Rapport du projet de programmation linéaire

Maxence Ahlouche Maxime Arthaud Korantin Auguste
Martin Carton Thomas Forgione Thomas Wagner

21 octobre 2013

Table des matières

1 Présentation de l'équipe

Cette équipe a été menée par Maxence Ahlouche, assisté de son Responsable Qualité Thomas Wagner. Les autres membres de l'équipe sont Martin Carton, Thomas Forgione, Maxime Arthaud, et Korantin Auguste.

	TD1	TD2	TD3	TP1	TP2	TP3
Maxence Ahlouche (CPC)				Abs.		
Maxime Arthaud					Abs.	
Korantin Auguste						
Carton Martin						
Thomas Forgione						
Thomas Wagner (RQ)						

2 Problème du sac à dos

2.1 Résolution exacte

Nous avons implémenté un algorithme de programmation dynamique, qui permet de résoudre le problème du sac à dos. Toutefois, il fonctionne uniquement si les poids des objets sont des entiers.

Sa complexité en temps est en $O(nW)$ et celle en mémoire en $O(W)$, avec n le nombre d'objets et W le poids maximum du sac.

Nous l'avons testé [1] sur plusieurs instances du problème (cf table 1), et l'algorithme est rapide.

2.2 Résolution approchée

Nous avons aussi implémenté l'algorithme glouton : celui-ci consiste à choisir les « meilleurs » objets jusqu'à que la masse maximale soit dépassée. Le critère déterminant quels sont les meilleurs objets peut être la masse faible, le prix élevé, ou le rapport prix/masse élevé.

Cet algorithme est beaucoup plus rapide que le précédent, mais n'est qu'un algorithme approché. La table 1 montre quelques-uns des résultats obtenus.

1. En utilisant le générateur de problèmes trouvé à l'adresse suivante : `http://www.diku.dk/~pisinger/codes.html`.

Paramètres du générateur/ masse maximale autorisée	Résultat optimum	Prix le plus élevé	Masse la plus faible	Meilleur ratio prix/masse
50 25 1 1 1000/20	85	49/42.4%	67/21.2%	81/4.7%
500 25 1 1 1000/500	2016	1125/44.2%	1725/14.4%	1983/1.6%
5000 25 1 1 1000/500	5540	1175/79%	4577/17.4%	5540/0%
50000 25 1 1 1000/500	11195	1175/90%	6684/40.3%	11195/0%
50000 1000 1 1 1000/500	118260	5959/95%	101857/13.9%	118147/0.1%
50000 5000 1 1 1000/100	100847	14931/85.2%	93532/7.3%	100282/0.6%

TABLE 1 – Résultats de l'algorithme glouton

On remarque qu'en général, la résolution approchée en considérant le ratio prix/masse donne de très bon résultats, voire le meilleur résultat. Le résultat que fournit cet algorithme est le moins bon quand la masse maximale autorisée est faible comparée à l'amplitude des valeurs que peuvent prendre le prix et la masse des objets.

Toutefois cet algorithme est beaucoup plus rapide, et a une complexité en temps de $O(n \log n)$ (pour le tri des objets), et ne nécessite en mémoire que la liste des objets.

3 Problème d'optimisation linéaire

Le but est maximiser une fonction linéaire sous certaines contraintes (inéquations linéaires).

3.1 Existence de solutions et autre blablas mathématiques

3.1.1 Formalisation du problème

Considérons le problème suivant :

$$(P) \quad \max_{x \in C \subset \mathbb{R}^n} f(x)$$

Nous nous placerons dans le cas où f est linéaire, $x \geqslant 0$, et où C est décrit par des contraintes d'inégalités linéaires, c'est-à-dire qu'il existe une matrice A et un vecteur b tels que $Ax \leqslant b$.

3.1.2 Existence de solutions

Pour un tel problème, trois possibilités s'offrent à nous :
— les contraintes sont incompatibles ;
— la fonction est non majorée sur C ;
— le problème admet un maximum sur C.
Nous savons de plus que C est un polyèdre convexe. Un théorème garantit alors que si ce problème à une solution, alors il a une solution en un de ses sommets. Nous allons donc chercher les solutions parmi les sommets de C.

3.2 Algorithme du simplexe

Le principe de cet algorithme est de considérer un des sommets du polyèdre, puis de se déplacer en suivant les arêtes de ce polyèdre en augmentant à chaque itération le gain. L'algorithme se terminera lorsque nous nous trouverons sur un sommet, dont tous les sommets adjacents présentent un gain plus faible. La convexité du polyèdre nous garantit que le résultat est optimal.

L'algorithme du simplexe a une complexité dans le pire des cas exponentielle, mais en pratique, cet algorithme est efficace.

Cet algorithme ne permet pas de maximiser une fonction pour des variables entières (par exemple pour connaitre un nombre de produits à produire, donc un nombre entier) à produire pour maximiser un gain (bien qu'on pourrait en pratique l'utiliser en considérant que la solution optimale entière est suffisamment proche de la solution optimale réelle).

3.2.1 Forme standard et tableau canonique

Pour résoudre le problème, la première étape est le mettre sous forme standard. Pour cela on ajoute à chaque inéquation j de la forme $\sum a_{j,i}x_i \leq 0$ une variable dite d'écart pour la transformer en égalité : $\sum a_{j,i}x_i + s_j = 0$ où $s_j \geq 0$.

Les inéquations de la forme $\sum a_i x_i \geq 0$ sont d'abord multipliées par -1 avant cette étape.

On construit ensuite un tableau dit canonique représentant le problème comme suit :

— la première ligne de la matrice est $[m_0, m_1, \cdots, m_n, 0, \cdots, 0]$ où les (m_i) sont les coefficients du problème $\min \sum m_i x_i$ et auquel on ajoute autant de 0 qu'on a ajouté de variables d'écart.

— les autres lignes de la matrice sont $[a_{j,0}, \cdots, a_{j,n}, 0, \cdots, 0, 1, 0, \cdots, 0]$ où les 1 sont placés de manière à former une matrice identité (ils correspondent aux variables d'écart ajoutées).

3.2.2 Algorithme

```
Entrée : matrice (un tableau canonique)
Sortie : le résultat optimum
         les quantités de chaque produit à produire
         les quantités de chaque ressources restantes

base = indices des variables de base de la matrice

tant qu'il reste des nombres strictement positifs sur la première ligne:
    à_ajouter = indice de la colonne dont le premier élément est maximal
    à_retirer = indice de la ligne (>1) telle que :
                matrice[à_retirer, dernière colonne]/matrice[à_retirer, à_ajouter]
                est minimum

    remplacer le (à_retirer)ième élément de base par à_ajouter

    diviser la ligne à_retirer de matrice par matrice[à_retirer, à_ajouter]
    et soustraire aux autres lignes y le vecteur :
        matrice[y,a_ajouter] * matrice[a_retirer,] / matrice[a_retirer, a_ajouter]
```

```
pour chaque variable d'origine n (n dans [0, nombre de variables hors base[):
  à_produire[base[n]] = matrice[n+1, dernière colonne]

pour chaque variable d'écart n (n dans [nombre de variables hors base, nombre de
    varibles total[):
  restes[base[n]] = matrice[n+1, dernière colonne]

retourner (-matrice[0, dernière colonne], à_produire, restes)
```

3.2.3 Tests

3.2.4 Dégénérescence

Un problème du simplexe est dit dégénéré si plus de deux contraintes vont devoir être nulles en un sommet. Graphiquement (en 2D), cela veut dire qu'au moins 3 droites vont se rencontrer en un sommet.

Ceci va empêcher l'algorithme du simplexe de progresser entre deux itérations : il va simplement changer de base. Le problème étant que sur des cas particuliers, il pourra changer de base sans progresser, puis boucler à l'infini en faisant un cycle sur des bases qui n'améliorent pas la solution.

Pour éviter cela, on peut utiliser des règles d'anti-cyclage, dont la règle de Bland, qui consiste à choisir judicieusement les variables qu'on fera entrer et sortir de la base, dans le cas où il y aurait plusieurs possibilités autant intéressantes les unes que les autres. Elle consiste simplement à se baser sur l'indice des variables.

4 Annexe

Listings

Listing 1 – Codes relatifs au problème du sac à dos

```python
#!/usr/bin/python
# -*- coding: UTF-8 -*-

def knapsack(objects, max_mass):
    """
    Résoud le problème du sac à dos avec de la programmation dynamique.
    Fonctionne seulement avec des valeurs entières.

    objects est une liste de couple (masse, prix) représentant les objets.

    >>> objects = ((2,3),(3,4),(4,5),(5,6))
    >>> knapsack(objects, 5)
    7
    """
    assert isinstance(max_mass, int) and all(isinstance(x[0], int) for x in
        objects)
```

```
    current_line = [0 for i in range(max_mass+1)]
    prev_line = current_line[:]

    for i in range(0,len(objects)):
        object_mass, price = objects[i]
        for masse in range(max_mass + 1):
            if object_mass <= masse:
                current_line[masse] = max(prev_line[masse],
                    prev_line[masse-object_mass] + price)
            else:
                current_line[masse] = prev_line[masse]

        prev_line = current_line[:]

    return current_line[max_mass]

def best_ratio(x): return x[1]/x[0]
def less_mass(x):  return -x[0]
def best_price(x): return x[1]

def greedy(objects, max_mass, key):
    """
        Algorithme approché du glouton.
        Nécessite de trier les objects selon un critère `key`.
        Par exemple
            greedy(obj, max_mass, less_mass)
        choisit les objects en commençant par les moins lourds.
    """
    cost, mass = 0, 0
    objects = sorted(objects, key=key, reverse=True)

    for o in objects:
        if o[0] + mass <= max_mass:
            mass += o[0]
            cost += o[1]

            if mass == max_mass:
                break

    return cost

def read_testfile(path):
    """
        Lit un fichier généré par le générateur trouvé ici:
            http://www.diku.dk/~pisinger/codes.html
        Retourne une liste de couples (masse, valeur) considérée comme bon
        exemple.
    """
    with open(path, 'r') as f:
        objects = []
        line = f.readline()
        nb_objs = int(line)
        for i in range(0, nb_objs):
            line = f.readline()
            dummy, a, b = map(int, line.split())
            objects.append((b, a))
        return objects
```

Listing 2 – Tests du sac à dos

```
#!/usr/bin/python2
```

```python
# -*- coding: utf-8 -*-

import datetime
import knapsack as sad

def test_sad(obj, file_name, max_mass):
    start = datetime.datetime.now()
    best = sad.knapsack(obj, max_mass)
    exec_time = datetime.datetime.now() - start
    print("Tested %s with max_mass=%s in %s" % (file_name, max_mass, exec_time))
    print("    optimum:    %s" % best)
    r = sad.greedy(obj, max_mass, sad.best_price)
    print("    best price: %s / %.1f%%" % (r, 100.*(best-r)/best))
    r = sad.greedy(obj, max_mass, sad.less_mass)
    print("    less mass:  %s / %.1f%%" % (r, 100.*(best-r)/best))
    r = sad.greedy(obj, max_mass, sad.best_ratio)
    print("    best ratio: %s / %.1f%%" % (r, 100.*(best-r)/best))

if __name__ == '__main__':
    sad_files = [
            "tests/50,25,1,1,1000.in",
            "tests/500,25,1,1,1000.in",
            "tests/5000,25,1,1,1000.in",
            "tests/5000,300,1,1,1000.in",
            "tests/50000,25,1,1,1000.in",
            "tests/50000,300,1,1,1000.in",
            "tests/50000,1000,1,1,1000.in",
            "tests/50000,5000,1,1,1000.in"
    ]

    for p in sad_files:
        obj = sad.read_testfile(p)
        test_sad(obj[:], p, 20)
        test_sad(obj[:], p, 100)
        test_sad(obj[:], p, 500)
        print("")
```

Listing 3 – Codes relatifs au simplexe

```python
#!/usr/bin/python2
# -*- coding: utf-8 -*-
from __future__ import division
import numpy as np

"""
Fonction utilisée par la fonction simplexe, elle ne devrait pas être appelée
directement.
La matrice d'entrée doit avoir la forme suivante :
    Les premières colonnes correspondent aux produits :
    [bénéfice du produit, cout en ressource 1, cout en ressource 2, ...]
    (ce sont les variables libres)

    Les dernières colonnes correspondent aux variables de base :
    [0, ..., 0, 1, 0, ..., 0]
    sachant que le carré des variables de base forme une matrice identité.

    Et sur la toute dernière colonne :
    [0, stock en ressource 1, stock en ressource 2, ...]

Attention: la matrice doit être une matrice flottante pour numpy !
Càd déclarée avec np.array([...], dtype='f')
```

```
Retourne un triplet de la forme (gain, [a_produire...], [restes...]) où :
    - a_produire[n] est la quantité de produit n à produire;
    - restes[m]     est ce qu'il reste en ressource m.
"""

def simplexe_aux(matrice):
    size_y, size_x = matrice.shape

    # variables de base
    base = list(range(size_x-1-(size_y-1), size_x-1))

    # indice de colonne (pour le x à ajouter de la base)
    a_ajouter = np.argmax(matrice[0,])
    while matrice[0,a_ajouter] > 0:
        # indice de ligne (pour le x à retirer de la base)
        a_retirer = None
        meilleur_ratio = 0

        # la première ligne est pour z, on n'y cherche pas le ratio
        for y in range(1, size_y):
            if matrice[y,a_ajouter] == 0: #todo: comp float ?
                continue
            ratio = matrice[y,-1] / matrice[y,a_ajouter]
            if a_retirer is None or ratio < meilleur_ratio:
                a_retirer = y
                meilleur_ratio = ratio

        base[a_retirer-1] = a_ajouter

        # opérations sur les lignes
        for y in range(size_y):
            if y == a_retirer:
                matrice[y,] /= matrice[y,a_ajouter]
            else:
                ratio = matrice[y,a_ajouter] / matrice[a_retirer, a_ajouter]
                matrice[y,] -= ratio * matrice[a_retirer,]

        a_ajouter = np.argmax(matrice[0,])

    # recherche des quantités à produire (au début de la liste)
    # et des restes (à la fin)
    a_produire_et_restes = [0]*(size_x-1)
    for n in range(len(base)):
        a_produire_et_restes[base[n]] = matrice[n+1,-1]

    return -matrice[0,-1],                              \
           a_produire_et_restes[0:size_x-size_y],       \
           a_produire_et_restes[size_x-size_y:-1]


def simplexe(contraintes, profit):
    """
    Cette fonction prend en entrée la matrice contenant les
    inéquations définissant le problème, telle que définie dans le
    slide 3 des séances 2 et 3. Elle prend également la fonction
    profit, les variables devant être dans le même ordre que dans
    l'autre matrice.
    Elle transforme cette matrice en une matrice utilisable par la
    fonction simplexe_aux.
    """

    nb_mat, nb_pdt = contraintes.shape
```

```
    nb_pdt -= 1 # Contraintes contient également les stocks

    m = np.array([[0] * (nb_pdt + nb_mat + 1)] * (nb_mat + 1), dtype='f')

    # Ajout du profit
    m[0,] = profit + [0] * (nb_mat + 1)

    for i in range(nb_mat):
        # Ajout des contraintes sur les variables libres
        m[i+1,:nb_pdt] = contraintes[i,:nb_pdt]
        # Ajout des variables de base
        m[i+1,nb_pdt+i] = 1

    # Ajout des stocks
    m[1:,-1] = contraintes[:,-1]

    return simplexe_aux(m)

def recherche_initial(matrice):
    '''
    Retourne un sommet initial, ou None s'il n'y a pas de solution
    '''
    size_y, size_x = matrice.shape
    prob_artificiel = np.zeros((size_y, size_x + (size_y - 1)))
    prob_artificiel[1:, 0:size_x-1] = matrice[1:, 0:size_x-1]
    prob_artificiel[:, -1] = matrice[:, -1]
    prob_artificiel[0, size_x-1:-1] = 1.0
    prob_artificiel[1:, size_x-1:-1] = np.identity(size_y - 1)

    solution, benef_max = simplexe(prob_artificiel)
    if benef_max != 0:
        return None
    else:
        return solution[:size_x-1]

if __name__ == '__main__':
    np.set_printoptions(precision=2, suppress=True)

    m = np.array([
        [7, 9, 18, 17, 0, 0, 0, 0],
        [2, 4, 5, 7,   1, 0, 0, 42],
        [1, 1, 2, 2,   0, 1, 0, 17],
        [1, 2, 3, 3,   0, 0, 1, 24]
    ], dtype='f')
    direct = simplexe_aux(m)
    print(direct)

    contraintes = np.array([[2,4,5,7,42],[1,1,2,2,17],[1,2,3,3,24]])
    profit = [7,9,18,17]
    assert(direct == simplexe(contraintes, profit))

    print("==========")

    m = np.array([
      [5, 8, 0, 0, 0],
      [5, 2, 1, 0, 42],
      [4, 7, 0, 1, 26]], dtype='f')
    print(simplexe_aux(m))
```