Column Generation

Because we cannot spend all day pricing

Column Generation

We now have a mechanism to price all variables not in the pool

...But we still need to handle an exponential number of them

- I.e. enumerating paths would still be prohibitively expensive

What can we do about it?

Column Generation

We now have a mechanism to price all variables not in the pool

...But we still need to handle an exponential number of them

- I.e. enumerating paths would still be prohibitively expensive

What can we do about it?

- There is no need to find all paths with negative $\frac{\partial}{\partial x_j} f(x) < 0$
- We just need to determine whether one such path exists

Hence, we can build a variable with the most negative $\frac{\partial}{\partial x_j} f(x) < 0$

- Since variables correspond to columns in LP
- ...This approach is called Column Generation

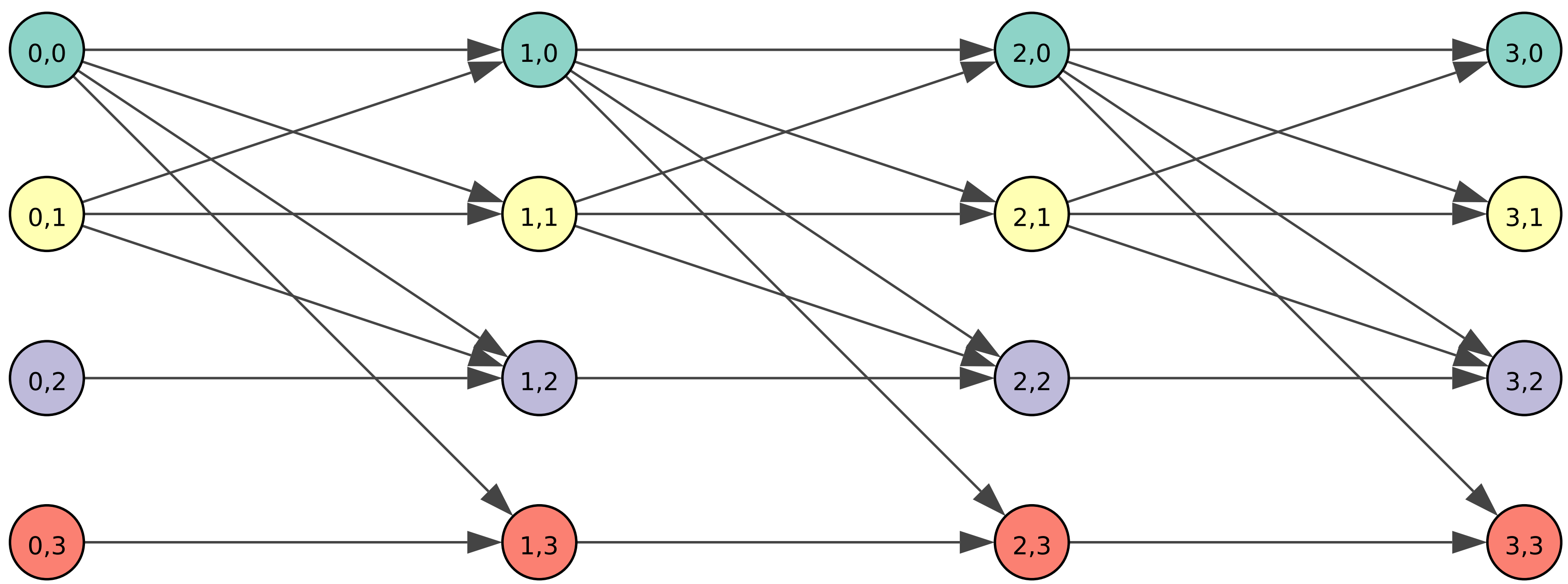
Pricing ad Optimization

In practice, we view pricing as an optimization problem

- In our case we are looking at paths
- ...So it make sense to visualize them on the Time Unfolded Graph

In [15]:
```
ig.plot(tug, **util.get_visual_style(tug), bbox=(700, 300), margin=50)
```

Out[15]:



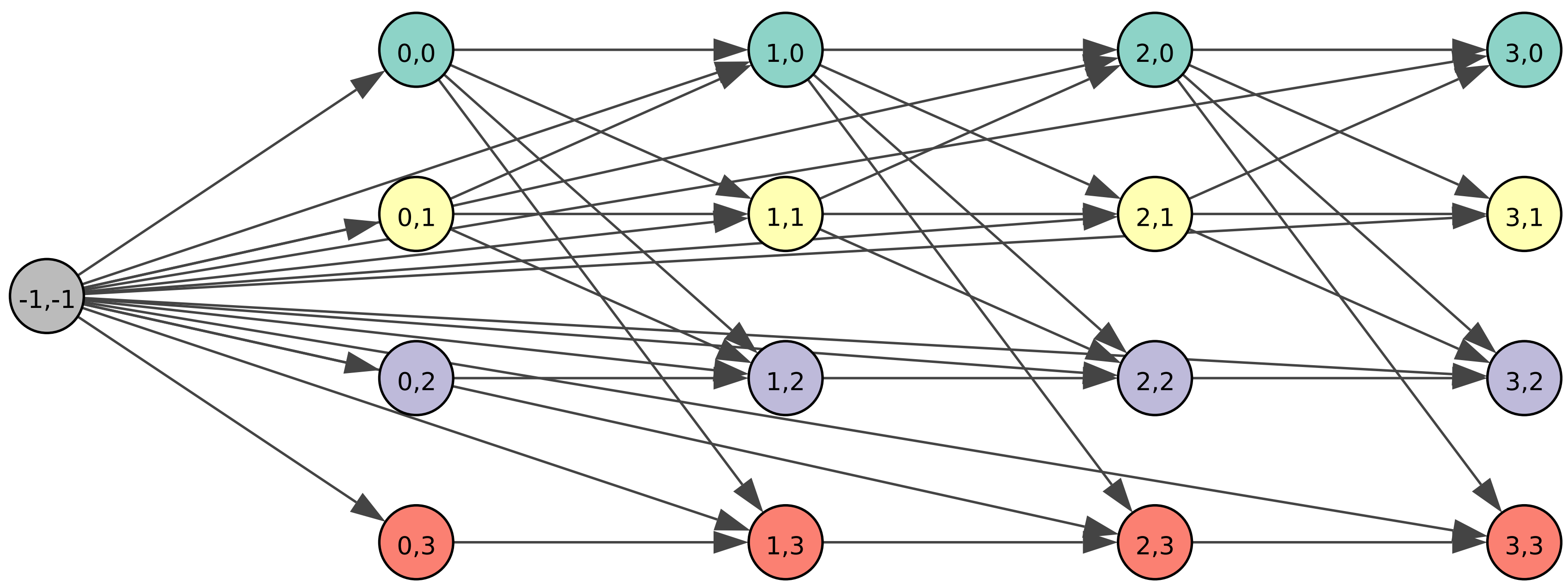
Pricing ad Optimization

When you want to consider all paths in a directed graph

- ...It is convenient to add a fake source node
- Then you can assume that all paths start from that node

In [16]:
```
ig.plot(tugs, **util.get_visual_style(tugs), bbox=(700, 300), margin=50)
```

Out[16]:



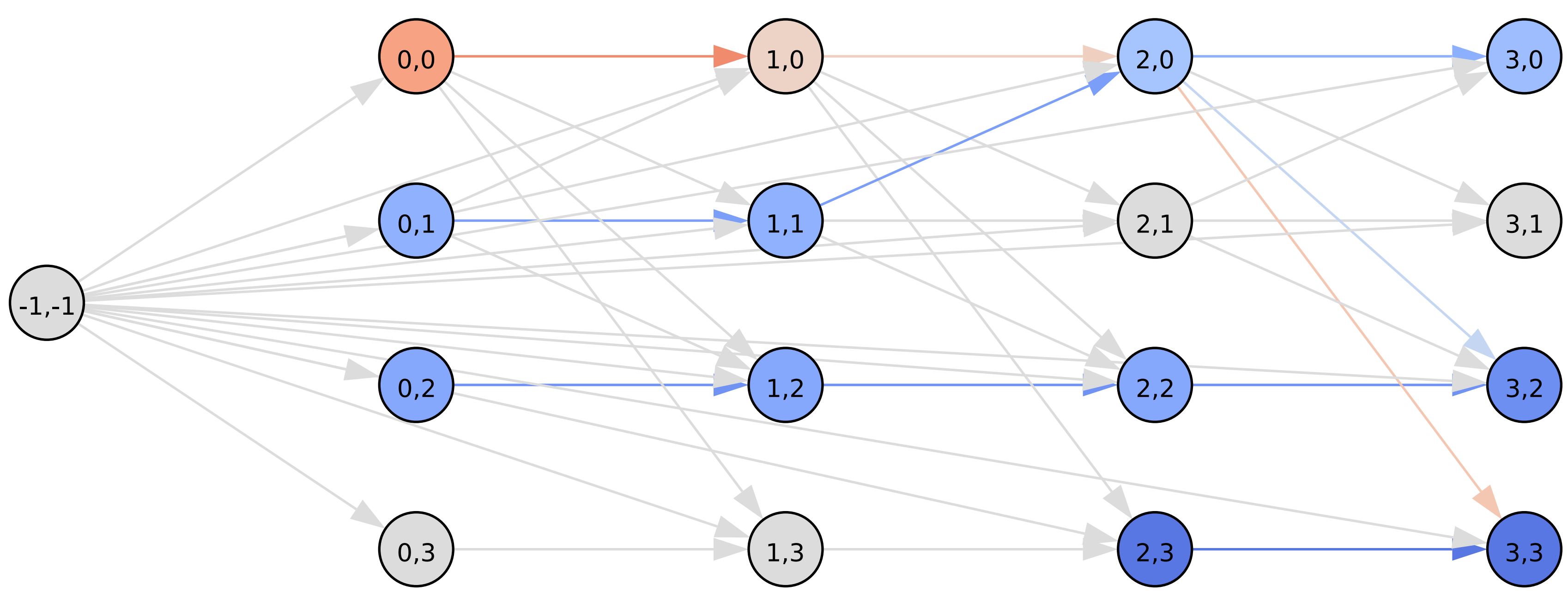
Pricing as Optimization

We can treat our residual as node and arc weights

- In the plot, a grey shade corresponds to near-zero weight
- A blue shade is used for negative weights, and a red share for positive ones

In [17]:
```
visual_style = util.get_visual_style(tugs, vertex_weights=nres0, edge_weights=ares0)
ig.plot(tugs, **visual_style, bbox=(700, 300), margin=50)
```

Out[17]:



Shortest Path Algorithms

Optimization problems over graphs

...Are often amenable to dedicated, very efficient, algorithms

What about our graph?

Shortest Path Algorithms

Optimization problems over graphs

...Are often amenable to dedicated, very efficient, algorithms

What about our graph?

Weights can be negative

- ...So one may think of using the Bellman-Ford algorithm, which runs in $O(n_v n_e)$

...But this is a Direct, Acyclic Graph (DAG)

- Meaning that we can process the nodes in topological order
- ...And apply Dijkstra algorithm, which runs in $O(n_e)$

Dijkstra's Algorithm for DAGs

Intuitively, we proceed as follows

- $Q = [0]$ # We enqueue the fake source node
- $sp_i = [], \forall i = 0..n_v$ # All shortest paths are empy
- while $|Q| > 0$:
  - pop a node i from Q
  - append i to sp_i# Extend the shortest path
  - for j successor of i:
    - mark the arc (i, j) as visited
    - if the shortest path passing for i is shorter than sp_j
      - update sp_j# Keep only the shortest path to j
    - if all ingoing arcs for j have been visited
      - append j in Q

Pricing via Shortest Paths

The approach is implemented in the `solve_pricing_problem` function

...Which returns shortest paths to all TUG nodes

In [19]:
```
ncosts_a, npaths_a = util.solve_pricing_problem(tug, rflows0, rpaths0,
                                                 node_counts, arc_counts, filter_paths=False)
print('COST: PATH')
util.print_solution(tug, ncosts_a, npaths_a, sort=None)
```

```
COST: PATH
-39.66: 0,2 > 1,2 > 2,2 > 3,2
-31.37: 0,1 > 1,1 > 2,0 > 3,0
-31.37: 0,1 > 1,1 > 2,0 > 3,3
-27.36: 0,2 > 1,2 > 2,2
-23.18: 0,1 > 1,1 > 2,0
-23.18: 0,1 > 1,1 > 2,0 > 3,1
-16.42: 0,2 > 1,2
-14.68: 0,1 > 1,1
-14.68: 0,1 > 1,1 > 2,1
-11.77: 0,1 > 1,0 > 2,3
-5.47: 0,2
-4.89: 0,1
-3.60: 0,1 > 1,0
0.00: 0,3
0.00: 1,3
4.61: 0,0
```

Pricing via Shortest Paths

We can ask for paths with a negative weighs/gradient term

In [20]:
```
ncosts, npaths = util.solve_pricing_problem(tug, rflows0, rpaths0,
                                             node_counts, arc_counts, filter_paths=True)
print('COST: PATH')
util.print_solution(tug, ncosts, npaths, sort=None)
```

```
COST: PATH
-39.66: 0,2 > 1,2 > 2,2 > 3,2
-31.37: 0,1 > 1,1 > 2,0 > 3,0
-31.37: 0,1 > 1,1 > 2,0 > 3,3
-27.36: 0,2 > 1,2 > 2,2
-23.18: 0,1 > 1,1 > 2,0
-23.18: 0,1 > 1,1 > 2,0 > 3,1
-16.42: 0,2 > 1,2
-14.68: 0,1 > 1,1
-14.68: 0,1 > 1,1 > 2,1
-11.77: 0,1 > 1,0 > 2,3
-5.47: 0,2
-4.89: 0,1
-3.60: 0,1 > 1,0
```

- Returning multiple paths is usually a good idea
- ...Since it typically speeds up the convergence of our dynamic method

Let's Loop!

Time to start iterating

Does it Work?

Every complex endeavor is worth a double (or triple) check

Let's check again our baseline result:

In [21]:
```
rflows0, rpaths0 = util.solve_path_selection_full(tug, node_counts, arc_counts,
                                                  initial_paths=path_pool, verbose=0)
print('FLOW: PATH')
util.print_solution(tug, rflows0, rpaths0, sort='descending')
sse = util.get_reconstruction_error(tug, rflows0, rpaths0, node_counts, arc_counts)
print(f'RSSE: {np.sqrt(sse):.2f}')
```

```
FLOW: PATH
1.96: 0,0 > 1,0 > 2,0 > 3,2
1.86: 0,0 > 1,0 > 2,0 > 3,3
0.79: 0,0 > 1,0 > 2,0 > 3,0
RSSE: 25.58
```

Does it Work?

Every complex endeavor is worth a double (or triple) check

Let's try adding paths with non-negative gradient terms

In [22]:
```
p_paths = [p for p, c in zip(npaths_a, ncosts_a) if c >= 0]
path_pool1_p = path_pool + p_paths

rflows1_p, rpaths1_p = util.solve_path_selection_full(tug, node_counts, arc_counts,
                                                  initial_paths=path_pool1_p, verbose=0)
print('FLOW: PATH')
util.print_solution(tug, rflows1_p, rpaths1_p, sort='descending')
sse = util.get_reconstruction_error(tug, rflows1_p, rpaths1_p, node_counts, arc_counts)
print(f'RSSE: {np.sqrt(sse):.2f}')
```

```
FLOW: PATH
1.96: 0,0 > 1,0 > 2,0 > 3,2
1.86: 0,0 > 1,0 > 2,0 > 3,3
0.79: 0,0 > 1,0 > 2,0 > 3,0
RSSE: 25.58
```

Does it Work?

Every complex endeavor is worth a double (or triple) check

Let's try adding paths with non-negative gradient terms

In [22]:
```
p_paths = [p for p, c in zip(npaths_a, ncosts_a) if c >= 0]
path_pool1_p = path_pool + p_paths

rflows1_p, rpaths1_p = util.solve_path_selection_full(tug, node_counts, arc_counts,
                                                      initial_paths=path_pool1_p, verbose=0)
print('FLOW: PATH')
util.print_solution(tug, rflows1_p, rpaths1_p, sort='descending')
sse = util.get_reconstruction_error(tug, rflows1_p, rpaths1_p, node_counts, arc_counts)
print(f'RSSE: {np.sqrt(sse):.2f}')
```

```
FLOW: PATH
1.96: 0,0 > 1,0 > 2,0 > 3,2
1.86: 0,0 > 1,0 > 2,0 > 3,3
0.79: 0,0 > 1,0 > 2,0 > 3,0
RSSE: 25.58
```

- This is perfectly useless
- ...Just as expected!

Does it Work?

Every complex endeavor is worth a double (or triple) check

Now, let's try again with paths having negative gradient terms

In [23]:
```
n_paths = [p for p, c in zip(npaths_a, ncosts_a) if c < 0]
path_pool1 = path_pool + n_paths
rflows1, rpaths1 = util.solve_path_selection_full(tug, node_counts, arc_counts,
                                                  initial_paths=path_pool1, verbose=0)
print('FLOW: PATH')
util.print_solution(tug, rflows1, rpaths1, sort='descending')
sse = util.get_reconstruction_error(tug, rflows1, rpaths1, node_counts, arc_counts)
print(f'RSSE: {np.sqrt(sse):.2f}')
```

```
FLOW: PATH
5.79: 0,2 > 1,2 > 2,2 > 3,2
3.05: 0,1 > 1,1 > 2,0 > 3,3
2.52: 0,1 > 1,1 > 2,0 > 3,0
1.70: 0,1 > 1,0 > 2,3
1.13: 0,0 > 1,0 > 2,0 > 3,2
0.87: 0,0 > 1,0 > 2,0 > 3,3
0.34: 0,0 > 1,0 > 2,0 > 3,0
RSSE: 15.18
```

Does it Work?

Every complex endeavor is worth a double (or triple) check

Now, let's try again with paths having negative gradient terms

In [23]:
```
n_paths = [p for p, c in zip(npaths_a, ncosts_a) if c < 0]
path_pool1 = path_pool + n_paths
rflows1, rpaths1 = util.solve_path_selection_full(tug, node_counts, arc_counts,
                                                  initial_paths=path_pool1, verbose=0)
print('FLOW: PATH')
util.print_solution(tug, rflows1, rpaths1, sort='descending')
sse = util.get_reconstruction_error(tug, rflows1, rpaths1, node_counts, arc_counts)
print(f'RSSE: {np.sqrt(sse):.2f}')
```

```
FLOW: PATH
5.79: 0,2 > 1,2 > 2,2 > 3,2
3.05: 0,1 > 1,1 > 2,0 > 3,3
2.52: 0,1 > 1,1 > 2,0 > 3,0
1.70: 0,1 > 1,0 > 2,3
1.13: 0,0 > 1,0 > 2,0 > 3,2
0.87: 0,0 > 1,0 > 2,0 > 3,3
0.34: 0,0 > 1,0 > 2,0 > 3,0
RSSE: 15.18
```

This time we have a better solution!

The Column Generation Code

Our CG code can be found in the `trajectory_extraction_cg` function

First, we define our initial path pool

```
paths = [[v.index] for v in tug.vs]
```

- We use one path per node, consisting of the node itself

Then we start looping:

```
for it in range(max_iter):
    # Solve the master problem
    ...
    # Solve the pricing problem
    ...
```

- We control the total run-time via an iteration limit

The Column Generation Code

Our CG code can be found in the `trajectory_extraction_cg` function

When we add the new paths, we take care of discarding duplicates

```
old_as_set = set([tuple(p) for p in paths])
found_as_set = set([tuple(p) for p in np])
new_as_set = old_as_set.union(found_as_set)
```

- Duplicates should not theoretically arise
- ...But they may in practice due to numerical errors
- ...Or when we use approximate solvers

We trigger an eary stop if no new path can be added:

```
if nnew == 0: break
```

Column Generation in Action

Let's test CG on that graph that took ~10 sec with the baseline

In [24]:
```
g8_5, t8_5, f8_5, p8_5, nc8_5, ac8_5 = util.get_default_benchmark_graph(nnodes=8, eoh=5, seed=42
%time f8_5, p8_5 = util.trajectory_extraction_cg(t8_5, nc8_5, ac8_5, max_iter=30, verbose=1)
```

```
It.0, sse: 310.26, #paths: 45, new: 5
It.1, sse: 91.33, #paths: 46, new: 1
It.2, sse: 0.00, #paths: 46, new: 0
CPU times: user 66.6 ms, sys: 17.1 ms, total: 83.7 ms
Wall time: 69.4 ms
```

What if the graph is bigger?

In [25]:
```
g20_7, t20_7, f20_7, p20_7, nc20_7, ac20_7 = util.get_default_benchmark_graph(nnodes=20, eoh=7,
%time f20_7, p20_7 = util.trajectory_extraction_cg(t20_7, nc20_7, ac20_7, max_iter=30, verbose=1
```

```
It.0, sse: 160.78, #paths: 176, new: 36
It.1, sse: 11.04, #paths: 177, new: 1
It.2, sse: 7.59, #paths: 189, new: 12
It.3, sse: 0.00, #paths: 189, new: 0
CPU times: user 372 ms, sys: 3.68 ms, total: 375 ms
Wall time: 370 ms
```

Column Generation in Action

Let's scale up even more

With this graph, the total number of paths is $O(40^{10})$

```
In [26]: g40_10, t40_10, f40_10, p40_10, nc40_10, ac40_10 = util.get_default_benchmark_graph(nnodes=40, e
         %time f40_10, p40_10 = util.trajectory_extraction_cg(t40_10, nc40_10, ac40_10, max_iter=30, verb

         It.0, sse: 947.53, #paths: 572, new: 172
         It.1, sse: 450.66, #paths: 676, new: 104
         It.2, sse: 134.47, #paths: 765, new: 89
         It.3, sse: 9.58, #paths: 774, new: 9
         It.4, sse: 8.88, #paths: 943, new: 169
         It.5, sse: 8.00, #paths: 1088, new: 145
         It.6, sse: 0.00, #paths: 1088, new: 0
         CPU times: user 7.42 s, sys: 20.2 ms, total: 7.44 s
         Wall time: 7.42 s
```

- The adds a small fraction of the total paths
- Convergence is fast
- ...And we manage to prove optimality!

Some Considerations

Column Generation is not easy approach to setup

...But when it works, it can provide many advantages

- The master can stay remarkably clean
- Complicated constraints can be moved in the variable definitions
- ...And tackled in the pricing problem
- Scalability is pretty good, given the humongous search space

CG makes you want to write models with massive number of variables

Some caveats

- A heuristic may still be faster (no sound mathematical theory, though...)
- It works well when you can put all its advantages to use
- ...In particular, the master problem structure should be very clean