linear constraints $\{\mathbf{y} \le \mathbf{x_1} + \delta_\mathbf{1}\bar{\mathbf{y}}, \ldots, \mathbf{y} \le \mathbf{x_n} + \delta_\mathbf{n}\bar{\mathbf{y}}, \sum_{i=1}^{n} \delta_\mathbf{i} \le n-k\}$.

4.7. Algorithm 5: using max-min transformations

Another way to make the sorted version of the objective appear, without using specific constraints with associated propagation mechanisms like in the two latter algorithms, is to use a set of “max-min transformations”. This solution, introduced in [29] (and cited in [34]) for dealing with leximin-optimization problems with non-convex sets of alternatives, is based on the following idea: replacing two elements $\mathbf{u_i}$, $\mathbf{u_j}$ of the objective vector by two variables $\mathbf{m}$ and $\mathbf{M}$ respectively standing for the minimum and the maximum of the two elements does not change the leximin-optimal set of solutions. In the following, we will use the notation $(\mathbf{M}, \mathbf{m}) == \mathbf{MaxMin}(\mathbf{x}, \mathbf{y})$ as a shortcut for the couple of constraints $\mathbf{M} == \mathbf{Max}(\mathbf{x}, \mathbf{y})$ and $\mathbf{m} == \mathbf{Min}(\mathbf{x}, \mathbf{y})$.

Proposition 5. *Let $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network, $\vec{\mathbf{u}}$ be an objective vector, and $\mathbf{u_i}$ and $\mathbf{u_j}$ be two distinct variables from $\vec{\mathbf{u}}$. We introduce two new variables $\mathbf{M}$ and $\mathbf{m}$, and define: $\mathcal{X}' = \mathcal{X} \cup \{\mathbf{M}, \mathbf{m}\}$, $\mathcal{D}' = \mathcal{D} \cup \{\mathcal{D}_\mathbf{M}, \mathcal{D}_\mathbf{m}\}$, with $\mathcal{D}_\mathbf{M} = \mathcal{D}_\mathbf{m} = [\![\min(\underline{\mathbf{u_i}}, \underline{\mathbf{u_j}}), \max(\overline{\mathbf{u_i}}, \overline{\mathbf{u_j}})]\!]$, and $\mathcal{C}' = \mathcal{C} \cup \{(\mathbf{M}, \mathbf{m}) == \mathbf{MaxMin}(\mathbf{u_i}, \mathbf{u_j})\}$. We also define $\vec{\mathbf{v}}$ the vector made of elements from $\vec{\mathbf{u}}$, except that $\mathbf{u_i}$ and $\mathbf{u_j}$ have been replaced by $\mathbf{M}$ and $\mathbf{m}$. We have leximinOpt$(\mathcal{X}, \mathcal{D}, \mathcal{C}, \vec{\mathbf{u}})$ = leximinOpt$(\mathcal{X}', \mathcal{D}', \mathcal{C}', \vec{\mathbf{v}})_{\downarrow \mathcal{X}}$.*

The proof of this proposition is obvious. By iteratively applying this transformation rule, we can replace the initial objective vector $\vec{\mathbf{u}} = \vec{\mathbf{u}}^{(\mathbf{0})}$ by a new one $\vec{\mathbf{u}}^{(\mathbf{1})}$ in the following way (introducing a new vector of intermediate variables $\vec{\mathbf{m}}^{(\mathbf{1})}$):

$$\begin{aligned}
\mathbf{m_n^{(1)}} &== \mathbf{u_n^{(0)}} \\
(\mathbf{u_n^{(1)}}, \mathbf{m_{n-1}^{(1)}}) &== \mathbf{MaxMin}(\mathbf{m_n^{(1)}}, \mathbf{u_{n-1}^{(0)}}) \\
(\mathbf{u_{n-1}^{(1)}}, \mathbf{m_{n-2}^{(1)}}) &== \mathbf{MaxMin}(\mathbf{m_{n-1}^{(1)}}, \mathbf{u_{n-2}^{(0)}}) \\
&\ldots \\
(\mathbf{u_2^{(1)}}, \mathbf{m_1^{(1)}}) &== \mathbf{MaxMin}(\mathbf{m_2^{(1)}}, \mathbf{u_1^{(0)}}) \\
\mathbf{u_1^{(1)}} &== \mathbf{m_1^{(1)}}
\end{aligned}$$

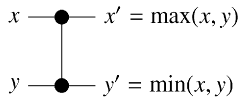
which is equivalent to:

$$\begin{aligned}
&\mathbf{u_1^{(1)}} == \mathbf{Min}\{\mathbf{u_1^{(0)}}, \mathbf{u_2^{(0)}}, \ldots, \mathbf{u_n^{(0)}}\} \\
&\mathbf{u_i^{(1)}} == \mathbf{Max}(\mathbf{u_{i-1}^{(0)}}, \mathbf{Min}\{\mathbf{u_i^{(0)}}, \mathbf{u_{i+1}^{(0)}}, \ldots, \mathbf{u_n^{(0)}}\}), \quad \forall i \in [\![2, n]\!]
\end{aligned}$$

Using this reformulation, the minimum of the objective variables appears naturally as a new variable $\mathbf{u_1^{(1)}}$, and this variable is clearly the only one cardinality-minimal saturated subset within the new vector of objective variables $\vec{\mathbf{u}}^{(\mathbf{1})}$. Like in Algorithm 2, this variable $\mathbf{u_1^{(1)}}$ will be set to its maximal value $\widehat{m}$, before processing a new vector of objective variables $\vec{\mathbf{u}}^{(\mathbf{2})}$, and so on.

Interestingly, it was not noticed by previous authors that a max-min transformation can be interpreted as a *comparator*, and the entire sorting process as a *comparison network* [8, page 704].

Definition 9 *(Comparator).* A *comparator* is a device with two inputs x and y and two outputs x' and y', that performs the following functions: $x' = \max(x, y)$ and $y' = \min(x, y)$. A comparator will be represented as follows:



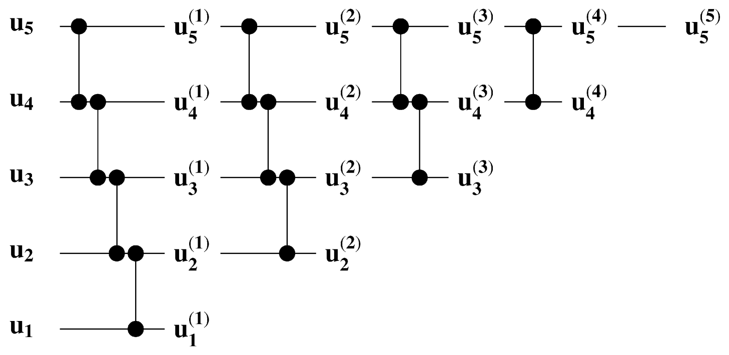


Fig. 2. The comparison network of the sorting algorithm implicitly used in Algorithm 5 for $n = 5$.

In the algorithm proposed here (Algorithm 5), there is an implicit use of a sorting algorithm implemented using a comparison network: the successive reformulations of the objective variables correspond to the use of comparators. Finally, the entire algorithm implicitly uses the sorting algorithm presented in Fig. 2.

Each comparator of Fig. 2 is "implemented" by two constraints in Algorithm 5, and each dot corresponds to a different variable. Notice that the variables and the constraints are introduced layer by layer, since at one step we only need the minimal variable (the $\mathbf{u}_\mathbf{i}^{(\mathbf{i})}$ in the figure and in the algorithm). The layers are introduced at each step by the function `minLayer`. As said before, we need to restrict the set of admissible solutions to the ones such that the minimum objective variable is maximal before introducing a new layer. This is the role of lines 8 and 9 of Algorithm 5.

Algorithm 5: using max-min transformations.

```
   input : A constraint network $(\mathcal{X},\mathcal{D},\mathcal{C})$; an objective vector $(\mathbf{u}_1,\dots,\mathbf{u}_n) \in \mathcal{X}^n$
   output: A solution to the LeximinOptCSP, or "Inconsistent"
1  if solve$(\mathcal{X},\mathcal{D},\mathcal{C})$ ="Inconsistent" then return "Inconsistent";
2  $(\mathcal{X}',\mathcal{D}',\mathcal{C}') \leftarrow (\mathcal{X},\mathcal{D},\mathcal{C})$;
3  $\vec{\mathbf{u}}^{(0)} \leftarrow \vec{\mathbf{u}}$;
4  for $i \leftarrow 1$ to $n$ do
5      $\mathcal{X}' \leftarrow \mathcal{X}' \cup \{\mathbf{m}_\mathbf{i}^{(\mathbf{i})},\dots,\mathbf{m}_\mathbf{n}^{(\mathbf{i})}\} \cup \{\mathbf{u}_\mathbf{i}^{(\mathbf{i})},\dots,\mathbf{u}_\mathbf{n}^{(\mathbf{i})}\}$;
6      $\mathcal{D}' \leftarrow \mathcal{D}' \cup \{\mathcal{D}_{\mathbf{m}_\mathbf{i}^{(\mathbf{i})}},\dots,\mathcal{D}_{\mathbf{m}_\mathbf{n}^{(\mathbf{i})}}\} \cup \{\mathcal{D}_{\mathbf{u}_\mathbf{i}^{(\mathbf{i})}},\dots,\mathcal{D}_{\mathbf{u}_\mathbf{n}^{(\mathbf{i})}}\}$ with $\mathcal{D}_{\mathbf{u}_\mathbf{j}^{(\mathbf{i})}} = \mathcal{D}_{\mathbf{m}_\mathbf{j}^{(\mathbf{i})}} = [\![\min_k(\underline{\mathbf{u_k}}),\ max_k(\overline{\mathbf{u_k}})]\!]$;
7      $\mathcal{C}' \leftarrow \mathcal{C}' \cup$ minLayer$(i)$;
8      $\widehat{v_{(i)}} \leftarrow$ maximize$(\mathcal{X}',\mathcal{D}',\mathcal{C}',\mathbf{u}_\mathbf{i}^{(\mathbf{i})})$;
9      $\mathbf{u}_\mathbf{i}^{(\mathbf{i})} \leftarrow \widehat{v_{(i)}}(\mathbf{u}_\mathbf{i}^{(\mathbf{i})})$;
10 return $(\widehat{v_{(n)}})_{\downarrow\mathcal{X}}$;
```

Function `minLayer`(i)

```
   input : The index level of a layer
   output: A set of constraints implementing a set of max-min transformations as a new layer
1  $C \leftarrow \{\mathbf{m}_\mathbf{n}^{(\mathbf{i})} == \mathbf{u}_\mathbf{n}^{(\mathbf{i-1})}\}$;
2  for $j \leftarrow n-1$ down to $i$ do
3      $C \leftarrow C \cup \{(\mathbf{u}_\mathbf{j+1}^{(\mathbf{i})},\mathbf{m}_\mathbf{j}^{(\mathbf{i})}) == \mathbf{MaxMin}(\mathbf{m}_\mathbf{j+1}^{(\mathbf{i})},\mathbf{u}_\mathbf{j}^{(\mathbf{i-1})})\}$;
4  $C \leftarrow C \cup \{\mathbf{u}_\mathbf{i}^{(\mathbf{i})} == \mathbf{m}_\mathbf{i}^{(\mathbf{i})}\}$;
5  return $C$;
```

Example 1e. Here is an illustration of Algorithm 5 on the example. The table below shows the set of solutions for the initial objective variables and the new ones. At the first step, the variables $\vec{\mathbf{u}}^{(1)}$ are introduced, and $\mathbf{u}_\mathbf{1}^{(1)}$ is maximized. Fixing this variable to its maximal value restricts the set of solutions (that explains the empty cells in the table). At the second step the variables $\vec{\mathbf{u}}^{(2)}$ are introduced, and $\mathbf{u}_2^{(2)}$ is maximized. And at the last step $\mathbf{u}_3^{(3)}$ is introduced and maximized, leaving the unique leximin-optimal solution $(7, 3, 5)$.

$\mathbf{u_1}$	$\mathbf{u_2}$	$\mathbf{u_3}$	$\mathbf{u}_1^{(1)}$	$\mathbf{u}_2^{(1)}$	$\mathbf{u}_3^{(1)}$	$\mathbf{u}_2^{(2)}$	$\mathbf{u}_3^{(2)}$	$\mathbf{u}_3^{(3)}$
1	1	0	0	1	1			
5	5	3	**3**	5	5	**5**	5	5
7	3	5	**3**	7	5	**5**	7	**7**
1	2	1	1	1	2			
9	5	2	2	9	5			
3	4	3	**3**	3	4	3	4	
5	3	6	**3**	5	6	**5**	6	6
10	3	4	**3**	10	4	4	10	

4.8. Heuristics

All the constraint programming algorithms introduced for dealing with the LeximinOptCSP can benefit from a specific heuristics than exploits the particular semantics of the leximin preorder so as to guide the search process more rapidly towards good solutions. During the search for a leximin-optimal solution, the lowest element of the objective vector is crucial: increasing it first immediately gives a better solution. It thus gives the idea of a general method to design efficient variable and value choice heuristics dedicated to the leximin optimization: at each node of the search tree, the next variable to instantiate and the next value to assign to it should be chosen such that it increases as much as possible the value of the current lowest objective variable.

In most collective decision making or resource allocation problems, the objective vector, which is the utility profile, depends on a set of (0–1) decision variables. If it is the case, the next variable to instantiate should be a decision variable that increases the most the utility of the least satisfied agent or criterion. Of course, these considerations only give a clue to build efficient heuristics for dealing with the LeximinOptCSP, and must be adapted to each kind of problem at stake.

As we will see in next section, we applied this idea to design three particular heuristics dedicated to the three kinds of instances we used to test our algorithms. These heuristics have also been compared to two classical ones; the results will be presented in Section 4.8.

5. Applications and results

We implemented all the algorithms described in this article and we tested them on three different kinds of problems: (1) a simplified and linear model of a real-world application concerning the sharing of a constellation of Earth observation satellites, (2) fair combinatorial auctions, and (3) a generic model for the allocation problem of indivisible goods, where the agents have complex preferences expressed in weighted propositional logic. The experimental settings are the following: the implementations have been developed in Java 1.6.0, using the constraint programming tool Choco [23]. The tests have been conducted on a 2.1 GHz bi-processor PC with 3.8 GB memory and running Gnu/Linux 2.6.21 for the tests concerning the comparison of heuristics, and on a 1.6 GHz `SUNW UltraSPARC-IIIi` Sun station with 1 GB memory and running Solaris 10 for all the other tests.

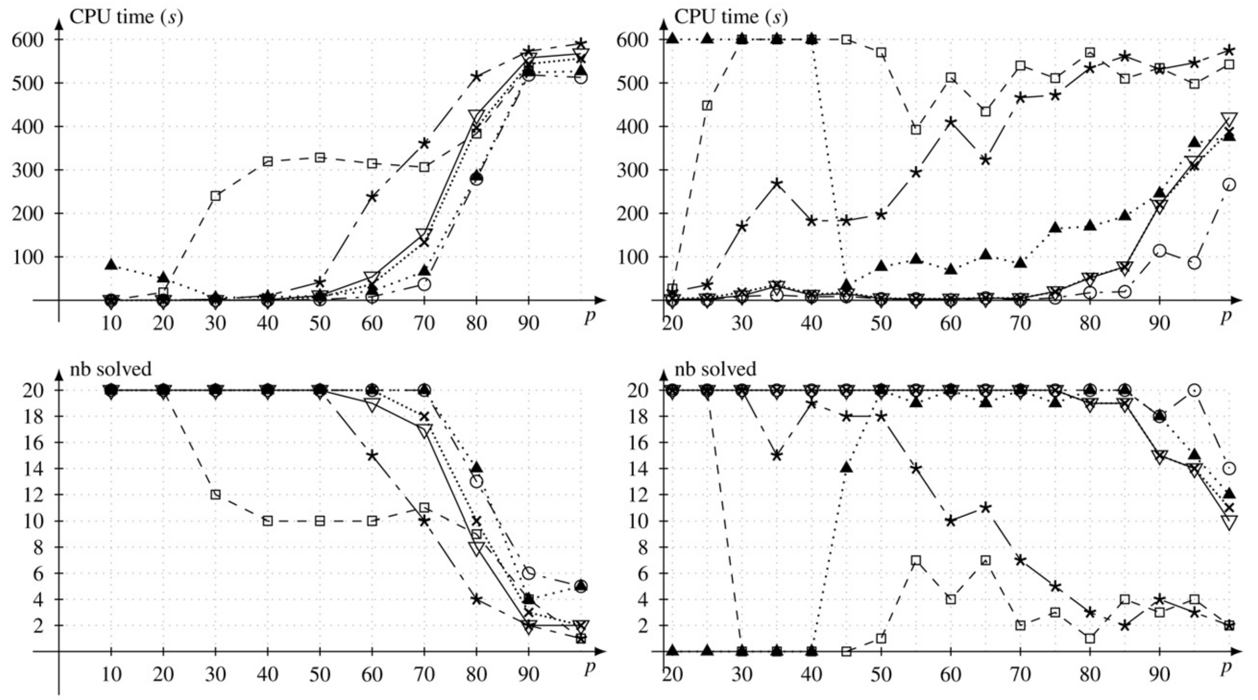
5.1. Allocation of a constellation of Earth observation satellites

Description This first application concerns the common exploitation of a constellation of agile Earth observation satellites, as described in [25]. From this application we have extracted a simplified multiagent resource allocation problem. In this problem, a set of objects $\mathcal{O}$ (standing for the resource) must be allocated to a set of agents $\mathcal{N}$. So as to approximate the real physical constraints (*e.g.* limited amount of on-board memory, limited agility of the satellite), we have introduced volume constraints over different subsets of variables, by attaching for each such constraint a volume to each object of the set, and a maximal admissible volume for this subset. There are also consumption constraints, that restrict the amount of objects that can be allocated to the same agent. The individual utility functions are specified by a set of weights $w_{i,o}$, one per pair (agent, object): given an allocation of the objects, the individual utility of an agent i is the sum of the weights $w_{i,o}$ of the objects o that she receives. The weights can be generated uniformly or such that they approximate the effect of priority levels.

The customizable generator of random instances that we have developed for testing our algorithms is available online.[8]

Experiments We conducted three sequences of experiments on this kind of problems, concerning respectively 4, 10 and 20 agents and a variable number of objects. For each number of agents and number of objects we tested the algorithms on 20

[8] http://www.cert.fr/dcsd/THESES/sbouveret/benchmark.



(a) Instances of the problem Earth observation satellites simplified, with 10 agents and p objects.

(b) Instances of the problem Earth observation satellites simplified, with 20 agents and p objects.

–□– – –□–	Algorithm based on the exhaustive comparison of all solutions, inspired by [13].
–○– · – ○–	Algorithm 1 : leximin-based branch-and-bound.
··▲·······▲··	Algorithm 2 : branching on saturated subsets.
··×··········×··	Algorithm 3 : based on the **Sort** constraint.
–▽——▽–	Algorithm 4 : based on the **AtLeast** meta-constraint.
- ★ — - ★ ·	Algorithm 5 : using max-min transformations.

Fig. 3. CPU times and number of instances solved within 10 minutes for each algorithm, run on instances of the problem Earth observation satellites simplified.

different instances, with a time limit of 10 minutes. When the number of agents is low (4 agents), the results of the tests (not shown in this article) do not bring to light any significant difference between the algorithms, probably because this kind of problems is too near to the monoagent case to show any difference between the approaches. However, this is not the case anymore when the number of agents increases. One can see for example in Fig. 3(a) that the algorithm based on the exhaustive comparison of all the leximin-optimal solutions is much less efficient than the other ones. One may notice also that Algorithm 5 based on the max-min transformations is not very efficient either. The best approaches on this kind of instances seems to be those based on the constraint **Leximin** (Algorithm 1) and on the saturated subsets (Algorithm 2). Finally, we can notice that the solving time of the latter approach increases when the number of objects tends to 0. This is not very surprising: when the number of objects decreases it becomes impossible to satisfy all the agents, thus creating a lot of equal (zero) utilities in the leximin-optimal profile, leading to make the search for saturated subsets become much harder. Things become even clearer with Fig. 3(b) (20 agents), where the algorithm based on the saturated subsets is completely inefficient on the instances with a few objects, whereas it has a reasonable solving time when the number of objects is higher. Here the best algorithms seem to be those based on the constraint **Leximin** (Algorithm 1), on the meta-constraint **AtLeast** (Algorithm 4) and on the constraint **Sort** (Algorithm 3). One can finally notice that the running times of the two last algorithms are very close on all the instances, which could be explained by the fact they both compute the sorted vector of objective variables, and that the propagation algorithm behind the constraint **Sort** uses implicitly the alternative definition of sorting used in Algorithm 4.

Heuristics We implemented for the experiments a dedicated variable choice heuristics, based on the principles given in Section 4.8: we choose the next variable to instantiate so as that it gives to the currently least satisfied agent the object she rates the most. We compared this heuristics with two classical ones: mindomain (the next variable to instantiate is

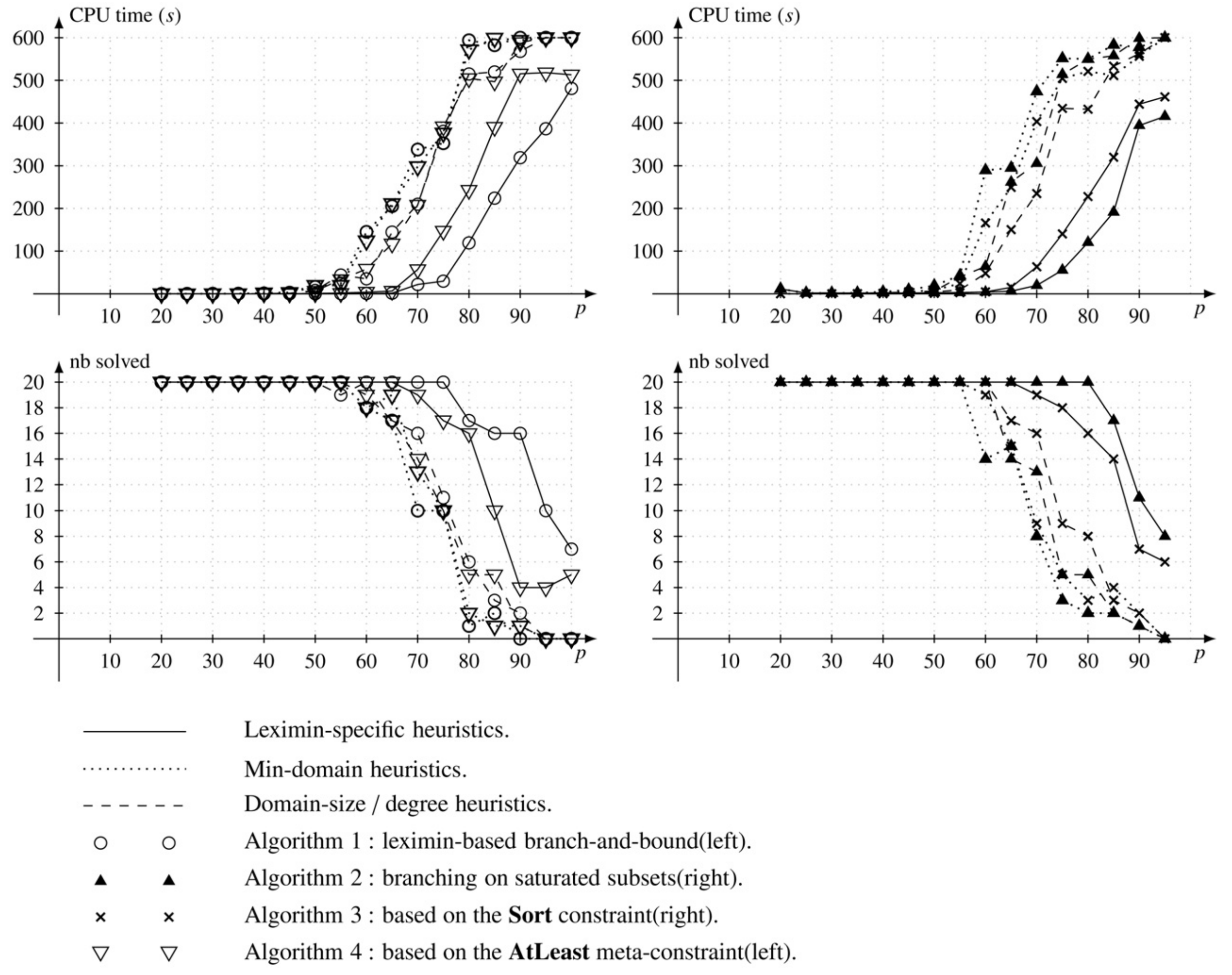


Fig. 4. CPU times and number of instances solved within 10 minutes for four algorithms, with different heuristics, on instances of the problem Earth observation satellites simplified, with 10 agents and p objects.

the one having the smallest domain) and dom/deg (the next variable to instantiate is the one having the smallest ration domain size/degree). The results are shown in Fig. 4. We can see in this figure that the leximin-specific heuristics is by far more efficient than the two other heuristics used, on each one of the four algorithms tested (1, 2, 3 and 4). For some sets of instances (see *e.g.* for 75 objects, Algorithm 1 or 2), the decrease in CPU time can reach one order of magnitude.

5.2. Fair combinatorial auctions

Description Combinatorial auctions (see *e.g.* [9,40])—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 a utilitarian point of view, namely maximizing the revenue of the auctioneer, which is the sum of the selected bids, whoever receives them. Even if fairness does not seem to be a relevant issue in combinatorial auctions, the WDP however inspired us a fair resource allocation problem with indivisible goods, where the agents express their preferences over bundles of items:

Definition 10 *(Fair CA instance).* Given a set of agents $\mathcal{N}$ and a set of objects $\mathcal{O}$, a *bid* b is a triple $(s(b), p(b), a(b)) \in 2^{\mathcal{O}} \times \mathbb{N} \times \mathcal{N}$ (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 $u_i(\mathcal{W}) = \sum\{p(b) \mid b \in \mathcal{W} \text{ and } a(b) = i\}$. 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 set of non-intersecting bids $\mathcal{W}' \subseteq \mathcal{B}$ with $(u_1(\mathcal{W}), \dots, u_n(\mathcal{W})) \prec_{leximin} (u_1(\mathcal{W}'), \dots, u_n(\mathcal{W}'))$.

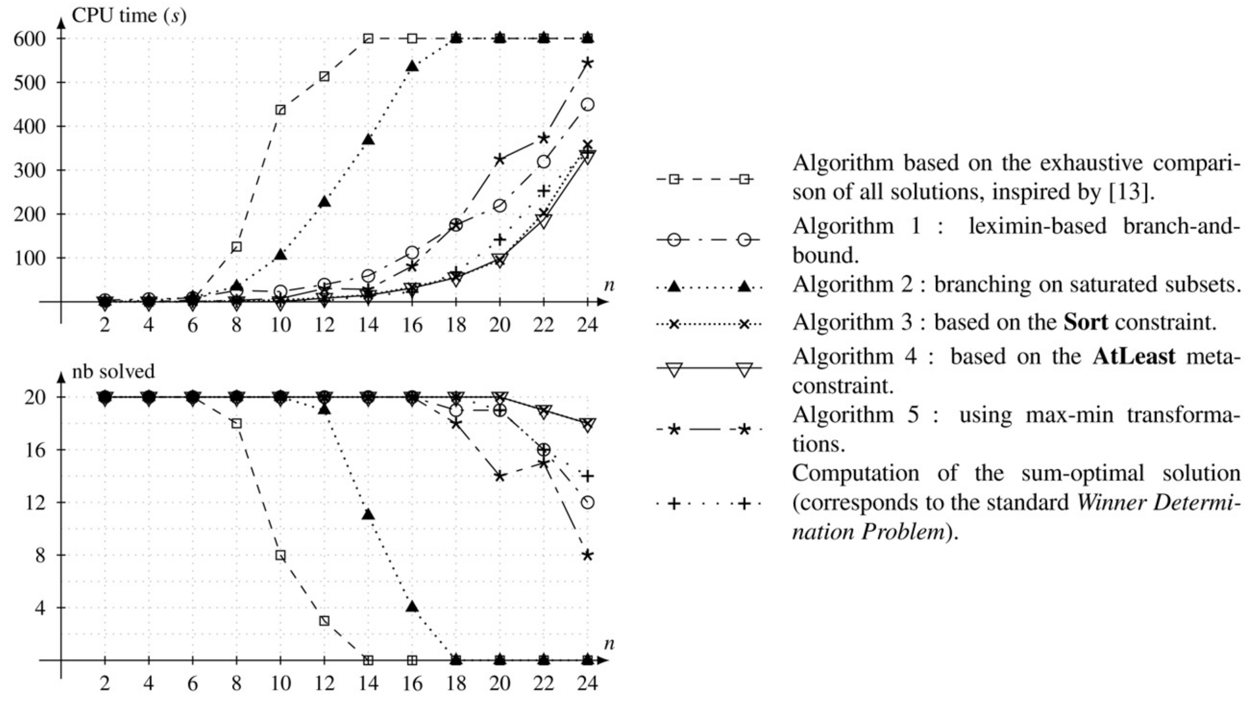


Fig. 5. CPU times and number of instances solved within 10 minutes for each algorithm, run on instances of the problem fair combinatorial auctions with *n* agents and 100 objects.

Experiments We have tested our different approaches for computing the leximin-optimal solution of a constraint network on an implementation of this problem. The test instances have been generated using CATS [26], 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. The results shown in Fig. 5 concern a variable number of agents, 100 objects and a number of bids approximately equal to 10 times the number of agents. For each number of agents we tested the algorithms on 20 different instances of kind "arbitrary", with a time limit of 10 minutes. We observe in the figure that the least efficient algorithm in this case is, like previously, the one based on the exhaustive comparison of all solutions. It is followed by the approach based on the saturated subsets, which is completely inefficient when the number of agents increases (that is, when the ratio objects/agents decreases), for the same reasons as before. The best algorithms are once again those based on the meta-constraint **AtLeast** and on the constraint **Sort**. However, one can notice that the algorithm based on the constraint **Leximin** is less efficient on this kind of instances than on the instances of the previous problem. We also compared the solving times of the algorithms with the time required to compute the sum-optimal solution (corresponding to a solution of the classical Winner Determination Problem) using constraint programming. One can see that there is no huge difference between the CPU time required to compute a sum-optimal solution and the CPU time required to compute a leximin-optimal solution.[9]

Heuristics As previously, we implemented for the experiments a dedicated variable choice heuristics, based on the principles given in Section 4.8: the next bid to allocate is the one with the higher price, among those of the currently least satisfied agent. All the results shown in this subsection use this specific heuristics.

5.3. Resource allocation problem with logical preferences

Description The last kind of problems we used to test our algorithms concerns the allocation of a set of indivisible goods to a set of agents. The agents may have complex preferences over the set of objects—that is, preferences that involve complementarity or substitutability relationships between the objects. Moreover, a set of admissibility constraints restricts the set of admissible allocations. A formal model for representing this kind of problems has been introduced in [6]. In this model, an instance of the resource allocation problem is defined as follows:

[9] At least if we use constraint programming to solve the WDP. Of course, the *ad hoc* solvers dedicated to this problem are far more efficient.

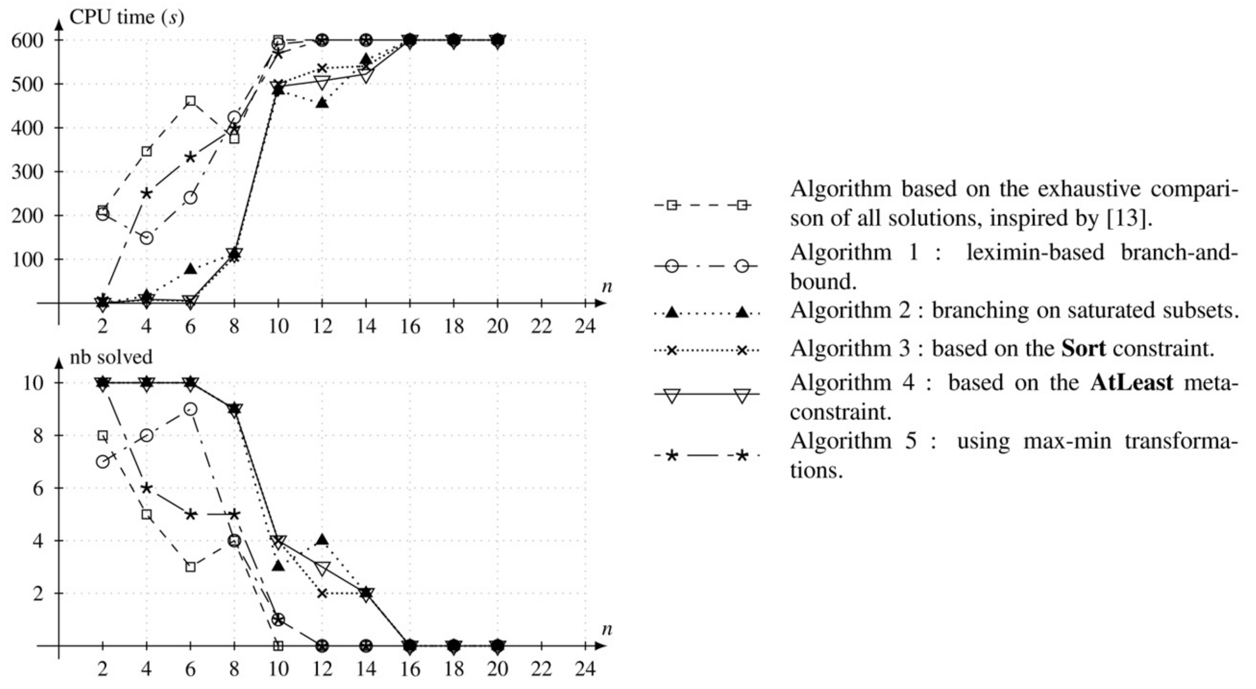


Fig. 6. CPU times and number of instances solved within 10 minutes for each algorithm, run on instances of the resource allocation problem with logical preferences with n agents and 20 objects.

Definition 11 *(Combinatorial resource allocation problem).*
Input:
- a finite set of agents $\mathcal{N} = \{1, \dots, n\}$,
- a finite set of objects $\mathcal{O}$,
- a set of admissibility constraints $\mathcal{C}$, made of propositional formulae from $L_{\mathcal{O}}^{alloc}$, the propositional language over the variables $alloc(o, i)$ ($o \in \mathcal{O}$ and $i \in \mathcal{N}$), meaning that the object o is allocated to the agent i,
- a set Δ_i of weighted formulae per agent i, made of a weight w and a propositional formula from $L_{\mathcal{O}}$, the propositional language over the variables $o \in \mathcal{O}$. Given a set of objects π_i allocated to the agent i, one can define the individual utility $u_i(\pi_i)$ of i by the sum of the weights of the formulae from Δ_i satisfied by π_i.

Output: An allocation $\vec{\pi}$, with $\pi_i \subseteq \mathcal{O}$ for all i, such that:
- all the admissibility constraints are satisfied,
- there is no other allocation $\vec{\pi}'$ also satisfying the admissibility constraints such that $(u_1(\pi_1), \dots, u_n(\pi_n)) \prec_{leximin} (u_1(\pi'_1), \dots, u_n(\pi'_n))$.

Experiments We implemented one customizable random generator, the description of which is not detailed here, dedicated to this model of resource allocation problem. The implementation of the model and the generator, as well as their complete description, are available online.[10]

We tested the algorithms on a set of instances created by this customizable generator. The results are shown in Fig. 6. The time limit is still 10 minutes, but the number of tested instances is now 10 for each number of agents and number of objects. These instances imply a great number of logical constraints, which explains why the algorithms are quite inefficient to solve them. One can see however that the best approaches are still those based on the meta-constraint **AtLeast** and on the constraint **Sort**, which is not the case for the algorithm based on the constraint **Leximin**.

Heuristics It is harder to design an efficient variable choice heuristics for this particular problem, since the attribution of an object to an agent is not "direct", due to the complex semantics of propositional logic. The heuristics we used in our experiments is the following: choose as the next object to allocate the one that appears in the formula with the highest weight, among those of the currently least satisfied agent. Although the difference between this heuristics and more classical ones is less blatant than it was for the two previous problems, it still gives better results. It would be interesting to investigate the potential gain of designing more complex heuristics for this particular application.

[10] http://www.cert.fr/dcsd/THESES/sbouveret/benchmark2007.

5.4. Conclusion and discussion

The aim of making the experiments was to find out which algorithms were the most interesting to use in the constraint programming framework. The experimental results clearly show that the most efficient algorithms in average are Algorithm 4 (based on the **AtLeast** meta-constraint), Algorithm 3 (based on the **Sort** constraint), and Algorithm 1 (leximin-based branch-and-bound), although the last one is slightly less efficient than the other ones on instances of fair combinatorial auctions and resource allocation with logical preferences, but slightly more efficient on instances of the problem Earth observation satellites simplified.

Unsurprisingly, the approach based on the exhaustive comparison of all solutions should be avoided (except on very small instances, where it is not worth using complex constraint propagation mechanisms).

The two Algorithms 4 (based on the **AtLeast** meta-constraint) and 3 (based on the **Sort** constraint) give very similar results on all kinds of instances. On the one hand it is not very surprising, because as we noticed in Section 4.6, these two algorithms are based on the same principle of sorting the objective variables, and the constraint **Sort** is indirectly based on the same trick as the one used in Algorithm 4. On the other hand, one could have expected that introducing the sorted variables one at a time in Algorithm 4 instead of introducing them at one time in Algorithm 3 would have made a difference. The experiments show that this is not the case. Therefore, one can use these two algorithms indifferently. The choice between the two can thus be driven by the requested implementation effort: Algorithm 3 (based on the **Sort** constraint) is very easy to implement if the constraint **Sort** is provided by the CP system used; Algorithm 4 (based on the **AtLeast** meta-constraint) is rather easy to implement if the CP system provides some cardinality meta-constraints, or if we choose to encode these constraints using a set of 0–1 variables and linear constraints.

Algorithm 1 (leximin-based branch-and-bound) seems to be quite efficient on instances of the problem Earth observation satellites simplified, but a little bit less in other instances, for some unknown reasons. In any case, it can be worth to try this algorithm, as it can give better results than the two latter ones. The fact that this Algorithm 1 gives better results in some cases gives the idea (as suggested by one anonymous reviewer of our previous article [7]) to mix the approach of Algorithm 4 (based on the **AtLeast** meta-constraint) with the approach of Algorithm 1, by using the constraint **Leximin** with the constraint **AtLeast** to provide more filtering. We could have expected that this approach would have been as efficient as the best of the two algorithms on each instance. However, the results of the experiments (not given here) show that mixing the two approaches is less efficient than using one of them alone. Our intuition is that the gain of using double filtering (**AtLeast** and **Leximin**) is not worth the cost of running the propagation algorithms at each node of the search tree.

Concerning the algorithm based on the saturated subsets (Algorithm 2), it seems to be reasonable, and can even be quite efficient on instances with only a few equal components in the leximin-optimal profile. However, as expected, it explodes when the number of saturated subsets increases, which is for example the case when the number of agents and the number of objects are similar in instances of the problem Earth observation satellites simplified (see *e.g.* Fig. 3(b), for small numbers of objects). Thus, this algorithm should be used only if we are sure that there is little chance to have equal components in the leximin-optimal objective vector. One can notice that it is not very surprising that this algorithm comes from the fuzzy CSP community: in this context, where the constraint satisfaction levels are continuous, it is quite unlikely that two constraints have the same satisfaction level.

Lastly, Algorithm 5 (using max-min transformations) appears to be quite inefficient. The probable reason is that the number of additional variables and constraints it introduces, only for sorting purposes, is too expensive to be efficient, even if the constraint propagation algorithms associated to the max-min constraints are rather simple.

To conclude on the relative performance of the algorithms introduced, it is not very surprising that in the CP framework, the most efficient algorithms are the ones that make the full use of constraint propagation algorithms (using global constraints). The two different approaches – **AtLeast** and **Sort** on the one hand, **Leximin** on the other hand – give slight differences in terms of performance, but it is still unclear which one it is better to use for a given particular instance.

Finally, one can see in Fig. 4 that the specific heuristics used, based on the considerations described in Section 4.8, gives quite good results on instances of the problem Earth observation satellites simplified, compared to the two other classical heuristics used (it is also efficient on the other kinds of problems, although the results are not shown here). In some cases, the gap in CPU-time implied by the use of this heuristics can be worth one order of magnitude. This is a quite interesting result: the experiments show that using this sort of heuristics (which is often easy to implement for a specific kind of problem) leads to a big improvement of the running time, compared to classical heuristics.

6. Conclusion and future work

Fairness is at the base of many real-world applications implying human agents, or seeking for a compromise between conflicting interests. We borrowed from the microeconomics field the idea that the leximin preorder is well-suited to address such fairness requirements as well as reconcile them with the crucial notion of Pareto-optimality. More generally, this preorder is adapted to all kind of multicriteria optimization problems where one has to find good compromises between a set of criteria or objective functions while ensuring Pareto-optimality.

Finding a leximin-optimal solution is not a trivial algorithmic problem. In this article we focused on the search for such solutions in a constraint network. We proposed a set of constraint programming algorithms, either adapted from other fields (such as Operational Research) or new, to address this problem. The reasons we invoked to justify the development of algorithms dedicated to this particular framework is that it provides effective, flexible and efficient tools for modeling and solving a wide range of combinatorial problems. Our approaches can be easily integrated in existing state-of-the-art CP systems and solvers, and thus heavily depend on the performance of the algorithms provided by these solvers.

We tested these algorithms on three different kinds of randomly generated problems: the first one is a linear problem inspired by a real-world application concerning the sharing of a constellation of Earth observing satellites, the second one is an adaptation of the combinatorial auctions framework to the leximin preorder, and the last one is a generic resource allocation problem concerning indivisible goods with logical constraints and preferences. Our experiments show that the best approaches are those based on the meta-constraint **AtLeast**, on the constraint **Sort**, followed by the algorithm based on the constraint **Leximin**.

This article is a contribution to a problem having a wide range of applications. It raises a lot of interesting questions and problems.

First of all, some CP approaches remain to be explored. For example, as one anonymous reviewer suggested, one could think of using the global cardinality constraint (see the work from [18,38] that describes fast bound consistency algorithms for the global cardinality constraint) to introduce and compute the *occurrence vector* corresponding to the objective vector. This approach is cited in [20], and in [15] where it is used for decomposing the Multiset Ordering constraint. Having this constraint, computing a leximin-optimal solution then comes down to compute a solution whose occurrence vector is lexicographically optimal (which can be easily done with a multi-step algorithm). It would be interesting to compare this approach with the ones we tested in this article, although the explicit introduction of the occurrence vector is probably very expensive in instances where the domains of the objective variables are large (which is the case in the experiments we performed). One may notice that this limitation also applies to the propagation algorithm introduced in the works from [15, 20] for the Multiset Ordering constraint. However the two latter works propose an adaptation of this algorithm to overcome this limitation. Our propagation algorithm for the constraint **Leximin** is actually based on this adaptation, which allows us to handle large domains.

A natural extension of our work concerns the development of incomplete algorithms dedicated to the computation of leximin-optimal solutions. One could think for example of adapting local search techniques to this particular problem. However it raises some particular and interesting difficulties, such as how to evaluate the quality of a solution, that is, the expected “distance” from a solution to a leximin-optimal one.

Another natural extension of our work is about giving up leximin-optimality (which is sometimes considered as an extreme way to aggregate individual utilities) and focusing on a “softer” modeling of fairness or compromises between the objective variables. An interesting direction is the use of Ordered Weighted Averages [46] to model fairness. It appears that most of the algorithms we introduced could be adapted to compute an OWA-optimal solution.

Acknowledgements

Authors would like to thank the anonymous reviewers for their comments and suggestions. This work has been partly supported by the project ANR-05-BLAN-0384 “Preference Handling and Aggregation in Combinatorial Domains”, funded by *Agence Nationale de la Recherche*.

References

[1] M. Ball, G.L. Donohue, K. Hoffman, Auctions for the safe, efficient and equitable allocation of airspace system resources, in: P. Cramton, Y. Shoham, R. Steinberg (Eds.), Combinatorial Auctions, MIT Press, 2006, pp. 507–538 (Chapter 22).
[2] N. Beldiceanu, M. Carlsson, Revisiting the cardinality operator and introducing cardinality-path constraint family, in: Proceedings of the 17th International Conference on Logic Programming (ICLP-01), Paphos, Cyprus, 2001.
[3] C. Bessière, Constraint propagation, in: F. Rossi, P. van Beck, T. Walsh (Eds.), Handbook of Constraint Programming, Foundations of Artificial Intelligence, Elsevier, 2006, pp. 29–83 (Chapter 3).
[4] C. Bessière, E. Hebrard, B. Hnich, Z. Kiziltan, T. Walsh, SLIDE: A useful special case of the CARDPATH Constraint, in: Proceedings of the 18th European Conference on Artificial Intelligence (ECAI-08), Patras, Greece, 2008.
[5] N. Bleuzen-Guernalec, A. Colmerauer, Narrowing a block of sortings in quadratic time, in: G. Smolka (Ed.), Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP-97), Schloss Hagenberg, Austria, Springer, 1997.
[6] S. Bouveret, H. Fargier, J. Lang, M. Lemaître, Allocation of indivisible goods: a general model and some complexity results, in: F. Dignum, V. Dignum, S. Koenig, S. Kraus, M.P. Singh, M. Wooldridge (Eds.), Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-05), Utrecht, The Netherlands, ACM, 2005.
[7] S. Bouveret, M. Lemaître, New constraint programming approaches for the computation of leximin-optimal solutions in constraint networks, in: M.M. Veloso (Ed.), Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07), Hyderabad, India, AAAI Press, 2007.
[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., MIT Press, 2001.
[9] P. Cramton, Y. Shoham, R. Steinberg (Eds.), Combinatorial Auctions, MIT Press, 2006.
[10] C. d’Aspremont, L. Gevers, Equity and the informational basis of collective choice, Review of Economic Studies 44 (2) (1977) 199–209.
[11] R. Dechter, Constraint Processing, Morgan Kaufmann, 2003.
[12] D. Dubois, P. Fortemps, Computing improved optimal solutions to max-min flexible constraint satisfaction problems, European Journal of Operational Research 118 (1999) 95–126.
[13] M. Ehrgott, Multicriteria Optimization, Lecture Notes in Economics and Mathematical Systems, vol. 491, Springer, 2000.
[14] H. Fargier, J. Lang, T. Schiex, Selecting preferred solutions in fuzzy constraint satisfaction problems, in: Proceedings of the First European Congress on Fuzzy Intelligent Technologies (EUFIT’93), Aachen, 1993.
[15] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh, Filtering algorithms for the multiset ordering constraint, Artificial Intelligence 173 (2) (2009) 299–328, this issue.

[16] T. Frühwirth, L. Michel, C. Schulte, Constraints in procedural and concurrent languages, in: F. Rossi, P. van Beck, T. Walsh (Eds.), Handbook of Constraint Programming, Foundations of Artificial Intelligence, Elsevier, 2006, pp. 453–494 (Chapter 13).

[17] R.S. Garfinkel, G.L. Nemhauser, Integer Programming, Wiley, 1972.

[18] I. Katriel, S. Thiel, Fast bound consistency for the global cardinality constraint, in: F. Rossi (Ed.), Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03), Kinsale, County Cork, Ireland, Springer, 2003.

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

[20] Z. Kiziltan, Symmetry breaking ordering constraints, Ph.D. thesis, Uppsala University, 2004.

[21] D.E. Knuth, The Art of Computer Programming, vol. 1, Fundamental Algorithms, Addison-Wesley, 1968.

[22] S.-C. Kolm, Justice et Équité, Cepremap, CNRS Paris, 1972, English translation: Justice and Equity, MIT Press, 1998.

[23] F. Laburthe, CHOCO: Implementing a CP kernel, in: Proceedings of TRICS'2000, Workshop on Techniques for Implementing CP Systems, Singapore, 2000, http://sourceforge.net/projects/choco.

[24] M. Lemaître, G. Verfaillie, N. Bataille, Exploiting a common property resource under a fairness constraint: a case study, in: T. Dean (Ed.), Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden, Morgan Kaufmann, 1999.

[25] M. Lemaître, G. Verfaillie, F. Jouhaud, J.-M. Lachiver, N. Bataille, Selecting and scheduling observations of agile satellites, Aerospace Science and Technology 6 (2002) 367–381.

[26] K. Leyton-Brown, M. Pearson, Y. Shoham, Towards a universal test suite for combinatorial auction algorithms, in: Proceedings of the 2nd ACM Conference on Electronic Commerce (EC-00), Minneapolis, MN, ACM, 2000.

[27] H. Luss, On equitable resource allocation problems: a lexicographic minimax approach, Operations Research 47 (3) (1999) 361–378.

[28] J.-L. Marichal, Aggregation operators for multicriteria decision aid, Ph.D. thesis, Faculté des Sciences de l'Université de Liège, 1999.

[29] M. Maschler, J.A.M. Potters, S.H. Tijs, The general nucleolus and the reduced game property, International Journal of Game Theory 21 (1992) 85–106.

[30] K. Mehlhorn, S. Thiel, Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint, in: R. Dechter (Ed.), Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP-00), Singapore, Springer, 2000.

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

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

[33] H. Moulin, Fair Division and Collective Welfare, MIT Press, 2003.

[34] W. Ogryczak, On the lexicographic minimax approach to location problems, European Journal of Operational Research 100 (1997) 566–585.

[35] W. Ogryczak, T. Śliwiński, On solving linear programs with the ordered weighted averaging objective, European Journal of Operational Research 148 (2003) 80–91.

[36] G. Pesant, J.-C. Régin, sPREAD: A balancing constraint based on statistics, in: P. van Beek (Ed.), Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP-04), Sitges, Spain, Springer, 2005.

[37] J.A.M. Potters, S.H. Tijs, The nucleolus of a matrix game and other nucleoli, Mathematics of Operations Research 17 (1992) 164–174.

[38] C.-G. Quimper, A. Golynski, A. López-Ortiz, P. van Beek, An efficient bounds consistency algorithm for the global cardinality constraint, Constraints 10 (2) (2005) 115–135.

[39] F. Rossi, P. van Beck, T. Walsh (Eds.), Handbook of Constraint Programming, Foundations of Artificial Intelligence, Elsevier, 2006.

[40] T.W. Sandholm, Algorithm for optimal winner determination in combinatorial auctions, Artificial Intelligence 134 (2002) 1–54.

[41] A.K. Sen, Collective Choice and Social Welfare, North-Holland, 1970.

[42] P. van Beek, Backtracking search algorithms, in: F. Rossi, P. van Beck, T. Walsh (Eds.), Handbook of Constraint Programming, Foundations of Artificial Intelligence, Elsevier, 2006, pp. 85–134 (Chapter 4).

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

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

[45] W.-J. van Hoeve, I. Katriel, Global Constraints, in: F. Rossi, P. van Beck, T. Walsh (Eds.), Handbook of Constraint Programming, Foundations of Artificial Intelligence, Elsevier, 2006, pp. 169–208 (Chapter 6).

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

[47] R.R. Yager, On the analytic representation of the leximin ordering and its application to flexible constraint propagation, European Journal of Operational Research 102 (1) (1997) 176–192.

[48] H.P. Young, Equity in Theory and Practice, Princeton University Press, 1994.

[49] J. Zhou, A permutation-based approach for solving the job-shop problem, Constraints 2 (2) (1997) 185–213.