MASTER EN SCIENCES INFORMATIQUES

Une implémentation de l'algorithme de Karmarkar

Van Kerm, Alain

Award date:
1989

Awarding institution:
Universite de Namur

Facultés Universitaires Notre Dame De la Paix
Institut d'Informatique
B-5000 NAMUR

UNE IMPLEMENTATION DE L'ALGORITHME DE KARMARKAR

Mémoire présenté pour l'obtention du grade de Licencié et Maître en informatique par

Alain VAN KERM

Année académique 1988-1989

ABSTRACT

L'algorithme de Karmarkar est un algorithme de résolution de problèmes de programmation linéaire simple. Son principal attrait provient de ses performances théoriques meilleures que celles du célèbre algorithme du simplexe mis au point par Dantzig. Nous avons implémenté l'algorithme de Karmarkar en lui apportant quelques modifications permettant une réduction du nombre d'itérations et l'obtention de la solution duale. Nous nous sommes soucié des performances de place pour le stockage des données et de temps d'exécution. Cela nous a amené à utiliser des matrices creuses. Nous avons également réalisé un outil permettant un encodage simple et clair pour les problèmes de "faible" taille de manière à offrir un cadre de travail relativement agréable.

The Karmarkar algorithm is a linear programming algorithm. Its main interest comes from its theoretical performances that are better than those of the well-known simplex algorithm. We have implemented this algorithm making some modifications so that the number of iterations can be reduced and that the dual solution can be obtained. We have been concerned with the memory space necessary for the data storage and with the execution time. This has led us to the use of sparse matrices. We also have achieved a tool allowing simple and clear encoding of "small" linear programming problem. So a relatively attractive working environment is offered to the user.

Mes remerciements s'adressent à M. J. Fichefet, promoteur de ce mémoire, pour la confiance qu'il m'a accordée. Ils vont également à tous ceux qui d'une manière ou d'une autre m'ont assisté durant le *"douloureux accouchement"*, amis, parents et autres personnes.

I Table des matières

II. Prérequis et notations utilisées

prérequis

Un minimum de connaissances en algèbre linéaire, en analyse mathématique et en programmation linéaire est nécessaire à la bonne compréhension de ce travail.

Notations et terminologie employées

- un vecteur est représenté par une lettre grasse minuscule.
  exemple : $\mathbf{x} = (x_1, x_2, \ldots, x_{n-1}, x_n)^t$

- Une matrice est représentée par une lettre grasse majuscule. Les coefficients de la matrice sont representés par la lettre non grasse minuscule correspondante, indicée du numéro de ligne et du numéro de colonne
exemple :

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots. & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m-1,1} & a_{m-1,2} & \ldots & a_{m-1,n} \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix}$$

- Tout nombre réel ou entier est représenté en caractère non gras.
  exemple : $y = 9, \quad \alpha = 0.25$

- Le vecteur **0** est le vecteur $\mathbf{0} = (0, 0, \ldots, 0)^t$.

- Le vecteur $\mathbf{e_i}$ est le vecteur dont toutes les composantes sont nulles sauf la $i^{ème}$ qui est égale à 1: $\mathbf{e_i} = (0, \ldots, 0, 1, 0, \ldots, 0)^t$.
  ième composante

- Le vecteur **e** est le vecteur dont toutes les composantes sont 1: $\mathbf{e} = (1, \ldots, 1)^t$.

- La matrice **O** est la matrice carrée dont tous les coéfficients sont nuls.

- La matrice **I** est la matrice identité.

• Les dimensions des matrices carrées **I** et **O** ainsi que des vecteurs sont implicites au contexte dans lequel ces matrices et vecteurs sont utilisés. On les spécifiera parfois explicitement entre parenthèses de la manière suivante : $\mathbf{I}^{(n)}$, $\mathbf{O}^{(n+1)}$, $\mathbf{e}^{(m)}$, etc...

• La plupart du temps, nous utiliserons la notation matricielle suivante pour décrire les problèmes de programmation linéaire simple (ces derniers seront également noté PLS en abrégé).

$$\begin{cases} \text{Minimiser (Maximiser)} & \mathbf{c}^t.\mathbf{x} \\ \text{Avec} & \mathbf{A}.\mathbf{x} \geq (\leq, =)\ \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}\ (\leq \mathbf{0},\ \text{quelconque}) \end{cases}$$

• La fonction $\mathbf{c}^t.\mathbf{x}$ à optimiser, est appelée *fonction-objectif.*

• Le vecteur **c** est appelé *vecteur des contributions à la fonction-objectif.* Une quantité c_j est appelée *contribution de la variable* x_j *à la fonction-objectif.*

• La matrice **A** est appelée *matrice des contraintes.*

• les x_j sont les *variables* du PLS, le vecteur **x** est appelé*vecteur des variables* du PLS.

• Les contraintes $\mathbf{x} \geq \mathbf{0}$ ($\leq \mathbf{0}$, quelconque) sont appelées *contraintes de signe sur les variables.*

• Le vecteur **b** est appelé *membre de droite du système des contraintes.*

• Nous désignerons par *point réalisable* tout n-uple $(x_1, \ldots, x_n)$ vérifiant toutes les contraintes du PLS. Le terme *programme* en est un synonyme.

• un point réalisable (programme) donnant à la fonction-objectif sa valeur optimale finie est appelé *point réalisable optimal* (*programme optimal*).

• L'ensemble des points réalisables constitue le *domaine réalisable* du PLS.

• *L'intérieur strict du domaine réalisable* est l'ensemble des points réalisables répondant de manière stricte aux contraintes de signe sur les variables.

• Lorsque les contraintes autres que les contraintes de signe sur les variables sont des égalités et qu'elles sont non redondantes (Les lignes de **A** sont linéairement indépendantes), le système composé de ces contraintes est dit *indépendant* (**A** est de *rang plein*).

INTRODUCTION

Depuis 1951, l'algorithme du simplexe s'est révélé le plus performant des algorithmes de résolution de problèmes linéaires en tout genre. Cependant, il demeure une insatisfaction du coté des performances théoriques de celui-ci, bien qu'en pratique les performances soient satisfaisantes en bien des points.

Diverses méthodes ont vu le jour, dans l'espoir de rivaliser avec l'algorithme du simplexe. L' algorithme de Karmarkar que nous allons détailler dans ce travail fait partie de ces méthodes. Nous l'avons implémenté non sans lui avoir apporté quelques modifications qui, nous l'espèrons, devraient en améliorer les performances. Ces performances nous interesserons sur trois points de vue. Le premier concerne les possibilités de résultats. nous nous soucierons particulièrement de la connaissance de la solution duale du problème à résoudre. En effet, la connaissance de cette solution se révèle plus qu'intéressante dans la plupart des problèmes de nature économique. Nous laisserons de coté les autres possibilités de résultats qui pourraient être souhaitées : analyses de sensibilité, de post-optimisation, problèmes de paramètrage et de programmation mixte. En effet, l'état de développement de l'algorithme n'est pas, à notre connaissance, encore assez avancé que pour répondre de manière satisfaisante à ces attentes. Les autres aspects des performances auxquels nous nous attacherons ont trait à l'implémentation de cet algorithme sur ordinateur. Afin de pouvoir envisager de façon raisonnable le traitement de problèmes de moyenne et de grande taille, nous devons veiller à ce que l'algorithme ne demande ni un temps de résolution trop élevé, ni un espace de stockage excessif. Ces deux aspects espace et temps sont, en informatique, bien souvent en conflit l'un avec l'autre. Nous avons tenté de réaliser un compromis satisfaisant grâce à quelques modifications apportées a l'algorithme initial et grâce à un mode de représentation de matrices peu couteux dans le cas de matrices à faible densité, c'est-a-dire dans le cas de matrices possèdant un petit nombre d'éléments non nuls.

Ce travail est structuré comme suit. Le chapitre premier donne les explications nécessaires à une bonne compréhension du fonctionnement de l'algorithme. Le chapitre suivant est consacré aux modifications et améliorations que nous suggérons d'apporter à l'algorithme. Enfin, le dernier chapitre décrit le logiciel que nous avons élaboré. Ce logiciel concerne tant l'algorithme de Karmarkar que son cadre de travail. Nous avons en effet implémenté un outil permettant l'encodage simple et convivial des problèmes à traiter par l'algorithme de Karmarkar. Ce chapitre est cloturé par un petit exemple simple d'utilisation du logiciel. En fin de travail, sera donnée la liste des ouvrages qui ont contribués d'une manière ou d'une autre à l'élaboration de celui-ci.

chapitre 1

L'ALGORITHME DE KARMARKAR

1.1 Introduction

En novembre 1979, le russe L.G. Khatchyan publie un rapport dans lequel est présentée une nouvelle méthode d'optimisation dans le domaine de la programmation linéaire. La méthode, basée sur les principes d'optimisation convexe et d'optimisation sur un ellipsoïde, devait en théorie être plus performante que la méthode du simplexe mise au point par Dantzig vingt-huit ans plus tôt. En effet, le nombre d'itérations necessaire pouvait être borné par une fonction polynômiale en la taille du problème posé, alors que les meilleurs résultats théoriques obtenus pour la méthode du simplexe n'aboutissent qu'à une borne de type exponentiel en la taille du problème sur le nombre maximum d'itérations. D'un point de vue pratique, la méthode de Khatchyan s'est avérée bien inférieure à celle du simplexe tant au point de vue rapidité qu'au point de vue capacité à résoudre des problèmes de grande taille. Elle n'en a, cependant, pas moins conservé son attrait théorique.

En decembre 1984, c'est au tour de N. Karmarkar de publier un rapport dans lequel il décrit un nouvelle méthode de résolution de problèmes en programmation linéaire simple. Cette nouvelle méthode, aux dires de son auteur, serait de 50 à 100 fois plus rapide que la méthode du simplexe et , en tout cas, très efficace pour la résolution de problèmes de grande taille à matrice de contraintes très creuse. Karmarkar parvenait, lui aussi, à trouver une borne de type polynômiale pour sa méthode. Son résultat? Une complexité théorique de l'ordre de $O(n^{3.5}.L^2)$ où n représente le nombre de variables du problème et L le nombre de bits nécessaires au stockage des données. En comparaison, la méthode de Khatchyan avait une complexité théorique de l'ordre de $O(n^6.L^2)$.

On peut émettre quelques doutes quant aux performances annoncées pour l'algorithme de Karmarkar. Les expérimentations réalisées et les modifications apportées par différents auteurs ne

donnent toujours pas de résultats comparables avec ceux de la méthode du simplexe. Cependant, l'optimisme est de rigueur vu la "jeunesse" de l'algorithme de Karmarkar.

Ce premier chapitre est consacré à l'étude de l'algorithme de Karmarkar. Afin de faciliter la compréhension de cet algorithme, nous en donnerons d'abord le principe général (section 1.2). Ensuite les différents aspects de l'algorithme seront examinés plus en détail (section 1.3). Nous discuterons ainsi des hypothèses de travail, des méthodes qu'utilise l'algorithme et des résultats de complexité théorique obtenus. Afin de pouvoir être résolu par l'algorithme de Karmarkar, un problème de programmation linéaire simple doit d'abord être transformé et mis sous une forme utilisable par l'algorithme (section 1.4). Ce problème ne pourra toutefois être résolu que si l'on en connait un point de départ. Une méthode est proposée pour obtenir un tel point (section 1.5).

1.2 Premier aperçu de l'algorithme de Karmarkar

L'algorithme de Karmarkar relève des méthodes itératives travaillant dans l'intérieur strict du domaine réalisable du problème de programmation linéaire à résoudre. Partant d'un point initial donné, l'algorithme permet de construire une suite de points réalisables strictement positifs appartenant à l'intérieur strict du domaine réalisable, en résolvant à chaque itération le sous-problème suivant : Etant donné le point réalisable $\mathbf{x}^k$, trouver un nouveau point réalisable $\mathbf{x}^{k+1}$ du PLS meilleur que $\mathbf{x}^k$ au sens de la fonction-objectif. On peut donc construire la suite $(\mathbf{x}^k)$ qui doit normalement converger vers un programme optimale.

Idée fondamentale de l'algorithme.

L'idée fondamentale de l'algorithme est celle-ci : Plutôt que de résoudre le sous-problème obtenu à chaque itération dans l'espace initial, transformons le en un problème de minimisation d'une fonction linéaire sur une sphère, partant du centre de la sphère.

Lors de chaque itération, le point réalisable courant $\mathbf{x}^k$ est envoyé sur le centre du simplexe unité $S=\{ \mathbf{x} \in R^n \mid \mathbf{x} \geq \mathbf{0}, \mathbf{e}^t.\mathbf{x}=1 \}$ en utilisant une transformation projective T. On résoud ensuite le problème transformé, en en restreignant le domaine à une sphère, en utilisant la méthode du gradient projeté, faisant un pas dans l'espace transformé en partant du centre de la sphère. On obtient ainsi un nouveau point dans l'espace transformé. On applique alors à ce point l'inverse T^{-1} de la transformation projective T, ce qui nous donne un point amélioré $\mathbf{x}^{k+1}$ dans l'espace de départ. On peut ensuite décider de faire une nouvelle itération avec $\mathbf{x}^{k+1}$ comme point courant ou arrêter l'algorithme si l'on estime que le nouveau point trouvé est suffisamment proche de la solution optimale.

1.3 Examen détaillé de l'algorithme de Karmarkar

1.3.1 Introduction

Nous décrirons dans cette section le détail du fonctionnement de l'algorithme de Karmarkar. Le paragraphe 1.3.2 donne les hypothèses de travail, les paragraphes 1.3.3 , 1.3.4 et 1.3.5 décrivent chacun une étape de l'algorithme. Enfin, le paragraphe 1.3.6 donne l'énoncé de l'algorithme et le paragraphe 1.3.7 les résultats de complexité théoriques obtenus.

1.3.2 Hypothèses de travail

L'algorithme de Karmarkar tel que nous le présentons permet de résoudre un programme linéaire mis sous la forme suivante [1] :

$$(1.1)\quad \begin{cases} \text{Minimiser} \quad \mathbf{c}^t.\mathbf{x} & \\ \text{Avec} \quad \mathbf{A}.\mathbf{x} = \mathbf{0} & (1.1a) \\ \mathbf{e}^t.\mathbf{x} = 1 & (1.1b) \\ \mathbf{x} \geq \mathbf{0} & \end{cases}$$

Où $\mathbf{x}$ et $\mathbf{c}$ sont des vecteurs de R^n,
$\mathbf{A}$ est une matrice de dimension $(m \times n)$.
$\mathbf{e}$ est le vecteur de R^n dont toutes les composantes sont égales à 1.

[1] La section 1.4 traite de la classe des problèmes traitables par l'algorithme tandis que la section 2.3 propose une moyen d'étendre cette classe.

et vérifiant les hypothèses suivantes :

(H.1) Le problème posséde une solution optimale finie telle que le minimum de $\mathbf{c}^t.\mathbf{x}$ est zéro.
Ou encore : $\exists\ \mathbf{x}^*$ sol. opt. finie telle que $\mathbf{c}^t.\mathbf{x}^* = 0$.
Remarquons que cette condition implique que $\mathbf{c}^t.\mathbf{x}^* \geq 0$ pour toute solution réalisable $\mathbf{x}$.

(H.2) Le système déterminé par les contraintes (1.1a) et (1.1b) possède une solution dont toutes les composantes sont **strictement** supérieures à zéro.
Ou encore : $\exists\ \mathbf{x}^\circ$ solution de (1.1a) et (1.1b) telle que $x_i^\circ > 0$, i=1,..,n.

(H.3) Le système déterminé par les contraintes (1.1a) et (1.1b) est indépendant , c'est-à-dire qu'il ne contient pas d'équation redondante.

1.3.3 Transformation en un problème de minimisation sur une sphère

Supposons que l'on ait à resoudre un problème répondant à nos hypothèses de travail (cfr. paragraphe 1.3.2), et que l'on possède le point courant $\mathbf{x}^k$, obtenu lors de la $k^{ième}$ itération.

Pour transformer le sous-probléme de la $k+1^{ième}$ itération en un problème de minimisation sur une sphère, N. Karmarkar propose d'utiliser une transformation projective qui envoie le point courant sur le centre du simplexe unité $S=\{\ \mathbf{x} \in R^n \mid \mathbf{x} \geq \mathbf{0},\ \mathbf{e}^t\mathbf{x}=1\ \}$ et de restreindre le domaine du problème transformé à une sphère de même centre que le simplexe S et inscrite dans celui-ci. Cette transformation projective est donnée par :

$$T(\mathbf{x}) = \mathbf{y} = \mathbf{D}^{-1}.\mathbf{x} \ /\ \mathbf{e}^t.\mathbf{D}^{-1}.\mathbf{x} \qquad (1.2)$$

où $\mathbf{e}^t = (1, \ldots, 1) \in R^n$

$\mathbf{D}$ est la matrice diagonale $\begin{bmatrix} x_1^k & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & x_n^k \end{bmatrix}$

Soit $\mathbf{a}^\circ$, le point (1/n, ... , 1/n), centre du simplexe S. On a

$$\mathbf{a}^\circ = T(\mathbf{x}^k) = \mathbf{D}^{-1}.\mathbf{x}^k \;/\; \mathbf{e}^t.\mathbf{D}^{-1}.\mathbf{x}^k = \mathbf{e}/n$$

Réexprimons le sous-problème de la $k+1^{ième}$ itération en fonction des variables transformées. Nous obtenons le sous-problème

$$(1.3)\quad \begin{cases} \text{Minimiser} & \mathbf{c}^t.\mathbf{D}.\mathbf{y} \;/\; \mathbf{e}^t.\mathbf{D}.\mathbf{y} \\ \text{Avec} & \mathbf{A}.\mathbf{D}.\mathbf{y} = \mathbf{0} \\ & \mathbf{e}^t.\mathbf{y} = 1 \\ & \mathbf{y} \geq \mathbf{0} \end{cases}$$

Si nous restreignons le domaine réalisable de ce sous-problème à une sphère de centre $\mathbf{a}^\circ$ inscrite dans le simplexe nous obtenons le sous-problème suivant :

$$(1.4)\quad \begin{cases} \text{Minimiser} & \mathbf{c}^t.\mathbf{D}.\mathbf{y} \;/\; \mathbf{e}^t.\mathbf{D}.\mathbf{y} \\ \text{Avec} & \mathbf{A}.\mathbf{D}.\mathbf{y} = \mathbf{0} \\ & \mathbf{e}^t.\mathbf{y} = 1 \\ & \mathbf{y} \geq \mathbf{0} \\ & \| \mathbf{a}^\circ - \mathbf{y} \| \leq \alpha.r \quad , \alpha \in [0, 1[\end{cases}$$

où $r = 1 / \sqrt{n(n+1)}$ est le rayon de la plus grande sphère qui peut être inscrite dans le simplexe S.

Remarquons que la contrainte $\mathbf{y} \geq \mathbf{0}$ est redondante avec la contrainte $\| \mathbf{a}^\circ - \mathbf{y} \| \leq \alpha.r$, $\alpha \in [0, 1[$ puisque cette dernière force $\mathbf{y}$

à appartenir à S et donc à être positif. Elle peut donc être supprimée du problème (1.4).

Notons également que la fonction à minimiser est non linéaire. Afin de pouvoir utiliser la méthode du gradient projeté, Karmarkar décide d'approximer cette fonction par la fonction linéaire

$$g(\mathbf{y}) = \mathbf{c}^t.\mathbf{D}.\mathbf{y}$$

Cette approximation se justifie par le fait que $g(\mathbf{y})$ est positive en tout point réalisable de (1.4) et par le fait que, si le minimum de la fonction $\mathbf{c}^t.\mathbf{x}$ est zéro, alors le minimum de $g(\mathbf{y})$ l'est aussi.

Finalement, nous obtenons le sous-problème

$$(1.5) \quad \begin{cases} \text{Minimiser} & \mathbf{c}^t.\mathbf{D}.\mathbf{y} \\ \text{Avec} & \mathbf{A}.\mathbf{D}.\mathbf{y} = \mathbf{0} \\ & \mathbf{e}^t.\mathbf{y} = 1 \\ & \| \mathbf{a}^\circ - \mathbf{y} \| \leq \alpha.r \quad , \alpha \in [0, 1[\end{cases}$$

Il s'agit bien d'un problème de minimisation d'une fonction linéaire sur une sphère (de centre $\mathbf{a}^\circ$ et de rayon $\alpha.r$), partant du centre de la sphère, que nous pouvons résoudre par la méthode du gradient projeté.

1.3.4 Application de la méthode du gradient projeté

Le but de la méthode du gradient projeté est de trouver une direction de descente pour la fonction linéaire à minimiser. Cette direction de descente va permettre d'obtenir, au départ d'un point réalisable , un nouveau point réalisable amélioré. (Cette méthode est

la méthode de la plus forte pente couplée avec une projection orthogonale du gradient négatif de la fonction sur l'hyperplan des contraintes).

Posons

$$\mathbf{B} = \begin{bmatrix} \mathbf{A.D} \\ \hline \mathbf{e}^t \end{bmatrix}, \text{ matrice de dimension } (m+1 \times n).$$

Le problème

$$(1.6) \begin{cases} \text{Minimiser} \quad \mathbf{c}^t.\mathbf{D}.\mathbf{y} & (1.6a) \\ \text{Avec} \quad \mathbf{B}.\mathbf{y} = \begin{bmatrix} 0 \\ \vdots \\ 1 \end{bmatrix} & (1.6b) \\ \| \mathbf{a}^\circ - \mathbf{y} \| \leq \alpha.r \quad , \alpha \in [0, 1[& (1.6c) \end{cases}$$

est équivalent au problème (1.5). De ce problème, on connait un point réalisable $\mathbf{y}^k = (1/n, \ldots, 1/n)^t$. Il nous faut chercher un vecteur direction $\mathbf{d}$ vérifiant $\nabla g(\mathbf{y}).\mathbf{d} < 0$ [1] de manière à faire décroître $g(\mathbf{y})$ en $\mathbf{y}^{k+1} = \mathbf{y}^k + \lambda.\mathbf{d}$. De plus, si nous imposons à $\mathbf{d}$ de vérifier $\mathbf{B}.\mathbf{d} = \mathbf{0}$, nous serons assurés que le nouveau point $\mathbf{y}^{k+1}$ vérifie les contraintes (1.6b) de notre problème.

Montrons que si nous choisissons $\mathbf{d}$ égale à la projection du vecteur gradient de $g(\mathbf{y})$ sur le sous-espace nul de $\mathbf{B}$, changée de signe et que si nous imposons à $\mathbf{d}$ de vérifier $\mathbf{B}.\mathbf{d} = \mathbf{0}$, alors la condition $\nabla g(\mathbf{y}).\mathbf{d} < 0$ est vérifiée et $\mathbf{d}$ est une direction de descente pour $g(\mathbf{y})$.

Soit $M = \{ \mathbf{d} \in R^n \mid \mathbf{B}.\mathbf{d} = \mathbf{0} \}$ le sous-espace nul de $\mathbf{B}$.
Soit N le sous-espace orthogonal à M.
Tout vecteur de R^n peut s'exprimer comme la somme d'un vecteur de M et d'un vecteur de N.

---

[1] Nous utiliserons le symbole ∇ pour désigner le gradient d'une fonction. Ainsi $\nabla g(y)$ se lit "gradient de la fonction g(y)"

$\nabla g(\mathbf{y})$ peut donc s'écrire :

$$\nabla g(\mathbf{y}) = \mathbf{d} + \mathbf{B}^t.\beta \qquad \text{où } \mathbf{d} \in M,\ \beta \in R^{m+1} \text{ et } \mathbf{B}^t.\beta \in N.$$

Dans cette expression, isolons $\mathbf{d}$.

$$\mathbf{d} = -\nabla g(\mathbf{y}) - \mathbf{B}^t.\beta \qquad (1.7)$$

Multiplions les deux membres de l'égalité par la matrice $\mathbf{B}$.

$$\mathbf{B}.\mathbf{d} = -\mathbf{B}.\nabla g(\mathbf{y}) - \mathbf{B}.\mathbf{B}^t.\beta$$

Puisque $\mathbf{d}$ appartient à M, on a

$$\mathbf{0} = -\mathbf{B}.-g(\mathbf{y}) - \mathbf{B}.\mathbf{B}^t.\beta$$

Dans cette expression, isolons β.

$$\beta = -(\mathbf{B}.\mathbf{B}^t)^{-1}.\mathbf{B}.\nabla g(\mathbf{y})$$

L'expression (1.7) peut alors se récrire

$$\begin{aligned}\mathbf{d} &= -\nabla g(\mathbf{y}) + \mathbf{B}^t.(\mathbf{B}.\mathbf{B}^t)^{-1}.\mathbf{B}.\nabla g(\mathbf{y}) \\ &= -(\mathbf{I} - \mathbf{B}^t.(\mathbf{B}.\mathbf{B}^t)^{-1}.\mathbf{B}).\nabla g(\mathbf{y}) \\ &= -\mathbf{P}_B.\nabla g(\mathbf{y})\end{aligned}$$

où $\mathbf{P}_B$ est la matrice de projection sur le sous-espace nul de $\mathbf{B}$ et vaut :

$$\mathbf{P}_B = \mathbf{I} - \mathbf{B}^t.(\mathbf{B}.\mathbf{B}^t)^{-1}.\mathbf{B}$$

Vérifions que $\mathbf{d} = -\mathbf{P}_B.\nabla g(\mathbf{y})$ est bien une direction de descente. De (1.7), on peut déduire que

$$\nabla g(\mathbf{y}) + \mathbf{d} = -\mathbf{B}^t.\beta$$

c-à-d que le vecteur $\nabla^t g(\mathbf{y})+\mathbf{d}$ est orthogonal au vecteur $\mathbf{d}$. On peut donc écrire

$$\begin{aligned}\nabla^t g(\mathbf{y}).\mathbf{d} &= ((\nabla^t g(\mathbf{y}) + \mathbf{d}^t) - \mathbf{d}^t).\mathbf{d} \\ &= (\nabla^t g(\mathbf{y}) + \mathbf{d}^t).\mathbf{d} - \mathbf{d}^t.\mathbf{d} \\ &= 0 - \mathbf{d}^t.\mathbf{d} \\ &= -\|\mathbf{d}\|^2\end{aligned}$$

et en conclure que $\mathbf{d} = -\mathbf{P}_B.\nabla g(\mathbf{y})$ est bien une direction de descente pour la fonction linéaire $g(\mathbf{y})$. On peut donc faire un pas dans cette direction, au départ du point $\mathbf{a}^\circ$, afin d'obtenir le point $\mathbf{y}^{k+1} = \mathbf{a}^\circ + \lambda.\mathbf{d}$ tel que $g(\mathbf{y}^{k+1}) < g(\mathbf{a}^\circ)$.

Afin que $\mathbf{y}^{k+1}$ soit un point réalisable de (1.6), il lui faut encore vérifier la contrainte (1.6c) :

$$|| \mathbf{a}^\circ - \mathbf{y}^{k+1} || \leq \alpha.r \quad , \alpha \in [0, 1[$$

c'est-à-dire

$$|| \mathbf{a}^\circ - \mathbf{a}^\circ - \lambda.\mathbf{d} || \leq \alpha.r \quad , \alpha \in [0, 1[$$

ou encore

$$\lambda . || \mathbf{d} || \leq \alpha.r \quad , \alpha \in [0, 1[$$

Ce qui signifie que la valeur de λ pour lequel le pas effectué dans la direction de $\mathbf{d}$ est le plus grand et de longueur $\alpha.r$ est:

$$\lambda = \alpha.r / || \mathbf{d} || \quad , \alpha \in [0, 1[$$

En conclusion, appliquer la méthode du gradient projeté à notre problème (1.6), revient à :

1) calculer $\mathbf{d} = -\mathbf{P}_B .\nabla g(\mathbf{y})$
   c-à-d. $\mathbf{d} = -(\mathbf{I} - \mathbf{B}^t.(\mathbf{B}.\mathbf{B}^t)^{-1}.\mathbf{B}) . \mathbf{D}.\mathbf{c}$
2) calculer $\mathbf{y}^{k+1} = \mathbf{a}^\circ + \lambda.\mathbf{d}$
   c-à-d. $\mathbf{y}^{k+1} = (1/n, \ldots , 1/n)^t + \alpha.r.\mathbf{d} / || \mathbf{d} ||$

1.3.5 Retour à l'espace de départ

Ayant trouvé, dans l'espace transformé le point amélioré $\mathbf{y}^{k+1}$, réalisable pour le problème (1.5), nous pouvons obtenir un nouveau point $\mathbf{x}^{k+1}$, point amélioré réalisable pour le problème de départ (1.1). Pour cela, il suffit d'appliquer au point $\mathbf{y}^{k+1}$ la transformation projective inverse de T donnée par

$$T^{-1}(\mathbf{y}) = \mathbf{D}.\mathbf{y} \;/\; \mathbf{e}^t.\mathbf{D}.\mathbf{y}$$

où **D** est la même matrice que dans l'expression de T (1.2).

On obtient donc $\mathbf{x}^{k+1}$ par:

$$\mathbf{x}^{k+1} = T^{-1}(\mathbf{y}^{k+1})$$

1.3.6 Enoncé de l'algorithme

Les paragraphes précédents ont décrit le travail à faire lors de chaque itération. Il nous reste à donner l'algorithme complet, c'est-à-dire muni de sa boucle principale et de son test d'arrêt.

Algorithme de N. Karmarkar.

Soit un problème de programmation linéaire répondant à nos hypothèses de travail (cfr. paragraphe1.3.2) à résoudre, étant donné le point initial réalisable $\mathbf{x}^\circ > \mathbf{0}$.

$\mathbf{x} := \mathbf{x}^\circ$

faire

Définir

$\mathbf{D} := \text{diag}(x_1, \ldots, x_n)$

$$\mathbf{B} := \begin{bmatrix} \mathbf{A}\,\mathbf{D} \\ \hline \mathbf{e}^t \end{bmatrix}$$

Projeter $\mathbf{D}.\mathbf{c}$ sur le sous-espace nul de $\mathbf{B}$

$\mathbf{d} := -\left(\mathbf{I} - \mathbf{B}^t.(\mathbf{B}.\mathbf{B}^t)^{-1}.\mathbf{B} \right) . \mathbf{D}.\mathbf{c}$

Trouver un point amélioré dans l'espace projeté

$\mathbf{newy} := (1/n, \ldots, 1/n)^t + \alpha.r.\mathbf{d} / \| \mathbf{d} \|$

Renvoyer le point amélioré dans l'espace de départ

$\mathbf{newx} := \mathbf{D}.\mathbf{newy} / \mathbf{e}^t.\mathbf{D}.\mathbf{newy}$

Changer de point courant

$\mathbf{x} := \mathbf{newx}$

Tant que $\mathbf{c}^t.\mathbf{x} > \varepsilon$

Remarques :

1) On peut constater que la projection du point courant sur le point (1/n, ... , 1/n) n'apparaît pas explicitement dans l'énoncé de l'algorithme, cependant elle est implicite dans la définition de **D**.

2) Le test d'arrêt est un **test à seuil**. Nous avons choisi un test à seuil sur la valeur de la fonction-objectif. Mais d'autres tests sont possibles, parmi lesquels :

— Tests à seuil sur la fonction-objectif :
  — $\mathbf{c}^t.\mathbf{x}^{(k+1)} / \mathbf{c}^t.\mathbf{x}^{(k)} < \varepsilon$
  — $\mathbf{c}^t.\mathbf{x}^{(k)} \leq \varepsilon.\mathbf{c}^t.\mathbf{x}^{(0)}$
  — $\mathbf{c}^t.\mathbf{x}^{(k-1)} - \mathbf{c}^t.\mathbf{x}^{(k)} \leq \varepsilon$

— Test à seuil sur la valeur d'une variable dont la valeur à l'optimum est connue.

1.3.7 Convergence de l'algorithme

Afin de prouver la convergence de l'algorithme de Karmarkar en un temps polynomial, une nouvelle fonction (dite "fonction potentielle" et notée f_{pot}) est introduite :

$$f_{pot}(x) = \sum_{i=1}^{n} \ln (\mathbf{c}^t.\mathbf{x} / x_i)$$

Cette fonction a la propriété de conserver sa morphologie dans l'espace transformé, où exprimée en fonction des variables transformées, elle s'écrit

$$f'_{pot}(y) = \sum_{i=1}^{n} \ln (\mathbf{c}^t.\mathbf{D}.\mathbf{y} / y_i) + \text{constante}$$

Remarquons d'une part que, dans un voisinage proche de $\mathbf{a}^\circ$, cette fonction peut être approximée par $g(\mathbf{y}) = \mathbf{c}^t.\mathbf{D}.\mathbf{y}$ qui est la fonction que nous avons choisi de minimiser dans l'espace transformé. D'autre part, toute réduction dans $g(\mathbf{y})$ implique une réduction dans $f'_{pot}(\mathbf{y})$. Fort de ces deux constats, il ne nous reste plus à prouver que, pour autant que la valeur du paramètre α soit bien choisie, chaque itération de l'algorithme provoque une diminution de la fonction potentielle f'_{pot} en $\mathbf{y}^{k+1}$, ce qui implique une diminution equivalente dans f_{pot} en $\mathbf{x}^{k+1}$. Le lecteur intéressé par les théorèmes et les démonstrations qui concourent à cette preuve peuvent consulter [MER](1) . Retenons simplement que l'on aboutit au résultat suivant :

L'algorithme trouve un point admissible $\mathbf{x}$ *tel que*

$\mathbf{c}^t.\mathbf{x}/\mathbf{c}^t.\mathbf{a}^\circ \leq 2^{-q}$ *en* $O(\ n\ (q+\ln\ n)\ )$ *itération(s)*

avec $q = O(L)$,

$L = \log\ (\ 1+Dmax\ ) + \log\ (\ 1+\mu\ )$,

$Dmax = \max\ \{$ dét($\mathbf{X}$) tq. $\mathbf{X}$ est une sous-matrice carrée extraite de $\mathbf{A}$ $\}$

$\mu = \max\ \{\ |ci|,\ |bi|,$ pour $i=1, \ldots, n\ \}$

Ou plus simplement

Si l'on effectue de l'ordre de $O(\ n\ (q+\ln\ n)\ )$ *itération(s) de l'algorithme, la solution trouvée est à* $2^{-O(L)}$ *de l'optimum.*

Ces résultats, comme nous l'avons signalé précedemment, dépendent du choix d'une "bonne" valeur pour le paramètre α. Cette valeur doit être comprise entre 0 et 1, ceci afin que la sphère sur laquelle on minimise soit totalement inscrite dans le simplexe S. Karmarkar considère la valeur 0.25 comme un bon compromis. Cependant, en pratique, il ne semble pas contre-indiqué de donner à α une valeur proche de 1, ou même supérieure à 1. La plupart des auteurs suggèrent de fixer α à une valeur proche de 1 [MER], [TOM], [TUR] et [LIS] ou d'utiliser des techniques permettant de donner à α une valeur appropriée lors de chaque itération [YE], [TOM], [LIS]. Le

(1) Les [ ] renvoient à la bibliographie

gain en sera une diminution sensible du nombre d'itérations nécessaire pour atteindre une précision donnée. La section 2.2 discute plus en détail d'une méthode donnant à α une valeur pouvant être supérieur à un mais néanmoins valable lors de chaque itération.

1.4 Problèmes traitables par l'algorithme

1.4.1 introduction

Nous avons vu, lors de l'énoncé des hypothèses de travail (paragraphe1.3.2), que l'algorithme de Karmarkar traitait des problèmes d' une forme particulière. Le présent paragraphe a pour but de montrer comment élargir la classe des problèmes traitables par l'algorithme.

Nous pouvons dès maintenant insister sur le fait qu'un problème sera traitable par l'algorithme (c'est-à-dire qu'il aboutira a une solution optimale) uniquement s'il répond aux conditions suivantes :

1) Le problème possède un domaine réalisable non vide, ce qui signifie qu'il existe au moins un programme admissible pour ce problème.
2) Le problème possède une solution optimale finie.
3) La valeur optimale de la fonction-objectif est connue.

Nous verrons dans la section 2.3 au chapitre 2 comment la troisième condition pourra être supprimée.

Si nous disposons d'un tel problème, nous pourrons le traiter par Karmarkar moyennant quelques transformations. La première étape consiste à ramener le PLS quelconque de départ à un PLS sous forme standard

$$\begin{cases} \text{Minimiser} & \mathbf{c}^t.\mathbf{x} \\ \text{Avec} & \mathbf{A}.\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{cases}$$

Nous pourrons ensuite transformer ce PLS en un PLS traitable par l'algorithme et donc de forme

$$\begin{cases} \text{Minimiser} & \mathbf{c}^t.\mathbf{x} \\ \text{Avec} & \mathbf{A}.\mathbf{x} = \mathbf{0} \\ & \mathbf{e}^t.\mathbf{x} = 1 \\ & \mathbf{x} \geq \mathbf{0} \end{cases}$$

1.4.2 Mise sous forme standard

On peut toujours transformer un PLS quelconque en un PLS sous forme standard grâce aux règles suivantes.

1) Un problème de maximisation peut se transformer en un problème de minimisation sachant que :

$$\begin{cases} \text{Maximiser} & \mathbf{c}^t.\mathbf{x} \\ & \dots \end{cases} \Leftrightarrow \begin{cases} -\text{ Minimiser} & -\mathbf{c}^t.\mathbf{x} \\ & \dots \end{cases}$$

2) Une contrainte d'inégalité peut être changée en une contrainte d'égalité par l'introduction d'une variable supplémentaire v dite "variable d'écart" et d'une contrainte de non-négativité sur cette variable.

$$\begin{cases} \dots \\ \sum_{j=1}^{n} a_{ij}.x_j \geq b_i \\ \dots \end{cases} \Leftrightarrow \begin{cases} \dots \\ \sum_{j=1}^{n} a_{ij}.x_j - v = b_i \\ \dots \\ v \geq 0 \end{cases}$$

$$\begin{cases} \cdots \\ \sum_{j=1}^{n} a_{ij}.x_j \leq b_i \\ \cdots \end{cases} \quad \Leftrightarrow \quad \begin{cases} \cdots \\ \sum_{j=1}^{n} a_{ij}.x_j + v = b_i \\ \cdots \\ v \geq 0 \end{cases}$$

3) Une variable x_k non contrainte, dite "variable libre", peut être remplacée par deux variables x_k^+ et x_k^- en effectuant le changement de variable $x_k = x_k^+ - x_k^-$ et en ajoutant des contraintes de non-négativité sur les deux nouvelles variables.

$$\begin{cases} \cdots \\ \sum_{\substack{j=1 \\ j \neq k}}^{n} a_{ij}.x_j + a_{ik}.x_k = b_i \\ \cdots \\ x_k \text{ quelconque} \end{cases} \quad \Leftrightarrow \quad \begin{cases} \cdots \\ \sum_{\substack{j=1 \\ j \neq k}}^{n} a_{ij}.x_j + a_{ik}.(x_k^+ - x_k^-) = b_i \\ \cdots \\ x_k^+ \geq 0,\ x_k^- \geq 0 \end{cases}$$

4) Une variable x_k soumise à une contrainte de négativité peut être remplacée par une variable x_k^- soumise à une contrainte de non-négativité.

$$\begin{cases} \cdots \\ \sum_{\substack{j=1 \\ j \neq k}}^{n} a_{ij}.x_j + a_{ik}.x_k = b_i \\ \cdots \\ x_k \leq 0 \end{cases} \quad \Leftrightarrow \quad \begin{cases} \cdots \\ \sum_{\substack{j=1 \\ j \neq k}}^{n} a_{ij}.x_j - a_{ik}.x_k^- = b_i \\ \cdots \\ x_k^- \geq 0 \end{cases}$$

Une fois que l'on connait une solution optimale du problème standardisé, on obtient aisément une solution du problème initial. Il

suffit pour cela d'appliquer les règles 3 et 4 (mais cette fois dans l'autre sens) si l'on a effectué des changements de variable lors de la standardisation. Dans le cas du passage d'une maximisation à une minimisation, le signe de l'optimum peut être retrouvé par la règle 1 (également dans l'autre sens).

1.4.3 mise sous forme traitable par l'algorithme

Supposons que nous disposons du problème standardisé suivant:

$$(1)\quad \begin{cases} \text{Minimiser} & \mathbf{c}^t.\mathbf{x} & \\ \text{Avec} & \mathbf{A}.\mathbf{x} = \mathbf{b} & , \mathbf{A} \text{ de dimension } m \times n \\ & \mathbf{x} \geq \mathbf{0} & \mathbf{c} \text{ et } \mathbf{x} \in R^n \end{cases}$$

et que la valeur optimale de la fonction-objectif, z^*, soit connue. Comme nous supposons que la solution optimale est finie, nous pouvons théoriquement trouver une borne supérieure sur la somme des variables. Soit σ cette borne supérieure. Nous ne changeons pas le problème (1) si nous ajoutons que la somme des variables est inférieure à sa borne supérieure σ, c'est-à-dire si nous ajoutons la contrainte

$$\mathbf{e}^{(n)t}.\mathbf{x} \leq \sigma \qquad (2)$$

Effectuons le changement de variable $\mathbf{x}' = \mathbf{x}/\sigma = (x_1', \ldots, x_n')^t$ et introduisons une variable d'écart v pour la contrainte (2) exprimée en fonction de $\mathbf{x}'$. Nous obtenons le problème

$$\begin{cases} \text{Minimiser} & \mathbf{c}^t.\mathbf{x}' & \\ \text{Avec} & \mathbf{A}.\mathbf{x}' + 0.v = \mathbf{b}/\sigma & \\ & \mathbf{e}^{(n)t}.\mathbf{x}' + v = 1 & \\ & v \geq 0 & \\ & \mathbf{x}' \geq \mathbf{0} & \end{cases}$$

Ce qui donne, en posant $\mathbf{x''} = (x_1', \ldots, x_n', v)^t$ et $\mathbf{c''} = (c_1, \ldots, c_n, 0)^t$

$$\begin{cases} \text{Minimiser} & \mathbf{c''^t}.\mathbf{x''} \\ \text{Avec} & [\mathbf{A} \mid \mathbf{0}].\ \mathbf{x''} = \mathbf{b}/\sigma \quad (3) \\ & \mathbf{e}^{(n+1)t}.\mathbf{x''} = 1 \\ & \mathbf{x''} \geq \mathbf{0} \end{cases}$$

Afin d'introduire **0** dans le membre de droite de (3), multiplions ce dernier par $\mathbf{e}^{(n+1)t}\mathbf{x''}$ (égal à 1), et faisons-le passer dans le membre de gauche. Modifions aussi la fonction-objectif afin que sa valeur à l'optimum soit 0. Nous obtenons

$$\begin{cases} \text{Minimiser} & \mathbf{c''^t}.\mathbf{x''} - z^*.(\mathbf{e}^{(n+1)t}.\mathbf{x''}) \\ \text{Avec} & (\ [\mathbf{A} \mid \mathbf{0}] - \mathbf{b}.\mathbf{e}^{(n+1)t}/\sigma\ )\ \mathbf{x''} = \mathbf{0} \\ & \mathbf{e}^{(n+1)t}.\mathbf{x''} = 1 \\ & \mathbf{x''} \geq \mathbf{0} \end{cases}$$

Ce problème est sous forme traitable par l'algorithme de Karmarkar et, une fois résolu, on peut retrouver la solution du problème donné sous sa forme standard (1) par

$$\mathbf{x} = \sigma.(x_1'', \ldots, x_n'').$$

Cependant, la nouvelle matrice des contraintes

$$A'' = \begin{bmatrix} [\mathbf{A} \mid \mathbf{0}] - \mathbf{b}.\mathbf{e}^{(n+1)t}/\sigma \\ \hline \mathbf{e}^{(n+1)t} \end{bmatrix}$$

ainsi obtenue risque de devenir beaucoup plus dense (c'est-à-dire risque de contenir beaucoup plus d'éléments non nuls) que la matrice **A** puisque, dans la plupart des cas, le vecteur **b** risque d'être assez

dense. Dans le cadre d'une implémentation de l'algorithme tenant compte du caractère creux de la matrice des contraintes, il sera donc préférable de considérer une autre façon d'obtenir un problème traitable par l'algorithme qui permette de conserver le caractère creux de **A**.

méthode conservant le caractère creux

Afin d'introduire **0** dans le membre de droite de (3), ajoutons une nouvelle variable ξ que nous forçons à être égale à 1, multiplions le membre de droite de (3) par ξ et faisons passer ce membre dans le membre de gauche. Modifions également la fonction-objectif de sorte que sa valeur optimale soit 0. Nous obtenons

$$\begin{cases} \text{Minimiser} \quad \mathbf{c''}^t.\mathbf{x''} - z^*.\xi \\ \text{Avec} \quad [\mathbf{A} \mid \mathbf{0}] \,.\, \mathbf{x''} - \mathbf{b}.\xi/\sigma = \mathbf{0} \\ \qquad\qquad [\mathbf{e}^t \mid 1] \,.\, \mathbf{x''} = 1 \qquad (4) \\ \qquad\qquad\qquad\quad \xi = 1 \qquad (5) \\ \qquad\qquad\qquad\quad \mathbf{x''} \geq \mathbf{0} \\ \qquad\qquad\qquad\quad \xi \geq 0 \end{cases}$$

Tenant compte de l'égalité (4), l'égalité (5) peut se récrire

$$[\mathbf{e}^t \mid 1] \,.\, \mathbf{x''} + \xi = 2$$

Inversément, (5) peut se récrire

$$[\mathbf{e}^t \mid 1] \,.\, \mathbf{x''} - \xi = 0$$

Si nous posons $x'''=(x_1''/2, \ldots, x_{n+1}''/2, \xi/2)$ et $c'''=(c_1', \ldots, c_n', 0, -z^*)$, nous obtenons le problème

$$\begin{cases} \text{Minimiser} \quad \mathbf{c'''}^t.\mathbf{x'''} \\ \text{Avec} \quad [\mathbf{A} \mid \mathbf{0} \mid -\mathbf{b}/\sigma\,].\,\mathbf{x'''} = \mathbf{0} \\ \qquad\quad [\mathbf{e}^t \mid 1 \mid -1].\,\mathbf{x'''} = 0 \\ \qquad\quad\;\; [\mathbf{e}^t \mid 1 \mid 1].\,\mathbf{x'''} = 1 \\ \qquad\qquad\qquad\quad \mathbf{x'''} \geq \mathbf{0} \end{cases}$$

traitable par l'algorithme de Karmarkar et dont la matrice des contraintes

$$A''' = \begin{bmatrix} \mathbf{A} & \mathbf{0} & -\mathbf{b}/\sigma \\ \hline \mathbf{e}^t & 1 & -1 \\ \hline \mathbf{e}^t & 1 & 1 \end{bmatrix}$$

hérite du caractère peu dense de **A** lorsque le nombre de lignes et de colonnes de **A** est assez élevé. Notons que le (très faible) prix à payer pour cela est l'ajout d'une variable supplémentaire pour laquelle la colonne correspondante dans la matrice des contraintes est fortement creuse lorsque le nombre de lignes est assez grand.

La solution du problème standard initial (1) s'obtient via

$$\mathbf{x} = 2.\sigma.(x_1'', \ldots, x_n'').$$

1.5 Obtention d'un point de départ

Afin de pouvoir démarrer l'algorithme, nous devons disposer d'un point réalisable $\mathbf{x}^\circ$ à composantes strictement positives (cfr. hypothèses de travail). Etant donné n'importe quel point $\hat{\mathbf{x}}$ de R^n tel que $\hat{\mathbf{x}} > \mathbf{0}$, et à condition que le problème possède un point intérieur réalisable $\mathbf{x}^\circ$, on peut obtenir un point initial réalisable pour ce problème de forme

$$\begin{cases} \text{Minimiser} & \mathbf{c}^t.\mathbf{x} \\ \text{Avec} & \mathbf{A}.\mathbf{x} = \mathbf{0} \\ & \mathbf{e}^t.\mathbf{x} = 1 \\ & \mathbf{x} \geq \mathbf{0} \end{cases}$$

en résolvant le problème suivant :

$$\begin{cases} \text{Minimiser } \lambda \\ \text{Avec } [\,\mathbf{A} \mid -\mathbf{A}.\hat{\mathbf{x}}\,] \cdot \begin{bmatrix} \mathbf{x} \\ \lambda \end{bmatrix} = \mathbf{0} \\ [\mathbf{e}^t \mid 1] \cdot \begin{bmatrix} \mathbf{x} \\ \lambda \end{bmatrix} = 1 \\ \begin{bmatrix} \mathbf{x} \\ \lambda \end{bmatrix} \geq \mathbf{0} \end{cases}$$

Un point réalisable de départ pour ce dernier problème est donné par

$$\begin{bmatrix} \mathbf{x}^\circ \\ \lambda^\circ \end{bmatrix} = \frac{1}{\sum_{i=1}^{n} \hat{x}_i + 1} \begin{bmatrix} \hat{\mathbf{x}} \\ 1 \end{bmatrix}$$

En particulier pour $\hat{\mathbf{x}} = (1, \ldots, 1)$, le point de départ est

$$\mathbf{x}^\circ = \mathbf{e} / (n+1), \qquad \lambda^\circ = 1 / (n+1)$$

Notons que résoudre ce problème équivaut à trouver une solution au système d'inégalités. (Ceci sera utile au paragraphe 2.3.2)

$$\begin{cases} \mathbf{A}.\mathbf{x} = \mathbf{0} \\ \mathbf{e}^t.\mathbf{x} = 1 \\ \mathbf{x} \geq \mathbf{0} \end{cases}$$

chapitre 2

AMÉLIORATIONS DE L'ALGORITHME DE KARMARKAR

2.1 Introduction

Le chapitre premier a mis en lumière le fonctionnement de l'algorithme de Karmarkar. Les résultats de complexité théorique obtenus pour celui-ci se sont révélés fort intéressant. Cependant, en pratique, le coût d'une itération de l'algorithme est tel qu'il rend celui-ci bien moins performant au point de vue temps de résolution que l'algorithme du simplexe. De plus, l'algorithme tel que nous l'avons énoncé ne fournit pas de solution optimale duale et les hypothèses de travail nous empêchent de considérer des PLS pour lesquels nous ne connaitrions pas la valeur optimale de la fonction-objectif.

Nous nous efforcerons dans ce chapitre d'apporter quelques modifications bénéfiques à l'algorithme afin qu'il réponde mieux aux performances qui nous préoccupent. La section 2.2 propose une petite procédure permettant de réduire le nombre d'itérations de l'algorithme tout en conservant le même test d'arrêt. La section 2.3 montre comment, tout en étendant la classe des problèmes traitables aux PLS dont l'optimum est inconnu, nous pouvons obtenir la solution duale du problème considéré. Enfin, la section 2.4 donnera un moyen moins couteux que celui donné dans l'algorithme pour calculer la projection du vecteur $\mathbf{Dc}$ sur l'espace nul de $\mathbf{B}$, réduisant ainsi le travail à effectuer lors de chaque itération.

2.2 Recherche d'un meilleur pas

2.2.1 Introduction

Nous avons déjà signalé que, pour des besoins théoriques, la grandeur du pas effectué lors de la minimisation doit être supérieur à zéro et strictement inférieur à r, rayon de la plus grande sphère inscrite dans le simplexe S. Pour cela, nous avions choisi un multiplicateur α appartenant à l'intervalle [0,1[, et nous faisions un pas de grandeur $\alpha.r$ dans la direction de descente de la fonction à minimiser. Cependant, on peut constater que plus α est choisi grand, plus le nombre total d'itérations de l'algorithme décroît [MER],[TOM],[LIS]. Le multiplicateur α peut même être choisi supérieur à 1 tout en restant convenable pour l'algorithme [TOM], [LIS]. Cette section propose d'examiner un moyen de donner à α, lors de chaque itération, une valeur grande et appropriée. L'approche que nous suivrons est celle de Tomlin [TOM].

2.2.2 Méthode du plus grand pas possible.

La méthode que nous allons décrire permet de donner au multiplicateur α une valeur "appropriée" pour l'itération en cours. Nous entendons par valeur "appropriée" une valeur telle que les méthodes de minimisation mise en oeuvre dans l'algorithme de Karmarkar restent toujours valables. Pour cela il faut que :

1) $\mathbf{y}^{k+1}$, point amélioré dans l'espace transformé, appartienne à l'intérieur strict du simplexe S afin qu'il soit réalisable (et donc que $\mathbf{x}^{k+1}$ le soit aussi)

2) $\mathbf{c}^t.\mathbf{x}^{k+1}$ soit strictement inférieur à $\mathbf{c}^t.\mathbf{x}^k$, ceci à des fins de convergence.

Ces deux conditions nécessitent quelque attention puisqu'elles vont nous permettre de déterminer les conditions à poser sur α. Afin de faciliter l'écriture des expressions, posons $\delta = -r / \|\mathbf{d}\|$. Lors de la $k^{ième}$ itération, le point $\mathbf{y}^{k+1}$ est obtenu par $\mathbf{y}^{k+1} = \mathbf{a}° - \alpha.\delta.\mathbf{d}$. Les vecteurs $\mathbf{a}°$ et $\delta\mathbf{d}$ appartenant au simplexe S, $\mathbf{y}^{k+1}$, somme de deux vecteurs de S, appartient à l'intérieur strict de S pour autant que $\mathbf{y}^{k+1}$ soit supérieur à zéro. C'est-à-dire :

$$\begin{aligned} & \mathbf{a}° - \alpha.\delta.\mathbf{d} > 0 \\ \Leftrightarrow \quad & \alpha.\mathbf{d} > \mathbf{a}°/\ \delta \\ \Leftrightarrow \quad & \begin{cases} \alpha > 1/\ (n.\delta.d_i) & \text{pour } i=1,.....,n \text{ tel que } d_i > 0 \\ \alpha < 1/\ (n.\delta.d_i) & \text{pour } i=1,.....,n \text{ tel que } d_i < 0 \end{cases} \quad (2.1) \end{aligned}$$

Mais, nous désirons aussi que $\mathbf{c}^t.\mathbf{x}^{k+1} < \mathbf{c}^t.\mathbf{x}^k$. Posons

$$\mu = n.\delta \sum_{j=1}^{n} c_j .x^k_j .d_j \quad \text{et} \quad \nu = n.\delta \sum_{j=1}^{n} x^k_j .d_j$$

et voyons comment $\mathbf{c}^t.\mathbf{x}^{k+1}$ peut s'exprimer en fonction de $\mathbf{c}^t.\mathbf{x}^k$.

$$\begin{aligned} \mathbf{c}^t.\mathbf{x}^{k+1} &= \mathbf{c}^t.\ \mathbf{D}.\mathbf{y}^{k+1} / (\mathbf{e}^t.\mathbf{D}.\mathbf{y}^{k+1}) \\ &= \mathbf{c}^t.\ \mathbf{D}.(\mathbf{a}° - \alpha\delta.\mathbf{d}) / (\mathbf{e}^t.\mathbf{D}.(\mathbf{a}° - \alpha\delta.\mathbf{d})) \\ &= \sum_{j=1}^{n} [c_j.x^k_j.(1/n - \alpha\delta.d_j) / (\sum_{j=1}^{n} x_j/n - \alpha\delta \sum_{j=1}^{n} x_j.d_j)] \\ &= \sum_{j=1}^{n} c_j.x^k_j . (1 - n.\alpha.\delta.dj) / (1- \alpha.\nu) \end{aligned}$$

$$= (\mathbf{c}^t.\mathbf{x}^k - \alpha.\mu) / (1- \alpha.v) \qquad (2.2)$$

Dans cette expression, examinons le dénominateur.

$$1 - \alpha.v = 1 - \alpha.n.\delta \sum_{j=1}^{n} (x_j^k d_j)$$

$$\geq 1 - \alpha.n.\delta \sum_{j \mid d_j<0} x_j^k d_j$$

De (2.1), on peut tirer que $\alpha.n.\delta.d_j \geq 1$ pour $d_j<0$. On sait également que

$$\mathbf{e}^t\mathbf{x}^k = \sum_{j=1}^{n} x^k_j = 1 \geq \sum_{j \mid d_j<0}^{n} x^k_j$$

Puisque $\mathbf{x}^k$ est une solution admissible d'un problème traitable par l'algorithme.

On peut donc conclure que

$$1 - \alpha.v \geq 1 - \sum_{j \mid d_j<0}^{n} x_j \geq 0$$

Le dénominateur de (2.2) étant positif, nous aurons une diminution dans la fonction-objectif lorsque $\alpha.\mu$ sera strictement positif, c'est-à-dire lorsque α sera du même signe que μ. Le cas où m est négatif ne devrait arriver que dans de très rare cas, lorsqu'il y a oscillation dans la fonction-objectif. Ceci peut arriver lorsque lors de la mise sous forme acceptable par l'algorithme (cfr. paragraphe 1.4.3), la valeur de la borne supérieure sur la somme des variables est choisie trop petite.

Finalement nous obtenons la procedure suivante qui détermine à chaque itération de l'algorithme de Karmarkar la valeur du multiplicateur α.

Soit $\bar{\alpha}$ un multiplicateur tel que $0 < \bar{\alpha} < 1$

Calculer $\Phi = \min_{d_j<0} (1/ (n.\delta.d_j))$

Calculer $\mu = n.\delta \sum_{j=1}^{n} c_j x_j d_j$

Si $\mu \leq 0$

$\alpha = \bar{\alpha}.\Phi$

Sinon

$\alpha = -\bar{\alpha}$

2.2.3 Evaluation

Cette procédure de recherche de α est fort peu couteuse en temps calcul par rapport au travail effectué lors d'une itération de l'algorithme de Karmarkar. La complexité de la procédure est de l'ordre de O(n) tandis que celle d'une itération de Karmarkar est de l'ordre de $O(m^3)$. De plus le gain que nous pouvons en retirer est plus que considérable puisqu'il permet de diminuer de façon très sensible le nombre d'itérations de l'algorithme Karmarkar.

Le coût en place peut être estimé nul, puisque la procédure est très petite et que la place nécessaire ne varie pas en fonction de la taille du problème à traiter.

2.3 Fonction à valeur optimale inconnue et solution duale

2.3.1 Introduction

Comme nous l'avons souligné, nous désirerions que l'algorithme de Karmarkar puisse s'appliquer à un grand nombre de problème et qu'il nous fournisse la solution duale du problème à résoudre en même temps que sa solution primale. Pour cela, deux optiques sont envisageables. La première consiste à modifier l'algorithme de Karmarkar afin qu'il converge vers la solution duale en même temps que vers la solution primale et qu'il puisse traiter des problèmes pour lesquels la valeur optimale de la fonction-objectif est inconnue à l'avance. De telles modifications sont proposées par Gay et par Ye [GAY], [YE]. La deuxième optique consiste à transformer le problème à résoudre de telle sorte que sa résolution par l'algorithme de Karmarkar nous donne aussi la solution duale et que la valeur optimale de la fonction-objectif de ce problème transformé soit connue [TO1], [TO2]. Dans ce cas, le travail est fait en amont de l'algorithme de Karmarkar et ce dernier ne requiert aucune modification. Nous avons choisi d'opérer de cette manière bien que, nous le constaterons lors de l'évaluation (paragraphe2.3.3) , ce ne soit pas la meilleure solution.

2.3.2 Passage à la forme primale-duale

Dans l'optique d'une résolution d'un PLS par l'algorithme de Karmarkar, la transformation de celui-ci sous sa forme primale-duale offre trois avantages. D'abord, elle permet d'obtenir un programme optimal du problème dual en même temps qu'un programme optimal du problème primal. Ensuite, elle permet de remplacer la fonction-objectif du problème initial par une fonction-

objectif ayant zéro pour optimum. Ceci permet la résolution par Karmarkar de PLS ayant un optimum inconnu à l'avance. Enfin, on pourra disposer d'un point de départ immédiat pour le problème sous forme primale-duale, ce qui évitera une phase supplémentaire pour la résolution du problème.

Nous supposerons ici que le PLS à résoudre est un PLS de maximisation, que les contraintes sont des égalités ou des inégalités et que les variables sont libres ou non-négatives. Tout problème pourra être ramené à cette forme en utilisant les règles 1 et 4 données au paragraphe 1.4.2 et la règle suivante :

Une inégalité peut être transformée en une inégalité de sens contraire en multipliant ses membres de gauche et de droite par -1 .

$$\left(\begin{array}{l} \cdots \\ \sum_{j=1}^{n} a_{ij}.x_j \leq b_i \\ \cdots \end{array} \right) \Leftrightarrow \left(\begin{array}{l} \cdots \\ \sum_{j=1}^{n} - a_{ij}.x_j \geq - b_i \\ \cdots \end{array} \right)$$

Soit donc le PLS suivant[1]

$$(2.3) \quad \left(\begin{array}{ll} \text{Maximiser} & \mathbf{c}^{(p)t}\,\mathbf{x}^{(p)} + \mathbf{c}^{(q)t}\,\mathbf{x}^{(q)} \\ \text{Avec} & \mathbf{A}^{(ip)}\,\mathbf{x}^{(p)} + \mathbf{A}^{(iq)}\,\mathbf{x}^{(q)} \leq \mathbf{b}^{(i)} \\ & \mathbf{A}^{(ep)}\,\mathbf{x}^{(p)} + \mathbf{A}^{(eq)}\,\mathbf{x}^{(q)} = \mathbf{b}^{(e)} \\ & \mathbf{x}^{(p)} \geq \mathbf{0} \\ & \mathbf{x}^{(q)} \text{ quelconque} \end{array} \right.$$

[1] où les lettres i,e,p,q peuvent respectivement être associées aux mots "inégalité", "égalité", "positive" et "quelconque".

où $\mathbf{c}^{(p)}, \mathbf{x}^{(p)} \in R^p$, $\mathbf{c}^{(q)}, \mathbf{x}^{(q)} \in R^q$, $\mathbf{b}^{(i)} \in R^i$, $\mathbf{b}^{(e)} \in R^e$

$\mathbf{A}^{(ip)}, \mathbf{A}^{(iq)}, \mathbf{A}^{(ep)}, \mathbf{A}^{(eq)}$ sont respectivement de dimension $(i \times p)$, $(i \times q)$, $(e \times p)$, $(e \times q)$.

Son dual s'écrit

$$(2.4) \quad \begin{cases} \text{Minimiser} & \mathbf{y}^{(i)t}\,\mathbf{b}^{(i)} + \mathbf{y}^{(e)t}\,\mathbf{b}^{(e)} \\ \text{Avec} & \mathbf{y}^{(i)t}\,\mathbf{A}^{(ip)} + \mathbf{y}^{(e)t}\,\mathbf{A}^{(ep)} \geq \mathbf{c}^{(p)} \\ & \mathbf{y}^{(i)t}\,\mathbf{A}^{(iq)} + \mathbf{y}^{(e)t}\,\mathbf{A}^{(eq)} = \mathbf{c}^{(q)} \\ & \mathbf{y}^{(i)} \geq \mathbf{0} \\ & \mathbf{y}^{(e)} \text{ quelconque} \end{cases}$$

où $\mathbf{c}^{(p)} \in R^p$, $\mathbf{c}^{(q)} \in R^q$, $\mathbf{b}^{(i)}, \mathbf{y}^{(i)} \in R^i$, $\mathbf{b}^{(e)}, \mathbf{y}^{(e)} \in R^e$

$\mathbf{A}^{(ip)}, \mathbf{A}^{(iq)}, \mathbf{A}^{(ep)}, \mathbf{A}^{(eq)}$ sont respectivement de dimension $(i \times p)$, $(i \times q)$, $(e \times p)$, $(e \times q)$.

Nous savons, par la théorie de la dualité, que $\forall\ [\mathbf{x}^{(p)t} \mid \mathbf{x}^{(q)t}]^t$ programme réalisable de (2.3) et $\forall\ [\mathbf{y}^{(i)t} \mid \mathbf{y}^{(e)t}]^t$ programme réalisable de (2.4), on a que

$$[\mathbf{c}^{(p)t} \mid \mathbf{c}^{(q)t}] \cdot \begin{bmatrix} \mathbf{x}^{(p)} \\ \mathbf{x}^{(q)} \end{bmatrix} \geq [\mathbf{y}^{(i)t} \mid \mathbf{y}^{(e)t}] \cdot \begin{bmatrix} \mathbf{b}^{(i)} \\ \mathbf{b}^{(e)} \end{bmatrix}$$

et que $[\mathbf{x}^{(p)*t} \mid \mathbf{x}^{(q)*t}]^t$ et $[\mathbf{y}^{(i)*t} \mid \mathbf{y}^{(e)*t}]^t$ sont des programmes réalisables optimaux respectivement de (2.3) et (2.4) si et seulement si

$$[\mathbf{c}^{(p)t} \mid \mathbf{c}^{(q)t}] \cdot \begin{bmatrix} \mathbf{x}^{(p)*} \\ \mathbf{x}^{(q)*} \end{bmatrix} = [\mathbf{y}^{(i)*t} \mid \mathbf{y}^{(e)*t}] \cdot \begin{bmatrix} \mathbf{b}^{(i)} \\ \mathbf{b}^{(e)} \end{bmatrix}$$

Nous pouvons donc dire que trouver un programme optimal réalisable fini pour le problème (2.3) et pour son dual (2.4) est équivalent à trouver une solution finie pour le système d'inéquations suivant:

$$
\begin{cases}
\mathbf{A}^{(ip)}\,\mathbf{x}^{(p)} + \mathbf{A}^{(iq)}\,\mathbf{x}^{(q)} \leq \mathbf{b}^{(i)} \\
\mathbf{A}^{(ep)}\,\mathbf{x}^{(p)} + \mathbf{A}^{(eq)}\,\mathbf{x}^{(q)} = \mathbf{b}^{(e)} \\
\mathbf{y}^{(i)t}\,\mathbf{A}^{(ip)} + \mathbf{y}^{(e)t}\,\mathbf{A}^{(ep)} \geq \mathbf{c}^{(p)} \\
\mathbf{y}^{(i)t}\,\mathbf{A}^{(iq)} + \mathbf{y}^{(e)t}\,\mathbf{A}^{(eq)} = \mathbf{c}^{(q)} \\
\mathbf{c}^{(p)t}\,\mathbf{x}^{(p)} + \mathbf{c}^{(q)t}\,\mathbf{x}^{(q)} - \mathbf{y}^{(i)t}\,\mathbf{b}^{(i)} - \mathbf{y}^{(e)t}\,\mathbf{b}^{(e)} = 0 \\
\mathbf{x}^{(p)} \geq \mathbf{0} \\
\mathbf{x}^{(q)} \text{ quelconque} \\
\mathbf{y}^{(i)} \geq \mathbf{0} \\
\mathbf{y}^{(e)} \text{ quelconque}
\end{cases}
$$

Remarquons que ce système est aussi un PLS où la fonction-objectif a tous ses coéfficients nuls. On peut donc le standardiser et obtenir le système

$$
\begin{cases}
\mathbf{A}' \,.\, \mathbf{x}' = \mathbf{b}' \\
\mathbf{x}' \geq \mathbf{0}
\end{cases}
$$

où $\mathbf{b}' = [\,\mathbf{b}^{(i)t} \mid \mathbf{b}^{(e)t} \mid \mathbf{c}^{(p)t} \mid \mathbf{c}^{(q)t}\,]^t$,

$$
\mathbf{A}' = \begin{bmatrix}
\mathbf{A}^{(ip)} & \mathbf{A}^{(iq)} & -\mathbf{A}^{(iq)} & \mathbf{I}^{(i)} & \mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{O} \\
\mathbf{A}^{(ep)} & \mathbf{A}^{(eq)} & -\mathbf{A}^{(eq)} & \mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{O} \\
\mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{A}^{(ip)t} & \mathbf{A}^{(ep)t} & -\mathbf{A}^{(ep)t} & -\mathbf{I}^{(p)} \\
\mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{O} & \mathbf{A}^{(iq)t} & \mathbf{A}^{(eq)t} & -\mathbf{A}^{(eq)t} & \mathbf{O} \\
\mathbf{c}^{(p)t} & \mathbf{c}^{(q)t} & -\mathbf{c}^{(q)t} & \mathbf{O} & -\mathbf{b}^{(i)t} & -\mathbf{b}^{(e)t} & \mathbf{b}^{(e)t} & \mathbf{O}
\end{bmatrix}
$$

$\mathbf{x}' = [\mathbf{x}^{(p)t} \mid \mathbf{x}^{(q^+)t} \mid \mathbf{x}^{(q^-)t} \mid \mathbf{u}^t \mid \mathbf{y}^{(i)t} \mid \mathbf{y}^{(e^+)t} \mid \mathbf{y}^{(e^-)t} \mid \mathbf{v}^t]^t$
où $\mathbf{u}$ et $\mathbf{v}$ sont des variables d'écart et où les variables $\mathbf{x}^{(q^+)t}$, $\mathbf{x}^{(q^-)t}$, $\mathbf{y}^{(e^+)t}$ et $\mathbf{y}^{(e^-)t}$ sont obtenues après élimination des variables libres (cfr. règle 3, paragraphe 1.4.2).

Ce problème standard peut être transformé en un problème traitable par l'algorithme de Karmarkar. Utilisons pour cela la méthode décrite au paragraphe 1.4.3 conservant le caractère creux de A'. Nous obtenons le système

$$(2.5)\quad \begin{cases} [\mathbf{A'} \mid \mathbf{0} \mid -\mathbf{b'}/\sigma\,] \,.\, \mathbf{x''} = \mathbf{0} \\ [\mathbf{e}^t \mid 1 \mid -1] \,.\, \mathbf{x''} = 0 \\ [\mathbf{e}^t \mid 1 \mid 1] \,.\, \mathbf{x''} = 1 \\ \mathbf{x''} \geq \mathbf{0} \end{cases}$$

où $\mathbf{x''} = [x'/2\sigma \mid v/2 \mid x/2] \in R^s$,
et où σ est l'estimation d'une borne supérieure sur la somme des variables.

Trouver un point solution de ce système d'inéquations revient à trouver un point initial réalisable d' un PLS ayant le même système d'inéquations comme système de contraintes. Nous pouvons dès lors appliquer la procédure de recherche d'un point initial réalisable décrite au paragraphe 1.5 au système (2.5). Soit A'' la matrice des contraintes de (2.5), soit $\hat{\mathbf{x}}$ un vecteur quelconque de R^s, résolvons par Karmarkar le problème suivant :

$$(2.6)\quad \begin{cases} \text{Minimiser } \lambda \\ \text{Avec } [\,\mathbf{A''} \mid -\mathbf{A''}.\hat{\mathbf{x}}\,] \,.\, \begin{bmatrix} \mathbf{x''} \\ \lambda \end{bmatrix} = \mathbf{0} \\ [\mathbf{e}^t \mid 1] \,.\, \begin{bmatrix} \mathbf{x''} \\ \lambda \end{bmatrix} = 1 \\ \begin{bmatrix} \mathbf{x''} \\ \lambda \end{bmatrix} \geq \mathbf{0} \end{cases}$$

où $\mathbf{x''} \in R^s$

Puisque nous supposons que le problème possède une solution finie, nous sommes assurés que le minimum de la fonction-objectif du problème (2.6) est zéro et nous obtenons ainsi un programme optimal de notre problème sous forme primale-duale.
Nous pouvons connaître un point de départ pour la résolution de (2.6). En effet, soit $\hat{\mathbf{x}}$ un point de R^s, un point initial du système (2.6) est

$$\begin{bmatrix} \mathbf{x}''^{\circ} \\ \lambda^{\circ} \end{bmatrix} = \frac{1}{\sum_{i=1}^{s} \hat{x}_i + 1} \begin{bmatrix} \hat{\mathbf{x}} \\ 1 \end{bmatrix}$$

En particulier, pour $\hat{\mathbf{x}} = (1, \ldots, 1)^t$, le point de départ est

$$\mathbf{x}''^{\circ} = (1/(s+1), \ldots, 1/(s+1))^t, \quad \lambda^{\circ} = 1/(s+1)$$

2.3.3 Evaluation

Le principal désavantage de la méthode qui vient d'être décrite provient de la taille de la matrice des contraintes du problème mis sous sa forme primale-duale. En effet, soient i le nombre d'inégalités, e le nombre d'égalités, p le nombre de variables non négatives, et q le nombre de variables quelconques du problème sous sa forme primale, la matrice des contraintes du primal-dual est de dimension ((i+e+p+q+1) × 2.(i+e+p+q)). En comparaison, la standardisation du même problème donnerait une matrice des contraintes de dimension ((i+e) × (i+p+2q)). Dans le pire des cas, lorsque le problème initial ne comprend que des égalités et des variables non négatives et que le nombre d'égalités est égal au nombre de variables, la matrice des contraintes du problème primal-dual atteindra huit fois la taille de celle du problème standardisé. Dans le meilleur des cas, la taille de la matrice des contraintes du

primal-dual fera de deux à trois fois la taille de la matrice des contraintes du problème standardisé.

Dès lors, pourquoi choisir cette méthode plutôt que d'autres qui travaillent à partir d'un problème sous forme standard et fournissent un moyen d'obtenir la solution duale [GAY], [YE] ? Nous considérerons cette question au point de vue de la taille des données à manipuler et du temps nécessaire pour effectuer les calculs. Notons d'abord que si en réalité la taille de la matrice du primal-dual peut atteindre quatre fois la taille de la matrice du primal, très souvent le nombre de ses éléments non nuls sera moins de deux fois plus élevés. Cette propriété peut être exploitée à bon escient par l'utilisation de matrices creuses (cfr. section 3.2). Remarquons ensuite que, en pratique, beaucoup de problèmes traités par la programmation linéaire simple sont des problèmes comportant bon nombre d'inégalités. Dès lors, la taille du problème primal-dual est rarement huit fois plus élevée que celle du problème standard. On pourrait aussi arguer que les variantes de l'algorithme de Karmarkar proposées par Gay et Ye nécessitent le calcul d'un pseudo-inverse(1) pour obtenir la solution duale et que cela implique un supplément d'espace de stockage et de temps calcul. Toutefois ce supplément est bien moindre que celui requis pour la résolution par passage au primal-dual puisque d'une part, dans les deux méthodes, la complexité d'une itération de l'algorithme est approximativement de l'ordre $O(m^3)$ et puisque d'autre part le stockage du pseudo-inverse peut être factorisé.

Un autre fait important que l'on peut mentionner est que les méthodes de Gay et de Ye nécessitent une première phase pour l'algorithme afin d'obtenir un point de départ, ce qui nécessite un travail supplémentaire de l'ordre de $O(n^{3.5}.L^2)$ (cfr. paragraphe 1.3.6 et section 1.1) tandis que la méthode primale-duale dispose d'un point de départ direct.

En conclusion les seules raisons valables que nous pouvons avancer pour expliquer notre choix sont celles-ci: L'algorithme de Karmarkar tel que nous l'avons developpé au chapitre 1 ne nécessite aucune modification pour fournir la solution duale; nous évitons

(1) Moore-Penrose generalized inverse.

une première phase nécessaire pour trouver un point de départ et l'utilisation de matrices et vecteurs creux peut amoindrir de façon sensible le supplément de place et de temps calcul occasionné. Ceci dit, si nous n'avions pas été intéressé par la connaissance de la solution duale, notre choix se serait orienté différemment.

2.4 Calcul de la projection via la méthode des moindres carrés

2.4.1 Introduction

Lors de chaque itération, nous devons calculer la projection d'un vecteur sur un sous-espace, en l'occurence la projection de **Dc** sur **B**. Nous avons vu que cette projection pouvait-être calculée par

$$\mathbf{d} = -\left(\mathbf{I}-\mathbf{B}^t(\mathbf{B}\mathbf{B}^t)^{-1}\mathbf{B}\right) . \mathbf{Dc} \qquad (2.7)$$

Cependant ce calcul implémenté sur ordinateur risque d'être fort couteux en place et en temps puisqu'il inclut le calcul de l' inverse d'une matrice symétrique. Or, il s'avère que calculer une telle projection revient à trouver le résidu d'un problème des moindres carrés.

2.4.2 Méthode des moindres carrés

Le calcul de la projection de **D.c** sur l'espace nul de **B** peut être effectué via la résolution d'un problème des moindres carrés. En effet, calculer la projection d'un vecteur sur un sous-espace revient à minimiser la distance entre ce vecteur et tout vecteur de ce sous-espace.

Nous pouvons donc effectuer les calculs suivants en lieu et place de (2.7).

1) résoudre le problème des moindres carrés:
trouver $\mathbf{x} = \min \| \mathbf{B}^t.\mathbf{x} - \mathbf{D}.\mathbf{c} \|$
(ce qui revient à trouver **x** tel que :
$\mathbf{B}^t.\mathbf{x} = \mathbf{D}.\mathbf{c} \Leftrightarrow \mathbf{B}.\mathbf{B}^t.\mathbf{x} = \mathbf{B}.\mathbf{D}.\mathbf{c} \Leftrightarrow \mathbf{x} = (\mathbf{B}.\mathbf{B}^t)^{-1}.\mathbf{B}.\mathbf{D}.\mathbf{c}$)

2) calculer le résidu (c'est-à-dire la différence entre **D.c** et $\mathbf{B}^t.\mathbf{x}$)

$\mathbf{d} = \mathbf{D.c} - \mathbf{B}^t.\mathbf{x}$

$(\Leftrightarrow \mathbf{d} = \mathbf{D.c} - \mathbf{B}^t.(\mathbf{B}.\mathbf{B}^t)^{-1}.\mathbf{B}.\mathbf{D.c} \Leftrightarrow \mathbf{d} = (\mathbf{I} - \mathbf{B}^t.(\mathbf{B}.\mathbf{B}^t)^{-1}.\mathbf{B}).\mathbf{D.c})$

3) trouver le négatif de **d**

$\mathbf{d} = -\mathbf{d}$

Afin de calculer $\mathbf{x} = \min || \mathbf{B}^t.\mathbf{x} - \mathbf{D.c} ||$, nous effectuerons la décomposition QR de $\mathbf{B}^t$. Il s'agit d'une décomposition de la matrice $\mathbf{B}^t$ telle que :

$$\mathbf{Q}.\mathbf{B}^t = \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{O} \end{bmatrix} = \mathbf{R},$$

où $\mathbf{B}^t$ étant de dimension $(n \times m)$, $n \geq m$ et de rang plein, $\mathbf{R}_1$ est une matrice triangulaire supérieure de dimension $(m \times m)$ et **Q** est une matrice orthogonale ($\mathbf{Q}.\mathbf{Q}^t = \mathbf{I} = \mathbf{Q}^t.\mathbf{Q}$) de dimension $(m \times n)$, pouvant être exprimée sous forme d'un produit de m matrices hermitiennes élémentaires[1] .

$$\mathbf{Q} = \mathbf{H}_m.\mathbf{H}_{m-1} \ldots \mathbf{H}_2.\mathbf{H}_1$$

Soient

$$\mathbf{Q}.\mathbf{D.c} = \begin{bmatrix} \hat{\mathbf{c}}_1 \\ \hat{\mathbf{c}}_2 \end{bmatrix} \text{ et } \mathbf{R} = \mathbf{Q}.\mathbf{B}^t = \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{O} \end{bmatrix},$$

nous pouvons calculer le vecteur **d**, projection de **D.c** sur l'espace nul de **B** via

$$\mathbf{x} = \mathbf{R}_1^{-1}.\hat{\mathbf{c}}_1 \quad \text{et} \quad \mathbf{d} = \mathbf{D.c} - \mathbf{B}^t.\mathbf{x}$$

[1] traduction de l'anglais "elementary hermitian matrices" Aussi appelées "householder transformations", et "elementary reflectors" dans la littérature mathématique [LIN]

2.4.3 algorithme mis en oeuvre

La matrice **Q** pouvant s'exprimer comme un produit de m matrices hermitiennes élémentaires nous calculerons **R** et $\hat{\mathbf{c}}_1$ de façon itérative selon l'algorithme suivant:

$$
\begin{array}{l}
\mathbf{R} := \mathbf{B}^t,\ \hat{\mathbf{c}} := \mathbf{D}.\mathbf{c} \\
\text{pour I de 1 à m} \\
\quad \text{calculer } \mathbf{H}_I \\
\quad \mathbf{R} := \mathbf{H}_I.\mathbf{R} \\
\quad \hat{\mathbf{c}} := \mathbf{H}_I.\hat{\mathbf{c}}
\end{array}
$$

Lors de chaque itération nous déterminons une matrice hermitienne élémentaire et nous en effectuons le produit avec le résultat du produit de l'itération précédente de manière à obtenir

$$\mathbf{R} = \mathbf{H}_I .(\mathbf{H}_{I-1} \ldots (\mathbf{H}_2.(\mathbf{H}_1.\mathbf{B}^t)) \cdots)$$
$$\hat{\mathbf{c}} = \mathbf{H}_I .(\mathbf{H}_{I-1} \ldots (\mathbf{H}_2.(\mathbf{H}_1.\mathbf{D}.\mathbf{c})) \cdots)$$

Lors de le $I^{ième}$ itération, la matrice hermitienne élémentaire $\mathbf{H}_I$ sera choisie de telle sorte que:

1) Elle soit orthogonale. Ainsi le produit des I matrices $\mathbf{H}_I \ldots \mathbf{H}_1$ sera lui-même orthogonal et en fin d'algorithme le produit $\mathbf{H}_m \ldots \mathbf{H}_1$ équivalent à la matrice **Q** sera orthogonal, répondant ainsi aux propriétés attendues pour **Q**.

2) Elle ait la structure suivante:

$$\mathbf{H}_I = \begin{bmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{O} & \mathbf{H}_I' \end{bmatrix} \quad \text{de dimension } (n \times n)$$

avec $\mathbf{H}_I'$ de dimension $(n\text{-}I+1) \times (n\text{-}I+1)$.

Ainsi, si nous supposons que, à la lième itération, la matrice **R** et le vecteur $\hat{\mathbf{c}}$ puissent être partitionnés de la manière suivante

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}''' & \mathbf{R}'' \\ \mathbf{O} & \mathbf{R}' \end{bmatrix} \quad \text{et } \hat{\mathbf{c}} = \begin{bmatrix} \hat{\mathbf{c}}'' \\ \hat{\mathbf{c}}' \end{bmatrix} \qquad (2.8)$$

avec $\mathbf{R}'''$ triangulaire supérieure de dimension (l-1) × (l-1)
$\mathbf{R}''$ de dimension (l-1) × (m-l+1)
$\mathbf{R}'$ de dimension (n-l+1) × (m-l+1)
$\hat{\mathbf{c}}''$ de dimension (l-1) × 1
$\hat{\mathbf{c}}'$ de dimension (n-l+1) × 1,

le produit de $\mathbf{H}_l$ par **R** donne

$$\begin{bmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{O} & \mathbf{H}_l' \end{bmatrix} \begin{bmatrix} \mathbf{R}''' & \mathbf{R}'' \\ \mathbf{O} & \mathbf{R}' \end{bmatrix} = \begin{bmatrix} \mathbf{R}''' & \mathbf{R}'' \\ \mathbf{O} & \mathbf{H}_l'.\mathbf{R}' \end{bmatrix}$$

De cette façon, les l-1 premières lignes et les l-1 premières colonnes de **R** restent inchangées. Le produit de $\mathbf{H}_l$ par $\hat{\mathbf{c}}$ donne

$$\begin{bmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{O} & \mathbf{H}_l' \end{bmatrix} \cdot \begin{bmatrix} \hat{\mathbf{c}}'' \\ \hat{\mathbf{c}}' \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{c}}'' \\ \mathbf{H}_l'.\hat{\mathbf{c}}' \end{bmatrix}$$

et laisse les l-1 premières composantes de $\hat{\mathbf{c}}$ inchangées.

3) Son produit avec **R** introduise des zéros dans les n-l dernières composantes de la $l^{ième}$ colonne de la matrice résultat du produit, résultat que nous rangeons dans la matrice **R**. Ainsi lors de la (l+1)ième itération, la matrice **R** pourra être repartitionnée comme souhaité en (2.8) et, en fin d'algorithme, **R** aura bien la structure attendue.

Afin que ces trois propriétés sur $\mathbf{H}_l$ soit verifiées, nous déterminerons $\mathbf{H}_l$ de la façon suivante:

$$\mathbf{H}_l = \begin{bmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{O} & \mathbf{H}_l' \end{bmatrix} \quad \text{de dimension } (n \times n)$$

$\mathbf{H}_l' = \mathbf{I} - (\mathbf{u}_l.\mathbf{u}_l^t) / \rho_l$, de dimension $(n-l+1) \times (n-l+1)$

$\rho_l = 1 + r'_{11} / (\sigma.\|\mathbf{R}'_{(1)})$ où $\mathbf{R}'_{(1)}$ est la $l^{ième}$ col. de R'

$\mathbf{u}_l = \mathbf{e}_1 + \mathbf{R}'_{(1)} / (\sigma.\|\mathbf{R}'_{(1)})$

$$\sigma = \begin{cases} +1 & \text{si } r'_{11} \geq 0 \\ -1 & \text{si } r'_{11} < 0 \end{cases}$$

L'algorithme final se présente donc comme suit :

$\mathbf{R} := \mathbf{B}^t$

$\hat{\mathbf{c}} := \mathbf{D}.\mathbf{c}$

pour l de 1 à m

  calculer $\mathbf{u}_l$

  pour j de 1 à m-l+1

    $\mathbf{R}_{(i)}' := (\mathbf{I} - \mathbf{u}_l.\mathbf{u}_l^t / \rho_l) . \mathbf{R}_{(i)}'$ (2.9)

  $\hat{\mathbf{c}}' := (\mathbf{I} - \mathbf{u}_l.\mathbf{u}_l^t / \rho_l) . \hat{\mathbf{c}}'$ (2.10)

dans lequel le calcul (2.9) est équivalent à

$t := \mathbf{u}_l^t . \mathbf{R}_{(i)}' / \rho_l$

$\mathbf{R}_{(i)}' := \mathbf{R}_{(i)}' - t.\mathbf{u}_l$

et le calcul de (2.10) est équivalent à

$t := \mathbf{u}_l^t . \hat{\mathbf{c}}' / \rho_l$

$\hat{\mathbf{c}}' := \hat{\mathbf{c}}' - t.\mathbf{u}_l$

2.4.4 Evaluation

Supposons que la matrice **B** soit de dimension (m × n) avec n "pas beaucoup plus grand" que m. L'algorithme mis en oeuvre pour trouver la matrice **R** et le vecteur $\hat{\mathbf{c}}_1$ demande environ $nm^2 - m^3/3$ multiplications et autant d'additions [LIN], soit en prenant n égal à m, $2m^3/3$ multiplications et additions. L'inversion de la matrice **R**, triangulaire supérieure demande environ $m^3/6$ multiplications et autant d'additions. Pour obtenir approximativement un résultat équivalent, le calcul explicite de la projection via (2.7) requérerait environ $9m^3/6$ multiplications et additions, ($m^3/2$ pour l'inversion de $\mathbf{B}.\mathbf{B}^t$ et m^3 pour les produits matriciels de **B** par $\mathbf{B}^t$ et de $\mathbf{B}^t$ par $(\mathbf{B}.\mathbf{B}^t)^{-1}$) contre $5m^3/6$ pour le calcul via la méthode des moindres carrés.

Naturellement ces résultats ne tiennent pas la comparaison avec le travail demandé à chaque itération par l'algorithme du simplexe (un pivotage de matrice) puisque celui-ci est de l'ordre de $O(m^2)$.

chapitre 3

IMPLÉMENTATION

3.1 Introduction

Une implémentation de l'algorithme de Karmarkar a été réalisée. L'algorithme implémenté est celui présenté au chapitre 1 amélioré par les méthodes décrites au chapitre 2. Nous avons également implémenté un petit outil apte à améliorer le cadre de travail de l'utilisateur. Afin de donner une idée assez complète de ce qui a été réalisé, nous présentons à la section 3.3 la découpe modulaire qui a guidé notre travail. Les spécifications externes des modules qui ont demandé une implémentation de notre part sont données à la section 3.4. Enfin, la section 3.5 fournit un petit exemple d'utilisation du logiciel. La section 3.1 concerne le concept de matrice creuse. Celui-ci est largement utilisé à travers tout le logiciel. Nous le présentons ici car il intervient de façon très sensible dans les performances d'espace mémoire utilisé et de temps de résolution.

3.2 Matrices et vecteurs creux

3.2.1 Introduction

Il n'est pas rare en programmation linéaire simple que l'on ait à traiter des problèmes dont la matrice des contraintes possède un faible pourcentage d'éléments différents de zéro. De telles matrices sont appelées "matrices creuses". Il est intéressant, lors de l'implémentation d'un algorithme basé sur le traitement de matrices de s'attarder à l'examen de cette propriété.

En effet, si la matrice est grande, il se peut qu'elle ne puisse être stockée dans son entiereté en mémoire interne (ceci dépend de la taille de la matrice et de la capacité de mémoire interne dont on dispose). Dans ce cas, l'usage d'une mémoire externe se révèlera nécessaire. Le transfert entre mémoire interne et mémoire externe étant fort couteux en temps, ceci entraînera un dégradation notoire des performances de l'algorithme.

Par contre si la matrice est grande mais relativement creuse, en n'en stockant que les éléments non nuls, il sera possible, dans les limites de capacité de la mémoire interne, d'en conserver une représentation en mémoire. De plus, bon nombre d'opérations incluant des éléments nuls seront exécutées alors que leur résultat peut trivialement être connu à l'avance (par exemple, une multiplication par un élément nul donne zéro). Nous pouvons donc espérer améliorer les performances de l'algorithme également au niveau du temps d'exécution.

D'aventure, si la capacité de mémoire interne est toujours trop faible, on pourra néanmoins diminuer le temps de transfert entre mémoire interne et mémoire externe puisqu'on ne transfèrera que des éléments non nuls.

3.2.2 Représentation choisie.

Parmi les différentes manières de représenter une matrice creuse, nous avons choisi celle connue sous le nom de "représentation par listes chaînées" [TEW]. Nous y avons apporté une légère modification en introduisant le concept de "prise de terre".(1)

Les explications qui suivent considèrent le cas d'une matrice mémorisée "colonne par colonne" mais rien n'empêche une mémorisation "ligne par ligne". Remarquons aussi qu'un vecteur creux n'est qu'un cas particulier de matrice creuse ayant un nombre de colonne égal à un.

Soit **A** la matrice de dimension (m × n) que nous désirons représenter. Chaque éléments a_{ij} non nul est mémorisé comme élément de sa colonne j. Chaque élément non nul d'une colonne j est représenté par un triplet (i,v,a) où la composante i désigne l'indice ligne de l'élément, la composante v est la valeur de cet élément et la composante a est l'adresse du triplet représentant l'élément non nul suivant dans la colonne j. Une colonne j est donc representée par une liste chaînée de triplet (i,v,a). Le dernier élément de cette liste sera, par convention, l'élément "prise de terre" représenté par le triplet (32767, 0., 0). L'utilisation de cette "prise de terre" permet de généraliser les traitements puisque dans notre cas toute liste chaînée possède au moins un triplet.

Chaque liste chaînée (et donc chaque colonne) peut être identifiée par l'adresse de son premier élément non nul. Afin d'avoir la représentation complète de la matrice nous devons donc mémoriser ces adresses. Pour cela nous utiliserons une table CA dont chaque élément j est l'adresse du premier élément de la liste chaînée représentant la $j^{ème}$ colonne de la matrice **A**.

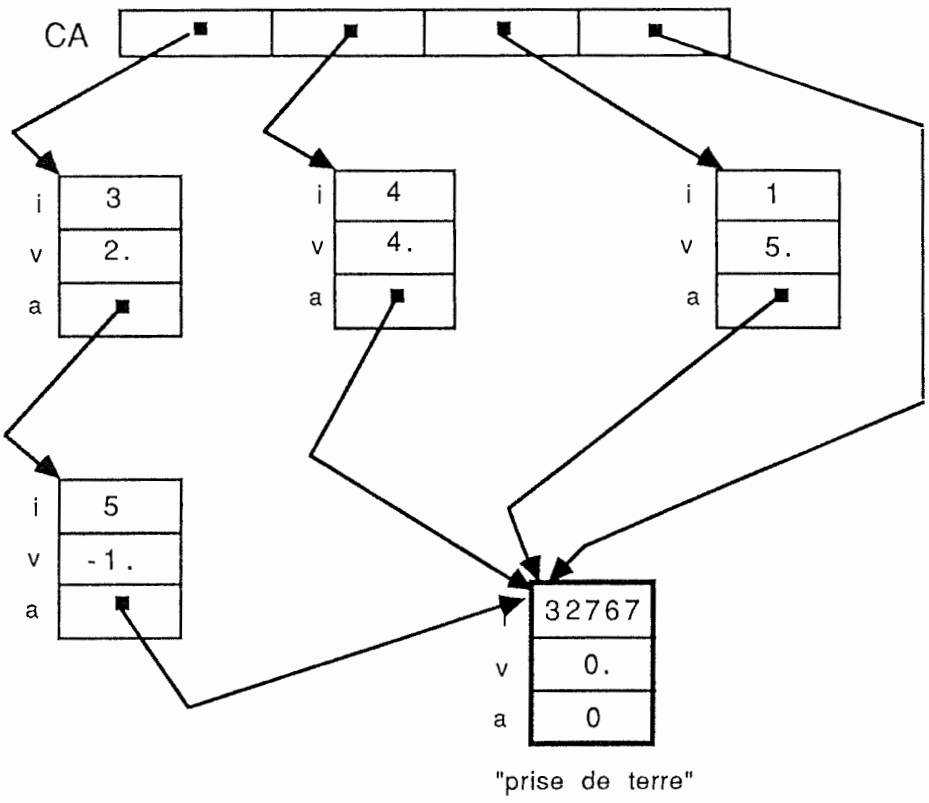
Il est clair que, afin que les dimensions de la matrice **A** soient respectées, chaque composante i des triplets (à l'exception du triplet "prise de terre") sera inférieure au nombre de ligne m et que seuls les n premiers éléments de la table CA doivent être considérés.

(1) Nom choisi par analogie avec les prises de terre électriques.

exemple.
Soit la matrice suivante à représenter:

$$\begin{bmatrix} 0 & 0 & 5. & 0 \\ 0 & 0 & 0 & 0 \\ 2. & 0 & 0 & 0 \\ 0 & 4. & 0 & 0 \\ -1. & 0 & 0 & 0 \end{bmatrix}$$

Sa représentation sera donnée par :



3.2.3 Evaluation

Le principal avantage qu'offre cette représentation par rapport à d'autres représentations est que les triplets ne doivent pas

nécessairement être contigüs. De ce fait, les traitements d'insertion et de suppression de triplets, lorsque suite à un traitement quelconque un élément nul devient non nul et vice-versa, seront plus simples et plus rapides.

En outre, puisque nous utiliserons les fonctions d'allocations dynamiques de mémoire offertes dans la librairie STDLIB du langage C pour allouer l'espace nécessaire au stockage des triplets, nous pourrons ignorer comment est faite cette allocation.

Il reste à savoir à partir de quel moment la représentation que nous utiliserons sera plus avantageuse qu'une représentation classique par table à deux dimensions. Pour cela, négligeons la taille de la table CA, de l'élément "prise de terre" et du code supplémentaire pour les traitements. Soit **A**, matrice de dimension $(m \times n)$, τ le nombre d'éléments non nuls de **A**, t_t la taille en octets d'un triplet (i,v,a) et t_v la taille en octets de la composante v de ce triplet.

Notre représentation sera avantageuse, au point de vue de l'espace mémoire nécessaire, lorsque l'on aura la relation

$$\tau.t_t < m.n.t_v$$

c'est-à-dire lorsque

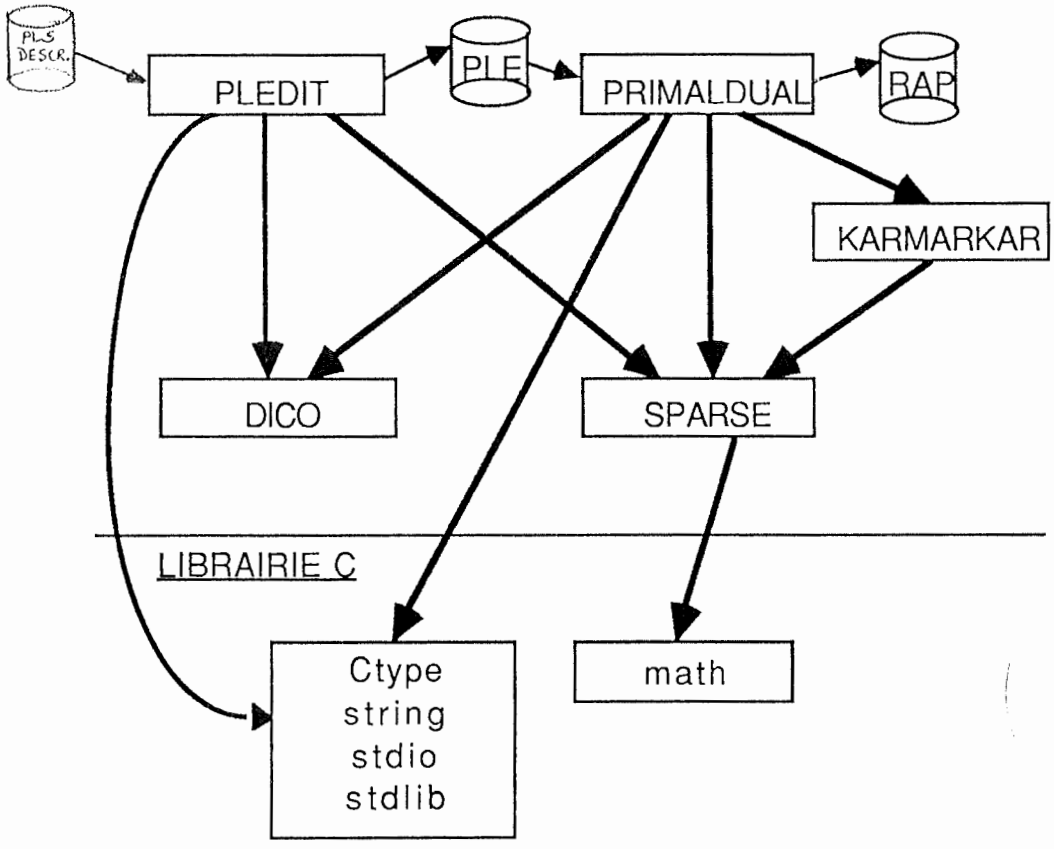
$$\tau / (m.n) < t_v / t_t$$

En ce qui concerne notre implémentation, nous avons t_v égal à 8 et t_t égal à 14. On peut donc conclure que la représentation par liste chaînée est avantageuse lorsque la matrice possède moins de 57 % d'éléments non nuls.

En ce qui concerne le coût en temps, il est plus difficile de juger. Tout dépend du type de traitement que nous désirons faire sur la matrice. Notons toutefois que ce sont les traitements d'insertion et de parcours "non classique" des éléments de la matrice (par exemple le parcours d'une colonne du dernier au premier élément) qui seront les plus couteux.

3.3 Découpe modulaire

On souhaite disposer d'une implémentation de l'algorithme de Karmarkar auquel on apporte les quelques améliorations enoncées au chapitre 2. On désire également disposer d'un cadre de travail relativement convivial permettant l'encodage clair et simple des données de PLS de "faible" taille. La décomposition modulaire effectuée est donnée dans le schéma ci-dessous. Chaque cadre désigne un module. Les flèches reliant deux modules expriment la relation 'utilise' existant entre ces modules.



structure hiérarchique des modules

Ceci nous permet d'identifier les modules qui nécessiteront une implémentation de notre part. Nous pouvons d'ores et déjà donner une brève description des leurs fonctionnalités.

module PLEDIT : Ce module sert à effectuer la traduction d'un fichier contenant la description d'un PLS sous un format simple et clair en un fichier d'entrée adéquat pour le module PRIMALDUAL. La construction du fichier contenant la description du PLS peut être effectuée en utilisant un éditeur quelconque.

module PRIMALDUAL : le module sert à résoudre le problème donné en entrée sous un format adéquat par l'algorithme de Karmarkar en utilisant la méthode de passage au primal-dual décrite à la section 2.3. Un rapport concernant cette résolution est fourni en sortie.

module KARMARKAR : Ce module permet d'isoler l'algorithme amélioré de Karmarkar qui sert à la résolution par passage au primal-dual.

module SPARSE : Ce module fournit quelques facilités d'assez bas niveau pour l'utilisation de matrices et vecteurs creux.

module DICO : Ce module offre quelques facilités pour le traitement de chaînes de caractères. (traitement des noms de variables et des noms de contraintes.

3.4 Spécifications externes des modules

3.4.1 Spécification externe du module PLEDIT

Ce module offre des facilités d'encodage sous forme conviviale d'un problème de programmation linéaire simple. Il est en effet possible à l'utilisateur de représenter des PLS possèdant des caractéristiques suivantes : Contraintes d'égalités, d'inégalités de type '≥' ou '≤', bornes supérieures et/ou inférieures sur certaines des variables, fonction linéaire à maximiser ou minimiser.

Le module est constitué d'un programme en langage C répondant aux spécifications données ci-dessous.

arg : string fnamein (par saisie écran)
fichier_texte filein
string fnameout (par saisie écran)
string fnamedia (par saisie écran)

pré : fnamein, fnameout et fnamedia répondent aux règles de désignation d'un fichier

Rés : fichier_texte fileout

post: • Si le fichier filein de nom fnamein respecte toutes les règles de description d'un PLS (données ci-après), alors fileout est un fichier de nom fnameout respectant le format PLE (décrit ci-après) et les relations entre filein et fileout répondent aux règles de correspondance données ci-après.

• Si le fichier filein de nom fnamein ne respecte pas toutes les règles de description d'un PLS alors fileout est un fichier diagnostique de nom fnamedia. Ce fichier reprend les informations, dans le même ordre, du fichier filein. Chaque information ne répondant pas aux règles de description d'un PLS est suivie en fin de *phrase* (cfr.règles de description) d'un message explicitant la ou les erreurs rencontrées.

Règles de description d'un PLS[1]

La description d'un PLS est constituée de quatre sections : Les sections "scénario", "objectif", "variables" et "contraintes". Ces sections sont obligatoires et doivent apparaître dans cet ordre. Nous utiliserons un certain nombre de symboles pour décrire la syntaxe des *phrases* constituant ces sections. En voici la signification:

1.accolades: $\begin{pmatrix} \text{proposition 1} \\ : \\ \text{proposition n} \end{pmatrix}$ alternative entre les n propositions énoncées.

2. crochets: [proposition] proposition facultative.

3. petits points: [proposition]··· proposition facultative pouvant être répétée un nombre illimité de fois.

4. symboles gras: symboles terminaux.

5. digit: un des symboles terminaux '0', ... , '9'.

6. cnb: un des symboles terminaux alphanumériques ou spéciaux différents de ' ', ';', '=', tab, newline, formfeed.

7. c: un symbole terminal différent de ';'

a) section "scénario"

Cette section est destinée à donner un nom au PLS qui va être décrit. Elle est composée d'une et une seule phrase dont la syntaxe est

$\begin{pmatrix} \mathbf{S} \\ \mathbf{s} \end{pmatrix}$ [cnb]··· **=** nom_de _scénario **;**

Où nom_de_scénario est constitué d'au moins 1 et d'au plus 50 caractères non blancs.

b) section "objectif"

Cette section est destinée à décrire le type d'optimisation du PLS. Elle est composée d'une et une seule phrase dont la syntaxe est

$\begin{pmatrix} \mathbf{O} \\ \mathbf{o} \end{pmatrix}$ [cnb]··· **=** [-] optimisation [c]··· **;**

où optimisation est '**MIN**' ou '**min**' pour une minimisation et '**MAX**' où '**max**' pour une maximisation.

(1) Ces règles s'inspirent en partie de [OMP]

c) section "variables"

Cette section est destinée à décrire les variables du PLS. Toute variable décrite est supposée répondre à une contrainte de non-négativité. Chaque variable est caractérisée par un nom, une contribution à la fonction-objectif, une borne supérieure et une borne inférieure. A chaque variable décrite correpond une phrase dont la syntaxe est

$$\begin{pmatrix} X \\ x \end{pmatrix} = \text{nom\_de\_var1 [\$nombre1] [> nombre2] [< nombre3] ;}$$

où

• nom_de_var1 est constitué d'au moins un caractère. Le premier caractère doit être alphabétique, les caractères suivants peuvent être alphanumériques, "." ou "_".

• nom_de_var1 est unique.

• l'option [$nombre1] désigne la contribution de la variable à la fonction-objectif. Si cette option est absente, la contribution est supposée être nulle.

• l'option [> nombre2] désigne la borne inférieur sur la variable. Si cette option est absente, la borne inférieure est supposée être 0.

• l'option [< nombre3] désigne la borne supérieure sur la variable. Si cette option est absente, la borne supérieure est supposée être $+\infty$.

• nombre1, nombre2, nombre3 respecte la syntaxe de *nombre* qui est

$$\textit{nombre} ::= [-] \begin{pmatrix} \text{digit [digit]...} \\ \text{digit [digit]. digit} \\ \text{.digit[digit]...} \end{pmatrix}$$

d) section "contraintes"

Cette section est destinée à décrire les contraintes du PLS. Chaque contrainte est caractérisée par un nom et un énoncé de contrainte. A chaque contrainte du PLS correspond une phrase de syntaxe suivante:

$$\begin{pmatrix} C \\ c \end{pmatrix} = \text{nom\_de\_contr1} \quad \textit{membre} \quad \begin{pmatrix} > \\ = \\ < \end{pmatrix} \quad \textit{membre} ;$$

$$\textit{membre} ::= \textit{elt\_somme} \left[\begin{pmatrix} + \\ - \end{pmatrix} \textit{elt\_somme} \right] \cdots$$

$$Coefficient ::= \begin{pmatrix} nombre \\ [-]\ (\ expression\_arythm\ ) \end{pmatrix}$$

$$elt\_somme ::= \begin{pmatrix} [-]\ \text{nom\_de\_var2} \\ coéfficient\ [*]\ \text{nom\_de\_var2} \\ coéfficient \\ [-]\ [\ membre\ ] \\ coéfficient\ [*]\ [\ membre\ ] \end{pmatrix}$$

$$expression\_arythm ::= coéfficient\ \left[\begin{pmatrix} + \\ - \\ * \\ / \end{pmatrix} coefficient\ \right]$$

où nom_de_var2 est soit identique à un nom_de_var1, nom d'une variable décrite dans la section "variables", soit constitué du caractère "X" ou "x", directement suivi d'un nombre entier compris entre 1 et le nombre de variables décrites dans la section "variables". Dans ce dernier cas, le nom_de_var 'Xn' désigne la $n^{ième}$ variable décrite dans la section "variables".

Format PLE

Enregistrement 1 (type 1):

S=nom_de_scénario2

Enregistrement 2 (type 2):

$$O=\begin{pmatrix} + \\ - \end{pmatrix}\begin{pmatrix} MAX \\ MIN \end{pmatrix}$$

Enregistrements 3 à N_1 (type 3):

X=nom_de_var3

Enregistrements N_1+1 à N_2 (type 4):

c nombre_virg_flot1 numero1

Enregistrements N_2+1 à N_3 (type 5):

u nombre_virg_flot2 numero2

Enregistrements N_3+1 à N_4 (type 6):

l nombre_virg_flot3 numero3

Enregistrements N_4+1 à N_5 (type 7):

type $\begin{pmatrix} > \\ = \\ < \end{pmatrix}$ nom_de_contr2

Enregistrements N_5+1 à N_6 (type 8):

coeff numéro4 numéro5 nombre_virg_flot4

Règles de correspondances

- nom_de_scénario1 = nom_de_scénario2.
- +**MAX** dans enregistrement de type 2 ⇔ **MAX** ou **max** dans la section "objectif".
- -**MAX** dans enregistrement de type 2 ⇔ -**MAX** ou -**max** dans la section "objectif".
- +**MIN** dans enregistrement de type 2 ⇔ **MIN** ou **min** dans la section "objectif"
- -**MIN** dans enregistrement de type 2 ⇔ -**MIN** ou -**min** dans la section "objectif".
- A chaque variable décrite dans la section "variables" correspond un enregistrement de type 3 où nom_de_var3 = nom_de_var1.
- A chaque option [$nombre1] présente dans la section "variables" correspond un enregistrement de type 4 où
  - valeur de nombre_virg_flot1 = valeur de nombre1,
  - numero1 = numero d'ordre, dans la section "variables", de la description de variable où est spécifiée cette option.
- A chaque option [> nombre2] présente dans la section "variables" correspond un enregistrement de type 6 où
  - valeur de nombre_virg_flot3 = valeur de nombre2,
  - numero3 = numero d'ordre, dans la section "variables", de la description de variable où est spécifiée cette option.
- A chaque option [< nombre3] présente dans la section "variables" correspond un enregistrement de type 5 où
  - valeur de nombre_virg_flot2 = valeur de nombre3,
  - numero2 = numero d'ordre, dans la section "variables", de la description de variable où est spécifiée cette option.

• A chaque contrainte décrite dans la section "contraintes" correspond, dans le même ordre, un enregistrement de type 7 où nom_de_contr 2 = nom_de_contr1 et où ">", "<" ou "=" est le même que dans la section "contraintes".

• Pour chaque contrainte décrite dans la section "contraintes", après transformation de cette contrainte de sorte qu'elle peut s'écrire selon la syntaxe

$$\text{nombre8 nom\_de\_var8 } [\begin{pmatrix}+\\-\end{pmatrix} \text{nombre8 nom\_de\_var8}]\cdots \begin{pmatrix}>\\=\\<\end{pmatrix} \text{ nombre10}$$

Où nombre8 et nombre1à suivent la syntaxe de *nombre* ,

- à chaque nombre8 différent de 0 correspond un enregistrement de type 8 où
  - valeur de nombre_virg_flot4 = valeur de nombre8,
  - numero4 = numero d'ordre de la description de la variable de nom nom_de_var8 dans la section "variables",
  - numero5 = numéro d'ordre de la description de la contrainte dans la section "contraintes".
- à nombre10 différent de 0 correspond un enregistrement de type 8 où
  - valeur de nombre10 = valeur de nombre_virg_flot4,
  - numéro4 = 0,
  - numero5 = numéro d'ordre de la description de la contrainte dans la section "contraintes".

3.4.2 Spécification externe du module SPARSE

Le module SPARSE offre des facilités pour l'utilisation de matrices et vecteurs creux. La représentation utilisée est celle par listes chaînées décrite à la section 3.2.

Le module peut être décomposé en deux parties, chacune d'elle décrivant un objet. La première concerne les coefficients d'une matrice ou d'un vecteur creux, la deuxième a trait au réservoir de coefficients de matrices ou vecteurs creux.

A) Coefficient d'une matrice ou d'un vecteur creux

Dans le cas de la représentation d'une matrice par listes chaînées, chaque coéfficient a_{ij} non nul est un triplet (indice, val, next). La composante indice de ce triplet indique la place de l'élément dans sa colonne (indice i) ou dans sa ligne (indice j) suivant que l'utilisateur représente sa matrice colonne par colonne ou ligne par ligne. Ces spécifications sont décrites pour le cas de représentation colonne par colonne. La composante val est la valeur du coéfficient représenté par le triplet. Cette valeur est, en principe, non nulle. La composante next est l'adresse de l'élément non nul suivant dans la liste chaînée représentant la colonne où se trouve ce coéfficient.

Une occurence de cet objet est définie dans le module et mise à disposition. Il s'agit de l'élément "prise de terre" (cfr. paragraphe 3.2.2). Cet objet est désigné par la variable C "sparse_null" de type SPARSE_ELT. L'adresse de cet variable est "P_SPARSE_NULL".

Le cycle de vie d'un coéfficient est composé de 3 phases : la déclaration ou réservation, l'utilisation et la libération. Ces phases doivent respecter cet ordre. Les phases de réservation et de libération dépendent de l'existence de l'occurence d'un objet réservoir (cfr. spécif. du réservoir)

Déclaration.

La déclaration se fait par une déclaration, en langage C, d'une variable de type SPARSE_ELT.

syntaxe: SPARSE_ELT variable-name [,variable-name]...;

Réservation.

La réservation d'un coéfficient se fait par une déclaration, en langage C, d'une variable de type pointeur vers un SPARSE_ELT et par un appel à la fonction "reserver" (voir spécification pour l'objet réservoir).

syntaxe: SPARSE_ELT *variable-name [,*variable-name]...;

Utilisation.

Trois fonctions sont mises à disposition de l'utilisateur afin de pouvoir accèder aux composantes indice, val et next d'un coéfficient déclaré ou réservé et de pouvoir en modifier les valeurs.

a) fonction d'accès et de modification de la composante next.

description : Cette fonction permet d'accèder à la valeur de la composante next d'un coéfficient ou d'en modifier la valeur.

nom de fonction (macro): next

arg : SPARSE_ELT p (par référence)

pré : -

Rés : SPARSE_ELT q (par référence)

post: L'utilisation de cette macro est équivalente à l'utilisation d'une variable en langage C, de type pointeur vers SPARSE_ELT, désignant l'adresse du coéfficient suivant dans la liste chaînée. Elle suit donc les règles d'utilisation d'une variable en C, y compris l'affectation qui permet ainsi d'en modifier la valeur.

syntaxe : SPARSE_ELT *next(SPARSE_ELT p);

b) fonction d'accès et de modification de la composante indice.

description : Cette fonction permet d'accèder à la valeur de la composante indice d'un coéfficient ou d'en modifier la valeur

nom de fonction (macro): indice

arg : SPARSE_ELT p (par référence)

pré : -

Rés : int i (par valeur)

post: L'utilisation de cette macro est équivalente à l'utilisation d'une variable, en langage C, de type int. Elle suit donc les règles d'utilisation d'une variable en C, y compris l'affectation qui permet ainsi d'en modifier la valeur.

syntaxe : int indice(SPARSE_ELT p);

c) fonction d'accès et de modification de la composante val.

description : Cette fonction permet d'accèder à la valeur de la composante val d'un coéfficient ou d'en modifier la valeur

nom de fonction (macro): val

arg : SPARSE_ELT p (par référence)

pré : -

Rés : double v (par valeur)

post: L'utilisation de cette macro est équivalente à l'utilisation d'une variable, en langage C, de type double. Elle suit donc les règles d'utilisation d'une variable en C, y compris l'affectation qui permet ainsi d'en modifier la valeur.

syntaxe : double val(SPARSE_ELT p);

Libération.

La libération d'un coéfficient se fait par un appel à la fonction "libérer" (voir spécification de l'objet réservoir).

B) Réservoir de coéfficients

Un réservoir de coefficients est un objet grâce auquel on peut réserver et libérer des coéfficients. Ceci permet de mieux utiliser la mémoire primaire de l'ordinateur utilisé grâce à la réutilisation, lors des réservations, des coéfficients précedemment libérés. Cet objet n'est utilisable qu'avec des systèmes permettant l'allocation dynamique de mémoire. Un réservoir de coéfficients est également appelé "freelist".

Le cycle de vie d'une freelist se compose de 3 phases : la déclaration, l'initialisation et l'utilisation. Ces phases doivent respecter cet ordre.

Déclaration.

La déclaration d'une freelist se fait par une déclaration, en langage C, d'une variable de type FREELIST.

syntaxe: FREELIST variable-name {,variable-name}...;

Initialisation.

L'initialisation a pour but de rendre opératoire une freelist déclarée. Lors de cette phase, l'utilisateur peut déterminer la taille initiale, en nombre de SPARSE_ELT, qu'il souhaite pour sa freelist.

a) Fonction d'initialisation d'une freelist.

description : Cette fonction initialise une freelist et en détermine la taille initiale.

nom de fonction : init_freelist

arg : FREELIST f (par référence)
int n (par valeur)

pré : n est strictement positif

Rés : SPARSE_ELT e (par référence)

post: • Si le système permet l'allocation des n SPARSE_ELT, e prend une valeur strictement positive.

- La taille de la freelist f est initialisée à n SPARSE_ELT.
- La taille des extensions ultérieures de la freelist f est fixée à 20% de la valeur n, avec arrondi à l'unité supérieure.
- Si la requête d'allocation ne peut être satisfaite par le système, e prend la valeur 0.

SPARSE_ELT *init_freelist(FREELIST f, int n);

Utilisation

Trois fonctions C permettent l'utilisation d'une freelist. Ces fonctions peuvent être utilisées à plusieurs reprises. Aucun ordre n'est imposé sur l'utilisation de ces fonctions.

a) modification de la taille des extensions

description : Cette fonction est destinée à modifier la taille, en nombre de SPARSE_ELT, des éventuelles extensions ultérieures.
nom de fonction : set_ext_size
arg : FREELIST f (par référence)
int n (par valeur)
pré : n est un entier strictement positif
Rés : f
post: La taille des éventuelles extensions ultérieures de la freelist f est fixée à n SPARSE_ELT.
syntaxe: FREELIST *set_ext_size(FREELIST f, int n)

b) réservation d'un coéfficient

description : Cette fonction permet à l'utilisateur d'obtenir un coéfficient à partir d'une freelist
nom de fonction : reserver
arg : FREELIST f (par référence)
pré : -
Rés : SPARSE_ELT p (par référence)
f

post: Si la demande d'obtention d'un SPARSE_ELT peut être satisfaite, un sparse_elt est retiré de la freelist f et p est la référence de ce sparse_elt. Autrement, la freelist f n'est pas modifiée et p vaut 0.

syntaxe : SPARSE_ELT *reserver(FREELIST f);

c) libération d'un coéfficient

description : Cette fonction permet de remettre dans une freelist un sparse_elt dont on n'a plus besoin.

nom de fonction (macro): liberer

arg : FREELIST f (par référence)

SPARSE_ELT p (par référence)

pré : -

Rés : f

post: Le sparse_elt p est remis dans la freelist f.

syntaxe : SPARSE_ELT *liberer(FREELIST f, SPARSE_ELT p);

3.4.3 Spécification externe du module KARMARKAR

Le module KARMARKAR consiste en une fonction en langage C, implémentation d'une version améliorée de l'algorithme de Karmarkar.

L'algorithme implémenté est dérivé de l'algorithme présenté à la section 1.3 (cfr. en particulier le paragraphe 1.3.6). Les modifications apportées à cet algorithme sont les suivantes : Le calcul de la projection du vecteur $\mathbf{D}.\mathbf{c}$ sur l'espace nul de $\mathbf{B}$ se fait par la méthode des moindres carrés. La section 2.4 décrit ce calcul. La valeur du pas α est recalculée à chaque itération de l'algorithme en utilisant la méthode décrite dans la section 2.2 où l'on choisit α égal à 0.96.

Outre ces modifications, l'algorithme implémenté se caractérise par une utilisation mixte de matrices et vecteurs creux (cfr. section 3.2) et de matrices et vecteurs pleins ainsi que par une allocation dynamique de mémoire primaire pour le stockage de ces matrices et vecteurs.

Ce module utilise les facilités offertes par le module SPARSE.

L'algorithme résoud un PLS de la forme

$$(3.1)\quad \begin{cases} \text{Minimiser} \quad \mathbf{c}^t.\mathbf{x} & \\ \text{Avec} \quad \mathbf{A}.\mathbf{x} = \mathbf{0} & (3.1a) \\ \mathbf{e}^t.\mathbf{x} = 1 & (3.1b) \\ \mathbf{x} \geq \mathbf{0} & \end{cases}$$

Où $\mathbf{x}$ et $\mathbf{c}$ sont des vecteurs de R^n,
$\mathbf{A}$ est une matrice de dimension $(m \times n)$.
$\mathbf{e}$ est le vecteur de R^n dont toutes les composantes sont égales à 1.

pour autant que les conditions suivantes soient vérifiées:

• Le problème possède une solution optimale finie telle que le minimum de la fonction-objectif $\mathbf{c}^t.\mathbf{x}$ est zéro.
• On connait un point de départ pour l'algorithme. Ce point est une solution admissible du système formé des contraintes (3.1a) et (3.1b), et ses composantes sont toutes strictement positives.
• La matrice $\mathbf{B} := \begin{bmatrix} \mathbf{A} & \mathbf{D} \\ \hline \mathbf{e}^t & \end{bmatrix}$ est de rang plein.

Les données en entrée et en sortie sont de deux natures: celles concernant le PLS à résoudre (matrice **A**, vecteur **c**, point de départ, solution obtenue) et celles propres à l'exécution de l'algorithme (précision du test d'arrêt, nombre maximum d'itérations, nombre d'itérations effectuées, statut de l'exécution, temps CPU utilisé).

nom de fonction : KARMARKAR

arg : • chaîne de SPARSE_ELT C (par référence)
(Liste chaînée representant le vecteur C, vecteur des contributions à la fonction-objectif).
• table de références vers des sparse_elt CA (par référence)
(Tables dont les éléments sont les références vers les listes chaînées de sparse_elt constituant les colonnes de la matrice A).
• table de double d (par référence)
(point de départ de l'algorithme).
• int m (par valeur)
(nombre de lignes de la matrice **A**).
• int n (par valeur)
(nombre de colonnes de la matrice **A**, nombre de variables du PLS).
• double epsilone (par valeur)
(valeur de ε pour le test d'arrêt $\mathbf{c}^t.\mathbf{x} < \varepsilon$).
• int maxiter (par valeur)
(nombre maximum d'itérations pour l'algorithme).

pré : • La représentation de **c** et de **A** sous forme de vecteur et matrice creux déterminées par C et CA respectent les dimensions ($m \times n$) pour **A** et ($n \times 1$) pour **c**.

• La table d est la représentation sous forme de vecteur plein d'un point de départ pour l'algorithme. Ce point est donc une solution à composantes strictement positives du système donné par (3.1a) et (3.1b).

• La valeur de epsilone est strictement positive.

• La valeur de maxiter est strictement positive.

Rés : • d

• int statut (par valeur)

• int niter (par référence)

• string tps_CPU (par affichage a l'écran)

post: • Si aucun problème d'allocation de mémoire primaire ne survient, et si l'algorithme trouve une solution admissible telle que la valeur de la fonction-objectif est strictement inférieure à epsilone en moins de maxiter itérations, alors d peut être considéré comme solution optimale du PLS, niter est le nombre d'itérations effectuées pour obtenir cette solution, le temps CPU utilisé pour la résolution est affiché a l'écran, et le statut prend la valeur PROBLEM_SOLVED.

• Si aucun problème d'allocation de mémoire primaire ne survient, et si l'algorithme ne trouve pas de solution admissible telle que la valeur de la fonction-objectif est strictement inférieure à epsilone en un nombre d'itérations inférieur ou égal à maxiter, alors d ne peut être considéré comme solution optimale du PLS, niter est égal à maxiter, le temps CPU utilisé pour la tentative de résolution est affiché a l'écran, et le statut prend la valeur MORE_THAN_MAX.

• Si aucun problème d'allocation de mémoire primaire ne survient, et si l'algorithme en arrive à constater que la matrice B n'est pas de rang plein, alors d ne peut être considéré comme solution optimale du PLS, niter est le nombre d'itérations complètement effectuées et le statut a pour valeur REDUNDANT_PL.

• Si l'algorithme rencontre des problèmes d'allocation de mémoire primaire, d ne peut être considéré comme solution optimale du PLS, niter est le nombre d'itérations complètement effectuées et le statut a pour valeur UNABLE_TO_ALLOC.

syntaxe : int KARMARKAR(SPARSE_ELT *C, SPARSE_ELT **CA, double d, int m, int n, double epsilone, int maxiter, int *niter);

3.4.4 Spécification externe du module PRIMALDUAL

Le module PRIMALDUAL consiste en un programme en langage C permettant d'obtenir la solution primale et la solution duale d'un PLS par transformation de ce PLS en sa forme primale-duale selon la méthode décrite à la section 2.3 et en faisant appel au module KARMARKAR pour la résolution de ce nouveau problème.

Le PLS est donné en entrée dans un fichier texte. Ce fichier décrit le PLS en utilisant un format propre à notre implémentation. Nous appelerons ce format le "format PLE". Ce fichier texte est normalement obtenu en sortie du module PLEDIT. Un rapport est fourni en sortie du module PRIMALDUAL dans un fichier texte.

Le programme implémenté dans le module PRIMALDUAL se caractérise également par l'utilisation de matrices et vecteurs creux et par une allocation dynamique de mémoire primaire pour le stockage de ces matrices et vecteurs creux par l'utilisation d'un objet réservoir (cfr. spécif du module SPARSE).

arg : • string fnamein (par saisie écran)
- fichier texte filein
- string fnameout (par saisie écran)
- double eps (par saisie écran)
- double upperbound (par saisie écran)

pré : • le string fnamein répond aux règles de désignation d'un fichier et est le nom d'un fichier texte filein sous format PLE (la description de ce format est donnée dans la spécification du module PLEDIT).
- Le string fnameout répond au règles de désignation d'un fichier.
- eps est strictement supérieur à 0.
- upperbound est strictement supérieur à 0.

Rés : • fichier texte fileout
- string temps_CPU (par affichage a l'écran)

post: • Le fichier texte fileout a pour nom filenameout. Ce fichier fournit un rapport sur la tentative de résolution du PLS donné dans filein par l'algorithme implémenté dans le module KARMARKAR après transformation

en primal-dual. Cette transformation est faite conformément à la méthode décrite à la section 2.3 en utilisant upperbound comme borne supérieure sur la somme des variables (cfr. paragraphe 1.4.3 et 2.3.2). L'argument eps est utilisé comme argument epsilone pour le module KARMARKAR, l' argument niter est fixé à 100.

Le rapport contient les renseignements suivants :

- Informations sur le primal : Nombre de variables, de contraintes, de contraintes d'inégalité, d'égalité et de borne, et nombre de coéfficients non nuls pour la matrice **A**.
- Informations sur le primal-dual : Nombre de variables, de contraintes et nombre de coéfficients non nuls pour la matrice des contraintes, borne supérieure sur la somme des variables utilisée.
- Informations sur la résolution par KARMARKAR : valeur de ε utilisée pour le test d'arrêt, nombre d'itérations effectuées, statut de l'exécution.

Si, après exécution de KARMARKAR, le statut a pour valeur PROBLEM_SOLVED, le rapport contient aussi les renseignements suivants :

- Solution du primal-dual : Valeur et type des variables du primal-dual. type 'X' = variable issue du primal, type 'Y' = variable issue du dual, type 'V' = variable d'écart, type 'W', 'K' et 'S' = variables v/2, ξ/2 (cfr. paragraphe 1.4.3 et 2.3.2) et λ (cfr. paragraphe 2.3.2).
- Solution du primal : Nom et valeur des variables du primal.
- Solution du dual : Nom et valeur des variables du dual.

3.4.1 Spécification externe du module DICO

Le module DICO décrit un objet appelé "dico" et les opérations associées à cet objet.

Un dico est un ensemble de couples (string,no_ordre). La composante string d'un couple est une chaîne de caractères. La composante no_ordre est un entier positif et représente un numéro d'ordre attribué au couple. Ce numéro est déterminé par l'ordre d'ajout du couple au dico. Ainsi, lorsque le dico est constitué de n couples (string,no_ordre), ces couples peuvent être ordonnés du premier au $n^{ième}$ de telle sorte que le premier ait sa composante no_ordre égale à 1, le deuxième sa composante égale à 2 et ainsi de suite jusqu'au $n^{ième}$ couple.

Le cycle de vie d'un dico est composé de 3 phases: la déclaration, l'initialisation et l'utilisation. Ces phases doivent respecter cet ordre.

Déclaration.

La déclaration d'un dico se fait par une déclaration, en langage C, d'une variable de type DICO. La déclaration met un dico à la disposition de l'utilisateur.

syntaxe: DICO variable-name [,variable-name]...;

Initialisation.

L'initialisation a pour but de rendre opératoire un dico déclaré. Le dico est initialisé au vide, il n'est donc constitué d'aucun couple. Lors de cette phase d'initialisation, l'utilisateur peut définir son dico "UNIQUE" ou "MULTIPLE". Un dico "UNIQUE" est tel que la composante string de tous ses couples est différente de celle des autres couples du dico. Un dico "MULTIPLE" peut contenir des couples dont la composante string est identique à celle d'autres couples du dico.

a) fonction d'initialisation d'un dico.

nom de fonction : init_dico
arg : DICO d (par référence)
char t (par valeur)
pré : t est égale à UNIQUE ou MULTIPLE.
Rés : d
post: Le dico d est défini "UNIQUE" ou "MULTIPLE" selon la valeur de t. Le dico d est initialisé au vide.
syntaxe : DICO *init_dico(DICO d, char t);

Utilisation.

Au cours de cette phase, l'utilisateur peut ajouter des couples à son dico et accèder aux informations contenues dans le dico. Trois fonctions sont à disposition de l'utilisateur. Ces fonctions peuvent être utilisées à plusieurs reprises durant la phase d'utilisation. Aucun ordre n'est imposé sur ces fonctions.

a) fonction d'ajout.

description : Cette fonction permet d'ajouter un couple où seule la composante string est déterminée par l'utilisateur à un dico.
nom de fonction : add_dico
arg : DICO d (par référence)
string s (par valeur)
pré : -
Rés : d
int n (par valeur)
post: • Si la capacité du dico (2000 caractères pour l'ensemble des composantes string) n'est pas dépassée, et si le dico d est défini "multiple", le couple (s,n) où n est le numéro d'ordre attribué à ce couple est ajouté au dico d.
• Si la capacité du dico n'est pas dépassée, et si le dico d est défini "unique", le couple (s,n) où n est le numéro d'ordre attribué à ce couple est ajouté a d pour autant qu'aucun autre couple du dico n'ait sa composante string égale à s. Si ce n'est pas le cas, n prend la valeur 0 et le dico reste inchangé

syntaxe : int add_dico(DICO d, string s);

b) fonction d'accès à la composante string.

description : Cette fonction permet d'accèder à la composante string du couple dont la composante no_ordre est déterminée par l'utilisateur.

nom de fonction : read_dico

arg : DICO d (par référence)
int n (par valeur)

pré : -

Rés : string s (par valeur)

post: • S'il existe dans le dico d un couple dont la composante no_ordre est égale à n, s prend la valeur de la composante string de ce couple.
• S'il n'existe pas dans d de couple dont la composante no_ordre est égale à n, s prend la valeur "".

syntaxe : string read_dico(DICO d, int n);

c) fonction d'accès à la composante no ordre.

description : Cette fonction permet d'accèder à la composante no_ordre d'un ou du couple dont la composante string est déterminée par l'utilisateur.

nom de fonction : dico_search

arg : DICO d (par référence)
string s (par valeur)

pré : -

Rés : int n (par valeur)

post: • S'il existe dans d un seul couple dont la composante string est égale à s, n prend la valeur de la composante no_ordre de ce couple.
• S'il existe dans d plusieurs couples dont la composante string est égale à s, n prend la valeur de la composante no_ordre d'un de ces couples.
• S'il n'existe pas dans d de couple dont la composante string est égale à s, n prend une valeur négative.

syntaxe : int dico_search(DICO d, string s);

3.1 Exemple d'utilisation

l'exemple que nous proposons est issu de [FIC] .

Enoncé

L'entreprise POLLUX est spécialisée dans la fabrication du gaz amoniac (NH_3) et du chlorure d'amonium (NH_4Cl). Actuellement, elle dispose de 50 unités d'azote, 180 d'hydrogène et 40 de chlore. Lorsqu'elle vend sur le marché une unité de gaz amoniac, elle réalise un bénéfice de 40 frs. Une unité de chlorure d'amonium lui rapporte un bénéfice de 50 frs. Elle souhaiterait établir un plan de production optimal, basé sur ses bénéfices et l'état actuel de ses stocks.

Soit x1 et x2 les quantités produites en NH3 et NH4Cl respectivement. Le problème de l'entreprise Pollux peut se représenter par le problème de programmation linéaire suivant :

$$\begin{cases} \text{Maximiser} & 40\,x_1 + 50\,x_2 \\ \text{Avec} & x_1 + x_2 \leq 50 \\ & 3\,x_1 + 4\,x_2 \leq 180 \\ & x_2 \leq 40 \\ & x1 \geq 0 \\ & x2 \geq 0 \end{cases}$$

Utilisation du logiciel

A l'aide d'un éditeur quelconque nous construisons le fichier suivant :

```
scenario = POLLUX;
objectif = MAXimiser les benefices;

X = NH3      $ 40 ;
X = NH4Cl    $ 50 ;

C = stock_azote =       NH3 +    NH4Cl   < 50;
C = stock_hydro =    3 NH3 + 4 NH4Cl   < 180;
C = stock_chlore=                NH4Cl   < 40;
```

Une fois complet, ce fichier est soumis au programme PLEDIT afin d'en obtenir une traduction sous format PLE.

```
work>run pledit

Fichier en entree (PLS) : pollux.txt
Fichier en sortie (format PLE) : pollux.ple
Fichier diagnostique : pol.dia

SCENARIO <POLLUX>
traduction effectuee.
Fichier de sortie correct.
```

Le fichier sous format PLE obtenu en sortie du programme PLEDIT est le suivant :

```
S=POLLUX
O=+MAX
X=NH3
X=NH4Cl
c 5.000000e+01      2
c 4.000000e+01      1
type < stock_azote
type < stock_hydro
type < stock_chlore
coeff      0      1 5.000000e+01
coeff      0      2 1.800000e+02
coeff      0      3 4.000000e+01
coeff      1      1 1.000000e+00
coeff      1      2 3.000000e+00
coeff      2      1 1.000000e+00
coeff      2      2 4.000000e+00
coeff      2      3 1.000000e+00
```

Il nous reste à soumettre ce fichier au programme PRIMALDUAL afin de résoudre le problème.

```
work>run primaldual

fichier en entree (format PLE) : pollux.ple
fichier en sortie (RAPPORT) : pollux.rapport

precision du test d'arret : 1e-7
borne plausible sur la somme des variables : 16e2
SCENARIO <POLLUX>

->Adjonction des contraintes de bornes a la matrice des coefficients.
->Adjonction des contraintes duales a la matrice des coefficients.
->Adjonction de la contrainte : FE primale - FE duale = 0.
->Ajout des variables d'ecart pour les contraintes d'inegalite.
->Ajout d'une contrainte et de deux variables pour l'algo de KARMARKAR.
->Ajout d'une variable de depart.
->Execution de l'algorithme de KARMARKAR.
  iteration  11    valeur de la fonction objectif: 3.514871e-08
 ELAPSED:    0 00:00:00.73  CPU: 0:00:00.13  BUFIO: 11  DIRIO: 0  FAULTS: 2
```

Les résultats obtenus sont les suivants :

```
*****************************************
*** RAPPORT D'EXECUTION DE PRIMALDUAL ***
*****************************************
SCENARIO : POLLUX

Statut de l'execution : PROBLEM_SOLVED
Precision du test d'arret : 1.000000e-07
Borne sur la somme des variables : 1.600000e+03
Nombre maximum d'iterations : 100
Nombre d'iterations effectuees : 11

Donnees sur le PRIMAL :
           2 variables
           3 contraintes
           3 contraintes d'inegalite
           0 contraintes d'egalite
           0 contraintes de borne
           5 coefficients non nuls =  83% de remplissage

Donnees sur le PRIMAL-DUAL :
          13 variables
           7 contraintes (toutes d'egalite)
          37 coefficients non nuls =  40% de remplissage

Solution du PRIMAL-DUAL transforme pour Karmarkar :

|     No ref.(type)     valeur        |
-----------------------------------------
    X    1 (type X)    = 0.006250
    X    2 (type X)    = 0.009375
    X    3 (type Y)    = 0.003125
    X    4 (type Y)    = 0.003125
    X    5 (type Y)    = 0.000000
    X    6 (type V)    = 0.000000
    X    7 (type V)    = 0.000000
    X    8 (type V)    = 0.003125
    X    9 (type V)    = 0.000000
    X   10 (type V)    = 0.000000
    X   11 (type W)    = 0.475000
    X   12 (type K)    = 0.500000
    X   13 (type S)    = 0.000000

Solution du PRIMAL :

|    nom de variable                    No.ref.        valeur      |
-------------------------------------------------------------------
    NH3                                (X    1)   = 20.000039
    NH4Cl                              (X    2)   = 30.000140

Solution du DUAL :

|    nom de contrainte                   No.ref.        valeur      |
-------------------------------------------------------------------
    stock_azote                        (X    3)     = 10.000065
    stock_hydro                        (X    4)     = 10.000134
    stock_chlore                       (X    5)     = 0.000035
```

CONCLUSION

L'implémentation que nous avons réalisée de l'algorithme de Karmarkar ne peut certes pas rivaliser avec celles qui ont déjà été faites pour l'algorithme du simplexe. Néanmoins, elle devrait se montrer satisfaisante tant au point de vue temps de résolution qu'au point de vue espace de stockage requis, grâce notamment à l'utilisation de matrices creuses et aux améliorations apportées. Il faut toutefois ajouter qu'aucun soin particulier n'a été pris concernant les éventuelles pertes de précision durant l'exécution de l'algorithme. Le test d'arrêt ne nous a pas non plus préoccupé. Il serait encore possible d'améliorer l'algorithme en y apportant d'autres modifications qui permettraient, par exemple, le traitement implicite des contraintes de bornes supérieurs [TO2], ou la diminution progressive de la taille du problème en éliminant les variables tendant vers zéro comme suggéré dans [TUR] ou encore une méthode plus rapide encore de calcul de la projection [TUR].

Malgré tout, l'algorithme ne semble pas en mesure de rivaliser avec l'algorithme du simplexe, tant au point de vue possibilitésde résultats qu'aux points de vue performances informatiques. Signalons cependant que l'algorithme est encore "jeune" par rapport à celui du simplexe et qu'il n'est pas impossible qu'il soit encore étendu et amélioré de façon sensible.

BIBLIOGRAPHIE

[FIC] J. FICHEFET :
"Eléments de programmation linéaire", Faculté Universitaire N. D. De La Paix, institut d'informatique, notes de cours, janvier 1982.

[TO1] P.TOLLA :
"Validation numérique de l'algorithme de Karmarkar", Université de Paris-Dauphine, Cahier du LAMSADE n°76, avril 1987.

[LIS] A. LISSER, N. MACULAN, M. MINOUX :
"Large steps preserving polynomiality in Karmarkar's algorithm", Université de Paris-Dauphine, Cahier du LAMSADE n°77, mai 1987.

[TO2] P. TOLLA :
"Amélioration des performances de l'algorithme de Karmarkar dans le cas de programmes linéaires à variables bornées supérieurement", Université de Paris-Dauphine, Cahier du LAMSADE n°82, novembre 1987.

[TOM] J.A. TOMLIN :
"An experimental approach to Karmarkar's projective method for linear programming", North-Holland, Mathematical programming study 31 (1987), p. 175-191.

[TUR] K. TURNER :
"A variable-metric variant of the Karmarkar algorithm for linear programming", Rice University, Department of mathematical sciences, technical report 87-13, May 1987.

[MER] J-M. MERNIER :
"Une nouvelle approche algorithmique des problèmes de programmation linéaire : l'algorithme de Karmarkar", Faculté Universitaire N.D. De La Paix, Faculté des sciences, Mémoire de fin d'étude, 1986.

[GAY] D.M. GAY :
"A variant of Karmarkar's linear programming algorithm for problems in standard form", North-Holland, Mathematical programming 37 (1987) p. 81-90.

[YE] Y. YE, M. KOJIMA :
"Recovering optimal dual solutions in Karmarkar's polynomial algorithm for linear programming", North-Holland, Mathematical programming 39 (1987) p. 305-317.

[LIN] J.J. DONGARRA, G.B. MOLER, J.R. BUNCH, G.W. STEWART :
"Linpack user's guide"

[TEW] TEWARSON :
"Sparse matrices", Mathematics in science and engineering, volume 99.

[OMP] B. DE DECKER, F. LOUVEAUX, C. MORTIER, G. SCHEPENS, A. VAN LOOVEREN :
"OMP optimisation (linear and mixed integer programming with OMP)", Beyers and Partners PVBA, octobre 1987.

Facultés Universitaires Notre Dame De la Paix
Institut d'Informatique
B-5000 NAMUR

UNE IMPLEMENTATION DE L'ALGORITHME DE KARMARKAR (COMPLEMENT)

promoteur J. FICHEFET

Complément au
mémoire présenté pour l'obtention
du grade de Licencié et Maître en
informatique par

Alain VAN KERM

Année académique 1988-1989

COMPLEMENT :

VALIDATION DE L'ALGORITHME IMPLEMENTE ET EXPERIMENTATION REALISEE

I. Introduction

ce document présente d'une part la manière dont les tests ont été effectués afin de valider le logiciel de résolution de PLS par la méthode primal-duale appliquée à l'algorithme de Karmarkar.

D'autres part, il présente l'expérimentation qui a été réalisée avec ce logiciel pour différents problèmes tests afin de mettre en évidence les différents paramètres qui influencent la justesse des solutions obtenues ainsi que la vitesse de résolution.

II. Validation du logiciel de résolution

La validation du logiciel de résolution a été effectuée en trois parties distinctes selon la nature des PLS à résoudre:

1. Validation pour des PLS non redondants et à solution optimale finie.
2. Validation pour des PLS non redondants et sans solution finie.
3. Validation pour des PLS redondants.

II.1 PLS non redondants et à solution optimale finie

Nous avons procédé de la manière suivante:

a) Validation pour des PLS simples et de petite taille caractérisés par une fonction à maximiser et des contraintes d'inégalités du type '≤', par comparaison des résultats obtenus avec les résultats attendus.

Exemple:

(1)

```
scenario = POLLUX;

objectif = MAXimiser les benefices;

X = NH3      $ 40 ;
X = NH4Cl    $ 50 ;

C = stock_azote =       NH3 +    NH4Cl  < 50;
C = stock_hydro =     3 NH3 + 4 NH4Cl  < 180;
C = stock_chlore=                NH4Cl  < 40;
```

dont les résultats attendus sont :

NH3=20, NH4Cl=30, stock_azote=10, stock_hydro=10, stock_chlore=0.

b) Validation pour des PLS caractérisés par une fonction à maximiser et des contraintes d'inégalités du type '≥' et '≤', par comparaison avec les PLS équivalents obtenus en remplacant chaque inégalité de type '≥' par une inégalité de type '≤' en utilisant la règle donnée au paragraphe 2.3.2.

Exemple:

```
scenario = POLLUX;

objectif = MAXimiser les benefices;

X = NH3      $ 40 ;
X = NH4Cl    $ 50 ;

C = stock_azote =     -[ NH3 +  NH4Cl ]    > -50;
C = stock_hydro =   -[ 3 NH3 + 4 NH4Cl ] > -180;
C = stock_chlore=                NH4Cl  < 40;
```

doit donner des résultats équivalents à ceux de (1)

c) Validation pour des PLS caractérisés par une fonction à maximiser et des contraintes d'inégalités et d'égalités par comparaison avec les PLS equivalents obtenus en remplaçant chaque égalité par deux inégalités de sens contraire.

Exemple:

```
scenario = POLLUX;

objectif = MAXimiser les benefices;

X = NH3      $ 40 ;
X = NH4Cl    $ 50 ;

C = stock_azote =       NH3 +  NH4Cl    = 40;
C = stock_hydro =    3 NH3 + 4 NH4Cl    < 180;
C = stock_chlore=              NH4Cl    < 40;
```

doit donner des résultats équivalents à ceux de

```
scenario = POLLUX;

objectif = MAXimiser les benefices;

X = NH3      $ 40 ;
X = NH4Cl    $ 50 ;

C = stock_azote =  -[ NH3 +  NH4Cl ]  > -40;
C = stock_azote2=      NH3 +  NH4Cl   <  40;
C = stock_hydro =    3 NH3 + 4 NH4Cl  < 180;
C = stock_chlore=              NH4Cl  < 40;
```

d) Validation pour des PLS caractérisés par une fonction à minimiser par comparaison avec les PLS equivalents obtenus en remplaçant la fonction à minimiser par une fonction à maximiser en utilisant la règle 1 du paragraphe 1.4.2.

Exemple:

```
scenario = POLLUX;

objectif = -MINimiser les benefices;

X = NH3      $ -40 ;
X = NH4Cl    $ -50 ;

C = stock_azote =       NH3 +   NH4Cl  < 50;
C = stock_hydro =    3 NH3 + 4 NH4Cl   < 180;
C = stock_chlore=              NH4Cl   < 40;
```

doit donner des résultats équivalents à ceux de (1)

e) Validation pour des PLS comprenant des contraintes de bornes supérieures (inférieures) par comparaison avec les PLS équivalents où les contraintes de bornes sont données en section "contraintes".

Exemple:

```
scenario = POLLUX;

objectif = MAXimiser les benefices;

X = NH3      $ 40 ;
X = NH4Cl    $ 50 <25 >15;

C = stock_azote =       NH3 +    NH4Cl  < 50;
C = stock_hydro =     3 NH3 + 4 NH4Cl  < 180;
C = stock_chlore=               NH4Cl  < 40;
```

doit donner des résultats équivalents à ceux de

```
scenario = POLLUX;

objectif = MAXimiser les benefices; '

X = NH3      $ 40 ;
X = NH4Cl    $ 50 ;

C = stock_azote =       NH3 +    NH4Cl  < 50;
C = stock_hydro =     3 NH3 + 4 NH4Cl  < 180;
C = stock_chlore=               NH4Cl  < 40;
C = borne_sup    =              NH4Cl  < 25;
C = borne_inf    =              NH4Cl  > 15;
```

f) Validation de la solution duale en résolvant le dual d'un PLS dont la solution primale est connue.

Exemple:

```
scenario = DUAL_DE_POLLUX;
objectif = MINimiser;

X = stock_azote    $ 50 ;
X = stock_hydro    $ 180;
X = stock_chlore   $ 40;

C = NH3    = X1 + 3X2       > 40;
C = NH4Cl  = X1 + 4X2 + X3  > 50;
```

doit donner comme solution primale, la solution duale de (1) et comme solution duale, la solution primale de (1).

Dans tous les cas, nous avons obtenus des résultats répondant à nos attentes: l'algorithme a convergé vers une solution optimale.

II.2 PLS non redondants et sans solution finie.

Deux cas sont à considérér: le cas où Il est impossible de trouver une solution optimale pour le PLS et celui où la solution optimale du PLS est infinie.

a) Il est impossible de trouver une solution pour le PLS.

Nous avons incorporé des contraintes entraînant l'inexistence de solution dans un PLS pour lequel nous connaissions une solution optimale finie.

Exemples:

```
scenario = POLLUX;
objectif = MAXimiser les benefices;

X = NH3      $ 40 ;
X = NH4Cl    $ 50 ;

C = stock_azote =       NH3 +    NH4Cl   < 50;
C = impossible  =       NH3 +    NH4Cl   > 60;
C = stock_hydro =     3 NH3 + 4 NH4Cl   < 180;
C = stock_chlore=                NH4Cl   < 40;
```

```
scenario = POLLUX;
objectif = MAXimiser les benefices;

X = NH3      $ 40 ;
X = NH4Cl    $ 50 ;

C = stock_azote =       NH3 +    NH4Cl   < 50;
C = impossible  =                NH4Cl   = 60;
C = stock_hydro =     3 NH3 + 4 NH4Cl   < 180;
C = stock_chlore=                NH4Cl   < 40;
```

b) solution optimale infinie.

Nous avons modifié un PLS pour lequel nous connaissions une solution optimale finie de sorte que la solution optimale soit infinie.

Exemple:

```
scenario = POLLUX;
objectif = MAXimiser les benefices;

X = NH3      $ 40 ;
X = NH4Cl    $ 50 ;

C = stock_azote =       NH3 +    NH4Cl   > 50;
C = stock_hydro =     3 NH3 + 4 NH4Cl   > 200;
C = stock_chlore=                NH4Cl   > 40;
```

Dans chacun de ces cas, l'algorithme s'est comporté comme prévu. Il n'y a pas eu de convergence et le nombre maximum d'itérations permises a été atteint.

II.3 PLS redondants.

Remarquons que seules les contraintes d'égalité du PLS donné peuvent entraîner une redondance puisque les contraintes d'inégalités sont transformées en contraintes d'égalités par l'ajout de variables d'écarts différentes pour chaque inégalités lors de la mise sous forme traitable par l'algorithme de Karmarkar.

a) Nous avons introduit dans un PLS pour lequel nous connaissions une solution optimale finie une égalité combinaison linéaire des autres égalités.

Exemple:

```
scenario = POLLUX;
objectif = MAXimiser les benefices;

X = NH3      $ 40 ;
X = NH4Cl    $ 50 ;

C = stock_azote =       NH3 +    NH4Cl   = 50;
C = redondance  =       NH3 +    NH4Cl   = 50;
C = stock_hydro =     3 NH3 + 4 NH4Cl   < 180;
C = stock_chlore=               NH4Cl   < 40;
```

Dans la plupart des cas, l'algorithme a détecté la redondance mais pas nécessairement à la première itération. Il est arrivé que la redondance n'ait pas été détectée mais il n'y pas eu, dans ce cas, de convergence.

III. Expérimentation

III.1 Cadre d'expérimentation

Cette expérimentation a été réalisée sur système VAX/VMS pourvu d'un processeur QUICK 6220.

Les différents problèmes qui ont servi à l'expérimentation sont donnés au paragraphe III.3.

Lorsqu'un temps d'exécution est mentionné, il s'agit du temps CPU consommé pour une exécution de la fonction KARMARKAR. Le chronomètre est déclenché directement après l'entrée dans la fonction et est arrêté juste avant la sortie de cette fonction. Nous avons pu constater de légères variations dans ce temps CPU consommé lors d'exécutions travaillant avec des données strictement identiques. Ces variations ne sont cependant que de l'ordre de quelques centièmes de secondes. Les temps sont exprimés dela façon suivante :

mm:ss.cc	si le temps dépasse la minute.
ss.cc	si le temps est inférieur à la minute.

III.2 Paramètres intervenant dans l'expérimentation

Les paramètres que nous avons fait varier lors de l'expérimentation sont :

- la valeur de ε pour le test d'arrêt $\mathbf{c}^t.\mathbf{x} < \varepsilon$.
- la façon de déterminer la valeur du multiplicateur α (cfr. paragraphe 1.3.4).
- la valeur du multiplicateur $\overline{\alpha}$ dans la procédure de recherche du plus grand pas (cfr. paragraphe 2.2.2).

- la valeur de σ, borne supérieure sur la somme des variables.

Nous avons essayé de déterminer l'influence qu'ont ces paramètres sur la vitesse de résolution (et donc sur le nombre d'itérations) et sur la justesse des résultats obtenus.

III.3 variation de ε.

Dans le cas du passage par la forme primale-duale le test d'arrêt est en fait $\lambda < \varepsilon$ où λ est une variable artificielle introduite afin de disposer d'un point de départ pour l'algorithme de résolution. L'algorithme converge vers une solution optimale tant que λ converge vers zéro.

Nous avons décidé de résoudre différents problèmes avec différentes valeurs pour ε. Les résultats sont presentés dans les tableaux ci-dessous en même temps que les énoncés des problèmes et de leur caractéristiques de taille. Lorsque se trouve la mention **OSC**, cela signifie que l'on constate une oscillation dans la fonction-objectif qui empêche de vérifier le test d'arrêt. Il y a donc dans ce cas une non-convergence à partir d'un précision donnée.

1) problème POLLUX

```
scenario = POLLUX;
objectif =  MAXimiser les benefices;

X = NH3      $  40 ;
X = NH4Cl    $  50 ;

C = stock_azote =       NH3 +    NH4Cl   < 50;
C = stock_hydro =     3 NH3 + 4 NH4Cl   < 180;
C = stock_chlore=               NH4Cl   < 40;

Donnees sur le PRIMAL :
           2 variables
           3 contraintes
           3 contraintes d'inegalite
           0 contraintes d'egalite
           0 contraintes de borne
           5 coefficients non nuls =  83% de remplissage

Donnees sur le PRIMAL-DUAL :
          13 variables
           7 contraintes (toutes d'egalite)
          37 coefficients non nuls =  40% de remplissage
```

	résultats attendus	1e-5	1e-8	1e-10
NH3	20.	20.11763	19.99987	20.00000
NH4Cl	30.	29.94794	30.00012	30.00000
stock_azote	10.	10.17898	10.00002	10.00000
stock_hydro	10.	9.97164	10.00002	10.00000
stock_chlore	0.	0.07664	0.00003	0.00000
fct-obj.	2300.	2302.102	2300.001	2300.000
nbre d'itér.		7	12	16
temps CPU		00.07	00.18	00.23

2) Problème P1

```
S= PROBLEME_TEST_P1;
O= MIN;

x=v1  $1;  x=v2  $1;  x=v3  $1;

c= C1=    -x1 + x2 - x3 = 4;
c= C2=    -x1      + x3 = 0;

Donnees sur le PRIMAL :
           3 variables
           2 contraintes
           0 contraintes d'inegalite
           2 contraintes d'egalite
           0 contraintes de borne
           5 coefficients non nuls =  83% de remplissage

Donnees sur le PRIMAL-DUAL :
          13 variables
           7 contraintes (toutes d'egalite)
          39 coefficients non nuls =  42% de remplissage
```

	résultats attendus	1e-5	1e-8	1e-10
v1	0.	0.03764	0.00209	0.00000
v2	4.	4.04676	4.00037	4.00000
v3	0.	0.03764	0.00209	0.00000
C1	1.	1.00946	0.99832	1.00000
C2	0.	0.00000	0.00000	0.00000
fct-obj.	4.	4.12204	4.00456	4.00000
nbre d'itér.		6	9	**OSC** (23)
temps CPU		00.07	00.18	00.23

3) Problème P2

```
S= PROBLEME_TEST_P2;
O= MIN;

x=V1 $1;  x=V2 $2;  x=V3 $-3;
x=V4 $4;  x=V5 $1;

c= C1=   V1 + V2                    = 4;
c= C2=        V2 + V3               = 3;
c= C3=             -V3 + V4         = 0;
c= C4=   V1 + V2 + V3 + V4 + 2 V5   = 10;

Donnees sur le PRIMAL :
          5 variables
          4 contraintes
          0 contraintes d'inegalite
          4 contraintes d'egalite
          0 contraintes de borne
         11 coefficients non nuls =  55% de remplissage

Donnees sur le PRIMAL-DUAL :
         21 variables
         11 contraintes (toutes d'egalite)
         77 coefficients non nuls =  33% de remplissage
```

	résultats attendus	1e-5	1e-8	1e-10
V1	4.	3.98042	4.00001	4.00000
V2	0.	0.03654	0.00001	0.00000
V3	3.	2.98047	3.00001	3.00000
V4	3.	2.98047	3.00001	3.00000
V5	0.	0.03654	0.00001	0.00000
C1	0.75	0.75704	0.75436	0.75436
C2	0.50	0.50814	0.50870	0.50871
C3	3.75	3.75932	3.75435	3.75435
C4	0.25	0.24750	0.24565	0.24465
fct-obj.	7.	7.07051	7.00004	7.00000
nbre d'itér.		7	12	15
temps CPU		00.21	00.34	00.45

4) Problème P3

```
S = PROBLEME_TEST_P3;
O = MIN;

x=v1 $1;  x=v2 $-1;  x=v3 $1;  x=v4 $1;
x=v5 $2;  x=v6 $-2;  x=v7 $1;  x=v8;

c=C1=   X1 + X2 + X3                              = 4;
c=C2=   X1      + X3          + X6      + X8      = 27;
c=C3=   X1           + X4 + X5      + X7          = 19;

Donnees sur le PRIMAL :
          8 variables
          3 contraintes
          0 contraintes d'inegalite
          3 contraintes d'egalite
          0 contraintes de borne
         11 coefficients non nuls =  45% de remplissage

Donnees sur le PRIMAL-DUAL :
         25 variables
         13 contraintes (toutes d'egalite)
         88 coefficients non nuls =  27% de remplissage
```

	résultats attendus	1e-5	1e-8	1e-10
V1	0.	0.04789	0.00006	0.00000
V2	4.	3.96513	3.99994	4.00000
V3	0.	0.04062	0.00006	0.00000
V4	9.5	9.42495	9.49990	9.50000
V5	0.	0.17181	0.00020	0.00000
V6	27.	26.92318	26.99993	27.00000
V7	9.5	9.42495	9.49990	9.50000
V8	0.	0.05678	0.00001	0.00000
C1	−1.	−1.00323	−0.99999	−1.00000
C2	−2.	−1.98254	−1.99999	−2.00001
C3	1.	0.99777	1.00001	0.99999
fct-obj.	−39.	−38.52946	−38.99950	−38.99999
nbre d'itér.		7	12	16
temps CPU		00.29	00.55	00.70

5) Problème P4

```
S = PROBLEME_TEST_P4;
O = MIN;

x=v1 $1;  x=v2 $-1;  x=v3 $1;  x=v4 $1;
x=v5 $2;  x=v6 $-2;  x=v7 $1;  x=v8;

c=C1=   X1 + X2 + X3                              = 4;
c=C2=   X1      + X3           + X6      + X8     = 27;
c=C3=                X4 + X5                      = 12;
c=C4=   X1      + X3 - X4                         = 2;
c=C5=                     X5 + X6                 = 12;
c=C6=   X1           + X4 + X5      + X7          = 19;

Donnees sur le PRIMAL :
          8 variables
          6 contraintes
          0 contraintes d'inegalite
          6 contraintes d'egalite
          0 contraintes de borne
         18 coefficients non nuls =  37% de remplissage

Donnees sur le PRIMAL-DUAL :
         31 variables
         16 contraintes (toutes d'egalite)
        124 coefficients non nuls =  25% de remplissage
```

	résultats attendus	1e-5	1e-8	1e-10
V1	4.	3.94136	3.99976	3.99999
V2	0.	0.10313	0.00008	0.00000
V3	0.	0.03130	0.00023	0.00002
V4	2.	1.94756	1.99996	2.00001
V5	10.	10.10065	10.00008	10.00000
V6	2.	1.94756	1.99996	2.00001
V7	3.	3.10809	3.00029	3.00002
V8	21.	21.17546	21.00017	21.00000
C1	–3.	–3.08869	–3.00008	–3.00005
C2	0.	0.01738	0.00001	0.00007
C3	3.	3.10386	3.00003	3.00005
C4	3.	3.11487	3.00009	3.00006
C5	–2.	–2.05237	–2.00001	–2.00007
C6	1.	0.95750	0.99999	1.00005
fct-obj.	25.	25.23136	25.00039	25.00003
nbre d'itér.		7	12	**OSC**
temps CPU		00.44	00.72	11.21

6) Problème P5

```
S= PROBLEME_TEST_P5;
O= MAX;

x=Y1 $139;  x=Y2  $88;  x=Y3 $133;  x=Y4 $137;
x=Y5 $165;  x=Y6     ;  x=Y7     ;  x=Y8     ;
x=Y9     ;  x=Y10    ;  x=Y11    ;

c=C1=   1.4X1 + 1.8X2 + 1.5X3 + 1.4X4 +1.5X5                    < 420.;
c=C2=   9.8X1 + 2.4x2 + 1.4x3 + 1.4x4 +1.4X5                    < 415.;
c=C3=   4.0X1 + 0.4X2 + 1.4X3 + 1.4X4 + 1.4X5                   < 355.;
c=C4=   2.8X1 + 0.6X2 + 1.3X3 + 1.4X4 + 1.5X5 + 5.5X6           < 345.;
c=C5=   2.2X1 + 0.4X2 + 1.3X3 + 1.5X4 + 1.2X5 + 5.5X7           < 160.;
c=C6=   2.2X1 + 0.4X2 + 1.3X3 + 1.3X4 + 1.2X5 + 5.5X8           < 95.;
c=c7=   2.2x1 + 0.6X2 + 1.3X3 + 1.3X4 + 1.2X5 + 5.5X9           < 380;
c=C8=   2.6X1 + 5.8X2 + 1.5X3 + 1.5X4 + 1.2X5 + 5.5X10          < 395.;
C=C9=   0.6X1 + 4.0X2 + 1.3X4 + 5.5X11                          < 270.;
c=C10=  0.6X1 + 1.2X2 + 1.3X3 + 1.3X4 + 2.6X5                   < 230.;
c=C11=  0.6X1 + 1.8X2 + 1.2X3 + 1.2X4 + 1.2X5                   < 310.;
c=C12=  0.6X1 + 1.8X2 + 1.5X3 + 1.4X4 + 1.4X5                   < 420.;
c=C13=  16.X1 + 12.X4 + 35.X5 + 50.X6 + 50.X7                   < 5200.;
c=C14=  20.X1 + 36.X4 + 50.X5 + 50.X8 + 50.X9                   < 5200.;
c=C15=  16.X1 + 12.X4 + 35.X5 + 50.X10 + 50.X11                 < 3600.;
c=C16=  0.1X2 + 0.9X3 + 0.8X4 + 2.3X5 - [X6+X7+X8+X9+X10+X11] = 0;

Donnees sur le PRIMAL :
          11 variables
          16 contraintes
          15 contraintes d'inegalite
           1 contraintes d'egalite
           0 contraintes de borne
          89 coefficients non nuls =  50% de remplissage

Donnees sur le PRIMAL-DUAL :
          57 variables
          29 contraintes (toutes d'egalite)
         310 coefficients non nuls =  18% de remplissage
```

	résultats attendus	1e-5	1e-8	1e-10
Y1	0.	0.60270	0.00040	0.00001
Y2	54.80	49.32048	54.77725	54.78083
Y3	20.35	16.30939	20.91312	20.91623
Y4	0.	6.52846	0.00319	0.00001
Y5	38.85	39.46503	38.24796	38.24703
Y6	40.97	40.81083	41.37534	41.37631
Y7	11.08	9.22174	11.81709	11.81818
Y8	0.	0.05850	0.00009	0.00000
Y9	49.82	49.92961	49.82533	49.82615
Y10	0.	5.36150	0.00287	0.00003
Y11	9.24	10.81302	9.251899	9.25027
fct-obj.	13939.2	13999.26	13913.25	13913.36
nbre d'itér.		8	17	20
temps CPU		02.01	04.27	05.08

Comme l'on pouvait s'y attendre, on constate que plus le problème est grand, plus la solution est imprécise et que le nombre d'itérations necessaire croît avec la taille du problème si l'on désire garder une bonne précision pour la solution.

III.4 Variation dans la détermination de α

Nous avons testé les problèmes en utilisant deux politiques de détermination du multiplicateur α. La première est de donner à α une valeur fixe, la deuxième consiste à appliquer la procédure de recherche d'un grand pas présentée à la section 2.2 avec $\overline{\alpha}$ fixé à 0.96. Voici les résultats obtenus:

	Nbre d'itérations			temps CPU		
	α=0.25	α=0.99	proc.	α=0.25	α=0.99	proc.
Pollux(ε=1e-5)	65	17	7	00.81	00.24	00.07
Pollux(ε=1e-10)	144	36	16	02.01	00.54	00.23
P1(ε=1e-5)	63	16	6	00.88	00.22	00.09
P1(ε=1e-10)	168	48	23	02.08	00.70	00.36
P2(ε=1e-5)	75	19	7	02.18	00.54	00.21
P2(ε=1e-10)	265	53	15	07.05	01.38	00.45
P3(ε=1e-5)	84	22	7	03.39	00.92	00.29
P3(ε=1e-10)	203	52	16	08.46	2.10	00.70
P4(ε=1e-5)	87	22	7	05.20	1.32	00.44
P4(ε=1e-10)	219	54	OSC	13.18	3.24	OSC
P5(ε=1e-5)	120	32	8	30.24	08.08	02.01
P5(ε=1e-10)	290	75	20	1:12.72	19.02	5.08

On peut constater qu'en moyenne, l'exécution de l'algorithme avec α=0.25 demande quatre fois plus d'itérations et de temps CPU qu'avec α=0.99, et environ douze fois plus qu'avec la procédure de recherche. Les différences observées au niveau de la précision de la solution optimale sont minimes. Il faut cependant mettre un cas en évidence: on a pu constater une non-convergence pour le problème P4 avec la procédure de recherche et une convergence avec α fixé. En réalité, pour ce cas, la fonction-objectif se met à osciller dans

l'intervalle]1e-10, 1e-9] et ne vérifie jamais le test d'arrêt fixé à 1e-10.

III.5 variation de $\bar{\alpha}$ dans la procédure de recherche

Nous avons fait varier le multiplicateur $\bar{\alpha}$ dans un intervalle réduit et receuilli les résultats pour quatre des problèmes testés. Les résultats obtenus furent les suivants:

	Nbre d'itérations				temps CPU			
	P1	P3	P4	P5	P1	P3	P4	P5
$\bar{\alpha}$=0.8	22	19	17	21	00.32	00.77	01.14	05.31
$\bar{\alpha}$=0.9	14	16	OSC	20	00.14	00.68	OSC	05.02
$\bar{\alpha}$=0.96	23	16	OSC	20	00.36	00.70	OSC	05.08
$\bar{\alpha}$=0.975	38	19	OSC	20	00.50	00.84	OSC	05.20
$\bar{\alpha}$=.99	12	OSC	OSC	18	00.18	OSC	OSC	04.99

On remarque de grands écarts dans les performances pour le problème P1, les meilleurs de ces performances étant obtenues avec $\bar{\alpha}$=0.9 et $\bar{\alpha}$=0.99. Pour le problème P3 les meilleures résultats sont obtenus lorsque $\bar{\alpha}$=0.9 et $\bar{\alpha}$=0.96. Pour le problème P5 lorsque $\bar{\alpha}$=0.99. Cependant $\bar{\alpha}$=0.99 entraîne une oscillation dans la résolution du problème P4 (Notons que tous les cas d'oscillations se sont rencontrés dans l'intervalle]1e-10, 1e-9]). Il n'est donc pas aisé, au vu de ces résultats, de se prononcer quant à la "bonne" valeur à donner à $\bar{\alpha}$. Il semble que cela dépende en majeure partie du problème à traiter.

Néanmoins, Il apparaît que plus le multiplicateur est choisi "petit", plus les risques d'oscillations diminuent, et donc qu'un "petit" $\bar{\alpha}$ préserve mieux la précision de la solution finale mais augmente le nombre d'itérations et donc le temps de résolution.

III.6 Variation de σ

Rappelons que afin de pouvoir transformer le PLS à résoudre en un PLS traitable par l'algorithme de Karmarkar, nous avions besoin de connaître une borne supérieure sur la somme des variables, σ (cfr. paragraphe 1.4.3). Lors de la résolution par le programme PPRIMALDUAL, il est demandé d'estimer cette borne supérieure et de la fournir en argument. Dans notre cas, le problème que l'on désire transformer est le problème initial mis sous forme primale-duale. Il s'agit donc de tenir également compte des variables duales dans cette estimation.

Différentes valeurs pour cette borne σ ont été essayées pour un même problème. Ce problème était le problème P3 avec ε=1e−10 et $\bar{\alpha}$=0.96. Voici les résultats obtenus pour les variables V1,V2,V6 et pour la fonction-objectif.

σ	V1	V2	V6	fct-obj.
1.5e3	0.00000012	4.00000015	26.99999973	38.999997
1e4	0.00000271	4.00000069	26.99998930	38.999948
1e6	0.00020902	3.99985234	26.99937099	38.996718
1e8	0.01778318	3.98288562	26.97693005	38.786385
1e10	1.50056983	5.03117685	24.26638966	22.850455
1e12	52.59879984	183.53868654	176.47225034	22.439470

On remarque que plus l'estimation de la borne supérieure est éloignée de la réalité, plus la valeur optimale de la fonction-objectif devient imprécise. On en arrive même au point de n'avoir plus aucun chiffre exact pour σ=1e12. Il est donc important pour la justesse de la solution de faire une bonne estimation pour σ. Or il n'est pas toujours évident de trouver une bonne estimation pour σ au premier essai et si l'estimation que l'on a faite est inférieure à la réalité, l'algorithme ne convergera pas.

III.7 Conclusion

Différentes sources d'imprécisions et de variations dans les performances ont pu être décelées. Ces sources sont la taille du problème, une mauvaise valeur pour $\bar{\alpha}$ dans la procédure de recherche d'un grand pas et des valeurs inadaptées pour le test d'arrêt et l'estimation d'une borne supérieure sur la somme des variables. Il est donc nécessaire de tâtonner quelque peu avant d'obtenir les meilleures performances possibles tant au point de vue justesse de la solution qu'au point de vue rapidité d'exécution. Ceci constitute un net désavantage par rapport, par exemple, à la méthode du simplexe qui ne nécessite aucun tâtonnement de ce genre.