

Université Lille I
Licence mention informatique
INFO 204 : ALGO

François Lemaire

1^{er} septembre 2010

Document de cours rédigé par François Boulier (2003-2008).

Introduction

Ce cours est une introduction à la recherche opérationnelle et à l'algorithmique de l'optimisation. L'expression « recherche opérationnelle » date de la seconde guerre mondiale. Elle a été inventée au Royaume Uni et a signifié initialement : optimisation des *opérations* militaires. La première équipe de recherche opérationnelle a été créée en 1938 et placée sous la direction du quartier général de la *Royal Air Force*. Elle s'est illustrée pendant la guerre sur de nombreux problèmes : recoupement des données obtenues par les différentes stations de radar, gestion des équipages lors de la maintenance des avions et surtout, optimisation de la stratégie d'attaque des sous-marins allemands en surface. Parmi ses membres, on peut citer le physicien Patrick Blackett. Après la guerre, les méthodes d'optimisation ont été considérablement appliquées dans le domaine économique (maximisation d'un profit, minimisation d'un coût).

Faire de la recherche opérationnelle consiste en pratique à modéliser mathématiquement un problème donné puis à résoudre le problème modélisé. La première étape demande du savoir-faire et de l'expérience (certains parlent d'« art »). Pour la seconde, on dispose d'algorithmes rigoureux. La discipline s'est développée avec l'informatique : modéliser mathématiquement des problèmes complexes ne servirait à rien si on ne disposait pas d'ordinateurs pour mener les calculs.

Ce cours aborde les deux aspects de la recherche opérationnelle : on s'initie à la modélisation mathématique de problèmes qu'on résout par logiciel (AMPL) et on étudie plusieurs algorithmes importants mis en œuvre par ces logiciels (méthode des moindres carrés, simplexe, algorithmes de théorie des graphes).

La partie pratique de ce cours (modélisation d'une situation, utilisation d'un logiciel) est immédiatement utilisable par tout étudiant entrant dans le monde de l'entreprise avec une Licence. La partie théorique (algorithmique, techniques de preuves, calculs de complexité) s'adresse aux étudiants qui poursuivront des études en Master d'informatique. Dans les feuilles d'exercices, on insiste davantage que dans ce support sur un savoir-faire : être capable de reconnaître un problème d'optimisation dans un problème qui n'est pas *a priori* formulé comme tel. Ce savoir-faire est important du point de vue théorique et du point de vue pratique.

La première section de ce document est consacrée à des rappels d'algèbre linéaire. Notion de matrice et algorithme du pivot de Gauss : paramétrage de l'ensemble des solutions dans le cas où il y a plus d'inconnues que d'équations. On conclut cette section par l'étude de la méthode des moindres carrés qui consiste à optimiser un critère non linéaire et qui s'utilise lorsqu'il y a plus d'équations que d'inconnues.

La deuxième section est consacrée à la programmation linéaire. On y apprend à modéliser des problèmes sous la forme de programmes linéaires en nombres réels ou en nombres entiers : vocabulaire, difficulté de la résolution en nombres entiers par rapport à la résolution en nombres réels, modélisations particulières (report de stocks, minimisation d'une somme ou d'un maxi-

mum de valeurs absolues), comment un modèle peut devenir gros. On insiste sur l'étude de la sensibilité de l'objectif réalisé vis-à-vis d'une perturbation des paramètres. Cette information est particulièrement importante pour l'aide à la décision : elle détermine les contraintes dont il faut s'affranchir en priorité pour améliorer la situation. La mise en œuvre s'effectue avec le logiciel AMPL.

La troisième section est dédiée à l'étude de l'algorithme du simplexe qui constitue la principale méthode de résolution de programmes linéaires en nombres réels. On commence par la résolution graphique qui permet de soutenir l'intuition. On continue par l'algorithme du tableau simplicial, qui permet de résoudre des problèmes de plus de deux variables, inaccessibles à la résolution graphique. On aborde ensuite une méthode de résolution des problèmes de démarrage qui montre comment résoudre les programmes linéaires en nombres réels généraux et qui fournit un bel exemple de réduction de problème. On conclut par l'étude de la dualité dans les programmes linéaires et d'une de ses principales applications : l'analyse de la sensibilité de l'objectif réalisé vis-à-vis d'une perturbation des paramètres.

La quatrième section est consacrée à la théorie des graphes. Définitions (graphes orientés ou non, connexité) et représentations informatiques d'un graphe (listes de successeurs, matrices d'incidence ou d'adjacence). On y étudie quelques algorithmes génériques (parcours en profondeur ou en largeur d'abord, tri topologique des sommets des graphes sans cycle) et quelques algorithmes d'optimisation : recherche d'un chemin de valeur minimale (Bellman, Dijkstra), calcul d'un flot de valeur maximale (Ford-Fulkerson), d'un arbre couvrant de valeur minimale (Kruskal) et d'un ordonnancement de tâches optimal (méthode MPM). En même temps que le principe des algorithmes, on étudie les preuves de correction ainsi que les meilleures structures de données connues permettant de les implanter (piles, files, files avec priorité, ensembles disjoints). Ces structures étant précisées, on obtient une borne de complexité en temps, dans le pire des cas, des méthodes. On montre aussi comment certains de ces problèmes peuvent se coder au moyen de programmes linéaires, ce qui est très utile dès qu'on a affaire à une variante exotique d'un des problèmes étudiés (flot maximal à coût minimal, flot maximal avec déperdition du flux le long du réseau par exemple).

Des compléments sur le langage du logiciel AMPL sont donnés en dernière section.

Ce cours est évidemment très incomplet. Parmi les grandes absentes, mentionnons toute la branche « mathématiques appliquées » de l'optimisation.

Ce cours a été rédigé et mis à jour par François Boulier jusqu'en 2008, à partir des notes de cours de Bernhard Beckermann (une importante source d'inspiration pour la théorie du simplexe et ses applications), Florent Cordellier, Gérard Jacob, Nathalie Revol et des livres [5] (pour AMPL), [7] (riche d'une douzaine de très intéressants problèmes de modélisation), [4] [3] (pour la théorie des graphes) et [1] (pour la méthode des moindres carrés). Sans oublier [2] pour certaines informations historiques.

La plupart des photographies incluses dans le support de cours proviennent du site [6] de l'université de Saint-Andrews.

Ce document a été réalisé par François Boulier en collaboration avec Carine Jaubertie et Nicolas Jozefowicz. Quelle est son originalité vis-à-vis des ouvrages cités en référence ? il s'efforce de présenter ensemble la théorie des algorithmes et leur mise en pratique via un logiciel : AMPL.

Table des matières

1	Rappels d’algèbre linéaire	7
1.1	Les matrices	7
1.1.1	Opérations entre matrices	8
1.1.2	Matrices particulières	9
1.2	Le pivot de Gauss	9
1.2.1	Écriture matricielle d’un système linéaire	10
1.2.2	Le pivot de Gauss	10
1.3	La méthode des moindres carrés	13
1.3.1	La méthode	14
1.3.2	Exemple	14
2	La programmation linéaire	16
2.1	Un exemple d’aciérie	16
2.1.1	Modélisation mathématique	16
2.1.2	Première résolution en AMPL	17
2.1.3	Première modélisation en AMPL	18
2.1.4	Ajout d’un produit	19
2.1.5	Un bricolage	19
2.1.6	Une digression : résolution en nombres entiers	20
2.1.7	Planification sur plusieurs semaines	20
2.2	Sensibilité par rapport à de petites perturbations	22
2.2.1	Valeur marginale d’une contrainte	23
2.2.2	Coût réduit d’une variable	23
2.3	Comment un programme linéaire peut devenir gros	24
2.4	Nommer des quantités intermédiaires	24
2.5	Linéarité des contraintes	25
2.5.1	Paramètres calculés	25
2.5.2	Expressions conditionnelles	26
2.6	Autres problèmes classiques	27
2.6.1	Minimiser une somme de valeurs absolues	27
2.6.2	Minimiser un maximum en valeur absolue	27
2.7	Variables entières et binaires	28
2.7.1	Le solveur à utiliser : cplex	28
2.7.2	Modélisation d’un « ou »	28
2.7.3	Modélisation d’un « et »	29

3	Le simplexe	30
3.1	Vocabulaire	31
3.2	Quelques programmes linéaires particuliers	32
3.3	L'algorithme du tableau simplicial	33
3.3.1	Dans un cas favorable	34
3.3.2	Problèmes de démarrage	42
3.4	La dualité	45
3.4.1	Construction du dual d'un programme linéaire	46
3.4.2	Interprétation du programme dual	47
3.4.3	Résoudre le primal équivaut à résoudre le dual	49
3.4.4	Sensibilité de l'objectif à une perturbation des seconds membres	51
3.4.5	Critères d'optimalité	54
4	Théorie des graphes	56
4.1	Vocabulaire	56
4.2	Représentations d'un graphe	58
4.3	Complexité	59
4.4	Exemples de problèmes rencontrés	61
4.5	Algorithmes de parcours	63
4.5.1	Parcours « en largeur d'abord »	63
4.5.2	Parcours « en profondeur d'abord »	65
4.5.3	Calcul du nombre de composantes connexes d'un graphe	69
4.6	Recherche d'un chemin de valeur minimale	69
4.6.1	Propriétés des chemins de valeur minimale	70
4.6.2	Cas des graphes sans circuits : l'algorithme de Bellman	70
4.6.3	Cas des graphes valués positivement : l'algorithme de Dijkstra	71
4.6.4	Modélisation au moyen de programmes linéaires	76
4.7	Ordonnancement de tâches : la méthode MPM	77
4.8	Flots de valeur maximale	79
4.8.1	Correction	82
4.8.2	Complexité	85
4.8.3	Modélisation au moyen de programmes linéaires	86
4.9	Arbres couvrants de valeur minimale	88
4.9.1	Implantation de l'algorithme de Kruskal	91
5	Compléments sur le langage AMPL	94
5.1	Les commentaires	94
5.2	Les paramètres	94
5.2.1	Paramètres indicés par un même ensemble	94
5.2.2	Paramètres indicés par deux ensembles	95
5.2.3	Laisser des valeurs indéfinies	95
5.2.4	Paramètres indicés par trois ensembles	96
5.3	L'affichage	96
5.4	Les ensembles	97
5.4.1	Les ensembles non ordonnés	97
5.4.2	Les intervalles	97

5.4.3	Les opérations ensemblistes	97
5.4.4	Désigner un élément ou une partie d'un ensemble	97
5.4.5	Désigner un élément d'un ensemble de symboles	98
5.4.6	L'opérateur « : »	99
5.4.7	Ensembles ordonnés	99
5.4.8	Ensembles circulaires	100
5.4.9	Sous-ensembles d'un ensemble ordonné	100
5.4.10	La fonction ord	100
5.4.11	Produits cartésiens	100
5.4.12	Ensembles d'ensembles	101
5.5	Les opérateurs arithmétiques et logiques	102
5.6	Quelques commandes de l'interprète AMPL	102

Chapitre 1

Rappels d'algèbre linéaire

1.1 Les matrices

Définition 1 On appelle matrice $A \in \mathbb{R}^{m \times n}$ (de type $m \times n$) tout tableau à m lignes et n colonnes ayant comme éléments des nombres réels.

Un vecteur de \mathbb{R}^m est vu comme une matrice de type $m \times 1$. Pourquoi des matrices ? Elles permettent de représenter utilement des systèmes d'équations linéaires.

$$\begin{cases} 3x_1 + x_2 - x_3 &= 1 \\ 2x_1 + x_3 &= 5 \end{cases}$$

peut se représenter par

$$\begin{pmatrix} 3 & 1 & -1 \\ 2 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 5 \end{pmatrix}.$$

Pour désigner les éléments d'une matrice, on utilise des listes d'indices. On note

$$A = (A_{\ell}^j)_{\substack{j \in J \\ \ell \in L}} \quad \begin{array}{l} \text{colonnes} \\ \text{lignes} \end{array}$$

La matrice de l'exemple est de type 2×3 . On la note

$$A = A_{(1,2)}^{(1,2,3)}.$$

On désigne la j -ème colonne de A par

$$A^j = \begin{pmatrix} A_1^j \\ \vdots \\ A_m^j \end{pmatrix}.$$

On désigne sa ℓ -ième ligne par

$$A_{\ell} = (A_{\ell}^1 \quad \cdots \quad A_{\ell}^n).$$

Cette notation permet de désigner facilement des sous-matrices de A . Sur l'exemple,

$$A_{(1,2)}^{(1,3)} = \begin{pmatrix} 3 & -1 \\ 2 & 1 \end{pmatrix}$$

désigne la matrice obtenue en supprimant la deuxième colonne de A . On note tA la transposée de A . Les indices de lignes (resp. de colonnes) de A correspondent aux indices de colonnes (resp. de lignes) de tA . Sur l'exemple

$${}^tA = \begin{pmatrix} 3 & 2 \\ 1 & 0 \\ -1 & 1 \end{pmatrix}.$$

Définition 2 Un ensemble de colonnes $\{A^1, \dots, A^r\}$ d'une matrice A est dit linéairement indépendant si

$$\lambda_1 A^1 + \dots + \lambda_r A^r = 0 \quad \Rightarrow \quad \lambda_1 = \dots = \lambda_r = 0.$$

Définition 3 On appelle base d'une matrice A un ensemble linéairement indépendant maximal de colonnes de A (par extension, on appelle base la liste des indices de ces colonnes).

Proposition 1 Toutes les bases d'une matrice A ont même nombre d'éléments.

Définition 4 Ce nombre est appelé le rang de A .

1.1.1 Opérations entre matrices

Soient $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{p \times q}$ et $\lambda \in \mathbb{R}$. Les opérations élémentaires sont les suivantes.

1. La multiplication par un scalaire

$$C = \lambda A = (\lambda A_\ell^j)_{\ell \in L}^{j \in J}.$$

2. La somme de A et de B ne peut se faire que si $m = p$ et $n = q$

$$C = A + B = (A_\ell^j + B_\ell^j)_{\ell \in L}^{j \in J}.$$

3. Le produit de A et de B ne peut se faire que si $n = p$. En supposant $A = A_L^J$ et $B = B_J^K$

$$C = AB = \left(\sum_{j \in J} A_\ell^j B_j^k \right)_{\ell \in L}^{k \in K}.$$

Le produit est associatif mais pas commutatif.

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix} \quad AB = \begin{pmatrix} 0 & 2 \\ 1 & 2 \end{pmatrix} \neq BA = \begin{pmatrix} 2 & 2 \\ 1 & 0 \end{pmatrix}.$$

1.1.2 Matrices particulières

1. La matrice 0 (élément neutre pour l'addition).
2. La matrice unité de taille m . On la note U_m . C'est une matrice carrée de type $m \times m$ avec des 1 sur la diagonale principale et des 0 partout ailleurs.

Définition 5 Une matrice carrée A de type $m \times m$ est dite inversible s'il existe une matrice, notée A^{-1} , telle que $AA^{-1} = U_m$.

Proposition 2 L'inverse d'une matrice inversible A de type $m \times m$ est un inverse aussi bien à gauche qu'à droite :

$$AA^{-1} = A^{-1}A = U_m.$$

Note : il est évident que si A a un inverse à gauche B et un inverse à droite C alors $B = C$ puisque $C = (BA)C$, $B = B(AC)$ et $B(AC) = (BA)C$. La proposition est plus compliquée à démontrer.

1.2 Le pivot de Gauss



FIGURE 1.1 – Carl Friedrich Gauss (1777–1855) en 1803. Il invente la méthode des moindres carrés pour déterminer l'orbite de Cérès en 1801. Il l'applique à nouveau vers 1810 pour l'astéroïde Pallas et publie à cette occasion la méthode qui lui a permis de résoudre le système des « équations normales » ; méthode qu'on appelle aujourd'hui le « pivot de Gauss » [2].

1.2.1 Écriture matricielle d'un système linéaire

On s'intéresse au problème suivant : étant donné un système linéaire, décrire l'ensemble de ses solutions. Voici un exemple de système linéaire :

$$\begin{cases} 2x_1 - x_2 + 4x_3 - 2x_4 &= 0, \\ -2x_1 + 2x_2 - 3x_3 + 4x_4 &= 0, \\ 4x_1 - x_2 + 8x_3 + x_4 &= 1 \end{cases}$$

Résoudre le système revient à résoudre le système

$$Ax = b$$

où la matrice $A \in \mathbb{R}^{m \times n}$ des coefficients du système et le vecteur $b \in \mathbb{R}^m$ des membres gauches des équations sont donnés et où $x = {}^t(x_1 \dots x_n)$ désigne le vecteur des inconnues.

$$\begin{pmatrix} 2 & -1 & 4 & -2 \\ -2 & 2 & -3 & 4 \\ 4 & -1 & 8 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Trois cas peuvent se produire : le système n'a aucune solution, le système a exactement une solution, le système a une infinité de solutions. La résolution peut se mener automatiquement par l'algorithme du « pivot de Gauss » ou sa variante, le « pivot de Gauss–Jordan ».

1.2.2 Le pivot de Gauss

Définition 6 Deux systèmes qui ont même ensemble de solutions sont dits équivalents.

On transforme un système en un système équivalent, plus simple, sur lequel on lit toutes les informations désirées. Les opérations suivantes ne changent pas les solutions d'un système (parce qu'elles sont réversibles). Il faut les faire à la fois sur les membres gauche et droit des équations.

1. Multiplier une ligne par un scalaire non nul.
2. Ajouter à une ligne un multiple d'une autre ligne.
3. Échanger deux lignes.

Le « pivot de Gauss » applique les opérations précédentes sur le système dans le but d'obtenir un système équivalent où chaque équation (on les lit du bas vers le haut) introduit au moins une nouvelle inconnue. Reprenons l'exemple qui précède. Après pivot de Gauss :

$$\begin{cases} 2x_1 - x_2 + 4x_3 - 2x_4 &= 0, \\ x_2 + x_3 + 2x_4 &= 0, \\ -x_3 + 3x_4 &= 1 \end{cases}$$

L'équation du bas introduit x_3 et x_4 . L'équation du milieu introduit x_2 . L'équation du haut introduit x_1 . Plutôt que de manipuler le système d'équations, on manipule la matrice (A, b) formée de la matrice A des coefficients du système, bordée par le vecteur des seconds membres.

$$(A, b) = \left(\begin{array}{cccc|c} 2 & -1 & 4 & -2 & 0 \\ -2 & 2 & -3 & 4 & 0 \\ 4 & -1 & 8 & 1 & 1 \end{array} \right)$$

La matrice (A', b') obtenue après pivot de Gauss :

$$(A', b') = \left(\begin{array}{cccc|c} 2 & -1 & 4 & -2 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 0 & 0 & -1 & 3 & 1 \end{array} \right)$$

Le fait que chaque équation introduit au moins une inconnue se traduit par le fait que la diagonale principale est formée d'éléments non nuls (ce sont les « pivots ») et que les éléments sous les pivots sont nuls. Remarque : on pourrait très bien avoir une diagonale avec des « décrochements ». La variante de Gauss–Jordan est un pivot de Gauss « poussé au maximum » où on impose que les pivots soient égaux à 1 et que tous les éléments d'une colonne contenant un pivot (sauf le pivot bien sûr) soient nuls. Voici le système et la matrice (A'', b'') obtenus après pivot de Gauss–Jordan :

$$\begin{cases} x_1 + 15/12 x_4 = 5/2 \\ x_2 + 5 x_4 = 1 \\ x_3 - 3 x_4 = -1 \end{cases} \quad (A'', b'') = \left(\begin{array}{cccc|c} 1 & 0 & 0 & 15/12 & 5/2 \\ 0 & 1 & 0 & 5 & 1 \\ 0 & 0 & 1 & -3 & -1 \end{array} \right)$$

On peut maintenant décrire l'ensemble des solutions du système en servant de x_4 comme paramètre.

$$\left\{ \begin{pmatrix} 5/2 - 15/12 x_4 \\ 1 - 5 x_4 \\ -1 + 3 x_4 \\ x_4 \end{pmatrix}, \quad x_4 \in \mathbb{R} \right\}$$

Algorithme

On procède colonne par colonne. On commence à la colonne 1. On cherche un élément non nul (un pivot). On choisit $A_1^1 = 2$. On ajoute un multiple de la première ligne à chacune des lignes qui suivent de façon à faire apparaître des zéros sous le pivot. On obtient

$$\left(\begin{array}{ccccc} 2 & -1 & 4 & -2 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 0 & 1 & 0 & 5 & 1 \end{array} \right)$$

On passe à la colonne 2. On cherche un élément non nul sur cette colonne qui ne soit pas sur la même ligne que le pivot précédent (on veut en fait que sur la ligne du nouveau pivot il y ait un zéro au niveau de la première colonne). On choisit le 1 sur la deuxième ligne. On ajoute un multiple de la deuxième ligne à la troisième de façon à faire apparaître des zéros sous le pivot.

$$\left(\begin{array}{ccccc} 2 & -1 & 4 & -2 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 0 & 0 & -1 & 3 & 1 \end{array} \right)$$

C'est fini pour le pivot de Gauss. Pour obtenir un système sous forme de Gauss–Jordan, on peut continuer comme suit. On ajoute un multiple de la troisième ligne aux deux qui précèdent pour faire apparaître des zéros sur la troisième colonne

$$\left(\begin{array}{ccccc} 2 & -1 & 0 & 10 & 4 \\ 0 & 1 & 0 & 5 & 1 \\ 0 & 0 & -1 & 3 & 1 \end{array} \right)$$

On ajoute un multiple de la deuxième ligne à la première pour faire apparaître un zéro sur la deuxième colonne

$$\begin{pmatrix} 2 & 0 & 0 & 15 & 5 \\ 0 & 1 & 0 & 5 & 1 \\ 0 & 0 & -1 & 3 & 1 \end{pmatrix}$$

On multiplie enfin chaque ligne par un scalaire approprié pour que les pivots valent 1.

$$\begin{pmatrix} 1 & 0 & 0 & 15/2 & 5/2 \\ 0 & 1 & 0 & 5 & 1 \\ 0 & 0 & 1 & -3 & -1 \end{pmatrix}$$

Dans le cas général, il peut arriver qu'on ne puisse pas trouver de pivot sur la colonne considérée. Dans ce cas, on passe à la colonne suivante. Le résultat du pivot de Gauss n'est pas unique. À chaque étape, le choix du pivot influe sur les calculs et leur résultat. Il y a pourtant des propriétés du système qui ne dépendent pas du choix du pivot.

Propriétés des systèmes linéaires

Soit (A, b) un système et (A', b') un système équivalent sous forme de Gauss ou de Gauss–Jordan. Le système est sans solutions si et seulement si il y a un pivot dans la colonne b' . Le système a une unique solution si et seulement si toutes les colonnes sauf b' contiennent un pivot. Le système a une infinité de solutions si et seulement si b' et au moins une colonne de A' ne contiennent pas de pivot. C'est le cas de l'exemple qui précède (colonne x_4). Dans ce cas, on peut décrire l'ensemble des solutions du système en se servant des inconnues correspondant aux colonnes sans pivots comme paramètres.

Proposition 3 *L'algorithme du pivot de Gauss ne change ni les bases ni le rang d'une matrice.*

Preuve Il suffit de montrer que le pivot de Gauss ne change pas les bases d'une matrice $A = (A^1 \cdots A^r)$. Les arguments sont les suivants : dire que $\lambda_1 A^1 + \cdots + \lambda_r A^r = 0$ c'est dire que le vecteur ${}^t(\lambda_1 \cdots \lambda_r)$ est solution du système $Ax = 0$; le pivot de Gauss ne change pas les solutions d'un système. \square

Corollaire : on peut lire le rang et une base d'une matrice A sur la matrice A' obtenue après pivot de Gauss : le rang est égal au nombre de pivots ; l'ensemble des indices des colonnes contenant un pivot forme une base. Sur l'exemple, on a trouvé une base $I = (1, 2, 3)$. Le rang du système est donc 3.

Proposition 4 *Le rang d'une matrice de type $m \times n$ est inférieur ou égal au $\min(m, n)$.*

Preuve Le rang de A est inférieur ou égal à n (définition). La proposition 3 implique que le rang est inférieur ou égal à m puisqu'il y a au plus un pivot par ligne de la matrice A' . \square

Éléments supplémentaires

Chacune des trois règles de transformation mises en œuvre par le pivot de Gauss revient à multiplier à gauche la matrice (A, b) par une matrice inversible (la matrice est inversible parce que la transformation qu'elle code est réversible) de type $m \times m$. Sur un système de trois équations par exemple, ajouter la première ligne à la deuxième revient à multiplier à gauche la matrice du système par la matrice

$$L_1 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Soustraire deux fois la première ligne à la troisième revient à multiplier à gauche par la matrice

$$L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}$$

Soustraire la deuxième ligne à la troisième revient à multiplier à gauche par la matrice

$$L_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}$$

Par conséquent

$$(A', b') = L_3 L_2 L_1 (A, b).$$

La matrice $L = L_3 L_2 L_1$ a pour inverse $L^{-1} = L_1^{-1} L_2^{-1} L_3^{-1}$.

Question 1. Quelles sont les matrices correspondant aux deux autres types de transformation ?

Cette constatation permet de démontrer facilement que le pivot de Gauss ne change pas l'ensemble des solutions d'un système. En effet (en notant 0 le vecteur colonne formé de m zéros),

$$Ax = 0 \quad \Rightarrow \quad A'x = L Ax = L 0 = 0; \quad A'x = 0 \quad \Rightarrow \quad Ax = L^{-1} A'x = L^{-1} 0 = 0.$$

1.3 La méthode des moindres carrés

Il s'agit d'une méthode d'optimisation très utilisée : on minimise une somme de carrés. Elle est due à Legendre (1805) et à Gauss (1801). Gauss l'a utilisée dès 1801 pour calculer l'orbite de la planète Cérès à partir d'observations très courtes [2]. Il existe aussi des méthodes pour minimiser le maximum des valeurs absolues (Euler) ou la somme des valeurs absolues (Laplace) des écarts. C'est la méthode des moindres carrés qui conduit aux calculs les plus agréables. On y optimise un critère non linéaire. On verra dans les chapitres suivants que les deux autres problèmes d'optimisation relèvent de la programmation linéaire.

1.3.1 La méthode

On considère un système \mathcal{S} de m équations linéaires à n inconnues x_1, \dots, x_n avec $m > n$ (il y a plus d'équations que d'inconnues, les inconnues sont les x_i , les quantités a_{ij} et b_i sont supposées connues).

$$\mathcal{S} \begin{cases} x_1 a_{11} + \dots + x_n a_{1n} & = b_1 \\ x_1 a_{21} + \dots + x_n a_{2n} & = b_2 \\ & \vdots \\ x_1 a_{m1} + \dots + x_n a_{mn} & = b_m \end{cases}$$

Dans le cas général le système \mathcal{S} n'a aucune solution mais on peut chercher une solution approchée $(\bar{x}_1, \dots, \bar{x}_n)$ telle que la somme de carrés

$$e_1^2 + \dots + e_m^2$$

soit minimale où

$$e_i \stackrel{\text{def}}{=} \bar{x}_1 a_{i1} + \dots + \bar{x}_n a_{in} - b_i, \quad 1 \leq i \leq m$$

désigne l'écart entre la valeur b_i donnée et la valeur $\bar{x}_1 a_{i1} + \dots + \bar{x}_n a_{in}$ fournie par la solution approchée. On peut montrer que cette solution minimale existe et est unique. Matriciellement, \mathcal{S} s'écrit $Ax = b$ c'est-à-dire

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ a_{21} & & a_{2n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

On multiplie à gauche les deux membres de l'égalité par la transposée ${}^t A$ de la matrice A c'est-à-dire

$$\begin{pmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ \vdots & & & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{pmatrix}$$

On obtient un nouveau système ${}^t A A x = {}^t A b$ de n équations à n inconnues, appelé « système des équations normales ». La matrice ${}^t A A$ des coefficients est symétrique. On peut montrer que la solution de ce nouveau système (qui peut s'obtenir par pivot de Gauss par exemple) est la solution approchée $(\bar{x}_1, \dots, \bar{x}_n)$ désirée.

1.3.2 Exemple

On considère les sept points suivants dans le plan (x, y)

i	1	2	3	4	5	6	7
x_i	-3	-2	1	2	3	4	5
y_i	2	0.5	-1	-1	0	2	4

On suppose qu'ils ont été mesurés par un certain procédé et qu'en théorie, ils devraient tous appartenir au graphe d'une même parabole, dont on cherche à identifier les coefficients α, β, γ :

$$y = f(x) = \alpha x^2 + \beta x + \gamma.$$

On cherche donc un triplet $(\bar{\alpha}, \bar{\beta}, \bar{\gamma})$ qui minimise la somme des carrés

$$e_1^2 + \dots + e_7^2$$

où les quantités

$$e_i \stackrel{\text{def}}{=} \bar{\alpha} x_i^2 + \bar{\beta} x_i + \bar{\gamma} - y_i, \quad 1 \leq i \leq 7$$

désignent les écarts entre les ordonnées y_i mesurées et les ordonnées $f(x_i)$ des points de la courbe d'abscisse x_i . Attention : ce sont les valeurs x_i et y_i qui sont connues et les valeurs α , β et γ qui ne le sont pas. Le système surdéterminé est le système $Ax = b$ suivant. Les colonnes de la matrice A sont dans l'ordre, le vecteur des x_i^2 , le vecteur des x_i et un vecteur de 1. Le vecteur b est le vecteur des y_i .

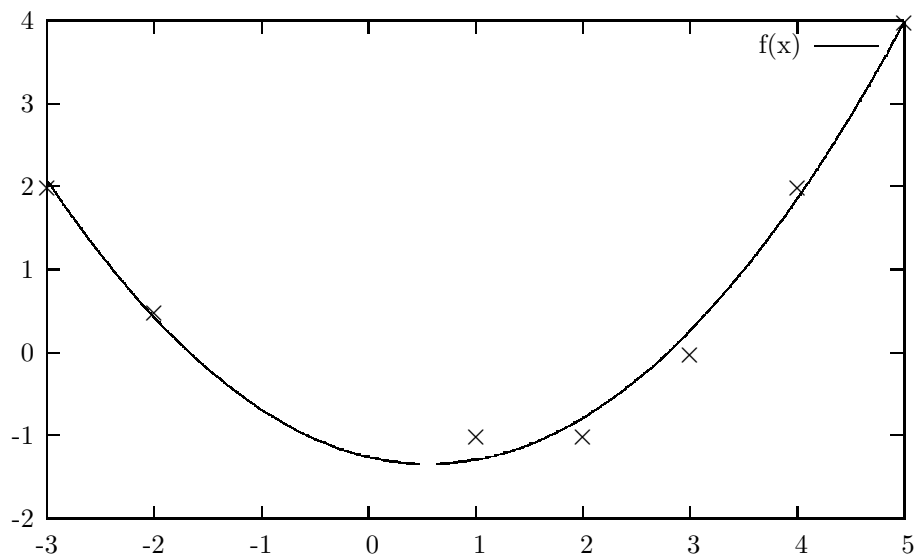
$$A = \begin{pmatrix} 9 & -3 & 1 \\ 4 & -2 & 1 \\ 1 & 1 & 1 \\ 4 & 2 & 1 \\ 9 & 3 & 1 \\ 16 & 4 & 1 \\ 25 & 5 & 1 \end{pmatrix}, \quad x = \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}, \quad b = \begin{pmatrix} 2 \\ 0.5 \\ -1 \\ -1 \\ 0 \\ 2 \\ 4 \end{pmatrix}.$$

En multipliant les deux membres de l'équation $Ax = b$ par la matrice transposée de A , on obtient le système carré ${}^tAAx = {}^tAb$ suivant.

$${}^tAA = \begin{pmatrix} 1076 & 190 & 68 \\ 190 & 68 & 10 \\ 68 & 10 & 7 \end{pmatrix}, \quad {}^tAb = \begin{pmatrix} 147 \\ 18 \\ 6.5 \end{pmatrix}.$$

En résolvant ce système, on obtient les coefficients de la parabole :

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} 0.26988 \\ -1.2584 \\ -.30431 \end{pmatrix}.$$



Chapitre 2

La programmation linéaire

Dans l'expression « programmation linéaire » le mot « programmation » signifie « planification de tâches, d'activités ». Cet usage date des années 1940. Ce n'est que plus récemment que le mot a pris un tout autre sens, au moins chez les informaticiens. L'adjectif « linéaire » signifie que les contraintes et les objectifs à atteindre sont modélisés par des expressions linéaires en les variables du problème. Ce chapitre est une introduction à la modélisation de problèmes par des programmes linéaires ainsi qu'une introduction au logiciel AMPL, dont une version gratuite est accessible sur le site [5].

2.1 Un exemple d'aciérie

2.1.1 Modélisation mathématique

Une aciérie produit des bandes et des rouleaux métalliques. Elle fonctionne 40 heures par semaine. Les vitesses de production sont de 200 bandes par heure et de 140 rouleaux par heure. Les bandes sont vendues 25 euros l'unité ; les rouleaux 30 euros l'unité. Le marché est limité : il est impossible de vendre plus de 6000 bandes et 4000 rouleaux par semaine. Comment maximiser le profit ? Pour modéliser, on dégage

1. les variables (ce qu'on cherche à calculer),
2. les paramètres (les données numériques présentes dans l'énoncé ou qui se calculent facilement à partir de ces dernières),
3. les contraintes,
4. l'objectif (il n'y en a qu'un).

Les variables et les paramètres ont des dimensions qu'il est important de préciser. Il faut aussi déterminer à quel ensemble appartiennent les variables (en général \mathbb{R} ou \mathbb{N}). À chaque contrainte, il est souhaitable d'associer un commentaire (parfois un simple mot-clef). Sur l'exemple, les variables sont

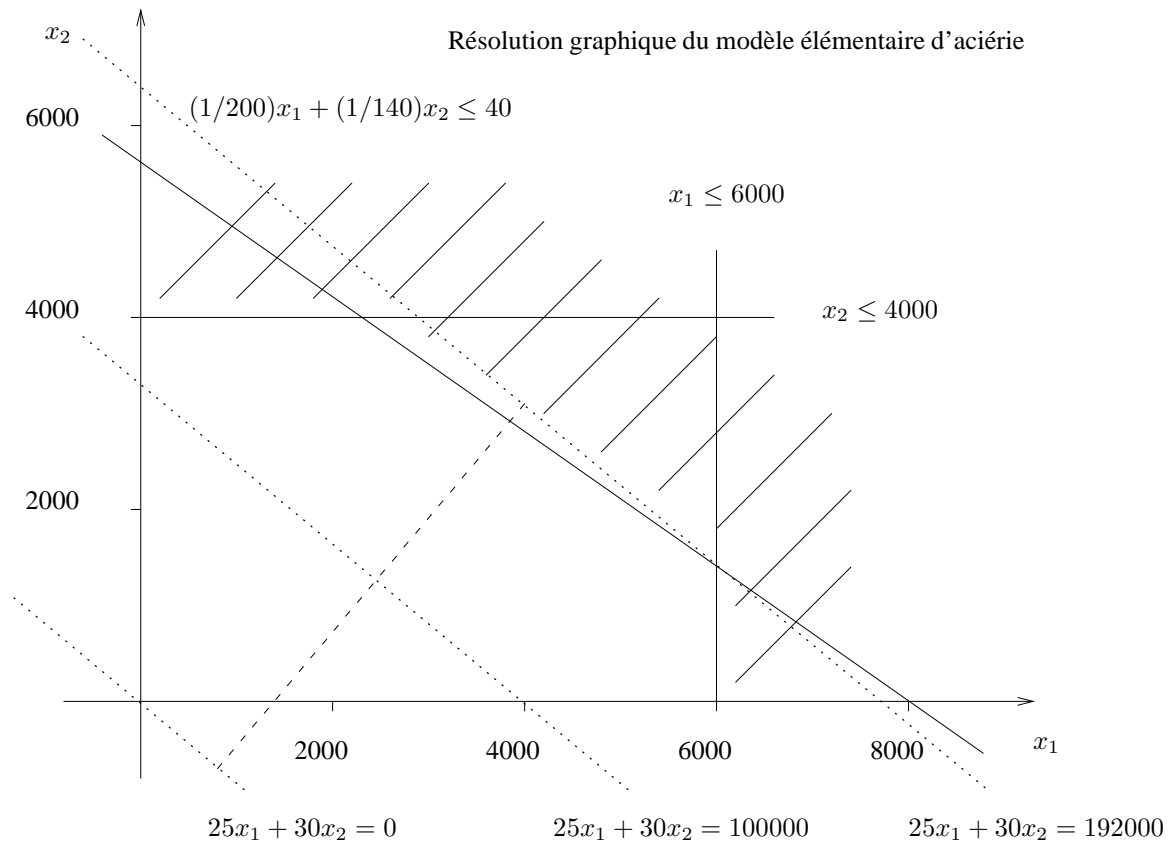
1. x_1 le nombre de bandes à produire
2. x_2 le nombre de rouleaux à produire

On choisit de chercher des valeurs $x_1, x_2 \in \mathbb{R}$ (mais c'est discutable). L'objectif consiste à maximiser $25x_1 + 30x_2$ euros. Les paramètres 25 et 30 ont la dimension d'euros par unité

de produit. Les valeurs numériques qui figurent dans les contraintes sont aussi des paramètres. Mathématiquement, le programme linéaire s'écrit :

$$\begin{aligned}
 25x_1 + 30x_2 &= z[\max] \\
 x_1 &\leq 6000 && \text{(limitation de marché : bandes)} \\
 x_2 &\leq 4000 && \text{(limitation de marché : rouleaux)} \\
 (1/200)x_1 + (1/140)x_2 &\leq 40 && \text{(limitation de la production)} \\
 x_1, x_2 &\geq 0
 \end{aligned}$$

Ce programme linéaire en deux variables peut se résoudre graphiquement.



2.1.2 Première résolution en AMPL

Le logiciel AMPL a été conçu et réalisé par Fourer, Gay et Kernighan (l'un des inventeurs du langage C). Il est constitué d'un logiciel d'interface qui délègue les calculs à des solveurs externes (minos, cplex, ...). Dans ce cours, on s'appuie sur une version bridée gratuite. L'interface est en mode texte. On effectue ici les calculs interactivement via l'interpréteur. Le solveur utilisé s'appelle « minos ». La solution trouvée est assez évidente : chaque heure passée à produire des bandes rapporte $200 \times 25 = 5000$ euros ; chaque heure passée à produire des rouleaux rapporte $140 \times 30 = 4200$ euros. La solution optimale consiste donc à produire des bandes au maximum.

```

ampl: var x1 >= 0;
ampl: var x2 >= 0;

```

```

ampl: maximize z : 25*x1 + 30*x2;
ampl: subject to bandes : x1 <= 6000;
ampl: subject to rouleaux : x2 <= 4000;
ampl: subject to production : (1/200)*x1 + (1/140)*x2 <= 40;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 192000
ampl: display z, x1, x2;
z = 192000
x1 = 6000
x2 = 1400

```

2.1.3 Première modélisation en AMPL

On essaie d'abstraire au maximum le modèle. En particulier, on sépare le modèle des données numériques. Pour cela, on dégage de l'énoncé du problème des « ensembles ». Ce sont ce par quoi les paramètres, les variables et les contraintes seront indicées. Pour un informaticien, les ensembles sont un peu l'analogue des « types énumérés » des langages de programmation classiques. Ici il y en a un : l'ensemble des produits fabriqués par l'aciérie.

```

set PROD;
param heures_ouvrees >= 0;
param vitesse_production {PROD} >= 0;
param prix_vente {PROD} >= 0;
param vente_max {PROD} >= 0;
var qte_produite {p in PROD} >= 0, <= vente_max [p];
maximize profit :
    sum {p in PROD} qte_produite [p] * prix_vente [p];
subject to production_limitee :
    sum {p in PROD}
        (qte_produite [p] / vitesse_production [p]) <= heures_ouvrees;
data;

set PROD := bandes rouleaux;
param heures_ouvrees := 40;
param: vitesse_production prix_vente vente_max :=
bandes      200      25      6000
rouleaux    140      30      4000;

```

Résolution avec AMPL. Accolés aux variables, les suffixes « lb » et « ub » (pour *lower bound* et *upper bound*) en désignent les bornes inférieures et supérieures.

```

ampl: model acieriel;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 192000
ampl: display qte_produite.lb, qte_produite, qte_produite.ub;
:      qte_produite.lb qte_produite qte_produite.ub      :=
bandes      0      6000      6000
rouleaux    0      1400      4000

```

2.1.4 Ajout d'un produit

Cette façon de modéliser est non seulement plus lisible mais plus facile à faire évoluer. Supposons qu'on souhaite ajouter un produit supplémentaire à l'ensemble des produits fabriqués. Il suffit de modifier les données. Le modèle abstrait ne change pas.

```
data;

set PROD := bandes rouleaux poutres;
param heures_ouvrees := 40;
param: vitesse_production prix_vente vente_max :=
bandes      200      25      6000
rouleaux    140      30      4000
poutres     160      29      3500;
```

On résout.

```
ampl: model acierie2;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 196400
ampl: display qte_produite.lb, qte_produite, qte_produite.ub;
:      qte_produite.lb qte_produite qte_produite.ub      :=
bandes      0      6000      6000
poutres      0      1600      3500
rouleaux      0      0      4000
```

2.1.5 Un bricolage

Le profit a augmenté mais on s'aperçoit que la solution optimale consiste à ne plus produire de rouleaux du tout. Ce n'est pas très réaliste. Il serait très difficile de modéliser mathématiquement le fait qu'une aciérie ne peut pas abandonner purement et simplement un marché. On évite alors la solution irréaliste par un « bricolage » : on impose une vente minimale pour chaque produit. Le programme linéaire ainsi obtenu mélange des inégalités dans les deux sens.

```
set PROD;
param heures_ouvrees >= 0;
param vitesse_production {PROD} >= 0;
param prix_vente {PROD} >= 0;
param vente_max {PROD} >= 0;
param vente_min {PROD} >= 0;
var qte_produite {p in PROD} >= vente_min [p], <= vente_max [p];
maximize profit :
    sum {p in PROD} qte_produite [p] * prix_vente [p];
subject to production_limitee :
    sum {p in PROD}
        (qte_produite [p] / vitesse_production [p]) <= heures_ouvrees;
data;

set PROD := bandes rouleaux poutres;
param heures_ouvrees := 40;
param: vitesse_production prix_vente vente_max vente_min :=
bandes      200      25      6000      1000
rouleaux    140      30      4000      500
poutres     160      29      3500      750;
```

On résout

```
ampl: model acierie3;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 194828.5714
ampl: display qte_produite.lb, qte_produite, qte_produite.ub;
:      qte_produite.lb qte_produite qte_produite.ub      :=
bandes      1000      6000      6000
poutres      750      1028.57      3500
rouleaux      500      500      4000
```

2.1.6 Une digression : résolution en nombres entiers

On voit que le nombre de poutres produites n'est pas un entier. Essayons de résoudre le problème en imposant que les variables soient entières. Il suffit pour cela d'ajouter le qualificatif « integer » aux variables et d'utiliser le solveur cplex. Le solveur minos ne permet pas la résolution de programmes linéaires en nombres entiers qui fait appel à d'autres algorithmes que la résolution de problèmes en nombres réels. On remarque au passage que la solution optimale de ce nouveau problème ne s'obtient pas en tronquant purement et simplement la solution optimale précédente.

```
...
var qte_produite {p in PROD} integer >= vente_min [p], <= vente_max [p];
...
```

On résout.

```
ampl: model acierie3_bis;
ampl: option solver cplex;
ampl: solve;
CPLEX 8.0.0: optimal integer solution within mipgap or absmipgap;
objective 194818
1 MIP simplex iterations
0 branch-and-bound nodes
ampl: display qte_produite.lb, qte_produite, qte_produite.ub;
:      qte_produite.lb qte_produite qte_produite.ub      :=
bandes      1000      5999      6000
poutres      750      1027      3500
rouleaux      500      502      4000
```

2.1.7 Planification sur plusieurs semaines

Revenons aux programmes en nombres réels. On souhaite planifier l'activité de l'aciérie sur $N = 3$ semaines. Le prix de vente et le nombre d'heures ouvrées varient avec les semaines. L'aciérie peut décider de stocker une partie de sa production d'une semaine sur l'autre mais elle ne dispose que d'une capacité limitée de stockage : 1000 unités de produits au maximum. Le stock initial est vide. On impose que le stock final le soit aussi. On introduit alors un ensemble SEMS de semaines : un intervalle (c'est-à-dire un ensemble d'entiers consécutifs). Les prix de vente et les quantités produites sont maintenant indicés à la fois par les produits et par les semaines. La contrainte sur la capacité de production est maintenant indicée par les semaines :

il y a en fait autant de contraintes que de semaines. On éprouve le besoin d'affecter à des variables différentes les quantités produites, les quantités vendues et les quantités disponibles en stock en début de semaine. Une contrainte d'égalité apparaît, qui exprime les relations qui lient toutes ces variables : ce qui est produit au cours de la semaine s plus ce qui est disponible en stock au début de la semaine s est égal à ce qui est vendu au cours de la semaine s plus ce qui est disponible en début de semaine suivante. En fait, cette contrainte est indicée par les semaines et les produits : il ne s'agit pas d'une mais de $3N$ contraintes. Remarquer qu'il faut être très précis sur le sens de la variable *qte_stock*. La relation d'équilibre n'est pas la même suivant qu'elle représente les quantités disponibles en début de semaine ou les quantités mises en stock en fin de semaine. Des contraintes d'égalités sont utilisées pour exprimer le fait que les stocks initiaux et finaux sont vides. Ces contraintes vont se comporter en fait comme des affectations.

```

set PROD;
param vitesse_production {PROD} >= 0;
param vente_min {PROD} >= 0;
param vente_max {PROD} >= 0;
param N integer >= 0;
set SEMS := 1 .. N;
param heures_ouvrees {SEMS} >= 0;
param prix_vente {SEMS, PROD} >= 0;
param stock_max >= 0;
var qte_produite {SEMS, PROD} >= 0;
var qte_vendue {s in SEMS, p in PROD} >= vente_min [p], <= vente_max [p];
var qte_stock {1 .. N+1, PROD} >= 0;

maximize profit :
    sum {s in SEMS, p in PROD} qte_vendue [s, p] * prix_vente [s, p];

subject to production_limitee {s in SEMS} :
    sum {p in PROD}
        (qte_produite [s, p] / vitesse_production [p]) <= heures_ouvrees [s];

subject to stock_initial {p in PROD} :
    qte_stock [1, p] = 0;

subject to stock_final {p in PROD} :
    qte_stock [N+1, p] = 0;

subject to equilibre {s in SEMS, p in PROD} :
    qte_stock [s, p] + qte_produite [s, p] =
        qte_stock [s+1, p] + qte_vendue [s, p];

data;

set PROD := bandes rouleaux poutres;
param: vitesse_production vente_max vente_min :=
bandes          200          6000          1000
rouleaux        140          4000           500
poutres         160          3500           750;

param heures_ouvrees :=
1          40
2          20

```

```

3          35;

param N := 3;
param stock_max := 1000;
param prix_vente:
    bandes rouleaux poutres :=
1      25      30      29
2      27      30      28
3      29      30      20;

```

On résout. Remarquer la façon dont le logiciel affiche les contenus de variables différentes indicées de la même façon. La solution optimale trouvée n'est plus évidente du tout.

```

ampl: model acierie4;
ampl: solve;
ampl: display qte_produite, qte_stock, qte_vendue;
:      qte_produite qte_stock qte_vendue      :=
1 bandes      4044.64      0      2044.64
1 poutres      1500      0      750
1 rouleaux      1456.25      0      500
2 bandes      4000      2000      6000
2 poutres      0      750      750
2 rouleaux      0      956.25      500
3 bandes      6000      0      6000
3 poutres      750      0      750
3 rouleaux      43.75      456.25      500
4 bandes      .      0      .
4 poutres      .      0      .
4 rouleaux      .      0      .

```

2.2 Sensibilité par rapport à de petites perturbations

Revenons au premier modèle d'aciérie. Après résolution, on peut aussi s'intéresser à une contrainte. Prenons l'exemple de *production_limitee*. La valeur du corps de la contrainte (c'est-à-dire la somme en partie gauche de l'inégalité) en la solution optimale s'obtient en accolant le suffixe « body » à l'identificateur de la contrainte. Les suffixes « lb » et « ub » désignent les bornes inférieures et supérieures du corps de la contrainte. Les champs « body » et « ub » ont même valeur : on voit que la contrainte est active. Remarquer que le logiciel AMPL donne $-\infty$ comme borne inférieure au lieu de zéro. Cela tient au fait que le modèle ne comporte pas de borne inférieure explicite pour la contrainte.

```

ampl: model acierie1;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 192000
ampl: display production_limitee.body, production_limitee.ub;
production_limitee.body = 40
production_limitee.ub = 40

ampl: display production_limitee.lb;
production_limitee.lb = -Infinity

```

2.2.1 Valeur marginale d'une contrainte

Si on oublie d'accoler le suffixe « body » à l'identificateur de la contrainte, on obtient une toute autre valeur :

```
ampl: display production_limitee;  
production_limitee = 4200
```

Sans ce suffixe, la valeur fournie par AMPL est ce qu'on appelle la « valeur marginale » (ou *shadow price*) de la contrainte. Son sens est le suivant : si on augmente la valeur de la borne d'une « petite » quantité ε alors l'objectif réalisé en la solution optimale augmente de 4200ε . Sur l'exemple $\varepsilon = 1$ est suffisamment petit. On voit que si l'aciérie fonctionne une heure de plus par semaine, l'objectif réalisé passe de 192000 à $192000 + 4200 = 196200$ euros.

```
ampl: let heures_ouvrees := 41;  
ampl: solve;  
MINOS 5.5: optimal solution found.  
1 iterations, objective 196200  
ampl: display production_limitee.body, production_limitee.ub;  
production_limitee.body = 41  
production_limitee.ub = 41
```

Attention : une contrainte a généralement deux bornes (une inférieure et une supérieure). La valeur marginale de la contrainte s'applique à la borne *la plus proche* de la valeur de la contrainte. La valeur marginale d'une contrainte n'a de sens que pour les programmes linéaires en variables réelles. On approfondira cette notion ultérieurement.

2.2.2 Coût réduit d'une variable

Le « coût réduit » d'une variable est une notion analogue à celle de la valeur marginale d'une contrainte. Cette notion s'applique dans le cas des contraintes anonymes imposées à certaines variables lors de leur déclaration. Exemple de la variable *qte_produite*, bornée par le paramètre *vente_max*. Le coût réduit s'obtient en accolant le suffixe « rc » (ou *reduced cost*) à l'identificateur de la variable.

```
ampl: reset;  
ampl: model acieriel;  
ampl: solve;  
MINOS 5.5: optimal solution found.  
2 iterations, objective 192000  
ampl: display qte_produite.rc;  
qte_produite.rc [*] :=  
    bandes 4  
rouleaux 7.10543e-15
```

Son sens est le suivant : si on augmente d'une « petite » quantité ε la vente maximale de bandes, le profit réalisé en la solution optimale augmente de 4ε . Si par contre on augmente la vente maximale de rouleaux, le profit n'est pas modifié. C'est normal : cette dernière contrainte est inactive.


```

ampl: let vente_max ["bandes"] := 6001;
ampl: solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 192004

```

Attention : une variable a généralement deux bornes (une inférieure et une supérieure). Le coût réduit de la variable s'applique à la borne *la plus proche* de la valeur de la variable. Le coût réduit d'une variable n'a lui aussi de sens que pour les programmes linéaires en variables réelles.

2.3 Comment un programme linéaire peut devenir gros

Les programmes linéaires ci-dessus sont petits. On rencontre souvent des programmes linéaires comportant des milliers de variables et de contraintes. Comment un programme linéaire peut-il devenir aussi gros ? On pourrait imaginer qu'on les obtient comme les programmes linéaires ci-dessus mais en ajoutant un grand nombre de données et en tenant compte d'un grand nombre de détails. En fait non. Comme on l'a vu dans l'un des modèles d'aciérie, la modélisation d'un problème n'est pas une construction très rigoureuse (penser à la production minimale imposée pour éviter une solution irréaliste). Complicquer considérablement de tels modèles n'a donc pas vraiment de sens. Et même dans des domaines où une modélisation rigoureuse peut être faite, une multiplication de détails rend le modèle difficile à comprendre : un modèle est une sorte de carte routière ; un modèle très précis est aussi utile qu'une carte au 1/1. En fait, les gros programmes linéaires sont souvent le résultat de la combinaison de petits programmes. Le dernier modèle de la section précédente fournit un exemple. Dans ce modèle, en notant N le nombre de semaines et P le nombre de produits, on obtient $3NP$ variables et $NP + 2P + N$ contraintes. Avec 10 produits et 10 semaines, ce qui est peu, on a déjà 300 variables. Le modèle peut encore se compliquer. On peut imaginer que les produits sont fabriqués dans plusieurs aciéries, qu'il y a un problème de gestion de production dans chaque aciérie. Cela n'est pas bien gênant parce qu'on peut alors résoudre séparément les problèmes de chaque aciérie. Mais s'il faut aussi gérer des moyens de transports communs à toutes les aciéries pour amener les produits à leur destination, on se retrouve nécessairement confronté à un seul, gros, programme linéaire.

2.4 Nommer des quantités intermédiaires

Il est souvent utile de nommer certaines des quantités qui apparaissent dans les programmes linéaires en les stockant dans des variables auxiliaires. Il est alors plus facile de vérifier le modèle à l'exécution : on peut consulter les valeurs de ces variables et vérifier qu'elles sont sensées. Cela peut aussi parfois raccourcir l'écriture du programme. Dans le dernier modèle de gestion d'aciérie, il peut être intéressant de déclarer des variables supplémentaires pour stocker les profits de chaque semaine.

```

...
var profit_hebdomadaire {SEMS} >= 0;

maximize profit : sum {s in SEMS} profit_hebdomadaire [s];

```

```

subject to calcul_profit_hebdomadaire {s in SEMS} :
    profit_hebdomadaire [s] =
        sum {p in PROD} qte_vendue [s, p] * prix_vente [s, p];
...

```

2.5 Linéarité des contraintes

Le logiciel AMPL et les solveurs associés ne permettent de résoudre que des programmes dont les contraintes et l'objectif sont linéaires (à de petites exceptions près). Dans une expression linéaire on a le droit à des sommes de constantes, de variables et de constantes multipliées par des variables. On a le droit de multiplier ou de diviser des constantes entre elles mais le produit ou le rapport de deux variables est interdit. Dans une expression linéaire on a le droit à des instructions *if*, *max*, *abs* etc ... mais l'expression booléenne du *if* ou les opérandes de *max*, *abs* etc ... ne doivent contenir que des expressions constantes. Pour éviter d'engendrer des non linéarités, il est conseillé d'utiliser des paramètres calculés à la place de variables à chaque fois que c'est possible.

2.5.1 Paramètres calculés

Supposons qu'on veuille calculer le taux hebdomadaire d'occupation du marché c'est-à-dire, pour chaque semaine *s*, la quantité totale de produits vendus divisée par la quantité totale de produits vendables. On peut écrire :

```

...
var taux_occupation_marche {SEMS} >= 0;
...
subject to calcul_taux_occupation_marche {s in SEMS} :
    taux_occupation_marche [s] =
        (sum {p in PROD} qte_vendue [s, p]) / (sum {p in PROD} vente_max [p]);

```

L'expression n'étant pas tellement lisible, on est tenté d'appliquer la technique recommandée précédemment et d'affecter à une variable auxiliaire la quantité totale de produits vendables. Ce n'est malheureusement pas possible parce que le programme linéaire ainsi obtenu n'est plus linéaire :

```

...
var vente_totale_max >= 0;
var taux_occupation_marche {SEMS} >= 0;
...
subject to calcul_vente_totale_max :
    vente_totale_max = sum {p in PROD} vente_max [p];
# Contrainte NON LINEAIRE !!!
subject to calcul_taux_occupation_marche {s in SEMS} :
    taux_occupation_marche [s] =
        (sum {p in PROD} qte_vendue [s, p]) / vente_totale_max;

```

Une solution consiste à faire de la quantité totale de produits vendables un paramètre et non plus une variable. On évite ainsi la non linéarité :

```

...
param vente_totale_max := sum {p in PROD} vente_max [p];
var taux_occupation_marche {SEMS} >= 0;
...
subject to calcul_taux_occupation_marche {s in SEMS} :
    taux_occupation_marche [s] =
        (sum {p in PROD} qte_vendue [s, p]) / vente_totale_max;

```

2.5.2 Expressions conditionnelles

Dans le dernier modèle d'aciérie, on avait indiqué la variable *qte_stock* de 1 à $N + 1$ et pas de 1 à N comme les autres variables en raison de la formulation de la contrainte d'équilibre.

```

...
var qte_stock {1 .. N+1, PROD} >= 0;
...
subject to stock_initial {p in PROD} :
    qte_stock [1, p] = 0;
subject to stock_final {p in PROD} :
    qte_stock [N+1, p] = 0;
subject to equilibre {s in SEMS, p in PROD} :
    qte_stock [s, p] + qte_produite [s, p] =
        qte_stock [s+1, p] + qte_vendue [s, p];

```

Une solution plus économe en nombre de variables (mais pas forcément plus lisible) consiste à traiter à part la dernière semaine. On peut pour cela utiliser des expressions conditionnelles. Le programme reste linéaire parce que dans l'expression booléenne du « if » ne figurent que des constantes.

```

...
var qte_stock {SEMS, PROD} >= 0;
...
subject to stock_initial {p in PROD} :
    qte_stock [1, p] = 0;
subject to equilibre {s in SEMS, p in PROD} :
    qte_stock [s, p] + qte_produite [s, p] =
        if s = N then
            qte_vendue [s, p]
        else
            qte_stock [s+1, p] + qte_vendue [s, p]);
...

```

On résout.

```

ampl: model acierie8;
ampl: solve;
MINOS 5.5: optimal solution found.
7 iterations, objective 489866.0714
ampl: display qte_produite, qte_stock, qte_vendue;
:      qte_produite qte_stock qte_vendue      :=
1 bandes      4044.64      0      2044.64
1 poutres      1500      0      750
1 rouleaux     1456.25      0      500
2 bandes      4000      2000      6000
2 poutres      0      750      750

```

2 rouleaux	0	956.25	500
3 bandes	6000	0	6000
3 poutres	750	0	750
3 rouleaux	43.75	456.25	500

On pourrait aussi éviter la contrainte sur le stock initial en utilisant une deuxième expression conditionnelle dans la contrainte d'équilibre mais ce n'est pas souhaitable : cela rend le programme linéaire encore moins lisible sans économiser davantage de variables.

2.6 Autres problèmes classiques

2.6.1 Minimiser une somme de valeurs absolues

On suppose que les variables x_k contiennent des nombres positifs ou négatifs. On souhaite minimiser la somme des valeurs absolues des x_k . La solution qui vient immédiatement à l'esprit fournit un programme non linéaire. Il ne peut pas être résolu par les solveurs *minos* ou *cplex*.

```
set INDICES;
var x {INDICES};

# objectif NON LINEAIRE !!!
minimize somme_vabs :
    sum {k in INDICES} abs (x [k]);
```

Il existe une solution astucieuse. Tout réel x_k peut être vu comme la différence $z_k - t_k$ de deux réels positifs ou nuls. Minimiser la valeur absolue de x_k équivaut à minimiser la somme $z_k + t_k$.

```
set INDICES;
var x {INDICES};
var z {INDICES} >= 0;
var t {INDICES} >= 0;

subject to changement_de_variable {k in INDICES} :
    x [k] = z [k] - t [k];
minimize somme_vabs :
    sum {k in INDICES} z [k] + t [k];
```

2.6.2 Minimiser un maximum en valeur absolue

Il suffit de procéder ainsi ;

```
set INDICES;
var x {INDICES};
var borne >= 0;

subject to borne_inf {k in INDICES} : - borne <= x [k];
subject to borne_sup {k in INDICES} :      x [k] <= borne;
minimise max_vabs : borne;
```

2.7 Variables entières et binaires

Par défaut, les variables AMPL sont de type *réel* (ou plutôt *flottant*). On peut imposer le type *entier* ou *binaire* au moyen des qualificatifs *integer* et *binary*. Noter que pour AMPL, une variable *binaire* n'est rien d'autre qu'une variable *integer* comprise entre zéro ou un. La résolution d'un programme linéaire en nombres entiers est beaucoup plus difficile que la résolution d'un programme de même taille en nombres réels. Par conséquent, autant que possible, il vaut mieux éviter ce type de variable.

2.7.1 Le solveur à utiliser : cplex

Le solveur par défaut, *minos*, ne résout pas les programmes linéaires en nombres entiers. Il faut demander à AMPL d'utiliser le solveur *cplex*. Ce qu'on peut faire avec la commande :

```
ampl: option solver cplex;
```

2.7.2 Modélisation d'un « ou »

Supposons qu'on dispose de deux variables binaires b et c et qu'on souhaite affecter à une variable a (non nécessairement binaire, ce qui peut être utile) la valeur $a = b$ ou c . Comme aucun des solveurs *minos* et *cplex* ne permet de manipuler de contraintes logiques, il faut ruser. On peut par exemple imposer :

$$0 \leq a \leq 1, \quad a \geq b, \quad a \geq c, \quad b + c \geq a.$$

Si a est de type *binaire*, la première contrainte est inutile.

```
# fichier ou
var a >= 0, <= 1;
var b binary;
var c binary;
subject to ou_1 : a >= b;
subject to ou_2 : a >= c;
subject to ou_3 : b + c >= a;
```

Voici des exemples d'exécution. On observe que si on oublie de demander l'utilisation de *cplex*, on obtient un avertissement de *minos* et une solution aberrante.

```
ampl: model ou;
ampl: let b := 1;
ampl: let c := 1;
ampl: solve;
MINOS 5.5: ignoring integrality of 2 variables
MINOS 5.5: optimal solution found.
0 iterations, objective 0
Objective = find a feasible point.
ampl: display a;
a = 0
ampl: option solver cplex;
ampl: solve;
CPLEX 8.0.0: optimal integer solution; objective 0
```

```

0 MIP simplex iterations
0 branch-and-bound nodes
Objective = find a feasible point.
ampl: display a;
a = 1
ampl: let b := 0;
ampl: let c := 0;
ampl: solve;
CPLEX 8.0.0: optimal integer solution; objective 0
0 MIP simplex iterations
0 branch-and-bound nodes
Objective = find a feasible point.
ampl: display a;
a = 0

```

On peut parfois se passer de certaines contraintes. Si par exemple l'objectif est de maximiser a (ou a multiplié par une constante positive) alors les deux contraintes

$$a \geq b, \quad a \geq c$$

qui contraignent la valeur inférieure de a sont superflues. Inversement, si l'objectif est de minimiser a , c'est de la contrainte suivante qu'on peut se passer.

$$b + c \geq a$$

2.7.3 Modélisation d'un « et »

Supposons qu'on dispose de deux variables binaires b et c et qu'on souhaite affecter à une variable a (non nécessairement binaire) la valeur $a = b$ et c . Il suffit de poser

$$0 \leq a \leq 1, \quad a \leq b, \quad a \leq c, \quad a \geq b + c - 1.$$

Comme dans le cas du « ou » la nature de l'objectif permet de se passer parfois de certaines contraintes.

Chapitre 3

Le simplexe

Il permet de résoudre des programmes linéaires en nombres réels uniquement. Bien que sa complexité dans le pire des cas soit assez mauvaise, il se comporte très bien sur les programmes linéaires issus de « vrais » problèmes et souvent bien mieux que des algorithmes concurrents de meilleure complexité dans le pire des cas ! Il a été inventé par George Dantzig en 1947.



FIGURE 3.1 – George Bernard Dantzig (1914–2005).

On va présenter l'algorithme de Dantzig sur l'exemple suivant.

Un boulanger veut produire des rouleaux de pâte brisée (prix de vente : 4 euros le rouleau) et des rouleaux de pâte feuilletée (prix de vente : 5 euros le rouleau). La production d'un rouleau de pâte brisée nécessite 2 kilos de farine et 1 kilo de beurre. Celle d'un rouleau de pâte feuilletée 1 kilo de farine, 2 kilos de beurre et 1 kilo de sel (salée, la pâte). Comment maximiser les bénéfices sachant qu'il dispose d'un stock de 8000 kilos de farine, 7000 kilos de beurre et 3000 kilos de sel ? On a affaire à un programme linéaire en deux variables :

1. x_1 le nombre de milliers de rouleaux de pâte brisée à produire
2. x_2 le nombre de milliers de rouleaux de pâte feuilletée à produire

Mathématiquement, le programme linéaire s'écrit :

$$\begin{aligned} 4000 x_1 + 5000 x_2 &= z[\max] \\ 2 x_1 + x_2 &\leq 8 && (\text{farine}) \\ x_1 + 2 x_2 &\leq 7 && (\text{beurre}) \\ x_2 &\leq 3 && (\text{sel}) \\ x_1, x_2 &\geq 0 && (x_1, x_2 \in \mathbb{R}) \end{aligned}$$

3.1 Vocabulaire

On cherche soit à maximiser soit à minimiser une fonction linéaire $z = f_1 x_1 + \dots + f_n x_n$ où x est un vecteur de \mathbb{R}^n qui appartient à un domaine \mathcal{D} décrit par des contraintes linéaires (égalités, inégalités larges). Exemple

$$\left\{ \begin{array}{ll} A_i^1 x_1 + A_i^2 x_2 + \dots + A_i^n x_n = a_i & (1 \leq i \leq p) \\ B_j^1 x_1 + B_j^2 x_2 + \dots + B_j^n x_n \leq b_j & (1 \leq j \leq q) \\ C_k^1 x_1 + C_k^2 x_2 + \dots + C_k^n x_n \geq c_k & (1 \leq k \leq r) \\ x_1, \dots, x_s \geq 0 \\ x_{s+1}, \dots, x_t \leq 0 \\ x_{t+1}, \dots, x_n & \text{non astreintes de signe} \end{array} \right.$$

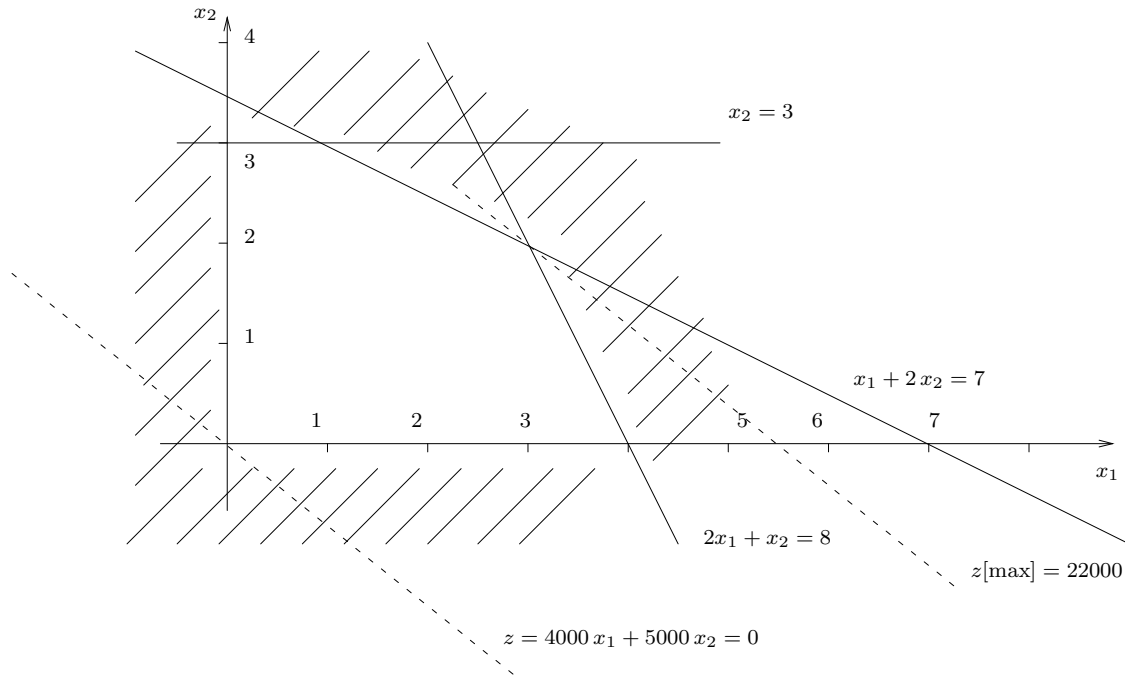
Le domaine \mathcal{D} est le *domaine des solutions réalisables*. Un $x \in \mathcal{D}$ est une *solution réalisable*. Le vecteur colonne f est l'*objectif économique*, ses composantes f_i sont les *coûts* (même s'il s'agit de bénéfices). Remarquer que $z = {}^t f x$. Soit $x \in \mathcal{D}$ une solution réalisable. Une contrainte d'inégalité $B_j^1 x_1 + \dots + B_j^n x_n \leq b_j$ est dite *active* si $B_j^1 x_1 + \dots + B_j^n x_n = b_j$. Elle est dite *inactive* sinon. Écriture matricielle :

$$\left\{ \begin{array}{ll} {}^t f x &= z[\max] \text{ ou } [\min] \\ A x &= a \\ B x &\leq b \\ C x &\geq c \\ x_1, \dots, x_s &\geq 0 \\ x_{s+1}, \dots, x_t &\leq 0 \end{array} \right.$$

Sur l'exemple du boulanger on a :

$$B = \begin{pmatrix} 2 & 1 \\ 1 & 2 \\ 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 8 \\ 7 \\ 3 \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad f = \begin{pmatrix} 4000 \\ 5000 \end{pmatrix}.$$

Les programmes linéaires en deux variables peuvent se résoudre graphiquement.



1. Chaque contrainte d'inégalité est matérialisée par une droite délimitant deux demi-plans. On hachure le demi-plan correspondant aux solutions non réalisables. Le domaine \mathcal{D} des solutions réalisables est un polygone (un polyèdre dans le cas général).
2. Soit $k \in \mathbb{R}$ un nombre réel. L'ensemble des solutions réalisables pour lesquelles l'objectif économique vaut k est l'intersection du polygone \mathcal{D} avec la droite

$${}^t f x = k.$$

Définition 7 On appelle solution optimale toute solution réalisable pour laquelle l'objectif économique est optimal (maximal dans le cas d'une maximisation ou minimal dans le cas d'une minimisation).

Proposition 5 Si elle existe, la solution optimale recherchée est atteinte en (au moins) un sommet du polygone \mathcal{D} .

Elle pourrait ne pas exister : cas d'un polygone ouvert ou d'un polygone vide. Il se peut aussi qu'il y ait une infinité de solutions optimales : cas d'une contrainte parallèle aux droites de l'objectif.

3.2 Quelques programmes linéaires particuliers

Définition 8 Un programme linéaire est dit sous forme canonique s'il s'écrit

$$\begin{cases} {}^t f x = z[\max] \\ Bx \leq b \\ x \geq 0 \end{cases} \quad \text{ou} \quad \begin{cases} {}^t f x = z[\min] \\ Cx \geq c \\ x \geq 0 \end{cases}$$

L'exemple du boulanger est sous forme canonique.

Dans un programme linéaire sous forme canonique, toutes les inégalités sont dans le même sens, on ne trouve pas de contraintes d'égalité et toutes les variables sont positives ou nulles.

Définition 9 *Un programme linéaire est dit sous forme standard s'il s'écrit*

$$\begin{cases} {}^t f x &= z[\max] \\ A x &= a \\ x &\geq 0 \end{cases}$$

Dans un programme linéaire sous forme standard, toutes les contraintes sont des égalités (en général avec beaucoup plus d'inconnues que d'équations).

Les versions algébriques de l'algorithme du simplexe commencent par transformer le programme linéaire à traiter en un programme linéaire équivalent sous forme standard.

Proposition 6 *Tout programme linéaire peut être mis sous forme standard (resp. sous forme canonique).*

Tout problème de minimisation est équivalent à un problème de maximisation : maximiser $4000 x_1 + 5000 x_2$ équivaut à minimiser $-4000 x_1 - 5000 x_2$. Toute inégalité \geq est équivalente à une inégalité \leq en multipliant ses termes par -1 . Par exemple, $2x_1 + x_2 \leq 8$ équivaut à $-2x_1 - x_2 \geq -8$. Toute égalité est équivalente à deux inégalités : $a = 0$ équivaut au système : $a \leq 0, a \geq 0$. Toute inégalité \leq peut être transformée en une égalité au moyen d'une « variable d'écart ». Prenons la contrainte sur la farine dans l'exemple du boulanger : $2x_1 + x_2 \leq 8$ équivaut à $2x_1 + x_2 + x_3 = 8$ avec $x_3 \geq 0$. La variable d'écart x_3 a une signification économique : elle représente la quantité de farine restant en stock. Toute inégalité \geq peut être transformée en une égalité au moyen d'une variable d'écart. Par exemple, $2x_1 + x_2 \geq 8$ équivaut à $2x_1 + x_2 - x_3 = 8$ avec $x_3 \geq 0$. Toute variable non supposée positive ou nulle peut être remplacée par une ou deux variables supposées positives ou nulles : une variable négative ou nulle $x_1 \leq 0$ peut être remplacée par $x_2 \geq 0$ avec $x_2 = -x_1$; une variable x_1 non astreinte de signe peut être remplacée par la différence $x_2 - x_3$ de deux variables $x_2, x_3 \geq 0$.

Question 2. Mettre sous forme canonique et sous forme standard le programme linéaire

$$\begin{cases} 2x_1 - 3x_2 &= z[\min] \\ 2x_1 - x_3 &\geq -5 \\ x_1 + x_2 &= 8 \\ x_1, x_2 &\geq 0 \end{cases}$$

3.3 L'algorithme du tableau simplicial

On commence par transformer le programme linéaire à traiter en un programme linéaire équivalent sous forme standard. Il ne reste alors plus qu'à déterminer une solution optimale d'un programme linéaire sous forme standard.

L'algorithme du simplexe consiste à se déplacer d'un sommet du polyèdre en un autre sommet du polyèdre tout en augmentant l'objectif économique. Ce raisonnement est valable parce que le polyèdre des solutions réalisables est convexe : il n'y a pas de risque de se trouver coincé dans un minimum local. La convexité découle du fait que les contraintes sont données par des expressions linéaires.

3.3.1 Dans un cas favorable

On part de l'exemple du boulanger qui présente l'avantage d'être sous forme canonique avec un vecteur de seconds membres positif ou nul. On le met sous forme standard en utilisant trois variables d'écart x_3, x_4 et x_5 .

$$\begin{array}{rcll} 4000 x_1 + 5000 x_2 & = & z [\max] & \\ 2 x_1 + x_2 + x_3 & = & 8 & \\ x_1 + 2 x_2 + x_4 & = & 7 & \text{ou dit autrement,} \\ x_2 + x_5 & = & 3 & \\ x_1, \dots, x_5 & \geq & 0 & \end{array} \quad \begin{array}{rcl} {}^t f x & = & z[\max] \\ Ax & = & a \\ x & \geq & 0 \end{array}$$

Initialisation

On commence par extraire de la matrice une base I qui soit « réalisable » et telle que le programme linéaire soit « sous forme canonique par rapport à elle ». Comme on va le voir, la liste $I = (3, 4, 5)$ des indices des variables d'écart fournit une telle base.

$$A = \begin{pmatrix} 2 & 1 & 1 & 0 & 0 \\ 1 & 2 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

Définition 10 Soit I une base. Les variables dont les indices appartiennent à I sont appelées variables de base. Les autres sont appelées variables hors base.

Les variables de base sont ici les variables d'écart x_3, x_4 et x_5 .

Définition 11 Un programme linéaire sous forme standard est dit sous forme canonique par rapport à une base I si la matrice A^I est l'identité et les coûts correspondant aux variables de base sont nuls.

Le programme linéaire de l'exemple est sous forme canonique par rapport à la base I .

Définition 12 On suppose une base fixée. On appelle solution de base la solution \bar{x} du système $Ax = a$ obtenue en posant les variables hors base égales à zéro.

La solution de base I est ici

$$\bar{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 8 \\ 7 \\ 3 \end{pmatrix}.$$

Définition 13 Une base est dite réalisable si la solution de base qui lui correspond est réalisable.

La solution de base est par définition une solution du système $Ax = a$. Il ne reste plus qu'à vérifier la contrainte de positivité des variables, qui n'est pas exprimée par les équations. La base I choisie dans l'exemple est donc réalisable.

Proposition 7 Les valeurs des variables de base dans la solution de base se lisent dans le vecteur a .

C'est une conséquence du fait que la matrice A^I est l'identité.

On associe un sommet du polyèdre \mathcal{D} à chaque solution de base : on l'obtient en supprimant, dans la solution de base, les coordonnées correspondant aux variables introduites par la mise sous forme standard. Dans notre cas, on supprime les coordonnées d'indice 3, 4 et 5, qui correspondent aux variables d'écart. Sur l'exemple, le sommet associé à la solution de base I est l'origine $x_1 = x_2 = 0$.

On définit z_0 comme la constante pour laquelle la droite de l'objectif ${}^t f x = z_0$ passe par le sommet associé à la solution de base \bar{x} . Dit autrement,

Définition 14 On définit z_0 comme l'objectif économique évalué sur la solution de base : $z_0 \stackrel{\text{def}}{=} {}^t f \bar{x}$.

On forme un premier « tableau simplicial » à partir du programme linéaire et de la base I choisie en bordant la matrice A à droite avec le vecteur a , à gauche avec le vecteur des variables de base et en-dessous avec l'objectif. On place chaque variable de base sur la ligne où figure sa valeur dans la solution de base. On initialise la case en bas à droite à $-z_0$. Comme la base initiale est formée des variables d'écart qui ne contribuent pas à l'objectif on a $z_0 = 0$.

	x_1	x_2	x_3	x_4	x_5	
x_3	2	1	1	0	0	8
x_4	1	2	0	1	0	7
x_5	0	1	0	0	1	3
	4000	5000	0	0	0	0

Invariants de boucle

L'algorithme du tableau simplicial s'écrit schématiquement ainsi :

former le premier tableau simplicial

tant que ce n'est pas fini **faire**

 former le tableau simplicial suivant (et changer de base)

fait

Définition 15 Un invariant de boucle « tant que » est une propriété vraie à chaque fois que la condition du « tant que » est évaluée.

Un invariant de boucle « tant que » doit donc être vrai au début de la première itération. Il doit aussi être vrai lorsque la boucle s'arrête, c'est-à-dire lorsque la condition s'évalue en « faux ». Combiné à la négation de la condition, l'invariant permet alors de déduire les propriétés des variables calculées par la boucle.

Proposition 8 (invariants de boucle)

Les propriétés suivantes sont des invariants de l'itération principale de l'algorithme du tableau simplicial :

1. Le programme linéaire est sous forme canonique par rapport à la base.
2. La base est réalisable (les éléments de la dernière colonne sont positifs ou nuls).
3. Les variables de base sont listées sur la première colonne.
4. La case en bas à droite vaut $-z_0$.

L'itération principale

Il s'agit d'un pivot de Gauss–Jordan muni d'une stratégie spéciale pour choisir le pivot. Dans ce paragraphe, on se contente d'énoncer l'algorithme. On remarque que l'algorithme ne peut pas conclure au fait que le programme linéaire est sans solution réalisable. Ce cas-là est traité dans la section intitulée « problèmes de démarrage ».

Pour choisir la colonne s du pivot, on cherche sur la dernière ligne (objectif) le coefficient (le coût) le plus grand.

Si ce coût est négatif ou nul on s'arrête : une solution optimale est trouvée.

S'il est positif, la colonne du pivot est trouvée.

Sur l'exemple, la colonne du pivot est la colonne $s = 2$ (correspondant à x_2).

On cherche ensuite s'il existe au moins un élément strictement positif sur la colonne du pivot (on ne cherche pas sur la ligne de l'objectif).

S'il n'y en a pas alors on s'arrête : le problème n'est pas borné (il n'y a pas de solution optimale).

Parmi toutes les lignes contenant un élément strictement positif, on choisit pour ligne du pivot une ligne r telle que le rapport a_r/A_r^s est minimal.

Sur l'exemple, on est amené à comparer les rapports $8/1$, $7/2$ et $3/1$. Le rapport minimal est $3/1$. La ligne du pivot est donc la ligne $r = 3$.

On applique une étape du pivot de Gauss–Jordan : on multiplie la ligne r par l'inverse du pivot (de telle sorte que le pivot devienne 1) et on ajoute un multiple de cette ligne à toutes les autres lignes (ligne d'objectif et dernière colonne incluses) de façon à faire apparaître des zéros sur la colonne du pivot.

Enfin on change de base en faisant sortir de la base la variable x_5 qui se trouve sur la ligne du pivot dans le tableau simplicial et en la remplaçant par la variable x_2 correspondant à la colonne du pivot. Sur l'exemple, on obtient

	x_1	x_2	x_3	x_4	x_5	
x_3	2	0	1	0	-1	5
x_4	1	0	0	1	-2	1
x_2	0	1	0	0	1	3
	4000	0	0	0	-5000	-15000

Et on recommence.

Proposition 9 *Les invariants sont à nouveau vérifiés.*

On vérifie que le nouveau programme linéaire est sous forme canonique par rapport à I' . Au passage, on remarque que l'ancienne base I est toujours une base (le pivot de Gauss ne change pas les bases) mais que le nouveau programme linéaire n'est pas sous forme canonique par rapport à elle.

La solution de base I' obtenue en posant les variables hors base x_1 et x_5 égales à zéro est

$$\bar{x}' = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \\ 5 \\ 1 \\ 0 \end{pmatrix}.$$

Comme toutes ses coordonnées sont positives ou nulles, la base I' est réalisable. On voit aussi que les valeurs des variables de base I' se lisent sur la colonne a' et que le réel en bas à droite vaut bien $-z'_0 = -{}^t f \bar{x}' = -4000 x_1 - 5000 x_2 = -15000$.

Proposition 10 *Le sommet associé à la nouvelle base est adjacent au sommet associé à l'ancienne.*

Le sommet associé à la nouvelle base est le sommet $(x_1, x_2) = (0, 3)$. La droite parallèle à la droite de l'objectif qui passe par ce sommet a pour équation $4000 x_1 + 5000 x_2 = 15000$.

Proposition 11 *La valeur de z_0 a augmenté.*

Quelques remarques

Le fait de choisir une colonne sur laquelle le coût est non nul fait que la variable qui entre en base est bien une variable hors base (cf. le premier invariant).

Le choix de la colonne du pivot est heuristique. On peut en fait choisir n'importe quelle colonne sur laquelle le coût est positif. Sur l'exemple, on aurait pu faire entrer en base la variable x_1 . On s'interdit de choisir une colonne sur laquelle le coût est négatif pour ne pas faire diminuer l'objectif réalisé.

Le choix de la ligne n'est pas heuristique. On voit sur l'exemple que si on choisit une autre ligne que celle donnée par l'algorithme on obtient des nombres négatifs sur la colonne des seconds membres c'est-à-dire une base non réalisable.

Fin de l'exemple

La colonne du pivot est la colonne $s = 1$ correspondant à x_1 . La ligne du pivot est la ligne $r = 2$. La variable x_4 sort de la base. La variable x_1 y entre. Le programme linéaire est sous forme canonique par rapport à la base $I'' = (3, 1, 2)$. La solution de base I'' correspond au sommet $(x_1, x_2) = (1, 3)$.

	x_1	x_2	x_3	x_4	x_5	
x_3	0	0	1	-2	3	3
x_1	1	0	0	1	-2	1
x_2	0	1	0	0	1	3
	0	0	0	-4000	3000	-19000

La colonne du pivot est la colonne $s = 5$ correspondant à x_5 . La ligne du pivot est la ligne $r = 1$. La variable x_3 sort de la base. La variable x_5 entre. Le programme linéaire est sous forme canonique par rapport à la base $I''' = (5, 1, 2)$. La solution de base I''' correspond à la solution $(x_1, x_2) = (3, 2)$.

	x_1	x_2	x_3	x_4	x_5	
x_5	0	0	1/3	-2/3	1	1
x_1	1	0	2/3	-1/3	0	3
x_2	0	1	-1/3	2/3	0	2
	0	0	-1000	-2000	0	-22000

Tous les coûts sont négatifs ou nuls (critère d'arrêt de l'algorithme). La solution de base I''' est donc optimale. L'objectif atteint pour cette solution est $z_0''' = 22000$.

On vérifie que les trois propositions énoncées ci-dessus sont vérifiées à chaque itération.

Programme sous forme canonique associé à un tableau simplicial

On a compris sur un exemple qu'à tout programme linéaire sous forme canonique avec seconds membres positifs ou nuls on pouvait associer un tableau simplicial satisfaisant les invariants. La réciproque est vraie aussi.

Proposition 12 *À tout tableau simplicial satisfaisant les invariants il est possible d'associer un programme linéaire sous forme canonique avec seconds membres positifs ou nuls.*

Il suffit pour cela de traduire le tableau simplicial comme un programme linéaire sous forme standard puis d'interpréter les variables de base comme des variables d'écart. Considérons à titre d'exemple le deuxième tableau simplicial obtenu à partir de l'exemple du boulanger :

	x_1	x_2	x_3	x_4	x_5	
x_3	2	0	1	0	-1	5
x_4	1	0	0	1	-2	1
x_2	0	1	0	0	1	3
	4000	0	0	0	-5000	-15000

Ce tableau correspond au programme linéaire sous forme standard

$$\left\{ \begin{array}{rcl} 4000 x_1 - 5000 x_5 + 15000 & = & z \text{ [max]} \\ 2 x_1 + x_3 - x_5 & = & 5 \\ x_1 + x_4 - 2 x_5 & = & 1 \\ x_2 + x_5 & = & 3 \\ x_1, \dots, x_5 & \geq & 0 \end{array} \right.$$

Il s'agit en fait du programme linéaire obtenu en interprétant l'équation $x_2 + x_5 = 3$ comme la règle de substitution $x_2 \rightarrow 3 - x_5$ et en l'utilisant pour réécrire toutes les autres équations ainsi que l'objectif économique du programme initial. Interprétons maintenant les variables de base comme des variables d'écart. On obtient le programme linéaire suivant :

$$\left\{ \begin{array}{rcl} 4000 x_1 - 5000 x_5 + 15000 & = & z \text{ [max]} \\ 2 x_1 - x_5 & \leq & 5 \\ x_1 - 2 x_5 & \leq & 1 \\ x_5 & \leq & 3 \\ x_1, x_5 & \geq & 0 \end{array} \right.$$

On remarque que c'est bien le premier invariant qui rend cette transformation possible : chaque variable choisie comme variable d'écart apparaît avec un coefficient 1 dans exactement une équation et n'apparaît pas dans l'objectif économique. D'où l'expression « programme linéaire sous forme canonique par rapport à une base ».

La transformation est réversible : si on construit un premier tableau simplicial à partir de ce programme, on retrouve le tableau simplicial ci-dessus (au nommage des variables d'écart près). Résoudre ce programme linéaire équivaut donc à résoudre l'exemple du boulanger.

Problèmes bornés

Revenons au tableau simplicial :

	x_1	x_2	x_3	x_4	x_5	
x_3	2	0	1	0	-1	5
x_4	1	0	0	1	-2	1
x_2	0	1	0	0	1	3
	4000	0	0	0	-5000	-15000

La solution réalisable courante est la solution de base

$$\bar{x}' = \begin{pmatrix} 0 \\ 3 \\ 5 \\ 1 \\ 0 \end{pmatrix}.$$

L'objectif est de maximiser $4000x_1 - 5000x_5 + 15000$. On voit que les variables de base x_2 , x_3 et x_4 ont une valeur non nulle mais ne contribuent pas à l'objectif (on ne perdrait rien à diminuer leur valeur) alors que la variable x_1 a une valeur nulle mais est associée à un coût positif dans l'objectif. On a tout intérêt à augmenter sa valeur. Il suffit pour cela de la faire entrer en base. Par un raisonnement similaire, on voit qu'on ne peut pas modifier la valeur de x_5 sans faire diminuer l'objectif. On en déduit que lorsque tous les coûts sont négatifs ou nuls, il est impossible de faire entrer une variable en base sans faire diminuer l'objectif : on a atteint un maximum local et donc le maximum global recherché puisque le polyèdre des solutions réalisables est convexe.

Les raisonnements tenus ci-dessus justifient le critère d'arrêt de l'algorithme dans le cas de problèmes bornés.

Théorème 1 (condition suffisante d'optimalité)

Considérons un programme linéaire sous forme canonique par rapport à une base réalisable I . Si les coûts correspondant aux variables hors base sont négatifs ou nuls alors la solution de base I est la solution optimale et l'objectif à l'optimum est z_0 .

Problèmes non bornés

Modifions un peu le deuxième tableau simplicial de telle sorte que l'algorithme s'arrête en indiquant que le programme linéaire est non borné. Il suffit de rendre négatifs tous les coefficients présents sur la colonne du pivot.

	x_1	x_2	x_3	x_4	x_5	
x_3	-2	0	1	0	-1	5
x_4	-1	0	0	1	-2	1
x_2	0	1	0	0	1	3
	4000	0	0	0	-5000	-15000

Considérons le programme linéaire sous forme canonique associé à ce tableau simplicial.

$$\left\{ \begin{array}{rcl} 4000 x_1 - 5000 x_5 + 15000 & = & z \text{ [max]} \\ -2 x_1 - x_5 & \leq & 5 \\ -x_1 - 2 x_5 & \leq & 1 \\ x_5 & \leq & 3 \\ x_1, x_5 & \geq & 0 \end{array} \right.$$

Parce que tous les coefficients situés sur la colonne du pivot sont négatifs, on voit qu'on peut attribuer à x_1 une valeur arbitrairement grande sans violer les contraintes. Parce que le coût associé à x_1 est positif, on voit qu'augmenter la valeur de x_1 augmente l'objectif réalisé.

On a donc bien affaire à un programme linéaire non borné.

Théorème 2 (*problème non borné*)

Considérons un programme linéaire sous forme canonique par rapport à une base réalisable I . S'il existe un indice de colonne s tel que le coût f_s soit positif et tous les coefficients de la colonne s soient négatifs ou nuls alors le programme linéaire n'est pas borné.

Question 3. Appliquer l'algorithme sur l'exemple suivant. Vérifier qu'il s'agit d'un programme linéaire non borné. Associer au tableau simplicial final un programme linéaire sous forme canonique et refaire le raisonnement ci-dessus.

$$\left\{ \begin{array}{rcl} -x_1 + x_2 & = & z \text{ [max]} \\ x_1 - 2x_2 & \leq & 2 \\ -2x_1 + x_2 & \geq & -4 \\ -3x_1 + x_2 & \leq & 6 \\ x_1, x_2 & \geq & 0 \end{array} \right.$$

Question 4. Comment interpréter le cas où tous les coefficients situés sur une colonne (y compris le coût) sont négatifs ?

Sommets multiples

Dans certains cas, il se peut que l'élément de la colonne des seconds membres situé sur la ligne du pivot, a_r , soit nul. Dans ce cas, la valeur de la variable de base x_s dans la solution de base est nulle et l'objectif réalisé n'augmente pas. Interprétation graphique. On passe d'un sommet à un sommet adjacent sans déplacer la droite de l'objectif. Comme le polyèdre est convexe, les deux sommets sont superposés : on a affaire à un sommet « multiple ».

Les sommets multiples sont rares dans les petits programmes linéaires écrits avec un minimum de contraintes et de variables (tels ceux qu'on rencontre dans les feuilles de TD). Ils apparaissent plus naturellement dans des modèles plus volumineux lorsqu'on rajoute, par exemple, des contraintes et des variables « mathématiquement inutiles » qui servent à nommer des quantités intermédiaires et simplifient l'écriture des modèles.

Question 5. Appliquer l'algorithme sur l'exemple ci-dessous. On s'aperçoit de la présence d'un sommet multiple dès la première itération : on a le choix entre deux indices de ligne pour le premier pivot (clarifier cette affirmation).

$$\begin{cases} x_1 - x_2 = z [\max] \\ x_1 - 2x_2 \leq 2 \\ 2x_1 - x_2 \leq 4 \\ x_1 + x_2 \leq 5 \\ x_1, x_2 \geq 0 \end{cases}$$

Le problème de l'arrêt

Dans le cas où l'objectif réalisé augmente strictement à chaque itération l'algorithme s'arrête nécessairement : des valeurs différentes de l'objectif réalisé correspondent à des sommets différents du polyèdre \mathcal{D} or \mathcal{D} n'a qu'un nombre fini de sommets. L'algorithme n'est donc susceptible de boucler que dans le cas où l'objectif réalisé garde la même valeur, c'est-à-dire dans le cas de sommets multiples.

La stratégie adoptée ne garantit pas l'arrêt de l'algorithme.

Il se peut que l'algorithme énumère cycliquement une série de bases réalisables sans que l'objectif réalisé change. Voici un exemple :

$$\begin{cases} (3/4)x_1 - 20x_2 + (1/2)x_3 - 6x_4 = z [\max] \\ (1/4)x_1 - 8x_2 - x_3 + 9x_4 \leq 0 \\ (1/2)x_1 - 12x_2 - (1/2)x_3 + 3x_4 \leq 0 \\ x_3 \leq 1. \end{cases}$$

Après mise sous forme standard et ajout de trois variables d'écart x_5, x_6 et x_7 , on obtient un premier tableau simplicial et une première base (x_5, x_6, x_7) . En appliquant la stratégie que nous avons énoncée, on peut obtenir (« peut » parce qu'on a parfois plusieurs choix possibles pour la ligne) successivement les bases (x_1, x_6, x_7) , (x_1, x_2, x_7) , (x_3, x_2, x_7) , (x_3, x_4, x_7) , (x_5, x_4, x_7) et (x_5, x_6, x_7) . Le tableau simplicial obtenu à la dernière étape est égal au premier tableau simplicial. L'objectif réalisé en la solution de base a toujours gardé la valeur $z_0 = 0$.

Théorème 3 (stratégie garantie)

La stratégie suivante de sélection du pivot garantit l'arrêt de l'algorithme.

1. choisir la colonne possible de plus petit indice (s est le plus petit indice i tel que $f_i > 0$).
2. choisir la ligne possible de plus petit indice (prendre pour r le plus petit indice i tel que $A_r^s > 0$ et tel que la base obtenue après pivot soit réalisable).

Cette stratégie garantie est rarement utilisée parce que, sur beaucoup de problèmes pratiques, elle nécessite beaucoup plus d'itérations que la stratégie usuelle pour atteindre la solution optimale. Une bonne méthode consiste à appliquer la stratégie non garantie tout en conservant un petit historique des valeurs de l'objectif réalisé. Cet historique permet de vérifier que les valeurs de l'objectif réalisé ne restent pas stationnaires trop longtemps. Si ce n'est pas le cas, il suffit d'appliquer la stratégie garantie jusqu'à ce que la valeur de l'objectif réalisé change puis d'appliquer à nouveau la stratégie usuelle.

Complexité de l'algorithme

L'exemple suivant montre que l'algorithme du tableau simplicial (muni d'une stratégie qui garantit l'arrêt) a une complexité en temps exponentielle en le nombre de variables (ou de contraintes) dans le pire des cas.

$$\begin{array}{rcl}
 2^{n-1} x_1 + 2^{n-2} x_2 + \cdots + 2 x_{n-1} + x_n & = & z[\max] \\
 x_1 & \leq & 5 \\
 4 x_1 + x_2 & \leq & 25 \\
 8 x_1 + 4 x_2 + x_3 & \leq & 125 \\
 & \vdots & \\
 2^n x_1 + 2^{n-1} x_2 + \cdots + 4 x_{n-1} + x_n & \leq & 5^n \\
 x_1, \dots, x_n & \geq & 0.
 \end{array}$$

Ce programme linéaire a n variables, n contraintes. Le polyèdre des solutions réalisables a 2^n sommets. L'algorithme du tableau simplicial, partant de $x_1 = \cdots = x_n = 0$ énumère tous les sommets avant d'atteindre la solution optimale $(x_1, \dots, x_n) = (0, \dots, 0, 5^n)$. Si la complexité dans le pire des cas était représentative de la complexité pour les « vrais » exemples alors le simplexe serait inutilisable. Penser que certains problèmes comportent des centaines de milliers de variables. Heureusement, ce n'est pas le cas !

3.3.2 Problèmes de démarrage

On dit qu'un problème P peut se *réduire* en un problème Q si toute instance de P peut se reformuler en une instance de Q .

Supposons qu'un problème P puisse se réduire en un problème Q et qu'on connaisse un algorithme de résolution pour toutes les instances de Q . Alors on dispose d'un algorithme de résolution pour toutes les instances de P : il suffit, étant donné une instance de P , de la reformuler en une instance de Q , de déterminer la solution du problème reformulé avec l'algorithme connu puis de réinterpréter la solution obtenue comme une solution du problème initial.

Pour pouvoir traiter les programmes linéaires généraux, la difficulté consiste à former un premier tableau simplicial satisfaisant les invariants. En effet, former ce premier tableau équivaut à identifier un sommet du polyèdre des solutions réalisables (via la solution de base) et donc, en particulier, à démontrer que le programme linéaire admet au moins une solution réalisable. Dans le « cas favorable » (inégalités \leq avec seconds membres positifs ou nuls), l'origine est une solution évidente mais c'est faux en général où il se peut que le programme linéaire soit infaisable.

Dans cette section, on montre que le problème de la recherche d'un premier tableau simplicial peut se réduire en un problème d'optimisation, la résolution d'un autre programme linéaire, appelé « programme artificiel », qui relève, lui, du cas favorable¹. Dit autrement, si on sait résoudre les programmes linéaires dans le cas favorable, on sait résoudre les programmes linéaires dans tous les cas. Une fois le programme artificiel résolu, on détermine facilement si le problème initial admet au moins une solution. Si c'est le cas, il ne reste plus qu'à résoudre

1. La méthode présentée ici est la méthode « des deux phases ». D'autres méthodes existent comme, par exemple, celle du « grand M ».

le programme linéaire initial en partant de la base trouvée. Prenons pour programme linéaire initial :

$$(PL) \begin{cases} -x_1 - x_2 = z [\max] \\ -3x_1 - 4x_2 \leq -12 \\ 2x_1 + x_2 \leq 4 \\ x_1, x_2 \geq 0 \end{cases}$$

La mise sous forme standard introduit deux variables d'écart.

$$(PL) \begin{cases} -x_1 - x_2 = z [\max] \\ 3x_1 + 4x_2 - x_3 = 12 \\ 2x_1 + x_2 + x_4 = 4 \\ x_1, x_2, x_3, x_4 \geq 0 \end{cases}$$

La base (3, 4) n'est pas réalisable puisque la solution de base comporte une coordonnée négative (on a $x_3 = -12$). Voici le programme artificiel associé à l'exemple.

$$(PLA) \begin{cases} x_5 = w [\min] \\ 3x_1 + 4x_2 - x_3 + x_5 = 12 \\ 2x_1 + x_2 + x_4 = 4 \\ x_1, x_2, x_3, x_4, x_5 \geq 0 \end{cases}$$

On a introduit une « variable artificielle » x_5 dans l'équation qui pose problème. On a pris pour « objectif artificiel » : minimiser cette variable.

Explication. Le programme artificiel relève du cas favorable : la base évidente (x_5, x_4) est réalisable. Il y a bijection entre les solutions du programme linéaire initial et les solutions du programme artificiel telles que $x_5 = 0$. Pour déterminer si le programme initial admet au moins une solution, il suffit donc de déterminer si le programme artificiel a une solution telle que $x_5 = 0$. C'est ce qu'on fait en prenant pour objectif : minimiser x_5 .

Pour pouvoir appliquer l'algorithme du tableau simplicial il reste une difficulté technique à lever : la variable x_5 est en base alors que le coût f_5 est non nul (l'un des invariants n'est pas satisfait). Il suffit de reformuler l'objectif en utilisant l'équation où figure la variable artificielle :

$$3x_1 + 4x_2 - x_3 + x_5 = 12 \quad \Rightarrow \quad -x_5 = 3x_1 + 4x_2 - x_3 - 12.$$

Le nouvel objectif à maximiser est donc :

$$3x_1 + 4x_2 - x_3 - 12 = w [\max].$$

Pour déterminer w_0 , on évalue l'objectif $-x_5$ (ou encore $3x_1 + 4x_2 - x_3 - 12$) sur la solution de base. Attention à la constante cette fois. On obtient $w_0 = -12$. On obtient ainsi un premier tableau simplicial satisfaisant tous les invariants de l'algorithme du tableau simplicial

	x_1	x_2	x_3	x_4	x_5	
x_5	3	4	-1	0	1	12
x_4	2	1	0	1	0	4
	3	4	-1	0	0	12

On applique au programme artificiel l'algorithme donné dans le cas favorable. On trouve en une itération le tableau simplicial suivant :

	x_1	x_2	x_3	x_4	x_5	
x_2	$3/4$	1	$-1/4$	0	$1/4$	3
x_4	$5/4$	0	$1/4$	1	$-1/4$	1
	0	0	0	0	-1	0

On a une base réalisable $I' = (2, 4)$ avec une solution de base I'

$$\bar{x}' = \begin{pmatrix} 0 \\ 3 \\ 0 \\ 1 \\ 0 \end{pmatrix}.$$

L'optimum du programme artificiel est $w_0 = 0$. La variable artificielle x_5 est hors base. On obtient une première base réalisable pour le programme linéaire initial en « supprimant » purement et simplement la variable artificielle. On obtient ainsi un programme linéaire équivalent au programme linéaire initial avec une première base réalisable $I = (2, 4)$:

$$\begin{cases} -x_1 - x_2 = z [\max] \\ 3/4 x_1 + x_2 - 1/4 x_3 = 3 \\ 5/4 x_1 + 1/4 x_3 + x_4 = 1 \\ x_1, x_2, x_3, x_4 \geq 0 \end{cases}$$

Synthétisons :

Proposition 13 *Le programme linéaire initial admet au moins une solution réalisable si et seulement si l'objectif réalisé à l'optimum du programme artificiel est $w_0 = 0$.*

Proposition 14 *Supposons que l'objectif réalisé à l'optimum du programme artificiel soit $w_0 = 0$. La solution optimale du programme artificiel fournit une première base réalisable au programme linéaire initial si et seulement si aucune variable artificielle n'est en base.*

On remarque que le programme linéaire obtenu ne satisfait pas tous les invariants de l'algorithme du tableau simplicial puisque les coûts correspondant aux variables de base ne sont pas tous nuls. On applique la même technique que précédemment : on réécrit l'objectif en servant de la première équation.

$$3/4 x_1 + x_2 - 1/4 x_3 = 3 \Rightarrow x_2 = 3 - 3/4 x_1 + 1/4 x_3.$$

Et donc $-x_1 - x_2 = -1/4 x_1 - 1/4 x_3 - 3$. Le nouvel objectif est (on peut supprimer la constante) :

$$-1/4 x_1 - 1/4 x_3 = z[\max].$$

Pour déterminer z_0 , on évalue l'objectif (attention à la constante) sur la solution de base : $z_0 = -3$.

	x_1	x_2	x_3	x_4	
x_2	$3/4$	1	$-1/4$	0	3
x_4	$5/4$	0	$1/4$	1	1
	$-1/4$	0	$-1/4$	0	3

On peut appliquer l'algorithme étudié dans le cas favorable. Sur l'exemple, il s'arrête tout de suite puisque tous les coûts sont négatifs ou nuls (mais c'est un hasard dû à l'exemple). La solution optimale du programme linéaire initial est donc

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \end{pmatrix}.$$

L'objectif réalisé à l'optimum est $z_0 = -3$.

Remarques

Dans le cas général, on peut être amené à introduire plusieurs variables artificielles. L'objectif du programme linéaire artificiel consiste alors à minimiser leur somme.

Il existe des cas « dégénérés » où, à l'optimum du programme artificiel, l'objectif réalisé est nul (prouvant que le programme initial a au moins une solution) mais où la base réalisable contient au moins une variable artificielle (ayant alors forcément une valeur nulle).

C'est gênant parce qu'alors, en « supprimant » la variable artificielle, on « perd » la base finale du programme artificiel qui doit nous servir de première base pour traiter le programme initial mais ce n'est pas rédhibitoire : comme sa valeur est nulle dans la solution de base, on ne modifie pas l'objectif réalisé à l'optimum du programme artificiel en la rendant hors base. Il suffit pour cela de la remplacer dans la base par une variable non artificielle en effectuant une étape supplémentaire du pivot de Gauss–Jordan.

Question 6. Résoudre le programme linéaire suivant. Le programme artificiel associé comporte deux variables artificielles. Après résolution, l'une d'elles reste en base. La chasser de la base finale en effectuant une étape supplémentaire du pivot de Gauss–Jordan.

$$\begin{cases} x_1 - 2x_2 + x_3 = z \text{ [min]} \\ 3x_1 + x_2 - x_3 = 1 \\ -2x_1 + x_2 - 2x_3 = 1 \\ x_1, x_2, x_3 \geq 0 \end{cases}$$

3.4 La dualité

Deux problèmes d'optimisation sont dits en dualité si l'un est un problème de maximisation, l'autre un problème de minimisation et si résoudre l'un équivaut à résoudre l'autre, dans le sens où la solution optimale de l'un se déduit de la solution optimale de l'autre.

On rencontre deux cas de dualité dans ce polycopié. Le premier est décrit dans la section suivante. Le second apparaît dans la section concernant le calcul du flot maximal pouvant transiter par un réseau de transport.

Pouvoir associer un problème dual à un problème d'optimisation fournit en général des outils pour vérifier qu'une solution réalisable est bien une solution optimale. C'est le cas du théorème des écarts complémentaires énoncé dans la section suivante et du théorème « flot maximal et coupe minimale » dans les problèmes de flots maximaux.

3.4.1 Construction du dual d'un programme linéaire

À tout programme linéaire, on peut faire correspondre un autre programme linéaire, *dual* du premier (celui-ci étant appelé *primal*). Cette transformation associe à chaque variable du primal une contrainte du dual et donc une variable d'écart (on peut supposer sans perte de généralité que le programme ne comporte pas de contraintes d'égalité). Elle associe à chaque contrainte du primal une variable du dual. Enfin, la transformation est involutive :

Proposition 15 *Le dual d'un dual est un programme linéaire équivalent au primal.*

On illustre la méthode de construction du dual sur deux exemples sous forme canonique. Cela suffit pour le cas général puisque tout programme linéaire peut être transformé en un programme équivalent sous forme canonique.

Exemple de production de papier

Deux usines produisent du papier de trois qualités différentes. Elles ont des commandes pour chaque type de papier : la compagnie qui gère les usines a des contrats pour fournir 16 tonnes de papier de qualité inférieure, 5 tonnes de papier de qualité moyenne et 20 tonnes de papier de qualité supérieure. Il coûte 1000 euros par jour pour faire fonctionner l'usine *A* et 2000 euros par jour pour l'usine *B*. L'usine *A* produit 8 tonnes de papier de qualité inférieure, 1 tonne de papier de qualité moyenne et 2 tonnes de papier de qualité supérieure par jour. L'usine *B* produit 2 tonnes de papier de qualité inférieure, 1 tonne de papier de qualité moyenne et 7 tonnes de papier de qualité supérieure par jour. On cherche combien de jours chaque usine doit fonctionner afin de satisfaire la demande de la façon la plus économique.

Le programme linéaire est un exemple de programme de *satisfaction de demande*. Les variables x_1 et x_2 représentent les nombres de jours de fonctionnement des usines *A* et *B*.

$$\begin{cases} 1000 x_1 + 2000 x_2 = z & [\text{min}] \\ 8 x_1 + 2 x_2 \geq 16 & (\text{qualité inférieure}) \\ x_1 + x_2 \geq 5 & (\text{qualité moyenne}) \\ 2 x_1 + 7 x_2 \geq 20 & (\text{qualité supérieure}) \\ x_1, x_2 \geq 0. \end{cases}$$

Pour former le programme linéaire dual on commence par former la matrice des coefficients des contraintes et de l'objectif avec seconds membres. On place la ligne de l'objectif en bas.

$$\left(\begin{array}{cc|c} 8 & 2 & 16 \\ 1 & 1 & 5 \\ 2 & 7 & 20 \\ 1000 & 2000 & \end{array} \right)$$

On prend la transposée de cette matrice

$$\left(\begin{array}{ccc|c} 8 & 1 & 2 & 1000 \\ 2 & 1 & 7 & 2000 \\ 16 & 5 & 20 & \end{array} \right)$$

Le dual s'obtient en interprétant la matrice transposée en un programme linéaire. Par rapport au primal, il faut changer le sens des inégalités et la nature de l'objectif :

$$\begin{cases} 16 y_1 + 5 y_2 + 20 y_3 = w & [\text{max}] \\ 8 y_1 + y_2 + 2 y_3 \leq 1000 & (\text{usine } A) \\ 2 y_1 + y_2 + 7 y_3 \leq 2000 & (\text{usine } B) \\ y_1, y_2, y_3 \geq 0 \end{cases}$$

À la variable x_1 du primal correspond la contrainte « usine A » du dual et donc la variable d'écart y_4 . À la variable x_2 du primal correspond la contrainte « usine B » du dual et donc la variable d'écart y_5 . Quelques observations :

1. les variables x_1, x_2 du primal sont différentes des variables y_1, y_2, y_3 du dual ;
2. le primal a trois contraintes et deux variables alors que le dual a deux contraintes et trois variables ;
3. les inégalités du primal sont des \geq alors que celles du dual sont des \leq ;
4. c'est parce que les coefficients de l'objectif du primal sont positifs, que les seconds membres du dual sont positifs ou nuls ;
5. l'objectif du primal est une minimisation alors que l'objectif du dual est une maximisation.

Exemple du boulanger

Prenons pour primal l'exemple du boulanger

$$\begin{aligned} 4000 x_1 + 5000 x_2 &= z & [\text{max}] \\ 2 x_1 + x_2 &\leq 8 & (\text{farine}) \\ x_1 + 2 x_2 &\leq 7 & (\text{beurre}) \\ x_2 &\leq 3 & (\text{sel}) \\ x_1, x_2 &\geq 0 \end{aligned}$$

Son dual s'obtient comme précédemment en formant la matrice des coefficients des contraintes et de l'objectif avec seconds membres et en interprétant sa transposée comme un programme linéaire (en changeant le sens des inégalités et le problème de maximisation en une minimisation).

$$\begin{aligned} 8 y_1 + 7 y_2 + 3 y_3 &= w & [\text{min}] \\ 2 y_1 + y_2 &\geq 4000 & (\text{pâte brisée}) \\ y_1 + 2 y_2 + y_3 &\geq 5000 & (\text{pâte feuilletée}) \\ y_1, y_2, y_3 &\geq 0 \end{aligned}$$

À la variable x_1 du primal correspond la contrainte « pâte brisée » du dual et donc la variable d'écart y_4 . À la variable x_2 du primal correspond la contrainte « pâte feuilletée » du dual et donc la variable d'écart y_5 .

3.4.2 Interprétation du programme dual

Il est toujours possible de donner un sens au programme dual mais l'interprétation obtenue, qui décrit la stratégie d'un concurrent, est souvent assez artificielle, en tous cas pour les programmes modélisant le monde de l'entreprise.

L'exemple de la production de papier

Considérons les contraintes du dual : les coefficients ont la dimension de tonnes de papier par jour ; les seconds membres ont la dimension d'euros par jour. Les variables ont donc obligatoirement la dimension d'euros par tonne de papier.

Voici une interprétation possible.

On a affaire à un concurrent qui tente de faire fermer les deux usines. Le concurrent vend à l'entreprise du papier des trois différentes qualités au prix le plus élevé possible (bien sûr) mais tout en s'assurant que les prix soient compétitifs vis-à-vis des deux usines.

Les variables y_1 , y_2 et y_3 représentent les prix pratiqués par le concurrent pour les trois qualités de papiers. Ces variables ont bien la dimension d'euros par tonne.

L'entreprise va devoir acheter 16 tonnes de papier de qualité inférieure, 5 tonnes de papier de qualité moyenne et 20 tonnes de papier de qualité supérieure. Elle va donc dépenser $16 y_1 + 5 y_2 + 20 y_3$ euros. L'objectif du concurrent consiste à maximiser cette somme

$$16 y_1 + 5 y_2 + 20 y_3 = w[\max]$$

Maintenant, les prix doivent être inférieurs à ce que coûterait la production de cette même quantité de papier par l'usine A : chaque jour de fonctionnement de cette usine coûte 1000 euros à l'entreprise mais lui permet d'économiser $8 y_1 + y_2 + 2 y_3$. Il faut donc que

$$8 y_1 + y_2 + 2 y_3 \leq 1000.$$

Le même raisonnement, tenu pour l'usine B conduit à la contrainte :

$$2 y_1 + y_2 + 7 y_3 \leq 2000.$$

Voici les solutions optimales et les objectifs réalisés à l'optimum. On constate que les objectifs réalisés à l'optimum sont égaux.

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}, \quad z = 7000, \quad \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 600 \\ 200 \end{pmatrix}, \quad w = 7000.$$

L'exemple du boulanger

Considérons les contraintes du dual : les coefficients ont la dimension de kilos par rouleau ; les seconds membres ont la dimension d'euros par milliers de rouleaux. Les variables du dual ont donc la dimension d'euros par milliers de kilos c'est-à-dire d'euros par tonne.

Voici une interprétation possible.

Le programme dual modélise le comportement d'un concurrent du boulanger qui tente de le démotiver. Il lui propose de lui racheter ses matières premières en stock à des prix y_1 pour la farine, y_2 pour le beurre et y_3 pour le sel de telle sorte que ni la production de pâte brisée ni celle de pâte feuilletée ne soit rentable pour le boulanger. L'objectif du concurrent consiste naturellement à minimiser le coût de rachat du stock complet

$$8 y_1 + 7 y_2 + 3 y_3 = w[\min]$$

Comme 2 tonnes de farine et 1 tonne de beurre permettent de produire 1 millier de rouleaux de pâte brisée qui rapportent 4000 euros, les prix de rachat y_1 de la farine et y_2 du beurre doivent donc satisfaire la contrainte

$$2y_1 + y_2 \geq 4000.$$

Comme 1 tonne de farine, 2 tonnes de beurre et 1 tonne de sel permettent de produire 1 millier de rouleaux de pâte feuilletée qui rapporte 5000 euros, les prix de rachat y_1 de la farine et y_2 du beurre doivent satisfaire aussi

$$y_1 + 2y_2 + y_3 \geq 5000.$$

Voici les solutions optimales et les objectifs réalisés à l'optimum. On constate à nouveau que les objectifs réalisés à l'optimum sont égaux.

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}, \quad z = 22000, \quad \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1000 \\ 2000 \\ 0 \end{pmatrix}, \quad w = 22000.$$

3.4.3 Résoudre le primal équivaut à résoudre le dual

Si on admet que le dual d'un programme linéaire modélise l'activité d'un concurrent, le premier point du théorème ci-dessous (le fait que les objectifs réalisés à l'optimum soient égaux) devient assez intuitif.

Théorème 4 *Quand deux programmes linéaires sont en dualité seuls trois cas peuvent se produire.*

1. *Le primal et le dual ont chacun au moins une solution réalisable. Alors les deux programmes sont bornés et réalisent le même objectif à l'optimum (principe de la dualité de Von Neumann).*
2. *L'un des deux programmes linéaires a au moins une solution réalisable mais l'autre pas. Alors celui des deux qui a au moins une solution est non borné.*
3. *Aucun des deux programmes linéaires n'a de solution réalisable.*

Le premier cas est de loin le plus fréquent en pratique. Remarquer que si les objectifs à l'optimum sont les mêmes, les solutions optimales ne le sont pas. Ça n'aurait aucun sens d'ailleurs puisque les variables ont des dimensions différentes.

La proposition suivante montre qu'on peut non seulement lire l'objectif réalisé à l'optimum mais aussi la solution optimale d'un programme linéaire dans le tableau simplicial final de son dual. On a formulé la proposition à partir de l'algorithme du tableau simplicial pour des raisons de lisibilité mais il en existe des formulations plus générales, indépendantes de tout algorithme.

Proposition 16 *À chaque variable du primal x_i correspond une variable d'écart du dual y_j . La valeur de x_i dans la solution optimale du primal est égale à l'opposé $-f_j$ du coût correspondant à y_j dans le tableau simplicial final du dual.*

L'exemple de la production de papier

Le théorème et la proposition fournissent donc une méthode simple de résolution de programmes de satisfaction de demande (une méthode qui évite l'emploi des variables artificielles). Reprenons le premier exemple.

$$\left\{ \begin{array}{ll} 1000 x_1 + 2000 x_2 = z & [\min] \\ 8 x_1 + 2 x_2 \geq 16 & (\text{qualité inférieure}) \\ x_1 + x_2 \geq 5 & (\text{qualité moyenne}) \\ 2 x_1 + 7 x_2 \geq 20 & (\text{qualité supérieure}) \\ x_1, x_2 \geq 0. \end{array} \right.$$

On peut montrer (par exemple en utilisant la technique des variables artificielles) que la solution optimale de ce programme linéaire est

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}.$$

L'objectif réalisé à l'optimum est $z_0 = 7000$. Son dual relève du cas favorable (forme canonique et seconds membres positifs).

$$\left\{ \begin{array}{ll} 16 y_1 + 5 y_2 + 20 y_3 = w & [\max] \\ 8 y_1 + y_2 + 2 y_3 \leq 1000 & (\text{usine A}) \\ 2 y_1 + y_2 + 7 y_3 \leq 2000 & (\text{usine B}) \\ y_1, y_2, y_3 \geq 0 \end{array} \right.$$

On introduit deux variables d'écart y_4 (associée à x_1) et y_5 (associée à x_2). On forme un tableau simplicial

	y_1	y_2	y_3	y_4	y_5	
y_4	8	1	2	1	0	1000
y_5	2	1	7	0	1	2000
	16	5	20	0	0	0

Des calculs pénibles à mener à la main aboutissent au tableau simplicial final :

	y_1	y_2	y_3	y_4	y_5	
y_2	$52/5$	1	0	$7/5$	$-2/5$	600
y_3	$-6/5$	0	1	$-1/5$	$1/5$	200
	-12	0	0	-3	-2	-7000

La solution optimale du dual est donc

$$\bar{y} = \begin{pmatrix} 0 \\ 600 \\ 200 \end{pmatrix}.$$

L'objectif réalisé est $w_0 = 7000$. Le théorème est vérifié. Les variables d'écart y_3 et y_4 sont associées aux variables x_1 et x_2 respectivement. Au signe près, les coûts f_4 et f_5 du tableau simplicial final du dual donnent la solution optimale du primal. La proposition est vérifiée.

L'exemple du boulanger

Reprenons le programme linéaire modélisant l'activité du boulanger et son dual.

$$\left\{ \begin{array}{lcl} 4000 x_1 + 5000 x_2 & = & z \quad [\text{max}] \\ 2 x_1 + x_2 & \leq & 8 \quad (\text{farine}) \\ x_1 + 2 x_2 & \leq & 7 \quad (\text{beurre}) \\ x_2 & \leq & 3 \quad (\text{sel}) \\ x_1, x_2 & \geq & 0 \end{array} \right. \quad \left\{ \begin{array}{lcl} 8 y_1 + 7 y_2 + 3 y_3 & = & w \quad [\text{min}] \\ 2 y_1 + y_2 & \geq & 4000 \quad (\text{pâte brisée}) \\ y_1 + 2 y_2 + y_3 & \geq & 5000 \quad (\text{pâte feuilletée}) \\ y_1, y_2, y_3 & \geq & 0 \end{array} \right.$$

Ici, c'est le primal qui est facile à résoudre. Voici le tableau simplicial final calculé lors d'un cours précédent :

	x_1	x_2	x_3	x_4	x_5	
x_5	0	0	1/3	-2/3	1	1
x_1	1	0	2/3	-1/3	0	3
x_2	0	1	-1/3	2/3	0	2
	0	0	-1000	-2000	0	-22000

Le dual du dual est un programme linéaire équivalent au primal. Les variables y_1 , y_2 et y_3 du dual correspondent aux variables d'écart x_3 , x_4 et x_5 du primal. D'après le théorème et la proposition précédents, on trouve que l'objectif réalisé à l'optimum par le dual est $w_0 = 22000$ et que sa solution optimale est :

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1000 \\ 2000 \\ 0 \end{pmatrix}.$$

3.4.4 Sensibilité de l'objectif à une perturbation des seconds membres

On rappelle qu'à chaque contrainte du primal correspond une variable du dual.

Théorème 5 *Si on augmente d'une unité (suffisamment petite) le second membre d'une contrainte, on augmente l'objectif réalisé d'un nombre d'unités égal à la valeur à l'optimum de la variable duale associée à la contrainte.*

La valeur à l'optimum de la variable duale est appelée valeur marginale de la contrainte.

Lorsqu'une contrainte est inactive, on peut modifier son second membre sans affecter la solution optimale du programme. On en déduit :

Proposition 17 *Si une contrainte est inactive alors sa valeur marginale est nulle.*

Le texte qui suit justifie le théorème, sans le démontrer. Supposons par exemple, dans le problème de production de papier, que la quantité de papier de qualité moyenne commandée passe de 5 à 6 tonnes. Cette perturbation modifie un des coûts du dual. Elle ne modifie pas les contraintes. Elle ne modifie donc pas le polyèdre des solutions réalisables du dual. Supposer cette modification petite, c'est supposer que le sommet du polyèdre correspondant à la solution

optimale du dual ne change pas. On en conclut que l'objectif réalisé à l'optimum du dual augmente de la valeur à l'optimum de la variable duale y_2 associée à la contrainte.

$$\left\{ \begin{array}{l} 1000 x_1 + 2000 x_2 = z \quad [\min] \\ 8 x_1 + 2 x_2 \geq 16 \\ x_1 + x_2 \geq 56 \\ 2 x_1 + 7 x_2 \geq 20 \\ x_1, x_2 \geq 0. \end{array} \right. \quad \left\{ \begin{array}{l} 16 y_1 + 56 y_2 + 20 y_3 = w \quad [\max] \\ 8 y_1 + y_2 + 2 y_3 \leq 1000 \\ 2 y_1 + y_2 + 7 y_3 \leq 2000 \\ y_1, y_2, y_3 \geq 0 \end{array} \right.$$

Appliquons maintenant le théorème 4. À l'optimum, les objectifs réalisés par le primal et le dual sont égaux. L'objectif réalisé à l'optimum par le primal est donc augmenté de y_2 , c'est-à-dire de la valeur marginale de la contrainte modifiée.

Voici une interprétation possible, basée sur l'algorithme du tableau simplicial de ce qu'on entend par une petite modification : une modification des seconds membres qui n'affecte pas le choix des pivots lors du déroulement de l'algorithme du tableau simplicial est suffisamment petite.

Valeurs marginales en AMPL

Voici le modèle AMPL de l'exemple de production de papier.

```
set USINES;
set PAPIERS;
param qte_commandee {PAPIERS} >= 0;
param cout_quotidien {USINES} >= 0;
param production_quotidienne {USINES, PAPIERS} >= 0;
var jours_ouvres {USINES} >= 0;
minimize cout_total :
    sum {u in USINES} (jours_ouvres [u] * cout_quotidien [u]);
subject to commande_satisfaite {p in PAPIERS} :
    sum {u in USINES}
        jours_ouvres [u] * production_quotidienne [u, p] >= qte_commandee [p];
data;
set USINES := A B;
set PAPIERS := inf moy sup;
param qte_commandee :=
    inf 16
    moy 5
    sup 20;
param cout_quotidien :=
    A 1000
    B 2000;
param production_quotidienne :
    inf moy sup :=
    A 8 1 2
    B 2 1 7;
```

Résolution. On vérifie successivement que la contrainte sur le papier de qualité inférieure est inactive et qu'une augmentation de 10 tonnes de la quantité commandée de papier de qualité supérieure produit une augmentation de $10 \times 200 = 2000$ euros des coûts.

```

ampl: model papier.ampl;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 7000
ampl: let qte_commandee ["inf"] := 17;
ampl: solve;
MINOS 5.5: optimal solution found.
0 iterations, objective 7000
ampl: let qte_commandee ["inf"] := 16;
ampl: let qte_commandee ["sup"] := 30;
ampl: solve;
MINOS 5.5: optimal solution found.
0 iterations, objective 9000

```

On constate enfin qu’une augmentation de 11 tonnes de la quantité commandée de papier de qualité supérieure n’est pas « suffisamment petite ».

```

ampl: let qte_commandee ["sup"] := 31;
ampl: solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 9269.230769

```

Les valeurs des variables duales associées aux contraintes sont appelées *shadow prices* en AMPL. On les obtient par la commande « *display nom_de_la_contrainte* ».

```

ampl: model papier.ampl;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 7000
ampl: display commande_satisfaite;
commande_satisfaite [*] :=
inf      0
moy     600
sup     200

```

Attention : en AMPL, ce sont les bornes des contraintes qui jouent le rôle des seconds membres (on les obtient en accolant les suffixes « lb » et « ub » au nom de la contrainte). Et il y a deux bornes (l’une d’elles pouvant valoir $\pm\infty$). Par convention, le logiciel affiche la valeur marginale correspondant à la borne *la plus proche* du corps de la contrainte (la valeur du corps s’obtient en accolant le suffixe « body » au nom de la contrainte). Comment le logiciel procède-t-il ? Une contrainte AMPL munie de deux bornes se traduit par deux contraintes mathématiques. Le tableau simplicial calculé par le logiciel (ou ce qui en tient lieu) comporte deux variables duales. L’une des deux variables au moins est nulle puisque les deux contraintes mathématiques ne peuvent pas être actives en même temps. Le logiciel choisit celle qui est non nulle, si elle existe.

```

ampl: display commande_satisfaite.lb, commande_satisfaite.body,
                                         commande_satisfaite.ub;
commande_satisfaite.lb commande_satisfaite.body commande_satisfaite.ub :=
inf          16          28          Infinity
moy           5           5          Infinity
sup          20          20          Infinity

```

Des interprétations similaires s’appliquent à ce que le logiciel AMPL appelle les *coûts réduits* des variables.

3.4.5 Critères d'optimalité

On conclut par deux théorèmes permettant de décider si une solution réalisable est optimale ou pas. On montre comment appliquer en pratique le théorème des écarts complémentaires.

Théorème 6 Soient \bar{x} et \bar{y} deux solutions réalisables de deux programmes linéaires en dualité, d'objectifs économiques notés f et g respectivement.

Alors ${}^t f \bar{x} = {}^t g \bar{y}$ si et seulement si \bar{x} et \bar{y} sont des solutions optimales.

Le théorème suivant est connu sous le nom de « théorème des écarts complémentaires » ou « théorème des relations d'exclusion ».

Théorème 7 (théorème des écarts complémentaires)

On considère un programme linéaire comportant n variables x_1, \dots, x_n et m contraintes. Son dual comporte m variables y_1, \dots, y_m et n contraintes. À chaque variable x_i correspond une variable d'écart y_{m+i} . À chaque variable y_i correspond une variable d'écart x_{n+i} .

Soient \bar{x} une solution réalisable du premier et \bar{y} une solution réalisable du second.

Les solutions réalisables \bar{x} et \bar{y} sont optimales si et seulement si $\bar{x}_i \bar{y}_{m+i} = \bar{y}_j \bar{x}_{n+j} = 0$ pour tous $1 \leq i \leq n$ et $1 \leq j \leq m$.

Exemple d'utilisation

Le théorème des écarts complémentaires permet de déterminer si une solution réalisable \bar{x} d'un programme linéaire est optimale ou pas. On peut l'utiliser ainsi : on part de la solution réalisable \bar{x} ; ensuite on pose que tous les produits $x_i y_j$ mentionnés dans le théorème sont nuls ; on en déduit des conditions (un système d'équations) sur une solution du dual ; on résout ce système ; s'il admet une solution \bar{y} et si cette solution est bien réalisable alors \bar{x} et \bar{y} satisfont les hypothèses du théorème et \bar{x} est prouvée optimale. Voici la démarche illustrée sur l'exemple de production de papier.

Oublions temporairement que nous avons déjà démontré à de nombreuses reprises que

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

est la solution optimale du programme linéaire modélisant l'exemple de production de papier et démontrons-le au moyen du théorème des écarts complémentaires. En substituant ces valeurs dans les équations et en comparant les valeurs obtenues aux seconds membres, on obtient les valeurs des variables d'écart. Voici le programme sous forme standard et les valeurs de toutes les variables. Toutes les coordonnées de la solution candidate sont positives ou nulles : cette solution est bien réalisable.

$$\left\{ \begin{array}{lcl} 1000 x_1 + 2000 x_2 & = & z \quad [\text{min}] \\ 8 x_1 + 2 x_2 - x_3 & = & 16 \\ x_1 + x_2 - x_4 & = & 5 \\ 2 x_1 + 7 x_2 - x_5 & = & 20 \\ x_1, x_2 & \geq & 0. \end{array} \right. \quad \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 12 \\ 0 \\ 0 \end{pmatrix}.$$

Voici maintenant le dual sous forme standard ainsi que la correspondance entre les variables de chaque programme et les variables d'écart de l'autre.

$$\left\{ \begin{array}{lcl} 16 y_1 + 5 y_2 + 20 y_3 & = & w \quad [\text{max}] \\ 8 y_1 + y_2 + 2 y_3 + y_4 & = & 1000 \\ 2 y_1 + y_2 + 7 y_3 + y_5 & = & 2000 \\ y_1, y_2, y_3 & \geq & 0 \end{array} \right. \quad \begin{array}{lcl} x_1 & \leftrightarrow & y_4 \\ x_2 & \leftrightarrow & y_5 \\ x_3 & \leftrightarrow & y_1 \\ x_4 & \leftrightarrow & y_2 \\ x_5 & \leftrightarrow & y_3 \end{array}$$

On cherche des valeurs pour les variables duales telles que les produits entre les cinq couples de variables soient nuls :

$$\begin{array}{lcl} x_1 y_4 & = & 3 \times y_4 = 0 \\ x_2 y_5 & = & 2 \times y_5 = 0 \\ x_3 y_1 & = & 12 \times y_1 = 0 \\ x_4 y_2 & = & 0 \times y_2 = 0 \\ x_5 y_3 & = & 0 \times y_3 = 0 \end{array}$$

Ces relations impliquent que $y_1 = y_4 = y_5 = 0$ et que y_2 et y_3 satisfont

$$\left\{ \begin{array}{lcl} y_2 + 2 y_3 & = & 1000 \\ y_2 + 7 y_3 & = & 2000 \end{array} \right. \quad \text{solution} \quad \begin{pmatrix} y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 600 \\ 200 \end{pmatrix}.$$

La solution du dual obtenue est réalisable. Les hypothèses du théorème des écarts complémentaires sont satisfaites. La solution candidate du primal est donc bien une solution optimale. On peut vérifier par acquis de conscience que les objectifs réalisés à l'optimum sont égaux. En effet, $z_0 = w_0 = 7000$.

Chapitre 4

Théorie des graphes

Un graphe est une structure de données pouvant représenter des situations très diverses : réseaux d'autoroutes, planning de succession de tâches, courants dans un circuit électrique ... Elle s'enrichit quand on attribue aux sommets ou aux arcs/arêtes une valeur, c'est-à-dire un nombre représentant une longueur, une capacité de transport, une probabilité de transition ... On dit qu'on a affaire à un graphe valué. Dans ce chapitre, nous allons nous concentrer sur les problèmes de théorie des graphes qui sont des problèmes d'optimisation : trouver un chemin de valeur minimale entre deux sommets (on minimise la valeur du chemin), le flot maximal pouvant s'écouler par un réseau de canalisations (on maximise un flot) ... Ce chapitre doit beaucoup aux livres [4] et [3]. Le premier met l'accent sur la pratique des algorithmes. Le deuxième sur la théorie : les preuves de correction, de complexité et les subtilités d'implantation. Chacun des deux livres est nettement plus complet que ce cours.

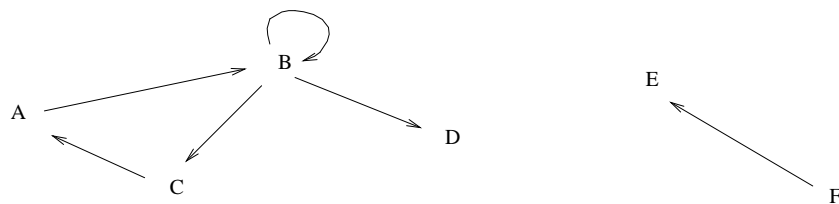
4.1 Vocabulaire

Certaines définitions varient suivant les ouvrages. En général, ces variations sont minimales. On distingue les *graphes orientés* (des *sommets* reliés par des *arcs* ou des *flèches*) des *graphes non orientés* (des sommets reliés par des *arêtes*). Cette distinction n'est pas très fondamentale :

- tout algorithme conçu pour les graphes non orientés peut aussi être appliqué aux graphes orientés : il suffit de ne pas tenir compte du sens des flèches ;
- tout graphe non orienté est équivalent à un graphe orienté : il suffit de remplacer chaque arête par deux arcs.

Formellement, un graphe est un couple $G = (S, A)$ où S est l'ensemble des sommets et $A \subset S \times S$ est l'ensemble des arcs ou des arêtes. Dans le cas d'un graphe orienté, chaque couple $(x, y) \in A$ représente un arc. La première composante x du couple est la *source* (ou l'*origine*) de l'arc. La seconde y en est le *but* (ou la *destination*). Dans le cas d'un graphe non orienté, chaque couple $(x, y) \in A$ représente une arête d'*extrémités* x et y . Dans ce cas, les couples (x, y) et (y, x) désignent la même arête¹. Voici un exemple de graphe orienté avec $S = \{A, B, C, D, E, F\}$ et $A = \{(A, B), (B, B), (B, C), (C, A), (B, D), (F, E)\}$.

1. Certains auteurs [3] interdisent la présence de boucles (x, x) dans les graphes non orientés. Certains autorisent la présence d'arcs et d'arêtes dans un même graphe.



Un graphe $G' = (S', A')$ est un *sous-graphe* de G si $S' \subset S$ et $A' \subset A \cap S' \times S'$. Soit S' un sous-ensemble de S . On dit que $G' = (S', A')$ est le *sous-graphe de G engendré par S'* si $A' = A \cap S' \times S'$.

Soit $G = (S, A)$ un graphe orienté. Un *chemin* dans G est une suite de sommets x_1, \dots, x_k de S (un même sommet pouvant apparaître plusieurs fois) telle que, quel que soit $1 \leq i < k$, le couple $(x_i, x_{i+1}) \in A$.

Soit $G = (S, A)$ un graphe orienté ou non. Une *chaîne* dans G est une suite de sommets x_1, \dots, x_k de S (un même sommet pouvant apparaître plusieurs fois) telle que, quel que soit $1 \leq i < k$, l'un des couples $(x_i, x_{i+1}), (x_{i+1}, x_i) \in A$. Remarque : pour désigner sans ambiguïté une chaîne dans un graphe orienté, il peut être nécessaire de préciser aussi les arcs² (la difficulté se présente dans l'algorithme de Ford–Fulkerson pour désigner les « chaînes améliorantes »).

La *longueur* d'un chemin (ou d'une chaîne) x_1, \dots, x_k est le nombre d'arcs (ou d'arêtes) qui le composent : $k - 1$. Un chemin qui se referme sur lui-même est un *circuit*. Un graphe qui ne comporte aucun circuit est dit *acyclique*. Une chaîne qui se referme sur elle-même est un *cycle*. Un chemin (une chaîne) est *simple* s'il (si elle) ne passe pas deux fois par le même arc (la même arête). Un chemin (une chaîne) est *élémentaire* s'il (si elle) ne passe pas deux fois par le même sommet. Tout chemin (toute chaîne) élémentaire est simple. De tout chemin (chaîne) on peut extraire un chemin (une chaîne) élémentaire ayant même source et même destination (mêmes extrémités).

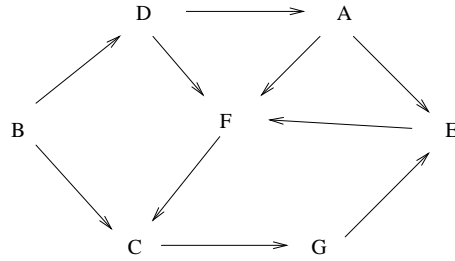
Dans un graphe orienté, un sommet y est *accessible* à partir d'un sommet x s'il existe un chemin de x vers y . Dans un graphe non orienté, un sommet y est *accessible* à partir d'un sommet x s'il existe une chaîne entre x et y . Un graphe orienté G est *fortement connexe* si pour tous sommets x et y il existe un chemin de x vers y et un chemin de y vers x . Un sous-graphe G' d'un graphe orienté G est une *composante fortement connexe* de G si G' est fortement connexe et n'est pas strictement inclus dans un autre sous-graphe fortement connexe de G . Un graphe fortement connexe n'a qu'une seule composante fortement connexe.

La *connexité* est une propriété des graphes orientés ou non. Un graphe G est *connexe* si pour tous sommets x et y il existe une chaîne d'extrémités x et y . Un sous-graphe G' de G est une *composante connexe* de G si G' est connexe et n'est pas strictement inclus dans un autre sous-graphe connexe de G . Dans un graphe non orienté, la composante connexe d'un sommet x est l'ensemble des sommets accessibles à partir de x par contre, dans un graphe orienté, la composante fortement connexe d'un sommet x n'est en général pas l'ensemble des sommets accessibles à partir de x (elle est incluse dans l'ensemble). Le graphe suivant est connexe. La composante fortement connexe de A est $(\{A\}, \emptyset)$. Une autre composante fortement connexe est

$$(\{E, F, C, G\}, \{(E, F), (F, C), (C, G), (G, E)\}).$$

L'ensemble des sommets accessibles à partir de A est $\{A, F, E, C, G\}$.

2. Certains auteurs définissent les chemins et les chaînes comme des suites d'arcs ou d'arêtes. D'autres précisent à la fois les sommets et les arcs ou les arêtes empruntées.



4.2 Représentations d'un graphe

Soit $G = (S, A)$ un graphe avec $S = \{x_1, \dots, x_n\}$ et $A = \{a_1, \dots, a_m\}$. Voici les principales représentations possibles de G .

La représentation sagittale

Elle s'obtient en dessinant le graphe.

La représentation par matrice d'adjacence

Deux sommets x et y sont dits *adjacents* s'il existe un arc de x vers y ou de y vers x (s'il existe une arête entre x et y dans le cas de graphes non orientés). La matrice d'adjacence est une matrice B de taille $n \times n$ indexée par les sommets. Par convention B_j^k vaut 1 si l'arc $(x_j, x_k) \in A$ et 0 sinon. Si le graphe est valué on met des poids plutôt que des 1 dans la matrice. Dans le cas des graphes non orientés, la matrice est symétrique.

La représentation par matrice d'incidence

On dit qu'un sommet x est *incident* à un arc a (ou encore que a est incident à x) si x est la source ou la destination de a (si x est une extrémité de a dans le cas de graphes non orientés). La matrice d'incidence C est de taille $n \times m$. Les indices de ligne sont les sommets. Les indices de colonne sont les arcs (ou les arêtes). Soient x_j un sommet et a_k un arc. Par convention C_j^k vaut -1 si x_j est la source de a_k , 1 si x_j est la destination (mais pas la source) de a_k et 0 sinon. Dans le cas des graphes non orientés, C_j^k vaut 1 si x_j est une extrémité de a_k .

La représentation par liste de successeurs

Il s'agit d'un tableau *succ* de n listes chaînées. Le tableau est indexé par les sommets. La liste *succ*(x_i) contient tous les successeurs de x_i .

La représentation par liste de prédecesseurs

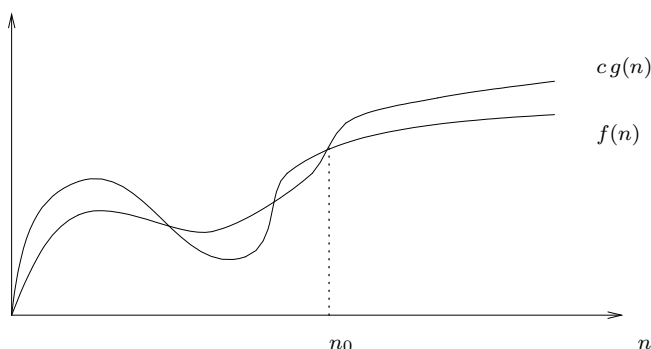
Il s'agit d'un tableau *pred* de n listes chaînées. Le tableau est indexé par les sommets. La liste *pred*(x_i) contient tous les prédecesseurs de x_i .

4.3 Complexité

Fonctions d'une variable.

Définition 16 Soient f et g deux fonctions de \mathbb{R} dans \mathbb{R} . On dit que $f \in O(g)$ s'il existe deux constantes réelles $n_0, c > 0$ telles que $f(n) \leq c g(n)$ pour tout $n > n_0$.

Nous nous servirons de cette notation de la façon suivante : la fonction $f(n)$ désignera le temps de calcul ou le volume mémoire consommé par une procédure en fonction de la taille n de sa donnée ; la fonction g sera une fonction décrite par une expression analytique simple (par exemple, $n, n^2, n \log n, \dots$). La notation « $f \in O(g)$ » signifiera que, pour des données suffisamment grandes, le temps de calcul (ou le volume mémoire) consommé par la procédure croît moins vite que $g(n)$ (à une constante multiplicative près). On ne s'intéresse pas au comportement de la procédure pour de petites valeurs de n . Le graphique suivant illustre l'expression « $f \in O(g)$ » de façon intuitive.



L'exemple des logarithmes. On rappelle que si $n = 2^k$ alors k est le *logarithme en base 2* de n , noté $\log_2(n)$. Plus généralement, si b est un réel positif et $n = b^k$ alors k est le *logarithme en base b* de n , noté $\log_b(n)$.

Tous les logarithmes de n sont égaux à une constante multiplicative près. En effet, $\log_b(n) = \ln(n) / \ln(b)$ où « \ln » désigne le classique *logarithme neperien*. Par conséquent, lorsqu'on écrit qu'une opération a une complexité en $O(\log n)$, on ne précise pas la base.

Considérons par exemple le cas d'une opération compliquée, composée de k opérations élémentaires lorsque la donnée est de taille $n = 2^k$. Comme $k = \log_2(n)$, la complexité de l'opération compliquée, exprimée en fonction de la taille de la donnée, c'est-à-dire n , est un $O(\log n)$.

Fonctions de deux variables. La complexité en mémoire (d'une structure de données ou d'un algorithme) ou en temps (d'un algorithme) est toujours exprimée en fonction de quantités « naturelles » associées à la donnée. Dans le cadre de la théorie des graphes, ces quantités sont le nombre de sommets n et le nombre d'arcs m . Il nous faut donc étendre la définition précédente aux fonctions de deux variables réelles.

Définition 17 Soient f et g deux fonctions de \mathbb{R}^2 dans \mathbb{R} . On dit que $f \in O(g)$ s'il existe trois constantes $n_0, m_0, c > 0$ telles que $f(n, m) \leq c g(n, m)$ pour tous $n > n_0$ et $m > m_0$.

Dans le cas des graphes, les deux variables n et m ne sont pas indépendantes : on a $m \leq n^2$ avec égalité dans le cas des graphes complets et $n - 1 \leq m$ dès qu'on suppose le graphe connexe. Ainsi,

$$\begin{array}{ll} O(m) & \subset O(n^2) \text{ dans tous les cas} \\ O(n) & \subset O(m) \text{ pour tous les graphes connexes} \end{array}$$

Complexité des représentations

La taille de la représentation du graphe varie en fonction de la représentation utilisée. Dans le cas d'une matrice d'adjacence, c'est un $O(n^2)$, dans le cas d'une matrice d'incidence, c'est un $O(nm)$ et dans le cas de listes de successeurs ou prédécesseurs, c'est un $O(m + n)$ (il y a autant de maillons que d'arcs, c'est-à-dire m , plus la taille de *succ*, c'est-à-dire n fois la taille d'un pointeur). La matrice d'adjacence convient aux graphes presque complets. La matrice d'incidence convient aux graphes très creux (cas où $m \ll n$).

Complexité des algorithmes

On s'intéresse ici à la complexité en temps dans le pire des cas. Les exemples types ci-dessous apparaissent dans beaucoup d'algorithmes étudiés dans ce chapitre.

Parcours de tous les successeurs de tous les sommets.

```
pour tout sommet  $x$  faire
  pour tout successeur  $y$  de  $x$  faire
    instruction ayant un coût constant
  fait
fait
```

Si on suppose le graphe représenté par des listes de successeurs, l'instruction est exécutée m fois, moins si, par exemple, on se restreint à des sommets x appartenant à une même composante connexe. La complexité en temps dans le pire des cas est dans tous les cas un $O(m)$. Si on suppose le graphe représenté par une matrice d'adjacence, l'instruction est toujours exécutée m fois mais la complexité en temps dans le pire des cas est un $O(n^2)$.

Parcours de tous les sommets puis de tous les arcs.

```
pour tout sommet  $x$  faire
  instruction ayant un coût constant
fait
pour tout arc  $a$  faire
  instruction ayant un coût constant
fait
```

La première boucle a une complexité $f_1(n, m) \in O(n)$. En supposant le graphe représenté par des listes de successeurs, la seconde a une complexité $f_2(n, m) \in O(m)$. Le tout a donc une complexité $f_1(n, m) + f_2(n, m) \in O(n + m)$. Maintenant, si on suppose le graphe connexe $O(n + m) = O(m)$.

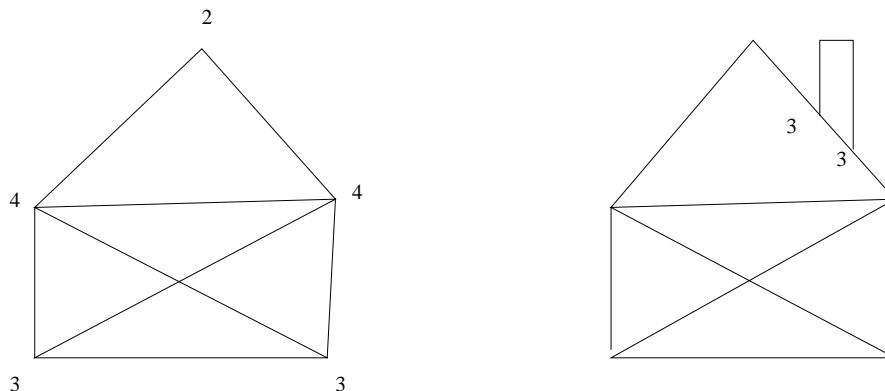
4.4 Exemples de problèmes rencontrés

Historiquement, on date le début de la théorie des graphes avec le problème des sept ponts de Königsberg (posé et résolu par Euler en 1736) qui enjambent la « Pregel ».



FIGURE 4.1 – Leonhard Euler (1707–1783).

Variantes du même problème : peut-on dessiner la figure ci-dessous sans lever le crayon et en ne traçant qu’une seule fois chaque arête (peut-on trouver un circuit *eulérien* qui passe par tous les sommets) ?



Théorème 8 (Euler)

Dans un graphe connexe, il existe un chemin eulérien passant par tous les sommets si et seulement si tous les sommets sont de degré³ pair sauf éventuellement deux qui doivent alors être choisis comme sommets de départ et d’arrivée.

3. Le degré d’un sommet est le nombre d’arêtes admettant le sommet pour extrémité.



FIGURE 4.2 – Carte historique de Königsberg datant de l’époque d’Euler.

En 1852, Francis Guthrie, cartographe anglais, remarque qu’il est possible de colorier la carte des cantons d’Angleterre avec quatre couleurs sans attribuer la même couleur à deux cantons limitrophes. Il demande à son frère Frederick si cette propriété est vraie en général. Celui-ci communique la conjecture à De Morgan et, en 1878, Cayley la publie. Après plusieurs « fausses preuves » la première preuve véritable est établie en 1976 par deux Américains (Appel et Haken). La démonstration, qui a exigé la vérification au cas par cas d’environ 1500 cas particuliers est l’exemple le plus important de preuve effectuée par ordinateur. Elle ne peut pas être vérifiée à la main. La question de l’existence d’une démonstration « à visage humain » est toujours ouverte.

Théorème 9 (*Appel et Haken*)

Toute carte de géographie peut être coloriée avec quatre couleurs sans que la même couleur soit attribuée à deux pays limitrophes.

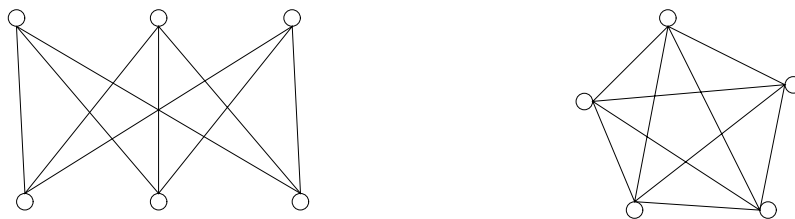
Les graphes planaires (ceux qui peuvent être dessinés sans que deux arêtes se coupent) ont été fortement étudiés. Le théorème suivant est dû à Euler

Théorème 10 (*Euler*)

Soit G un graphe planaire et connexe à n sommets, m arêtes et f faces. Alors $n - m + f = 2$.

Ce théorème, qui se démontre assez facilement par récurrence sur m , implique que les deux graphes ci-dessous ne sont pas planaires. Le premier est le graphe $K_{3,3}$ obtenu à partir de deux

ensembles de trois sommets, en reliant chaque sommet du premier ensemble à chaque sommet du second. Le second graphe, K_5 , est formé de cinq sommets reliés deux-à-deux.



C'est « l'autre implication » qui est difficile dans le théorème suivant, démontré par Kuratowski en 1930.

Théorème 11 (*Kuratowski*)

Un graphe est planaire si et seulement si aucun de ses sous-graphes ne se réduit⁴ à l'un des deux graphes ci-dessus.

4.5 Algorithmes de parcours

Les deux algorithmes ci-dessous généralisent les algorithmes de parcours d'arbres à partir d'une racine. On montre dans les deux cas comment adapter les algorithmes pour construire un arbre couvrant du graphe. La difficulté consiste (par comparaison avec les algorithmes de parcours d'arbres) à éviter de considérer plusieurs fois les mêmes sommets. On résout le problème en coloriant les sommets parcourus.

4.5.1 Parcours « en largeur d'abord »

Il s'obtient au moyen d'une file. On utilise trois couleurs : bleu, vert et rouge.

Les propriétés suivantes sont des invariants de boucle :

1. un sommet est vert si et seulement s'il est dans la file ;
2. si x est rouge alors tous ses voisins sont verts ou rouges ;
3. si x est vert ou rouge alors il existe une chaîne entre s et x ne passant que par des sommets rouges (à l'exception de x).

À la fin de l'exécution de la procédure, la composante connexe de s est l'ensemble des sommets rouges (combiner les invariants avec la condition d'arrêt, qui implique qu'il n'y a plus de sommet vert). On peut modifier facilement la procédure pour tenir compte des orientations. À la fin, l'ensemble des sommets rouges est l'ensemble des sommets accessibles à partir de s .

Complexité. On s'intéresse à la complexité en temps de la procédure. On rappelle que le graphe comporte n sommets et m arêtes. Pour pouvoir mener le calcul, il faut préciser la représentation du graphe et l'implantation des structures de données utilisées (ici, une file). On suppose que le graphe est représenté par des listes de voisins. Par conséquent, parcourir tous les successeurs de tous les sommets a une complexité en $O(m)$. On décrit dans un des paragraphes suivants une implantation d'une file pour laquelle toutes les opérations ont une complexité en

4. en un sens qu'il faudrait préciser.

procédure en_largeur (G, s)

Parcours en largeur d'abord de la composante connexe de s , sans tenir compte des orientations.

début

colorier en bleu tous les sommets sauf s ← unit. $O(1)$ total $O(n)$

vider la file

colorier s en vert et l'enfiler

tant que la file n'est pas vide faire

défiler x ← unit. $O(1)$ total $O(n)$

pour tout voisin y de x faire

si y est bleu alors

colorier y en vert et l'enfiler ← unit. $O(1)$

fini

fait

colorier x en rouge

fait

fin

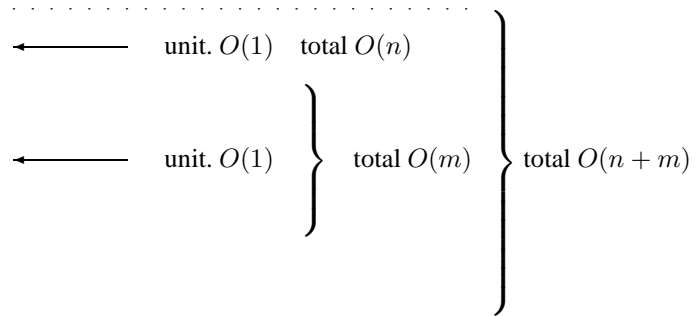


FIGURE 4.3 – Algorithme de parcours en largeur d'abord

$O(1)$. Le corps de la boucle intérieure a donc au total une complexité en temps, dans le pire des cas, en $O(m)$. Chaque sommet de la composante connexe de s est enfilé et défilé une fois. Le coût total des opérations « enfiler » est déjà compté dans le $O(m)$. Le coût total des opérations « défiler » est en $O(n)$. La boucle extérieure a donc une complexité en temps dans le pire des cas en $O(n + m)$. Avec les initialisations, on obtient au final une complexité en temps dans le pire des cas en $O(n + m)$.

Application : calcul d'une plus courte chaîne

On peut modifier la procédure pour calculer explicitement une plus courte chaîne (en nombre d'arcs) entre s et tout autre sommet x accessible à partir de s . L'algorithme est donné en figure 4.4. Pour tout x appartenant à la composante connexe, on calcule un prédécesseur $\text{pred}(x)$ de x dans cette chaîne ainsi que la longueur $d(x)$ de cette chaîne. On obtient ainsi un *arbre couvrant* de la composante connexe de s . On adopte les invariants de boucle supplémentaires suivants :

1. si x est vert ou rouge alors $\text{pred}(x)$ est rouge, fournit le prédécesseur de x dans une chaîne de longueur minimale, égale à $d(x)$, entre s et x .

On peut facilement modifier cet algorithme pour tenir compte des orientations.

Implantation d'une file

On plante facilement une file utilisable pour le parcours en largeur d'abord en utilisant un tableau T à n éléments indicés de 0 à n (il y a $n + 1$ emplacements pour distinguer la file vide d'une file comportant n éléments) et deux indices d et f . L'indice d est l'indice de l'élément en début de file. L'indice f est l'indice de l'emplacement qui suit le dernier élément enfilé. La file est vide si $d = f$. On incrémente ces deux indices « modulo $n + 1$ ».

procédure plus_courte_chaîne (G, s)

Plus courte chaîne entre s et tout autre sommet sans tenir compte des orientations

début

colorier en bleu tous les sommets sauf s

vider la file

$d(s) := 0$

pred(s) est indéfini

colorier s en vert et l'enfiler

tant que la file n'est pas vide **faire**

 défiler x

pour tout voisin bleu y de x **faire**

 colorier y en vert et l'enfiler

$d(y) := d(x) + 1$

 pred(y) := x

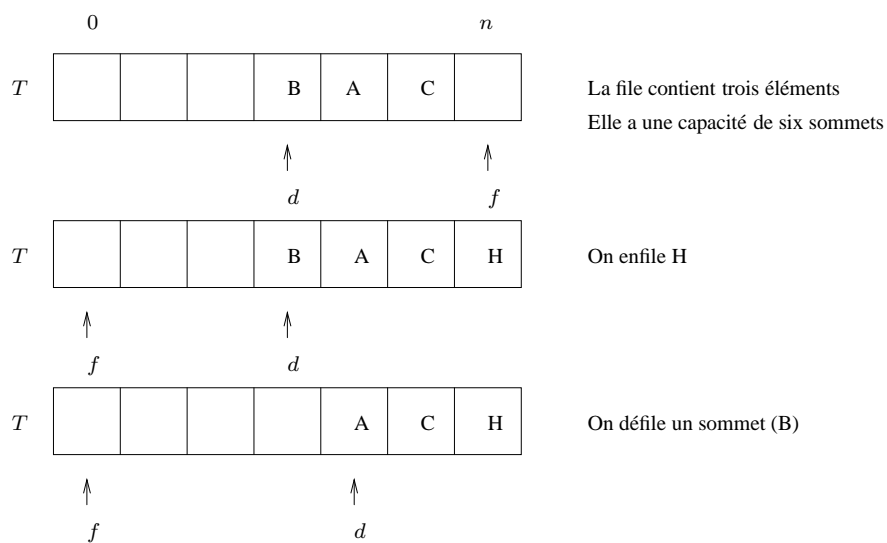
fait

 colorier x en rouge

fait

fin

FIGURE 4.4 – Le calcul d'une plus courte chaîne entre un sommet distingué et tous les autres sommets est une application immédiate du parcours en largeur d'abord



4.5.2 Parcours « en profondeur d'abord »

On donne dans la figure 4.5 le parcours de graphe orienté. Il s'obtient au moyen d'une pile. On utilise trois couleurs : bleu, vert et rouge. On adopte les invariants de boucle suivants :

1. un sommet est vert si et seulement s'il est dans la pile ;
2. un sommet rouge a des successeurs verts ou rouges ;
3. la pile contient un chemin entre s et le sommet de pile.

```

procédure en_profondeur ( $G, s$ )
Parcours en profondeur d'abord des sommets accessibles à partir de  $s$ .
début
    colorier tous les sommets en bleu sauf  $s$ 
    vider la pile
    colorier  $s$  en vert et l'empiler
    tant que la pile n'est pas vide faire
         $x :=$  le sommet de la pile
        si  $x$  a un voisin bleu  $y$  alors
            colorier  $y$  en vert et l'empiler
        sinon
            colorier  $x$  en rouge
            dépiler  $x$ 
        finsi
    fait
fin

```

FIGURE 4.5 – Algorithme de parcours profondeur d'abord

Le traitement d'un sommet commence au moment où il est colorié en vert. Il se termine lorsqu'il est colorié en rouge. La proposition suivante énonce une propriété clef du parcours en profondeur. Cette propriété s'applique directement pour trier topologiquement un graphe acyclique.

Proposition 18 (*propriété clef du parcours en profondeur d'abord*)

Soit x un sommet qui vient juste d'être colorié en vert. Tous les sommets bleus accessibles à partir de x seront coloriés en rouge avant que x ne le soit.

À la fin de l'exécution, l'ensemble des sommets rouges est l'ensemble des sommets accessibles à partir de s . On peut modifier facilement l'algorithme pour ne pas tenir compte des orientations. On peut adapter l'algorithme pour calculer un arbre couvrant de l'ensemble des sommets accessibles à partir de s mais les chemins ainsi obtenus ne sont pas de longueur minimale (en parcourant le graphe en profondeur d'abord, on néglige des chaînes « latérales » plus courtes). C'est un inconvénient vis-à-vis du parcours en largeur d'abord. Avantage : l'arbre couvrant engendré par le parcours en profondeur d'abord n'a pas besoin d'être explicité : on le lit dans la pile.

Pour pouvoir mener l'analyse de complexité, il faut préciser comment on détermine si « x a un voisin bleu y ». C'est ce qui est fait dans l'algorithme de la figure suivante. On suppose le graphe représenté par des listes de successeurs. On attribue à tout sommet x un indice $i(x)$ égal à l'indice du dernier successeur de x traité dans la liste $\text{succ}(x)$. On note $\text{succ}(i, x)$ le i ème élément de la liste $\text{succ}(x)$ des successeurs de x et $|\text{succ}(x)|$ la longueur de cette liste. Les indices commencent à 1.

Complexité. La boucle intérieure parcourt tous les successeurs de tous les sommets. Comme le graphe est supposé représenté par des listes de successeurs, elle est exécutée au total $O(m)$

fois. Chaque sommet accessible à partir de s est empilé et dépilé exactement une fois et chaque opération de pile a une complexité en $O(1)$. La complexité en temps dans le pire des cas de l'ensemble des opérations de pile est donc en $O(n)$. L'initialisation a une complexité en $O(n)$. Au total, l'algorithme a une complexité en temps, dans le pire des cas, en $O(n + m)$.

procédure en_profondeur (G, s)

Parcours en profondeur d'abord des sommets accessibles à partir de s .

début

pour tout sommet x faire	}	total $O(n)$
colorier x en bleu		
$i(x) := 0$		

fait

vider la pile

colorier s en vert et l'empiler

tant que la pile n'est pas vide faire

$x :=$ le sommet de la pile	←	unit. $O(1)$	total $O(n)$
-----------------------------	---	--------------	--------------

$i(x) := i(x) + 1$

trouvé := faux

tant que $i(x) \leq |succ(x)|$ et non *trouvé* faire

 si $succ(i(x), x)$ est bleu alors

trouvé := vrai

 sinon

$i(x) := i(x) + 1$

 finsi

fait

si *trouvé* alors

 colorier $succ(i(x), x)$ en vert et l'empiler

sinon

 colorier x en rouge

 dépiler x

fin

fait

fin

Application : détermination des sommets d'articulation

Un sommet d'articulation d'un graphe est un sommet dont la suppression rend le graphe non connexe. Leur détermination a plusieurs applications. Par exemple, ils font partie des chemins critiques rencontrés dans la méthode MPM. On peut montrer que la racine s est un sommet d'articulation si et seulement si s est le prédécesseur de deux sommets distincts dans l'arbre couvrant engendré par le parcours en profondeur d'abord du graphe, vu comme un graphe non orienté.

Application : tri topologique d'un graphe acyclique

C'est une application très importante. Les graphes acycliques sont des graphes plus compliqués que les arbres mais qui ne présentent pas toute la difficulté des graphes généraux. On les rencontre dans de nombreuses applications (chemins de valeur minimale, méthode MPM). Pour ce type de graphes, on peut souvent concevoir des algorithmes qui s'arrangent pour traiter tous les prédécesseurs du sommet courant avant de traiter ce sommet. On obtient ainsi des algorithmes beaucoup plus simples et efficaces que pour les graphes généraux. C'est le tri topologique qui donne l'ordre suivant lequel traiter les sommets.

On considère pour simplifier un graphe dont tous les sommets sont accessibles à partir de s . La première proposition permet de décider si le graphe comporte un ou plusieurs circuits. La seconde permet de trier topologiquement les sommets d'un graphe acyclique.

Proposition 19 *Un graphe comporte un circuit si et seulement si, lors d'un parcours en profondeur, l'un des successeurs du sommet x en haut de pile est vert.*

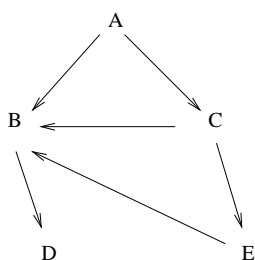
Preuve L'implication \Leftarrow . On suppose qu'un des successeurs y du sommet x en haut de pile est vert et on montre que le graphe comporte un circuit. Les sommets x et y sont verts. Ils sont donc dans la pile et il existe un chemin de y vers x . Ce chemin forme un circuit avec l'arc qui va de x vers y .

L'implication \Rightarrow . On suppose que le graphe comporte un circuit et on montre qu'à une itération de l'algorithme, l'un des successeurs du sommet en haut de pile est vert. Soit \mathcal{C} un circuit de G et y le premier sommet de \mathcal{C} rencontré lors d'un parcours en profondeur. Le sommet y est vert. Tous les autres sommets du circuit sont encore bleus. D'après la proposition 18, ils seront traités et donc coloriés en vert avant que y ne soit colorié en rouge. Ce sera le cas en particulier du sommet x qui précède y dans \mathcal{C} . Lorsque x sera en sommet de pile, le sommet vert y sera détecté. \square

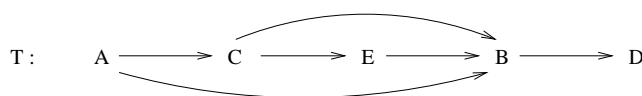
Le tri topologique des sommets d'un graphe acyclique G consiste à ranger les sommets dans un tableau T indicé de 1 à n de telle sorte que, quel que soit $1 \leq i \leq n$, tous les prédécesseurs dans G du i ème élément de T figurent dans T à des indices strictement inférieurs à i . Le tri topologique d'un graphe G peut être vu comme un alignement de tous les sommets le long d'une droite horizontale de telle sorte que tous les arcs de G soient orientés de la gauche vers la droite.

Le tri topologique n'a pas de sens pour les graphes non orientés ou les graphes avec circuits.

Proposition 20 *Pour trier topologiquement un graphe acyclique, il suffit d'enregistrer les sommets dans T au fur et à mesure qu'ils sont coloriés en rouge lors d'un quelconque parcours en profondeur. Le tableau T doit être rempli de la droite vers la gauche : en partant de l'indice n jusqu'à l'indice 1.*



Tri topologique d'un graphe acyclique



Le tri topologique a la même complexité en temps dans le pire des cas que le parcours en profondeur, c'est-à-dire $O(n + m)$.

Remarque pratique : pour trier topologiquement un graphe, il n'est pas nécessaire de connaître à l'avance le sommet s passé en paramètre à l'algorithme de parcours en profondeur. On peut le choisir arbitrairement. Si certains sommets du graphe sont encore bleus après le parcours, il suffit de poursuivre le remplissage du tableau T en choisissant arbitrairement un autre sommet s parmi les sommets bleus et en rappelant l'algorithme de parcours.

4.5.3 Calcul du nombre de composantes connexes d'un graphe

La fonction donnée en figure 4.6, permet de compter le nombre de composantes connexes d'un graphe. Il faut adapter un peu les procédures précédentes, qui ne doivent plus initialiser la couleur des sommets. Ce sont les versions pour graphes non orientés qui doivent être appelées.

```

fonction nombre_de_composantes_connexes ( $G$ )
début
    colorier tous les sommets en bleu
    compteur := 0
    tant que il existe un sommet bleu  $s$  faire
        en_largeur ( $G, s$ ) (ou en_profondeur ( $G, s$ ))
        compteur := compteur + 1
    fait
    retourner compteur
fin

```

FIGURE 4.6 – Le calcul du nombre de composantes connexes d'un graphe est une application immédiate des algorithmes génériques de parcours

4.6 Recherche d'un chemin de valeur minimale

Il faut distinguer la recherche d'un chemin de valeur minimale de la recherche du plus court chemin en nombre d'arcs ou d'arêtes (traité plus haut) qui n'en est qu'un cas particulier. On s'intéresse dans cette section à des graphes valués c'est-à-dire dont les arcs a (les arêtes) sont munis de valeurs $v(a)$. La valeur d'un chemin (d'une chaîne et plus généralement d'un sous-graphe) est définie comme la somme des valeurs des arcs (arêtes) qui le composent. La recherche d'un chemin de valeur minimale n'a pas de sens dans le cas où le graphe comporte un *circuit absorbant* c'est-à-dire un circuit dont la valeur est négative. Dans ce cas, le chemin de valeur minimale est infini et a pour valeur $-\infty$. On ne décrit que deux algorithmes qui ne s'appliquent pas dans tous les cas mais souvent en pratique : les algorithmes de Bellman et de Dijkstra. Parmi les grands absents de cette section, mentionnons l'algorithme de Ford qui s'applique dans tous les cas. Tous ces algorithmes déterminent un chemin de valeur minimale entre un sommet s donné et tous les sommets accessibles à partir de s . Le sommet s est appelé

une *racine*. On montre en fin de section que le problème du calcul d'un chemin de valeur minimale *entre deux sommets donnés* est la solution optimale d'un programme linéaire en variables réelles.

4.6.1 Propriétés des chemins de valeur minimale

Définition 18 Soit x un sommet quelconque d'un graphe valué G . On note $\pi(x)$ la valeur d'un chemin (d'une chaîne) de valeur minimale de s à x (entre s et x). Si x n'est pas accessible à partir de s on pose $\pi(x) = +\infty$.

Proposition 21 S'il existe un arc de y vers x alors $\pi(x) \leq \pi(y) + v(y, x)$.

Proposition 22 Tout sous-chemin d'un chemin de valeur minimale est un chemin de valeur minimale.

Proposition 23 Pour tout sommet x on a $\pi(x) = \min_{y \in \text{pred}(x)} (\pi(y) + v(y, x))$.

4.6.2 Cas des graphes sans circuits : l'algorithme de Bellman



FIGURE 4.7 – Richard Ernest Bellman (1920–1984).

Il s'applique aux graphes orientés acycliques. L'algorithme de Bellman met en application un schéma d'algorithme général applicable dans de nombreux contextes aux graphes sans circuit : *avant de traiter un sommet x , on peut s'arranger pour traiter tous les prédecesseurs de x* . Pour mettre ce schéma en application, il suffit de trier topologiquement les sommets du graphe. L'algorithme de Bellman, donné en figure 4.8, calcule $\pi(x)$ pour tout $x \in S$ en appliquant directement la proposition 23. Pour des raisons de lisibilité, on distingue la valeur $\pi(x)$ calculée de la valeur théorique $\pi(x)$ recherchée.

```

procédure Bellman ( $G, s$ )
début
    trier topologiquement les sommets du graphe  $G$ 
     $\text{pi}(s) := 0$ 
     $\text{pred}(s)$  est indéfini
    pour tous les sommets  $x$  en suivant l'ordre topologique faire
        déterminer un prédécesseur  $y$  de  $x$  tel que  $\text{pi}(y) + v(y, x)$  soit minimal
         $\text{pred}(x) := y$ 
         $\text{pi}(x) := \text{pi}(y) + v(y, x)$ 
    fait
fin

```

FIGURE 4.8 – L'algorithme de Bellman

Complexité. On suppose le graphe représenté à la fois par des listes de successeurs et des listes de prédécesseurs. Le tri topologique a une complexité en $O(n + m)$. La boucle parcourt tous les prédécesseurs de tous les sommets. Elle a donc une complexité en temps dans le pire des cas en $O(m)$. La complexité en temps totale est donc un $O(n + m)$.

4.6.3 Cas des graphes valués positivement : l'algorithme de Dijkstra

L'algorithme de Dijkstra s'applique dans le cas de graphes avec circuits. Il suffit que tous les arcs aient des valeurs positives ou nulles.



FIGURE 4.9 – Edsger Wybe Dijkstra (1930–2002).

Le principe de l'algorithme

On suppose que $\pi(x)$ est connu pour tous les sommets rouges x (le traitement est fini pour eux). On colorie en vert les voisins des sommets rouges (ils sont en cours de traitement) et en bleu les autres sommets, qu'on n'a pas encore commencé à traiter.

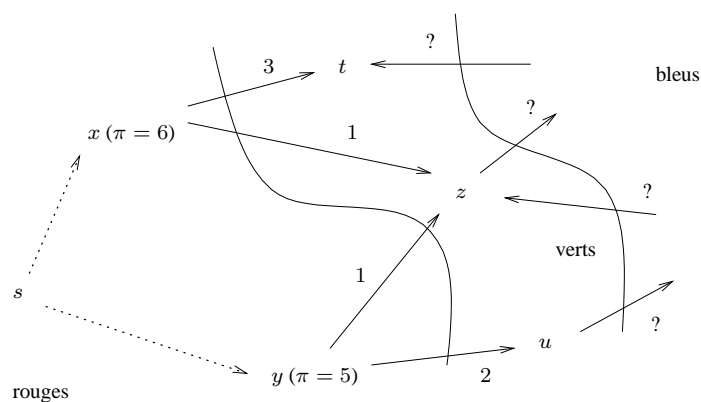
Sur le graphique ci-dessous, on suppose représentés tous les sommets rouges et tous les sommets verts.

Question : y a-t-il un sommet vert qui puisse être colorié en rouge ?

Réponse : oui, z , car $\pi(z) = 6$.

Pourquoi ? Parce que tous les autres chemins de s vers z commencent par des sous-chemins, passant par t ou u , de valeurs supérieures à 6 et donc, *parce que le graphe est valué positivement*, les valeurs de ces autres chemins sont elles-aussi supérieures à 6.

De façon générale, il suffit à chaque itération de choisir le chemin de valeur minimale parmi les chemins partant de s , n'empruntant que des sommets rouges et aboutissant à un sommet vert.



L'algorithme de Dijkstra est *glouton* : il effectue un choix *local*. Le chemin choisi est obtenu par sélection dans un sous-ensemble de l'ensemble de tous les chemins possibles (les chemins qui passent par les sommets bleus ne sont pas considérés). La notion d'algorithme glouton peut se formaliser. Voir [3, chapitre 16].

Une réalisation

On s'intéresse à l'algorithme donné en figure 4.10. En même temps que $\pi(x)$, on calcule un prédécesseur $\text{pred}(x)$ de chaque sommet x dans un chemin de valeur minimale de s vers x .

On attribue trois couleurs aux sommets. On associe une variable $\text{pi}(x)$ à tout sommet x . Pour des raisons de lisibilité, on distingue la valeur $\text{pi}(x)$ calculée par l'algorithme de la valeur théorique $\pi(x)$ qu'on cherche. La valeur de pi des sommets verts est calculée incrémentalement. Les sommets bleus sont ceux pour lesquels aucune valeur de pi n'est encore connue.

Définition 19 On dit qu'un chemin d'un sommet s vers un sommet z est un chemin rouge s'il ne passe que par des sommets rouges (à l'exception de z éventuellement).

On adopte les invariants de boucle suivants :

1. si z est un sommet vert ou rouge alors $\text{pi}(z)$ est la valeur minimale des chemins rouges de s vers z et $\text{pred}(z)$ donne le prédécesseur de z dans un tel chemin ;

```

procédure Dijkstra ( $G, s$ )
début
  colorier tous les sommets sauf  $s$  en bleu
  colorier  $s$  en vert
   $\text{pi}(s) := 0$ 
   $\text{pred}(s)$  est indéfini
  tant que il existe au moins un sommet vert faire
    parmi les sommets verts, en choisir un,  $x$ , tel que  $\text{pi}(x)$  soit minimal
    colorier  $x$  en rouge
    pour tous les successeurs  $y$  de  $x$  faire
      si  $y$  est bleu ou ( $y$  est vert et  $\text{pi}(y) > \text{pi}(x) + v(x, y)$ ) alors
        colorier  $y$  en vert
         $\text{pi}(y) := \text{pi}(x) + v(x, y)$ 
         $\text{pred}(y) := x$ 
      finsi
    fait
  fait
fin

```

FIGURE 4.10 – Première version de l’algorithme de Dijkstra

2. un sommet rouge a des successeurs verts ou rouges.

Il est évident que l’algorithme s’arrête. Supposons les invariants corrects. Lorsque l’algorithme s’arrête, tous les sommets accessibles à partir de s sont rouges. D’après la proposition 23 on a $\text{pi}(z) = \pi(z)$ pour tout sommet z . Il suffit donc de prouver les invariants. On les suppose donc satisfaits au début d’une itération. Pour montrer qu’ils sont à nouveau vrais au début de l’itération suivante, il suffit de montrer la proposition ci-dessous.

Lemme 1 *Tout chemin de valeur minimale de s vers un sommet vert quelconque passe par un sommet vert z tel que $\text{pi}(z) = \pi(z)$.*

Preuve Considérons en effet un chemin de valeur minimale de s vers un sommet vert quelconque et notons z le premier sommet vert rencontré sur ce chemin. Le chemin de s vers z est rouge et de valeur minimale. D’après le premier invariant, $\text{pi}(z) = \pi(z)$. \square

Proposition 24 *Le sommet x sélectionné par l’algorithme vérifie $\text{pi}(x) = \pi(x)$.*

Preuve Considérons un chemin de valeur minimale de s vers le sommet vert x sélectionné par l’algorithme. Ce chemin passe, d’après le lemme, par un sommet vert z tel que $\text{pi}(z) = \pi(z)$. Comme les arcs ont des valeurs positives, $\pi(z) = \text{pi}(z) \leq \pi(x)$. Le choix fait par la procédure implique que $\text{pi}(x) \leq \text{pi}(z)$. Par conséquent $\pi(z) \leq \pi(x) \leq \text{pi}(x) \leq \pi(z)$ et $\text{pi}(x) = \pi(x)$. \square

Complexité. Si on s’y prend naïvement, l’algorithme de Dijkstra a une complexité en temps dans le pire des cas en $O(n^2)$. On verra ensuite que, si on s’y prend mieux, on obtient une complexité en $O(m \log n)$.

L'initialisation a une complexité en $O(n)$. La boucle extérieure est exécutée $O(n)$ fois puisque chacun des sommets accessibles à partir de s est colorié une seule fois en rouge. À chaque itération, la recherche du sommet x tel que $\text{pi}(x)$ est minimal coûte $O(n)$ comparaisons si on s'y prend naïvement, ce qui donne $O(n^2)$ comparaisons au total. Supposons le graphe implanté au moyen de listes de successeurs, la boucle intérieure est exécutée au total $O(m)$ fois. On obtient ainsi une complexité en temps en $O(n^2 + m)$ c'est-à-dire en $O(n^2)$ puisque $m \leq n^2$ quel que soit le graphe.

Une réalisation plus sophistiquée

Il est possible de gérer l'ensemble des sommets verts au moyen d'une *file avec priorité*, c'est-à-dire une file dans laquelle les sommets x tels que $\text{pi}(x)$ est minimal sortent avant les autres. Une telle file peut se réaliser sous la forme d'un *tas binaire* comportant au maximum n sommets. Les *opérations de tas* sont l'insertion d'un nouveau sommet (sommet bleu colorié en vert), la mise-à-jour de la position d'un sommet déjà présent (sommet vert dont la valeur de pi est diminuée) et l'extraction d'un sommet de valeur de pi minimale. L'algorithme ainsi obtenu est donné en figure 4.11.

procédure Dijkstra (G, s)

début

colorier tous les sommets sauf s en bleu ← total $O(n)$

vider le tas binaire

$\text{pi}(s) := 0$

$\text{pred}(s)$ est indéfini

colorier s en vert et l'insérer dans le tas binaire

tant que le tas binaire n'est pas vide **faire**

extraire un sommet x du tas tel que $\text{pi}(x)$ soit minimal ← unit. $O(\log n)$
total $O(n \log n)$

colorier x en rouge

pour tous les successeurs y de x **faire**

si y est bleu **alors**

$\text{pi}(y) := \text{pi}(x) + v(x, y)$

$\text{pred}(y) := x$

colorier y en vert et l'insérer dans le tas ← unit. $O(\log n)$

sinon si y est vert et $\text{pi}(y) > \text{pi}(x) + v(x, y)$ **alors**

$\text{pi}(y) := \text{pi}(x) + v(x, y)$

$\text{pred}(y) := x$

mettre à jour la position de y dans le tas ← unit. $O(\log n)$

finsi

fait

fait

fin

} total $O(m \log n)$

FIGURE 4.11 – Implantation de l'algorithme de Dijkstra avec un tas binaire

Complexité. Chacune des trois opérations de tas a une complexité en temps dans le pire des cas en $O(\log n)$. Supposons le graphe implanté par des listes de successeurs. Le corps de la boucle intérieure est exécuté $O(m)$ fois. Chaque exécution a une complexité en $O(\log n)$, ce qui donne, au total une complexité en temps dans le pire des cas en $O(m \log n)$ pour la boucle intérieure. Chaque sommet est extrait du tas et colorié en rouge exactement une fois, ce qui donne une complexité en temps dans le pire des cas en $O(n \log n)$ pour toutes les extractions de sommets. Supposons le graphe connexe. On trouve : $O(n \log n) \subset O(m \log n)$. La complexité en temps dans le pire des cas de la boucle extérieure est donc en $O(m \log n)$. En tenant compte des initialisations, on obtient une complexité en temps dans le pire des cas pour tout l'algorithme en $O(m \log n + n)$ c'est-à-dire en $O(m \log n)$ puisque le graphe est connexe.

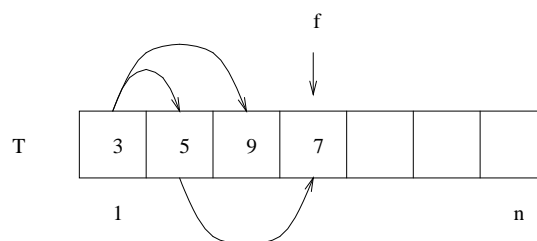
Remarque : dans le cas d'un graphe complet, on a $m = n^2$ et la complexité $O(n^2 \log n)$ de la réalisation plus sophistiquée est moins bonne que celle $O(n^2)$ de la réalisation naïve. La réalisation sophistiquée est surtout intéressante pour des graphes comportant relativement peu d'arcs.

Implantation d'un tas binaire

On peut implanter un *tas binaire* adapté à l'algorithme de Dijkstra au moyen d'un tableau T de n sommets indicés de 1 à n et d'un indice f valant le nombre de sommets présents dans le tas.

On munit le tableau d'une structure d'arbre en posant que les fils gauche et droit de T_i sont T_{2i} et T_{2i+1} . L'idée consiste à maintenir en permanence la propriété suivante : *tout sommet présent dans le tas binaire a une valeur de p_i inférieure ou égale à celle de ses fils gauche et droit.*

La structure de données ainsi obtenue est parfois appelée *minimier* (une variante du *maximier* enseigné dans l'UE API2).



On remarque que le tas est un arbre binaire équilibré. Sa hauteur en nombre d'arcs est $\lfloor \log_2 f \rfloor$ qui appartient à $O(\log n)$.

Pour insérer un élément x dans le tas, il suffit de le placer à l'indice $f + 1$ puis de le permuter avec son père tant qu'il lui est inférieur. L'insertion d'un sommet dans un tas de taille n a une complexité en $O(\log n)$.

L'élément en T_1 est minimal. Une fois extrait, il faut reconstruire le tas. Pour cela, on place le dernier élément T_f en première place puis on le permute avec le plus petit de ses deux fils tant qu'il est supérieur à l'un des deux. L'extraction d'un élément minimal d'un tas de taille n a donc une complexité en $O(\log n)$ aussi.

L'opération de mise-à-jour de la position d'un sommet déjà présent dans le tas dont la valeur de p_i est diminuée est une variante de l'opération d'insertion mais pose une difficulté : pour la réaliser avec une complexité en $O(\log n)$, il faut pouvoir déterminer rapidement la position dans le tas du sommet mis-à-jour. Il suffit pour cela de se donner un deuxième tableau

P de n entiers, indicé par les sommets et de maintenir la propriété suivante : pour tout sommet x présent dans le tas, P_x donne l'indice de x dans T .

4.6.4 Modélisation au moyen de programmes linéaires

Le calcul d'un chemin de valeur minimale entre deux sommets donnés peut être modélisé par un programme linéaire en variables réelles modélisant ce qu'on appelle un « réseau de transport ». Ces programmes linéaires sont des programmes linéaires d'un type particulier : il y a une variable par arc (représentant des quantités en transit ou *flux*), les contraintes sont soit des bornes sur les flots soit des bornes sur la différence entre la somme des flots sortants moins la somme des flots entrants pour chaque sommet. Ces programmes linéaires ont de bonnes propriétés : si les paramètres sont tous des entiers alors le programme admet au moins une solution optimale dont les coordonnées sont entières. De plus, si le solveur utilisé est un solveur qui cherche des solutions « extrêmes » (comme l'algorithme du tableau simplicial qui cherche des solutions sur les sommets du polyèdre des solutions réalisables) alors la solution optimale trouvée aura des coordonnées entières. Il existe des solveurs spécialisés pour les programmes linéaires modélisant des réseaux de transport nettement plus efficaces que les solveurs de programmes linéaires en variables réelles généraux. Le logiciel AMPL offre une syntaxe (que nous ne présentons pas ici) permettant de mettre en évidence ce type de structure.

```
# Fichier chemin_de_valeur_minimale.ampl
set SOMMETS;
# symbolic parce qu'un paramètre est censé être un nombre
param depart symbolic in SOMMETS;
param arrivee symbolic in SOMMETS, <> depart;
# ARCS est un sous-ensemble du produit cartésien SOMMETS x SOMMETS
# Les arcs relient des sommets différents.
set ARCS within { x in SOMMETS, y in SOMMETS : x <> y };
param valeur {ARCS} >= 0;
# check permet de vérifier que les paramètres satisfont certaines propriétés
# On vérifie ici que pour tout arc (x,y) tel que (y,x) existe v(x,y)=v(y,x).
check : forall { (x,y) in ARCS : (y,x) in ARCS }
    valeur [x,y] = valeur [y,x];
# use [x,y] = 1 ssi le chemin de valeur minimale emprunte (x,y)
var use {ARCS} >= 0;
minimize valeur_totale :
    sum { (x,y) in ARCS } use [x,y] * valeur [x,y];
subject to au_depart :
    sum { (depart,y) in ARCS } use [depart,y] = 1;
# Cette loi est valable pour tous les sommets sauf le départ et l'arrivée.
subject to loi_de_conservation
    { c in SOMMETS : c <> depart and c <> arrivee } :
    sum { (x,c) in ARCS } use [x,c] = sum { (c,y) in ARCS } use [c,y];

data;
set SOMMETS := A B C D;
param depart := B;
param arrivee := D;
# Remarquer la déclaration conjointe de l'ensemble ARCS et du paramètre valeur
param : ARCS : valeur :=
    B A      8
    B C      3
```

```

C A    2
A C    2
A D    5;

```

Résolution.

```

ampl: model chemin_de_valeur_minimale.ampl;
ampl: solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 10
ampl: display use;
use :=
A C    0
A D    1
B A    0
B C    1
C A    1
;

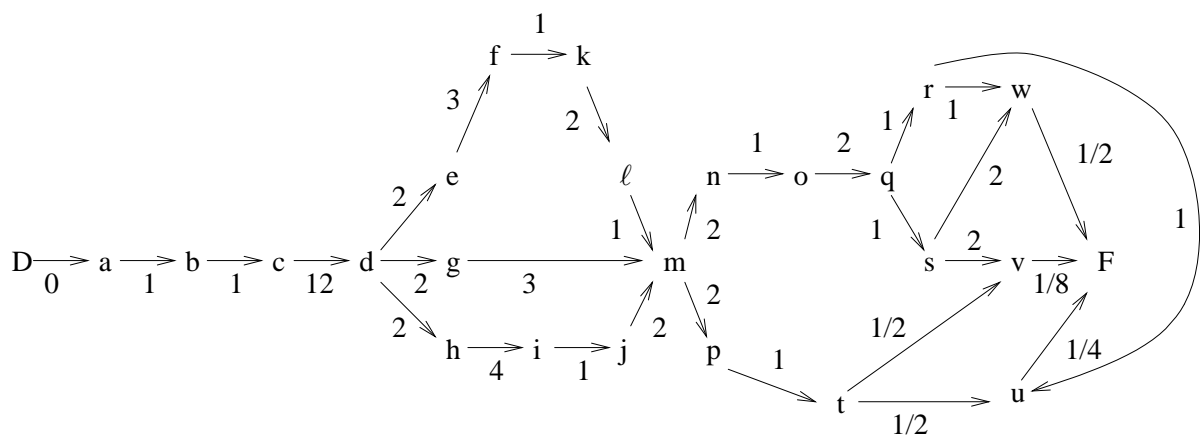
```

4.7 Ordonnancement de tâches : la méthode MPM

Il s'agit d'une application directe des méthodes de recherche des chemins de valeur maximale. On cherche à ordonner un ensemble de tâches en tenant compte de contraintes temporelles : telle tâche doit être terminée avant que telle autre tâche puisse être commencée.

La méthode MPM (ou méthode française des potentiels) permet de déterminer un ordonnancement qui minimise le temps total de réalisation du projet, de déterminer la date de début au plus tôt et la date de début au plus tard de chaque tâche, les tâches « critiques » c'est-à-dire celles dont il est impossible de retarder le début sans retarder l'ensemble du projet ainsi que les marges dont on dispose pour les tâches non critiques. On l'illustre sur l'exemple donné en figure 4.12.

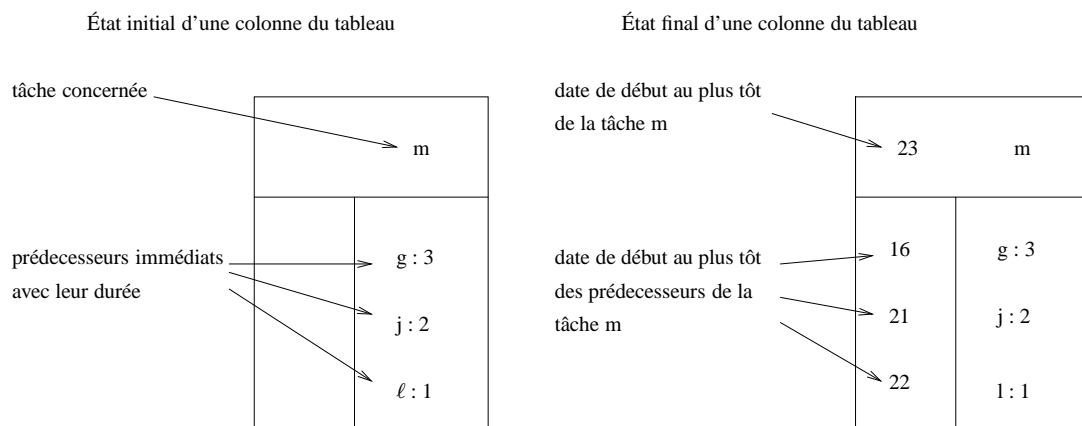
Le problème d'ordonnancement peut se représenter par un graphe orienté et sans circuit. On associe un sommet à chacune des tâches. On rajoute deux tâches artificielles : D (pour *début*) et F (pour *fin*). Un arc d'un sommet s vers un sommet t signifie que la tâche s doit être terminée avant que t puisse commencer. La valeur de l'arc est la durée de la tâche s .



Il est souvent plus pratique de le représenter par un tableau construit comme suit. On attribue une colonne à chaque tâche du tableau.

	Opération	Durée en quinzaines	Tâches antérieures
a	approbation du plan du livre	1	aucune
b	signature du contrat	1	a
c	remise du manuscrit	12	b
d	approbation du comité de lecture	2	c
e	composition du texte	3	d
f	correction par les correcteurs de l'imprimerie	1	e
g	clichage et tirage des hors-texte	3	d
h	exécution des dessins et figures	4	d
i	révision des dessins par l'auteur	1	h
j	correction des dessins : clichage des figures	2	i
k	première correction des épreuves par l'auteur	2	f
ℓ	exécution des premières corrections à l'imprimerie	1	k
m	seconde correction des épreuves par l'auteur	2	g, j, ℓ
n	exécution des secondes corrections à l'imprimerie	1	m
o	tirage du livre	2	n
p	établissement de la prière d'insérer, des listes d'exemplaires presse et d'hommage	1	m
q	pliage	1	o
r	brochage	1	q
s	reliure de certains exemplaires	2	q
t	impression de la prière d'insérer	1/2	p
u	envoi des exemplaires presse	1/4	r, t
v	envoi des hommages	1/8	s, t
w	envoi des contingents aux libraires	1/2	r, s

FIGURE 4.12 – Tâches à effectuer pour éditer un livre (exemple extrait de [4, section 4.3])



L'algorithme est simple (il applique le principe même de l'algorithme de Bellman). Supposons connues les dates de début au plus tôt $\theta_1, \dots, \theta_n$ de toutes les tâches antérieures t_1, \dots, t_n d'une tâche t . Il est alors facile de calculer la date de début au plus tôt θ de t : en notant d_1, \dots, d_n les durées des tâches t_1, \dots, t_n on a

$$\theta = \max_{i=1}^n (\theta_i + d_i).$$

Au fur et à mesure qu'on calcule ces dates de début au plus tôt, on les reporte dans les sous-

colonnes de gauche des colonnes du tableau. La date de début au plus tôt de la tâche F donne la durée totale du projet (sur l'exemple, 31 quinzaines et demie).

On encadre ensuite les tâches critiques dans le tableau. Ce sont celles qu'on ne peut pas retarder sans retarder la durée totale du projet. Elles se calculent en partant de la fin. La tâche F est critique. Si t est critique alors, parmi les tâches prédecesseurs de t , sont critiques les tâches t_i telles que $\theta_i + d_i = \theta$. Un chemin critique est un chemin de D à F uniquement composé de tâches critiques. Il y a deux chemins critiques sur l'exemple.

Voici le tableau après calcul des tâches critiques.

0	a	1	b	2	c	14	d	16	e	19	f
0	$D : 0$	0	$a : 1$	1	$b : 1$	2	$c : 12$	14	$d : 2$	16	$e : 3$

16	g	16	h	20	i	21	j	20	k	22	l
14	$d : 2$	14	$d : 2$	16	$h : 4$	20	$i : 1$	19	$f : 1$	20	$k : 2$

23	m	25	n	26	o	25	p	28	q	29	r
16	$g : 3$	23	$m : 2$	25	$n : 1$	23	$m : 2$	26	$o : 2$	28	$q : 1$
21	$j : 2$										
22	$l : 1$										

29	s	26	t	30	u	31	v	31	w	$31\frac{1}{2}$	F
28	$q : 1$	25	$p : 1$	29	$r : 1$	26	$t : \frac{1}{2}$	29	$r : 1$	30	$u : \frac{1}{4}$
				26	$t : \frac{1}{2}$	29	$s : 2$	29	$s : 2$	31	$v : \frac{1}{8}$
										31	$w : \frac{1}{2}$

On détermine enfin la date de début au plus tard de chaque tâche, c'est-à-dire la date après laquelle on ne peut pas retarder le début de la tâche sans retarder l'ensemble du projet. Pour les tâches critiques la situation est simple : la date de début au plus tôt est égale à la date de début au plus tard. Les autres dates de début au plus tard se déterminent elles-aussi en partant de la fin : supposons déterminées les dates de début au plus tard $\theta_1^*, \dots, \theta_n^*$ de toutes les tâches successeurs d'une tâche t et notons d la durée de t . On a

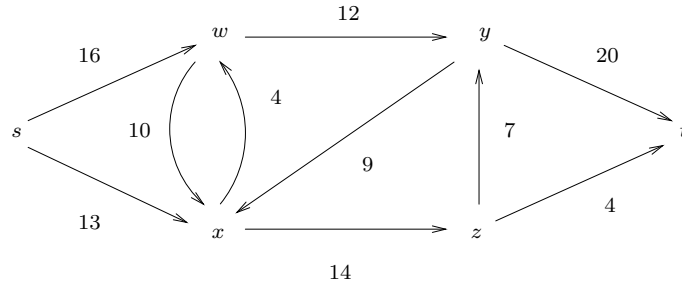
$$\theta^* = \min_{i=1}^n (\theta_i^* - d).$$

Remarque : les problèmes d'ordonnancement s'énoncent assez facilement sous la forme de programmes linéaires en variables réelles : les dates de début au plus tôt et au plus tard des tâches peuvent donc se calculer aussi grâce au simplexe.

4.8 Flots de valeur maximale

On considère un *réseau de transport* c'est-à-dire un triplet (G, s, t) où $G = (S, A)$ est un graphe orienté valué positivement et $s, t \in S$ sont deux sommets distingués appelés *source*

et *destination* du réseau. La restriction à une seule source et une seule destination n'est pas très importante : on peut toujours y ramener un réseau à plusieurs sources ou destinations en ajoutant une source ou une destination artificielles (appelées souvent *supersource* et *superdestination*). On suppose tous les sommets accessibles à partir de s . On suppose t accessible à partir de tout sommet. Les valeurs des arcs de G sont appelées les *capacités* des arcs. On note $c(u, v)$ la capacité de l'arc (u, v) . Si $(u, v) \notin A$ on pose $c(u, v) = 0$. Voici un exemple de réseau de transport.



Le problème à résoudre consiste à déterminer le *flot maximal* pouvant transiter par le réseau. À chaque arc (u, v) du réseau, on associe une variable, appelée le *flux* transitant par cet arc.

Définition 20 (*définition des flux*)

Un flux est une fonction $f : S \times S \rightarrow \mathbb{R}$ satisfaisant les axiomes :

1. $f(u, v) \leq c(u, v)$ pour tous $u, v \in S$ (contrainte de capacité) ;
2. $f(u, v) = -f(v, u)$ pour tous $u, v \in S$;
3. $\sum_{v \in S} f(u, v) = 0$ pour tout sommet u autre que s et t (loi de conservation des flux).

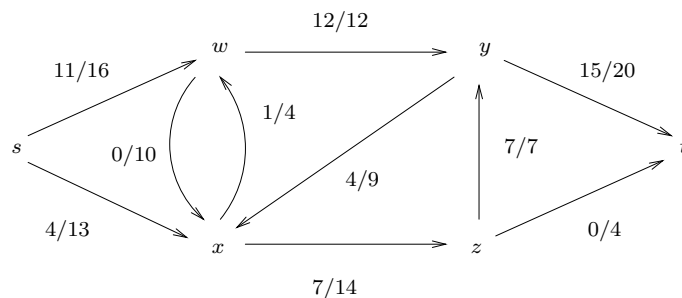
Définition 21 (*définition des flots*)

Un flot est la donnée d'un réseau de transport et d'un flux. La valeur d'un flot est la somme des flux partant de la source.

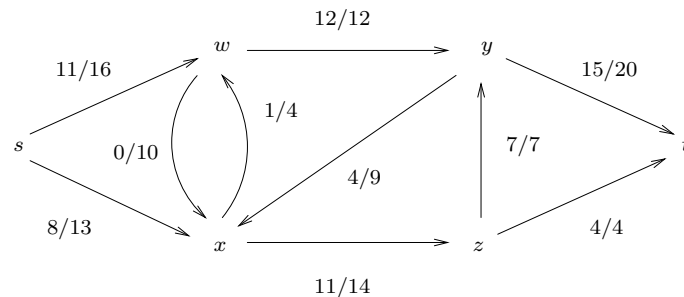
On montre facilement que cette valeur est égale à la somme des flux arrivant sur la destination. Voici un exemple de flot (on note *flux/capacité*) de valeur 15. Vérifions le troisième axiome au niveau du sommet y :

$$\sum_{v \in S} f(y, v) = (-12) + 4 + (-7) + 15 = 0.$$

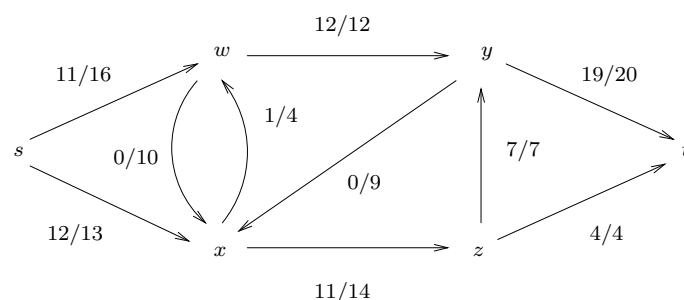
On remarque qu'on n'a considéré dans la somme que les sommets voisins de y : si v n'est pas voisin de y alors $c(y, v) = c(v, y) = 0$ et $f(y, v) = f(v, y) = 0$ d'après les deux premiers axiomes. On remarque aussi que $f(y, w) = -f(w, y) = -12$ d'après le deuxième axiome.



L'algorithme de Ford–Fulkerson est un algorithme incrémental : il part d'un flot de valeur initialement nulle et tente d'augmenter sa valeur petit à petit. La question est : étant donné un flot, comment augmenter sa valeur ? Une façon évidente consiste à chercher un chemin de s vers t formé d'arcs non saturés, c'est-à-dire d'arcs étiquetés f/c tels que $f < c$. Le chemin $(s, x), (x, z), (z, t)$ qu'on peut encore noter $s \rightarrow x \rightarrow z \rightarrow t$ en fournit un exemple. Le long de ce *chemin améliorant*, on peut augmenter le flux (et donc la valeur du flot) du minimum des écarts entre les capacités et les flux c'est-à-dire de $\Delta = \min(13 - 4, 14 - 7, 4 - 0) = 4$. On obtient un nouveau flot de valeur 19.



On peut encore augmenter la valeur du flot mais il n'y a plus de façon évidente de le faire. Un rapide coup d'œil montre que les 4 unités transitant de y vers x vont « dans le mauvais sens » : en effet, pour arriver à la destination, ces unités devraient passer par w ou par z et tous les arcs partant de ces deux sommets sont saturés. Une solution consiste à diminuer de 4 le flux de y vers x et d'augmenter de 4 le flux de y vers t . Bien sûr, pour maintenir la loi de conservation des flux, il faut augmenter de 4 le flux de s vers x . On obtient ainsi un nouveau flot de valeur 23.



La suite d'arcs $(s, x), (y, x), (y, t)$ ne forme plus un chemin mais une *chaîne améliorante*, qu'on peut noter $s \rightarrow x \leftarrow y \rightarrow z$ pour mieux faire ressortir l'arc (y, x) qui est parcouru à l'envers. Plus précisément,

Définition 22 (*définition des chaînes améliorantes*)

Une chaîne améliorante est une chaîne élémentaire entre s et t composée d'arcs (u, v) directs (parcours dans le sens de la flèche) ou indirects (parcours en remontant la flèche) tels que

1. si (u, v) est direct alors $\delta(u, v) \stackrel{\text{def}}{=} c(u, v) - f(u, v) > 0$,
2. si (u, v) est indirect alors $\delta(u, v) \stackrel{\text{def}}{=} f(v, u) > 0$.

Le long d'une chaîne améliorante, il est possible d'augmenter le flux de la quantité Δ , égale au minimum des $\delta(u, v)$ pour tous les arcs (u, v) appartenant à la chaîne. Remarquer qu'il ne

```

procédure Ford_Fulkerson ( $G, s, t$ )
début
  initialiser  $f(u, v)$  à zéro pour tout arc  $(u, v) \in A$ 
  tant que il existe une chaîne améliorante faire
    calculer  $\Delta$ 
    pour tout arc  $(u, v)$  de la chaîne faire
      augmenter  $f(u, v)$  de  $\Delta$  si  $(u, v)$  est direct
      diminuer  $f(v, u)$  de  $\Delta$  si  $(u, v)$  est indirect
    fait
  fait
fin

```

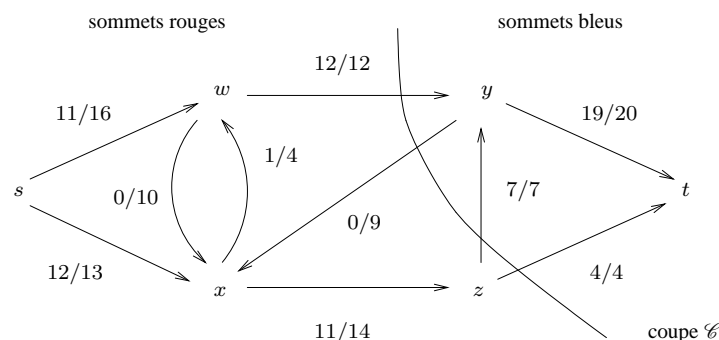
FIGURE 4.13 – Le schéma d’algorithme de Ford–Fulkerson

suffit pas d’énumérer les sommets des chaînes améliorantes : il faut préciser les arcs empruntés. L’algorithme de Ford et Fulkerson est donné en figure 4.13.

Une façon efficace de déterminer s’il existe une chaîne améliorante entre s et t consiste à tenter de la construire à partir de s en adaptant le parcours en largeur d’abord d’un graphe non orienté, plus précisément la fonction *plus_courte_chaîne*. On obtient la fonction donnée en figure 4.14. Cette façon de faire fournit la variante de l’algorithme de Ford–Fulkerson dite d’Edmonds–Karp.

Il y a plusieurs différences vis-à-vis de l’algorithme générique de parcours : on ne suit un arc que s’il est susceptible de faire partie d’une chaîne améliorante (s’il est direct et pas saturé ou indirect avec un flux non nul). On s’arrête dès que la destination est atteinte. On implante enfin une fonction plutôt qu’une procédure. La fonction retourne *vrai* s’il existe une chaîne améliorante et *faux* sinon. Si la chaîne existe, on l’obtient à partir de la destination du réseau en suivant les pointeurs *pred*. En même temps que la chaîne, on calcule pour chaque sommet x un entier $d(x)$ égal à la longueur minimale des chaînes améliorantes entre s et x . Ce compteur ne contribue pas à l’algorithme. Il est utilisé dans le calcul de complexité.

La fonction *chaîne_améliorante_?* appliquée au flot obtenu précédemment retourne *faux*. Voici le coloriage obtenu à la fin de l’exécution.



4.8.1 Correction

À chaque fois que la fonction *chaîne_améliorante_?* retourne *vrai* la valeur du flot est augmentée d’une quantité $\Delta > 0$. Elle n’est donc pas maximale. Il suffit de prouver que le flot est

```

fonction chaîne_améliorante_? ( $G, s, t, f$ )
début
  colorier tous les sommets en bleu sauf  $s$ 
  vider la file
   $d(s) := 0$ 
   $\text{pred}(s)$  est indéfini
  colorier  $s$  en vert et l'enfiler
  tant que la file n'est pas vide et  $t$  n'est pas rouge faire
    défiler  $x$ 
    pour tout voisin bleu  $y$  de  $x$  faire
      si l'arc est direct et pas saturé ou indirect avec un flux non nul ...
      si  $(x, y) \in A$  et  $c(x, y) > f(x, y)$  ou  $(y, x) \in A$  et  $f(y, x) > 0$  alors
        colorier  $y$  en vert et l'enfiler
         $d(y) := d(x) + 1$ 
         $\text{pred}(y) := x$ 
      finsi
    fait
    colorier  $x$  en rouge
  fait
  retourner  $t$  est rouge
fin

```

FIGURE 4.14 – La fonction « chaîne_améliorante_? » recherche la chaîne améliorante la plus courte possible dans un flot. Muni de cette fonction, le schéma d'algorithme de Ford–Fulkerson devient l'algorithme d'Edmonds–Karp.

maximal lorsque la fonction retourne *faux*.

Définition 23 (*définition des coupes*)

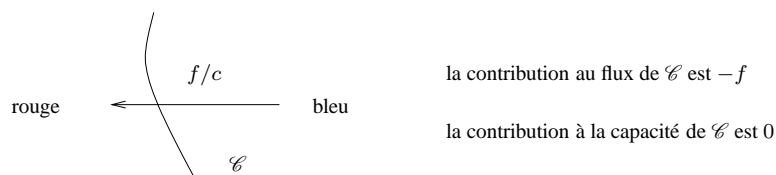
On obtient une coupe d'un réseau de transport en coloriant une partie des sommets (dont la source) en rouge et l'autre partie (dont la destination) en bleu.

Lorsqu'elle retourne *faux*, la fonction *chaîne_améliorante_?* exhibe une coupe (voir le dessin ci-dessus). Ces coupes-là sont particulières puisque les sommets rouges sont nécessairement reliés entre eux. La définition que nous avons adoptée est plus générale et plus simple.

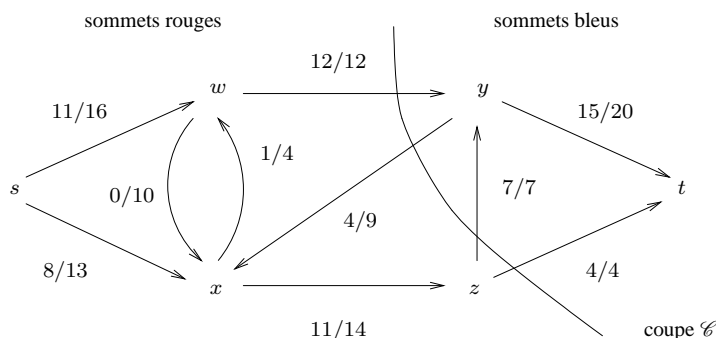
Définition 24 (*flux et capacité d'une coupe*)

Supposons fixée une coupe d'un réseau de transport. Le flux traversant la coupe est la somme des flux $f(u, v)$ tels que u est rouge et v est bleu ; la capacité de la coupe est la somme des capacités $c(u, v)$ telles que u est rouge et v est bleu.

Bien que les définitions soient très semblables, les flux et les capacités des coupes ne se calculent pas de la même façon, en raison des arcs qui vont du bleu vers le rouge :



Considérons la coupe \mathcal{C} obtenue à la fin de l'exemple (voir le dessin à la fin de la section précédente). Sa capacité vaut $12 + 7 + 4 = 23$. Le flux qui la traverse vaut aussi $12 + 7 + 4 = 23$. Considérons maintenant la même coupe mais avec le flot obtenu à l'étape précédente (dessin ci-dessous). Le flux qui la traverse vaut $12 + (-4) + 7 + 4 = 19$. Sa capacité n'a pas changé.



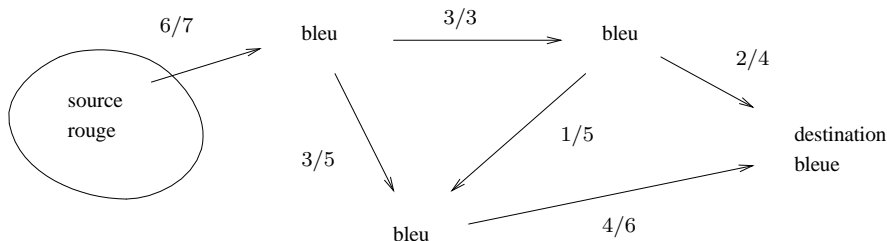
Lemme 2 *Le flux traversant une coupe est inférieur ou égal à sa capacité.*

La proposition suivante est une proposition clef.

Proposition 25 *La valeur du flot est égale au flux traversant n'importe quelle coupe du réseau.*

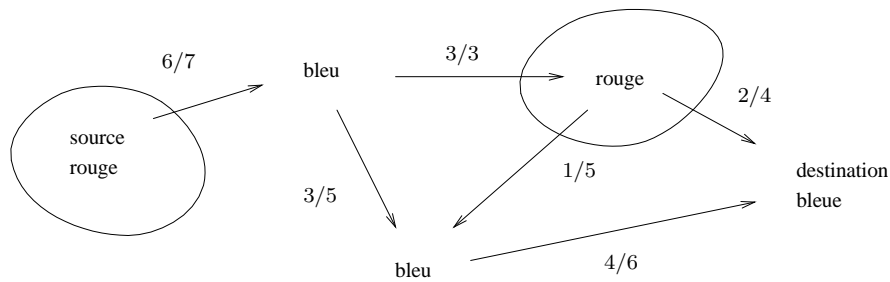
Preuve Par récurrence sur le nombre de sommets rouges.

Base de la récurrence. Il y a toujours au moins un sommet rouge : la source. Si seule la source est rouge alors le flux traversant la coupe est égal à la valeur du flot, par définition du flot.



Quand seule la source est rouge, le flux traversant la coupe est égal à la valeur du flot, par définition

Cas général. On considère une coupe quelconque dont au moins deux sommets sont bleus. On suppose par récurrence que cette coupe vérifie la proposition. On colorie l'un des sommets bleus en rouge (pas la destination, bien sûr). On doit montrer que la nouvelle coupe vérifie encore la proposition. Plutôt que d'écrire rigoureusement la preuve, on illustre l'argument clef sur l'exemple précédent.



Le flux traversant la nouvelle coupe est égal à l'ancien flux + $\underbrace{(-3) + 1 + 2}_{=0}$ (loi de conservation des flux)

□

On peut vérifier la proposition sur les exemples précédent. Elle implique immédiatement que la valeur d'un flot est égale à la somme des flux arrivant sur la destination. Combinée au lemme, elle implique la proposition suivante.

Proposition 26 *La valeur d'un flot est inférieure ou égale à la capacité de toutes les coupes du réseau. En particulier, un flot de valeur égale à la capacité d'une coupe est nécessairement maximal.*

Lorsqu'elle retourne *faux*, la fonction *chaîne_améliorante_?* exhibe une coupe dont le flux est égal à la capacité. Le flot est donc maximal et l'algorithme est correct. On peut montrer plus généralement le théorème suivant.

Théorème 12 (*flot maximal et coupe minimale*)

Le flot maximal pouvant transiter par un réseau de transport est égal à la capacité minimale des coupes du réseau.

On remarque qu'on a affaire à une dualité (cf. l'introduction de la section sur la dualité dans le chapitre consacré au simplexe) : maximiser le flot pouvant transiter par un réseau de transport équivaut à minimiser les capacités des coupes du réseau. Cette dualité fournit un outil permettant de démontrer qu'une solution réalisable (un flux quelconque) est optimal.

4.8.2 Complexité

La variante d'Edmonds–Karp de l'algorithme de Ford–Fulkerson a une complexité en temps dans le pire des cas en $O(nm^2)$ où n désigne le nombre de sommets et m le nombre d'arcs. D'autres implantations de l'algorithme de Ford–Fulkerson ont d'autres complexités. Si on s'y prend mal, l'algorithme peut même ne pas s'arrêter.

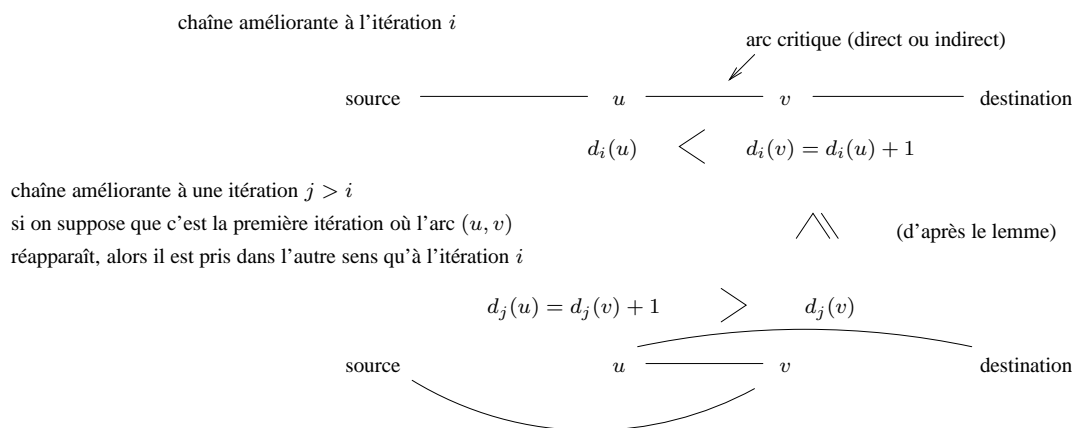
L'entier $d(u)$ calculé par la fonction *chaîne_améliorante_?* fournit la longueur minimale des chaînes améliorantes entre s et u . Le lemme 3 ci-dessous montre que, quel que soit u , cette longueur croît au sens large à chaque itération où elle est calculée. Dans l'analyse qui suit, on note $d_i(u)$ la valeur de $d(u)$ à l'itération i .

Définition 25 (*définition des arcs critiques*)

Un arc appartenant à une chaîne améliorante est dit critique pour cette chaîne s'il est direct et devient saturé ou s'il est indirect et son flux devient nul après augmentation du flux le long de la chaîne.

Si un arc direct (respectivement indirect) est critique à une certaine itération, il peut à nouveau apparaître dans une chaîne améliorante à une autre itération mais il est alors forcément indirect (respectivement direct). Voir le schéma ci-dessous. Ces considérations, combinées au lemme 3, et au fait que l'entier d ne peut pas excéder le nombre n de sommets, montre qu'un arc ne peut être critique que $O(n)$ fois.

Toute chaîne admet au moins un arc critique. Il y a m arcs. La fonction *chaîne_améliorante_?* ne peut donc être appelée que $O(nm)$ fois. Chaque appel à cette fonction a une complexité en temps, dans le pire des cas, en $O(m)$. Par conséquent, l'algorithme d'Edmonds–Karp a une complexité en temps, dans le pire des cas, en $O(nm^2)$.



Lemme 3 *Quel que soit le sommet v , l'entier $d(v)$ augmente au sens large à chaque itération où il est calculé.*

Preuve On raisonne par l'absurde. On suppose qu'il existe un indice i et un sommet v tel que $d_i(v) > d_{i+1}(v)$ et on cherche une contradiction. Si plusieurs sommets v peuvent être choisis, on considère celui pour lequel $d_{i+1}(v)$ est minimal.

Dans le parcours effectué par la fonction *chaîne_améliorante_?* à l'itération $i + 1$, notons u le prédécesseur de v . On a $d_{i+1}(u) = d_{i+1}(v) - 1$. En raison de l'hypothèse de minimalité faite sur v , on a $d_i(u) \leq d_{i+1}(u)$.

À l'itération i , le sommet u est accessible depuis la source par la fonction *chaîne_améliorante_?* sinon u serait inaccessible aussi à l'itération $i + 1$.

Dans le parcours à l'étape i , le sommet u ne peut pas être atteint après v sinon on aurait $d_i(u) > d_i(v) > d_{i+1}(v) > d_{i+1}(u)$, ce qui contredirait le fait que $d_i(u) \leq d_{i+1}(u)$.

Dans le parcours à l'étape i , le sommet u doit donc être atteint avant v . L'arc (u, v) (direct ou indirect), qui est emprunté à l'itération $i + 1$, est empruntable à l'itération i . Comme *chaîne_améliorante_?* réalise un parcours en largeur d'abord, il est donc emprunté à l'itération i et on a $d_i(v) = d_i(u) + 1$. Mais alors $d_i(u) + 1 \leq d_{i+1}(u) + 1 = d_{i+1}(v)$, ce qui contredit l'hypothèse que $d_i(v) > d_{i+1}(v)$.

Cette dernière contradiction conclut la preuve du lemme. \square

4.8.3 Modélisation au moyen de programmes linéaires

Le calcul du flot de valeur maximal pouvant transiter par un réseau de transport se modélise très facilement par un programme linéaire en variables réelles. Voir les remarques faites en

section 4.6.4. La grande puissance d'expression des programmes linéaires permet également de traiter facilement beaucoup de variantes du problème du flot maximal (flot maximal à coût minimal, flot avec perte du flux le long des arcs ...). Voici un programme linéaire modélisant le problème précédent.

```
# Fichier flot_maximal.ampl
set SOMMETS;
param source symbolic in SOMMETS;
param destination symbolic in SOMMETS;
set SOMMETS_INTERNES := SOMMETS diff {source, destination};
set ARCS within {SOMMETS, SOMMETS};
param capacite {ARCS} >= 0;
var flux {(x,y) in ARCS} >= 0, <= capacite [x,y];

maximize valeur_du_flot :
    sum {(source,x) in ARCS} flux [source,x];
subject to loi_de_conservation {x in SOMMETS_INTERNES} :
    sum {(y,x) in ARCS} flux [y,x] = sum {(x,y) in ARCS} flux [x,y];

data;

set SOMMETS := s, w, x, y, z, t;
param source := s;
param destination := t;
param : ARCS : capacite :=
    s w    16
    s x    13
    w y    12
    x w     4
    w x    10
    y x     9
    x z    14
    z y     7
    z t     4
    y t    20;
```

Résolution en AMPL. On trouve un flot maximal un peu différent. La valeur du flot maximal est bien sûr la même.

```
ampl: model flot_maximal.ampl;
ampl: solve;
MINOS 5.5: optimal solution found.
0 iterations, objective 23
ampl: option display_lcol 0;
ampl: display flux;
flux [*,*]
:   t    w    x    y    z    :=
s   .   12   11   .    .
w   .    .    0   12   .
x   .    0    .    .   11
y  19    .    0    .    .
z   4    .    .    7    .
;
```


4.9 Arbres couvrants de valeur minimale

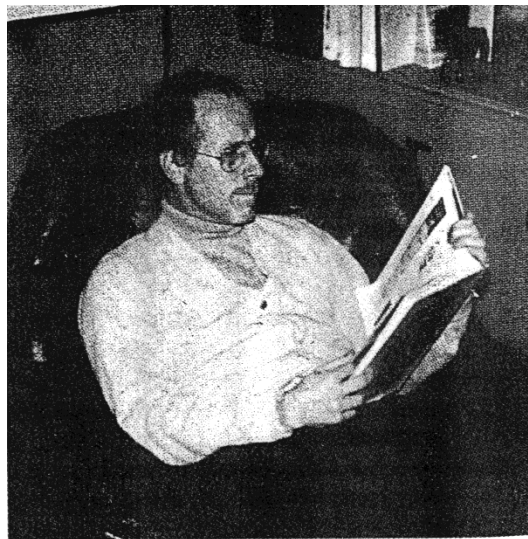
Définition 26 *Un arbre est un graphe non orienté, connexe et sans cycle.*

Les arbres de la théorie des graphes sont différents des arbres qu'on rencontre traditionnellement en programmation. Pour éviter les confusions, on appelle ces derniers des *arborescences* — en tous cas dans cette section. Voici deux différences : les arborescences sont souvent implantées en utilisant des pointeurs qui seraient mieux représentés par des arcs que par des arêtes ; dans une arborescence, on distingue l'un des sommets : la racine.

Un arbre est un graphe planaire admettant une seule face. D'après le théorème d'Euler sur les graphes planaires, un arbre comporte donc $n - 1$ arêtes. Le théorème suivant recense les différentes caractérisations des arbres.

Théorème 13 *Soit $G = (S, A)$ un graphe non orienté ayant n sommets. Les propriétés suivantes sont équivalentes.*

1. G est un arbre,
2. G est connexe et possède $n - 1$ arêtes,
3. G est sans cycle et possède $n - 1$ arêtes,
4. pour tous sommets $x, y \in S$ il existe une unique chaîne élémentaire d'extrémités x et y ,
5. G est connexe mais, quelle que soit l'arête de A , il ne l'est plus si on lui retire cette arête,
6. G est sans cycle mais, quelle que soit l'arête de $S \times S$, il ne l'est plus si on lui ajoute cette arête.



Joseph B. Kruskal

FIGURE 4.15 – Joseph Bernard Kruskal (1928–...)

L'algorithme de Kruskal, inventé en 1956, permet de calculer un arbre couvrant de valeur minimale d'un graphe connexe G . Il applique une stratégie gloutonne. Aucune racine n'est précisée. L'algorithme, donné dans la figure 4.16, maintient l'invariant suivant :

1. T est inclus dans un arbre couvrant de valeur minimale de G .

Aussi bien dans l'algorithme que dans les analyses qui le suivent, T est un ensemble d'arêtes. En toute rigueur, ce n'est donc pas un graphe puisque l'ensemble des sommets n'est pas précisé. Cet abus de langage allège considérablement les notations et ne crée pas de confusion.

fonction Kruskal (G)

Retourne un arbre couvrant de valeur minimale de G

début

$T := \emptyset$

trier les arêtes par valeur croissante

pour chaque arête (x, y) par valeur croissante faire

si $T \cup \{(x, y)\}$ est sans cycle alors

$T := T \cup \{(x, y)\}$

finsi

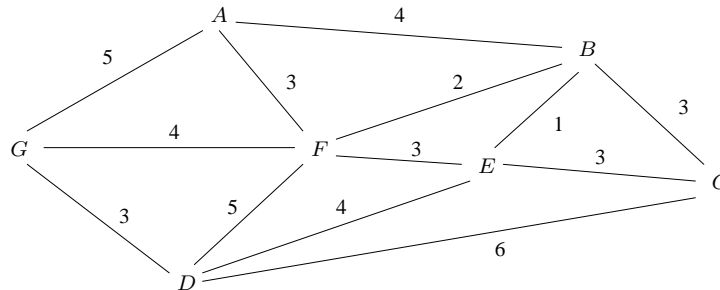
fait

retourner T

fin

FIGURE 4.16 – Première version de l'algorithme de Kruskal

Appliqué au graphe suivant, l'algorithme sélectionne l'arête de valeur 1 puis celle de valeur 2. Pour les arêtes de valeur 3, il doit choisir soit (B, C) soit (E, C) . On voit que le résultat calculé n'est pas défini de façon unique. Il sélectionne aussi les arêtes (A, F) et (G, D) . Pour les arêtes de valeur 4, il doit choisir soit (G, F) soit (D, E) . Aucune autre arête ne peut ensuite être rajoutée. Le nombre d'arêtes est égal au nombre de sommets moins un.



Il est évident que l'algorithme s'arrête.

Considérons le graphe T retourné par l'algorithme.

Quelle que soit l'arête a de $A \setminus T$, le graphe $T \cup \{a\}$ comporte un cycle donc, comme G est connexe, T l'est aussi. Par construction, T est sans cycle. Par conséquent, d'après le théorème (le point 1), T est un arbre. L'argument qui prouve la connexité de T prouve aussi que T couvre G . Le graphe T est donc un arbre couvrant de G .

Deux arbres couvrants de G ont le même nombre d'arêtes d'après le théorème.

Donc, si on suppose l'invariant vérifié, T est égal à un arbre couvrant de valeur minimale de G et l'algorithme est correct.

Il ne reste plus qu'à montrer l'invariant.

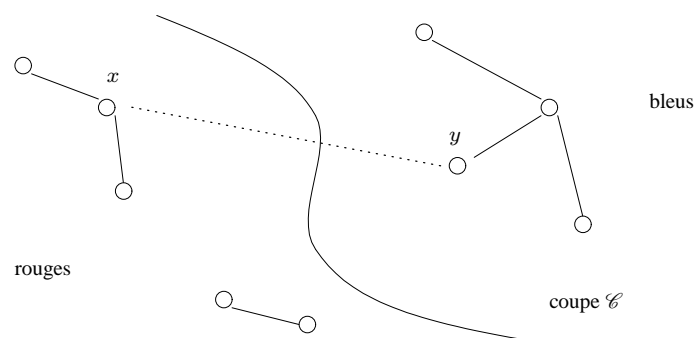
Proposition 27 La propriété « T est inclus dans un arbre couvrant de valeur minimale de G » est un invariant de boucle de l'algorithme de Kruskal.

Preuve Initialement, T est vide et la propriété est vérifiée.

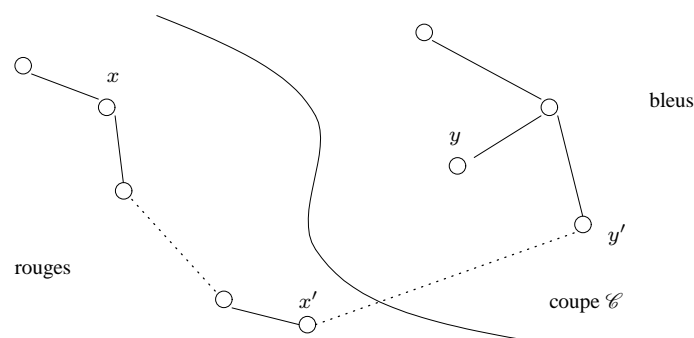
Le cas général. On suppose T inclus dans un arbre couvrant minimal \bar{T} de G . On note (x, y) l'arête choisie par l'algorithme. On doit montrer que $T \cup \{(x, y)\}$ est inclus dans un arbre couvrant minimal \bar{T}' de G (attention : ce n'est pas forcément \bar{T} car le graphe G peut admettre plusieurs arbres couvrants minimaux).

On utilise une technique de preuve fondée sur la notion de *coupe* d'un graphe, proche de la technique employée pour Ford–Fulkerson.

Le graphe T est formé d'une ou plusieurs composantes connexes. On obtient une *coupe* de G en coloriant certains sommets en rouge et les autres en bleu de telle sorte qu'aucune arête de T ne soit bicolore. Parmi toutes les coupes possibles, on en considère une, \mathcal{C} , telle que (x, y) soit bicolore.



L'arbre \bar{T} contient au moins une arête bicolore sinon il ne serait pas connexe. S'il contient (x, y) , la proposition est prouvée. Supposons donc qu'il ne la contienne pas. Il contient une chaîne d'extrémités x et y avec laquelle l'arête (x, y) forme un cycle. Cette chaîne contient au moins une arête bicolore (x', y') puisque x et y sont de couleurs différentes. Cette arête n'appartient pas à T car elle est bicolore.



Notons \bar{T}' le graphe $(\bar{T} \setminus \{(x', y')\}) \cup \{(x, y)\}$ obtenu en substituant (x, y) à (x', y') dans \bar{T} . Le graphe \bar{T}' comporte une chaîne d'extrémités x' et y' passant par (x, y) . Comme \bar{T} est connexe, le graphe \bar{T}' l'est donc aussi. Le graphe \bar{T}' comporte le même nombre d'arêtes que \bar{T} . Par conséquent, d'après le théorème (point 2), \bar{T}' est un arbre. L'argument qui prouve que \bar{T}' est connexe prouve aussi que \bar{T}' couvre G .

L'arête (x, y) , qui est de valeur minimale parmi les arêtes qu'il est possible d'ajouter à T sans former de cycle, est donc minimale parmi les arêtes bicolores et on a $v(x', y') \geq v(x, y)$. Comme $v(\bar{T}') = v(\bar{T}) - v(x', y') + v(x, y)$ on a $v(\bar{T}') \leq v(\bar{T})$. Comme \bar{T} est de valeur

minimale, $v(\bar{T}') \geq v(\bar{T})$ et les deux arbres ont même valeur. Par conséquent, $T \cup \{(x, y)\}$ est inclus dans un arbre, \bar{T}' , de valeur minimale parmi les arbres couvrants de G . \square

4.9.1 Implantation de l'algorithme de Kruskal

Il s'agit de tester efficacement si $T \cup \{a\}$ est sans cycle ou, ce qui revient au même, si les deux extrémités de a appartiennent à la même composante connexe. Les composantes connexes de T sont les classes d'équivalence de la relation : x est équivalent à y si x est accessible à partir de y .

$T \cup \{(x, y)\}$ est sans cycle si et seulement si $x \not\equiv y$.

Dans l'algorithme donné en figure 4.17, on convient de représenter chaque classe par un de ses éléments, appelé *représentant canonique* de la classe. Être capable de calculer, pour chaque classe d'équivalence, un représentant canonique, permet de *transformer les équivalences en égalités*. Ce principe général est le fondement d'un grand nombre d'algorithmes et de théories extrêmement importants⁵.

$x \equiv y$ si et seulement si $\text{rep. canonique}(x) = \text{rep. canonique}(y)$.

À chaque fois qu'une arête (x, y) est ajoutée à T , les sommets x et y doivent être déclarés équivalents. Il faut donc pouvoir fusionner les deux classes en une seule.

fonction Kruskal (G)

Retourne un arbre couvrant de valeur minimale de G

début

$T := \emptyset$

pour chaque sommet x faire

 créer la classe d'équivalence $\{x\}$

fait

trier les arêtes par valeur croissante

pour chaque arête (x, y) par valeur croissante faire

 si x et y ont des représentants canoniques différents alors

 fusionner la classe de x et celle de y en une seule

$T := T \cup \{(x, y)\}$

 fin

fait

retourner T

fin

} total $O(n)$

total $O(m \log m)$

} total $O(m \log n)$

} total $O(m \log m)$

FIGURE 4.17 – Deuxième version de l'algorithme de Kruskal

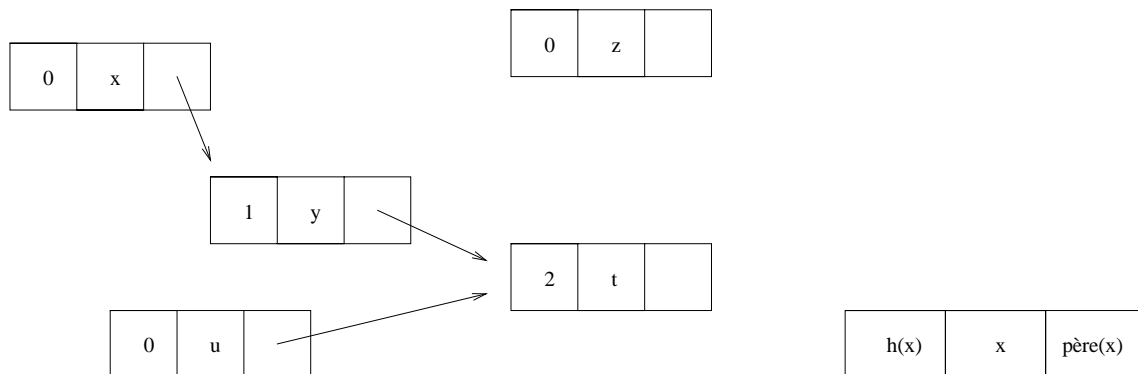
5. Dans le cadre de la théorie des bases de Gröbner par exemple, ce principe permet de décider si deux expressions polynomiales en plusieurs variables sont équivalentes, sachant que les variables sont liées par un ensemble de relations polynomiales.

Complexité. La boucle initiale a une complexité en $O(n)$. Le tri des arêtes a une complexité en $O(m \log m)$. L'algorithme effectue $O(m)$ opérations sur les classes d'équivalence. On montre dans le paragraphe suivant que chaque opération de classe d'équivalence a une complexité en $O(\log n)$. La boucle finale a donc une complexité en temps, dans le pire des cas, en $O(m \log n)$.

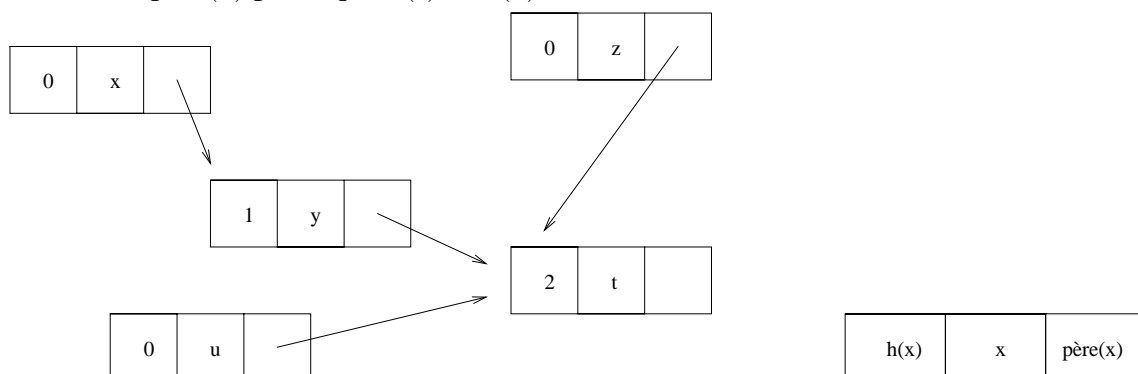
Le graphe est supposé connexe, donc $m \geq n - 1$ et la complexité totale de l'algorithme de Kruskal est dominée par celle du tri des arêtes : $O(n) + O(m \log m) + O(m \log n) = O(m \log m)$.

Implantation des classes d'équivalence

Reste à préciser comment implanter les relations d'équivalence. La structure de données exposée ici est connue sous le nom d'*ensembles disjoints*. À tout sommet x on associe un pointeur père(x) et un compteur $h(x)$. Chaque classe d'équivalence est représentée par une arborescence (voir la définition en début de section). Les arcs sont donnés par les pointeurs père. Le représentant canonique de la classe est la racine de l'arborescence. Le compteur $h(x)$, dont l'utilité est heuristique, vaut la hauteur de l'arborescence des prédécesseurs de x . L'exemple suivant montre une implantation de deux classes d'équivalence $\{x, y, t, u\}$ de représentant canonique t et $\{z\}$ de représentant canonique z .





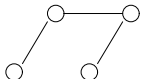
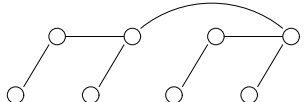
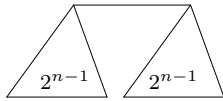
Pour fusionner les classes d'équivalence de deux sommets x et z (on raisonne sur l'exemple), on détermine les représentants canoniques t et z des classes des deux sommets puis, soit on affecte t à père(z) soit on affecte z à père(t). Ce sont les compteurs c qui permettent de choisir : on affecte t à père(z) parce que $h(t) > h(z)$.



Ce choix heuristique ralentit la croissance de la hauteur de l'arborescence et accélère donc les accès aux représentants canoniques. La mise-à-jour des compteurs c est très facile : dans le

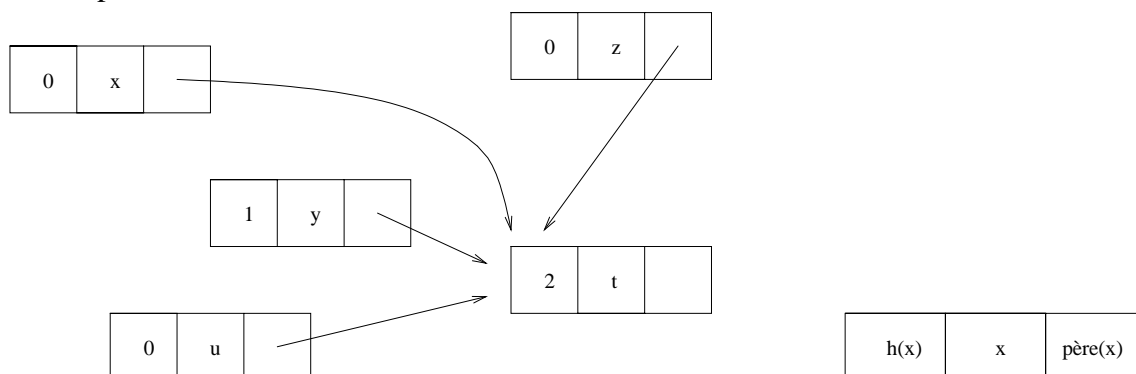
cas où on fusionne deux classes d'équivalence de hauteurs différentes, ils ne sont pas modifiés. Dans le cas où on fusionne deux classes d'équivalence de même hauteur, il suffit d'incrémenter le compteur associé au représentant canonique de la nouvelle classe.

Le raisonnement élémentaire suivant montre qu'il faut au moins 2^k sommets pour produire une classe de hauteur k . Comme une classe comporte au plus n éléments, chaque accès au représentant canonique d'une classe (et donc chaque opération de classe d'équivalence) a une complexité en $O(\log n)$.

hauteur $h(x)$	0	1	2	3	n
configuration minimale					
nombre de sommets	1	2	4	8	2^n

Combien de sommets faut-il au minimum pour obtenir une classe d'une hauteur donnée ?

Maintenant, on peut appliquer une deuxième heuristique naturelle qui n'améliore pas la complexité de l'algorithme de Kruskal mais qui améliore la complexité [3, chapitre 21] des opérations sur les classes d'équivalence. On raisonne toujours sur l'exemple. À chaque fois qu'on détermine le représentant canonique de la classe d'un sommet (mettons) x , on peut après coup affecter à $\text{père}(x)$ l'adresse de ce représentant canonique et ainsi court-circuiter le sommet intermédiaire y . Cette heuristique a un effet secondaire : les compteurs $h(y)$ et $h(t)$ ne donnent plus exactement les hauteurs des arborescences de prédécesseurs des sommets y et t mais des bornes supérieures sur ces hauteurs.



Chapitre 5

Compléments sur le langage AMPL

5.1 Les commentaires

Voici deux façons d’inclure des commentaires dans un modèle AMPL.

```
/*  
  Ceci est un commentaire rédigé sur  
  plusieurs lignes.  
*/  
  
var x >= 0; # en euros/tonne
```

5.2 Les paramètres

5.2.1 Paramètres indicés par un même ensemble

Considérons l’exemple suivant.

```
set PROD;  
param prix_achat {PROD};  
param prix_vente {PROD};  
  
data;  
set PROD := A B F;  
param prix_achat :=  
  A 10  
  B 37  
  F 17;  
param prix_vente :=  
  A 12  
  B 29  
  F 20;
```

Comme *prix_achat* et *prix_vente* sont indicés par un même ensemble, il est possible de déclarer les deux d’un coup.

```
data;  
set PROD := A B F;
```

```

param : prix_achat prix_vente :=
    A      10      12
    B      37      29
    F      17      20;

```

Il est même possible de déclarer en une fois non seulement les deux paramètres mais aussi l'ensemble *PROD*. Il suffit de procéder comme suit :

```

data;
param : PROD : prix_achat prix_vente :=
    A      10      12
    B      37      29
    F      17      20;

```

5.2.2 Paramètres indicés par deux ensembles

Le premier indice est l'indice de ligne. Le second est l'indice de colonne. Si on s'est trompé, on peut insérer « (tr) » entre l'identificateur du paramètre et le « : » dans la partie « data », indiquant ainsi qu'on donne la transposée de la matrice.

```

set PLATS;
set VITAMINES;
param apport {PLATS, VITAMINES};

data;

set PLATS := potjevleesch carbonnade waterzoi;
set VITAMINES := A C D;

param apport :
    A      C      D :=
potjevleesch 18    20    44
carbonnade   5     30    69
waterzoi     40    5     30;

```

5.2.3 Laisser des valeurs indéfinies

Il suffit de mettre un « . » à la place de la valeur indéfinie. En voici une utilisation lors de la définition d'une matrice symétrique.

```

param N integer >= 1;
param matsym {lig in 1 .. N, lig .. N} >= 0;

data;
param N := 3;
param matsym :
    1    2    3 :=
1    4    5    8
2    .    7   11
3    .    .    2;

```


5.2.4 Paramètres indicés par trois ensembles

Voici un exemple de déclaration d'un paramètre *cube* à trois dimensions.

```
# fichier cube
set INDICES := 1 .. 2;
param cube {INDICES, INDICES, INDICES};

data;
param cube :=
    [*, *, 1] : 1    2 :=
        1    3    5
        2    4    6
    [*, *, 2] : 1    2 :=
        1    2   11
        2    0    7;
```

5.3 L'affichage

Le format d'affichage par défaut du paramètre *cube* (le format « liste ») n'est pas très agréable.

```
ampl: model cube;
display cube;
cube :=
1 1 1    3
1 1 2    2
1 2 1    5
1 2 2   11
2 1 1    4
2 1 2    0
2 2 1    6
2 2 2    7
;
```

On peut obtenir un affichage sous forme plus compacte en modifiant la valeur de la variable prédéfinie *display_1col*, qui contient la taille en dessous de laquelle les données sont affichées au format « liste ». Par défaut sa valeur est 20. Pour forcer l'affichage du cube sous forme plus compacte, il suffit donc d'affecter à *display_1col* une valeur inférieure au nombre d'éléments de *cube*. Par exemple zéro.

```
ampl: option display_1col 0;
ampl: display cube;
cube [1,*,*]
:    1    2    :=
1    3    2
2    5   11

    [2,*,*]
:    1    2    :=
1    4    0
2    6    7
;
```

5.4 Les ensembles

5.4.1 Les ensembles non ordonnés

On signale au passage l'existence de la fonction *card*.

```
set PROD;  
param nb_prod := card (PROD);
```

5.4.2 Les intervalles

Les intervalles sont des ensembles de nombres ordonnés (on peut spécifier *by* « *pas* » si on le souhaite).

```
param N >= 1;  
set ANNEES := 1 .. N;
```

5.4.3 Les opérations ensemblistes

Les opérations ensemblistes permettent de définir des ensembles à partir d'ensembles *A* et *B* existants.

A union B,
A inter B,
A diff B dans *A* mais pas dans *B*,
A symdiff B dans *A* ou *B* mais pas les deux.

```
set HUILES_VEGETALES;  
set HUILES_ANIMALES;  
set HUILES := HUILES_VEGETALES union HUILES_ANIMALES;
```

5.4.4 Désigner un élément ou une partie d'un ensemble

On dispose des opérateurs suivants

in pour l'appartenance
within pour l'inclusion.

```
# Fichier exemple  
set NOEUDS;  
param racine in NOEUDS;  
set FEUILLES within NOEUDS := NOEUDS diff { racine };  
  
data;  
set NOEUDS := 3 17 24 45;  
param racine := 17;
```

Exemple d'exécution

```

$ ampl
ampl: model exemple;
ampl: display NOEUDS;
set NOEUDS := 3 17 24 45;
ampl: display racine;
racine = 17
ampl: display FEUILLES;
set FEUILLES := 3 24 45;

```

5.4.5 Désigner un élément d'un ensemble de symboles

Implicitement

On peut vouloir manipuler des noeuds désignés par des symboles plutôt que des nombres. Il faut alors ajouter le qualificatif *symbolic* au paramètre *racine* : les paramètres en effet sont censés désigner des quantités numériques.

```

# Fichier exemple
set NOEUDS;
param racine symbolic in NOEUDS;
set FEUILLES within NOEUDS := NOEUDS diff { racine };

data;
set NOEUDS := A B C D
param racine := B;

```

On continue l'exemple précédent pour montrer un bel exemple de paramètre calculé. On suppose que les noeuds appartiennent à un arbre qu'on décrit grâce la fonction *prédecesseur*. Cette fonction est codée par un paramètre indicé par l'ensemble des noeuds moins la racine. On signale qu'un noeud ne peut pas être son propre prédecesseur. Enfin on affecte au paramètre *prof* la *profondeur* de chaque noeud.

```

param pred {n in NOEUDS diff {racine}} in NOEUDS diff {n};
param prof {n in NOEUDS} :=
    if n = racine then 0 else 1 + prof [pred [n]];

```

Explicitement

Il est parfois utile dans un modèle de manipuler explicitement un élément d'un ensemble. Cette opération est à éviter autant que faire se peut puisque les éléments des ensembles ne sont pas censés être connus dans le modèle. Il vaut mieux utiliser la méthode de la section précédente. Quand c'est nécessaire, il suffit de mettre des guillemets autour du symbole.

```

set PROD;
param prix {PROD} >= 0;
subject to contrainte_speciale :
    prix ["machin"] <= prix ["truc"];

```

Il est parfois utile aussi de donner la valeur d'un ensemble dans le modèle et non dans les données. Cette opération aussi est à éviter autant que faire se peut. Il suffit de mettre des guillemets autour des symboles.

```

set PROD := { "machin", "truc", "chose" };

```

5.4.6 L'opérateur « : »

Il est suivi d'une expression logique. Voici comment sélectionner l'ensemble des éléments positifs d'un ensemble de nombres.

```
set SIGNES;  
set POSITIFS within SIGNES := { i in SIGNES : i >= 0 };
```

Voici un exemple de matrice triangulaire supérieure.

```
param N integer >= 0;  
param matrice {l in 1 .. N, c in 1 .. N : l <= c};
```

Voici deux déclarations possibles de l'ensemble des nombres premiers inférieurs ou égaux à N .

```
param N integer >= 0;  
set premiers1 := { n in 2 .. N : forall {m in 2 .. n-1} n mod m <> 0};  
set premiers2 := { n in 2 .. N : not exists {m in 2 .. n-1} n mod m = 0};
```

5.4.7 Ensembles ordonnés

On a vu les intervalles qui sont des ensembles de nombres ordonnés. On peut manipuler des ensembles ordonnés de symboles également. Voici une version « symbolique » de l'exemple de la section 2.5.2. On considère un ensemble d'employés de différents grades. Les nouveaux employés du grade g sont soit des anciens employés de ce grade soit d'anciens employés promus depuis le grade précédent. Les expressions conditionnelles permettent de gérer les cas particuliers du premier et du dernier grade. Les fonctions *first* et *last* permettent d'obtenir le premier ou le dernier élément d'un ensemble ordonné. Les fonctions *next* et *prev* permettent d'obtenir le successeur ou le prédécesseur d'un élément d'un ensemble ordonné.

```
set GRADES ordered;  
var anciens {GRADES} >= 0;  
var nouveaux {GRADES} >= 0;  
# promus [g] = ceux qui passent de g à g+1  
var promus {g in GRADES : g <> last (GRADES)} >= 0;  
subject to calcule_nouveaux {g in GRADES} :  
    nouveaux [g] = anciens [g] +  
        (if g <> first (GRADES) then promus [prev (g)] else 0) -  
        (if g <> last (GRADES) then promus [g] else 0);
```

Remarque : il est parfois nécessaire de préciser l'ensemble auquel appartient l'élément dont on cherche le successeur ou le prédécesseur. Voici un exemple.

```
set Ens ordered;  
set Fns ordered within Ens;  
  
data;  
set Ens := A B C D E;  
set Fns := B D E;
```

Quel est le prédécesseur de D ? La réponse dépend de l'ensemble considéré :

```
ampl: display prev ("D", Ens);  
prev('D', Ens) = C
```

```
ampl: display prev ("D", Fns);  
prev('D', Fns) = B
```

5.4.8 Ensembles circulaires

Il est également possible de définir des ensembles circulaires (ou cycliques). Il suffit de mettre le qualificatif *circular* au lieu de *ordered*. Le prédécesseur du premier élément est le dernier élément. Le successeur du dernier élément est le premier élément.

5.4.9 Sous-ensembles d'un ensemble ordonné

On peut demander qu'un sous-ensemble d'un ensemble ordonné A soit ordonné suivant le même ordre que A ou suivant l'ordre inverse de celui de A .

```
set A ordered;  
set B within A ordered by A;  
set C within A ordered by reversed A;
```

La même possibilité existe pour les ensembles circulaires. Il suffit d'utiliser les expressions *circular by* et *circular by reversed*.

5.4.10 La fonction ord

Il permet d'obtenir le numéro d'ordre d'un élément dans un ensemble ordonné. L'ensemble des sommets de numéro d'ordre inférieur à la racine.

```
set NOEUDS ordered;  
param racine symbolic in NOEUDS;  
set NOEUDS_INF_RACINE within NOEUDS :=  
  { n in NOEUDS : ord (n) < ord (racine, NOEUDS) };
```

5.4.11 Produits cartésiens

On les a déjà rencontrés dans le cadre de paramètres à deux dimensions. Voici un exemple de modélisation de carte routière. Un couple (x, y) appartient à *ROUTES* s'il existe une route de x vers y . Pour les routes à double sens, il faut stocker à la fois (x, y) et (y, x) .

```
set CARREFOURS;  
set ROUTES within { CARREFOURS, CARREFOURS };  
param longueur {ROUTES} >= 0;  
  
data;  
set CARREFOURS := A B C D;  
set ROUTES := (B,A) (B,C) (A,D) (A,C) (C,A);  
param : longueur :=  
  B A      8  
  B C      3  
  C A      2  
  A C      2  
  A D      5;
```

On améliore la déclaration comme suit. On profite de la possibilité de définir d'un seul coup l'ensemble *ROUTES* et le paramètre *longueur*. On vérifie aussi que la carte est cohérente : d'une part une route relie deux carrefours différents, d'autre part si (x, y) et (y, x) appartiennent à *ROUTES* alors ces deux routes ont même longueur.

```

set CARREFOURS;
set ROUTES within { x in CARREFOURS, y in CARREFOURS : x <> y };
param longueur {ROUTES} >= 0;

check : forall { (x,y) in ROUTES : (y,x) in ROUTES }
    longueur [x,y] = longueur [y,x];

data;
set CARREFOURS := A B C D;
param : ROUTES : longueur :=
    B A      8
    B C      3
    C A      2
    A C      2
    A D      5;

```

5.4.12 Ensembles d'ensembles

Le langage AMPL permet de définir des ensembles indicés par d'autres ensembles. Il y a un ensemble de clients et un ensemble de fournisseurs pour chaque entreprise. À partir de ces ensembles, on crée l'ensemble des clients de toutes les entreprises ainsi que, pour chaque entreprise, l'ensemble de tous les couples (fournisseur, client).

```

set ENTREPRISES;
set CLIENTS {ENTREPRISES};
set FOURNISSEURS {ENTREPRISES};

set TOUS_LES_CLIENTS := union {e in ENTREPRISES} CLIENTS [e];

set COUPLES {e in ENTREPRISES} := {FOURNISSEURS [e], CLIENTS [e]};

data;
set ENTREPRISES := e1 e2;
set CLIENTS [e1] := c1 c2;
set CLIENTS [e2] := c1 c3;
set FOURNISSEURS [e1] := f1 f2 f3;
set FOURNISSEURS [e2] := f4;

```

Voici, pour fixer les idées, ce qu'on obtient à l'exécution.

```

ampl: display CLIENTS;
set CLIENTS[e1] := c1 c2;
set CLIENTS[e2] := c1 c3;

ampl: display TOUS_LES_CLIENTS;
set TOUS_LES_CLIENTS := c1 c2 c3;

ampl: display COUPLES;
set COUPLES[e1] := (f1,c1) (f1,c2) (f2,c1) (f2,c2) (f3,c1) (f3,c2);
set COUPLES[e2] := (f4,c1) (f4,c3);

```

5.5 Les opérateurs arithmétiques et logiques

Le tableau suivant donne les opérateurs par priorité croissante. Il n’y a aucune difficulté à appliquer ces opérateurs sur des paramètres soit dans le corps des contraintes soit pour définir des paramètres calculés. Lorsqu’on les applique à des variables, on court le risque d’obtenir des expressions non linéaires.

L’opérateur `less` retourne la différence de ses deux opérandes si elle est positive et zéro sinon. L’opérateur `div` retourne le quotient de la division euclidienne de ses deux opérandes.

Notation standard	Notation alternative	type des opérandes	type du résultat
<code>if - then - else</code>		logique, arithmétique	arithmétique
<code>or</code>	<code> </code>	logique	logique
<code>exists forall</code>		logique	logique
<code>and</code>	<code>&&</code>	logique	logique
<code>not (unaire)</code>	<code>!</code>	logique	logique
<code>< <= = <> > >=</code>	<code>< <= == != > >=</code>	arithmétique	logique
<code>in not in</code>		objet, ensemble	logique
<code>+ - less</code>		arithmétique	arithmétique
<code>sum prod min max</code>		arithmétique	arithmétique
<code>* / div mod</code>		arithmétique	arithmétique
<code>+ - (unaires)</code>		arithmétique	arithmétique
<code>^</code>	<code>**</code>	arithmétique	arithmétique

Voici un tableau des principales fonctions mathématiques.

<code>abs(x)</code>	valeur absolue
<code>ceil(x)</code>	plus petit entier supérieur ou égal
<code>exp(x)</code>	exponentielle e^x
<code>floor(x)</code>	plus grand entier inférieur ou égal
<code>log(x)</code>	logarithme neperien $\ln x$
<code>log10(x)</code>	logarithme en base 10
<code>sqrt(x)</code>	racine carrée

5.6 Quelques commandes de l’interprète AMPL

La commande « *model nom-de-fichier* » permet de charger un modèle en mémoire. Il est parfois utile d’enregistrer dans des fichiers différents le modèle et ses données (par exemple dans le cas où on souhaite traiter différents jeux de données). Dans ce cas, on charge en mémoire le modèle avec la commande « *model nom-de-fichier* » et les données avec la commande « *data nom-de-fichier* ».

La commande « *reset* » réinitialise la mémoire de l’interprète. Elle est utile en particulier lorsqu’on souhaite utiliser la commande « *model* » plusieurs fois de suite (lors de la mise au point des modèles).

```

ampl: model conges.ampl;
ampl: model conges.ampl;

conges.ampl, line 1 (offset 4):
    PRODUITS is already defined
context:  set >>> PRODUITS; <<<

...

conges.ampl, line 13 (offset 390):
    stock_final_impose is already defined
context:  param >>> stock_final_impose <<< >= 0;

Bailing out after 10 warnings.
ampl: reset;
ampl: model conges.ampl;

```

La commande « *option solver nom-de-solveur* » spécifie le solveur à utiliser.

La commande « *solve* » permet de résoudre un modèle chargé en mémoire.

La commande « *display* » permet d’afficher les valeurs des variables, les valeurs marginales des contraintes etc ... Elle permet d’afficher plusieurs données en même temps. Sur l’exemple, on affiche une variable et ses bornes inférieure et supérieure. Ensuite, on affiche une la valeur du corps d’une contrainte et sa valeur marginale.

```

mpl: model conges.ampl;
ampl: option solver cplex;
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 10231.17857
9 dual simplex iterations (1 in phase I)
ampl: display profit_mensuel.lb, profit_mensuel, profit_mensuel.ub;
:   profit_mensuel.lb profit_mensuel profit_mensuel.ub   :=
Jan      -Infinity      1984      Infinity
Fev      -Infinity      1519.68    Infinity
Mar      -Infinity      1487      Infinity
Avr      -Infinity      800      Infinity
Mai      -Infinity      1855      Infinity
Jun      -Infinity      2585.5     Infinity
;
ampl: display limitation_de_la_vente.body, limitation_de_la_vente;
:   limitation_de_la_vente.body limitation_de_la_vente   :=
Jan P1      55      7
Jan P2      50      1.8
...
Jun P5      95      9.2
Jun P6      55      2.4
Jun P7      34.5     0

```

La contrainte *limitation_de_la_vente* est indiquée par des mois et des produits. Il est possible de n’afficher que les valeurs marginales du mois de mai en utilisant la syntaxe suivante.

```

ampl: display {p in PRODUITS} limitation_de_la_vente ["Mai", p];
limitation_de_la_vente['Mai',p] [*] :=
P1  10
P2  6

```



```
P3    8
P4    4
P5   11
P6    9
P7    3
;
```

La commande *let* permet de modifier la valeur d'un paramètre sous l'interprète. Dans l'exemple suivant, *prix* est une variable, *delta* et *prix_approximatif* sont des paramètres.

```
ampl: model elasticite3.ampl;
ampl: option solver cplex;
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 1992470.886
79 dual simplex iterations (0 in phase I)
ampl: let {p in PRODUITS_LAITIERS} prix_approximatif[p] := prix[p];
ampl: let delta := delta / 2; solve;
CPLEX 8.0.0: optimal solution; objective 1993220.536
37 dual simplex iterations (2 in phase I)
```

Index

- ., 99
- abs, 25
- abus de langage, 89
- accessible, 57
- acyclique, 57
- adjacent, 58
- affichage, 96
- algorithme d'Edmonds–Karp, 82, 85
- algorithme de Bellman, 70, 79
- algorithme de Dijkstra, 71
- algorithme de Ford–Fulkerson, 81
- algorithme de Kruskal, 89
- algorithme glouton, 72, 89
- AMPL, 17
- arborescence, 88
- arbre, 88
- arc, 56
- arc critique, 85
- arc direct, 81
- arc indirect, 81
- arc saturé, 81
- arête, 56
- arrêt du simplexe, 41
- base d'une matrice, 8, 12
- base de Gröbner, 91
- base réalisable, 34
- Bellman, Richard Ernest, 70
- binary, 28
- Blackett, Patrick, 2
- body, 22, 53
- but, 56
- capacité, 80
- card, 97
- chaîne, 57
- changement de solveur, 20, 28
- check, 100
- chemin, 57
- chemin améliorant, 81
- chemin de valeur minimale, 70
- choix de la colonne, 37
- choix de la ligne, 37
- choix local, 72
- circuit, 57
- circuit absorbant, 70
- circuit eulérien, 61
- circular, 100
- classe d'équivalence, 91
- commentaire, 94
- complexité, 59
- complexité du simplexe, 42
- composante connexe, 57, 69
- composante fortement connexe, 57
- connexe, 57
- contrainte, 16
- contrainte inactive, 23, 51
- coupe, 84, 90
- couple, 100
- coût, 31
- coût réduit, 23, 53
- cplex, 17
- critère d'optimalité, 39, 54
- critère de réalisabilité, 44
- cycle, 57
- \mathcal{D} , 31
- déclaration de paramètres, 95
- Dantzig, George, 30
- data, 102
- date de début au plus tôt, 77
- date de début au plus tard, 77
- degré d'un sommet, 61
- destination, 80
- $d_i(u)$, 85
- diff, 97
- Dijkstra, Edsger Wybe, 71
- display, 103

écriture matricielle, 10
 élémentaire, 57
 ensemble, 18, 97
 ensemble circulaire, 100
 ensemble d'ensembles, 101
 ensemble ordonné, 99
 ensembles disjoints, 92
 équations normales, 14
 Euler, Leonhard, 61
 exists, 99
 extrémité, 56

 file, 64
 file avec priorité, 74
 first, 99
 flèche, 56
 flot maximal, 80
 flot maximal par programme linéaire, 87
 flux, 80
 forall, 99, 100
 forme canonique, 33
 forme canonique par rapport à une base, 34
 forme standard, 33
 fortement connexe, 57

 Gauss, Carl Friedrich, 9
 graphe, 56
 graphe acyclique, 68
 graphe non orienté, 56
 graphe orienté, 56
 graphe planaire, 62

 if, 25, 26, 98
 in, 97
 incident, 58
 integer, 20
 inter, 97
 intervalle, 97
 invariant de boucle, 35

 laisser une valeur indéfinie, 95
 last, 99
 lb, 18, 22, 53
 let, 23, 104
 logarithme, 59

 matrice, 7
 matrice inversible, 9

 max, 25
 méthode des deux phases, 43
 méthode MPM, 77
 minos, 17
 model, 102
 modèle AMPL, 18
 moindres carrés, 13

 next, 99

 O, 59
 objectif économique, 16, 31
 objectif artificiel, 43
 opérateurs, 102
 option display_1col, 87, 96
 option solver, 20, 28, 103
 ord, 100
 ordered, 99, 100
 ordered by reversed, 100
 ordonnancement, 77
 ordonnancement par programme linéaire, 79
 origine, 56

 param, 94
 paramètre, 16, 94
 paramètre calculé, 25
 paramètre symbolique, 98
 parcours en largeur, 63
 parcours en profondeur, 66
 pivot de Gauss, 10
 pivot de Gauss–Jordan, 11
 plus court chemin, 82
 plus court chemin par programme linéaire, 76
 plus courte chaîne, 64
 polygone des solutions réalisables, 32
 ponts de Königsberg, 61
pred, 58
 prev, 99
 problème de démarrage, 43
 problème dual, 45, 85
 produit cartésien, 100
 programmation linéaire, 16
 programme artificiel, 43
 programme dual, 46
 programme non borné, 40
 programme primal, 46

 racine, 88

rang d'une matrice, 8, 12
 rc, 23
 réduction de problème, 42
 représentant canonique, 91
 représentation sagittale, 58
 réseau de transport, 80
 reset, 102
 résolution en nombres entiers, 20
 résolution graphique, 17, 31

 sensibilité de l'objectif à l'optimum, 51
 shadow price, 23, 53
 solution de base, 34
 solution optimale, 32
 solution réalisable, 31
 solve, 103
 sommet, 56
 sommet associé à une solution de base, 35
 sommet d'articulation, 67
 sommet multiple, 40
 source, 56, 80
 sous-graphe, 57
 sous-graphe engendré, 57
 stock, 21, 26
succ, 58
 superdestination, 80
 supersource, 80
 symbolic, 98
 symdiff, 97

 tableau simplicial, 35
 tâche critique, 77
 tas binaire, 75
 théorème des écarts complémentaires, 54
 théorème des quatre couleurs, 62
 théorème des relations d'exclusion, 54
 théorème du flot maximal et de la coupe minimale, 85
 transformer les équivalences en égalités, 91
 tri topologique, 68

 ub, 18, 22, 53
 union, 97, 101

 valeur marginale, 23, 51, 53
 variable, 16
 variable artificielle, 43
 variable d'écart, 33
 variable de base, 34
 variable duale, 51
 variable hors base, 34

 within, 97

 z_0 , 35

Bibliographie

- [1] Jacques Baranger. *Introduction à l'analyse numérique*. Hermann, Paris, 1977.
- [2] Jean-Luc Chabert, Évelyne Barbin, Michel Guillemot, Anne Michel-Pajus, Jacques Borowczyk, Ahmed Djebbar, and Jean-Claude Martzloff. *Histoire d'algorithmes. Du caillou à la puce*. Belin, Paris, 1994.
- [3] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique*. Dunod, Paris, 2ème edition, 2002.
- [4] Robert Faure, Bernard Lemaire, and Christophe Picouleau. *Précis de recherche opérationnelle (méthodes et exercices d'application)*. Dunod, Paris, 5ème edition, 2000.
- [5] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL A Modeling Language For Mathematical Programming*. International Thomson Publishing, 1993. <http://www.ampl.com>.
- [6] John J. O'Connor and Edmund F. Robertson. The MacTutor History of Mathematics archive. <http://www-history.mcs.st-andrews.ac.uk/history>, 2006.
- [7] Hilary Paul Williams. *Model Building in Mathematical Programming*. John Wiley and Sons, New York, 3rd edition, 1990.