Table des matières

CHAPITRE 3

Logiciels de PL et Xpress-MP

3.1 Introduction

Grâce aux progrès de l'informatique, l'offre de logiciels commerciaux permettant de résoudre des PL de plus en plus gros a considérablement augmenté. On peut maintenant résoudre en routine des PL à 100 000 variables et contraintes et des PLNE à 1 000 variables et contraintes avec les codes les plus performants. Les prix de ces logiciels autrefois réservés à une petite communauté ont fortement diminué et deviennent très abordables.

Ce chapitre a pour but de présenter les logiciels les plus connus qui permettent de résoudre les problèmes de PL, dont le logiciel Xpress. L'objectif n'est pas d'argumenter en faveur de tel ou tel logiciel, mais de donner quelques indices pour effectuer un investissement dans ce domaine. Pour ce livre, nous avons retenu le logiciel Visual Xpress version 3 pour deux raisons principales. La première est que la syntaxe de Xpress est très intuitive, comme nous avons pu le constater dans les enseignements que nous effectuons. La deuxième est que Visual Xpress (version Windows) est un logiciel *tout en un*. Il contient à la fois un modeleur et un solveur, mais leur accès est bien séparé dans l'interface, ce qui facilite la compréhension de la méthodologie de résolution d'un modèle. Une version de Visual Xpress est donnée sur le CD-Rom accompagnant ce livre.

Bien que l'éditeur ait continué à faire évoluer son produit, nous avons préféré garder la version 3 du logiciel pour transcrire les exemples de ce livre. La nouvelle version (aujourd'hui nommée Xpress-IVE) propose un langage de modélisation plus proche du langage informatique que du langage mathématique et nous souhaitons éviter ceci. L'expérience de l'enseignement auprès des étudiants montre qu'une version plus informatisée du langage est plus difficile à appréhender. De plus, l'éditeur Dash a mis au point un outil pour traduire automatiquement les modèles de Visual Xpress vers Xpress-IVE. Ainsi le lecteur qui souhaite faire évoluer son logiciel pourra le faire sans soucis. Une version téléchargeable gratuite (bien que bridée) de Xpress-IVE est disponible sur le site de la société Artelys (distributeur Français pour Dash) : *http://www.artelys.fr*.

Le paragraphe 3.2 présente le principe de fonctionnement d'un logiciel de PL et les principaux produits commercialisés actuellement. Le paragraphe suivant introduit le langage de modélisation Xpress. Le paragraphe 3.4 présente un tour d'horizon des différents langages de modélisation au travers d'un exemple simple. Les deux derniers paragraphes présentent les chapitres d'applications qui suivent, ainsi que certaines précautions à prendre. Pour finir, citons une référence Internet, mise à jour tous les mois, qui donne des réponses à toutes les questions que pourraient se poser des utilisateurs ou des concepteurs de logiciels de PL, il s'agit d'une *FAQ (Frequently Asked Questions* ou *Foire Aux Questions*) en anglais : *http://ww-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html*

3.2 Les logiciels actuels

3.2.1 Principes de fonctionnement

La résolution d'un problème de programmation linéaire s'effectue en deux grandes phases : la modélisation, puis la résolution (on pourrait aussi ajouter une troisième phase d'analyse des résultats). Historiquement, les premiers logiciels ne servaient qu'à résoudre des programmes linéaires déjà modélisés et donnés sous forme numérique. Il s'agissait donc de *codes algorithmiques* de résolution, appelés aussi *solveurs*. Par la suite sont venus se greffer les *langages de modélisation* ou *modeleurs*.

Les solveurs sont distribués sous forme de logiciels autonomes, ou encore de bibliothèques de sous-programmes à inclure dans des programmes d'application. Le programme linéaire doit être au préalable préparé en mémoire, sous forme de matrices ou de listes de valeurs numériques. La plupart des solveurs offrent cependant une fonction permettant de charger un PL saisi au préalable dans un fichier texte.

Selon les produits, les PL sont stockés dans des fichiers sous forme matricielle ou sous forme d'équations proches de l'écriture mathématique, par exemple "3*X1+2*X2 = 3". Le format matriciel MPS a été popularisé par le premier solveur célèbre, MPSX d'IBM. Il permet de découper la matrice d'un grand PL en lignes de longueur fixe, de définir des portions de lignes et de colonnes, etc. Ce format est encore utilisé comme moyen d'échanger des données entre les logiciels actuels.

Les modeleurs aident l'utilisateur à formuler son problème correctement et à analyser les solutions obtenues après résolution par un solveur. Un modeleur s'appuie sur un langage de description du problème et lie cette modélisation symbolique (le modèle) à un jeu de données. Typiquement, un modeleur permet de définir des paramètres (par exemple un nombre de produits à fabriquer), des variables indicées, des tableaux de données, des sommations de variables indicées, etc. L'utilisateur peut ainsi formuler son problème de façon compacte, dans une syntaxe proche de l'écriture mathématique.

Avec un solveur, l'ajout d'un produit supplémentaire dans un problème de planification de production oblige à insérer des nouvelles colonnes ou lignes de valeurs numériques dans la matrice du programme linéaire. Un modeleur permet de définir un modèle générique, paramétré par le nombre n de produits. Ce modèle peut être rendu complètement indépendant des valeurs numériques, stockées dans des fichiers séparés. Pour ajouter un nouveau produit, il suffira de préciser la nouvelle valeur de n et de modifier les fichiers de données.

Le modèle ne sera pas changé. Cette souplesse accélère les développements, diminue les risques d'erreurs et donne des modèles faciles à documenter et à relire.

Comment marche un modeleur ? Il commence par compiler le modèle pour détecter d'éventuelles erreurs de syntaxe. Si la syntaxe est correcte, il ouvre les fichiers de données et se charge de la tâche fastidieuse de générer le programme linéaire sous forme matricielle. La matrice des valeurs numériques, qui peut être énorme par rapport au modèle sous forme symbolique, sera traitée par un solveur.

Aujourd'hui, certains modeleurs peuvent être achetés seuls puis interfacés avec divers solveurs d'autres éditeurs de logiciels, par exemple grâce au format d'échange matriciel MPS. D'autres modeleurs sont étroitement imbriqués avec un solveur propriétaire, c'est une des raisons pour laquelle l'amalgame entre modeleur et solveur est très vite fait. Il est tout de même important de bien faire la distinction.

3.2.2 Les principaux solveurs

En tête de liste on retrouve deux concurrents : OSL, renommé Optimization Solutions (d'IBM) et Cplex Callable Library (d'ILOG), qui ont été les premiers à proposer des codes commerciaux robustes et rapides. OSL a été le premier solveur commercial comprenant un algorithme de point intérieur. Il n'est pas proposé avec un modeleur particulier. Robuste et rapide, il est interfaçable avec la plupart des langages de modélisation. Depuis une récente acquisition de Cplex Inc. par ILOG, l'offre du solveur Cplex s'est beaucoup développée. On le retrouve par exemple comme solveur interne dans des logiciels d'optimisation comme ILOG Planner.

A côté de ces deux grands, on trouve d'autres codes très robustes et qui ont su se faire une place sur le marché des logiciels d'optimisation. Citons C-Whiz de Ketron Management Science qui tourne sur mainframe et PC, HS/LP, le solveur de Harvely System Inc. qui s'interface très bien avec le modeleur OMNI de la même société, LAMPS d'Advanced Mathematical Software Ltd., LINDO de Lindo Systems Inc., GAUSS d'Aptech Systems Inc., LOQO de l'université de Princeton qui propose des versions gratuites pour le monde académique, MINOS de Stanford Business Software Inc., et bien d'autres encore. Des produits plus connus comme Maple, Matlab ou Excel proposent aussi des extensions permettant de résoudre des PL. Enfin, il existe même un solveur simple du domaine public, LPSolve, diffusé avec son source en C.

3.2.3 Les principaux modeleurs

Parmi les modeleurs, AMPL est le plus connu et le plus célèbre. Sa particularité est de pouvoir appeler pratiquement tous les solveurs existants. Les éditeurs de codes algorithmiques en ont fait une référence et se servent de la popularité d'AMPL pour promouvoir leur solveur. C'est sans doute ce qui a rendu AMPL plus célèbre encore. Ce type de langage, qui est très utile aux décideurs d'entreprise en simplifiant la modélisation et la résolution de problèmes d'optimisation, est en pleine expansion. L'offre s'est considérablement étoffée ces cinq dernières années.

Les autres langages de modélisation les plus connus, et qui seront brièvement décrits au paragraphe 3.3, sont GAMS (de Gams Developpement Corp.), qui permet des modèles non

linéaires, OPL Studio (d'ILOG), qui peut appeler des solveurs de propagation de contraintes en parallèle avec la programmation linéaire, Lingo (de Lindo Systems), très employé dans le monde académique pour les enseignements outre-Atlantique, et MPL for Windows (de Maximal Software). On ne peut pas dire qu'un langage est définitivement supérieur à un autre. En fait, pour choisir un langage, la meilleure façon est de tester les versions limitées qui sont proposées sur les sites web des éditeurs de logiciels.

3.2.4 Les environnement de développement intégrés (EDI)

Pour satisfaire une clientèle de plus en plus exigeante, les interfaces de développement (EDI) sont apparues et proposent des menus, des fenêtres et d'autres fonctionnalités pour aider l'utilisateur à rédiger et déboguer son modèle. Pour les PC, la plupart des éditeurs fournissent un EDI agréable à utiliser. Cette interface est souvent la version disponible gratuitement sur le web. Parmi les éditeurs déjà cités, ceux qui proposent une interface sont GAMS, OPL Studio, Lingo, MPL for Windows et, bien sûr, Visual Xpress, qui sera utilisé dans ce livre. La nouvelle version Xpress-IVE propose elle aussi un EDI très proche de celui de Visual Xpress.

3.2.5 Sites web des éditeurs de logiciels

Un article de R. Fourer [Fourer 1999] très complet reprend les caractéristiques des différents logiciels et montre tous les liens que l'on peut trouver entre les langages de modélisation, les solveurs utilisés et les environnements de développement intégrés. La liste suivante donne les sites web des éditeurs de logiciels dont nous avons parlé. D'autres sites sont indiqués dans l'article de Fourer. Il existe aussi des versions gratuites de ce type de logiciel, mais plus à l'attention des développeurs, et l'on peut en retrouver une liste exhaustive sur la FAQ citée en introduction de ce chapitre.

- AMPL *http://www.ampl.com*
- Cplex *http://www.ilog.fr*
- C-Whizz *http://www.keytronms.com*
- GAMS *http://www.gams.com*
- GAUSS *http://www.aptech.com*
- HS/LP *http://www.harvely.com*
- LAMPS *info@amsoft.demon.co.uk*
- Lindo et Lingo *http://www.lindo.com*
- LOQO *http://www.princetn.edu/~rvdb*
- MINOS *http://www.SBSI-SOL-Optimize.com*
- MPL for Windows *http://www.maximal-usa.com*
- OPL Studio *http://www.ilog.fr*
- OSL *http://www6.software.ibm.com/es/oslv2/startme.html*

- Xpress *http://www.dash.co.uk*
- Xpress *http://www.artelys.fr* (distributeur Français de Dash)

3.3 Visual Xpress (interface et langage)

3.3.1 Préliminaires

Visual Xpress (Vx) est la version 3 pour Windows du logiciel Xpress, qui est également disponible sur d'autres plates-formes comme Unix. Il intègre un modeleur, un solveur et un éditeur pour saisir et modifier les modèles et les données. Il existe une version *Hyper*, non bridée mais nécessitant une clé de protection (*dongle*), et une version *Student*, limitée. Avec la nouvelle version Xpress-IVE, il existe les mêmes offres et une version *Student* est téléchargeable gratuitement sur le site web *http://www.artelys.fr*. Une version *Hyper* est aussi disponible sur ce site mais pour une durée de 30 jours uniquement.

L'objectif de cette partie est de décrire l'environnement de Vx Student pour éviter certains pièges, de rappeler sa syntaxe de base, le processus de développement et de résolution. Pour toute question le lecteur pourra se référer au guide fourni avec Visual Xpress Student. Dans ce guide, la section "A quick tour of Visual Xpress" décrit aussi les principales fonctionnalités de l'interface et le principe de développement en Vx, mais en anglais uniquement.

3.3.2 Visual Xpress version 3

Au moment de la rédaction de la nouvelle version de ce livre, la version 3 de Xpress a été remplacée par Xpress-IVE sur le serveur web de Dash et d'Artelys. Pour des raisons de simplicité d'utilisation, nous pensons que la précédente version est encore la mieux adaptée pour intégrer la programmation linéaire et le processus de modélisation. Dans la version 3, l'interface homme-machine permet de charger directement un fichier *mod* pour travailler. Comme dans tout logiciel, le chargement des fichiers se fait par l'intermédiaire du menu *Fichier*. Les fichiers modèles sont désormais portables entre les versions Unix et Windows de Xpress.

La version 3 est un exécutable unique 32 bits qui se configure automatiquement en version limitée Student si la clé de protection livrée avec les licences Hyper n'est pas détectée au moment du démarrage. Les limitations du mode Student ont été un peu relâchées en version 3. Le logiciel fonctionne sous Windows 95, 98, 2000 et NT. Les noms de fichiers ne sont pas limités à 8 caractères, grâce au mode 32 bits. On peut traiter des programmes à 300 variables et 800 variables + contraintes. Les limitations majeures qui subsistent sont les deux indices par variable et les 50 variables entières. Le CD-Rom fourni avec le livre contient la version 3.0. La nouvelle version *Student* de Xpress-IVE (téléchargeable sur le serveur http://www.artelys.fr.) est soumise aux mêmes limitations.

3.3.3 Xpress-IVE, une nouvelle version de Visual Xpress

Nous ne pouvions pas rééditer ce livre sans parler de la nouvelle version de Xpress. Dans un soucis de compétitivité, Dash a produit une nouvelle version de son langage de

modélisation accompagné d'une nouvelle version du solveur et de l'EDI. Cette nouvelle version améliore de nombreux points de la version 3 utilisée dans ce livre.

La nouvelle version du langage de modélisation s'est beaucoup enrichie. Elle a tellement changé que le langage de modélisation a été renommé *MOSEL modelling language*. D'une simplicité d'expression en langage pseudo-mathématique, nous sommes passés à un langage très proche de l'informatique. Ce langage, bien que plus puissant, est plus difficile à appréhender et donc un peu contraire à l'esprit de ce livre. Dans la nouvelle édition de ce livre, nous avons donc choisi de garder l'ancien langage. De plus, un traducteur automatique de Visual Xpress à Mosel est déjà opérationnel et sera prochainement disponible. Dans la suite de l'ouvrage, à l'exception du paragraphe qui décrit les autres langages de modélisation, nous ne parlerons que de Visual Xpress version 3.

3.3.4 Résolution de modèles

Avant de résoudre un problème chargé dans l'éditeur, assurez-vous du sens de l'optimisation (Min ou Max) dans *Problem/Optimiser options*. Par défaut, c'est Min. On peut aussi effectuer un double-clic sur Min ou Max dans la barre d'état de Vx pour inverser le sens de l'optimisation. Il faut ensuite résoudre le PL avec l'icône *Solve LP* ou le menu *Run/Solve LP*. Si le modèle n'est pas correct, des messages d'erreurs apparaissent dans une fenêtre en bas d'écran. Dans ce cas, double-cliquez sur les messages pour vous positionner dans le modèle à l'emplacement des erreurs et corrigez.

Si le modèle est correct, la matrice du PL est générée et passée à l'optimiseur. Une fois la résolution terminée, le nombre d'itérations et la valeur de la fonction objectif s'affichent. Si le compte-rendu affiche *N/A* (*not available*), c'est qu'il y a eu un problème de fichiers de données ou de taille de modèle. Ce genre d'erreurs n'est pas affiché en bas d'écran, contrairement aux erreurs du modeleur : il faut cliquer sur *View log* pour obtenir des détails.

Pour activer la recherche arborescente pour un PLNE, il faut commencer par utiliser la commande *Solve LP*, puis cliquer sur l'icône *Solve IP* (Integer Program) ou le menu *Run/Solve Global*. A noter que Vx Student, contrairement aux autres versions, accepte qu'on clique directement sur *Solve IP*, mais les solutions du PL relaxé ne sont alors pas consultables.

La version Student, limitée à deux indices par variable, permet une présentation claire et agréable des résultats sous forme de matrices, grâce à l'icône *View Results*.

3.3.5 Syntaxe Visual Xpress

Deux exemples pour commencer

Voici un premier exemple simple de PL Xpress dit *non générique* (sans indices). Les mots clés de Xpress sont soulignés, les autres ont été choisis par l'utilisateur :

```
MODEL Toto
VARIABLES
X1
X2
CONSTRAINTS
```

```
Profit: 2*X1 + 3*X2$
Cont1 : 3*X1 - X2 < 7
Cont2 : X1 + X2 = 5
END
```

Voici un modèle plus compliqué, pour un problème d'affectation de *n* étudiants à *n* sujets de projets. Chaque étudiant *i* a défini une note (préférence) *p*(*i*,*j*) pour le projet *j*. Le but est d'affecter les étudiants aux projets de façon à maximiser la satisfaction totale (somme des notes). Ce type de PL ayant automatiquement des variables en 0-1 à l'optimum, on n'a pas à déclarer que les variables doivent valoir 0 ou 1 (voir chapitre 2).

```
MODEL Affect
LET
n = 3
TABLES
p(n,n)
VARIABLES
x(n,n)
DISKDATA
p = Monfich.dat
CONSTRAINTS
Satisfaction        : Sum(i=1:n,j=1:n)p(i,j)*x(i,j)$
UnEtudParProj(j=1:n): Sum(i=1:n)x(i,j) = 1
UnProjParEtud(i=1:n): Sum(j=1:n)x(i,j) = 1
END
```

La matrice des préférences a été saisie au préalable dans le fichier *Monfich.dat*, qui est lu par la commande *Diskdata*. La fonction objectif (*à maximiser* au moment de la résolution) est la somme des préférences obtenue par l'affectation. Le premier groupe de contraintes garantit qu'un seul étudiant est affecté à chaque projet. Le deuxième oblige à affecter chaque projet à un seul étudiant. Notez que le même modèle mathématique peut être réutilisé pour un problème d'affectation 10×10 ; il suffit de changer la valeur de *n* et de fournir le fichier de préférences correspondant.

Structure générale des modèles et identificateurs

Un modèle commence par *Model* (avec un nom au choix) et se termine par *End*. Il peut comporter les sections suivantes : *Let* (définition de constantes comme Const en Pascal), *Tables* (définition de tableaux de constantes avec leurs dimensions), *Data* (initialisation de tables en dur), *Diskdata* (chargement de tableaux à partir d'un fichier), *Assign* (calcul de constantes à partir d'autres constantes), *Constraints* (fonction objectif et contraintes) et *Bounds* (bornes simples sur des variables et contraintes d'intégrité). Tous ces noms sont des mots clés et doivent être seuls sur leur ligne (sauf *Model* qui est suivi d'un nom). Un descriptif complet des différentes sections est donné dans l'aide de Visual Xpress. Voici cependant quelques bases suffisantes pour développer de nombreux modèles.

Les sections *Let*, *Tables*, *Data*, *DiskData*, *Assign* et *Variables* peuvent exister en plusieurs exemplaires, à condition que tout identificateur soit déclaré avant d'être initialisé ou utilisé. Les remarques suivantes sont valides pour tous les modèles :

- On peut mettre des lignes blanches, ou des espaces dans chaque ligne, pour aérer.

- On peut mettre des commentaires après un point d'exclamation (!).
- Les caractères accentués dans les commentaires peuvent provoquer des anomalies diverses.
- Les identificateurs sont *case significant* comme en C, *sauf les mots clés Xpress*.

Les identificateurs (noms de tables, de variables, de contraintes) choisis par l'utilisateur doivent contenir des lettres non accentuées, des chiffres ou des blancs soulignés (*underscore*). Le premier caractère doit être une lettre. Cependant, *seuls les seize premiers caractères sont significatifs* : ainsi, deux identificateurs *CapaciteDesDepotsNord* et *CapaciteDesDepotsSud* donnent une erreur "Duplicate name".

Identificateurs indicés : les noms de tables indicées obéissent à la règle normale : ils doivent être non ambigus sur seize caractères. Par contre, les noms de *variables* et de *contraintes* sont affichés dans le listing de résultats sur *huit caractères*, un indice prenant deux caractères sur ces huit. Par exemple, une matrice de variables PrixDeVente(4,4) donnera les variables Prix0101, Prix0102, etc. Attention : en vue de ce listing, Vx contrôle donc dès la compilation que ces noms sont non ambigus sur six caractères (pour un indice) ou quatre caractères (pour deux indices).

a) Section Let

Dans cette section, chaque ligne définit une constante. Une seule constante peut être définie par ligne et, comme en Pascal, une constante peut être définie à partir d'autres :

```
LET
m  = 3
m2 = m*m
```

b) Section Tables

La rubrique *Tables* sert à définir les tables de *constantes* (mais pas celles de *variables*). Une seule table peut être définie par ligne, avec au plus deux dimensions en Vx Student. Chaque table doit ensuite être initialisée dans une section *Data*, *Diskdata* ou *Assign*. Une constante non indicée peut être mise dans *Tables*, mais la section *Let* est préférable. Exemple :

```
TABLES
C(m)
P(n,m)
```

c) Section Data

Dans cette section, recommandée seulement pour les petites tables, une ligne sert à initialiser soit un vecteur, soit une ligne de matrice (dans le second cas, il faut donner les indices du premier élément à initialiser dans la ligne). Voici un exemple avec deux tables C(3) et P(2,3) :

```
DATA
C      = 9,15,5.5
P(1,1) = 7,9,8
P(2,1) = 5,8,7
```

On peut initialiser seulement une partie du tableau (P(1,2) = 9,8 est valide). Les éléments non définis sont mis à 0 par Xpress.

d) Section DiskData

Chaque ligne de cette section permet d'initialiser une table à partir d'un fichier texte, par exemple `p = MonFich.dat` (il n'y a plus de possibilité de chargement ligne par ligne comme dans *Data*). La table p doit être tapée entièrement dans *Monfich.dat*, ligne par ligne, avec des virgules entre les nombres sauf après le dernier de chaque ligne. L'extension *.dat* est recommandée car utilisée par défaut par Vx pour les données. Si on stocke plusieurs tableaux sur disque, il faudra (malheureusement) un fichier par tableau. Avec Vx Student, les noms de fichiers sont limités à huit caractères (hors suffixe *.dat*). Les fichiers de données doivent se trouver dans le même répertoire que le fichier modèle. Exemple :

```
DISKDATA
P = Monfich.dat
```

Avec le contenu suivant pour Monfich.dat :

```
7,9,8
5,8,7
```

e) Section Variables

Cette section est analogue à Tables, mais utilisée pour déclarer des variables. Chaque ligne contient une variable non indicée ou bien un tableau de variables à un ou deux indices. Exemple :

```
Variables
X
Y(m)
Z(n,m)
```

Rappel : comme les noms de contraintes, les noms de variables apparaîtront dans le listing de résultats et doivent être non ambigus sur huit caractères (si pas d'indice), six caractères (si un indice) ou quatre caractères (si deux indices).

f) Section Constraints

Cette section contient en fait la fonction objectif et les contraintes. *Toutes doivent avoir un nom* au choix terminé par "**:**". La fonction objectif est reconnaissable par un "$" à la fin et par l'absence de second membre ; elle peut se situer n'importe où au milieu des contraintes. On n'indique pas *Max* ou *Min* : le sens de l'optimisation est indiqué en utilisant le menu *Problem/Optimiser options*. Si la fonction objectif est absente, Xpress calcule *une* solution au système de contraintes.

```
CONSTRAINTS
Objectif: 3*X+Sum(i=1:n)Y(i)$
```

Les vraies contraintes sont des égalités ou inégalités linéaires : somme de termes de la forme A*X, avec une constante A et une variable X, et un second membre constant après les signes =, < ou >. Ces deux derniers signes signifient en fait >= et <=. Ces dernières formes sont également acceptées dans les dernières versions de Vx.

Une contrainte trop longue peut prendre plusieurs lignes, mais les lignes non terminées doivent avoir un "&" comme dernier caractère pour pouvoir poursuivre la contrainte sur la ligne suivante.

```
Longue: 3*X + &
        5*Y = 9
```

On peut laisser des expressions *constantes* à calculer, exemple `1/3*N*X` (N défini dans *Let*), mais ces calculs interviennent sur une seule variable à la fois.

Des combinaisons linéaires de variables sont possibles aussi dans le second membre, et on peut mettre la constante après la variable dans un produit : `2*X – Z < Y*5`.

Les contraintes de positivité sur les variables sont *implicites*, et les contraintes d'intégrité sur des variables doivent être mises dans une section *Bounds*. Les contraintes de bornes sur une variable doivent aussi être mises *de préférence* dans une section *Bounds* où elles ne sont pas comptées dans les 300 contraintes de Vx Student et sont gérées plus rapidement (algorithme du simplexe à variables bornées).

L'opérateur *Sum* permet de faire des combinaisons linéaires de variables indicées, par exemple `Sum(i=1:n)A(i)*X(i)`. *Sum* est suivi d'une expression de la forme `a(j)*x(j)`, où `x(j)` est la variable. Ainsi, on peut écrire `Sum(j=1:n)x(i)-Sum(j=1:n)y(i)`, mais pas `Sum(j=1:n)(x(i)-y(i))`. Pour les doubles sommes, on n'écrit pas des Sum de Sum, mais on met les plages de variation des deux indices dans la même somme, par exemple `Sum(i=1:m,j=1:n)X(i,j)`.

Une contrainte peut être générique (groupe de contraintes de même nature) ; son nom a alors une plage de variation d'indices, comme Sum. Exemple :

```
CapaDepot(i=1:NbDepots) :
```

Les indices sont des variables "muettes" et ne doivent pas être déclarées. Si une variable indicée comme `X(i)` apparaît dans une contrainte, `i` doit être défini devant, soit dans un Sum, soit dans une contrainte générique.

g) Section Bounds

Toutes les contraintes exprimant une borne inférieure ou supérieure sur une variable peuvent être déclarées dans une section *Bounds* (*bornes*), après les contraintes. *Dans ce cas, elles ne sont pas comptées dans le nombre limité de contraintes de Vx Student*. Voici un exemple dans lequel on indique que les variables X(3) à X(8) doivent valoir au plus 10 :

```
BOUNDS
X(i=3:8) < 10
```

On aurait pu évidemment mettre ces bornes dans la section *Constraints*, mais en leur donnant un nom et en consommant des contraintes sur le nombre limité autorisé par les versions bridées :

```
Borne(i=3:8): X(i) < 10
```

Attention, si X est une matrice de variable, mettre X(n,n) dans *Bounds* ne définit une borne que sur *une* variable. Pour mettre des bornes sur toutes les variables, il faut écrire :

```
X(i=1:n,j=1:n) < 10
```

Sans précision, une variable est considérée continue, positive ou nulle (≥ 0). Dans la section *Bounds*, on peut l'obliger à être entière (.ui. ou *unbounded integer*), entière bornée (avec $, *cf* exemple ci-dessous), en 0-1 (.bv. ou *binary variable*) ou non contrainte en signe (.fr. *free variable*). L'exemple ci-dessous présente X, une variable non indicée qui doit être entière, Y une variable non indicée en 0-1, et Z un vecteur de trois variables dont seules les deux dernières doivent prendre les valeurs 0, 1, 2 et 3 :

```
BOUNDS
```

```
X .ui.
Y .bv.
Z(i=2:3) $ 3
```

Autres particularités

a) Format sparse

La plupart des matrices de PL sont peu denses (*sparse* en anglais) : elles ont une majorité de 0. L'exemple typique est quand on veut coder un graphe, en stockant dans un fichier disque sa matrice d'adjacence *A*. On rappelle [Prins 1994] que pour un graphe à *n* sommets, la matrice *A* est $n \times n$, à éléments 0-1, et que $A_{ij} = 1$ si et seulement si l'arc (i,j) existe. Voici comment on définirait cette matrice :

```
LET
n = ...   ! Nombre de noeuds du graphe

TABLES
A(n,n)    ! Matrice d'adjacence (en 0-1)
DISKDATA
A = Matrice.dat
```

Pour un graphe à 20 sommets, on doit alors saisir 400 nombres (20 lignes de 20 nombres 0 ou 1). Or, par exemple, dans un réseau routier, chaque nœud a en moyenne 4 voisins, d'où 80 arcs seulement. Pour alléger la saisie, Xpress a un format *sparse* où on tape uniquement les éléments non nuls d'un tableau. Dans le modèle, la matrice est alors lue dans une section *Diskdata -s* (avec "s" comme sparse). Dans le fichier de données, on saisit seulement les éléments non nuls sous la forme *i*, *j*, *valeur* à raison d'un triplet par ligne (par exemple 3,7,1 pour A[3,7] = 1). Xpress initialise à 0 les éléments non saisis.

b) Filtrage des variables et contraintes

Le format *sparse* économise de la saisie mais ne change rien à la taille du PL généré (le § 3.3.5 décrit des éléments de syntaxe avancée et peut résoudre ce problème). Soit par exemple un problème de circulation de véhicules dans un réseau à *n* nœuds (c'est un problème de flot dans un graphe). Une matrice *Capa* donne la capacité de chaque arc du réseau en véhicules/heure. Elle sert aussi de matrice d'adjacence décrivant le réseau, en convenant que la liaison (arc) (*i*,*j*) n'existe pas si *Capa*(*i*,*j*) = 0.

Pour obtenir un modèle plus petit et donc plus rapide à résoudre, il est intéressant de définir des variables ou des contraintes seulement pour les arcs (*i*,*j*) existant réellement. Pour calculer le débit du réseau, on a par exemple besoin de variables Flux(*i*,*j*) donnant le flux sur chaque arc, et ce flux ne peut excéder la capacité de l'arc. L'opérateur de filtrage "|", qu'on pourrait lire "tel que", permet de préciser une *condition logique* pour générer seulement les variables et contraintes nécessaires :

```
VARIABLES
Flux(i=1:n,j=1:n | Capa(i,j) >= 0)   ! Flux pour les seuls arcs existants
```

```
CONSTRAINTS   ! Idem pour les contraintes de respect des capacites
Limite(i=1:n,j=1:n | Capacite(i,j) <> 0) : Flux(i,j) <= Capacite(i,j)
```

Le filtrage permet aussi de faire passer dans les versions bridées des problèmes qui auraient sinon un trop grand nombre de variables. Les opérateurs logiques sont les suivants, ceux du type *.op.* sont hérités du langage Fortran :

- =, = = ou *.eq*. (equal to) <> ou *.ne*. (not equal to) < ou *.lt*. (less than)
- > ou *.gt*. (greater than) <= ou *.le*. (less or equal) >= ou *.ge*. (greater or equal)
- *.and*. (et logique) *.or*. (ou logique) *.not*. (négation)

Exemple pour ignorer les contraintes de capacités sur les arcs d'origine 7 d'un graphe :

```
Limite(i=1:n,j=1:n| i<>7 .and. Capa(i,j) >= 0): Flux(i,j) < Capa(i,j)
```

c) Assign

Cette section est bien pratique pour calculer des données à partir d'autres données. Voici un exemple où on calcule un distancier à partir de deux tableaux X et Y de coordonnées de points, et un tableau des carrés des entiers de 1 à *n*. La section *Assign* intervient alors après l'initialisation de X et Y par *Data*. S'il n'y avait eu que TabCarres, on aurait pu mettre *Assign* aussitôt après *Tables*.

```
TABLES
X(n)
Y(n)
Dist(n,n)
TabCarres(n)

DATA
X = 0,0,2,2,3,3
Y = 2,3,2,3,2,3

ASSIGN
Dist(i=1:n,j=1:n) = (Abs(X(i)-X(j))^2 + Abs(Y(i)-Y(j))^2)^0.5
TabCarres(i=1:n)  = i*i
```

La fonction racine carrée n'existe pas, mais la fonction puissance ^ existe : ^0.5 est équivalent à la racine carré de 0.5.

3.3.6 Syntaxe avancée

En Xpress, on sait qu'on peut lire des tables à partir de fichiers en format *dense* ou *sparse*. Par exemple, une table *T*(*n*) peut être lue en format dense par *Diskdata*, ou en format sparse par *Diskdata -s*. Pour coder un grand graphe de *n* nœuds et *m* arcs, on peut donc définir une matrice d'adjacence *A*(*n*,*n*) et la lire en format sparse. Hélas, Xpress alloue réellement la matrice complète *A*(*n*,*n*), ce qui peut poser des problèmes de mémoire pour des grands graphes. On peut aussi définir une table *Arcs*(*m*,2), avec l'arc *k* reliant les nœuds *A*(*k*,1) et *A*(*k*,2), et la lire en format normal. Mais cela suppose de connaître exactement le nombre d'arcs *m*. Comment remédier à ces problèmes ?

Une table `T` est en Xpress *dense* par défaut, c'est à dire que tous les éléments sont alloués.

Il n'existe pas d'attribut pour spécifier qu'une table est dense. On peut aussi définir une table *éparse* avec l'attribut *-e* suivi du nombre maximal d'éléments susceptibles d'être chargés dans la table. Seuls les éléments chargés seront alloués, grâce à une technique de hashing, au prix d'une légère perte de performance. Ci-dessous, *T* est déclarée avec un indice pouvant atteindre *n*, mais au plus dix éléments seront chargés. Attention : les attributs *-d* et *-s* sont possibles pour une table mais indiquent un stockage en précision *single* ou *double* !

```
LET n = 20
TABLES
T(n) -e10
```

Qu'une table soit dense ou éparse, on peut la lire en format dense (*-d*) ou sparse (*-s*) avec un *Diskdata*. Attention : si on ne précise pas l'attribut du *Diskdata*, Xpress suppose *-d* pour une table dense, *-s* pour une table éparse.

a) Table dense, format de fichier dense

La table *T* est déclarée sans le *-e*10, et le *Diskdata* sans attribut ou avec *-s*. Le fichier doit contenir au plus *n* nombres, séparés par des virgules. Si le fichier a moins de *n* nombres, les derniers éléments du tableau sont mis à 0.

b) Table dense, format de fichier sparse

C'est le mode que nous utilisons souvent pour les graphes. Le fichier est déclaré par un *Diskdata -s* et doit définir un élément sur chaque ligne (indice et valeur). Là encore, les éléments manquants sont mis à 0. Comme nous l'avons dit, *T* est hélas allouée avec *n* éléments.

c) Table éparse, format de fichier dense

On déclare par exemple T(n) -e10 (on peut mettre un nom de constante ou une expression au lieu de 10). Cette déclaration ne consomme pas de mémoire. On doit alors préciser *Diskdata -d* (sinon c'est *-s* qui est supposé, la table n'étant pas dense). Le fichier doit contenir au plus dix nombres.

```
TABLES
T(n) -e10
DISKDATA -d
T = Monfich.dat
DISKDATA -o
Contenu.dat = T
VARIABLES
X(i=1:entries(T))
```

Supposons que le fichier contienne 7,0,15. Seuls *T*(1), *T*(2) et *T*(3) sont alloués et reçoivent 7, 0 et 15. Les autres éléments ne sont pas alloués, et y faire référence donne des résultats fantaisistes (erreur rarement détectée). Le *Diskdata -o* (comme output) est bien commode pour écrire le contenu de *T* dans un fichier. Pour un tableau épars comme *T*, seuls les éléments alloués sont écrits. On peut préciser *-os* ou *-od* pour un listing sparse ou dense. La fonction *entries* donne le nombre d'éléments réellement alloués et permet de définir ici les variables *X*(1), *X*(2) et *X*(3). On peut donc avec cette technique déclarer un graphe avec une liste d'arcs *Arcs* de taille minimale :

```
LET
```

```
n = 10
n2 = n*n
TABLES
Arcs(n2,2) -e n2
```

Le graphe peut avoir en théorie n^2 arcs. Comme cela ne coûte rien en mémoire pour une table éparse, on écrit que Arcs a n^2 lignes (uniquement pour que Xpress ne dise rien s'il voit des indices allant jusqu'à n^2), et qu'effectivement n^2 éléments puissent être alloués dans le pire des cas. Il ne reste plus qu'à taper les arcs dans un fichier qui sera lu par un *Diskdata -d*. Le numéro du dernier arc alloué en mémoire sera connu avec entries(Arcs/2) (attention, Arcs a ici deux éléments par ligne !).

d) Table éparse, format de fichier sparse

Considérons les déclarations suivantes :

```
LET
n=20
TABLES
T(n) -e10
DISKDATA -s
T = Monfich.dat
DISKDATA -o
Contenu.dat = T
VARIABLES
X(i=1:N|exist(T(i)))
```

Et supposons que Monfich.dat contienne :

```
2,17
4,18
6,0
```

Seuls T(2), T(4) et T(6) seront alloués. Le *Diskdata -o* va répertorier uniquement ces éléments. Contrairement au format de fichier dense, qui range les nombres lus consécutivement à partir de T(1), on a ici des "trous" dans la table. Pour éviter de générer des variables pour les éléments non alloués, on dispose de la fonction booléenne *exist*. Avec cette technique, on peut donc définir une matrice de graphe A(n,n) -e n2 et la lire en format sparse pour consommer uniquement la place nécessaire pour les éléments lus dans le fichier.

3.4 Les autres langages

3.4.1 Un exemple simple

Nous présentons ici la traduction d'un modèle mathématique simple dans plusieurs langages de modélisation. Nous avons donc choisi un petit problème d'affectation simple (celui du paragraphe 3.3) et allons le décliner dans les langages les plus connus. Parmi les produits phares disposant d'un langage de modélisation, nous avons sélectionné Xpress (Dash Associates) pour rappel (version 3 et nouvelle version), AMPL, GAMS (Gams Development Corp.), OPL Studio (ILOG), Lingo (Lindo Systems) et MPL (Maximal Software).

Le but de ce paragraphe n'est pas de montrer quel langage (ou logiciel) est le meilleur, mais d'illustrer la facilité d'écriture des langages de modélisation actuels et la similitude avec l'écriture mathématique pour certains et informatique pour d'autres. Les divers langages sont si proches qu'il est très facile d'adapter les modèles Xpress de ce livre, au cas où vous utiliseriez un autre logiciel.

Problème : trois étudiants doivent choisir un projet parmi trois proposés. Pour satisfaire tout le monde, le professeur demande à chacun de donner une note de préférence à chaque projet (de 3 à 1 : 3 pour le projet préféré, 1 pour le moins préféré). Le premier étudiant donne les notes 1, 2 et 3 aux projets 1, 2 et 3 (il voudrait le projet 3 en priorité), le deuxième, les notes 3, 2, 1, le troisième les notes 3, 1, 2. Pour choisir à qui attribuer quel projet, le professeur écrit un modèle de programmation linéaire maximisant la satisfaction totale des étudiants. Soit n le nombre d'étudiants et de projets, p_{ij} la note donnée par l'étudiant i au projet j et x_{ij}, une variable binaire valant 1 si l'étudiant i récupère le projet j. Le programme linéaire s'écrit de la manière suivante :

$$(1) \quad \text{Max} \sum_{i=1}^{n}\sum_{j=1}^{n} p_{ij}\, x_{ij}$$

$$(2) \quad \forall j = 1\ldots n : \sum_{i=1}^{n} x_{ij} = 1$$

$$(3) \quad \forall i = 1\ldots n : \sum_{j=1}^{n} x_{ij} = 1$$

$$(4) \quad \forall i = 1\ldots n, \forall j = 1\ldots n : x_{ij} \in \{0,1\}$$

La fonction objectif (1) maximise la somme des préférences pour les affectations choisies, la contrainte (2) spécifie qu'un seul étudiant est affecté par projet, la contrainte (3) que chaque projet est pris par un étudiant. La contrainte (4) donne le domaine de variation des variables. Ce modèle a automatiquement des variables entières à l'optimum (voir chapitre 2), c'est pourquoi la contrainte (4) ne sera pas reprise dans les modèles ci-dessous, mais remplacée par la contrainte (4').

$$(4') \quad \forall i = 1\ldots n, \forall j = 1\ldots n : x_{ij} \geq 0$$

3.4.2 Visual Xpress

Avec les connaissances vues au paragraphe précédent, on peut transcrire facilement le modèle mathématique d'affectation en modèle Xpress ci-dessous. La matrice des notes données par les étudiants aux projets est lue dans un fichier texte. Le fichier texte contenant le modèle a toujours l'extension *.mod* (Visual Xpress 3 pour Windows ou versions Unix). En général, on donne l'extension *.dat* à tous les fichiers de données.

```
MODEL Affect

LET

n = 3    ! Nombre de projets/d'etudiants
```

```
TABLES

p(n,n)   ! Note de preference des eleves pour les projets

VARIABLES

x(n,n)   ! Variables binaire d'affectation des projets

DISKDATA

p = Monfich.dat

CONSTRAINTS ! Maximiser la satisfaction generale

Satisfaction        : Sum(i=1:n,j=1:n)p(i,j)*x(i,j)$
UnEtudParProj(j=1:n): Sum(i=1:n)x(i,j) = 1
UnProjParEtud(i=1:n): Sum(j=1:n)x(i,j) = 1

END
```

Le fichier des données *MonFich.dat* est détaillé ci-dessous.

```
1,2,3
3,2,1
3,1,2
```

3.4.3 Xpress-IVE

La nouvelle syntaxe du langage de modélisation (appelé MOSEL) est plus riche et plus complète que celle de Visual Xpress. Elle ressemble fort à celle d'ILOG OPL Studio. Comme dans ce dernier, on peut écrire des procédures et des fonctions que l'on pourra appeler durant l'exécution et la résolution d'un modèle. Voici la traduction de l'exemple précédent.

```
model "Affect"
 uses "mmxprs"

 declarations
  eleves = 1..3                      ! Ensemble des eleves
  projets = 1..3                     ! Ensemble des projets

  p: array(eleves,projets) of integer ! note de preference
                                      ! des eleves pour les projets
  x: array(eleves,projets) of mpvar   ! variables binaires d'affectation
 end-declarations

 initializations from 'Monfich.dat'
  p
 end-initializations

! Objectif: maximiser la satisfaction globale
 Satisfaction:= sum(i in eleves, j in projets) p(i,j)*x(i,j)
```

```
! Un etudiant par projet
 forall(j in projets) sum(i in eleves) x(i,j) = 1

! Un projet par etudiant
 forall(i in eleves) sum(j in projets) x(i,j) = 1

! resolution de l'exemple
 maximize(Satisfaction)

end-model
```

Le fichier de données *Monfich.dat* est détaillé ci-dessous.

```
p :[ 1 2 3
     3 2 1
     3 1 2]
```

La syntaxe est constituée de blocs. Les déclarations de données et variables sont contenues dans un premier bloc, puis un second bloc d'initialisation sert à charger les données externes depuis un fichier. A l'instar des langages informatiques, les données et les variables sont typées. Des mots-clefs comme *array*, *integer* ou *mpvar* sont utilisés. Comme dans le langage de programmation pascal, on peut voir des appels à des modules externes par la commande `uses "mmxprs"`. Ceci permet de ne charger que les modules nécessaires, mais impose de connaître à l'avance les modules auxquels on devra faire appel. Le sens de l'optimisation est défini directement dans le programme. Le fichier de données a une syntaxe particulière qui ne permet pas l'importation automatique, mais les nombreuses fonctionnalités du langage permettent d'importer des données depuis des bases de données ou des tableurs connus. Une fonction simple de lecture comme en pascal permet aussi de créer ses propres formats de fichier aussi bien à la lecture qu'à l'écriture.

3.4.4 AMPL

Très élégant, AMPL offre de puissantes possibilités d'indexation. Il devait absolument figurer dans ce tour d'horizon des langages de modélisation. Sa syntaxe est très intuitive et facile à apprendre. Les commentaires débutent par un #. On peut résoudre ce modèle interactivement sur le site *http://www.ampl.com*, à la rubrique *Try it!*

```
param n := 3;           # nombre d'etudiants/projets
param p{1..n,1..n} > 0; # notes des etudiants aux projets
var x{1..n,1..n} >= 0;  # x[i,j] = 1 si l'etudiant i a le projet j

maximize Satisfaction:
  sum{i in 1..n, j in 1..n} p[i,j]*x[i,j];

subject to UnEtudParProjet {j in 1..n}:
  sum{i in 1..n} x[i,j] = 1;
subject to UnProjParEtud {i in 1..n}:
  sum{j in 1..n} x[i,j] = 1;
```

Le fichier de données est écrit de la manière suivante :

```
Param p : 1  2  3 :=
      1   1  2  3
      2   3  2  1
      3   3  1  2
```

3.4.5 GAMS

GAMS permet d'écrire des modèles éventuellement non linéaires. Une section *Sets* déclare les indices et leur domaine de variation. On peut noter une particularité de GAMS qui consiste à mettre la valeur de la fonction objectif dans une variable libre (ici z). Cette variable est ensuite utilisée pour la résolution (deux dernières lignes du modèle).

```
$include "MonFich.gms"
sets i 'indice des etudiants' /1*3/
     j 'indice des projets'   /1*3/ ;

variable z 'valeur de la fonction objectif' ;
positive variable x(i,j) 'affectation' ;

equations
  Satisfaction
  UnEtudParProj
  UnProjParEtud ;

Satisfaction     .. z =e= Sum((i,j), p(i,j)*x(i,j));
UnEtudParProj(j) .. Sum(i, x(i,j)) =e= 1;
UnProjParEtud(i) .. Sum(j, x(i,j)) =e= 1;

model affect /all/ ;
solve affect maximizing z using lp ;
```

Le fichier *MonFich.gms* est écrit de la manière suivante :

```
table p(i,j) 'tableau des notes des etudiants aux projets'
      1   2   3
  1   1   2   3
  2   3   2   1
  3   3   1   2
```

3.4.6 ILOG OPL Studio

ILOG OPL Studio fournit une interface graphique très évoluée et de nombreuses fonctionnalités. La syntaxe est très proche du langage mathématique que l'on énonce oralement en lisant des équations (mais en anglais). Cette syntaxe a pour origine AMPL, c'est pourquoi on retrouve tant de similitudes entre les deux langages. La notion de projet associe un fichier modèle à un fichier de données sans avoir à nommer le fichier de données dans le fichier modèle. Tout ce qui est noté avec **…** est lu directement dans le fichier attaché.

```
int n = 3;
range Etudiants 1..n;
range Projets   1..n;
int p[Etudiants,Projets] = ...;

var float+ x[Etudiants,Projets];

maximize
   sum(i in Etudiants, j in Projets) p[i,j]*x[i,j]
subject to {
      forall(j in Projets)
           sum(i in Etudiants) x[i,j] = 1;
      forall(i in Etudiants)
           sum(j in Projets) x[i,j] = 1;
};
```

Le fichier *Affect.dat* reprend le nom de la constante p qui contient les notes attribuées par les étudiants aux trois projets.

```
p = [
 [ 1, 2, 3],
 [ 3, 2, 1],
 [ 3, 1, 2]];
```

3.4.7 Lingo

Lingo est très utilisé outre-Atlantique. Sa syntaxe très structurée est reconnaissable aux arrobas @ qui signalent les opérateurs et fonctions mathématiques du langage. Comme GAMS, il permet aussi l'écriture de programmes non linéaires. A part les lignes signalant un début ou une fin de section (comme DATA: et ENDDATA), toute ligne Lingo, y compris de commentaire, doit se terminer par un point-virgule : l'absence de ce signe peut donner des erreurs difficiles à détecter, voire des résultats faux.

```
MODEL:
! Probleme d'affectation de n etudiants a n projets;
DATA:
   n = 3;
ENDDATA

! Affectation des n etudiants aux n projets;
SETS:
   Etudiants/1..n/;
   Projets/1..n/;
   Affectations(Etudiants,projets): p, x;
ENDSETS

DATA:
   p = @FILE(affect.ldt);
ENDDATA
```

```
[Satisfaction]MAX = @SUM(Affectations(i,j): p(i,j)*x(i,j));
! Un etudiant par projet;
@FOR(Projets(j):[UnEtudParProj] @SUM(Etudiants(i): x(i,j)) = 1);
! Un projet par etudiant;
@FOR(Eleves(i) :[UnProjParEtud] @SUM(Projets(j): x(i,j)) = 1);

END
```

Le fichier *affect.ldt* est des plus simples : une ligne par ligne de la matrice et pas de séparateurs particuliers.

```
1 2 3
3 2 1
3 1 2
```

3.4.8 MPL for Windows

Le dernier langage présenté ici est MPL for Windows de Maximal Software. Encore une fois, sa syntaxe est très proche des autres langages de modélisation. Les commentaires sont placés entre accolades.

```
TITLE
  Affect ;

DATA
  n := 3 ; {nombre d'etudiants/projets}
  p := DATAFILE("affect.dat") ;

INDEX
  Etudiants := 1..n ;
  Projets   := 1..n ;

VARIABLES
  x[Etudiants,Projets] ;

MODEL
  MAX Satisfaction = SUM(Etudiants,Projets : p*x) ;

SUBJECT TO
  UnEtudParProj[Projets] :
    SUM(Etudiants : x) = 1 ;
UnProjParEtud[Etudiants] :
    SUM(Projets : x) = 1 ;

END
```

Le fichier de chargement des notes est identique à celui de Xpress, les virgules sont facultatives (format Lingo). Il est décrit simplement de la manière suivante :

```
1,2,3
3,2,1
3,1,2
```

3.5 Présentation des chapitres d'applications

Les chapitres suivants (4 à 14) contiennent des problèmes de programmation linéaire classés par domaines d'application. Ils ont tous la même structure. Une introduction présente le domaine ou le thème étudié, une série d'environ six applications est présentée, et un paragraphe "Références et compléments" permet au lecteur intéressé de trouver des compléments d'information ou des points d'entrée pour approfondir le sujet. Chaque application est décrite selon le plan suivant : énoncé d'un problème réaliste accompagné de ses données, modélisation mathématique pas à pas (indépendante du logiciel), traduction du modèle mathématique dans le langage Xpress, présentation et discussion des résultats.

Le chapitre 4 rassemble des problèmes de mélange ou de séparation de composés, rencontrés dans les industries minières et de process. Ces problèmes, qui font rarement appel à des variables entières, sont les plus simples et conviennent bien aux débutants. Les deux chapitres suivants sont consacrés à deux autres groupes importants d'applications industrielles : les problèmes d'ordonnancement (chapitre 5) et les problèmes de planification de production (chapitre 6).

Dans beaucoup d'applications, il faut remplir un espace limité (caisses, cales de navires, disques d'ordinateurs) avec des objets ou, au contraire, découper des matériaux pour en extraire des motifs tout en minimisant les chutes. Ce genre de problème fait l'objet du chapitre 7 sur la découpe et le conditionnement (*cutting and packing* en anglais).

Les problèmes de flux, au sens large, soulèvent de très nombreux problèmes intéressants en optimisation. Nous avons choisi de les aborder au travers de trois domaines représentatifs : le transport terrestre au chapitre 8, le transport aérien au chapitre 9, et enfin le monde foisonnant des télécommunications au chapitre 10. Plusieurs modèles sont transposables entre ces trois chapitres, mais on y trouve aussi des applications plus spécifiques, comme les tournées de véhicules en transport routier, les problèmes de correspondance d'avions en transport aérien, et tout ce qui concerne le dimensionnement et la fiabilité des réseaux en télécommunications.

Pour éviter de nous adresser seulement aux scientifiques, industriels et logisticiens, nous avons tenu à présenter des domaines d'application plus récents ou moins connus de la programmation linéaire. Le chapitre 11 est ainsi consacré aux applications en économie et en optimisation financière. Les problèmes d'emploi du temps et de gestion du personnel font l'objet du chapitre 12. La programmation linéaire peut également rendre de grands services aux collectivités locales, aux administrations et au secteur public en général : le chapitre 13 présente six exemples. Enfin, pour décompresser un peu et finir sur une note plus légère, le dernier chapitre (le 14) présente divers jeux et casse-tête.

Les noms des modèles en langage Xpress suivent les conventions suivantes : huit caractères maximum, le premier caractère est une lettre correspondant au chapitre (A à K pour les chapitres 4 à 14), le deuxième est un chiffre donnant le numéro séquentiel du problème dans son chapitre. Par exemple, un problème dont le nom commence par B2 est le

deuxième du chapitre 5.

L'annexe 1 contient une liste récapitulative des problèmes ainsi qu'une classification selon le modèle théorique classique sous-jacent (par exemple problèmes de flots, problèmes de recouvrement). Elle permet un aiguillage plus rapide vers les différents problèmes. Chaque modèle peut être testé sur un PC grâce à la version limitée de Xpress et aux répertoires de fichiers modèles et de données fournis sur le CD-Rom joint au livre. Le contenu du CD-Rom est rappelé dans l'annexe 2.

3.6 Mise en garde

Les modèles des chapitres suivants ont été développés avec la version Student, gratuite mais limitée, de Visual Xpress 3.0. Les résultats indiqués sont en général ceux obtenus avec cette version. Quelques problèmes ont dû être résolus avec la version *Hyper*, car ils nécessitent plus de cinquante variables entières ou plus de deux indices par variable.

La version plus récente de Xpress-IVE peut être obtenue sur *http://www.dash.co.uk*. ou sur *http://www.artelys.fr*. L'interface utilisateur est un peu modifiée mais reste globalement identique. Dans cette nouvelle version, le texte Xpress des modèles n'est plus valable mais un traducteur automatique sera prochainement disponible.

Beaucoup de problèmes du livre ont plusieurs solutions optimales. Ainsi par exemple selon l'ordre des contraintes ou le solveur utilisé la solution optimale trouvée pourra être différente de celle que nous indiquons (valeurs des variables différentes, mais même valeur de la fonction objectif).

Les problèmes du livre ne sont pas de grande taille : cela n'est pas utile pour la partie modélisation, et il faut pouvoir les résoudre avec la version limitée de Xpress. En tout cas, ils sont assez grands pour ne pas être traitables à la main. Si vous décidez d'acquérir une version industrielle d'un logiciel de programmation linéaire, il vous sera possible de traiter des PL de grande taille dans deux cas : problèmes à variables continues (mélanges par exemple) et problèmes à matrice totalement unimodulaire comme les problèmes de flots (voir chapitre 2).

Pour les autres problèmes, à variables entières, vous devez prendre des précautions. Il s'agit en général de problèmes NP-difficiles. Il est hors de question ici de détailler ce qu'est un problème NP-difficile, voir par exemple [Garey 1979] pour en savoir plus. Disons simplement que de tels problèmes n'ont pas, à l'heure actuelle, d'algorithmes polynomiaux pour les résoudre. Un algorithme polynomial effectue un nombre d'opérations croissant comme un polynôme de la taille des données, c'est par exemple le cas de l'algorithme de Dijkstra, qui calcule un plus court chemin en $O(n^2)$ dans un graphe à n nœuds.

Les problèmes de recouvrement et du voyageur de commerce, abordés dans ce livre, sont des exemples de problèmes NP-difficiles, quelle que soit la façon dont ils sont modélisés. Les approches basées sur la programmation linéaire en nombres entiers n'échappent pas à cette déplaisante propriété : même si les algorithmes de recherche arborescente des solveurs de PLNE n'énumérent qu'une faible partie de l'espace des solutions réalisables, le temps de calcul croît exponentiellement avec la taille des données.

Bien que la résolution puisse être très rapide sur certains jeux de données, ne vous attendez donc pas à résoudre optimalement et rapidement des cas de grande taille, même avec un solveur industriel. En revanche, les méthodes arborescentes vous donneront généralement de très bonnes solutions, bien que non optimales : elles devront être interrompues avant la fin, la visite complète de l'arborescence prenant trop de temps.

3.7 Références et compléments

A part les manuels d'utilisation joints aux logiciels, il existe des livres en librairie pour présenter et enseigner certains des langages mentionnés dans ce chapitre. Tous ces livres sont accompagnés d'une version limitée du logiciel. Il existe ainsi un livre sur GAMS, écrit par ses concepteurs, Brooke, Kendrick et Meeraus [Brooke 1988], mais il s'agit plutôt d'un manuel d'utilisation.

Les deux livres suivants sont plus intéressants, car ils incluent des exemples de modélisation. Fourer, Gay et Kernighan ont écrit un livre très clair sur AMPL, logiciel qu'ils ont conçu dans les laboratoires de recherche d'ATT [Fourer 1999a]. Schrage est l'auteur d'un livre sur Lindo qui fourmille d'exemples [Schrage 1997]. Lindo est un solveur, mais il permet de lire des PL sous un forme très parlante, bien que non générique. On peut par exemple entrer une contrainte sous la forme 3*X1-2*X2 <= 17. Il est actuellement caché comme solveur dans le logiciel Lingo, plus récent, qui inclut un modeleur de haut niveau. L'intérêt du livre de Schrage est que tous les exemples Lindo sont compris par Lingo, logiciel dont une version gratuite est téléchargeable sur *http://www.lindo.com*.

Concernant l'optique choisie par ce livre, la modélisation d'applications choisies, il existe un ouvrage anglais de la même veine, écrit par H.P. Williams [Williams 1993]. Au moment où nous écrivons ces lignes, il est épuisé, mais il sera certainement réédité. Il contient vingt études de cas, idéales pour des projets d'étudiant. Les résultats fournis sont également obtenus avec Xpress mais, à part quelques exceptions, l'auteur ne donne que le modèle mathématique, sans la traduction en Xpress. Le livre est vendu sans logiciel, et il ne contient aucune partie sur Xpress et sur les logiciels d'optimisation en général.

```
                  Sum(i=1:n)CS*XS(i) + &
                  Sum(i=1:n)CSt*S(i)   $

Periode1       : XN(1) + XS(1) + Sinit  = D(1) + S(1)
Periode(i=2:n) : XN(i) + XS(i) + S(i-1) = D(i) + S(i)

BOUNDS

XN(i=1:n) < Capa

END
```

On peut en fait condenser les deux contraintes `Periode1` et `Periode(i=1:n)` en une seule équation de la façon suivante, où `Sum(|i>1)S(i-1)` ne fait apparaître le terme `S(i-1)` dans la contrainte que si `i>1`, et `Sum(|i=1)Sinit` ne fait apparaître `Sinit` que lorsque `i=1`.

```
Periode(i=1:n) : XN(i)+XS(i)+Sum(|i>1)S(i-1)+Sum(|i=1)SInit=D(i)+S(i)
```

6.2.4 Résultats

La solution optimale consiste à produire en heures normales les quantités données au tableau 6.2 (en milliers d'unités). A ces quantités s'ajoutent 10 000 bicyclettes fabriquées en heures supplémentaires au mois de juin, 15 000 au mois de juillet et 15 000 au mois d'août. Il faut alors stocker 3 000 bicyclettes au mois d'avril. Ces bicyclettes seront utilisées pour satisfaire la demande du mois suivant. Les coûts de fabrication et de stockage s'élèvent à 45 590 000 F.

Tableau 6.2 – Quantités à produire en heures normales en milliers d'unités

Jan	Fév	Mars	Avril	Mai	Juin	Juill	Août	Sept	Oct	Nov	Déc
28	15	15	28	30	30	30	30	26	14	25	30

6.3 Production de verres

6.3.1 Problème

Une entreprise du nord de la France concentre son activité sur la production de verres pour la table. Elle propose six modèles différents (V1 à V6), produits par lots de 1 000 verres, et souhaite planifier sa production sur un horizon de 12 semaines. Les lots peuvent être incomplets (moins de 1 000 verres). La demande en nombre de lots de 1 000 verres pour les 12 semaines à venir et pour chacun des modèles est connue et reportée au tableau 6.3.

(1) $\text{Min} \sum_{j=1}^{m} y_j$

(2) $\forall i = 1 \ldots n : \sum_{j=1}^{m} x_{ij} = 1$

(3) $\forall j = 1 \ldots m : \sum_{i=1}^{n} a_i x_{ij} \leq b.y_j$

(4) $\forall i = 1 \ldots n, \forall j = 1 \ldots m : x_{ij} \in \{0,1\}$

(5) $\forall j = 1 \ldots m : y_j \in \{0,1\}$

Voici une autre modélisation utilisant moins de variables : nous conservons les variables booléennes x_{ij} du modèle précédent (mais pas les variables y_j) et utilisons une variable supplémentaire *NbU* égale au nombre de disquettes utilisées. On obtient le second programme linéaire en nombres entiers suivant.

(6) Min *NbU*

(7) $\forall i = 1 \ldots n : NbU \geq \sum_{j=1}^{m} j.x_{ij}$

(8) $\forall i = 1 \ldots n : \sum_{j=1}^{n} x_{ij} = 1$

(9) $\forall j = 1 \ldots m : \sum_{i=1}^{n} a_i x_{ij} \leq b$

(10) $\forall i = 1 \ldots n, \forall j = 1 \ldots m : x_{ij} \in \{0,1\}$

(11) $NbU \geq 0$

La fonction objectif s'exprime tout simplement par la contrainte (6). On suppose que les disquettes sont remplies à partir de *j=1*. Grâce aux contraintes (2) ou (8), la disquette contenant le fichier *i* a un indice *k* calculable par la relation (12).

(12) $k = \sum_{j=1}^{m} j.x_{ij}$

Il faut que *NbU* soit supérieur au plus grand indice de disquette utilisée. D'où la contrainte (7). La contrainte (8) indiquant qu'un fichier doit être stocké sur une seule disquette reste identique à la contrainte (2) de la modélisation précédente. Le respect des capacités des disquettes est traduit par la contrainte (9).

Enfin, toutes les variables x_{ij} sont binaires et *NbU* doit être positif (inutile de préciser qu'il s'agit d'un entier car cette contrainte est automatiquement vérifiée à l'optimum). Dans ce modèle, la minimisation va *écraser NbU* et stocker les fichiers sur les disquettes de telle

7.7 Découpes de barres d'acier

7.7.1 Problème

L'entreprise SchooldDesk fabrique des bureaux pour les écoles maternelles et primaires, les collèges et les lycées. Les pieds de ces bureaux ont tous le même diamètre, mais sont de longueurs différentes : 40 cm pour les plus petits, 60 cm pour les moyens, 70 cm pour les grands. Ils sont découpés dans des barres d'acier de 1,50 ou 2 m de longueur. Cette entreprise reçoit une commande de 108 petits bureaux, 125 moyens et 100 grands. Comment satisfaire cette commande en minimisant les chutes ?

7.7.2 Modélisation

Ce problème est assez proche du problème précédent de découpe de plaques de tôle. La modélisation exploite le fait qu'il existe un nombre réduit de plans de découpes *(patterns)* des barres d'acier. La barre de longueur de 1,50 m peut par exemple être découpée en deux pieds de 70 cm. Il reste alors 10 cm de chute qui ne peuvent être utilisés pour découper d'autres types de pieds. On peut aussi découper cette barre en un pied de 60 cm et un autre de 70 cm. Il reste alors 20 cm de chute. Le tableau 7.8 résume les différentes possibilités.

Tableau 7.8 – Plans de découpes possibles pour chaque type de barre

	Numéro de plan de découpe	Pieds de type 1 (40 cm)	Pieds de type 2 (60 cm)	Pieds de type 3 (70 cm)	Chute (en cm)
Barres de type 1 (1,50 m)	1	0	0	2	10
	2	0	1	1	20
	3	2	0	1	0
	4	0	2	0	30
	5	2	1	0	10
	6	3	0	0	30
Barres de type 2 (2 m)	7	0	1	2	0
	8	0	2	1	10
	9	1	0	2	20
	10	3	0	1	10
	11	0	3	0	20
	12	5	0	0	0
	13	1	1	1	30
	14	2	2	0	0
	15	3	1	0	20

Notons p le nombre de types de pieds différents à fabriquer, b le nombre de types de barres d'acier dans lesquelles les pieds sont découpés, n le nombre de plans de découpe. Soit x_i le nombre de barres découpées suivant le plan de découpe i. L'objectif est de minimiser les

chutes, c'est-à-dire de minimiser la différence entre la longueur totale de barres découpées et la longueur totale des pieds commandés.

Notons n_k le nombre de plans de découpe possibles des barres de type k et L_k leur longueur. Si D_j et H_j sont respectivement la demande et la hauteur de chaque pied j, les deux premiers termes de la fonction (1) représentent la longueur totale de barres découpées, le dernier terme la longueur totale de pieds commandés.

$$(1) \quad \text{Min} \sum_{i=1}^{n_1} L_1 x_i + \sum_{i=n_1+1}^{n} L_2 x_i - \sum_{j=1}^{p} D_j H_j$$

$$(2) \quad \forall j = 1 \ldots p : \sum_{i=1}^{n} a_{ij} x_i \geq D_j$$

$$(3) \quad \forall i = 1 \ldots n : x_i \in \mathbb{N}$$

Dans cette fonction, le terme correspondant à la somme des longueurs de pieds commandés peut être omis, car il s'agit d'une constante qui n'intervient pas dans la minimisation. La contrainte (2), où a_{ij} est le nombre de pieds de type j contenus dans le plan de découpe i, est la contrainte de satisfaction de la demande. La contrainte (3) impose que les variables x_i soient positives et entières.

7.7.3 Modèle Xpress

Le modèle *D6Barres* donne la traduction Xpress de ce programme mathématique. Cette traduction est directe. Attention cependant aux quantités commandées : pour fabriquer un bureau, il faut quatre pieds. Les demandes pour chaque type de pied sont donc respectivement 432, 500 et 400.

```
MODEL D6Barres

LET

p  = 3          ! Nombre de types de pieds
b  = 2          ! Nombre de types de barres d'acier
n1 = 6          ! Nombre de plans de decoupe des barres de 1,5 m
n2 = 9          ! Nombre de plans de decoupe des barres de 2 m
n  = n1+n2      ! Nombre total de plans de decoupe

TABLES

A(n,p)          ! A(i,j) = nombre de pieds de type j contenus dans
                ! le plan de decoupe i
D(p)            ! Quantites de chaque type de pied demandees
H(p)            ! Hauteur de chaque type de pied
L(b)            ! Longueur de chaque type de barre


DISKDATA

A  = D6A.dat
```

```
D  = D6Demand.dat
H  = D6Hauteu.dat
L  = D6Longue.dat

VARIABLES

X(n)           ! Quantites de barres decoupees pour chaque configuration

CONSTRAINTS    ! Minimiser (surface decoupee - surface utilisee)

Chutes         : Sum(i=1:n1)L(1)*X(i)+Sum(i=n1+1:n)L(2)*X(i)&
                 -Sum(j=1:p)D(j)*H(j)$
Demande(j=1:p) : Sum(i=1:n)A(i,j)*X(i) > D(j)

BOUNDS

X(i=1:n) .ui.

END
```

7.7.4 Résultats

A l'optimum, il faut découper une barre de 1,50 m suivant le plan de découpe 1, 100 suivant le plan de découpe 3, une suivant le plan de découpe 5, ainsi que 99 barres de 2 m suivant le plan de découpe 7, 100 suivant le plan de découpe 8, 6 suivant le plan de découpe 12 et 100 suivant le plan de découpe 14. La chute est alors de 1020 cm et il se trouve dans cet exemple que le nombre de chaque type de pied découpé correspond exactement à la demande.

7.8 Références et compléments

Tous les problèmes de ce chapitre sont NP-difficiles, et la programmation linéaire ne peut traiter que des cas de taille modeste. Le problème de chargement de wagons du § 7.2 est aussi connu comme un problème d'ordonnancement de n tâches non fragmentables sur m machines équivalentes, appelé *problème des m processeurs*. Minimiser la charge maximale des wagons correspond alors à minimiser la durée totale de l'ordonnancement. La formulation par PL, peu structurée, dépasse difficilement trois machines et trente tâches. Pour deux machines, une méthode de programmation dynamique (sorte d'optimisation récursive) de complexité $O(nB)$ peut résoudre des problèmes avec cent objets [Martello 1990].

Il existe des méthodes arborescentes spécialisées convenant jusqu'à cent tâches [Ho 1995]. Quand les méthodes optimales prennent trop de temps, on peut trouver de bonnes solutions avec l'heuristique *LPT* (Longest Processing Time), qui place à chaque itération, sur la machine la moins chargée, l'objet libre de plus grand poids. Le ratio de la solution *LPT* à la solution optimale n'est que de $4/3-1/(3m)$ [Graham 1969]

Les problèmes de chargements de péniche du § 7.3 sont des *problèmes de sac à dos* (*knapsack problems*), reconnaissables à leur unique contrainte de capacité. Le sac à dos en variables fractionnaires est facile : il suffit de remplir le contenant dans l'ordre décroissant des coûts par unité de taille. Les sacs à dos en variables entières sont NP-difficiles, mais des

cas de taille respectable peuvent être traités par programmation dynamique ou par recherches arborescentes [Syslo 1983] [Martello 1979].

Le problème de chargement de réservoirs (*loading problem*) est décrit dans un livre de Christofides *et al*. [Christofides 1979a]. Une méthode arborescente y est présentée pour les instances de grande taille (35 liquides, 70 réservoirs), ainsi que des algorithmes pour le cas dynamique comprenant des suites de chargements et déchargements.

Le problème de sauvegarde de fichiers sur disquettes du § 7.5 est un problème dit de *bin-packing* dans lequel on cherche à répartir n objets i de poids a_i dans un nombre minimal de boîtes parmi m disponibles, chacune de capacité b. Ce problème est NP-difficile. Il est généralement résolu par des heuristiques. Le lecteur intéressé est renvoyé à l'ouvrage de Coffman [Coffman 1996] qui contient un résumé de plus d'une centaine de références et à celui de Martello [Martello 1990].

Les problèmes de découpe comme ceux du § 7.6 et du § 7.7 sont très combinatoires et contiennent généralement de très nombreuses variables. Les contraintes sont très variées, comme par exemple l'exigence de couper bord à bord ou coupes guillotine (*guillotine cuts*).

Parmi les méthodes optimales, il existe des méthodes arborescentes, voir par exemple [Hifi 1996]. Une autre méthode exacte appelée *génération de colonnes* a été proposée par Gilmore et Gomory [Gilmore 1961][Gilmore 1963] pour traiter des problèmes à une dimension. Elle consiste à résoudre d'abord un problème comprenant un sous-ensemble très réduit des colonnes du programme linéaire complet, ce dernier pouvant être impossible à générer entièrement. Des colonnes prometteuses sont ensuite ajoutées progressivement. Quand la méthode marche bien, l'optimum est trouvé après avoir généré une faible partie des colonnes du modèle complet.

Les cas de grande taille doivent faire appel à des métaheuristiques comme la méthode tabou [Lodi 1999] ou les algorithmes génétiques [Jakobs 1996]. Des motifs équivalents en termes de chutes peuvent avoir des coûts de découpe très différents. Voir Chu pour des méthodes minimisant le coût [Chu 1999]. La formulation du § 7.6 est un *problème de recouvrement*, comme le problème de localisation de relais de téléphonie mobile du chapitre 10. Elle est assez efficace et peut gérer une centaine de motifs, mais ce nombre est encore faible par rapport au nombre énorme de motifs quand les rectangles sont petits par rapport aux plaques-mères. Sweeney *et al.* proposent une bibliographie très complète comprenant plus de 400 références sur les problèmes de chargement et découpe [Sweeney 1992].

14.4.4 Résultats

Après résolution, nous obtenons la solution donnée au tableau 14.2.

Tableau 14.2 – Solution du problème 14.4

Prénom	Nationalité	Exposé programmé le
Arabinda	Indien	9 août
Éric	Français	7 août
Hitoshi	Japonais	10 août
Michael	Américain	11 août
Zhicheng	Chinois	8 août

14.5 Grille et jetons

14.5.1 Problème

Partant de la grille carrée de taille 4×4 recouverte par seize jetons de la figure 14.1, comment enlever six jetons en laissant un nombre pair de jetons dans chaque ligne et chaque colonne de la grille ?

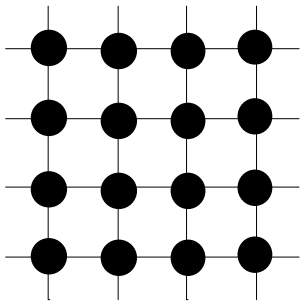

Figure 14.1 – Grille de départ

14.5.2 Modélisation

La constante n représente le nombre de jetons maximal que l'on peut placer en colonne ou en ligne. P désigne le nombre de jetons à laisser sur la grille. Une variable binaire x_{ij} est fixée à 1 si la case (i,j) est couverte par un jeton, 0 sinon. Sur la grille finale, P jetons doivent être positionnés, contrainte (1).

$$(1) \quad \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} = P$$

Pour vérifier qu'il y a un nombre pair de jetons par ligne ou par colonne, deux types de variables qui représentent la demi-somme des jetons sont nécessaires. NL_i (resp. NC_i)

représente la moitié de la somme des jetons de la ligne *i* (resp. de la colonne *i*). Pour être sûr d'obtenir un nombre de jetons pair, il suffit que ces deux types de variables soient entiers. Les contraintes (2) et (3) permettent d'exprimer ces demi-sommes pour les lignes et les colonnes. Les contraintes (4), (5) et (6) définissent le domaine d'appartenance des variables. Comme il a été mentionné en introduction de ce chapitre, il n'y a pas de fonction objectif pour ce problème, mais seulement un ensemble de contraintes à satisfaire.

(1) $\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij} = P$

(2) $\forall i = 1 \ldots n : \sum_{j=1}^{n} x_{ij} = 2.NL_i$

(3) $\forall j = 1 \ldots n : \sum_{i=1}^{n} x_{ij} = 2.NC_j$

(4) $\forall i = 1 \ldots n, \forall j = 1 \ldots n : x_{ij} \in \{0,1\}$

(5) $\forall i = 1 \ldots n : NL_i \in I\!N$

(6) $\forall j = 1 \ldots n : NC_j \in I\!N$

14.5.3 Modèle Xpress

La transcription en langage Xpress est donnée par le modèle *K4Pair*. Dans la section *Constraints*, il n'y a pas de fonction objectif mais seulement des contraintes. Les variables *NL* et *NC* doivent être entières. Comme la grille de départ n'a que quatre jetons au maximum en ligne et en colonne, les valeurs de ces variables sont limitées à 2 dans la section *Bounds*.

```
MODEL K4Pair

LET

n = 4
P = 10

VARIABLES

x(n,n) ! Variable binaire =1 si la case (i,j) est occupee par un jeton
NL(n)  ! nombre de jetons en ligne (divise par 2)
NC(n)  ! nombre de jetons en colonne (divise par 2)

CONSTRAINTS ! Pas d'objectif, on cherche une solution realisable

Total      : Sum(i=1:n,j=1:n) x(i,j) = P
Lig(i=1:n) : Sum(j=1:n) x(i,j) = 2*NL(i)
Col(j=1:n) : Sum(i=1:n) x(i,j) = 2*NC(j)

BOUNDS

x(i=1:n,j=1:n) .bv.
NC(i=1:n) .ui. 2
```

Index

D

E

F

N

O

P

R

S

T

V

X