Matching mechanisms for kidney transplantations

Tristan François

February 6, 2021

1 Two Simple Approaches

Question 1

Algorithm 1: Direct Donation

Input: A number n of patient and, for all patients $i \in \mathbb{N}$, their set of compatible donors $K_i \subseteq [\![1, n]\!]$.

Output: A set of pairs donor-patient pairs M and a waiting list w.

```
initialization
    M = ∅;
    w = [];
begin
    for i = 1 to n do
        if k_i ∈ K_i then
            M = M ∪ {(k_i, t_i)};
        else
            w.add(t_i)
    return c, w
```

The implementation in python is pretty straightforward. The python function takes as parameter an integer n and an integer set array of size n. The use of python set allows to test the if condition in constant times. The only difference with pseudo-code is the use of a list rather than a set for the variable c. This choice was made to avoid the unnecessary heaviness of the set data structure, but does not change the complexity.

Question 2

Our matching algorithm must prioritize the patients with the highest priority. It must also ensure that all patients will leave with a donor that suited them better than their paired donor and the waiting list, otherwise some patients may not participate in the exchange program.

Our greedy algorithm therefore consists of browsing the patients in descending order of priority and matching them to their preferred donor whose paired patient does not prefer the waiting list or his donor to the donor of the priority patient. Note that a patient can be matched to his own donor.

Algorithm 2: Greedy Matching

Input: A number n of patient, a strict priority list of the patients U and, for all patients $i \in \mathbb{N}$, their set of compatible donors $K_i \subseteq [\![1, n]\!]$ and their strict preference relation P_i over $K_i \cup \{k_i, w\}$. The priority list is an integer list that starts with the index of the highest priority patient and goes to the index of the lowest priority patient.

Output: A matching M.

```
1  initialization
2      M = ∅;
3      is_matched = Boolean[1..n];
4      sorted_K = IntegerList[1..n];
5      for i = 1 to n do
6          is_matched[i] = False;
7          sorted_K[i] = sort(K_i, P_i) ;                    // O(n log(n))
8  begin
       /* Iterate over all patient by decreasing priority     */
9      for u ∈ U do
10         if not is_matched[u] then
11             for v ∈ sorted_K[u] do
12                 if not is_matched[v] then
                       /* If t_v prefer k_u over k_v and w     */
13                     if k_v P_v k_u and k_v P_v w then
14                         M = M ∪ {((k_u, t_v), (k_v, t_u))};
15                         is_matched[u] = True;
16                         is_matched[v] = True;
17                         break;
18     return M
```

We start by initializing an *is_matched* array to keep track of which pair has already been matched and an integer list array *sorted_K* that contains, for each patient of index i, the indexes of the preferred donors in descending order of preference. The sort function returns a list of the elements of K_i sorted in descending order according to the preference relation P_i.

The implementation is almost exactly the pseudo-code.

2 Efficient stategy-proof exchange mechanism

Question 3

Let (k_1, t_1) be a pair of donor-patient and assume there is no cycle. Consider the chain starting from (k_1, t_1). Since there is a finite number n of donor-patient pairs and there is no cycle, the chain cannot be infinite. Finally, since the only way for a chain to stop is that w belong to the chain, (k_1, t_1) is indeed a tail of a w-chain.

Thus either there exists a cycle, or each pair is the tail of some w-chain.

Question 4

The general sequence of the algorithm is shown in the algorithm below. For the sake of brevity, we omit some auxiliary functions like `GetCycles`, which returns the cycles in a directed graph, or `Update`, which updates the state of the graph according to the procedure described in the project description. The others not detailed functions will be discussed after the algorithm and the pseudo-code of `SelectChainA` is given in algorithm 4.

Algorithm 3: Cycles and Chains Matching Algorithm

Input: A number n of patient, a strict priority list of the patients U and, for all patients $i \in \mathbb{N}$, their set of compatible donors $K_i \subseteq [\![1, n]\!]$, their strict preference relation P_i over $K_i \cup \{k_i, w\}$ and the initial graph G.

Output: A matching M.

```
initialization
    M = ∅;
    removed = ∅;
    passive = ∅;
    unavailable = ∅;
begin
    while |removed| + |passive| ≠ n do
        cycles = GetCycles(G);
        if |cycles| ≠ 0 then
            for cycle ∈ cycles do
                AssignCycle(cycle);      // modifies M and removed
        else
            chain = SelectChainA(G);
            /* modifies M, passive and unavailable              */
            AssignChain(chain);
        Update(G);
    return M
```

At any time, we keep up to date the *removed* set of vertices removed from

the graph during the cycle deletion, the *passive* set of vertices which are still in the graph but whose target no longer changes, as well as the *unavailable* set of vertices to which we can no longer point. Those sets allow to update the graph without difficulty. Each time we choose a w-chain, all of its vertices are added to *passive* and all vertices except the head are added to *unavailable*. These operations are performed by the `AssignCycle` and `AssignChain` functions.

We still have to discuss a crucial part of the algorithm, the chain selection rule A. According to the rule statement, the first thing to do is find the longest chains.

In order to quickly obtain the longest chains, it should be noted that the transposed graph is a tree with root w. A depth-first search gives us the depth of all vertices in the tree. Once the depths are obtained, it is very easy to find the longest chains. This is what the omitted `GetLongestChains` function does.

Once we have the longest chains, we still need to find the one with the patient with the highest priority. For this, we take all the patients who appear in the longest chains, and we sort them in decreasing order of priority. Then, we iterate on this patient list, look to which chain the patient belongs to, and remove all the other chains. When there's only one chain left, we've found the one we're interested in.

Algorithm 4: Chain Selection Rule A

```
1  Function SelectChainA():
2      chains = GetLongestChains();
3      nodes = ∅;
4      for chain ∈ chains do
5          nodes = nodes ∪ chain
6      for node ∈ nodes by decreasing priority do
7          new_chains = ∅;
8          for chain ∈ chains do
9              if node ∈ chain then
10                 new_chains = new_chains ∪ chain ;
11         if |new_chains| = 1 then
12             return new_chains.pop();
13         else if |new_chains| > 1 then
14             chains = new_chains;
```

For the implementation, readability was preferred over pure efficiency, which is why an object structure with heavy use of python set was preferred. The use of python set is also due to the fact that we often have to test the membership of an element in a set, whether for the construction of the graph, in the cycle search, or in the selection of the chain. To make things a bit simpler, we can notice that, since a donor always points to his paired patient, there is no need to have a separate node for the donor and the patient. Thus in the implementation, there is only one node in the graph for each donor-patient pair.

Question 5

Let's run the Cycles and Chains Matching Algorithm with rule A on this example. Graphs associated with every round are shown in figure 1.

The initial graph contains a cycle $C_1 = (k_3, t_3, k_2, t_2, k_{11}, t_{11})$. The first round of the algorithm therefore consists in detecting this cycle, assigning the kidneys to the patients and then eliminating the cycle.

After removing the first circle, a new circle $C_2 = (k_5, t_5, k_7, t_7, k_6, t_6)$ is formed. In the second round, this cycle is therefore deleted.

At the start of the third round, there is no more circle. The graph contains longest w-chains : $W_1 = (k_8, t_8, k_4, t_4, k_9, t_9)$ and $W_2 = (k_{10}, t_{10}, k_1, t_1, k_9, t_9)$. Since the highest priority patient t_1 is only present in W_2, W_2 is chosen. The kidneys are assigned and the chain remains in the graph for the next round but patients pointing to other kidneys than k_10 in the chain are redirected.

After the redirect, a new circle $C_3 = (k_4, t_4, k_8, t_8)$ has formed. The fourth round therefore consists in the elimination of this cycle.

There is no new circle and the patient t_{12} now points to k_{10} forming a chain of maximum length $W_3 = (k_{12}, t_{12}, k_{10}, t_{10}, k_1, t_1, k_9, t_9)$. Round 5 assigns kidneys k_{10} to patient t_{12} and there are no more patients with no kidneys. The algorithm stops.

The final matching is:

$$\begin{pmatrix} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & t_{10} & t_{11} & t_{12} \\ k_9 & k_{11} & k_2 & k_8 & k_7 & k_5 & k_6 & k_4 & w & k_1 & k_3 & k_{10} \end{pmatrix}$$

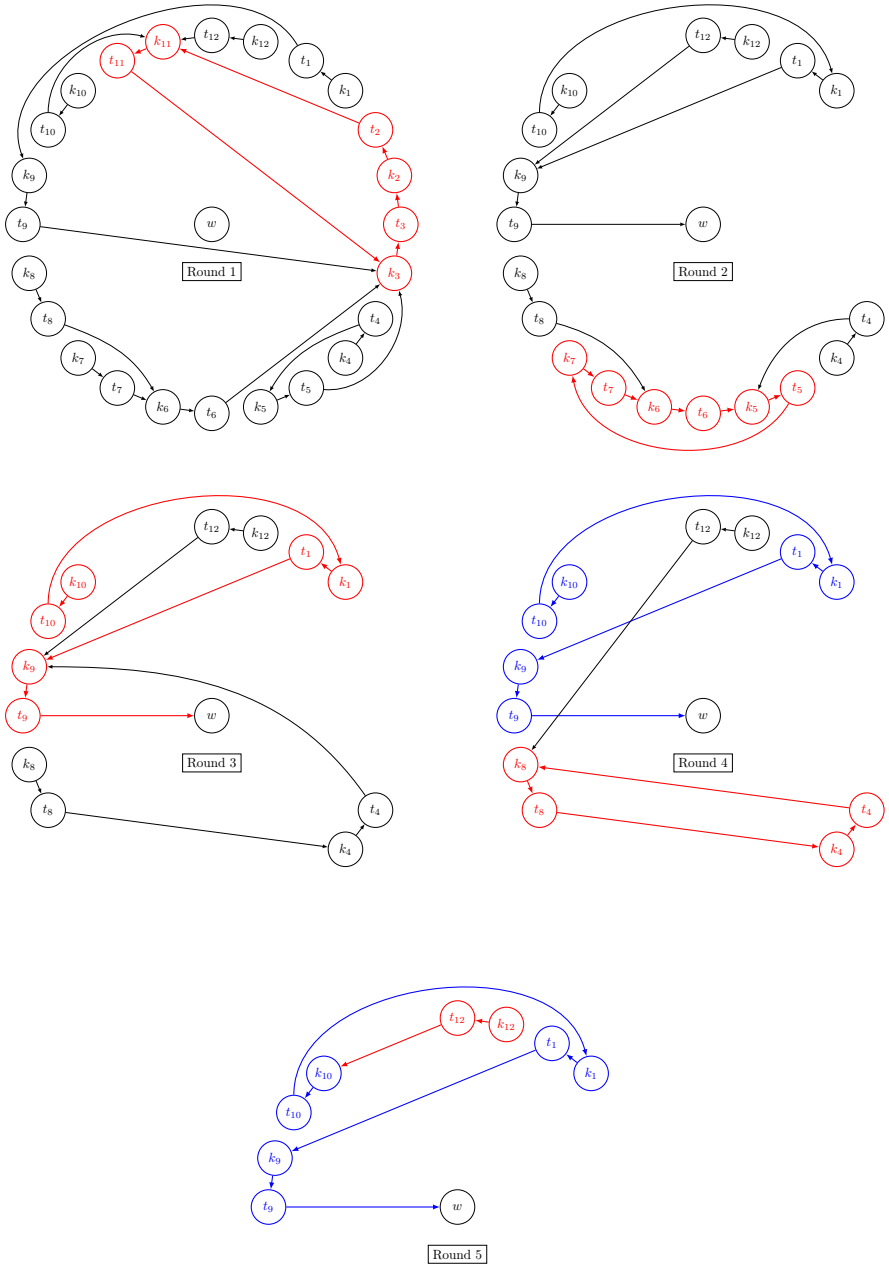

Figure 1: State of the graph at every round.

Question 6

If the selection rule keep in the procedure any selected w-chain at a non-terminal round, then every unasigned kidney is available at every round of the procedure. Therefore, at every round of the algorithm, a patient can only be assigned to his best choice among remaining kidneys. The only way to give him a better kidney is to take it back from a patient whose assignement occurs in a previous round. Since this patient had his best choice at the time, he will strictly disprefer the new matching. Thus, the exchange mechanism is efficient.

Question 7

If we swap k_8 and k_9 in the order of preference of the twelfth patient, then in round 3 the longest w-chain becomes $(k_{12}, t_{12}, k_8, t_8, k_4, t_4, k_9, t_9)$ and t_{12} is assigned to k_8, which he strictly prefers to k_{10}, that he would get by being honest about his preferences. Therefore the exchange mechanism with chain selection rule A is not strategy-proof.

Question 8

The first two rounds are the same as with rule A. In the third round, we choose the w-chain starting with (k_1, t_1), the highest priority pair. As with rule A, during the fourth round we delete the cycle (k_4, t_4, k_8, t_8). Finally, on round 5 and 6, pairs (k_{10}, t_{10}) then (k_{12}, t_{12}) come to lengthen the chain formed in the third round. The final matching is the same as the one obtained with rule A.

Graphs associated with every round are shown in figure 2.

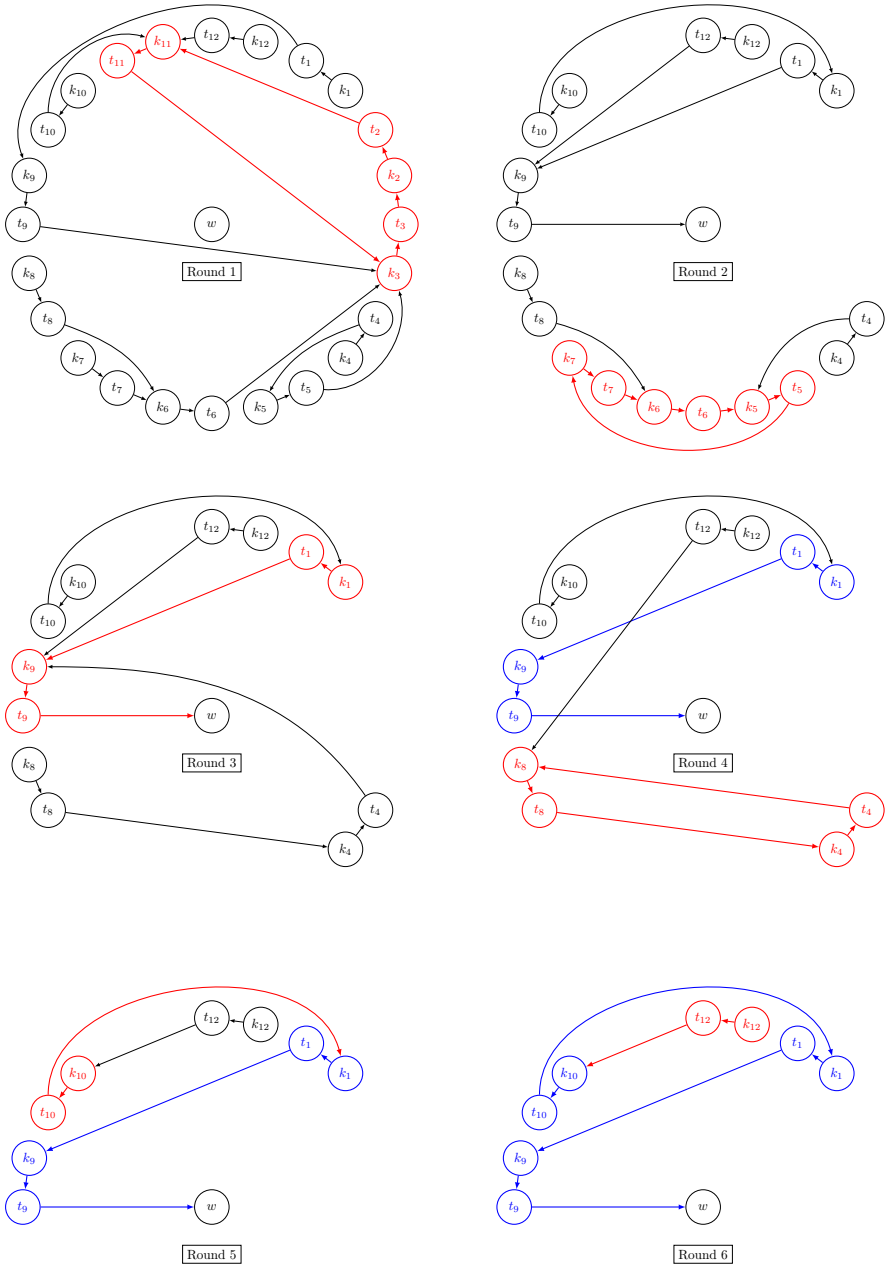

Figure 2: Problem Graph at the beginning of each round.

In agreement with Theorem 2, we can notice that agents cannot hope to get

a better assignment by lying about their preferences. For example, the strategy used in the previous question by the twelfth patient is no longer effective, because whatever his preferences, he will be affected last.

3 Integer programming formulation

Question 9

In order to list all the minimal infeasible paths, we will simply use a recursive function `Aux` which, given a graph G, a vertex v and an integer k, returns all the paths of lengths $k+1$ starting at v. In order to avoid cycles, we add an *inpath* variable, a set containing the vertices already in the current path. Finally, we called this function on all the vertices of the graph.

Algorithm 5: Minimal infeasible paths

Input: $G = (V, E)$ a graph and k the threshold.
Output: The set of all minimal infeasible paths.

```
1  begin
2      paths = ∅;
3      for v ∈ V do
4          paths = paths ∪ {Aux(G, v, k, ∅)};
5      return paths;
   /* Return the list of all minimal infeasible path starting
      from v                                                    */
6  Function Aux(G, v, k, inpath):
7      if k = 0 then
8          return {[v]};
9      inpath = inpath ∪ {v};
10     paths = ∅;
11     for neighbour ∈ v.neighbours do
12         if neighbour ∉ inpath then
13             for path ∈ Aux(G, neighbour, k − 1, inpath) do
14                 paths = paths ∪ {path.add_first(v)};
15     inpath = inpath \ {v};
16     return paths;
```

The implementation in python is very similar to the pseudo-code, except that the algorithm returns a list of paths rather than a set. The graph is given in the form of adjacency lists, in order to be able to iterate quickly on the neighbors.

file	CPU time	$\|V\|$	$\|E\|$	number of paths
test2.txt	$0.097 * 10^{-3}$ seconds	12	18	45
test3.txt	$31.008 * 10^{-3}$ seconds	17	229	28188

Figure 3: Mean execution time over 1000 iterations on an Intel i5-6200U processor.

Question 10

Let n be the number of patients and M the adjacency matrix of the graph. We will denote by $m_{i,j}$ the coefficients of M. We therefore have $m_{i,j} = 1$ if the kidney k_i is compatible with the patient t_j and $m_{i,j} = 0$ otherwise. We also define Π as the set of minimal infeasible paths.

For all $(i,j) \in [\![1,n]\!]^2$, we define a variable $a_{i,j}$ which is equal to 1 if the kidney k_i is assigned to the patient t_j and equal to 0 otherwise. We denote by A the assignment matrix whose coefficients are $a_{i,j}$. Since we are trying to maximize the number of transplants, our objective function will be:

$$\max_{A} \sum_{(i,j)\in[\![1,n]\!]^2} a_{i,j} \tag{1}$$

However, we are subject to a few of constraints. To begin with, by definition of $a_{i,j}$, we have the following first set of constraints.

$$\forall (i,j) \in [\![1,n]\!]^2,\ a_{i,j} \in \{0,1\} \tag{2}$$

In addition a kidney k_i can only be assigned to a patient t_j if t_j is compatible with k_i, in other words $a_{i,j}$ cannot be equal to 1 if $m_{i,j}$ is 0. Therefore $a_{i,j}$ is necessarily less than or equal to $m_{i,j}$. So we have a new set of constraints:

$$\forall (i,j) \in [\![1,n]\!]^2,\ a_{i,j} \leq m_{i,j} \tag{3}$$

In addition, a patient cannot receive more than 1 kidney and a donor will not donate his kidney if his paired patient does not receive a kidney. Thus, we have:

$$\forall j \in [\![1,n]\!],\ \sum_{i=1}^{n} a_{i,j} \leq 1 \tag{4}$$

$$\forall j \in [\![1,n]\!],\ \sum_{i=1}^{n} a_{j,i} \leq \sum_{i=1}^{n} a_{i,j} \tag{5}$$

Finally, we need to verify that there is no exchange cycle of length more than the threshold K. To do this, it suffices to check that none of the minimal infeasible paths is taken. We thus obtain our last set of constraints:

$$\forall \{i_1, i_2, \ldots, i_{K+1}\} \in \Pi,\ \sum_{k=1}^{K} a_{i_k, i_{k+1}} \leq K - 1 \tag{6}$$

Our integer linear program is therefore the maximization of 1 subject to 2, 3, 4, 5 and 6.

Question 11

The relaxed problem is the same as before with constraint 2 replaced by a new constraint:

$$\forall (i,j) \in [\![1,n]\!]^2,\ 0 \leq a_{i,j} \leq 1 \tag{7}$$

In the following algorithm, we assume given a function allowing to solve a linear program and returning the solution and the value of the objective function. If the problem is infeasible, the value of the objective function will be $-\infty$. We also assume given a function that test if the solution is an integer one.

At each node of the tree, we will force a variable $a_{i,j}$ to take the value 0 on one branch and 1 on the other by adding the constraints $a_{i,j} \leq 0$ and $a_{i,j} \geq 1$, then we'll take the best solution of the two branches.

Algorithm 6: Branch and Bound

Input: A linear problem *prob* and a set of relaxed variables *vars*
Output: A solution to the problem

```
1  begin
2      return BranchAndBound(prob, vars, 0, ∅)
3  Function BranchAndBound(prob, vars, best, solution):
4      score, new_sol = prob.solve();
5      if score = −∞ then
6          return best, solution;
7      else if score < best then
8          return best, solution;
9      else if IsIntegerSolution(new_solution) then
10         return score, new_sol;
11     else
12         x = vars.pop();
13         prob.add_constraint(x ≥ 1);
14         score_1, new_sol_1 = BranchAndBound(prob, vars, best, solution);
15         prob.remove_constraint(x ≥ 1);
16         prob.add_constraint(x ≤ 0);
17         score_2, new_sol_2 = BranchAndBound(prob, vars, best, solution);
18         prob.remove_constraint(x ≤ 0);
19         vars.add(x);
20         if score_1 > score_2 then
21             return score_1, new_sol_1;
22         else
23             return score_2, new_sol_2;
```

For the implementation, we used the Python `Pulp` library as the linear program solver. Unfortunately, it is not possible to simply remove a constraint from

a linear problem with this library. Instead of the problem, we therefore pass the variables, the objective function and the list of constraints to each recursive call and we recreate the problem each time. Other than that, the implementation is very similar to pseudo-code.

Question 12

The results of the previous algorithm are available in the following figure. Note that the execution time is very good because, in both cases, the relaxed problem directly gives an entire solution. The Branch-and-Bound algorithm is therefore unnecessary for the cases below. However, there are cases where the linear program does not give integer solutions. The code provided with the report contains functions capable of generating such cases.

file	CPU time	selected cycles	objective value
test2.txt	0.041 seconds	$(0, 8, 2)$, $(1, 10, 9)$, $(3, 4, 5)$	9
test3.txt	1.658 seconds	$(0, 1, 14)$, $(2, 6)$, (3), (4), (5), (7), $(8, 16)$, $(9, 10)$, (11), (12), (13), (15)	17

Figure 4: Results and mean execution time over 10 iterations on an Intel i5-6200U processor.

Question 13