

Constrained Optimization for Data Mining

Definitely niche, but also a great example

Constrained Optimization for Data Mining

Let's consider a data mining problem for web analytics



- A company wants to analyze user behavior on their web site
- ...With the goal of optimizing its structure
- For privacy reason, the company does not want to resort to tracking
- ...And plan to relies on simple page/link-click counts

Constrained Optimization for Data Mining

Our **input** consists of page and link **counts** for multiple time steps

t	0	1	...	0,1	0,3	1,2	...
0	35	12	...	21	7	9	...
1	42	14	...	22	11	10	...
2	38	9	...	17	10	8	...

- Each simple number refers to a page, each pair to a link
- Cells contain presence/link-click counts for different value of the time t

Our **output** consists of navigation paths on the web site

A path specifies which page is visited at every point of time, e.g.:

$$\{(2, 0), (3, 0), (4, 1), (5, 3)\}$$

- In this case the path starts at time **2**, stays at page **0** for two time units
- ...Then moves to **1** and then **3**

How would you tackle this problem?

Constrained Optimization for Data Mining

The main issue is representing and handling **paths**

- A path is **combinatorial** object (\Rightarrow not differentiable)
- Nodes in a path must be **connected**

In other words, the main issue is dealing with **constraints**

Constrained Optimization for Data Mining

The main issue is representing and handling **paths**

- A path is **combinatorial** object (\Rightarrow not differentiable)
- Nodes in a path must be **connected**

In other words, the main issue is dealing with **constraints**

We will see how to tackle the problem directly via Constrained Optimization

- The approach will work decently well
- ...But make no mistake: this won't be SotA

Our main purpose is seeing **many** CO methods in action!

Constrained Optimization for Data Mining

This is a **very challenging** problem!



- There are **many viable paths**!
- ...And we start with **quite poor information**

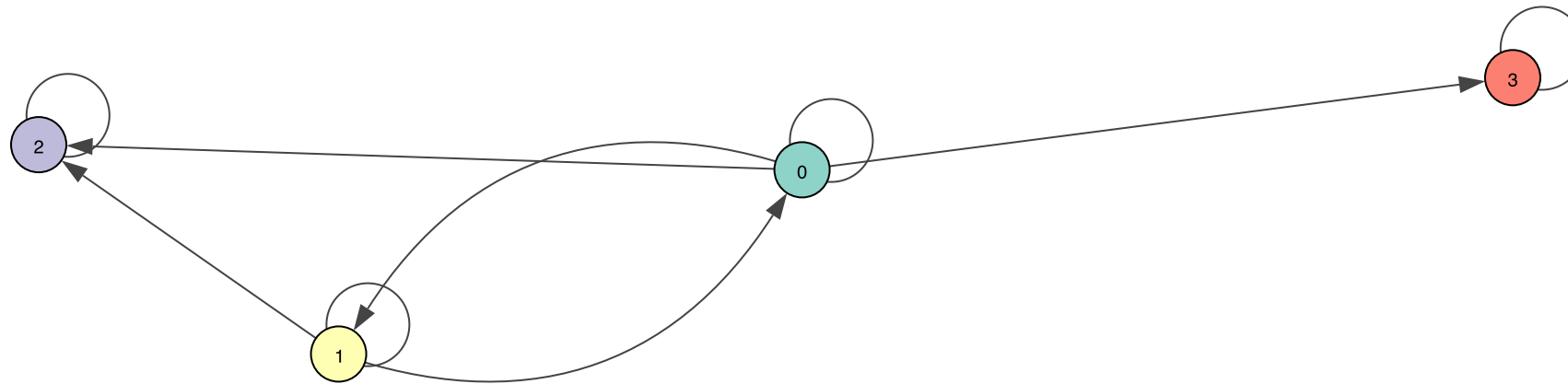
Web Site as Graph

Our web site can be represented as a **directed graph**

We will generate one at random, with a realistic structure

```
In [3]: g = util.build_website_graph(nnodes=4, rate=3, extra_arc_fraction=0.25, seed=42)  
ig.plot(g, **util.get_visual_style(g), bbox=ig_bbox, margin=50)
```

Out[3]:



- The method generates **nnodes** vertexes in a **tree structure** as a base
- The #children per vertex follows a **Poisson distribution** with specified rate
- ...Then a **fraction of the missing arcs** is added at random

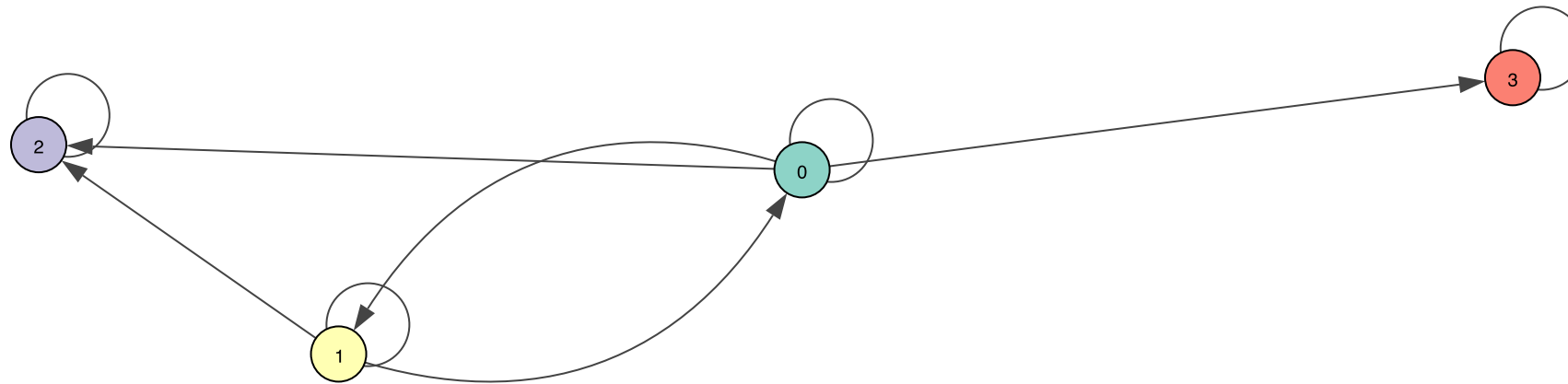
Web Site as Graph

Our web site can be represented as a **directed graph**

We will generate one at random, with a realistic structure

```
In [4]: g = util.build_website_graph(nnodes=4, rate=3, extra_arc_fraction=0.25, seed=42)  
ig.plot(g, **util.get_visual_style(g), bbox=ig_bbox, margin=50)
```

Out[4]:



- The graph is handled via the [python-igraph](#) library
- ...Which provides a fast C++ implementation of many graph primitives
- The library also include a good selection of graph algorithms

Ground Truth Generation

We obtain realistic counts by routing "flow" along random paths

For one path, this can be done via a function from the utility module:

```
In [5]: home = g.vs[0] # Home page
        eoh = 4 # End of Horizon

        flow, path = util.route_random_flow(home, min_units=1, max_units=10, eoh=eoh, seed=10)
        print(f'{flow:.2f}: {">".join(str(v) for v in path)}')
```

3.69: (1, 0)>(2, 3)>(3, 3)

- The first vertex represents the home page
- The "flow" represents the amount of users that traverse the path
- **eoh** is the number of time units over which we assume to have counts

Ground Truth Generation

A second function performs random routing for multiple paths

We will start from a simple example with a very small number of paths:

```
In [7]: flows, paths = util.build_random_paths(g, min_paths=3, max_paths=5,  
                                              min_units=1, max_units=10, eoh=eoh, seed=42)  
print('FLOW: PATH')  
util.print_ground_truth(flows, paths, sort='descending')
```

```
FLOW: PATH  
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
```

- Paths may start from any page
- Paths may start at any time step within the horizon

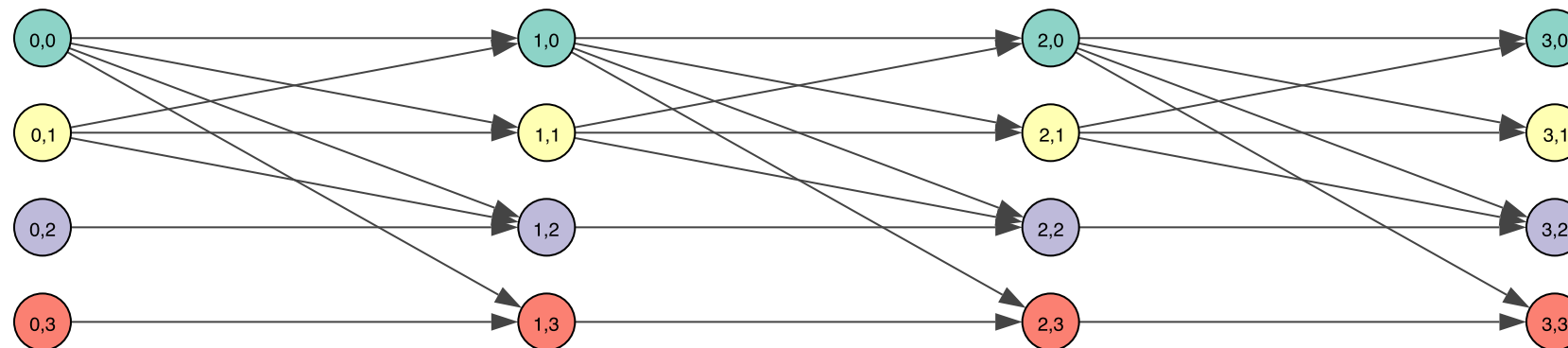
The generated paths represent our **ground truth**

Time-Unfolded Graph

Our paths may be seen as traversal of a **time-unfolded** version of the graph

```
In [9]: tug = util.build_time_unfolded_graph(g, eoh=eoh)  
ig.plot(tug, **util.get_visual_style(tug), bbox=ig_bbox, margin=50)
```

Out [9]:



- We create **eoh** replicas of the vertexes, each referring to a specific time step
- We create **eoh** replicas of the edges, linking vertexes in adjacent time step

This representation is referred to as **Time Unfolded Graph**

Computing Counts

We can now compute counts for all vertexes and edges in the TUG

```
In [10]: node_counts, arc_counts = util.get_counts(tug, flows, paths)
print('NODE COUNTS')
print('\t'.join(f'{k}:{v:.2f}' for k, v in node_counts.items()))
print('ARC COUNTS')
print('\t'.join(f'{k}:{v:.2f}' for k, v in arc_counts.items()))
```

NODE COUNTS

(0, 0):0.00	(0, 1):4.89	(0, 2):5.47	(0, 3):0.00	(1, 0):3.32	(1, 1):4.89
(1, 2):5.47	(1, 3):0.00	(2, 0):8.22	(2, 1):0.00	(2, 2):5.47	(2, 3):8.17
(3, 0):4.89	(3, 1):0.00	(3, 2):8.79	(3, 3):11.91		

ARC COUNTS

(1, 0, 0):0.00	(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):3.32	(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):4.89	(3, 0, 1):0.00	(3, 1, 1):0.00	(3, 0, 2):3.32	(3, 2, 2):5.47	(3, 0, 3):0.00
(3, 3, 3):8.17	(3, 1, 0):0.00	(3, 1, 2):0.00			

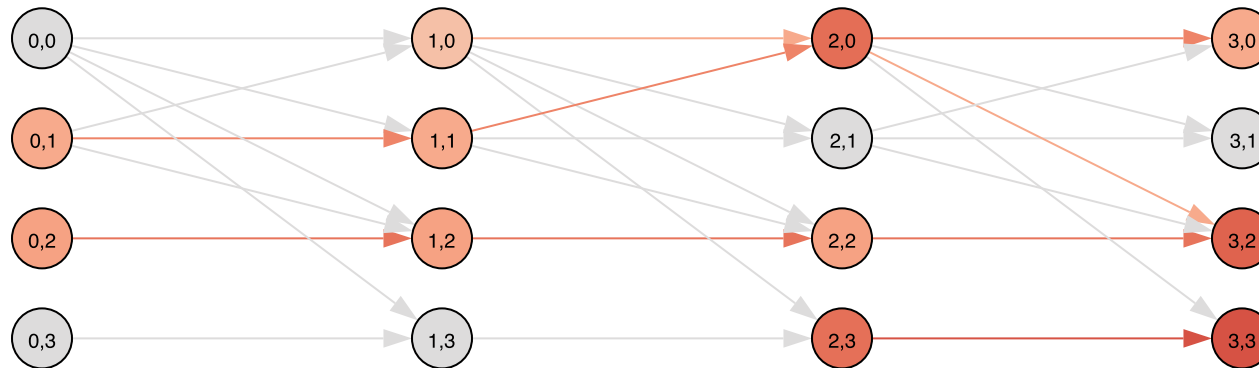
- TUG nodes/vertexes are labeled with *(time, node)* pairs
- TUG arcs are labeled with *(time, source, destination)* triplets

Computing Counts

We can inspect the arc counts visually on the TUG

```
In [11]: visual_style = util.get_visual_style(tug, vertex_weights=node_counts, edge_weights=arc_count)
         ig.plot(tug, **visual_style, bbox=(700, 250), margin=50)
```

Out[11]:



- A grey shade corresponds to lower counts
- A red shade corresponds to higher counts

These counts are our **available information**

Problem Formulation

By far the most important step of any solution process

Problem Formulation

Every good approach starts with a problem formulation

- If you don't have a formulation
- Odds are that you will come up with a patched-up solution

Let's try to come up with a formulation for our problem!

Problem Formulation

Every good approach starts with a problem formulation

- If you don't have a formulation
- Odds are that you will come up with a patched-up solution

Let's try to come up with a formulation for our problem!

We can introduce a variable x_j for each path

- The value of x_j represents the flow associated to the path
- Then we can compute the estimated count per TUG node/arc
- ...By simply summing the x_j values of paths that pass through the node/arc

Problem Formulation

Every good approach starts with a problem formulation

- If you don't have a formulation
- Odds are that you will come up with a patched-up solution

Let's try to come up with a formulation for our problem!

This approach is remarkably simple

- Computing counts is easy
- Connectivity constraints are satisfied by construction

Basically, we handle some constraints in the problem formulation itself

This is a first, powerful, and underestimated method to deal with constraints

Path Formulation

We will call this approach the **path formulation**

Formally, our problem can be stated as:

$$\operatorname{argmin}_x \left\{ \|Vx - v\|_2^2 + \|Ex - e\|_2^2 \mid x \geq 0 \right\}$$

- For simplicity, here we use **linear indexes** for TUG nodes and arcs
- V is a matrix such that $V_{ij} = 1$ iff path j passes through node i
- E is a matrix such that $E_{kj} = 1$ iff path j passes through arc k

Path variables cannot be negative (it would make no sense)

- Hence the path formulation is itself a **constrained optimization problem**
- ...Though the constraints are in this case very simple

Problem Reduction

For an squared L2 norm in the form $\|Ax - b\|_2^2$ we have that:

$$\begin{aligned}\|Ax - b\|_2^2 &= (Ax - b)^T (Ax - b) \\ &= x^T A^T Ax - x^T A^T b - b^T Ax + b^T b \\ &\propto \frac{1}{2} x^T (A^T A) x - \frac{1}{2} x^T A^T b - \frac{1}{2} b^T Ax \\ &= \frac{1}{2} x^T (A^T A) x + (-A^T b)^T x\end{aligned}$$

- This is true since $x^T A^T b$ and $b^T Ax$ are scalar
- ...And $y^T x = x^T y$ if the quantity is a scalar
- The scaling factor $1/2$ will become convenient later

This reduction is valid for any least squares problem

Problem Reduction

We can use the relation to reduce our problem to a more compact form

In particular, we have that:

$$\begin{aligned} & \|Vx - v\|_2^2 + \|Ex - e\|_2^2 \\ & \propto \frac{1}{2} \|Vx - v\|_2^2 + \frac{1}{2} \|Ex - e\|_2^2 \\ & = \frac{1}{2} x^T (V^T V) x + (-V^T v)^T x + \frac{1}{2} x^T (E^T E) x + (-E^T e)^T x \\ & = \frac{1}{2} x^T P x + q^T x \end{aligned}$$

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

Path Formulation as Convex Quadratic Programming

Therefore, the path formulation can be reduced to:

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

...Which is a **quadratic program**

- I.e. a problem where we want to minimize a **quadratic form**
- ...Subject to **linear constraints**

Our problem is also **convex**

- This is true since $P = V^T V + E^T E$
- ...And it is therefore guaranteed semi-definite positive

Convex quadratic programs can be solved **in polynomial time**