Algorithmes de programmation par contraintes pour la recherche d'allocations leximin-optimales

Sylvain Bouveret

Office National d'Études et de Recherches Aérospatiales
Centre National d'Études Spatiales
Institut de Recherche en Informatique de Toulouse

Séminaire de l'unité BIA, INRA Toulouse
Le 20 octobre 2006

Fairness in combinatorial problems. . .

Many real-world combinatorial problems. . .

- Nurse rostering problem.
- Balanced timetables.
- Fair allocation of airport and airspace resources (to several airlines).
- Fair share of Earth Observation Satellites.

. . . imply human **agents** and thus involve directly or indirectly the concept of **fairness**.

All of these problems are combinatorial collective decision making problems, under admissibility constraints, and fairness requirements.

How can we model and solve this kind of problems in a Constraint Programming framework ?

Fairness in combinatorial problems...

Many real-world combinatorial problems...

- Nurse rostering problem.
- Balanced timetables.
- Fair allocation of airport and airspace resources (to several airlines).
- Fair share of Earth Observation Satellites.

...imply human **agents** and thus involve directly or indirectly the concept of **fairness**.

All of these problems are combinatorial collective decision making problems, under admissibility constraints, and fairness requirements.

How can we model and solve this kind of problems in a Constraint Programming framework ?

Fairness in combinatorial problems. . .

Many real-world combinatorial problems. . .

- Nurse rostering problem.
- Balanced timetables.
- Fair allocation of airport and airspace resources (to several airlines).
- Fair share of Earth Observation Satellites.

. . . imply human **agents** and thus involve directly or indirectly the concept of **fairness**.

All of these problems are combinatorial collective decision making problems, under admissibility constraints, and fairness requirements.

How can we model and solve this kind of problems in a Constraint Programming framework ?

Outline

1. Formalizing and modeling the problem
   - CSP and constraint programming
   - Fairness in collective decision making
   - Leximin and Constraint Programming
2. Two constraint programming algorithms
   - Using a set of cardinality constraints
   - Using a multiset ordering constraint
   - Sort and conquer
   - Using the saturated subsets
3. Application, tests and results
   - Benchmark and results

Outline

1. Formalizing and modeling the problem
   - CSP and constraint programming
   - Fairness in collective decision making
   - Leximin and Constraint Programming
2. Two constraint programming algorithms
   - Using a set of cardinality constraints
   - Using a multiset ordering constraint
   - Sort and conquer
   - Using the saturated subsets
3. Application, tests and results
   - Benchmark and results

Constraint networks

Constraint network [Montanari, 1974]

A constraint network is based on :

- a set of variables $\mathcal{X} = \{x_1, \ldots, x_p\}$;
- a set of domains $\mathcal{D} = \{d_{x_1}, \ldots, d_{x_p}\}$;
- a set of constraints $\mathcal{C}$, with, for all $C \in \mathcal{C}$:
  - $X(C)$ the scope of the constraint,
  - $R(C)$ the set of allowed tuples of the constraint.

Montanari, U. (1974).
Networks of constraints: Fundamental properties and applications to picture processing.
Information Sciences, 7:95–132.

The Constraint Satisfaction Problem

Classical CSP

Given : A constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$.
Is there a complete consistent instantiation v *of* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *?*

$\rightsquigarrow$ **NP**-complete.

CSP with objective variable

Given : A constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ and an objective variable $o \in \mathcal{X}$, such that $d_o \subset \mathbb{N}$.
What is the maximal value α *of* d_o *such that there is a complete consistent instantiation* $\widehat{v}$ *with* $\widehat{v}(o) = \alpha$ *?*

$\rightsquigarrow$ **NP**-complete (for the associated decision problem).

The Constraint Satisfaction Problem

Classical CSP

Given : A constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$.
Is there a complete consistent instantiation v of $(\mathcal{X}, \mathcal{D}, \mathcal{C})$?

$\rightsquigarrow$ **NP**-complete.

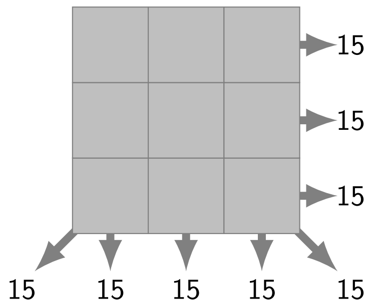
CSP with objective variable

Given : A constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ and an objective variable $o \in \mathcal{X}$, such that $d_o \subset \mathbb{N}$.
What is the maximal value α of d_o such that there is a complete consistent instantiation $\widehat{v}$ with $\widehat{v}(o) = \alpha$?

$\rightsquigarrow$ **NP**-complete (for the associated decision problem).

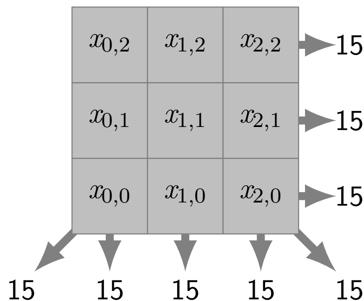
Example of a CSP

The magic square, aka. *Lo Shu* square



Example of a CSP

The magic square, aka. *Lo Shu* square



Variables:

x_i, j, with $(i, j) \in \{0, 1, 2\}^2$.

Domains:

$\forall i, j, \mathcal{D}_{x_{i,j}} = \{1, \dots, 9\}$.

Constraints:

$\textbf{alldiff}(x_{i,j} | (i, j) \in \{0, 1, 2\}^2)$,
$\forall i, \sum_{j=0}^{2} x_{i,j} = 15$,
$\forall j, \sum_{i=0}^{2} x_{i,j} = 15$,
$x_{0,0} + x_{1,1} + x_{2,2} = 15$,
$x_{2,0} + x_{1,1} + x_{0,2} = 15$.

Solving CSP with Constraint Programming

Constraint Programming is a solving framework for dealing with Constraint Satisfaction Problems. It is based on:

- a search algorithm,
- some contraint propagation mechanisms reacting on some events on the variables (bounds updated, instantiation, . . .).

Coming back on the example

$x_{0,2}$	$x_{1,2}$	$x_{2,2}$
$x_{0,1}$	$x_{1,1}$	$x_{2,1}$
$x_{0,0}$	$x_{1,0}$	$x_{2,0}$

$x_{0,0}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{0,1}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{0,2}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{1,0}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{1,1}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{1,2}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{2,0}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{2,1}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{2,2}$: $\{1,2,3,4,5,6,7,8,9\}$

Events queue:

Coming back on the example

$x_{0,2}$	$x_{1,2}$	$x_{2,2}$
$x_{0,1}$	$x_{1,1}$	$x_{2,1}$
$x_{0,0}$	$x_{1,0}$	$x_{2,0}$

$x_{0,0}$: $\{1\}$
$x_{0,1}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{0,2}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{1,0}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{1,1}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{1,2}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{2,0}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{2,1}$: $\{1,2,3,4,5,6,7,8,9\}$
$x_{2,2}$: $\{1,2,3,4,5,6,7,8,9\}$

Events queue:

awakeOnInst($\mathbf{alldiff}(x_{i,j} | (i,j) \in \{0,1,2\}^2)$, $x_{0,0}$),
awakeOnInst($\sum_{i=0}^{2} x_{i,0} = 15$, $x_{0,0}$),
awakeOnInst($\sum_{j=0}^{2} x_{0,j} = 15$, $x_{0,0}$),
awakeOnInst($x_{0,0} + x_{1,1} + x_{2,2} = 15$, $x_{0,0}$).

Coming back on the example

$x_{0,2}$	$x_{1,2}$	$x_{2,2}$
$x_{0,1}$	$x_{1,1}$	$x_{2,1}$
$x_{0,0}$	$x_{1,0}$	$x_{2,0}$

1

$x_{0,0}$: $\{1\}$
$x_{0,1}$: $\{2,3,4,5,6,7,8,9\}$
$x_{0,2}$: $\{2,3,4,5,6,7,8,9\}$
$x_{1,0}$: $\{2,3,4,5,6,7,8,9\}$
$x_{1,1}$: $\{2,3,4,5,6,7,8,9\}$
$x_{1,2}$: $\{2,3,4,5,6,7,8,9\}$
$x_{2,0}$: $\{2,3,4,5,6,7,8,9\}$
$x_{2,1}$: $\{2,3,4,5,6,7,8,9\}$
$x_{2,2}$: $\{2,3,4,5,6,7,8,9\}$

Events queue:

awakeOnInst($\sum_{i=0}^{2} x_{i,0} = 15,\ x_{0,0}$),
awakeOnInst($\sum_{j=0}^{2} x_{0,j} = 15,\ x_{0,0}$),
awakeOnInst($x_{0,0} + x_{1,1} + x_{2,2} = 15,\ x_{0,0}$),
awakeOnInf($\sum_{i=0}^{2} x_{i,0} = 15,\ x_{0,1}$),

...

Coming back on the example

$x_{0,2}$	$x_{1,2}$	$x_{2,2}$
$x_{0,1}$	$x_{1,1}$	$x_{2,1}$
$x_{0,0}$	$x_{1,0}$	$x_{2,0}$

1

$x_{0,0}$: $\{1\}$
$x_{0,1}$: $\{2,3,4,5,6,7,8,9\}$
$x_{0,2}$: $\{2,3,4,5,6,7,8,9\}$
$x_{1,0}$: $\{5,6,7,8,9\}$
$x_{1,1}$: $\{2,3,4,5,6,7,8,9\}$
$x_{1,2}$: $\{2,3,4,5,6,7,8,9\}$
$x_{2,0}$: $\{5,6,7,8,9\}$
$x_{2,1}$: $\{2,3,4,5,6,7,8,9\}$
$x_{2,2}$: $\{2,3,4,5,6,7,8,9\}$

Events queue:

awakeOnInst($\sum_{j=0}^{2} x_{0,j} = 15,\ x_{0,0}$),
awakeOnInst($x_{0,0} + x_{1,1} + x_{2,2} = 15,\ x_{0,0}$),
awakeOnInf($\sum_{i=0}^{2} x_{i,0} = 15,\ x_{0,1}$),
awakeOnInf($\sum_{i=0}^{2} x_{i,0} = 15,\ x_{0,2}$),

...

Coming back on the example

$x_{0,2}$	$x_{1,2}$	$x_{2,2}$
$x_{0,1}$	$x_{1,1}$	$x_{2,1}$
$x_{0,0}$	$x_{1,0}$	$x_{2,0}$

$x_{0,0}$: $\{1\}$
$x_{0,1}$: $\{5, 6, 7, 8, 9\}$
$x_{0,2}$: $\{5, 6, 7, 8, 9\}$
$x_{1,0}$: $\{5, 6, 7, 8, 9\}$
$x_{1,1}$: $\{2, 3, 4, 5, 6, 7, 8, 9\}$
$x_{1,2}$: $\{2, 3, 4, 5, 6, 7, 8, 9\}$
$x_{2,0}$: $\{5, 6, 7, 8, 9\}$
$x_{2,1}$: $\{2, 3, 4, 5, 6, 7, 8, 9\}$
$x_{2,2}$: $\{2, 3, 4, 5, 6, 7, 8, 9\}$

Events queue:

awakeOnInst($x_{0,0} + x_{1,1} + x_{2,2} = 15,\ x_{0,0}$),
awakeOnInf($\sum_{i=0}^{2} x_{i,0} = 15,\ x_{0,1}$),
awakeOnInf($\sum_{i=0}^{2} x_{i,0} = 15,\ x_{0,2}$),
awakeOnInf($\sum_{i=0}^{2} x_{i,0} = 15,\ x_{1,0}$),

...

Coming back on the example

$x_{0,2}$	$x_{1,2}$	$x_{2,2}$
$x_{0,1}$	$x_{1,1}$	$x_{2,1}$
$x_{0,0}$	$x_{1,0}$	$x_{2,0}$

$x_{0,0}$: $\{1\}$
$x_{0,1}$: $\{5,6,7,8,9\}$
$x_{0,2}$: $\{5,6,7,8,9\}$
$x_{1,0}$: $\{5,6,7,8,9\}$
$x_{1,1}$: $\{5,6,7,8,9\}$
$x_{1,2}$: $\{2,3,4,5,6,7,8,9\}$
$x_{2,0}$: $\{5,6,7,8,9\}$
$x_{2,1}$: $\{2,3,4,5,6,7,8,9\}$
$x_{2,2}$: $\{5,6,7,8,9\}$

Events queue:

awakeOnInf($\sum_{i=0}^{2} x_{i,0} = 15,\ x_{0,1}$),
awakeOnInf($\sum_{i=0}^{2} x_{i,0} = 15,\ x_{0,2}$),
awakeOnInf($\sum_{i=0}^{2} x_{i,0} = 15,\ x_{1,0}$),
awakeOnInf($\sum_{i=0}^{2} x_{i,0} = 15,\ x_{1,1}$),

...

Coming back on the example

$x_{0,2}$	$x_{1,2}$	$x_{2,2}$
$x_{0,1}$	$x_{1,1}$	$x_{2,1}$
$x_{0,0}$	$x_{1,0}$	$x_{2,0}$

$x_{0,0}$: $\{1\}$
$x_{0,1}$: $\{5, 6, 7, 8, 9\}$
$x_{0,2}$: $\{5, 6, 7, 8, 9\}$
$x_{1,0}$: $\{5, 6, 7, 8, 9\}$
$x_{1,1}$: $\{5, 6, 7, 8, 9\}$
$x_{1,2}$: $\{2, 3, 4, 5, 6, 7, 8, 9\}$
$x_{2,0}$: $\{5, 6, 7, 8, 9\}$
$x_{2,1}$: $\{2, 3, 4, 5, 6, 7, 8, 9\}$
$x_{2,2}$: $\{5, 6, 7, 8, 9\}$

Events queue:

...

awakeOnInf(**alldiff** $(x_{i,j} | (i, j) \in \{0, 1, 2\}^2)$, $x_{1,0}$),

...

Coming back on the example

$x_{0,2}$	$x_{1,2}$	$x_{2,2}$
$x_{0,1}$	$x_{1,1}$	$x_{2,1}$
$x_{0,0}$	$x_{1,0}$	$x_{2,0}$

1

$x_{0,0}$: $\{1\}$
$x_{0,1}$: $\{5,6,7,8,9\}$
$x_{0,2}$: $\{5,6,7,8,9\}$
$x_{1,0}$: $\{5,6,7,8,9\}$
$x_{1,1}$: $\{5,6,7,8,9\}$
$x_{1,2}$: $\{2,3,4,5,6,7,8,9\}$
$x_{2,0}$: $\{5,6,7,8,9\}$
$x_{2,1}$: $\{2,3,4,5,6,7,8,9\}$
$x_{2,2}$: $\{5,6,7,8,9\}$

Events queue:

...

awakeOnInf($\mathbf{alldiff}(x_{i,j} | (i,j) \in \{0,1,2\}^2)$, $x_{1,0}$) $\rightsquigarrow$ **Inconsistent!**,

...

Individual utility and collective decision

Given a collective decision making problem, the preferences of an agent regarding the set of admissible decisions $\mathcal{S}$ is given by a **utility function**:

Individual utility function

Given an agent a_i and the set of admissible decisions $\mathcal{S}$, the individual utility function of a_i is a function $u_i : \mathcal{S} \rightarrow \mathbb{N}$.

The quality of a collective decision is measured using its associated **utility profile**, that is, the vector of its associated utilities.

"[Welfarism] judges a collective action on the basis of the utility levels enjoyed by the individual agents, and on those levels only" [Moulin, 1988]

Moulin, H. (1988).
Axioms of Cooperative Decision Making.
Cambridge University Press.

Individual utility and collective decision

Given a collective decision making problem, the preferences of an agent regarding the set of admissible decisions $\mathcal{S}$ is given by a **utility function**:

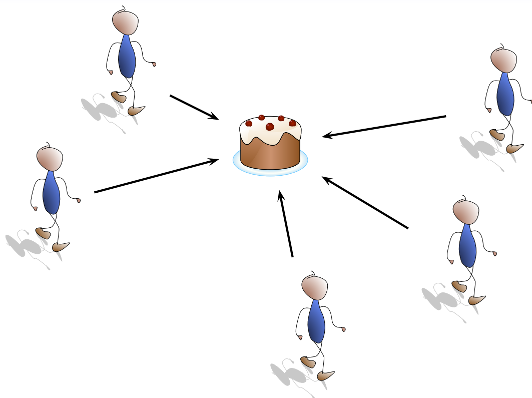
Individual utility function

Given an agent a_i and the set of admissible decisions $\mathcal{S}$, the individual utility function of a_i is a function $u_i : \mathcal{S} \rightarrow \mathbb{N}$.

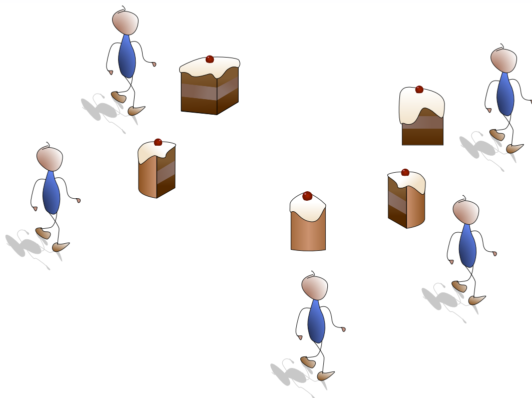
The quality of a collective decision is measured using its associated **utility profile**, that is, the vector of its associated utilities.
"[Welfarism] judges a collective action on the basis of the utility levels enjoyed by the individual agents, and on those levels only" [Moulin, 1988]

Moulin, H. (1988).
Axioms of Cooperative Decision Making.
Cambridge University Press.

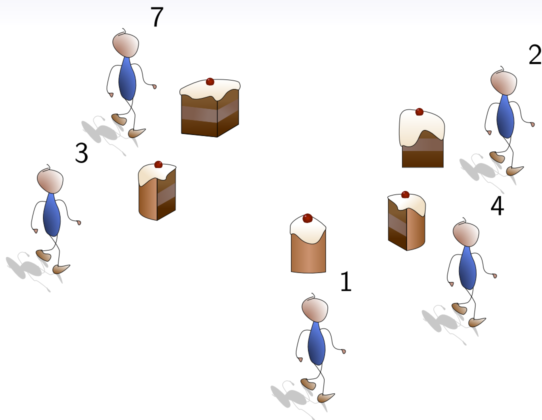
Decision making and *welfarism*



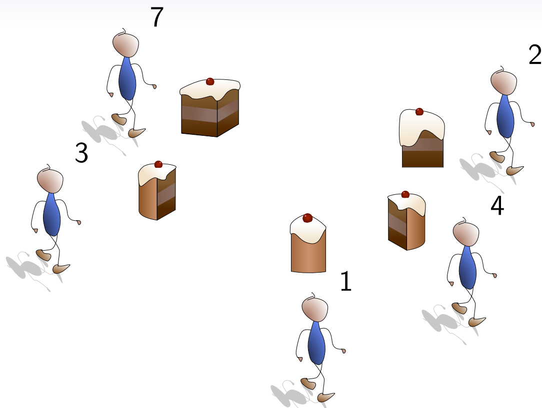
Decision making and *welfarism*



Decision making and *welfarism*



Decision making and *welfarism*



Utility profile: $\overrightarrow{u} = \langle 3, 4, 2, 1, 7 \rangle$

Social Welfare Ordering

We make use of a **Social Welfare Ordering** to compare the utility profiles.

Social Welfare Ordering

A **Social Welfare Ordering** is a preorder $\preceq$ on $\mathbb{N}^n$.

Examples

- Classical utilitarian Social Welfare Ordering.
- Egalitarian Social Welfare Ordering.
- Leximin Social Welfare Ordering.

Social Welfare Ordering

We make use of a **Social Welfare Ordering** to compare the utility profiles.

Social Welfare Ordering

A **Social Welfare Ordering** is a preorder $\preceq$ on $\mathbb{N}^n$.

Examples

- Classical utilitarian Social Welfare Ordering.
- Egalitarian Social Welfare Ordering.
- Leximin Social Welfare Ordering.

Classical Social Welfare Orderings

- Classical utilitarian SWO.
- Egalitarian SWO.
- Leximin SWO.

Classical Social Welfare Orderings

- Classical utilitarian SWO.
- Egalitarian SWO.
- Leximin SWO.

Classical utilitarian SWO [Harsanyi]

$\overrightarrow{u} \preceq \overrightarrow{v} \Leftrightarrow \sum_{i=1}^{n} u_i \leq \sum_{i=1}^{n} v_i$.

Features

The agents are utility "producers".
It is indifferent to inequalities between agents $\leadsto$ it can lead to very unfair decisions.

Classical Social Welfare Orderings

- Classical utilitarian SWO.
- Egalitarian SWO.
- Leximin SWO.

Classical utilitarian SWO [Harsanyi]

$\overrightarrow{u} \preceq \overrightarrow{v} \Leftrightarrow \sum_{i=1}^{n} u_i \leq \sum_{i=1}^{n} v_i$.

Fairness and utilitarian SWO

$\langle 10, 10, 10, 10 \rangle \preceq \langle 41, 0, 0, 0 \rangle$, whereas $\langle 10, 10, 10, 10 \rangle$ is more equitable than $\langle 41, 0, 0, 0 \rangle$.

Classical Social Welfare Orderings

- Classical utilitarian SWO.
- Egalitarian SWO.
- Leximin SWO.

Egalitarian SWO [Rawls]

$\overrightarrow{u} \preceq \overrightarrow{v} \Leftrightarrow \min_{i=1}^{n} u_i \leq \min_{i=1}^{n} v_i$.

Features

It only takes the least satisfied agent into account $\leadsto$ natural inclination to fairness.

Classical Social Welfare Orderings

- Classical utilitarian SWO.
- Egalitarian SWO.
- Leximin SWO.

Egalitarian SWO [Rawls]

$\overrightarrow{u} \preceq \overrightarrow{v} \Leftrightarrow \min_{i=1}^{n} u_i \leq \min_{i=1}^{n} v_i$.

Features

It only takes the least satisfied agent into account $\leadsto$ natural inclination to fairness.
On the other hand, it can lead to non Pareto-optimal decisions (drowning effect).

Classical Social Welfare Orderings

- Classical utilitarian SWO.
- Egalitarian SWO.
- Leximin SWO.

Egalitarian SWO [Rawls]

$\overrightarrow{u} \preceq \overrightarrow{v} \Leftrightarrow \min_{i=1}^{n} u_i \leq \min_{i=1}^{n} v_i$.

Egalitarian SWO and Pareto-efficiency

$\langle 1, 1, 1, 1 \rangle \sim \langle 1000, 1, 1000, 1000 \rangle$, whereas $\langle 1, 1, 1, 1 \rangle$ and $\langle 1000, 1, 1000, 1000 \rangle$ are very different !

Classical Social Welfare Orderings

- Classical utilitarian SWO.
- Egalitarian SWO.
- Leximin SWO.

Leximin SWO

Let $\overrightarrow{x}$ be a vector. We write $\overrightarrow{x}^{\uparrow}$ the sorted version of $\overrightarrow{x}$.
$\overrightarrow{u} \succ_{leximin} \overrightarrow{v} \Leftrightarrow \exists k$ such that $\forall i \leq k,\ u_i^{\uparrow} = v_i^{\uparrow}$ and $u_{k+1}^{\uparrow} > v_{k+1}^{\uparrow}$.
In other words, a lexicographic comparison on the sorted vectors.

Performing a leximin comparison...

Two vectors to compare: $\overrightarrow{u} = \langle 4, 10, 3, 5 \rangle$ and $\overrightarrow{v} = \langle 4, 3, 6, 6 \rangle$.

- We sort the vectors: $\begin{cases} \overrightarrow{u}^{\uparrow} = \langle 3, 4, 5, 10 \rangle \\ \overrightarrow{v}^{\uparrow} = \langle 3, 4, 6, 6 \rangle \end{cases}$
- We compare the sorted vectors lexicographically: $\overrightarrow{u}^{\uparrow} \prec_{lexico} \overrightarrow{v}^{\uparrow}$

Classical Social Welfare Orderings

- Classical utilitarian SWO.
- Egalitarian SWO.
- Leximin SWO.

Leximin SWO

Let $\overrightarrow{x}$ be a vector. We write $\overrightarrow{x^{\uparrow}}$ the sorted version of $\overrightarrow{x}$.
$\overrightarrow{u} \succ_{leximin} \overrightarrow{v} \Leftrightarrow \exists k$ such that $\forall i \leq k,\ u_i^{\uparrow} = v_i^{\uparrow}$ and $u_{k+1}^{\uparrow} > v_{k+1}^{\uparrow}$.
In other words, a lexicographic comparison on the sorted vectors.

Features

It takes all agents into account, in the order of their satisfaction level $\leadsto$ natural inclination to fairness.
It both refines the order induced by the egalitarian SWO and the Pareto order.

Classical Social Welfare Orderings

- Classical utilitarian SWO.
- Egalitarian SWO.
- Leximin SWO.

Leximin SWO

Let $\overrightarrow{x}$ be a vector. We write $\overrightarrow{x}^{\uparrow}$ the sorted version of $\overrightarrow{x}$.
$\overrightarrow{u} \succ_{leximin} \overrightarrow{v} \Leftrightarrow \exists k$ such that $\forall i \leq k,\ u_i^{\uparrow} = v_i^{\uparrow}$ and $u_{k+1}^{\uparrow} > v_{k+1}^{\uparrow}$.
In other words, a lexicographic comparison on the sorted vectors.

Leximin SWO and Pareto-efficiency

$\langle 1, 1, 1, 1 \rangle \prec \langle 1000, 1, 1000, 1000 \rangle$ (the second value of the ordered vectors is discriminatory).

The Constraint Satisfaction Problem

Classical CSP

Given : A constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$.
Is there a complete consistent instantiation v of $(\mathcal{X}, \mathcal{D}, \mathcal{C})$?

CSP with objective variable

Given : A constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ and an objective variable $o \in \mathcal{X}$, such that $d_o \subset \mathbb{N}$.
What is the maximal value α of d_o such that there is a complete consistent instantiation $\widehat{v}$ with $\widehat{v}(o) = \alpha$?

Leximin-CSP (as a multi-objective CSP)

Given : A constraint network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ and a vector of variables $\overrightarrow{u} = \langle u_1, \ldots, u_n \rangle$ ($\forall i,\ u_i \in \mathcal{X}$ and $d_{u_i} \in \mathbb{N}$) called **objective vector**.
What is the leximin-optimal vector $\langle \alpha_1, \ldots, \alpha_n \rangle$ of $\langle d_{u_1}, \ldots, d_{u_n} \rangle$ such that there is a complete consistent instantiation $\widehat{v}$ with $\widehat{v}(u_i) = \alpha_i$ forall i ?

Outline

1. Formalizing and modeling the problem
   - CSP and constraint programming
   - Fairness in collective decision making
   - Leximin and Constraint Programming
2. Two constraint programming algorithms
   - Using a set of cardinality constraints
   - Using a multiset ordering constraint
   - Sort and conquer
   - Using the saturated subsets
3. Application, tests and results
   - Benchmark and results

Algorithm 1

Using some cardinality constraints

An alternative definition of sorting

Proposition

$\langle y_1, \ldots, y_n \rangle$ is the permutation of $\langle u_1, \ldots, u_n \rangle$ sorted in non-decreasing order if and only if:

- y_1 is the maximal value such that all the u_i are g.e.q than y_1;
- y_2 is the maximal value such that at least $n-1$ values among the u_i are g.e.q than y_2 and all the u_i are g.e.q than y_1;

$$\vdots$$

- y_n is the maximal value such that at least 1 value among the u_i is g.e.q than y_n, at least 2 values among the u_i are g.e.q than y_{n-1}, $\ldots$, and all the u_i are g.e.q than y_1.

An alternative definition of sorting

Proposition

$\langle y_1, \ldots, y_n \rangle$ is the permutation of $\langle u_1, \ldots, u_n \rangle$ sorted in non-decreasing order if and only if:

- y_1 is the maximal value such that all the u_i are g.e.q than y_1;
- y_2 is the maximal value such that at least $n-1$ values among the u_i are g.e.q than y_2 and all the u_i are g.e.q than y_1;

$$\vdots$$

- y_n is the maximal value such that at least 1 value among the u_i is g.e.q than y_n, at least 2 values among the u_i are g.e.q than y_{n-1}, $\ldots$, and all the u_i are g.e.q than y_1.

An alternative definition of sorting

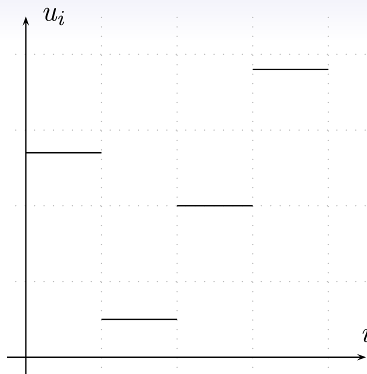
Proposition

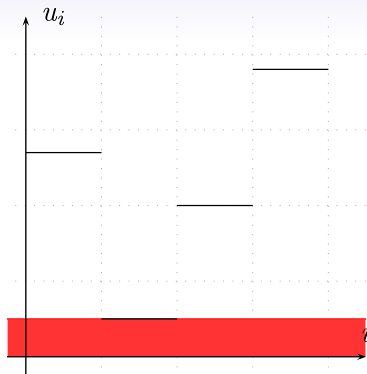
$\langle y_1, \ldots, y_n \rangle$ is the permutation of $\langle u_1, \ldots, u_n \rangle$ sorted in non-decreasing order if and only if:

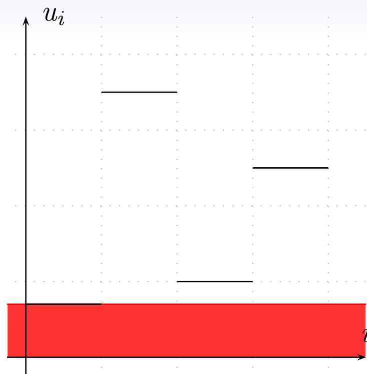
- y_1 is the maximal value such that all the u_i are g.e.q than y_1;
- y_2 is the maximal value such that at least $n-1$ values among the u_i are g.e.q than y_2 and all the u_i are g.e.q than y_1;

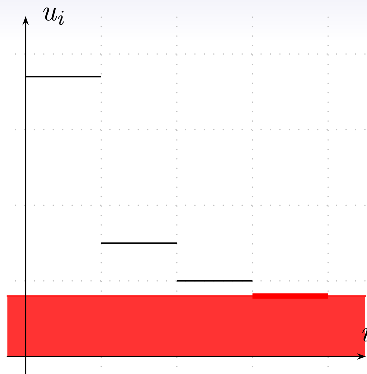
$$\vdots$$

- y_n is the maximal value such that at least 1 value among the u_i is g.e.q than y_n, at least 2 values among the u_i are g.e.q than y_{n-1}, $\ldots$, and all the u_i are g.e.q than y_1.

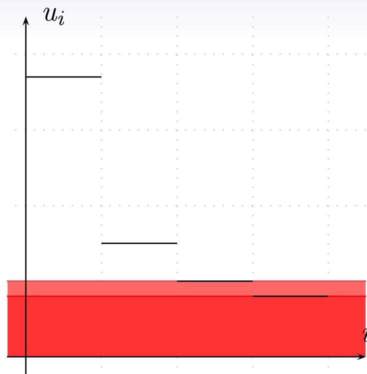

u_i
i

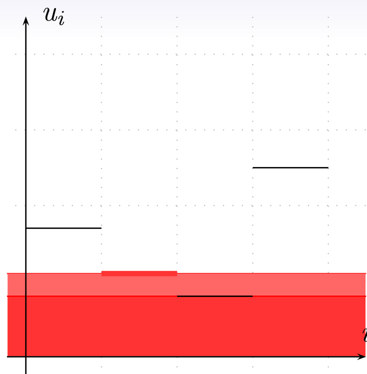

u_i
i

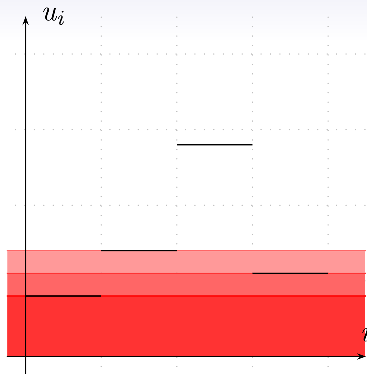

u_i
i

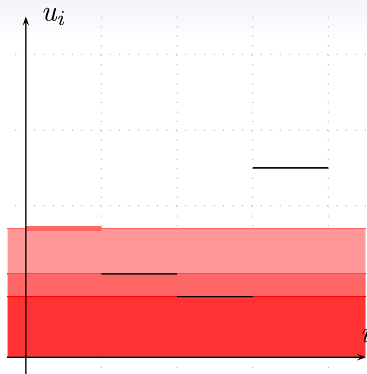


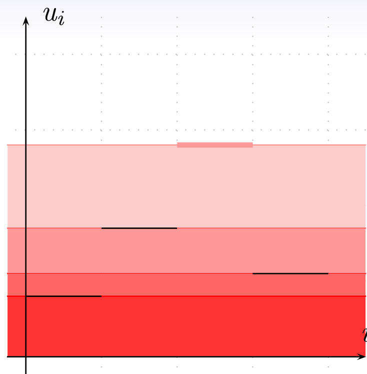
u_i
i


u_i
i


u_i
i


u_i
i


u_i
i


u_i
i

The meta-constraint AtLeast

The algorithm is based on the meta-constraint **AtLeast**

Constraint **AtLeast** [Van Hentenryck et al., 1992]

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

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

Algorithm description

Input: $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; $\langle u_1, \ldots, u_n \rangle \in \mathcal{X}^n$ objective variables (utilities of agents)
Output: The leximin-optimal instantiation or "Inconsistent"

Algorithm

if $\text{solve}(\mathcal{X}, \mathcal{D}, \mathcal{C}) =$ **"Inconsistent" then return "Inconsistent"**

$\mathcal{X}' \leftarrow \mathcal{X} \cup \{y_1, \ldots, y_n\}$;
$\mathcal{D}' \leftarrow \mathcal{D} \cup \{[\![m, M]\!], \ldots, [\![m, M]\!]\}$;
$\mathcal{C}' \leftarrow \mathcal{C}$;

for $i \leftarrow 1$ **to** n **do**
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;
  $\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;
  $d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;
end
return $\widehat{v}_{\downarrow \mathcal{X}}$

Algorithm description

Input: $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; $\langle u_1, \ldots, u_n \rangle \in \mathcal{X}^n$ objective variables (utilities of agents)
Output: The leximin-optimal instantiation or "Inconsistent"

Algorithm

if $\text{solve}(\mathcal{X}, \mathcal{D}, \mathcal{C}) =$ **"Inconsistent" then return "Inconsistent"**

Introduction of the variables $\mathrm{y_i}$

$\mathcal{X}' \leftarrow \mathcal{X} \cup \{y_1, \ldots, y_n\}$;
$\mathcal{D}' \leftarrow \mathcal{D} \cup \{[\![m, M]\!], \ldots, [\![m, M]\!]\}$;
$\mathcal{C}' \leftarrow \mathcal{C}$;

for $i \leftarrow 1$ **to** n **do**
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;
  $\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;
  $d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;
end
return $\widehat{v}_{\downarrow \mathcal{X}}$

Algorithm description

Input: $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; $\langle u_1, \ldots, u_n \rangle \in \mathcal{X}^n$ objective variables (utilities of agents)
Output: The leximin-optimal instantiation or "Inconsistent"

Algorithm

if solve$(\mathcal{X}, \mathcal{D}, \mathcal{C})$ = **"Inconsistent" then return "Inconsistent"**

Introduction of the variables y_i

$\mathcal{X}' \leftarrow \mathcal{X} \cup \{y_1, \ldots, y_n\}$;
$\mathcal{D}' \leftarrow \mathcal{D} \cup \{[\![m, M]\!], \ldots, [\![m, M]\!]\}$;
$\mathcal{C}' \leftarrow \mathcal{C}$;

Introduction of a constraint on y_i

for $i \leftarrow 1$ **to** n **do**
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;
  $\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;
  $d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;
end
return $\widehat{v}_{\downarrow \mathcal{X}}$

Algorithm description

Input: $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; $\langle u_1, \ldots, u_n \rangle \in \mathcal{X}^n$ objective variables (utilities of agents)
Output: The leximin-optimal instantiation or "Inconsistent"

Algorithm

if $\text{solve}(\mathcal{X}, \mathcal{D}, \mathcal{C}) =$ **"Inconsistent" then return "Inconsistent"**

Introduction of the variables $\mathrm{y_i}$

$\mathcal{X}' \leftarrow \mathcal{X} \cup \{y_1, \ldots, y_n\}$;
$\mathcal{D}' \leftarrow \mathcal{D} \cup \{[\![m, M]\!], \ldots, [\![m, M]\!]\}$;
$\mathcal{C}' \leftarrow \mathcal{C}$;

Introduction of a constraint on $\mathrm{y_i}$

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;
$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$; Computation of $\mathrm{y_i}$
$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;

end

return $\widehat{v}_{\downarrow \mathcal{X}}$

Algorithm description

Input: $(\mathcal{X}, \mathcal{D}, \mathcal{C})$; $\langle u_1, \ldots, u_n \rangle \in \mathcal{X}^n$ objective variables (utilities of agents)
Output: The leximin-optimal instantiation or "Inconsistent"

Algorithm

if $\text{solve}(\mathcal{X}, \mathcal{D}, \mathcal{C})$ = **"Inconsistent" then return "Inconsistent"**

Introduction of the variables y_i

$\mathcal{X}' \leftarrow \mathcal{X} \cup \{y_1, \ldots, y_n\}$;
$\mathcal{D}' \leftarrow \mathcal{D} \cup \{[\![m, M]\!], \ldots, [\![m, M]\!]\}$;
$\mathcal{C}' \leftarrow \mathcal{C}$;

Introduction of a constraint on y_i

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;

$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$; — Computation of y_i

$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$; — Updating of the domains

end

return $\widehat{v}_{\downarrow \mathcal{X}}$

Running of the algorithm on an example

A sharing problem

- An allocation problem with 3 agents and 3 objects.
- Constraint: the allocation has to be a matching (*i.e.* one different object per agent).
- The utility functions of the agents are defined by a set of weights $w(a_i, o_j)$, the utility function of the agent a_i being $u_i = \sum_{o_j | a_i \leftarrow o_j} w(a_i, o_j)$.
- The weights are the following:

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

Running of the algorithm on an example

Algorithm :

$\mathcal{X}' \leftarrow \mathcal{X} \cup \{y_1, \ldots, y_n\}$;

$\mathcal{D}' \leftarrow \mathcal{D} \cup \{[\![m, M]\!], \ldots, [\![m, M]\!]\}$;

$\mathcal{C}' \leftarrow \mathcal{C}$;

Initial constraint network:

$\mathcal{X}' = \{a_1, a_2, a_3, u_1, u_2, u_3\} \cup \{\mathbf{y_1}, \mathbf{y_2}, \mathbf{y_3}\}$

$\mathcal{D}' = \{\{o_1, o_2, o_3\}, \{o_1, o_2, o_3\}, \{o_1, o_2, o_3\}, [\![3, 7]\!], [\![3, 9]\!], [\![1, 7]\!]\}$
$\cup \{[\![\mathbf{1}, \mathbf{9}]\!], [\![\mathbf{1}, \mathbf{9}]\!], [\![\mathbf{1}, \mathbf{9}]\!]\}$

$\mathcal{C}' = \{\textbf{alldifferent}(\{a_1, a_2, a_3\}), u_i = \sum_{o_j} (a_i = o_j)\ \forall i\}$

Running of the algorithm on an example

Algorithm :

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;

$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;

$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;

end

a_1	a_2	a_3	u_1	u_2	u_3	y_1	y_2	y_3
o_1	o_2	o_3	3	9	1	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$
o_1	o_3	o_2	3	8	7	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$
o_2	o_1	o_3	5	3	1	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$
o_2	o_3	o_1	5	8	3	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$
o_3	o_1	o_2	7	3	7	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$
o_3	o_2	o_1	7	9	3	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$	$\{1,\ldots,9\}$

Running of the algorithm on an example

Algorithm :

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;

$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;

$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;

end

a_1	a_2	a_3	u_1	u_2	u_3	y_1	y_2	y_3
o_1	o_2	o_3	3	9	1	**1**	{1,...,9}	{1,...,9}
o_1	o_3	o_2	3	8	7	**{1,2,3}**	{1,...,9}	{1,...,9}
o_2	o_1	o_3	5	3	1	**1**	{1,...,9}	{1,...,9}
o_2	o_3	o_1	5	8	3	**{1,2,3}**	{1,...,9}	{1,...,9}
o_3	o_1	o_2	7	3	7	**{1,2,3}**	{1,...,9}	{1,...,9}
o_3	o_2	o_1	7	9	3	**{1,2,3}**	{1,...,9}	{1,...,9}

Running of the algorithm on an example

Algorithm :

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;

$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;

$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;

end

a_1	a_2	a_3	u_1	u_2	u_3	y_1	y_2	y_3
o_1	o_2	o_3	3	9	1	1	{1,. . . ,9}	{1,. . . ,9}
o_1	**o_3**	**o_2**	**3**	**8**	**7**	{1,2,**3**}	**{1,. . . ,9}**	**{1,. . . ,9}**
o_2	o_1	o_3	5	3	1	1	{1,. . . ,9}	{1,. . . ,9}
o_2	**o_3**	**o_1**	**5**	**8**	**3**	{1,2,**3**}	**{1,. . . ,9}**	**{1,. . . ,9}**
o_3	**o_1**	**o_2**	**7**	**3**	**7**	{1,2,**3**}	**{1,. . . ,9}**	**{1,. . . ,9}**
o_3	**o_2**	**o_1**	**7**	**9**	**3**	{1,2,**3**}	**{1,. . . ,9}**	**{1,. . . ,9}**

Running of the algorithm on an example

Algorithm :

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;

$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;

$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;

end

a_1	a_2	a_3	u_1	u_2	u_3	y_1	y_2	y_3
o_1	o_2	o_3	3	9	1	1	{1,...,9}	{1,...,9}
o_1	o_3	o_2	3	8	7	3	{1,...,9}	{1,...,9}
o_2	o_1	o_3	5	3	1	1	{1,...,9}	{1,...,9}
o_2	o_3	o_1	5	8	3	3	{1,...,9}	{1,...,9}
o_3	o_1	o_2	7	3	7	3	{1,...,9}	{1,...,9}
o_3	o_2	o_1	7	9	3	3	{1,...,9}	{1,...,9}

Running of the algorithm on an example

Algorithm :

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n-i+1)\}$;

$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;

$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;

end

a_1	a_2	a_3	u_1	u_2	u_3	y_1	y_2	y_3
o_1	o_2	o_3	3	9	1	1	{1,...,9}	{1,...,9}
o_1	o_3	o_2	3	8	7	3	**{3,...,7}**	{1,...,9}
o_2	o_1	o_3	5	3	1	1	{1,...,9}	{1,...,9}
o_2	o_3	o_1	5	8	3	3	**{3,...,5}**	{1,...,9}
o_3	o_1	o_2	7	3	7	3	**{3,...,7}**	{1,...,9}
o_3	o_2	o_1	7	9	3	3	**{3,...,7}**	{1,...,9}

Running of the algorithm on an example

Algorithm :

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;

$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;

$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;

end

a_1	a_2	a_3	u_1	u_2	u_3	y_1	y_2	y_3
o_1	o_2	o_3	3	9	1	1	{1,...,9}	{1,...,9}
$\mathbf{o_1}$	$\mathbf{o_3}$	$\mathbf{o_2}$	**3**	**8**	**7**	**3**	{3,...,**7**}	**{1,...,9}**
o_2	o_1	o_3	5	3	1	1	{1,...,9}	{1,...,9}
o_2	o_3	o_1	5	8	3	3	{3,...,5}	{1,...,9}
$\mathbf{o_3}$	$\mathbf{o_1}$	$\mathbf{o_2}$	**7**	**3**	**7**	**3**	{3,...,**7**}	**{1,...,9}**
$\mathbf{o_3}$	$\mathbf{o_2}$	$\mathbf{o_1}$	**7**	**9**	**3**	**3**	{3,...,**7**}	**{1,...,9}**

Running of the algorithm on an example

Algorithm :

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;

$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;

$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;

end

a_1	a_2	a_3	u_1	u_2	u_3	y_1	y_2	y_3
o_1	o_2	o_3	3	9	1	1	{1,. . . ,9}	{1,. . . ,9}
o_1	o_3	o_2	3	8	7	3	**7**	{1,. . . ,9}
o_2	o_1	o_3	5	3	1	1	{1,. . . ,9}	{1,. . . ,9}
o_2	o_3	o_1	5	8	3	3	{3,. . . ,5}	{1,. . . ,9}
o_3	o_1	o_2	7	3	7	3	**7**	{1,. . . ,9}
o_3	o_2	o_1	7	9	3	3	**7**	{1,. . . ,9}

Running of the algorithm on an example

Algorithm :

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n-i+1)\}$;

$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;

$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;

end

a_1	a_2	a_3	u_1	u_2	u_3	y_1	y_2	y_3
o_1	o_2	o_3	3	9	1	1	{1,...,9}	{1,...,9}
o_1	o_3	o_2	3	8	7	3	7	**{7,8}**
o_2	o_1	o_3	5	3	1	1	{1,...,9}	{1,...,9}
o_2	o_3	o_1	5	8	3	3	{3,...,5}	{1,...,9}
o_3	o_1	o_2	7	3	7	3	7	**7**
o_3	o_2	o_1	7	9	3	3	7	**{7,8,9}**

Running of the algorithm on an example

Algorithm :

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;

$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;

$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;

end

a_1	a_2	a_3	u_1	u_2	u_3	y_1	y_2	y_3
o_1	o_2	o_3	3	9	1	1	{1,...,9}	{1,...,9}
o_1	o_3	o_2	3	8	7	3	7	{7,8}
o_2	o_1	o_3	5	3	1	1	{1,...,9}	{1,...,9}
o_2	o_3	o_1	5	8	3	3	{3,...,5}	{1,...,9}
o_3	o_1	o_2	7	3	7	3	7	7
$\mathbf{o_3}$	$\mathbf{o_2}$	$\mathbf{o_1}$	**7**	**9**	**3**	**3**	**7**	{7,8,**9**}

Running of the algorithm on an example

Algorithm :

for $i \leftarrow 1$ **to** n **do**

$\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{AtLeast}(\{u_1 \geq y_i, \ldots, u_n \geq y_i\}, n - i + 1)\}$;

$\widehat{v} \leftarrow \mathbf{maximize}(y_i, (\mathcal{X}, \mathcal{D}, \mathcal{C}))$;

$d_{y_i} \leftarrow \{\widehat{v}(y_i)\}$;

end

a_1	a_2	a_3	u_1	u_2	u_3	y_1	y_2	y_3
o_1	o_2	o_3	3	9	1	1	{1,...,9}	{1,...,9}
o_1	o_3	o_2	3	8	7	3	7	{7,8}
o_2	o_1	o_3	5	3	1	1	{1,...,9}	{1,...,9}
o_2	o_3	o_1	5	8	3	3	{3,...,5}	{1,...,9}
o_3	o_1	o_2	7	3	7	3	7	7
o_3	o_2	o_1	7	9	3	3	7	**9**

Algorithm 2

An algorithm inspired by a multiset ordering constraint

A second algorithm

We introduce a new constraint:

Constraint Leximin

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

This constraint is based on the multiset ordering constraint, introduced in [Frisch et al., 2003] (algorithm to enforce GAC in $O(n \log(n))$).

Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., and Walsh, T. (2003).
Multiset ordering constraints.
In *Proc. of IJCAI'03*, Acapulco, Mexico.

Description of the algorithm

Algorithm

$v \leftarrow \mathbf{solve}(\mathcal{X}, \mathcal{D}, \mathcal{C})$

while $v \neq$ **"Inconsistent"** **do**

$\quad \widehat{v} \leftarrow v$

$\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{Leximin}(\widehat{v}(\overrightarrow{u}), \overrightarrow{u})\}$

$\quad v \leftarrow \mathbf{solve}(\mathcal{X}, \mathcal{D}, \mathcal{C})$

end

if $\widehat{v} \neq \mathbf{null}$ **then** **return** $\widehat{v}$

else **return** **"Inconsistent"**

Algorithm 3

Sort and conquer

We introduce the sorted version of the objective vector

Idea : Introduce additional variables $\langle y_1, \ldots, y_n \rangle$ representing the sorted version of the objective vector, and perform successive maximizations.

1. Maximize y_1 : $\widehat{y_1}$.
2. Maximize y_2 under the constraint $y_1 = \widehat{y_1}$: $\widehat{y_2}$.

n. Maximize y_n under the constraints $y_1 = \widehat{y_1}, \ldots, y_{n-1} = \widehat{y_{n-1}}$.

We introduce the sorted version of the objective vector

Idea : Introduce additional variables $\langle y_1, \ldots, y_n \rangle$ representing the sorted version of the objective vector, and perform successive maximizations.

1. Maximize y_1 : $\widehat{y_1}$.
2. Maximize y_2 under the constraint $y_1 = \widehat{y_1}$: $\widehat{y_2}$.

n. Maximize y_n under the constraints $y_1 = \widehat{y_1}, \ldots, y_{n-1} = \widehat{y_{n-1}}$.

We introduce the sorted version of the objective vector

Idea : Introduce additional variables $\langle y_1, \ldots, y_n \rangle$ representing the sorted version of the objective vector, and perform successive maximizations.

1. Maximize y_1 : $\widehat{y_1}$.
2. Maximize y_2 under the constraint $y_1 = \widehat{y_1}$: $\widehat{y_2}$.

n. Maximize y_n under the constraints $y_1 = \widehat{y_1}, \ldots, y_{n-1} = \widehat{y_{n-1}}$.

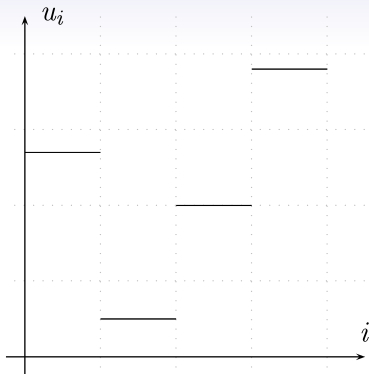
We introduce the sorted version of the objective vector

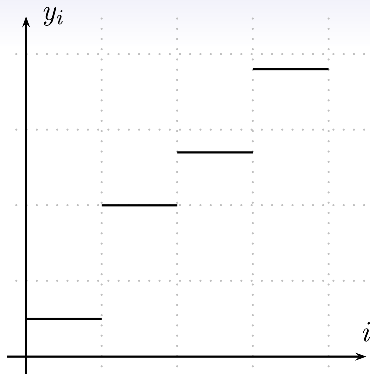
Idea : Introduce additional variables $\langle y_1, \ldots, y_n \rangle$ representing the sorted version of the objective vector, and perform successive maximizations.

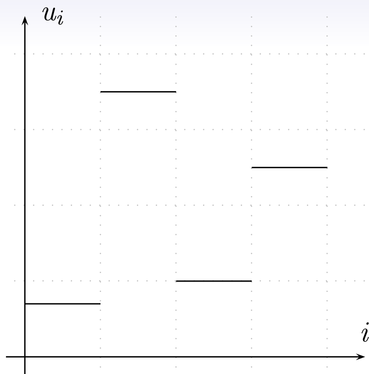
1. Maximize y_1 : $\widehat{y_1}$.
2. Maximize y_2 under the constraint $y_1 = \widehat{y_1}$: $\widehat{y_2}$.

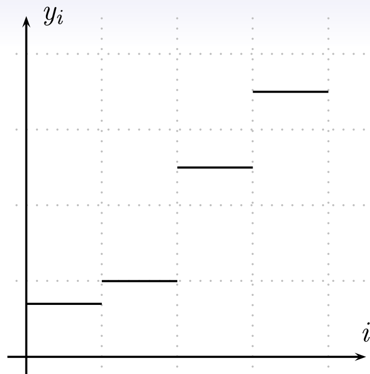
$$\vdots$$

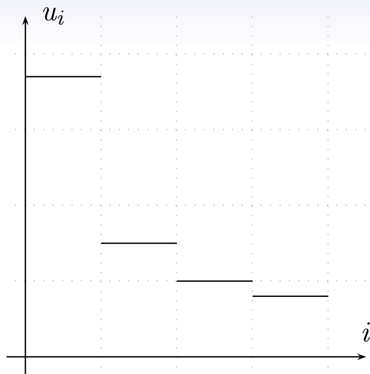
n. Maximize y_n under the constraints $y_1 = \widehat{y_1}, \ldots, y_{n-1} = \widehat{y_{n-1}}$.

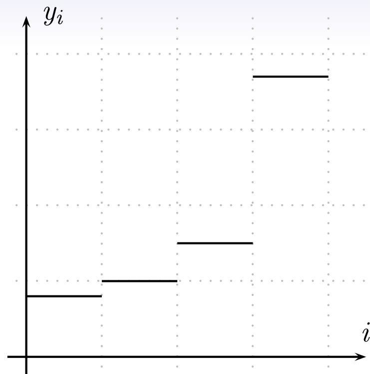

u_i
i

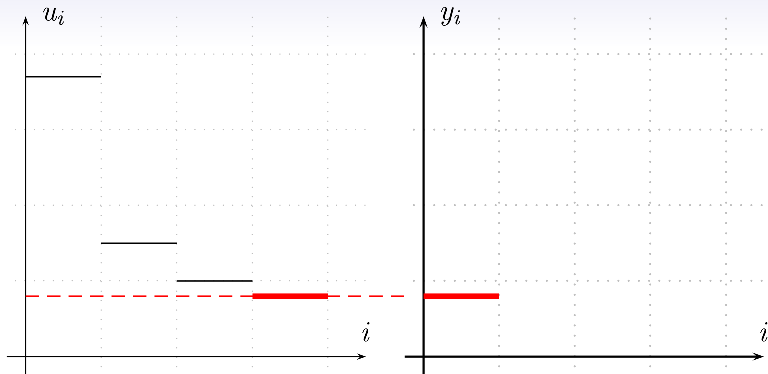

y_i
i


u_i
i

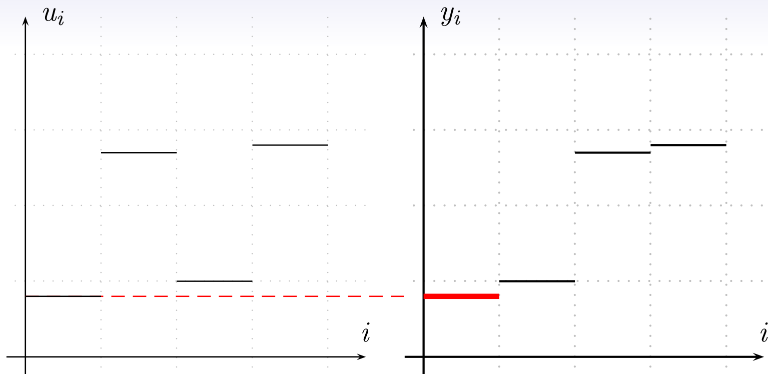

y_i
i

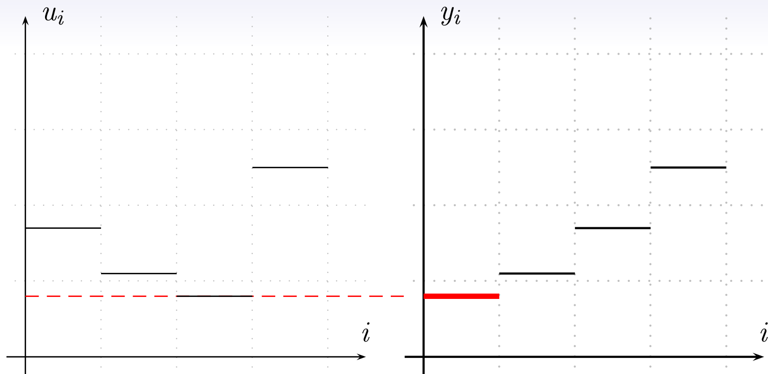

u_i
i

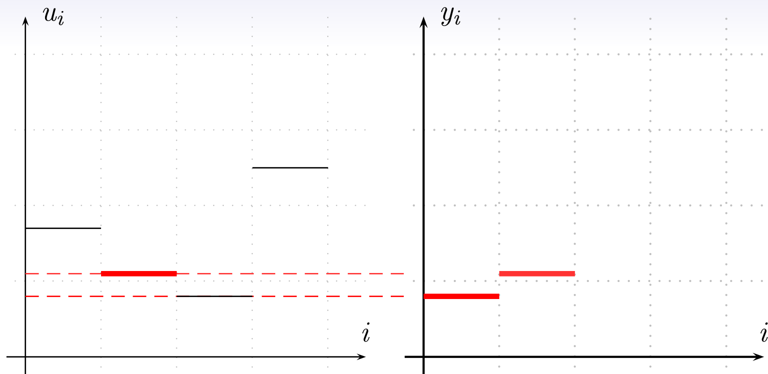

y_i
i

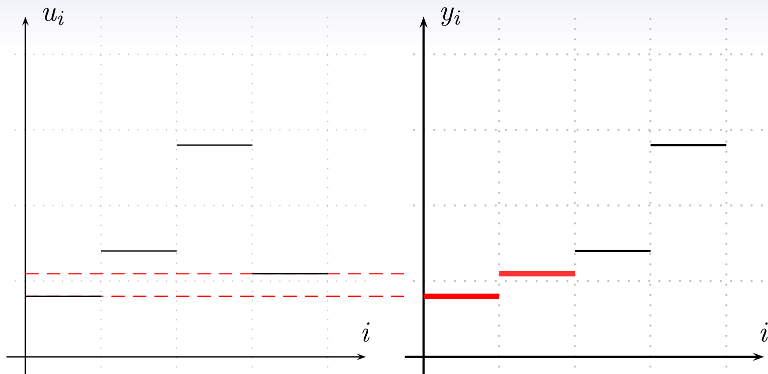


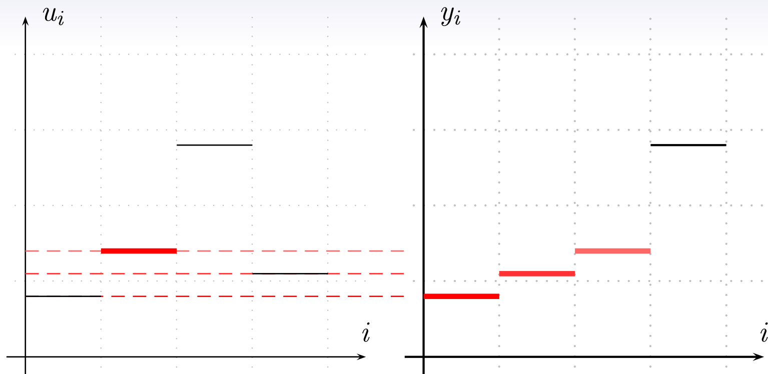
u_i
i
y_i
i

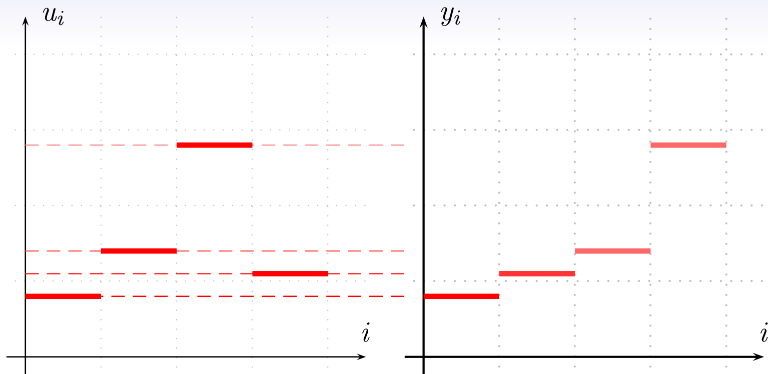

u_i
i
y_i
i


u_i
i
y_i
i


u_i
i
y_i
i


u_i
i
y_i
i


u_i
i
y_i
i



u_i
i
y_i
i

A sorting constraint

How to ensure that $\langle y_1, \ldots, y_n \rangle$ is the sorted version of the objective vector ?

Constraint Sort

Let $\overrightarrow{x}$ and $\overrightarrow{x'}$ be two vectors of variables of the same size. The constraint $\mathbf{Sort}(\overrightarrow{x}, \overrightarrow{x'})$ holds on the variables of $\overrightarrow{x}$ and $\overrightarrow{x'}$ and only allows the tuples such that $\overrightarrow{x'}$ is the sorted version in incerasing order of $\overrightarrow{x}$.

We use the work from [Bleuzen-Guernalec and Colmerauer, 1997] (a filtering algorithm running in $O(n \log(n))$), and more recently, of [Mehlhorn and Thiel, 2000].

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

Mehlhorn, K. and Thiel, S. (2000).

A sorting constraint

How to ensure that $\langle y_1, \ldots, y_n \rangle$ is the sorted version of the objective vector ?

Constraint **Sort**

Let $\overrightarrow{x}$ and $\overrightarrow{x'}$ be two vectors of variables of the same size. The constraint $\mathbf{Sort}(\overrightarrow{x}, \overrightarrow{x'})$ holds on the variables of $\overrightarrow{x}$ and $\overrightarrow{x'}$ and only allows the tuples such that $\overrightarrow{x'}$ is the sorted version in incerasing order of $\overrightarrow{x}$.

We use the work from [Bleuzen-Guernalec and Colmerauer, 1997] (a filtering algorithm running in $O(n \log(n))$), and more recently, of [Mehlhorn and Thiel, 2000].

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

Mehlhorn, K. and Thiel, S. (2000).

The algorithm based on the sorting constraint

Algorithm

if $\text{solve}(\mathcal{X}, \mathcal{D}, \mathcal{C}) =$ **"Inconsistent"** **then return** **"Inconsistent"**

$\mathcal{X}' \leftarrow \mathcal{X} \cup \{Y_1, \ldots, Y_n\}$

$\mathcal{D}' \leftarrow \mathcal{D} \cup \{\mathcal{D}_{Y_1}, \ldots, \mathcal{D}_{Y_n}\}$, $\mathcal{D}_{Y_i} = [\![\min_j(\underline{U_j}), \max_j(\overline{U_j})]\!]$

$\mathcal{C}' \leftarrow \mathcal{C} \cup \{\textbf{Sort}(\overrightarrow{U}, \overrightarrow{Y})\}$ **for** $i \leftarrow 1$ **to** n **do**

$\widehat{v}_{(i)} \leftarrow \textbf{maximize}(\mathcal{X}', \mathcal{D}', \mathcal{C}', Y_i)$ $\mathcal{D}_{Y_i} \leftarrow \{\widehat{v}_{(i)}(Y_i)\}$

end

return $\widehat{v}_{(n)\downarrow\mathcal{X}}$

Algorithm 4

Using the saturated subsets

Principle of the algorithm

This algorithm comes from the litterature on fuzzy CSPs [Dubois and Fortemps, 1999].

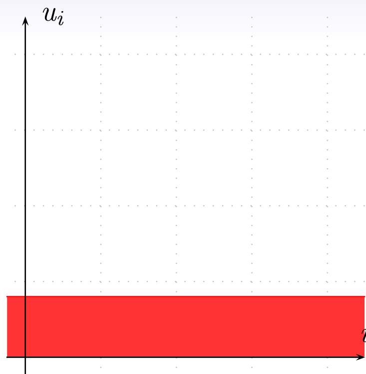
Saturated subset

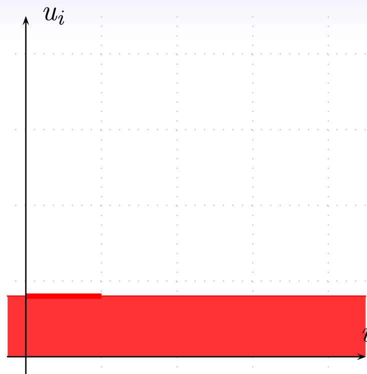
Let $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint networtk, $\overrightarrow{u}$ be a set of objective variables, and α be the maximal admissible value of $\min(\overrightarrow{u})$. A subset $\mathcal{S}$ of variables from $\overrightarrow{u}$ is said **saturated** if there is a solution with all $u_i \in \mathcal{S}$ instantiated to α and all $u_i \notin \mathcal{S}$ instantiated to a strictly greater value than α.

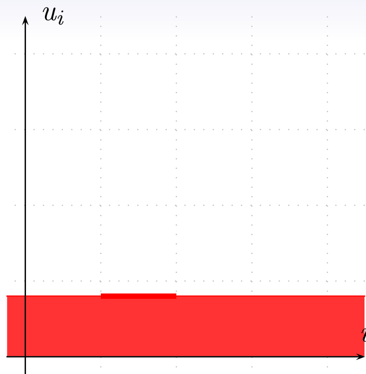
Dubois, D. and Fortemps, P. (1999).
Computing improved optimal solutions to max-min flexible constraint satisfaction problems.
European Journal of Operational Research.

Principle of the algorithm

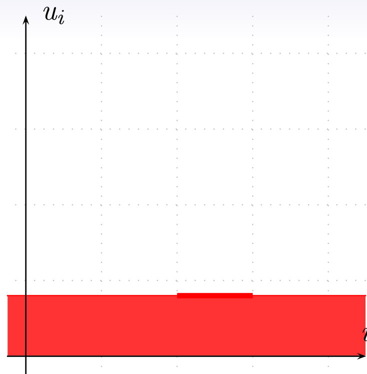
- At each step of the algorithm, we maximize the current component y_i of the leximin vector (as in the previous algorithms).
- For each cardinality-minimal saturated subset $\mathcal{S}$:
  - all the $u_i \in \mathcal{S}$ are fixed to $\widehat{y_i}$ and are removed from $\overrightarrow{u}$,
  - we restart the procedure on the new constraint network until no variable remains in $\overrightarrow{u}$.
- At the end, we compare all the potential leximin-optimal vectors found.

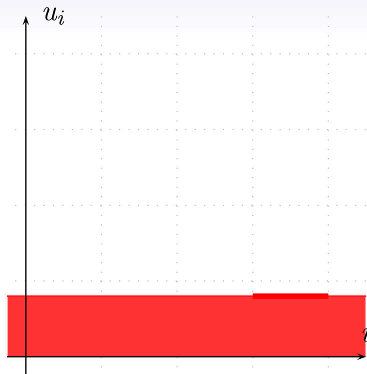

u_i
i

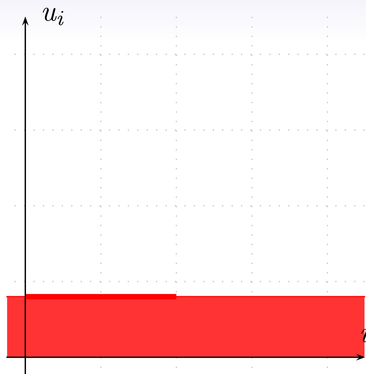

u_i
i

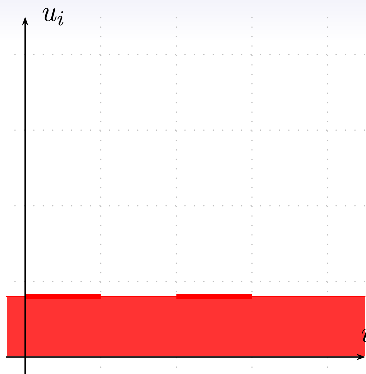


u_i
i


u_i
i


u_i
i


u_i
i


u_i
i

Major drawback of the algorithm

In the worst case, we can have to explore all the possible subsets of $\overrightarrow{u}$ (if the cardinality-minimal saturated subset is of size n) $\leadsto$ potentially an exponential number of successive resolutions.

Algorithm thus to be used only if we are sure that the cardinality-minimal saturated subsets are of little size.

Outline

1. Formalizing and modeling the problem
   - CSP and constraint programming
   - Fairness in collective decision making
   - Leximin and Constraint Programming
2. Two constraint programming algorithms
   - Using a set of cardinality constraints
   - Using a multiset ordering constraint
   - Sort and conquer
   - Using the saturated subsets
3. Application, tests and results
   - Benchmark and results

Implementation

Implementation of the algorithms:

The two algorithms have been implemented using Choco[F. Laburthe and the OCRE project team, 2000] in Java, and the first one has also been implemented using CPLEX.

Remark

Our specific constraint **AtLeast** can be encoded with a set of linear constraints: $\mathbf{AtLeast}(\{x_1 \geq y, \ldots, x_n \geq y\}, k) \Leftrightarrow$ $\{x_1 + \delta_1 \overline{y} \geq y, \ldots, x_n + \delta_n \overline{y} \geq y, \sum_{i=1}^{n} \delta_i \leq n - k\}$, with $\{\delta_1, \ldots, \delta_n\}$ 0–1 variables.

F. Laburthe and the OCRE project team (2000).
CHOCO: Implementing a CP kernel.
In *Proceedings of TRICKS'2000, Workshop on techniques for implementing Constraint Programming systems*, Singapore.
http://sourceforge.net/projects/choco.

Implementation

Implementation of the algorithms:

The two algorithms have been implemented using Choco[F. Laburthe and the OCRE project team, 2000] in Java, and the first one has also been implemented using CPLEX.

Remark

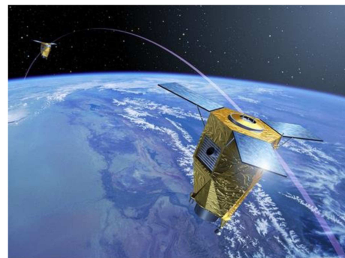
Our specific constraint **AtLeast** can be encoded with a set of linear constraints:
$\textbf{AtLeast}(\{x_1 \geq y, \ldots, x_n \geq y\}, k) \Leftrightarrow$
$\{x_1 + \delta_1 \overline{y} \geq y, \ldots, x_n + \delta_n \overline{y} \geq y, \sum_{i=1}^{n} \delta_i \leq n - k\}$, with $\{\delta_1, \ldots, \delta_n\}$ 0–1 variables.

F. Laburthe and the OCRE project team (2000).
CHOCO: Implementing a CP kernel.
In *Proceedings of TRICKS'2000, Workshop on techniques for implementing Constraint Programming systems*, Singapore.
http://sourceforge.net/projects/choco.

Our real world application

The algorithms have been tested using a (very) simplified model extracted from a real world application:

- Fair share of a constellation of Earth Observation Satellites cofunded by several countries.
- Our simplified model is a multiagent resource allocation with consumption and volume resources.



Instance generator:

We implemented a random instance generator for our simplified problem.

Available online: `http://www.cert.fr/dcsd/THESES/sbouveret/benchmark`

Combinatorial auctions

We have also tested our algorithms on combinatorial auctions instances. A brief description:

- a set of bidders (individuals) and a set of objects;
- each bidder can place some bids, a bid being a set of objects associated with the price the bidder is willing to pay for this set;
- in the traditional formulation, we have to maximize the revenue of the auctioneer (the sum of the prices of non-overlapping selected bids);
- in our formulation, we have to maximize the leximin preorder on the potential utility vectors.

Realistic instances have been generated using CATS (`http://cats.stanford.edu`).

General tendency of the results

- The algorithm based on the **AtLeast** constraint is often the most efficient, followed by the algorithm based on the **Sort** constraint (which has similar running times, except when the number of agents increases).
- The algorithm inspired by the **Multiset ordering** constraint is efficient on the instances of our resource allocation problem with a high number of agents, but completely inefficient with the combinatorial auctions instances.
- The algorithm from [Dubois and Fortemps, 1999] explodes when the number of equal components in the leximin-optimal vector increases.
- Cplex completely outperforms Choco.

Summary

- **Studied problem:** computation of a leximin-optimal allocation of a constraint network.
- **Justification:** the leximin preorder ensures some interesting properties of fairness and efficiency for collective decision making problems.
- **Algorithms:** introduction of two algorithms (the first one being original) based on the CP framework.
- **Implementation:** implementation and testing of the algorithms in Java with Choco (and CPLEX for the first one).

Summary

- **Studied problem:** computation of a leximin-optimal allocation of a constraint network.
- **Justification:** the leximin preorder ensures some interesting properties of fairness and efficiency for collective decision making problems.
- **Algorithms:** introduction of two algorithms (the first one being original) based on the CP framework.
- **Implementation:** implementation and testing of the algorithms in Java with Choco (and CPLEX for the first one).

Summary

- **Studied problem:** computation of a leximin-optimal allocation of a constraint network.
- **Justification:** the leximin preorder ensures some interesting properties of fairness and efficiency for collective decision making problems.
- **Algorithms:** introduction of two algorithms (the first one being original) based on the CP framework.
- **Implementation:** implementation and testing of the algorithms in Java with Choco (and CPLEX for the first one).

Summary

- **Studied problem:** computation of a leximin-optimal allocation of a constraint network.
- **Justification:** the leximin preorder ensures some interesting properties of fairness and efficiency for collective decision making problems.
- **Algorithms:** introduction of two algorithms (the first one being original) based on the CP framework.
- **Implementation:** implementation and testing of the algorithms in Java with Choco (and CPLEX for the first one).

Current and future work

- **Possible extensions:**
  - More general and softer modeling of fairness (*e.g.* OWA).
  - Applying the algorithms to other practical fields and applications.

Current and future work

- **Possible extensions:**
  - More general and softer modeling of fairness (*e.g.* OWA).
  - Applying the algorithms to other practical fields and applications.

Current and future work

- **Possible extensions:**
  - More general and softer modeling of fairness (*e.g.* OWA).
  - Applying the algorithms to other practical fields and applications.

This is the end.

For more information:
sylvain.bouveret@cert.fr
http://www.cert.fr/dcsd/THESES/sbouveret

Random instance generator available online:
http://www.cert.fr/dcsd/THESES/sbouveret/benchmark