جامعة مولاي إسماعيل
UNIVERSITÉ MOULAY ISMAÏL

المدرسة الوطنية العليا للفنون و المهن
ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS

Projet d'expertise

Automatisation et optimisation d'un processus de distribution des produits aux machines par la recherche opérationnelle

Réalisé par :
NASR EDDINE Sara
BOUNEDDI Mehdi

Encadré par :
Mr MASROUR Tawfik

Année universitaire : 2018/2019

Remerciements

Au terme de ce travail, nous exprimons nos sincères remerciements à notre école pour l'excellente formation qu'elle nous donne, ainsi que pour les moyens techniques qu'elle met à notre disposition.

Nous tenons à présenter nos vifs remerciements et notre profonde gratitude à notre cher professeur et encadrant Mr MASROUR Tawfik pour ses innombrables conseils et orientations, pour le partage de son expérience et connaissances, nous le remercions chaleureusement car il nous a prêté main forte pour accomplir notre tâche dans les bonnes conditions.

Par la même occasion, nous tenons à remercier notre chère professeur Mme EL HASSANI Ibtissam qui nous a soutenu, épaulé et présenté son aide durant toute la période de notre projet. Ce travail n'aurait pas pu être réalisé dans les contraintes de temps et de qualité recherchés, sans votre précieuse aide et réactivité.

Nous remercions aussi tous les membres de l'atelier de l'intelligence artificielle qui étaient tout le temps à notre écoute et portaient beaucoup d'intérêts à notre travail.

Trouvez dans ce modeste travail toute notre gratitude et respect, et merci à vous tous.

Résumé

Notre projet d'expertise est la suite d'un projet de fin d'études réalisé à Yazaki Meknès (PFE 2018). Il consiste en l'automatisation et l'optimisation de l'opération de distribution des produits aux machines. Pour ce faire, nous avons eu recours à la recherche opérationnelle, nous avons donc compris le concept de single Knapsack problems, puis nous avons exécuté quelques méthodes de résolution par programmation dynamique et par heuristiques sur le langage python, ensuite nous avons modélisé le processus de distribution des produits aux machines (c'est-à-dire modéliser le problème industriel), et finalement nous avons proposé une heuristique pour résoudre ce problème de sac à dos multiples (MKP) et ainsi l'implémenter sur python.

Table des matières

Table des matières

Introduction Générale

Dans le cadre de la concurrence existant entre les différentes sociétés de câblage au Maroc, Yazaki cherche à augmenter de plus en plus sa productivité en optimisant ses différents processus de production; c'est dans cette perspective que Yazaki Meknès se voit obligée d'automatiser l'affectation d'un nombre important d'articles semi-finis aux machines de coupes.

La distribution de ces produits sur les machines de coupe consiste à connaitre le nombre d'articles à mettre dans chaque ensemble de machines, cela doit se faire dans la manière la plus optimale. Cette opération qui se fait manuellement nécessite alors une sorte d'automatisation et d'optimisation, pour diminuer le temps énorme qu'elle prend ainsi qu'augmenter le volume de production.

Afin de résoudre notre problème par recherche opérationnelle et ainsi optimiser ces opérations, une modélisation de ce dernier s'avère indispensable. Dans notre cas, nous nous sommes basés sur les résultats du travail effectué par Mme TAQUI Wissal durant son stage de fin d'études, et qui l'ont menée à considérer cette distribution similaire à un problème de sac-à-dos multiple (Multiple Knapsack Problem 'MKP').

Dans une première partie nous allons commencer par faire une introduction sur le problème de knapsack dans ses deux formes (single et multiple) ainsi que les méthodes de résolutions (que ça soit par programmation dynamique ou par heuristique) ; on passera ensuite à une deuxième partie où l'on va expliquer notre modélisation du problème ainsi que notre heuristique pour l'obtention d'une solution faisable en un temps précis ; et on terminera enfin par une présentation du problème d'affectation ainsi que notre modélisation avec des perspectives de résolution et une conclusion générale.

En parallèle avec ce plan, nous allons réaliser pour chaque exemple traité un essai de programmation sur Python accompagné d'une introduction sur les bibliothèques utilisées ainsi qu'une explication détaillée des scriptes employés.

Chapitre 1 : Problème de sac-à-dos

Définition :

Le problème du sac à dos, noté également KP (Knapsack problem) est un problème d'optimisation combinatoire. Il modélise une situation analogue au remplissage d'un ou plusieurs sac à dos, ne pouvant supporter plus d'un certain poids, avec tout ou partie d'un ensemble donné d'objets ayant chacun un poids et une valeur. Les objets mis dans le sac à dos doivent maximiser la valeur totale, sans dépasser sa capacité maximale.

I. Single knapsack (KP)

1. Concept

Le problème du sac à dos se présente sous la forme mathématique suivante :

$$(KP)\begin{cases} \max \sum_{i=1}^{n} p_i .. x_i , \\ \sum_{i=1}^{n} w_i .. x_i \leq c, \\ x_i \in \{0,1\}, i \in \{1,..,n\} \end{cases}$$

Etant donné pour chacun des n objets X_i un poids W_i et un profit P_i , le but est de remplir le sac de capacité C de telle sorte que la somme des profits des objets choisis soit maximisée et à condition que la somme de leurs poids ne dépasse pas C.

X_i est une variable binaire : si l'objet i est mis dans le sac, la valeur de X_i prend 1 ; dans l'autre cas, elle prend 0.

2. Le cas fractionnaire

Dans un premier cas, on admettra que les objets à mettre dans le sac peuvent être pris partiellement, c'est-à-dire qu'on a le droit de diviser un objet en parties, en 'fractions'.

Ce cas est une relaxation du problème original, puisqu'il nous aide à trouver le résultat le plus optimal dans des conditions idéales : en suivant la GREEDY METHOD, ou bien la méthode gourmande, qui, comme son nom l'indique, consiste à remplir tout le sac jusqu'à atteindre sa capacité maximale. Pour ce faire, il faut :

-Calculer le ratio $r_i = \frac{Pi}{Wi}$,

-Classer ces ratios selon un ordre décroissant,

-Calculer la somme $\sum Pi.Xi$ en suivant l'ordre des ratios jusqu'à atteindre le cas $\sum Wi.Xi > C$ puis supprimer le dernier objet mis et le remplacer par sa fraction pour avoir $\sum Wi.Xi = C$, la somme des profits équivalente représente le maximum qu'on peut espérer.

3. Le cas 0/1 (programmation dynamique+ heuristique)

Dans ce cas, les objets doivent être pris entièrement, c'est-à-dire que si on dépasse la capacité, l'objet responsable du dépassement est automatiquement rejeté ou remplacé si c'est possible par un autre.

Afin de résoudre ce problème, on procède ou bien par la programmation dynamique qui nous donne le résultat le plus optimal même si elle prend beaucoup de temps de calcul dans le cas où l'on a un nombre important d'objets, ou bien en suivant une heuristique qui nous donne une solution faisable mais pas forcément optimale e cherche par la suite à l'améliorer.

La programmation dynamique :

Définition :

La programmation dynamique est une méthode algorithmique pour résoudre des problèmes d'optimisation.

Donc le principe est de créer un algorithme à suivre dans la résolution du problème.

Voici un exemple de programmation dynamique :

Soit le tableau suivant :

Objet	1	2	3	4
Poids W_i	2	3	4	5
Profit P_i	1	2	5	6

Le nombre d'objets est n=4, La capacité du sac à remplir est C=8=m

Notre algorithme commence par créer un tableau V de n+1 lignes et m+1 colonnes, en initialisant les cases par des 0, ce qui donne dans notre cas:

i \ w	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0

Ensuite, et afin de remplir les autres cases, on suit l'algorithme suivant :

Pour i de 0 à n :

Pour w de 0 à m :

Si **w < w(i)** alors:

V[i,w] = V[i-1,w]

Sinon:

V[i,w]= max{V[i-1,w] ; V[i-1,w-w(i)]+P(i)}

D'où :

Le tableau V devient:

i \ w	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7
4	0	0	1	2	5	6	6	7	8

Finalement, et pour connaitre les objets à mettre, on commence par trouver la case V[i,w] où le profit P est maximal dans le tableau, l'idée est de :

- Soustraire le profit de l'objet correspondant à la ligne i du profit P,
- Mettre cet objet dans le sac,

- Parcourir le tableau et chercher la première case correspondant au résultat P trouvé et recommencer.
- La condition d'arrêt est P <= 0.

Dans notre cas :

P= 8 ;

P= 8 – 6 (profit de l'objet 4) ;

P= 2 ;

P= 2 – 2 (profit de l'objet correspondant à la ligne de la première case où V[i,w]=2 c'est-à-dire l'objet 2)

P= 0 (on s'arrête)

Les objets pris sont dans notre cas l'objet 2 et 4.

Programmation sur python :

On souhaite traiter les objets se trouvant dans le fichier Excel suivant :

	A	B	C	D
1	objet	profit	poids	Capacité
2	1	1	2	8
3	2	2	3	
4	3	5	4	
5	4	6	5	

Voici les étapes suivies lors de la programmation : (**N.B :** lire les commentaires)

1ère étape :

```
import numpy as np # appel de la bibliothèque des vecteurs et matrices

import pandas as pd # appel de la biblothèque qui traite
                    #les fichiers externes(excel dans notre cas)

#appel du fichier excel
df=pd.read_excel("C:\\Users\\dell\\Documents\\KONAMI\\fichierspd\\projet.xlsx")

#convertir les colonnes du tableau en listes
profit=df['profit'].tolist()
poids=df['poids'].tolist()
cap=df['Capacité'].tolist()

#construir la matrice items
items = [poids,profit]
maxWeight=int(cap[0])
```

Résultat :

Nom	Type	Taille	Valeur
cap	list	4	[8.0, nan, nan, nan]
df	DataFrame	(4, 4)	Column names: objet, profit, poids, Capacité
items	list	2	[[2, 3, 4, 5], [1, 2, 5, 6]]
maxWeight	int	1	8
poids	list	4	[2, 3, 4, 5]
profit	list	4	[1, 2, 5, 6]

2ème étape :

```
#backpack est une fonction qui execute l'algo de l'exemple
def backpack(maxWeight,items):
    matrix = [[0 for col in range(maxWeight+1)] for row in range(len(items[0]))]
    for row in range(len(items[0])):
        for col in range(maxWeight+1):
            if items[0][row] > col:
                matrix[row][col] = matrix[row-1][col]
            else:
                matrix[row][col] = max(matrix[row-1][col], matrix[row-1][col-items[0][row]]+ items[1][row])
    for i in range(len(items[0])):
        print(matrix[i])
    global packed
    packed=[]
    col = maxWeight

    for row in range(len(items[0])-1,-1,-1):
        if row == 0 and matrix[row][col] != 0:
            packed.insert(0,row)
        if matrix[row][col] != matrix[row-1][col]:
            packed.insert(0,row)
            col -= items[0][row]
    print(packed)
    print('Max value is ', matrix[len(items[0])-1][maxWeight])
    return packed

#execution de la fonction backpack
backpack(maxWeight,items)
```

Résultat :

Nom	Type	Taille	Valeur
cap	list	4	[8.0, nan, nan, nan]
df	DataFrame	(4, 4)	Column names: objet, profit, poids, Capacité
items	list	2	[[2, 3, 4, 5], [1, 2, 5, 6]]
maxWeight	int	1	8
packed	list	2	[1, 3]
poids	list	4	[2, 3, 4, 5]
profit	list	4	[1, 2, 5, 6]

Console IPython

Console 1/A

```
...:                 col -= items[0][row]
...:         print(packed)
...:         print('Max value is ', matrix[len(items[0])-1][maxWeight])
...:         return packed
...:
...:
...: #execution de la fonction backpack
...: backpack(maxWeight,items)
[0, 0, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 1, 2, 2, 3, 3, 3, 3]
[0, 0, 1, 2, 5, 5, 6, 7, 7]
[0, 0, 1, 2, 5, 6, 6, 7, 8]
[1, 3]
Max value is  8
```

3ème étape :

```
53 a=np.array(df) # convertir df en matrice
54 b=[0, 0, 0, 0] # construire le vecteur
55 n=a.shape[0]
56 d=a.tolist()
57 for i in range(0,n):
58     if d.index(d[i]) in packed:
59         c=np.array(d[i])
60         m=np.vstack((b,c))
61         b=m
62         print(m)
63
64 p=np.delete(m,0,axis=0)
65 p=np.delete(p,3,axis=1)
66
67 print(p)
```

Résultat :

```
[[ 0.  0.  0.  0.]
 [ 2.  2.  3. nan]]
[[ 0.  0.  0.  0.]
 [ 2.  2.  3. nan]
 [ 4.  6.  5. nan]]
[[2. 2. 3.]
 [4. 6. 5.]]
```

4ème étape :

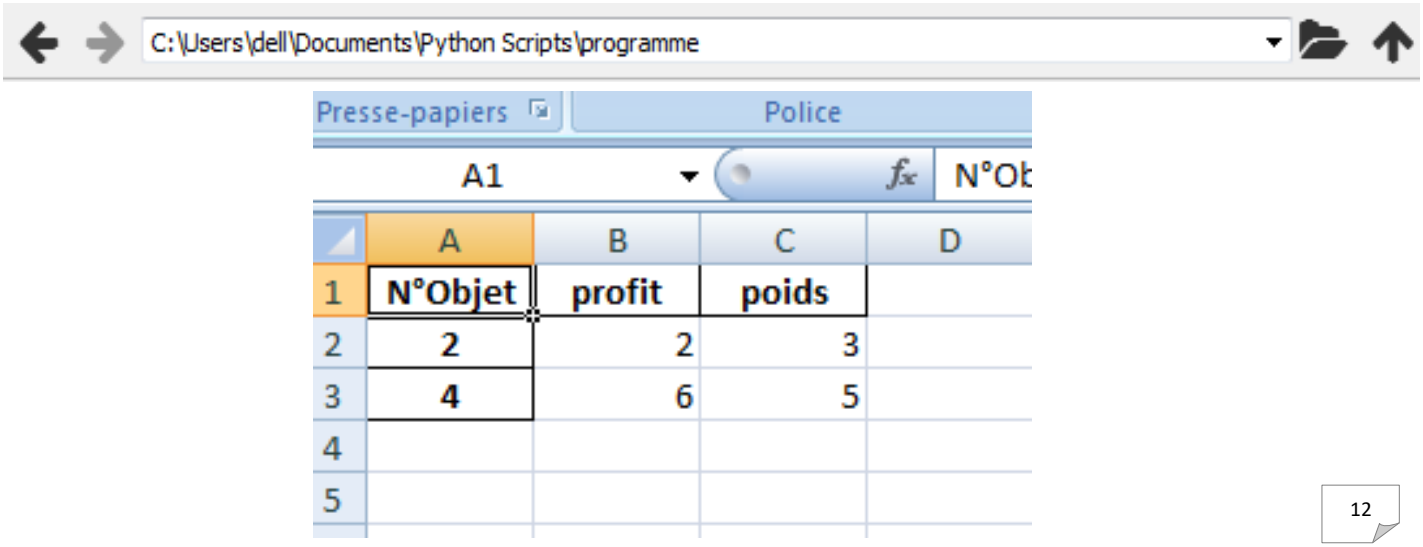
Dans cette étape, on veut que le programme nous donne en sortie un fichier Excel contenant les objets choisis ainsi que leurs poids et profits :

```
69 df2=pd.DataFrame(p,columns=['N°Objet','profit','poids'])# construire la dataframe de sortie
70
71
72 df2=df2.set_index('N°Objet')#mettre le numéro d'objet en index
73
74 df2.to_excel('Objets_à_prendre.xlsx')# convertir le résultat en fichier excel
```

Résultat :

N°Objet	profit	poids
2.0	2.0	3.0
4.0	6.0	5.0

Le résultat final est enregistré sous forme de fichier Excel dans :



Branch and bound :

Définition 1 :

Une heuristique est une méthode de calcul qui fournit rapidement une solution réalisable, pas nécessairement optimale ou exacte, pour un problème d'optimisation difficile.

Définition 2 :

Les méthodes de branch-and-bound sont des méthodes basées sur une énumération "intelligente" des solutions admissibles d'un problème d'optimisation combinatoire.

Voici un exemple : (l'algorithme de Horowitz-Sah (HS))

Soit le tableau suivant :

Objet	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Poids W_i	**31**	**10**	**20**	**19**	**4**	**3**	**6**
Profit P_i	**70**	**20**	**39**	**37**	**7**	**5**	**10**

Le nombre d'objets est n=7 et la capacité est C=50.

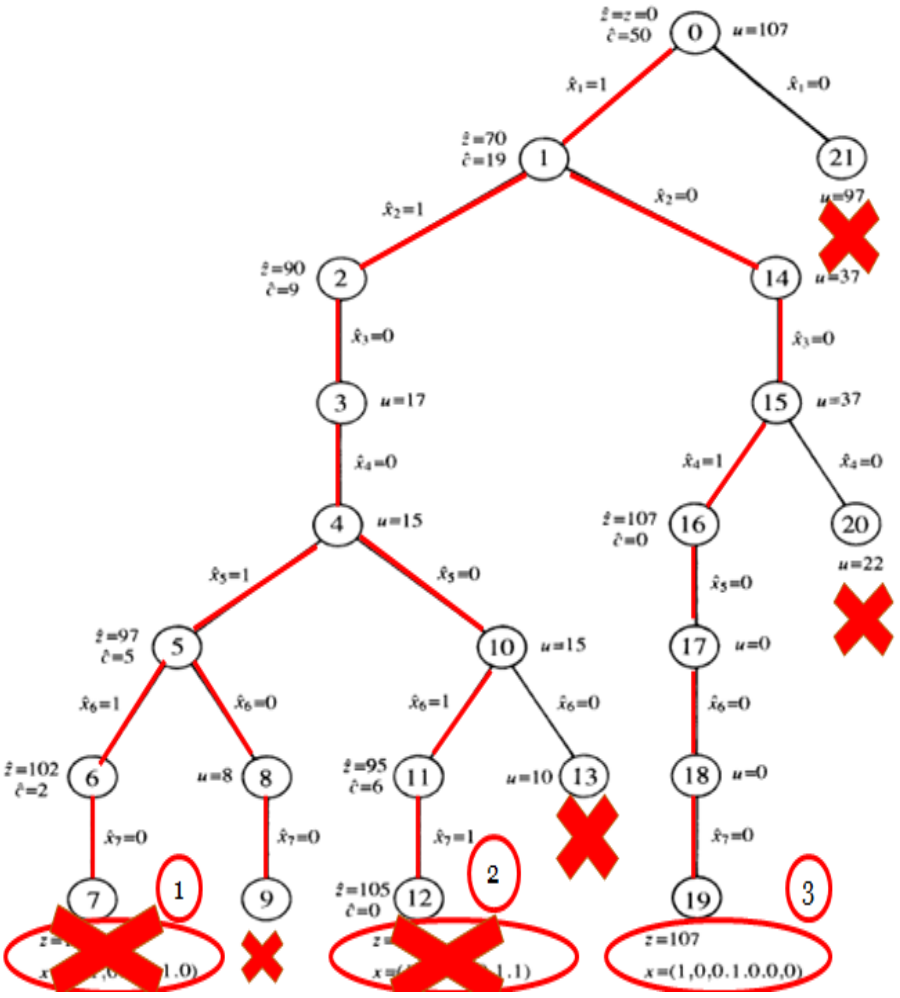
On suppose que les objets sont déjà classés par ordre décroissant en fonction du ratio $r_i = \frac{Pi}{Wi}$ et en appliquant la GREEDY METHOD on calcule la valeur du profit qu'on va noter U et qui sera notre borne supérieure (upper bound), donc si on trouve un profit supérieur à cette valeur on saura automatiquement que notre calcul est faux.

-Procédure

L'idée est de parcourir chacune des branches dans le but de remplir le sac jusqu'à atteindre sa capacité, ensuite à chaque fois enlever le dernier objet pris et le remplacer si c'est possible par un ou plusieurs suivant l'ordre et dans le but de maximiser le profit.

A chaque fois on trouve un résultat, ce dernier devient notre borne inférieure et notre recherche devient de plus en plus réduite.

On s'arrête ou bien après un nombre prédéfini d'itérations (ou une durée précise), ou bien une fois on atteint la borne supérieure.



REMARQUE:

Ceci n'est qu'un exemple d'heuristique qui illustre le raisonnement suivi.

II. Multiple Knapsack (MKP)

1. Concept

Le problème du sac à dos multiple se présente sous la forme mathématique suivante :

$$(MKP)\begin{cases} \max \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij}, \\ \text{s.c} \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \ i \in \{1,...,m\}, \\ \sum_{i=1}^{m} x_{ij} \leq 1, \ j \in \{1,...,n\}, \\ x_{ij} \in \{0,1\}, \ i \in \{1,...,m\}, \ j \in \{1,...,n\}. \end{cases}$$

Le principe est le même que celui du cas single : Remplir un sac de capacité C de telle sorte que la somme des profits des objets choisis soit maximisée et à condition que la somme de leurs poids ne dépasse pas C.

Sauf que cette fois nous n'avons pas qu'un seul sac, mais plusieurs à remplir (Soient m sacs). Le but est donc de trouver une combinaison d'objets X_{ij} de poids W_j et de profit P_j à mettre dans des sacs C_i, avec l'objectif de maximiser la somme des profits de ces objets choisis et à condition que la somme des poids des objets mis dans chaque sac ne dépasse pas la capacité de ce dernier.

Avec X_{ij} est une variable binaire : si l'objet j est mis dans le sac i, la valeur de X_{ij} prend 1 ; dans l'autre cas, elle prend 0.

Afin d'écarter les cas triviaux, nous nous plaçons dans le cas où :

$\max_{j \in N} w_j \leq \max_{i \in \{1,...,m\}} c_i$: il n'existe pas d'objet dont le poids est supérieur à toutes les capacités.

$\min_{i \in \{1,...,m\}} c_i \geq \min_{j \in N} w_j$: il n'existe pas de sac dont la capacité est inférieure à tous les poids des objets.

$\sum_{j=1}^{n} w_j \geq \sum_{i=1}^{m} c_i$: la somme des poids est toujours supérieure à la somme des capacités, sinon on n'aura pas de problème. (solution triviale).

2. Le cas fractionnaire :

Comme nous l'avons déjà mentionné dans le cas single, le fait de rendre les objets fractionnaires est une relaxation du problème original qui nous donne la solution la plus optimale dans des conditions idéales : on utilise la GREEDY METHOD pour résoudre le problème.

On considère dans ce cas que l'ensemble des sacs constituent un seul de capacité égale à la somme de toutes les capacités.

REMARQUE:

Le cas fractionnaire est d'une très grande utilité dans la résolution de ce genre de problèmes, il nous aide à déterminer dans notre situation le résultat parfait vers lequel on cherche à converger dans notre recherche.

3. Le cas 0/1 :

Toujours dans l'analogie avec le single knapsack, dans le cas 0/1 les objets doivent être pris entièrement, c'est-à-dire que si on dépasse la capacité d'un sac C_i, l'objet responsable du dépassement est automatiquement rejeté ou mis dans un autre sac et remplacé si c'est possible par un autre objet.

La programmation dynamique dans ce cas consiste à faire toutes les itérations pour parcourir toutes les possibilités et trouver la plus optimale, ceci prendra une infinité de temps et de mémoire pour un nombre important d'objets et de sacs. De plus, un algorithme qui donne la solution optimale pour n'importe quel nombre de sacs n'existe pas (il existe quelques propositions sur internet mais rien n'est prouvé sûr).

Cependant il existe quelques méthodes de résolution (des heuristiques) qui peuvent approcher la solution optimale à travers des itérations en faisant une recherche intelligente, comme l'algorithme de Martello et Toth que nous allons voir en exemple :

Soient les données suivantes :

$$
\begin{aligned}
n &= 9\ ; \\
m &= 2\ ; \\
(p_j) &= (80,\ 20,\ 60,\ 40,\ 60,\ 60,\ 65,\ 25,\ 30); \\
(w_j) &= (40,\ 10,\ 40,\ 30,\ 50,\ 50,\ 55,\ 25,\ 40); \\
(c_i) &= (100,\ 150).
\end{aligned}
$$

Avec n le nombre d'objets et m le nombre de sacs.

Les objets sont classés selon la GREEDY METHOD (par ordre décroissant du ratio r_j) et les sacs par ordre croissant.

L'idée est de commencer tout d'abord par remplir les sacs par ordre jusqu'à saturation, ensuite on fait un réaménagement des objets déjà pris de telle sorte que le premier soit dans le deuxième sac et le deuxième objet dans le premier sac puis le troisième dans le deuxième sac et ainsi de suite jusqu'au dernier déjà pris :

Tout d'abord :

$$
\begin{aligned}
(y_j) &= (1,\ 1,\ 1,\ 2,\ 2,\ 2,\ 0,\ 0,\ 0), \\
z &= 320\ .
\end{aligned}
$$

Puis réaménagement :

$$
(y_j) = (2,\ 1,\ 2,\ 1,\ 2,\ 1,\ 0,\ 0,\ 0),
$$

Avec :

$$
(\overline{c}_i) = (10,\ 20).
$$

Une première amélioration à faire est d'ajouter si c'est possible un objet dont le poids est inférieur à la capacité résiduelle (la capacité qui reste) des deux sacs combinés, suivie d'une deuxième amélioration qui consiste à remplacer un objet déjà pris par un autre dans le but d'augmenter le profit :

Première amélioration :

$$
\begin{aligned}
(y_j) &= (1,\ 1,\ 2,\ 2,\ 2,\ 1,\ 0,\ 2,\ 0), \\
z &= 345\ .
\end{aligned}
$$

Avec :

$$
(\overline{c}_i) = (0,\ 5),
$$

Deuxième amélioration :

$$(y_j) = (1, 1, 2, 2, 0, 1, 2, 2, 0),$$
$$z = 350,$$

Avec :

$$(\overline{c}_i) = (0, 0),$$

REMARQUE:

Le but de cet exemple est de connaitre la logique avec laquelle on procède dans cette heuristique et non pas l'algorithme.

Donc pour résumer, afin de résoudre ce genre de problèmes, on doit chercher une heuristique selon les conditions données ainsi que le temps de recherche voulu.

Dans la partie suivante, nous allons faire une modélisation du problème industriel et proposer ensuite une heuristique selon les contraintes imposées.

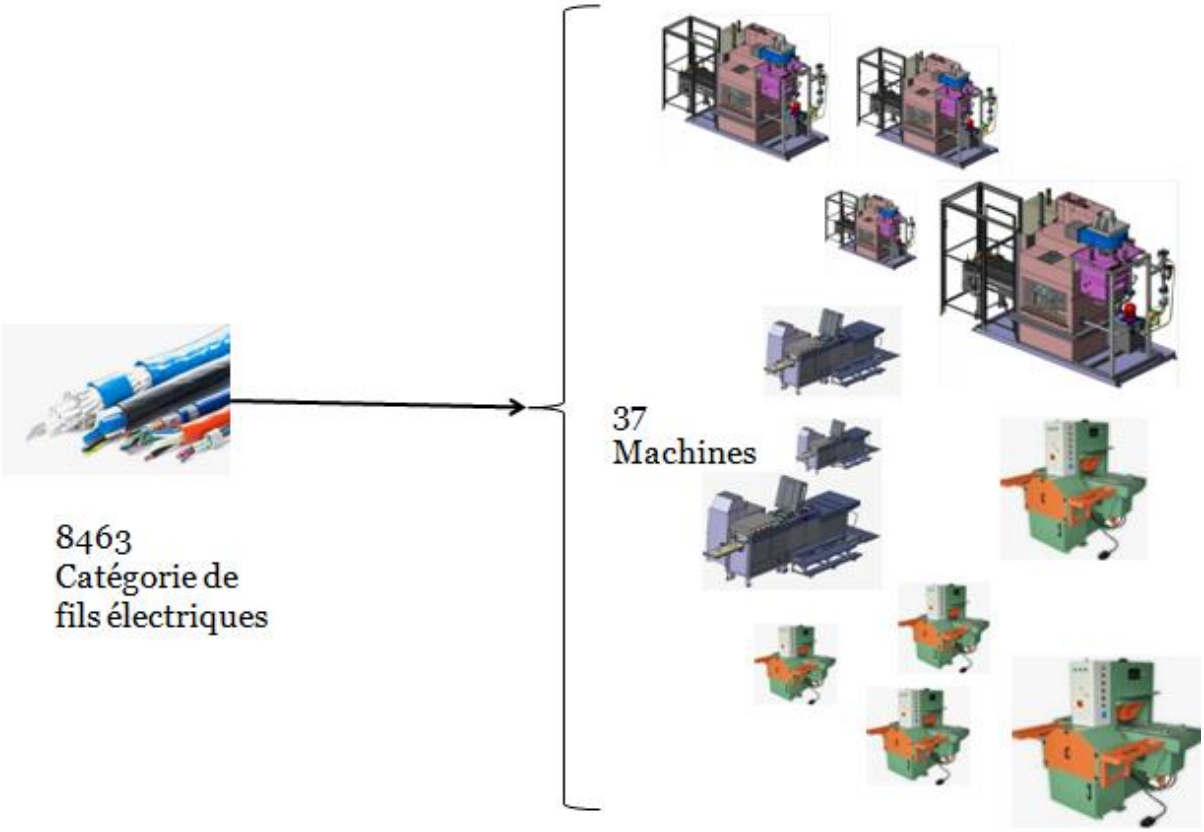
Chapitre 2 : Le problème industriel

I. La modélisation du problème

1. Modélisation

D'après l'étude effectuée au sein de l'usine (YAZAKI), le but est de connaitre le nombre d'articles à mettre dans chaque groupement de machines pour pouvoir en traiter le maximum.

Voici un peu à quoi ressemble notre problème :

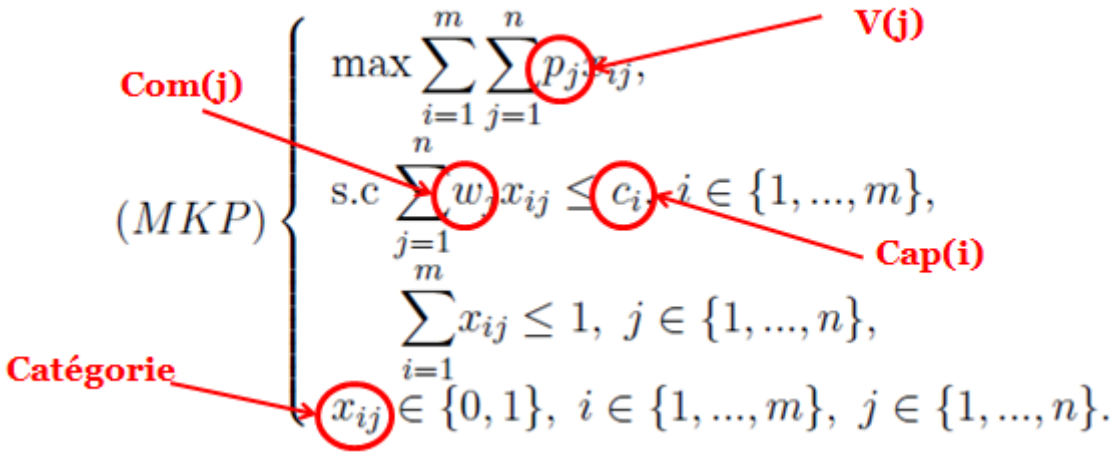


On cherche à distribuer n=8463 catégories de fils électriques sur m=37 machines, pour chaque catégorie, nous avons un **volume commandé (c'est à dire un nombre d'articles commandées),** et chaque machine a une **capacité** qu'elle ne doit pas dépasser. L'objectif est de maximiser le nombre d'articles traités par les machines, c'est-à-dire le **volume traité**. S'ajoute à tout ceci la contrainte de faire passer la commande toute entière, ça veut dire qu'un

volume commandé doit être traité à la fois et non pas fractionné ou divisé sur plusieurs machines.

Notre situation est assimilée à un problème de sac-à-dos multiple, mais afin de faire une modélisation, nous devons faire une analogie entre les paramètres du MKP et les contraintes imposées ci-dessus. Donc nous auront dans notre cas :

W_j : le volume commandé = **Com_j**
C_i : la capacité de la machine i = **Cap_i**
P_j : le volume traité par une machine = **V_j**
X_{ij} : prend 1 si tout le volume commandé de la catégorie j est traité par la machine i et 0 sinon.



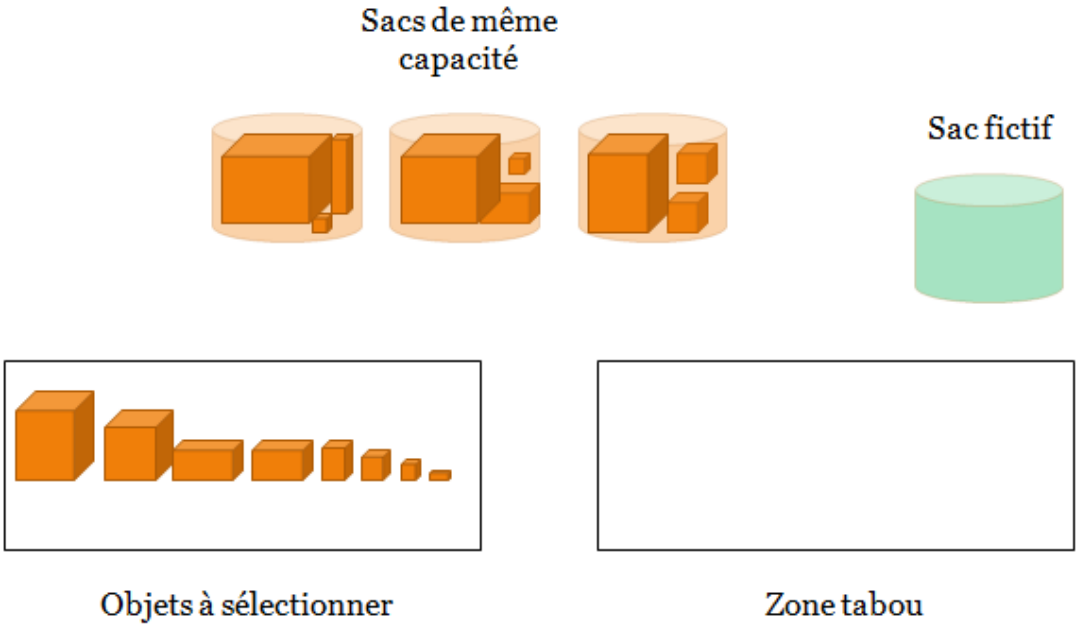
Comme il est déjà mentionné, ce problème ne peut pas être résolu par programmation dynamique puisqu'on aura besoin d'un temps exponentiel pour trouver la solution optimale, d'où la nécessité d'une heuristique.

L'idée est de trouver une solution faisable selon les conditions imposées et puis chercher à l'optimiser sans dépasser un nombre d'itérations prédéfini.

2. La méthode de résolution proposée

Comme méthode de résolution, on commence tout d'abord par relaxer le problème en supposant que les machines ont la même capacité. S'ajoute à cela la condition que les machines doivent être remplies d'une manière équilibrée pour satisfaire le cahier de charges.

Le principe de l'heuristique :



Notre heuristique consiste à classer en premier temps les **n** commandes selon le volume commandé et suivant un ordre décroissant, puis les placer l'une après l'autre dans les **m** machines (**m<n**) jusqu'à la dernière (la $m^{ème}$ machine); ensuite on complète les machines où il y a le plus d'espace jusqu'à arriver à la saturation, la somme des volumes traités des commandes est la première solution faisable qu'on va chercher à optimiser par la suite.

Ensuite, et pour une première optimisation, l'idée est de remplacer le sac (la machine) avec le moins de profit (volume traité) par un sac fictif qui contient une combinaison d'objets (catégories) plus profitable. Si c'est le cas, on prend ces objets à la place de ceux déjà existants; dans le cas contraire, on met les objets déjà existants et les objets du sac fictif dans une zone qu'on appellera zone tabou et on parcourt les autres objets non sélectionnés. Le sac fictif est rempli suivant l'ordre déjà effectué (décroissant).

Après chaque optimisation, on refait la même procédure.

L'algorithme :

Les entrées:

-le nombre d'objets n ;

-le nombre de machines m ;

-W_j le poids de l'objet j ;

-C_i la capacité de la machine i ;

Les sorties :

-Z la somme des profits ;

-(X_j) le vecteur d'objets pris dans les sacs ;

[Initialisation] :

Z = 0, (X_j) = 0, op = 0, Z_i = 0, $\overline{C_1}$= 0, S_i = Ø, U =$\sum_1^m$ Ci;

[Première solution] :

Pour i de 1 à m :

$\overline{C_1}$= C_i ;

Fin pour ;

Pour j de 1 à m :

$\overline{C_1}$=$\overline{C_1}$ - W_j;

Z_i=Z_i+W_j;

X_j=1;

{j}⊆ S_i;

Fin pour;

Pour j de m+1 à n :

i= indice (max $\overline{C_1}$)

Si W_j <= $\overline{C_1}$:

$\overline{C_1}$=$\overline{C_1}$ – W_j ;

Z_i=Z_i + W_j;

X_j=1;

{j}⊆ S_i;

Fin si ;

Fin pour ;

Z=$\sum_1^m$ Zi ;

Return Z, (X_j) ;

Si Z = U : Fin.

[Optimisation] :

C(test) = C_1; Z(test) = 0; S(test) = Ø; T = Ø; a = 0;

(*) {a = min(Z_i) ;

Pour i de 1 à m :

Si Z_i = a :

$S_i \subseteq T$;

Fin si ;}

Tant que k<n et Z<U :

Pour j de 1 à n :

Si j est un objet non sélectionné :

Si W_j <= C(test) :

C(test) = C(test) – W_j;

Z(test) = Z(test) + W_j;

{j}⊆ S(test) ;

Fin si ;

Fin si ;

Fin pour;

Pour j de 1 à n :

Si j dans T :

Si W_j <= C(test) :

C(test) = C(test) – W_j;

Z(test) = Z(test) + W_j;

{j}⊆ S(test) ;

Fin si ;

Fin si ;

Fin pour ;

Si Z(test) > a :

Pour i de 1 à m :

Si Z_i =a :

Pour j dans S_i :

$\overline{C_1} = C_1$;

$Z_i = 0$;

$S_i = \emptyset$;

Fin pour ;

Fin si ;

Fin pour ;

i = indice(min Z_i);

Z_i = Z(test);

$\overline{C_1}$ = C(test);

S_i = S(test);

Pour j dans T :

i= indice (max $\overline{C_1}$)

Si W_j <= $\overline{C_1}$:

$\overline{C_1}=\overline{C_1} - W_j$;

$Z_i=Z_i + W_j$;

$X_j=1$;

$\{j\} \subseteq S_i$;

Fin si ;

Fin pour ;

$Z=\sum_1^m Zi$;

(*) ;

Sinon :

S(test) ⊆ T ;

(*) ;

Fin si ;

k = k + 1 ;

Le programme sur python :

Afin de mieux comprendre l'algorithme, nous allons traiter l'exemple suivant :

Soient :

n = 8 ;

m = 3 ;

$C_1 = C_2 = C_3$;

Objet	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Poids	18	15	13	11	10	8	6	2

1ère étape : importer les données (les entrées) et la création des matrices.

```
import numpy as np
n=int(input())
m=int(input())
#création des listes
c=list()#capacités restant dans le sac(variable)
x=list()#objets
z=list()#profits
w=list()#poids
S=np.zeros((n,m))
Cap=[]#capacités des sacs(cst)

#création des cases des listes initialisées par des 0
for i in range(0,n):
    x.append(i)
    x[i]=0

for i in range(0,m):
    z.append(i)
    z[i]=0

for i in range(0,m):
    c.append(i)
    c[i]=int(input())#faire entrer les valeurs des capacités
    Cap.append(c[i])

u=0
for i in range(0,m):
    u=u+c[i]

for i in range(0,n):
    w.append(i)
    w[i]=int(input())#faire entrer les valeurs des poids

```

2ème étape : la première solution.

```
#remplir les sacs selon la capacité maximale qui reste
for j in range(m+1,n):
    i=c.index(max(c))
    if w[j]<=c[i]:
     c[i]=c[i]-w[j]
     z[i]=z[i]+w[j]
     x[j]=1
     S[j][i]=1

#calculer le profit obtenu
Z=0
for i in range(0,m):
    Z=Z+int(z[i])

#poser une valeur op égale au profit obtenu jusqu'à présent
Op=Z

#afficher ce profit ainsi que les objets
print(Z)
print(x)
```

Résultat :

```
54
[1, 1, 1, 0, 0, 0, 1, 1]
```

3ème étape : l'optimisation

```
#création des variables de test
c_test=int(Cap[0])
z_test=0
S_test=np.zeros(n)#Sac test provisoir
taboo=np.zeros(n)#Sac taboo des objets rejetés
a=0


def tab():  #chercher le ou les sacs min et mettre leurs objets dans le sac taboo
    global a
    a=min(z)
    for i in range(0,m):
        if z[i]==a:
            for j in range(0,n):
                if S[j][i]==1:
                    taboo[j]=S[j][i]
    return a

tab()
k=0
```

Tant que le nombre d'itération n'est pas atteint et le profit n'est pas le plus optimal ;

```
while k<n and Z<u:

    for j in range(0,n):
        if taboo[j]==0 and x[j]==0:
            if w[j]<=c_test:            #mettre les objets qui ne sont ni
                c_test=c_test-w[j]      #taboo ni pris dans les sacs dans
                z_test=z_test+w[j]      #un sac test
                S_test[j]=1
    for j in range(0,n):
        if taboo[j]==1:
            if w[j]<=c_test:
                c_test=c_test-w[j]      #ajouter après si c'est possible
                z_test=z_test+w[j]      #les objets taboo
                S_test[j]=1
```

```
    if z_test>a:
        for i in range(0,m):
            if z[i]==a:
                for j in range(0,n):
                    c[i]=Cap[i]
                    z[i]=0
                    if int(S[j][i])==1:
                       x[j]=0
                       S[j][i]=0

        i=z.index(min(z))
        z[i]=z_test
        c[i]=c_test
        for j in range(0,n):
            S[j][i]=int(S_test[j])
            if int(S[j][i])==1:
                x[j]=1
                taboo[j]=0
        for j in range(0,n):
            if taboo[j]==1:              #on complète par les objets taboo si possible
                i=c.index(max(c))       #afin d'augmenter le profit global
                if w[j]<=c[i]:
                    c[i]=c[i]-w[j]
                    z[i]=z[i]+w[j]
                    x[j]=1
                    S[j][i]=1
        Z=0
        for i in range(0,m):
            Z=Z+int(z[i])
            Op=Z
        c_test=int(Cap[0])
        z_test=0
        S_test=np.zeros(n)
        taboo=np.zeros(n)
        tab()
        print(Z)
        print(x)
```

Dans l'autre cas :

```
    else:
        for i in range(0,n):
            if S_test[i]==1:
                taboo[i]=1              #dans le cas contraire on met ces objets
        c_test=int(Cap[0])              #dans le sac taboo et on reprend du début
        z_test=0
        S_test=np.zeros(n)
        tab()
    k=k+1
```

Le résultat final est :

```
58
[1, 0, 1, 1, 0, 1, 1, 1]
```

3. Limites et perspectives

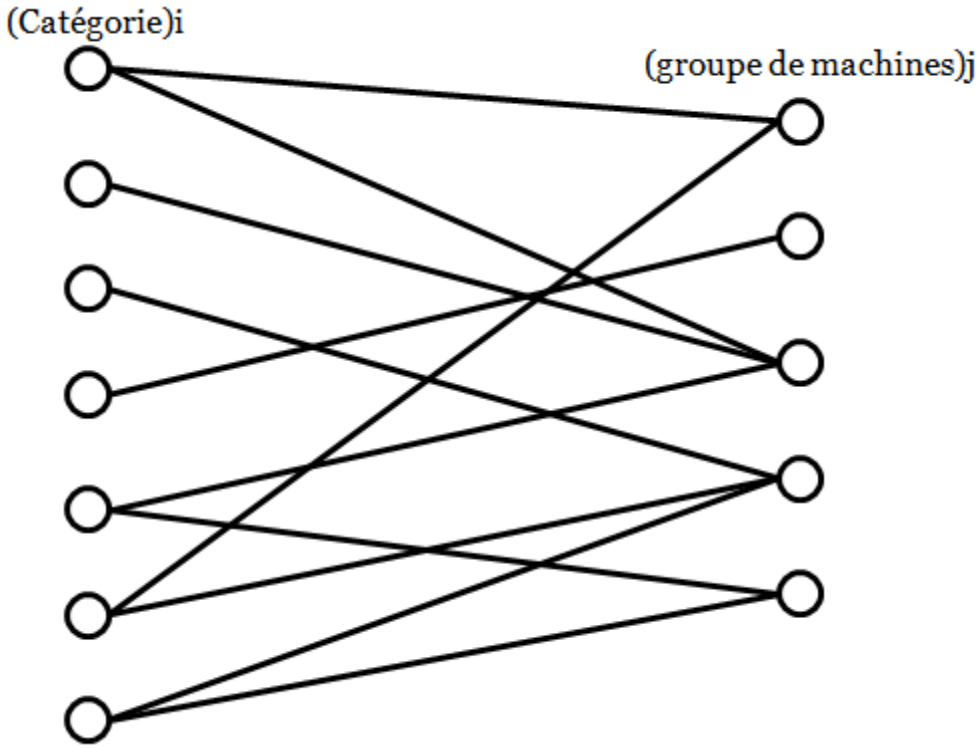
Notre heuristique a des limites dans son fonctionnement, ceci est dû au fait que la recherche se fait de la même façon pour toutes les itérations, donc on se trouve parfois dans des cas où la solution alterne entre deux valeurs ; afin de remédier à ce problème, nous avons fait des améliorations dans le programme en changeant les conditions de recherche (au lieu de remplacer le sac si on obtient un profit semblable au profit courant pour essayer d'autres combinaisons, le changement n'est devenu que dans le cas d'amélioration de profit) ; néanmoins l'inconvénient avec cette méthode est que le programme ne parcourt pas assez de cas et risque de ne faire que quelques optimisations même s'il reste encore du temps pour faire mieux.

Afin de performer le programme, il est possible de créer une heuristique pour améliorer le résultat trouvé en supprimant par exemple des cas déjà pris ou impossible d'être inclus dans une solution faisable. On peut aussi procéder par le Data Crunching, c'est-à-dire faire une étude préliminaire des données à traiter (les différentes commandes) afin de segmenter le problème.

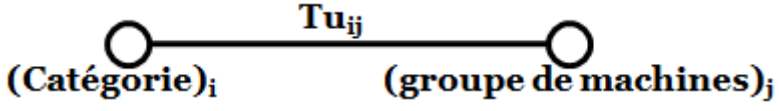
Conclusion et perspectives

Le problème du sac à dos est l'un des problèmes les plus connus dans la recherche opérationnelle, dont la formulation est fort simple, mais la résolution est plus complexe.

Dans notre travail nous avons proposé une heuristique pour la résolution d'un problème industriel similaire à celui du sac à dos multiple, cependant, dans le cas réel nous devons faire une analyse multicritère vu qu'on a aussi un problème d'affectation qui se pose lors de la distribution des articles sur les machines.



L'idée est de trouver pour chaque catégorie d'articles un groupe de machines où ces derniers vont être traités ; pour ce faire, on propose que l'affectation soit faite de sorte que le taux d'utilisation (Tu=$\frac{Charge}{capacité}$) dans tous ces groupements soit équilibré, donc la pondération va être selon ce taux d'utilisation :



Bibliographie et Webographie

Bibliographie

- Rapport du Projet de fin d'études de madame TAQUI Wissal intitulé **'Optimisation de la distribution de la zone de coupe par recherche opérationnelle'.**
- **Knapsack problems :Algorithms and computer implementations** de **Silvano MARTELLO** et **Paolo TOTH.**

Webographie

- https://en.wikipedia.org/wiki/Knapsack_probl
- https://www.youtube.com/watch?v=nLmhmB6NzcM&t=742s
- https://www.youtube.com/watch?v=oTTzNMHM05I
- https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python