Symmetries

Sometimes metrics are not enough

Unexpected Discrepancy

Our current solution seems apparently perfect

In [14]:
```
util.print_solution(tug, rflows, rpaths, sort='descending')
sse = util.get_reconstruction_error(tug, rflows, rpaths, node_counts, arc_counts)
print(f'RSSE: {np.sqrt(sse):.2f}')
```

```
8.17: 2,3 > 3,3
5.47: 0,2 > 1,2 > 2,2 > 3,2
3.74: 3,3
2.81: 0,1 > 1,1 > 2,0 > 3,0
2.09: 0,1 > 1,1 > 2,0 > 3,2
2.09: 1,0 > 2,0 > 3,0
1.24: 1,0 > 2,0 > 3,2
RSSE: 0.00
```

...And yet it does not match the ground truth!

In [7]:
```
util.print_ground_truth(flows, paths, sort='descending')
```

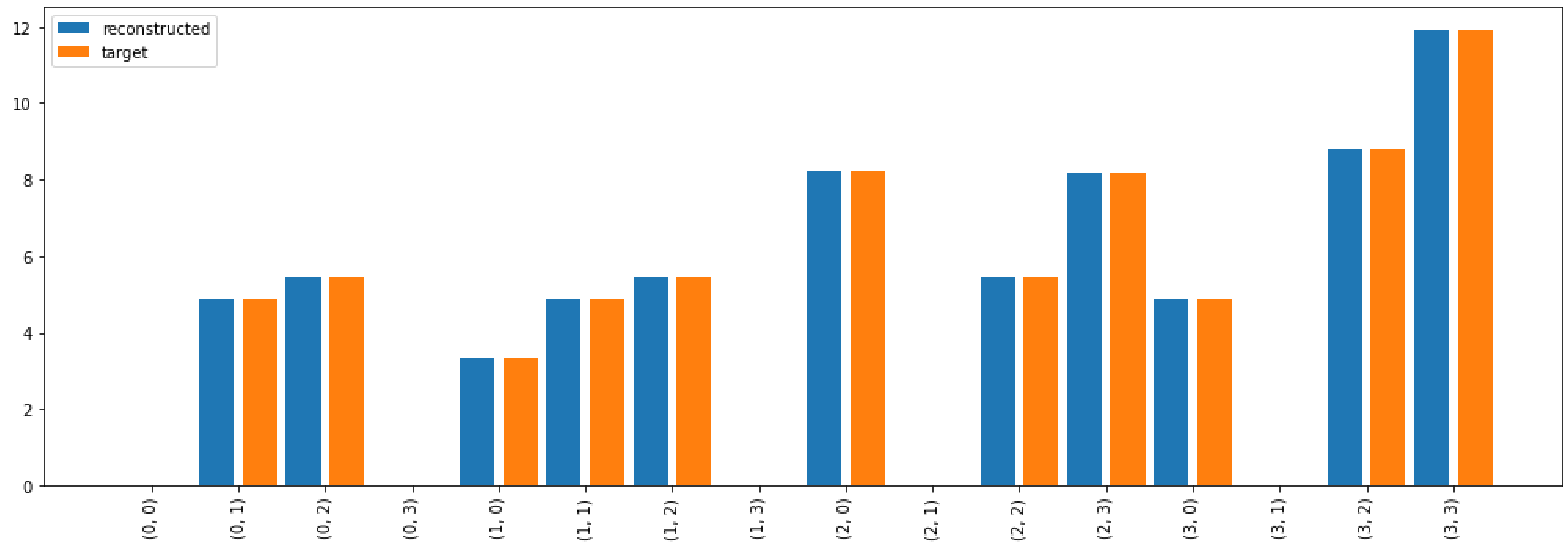
```
8.17: 2,3 > 3,3
5.47: 0,2 > 1,2 > 2,2 > 3,2
4.89: 0,1 > 1,1 > 2,0 > 3,0
3.74: 3,3
3.32: 1,0 > 2,0 > 3,2
```

Unexpected Discrepancy

The discrepancy is unexpected, due to the 0 reconstruction error

Indeed, we can check that the reconstructed counts match the true ones:

In [15]:
```
rnc, rac = util.get_counts(tug, rflows, rpaths)
util.plot_dict(rnc, figsize=figsize, label='reconstructed', data2=node_counts, label2='target',
```



Unexpected Discrepancy

What is going on?

Unexpected Discrepancy

What is going on?

We mentioned early on that the available information is poor

- There are many possible paths
- ...And many possible ways to explain the original counts!

How do we fix these symmetries?

Unexpected Discrepancy

What is going on?

We mentioned early on that the available information is poor

- There are many possible paths
- ...And many possible ways to explain the original counts!

How do we fix these symmetries?

- The only way is adding external information (e.g. a preference on paths)
- We can view this as a form of regularization

Occam's Razor

Intuitively, we could give priority to the simplest explanation

Image credit: xkcd 2541

A reasonable choice may be to use a small number of paths

How do we enforce this?

L1 Regularization and Path Number

We may think of using an L1 regularization

We would just need to add a linear term to the path formulation:

$$\mathop{\mathrm{arg\,min}}_x \left\{ \frac{1}{2} x^T P x + q^T x + \alpha x \mid x \geq 0 \right\}$$

...Which would translate into a correction on the q vector:

$$\mathop{\mathrm{arg\,min}}_x \left\{ \frac{1}{2} x^T P x + (q^T + \alpha) x \mid x \geq 0 \right\}$$

- This trick is implemented in the `solve_path_selection_full` function
- We just need to pass a value for the `alpha` argument

L1 Regularization and Path Number

Let's begin by trying $\alpha = 1$

In [16]:
```
rflows2, rpaths2 = util.solve_path_selection_full(tug, node_counts, arc_counts, verbose=0, alpha
print('FLOW: PATH')
util.print_solution(tug, rflows2, rpaths2, sort='descending')
sse = util.get_reconstruction_error(tug, rflows2, rpaths2, node_counts, arc_counts)
print(f'\nRSSE: {np.sqrt(sse):.2f}')
```

```
FLOW: PATH
8.10: 2,3 > 3,3
5.37: 0,2 > 1,2 > 2,2 > 3,2
2.58: 0,1 > 1,1 > 2,0 > 3,0
2.36: 3,3
1.98: 1,0 > 2,0 > 3,0
1.90: 0,1 > 1,1 > 2,0 > 3,2
1.17: 1,0 > 2,0 > 3,2
0.36: 0,1 > 1,1 > 2,0 > 3,3
0.06: 1,0 > 2,3 > 3,3
0.02: 0,1 > 1,0 > 2,0 > 3,0
0.02: 1,0 > 2,0 > 3,3

RSSE: 1.30
```

L1 Regularization and Path Number

Let's begin by trying $\alpha = 1$

```
In [16]: rflows2, rpaths2 = util.solve_path_selection_full(tug, node_counts, arc_counts, verbose=0, alpha
         print('FLOW: PATH')
         util.print_solution(tug, rflows2, rpaths2, sort='descending')
         sse = util.get_reconstruction_error(tug, rflows2, rpaths2, node_counts, arc_counts)
         print(f'\nRSSE: {np.sqrt(sse):.2f}')
```

```
FLOW: PATH
8.10: 2,3 > 3,3
5.37: 0,2 > 1,2 > 2,2 > 3,2
2.58: 0,1 > 1,1 > 2,0 > 3,0
2.36: 3,3
1.98: 1,0 > 2,0 > 3,0
1.90: 0,1 > 1,1 > 2,0 > 3,2
1.17: 1,0 > 2,0 > 3,2
0.36: 0,1 > 1,1 > 2,0 > 3,3
0.06: 1,0 > 2,3 > 3,3
0.02: 0,1 > 1,0 > 2,0 > 3,0
0.02: 1,0 > 2,0 > 3,3

RSSE: 1.30
```

- The RSSE grows (as it could be expcted)
- But we have more paths!

L1 Regularization and Path Number

What if we make α larger?

In [17]:
```
rflows2, rpaths2 = util.solve_path_selection_full(tug, node_counts, arc_counts, verbose=0, alpha
print('FLOW: PATH')
util.print_solution(tug, rflows2, rpaths2, sort='descending')
sse = util.get_reconstruction_error(tug, rflows2, rpaths2, node_counts, arc_counts)
print(f'\nRSSE: {np.sqrt(sse):.2f}')
```

```
FLOW: PATH
4.76: 2,3 > 3,3
4.27: 0,2 > 1,2 > 2,2 > 3,2
1.83: 0,1 > 1,1 > 2,0 > 3,0
1.42: 0,1 > 1,1 > 2,0 > 3,2
0.84: 0,1 > 1,1 > 2,0 > 3,3
0.82: 1,0 > 2,3 > 3,3
0.77: 1,0 > 2,0 > 3,0
0.29: 1,0 > 2,0 > 3,2
0.19: 0,1 > 1,0 > 2,3 > 3,3
0.15: 0,1 > 1,0 > 2,0 > 3,0
0.06: 0,1 > 1,0 > 2,0 > 3,2
0.04: 1,0 > 2,0 > 3,3
0.04: 0,1 > 1,0 > 2,0 > 3,3
0.02: 0,0 > 1,0 > 2,0 > 3,2

RSSE: 9.11
```

L1 Regularization and Path Number

What if we make α larger?

In [17]:
```
rflows2, rpaths2 = util.solve_path_selection_full(tug, node_counts, arc_counts, verbose=0, alpha
print('FLOW: PATH')
util.print_solution(tug, rflows2, rpaths2, sort='descending')
sse = util.get_reconstruction_error(tug, rflows2, rpaths2, node_counts, arc_counts)
print(f'\nRSSE: {np.sqrt(sse):.2f}')
```

```
FLOW: PATH
4.76: 2,3 > 3,3
4.27: 0,2 > 1,2 > 2,2 > 3,2
1.83: 0,1 > 1,1 > 2,0 > 3,0
1.42: 0,1 > 1,1 > 2,0 > 3,2
0.84: 0,1 > 1,1 > 2,0 > 3,3
0.82: 1,0 > 2,3 > 3,3
0.77: 1,0 > 2,0 > 3,0
0.29: 1,0 > 2,0 > 3,2
0.19: 0,1 > 1,0 > 2,3 > 3,3
0.15: 0,1 > 1,0 > 2,0 > 3,0
0.06: 0,1 > 1,0 > 2,0 > 3,2
0.04: 1,0 > 2,0 > 3,3
0.04: 0,1 > 1,0 > 2,0 > 3,3
0.02: 0,0 > 1,0 > 2,0 > 3,2

RSSE: 9.11
```

L1 Regularization and Path Number

Shouldn't L1 norm work as a sparsifier?

Not exactly: it simply results in a fixed penalty rate for raising a variable

- The solver will try to balance it with a larger reduction of the quadratic loss
- ...Which we can easily improve by including more nodes in each path

L1 Regularization and Path Number

Shouldn't L1 norm work as a sparsifier?

Not exactly: it simply results in a fixed penalty rate for raising a variable

- The solver will try to balance it with a larger reduction of the quadratic loss
- ...Which we can easily improve by including more nodes in each path

The truth is that when we use an L1 norm as sparsifier...

...We really wished our regularizer to be:

$$N_{paths} = \sum_{j=1}^{n} z_j \quad \text{with: } z_j = \begin{cases} 1 \text{ if } x_j > 0 \\ 0 \text{ otherwise} \end{cases}$$

- Which is inconvenient, since it is non-differentiable
- ...But what if we used an approach for non-differentiable optimization?

Path Consolidation Problem

Let's face an inconvenient truth

Path Consolidation Problem

For example, we could focus on the paths in the current solution:

- ...Minimize the number of used paths
- ...While preserving our reconstruction error

This is form of symmetry breaking (as a post-processing step)

By doing this, we obtain a "path consolidation problem" in the form:

$$
\begin{aligned}
\arg\min_{x} \quad & \|z\|_1 \\
\text{subject to:} \quad & Vx = v^* \\
& Ex = e^* \\
& x \leq Mz \\
& x \geq 0 \\
& z \in \{0, 1\}^n
\end{aligned}
$$

Path Consolidation Problem

Let's proceed to examine the formulation a bit better:

$$
\begin{aligned}
\underset{x}{\arg\min} \quad & \|z\|_1 \\
\text{subject to:} \quad & Vx = v^* \\
& Ex = e^* \\
& x \leq Mz \\
& x \geq 0 \\
& z \in \{0, 1\}^n
\end{aligned}
$$

- The terms V, E, and x are the same as before
- ...Except in this case we will consider a a subset of the paths
- v^* and e^* are the counts from the optimal path formulation solution
- We are requiring the (reconstructed) counts to be exactly the same

Path Consolidation Problem

Let's proceed to examine the formulation a bit better:

$$
\begin{aligned}
\arg\min_{x} \quad & \|z\|_1 \\
\text{subject to:} \quad & Vx = v^* \\
& Ex = e^* \\
& x \leq Mz \\
& x \geq 0 \\
& z \in \{0, 1\}^n
\end{aligned}
$$

- The z variables determine whether a path is used ($z_j = 1$) or not ($z_j = 0$)
- M is a constant large enough to make the constraint trivial if $z_j = 1$
- Constants such as these are often referred to as "big-Ms"
- Basically, $x \leq Mz$ is a linearization of the implication $x > 0 \Rightarrow z = 1$

Path Consolidation Problem

Let's proceed to examine the formulation a bit better:

$$
\begin{aligned}
\arg\min_{x} \quad & \|z\|_1 \\
\text{subject to:} \quad & Vx = v^* \\
& Ex = e^* \\
& x \leq Mz \\
& x \geq 0 \\
& z \in \{0, 1\}^n
\end{aligned}
$$

- All constraints are linear
- The cost function is linear
- Some variables are integer

This is a Mixed Integer Linear Program (MILP)