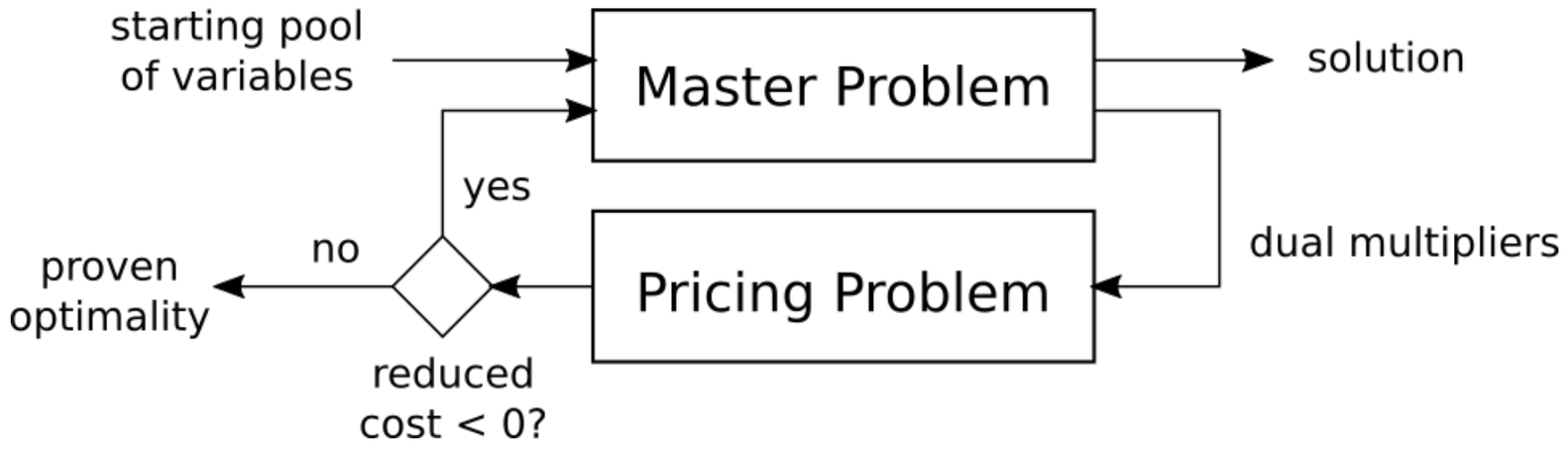
# Implementing Column Generation

# Implementing a CG Approach

## We can now start implementing our CG approach




- We need to define how to solve the restricted Master Problem
- ... And how to solve the Pricing Problem

**We'll discover that this case study is not a trivial one**

...But also that, with the right choices, it can be solved fast!

# Restricted Master Problem

**Dealing with the restricted Master Problem is the easy part**

...Since we can still rely on the cycle formulation:

$$\min -\sum_{j \in S} w_j x_j$$

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

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

Compared to the full formulation:

- We restrict the summations
- We switched the direction of optimization

# Restricted Master Problem

**In th code:**

- We add an option to relax all variables so that they are continuous
- In which case, we use the CLP solver instead of CBC
- We switch optimization and the constraint direction
- We add code to retrieve the dual multipliers

**The resulting API is:**

```
def cycle_formulation(pairs, cycles, tlim=None, relaxation=False, verbose=1):
    ...
```

- The summations are restricted by just passing a subset of cycles


# Restricted Master Problem

## Now, let's try to solve the LP

In [16]:
```
pairs, arcs, aplus = util.generate_compatibility_graph(size=12, seed=2)
cycles = util.find_all_cycles(aplus, max_length=4, cap=None)
sol, tme, duals = util.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: 0.000 sec, objective value: -6.0 (optimal)
(0, 7, 8, 3)
(5, 6)
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 $\lambda$ 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

# Pricing Problem

**Our pricing problem formulation is:**

$$\text{argmin} \sum_{i=1}^{m} y_i(-1 + \lambda_i^*)$$

$$\text{s.t. } y \text{ defines a cycle}$$

$$\sum_{i=1}^{m} y_i \leq C$$

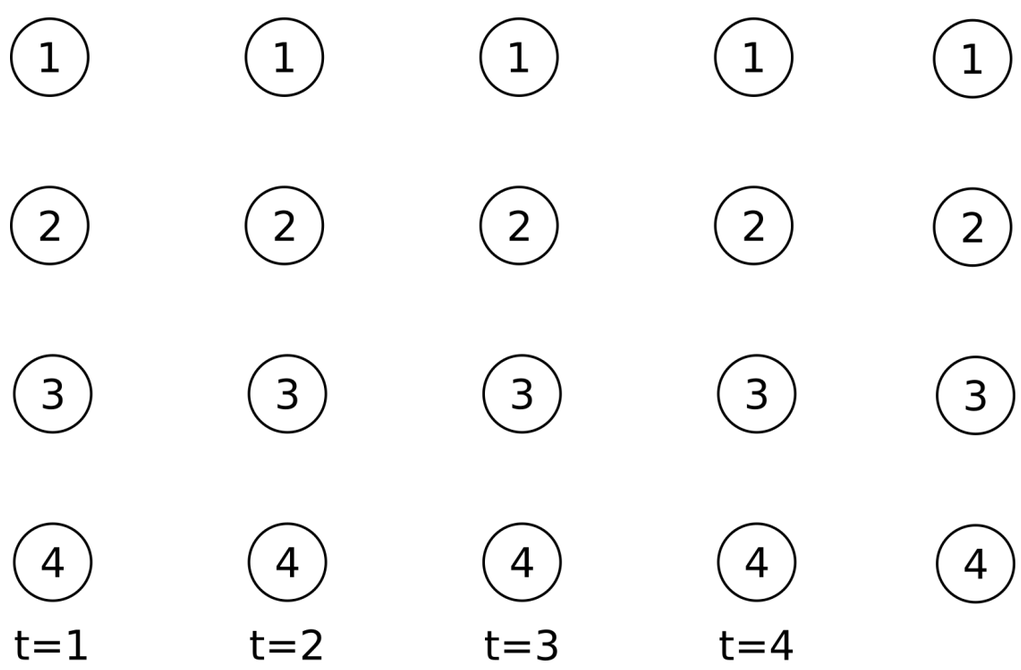
$$y_i \in \{0, 1\} \qquad \forall i = 1..m$$

- We need to choose nodes that form a minimum weight cycle
- ...Which makes this problem inconvenient to solve via LP or MILP

**So we'll use instead a different approach**

# Constrained Minimum Cycle Weight

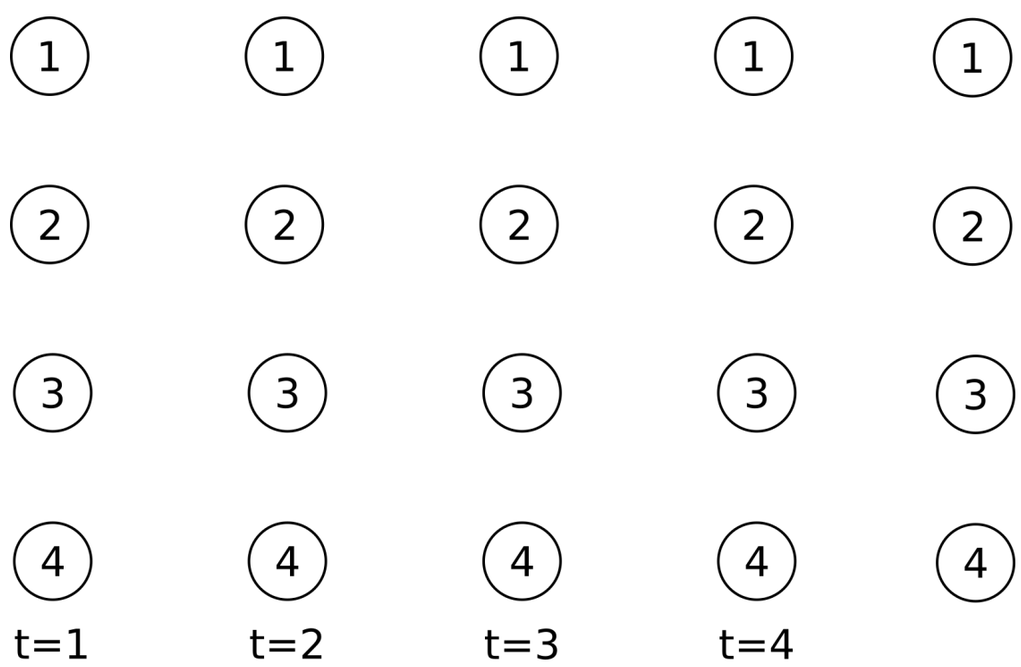
**We will base our pricing algorithm on a Time Unfolded version of our graph**




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

# Constrained Minimum Cycle Weight

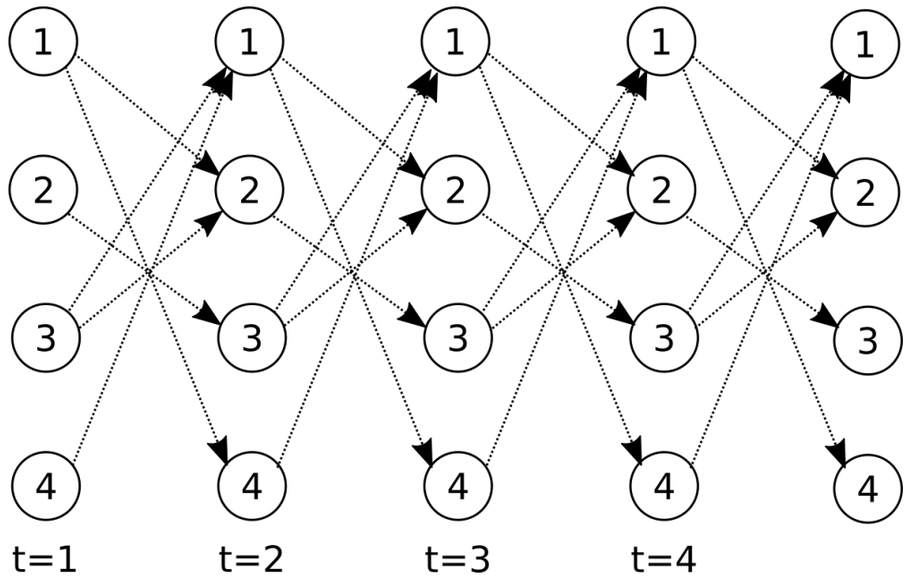
## We can use unfolding to account for one of our constraints




- Our cycles can contain at most $C$ nodes (say $C = 4$)
- For this reason, we will unfold $C + 1$ times

# 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

Now we need to select nodes that form a cycle on the original graph

# How do we do that over the TUG?

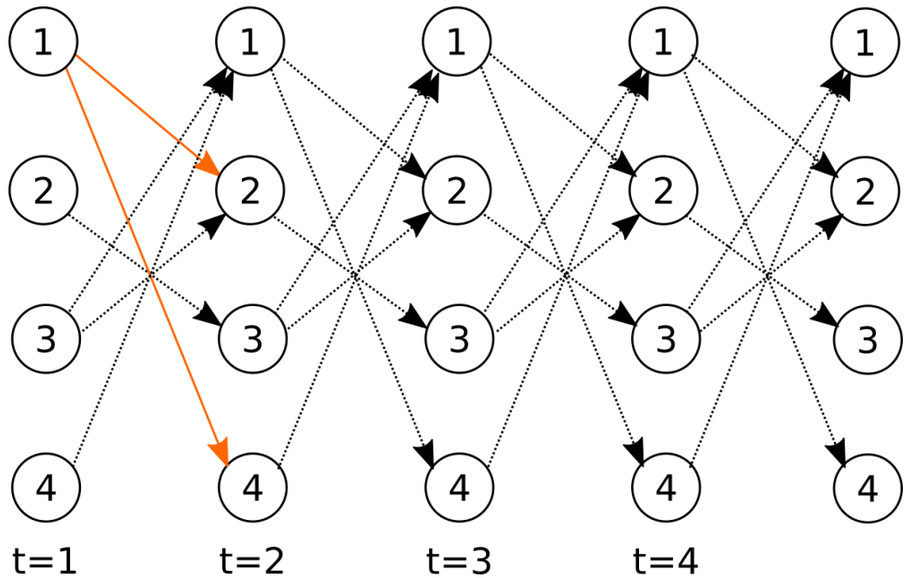
**Now we need to select nodes that form a cycle on the original graph**

# How do we do that over the TUG?

Since the graph is acyclic, we can use _Dijkstra's algorithm_

# Constrained Minimum Cycle Weight

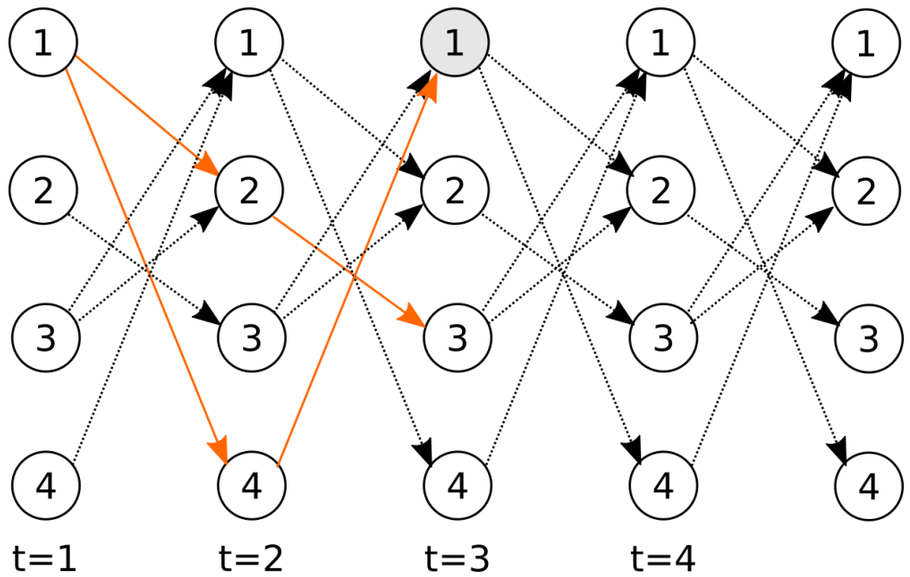
## We search for a shortest path, processing one layer at a time




- 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

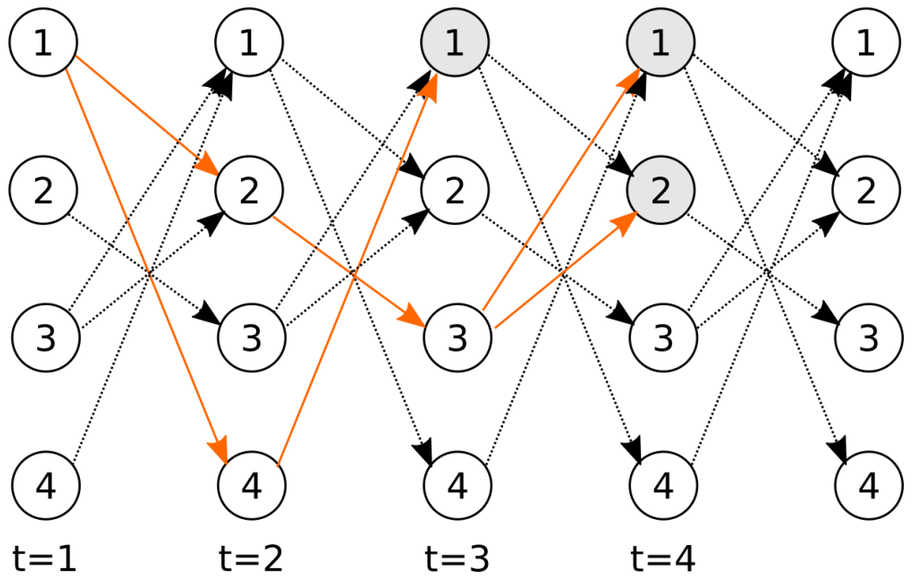
## We then start from all visited nodes, and proceed as before




- 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

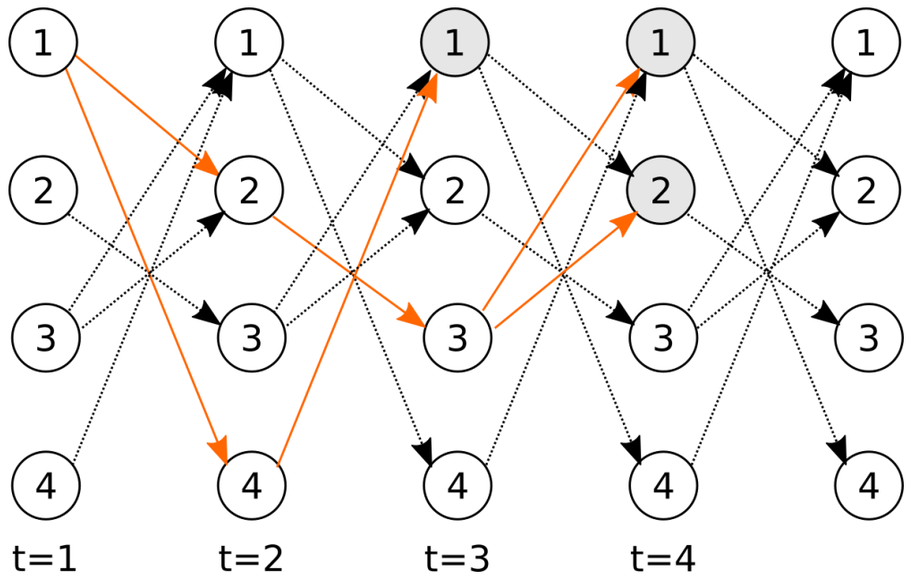
## Nodes that close a cycle count as non-visited




- 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 maximum length, or until no node is visited in the nest layer**




- 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

## The process returns (at most) one cycle per root node and per non-zero weight

For our example graph, we have:

In [17]:
```
weights = -np.ones(len(pairs)) + duals
scl, sct = util.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

# Constrained Minimum Cycle Weight

## Formulating pricing as optimization can be very beneficial

In our case, we get a massive speed improvement w.r.t. enumeration:

In [30]:
```
pairs, arcs, aplus = util.generate_compatibility_graph(size=150, seed=2)
%time cycles = util.find_all_cycles(aplus, max_length=4, cap=None)
```

```
CPU times: user 6.22 s, sys: 1.1 ms, total: 6.22 s
Wall time: 6.23 s
```

In [31]:
```
%time cycles2, _ = util.shortest_cycles(aplus, weights=-np.ones(len(pairs)), max_len=4)
```

```
CPU times: user 75.4 ms, sys: 0 ns, total: 75.4 ms
Wall time: 75.1 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
- We also add multiple columns at every iteration
- This is usually a good idea, since it reduces iteration overhead

# Column Generation - Correctness

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

We generate a graph:

```
In [33]: pairs, arcs, aplus = util.generate_compatibility_graph(size=100, seed=2)
```

Then we solve the GC formulation:

```
In [37]: cycles_cg, _  = util.cycle_formulation_cg(pairs, aplus, max_len=4, itcap=10)
```

```
(CG, it. 0), #cycles: 839, time: 0.02, 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 [38]: cycles_cf = util.find_all_cycles(aplus, max_length=4, cap=None)
sol, stime, duals = util.cycle_formulation(pairs, cycles_cf, tlim=10, verbose=0, relaxation=
print(f'(Full formulation) #cycles: {len(cycles_cf)}, time: {stime}, relaxation objective: {
```

```
(Full formulation) #cycles: 9890, time: 1.358, 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 [40]: sol, tme, _ = util.cycle_formulation(pairs, cycles_cg, tlim=30, verbose=1)

         Solution time: 0.033 sec, objective value: 36.0 (optimal)
```

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

- Otherwise, in principle we should start branching (Branch & Price)
- In practice, we are usually happy with this two-phase approach

Just make sure that your Master Problem yeilds a good bound!

# Column Generation - Scalability

## Now we will quickly test the method scalability

Let's try with 300 and 600 pairs:

In [41]:
```
%%time
pairs2, arcs2, aplus2 = util.generate_compatibility_graph(size=300, seed=2)
cycles_cg2, _  = util.cycle_formulation_cg(pairs2, aplus2, max_len=4, itcap=10)
_, _, _ = util.cycle_formulation(pairs2, cycles_cg2, tlim=30, verbose=1)
```

```
(CG, it. 0), #cycles: 8906, time: 0.283, relaxation objective: -122.00
(CG, it. 0), #cycles with negative reduced cost: 0
Solution time: 0.509 sec, objective value: 122.0 (optimal)
CPU times: user 1.83 s, sys: 19.9 ms, total: 1.85 s
Wall time: 1.85 s
```

In [44]:
```
%%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)
_, _, _ = util.cycle_formulation(pairs3, cycles_cg3, tlim=30, verbose=1)
```

```
(CG, it. 0), #cycles: 31010, time: 1.021, relaxation objective: -229.00
(CG, it. 0), #cycles with negative reduced cost: 0
Solution time: 1.456 sec, objective value: 229.0 (optimal)
CPU times: user 9.37 s, sys: 49.3 ms, total: 9.42 s
Wall time: 9.43 s
```

# Considerations

## Column generation is not an easy technique

...But when it works, it works very well

- The trick is finding a clean Master Problem formulation
- ...By including constraints as part of the variable definition
- In short: a clean problem with super-complicated variables

## There's no need to solve the pricing problem via dedicated methods

- We can use Linear Programming, MILP, or even SMT or CP
- The pricing problem can even be NP-hard
- ...We just needt it to be easy solve for the scale we need