Kevin QUACH
Dylan DE JESUS

Optimisation - TD3 - Init G1 (Mr Fusy)
Université Gustave Eiffel M1 Informatique 2022-2023

Compte Rendu TD3 Optimisation

Sommaire :

Nous avons réalisé ce TP la plupart du temps à deux sur des postes de l'université. Certaines tâches ont ensuite été divisées à la maison entre les deux membres du groupe avant de revoir l'intégralité du projet ensemble sur un PC de l'université.

Partie 1 - Prise en main de `lp_solve`

Exercice 1 :

On a copié le programme linéaire de l'exemple dans un fichier **Q1.lp** que l'on a lancé à l'aide de la commande : **lp_solve Q1.lp**

On obtient bien les bonnes valeurs.

Exercice 2 :

En changeant l'objectif de l'exemple à max : -2x1 + x2, on observe bien que x1 ne descend pas en dessous de 0 à cause de la contrainte de non négativité.

En essayant avec la ligne :

```
free x1, x2;
```

On obtient le commentaire suivant : This problem is unbounded.

On peut en déduire qu'en enlevant la contrainte de non négativité le problème ne possède plus de valeur limite pour x1.
En effet plus x1 est petit plus la fonction objectif obtiendra une valeur élevée, or en enlevant la contrainte sur x1 celui-ci peut descendre à l'infini ce qui fait qu'il n'y a pas de limite à la fonction objective expliquant le message du solveur.

Exercice 3 :

On traduit les variables de décision et les différentes contraintes de l'exercice 1 du Td1 en un fichier **Q3.lp,** puis on le lance avec la commande : **lp_solve Q3.lp**

On obtient bien les valeurs que l'on avait trouvé durant le TD, c'est-à-dire une valeur objective = 3250 avec x = 15 et y = 10.

Cela signifie que pour avoir un profit maximal qui vaudrait dans nôtre cas un profit de 3 250 € avec les ressources dont le brasseur dispose il faut produire 15 barils de bière blonde et 10 barils de bière brune.

Partie 2 - Un problème générique

Exercice 4 :

Le programme est composé de deux fonctions. Le **main()** permet de récupérer les différents éléments sur la ligne de commande et d'en extraire les différentes données.
On ouvre le fichier texte puis on récupère et stocke les différentes données permettant d'écrire un programme linéaire (le nombre de ressources, le nombre de produits, variables de décisions …)
Une fois terminée la collecte de données du fichier texte, on appelle la seconde fonction du programme **write_lp(...)**.

write_lp(...) prend toutes les données collectées et un fichier de sortie et y écrit le programme linéaire. Elle commence par l'écriture de la fonction objectif puis à celle des contraintes sur les ressources.

Dans le cas où l'option -int est donnée, le **main()** mettra une certaine variable à true et la fonction **write_lp(...)** ajoutera à la toute fin de l'écriture des contraintes sur les ressources une contrainte d'intégralité effective pour chaque variable de décision du programme.

Nous avons testé ce programme sur chacun des fichiers du dossier data fournis.

Pour exécuter ce programme python il faut lancer le script via un terminal à l'aide de la commande :
python3 generic.py [nom fichier texte formaté] [nom du fichier de sortie (.lp)] [option : -int]

Il faut donc lancer cette commande avec certaines restrictions : Le fichier d'entrée doit être un fichier texte formaté suivant les règles données dans le sujet.

Si l'option donnée est différente de "-int" alors le programme fonctionnera comme si on avait pas passé d'option.

On peut passer un nom de fichier qui n'existe pas pour le fichier de sortie (celui-ci sera créé) au contraire le fichier d'entrée doit exister.

Exercice 5 :

Nous avons remarqué que pour omettre les variables null, on peut utiliser l'option **-ia** de **lp_solve** lors du lancement du programme sur la ligne de commande.
On utilise alors la fonction `subprocess.run("lp_solve", "-ia", sys.argv[2])` pour lancer le solveur dans le programme **generic.py**.

Cependant on remarque que l'affichage n'est pas celui souhaité et que l'on obtient également un affichage des solutions intermédiaires.

Nous avons alors décidé de ne pas utiliser cette option. Nous avons préféré ajouter une redirection de la sortie de **lp_solve** vers un fichier **out.txt** dans nôtre programme **generic.py**.

```
os.system("lp_solve " + sys.argv[2] + " > " + "out.txt")
remove_null("out.txt")
```

La fonction **remove_null(filename)** prend un nom de fichier en paramètre (du format d'une réponse donnée par **lp_solve**) et va afficher le contenu de celui-ci en filtrant les variables non nulles.

On lance nôtre programme de la même manière que précédemment cependant il faut cette fois-ci que le fichier de sortie dans la ligne de commande corresponde obligatoirement à un fichier au format .lp.

Exercice 6 :

a) On obtient un bénéfice optimal de 21654.79332331
b) Il faut fabriquer 6 produits différents
c) On lance le programme comme fait précédemment en ajoutant la commande **time** devant. On obtient ces valeurs de temps en secondes ci-dessous :

Terminal

```
real    0m0,137s
user    0m0,074s
sys     0m0,028s
```

Exercice 7 :

En ajoutant les contrainte d'intégralité on obtient :

a) On trouve un bénéfice optimal de 21638.

b) Il faudra fabriquer 12 produits différents.
c) On lance le temps de l'exécution du programme et on obtient ces valeurs de temps.

Terminal

```
real    0m22,986s
user    0m22,801s
sys     0m0,064s
```

On remarque que l'on trouve pour le cas d'intégralité une valeur de bénéfice légèrement plus basse que pour le cas de l'exercice précédent. Il nous faudra fabriquer plus de produits différents dans le cas d'intégralité (2 fois plus) et l'on observe que l'on prend plus de 22 secondes de plus à résoudre le problème avec contrainte d'intégralité des variables que sans.

On peut expliquer ces observations par le fait que comme il n'est plus possible de fractionner un produit, lorsque l'on veut tendre vers la fonction max, il faudra mieux répartir les bénéfices des différents produits. (Par exemple : Si l'on peut obtenir la valeur maximale de 10 euros de bénéfices avec un produit qui aura un bénéfice de 6.7 alors on peut en fabriquer 1,5 quantités pour atteindre cet objectif, en ayant une contrainte d'intégralité on pourra juste fabriquer un seul de ce produit (6,7 de bénéfice) et compléter avec la fabrication d'un autre produit par exemple de bénéfice 3,3)

Enfin la différence de temps entre les deux cas peut s'expliquer par le fait qu'en ajoutant une contrainte d'intégralité on arrive à un problème NP-difficile.

Exercice 8 :

En prenant le fichier **bigdata.txt** on obtient :

Cas de non intégralité
a) On trouve un bénéfice maximum valant 5050533.10609334.
b) On devra produire 100 produits différents pour obtenir ce bénéfice maximal.
c) On obtient ces valeurs de temps :

Terminal

```
real    0m1,092s
user    0m1,015s
sys     0m0,017s
```

Cas d'intégralité

```
real    142m50,231s
user    142m46,253s
sys     0m3,005s
```

Probablement pour les mêmes raisons que sur **data.txt**, le temps d'exécution est beaucoup plus long avec les contraintes d'intégralité. Ce temps est d'autant plus long que le fichier est volumineux.
On rappelle qu'en ajoutant les contraintes d'intégralité on obtient un problème NP-difficile.

Partie 3 - Un problème de découpe

Exercice 9 :

On traduit le problème des barres d'aluminium vu dans le TD précédent par le fichier texte **Barre_alu.txt**.
En lançant le solveur sur ce fichier on obtient un optimum (minimum dans nôtre cas) de 135 unités (Donc 135 barres de 3m).

Pour minimiser le nombre de barres utilisées on doit découper nos barres de 3 mètres de la manière suivante :

```
Actual values of the variables:
x1                            41
x2                            1
x3                            0
x4                            0
x5                            50
x6                            0
x7                            43
```

Soit a = nombre de barres de 0.5 m, b = nombre de barres de 1 m, c = nombre de barres de 1.2 m.

Sachant que l'on a :
x1 = 6a
x2 = 4a + b
x5 = a + 2c
x7 = 3b

On peut écrire :

x1 = 41 : 41 * (6a) ⇔ 246a
x2 = 1 : 1 * (4a + b) ⇔ 4a + b
x5 = 50 : 50 * (a + 2c) ⇔ 50a + 100c
x7 = 43 : 43 * (3b) ⇔ 129b

x1 + x2 + x5 + x7 = 300a + 130b + 100c (Condition respectée)

On a bien **300*0.5 + 130*1 + 100*1.2 < 135 * 3 m (Condition respectée)**

Exercice 10 :

Pour calculer tous les découpages maximaux, on utilise une fonction récursive **decoupages_aux(...)** dans nôtre fichier **barres.py**.
De plus la fonction **decoupages_max(...)** doit impérativement prendre une liste des différentes portions possibles pour les découpages décroissante.

decoupages_aux(...) va, pour chaque découpage possible vérifier s'ils sont maximaux, et, s'ils le sont, les ajoutent dans une liste que l'on renvoie. On fait donc un parcours d'un arbre des différentes possibilités.

Avec une valeur de 300 au départ on a:

(300 - [0, 0, 0])

La première récursion donnera :

(180 - [1, 0, 0]) (200 - [0, 1, 0]) (250 - [0, 0, 1])

Exercice 11 :

On écrit un programme **barres_prog_lin.py** qui va générer le programme linéaire et lancer le solveur.
Le **main()** va récupérer les données nécessaires à la génération du programme linéaire, la longueur de la barre, la liste des longueurs de découpes et la liste du nombre de découpe requise pour chaque découpe.
À l'aide des fonctions définies précédemment, on récupère la liste des découpes maximales et on génère le programme linéaire avec la fonction **write_lp(...)**.

write_lp prend en paramètre les données récupérées et un fichier de sortie et va y écrire le programme linéaire. Cette fois on y écrit les contraintes d'intégralité par défaut car le nombre de découpe effectuée doit être un entier.

Avec des barres de base de 500 cm qu'on souhaite découper en 60 éléments de longueurs 200 cm, 100 éléments de 120 cm, 150 éléments de 100 cm et 350 éléments de 50 cm, on obtient bien un optimum de 114 barres.

Pour lancer le programme, exécutez la ligne de commande suivante:
python3 barres_prog_lin.py [Longueur de la barre] [Liste des longueurs de découpe] [Liste des quantités souhaitées pour chaque découpe] [Nom du fichier de sortie]

Les listes doivent être écrites sans espace entre les différents éléments :

Interdit => [1, 2, 3, 4, 5] ou [1,2,3,4,5] ou [1 ,2 , 3 ,4 ,5]
Autorisé => [1,2,3,4,5]