Optimisation Équitable

Cellier Roxane - Rey Soraya

December 4, 2022

Projet de Modélisation et Optimisation de Graphes, Programmation Linéaire

Année 2022-2023

Contents

Abstract

Ce projet vise à trouver des solutions efficaces, dans un processus de prise de décision, de manière à respecter une équité entre les partis intéressés. On vise donc à optimiser l'équilibre entre la satisfaction des différents points de vue considérés. Ainsi, nous allons nous intéresser à trois problèmes d'optimisation multidimensionnels sur des ensembles X définis par des ensembles de contraintes, dont chaque solution réalisable x est évaluée selon n fonctions modélisant les différents points de vue sur la qualité ou le coût de la solution. Cette évaluation multicritère sera réalisée par un vecteur $z(x) = (z_1(x), ..., z_n(x))$, $z_i(x)$ représentant la performance ou le coût de x selon le point de vue de i, avec n le nombre de points de vue. On cherche donc une solution équitable qui est optimale en ce qu'une amélioration sur une composante de la solution en dégrade forcément une autre. Nous allons dans un premier temps nous intéresser à la modélisation par un programme de ce problème d'optimisation équitable et à sa linéarisation, dans un second temps nous verrons le cas de la décision multicritère, c'est à dire lorsque les points de vue représentent des critères d'évaluation, nous intéresserons ensuite au cas de la décision multi-agents, où les points de vue représentent des évaluations individuelles sur la qualité d'une solution, enfin nous verrons le cas de la décision dans l'incertain où l'on évalue les solutions réalisables selon différents scénarios possibles.

Afin d'obtenir une solution équitable optimale nous allons utiliser une méthode classique de maximisation de la moyenne pondérée ordonnée des composantes $z_i(x)$. Soit une solution $x \in X$ qui maximise la fonction :

$$f(x) = \sum_{i=1}^{n} w_i z_{(i)}(x) \tag{1}$$

où w_i sont des poids positifs et décroissants lorsque i augmente que nous allons fixer selon l'énoncé et $(z_{(1)}(x), ..., z_{(n)}(x))$ représente le résultat d'un tri des composantes de $(z_1(x), ..., z_n(x))$ par ordre croissant.

Exemple 1 :
On cherche à sélectionner un ensemble de trois objets parmi cinq de manière à satisfaire équitablement deux agents sachant que les utilités des objets selon les deux agents sont respectivement $(5, 6, 4, 8, 1)$ et $(3, 8, 6, 2, 5)$. En choisissant le vecteur de pondération $w = (2, 1)$, le problème d'optimisation s'écrira :

$$max\, 2z_{(1)} + z_{(2)}$$

$$\begin{cases} z_1 = 5x_1 + 6x_2 + 4x_3 + 8x_4 + x_5 \\ z_2 = 3x_1 + 8x_2 + 6x_3 + 2x_4 + 5x_5 \\ x_1 + x_2 + x_3 + x_4 + x_5 = 3 \end{cases} \tag{2}$$

$$x_i \in \{0, 1\},\ i = 1, ..., 5$$

Ici notre programme a une fonction objectif qui est de la forme de $f(x)$, or cette fonction n'est pas linéaire. Comme évoqué précédemment nous allons pourtant maximiser cette fonction afin de trouver les solutions équitables optimales selon différents types de représentation de points de vue d'évaluation, nous devons donc procéder à une linéarisation de cette fonction afin de l'appliquer par la suite dans le reste du projet.

1 Linéarisation de $f(x)$

1.1 Démonstration : solution optimale

Pour tout vecteur $z \in \Re$, on note $L(z)$ le vecteur $(L_1(z), ..., L_n(z))$ dont la composante $L_k(z)$ est définie par $L_k(z) = \sum_{i=1}^{k} z_{(i)}$. Soit le programme linéaire en variable binaires suivant, pour k fixé :

$$\begin{gathered} min \sum_{i=1}^{n} a_{ik} z_i \\ \begin{cases} \sum_{i=1}^{n} a_{ik} = k \end{cases} \\ a_{ik} \in \{0,1\}, i = 1, ..., n \end{gathered} \tag{3}$$

Montrons que $L_k(z)$ est la valeur optimale de ce programme linéaire. La fonction objectif est une somme sur les éléments de z_i pondérée par les éléments binaires a_{ik}, qui permettent ainsi une "sélection" des éléments z_i. La contrainte indique que l'on "accepte" dans la somme exactement k éléments de z. On cherche donc à minimiser une somme de k éléments de z.

Par définition, on a pour tout i : $z_{(i)} < z_{(i+1)}$, et on en déduit assez facilement que $L_k(z)$ représente la somme des k plus petits éléments de z.

Ainsi $\forall i, j$ tels que $i \leq k < j$, on a $z_{(i)} < z_{(j)}$. Et donc :

$$L_k(z) = \sum_{l=1, l\neq i}^{k} z_{(l)} + z_{(i)} < \sum_{l=1, l\neq j}^{k} z_{(l)} + z_{(j)}$$

On voit qu'une permutation de deux éléments de part et d'autre de k augmente forcément la valeur de la somme, donc $L_k(z)$ est la somme des k plus petits éléments de z. Instinctivement, elle représente donc la plus petite somme de k éléments de z. On peut donc conclure qu'il s'agit bien de la valeur optimale du programme énoncé ci-dessus.

1.2 Dual du programme relaxé

On admettra que ce programme peut être relaxé en variables continues ce qui fait que $L_k(z)$ est également la valeur à l'optimum du programme linéaire suivant :

$$\begin{gathered} min \sum_{i=1}^{n} a_{ik} z_i \\ \begin{cases} \sum_{i=1}^{n} a_{ik} = k \\ a_{ik} \leq 1 \end{cases} \\ a_{ik} \geq 0, i = 1, ..., n \end{gathered} \tag{4}$$

Écrivons le dual de ce programme relaxé pour $k \in [1, ..., n]$. On notera r_k la variable duale associée à la première contrainte du primal et b_{ik} les variables duales associées au deuxième bloc de contraintes du primal, pour $i = 1, ..., n$.

$$\begin{gathered} max\, k r_k + \sum_{i=1}^{n} b_{ik} \\ \{\ r_k + b_{ik} \leq z_{(i)} \\ b_{ik} \leq 0, i = 1, ..., n \\ \mathrm{r}_k \in \Re \end{gathered} \tag{5}$$

Que l'on peut également réécrire pour correspondre aux formules de l'énoncé :

$$\begin{array}{c} max\, kr_k - \sum_{i=1}^{n} b_{ik} \\ \left\{ \; r_k - b_{ik} \leq z_{(i)} \right. \\ b_{ik} \geq 0, i = 1, ..., n \\ \mathrm{r}_k \in \Re \end{array} \tag{6}$$

1.3 Démonstration

Montrons que $f(x) = \sum_{k=1}^{n} w_k^{'} L_k(z(x))$ avec $w_k^{'} = (w_k - w_{k+1})$ pour $k = 1, ..., n-1$ et $w_n^{'} = w_n$. On notera que le vecteur $w^{'} = (w_1^{'}, ..., w_n^{'})$ a toutes ses composantes positives puisque w a ses composantes strictement décroissantes.

On peut déjà remarquer que :

$$L_1\big(z(x)\big) = z_{(1)}(x) \;\; et \;\; L_k\big(z(x)\big) - L_{k-1}\big(z(x)\big) = z_{(k)}(x) \tag{7}$$

Ainsi,

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} w_i z_{(i)}(x) \\ &= w_1 L_1\big(z(x)\big) + \sum_{i=2}^{n} w_i \Big(L_i\big(z(x)\big) - L_{i-1}\big(z(x)\big) \Big) \\ &= w_1 L_1\big(z(x)\big) + \sum_{i=2}^{n} w_i L_i\big(z(x)\big) - \sum_{i=2}^{n} w_i L_{i-1}\big(z(x)\big) \\ &= w_1 L_1\big(z(x)\big) + w_n L_n\big(z(x)\big) + \sum_{i=2}^{n-1} w_i L_i\big(z(x)\big) - \sum_{i'=1 \; (i'=i-1)}^{n-1} w_{i'+1} L_{i'}\big(z(x)\big) \\ &= w_1 L_1\big(z(x)\big) + w_n L_n\big(z(x)\big) + \sum_{i=2}^{n-1} w_i L_i\big(z(x)\big) - \sum_{i'=2}^{n-1} w_{i'+1} L_{i'}\big(z(x)\big) - w_2 L_1\big(z(x)\big) \\ &= (w_1 - w_2) L_1\big(z(x)\big) + w_n L_n\big(z(x)\big) + \sum_{i=2}^{n-1} (w_i - w_{i+1}) L_i\big(z(x)\big) \\ &= \sum_{i=1}^{n} (w_i - w_{i+1}) L_i\big(z(x)\big) \end{aligned}$$

En considérant $w^{'} = (w_k - w_{k+1})$ et $w_{n+1} = 0$, on a bien $f(x) = \sum_{k=1}^{n} w_k^{'} L_k\big(z(x)\big)$.

1.4 Maximisation de $f(x)$ linéarisée

En utilisant les résultats des démonstrations précédentes, la maximisation de $f(x)$ sur un ensemble X décrite par des contraintes linéaires peut s'écrire sous la forme du programme linéaire suivant :

$$\begin{array}{c} max \sum_{k=1}^{n} w_k^{'} \big(kr_k - \sum_{i=1}^{k} b_{ik} \big) \\ \left\{ \begin{array}{r} r_k - b_{ik} \leq z_i(x), i = 1, ..., n \\ x \in X \end{array} \right. \\ b_{ik} \geq 0, i = 1, ..., n \end{array} \tag{8}$$

Appliquons cette linéarisation sur l'exemple 1 introduit précédemment. Définissons tout d'abord nos variables :

On considère un vecteur de poids : $w = (2, 1)$ soit $w^{'} = (1, 1)$.
On considère la variable binaire $x_i \in [0, 1]$ qui modélise la sélection de l'objet i.
On considère deux agents, soit $n = 2$ et cinq objets, soit .
On considère les vecteurs $z_i(x) = \sum_{j=1}^{5} u_{ij}x_j$, ainsi :

$$z_1(x) = 5x_1 + 6x_2 + 4x_3 + 8x_4 + x_5$$
$$z_2(x) = 3x_1 + 8x_2 + 6x_3 + 2x_4 + 5x_5$$

On considère la sélection de trois objets parmi cinq soit : $\sum_{i=1}^{5} x_i = 3$
On aboutit donc au programme linéaire suivant :

$$\begin{cases} max \sum_{k=1}^{n} w_k^{'}(kr_k - \sum_{i=1}^{k} b_{ik}) \\ \begin{cases} r_k - b_{ik} \leq z_i(x), i = 1, ..., n \\ \sum_{i=1}^{5} x_i = 3 \end{cases} \\ x_l \in \{0, 1\},\ l \in [1, 5],\ i = 1, 2 \end{cases} \tag{9}$$

On peut réécrire le premier groupe de contraintes ainsi : $r_k - b_{ik} - z_i(x) \leq 0$, on obtient donc le programme linéaire suivant :

$$max\, r_1 - (b_{11} + b_{21}) + 2r_2 - (b_{12} + b_{22})$$

$$\begin{cases} \begin{cases} r_1 - b_{11} - (5x_1 + 6x_2 + 4x_3 + 8x_4 + x_5) \leq 0 \\ r_1 - b_{21} - (3x_1 + 8x_2 + 6x_3 + 2x_4 + 5x_5) \leq 0 \\ r_2 - b_{12} - (5x_1 + 6x_2 + 4x_3 + 8x_4 + x_5) \leq 0 \\ r_2 - b_{22} - (3x_1 + 8x_2 + 6x_3 + 2x_4 + 5x_5) \leq 0 \end{cases} \\ x_1 + x_2 + x_3 + x_4 + x_5 = 3 \end{cases} \tag{10}$$

$$x_l \in \{0, 1\}, l \in [1, 5],$$
$$\mathrm{r}_k \in \Re, k \in [1, 2],$$
$$\mathrm{b}_{ik} \geq 0, i \in [1, 2]$$

L'implémentation manuelle par Gurobi donne donc pour les matrice des contraintes et vecteurs de second membre et de fonction objective :

```
# Matrice des contraintes
a = [[1, -1,  0,  0,  0,  0,  -5,  -6,  -4,  -8,  -1],
     [1,  0, -1,  0,  0,  0,  -3,  -8,  -6,  -2,  -5],
     [0,  0,  0,  1, -1,  0,  -5,  -6,  -4,  -8,  -1],
     [0,  0,  0,  1,  0, -1,  -3,  -8,  -6,  -2,  -5],
     [0,  0,  0,  0,  0,  0,   1,   1,   1,   1,   1]]

# Second membre
b = [0,0,0,0, 3]

# Coefficients de la fonction objective
c = [1,-1,-1,2,-1,-1,0,0,0,0,0]
```

Nous obtenons en résultat la sélection des objets $2, 3$ et 4, ce qui donne $f(x) = 50$ pour l'utilité globale sur les deux agents. Cela semble cohérent au vu des utilités de ces objets pour nos individus. À noter que nous n'affichons dans le résultat final uniquement les variables d'affectations nous permettant de déterminer une solution, les variables r_k et b_{ik} n'étant pas utiles pour l'interprétation.

```
Solution optimale:
x_1 = -0.0
x_2 = 1.0
x_3 = 1.0
x_4 = 1.0
x_5 = -0.0

Valeur de la fonction objectif : 50.0
```

Solution optimale pour $w = (2, 1)$

2 Application au partage équitable de biens indivisibles

Dans cette partie, nous considérons un problème où n individus doivent se partager $p \geq n$ objets indivisibles de valeurs connues. Nous définissons la matrice :

$$U = \Big(u_{ij} \Big)_{i \in [\![1,n]\!],\, j \in [\![1,p]\!]} ,$$

où u_{ij} représente l'utilité (ou la valeur) de l'objet j pour l'agent i. Nous définissons également les variables binaires d'affectations $x_{ij} \in \{0,1\}$, où x_{ij} représente l'affectation de l'objet j à l'agent i. Nous supposons que les utilités sont additives par rapport aux objets. Ainsi, en considérant $z_i(x)$ l'utilité de l'ensemble des objets affectés à l'individu i, nous avons :

$$z_i(x) = \sum_{j=1}^{p} u_{ij} x_{ij} \ , \qquad i = 1, \ldots, n$$

Notre but ici est de répartir les p objets de façon équitable entre les n agents, en trouvant une affectation x qui a pour but de maximiser la fonction $f(x)$ définie précédemment, suivant un vecteur poids w à composantes strictement décroissantes.

2.1 Application à un problème donné

Dans un premier temps, nous allons chercher à modéliser le problème présenté dans l'article sous forme d'un programme linéaire, pour pouvoir l'implémenter de façon plus simple dans notre solveur. Nous voulons répartir $p = 6$ objets parmi $n = 3$ individus. Dans ce premier scénario, l'utilité d'un objet représente sa valeur marchande et ne dépends donc pas des agents. Ainsi, pour $i = 1, \ldots, 3$ nous avons :

$$z_i(x) = 325x_{i1} + 225x_{i2} + 210x_{i3} + 115x_{i4} + 75x_{i5} + 50x_{i6}$$

Nous prenons pour commencer un vecteur poids $w = (3, 2, 1)$, ce qui nous donne $w' = (1, 1, 1)$. Nous pouvons également déduire de l'énoncé de l'exemple des contraintes évidentes, telles que le fait que chaque objet ne peut être affecté qu'à une seule personne au maximum, ce qui se traduit mathématiquement, pour $j = 1, \ldots, 6$, par :

$$\sum_{i=1}^{3} x_{ij} \leq 1$$

Si l'on veut que tous nos objets soient affectés, on peut également ajouter que pour $i = 1, \ldots, 3$ et $j = 1, \ldots, 6$:

$$\sum_{i,j} x_{ij} = 6$$

De cette manière, nous pouvons représenter le problème actuel par un programme linéaire de la même forme que celui présenté dans la partie 1.4, en y ajoutant nos contraintes sur les x_{ij}. Nous obtenons ainsi le programme suivant :

$$\max \sum_{k=1}^{3} \Big(k r_k - \sum_{i=1}^{3} b_{ik} \Big)$$

$$\begin{cases} r_k - b_{ik} - z_i(x) & \leq & 0 \\ \sum_{i=1}^{3} x_{ij} & \leq & 1 \\ \sum_{i,j} x_{ij} & = & 6 \end{cases}$$

$$r_k \in \Re,\ x_{ij} \in \{0,1\},\ b_{ik} \geq 0$$
$$i \in [\![1,3]\!],\ j \in [\![1,6]\!],\ k \in [\![1,3]\!]$$

À partir de ce programme, nous pouvons distinguer les différents éléments et les différentes place de nos variables dans nos matrices et vecteurs, nous permettant une implémentation explicite pour une résolution par Gurobi, en suivant le même principe que la partie précédente. Grâce à l'article, nous savons que la solution optimale pour ces données correspond (à une permutation de ligne près) à :

$$x = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \text{ avec pour valeur } f(x) = 3 * 325 + 2 * 335 + 340 = 1\,985.$$

Nous pouvons vérifier que l'exécution du programme explicite nous renvoie ces valeurs. Nous retrouvons en effet l'affectation et l'optimal attendus :

```
Solution optimale:
x_1,1 = 0.0
x_1,2 = 1.0
x_1,3 = 0.0
x_1,4 = 1.0
x_1,5 = 0.0
x_1,6 = 0.0
x_2,1 = 1.0
x_2,2 = 0.0
x_2,3 = 0.0
x_2,4 = 0.0
x_2,5 = 0.0
x_2,6 = 0.0
x_3,1 = 0.0
x_3,2 = 0.0
x_3,3 = 1.0
x_3,4 = 0.0
x_3,5 = 1.0
x_3,6 = 1.0

Valeur de la fonction objectif : 1985.0
```

Solution optimale pour $w = (3, 2, 1)$

Nous voulons maintenant tester la même instance mais avec différents vecteurs de pondération. Nous allons donc cette fois utiliser les vecteurs $w = (10, 7, 1)$ et $w = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. Ce dernier à pour but de maximiser la satisfaction moyenne des individus dans leur globalité. Nous obtenons alors les deux résultats suivants :

```
Solution optimale:
x_1,1 = 0.0
x_1,2 = 1.0
x_1,3 = 0.0
x_1,4 = 1.0
x_1,5 = 0.0
x_1,6 = 0.0
x_2,1 = 1.0
x_2,2 = 0.0
x_2,3 = 0.0
x_2,4 = 0.0
x_2,5 = 0.0
x_2,6 = 0.0
x_3,1 = 0.0
x_3,2 = 0.0
x_3,3 = 1.0
x_3,4 = 0.0
x_3,5 = 1.0
x_3,6 = 1.0

Valeur de la fonction objectif : 5935.0
```

Solution optimale pour $w = (10, 7, 1)$

```
Solution optimale:
x_1,1 = 0.0
x_1,2 = 0.0
x_1,3 = 1.0
x_1,4 = 1.0
x_1,5 = 0.0
x_1,6 = 0.0
x_2,1 = 0.0
x_2,2 = 0.0
x_2,3 = 0.0
x_2,4 = 0.0
x_2,5 = 1.0
x_2,6 = 1.0
x_3,1 = 1.0
x_3,2 = 1.0
x_3,3 = 0.0
x_3,4 = 0.0
x_3,5 = 0.0
x_3,6 = 0.0

Valeur de la fonction objectif : 333.33333349227905
```

Solution optimale pour $w = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$

On peut voir que les deux premières pondérations semblent avoir la même affectation optimale. Le résultat final c'est cependant pas le même, mais cela est logique puisque les poids associés à chaque individu ne sont plus les mêmes. On peut constater que l'affectation n'est plus la même si l'on décide de considérer la satisfaction moyenne globale plutôt que la satisfaction individuelle pondérée.

2.2 Automatisation et Temps d'exécution

Dans cette partie, nous allons chercher à automatiser notre méthode de résolution pour nous permettre d'effectuer de multiples tests, et ainsi nous permettre d'estimer le temps moyen d'exécution de notre programme selon différentes entrées.

Ici, nous allons étudier le temps d'exécution en fonction du nombre d'individus. Pour ce faire, nous

allons considérer successivement des problèmes avec $n = 5, 10, 12$ et 15 individus voulant se partager $p = 5 * n$ objets. La résolution pour $n = 20$ étant extrêmement longue, il ne nous est pas possible de fournir une réponse dans ce cas de figure comme il était demandé, nous avons donc décider d'introduire un $n = 12$ pour avoir au minimum 4 entrées dans notre graphique. Pour chaque couple (n, p), nous allons tirer aléatoirement 10 instances d'utilités U et de vecteurs poids w. Chaque matrice U sera de taille (n, p) et sera composée de valeurs aléatoires tirées entre 0 et 100 (borne choisie arbitrairement). De même, les vecteurs w seront de taille n et calculés à partir de nombres tirés aléatoirement entre 1 et n (arbitraire) dont on aura trié la somme cumulée pour garantir une décroissance stricte. Le w' sera recalculé à chaque fois avant la résolution.

Pour le calcul des matrices et vecteurs, nous allons automatiser leur remplissage en effectuant des généralisations pas blocs. En suivant le même schéma que la résolution explicite, et parce que les contraintes suivront toujours la même forme, nous pouvons ainsi simplifier la construction des matrices de contraintes, et des vecteurs de second membre et de la fonction objectif.

Cette automatisation nous permet donc d'effectuer directement de multiples tests, grâce notamment aux constructions suivantes :

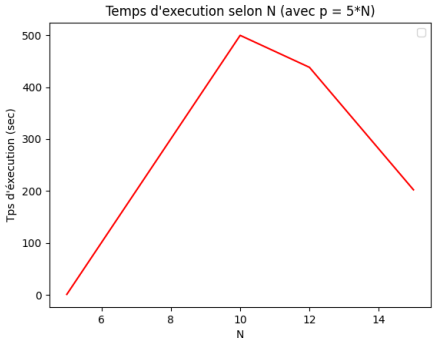
```
# calcul de w' selon le vecteur de poids w
w_prime = [w[i] - w[i+1] for i in range(len(w)-1)]
w_prime.append(w[-1])


## AUTOMATISATION DU MODÈLE SELON LES PARAMÈTRES
# sous matrice représentant les contraintes sur r_k et b_ik
sm_rkbik = np.bmat([np.ones((n,1)), -1*np.eye(n)])
sm_rb = np.kron(np.eye(n), sm_rkbik)

# sous mtrice représentant les contraintes sur les utilités z_i(x)
sm_zi = np.multiply(U, -1)
sm_z = np.kron(np.ones((n,1)), block_diag(*sm_zi))

# sous matrice représentant les contraintes du nombres d'objets affectés
sm_x = np.bmat([[np.kron(np.ones((1,n)), np.eye(p))], [np.ones((1,n*p))]])

# sous matrice pour le remplissage par 0 du reste
sm_zero = np.zeros((p+1, (n+1)*n))

# Assemblage de la matrice des contraintes
a = np.bmat( [[sm_rb, sm_z], [sm_zero, sm_x]] ).tolist()
```

Il est important de noter que, sur des grands jeux de données, l'aléatoire des tirages peut causer que certaines instances soient particulièrement plus longues à résoudre que les autres, pour une même configuration n et p. Nous avons décidé de stopper la résolution pour les instances sur lesquelles notre programme passait plus de 1000 secondes. Il est donc possible que certains de nos résultats se trouvent faussés.

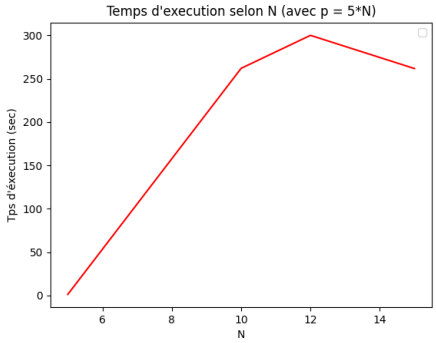
```
124161 23109     cutoff   59      81402.0000 81467.4282  0.08%  61.9  980s
124762 23029     cutoff   62      81402.0000 81466.8851  0.08%  61.9  986s
125310 22883     cutoff   68      81402.0000 81466.3557  0.08%  62.0  992s
125631 22829     cutoff   65      81402.0000 81466.0098  0.08%  62.0  995s
126238 22689 81408.1123   68   92 81402.0000 81465.2981  0.08%  62.1 1001s
126828 22561 81434.4078   64   72 81402.0000 81464.6127  0.08%  62.1 1006s
127463 22466 81409.1948   65   78 81402.0000 81463.9720  0.08%  62.2 1012s
127776 22380     cutoff   64      81402.0000 81463.6620  0.08%  62.2 1015s
```

Temps d'exécution $> 1000sec$ d'une instance avec n=15

Après l'exécution de nos tests, nous nous retrouvons finalement avec les graphiques suivants :



Graphique des temps d'exécution (test 1)



Graphique des temps d'exécution (test 2)

Le pic correspond à des exécutions ayant durées trop longtemps pour nous permettre d'obtenir un résultat. Il semblerai que la résolution dure en moyenne plus longtemps pour des instances de tailles $n = 10, 12$ que pour des instances de tailles plus grande. Cependant cela peut être fortement lié à l'aléatoire et aux tirages de nos matrices d'utilités. Globalement, nous pouvons remarquer malgré tout que le temps d'exécution semble augmenter proportionnellement à la taille de l'instance.

3 Application à la sélection multicritère de projets

Dans cette partie, nous considérons un problème où l'on doit sélectionner p projets dans l'optique de résoudre n objectifs. Nous pouvons définir comme précédemment la matrice U, dont les éléments u_{ij} représentent l'utilité du projet j pour satisfaire l'objectif i. Comme ce n'est plus un problème d'affectation mais simplement d'acceptation, nous pouvons cette fois considérer les variables binaires d'acceptations $x_j \in \{0,1\}$, qui représentent l'acceptation ou non du projet j. Ainsi, en considérant $z_i(x)$ la capacité d'un ensemble de projets à satisfaire l'objectif i, on a :

$$z_i(x) = \sum_{j=1}^{p} u_{ij} x_j$$

La notion de coût est introduite avec les variables c_k, pour $k \in [\![1, p]\!]$, ainsi que la notion de budget maximal qu'on cherchera à ne pas dépasser. Le budget est calculé suivant cette formule :

$$b = \tfrac{1}{2} \sum_{k=1}^{p} c_k$$

On va chercher encore une fois à maximiser la fonction $f(x)$ selon des vecteurs de pondération w à composantes strictement décroissantes. Nous voulons trouver une affectation x générant le meilleur taux de satisfactions, tout en restant dans les limites budgétaires.

3.1 Application à un problème donné

Dans un premier temps, nous allons prendre la main sur l'exemple présenté dans l'article, et tenter de le formuler explicitement sous forme d'un programme linéaire. Nous avons donc $p = 4$ projets à accepter pour satisfaire $n = 2$ chefs de service. Les utilités de chacun des projets sont relatives en fonction du service, nous pouvons donc écrire la matrice d'utilité grâce aux données de l'article :

$$U = \begin{pmatrix} 19 & 6 & 17 & 2 \\ 2 & 11 & 4 & 18 \end{pmatrix}$$

Les contraintes budgétaires nous sont également données, nous avons donc $C = (40k, 50k, 60k, 50k)$ pour un budget maximum de $b = 100k$. Les unités étant les mêmes, nous allons considérer pour ces valeurs leur nombre de milliers. Il faut alors considérer une nouvelle contrainte, puisque le coût des projets acceptés ne doit pas dépasser le budget alloué de l'entreprise. Cette contrainte budgétaire peut se traduire par :

$$\sum_{k=1}^{4} c_k x_k \leq b$$

Enfin pour cette première résolution, nous allons utiliser un vecteur de pondération $w = (2, 1)$, ce qui nous donne $w' = (1, 1)$. Ces définitions suffisent à exprimer notre problématique sous forme d'un programme linéaire à résoudre. Nous obtenons alors le programme suivant en dérivant de celui de la partie 1.4 :

$$\max\ r_1 - (b_{11} + b_{21}) + 2r_2 - (b_{12} + b_{22})$$

$$\begin{cases} r_k - b_{ik} - z_i(x) & \leq & 0 \\ \displaystyle\sum_{k=1}^{4} c_k x_k & \leq & b \end{cases}$$

$$r_k \in \Re,\ x_k \in \{0,1\},\ b_{ik} \geq 0$$
$$i = 1, 2\,,\ k = 1, 2$$

Nous pouvons donc de manière explicite coder les matrices et vecteur nécessaire à la résolution de notre problème pas Gurobi. Nous savons d'après l'article que la solution optimale pour cette instance de coûts et de pondération est $x = (1, 0, 0, 1)$ pour une valeur optimale de $f(x) = 61$. Nous pouvons donc vérifier que le résultat renvoyé par notre solveur est le bon :

```
Solution optimale:
x_1 = 1.0
x_2 = 0.0
x_3 = 0.0
x_4 = 1.0

Valeur de la fonction objectif : 61.0
```

Solution optimale pour $w = (2, 1)$

Nous allons maintenant tester pour des vecteurs de pondération différents. Nous allons rendre $w = (10, 1)$ et $w = (\frac{1}{2}, \frac{1}{2})$. Ce dernier vecteur cherche encore une fois à maximiser la satisfaction moyenne des individus. D'après l'article, la pondération visant la satisfaction moyenne renvoi comme vecteur d'acceptation $x = (1, 0, 1, 0)$ pour une valeur optimale $f(x) = 21$. Nous pouvons observer les résultats renvoyé par Gurobi :

```
Solution optimale:
x_1 = 1.0
x_2 = 0.0
x_3 = 0.0
x_4 = 1.0

Valeur de la fonction objectif : 221.0
```

Solution optimale pour $w = (10, 1)$

```
Solution optimale:
x_1 = 1.0
x_2 = 0.0
x_3 = 1.0
x_4 = 0.0

Valeur de la fonction objectif : 21.0
```

Solution optimale pour $w = (\frac{1}{2}, \frac{1}{2})$

La première pondération ne change pas les acceptations, mais modifie la valeur optimale de la fonction $f(x)$. La seconde pondération en revanche modifie l'affectation comme attendue ainsi que la valeur objective. En calculant les $z_i(x)$, nous pouvons constater une grande disparité entre les satisfactions de chacun des services, malgré une meilleure moyenne globale.

3.2 Automatisation et Temps d'exécution

Dans cette partie, nous allons à nouveau chercher à automatiser notre méthode de résolution pour nous permettre d'effectuer de multiples tests, et nous permettre d'estimer le temps moyen d'exécution de notre programme selon différentes entrées.

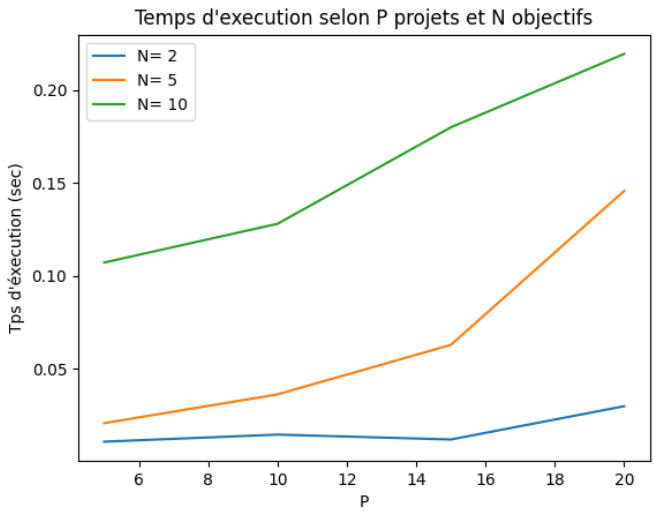
Cette fois cependant, nous allons observer le temps d'exécution selon le nombre de projet en plus du nombre d'objectif à satisfaire. Nous allons donc considérer $n = 2, 5$ et 10 objectifs, et $p = 5, 10, 15$ et 20 projets. Pour chaque n, nous allons calculer un vecteur poids w comme précédemment, qui restera le même pour toutes nos instances avec ce n. Pour chaque couple (n, p), nous allons, en plus de la matrice U des utilités, tirer aléatoirement 10 instances de vecteurs de coûts C de taille p, avec des valeurs choisies arbitrairement entre 0 et 100. Nous allons également calculer selon la formule le budget maximum à ne pas dépasser par l'entreprise.

De la même manière que pour l'affectation d'objets à des individus, nous pouvons automatiser le processus de construction des matrices de contraintes et des vecteurs de second membre et de fonction objectif en utilisant la construction par blocs. Les multiples tests se font alors suivant les boucles :

```python
# boucle de test sur différents n
for n in N:
    # stocke les temps des 10 instances pour ce n
    tps_N = []

    # valeurs de pondération arbitrairement choisies entre 1 et n*n (maximum peu probable)
    # (pour s'assurer une décroissance STRICTE, on fait la somme cumulée des tirages entre 1 et n)
    w = np.cumsum(np.random.randint(1,n, size=n)).tolist()
    # tri dans l'ordre décroissant du vecteur des pondérations
    w.sort(reverse=True)

    # boucle de test sur différents p
    for p in P:
```

```
        # stocke les temps des 10 instances pour ce p
        tps_P = []

        # boucle de test sur 10 instances
        for i in range(10):
            # valeurs d'utilité arbitrairement choisies entre 0 et 100
            U = np.random.randint(100, size=(n,p)).tolist()
            # coûts arbitrairement choisis entre 0 et 100
            C = np.random.randint(100, size=p).tolist()
            # calcul du budget selon les coûts
            B = sum(C) /2

            # execution et calcul du temps
            debut = time.time()
            resolutionCoutBudget(n,p,U,C,B,w)
            fin = time.time()
            tps_P.append(fin - debut)

        # calcul et stockage du temps moyen pour p
        tps_N.append(np.mean(tps_P))

    # Affichage et sauvegarde des temps d'execution pour une meilleure interprétation
    plt.plot(P, tps_N, label="N= %i" %n)
```

L'exécution de nos tests nous permets de tracer le graphique suivant :



Graphique des temps d'exécution

On constate assez intuitivement que l'augmentation du nombre d'objectifs et du nombre de projets allonge le temps d'exécution de la résolution. Cependant les résolutions restent encore très rapides pour les tailles d'instances testées.

4 Application à la recherche d'un chemin robuste dans un graphe

On se place maintenant dans un contexte de recherche d'itinéraires dans l'incertain. Dans un réseau modélisé par un graphe orienté, on recherche un chemin rapide pour aller d'un sommet initial à un sommet destination. Le problème est compliqué par le fait que plusieurs scénarios sont envisagés sur les temps de transport dans ce réseau. Plus précisément, on considère un ensemble $S = \{1, ..., n\}$ de scénarios possibles, et le temps pour parcourir chaque arc (i, j) du graphe est donné par le vecteur $(t^1_{i,j}, ..., t^n_{i,j})$ où $t^s_{i,j}$ représente le temps nécessaire pour parcourir l'arc (i, j) dans le scénario s donné, pour tout $s \in S$. Les temps sont additifs le long d'un chemin, ce qui signifie que le temps pour parcourir un chemin P dans le scénario s est défini par :

$$t^s(P) = \sum_{(i,j)\in P} t^s_{i,j}.$$

4.1 Le chemin le plus rapide

Nous avons formulé un programme linéaire qui, étant donné un graphe orienté G, permette de calculer le chemin le plus rapide du sommet initial au sommet de destination dans un scénario s donné. Pour cela, nous avons modélisé notre problème comme un problème de flot à coût minimum. Par la suite nous noterons l'ensemble des arêtes d'un graphe "E" et l'ensemble des sommets "V".

Définition des variables :

- x_{ij}: variable binaire, elle vaut 1 si je prend l'arc (i, j), 0 sinon
- $c_{i,j}$: coût de l'arc (i, j)

Définition de la fonction objectif :

$$min \sum_{(i,j)\in E} c_{i,j} x_{ij}$$

En effet, nous cherchons un chemin, nous cherchons donc à sélectionner des arcs (variables x_{ij}), et nous cherchons le chemin le plus rapide, soit le flot de coût minimum. On minimise la somme des coûts $c_{i,j}$ des arcs sélectionnés dans le chemin.

Définition des contraintes :
Nous sommes sur un problème de flot, il y a donc la contrainte de conservation de flot. La différence de la somme des arcs entrant et des arc sortants d'un sommet est égale à 1 si le sommet en question est le sommet de départ ou d'arrivée, et 0 sinon car nous sommes en variables binaires. Soit :

$$\sum x_{ij} - \sum x_{ji} = \begin{cases} 1 \iff i = a \\ 0 \iff i \neq a, g \\ 1 \iff i = g \end{cases} \tag{11}$$

Nous cherchons un chemin, il doit donc être continu. On pose donc la contrainte qu'il ne peut y avoir d'arc entrant sans arc sortant. Soit :

$$\sum x_{ij} \leq \sum x_{ki} | (k, i) \in E \tag{12}$$

Nous avons également la contrainte de visiter au plus une fois chaque sommets i, on ne peut pas avoir plusieurs arcs sortants pour un sommet donné car cela ne respecterai pas la conservation du flot en variables binaires. Soit :

$$\forall i, \sum x_{ki} \leq 1 \tag{13}$$

Enfin, nous avons la contrainte d'atteindre notre destination : le sommet g, ce qui se traduit par "la somme des arcs entrants dans g est égale à 1", soit " j'ai au moins un arc qui atteint g dans mon chemin". Soit :

$$\sum x_{ig} = 1 \tag{14}$$

On obtient donc le programme linéaire suivant :

$$
\begin{aligned}
& min \sum_{(i,j)\in E} c_{i,j}x_{ij} \\
& \begin{cases} \forall i \sum x_{ki} \leq 1 \\ \sum x_{ig} = 1 \\ \sum x_{ig} = 1 \end{cases} \qquad (15) \\
& x_{ij} \in \{0,1\},\ (i,j) \in E \\
& c_{ij} \in \Re,\ (i,j) \in E
\end{aligned}
$$

Nous avons appliqué ce programme linéaire au graphe de la figure 1 ci-dessous qui distingue deux scénarios, soit $S = \{1, 2\}$. Les vecteurs coûts étiquetants les arcs représentent les temps de parcours respectivement dans les scénarios 1 et 2. Nous nous intéressons au chemin le plus rapide depuis le sommet a vers le sommet g dans chacun des deux scénarios.

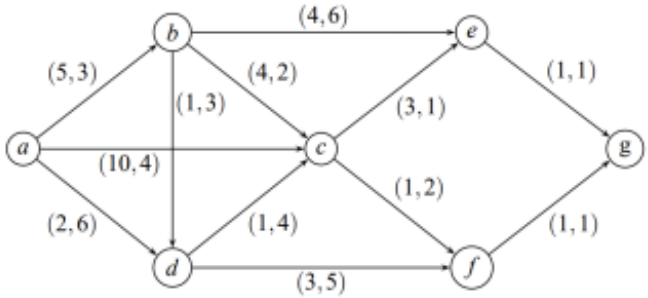


Figure 1: Une instance du problème de chemin robuste à 2 scnénarios

Implémentation gurobi :

```
def solve(G):

    V,E = G

    m = Model("mogplex")

    # déclaration des variables de décision
    x = {}
    for (i,j,_) in E:
        x[(i,j)] = m.addVar(vtype=GRB.BINARY, name = "x%s%s"%(i,j))

    c = {}
    for (i,j,k) in E:
        c[(i,j)] = k

    m.update()

    # définition de l'objectif

    obj = LinExpr()
    obj = 0
    for (i,j,_) in E:
        obj += c[(i,j)] * x[(i,j)]
```

```
m.setObjective(obj,GRB.MINIMIZE)

# définition des contraintes

for(i,j,_) in E :
    if i != 1 :
        preds = [(i2,j2) for (i2,j2,c2) in E if(j2==i)]
        m.addConstr(x[(i,j)]<=quicksum([x[ind] for ind in preds]),"Contrainte conservation %s"%

for i in V:
    if i != 1 :
        preds = [(i2,j2) for (i2,j2,c2) in E if(j2==i)]
        m.addConstr(quicksum([x[ind] for ind in preds])<=1,"Contrainte unique %s"%(i))

preds = [(i2,j2) for (i2,j2,c2) in E if j2==7]
m.addConstr(quicksum([x[ind] for ind in preds])==1, "Contrainte atteint sommet g")
```

Nous avons obtenu des résultats similaires à ce que nous avons calculé à la main, à savoir :

- scénario 1 : chemin optimal : $A \Rightarrow D \Rightarrow C \Rightarrow F \Rightarrow G$, coût du chemin : 5
- scénario 2 : chemin optimal : $A \Rightarrow C \Rightarrow E \Rightarrow G$, coût du chemin : 6

4.2 Chemin robuste

Ne sachant pas quel scénario va se produire et ne connaissant même pas leurs probabilités, nous allons chercher un chemin robuste, c'est à dire qui reste rapide dans les différents scénarios. Pour cela on cherche un chemin qui maximise $v_f(P) = f\big(-t^1(P), ..., -t^n(P)\big)$. En d'autres mots, nous allons maximiser la somme des temps de trajets négatifs pondérée par les poids des scénarios, représentant leurs probabilités. Cela revient toujours à calculer le chemin le plus rapide car le minimum de la somme des temps de trajets revient au maximum des temps de trajets négatifs.

Définition de la fonction objectif :

$$max \sum_{(i,j)\in E} (-t_{i,j} w_s) x_{i,j,n}$$

On s'aperçoit que l'on retrouve une fonction à maximiser de la forme de $f(x)$:

$$f(x) = \sum_{i=1}^{n} w_i z_{(i)}(x) \tag{16}$$

On va donc devoir effectuer un linéarisation de notre programme selon les même modalités que précédemment pour $f(x)$, à savoir :

$$\begin{cases} max \sum_{k=1}^{n} w_k^{'}(kr_k - \sum_{i=1}^{k} b_{ik}) \\ r_k - b_{ik} \leq z_i(x), i = 1, ..., n \\ x \in X \end{cases} \tag{17}$$

Appliquons notre programme linéarisé à l'instance de la figure 1, afin de trouver un chemin robuste du sommet a au sommet g.

Définition des variables :

- n = 2 scénarios

- w = (2,1) soit $w^{'}$ = (1,1)
- $z_i = \sum_{j=1}^{12} u_{ij} x_j$
- x_{ij} variable binaire qui vaut 1 si l'on prend l'arc (i, j) dans le chemin optimal, et 0 sinon. Nous avons 12 arcs, donc 12 variables x_{ij}.

En appliquant nos valeur numériques, cela donne :

$z_1(x) = 5x_1 + x_2 + 2x_3 + 4x_4 + 4x_5 + x_{6+x_7} + 3x_8 + 3x_9 + x_{10} + x_{11} + x_{12}$
$z_2(x) = 3x_1 + 3x_2 + 6x_3 + 6x_4 + 2x_5 + x_{6+4x_7} + 5x_8 + x_9 + 2x_{10} + x_{11} + x_{12}$

Définition des contraintes :

Nous avons dans un premier temps les contraintes sur les arcs afin de respecter le principe de conservation du flot et
d'obtenir un chemin continu depuis le sommet a jusqu'au sommet g. On peut les modéliser comme suit :

$$
\begin{gathered}
x_{ab} + x_{ac} + x_{ad} = 1 \\
x_{be} + x_{bc} + x_{bd} \leq 1 \\
x_{dc} + x_{df} \leq 1 \\
x_{ce} + x_{cf} \leq 1 \\
x_{ab} - x_{be} - x_{bc} - x_{bd} = 0 \\
x_{ad} + x_{bd} - x_{dc} - x_{df} = 0 \\
x_{bc} + x_{ac} + x_{dc} - x_{ce} - x_{cf} = 0 \\
x_{be} + x_{ce} - x_{eg} = 0 \\
x_{cf} + x_{df} - x_{fg} = 0 \\
x_{eg} + x_{fg} = 1
\end{gathered}
$$

Dans un second temps nous avons nos contraintes sur nos variables duales, r_k et b_{ik}, que l'on peut réécrire : $r_k - b_{ik} + z_i(x) \leq 0$.

Implémentation gurobi :

```
# Matrice des contraintes

a = [[1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0], # contraintes sur arcs
     [0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0],
     [1,0,0,-1,-1,-1,0,0,0,0,0,0,0,0,0,0,0,0],
     [0,0,1,0,0,1,-1,0,-1,-1,0,0,0,0,0,0,0,0],
     [0,1,0,0,1,0,1,0,-1,-1,0,0,0,0,0,0,0,0],
     [0,0,0,1,0,0,0,0,1,0,-1,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,1,0,1,0,-1,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0],
     [5,1,2,4,4,1,1,3,3,1,1,1,1,-1,0,0,0,0], # r1 z1
     [5,1,2,4,4,1,1,3,3,1,1,1,1,0,-1,0,0,0], # r1 z1
     [3,3,6,6,2,1,4,5,1,2,1,1,0,0,0,1,-1,0], # r2 Z2
     [3,3,6,6,2,1,4,5,1,2,1,1,0,0,0,1,0,-1]] # r2 z2

# Second membre
b = [1,1,1,1,0,0,0,0,0,1,0,0]
```

```
# Coefficients de la fonction objective
c = [0,0,0,0,0,0,0,0,0,0,0,0,1, -1, -1, 2, -1, -1]

# Création de modèle
m = Model("mogplex")

# Déclaration variables de décision
x = []
for i in colonnes:
    # les rk sont réels non bornés
    if i in colonnes_rk:
        x.append(m.addVar(vtype=GRB.CONTINUOUS, lb=-GRB.INFINITY, name="r%d" % (i + 1)))

    # les bik sont supérieurs ou égaux à 0
    if i in colonnes_bik:
        x.append(m.addVar(vtype=GRB.CONTINUOUS, lb=0, name="b%d" % (i + 1)))

    # les xi sont binaires (1 ou 0)
    if i in colonnes_x:
        x.append(m.addVar(vtype=GRB.BINARY, name="x%d" % (i + 1)))
# MAJ du modèle pour integrer les nouvelles variables
m.update()
obj = LinExpr();
obj = 0
for j in colonnes:
    obj += c[j] * x[j]

# Définition de l'objectif (maximisation de la fonction objectif)
m.setObjective(obj, GRB.MAXIMIZE)

# Définition des contraintes
for i in lignes_inf:
    m.addConstr(quicksum(a[i][j] * x[j] for j in colonnes) <= b[i], "Contrainte%d" % i)
for i in lignes_egal:
    m.addConstr(quicksum(a[i][j] * x[j] for j in colonnes) == b[i], "Contrainte%d" % i)
```

Nous avons obtenu des résultats qui semblent cohérent car le chemin est effet robuste, il a la même valeur dans les deux scénario cependant nous n'avons pas pu vérifier à la main s'il s'agissait bien du chemin optimal:
chemin optimal : $A \Rightarrow B \Rightarrow E \Rightarrow G$
valeur du chemin optimal : 10.

4.3 Étude de l'impact de la pondération des scénarios sur la robustesse du chemin optimal

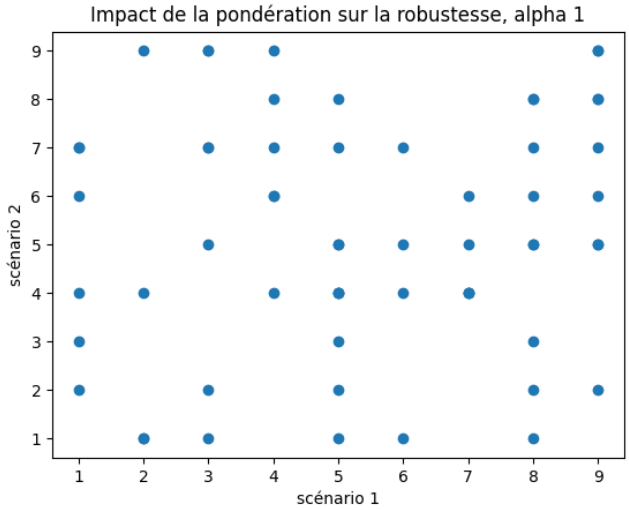
On souhaite étudier l'impact de la pondération w sur la robustesse de la solution proposée. Pour cela on utilise une famille de vecteurs de pondération $w_\alpha \in \Re^n$ définie pour tout $\alpha \geq 1$ par les poids :

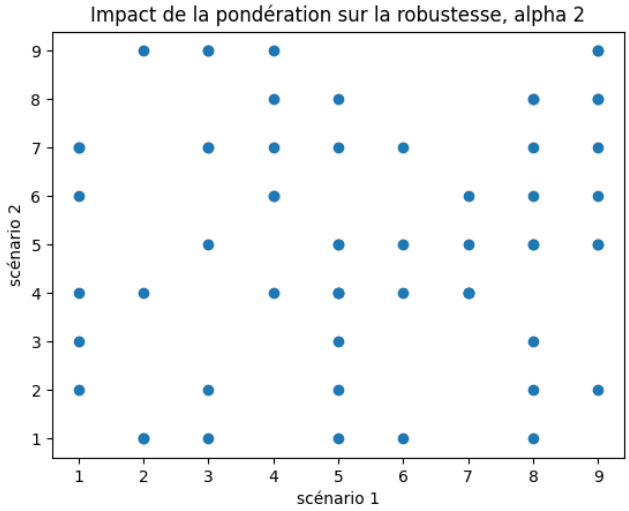
$$w_i(\alpha) = \left(\frac{n-i+1}{n}\right)^\alpha - \left(\frac{n-i}{n}\right)^\alpha, \; i = 1, ..., n \tag{18}$$

Pour simplifier on se placera dans le cas $n = 2$ et on tirera aléatoirement 20 fois des temps de parcours t^1_{ij} et t^2_{ij} pour les arcs (i, j) du graphe de la figure 1. Pour chacune des instances générées, nous avons calculé le chemin qui maximise v_f pour les vecteur $w(1), w(2), w(3), w(4), w(5)$ et nous avons représenté les 20 solutions ainsi obtenues dans le plan (t^1, t^2).

Pour ce faire nous avons généralisé notre algorithme de la question précédente en définissant des fonctions auxquelles nous faisons appel dans une double boucle pour les alpha et pour les 20 itérations par alpha.

Nos résultats d'implémentation pour cette question semblent laisser paraître que notre algorithme calcule le même chemin pour chaque alpha et ne montre pas d'effet de la pondération sur la robustesse, mais nous pouvons visualiser à travers nos résultats que notre algorithme ne fonctionne pas bien.





Nous avons donc tenter de repenser notre programme linéaire et nous avons écris une nouvelle formulation, nous avons essentiellement changer la manière dont nous écrivons nos contraintes. Notre matrice des contraintes ne contient que les contraintes liées à la conservation du flot et au fait que l'on part du sommet a pour arriver au sommet g. La contrainte restante sur les variables du programme linéarisé, nous l'avons ajouté manuellement cette fois sans faire passer les variables z de l'autre côté de l'inégalité. Nous avons également modifié notre manière de construire notre boucle en gérant l'affichage séparément.

Implémentation gurobi :

```
dico_arcs ={'x_1':'AB', 'x_2':'AC', 'x_3':'AD', 'x_4':'BE', 'x_5':'BC','x_6':'BD', 'x_7':'DC', 'x_8
            'x_11':'EG', 'x_12':'FG'}

# matrice des contraintes sur les arcs

a = [[1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],  # contraintes sur arcs
     [0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0],
     [1, 0, 0,-1,-1,-1, 0, 0, 0, 0, 0, 0],
     [0, 0, 1, 0, 0, 1,-1, 0,-1,-1, 0, 0],
     [0, 1, 0, 0, 1, 0, 1, 0,-1,-1, 0, 0],
     [0, 0, 0, 1, 0, 0, 0, 0, 1, 0,-1, 0],
     [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0,-1],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]]

# Explicitation des lignes pour les signes des contraintes
lignes_egal = [0, 4, 5, 6, 7, 8, 9]
lignes_inf = [1, 2, 3]
colonnes = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11]
# Second membre
bsecond = [1, 1, 1, 1, 0, 0, 0, 0, 0, 1]

def temps_trajets_aleatoires():
    t = []
    for i in range(12):
        t.append([int((rand.random() * 100) % 20)])
    return t

# définition des paramètres :
n = 2
iteration = 20
v = 7 # nombre de sommets
w_stockage = []
# initialisation des résultats
t1 = []
t2 = []

for alpha in range(1,6):
    w = np.zeros(n) #initialise les poids
    for i in range(n):
        w[i] = (n-(i+1)+1/n)**alpha - (n-(i+1)/n)**alpha
    w_prim = [w[i] - w[i + 1] for i in range(len(w) - 1)]
    w_prim.append(w[n - 1])
    w_stockage.append(w)
    t1_it = []
    t2_it = []
    for i in range(iteration):
        cout1 = temps_trajets_aleatoires()
        cout2 = temps_trajets_aleatoires()
        cout = np.array([cout1,cout2])
        range_sommets = v
        range_scenarios = n
        m = Model("moglpex")
```

```
        x = []
        # déclaration des variables
        for i in range(12):
            x.append(m.addVar(vtype = GRB.BINARY, name="x%d"%(i)))
        for s in range(range_scenarios):
            z = np.array([m.addVar(vtype=GRB.CONTINUOUS,lb=0,name="z%d"%(i+1))])
            r = np.array([m.addVar(vtype=GRB.CONTINUOUS, lb=-GRB.INFINITY, name="r%d" % (i + 1))])
            for k in range(range_scenarios):
                b = np.array([m.addVar(vtype=GRB.CONTINUOUS,lb=0,name="b%d_%d"%(s+1,k+1))])
        m.update()
        #définition des contraintes sur les arcs
        # on part de a
        # on arrive à g
        # somme des arc entrant - somme arcs sortant = 0
        for i in lignes_inf:
            m.addConstr(quicksum(a[i][j] * x[j] for j in colonnes) <= bsecond[i], "Contrainte d'inf
        for i in lignes_egal:
            m.addConstr(quicksum(a[i][j] * x[j] for j in colonnes) == bsecond[i], "Contrainte d'éga
        #définition des contraintes du PL :
        for s in range(range_scenarios):
            vari = 0
            for i in range(12):
                vari += quicksum([cout[s][i]*x[i]])
            m.addConstr(z[s] == vari, "définition de z")
        for k in range(range_scenarios) :
            for i in range(range_scenarios) :
                m.addConstr(r[k]-b[i][k]<=-z[i],"contrainte de mon PL pour des temps de trajets nég
        obj = LinExpr();
        obj = 0
        for k in range(range_scenarios):
            obj += w_prim[k] * ((k + 1) * r[k] - quicksum(b[i][k] for i in range(range_scenarios)))

        m.setObjective(obj, GRB.MAXIMIZE)
        m.optimize()
        # récupération du résultat pour l'itération
        vari = []
        for i in range(range_sommets):
            vari.append([x[i, j].x if isinstance(x[i, j], gurobipy.Var) else 0 for j in range(range
        x = np.array(vari)
        t1_it.append(z[0].x)
        t2_it.append(z[1].x)
    # récupération du résultat pour toutes les itérations
    t1.append(t1_it)
    t2.append(t2_it)
```