Graduate AI Project: Integer Programming Applications in Solving Static and Dynamic Kidney Exchange

Guofan Wu, Xiao Li, Department of Mechanical Engineering

ABSTRACT: Kidney transplantation is an efficient treatment in dealing with kidney malfunction. Arising from the lack of compatible donor for individuals, the kidney exchange allows each patient-donor pair to join in a "pool" where they can swap donor with each other. Our project purpose is to study this problem from AI prospective. In the first part, we make a solver based on Gurobi 5.5 to solve the static clearing problem in kidney exchange. Using this solver we have tested the influence of "altruist" donor on the overall performance. Then combining this static-case solver with a learning-based method, a dynamic-matching solver is proposed to improve the overall performance of the dynamic graphs which have multi-inputs. Several experiments are performed to test its performance with the myopia one as well as tune its parameters. We also add some comments to these results.

General Terms: Search, Linear Programming

Additional Key Words and Phrases: kidney exchange, breadth-first search, integer programming

1. INTRODUCTION

Kidney is the organ for human to filter out food toxi. The patients encoutnering kidney malfunction usually resort to two treatments: transplant and dialysis. Compared with dialysis, the average life quality and survival rate of transplant are much higher[2; 3]. But it requires a living donor which needs to be both blood and tissue compatible with the patients. The kidney exchange problem arises from the fact that the demands for a compatible live donor greatly surpass the current supply. In order to reduce the number of deaths among waiting patients, a large "pool" keeps opening for these impatible patient-donor pair where they can swap their donors with each other. In this project, we are attempting to understand the AI methods exploited in kidney exchange problem and build up our own solver for small-scale kidney-exchange problem. The part of this report could be divided into the following part.

— The basic clearing problem could be formulated as *an integer programming within a directed graph.* Different from the previous search problems in AI, space cost becomes a critical issue for this kind of search with respect to time cost. Thus, several specific techniques for saving memory are introduced here. Also, some results are shown here.
— Based on the previous static allocation solver, we propose two kinds of modified dynamic allocation solvers. One of them is performed online while the other could be treated as an offline algorithm. Both of them are highly related to the "potential" concepts in [2] which we'll give our understandings and comments.

2. STATIC ALLOCATION SOLVER

This solver deals with the static kidney exchange problems where the swapping pool is static. In this case, the pool could be treated as a direct graph where each patient-donor pair is a node of the graph(fig.1). In the graph, the compatiblity between donor of a given node and patient of another one is *indicated by an edge from the latter to the former* while its weight could be viewed as *the utility of the donor for this patient*. But under most circumstances, we would only treat them as equal for the static allocation. The tree basic clearing problem is to find out such a set of *disjoint cycles* in a given direct graph that the sum of their weights are *maximal*. The concept "disjoint" here means that none of the cycles share the same node and the weight of a cycle is the

sum of all the edges it contains. Here's the mathematical abstraction of the previous statements.

2.1. Kidney Exchange Abstraction: Problem Formulation

Given a directed weighted graph and a length cap

$$\mathcal{G} = \{\mathcal{V}, (\mathcal{E}, \mathcal{W})\}, \quad max\_length = l_c$$

find out **such** a set $\mathbf{C} = \{c_1, c_2, \cdots, c_n\} \subseteq \mathcal{C}$ where $\mathcal{C}$ are the set of all the cycles with length no greater than l_c **that** the following conditions are satisfied:

(1) $\forall x, y \in \mathbf{C}, x \cap y = \varnothing$ (disjoint properties)
(2) $\sum_{i=1,\cdots,n} w(c_i)$ is the maximum value (optimality)

2.2. Subproblem: Search All the Cycles

Based on the previous model, the first issue that needs to be dealt with arises: how to enumerate all the cycles in a given direct graph. Thus, after having tried several special algorithms listed in wikipedia, we come up with the idea of breadth-first search. We mention some of them in our milestone report. But now here's a detailed description about how the algorithm is performed and realized in C++.

The basic idea which lies behind the breadth-first is to find all the cycles **sharing the same vertex** at one time and **apply the same method to the others without replications.** Then the searching job has been decomposed into two separate tasks: finding all the cycles containing a given vertex, checking and deleting the same cycles found already. So the following two parts deal with them separately.

2.2.1. Finding all the Cycles Containing a Given Node. Now suppose we just want to search all the cycles having the same given node without any knowledge of the other cycles. A breadth-first search could accomplish that task from AI course. To easily illustrate how it is actually performed, we cite the following example. For figure cited here, if we want to find all the cycles with length no greater than 3 containing $v2$ initially, a tree would be constructed by following these steps.

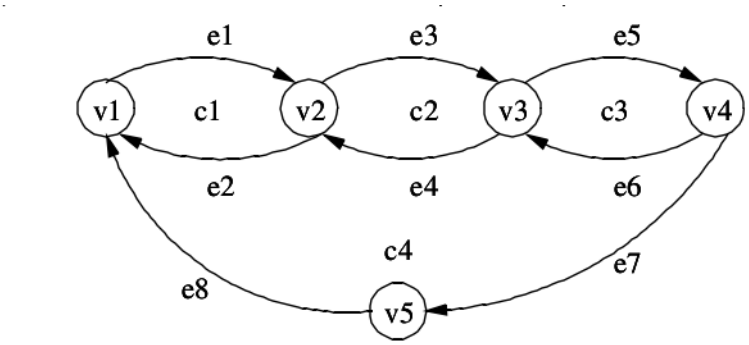

Figure 1: Example barter exchange market.

Fig. 1. Cited From [1]

(1) *Initialization:* Select the root node as $v2$

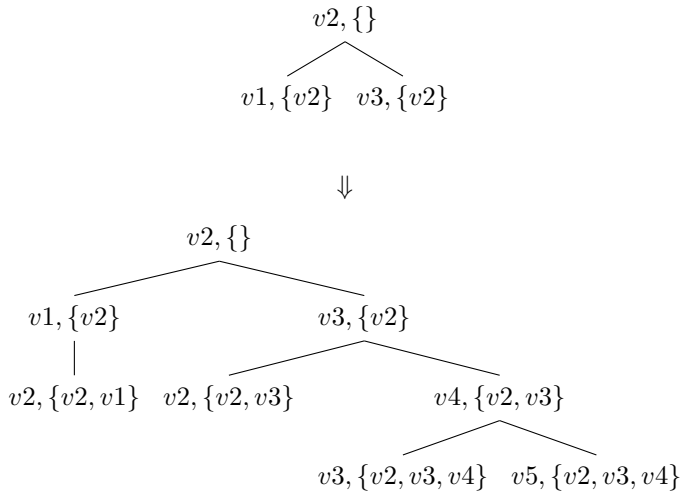

Fig2. Schematic Diagram of How the tree is constructed

(2) *Branching on root:* Add all the nodes which $v2$ has an edge with or simply $v2$ points to as the leaves
(3) For each leaf nodes, record all of its parents
(4) *Branching on leaves:* Take every leaf as the root of a new tree and branch on it by adding all its connecting nodes
(5) *Check for Cycles and Replication:* If some leaf happens to be the root node, then a new cycle is found. If some leaf happens to be one of its parents **other than the root**, then throw this leaf away and prune it.
(6) *Cycling:* Repeating the steps (3),(4),(5) until the tree level has reached the maximal length.

So from the picture, we could enumerate all the cycles less than 4 involving $v2$ as $\{v1, v2\}, \{v2, v3\}$. This holds true because of the commutative properties of a cycle which indicates that

$$\begin{aligned} c =& v_1 \to v_2 \to v_3 \to v_1 \\ =& v_2 \to v_3 \to v_1 \to v_2 \\ =& v_3 \to v_1 \to v_2 \to v_3 \end{aligned}$$

So that means by exploiting the previous process, we could always construct any cycle from the root node **as long as it contains the root node.** Though rigorous proof hasn't been given, it is intuitive enough for us to believe in this point. And the following pseudo-code actually reflects the previous steps.

Here we'll talk about some tricks in the data storage of the cycles. Since each node in the tree has to record its parents in the path, usually it could be accomplished by using arrays. Afraid of the fact that it might go beyond limitation as the cycle number becomes huge, what we actually implement here is **using different bits of a 64-bit integer to represent parents at different levels.** For example, if we put all the vertex in the previous example in the order $v_1 : 1, v_2 : 2, v_3 : 3, v_4 : 4, v_5 : 5$, then the tree should have the form where all the numbers are in hex:
In this way, we could save a lot of memory since the original array would at least cost 8 bytes since the cycle length is at least 2 while this kind of storage only requires 8

ALGORITHM 1: Searching_All_Cycles

Input: input graph $\mathcal{G}$;
a given vertex v in the graph;
the maximal length of a cycle l_m;
Output: the set $\mathbf{c}_{v_0}$ of all the cycles having v_0

```
queue ← empty;
c ← empty;
push_queue(v_0, {}, 0);
repeat
    node ← pop_queue();
    if node.number = root then
        c ← {root, node.parents};
        Add(c, c);
    end
    if node.depth < l_m then
        for each vertex v in V do
            if (connected(v_0, v)) then
                parents ← node.parents + {node.number};
                depth ← node.depth + 1;
                cnode = {v, parents, depth};
                push_queue(cnode);
            end
        end
    end
until queue_is_empty;
```

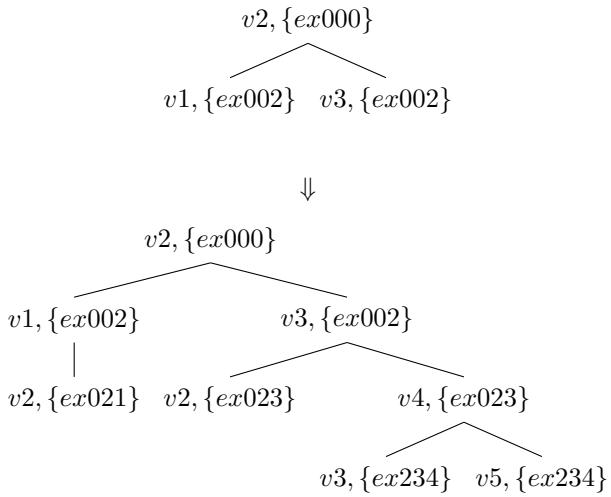

Fig3. Modified Tree Representation

bytes. Also, for this kind of representation, there's a limitation equation

$$\lceil \log_2(|\mathcal{V}|+1) \rceil \times l_m \leq 64$$

For our solver, $l_m = 3, |\mathcal{V}| \leq 511$ and thus

$$\lceil \log_2(|\mathcal{V}|+1) \rceil \times l_m \leq 9 \cdot 3 = 27 < 64$$

which satisfies the constraints.
So this kind of memory allocation is totally effective for our case.

2.2.2. Finding All the Cycles in a Graph. Then it would be quite natural to finish this task by recursively selecting and deleting nodes as is shown in the algorithm below.

ALGORITHM 2: Breadth-First_Search_on_Cycles

Input: l_m the cap of the cycle length;
$\mathcal{G}$ the input graph
Output: **c** the set of all the cycles within the length cap l_m
begin
  c $\leftarrow empty$;
  $\mathcal{G}' \leftarrow \mathcal{G}$;
  repeat
    Select a vertex $v \in \mathcal{G}'$;
    c $\leftarrow$ **c**+ Search_All_Cycles($\mathcal{G}', v, l_m$);
    $\mathcal{G}' \leftarrow \mathcal{G}'/v$
  until $\mathcal{G}' = empty$;
end

ALGORITHM 3: Static_Allocation

Input: l_m: the cycle length cap;
$\mathcal{G}$: the static graph;
Output: **P**: the set of saved patients vertex;
$n_{optimal}$: the optimal value of saved patients;
begin
  c $\leftarrow$ Breadth-First_Search_on_Cycles($l_m, \mathcal{G}$);
  $object \leftarrow 0$;
  for $c \in$ **c do**
    $weight(c) \leftarrow Node\_number(c)$;
    $object \leftarrow object + weight(c) \cdot c$;
  end
  for $v \in \mathcal{V}$ **do**
    $constaints(v) \leftarrow 0$;
    for $c \in$ **c do**
      if $v \in c$ **then**
        $constaints(v) \leftarrow constaints + c$;
      end
    end
  end
  $(\mathbf{P}, n_{optimal}) \leftarrow MIP\_Solver(object, constaints, Binary)$
end

It is obvious that this algorithm is complete.

3. EXPERIMENTAL RESULTS OF STATIC ALLOCATION ON THE ROLE OF ALTRUIST DONOR

Based on the previous steps, we simply take advantage of the existing IP solver Gurobi 5.5 for finding the optimal solution. Using the generator for kidney exchange, we perform an experiment on the results between graph including and without altruists. The percentage of altruists is $\%5$. From it, we could see the role of altruist donor in improving the overall performance of static kidney exchange pool.

Fig. 2. Performance Comparison for $l_m = 3$: Red: 5% Altruists; Blue: No Altruists

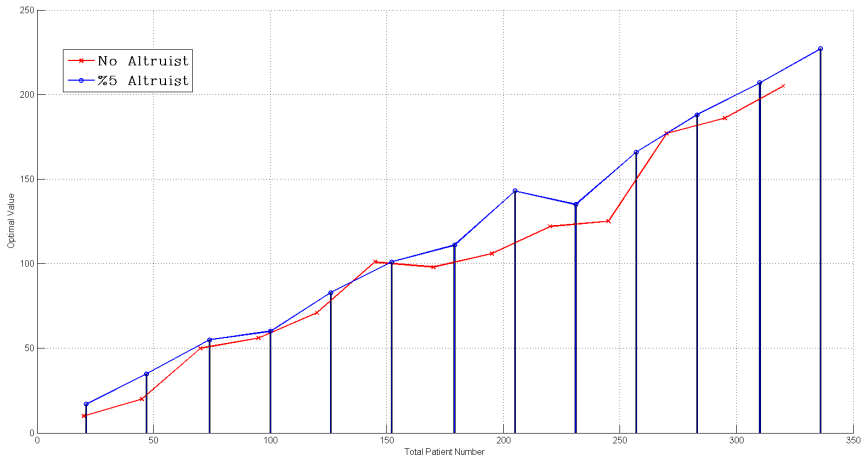

Table I. Results of Static Allocation with and without Altruists

No Altruists		With Altruists	
Total Number	Optimal Saved Number	Total Number	Optimal Saved Number
20	10	21	17
45	20	47	35
70	50	74	55
95	56	100	60
120	71	126	83
145	101	152	101
170	98	179	111
195	106	205	143
220	122	231	135
245	125	257	166
270	177	283	188
295	186	310	207
320	205	336	227

Table II. Percentage Comparison

No Altruist	5% Altruist	Increased Percentage
50%	81%	31%
44%	74%	30%
71%	74%	3%
59%	60%	5%
59%	66%	7%
70%	66%	-4%
58%	62%	4%
55%	58%	3%
51%	65%	14%
65%	66%	1%
63%	67%	4%
64%	68%	4%

From it, we could verify some of conclusions in the literature that the introduction of chain could could affect the overall performance significantly. In order to better prove that, the following table represents the percentage of the saved patients for these two cases.

From it we could see that in average case the ”altruist” donor can be treated as a kind of more efficient vertex in the graph because **there's more nodes connecting to it.**

4. DYNAMIC ALLOCATION SOLVER

When the pool transits from stationary to dynamic state, the situation becomes a bit different. The dominant challenge of dynamic allocation is **the trade-off between looking ahead and being greedy** from our perspective. We know that if the graph is adding more players at each turn, the **totally greedy allocation strategy for it is to perform static allocation recursively** given the current graph structure. But intuitively, it would be much better if some types of patient-donor pairs are saved so that they can trigger longer cycles or chains in the future. Thus, lots of dynamic algorithms are proposed in the literature [4],[2]. [4] proposes a sampling-based method in order to maximize the expected value of saved patients *given the probabilistic distribution of future input.* While in[2], the weights of different edge types are modified to **make bias towards some types of patients for future use** from learning.
Inspired by this kind of ”learning-biased” method, we come up with an online-learning based method for dynamic allocation. Its basic idea is just to **decide when to look ahead or become greedy based on some heuristics and conclusions.** Here's some assumptions and concerning comments of our dynamic algorithm:

(1) No altruists at each input: Since our solver isn't fast enough to handle large-scale instances, this assumption could spare lots of memory.
(2) Short cycles with length no greater than 3 suffices to make efficient allocations: This is a conclusion cited from [3] while it might not be the cased in real applications
(3) The compatibility problem of kidney exchange can be simplified as **only blood type compatible problem**: this assumption isn't quite proper in real practice because the tissue compatibility and the relationship between patient and live donor should also be taken into account.
(4) The distribution of each input is constant and all the inputs of the graph are $i.i.d$: it has the same problem as the previous one. But the static distribution is easily to quantify from a statistical approach. Also, the instance generator we utilize here is also based on constant distribution. So, it could be satisfied in simulation.

4.1. Online Learning Potential

In [2], a special concept ”potential” for different types of patient-donor pair is exploited to guide bias in the allocation. And this concept is learned offline based on a training set using optimization algorithm. As for its meaning, the potential of a given patient-donor pair reflects **its ability of triggering longer cycles in the future input**. From our heuristics, the potential is interpreted as the expected number of vertex in the next input which have an edge with or connect this type of vertex. Thus, if A blood-type patient is the most popular in a given region, then the pair with A type donor would be preferred since it is more likely to have more edges in the next input according to assumption (3). In addition, we know that the distribution is constant and the central limit theorem would guarantee that the frequency-based methods would converge to the actual distribution as the graph becomes larger. Summing up these statements, the following learning-based method of potential is proposed:
It's a just simple method according to our heuristics though problematic.

ALGORITHM 4: Online_Learning_Potential

Input: $input$: the current input;
n_A, n_B, n_{AB}, n_O: the accumulated number of patients of different blood types;
n_0: the total number of patients
Output: p_A, p_B, p_{AB}, p_O: the learned potential for corresponding donor with the same type
begin
$n_A \leftarrow n_A + input.patients\_number(A)$;
$n_B \leftarrow n_B + input.patients\_number(B)$;
$n_{AB} \leftarrow n_{AB} + input.patients\_number(AB)$;
$n_O \leftarrow n_O + input.patients\_number(O)$;
$n_0 \leftarrow n_0 + input.total\_patients\_number$;
$p_A \leftarrow n_A/n_0$;
$p_B \leftarrow n_B/n_0$;
$p_{AB} \leftarrow n_{AB}/n_0$;
$p_O \leftarrow n_O/n_0$;
end

4.2. Decide When to be Greedy and Look ahead

The learned potential can be utilized to look ahead according to our assumptions. We also assign additional parameters to decide when to become greedy. Because in real applications, time would become an important issue for many patients. After a certain period, it would be reasonable to make a greedy search for all the patients joined in previously. So here's the algorithm for the whole online dynamic allocation process.

4.3. Performance Evaluation

4.3.1. Comparison between Myopia and Dynamic Allocation. Using this algorithm, we have generated some results of the myopia allocation and our weighted allocation laws below for better comparison. The instance we exploit here is still the standard generator.

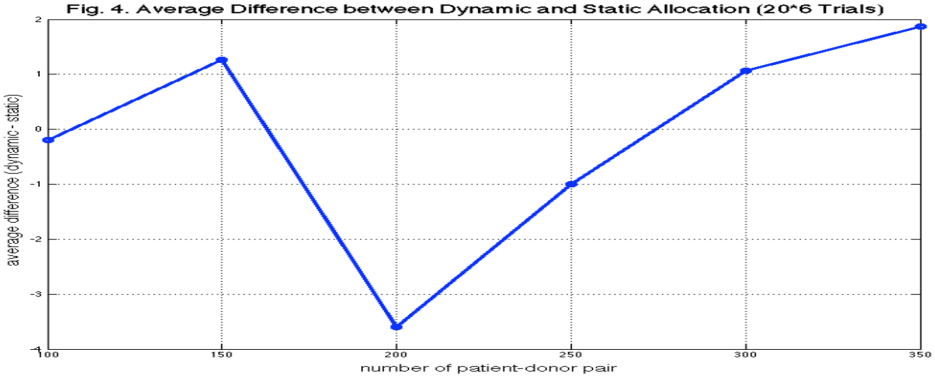

Fig. 3. Average Difference between Myopia and Dynamic Solvers

From it we could see that the performance of dynamic solver isn't better than the myopia one for small-scale problem, but it seems capable of saving more patients as the pool becomes larger. Since we are attempting to learn potential based via learning

ALGORITHM 5: Potential_Guided_Allocation

Input: l_m: the cycle length cap;
$\mathcal{G}$: the remaining graph;
$\{n_A, n_B, n_{AB}, n_O\}$: the accumulated frequency of each type;
n_0: the total patients' number
Output: **P**: the set of saved patients vertex;
$n_{optimal}$: the optimal value of saved patients;
begin
  c $\leftarrow$ Breadth-First_Search_on_Cycles($l_m, \mathcal{G}$);
  $\{p_A, p_B, p_{AB}, p_O\} \leftarrow$ Online_Learning_Potential($\mathcal{G}, \{n_A, n_B, n_{AB}, n_O\}, n_0$);
  $object \leftarrow 0$;
  for $c \in$ **c do**
    $weight(c) \leftarrow 0$;
    for $v_1, v_2 \in c, connect(v_1, v_2)$ **do**
      $weight(c) \leftarrow weight(c) + \underbrace{1 - \frac{1}{2}(p_{v1.patient\_type} + p_{v2.patient\_type})}_{biased\ weight}$;
    end
    $object \leftarrow object + weight(c) \cdot c$;
  end
  for $v \in \mathcal{V}$ **do**
    $constaints(v) \leftarrow 0$;
    for $c \in$ **c do**
      if $v \in c$ **then**
        $constaints(v) \leftarrow constaints + c$;
      end
    end
  end
  $(\mathbf{P}, n_{optimal}) \leftarrow MIP\_Solver(object, constaints, Binary)$
end

the probabilistic distribution, the randomness of problem could affect this solver to a great extent if the problem size is very small.

4.4. Tuning Parameter β

Since the parameter β in this method could be changed arbitrarily, we'd like to test its effects on the improvement of the overall performance. So the previous instances are utilized here for testing the performance of dynamic solver with different values of β. Here's the results.

From the table, the largest value has the best performance which is reasonable since we have saved many high-potential patients in the previous turns that could trigger more cycles for overall performance.

5. CONCLUSION

In this project, we have made two solvers for kidney exchange based on the current MIP solver: static and dynamic solver. The static solver could give the optimal value of a stationary compatibility based on the method of "branching on cycles". Using it, the importance of "altruist" donor in practice can be proved experimentally. The dynamic solver could be exploited for the dynamic graph where the patient-donor pairs keep coming in and out. It attempts to learn the "potentials" of different types of patient-donor pairs during the whole process based on some assumptions which seem quit

ALGORITHM 6: Online_Dynamic_Allocation

Input: Input set $\{g_{i1}, g_{i2}, \cdots, g_{in}\}$;
The greedy time period β;
The length cap l_m;
Output: The total number saved n_{do}
begin
$n_{do} \leftarrow 0$;
$\mathcal{G} \leftarrow empty$;
$\{n_A, n_B, n_{AB}, n_O\} \leftarrow \{0, 0, 0, 0\}$;
$n_0 \leftarrow 0$;
for $j \leftarrow 1$ *to* n **do**
$\mathcal{G} \leftarrow \mathcal{G} + g_{ij}$;
$\{n_A, n_B, n_{AB}, n_O\} \leftarrow$ Online_Learning_Potential($g_{ij}, \{n_A, n_B, n_{AB}, n_O\}, n_0$);
if $remainder(j/\beta)! = 0$ *and* $j < n$ **then**
$(\mathbf{c_j}, n_{oj}) \leftarrow$ Potential_Guided_Allocation($l_m, \mathcal{G}, \{n_A, n_B, n_{AB}, n_O\}, n_0$);
end
else
$(\mathbf{c_j}, n_{oj}) \leftarrow$ Potential_Guided_Allocation($l_m, \mathcal{G}$);
end
$n_{do} \leftarrow n_{do} + n_{oj}$;
$\mathcal{G} \leftarrow \mathcal{G}/\mathbf{c_j}$;
end
end

Table III. Average Optimal Number of Myopia and Dynamic Allocation for 20 Instances

l_m	3			
β	4			
Number per input	50			
Number of Patients	Myopia	Dynamic	Average Difference	Standard Deviation
100	39.87	39.67	-0.2	2.3664
150	69.8	71.1	1.26	2.96
200	99.2	95.6	-3.6	4.8
250	129	128	-1	7.44
300	163.7	164.7	1	6.85
350	197.2	199.1	1.9	8.08

Table IV. Performances for Different Values of β

l_m	3	
Number per input	50	
Number of Patients	350	
β	Average	Standard Deviation
2	197.8	11.296
3	197.2	10.53
4	197.13	10.9926
5	198.6	11.975

strong. The results show that it does have some tiny improvements over the myopia allocation one which treats the graph as static each time.

REFERENCES

D.J. Abraham, A. Blum, T. Sandholm, "Clearing Algorithms for Barter Exchange Markets: Enabling Nationwide Kidney Exchange", *ACM*, 2007.

J.P. Dickerson, A.D. Procaccia, T. Sandholm, "Dynamic Matching via Weighted Myopia with Application to Kidney Exchange", *AAAI*, 2012.

J.P. Dickerson, A.D. Procaccia, T. Sandholm, "Optimizing Kidney Exchange with Transplant Chains: Theory and Reality", *AAMAS*, 2012.

P. Awasthi, T. Sandholm, "Online Stochastic Optimization in the Large: Application to Kidney Exchange",*IJCAI*, 2009.