

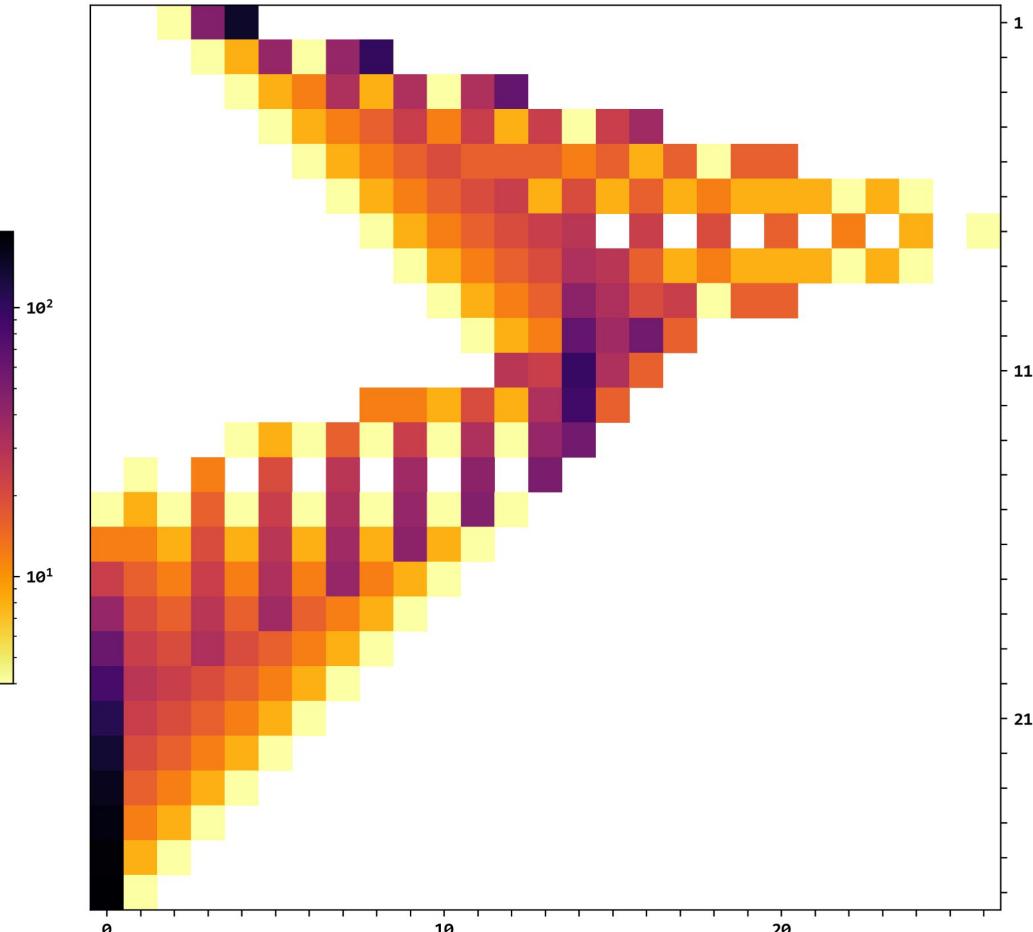
What is a B-Matrix?

Also known as a **network portrait**, it is a graph invariant data abstraction about the structural properties of a graph.

It is used in portrait divergence as a metric of similarity between networks.

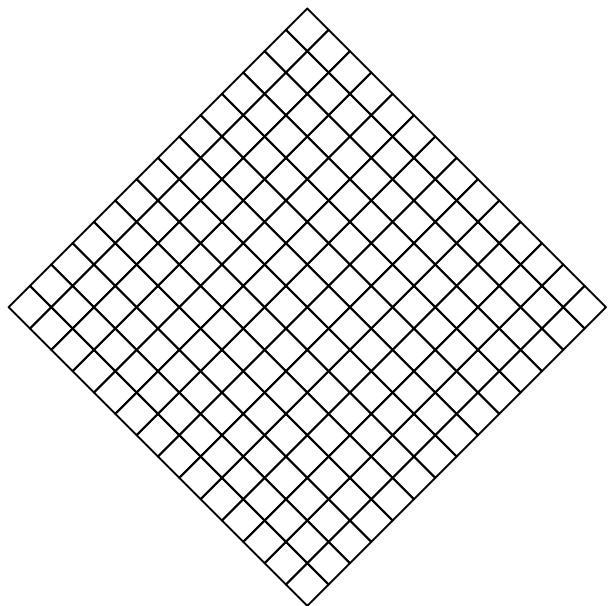
Bagrow, J. P., Boltt, E. M., Skufca, J. D., & Ben-Avraham, D. (2008). Portraits of complex networks. *EPL (Europhysics Letters)*.

Bagrow, J. P., & Boltt, E. M. (2019). An information-theoretic, all-scales approach to comparing networks. *Applied Network Science*.

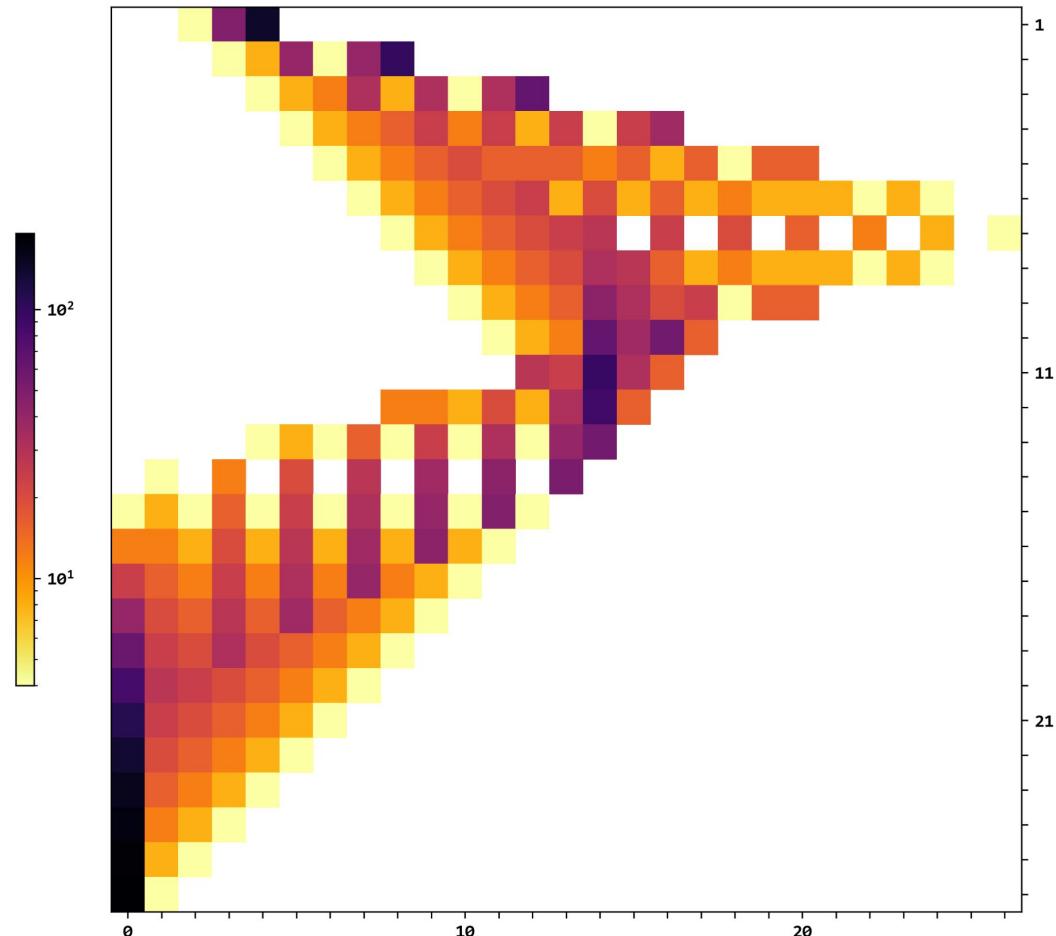


Graph | Square Graph (14x14)

Nodes 196
Edges 364

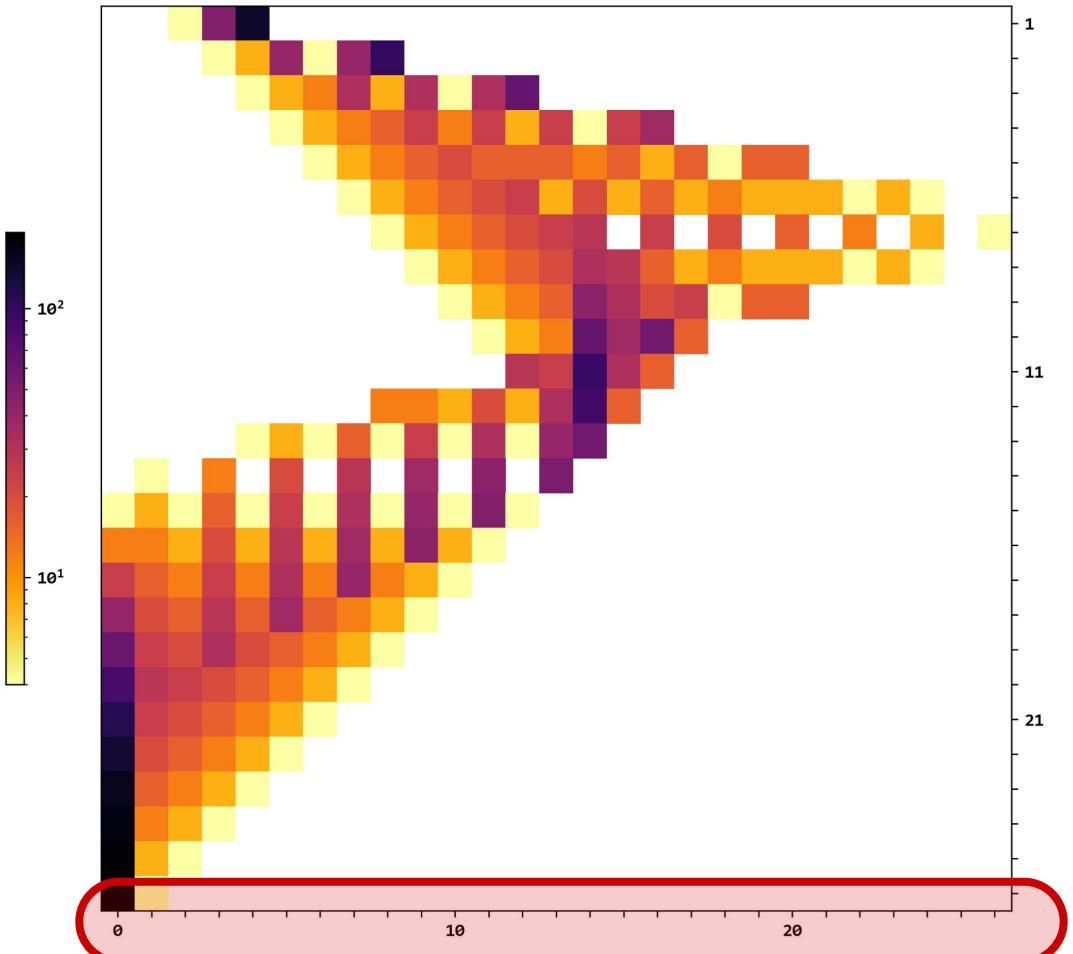


This grid network creates the B-Matrix portrait on the right.



The horizontal axis is
for **number of nodes**
reached by the
algorithm at a
given step.

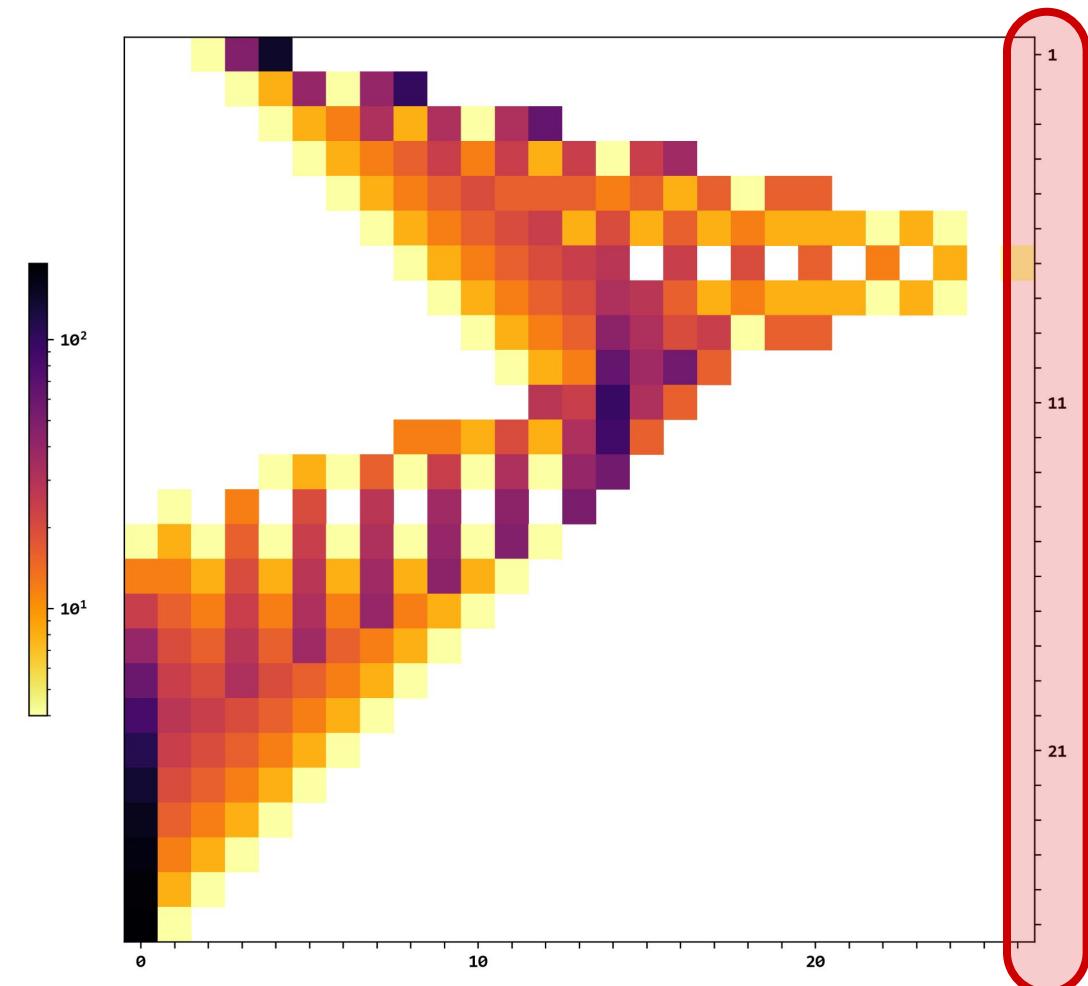
Has index-zero and
starts counting at
zeroth column, with
+1 integer steps.



The vertical axis is
the **number of hops**
the algorithm run
through.

Has index-zero and
starts counting at
zeroth row, with +1
integer steps.

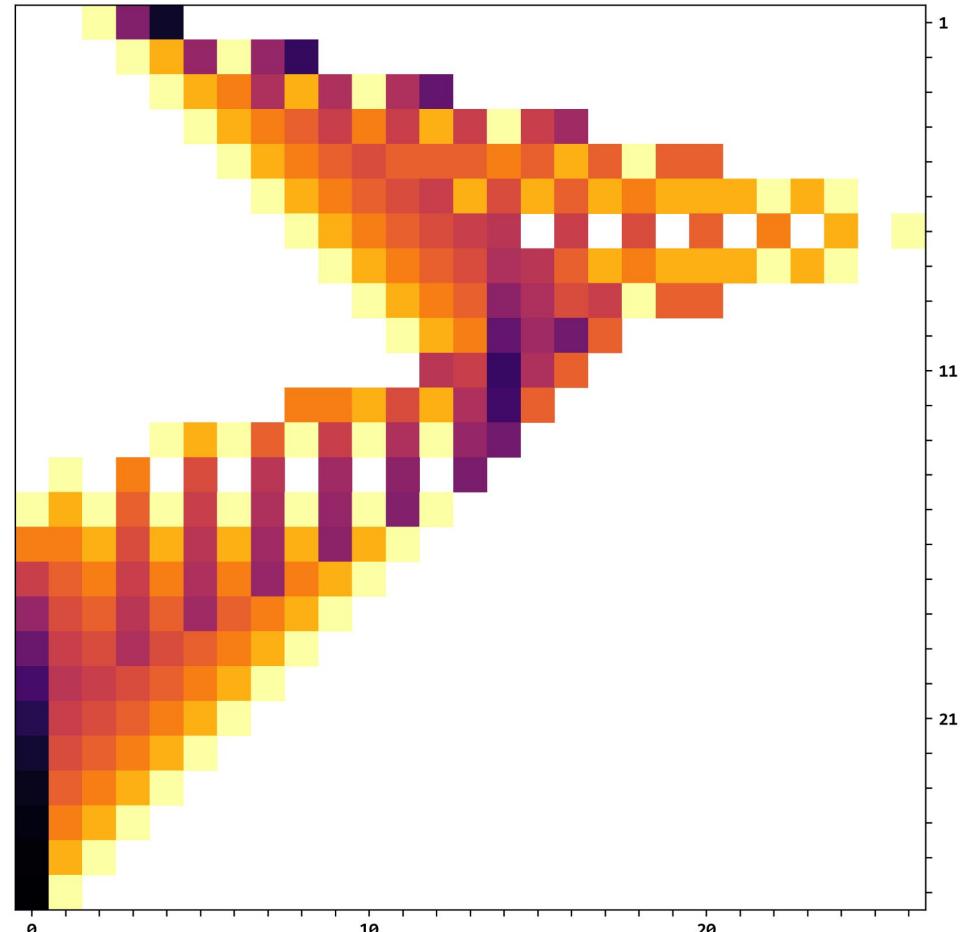
The zeroth row can be
omitted in visualizations.
The reason why is
covered later in
this explainer.

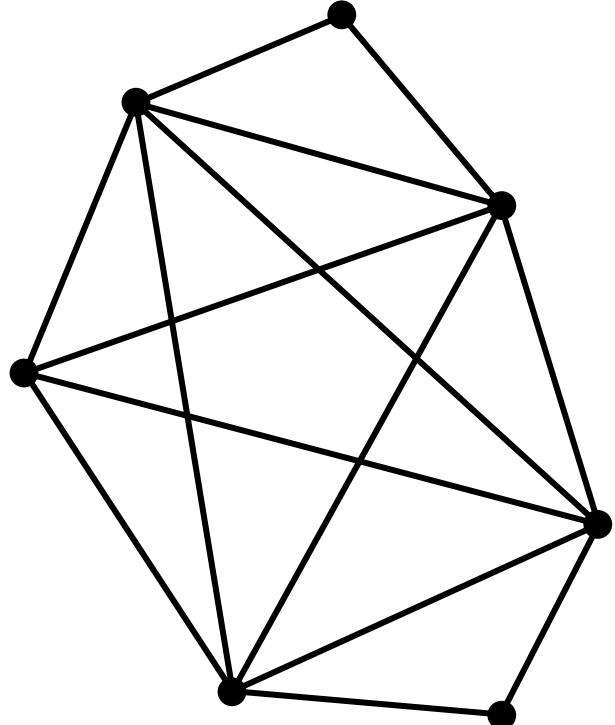


In this color-based
B-Matrix portrait the
**colormap encodes total
node count per cell.**

Starts counting at 1,
with +1 integer steps,
from a default of 0.
At the end of the
algorithm, cells at 0
become to NaN.

Because the difference
between the smallest
and largest values in
this matrix is larger
than 100, this
colormap is
log-transformed.





Consider this small graph as an explainer example of how `algorithm_BMatrix.py` works.

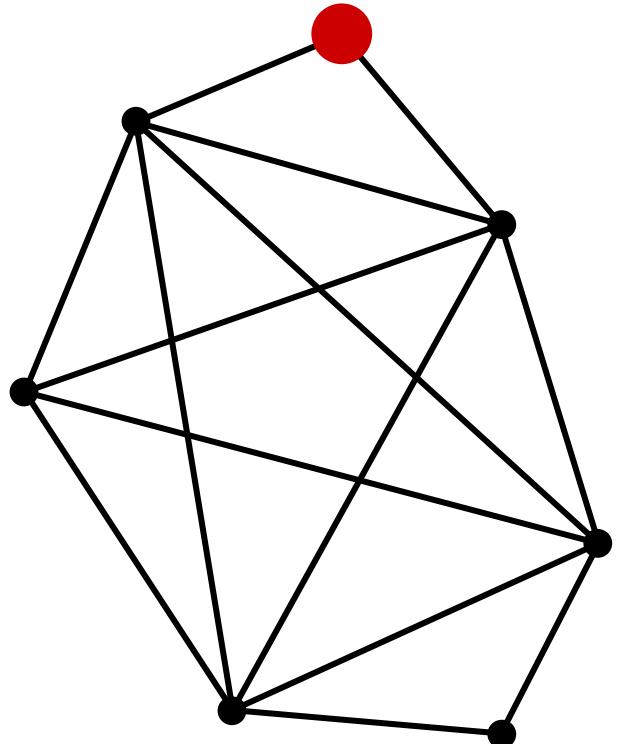
- It has 7 nodes and 14 edges

- Edgelist:

- 1-2
- 1-3
- 2-4
- 2-5
- 2-6
- 2-3
- 3-4
- 3-5
- 3-6
- 4-5
- 4-6
- 5-7
- 5-6
- 6-7

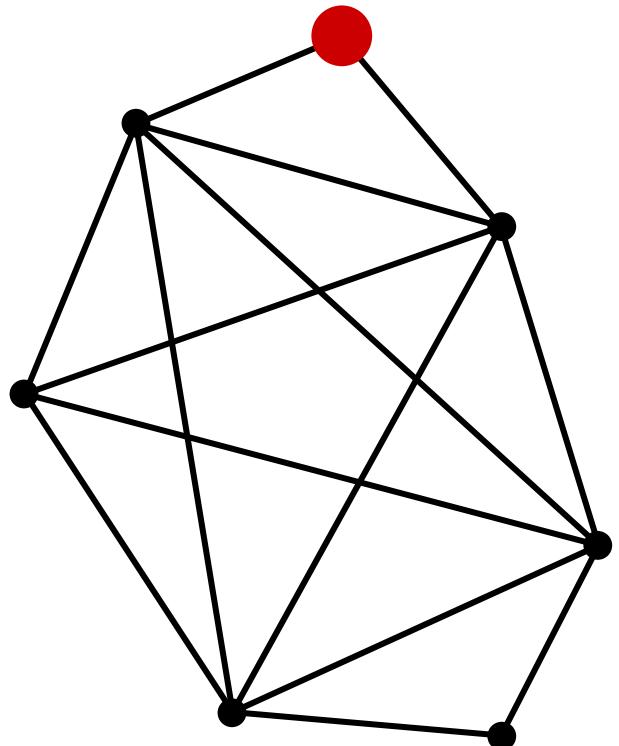
- Snippet:

```
G = nx.graph_atlas(1115)
B_matrix = algorithm_BMatrix(G)
```



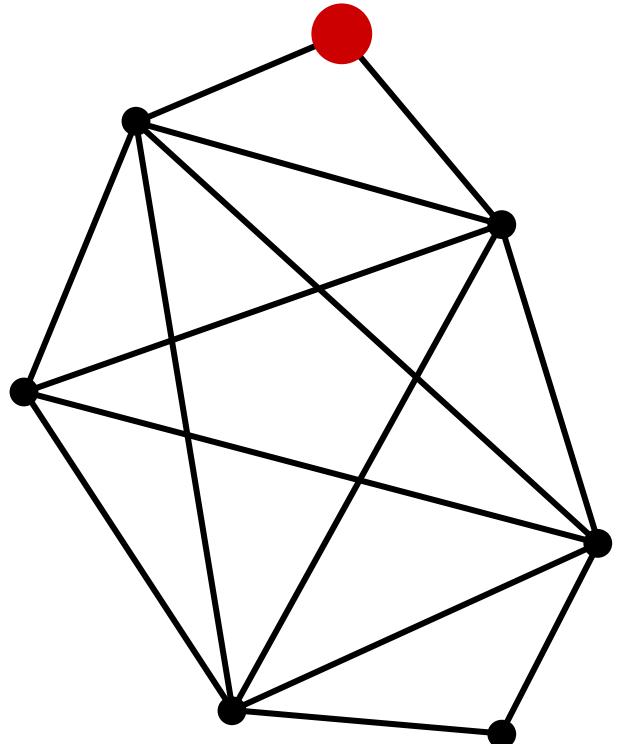
We start by picking a node, and initialize an empty matrix.

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0



Rows are for number of hops away from this starting node, and columns are for node counts.

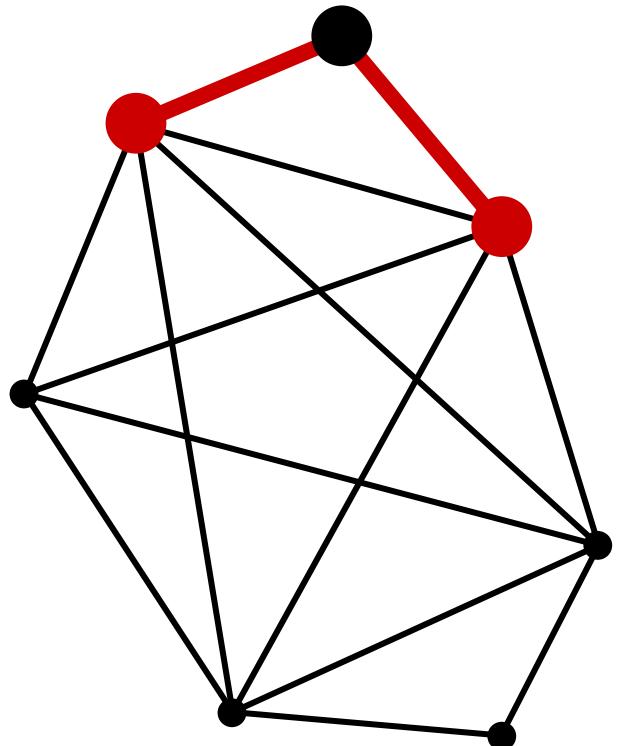
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0



We only have one node at hand (no hops yet), so we **flip the bit** at zeroth row and first column.

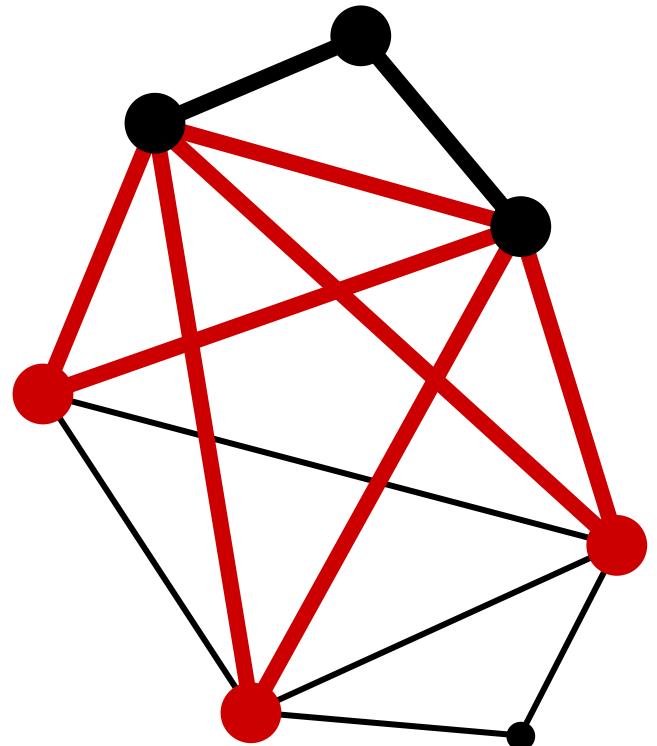


0	1	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0



Then at first hop, there are two nodes - so we flip the bit at the first row and second column.

0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

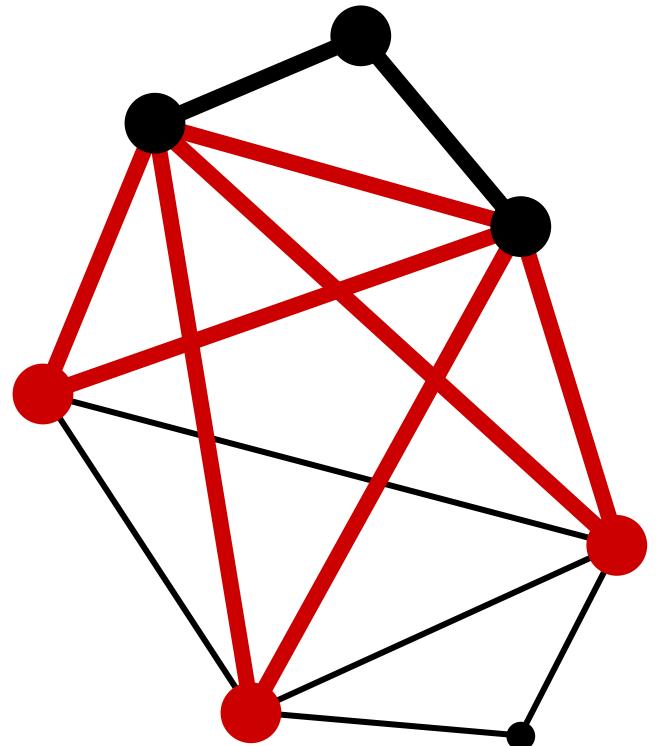


At second hop there are three nodes, so we flip the bit at the second row and third column.

A 4x6 grid representing a matrix or adjacency list. The grid has four rows and six columns. The values in the grid are as follows:

0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	0	0

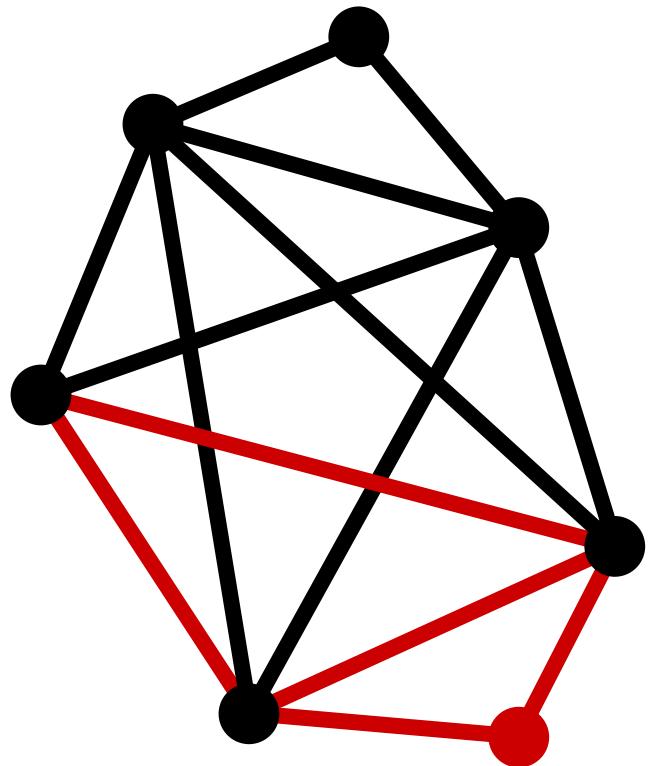
A red arrow points to the cell containing the value '1' in the second row and third column, indicating the specific bit being manipulated.



(in literature these hops are known as an *L*-shell, where *L* is the hop number).

0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	0	0

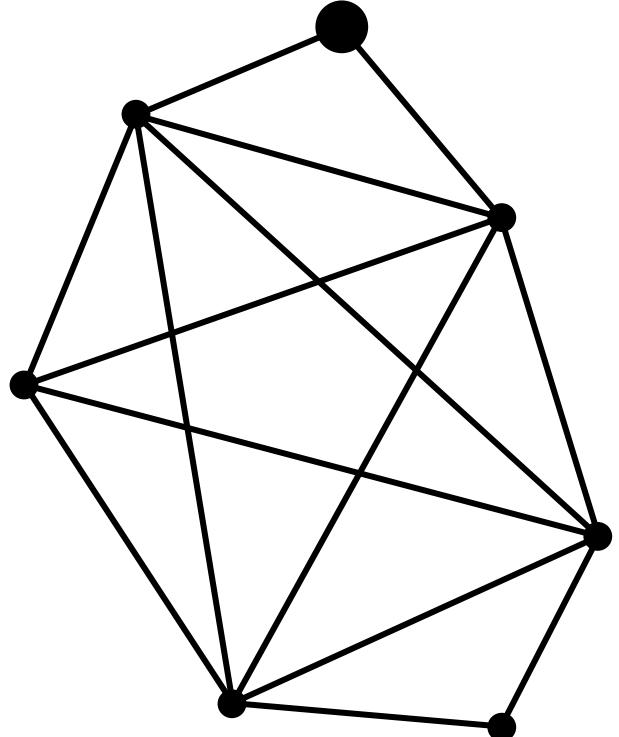
Bagrow, J. P., Boltt, E. M., Skufca, J. D., & Ben-Avraham, D. (2008). Portraits of complex networks. *EPL (Europhysics Letters)*, 81(6), 68004.



At third hop there is one node (last one), so we flip the bit at the third row and first column.

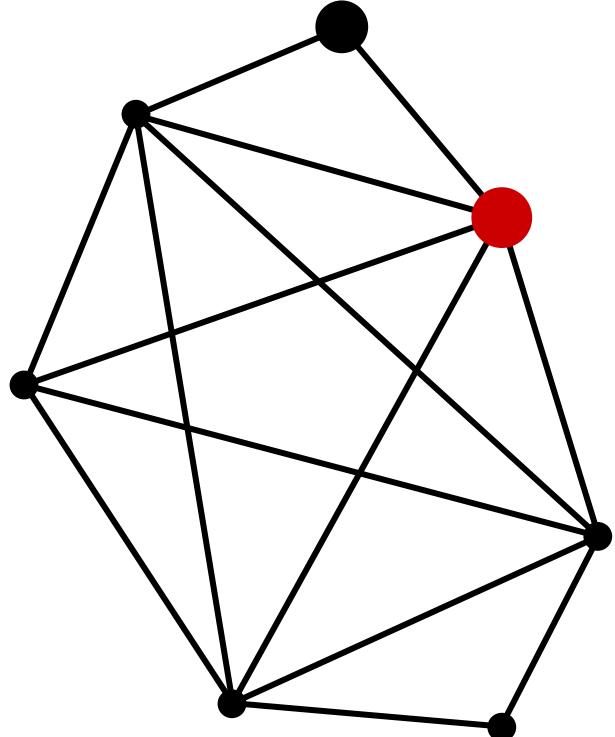
A 4x6 grid representing a matrix or adjacency list. The grid contains binary values (0 or 1). A red arrow points to the bottom-left cell (row 4, column 1), which contains a 1. The other cells in the grid contain 0s. The grid is as follows:

0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	1	0	0	0	0



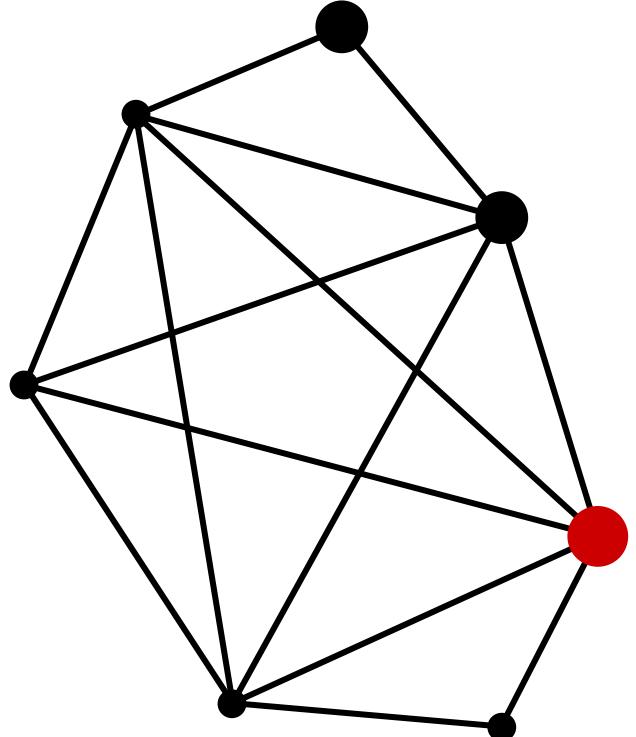
This completes the bit matrix of node 1 (of 7).

0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	1	0	0	0	0



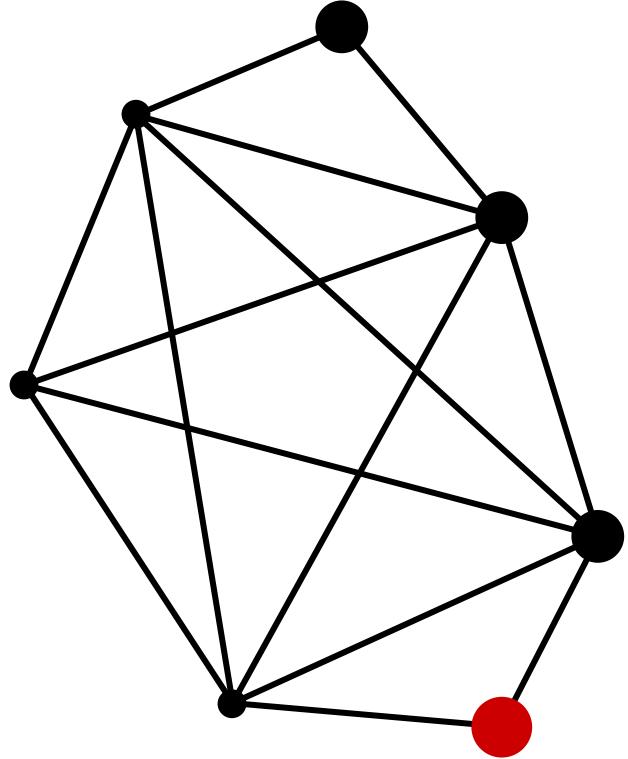
We repeat and get a bit matrix for node 2 (of 7), and we store the already completed matrix.

0	1	0	0	0	0
0	0	0	0	0	1
0	1	0	0	0	0
1	0	0	0	0	0



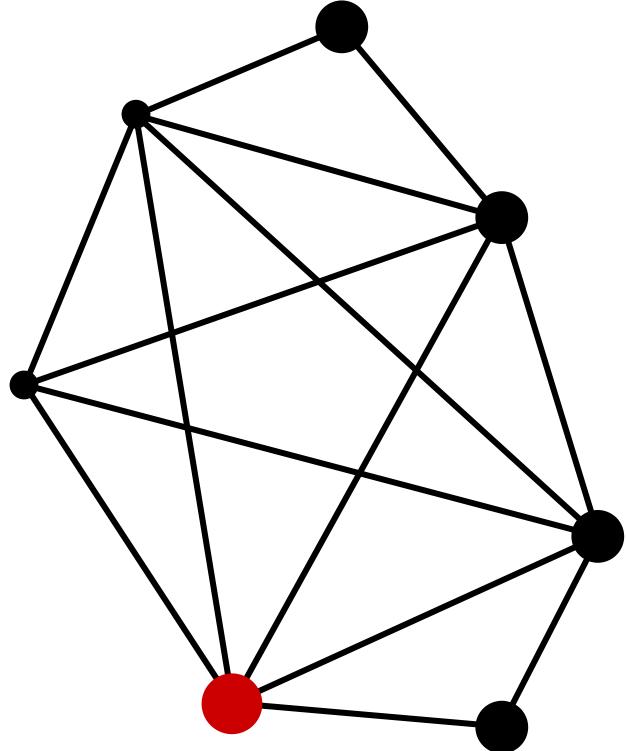
This is the bit matrix of node 3 (of 7), and we save the two already completed matrices.

0	1	0	0	0	0
0	0	0	0	0	1
0	1	0	0	0	0
1	0	0	0	0	0



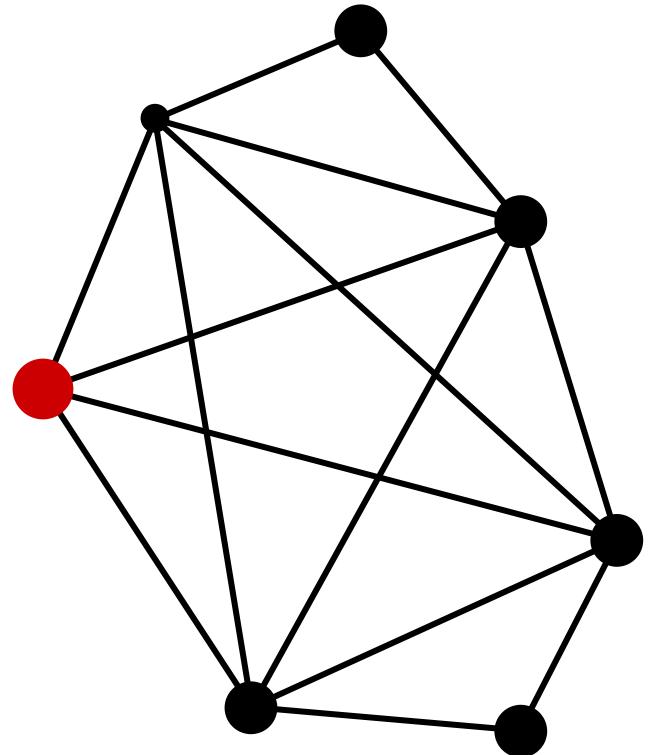
The bit matrix of node 4 (of 7), and we save the three already completed matrices.

0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	1	0	0	0	0



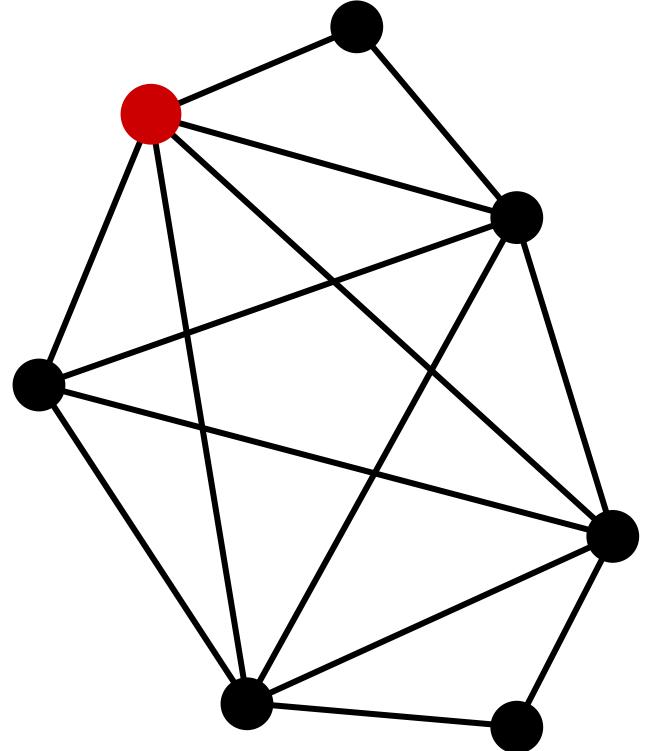
The bit matrix of node 5 (of 7), and we save the four already completed matrices.

0	1	0	0	0	0
0	0	0	0	0	1
0	1	0	0	0	0
1	0	0	0	0	0



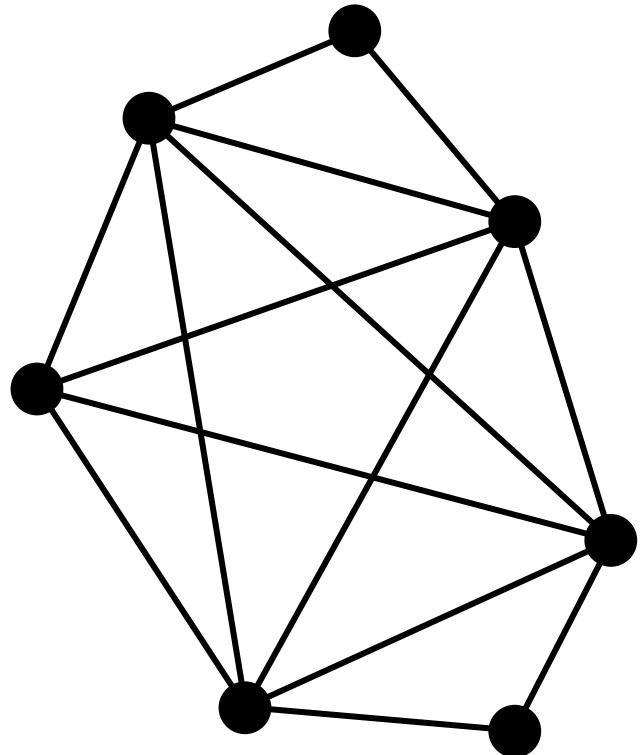
The bit matrix of node 6 (of 7), and we save the five already completed matrices.

0	1	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0
1	0	0	0	0	0



Finally, the bit matrix of node 7 (of 7), and we save the six already completed matrices.

0	1	0	0	0	0
0	0	0	0	0	1
0	1	0	0	0	0
1	0	0	0	0	0



We add these seven bit matrices element-wise into a single matrix. This ends the algorithm.

0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0

We're done!

The B-Matrix of the graph.

And it has a bunch
of properties.

0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0

For example, the zeroth row always has **total node count** in the first column position.

7 ↙

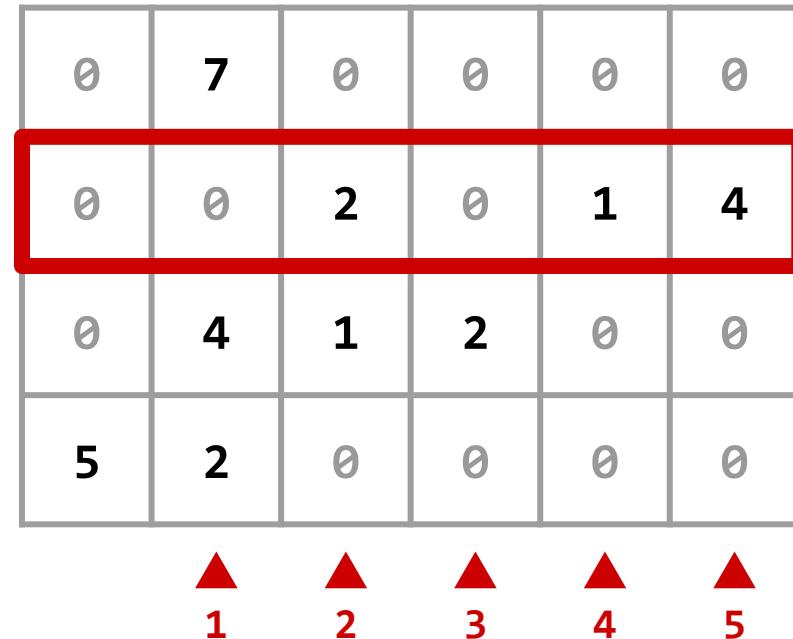
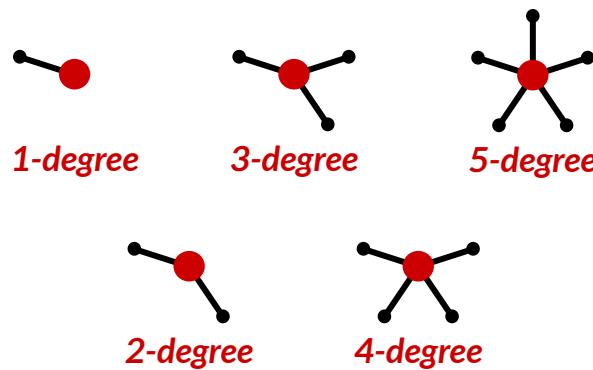
0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0

Also each row adds up to the total number of nodes.

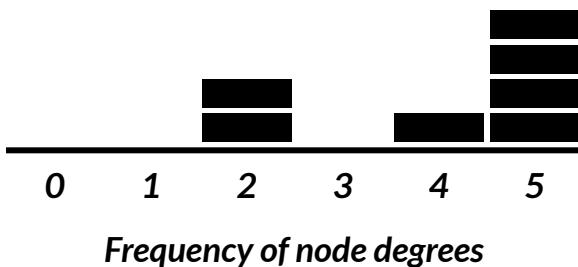
7 ↙	0	7	0	0	0	0
7 ↙	0	0	2	0	1	4
7 ↙	0	4	1	2	0	0
7 ↙	5	2	0	0	0	0

The first row marks the number of times different node counts happened at exactly one hop away from the starting node.

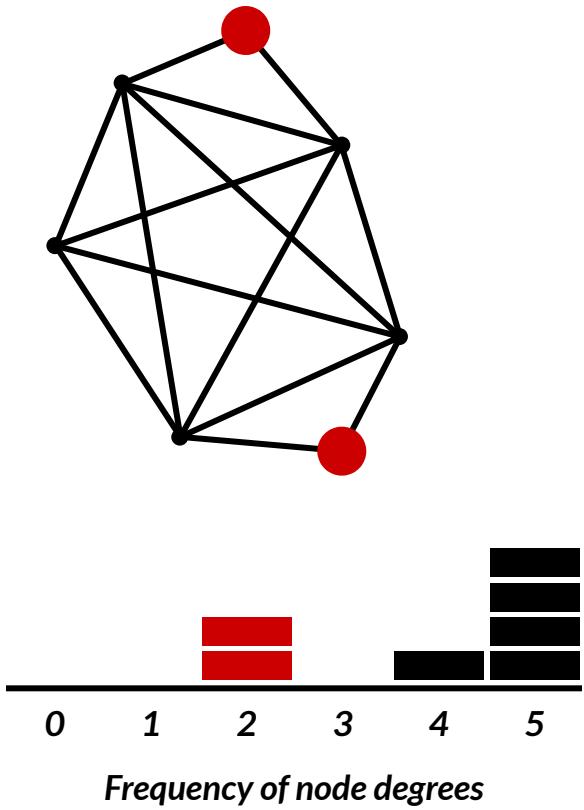
The **node degree** is how many edges a node has, so this row is effectively a record of frequency of node degrees.



We can visualize the distribution of node degrees with a **bar chart of counts**, or *histogram*.

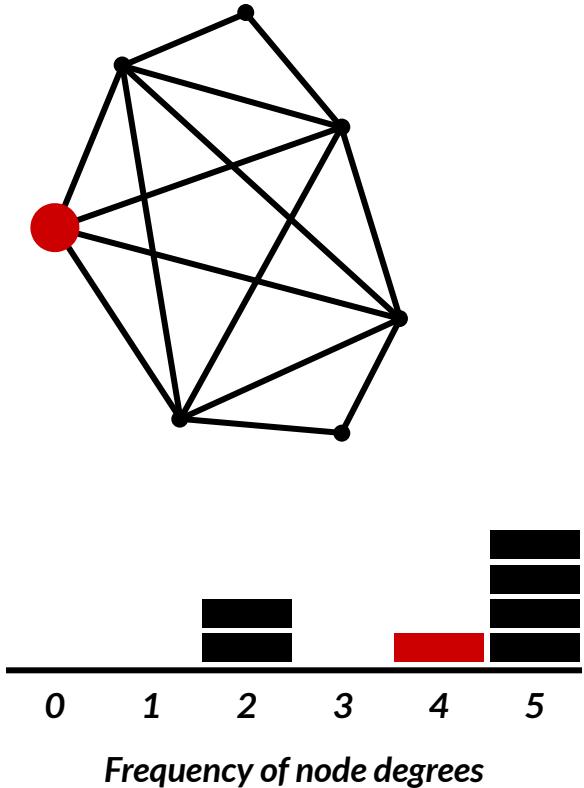


0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0



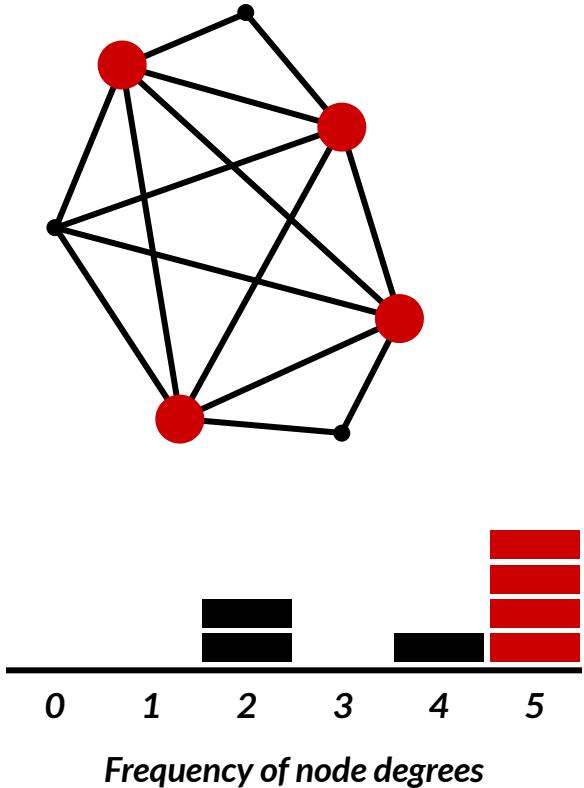
[Row 1, Column 2] Two nodes of degree 2

0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0



[Row 1, Column 4] One node of degree 4

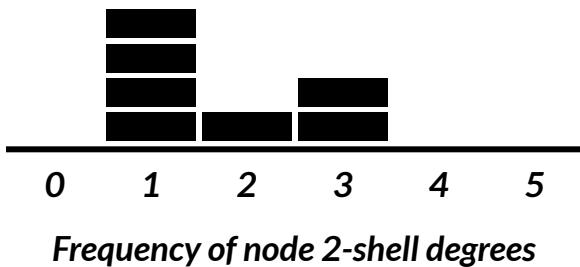
0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0



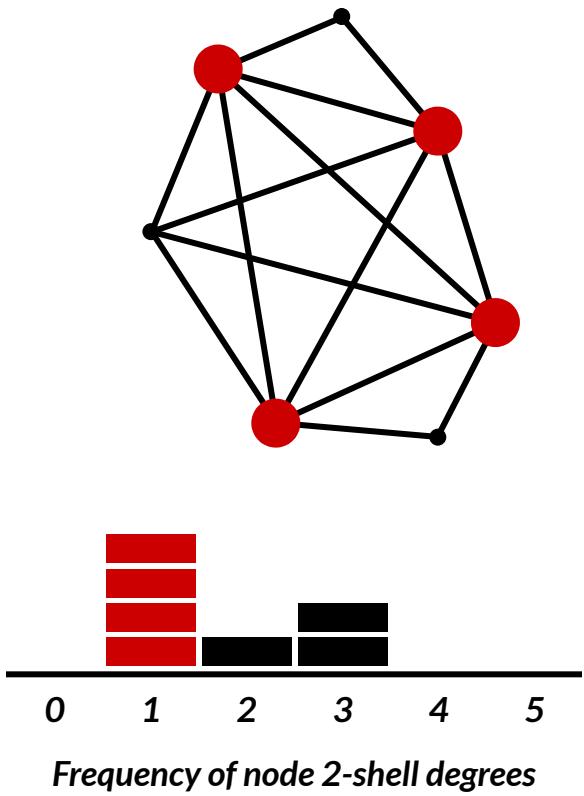
[Row 1, Column 5] Four nodes of degree 5

0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0

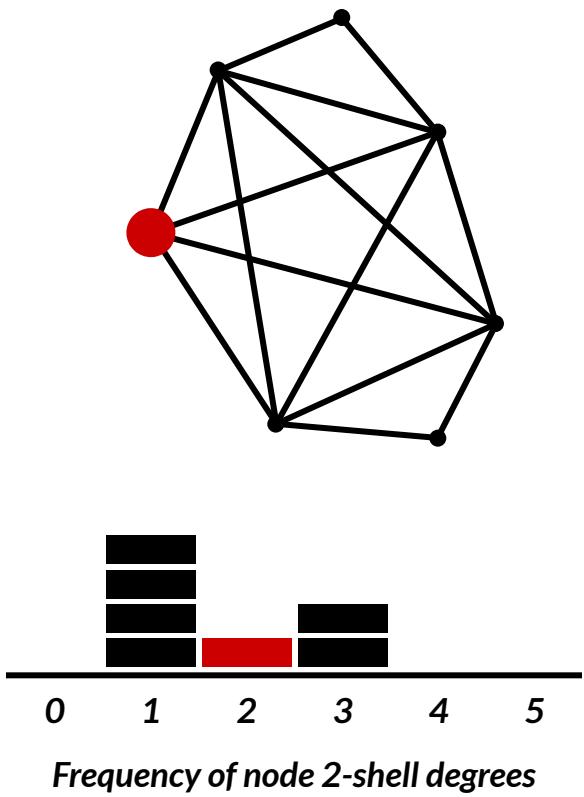
The next row is the distribution of nodes with *2-shell* degrees.



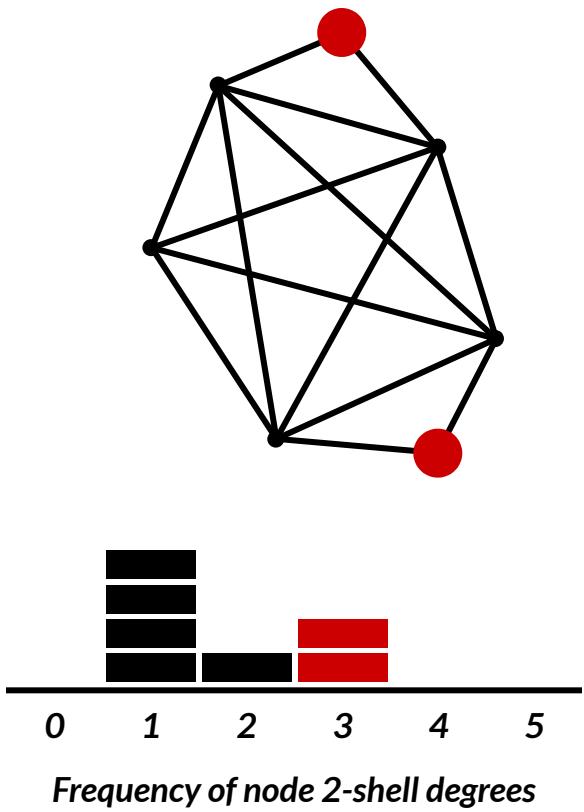
0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0



0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0



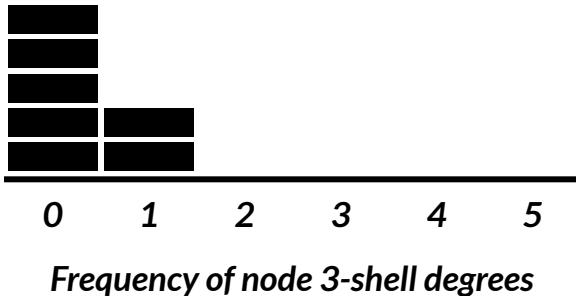
0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0



0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0

Finally, the last row is the maximum number of hops the algorithm got through before running out of nodes.

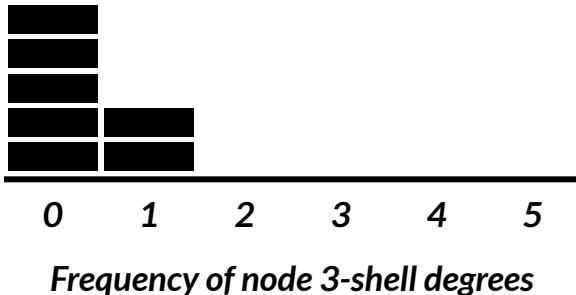
In other words, the final hop number is equivalent to the **diameter** of the graph, *L-shell* of 3 in this case.



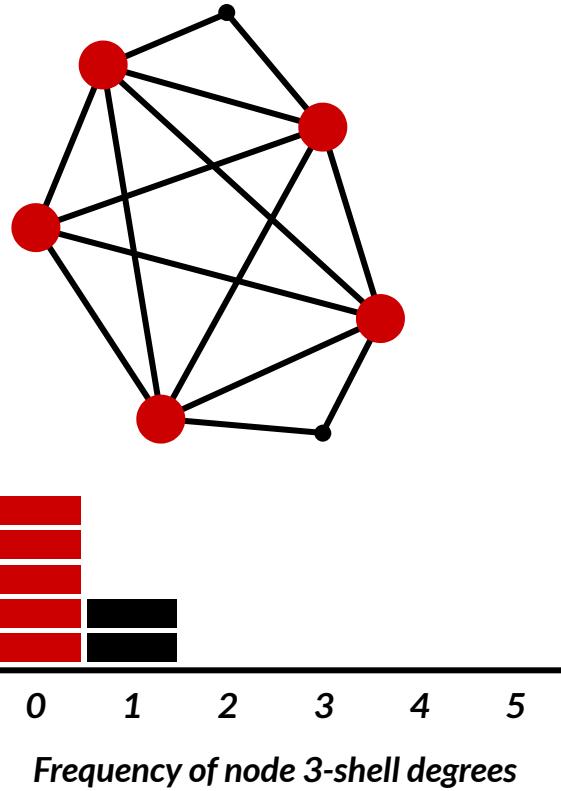
0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0

This means that the height of the B-Matrix depends on the graph diameter.

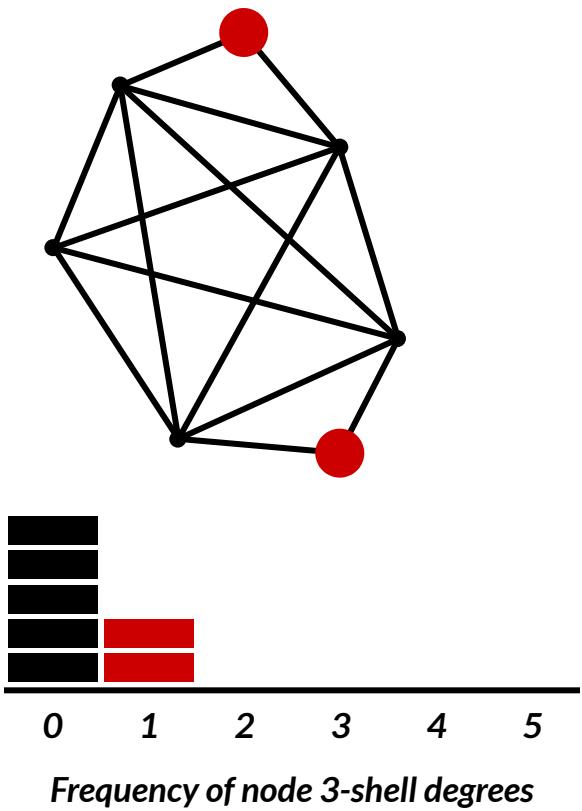
In this example, by the **third hop** there are only five instances where three hops completely covered all nodes, with two leftover nodes at the ends of the diameter.



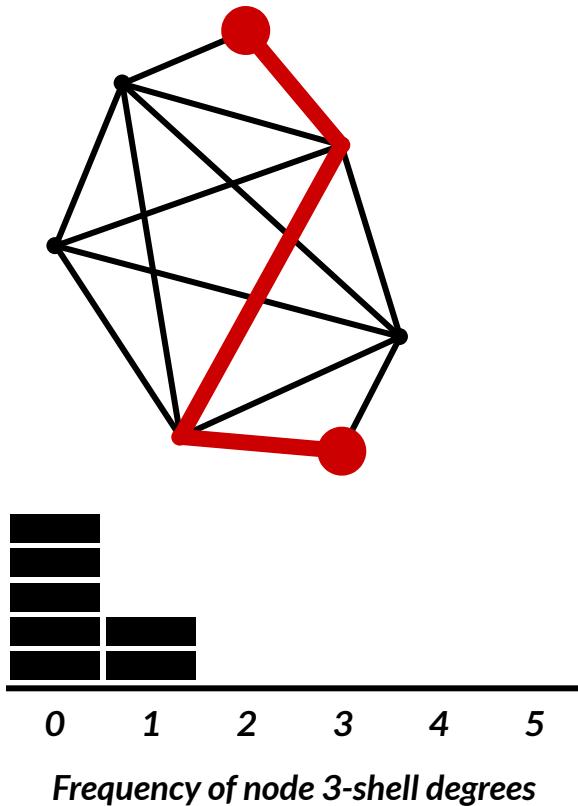
0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0



0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0



0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0



The **graph diameter** is the length of the shortest path between the two most distanced nodes.

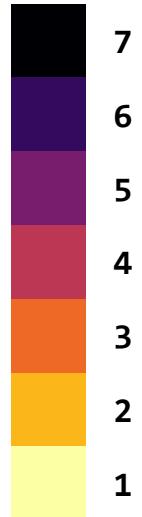
0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0

In contrast to this explainer example, the B-Matrix of a real-world graph can be very large.

It is not practical to show this data abstraction directly with numbers, we need a visual encoding.

0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0

In literature
this is solved
by mapping the
B-Matrix node
count to a
colormap
range.



0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0

And the result
is a **heatmap**.



However, we can do one more edit to increase information resolution in this heatmap idiom.

Notice that the zeroth row **always has the highest value** (which sets the colormap extreme).

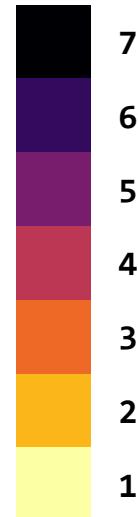
In large networks this value can be so high that the colormap must be log-transformed.



0	7	0	0	0	0
0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0

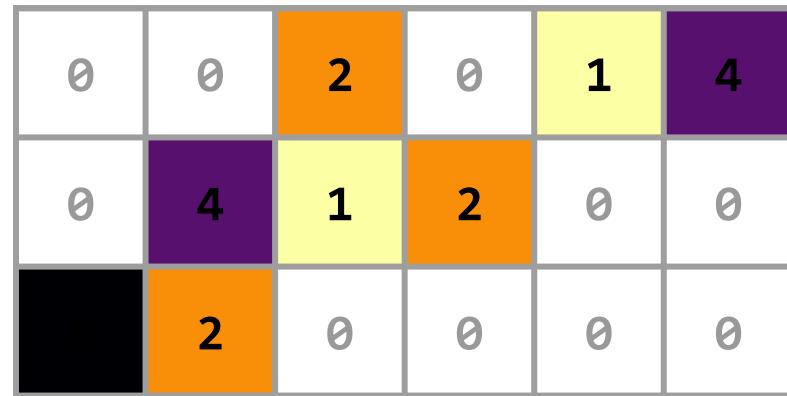
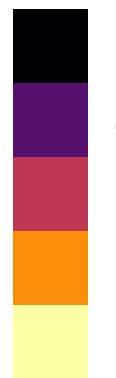
We can safely
remove the
zeroth row.

This is fine because
this row only contains
redundant data
(total node count).

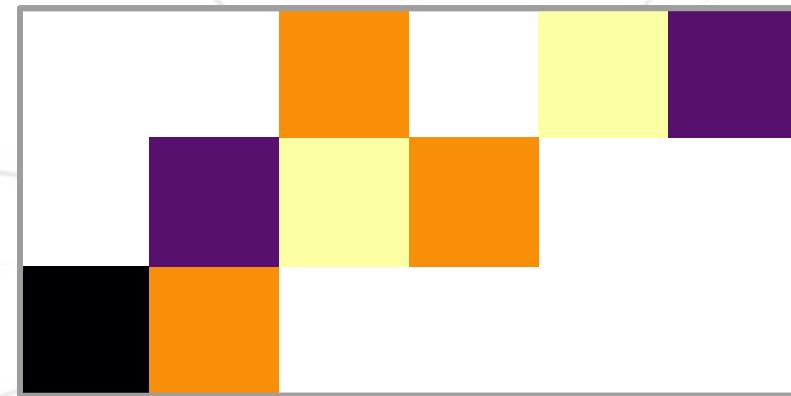


0	0	2	0	1	4
0	4	1	2	0	0
5	2	0	0	0	0

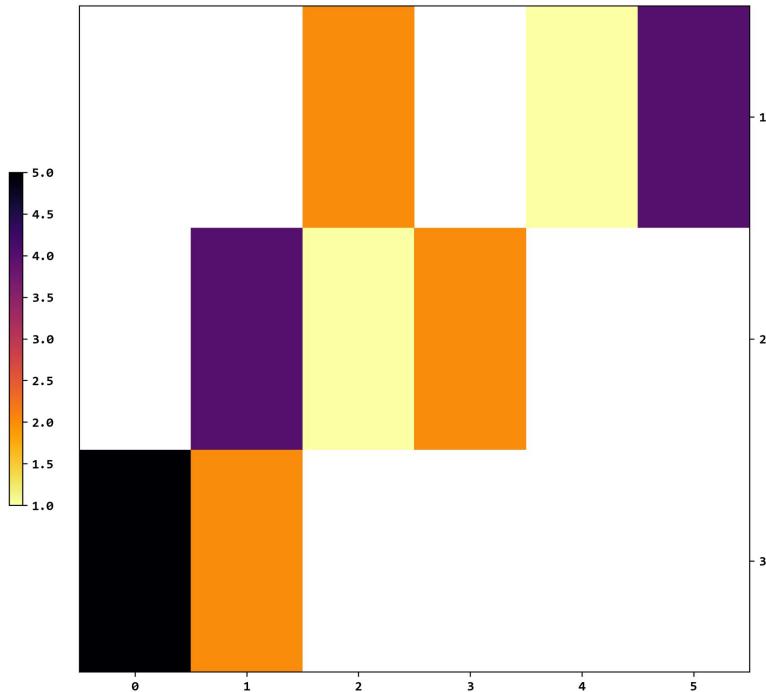
Then we can get higher fidelity by rescaling the colormap.



That's it!
This figure is the
network portrait
of the graph.

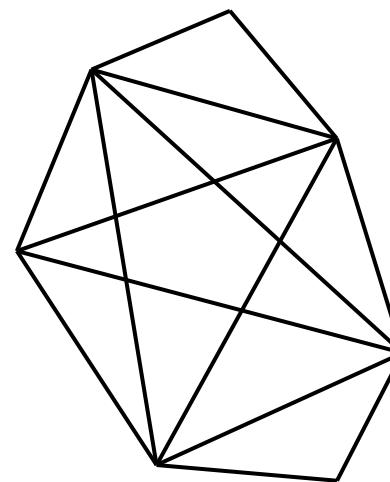


The script `plot_BMatrix_colormap.py` generates a network portrait from a B-Matrix (example data and figure below).



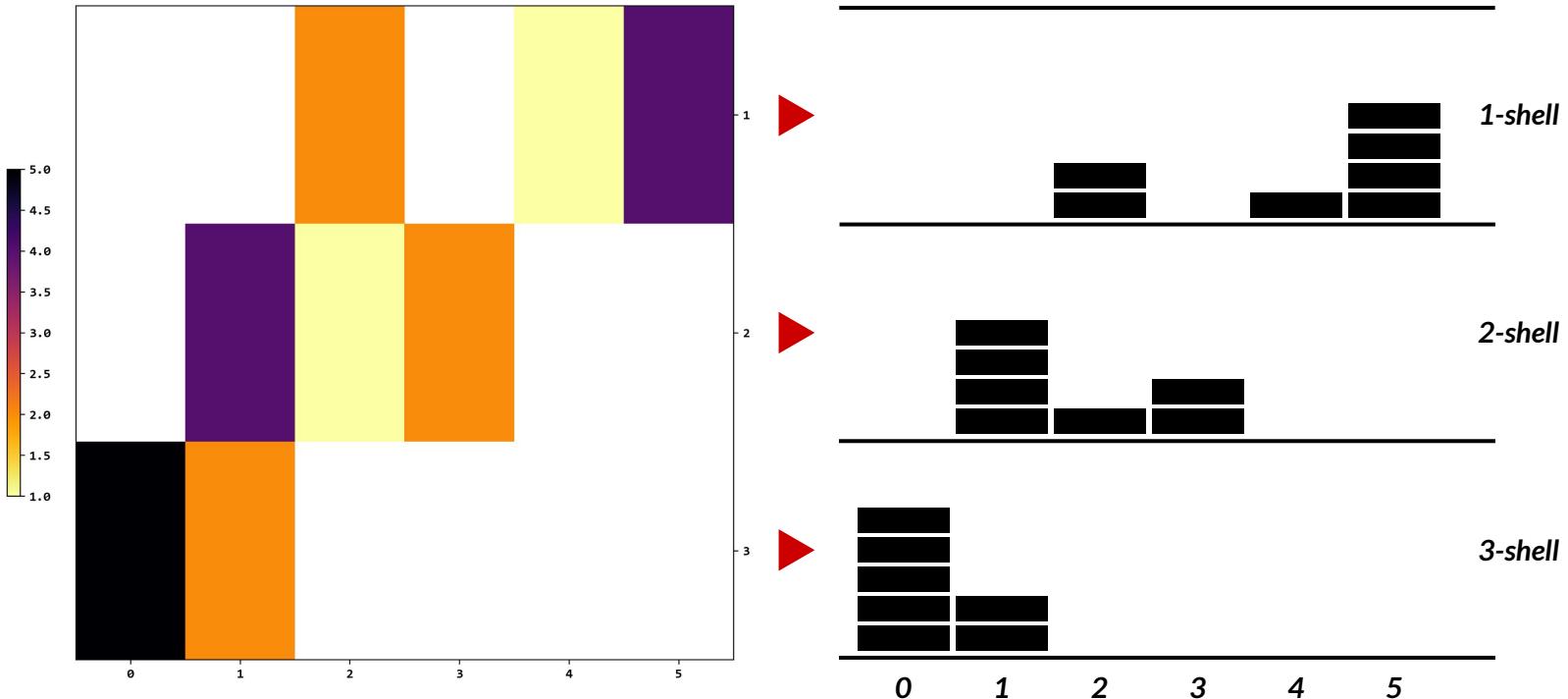
Graph | `networkx.graph_atlas(1115)`

Nodes	7
Edges	14

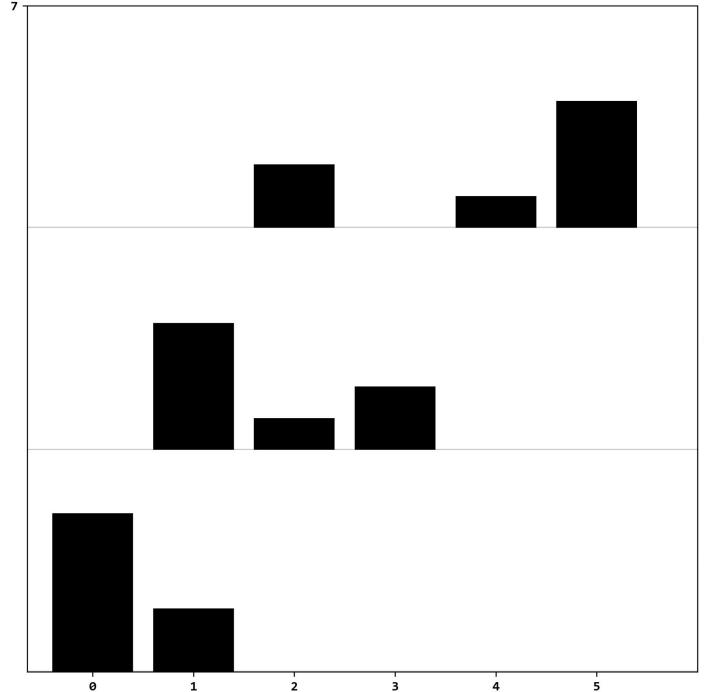
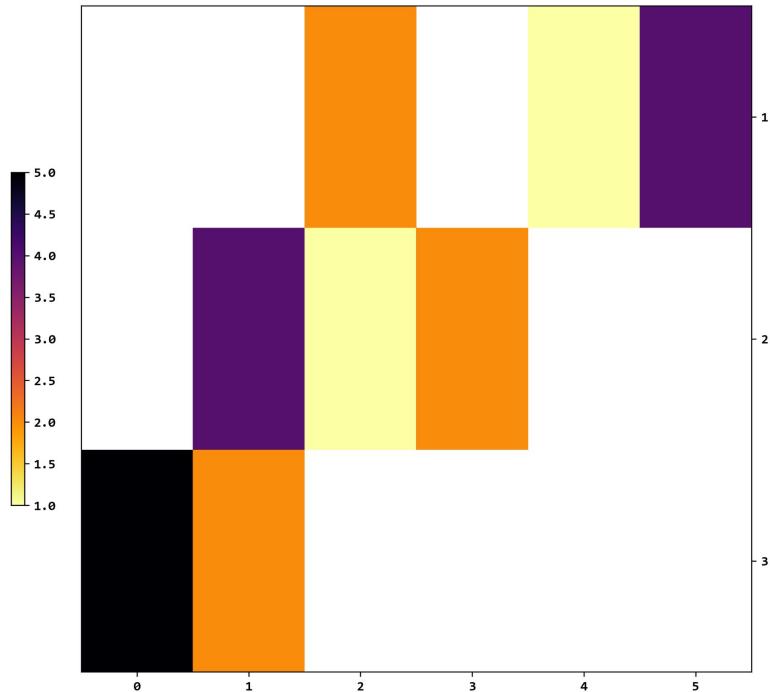


1-2
1-3
2-4
2-5
2-6
2-3
3-4
3-5
3-6
4-5
4-6
5-7
5-6
6-7

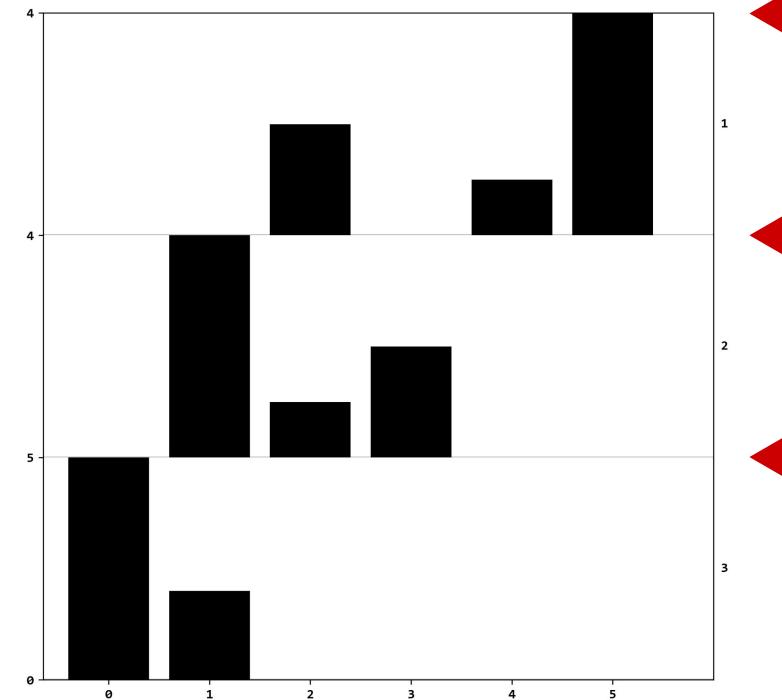
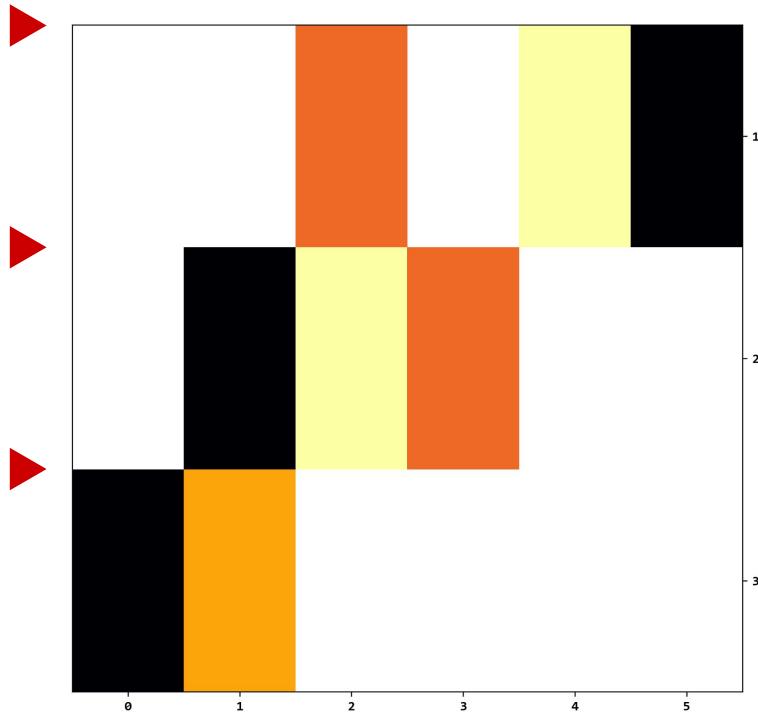
Recall that each row of the B-Matrix can be visually encoded as stacked bars for count data.



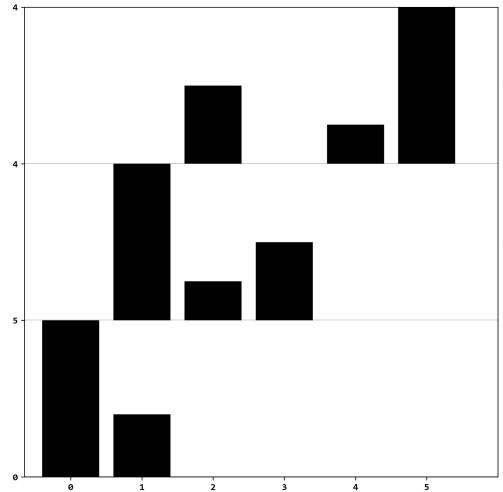
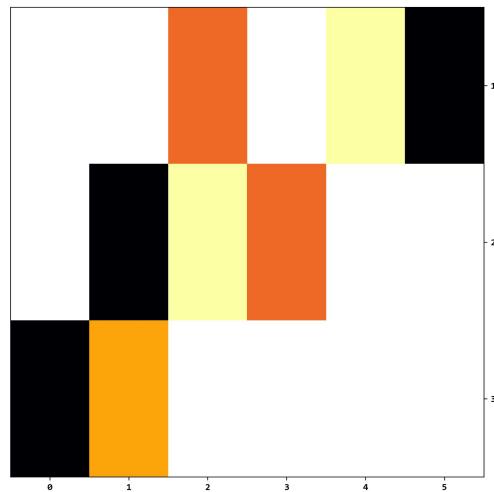
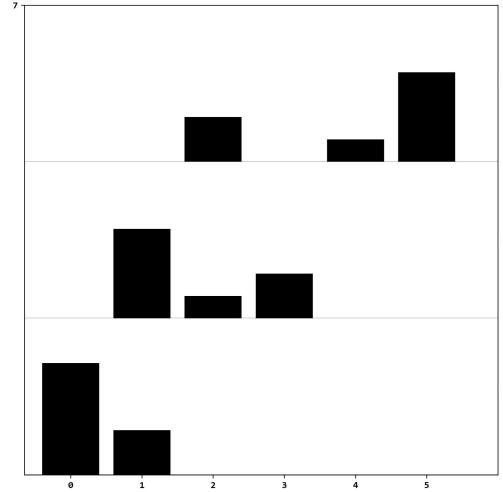
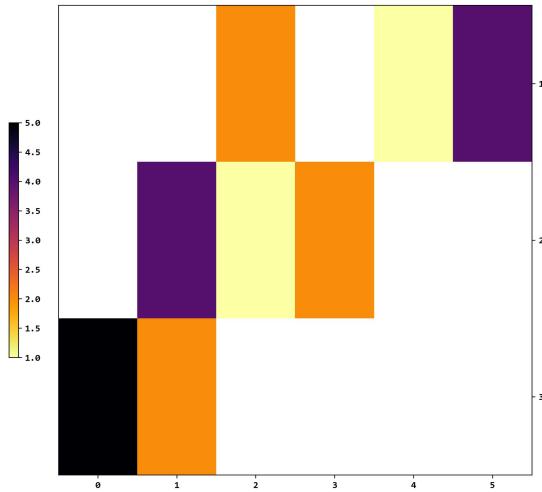
The script `plot_BMatrix_histogram.py` creates this visualization with bars instead of colors for encoding node counts.



To further support B-Matrix interpretation tasks, both visualization scripts have a **row_normalized=True** option to enhance information discovery within hop level.

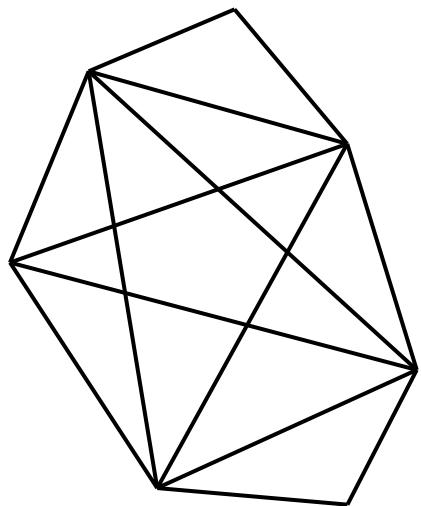


Both visualization scripts render and save these figures at **300dpi** and in **PNG format** with the non-interactive back-end `matplotlib.use("Agg")`.

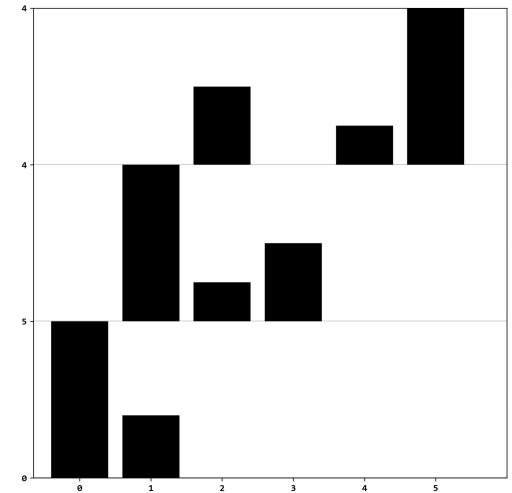
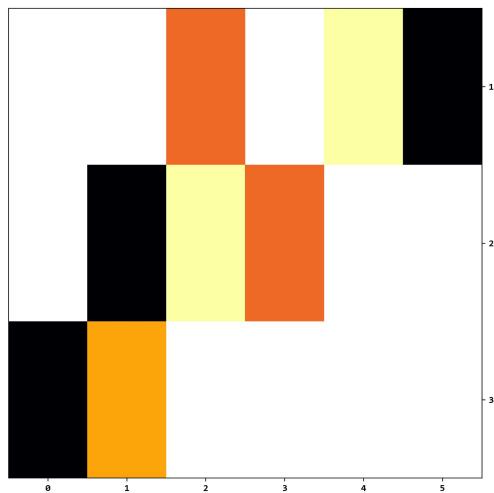
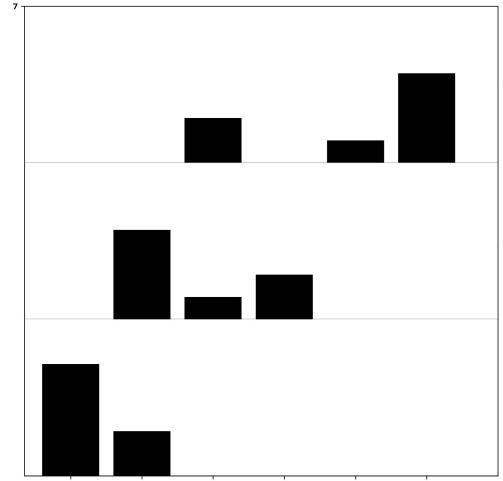
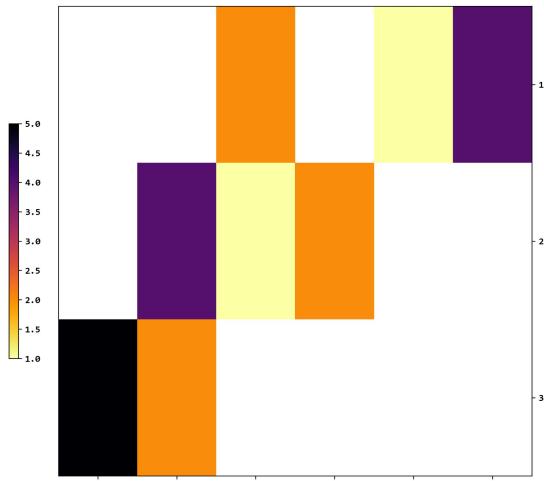


Graph | nx.graph_atlas(1115)

Nodes | 7
Edges | 14

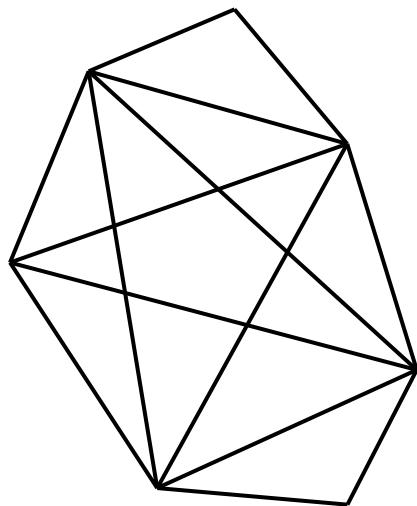


Example explainer
graph and its quartet
of B-Matrix visualizations.

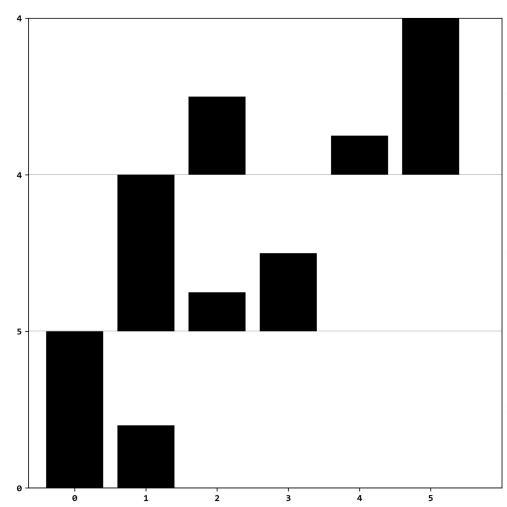
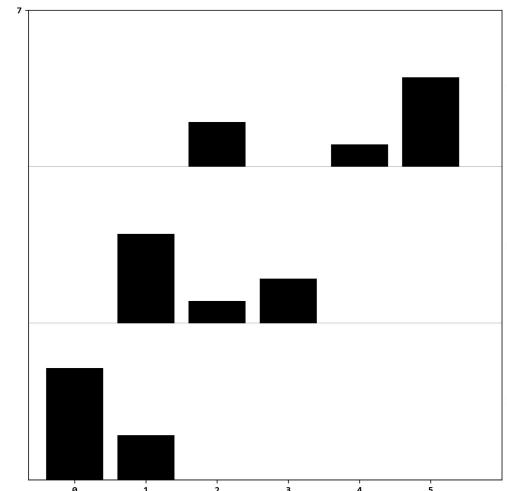
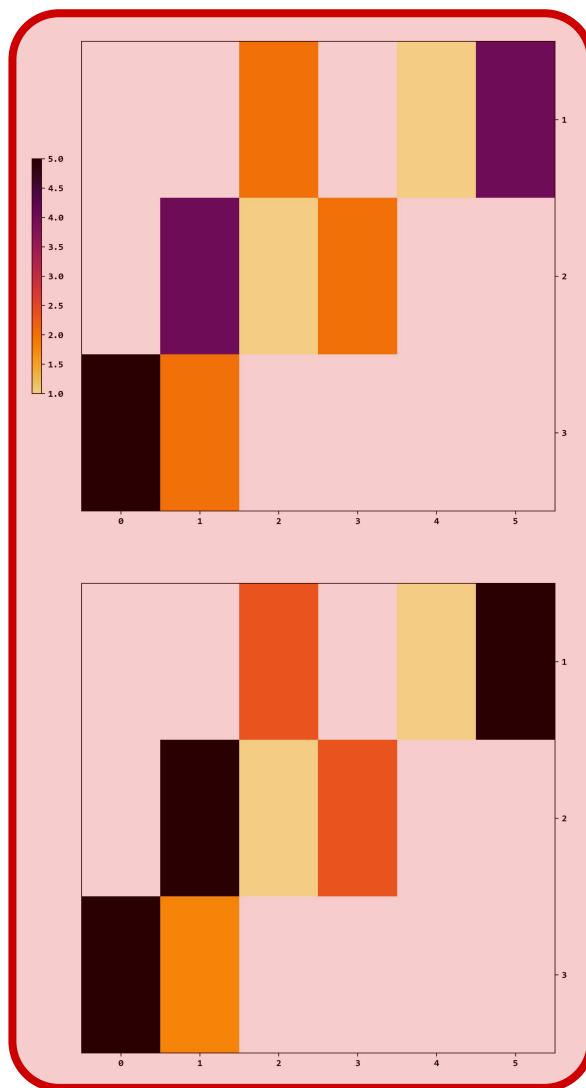


Graph | nx.graph_atlas(1115)

Nodes | 7
Edges | 14



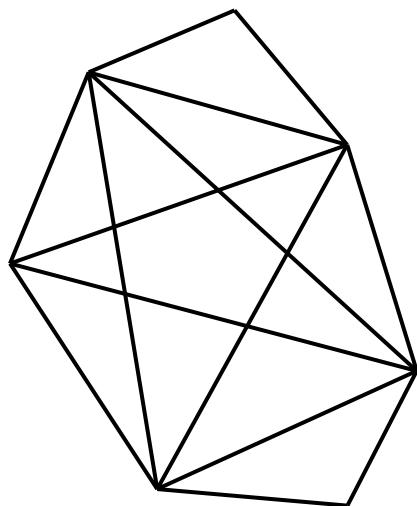
On this left side
node counts are encoded
with through a **colormap**.



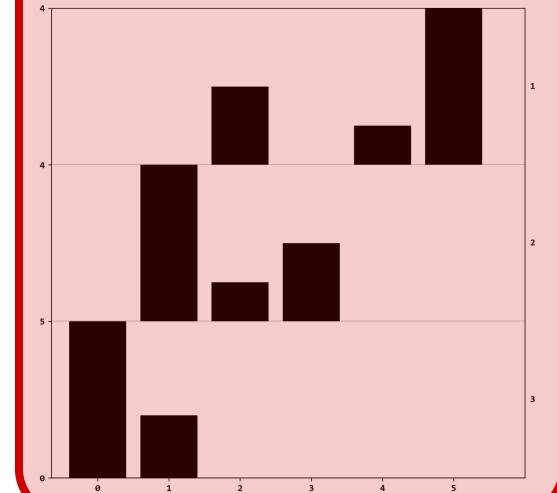
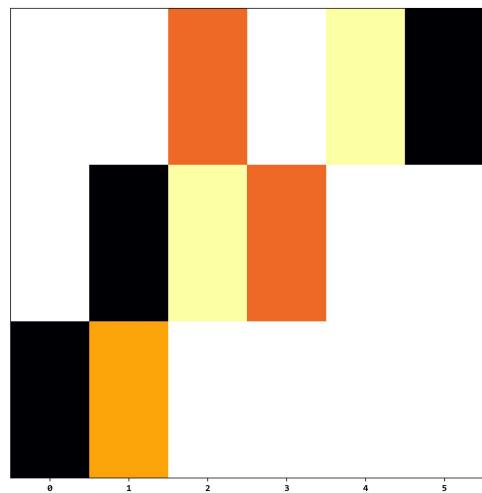
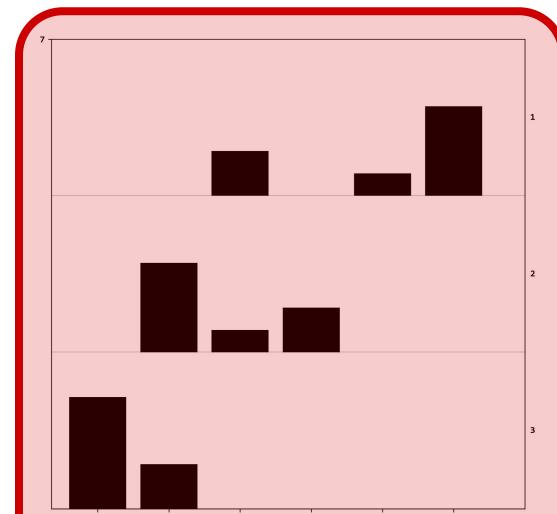
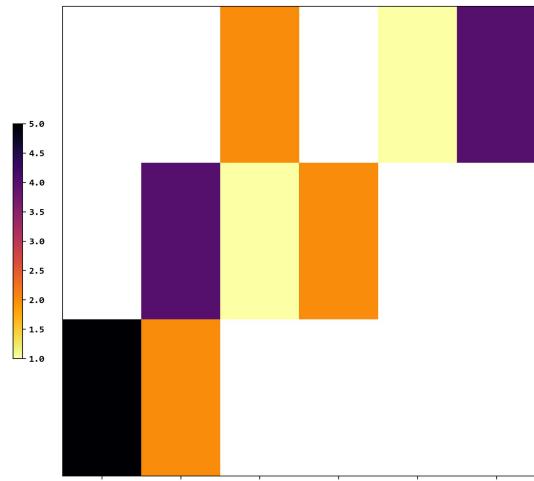
Graph | nx.graph_atlas(1115)

Nodes
Edges

7
14



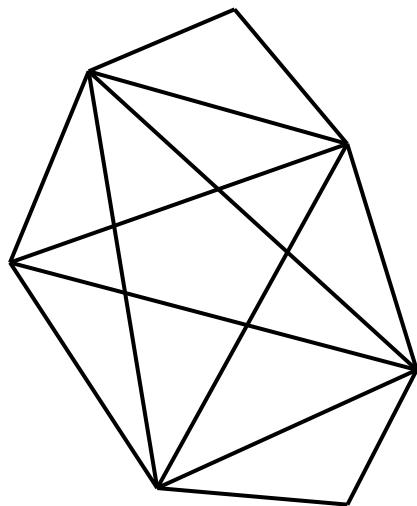
On this right side node counts are encoded with a **bar charts** (histograms).



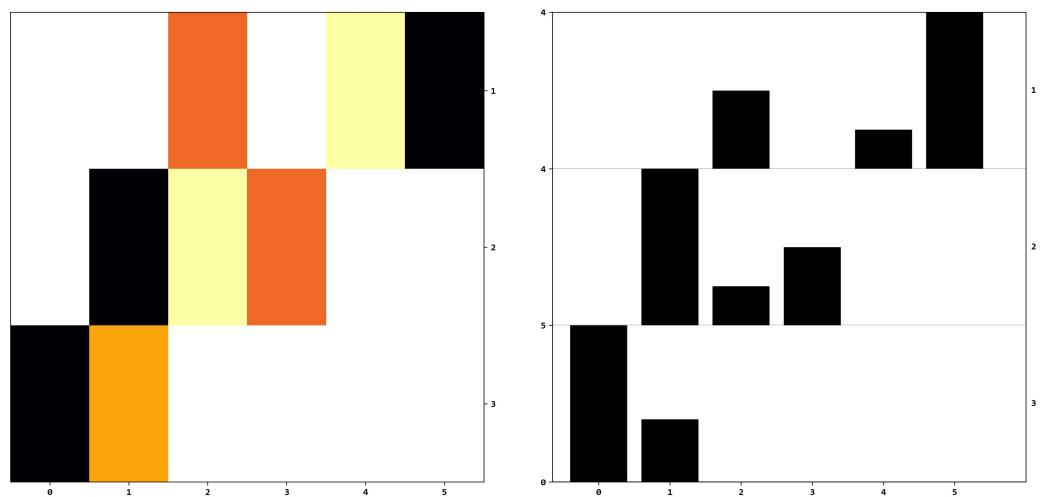
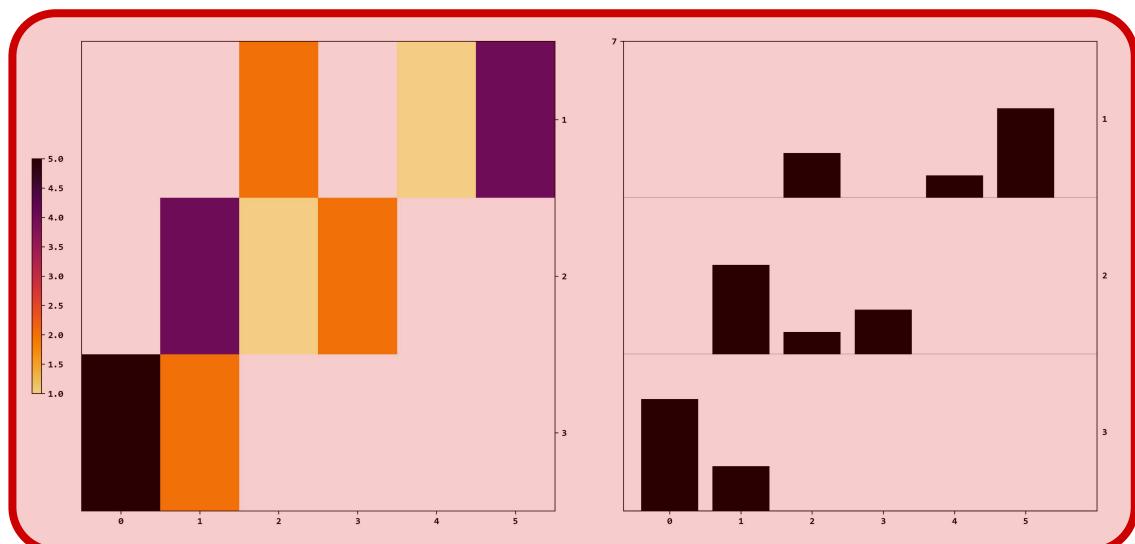
Graph | nx.graph_atlas(1115)

Nodes
Edges

7
14



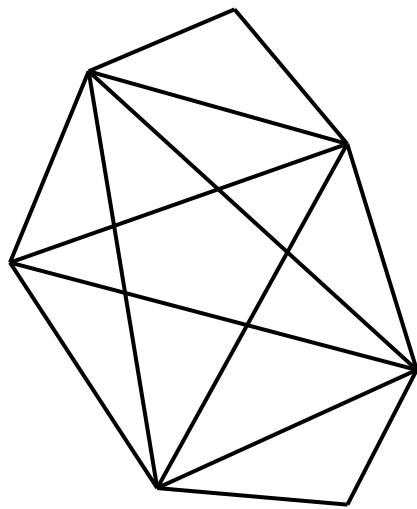
This top section has
global normalization
(shared range for
colormap and y-axis).



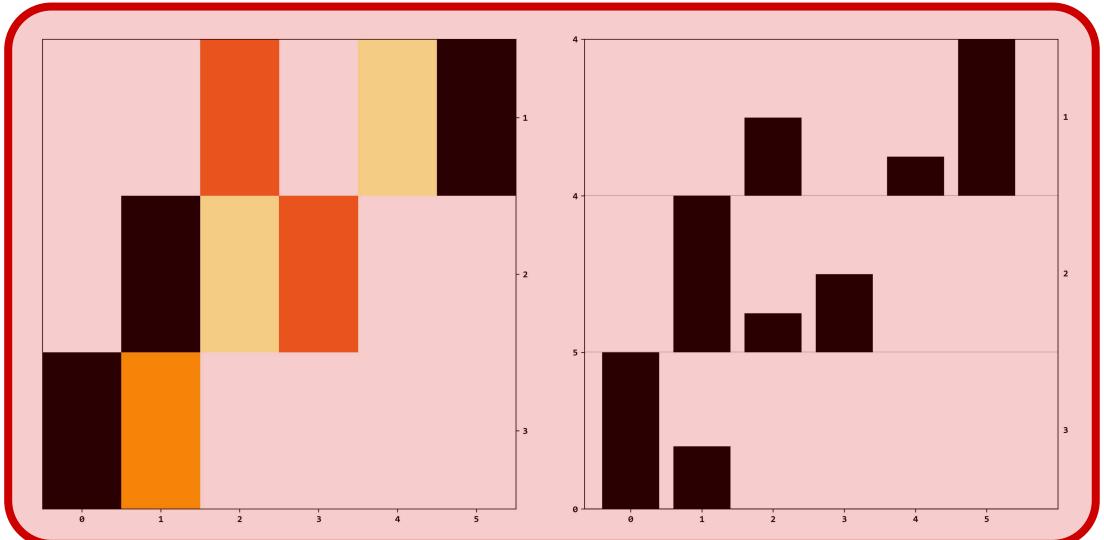
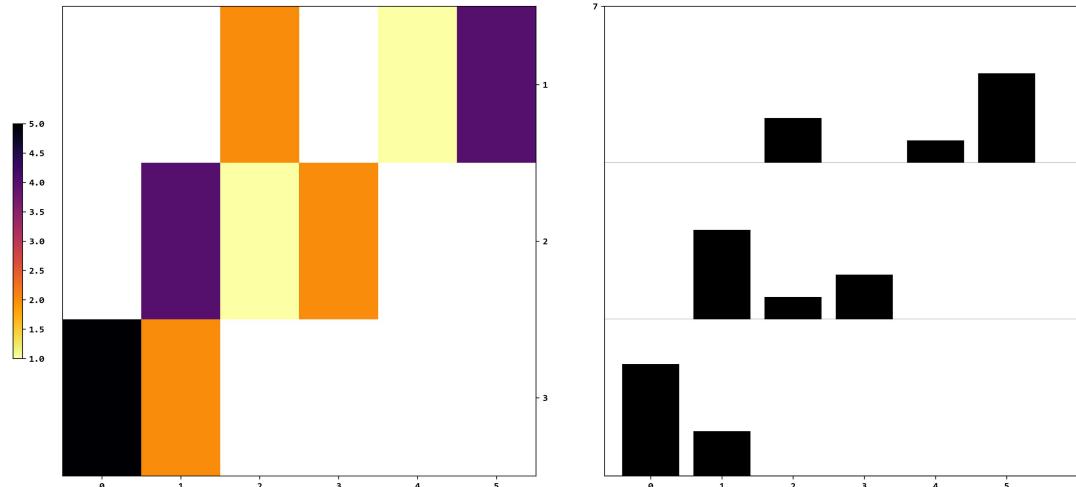
Graph | nx.graph_atlas(1115)

Nodes
Edges

7
14

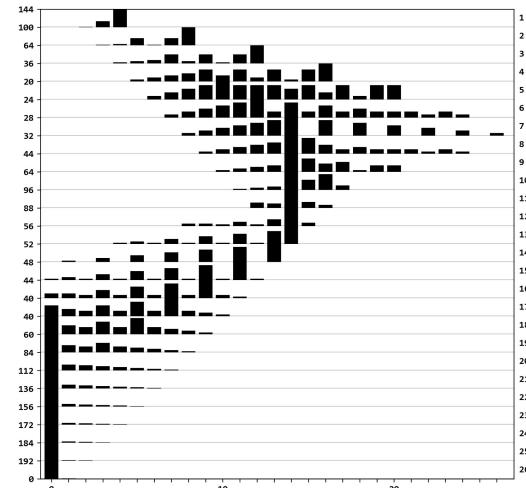
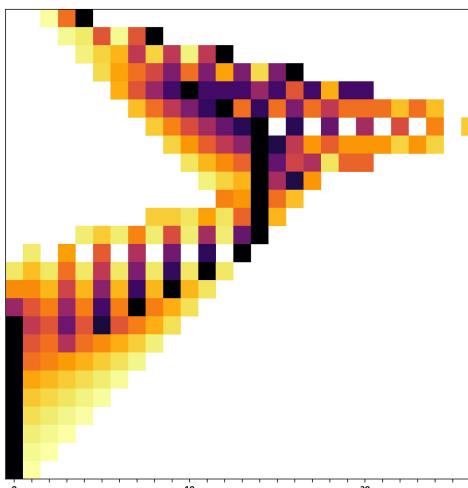
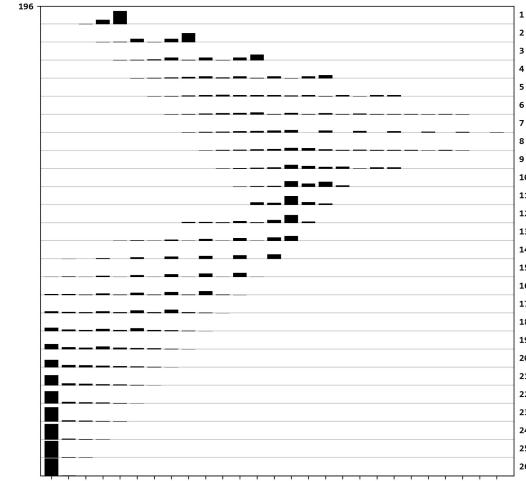
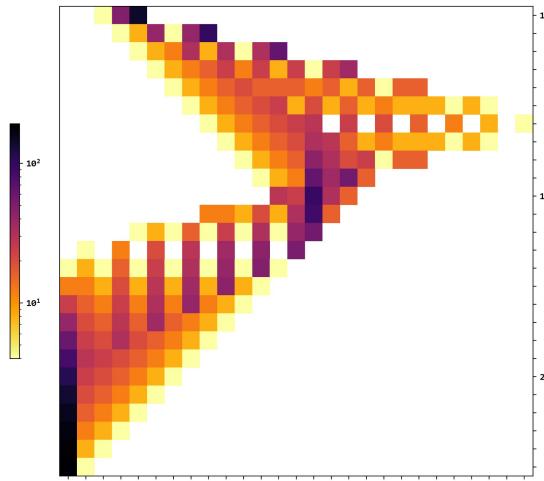
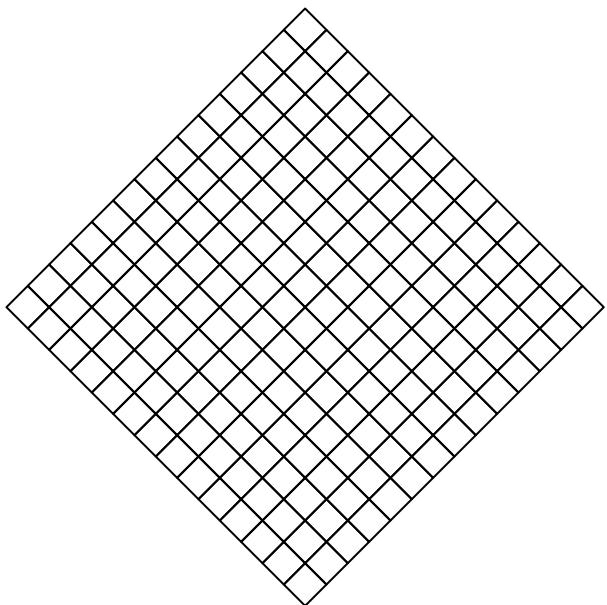


This bottom section has
per-row normalization (each
row has its own colormap and
y-axis range).



Graph | **Square Grid (14x14)**

Nodes 196
Edges 364



Returning to the example at the beginning, now with a quartet of visualizations.

Graph

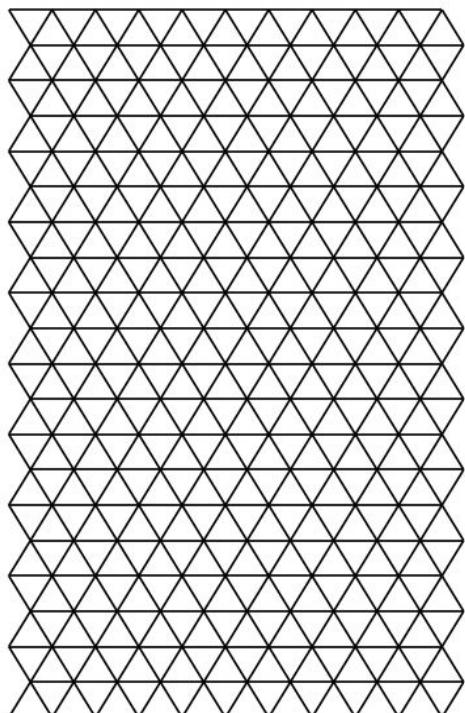
Planar triangle grid (20x20)

Nodes

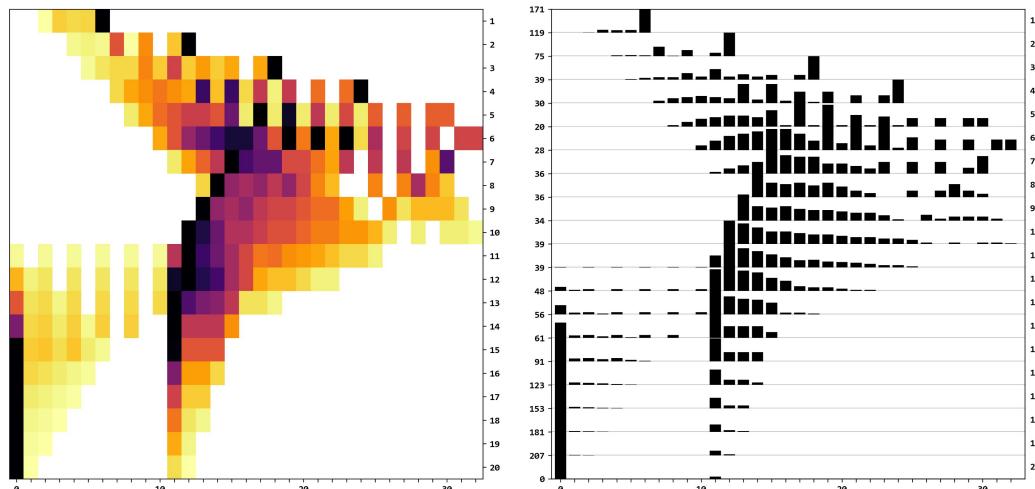
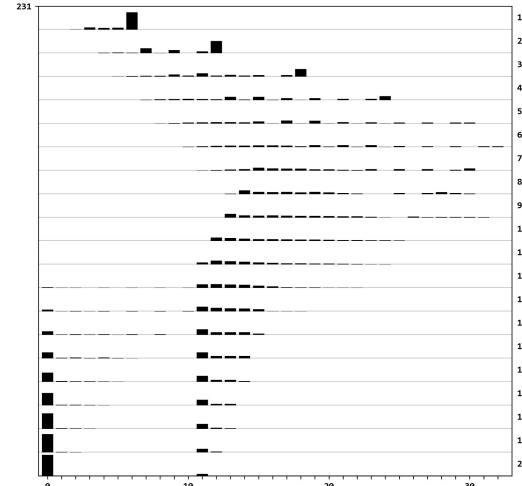
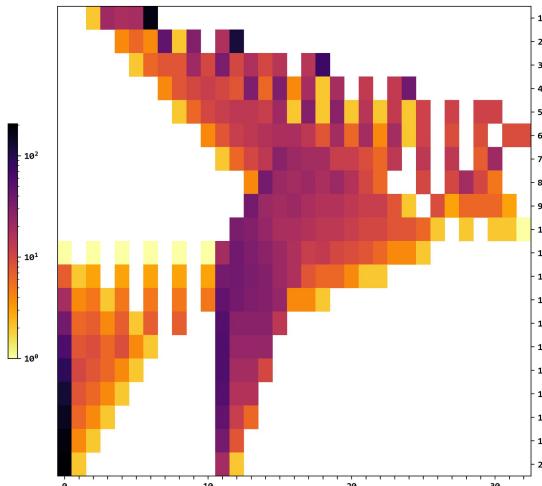
231

Edges

630



Another regular grid graph.



Graph

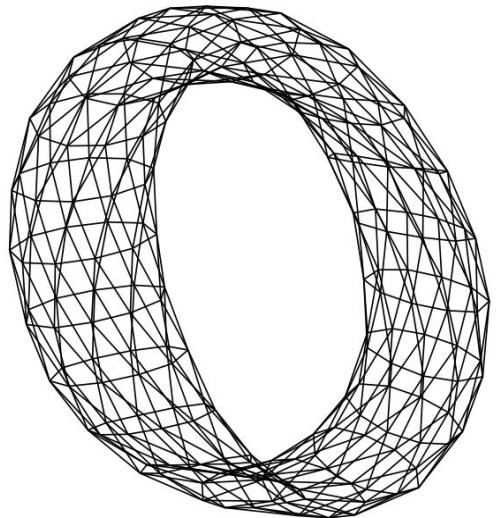
Closed triangle mesh (20x20)

Nodes

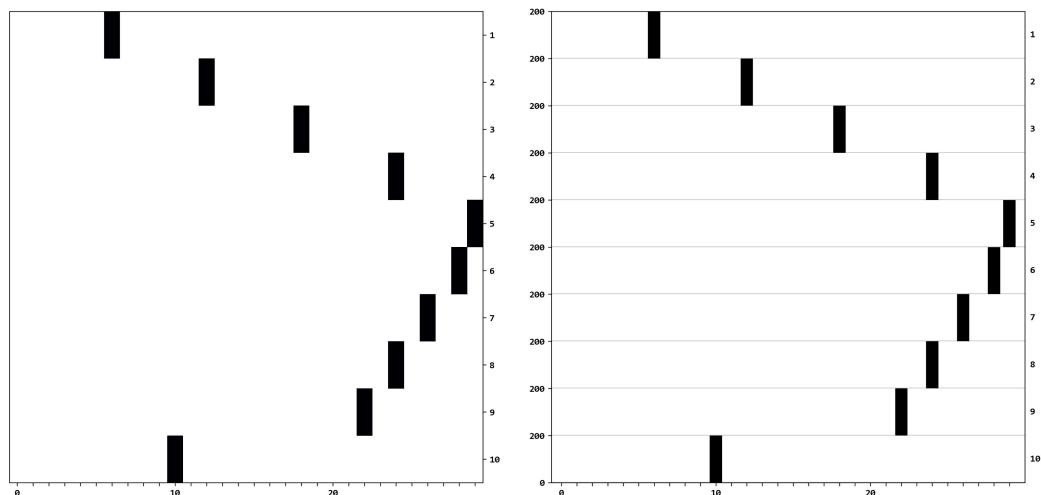
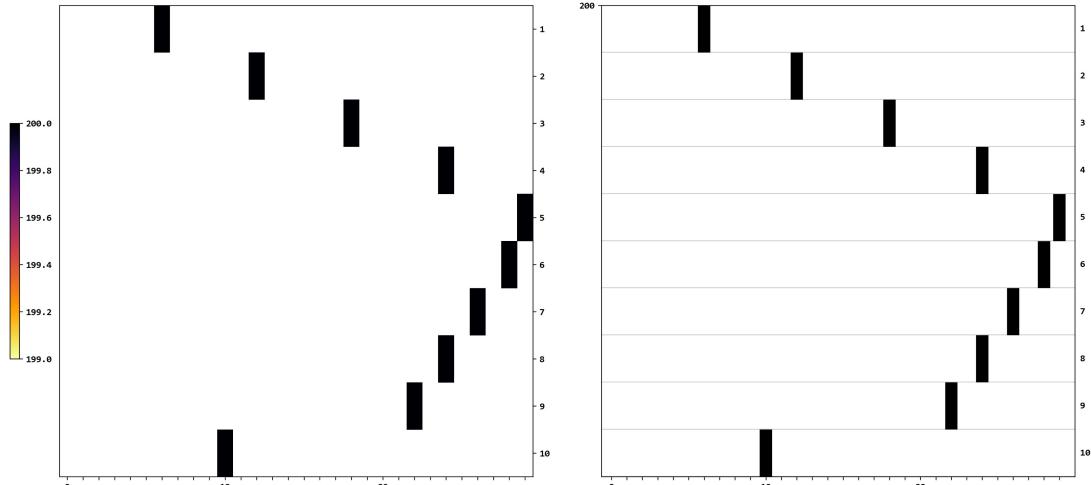
200

Edges

600



A periodic manifold has equal connections in all directions, so there is **no variance** in node degrees across hops.



Graph

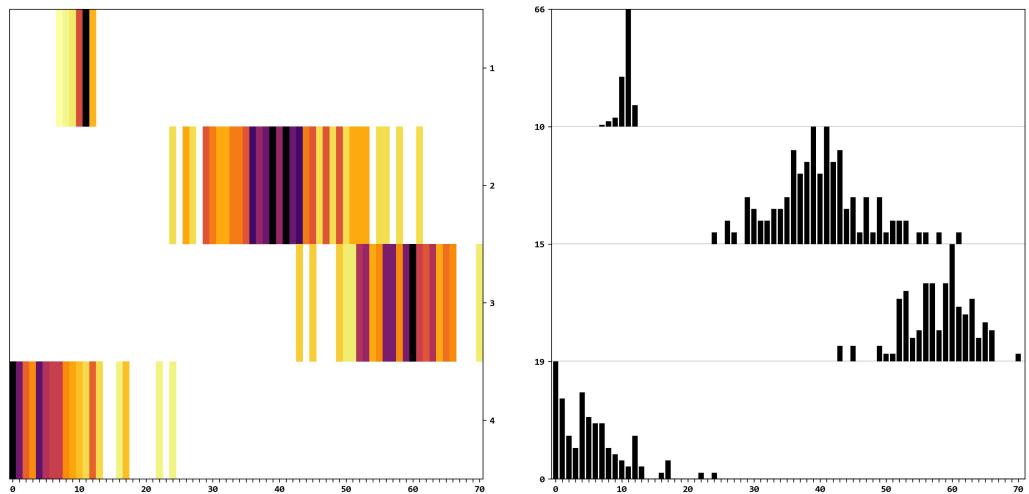
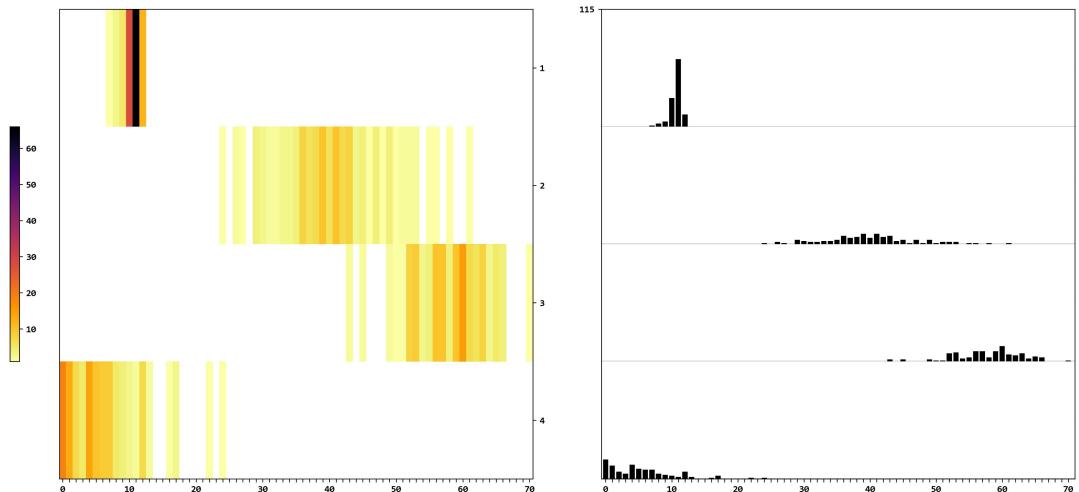
Fall 2000 College Football Games
personal.umich.edu/~mejn/netdata

Nodes

115

Edges

613



A real “small-world” network.

Graph

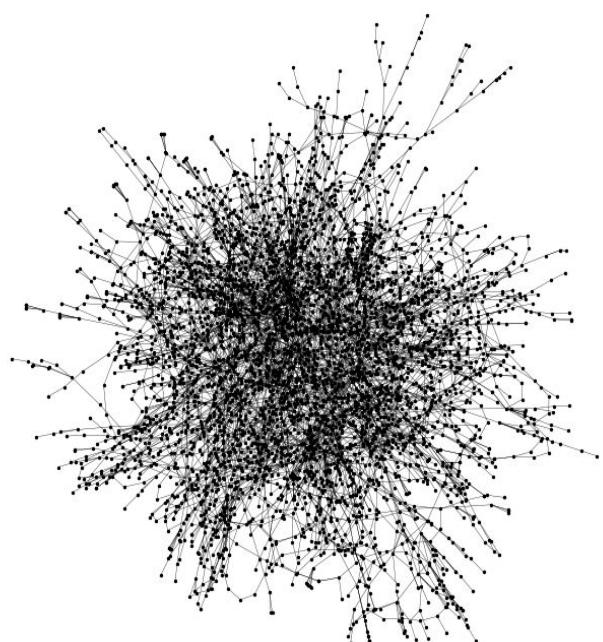
Western States Power Grid
personal.umich.edu/~mejn/netdata

Nodes

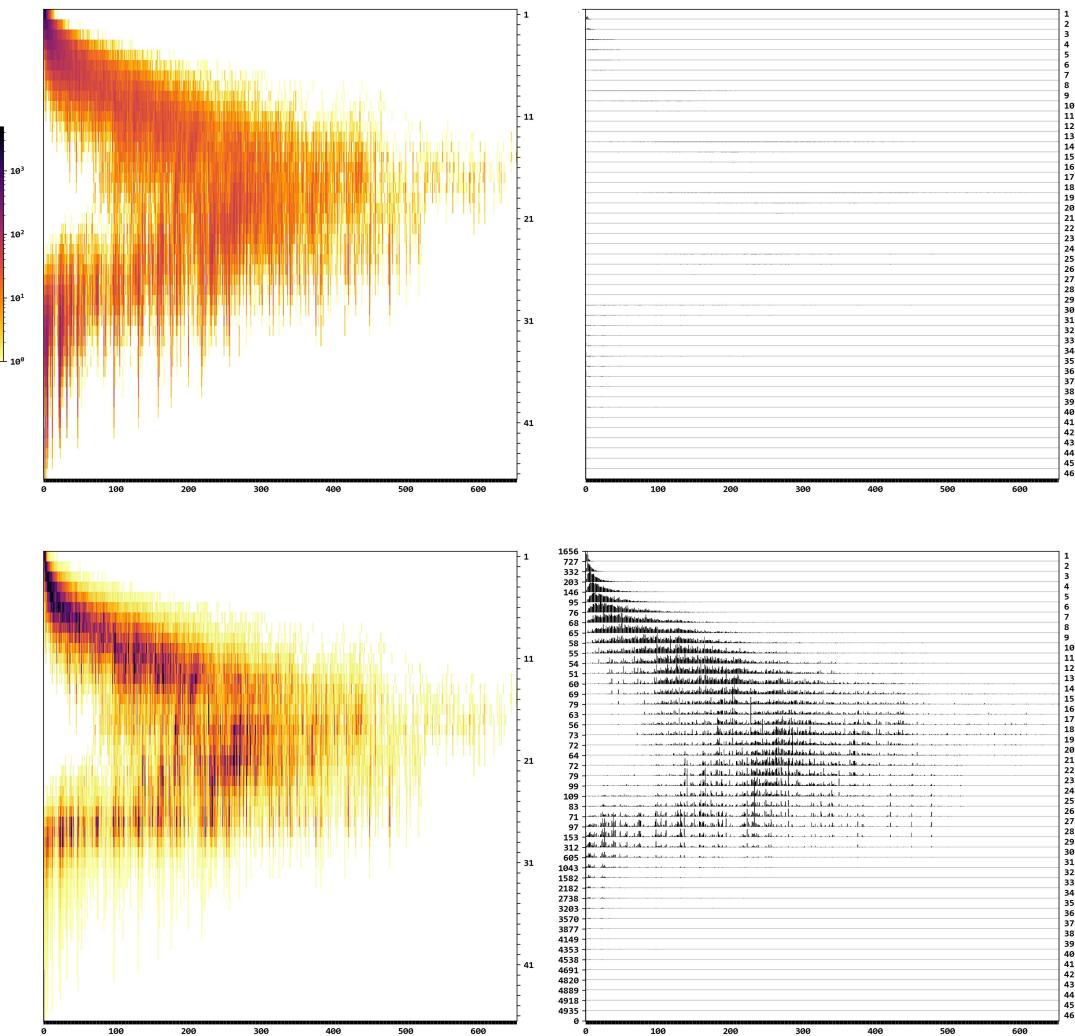
4941

Edges

6594



A real-world grid.



Graph

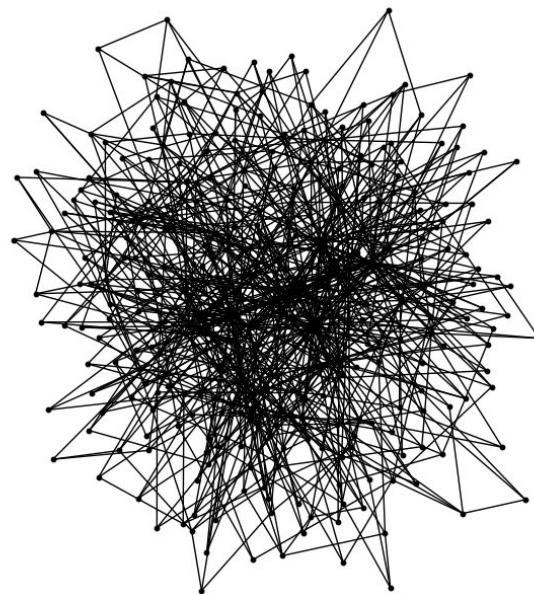
Barabási-Albert Network
nx.barabasi_albert_graph(300,3)

Nodes

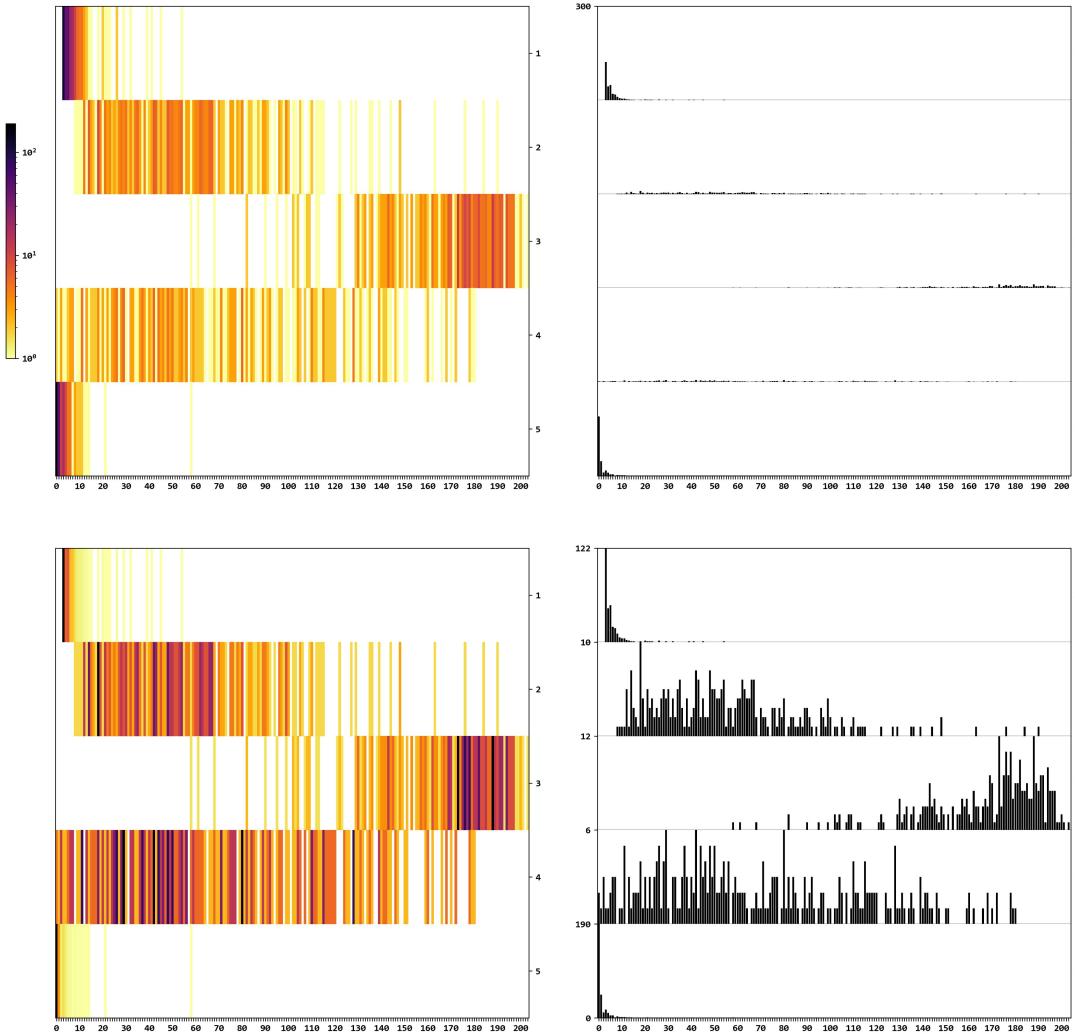
300

Edges

891



A model “small-world” graph.



Graph

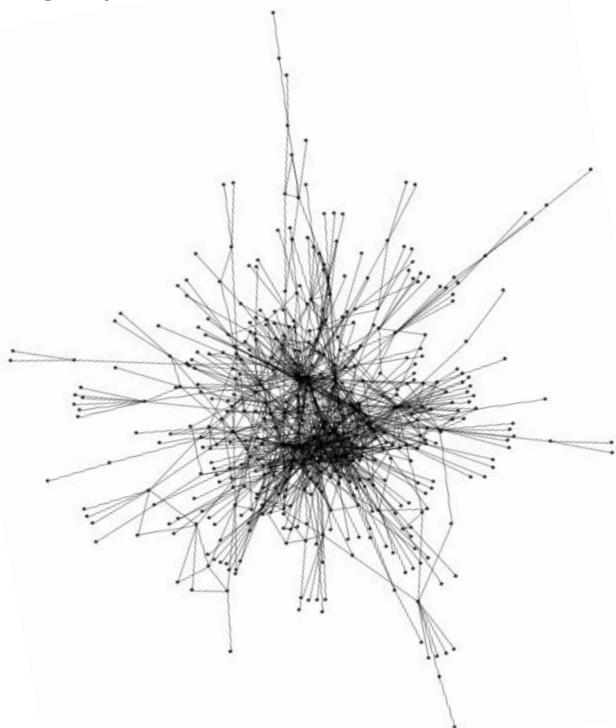
Semantic Network "Fortran" dirediredock/infobox_interlinker

Nodes

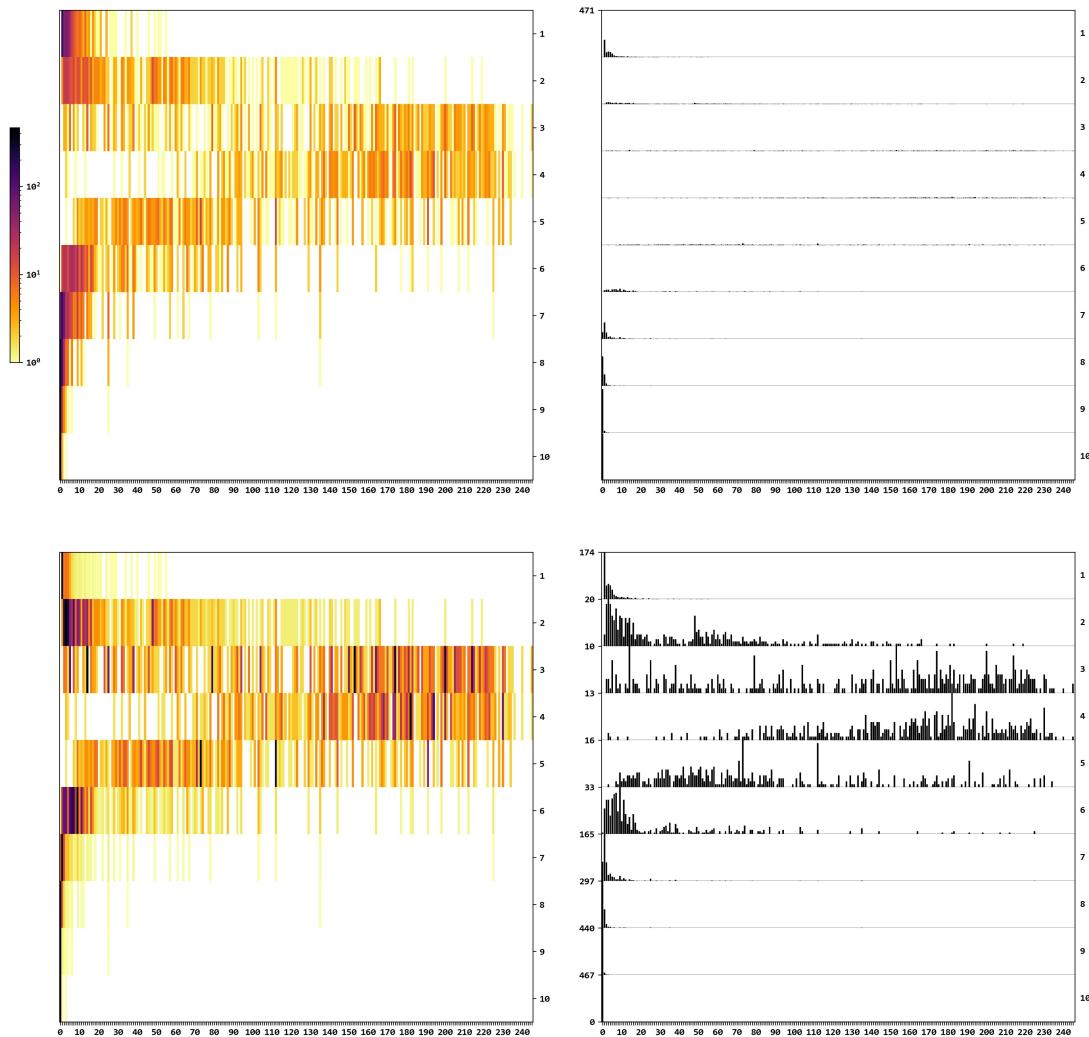
471

Edges

1169



A scraped
from Wikipedia.



Graph

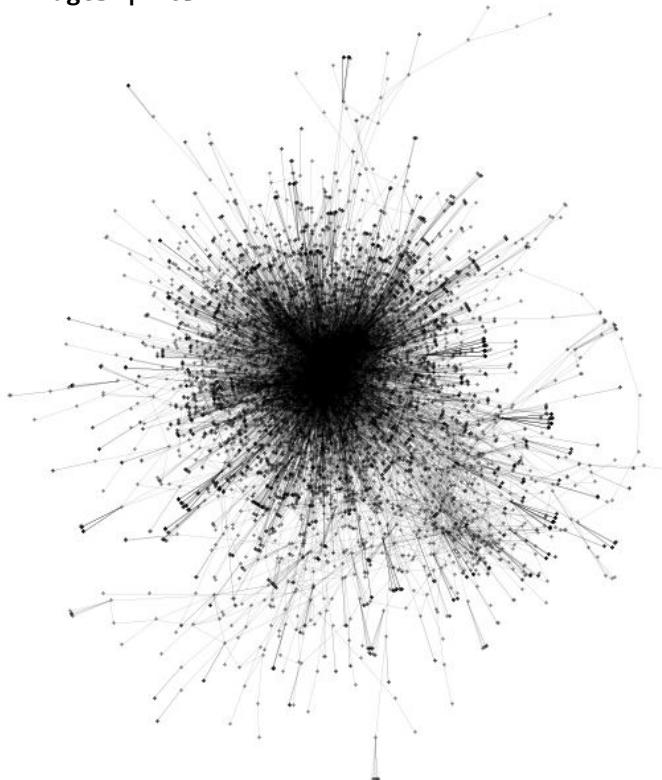
Semantic Network "Carl_Jung" dirediredock/infobox_interlinker

Nodes

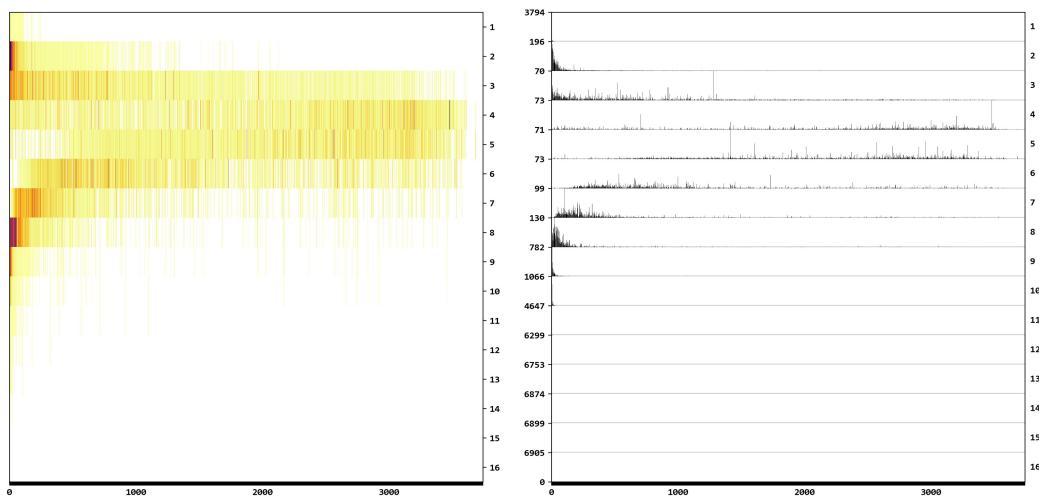
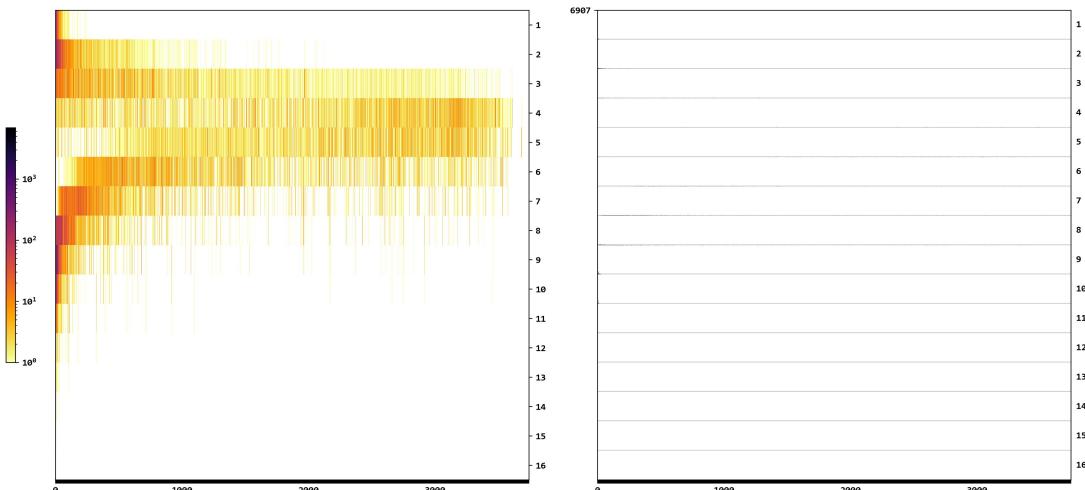
6907

Edges

16592



Another scraped
from Wikipedia.



Graph

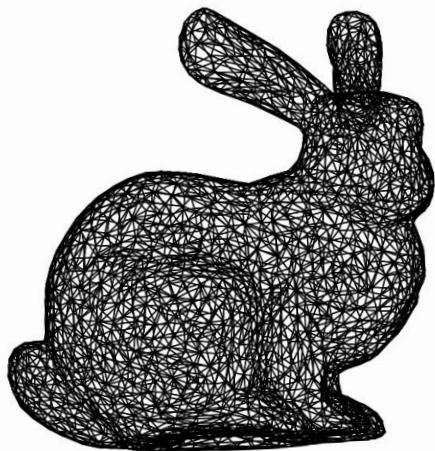
Stanford Bunny (edgelist)

Nodes

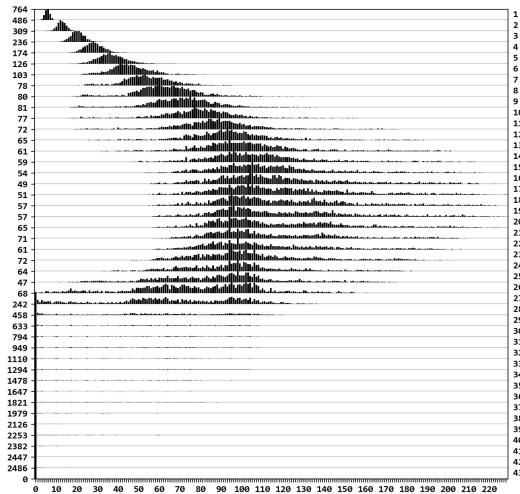
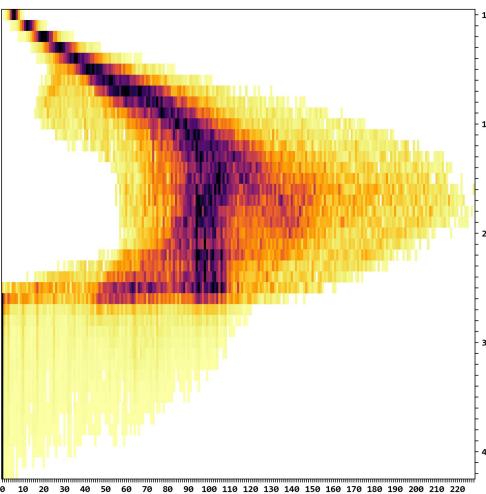
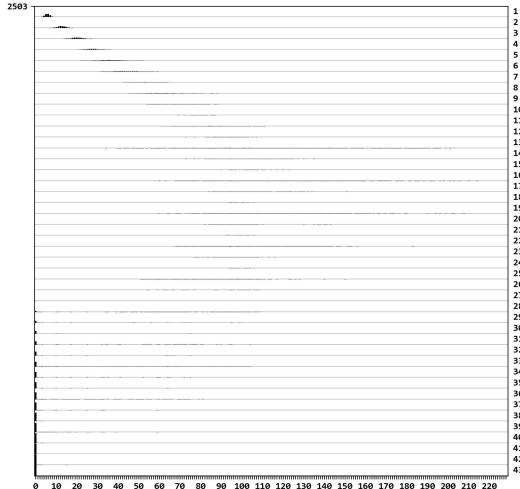
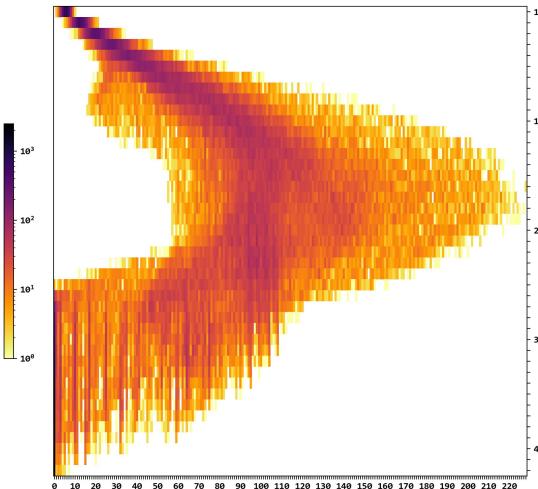
2593

Edges

7048

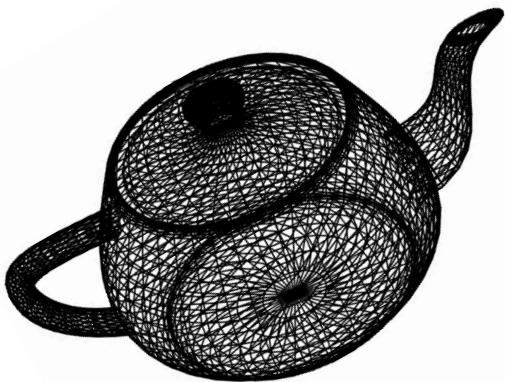


A famous example
from computer graphics.

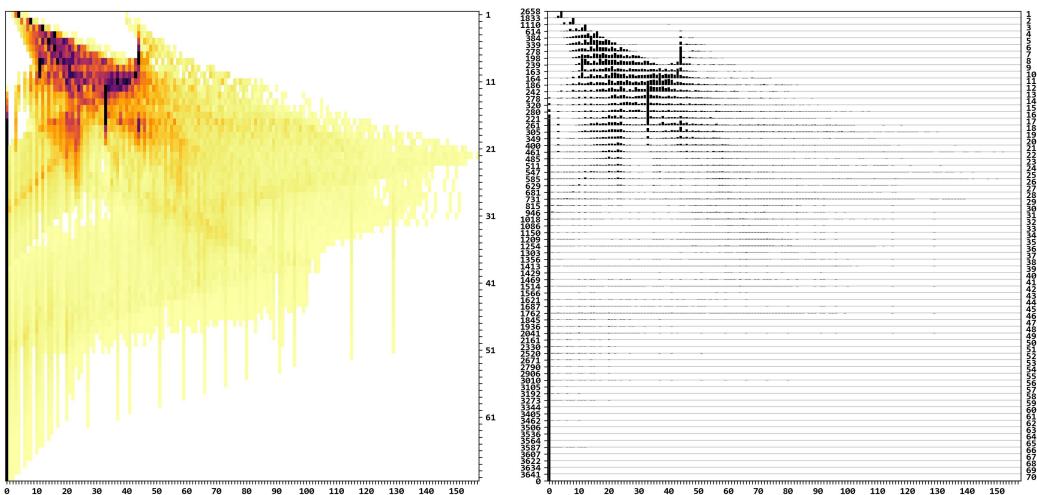
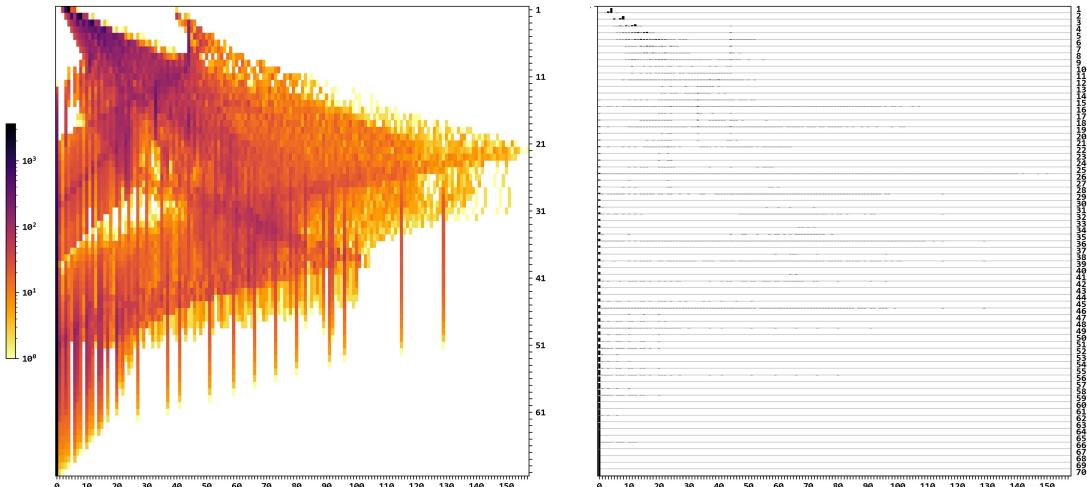


Graph | Utah Teapot (edgelist)

Nodes 3644
Edges 6870



Another famous example
from computer graphics.



IN SUMMARY

BMatrix_Explainer is a collection of Python scripts for the exploration and analysis of the B-Matrix of networks.

REQUIRED PYTHON LIBRARIES

`numpy`
`networkx`
`matplotlib`

MAIN SCRIPTS

`algorithm_BMatrix.py`

Modified from github.com/bagrow/portraits, exports the B-Matrix as a numpy array from a networkx graph object as input.

`plot_BMatrix_colormap.py`

Exports the canonical heatmap network portrait visualization, with the addition of optional row normalization.

`plot_BMatrix_histogram.py`

Exports an alternative monochromatic bar chart histogram visualization, also with optional per-row normalization.