Martin de CLOSETS
Erwan UMLIL

Février 2021

MATCHING MECHANISMS FOR KIDNEY TRANSPLANTATIONS

1 Introduction

Le présent rapport fait partie du rendu concernant le projet intitulé *Matching Mechanisms for Kidney Transplantations*, dans le cadre du cours *INF421: Design and Analysis of Algorithms*. Il détaille les choix algorithmiques et d'implémentation que nous avons effectués au cours du projet. Il contient également les réponses aux questions posées directement par l'énoncé.

Nous avons choisi de rédiger ce rapport en français et tout le code et le pseudo-code en anglais.

Le problème consiste en un ensemble N de paires (patient, rein). Le but est d'associer, du mieux possible, les reins disponibles aux patients demandeurs. A cette description générale s'ajoutent des contraintes, comme le fait que chaque patient n'est compatible qu'avec certains reins et possède un classement par préférence de ces reins. Nous allons étudier différentes solutions algorithmiques pour résoudre ce problème.

2 Architecture choisie

Nous avons choisi une modélisation en Java selon trois types de classes :

1. La classe *Patient* représente un patient et contient notamment les reins compatibles, les préférences de ce patient ainsi que le rein vers lequel il pointe actuellement. Ce type sera notamment celui des sommets des différents graphes utilisés.

2. Les classes héritant de *Matching* représentent le problème dans son ensemble, selon la partie traitée. Elles contiennent notamment l'ensemble des patients assignés et non assignés, un graphe, une méthode *match()* qui lance l'algorithme associé au mécanisme, ainsi qu'une méthode *runDirectDonation()* permettant d'exécuter un algorithme simple en guise de *preprocessing*.

3. Les classes héritant de *Graph* servent à représenter les graphes contenus dans les classes héritant de *Matching*.

Nos choix d'implémentation ont eu pour but la recherche d'une communication cohérente entre ces trois familles de classes. Au niveau des structures de données, nous avons beaucoup utilisé de tables de hachage, ce qui permet d'avoir accès aux patients en temps constant amorti.

3 Deux approches simples

Question 1. Nous commençons par concevoir un algorithme extrêmement simple : chaque rein est affecté au patient associé, si ceux-ci sont compatibles. Sinon, le patient est inscrit sur liste d'attente. Le pseudo-code du Direct Donation Algorithm est donné ci-dessous.

Algorithm 1: Direct Donation Algorithm

Input : a set N of patient-kidney pairs, $(K_i)_{i\in[[1,n]]}$
Output : a matching M of patient-kidney pairs
M := [];
foreach *(k, t) pair in N* **do**
  if *k is compatible with t* **then**
    Add (k, t) to M;
  else
    Add (w, t) to M;
return M;

On constate immédiatement que cet algorithme a une complexité temporelle en $O(n)$, où n est le nombre de paires (patient, rein).

Question 2. Une approche plus élaborée, bien que toujours très simple, pourrait être de considérer le patient le plus prioritaire, échanger le rein qui lui est associé avec celui qu'il préfère parmi les reins compatibles, puis le second patient le plus prioritaire, à qui l'on affecterait son rein préféré parmi ceux encore disponibles (par un échange bilatéral), etc.. Le pseudo-code de notre Greedy Matching est détaillé ci-dessous.

Algorithm 2: Greedy Matching

Input : an undirected graph $G(V, E)$ as described
Output : a matching M of patient-kidney pairs
M := [];
for *each vertex e in order of priority* **do**
  Let f be the most prefered vertex for e (if it exists);
  Remove e and f from $G(V, E)$;
  Add (k_f, t_e) and (k_e, t_f) to M ;
Assign the remaining vertices to their kidney or to the waiting list, according to their preferences ;
return M;

Si l'on note d le degré maximum des sommets du graphe, on obtient que la complexité temporelle de cet algorithme est en $O(n(d + \log n))$.

4 Vers un mécanisme *efficient* et *strategy-proof*

Question 3. Démontrons le lemme 1 :

Proof. Si chaque paire est la queue d'une *w-chain*, alors il n'y a rien à démontrer
Sinon, considérons une paire (k, t) qui n'est pas la queue d'une *w-chain* alors, en suivant les arêtes du graphe, on obtient une suite $(k, t)^{(i)}$ telle que $(k, t) = (k, t)^{(0)}$ et $t^{(i)}$ pointe vers $k^{(i+1)}$ pour tout $i \in \mathbb{N}$. L'ensemble N étant fini, il existe $(k, k') \in \mathbb{N}^2$ tel que $(k, t)^{(k)} = (k, t)^{(k')}$ et on obtient donc un cycle □

Question 4. Nous considérons ici un mécanisme plus complexe. Commençons par présenter le pseudo-code de l'algorithme principal du mécanisme *Cycles and Chains*.

Algorithm 3: Cycles And Chains Matching Algorithm

```
Input : a set of patient-kidney pairs and a selection rule
Output : a matching M of patient-kidney pairs
M = [];
Let G be a directed graph whose vertices are patients and kidneys and without edges;
while there is a not assigned Patient in the graph do
    Delete all edges in G;
    for each not assigned Patient P do
        Let k be the most preferred kidney for P among the not assigned one's (or the waiting list if it
         does not exist);
        Add the edge (P, k) to G;
    if there exists a cycle in G then
        Let C be such a cycle;
        Make the assignation in C and add patient/donor couples to M;
        Remove all vertices of C from G;
    else
        Select a w-chain W according to the selection rule;
        Make the assignation in W and add patient/donor couples to M;
return M;
```

Cet algorithme peut être exécuté avec la règle de sélection A ou la règle de sélection B. Ces sous-algorithmes de sélection de *w-chain* sont présentés ci-dessous. On remarque que lorsque ces algorithmes sont appelés, il n'y a aucun cycle dans le graphe.

Algorithm 4: Chain Selection Rule A Algorithm

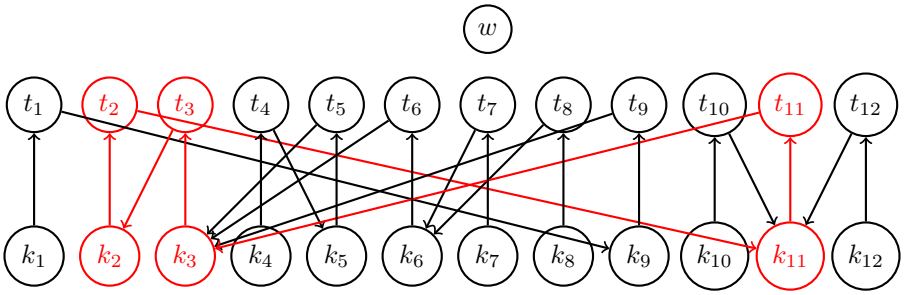
```
Input : a set of patient-kidney pairs
Output : the patient-kidney pair, tail of the selected w-chain
d := 0;
patients := [];
foreach not assigned patient P do
    Compute the size and the priorities of all patients in the w-chain starting with P;
    if size > d then
        d := size;
        patients := [(P, priorities)];
    else if size = d then
        Add (P, priorities) to patients;
foreach P in patients.keySet do
    Sort priorities(P);
return P of patients with minimal priorities(P) in lexicographical order;
```

Algorithm 5: Chain Selection Rule B Algorithm

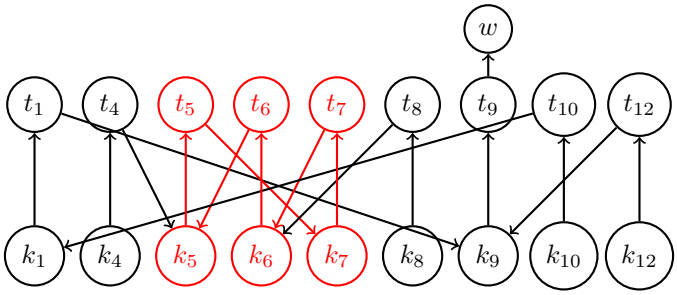
Input : a set of patient-kidney pairs
Output : the patient-kidney pair, tail of the selected *w-chain*
i := 1;
for j = *2 to* n **do**
  if t_j *has more priority than* t_i **then**
    i := j;
return (k_i, t_i);

L'algorithme de sélection selon la règle A est en $O(n^2)$ en temps, tandis que celui selon la règle B est en $O(n)$. Il en découle que l'algorithme principal est, temporellement, en $O(n^3)$.

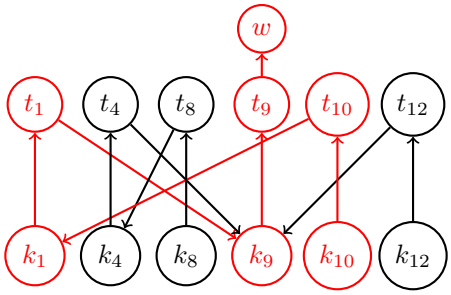
Question 5. Décrivons l'exécution de l'algorithme avec la règle A "à la main". Au début du premier *round*, la situation est décrite par le graphe suivant : chaque patient pointe vers le rein qu'il souhaite le plus obtenir.



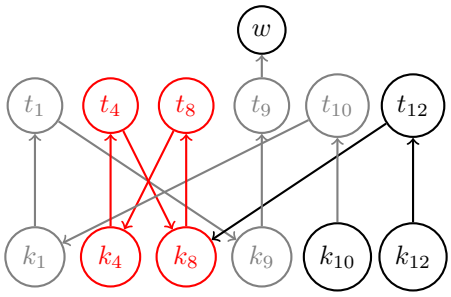
La séquence $(k_2, t_2, k_{11}, t_{11}, k_3, t_3)$ constitue un cyle, on effectue donc les affectations correspondantes et on retire les couples (patient, rein).



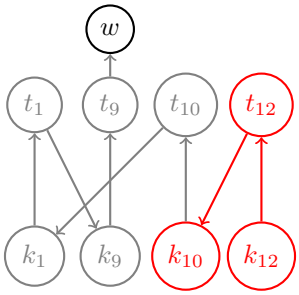
La séquence $(k_6, t_6, k_5, t_5, k_7, t_7)$ constitue un cycle, on effectue donc les affectations correspondantes et on retire les couples (patient, rein).



Il existe deux *w-chains* de taille maximale : l'une commençant en k_8, l'autre en k_{10}. Cette dernière est la seule à contenir k_1, elle est donc sélectionnée selon la règle A, les affectations sont réalisées, mais les noeuds ne sont pas supprimés du graphes. Ceux qui sont affectés sont représentés en gris sur les schémas suivants.



La séquence (k_4, t_4, k_8, t_8) constitue un cyle, on effectue donc les affectations correspondantes et on retire les couples (patient, rein).



La w - *chain* commencant en k_{12} est l'unique maximale et est donc sélectionnée.

On obtient la répartition finale suivante : $t_1 : k_9; t_2 : k_{11}; t_3 : k_2; t_4 : k_8; t_5 : k_7; t_6 : k_5; t_7 : k_6; t_8 : k_4; t_9 : w; t_{10} : k_1; t_{11} : k_3; t_{12} : k_{10}$.

Question 6. Nous donnons une preuve du théorème 1, après avoir montré trois invariants de boucle et la terminaison de l'algorithme. On désigne par *option* un élément pouvant être assigné à un patient, c'est-à-dire un élément de K_i pour t_i. On désigne par *round* une itération de la boucle principale du *Cycles and Chains Matching Algorithm*. Pour les trois invariants de boucle ci-dessous, on considère une règle de sélection de chaîne qui conserve dans la procédure chaque *w-chain* sélectionnée lors d'un *round* intermédiaire.

Invariant de boucle 1. *Tout patient n'est assigné qu'à au plus une option.*

Proof. Montrons cet invariant de boucle par récurrence.

Au début du premier *round*, tous les patients ont 0 option assignée.

On suppose que la propriété est vraie au début d'un certain *round*. Soit P un patient. Alors soit P n'a pas encore d'option assignée, soit P en a exactement une. Le premier cas étant trivial, nous nous plaçons dans le deuxième.

P peut avoir été exclu de la procédure, s'il a été assigné via un cycle, auquel cas il ne pourra pas être examiné à nouveau par l'algorithme.

Enfin, P peut avoir obtenu son option via une *w-chain*, auquel cas il est toujours présent dans la procédure. Supposons que la *w-chain* dans laquelle il se trouve soit à nouveau sélectionnée. L'algorithme indique que seul les patients de la *w-chain* qui n'ont pas déjà eu une assignation sont assignés. Par conséquent, à l'issue du *round*, P n'a toujours qu'une assignation.

La propriété est donc encore vérifiée à la fin du *round*. □

Invariant de boucle 2. *Tous les patients ayant déjà une option assignée préfèrent strictement cette option à toutes celles encore non assignées.*

Proof. Soit P un patient ayant déjà une option assignée. Considérons le *round* lors duquel cette assignation a été réalisée. Un tel *round* existe car il s'agit, dans l'algorithme, du seul moyen pour P d'avoir obtenu une option.

Au début de ce *round*, en vertu de l'invariant de boucle 1, P n'avait pas d'option assignée. D'après l'algorithme, il pointait alors vers l'option qu'il préférait parmi les options restantes. Cette préférence est stricte car la liste des préférences de chaque patient l'est.

Or on constate dans l'algorithme que le graphe reste inchangé jusqu'à l'assignation, à la fin du *round*. Comme P se voit assigné une option à l'issue de ce *round*, on en déduit qu'il s'agit de cette option vers laquelle il pointe, et donc qu'il la préfère strictement à toutes les autres options disponibles. □

Invariant de boucle 3. *Le matching partiel constitué des éléments déjà assignés est Pareto-efficient.*

Proof. Montrons cet invariant de boucle par récurrence.

Au début du premier *round*, le matching est vide donc la propriété est trivialement vraie.

On suppose que la propriété est vraie au début d'un certain *round*. On va considérer tous les échanges possibles au sein du matching constitué des paires patient-rein déjà formées, et montrer que tout échange conduit à un matching qui est strictement moins apprécié par au moins l'un des patients. Par négation, on aura alors montré que le matching partiel obtenu par l'algorithme est *Pareto-efficient*.

On distingue les anciens, qui étaient déjà assignés au début du *round*, et les nouveaux entrants, qui se voient assigner un rein lors du présent *round*. Par hypothèse de récurrence, aucun échange n'est possible parmi les anciens sans créer un matching strictement moins apprécié par au moins l'un des patients.

Par l'invariant de boucle 2, et sachant que les préférences de chaque patient sont constantes au cours de l'exécution de l'algorithme, tout échange entre un ancien et un nouveau conduirait à un matching strictement moins apprécié par l'ancien concerné.

Enfin, par ce même invariant de boucle, tout échange entre deux nouveaux conduirait à un matching strictement moins apprécié par les deux nouveaux concernés.

On en déduit que la propriété est reste vraie à l'issue du *round*. □

Terminaison. *Le Cycles and Chains Matching Algorithm termine en au plus n itérations.*

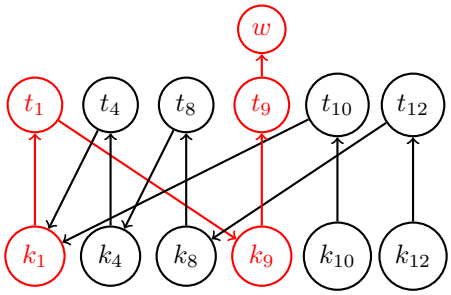
Proof. Lors de chaque round, au moins un patient voit un rein lui être assigné. Par conséquent, le nombre de patients n'ayant pas de rein assigné est un variant de boucle, valant initialement n, et décroissant strictement à chaque itération. Lorsqu'il est négatif, la boucle s'arrête et l'algorithme termine. □

Théorème 1. *Considérons une règle de sélection de chaîne qui conserve dans la procédure chaque w-chain sélectionnée lors d'un round intermédiaire. Alors avec une telle règle, le mécanisme d'échange est efficient.*

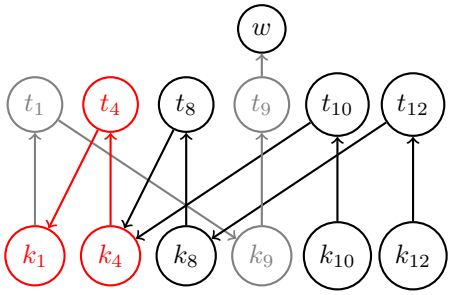
Proof. L'invariant de boucle 3 et la terminaison montrent que l'algorithme génère un mathcing *Pareto-efficient*. Par conséquent, le mécanisme d'échange est *efficient*. □

Question 7. On constate, dans l'exemple développé à la question 5, le fait suivant : si t_{12} ment sur ses réelles préférences et place k_8 avant k_9 dans sa liste de priorité, alors il sera affecté à k_8 lors du *round* 3. Or $k_8 > k_{10}$ pour lui, il a donc gagné à mentir sur ses préférences. Par conséquent, l'algorithme n'est donc pas *strategy-proof*.

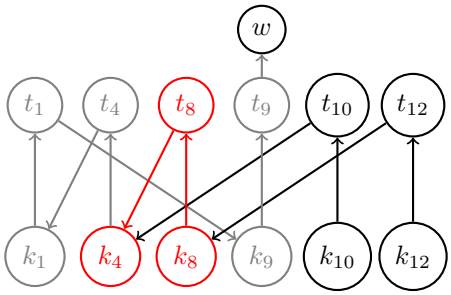
Question 8. Pour la règle de décision B, les trois premiers graphes sont identiques à ceux de la règle A. Cependant, c'est la *w-chain* commençant par k_1 qui est choisie. On obtient donc, au début du quatrième *round* :



On sélectionne la *w-chain* commençant par k_4.



On sélectionne la *w-chain* commençant par k_8.



Enfin, on assigne w à t_{10} et k_8 à t_{12}, ce qui termine l'algorithme.

Finlement, la répartition est la suivante : $t_1 : k_9; t_2 : k_{11}; t_3 : k_2; t_4 : k_1; t_5 : k_7; t_6 : k_5; t_7 : k_6; t_8 : k_4; t_9 : w; t_{10} : w; t_{11} : k_3; t_{12} : k_8$.

5 Programmation linéaire

Question 9. Pour calculer l'ensemble des *minimal infeasible paths*, nous proposons d'implémenter une fonction principale qui appelle une fonction récursive sur chacun des sommets du graphe. Ces algorithmes sont détaillés ci-dessous.

Algorithm 6: computeAllMinimalInfeasiblePaths

Input : a directed graph G(V,E), an integer K
Output : a list P of all minimal infeasible paths in G(V,E)
P := [], visited := [], current := [];
foreach $v \in V$ **do**
  computeMinimalInfeasiblePathsRec(G(V,E), v, K, 0, P, current, visited);
return P;

Algorithm 7: computeMinimalInfeasiblePathsRec

Input : a directed graph G(V,E), a vertex $v \in V$, an integer K, a integer c, a list of paths P, a list current, a set visited
Output : none
if $c = K + 1$ **then**
  return;
if $v \in$ visited **then**
  return;
Add v to visited;
Add v at the end of current;
if $c = K$ **then**
  Add a copy of current to P;
foreach $w \in N^+(v)$ **do**
  computeMinimalInfeasiblePathsRec(w, K, $c + 1$, P, current, visited);
Remove v from current;
Remove v from visited;

Nous employons ici une technique de rebroussement. La fonction récursive est clairement en $O(d^K)$ où d est le degré maximal des sommets du graphe. Par conséquent, la fonction globale est en $O(nd^K)$.

Question 10. Nous formulons ici un *integer linear program* (ILP) nous permettant de résoudre le problème.

$$Maximise \sum_{(i,j)\in E} x_{i,j} \text{ subject to :}$$

$$\forall i \in V, \sum_{j\in N^+(i)} x_{i,j} \leq 1$$

$$\forall j \in V, \sum_{i\in N^-(j)} x_{i,j} \leq 1$$

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

$$\forall \pi \text{ } \textit{minimal infeasible path}, \sum_{e\in\pi} x_{i,j} \leq K$$

$$\forall i \in V, \sum_{j\in N^+(i)} x_{i,j} - \sum_{k\in N^-(i)} x_{k,i} = 0$$

Question 11. L'algorithme présenté ci-dessous résoud cet ILP en utilisant la méthode *Branch-and-Bound*. Nous utilisons le solver SSC.

Algorithm 8: Branch and Bound (BB) Algorithm

Input : an ILP, an upper bound b and an intermediate result res
Output : result of the intermediate problem
Solve the ILP as an LP, that gives a value r;
b = min(r,b);
if $r \leq res$ *or problem infeasible* **then**
    return *res;*
if *the solution of the LP is integer* **then**
    $res = r$;
    return *res;*
else
    Let i be a not integer coordinate of the solution;
    return *max(BB(ILP + constraint* $x_i = 0$*, b, res), BB(ILP + constraint* $x_i = 1$*, b, res));*

Question 12. Cette méthode permet 9 affectations pour le fichier test2 en environ 70 ms et 3 pour test1 en environ 270 ms. Cette différence d'efficacité est dû au fait que le graphe de test2 est beaucoup plus dense que celui de test1 et qu'il présente donc plus de cycles.

Notre algorithme ne nous a pas permis d'obtenir des résultats pour test3 à cause d'une erreur mémoire que nous n'avons pu résoudre.

6 Analyse quantitative

Question 13. On se place dans la configuration statistique décrite par l'énoncé. Dans le répertoire *results/*, nous fournissons les résultats expérimentaux obtenus lors de l'exécution des algorithmes Direct Donation, Greedy Matching et Cycles and Chains Matching. Plus précisément, sont donnés le nombre de transplantations réalisées, ainsi que leur moyenne par algorithme.

On constate d'abord, comme on pouvait l'imaginer, que le nombre de transplantations augmente lorsque l'on passe de Direct Donation à Greedy Matching, et de Greedy Matching à Cycles and Chains Matching.

Nous essayons alors de faire tourner l'algorithme Direct Donation, en guise de *preprocessing*, avant les deux autres algorithmes. Logiquement, on constate alors que la moyenne des transplantations diminue. Cependant, dans le cas de Cycles and Chains Matching, cette diminution de la moyenne est très faible, typiquement inférieure à une transplantation. De plus, Direct Donation étant beaucoup plus rapide que cet algorithme, il peut permettre de diminuer la taille de l'ensemble des paires (patient, rein) sans perte importante de qualité du résultat. Finalement, ce prepocessing peut accélérer Cycles and Chains Matching.

Avec ou sans *preprocessing*, le plus efficace en termes de nombre de transplantations est de réaliser des échanges multilatéraux (Cycles and Chains Matching) plutôt que bilatéraux (Greedy Matching).