Chapitre 5

Préordre leximin et programmation par contraintes

Lors de l'ensemble des chapitres précédents, nous avons introduit entre autres un modèle formel générique permettant d'exprimer des instances du problème de partage de biens indivisibles, et nous avons aussi mis en évidence des résultats de complexité liés à l'expression compacte de ces instances. La partie que nous abordons maintenant a une vocation beaucoup plus pratique. Dans ce chapitre, nous allons nous pencher sur un problème algorithmique particulier pour lequel nous allons proposer plusieurs méthodes de résolution. Ces approches seront testées de manière expérimentale dans le chapitre suivant.

5.1 Exposé du problème

Le problème de maximisation de l'utilité collective dans le modèle de problème de partage que nous avons introduit au chapitre 3 se ramène finalement à un problème d'optimisation combinatoire très classique, pouvant être facilement spécifié et implanté dans un cadre de modélisation et de résolution de problèmes tel que celui de la programmation par contraintes. Cependant, cette traduction s'applique difficilement au préordre leximin, qui est pourtant un ordre de bien-être collectif pertinent dans un contexte de recherche d'équité, comme nous l'avons vu au chapitre 1. Le problème d'optimisation leximin dépasse en outre largement le cadre particulier des problèmes de partage équitable. C'est pourquoi nous avons choisi de nous y intéresser : il s'agit d'un problème algorithmique non trivial, pertinent dans le domaine du partage, et dont la portée dépasse largement ce cadre particulier.

5.1.1 Retour sur le préordre leximin

Le préordre leximin, introduit dans la définition 1.32 page 42 en tant qu'ordre de bien-être collectif, a de nombreuses vertus, que nous rappelons brièvement. Comme il s'agit d'un raffinement efficace de la fonction d'utilité collective égalitariste min, il hérite de toutes les propriétés mathématiques et «éthiques» de cette fonction : anonymat, insensibilité à une dilatation commune croissante quelconque des utilités, juste part garantie. Il vérifie en plus la propriété de réduction des inégalités, l'indépendance vis-à-vis des agents non concernés, et enfin l'unanimité. Son fonctionnement est fondé sur une comparaison successive des utilités des agents, du plus pauvre au plus riche, jusqu'à trouver une différence qui permet alors de discriminer les deux profils : il s'agit donc d'une comparaison lexicographique sur les vecteurs d'utilité triés.

Comme nous l'avons fait remarquer au chapitre 1, le préordre leximin ne se justifie pas exclusivement dans le cadre du partage. Hors de ce domaine, il est aussi employé pour l'agrégation de niveaux de satisfaction de contraintes floues [Dubois *et al.*, 1996; Dubois et Fortemps, 1999; Dubois *et al.*, 2001], ou dans le domaine de l'optimisation multicritère, lorsque l'on cherche à obtenir des solutions relativement bien équilibrées entre les critères, mais cependant Pareto-efficaces.

Notons que l'on fait parfois usage d'un autre raffinement du min que celui apporté par le préordre leximin (et par le préordre discrimin que nous avons évoqué au chapitre 1, mais qui est peu pertinent dans le contexte du partage), l'objectif étant bien sûr toujours de bénéficier des caractéristiques égalitaires de la fonction min tout en garantissant la Pareto-efficacité de la solution trouvée. L'un de ces raffinements classiques, notamment en recherche opérationnelle et en optimisation multicritère, est la méthode max-min / max-sum. Cette méthode consiste à sélectionner l'ensemble des solutions maximisant la somme des utilités (ou des critères) parmi les solutions optimales au sens de la fonction min. Nous pouvons remarquer que si cette solution est séduisante d'un point de vue algorithmique (le calcul d'une solution max-min-optimale et d'une solution optimale au sens de la somme sont des problèmes d'optimisation classiques et que l'on sait bien résoudre), en revanche elle est problématique d'un point de vue «microéconomique», car elle peut éventuellement aboutir à perdre tous les bénéfices de la fonction min, et à se ramener à faire de l'utilitarisme pur. Considérons par exemple un problème pour lequel l'un des agents joue le rôle de fauteur de troubles, et en particulier a des préférences (quasi-)irréalisables. Dans ce cas, cet agent jouera toujours le rôle de l'agent le plus pauvre, et empêchera le fonctionnement normal de la fonction min, en empêchant son action de filtrage des profils inégalitaires, et en laissant l'espace des alternatives min-optimales égal à l'espace des alternatives admissibles en entier. Ensuite, l'agrégateur utilitariste classique somme sera le seul en jeu pour discriminer l'ensemble des profils d'utilité : ce cas se ramène donc à de l'utilitarisme pur. Le préordre leximin, en revanche, permet de résoudre ce cas pathologique en ignorant simplement l'agent fauteur de troubles, et en discriminant les profils d'utilité des autres agents de la même manière que si cet agent particulier n'était pas présent dans le partage.

Ces arguments en faveur du préordre leximin nous incitant à nous intéresser à l'algorithmique qui lui est dédiée, nous allons donc nous pencher sur le problème de recherche d'une solution non dominée au sens du préordre leximin. Il y a deux manières d'aborder cette question :

- ▷ utiliser une fonction d'utilité collective représentant le préordre leximin (ce qui est possible lorsque l'espace des alternatives est fini ou infini dénombrable), et transformer ainsi le problème en optimisation monocritère classique ;
- ▷ aborder le problème directement dans le cadre de l'optimisation multicritère et développer une algorithmique dédiée.

Nous parlerons brièvement de la première approche dans la section 5.4, mais notre point de vue sera plutôt centré sur la vision multicritère du problème.

Notons qu'il existe un certain nombre de travaux récents sur l'optimisation multicritère (ou multiobjectif). Cependant, comme le rappelle [Ehrgott et Gandibleux, 2002], le domaine de l'optimisation et celui de la décision multicritère, tous deux très prolifiques, ont été longtemps séparés. Cela est d'autant plus surprenant que de nombreux problèmes de décision multiobjectif ont naturellement besoin de procédures permettant de calculer un optimum au sens de tous les critères (par exemple un optimum de Pareto). Réciproquement de nombreux problèmes combinatoires réels étudiés dans le cadre de l'optimisation monoobjectif nécessitent la prise en compte de plusieurs critères pour modéliser une application de manière réaliste.

La plupart des travaux dans le domaine de l'optimisation multicritère sont issus de la communauté de la recherche opérationnelle [Ehrgott, 2000], et sont orientés sur des techniques traditionnelles de ce domaine, telles que la programmation linéaire . En ce qui concerne la définition de l'ensemble des bonnes solutions, la plupart de ces travaux se concentrent sur la recherche de

l'ensemble des alternatives Pareto-optimales. Plus récemment, certains travaux se sont intéressés plus spécifiquement à des solutions faisant apparaître des compromis entre les critères, telles que les solutions leximin-optimales [Ogryczak, 1997], les moyennes pondérées ordonnées [Ogryczak et Śliwiński, 2003], ou encore la norme de Tchebycheff [Galand et Perny, 2006]. Notons cependant que la plupart de ces travaux sont dédiés à des problèmes particuliers tels que le problème sac-à-dos, celui du voyageur de commerce, ou encore le problème de plus court chemin dans un graphe [Galand et Perny, 2006, 2007; Perny *et al.*, 2007].

Si la prise en compte d'objectifs multiples dans les procédures d'optimisation commence à être étudiée abondamment par les chercheurs opérationnels, en revanche, l'extension du cadre de la programmation par contraintes au multicritère n'en est qu'à ses balbutiements. On peut citer toutefois quelques exceptions, comme par exemple [Rollón et Larrosa, 2006], qui étend l'algorithme classique *Bucket Elimination* au cadre des réseaux de contraintes multiobjectif.

5.2 Le problème de satisfaction de contraintes à critère max-leximin

Notre démarche s'appuie sur le cadre des problèmes de satisfaction de contraintes et de la programmation par contraintes, afin de bénéficier d'un cadre de modélisation et de résolution intuitif et puissant, que nous pourrons exploiter pour l'adapter au problème d'optimisation leximin. Il ne s'agit donc pas de redéfinir le cadre de modélisation et les algorithmes de base de la programmation par contraintes, mais bien d'exploiter pour la résolution de notre problème l'ensemble des outils qu'il nous fournit.

Le cadre de la programmation par contraintes constitue un outil flexible et efficace pour la modélisation et la résolution de problèmes combinatoires tels que la planification, les problèmes d'allocation de ressource ou encore les problèmes de configuration. Ce paradigme est fondé sur la notion de *réseau de contraintes*, que nous avons introduit au chapitre 3, dans la définition 3.3. Rappelons qu'un réseau de contraintes est un triplet $(\mathscr{X}, \mathcal{D}, \mathscr{C})$, où $\mathscr{X} = \{\mathbf{x_1}, \ldots, \mathbf{x_p}\}$ est un ensemble de variables, $\mathcal{D}$ est la fonction de domaine, qui associe à tout $\mathbf{x_i} \in \mathscr{X}$ un domaine fini $\mathscr{D}_{\mathbf{x_i}}$, et $\mathscr{C}$ un ensemble de contraintes spécifiant chacune un ensemble d'instanciations autorisées sur un sous-ensemble des variables.

Le problème central de la programmation par contraintes est le problème de satisfaction de contraintes ([CSP] pour *Constraint Satisfaction Problem*) :

Définition 5.1 (Problème de satisfaction de contraintes ([CSP])) *Le problème de satisfaction de contraintes est le problème de décision suivant :*

***Problème 14:** [CSP]*

INSTANCE : Un réseau de contraintes $(\mathscr{X}, \mathcal{D}, \mathscr{C})$.
QUESTION : $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ a-t-il une solution (une instanciation complète cohérente) ?

Comme nous l'avons déjà fait remarquer dans le chapitre 3, il s'agit d'un problème NP-complet.

Il existe une déclinaison de ce problème de satisfaction de contraintes en problème d'optimisation, que nous appellerons problème de satisfaction de contraintes avec variable objectif, et dans lequel une variable du réseau de contraintes, qui prend ses valeurs dans les entiers, doit être maximisée.

Définition 5.2 (Problème des satisfaction de contraintes avec variable objectif) *Le problème de satisfaction de contraintes avec variable objectif est le problème d'optimisation suivant :*

Problème 15: *Problème des satisfaction de contraintes avec variable objectif*

INSTANCE : *Un réseau de contraintes* $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ *et une variable objectif* $\mathbf{o} \in \mathscr{X}$ *telle que* $\mathscr{D}_{\mathbf{o}} \subsetneq \mathbb{N}$.
SOLUTION : *«Inconsistant» si* $sol(\mathscr{X}, \mathcal{D}, \mathscr{C}) = \varnothing$.
Sinon une solution $\widehat{v} = \text{argmax}_{v \in sol(\mathscr{X}, \mathcal{D}, \mathscr{C})}(v(\mathbf{o}))$.

Étant donnés un réseau de contraintes $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ *et une variable objectif* $\mathbf{o}$*, nous noterons* $max(\mathscr{X}, \mathcal{D}, \mathscr{C}, \mathbf{o})$ *l'ensemble des solutions* $v \in sol(\mathscr{X}, \mathcal{D}, \mathscr{C})$ *telles que* $v(\mathbf{o})$ *est maximal.*

La version décisionnelle de ce problème d'optimisation, pour laquelle on cherche à déterminer s'il existe une solution dont la valeur de la variable objectif est supérieure à un certain seuil K est un problème NP-complet.

Notons que cette déclinaison du problème de satisfaction de contraintes classique en problème d'optimisation est légèrement différente du cadre des problèmes de satisfaction de contraintes valués que nous avons évoqué au chapitre 3, puisqu'ici, la valuation (qui évalue la qualité d'une solution) n'est pas portée par les contraintes elles-mêmes, mais par une variable objectif.

L'expression de problèmes de partage à fonctions d'utilité «classiques» telles que $g^\star$, $g^{(e)}$ ou encore $g^{(N)}$ (introduites dans le chapitre 1) sous forme de problème de satisfaction de contraintes avec variable objectif est relativement immédiate : n variables jouent le rôle des utilités individuelles, une variable joue le rôle de l'utilité collective, et une contrainte lie ces variables entre elles. Ainsi par exemple pour $g^\star$, cette contrainte est simplement $\mathbf{u_c} = \sum_{i=1}^{n} \mathbf{u_i}$.

En revanche, nous devons introduire un nouveau problème d'optimisation pour prendre en compte la multiplicité des variables objectif, et l'utilisation du préordre leximin sur ces variables. La définition suivante introduit la notion de [MAXLEXIMINCSP], qui est adaptée du problème de satisfaction de contraintes avec variable objectif :

Définition 5.3 ([MAXLEXIMINCSP]) *Le problème [MAXLEXIMINCSP] le problème d'optimisation suivant :*

Problème 16: *[MAXLEXIMINCSP]*

INSTANCE : *Un réseau de contraintes* $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ *et un vecteur objectif* $(\mathbf{u_1}, \ldots, \mathbf{u_n}) \in \mathscr{X}^n$ *tel que* $\forall i,\ \mathscr{D}_{\mathbf{u_i}} \subsetneq \mathbb{N}$.
SOLUTION : *«Inconsistant» si* $sol(\mathscr{X}, \mathcal{D}, \mathscr{C}) = \varnothing$.
Sinon une solution $\widehat{v} \in \text{argmax}^{\preceq_{leximin}}_{v \in sol(\mathscr{X}, \mathcal{D}, \mathscr{C})}(v(\mathbf{u_1}), \ldots, v(\mathbf{u_n}))$.

Étant donnés un réseau de contraintes $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ *et un vecteur objectif* $\overrightarrow{\mathbf{u}}$*, nous noterons* $maxleximin(\mathscr{X}, \mathcal{D}, \mathscr{C}, \overrightarrow{\mathbf{u}})$ *l'ensemble des solutions* $v \in sol(\mathscr{X}, \mathcal{D}, \mathscr{C})$ *telles que* $(v(\mathbf{u_1}), \ldots, v(\mathbf{u_n}))$ *est non dominé au sens du préordre leximin, c'est-à-dire* $maxleximin(\mathscr{X}, \mathcal{D}, \mathscr{C}, \overrightarrow{\mathbf{u}}) = \text{argmax}^{\preceq_{leximin}}_{v \in sol(\mathscr{X}, \mathcal{D}, \mathscr{C})}(v(\mathbf{u_1}), \ldots, v(\mathbf{u_n}))$.

La version décisionnelle de ce problème d'optimisation, pour laquelle on cherche à déterminer s'il existe une solution dont la valeur de la variable objectif domine au sens du leximin un certain vecteur $\overrightarrow{K}$ est clairement un problème NP-complet, et ce pour les raisons suivantes. (1) L'appartenance à NP est immédiate, car la comparaison au sens du leximin de deux vecteurs donnés peut être effectuée en temps linéaire. (2) La NP-complétude est un corollaire de la NP-complétude du problème avec variable objectif simple : le [MAXLEXIMINCSP] est équivalent à ce problème si le vecteur objectif est de taille 1.

Nous allons donc nous intéresser à la résolution du problème [MAXLEXIMINCSP]. Notre objectif n'est pas de réinventer les algorithmes de base du domaine de la programmation par contraintes,

qui ont été largement étudiés dans la littérature et fonctionnent bien dans le cadre monocritère. Notre but est plutôt d'appliquer les outils fournis par ce cadre de modélisation et de résolution à notre problème d'optimisation multicritère leximin.

Introduisons avant de poursuivre quelques notations supplémentaires. Pour une variable donnée $\mathbf{x}$, nous noterons respectivement $\underline{\mathbf{x}}$ et $\overline{\mathbf{x}}$ pour désigner $\min(\mathscr{D}_{\mathbf{x}})$ et $\max(\mathscr{D}_{\mathbf{x}})$. Dans les algorithmes, nous utiliserons aussi les raccourcis suivants pour les réductions de domaines : $\underline{\mathbf{x}} \leftarrow \alpha$ pour désigner une modification de la fonction de domaine $\mathcal{D}$ telle que $\mathscr{D}_{\mathbf{x}} \leftarrow \mathscr{D}_{\mathbf{x}} \cap [\![\alpha, +\infty[\![$ (toutes les valeurs inférieures à α sont effacées du domaine de $\mathbf{x}$ — notons que si $\alpha < \underline{\mathbf{x}}$, $\mathscr{D}_{\mathbf{x}}$ n'est pas modifié), $\overline{\mathbf{x}} \leftarrow \alpha$ pour désigner $\mathscr{D}_{\mathbf{x}} \leftarrow \mathscr{D}_{\mathbf{x}} \cap]\!]-\infty, \alpha]\!]$ (toutes les valeurs supérieures à α sont effacées du domaine de $\mathbf{x}$ — notons que si $\alpha > \overline{\mathbf{x}}$, $\mathscr{D}_{\mathbf{x}}$ n'est pas modifié), et $\mathbf{x} \leftarrow \alpha$ pour désigner $\mathscr{D}_{\mathbf{x}} \leftarrow \{\alpha\}$ (toutes les valeurs différentes de α sont effacées du domaine de $\mathbf{x}$ — notons que si $\alpha \notin \mathscr{D}_{\mathbf{x}}$, alors $\mathscr{D}_{\mathbf{x}}$ devient vide). Pour deux fonctions de domaine $\mathcal{D}_1$ et $\mathcal{D}_2$ sur deux ensembles de variables $\mathscr{X}_1$ et $\mathscr{X}_2$, nous noterons $\langle \mathcal{D}_1, \mathcal{D}_2 \rangle$ pour désigner la fonction de domaine sur $\mathscr{X} = \mathscr{X}_1 \cup \mathscr{X}_2$ telle que $\forall \mathbf{x} \in \mathscr{X}$, $\mathcal{D}(\mathbf{x}) = \mathcal{D}_1(\mathbf{x})$ si $\mathbf{x} \in \mathscr{X}_1$ et $\mathcal{D}(\mathbf{x}) = \mathcal{D}_2(\mathbf{x})$ si $\mathbf{x} \in \mathscr{X}_2$. Enfin, nous pourrons noter une fonction de domaine sous sa forme explicite : $(\mathbf{x_1} : \mathscr{D}_{\mathbf{x_1}}, \ldots, \mathbf{x_n} : \mathscr{D}_{\mathbf{x_n}})$.

5.3 Programmation par contraintes et optimisation leximin

L'objectif de cette section est d'introduire des algorithmes de résolution dédiés au problème [MAXLEXIMINCSP], et fondés sur la programmation par contraintes, qui fournit un cadre de résolution dédié aux problèmes de satisfaction de contraintes avec ou sans variable objectif. Avant de nous intéresser aux algorithmes de résolution eux-mêmes, nous allons introduire une petite description du fonctionnement et des principes de base de la programmation par contraintes.

5.3.1 La programmation par contraintes

5.3.1.1 Les deux composantes d'un système de programmation par contraintes

La programmation par contraintes est un paradigme de programmation, issu du domaine des problèmes de satisfaction de contraintes, auxquels se sont intéressés les chercheurs en intelligence artificielle dès les années 70, et né du développement du cadre de la programmation par contraintes logique (CLP pour *Constraint Logic Programming*) dans les années 80. C'est dans ces années-là qu'ont été posées les bases théoriques du cadre sous sa forme actuelle, et qu'ont été développés les premiers systèmes de résolution fondés sur ce paradigme. Les principaux développements qui ont suivi dans les années 90 ont concerné dans un premier temps l'identification de nouveaux champs d'application de la programmation par contraintes, ce qui a conduit à la mise en évidence de nouveaux types de contraintes et à l'introduction des mécanismes de propagation et de filtrage associés. Dans un deuxième temps, un certain nombre d'extensions du problème de satisfaction de contraintes, assorties de leur cadre algorithmique, ont permis de prendre en compte de nouvelles caractéristiques telles que : la notion de préférence avec les problèmes de satisfaction de contraintes valués [Schiex *et al.*, 1995] ou semi-anneaux [Bistarelli *et al.*, 1997] ou encore la distribution avec les problèmes de satisfaction de contraintes distribués [Collin *et al.*, 1992; Faltings, 2006].

Nous allons introduire très rapidement quelques principes fondateurs de la programmation par contraintes. Pour une description plus formelle et plus détaillée de la programmation par contraintes, on pourra se référer par exemple à l'ouvrage de référence [Apt, 2003], à l'article [van Hentenryck *et al.*, 1992], ou à l'introduction du manuel d'OPL Studio [van Hentenryck, 1999]. Pour une vue d'ensemble détaillée et plus générale de la programmation par contraintes, des problèmes de satisfaction

de contraintes et de leurs extensions, on pourra consulter l'ouvrage [Rossi *et al.*, 2006].

La programmation par contraintes a pour objet la résolution de problèmes de satisfaction de contraintes, à variables objectif ou non. Ce cadre est construit sur les composantes suivantes [Apt, 2003, chapitre 3] :

- ▷ *le prétraitement*, dont l'objectif est de transformer le problème initial à traiter en un problème de forme équivalente plus simple que le problème initial, et dans un format accepté par l'algorithme de recherche ;
- ▷ une procédure d'*exploration de l'arbre de recherche*, elle-même fondée sur :
  - une procédure de branchement, dont le rôle est de séparer le problème à traiter en deux sous-problèmes complémentaires (plus simples), si cela est possible, en choisissant par exemple une variable à instancier à une valeur donnée (dans le cas d'un algorithme de type retour-arrière ou *branch-and-bound* par exemple),
  - une condition permettant de tester si le problème en cours est atomique ou s'il peut encore être traité par la procédure de branchement,
  - une condition d'arrêt de la procédure d'exploration ;
- ▷ une procédure de *propagation de contraintes*, dont le rôle est de transformer le problème en cours en un problème équivalent plus simple, par déduction d'informations selon l'affectation des variables en cours et la nature des contraintes du problème.

Si l'on met à part la procédure de prétraitement, qui occupe une place relativement marginale dans la résolution d'un problème, un système de programmation par contraintes est donc fondé sur deux composantes principales : celle qui concerne l'exploration de l'arbre de recherche, et celle qui concerne la propagation de contraintes. Le processus de résolution d'un problème de satisfaction de contraintes résulte d'un dialogue permanent entre ces deux composantes fondamentales, comme l'illustre la figure 5.1.

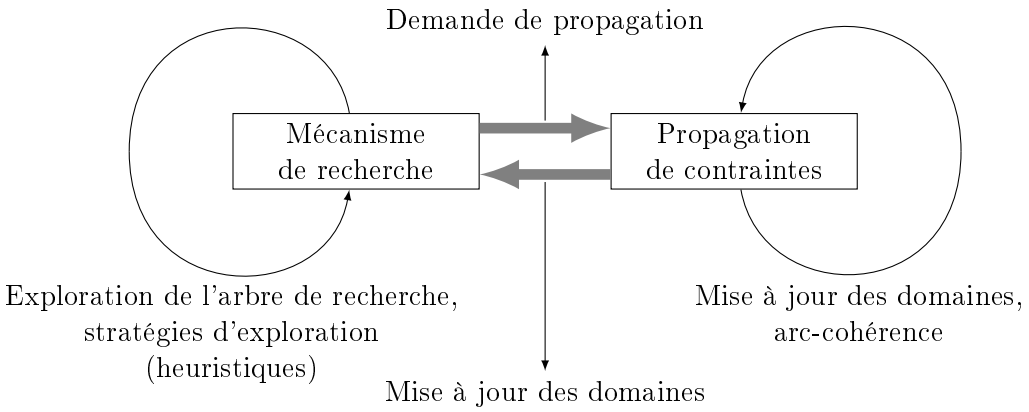


Figure 5.1 — Illustration simplifiée du principe de la programmation par contraintes.

Dans un système de programmation par contraintes, l'utilisateur a généralement assez peu d'emprise sur la composante d'exploration de l'arbre de recherche. L'algorithme de résolution (correspondant aux fonctions **solve** et **maximize** que nous allons introduire plus loin) est en général fixé par le système, et l'action de l'utilisateur se résume à spécifier un ensemble d'heuristiques permettant de guider la procédure de branchement dans son exploration de l'arbre de recherche en lui indiquant par exemple la prochaine variable à instancier, et la prochaine valeur à choisir dans son domaine.

En ce qui concerne la deuxième composante fondamentale d'un système de programmation par contraintes, celle-ci repose sur un ensemble de mécanismes de propagation d'information et de filtrage associés à un ensemble de contraintes. Cette composante comporte un langage de spécification

des contraintes permettant la description du problème à traiter, et un ensemble d'algorithmes de filtrage associés aux contraintes. Certains langages tels que Choco [Laburthe, 2000] permettent l'introduction de nouvelles contraintes dans le système, grâce à la spécification d'algorithmes de filtrage dédiés. L'objectif de ces algorithmes de filtrage est d'exploiter la sémantique des contraintes afin de tâcher d'éliminer au plus tôt possible (et de manière la plus efficace possible en terme de temps de calcul) les valeurs des domaines des variables qui ne font pas partie d'instanciations cohérentes du problème. Bien entendu, cette notion de filtrage est toujours fondée sur un compromis entre le temps de calcul nécessaire à la propagation de contraintes et la quantité d'information déduite.

Nous allons introduire de manière plus détaillée comment cette notion de filtrage a été formalisée, et comment elle a été implantée en programmation par contraintes.

5.3.1.2 Propagation de contraintes

La propagation de contraintes est l'une des techniques les mieux formalisées, les plus efficaces, et les plus abouties pour faire de l'inférence dans les réseaux de contraintes. Son objectif est de détecter et d'éliminer au plus tôt les éléments du problème qui n'ont aucune influence sur la consistance du réseau de contraintes. En d'autres termes, on cherche à transformer le réseau de contraintes initial en un réseau qui soit plus simple, et en même temps équivalent (en termes de solutions) au réseau initial, soit par suppression de contraintes qui sont toujours vérifiées (notion d'*entailement*), soit par suppression de valeurs des domaines des variables si ces valeurs ne peuvent pas faire partie d'une instanciation cohérente. Prenons un exemple. Soit un réseau de contraintes constitué de 2 variables $\mathbf{x_1}$ et $\mathbf{x_2}$ de même domaine $\{1, 2, 3\}$, et d'une contrainte $\mathbf{x_1} < \mathbf{x_2}$. À l'aide d'un raisonnement basique on peut s'apercevoir que $\mathbf{x_1}$ ne peut pas prendre la valeur 3, et que $\mathbf{x_2}$ ne peut pas prendre la valeur 1, à cause de la contrainte $\mathbf{x_1} < \mathbf{x_2}$. Nous pouvons donc supprimer ces valeurs des domaines de $\mathbf{x_1}$ et $\mathbf{x_2}$ sans restreindre l'ensemble des solutions du réseau. Tenir un tel raisonnement revient à effectuer une tâche de propagation de contraintes. On pourra trouver une description formelle de ces notions dans [Bessière, 2006].

La notion la plus classique du domaine de la propagation de contraintes est la notion d'*arc-cohérence*. Cette notion est due initialement à [Mackworth, 1977a], qui l'a définie dans un premier temps dans le cadre des contraintes binaires, puis l'a étendue aux contraintes n-aires.

Définition 5.4 (Support et arc-cohérence généralisée) *Soit* $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ *un réseau de contraintes,* $C \in \mathscr{C}$ *tel que* $\mathscr{X}(C) = \{\mathbf{x_1}, \ldots, \mathbf{x_k}\}$*, et* $\alpha \in \mathscr{D}_{\mathbf{x_i}}$*. Le* support *de* α *pour* $\mathbf{x_i}$ *et pour la contrainte* C *est l'ensemble des instanciations* $v \in \mathscr{X}(C)$ *telles que* $v(\mathbf{x_i}) = \alpha$ *et* $v \in \mathscr{R}(C)$*.*

C *est dite* arc-cohérente généralisée *(ou gac) si et seulement si* $\forall \mathbf{x} \in \mathscr{X}(C)$, $\forall \alpha \in \mathscr{D}_{\mathbf{x}}$, α *a un support non vide pour* $\mathbf{x}$ *et la contrainte* C*.*

$(\mathscr{X}, \mathcal{D}, \mathscr{C})$ *est* arc-cohérent généralisé *si et seulement si toutes ses contraintes le sont.*

Cette notion d'arc-cohérence généralisée est intuitive : une valeur d'un domaine qui n'a pas de support sur une contrainte ne peut conduire à une solution du réseau de contraintes. Le terme «généralisé» associé à l'arc-cohérence est dû au fait qu'historiquement cette notion a été introduite pour des contraintes binaires uniquement.

Nous pouvons définir, pour un réseau de contraintes donné potentiellement non arc-cohérent généralisé, un réseau de contraintes arc-cohérent équivalent :

Définition 5.5 (fonction *gac*) *Soit* $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ *un réseau de contraintes consistant et* $C \in \mathscr{C}$ *une contrainte. Nous noterons* $gac(\mathscr{X}, \mathcal{D}, \mathscr{C}, C)$ *le réseau de contraintes* $(\mathscr{X}, \mathcal{D}', \mathscr{C})$*, avec* $\mathcal{D}'$ *la fonction de domaine telle que pour tout* i $\mathcal{D}(\mathbf{x_i})'$ *est le sous-ensemble de* $\mathcal{D}(\mathbf{x_i})$ *de cardinalité maximale tel que tout* $\alpha \in \mathcal{D}(\mathbf{x_i})'$ *a un support non vide pour* C*.*

Nous noterons de même $gac(\mathscr{X},\mathcal{D},\mathscr{C})$ *le réseau de contraintes* $(\mathscr{X},\mathcal{D}',\mathscr{C})$*, avec* $\mathcal{D}'$ *la fonction de domaine telle que pour tout* i $\mathcal{D}(\mathbf{x_i})'$ *est le sous-ensemble de* $\mathcal{D}(\mathbf{x_i})$ *de cardinalité maximale tel que tout* $\alpha \in \mathcal{D}(\mathbf{x_i})'$ *a un support non vide pour toute contrainte de* $\mathscr{C}$*.*

Nous pouvons vérifier la validité de cette définition. Si le réseau de contraintes est consistant, alors chaque domaine contient au moins une valeur dont le support est non vide pour chaque contrainte ; donc il existe au moins un réseau de contraintes arc-cohérent généralisé inclus dans le réseau initial. De plus, pour une variable donnée $\mathbf{x_i}$, il ne peut y avoir deux sous-ensembles différents $\mathscr{D}'_{\mathbf{x_i}}$ et $\mathscr{D}''_{\mathbf{x_i}}$, tous deux inclus dans $\mathscr{D}_{\mathbf{x_i}}$, de cardinalité maximale, et tels que tout $\alpha \in \mathscr{D}'_{\mathbf{x_i}}$ et tout $\alpha \in \mathscr{D}''_{\mathbf{x_i}}$ ont un support non vide pour toute contrainte de $\mathscr{C}$. Donc la fonction $gac(\mathscr{X},\mathcal{D},\mathscr{C})$ définit bien un réseau de contraintes unique.

La question de la complexité du calcul de $gac(\mathscr{X},\mathcal{D},\mathscr{C})$, ou plus particulièrement de $gac(\mathscr{X},\mathcal{D},\mathscr{C},C)$ pour une contrainte donnée est fondamentale, car il s'agit d'une procédure de base de la propagation de contraintes.

La première procédure de calcul d'arc-cohérence sur un réseau de contraintes a été introduite dans [Mackworth, 1977a], restreint aux contraintes binaires (algorithme AC3), et dans [Mackworth, 1977b] pour les contraintes n-aires (algorithme GAC3). Cette procédure permet le calcul de l'arc-cohérence généralisée sur un réseau de contraintes en temps $O(er^3d^{r+1})$ et en espace $O(er)$, où e est le nombre de contraintes du réseau, r désigne la plus grande arité parmi les contraintes, et d la taille du plus grand domaine des variables. Depuis, de nombreux travaux dédiés à l'arc-cohérence (généralisée) ont conduit à l'introduction d'un certain nombre d'algorithmes plus performants : (G)AC4 [Mohr et Masini, 1998], AC6 [Bessière et Cordier, 1993], ou AC2001 [Bessière et Régin, 2001]. On pourra trouver une étude détaillée sur l'identification et la complexité de problèmes liés à l'arc-cohérence généralisée dans [Bessière *et al.*, 2007].

Il existe d'autres notions de cohérence dans les réseaux de contraintes. Certaines sont plus fortes que l'arc-cohérence généralisée, comme par exemple la chemin-cohérence [Montanari, 1974] — ou de manière plus générale la k-cohérence [Freuder, 1982] — ou la singleton-arc-cohérence. Ces propriétés permettent donc d'effectuer plus de réductions sur les domaines des variables au prix d'une complexité plus grande.

Il existe aussi des propriétés de cohérence plus faibles que l'arc-cohérence généralisée. Nous nous intéresserons plus particulièrement à l'une d'entre elles, la *borne-cohérence*, définie pour des variables ayant des domaines entiers (ou de manière plus générale totalement ordonnés) :

Définition 5.6 (Borne-cohérence) *Soit* $(\mathscr{X},\mathcal{D},\mathscr{C})$ *un réseau de contraintes et* $C \in \mathscr{C}$ *telle que* $\forall \mathbf{x} \in \mathscr{X}(C)$, $\mathscr{D}_{\mathbf{x}} \subsetneq \mathbb{N}$*.* C *est* borne-cohérente *(ou bc) si et seulement si* $\forall \mathbf{x} \in \mathscr{X}(C)$*, le support de* $\underline{\mathbf{x}}$ *et de* $\overline{\mathbf{x}}$ *sont non vides.*

Nous pouvons définir, à l'instar de l'arc-cohérence généralisée la fonction bc, qui transforme un réseau de contraintes en réseau borne-cohérent.

Définition 5.7 (fonction *bc*) *Soit* $(\mathscr{X},\mathcal{D},\mathscr{C})$ *un réseau de contraintes consistant et* $C \in \mathscr{C}$ *une contrainte. Nous noterons* $bc(\mathscr{X},\mathscr{D},\mathscr{C},C)$ *le réseau de contraintes* $(\mathscr{X},\mathscr{D}',\mathscr{C})$*, avec* $\mathcal{D}'$ *la fonction de domaine telle que pour tout* i $\mathcal{D}(x_i)'$ *est le sous-ensemble de* $\mathcal{D}(x_i)$ *de cardinalité maximale tel que* $\min(\mathscr{D}_{\mathbf{x_i}})$ *et* $\max(\mathscr{D}_{\mathbf{x_i}})$ *ont un support non vide pour* C*.*

Nous noterons de même $bc(\mathscr{X},\mathcal{D},\mathscr{C})$ *le réseau de contraintes* $(\mathscr{X},\mathcal{D}',\mathscr{C})$*, avec* $\mathcal{D}'$ *la fonction de domaine telle que pour tout* i $\mathcal{D}(x_i)'$ *est le sous-ensemble de* $\mathcal{D}(x_i)$ *de cardinalité maximale tel que tout* $\min(\mathscr{D}_{\mathbf{x_i}})$ *et* $\max(\mathscr{D}_{\mathbf{x_i}})$ *ont un support non vide pour toute contrainte de* $\mathscr{C}$*.*

Calculer la borne-cohérence sur un réseau de contraintes donné permet dans certains cas de simplifier la procédure de filtrage (au détriment bien entendu de la quantité d'information inférée), comme nous le verrons lors de l'introduction des contraintes globales.

5.3.1.3 Contraintes globales

Les travaux de ces quinze dernières années dans le domaine de la programmation par contraintes ont mis en évidence l'existence d'un certain nombre de «schémas» de contraintes spécifiques que l'on retrouve dans la modélisation d'un grand nombre de problèmes réels très différents. Les contraintes ainsi définies ont une arité variant avec le problème, mais une sémantique précise et commune à toutes les applications : de telles contraintes sont appelées *contraintes globales*. L'exemple le plus connu est certainement la contrainte globale **AllDifferent**, portant sur un ensemble de variables et interdisant l'instanciation de deux de ces variables à la même valeur. La puissance expressive et l'efficacité opérationnelle des algorithmes de propagation dédiés aux contraintes globales en ont fait un sujet de prédilection de la littérature de ces dernières années en matière de programmation par contraintes. On pourra trouver en particulier sur le sujet une réflexion sur la notion de globalité d'une contrainte dans [Bessière et van Hentenryck, 2003], centrée autour des concepts de globalité sémantique, globalité opérationnelle, et globalité algorithmique. On trouvera aussi une liste étendue de contraintes globales dans [Beldiceanu *et al.*, 2005], assortie d'un état de l'art sur le sujet dans [Beldiceanu *et al.*, 2007].

L'inconvénient des procédures de filtrage par arc-cohérence introduites ci-avant est leur complexité temporelle dépendant exponentiellement de l'arité des contraintes : il est inenvisageable d'appliquer ces procédures de filtrage sur des contraintes globales, qui peuvent avoir une arité importante. Deux stratégies sont envisageables pour contourner ce problème :

- ▷ décomposer les contraintes globales en un ensemble équivalent de contraintes d'arité inférieure (binaires si possible) ;
- ▷ développer des algorithmes de filtrage spécifiques, qui exploitent la sémantique des contraintes globales pour assurer l'arc-cohérence généralisée (ou la borne-cohérence) en temps raisonnable.

Ces deux approches sont parfaitement illustrées par les algorithmes dédiés au problème [MAXLEXIMINCSP] que nous allons introduire dans ce chapitre. Par exemple, l'algorithme 5, que nous allons présenter dans la section 5.4.3.2 fait usage d'une contrainte globale de tri d'un vecteur de variables, pour laquelle une procédure de filtrage efficace a été introduite dans [Bleuzen-Guernalec et Colmerauer, 1997] et dans [Mehlhorn et Thiel, 2000]. L'algorithme 8 que nous allons introduire dans la section 5.4.3.4 est en revanche fondé sur une décomposition de la contrainte **Sort** en contraintes **Max** et **Min** d'arité 3.

5.3.1.4 Programmation par contraintes événementielle

Pour la plupart des procédures de filtrage par arc-cohérence généralisée ou borne-cohérence, il est possible de tirer parti des propagations précédentes pour épargner du travail de vérification de support inutile. Par exemple, supposons qu'à une certaine étape, la contrainte linéaire $\mathbf{x} \geq \mathbf{y}$ soit arc-cohérente. Si la borne supérieure $\overline{\mathbf{y}}$ est réduite, alors il est inutile d'établir à nouveau l'arc-cohérence sur cette contrainte. En revanche, si c'est la borne inférieure $\underline{\mathbf{y}}$ qui est augmentée, alors certaines valeurs de $\mathbf{x}$ peuvent ne plus avoir de support, mais l'établissement de l'arc-cohérence se limite ici à supprimer les valeurs de $\mathbf{x}$ strictement inférieures à $\underline{\mathbf{y}}$.

Certains systèmes de programmation par contraintes actuels (en particulier le système Choco [Laburthe, 2000] que nous avons utilisé pour les expérimentations) traduisent cette prise en compte des propagations précédentes grâce à un système de programmation événementielle. Concrètement,

ce système fonctionne par dialogue entre la procédure d'exploration de l'arbre de recherche, qui se charge d'instancier les variables, et les procédures de propagation de contraintes. Celles-ci sont invoquées uniquement si cela est nécessaire, c'est-à-dire si le domaine d'une de leurs variables a été modifié (soit par la procédure d'exploration de l'arbre de recherche elle-même, soit par une propagation de contraintes antérieure). Ces procédures peuvent être déclenchées principalement par quatre types d'évènements :

1. augmentation de la borne inférieure d'une variable ;
2. diminution de la borne supérieure d'une variable ;
3. instanciation d'une variable (combinaison des deux premiers évènements) ;
4. effacement d'une valeur du domaine d'une variable.

Chaque fois que l'un de ces évènements survient, il est ajouté dans une file afin d'être traité dès que possible. Le traitement des évènements consiste à prévenir chaque contrainte concernée par l'évènement d'effectuer les filtrages nécessaires. Chaque contrainte peut réagir différemment à chaque type d'évènement et peut éventuellement déclencher d'autres évènements si le filtrage réduit les domaines des variables. La taille finie des domaines nous assure que cette procédure se termine (car un évènement n'est déclenché qu'en cas de réduction stricte d'un domaine, donc il ne peut y avoir un nombre infini d'évènements).

La plupart des systèmes de programmation par contraintes mettent à disposition un ensemble de contraintes classiquement utilisées, accompagnées de leurs algorithmes de filtrage associés aux évènements cités ci-dessous. Il est cependant possible dans certaines implantations de ces systèmes (comme par exemple Choco) de spécifier ses propres contraintes dotées d'algorithmes de filtrage spécifiques.

Dans la suite de ce chapitre, nous considérons que nous avons à notre disposition un système de programmation par contraintes autorisant la définition de contraintes spécifiques, et doté des deux fonctions suivantes (que nous utiliserons comme des boîtes noires) :

- ▷ **solve**$(\mathscr{X}, \mathcal{D}, \mathscr{C})$, qui renvoie une solution v du réseau de contraintes s'il y en a une, et «Inconsistant» sinon ;
- ▷ **maximize**$(\mathscr{X}, \mathcal{D}, \mathscr{C}, \mathbf{y})$, qui renvoie une solution du problème de satisfaction de contraintes avec variable objectif prenant en entrée $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ et $\mathbf{y}$.

Nous supposerons — contrairement à la plupart des solveurs de contraintes réels — que ces fonctions ne modifient pas le réseau de contraintes en entrée. Cette hypothèse est importante pour la définition des algorithmes introduits dans ce chapitre.

Équipés de ces deux fonctions, et de la possibilité d'utiliser des contraintes globales existantes et d'en définir des nouvelles, nous allons nous atteler à la résolution du problème [MAXLEXIMINCSP].

5.4 Algorithmes de programmation par contraintes

La recherche d'une solution leximin-optimale n'est pas un problème algorithmique très compliqué lorsque le réseau de contraintes en entrée n'a que très peu de solutions : puisqu'une solution leximin-optimale est aussi une solution maximin-optimale, on peut envisager de calculer toutes les solutions maximin-optimales (problème qui peut être modélisé comme un problème de satisfaction de contraintes à variable objectif unique) et de les comparer entre elles afin d'en trouver une leximin-optimale. Cette solution est suggérée par exemple dans [Ehrgott, 2000, p. 162], et, bien qu'elle puisse paraître relativement naïve, elle peut s'avérer efficace sur certaines classes de problèmes. En conséquence, ainsi que le suggère [Ogryczak, 1997], elle ne doit pas être complètement mise de côté.

Cependant, certaines instances possèdent un nombre démesuré de solutions maximin, et ainsi nécessitent une approche légèrement plus astucieuse pour envisager leur résolution en un temps raisonnable. Les aspects algorithmiques liés au calcul de solutions leximin-optimales ont été traités dans plusieurs travaux issus de plusieurs communautés différentes. Tout d'abord, les chercheurs opérationnels s'intéressent aux solutions leximin-optimales dans un contexte d'optimisation multi-critère (voir par exemple [Ehrgott, 2000]) : leur domaine d'application concerne par exemple les problèmes d'allocation équitable de ressource [Luss, 1999], les problèmes de répartition d'infrastructures [Ogryczak, 1997], ou encore les jeux matriciels [Potters et Tijs, 1992].

Si le préordre leximin est largement étudié dans la communauté de la théorie de la décision et en recherche opérationnelle, il suscite aussi l'intérêt dans le domaine des problèmes de satisfaction de contraintes flexibles, où il apparaît comme un opérateur d'agrégation de niveaux de satisfaction de contraintes floues pertinent [Dubois *et al.*, 1996; Dubois et Fortemps, 1999; Dubois *et al.*, 2001].

Si les algorithmes dédiés au calcul de solutions leximin-optimales ont donc été naturellement étudiés dans les domaines cités ci-dessus, ils n'ont en revanche jamais été traduits, à notre connaissance, dans le cadre de la programmation par contraintes. Notre première contribution sur le sujet concerne donc l'adaptation des algorithmes existants à ce cadre de modélisation et de résolution, adaptation fondée sur l'introduction :

1. d'algorithmes génériques de calcul de solutions leximin-optimales utilisant les fonctions **solve** et **maximize**, introduites ci-avant comme des «boîtes noires» fournies par les solveurs de contraintes ;

2. d'algorithmes de propagation de contraintes pour toutes les contraintes globales nécessaires utilisées dans les algorithmes introduits.

En outre, les algorithmes présentés dans les travaux évoqués ci-dessus ont souvent un champ d'application limité à des problèmes réalistes mais faciles (par exemple des problèmes continus avec des fonctions objectif linéaires, ou au moins convexes), ou peuvent rapidement devenir déraisonnables en pratique dans certains cas, comme nous allons le voir un peu plus loin dans cette section. Une exception cependant concerne le travail présenté dans [Ogryczak, 1997] et citant l'article [Maschler *et al.*, 1992], qui présente brièvement un algorithme efficace pour le calcul de solutions leximin-optimales dans le cas discret. Ce travail est à la base de l'algorithme 8 que nous présentons en section 5.4.3.4.

Notre seconde contribution sur le sujet concerne l'introduction de plusieurs algorithmes nouveaux qui s'appuient sur la puissance du cadre de la programmation par contraintes pour calculer des solutions leximin-optimales de différentes manières. La plupart de ces algorithmes sont fondés sur des mécanismes de propagation de contraintes existant dans la littérature et adaptés à notre problème.

Afin d'illustrer la manière dont fonctionnent les algorithmes, nous utiliserons l'exemple suivant qui sera décliné tout au long de la section :

Exemple 5.1 Soit $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ un réseau de contraintes, et soit $(\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}) \in \mathscr{X}^3$ un vecteur objectif. Nous supposons que l'ensemble des solutions du réseau de contraintes conduit à l'ensemble suivant de valeurs possibles pour le vecteur objectif : $(1,1,0)$, $(5,5,3)$, $(7,3,5)$, $(1,2,1)$, $(9,5,2)$, $(3,4,3)$, $(5,3,6)$ et $(10,3,4)$. Notons que cette instance possède 5 solutions maximin-optimales différentes, qui sont $(5,5,3)$, $(7,3,5)$, $(3,4,3)$, $(5,3,6)$ et $(10,3,4)$, et seulement une solution leximin-optimale, qui est $(7,3,5)$. On peut aussi remarquer que cette dernière solution est différente de la solution maximisant la somme des composantes du vecteur objectif (correspondant au point de vue utilitariste classique), qui est $(10,3,4)$.

5.4.1 Le leximin comme une fonction d'utilité collective

Intéressons-nous dans un premier temps à la résolution du problème [MAXLEXIMINCSP] par encodage du préordre leximin sous la forme d'une fonction d'utilité collective. Le but de cette approche est comme nous l'avons vu de se ramener à un problème d'optimisation monocritère, pour lequel il existe des algorithmes de résolution efficaces. On cherche donc à analyser l'efficacité pratique de la résolution du [MAXLEXIMINCSP] par l'introduction d'une variable objectif $\mathbf{u_c}$ représentant l'utilité collective, et liée au vecteur objectif initial $\overrightarrow{\mathbf{u}}$ par une contrainte représentant la fonction d'utilité collective.

Le premier problème concerne la taille du domaine de $\mathbf{u_c}$, à cause de la nature combinatoire de l'espace des alternatives. Si nous supposons que tous les $\mathscr{D}_{\mathbf{u_i}}$ sont identiques — ce qui constitue une hypothèse raisonnable, car pour que le préordre leximin soit pertinent, les composantes du vecteur objectif doivent être exprimées sur une échelle commune — et de taille m, alors on peut prouver que le nombre de classes d'équivalence pour le préordre leximin sur $\overrightarrow{\mathbf{u}}$ (correspondant à la taille minimale de $\mathscr{D}_{\mathbf{u_c}}$) est :

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

La preuve de ce résultat (voir par exemple [Knuth, 1968, exercice 1.2.6–60]) est rappelée dans l'annexe B. On montre aisément (voir la preuve dans la même annexe) que ce nombre est équivalent à m^n lorsque $m \to \infty$. Cela peut rapidement devenir un problème, lorsque m augmente, car la plupart des systèmes de programmation par contraintes ont des difficultés à prendre en compte et à traiter des domaines énormes de manière efficace.

Outre l'explosion de la taille du domaine, l'encodage du leximin par une fonction d'utilité collective pose un deuxième problème, lié à la manière de spécifier la fonction d'utilité collective par une contrainte entre $\mathbf{u_c}$ et $\overrightarrow{\mathbf{u}}$. Trois fonctions d'utilité collective représentant le préordre leximin sont connues :

- ▷ $g_1 : \overrightarrow{x} \mapsto -\sum_{i=i}^{n} n^{-x_i}$ (adapté d'une remarque dans [Frisch *et al.*, 2003]) ;
- ▷ $g_2 : \overrightarrow{x} \mapsto -\sum_{i=1}^{n} x_i^{-q}$, où $q > 0$ est assez grand [Moulin, 1988] (la détermination de l'indice q minimal tel que cette fonction représente l'ordre leximin ne semble pas une question facile à élucider — cette question est étudiée en détails dans l'annexe B).
- ▷ Une fonction moyenne pondérée ordonnée [Yager, 1988] $g_3 : \overrightarrow{x} \mapsto \sum_{i=i}^{n} w_i \cdot u_i^{\uparrow}$, où $w_1 \gg w_2 \gg \cdots \gg w_n$ (où $x \gg y$ signifie de manière informelle «x est beaucoup plus grand que y», voir aussi l'annexe B).

Dans le cas général, ni la contrainte $\mathbf{u_c} = -\sum_{i=i}^{n} n^{-\mathbf{u_i}}$, ni la contrainte $\mathbf{u_c} = -\sum_{i=1}^{n} \mathbf{u_i}^{-q}$ ne sont faciles à propager. Pour ce qui est de la contrainte $\mathbf{u_c} = \sum_{i=i}^{n} w_i \cdot u_i^{\uparrow}$ (fondée sur une moyenne pondérée ordonnée), nous pouvons remarquer que sa propagation est quasiment équivalente à la propagation d'une contrainte de tri sur le vecteur objectif. Si nous sommes en mesure de propager correctement cette contrainte, alors nous pouvons employer directement l'algorithme fondé sur la contrainte **Sort**, que nous allons introduire dans la section 5.4.3.2. L'utilisation d'une moyenne pondérée ordonnée s'avère donc inutile.

Ces remarques semblent donc dissuasives pour l'utilisation de cette méthode pour le calcul d'une solution leximin-optimale. Nous nous devons cependant de nuancer ce propos. Dans certains cas, cette approche peut s'avérer efficace — du moins en théorie. Par exemple, considérons le cas d'un problème d'allocation de ressource multiagent, pour lequel on doit attribuer un et un seul objet à chaque agent, l'allocation de l'objet j à l'agent i produisant l'utilité $u_i = w_{ij}$. Dans ce cas, l'utilité collective peut être calculée de manière linéaire, si l'on effectue une dilatation des poids w_{ij} préalable à la résolution du problème : chaque poids w_{ij} est dilaté en $-w_{ij}^{-q}$. Le calcul des utilités individuelles se fait par une simple contrainte linéaire sur les variables de décision représentant

l'allocation des objets aux agents, et le calcul de l'utilité collective par une simple somme sur les utilités des agents. Bien entendu, si le problème de propagation de la contrainte permettant le calcul de l'utilité collective est résolu dans ce cas précis, en revanche, le problème d'explosion du domaine de la variable $\mathbf{u_c}$ subsiste.

Voici un exemple illustrant la manière dont peut être traduit le préordre leximin sous la forme d'une fonction d'utilité collective :

Exemple 5.1.a Dans l'exemple 5.1 donné en début de section, la fonction d'utilité collective définie par $u_c : (u_1, u_2, u_3) \mapsto -(u_1+1)^{-9} - (u_2+1)^{-9} - (u_3+1)^{-9}$ est adéquate pour la représentation du préordre leximin. Le choix de l'exposant a été calculé de manière numérique (dichotomique) à l'aide de l'équation B.3 présentée en annexe B. Le remplacement de u_i par u_i+1 dans le calcul de l'utilité collective nous empêche d'être en dehors du domaine de définition de la fonction u_c (définie pour $u_i > 0$). Les valeurs des utilités collectives des solutions admissibles sont approximativement les suivantes : $u_c(1,1,0) = -1.00$, $u_c(5,5,3) = -4.01 \times 10^{-6}$, $u_c(7,3,5) = -3.92 \times 10^{-6}$, $u_c(1,2,1) = -3.96 \times 10^{-3}$, $u_c(9,5,2) = -5.09 \times 10^{-5}$, $u_c(3,4,3) = -8.14 \times 10^{-6}$, $u_c(5,3,6) = -3.94 \times 10^{-6}$ et $u_c(10,3,4) = -4.33 \times 10^{-6}$. Nous pouvons vérifier que le vecteur leximin-optimal $(7,3,5)$ est celui qui a l'utilité collective la plus élevée.

Notre opinion est que la résolution du problème d'optimisation leximin par l'introduction d'une fonction d'utilité collective pose non seulement les problèmes décrits ci-avant, mais en plus dissimule la véritable sémantique du leximin, et nous empêche de tirer partie de cette sémantique dans la résolution du problème d'optimisation. Les algorithmes que nous allons présenter par la suite s'appuieront donc sur une approche directe du problème, dans le cadre multicritère.

5.4.2 Une contrainte *ad-hoc* pour l'ordre leximin

Le premier algorithme que nous présentons ici est inspiré du principe du *branch-and-bound* (ou séparation-évaluation) pour la résolution de problèmes de satisfaction de contraintes valués par exemple. Le principe du *branch-and-bound* pour les problèmes de maximisation est fondé sur le maintien à chaque nœud de l'arbre de recherche de deux valeurs : une borne inférieure de la valeur objectif, qui correspond à une sous-estimation de la valeur optimale, actualisée à chaque fois qu'une solution est trouvée, et une borne supérieure de la valeur objectif, qui est une sur-estimation de la valeur objectif étant donné le nœud courant de l'arbre de recherche. À chaque nœud, la borne supérieure courante ub est comparée à la borne inférieure lb. Si $ub < lb$, alors la branche en cours d'exploration ne peut conduire à une solution optimale, et il est ainsi inutile de continuer à l'explorer (on élague la branche). La performance de l'algorithme dépend directement de notre capacité à trouver de bons majorants, et donc de notre puissance d'élagage.

Le principe du *branch-and-bound* est très facilement transposable au problème [MAXLEXIMINCSP], car le leximin définit un préordre total sur l'ensemble des tuples possibles du vecteur objectif. L'article [Fargier *et al.*, 2004b] propose une extension directe du *branch-and-bound* à n'importe quel ordre social, donc *a fortiori* au leximin. La procédure que nous présentons ici (algorithme 1) est une transcription un peu plus indirecte, faisant usage des outils mis à disposition par la programmation par contraintes.

Dans l'algorithme 1, les notions de base du *branch-and-bound* apparaissent de manière cachée. L'appel à **solve** de la ligne 5 correspond à l'exploration de l'arbre de recherche pour trouver une meilleure solution que la solution courante (nous supposons que la procédure **solve** s'arrête à la première solution trouvée). Ce qui joue le rôle de l'élagage et ainsi empêche la fonction **solve** d'explorer des branches non-optimales et de renvoyer une solution non optimale est le filtrage associé à la contrainte **Leximin** introduite à la ligne 4 dont voici la définition :

Algorithme 1 — Résolution du problème [MAXLEXIMINCSP] à la manière *branch-and-bound.*

entrée : Un réseau de contraintes $(\mathscr{X}, \mathcal{D}, \mathscr{C})$; un vecteur objectif $(\mathbf{u_1}, \ldots, \mathbf{u_n}) \in \mathscr{X}^n$.
sortie : Une solution au problème [MAXLEXIMINCSP].

1 $\widehat{v} \leftarrow$ **null** ; $v \leftarrow$ **solve**$(\mathscr{X}, \mathcal{D}, \mathscr{C})$;
2 **tant que** $v \neq$ *«Inconsistant»* **faire**
3 $\quad \widehat{v} \leftarrow v$;
4 $\quad \mathscr{C} \leftarrow \mathscr{C} \cup \{$**Leximin**$(\widehat{v}(\overrightarrow{\mathbf{u}}), \overrightarrow{\mathbf{u}})\}$;
5 $\quad v \leftarrow$ **solve**$(\mathscr{X}, \mathcal{D}, \mathscr{C})$;
6 **si** $\widehat{v} \neq$ **null alors retourner** $\widehat{v}$ **sinon retourner** *«Inconsistant»*;

Définition 5.8 (Contrainte Leximin) *Soient $\overrightarrow{\mathbf{x}}$ un vecteur de variables, $\overrightarrow{\lambda}$ un vecteur d'entiers, et v une instanciation. La contrainte* **Leximin**$(\overrightarrow{\lambda}, \overrightarrow{\mathbf{x}})$ *porte sur l'ensemble de variables de $\overrightarrow{\mathbf{x}}$, et est satisfaite par v si et seulement si $\overrightarrow{\lambda} \prec_{leximin} \overrightarrow{v(\mathbf{x})}$.*

Bien que cette contrainte n'existe pas en tant que telle dans la littérature, et donc *a fortiori* dans les systèmes de programmation par contraintes, on peut trouver dans le travail de [Frisch *et al.*, 2003] la description d'une contrainte très similaire, la contrainte **MultisetOrdering**, qui travaille sur les multi-ensembles . La sémantique de cette contrainte est la suivante : étant donnés deux multi-ensembles de variables $\mathscr{M}_1$ et $\mathscr{M}_2$, la contrainte **MultisetOrdering**$(\mathscr{M}_1, \mathscr{M}_2)$ porte sur toutes les variables de $\mathscr{M}_1$ et $\mathscr{M}_2$ et n'autorise que les valuations telles que $v(\mathscr{M}_1) \preceq_{multi} v(\mathscr{M}_2)$. L'ordre naturel $\preceq_{multi}$ sur les multi-ensembles est l'équivalent de l'ordre leximax sur les vecteurs, étendu au cas où les vecteurs peuvent être de taille différentes.

Moyennant quelques légères modifications, l'algorithme de filtrage introduit dans [Frisch *et al.*, 2003] peut être adapté à l'ordre leximin entre les vecteurs. Il assure l'arc-cohérence généralisée de cette contrainte en temps $O(n + d)$, où n est la longueur des vecteurs (ou des multi-ensembles), et d est la plus grande distance entre deux valeurs de tous les domaines des vecteurs. Dans le cas où cette valeur d est très grande, on peut bénéficier d'une variante de l'algorithme de filtrage, qui s'exécute en temps $O(n \log(n))$.

Proposition 5.1 *Si la fonction* ***solve*** *termine et est correcte, alors l'algorithme 1 termine et résout le problème [MAXLEXIMINCSP].*

Démonstration Si $sol(\mathscr{X}, \mathcal{D}, \mathscr{C}) = \varnothing$, alors le premier appel à **solve** renvoie «Inconsistant», et donc l'algorithme renvoie «Inconsistant». Dans le cas contraire, l'affectation $\widehat{v}$ est initialisée au premier passage de la boucle, et correspond à chaque itération à la dernière solution trouvée du réseau de contraintes additionné des contraintes **Leximin** des itérations précédentes. Donc l'algorithme renvoie une solution du réseau de contraintes augmenté des précédentes contraintes, donc *a fortiori* du réseau de contraintes initial. Notons v_r l'affectation retournée par l'algorithme, et supposons qu'il existe une instanciation v' solution du réseau de contraintes telle que $v'(\overrightarrow{\mathbf{u}}) \succ_{leximin} v_r(\overrightarrow{\mathbf{u}})$. v_r étant égale à l'instanciation optimale courante $\widehat{v}$ à la dernière itération, v' est donc une solution du réseau de contraintes de la ligne 5 à la dernière itération. Donc si **solve** est correcte, elle ne devrait pas retourner «Inconsistant». Or c'est le cas, car il s'agit de la dernière itération de l'algorithme. Il y a donc contradiction, ce qui prouve la proposition. ▲

5.4.3 Algorithmes itératifs

Contrairement à l'algorithme précédent, pour lequel l'optimisation était fondée directement sur le préordre leximin, les algorithmes suivants sont tous fondés sur une optimisation itérative, où, à chaque pas de l'algorithme, on essaie de maximiser la valeur d'une composante particulière de la version triée du vecteur objectif.

5.4.3.1 Brancher sur les sous-ensembles saturés

La solution algorithmique proposée dans le domaine des problèmes de satisfaction de contraintes flexibles [Dubois et Fortemps, 1999], ainsi que brièvement introduite dans [Ehrgott, 2000, page 145] est fondée sur une résolution de sous-problèmes maximin successifs. L'idée est de chercher, à chaque étape, tous les ensembles possibles de «pires» variables objectif, et de fixer explicitement leur valeur. Cette opération définit l'équivalent d'une «coupe-α forte» dans le domaine des problèmes de satisfaction de contraintes flexibles. Le terme «pires» fait référence à la notion de *sous-ensembles saturés de variables objectif* :

Définition 5.9 (Sous-ensemble saturé de variables objectif) *Soient $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ un réseau de contraintes et $\overrightarrow{\mathbf{u}}$ un vecteur de variables objectif. Soit $\widehat{m}$ la valeur maximale possible de la valeur la plus basse de $\overrightarrow{\mathbf{u}}$, ou en d'autres termes, $\widehat{m} = \max_{v \in sol(\mathscr{X},\mathcal{D},\mathscr{C})}\{\min_i\{v(\mathbf{u_i})\}\}$.*

Un sous-ensemble saturé de variables objectif *est un sous-ensemble $\mathscr{S}_{sat}$ de variables de $\overrightarrow{\mathbf{u}}$ tel qu'il existe $v \in sol(\mathscr{X}, \mathcal{D}, \mathscr{C})$ tel que $\forall \mathbf{x} \in \mathscr{S}_{sat}$, $v(\mathbf{x}) = \widehat{m}$ et $\forall \mathbf{y} \in \{\mathbf{u_i} \mid i \in [\![1, n]\!]\} \setminus \mathscr{S}_{sat}$, $v(\mathbf{y}) > \widehat{m}$.*

Clairement, les seuls sous-ensembles saturés qui peuvent conduire à une solution leximin-optimale sont ceux de cardinalité minimale. L'idée des algorithmes introduits dans [Dubois et Fortemps, 1999] pour le calcul de solutions leximin-optimales dans le contexte des problèmes de satisfaction de contraintes flexibles est fondée sur le calcul de ces sous-ensembles saturés de variables objectif de cardinalité minimale. Les algorithmes fonctionnent informellement de la manière suivante. Tout d'abord, on calcule la valeur maximin $\widehat{m}$ et les sous-ensembles saturés de variables objectif de cardinalité minimale. Ensuite, pour chaque sous-ensemble $\mathscr{S}_{sat}$ on enlève chaque variable de $\mathscr{S}_{sat}$ du vecteur objectif, et on fixe sa valeur à $\widehat{m}$. Puis on effectue la même opération pour chaque nouveau vecteur objectif obtenu, jusqu'à ce que l'on n'ait plus de variable.

Il peut y avoir, dans le cas général, plusieurs sous-ensembles saturés de cardinalité minimale à chaque pas de l'algorithme. L'algorithme peut donc être vu comme une procédure de branchement qui choisit à chaque nœud sur quel sous-ensemble saturé il va poursuivre l'exploration. La traduction dans le cadre de la programmation par contraintes de l'algorithme de recherche en profondeur d'abord introduit dans [Dubois et Fortemps, 1999] est présenté dans l'algorithme 2. Il est fondé sur la fonction `Explore`, qui est appelée de manière récursive afin d'explorer l'arbre de recherche : à chaque nœud, cette fonction calcule la valeur maximin (à l'aide de la fonction `FindMaximin`, qui effectue un appel à la fonction **maximize**), puis calcule les sous-ensembles saturés de cardinalité minimale (à l'aide de la fonction `FindSaturatedSubsets` qui effectue plusieurs appels à la fonction **solve**), puis explore successivement les sous-arbres induits par ces sous-ensembles saturés. À la toute fin de l'algorithme, une comparaison leximin est effectuée, car certaines branches de l'arbre de recherche peuvent conduire à des solutions sous-optimales.

Exemple 5.1.b L'arbre de recherche développé par l'algorithme 2 pour l'exemple 5.1 est tracé sur la figure 5.2. Sur le côté gauche de la figure, on peut voir les sous-ensembles saturés de variables objectif, et sur le côté droit les apparaissent les solutions restantes pour chaque nœud

Algorithme 2 — Résolution du problème [MAXLEXIMINCSP] en branchant sur des sous-ensembles saturés (version DFS).

entrée : Un réseau de contraintes $(\mathscr{X}, \mathcal{D}, \mathscr{C})$; un vecteur objectif $(\mathbf{u_1}, \ldots, \mathbf{u_n}) \in \mathscr{X}^n$.
sortie : Une solution au problème [MAXLEXIMINCSP].

```
1 sol ← Explore ((X, D, C), (u_1, ..., u_n));
2 si sol = ∅ alors retourner «Inconsistant»;
3 retourner LeximinOptimal(sol);          /* Comparaison leximin de toutes les solutions
  trouvées. */
```

Fonction `Explore`$((\mathscr{X}, \mathcal{D}, \mathscr{C}), (\mathbf{u_1}, \ldots, \mathbf{u_k}))$

entrée : Un réseau de contraintes $(\mathscr{X}, \mathcal{D}, \mathscr{C})$; un vecteur objectif $(\mathbf{u_1}, \ldots, \mathbf{u_k}) \in \mathscr{X}^k$.
sortie : Un ensemble de solutions leximin-optimales potentielles.

```
1 si u⃗ = ∅ alors retourner solve(X, D, C);
2 m̂ ← FindMaximin((X, D, C), (u_1, ..., u_k));
3 sat ← FindMinimalSaturatedSubsets((X, D, C), (u_1, ..., u_k), m̂);
4 sol ← ∅;
5 pour tous les S ∈ sat faire
6 |   pour tous les u_i ∈ S faire  u_i ← m̂;
7 |   pour tous les u_i ∉ S faire  u_i ← m̂ + 1;
8 |   sol ← sol ∪ Explore((X, D, C), (u_1, ..., u_k) \ S)
9 retourner sol;
```

de l'arbre de recherche. L'ensemble *sol* retourné par l'appel à `Explore` dans l'algorithme 2 est $\{(5,3,6),(7,3,5),(5,5,3)\}$.

La principale difficulté de cet algorithme est liée au calcul de sous-ensembles saturés, et, puisque dans le cas général il peut y en avoir plusieurs, de brancher sur ces sous-ensembles. Cependant, dans un certain nombre de cas connus, il existe à chaque pas de l'algorithme un sous-ensemble saturé inclus dans tous les autres, et donc un seul et unique sous-ensemble saturé de cardinalité minimale. En d'autres termes, la fonction `FindMinimalSaturatedSubsets` ne retourne qu'un seul sous-ensemble saturé à chaque pas. Dans ces cas-là, l'algorithme 2 ne produit aucun branchement, et il suffit de choisir à chaque pas l'unique sous-ensemble saturé de cardinalité minimale. Cela arrive typiquement dans les problèmes linéaires continus pour lesquels l'ensemble des alternatives est convexe [Ehrgott, 2000; Ogryczak, 1997; Luss, 1999; Potters et Tijs, 1992], ce qui explique le succès et l'efficacité de cet algorithme dans ce contexte. Un exemple d'application de l'algorithme sur un problème linéaire continu à 5 variables objectifs est illustré dans la figure 5.3.

Fonction `FindMinimalSaturatedSubsets`$((\mathscr{X}, \mathcal{D}, \mathscr{C}), (\mathbf{u_1}, \ldots, \mathbf{u_k}), \widehat{m})$

entrée : Un réseau de contraintes $(\mathscr{X}, \mathcal{D}, \mathscr{C})$; un vecteur objectif $(\mathbf{u_1}, \ldots, \mathbf{u_k}) \in \mathscr{X}^k$; un entier $\widehat{m}$.
sortie : L'ensemble de sous-ensembles saturés de cardinalité minimale du vecteur objectif pour $\widehat{m}$.

```
1 sat ← ∅ ; i ← 1;
2 tant que i ≤ k et sat = ∅ faire
3 |   pour tous les S ⊂ {u⃗} tels que |S| = i faire
4 |   |   si solve(X, D, C ∪ ⋃_{u_j ∈ S} {u_j = m̂} ∪ ⋃_{u_j ∉ S} {u_j > m̂}) ≠ «Inconsistant» alors
  |   |   sat ← sat ∪ S ;
5 |   i ← i + 1;
6 retourner sat;
```

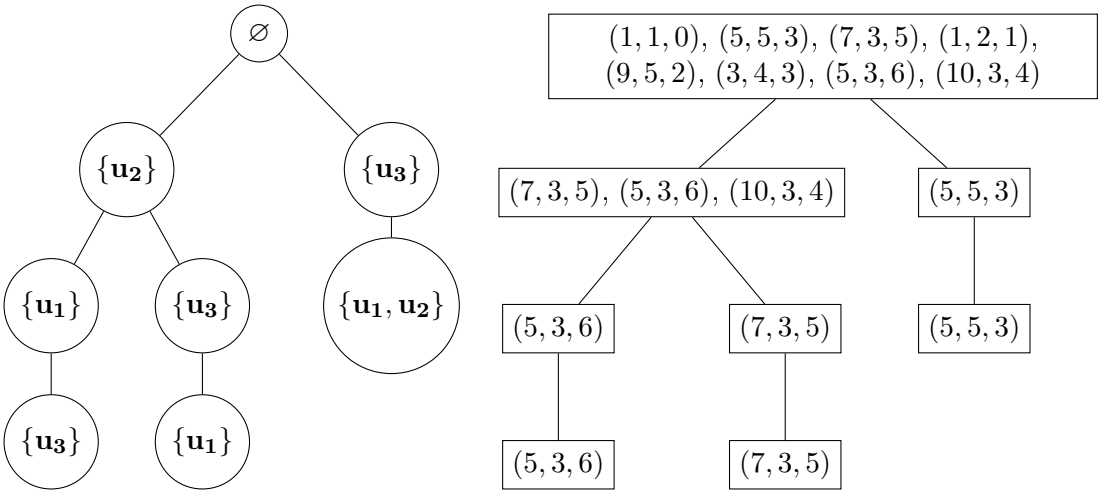
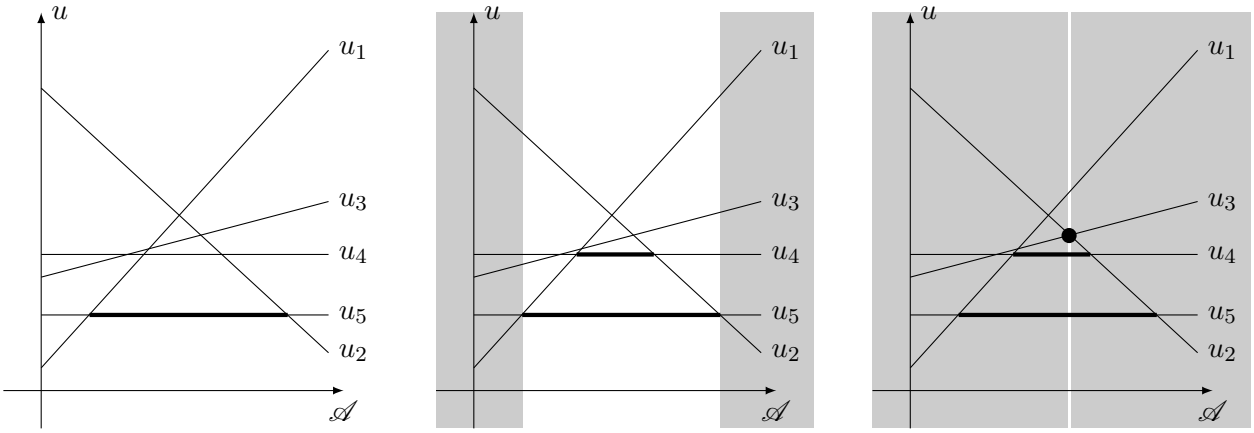


Figure 5.2 — L'arbre de recherche développé par l'algorithme 2 pour l'exemple 5.1.



L'axe des abscisses représente l'ensemble des alternatives admissibles, l'axe des ordonnées représente les valeurs des variables objectif. Les 5 droites sont les représentations graphiques des valeurs des 5 variables objectifs sur l'ensemble des alternatives.
Les traits gras représentent le lieu des solutions maximin à chaque pas. Les sous-ensembles saturés correspondant sont respectivement $\{\mathbf{u_5}\}$, $\{\mathbf{u_4}\}$ et $\{\mathbf{u_2}, \mathbf{u_3}\}$. Les zones grisées correspondent aux restrictions de domaines induites en fixant la valeur des variables des sous-ensembles saturés à la valeur maximin.

Figure 5.3 — Illustration de l'algorithme 2 sur un problème linéaire continu.

5.4.3.2 Trier pour régner

Cependant, dans un contexte de problèmes discrets tels que les [MAXLEXIMINCSP], il peut y avoir plusieurs sous-ensembles saturés de cardinalité minimale à chaque pas de l'algorithme, et donc leur calcul peut s'avérer très coûteux, ce qui au final rend ces algorithmes inutilisables en pratique. L'une des solutions pour résoudre ce problème est d'introduire de nouvelles variables pour remplacer les variables objectif en faisant en sorte que (1) l'introduction de ces variables ne modifie par l'ensemble des solutions leximin-optimales, et que (2) cela garantisse l'unicité du sous-ensemble saturé de cardinalité minimale.

Une manière intuitive de procéder est d'introduire de manière plus ou moins explicite la version triée du vecteur objectif. Les solutions leximin-optimales vis-à-vis du vecteur objectif trié sont très clairement les mêmes que les solutions leximin-optimales relatives au vecteur objectif non trié. De plus, les seuls sous-ensembles saturés sont constitués des k premières composantes du vecteur objectif trié, et donc les solutions leximin-optimales peuvent être calculées par des maximisations successives des premières composantes de ce vecteur trié.

Le cadre de la programmation par contraintes nous permet d'introduire de manière naturelle la version triée $\overrightarrow{\mathbf{y}}$ du vecteur objectif $\overrightarrow{\mathbf{u}}$ à l'aide d'une contrainte $\mathbf{Sort}(\overrightarrow{\mathbf{u}}, \overrightarrow{\mathbf{y}})$, définie comme suit :

Définition 5.10 (Contrainte Sort) *Soient $\overrightarrow{\mathbf{x}}$ et $\overrightarrow{\mathbf{x}}'$ deux vecteurs de variables de même longueur, et v une instanciation. La contrainte $\mathbf{Sort}(\overrightarrow{\mathbf{x}}, \overrightarrow{\mathbf{x}}')$ porte sur $\overrightarrow{\mathbf{x}} \cup \overrightarrow{\mathbf{x}}'$, et est satisfaite par v si et seulement si $v(\overrightarrow{\mathbf{x}}')$ est la version triée de $v(\overrightarrow{\mathbf{x}})$ dans l'ordre croissant.*

Cette contrainte a été étudiée en particulier dans deux articles, qui introduisent tous deux un algorithme de filtrage pour assurer la cohérence de borne sur cette contrainte. Le premier algorithme vient de [Bleuzen-Guernalec et Colmerauer, 1997] et s'exécute en temps $O(n \log n)$ (n étant la taille de $\overrightarrow{\mathbf{x}}$). Quelques années plus tard, [Mehlhorn et Thiel, 2000] développe un algorithme qui nécessite un temps d'exécution de $O(n)$ plus le temps requis par le tri des bornes de l'intervalle de $\overrightarrow{\mathbf{x}}$, ce qui peut s'avérer asymptotiquement plus rapide que $O(n \log n)$.

Notre méthode de calcul d'une solution leximin-optimale fondée sur la contrainte **Sort** (présentée dans l'algorithme 5) fonctionne de informellement de la manière suivante : ayant introduit la version triée $\overrightarrow{\mathbf{y}}$ du vecteur objectif $\overrightarrow{\mathbf{u}}$, elle maximise successivement les composantes de ce vecteur, sachant que la solution leximin-optimale est la solution qui maximise $\mathbf{y_1}$, et, étant donnée cette valeur maximale, maximise $\mathbf{y_2}$, *et caetera* jusqu'à $\mathbf{y_n}$.

Exemple 5.1.c Revenons sur notre exemple. Au début de l'algorithme, 3 nouvelles variables $(\mathbf{y_1}, \mathbf{y_2}, \mathbf{y_3})$ sont introduites, afin de représenter la version triée du vecteur objectif. Les instanciations admissibles pour $(\overrightarrow{\mathbf{u}}, \overrightarrow{\mathbf{y}})$ sont : $((1,1,0),(0,1,1))$, $((5,5,3),(3,5,5))$, $((7,3,5),(3,5,7))$, $((1,2,1),(1,1,2))$, $((9,5,2),(2,5,9))$, $((3,4,3),(3,3,4))$, $((5,3,6),(3,5,6))$ et $((10,3,4),(3,4,10))$.

- ▷ Pendant le premier pas de l'algorithme, la variable $\mathbf{y_1}$ est maximisée (sa valeur maximale est 3) et ensuite est fixée à sa valeur optimale 3. Les instanciations admissibles restantes sont donc : $((5,5,3),(\mathbf{3},5,5))$, $((7,3,5),(\mathbf{3},5,7))$, $((3,4,3),(\mathbf{3},3,4))$, $((5,3,6),(\mathbf{3},5,6))$ et $((10,3,4),(\mathbf{3},4,10))$.
- ▷ Pendant le second pas de l'algorithme, la variable $\mathbf{y_2}$ est maximisée (sa valeur maximale est 5), et ensuite est fixée à sa valeur optimale 5. Les instanciations admissibles restantes sont donc : $((5,5,3),(3,\mathbf{5},5))$, $((7,3,5),(3,\mathbf{5},7))$ et $((5,3,6),(3,\mathbf{5},6))$.
- ▷ Pendant le troisième et dernier pas de l'algorithme, $\mathbf{y_3}$ est maximisée (sa valeur maximale est 7) L'unique solution leximin-optimale est : $((7,3,5),(3,5,\mathbf{7}))$.

Proposition 5.2 *Si les deux fonctions* **maximize** *et* **solve** *sont correctes et terminent, alors l'algorithme 5 termine et renvoie une solution au problème [MAXLEXIMINCSP].*

Algorithme 5 — Résolution du problème [MAXLEXIMINCSP] en utilisant une contrainte de tri (Trier pour régner).

entrée : Un réseau de contraintes $(\mathscr{X}, \mathcal{D}, \mathscr{C})$; un vecteur objectif $(\mathbf{u_1}, \ldots, \mathbf{u_n}) \in \mathscr{X}^n$.
sortie : Une solution au problème [MAXLEXIMINCSP].

1 **si solve**$(\mathscr{X}, \mathscr{D}, \mathscr{C})$ =«*Inconsistant*» **retourner** «*Inconsistant*»;
2 $\mathscr{X}' \leftarrow \mathscr{X} \cup \{\mathbf{y_1}, \ldots, \mathbf{y_n}\}$;
3 $\mathcal{D}' \leftarrow \langle \mathscr{D}, (\mathbf{y_1} : \mathscr{D}_{\mathbf{y_1}}, \ldots, \mathbf{y_n} : \mathscr{D}_{\mathbf{y_n}}) \rangle$ avec $\mathscr{D}_{\mathbf{y_i}} = [\![\min_j(\underline{\mathbf{u_j}}), \max_j(\overline{\mathbf{u_j}})]\!]$;
4 $\mathscr{C}' \leftarrow \mathscr{C} \cup \{\mathbf{Sort}(\overrightarrow{\mathbf{u}}, \overrightarrow{\mathbf{y}})\}$;
5 **pour** $i \leftarrow 1$ **à** n **faire**
6 $\quad \widehat{v}_{(i)} \leftarrow \mathbf{maximize}(\mathscr{X}', \mathscr{D}', \mathscr{C}', \mathbf{y_i})$;
7 $\quad \mathbf{y_i} \leftarrow \widehat{v}_{(i)}(\mathbf{y_i})$;
8 **retourner** $\widehat{v}_{(n)\downarrow\mathscr{X}}$;

Démonstration Si $sol(\mathscr{X}, \mathcal{D}, \mathscr{C}) = \varnothing$ et si la fonction **solve** est correcte, alors l'algorithme 5 retourne «Inconsistant» de manière évidente. Nous supposerons dans la suite de la preuve que nous ne sommes pas dans ce cas-là, et nous emploierons les notations suivantes : $\mathscr{S}_i$ et $\mathscr{S}'_i$ seront respectivement les ensembles de solutions du réseau de contrainte $(\mathscr{X}', \mathcal{D}', \mathscr{C}')$ au début et à la fin de l'itération i.

Nous avons de manière évidente $\mathscr{S}_{i+1} = \mathscr{S}'_i$ pour tout $i \in [\![1, n-1]\!]$, ce qui prouve que si $\mathscr{S}_i \neq \varnothing$, alors l'appel à **maximize** à la ligne 6 ne renvoie pas «Inconsistant», et $\mathscr{S}_{i+1} \neq \varnothing$. Ainsi, $\widehat{v}_{(n)}$ est bien défini et $(\widehat{v}_{(n)})_{\downarrow\mathscr{X}}$ est clairement une solution de $(\mathscr{X}, \mathcal{D}, \mathscr{C})$.

Nous notons $\widehat{v} = \widehat{v}_{(n)}$ l'instanciation calculée par le dernier appel à **maximize** dans l'algorithme 5. Supposons qu'il y ait une instanciation $v \in sol(\mathscr{X}, \mathcal{D}, \mathscr{C})$ telle que $\widehat{v}(\overrightarrow{\mathbf{u}}) \prec_{leximin} v(\overrightarrow{\mathbf{u}})$. Nous définissons alors v^+ comme étant l'extension de v qui instancie chaque $\mathbf{y_i}$ à $v(\overrightarrow{\mathbf{u}})^{\uparrow}_i$. En raison de la contrainte **Sort**, $\widehat{v}(\overrightarrow{\mathbf{y}})$ et $v^+(\overrightarrow{\mathbf{y}})$ sont les versions triées respectives de $\widehat{v}(\overrightarrow{\mathbf{u}})$ et $v^+(\overrightarrow{\mathbf{u}})$. D'après la définition 1.32 du préordre leximin (au changement d'indice près), il existe un indice $i \in [\![0, n-1]\!]$ tel que $\forall j \in [\![1, i]\!]$, $\widehat{v}(\mathbf{y_j}) = v^+(\mathbf{y_j})$ et $\widehat{v}(\mathbf{y_{i+1}}) < v^+(\mathbf{y_{i+1}})$. À cause de la ligne 7, on a $\widehat{v}(\mathbf{y_{i+1}}) = \widehat{v}_{(n)}(\mathbf{y_{i+1}}) = \widehat{v}_{(i+1)}(\mathbf{y_{i+1}})$. Donc v^+ est une solution appartenant à l'ensemble $max(\mathscr{X}', \mathscr{D}', \mathscr{C}', \mathbf{y_{i+1}})$ dont la valeur de la variable objectif $v^+_{(i+1)}(\mathbf{y_{i+1}})$ est strictement plus grande que $\widehat{v}_{(i+1)}(\mathbf{y_{i+1}})$, ce qui contredit l'hypothèse de correction de **maximize**. Il n'existe donc pas de solution v de $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ telle que $\widehat{v}(\overrightarrow{\mathbf{u}}) \prec_{leximin} v(\overrightarrow{\mathbf{u}})$: donc $\widehat{v}(\overrightarrow{\mathbf{u}})$ est une solution leximin-optimale. ▲

5.4.3.3 Un nouvel algorithme utilisant une méta-contrainte de cardinalité

Avant de présenter ce nouvel algorithme, nous introduisons la notation suivante : étant donnés un vecteur de nombres entiers $\overrightarrow{x}$ et un entier α, $\sum_i(\alpha \leq x_i)$ désignera la cardinalité de l'ensemble $\{i \mid \alpha \leq x_i\}$. Cette notation est inspirée par la notion de réification dans un langage de programmation par contraintes tel qu'OPL [van Hentenryck, 1999], où $(\alpha \leq x_i)$ représente une valeur booléenne valant 1 si l'inégalité est satisfaite et 0 sinon.

L'algorithme précédent introduisait de manière explicite la version triée du vecteur objectif, nécessitant donc une propagation de la contrainte **Sort** portant sur l'intégralité de ce vecteur trié. Il est cependant possible, grâce à une petite astuce, d'accéder directement à la $i^{\text{ème}}$ composante du vecteur objectif trié sans recourir de manière explicite à l'intégralité du vecteur trié. Le nouvel algorithme que nous allons présenter exploite cette remarque, qui s'appuie sur la proposition (évidente) suivante :

Proposition 5.3 *Soit $\overrightarrow{x}$ un vecteur de nombres de taille n. Nous avons :*

$$x_i^{\uparrow} = \max\{\alpha \mid \sum_i (\alpha \leq x_i) \geq n - i + 1\}$$

En d'autres termes, le $i^{\text{ème}}$ minimum d'un vecteur de nombres de taille n est le nombre maximal α tel qu'au moins $n - i + 1$ composantes du vecteur sont supérieures ou égales à α.

L'algorithme fondé sur la méta-contrainte AtLeast Cette nouvelle approche du problème, présentée dans l'algorithme 6, fonctionne de manière relativement similaire à l'algorithme précédent, mais comme nous l'avons indiqué, il ne nécessite pas l'introduction de la version triée du vecteur objectif dans son intégralité. De manière informelle, il fonctionne comme suit :

- ▷ il calcule dans un premier temps la valeur maximale $\widehat{y_1}$ de $\mathbf{y_1}$ telle qu'il existe une solution v avec $\sum_i (\widehat{y_1} \leq v(\mathbf{u_i})) = n$ (ou en d'autres termes $\forall i,\ \widehat{y_1} \leq v(\mathbf{u_i})$) ;
- ▷ ensuite il fixe $\mathbf{y_1}$ à $\widehat{y_1}$ et calcule la valeur maximale $\widehat{y_2}$ de $\mathbf{y_2}$ telle qu'il existe une solution v avec $\sum_i (\widehat{y_2} \leq v(\mathbf{u_i})) \geq n - 1$;
- ▷ *et caetera* jusqu'à ce que, ayant fixé $\mathbf{y_{n-1}}$ à $\widehat{y_{n-1}}$, il calcule la valeur maximale $\widehat{y_n}$ de $\mathbf{y_n}$ telle qu'il existe une solution v avec $\sum_i (\widehat{y_n} \leq v(\mathbf{u_i})) \geq 1$.

Afin d'assurer la contrainte sur les $\mathbf{u_i}$, nous utilisons la méta-contrainte **AtLeast**, dérivée d'un *combinateur de cardinalité* introduit dans [van Hentenryck *et al.*, 1992], et existant dans la plupart des systèmes de programmation par contraintes :

Définition 5.11 (Méta-contrainte AtLeast) *Soient Γ un ensemble de p contraintes, et $k \in [\![1, p]\!]$ un entier. La méta-contrainte* **AtLeast**(Γ, k) *porte sur l'union des scopes de Γ et est satisfaite par v si et seulement si au moins k contraintes de Γ sont satisfaites v.*

Cette approche, présentée dans l'algorithme 6, est illustrée dans l'exemple suivant :

Exemple 5.1.d Nous appliquons l'algorithme 6 sur l'exemple 5.1 :

- ▷ Lors du premier pas de l'algorithme, une variable $\mathbf{y_1}$ est introduite, et l'on impose à toutes les variables objectif de prendre une valeur supérieure à celle de $\mathbf{y_1}$. Cela donne les solutions suivantes pour $(\overrightarrow{\mathbf{u}}, \mathbf{y_1})$: $((1,1,0),0)$, $((5,5,3),[\![0,\mathbf{3}]\!])$, $((7,3,5),[\![0,\mathbf{3}]\!])$, $((1,2,1),[\![0,1]\!])$, $((9,5,2),[\![0,2]\!])$, $((3,4,3),[\![0,\mathbf{3}]\!])$, $((5,3,6),[\![0,\mathbf{3}]\!])$ et $((10,3,4),[\![0,\mathbf{3}]\!])$. $\mathbf{y_1}$ est fixée à sa valeur maximale 3 (apparaissant en gras ci-avant), ce qui a pour effet de restreindre l'ensemble des instanciations admissibles à $\{((5,5,3),3),((7,3,5),3),((3,4,3),3),((5,3,6),3),((10,3,4),3)\}$.
- ▷ Lors du deuxième pas de l'algorithme, une variable $\mathbf{y_2}$ est introduite, et l'on impose qu'il y ait au moins deux variables objectif qui aient une valeur supérieure à celle de $\mathbf{y_2}$. Cela donne les solutions suivantes pour $(\overrightarrow{\mathbf{u}}, \mathbf{y_2})$: $((5,5,3),[\![0,\mathbf{5}]\!])$, $((7,3,5),[\![0,\mathbf{5}]\!])$, $((3,4,3),[\![0,4]\!])$, $((5,3,6),[\![0,\mathbf{5}]\!])$ et $((10,3,4),[\![0,4]\!])$. $\mathbf{y_2}$ est fixée à sa valeur maximale 5 (apparaissant en gras ci-avant), ce qui a pour effet de restreindre l'ensemble des instanciations admissibles à $\{((5,5,3),5),((7,3,5),5),((5,3,6),5)\}$.
- ▷ Lors du troisième pas de l'algorithme, une variable $\mathbf{y_3}$ est introduite, et l'on impose qu'il y ait au moins une variable objectif qui ait une valeur supérieure à celle de $\mathbf{y_3}$. Cela donne les solutions suivantes pour $(\overrightarrow{\mathbf{u}}, \mathbf{y_3})$: $((5,5,3),[\![0,5]\!])$, $((7,3,5),[\![0,\mathbf{7}]\!])$ et $((5,3,6),[\![0,6]\!])$. La valeur maximale de $\mathbf{y_3}$ est 7 (écrite en gras ci-avant), ce qui conduit à l'unique solution leximin-optimale $(7,3,5)$.

Proposition 5.4 *Si les fonctions* **maximize** *et* **solve** *sont toutes deux correctes et terminent, alors l'algorithme 6 termine et retourne une solution du problème [MAXLEXIMINCSP].*

Algorithme 6 — Résolution du problème [MAXLEXIMINCSP] en utilisant une méta-contrainte de cardinalité.

entrée : Un réseau de contraintes $(\mathscr{X}, \mathcal{D}, \mathscr{C})$; un vecteur objectif $(\mathbf{u_1}, \ldots, \mathbf{u_n}) \in \mathscr{X}^n$.
sortie : Une solution au problème [MAXLEXIMINCSP].

1 **si solve**$(\mathscr{X}, \mathcal{D}, \mathscr{C}) =$ *«Inconsistant»* **retourner** *«Inconsistant»*;
2 $(\mathscr{X}_0, \mathcal{D}'_0, \mathscr{C}_0) \leftarrow (\mathscr{X}, \mathcal{D}, \mathscr{C})$;
3 **pour** $i \leftarrow 1$ **à** n **faire**
4 $\quad \mathscr{X}_i \leftarrow \mathscr{X}_{i-1} \cup \{\mathbf{y_i}\}$;
5 $\quad \mathcal{D}_i \leftarrow \langle \mathcal{D}'_{i-1}, (\mathbf{y_i} : \mathscr{D}_{\mathbf{y_i}}) \rangle$ avec $\mathscr{D}_{\mathbf{y_i}} = [\![\min_j(\underline{\mathbf{u_j}}), \max_j(\overline{\mathbf{u_j}})]\!]$;
6 $\quad \mathscr{C}_i \leftarrow \mathscr{C}_{i-1} \cup \{\mathbf{AtLeast}(\{\mathbf{y_i} \leq \mathbf{u_1}, \ldots, \mathbf{y_i} \leq \mathbf{u_n}\}, n - i + 1)\}$;
7 $\quad \widehat{v}_{(i)} \leftarrow \mathbf{maximize}(\mathscr{X}_i, \mathcal{D}_i, \mathscr{C}_i, \mathbf{y_i})$;
8 $\quad \mathcal{D}'_i \leftarrow \mathcal{D}_i$ avec $\mathbf{y_i} \leftarrow \widehat{v}_{(i)}(\mathbf{y_i})$;
9 **retourner** $\widehat{v}_{(n)\downarrow\mathscr{X}}$;

Dans les preuves suivantes, nous écrirons sol_i et sol'_i pour désigner respectivement $sol(\mathscr{X}_i, \mathcal{D}_i, \mathscr{C}_i)$ et $sol(\mathscr{X}_i, \mathcal{D}'_i, \mathscr{C}_i)$. Nous noterons de même $(sol_i)_{\downarrow\mathscr{X}_j}$ et $(sol'_i)_{\downarrow\mathscr{X}_j}$ pour désigner les mêmes ensembles de solutions projetés sur $\mathscr{X}_j$ (avec $j < i$). Nous pouvons remarquer dès à présent que $sol_0 = sol(\mathscr{X}, \mathcal{D}, \mathscr{C})$, et que $\forall i,\ sol'_i \subseteq sol_i$ (à cause de la ligne 8 qui restreint le domaine de $\mathbf{y_i}$).

Lemme 18 *Si $sol_0 \neq \varnothing$, alors $\widehat{v}_{(n)}$ est bien défini et n'est pas égal à «Inconsistant».*

Démonstration Soit $i \in [\![1, n]\!]$. Supposons que $sol'_{i-1} \neq \varnothing$, et soit $v_{(i)} \in sol'_{i-1}$. Alors l'extension de $v_{(i)}$ qui instancie $\mathbf{y_i}$ à $\min_j(\mathbf{u_j})$ est une solution de $(\mathscr{X}_i, \mathcal{D}_i, \mathscr{C}_i)$ (puisqu'une seule contrainte a été ajoutée entre $\mathscr{C}_{i-1}$ et $\mathscr{C}_i$ et qu'elle est satisfaite de manière évidente par cette dernière instanciation). En conséquence, $sol_i \neq \varnothing$, et, si **maximize** est correcte, $\widehat{v}_{(i)} \neq$ «Inconsistant» et $\widehat{v}_{(i)} \in sol'_i$. Ainsi, $sol'_i \neq \varnothing$, ce qui prouve le lemme 18 par récurrence. ▲

Lemme 19 *Si $sol_0 \neq \varnothing$, alors $(\widehat{v}_{(n)})_{\downarrow\mathscr{X}_i} \in sol_i, \forall i \in [\![0, n]\!]$.*

Démonstration On a $sol'_i \subseteq sol_i$, et $(sol_{i+1})_{\downarrow\mathscr{X}_i} \subseteq sol'_i$ (puisque de $(\mathscr{X}_i, \mathscr{D}'_i, \mathscr{C}_i)$ à $(\mathscr{X}_{i+1}, \mathcal{D}_{i+1}, \mathscr{C}_{i+1})$ on a simplement ajouté une contrainte). Plus généralement, on a $(sol'_i)_{\downarrow\mathscr{X}_j} \subseteq (sol_i)_{\downarrow\mathscr{X}_j}$, et $(sol_{i+1})_{\downarrow\mathscr{X}_j} \subseteq (sol'_i)_{\downarrow\mathscr{X}_j}$, pour peu que $j \leq i$. Ainsi, $(\widehat{v}_{(n)})_{\downarrow\mathscr{X}_i} \in (sol'_n)_{\downarrow\mathscr{X}_i} \subseteq (sol_n)_{\downarrow\mathscr{X}_i} \subseteq \cdots \subseteq (sol_{i+1})_{\downarrow\mathscr{X}_i} \subseteq sol'_i \subseteq sol_i$. ▲

Lemme 20 *Si $sol_0 \neq \varnothing$, $\widehat{v}_{(n)}(\overrightarrow{\mathbf{y}})$ est égal à $\widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})^{\uparrow}$.*

Démonstration Pour tout $i \in [\![1, n]\!]$, $(\widehat{v}_{(n)})_{\downarrow\mathscr{X}_i}$ est une solution de sol_i d'après le lemme 19. D'après la proposition 5.3, l'instanciation $(\widehat{v}_{(n)})_{\downarrow\mathscr{X}_i}$ dans laquelle on a remplacé la valeur de $\mathbf{y_i}$ par $\widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})^{\uparrow}_i]$ satisfait la contrainte de cardinalité à l'itération i, et est donc une solution de sol_i. Par définition de la fonction **maximize**, on a donc $\widehat{v}_{(i)}(\mathbf{y_i}) \geq \widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})^{\uparrow}_i$. Puisque $\widehat{v}_{(i)}(\mathbf{y_i}) = \widehat{v}_{(n)}(\mathbf{y_i})$, on a $\widehat{v}_{(n)}(\mathbf{y_i}) \geq \widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})^{\uparrow}_i$.

Puisque $\widehat{v}_{(n)}$ est une solution de sol_n, au moins $n-i+1$ nombres parmi ceux du vecteur $\widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})$ sont supérieurs ou égaux à $\widehat{v}_{(n)}(\mathbf{y_i})$. Donc, les $n-i+1$ composantes les plus grandes de $\widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})$ au moins doivent être supérieures ou égales à $\widehat{v}_{(n)}(\mathbf{y_i})$. Ces composantes incluent $\widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})^{\uparrow}_i$, ce qui montre que $\widehat{v}_{(n)}(\mathbf{y_i}) \leq \widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})^{\uparrow}_i$, ce qui prouve enfin le lemme. ▲

Nous pouvons maintenant rassembler tous les résultats précédents et prouver la proposition 5.4.

Démonstration (proposition 5.4) Si $sol(\mathscr{X}, \mathcal{D}, \mathscr{C}) = \varnothing$, et si **solve** est correcte, alors l'algorithme 6 retourne de manière évidente «Inconsistant». Sinon, d'après le lemme 18,

cet algorithme retourne une instanciation $(\widehat{v}_{(n)})_{\downarrow\mathscr{X}}$ qui est, d'après le lemme 19, une solution de $(\mathscr{X}_0, \mathcal{D}_0, \mathscr{C}_0) = (\mathscr{X}, \mathcal{D}, \mathscr{C})$.

Supposons qu'il existe une instanciation $v \in sol(\mathscr{X}, \mathcal{D}, \mathscr{C})$ telle que $\widehat{v}_{(n)}(\overrightarrow{\mathbf{u}}) \prec_{leximin} v(\overrightarrow{\mathbf{u}})$. Alors, d'après la définition 1.32 du préordre leximin (au changement d'indice près), $\exists i \in [\![1, n]\!]$ tel que $\forall j < i$, $v(\overrightarrow{\mathbf{u}})^{\uparrow}_j = \widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})^{\uparrow}_j$ et $\widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})^{\uparrow}_i < v(\overrightarrow{\mathbf{u}})^{\uparrow}_i$. Soit $v^+_{(i)}$ l'extension de v qui instancie $\mathbf{y_1}, \ldots \mathbf{y_{i-1}}$ aux valeurs respectives $\widehat{v}_{(n)}(\mathbf{y_1}), \ldots \widehat{v}_{(n)}(\mathbf{y_{i-1}})$ et $\mathbf{y_i}$ à $v(\overrightarrow{\mathbf{u}})^{\uparrow}_i$. D'après le lemme 20, $\forall j$, $\widehat{v}_{(n)}(\mathbf{y_j}) = \widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})^{\uparrow}_j$. En réunissant toutes les égalités précédentes, nous obtenons $\forall j < i$ $v^+_{(i)}(\mathbf{y_j}) = \widehat{v}_{(n)}(\mathbf{y_j}) = v(\overrightarrow{\mathbf{u}})^{\uparrow}_j = (v^+_{(i)}(\overrightarrow{\mathbf{u}}))^{\uparrow}_j$. On a aussi $v^+_{(i)}(\mathbf{y_i}) = v(\overrightarrow{\mathbf{u}})^{\uparrow}_i = (v^+_{(i)}(\overrightarrow{\mathbf{u}}))^{\uparrow}_i$. D'après la proposition 5.3, $\forall j \leq i$ au moins $n - j + 1$ composantes de $(v^+_{(i)}(\overrightarrow{\mathbf{u}}))$ sont supérieures ou égales à $v^+_{(i)}(\mathbf{y_j})$, ce qui prouve que $v^+_{(i)}$ satisfait toutes les contraintes de cardinalité à l'itération i. Puisque cette instanciation satisfait aussi toutes les contraintes de $\mathscr{C}$ et instancie chaque variable de $\mathscr{X}_i$ à l'une de ses valeurs possibles, il s'agit d'une solution de sol_i, et $v^+_{(i)}(\mathbf{y_i}) = v(\overrightarrow{\mathbf{u}})^{\uparrow}_i > \widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})^{\uparrow}_i = \widehat{v}_{(i)}(\mathbf{y_i})$. Cette dernière inégalité contredit la définition de **maximize**, ce qui montre que $\widehat{v}_{(n)}(\overrightarrow{\mathbf{u}})$ est une solution leximin-optimale, prouvant ainsi la proposition 5.4. ▲

Cette transcription de la notion de tri des variables objectif par le biais de la contrainte de cardinalité n'est pas complètement nouvelle. Plus précisément, et de manière intéressante, cette approche est à la base du développement des algorithmes de filtrage de [Bleuzen-Guernalec et Colmerauer, 1997] et de [Mehlhorn et Thiel, 2000] pour la contrainte **Sort**, comme on nous l'a fait remarquer [1]. Les deux algorithmes présentés dans ces papiers s'appuient sur les travaux [Zhou, 1997], qui introduit un mécanisme de propagation de la contrainte de tri fondé sur la même idée que les contraintes **AtLeast**. Cette approche a cependant un intérêt particulier dans le cadre de l'optimisation leximin, car elle permet d'introduire les variables du vecteur trié au fur et à mesure de l'algorithme.

Propagation de la contrainte AtLeast À cause de sa généricité, la méta-contrainte **AtLeast** ne peut pas fournir de procédures de filtrage très efficaces. Heureusement, dans notre cas pour lequel toutes les contraintes de Γ sont de la forme $\mathbf{y} \leq \mathbf{x_i}$, la cohérence de borne peut être assurée à l'aide d'une procédure simple, présentée dans l'algorithme 7 (nous rappelons que la notation $\overline{\mathbf{x}} \leftarrow \alpha$ signifie que toutes les valeurs supérieures à α sont enlevées de $\mathscr{D}_{\mathbf{x}}$).

Algorithme 7 — Calcul de la cohérence de borne sur la méta-contrainte **AtLeast** portant sur des contraintes linéaires.

entrée : Un réseau de contraintes $(\mathscr{X}, \mathcal{D}, \mathscr{C})$, vecteur de variables $(\mathbf{x_1}, \ldots, \mathbf{x_n}) \in \mathscr{X}^n$, une variable $\mathbf{y} \in \mathscr{X}$, un entier $k \leq n$.
sortie : $bc(\mathbf{AtLeast}(\{\mathbf{y} \leq \mathbf{x_1}, \ldots, \mathbf{y} \leq \mathbf{x_n}\}, k))$, ou «Inconsistant».

1 **si** $\sum_i (\underline{\mathbf{y}} \leq \overline{\mathbf{x_i}}) < k$ **retourner** *«Inconsistant»*;
2 **if** $\sum_i (\underline{\mathbf{y}} \leq \overline{\mathbf{x_i}}) = k$ **then**
3 ⌊ **pour tous les** j **tels que** $\underline{\mathbf{y}} \leq \overline{\mathbf{x_j}}$ **faire** $\underline{\mathbf{x_j}} \leftarrow \underline{\mathbf{y}}$;
4 $\overline{\mathbf{y}} \leftarrow \overrightarrow{\overline{\mathbf{x}}}^{\downarrow}_k$; /* où $\overrightarrow{\overline{\mathbf{x}}} = (\overline{\mathbf{x_1}}, \ldots, \overline{\mathbf{x_n}})$ */
5 **retourner** $(\mathscr{X}, \mathcal{D}, \mathscr{C})$;

De manière informelle, cet algorithme fonctionne comme suit. Si les domaines des variables sont tels que la contrainte ne peut plus être satisfaite (ligne 1), la procédure renvoie «Inconsistant». Sinon, si seulement k variables parmi les variables $\mathbf{x_i}$ peuvent encore être supérieures à $\mathbf{y}$, alors ces

[1] Cette remarque nous a été signalée par Diego Olivier Fernandez Pons au cours d'une discussion.

variables ne peuvent être inférieures à $\mathbf{y}$, sinon la contrainte est violée (ligne 3). Dans tous les cas, la valeur de $\mathbf{y}$ ne peut être plus grande que la $k^{\text{ème}}$ plus grande valeur des variables $\mathbf{x_i}$.

Cet algorithme s'exécute en temps $O(n)$, puisque la sélection de la $k^{\text{ème}}$ plus grande valeur de $\overrightarrow{\overline{\mathbf{x}}}$ peut s'effectuer en $O(n)$ [Cormen *et al.*, 2001, page 189]. On peut de plus remarquer que cet algorithme est bien adapté à une implantation événementielle de la propagation de contraintes : dans le cas d'une mise à jour de l'un des $\overline{\mathbf{x_i}}$, seules les lignes 2–4 ont besoin d'être exécutées (car la mise à jour de $\overline{\mathbf{y}}$ se chargera de vider le domaine de $\mathbf{y}$ si la condition de la ligne 1 n'est plus satisfaite) ; dans le cas d'une mise à jour de $\underline{\mathbf{y}}$, seules les lignes 1–3 ont besoin d'être exécutées ; enfin, tout autre mise à jour ne nécessite aucune exécution de l'algorithme. La procédure peut aussi bénéficier du stockage du vecteur ordonné $\overrightarrow{\overline{\mathbf{x}}}^{\downarrow}$ et de sa mise à jour lorsque l'un des $\overline{\mathbf{x_i}}$ change, mise à jour qui nécessite un temps $O(n)$. Cette opération nous permet d'accéder à $\overrightarrow{\overline{\mathbf{x}}}^{\downarrow}_{k}$ en temps $O(1)$.

Nous pouvons noter que puisque toutes les contraintes en argument de la méta-contrainte **AtLeast** sont linéaires, cette méta-contrainte peut aussi être exprimée par un ensemble de contraintes linéaires, ce qui rend notre algorithme implantable dans un solveur linéaire (si toutes les autres contraintes sont linéaires). La méthode classique [Garfinkel et Nemhauser, 1972, page 11] consiste à exprimer notre contrainte **AtLeast** en introduisant n variables 0–1 $\{\delta_1, \ldots, \delta_n\}$, ainsi qu'un ensemble de contraintes linéaires $\{\mathbf{y} \leq \mathbf{x_1} + \delta_1\overline{\mathbf{y}}, \ldots, \mathbf{y} \leq \mathbf{x_n} + \delta_n\overline{\mathbf{y}}, \sum_{i=1}^{n} \delta_i \leq n - k\}$.

5.4.3.4 Transformations max-min

Il existe un autre moyen de faire apparaître la version triée du vecteur objectif, sans toutefois utiliser des contraintes spécifiques associées à leur mécanisme de propagation comme le font les deux algorithmes précédents : on peut utiliser un ensemble de «transformations max-min». Cette solution, introduite dans [Maschler *et al.*, 1992] (et citée dans [Ogryczak, 1997]) afin de traiter des problèmes d'optimisation leximin sur des ensembles non convexes d'alternatives, est fondée sur l'idée suivante : si l'on remplace deux composantes $\mathbf{u_i}$ et $\mathbf{u_j}$ du vecteur objectif par deux variables $\mathbf{m}$ et $\mathbf{M}$ qui représentent respectivement le minimum et le maximum de ces deux composantes, on ne change pas l'ensemble des solutions leximin-optimales.

Proposition 5.5 *Soient $(\mathscr{X}, \mathcal{D}, \mathscr{C})$ un réseau de contraintes, $\overrightarrow{\mathbf{u}}$ un vecteur objectif, et $\mathbf{u_i}$ et $\mathbf{u_j}$ deux variables distinctes de $\overrightarrow{\mathbf{u}}$. Soient aussi $\mathbf{m}$ et $\mathbf{M}$ deux nouvelles variables, et $(\mathscr{X}', \mathcal{D}', \mathscr{C}')$ le réseau de contraintes défini de la manière suivante : $\mathscr{X}' = \mathscr{X} \cup \{\mathbf{m}, \mathbf{M}\}$, $\mathcal{D}' = \langle \mathcal{D}, (\mathbf{m} : \mathscr{D}_{\mathbf{m}}, \mathbf{M} : \mathscr{D}_{\mathbf{M}}) \rangle$, avec $\mathscr{D}_{\mathbf{m}} = \mathscr{D}_{\mathbf{M}} = [\![\min(\underline{\mathbf{u_i}}, \underline{\mathbf{u_j}}), \max(\overline{\mathbf{u_i}}, \overline{\mathbf{u_j}})]\!]$, et $\mathscr{C}' = \mathscr{C} \cup \{\mathbf{m} = \mathbf{Min}(\mathbf{u_i}, \mathbf{u_j}), \mathbf{M} = \mathbf{Max}(\mathbf{u_i}, \mathbf{u_j})\}$. Nous définissons de même $\overrightarrow{\mathbf{u}'}$ comme étant le vecteur égal à $\overrightarrow{\mathbf{u}}$ sur toutes ses composantes, excepté $\mathbf{u_i}$ qui $\mathbf{u_j}$ qui sont remplacées par $\mathbf{m}$ et $\mathbf{M}$.*

Nous avons de manière évidente $maxleximin(\mathscr{X}, \mathcal{D}, \mathscr{C}, \overrightarrow{\mathbf{u}}) = maxleximin(\mathscr{X}', \mathcal{D}', \mathscr{C}', \overrightarrow{\mathbf{u}'})_{\downarrow \mathscr{X}}$.

En appliquant de manière récursive cette règle de reformulation, nous pouvons remplacer le vecteur objectif $\overrightarrow{\mathbf{u}}$ en introduisant des nouvelles variables $\overrightarrow{\mathbf{u}'}$ et les contraintes suivantes :

- ▷ $\forall i \in [\![1, n-1]\!]$, $\mathbf{u'_i} = \mathbf{Max}(\mathbf{Min}\{\mathbf{u_1}, \ldots, \mathbf{u_i}\}, \mathbf{u_{i+1}})$,
- ▷ $\mathbf{u'_n} = \mathbf{Min}\{\mathbf{u_1}, \ldots, \mathbf{u_n}\}$.

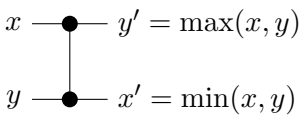
En utilisant cette reformulation, le minimum des variables objectif apparaît de manière naturelle comme une nouvelle variable, et cette variable constitue clairement le seul sous-ensemble saturé de cardinalité minimale parmi les variables du vecteur objectif. Tout comme dans l'algorithme 2, cette variable est fixée à sa valeur maximale $\widehat{m}$, puis enlevée de l'ensemble des variables objectif.

Il est très intéressant de remarquer que se cachent derrière les transformations max-min une interprétation de l'algorithme de tri fondée sur les réseaux de comparaisons [Cormen *et al.*, 2001,

page 704]. Les réseaux de comparaisons sont introduits en tant que modèles de calcul n'utilisant qu'un seul opérateur, l'opérateur de comparaison :

Définition 5.12 (Comparateur) *Un* comparateur *est un dispositif possédant deux entrées numériques x et y et deux sorties x' et y', et qui effectue les fonctions suivantes : $x' = \max(x, y)$ et $y' = \min(x, y)$.*

Un comparateur est représenté graphiquement de la manière suivante :



Un certain nombre d'algorithmes de tri de tableaux (ou vecteurs) ont été adaptés au modèle des réseaux de comparaisons. L'algorithme d'optimisation leximin qui a été proposé dans [Maschler *et al.*, 1992] et que nous introduisons ici utilise de manière implicite l'un de ces algorithmes de tri : chaque utilisation d'un comparateur correspond à une reformulation min-max. Cet algorithme de tri est une adaptation de l'algorithme de tri à bulles pour les réseaux de comparaisons, et est représenté de manière graphique dans la figure 5.4.

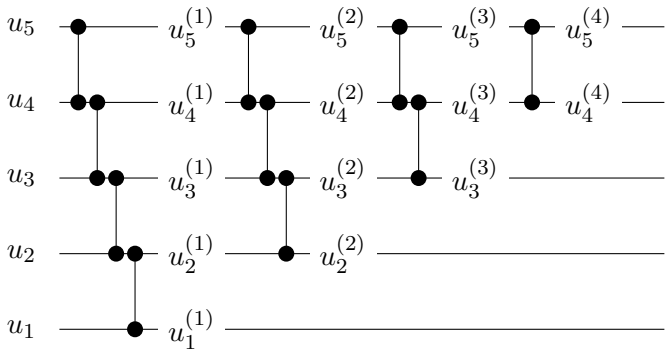


Figure 5.4 — Le réseau de comparaisons correspondant à l'algorithme de tri utilisé de manière implicite dans l'algorithme 8 pour $n = 5$.

Chaque comparateur de la figure 5.4 est «implanté» par deux contraintes dans l'algorithme 8, et chaque point correspond à une variable différente. Noter que les variables et les contraintes sont introduites couche par couche, puisqu'à chaque pas nous n'avons besoin que de la variable minimale (la variable $\mathbf{u_i^{(i)}}$ dans la figure et dans l'algorithme). Les couches sont introduites à chaque pas par la fonction `MinLayer`. Rappelons de plus qu'avant d'introduire une nouvelle couche, on doit restreindre l'ensemble des solutions admissibles à celles telles que la variable objectif minimum est maximale. C'est le rôle de la ligne 9 de l'algorithme 8.

Exemple 5.1.e Nous allons illustrer le fonctionnement de l'algorithme 8 sur l'exemple 5.1. L'algorithme fonctionne en substituant successivement les variables objectif, afin de faire apparaître la variable maximin de manière naturelle.

Le tableau ci-dessous liste l'ensemble des solutions pour les variables objectif initiales et pour les nouvelles. Au premier pas de l'algorithme, on introduit les variables $\overrightarrow{\mathbf{u^{(1)}}}$, et on maximise $\mathbf{u_3^{(1)}}$. On fixe cette dernière variable à sa valeur maximale, ce qui a pour effet de restreindre l'ensemble des solutions (ce qui explique les cellules vides dans le tableau). Au deuxième pas de l'algorithme, on introduit les variables $\overrightarrow{\mathbf{u^{(2)}}}$, et on maximise $\mathbf{u_2^{(2)}}$. Enfin, on introduit et on maximise $\mathbf{u_1^{(3)}}$ au tout dernier pas de l'algorithme.

Algorithme 8 — Résolution du problème [MAXLEXIMINCSP] par transformations max-min.

entrée : Un réseau de contraintes $(\mathscr{X}, \mathcal{D}, \mathscr{C})$; un vecteur objectif $(\mathbf{u_1}, \ldots, \mathbf{u_n}) \in \mathscr{X}^n$.
sortie : Une solution au problème [MAXLEXIMINCSP].

si solve$(\mathscr{X}, \mathcal{D}, \mathscr{C}) =$ *«Inconsistant»* **alors retourner** *«Inconsistant»*;
$(\mathscr{X}', \mathcal{D}', \mathscr{C}') \leftarrow (\mathscr{X}, \mathcal{D}, \mathscr{C})$;
$\overrightarrow{\mathbf{u}}^{(\mathbf{0})} \leftarrow \overrightarrow{\mathbf{u}}$;
pour $i \leftarrow 1$ **à** n **faire**
  $\mathscr{X}' \leftarrow \mathscr{X}' \cup \{\mathbf{y_i^{(i)}}, \ldots, \mathbf{y_n^{(i)}}\} \cup \{\mathbf{u_i^{(i)}}, \ldots, \mathbf{u_n^{(i)}}\}$;
  $\mathcal{D}' \leftarrow \langle \mathcal{D}', (\mathbf{y_i^{(i)}} : \mathscr{D}_{\mathbf{y_i^{(i)}}}, \ldots, \mathbf{y_n^{(i)}} : \mathscr{D}_{\mathbf{y_n^{(i)}}}), (\mathbf{u_i^{(i)}} : \mathscr{D}_{\mathbf{u_i^{(i)}}}, \ldots, \mathbf{u_n^{(i)}} : \mathscr{D}_{\mathbf{u_n^{(i)}}}) \rangle$ avec $\mathscr{D}_{\mathbf{u_j^{(i)}}} = \mathscr{D}_{\mathbf{y_j^{(i)}}} = [\![\min_k(\underline{\mathbf{u_k}}), \max_k(\overline{\mathbf{u_k}})]\!]$;
  $\mathscr{C} \leftarrow \mathscr{C} \cup \texttt{MinLayer}(\overrightarrow{\mathbf{u}}^{(\mathbf{i-1})}, \overrightarrow{\mathbf{u}}^{(\mathbf{i})}, \overrightarrow{\mathbf{y}}^{(\mathbf{i})})$;
  $\widehat{v}_{(i)} \leftarrow$ **maximize**$(\mathscr{X}', \mathcal{D}', \mathscr{C}', \mathbf{u_i^{(i)}})$;
  $\mathbf{u_i^{(i)}} \leftarrow \widehat{v}_{(i)}(\mathbf{u_i^{(i)}})$;
retourner $\widehat{v}_{(n)\downarrow\mathscr{X}}$;

Fonction $\texttt{MinLayer}(\{\mathbf{u_1}, \ldots, \mathbf{u_m}\}, \{\mathbf{v_1}, \ldots, \mathbf{v_m}\}, \{\mathbf{y_1}, \ldots, \mathbf{y_m}\})$

entrée : Trois vecteurs de taille m.
sortie : Un ensemble de contraintes.

$\mathscr{C} \leftarrow \{\mathbf{y_m} = \mathbf{u_m}\}$;
pour $i \leftarrow m$ **à** 2 **faire**
  $\mathscr{C} \leftarrow \mathscr{C} \cup \{\mathbf{v_i} = \mathbf{Max}(\mathbf{y_i}, \mathbf{u_{i-1}}), \mathbf{y_{i-1}} = \mathbf{Min}(\mathbf{y_i}, \mathbf{u_{i-1}})\}$;
$\mathscr{C} \leftarrow \{\mathbf{v_1} = \mathbf{y_1}\}$;
retourner $\mathscr{C}$;

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

5.4.4 Aspects heuristiques

Les algorithmes de programmation par contraintes dédiés au problème [MAXLEXIMINCSP] peuvent bénéficier d'une heuristique d'instanciation de variables spécifique qui tire partie de la sémantique particulière du leximin pour guider la recherche arborescente vers de meilleures solutions. En effet, lors de la recherche d'une solution leximin-optimale, la composante la plus basse du vecteur objectif est déterminante, car c'est elle qui doit être augmentée en premier pour obtenir une meilleure solution. Ceci nous donne une idée très simple d'heuristique : la prochaine variable à instancier devra augmenter le plus possible la plus basse valeur du vecteur objectif.

Dans les problèmes de décision collective ou d'allocation de ressources, le vecteur objectif, qui est le profil d'utilités, dépend en général de variables de décision (typiquement les variables 0–1 correspondants aux variables booléennes $\mathbf{alloc}(\mathbf{o}, \mathbf{i})$). Dans ce contexte, la prochaine variable à

instancier devra être la variable de décision qui fait le plus augmenter l'utilité de l'agent le moins satisfait au moment de la décision.

Le but d'une telle heuristique est de faire tendre le vecteur objectif relativement vite vers une solution assez égalitaire, pour laquelle il ne sera alors plus possible d'augmenter l'utilité du moins satisfait des agents. Dans ce cas, on devra ensuite chercher à augmenter l'utilité du deuxième agent dans l'ordre croissant d'utilités. Ainsi de suite jusqu'au dernier agent.

Naturellement, ces considérations ne donnent qu'une piste pour la définition d'une heuristique, qui doit être déclinée pour chaque instance traitée. On pourra prendre en compte d'autres considérations que l'utilité de l'agent le moins satisfait, comme par exemple la taille d'un lot à attribuer dans un problème d'allocations de ressources, ou le rapport du gain d'utilité espéré pour les agents sur la consommation de la ressource. Comme nous le verrons brièvement au chapitre 6 consacré aux expérimentations, ces aspects heuristiques sont cruciaux pour l'efficacité des algorithmes.

5.5 Conclusion : au-delà du leximin ?

Nous avons introduit dans ce chapitre un certain nombre d'approches du problème de calcul d'une solution leximin-optimale dans un réseau de contraintes. Nous avons justifié l'intérêt de se pencher sur ce problème, tout d'abord en rappelant l'ensemble des bonnes propriétés du préordre leximin pour l'agrégation des utilités des agents, et dans un deuxième temps en mettant en valeur la relative difficulté algorithmique de ce problème, et l'inadéquation de l'approche à base de transcription du préordre sous la forme d'une fonction d'utilité collective.

Nous nous devons cependant de tempérer ces commentaires sur le préordre leximin, car il peut poser quelques problèmes dans certaines situations concrètes que nous allons brièvement exposer. Rappelons tout d'abord que le préordre leximin a été introduit comme un raffinement de l'ordre social induit par la fonction min pour pallier les problèmes liés à l'idempotence de l'opérateur min : dans un problème à n agents, les profils d'utilité $(0, \ldots, 0)$ et $(0, 1000, \ldots, 1000)$ sont indifférents pour l'ordre social induit par la fonction min (car ils produisent la même utilité 0).

L'ordre leximin permet d'éviter ce type de problèmes en permettant de distinguer selon les composantes suivantes plusieurs profils qui ont un minimum identique. Cependant, dans de nombreux problèmes de partage, l'ordre leximin peut sembler presque aussi extrême que l'ordre induit par min, car il n'autorise aucune concession sur la composante minimum des profils d'utilité, et ce même si une légère diminution de cette composante minimum permettait d'augmenter de manière considérable les utilités des autres agents. Considérons par exemple les deux profils d'utilité suivants : $(10, \ldots, 10)$ et $(9, 1000, \ldots, 1000)$. Entre ces deux profils, le préordre leximin (tout comme le min) préférera le premier, alors qu'il peut sembler plus naturel dans beaucoup de cas de faire une concession sur la valeur minimale si les autres utilités sont vraiment meilleures[2]. En d'autres termes, il n'y a aucune compensation possible entre une infime perte d'utilité de l'agent le moins heureux et un immense gain sur les autres agents.

Cette absence de compensation soulève un autre problème pratique, liée au calcul d'une solution leximin-optimale. Lorsque l'on se place dans le contexte de la programmation par contrainte, où les domaines des variables sont finis, l'utilisation des algorithmes décrits dans ce chapitre ne pose pas de problème de correction. En revanche, il en est autrement lorsque l'on se place dans le cadre du calcul flottant, comme c'est le cas lorsque l'on traite le problème à l'aide d'un solveur linéaire (comme nous le ferons en partie dans le chapitre 6). À chaque étape de l'algorithme, une erreur

[2] Remarque : aurait-on accepté de faire cette concession si les deux profils avaient été $(1, 10, \ldots, 10)$ et $(0, 1000, \ldots, 1000)$?

d'approximation sur une composante du leximin peut avoir des répercussions énormes sur le reste du calcul : soit les composantes du vecteur sont très différentes de ce qu'elles devraient être en vertu de l'absence de compensation possible, soit l'algorithme s'arrête sur une incohérence aux étapes suivantes.

Prenons un exemple : considérons un problème pour lequel il existe deux profils d'utilité admissibles $(9, 10, 10, 10, 10)$ et $(9, 9.9, 1000, 1000, 1000)$. Supposons qu'au moment du calcul de la deuxième composante du leximin, le solveur fasse une erreur numérique et trouve 9.9 au lieu de 10. Cela a une immense répercussion sur la suite du vecteur, puisque les composantes suivantes seront égales à 1000, alors qu'elles devraient être égales à 100 si l'algorithme était correct. Cet exemple met en évidence la grande sensibilité du préordre leximin vis-à-vis des composantes faibles du vecteur objectif.

La mise en évidence de ces deux problèmes pose donc la question de la pertinence du préordre leximin dans les problèmes d'agrégation d'utilités. En d'autres termes, peut-on trouver un ordre social moins «drastique» que l'ordre leximin, c'est-à-dire qui admette des compromis sur la perte d'utilité des agents les moins riches ?

5.5.1 Un leximin à seuil

La notion d'égalité est relativement mal définie dans le domaine du calcul flottant. Dans ce contexte, il est nécessaire d'introduire une certaine tolérance aux approximations, en considérant que deux nombres sont égaux si leur différence est inférieure à un certain seuil représentant la précision machine (certains langages ou systèmes de calcul introduisent ce seuil de manière explicite : voir par exemple la variable `eps` dans le logiciel de calcul numérique *Matlab*).

De manière plus générale, un agent sera souvent indifférent entre deux utilités très proches, mais ne le sera pas si les utilités sont plus éloignées. Comme nous l'avons vu au chapitre 1, cette remarque est à la base de la définition des préférences de type semi-ordres, qui font intervenir un seuil d'indifférence q en deçà duquel la différence entre deux utilités n'est plus pertinente.

Peut-on adapter l'idée des préférences à seuil au préordre leximin ? D'un point de vue purement formel, la réponse est positive, et la notion de leximin à seuil peut être définie comme suit. Soient deux profils d'utilité $\overrightarrow{u}$ et $\overrightarrow{v}$, et $\varepsilon > 0$. Alors les deux relations $\prec^{(\varepsilon)}_{leximin}$ et $\sim^{(\varepsilon)}_{leximin}$ dont les suivantes :

$$\begin{cases} \overrightarrow{u} \prec^{(\varepsilon)}_{leximin} \overrightarrow{v} & \Leftrightarrow \quad \exists i \in [\![1,n]\!] \text{ t.q. } \begin{cases} \forall j < i, |u^{\uparrow}_j - v^{\uparrow}_j| \leq \varepsilon, \text{ et} \\ u^{\uparrow}_i > v^{\uparrow}_i + \varepsilon, \end{cases} \\ \overrightarrow{u} \sim^{(\varepsilon)}_{leximin} \overrightarrow{v} & \Leftrightarrow \quad \forall i \in [\![1,n]\!], \ |u^{\uparrow}_i - v^{\uparrow}_i| \leq \varepsilon. \end{cases}$$

La simplicité de cette extension cache néanmoins quelques problèmes. Le premier de ces problèmes est que cette relation n'est plus compatible avec la relation de dominance de Pareto qu'au sens faible (tout comme la fonction min). Considérons par exemple le cas où $\varepsilon = 10$, et les deux profils d'utilité suivants : $\overrightarrow{u} = (10, 10, 10)$ et $\overrightarrow{v} = (15, 15, 15)$. On a $\overrightarrow{u} \sim^{(\varepsilon)}_{leximin} \overrightarrow{v}$ alors que $\overrightarrow{u}$ domine $\overrightarrow{v}$ au sens de Pareto. Ce problème n'est pas vraiment rédhibitoire : à l'issu du processus de choix collectif, il suffit d'éliminer les décisions non Pareto-efficaces.

La relation leximin à seuil pose un autre problème légèrement plus sérieux. Considérons par exemple le cas où $\varepsilon = 10$, et introduisons les trois profils d'utilité suivants : $\overrightarrow{u} = (10, 30, 30)$, $\overrightarrow{v} = (10, 15, 70)$ et $\overrightarrow{w} = (10, 22, 50)$. Nous avons : $\overrightarrow{u} \prec^{(\varepsilon)}_{leximin} \overrightarrow{v}$ (la deuxième valeur est discriminante), $\overrightarrow{v} \prec^{(\varepsilon)}_{leximin} \overrightarrow{w}$ (la troisième valeur est discriminante), et $\overrightarrow{w} \prec^{(\varepsilon)}_{leximin} \overrightarrow{u}$ (la troisième valeur est

discriminante). Nous voyons donc que la relation $\prec_{leximin}^{(\varepsilon)}$ est non transitive, ce qui revient à dire — car il s'agit d'un préordre total — qu'elle est cyclique. La question est de choisir quelle alternative choisir dans ce cas-là. Cette question n'est pas nouvelle dans le domaine de la représentation et de l'agrégation de préférences, et il existe des solutions pour traiter ce problème de cyclicité de la relation de préférence [Vincke, 1989]. Nous pouvons cependant nous interroger sur la pertinence (et la complexité) de cette approche, sachant que le cadre du *welfarisme* cardinal nous fournit d'autres moyens d'introduire des concessions sur le préordre leximin, comme nous allons le voir.

5.5.2 Les moyennes pondérées ordonnées

Les moyennes pondérées ordonnées (OWA), qui ont été introduites dans le chapitre 1 en tant que famille de fonctions d'utilité collective, constituent un relâchement assez naturel du leximin (qui présente aussi l'avantage d'être facilement paramétrable selon l'importance de la concession que l'on veut faire par rapport au leximin). L'ensemble des OWA équitables est l'ensemble des OWA dont le vecteur de coefficients $\overrightarrow{w}$ est tel que $i > j \Rightarrow w_i < w_j$: c'est l'ensemble des OWA qui respectent le principe de réduction des inégalités.

Nous pouvons nous interroger sur la transcription des algorithmes de résolution du problème [MAXLEXIMINCSP] au problème de maximisation d'une fonction d'utilité collective de type OWA. La plupart des algorithmes introduits dans ce chapitre sont fondés sur la propriété d'absence de compensation entre les utilités des agents les moins riches et des agents les plus riches, qui implique que chaque composante du vecteur leximin-optimal peut être calculée séparément, ce qui n'est plus le cas pour les OWA.

En revanche, une grande partie du travail effectué autour de ces algorithmes concerne la définition de la notion de tri d'un vecteur sous forme de contrainte, et les algorithmes de propagation dédiés à cette notion de tri. Cette contrainte de tri est bien évidemment indispensable pour la définition de la variable d'utilité collective, définie par une moyenne pondérée ordonnée. Nous pouvons donc bénéficier de la contrainte **Sort**, ou encore de la définition du vecteur objectif trié par l'introduction de contraintes **Max** et **Min**, couplées avec une contrainte linéaire sur le vecteur objectif trié, afin de définir la fonction d'utilité collective OWA.

Notons que l'on trouve dans la littérature un article qui traite de moyennes pondérées ordonnées «équitables» dans un contexte de recherche opérationnelle : [Ogryczak et Śliwiński, 2003]. Cet article propose une modélisation sous forme de programme linéaire du problème de maximisation d'un OWA sous contraintes linéaires. Il pourrait être intéressant de comparer cette approche avec l'approche programmation par contraintes fondée sur la contrainte **Sort**.