Solving the Path Formulation

Which will be our baseline

Solving the Path Formulation

The path formulation consists in the Quadratic Program:

$$\arg\min_{x} \left\{ \frac{1}{2} x^T P x + q^T x \mid x \geq 0 \right\}$$

Where $\boldsymbol{P} = \boldsymbol{V}^T\boldsymbol{V} + \boldsymbol{E}^T\boldsymbol{E}$ and $\boldsymbol{q} = -\boldsymbol{V}^T\boldsymbol{v} - \boldsymbol{E}^T\boldsymbol{e}$

Solving the Path Formulation

The path formulation consists in the Quadratic Program:

$$\arg\min_x \left\{ \frac{1}{2} x^T P x + q^T x \mid x \geq 0 \right\}$$

Where $P = V^T V + E^T E$ and $q = -V^T v - E^T e$

Therefore, if we want to solve the problem we need:

- The binary matrix V, s.t. $V_{ij} = 1$ iff node i is in path j
- The binary matrix E, s.t. $E_{kj} = 1$ iff arc k is in path j
- The vector v, containing the node counts
- The vector e, containing the arc counts

In turn, to get these we need to define a set of paths on the TUG

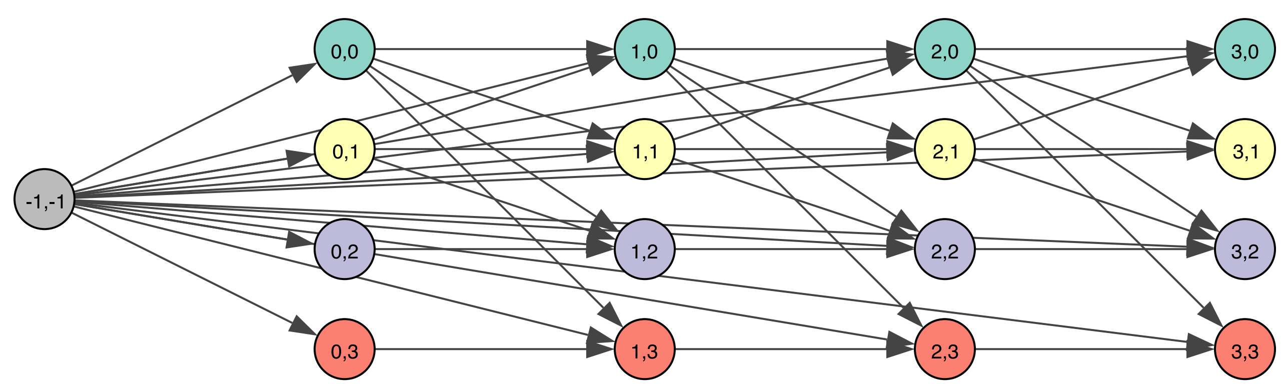
Path Enumeration

Unless we want to loose optimality, we need to consider all the TUG paths

First, we augment the Time Unfolded Graph with a fake source node

In [2]:
```
tugs, tugs_source = util._add_source_to_tug(tug)
ig.plot(tugs, **util.get_visual_style(tugs), bbox=(700, 250), margin=50)
```

Out[2]:



- The node is associate to the time step -1 and (original) index -1

Path Enumeration

Then we can use a depth-first traversal to enumerate all paths

In [3]:
```
tug_paths = util.enumerate_paths(tugs, tugs_source, exclude_source=True)
for i, p in enumerate(tug_paths):
    print(f'{i}: {p}')
```

```
0: [0]
1: [0, 4]
2: [0, 4, 8]
3: [0, 4, 8, 12]
4: [0, 4, 8, 13]
5: [0, 4, 8, 14]
6: [0, 4, 8, 15]
7: [0, 4, 9]
8: [0, 4, 9, 12]
9: [0, 4, 9, 13]
10: [0, 4, 9, 14]
11: [0, 4, 10]
12: [0, 4, 10, 14]
13: [0, 4, 11]
14: [0, 4, 11, 15]
15: [0, 5]
16: [0, 5, 8]
17: [0, 5, 8, 12]
18: [0, 5, 8, 13]
19: [0, 5, 8, 14]
20: [0, 5, 8, 15]
21: [0, 5, 9]
```

Path Enumeratation

By default we use TUG node indexes, but we can plot the original ones:

In [4]:
```python
tmp = util.tug_paths_to_original(tugs, tug_paths)
for i, p in enumerate(tmp):
    print(f'{i}: {p}')
```

```
0: [(0, 0)]
1: [(0, 0), (1, 0)]
2: [(0, 0), (1, 0), (2, 0)]
3: [(0, 0), (1, 0), (2, 0), (3, 0)]
4: [(0, 0), (1, 0), (2, 0), (3, 1)]
5: [(0, 0), (1, 0), (2, 0), (3, 2)]
6: [(0, 0), (1, 0), (2, 0), (3, 3)]
7: [(0, 0), (1, 0), (2, 1)]
8: [(0, 0), (1, 0), (2, 1), (3, 0)]
9: [(0, 0), (1, 0), (2, 1), (3, 1)]
10: [(0, 0), (1, 0), (2, 1), (3, 2)]
11: [(0, 0), (1, 0), (2, 2)]
12: [(0, 0), (1, 0), (2, 2), (3, 2)]
13: [(0, 0), (1, 0), (2, 3)]
14: [(0, 0), (1, 0), (2, 3), (3, 3)]
15: [(0, 0), (1, 1)]
16: [(0, 0), (1, 1), (2, 0)]
17: [(0, 0), (1, 1), (2, 0), (3, 0)]
18: [(0, 0), (1, 1), (2, 0), (3, 1)]
19: [(0, 0), (1, 1), (2, 0), (3, 2)]
20: [(0, 0), (1, 1), (2, 0), (3, 3)]
21: [(0, 0), (1, 1), (2, 1)]
```

Building the Matrices and Vectors

Now we can build the V and E matrices and the v and e vectors

These define the least squares terms $\|Vx - v\|_2^2$ and $\|Ex - e\|_2^2$

In [5]:
```
V, E = util._paths_to_coefficient_matrices(tug, tug_paths)
v, e = util._counts_to_target_vectors(tug, node_counts, arc_counts)
```

Building the Matrices and Vectors

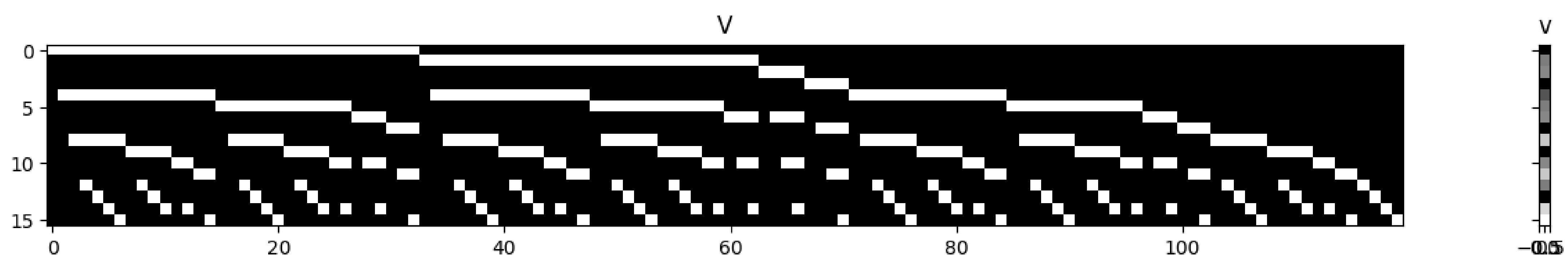
Now we can build the V and E matrices and the v and e vectors

These define the least squares terms $\|Vx - v\|_2^2$ and $\|Ex - e\|_2^2$

In [5]:
```
V, E = util._paths_to_coefficient_matrices(tug, tug_paths)
v, e = util._counts_to_target_vectors(tug, node_counts, arc_counts)
```

Here's a visualization of the V, v pair:

In [6]:
```
util.plot_matrix(V.toarray(), v, figsize=figsize, title='V', title_b='v')
```



Building the Matrices and Vectors

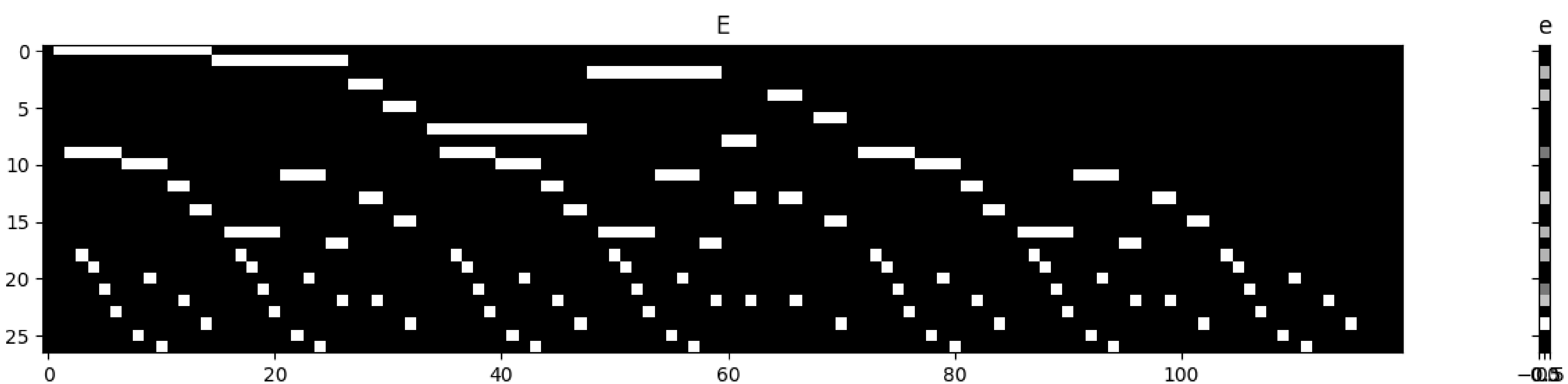
Now we can build the V and E matrices and the v and e vectors

These define the least squares terms $\|Vx - v\|_2^2$ and $\|Ex - e\|_2^2$

In [7]:
```
V, E = util._paths_to_coefficient_matrices(tug, tug_paths)
v, e = util._counts_to_target_vectors(tug, node_counts, arc_counts)
```

Here's the same for the E, e pair:

In [8]:
```
util.plot_matrix(E.toarray(), e, figsize=figsize, title='E', title_b='e')
```



Solving the QP Problem

The code for solving the QP is in the `solve_path_selection_full` function

Here's a relevant snippet:

```
# Enumerate all paths
tugs, tugs_source = _add_source_to_tug(tug)
paths = enumerate_paths(tugs, tugs_source, exclude_source=True)
# Build the path selection solver
prb = PathSelectionSolver(tug, node_counts, arc_counts)
# Solve the path selection problem
sol = prb.solve(paths, verbose=verbose, **settings)
```

- First we build a `PathSelectionSolver`, i.e. a custom class from the `util` module
- Then we call the `solve` method
- Otherwise, we get a (feasible, but) approximate solution

Solving the QP Problem

In turn, `PathSelectionSolver.solve` contains the following code:

```python
# Build the solver
self.mdl = piqp.SparseSolver()
# Recompute the problem matrices
P, c, A, b, G, h, x_lb, x_ub = self._recompute_matrices_piqp(paths)
# Setup the solver
self.mdl.setup(P, c, A, b, G, h, x_lb, x_ub)
# Solve the problem
sol = self.mdl.solve()
```

This is how we use the actual PIQP solver

- We build an `SparseSolver` object, we call `setup`, then we cal `solve`
- ...But first we need to compute a bunch of matrix and vector terms

Solving the QP Problem

Matrix construction happens in the `_recompute_matrices` function

We already know that:

$$P = V^T V + E^T E \quad \text{and} \quad q = -V^T v - E^T e$$

- Where V and E specify which nodes/arcs belong to each path
- ...And v and e are the counts for all TUG nodes and arcs

About the A matrix and l and u vectors

- They are meant to specify the problem constraints
- Since in our problem we have $x \geq 0$, then:

$$A = I \quad \text{and} \quad l = 0 \quad \text{and} \quad u = +\infty$$

Solving the QP Problem

Let's actually solve the problem and inspect the output

In [9]:
```
rflows, rpaths = util.solve_path_selection_full(tug, node_counts, arc_counts, verbose=1, so
```

Inspecting the Solution

The raw solver log does not relate much to our specific problem

But we can obtain clearer plots using some ad-hoc built functions:

```
In [10]: print('FLOW: PATH')
         util.print_solution(tug, rflows, rpaths, sort='descending')
         sse = util.get_reconstruction_error(tug, rflows, rpaths, node_counts, arc_counts)
         print(f'\nRSSE: {np.sqrt(sse):.2f}')
```

```
FLOW: PATH
8.17: 2,3 > 3,3
5.47: 0,2 > 1,2 > 2,2 > 3,2
3.74: 3,3
3.10: 0,1 > 1,1 > 2,0 > 3,0
1.79: 1,0 > 2,0 > 3,0
1.79: 0,1 > 1,1 > 2,0 > 3,2
1.53: 1,0 > 2,0 > 3,2

RSSE: 0.00
```

- We know see which paths have been used to "reconstruct" the counts
- The corresponding estimated flows
- And the Root Sum of Squared Errors