Kidney Paired Donation

Problem and Context

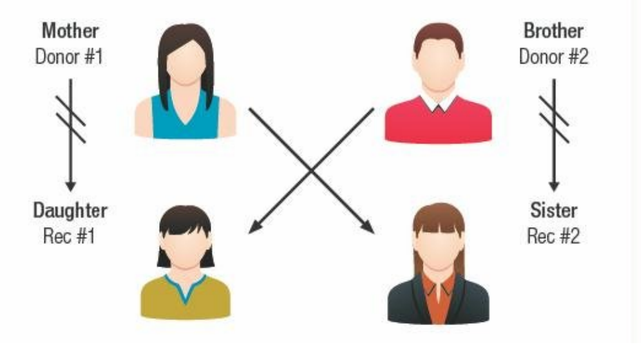
BBS
MULTIPLYING OPPORTUNITIES

Context

Let's consider a problem from the healthcare domain

...And in particular kidney transplantation from living donors

- Incompatibility issues are major bottleneck, putting lives at risk
- ...But sometimes we are in this kind of situation:



- There are two willing donor, with incompatible recipients
- ...But we can perform both transplants if we make an exchange!

KPD Operation

Operationally, it works as follows:

- Recipient-donor pairs enter a kidney paired donation program
- Periodically, the pairs must be matched so as to enable transplantation
- ...Then all planned surgeries are performed within a short time time frame

We can chain together more than two pairs

- E.g. $d_A \rightarrow r_B, d_B \rightarrow r_C, d_C \rightarrow r_A$

...But usually not too many

- Surgeries are then performed in short order
- ...Since even one withdrawn donor causes the whole exchange to fail

KPD Description

So, we can say the following about our requirements:

Given a set of donor-recipient pairs:

- We want to select several groups of pairs
- Every group should correspond to a viable set of exchanges
- Groups should include at most C pairs
- No patient should be included in two groups
- We want to perform as many transplants as possible

KPD Description

So, we can say the following about our requirements:

Given a set of donor-recipient pairs:

- We want to select several groups of pairs
- Every group should correspond to a viable set of exchanges
- Groups should include at most C pairs
- No patient should be included in two groups
- We want to perform as many transplants as possible

Even if we have data, this is not a Machine Learning problem

- Our key concern is no estimation, or diagnosis
- ...But rather making optimal decisions under constraints

KPD Management

Managing a KPD program is hard

- The wait list for kidney transplats grew by $> 44,000$ units in 2023
- They are not all for KPD, but the number is still large

We cannot plan exchanges for such numbers by hand

...But we could use a decision support tool

KPD Management

Managing a KPD program is hard

- The wait list for kidney transplats grew by $> 44,000$ units in 2023
- They are not all for KPD, but the number is still large

We cannot plan exchanges for such numbers by hand

...But we could use a decision support tool

There are AI technique that can help us doing that

- However, before we can start developing an approach
- ...We need to find a way to model our system

Kidney Paired Donation

Problem Formalization

BBS
MULTIPLYING OPPORTUNITIES

What do we Need to Define?

Let's check again our requirements

Given a set of donor-recipient pairs:

- We want to select several groups of pairs
- Every group should correspond to a viable set of exchanges
- Groups should include at most C pairs
- No patient should be included in two groups
- We want to perform as many transplants as possible

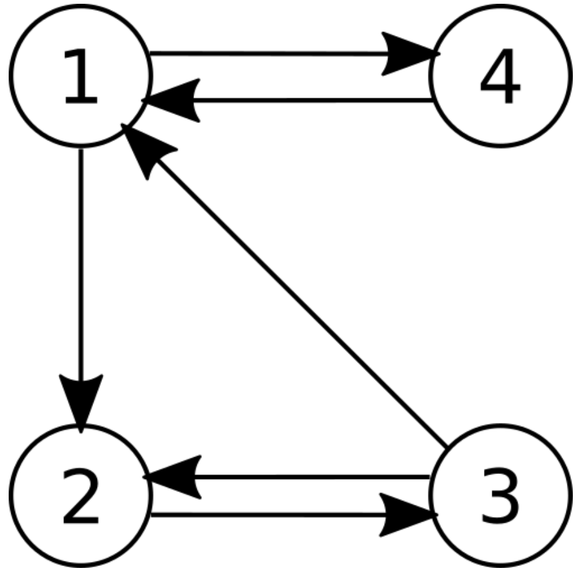
Most of them are relatively clear

...Except for the one about "viable" sets of exchanges

- We'll need to formalize what we mean by that

Compatibility Graph

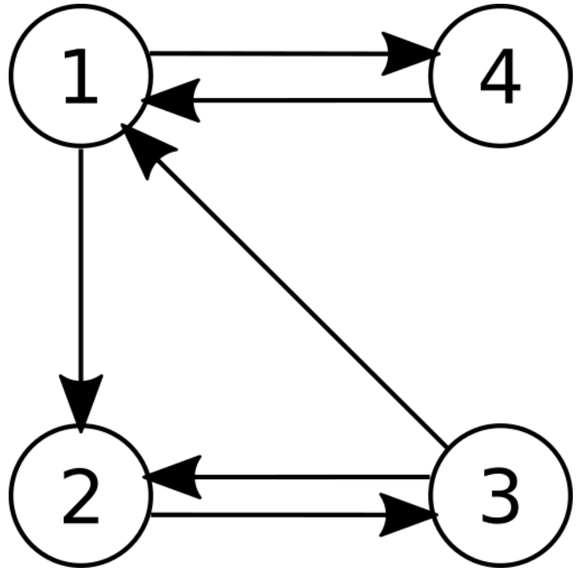
The KPD entities and their relation can be represented as a graph



- Recipient-donor pairs (r_i, d_i) in the programs can be seen as nodes in a graph
- The graph contains an arc from pair i to pair j iff d_i is compatible with r_j
- In the example there are four pairs
- The donor in pair 1 is compatible with the recipient in pair 2, and so on

Viable Exchanges as Cycles

In this representations, kidney exchanges correspond to cycles



- For example $\{1, 2, 3\}$ defines a valid cycle
- ...Corresponding to the exchange $d_1 \rightarrow r_2, d_2 \rightarrow r_3, d_3 \rightarrow r_1$
- ...And leading to 3 transplants

A Better Problem Formulation

This is enough to refine our problem formulation

- We want to select sets of nodes
- Every set should correspond to a cycle
- A set can include at most C nodes
- No node can be included in two sets
- We want to maximize the sum of the sizes of the selected sets

A Better Problem Formulation

This is enough to refine our problem formulation

- We want to select sets of nodes
- Every set should correspond to a cycle
- A set can include at most C nodes
- No node can be included in two sets
- We want to maximize the sum of the sizes of the selected sets

This information is precise enough to:

- Define a mathematical model for our decisions, constraints, and objective
- Use an optimization tool to find an optimal solution

Kidney Paired Donation

Building a Mathematical Model

BBS
MULTIPLYING OPPORTUNITIES
BOLOGNA BUSINESS SCHOOL

A Guidline for Optimization Modeling

When building an optimization model:

- Start by choosing how to model the decisions
- Then, consider the constraints one by one
  - Define how to model then with the chosen variables
  - Introduce additional variable as needed
- Then, do the same for the problem objective

During this process, it is very common to have difficulties

When that happens, try thinking about:

- Alternative ways to formulte the constraints
- ...But even more, alternative ways to represent decisions

Our decision variables need to identify groups of nodes

Can you think of some possible design choices?

Cycle Formulation

We'll use a binary x_j variable for every cycle in the graph

- $x_j = 1$ iff the j-th cycle is chosen for surgery, and 0 otherwise
- With this notation, we only select valid exchanges

...But what about the other constraints?

"No node can be included in two groups":

$$\sum_{j=1}^{n} a_{ij} x_{ij} \leq 1 \qquad \forall i = 1..m$$

- n is the number of cycles, m is the number of nodes
- $a_{ij} = 1$ if node i is in cycle j

Cycle Formulation

"Too large groups/cycles should not be considered":

- We do not need an equation for this
- ...Since we can simply avoid building variables for those cycles

"We want to maximize the total number of transplants":

$$\max \sum_{j=1}^{n} w_j x_{ij}$$

- w_j is the number of transplants associated to cycle j
- This is our objetive function

Cycle Formulation

Therefore, the cycle formulation consists in the Mathematical Model

$$\max \sum_{j=1}^{n} w_j x_j$$

$$\text{s.t.} \sum_{j=1}^{n} a_{ij} x_j \leq 1 \qquad \forall i = 1..m$$

$$x_j \in \{0, 1\} \qquad \forall j = 1..n$$

This is an example of an Integer Linear Program

- I.e. a mathematical model with integer variables
- ...And linear constraints and cost

Declarative Optimization

ILPs are a sub-case of Declarative Optimization Approaches

The main idea in declarative optimization is to:

- Define a model by having and domain and optimization expert cooperate
- Use general solution algorithm to obtain optimal solutions

It's not one of the AI techniques in the spotlight right now, but it can be very useful!

Just beware that several of the problems tackled by these approaches are very difficult

- There may be trillions or more of possible solutions to explore
- ...But the solution approaches are much more clever than simple enumeration

BBS

Kidney Paired Donation

Implementing the Cycle Formulation

BBS
MULTIPLYING OPPORTUNITIES

Generating the Benchmark

We will use synthetic data for this use case

In the real world, compatibility is determined by:

- The blood type of the donor and the recipient
- A number of very variable factors linked to their immune systems

Accordingly, the benchmark generation algorithm works by:

- Building a fixed number of pairs
- Assigning blood types to each pair
- Building compatibility arcs based on blood types

Generating the Benchmark

In particular, we only focus on incompatible blood types

```
In [2]: pairs, arcs, aplus = util.generate_compatibility_graph(size=12, seed=2)
        pairs

Out[2]: {0: pair(recipient='B+', donor='A+'),
         1: pair(recipient='B+', donor='A+'),
         2: pair(recipient='O+', donor='B+'),
         3: pair(recipient='A+', donor='B+'),
         4: pair(recipient='O+', donor='A+'),
         5: pair(recipient='O+', donor='A-'),
         6: pair(recipient='A-', donor='O+'),
         7: pair(recipient='A+', donor='B+'),
         8: pair(recipient='B+', donor='A+'),
         9: pair(recipient='O+', donor='A+'),
         10: pair(recipient='O+', donor='A+'),
         11: pair(recipient='A-', donor='A+')}
```

- Compatible pairs would not need to go through a KPD program
- The blood type prevalence reflects the Italian distribution
- In the pairs, we are neglecting all other factors that impact compatibility

Generating the Benchmark

Arcs are first determined based on blood type compatibility

...Then a small (random) fraction of them (5%) is removed

```
In [3]: aplus

Out[3]: {0: [3, 7],
         1: [3, 7],
         2: [0, 1, 8],
         3: [0, 1, 8],
         4: [3, 7],
         5: [3, 6, 7, 11],
         6: [0, 1, 2, 3, 4, 5, 7, 8, 9, 10],
         7: [0, 1, 8],
         8: [3, 7],
         9: [3, 7],
         10: [3, 7],
         11: [3, 7]}
```

- This simulated the other compatibility factors
- ...Which are therefore accounted for at the graph level

Enumerating Cycles

Building our ILP requires to enumerate all cycles in the graph

...Up to a maximum size of C, that is

```
In [4]: cycles = util.find_all_cycles(aplus, max_length=4, cap=None)
        for c in cycles: print(set(c))
```

```
{8, 1, 3, 7}
{0, 7}
{8, 3}
{0, 1, 3, 7}
{0, 3}
{0, 8, 3, 7}
{8, 1, 3, 7}
{1, 7}
{5, 6}
{8, 7}
{0, 1, 3, 7}
{1, 3}
{0, 8, 3, 7}
```

Therse are the cycles for the small graph we are using now, with $C = 4$

BBS

Cycle Formulation - Implementation

Once we have all cycles, we can build the Cycle Formulation model

- The code can be found in the `notebooks/util/util_kep.py` file
- We are going to inspect it, just to have an idea of how these models are built

All the work is done by a single function:

```
def cycle_formulation(pairs, cycles, tlim=None, verbose=1):
    infinity, ncycles, npairs = slv.infinity(), len(cycles), len(pairs)
    slv = pywraplp.Solver.CreateSolver('CBC') # Build the solver
    ...
```

- We start by building a solver object
- We use the CBC solver, via Google OR-Tools
- It's the fastest MIP solver with a fully permissive license

BBS

Cycle Formulation - Implementation

Variables are built with `IntVar`, constraints posted with `Add`

```
def cycle_formulation(pairs, cycles, tlim=None, verbose=1):
    ...
    x = [slv.IntVar(0, 1, f'x_{j}') for j in range(ncycles)] # variables
    for i in range(npairs): # constraints
        slv.Add(sum(x[j] for j in cpp[i]) <= 1)
    ...
```

We set the objective with `Maximize` or `Minimize`

```
def cycle_formulation(pairs, cycles, tlim=None, verbose=1):
    ...
    slv.Maximize(sum(len(c) * x[j] for j, c in enumerate(cycles))) # objective
    if tlim is not None: # time limit
        slv.SetTimeLimit(1000*tlim)
    ...
```

- Time limits are enforced with `SetTimeLimit`

Solving the Problem

We can now solve the model to obtain a solution:

In [5]:
```
pairs, arcs, aplus = util.generate_compatibility_graph(size=12, seed=2)
cycles, ctime = util.find_all_cycles(aplus, max_length=4, cap=None, return_wall_time=True)
print(f'Size 12, number of cycles: {len(cycles)}, time: {ctime:.3f} s')
sol, stime, _ = util.cycle_formulation(pairs, cycles, tlim=10, verbose=0)
print(f'Size 12, time: {stime:.3f} s, Solution:', {k for k, v in sol.items() if v != 0 and k != 'objective'})
for k, v in sol.items():
    if k != 'objective' and v > 0:
        print(f'{k} corresponds to:', cycles[int(k.split('_')[1])])
```

```
Size 12, number of cycles: 13, time: 0.001 s
Size 12, time: 0.031 s, Solution: {'x_2', 'x_1', 'x_8'}
x_1 corresponds to: (0, 7)
x_2 corresponds to: (3, 8)
x_8 corresponds to: (5, 6)
```

- Every selected variable corresponds to a cycle
- ...And therefore to a viable set of exchanges

Scalability

As a long as th cycle is not too large, we can get the optimal solution quickly

In [6]:
```
pairs, arcs, aplus = util.generate_compatibility_graph(size=150, seed=2)
cycles, ctime = util.find_all_cycles(aplus, max_length=4, cap=None, return_wall_time=True)
sol, stime, _ = util.cycle_formulation(pairs, cycles, tlim=10, verbose=0)
print(f'Size 150, number of cycles: {len(cycles)}, enumeration time: {ctime:.3f} s, solution time: {stime:.3f} s')
```

```
Size 150, number of cycles: 43206, enumeration time: 6.599 s, solution time: 1.632 s
```

...But the complexity grows quickly with the graph size:

In [7]:
```
pairs2, arcs2, aplus2 = util.generate_compatibility_graph(size=200, seed=2)
cycles2, ctime2 = util.find_all_cycles(aplus2, max_length=4, cap=None, return_wall_time=True)
sol2, stime2, _ = util.cycle_formulation(pairs2, cycles2, tlim=10, verbose=0)
print(f'Size 200, number of cycles: {len(cycles2)}, enumeration time: {ctime2:.3f} s, solution time: {stime2:.3f} s')
```

```
Size 200, number of cycles: 114788, enumeration time: 25.387 s, solution time: 5.954 s
```

BBS

Improving Scalability

There are approaches that can be used to dramatically improve scalability

...They are however quite complex, so we'll just see the end result for 200- and 600-node graphs

In [8]:
```
%%time
pairs3, arcs3, aplus3 = util.generate_compatibility_graph(size=200, seed=2)
cycles_cg3, _  = util.cycle_formulation_cg(pairs3, aplus3, max_len=4, itcap=10, verbose=0)
sol3, _, _ = util.cycle_formulation(pairs3, cycles_cg3, tlim=30, verbose=0)
print(f'Number of transplants: {sol3["objective"]}')
```

```
Number of transplants: 74.0
CPU times: user 784 ms, sys: 0 ns, total: 784 ms
Wall time: 784 ms
```

In [9]:
```
%%time
pairs3, arcs3, aplus3 = util.generate_compatibility_graph(size=600, seed=2)
cycles_cg3, _  = util.cycle_formulation_cg(pairs3, aplus3, max_len=4, itcap=10, verbose=0)
sol3, _, _ = util.cycle_formulation(pairs3, cycles_cg3, tlim=30, verbose=0)
print(f'Number of transplants: {sol3["objective"]}')
```

```
Number of transplants: 229.0
CPU times: user 9.42 s, sys: 19.7 ms, total: 9.44 s
Wall time: 9.46 s
```

BBS