

## **From Pricing...**

---

Because there's an elephant in the room, and it's HUGE

# Scalability, or Lack Thereof

## Our current approach as one, massive, limitations

The number paths in graph scales **exponentially** on its size

- Meaning that path enumeration becomes quickly very expensive
- ...And the path formulation size grows at the same rate

## Let's check the solution time for our small example graph:

```
In [2]: %time rflows, rpaths = util.solve_path_selection_full(tug, node_counts, arc_counts, verbose=0)
```

```
CPU times: user 146 ms, sys: 8.7 ms, total: 155 ms  
Wall time: 149 ms
```

```
/usr/local/lib/python3.6/dist-packages/scipy/sparse/_index.py:82: SparseEfficiencyWarning: Changing the sparsity structure of a csc_matrix is expensive. lil_matrix is more efficient.  
  self._set_intXint(row, col, x.flat[0])
```

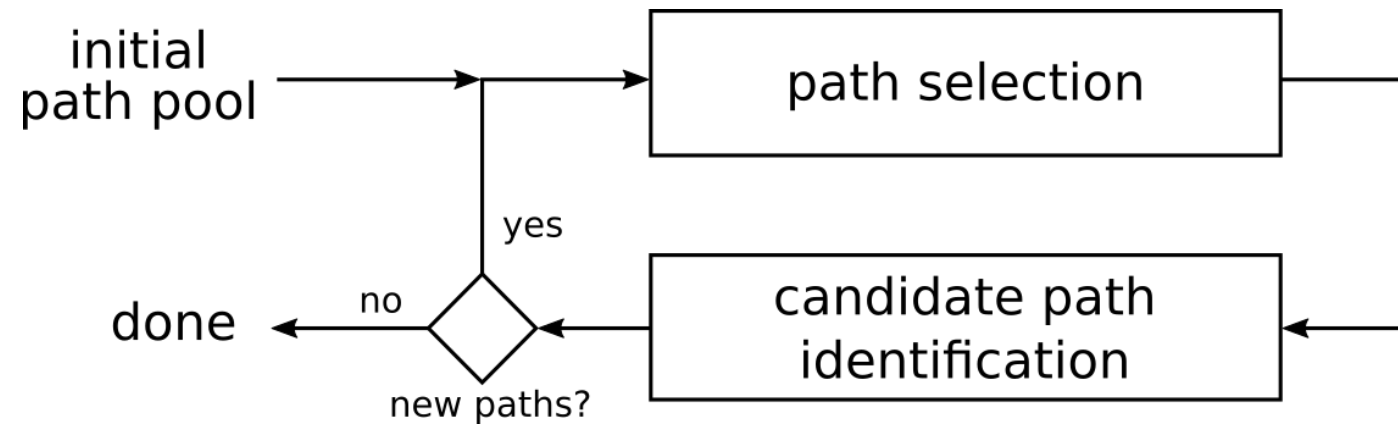
...And the for a slightly larger graph (8 nodes, 5 time steps):

```
In [3]: 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.solve_path_selection_full(t8_5, nc8_5, ac8_5, verbose=0)
```

# Adding Variables on Demand

What if we had a way to **add variables on demand**?

Then could think of:

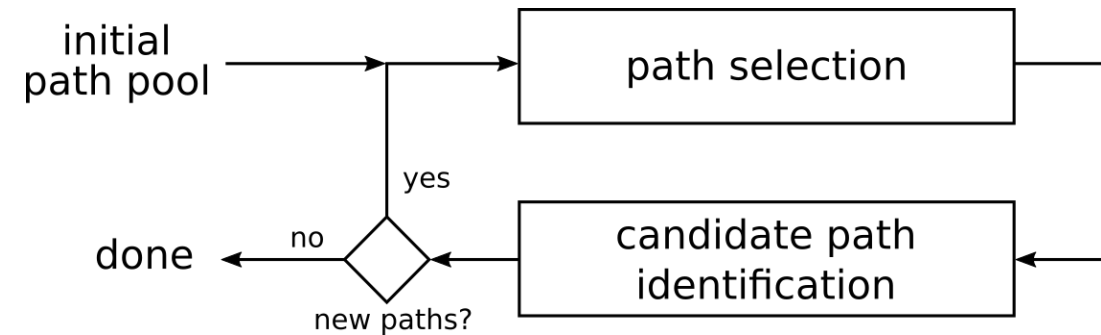


- Solving the Path Formulation with a **subset of paths**
- ...Then searching for new paths to be added
  - If we find some, we add them to the pool and we repeat
  - If we find none, we are done

**An approach such as this may strongly **mitigate our scalability issues****

# Adding Variables on Demand

What do we need to pull this off?

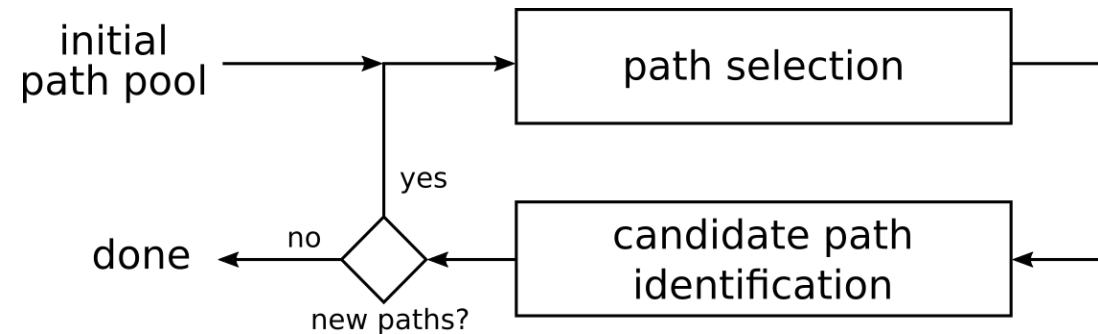


1. The ability to use a limited pool in the path formulation
2. A way to identify new paths to be added

Point 1 is trivial, but what about point 2?

# Adding Variables on Demand

What do we need to pull this off?



1. The ability to use a limited pool in the path formulation
2. A way to identify new paths to be added

Point 1 is trivial, but what about point 2?

**We could split the enumeration in multiple "chunks"**

- That would allow to obtain the first solutions more quickly
- But would still need to complete the enumeration to prove optimality

**What we need is a way to identify *useful* paths that are not yet in the pool**

# Identifying Useful Variables

Let's recall the structure of the Path Formulation

$$\arg \min_x \{ f(x) \mid x \geq 0 \} \quad \text{with: } f(x) = \frac{1}{2} x^T P x + q^T x$$

- We can view missing variables are having **value 0** in the current solution
- So, we are looking for variables that can be **raised to reduce error**
- Since the problem is convex, we could start by **looking at the gradient**

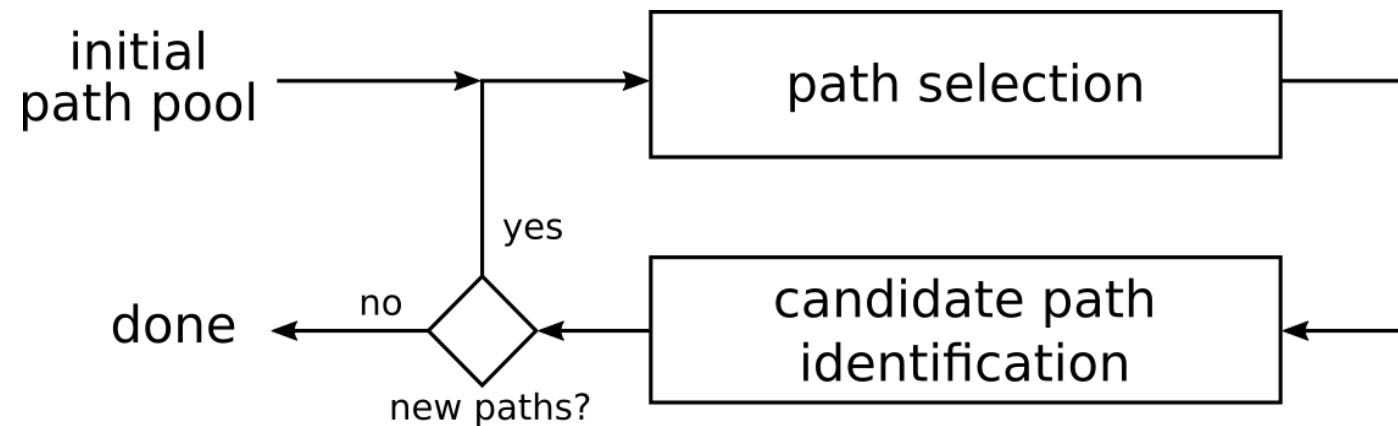
**Hence, we could search for variables with a **negative gradient term****

$$\frac{\partial}{\partial x_j} f(x) < 0$$

- This a **necessary** condition in general
- Later in the course we will found out why

# Pricing Problem

Let's revisit and generalize our schema



- The "main" problem may not involve paths
  - ...So we will call it just a **master problem**
- We look for additional variables such that  $\frac{\partial}{\partial x_j} f(x) < 0$ 
  - It's a bit like we are assigning a "price tag" to them
  - If the price is positive, we skip the variable (we know it's useless now)
  - If the price is negative, the variable **may** be useful
  - For this reasons, we call the second component **pricing problem**

# The Path Selection Gradient

We need to compute gradient terms for variables **not yet in the pool**

This is easiest if we differentiate our original (equivalent) objective:

$$f(x) = \frac{1}{2} \|Vx - \hat{v}\|_2^2 + \frac{1}{2} \|Ex - \hat{e}\|_2^2$$

- ...Since node contribution mix-up in the  $P$  matrix and  $q$  vectors

**The least square objective can be rewritten as:**

$$f(x) = \frac{1}{2} \sum_{i=1}^{n_v} \left( \sum_{j=1}^n V_{ij} x_j - \hat{v}_i \right)^2 + \frac{1}{2} \sum_{k=1}^{n_e} \left( \sum_{j=1}^n E_{kj} x_j - \hat{e}_k \right)^2$$

- Where  $n_v$  is the number of nodes and  $n_e$  the number of arcs



# The Path Selection Gradient

We can differentiate the expression to obtain

$$\frac{\partial}{\partial x_j} f(x) = \sum_{i=1}^{n_v} \left( \sum_{j=1}^n V_{ij} x_j - \hat{v}_i \right) V_{ij} + \sum_{k=1}^{n_e} \left( \sum_{j=1}^n E_{kj} x_j - \hat{e}_k \right) E_{kj}$$

Some expressions in the formula are simply the node/edge **residuals**:

$$r_i^v = \sum_{j=1}^n V_{ij} x_j - \hat{v}_i \quad \text{and} \quad r_k^e = \sum_{j=1}^n E_{kj} x_j - \hat{e}_k$$

Hence we can rewrite the gradient terms as:

$$\frac{\partial}{\partial x_j} f(x) = \sum_{i=1}^{n_v} r_i^v V_{ij} + \sum_{k=1}^{n_e} r_k^e E_{kj}$$

# The Path Selection Gradient

Now, let's parse the meaning of our gradient term:

$$\frac{\partial}{\partial x_j} f(x) = \sum_{i=1}^{n_v} r_i^v V_{ij} + \sum_{k=1}^{n_e} r_k^e E_{kj}$$

- For every (TUG) node  $i$  included in the path, we add  $r_i^v$
- For every (TUG) arc  $k$  included in the path, we add  $r_k^e$

**This is a simple computation that we can perform on any path**

...Including those that are not yet in the path formulation pool

- Just don't forget that this condition identifies (potentially) useful paths
- ...But only w.r.t. the current path formulation solution!

# A Look at the Residuals

## Let's try it out

First, we enumerate all TUG paths

```
In [4]: tugs, tugs_source = util._add_source_to_tug(tug)
        tug_paths = util.enumerate_paths(tugs, tugs_source, exclude_source=True)
```

Then, we run the path formulation with a limited pool of paths

```
In [5]: path_pool = tug_paths[:10]
        rflows0, rpaths0 = util.solve_path_selection_full(tug, node_counts, arc_counts,
                                                         initial_paths=path_pool, verbose=0)
        sse = util.get_reconstruction_error(tug, rflows0, rpaths0, node_counts, arc_counts)
        util.print_solution(tug, rflows0, rpaths0, sort='descending')
        print(f'RSSE: {np.sqrt(sse):.2f}')
```

```
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
```

- Since we are restricted to a subset of paths, the RSSE is no longer 0

# A Look at the Residuals

Then we can extract the residuals, i.e.  $Vx - \hat{v}$  and  $Ex - \hat{e}$

```
In [6]: nres0, ares0 = util._get_residuals(tug, rflows0, rpaths0, node_counts, arc_counts)
print('NODE RESIDUALS')
print('\t'.join(f'{k}:{v:.2f}' for k, v in nres0.items()))
print('ARC RESIDUALS')
print('\t'.join(f'{k}:{v:.2f}' for k, v in ares0.items()))
```

NODE RESIDUALS

(0, 0):4.61	(0, 1):-4.89	(0, 2):-5.47	(0, 3):0.00	(1, 0):1.29	(1, 1):-4.89
(1, 2):-5.47	(1, 3):0.00	(2, 0):-3.60	(2, 1):0.00	(2, 2):-5.47	(2, 3):-8.17
(3, 0):-4.10	(3, 1):0.00	(3, 2):-6.83	(3, 3):-10.05		

ARC RESIDUALS

(1, 0, 0):4.61	(1, 0, 1):0.00	(1, 1, 1):-4.89	(1, 0, 2):0.00	(1, 2, 2):-5.47	(1, 0, 3):0.00
(1, 3, 3):0.00	(1, 1, 0):0.00	(1, 1, 2):0.00	(2, 0, 0):1.29	(2, 0, 1):0.00	(2, 1, 1):0.00
(2, 0, 2):0.00	(2, 2, 2):-5.47	(2, 0, 3):0.00	(2, 3, 3):0.00	(2, 1, 0):-4.89	(2, 1, 2):0.00
(3, 0, 0):-4.10	(3, 0, 1):0.00	(3, 1, 1):0.00	(3, 0, 2):-1.36	(3, 2, 2):-5.47	(3, 0, 3):1.86
(3, 3, 3):-8.17	(3, 1, 0):0.00	(3, 1, 2):0.00			

- This is enough information to compute  $\frac{\partial}{\partial x_j} f(\mathbf{x})$  for all paths
- ...Except that doing that would still not solve all our issues :-)