Jusqu'à présent nous avons utilisé le solveur intégré de *Calc* pour résoudre nos problèmes. Souvent, à partir des données du problème, nous avons besoin de traitement en amont (calcul des matrices de distances, dynamicité des données...). Il serait intéressant d'automatiser ces traitements, c'est ce que nous allons faire ici. Nous aurions pu le faire dans *Calc* en utilisant le LibreOffice Basic, mais nous allons utilisé *Python* pour illustrer notre propos.

1 Installation

Nous allons utiliser la bibliothèque *pulp* :

- *pulp* va nous servir de modeleur pour décrire nos problèmes,
- *python* servira à faire les calculs nécéssaires,
- *pulp* peut résoudre lui même le problème ou appeler un solveur externe (glpk, cplex, gurobi...)

Sous linux ou via X2go

Dans un terminal, tapez l'instruction suivante :

```
pip3 install pulp
```

Vous pouvez maintenant lancer *python* (nous prendrons la version 3) et utiliser n'importe quel éditeur de texte.

```
python3
```

Sous windows

Vous pouvez installer *python* et un éditeur de texte ou bien une solution tout-en-un comme *Thonny*. Pour ajouter *pulp*, il faut alors aller dans "Outils" et "Gestion des paquets".

Autre solution, vous pouvez utiliser *repl.it* et travailler dans un bac à sable sur votre navigateur. *pulp* s'installe alors à la volée si nécessaire. Avec cette utilisation en ligne, les temps de calcul sont évidemment plus longs.

2 Utilisation

2.1 Déclaration du problème

Pour voir l'utilisation, nous allons reprendre l'exemple des caisses à charger sur des wagons. Nous avions 16 caisses à charger dans 3 wagons et nous voulions minimiser le poids du plus lourd wagon.

```
caisses=[34,6,8,17,16,5,13,21,25,21,14,13,33,9,25,25]
nb_caisses=len(caisses)
nb_wagons=3
```

Il faut commencer par charger la bibliothèque *pulp* et créer un nouveau problème d'optimisation :

```
from pulp import *
problem = LpProblem("Wagons", LpMinimize)
```

LpProblem prend deux paramètres : le nom du problème et le type (LpMinimize ou LpMaximize).

2.2 Variables

Les variables de décisions sont définies grâce à la méthode **LpVariable**.

```
LpVariable(name, lowBound, upBound, cat)
```

avec

name le nom de la variable,

lowBound la borne inférieure de la variable (optionnel)

upBound la borne supérieure de la variable (optionnel)

cat le type de variable : Integer, Binary or Continuous (par défaut)

Nous pouvons donc définir notre variable k comme :

```
k=LpVariable(name="k",lowBound=0,upBound=1000,cat="Integer")
```

Il faut ensuite une variable w_{ij} binaire indiquant si la caisse j est dans le wagon i. Le plus simple est d'utiliser un dictionnaire pour stocker toutes les variables.

```
w={}
for i in range(nb_wagons):
    for j in range(nb_caisses):
        w[i,j]=LpVariable(name='w_%s_%s'%(j,i),cat='Binary')
```

2.3 Contraintes

Il nous faut ajouter maintenant des contraintes linéaires au problème. La méthode **lpSum** calcule la somme des éléments d'un vecteur contenant des variables de décisions et renvoie une expression affine. Pour l'ajouter au problème, il me suffit ensuite d'écrire

`problem += lpSum(vecteur) == 1`

Dans notre exemple, je veux pouvoir dire que toutes les caisses sont chargées :

```
for j in range(nb_caisses):
    problem += lpSum(w[i,j] for i in range(nb_wagons))==1
```

Mais on veut aussi que le poids de chaque wagon soit contraint par k

```
for i in range(nb_wagons):
    problem += lpSum(w[i,j]*caisses[j] for j in range(nb_caisses))<=k
```

2.4 Fonction objective

La fonction objective s'introduit de la même façon (par ajout à *problem*). Dans notre cas, l'objectif est simplement de minimiser k, donc pas besoin de calculer une somme, nous pouvons directement écrire :

```
problem += k
```

2.5 Résolution

Avant de résoudre le problème, nous pouvons l'afficher pour vérification.

```
print(problem)
problem.solve()
```

La valeur de la fonction objective est donnée par

```
print(value(problem.objective))
```

Les valeurs des variables sont récupérées dans le champ **varValue**.

```
print(k.varValue)
```

Pour utiliser un autre solveur (installé), il suffit de le passer en paramètre :

```
problem.solve(GLPK())
```

2.6 Code complet

Pour avoir un code réutilisable facilement, nous n'avons plus qu'à mettre le tout dans une fonction :

```
def wagon(caisses, nb_wagons):
    nb_caisses=len(caisses)
    problem = LpProblem("Wagons", LpMinimize)
    w={}
    for i in range(nb_wagons):
      for j in range(nb_caisses):
```

```
        w[i,j]=LpVariable(name='w_%s_%s'%(j,i),cat='Binary')
    k=LpVariable(name="k",lowBound=0,upBound=1000,cat="Integer")
    for j in range(nb_caisses):
      problem += lpSum(w[i,j] for i in range(nb_wagons))==1
    for i in range(nb_wagons):
      problem += lpSum(w[i,j]*caisses[j] for j in range(nb_caisses))<=k
    problem += k
    problem.solve()
    sol=[0]*nb_caisses
    for j in range(nb_caisses):
      for i in range(nb_wagons):
        if w[i,j].varValue==1:
            sol[j]=i
    return k.varValue, sol
```

Nous pouvons alors tester

```
caisses=[34,6,8,17,16,5,13,21,25,21,14,13,33,9,25,25]
print(wagon(caisses, 3))
#j'ajoute des caisses et un wagon
caisses=caisses+[35,23,12,10]
print(wagon(caisses, 4))
```

3 Travail à faire

Refaites en *python* tous les exercices faits précédemment !