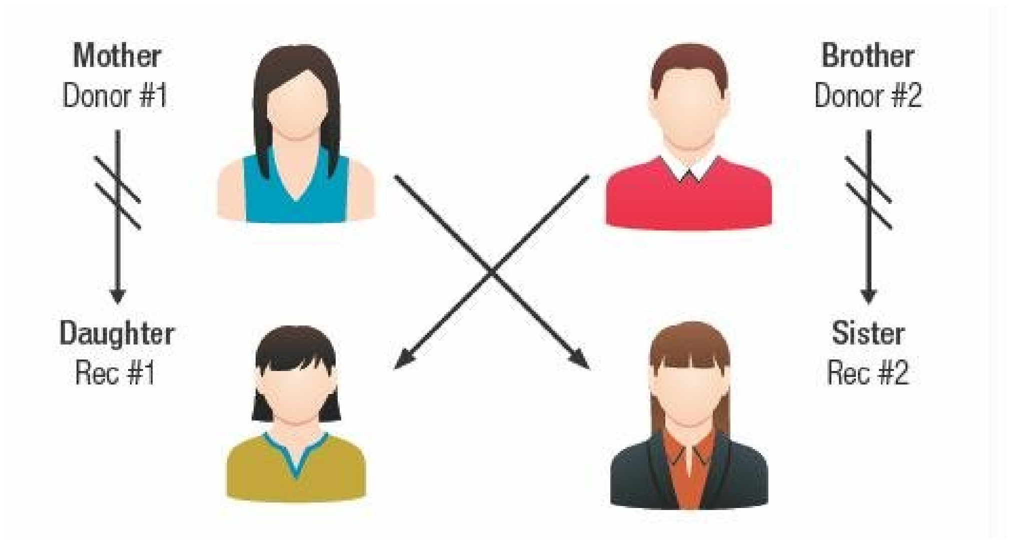
Kidney Exchange Problem

Kidney Exchange

Let's consider now a problem from a different (medical domain)

We consider organ transplantation from living donors (e.g. kidney)

- Incompatibility issues are major bottleneck
- ...But sometimes we are in this kind of situation:



- There are two willing donor, with incompatible recipients
- ...But an exchange can be performed!

Kidney Exchange

Operationally, it works as follows:

- Recipient-donor pairs enter a kidney exchange program
- Periodically, the pairs must be matched so as to enable transplantation
  - Matches can be simple (2 pairs) or more complex (3 or more pairs)
- Surgery is performed within a short time time frame
  - Usually this sets a limit on the size of each chain

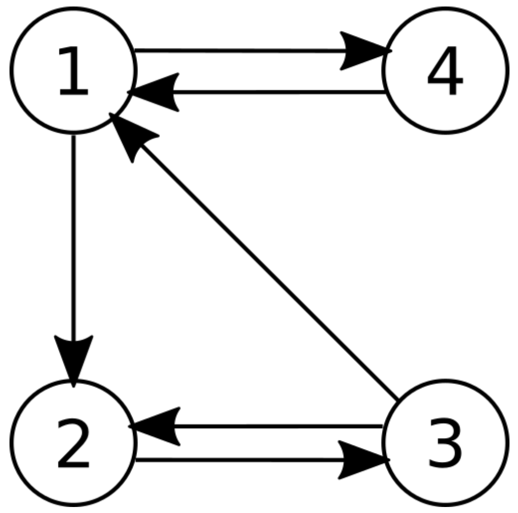
The matching problem is know as Kidney Exchange Problem (KEP)

- Consists in assigning donor to recipients so as to form closed chains
- ...And so as to maximize (typically) the number of transplants
- It is an online problem, but we will tackle in its offline version

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

From this perspective, the KEP consists in selecting cycles in the graph

- Only cycles up to a maximum length are considered
- The weight of a cycle is given by its number of nodes/arcs
- The objective is to maximize the weight of the selected cycles
- Cycles are mutually exclusive if they share at least one node

The key ideas are clear, but translating them to a model is another story

- Defining a combinatorial model for building cycles can be complicated
  - E.g. Traveling Salesaman Problem, all Vehicle Routing Problem variants...
- But what if we had access to a precomputed set of cycles?

This is the key idea in the so-called cycle formulation

Cycle Formulation

The cycle formulation consists in the following Integer Program

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

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

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

- 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

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:

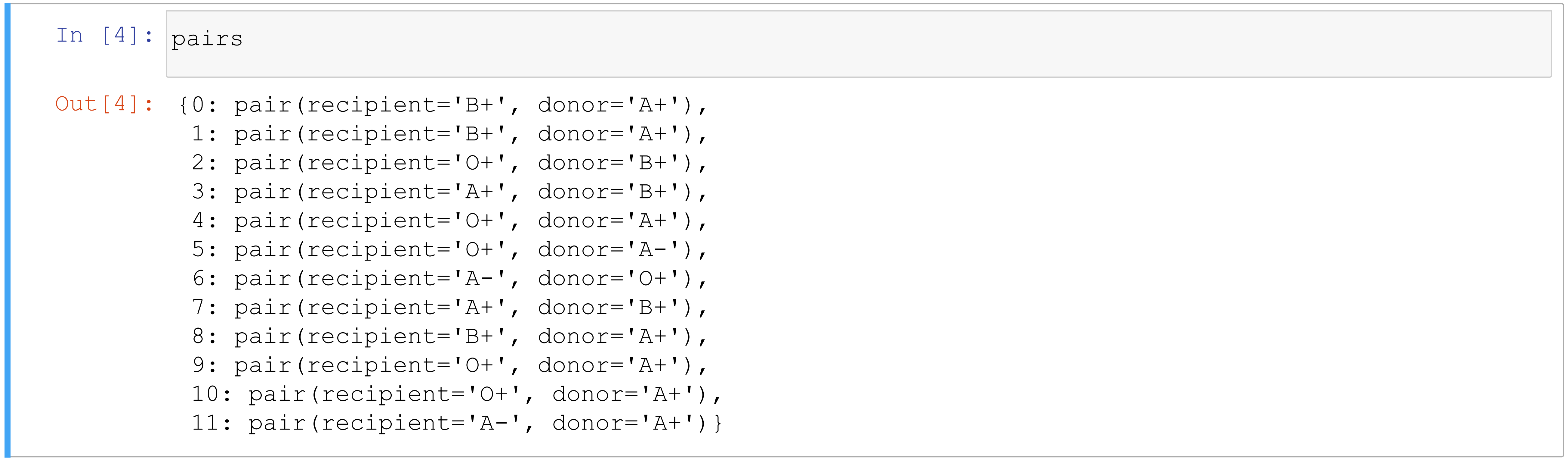
```
In [3]: pairs, arcs, aplus = er.generate_compatibility_graph(size=12, seed=2)
```

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

The approach is designed to be reasonably realistic

Generating the Benchmark

The generated pairs are associated to incompatible blood types



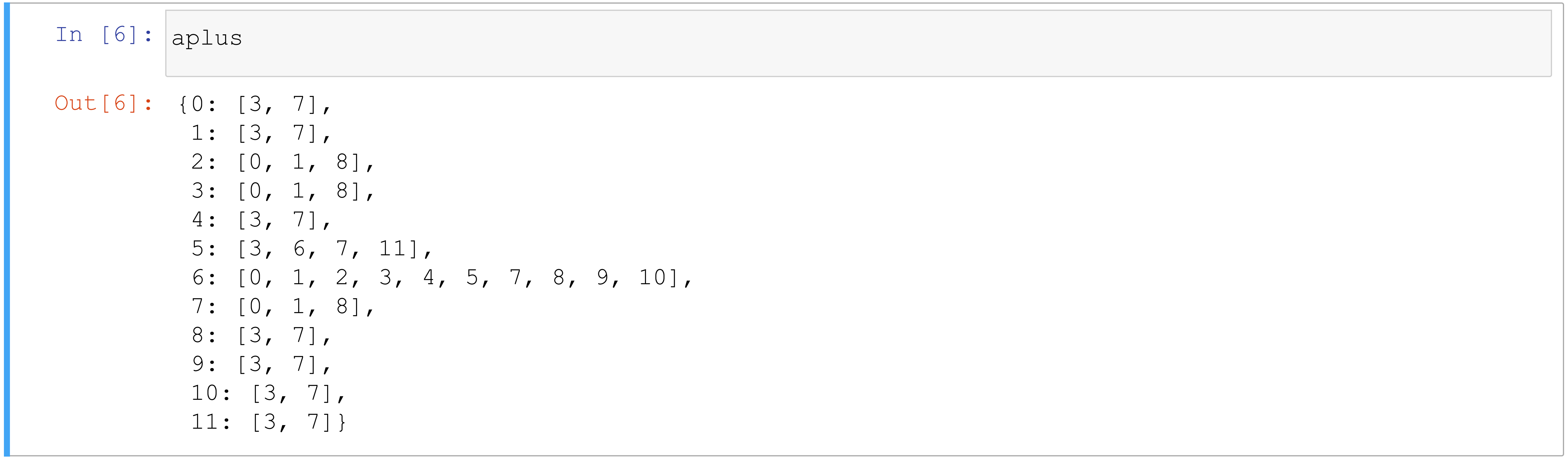
```
In [4]: pairs

Out[4]: {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+')}
```

- The blood type prevalence reflects the Italian distribution
- Incompatibility is mainly due to blood type
- ...Plus a number of more complex, but less impacting, factors

Generating the Benchmark

Arcs are first determined based on blood type compatibility

- Then a small (random) fraction of them (5%) is removed
- ...So as to simulate the other compatibility factors



```
In [6]: aplus
```

```
Out[6]: {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]}
```

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 [7]: cycles = er.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

```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
    ...
```

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

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

```python
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:

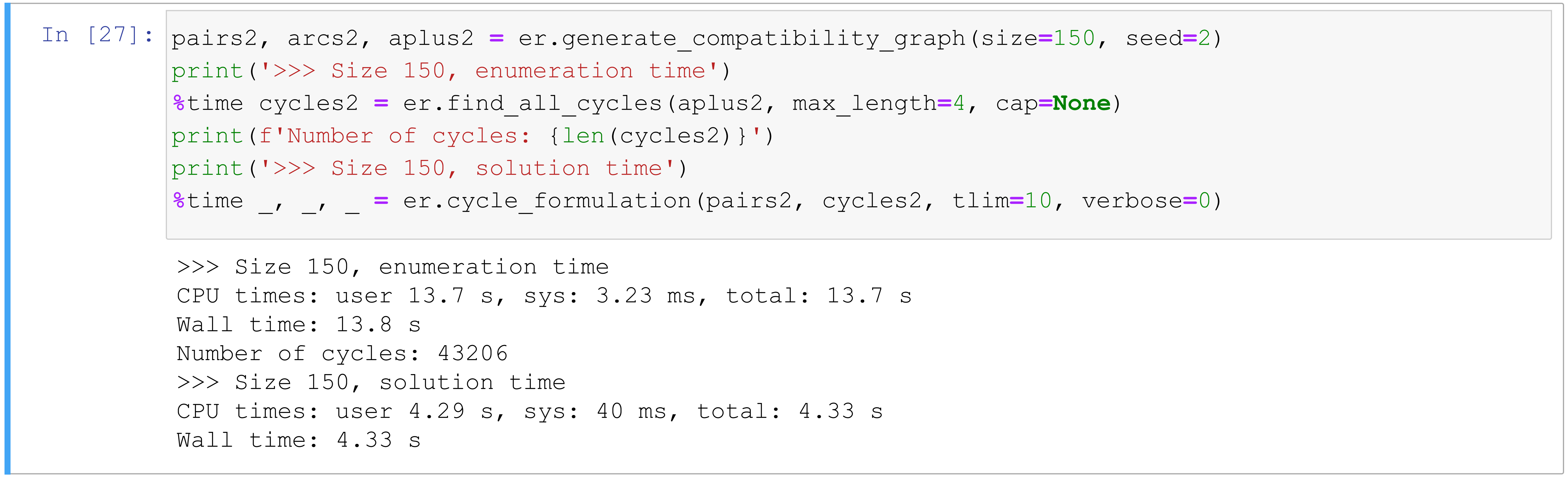
```
In [6]: pairs, arcs, aplus = er.generate_compatibility_graph(size=12, seed=2)
        cycles = er.find_all_cycles(aplus, max_length=4, cap=None)
        sol, tme, _ = er.cycle_formulation(pairs, cycles, tlim=10, verbose=1)
        print({k for k, v in sol.items() if v != 0 and k != 'objective'})

        Solution time: 0.003, objective value: 6.0 (optimal)
        {'x_6', 'x_1', 'x_2'}
```

Scalability Issue

The main drawback of the cycle formulation is the limited scalability

- The number of cycles grows exponentially with the graph size
  - Actually high-degree polynomial: the exponential factor is the max length
- The enumeration becomes more expensive and the model becomes larger

Both can quickly become major bottlenecks



In [27]:

```
pairs2, arcs2, aplus2 = er.generate_compatibility_graph(size=150, seed=2)
print('>>> Size 150, enumeration time')
%time cycles2 = er.find_all_cycles(aplus2, max_length=4, cap=None)
print(f'Number of cycles: {len(cycles2)}')
print('>>> Size 150, solution time')
%time _, _, _ = er.cycle_formulation(pairs2, cycles2, tlim=10, verbose=0)
```

```
>>> Size 150, enumeration time
CPU times: user 13.7 s, sys: 3.23 ms, total: 13.7 s
Wall time: 13.8 s
Number of cycles: 43206
>>> Size 150, solution time
CPU times: user 4.29 s, sys: 40 ms, total: 4.33 s
Wall time: 4.33 s
```

Column Generation

Column Generation and Dual Multipliers

Column Generation is a technique for solving problems with many variables

- Main idea: dynamically generate only the variables that are needed
- I.e. that have a chance of being part of an optimal solution

The technique is mainly designed for Linear Programs

How is this achieved?

Say we have a Linear Program in the form:

$$\min\{cx \mid Ax \geq b, x \geq 0\}$$

- This can be solved in polynomial time with Interior Point algorithm
- ...Or in pseudo-polynomial time with the Simplex algorithm
- Both provide optimal values x^* for the primal variables
- ...But also a vector of optimal dual multipliers λ (or dual variables)

Column Generation and Dual Multipliers

The dual multipliers are associated to the problem constraints

They stem from a formulation where constraints become cost terms

- They can be interpreted as penalties/rewards
  - If a constraint $\boldsymbol{i}$ is violated, we receive a penalty given by $\lambda_i \times$ the violation
  - If it is satisfied with a slack, we receive a reward given by $\lambda_i \times$ the slack
- The alternative formulation is called a Lagrangian relaxation

For any optimal LP solution x^* we have that:

- If a constraint $\boldsymbol{i}$ is satisfied with a slack, then $\lambda_i = 0$
  - In the alternative formulation, we receive no reward for the slack
  - ...Since we need to incentive for satisfying the constraint
- If a constraint $\boldsymbol{i}$ is tight, then $\lambda_i \geq 0$:
  - Any violation would incur a penalty proportional to λ_i
  - ...So there is an incentive for not violating the constraints

Cycle Formulation LP

Let's see this in action on our problem

- First, we need to consider the LP relaxation of our integer program
- ...And we need to rewrite it in standard form:

$$\min z = \sum_{i=j}^{n} -w_j x_j$$

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

$$x_j \geq 0 \qquad \forall j = 1..n$$

Besides removing the integrality constraints, we:

- ...Switched the optimization direction (from max to min)
- ...Switched the direction of the constraints

Cycle Formulation LP

We can now modify our model-building function

The function should build a Linear Program in the expected format:

```
def cycle_formulation(pairs, cycles, tlim=None, relaxation=False, verbose=1):
    if relaxation: # Build the solver
        slv = pywraplp.Solver.CreateSolver('CLP')
    else:
        slv = pywraplp.Solver.CreateSolver('CBC')
    ...
```

- We have added an optional `relaxation` parameter
- We use the CLP solver (the fastest with a fully permissive license)

Cycle Formulation LP

We need to build continuous, rather than integer, variables

```
def cycle_formulation(pairs, cycles, tlim=None, relaxation=False, verbose=1):
    ...
    if relaxation: # variables
        x = [slv.NumVar(0, infinity, f'x_{j}') for j in range(ncycles)]
    else:
        x = [slv.IntVar(0, 1, f'x_{j}') for j in range(ncycles)]
```

We also post the constraints in the format used in our theoretical arguments:

```
def cycle_formulation(pairs, cycles, tlim=None, relaxation=False, verbose=1):
    ...
    for i in range(npairs): # constraints
        if relaxation:
            slv.Add(-sum(x[j] for j in cpp[i]) >= -1)
        else:
            slv.Add(sum(x[j] for j in cpp[i]) <= 1)
```

Cycle Formulation LP

The same goes for the problem objective

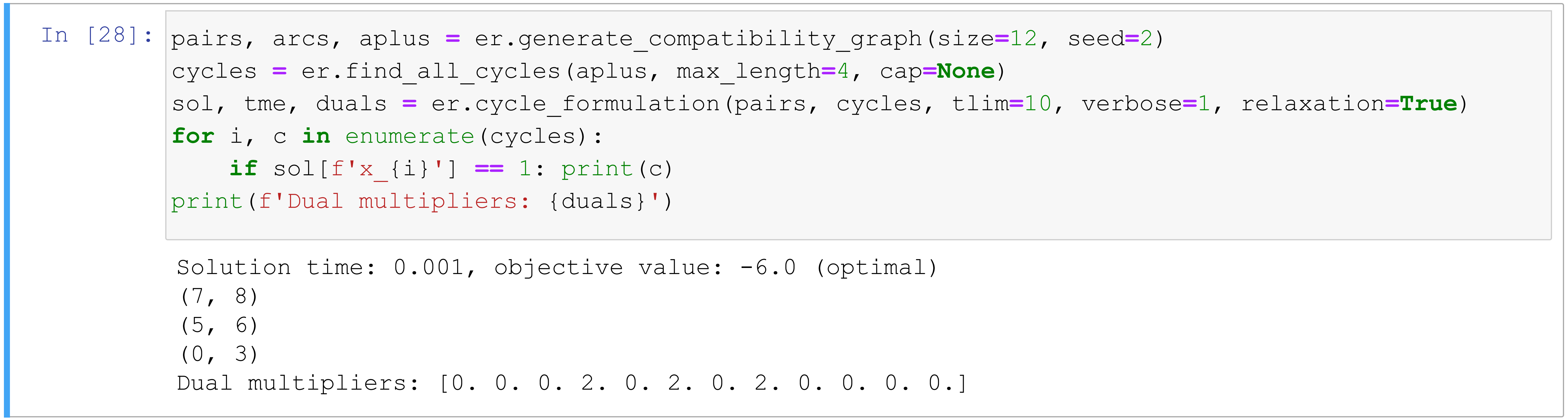
```
def cycle_formulation(pairs, cycles, tlim=None, relaxation=False, verbose=1):
    obj = sum(len(c) * x[j] for j, c in enumerate(cycles))
    if relaxation: # objective
        slv.Minimize(-obj)
    else:
        slv.Maximize(obj)
```

Once we have a solution, we can obtain the dual multipliers from the solver

```
def cycle_formulation(pairs, cycles, tlim=None, relaxation=False, verbose=1):
    ...
    duals = None
    if status in (slv.OPTIMAL, slv.FEASIBLE):
        ...
        if relaxation:
            duals = np.array([c.dual_value() for c in slv.constraints()])
    ...
```

Cycle Formulation LP

Now, let's try to solve the LP



In [28]:

```python
pairs, arcs, aplus = er.generate_compatibility_graph(size=12, seed=2)
cycles = er.find_all_cycles(aplus, max_length=4, cap=None)
sol, tme, duals = er.cycle_formulation(pairs, cycles, tlim=10, verbose=1, relaxation=True)
for i, c in enumerate(cycles):
    if sol[f'x_{i}'] == 1: print(c)
print(f'Dual multipliers: {duals}')
```

```
Solution time: 0.001, objective value: -6.0 (optimal)
(7, 8)
(5, 6)
(0, 3)
Dual multipliers: [0. 0. 0. 2. 0. 2. 0. 2. 0. 0. 0. 0.]
```

- We have one multiplier per constraint, i.e. one per graph node in our case
- The non-zero λ are associated to nodes used by the selected cycles
  - ...Meaning that their associated constraints are tight
- The cost is negative, since we have negated the original objective formula

Reduced Costs

Now, let x be a feasible LP solution, and λ its dual multiplier vector

The multipliers can be used to compute the "gradient of the constrained problem"

- Technically, we will use the gradient of the Lagrangian relaxation
- Such gradient comes as a closed-form formula, given by:

$$\nabla_x \left(cx + \lambda(b - Ax)\right)$$

- The expression $b - Ax$ represent the violation/slack for constraints $Ax \geq b$

The gradient can be rewritten as:

$$c - \lambda A$$

- Each component this gradient is known as reduced cost
- If x^* is optimal, the gradient (i.e. the reduced costs) will be null

Reduced Costs

Now, for a suboptimal solution x, the gradient will be non null

I.e. there will be negative components (reduced costs) in:

$$c - \lambda A$$

Actually, this is just a necessary condition:

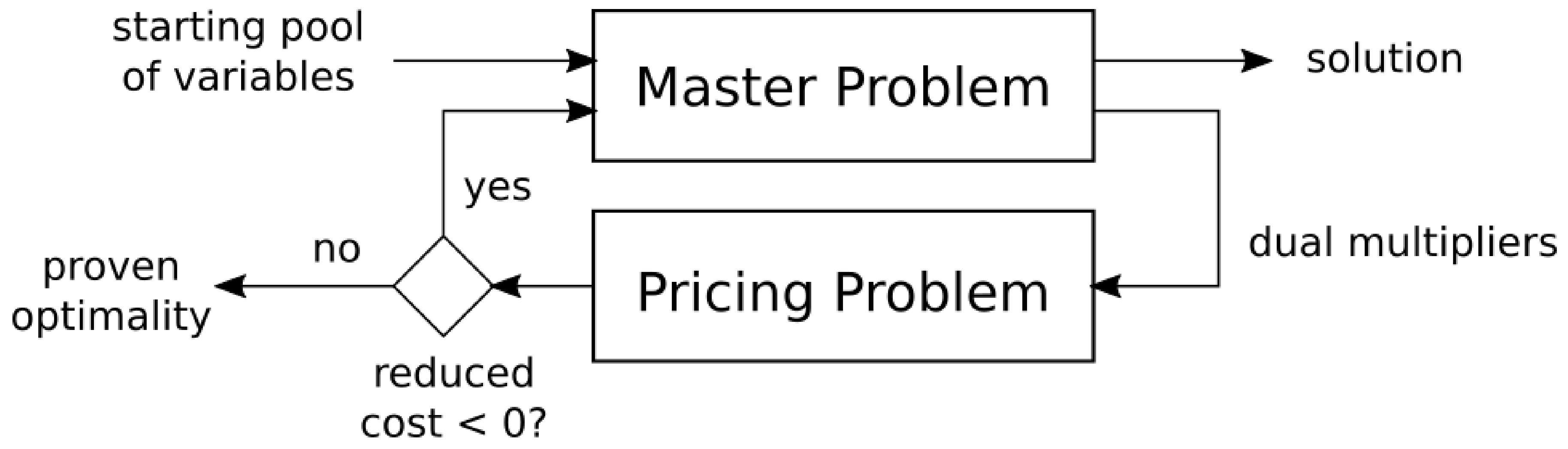
- Moving in the gradient direction may improve the cost or keep it unaltered
  - ...It will never make it worse
- Overall, there is a chance for improving x only if some reduced cost is negative

The reduced costs come in closed-form

- Given λ, for computing the reduced costs we just need the the coefficients in A
- This mean we can compute the reduced costs
- ...Also for variables that are not yet in the problem

Reduced Costs and Column Generation

This observation suggest as criterion for dynamically adding variables



- We start by solving an LP with a subset of the variables
  - Its solution will be feasible, but not necessarily optimal
- We consider the remaining variables (with their coefficients in $\boldsymbol{A}$)
  - ...And we compute their reduced costs (pricing problem)
- If there are no variables with negative r.c., we know our LP solution is optimal
- Otherwise, we repeat add such variables and repeat the process

Pricing Problem

In our case the pricing problem should consider (in principle) all cycles

This is an issue, since enumeration can be very expensive

- But we do not need to enumerate!
- ...We can focus on the most negative red. costs

This is enough to check whether there is any negative reduced cost

Let's have a look at our specific reduced costs structure:

$$c - \lambda^* A = \begin{pmatrix} \vdots \\ -w_j - \sum_{i=1}^{m} -a_{ij}\lambda_i \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ \sum_{i=1}^{m} a_{ij}(-1 + \lambda_i) \\ \vdots \end{pmatrix}$$

- The equivalence holds since $w_j = \sum_{i=1}^{m} a_{ij}$ (num. nodes in the cycle)

Pricing Problem

Let's try to understand this a bit better

$$\begin{pmatrix} \vdots \\ \sum_{i=1}^{m} a_{ij}(-1 + \lambda_i^*) \\ \vdots \end{pmatrix}$$

- If we include node i in a cycle j, then $a_{ij} = 1$
- Therefore, we incur a cost of $-1 + \lambda_i^*$

So, searching for the most negative reduced costs...

- ...Is equivalent to searching for minimum weight cycles
- ...With weights given by: $-1 + \lambda_i^*$

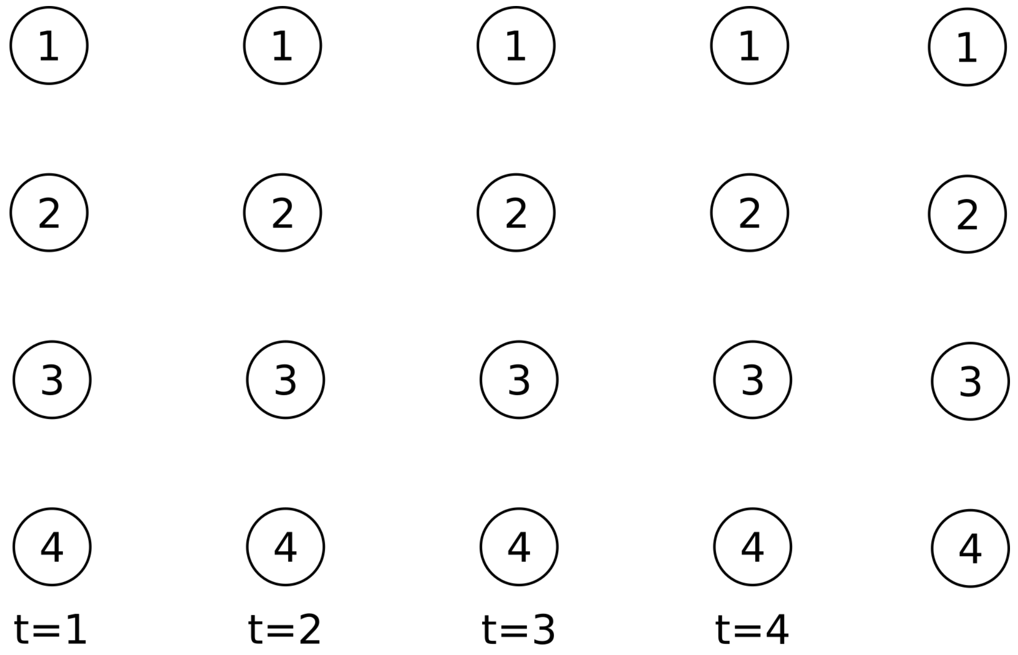
We also have a constraint on the maximum number of nodes per cycles

Constrained Minimum Cycle Weight

We will base our pricing algorithm on a Time Unfolded version of our graph

- An unfolded graph contains one copy of each original node per time unit
- In our case, time units correspond to possible cycle lengths

For our example graph, we get:

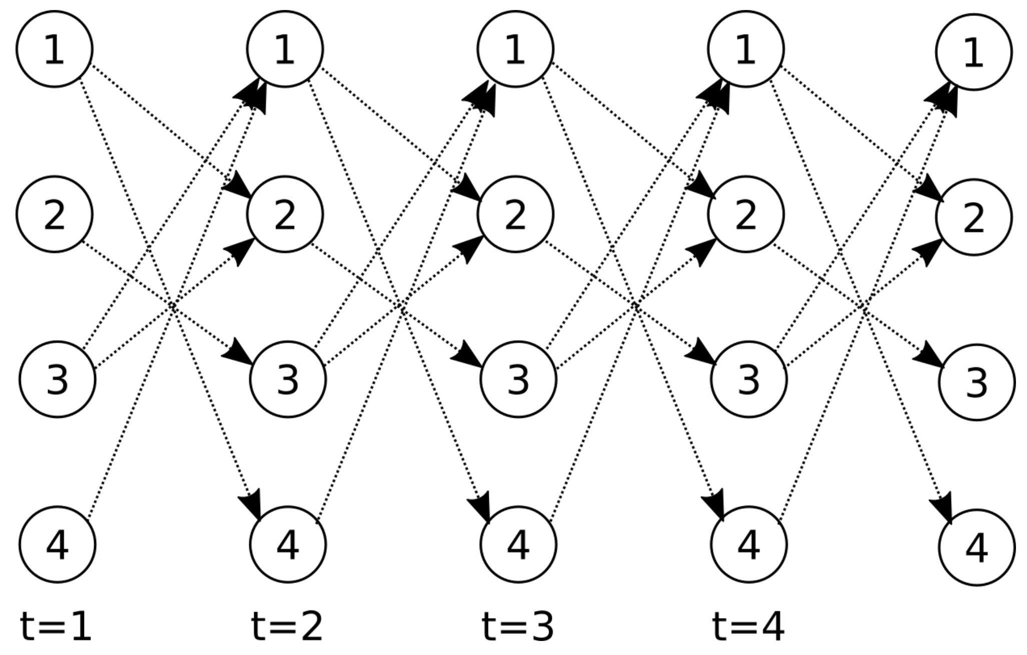


Since the maximum length is 4, there is no need to unfold beyond that

Constrained Minimum Cycle Weight

The time-unfolded graph is layered

- There are no arcs between nodes associated to the same time unit
- Arcs connect node associated to contiguous time units

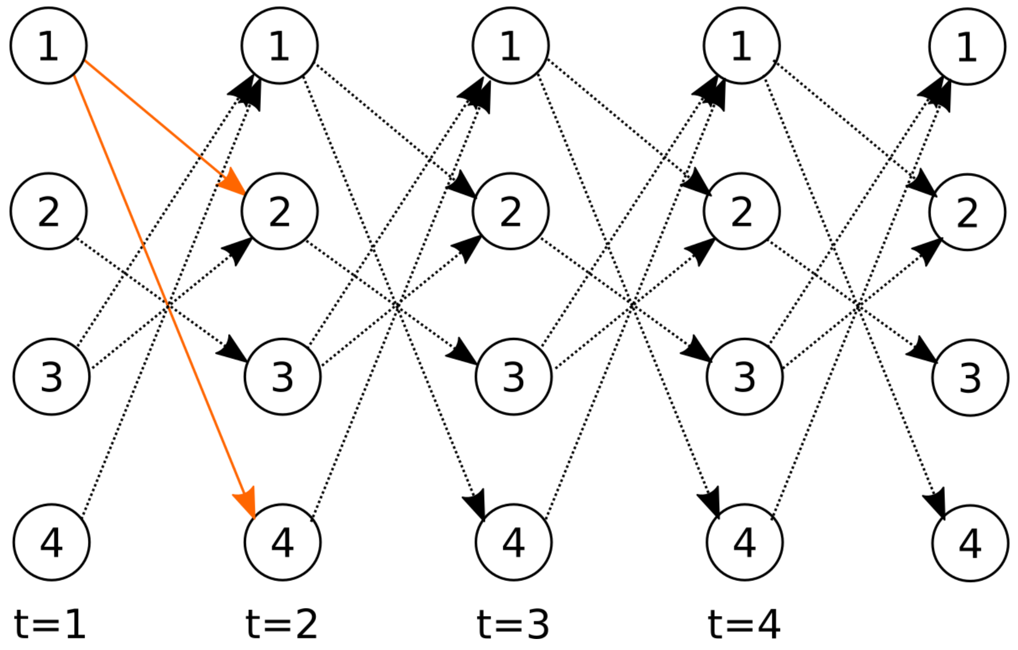


Since the graph is acyclic, shortest paths can be found using Dijkstra's algorithm

Constrained Minimum Cycle Weight

We can process one layer at a time

- We start at layer 1, from a given root node (1, in the figure)
- We consider all outgoing arcs

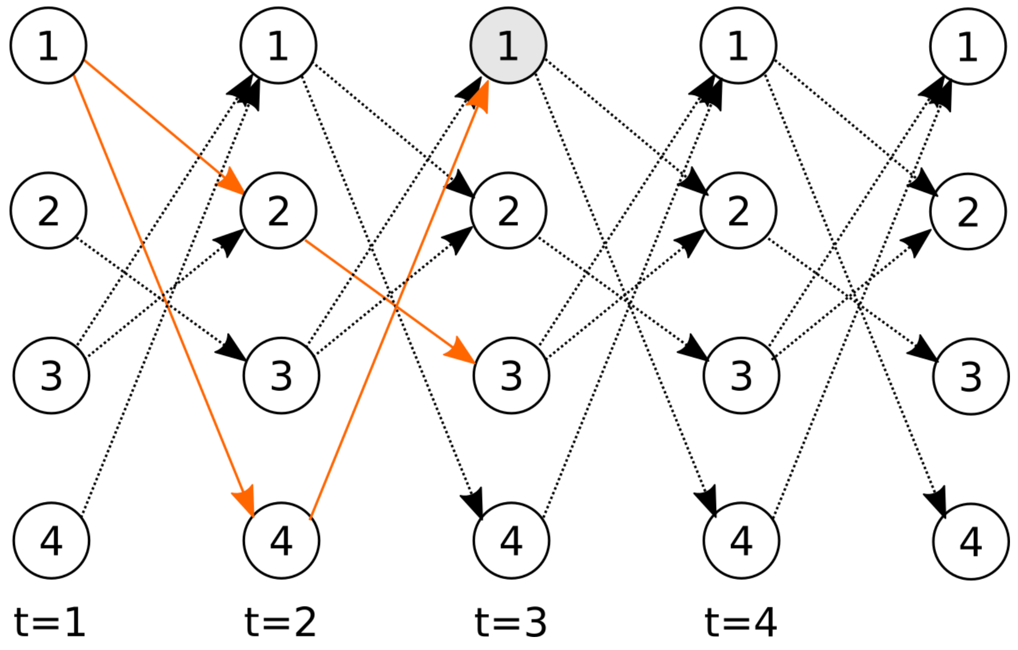


We update the shortest path to the destination nodes as usual in Dijkstra's

Constrained Minimum Cycle Weight

We then start from all visited nodes, and proceed as before

- If we end up visiting the root node again, we have found a cycle
- This is a shortest cycle including the root node, for the current length

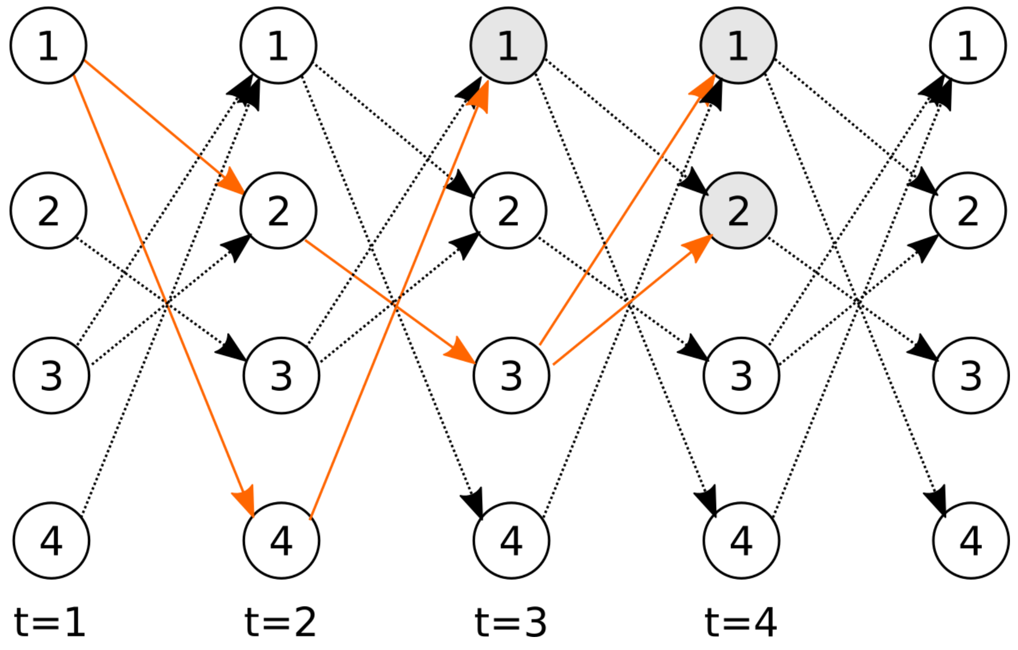


We store all these cycles (in this case, we store the cycle 1-4 for the path 1-4-1)

Constrained Minimum Cycle Weight

Nodes that close a cycle count as non-visited

- If we end up visiting a non-root node that is on the shortest path
- ...Then we have found a path with a sub-cycles

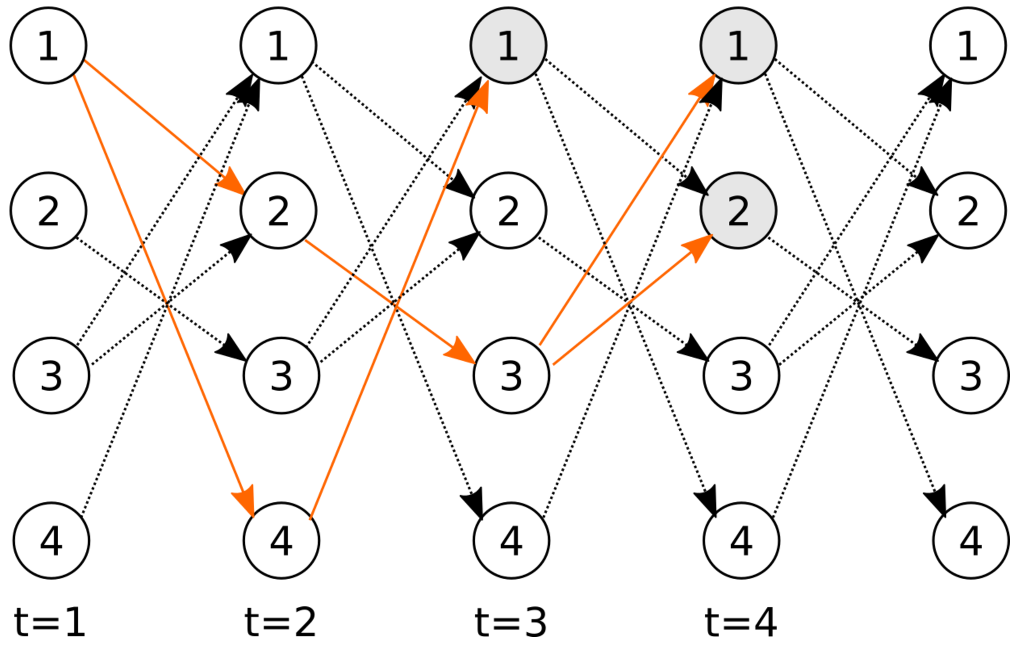


We do not store such paths (e.g., we store a cycle for 1-2-3-1, but not 1-2-3-2)

Constrained Minimum Cycle Weight

We proceed until that maximum length is reached

- ...Or until a given layer contains no visited nodes
- Then we can restart from another root node



In the next restart, we can ignore all arcs pointing to already considered roots

- Since all shortest cycles containing those nodes have already been found

Constrained Minimum Cycle Weight - Implementation

The bulk of the computation is done by this function

```python
def shortest_cycles_from_root(root, aplus, weights, max_len):
    spt = {root: {root}} # initial shortest paths
    dst = {root: weights[root]} # shortest path distances
    cycles, ccosts = [], []
    for k in range(max_len): # loop over the possible cycle lengths
        ndst, nspt = {}, {}
        for i in dst: # process all visited nodes
            for j in aplus[i]: # loop over outgoing arcs
                if j == root: # detect cycles
                    cycles.append(spt[i])
                    ccosts.append(dst[i])
                elif j in spt[i]: # skip subcycles
                    continue
                elif j not in ndst or dst[i] + weights[j] < ndst[j]: # Dijkstra update
                    ndst[j] = dst[i] + weights[j]
                    nspt[j] = spt[i] | {j}
```

Constrained Minimum Cycle Weight - Implementation

Initially, the only shortest path include the root alone

```python
def shortest_cycles_from_root(root, aplus, weights, max_len):
    spt = {root: {root}} # initial shortest paths
    dst = {root: weights[root]} # shortest path distances
```

We process one layer at a time:

```python
def shortest_cycles_from_root(root, aplus, weights, max_len):
    ...
    cycles, ccosts = [], []
    for k in range(max_len): # loop over the possible cycle lengths
        ...
    return cycles, ccosts
```

- At the end of the computation we return the shortest paths
- ...And their costs/weights

Constrained Minimum Cycle Weight - Implementation

At each layer, we build the shortest paths to the next layer

```python
def shortest_cycles_from_root(root, aplus, weights, max_len):
    ...
    for k in range(max_len): # loop over the possible cycle lengths
        ndst, nspt = {}, {}
        for i in dst: # process all visited nodes
            for j in aplus[i]: # loop over outgoing arcs
                ...
        dst, spt = ndst, nspt
    ...
```

- We process all "visited" nodes, which are in `dst`
- I.e. those having a valid shortest path for the current length
- Then we loop over all their outgoing arcs

Constrained Minimum Cycle Weight - Implementation

How we deal with each outgoing arc depends on the current situation

```python
def shortest_cycles_from_root(root, aplus, weights, max_len):
            ...
            for j in aplus[i]: # loop over outgoing arcs
                if j == root: # detect cycles
                    cycles.append(spt[i])
                    ccosts.append(dst[i])
                elif j in spt[i]: # skip subcycles
                    continue
                elif j not in ndst or dst[i] + weights[j] < ndst[j]: # Dijkstra update
                    ndst[j] = dst[i] + weights[j]
                    nspt[j] = spt[i] | {j}
            ...
```

- If we detect a cycle to the root, we store it
- If we detect a sub-cycle, we disregard it
- Otherwise, we keep building the shortest path using Dijkstra's method

Constrained Minimum Cycle Weight - Implementation

The previous function is called for each possible root

```python
def shortest_cycles(aplus, weights, max_len):
    aminus = {i:[] for i in aplus} # graph in backward star format
    for i, alist in aplus.items():
        for j in alist: aminus[j].append(i)
    aplus = copy.deepcopy(aplus) # copy of the forward star
    cycles, ccosts = [], []
    for root in range(len(weights)): # loop over possible roots
        # Collect shortest paths from this root
        tcl, tct = shortest_cycles_from_root(root, aplus, weights, max_len)
        cycles += tcl
        ccosts += tct
        for i in aminus[root]: # remove all forward arcs twd the processed root
            aplus[i].remove(root)
    return cycles, ccosts
```

Constrained Minimum Cycle Weight

The process returns (at most) one cycle per root node and per valid weight

For our example graph, we have:

```
In [29]: weights = -np.ones(len(pairs)) + duals
         scl, sct = er.shortest_cycles(aplus, weights, max_len=4)
         print(scl)
         print(sct)

         [{0, 3}, {0, 7}, {0, 1, 3, 7}, {1, 3}, {1, 7}, {8, 1, 3, 7}, {8, 3}, {5, 6}, {8, 7}]
         [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

- All shortest cycles have non-negative reduced costs
- This is expected since the dual multiplier refer to an optimal solution

Focusing on minimum weight cycles gives a massive speed improvement:

```
In [30]: pairs2, arcs2, aplus2 = er.generate_compatibility_graph(size=150, seed=2)
         %time cycles2, _ = er.shortest_cycles(aplus2, weights=-np.ones(len(pairs2)), max_len=4)

         CPU times: user 124 ms, sys: 3.28 ms, total: 128 ms
         Wall time: 128 ms
```

Column Generation

We can now inspect the column generation method itself

```
def cycle_formulation_cg(pairs, aplus, max_len, itcap=10, tol=1e-3, verbose=1):
    weights = -np.ones(len(pairs)) # initial cycle pool
    cycles, _ = er.shortest_cycles(aplus, weights, max_len=max_len)
    converged = False # main loop
    for itn in range(itcap):
        sol, stime, duals = er.cycle_formulation(pairs, cycles, verbose=0, relaxation=Tru
        if verbose > 0: ...
        weights = -np.ones(len(pairs)) + duals # shortest paths
        scl, sct = er.shortest_cycles(aplus, weights, max_len=max_len)
        nrc_cycles = [scl[i] for i, c in enumerate(sct) if c < -tol] # negative r.c.
        if verbose > 0: ...
        if len(nrc_cycles) == 0: # no improvement possible
            converged = True
            break
        else: cycles += nrc_cycles # add new arcs
    return cycles, converged
```

Column Generation

The initial pool of variables corresponds to all shortest cycles

```
def cycle_formulation_cg(pairs, aplus, max_len, itcap=10, tol=1e-3, verbose=1):
    weights = -np.ones(len(pairs)) # initial cycle pool
    cycles, _ = er.shortest_cycles(aplus, weights, max_len=max_len)
```

- The cycle weight is just the number of nodes

Then we start the main loop

```
def cycle_formulation_cg(pairs, aplus, max_len, itcap=10, tol=1e-3, verbose=1):
    ...
    converged = False # main loop
    for itn in range(itcap):
        ...
    return cycles, converged
```

- At the end we return the optimized cycle pool, plus convergence flag

Column Generation

At each iteration, we solve the LP relaxation

```
def cycle_formulation_cg(pairs, aplus, max_len, itcap=10, tol=1e-3, verbose=1):
    ...
    for itn in range(itcap):
        sol, stime, duals = er.cycle_formulation(pairs, cycles, verbose=0, relaxation=Tru
        if verbose > 0: ...
```

Then we find all shortest cycles

```
def cycle_formulation_cg(pairs, aplus, max_len, itcap=10, tol=1e-3, verbose=1):
    ...
    for itn in range(itcap):
        ...
        weights = -np.ones(len(pairs)) + duals # shortest paths
        scl, sct = er.shortest_cycles(aplus, weights, max_len=max_len)
        ...
```

Column Generation

Then we detect the cycles with negative reduced costs

```
def cycle_formulation_cg(pairs, aplus, max_len, itcap=10, tol=1e-3, verbose=1):
    ...
    for itn in range(itcap):
        ...
        nrc_cycles = [scl[i] for i, c in enumerate(sct) if c < -tol] # negative r.c.
        if verbose > 0: ...
        if len(nrc_cycles) == 0: # no improvement possible
            converged = True
            break
        else: cycles += nrc_cycles # add new arcs
```

- LP solvers operate withing tolerances, so it's a good idea to use one
- If there are not cycles with negative r.c. we have converged
- Otherwise, we add all arcs with negative r.c. to the problem

Adding multiple arcs is usually a good idea in Column Generation

Column Generation - Correctness

It's time to test the approach. We will initially focus on correctness

We generate a graph:

```
In [31]: pairs, arcs, aplus = er.generate_compatibility_graph(size=100, seed=2)
```

Then we solve the GC formulation:

```
In [32]: cycles_cg, _  = er.cycle_formulation_cg(pairs, aplus, max_len=4, itcap=10)
```

```
(CG, it. 0), #cycles: 839, time: 0.057, relaxation objective: -36.00
(CG, it. 0), #cycles with negative reduced cost: 0
```

And we compare it with the approach based on full enumeration:

```
In [33]: cycles_cf = er.find_all_cycles(aplus, max_length=4, cap=None)
         sol, stime, duals = er.cycle_formulation(pairs, cycles_cf, tlim=10, verbose=0, relaxation=True)
         print(f'(Full formulation) #cycles: {len(cycles_cf)}, time: {stime}, relaxation objective: {sol
```

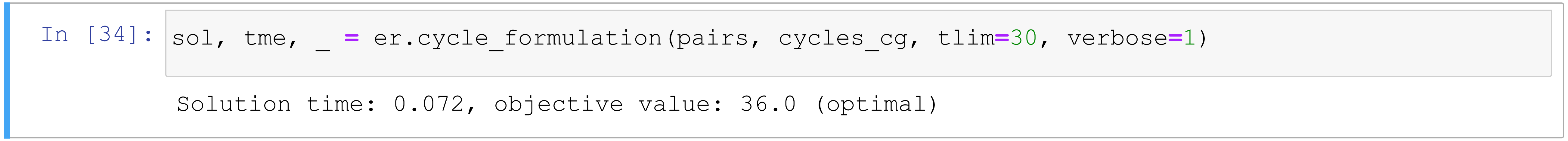
```
(Full formulation) #cycles: 9890, time: 1.582, relaxation objective: -36.00
```

Column Generation - Downstream IP

After we solve the CG formulation, we still don't have an actual solution

- We have an optimal solution of the LP relaxation
- ...Which may violate the integrality constraints

A simple strategy: keep the set of variables and solve the original problem



```
In [34]: sol, tme, _ = er.cycle_formulation(pairs, cycles_cg, tlim=30, verbose=1)

         Solution time: 0.072, objective value: 36.0 (optimal)
```

This one is guarantee optimal only if the LP-IP gap is zero (as in our case)

- Otherwise, we should start branching (leading to Branch & Price approaches)
- In practice, if the master problem formulation has a good LP bound
- ...Then even this simple sequential approach will lead good results

Column Generation - Scalability

Now we will quickly test the method scalability

Let's try with 300 pairs:

```
In [38]: %%time
pairs2, arcs2, aplus2 = er.generate_compatibility_graph(size=300, seed=2)
cycles_cg2, _  = er.cycle_formulation_cg(pairs2, aplus2, max_len=4, itcap=10)
_, _, _ = er.cycle_formulation(pairs2, cycles_cg2, tlim=30, verbose=1)
```

```
(CG, it. 0), #cycles: 8906, time: 0.548, relaxation objective: -122.00
(CG, it. 0), #cycles with negative reduced cost: 0
Solution time: 0.952, objective value: 122.0 (optimal)
CPU times: user 3.3 s, sys: 6.71 ms, total: 3.31 s
Wall time: 3.31 s
```

Again with 600 pairs:

```
In [39]: %%time
pairs3, arcs3, aplus3 = er.generate_compatibility_graph(size=600, seed=2)
cycles_cg3, _  = er.cycle_formulation_cg(pairs3, aplus3, max_len=4, itcap=10)
_, _, _ = er.cycle_formulation(pairs3, cycles_cg3, tlim=30, verbose=1)
```

```
(CG, it. 0), #cycles: 31010, time: 2.445, relaxation objective: -229.00
(CG, it. 0), #cycles with negative reduced cost: 0
Solution time: 2.821, objective value: 229.0 (optimal)
```

Considerations

Column Generation can be a very useful technique

- It is little known outside the Combinatorial Optimization community
- ...But it can provide massive scalability improvements
- ...And it can lead to simpler problem formulations

The key is a clean master problem

- The method works well if the master problem has a high-quality LP relaxation
- Usually, this is the case if the master has a clean structure
  - E.g. multi-knapsack, set covering, assignment problems...
- The trick is packing the complexity in the definition of the problem variables

Combinatorial optimization can be used also in the pricing problem

- If pricing problem does not admit clean formulation...
- ...You can still try and solve it using a combinatorial method (e.g. CP, SMT, MIP...)