TD machine 1

Programmation linéaire

Le but de ce TD est d'utiliser le solveur glpk (à partir du langage python) pour résoudre des problèmes de programmation linéaire. Certains problèmes proposés ci-dessous ont été vus (et pour la plupart d'entre eux déjà résolus...) dans les séances précédentes de TD. On s'intéresse dans ce TD à des problèmes de programmation linéaire de type problème de production (par simplexe), et des problèmes d'affectations vus comme des programmes linéaires en variables entières ou binaires (affectation simple, problème de bureau, problème de plan de vols ...).

Le solveur glpk est utilisé via un module python appelé pymathprog [1]. Cette interface de glpk est à la fois souple et puissante mais un peu longue à décrire (et donc à utiliser) pour un TP de 2H ! Pour faciliter les choses, une fonction `linprog` a été écrite dont voici l'entête et une brève description :

```
def linprog(c, ineq= None, eq= None, sens="min", lb = None, ub = None, typevar = float, verb = 0):
    """
    Une fonction linprog (un peu comme celle de Matlab) pour résoudre des PL en python
    et qui utilise le module pymathprog (qui lui est une vraie interface python sur le
    solveur glpk).

    Utilisation :
       stat, F, X = linprog( c, ineq=(A,b), eq=(Ae,be), sens="min"|"max",
                             lb= , ub=, typevar=float|int|bool, verb=0|1|2|3)
```

Cette fonction permet de résoudre un PL posé sous la forme suivante :

$$\min_{\mathbf{x}} \text{ ou } \max_{\mathbf{x}} f(x) = \mathbf{c}^\top \mathbf{x}$$
$$\begin{cases} A\mathbf{x} \leq \mathbf{b} \\ A_e\mathbf{x} = \mathbf{b}_e \\ \mathbf{lb} \leq \mathbf{x} \leq \mathbf{ub} \end{cases}$$

Pour `linprog`, le seul argument obligatoire est le vecteur c, tous les autres sont des arguments optionnels nommés (pouvant donc être mis dans n'importe quel ordre) :

- `c` doit être un tableau `numarray` 1-d, le nombre d'éléments de ce tableau définit le nombre de variables n du PL.
- `ineq=` si il y a des contraintes d'inégalité vous devez les entrer utilisant `ineq=(A,b)` c'est à dire que l'argument effectif est un t-uple à deux éléments, le premier étant la matrice A (un tableau `numarray` 2-d, dont le nombre de colonnes doit être égal à n et le nombre de lignes m définit donc le nombre de contraintes d'inégalité), et le deuxième le vecteur b (un tableau `numarray` 1-d avec m éléments).
- `eq=` Idem que précédemment mais pour introduire les contraintes d"égalité.
- `sens=` une chaîne de caractère `sens='min'` pour minimiser (défaut) et `sens='max'` pour maximiser.
- `lb` et `ub` permettent respectivement d'introduire des bornes sur la solution x. Ces deux arguments doivent être des `numarray` 1-d à n éléments. Par défaut (si vous ne les définissez pas) on a $lb_i = 0$ et $ub_i = +\infty$, $\forall i$.
- `typevar=` permet de définir la nature des variables `typevar=float` pour des variables continues (défaut), `typevar=int` pour des variables entières et `typevar=bool` pour des variables binaires.

1. Attention ce module est disponible sur les machines de l'école uniquement pour python3 (utilisation obligatoire de python3 et/ou de l'IDE spyder3).

— `verb=` niveau de bavardage du solveur [2] ; par défaut ce paramètre est mis à 0 (aucune sortie) mais vous pouvez obtenir plus ou moins d'informations en le réglant à 1, 2 ou 3.
— En sortie la fonction renvoie 3 arguments :
  — `stat` un indicateur (5 lorsque l'optimum a été trouvé, 6 lorsque f est non minorée/majorée, 4 et 1 lorsque le domaine est vide),
  — `F` la valeur de f à l'optimum (ou `None` si elle n'est pas définie),
  — et `X` la (une) solution optimale (ou `None` si elle n'est pas définie).

Un exemple d'utilisation de `linprog`

Prenons comme exemple l'exercice 1 du TD1 :

$$\begin{array}{l} \max_{x\in\mathbb{R}^2} f(x) = 1700x_1 + 3200x_2 \\ x_1, x_2 \geq 0 \end{array}$$

$$\begin{array}{rcrcl} & & 3x_2 & \leq & 39 \\ 1.5x_1 & + & 4x_2 & \leq & 60 \\ 2x_1 & + & 3x_2 & \leq & 57 \\ 3x_1 & & & \leq & 57 \end{array}$$

Voici un script possible pour résoudre ce problème [3] :

```
# -*- coding: utf-8 -*-
from util_gro_tp1 import linprog, affiche_resultat
import numpy as np

# on définit le vecteur c de la fct coût
c = np.array([1700, 3200])

# on définit la matrice A et le vecteur b des contraintes d'inégalité
A = np.array([ [  0, 3],
               [1.5, 4],
               [  2, 3],
               [  3, 0]])
b = np.array([39, 60, 57, 57])

stat, F, X = linprog(c, ineq=(A,b), sens = "max")
affiche_resultat(stat, F, X)
```

Dans ce problème les inégalités 1 et 4 sont de simples bornes supérieures sur les deux variables. On peut donc utiliser aussi :

```
# -*- coding: utf-8 -*-
from util_gro_tp1 import linprog, affiche_resultat
import numpy as np

# on définit le vecteur c de la fct coût
c = np.array([1700, 3200])

# on définit la matrice A et le vecteur b des contraintes d'inégalité
A = np.array([ [1.5, 4],
               [  2, 3]])
b = np.array([60, 57])
bornes_sup = np.array( [19, 13] )

stat, F, X = linprog(c, ineq=(A,b), ub = bornes_sup, sens = "max")
affiche_resultat(stat, F, X)
```

et si on voulait contraindre les variables à être entières il faudrait rajouter l'argument `typevar=int`.

2. Fonctionne si vous utilisez python3 à partir d'un terminal ; si vous utilisez spider3 lancer le à partir d'un terminal, vous aurez les messages du solveur dans ce dernier.

3. Le fichier module `util_gro_tp1.py` qui contient la fonction `linprog` et d'autres utilitaires sera disponible sur Arche (ou ailleurs...). Dans ce fichier module se trouve une petite fonction d'affichage (`affiche_resultat`) qui est utilisée dans l'exemple.

Exercice 1. *Programmation linéaire*

Résoudre les problèmes suivants (cf. feuille TD3) en utilisant la fonction `linprog` précédente.

(a) Problème de production

$$\begin{cases} \max\left[F = 50x_1 + 40x_2 + 70x_3 + 80x_4\right] \\ 2x_1 + 4x_2 + 8x_3 + 6x_4 \leq 100 \\ 10x_1 + 8x_2 + 6x_3 + 10x_4 \leq 160 \\ x_1 + x_2 + 2x_3 + 2x_4 \leq 20 \\ x_1,\, x_2,\, x_3,\, x_4 \geq 0 \end{cases}$$

(b) optimum unique à base dégénérée

$$\begin{cases} \max\left[F = x_1 + x_2\right] \\ 3x_1 + 2x_2 \leq 15 \\ 3x_1 + 4x_2 \leq 21 \\ x_2 \leq 3 \\ x_1,\, x_2 \geq 0 \end{cases}$$

(c) optimum infini

$$\begin{cases} \max\left[F = x_1 + 3x_2\right] \\ x_1 + x_2 \geq 3 \\ x_1 - 2x_2 \geq 5 \\ -2x_1 + x_2 \leq 5 \\ x_1,\, x_2 \geq 0 \end{cases}$$

(d) ensemble des solutions réalisables vide

$$\begin{cases} \max\left[F = x_1 - x_2\right] \\ x_1 + 2x_2 \leq 5 \\ 2x_1 + x_2 \leq 6 \\ x_1 + x_2 \geq 4 \\ x_1,\, x_2 \geq 0 \end{cases}$$

Exercice 2. *Programmation linéaire en variables* $\{0,1\}$ *– Problème de sac à dos.*

Résoudre chacun des problèmes de "sac-à-dos" suivants en utilisant la fonction `linprog` précédente (ne pas oublier de mettre `typevar=bool`).

$$(P_1)\begin{cases} \max\left[F(x) = 6x_1 + 9x_2 + 8x_3 + 4x_4\right] \\ 4x_1 + 6x_2 + 5x_3 + 4x_4 \leq 8 \\ 3x_1 + 6x_2 + 4x_3 + 3x_4 \leq 7 \\ x_1, x_2, x_3, x_4 \in \{0,1\} \end{cases} \qquad (P_2)\begin{cases} \max\left[F(x) = 6x_1 - 9x_2 + 8x_3 + 4x_4\right] \\ 4x_1 - 6x_2 + 5x_3 - 4x_4 \leq 2 \\ 3x_1 - 6x_2 - 4x_3 + 3x_4 \leq 1 \\ x_1, x_2, x_3, x_4 \in \{0,1\} \end{cases}$$

Exercice 3. *Stockage.*

On dispose de m clefs USB sur lesquelles on veut stocker n fichiers de taille $t_1, \cdots, t_n$. Les clefs USB ont toutes la même capacité de stockage S (en GigaOctets). Tous les fichiers doivent être stockés. Pour qu'un stockage soit admissible il faut que la somme des tailles des fichiers stockés sur une clef soit inférieure ou égale à S (on ne peut pas dépasser la capacité d'une clef...). De plus, un fichier n'est stocké qu'une seule fois sur une seule clef.

Déterminer un stockage consiste alors à déterminer sur quelle clef j est stocké le fichier i donné. On cherche à déterminer le stockage (admissible) des fichiers sur les clefs USB de façon à minimiser le nombre de clefs utilisées.

Modélisation (Rappel)

On introduit les variables suivantes :

- les variables binaires y_j pour indiquer si la clef USB j est utilisée ($y_j = 1$) ou non ($y_j = 0$).
- les variables binaires x_{ij} pour indiquer si le fichier i est stocké sur la clef j ($x_{ij} = 1$) ou non ($x_{ij} = 0$).

Le problème du stockage se modélise alors par le programme linéaire suivant (cf. TD1) :

$$(P)\begin{cases} \min \displaystyle\sum_{j=1}^{m} y_j \\ \displaystyle\sum_{i=1}^{n} t_i x_{ij} \leq S y_j, \quad \forall j = 1, \cdots, m \\ \displaystyle\sum_{j=1}^{m} x_{ij} = 1, \quad \forall i = 1, \cdots, n \\ y_j \in \{0,1\}, \quad \forall j = 1, \cdots, m \\ x_{ij} \in \{0,1\}, \quad \forall i = 1, \cdots, n \end{cases}$$

Travail demandé.

Les variables x_{ij} et y_j doivent être regroupées dans un seul vecteur $\mathbf{x}$ de $(n+1)m$ composantes. On choisit un regroupement *colonne par colonne* :

$$\mathbf{x} = (x_{11}, x_{21}, \cdots, x_{n1} \,|\, x_{12}, x_{22}, \cdots, x_{n2} \,|\, \cdots \,|\, x_{1m}, \cdots, x_{nm} \,|\, y_1, \cdots, y_m)^\top$$

1. Écrire le problème (P) sous la forme suivante

$$(PL) \begin{cases} \min\limits_{\mathbf{x}} F(\mathbf{x}) = \mathbf{p}^\top \mathbf{x} \\ A\mathbf{x} \leq \mathbf{b} \\ B\mathbf{x} = \mathbf{d} \\ \mathbf{x} \in \{0,\, 1\} \end{cases}$$

   en explicitant les matrices A et B ainsi que les vecteurs $\mathbf{p}$, $\mathbf{b}$ et $\mathbf{d}$.

   Remarque : on pourra coder cet exercice dans un seul fichier module python[4].

2. Écrire une fonction python pour construire les matrices A et B et les vecteurs $\mathbf{b}$ et $\mathbf{d}$ à partir des paramètres S, m et du tableau des tailles des fichiers :

$$\mathbf{t} = [t_1,\, t_2,\, \cdots, t_n]\,.$$

3. Écrire une autre fonction qui, à partir du vecteur solution $\mathbf{x}$ obtenu (ainsi que des paramètres n et m) permet de retrouver les numéros des fichiers stockés sur une clef donnée. Pour cela, on pourra extraire de la solution les $n \times m$ premières composantes puis les reformater en une matrice $n \times m$ grace à la fonction numpy `reshape`[5].
4. Écrire à la suite (ou dans un autre fichier) une partie script qui, à partir des données *jouets* suivantes :

$$n = 3,\ m = 4,\ S = 4,\ t = [1,\, 2,\, 3]$$

   permet de résoudre (PL). C'est à dire (i) calcul des matrices et vecteurs, (ii) appel à `linprog` et enfin (iii) calcul de la répartition des fichiers sur chaque clef.

   On pourra aussi utiliser la fonction `printkey` fournie dans le module `util_gro_tp1.py` pour afficher graphiquement la distribution des fichiers sur les clefs. Ces données jouets vous permettront de bien vérifier vos deux matrices.
5. Résoudre maintenant (PL) avec les données numériques : $n = 11$, $m = 4$ et $S = 8$.
   Les tailles des fichiers sont $\mathbf{t} = [1,\, 3,\, 2,\, 1,\, 2,\, 4,\, 2,\, 3,\, 5,\, 2,\, 2]$.
6. Faire d'autres essais numériques en augmentant le nombre de fichiers et en générant aléatoirement leur taille $\mathbf{t}$. Vous pouvez par exemple utiliser $S = 500$, $n = 50$, des tailles de fichiers obtenues par une loi uniforme sur $[\![40, 79]\!]$ (qu'on peut obtenir avec `np.floor(40*(np.random.rand(n)+1))`) et $m = 10$ clefs max.

   En général le solveur trouve rapidement une solution optimale mais certaines instances peuvent être longue à résoudre (glpk n'est pas un solveur très rapide...).
7. *Méthode heuristique.* Une alternative possible à (PL) est de ne pas chercher une solution optimale mais de se contenter d'une solution "proche". Une méthode consiste à stocker les fichiers par taille décroissante : on commence par stocker le plus gros fichier, puis le deuxième plus gros et ensuite de suite ... On rajoute une clef USB au fur et à mesure lorsqu'il n'y a plus de place sur les clefs déjà utilisées.

   Écrire une fonction python correspondant à cette méthode. Comparer avec la solution optimale trouvée précédemment (dans certains cas une solution optimale n'est pas obtenue).
8. Que faut-il changer dans la modélisation pour avoir deux sauvegardes (sur deux clés différentes !) de chaque fichier ?

4. Éventuellement vous pouvez coder les deux fonctions suivantes dans un autre fichier.
5. Attention il faut utiliser l'ordre par colonne avec l'option `order='F'`.