Matching Mechanisms for Kidney Exchange

Question 1 L'algorithme de donation directe ne prend pas en compte la liste des préférences des patients. On peut donc utiliser directement la matrice d'adjacence pour tester en temps constant la propriété "$k_i \in K_i$". En voici le code :

Algorithm 1: Donation directe

Data: matrice d'adjacence K telle que $K_{i,j}$ vaut 1 si $k_i \in K_j$ et 0 sinon
Result: Une liste l telle que $l[i]$ contienne le rein associé au patient i.
$l = [w, ..., w]$;
for $i = 1$ ***to*** n **do**
  if $K_{i,i} = 1$ **then**
    $l[i] \leftarrow i$
  else
end

Cf. notebook pour l'implémentation et le résultat.

Question 2 L'algorithme glouton consiste à parcourir l'ensemble des patients par ordre décroissant de priorité (donc ici en partant du premier patient vu que les patients sont déja triés par ordre de priorité). Pour chaque paire (k_i, t_i) que l'on considère, on affecte le rein k_p le plus haut placé dans la liste des préférences de t_i encore non affecté et tel que l'arrête $(k_i, t_i), (k_p, t_p)$ existe dans le graphe.

Algorithm 2: Algorithme glouton

Data: matrice d'adjacence K et une liste P des listes préférence ($P[i]$ est la liste des reins que veut recevoir le patient t_i, par ordre de préférence.
Result: Une liste l telle que $l[i]$ contienne le rein associé au patient i.
$l = [w, ..., w]$;
for $i = 1$ ***to*** n **do**
  $j \leftarrow 0$ (indice qui parcourt la liste des préférences $P[i]$
  while $k_i \notin K_{P[i][j]}$ *ou* $k_{P[i][j]}$ *déjà assigné* **do**
    $j \leftarrow j + 1$
  end
  $l[i] \leftarrow P[i][j]$ (ou bien w si j est trop grand)
end

Cf. notebook pour l'implémentation et le résultat.

Question 3 On suppose qu'il n'y a pas de cycle. Considérons une paire (k_i, t_i). Lorsque l'on construit la liste l des successeurs de (k_i, t_i), à chaque étape, on ajoute un élément qui n'est pas déja présent dans l (sinon on crée un cycle). Comme le nombre de sommets est fini on doit nécessairement ajouter w avant d'épuiser l'ensemble des sommets disponibles. Ainsi, (k_i, t_i) est bien la queue d'une w-chaîne, ce qui conclut la démonstration.

Question 4 *Cf. notebook.*

Question 5 *Cf. notebook.*

Question 6

Notations :

— Lorsque $V' \subset V$, on traduit par "assignation sur V'" un matching ne mettant en jeu que des paires patients-donneurs de V'

— On dit que le rein i est **affecté provisoirement** au patient j si j pointe vers i et que i est encore disponible dans le graphe. On dit que le rein i est **affecté définitivement** au patient j si i a été retiré du graphe au cours de l'exécution de l'algorithme et que c'est à j qu'il est donné.

— A l'étape i de l'algorithme on note F_i l'ensemble des reins affectés définitivement et et P_i l'ensemble des reins affectés provisoirement. Au cours de l'exécution de l'algorithme, plusieurs patients peuvent donc être affectés provisoirement au même rein. Si l'algorithme termine, on doit avoir à la fin $F = V$ et $P = \emptyset$.

— A l'étape i de l'algorithme on note $r_i = (r_{i,1}, ..., r_{i,n})$ la liste telle que $\forall j \in [|1, n|]$, $r_{i,j}$ est le rang du rein affecté provisoirement ou définitivement au patient j dans sa liste de préférences

— Si l_1 et l_2 sont deux listes de même longueur on note $l_1 < l_2$ pour signifier que tous les éléments de l_2 sont terme à terme $\geq$ aux éléments de l_1 et qu'il existe i_0 vérifiant $l_1[i_0] < l_2[i_0]$

Lemme 1 : L'algorithme de sélection des cycles et w-chaînes (peut importe la règle) termine

Preuve : A chaque étape, le cadrinal de F augmente strictement (soit après avoir ajouté un cycle, soit après avoir affecté les reins le long d'une w-chaîne). On finit donc par aboutir à la situation où $F = V$ ce qui est la fin de l'algorithme.

Lemme 2 : A chaque étape i de l'éxecution de l'algorithme, les sommets déjà assignés le sont de manière efficiente. Autrement dit, il n'existe pas d'autre assignation sur F_i associé à une liste de rangs $\rho := (\rho_k)_{k \in F_i}$ vérifiant $\rho < (r_{i,k})_{k \in F_i}$

Preuve :

— Ceci est vrai au départ puisque qu'aucun sommet n'est assigné et l'assignation vide est efficiente

— Supposososns que la propriété est vraie à l'étape i. Montrons qu'elle l'est encore à l'étape $i + 1$. Considérons S_i l'ensemble des sommets que l'on ajoute à l'étape i, c'est à dire $F_{i+1} = F_i \cup S_i$. On raisonne par l'absurde et on supppose qu'il existe une meilleure assignation M^* sur F_{i+1} (i.e associé à une liste de rangs $\rho := (\rho_k)_{k \in F_{i+1}}$ vérifiant $\rho < r_{F_{i+1}} := (r_{i,k})_{k \in F_{i+1}}$).

1. Cette assignation ne peut relier un patient t de F_i à un rein k de S_i. En effet, si le patient t préférait le rein de t', il l'aurait reçu plus tôt car il est ajouté à F avant t et lors de son ajout, le rein de t était encore disponible. On aurait donc une contradiction avec la condition $\rho < r_{F_{i+1}}$.
2. Vu le point 1., 2 cas se présentent :
   Cas 1 : M^* est l'union d'une assignation S^* sur S_i et d'une autre F^* sur F_i. (en d'autres termes M^* garde les ensembles S_i et F_i séparés). 3 cas se présentent
   1.1 S_i correspond à l'ajout d'un cycle. Comme l'assignation sur F_i est déjà optimale cela signifique que c'est S^* qui est strictement meilleure que le cycle. On constate aisément qu'un cycle à p sommets constitute déja une assignation efficiente sur ces p sommets ce qui contredit la phrase précédente.

1.2 S_i correspond à l'ajout d'une w-chaîne indépendante de F_i. Le raisonnement est le même qu'avec le cycle : une w-chaîne à p sommets est déja une assignation efficiente sur ces p sommets.

1.3 S_i correspond à l'ajout de sommets se "raccordant" au bout d'une w-chaîne de F_i. On note t le patient de S_i relié à un rein k de F_i. Comme S^* et F^* sont deux assignations séparées, cela signifie que t préfère être relié à un rein de S_i ce qui est absurde car au moment de l'ajout, les reins de S_i sont encore disponibles.

Cas 2 : M^* relie un patient t de de S_i à un rein k de F_i. Dans ce cas, le patient t' initialement relié à k dans F_i doit préférer être relié à un autre rein k' de F_i car les liaisons entre F_i et S_i sont interdites (cf 1). 2 cas se présentent

2.1 Initialement, le rein k' est disponible (c'est la "tête" d'une w-chaîne) et dans ce cas on aurait une contradiction avec l'hypothèse de récurrence (si le rein k' était disponible t l'aurait reçu plus tôt car d'après l'hypothèse de récurrence, l'assignation sur F_i est optimale)

2.2 le rein k' n'était pas disponible dans l'assignation initiale. On note alors t'' le patient initialement relié à k' et on construit ainsi une suite $t^{(n)}$ de patients qui finit par aboutir au cas 2.1 ou à la création d'une liaison entre F_i et S_i ce qui est interdit (cf. 1)

Dans tous les cas, l'hypothèse d'existence d'une meilleure assignation M^* conduit à une absurdité. On en déduit que la propriété d'efficience des l'assignation sur F reste vraie à l'étape $i+1$

— Conclusion : On a montré par récurrence que le lemme 2 est vrai

Preuve du théorème 1 : Cela découle des lemmes 1 et 2 : au moment ou l'algorithme temrine (mettons à l'étape N), on a une assignation efficiente sur F_N or $F_N = V$.

Question 7 Si le patient 12 change unilatéralement sa liste de choix en mettant $k_8 \succ k_9 \succ k_3 \succ k_{11} \succ k_{12} \succ k_{10}$ au lieu de sa réelle liste de préférences ($k_{11} \succ k_3 \succ k_9 \succ k_8 \succ k_{10} \succ k_{12}$), l'éxecution de l'algorithme avec la règle de choix A lui affecte le rein k_8. Avec une liste de préférences hônnete, le patient 12 obtient le rein k_{10}. Dans la liste originale, $k_8 \succ k_{10}$ ce qui montre que l'algorithme avec la règle de choix A n'est pas "strategy proof"

Question 8

— Par exemple, si on tente de changer l'ordre de préférence des reins du patient 12, on constate que pour toutes les permutations possibles de la liste de préférences, on n'obtient pas une meilleure assignation que k_8. On comprend intuitivement que la règle de choix B permet d'être "strategy-proof" : en basant la règle de choix sur la longueur de la w chaîne, on laisse plus de liberté à chacun pour pouvoir ajuster son choix de manière à s'insérer dans une longue w-chaîne.

— *Cf. le notebook l'execution de cet exemple.*

Question 9 *Cf. notebook.*

Question 10 En notant MIP l'ensemble des chemins minimaux infaisables, N son cardinal, on peut proposer la formulation suivante :

$$\text{maximiser} \quad \sum_{(i,j)\in E} x_{i,j}$$

$$\text{avec} \quad 1. \sum_{(i,k)\in E} x_{i,k} = \sum_{(k,j)\in E} x_{k,j}, \qquad \forall k \in V$$

$$2. \quad x_{i,j} \in \{0,1\}, \qquad \forall (i,j) \in V^2$$

$$3. \quad \sum_{(i,k)\in E} x_{i,k} \leq 1 \qquad \forall k \in V$$

$$4. \quad \sum_{(i,j)\in p} x_{i,j} \leq K \qquad \forall p \in MIP$$

$$5. \quad x_{i,j} = 0 \text{ si } i \text{ et } j \text{ incompatibles}$$

La condition 1 traduit le fait que l'on ne considère que les cycles (somme des dergés entrants = somme des degrés sortants. La condition 2 est la condition d'intégralité que l'on va relaxer dans l'approche "Branch and Bound". La condition 3 traduit le fait qu'un rein ne peut être assigné à plusieurs patients . La condition 4 est le fait que l'on n'accepte pas les chemins infaisables. Afin d'implémenter la méthode BB à la question suivante, il est utile de reformuler ce problème ILP matriciellement : Si on pose :

— $z^T := (1, ..., 1) \in \mathbb{R}^{n^2}$

— $L \in \mathcal{M}_{n,n^2}(\mathbb{R})$ définie par $L_{k,(i,j)} = \mathbb{1}_{i=k}$

— $C \in \mathcal{M}_{n,n^2}(\mathbb{R})$ définie par $C_{k,(i,j)} = \mathbb{1}_{j=k}$

— IMP la matrice de taille $N \times n^2$ telle que chaque ligne représente un chemin minimal infaisable. A la ligne $k \in [|1, N|]$, $\text{IMP}_{k,(i,j)}$ vaut 1 si (i, j) est dans le chemin k et 0 sinon

— $b^T := (K, ..., K) \in \mathbb{R}^N$

— $H^T = (\mathbb{1}_{i,j \text{ non compatibles}})_{1\leq(i-1)n+j\leq n^2} \in \mathbb{R}^{n^2}$

— $X := (x_{1,1}, x_{1,2}, ..., x_{n,n})^T$ l'inconnue du problème

On a alors la formulation suivante :

$$\text{maximiser} \quad z^T X$$

$$\text{avec} \quad 1. \quad (L - C)X = 0$$
$$2. \quad X \in \{0,1\}^{n^2}$$
$$3. \quad CX \leq 1$$
$$4. \quad \text{IMP}X \leq b$$
$$5. \quad H^T X = 0$$

Question 11
Par manque de temps, nous ne donnerons ici que le pseudo-code de notre algorithme, et non son implémentation en Python.

Variables et notations : Voici les noms de variables que l'on utilisera pour la suite de la descrption de l'algorithme.

— **meilleureSolEnt** initialisée à 0, servira à retenir la plus grande valeur d'une solution entière au cours de l'opération Branch and Bound

— **XChampion** est une liste de taille n^2 contenant la meilleure solution entière que l'on a trouvée.

— **U** est une liste de taille n^2 à valeurs dans {0,1} qui désigne l'état des variables fixées au cours du parcours de l'arbre des sous-problèmes. On initialise U à $[-1, ..., -1]$. Si $U[(i-1)n+j] = -1$, cela signifie que que la condition d'intégralité sur $x_{i,j}$ est relaxée.

— On définit les matrices qui serviront à résoudre les sous-problèmes via un algorithme LP disponible dans la librairie scipy :

→ $\mathbf{A_{eq}(U)} := \begin{pmatrix} \frac{L-C}{H^T} \\ \hline \frac{l_1(U)}{l_2(U)} \end{pmatrix}$ avec $l_1(U)$ et $l_2(U)$ deux listes de taille n^2 définies par $l_1(U)[i] = U[i]$ si $U[i] \neq -1$ et 0 sinon ; $l_2(U)[i] = 1 - U[i]$ si $U[i] \neq -1$ et 0 sinon. Ces deux listes serviront à effectuer un test d'égalité efficace pour imposer les conditions d'intégralité sur les arrêtes définies par U. En effet en gardant la notation $z^T := (1, ..., 1) \in \mathbb{R}^{n^2}$, on vérifie que X a bien ses premières variables fixées (par les valeurs non relaxées qu'impose U) si et seulement si $l_1(U)^T X = l_1(U)^T z$ et $l_2(U)^T X = l_2(U)^T z$

→ $\mathbf{B_{ineq}} := \begin{pmatrix} \frac{C}{IMP} \end{pmatrix}$

→ $\mathbf{B_{eq}(U)} = (0, ..., 0, l_1(U)^T z, l_2(U)^T z) \in \mathbb{R}^{n+3}$

→ $\mathbf{B_{ineq}} = (1, .., 1, K, ..., K) \in \mathbb{R}^{n+N}$

De cette manière, si on cherche à résoudre le problème formulé en 10 avec une relaxation portant sur les arrêtes définies par U, il suffit de résoudre :

$$\boxed{\begin{array}{lll} \text{maximiser} & z^T X & \\ & & \\ \text{avec} & 1. \quad A_{eq}(U)X = B_{eq}(U) & \\ & 2. \quad A_{ineq}X \leq B_{ineq} & \\ & 3. \quad 0 \leq X \leq 1 & \end{array}} \tag{1}$$

Fonctions utilisées : Pour obtenir notre résultat, nous avons besoin de cinq fonctions :

— Une fonction 'prochainIndiceNonFixe' qui cherche le premier indice de la variable globale U qui est différent de -1 (son code, trivial, n'est pas donné ici) ;

— Les fonctions $A_{eq}(U)$ et $B_{eq}(U)$, décrites plus haut, et dont nous ne donnons pas les codes ici pour la même raison ;

— Une fonction 'scipy.optimize.linprog' qui effectue une résolution de problème linéaire. Cette fonction prend trois arguments, qui correspondent à $z, A_{ineq}, B_{ineq}, A_{eq}(U), B_{eq}(U)$ (nous avons défini ces variables pour nous conformer à la documentation de scipy) ;

— Une procédure exploreBranche, décrite ci-dessous. Cette procédure met à jour les variables globales en explorant toute une branche de l'algorithme de Branch and Bound. La branche en question est donnée par le vecteur U, qui fixe une partie des variables.

Ainsi, pour obtenir le résultat global, il suffit d'initialiser les variables comme indiqué ci-dessus et d'appeler exploreBranche().

Algorithm 3: exploreBranche()

Data: Toutes les variables globales ont étées définies comme indiqué plus haut.

Result: Cette fonction ne renvoie rien. Cependant, elle explore récursivement toute la branche de relaxation de l'algorithme de Branch and Bound qui est permise en respectant le vecteur U au moment de l'appel. En sortie d'algorithme, la fonction rétablit le vecteur U tel qu'il était au moment de l'appel de la fonction.

```
# Utilisation d'un algorithme de résolution de problème linéaire
resX = scipy.optimize.linprog(z, A_ineq, B_ineq, A_eq(U), B_eq(U))

# Mise à jour de U
indicesFixesAvant = prochainIndiceNonFixe(U)
for i = 1 to n^2 do
    if resX[i] est entier then
        U[i] = resX[i]
    end
end

# Trois conditions d'arrêt
if resX == None then
    break
end
else if z^T resX < meilleureSolEnt then
    break
end
else if -1 not in U then
    # On a une solution entière !
    # Elle est meilleure que celle qu'on connaît sinon on serait rentré dans le if
    précédent
    meilleureSolEnt = z^T resX
    XChampion = resX
end
else
    k = prochainIndiceNonFixe(U)
    U[k] = 0
    exploreBranche()
    U[k] = 1
    exploreBranche()
end

# Rétablissement de U
for i = indicesFixesAvant to n^2 do
    U[i] = -1
end
```