22 MAI 2019

MINI PROJET

SAC A DOS MULTIDIMENSIONNEL

ALI ABAKAR – ADEDIRAN FLORE - OUSMANE BACHIR – AW AMADOU – YVES FERNAND

UPHF Valencienne / Master CDSI

Introduction :

Le but de ce travail est la proposition d'un algorithme de résolution d'un problème de sac à dos multidimensionnel

1. Présentation du problème et format des données :
   Il modélise une situation analogue au remplissage d'un sac à dos de taille fixe :
   -poids du sac fixé
   -ensemble d'objets donné
   -chacun a un poids et une valeur
   Les objets mis dans le sac à dos doivent maximiser la valeur totale, sans dépasser le poids maximum du sac.
   Dans le cas du sac à dos multidimensionnel de dimension m, à chaque objet est associé m poids. Nous avons aussi M contraintes.

$$(\text{SADM}) \begin{cases} \max & \sum_{j=1}^{n} c_j x_j \\ \text{s.c. :} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad i \in M = \{1, \ldots, m\} \\ & x_j \in \{0,1\} \qquad j \in N = \{1, \ldots, n\} \end{cases}$$

   Ainsi dans notre implémentation nous avons :
   - Un tableau à une dimension de taille n pour les valeurs que nous utilisons comme ceci : c[n]
   - Un tableau à une dimension de taille n pour les objets à sélectionner x[n], 0 si non sélectionné et un 1 si oui.
   - Un tableau à une dimension de taille m, constraints[m] contenants les poids maximums des m dimensions
   - Un tableau à deux dimension de m lignes et n colonnes contenant sur chaque ligne les n poids associés à la m colonne que nous appelons weight[m][n]

   En code python le problème s'écrit de la manière suivante :

```
for j in range(m):
    if sum( x*weight[j] ) >constraints[j]:
        return 0

return 1
```

   Si la somme du produit des objets sélectionnés et des poids correspondant est supérieure à la contrainte la fonction retourne 0, sinon si pour tous les sacs la contrainte n'est pas violée alors le problème du SADM est respecté.

   Nous utilisons des tableaux Numpy, une bibliothèque compilée en C qui accélère grandement le parcours des données en python.

Pour toutes la fonction nous utilisons des valeurs globales pour les valeurs de poids valeurs et contraintes puisque ce sont des données statiques, nous gagnons en performance à ne pas les copier à chaque appel de fonction.

2. Méthodes de résolution

Il existe deux grandes catégories de méthodes de résolution de problèmes d'optimisation combinatoire : les méthodes exactes et les méthodes approchées. Les méthodes approchées, encore appelées heuristiques, permettent d'obtenir rapidement une solution approchée, donc pas nécessairement optimale. Nous allons utiliser l'algorithme tabou que nous adapterons à notre façon.

3. Implémentation :

3.1. Solution de départ :

Nous avons plusieurs possibilités de solution initiale

- Une solution non optimale par algorithme de glouton rapide : consistant à réaliser une solution faisable minimale : cette méthode consiste à prendre les i premiers élément tant que toutes les contraintes sont satisfaites et à s'arrêter sinon. Cette voie vite été abandonnée pour mauvais résultats.
- Une solution vide : Prend trop de temps.
- Un sac plein : très bon résultats, cependant il faut une phase destructive plus longue en début de recherche.
- Une solution de départ aléatoire : Fourni de très bon résultat et permet de garder une certaine constance malgré le changement d'instance. Cependant la fonction la fonction random que nous utilisons est plus généreuse en « 1 ». Ce qui nous amène souvent initialement en zones de solution infaisable.

```
x = np.random.randint(2,size=n)
```

X est initialement un tableau de 0 et de 1 aléatoire de taille n.

3.2. Contraintes de substitution et meilleur variable à changer :

Afin de réduire le temps de calcul pour trouver quelle variable mettre ou enlever du sac, nous utilisons des opérations mathématiques pour trouver une contrainte satisfaisant plus ou moins légèrement toutes les m contraintes

- Tout d'abord nous calculons de nouvelles contraintes b_i résultant de la difference entre la valeur de la contrainte et la somme des poids des objets sélectionnés.
  Cela nous permet de savoir si le sac est plein pour une certaine contrainte.

$$b_i^{'} = b_i - \sum_{j=1}^{n} (a_{ij} : \text{for j with } x_j = 1)$$

En code cela donne :

```
for i in range(m):
    b[i] = constraints[i] - np.sum(weight[i]*x)
```

Ensuite selon ces nouvelles nous calculons les coefficients suivants :

$$\text{If } b_i^{'} > 0, \text{ set } w_i = 1/b_i^{'}$$
$$\text{If } b_i^{'} \leq 0, \text{ set } w_i = 2 + |b_i^{'}|$$

```
if b[i] > 0 :
    w[i] = (1/b[i])
else:
    w[i] = (2 + abs(b[i]))
```

Rappelons que si b est positif la capacité est supérieur à la somme des poids actifs et inversement.

Et nous calculons les nouveaux coefficients de la façon suivante :

$$s_j = \sum_{i=1}^{m} w_i a_{ij}$$

$$s_0 = \sum_{i=1}^{m} w_i b_i$$

Où Sj sont les nouveaux coefficients, et S_0 la nouvelle contrainte telle que :

$$\sum_{j=1}^{n} s_j x_j \leq s_0$$

En python cela nous donne :

```
#ici on recalcule les coefficients des poids
for i in range(m):
    s[i] = w[i]*weight[i]
    new_constraints[i]= w[i] * constraints[i]

#Calcul des d'une contrainte de substitution unique pour les m sac
s = np.sum(s,axis=0)
s0 = sum(new_constraints)
#la fonction retourne le rapport valeur et poids des objets dans un tableau pour evaluation du mouvement.
return c/s
```

La valeurs c/s nous permettra de choisir quels coefficients basculer à 0 ou 1.

3.2.1. Mettre ou enlever un élément du sac :

Dans notre implémentation nous avons 2 phases :

- Une phase destructive ou nous enlevons des éléments du sac :

L'élément à enlever est choisi de la manière suivante :

$$\min(c_j / s_j : x_j = 1)$$

Rappelons que Sj est grand pour les contraintes non respectées.

```
def remove(values_over_weight,tabu_list)    :

    global x
    i=0
    #tri croissant du rapport valeur/poids
    temp = values_over_weight.argsort()
    temp = list(temp)
    for tabu in tabu_list:
        if tabu in temp:
            temp.remove(tabu)
    #ne pa choisir l'element s'il est deja 0
    while x[temp[i]] == 0:
        i+=1
    #la boucle s'est arrete ,  le plus petit element a 1 est mis a 0
    x[temp[i]] = 0
    return temp[i]
```

Values_over_weight est ici est ici le tableau contenant les éléments du rapport c/j
Nous faisons un tri des éléments, nous retirons les éléments présents dans la liste taboue.

- Une phase constructive où nous mettons les éléments dans le sac.

$$\max(c_j / s_j : x_j = 0)$$

Une procédure similaire mais inverse y est appliquée.

3.3. <u>Liste taboue :</u>

Pour cette liste nous partons d'un raisonnement simplissime, la taille de la liste est statique

Cette liste est entièrement vidée après k itérations, après plusieurs test 7 est une bonne valeur pour k. Nous n'implémentons pas ici des mécanismes de pénalité, ou de fréquence qui sont gourmand en ressources et surtout n'apporte des résultats sensiblement meilleurs. Le rapport temps de calcul et performance nous ont fait abandonner ces mécanismes. Par contre nous utilisons 2 listes taboues, l'une pour la phase destructive et l'autre pour la constructive.

```
if iteration%7==0:
    c_tabu_list=[]
    d_tabu_list=[]
```

A noter que ce sont les éléments issus de la phase constructive qui constituent les éléments de la liste tabou pour la phase destructive : « ne détruis pas ce qui vient d'être construit récemment ». Et inversement.

3.4. Evénement critiques :

Nous définissons un évènement critique comme étant le point de passage d'une solution faisable vers une autre non faisable et vice-versa. Quand nous passons d'un évènement faisable au non faisable nous enclenchons la procédure suivante : constructive_critical_proc

```
def critical_constructive_proc():
    global c,x,latest_move
    x[latest_move] = 0
    sol = find_best_solution(1)
    if sol!=-1:
        x[sol] = 1

    x[latest_move] = 1
    sol = find_best_solution2(1)
    if sol!=10000:
            x[sol] = 0
```

- Nous commençons par retirer l'élément coupable de l'évènement critique : nous revenons alors au non faisable.
- Nous cherchons l'élément ayant la plus grande valeur c étant possible sans dépasser la capacité réelle du sac.
- Si un tel élément n'a pas été trouvé alors notre fonction find_best_solution retourne -1
  Sinon cet élément est mis dans le sac. Nous revenons à un point critique.
- Nous remettons l'élément coupable, nous sommes alors dans la zone infaisable.
- Puis nous cherchons l'élément le plus inutile présent dans le sac à dos : une petite valeur et un poids suffisamment élevé pour nous ramener dans la région faisable.
  Si un tel élément est trouvé nous l'enlevons du sac, sinon nous continuons vers la phase destructive.

La meme procédure mais inverse serait appliquée pour là pour les évènement critique destructive, cependant nous avons vite fait d'annuler l'utilisation cette procédure, qui est couteuse en ressource et décroit grandement nos résultats.

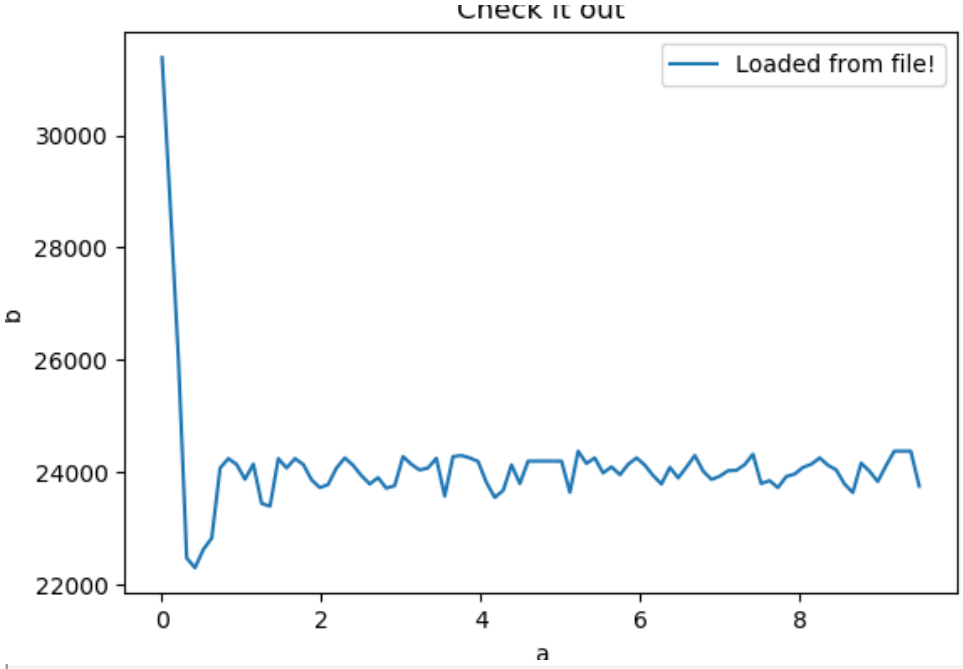
3.5. Synthèse de l'algorithme utilisé :

- Nous commençons par une phrase destructive qui consiste à enlever 5 éléments du sac.

En effet nos solutions de départ sont largement au-dessus de la capacité du sac. Cette phase est répétée tant que nous ne sommes pas dans les zones faisables. Le dernier élément enlevé est mis dans la liste taboue pour la construction.

- Quand nous somme dans les zones dans les zones faisables nous mettons des éléments dans le sac tant que cela est faisable. Le dernier élément est mis dans la liste taboue pour la destruction.
- Nous appelons la procédure de gestion de d'évènement critique qui dans notre implémentation est appelée dans tous les cas.
- La liste taboue est vidée chaque 7 itérations.
  Une utilisation de l'intensification ici serait de réduire la taille de la liste taboue, et diminuer le nombre d'itérations de la phase destructive.

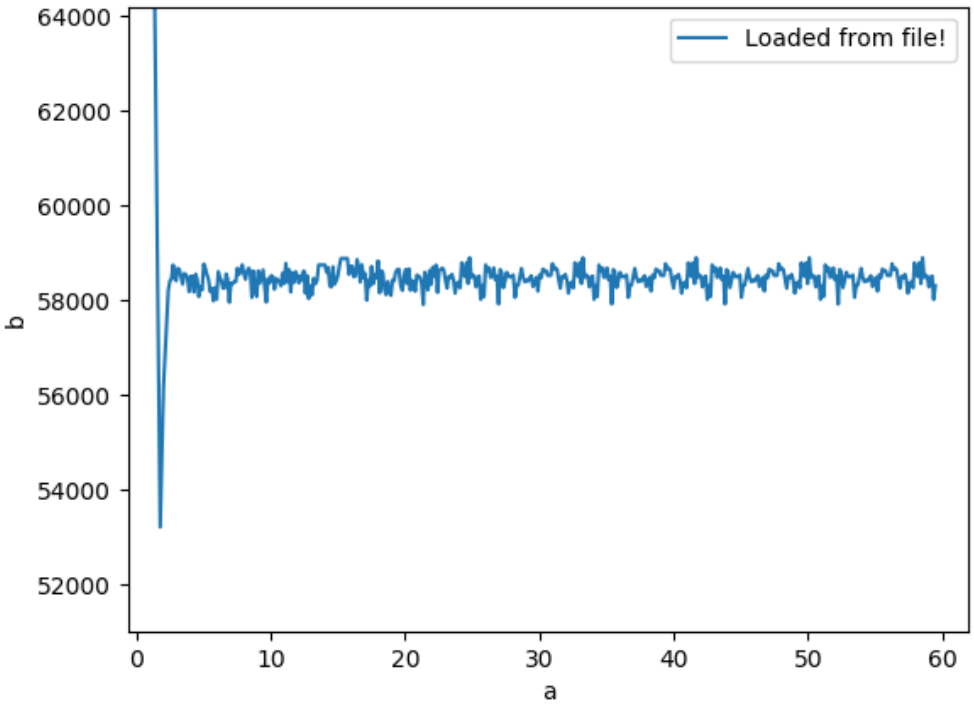
4. Résultats :




```
115516 129
0 2 8 10 11 18 20 22 23 30 31 32 35 36 39 42 45 47 51 53 55 63 64 65 69 79 92 100 105 106 113 114 115 116 121 131
```

Ici nous avons une courbe qui a en abscisse le temps, et en ordonnées la somme des éléments mis dans le sac pour l'instance 100M5_1.dat.
Après plusieurs essais nous décidons d'enlever la procédure d'évènement critique pour la phase destructive ; une taille de liste tabou fixé à 7.

Pour 250M30_1.dat



Avec une valeur de 59125 comme meilleur solution.

Et pour 500M30_1

```
C:\Downloads\sacado>python sacado.py 500M30_1.dat stesssss 30
115516 29.675578594207764
```

Nous avons 115516 comme meilleur résultat.

Conclusion :

Au terme de ce travail, nous avons pu maitriser le problème du sac à dos. Nous avons proposé notre implémentation de l'algorithme tabou. D'après le résultat nous avons gardé la meilleure variante de notre algorithme qui nous donne des résultats satisfaisant dans un temps réduit meme avec un language de haut niveau comme python.

Référence :

A tabu search based heuristic for the 0/1 Multiconstrained Knapsack Problem
Johan Oppen Solveig
Irene Grüner
Arne Løkketangen