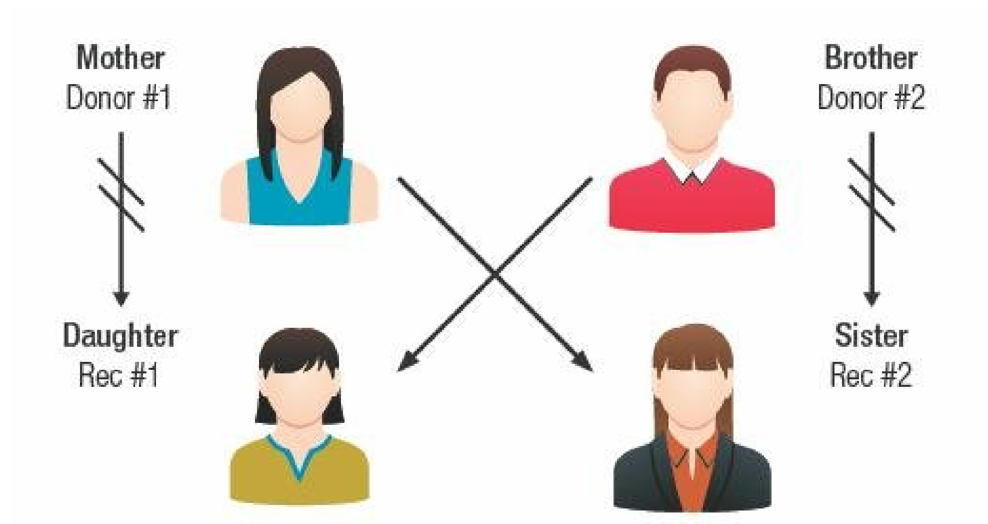
Kidney Paired Donation

Kidney Paired Donation

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!

Kidney Paired Donation

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. $d_A \to r_B, d_B \to r_C, d_C \to r_A$

...But usually not too many

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- ...Since even one withdrawn donor causes the whole exchange to fail

Kidney Paired Donation

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

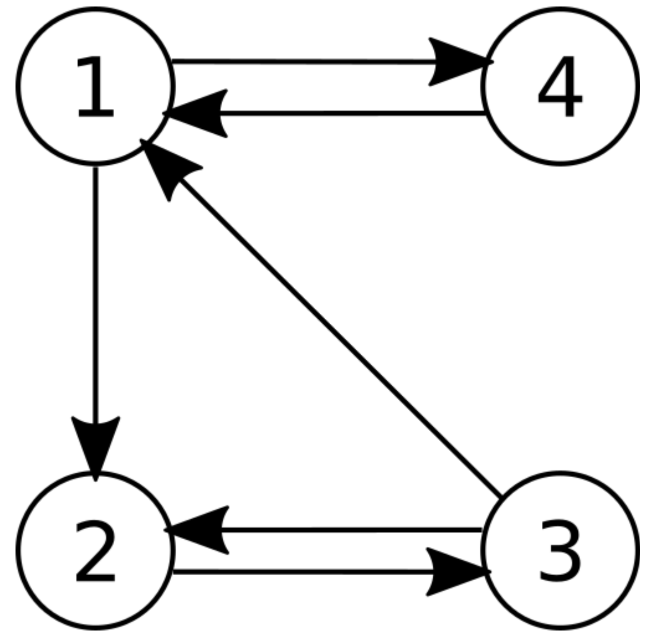
We cannot plan exchanges for such numbers by hand
...But we could use a decision support tool

The matching problem is know as Kidney Exchange Problem (KEP)

- We want to choose groups of pairs for exchanges
- ...And typically to maximize the number of transplants

Problem Formulation

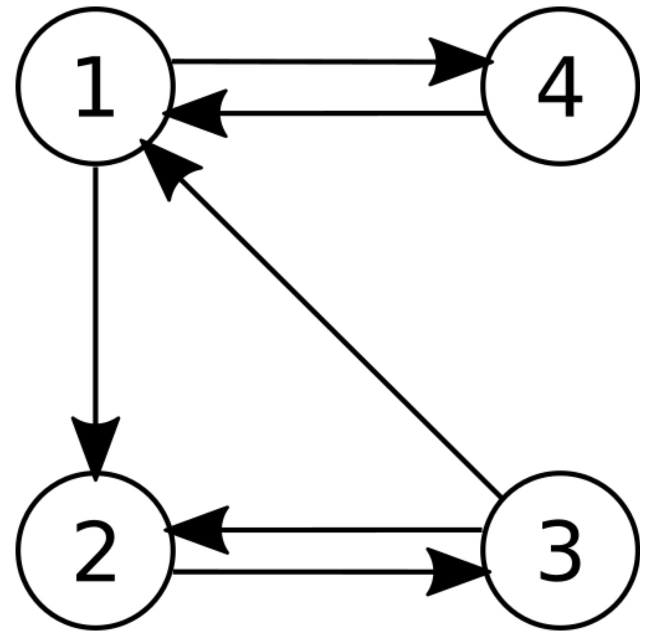
The KEP admits a graph-based formulation



- 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 the 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

Problem Formulation

In this representations, exchanges are 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

Problem Formulation

This is enough to start defining a combinatorial optimization problem

- We want to select groups of nodes
- No node can be included in two groups
- Too large groups/cycles should not be considered
- Every group should correspond to a cycle
- A group/cycle with n nodes lead to n transplants
- We want to maximize the total number of transplants

Now we do we turn this into a formal optimization model

What do we start with?

A Guidline for Optimization Modeling

Whem building a CO model, this is usually a good approach:

- 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?

Assignment-Base Formulation for the KEP

We could use binary variables x_{ij}

- $x_{ij} = 1$ iff node i is part of the j-th cycle
- For m nodes, we can have at most $n = \lfloor {}^{m}/_{2} \rfloor$ cycles

Now we can attempt to formulate the constraints

Assignment-Base Formulation for the KEP

We could use binary variables x_{ij}

- $x_{ij} = 1$ iff node i is part of the j-th cycle
- For m nodes, we can have at most $n = \lfloor {}^{m}/_{2} \rfloor$ cycles

Now we can attempt to formulate the constraints

can be included in two groups":

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

Assignment-Base Formulation for the KEP

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

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

Assignment-Base Formulation for the KEP

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

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

"Every group should correspond to a cycle"

This is a tricky constraint to handle

- In Mathematical programming, it is hard to find a compact model
- Some Constraint Programming solver provide support for that
- ...But even those are pretty hard to find

Cycle Formulation

We'll circumvent the issue by changing our decision variables

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
- With this formulation, groups are cycles by construction

What about the other constraints?

"No node can be included in two groups":

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

- $a_{ij} = 1$ if node i is in cycle j
- This is basically a mutual exclusion constraint

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 following Integer Program

$$
\begin{aligned}
\max \sum_{j=1}^{n} w_j x_j & \\
\text{s.t.} \sum_{j=1}^{n} a_{ij} x_j \leq 1 & \qquad \forall i = 1..m \\
x_j \in \{0, 1\} & \qquad \forall j = 1..n
\end{aligned}
$$

- m is the number of pairs, n of cycles
- w_j is the weight of cycle j (i.e. its number of nodes)
- $a_{ij} = 1$ iff node i belongs to cycle j (and $a_{ij} = 0$ otherwise)
- The maximum length constraint is handle when generating the set of cycles

Implementing the Cycle Formulation

Generating the Benchmark

We will try to build a cycle formulation approach

...But first we need to obtain a benchmark (a dataset)

- We will use synthetic data, obtain via the following function:

```
In [7]: pairs, arcs, aplus = util.generate_compatibility_graph(size=12, seed=2)
```

- The function generates a fixed number of pairs
- ...And their compatibility graph

The approach is designed to be reasonably realistic

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

Generating the Benchmark

The generated pairs are associated to incompatible blood types

```
In [8]: pairs

Out[8]: {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 [9]: aplus

Out[9]: {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

We enumerate cycles using simple Depth First Search with limited depth

```python
def cycle_next(seq, nsteps, aplus, cycles, cap=None):
    node = seq[-1]
    successors = np.array(aplus[node]) # Consider all possible successors
    np.random.shuffle(successors) # ...in randomized order
    for dst in successors:
        # Early exit if the capacity has been exceeded
        if cap is not None and len(cycles) >= cap: return
        if dst == seq[0] and dst == min(seq): # close the cycle
            cycles.add(tuple(seq))
        elif nsteps > 0 and dst not in seq:
            cycle_next(seq+[dst], nsteps-1, aplus, cycles, cap) # recursive call
```

- Cycles are stored as tuples, which mean that the node ordering matters
- ...So we take only the ordering that starts with the minimum index
- There is a capacity parameter to limit the number of enumerated cycles

Enumerating Cycles

We use a second function to start the enumeration from all possible sources

```python
def find_all_cycles(aplus, max_length, cap=None, seed=42):
    cycles = set()
    roots = np.array(list(aplus.keys()))
    np.random.seed(seed)
    np.random.shuffle(roots)
    for node in roots:
        if cap is None or len(cycles) < cap:
            cycle_next([node], max_length-1, aplus, cycles, cap)
    return list(cycles)
```

We can now enumerate the cycles for our graph (HP: max length of 4)

In [13]:
```python
cycles = util.find_all_cycles(aplus, max_length=4, cap=None)
print(sorted(cycles))
```

```
[(0, 3), (0, 3, 1, 7), (0, 3, 8, 7), (0, 7), (0, 7, 1, 3), (0, 7, 8, 3), (1, 3), (1, 3, 8, 7), (1, 7), (1, 7, 8, 3), (3, 8), (5, 6), (7, 8)]
```

Cycle Formulation - Implementation

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

```python
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
    cpp = {i:[] for i in range(npairs)} # group cycles by pair
    for j, cycle in enumerate(cycles):
        for i in cycle: cpp[i].append(j)
    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)
    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)
    status = slv.Solve() # solve
    # Extract results and return
    ...
```

Cycle Formulation - Implementation

We use the CBC solver, via Google OR-Tools

```
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
    ...
```

- It's the fastest MIP solver with a fully permissive license

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)
    ...
```

- The `cpp` dictionary contains cycles, grouped by the pair/node they use

Cycle Formulation - Implementation

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

```python
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`

We can now solve the cycle formulation:

In [23]:
```python
pairs, arcs, aplus = util.generate_compatibility_graph(size=12, seed=2)
cycles = util.find_all_cycles(aplus, max_length=4, cap=None)
sol, tme, _ = util.cycle_formulation(pairs, cycles, tlim=10, verbose=1)
print({k for k, v in sol.items() if v != 0 and k != 'objective'})
```

```
Solution time: 0.002 sec, objective value: 6.0 (optimal)
{'x_2', 'x_1', 'x_8'}
```