TELECOM Nancy 2A 2019-20 *Graphes et Recherche Op´erationnelle*

TD machine 1

Programmation lineaire ´

Le but de ce TD est d’utiliser le solveur glpk (`a partir du langage python) pour r´esoudre des probl`emes de programmation lin´eaire. Certains probl`emes propos´es ci-dessous ont ´et´e vus (et pour la plupart d’entre eux d´ej`a r´esolus...) dans les s´eances pr´ec´edentes de TD. On s’int´eresse dans ce TD `a des probl`emes de programmation lin´eaire de type probl`eme de production (par simplexe), et des probl`emes d’affectations vus comme des programmes lin´eaires en variables enti`eres ou binaires (affectation simple, probl`eme de bureau, probl`eme de plan de vols ...).

Le solveur glpk est utilis´e via un module python appel´e pymathprog 1. Cette interface de glpk est `a la fois souple et puissante mais un peu longue `a d´ecrire (et donc `a utiliser) pour un TP de 2H ! Pour faciliter les choses, une fonction linprog a ´et´e ´ecrite dont voici l’entˆete et une br`eve 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 r´esoudre 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 r´esoudre un PL pos´e sous la forme suivante :

minx ou maxx *f*(*x*) = c*>*

x

 

*A*x *≤* b

*A*ex = be

lb *≤* x *≤* ub

Pour linprog, le seul argument obligatoire est le vecteur *c*, tous les autres sont des arguments optionnels nomm´es (pouvant donc ˆetre mis dans n’importe quel ordre) :

— c doit ˆetre un tableau numarray 1-d, le nombre d’´el´ements de ce tableau d´efinit le nombre de variables *n* du PL.

— ineq= si il y a des contraintes d’in´egalit´e vous devez les entrer utilisant ineq=(A,b) c’est `a dire que l’argument effectif est un t-uple `a deux ´el´ements, le premier ´etant la matrice *A* (un tableau numarray 2-d, dont le nombre de colonnes doit ˆetre ´egal `a *n* et le nombre de lignes *m* d´efinit donc le nombre de contraintes d’in´egalit´e), et le deuxi`eme le vecteur *b* (un tableau numarray 1-d avec *m* ´el´ements).

— eq= Idem que pr´ec´edemment mais pour introduire les contraintes d”´egalit´e. — sens= une chaˆıne de caract`ere sens=’min’ pour minimiser (d´efaut) et sens=’max’ pour maximiser. — lb et ub permettent respectivement d’introduire des bornes sur la solution *x*. Ces deux arguments

doivent ˆetre des numarray 1-d `a *n* ´el´ements. Par d´efaut (si vous ne les d´efinissez pas) on a *lbi* = 0 et *ubi* = +*∞*, *∀i*.

— typevar= permet de d´efinir la nature des variables typevar=float pour des variables continues (d´efaut), typevar=int pour des variables enti`eres et typevar=bool pour des variables binaires.

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

1

— verb= niveau de bavardage du solveur 2; par d´efaut ce param`etre est mis `a 0 (aucune sortie) mais vous pouvez obtenir plus ou moins d’informations en le r´eglant `a 1, 2 ou 3.

— En sortie la fonction renvoie 3 arguments :

— stat un indicateur (5 lorsque l’optimum a ´et´e trouv´e, 6 lorsque *f* est non minor´ee/major´ee, 4 et 1 lorsque le domaine est vide),

— F la valeur de *f* `a l’optimum (ou None si elle n’est pas d´efinie),

— et X la (une) solution optimale (ou None si elle n’est pas d´efinie).

Un exemple d’utilisation de linprog

Prenons comme exemple l’exercice 1 du TD1 :

max*x∈*R2 *f*(*x*) = 1700*x*1 + 3200*x*2

*x*1*, x*2 *≥* 0

3*x*2 *≤* 39

1*.*5*x*1 + 4*x*2 *≤* 60

2*x*1 + 3*x*2 *≤* 57

3*x*1 *≤* 57

Voici un script possible pour r´esoudre ce probl`eme 3:

# -\*- coding: utf-8 -\*-

from util\_gro\_tp1 import linprog, affiche\_resultat

import numpy as np

# on d´efinit le vecteur c de la fct co^ut

c = np.array([1700, 3200])

# on d´efinit la matrice A et le vecteur b des contraintes d’in´egalit´e

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 probl`eme les in´egalit´es 1 et 4 sont de simples bornes sup´erieures 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 d´efinit le vecteur c de la fct co^ut

c = np.array([1700, 3200])

# on d´efinit la matrice A et le vecteur b des contraintes d’in´egalit´e

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 `a ˆetre enti`eres il faudrait rajouter l’argument typevar=int.

2. Fonctionne si vous utilisez python3 `a partir d’un terminal ; si vous utilisez spider3 lancer le `a 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 utilis´ee dans l’exemple.

2

Exercice 1. *Programmation lin´eaire*

R´esoudre les probl`emes suivants (cf. feuille TD3) en utilisant la fonction linprog pr´ec´edente. (a) Probl`eme de production (b) optimum unique `a base d´eg´en´er´ee





max [*F* = 50*x*1 + 40*x*2 + 70*x*3 + 80*x*4]

2*x*1 + 4*x*2 + 8*x*3 + 6*x*4 *≤* 100

10*x*1 + 8*x*2 + 6*x*3 + 10*x*4 *≤* 160





*x*1 + *x*2 + 2*x*3 + 2*x*4 *≤* 20

*x*1*, x*2*, x*3*, x*4 *≥* 0

max [*F* = *x*1 + *x*2] 3*x*1 + 2*x*2 *≤* 15 3*x*1 + 4*x*2 *≤* 21 *x*2 *≤* 3

*x*1*, x*2 *≥* 0

(c) optimum infini (d) ensemble des solutions r´ealisables vide

 

max [*F* = *x*1 + 3*x*2] *x*1 + *x*2 *≥* 3

*x*1 *−* 2*x*2 *≥* 5

*−*2*x*1 + *x*2 *≤* 5

*x*1*, x*2 *≥* 0

 

max [*F* = *x*1 *− x*2] *x*1 + 2*x*2 *≤* 5

2*x*1 + *x*2 *≤* 6

*x*1 + *x*2 *≥* 4

*x*1*, x*2 *≥* 0

Exercice 2. *Programmation lin´eaire en variables {*0*,* 1*} – Probl`eme de sac `a dos*. R´esoudre chacun des probl`emes de ”sac-`a-dos” suivants en utilisant la fonction linprog pr´ec´edente (ne pas oublier de mettre typevar=bool).

(*P*1)



max [*F*(*x*) = 6*x*1 + 9*x*2 + 8*x*3 + 4*x*4] 4*x*1 + 6*x*2 + 5*x*3 + 4*x*4 *≤* 8

3*x*1 + 6*x*2 + 4*x*3 + 3*x*4 *≤* 7



*x*1*, x*2*, x*3*, x*4 *∈ {*0*,* 1*}*

(*P*2)



max [*F*(*x*) = 6*x*1 *−* 9*x*2 + 8*x*3 + 4*x*4] 4*x*1 *−* 6*x*2 + 5*x*3 *−* 4*x*4 *≤* 2

3*x*1 *−* 6*x*2 *−* 4*x*3 + 3*x*4 *≤* 1



*x*1*, x*2*, x*3*, x*4 *∈ {*0*,* 1*}*

Exercice 3. *Stockage*.

On dispose de *m* clefs USB sur lesquelles on veut stocker *n* fichiers de taille *t*1*, · · · , tn*. Les clefs USB ont toutes la mˆeme capacit´e de stockage *S* (en GigaOctets). Tous les fichiers doivent ˆetre stock´es. Pour qu’un stockage soit admissible il faut que la somme des tailles des fichiers stock´es sur une clef soit inf´erieure ou ´egale `a *S* (on ne peut pas d´epasser la capacit´e d’une clef...). De plus, un fichier n’est stock´e qu’une seule fois sur une seule clef.

D´eterminer un stockage consiste alors `a d´eterminer sur quelle clef *j* est stock´e le fichier *i* donn´e. On cherche `a d´eterminer le stockage (admissible) des fichiers sur les clefs USB de fa¸con `a minimiser le nombre de clefs utilis´ees.

*Mod´elisation* (Rappel)

On introduit les variables suivantes :

— les variables binaires *yj* pour indiquer si la clef USB *j* est utilis´ee (*yj* = 1) ou non (*yj* = 0). — les variables binaires *xij* pour indiquer si le fichier *i* est stock´e sur la clef *j* (*xij* = 1) ou non (*xij* = 0).

Le probl`eme du stockage se mod´elise alors par le programme lin´eaire suivant (cf. TD1) :



minX*m*

*yj*

(*P*)

X*n i*=1

*j*=1

*tixij ≤ Syj , ∀j* = 1*, · · · , m*



X*m j*=1

*xij* = 1*, ∀i* = 1*, · · · , n*

*yj ∈ {*0*,* 1*}, ∀j* = 1*, · · · , m xij ∈ {*0*,* 1*}, ∀i* = 1*, · · · , n*

3

Travail demand´e.

Les variables *xij* et *yj* doivent ˆetre regroup´ees dans un seul vecteur x de (*n*+1)*m* composantes. On choisit un regroupement *colonne par colonne* :

x = (*x*11*, x*21*, · · · , xn*1 *| x*12*, x*22*, · · · , xn*2 *| · · · | x*1*m, · · · , xnm | y*1*, · · · , ym*)*>*

1. Ecrire le probl`eme ( ´ *P*) sous la forme suivante

(*P L*)

 

x*F*(x) = p*>*x

min

*A*x *≤* b

*B*x = d

x *∈ {*0*,* 1*}*

en explicitant les matrices *A* et *B* ainsi que les vecteurs p, b et d.

*Remarque : on pourra coder cet exercice dans un seul fichier module python* 4*.* 2. Ecrire une fonction python pour construire les matrices ´ *A* et *B* et les vecteurs b et d `a partir des param`etres *S*, *m* et du tableau des tailles des fichiers :

t = [*t*1*, t*2*, · · · , tn*] *.*

3. Ecrire une autre fonction qui, `a partir du vecteur solution ´ x obtenu (ainsi que des param`etres *n* et *m*) permet de retrouver les num´eros des fichiers stock´es sur une clef donn´ee. Pour cela, on pourra extraire de la solution les *n × m* premi`eres composantes puis les reformater en une matrice *n × m* grace `a la fonction numpy reshape 5.

4. Ecrire `a la suite (ou dans un autre fichier) une partie script qui, `a partir des donn´ees ´ *jouets* suivantes :

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

permet de r´esoudre (*P L*). C’est `a dire (*i*) calcul des matrices et vecteurs, (*ii*) appel `a linprog et enfin (*iii*) calcul de la r´epartition 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 donn´ees jouets vous permettront de bien v´erifier vos deux matrices.

5. R´esoudre maintenant (*P L*) avec les donn´ees num´eriques : *n* = 11, *m* = 4 et *S* = 8. Les tailles des fichiers sont t = [1*,* 3*,* 2*,* 1*,* 2*,* 4*,* 2*,* 3*,* 5*,* 2*,* 2].

6. Faire d’autres essais num´eriques en augmentant le nombre de fichiers et en g´en´erant al´eatoirement leur taille t. Vous pouvez par exemple utiliser *S* = 500, *n* = 50, des tailles de fichiers obtenues par une loi uniforme sur J40*,* 79K (qu’on peut obtenir avec np.floor(40\*(np.random.rand(n)+1))) et *m* = 10 clefs max.

En g´en´eral le solveur trouve rapidement une solution optimale mais certaines instances peuvent ˆetre longue `a r´esoudre (glpk n’est pas un solveur tr`es rapide...).

7. *M´ethode heuristique*. Une alternative possible `a (*P L*) est de ne pas chercher une solution optimale mais de se contenter d’une solution ”proche”. Une m´ethode consiste `a stocker les fichiers par taille d´ecroissante : on commence par stocker le plus gros fichier, puis le deuxi`eme plus gros et ensuite de suite ... On rajoute une clef USB au fur et `a mesure lorsqu’il n’y a plus de place sur les clefs d´ej`a utilis´ees.

Ecrire une fonction python correspondant `a cette m´ethode. Comparer avec la solution optimale ´ trouv´ee pr´ec´edemment (dans certains cas une solution optimale n’est pas obtenue). 8. Que faut-il changer dans la mod´elisation pour avoir deux sauvegardes (sur deux cl´es diff´erentes !) de chaque fichier ?

4. Eventuellement 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’.

4