

RO202 - Initiation à la Recherche Opérationnelle

Zacharie ALES, Tiphaine GEORGE
2025 - 2026

EXERCICES 4 - Programmation linéaire en nombres entiers

Exercise 1 Méthode de Dakin

La méthode de Dakin est un algorithme de branch-and-bound dans lequel le branchement est effectué sur la variable qui a la **plus grande partie fractionnaire** dans la solution en cours.

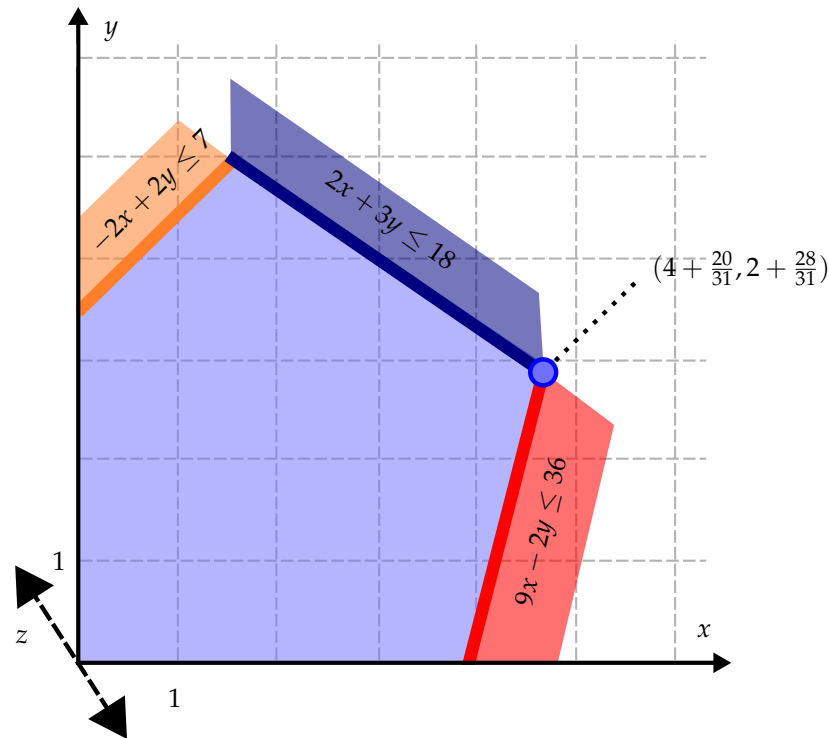
Utiliser cette méthode pour trouver la solution optimale de l'exemple suivant :

$$\begin{array}{ll} \max z = & 3x + 2y \\ \text{s.c.} & -2x + 2y \leq 7 \\ & 2x + 3y \leq 18 \\ & 9x - 2y \leq 36 \\ & x, y \in \mathbb{N} \end{array}$$

Précisions :

- A chaque étape, le sommet sur lequel sera effectué le branchement sera celui dont l'objectif de la relaxation linéaire est le meilleur ;
- En chaque noeud de l'arborescence, vous aurez besoin de déterminer les solutions d'un programme linéaire continu. Vous les trouverez de façon géométrique grâce à la figure suivante.
- Pour chaque programme linéaire continu résolu, vous indiquerez quelle est la solution correspondante ainsi que le noeud de l'arborescence associé et la valeur de l'objectif.
- Afin de gagner du temps, voici une liste des solutions fractionnaires considérées lors de la résolution (pas nécessairement dans cet ordre) :

x	y	z
4	$3 + \frac{1}{3}$	$18 + \frac{2}{3}$
$4 + \frac{4}{9}$	2	$17 + \frac{1}{3}$
$4 + \frac{1}{2}$	3	$19 + \frac{1}{2}$
$4 + \frac{20}{31}$	$2 + \frac{28}{31}$	$19 + \frac{23}{31}$



Exercice 2 Le problème du sac-à-dos

Pour embarquer dans l'avion, votre valise ne doit pas peser plus de K kg. Il ne va pas être possible d'emporter les n objets que vous vouliez y placer et vous devez faire des choix. Vous avez pesé chaque objet et vous notez p_i le poids de l'objet $i \in \{1, \dots, n\}$. Pour optimiser le contenu de la valise, vous avez attribué à chaque objet i une note d'utilité c_i entre 1 et 20. Vous voulez maximiser l'utilité globale de la valise, égale à la somme des utilités des objets emportés.

1. On considère une valise de capacité $K = 17$ et 4 objets de poids respectifs 3, 7, 9 et 6 kg et d'utilités respectives 8, 18, 20 et 11. A chaque objet i ($i \in \{1, 2, 3, 4\}$), on associe une variable binaire x_i égale à 1 si l'objet i est mis dans la valise et 0 sinon.
 - a) Exprimer l'objectif de ce problème.
 - b) Exprimer la contrainte permettant d'assurer que la capacité de la valise n'est pas dépassée.
2. Ecrire le programme mathématique P associé à n objets $i \in \{1, \dots, n\}$ de poids p_i et d'utilité c_i .
3. On suppose maintenant que les objets sont fractionnables. Soit P_R le programme P dans lequel on a remplacé $x_i \in \{0, 1\}$ par $x_i \in [0, 1]$, pour tout $i \in \{1, \dots, n\}$.
Le programme P_R (relaxation continue de P) peut se résoudre de la façon suivante :
 - a - Trier les ratios $\frac{c_i}{p_i}$ par valeur décroissante.
 - b - Mettre les objets dans le sac dans cet ordre tant qu'on ne dépasse pas K .
 - c - Mettre une fraction du premier objet qui dépasse K dans le sac de façon à compléter le poids de la valise (jusqu'à K).

Supposons pour simplifier que les objets soient classés par ratio $\frac{c_i}{p_i}$ décroissant. Soit $imax$ l'indice du dernier objet mis complètement dans le sac (i.e., $\sum_{i=1}^{imax} p_i \leq K$ et $\sum_{i=1}^{imax+1} p_i > K$). On a donc :

- les objets 1 à $imax$ sont entièrement dans le sac ($x_i = 1$ pour $i \in \{1, \dots, imax\}$);
- l'objet $imax + 1$ est partiellement dans le sac ($x_{imax+1} = \frac{K - \sum_{i=1}^{imax} p_i}{p_{imax+1}}$);
- les autres objets ne sont pas dans le sac ($x_i = 0$ pour $i \in \{imax + 2, \dots, n\}$).

Quelle est la solution de P_R pour l'instance mentionnée dans la question 1 ?

4. Résoudre le problème P pour cette instance par une méthode arborescente. En chaque nœud on obtient un majorant en résolvant la relaxation linéaire du problème en ce nœud.

Précisions : la séparation (en 2 sous-arbres) doit être faite sur la variable x_i non entière qui a la plus grande utilité : $x_i = 1$ ou $x_i = 0$. On déduit éventuellement des implications consécutives à chaque choix (c'est-à-dire qu'on fixe à 0 les variables associées à des objets qui ne pourront plus entrer dans la valise une fois que ce choix est fait). Par exemple, si les objets 3 et 4 sont dans le sac, il n'y a plus de place pour les objets 1 et 2 et on peut donc fixer $x_1 = x_2 = 0$.

Exercice 3 Implémentation du *branch-and-bound*

La classe `BBTree` (fichier `bbTree.py`) représente l'arborescence d'un algorithme de *branch-and-bound* pour un problème donné. Chaque noeud d'un `BBTree` est représenté par une instance de la classe `BBNode` (fichier `bbNode.py`).

Attributs de `BBNode`

- `tableau` : tableau représentant le PL associé au noeud ;
- `depth` : profondeur du sommet dans l'arbre (utilisé uniquement pour effectuer les affichages écran).

Méthodes de `BBNode`

- `bbNode.create_root(A, rhs, obj, isMinimization)`
Constructeur d'un noeud racine ;
- `bbNode.create_non_root(parent, newA, newRhs)`
Constructeur d'un noeud qui n'est pas une racine
(`newA` et `newRhs` correspondent à la dernière contrainte de branchement ajoutée) ;
- `branch(tree)`
Effectue la résolution de la relaxation linéaire du problème associé au sommet. Si une solution fractionnaire est obtenue, le branchement est effectué et la méthode `branch` est appelée sur les deux fils ainsi créés (vous devrez implémenter cette méthode).

Attributs de `BBTree`

- `bestSolution` : meilleure solution entière actuellement trouvée dans l'arbre (ou `None` si aucune solution entière n'a été trouvée jusqu'ici) ;
- `bestObjective` : valeur de l'objectif de la solution `bestSolution` ;
- `root` : noeud racine de l'arborescence.

Méthodes de `BBTree`

- `BBTree(bbNode)`
Constructeur d'un arbre dont la racine est passée en argument ;
- `solve()`
Résolution du PLNE associé à la racine ;
- `displaySolution()`
Affiche la solution obtenue après résolution.

1. Compléter la méthode `BBNode.branch()` en implémentant l'algorithme représenté en Figure 1 (pensez à prendre en compte l'attribut `isMinimization`).

Indications :

↙ Utilisez votre méthode `Tableau.pivot()` du TP précédent

- `Tableau.applySimplexPhase1And2()` : résout la relaxation linéaire d'un tableau (même lorsque la base initiale n'est pas réalisable) ;
- `t.bestSolution` sera `None` après `t.applySimplexPhase1And2()` si le programme linéaire est infaisable ;
- `isFractional(d)` : teste si un float une variable de type float correspond à un entier ;
- `math.ceil(double)` : renvoie la partie supérieure d'un réel ;
- `math.floor(double)` : renvoie la partie inférieure d'un réel.

2. Ajouter un affichage graphique permettant de visualiser les étapes de l'algorithme. Afin de visualiser la structure de l'arbre, la marge de gauche d'un sommet sera proportionnelle à sa profondeur comme dans l'exemple ci-dessous :

root : $x[2] = 0,77$

$x_2 \leq 0.0$: Integer solution : $z = 4,00$, $(x_1) = (2)$

$$x_2 \geq 1.0 : x[3] = 1.25$$

$x_2 \leq 1.0 : x[1] = 0,33$

x1 >= 1.0 : Node cut : relaxation worst than bound
(node relaxation : 9.00, best integer solution : 10.0)

x3 >= 2.0 : Node cut : infeasible solution

Optimal solution : $z = 10,00, (x_2, x_3) = (1, 1)$

Données :

bestSolution : Meilleure solution entière connue (null si aucune solution connue)

si R admet une solution S alors

si bestSolution est null **ou** si S est meilleure que bestSolution **alors**

$$\text{bestSolution} \leftarrow S$$

id ← indice de la 1ère variable fractionnaire dans S ← Valeur de x_{id} dans S

Créer un sommet contenant la contrainte $x_{jd} \geq \lceil S(x_{jd}) \rceil$ et brancher dessus

Exercice 4 Problème de séparation par l'inégalité de couverture

On définit une *couverture* C comme un ensemble d'objets ne tenant pas dans le sac (i.e., $C \subseteq \{1, \dots, n\}$ tel que $\sum_{i \in C} p_i > K$). L'inégalité $\sum_{i \in C} x_i \leq |C| - 1$ est appelée *inégalité de couverture*.

2. On note \hat{x} une solution optimale fractionnaire de P . Le problème de séparation consiste à chercher C telle que l'inégalité associée soit violée par la solution actuelle \hat{x} (*i.e.*, trouver C tel que l'inégalité suivante ne soit pas vérifiée : $\sum_{i \in C} \hat{x} \leq |C| - 1$). Afin d'identifier une inégalité qui soit la plus violée, nous cherchons à minimiser $|C| - \sum_{i \in C} \hat{x}$.
- a) Modéliser le problème de séparation par un programme linéaire en variables 0-1 avec

$$z_i = \begin{cases} 1 & \text{si l'objet } i \text{ est dans la couverture } C \\ 0 & \text{sinon} \end{cases}.$$

Indication : ici \hat{x} est une donnée, pas une variable.

- b) Quelle condition doit vérifier la valeur optimale de ce programme pour que l'inégalité de couverture obtenue soit violée par la solution fractionnaire \hat{x} ?
3. Considérons l'instance de la question 1 de l'exercice 2. Soit \hat{x} la solution de sa relaxation linéaire. Résoudre le problème (pas besoin de branch-and-bound il est facile de trouver son optimum) de séparation et donner l'inégalité de couverture la plus violée par \hat{x} .

Remarque : Le problème étant ici très simple, il n'est pas demandé d'utiliser le simplexe pour le résoudre.

4. Implémentation

Nous allons ici ajouter des coupes de couvertures dans l'algorithme de *branch-and-bound* pour la résolution de problèmes de sac-à-dos. Pour ce faire, nous définissons une nouvelle classe `KnapsackBBNode` (fichier `knapsackBBNode.py`) héritant de `BBNode`.

Cette classe contient deux attributs supplémentaires par rapport à `BBNode` :

- `double[] p` : poids des objets ;
- `double K` : capacité du sac à dos.

- a) La méthode `branch()` de `KnapsackBBNode` sera identique à celle de `BBNode` à l'exception des deux points suivants :

- on essaiera de générer une coupe lorsque la relaxation fournie une solution non entière en appelant simplement la méthode `KnapsackBBNode.generateCut()` (que vous implanterez dans la question suivante) ;
- on branche sur des noeuds de type `KnapsackBBNode` plutôt que sur des noeuds de type `BBNode`.

Définir cette méthode.

- b) Les problèmes de séparation étant résolus à chaque sommet non terminal, il est déconseillé de les résoudre par des programmes linéaires en nombres entiers, qui peuvent être coûteux en temps de calculs. Pour cet exercice, nous utiliserons l'algorithme polynomial approché représenté en Figure 2.

Implémenter cet algorithme dans la méthode `KnapsackBBNode.generateCut()`.

Indication : Vous pourrez utiliser la méthode `addCoverCutToTableau(cover)` afin d'ajouter la contrainte de couverture au tableau. Pour ce faire, il faudra mettre les sommets de la couverture dans un tableau.

Remarque : l'algorithme a pour avantage d'être polynomial mais a comme désavantage de ne pas garantir l'obtention de la coupe la plus violée. Il est également possible que l'algorithme ne trouve aucune couverture bien qu'il en existe.

- c) Ajouter un affichage similaire à celui ci-dessous lorsqu'une coupe est générée :

```
Found cover [1, 2, 3]
Relaxation before the cut : 39,83, (x2, x3, x4) = (1, 1, 0.17)
Relaxation after the cut : 38,50, (x1, x2, x3, x4) = (0.50, 0.50,
1, 0.50)
```

- d) Comparer le nombre de sommets de l'arborescence avec et sans coupes.
- e) Considérer une instance de plus grande taille et évaluer l'amélioration du temps de résolution apportée par l'utilisation de ces coupes.

Exercise 5 Problème de stable

Soit $G = (V, E)$ un graphe non orienté. On appelle *stable* un ensemble de sommets $S \subseteq V$ non adjacents (i.e., un ensemble tel qu'il n'existe aucune arête du graphe reliant deux sommets de S). Le

Données :

n : nombre d'objets

\hat{x} : solution optimale fractionnaire

p : poids des objets

K : capacité du sac-à-dos

$\text{couverture} \leftarrow \emptyset$

$\text{poidsCouverture} \leftarrow 0$

$\text{indice} \leftarrow 0$

↙ Tant qu'une couverture n'a pas été trouvée et que tous les sommets n'ont pas été testés

tant que $\text{poidsCouverture} < K$ **et** $\text{indice} < n$ **faire**

si $\hat{x}[\text{indice}]$ est non nul **alors**

$\text{couverture} \leftarrow \text{couverture} \cup \{\text{indice}\}$

$\text{poidsCouverture} \leftarrow \text{poidsCouverture} + p[\text{indice}]$

$\text{indice} \leftarrow \text{indice} + 1$

↙ Si une couverture a été trouvée

si $\text{poidsCouverture} > K$ **alors**

↙ Utiliser la méthode `addCoverCutToTableau(List)`

 Ajouter la coupe aux contraintes

 Résoudre la relaxation linéaire

FIGURE 2 – Algorithme polynomial de séparation des coupes de couvertures.

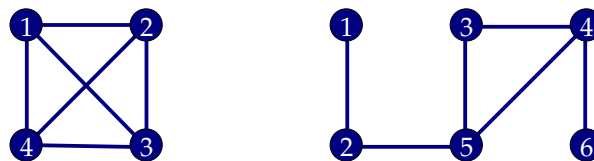


FIGURE 3 – Instances de problème de stable maximal.

problème de stable maximal consiste à identifier un ensemble de stable du graphe qui soit de taille maximale.

1. Proposer une modélisation du problème de stable maximal sous la forme d'un programme linéaire en nombres entiers.
2. Résoudre les deux problèmes de stables représentés en Figure 3 en utilisant votre implémentation de l'algorithme de *branch-and-bound*.

Exercice 6 Problème de production

Vous êtes un fournisseur de voitures qui doit satisfaire chaque jour la demande de ses clients. Pour simplifier, supposons que vous ne produisez qu'un seul type de voitures.

Chaque jour $t \in \{1, 2, \dots, T\}$, vous devez fournir d_t voitures à vos clients. Pour satisfaire la demande du jour t , vous avez la possibilité :

- de produire p_t voitures le jour même (sachant que vous ne pouvez pas en produire plus de 10 par jour); et/ou
- d'utiliser une partie des voitures que vous avez en stock (le stock en fin de journée t , noté s_t , correspond aux voitures que vous avez produits au cours des jours 1 à t et qui n'ont pas encore été livrées pour satisfaire les demandes d_1 à d_t).

Remarque : Vous êtes obligé chaque jour de totalement satisfaire la demande de vos clients.

1. Pour tout $t > 1$, déterminer une équation traduisant le lien entre s_t , p_t , d_t et s_{t-1} .
2. Supposons maintenant que vos entrepôts de stockage soient peu sécurisés et que vous perdiez chaque jour 60% de vos stocks. Adapter l'équation de la question précédente en conséquence.
3. La production et le stockage d'une voiture ont un coût qui varie d'un jour à l'autre. On note :

- c_t^p le coût de production d'une voiture le jour t ;
- c_t^s le coût de stockage d'une voiture entre le jour t et le jour $t + 1$.

Proposer une modélisation de problème permettant de minimiser les coûts sous la forme d'un programme linéaire en nombres entiers.

4. Résoudre l'instance suivante en utilisant votre implémentation de l'algorithme de *branch-and-bound* :
 - $t = 5$;
 - $d_t = \{2, 11, 3, 1, 11\}$;
 - $c_t^p = \{5, 3, 9, 5, 8\}$;
 - $c_t^s = \{7, 1, 9, 4, 3\}$.

Remarque : Vos stocks sont initialement vides.

5. Résoudre à nouveau cette instance en supposant cette fois que vous avez amélioré la sécurité de vos entrepôts et qu'il n'y a plus de perte de stock d'un jour à l'autre. Qu'observer vous et à quoi cela est-il dû ?

Exercice 7

On veut montrer que tout programme quadratique en variables 0-1 peut se ramener à un programme linéaire en 0-1. Pour cela, on va "linéariser" le programme en remplaçant chaque terme quadratique $x_i x_j$ par une seule variable 0-1 : z_{ij} .

1. Donner les contraintes à ajouter pour que le programme linéaire obtenu soit équivalent au programme quadratique initial, c'est-à-dire qu'on a bien $z_{ij} = x_i x_j$ pour tout i et tout j .
2. Une telle transformation est-elle possible si les variables sont continues (dans \mathbb{R}) ?

Exercice 8

On rappelle le modèle du problème de localisation vu en cours :

$$\left\{ \begin{array}{ll} \min z = \sum_{j=1}^n f_j y_j + \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij} & \\ \text{s.c.} & \sum_{j=1}^n x_{ij} = 1 \quad \forall i \quad (1) \\ & x_{ij} \leq y_j \quad \forall i, j \quad (2) \\ & y_i, x_{ij} \in \{0, 1\} \quad \forall i, j \end{array} \right.$$

comprenant les variables

- $x_{ij} = \begin{cases} 1 & \text{si le client } i \text{ est approvisionné par le centre } j \\ 0 & \text{sinon} \end{cases}$
- $y_j = \begin{cases} 1 & \text{si le centre } j \text{ est construit} \\ 0 & \text{sinon} \end{cases}$

1. Donner un modèle équivalent mais plus compact du problème (avec moins de contraintes).
Remarque : Un modèle est dit équivalent s'il contient les mêmes solutions. Dans le cas d'un programme linéaire en nombre entiers, cela signifie que les deux modèles possèdent les mêmes solutions entières (mais pas nécessairement les mêmes solutions continues).
2. On effectue une relaxation continue de ces deux programmes linéaires afin d'obtenir des bornes sur la solution optimale entière. L'une des deux relaxations fournit-elle une meilleure borne que l'autre ? Pourquoi ?

Exercice 9 Dualité Flot/Coupe

Soit un réseau de transport $G = (V, E, C)$ muni d'une source S et d'un puits T , ayant n sommets et m arcs. On veut prouver l'égalité entre une coupe minimale et un flot maximal à l'aide de la dualité

en programmation linéaire. Pour cela on va écrire les deux problèmes sous forme de programmes linéaires duaux. Ces programmes seront obtenus en considérant tous les chemins π_i $i \in \{1, \dots, p\}$ reliant la source S au puits T ; on note f_i le flot routé le long du chemin π_i . Deux chemins sont différents s'ils ont au moins un arc différent. Attention : le nombre de ces chemins peut être très grand. Il est clair que le flot total entre S et T est égal à la somme des flots sur tous les chemins

$F := \sum_{i=1}^p f_i$. Pour le problème du flot maximum, on cherche donc les variables f_i qui maximisent F .

1. Ecrire le programme associé à la recherche d'une coupe minimale. On prendra comme variables : $y_e = 1$ si l'arc $e \in E$ est dans la coupe et $y_e = 0$ sinon, $e \in \{1, \dots, m\}$.
2. Ecrire le programme associé à la recherche d'un flot maximal en considérant les variables f_i et non les flux sur chaque arc comme dans le modèle vu en cours.
3. Vérifier que les relaxations continues de ces programmes sont les mêmes. Peut-on en déduire directement l'égalité des valeurs flot max/coupe min ?

Exercice 10 Recherche de plus courts chemins

Soit un graphe orienté $G = (V, A)$ et deux sommets v_0 et v_n de V . On associe une distance d_{ij} à chaque arc (i, j) de A . On cherche un chemin de v_0 à v_n de plus courte distance (distance = somme des valeurs des arcs empruntés par le chemin).

1. Définir les variables associées à ce problème et l'écrire sous forme d'un PL0-1.
2. On veut maintenant 2 chemins arcs-disjoints (mais pas obligatoirement sommets-disjoints) dont la somme des longueurs est minimale, l'un de v_0 à v_n et l'autre de v_1 à v_{n-1} . Définir les variables et écrire ce nouveau problème sous forme d'un PL0-1.