XGI Documentation

Release 0.8.5

April 22, 2024

HOME

The Comple**X** **G**roup **I**nteractions (XGI) library provides data structures and algorithms for modeling and analyzing complex systems with group (higher-order) interactions.

Many datasets can be represented as graphs, where pairs of entities (or nodes) are related via links (or edges). Examples are road networks, energy grids, social networks, neural networks, etc. However, in many other datasets, more than two entities can be related at a time. For example, many scientists (entities) can collaborate on a scientific article together (links), and multiple email accounts (entities) can all participate on the same email thread (links). In this latter case, graphs no longer present a viable alternative to represent such datasets. It is for this kind of datasets, where the interactions are given among groups of more than two entities (also called higher-order interactions), that XGI was designed for.

XGI is implemented in pure Python and is designed to seamlessly interoperate with the rest of the Python scientific stack (numpy, scipy, pandas, matplotlib, etc). XGI is designed and developed by network scientists with the needs of network scientists in mind.

- Repository: https://github.com/xgi-org/xgi
- PyPI: latest release
- Twitter: @xginets
- List of Contributors
- Projects Using XGI

Sign up for our mailing list and follow XGI on Twitter or Mastodon!

CHAPTER

ONE

INSTALLATION

To install and use XGI as an end user, execute

```
pip install xgi
```

To install for development purposes, first clone the repository and then execute

```
pip install -e .['all']
```

If that command does not work, you may try the following instead

```
pip install -e .\[all\]
```

XGI was developed and tested for Python 3.8-3.12 on Mac OS, Windows, and Ubuntu.

CHAPTER

TWO

ACADEMIC REFERENCES

- The Why, How, and When of Representations for Complex Systems, Leo Torres, Ann S. Blevins, Danielle Bassett, and Tina Eliassi-Rad.
- Networks beyond pairwise interactions: Structure and dynamics, Federico Battiston, Giulia Cencetti, Iacopo Iacopini, Vito Latora, Maxime Lucas, Alice Patania, Jean-Gabriel Young, and Giovanni Petri.
- What are higher-order networks?, Christian Bick, Elizabeth Gross, Heather A. Harrington, Michael T. Schaub.

CHAPTER

THREE

CONTRIBUTING

If you want to contribute to this project, please make sure to read the contributing guidelines. We expect respectful and kind interactions by all contributors and users as laid out in our code of conduct.

The XGI community always welcomes contributions, no matter how small. We're happy to help troubleshoot XGI issues you run into, assist you if you would like to add functionality or fixes to the codebase, or answer any questions you may have.

Some concrete ways that you can get involved:

- **Get XGI updates** by following the XGI Twitter account, signing up for our mailing list, or starring this repository.
- **Spread the word** when you use XGI by sharing with your colleagues and friends.
- **Request a new feature or report a bug** by raising a new issue.
- **Create a Pull Request (PR)** to address an open issue or add a feature.
- **Join our Zulip channel** to be a part of the daily goings-on of XGI.

CHAPTER

FOUR

FUNDING

The XGI package has been supported by NSF Grant 2121905, HNDS-I: Using Hypergraphs to Study Spreading Processes in Complex Social Networks.

CHAPTER

FIVE

LICENSE

CHAPTER

SIX

CONTRIBUTORS

The XGI project has been helped by invaluable contributions from many members of the community. All listings are alphabetical.

6.1 Core team

6.2 Core team alumni

6.3 Contributors

CHAPTER

SEVEN

WHAT ARE HIGHER-ORDER INTERACTIONS?

Our world is highly interconnected, and examples of these connections include friends connecting on Facebook or meeting up for coffee; academic articles written by several authors; trains, busses, and planes traveling between cities; and even molecules held together by chemical bonds. Network science can describe each of these examples as a collection of *nodes* (The entities in the system, whether they be people, transit stations, or atoms) and *edges* (The connections between these entities, whether they be friendships, train routes connecting two cities, or a paper that co-authors wrote together). Traditional network science, however, assumes that only two of these entities may interact or associate at once, forming a *pairwise interaction*. The collection of these interactions is known as a *pairwise network*. This is often not the case in the physical world. Emails are often sent to more than one recipient, papers may be written by more than two authors, and friends may interaction not only one-to-one but in group settings as well. These group interactions are also known as *higher-order interactions*. For example, consider a group of friends. A higher-order network wouldn't just describe who is friends with whom individually, but also how sub-groups form and interact with one another. For example, the dynamics can change when three or more friends are together rather than just two.

A *higher-order network* is formed by the collection of these higher-order interactions. A higher-order interaction is represented most typically by a *hypergraph* or a *simplicial complex*. A hypergraph is a higher-order interaction network where each interaction is a set (a unique collection) listing the entities that participate in that interaction, also known as a *hyperedge*. A simplicial complex is a special type of hypergraph where, if an interaction occurs, it implies that every every possible sub-interaction also occurs. E.g., if authors 1, 2, and 3 wrote a paper together, then authors 1 and 2, 2 and 3, and 1 and 3 must have also co-authored a paper together (and 1, 2, 3 must all have sole-authored papers).

7.1 Why higher-order interactions?

Higher-order interactions can reveal more nuanced and sophisticated patterns of connection and can naturally encode different scales of interaction which are inaccessible to pairwise network representations. Higher-order networks can be helpful for describing social networks, ecological communities, co-authorship or citation networks, email, protein interactions, and many more examples. Higher-order networks can also exhibit rich dynamical behavior for simple models of contagion, synchronization, and opinion formation.

CHAPTER

EIGHT

XGI-DATA

XGI-DATA is a repository of openly available hypergraph datasets in JSON format with corresponding documentation of network statistics, limitations of the data, and methods of collection. They are hosted in the XGI Community on Zenodo. This is loosely inspired by Datasheets for Datasets by Gebru et al.

8.1 Dataset format

The xgi-data format for higher-order datasets is a JSON data structure with the following structure:

- `hypergraph-data`: This tag accesses the attributes of the entire hypergraph dataset such as the authors or dataset name.
- `node-data`: This tag accesses the nodes of the hypergraph and their associated properties as a dictionary where the keys are node IDs and the corresponding values are dictionaries. If a node doesn't have any properties, the associated dictionary is empty.
  - `name`: This tag accesses the node's name if there is one that is different from the ID specified in the hyperedges.
  - Other tags are user-specified based on the particular attributes provided by the dataset.
- `edge-data`: This tag accesses the hyperedges of the hypergraph and their associated attributes.
  - `name`: This tag accesses the edge's name if one is provided.
  - `timestamp`: This is the tag specifying the time associated with the hyperedge if it is given. All times are stored in ISO8601 standard.
  - Other tags are user-specified based on the particular attributes provided by the dataset.
- `edge-dict`: This tag accesses the edge IDs and the corresponding nodes which participate in that hyperedge.

8.2 Loading datasets

Loading a dataset using XGI is as simple as the following two lines:

```
import xgi
H = xgi.load_xgi_data("<dataset_name>")
```

XGI-DATA uses an HTTP request to load the hypergraph dataset.

8.3 Network Statistics

CHAPTER

NINE

GALLERY

Here are several examples of figures made using XGI:

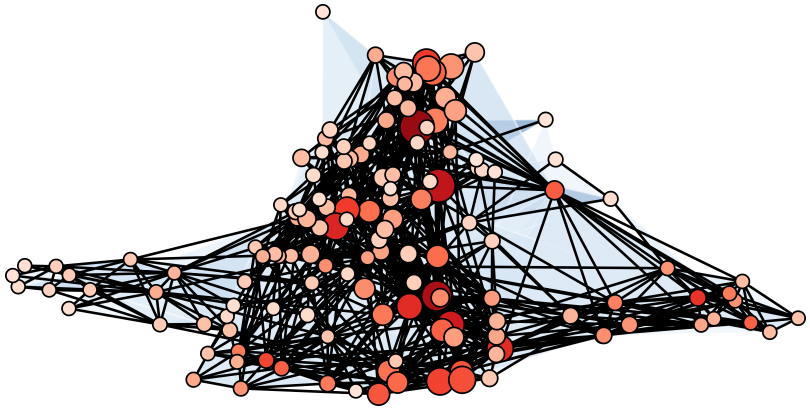


Fig. 1: A visualization of the email-enron dataset with hyperedges of sizes 2 and 3 (all isolated nodes removed). The nodes are colored by their degree and their size proportional to the Clique motif Eigenvector Centrality. (Fig. 2 in XGI: A Python package for higher-order interaction networks)

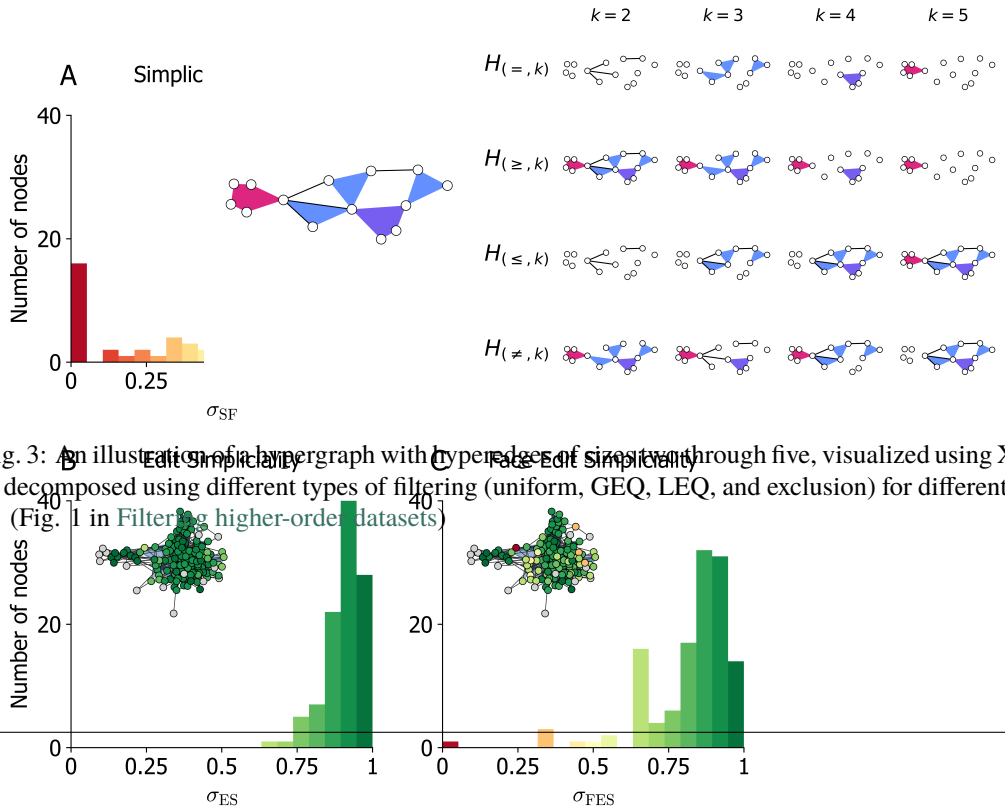


Fig. 3: An illustration of a hypergraph with hyperedges of sizes two through five, visualized using XGI. This hypergraph is decomposed using different types of filtering (uniform, GEQ, LEQ, and exclusion) for different filtering parameters, k. (Fig. 1 in Filtering higher-order datasets)

CHAPTER

TEN

PROJECTS USING XGI

XGI has been used in a variety of published work and other software.

Articles are listed by year and then alphabetically by the last name of the first author (and title if necessary).

10.1 Published work

10.1.1 2024

Nicholas W. Landry, Ilya Amburg, Mirah Shi, and Sinan Aksoy, "Filtering higher-order datasets", *Journal of Physics: Complexity* **5** 015006 (2024).

Paper Code

Nicholas W. Landry, Jean-Gabriel Young, and Nicole Eikmeier, "The simpliciality of higher-order networks", *EPJ Data Science* **13**, 17 (2024).

Paper Code

10.1.2 2023

Gonzalo Contreras-Aso, Regino Criado, Guillermo Vera de Salas, and Jinling Yang, "Detecting communities in higher-order networks by using their derivative graphs", *Chaos, Solitons, and Fractals* **177**, 114200 (2023).

Paper Code

Nicholas W. Landry and Juan Restrepo, "Opinion disparity in hypergraphs with community structure", *Physical Review E* **108**, 034311 (2023).

Paper Code

Timothy LaRock and Renaud Lambiotte, "Encapsulation Structure and Dynamics in Hypergraphs", *Journal of Physics: Complexity* **4**, 045007 (2023).

Paper Code

Nicolò Ruggieri, Martina Contisciani, Federico Battiston, and Caterina de Bacco, "Community detection in large hypergraphs", *Science Advances* **9**, eadg9159 (2023).

Paper Code

Yuanzhao Zhang, Maxime Lucas, and Federico Battiston, "Higher-order interactions shape collective dynamics differently in hypergraphs and simplicial complexes", *Nature Communications* **14**, 1605 (2023).

Paper Code

10.2 Preprints

10.2.1 2024

Gonzalo Contreras-Aso, Regino Criado, and Miguel Romance, "Beyond directed hypergraphs: heterogeneous hypergraphs and spectral centralities", arXiv:2403.11825 (2024).

Paper Code

Maxime Lucas, Luca Gallo, Arsham Ghavasieh, Federico Battiston, and Manlio De Domenico, "Functional reducibility of higher-order networks", arXiv:2404.08547 (2024).

Paper Code

10.2.2 2023

Gonzalo Contreras-Aso, Cristian Pérez-Corral, and Miguel Romance, "Uplifting edges in higher order networks: spectral centralities for non-uniform hypergraphs", arXiv:2310.20335 (2023).

Paper Code

Iacopo Iacopini, Márton Karsai, and Alain Barrat, "The temporal dynamics of group interactions in higher-order social networks", arXiv:2306.09967 (2023).

Paper Code

Yuanzhao Zhang, Per Sebastian Skardal, Federico Battiston, Giovanni Petri, Maxime Lucas, "Deeper but smaller: Higher-order interactions increase linear stability but shrink basins", arXiv:2309.16581 (2023).

Paper Code

10.3 Theses

Leonie Neuhäuser, "Modelling the effect of groups on network structure and dynamics", (2023).

Thesis

Thomas Robiglio, "Higher-order structures in face-to-face interaction networks", (2023).

Thesis

10.4 Software packages

- hypercontagion
- hyperspec
- pynetflow
- segram
- simplicial-kuramoto

CHAPTER

ELEVEN

TUTORIALS

11.1 Getting started

11.1.1 XGI in 1 minute

Hello!

Let's dive right in and import our XGI library. We just need to execute the following code snippet:

[1]:
```
import xgi
```

Now we're ready to start using XGI!

Creating a Hypergraph

You have the flexibility to create a hypergraph from various data structures. One of the simplest ways is by using a list of hyperedges. Check out this code snippet:

[2]:
```
hyperedges = [[1, 2, 3], [3, 4], [4, 5, 6, 7]]
H = xgi.Hypergraph(hyperedges)
```

Adding Nodes and Hyperedges

Once you have a hypergraph, you can start adding single nodes and hyperedges using these convenient functions:

[3]:
```
H.add_node(8)
H.add_edge([7, 8])
```

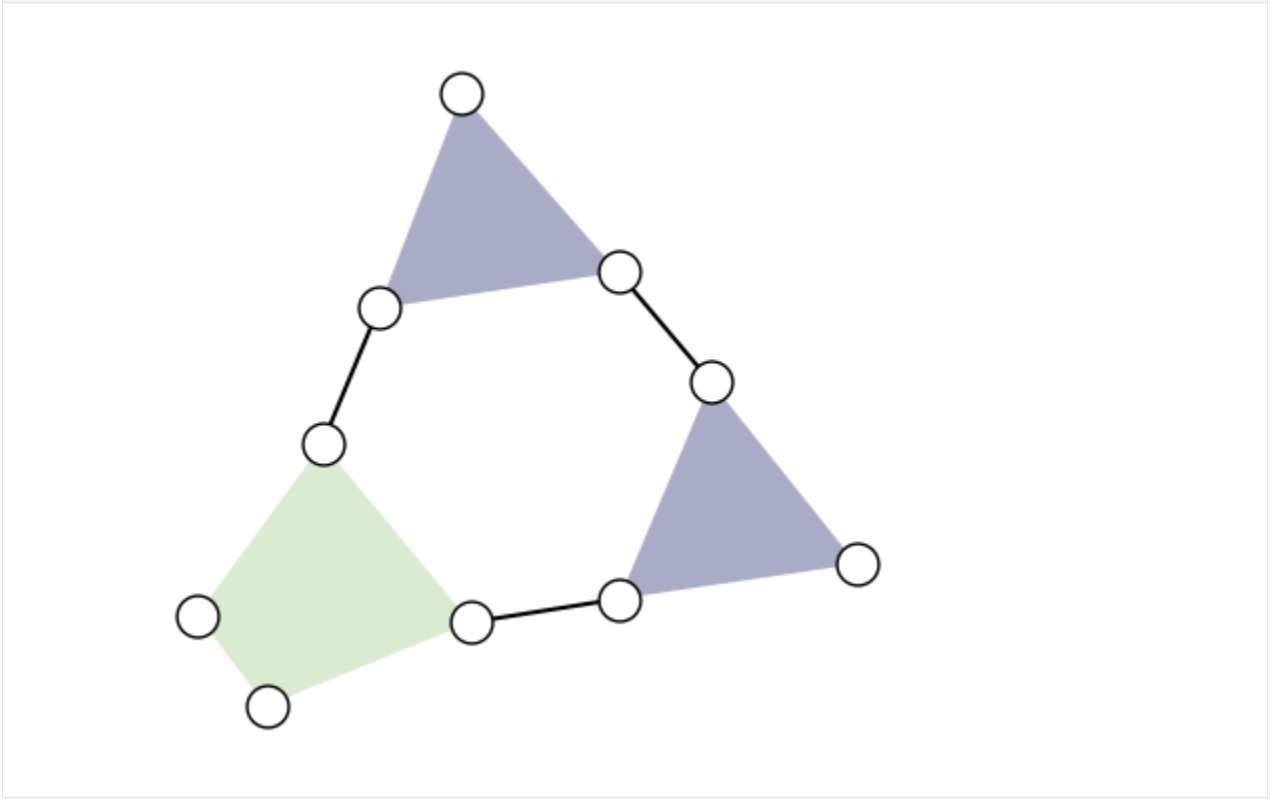
If you want to add multiple nodes or hyperedges at once, XGI has got you covered:

[4]:
```
H.add_nodes_from([9, 10])
H.add_edges_from([[1, 10], [8, 9, 10]])
```

Visualizing Your Hypergraph

Visualization is key to understanding complex data structures. With XGI, you can effortlessly plot your hypergraph using the default drawing setup:

[5]:
```
xgi.draw(H);
```



For more options, explore the focus tutorial on plotting or consult the documentation!

Exploring Hypergraph Details

XGI provides handy functions to gain insights into your hypergraph. For a quick summary, simply print the hypergraph:

[6]:
```
print(H)
```

```
Unnamed Hypergraph with 10 nodes and 6 hyperedges
```

You can also access the number of nodes and hyperedges easily:

[7]:
```
print(H.num_nodes)
print(H.num_edges)
```

```
10
6
```

And if you're curious about all the hyperedges in your hypergraph, you can retrieve them as a list:

[8]:
```
H.edges.members()
```

```
[8]: [{1, 2, 3}, {3, 4}, {4, 5, 6, 7}, {7, 8}, {1, 10}, {8, 9, 10}]
```

Wrapping Up

Time flies! Don't forget to check out tutorials here! And guess what? XGI doesn't stop at hypergraphs — it can handle simplicial complexes and directed hypergraphs too!

11.1.2 XGI in 5 minutes

Hello!

If you're new to XGI, you might want to check out the XGI in 1 minute tutorial for a quick introduction.

Getting Started

Let's import XGI in the usual way, and this time we'll need a few other standard Python libraries too.

```
[1]: import matplotlib.pyplot as plt

     import xgi
```

To check the version of XGI you have, simply type:

```
[2]: xgi.__version__
```

```
[2]: '0.7.4'
```

Creating a Random Hypergraph

In XGI, you have several options to create a hypergraph. You can build an empty one and add nodes and edges manually (as we have seen in the XGI in 1 minute tutorial), or you can use our handy generators. For this tutorial, let's use a simple generator to create a random hypergraph. If you're curious about other generators, feel free to explore the focus tutorial on generators or consult the documentation.

To create a random hypergraph, use this function:

```
[3]: N = 20
     ps = [0.1, 0.01]
     H = xgi.random_hypergraph(N, ps, seed=1)
```

This function generates a random hypergraph with `N` nodes, connecting any `d+1` nodes with a hyperedge using probability `ps[d-1]`.

You can also print the hypergraph and access the list of nodes and edges like this:

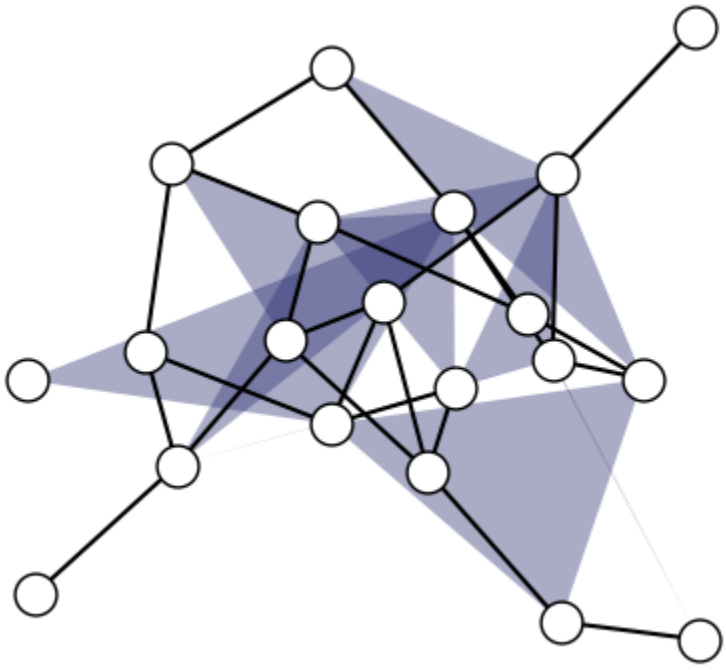
```
[4]: print(H)
     print(H.nodes)
     print(H.edges.members())
```

```
Unnamed Hypergraph with 20 nodes and 36 hyperedges
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[{0, 3}, {0, 6}, {0, 15}, {1, 9}, {1, 10}, {1, 13}, {2, 4}, {16, 2}, {3, 5}, {3, 14},
↪{19, 5}, {6, 14}, {10, 7}, {16, 7}, {17, 7}, {18, 7}, {8, 18}, {9, 17}, {9, 19}, {10,␣
↪15}, {16, 10}, {16, 12}, {17, 12}, {18, 14}, {19, 15}, {0, 4, 14}, {1, 15, 7}, {1, 12,␣
↪14}, {17, 2, 6}, {18, 3, 5}, {18, 3, 6}, {11, 17, 3}, {3, 12, 15}, {18, 15, 7}, {10,␣
↪19, 15}, {18, 12, 14}]
```

Plotting a random hypergraph

Visualization is crucial for understanding complex data structures. To plot your hypergraph using the default layout, use this:

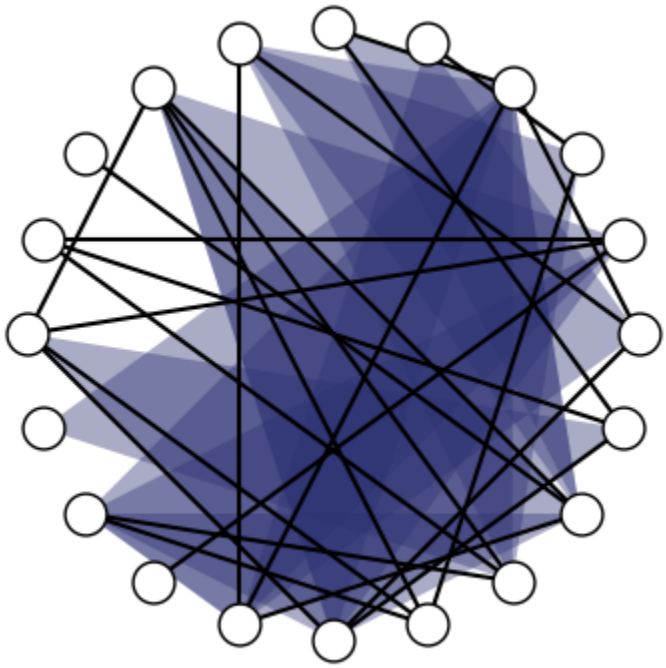
[5]:
```
pos = xgi.barycenter_spring_layout(H, seed=1)
xgi.draw(H, pos=pos);
```



XGI also offers several ways to personalize your plots for higher-order structures. For more options, explore the focus tutorial on plotting or consult the documentation.

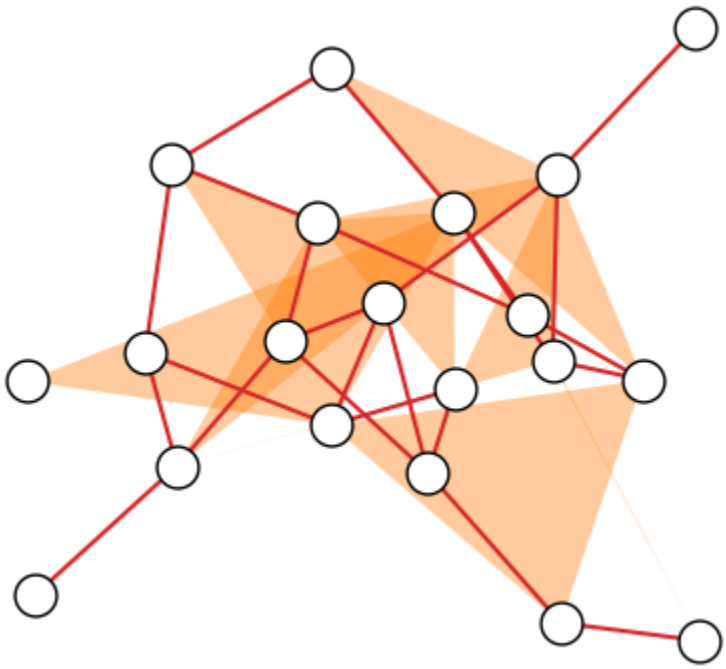
For example, you can place the nodes on a circle:

[6]:
```
pos_circular = xgi.circular_layout(H)
xgi.draw(H, pos=pos_circular);
```



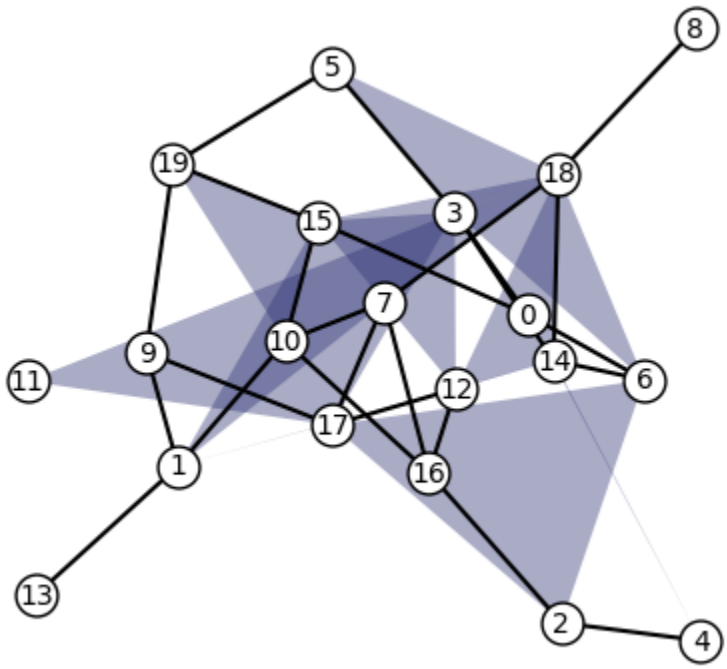
Or change the colors of hyperedges and links:

[7]:
```
xgi.draw(H, dyad_color="tab:red", edge_fc="tab:orange", pos=pos);
```



Adding node labels is another handy feature:

[8]:
```
xgi.draw(H, node_labels=True, pos=pos);
```



Accessing the maximum order of your hypergraph

You might need to know the maximum number of edges in your hypergraph. To do that, simply use this:

[9]:
```
xgi.max_edge_order(H)
```

[9]:
```
2
```

This can be particularly helpful when working with larger and more complex structures.

Listing All Edge Sizes

To access all the edge sizes in your hypergraph, use this:

[10]:
```
xgi.unique_edge_sizes(H)
```

[10]:
```
[2, 3]
```

Histogram of the edge sizes

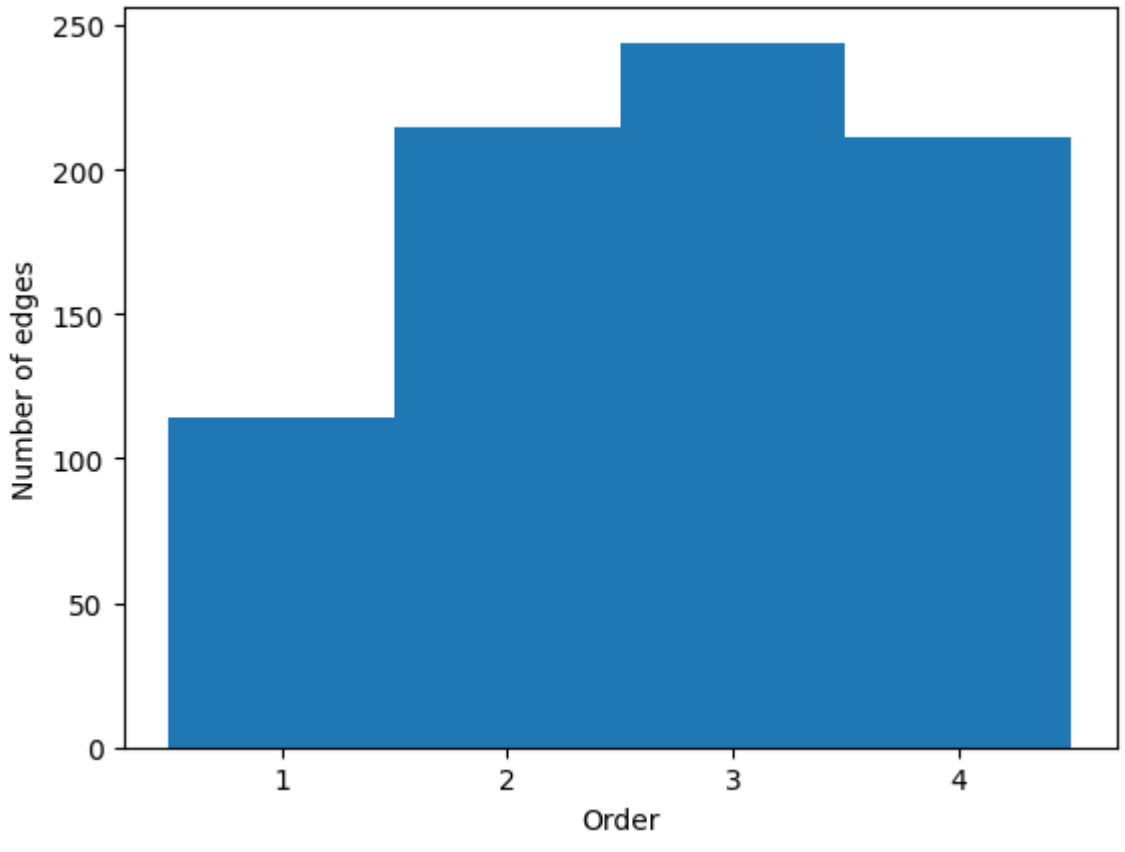
Let's take things up a notch and perform a more detailed analysis on a hypergraph: plotting a histogram of the edges' orders.

First, let's create a larger and more intricate random hypergraph:

[11]:
```
N_new = 50
ps_new = [0.1, 0.01, 0.001, 0.0001]
H_new = xgi.random_hypergraph(N_new, ps_new)
```

To access the order of the edges, use the `stats` function. For a deeper dive into this, check out the focus tutorial on statistics or consult the documentation. You can obtain a list of all the edge orders and then create a histogram in the usual way.

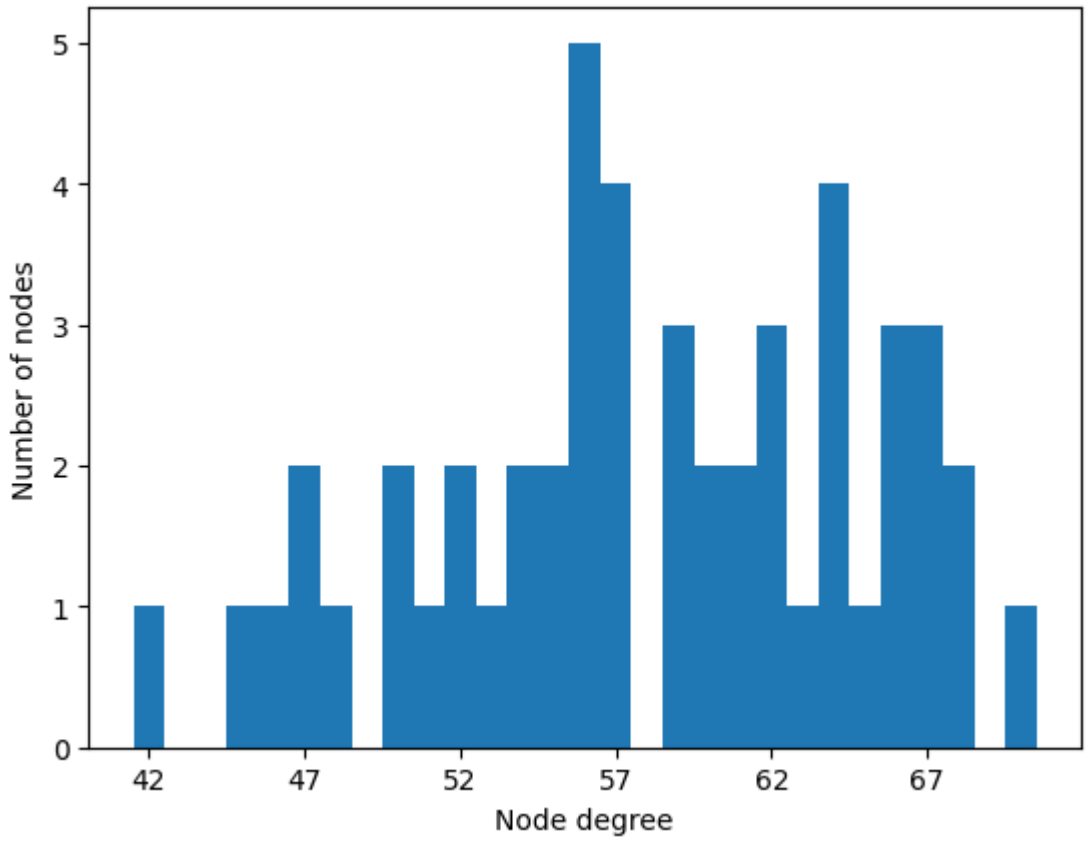
[12]:
```
list_of_orders = H_new.edges.order.aslist()
plt.hist(
    list_of_orders,
    bins=range(min(list_of_orders), max(list_of_orders) + 2, 1),
    align="left",
)
plt.xticks(range(min(list_of_orders), max(list_of_orders) + 1, 1))
plt.xlabel("Order")
plt.ylabel("Number of edges");
```



Histogram of the Node Degrees

Similarly, using the `stats` function, you can create a histogram of the node degrees in your hypergraph:

[13]:
```
nodes_degrees_list = H_new.nodes.degree.aslist()
plt.hist(
    nodes_degrees_list,
    bins=range(min(nodes_degrees_list), max(nodes_degrees_list) + 1, 1),
    align="left",
)
plt.xticks(range(min(nodes_degrees_list), max(nodes_degrees_list) + 1, 5))
plt.xlabel("Node degree")
plt.ylabel("Number of nodes");
```



Wrapping Up

Well done! You've covered a lot in just 5 minutes with XGI. We hope you enjoyed this tutorial, and there's much more to explore! Check out other tutorials here!

11.1.3 XGI in 15 minutes

Hello! If you are new to XGI you might want to check out the XGI in 1 minute or the XGI in 5 minutes tutorials for a quick introduction.

The starting point is always to import our Python library and other standard libraries, this is simply done using:

[1]:
```
import matplotlib.pyplot as plt

import xgi
```

Uploading a dataset

In this tutorial we will construct a hypergraph describing real world data! With XGI we provide a companion data repository, xgi-data, with which you can easely load several datasets in standard format:

[2]:
```
H_enron = xgi.load_xgi_data("email-enron")
```

The 'email-enron' dataset, for example, has a corresponding datasheet explaining its characteristics. The nodes (individuals) in this dataset contain associated email addresses and the edges (emails) contain associated timestamps. These attributes can be accessed by simply typing `H.nodes[id]` or `H.edges[id]` respectively.

[3]:
```
print(f"The hypergraph has {H_enron.num_nodes} nodes and {H_enron.num_edges} edges")
```

```
The hypergraph has 148 nodes and 10885 edges
```

We can also print a summary of the hypergraph:

[4]:
```
print(H_enron)
```

```
Hypergraph named email-Enron with 148 nodes and 10885 hyperedges
```

The dataset is completely formatted. You can access nodes and edges or their attributes in a very simple way:

[5]:
```
print("The first 10 node IDs are:")
print(list(H_enron.nodes)[:10])
print("The first 10 edge IDs are:")
print(list(H_enron.edges)[:10])
print("The attributes of node '4' are")
print(H_enron.nodes["4"])
print("The attributes of edge '6' are")
print(H_enron.edges["6"])
```

```
The first 10 node IDs are:
['4', '1', '117', '129', '51', '41', '65', '107', '122', '29']
The first 10 edge IDs are:
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
The attributes of node '4' are
{'name': 'robert.badeer@enron.com'}
```

(continues on next page)

(continued from previous page)

```
The attributes of edge '6' are
{'timestamp': '2000-02-22T08:07:00'}
```

It is also possible to access nodes of edges in particular variable types, for example we can create a dictionary containing the edges of our hypergraph and their members:

[6]:
```
edges_dictionary = H_enron.edges.members(dtype=dict)
print(list(edges_dictionary.items())[:5])
```

```
[('0', {'4', '1'}), ('1', {'129', '1', '117'}), ('2', {'1', '51'}), ('3', {'1', '51'}), (
↪'4', {'1', '41'})]
```

Cleaning up a hypergraph dataset

You can check if your hypergraph is connected using the function:

[7]:
```
xgi.is_connected(H_enron)
```

[7]:
```
False
```

We can count the number of isolated nodes and multi-edges in the following way:

[8]:
```
isolated_nodes = H_enron.nodes.isolates()
print("Number of isolated nodes: ", len(isolated_nodes))
duplicated_edges = H_enron.edges.duplicates()
print("Number of duplicated edges: ", len(duplicated_edges))
```

```
Number of isolated nodes:  5
Number of duplicated edges:  9371
```

We can clean up this dataset to remove isolated nodes and multi-edges, and replace all IDs with integer IDs using the `cleanup` function:

[9]:
```
H_enron_cleaned = H_enron.cleanup(
    multiedges=False, singletons=False, isolates=False, relabel=True, in_place=False
)

print(H_enron_cleaned)
```

```
Hypergraph named email-Enron with 143 nodes and 1459 hyperedges
```

We can see that 5 isolated nodes were removed and 9371 duplicated edges were removed. We can check it:

[10]:
```
len(H_enron.nodes) == len(H_enron_cleaned.nodes) + len(isolated_nodes)
```

[10]:
```
True
```

[11]:
```
len(H_enron.edges) == len(H_enron_cleaned.edges) + len(duplicated_edges)
```

[11]:
```
False
```

We can check that the hypergraph is now connected:

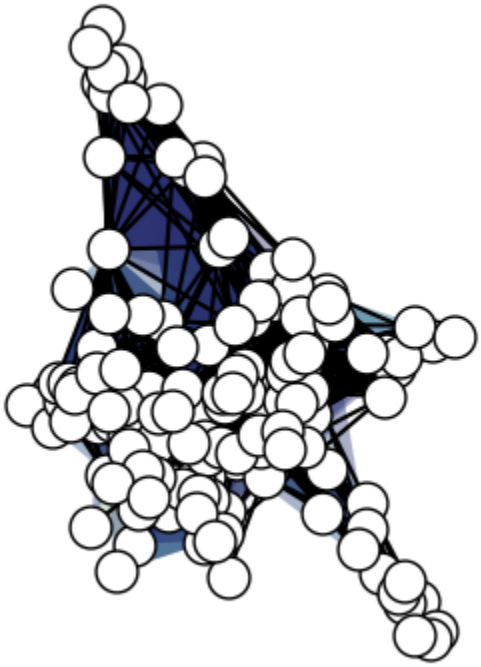
[12]:
```
xgi.is_connected(H_enron_cleaned)
```

[12]:
```
True
```

Drawing

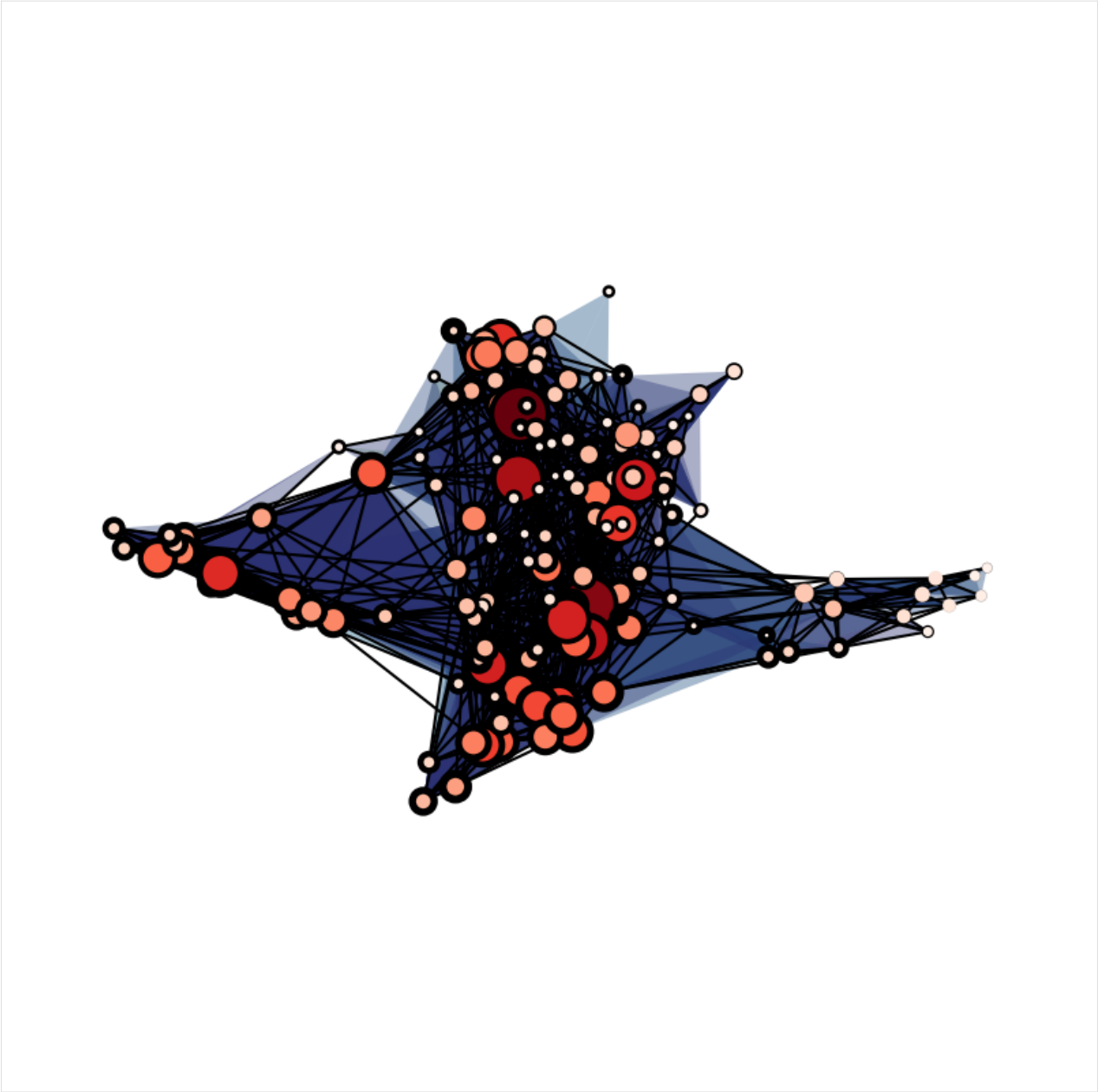
Visualization is crucial for understanding complex data structures. You can use the default drawing function:

[13]:
```
xgi.draw(H_enron_cleaned);
```



When dealing with large structures like this e-mail dataset the visualization can be cumberstome to interpret. To help you with that XGI provides options for plotting hypergraph using the features of nodes and edges, for example:

[14]:
```
fig, ax = plt.subplots(figsize=(10, 10))
xgi.draw(
    H_enron_cleaned,
    node_size=H_enron_cleaned.nodes.degree,
    node_lw=H_enron_cleaned.nodes.average_neighbor_degree,
    node_fc=H_enron_cleaned.nodes.degree,
    ax=ax,
);
```

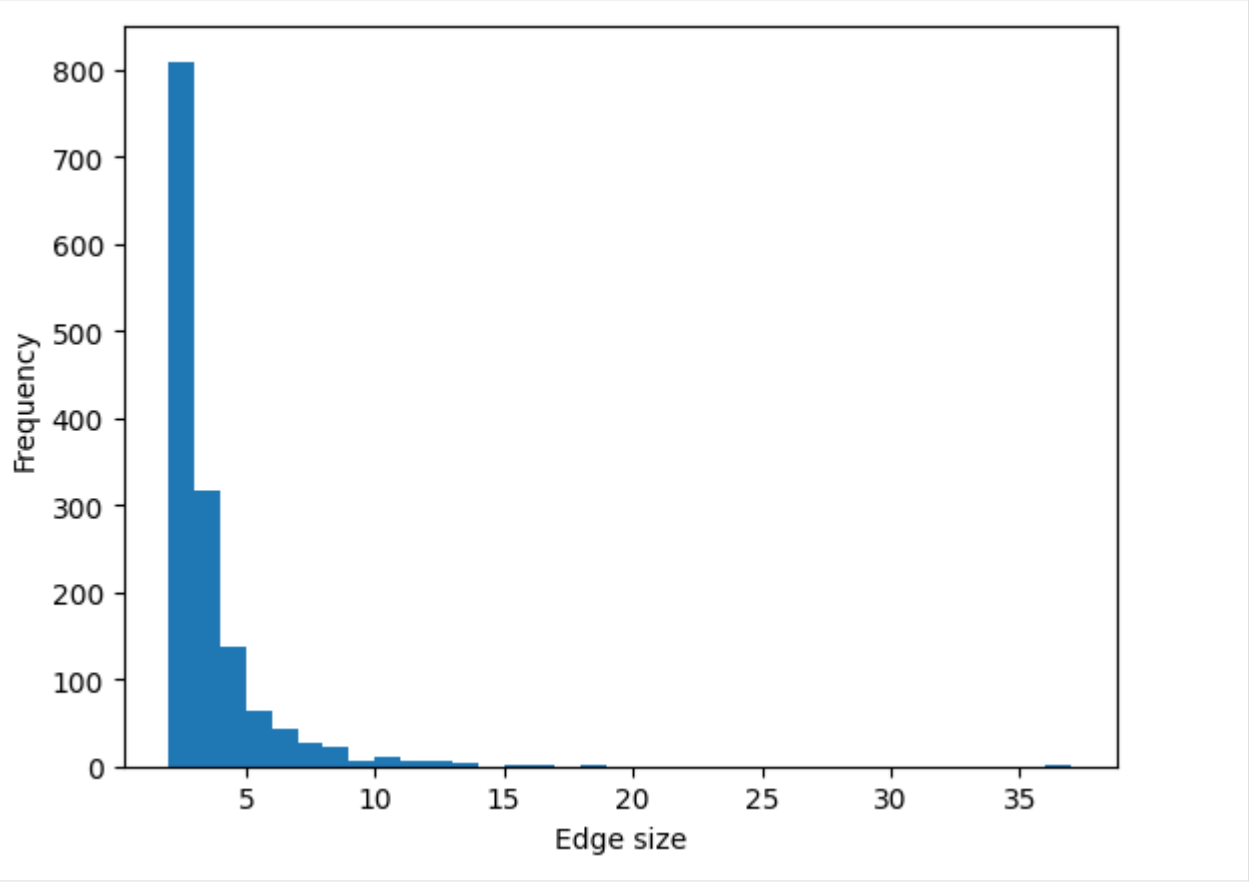


In this case we are plotting the hypergraph with the size and color of nodes depending on their degrees and the width of the edges nodes markers depending on their average neighbor degree.

Histograms of edges sizes and nodes' degrees

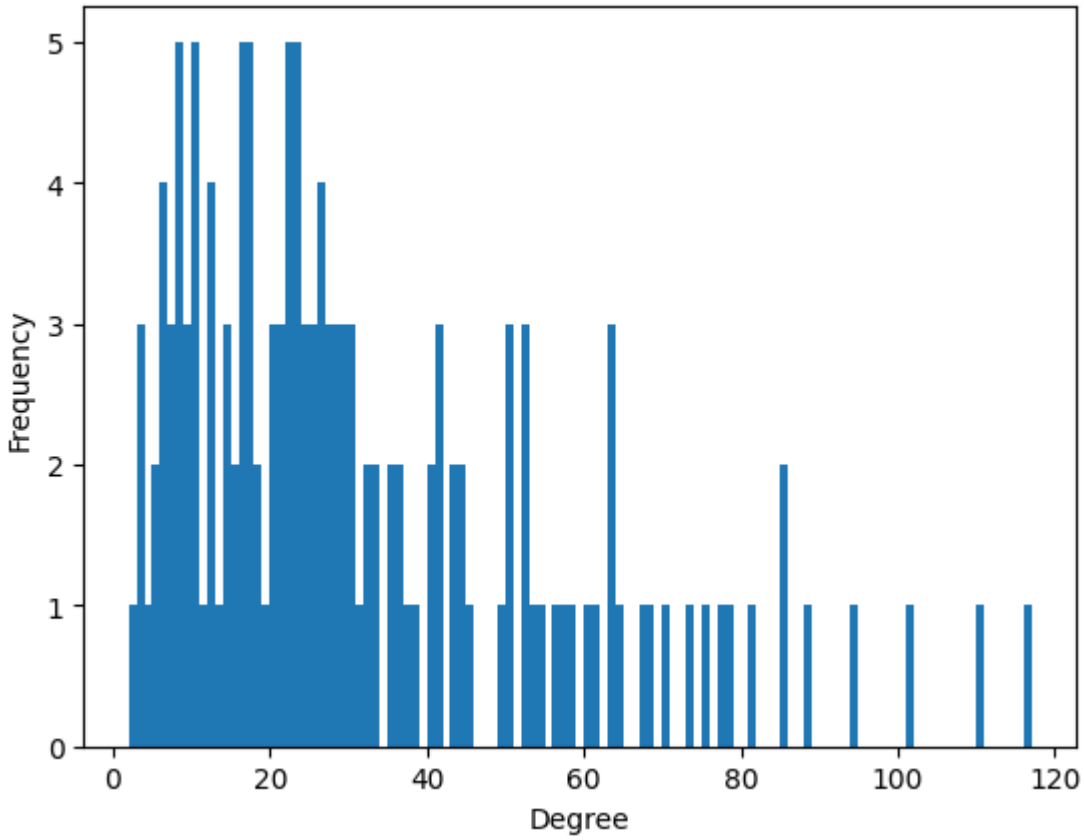
It might me useful for a first analysis of you dataset to plot some histrograms representing relevant feautures of you higher-order structure. For example if you want to plot a histogram for the edges sizes:

[15]:
```
list_of_edges_sizes = H_enron_cleaned.edges.size.aslist()
ax = plt.subplot(111)
ax.hist(
    list_of_edges_sizes,
    bins=range(min(list_of_edges_sizes), max(list_of_edges_sizes) + 1, 1),
)
ax.set_xlabel("Edge size")
ax.set_ylabel("Frequency");
```



Or you can plot a histogram for the nodes' degrees (the degree of a node is the number of edges it belongs to):

[16]:
```
list_of_nodes_degrees = H_enron_cleaned.nodes.degree.aslist()
ax = plt.subplot(111)
ax.hist(
    list_of_nodes_degrees,
    bins=range(min(list_of_nodes_degrees), max(list_of_nodes_degrees) + 1, 1),
)
ax.set_xlabel("Degree")
ax.set_ylabel("Frequency");
```



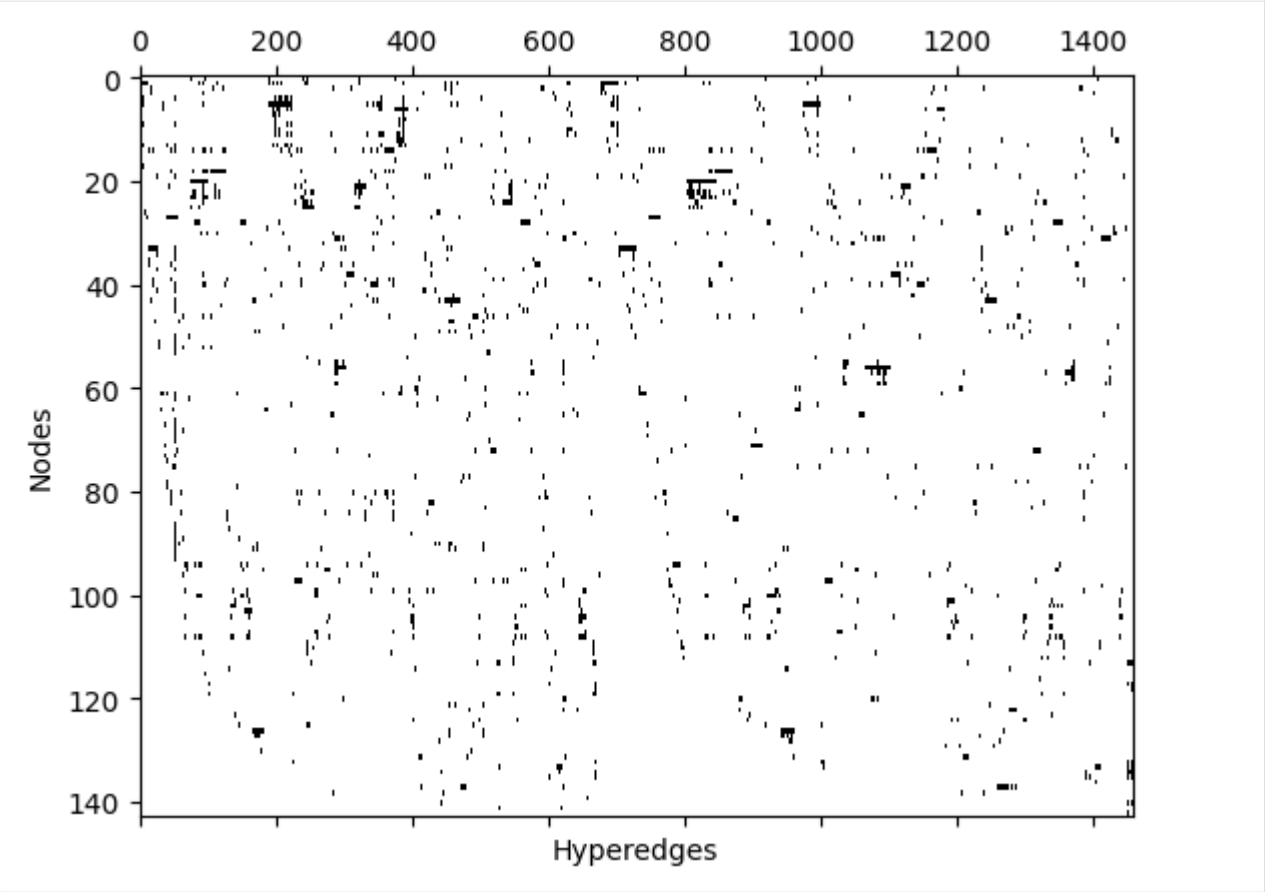
Incidence and Adjacency Matrices

Any hypergraph can be expressed as an $N \times M$ incidence matrix, I, where N is the number of nodes and M is the number of edges. Rows indicate the node ID and the columns indicate the edge ID. $I_{i,j} = 1$ if node i is a member of edge j and zero otherwise. XGI allows you to access the incidence matrix in the following way:

[17]:
```
I = xgi.incidence_matrix(H_enron_cleaned, sparse=False)
```

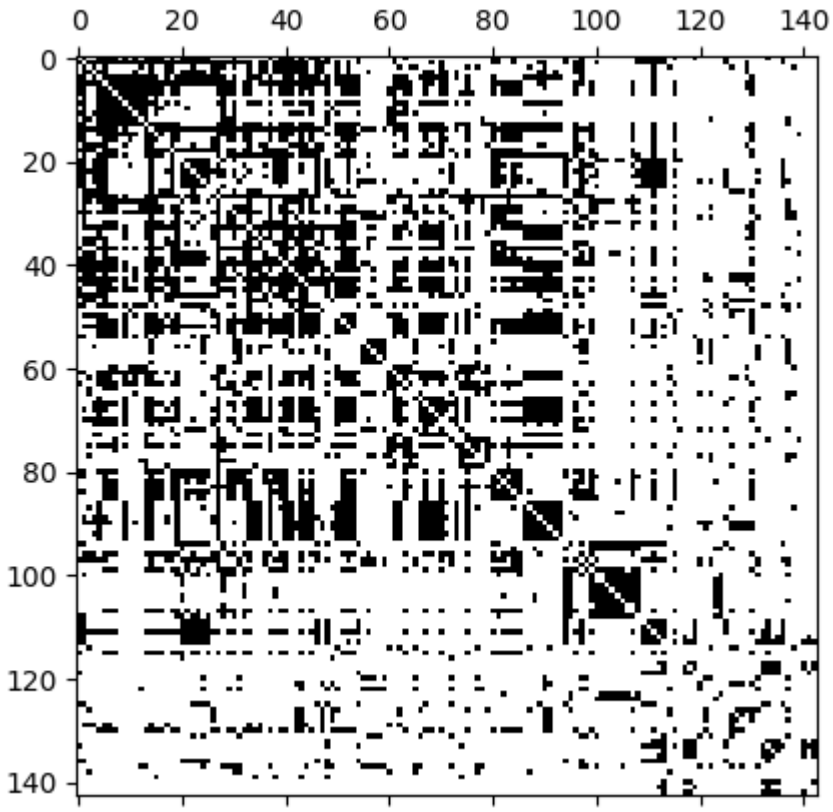
Then you can visualize it:

[18]:
```
plt.spy(I, aspect="auto")
plt.xlabel("Hyperedges")
plt.ylabel("Nodes")
plt.show()
```



We can represent a hypergraph with an $N \times N$ adjacency matrix, A, where N is the number of nodes. Notice that the adjacency matrix is a lossy format: different hypergraphs can create the same adjacency matrix. $A_{i,j} = 1$ if there is at least one hyperedge containing both nodes i and j. XGI allows you to access the incidence matrix and visualize it in the following way:

[19]:
```
A = xgi.adjacency_matrix(H_enron_cleaned, sparse=False)
plt.spy(A);
```



If you are interested in other hypergraph matrices such as Laplacians, you can check the documentatation about the linear algebra package.

Algorithms

The algorithms package contains different algorithms you can run on your higher-order structure. For example you can compute the density and degree assortativity of your structure:

[20]:
```
print("The density of the hypergraph is:", xgi.density(H_enron_cleaned))
print(
    "The assortativity of the hypergraph is:", xgi.degree_assortativity(H_enron_cleaned)
)
```

```
The density of the hypergraph is: 1.3084764540479412e-40
The assortativity of the hypergraph is: 0.2250663686125537
```

Or you can access a dictionary containing the local clustering coefficient (overlap of the edges connected to a given node, normalized by the size of the node's neighborhood, for more details you can see this paper) of your structures:

[21]:
```
local_clustering_dict = xgi.local_clustering_coefficient(H_enron_cleaned)
print(local_clustering_dict)
```

```
{0: 0.7804612787100978, 1: 0.6780441301803274, 2: 0.7689183265963335, 3: 0.
↪791038445681303, 4: 0.7856925737994167, 5: 0.6965494874182317, 6: 0.7072322879888286,␣
↪7: 0.7730900667477023, 8: 0.7118552061973111, 9: 0.7683540442301209, 10: 0.
↪8031870323887131, 11: 0.76833559781224, 12: 0.7490487746981251, 13: 0.7755923252981957,
```

(continues on next page)

(continued from previous page)

```
↪ 14: 0.6848194711637989, 15: 0.7573675775256802, 16: 0.4584199134199135, 17: 0.
↪638446173924115, 18: 0.7017206124203775, 19: 0.8031870323887131, 20: 0.
↪7009358405995354, 21: 0.738118399364662, 22: 0.7075910814158463, 23: 0.
↪7223895333094579, 24: 0.739291988588228, 25: 0.7710850845455389, 26: 0.
↪7821012321012321, 27: 0.768337509507563, 28: 0.7700134952824707, 29: 0.
↪7490487746981251, 30: 0.7856925737994167, 31: 0.7659625084060587, 32: 0.
↪7786708147819262, 33: 0.7038081247195713, 34: 0.7727459637785723, 35: 0.
↪7573675775256802, 36: 0.7861972454329856, 37: 0.7797302518270263, 38: 0.
↪7759247429139076, 39: 0.7659625084060587, 40: 0.7009358405995354, 41: 0.
↪7757340067340067, 42: 0.7573675775256802, 43: 0.6975586879901238, 44: 0.
↪7573675775256802, 45: 0.7757340067340067, 46: 0.7759756479670126, 47: 0.7288875146018,␣
↪48: 0.7812697583387238, 49: 0.7727459637785723, 50: 0.6644212904016823, 51: 0.
↪5541149591149591, 52: 0.49949772449772456, 53: 0.49949772449772456, 54: 0.
↪638446173924115, 55: 0.7683540442301209, 56: 0.7104077955538087, 57: 0.
↪7725113110513565, 58: 0.7118552061973111, 59: 0.7887064044826144, 60: 0.
↪7812697583387238, 61: 0.7804612787100978, 62: 0.42930819180819185, 63: 0.
↪7727459637785723, 64: 0.7757340067340067, 65: 0.7866768592959072, 66: 0.
↪4117462894248609, 67: 0.4727321920503739, 68: 0.4584199134199135, 69: 0.
↪4258333333333334, 70: 0.5541149591149591, 71: 0.7821012321012321, 72: 0.
↪7725113110513565, 73: 0.3619137806637807, 74: 0.4584199134199135, 75: 0.
↪7887064044826144, 76: 0.0, 77: 0.638446173924115, 78: 0.4727321920503739, 79: 0.
↪4798340548340549, 80: 0.782342244842245, 81: 0.7821012321012321, 82: 0.
↪7894392381608293, 83: 0.4727321920503739, 84: 0.7786708147819262, 85: 0.
↪638446173924115, 86: 0.0, 87: 0.4584199134199135, 88: 0.4798340548340549, 89: 0.
↪3619137806637807, 90: 0.7786708147819262, 91: 0.7490487746981251, 92: 0.
↪4258333333333334, 93: 0.0, 94: 0.7827884699639654, 95: 0.7680722410445265, 96: 0.
↪7490487746981251, 97: 0.7690633411984246, 98: 0.6644212904016823, 99: 0.
↪7680722410445265, 100: 0.7856925737994167, 101: 0.7683540442301209, 102: 0.
↪7725113110513565, 103: 0.7894392381608293, 104: 0.7659625084060587, 105: 0.
↪7866768592959072, 106: 0.7490487746981251, 107: 0.7752549928556385, 108: 0.
↪7035118435926735, 109: 0.7288875146018, 110: 0.7288875146018, 111: 0.7866768592959072,␣
↪112: 0.5978351972101971, 113: 0.7812697583387238, 114: 0.4584199134199135, 115: 0.
↪4117462894248609, 116: 0.0, 117: 0.4727321920503739, 118: 0.4798340548340549, 119: 0.
↪5978351972101971, 120: 0.7730900667477023, 121: 0.3619137806637807, 122: 0.
↪7730900667477023, 123: 0.7118552061973111, 124: 0.7573675775256802, 125: 0.
↪5978351972101971, 126: 0.7752549928556385, 127: 0.5978351972101971, 128: 0.
↪4117462894248609, 129: 0.502430145611964, 130: 0.4117462894248609, 131: 0.
↪685872720521843, 132: 0.638446173924115, 133: 0.7821012321012321, 134: 0.
↪49949772449772456, 135: 0.5978351972101971, 136: 0.40510101010101013, 137: 0.
↪791038445681303, 138: 0.4117462894248609, 139: 0.40510101010101013, 140: 0.
↪16666666666666666, 141: 0.40510101010101013, 142: 0.40510101010101013}
```

Stats

The stats package is one of the features that sets XGI apart from other libraries. It provides a common interface to all statistics that can be computed from a network, its nodes, or edges. This package allows you, for example, to filter the nodes of a hypergraph with a certain degree:

[22]:
```
nodes_degree_2 = H_enron_cleaned.nodes.filterby("degree", 20)
print(nodes_degree_2)
```

```
[8, 58, 123]
```

Or you can perform more complex tasks such as creating a dataframe containing different statistics:

[23]:
```
df = H_enron_cleaned.nodes.multi(["degree", "clustering_coefficient"]).aspandas()
print(df)
```

```
     degree  clustering_coefficient
0        44                0.548792
1       101                0.452685
2        57                0.529268
3        36                0.606272
4        50                0.569712
..      ...                     ...
138       8                0.535714
139       6                0.333333
140       4                1.000000
141       6                1.000000
142       6                1.000000

[143 rows x 2 columns]
```

You can learn more about the stats package with the focus tutorial on statics or checking the documentation.

Wrapping Up

Well done! You've covered a lot in just 15 minutes with XGI. We hope you enjoyed this tutorial, and there's much more to explore! Check out other tutorials here!

11.2 Focus tutorials

11.2.1 Basic hypergraph functionality

This tutorial will give a brief introduction to using the XGI library to construct hypergraphs and perform basic operations on them.

[13]:
```
import random

import numpy as np

import xgi
```

Loading hypergraphs from different formats

We handle loading hypergraphs in many different formats, but the hypergraph constructor takes five main data formats: * A Hypergraph object * A hyperedge list * A hyperedge dictionary * A 2-column pandas dataframe specifying (node, edge) bipartite edges * An incidence matrix (A Numpy or Scipy matrix)

[14]:
```
n = 1000
m = 1000

min_edge_size = 2
max_edge_size = 25

# hyperedge list
hyperedge_list = [
    random.sample(range(n), random.choice(range(min_edge_size, max_edge_size +␣
↪1)))
    for i in range(m)
]

# hyperedge dict
hyperedge_dict = {
    i: random.sample(range(n), random.choice(range(min_edge_size, max_edge_
↪size + 1)))
    for i in range(m)
}

# pandas dataframe
df = xgi.to_bipartite_pandas_dataframe(xgi.load_xgi_data("email-enron"))

# incidence matrix
incidence_matrix = np.random.randint(0, high=2, size=(n, m), dtype=int)
```

Loading a hyperedge list

When a user gives a hyperedge list, the system automatically creates system edge IDs.

[15]:
```
H = xgi.Hypergraph(hyperedge_list)
print(f"The hypergraph has {H.num_nodes} nodes and {H.num_edges} edges")
```

```
The hypergraph has 1000 nodes and 1000 edges
```

Loading a hyperedge dictionary

When a user gives a hyperedge dictionary, the system uses the edge IDs specified in the dictionary.

[16]:
```
H = xgi.Hypergraph(hyperedge_dict)
print(f"The hypergraph has {H.num_nodes} nodes and {H.num_edges} edges")
```

```
The hypergraph has 1000 nodes and 1000 edges
```

Loading an incidence matrix

When a user gives an incidence matrix, the system transforms the non-zero entries into lists of rows and columns specifying a bipartite edge list.

[17]:
```
H = xgi.Hypergraph(incidence_matrix)
print(f"The hypergraph has {H.num_nodes} nodes and {H.num_edges} edges")
```

```
The hypergraph has 1000 nodes and 1000 edges
```

Loading a Pandas dataframe

When a user gives a Pandas dataframe, the system automatically imports the first two columns as lists of node and edge indices specifying a bipartite edge list.

[18]:
```
H = xgi.Hypergraph(df)
print(f"The hypergraph has {H.num_nodes} nodes and {H.num_edges} edges")
```

```
The hypergraph has 143 nodes and 10885 edges
```

Simple functions

The Hypergraph class can do simple things like * output an incidence matrix * output the adjacency matrix for s-connectedness * output the dual of the hypergraph * find if the hypergraph is connected

Output relevant matrices

[19]:
```
# The incidence matrix
I = xgi.incidence_matrix(H, sparse=True)
# The adjacency matrix
A = xgi.adjacency_matrix(H)
# The clique motif matrix
W = xgi.clique_motif_matrix(H)
```

Forming the dual

[20]:
```
D = H.dual()
```

Testing whether the hypergraph is connected

[21]:
```
n = 1000
m = 100

min_edge_size = 2
max_edge_size = 10

# hyperedge list
```

(continues on next page)

(continued from previous page)

```
hyperedge_list = [
    random.sample(range(n), random.choice(range(min_edge_size, max_edge_size +␣
↪1)))
    for i in range(m)
]
H = xgi.Hypergraph(hyperedge_list)
```

[22]:
```
is_connected = xgi.is_connected(H)
if is_connected:
    print(f"H is connected")
else:
    print(f"H is not connected")

print(f"The sizes of the connected components are:")
print([len(component) for component in xgi.connected_components(H)])

node = np.random.choice(H.nodes)
print(
    f"The size of the component containing node {node} is {len(xgi.node_
↪connected_component(H, node))}"
)
```

```
H is not connected
The sizes of the connected components are:
[438, 4, 3, 3, 3, 3, 2, 4, 4, 4]
The size of the component containing node 713 is 438
```

Constructing subhypergraphs

A subhypergraph can be induced by a node subset, an edge subset, or an arbitrary combination of both. These examples are presented below.

[23]:
```
# A subhypergraph induced on nodes
node_subhypergraph = xgi.subhypergraph(H, nodes=list(range(100)))
# A subhypergraph induced on edges
edge_subhypergraph = xgi.subhypergraph(H, edges=list(range(100)))
# A subhypergraph induced on both nodes and edges
arbitrary_subhypergraph = xgi.subhypergraph(
    H, nodes=list(range(100)), edges=list(range(100))
)
```

Converting to other formats

Below are examples showing how to convert a hypergraph to a hyperedge list, a hyperedge dict, or an incidence matrix.

[24]:
```
# Convert to a hyperedge list
h_list = xgi.to_hyperedge_list(H)
# Convert to a hyperedge dict
h_dict = xgi.to_hyperedge_dict(H)
# Convert to an incidence matrix
h_I = xgi.to_incidence_matrix(H)
```

[ ]:
```

```

11.2.2 Read and Write

[ ]:
```
import random

import xgi
```

Importing and exporting hypergraph data

When working with empirical hypergraph data, the following file formats representing hypergraphs are commonly seen in practice: * A hyperedge list, where each line represents a hyperedge * A bipartite edge list where each line contains two entries: a node ID and an edge ID * An incidence matrix

The `readwrite` module provides functionality to import and export these file formats.

Example hypergraph

[2]:
```
n = 10
m = 10

min_edge_size = 2
max_edge_size = 10

# hyperedge list
hyperedge_list = [
    random.sample(range(n), random.choice(range(min_edge_size, max_edge_size +␣
↪1)))
    for i in range(m)
]
H = xgi.Hypergraph(hyperedge_list)
```

JSON

These functions import and export the hypergraph to a standardized JSON format.

[3]:
```
# Write the example hypergraph to a JSON file
xgi.write_json(H, "hypergraph_json.json")
# Load the file just written and store it in a new hypergraph
H_json = xgi.read_json("hypergraph_json.json")
```

Edge List

These functions import and export the hypergraph as an edge list, with user specified delimiters.

[4]:
```
# Write the hypergraph to a file as a hyperedge list
xgi.write_edgelist(H, "hyperedge_list.csv", delimiter=",")
# Read the file just written as a new hypergraph
H_el = xgi.read_edgelist("hyperedge_list.csv", delimiter=",", nodetype=int)
```

Bipartite Edge List

These functions import and export the hypergraph as a bipartite edge list with user-specified delimiters.

[5]:
```
# Write the hypergraph as a bipartite edge list
xgi.write_bipartite_edgelist(H, "bipartite_edge_list.csv", delimiter=",")
# Read the file just written as a new hypergraph
H_bel = xgi.read_bipartite_edgelist(
    "bipartite_edge_list.csv", delimiter=",", nodetype=int
)
```

11.2.3 Basic simplicial complex usage

[1]:
```
import random

import xgi
```

11.2.4 Creating a Simplicial Complex

Simplicial complexes can be created directly using the `SimplicialComplex` class. It inherits a number of properties from the `Hypergraph` one, but it has also some significant differences due to the closure property required for valid simplicial complexes.

Let us illustrate these differences by comparing directly the construction of a simplicial complex and of a hypergraph containing the same hyperedges/simplices. We begin by creating a simplicial complex `SC` and a hypergraph `H`.

[2]:
```
SC = xgi.SimplicialComplex()
H = xgi.Hypergraph()
```

We add to both of them 5 nodes

[3]:
```
SC.add_nodes_from(range(5))
H.add_nodes_from(range(5))
```

[4]:
```
SC.nodes, H.nodes
```

[4]:
```
(NodeView((0, 1, 2, 3, 4)), NodeView((0, 1, 2, 3, 4)))
```

We then add the simplex `[0,1,2]` to the simplicial complex and the same (as hyperedge) to the hypergraph

[5]:
```
SC.add_simplex([0, 1, 2])
H.add_edge([0, 1, 2])
```

Note that if we try to use `add_edge` on the simplicial complex, it will not work. This is done in order to prevent the misuse of hyperedge addition that would not protect the closure property of simplicial complexes.

[6]:
```
# SC.add_edge([0,1,2])      # try uncommenting and executing this cell
```

Now, what is this closure property we have mentioned?

By definition, a simplicial complex K is valid if for any simplex $\sigma \in K$, all faces $\omega \subset \sigma$ are also $\omega \in K$. This implies that we must make sure that each face is properly added whenever we add a new simplex. The function `SC.add_simplex` and `SC.add_simplices_from` already take care of this directly. This does not happen instead when adding hyperedges to hypergraphs.

We can check this easily by asking how many edges are present in the simplicial complex versus the hypergraph. Previously, we only added the simplex/hyperedge `[0,1,2,3]`. The closure requires also the addition of the three edges (`[0,1]`, `[0,2]`, `[1,2]`), so there should be 4 simplices in `SC` and only one in `H`. Correctly, we find:

[7]:
```
SC.edges
```

[7]:
```
EdgeView((0, 1, 2, 3))
```

[8]:
```
H.edges
```

[8]:
```
EdgeView((0,))
```

We can further check that this is correct by checking how many hyperedges of a certain dimension ("order") are present in the simplicial complex. We can do this using the `order` key as follows:

[9]:
```
for dim in [0, 1, 2]:
    print("# edges in H at order", dim, H.edges.filterby("order", dim))
    print("# edges in SC at order", dim, SC.edges.filterby("order", dim), "\n")
```

```
# edges in H at order 0 []
# edges in SC at order 0 []

# edges in H at order 1 []
# edges in SC at order 1 [1, 2, 3]

# edges in H at order 2 [0]
# edges in SC at order 2 [0]
```

Finally, we can directly access the simplex composition by using the `members` function of `edges`.

[10]:
```
for dim in [0, 1, 2]:
    edge_sel = list(SC.edges.filterby("order", dim))
    print(
        "Composition of simplices at order",
        dim,
        [SC.edges.members(x) for x in edge_sel],
        "\n",
    )
```

```
Composition of simplices at order 0 []

Composition of simplices at order 1 [frozenset({0, 1}), frozenset({0, 2}),␣
↪frozenset({1, 2})]

Composition of simplices at order 2 [frozenset({0, 1, 2})]
```

As opposed to what happens in the corresponding hypergraph

[11]:
```
for dim in [0, 1, 2]:
    edge_sel = list(H.edges.filterby("order", dim))
    print(
        "Composition of edges at order",
        dim,
        [H.edges.members(x) for x in edge_sel],
        "\n",
    )
```

```
Composition of edges at order 0 []

Composition of edges at order 1 []

Composition of edges at order 2 [{0, 1, 2}]
```

Loading from a list of facets

We can also provide the `SimplicialComplex` class with a list of facets –maximal faces, that is, simplices that are not contained in any other simplices–. Note that if the list contains a simplex that it's not a facet, this will not throw an error but the simplex will simply become part of the closure of a larger one in the list.

[12]:
```
facets = [[0, 1, 2], [0, 1, 3, 4]]
SC1 = xgi.SimplicialComplex(facets)
```

[13]:
```
SC1.edges.members()
```

[13]:
```
[frozenset({0, 1, 2}),
 frozenset({0, 1, 3, 4}),
 frozenset({0, 1}),
 frozenset({1, 2}),
 frozenset({0, 1, 4}),
 frozenset({0, 4}),
```

(continues on next page)

(continued from previous page)

```
 frozenset({3, 4}),
 frozenset({0, 3, 4}),
 frozenset({0, 1, 3}),
 frozenset({0, 3}),
 frozenset({1, 3, 4}),
 frozenset({1, 4}),
 frozenset({0, 2}),
 frozenset({1, 3})]
```

[14]:
```
n = 20
m = 40

min_edge_size = 2
max_edge_size = 4

hyperedge_list = [
    random.sample(range(n), random.choice(range(min_edge_size, max_edge_size +␣
↪1)))
    for i in range(m)
]
SC_big = xgi.SimplicialComplex(hyperedge_list)
str(SC_big)
```

[14]:
```
'Unnamed SimplicialComplex with 20 nodes and 188 simplices'
```

Loading from a pandas DataFrame

[15]:
```
df = xgi.to_bipartite_pandas_dataframe(xgi.load_xgi_data("email-enron"))
```

When a user gives a Pandas dataframe, the system automatically imports the first two columns as lists of node and edge indices specifying a bipartite edge list. Below we compare the results of importing the dataframe as a hypergraph, in which all edges are retained just as they are in the dataframe, and as a simplicial complex, in which we cannot have repeated simplices.

[16]:
```
H = xgi.Hypergraph(df[:300])
str(H), H.edges.members()[:30]
```

[16]:
```
('Unnamed Hypergraph with 2 nodes and 285 hyperedges',
 [{'4'},
  {'4'},
  {'1', '4'},
  {'4'},
  {'4'},
  {'4'},
  {'4'},
  {'4'},
  {'4'},
  {'4'},
  {'4'},
  {'4'},
  {'4'},
  {'4'},
```

(continues on next page)

(continued from previous page)

```
 {'4'},
 {'4'},
 {'4'},
 {'4'},
 {'4'},
 {'4'},
 {'4'},
 {'4'},
 {'4'},
 {'4'},
 {'4'},
 {'4'},
 {'1', '4'},
 {'4'},
 {'1', '4'},
 {'1', '4'},
 {'4'}])
```

[17]:
```
F = xgi.SimplicialComplex(df[:300])
print(f"The hypergraph has {F.num_nodes} nodes and {F.num_edges} edges")
F.edges.members()[:30]
```

```
The hypergraph has 2 nodes and 3 edges
```

[17]:
```
[frozenset({'4'}), frozenset({'1', '4'}), frozenset({'1'})]
```

In many cases we start from a list of facets that can include very large simplices. This however can become quickly very cumbersome because the number of subfaces scales factorially with the order of the facet. Moreover, even when we have large facets, we might be interested in quantities that are defined only on simplices of smaller dimension. One way to approach this is by reducing the maximum size of the simplices that are added to the`SimplicialComplex` constructor. Alternatively, it is possible to provide to the `add_simplices_from` function a maximum order. In that case, any simplex with order larger than `max_order` will be substituted with its faces of order `max_order` (the closure property is automatically enforced by the function).

[18]:
```
X = xgi.SimplicialComplex()
X.add_simplices_from(hyperedge_list, max_order=4)
str(X)
```

[18]:
```
'Unnamed SimplicialComplex with 20 nodes and 188 simplices'
```

and we can also check that this is the case explicitly closing the simplicial complex and checking that the size does not change.

[19]:
```
X.close()
str(X)
```

[19]:
```
'Unnamed SimplicialComplex with 20 nodes and 188 simplices'
```

11.2.5 Generative Models

The `generators` module provides functionality to generate common models of hypergraphs, both non-uniform and uniform.

[1]:
```
import random

import numpy as np

import xgi
```

Hypergraph generative models

Uniform configuration model

[2]:
```
n = 1000
m = 3
k = {i: random.randint(10, 30) for i in range(n)}
H = xgi.uniform_hypergraph_configuration_model(k, m)
```

```
/Users/lordgrilo/Dropbox (ISI Foundation)/development/xgi-local/xgi-repo/xgi/
↪generators/uniform.py:61: UserWarning: This degree sequence is not␣
↪realizable. Increasing the degree of random nodes so that it is.
  warnings.warn(
```

Erdős–Rényi model

[3]:
```
n = 1000
ps = [0.01, 0.001]
H = xgi.random_hypergraph(n, ps)
```

Non-uniform configuration model

[4]:
```
n = 1000
k1 = {i: random.randint(10, 30) for i in range(n)}
k2 = {i: sorted(k1.values())[i] for i in range(n)}
H = xgi.chung_lu_hypergraph(k1, k2)
```

Non-uniform DCSBM hypergraph

[5]:
```
n = 1000
k1 = {i: random.randint(1, 100) for i in range(n)}
k2 = {i: sorted(k1.values())[i] for i in range(n)}
g1 = {i: random.choice([0, 1]) for i in range(n)}
g2 = {i: random.choice([0, 1]) for i in range(n)}
omega = np.array([[100, 10], [10, 100]])
H = xgi.dcsbm_hypergraph(k1, k2, g1, g2, omega)
```

```
/Users/lordgrilo/Dropbox (ISI Foundation)/development/xgi-local/xgi-repo/xgi/
↪generators/nonuniform.py:186: UserWarning: The sum of the degree sequence␣
↪does not match the entries in the omega matrix
  warnings.warn(
```

Simplicial Complex Generative Models

Random simplicial complex model

(from Iacopini et al. 2019)

Given n nodes and a vector of probabilities $\vec{p} = [p_1, p_2, \ldots, p_d]$, where d is the maximal simplex dimension desired, the model creates simplices at each dimension with the corresponding probability (p_1 for edges, p_2 for 2-simplices, etc).

[6]:
```
n = 20
ps = [0.1, 0.2, 0.1]
SC = xgi.random_simplicial_complex(n, ps)
```

Random flag complex model in 2D

The model creates an Erdos-Renyi network with n nodes and probability p for any pair of edges. It then promotes all 3-cliques to 2-simplices.

[7]:
```
n = 50
p = 0.1
SC = xgi.random_flag_complex_d2(n, p)
```

Generalized random flag complex model

[8]:
```
n = 30
p = 0.2
SC = xgi.random_flag_complex(n, p, max_order=3)
```

Flag complex from graph

It is also possible to construct flag (clique) complexes starting from an existing network. Of course, let's show this using Zachary's Karate club and including simplices up to te

[11]:
```
import networkx as nx

G = nx.karate_club_graph()

SC_KC = xgi.flag_complex(G, max_order=3)
```

11.2.6 Visualizing hypergraphs

Visualizing hypergraphs, just like pairwise networks, is a hard task and no algorithm can work nicely for any given input structure. Here, we show how to visualize some toy structures using the visualization function contained in the `drawing` module that is often inspired by networkx and relies on matplotlib.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np

import xgi
```

Basics

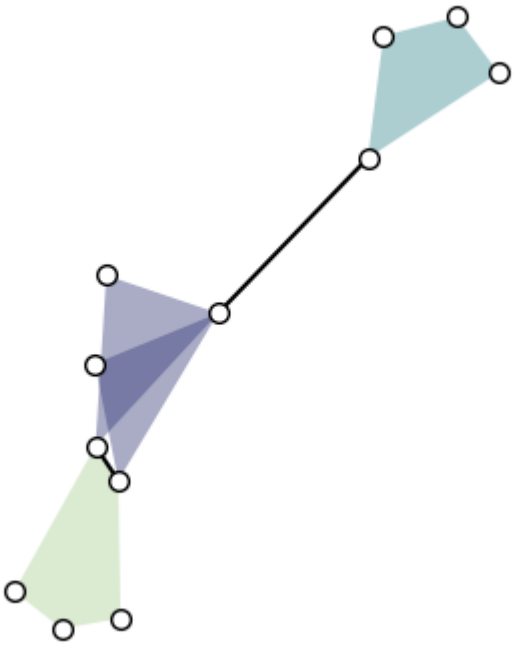
Les us first create a small toy hypergraph containing edges of different sizes.

[2]:
```
H = xgi.Hypergraph()
H.add_edges_from(
    [[1, 2, 3], [3, 4, 5], [3, 6], [6, 7, 8, 9], [1, 4, 10, 11, 12], [1, 4]]
)
```

It can be quickly visualized simply with

[3]:
```
xgi.draw(H)
```

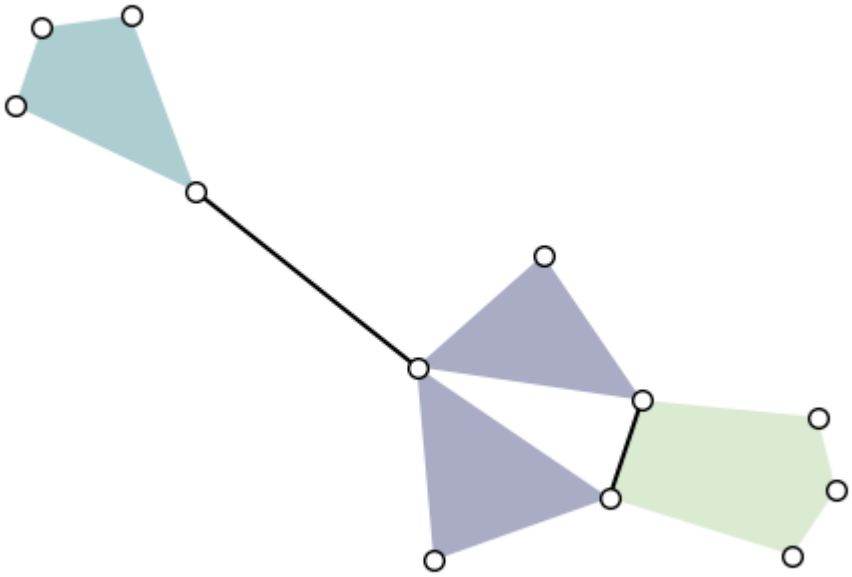
[3]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x17f5c7b20>,
  <matplotlib.collections.LineCollection at 0x17f5e8820>,
  <matplotlib.collections.PatchCollection at 0x17f5c7f70>))
```



By default, the **layout** is computed by `xgi.barycenter_spring_layout`. For a bit more control, we can compute a layout externally (that we fix with a random seed), so that we can reuse it:

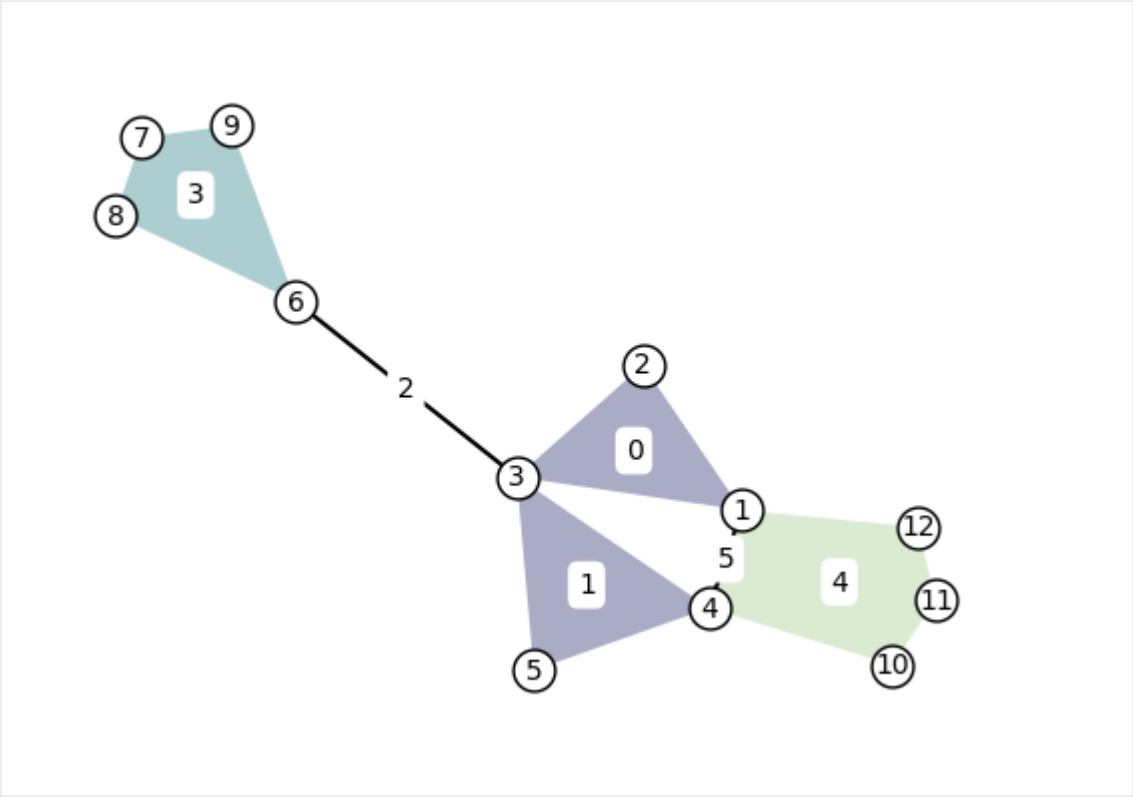
[4]:
```
pos = xgi.barycenter_spring_layout(H, seed=1)

fig, ax = plt.subplots()
xgi.draw(H, pos=pos, ax=ax)
```

[4]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x17f81d9f0>,
  <matplotlib.collections.LineCollection at 0x17f88ad70>,
  <matplotlib.collections.PatchCollection at 0x17f8c71f0>))
```



Labels can be added to the nodes and hyperedges with arguments `node_labels` and `hyperedge_labels`. If `True`, the IDs are shown. To display user-defined labels, pass a dictionary that contains (id: label). Additional keywords related to the font can be passed as well:

[5]:
```
xgi.draw(H, pos, node_labels=True, node_size=15, hyperedge_labels=True)
plt.show()
```

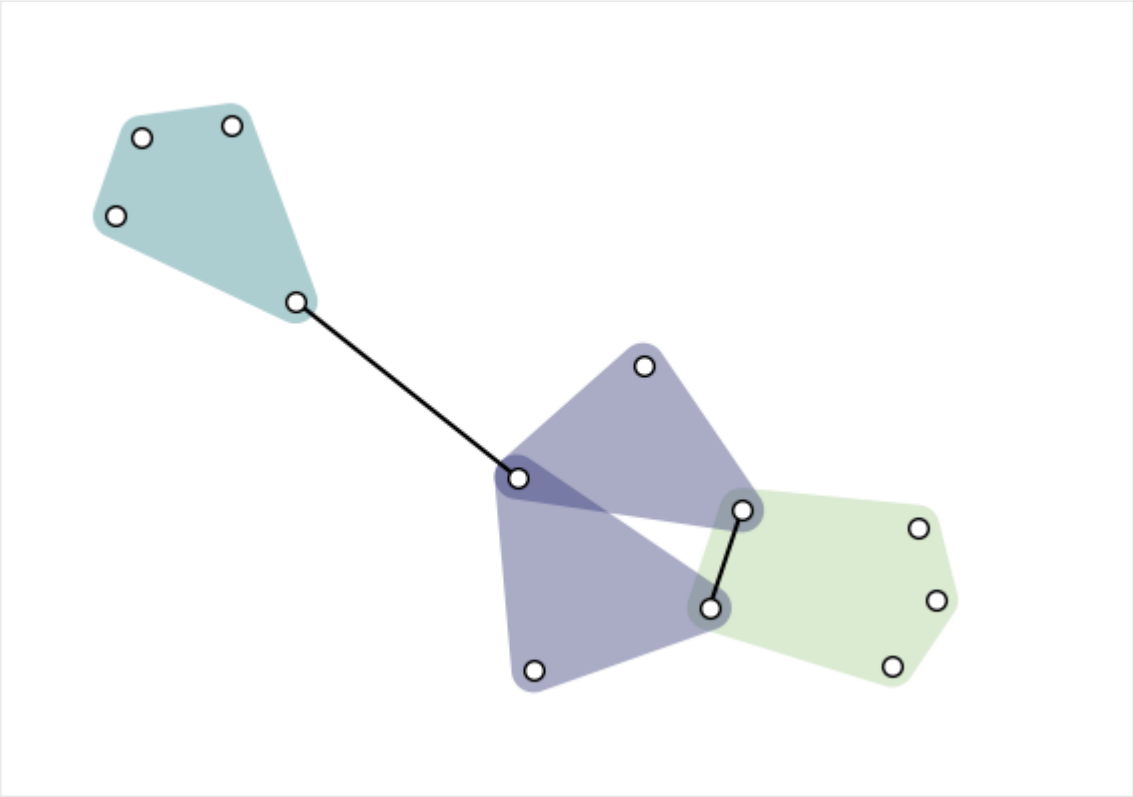


Note that by default, the nodes are drawn too small (size 7) to display the labels nicely. For better visuals, increase the node size to at least 15 when displaying node labels.

Other types of visualizations

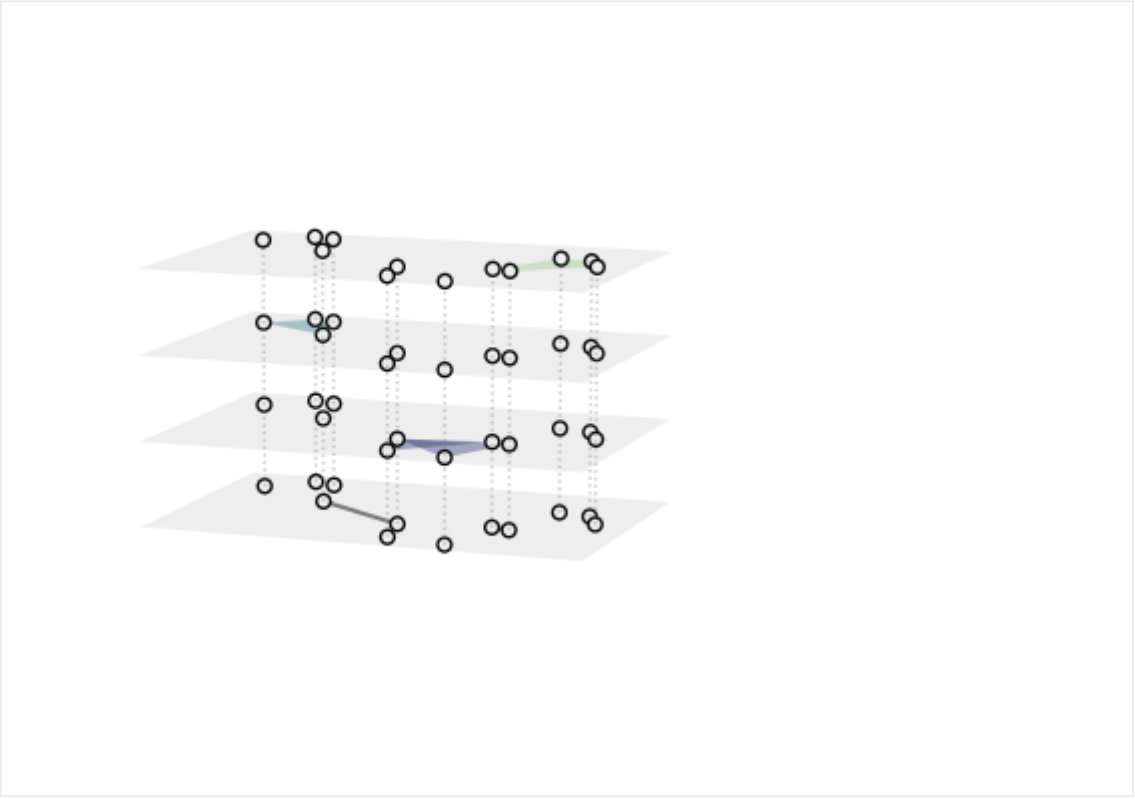
Another common way of visualizing hypergraph is with **convex hulls** as hyperedges. This can be done simply by setting `hull=True`:

[6]:
```
fig, ax = plt.subplots()
xgi.draw(H, pos=pos, ax=ax, hull=True)

plt.show()
```



XGI also offer a function to visualize a hypergraph as a **multilayer**, where each layer contains hyperedges of a given order:

[7]:
```
ax3 = plt.axes(projection="3d")  # requires a 3d axis
xgi.draw_multilayer(H, ax=ax3)

plt.show()
```



Colors and sizes

The drawing functions offer great flexibility in choosing the width, size, and color of the elements that are drawn.

By default, hyperedges are colored according to their order. Hyperedge and node colors can be set manually to a single color, a list of colors, or a by an array/dict/Stat of numerical values.

In the latter case, numerical values are mapped to colors via a colormap which can be changed manually (see colormaps for details):

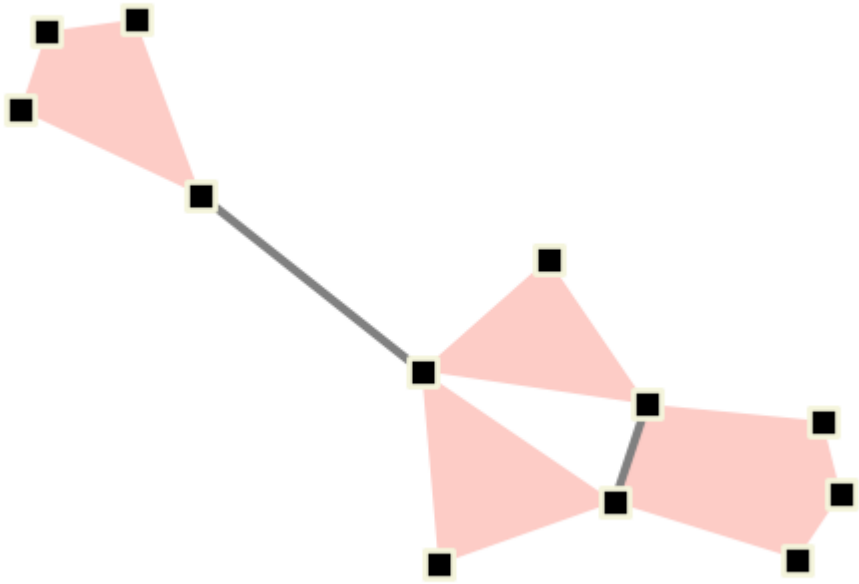
First, let's use single values:

[8]:
```
fig, ax = plt.subplots()

xgi.draw(
    H,
    pos=pos,
    ax=ax,
    node_fc="k",
    node_ec="beige",
    node_shape="s",
    node_size=10,
    node_lw=2,
    edge_fc="salmon",
    dyad_color="grey",
    dyad_lw=3,
)
```

(continues on next page)

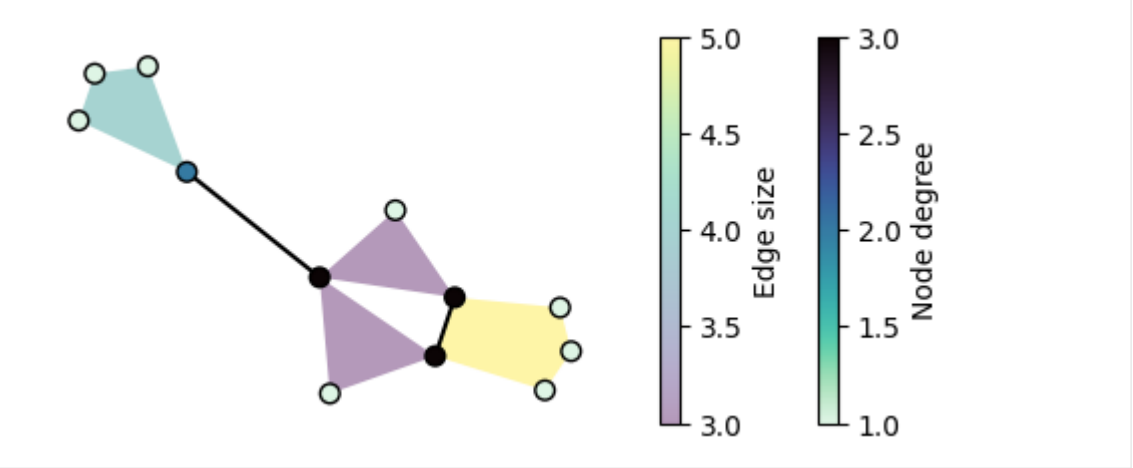
(continued from previous page)

```
plt.show()
```



Now we can use statistic to set the colors and sizes, and change the colormaps:

[9]:
```
fig, ax = plt.subplots(figsize=(6, 2.5))

ax, collections = xgi.draw(
    H,
    pos=pos,
    node_fc=H.nodes.degree,
    edge_fc=H.edges.size,
    edge_fc_cmap="viridis",
    node_fc_cmap="mako_r",
)

node_col, _, edge_col = collections

plt.colorbar(node_col, label="Node degree")
plt.colorbar(edge_col, label="Edge size")

plt.show()
```



Layouts

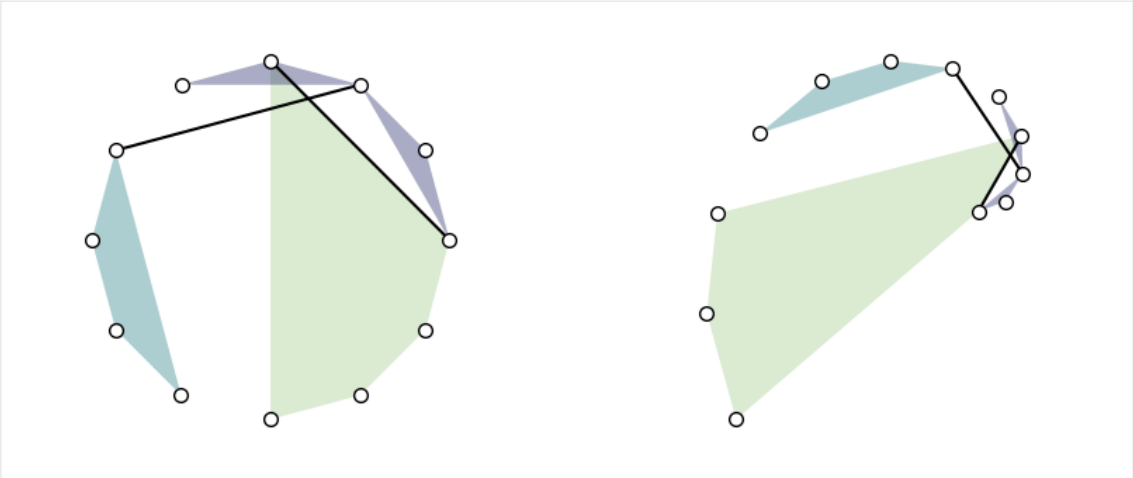
Other layout algorithms are available: * `random_layout`: to position nodes uniformly at random in the unit square (exactly as networkx). * `pairwise_spring_layout`: to position the nodes using the Fruchterman-Reingold force-directed algorithm on the projected graph. In this case the hypergraph is first projected into a graph (1-skeleton) using the `xgi.convert_to_graph(H)` function and then networkx's spring_layout is applied. * `barycenter_spring_layout`: to position the nodes using the Fruchterman-Reingold force-directed algorithm using an augmented version of the the graph projection of the hypergraph, where *phantom nodes* (that we call barycenters) are created for each edge of order $d > 1$ (composed by more than two nodes). Weights are then assigned to all hyperedges of order 1 (links) and to all connections to phantom nodes within each hyperedge to keep them together. Weights scale with the size of the hyperedges. Finally, the weighted version of networkx's spring_layout is applied. * `weighted_barycenter_spring_layout`: same as `barycenter_spring_layout`, but here the weighted version of the Fruchterman-Reingold force-directed algorithm is used. Weights are assigned to all hyperedges of order 1 (links) and to all connections to phantom nodes within each hyperedge to keep them together. Weights scale with the order of the group interaction.

Each layout returns a dictionary that maps nodes ID into (x, y) coordinates.

For example:

[10]:
```
plt.figure(figsize=(10, 4))

ax = plt.subplot(1, 2, 1)
pos_circular = xgi.circular_layout(H)
xgi.draw(H, pos=pos_circular, ax=ax)

ax = plt.subplot(1, 2, 2)
pos_spiral = xgi.spiral_layout(H)
xgi.draw(H, pos=pos_spiral, ax=ax)
```

[10]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x17fc295d0>,
  <matplotlib.collections.LineCollection at 0x17fc291e0>,
  <matplotlib.collections.PatchCollection at 0x17fa2f700>))
```



Simplicial complexes

Simplicial complexes can be visualized using the same functions as for Hypergraphs.

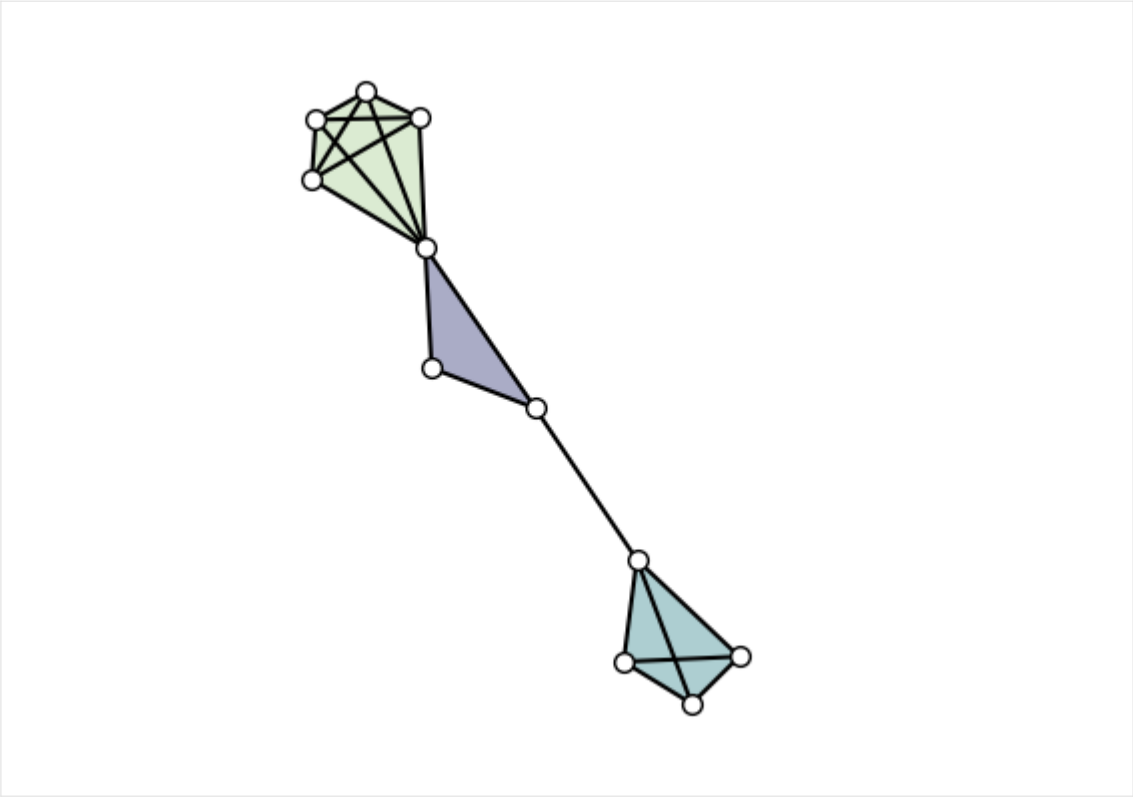
Technical note: By definition, a simplicial complex object contains all sub-simplices. This would make the visualization heavy since all sub-simplices contained in a maximal simplex would overlap. The automatic solution for this, implemented by default in all the layouts, is to convert the simplicial complex into a hypergraph composed solely by its maximal simplices.

Visual note: To visually distinguish simplicial complexes from hypergraphs, the `draw` function will also show all links contained in each maximal simplices (while omitting simplices of intermediate orders).

[11]:
```
SC = xgi.SimplicialComplex()
SC.add_simplices_from([[3, 4, 5], [3, 6], [6, 7, 8, 9], [1, 4, 10, 11, 12], [1,
↪ 4]])
```

[12]:
```
pos = xgi.barycenter_spring_layout(SC, seed=1)
```

[13]:
```
xgi.draw(SC, pos=pos)

plt.show()
```



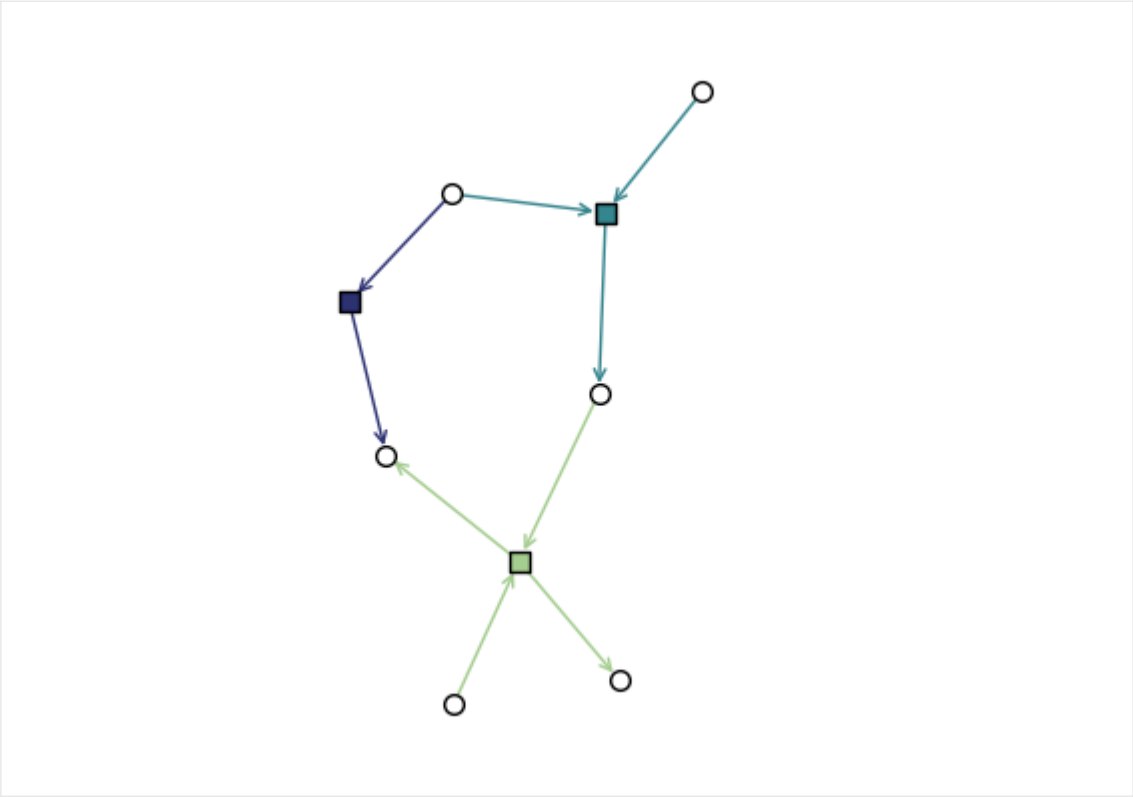
DiHypergraphs

We visualize dihypergraphs as directed bipartite graphs.

[14]:
```
diedges = [({0, 1}, {2}), ({1}, {4}), ({2, 3}, {4, 5})]
DH = xgi.DiHypergraph(diedges)
```

[15]:
```
xgi.draw_bipartite(DH)
```

[15]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x17f9c4cd0>,
  <matplotlib.collections.PathCollection at 0x17f9c4610>))
```

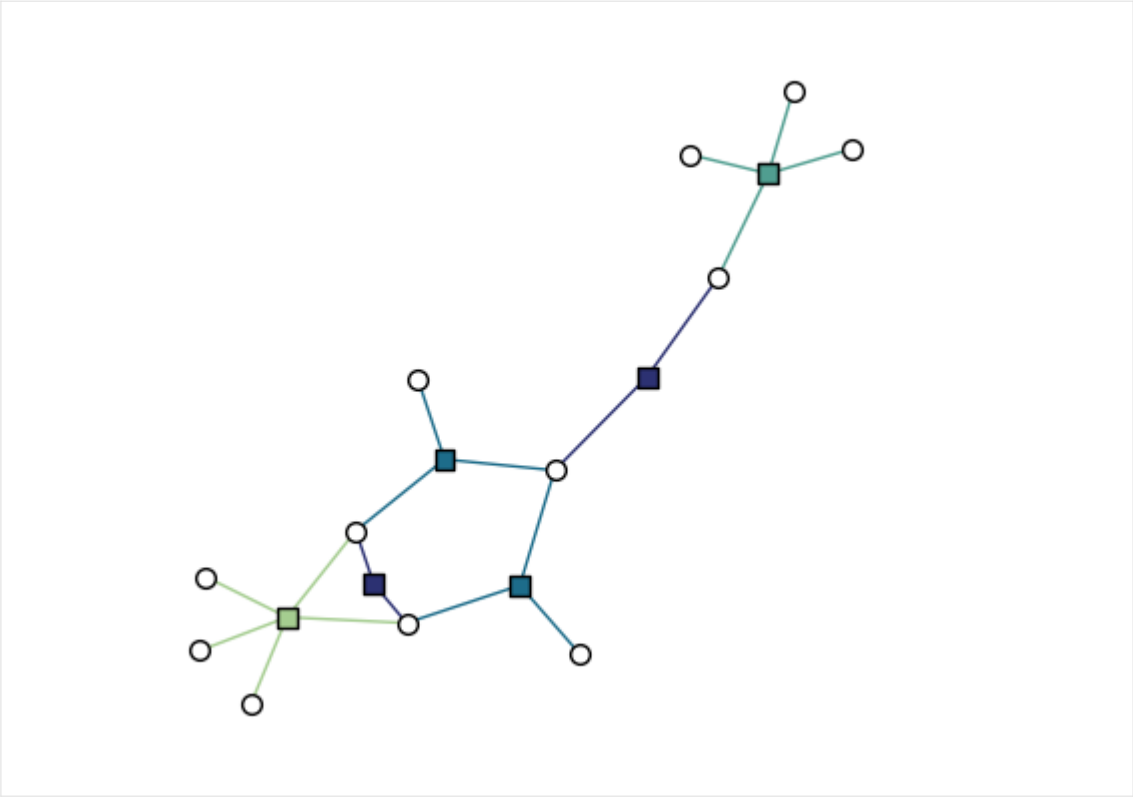


Drawing hypergraphs as bipartite graphs

Not only can we draw dihypergraphs as bipartite graphs, but we can also draw undirected hypergraphs as bipartite graphs.

[16]:
```
xgi.draw_bipartite(H)
```

[16]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x17f5dbac0>,
  <matplotlib.collections.PathCollection at 0x17f5dbe20>,
  <matplotlib.collections.LineCollection at 0x17f5eb9d0>))
```



Now, instead of a single dictionary specifying node positions, we must specify both node and edge marker positions. The `bipartite_spring_layout` method returns a tuple of dictionaries:

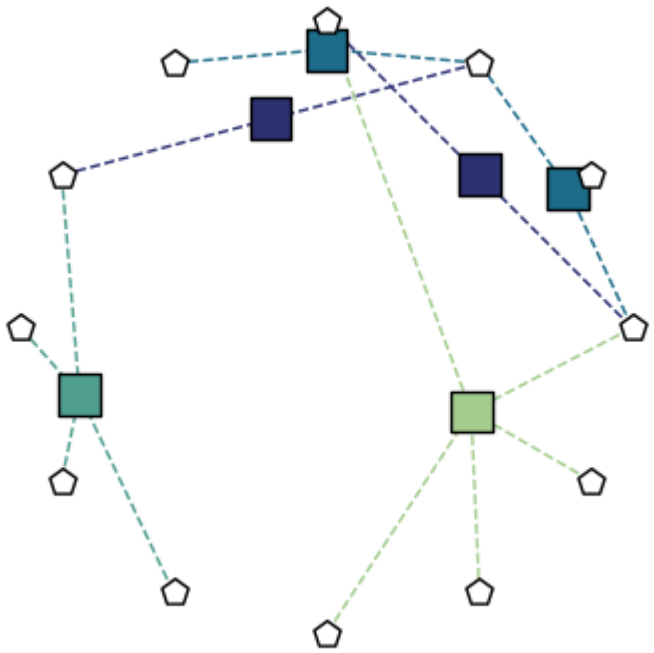
[17]:
```
pos = xgi.bipartite_spring_layout(H)
pos
```

[17]:
```
({1: array([0.04220608, 0.26866056]),
  2: array([0.49436403, 0.17768246]),
  3: array([0.14714446, 0.03094405]),
  4: array([-0.12267079,  0.35109355]),
  5: array([0.01132151, 0.53141135]),
  6: array([ 0.05306611, -0.55582422]),
  7: array([-0.20957516, -0.80710168]),
  8: array([ 0.07044058, -0.98534635]),
  9: array([-0.10909147, -1.        ]),
  10: array([-0.34864708,  0.56110341]),
  11: array([-0.2442516 ,  0.68201836]),
  12: array([0.04980448, 0.62431623])},
 {0: array([0.2778255 , 0.16929486]),
  1: array([0.06275829, 0.32248053]),
  2: array([ 0.11469353, -0.27099664]),
  3: array([-0.03209532, -0.8173815 ]),
  4: array([-0.13396712,  0.50866042]),
  5: array([-0.12332603,  0.20898462])})
```

We can change the style of any of the plot elements just as we can for `draw`. In addition, we can use any of the layouts described above with the `edge_positions_from_barycenters` function, to place the edge markers directly between their constituent nodes.

[18]:
```
node_pos = xgi.circular_layout(H)
edge_pos = xgi.edge_positions_from_barycenters(H, node_pos)
xgi.draw_bipartite(
    H,
    node_shape="p",
    node_size=10,
    edge_marker_size=15,
    dyad_style="dashed",
    pos=(node_pos, edge_pos),
)
```

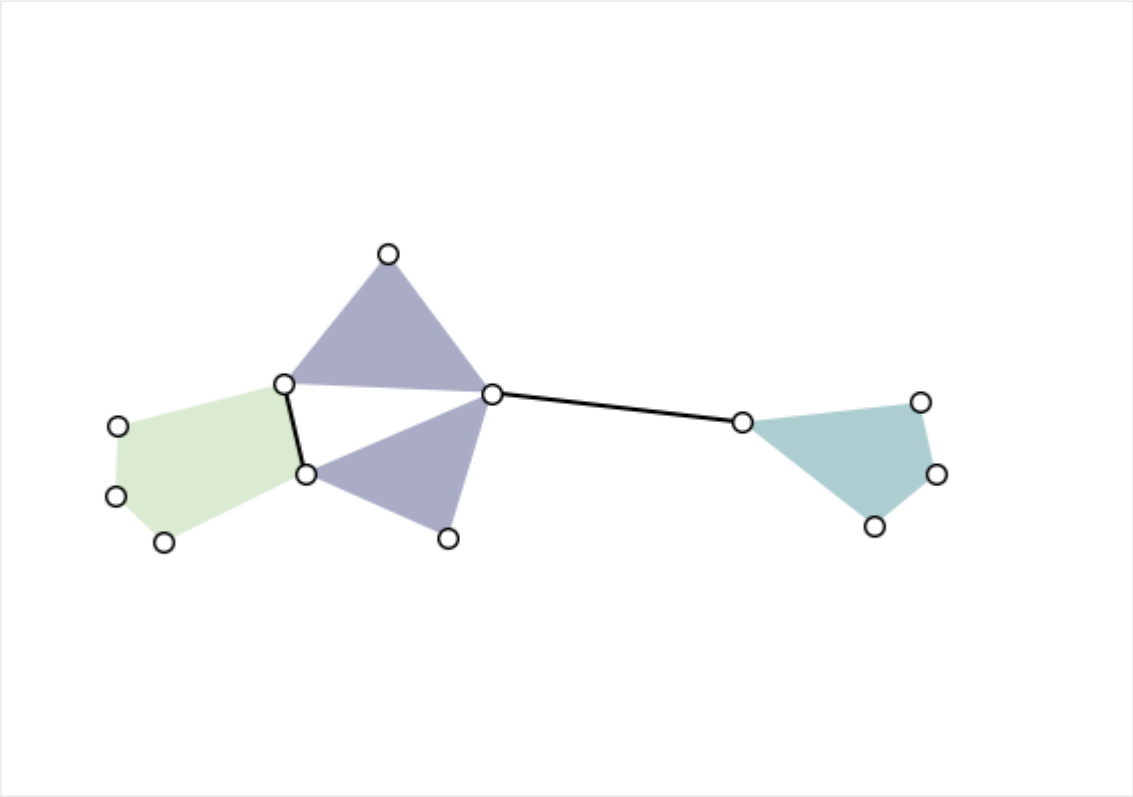
[18]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x17f5e8e80>,
  <matplotlib.collections.PathCollection at 0x17f9a2b00>,
  <matplotlib.collections.LineCollection at 0x17f9c4b80>))
```



Rotating a hypergraph

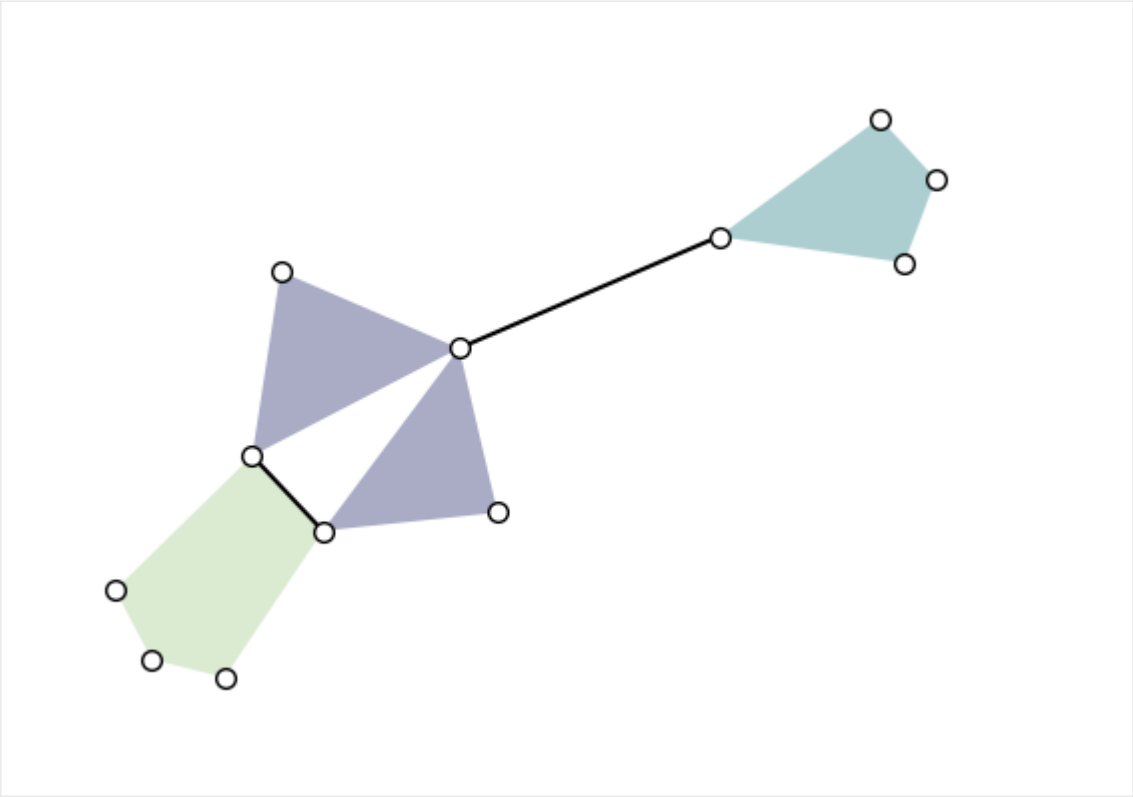
For some hypergraphs, it can be helpful to rotate the positions of the nodes relative to the principal axis. We can do this by generating node positions with any of the functions previously described and then using the function `pca_transform()`. For example:

[19]:
```
pos = xgi.barycenter_spring_layout(H, seed=1)

transformed_pos = xgi.pca_transform(pos)
xgi.draw(H, transformed_pos)

plt.show()
```



We can also rotate the node positions relative to the principal axis:

[20]:
```
# rotation in degrees
transformed_pos = xgi.pca_transform(pos, 30)
xgi.draw(H, transformed_pos)

plt.show()
```



Larger example: generative model

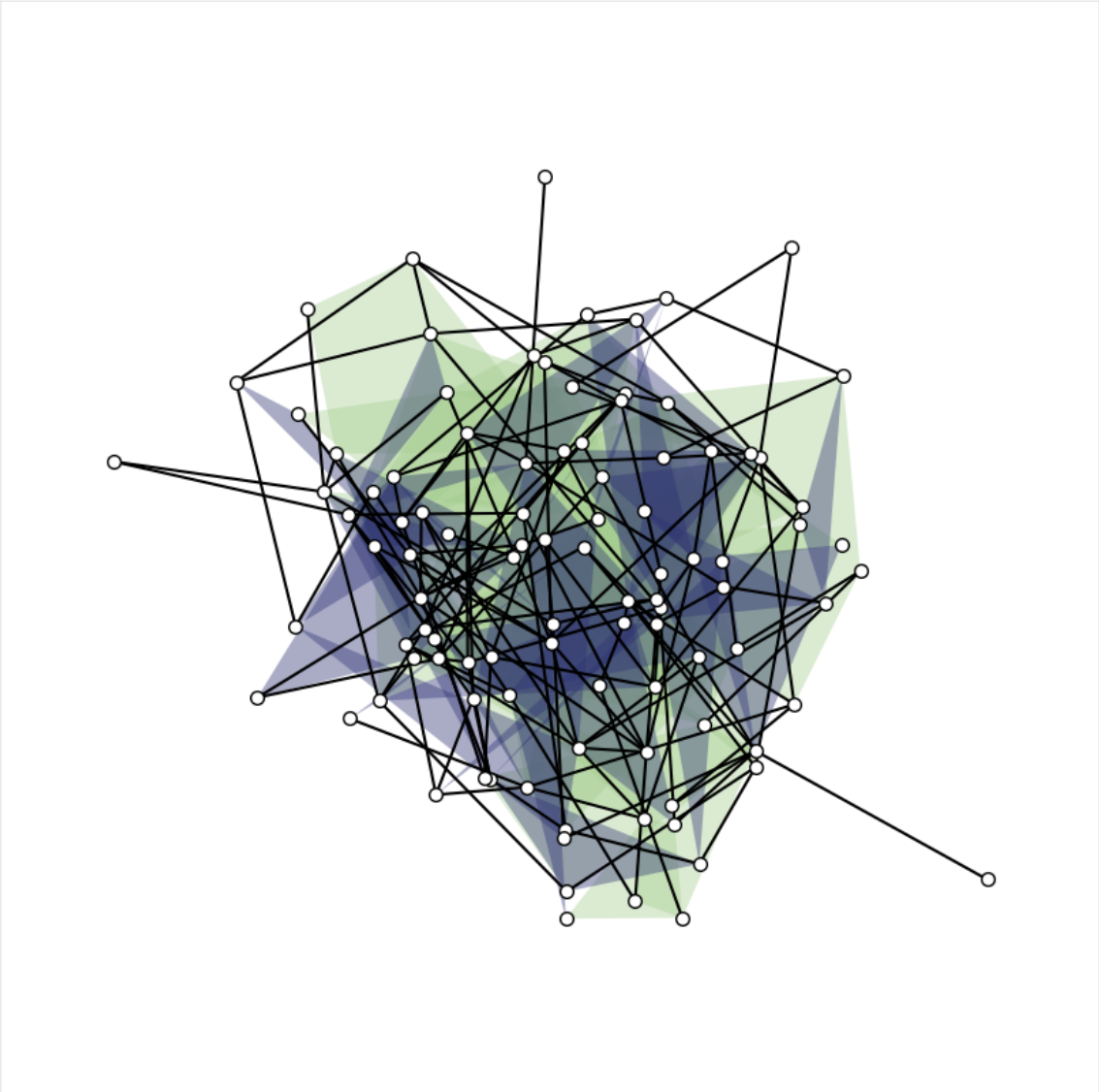
We generate and visualize a random hypergraph.

[21]:
```
n = 100
is_connected = False
while not is_connected:
    H_random = xgi.random_hypergraph(n, [0.03, 0.0002, 0.00001])
    is_connected = xgi.is_connected(H_random)
pos = xgi.barycenter_spring_layout(H_random, seed=1)
```

Since there are more nodes we reduce the `node_size`.

[22]:
```
plt.figure(figsize=(10, 10))
ax = plt.subplot(111)
xgi.draw(H_random, pos=pos, ax=ax)
```

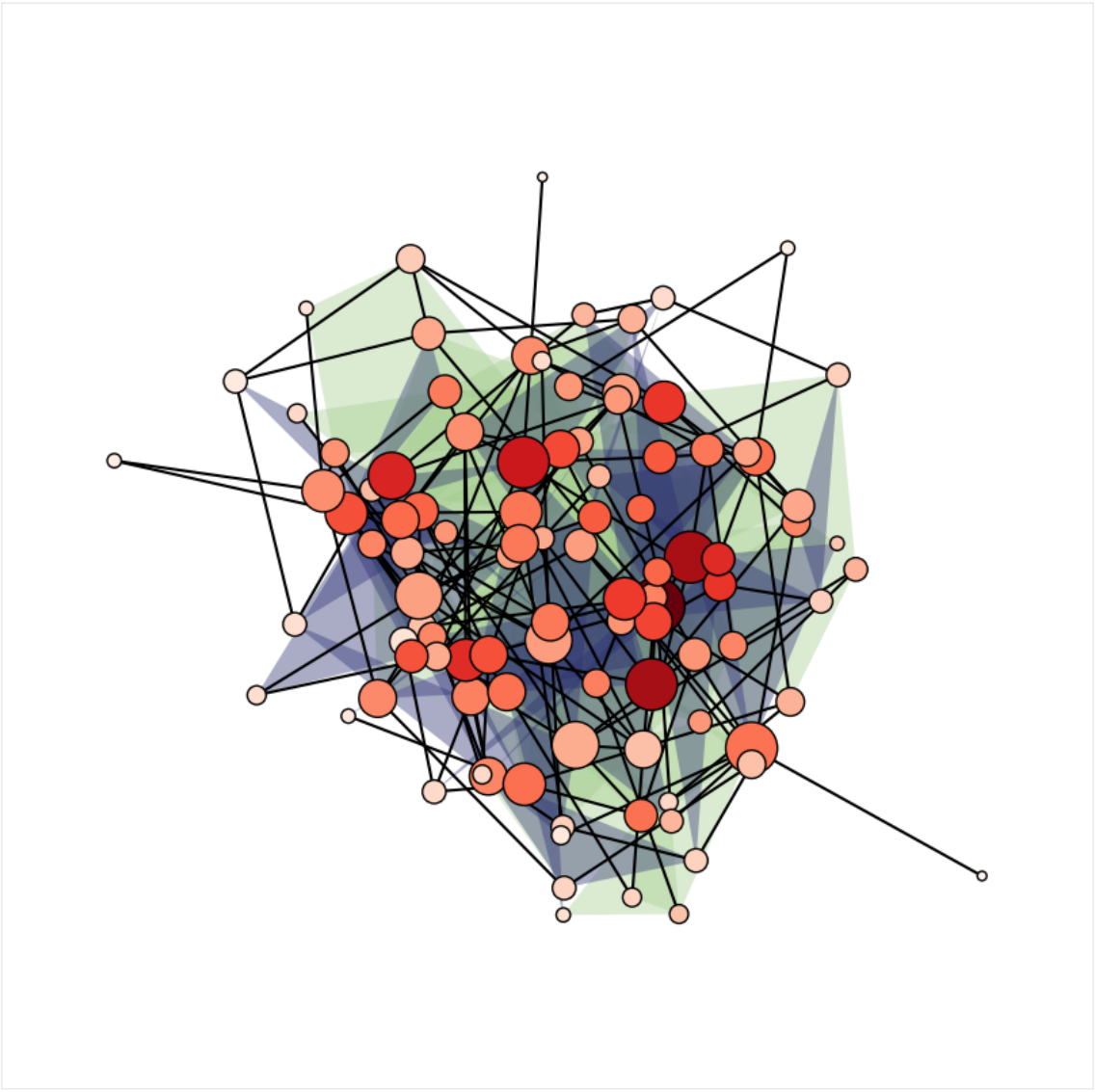
[22]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x17fc79a50>,
  <matplotlib.collections.LineCollection at 0x17fc78850>,
  <matplotlib.collections.PatchCollection at 0x17fdbd0c0>))
```



We can even size/color the nodes and edges by NodeStats or EdgeStats (e.g., degree, centrality, size, etc.)!

[23]:
```
plt.figure(figsize=(10, 10))
ax = plt.subplot(111)
xgi.draw(
    H_random,
    pos=pos,
    ax=ax,
    node_size=H_random.nodes.degree,
    node_fc=H_random.nodes.clique_eigenvector_centrality,
)
```

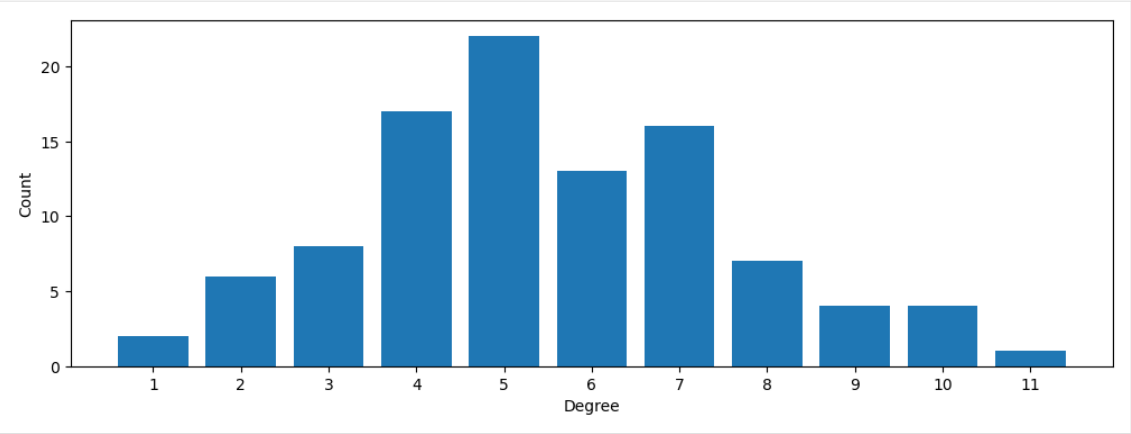
[23]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x17fcf2500>,
  <matplotlib.collections.LineCollection at 0x17fc791b0>,
  <matplotlib.collections.PatchCollection at 0x17fcf2320>))
```



Degree distribution

Using its simplest (higher-order) definition, the degree is the number of hyperedges (of any size) incident on a node.

[24]:
```
centers, heights = xgi.degree_histogram(H_random)

plt.figure(figsize=(12, 4))
ax = plt.subplot(111)

ax.bar(centers, heights)
ax.set_ylabel("Count")
ax.set_xlabel("Degree")
ax.set_xticks(np.arange(1, max(centers) + 1, step=1));
```



[25]:
```
plt.close("all")
```

11.2.7 Statistics

You may have noticed that most of the functionality in the `Hypergraph` and `SimplicialComplex` classes takes care of modifying the unerlying structure of the network, and that these classes provide very limited functionality to compute statistics (a.k.a. measures) from the network. This is done via the `stats` package, explored here.

The stats package is one of the features that sets `xgi` apart from other libraries. It provides a common interface to all statistics that can be computed from a network, its nodes, or edges.

Introduction to Stat objects

Consider the degree of the nodes of a network `H`. After computing the values of the degrees, one may wish to store them in a dict, a list, an array, a dataframe, etc. Through the stats package, `xgi` provides a simple interface that seamlessly allows for this type conversion. This is done via the `NodeStat` class.

[1]:
```
import xgi

H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
H.nodes.degree
```

[1]:
```
NodeStat('degree')
```

This `NodeStat` object is essentially a wrapper over a function that computes the degrees of all nodes. One of the main features of `NodeStat` objects is lazy evaluation: `H.nodes.degree` will not compute the degrees of nodes until a specific output format is requested.

[2]:
```
H.nodes.degree.asdict()
```

[2]:
```
{1: 1, 2: 2, 3: 3, 4: 2, 5: 2}
```

[3]:
```
H.nodes.degree.aslist()
```

[3]:
```
[1, 2, 3, 2, 2]
```

[4]:
```
H.nodes.degree.asnumpy()
```

[4]:
```
array([1, 2, 3, 2, 2])
```

To compute the degrees of a subset of the nodes, call `degree` from a smaller `NodeView`.

[5]:
```
H.nodes([3, 4, 5]).degree.asdict()
```

[5]:
```
{3: 3, 4: 2, 5: 2}
```

Alternatively, to compute the degree of a single node, use square brackets.

[6]:
```
H.nodes.degree[4]
```

[6]:
```
2
```

Make sure the accessed node is in the underlying view.

[7]:
```
# This will raise an exception
# because node 4 is not in the view [1, 2, 3]
#
# H.nodes([1, 2, 3]).degree[4]
#
```

args and kwargs may be passed to `NodeStat` objects, which will be stored and used when the evaluation finally takes place. For example, use the `order` keyword of `degree` to count only those edges of the specified order.

[8]:
```
H.nodes.degree(order=3)
```

[8]:
```
NodeStat('degree', kwargs={'order': 3})
```

[9]:
```
H.nodes.degree(order=3).aslist()
```

[9]:
```
[0, 1, 1, 1, 1]
```

The stats package provides some convenience functions for numerical operations.

[10]:
```
H.nodes.degree.max(), H.nodes.degree.min()
```

[10]:
```
(3, 1)
```

[11]:
```
import numpy as np

st = H.nodes([1, 2, 3]).degree(order=3)
np.round([st.max(), st.min(), st.mean(), st.median(), st.var(), st.std()], 3)
```

[11]:
```
array([1.   , 0.   , 0.667, 1.   , 0.222, 0.471])
```

As a convenience, each node statistic may also be accessed directly through the network itself.

[12]:
```
H.degree()
```

[12]:
```
{1: 1, 2: 2, 3: 3, 4: 2, 5: 2}
```

Note however that `H.degree` is a method that simply returns a dict, not a `NodeStat` object and thus does not support the features discussed above.

Node attributes

Node attributes can be conceived of as a node-object mapping and thus they can also be accessed using the `NodeStat` interface and all its funcitonality.

[13]:
```
H.add_nodes_from(
    [
        (1, {"color": "red", "name": "horse"}),
        (2, {"color": "blue", "name": "pony"}),
        (3, {"color": "yellow", "name": "zebra"}),
        (4, {"color": "red", "name": "orangutan", "age": 20}),
        (5, {"color": "blue", "name": "fish", "age": 2}),
    ]
)
```

Access all attributes of all nodes by specifying a return type.

[14]:
```
H.nodes.attrs.asdict()
```

[14]:
```
{1: {'color': 'red', 'name': 'horse'},
 2: {'color': 'blue', 'name': 'pony'},
 3: {'color': 'yellow', 'name': 'zebra'},
 4: {'color': 'red', 'name': 'orangutan', 'age': 20},
 5: {'color': 'blue', 'name': 'fish', 'age': 2}}
```

Access all attributes of a single node by using square brackets.

[15]:
```
H.nodes.attrs[1]
```

[15]:
```
{'color': 'red', 'name': 'horse'}
```

Access a single attribute of all nodes by specifying a return type.

[16]:
```
H.nodes.attrs("color").aslist()
```

[16]:
```
['red', 'blue', 'yellow', 'red', 'blue']
```

If a node does not have the specified attribute, `None` will be used.

[17]:
```
H.nodes.attrs("age").asdict()
```

[17]:
```
{1: None, 2: None, 3: None, 4: 20, 5: 2}
```

Use the `missing` keyword argument to change the imputed value.

[18]:
```
H.nodes.attrs("age", missing=100).asdict()
```

[18]:
```
{1: 100, 2: 100, 3: 100, 4: 20, 5: 2}
```

Filtering

`NodeView` objects are aware of existing `NodeStat` objects via the `filterby` method.

[19]:
```
H.degree()
```

[19]:
```
{1: 1, 2: 2, 3: 3, 4: 2, 5: 2}
```

[20]:
```
H.nodes.filterby("degree", 2)  # apply the filter to all nodes
```

[20]:
```
NodeView((2, 4, 5))
```

[21]:
```
H.nodes([1, 2, 3]).filterby(
    "degree", 2
)  # apply the filter only to the subset of nodes [1, 2, 3]
```

[21]:
```
NodeView((2,))
```

Nodes can be filtered by attribute via the `filterby_attr` method.

[22]:
```
H.nodes.filterby_attr("color", "red")
```

[22]:
```
NodeView((1, 4))
```

Since `filterby*` methods return a `NodeView` object, multiple filters can be chained, as well as other `NodeStat` calls. The following call computes the local clustering coefficient of those nodes with degree equal to 2 and "color" attribute equal to "blue", and outputs the result as a dict.

[23]:
```
(
    H.nodes.filterby("degree", 2)
    .filterby_attr("color", "blue")
    .clustering_coefficient.asdict()
)
```

[23]:
```
{2: 0.6666666666666666, 5: 1.0}
```

For example, here is how to access the nodes with maximum degree.

[24]:
```
H.nodes.filterby("degree", H.nodes.degree.max())
```

[24]:
```
NodeView((3,))
```

Set operations

Another way of chaining multiple results of `filterby*` methods is by using set operations. Indeed, chaining two filters is the same as intersecting the results of two separate calls:

[25]:
```
print(H.nodes.filterby("degree", 2).filterby_attr("color", "blue"))
print(H.nodes.filterby("degree", 2) & H.nodes.filterby_attr("color", "blue"))
```

```
[2, 5]
[2, 5]
```

Other set operations are also supported.

[26]:
```
nodes1 = H.nodes.filterby("degree", 2)
nodes2 = H.nodes.filterby_attr("color", "blue")
print(f"nodes1 - nodes2 = {nodes1 - nodes2}")
print(f"nodes2 - nodes1 = {nodes2 - nodes1}")
print(f"nodes1 & nodes2 = {nodes1 & nodes2}")
print(f"nodes1 | nodes2 = {nodes1 | nodes2}")
print(f"nodes1 ^ nodes2 = {nodes1 ^ nodes2}")
```

```
nodes1 - nodes2 = [4]
nodes2 - nodes1 = []
nodes1 & nodes2 = [2, 5]
nodes1 | nodes2 = [2, 4, 5]
nodes1 ^ nodes2 = [4]
```

Multiple statistics

One can obtain multiple node statistics at the same time via the `multi` method, which returns `MultiNodeStat` objects.

[27]:
```
H.nodes.multi(["degree", "clustering_coefficient"])
```

[27]:
```
MultiNodeStat(degree, clustering_coefficient)
```

Objects of class `MultiNodeStat` also support lazy evaluation and type conversion.

[28]:
```
H.nodes.multi(["degree", "clustering_coefficient"]).asdict()
```

[28]:
```
{1: {'degree': 1, 'clustering_coefficient': 1.0},
 2: {'degree': 2, 'clustering_coefficient': 0.6666666666666666},
 3: {'degree': 3, 'clustering_coefficient': 0.6666666666666666},
 4: {'degree': 2, 'clustering_coefficient': 1.0},
 5: {'degree': 2, 'clustering_coefficient': 1.0}}
```

There are more options for the output type of a `MultiNodeStat` object.

[29]:
```
ms = H.nodes.multi(["degree", "clustering_coefficient"])

from pprint import pprint

print("# dict of dicts, nodes first:")
pprint(ms.asdict())
print()
print("# dict of dicts, stats first:")
pprint(ms.asdict(transpose=True))
print()
print("# dict of lists:")
pprint(ms.asdict(list))
print()
print("# list of lists, nodes first:")
pprint(ms.aslist())
print()
print("# list of lists, stats first:")
pprint(ms.aslist(transpose=True))
```

(continues on next page)

(continued from previous page)

```
print()
print("# list of dicts:")
pprint(ms.aslist(dict))
```

```
# dict of dicts, nodes first:
{1: {'clustering_coefficient': 1.0, 'degree': 1},
 2: {'clustering_coefficient': 0.6666666666666666, 'degree': 2},
 3: {'clustering_coefficient': 0.6666666666666666, 'degree': 3},
 4: {'clustering_coefficient': 1.0, 'degree': 2},
 5: {'clustering_coefficient': 1.0, 'degree': 2}}

# dict of dicts, stats first:
{'clustering_coefficient': {1: 1.0,
                            2: 0.6666666666666666,
                            3: 0.6666666666666666,
                            4: 1.0,
                            5: 1.0},
 'degree': {1: 1, 2: 2, 3: 3, 4: 2, 5: 2}}

# dict of lists:
{1: [1, 1.0],
 2: [2, 0.6666666666666666],
 3: [3, 0.6666666666666666],
 4: [2, 1.0],
 5: [2, 1.0]}

# list of lists, nodes first:
[[1, 1.0], [2, 0.6666666666666666], [3, 0.6666666666666666], [2, 1.0], [2, 1.
↪0]]

# list of lists, stats first:
[[1, 2, 3, 2, 2], [1.0, 0.6666666666666666, 0.6666666666666666, 1.0, 1.0]]

# list of dicts:
[{'clustering_coefficient': 1.0, 'degree': 1},
 {'clustering_coefficient': 0.6666666666666666, 'degree': 2},
 {'clustering_coefficient': 0.6666666666666666, 'degree': 3},
 {'clustering_coefficient': 1.0, 'degree': 2},
 {'clustering_coefficient': 1.0, 'degree': 2}]
```

Multiple statistics: dataframes

`MultiNodeStat` objects can immediately be output as a Pandas dataframe.

[30]:
```
df = H.nodes.multi(["degree", "clustering_coefficient"]).aspandas()
df
```

[30]:

	degree	clustering_coefficient
1	1	1.000000
2	2	0.666667
3	3	0.666667
4	2	1.000000

(continues on next page)

(continued from previous page)

```
5       2                 1.000000
```

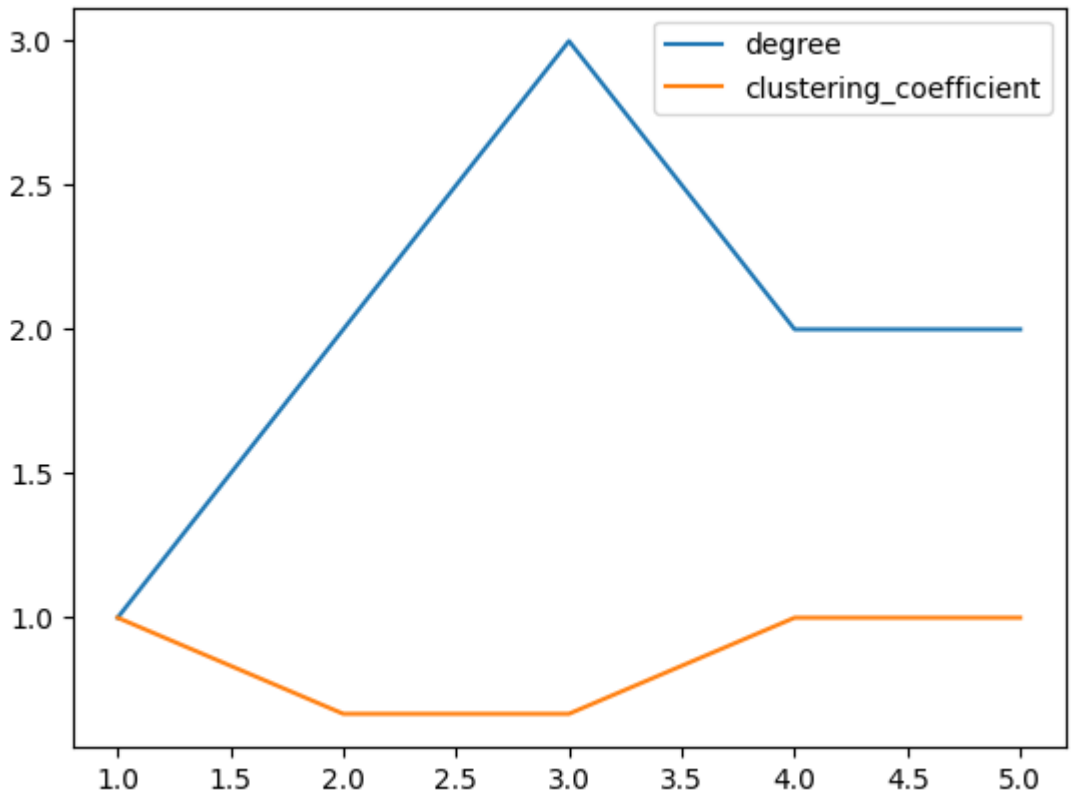
For example, it is now easy to get the per-degree average local clustering coefficient.

[31]:
```
df.groupby("degree").agg("mean")
```

[31]:
```
        clustering_coefficient
degree
1                     1.000000
2                     0.888889
3                     0.666667
```

One may also immediately plot any statistics.

[32]:
```
H.nodes.multi(["degree", "clustering_coefficient"]).aspandas().plot();
```



The `multi` method also accepts `NodeStat` objects, useful when passing arguments to each `NodeStat`, or when requesting attributes.

[33]:
```
H.nodes.multi(["degree", H.nodes.degree(order=3), H.nodes.attrs("color")]).
↪aspandas()
```

[33]:
```
   degree  degree(order=3) attrs(color)
1       1                0          red
2       2                1         blue
3       3                1       yellow
4       2                1          red
```

(continues on next page)

(continued from previous page)

```
5       2                1          blue
```

Edge statistics

Every feature showcased above (lazy evaluation, type conversion, filtering, set operations, and multi objects) is supported for edge-quantity or edge-attribute mappings, via `EdgeStat` objects.

[34]:
```
H.edges.order
```

[34]:
```
EdgeStat('order')
```

[35]:
```
H.edges.order.asdict()
```

[35]:
```
{0: 2, 1: 3, 2: 2}
```

[36]:
```
H.edges.filterby("order", 3)
```

[36]:
```
EdgeView((1,))
```

[37]:
```
H.edges.multi(["order", "size"]).aspandas()
```

[37]:
```
   order  size
0      2     3
1      3     4
2      2     3
```

User-defined statistics

Suppose during the course of your research you come up with a new node-level statistic. For the purpose of this tutorial, we are going to define a statistic called `user_degree`. The `user_degree` of a node is simply its standard degree times 10.

Since this is also a node-quantity mapping, we would like to give it the same interface as `degree` and all the other `NodeStats`. The stats package provides a simple way to do this. Simply use the `nodestat_func` decorator.

[38]:
```
@xgi.nodestat_func
def user_degree(net, bunch):
    """The user degree of a bunch of nodes in net."""
    return {n: 10 * net.degree(n) for n in bunch}
```

Now `user_degree` is a valid stat that can be computed on any hypergraph:

[39]:
```
H.nodes.user_degree.asdict()
```

[39]:
```
{1: 10, 2: 20, 3: 30, 4: 20, 5: 20}
```

Every single feature showcased above is available for use with `user_degree`, including filtering nodes and multi stats objects.

[40]:
```
H.nodes.filterby("user_degree", 20)
```

[40]:
```
NodeView((2, 4, 5))
```

[41]:
```
H.nodes.multi(["degree", "user_degree"]).aspandas()
```

[41]:

	degree	user_degree
1	1	10
2	2	20
3	3	30
4	2	20
5	2	20

The `@xgi.nodestat_func` decorator works on any function or callable that admits two parameters: `net` and `bunch`, where `net` is the network and `bunch` is an iterable of nodes in `net`. Additionally, the function must return a dictionary with pairs of the form `node: value`, where `node` is an element of `bunch`. The library will take care of type conversions, but the output value of this function must always be a dict.

User-defined edge statistics can similarly be defined using the `@xgi.edgestat` decorator.

[ ]:
```

```

11.2.8 Directed Hypergraphs

[1]:
```
import matplotlib.pyplot as plt

import xgi
```

A *directed hypergraph* (or *dihypergraph*), is a hypergraph which keeps track of senders and receivers in a given interaction. As defined in "Hypergraph Theory: An Introduction" by Alain Bretto, dihypergraphs are a set of nodes and a set of directed edges.

We define a directed hyperedge $\overrightarrow{e_i} \in E$ as an ordered pair (e_i^+, e_i^-), where the *tail* of the edge, e_i^+, is the set of senders and the *head*, e_i^-, is the set of receivers. Both are subsets of the node set. We define the members of $\overrightarrow{e_i}$ as $e_i = e_i^+ \cup e_i^-$ and the edge size as $s_i = |e_i|$. Likewise, we define the in-degree, out-degree, and degree of a node i as

$$k_i^{in} = \sum_j^M \mathbf{1}(i \in e_j^-),$$

$$k_i^{out} = \sum_j^M \mathbf{1}(i \in e_j^+),$$

$$k_i = \sum_j^M \mathbf{1}(i \in e_j),$$

respectively, where $\mathbf{1}$ is the indicator function.

These types of hypergraphs are useful for representing, for example, chemical reactions (which have reactants and products) and emails (sender and receivers).

We start by building a dihypergraph.

Building a dihypergraph

We can either build a dihypergraph node-by-node and edge-by-edge, or we can initialize a dihypergraph through its constructor.

We start by building a dihypergraph from the bottom up.

[2]:
```
DH = xgi.DiHypergraph()
print(DH)

DH.add_node(0, name="test")
DH.add_edge(
    [{1, 2, 3}, {3, 4}]
)  # Notice that the head and the tail need not be disjoint.

DH.add_nodes_from([5, 6, 7])
edges = [[{1, 2}, {5, 6}], [{4}, {1, 3}]]
DH.add_edges_from(edges)
DH["name"] = "test"

print("Now that we've added nodes and edges, we have a " + str(DH))
```

```
Unnamed DiHypergraph with 0 nodes and 0 hyperedges
Now that we've added nodes and edges, we have a DiHypergraph named test with 8␣
↪nodes and 3 hyperedges
```

We can also add edge with attributes!

[3]:
```
edges = [
    (([0, 1], [1, 2]), "one", {"color": "red"}),
    (([2, 3, 4], []), "two", {"color": "blue", "age": 40}),
]
DH.add_edges_from(edges)
```

We can also use the constructor to initialize a dihypergraph:

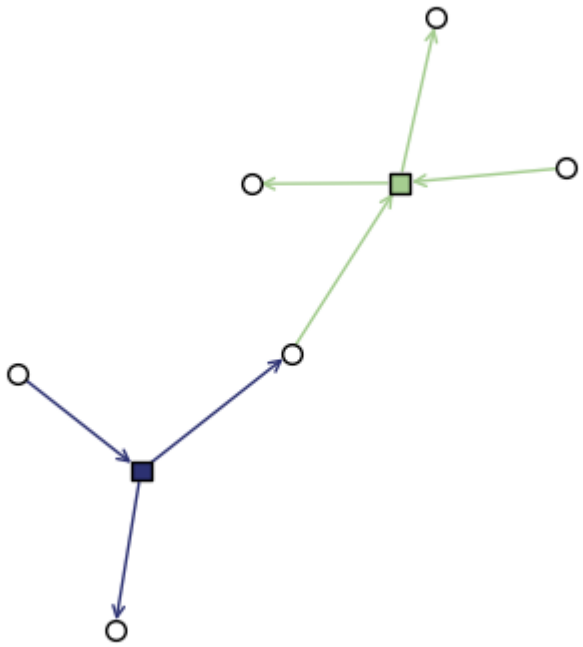
[4]:
```
# from a list
DH1 = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])

# from a dict
DH2 = xgi.DiHypergraph({1: ({1, 2, 3}, {3, 4}), 2: ({1, 2}, {3})})

# from another dihypergraph
DH3 = xgi.DiHypergraph(DH1)
```

Drawing

We can draw a dihypergraph using the function `draw_bipartite()`

[5]:
```
xgi.draw_bipartite(DH1)
```

[5]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x139f08c90>,
  <matplotlib.collections.PathCollection at 0x139f0a8d0>))
```



Views

Nodes and edges are represented by `DiNodeView` and `DiEdgeView` respectively.

[6]:
```
DH.nodes
```

[6]:
```
DiNodeView((0, 1, 2, 3, 4, 5, 6, 7))
```

[7]:
```
DH.edges
```

[7]:
```
DiEdgeView((0, 1, 2, 'one', 'two'))
```

We can access directed edges with the `dimembers()` method and the union of the head and tail with `members()`.

[8]:
```
print("Edge 0:")
print(DH.edges.dimembers(0))
print(DH.edges.members(0))
print("\nThe edge list as a whole:")
```

(continues on next page)

(continued from previous page)

```
print(DH.edges.dimembers())
print(DH.edges.members())
```

```
Edge 0:
({1, 2, 3}, {3, 4})
{1, 2, 3, 4}

The edge list as a whole:
[({1, 2, 3}, {3, 4}), ({1, 2}, {5, 6}), ({4}, {1, 3}), ({0, 1}, {1, 2}), ({2,␣
↪3, 4}, set())]
[{1, 2, 3, 4}, {1, 2, 5, 6}, {1, 3, 4}, {0, 1, 2}, {2, 3, 4}]
```

The naming convention is the same for node memberships.

[9]:
```
print("memberships for node 0:")
print(DH.nodes.dimemberships(0))
print(DH.nodes.memberships(0))

print("\nAll node memberships:")
print(DH.nodes.dimemberships())
print(DH.nodes.memberships())
```

```
memberships for node 0:
(set(), {'one'})
{'one'}

All node memberships:
{0: (set(), {'one'}), 1: ({2, 'one'}, {0, 1, 'one'}), 2: ({'one'}, {0, 1, 'two
↪'}), 3: ({0, 2}, {0, 'two'}), 4: ({0}, {2, 'two'}), 5: ({1}, set()), 6: ({1},
↪ set()), 7: (set(), set())}
{0: {'one'}, 1: {0, 1, 2, 'one'}, 2: {0, 1, 'one', 'two'}, 3: {0, 2, 'two'}, 4:
↪ {0, 2, 'two'}, 5: {1}, 6: {1}, 7: set()}
```

We can also access the head and tail of an edge:

[10]:
```
print("Head and tail of edge 0:")
print(DH.edges.head(0))
print(DH.edges.tail(0))

print("\nThe head as a whole:")
print(DH.edges.head())

print("\nThe tail as a whole:")
print(DH.edges.tail())
```

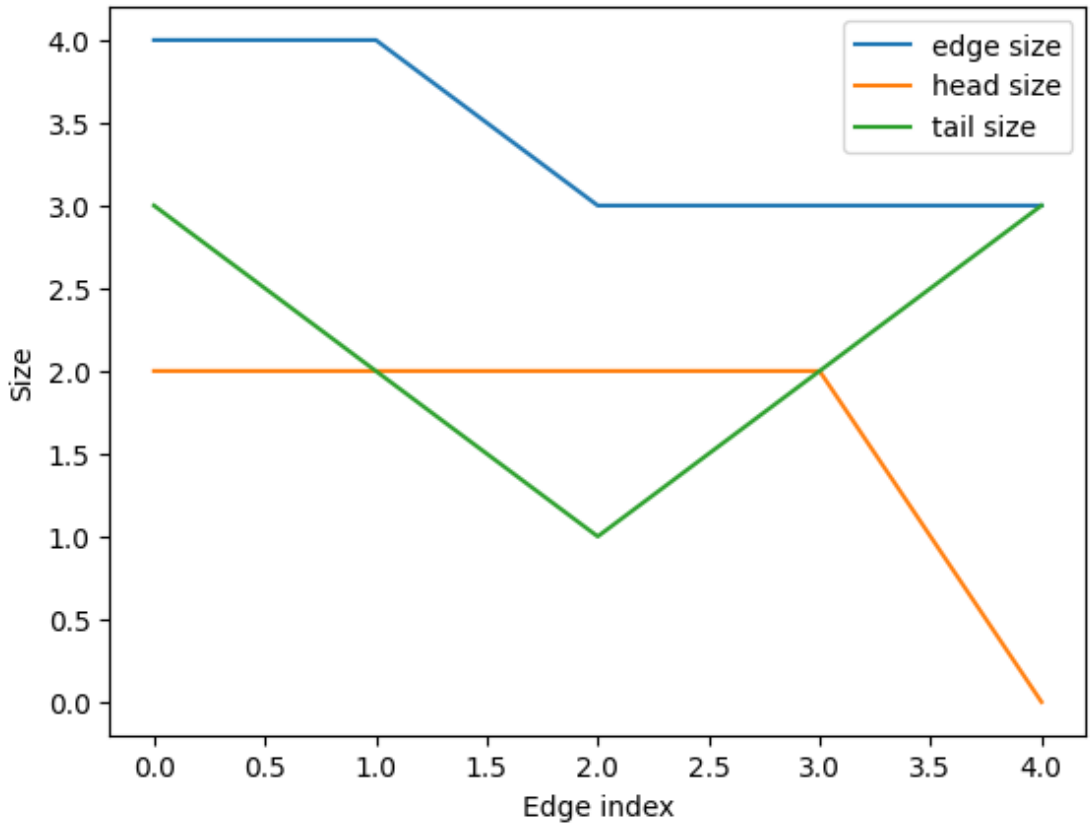
```
Head and tail of edge 0:
{3, 4}
{1, 2, 3}

The head as a whole:
[{3, 4}, {5, 6}, {1, 3}, {1, 2}, set()]

The tail as a whole:
[{1, 2, 3}, {1, 2}, {4}, {0, 1}, {2, 3, 4}]
```

Stats

The `DiNodeStat` and `DiEdgeStat` represent directed node and edge statistics. For nodes, we have `in_degree`, `out_degree`, and `degree` and for edges, we have `size`, `order`, `head_size`, and `tail_size`.

[11]:
```
s = DH.edges.size.asnumpy()
s_in = DH.edges.head_size.asnumpy()
s_out = DH.edges.tail_size.asnumpy()
```

[12]:
```
plt.plot(s, label="edge size")
plt.plot(s_in, label="head size")
plt.plot(s_out, label="tail size")
plt.legend()
plt.ylabel("Size")
plt.xlabel("Edge index")
plt.show()
```



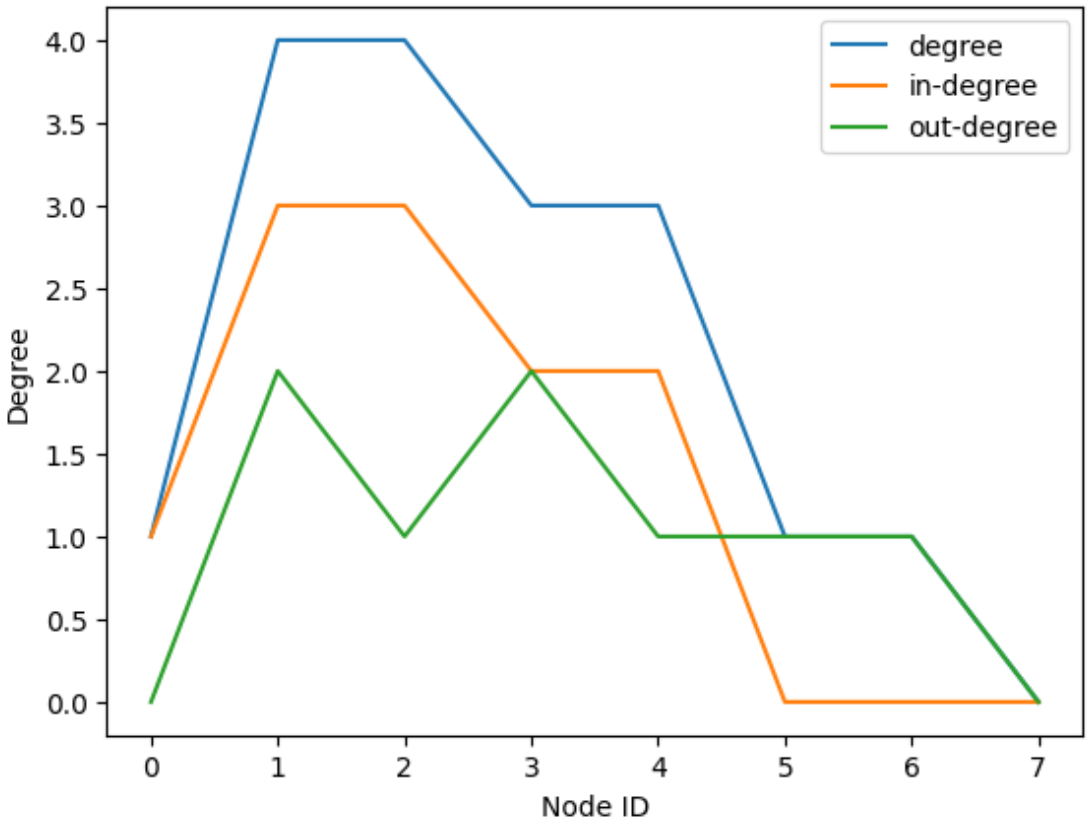
[13]:
```
k = DH.nodes.degree.asnumpy()
k_in = DH.nodes.in_degree.asnumpy()
k_out = DH.nodes.out_degree.asnumpy()
```

[14]:
```
plt.plot(k, label="degree")
plt.plot(k_in, label="in-degree")
plt.plot(k_out, label="out-degree")
```

(continues on next page)

(continued from previous page)

```
plt.legend()
plt.ylabel("Degree")
plt.xlabel("Node ID")
plt.show()
```



We can convert from dihypergraphs to hypergraphs through the constructor...

[15]:
```
H = xgi.Hypergraph(DH1)
H.edges.members()
```

[15]:
```
[{1, 2, 5, 6}, {1, 3, 4}]
```

...or through the convert module.

[16]:
```
H = xgi.to_hypergraph(DH1)
H.edges.members()
```

[16]:
```
[{1, 2, 5, 6}, {1, 3, 4}]
```

11.3 In Depth tutorials

11.3.1 In Depth 1 - Drawing nodes

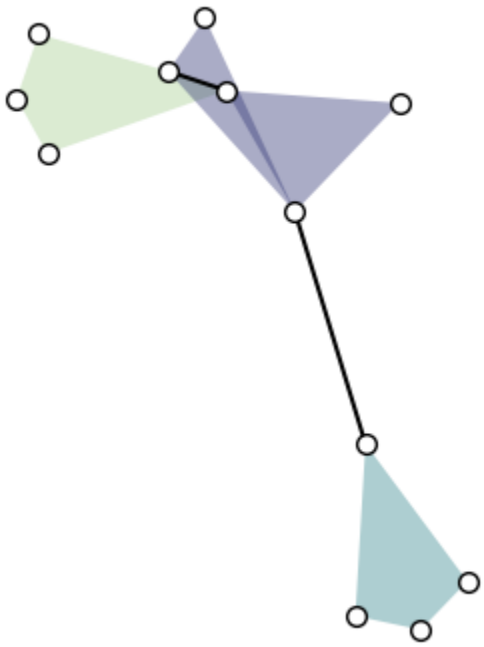
Here we show the fuctionalities and parameters of `xgi.draw_nodes()`. It is similar to the networkx corresponding function (+ some bonus) and heavily relies on matplotlib's scatter function.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np

import xgi
```

Les us first create a small toy hypergraph containing edges of different sizes.

[2]:
```
edges = [[1, 2, 3], [3, 4, 5], [3, 6], [6, 7, 8, 9], [1, 4, 10, 11, 12], [1, 4]]

H = xgi.Hypergraph(edges)

pos = xgi.barycenter_spring_layout(H, seed=42)  # fix position
```

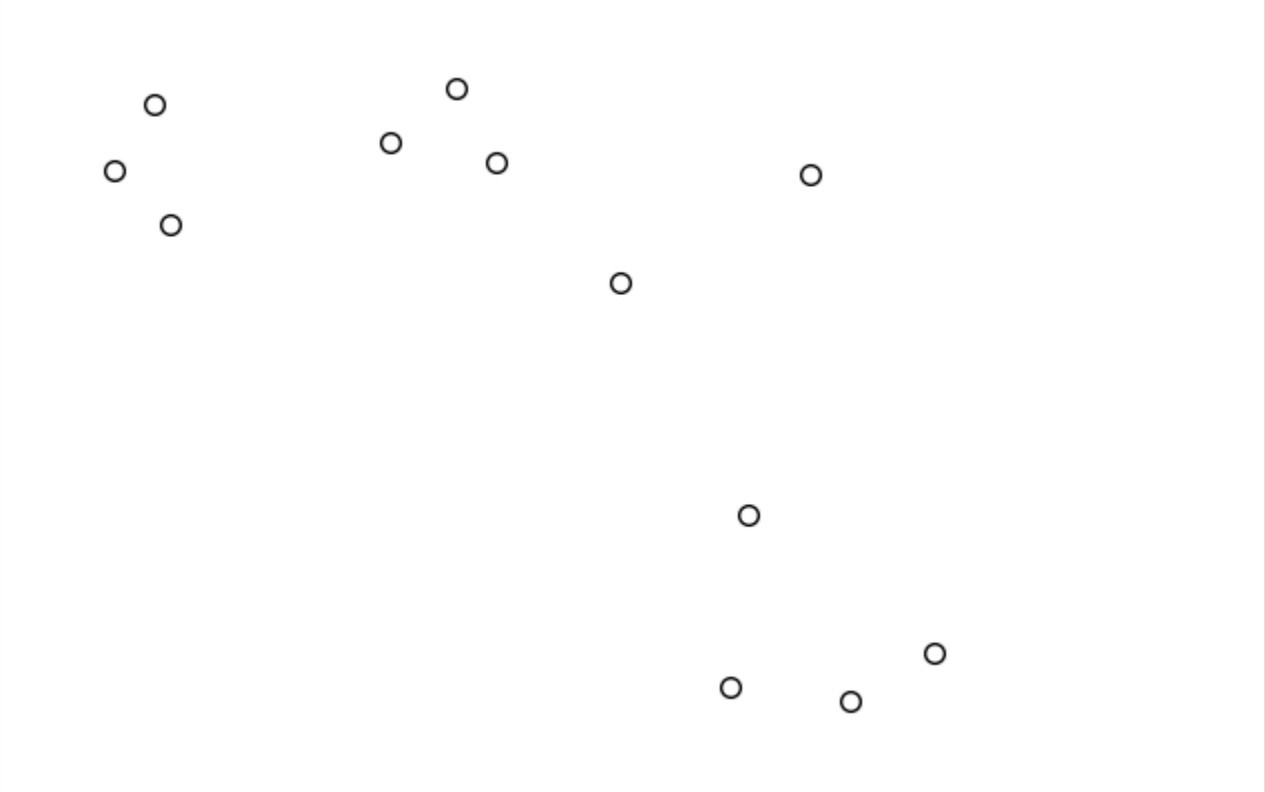
[3]:
```
xgi.draw(H, pos);
```



Basics

Let's jump right into how `xgi.draw_nodes()` works. By default, it gives:

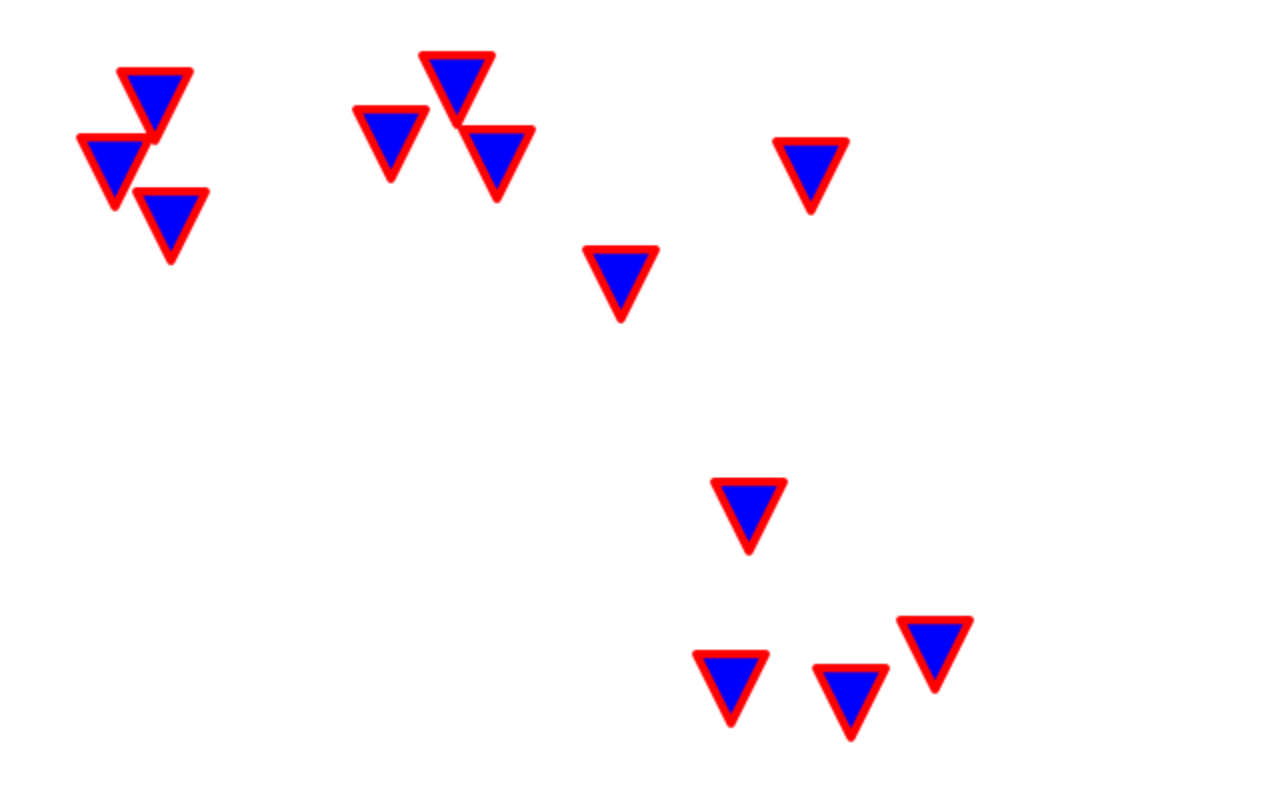
[4]:
```
ax, node_collection = xgi.draw_nodes(H, pos)
```



Note that it returns a tuple `(Axes, PathCollection)`. The `PathCollection` is what matplotlib's `plt.scatter()` returns and can be used later to plot a colorbar.

The color, size, linewidth, and shape of the nodes can all be customised:

[5]:
```
ax, node_collection = xgi.draw_nodes(
    H, pos, node_fc="b", node_ec="r", node_shape="v", node_size=25, node_lw=3
)
```

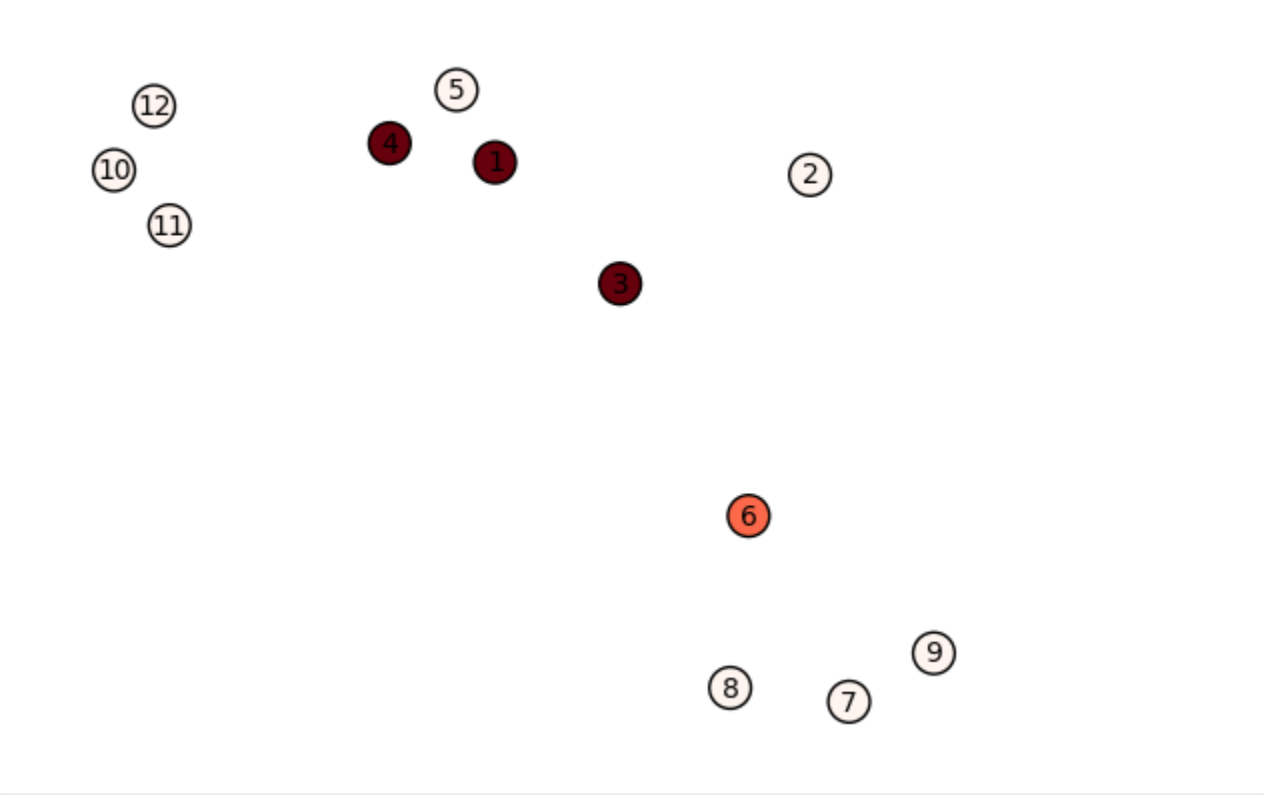


Colormaps

In XGI, you can easily color nodes according to a NodeStat, or just an array or a dict:

[6]:
```
ax, node_collection = xgi.draw_nodes(
    H, pos, node_fc=H.nodes.degree(), node_labels=True, node_size=15
)
print(H.nodes.degree().asdict())
```

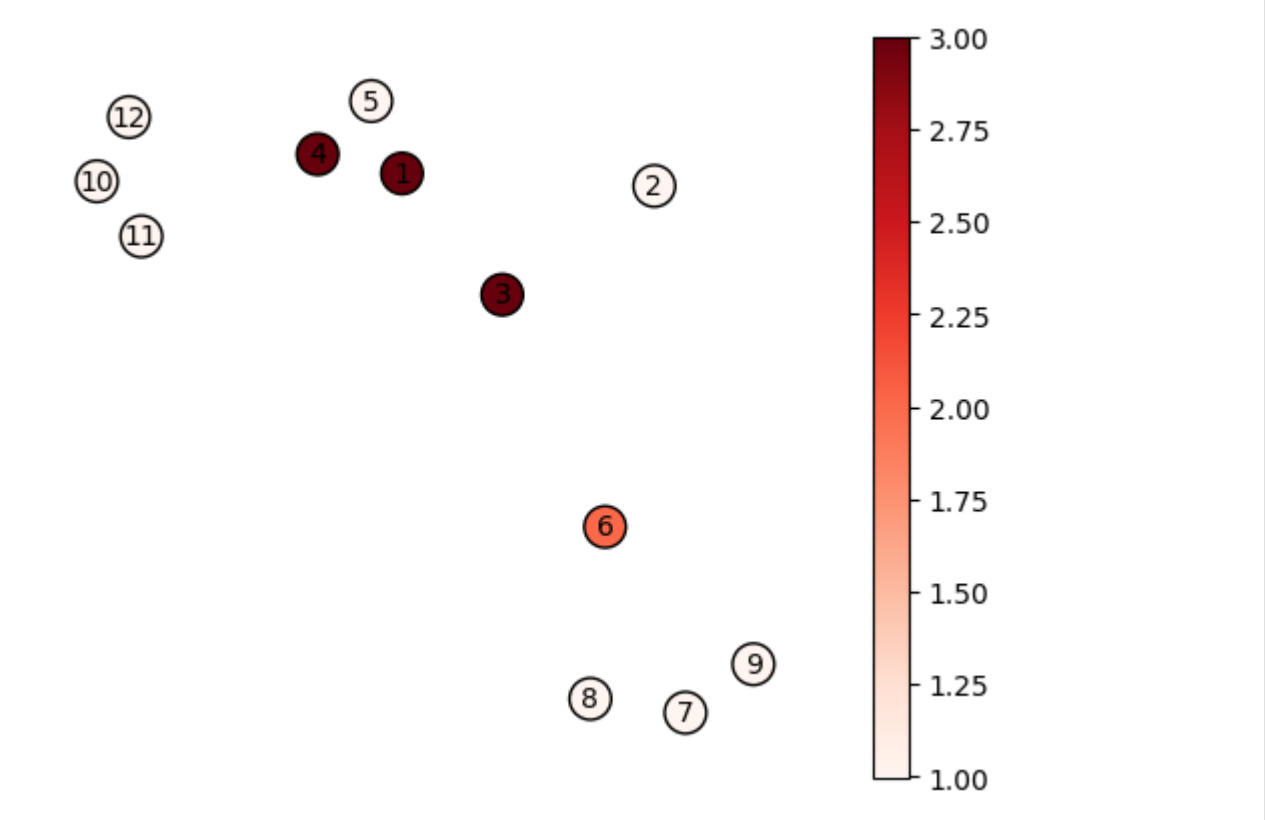
```
{1: 3, 2: 1, 3: 3, 4: 3, 5: 1, 6: 2, 7: 1, 8: 1, 9: 1, 10: 1, 11: 1, 12: 1}
```



By default, the colormap used is "Reds". To visualise the values corresponding to the colors, nothing easier than plotting a colorbar:

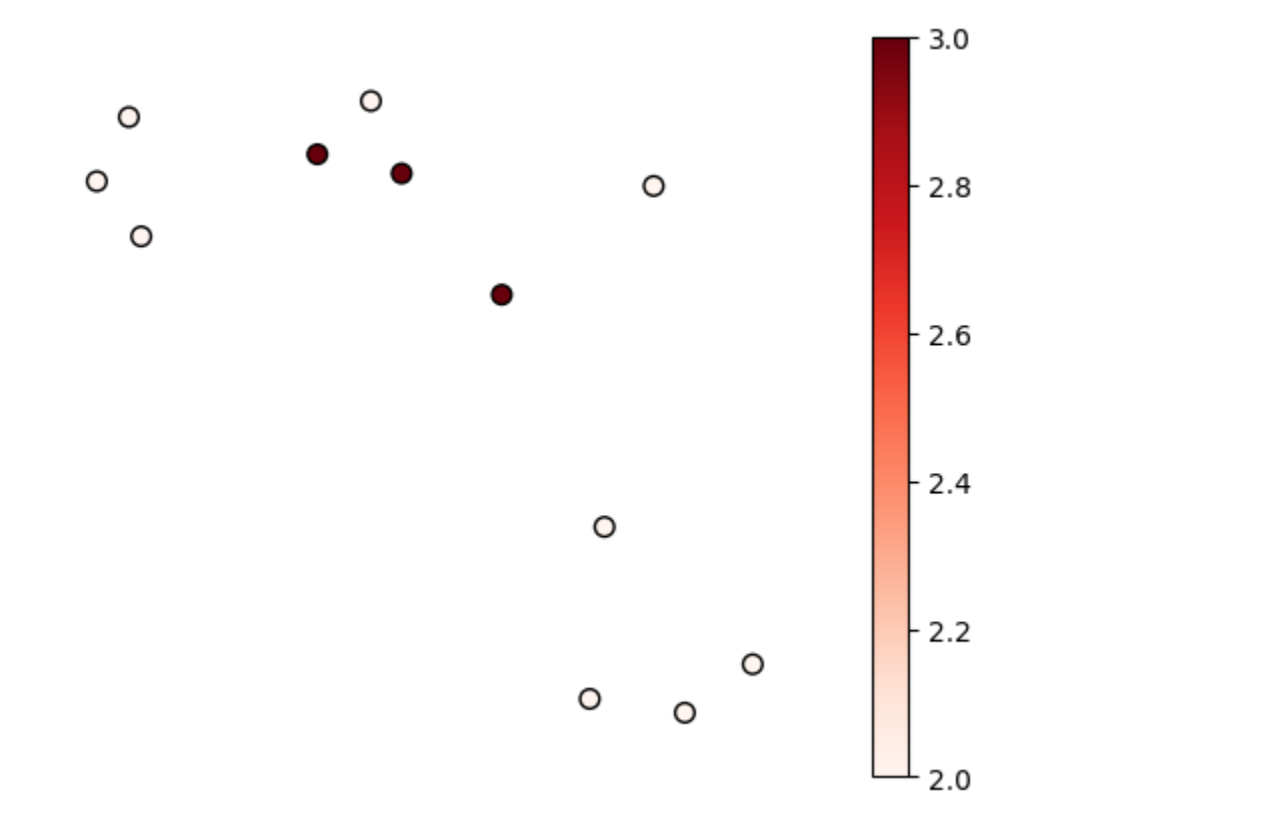
[7]:
```
ax, node_collection = xgi.draw_nodes(
    H, pos, node_fc=H.nodes.degree(), node_labels=True, node_size=15
)
print(H.nodes.degree().asdict())
plt.colorbar(node_collection)
plt.show()
```

```
{1: 3, 2: 1, 3: 3, 4: 3, 5: 1, 6: 2, 7: 1, 8: 1, 9: 1, 10: 1, 11: 1, 12: 1}
```



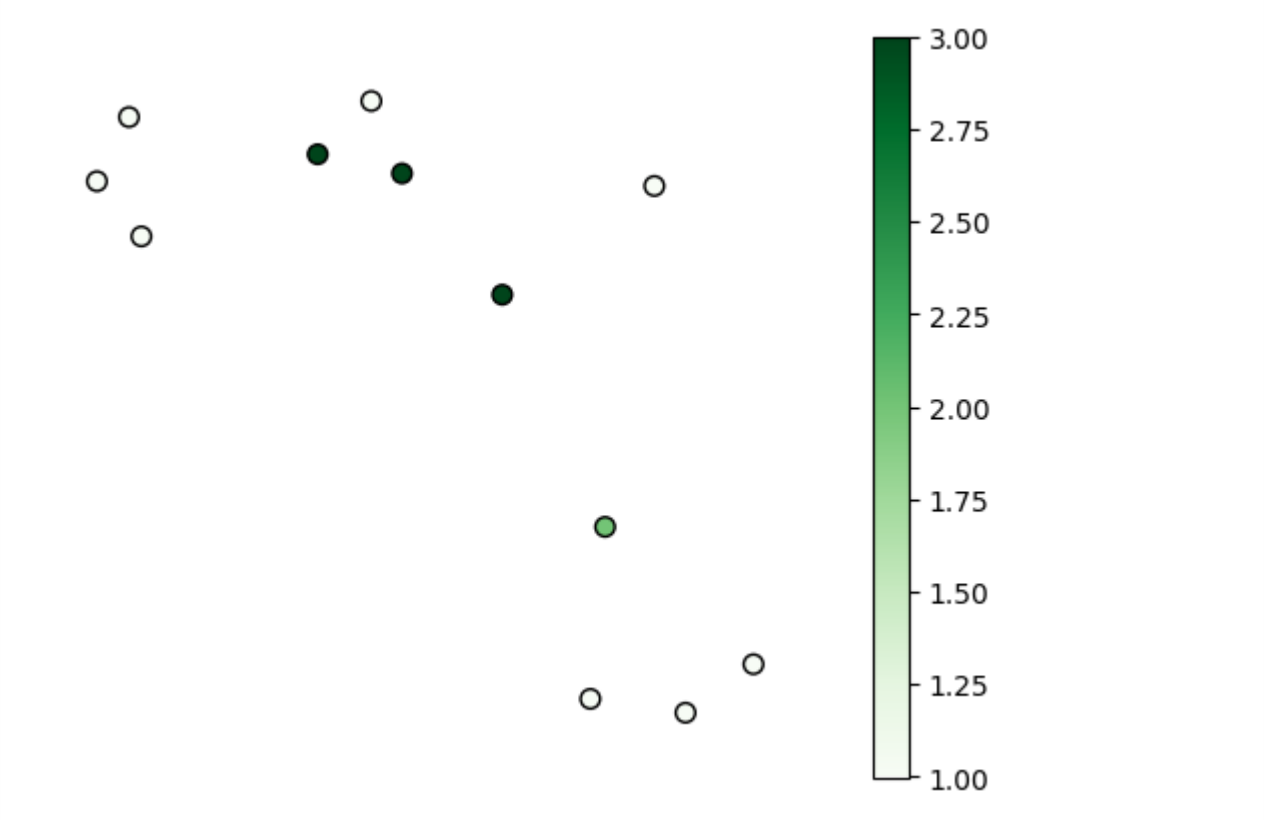
You can specify a `vmin` and `vmax` for the node colors:

[8]:
```
ax, node_collection = xgi.draw_nodes(H, pos, node_fc=H.nodes.degree(), vmin=2)
plt.colorbar(node_collection)
plt.show()
```



It's also easy to specify a different colormap:

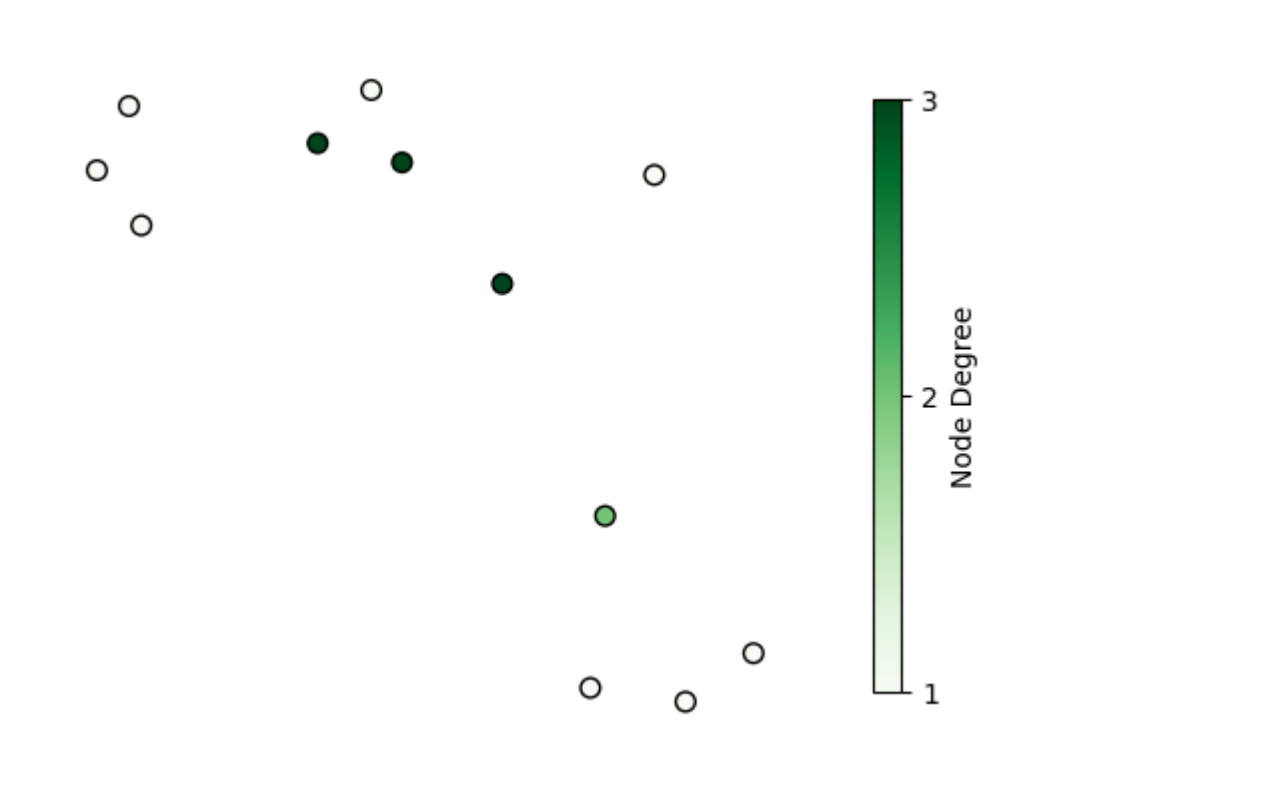
[9]:
```
ax, node_collection = xgi.draw_nodes(
    H, pos, node_fc=H.nodes.degree(), node_fc_cmap="Greens"
)

plt.colorbar(node_collection)
plt.show()
```



And of course, all the colorbar customisation offered by matplotlib are available, for example:

[10]:
```
ax, node_collection = xgi.draw_nodes(
    H, pos, node_fc=H.nodes.degree(), node_fc_cmap="Greens"
)

plt.colorbar(node_collection, label="Node Degree", shrink=0.8, ticks=[1, 2, 3])
```

[10]:
```
<matplotlib.colorbar.Colorbar at 0x17e8018b0>
```

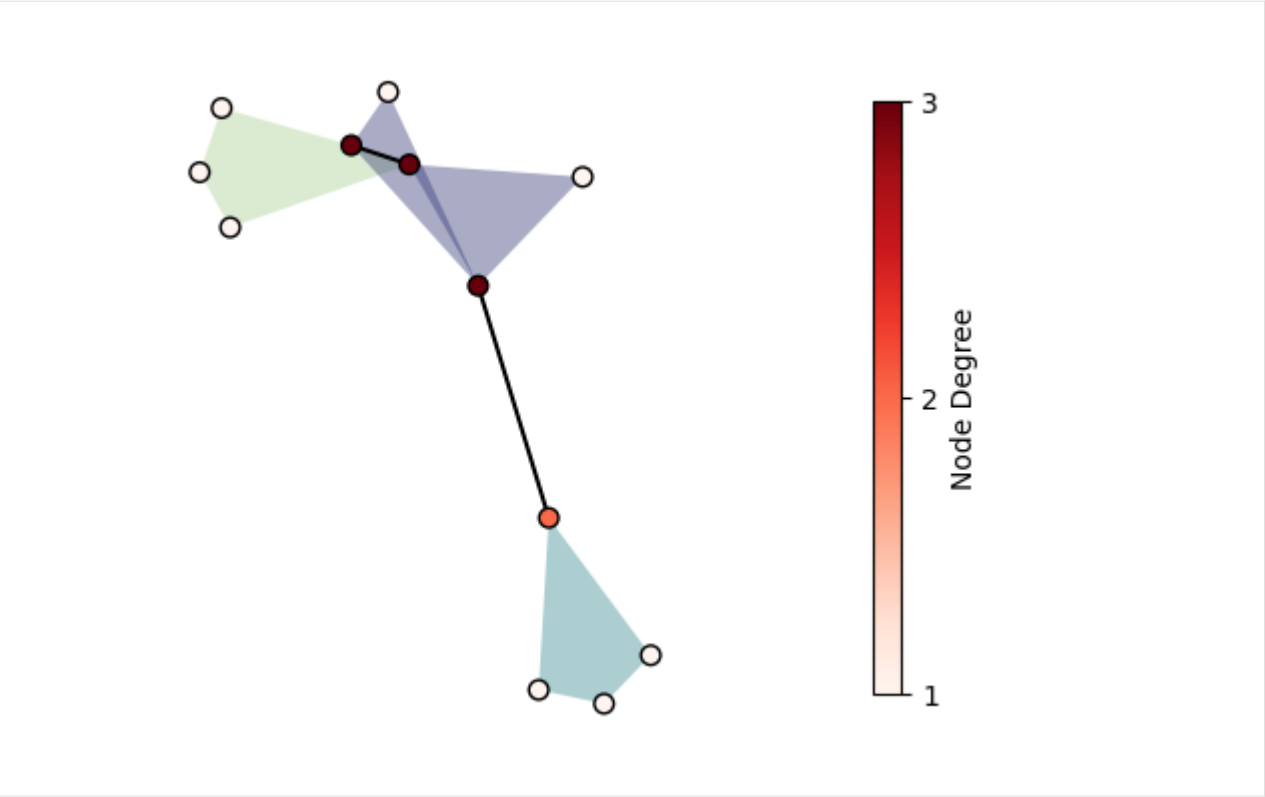


Combine with edges

This can be either via `xgi.draw()`:

[11]:
```
ax, collections = xgi.draw(H, pos=pos, node_fc=H.nodes.degree())

(node_collection, _, _) = collections

plt.colorbar(node_collection, label="Node Degree", shrink=0.8, ticks=[1, 2, 3])
```

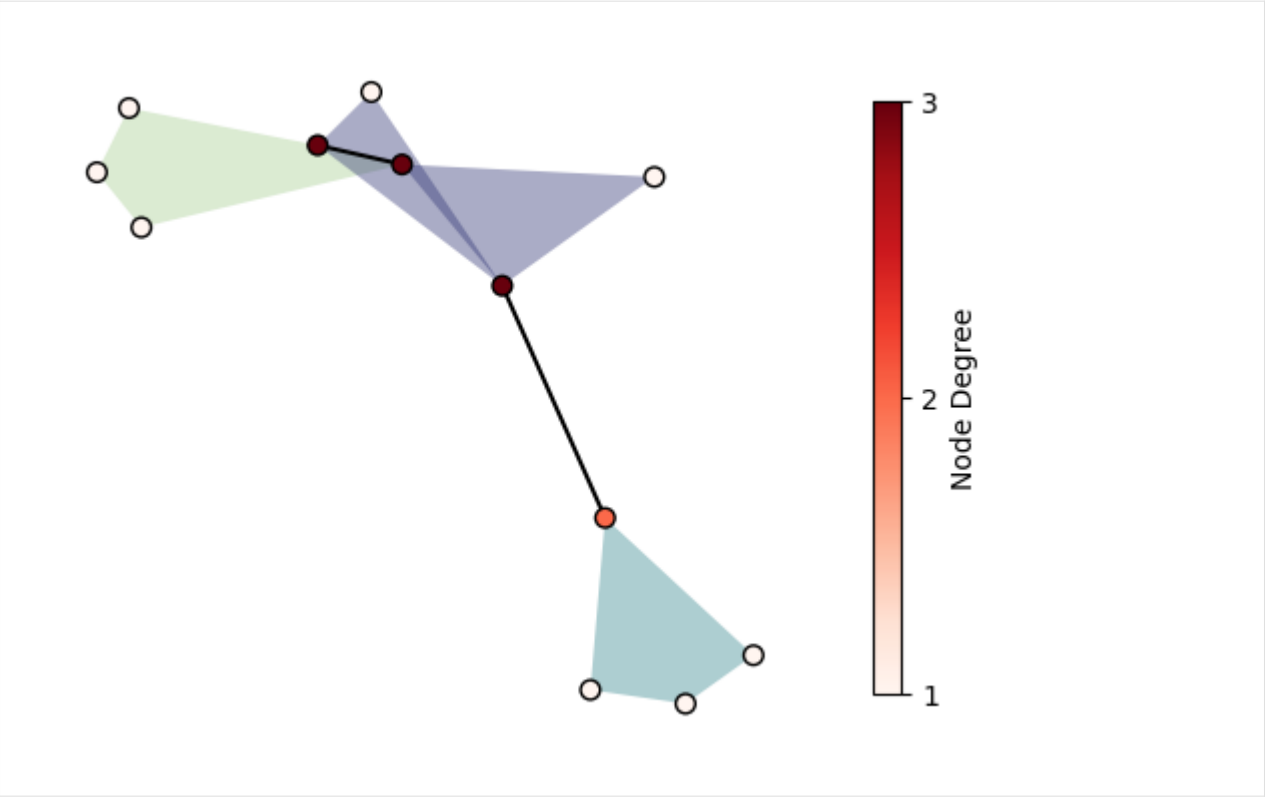
[11]:
```
<matplotlib.colorbar.Colorbar at 0x17e9d85e0>
```



Or by combining it with `xgi.hyperedges()` for more control:

[12]:
```
ax, node_collection = xgi.draw_nodes(H, pos=pos, node_fc=H.nodes.degree(), zorder=5)

ax = xgi.draw_hyperedges(H, pos=pos)

plt.colorbar(node_collection, label="Node Degree", shrink=0.8, ticks=[1, 2, 3])
```

[12]:
```
<matplotlib.colorbar.Colorbar at 0x17ea773d0>
```

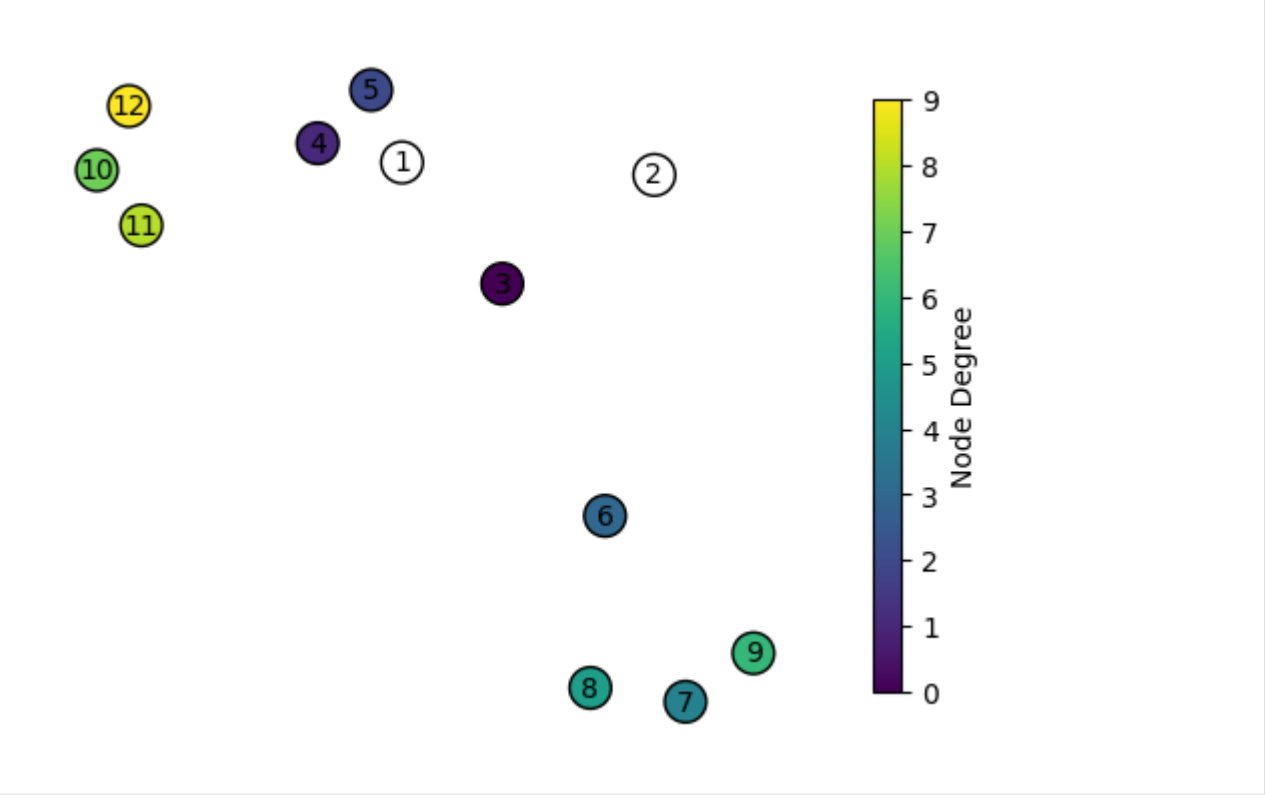


Notice that here we had to specify the `zorder` of the nodes which is 0 by default in `xgi.draw_nodes()`!

Non-finite values for node colors

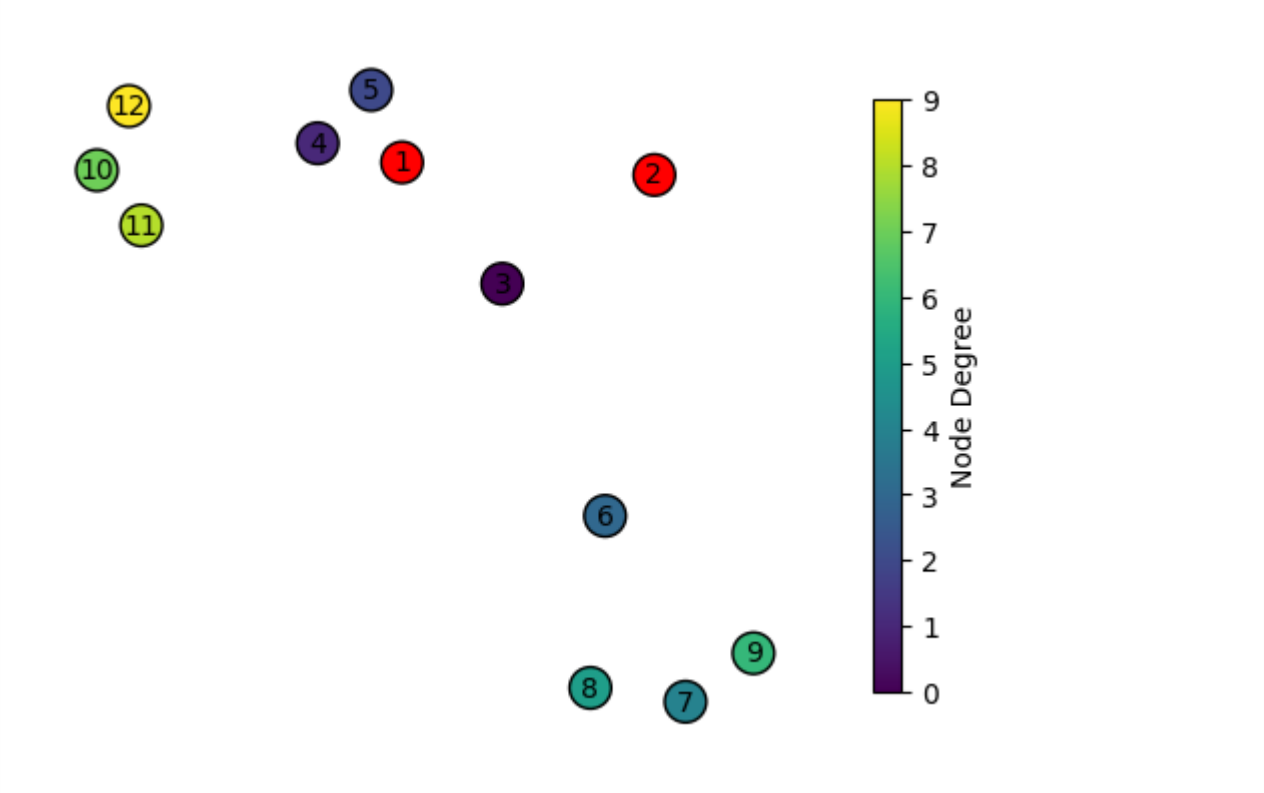
By default, non-finite values (inf, nan) are plotted in the "bad" color of the cmap. Here, it is white by default:

[13]:
```
values = [np.inf, np.nan] + list(range(H.num_nodes - 2))  # values for node colors

ax, node_collection = xgi.draw_nodes(
    H, pos, node_fc=values, node_fc_cmap="viridis", node_labels=True, node_size=15
)

plt.colorbar(node_collection, label="Node Degree", shrink=0.8)
plt.show()
```



The default can be changed, for example here to red:

[14]:
```
values = [np.inf, np.nan] + list(range(H.num_nodes - 2))

cmap = plt.cm.viridis
cmap.set_bad("red")  # set bad

ax, node_collection = xgi.draw_nodes(
    H, pos, node_fc=values, node_fc_cmap=cmap, node_labels=True, node_size=15
)

plt.colorbar(node_collection, label="Node Degree", shrink=0.8)
plt.show()
```



[ ]:

11.3.2 In depth 2 - Drawing hyperedges

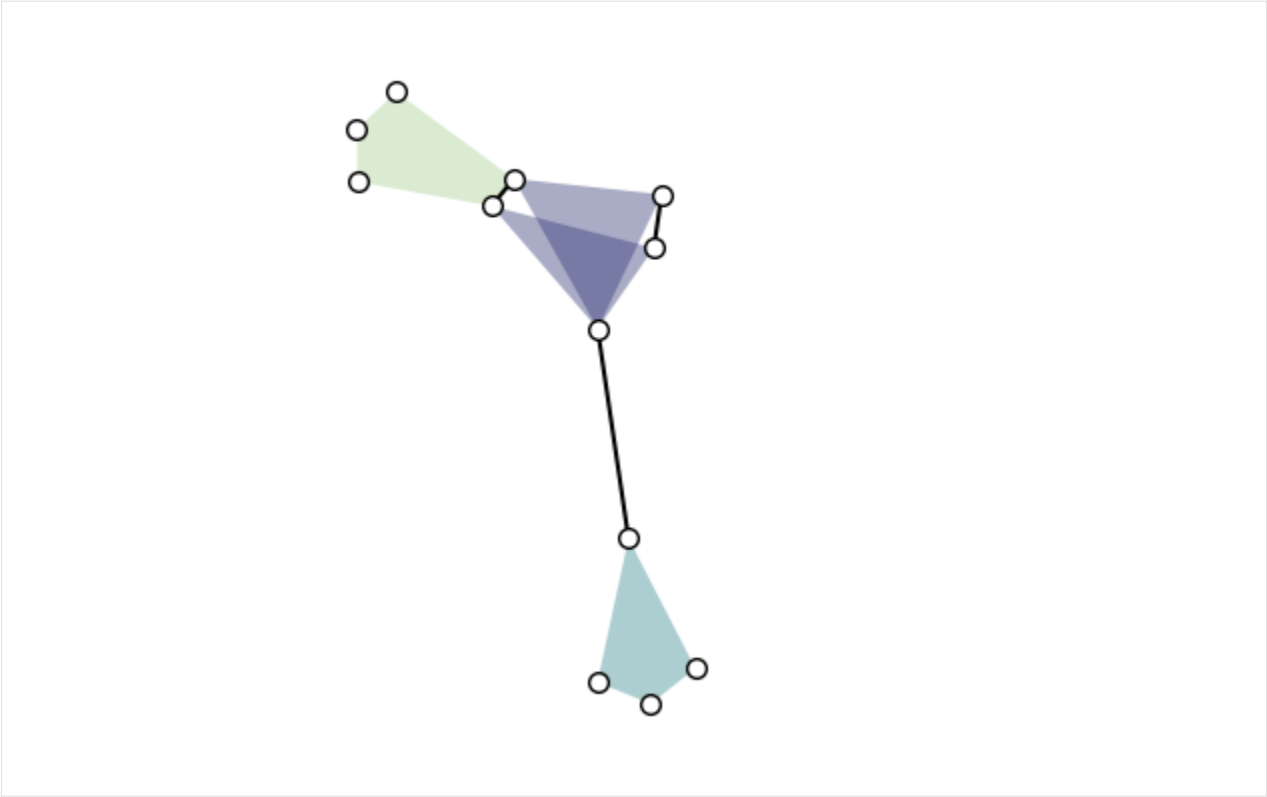
Here we show the fuctionalities and parameters of `xgi.draw_hyperedges()`. It is similar to the networkx corresponding function (+ some bonus) and heavily relies on matplotlib's Collection plotting.

[1]:
```
import matplotlib.pyplot as plt

import xgi
```

Les us first create a small toy hypergraph containing edges of different sizes.

[2]:
```
edges = [[1, 2, 3], [3, 4, 5], [3, 6], [6, 7, 8, 9], [1, 4, 10, 11, 12], [1, 4], [2, 5]]
H = xgi.Hypergraph(edges)

pos = xgi.barycenter_spring_layout(H, seed=42)  # fix position
```

[3]:
```
xgi.draw(H, pos=pos);
```

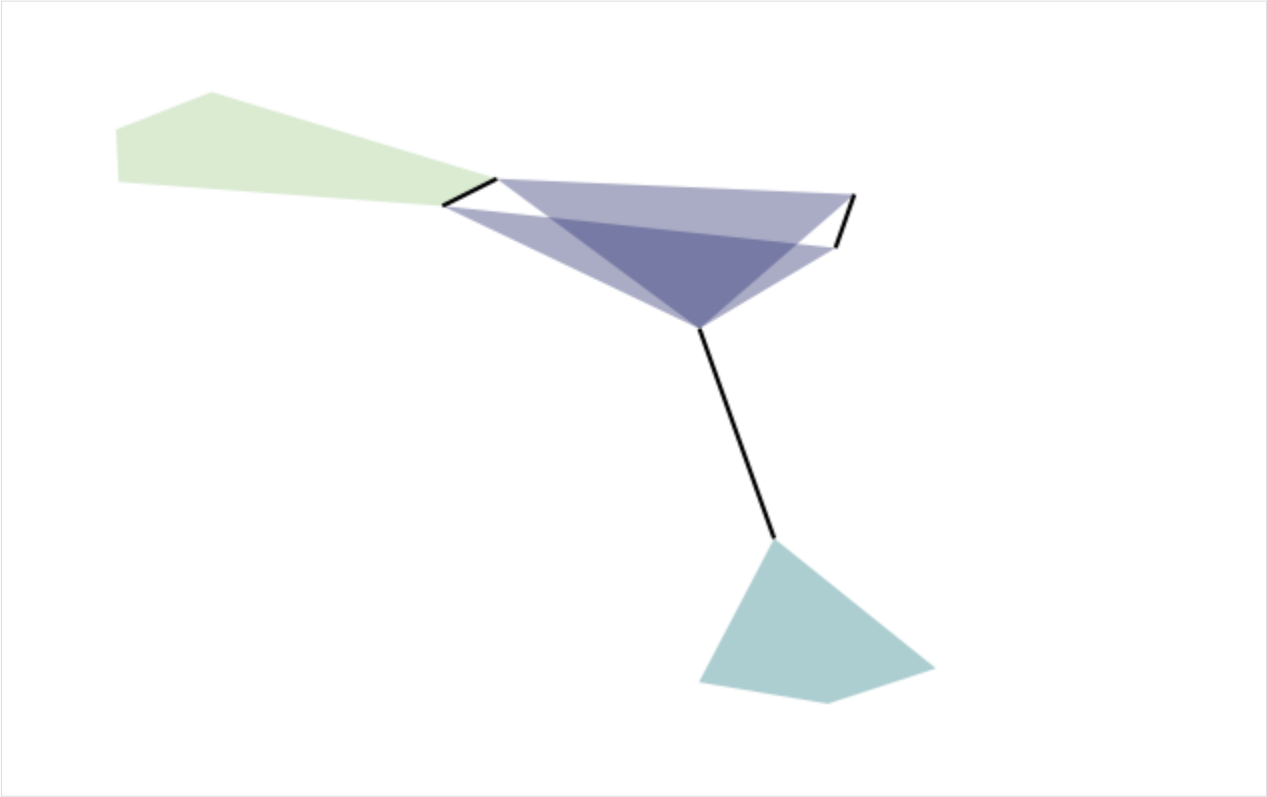


Basics

Let's jump right into how `xgi.draw_hyperedges()` works. By default, it gives:

[4]:
```
xgi.draw_hyperedges(H, pos=pos)
```

[4]:
```
(<Axes: >,
 (<matplotlib.collections.LineCollection at 0x177de6b50>,
  <matplotlib.collections.PatchCollection at 0x177de6fa0>))
```

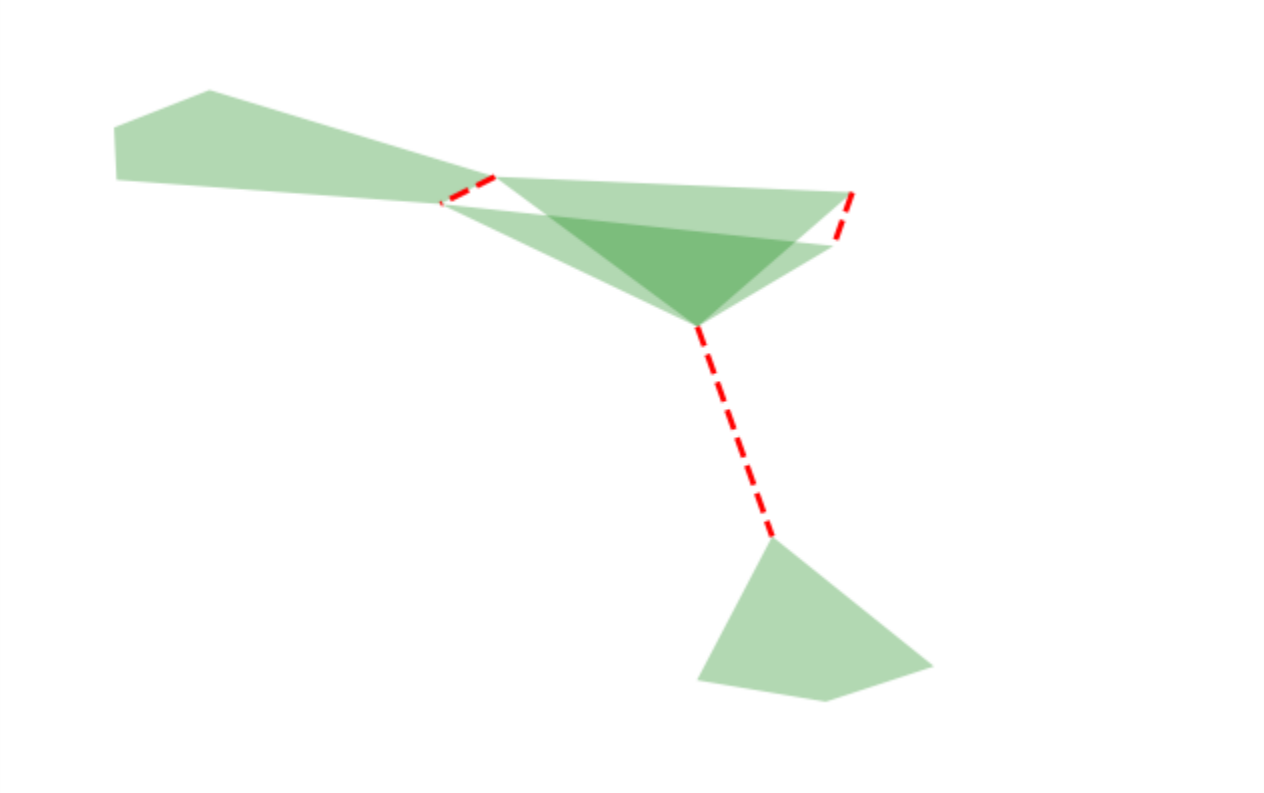


Notice that this function returns a tuple `(ax, collections)` where `collections` is a tuple `(dyad_collection, edge_collection)`. The collections can be used to plot colorbars as we will see later.

The color, linewidth, transparancy, and style of the hyperedges can all be customised, for example with single values:

[5]:
```
xgi.draw_hyperedges(
    H, pos=pos, dyad_color="r", dyad_lw=2, dyad_style="--", edge_fc="g", alpha=0.3
)
```

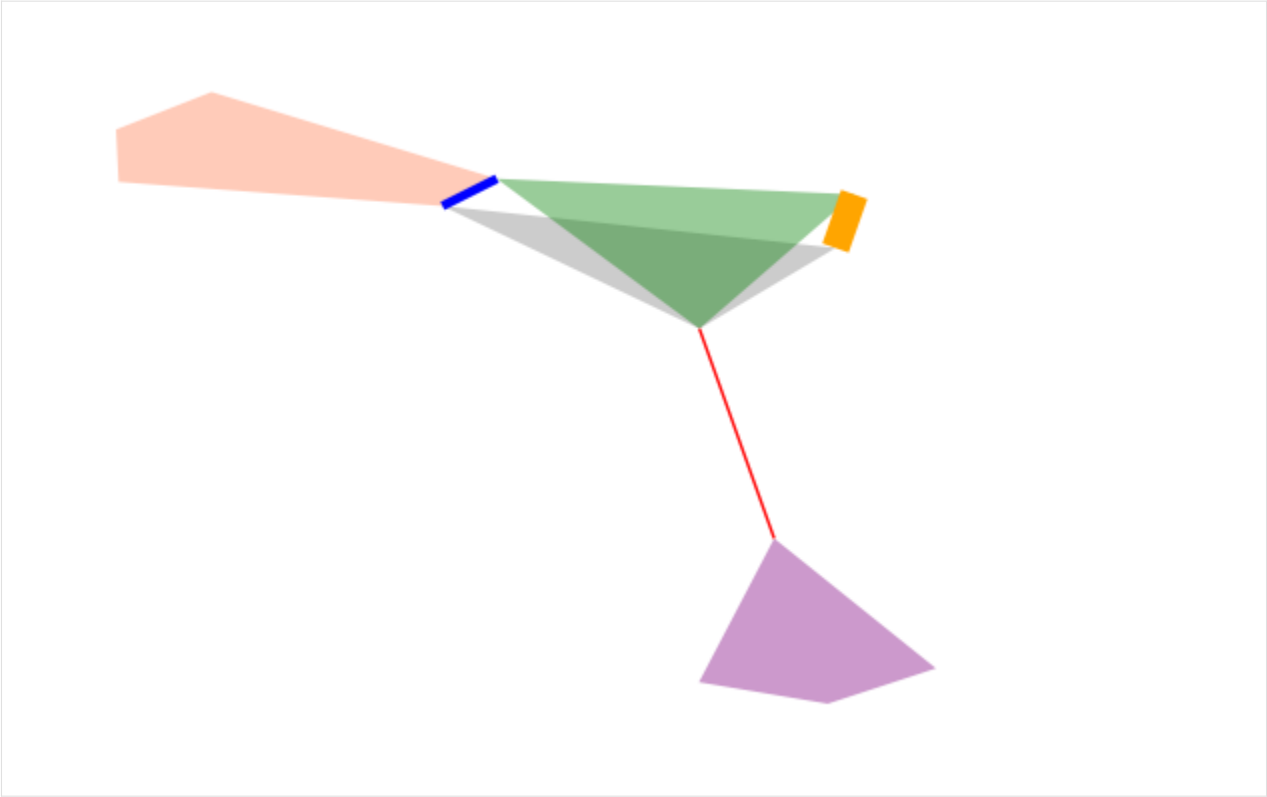
[5]:
```
(<Axes: >,
 (<matplotlib.collections.LineCollection at 0x177e11b80>,
  <matplotlib.collections.PatchCollection at 0x177e3c4c0>))
```



Or with multiple values:

[6]:
```
xgi.draw_hyperedges(
    H,
    pos=pos,
    dyad_color=["r", "b", "orange"],
    dyad_lw=[1, 2, 5],
    edge_fc=["g", "grey", "purple", "coral"],
)
```

[6]:
```
(<Axes: >,
 (<matplotlib.collections.LineCollection at 0x177e5ad60>,
  <matplotlib.collections.PatchCollection at 0x177e70400>))
```

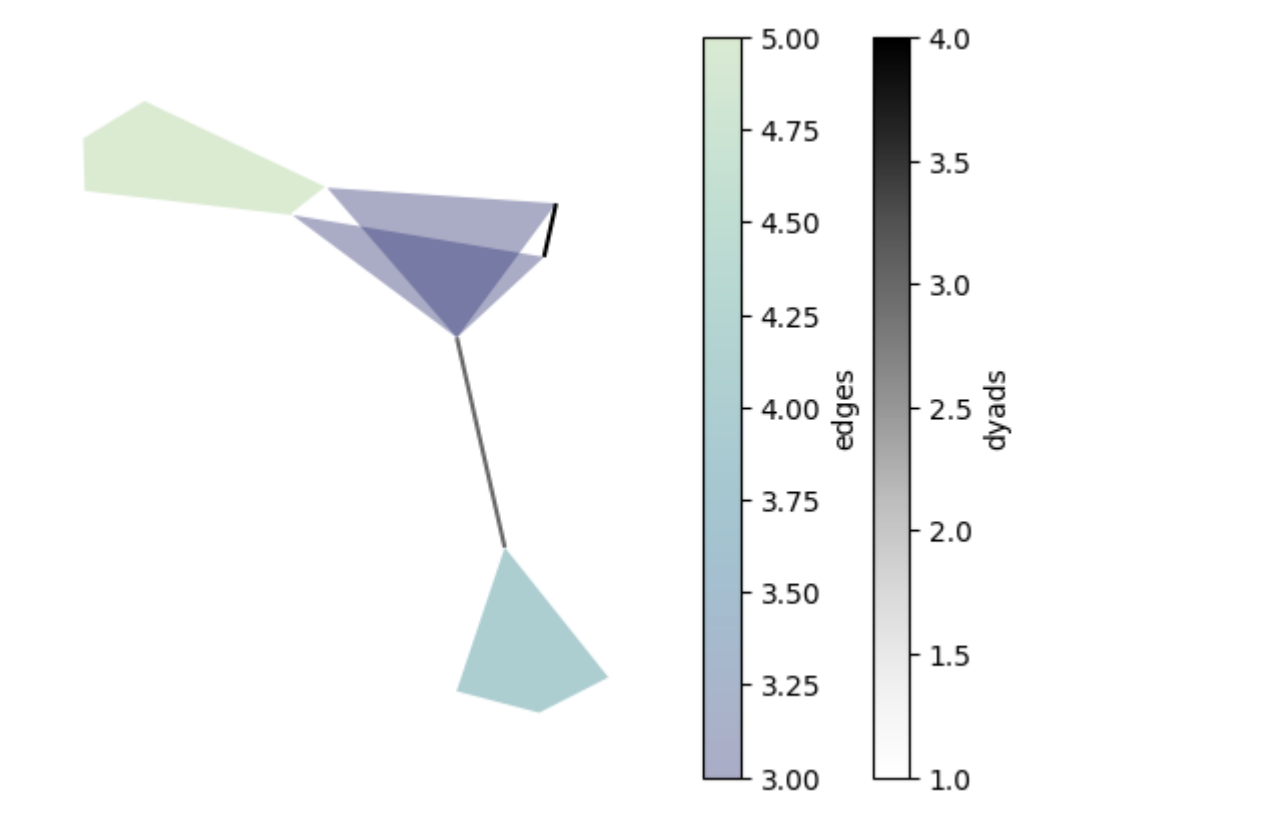


Arrays of floats and colormaps

In XGI, you can easily color hyperedges according to an EdgeStat, or just an array or a dict with float values:

[7]:
```
ax, (dyad_collection, edge_collection) = xgi.draw_hyperedges(
    H,
    pos=pos,
    dyad_color=[3, 1, 4],
    edge_fc=H.edges.size,
)

plt.colorbar(dyad_collection, label="dyads")
plt.colorbar(edge_collection, label="edges")
```

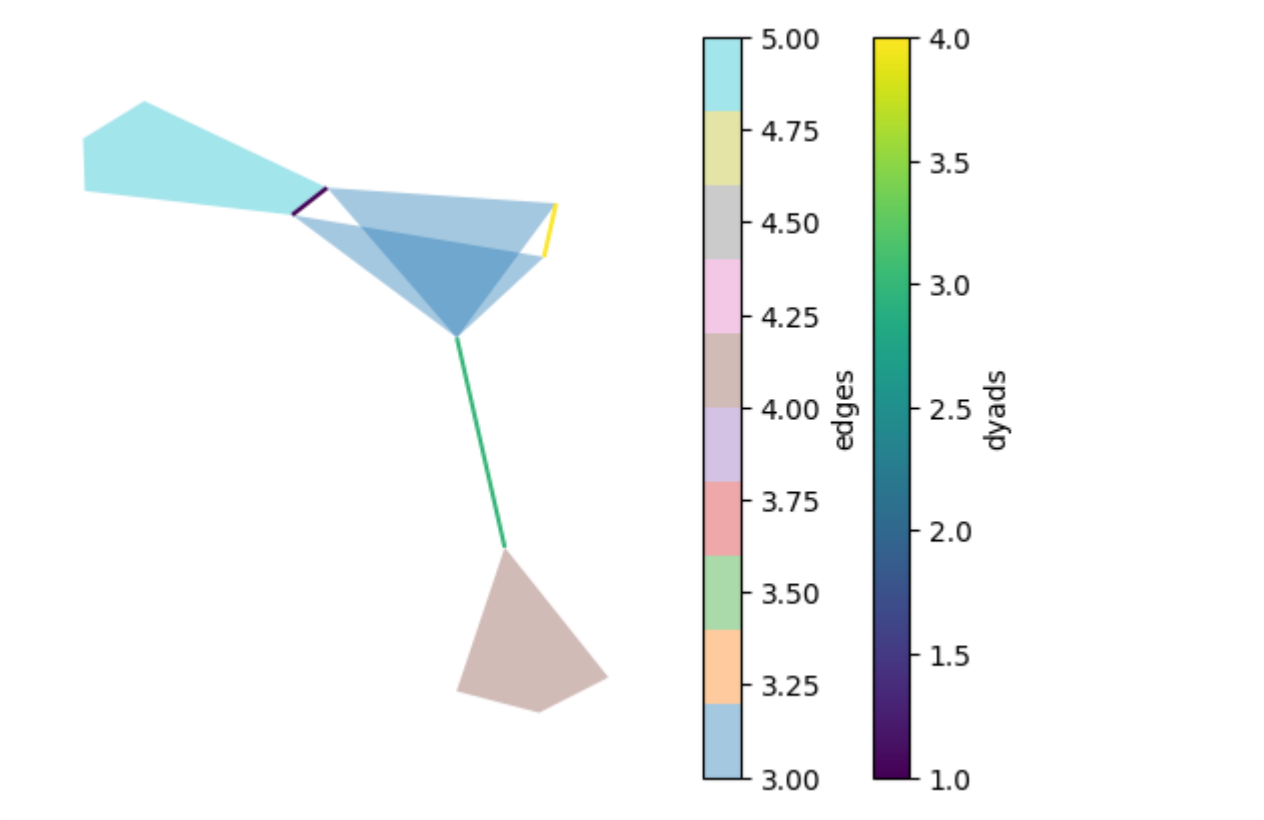
[7]:
```
<matplotlib.colorbar.Colorbar at 0x177f35fa0>
```



By default, the colormaps used are “crest_r” and “Greys”. These can be changed:

[8]:
```
ax, (dyad_collection, edge_collection) = xgi.draw_hyperedges(
    H,
    pos=pos,
    dyad_color=[3, 1, 4],
    edge_fc=H.edges.size,
    dyad_color_cmap="viridis",
    edge_fc_cmap="tab10",
)

plt.colorbar(dyad_collection, label="dyads")
plt.colorbar(edge_collection, label="edges")
```

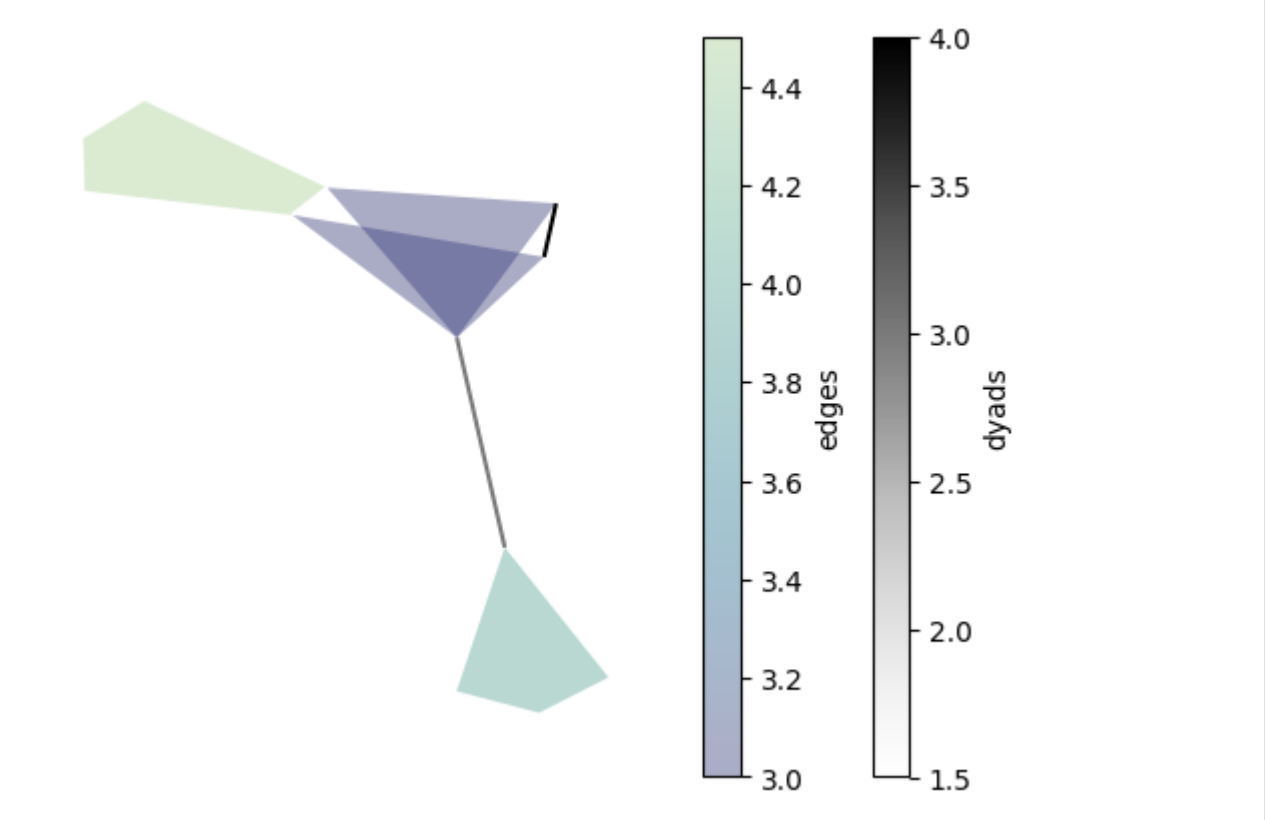
[8]:
```
<matplotlib.colorbar.Colorbar at 0x28805d790>
```



You can specify a `vmin` and `vmax` for the dyad and edge colors:

[9]:
```
ax, (dyad_collection, edge_collection) = xgi.draw_hyperedges(
    H, pos=pos, dyad_color=[3, 1, 4], edge_fc=H.edges.size, dyad_vmin=1.5, edge_vmax=4.5
)

plt.colorbar(dyad_collection, label="dyads")
plt.colorbar(edge_collection, label="edges")
```

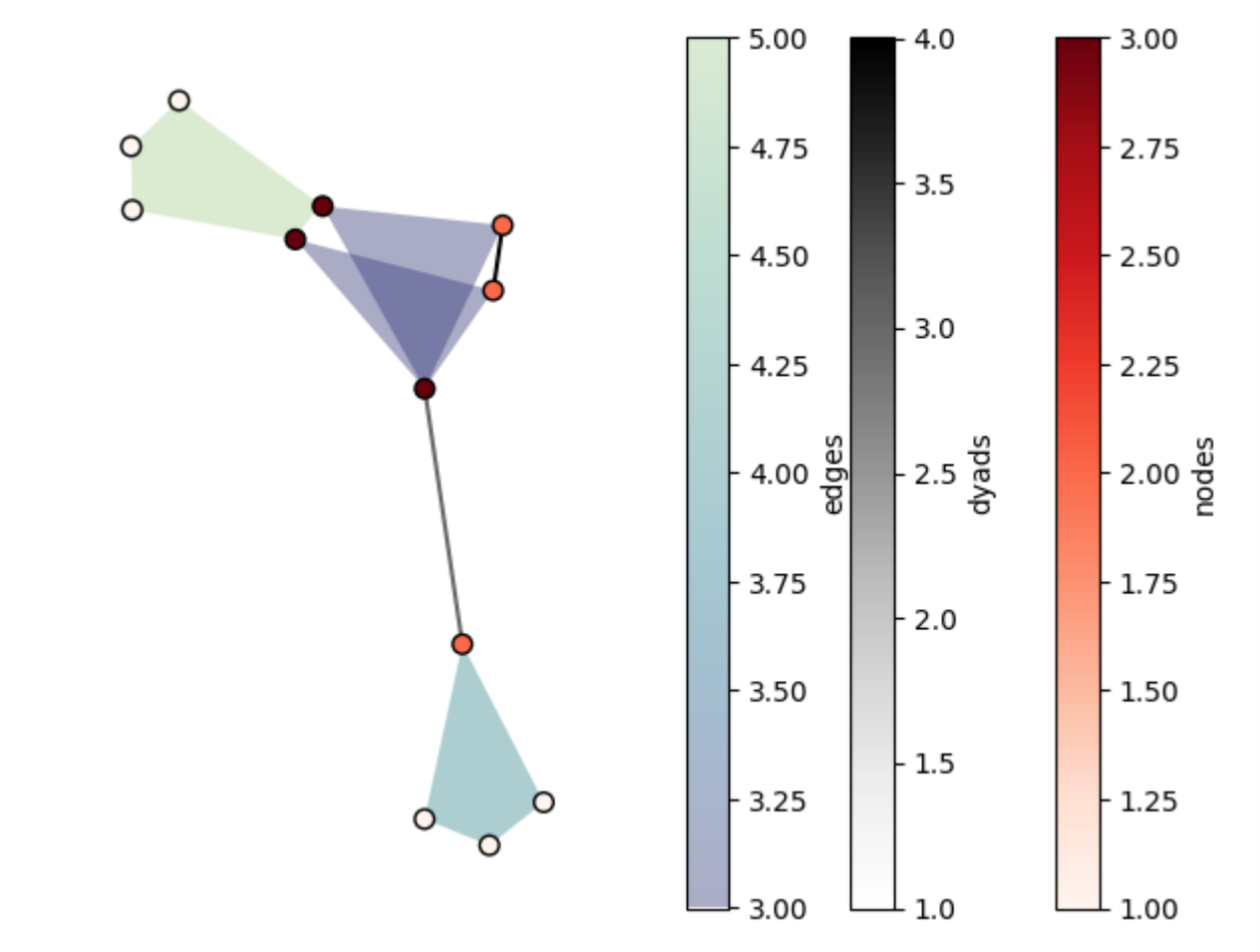
[9]:
```
<matplotlib.colorbar.Colorbar at 0x288166b20>
```



Combine with nodes

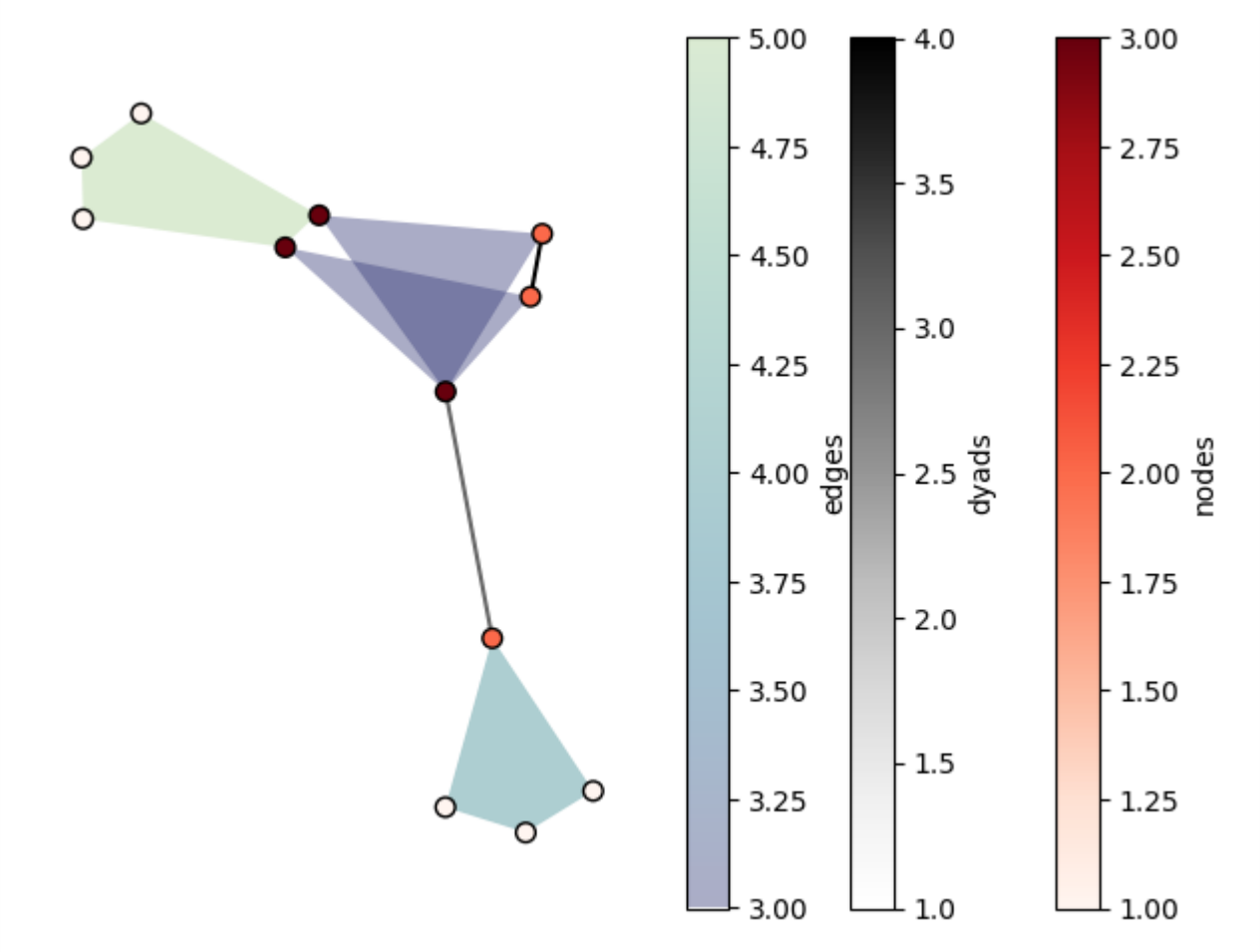
This can be done either via `xgi.draw()`:

[10]:
```
ax, collections = xgi.draw(
    H,
    pos=pos,
    node_fc=H.nodes.degree,
    dyad_color=[3, 1, 4],
    edge_fc=H.edges.size,
)

(node_collection, dyad_collection, edge_collection) = collections
plt.colorbar(node_collection, label="nodes")
plt.colorbar(dyad_collection, label="dyads")
plt.colorbar(edge_collection, label="edges")

plt.tight_layout()
```



or plotting the nodes and the edges separately:

[11]:
```
ax, (dyad_collection, edge_collection) = xgi.draw_hyperedges(
    H, pos=pos, dyad_color=[3, 1, 4], edge_fc=H.edges.size
)

ax, node_collection = xgi.draw_nodes(H, pos=pos, node_fc=H.nodes.degree, zorder=3)

plt.colorbar(node_collection, label="nodes")
plt.colorbar(dyad_collection, label="dyads")
plt.colorbar(edge_collection, label="edges")

plt.tight_layout()
```



11.3.3 In Depth 3 - Drawing DiHypergraphs

Here we show the fuctionalities and parameters of `xgi.draw_bipartite()`. It is similar to the networkx corresponding function (+ some bonus) and heavily relies on matplotlib's Collection plotting.

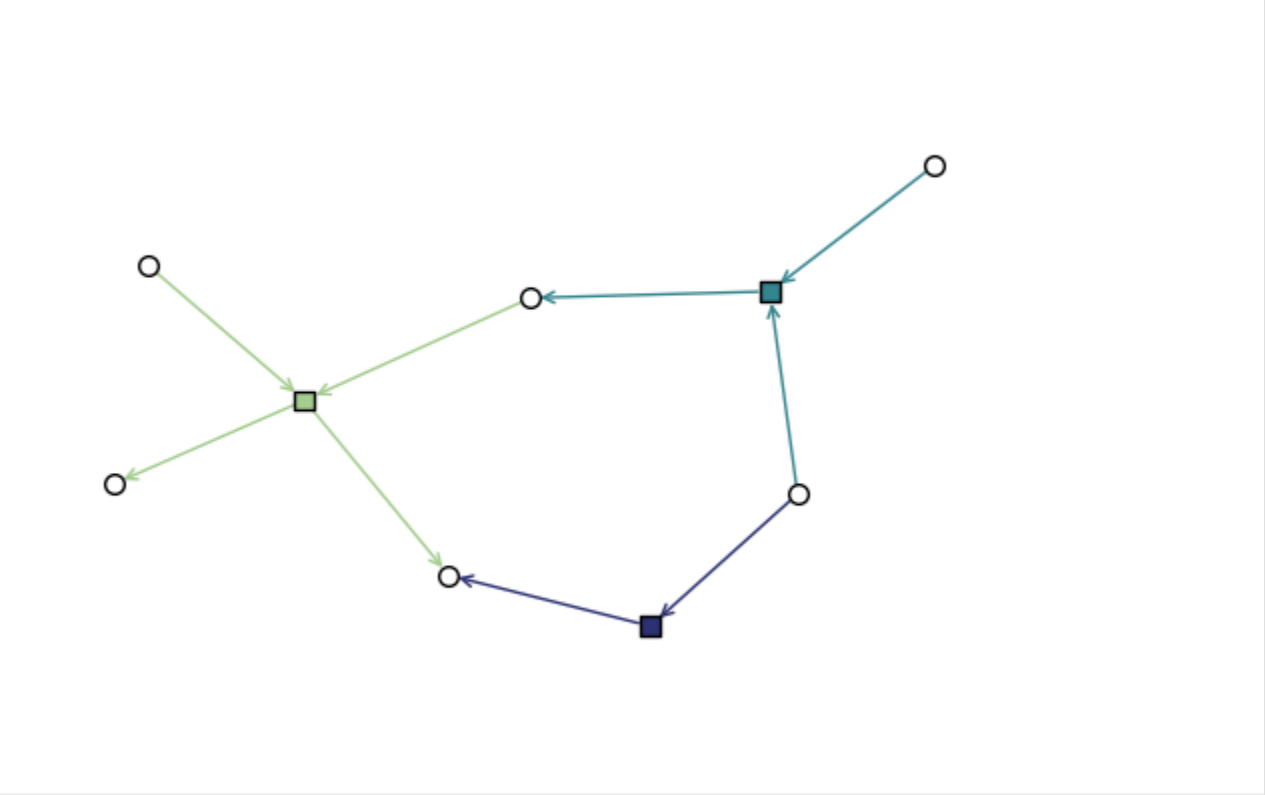
[1]:
```
import matplotlib.pyplot as plt

import xgi
```

Les us first create a small toy hypergraph containing edges of different sizes.

[2]:
```
diedges = [({0, 1}, {2}), ({1}, {4}), ({2, 3}, {4, 5})]
DH = xgi.DiHypergraph(diedges)
```

[3]:
```
xgi.draw_bipartite(DH)
```

[3]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x2a0b7da80>,
  <matplotlib.collections.PathCollection at 0x2a0b7dc00>))
```



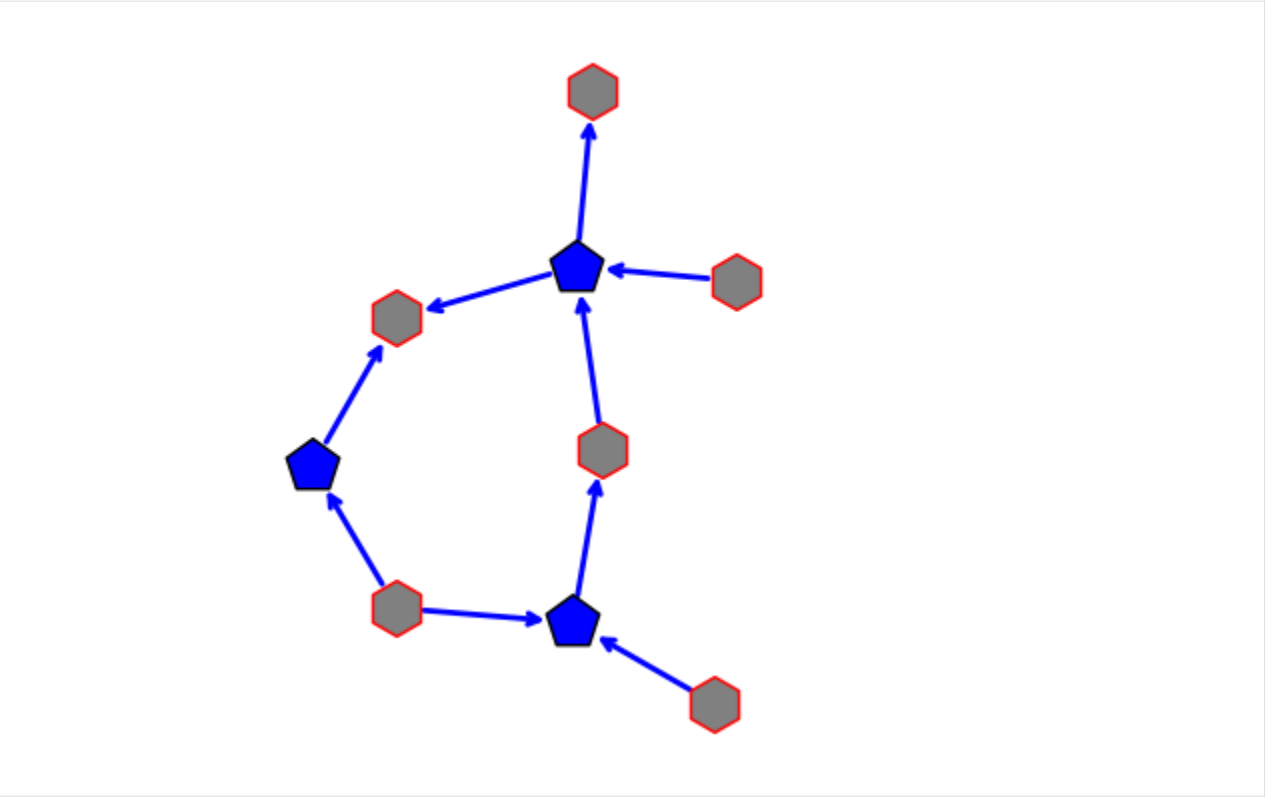
Basics

Notice that this function returns a tuple `(ax, collections)` where `collections` is a tuple `(node_collection, edge_node_collection)`. The collections can be used to plot colorbars as we will see later.

The color, linewidth, transparancy, and style of the hyperedges can all be customised, for example with single values:

[4]:
```
xgi.draw_bipartite(
    DH,
    node_shape="h",
    node_fc="grey",
    node_ec="r",
    node_size=20,
    edge_marker_size=20,
    edge_marker_shape="p",
    dyad_lw=2,
    arrowsize=10,
    edge_marker_fc="b",
    dyad_color="b",
)
```

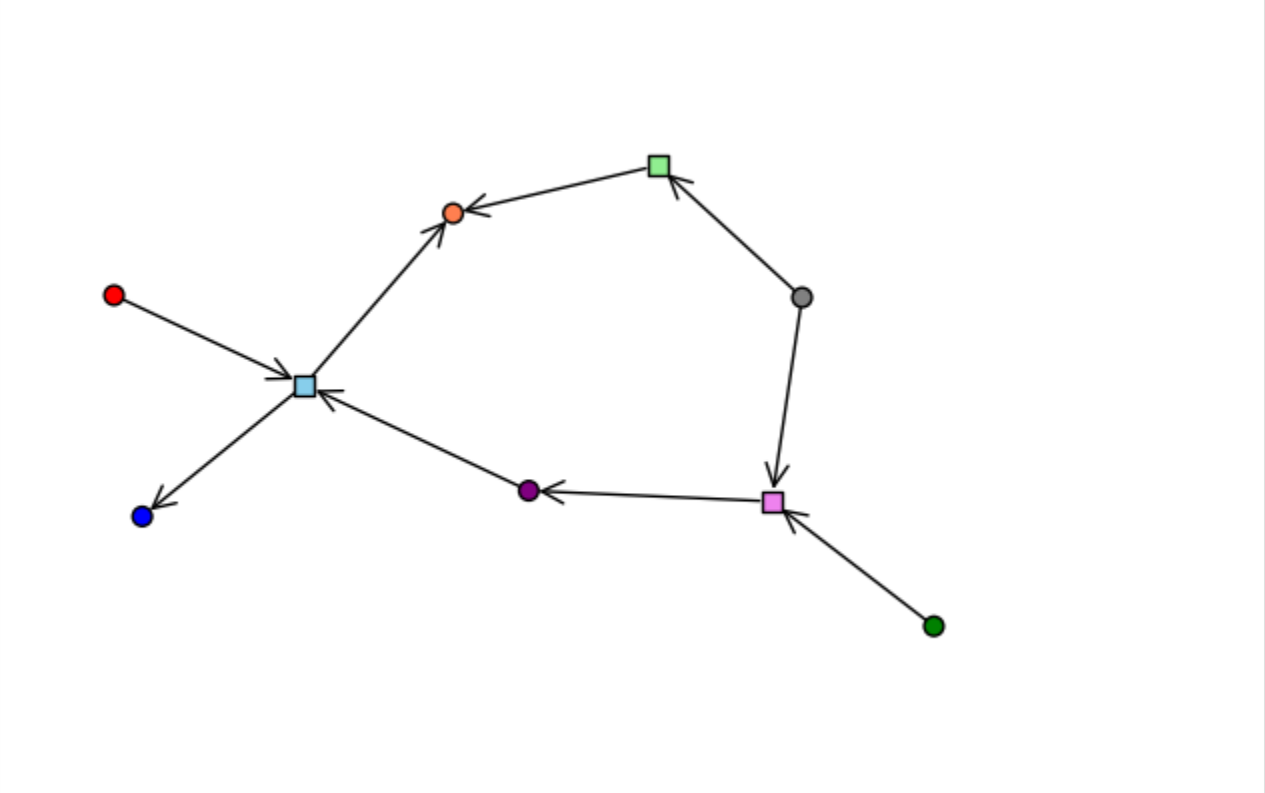
[4]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x2a0cf9b40>,
  <matplotlib.collections.PathCollection at 0x2a0b4faf0>))
```



Or with multiple values:

[5]:
```
xgi.draw_bipartite(
    DH,
    node_fc=["g", "grey", "purple", "coral", "r", "b"],
    arrowsize=20,
    edge_marker_fc=["violet", "lightgreen", "skyblue"],
    dyad_color="k",
)
```

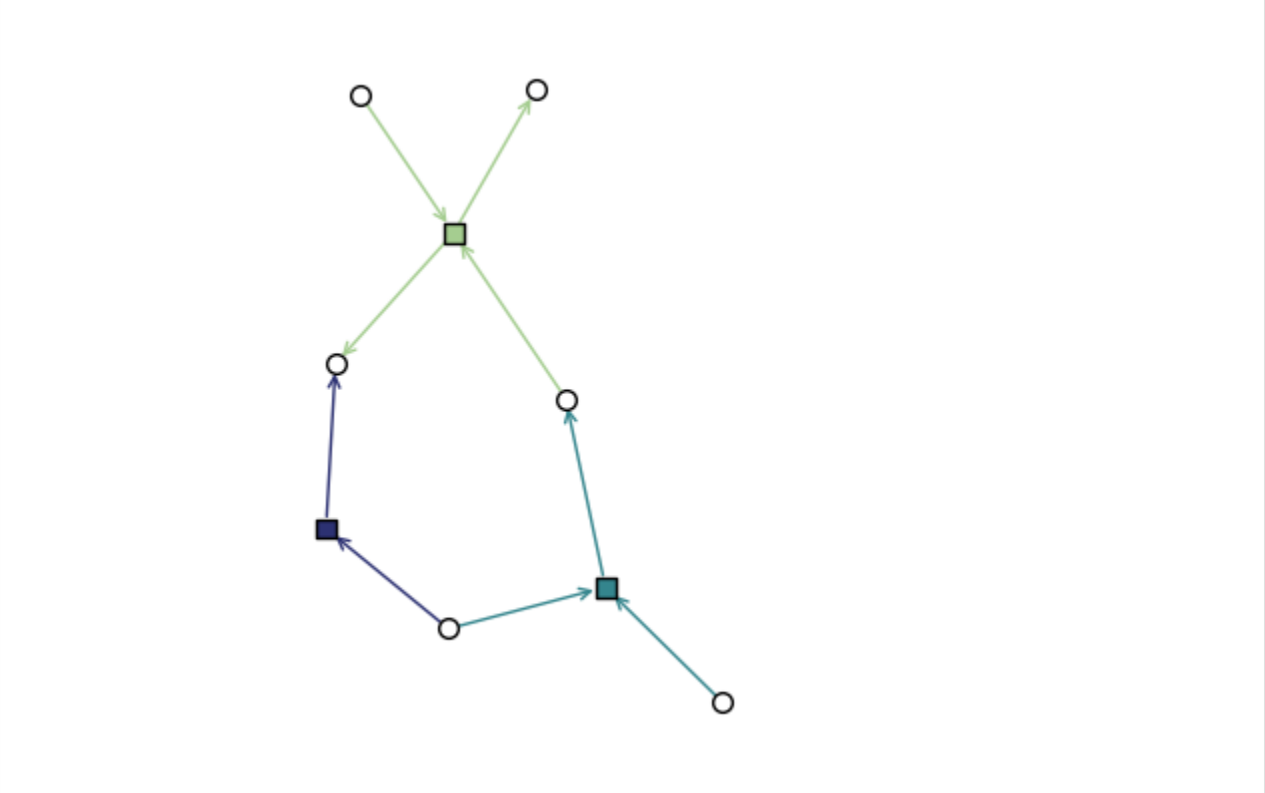
[5]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x2a0d69480>,
  <matplotlib.collections.PathCollection at 0x109d574f0>))
```



Adding node and edge labels:

[6]:
```
xgi.draw_bipartite(DH)
```

[6]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x2a0dd0f70>,
  <matplotlib.collections.PathCollection at 0x2a0cd47c0>))
```

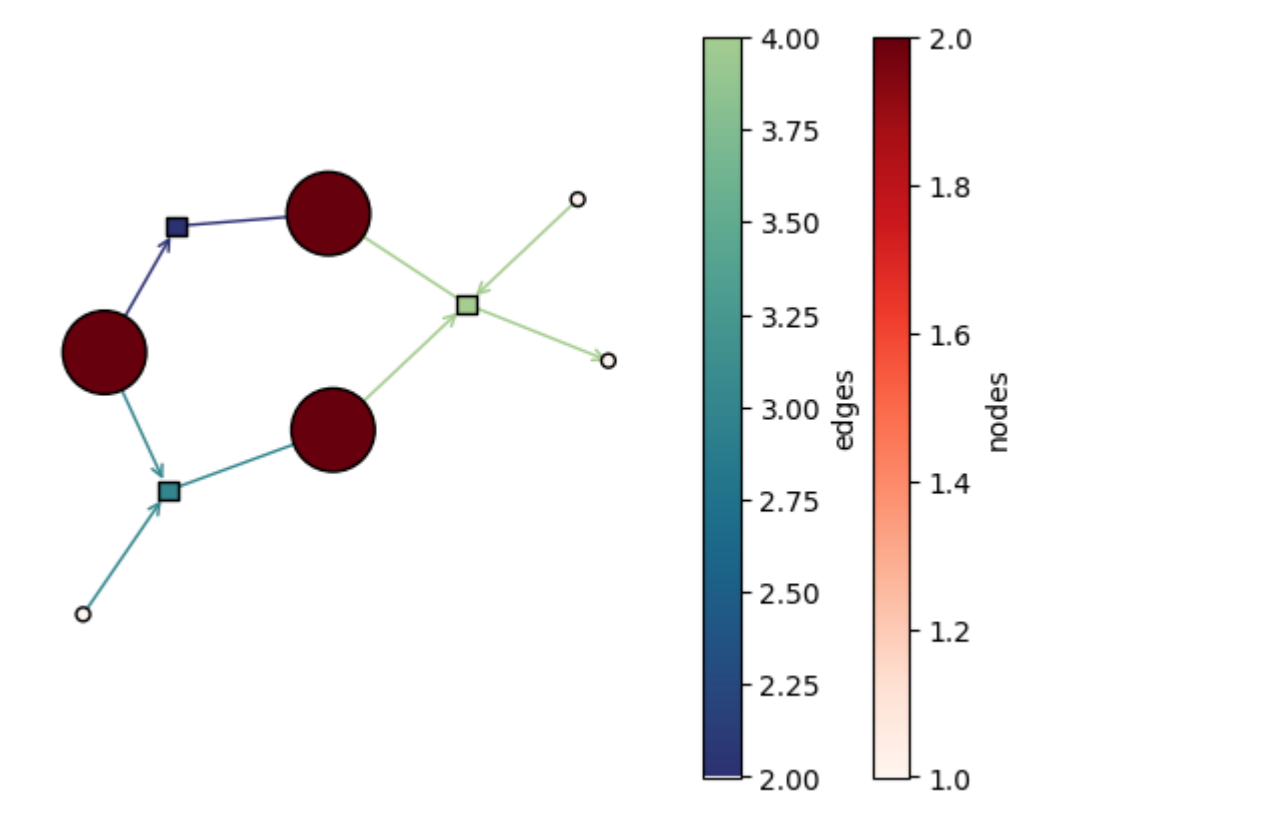


Arrays of floats and colormaps

In XGI, you can easily color hyperedges according to an EdgeStat, or just an array or a dict with float values:

[7]:
```
ax, collections = xgi.draw_bipartite(
    DH,
    node_fc=DH.nodes.degree,
    edge_marker_fc=DH.edges.size,
    node_size=DH.nodes.degree,
)

node_coll, edge_marker_coll = collections

plt.colorbar(node_coll, label="nodes")
plt.colorbar(edge_marker_coll, label="edges")
```

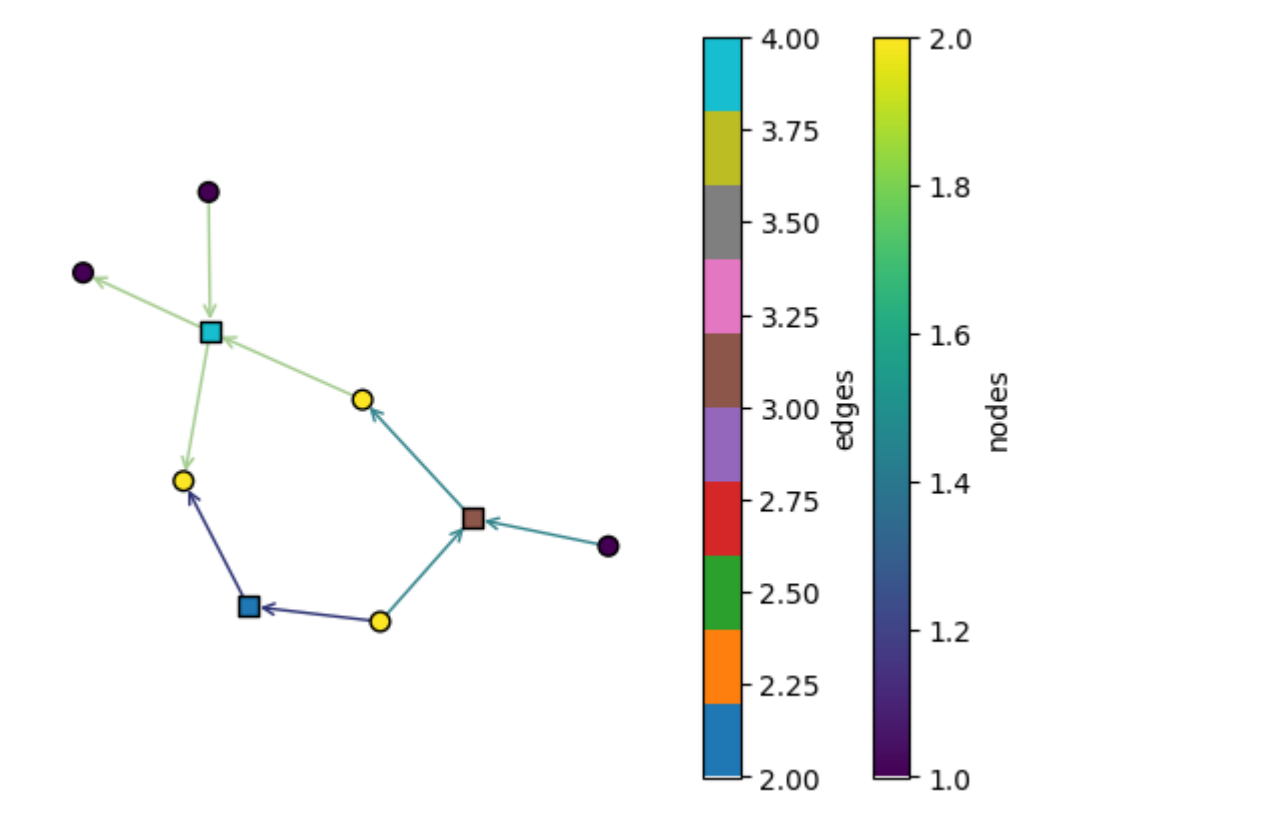
[7]:
```
<matplotlib.colorbar.Colorbar at 0x2a0ea4160>
```



By default, the colormaps used are “crest_r” and “Reds”. These can be changed:

[8]:
```
ax, collections = xgi.draw_bipartite(
    DH,
    node_fc=DH.nodes.degree,
    edge_marker_fc=DH.edges.size,
    node_fc_cmap="viridis",
    edge_marker_fc_cmap="tab10",
)

node_coll, edge_marker_coll = collections

plt.colorbar(node_coll, label="nodes")
plt.colorbar(edge_marker_coll, label="edges")
```

[8]:
```
<matplotlib.colorbar.Colorbar at 0x2a0fba560>
```

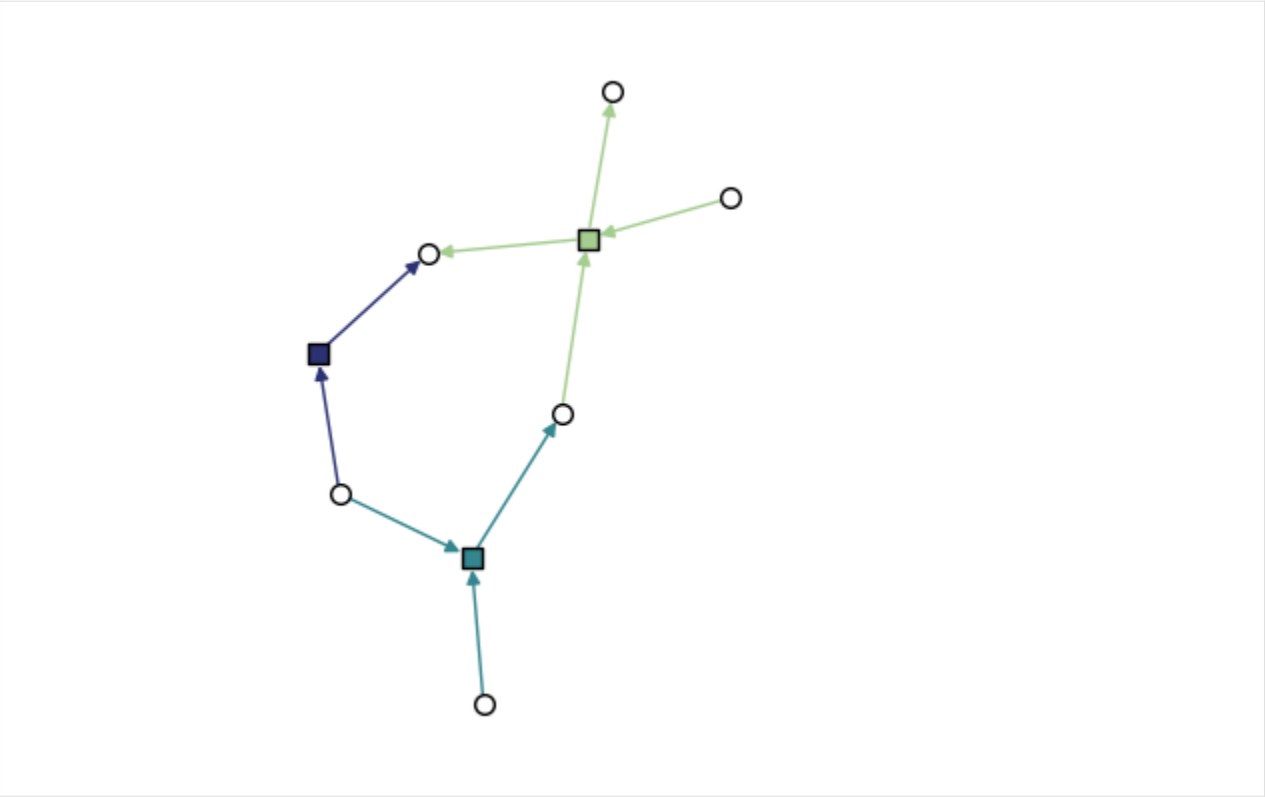


Styling of arrows

By default, the `arrowstyle` used is "-|>":

[9]:
```
xgi.draw_bipartite(DH, arrowstyle="-|>")
```

[9]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x2a1045450>,
  <matplotlib.collections.PathCollection at 0x2a0cd40d0>))
```

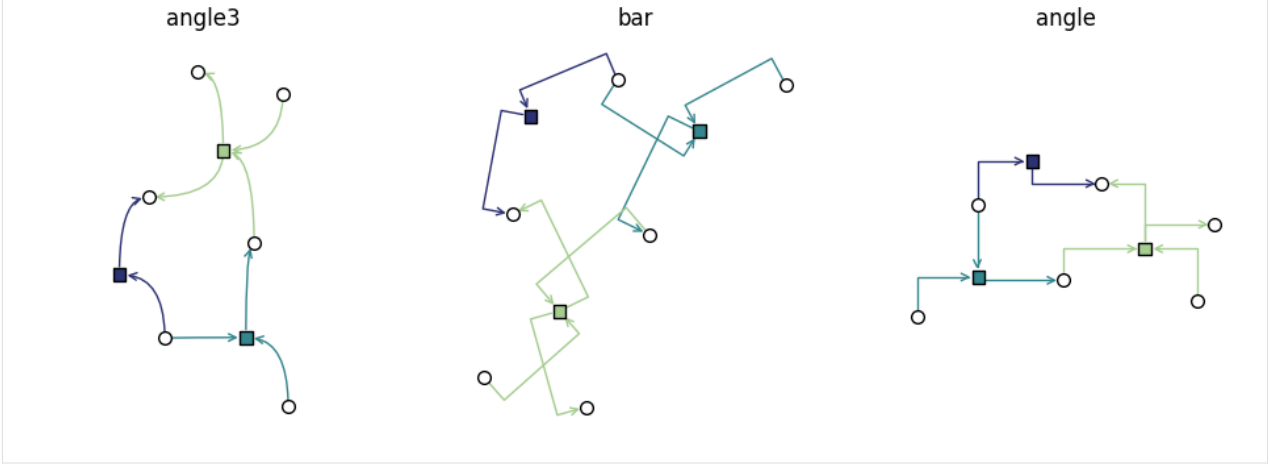


Other styles can be used, see the full list from Matplotlib

[10]:
```
styles = ["simple", "fancy", "]->"]

fig, axs = plt.subplots(1, len(styles), figsize=(10, 3))

for i, style in enumerate(styles):
    ax = axs[i]
    xgi.draw_bipartite(DH, arrowstyle=style, ax=ax)
    ax.set_title(f"{style}")
```

There is a second argument to style arrows: `connectionstyle`. The default value is `"arc3"`, but other values can be used

[11]:
```
styles = ["angle3", "bar", "angle"]

fig, axs = plt.subplots(1, len(styles), figsize=(12, 4))

for i, style in enumerate(styles):
    ax = axs[i]
    xgi.draw_bipartite(DH, connectionstyle=style, ax=ax)
    ax.set_title(f"{style}")
```

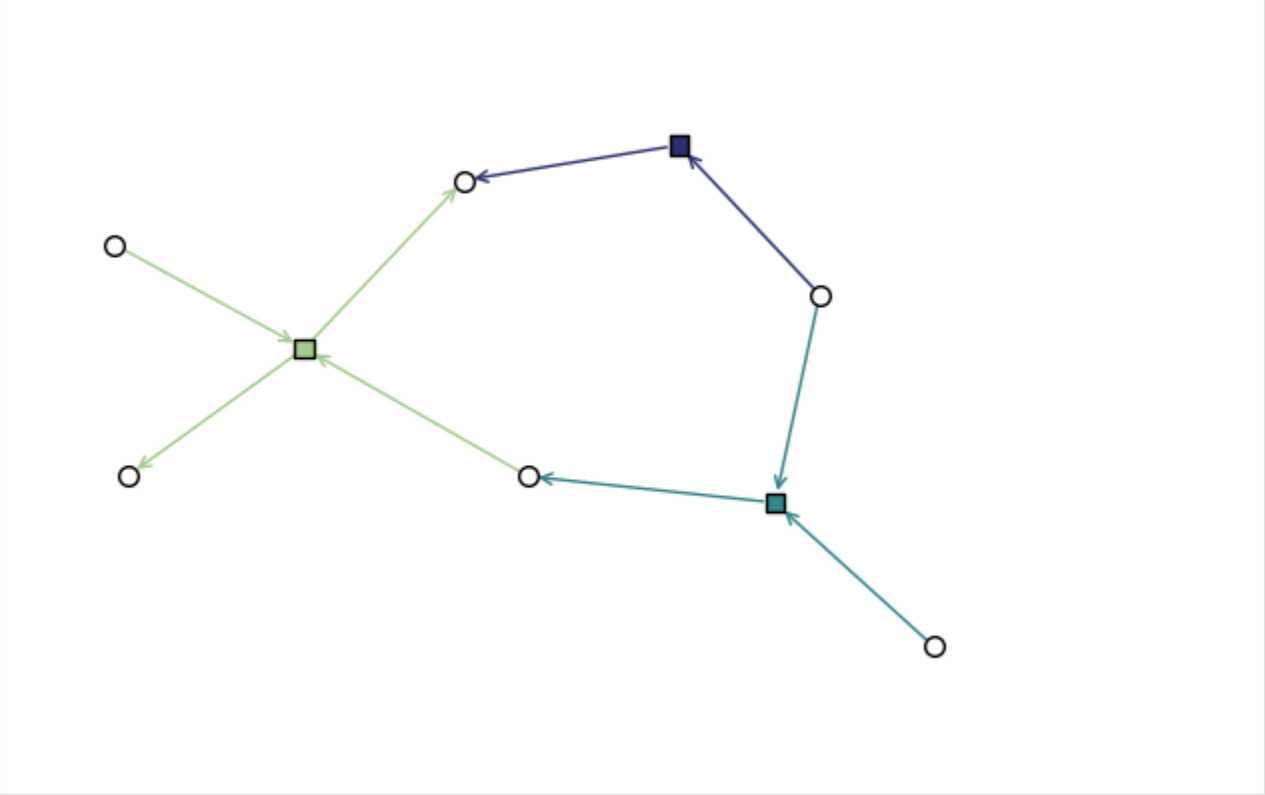


Layout

The initial layout is computed based on the bipartite network representation of the hypergraph:

[12]:
```
pos = xgi.bipartite_spring_layout(DH)
xgi.draw_bipartite(DH, pos=pos)
```

[12]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x2a1088430>,
  <matplotlib.collections.PathCollection at 0x2a12e8af0>))
```

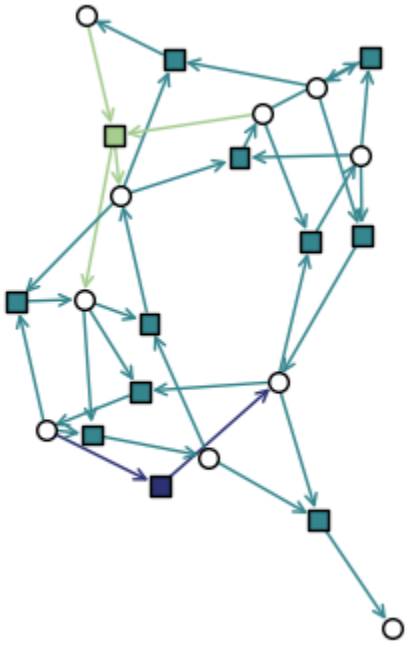


A larger example

[13]:
```
edges = [
    [[8], [0]],
    [[1, 2], [0]],
    [[0, 3], [1]],
    [[1, 3], [2]],
    [[1, 5], [3]],
    [[2, 5], [4]],
    [[3, 4], [5, 6]],
    [[6, 7], [5]],
    [[5, 8], [6]],
    [[6, 8], [7]],
    [[6, 0], [8]],
    [[7, 0], [9]],
]

DH = xgi.DiHypergraph(edges)
```

[14]:
```
pos = xgi.bipartite_spring_layout(DH)
xgi.draw_bipartite(DH, pos=pos)
```

[14]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x2a134c970>,
  <matplotlib.collections.PathCollection at 0x2a134cdc0>))
```

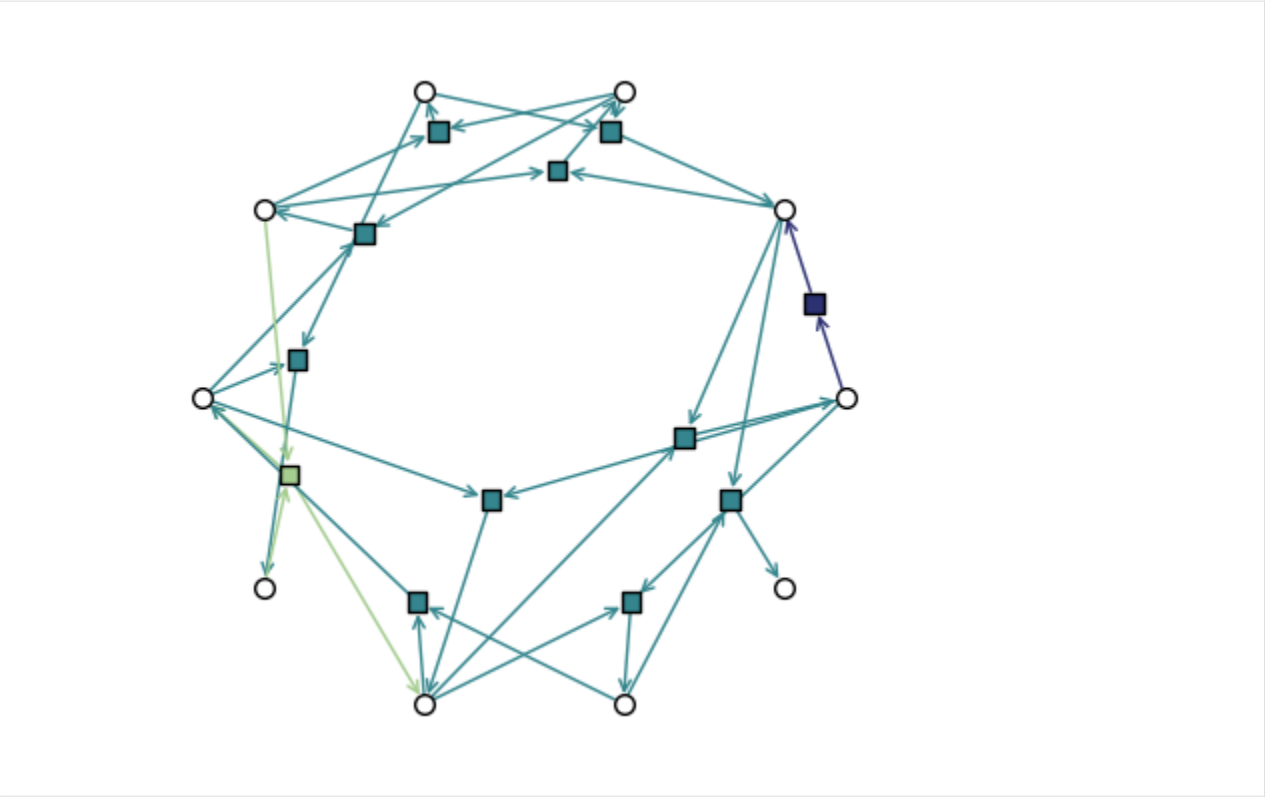


But, we can also place the edge markers at the barycenters of the node positions using `edge_positions_from_barycenters`:

[15]:
```
node_pos = xgi.circular_layout(DH)
edge_pos = xgi.edge_positions_from_barycenters(DH, node_pos)
```

[16]:
```
xgi.draw_bipartite(DH, pos=(node_pos, edge_pos))
```

[16]:
```
(<AxesSubplot: >,
 (<matplotlib.collections.PathCollection at 0x2a13dcd90>,
  <matplotlib.collections.PathCollection at 0x2a13dd1e0>))
```



11.3.4 In Depth 4 - Drawing multilayer-style

Here we show the fuctionalities and parameters of `xgi.draw_multilayer()`.

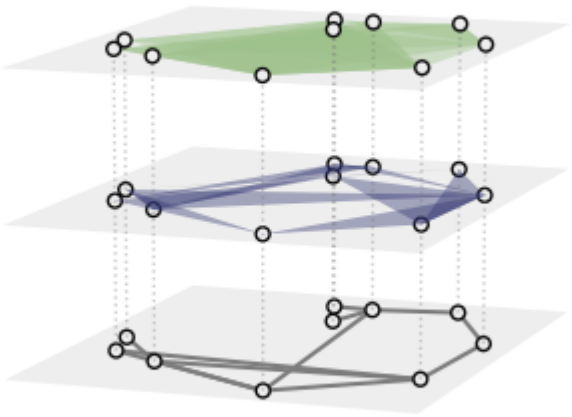
[1]:
```
import matplotlib.pyplot as plt

import xgi
```

Les us first create a small toy hypergraph containing edges of different sizes.

[2]:
```
H = xgi.random_hypergraph(N=10, ps=[0.2, 0.05, 0.05], seed=1)
```

[3]:
```
ax = plt.axes(projection="3d")
xgi.draw_multilayer(H, ax=ax)

plt.show()
```

```
/Users/maxime/Dropbox (ISI Foundation)/WORK/SCIENCE/xgi/xgi/drawing/draw.py:1479:␣
↪UserWarning: No data for colormapping provided via 'c'. Parameters 'cmap' will be␣
↪ignored
  node_collection = ax.scatter(
```

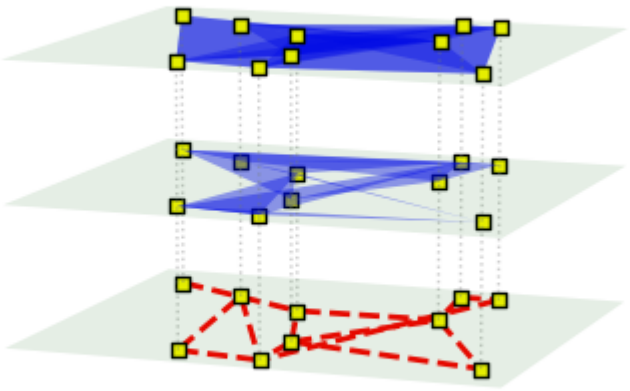


Basics

Notice that this function returns a tuple `(ax, collections)` where `collections` is a tuple `(node_collection, edge_collection)`. The collections can be used to plot colorbars as we will see later.

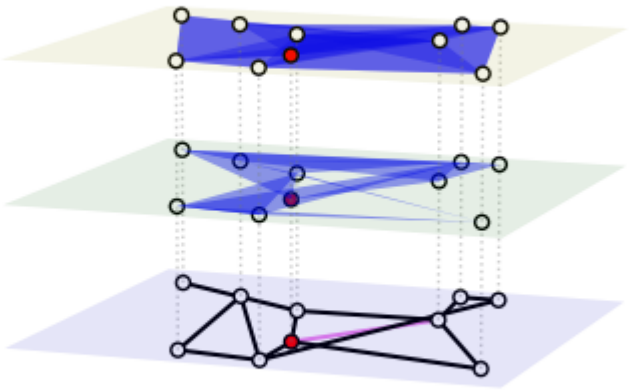
The color, linewidth, transparancy, and style of the hyperedges can all be customised, for example with single values:

[4]:
```
pos = xgi.barycenter_spring_layout(H, seed=1)

ax = plt.axes(projection="3d")
xgi.draw_multilayer(
    H,
    ax=ax,
    pos=pos,
    node_fc="yellow",
    node_shape="s",
    dyad_color="r",
    dyad_style="--",
    dyad_lw=2,
    edge_fc="b",
    layer_color="g",
)

plt.show()
```



Or with multiple values:

[5]:
```
pos = xgi.barycenter_spring_layout(H, seed=1)

ax = plt.axes(projection="3d")
xgi.draw_multilayer(
    H,
    ax=ax,
    pos=pos,
    node_fc=["red"] + ["white"] * 9,
    dyad_color=["violet"] + ["k"] * 12,
    edge_fc="b",
    layer_color=["b", "g", "y"],
)

plt.show()
```

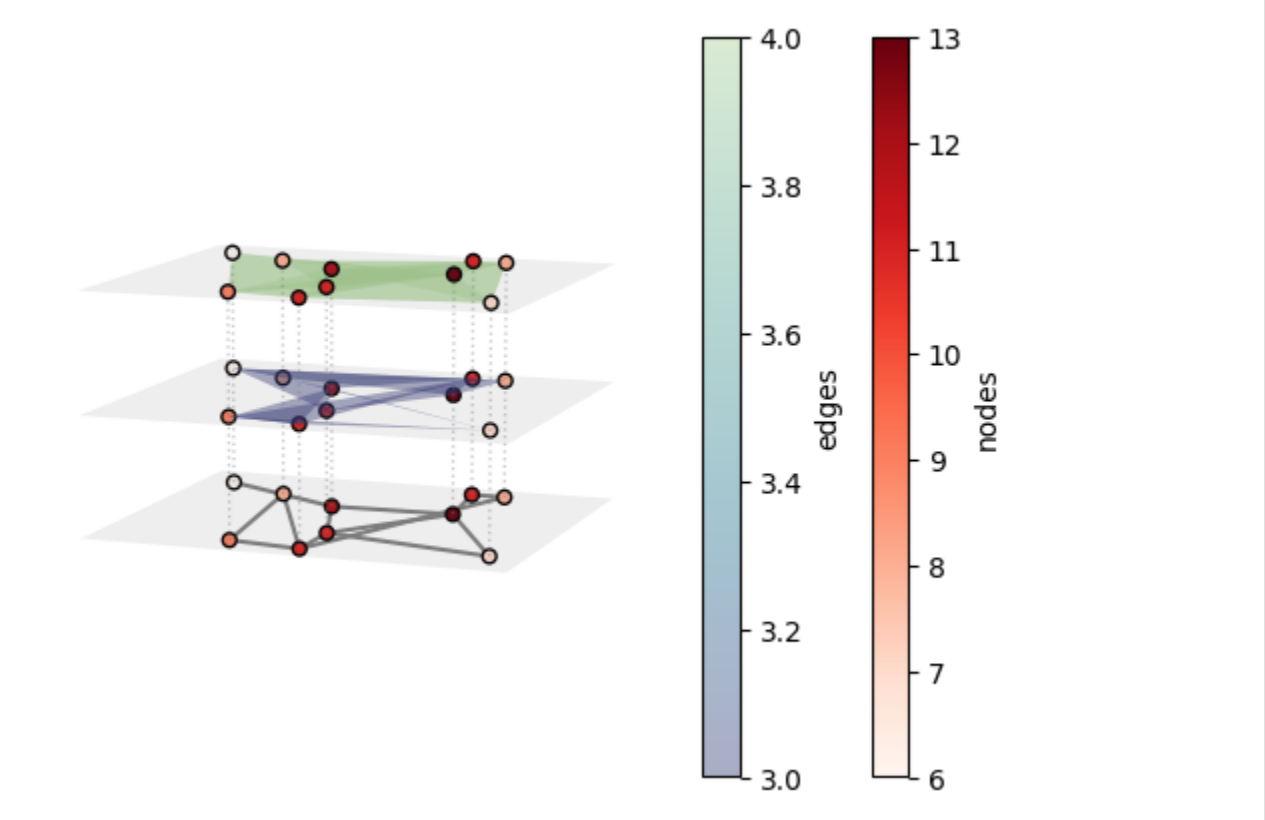


Arrays of floats and colormaps

In XGI, you can easily color hyperedges according to an EdgeStat, or just an array or a dict with float values:

[6]:
```
pos = xgi.barycenter_spring_layout(H, seed=1)

ax = plt.axes(projection="3d")
ax, collections = xgi.draw_multilayer(
    H, ax=ax, pos=pos, edge_fc=H.edges.size, node_fc=H.nodes.degree
)

node_coll, edge_coll = collections

plt.colorbar(node_coll, label="nodes")
plt.colorbar(edge_coll, label="edges")
```

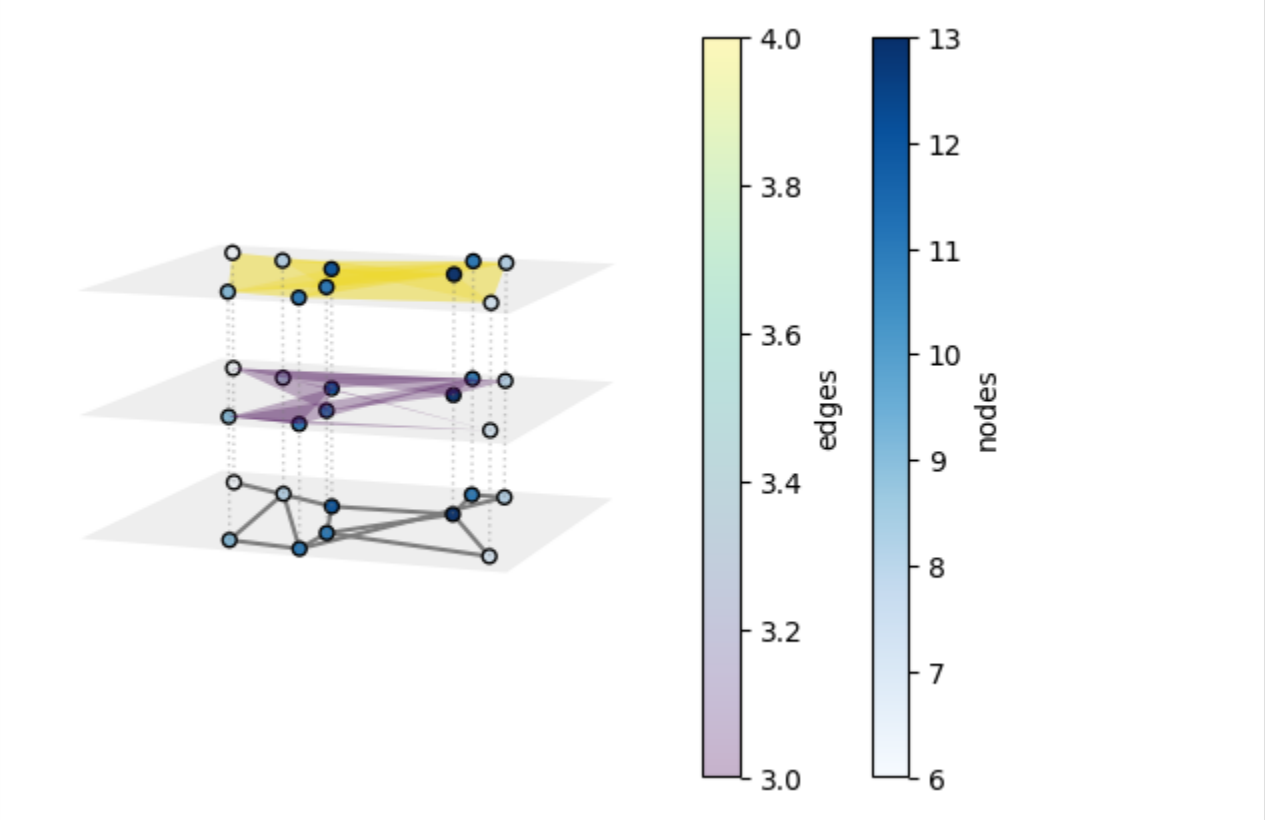
[6]:
```
<matplotlib.colorbar.Colorbar at 0x2877d6b50>
```



By default, the colormaps used are “crest_r” and “Reds”. These can be changed:

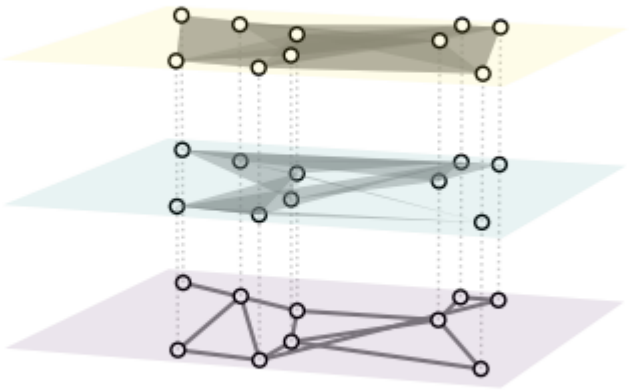
[7]:
```
pos = xgi.barycenter_spring_layout(H, seed=1)

ax = plt.axes(projection="3d")
ax, collections = xgi.draw_multilayer(
    H,
    ax=ax,
    pos=pos,
    edge_fc=H.edges.size,
    node_fc=H.nodes.degree,
    node_fc_cmap="Blues",
    edge_fc_cmap="viridis",
    alpha=0.3,
)

node_coll, edge_coll = collections

plt.colorbar(node_coll, label="nodes")
plt.colorbar(edge_coll, label="edges")
```

[7]:
```
<matplotlib.colorbar.Colorbar at 0x2879bc850>
```



You can even have a cmap for the layers instead:

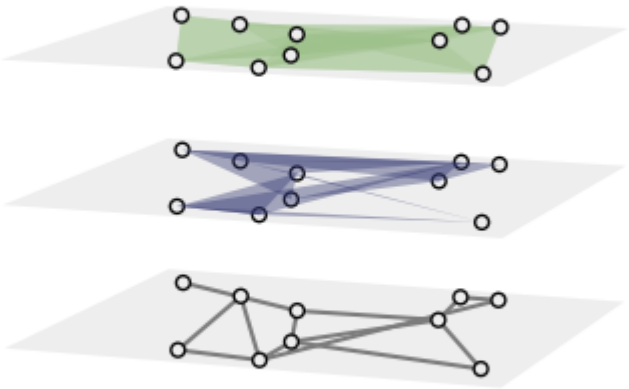
[8]:
```
pos = xgi.barycenter_spring_layout(H, seed=1)

ax = plt.axes(projection="3d")
ax, collections = xgi.draw_multilayer(
    H, ax=ax, pos=pos, edge_fc="grey", layer_color=[2, 3, 4], layer_cmap="viridis"
)

node_coll, edge_coll = collections

plt.show()
```



Styling of interlayer links

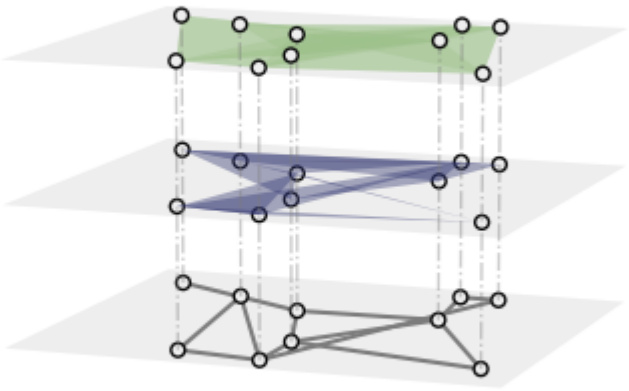
First you can simply remove them with `conn_lines=False`:

[9]:
```
pos = xgi.barycenter_spring_layout(H, seed=1)

ax = plt.axes(projection="3d")
ax, collections = xgi.draw_multilayer(H, ax=ax, pos=pos, conn_lines=False)

node_coll, edge_coll = collections

plt.show()
```



Or you can change their style with `conn_lines_style`:

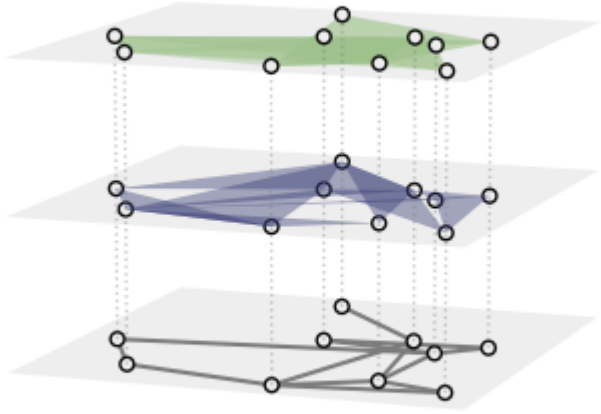
[10]:
```
pos = xgi.barycenter_spring_layout(H, seed=1)

ax = plt.axes(projection="3d")
ax, collections = xgi.draw_multilayer(H, ax=ax, pos=pos, conn_lines_style="-.")

node_coll, edge_coll = collections

plt.show()
```



Set the height between layers and the view angles

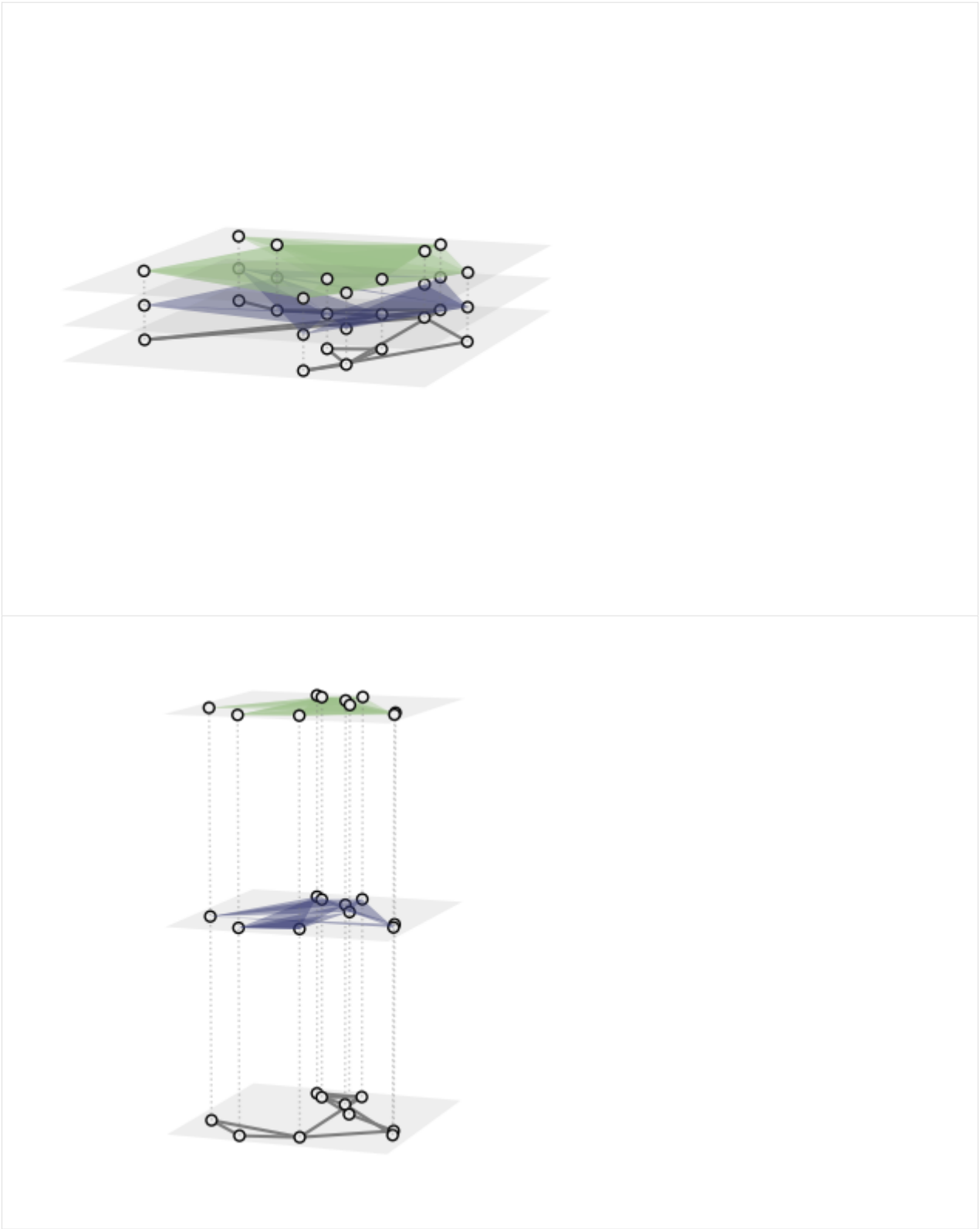
By default the height between layers is `sep=0.4`:

[11]:
```
ax = plt.axes(projection="3d")
xgi.draw_multilayer(H, ax=ax, sep=0.4)

plt.show()
```



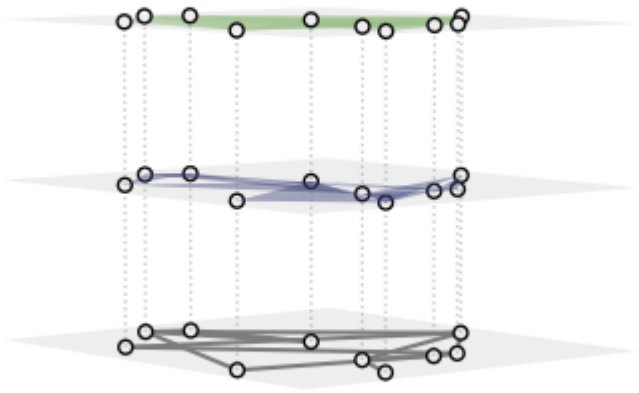
This can be changed!

[12]:
```
ax = plt.axes(projection="3d")
xgi.draw_multilayer(H, ax=ax, sep=0.1)

plt.show()

ax = plt.axes(projection="3d")
xgi.draw_multilayer(H, ax=ax, sep=1)

plt.show()
```

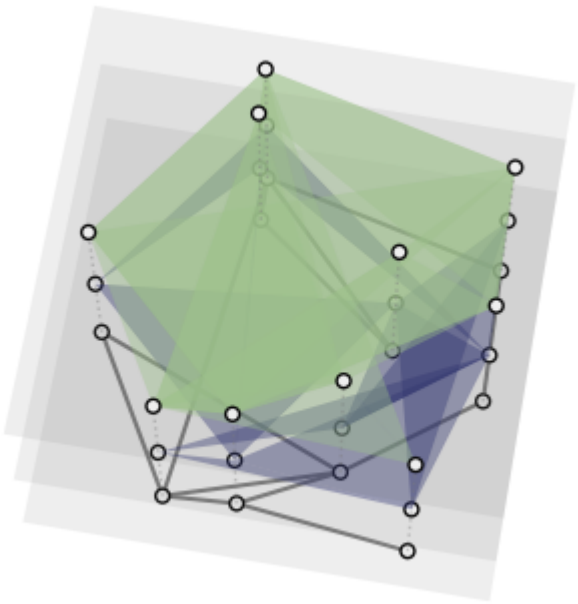


Similarly, the default view angles are `h_angle=10`, `v_angle=20`, but can be changed:

[13]:
```
ax = plt.axes(projection="3d")
xgi.draw_multilayer(H, ax=ax, h_angle=5, v_angle=50)

plt.show()
```



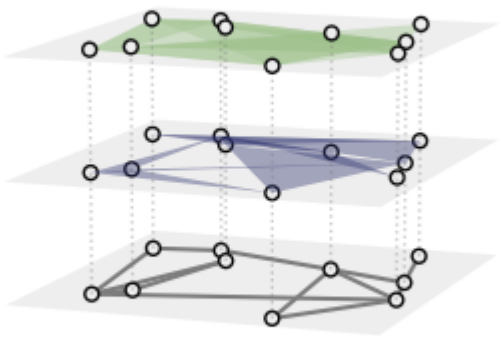
[14]:
```
ax = plt.axes(projection="3d")
xgi.draw_multilayer(H, ax=ax, h_angle=70, v_angle=10)

plt.show()
```



Control axis size

You can control the size of the axis by passing a custom made `Axis3d` like so:

[15]:
```
_, ax = plt.subplots(figsize=(4, 4), subplot_kw={"projection": "3d"})
xgi.draw_multilayer(H, ax=ax)

plt.show()
```



[16]:
```
plt.close(
    "all"
)  # closes existing ax3d to avoid bugs in next notebooks when running notebook tests
```

11.4 Case studies

11.4.1 Case study - Zhang et al., 2023

Here, using XGI, we reproduce figures 1 and 2 of the paper
"Higher-order interactions shape collective dynamics differently in hypergraphs and simplicial complexes"
Yuanzhao Zhang , *Maxime Lucas*, Federico Battiston
Nature Communications, 14(1), 2023, p.1605

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sb
from mpl_toolkits.axes_grid1.inset_locator import inset_axes

import xgi

sb.set_context("paper")
sb.set_theme(style="ticks")
```

[2]:
```
# utility functions to compute the Lyapunov exponents from the Laplacians
```

(continues on next page)

(continued from previous page)

```
def compute_eigenvalues(H, order, weight, rescale_per_node=True):
    """Returns the Lyapunov exponents of corresponding to the Laplacian of order d.

    Parameters
    ----------
    HG : xgi.HyperGraph
        Hypergraph
    order : int
        Order to consider.
    weight: float
        Weight, i.e coupling strenght gamma in [1]_.
    rescale_per_node: bool, (default=True)
        Whether to rescale each Laplacian of order d by d (per node).

    Returns
    -------
    lyap : array
        Array of dim (N,) with unsorted Lyapunov exponents
    """

    # compute Laplacian
    L = xgi.laplacian(H, order, rescale_per_node=rescale_per_node)
    K = xgi.degree_matrix(H, order)

    # compute eigenvalues
    eivals, _ = np.linalg.eig(L)
    lyap = -(weight / np.mean(K)) * eivals
    return lyap


def compute_eigenvalues_multi(H, orders, weights, rescale_per_node=True):
    """Returns the Lyapunov exponents of corresponding to the muliotder Laplacian.

    Parameters
    ----------
    HG : xgi.HyperGraph
        Hypergraph
    orders : list of int
        Orders of interactions to consider.
    weights: list of float
        Weight of each order, i.e coupling strenghts gamma_i in [1]_.
    rescale_per_node: bool, (default=True)
        Whether to rescale each Laplacian of order d by d (per node).

    Returns
    -------
    lyap : array
        Array of dim (N,) with unsorted Lyapunov exponents
    """

    # compute multiorder Laplacian
    L_multi = xgi.multiorder_laplacian(
```

(continues on next page)

(continued from previous page)

```
        H, orders, weights, rescale_per_node=rescale_per_node
    )

    # compute eigenvalues
    eivals_multi, _ = np.linalg.eig(L_multi)
    lyap_multi = -eivals_multi
    return lyap_multi
```

Fig. 1

Generate random structures

[3]:
```
N = 50  # number of nodes
ps = [
    0.1,
    0.1,
]  # ps[i] is the wiring probability of any i+2 nodes (ps[0] is for edges, e.g.)
alpha = 0.5  # ratio between coupling strength of 1st and 2nd order interaction (must be␣
↪in [0,1])
# rescale = True # whether to rescale

n_repetitions = 3  # number of realisations of random structures

# generate random hypergraphs
HGs = [xgi.random_hypergraph(N, ps) for i in range(n_repetitions)]
```

[4]:
```
# generate random simplicial complex
MSCs = [xgi.random_flag_complex_d2(N, p=0.5) for i in range(n_repetitions)]
```

Compute Lyapunov exponents

[5]:
```
alphas = np.arange(0, 1.01, 0.1)
n_alpha = len(alphas)

lyaps_HG = np.zeros((n_alpha, N, n_repetitions))

# compute Lyapunov exponents for all alpha values
for j, HG in enumerate(HGs):  # for all hypergraphs
    for i, alpha in enumerate(alphas):
        lyap_1 = compute_eigenvalues(HG, order=1, weight=1 - alpha)
        lyap_2 = compute_eigenvalues(HG, order=2, weight=alpha)
        lyap_multi = compute_eigenvalues_multi(
            HG, orders=[1, 2], weights=[1 - alpha, alpha]
        )

        lyap_multi = np.sort(lyap_multi)[::-1]
        lyaps_HG[i, :, j] = lyap_multi
```

(continues on next page)

(continued from previous page)

```
lyaps_MSC = np.zeros((n_alpha, N, n_repetitions))

# compute Lyapunov exponents for all alpha values
for j, MSC in enumerate(MSCs):  # for all simplicial complexes
    for i, alpha in enumerate(alphas):
        lyap_1 = compute_eigenvalues(MSC, order=1, weight=1 - alpha)
        lyap_2 = compute_eigenvalues(MSC, order=2, weight=alpha)
        lyap_multi = compute_eigenvalues_multi(
            MSC, orders=[1, 2], weights=[1 - alpha, alpha]
        )

        lyap_multi = np.sort(lyap_multi)[::-1]
        lyaps_MSC[i, :, j] = lyap_multi
```

[6]:
```
# average and std over the random realisations
# consider the second largest exponents only
means_HG = np.mean(lyaps_HG[:, 1, :], axis=1)
std_HG = np.std(lyaps_HG[:, 1, :], axis=1)

means_MSC = np.mean(lyaps_MSC[:, 1, :], axis=1)
std_MSC = np.std(lyaps_MSC[:, 1, :], axis=1)
```

Plot results

[7]:
```
fig, ax = plt.subplots(figsize=(4, 2.7))

ax.errorbar(
    alphas,
    means_HG,
    yerr=std_HG,
    fmt="-o",
    color="C0",
    ecolor="gray",
    elinewidth=3,
    capsize=0,
    label="hypergraph",
)

ax.errorbar(
    alphas,
    means_MSC,
    yerr=std_MSC,
    fmt="-o",
    color="C2",
    ecolor="gray",
    elinewidth=3,
    capsize=0,
    label="simplicial complex",
)
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel(r"$\lambda_2$")
ax.set_xlabel(r"$\alpha$")

ax.set_xticks([0, 0.5, 1])

sb.despine()
ax.legend(frameon=False)

fig_name = f"lambda2_HG_SC_N_{N}_ps_{ps}_nrep_{n_repetitions}"
# plt.savefig(f"{fig_name}.pdf", dpi=250, bbox_inches="tight")

plt.show()
```

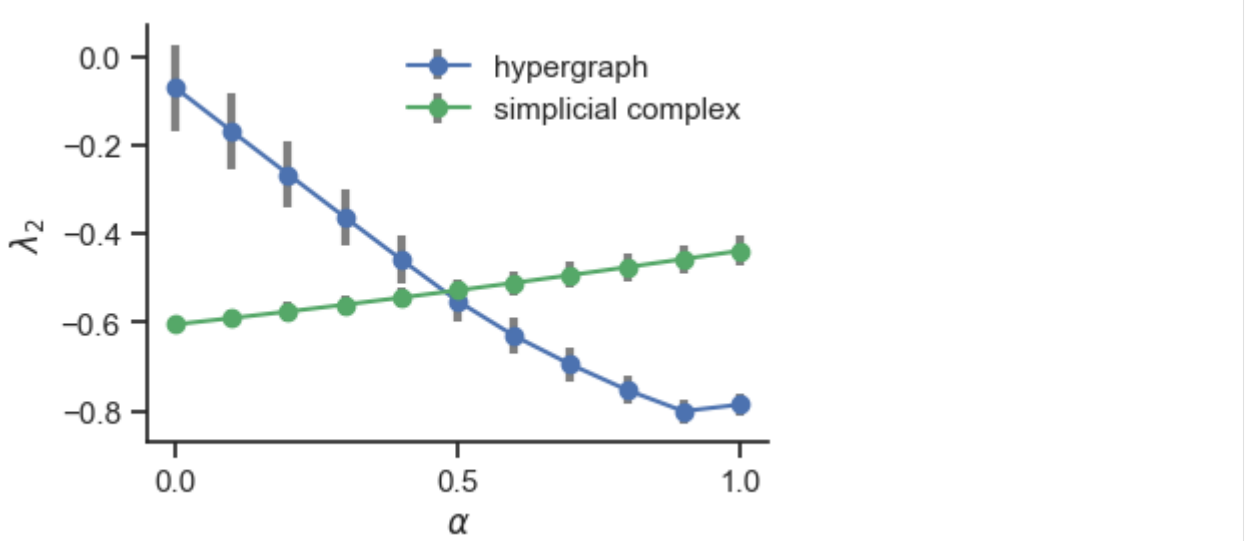


Fig. 2

Generate random structures and compute exponents

[8]:
```
N = 50  # number of nodes

p_1s = [0.2, 0.4, 0.6, 0.8]  # wiring probability of 1-hyperedges
p_2 = 0.05  # wiring probability of 2-hyperedges
alphas = np.arange(0, 1.01, 0.05)

HGs_2 = []
lyaps_HG_2 = np.zeros((len(p_1s), len(alphas), N))

for j, p_1 in enumerate(p_1s):
    ps = [
        p_1,
        p_2,
    ]  # ps[i] is the wiring probability of any i+2 nodes (ps[0] is for edges, e.g.)
```

(continues on next page)

(continued from previous page)

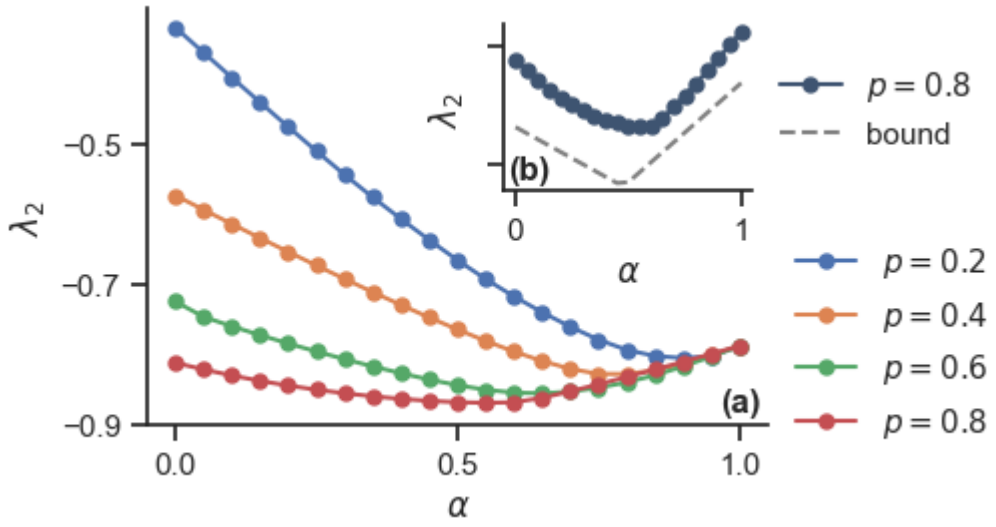
```
    # generate hyperedges
    HG = xgi.random_hypergraph(N, ps, seed=0)
    HGs_2.append(HG)

    # compute exponents
    for i, alpha in enumerate(alphas):
        lyap_1 = compute_eigenvalues(HG, order=1, weight=1 - alpha)
        lyap_2 = compute_eigenvalues(HG, order=2, weight=alpha)
        lyap_multi = compute_eigenvalues_multi(
            HG, orders=[1, 2], weights=[1 - alpha, alpha]
        )

        lyap_multi = np.sort(lyap_multi)[::-1]
        lyaps_HG_2[j, i, :] = lyap_multi
```

[9]:
```
def bound_multi(H, alpha, rescale_per_node=True):
    """Returns the lower bound N/(N-1) k_min

    Parameters
    ----------
    HG : xgi.HyperGraph
        Hypergraph
    orders : list of int
        Orders of interactions to consider.
    weights: list of float
        Weight of each order, i.e coupling strenghts gamma_i in [1]_.
    rescale_per_node: bool, (default=False)
        Whether to rescale each Laplacian of order d by d (per node).

    Returns
    -------
    float, bound of the Lyapunov exponent
    """
    L_multi = xgi.multiorder_laplacian(
        H, orders=[1, 2], weights=[1 - alpha, alpha], rescale_per_node=rescale_per_node
    )
    K_multi = np.diag(L_multi)
    N = H.num_nodes
    return -(N / (N - 1)) * np.min(K_multi)
```

[10]:
```
# compute theoretical bound
bound = np.array([bound_multi(HGs_2[-1], alpha) for alpha in alphas])
```

Plot results

[11]:
```
fig, ax = plt.subplots(figsize=(4, 2.7))

# set gradient palette (might need to re-run to make it effective)
palette = sb.dark_palette("#69d", reverse=True)
sb.set_palette(palette)

# plot curves
for i, p_1 in enumerate(p_1s):
    ax.plot(alphas, lyaps_HG_2[i, :, 1], "o-", label=f"$p={p_1:.1f}$", ms=5)

ax.set_ylabel(r"$\lambda_2$")
ax.set_xlabel(r"$\alpha$")

ax.set_yticks([-0.5, -0.7, -0.9])
ax.set_xticks([0, 0.5, 1])

ax.legend(frameon=False, loc="center left", bbox_to_anchor=(1, 0.2))

# add inset with bound
# Create inset of width 30% and height 40% of the parent axes' bounding box
# at the lower left corner (loc=3)
axins = inset_axes(ax, width="40%", height="40%")
k = 3
axins.plot(
    alphas, lyaps_HG_2[k, :, 1], "o-", c=f"C{k}", label=f"$p={p_1s[k]:.1f}$", ms=5
)
axins.plot(alphas, bound, "--", label="bound", c="grey")

axins.set_ylabel(r"$\lambda_2$")
axins.set_xlabel(r"$\alpha$")
axins.set_yticklabels([])

axins.legend(frameon=False, loc="center left", bbox_to_anchor=(1, 0.5))

sb.despine()

ax.text(
    0.99, 0.02, "(a)", transform=ax.transAxes, va="bottom", ha="right", weight="bold"
)
axins.text(
    0.015, 0.03, "(b)", transform=axins.transAxes, va="bottom", ha="left", weight="bold"
)

fig_name = f"phase_diagram_lines_p2_{p_2}"
# plt.savefig(f"{fig_name}.pdf", dpi=250, bbox_inches="tight")

plt.show()
```



[ ]:

11.4.2 Case study - Simulating the simplicial Kuramoto model

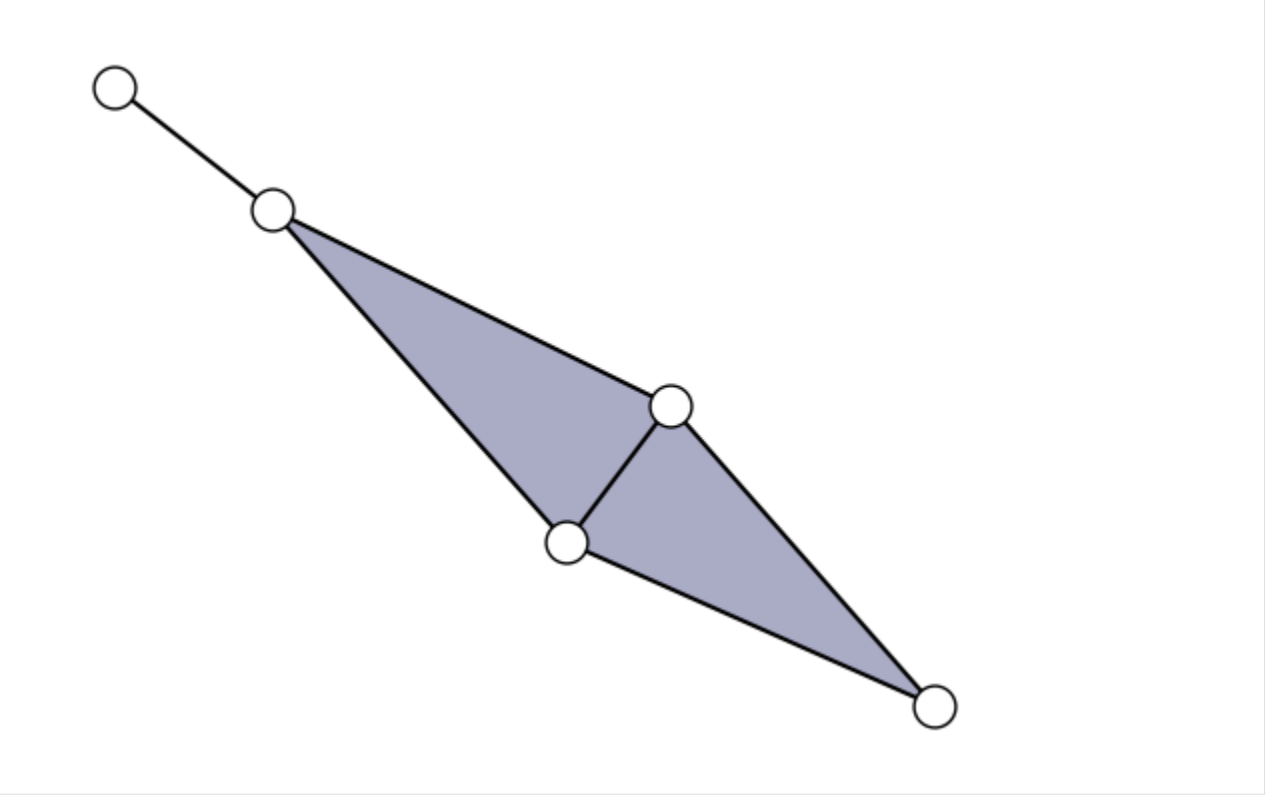
[1]:
```
import matplotlib.pyplot as plt
import numpy as np

import xgi
```

The simplicial Kuramoto model is a direct generalization of the Kuramoto model on graphs, which allows us to describe the interaction between oscillating simplices in a simplicial complex.

In order to simulate the simplicial Kuramoto model one needs an oriented simplicial complex. This can be achieved in XGI through the `SimplicialComplex` class together with a dictionary specifying the orientation of every simplex.

Let us start by defining a simple simplicial complex.

[2]:
```
S = xgi.SimplicialComplex([[1, 2, 3], [2, 3, 4], [4, 5]])
xgi.draw(S);
```



[3]:
```
S.edges.members()
```

[3]:
```
[frozenset({1, 2, 3}),
 frozenset({2, 3, 4}),
 frozenset({4, 5}),
 frozenset({2, 4}),
 frozenset({1, 2}),
 frozenset({3, 4}),
 frozenset({2, 3}),
 frozenset({1, 3})]
```

We see above the list of simplices contained in the complex. We define the reference orientation of each one of them as the one given by the sorted list nodes in the simplex. The orientation will then be a boolean value: 0 for the reference orientation and 1 for the opposite one. For example, giving orientation 0 to {1,2,3} means that we are considering the oriented simplex [1,2,3] while 1 is associated to [2,1,3] (and all of the equivalent orientations).

We thus build a dictionary which associates the boolean orientation value to the ID of each (non singleton) simplex.

[4]:
```
orientations = {idd: 0 for idd in list(S.edges.filterby("order", 1, mode="geq"))}
```

If we want to flip the orientation of, say, the face [1,2,3] we just do the following:

[5]:
```
orientations[0] = 1
```

Having now an oriented simplicial complex we can run a simulation of the simplicial Kuramoto dynamics. First, however we need to define a few parameters: - order : the order of the oscillating simplices (0 for nodes, 1 for edges, 2 for faces, ...); - omega : the natural frequencies of each oscillator; - theta0 : the initial phases of the oscillators; - sigma : the strength of the interactions; - T : the time horizon of the simulation; - n_steps : the number of integration steps.

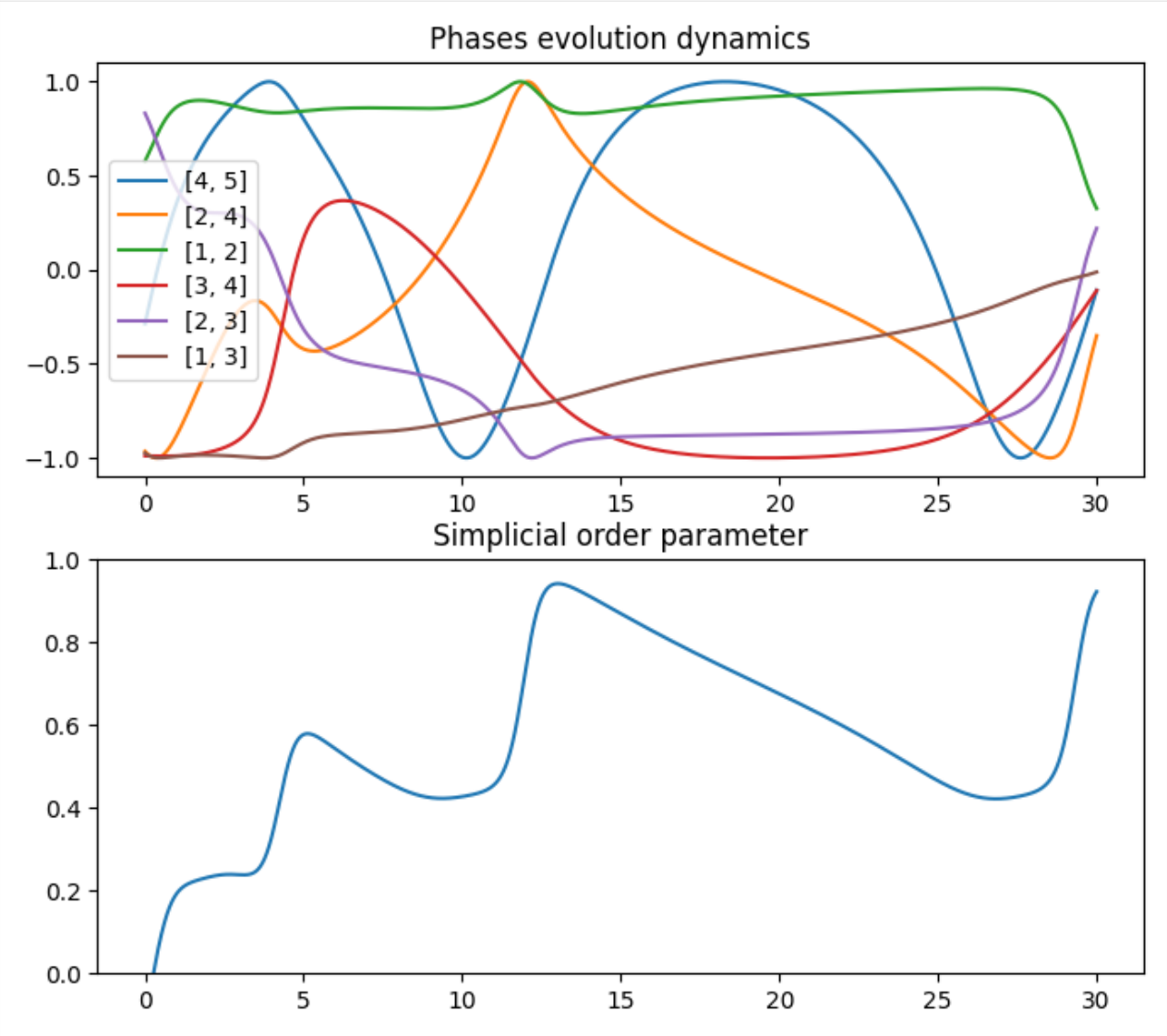
[6]:
```
order = 1
n = len(S.edges.filterby("order", order))  # Number of oscillating simplices

omega = np.random.rand(n, 1)
theta0 = 2 * np.pi * np.random.rand(n, 1)

sigma = 0.4
T = 30
n_steps = 5000
```

We then simulate the dynamics and compute the simplicial order parameter with the two functions - `simulate_simplicial_kuramoto` - `compute_simplicial_order_parameter`

[7]:
```
(
    theta,
    theta_minus,
    theta_plus,
    om1_dict,
    o_dict,
    op1_dict,
) = xgi.synchronization.simulate_simplicial_kuramoto(
    S, orientations, order, omega, sigma, theta0, T, n_steps, True
)

r = xgi.synchronization.compute_simplicial_order_parameter(theta_minus, theta_plus)
```

The output phases timeseries is contained in *theta* while *theta_minus* and *theta_plus* are, respectively, the projection of the dynamics onto lower order and higher order simplices.

[8]:
```
fig, axs = plt.subplots(2, 1)
fig.set_figheight(7)
fig.set_figwidth(8)

labels_list = [
    "[%s]" % ", ".join(map(str, list(S.edges.members()[idx])))
    for idx in list(o_dict.values())
]


axs[0].plot(np.linspace(0, T, n_steps), np.sin(np.transpose(theta)))
axs[0].set_title("Phases evolution dynamics")
axs[0].legend(labels_list)

axs[1].plot(np.linspace(0, T, n_steps), r)
axs[1].set_title("Simplicial order parameter")
axs[1].set_ylim((0, 1))

plt.show()
```


Phases evolution dynamics
[4, 5]
[2, 4]
[1, 2]
[3, 4]
[2, 3]
[1, 3]
1.0
0.5
0.0
−0.5
−1.0
0
5
10
15
20
25
30
Simplicial order parameter
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25
30

CHAPTER

TWELVE

RECIPES

12.1 1. Simplicial complex from pairwise data

This recipe show you how to create a simplicial complex by flagging the cliques (k-cliques are promoted to simplices of order $k-1$).

[1]:
```
import networkx as nx

import xgi
```

[2]:
```
G = nx.barabasi_albert_graph(n=100, m=2, seed=1)
H = xgi.flag_complex_d2(G)

print(H)
```

```
Unnamed SimplicialComplex with 100 nodes and 219 simplices
```

12.2 2. Laplacian spectrum

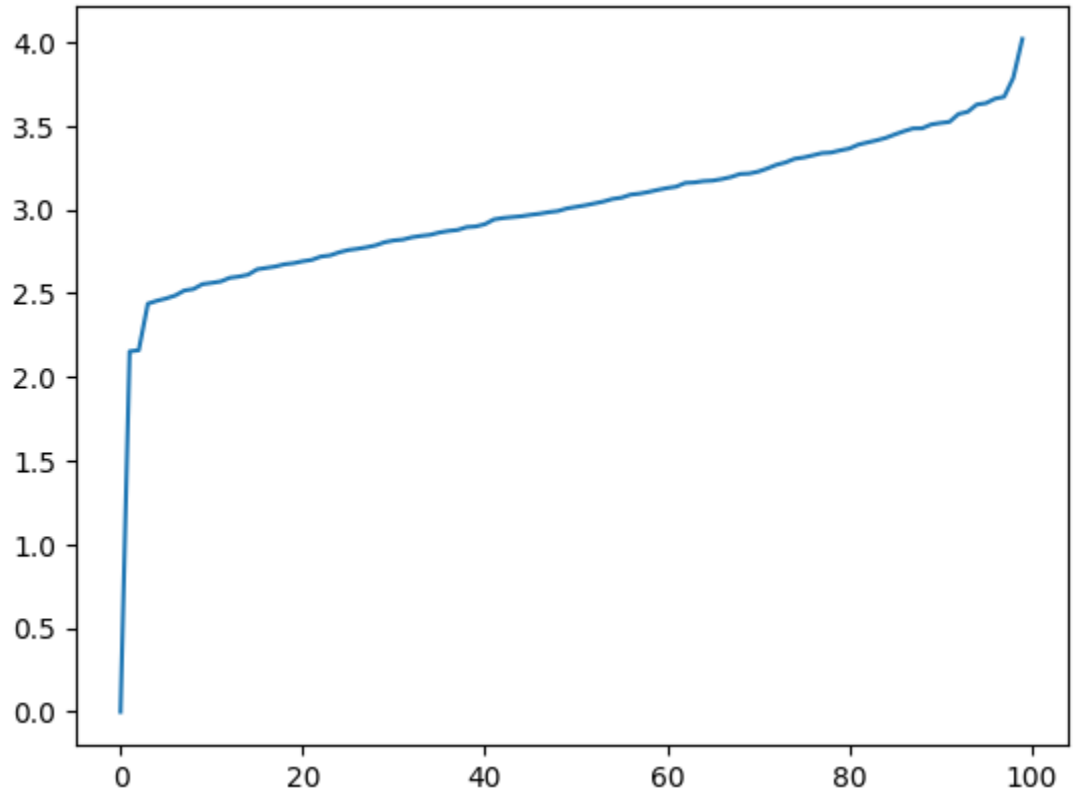
This recipe allows you write the multiorder Laplacian of a hypergraph. It also shows how to compute the eigenvalues and eigenvectors and how to plot the Laplacian spectrum.

[3]:
```
import matplotlib.pyplot as plt
import numpy as np
from numpy.linalg import eigh

import xgi
```

[4]:
```
H = xgi.random_hypergraph(N=100, ps=[0.2, 0.02], seed=1)
```

[5]:
```
orders = np.array(xgi.unique_edge_sizes(H)) - 1
L_multi = xgi.multiorder_laplacian(H, orders=orders, weights=[1] * len(orders))
eivals, eivects = eigh(L_multi)
```

[6]:
```
plt.plot(eivals);
```



12.3 3. Adjacency tensor

This recipe shows you how to retrieve the adjacency tensor at a given order of a hypergraph.

[7]:
```
from itertools import permutations

import xgi


def adjacency_tensor(H, order):
    N = H.num_nodes
    shape = tuple([N] * (order + 1))
    tensor = np.zeros(shape)

    edges = H.edges.filterby("order", order)
    for id, members in edges.members(dtype=dict).items():
        for idcs in permutations(members):
            tensor[idcs] = 1

    return tensor
```

[8]:
```
print(adjacency_tensor(H, 1).shape)
print(adjacency_tensor(H, 2).shape)
```

```
(100, 100)
(100, 100, 100)
```

12.4 4. Create random hypergraph

This recipe allows you to create a random hypergraph. It then shows you how to print a short summary of the hypergraph.

[9]:
```
import xgi

N = 50  # number of nodes
ps = [0.5, 0.2, 0.1]  # probabilities of edges of each order

H = xgi.random_hypergraph(N, ps)

print(H)
```

```
Unnamed Hypergraph with 50 nodes and 27705 hyperedges
```

12.5 5. Clean-up

This recipe shows you how to remove singletons, isolated nodes, multiedges from your hypergraph dataset, this is achieved with the `cleanup` function that also relabels your nodes and edges with integer labels.

[10]:
```
import xgi

H_enron = xgi.load_xgi_data("email-enron")
print(H_enron)
```

```
Hypergraph named email-Enron with 148 nodes and 10885 hyperedges
```

[11]:
```
H_enron.cleanup()
print(H_enron)
```

```
Hypergraph named email-Enron with 143 nodes and 1459 hyperedges
```

12.6 6. Add two hypergraphs

This recipe allows you to add two hypergraphs. This is done by merging the two hypergraphs and then removing the duplicate instances of edges.

[12]:
```
import xgi

H1 = xgi.Hypergraph([[1, 2, 3], [3, 4]])
H2 = xgi.Hypergraph([[1, 2], [3, 4], [4, 5, 6]])
# create an hypergraph by merging H1 and H2
H_res = H1 << H2
# remove duplicated edges
H_res.merge_duplicate_edges()
```

(continues on next page)

(continued from previous page)

```
# print the nodes and edges in order to see that everything is correct
print(H_res.nodes)
print(H_res.edges.members())
```

```
[1, 2, 3, 4, 5, 6]
[{1, 2, 3}, {1, 2}, {4, 5, 6}, {3, 4}]
```

12.7 7. Filterby

This recipe shows you how to filter nodes and edges of a hypergraph based on the values of some statistics.

[13]:
```
H = xgi.Hypergraph([{1, 2, 3}, {1, 2}, {4, 5, 6}, {3, 4}])
# filter the nodes of degree 2 and print them
print(H.nodes.filterby("degree", 2))
# filter the edges of size 2 and print them
print(H.edges.filterby("size", 2).members())
```

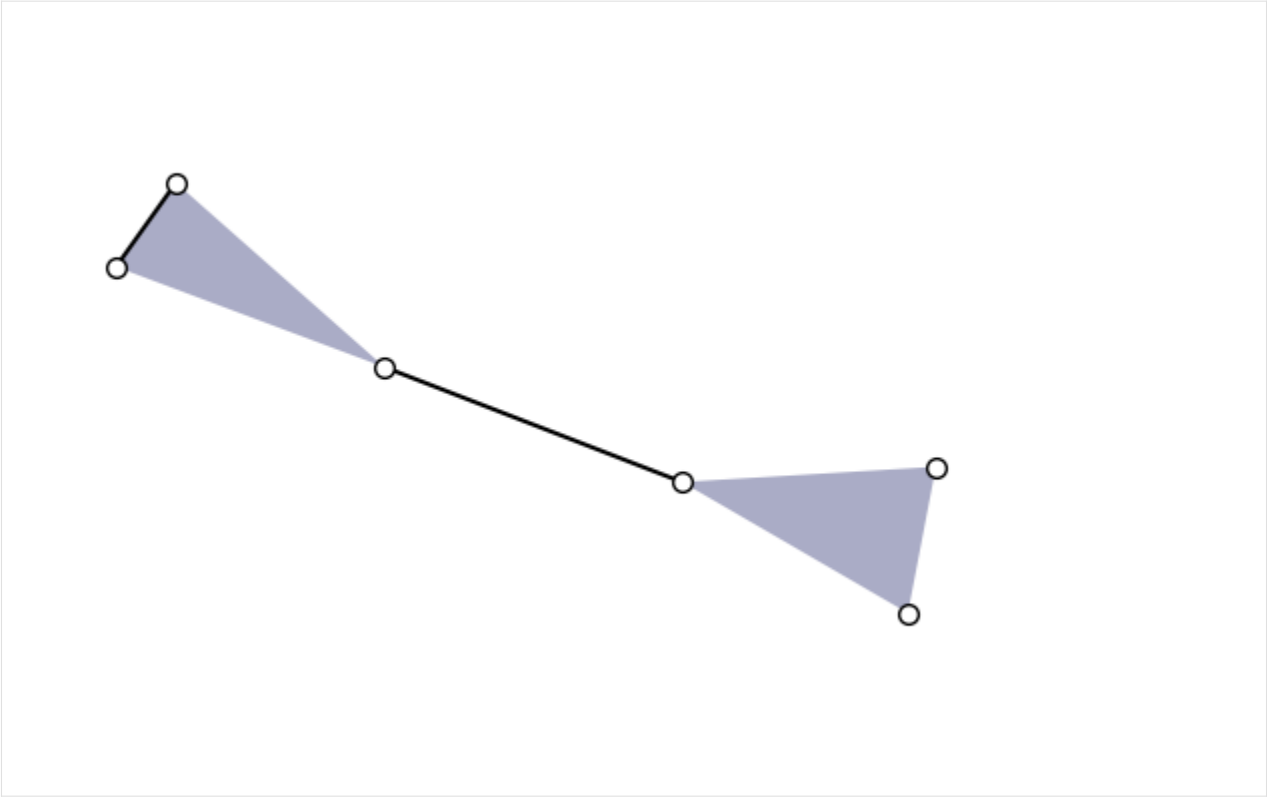
```
[1, 2, 3, 4]
[{1, 2}, {3, 4}]
```

12.8 8. Plot a hypergraph showing one order only

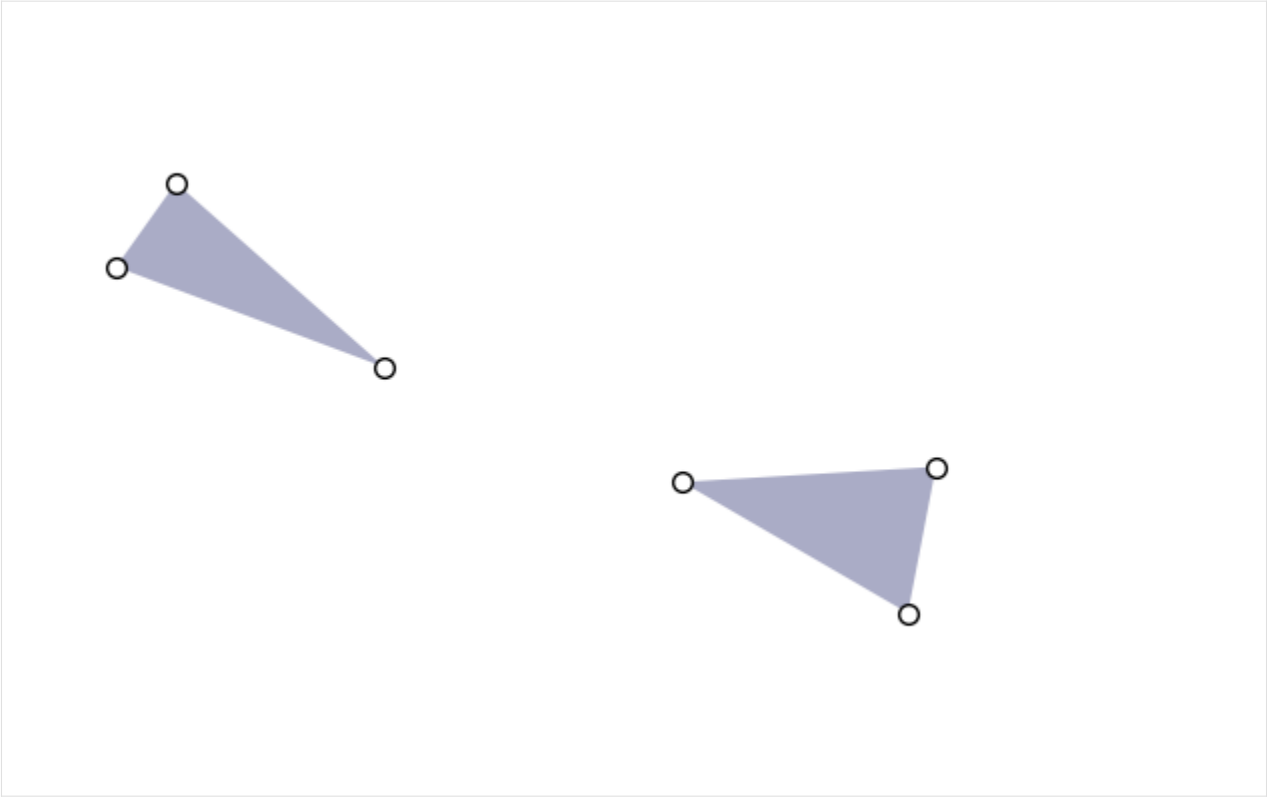
This recipe shows you how to plot a hypergraph showing only the edges of a certain order.

[14]:
```
import xgi

H = xgi.Hypergraph([{1, 2, 3}, {1, 2}, {4, 5, 6}, {3, 4}])
pos = xgi.barycenter_spring_layout(H, seed=1)
```

[15]:
```
# plot it with all orders
xgi.draw(H, pos);
```



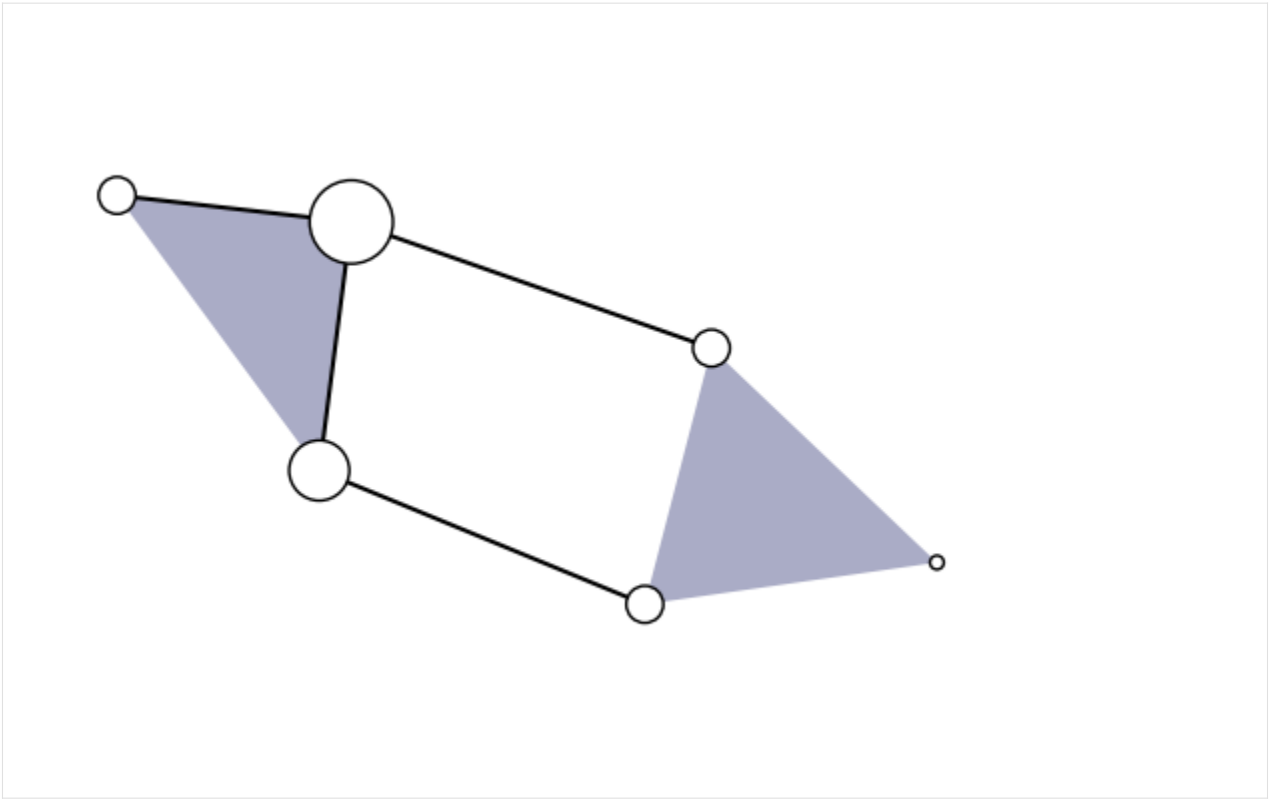
[16]:
```
# plot only edges of order 2
H_order_2 = xgi.subhypergraph(H, edges=H.edges.filterby("order", 2))
xgi.draw(H_order_2, pos=pos);
```



12.9 9. Plot with stats

This recipe shows you how use the statistics of a hypergraph when plotting it. In this case we modify the size of the nodes according to their degree.

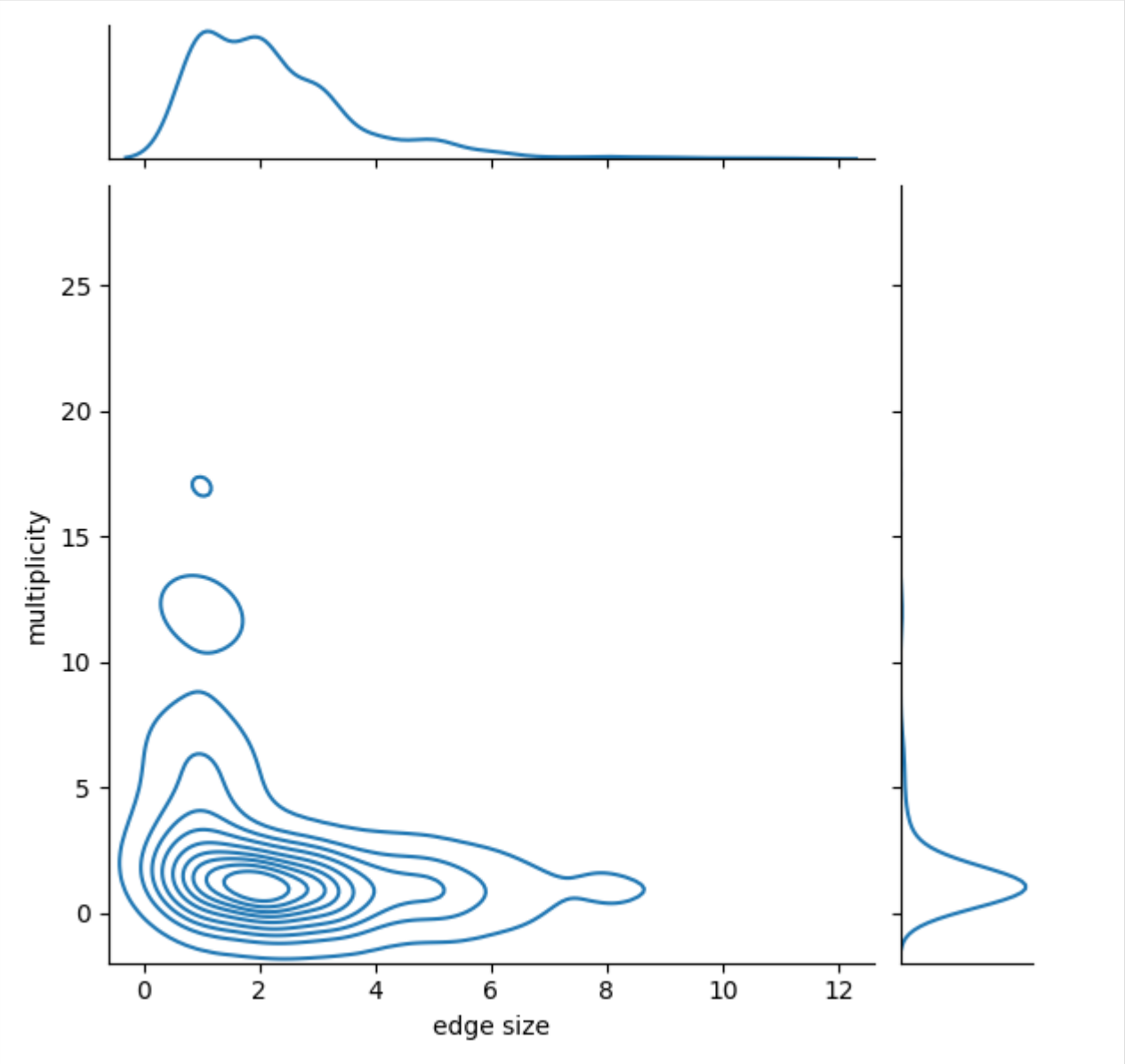
[17]:
```
H = xgi.Hypergraph([{1, 2, 3}, {1, 2}, {4, 5, 6}, {3, 4}, {1, 5}, {1, 3}])
pos = xgi.barycenter_spring_layout(H, seed=1)
# plot with node size corresponding to the degree
xgi.draw(H, pos, node_size=H.nodes.degree);
```



12.10 10. Merging multiedges

This recipe allows you to merge multiedges in a hypergraph. It also shows how to visualize the multiplicity distibution for the edges.

[18]:
```
import pandas as pd
import seaborn as sns

import xgi

H = xgi.load_xgi_data("diseasome")
H.merge_duplicate_edges(rename="tuple", multiplicity="weight")

edge_size = H.edges.size.asnumpy()
multiplicity = H.edges.attrs("weight").asnumpy()
df = pd.DataFrame.from_dict(
    {
        "edge size": H.edges.size.aslist(),
        "multiplicity": H.edges.attrs("weight", missing=1).aslist(),
    }
)

g = sns.jointplot(data=df, x="edge size", y="multiplicity", kind="kde")
sns.despine()
```



12.11 11. Degree distribution

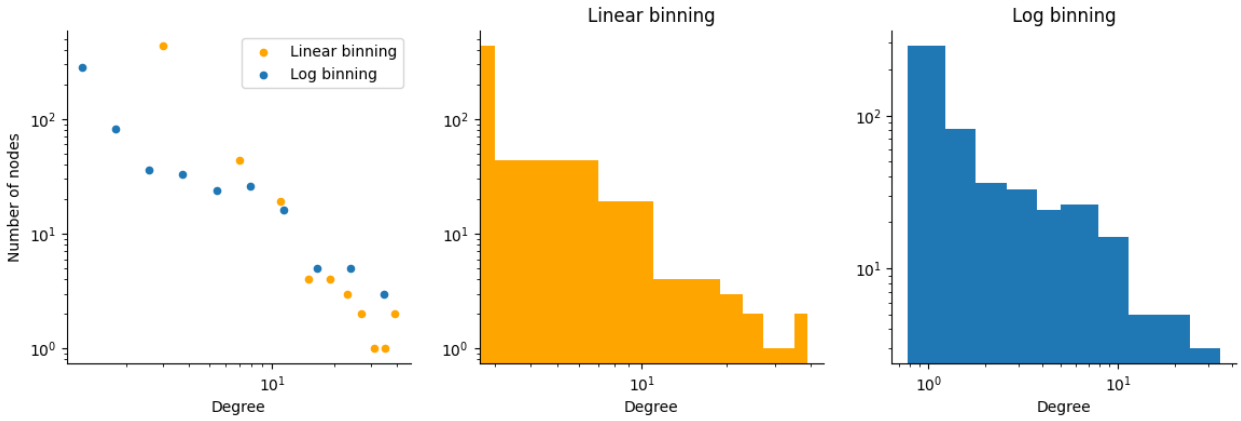
This recipe shows how to plot the degree distribution of a hypergraph in different layouts.

[19]:
```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

import xgi

H = xgi.load_xgi_data("diseasome")
df1 = H.nodes.degree.ashist(bin_edges=True)
```

(continues on next page)

(continued from previous page)

```
df2 = H.nodes.degree.ashist(bin_edges=True, log_binning=True)

plt.figure(figsize=(14, 4))
plt.subplot(131)
df1.plot(
    "bin_center",
    "value",
    ax=plt.gca(),
    logx=True,
    logy=True,
    kind="scatter",
    color="orange",
    label="Linear binning",
)
df2.plot(
    "bin_center",
    "value",
    ax=plt.gca(),
    logx=True,
    logy=True,
    kind="scatter",
    label="Log binning",
)
plt.ylabel("Number of nodes")
plt.xlabel("Degree")
plt.legend()
sns.despine()

plt.subplot(132)
plt.title("Linear binning")
plt.bar(df1.bin_lo, df1.value, width=df1.bin_hi - df1.bin_lo, log=True, color="orange")
plt.xscale("log")
plt.xlabel("Degree")
sns.despine()

plt.subplot(133)
plt.title("Log binning")
plt.bar(df2.bin_lo, df2.value, width=df2.bin_hi - df2.bin_lo, log=True)
plt.xscale("log")
plt.xlabel("Degree")
sns.despine()

plt.show()
```

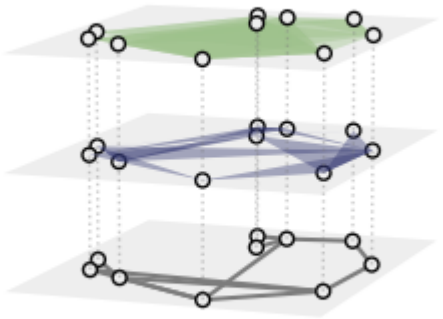


12.12 12. Multilayer visualization of a hypergraph

This recipe shows how to plot the multilayer visualization of a hypergraph. This plotting function diplays higher-order structes in 3D showing hyperedges/simplices of different orders on superimposed layers.

[20]:
```
import matplotlib.pyplot as plt

import xgi

H = xgi.random_hypergraph(N=10, ps=[0.2, 0.05, 0.05], seed=1)

_, ax = plt.subplots(figsize=(4, 4), subplot_kw={"projection": "3d"})
xgi.draw_multilayer(H, ax=ax)
```

[20]:
```
(<Axes3DSubplot: >,
 (<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x144a8a690>,
  <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x144fe5bd0>))
```



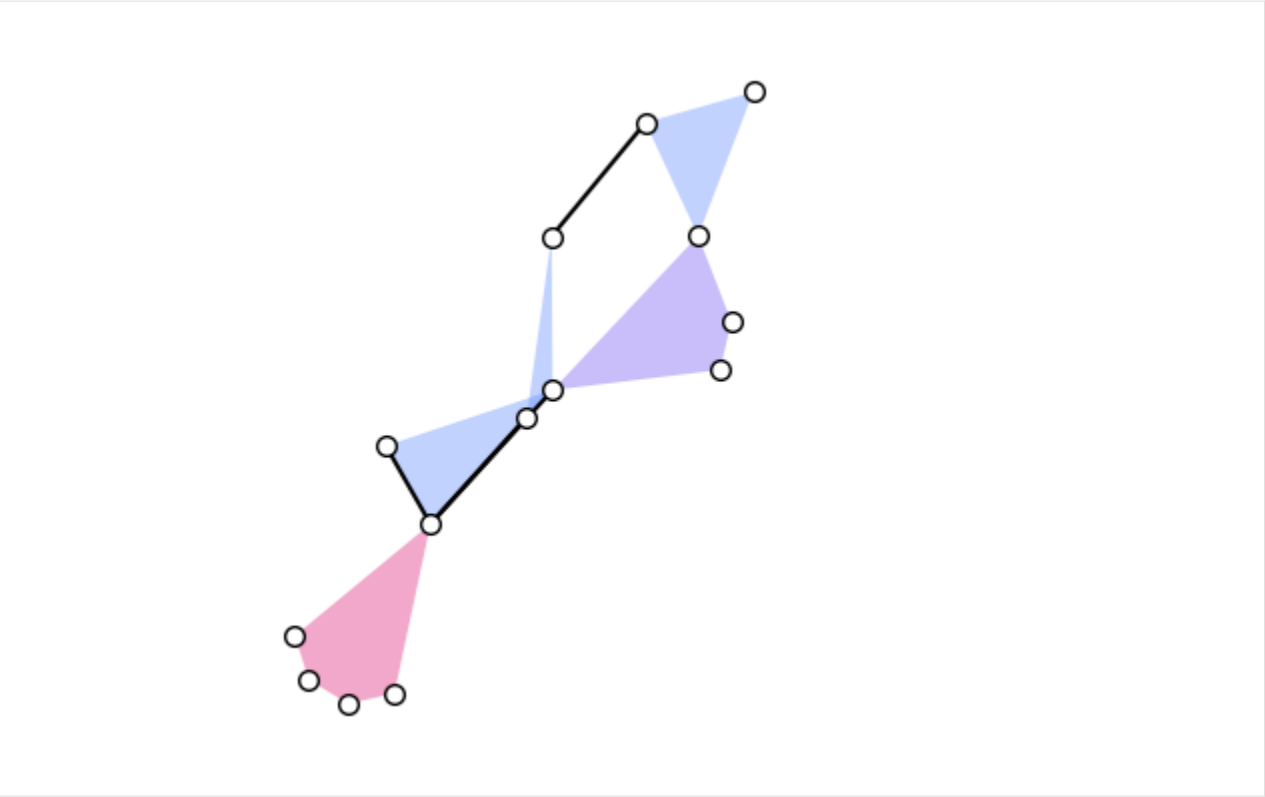
12.13 13. Specifying the colours of hyperedges

This recipe will show how to construct a hypergraph and specify the colours of the different hyperedges. There are three options to do it. First of all we create the hypergraph and specify the colors we want to use.

[21]:
```
import xgi

links = [[1, 2], [1, 3], [5, 6], [1, 7]]
triangles = [[3, 5, 7], [2, 7, 1], [6, 10, 15]]
squares = [[7, 8, 9, 10]]
pentagons = [[1, 11, 12, 13, 14]]
edges = links + triangles + squares + pentagons

H = xgi.Hypergraph(edges)
pos = xgi.barycenter_spring_layout(H, seed=2)

link_color = "#000000"
triangle_color = "#648FFF"
square_color = "#785EF0"
pentagon_color = "#DC267F"
colors = [link_color, triangle_color, square_color, pentagon_color]
```

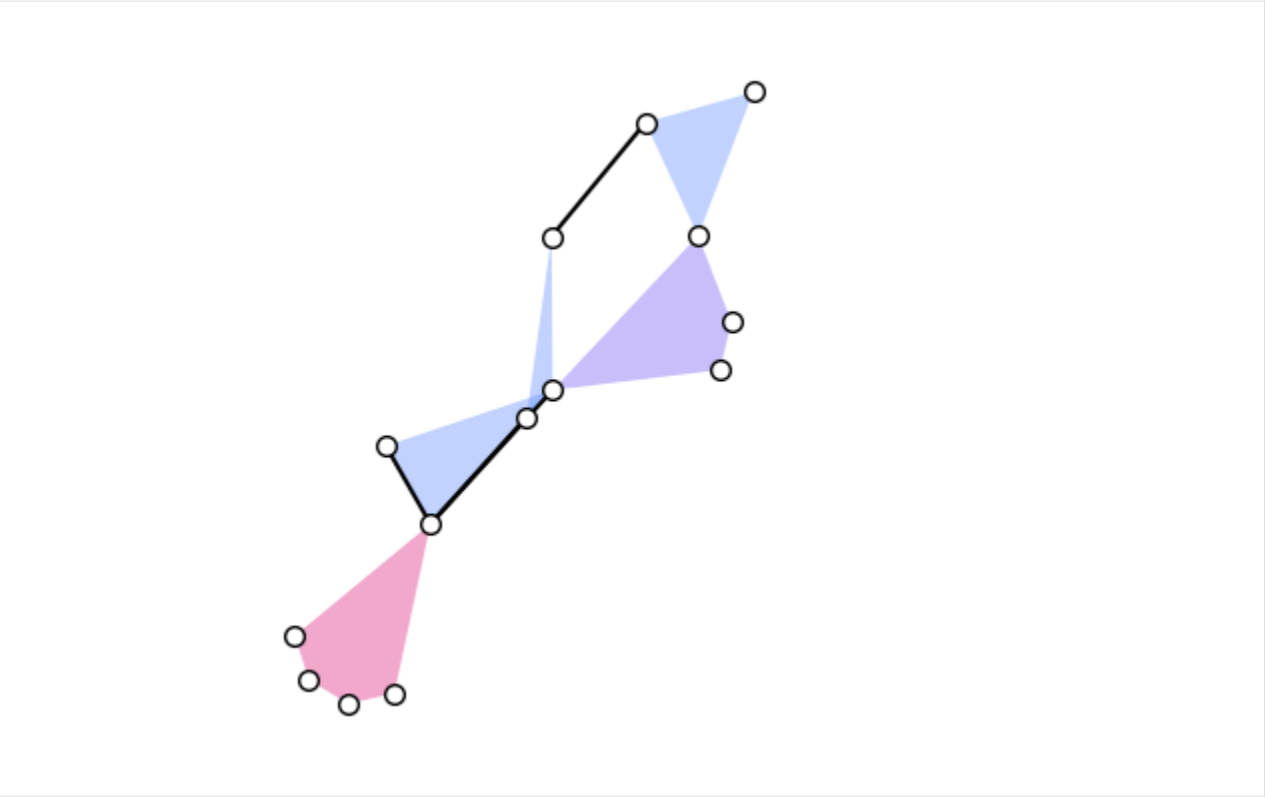
Option 1: input colors that are lists/arrays with the right number of elements in the right order

[22]:
```
edge_color = [colors[i - 2] for i in H.edges.filterby("order", 1, "gt").size.aslist()]

xgi.draw(H, pos=pos, dyad_color=link_color, edge_fc=edge_color);
```



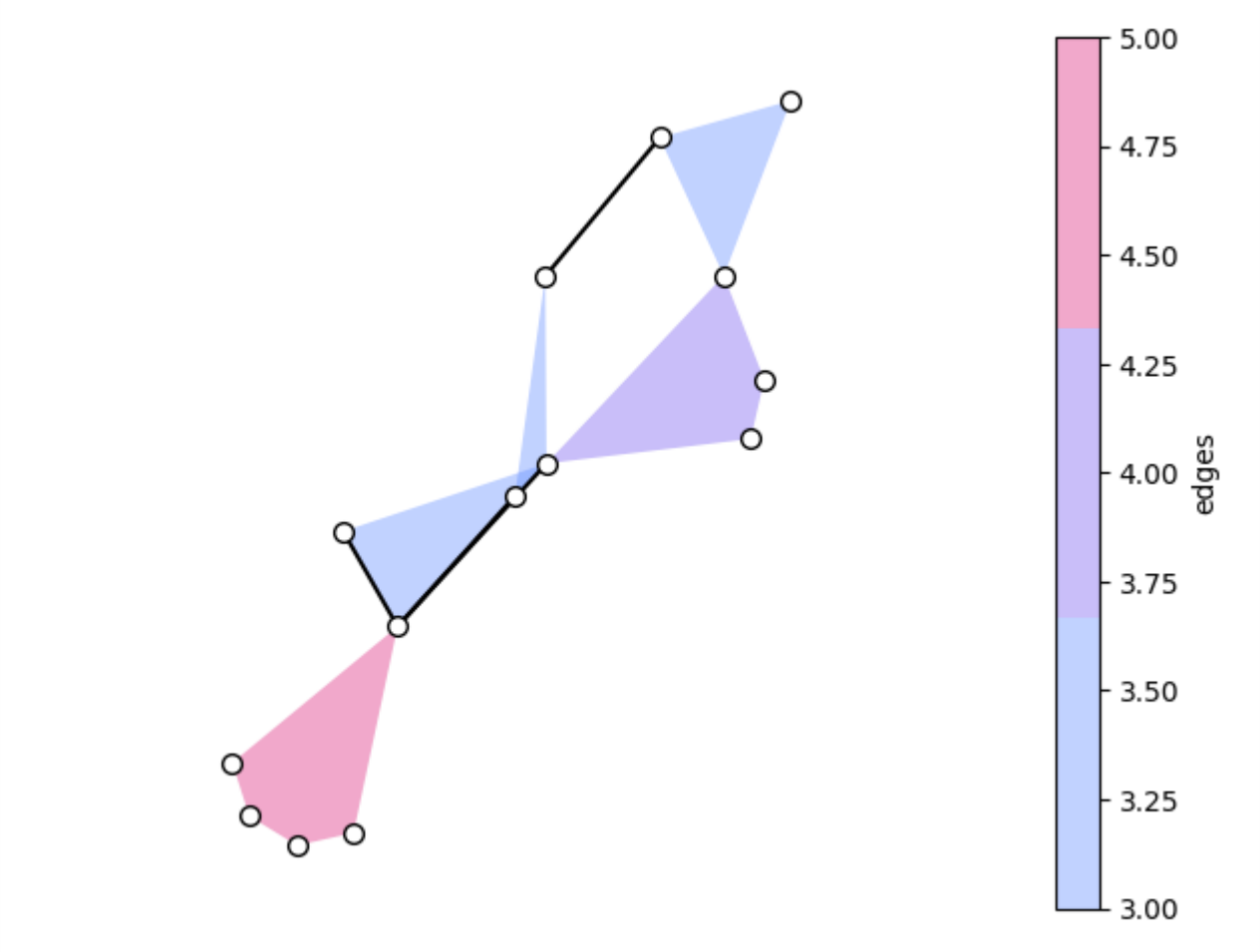
Option 2: a dictionary where keys are hyperedge ids and values are colours:

[23]:
```
color_dict = {idx: colors[i - 2] for idx, i in H.edges.size.asdict().items()}

xgi.draw(H, pos=pos, dyad_color=color_dict, edge_fc=color_dict);
```



Option 3: just create a cmap that has the colours you want. This works in this case because the edges are already plotted with colors corresponding to their size. This option also allows to show an associated colorbar.

[24]:
```
from matplotlib.colors import ListedColormap

cmap = ListedColormap(colors[1:])

_, (node_collection, dyad_collection, edge_collection) = xgi.draw(
    H, pos=pos, dyad_color=link_color, edge_fc_cmap=cmap
)

plt.colorbar(edge_collection, label="edges")

plt.tight_layout()
```



12.14 14. Flag a triangular lattice

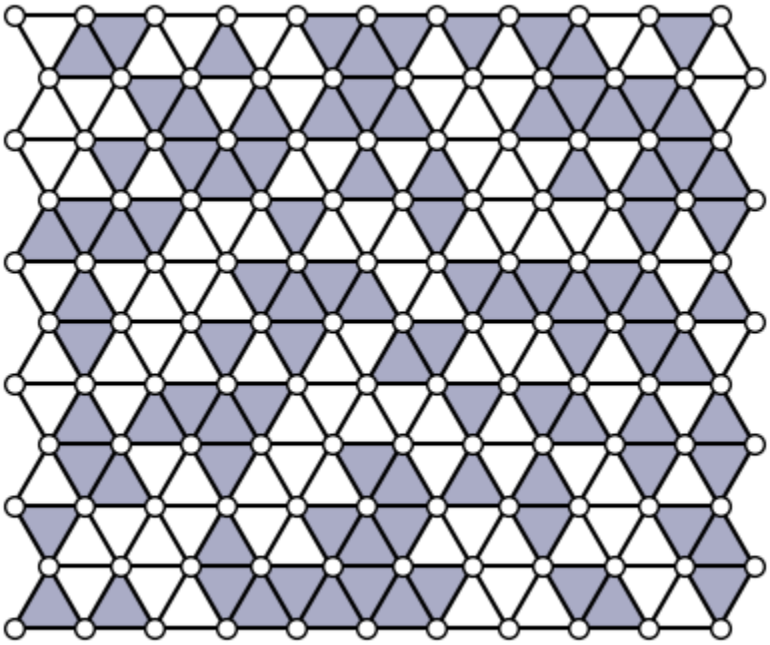
This recipe shows how you can create a simplicial complex object by randomly flagging a triangular lattice generated using NetworkX. This recipe uses the `xgi.flag_complex_d2()` (see documentation) function, but needs a small trick to levarage the way nodes are encoded in the NetworkX function to plot correctly the lattice.

[25]:
```
import networkx as nx

import xgi

m, n = 10, 20
p = 0.5

G = nx.triangular_lattice_graph(m, n, with_positions=True)
pos = nx.get_node_attributes(G, "pos")

mapping = {i: list(G.nodes)[i] for i in range(0, len(list(G.nodes)))}
inv_mapping = {v: k for k, v in mapping.items()}

G_aux = nx.relabel_nodes(G, inv_mapping)
```

(continues on next page)

(continued from previous page)

```
S = xgi.flag_complex_d2(G_aux, p2=p)

pos = {inv_mapping[k]: v for k, v in pos.items()}
xgi.draw(S, pos=pos);
```



12.15 14. Compute the average path length in a hypergraph

This recipe shows how to compute the average path length in a hypergraph. You can compute all shortest path lengths with `xgi.shortest_path_length()` (see documentation). The `xgi.shortest_path_length()` function returns an infinite length for disconnected nodes. To allow the computation of the average path length in any case we replace `np.inf` with 0 for disconnected nodes and remove length-0 paths for self-loops.

[26]:
```
import numpy as np

import xgi

H = xgi.random_hypergraph(N=100, ps=[0.2, 0.02], seed=1)
N = H.num_nodes
spl = xgi.shortest_path_length(H)

lens = []
for tup in spl:
    lens += tup[1].values()

# remove lengths 0 for self-loops
```

(continues on next page)

(continued from previous page)

```
lens = [i for i in lens if i != 0]

# replace inf by 0 for disconnected nodes
lens = [0 if i == np.inf else i for i in lens]

avg_shortest_path = np.sum(lens) / (N * (N - 1))
print("The average shortest path length is", avg_shortest_path)
```

```
The average shortest path length is 1.1086868686868687
```

12.16 16. Get all of the node IDs that have maximum degree

This recipe demonstrates how to get all of the indices corresponding to the maximum degree, because `argmax` only returns the first index corresponding to the maximal value.

[27]:
```
H = xgi.Hypergraph([[1, 2, 3, 4], [1, 2, 3], [1, 2]])

[k for k, v in H.degree().items() if v == H.nodes.degree.max()]
```

[27]:
```
[1, 2]
```

12.17 17. Get all the node IDs corresponding to the 100th largest degree

The `argsort` method allows us to access the node IDs by their statistical rank. Here we get the 100th largest degree and find all the node IDs that share that degree.

[28]:
```
H = xgi.load_xgi_data("email-enron")
ids = H.nodes.degree.argsort()
i = ids[-100]
d = H.degree()[i]
matching_ids = [k for k, v in H.degree().items() if v == d]
print(f"Nodes {', '.join(matching_ids)} have degree {d}")
```

```
Nodes 98, 64 have degree 49
```

12.18 18. Define a custom filtering function

In addition to the pre-defined filtering functionality, one can also define a custom comparison operator to compare statistics and attributes.

First, we show an example for numerical statistics and second, an example for attributes.

Numerical statistics

[29]:
```
import xgi

outsiderange = lambda val, arg: arg[0] > val or val > arg[1]
```

(continues on next page)

(continued from previous page)

```
H = xgi.load_xgi_data("email-enron")
print(f"The total number of nodes is {H.num_nodes}")

# Get all of nodes that have degree less than 3 or greater than 20
nodes = H.nodes.filterby("degree", [3, 20], mode=outsiderange)
print(f"The number of nodes with degree less than 3 or greater than 20 is {len(nodes)}")
```

```
The total number of nodes is 148
The number of nodes with degree less than 3 or greater than 20 is 128
```

Attributes

[30]:
```
import datetime

import xgi

date1 = datetime.datetime(2000, 1, 1)
date2 = datetime.datetime(2001, 1, 1)
datecompare = (
    lambda date, arg: arg[0] <= datetime.datetime.fromisoformat(date) <= arg[1]
)

H = xgi.load_xgi_data("email-enron")
print(f"The total number of hyperedges is {H.num_edges}")

# Get all of the dates between 01JAN2000 and 01JAN2001
e = H.edges.filterby_attr("timestamp", [date1, date2], mode=datecompare)
print(f"The number of hyperedges between 01JAN2000 and 01JAN2001 is {len(e)}")
```

```
The total number of hyperedges is 10885
The number of hyperedges between 01JAN2000 and 01JAN2001 is 3992
```

CHAPTER

THIRTEEN

XGI.CORE.HYPERGRAPH.HYPERGRAPH

class xgi.core.hypergraph.**Hypergraph**(*incoming_data=None*, ***attr*)

Bases: `object`

A hypergraph is a collection of subsets of a set of *nodes* or *vertices*.

A hypergraph is a pair (V, E), where V is a set of elements called *nodes* or *vertices*, and E is a set whose elements are subsets of V, that is, each $e \in E$ satisfies $e \subset V$. The elements of E are called *hyperedges* or simply *edges*.

The Hypergraph class allows any hashable object as a node and can associate attributes to each node, edge, or the hypergraph itself, in the form of key/value pairs. In this representation, multiedges are allowed.

Parameters

- **incoming_data** (*input hypergraph data, optional*) – Data to initialize the hypergraph. If None (default), an empty hypergraph is created, i.e. one with no nodes or edges. The data can be in the following formats:
  - hyperedge list
  - hyperedge dictionary
  - 2-column Pandas dataframe (bipartite edges)
  - Incidence matrix: numpy ndarray or scipy.sparse array
  - Hypergraph object
  - SimplicialComplex object
- ****attr** (*dict, optional*) – Attributes to add to the hypergraph as key, value pairs. By default, None.

Notes

Unique IDs are assigned to each node and edge internally and are used to refer to them throughout.

The *attr* keyword arguments are added as hypergraph attributes. To add node or edge attributes see *add_node()* and *add_edge()*.

In addition to the methods listed in this page, other methods defined in the *stats* package are also accessible via the *Hypergraph* class. For more details, see the tutorial.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [4], [5, 6], [6, 7, 8]])
>>> H.nodes
NodeView((1, 2, 3, 4, 5, 6, 7, 8))
>>> H.edges
EdgeView((0, 1, 2, 3))
```

Attributes

edges	An `EdgeView` of this network.
nodes	A `NodeView` of this network.
num_edges	The number of edges in the hypergraph.
num_nodes	The number of nodes in the hypergraph.
is_frozen	Checks whether a dihypergraph is frozen

Methods that modify the structure

add_node	Add one node with optional attributes.
add_edge	Add one edge with optional attributes.
add_nodes_from	Add multiple nodes with optional attributes.
add_edges_from	Add multiple edges with optional attributes.
add_node_to_edge	Add one node to an existing edge.
add_weighted_edges_from	Add multiple weighted edges with optional attributes.
set_node_attributes	Sets node attributes from a given value or dictionary of values.
set_edge_attributes	Set the edge attributes from a value or a dictionary of values.
update	Add nodes or edges to the hypergraph.
remove_node	Remove a single node.
remove_edge	Remove one edge.
remove_nodes_from	Remove multiple nodes.
remove_edges_from	Remove multiple edges.
remove_node_from_edge	Remove a node from an existing edge.
clear	Remove all nodes and edges from the graph.
clear_edges	Remove all edges from the graph without altering any nodes.
merge_duplicate_edges	Merges edges which have the same members.
cleanup	Removes potentially undesirable artifacts from the hypergraph.
freeze	Method for freezing a hypergraph which prevents it from being modified
double_edge_swap	Swap the edge memberships of two selected nodes, given two edges.
random_edge_shuffle	Randomly redistributes nodes between two hyperedges.

Methods that return other hypergraphs

copy	A deep copy of the hypergraph.
dual	The dual of the hypergraph.

add_edge(*members*, *id=None*, ***attr*)

Add one edge with optional attributes.

Parameters

- **members** (*Iterable*) – An iterable of the ids of the nodes contained in the new edge.
- **id** (*hashable, optional*) – Id of the new edge. If None (default), a unique numeric ID will be created.
- ****attr** (*dict, optional*) – Attributes of the new edge.

Raises

XGIError – If *members* is empty.

See also:

add_edges_from
Add a collection of edges.

set_edge_attributes

Examples

Add edges with or without specifying an edge id.

```
>>> import xgi
>>> H = xgi.Hypergraph()
>>> H.add_edge([1, 2, 3])
>>> H.add_edge([3, 4], id='myedge')
>>> H.edges
EdgeView((0, 'myedge'))
```

Access attributes using square brackets. By default no attributes are created.

```
>>> H.edges[0]
{}
>>> H.add_edge([1, 4], color='red', place='peru')
>>> H.edges
EdgeView((0, 'myedge', 1))
>>> H.edges[1]
{'color': 'red', 'place': 'peru'}
```

add_edges_from(*ebunch_to_add*, ***attr*)

Add multiple edges with optional attributes.

Parameters

- **ebunch_to_add** (*Iterable*) – An iterable of edges. This may be an iterable of iterables (Format 1), where each element contains the members of the edge specified as valid node IDs. Alternatively, each element could also be a tuple in any of the following formats:

- Format 2: 2-tuple (members, edge_id), or
- Format 3: 2-tuple (members, attr), or
- Format 4: 3-tuple (members, edge_id, attr),

where *members* is an iterable of node IDs, *edge_id* is a hashable to use as edge ID, and *attr* is a dict of attributes. Finally, *ebunch_to_add* may be a dict of the form *{edge_id: edge_members}* (Format 5).

Formats 2 and 3 are unambiguous because *attr* dicts are not hashable, while *id`s must be. In Formats 2-4, each element of `ebunch_to_add* must have the same length, i.e. you cannot mix different formats. The iterables containing edge members cannot be strings.

- **attr** (***kwargs, optional*) – Additional attributes to be assigned to all edges. Attributes specified via *ebunch_to_add* take precedence over *attr*.

See also:

add_edge
Add a single edge.

add_weighted_edges_from
Convenient way to add weighted edges.

set_edge_attributes

Notes

Adding the same edge twice will create a multi-edge. Currently cannot add empty edges; the method skips over them.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph()
```

When specifying edges by their members only, numeric edge IDs will be assigned automatically.

```
>>> H.add_edges_from([[0, 1], [1, 2], [2, 3, 4]])
>>> H.edges.members(dtype=dict)
{0: {0, 1}, 1: {1, 2}, 2: {2, 3, 4}}
```

Custom edge ids can be specified using a dict.

```
>>> H = xgi.Hypergraph()
>>> H.add_edges_from({'one': [0, 1], 'two': [1, 2], 'three': [2, 3, 4]})
>>> H.edges.members(dtype=dict)
{'one': {0, 1}, 'two': {1, 2}, 'three': {2, 3, 4}}
```

You can use the dict format to easily add edges from another hypergraph.

```
>>> H2 = xgi.Hypergraph()
>>> H2.add_edges_from(H.edges.members(dtype=dict))
>>> H.edges == H2.edges
True
```

Alternatively, edge ids can be specified using an iterable of 2-tuples.

```
>>> H = xgi.Hypergraph()
>>> H.add_edges_from([([0, 1], 'one'), ([1, 2], 'two'), ([2, 3, 4], 'three')])
>>> H.edges.members(dtype=dict)
{'one': {0, 1}, 'two': {1, 2}, 'three': {2, 3, 4}}
```

Attributes for each edge may be specified using a 2-tuple for each edge. Numeric IDs will be assigned automatically.

```
>>> H = xgi.Hypergraph()
>>> edges = [
...     ([0, 1], {'color': 'red'}),
...     ([1, 2], {'age': 30}),
...     ([2, 3, 4], {'color': 'blue', 'age': 40}),
... ]
>>> H.add_edges_from(edges)
>>> {e: H.edges[e] for e in H.edges}
{0: {'color': 'red'}, 1: {'age': 30}, 2: {'color': 'blue', 'age': 40}}
```

Attributes and custom IDs may be specified using a 3-tuple for each edge.

```
>>> H = xgi.Hypergraph()
>>> edges = [
...     ([0, 1], 'one', {'color': 'red'}),
...     ([1, 2], 'two', {'age': 30}),
...     ([2, 3, 4], 'three', {'color': 'blue', 'age': 40}),
... ]
>>> H.add_edges_from(edges)
>>> {e: H.edges[e] for e in H.edges}
{'one': {'color': 'red'}, 'two': {'age': 30}, 'three': {'color': 'blue', 'age':␣
↪40}}
```

add_node(*node*, ***attr*)

Add one node with optional attributes.

Parameters

- **node** (*node*) – A node can be any hashable Python object except None.
- **attr** (*keyword arguments, optional*) – Set or change node attributes using key=value.

See also:

add_nodes_from, *set_node_attributes*

Notes

If node is already in the hypergraph, its attributes are still updated.

add_node_to_edge(*edge*, *node*)

Add one node to an existing edge.

If the node or edge IDs do not exist, they are created.

Parameters

- **edge** (*hashable*) – edge ID
- **node** (*hashable*) – node ID

See also:

add_node, *add_edge*, *remove_node_from_edge*

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph()
>>> H.add_edge(['apple', 'banana'], 'fruits')
>>> H.add_node_to_edge('fruits', 'pear')
>>> H.add_node_to_edge('veggies', 'lettuce')
>>> d = H.edges.members(dtype=dict)
>>> {id: sorted(list(e)) for id, e in d.items()}
{'fruits': ['apple', 'banana', 'pear'], 'veggies': ['lettuce']}
```

add_nodes_from(*nodes_for_adding*, ***attr*)

Add multiple nodes with optional attributes.

Parameters

- **nodes_for_adding** (*iterable*) – An iterable of nodes (list, dict, set, etc.). OR An iterable of (node, attribute dict) tuples. Node attributes are updated using the attribute dict.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified via keyword arguments.

See also:

add_node, *set_node_attributes*

add_weighted_edges_from(*ebunch*, *weight='weight'*, ***attr*)

Add multiple weighted edges with optional attributes.

Parameters

- **ebunch_to_add** (*iterable of edges*) – Each edge given in the list or container will be added to the graph. The edges must be given as tuples of the form (node1, node2, ..., noden, weight).
- **weight** (*string, optional*) – The attribute name for the edge weights to be added, by default "weight".
- **attr** (*keyword arguments, optional*) – Edge attributes to add/update for all edges.

See also:

add_edge
Add a single edge.

add_edges_from
Add multiple edges.

set_edge_attributes, get_edge_attributes

Notes

Adding the same edge twice creates a multiedge.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph()
>>> edges = [(0, 1, 0.3), (0, 2, 0.8)]
>>> H.add_weighted_edges_from(edges)
>>> H.edges[0]
{'weight': 0.3}
```

cleanup(*isolates=False*, *singletons=False*, *multiedges=False*, *connected=True*, *relabel=True*, *in_place=True*)

Removes potentially undesirable artifacts from the hypergraph.

Parameters

- **isolates** (*bool, optional*) – Whether isolated nodes are allowed, by default False.
- **singletons** (*bool, optional*) – Whether singleton edges are allowed, by default False.
- **multiedges** (*bool, optional*) – Whether multiedges are allowed, by default False.
- **connected** (*bool, optional*) – Whether the returned hypergraph should be connected. If true, returns the hypergraph induced on the largest connected component. By default, False.
- **relabel** (*bool, optional*) – Whether to convert all node and edge labels to sequential integers, by default True.
- **in_place** (*bool, optional*) – Whether to modify the current hypergraph or output a new one, by default True.

clear(*hypergraph_attr=True*)

Remove all nodes and edges from the graph.

Also removes node and edge attributes, and optionally hypergraph attributes.

Parameters
hypergraph_attr (*bool, optional*) – Whether to remove hypergraph attributes as well. By default, True.

clear_edges()

Remove all edges from the graph without altering any nodes.

copy()

A deep copy of the hypergraph.

A deep copy of the hypergraph, including node, edge, and hypergraph attributes.

Returns
H – A copy of the hypergraph.

Return type
Hypergraph

double_edge_swap(*n_id1*, *n_id2*, *e_id1*, *e_id2*)

Swap the edge memberships of two selected nodes, given two edges.

Parameters

- **n_id1** (*hashable*) – The ID of the first node, originally a member of the first edge.
- **n_id2** (*hashable*) – The ID of the second node, originally a member of the second edge.
- **e_id1** (*hashable*) – The ID of the first edge.
- **e_id2** (*hashable*) – The ID of the second edge.

Raises

- **IDNotFound** – If user specifies nodes or edges that do not exist or nodes that are not part of edges.
- **XGIError** – If the swap does not preserve edge sizes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [3, 4]])
>>> H.double_edge_swap(1, 4, 0, 1)
>>> H.edges.members()
[{2, 3, 4}, {1, 3}]
```

dual()

The dual of the hypergraph.

In the dual, nodes become edges and edges become nodes.

Returns
The dual of the hypergraph.

Return type
Hypergraph

property edges

An EdgeView of this network.

freeze()

Method for freezing a hypergraph which prevents it from being modified

See also:

frozen
Method that raises an error when a user tries to modify the hypergraph

is_frozen
Check whether a hypergraph is frozen

Examples

```
>>> import xgi
>>> edges = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(edges)
>>> H.freeze()
>>> H.add_node(5)
Traceback (most recent call last):
xgi.exception.XGIError: Frozen higher-order network can't be modified
```

property is_frozen
Checks whether a dihypergraph is frozen

Returns
True if hypergraph is frozen, false if not.

Return type
bool

See also:

freeze
A method to prevent a hypergraph from being modified.

Examples

```
>>> import xgi
>>> edges = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(edges)
>>> H.freeze()
>>> H.is_frozen
True
```

merge_duplicate_edges(*rename='first'*, *merge_rule='first'*, *multiplicity=None*)
Merges edges which have the same members.

Parameters

- **rename** (*str, optional*) – Either “first” (default), “tuple”, or “new”. If “first”, the new edge ID is the first of the sorted duplicate edge IDs. If “tuple”, the new edge ID is a tuple of the sorted duplicate edge IDs. If “new”, a new ID will be selected automatically.
- **merge_rule** (*str, optional*) – Either “first” (default) or “union”. If “first”, takes the attributes of the first duplicate. If “union”, takes the set of attributes of all the duplicates.
- **multiplicity** (*str, optional*) – The attribute in which to store the multiplicity of the hyperedge, by default None.

Raises
XGIError – If invalid rename or merge_rule specified.

Warns

- **If the user chooses merge_rule=”union”. Tells the**

- **user that they can no longer draw based on this stat.**

Examples

```
>>> import xgi
>>> edges = [{1, 2}, {1, 2}, {1, 2}, {3, 4, 5}, {3, 4, 5}]
>>> edge_attrs = dict()
>>> edge_attrs[0] = {"color": "blue"}
>>> edge_attrs[1] = {"color": "red", "weight": 2}
>>> edge_attrs[2] = {"color": "yellow"}
>>> edge_attrs[3] = {"color": "purple"}
>>> edge_attrs[4] = {"color": "purple", "name": "test"}
>>> H = xgi.Hypergraph(edges)
>>> H.set_edge_attributes(edge_attrs)
>>> H.edges
EdgeView((0, 1, 2, 3, 4))
```

There are several ways to rename the duplicate edges after merging:

1. The merged edge ID is the first duplicate edge ID.

```
>>> H1 = H.copy()
>>> H1.merge_duplicate_edges()
>>> H1.edges
EdgeView((0, 3))
```

2. The merged edge ID is a tuple of all the duplicate edge IDs.

```
>>> H2 = H.copy()
>>> H2.merge_duplicate_edges(rename="tuple")
>>> H2.edges
EdgeView(((0, 1, 2), (3, 4)))
```

3. The merged edge ID is assigned a new edge ID.

```
>>> H3 = H.copy()
>>> H3.merge_duplicate_edges(rename="new")
>>> H3.edges
EdgeView((5, 6))
```

We can also specify how we would like to combine the attributes of the merged edges:

1. The attributes are the attributes of the first merged edge.

```
>>> H4 = H.copy()
>>> H4.merge_duplicate_edges()
>>> H4.edges[0]
{'color': 'blue'}
```

2. The attributes are the union of every attribute that each merged edge has. If a duplicate edge doesn't have that attribute, it is set to None.

```
>>> H5 = H.copy()
>>> H5.merge_duplicate_edges(merge_rule="union")
>>> H5.edges[0] == {'color': {'blue', 'red', 'yellow'}, 'weight':{2, None}}
True
```

3. We can also set the attributes to the intersection, i.e., if a particular attribute is the same across the duplicate edges, we use this attribute, otherwise, we set it to None.

```
>>> H6 = H.copy()
>>> H6.merge_duplicate_edges(merge_rule="intersection")
>>> H6.edges[0] == {'color': None, 'weight': None}
True
>>> H6.edges[3] == {'color': 'purple', 'name': None}
True
```

We can also choose to store the multiplicity of the edge as an attribute. The user simply provides the string of the attribute which stores it. Note that this will not prevent other attributes from being over written (e.g., weight), so be careful that the attribute is not already in use.

```
>>> H7 = H.copy()
>>> H7.merge_duplicate_edges(multiplicity="mult")
>>> H7.edges[0]['mult'] == 3
True
```

property nodes

A `NodeView` of this network.

property num_edges

The number of edges in the hypergraph.

Returns
The number of edges in the hypergraph.

Return type
int

See also:

num_nodes
returns the number of nodes in the hypergraph

Examples

```
>>> import xgi
>>> hyperedge_list = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(hyperedge_list)
>>> H.num_edges
2
```

property num_nodes

The number of nodes in the hypergraph.

Returns
The number of nodes in the hypergraph.

Return type
int

See also:

num_edges
returns the number of edges in the hypergraph

Examples

```
>>> import xgi
>>> hyperedge_list = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(hyperedge_list)
>>> H.num_nodes
4
```

random_edge_shuffle(*e_id1=None*, *e_id2=None*)

Randomly redistributes nodes between two hyperedges.

The process is as follows:

1. randomly select two hyperedges
2. place all their nodes into a single bucket
3. randomly redistribute the nodes between those two hyperedges

Parameters

- **e_id1** (*node ID, optional*) – ID of first edge to shuffle.
- **e_id2** (*node ID, optional*) – ID of second edge to shuffle.

Note: After shuffling, the sizes of the two hyperedges are unchanged. Edge IDs and attributes are also unchanged. If the same node appears in both hyperedges, then this is still true after reshuffling. If either *e_id1* or *e_id2* is not provided, then two random edges are selected.

Philip S C., 2020. "Configuration models of random hypergraphs." Journal of Complex Networks, 8(3). https://doi.org/10.1093/comnet/cnaa018

Example

```
>>> import xgi
>>> random.seed(42)
>>> H = xgi.Hypergraph([[1, 2, 3], [3, 4], [4, 5]])
>>> H.random_edge_shuffle()
>>> H.edges.members()
[{2, 4, 5}, {3, 4}, {1, 3}]
```

remove_edge(*id*)

Remove one edge.

Parameters
id (*Hashable*) – edge ID to remove

Raises
XGIError – If no edge has that ID.

See also:

remove_edges_from
Remove multiple edges.

remove_edges_from(*ebunch*)

Remove multiple edges.

Parameters
ebunch (*Iterable*) – Edges to remove.

Raises
xgi.exception.IDNotFound – If an id in ebunch is not part of the network.

See also:

remove_edge
remove a single edge.

remove_node(*n*, *strong=False*)

Remove a single node.

The removal may be weak (default) or strong. In weak removal, the node is removed from each of its containing edges. If it is contained in any singleton edges, then these are also removed. In strong removal, all edges containing the node are removed, regardless of size.

Parameters

- **n** (*node*) – A node in the hypergraph
- **strong** (*bool, optional*) – Whether to execute weak or strong removal. By default, False.

Raises
XGIError – If n is not in the hypergraph.

See also:

remove_nodes_from

remove_node_from_edge(*edge*, *node*)

Remove a node from an existing edge.

Parameters

- **edge** (*hashable*) – The edge ID
- **node** (*hashable*) – The node ID

Raises
XGIError – If either the node or edge does not exist.

See also:

remove_node, *remove_edge*, *add_node_to_edge*

Notes

If edge is left empty as a result of removing node from it, the edge is also removed.

remove_nodes_from(*nodes*)

Remove multiple nodes.

Parameters

nodes (*iterable*) – An iterable of nodes.

See also:

remove_node

set_edge_attributes(*values*, *name=None*)

Set the edge attributes from a value or a dictionary of values.

Parameters

- **values** (*scalar value, dict-like*) – What the edge attribute should be set to. If *values* is not a dictionary, then it is treated as a single attribute value that is then applied to every edge in *H*. This means that if you provide a mutable object, like a list, updates to that object will be reflected in the edge attribute for each edge. The attribute name will be *name*. If *values* is a dict or a dict of dict, it should be keyed by edge ID to either an attribute value or a dict of attribute key/value pairs used to update the edge's attributes.
- **name** (*string, optional*) – Name of the edge attribute to set if values is a scalar. By default, None.

See also:

set_node_attributes, *add_edge*, *add_edges_from*

Notes

Note that if the dict contains edge IDs that are not in *H*, they are silently ignored.

set_node_attributes(*values*, *name=None*)

Sets node attributes from a given value or dictionary of values.

Parameters

- **values** (*scalar value, dict-like*) – What the node attribute should be set to. If *values* is not a dictionary, then it is treated as a single attribute value that is then applied to every node in *H*. This means that if you provide a mutable object, like a list, updates to that object will be reflected in the node attribute for every node. The attribute name will be *name*.

  If *values* is a dict or a dict of dict, it should be keyed by node to either an attribute value or a dict of attribute key/value pairs used to update the node's attributes.
- **name** (*string, optional*) – Name of the node attribute to set if values is a scalar, by default None.

See also:

set_edge_attributes, *add_node*, *add_nodes_from*

Notes

After computing some property of the nodes of a hypergraph, you may want to assign a node attribute to store the value of that property for each node.

If you provide a list as the second argument, updates to the list will be reflected in the node attribute for each node.

If you provide a dictionary of dictionaries as the second argument, the outer dictionary is assumed to be keyed by node to an inner dictionary of node attributes for that node.

Note that if the dictionary contains nodes that are not in *G*, the values are silently ignored.

update(*, *edges=None*, *nodes=None*)

Add nodes or edges to the hypergraph.

Parameters

- **edges** (*Iterable, optional*) – Edges to be added. By default, None.
- **nodes** (*Iterable, optional*) – Nodes to be added. By default, None.

See also:

add_edges_from
Add multiple edges.

add_nodes_from
Add multiple nodes.

CHAPTER

FOURTEEN

XGI.CORE.SIMPLICIALCOMPLEX.SIMPLICIALCOMPLEX

class xgi.core.simplicialcomplex.**SimplicialComplex**(*incoming_data=None*, ***attr*)

Bases: *Hypergraph*

A class to represent undirected simplicial complexes.

A simplicial complex is a collection of subsets of a set of *nodes* or *vertices*. It is a pair (V, E), where V is a set of elements called *nodes* or *vertices*, and E is a set whose elements are subsets of V, that is, each $e \in E$ satisfies $e \subset V$. The elements of E are called *simplices*. Additionally, if a simplex is part of a simplicial complex, all its faces must be too. This makes simplicial complexes a special case of hypergraphs.

The SimplicialComplex class allows any hashable object as a node and can associate attributes to each node, simplex, or the simplicial complex itself, in the form of key/value pairs.

Parameters

- **incoming_data** (*input simplicial complex data, optional*) – Data to initialize the simplicial complex. If None (default), an empty simplicial complex is created, i.e. one with no nodes or simplices. The data can be in the following formats:
  - simplex list
  - simplex dictionary
  - 2-column Pandas dataframe (bipartite edges)
  - Incidence matrix: numpy ndarray or scipy.sparse array
  - SimplicialComplex object
  - Hypergraph object
- ****attr** (*dict, optional*) – Attributes to add to the simplicial complex as key, value pairs. By default, None.

See also:

Hypergraph

Notes

Unique IDs are assigned to each node and simplex internally and are used to refer to them throughout.

The *attr* keyword arguments are added as simplicial complex attributes. To add node or simplex attributes see `add_node()` and *add_simplex()*. Methods such as *add_simplex()* replace `Hypergraph` methods such as *add_edge()* which here raise an error.

Examples

```
>>> import xgi
>>> S = xgi.SimplicialComplex([[1, 2, 3], [4], [5, 6], [6, 7, 8]])
>>> S.nodes
NodeView((1, 2, 3, 4, 5, 6, 7, 8))
>>> S.edges
EdgeView((0, 1, 2, 3, 4, 5, 6, 7, 8, 9))
```

Attributes

edges	An `EdgeView` of this network.
nodes	A `NodeView` of this network.
num_edges	The number of edges in the hypergraph.
num_nodes	The number of nodes in the hypergraph.
is_frozen	Checks whether a simplicial complex is frozen

Methods

add_node	Add one node with optional attributes.
add_nodes_from	Add multiple nodes with optional attributes.
add_simplex	Add a simplex to the simplicial complex, and all its subfaces that do not exist yet.
add_simplices_from	Add multiple edges with optional attributes.
add_weighted_simplices_from	Add weighted simplices in *ebunch_to_add* with specified weight attr
remove_simplex_id	Remove a simplex with a given id.
remove_simplex_ids_from	Remove all simplicies specified in ebunch.
remove_node	Remove a single node.
remove_nodes_from	Remove multiple nodes.
close	Adds all missing subfaces to the complex.
has_simplex	Whether a simplex appears in the simplicial complex.
cleanup	Removes potentially undesirable artifacts from the hypergraph.
freeze	Method for freezing a simplicial complex which prevents it from being modified

Inherited methods that cannot be used

add_edge	Deprecated in SimplicialComplex.
add_edges_from	Deprecated in SimplicialComplex.
add_weighted_edges_from	Deprecated in SimplicialComplex.
remove_edge	Deprecated in SimplicialComplex.
remove_edges_from	Deprecated in SimplicialComplex.

add_edge(*edge*, *id=None*, ***attr*)

Deprecated in SimplicialComplex. Use add_simplex instead

add_edges_from(*ebunch_to_add*, *max_order=None*, ***attr*)

Deprecated in SimplicialComplex. Use add_simplices_from instead

add_node_to_edge(*edge*, *node*)

add_node_to_edge is not implemented in SimplicialComplex.

add_simplex(*members*, *id=None*, ***attr*)

Add a simplex to the simplicial complex, and all its subfaces that do not exist yet.

Simplex attributes can be specified with keywords or by directly accessing the simplex's attribute dictionary. The attributes do not propagate to the subfaces.

Parameters

- **members** (*Iterable*) – An iterable of the ids of the nodes contained in the new simplex.
- **id** (*hashable, optional*) – Id of the new simplex. If None (default), a unique numeric ID will be created.
- ****attr** (*dict, optional*) – Attributes of the new simplex.

Raises

XGIError – If *members* is empty.

See also:

add_simplices_from

Add a collection of simplices.

Notes

Currently cannot add empty simplices.

Examples

Add simplices with or without specifying an simplex id.

```
>>> import xgi
>>> S = xgi.SimplicialComplex()
>>> S.add_simplex([1, 2, 3])
>>> S.edges.members()
[frozenset({1, 2, 3}), frozenset({2, 3}),
    frozenset({1, 2}), frozenset({1, 3})]
```

(continues on next page)

(continued from previous page)

```
>>> S.add_simplex([3, 4], id='myedge')
>>> S.edges
EdgeView((0, 1, 2, 3, 'myedge'))
```

Access attributes using square brackets. By default no attributes are created.

```
>>> S.edges[0]
{}
>>> S.add_simplex([1, 4], color='red', place='peru')
>>> S.edges
EdgeView((0, 1, 2, 3, 'myedge', 4))
>>> S.edges[4]
{'color': 'red', 'place': 'peru'}
```

add_simplices_from(*ebunch_to_add*, *max_order=None*, ***attr*)

Add multiple edges with optional attributes.

Parameters

- **ebunch_to_add** (*Iterable*) – An iterable of simplices. This may be an iterable of iterables (Format 1), where each element contains the members of the simplex specified as valid node IDs. Alternatively, each element could also be a tuple in any of the following formats:
  - Format 2: 2-tuple (members, simplex_id), or
  - Format 3: 2-tuple (members, attr), or
  - Format 4: 3-tuple (members, simplex_id, attr),

  where *members* is an iterable of node IDs, *simplex_id* is a hashable to use as simplex ID, and *attr* is a dict of attributes. Finally, *ebunch_to_add* may be a dict of the form *{simplex_id: simplex_members}* (Format 5).

  Formats 2 and 3 are unambiguous because *attr* dicts are not hashable, while *id`s must be. In Formats 2-4, each element of `ebunch_to_add* must have the same length, i.e. you cannot mix different formats. The iterables containing simplex members cannot be strings.
- **max_order** (*int, optional*) – Maximal dimension of simplices to add. If None (default), adds all simplices. If int, and *ebunch_to_add* contains simplices of order > *max_order*, creates and adds all its subfaces up to *max_order*.
- **attr** (***kwargs, optional*) – Additional attributes to be assigned to all simplices. Attributes specified via *ebunch_to_add* take precedence over *attr*.

See also:

add_simplex
add a single simplex

add_weighted_simplices_from
convenient way to add weighted simplices

Notes

Adding the same simplex twice will add it only once. Currently cannot add empty simplices; the method skips over them.

Examples

```
>>> import xgi
>>> S = xgi.SimplicialComplex()
```

When specifying simplices by their members only, numeric simplex IDs will be assigned automatically.

```
>>> S.add_simplices_from([[0, 1], [1, 2], [2, 3, 4]])
>>> S.edges.members(dtype=dict)
{0: frozenset({0, 1}), 1: frozenset({1, 2}), 2: frozenset({2, 3, 4}), 3:␣
↪frozenset({2, 3}), 4: frozenset({2, 4}), 5: frozenset({3, 4})}
```

Custom simplex ids can be specified using a dict.

```
>>> S = xgi.SimplicialComplex()
>>> S.add_simplices_from({'one': [0, 1], 'two': [1, 2], 'three': [2, 3, 4]})
>>> S.edges.members(dtype=dict)
{'one': frozenset({0, 1}), 'two': frozenset({1, 2}), 'three': frozenset({2, 3,␣
↪4}), 0: frozenset({2, 3}), 1: frozenset({2, 4}), 2: frozenset({3, 4})}
```

You can use the dict format to easily add simplices from another simplicial complex.

```
>>> S2 = xgi.SimplicialComplex()
>>> S2.add_simplices_from(S.edges.members(dtype=dict))
>>> list(S.edges) == list(S2.edges)
True
```

Alternatively, simplex ids can be specified using an iterable of 2-tuples.

```
>>> S = xgi.SimplicialComplex()
>>> S.add_simplices_from([([0, 1], 'one'), ([1, 2], 'two'), ([2, 3, 4], 'three
↪')])
>>> S.edges.members(dtype=dict)
{'one': frozenset({0, 1}), 'two': frozenset({1, 2}), 'three': frozenset({2, 3,␣
↪4}), 0: frozenset({2, 3}), 1: frozenset({2, 4}), 2: frozenset({3, 4})}
```

Attributes for each simplex may be specified using a 2-tuple for each simplex. Numeric IDs will be assigned automatically.

```
>>> S = xgi.SimplicialComplex()
>>> simplices = [
...     ([0, 1], {'color': 'red'}),
...     ([1, 2], {'age': 30}),
...     ([2, 3, 4], {'color': 'blue', 'age': 40}),
... ]
>>> S.add_simplices_from(simplices)
>>> {e: S.edges[e] for e in S.edges}
{0: {'color': 'red'}, 1: {'age': 30}, 2: {'color': 'blue', 'age': 40}, 3: {}, 4:
↪ {}, 5: {}}
```

Attributes and custom IDs may be specified using a 3-tuple for each simplex.

```
>>> S = xgi.SimplicialComplex()
>>> simplices = [
...     ([0, 1], 'one', {'color': 'red'}),
...     ([1, 2], 'two', {'age': 30}),
...     ([2, 3, 4], 'three', {'color': 'blue', 'age': 40}),
... ]
>>> S.add_simplices_from(simplices)
>>> {e: S.edges[e] for e in S.edges}
{'one': {'color': 'red'}, 'two': {'age': 30}, 'three': {'color': 'blue', 'age':␣
↪40}, 0: {}, 1: {}, 2: {}}
```

add_weighted_edges_from(*ebunch_to_add*, *max_order=None*, *weight='weight'*, ***attr*)

Deprecated in SimplicialComplex. Use add_weighted_simplices_from instead

add_weighted_simplices_from(*ebunch_to_add*, *max_order=None*, *weight='weight'*, ***attr*)

Add weighted simplices in *ebunch_to_add* with specified weight attr

Parameters

- **ebunch_to_add** (*iterable of simplices*) – Each simplex given in the list or container will be added to the graph. The simplices must be given as tuples of the form (node1, node2, ..., noden, weight).
- **max_order** (*int, optional*) – The maximum order simplex to add, by default None.
- **weight** (*string, optional*) – The attribute name for the simplex weights to be added. By default, "weight".
- **attr** (*keyword arguments, optional (default= no attributes)*) – simplex attributes to add/update for all simplices.

See also:

add_simplex
add a single simplex

add_simplices_from
add multiple simplices

Notes

Adding the same simplex twice will add it only once.

Example

```
>>> import xgi
>>> S = xgi.SimplicialComplex()
>>> simplices = [(0, 1, 0.3), (0, 2, 0.8)]
>>> S.add_weighted_simplices_from(simplices)
>>> S.edges[0]
{'weight': 0.3}
```

cleanup(*isolates=False*, *connected=True*, *relabel=True*, *in_place=True*)

Removes potentially undesirable artifacts from the hypergraph.

Parameters

- **isolates** (*bool, optional*) – Whether isolated nodes are allowed, by default False.
- **singletons** (*bool, optional*) – Whether singleton edges are allowed, by default False.
- **multiedges** (*bool, optional*) – Whether multiedges are allowed, by default False.
- **connected** (*bool, optional*) – Whether the returned hypergraph should be connected. If true, returns the hypergraph induced on the largest connected component. By default, False.
- **relabel** (*bool, optional*) – Whether to convert all node and edge labels to sequential integers, by default True.
- **in_place** (*bool, optional*) – Whether to modify the current hypergraph or output a new one, by default True.

close()

Adds all missing subfaces to the complex.

See also:

add_simplex
add a single simplex

add_weighted_simplices_from
convenient way to add weighted simplices

Notes

Adding the same simplex twice will add it only once. Currently cannot add empty simplices; the method skips over them.

copy()

A deep copy of the simplicial complex.

A deep copy of the simplicial complex, including node, edge, and network attributes.

Returns
S – A copy of the simplicial complex.

Return type
SimplicialComplex

freeze()

Method for freezing a simplicial complex which prevents it from being modified

See also:

frozen
Method that raises an error when a

user

is_frozen
Check whether a simplicial complex is frozen

Examples

```
>>> import xgi
>>> edges = [[1, 2], [2, 3, 4]]
>>> S = xgi.SimplicialComplex(edges)
>>> S.freeze()
>>> S.add_node(5)
Traceback (most recent call last):
xgi.exception.XGIError: Frozen higher-order network can't be modified
```

has_simplex(*simplex*)

Whether a simplex appears in the simplicial complex.

Parameters
simplex (*list or set*) – An iterable of hashables that specifies an simplex

Returns
Whether or not simplex is as a simplex in the simplicial complex.

Return type
bool

Examples

```
>>> import xgi
>>> S = xgi.SimplicialComplex([[1, 2], [2, 3, 4]])
>>> S.has_simplex([1, 2])
True
>>> S.has_simplex({1, 3})
False
```

property is_frozen

Checks whether a simplicial complex is frozen

Returns
True if simplicial complex is frozen, false if not.

Return type
bool

See also:

freeze
A method to prevent a simplicial complex from being modified.

Examples

```
>>> import xgi
>>> edges = [[1, 2], [2, 3, 4]]
>>> S = xgi.SimplicialComplex(edges)
>>> S.freeze()
>>> S.is_frozen
True
```

remove_edge(*id*)

Deprecated in SimplicialComplex. Use remove_simplex_id instead

remove_edges_from(*ebunch*)

Deprecated in SimplicialComplex. Use remove_simplex_ids_from instead

remove_node(*n*)

Remove a single node.

The removal is strong meaning that all edges containing the node are removed.

Parameters
n (*node*) – A node in the simplicial complex

Raises
XGIError – If n is not in the simplicial complex.

See also:

remove_nodes_from

remove_nodes_from(*nodes*)

Remove multiple nodes.

Parameters
nodes (*iterable*) – An iterable of nodes.

See also:

remove_node

remove_simplex_id(*id*)

Remove a simplex with a given id.

This also removes all simplices of which this simplex is face, to preserve the simplicial complex structure.

Parameters
id (*Hashable*) – edge ID to remove

Raises
XGIError – If no edge has that ID.

See also:

remove_edges_from
remove a collection of edges

remove_simplex_ids_from(*ebunch*)

Remove all simplicies specified in ebunch.

Parameters

ebunch (*list or iterable of hashables*) – Each edge id given in the list or iterable will be removed from the Simplicialcomplex.

Raises

xgi.exception.IDNotFound – If an id in ebunch is not part of the network.

See also:

remove_simplex_id
remove a single simplex by ID.

CHAPTER

FIFTEEN

XGI.CORE.DIHYPERGRAPH.DIHYPERGRAPH

Warning: This is an experimental module. It is not yet stable and may change in the future.

class xgi.core.dihypergraph.**DiHypergraph**(*incoming_data=None*, ***attr*)

Bases: object

A dihypergraph is a collection of directed interactions of arbitrary size.

Warning: This is currently an experimental feature.

More formally, a directed hypergraph (dihypergraph) is a pair (V, E), where V is a set of elements called *nodes* or *vertices*, and E is the set of directed hyperedges. A directed hyperedge is an ordered pair, (e^+, e^-), where $e^+ \subset V$, the set of senders, is known as the "tail" and $e^- \subset V$, the set of receivers, is known as the "head". The equivalent undirected edge, is $e = e^+ \cap e^-$ and the edge size is defined as $|e|$.

The DiHypergraph class allows any hashable object as a node and can associate attributes to each node, edge, or the hypergraph itself, in the form of key/value pairs.

Multiedges and self-loops are allowed.

Parameters

- **incoming_data** (*input directed hypergraph data (optional, default: None)*) – Data to initialize the dihypergraph. If None (default), an empty hypergraph is created, i.e. one with no nodes or edges. The data can be in the following formats:
  - directed hyperedge list
  - directed hyperedge dictionary
  - DiHypergraph object.
- ****attr** (*dict, optional, default: None*) – Attributes to add to the hypergraph as key, value pairs.

Notes

Unique IDs are assigned to each node and edge internally and are used to refer to them throughout.

The *attr* keyword arguments are added as hypergraph attributes. To add node or edge attributes see *add_node()* and *add_edge()*.

In addition to the methods listed in this page, other methods defined in the *stats* package are also accessible via the *DiHypergraph* class. For more details, see the tutorial.

References

Bretto, Alain. "Hypergraph theory: An introduction." Mathematical Engineering. Cham: Springer (2013).

Examples

```
>>> import xgi
>>> DH = xgi.DiHypergraph([([1, 2, 3], [4]), ([5, 6], [6, 7, 8])])
>>> DH.nodes
DiNodeView((1, 2, 3, 4, 5, 6, 7, 8))
>>> DH.edges
DiEdgeView((0, 1))
>>> [[sorted(h), sorted(t)] for h, t in DH.edges.dimembers()]
[[[1, 2, 3], [4]], [[5, 6], [6, 7, 8]]]
>>> [sorted(e) for e in DH.edges.members()]
[[1, 2, 3, 4], [5, 6, 7, 8]]
```

Attributes

edges	An `DiEdgeView` of this network.
nodes	A `DiNodeView` of this network.
num_edges	The number of directed edges in the dihypergraph.
num_nodes	The number of nodes in the dihypergraph.
is_frozen	Checks whether a hypergraph is frozen

Methods that modify the structure

add_node	Add one node with optional attributes.
add_edge	Add one edge with optional attributes.
add_nodes_from	Add multiple nodes with optional attributes.
add_edges_from	Add multiple directed edges with optional attributes.
remove_node	Remove a single node.
remove_edge	Remove one edge.
remove_nodes_from	Remove multiple nodes.
remove_edges_from	Remove multiple edges.
clear	Remove all nodes and edges from the graph.
copy	A deep copy of the dihypergraph.
cleanup	
freeze	Method for freezing a dihypergraph which prevents it from being modified

add_edge(*members*, *id=None*, ***attr*)

Add one edge with optional attributes.

Parameters

- **members** (*Iterable*) – An list or tuple (size 2) of iterables. The first entry contains the elements of the tail and the second entry contains the elements of the head.
- **id** (*hashable, default None*) – Id of the new edge. If None, a unique numeric ID will be created.
- ****attr** (*dict, optional*) – Attributes of the new edge.

Raises

XGIError – If *members* is empty or is not a list or tuple.

See also:

add_edges_from
Add a collection of edges.

Examples

Add edges with or without specifying an edge id.

```
>>> import xgi
>>> DH = xgi.DiHypergraph()
>>> DH.add_edge(([1, 2, 3], [2, 3, 4]))
>>> DH.add_edge(([3, 4], set()), id='myedge')
```

add_edges_from(*ebunch_to_add*, ***attr*)

Add multiple directed edges with optional attributes.

Parameters

- **ebunch_to_add** (*Iterable*) – Note that here, when we refer to an edge, as in the *add_edge* method, it is a list or tuple (size 2) of iterables. The first entry contains the elements of the tail and the second entry contains the elements of the head.

An iterable of edges. This may be an iterable of edges (Format 1), where each edge is in the format described above.

Alternatively, each element could also be a tuple in any of the following formats:

– Format 2: 2-tuple (edge, edge_id), or

– Format 4: 3-tuple (edge, edge_id, attr),

where *edge* is in the format described above, *edge_id* is a hashable to use as edge ID, and *attr* is a dict of attributes. Finally, *ebunch_to_add* may be a dict of the form *{edge_id: edge_members}* (Format 5).

Formats 2 and 3 are unambiguous because *attr* dicts are not hashable, while *id`s must be. In Formats 2-4, each element of `ebunch_to_add* must have the same length, i.e. you cannot mix different formats. The iterables containing edge members cannot be strings.

- **attr** (***kwargs, optional*) – Additional attributes to be assigned to all edges. Attribues specified via *ebunch_to_add* take precedence over *attr*.

See also:

add_edge
Add a single edge.

Notes

Adding the same edge twice will create a multi-edge. Currently cannot add empty edges; the method skips over them.

Examples

```
>>> import xgi
>>> DH = xgi.DiHypergraph()
```

When specifying edges by their members only, numeric edge IDs will be assigned automatically.

```
>>> DH.add_edges_from([([0, 1], [1, 2]), ([2, 3, 4], [])])
>>> DH.edges.dimembers(dtype=dict)
{0: ({0, 1}, {1, 2}), 1: ({2, 3, 4}, set())}
```

Custom edge ids can be specified using a dict.

```
>>> DH = xgi.DiHypergraph()
>>> DH.add_edges_from({'one': ([0, 1], [1, 2]), 'two': ([2, 3, 4], [])})
>>> DH.edges.dimembers(dtype=dict)
{'one': ({0, 1}, {1, 2}), 'two': ({2, 3, 4}, set())}
```

You can use the dict format to easily add edges from another hypergraph.

```
>>> DH2 = xgi.DiHypergraph()
>>> DH2.add_edges_from(DH.edges.dimembers(dtype=dict))
>>> DH.edges == DH2.edges
True
```

Alternatively, edge ids can be specified using an iterable of 2-tuples.

```
>>> DH = xgi.DiHypergraph()
>>> DH.add_edges_from([([0, 1], [1, 2]), 'one'), (([2, 3, 4], []), 'two')])
>>> DH.edges.dimembers(dtype=dict)
{'one': ({0, 1}, {1, 2}), 'two': ({2, 3, 4}, set())}
```

Attributes for each edge may be specified using a 2-tuple for each edge. Numeric IDs will be assigned automatically.

```
>>> DH = xgi.DiHypergraph()
>>> edges = [
...     (([0, 1], [1, 2]), {'color': 'red'}),
...     (([2, 3, 4], []), {'color': 'blue', 'age': 40}),
... ]
>>> DH.add_edges_from(edges)
>>> {e: DH.edges[e] for e in DH.edges}
{0: {'color': 'red'}, 1: {'color': 'blue', 'age': 40}}
```

Attributes and custom IDs may be specified using a 3-tuple for each edge.

```
>>> DH = xgi.DiHypergraph()
>>> edges = [
...     (([0, 1], [1, 2]), 'one', {'color': 'red'}),
...     (([2, 3, 4], []), 'two', {'color': 'blue', 'age': 40}),
... ]
>>> DH.add_edges_from(edges)
>>> {e: DH.edges[e] for e in DH.edges}
{'one': {'color': 'red'}, 'two': {'color': 'blue', 'age': 40}}
```

add_node(*node*, ***attr*)

Add one node with optional attributes.

Parameters

- **node** (*node*) – A node can be any hashable Python object except None.
- **attr** (*keyword arguments, optional*) – Set or change node attributes using key=value.

See also:

add_nodes_from

Notes

If node is already in the dihypergraph, its attributes are still updated.

add_nodes_from(*nodes_for_adding*, ***attr*)

Add multiple nodes with optional attributes.

Parameters

- **nodes_for_adding** (*iterable*) – An iterable of nodes (list, dict, set, etc.). OR An iterable of (node, attribute dict) tuples. Node attributes are updated using the attribute dict.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Update attributes for all nodes in nodes. Node attributes specified in nodes as a tuple take precedence over attributes specified via keyword arguments.

See also:

add_node

clear(*hypergraph_attr=True*)

Remove all nodes and edges from the graph.

Also removes node and edge attributes, and optionally hypergraph attributes.

Parameters
hypergraph_attr (*bool, optional*) – Whether to remove hypergraph attributes as well. By default, True.

copy()

A deep copy of the dihypergraph.

A deep copy of the dihypergraph, including node, edge, and hypergraph attributes.

Returns
DH – A copy of the hypergraph.

Return type
DiHypergraph

property edges

An `DiEdgeView` of this network.

freeze()

Method for freezing a dihypergraph which prevents it from being modified

See also:

frozen
Method that raises an error when a user tries to modify the hypergraph

is_frozen
Check whether a dihypergraph is frozen

Examples

```
>>> import xgi
>>> diedgelist = [([1, 2], [2, 3, 4])]
>>> DH = xgi.DiHypergraph(diedgelist)
>>> DH.freeze()
>>> DH.add_node(5)
Traceback (most recent call last):
xgi.exception.XGIError: Frozen higher-order network can't be modified
```

property is_frozen

Checks whether a hypergraph is frozen

Returns
True if hypergraph is frozen, false if not.

Return type
bool

See also:

freeze
A method to prevent a hypergraph from being modified.

Examples

```
>>> import xgi
>>> edges = [([1, 2], [2, 3, 4])]
>>> DH = xgi.DiHypergraph(edges)
>>> DH.freeze()
>>> DH.is_frozen
True
```

property nodes

A `DiNodeView` of this network.

property num_edges

The number of directed edges in the dihypergraph.

Returns
The number of directed edges in the dihypergraph.

Return type
int

See also:

num_nodes
returns the number of nodes in the dihypergraph

Examples

```
>>> import xgi
>>> hyperedge_list = [([1, 2], [2, 3, 4])]
>>> DH = xgi.DiHypergraph(hyperedge_list)
>>> DH.num_edges
1
```

property num_nodes

The number of nodes in the dihypergraph.

Returns
The number of nodes in the dihypergraph.

Return type
int

See also:

num_edges
returns the number of edges in the dihypergraph

Examples

```
>>> import xgi
>>> hyperedge_list = [([1, 2], [2, 3, 4])]
>>> DH = xgi.DiHypergraph(hyperedge_list)
>>> DH.num_nodes
4
```

remove_edge(*id*)

Remove one edge.

Parameters
id (*Hashable*) – edge ID to remove

Raises
XGIError – If no edge has that ID.

See also:

remove_edges_from
Remove multiple edges.

remove_edges_from(*ebunch*)

Remove multiple edges.

Parameters
ebunch (*Iterable*) – Edges to remove.

Raises
xgi.exception.IDNotFound – If an id in ebunch is not part of the network.

See also:

remove_edge
remove a single edge.

remove_node(*n*, *strong=False*)

Remove a single node.

The removal may be weak (default) or strong. In weak removal, the node is removed from each of its containing edges. If it is contained in any singleton edges, then these are also removed. In strong removal, all edges containing the node are removed, regardless of size.

Parameters

- **n** (*node*) – A node in the dihypergraph
- **strong** (*bool (default False)*) – Whether to execute weak or strong removal.

Raises
XGIError – If n is not in the dihypergraph.

See also:

remove_nodes_from

remove_nodes_from(*nodes*)

Remove multiple nodes.

Parameters
nodes (*iterable*) – An iterable of nodes.

See also:

remove_node

set_edge_attributes(*values*, *name=None*)

Set the edge attributes from a value or a dictionary of values.

Parameters

- **values** (*scalar value, dict-like*) – What the edge attribute should be set to. If *values* is not a dictionary, then it is treated as a single attribute value that is then applied to every edge in *DH*. This means that if you provide a mutable object, like a list, updates to that object will be reflected in the edge attribute for each edge. The attribute name will be *name*. If *values* is a dict or a dict of dict, it should be keyed by edge ID to either an attribute value or a dict of attribute key/value pairs used to update the edge's attributes.
- **name** (*string, optional*) – Name of the edge attribute to set if values is a scalar. By default, None.

See also:

set_node_attributes, *add_edge*, *add_edges_from*

Notes

Note that if the dict contains edge IDs that are not in *DH*, they are silently ignored.

set_node_attributes(*values*, *name=None*)

Sets node attributes from a given value or dictionary of values.

Parameters

- **values** (*scalar value, dict-like*) – What the node attribute should be set to. If *values* is not a dictionary, then it is treated as a single attribute value that is then applied to every node in *DH*. This means that if you provide a mutable object, like a list, updates to that object will be reflected in the node attribute for every node. The attribute name will be *name*.

  If *values* is a dict or a dict of dict, it should be keyed by node to either an attribute value or a dict of attribute key/value pairs used to update the node's attributes.
- **name** (*string, optional*) – Name of the node attribute to set if values is a scalar, by default None.

See also:

set_edge_attributes, *add_node*, *add_nodes_from*

Notes

After computing some property of the nodes of a hypergraph, you may want to assign a node attribute to store the value of that property for each node.

If you provide a list as the second argument, updates to the list will be reflected in the node attribute for each node.

If you provide a dictionary of dictionaries as the second argument, the outer dictionary is assumed to be keyed by node to an inner dictionary of node attributes for that node.

Note that if the dictionary contains nodes that are not in *G*, the values are silently ignored.

CHAPTER

SIXTEEN

CORE FUNCTIONALITY

Modules

hypergraph	Base class for undirected hypergraphs.
dihypergraph	Base class for directed hypergraphs.
simplicialcomplex	Base class for undirected simplicial complexes.
views	View classes for hypergraphs.
diviews	View classes for dihypergraphs.
globalviews	View of Hypergraphs as a subhypergraph or read-only.

16.1 xgi.core.hypergraph

Base class for undirected hypergraphs.

Classes

Hypergraph	A hypergraph is a collection of subsets of a set of *nodes* or *vertices*.

16.2 xgi.core.dihypergraph

Base class for directed hypergraphs.

Warning: This is currently an experimental feature.

Classes

DiHypergraph	A dihypergraph is a collection of directed interactions of arbitrary size.

16.3 xgi.core.simplicialcomplex

Base class for undirected simplicial complexes.

The SimplicialComplex class allows any hashable object as a node and can associate key/value attribute pairs with each undirected simplex and node.

Multi-simplices are not allowed.

Classes

SimplicialComplex	A class to represent undirected simplicial complexes.

16.4 xgi.core.views

View classes for hypergraphs.

A View class allows for inspection and querying of an underlying object but does not allow modification. This module provides View classes for nodes and edges of a hypergraph. Views are automatically updaed when the hypergraph changes.

Classes

IDView	Base View class for accessing the ids (nodes or edges) of a Hypergraph.
NodeView	An IDView that keeps track of node ids.
EdgeView	An IDView that keeps track of edge ids.

16.4.1 xgi.core.views.IDView

class xgi.core.views.**IDView**(*network*, *ids=None*)

Bases: `Mapping`, `Set`

Base View class for accessing the ids (nodes or edges) of a Hypergraph.

Can optionally keep track of a subset of ids. By default all node ids or all edge ids are kept track of.

Parameters

- **network** (*Hypergraph or Simplicial Complex*) – The underlying network
- **ids** (*iterable*) – A subset of the keys in id_dict to keep track of.

Raises
XGIError – If ids is not a subset of the keys of id_dict.

Methods

from_view	Create a view from another view.
neighbors	Find the neighbors of an ID.
duplicates	Find IDs that have a duplicate.
lookup	Find IDs with the specified bipartite neighbors.
filterby	Filter the IDs in this view by a statistic.
filterby_attr	Filter the IDs in this view by an attribute.

duplicates()

Find IDs that have a duplicate.

An ID has a 'duplicate' if there exists another ID with the same bipartite neighbors.

Returns
A view containing only those IDs with a duplicate.

Return type
IDView

Raises
TypeError – When IDs are of different types. For example, ("a", 1).

Notes

The IDs returned are in an arbitrary order, that is duplicates are not guaranteed to be consecutive. For IDs with the same bipartite neighbors, only the first ID added is not a duplicate.

See also:

IDView.lookup

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[0, 1, 2], [3, 4, 2], [0, 1, 2]])
>>> H.edges.duplicates()
EdgeView((2,))
```

Order does not matter:

```
>>> H = xgi.Hypergraph([[2, 1, 0], [0, 1, 2]])
>>> H.edges.duplicates()
EdgeView((1,))
```

Repetitions matter:

```
>>> H = xgi.Hypergraph([[0, 1], [1, 0]])
>>> H.edges.duplicates()
EdgeView((1,))
```

filterby(*stat*, *val*, *mode='eq'*)

Filter the IDs in this view by a statistic.

Parameters

- **stat** (str or *xgi.stats.NodeStat*/*xgi.stats.EdgeStat*) – *NodeStat*/*EdgeStat* object, or name of a *NodeStat*/*EdgeStat*.
- **val** (*Any*) – Value of the statistic. Usually a single numeric value. When mode is 'between', must be a tuple of exactly two values.
- **mode** (*str or function, optional*) – How to compare each value to *val*. Can be one of the following.
  - 'eq' (default): Return IDs whose value is exactly equal to *val*.
  - 'neq': Return IDs whose value is not equal to *val*.
  - 'lt': Return IDs whose value is less than *val*.
  - 'gt': Return IDs whose value is greater than *val*.
  - 'leq': Return IDs whose value is less than or equal to *val*.
  - 'geq': Return IDs whose value is greater than or equal to *val*.
  - 'between': In this mode, *val* must be a tuple *(val1, val2)*. Return IDs whose value *v* satisfies *val1* <= *v* <= *val2*.
  - function, must be able to call *mode(statistic, val)* and have it map to a bool.

See also:

IDView.filterby_attr : For more details, see the tutorial.

Examples

By default, return the IDs whose value of the statistic is exactly equal to *val*.

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> n = H.nodes
>>> n.filterby('degree', 2)
NodeView((2, 4, 5))
```

Can choose other comparison methods via *mode*.

```
>>> n.filterby('degree', 2, 'eq')
NodeView((2, 4, 5))
>>> n.filterby('degree', 2, 'neq')
NodeView((1, 3))
>>> n.filterby('degree', 2, 'lt')
NodeView((1,))
>>> n.filterby('degree', 2, 'gt')
NodeView((3,))
>>> n.filterby('degree', 2, 'leq')
NodeView((1, 2, 4, 5))
>>> n.filterby('degree', 2, 'geq')
NodeView((2, 3, 4, 5))
```

(continues on next page)

(continued from previous page)

```
>>> n.filterby('degree', (2, 3), 'between')
NodeView((2, 3, 4, 5))
```

Can also pass a `NodeStat` object.

```
>>> n.filterby(n.degree(order=2), 2)
NodeView((3,))
```

filterby_attr(*attr*, *val*, *mode='eq'*, *missing=None*)

Filter the IDs in this view by an attribute.

Parameters

- **attr** (*string*) – The name of the attribute
- **val** (*Any*) – A single value or, in the case of 'between', a list of length 2
- **mode** (*str or function, optional*) – Comparison mode. Valid options are 'eq' (default), 'neq', 'lt', 'gt', 'leq', 'geq', or 'between'. If a function, must be able to call *mode(attribute, val)* and have it map to a bool.
- **missing** (*Any, optional*) – The default value if the attribute is missing. If None (default), ignores those IDs.

See also:

IDView.filterby : Identical method. For more details, see the tutorial.

Notes

Beware of using comparison modes ("lt", "gt", "leq", "geq") when the attribute is a string. For example, the string comparison *'10' < '9'* evaluates to *True*.

classmethod from_view(*view*, *bunch=None*)

Create a view from another view.

Allows to create a view with the same underlying data but with a different bunch.

Parameters

- **view** (IDView) – The view used to initialize the new object
- **bunch** (*iterable*) – IDs the new view will keep track of

Returns

A view that is identical to *view* but keeps track of different IDs.

Return type

IDView

property ids

The ids in this view.

Notes

Do not use this property for membership check. Instead of *x in view.ids*, always use *x in view*. The latter is always faster.

lookup(*neighbors*)

Find IDs with the specified bipartite neighbors.

Parameters
neighbors (*Iterable*) – An iterable of IDs.

Returns
A view containing only those IDs whose bipartite neighbors match *neighbors*.

Return type
IDView

See also:

IDView.duplicates

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[0, 1, 2], [3, 4], [3, 4, 2]])
>>> H.edges.lookup([3, 4])
EdgeView((1,))
>>> H.add_edge([3, 4])
>>> H.edges.lookup([3, 4])
EdgeView((1, 3))
```

Can be used as a boolean check for edge existence:

```
>>> if H.edges.lookup([3, 4]): print('An edge with members [3, 4] exists')
An edge with members [3, 4] exists
```

Can also be used to check for nodes that belong to a particular set of edges:

```
>>> H = xgi.Hypergraph([['a', 'b', 'c'], ['a', 'd', 'e'], ['c', 'd', 'e']])
>>> H.nodes.lookup([0, 1])
NodeView(('a',))
```

neighbors(*id*, *s=1*)

Find the neighbors of an ID.

The neighbors of an ID are those IDs that share at least one bipartite ID.

Parameters

- **id** (*hashable*) – ID to find neighbors of.
- **s** (*int, optional*) – The intersection size s for two edges or nodes to be considered neighbors. By default, 1.

Returns
A set of the neighboring IDs

Return type
set

See also:

`edge_neighborhood`

Examples

```
>>> import xgi
>>> hyperedge_list = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(hyperedge_list)
>>> H.nodes.neighbors(1)
{2}
>>> H.nodes.neighbors(2)
{1, 3, 4}
```

16.4.2 xgi.core.views.NodeView

class xgi.core.views.**NodeView**(*H*, *bunch=None*)

Bases: *IDView*

An IDView that keeps track of node ids.

Parameters

- **hypergraph** (Hypergraph) – The hypergraph whose nodes this view will keep track of.
- **bunch** (*optional iterable, default None*) – The node ids to keep track of. If None (default), keep track of all node ids.

See also:

IDView

Notes

In addition to the methods listed in this page, other methods defined in the *stats* package are also accessible via the *NodeView* class. For more details, see the tutorial.

Attributes

ids	The ids in this view.

Methods

memberships	Get the edge ids of which a node is a member.
isolates	Nodes that belong to no edges.
neighbors	Find the neighbors of an ID.
duplicates	Find IDs that have a duplicate.
lookup	Find IDs with the specified bipartite neighbors.
filterby	Filter the IDs in this view by a statistic.
filterby_attr	Filter the IDs in this view by an attribute.

isolates(*ignore_singletons=False*)

Nodes that belong to no edges.

When ignore_singletons is True, a node is considered isolated from the rest of the hypergraph when it is included in no edges of size two or more. In particular, whether the node is part of any singleton edges is irrelevant to determine whether it is isolated.

When ignore_singletons is False (default), a node is isolated only when it is a member of exactly zero edges, including singletons.

Parameters
ignore_singletons (*bool, optional*) – Whether to consider singleton edges. By default, False.

Return type
NodeView containing the isolated nodes.

See also:

EdgeView.singletons()

memberships(*n=None*)

Get the edge ids of which a node is a member.

Gets all the node memberships for all nodes in the view if n not specified.

Parameters
n (*hashable, optional*) – Node ID. By default, None.

Returns
Edge memberships.

Return type
dict of sets if n is None, otherwise a set

Raises
XGIError – If *n* is not hashable or if it is not in the hypergraph.

16.4.3 xgi.core.views.EdgeView

class xgi.core.views.**EdgeView**(*H*, *bunch=None*)

Bases: *IDView*

An IDView that keeps track of edge ids.

Parameters

- **hypergraph** (Hypergraph) – The hypergraph whose edges this view will keep track of.
- **bunch** (*optional iterable, default None*) – The edge ids to keep track of. If None (default), keep track of all edge ids.

See also:

IDView

Notes

In addition to the methods listed in this page, other methods defined in the *stats* package are also accessible via the *EdgeView* class. For more details, see the tutorial.

Attributes

ids	The ids in this view.

Methods

members	Get the node ids that are members of an edge.
singletons	Edges that contain exactly one node.
maximal	Returns the maximal edges as an EdgeView.
neighbors	Find the neighbors of an ID.
duplicates	Find IDs that have a duplicate.
lookup	Find IDs with the specified bipartite neighbors.
filterby	Filter the IDs in this view by a statistic.
filterby_attr	Filter the IDs in this view by an attribute.

maximal(*strict=False*)

Returns the maximal edges as an EdgeView.

Maximal edges are those that are not subsets of any other edges in the hypergraph. The strict keyword determines whether the subsets are strict or non-strict.

Parameters

strict (*bool, optional*) – Whether maximal edges must strictly include all of its subsets (*strict=True*) or whether maximal multiedges are permitted (*strict=False*), by default False. See Notes for more details.

Returns

The maximal edges

Return type

EdgeView

Notes

This function implements two definitions of maximal hyperedges: strict and non-strict. For the strict case (*strict=True*), we enforce that a maximal edge must strictly include all of its subsets and by this definition, multiedges can't be included. For the non-strict case (*strict=False*), then we add all the maximal multiedges with non-strict inclusion.

There are methods for eliminating these duplicates by running *H.cleanup()* or *H.remove_edges_from(H.edges.duplicates())*

References

https://stackoverflow.com/questions/14106121/efficient-algorithm-for-finding-all-maximal-subsets

Example

```
>>> import xgi
>>> H = xgi.Hypergraph([{1, 2, 3},{1, 2}, {2, 3}, {2}, {2}, {3, 4}, {1, 2, 3}])
>>> H.edges.maximal()
EdgeView((0, 5, 6))
>>> H.edges.maximal().members()
[{1, 2, 3}, {3, 4}, {1, 2, 3}]
```

members(*e=None*, *dtype=<class 'list'>*)

Get the node ids that are members of an edge.

Parameters

- **e** (*hashable, optional*) – Edge ID. By default, None.
- **dtype** (*{list, dict}, optional*) – Specify the type of the return value. By default, list.

Returns

- *list (if dtype is list, default)* – Edge members.
- *dict (if dtype is dict)* – Edge members.
- *set (if e is not None)* – Members of edge e.

Raises

- **TypeError** – If *e* is not None or a hashable
- **XGIError** – If *dtype* is not dict or list
- **IDNotFound** – If *e* does not exist in the hypergraph

singletons()

Edges that contain exactly one node.

Return type
EdgeView containing the singleton edges.

See also:

NodeView.isolates()

16.5 xgi.core.diviews

View classes for dihypergraphs.

A View class allows for inspection and querying of an underlying object but does not allow modification. This module provides View classes for nodes and edges of a dihypergraph. Views are automatically updaed when the dihypergraph changes.

Classes

DiIDView	Base View class for accessing the ids (nodes or edges) of a DiHypergraph.
DiNodeView	An DiIDView that keeps track of node ids.
DiEdgeView	An DiIDView that keeps track of edge ids.

16.5.1 xgi.core.diviews.DiIDView

class xgi.core.diviews.**DiIDView**(*network*, *ids=None*)

Bases: Mapping, Set

Base View class for accessing the ids (nodes or edges) of a DiHypergraph.

Can optionally keep track of a subset of ids. By default all node ids or all edge ids are kept track of.

Parameters

- **id_dict** (*dict*) – The original dict this is a view of.
- **id_attrs** (*dict*) – The original attribute dict this is a view of.
- **ids** (*iterable*) – A subset of the keys in id_dict to keep track of.

Raises

XGIError – If ids is not a subset of the keys of id_dict.

Methods

from_view	Create a view from another view.
filterby	Filter the IDs in this view by a statistic.
filterby_attr	Filter the IDs in this view by an attribute.

filterby(*stat*, *val*, *mode='eq'*)

Filter the IDs in this view by a statistic.

Parameters

- **stat** (str or *xgi.stats.DiNodeStat*/*xgi.stats.DiEdgeStat*) – *DiNodeStat*/*DiEdgeStat* object, or name of a *DiNodeStat*/*DiEdgeStat*.
- **val** (*Any*) – Value of the statistic. Usually a single numeric value. When mode is 'between', must be a tuple of exactly two values.
- **mode** (*str or function, optional*) – How to compare each value to *val*. Can be one of the following.
  - 'eq' (default): Return IDs whose value is exactly equal to *val*.
  - 'neq': Return IDs whose value is not equal to *val*.
  - 'lt': Return IDs whose value is less than *val*.
  - 'gt': Return IDs whose value is greater than *val*.
  - 'leq': Return IDs whose value is less than or equal to *val*.

- 'geq': Return IDs whose value is greater than or equal to *val*.
- 'between': In this mode, *val* must be a tuple *(val1, val2)*. Return IDs whose value *v* satisfies *val1* <= *v* <= *val2*.
- function, must be able to call *mode(statistic, val)* and have it map to a bool.

See also:

IDView.filterby_attr : For more details, see the tutorial.

Examples

By default, return the IDs whose value of the statistic is exactly equal to *val*.

```
>>> import xgi
>>> H = xgi.DiHypergraph([([1, 2, 3], [2, 3, 4, 5]), ([3, 4, 5], [])])
>>> n = H.nodes
>>> n.filterby('degree', 2)
DiNodeView((3, 4, 5))
```

Can choose other comparison methods via *mode*.

```
>>> n.filterby('degree', 2, 'eq')
DiNodeView((3, 4, 5))
>>> n.filterby('degree', 2, 'neq')
DiNodeView((1, 2))
>>> n.filterby('degree', 2, 'lt')
DiNodeView((1, 2))
>>> n.filterby('degree', 2, 'gt')
DiNodeView(())
>>> n.filterby('degree', 2, 'leq')
DiNodeView((1, 2, 3, 4, 5))
>>> n.filterby('degree', 2, 'geq')
DiNodeView((3, 4, 5))
>>> n.filterby('degree', (2, 3), 'between')
DiNodeView((3, 4, 5))
```

filterby_attr(*attr*, *val*, *mode='eq'*, *missing=None*)

Filter the IDs in this view by an attribute.

Parameters

- **attr** (*string*) – The name of the attribute
- **val** (*Any*) – A single value or, in the case of 'between', a list of length 2
- **mode** (*str or function, optional*) – Comparison mode. Valid options are 'eq' (default), 'neq', 'lt', 'gt', 'leq', 'geq', or 'between'. If a function, must be able to call *mode(attribute, val)* and have it map to a bool.
- **missing** (*Any, optional*) – The default value if the attribute is missing. If None (default), ignores those IDs.

See also:

DiIDView.filterby : Identical method. For more details, see the tutorial.

Notes

Beware of using comparison modes ("lt", "gt", "leq", "geq") when the attribute is a string. For example, the string comparison *'10' < '9'* evaluates to *True*.

classmethod from_view(*view*, *bunch=None*)

Create a view from another view.

Allows to create a view with the same underlying data but with a different bunch.

Parameters

- **view** (IDView) – The view used to initialze the new object
- **bunch** (*iterable*) – IDs the new view will keep track of

Returns
A view that is identical to *view* but keeps track of different IDs.

Return type
DiIDView

property ids

The ids in this view.

Notes

Do not use this property for membership check. Instead of *x in view.ids*, always use *x in view*. The latter is always faster.

16.5.2 xgi.core.diviews.DiNodeView

class xgi.core.diviews.**DiNodeView**(*H*, *bunch=None*)

Bases: *DiIDView*

An DiIDView that keeps track of node ids.

> **Warning:** This is currently an experimental feature.

Parameters

- **hypergraph** (DiHypergraph) – The hypergraph whose nodes this view will keep track of.
- **bunch** (*optional iterable, default None*) – The node ids to keep track of. If None (default), keep track of all node ids.

See also:

DiIDView

Notes

In addition to the methods listed in this page, other methods defined in the *stats* package are also accessible via the *NodeView* class. For more details, see the tutorial.

Attributes

ids	The ids in this view.

Methods

memberships	Get the edge ids of which a node is a member.
dimemberships	Get the edge ids of which a node is a member.
filterby	Filter the IDs in this view by a statistic.
filterby_attr	Filter the IDs in this view by an attribute.
isolates	Nodes that belong to no edges.

dimemberships(*n=None*)

Get the edge ids of which a node is a member.

Gets all the node memberships for all nodes in the view if n not specified.

Parameters
n (*hashable, optional*) – Node ID. By default, None.

Returns
otherwise the directed memberships of a single node.

Return type
dict of directed node memberships if n is None,

Raises
XGIError – If *n* is not hashable or if it is not in the hypergraph.

isolates()

Nodes that belong to no edges.

When ignore_singletons is True, a node is considered isolated from the rest of the hypergraph when it is included in no edges of size two or more. In particular, whether the node is part of any singleton edges is irrelevant to determine whether it is isolated.

When ignore_singletons is False (default), a node is isolated only when it is a member of exactly zero edges, including singletons.

Return type
NodeView containing the isolated nodes.

See also:

EdgeView.singletons()

memberships(*n=None*)

Get the edge ids of which a node is a member.

Gets all the node memberships for all nodes in the view if n not specified.

Parameters
n (*hashable, optional*) – Node ID. By default, None.

Returns
Node memberships, regardless of whether that node is a sender or receiver.

Return type
dict of sets if n is None, otherwise a set

Raises
XGIError – If *n* is not hashable or if it is not in the dihypergraph.

16.5.3 xgi.core.diviews.DiEdgeView

class xgi.core.diviews.**DiEdgeView**(*H*, *bunch=None*)

Bases: *DiIDView*

An DiIDView that keeps track of edge ids.

> **Warning:** This is currently an experimental feature.

Parameters

- **hypergraph** (DiHypergraph) – The hypergraph whose edges this view will keep track of.
- **bunch** (*optional iterable, default None*) – The edge ids to keep track of. If None (default), keep track of all edge ids.

See also:

DiIDView

Notes

In addition to the methods listed in this page, other methods defined in the *stats* package are also accessible via the *EdgeView* class. For more details, see the tutorial.

Attributes

ids	The ids in this view.

Methods

members	Get the edges of a directed hypergraph.
dimembers	Get the node ids that are members of an edge.
head	Get the node ids that are in the head of a directed edge.
tail	Get the node ids that are in the tail of a directed edge.
filterby	Filter the IDs in this view by a statistic.
filterby_attr	Filter the IDs in this view by an attribute.

dimembers(*e=None*, *dtype=<class 'list'>*)

Get the node ids that are members of an edge.

Parameters

- **e** (*hashable, optional*) – Edge ID. By default, None.
- **dtype** (*{list, dict}, optional*) – Specify the type of the return value. By default, list.

Returns

- *list (if dtype is list, default)* – Directed edges.
- *dict (if dtype is dict)* – Directed edges.
- *set (if e is not None)* – A single directed edge.
- *In all of these cases, a directed edge is*
- *a 2-tuple of sets, where the first entry*
- *is the tail, and the second entry is the head.*

Raises

- **TypeError** – If *e* is not None or a hashable
- **XGIError** – If *dtype* is not dict or list
- **IDNotFound** – If *e* does not exist in the hypergraph

head(*e=None*, *dtype=<class 'list'>*)

Get the node ids that are in the head of a directed edge.

Parameters

- **e** (*hashable, optional*) – Edge ID. By default, None.
- **dtype** (*{list, dict}, optional*) – Specify the type of the return value. By default, list.

Returns

- *list (if dtype is list, default)* – Head members.
- *dict (if dtype is dict)* – Head members.
- *set (if e is not None)* – Members of the head of edge e.

Raises

- **TypeError** – If *e* is not None or a hashable
- **XGIError** – If *dtype* is not dict or list
- **IDNotFound** – If *e* does not exist in the hypergraph

members(*e=None*, *dtype=<class 'list'>*)

Get the edges of a directed hypergraph.

Parameters

- **e** (*hashable, optional*) – Edge ID. By default, None.
- **dtype** (*{list, dict}, optional*) – Specify the type of the return value. By default, list.

Returns

- *list (if dtype is list, default)* – Edge members.
- *dict (if dtype is dict)* – Edge members.
- *set (if e is not None)* – Members of edge e.
- *The members of an edge are the union of*
- *its head and tail sets.*
- *The*

Raises

- **TypeError** – If *e* is not None or a hashable
- **XGIError** – If *dtype* is not dict or list
- **IDNotFound** – If *e* does not exist in the hypergraph

sources(*e=None*, *dtype=<class 'list'>*)

Get the nodes that are sources (senders) in the directed edges.

See also:

tail
identical method

tail(*e=None*, *dtype=<class 'list'>*)

Get the node ids that are in the tail of a directed edge.

Parameters

- **e** (*hashable, optional*) – Edge ID. By default, None.
- **dtype** (*{list, dict}, optional*) – Specify the type of the return value. By default, list.

Returns

- *list (if dtype is list, default)* – Tail members.
- *dict (if dtype is dict)* – Tail members.
- *set (if e is not None)* – Tail members of edge e.

Raises

- **TypeError** – If *e* is not None or a hashable
- **XGIError** – If *dtype* is not dict or list
- **IDNotFound** – If *e* does not exist in the hypergraph

targets(*e=None*, *dtype=<class 'list'>*)

Get the nodes that are sources (senders) in the directed edges.

See also:

head
identical method

16.6 xgi.core.globalviews

View of Hypergraphs as a subhypergraph or read-only.

In some algorithms it is convenient to temporarily morph a hypergraph to exclude some nodes or edges. It should be better to do that via a view than to remove and then re-add. This module provides those graph views.

The resulting views are essentially read-only graphs that report data from the original graph object.

Note: Since globalviews look like hypergraphs, one can end up with view-of-view-of-view chains. Be careful with chains because they become very slow with about 15 nested views. Often it is easiest to use .copy() to avoid chains.

Functions

xgi.core.globalviews.**subhypergraph**(*H*, *nodes=None*, *edges=None*, *keep_isolates=True*)

View of *H* applying a filter on nodes and edges.

subhypergraph_view provides a read-only view of the induced subhypergraph that includes nodes, edges, or both based on what the user specifies. This function automatically filters out edges that are not subsets of the nodes. This function may create isolated nodes.

If the user only specifies the nodes to include, the function returns an induced subhypergraph on the nodes.

If the user only specifies the edges to include, the function returns all of the nodes and the specified edges.

If the user specifies both nodes and edges to include in the subhypergraph, then the function returns a subhypergraph with the specified nodes and edges from the list of specified hyperedges that are induced by the specified nodes.

Parameters

- **H** (hypergraph.Hypergraph) – A hypergraph
- **nodes** (*list or set, optional*) – A list of the nodes desired for the subhypergraph. If None (default), uses all the nodes.
- **edges** (*list or set, optional*) – A list of the edges desired for the subhypergraph. If None (default), uses all the edges.
- **keep_isolates** (*bool, optional*) – Whether to keep isolated nodes in the subhypergraph. By default, True.

Returns
A read-only hypergraph view of the input hypergraph.

Return type
Hypergraph object

CHAPTER

SEVENTEEN

STATS PACKAGE

Statistics of networks, their nodes, and edges.

Any mapping that assigns some quantity to each node of a network is considered a node statistic. For example, the degree is a node-integer mapping, while a node attribute that assigns a string label to each node is a node-string mapping. The *stats* package provides a common interface to all such mappings.

Each such mapping is accessible via the *H.nodes* view. For example, the degree of all nodes supports type conversion using the *as** methods.

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.nodes.degree.asdict()
{1: 1, 2: 2, 3: 3, 4: 2, 5: 2}
>>> H.nodes.degree.aslist()
[1, 2, 3, 2, 2]
```

Another feature is the ability to filter the nodes of a network by degree.

```
>>> H.nodes.filterby('degree', 2)
NodeView((2, 4, 5))
```

The power of the stats package is that any other node statistic that can be conceived of as a node-quantity mapping is given the same interface. For example, node attributes get the same treatment:

```
>>> H.add_nodes_from([
...     (1, {"color": "red", "name": "horse"}),
...     (2, {"color": "blue", "name": "pony"}),
...     (3, {"color": "yellow", "name": "zebra"}),
...     (4, {"color": "red", "name": "orangutan", "age": 20}),
...     (5, {"color": "blue", "name": "fish", "age": 2}),
... ])
>>> H.nodes.attrs('color').asdict()
{1: 'red', 2: 'blue', 3: 'yellow', 4: 'red', 5: 'blue'}
>>> H.nodes.attrs('color').aslist()
['red', 'blue', 'yellow', 'red', 'blue']
>>> H.nodes.filterby_attr('color', 'red')
NodeView((1, 4))
```

Many other features are available, including edge-statistics, and user-defined statistics. For more details, see the tutorial.

Available statistics

Hypergraphs and simplicial complexes

Statistics of nodes

average_neighbor_degree	Average neighbor degree.
clique_eigenvector_centrality	Compute the clique motif eigenvector centrality of a hypergraph.
clustering_coefficient	Local clustering coefficient.
degree	Node degree.
h_eigenvector_centrality	Compute the H-eigenvector centrality of a hypergraph.
local_clustering_coefficient	Compute the local clustering coefficient.
node_edge_centrality	Computes node centralities.
two_node_clustering_coefficient	Return the clustering coefficients for each node in a Hypergraph.

Statistics of edges

order	Edge order.
size	Edge size.
node_edge_centrality	Computes edge centralities.

Corresponding decorators

nodestat_func	Decorate arbitrary functions to behave like NodeStat objects.
edgestat_func	Decorate arbitrary functions to behave like EdgeStat objects.

17.1 xgi.stats.nodestat_func

xgi.stats.**nodestat_func**(*func*)

Decorate arbitrary functions to behave like NodeStat objects.

Parameters

func (*callable*) – Function or callable with signature *func(net, bunch)*, where *net* is the network and *bunch* is an iterable of nodes in *net*. The call *func(net, bunch)* must return a dict with pairs of the form *(node: value)* where *node* is in *bunch* and *value* is the value of the statistic at *node*.

Returns

The decorated callable unmodified, after registering it in the *stats* framework.

Return type

callable

See also:

edgestat_func()

Notes

The user must make sure that *func* is such that, if *res* is defined as *res = func(net, bunch)*, then *res* has keys in the same order as they are found in *bunch*. Since python dicts preserve order, it is enough for *func* to create the returned dict by iterating over *bunch*.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2], [3, 4], [4, 5, 6]])
```

The following function defines a node-integer mapping.

```
>>> def my_degree(net, bunch):
...     return {n: 10 * net.degree(n) for n in bunch}
```

Node statistics can be called from the network or from the NodeView.

```
>>> H.degree()
{1: 1, 2: 1, 3: 1, 4: 2, 5: 1, 6: 1}
>>> H.nodes.degree
NodeStat('degree')
```

However, *my_degree* is not recognized as a node statistic.

```
>>> H.my_degree()
Traceback (most recent call last):
AttributeError:...
```

```
>>> H.nodes.my_degree
Traceback (most recent call last):
AttributeError:...
```

Use the *nodestat_func* decorator to turn *my_degree* into a valid stat.

```
>>> original_my_degree = my_degree
>>> my_degree = xgi.nodestat_func(my_degree)
>>> H.my_degree()
{1: 10, 2: 10, 3: 10, 4: 20, 5: 10, 6: 10}
>>> H.nodes.my_degree
NodeStat('my_degree')
```

Now the entirety of the interface of stat objects is available.

```
>>> H.nodes.filterby('my_degree', 20)
NodeView((4,))
>>> H.nodes.multi(['degree', 'my_degree']).aspandas()
   degree  my_degree
1       1         10
2       1         10
3       1         10
4       2         20
```

(continues on next page)

(continued from previous page)

```
5       1          10
6       1          10
```

Note the passed function is left unmodified.

```
>>> my_degree is original_my_degree
True
```

The previous usage of *nodestat* is made for explanatory purposes. A more typical use of *nodestat* is the following.

```
>>> @xgi.nodestat_func
... def my_degree(net, bunch):
...     return {n: 10 * net.degree(n) for n in bunch}
```

17.2 xgi.stats.edgestat_func

xgi.stats.**edgestat_func**(*func*)

Decorate arbitrary functions to behave like *EdgeStat* objects.

Works identically to `nodestat()`. For extended documentation, see *nodestat_func()*.

Parameters

func (*callable*) – Function or callable with signature *func(net, bunch)*, where *net* is the network and *bunch* is an iterable of edges in *net*. The call *func(net, bunch)* must return a dict with pairs of the form *(edge: value)* where *edge* is in *bunch* and *value* is the value of the statistic at *edge*.

Returns

The decorated callable unmodified, after registering it in the *stats* framework.

Return type

callable

See also:

nodestat_func()

Corresponding modules

nodestats	Node statistics.
edgestats	Edge statistics.

17.3 xgi.stats.nodestats

Node statistics.

This module is part of the stats package, and it defines node-level statistics. That is, each function defined in this module is assumed to define a node-quantity mapping. Each callable defined here is accessible via a *Network* object, or a `NodeView` object. For more details, see the tutorial.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.degree()
{1: 1, 2: 2, 3: 3, 4: 2, 5: 2}
>>> H.nodes.degree.asdict()
{1: 1, 2: 2, 3: 3, 4: 2, 5: 2}
```

Functions

xgi.stats.nodestats.**attrs**(*net*, *bunch*, *attr=None*, *missing=None*)

Access node attributes.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.
- **attr** (*str | None (default)*) – If None, return all attributes. Otherwise, return a single attribute with name *attr*.
- **missing** (*Any*) – Value to impute in case a node does not have an attribute with name *attr*. Default is None.

Returns

If attr is None, return a nested dict of the form *{node: {"attr": val}}*. Otherwise, return a simple dict of the form *{node: val}*.

Return type

dict

Notes

When requesting all attributes (i.e. when *attr* is None), no value is imputed.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.add_nodes_from([
...         (1, {"color": "red", "name": "horse"}),
...         (2, {"color": "blue", "name": "pony"}),
...         (3, {"color": "yellow", "name": "zebra"}),
...         (4, {"color": "red", "name": "orangutan", "age": 20}),
...         (5, {"color": "blue", "name": "fish", "age": 2}),
...     ])
```

Access all attributes as different types.

```
>>> H.nodes.attrs.asdict()
{1: {'color': 'red', 'name': 'horse'},
 2: {'color': 'blue', 'name': 'pony'},
 3: {'color': 'yellow', 'name': 'zebra'},
 4: {'color': 'red', 'name': 'orangutan', 'age': 20},
 5: {'color': 'blue', 'name': 'fish', 'age': 2}}
>>> H.nodes.attrs.asnumpy()
array([{'color': 'red', 'name': 'horse'},
       {'color': 'blue', 'name': 'pony'},
       {'color': 'yellow', 'name': 'zebra'},
       {'color': 'red', 'name': 'orangutan', 'age': 20},
       {'color': 'blue', 'name': 'fish', 'age': 2}],
      dtype=object)
```

Access a single attribute as different types.

```
>>> H.nodes.attrs('color').asdict()
{1: 'red', 2: 'blue', 3: 'yellow', 4: 'red', 5: 'blue'}
>>> H.nodes.attrs('color').aslist()
['red', 'blue', 'yellow', 'red', 'blue']
```

By default, None is imputed when a node does not have the requested attribute.

```
>>> H.nodes.attrs('age').asdict()
{1: None, 2: None, 3: None, 4: 20, 5: 2}
```

Use *missing* to change the imputed value.

```
>>> H.nodes.attrs('age', missing=100).asdict()
{1: 100, 2: 100, 3: 100, 4: 20, 5: 2}
```

xgi.stats.nodestats.**average_neighbor_degree**(*net*, *bunch*)

Average neighbor degree.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.

Return type
dict

Examples

```
>>> import xgi, numpy as np
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> np.round(H.nodes.average_neighbor_degree.asnumpy(), 3)
array([2.5  , 2.   , 1.75 , 2.333, 2.333])
```

xgi.stats.nodestats.**clique_eigenvector_centrality**(*net*, *bunch*, *tol=1e-06*)

Compute the clique motif eigenvector centrality of a hypergraph.

Parameters

- **net** (*xgi.Hypergraph*) – The hypergraph of interest.

- **bunch** (*Iterable*) – Nodes in *net*.
- **tol** (*float > 0, default: 1e-6*) – The desired L2 error in the centrality vector.

Returns
Centrality, where keys are node IDs and values are centralities.

Return type
dict

References

Three Hypergraph Eigenvector Centralities, Austin R. Benson, https://doi.org/10.1137/18M1203031

xgi.stats.nodestats.**clustering_coefficient**(*net*, *bunch*)

Local clustering coefficient.

This clustering coefficient is defined as the clustering coefficient of the unweighted pairwise projection of the hypergraph, i.e., *num / denom*, where *num* equals *A^3[n, n]* and *denom* equals *nu*(nu-1)/2*. Here *A* is the adjacency matrix of the network and *nu* is the number of pairwise neighbors of *n*.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.

Return type
dict

Notes

This is a direct generalization of the definition of local clustering coefficient for graphs. It has not been tested on hypergraphs.

Examples

```
>>> import xgi, numpy as np
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.nodes.two_node_clustering_coefficient.asnumpy()
array([0.41666667, 0.45833333, 0.58333333, 0.66666667, 0.66666667])
```

xgi.stats.nodestats.**degree**(*net*, *bunch*, *order=None*, *weight=None*)

Node degree.

The degree of a node is the number of edges it belongs to.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.
- **order** (*int | None*) – If not None (default), only count the edges of the given order.
- **weight** (*str | None*) – If not None, specifies the name of the edge attribute that determines the weight of each edge.

Return type
dict

xgi.stats.nodestats.**h_eigenvector_centrality**(*net*, *bunch*, *max_iter=10*, *tol=1e-06*)

Compute the H-eigenvector centrality of a hypergraph.

Parameters

- **net** (*xgi.Hypergraph*) – The hypergraph of interest.
- **bunch** (*Iterable*) – Nodes in *net*.
- **max_iter** (*int, default: 10*) – The maximum number of iterations before the algorithm terminates.
- **tol** (*float > 0, default: 1e-6*) – The desired L2 error in the centrality vector.

Returns
Centrality, where keys are node IDs and values are centralities.

Return type
dict

References

Three Hypergraph Eigenvector Centralities, Austin R. Benson, https://doi.org/10.1137/18M1203031

xgi.stats.nodestats.**local_clustering_coefficient**(*net*, *bunch*)

Compute the local clustering coefficient.

This clustering coefficient is based on the overlap of the edges connected to a given node, normalized by the size of the node's neighborhood.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.

Returns
keys are node IDs and values are the clustering coefficients.

Return type
dict

References

"Properties of metabolic graphs: biological organization or representation artifacts?" by Wanding Zhou and Luay Nakhleh. https://doi.org/10.1186/1471-2105-12-132

"Hypergraphs for predicting essential genes using multiprotein complex data" by Florian Klimm, Charlotte M. Deane, and Gesine Reinert. https://doi.org/10.1093/comnet/cnaa028

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(3, [1, 1])
>>> H.nodes.local_clustering_coefficient.asdict()
{0: 1.0, 1: 1.0, 2: 1.0}
```

xgi.stats.nodestats.**node_edge_centrality**(*net*, *bunch*, *f=<function <lambda>>*, *g=<function <lambda>>*, *phi=<function <lambda>>*, *psi=<function <lambda>>*, *max_iter=100*, *tol=1e-06*)

Computes node centralities.

Parameters

- **net** (Hypergraph) – The hypergraph of interest
- **bunch** (*Iterable*) – Edges in *net*
- **f** (*lambda function, default: x^2*) – The function f as described in Tudisco and Higham. Must accept a numpy array.
- **g** (*lambda function, default: x^0.5*) – The function g as described in Tudisco and Higham. Must accept a numpy array.
- **phi** (*lambda function, default: x^2*) – The function phi as described in Tudisco and Higham. Must accept a numpy array.
- **psi** (*lambda function, default: x^0.5*) – The function psi as described in Tudisco and Higham. Must accept a numpy array.
- **max_iter** (*int, default: 100*) – Number of iterations at which the algorithm terminates if convergence is not reached.
- **tol** (*float > 0, default: 1e-6*) – The total allowable error in the node and edge centralities.

Returns

The node centrality where keys are node IDs and values are associated centralities and the edge centrality where keys are the edge IDs and values are associated centralities.

Return type

dict, dict

Notes

In the paper from which this was taken, it includes general functions for both nodes and edges, nodes and edges may be weighted, and one can choose different norms for normalization, all of which are not yet implemented.

This method does not output the node centralities even though they are computed.

References

Node and edge nonlinear eigenvector centrality for hypergraphs, Francesco Tudisco & Desmond J. Higham, https://doi.org/10.1038/s42005-021-00704-2

xgi.stats.nodestats.**two_node_clustering_coefficient**(*net*, *bunch*, *kind='union'*)

Return the clustering coefficients for each node in a Hypergraph.

This definition averages over all of the two-node clustering coefficients involving the node.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.
- **kind** (*str*) – The type of two-node clustering coefficient. Types are:
  - "union"
  - "min"
  - "max"
- **default** (*by*) –
- **"union".** –

Returns

nodes are keys, clustering coefficients are values.

Return type

dict

References

"Clustering Coefficients in Protein Interaction Hypernetworks" by Suzanne Gallagher and Debra Goldberg. DOI: 10.1145/2506583.2506635

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(3, [1, 1])
>>> H.nodes.two_node_clustering_coefficient.asdict()
{0: 0.5, 1: 0.5, 2: 0.5}
```

17.4 xgi.stats.edgestats

Edge statistics.

This module is part of the stats package, and it defines edge-level statistics. That is, each function defined in this module is assumed to define an edge-quantity mapping. Each callable defined here is accessible via a *Network* object, or a `EdgeView` object. For more details, see the tutorial.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.order()
{0: 2, 1: 3, 2: 2}
>>> H.edges.order.asdict()
{0: 2, 1: 3, 2: 2}
```

Functions

xgi.stats.edgestats.**attrs**(*net*, *bunch*, *attr=None*, *missing=None*)

Access edge attributes.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.
- **attr** (*str | None (default)*) – If None, return all attributes. Otherwise, return a single attribute with name *attr*.
- **missing** (*Any*) – Value to impute in case an edge does not have an attribute with name *attr*. Default is None.

Returns

If attr is None, return a nested dict of the form *{edge: {"attr": val}}*. Otherwise, return a simple dict of the form *{edge: val}*.

Return type

dict

Notes

When requesting all attributes (i.e. when *attr* is None), no value is imputed.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph()
>>> edges = [
...     ([0, 1], 'one', {'color': 'red'}),
...     ([1, 2], 'two', {'color': 'black', 'age': 30}),
...     ([2, 3, 4], 'three', {'color': 'blue', 'age': 40}),
... ]
>>> H.add_edges_from(edges)
```

Access all attributes as different types.

```
>>> H.edges.attrs.asdict()
{'one': {'color': 'red'},
 'two': {'color': 'black', 'age': 30},
```

(continues on next page)

(continued from previous page)

```
 'three': {'color': 'blue', 'age': 40}}
>>> H.edges.attrs.asnumpy()
array([{'color': 'red'},
       {'color': 'black', 'age': 30},
       {'color': 'blue', 'age': 40}],
       dtype=object)
```

Access a single attribute as different types.

```
>>> H.edges.attrs('color').asdict()
{'one': 'red', 'two': 'black', 'three': 'blue'}
>>> H.edges.attrs('color').aslist()
['red', 'black', 'blue']
```

By default, None is imputed when a node does not have the requested attribute.

```
>>> H.edges.attrs('age').asdict()
{'one': None, 'two': 30, 'three': 40}
```

Use *missing* to change the imputed value.

```
>>> H.edges.attrs('age', missing=100).asdict()
{'one': 100, 'two': 30, 'three': 40}
```

xgi.stats.edgestats.**order**(*net*, *bunch*, *degree=None*)

Edge order.

The order of an edge is the number of nodes it contains minus 1.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Edges in *net*.
- **degree** (*int | None*) – If not None (default), count only those member nodes with the specified degree.

Return type

dict

See also:

size

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.edges.order.asdict()
{0: 2, 1: 3, 2: 2}
>>> H.edges.order(degree=2).asdict()
{0: 0, 1: 2, 2: 1}
```

xgi.stats.edgestats.**size**(*net*, *bunch*, *degree=None*)

Edge size.

The size of an edge is the number of nodes it contains.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Edges in *net*.

Return type
dict

See also:

order

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.edges.size.asdict()
{0: 3, 1: 4, 2: 3}
```

xgi.stats.edgestats.**node_edge_centrality**(*net*, *bunch*, *f=<function <lambda>>*, *g=<function <lambda>>*, *phi=<function <lambda>>*, *psi=<function <lambda>>*, *max_iter=100*, *tol=1e-06*)

Computes edge centralities.

Parameters

- **net** (Hypergraph) – The hypergraph of interest
- **bunch** (*Iterable*) – Edges in *net*
- **f** (*lambda function, default: x^2*) – The function f as described in Tudisco and Higham. Must accept a numpy array.
- **g** (*lambda function, default: x^0.5*) – The function g as described in Tudisco and Higham. Must accept a numpy array.
- **phi** (*lambda function, default: x^2*) – The function phi as described in Tudisco and Higham. Must accept a numpy array.
- **psi** (*lambda function, default: x^0.5*) – The function psi as described in Tudisco and Higham. Must accept a numpy array.
- **max_iter** (*int, default: 100*) – Number of iterations at which the algorithm terminates if convergence is not reached.
- **tol** (*float > 0, default: 1e-6*) – The total allowable error in the node and edge centralities.

Returns
The edge centrality where keys are the edge IDs and values are associated centralities.

Return type
dict, dict

Notes

In the paper from which this was taken, it is more general in that it includes general functions for both nodes and edges, nodes and edges may be weighted, and one can choose different norms for normalization.

This method does not output the node centralities even though they are computed.

References

Node and edge nonlinear eigenvector centrality for hypergraphs, Francesco Tudisco & Desmond J. Higham, https://doi.org/10.1038/s42005-021-00704-2

Corresponding classes

NodeStat	An arbitrary node-quantity mapping.
EdgeStat	An arbitrary edge-quantity mapping.
MultiNodeStat	Multiple node-quantity mappings.
MultiEdgeStat	Multiple edge-quantity mappings.

17.5 xgi.stats.NodeStat

class xgi.stats.**NodeStat**(*network*, *view*, *func*, *args=None*, *kwargs=None*)

Bases: IDStat

An arbitrary node-quantity mapping.

NodeStat objects represent a mapping that assigns a value to each node in a network. For more details, see the tutorial.

Attributes

name	Name of this stat.

Methods

asdict	Output the stat as a dict.
aslist	Output the stat as a list.
asnumpy	Output the stat as a numpy array.
aspandas	Output the stat as a pandas series.
ashist	Return the distribution of a numpy array.
max	The maximum value of this stat.
mean	The arithmetic mean of this stat.
median	The median of this stat.
min	The minimum value of this stat.
std	The standard deviation of this stat.
var	The variance of this stat.
moment	The statistical moments of this stat.
argmin	The ID corresponding to the minimum of the stat
argmax	The ID corresponding to the maximum of the stat
argsort	Get the list of IDs sorted by stat value.

argmax()

The ID corresponding to the maximum of the stat

When the maximal value is not unique, returns first ID corresponding to the maximal value.

Returns
The ID to which the maximum value corresponds.

Return type
hashable

argmin()

The ID corresponding to the minimum of the stat

When the minimum value is not unique, returns first ID corresponding to the minimum value.

Returns
The ID to which the minimum value corresponds.

Return type
hashable

argsort(*reverse=False*)

Get the list of IDs sorted by stat value.

When values are not unique, the order of the IDs is preserved.

Parameters
reverse (*bool*) – Whether the sorting should be ascending or descending.

Returns
The IDs sorted in ascending or descending order.

Return type
list

asdict()

Output the stat as a dict.

Notes

All stats are stored as dicts and therefore this method incurs in no overhead as type conversion is not necessary.

ashist(*bins=10*, *bin_edges=False*, *density=False*, *log_binning=False*)

Return the distribution of a numpy array.

Parameters

- **vals** (*Numpy array*) – The array of values
- **bins** (*int, list, or Numpy array*) – The number of bins or the bin edges.
- **bin_edges** (*bool*) – Whether to also output the min and max of each bin, by default, False.
- **density** (*bool*) – Whether to normalize the resulting distribution.
- **log_binning** (*bool*) – Whether to bin the values with log-sized bins. By default, False.

Returns

A two-column table with "bin_center" and "value" columns, where "value" is a count or a probability. If *bin_edges* is True, outputs two additional columns, *bin_lo* and *bin_hi*, which outputs the left and right bin edges respectively.

Return type

Pandas DataFrame

Notes

Originally from https://github.com/jkbren/networks-and-dataviz

aslist()

Output the stat as a list.

asnumpy()

Output the stat as a numpy array.

aspandas()

Output the stat as a pandas series.

Notes

The *name* attribute of the returned series is set using the *name* property.

max()

The maximum value of this stat.

mean()

The arithmetic mean of this stat.

median()

The median of this stat.

min()

The minimum value of this stat.

moment(*order=2*, *center=False*)

The statistical moments of this stat.

Parameters

- **order** (*int (default 2)*) – The order of the moment.
- **center** (*bool (default False)*) – Whether to compute the centered (False) or uncentered/raw (True) moment.

property name

Name of this stat.

The name of a stat is used to populate the keys of dictionaries in *MultiStat* objects, as well as the names of columns of pandas dataframes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> da, d3 = H.nodes.degree, H.nodes.degree(order=3)
>>> da.name, d3.name
('degree', 'degree(order=3)')
>>> H.nodes.multi([da, d3]).asdict(transpose=True).keys()
dict_keys(['degree', 'degree(order=3)'])
>>> H.nodes.multi([da, d3]).aspandas().columns
Index(['degree', 'degree(order=3)'], dtype='object')
```

std()

The standard deviation of this stat.

sum()

The sum of this stat.

var()

The variance of this stat.

17.6 xgi.stats.EdgeStat

class xgi.stats.**EdgeStat**(*network*, *view*, *func*, *args=None*, *kwargs=None*)

Bases: IDStat

An arbitrary edge-quantity mapping.

EdgeStat objects represent a mapping that assigns a value to each edge in a network. For more details, see the tutorial.

Attributes

name	Name of this stat.

Methods

asdict	Output the stat as a dict.
aslist	Output the stat as a list.
asnumpy	Output the stat as a numpy array.
aspandas	Output the stat as a pandas series.
ashist	Return the distribution of a numpy array.
max	The maximum value of this stat.
mean	The arithmetic mean of this stat.
median	The median of this stat.
min	The minimum value of this stat.
std	The standard deviation of this stat.
var	The variance of this stat.
moment	The statistical moments of this stat.
argmin	The ID corresponding to the minimum of the stat
argmax	The ID corresponding to the maximum of the stat
argsort	Get the list of IDs sorted by stat value.

argmax()

The ID corresponding to the maximum of the stat

When the maximal value is not unique, returns first ID corresponding to the maximal value.

Returns
The ID to which the maximum value corresponds.

Return type
hashable

argmin()

The ID corresponding to the minimum of the stat

When the minimum value is not unique, returns first ID corresponding to the minimum value.

Returns
The ID to which the minimum value corresponds.

Return type
hashable

argsort(*reverse=False*)

Get the list of IDs sorted by stat value.

When values are not unique, the order of the IDs is preserved.

Parameters
reverse (*bool*) – Whether the sorting should be ascending or descending.

Returns
The IDs sorted in ascending or descending order.

Return type
list

asdict()

Output the stat as a dict.

Notes

All stats are stored as dicts and therefore this method incurs in no overhead as type conversion is not necessary.

ashist(*bins=10*, *bin_edges=False*, *density=False*, *log_binning=False*)

Return the distribution of a numpy array.

Parameters

- **vals** (*Numpy array*) – The array of values
- **bins** (*int, list, or Numpy array*) – The number of bins or the bin edges.
- **bin_edges** (*bool*) – Whether to also output the min and max of each bin, by default, False.
- **density** (*bool*) – Whether to normalize the resulting distribution.
- **log_binning** (*bool*) – Whether to bin the values with log-sized bins. By default, False.

Returns
A two-column table with "bin_center" and "value" columns, where "value" is a count or a probability. If *bin_edges* is True, outputs two additional columns, *bin_lo* and *bin_hi*, which outputs the left and right bin edges respectively.

Return type
Pandas DataFrame

Notes

Originally from https://github.com/jkbren/networks-and-dataviz

aslist()

Output the stat as a list.

asnumpy()

Output the stat as a numpy array.

aspandas()

Output the stat as a pandas series.

Notes

The *name* attribute of the returned series is set using the *name* property.

max()

The maximum value of this stat.

mean()

The arithmetic mean of this stat.

median()

The median of this stat.

min()

The minimum value of this stat.

moment(*order=2*, *center=False*)

The statistical moments of this stat.

Parameters

- **order** (*int (default 2)*) – The order of the moment.
- **center** (*bool (default False)*) – Whether to compute the centered (False) or uncentered/raw (True) moment.

property name

Name of this stat.

The name of a stat is used to populate the keys of dictionaries in *MultiStat* objects, as well as the names of columns of pandas dataframes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> da, d3 = H.nodes.degree, H.nodes.degree(order=3)
>>> da.name, d3.name
('degree', 'degree(order=3)')
>>> H.nodes.multi([da, d3]).asdict(transpose=True).keys()
dict_keys(['degree', 'degree(order=3)'])
>>> H.nodes.multi([da, d3]).aspandas().columns
Index(['degree', 'degree(order=3)'], dtype='object')
```

std()

The standard deviation of this stat.

sum()

The sum of this stat.

var()

The variance of this stat.

17.7 xgi.stats.MultiNodeStat

class xgi.stats.**MultiNodeStat**(*network*, *view*, *stats*)

Bases: MultiIDStat

Multiple node-quantity mappings.

For more details, see the tutorial.

Attributes

name	Name of this stat.

Methods

asdict	Output the stats as a dict of collections.
aslist	Output the stats as a list of collections.
asnumpy	Output the stats as a numpy array.
aspandas	Output the stats as a pandas dataframe.
ashist	Return the distributions of a numpy array.

argmax()

The ID corresponding to the maximum of the stat

When the maximal value is not unique, returns first ID corresponding to the maximal value.

Returns
The ID to which the maximum value corresponds.

Return type
hashable

argmin()

The ID corresponding to the minimum of the stat

When the minimum value is not unique, returns first ID corresponding to the minimum value.

Returns
The ID to which the minimum value corresponds.

Return type
hashable

argsort(*reverse=False*)

Get the list of IDs sorted by stat value.

When values are not unique, the order of the IDs is preserved.

Parameters
reverse (*bool*) – Whether the sorting should be ascending or descending.

Returns
The IDs sorted in ascending or descending order.

Return type
list

asdict(*inner=<class 'dict'>*, *transpose=False*)

Output the stats as a dict of collections.

Parameters

- **inner** (*dict (default) or list*) – The type of the inner collections. If dict (default), output a dict of dicts. If list, output a dict of lists.

- **transpose** (*bool (default False)*) – By default, output a dict of dicts whose outer keys are the nodes and inner keys are the specified stats. If True, the outer and inner keys are reversed. Only used when *inner* is *dict*.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> m = H.nodes.multi(['degree', 'clustering_coefficient'])
>>> m.asdict()
{1: {'degree': 1, 'clustering_coefficient': 1.0},
 2: {'degree': 2, 'clustering_coefficient': 0.6666666666666666},
 3: {'degree': 3, 'clustering_coefficient': 0.6666666666666666},
 4: {'degree': 2, 'clustering_coefficient': 1.0},
 5: {'degree': 2, 'clustering_coefficient': 1.0}}
>>> m.asdict(transpose=True)
{'degree': {1: 1, 2: 2, 3: 3, 4: 2, 5: 2},
'clustering_coefficient': {1: 1.0,
2: 0.6666666666666666,
3: 0.6666666666666666,
4: 1.0,
5: 1.0}}
```

ashist(*bins=10*, *bin_edges=False*, *density=False*, *log_binning=False*)

Return the distributions of a numpy array.

Parameters

- **vals** (*Numpy array*) – The array of values
- **bins** (*int, list, or Numpy array*) – The number of bins or the bin edges.
- **bin_edges** (*bool*) – Whether to also output the min and max of each bin, by default, False.
- **density** (*bool*) – Whether to normalize the resulting distribution.
- **log_binning** (*bool*) – Whether to bin the values with log-sized bins. By default, False.

Returns

Each entry of the list is a two-column table with "bin_center" and "value" columns, where "value" is a count or a probability. If *bin_edges* is True, outputs two additional columns, *bin_lo* and *bin_hi*, which outputs the left and right bin edges respectively.

Return type

list of Pandas DataFrames

Notes

Originally from https://github.com/jkbren/networks-and-dataviz

aslist(*inner=<class 'list'>*, *transpose=False*)

Output the stats as a list of collections.

Parameters

- **inner** (*list (default) or dict*) – The type of the inner collections. If list (default), output a list of lists. If dict, output a list of dicts.

- **transpose** (*bool (default False)*) – By default, output a list of lists where each inner list contains the stats of a single node. If True, each inner list contains the values of a single stat of all nodes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> m = H.nodes.multi(['degree', 'clustering_coefficient'])
>>> m.aslist() # doctest:
[[1, 1.0], [2, 0.6666666666666666], [3, 0.6666666666666666], [2, 1.0], [2, 1.0]]
>>> m.aslist(transpose=True)
[[1, 2, 3, 2, 2], [1.0, 0.6666666666666666, 0.6666666666666666, 1.0, 1.0]]
```

asnumpy()

Output the stats as a numpy array.

Notes

Equivalent to *np.array(self.aslist(inner=list))*.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.nodes.multi(['degree', 'clustering_coefficient']).asnumpy()
...
array([[1.        , 1.        ],
       [2.        , 0.66666667],
       [3.        , 0.66666667],
       [2.        , 1.        ],
       [2.        , 1.        ]])
```

aspandas()

Output the stats as a pandas dataframe.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.nodes.multi(['degree', 'clustering_coefficient']).aspandas()
...
   degree  clustering_coefficient
1       1                1.000000
2       2                0.666667
3       3                0.666667
4       2                1.000000
5       2                1.000000
```

max()

The maximum value of this stat.

mean()

The arithmetic mean of this stat.

median()

The median of this stat.

min()

The minimum value of this stat.

moment(*order=2*, *center=False*)

The statistical moments of this stat.

Parameters

- **order** (*int (default 2)*) – The order of the moment.
- **center** (*bool (default False)*) – Whether to compute the centered (False) or uncentered/raw (True) moment.

property name

Name of this stat.

The name of a stat is used to populate the keys of dictionaries in *MultiStat* objects, as well as the names of columns of pandas dataframes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> da, d3 = H.nodes.degree, H.nodes.degree(order=3)
>>> da.name, d3.name
('degree', 'degree(order=3)')
>>> H.nodes.multi([da, d3]).asdict(transpose=True).keys()
dict_keys(['degree', 'degree(order=3)'])
>>> H.nodes.multi([da, d3]).aspandas().columns
Index(['degree', 'degree(order=3)'], dtype='object')
```

statsclass

alias of *NodeStat*

statsmodule = <module 'xgi.stats.nodestats' from '/home/docs/checkouts/readthedocs.org/user_builds/xgi/envs/stable/lib/python3.12/site-packages/xgi/stats/nodestats.py'>

Module in which to search for mappings.

std()

The standard deviation of this stat.

sum()

The sum of this stat.

var()

The variance of this stat.

17.8 xgi.stats.MultiEdgeStat

class xgi.stats.**MultiEdgeStat**(*network*, *view*, *stats*)

Bases: MultiIDStat

Multiple edge-quantity mappings.

For more details, see the tutorial.

Attributes

name	Name of this stat.

Methods

asdict	Output the stats as a dict of collections.
aslist	Output the stats as a list of collections.
asnumpy	Output the stats as a numpy array.
aspandas	Output the stats as a pandas dataframe.
ashist	Return the distributions of a numpy array.

argmax()

The ID corresponding to the maximum of the stat

When the maximal value is not unique, returns first ID corresponding to the maximal value.

Returns
The ID to which the maximum value corresponds.

Return type
hashable

argmin()

The ID corresponding to the minimum of the stat

When the minimum value is not unique, returns first ID corresponding to the minimum value.

Returns
The ID to which the minimum value corresponds.

Return type
hashable

argsort(*reverse=False*)

Get the list of IDs sorted by stat value.

When values are not unique, the order of the IDs is preserved.

Parameters
reverse (*bool*) – Whether the sorting should be ascending or descending.

Returns
The IDs sorted in ascending or descending order.

Return type
list

asdict(*inner=<class 'dict'>*, *transpose=False*)

Output the stats as a dict of collections.

Parameters

- **inner** (*dict (default) or list*) – The type of the inner collections. If dict (default), output a dict of dicts. If list, output a dict of lists.
- **transpose** (*bool (default False)*) – By default, output a dict of dicts whose outer keys are the nodes and inner keys are the specified stats. If True, the outer and inner keys are reversed. Only used when *inner* is *dict*.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> m = H.nodes.multi(['degree', 'clustering_coefficient'])
>>> m.asdict()
{1: {'degree': 1, 'clustering_coefficient': 1.0},
 2: {'degree': 2, 'clustering_coefficient': 0.6666666666666666},
 3: {'degree': 3, 'clustering_coefficient': 0.6666666666666666},
 4: {'degree': 2, 'clustering_coefficient': 1.0},
 5: {'degree': 2, 'clustering_coefficient': 1.0}}
>>> m.asdict(transpose=True)
{'degree': {1: 1, 2: 2, 3: 3, 4: 2, 5: 2},
'clustering_coefficient': {1: 1.0,
2: 0.6666666666666666,
3: 0.6666666666666666,
4: 1.0,
5: 1.0}}
```

ashist(*bins=10*, *bin_edges=False*, *density=False*, *log_binning=False*)

Return the distributions of a numpy array.

Parameters

- **vals** (*Numpy array*) – The array of values
- **bins** (*int, list, or Numpy array*) – The number of bins or the bin edges.
- **bin_edges** (*bool*) – Whether to also output the min and max of each bin, by default, False.
- **density** (*bool*) – Whether to normalize the resulting distribution.
- **log_binning** (*bool*) – Whether to bin the values with log-sized bins. By default, False.

Returns
Each entry of the list is a two-column table with "bin_center" and "value" columns, where "value" is a count or a probability. If *bin_edges* is True, outputs two additional columns, *bin_lo* and *bin_hi*, which outputs the left and right bin edges respectively.

Return type
list of Pandas DataFrames

Notes

Originally from https://github.com/jkbren/networks-and-dataviz

aslist(*inner=<class 'list'>*, *transpose=False*)

Output the stats as a list of collections.

Parameters

- **inner** (*list (default) or dict*) – The type of the inner collections. If list (default), output a list of lists. If dict, output a list of dicts.
- **transpose** (*bool (default False)*) – By default, output a list of lists where each inner list contains the stats of a single node. If True, each inner list contains the values of a single stat of all nodes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> m = H.nodes.multi(['degree', 'clustering_coefficient'])
>>> m.aslist() # doctest:
[[1, 1.0], [2, 0.6666666666666666], [3, 0.6666666666666666], [2, 1.0], [2, 1.0]]
>>> m.aslist(transpose=True)
[[1, 2, 3, 2, 2], [1.0, 0.6666666666666666, 0.6666666666666666, 1.0, 1.0]]
```

asnumpy()

Output the stats as a numpy array.

Notes

Equivalent to *np.array(self.aslist(inner=list))*.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.nodes.multi(['degree', 'clustering_coefficient']).asnumpy()
...
array([[1.        , 1.        ],
       [2.        , 0.66666667],
       [3.        , 0.66666667],
       [2.        , 1.        ],
       [2.        , 1.        ]])
```

aspandas()

Output the stats as a pandas dataframe.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.nodes.multi(['degree', 'clustering_coefficient']).aspandas()
...
   degree  clustering_coefficient
1       1                1.000000
2       2                0.666667
3       3                0.666667
4       2                1.000000
5       2                1.000000
```

max()

The maximum value of this stat.

mean()

The arithmetic mean of this stat.

median()

The median of this stat.

min()

The minimum value of this stat.

moment(*order=2*, *center=False*)

The statistical moments of this stat.

Parameters

- **order** (*int (default 2)*) – The order of the moment.
- **center** (*bool (default False)*) – Whether to compute the centered (False) or uncentered/raw (True) moment.

property name

Name of this stat.

The name of a stat is used to populate the keys of dictionaries in *MultiStat* objects, as well as the names of columns of pandas dataframes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> da, d3 = H.nodes.degree, H.nodes.degree(order=3)
>>> da.name, d3.name
('degree', 'degree(order=3)')
>>> H.nodes.multi([da, d3]).asdict(transpose=True).keys()
dict_keys(['degree', 'degree(order=3)'])
>>> H.nodes.multi([da, d3]).aspandas().columns
Index(['degree', 'degree(order=3)'], dtype='object')
```

statsclass

alias of *EdgeStat*

statsmodule = <module 'xgi.stats.edgestats' from '/home/docs/checkouts/readthedocs.org/user_builds/xgi/envs/stable/lib/python3.12/site-packages/xgi/stats/edgestats.py'>

Module in which to search for mappings.

std()

The standard deviation of this stat.

sum()

The sum of this stat.

var()

The variance of this stat.

Directed hypergraphs

Statistics of nodes

degree	Node degree.
in_degree	Node in-degree.
out_degree	Node out-degree.

Statistics of edges

order	Edge order.
size	Edge size.
head_order	Head order.
head_size	Head size.
tail_order	Tail order.
tail_size	Tail size.

Corresponding decorators

dinodestat_func	Decorator that allows arbitrary functions to behave like *DiNodeStat* objects.
diedgestat_func	Decorator that allows arbitrary functions to behave like *DiEdgeStat* objects.

17.9 xgi.stats.dinodestat_func

xgi.stats.**dinodestat_func**(*func*)

Decorator that allows arbitrary functions to behave like *DiNodeStat* objects.

Works identically to `nodestat()`. For extended documentation, see *nodestat_func()*.

Parameters

func (*callable*) – Function or callable with signature *func(net, bunch)*, where *net* is the network and *bunch* is an iterable of edges in *net*. The call *func(net, bunch)* must return a dict with pairs of the form *(edge: value)* where *edge* is in *bunch* and *value* is the value of the statistic at *edge*.

Returns
The decorated callable unmodified, after registering it in the *stats* framework.

Return type
callable

See also:

nodestat_func(), *edgestat_func()*, *diedgestat_func()*

17.10 xgi.stats.diedgestat_func

xgi.stats.**diedgestat_func**(*func*)

Decorator that allows arbitrary functions to behave like *DiEdgeStat* objects.

Works identically to `nodestat()`. For extended documentation, see *nodestat_func()*.

Parameters
func (*callable*) – Function or callable with signature *func(net, bunch)*, where *net* is the network and *bunch* is an iterable of edges in *net*. The call *func(net, bunch)* must return a dict with pairs of the form *(edge: value)* where *edge* is in *bunch* and *value* is the value of the statistic at *edge*.

Returns
The decorated callable unmodified, after registering it in the *stats* framework.

Return type
callable

See also:

nodestat_func(), *dinodestat_func()*, *diedgestat_func()*

Corresponding modules

dinodestats	Node statistics.
diedgestats	Directed edge statistics.

17.11 xgi.stats.dinodestats

Node statistics.

This module is part of the stats package, and it defines node-level statistics. That is, each function defined in this module is assumed to define a node-quantity mapping. Each callable defined here is accessible via a *Network* object, or a `NodeView` object. For more details, see the tutorial.

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])
>>> H.degree()
{1: 2, 2: 1, 5: 1, 6: 1, 4: 1, 3: 1}
>>> H.nodes.degree.asdict()
{1: 2, 2: 1, 5: 1, 6: 1, 4: 1, 3: 1}
```

Functions

xgi.stats.dinodestats.**attrs**(*net*, *bunch*, *attr=None*, *missing=None*)

Access node attributes.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.
- **attr** (*str | None (default)*) – If None, return all attributes. Otherwise, return a single attribute with name *attr*.
- **missing** (*Any*) – Value to impute in case a node does not have an attribute with name *attr*. Default is None.

Returns

If attr is None, return a nested dict of the form *{node: {"attr": val}}*. Otherwise, return a simple dict of the form *{node: val}*.

Return type

dict

Notes

When requesting all attributes (i.e. when *attr* is None), no value is imputed.

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph()
>>> H.add_nodes_from([
...         (1, {"color": "red", "name": "horse"}),
...         (2, {"color": "blue", "name": "pony"}),
...         (3, {"color": "yellow", "name": "zebra"}),
...         (4, {"color": "red", "name": "orangutan", "age": 20}),
...         (5, {"color": "blue", "name": "fish", "age": 2}),
...     ])
```

Access all attributes as different types.

```
>>> H.nodes.attrs.asdict()
{1: {'color': 'red', 'name': 'horse'},
 2: {'color': 'blue', 'name': 'pony'},
 3: {'color': 'yellow', 'name': 'zebra'},
 4: {'color': 'red', 'name': 'orangutan', 'age': 20},
 5: {'color': 'blue', 'name': 'fish', 'age': 2}}
>>> H.nodes.attrs.asnumpy()
array([{'color': 'red', 'name': 'horse'},
       {'color': 'blue', 'name': 'pony'},
       {'color': 'yellow', 'name': 'zebra'},
       {'color': 'red', 'name': 'orangutan', 'age': 20},
       {'color': 'blue', 'name': 'fish', 'age': 2}],
      dtype=object)
```

Access a single attribute as different types.

```
>>> H.nodes.attrs('color').asdict()
{1: 'red', 2: 'blue', 3: 'yellow', 4: 'red', 5: 'blue'}
>>> H.nodes.attrs('color').aslist()
['red', 'blue', 'yellow', 'red', 'blue']
```

By default, None is imputed when a node does not have the requested attribute.

```
>>> H.nodes.attrs('age').asdict()
{1: None, 2: None, 3: None, 4: 20, 5: 2}
```

Use *missing* to change the imputed value.

```
>>> H.nodes.attrs('age', missing=100).asdict()
{1: 100, 2: 100, 3: 100, 4: 20, 5: 2}
```

xgi.stats.dinodestats.**degree**(*net*, *bunch*, *order=None*, *weight=None*)

Node degree.

The degree of a node is the number of edges it belongs to, regardless of whether it is in the head or the tail.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.
- **order** (*int | None*) – If not None (default), only count the edges of the given order.
- **weight** (*str | None*) – If not None, specifies the name of the edge attribute that determines the weight of each edge.

Return type

dict

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])
>>> H.nodes.degree.asdict()
{1: 2, 2: 1, 5: 1, 6: 1, 4: 1, 3: 1}
```

xgi.stats.dinodestats.**in_degree**(*net*, *bunch*, *order=None*, *weight=None*)

Node in-degree.

The in-degree of a node is the number of edges for which the node is in the head (it is a receiver).

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.
- **order** (*int | None*) – If not None (default), only count the edges of the given order.
- **weight** (*str | None*) – If not None, specifies the name of the edge attribute that determines the weight of each edge.

Return type
dict

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])
>>> H.nodes.in_degree.asdict()
{1: 1, 2: 1, 5: 0, 6: 0, 4: 1, 3: 0}
```

xgi.stats.dinodestats.**out_degree**(*net*, *bunch*, *order=None*, *weight=None*)

Node out-degree.

The out-degree of a node is the number of edges for which the node is in the tail (it is a sender).

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.
- **order** (*int | None*) – If not None (default), only count the edges of the given order.
- **weight** (*str | None*) – If not None, specifies the name of the edge attribute that determines the weight of each edge.

Return type
dict

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])
>>> H.nodes.out_degree.asdict()
{1: 1, 2: 0, 5: 1, 6: 1, 4: 0, 3: 1}
```

17.12 xgi.stats.diedgestats

Directed edge statistics.

This module is part of the stats package, and it defines edge-level statistics. That is, each function defined in this module is assumed to define an edge-quantity mapping. Each callable defined here is accessible via a *Network* object, or a `DiEdgeView` object. For more details, see the tutorial.

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])
>>> H.order()
{0: 3, 1: 2}
>>> H.edges.order.asdict()
{0: 3, 1: 2}
```

Functions

`xgi.stats.diedgestats.`**`attrs`**(*net*, *bunch*, *attr=None*, *missing=None*)

Access edge attributes.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Nodes in *net*.
- **attr** (*str | None (default)*) – If None, return all attributes. Otherwise, return a single attribute with name *attr*.
- **missing** (*Any*) – Value to impute in case an edge does not have an attribute with name *attr*. Default is None.

Returns

If attr is None, return a nested dict of the form *{edge: {"attr": val}}*. Otherwise, return a simple dict of the form *{edge: val}*.

Return type

dict

Notes

When requesting all attributes (i.e. when *attr* is None), no value is imputed.

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph()
>>> edges = [
...     ([{0, 1}, {2, 4}], 'one', {'color': 'red'}),
...     ([{1, 2}, {2, 0}], 'two', {'color': 'black', 'age': 30}),
...     ([{2, 3, 4}, {1}], 'three', {'color': 'blue', 'age': 40}),
... ]
>>> H.add_edges_from(edges)
```

Access all attributes as different types.

```
>>> H.edges.attrs.asdict()
{'one': {'color': 'red'},
 'two': {'color': 'black', 'age': 30},
 'three': {'color': 'blue', 'age': 40}}
>>> H.edges.attrs.asnumpy()
array([{'color': 'red'},
       {'color': 'black', 'age': 30},
       {'color': 'blue', 'age': 40}],
       dtype=object)
```

Access a single attribute as different types.

```
>>> H.edges.attrs('color').asdict()
{'one': 'red', 'two': 'black', 'three': 'blue'}
>>> H.edges.attrs('color').aslist()
['red', 'black', 'blue']
```

By default, None is imputed when a node does not have the requested attribute.

```
>>> H.edges.attrs('age').asdict()
{'one': None, 'two': 30, 'three': 40}
```

Use *missing* to change the imputed value.

```
>>> H.edges.attrs('age', missing=100).asdict()
{'one': 100, 'two': 30, 'three': 40}
```

xgi.stats.diedgestats.**order**(*net*, *bunch*, *degree=None*)

Edge order.

The order of a directed edge is the number of nodes contained in the union of the head and the tail minus 1.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Edges in *net*.

- **degree** (*int | None*) – If not None (default), count only those member nodes with the specified degree.

Return type
dict

See also:

size

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])
>>> H.edges.order.asdict()
{0: 3, 1: 2}
```

xgi.stats.diedgestats.**size**(*net*, *bunch*, *degree=None*)

Edge size.

The size of a directed edge is the number of nodes contained in the union of the head and the tail.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Edges in *net*.

Return type
dict

See also:

order

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])
>>> H.edges.size.asdict()
{0: 4, 1: 3}
```

xgi.stats.diedgestats.**head_order**(*net*, *bunch*, *degree=None*)

Head order.

The order of the head is the number of nodes it contains minus 1.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Edges in *net*.

Return type
dict

See also:

order

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])
>>> H.edges.head_order.asdict()
{0: 1, 1: 1}
```

xgi.stats.diedgestats.**head_size**(*net*, *bunch*, *degree=None*)

Head size.

The size of the head is the number of nodes it contains.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Edges in *net*.

Return type
dict

See also:

order

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])
>>> H.edges.head_size.asdict()
{0: 2, 1: 2}
```

xgi.stats.diedgestats.**tail_order**(*net*, *bunch*, *degree=None*)

Tail order.

The order of the tail is the number of nodes it contains minus 1.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Edges in *net*.

Return type
dict

See also:

order

Examples

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])
>>> H.edges.tail_order.asdict()
{0: 1, 1: 0}
```

xgi.stats.diedgestats.**tail_size**(*net*, *bunch*, *degree=None*)

Tail size.

The size of the tail is the number of nodes it contains.

Parameters

- **net** (*xgi.Hypergraph*) – The network.
- **bunch** (*Iterable*) – Edges in *net*.

Return type
dict

See also:

order

Examples

```
>>> import xgi
>>> H = xgi.DiHypergraph([[{1, 2}, {5, 6}], [{4}, {1, 3}]])
>>> H.edges.tail_size.asdict()
{0: 2, 1: 1}
```

Corresponding classes

DiNodeStat	An arbitrary node-quantity mapping.
DiEdgeStat	An arbitrary edge-quantity mapping.
MultiDiNodeStat	Multiple node-quantity mappings.
MultiDiEdgeStat	Multiple edge-quantity mappings.

17.13 xgi.stats.DiNodeStat

class xgi.stats.**DiNodeStat**(*network*, *view*, *func*, *args=None*, *kwargs=None*)

Bases: IDStat

An arbitrary node-quantity mapping.

NodeStat objects represent a mapping that assigns a value to each node in a network. For more details, see the tutorial.

Attributes

name	Name of this stat.

Methods

asdict	Output the stat as a dict.
aslist	Output the stat as a list.
asnumpy	Output the stat as a numpy array.
aspandas	Output the stat as a pandas series.
ashist	Return the distribution of a numpy array.
max	The maximum value of this stat.
mean	The arithmetic mean of this stat.
median	The median of this stat.
min	The minimum value of this stat.
std	The standard deviation of this stat.
var	The variance of this stat.
moment	The statistical moments of this stat.
argmin	The ID corresponding to the minimum of the stat
argmax	The ID corresponding to the maximum of the stat
argsort	Get the list of IDs sorted by stat value.

17.14 xgi.stats.DiEdgeStat

class xgi.stats.**DiEdgeStat**(*network*, *view*, *func*, *args=None*, *kwargs=None*)

Bases: IDStat

An arbitrary edge-quantity mapping.

EdgeStat objects represent a mapping that assigns a value to each edge in a network. For more details, see the tutorial.

Attributes

name	Name of this stat.

Methods

asdict	Output the stat as a dict.
aslist	Output the stat as a list.
asnumpy	Output the stat as a numpy array.
aspandas	Output the stat as a pandas series.
ashist	Return the distribution of a numpy array.
max	The maximum value of this stat.
mean	The arithmetic mean of this stat.
median	The median of this stat.
min	The minimum value of this stat.
std	The standard deviation of this stat.
var	The variance of this stat.
moment	The statistical moments of this stat.
argmin	The ID corresponding to the minimum of the stat
argmax	The ID corresponding to the maximum of the stat
argsort	Get the list of IDs sorted by stat value.

argmax()

The ID corresponding to the maximum of the stat

When the maximal value is not unique, returns first ID corresponding to the maximal value.

Returns
The ID to which the maximum value corresponds.

Return type
hashable

argmin()

The ID corresponding to the minimum of the stat

When the minimum value is not unique, returns first ID corresponding to the minimum value.

Returns
The ID to which the minimum value corresponds.

Return type
hashable

argsort(*reverse=False*)

Get the list of IDs sorted by stat value.

When values are not unique, the order of the IDs is preserved.

Parameters
reverse (*bool*) – Whether the sorting should be ascending or descending.

Returns
The IDs sorted in ascending or descending order.

Return type
list

asdict()

Output the stat as a dict.

Notes

All stats are stored as dicts and therefore this method incurs in no overhead as type conversion is not necessary.

ashist(*bins=10*, *bin_edges=False*, *density=False*, *log_binning=False*)

Return the distribution of a numpy array.

Parameters

- **vals** (*Numpy array*) – The array of values
- **bins** (*int, list, or Numpy array*) – The number of bins or the bin edges.
- **bin_edges** (*bool*) – Whether to also output the min and max of each bin, by default, False.
- **density** (*bool*) – Whether to normalize the resulting distribution.
- **log_binning** (*bool*) – Whether to bin the values with log-sized bins. By default, False.

Returns

A two-column table with "bin_center" and "value" columns, where "value" is a count or a probability. If *bin_edges* is True, outputs two additional columns, *bin_lo* and *bin_hi*, which outputs the left and right bin edges respectively.

Return type

Pandas DataFrame

Notes

Originally from https://github.com/jkbren/networks-and-dataviz

aslist()

Output the stat as a list.

asnumpy()

Output the stat as a numpy array.

aspandas()

Output the stat as a pandas series.

Notes

The *name* attribute of the returned series is set using the *name* property.

max()

The maximum value of this stat.

mean()

The arithmetic mean of this stat.

median()

The median of this stat.

min()

The minimum value of this stat.

moment(*order=2*, *center=False*)

The statistical moments of this stat.

Parameters

- **order** (*int (default 2)*) – The order of the moment.
- **center** (*bool (default False)*) – Whether to compute the centered (False) or uncentered/raw (True) moment.

property name

Name of this stat.

The name of a stat is used to populate the keys of dictionaries in *MultiStat* objects, as well as the names of columns of pandas dataframes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> da, d3 = H.nodes.degree, H.nodes.degree(order=3)
>>> da.name, d3.name
('degree', 'degree(order=3)')
>>> H.nodes.multi([da, d3]).asdict(transpose=True).keys()
dict_keys(['degree', 'degree(order=3)'])
>>> H.nodes.multi([da, d3]).aspandas().columns
Index(['degree', 'degree(order=3)'], dtype='object')
```

std()

The standard deviation of this stat.

sum()

The sum of this stat.

var()

The variance of this stat.

17.15 xgi.stats.MultiDiNodeStat

class xgi.stats.**MultiDiNodeStat**(*network*, *view*, *stats*)

Bases: MultiIDStat

Multiple node-quantity mappings.

For more details, see the tutorial.

Attributes

name	Name of this stat.

Methods

asdict	Output the stats as a dict of collections.
aslist	Output the stats as a list of collections.
asnumpy	Output the stats as a numpy array.
aspandas	Output the stats as a pandas dataframe.
ashist	Return the distributions of a numpy array.

argmax()

The ID corresponding to the maximum of the stat

When the maximal value is not unique, returns first ID corresponding to the maximal value.

Returns
The ID to which the maximum value corresponds.

Return type
hashable

argmin()

The ID corresponding to the minimum of the stat

When the minimum value is not unique, returns first ID corresponding to the minimum value.

Returns
The ID to which the minimum value corresponds.

Return type
hashable

argsort(*reverse=False*)

Get the list of IDs sorted by stat value.

When values are not unique, the order of the IDs is preserved.

Parameters
reverse (*bool*) – Whether the sorting should be ascending or descending.

Returns
The IDs sorted in ascending or descending order.

Return type
list

asdict(*inner=<class 'dict'>*, *transpose=False*)

Output the stats as a dict of collections.

Parameters

- **inner** (*dict (default) or list*) – The type of the inner collections. If dict (default), output a dict of dicts. If list, output a dict of lists.

- **transpose** (*bool (default False)*) – By default, output a dict of dicts whose outer keys are the nodes and inner keys are the specified stats. If True, the outer and inner keys are reversed. Only used when *inner* is *dict*.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> m = H.nodes.multi(['degree', 'clustering_coefficient'])
>>> m.asdict()
{1: {'degree': 1, 'clustering_coefficient': 1.0},
 2: {'degree': 2, 'clustering_coefficient': 0.6666666666666666},
 3: {'degree': 3, 'clustering_coefficient': 0.6666666666666666},
 4: {'degree': 2, 'clustering_coefficient': 1.0},
 5: {'degree': 2, 'clustering_coefficient': 1.0}}
>>> m.asdict(transpose=True)
{'degree': {1: 1, 2: 2, 3: 3, 4: 2, 5: 2},
'clustering_coefficient': {1: 1.0,
2: 0.6666666666666666,
3: 0.6666666666666666,
4: 1.0,
5: 1.0}}
```

ashist(*bins=10*, *bin_edges=False*, *density=False*, *log_binning=False*)

Return the distributions of a numpy array.

Parameters

- **vals** (*Numpy array*) – The array of values
- **bins** (*int, list, or Numpy array*) – The number of bins or the bin edges.
- **bin_edges** (*bool*) – Whether to also output the min and max of each bin, by default, False.
- **density** (*bool*) – Whether to normalize the resulting distribution.
- **log_binning** (*bool*) – Whether to bin the values with log-sized bins. By default, False.

Returns

Each entry of the list is a two-column table with "bin_center" and "value" columns, where "value" is a count or a probability. If *bin_edges* is True, outputs two additional columns, *bin_lo* and *bin_hi*, which outputs the left and right bin edges respectively.

Return type

list of Pandas DataFrames

Notes

Originally from https://github.com/jkbren/networks-and-dataviz

aslist(*inner=<class 'list'>*, *transpose=False*)

Output the stats as a list of collections.

Parameters

- **inner** (*list (default) or dict*) – The type of the inner collections. If list (default), output a list of lists. If dict, output a list of dicts.

- **transpose** (*bool (default False)*) – By default, output a list of lists where each inner list contains the stats of a single node. If True, each inner list contains the values of a single stat of all nodes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> m = H.nodes.multi(['degree', 'clustering_coefficient'])
>>> m.aslist() # doctest:
[[1, 1.0], [2, 0.6666666666666666], [3, 0.6666666666666666], [2, 1.0], [2, 1.0]]
>>> m.aslist(transpose=True)
[[1, 2, 3, 2, 2], [1.0, 0.6666666666666666, 0.6666666666666666, 1.0, 1.0]]
```

asnumpy()

Output the stats as a numpy array.

Notes

Equivalent to *np.array(self.aslist(inner=list))*.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.nodes.multi(['degree', 'clustering_coefficient']).asnumpy()
...
array([[1.        , 1.        ],
       [2.        , 0.66666667],
       [3.        , 0.66666667],
       [2.        , 1.        ],
       [2.        , 1.        ]])
```

aspandas()

Output the stats as a pandas dataframe.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.nodes.multi(['degree', 'clustering_coefficient']).aspandas()
...
   degree  clustering_coefficient
1       1                1.000000
2       2                0.666667
3       3                0.666667
4       2                1.000000
5       2                1.000000
```

max()

The maximum value of this stat.

mean()

The arithmetic mean of this stat.

median()

The median of this stat.

min()

The minimum value of this stat.

moment(*order=2*, *center=False*)

The statistical moments of this stat.

Parameters

- **order** (*int (default 2)*) – The order of the moment.
- **center** (*bool (default False)*) – Whether to compute the centered (False) or uncentered/raw (True) moment.

property name

Name of this stat.

The name of a stat is used to populate the keys of dictionaries in *MultiStat* objects, as well as the names of columns of pandas dataframes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> da, d3 = H.nodes.degree, H.nodes.degree(order=3)
>>> da.name, d3.name
('degree', 'degree(order=3)')
>>> H.nodes.multi([da, d3]).asdict(transpose=True).keys()
dict_keys(['degree', 'degree(order=3)'])
>>> H.nodes.multi([da, d3]).aspandas().columns
Index(['degree', 'degree(order=3)'], dtype='object')
```

statsclass

alias of *DiNodeStat*

statsmodule = <module 'xgi.stats.dinodestats' from '/home/docs/checkouts/readthedocs.org/user_builds/xgi/envs/stable/lib/python3.12/site-packages/xgi/stats/dinodestats.py'>

Module in which to search for mappings.

std()

The standard deviation of this stat.

sum()

The sum of this stat.

var()

The variance of this stat.

17.16 xgi.stats.MultiDiEdgeStat

class xgi.stats.**MultiDiEdgeStat**(*network*, *view*, *stats*)

Bases: MultiIDStat

Multiple edge-quantity mappings.

For more details, see the tutorial.

Attributes

name	Name of this stat.

Methods

asdict	Output the stats as a dict of collections.
aslist	Output the stats as a list of collections.
asnumpy	Output the stats as a numpy array.
aspandas	Output the stats as a pandas dataframe.
ashist	Return the distributions of a numpy array.

argmax()

The ID corresponding to the maximum of the stat

When the maximal value is not unique, returns first ID corresponding to the maximal value.

Returns
The ID to which the maximum value corresponds.

Return type
hashable

argmin()

The ID corresponding to the minimum of the stat

When the minimum value is not unique, returns first ID corresponding to the minimum value.

Returns
The ID to which the minimum value corresponds.

Return type
hashable

argsort(*reverse=False*)

Get the list of IDs sorted by stat value.

When values are not unique, the order of the IDs is preserved.

Parameters
reverse (*bool*) – Whether the sorting should be ascending or descending.

Returns
The IDs sorted in ascending or descending order.

Return type
list

asdict(*inner=<class 'dict'>*, *transpose=False*)

Output the stats as a dict of collections.

Parameters

- **inner** (*dict (default) or list*) – The type of the inner collections. If dict (default), output a dict of dicts. If list, output a dict of lists.
- **transpose** (*bool (default False)*) – By default, output a dict of dicts whose outer keys are the nodes and inner keys are the specified stats. If True, the outer and inner keys are reversed. Only used when *inner* is *dict*.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> m = H.nodes.multi(['degree', 'clustering_coefficient'])
>>> m.asdict()
{1: {'degree': 1, 'clustering_coefficient': 1.0},
 2: {'degree': 2, 'clustering_coefficient': 0.6666666666666666},
 3: {'degree': 3, 'clustering_coefficient': 0.6666666666666666},
 4: {'degree': 2, 'clustering_coefficient': 1.0},
 5: {'degree': 2, 'clustering_coefficient': 1.0}}
>>> m.asdict(transpose=True)
{'degree': {1: 1, 2: 2, 3: 3, 4: 2, 5: 2},
'clustering_coefficient': {1: 1.0,
2: 0.6666666666666666,
3: 0.6666666666666666,
4: 1.0,
5: 1.0}}
```

ashist(*bins=10*, *bin_edges=False*, *density=False*, *log_binning=False*)

Return the distributions of a numpy array.

Parameters

- **vals** (*Numpy array*) – The array of values
- **bins** (*int, list, or Numpy array*) – The number of bins or the bin edges.
- **bin_edges** (*bool*) – Whether to also output the min and max of each bin, by default, False.
- **density** (*bool*) – Whether to normalize the resulting distribution.
- **log_binning** (*bool*) – Whether to bin the values with log-sized bins. By default, False.

Returns
Each entry of the list is a two-column table with "bin_center" and "value" columns, where "value" is a count or a probability. If *bin_edges* is True, outputs two additional columns, *bin_lo* and *bin_hi*, which outputs the left and right bin edges respectively.

Return type
list of Pandas DataFrames

Notes

Originally from https://github.com/jkbren/networks-and-dataviz

aslist(*inner=<class 'list'>*, *transpose=False*)

Output the stats as a list of collections.

Parameters

- **inner** (*list (default) or dict*) – The type of the inner collections. If list (default), output a list of lists. If dict, output a list of dicts.
- **transpose** (*bool (default False)*) – By default, output a list of lists where each inner list contains the stats of a single node. If True, each inner list contains the values of a single stat of all nodes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> m = H.nodes.multi(['degree', 'clustering_coefficient'])
>>> m.aslist() # doctest:
[[1, 1.0], [2, 0.6666666666666666], [3, 0.6666666666666666], [2, 1.0], [2, 1.0]]
>>> m.aslist(transpose=True)
[[1, 2, 3, 2, 2], [1.0, 0.6666666666666666, 0.6666666666666666, 1.0, 1.0]]
```

asnumpy()

Output the stats as a numpy array.

Notes

Equivalent to *np.array(self.aslist(inner=list))*.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.nodes.multi(['degree', 'clustering_coefficient']).asnumpy()
...
array([[1.        , 1.        ],
       [2.        , 0.66666667],
       [3.        , 0.66666667],
       [2.        , 1.        ],
       [2.        , 1.        ]])
```

aspandas()

Output the stats as a pandas dataframe.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> H.nodes.multi(['degree', 'clustering_coefficient']).aspandas()
...
   degree  clustering_coefficient
1       1                1.000000
2       2                0.666667
3       3                0.666667
4       2                1.000000
5       2                1.000000
```

max()

The maximum value of this stat.

mean()

The arithmetic mean of this stat.

median()

The median of this stat.

min()

The minimum value of this stat.

moment(*order=2*, *center=False*)

The statistical moments of this stat.

Parameters

- **order** (*int (default 2)*) – The order of the moment.
- **center** (*bool (default False)*) – Whether to compute the centered (False) or uncentered/raw (True) moment.

property name

Name of this stat.

The name of a stat is used to populate the keys of dictionaries in *MultiStat* objects, as well as the names of columns of pandas dataframes.

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [2, 3, 4, 5], [3, 4, 5]])
>>> da, d3 = H.nodes.degree, H.nodes.degree(order=3)
>>> da.name, d3.name
('degree', 'degree(order=3)')
>>> H.nodes.multi([da, d3]).asdict(transpose=True).keys()
dict_keys(['degree', 'degree(order=3)'])
>>> H.nodes.multi([da, d3]).aspandas().columns
Index(['degree', 'degree(order=3)'], dtype='object')
```

statsclass

alias of *DiEdgeStat*

statsmodule = <module 'xgi.stats.diedgestats' from '/home/docs/checkouts/readthedocs.org/user_builds/xgi/envs/stable/lib/python3.12/site-packages/xgi/stats/diedgestats.py'>

Module in which to search for mappings.

std()

The standard deviation of this stat.

sum()

The sum of this stat.

var()

The variance of this stat.

CHAPTER

EIGHTEEN

ALGORITHMS PACKAGE

Modules

assortativity	Algorithms for finding the degree assortativity of a hypergraph.
centrality	Algorithms for computing the centralities of nodes (and edges) in a hypergraph.
clustering	Algorithms for computing nodal clustering coefficients.
connected	Algorithms related to connected components of a hypergraph.
shortest_path	Algorithms for computing shortest paths in a hypergraph.
properties	Functional interface to hypergraph methods and assorted utilities.

18.1 xgi.algorithms.assortativity

Algorithms for finding the degree assortativity of a hypergraph.

Functions

xgi.algorithms.assortativity.**dynamical_assortativity**(*H*)

Computes the dynamical assortativity of a uniform hypergraph.

Parameters
H (*xgi.Hypergraph*) – Hypergraph of interest

Returns
The dynamical assortativity

Return type
float

See also:

degree_assortativity

Raises
XGIError – If the hypergraph is not uniform, or if there are no nodes or no edges

References

Nicholas Landry and Juan G. Restrepo, Hypergraph assortativity: A dynamical systems perspective, Chaos 2022. DOI: 10.1063/5.0086905

xgi.algorithms.assortativity.**degree_assortativity**(*H*, *kind='uniform'*, *exact=False*, *num_samples=1000*)

Computes the degree assortativity of a hypergraph

Parameters

- **H** (Hypergraph) – The hypergraph of interest
- **kind** (*str, optional*) – the type of degree assortativity. valid choices are "uniform", "top-2", and "top-bottom". By default, "uniform".
- **exact** (*bool, optional*) – whether to compute over all edges or sample randomly from the set of edges. By default, False.
- **num_samples** (*int, optional*) – if not exact, specify the number of samples for the computation. By default, 1000.

Returns
the degree assortativity

Return type
float

Raises
XGIError – If there are no nodes or no edges

See also:

dynamical_assortativity

References

Phil Chodrow, Configuration models of random hypergraphs, Journal of Complex Networks 2020. DOI: 10.1093/comnet/cnaa018

18.2 xgi.algorithms.centrality

Algorithms for computing the centralities of nodes (and edges) in a hypergraph.

Functions

xgi.algorithms.centrality.**clique_eigenvector_centrality**(*H*, *tol=1e-06*)

Compute the clique motif eigenvector centrality of a hypergraph.

Parameters

- **H** (Hypergraph) – The hypergraph of interest.
- **tol** (*float, optional*) – The tolerance when computing the eigenvector. By default, 1e-6.

Returns
Centrality, where keys are node IDs and values are centralities. The centralities are 1-normalized.

Return type
dict

See also:

h_eigenvector_centrality

References

Three Hypergraph Eigenvector Centralities, Austin R. Benson, https://doi.org/10.1137/18M1203031

xgi.algorithms.centrality.**h_eigenvector_centrality**(*H*, *max_iter=100*, *tol=1e-06*)

Compute the H-eigenvector centrality of a uniform hypergraph.

Parameters

- **H** (Hypergraph) – The hypergraph of interest.
- **max_iter** (*int, optional*) – The maximum number of iterations before the algorithm terminates. By default, 100.
- **tol** (*float > 0, optional*) – The desired L2 error in the centrality vector. By default, 1e-6.

Returns
Centrality, where keys are node IDs and values are centralities. The centralities are 1-normalized.

Return type
dict

Raises
XGIError – If the hypergraph is not uniform.

See also:

clique_eigenvector_centrality

References

Three Hypergraph Eigenvector Centralities, Austin R. Benson, https://doi.org/10.1137/18M1203031

xgi.algorithms.centrality.**node_edge_centrality**(*H*, *f=<function <lambda>>*, *g=<function <lambda>>*, *phi=<function <lambda>>*, *psi=<function <lambda>>*, *max_iter=100*, *tol=1e-06*)

Computes the node and edge centralities

Parameters

- **H** (Hypergraph) – The hypergraph of interest
- **f** (*lambda function, optional*) – The function f as described in Tudisco and Higham. Must accept a numpy array. By default, $f(x) = x^2$.
- **g** (*lambda function, optional*) – The function g as described in Tudisco and Higham. Must accept a numpy array. By default, :math`g(x) = sqrt{x}`.
- **phi** (*lambda function, optional*) – The function phi as described in Tudisco and Higham. Must accept a numpy array. By default $\phi(x) = x^2$.
- **psi** (*lambda function, optional*) – The function psi as described in Tudisco and Higham. Must accept a numpy array. By default: $\psi(x) = \sqrt{x}$.

- **max_iter** (*int, optional*) – Number of iterations at which the algorithm terminates if convergence is not reached. By default, 100.
- **tol** (*float > 0, optional*) – The total allowable error in the node and edge centralities. By default, 1e-6.

Returns
The node centrality where keys are node IDs and values are associated centralities and the edge centrality where keys are the edge IDs and values are associated centralities. The centralities of both the nodes and edges are 1-normalized.

Return type
dict, dict

Notes

In the paper from which this was taken, it is more general in that it includes general functions for both nodes and edges, nodes and edges may be weighted, and one can choose different norms for normalization.

References

Node and edge nonlinear eigenvector centrality for hypergraphs, Francesco Tudisco & Desmond J. Higham, https://doi.org/10.1038/s42005-021-00704-2

xgi.algorithms.centrality.**line_vector_centrality**(*H*)

The vector centrality of nodes in the line graph of the hypergraph.

Parameters
H (Hypergraph) – The hypergraph of interest

Returns
Centrality, where keys are node IDs and values are lists of centralities.

Return type
dict

References

"Vector centrality in hypergraphs", K. Kovalenko, M. Romance, E. Vasilyeva, D. Aleja, R. Criado, D. Musatov, A.M. Raigorodskii, J. Flores, I. Samoylenko, K. Alfaro-Bittner, M. Perc, S. Boccaletti, https://doi.org/10.1016/j.chaos.2022.112397

xgi.algorithms.centrality.**katz_centrality**(*H*, *cutoff=100*)

Returns the Katz-centrality vector of a non-empty hypergraph H.

The Katz-centrality measures the relative importance of a node by counting how many distinct walks start from it. The longer the walk is the smaller its contribution will be (attenuation factor α). Initially defined for graphs, the Katz-centrality is here generalized to hypergraphs using the most basic definition of neighbors: two nodes that share an hyperedge.

Parameters

- **H** (*xgi.Hypergraph*) – Hypergraph on which to compute the Katz-centralities.
- **cutoff** (*int*) – Power at which to stop the series $A + \alpha A^2 + \alpha^2 A^3 + \dots$ Default value is 100.

Returns
c – c is a dictionary with node IDs as keys and centrality values as values. The centralities are 1-normalized.

Return type
dict

Raises
XGIError – If the hypergraph is empty.

Notes

[1] The Katz-centrality is defined as

$$c = [(I - \alpha A^t)^{-1} - I]\mathbf{1},$$

where A is the adjacency matrix of the the (hyper)graph. Since $A^t = A$ for undirected graphs (our case), we have:

$$\begin{aligned}
&[I + A + \alpha A^2 + \alpha^2 A^3 + \dots](I - \alpha A^t) \\
&= [I + A + \alpha A^2 + \alpha^2 A^3 + \dots](I - \alpha A) \\
&= (I + A + \alpha A^2 + \alpha^2 A^3 + \dots) - A - \alpha A^2 \\
&- \alpha^2 A^3 - \alpha^3 A^4 - \dots \\
&= I
\end{aligned}$$

And $(I - \alpha A^t)^{-1} = I + A + \alpha A^2 + \alpha^2 A^3 + \dots$ Thus we can use the power series to compute the Katz-centrality.
[2] The Katz-centrality of isolated nodes (no hyperedges contains them) is zero. The Katz-centrality of an empty hypergraph is not defined.

References

See https://en.wikipedia.org/wiki/Katz_centrality#Alpha_centrality (visited May 20 2023) for a clear definition of Katz centrality.

18.3 xgi.algorithms.clustering

Algorithms for computing nodal clustering coefficients.

Functions

xgi.algorithms.clustering.**clustering_coefficient**(*H*)
Return the clustering coefficients for each node in a Hypergraph.

This clustering coefficient is defined as the clustering coefficient of the unweighted pairwise projection of the hypergraph, i.e., $c = A^3_{i,i}/\binom{k}{2}$, where A is the adjacency matrix of the network and k is the pairwise degree of i.

Parameters
H (Hypergraph) – Hypergraph

Returns
nodes are keys, clustering coefficients are values.

Return type
dict

Notes

The clustering coefficient is undefined when the number of neighbors is 0 or 1, but we set the clustering coefficient to 0 in these cases. For more discussion, see https://arxiv.org/abs/0802.2512

See also:

local_clustering_coefficient, *two_node_clustering_coefficient*

References

"Clustering Coefficients in Protein Interaction Hypernetworks" by Suzanne Gallagher and Debra Goldberg. DOI: 10.1145/2506583.2506635

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(3, [1, 1])
>>> cc = xgi.clustering_coefficient(H)
>>> cc
{0: 1.0, 1: 1.0, 2: 1.0}
```

xgi.algorithms.clustering.**local_clustering_coefficient**(*H*)

Compute the local clustering coefficient.

This clustering coefficient is based on the overlap of the edges connected to a given node, normalized by the size of the node's neighborhood.

Parameters
H (Hypergraph) – Hypergraph

Returns
keys are node IDs and values are the clustering coefficients.

Return type
dict

Notes

The clustering coefficient is undefined when the number of neighbors is 0 or 1, but we set the clustering coefficient to 0 in these cases. For more discussion, see https://arxiv.org/abs/0802.2512

See also:

clustering_coefficient, *two_node_clustering_coefficient*

References

"Properties of metabolic graphs: biological organization or representation artifacts?" by Wanding Zhou and Luay Nakhleh. https://doi.org/10.1186/1471-2105-12-132

"Hypergraphs for predicting essential genes using multiprotein complex data" by Florian Klimm, Charlotte M. Deane, and Gesine Reinert. https://doi.org/10.1093/comnet/cnaa028

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(3, [1, 1])
>>> cc = xgi.local_clustering_coefficient(H)
>>> cc
{0: 1.0, 1: 1.0, 2: 1.0}
```

xgi.algorithms.clustering.**two_node_clustering_coefficient**(*H*, *kind='union'*)

Return the clustering coefficients for each node in a Hypergraph.

This definition averages over all of the two-node clustering coefficients involving the node.

Parameters

- **H** (Hypergraph) – Hypergraph
- **kind** (*string, optional*) – The type of two node clustering coefficient. Options are "union", "max", and "min". By default, "union".

Returns
nodes are keys, clustering coefficients are values.

Return type
dict

Notes

The clustering coefficient is undefined when the number of neighbors is 0 or 1, but we set the clustering coefficient to 0 in these cases. For more discussion, see https://arxiv.org/abs/0802.2512

See also:

clustering_coefficient, *local_clustering_coefficient*

References

"Clustering Coefficients in Protein Interaction Hypernetworks" by Suzanne Gallagher and Debra Goldberg. DOI: 10.1145/2506583.2506635

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(3, [1, 1])
>>> cc = xgi.two_node_clustering_coefficient(H, kind="union")
>>> cc
{0: 0.5, 1: 0.5, 2: 0.5}
```

18.4 xgi.algorithms.connected

Algorithms related to connected components of a hypergraph.

Functions

xgi.algorithms.connected.**connected_components**(*H*)

A function to find the connected components of a hypergraph.

Parameters
H (*Hypergraph object*) – The hypergraph of interest

Returns
An iterator where each entry is a component of the hypergraph.

Return type
iterable of sets

See also:

is_connected, *number_connected_components*, *largest_connected_component*, *largest_connected_hypergraph*

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(50, [0.1, 0.01], seed=1)
>>> print([len(component) for component in xgi.connected_components(H)])
[50]
```

xgi.algorithms.connected.**is_connected**(*H*)

A function to determine whether a hypergraph is connected.

Parameters
H (*Hypergraph object*) – The hypergraph of interest

Returns
Whether the hypergraph is connected.

Return type
bool

See also:

connected_components, *number_connected_components*, *largest_connected_component*, *largest_connected_hypergraph*

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(10, [0.5, 0.01], seed=1)
>>> print(xgi.is_connected(H))
True
```

xgi.algorithms.connected.**largest_connected_component**(*H*)

A function to find the largest connected component of a hypergraph.

Parameters
H (*Hypergraph object*) – The hypergraph of interest

Returns
The largest connected component (a set of nodes) of the hypergraph.

Return type
set

See also:

connected_components, *largest_connected_hypergraph*

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(50, [0.1, 0.01], seed=1)
>>> print(len(xgi.largest_connected_component(H)))
50
```

xgi.algorithms.connected.**largest_connected_hypergraph**(*H*, *in_place=False*)

A function to find the largest connected hypergraph from a data set.

Parameters

- **H** (Hypergraph) – The hypergraph of interest
- **in_place** (*bool, optional*) – If False, creates a copy; if True, modifies the existing hypergraph. By default, True.

Returns

- *None* – If in_place: modifies the existing hypergraph
- *Hypergraph* – If not in_place: the hypergraph induced on the nodes of the largest connected component.

See also:

connected_components, *largest_connected_component*

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(10, [0.1, 0.01], seed=1)
>>> H_gcc = xgi.largest_connected_hypergraph(H)
>>> print(H_gcc.num_nodes)
8
```

xgi.algorithms.connected.**node_connected_component**(*H*, *n*)

A function to find the connected component of which a node in the hypergraph is a part.

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **n** (*hashable*) – Node label

See also:

connected_components

Returns

Returns the connected component of which the specified node in the hypergraph is a part.

Return type

set

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(50, [0.1, 0.01], seed=1)
>>> comp = xgi.node_connected_component(H, 0)
>>> print(type(comp), len(comp))
<class 'set'> 50
```

xgi.algorithms.connected.**number_connected_components**(*H*)

A function to find the number of connected components of a hypergraph.

Parameters

H (*Hypergraph object*) – The hypergraph of interest

Returns

The number of connected components of the hypergraph.

Return type

int

See also:

is_connected, *connected_components*, *largest_connected_component*, *largest_connected_hypergraph*

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(50, [0.1, 0.01], seed=1)
>>> print(xgi.number_connected_components(H))
1
```

18.5 xgi.algorithms.shortest_path

Algorithms for computing shortest paths in a hypergraph.

Functions

xgi.algorithms.shortest_path.**single_source_shortest_path_length**(*H*, *source*)

Returns the distances from source to every other node in hypergraph H.

Parameters

- **H** (*xgi.Hypergraph*) – Hypergraph on which to compute the distances. Node indexes must be integers.
- **source** (*int*) – Index of the node from which to compute the distance to every other node.

Returns
dists – Dictionary where keys are node indexes and values are the distances from source.

Return type
dict

xgi.algorithms.shortest_path.**shortest_path_length**(*H*)

Returns a generator of tuples (source, dists) where dists is a dictonary containing the distances from source to every other node in hypergraph H, for all possible source in H.

Parameters
H (*xgi.Hypergraph*) – Hypergraph on which to compute the distances. Node indexes must be integers.

Returns
paths – Every tuple is of the form (source, dict_of_lengths), for every possible source.

Return type
generator of tuples

18.6 xgi.algorithms.properties

Functional interface to hypergraph methods and assorted utilities.

Functions

xgi.algorithms.properties.**degree_counts**(*H*, *order=None*)

Returns a list of the the number of occurrences of each degree value.

The counts correspond to degrees from 0 to max(degree).

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **order** (*int, optional*) – Order of edges to take into account. If None (default), consider all edges.

Returns

A list of frequencies of degrees. The degree values are the index in the list.

Return type

list

Notes

Note: the bins are width one, hence len(list) can be large (Order(num_edges))

The degree is defined as the number of edges to which a node belongs. A node belonging only to a singleton edge will thus have degree 1 and contribute accordingly to the degree count.

Examples

```
>>> import xgi
>>> hyperedge_list = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(hyperedge_list)
>>> xgi.degree_counts(H)
[0, 3, 1]
```

xgi.algorithms.properties.**degree_histogram**(*H*)

Returns a degree histogram including bin centers (degree values).

Parameters

H (*Hypergraph object*) – The hypergraph of interest

Returns

First entry is observed degrees (bin centers),
second entry is degree count (histogram height)

Return type

tuple of lists

Notes

Note: the bins are width one, hence there will be an entry for every observed degree.

Examples

```
>>> import xgi
>>> hyperedge_list = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(hyperedge_list)
>>> xgi.degree_histogram(H)
([1, 2], [3, 1])
```

xgi.algorithms.properties.**density**(*H*, *order=None*, *max_order=None*, *ignore_singletons=False*)

Hypergraph density.

The density of a hypergraph is the number of existing edges divided by the number of possible edges.

Let H have n nodes and m hyperedges. Then,

- *density(H)* = $\frac{m}{2^n-1}$,
- *density(H, ignore_singletons=True)* = $\frac{m}{2^n-1-n}$.

Here, 2^n is the total possible number of hyperedges on H, from which we subtract 1 because the empty hyperedge is not considered. We subtract an additional n when singletons are not considered.

Now assume H has a edges with order 1 and b edges with order 2. Then,

- *density(H, order=1)* = $\frac{a}{\binom{n}{2}}$,
- *density(H, order=2)* = $\frac{b}{\binom{n}{3}}$,
- *density(H, max_order=1)* = $\frac{a}{\binom{n}{1}+\binom{n}{2}}$,
- *density(H, max_order=1, ignore_singletons=True)* = $\frac{a}{\binom{n}{2}}$,
- *density(H, max_order=2)* = $\frac{m}{\binom{n}{1}+\binom{n}{2}+\binom{n}{3}}$,
- *density(H, max_order=2, ignore_singletons=True)* = $\frac{m}{\binom{n}{2}+\binom{n}{3}}$,

Parameters

- **order** (*int, optional*) – If not None, only count edges of the specified order. By default, None.
- **max_order** (*int, optional*) – If not None, only count edges of order up to this value, inclusive. By default, None.
- **ignore_singletons** (*bool, optional*) – Whether to consider singleton edges. Ignored if *order* is not None and different from 0. By default, False.

See also:

incidence_density()

Notes

If both *order* and *max_order* are not None, *max_order* is ignored.

xgi.algorithms.properties.**edge_neighborhood**(*H*, *n*, *include_self=False*)

The edge neighborhood of the specified node.

The edge neighborhood of a node *n* in a hypergraph *H* is an edgelist of all the edges containing *n* and its edges are all the edges in *H* that contain *n*. Usually, the edge neighborhood does not include *n* itself. This can be controlled with *include_self*.

Parameters

- **H** (Hypergraph) – THe hypergraph of interest
- **n** (*node*) – Node whose edge_neighborhood is needed.
- **include_self** (*bool, optional*) – Whether the edge_neighborhood contains *n*. By default, False.

Returns
An edgelist of the edge_neighborhood of *n*.

Return type
list

See also:

neighbors

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph([[1, 2, 3], [3, 4], [4, 5, 6]])
>>> H.nodes.neighbors(3)
{1, 2, 4}
>>> xgi.edge_neighborhood(H, 3)
[{1, 2}, {4}]
>>> xgi.edge_neighborhood(H, 3, include_self=True)
[{1, 2, 3}, {3, 4}]
```

xgi.algorithms.properties.**incidence_density**(*H*, *order=None*, *max_order=None*, *ignore_singletons=False*)

Density of the incidence matrix.

The incidence matrix of a hypergraph contains one row per node and one column per edge. An entry is non-zero when the corresponding node is a member of the corresponding edge. The density of this matrix is the number of non-zero entries divided by the total number of entries.

Parameters

- **order** (*int, optional*) – If not None, only count edges of the specified order. By default, None.
- **max_order** (*int, optional*) – If not None, only count edges of order up to this value, inclusive. By default, None.
- **ignore_singletons** (*bool, optional*) – Whether to consider singleton edges. Ignored if *order* is not None and different from 0. By default, False.

See also:

density()

Notes

If both *order* and *max_order* are not None, *max_order* is ignored.

The parameters *order*, *max_order* and *ignore_singletons* have a similar effect on the denominator as they have in *density()*.

xgi.algorithms.properties.**is_possible_order**(*H*, *d*)

Whether the specified order is between 0 (singletons) and the maximum order.

Parameters

- **H** (Hypergraph) – The hypergraph of interest.
- **d** (*int*) – Order for which to check.

Returns
Whether *d* is a possible order.

Return type
bool

xgi.algorithms.properties.**is_uniform**(*H*)

Order of uniformity if the hypergraph is uniform, or False.

A hypergraph is uniform if all its edges have the same order.

Returns d if the hypergraph is d-uniform, that is if all edges in the hypergraph (excluding singletons) have the same degree d. Returns False if not uniform.

Returns
d – If the hypergraph is d-uniform, return d, or False otherwise.

Return type
int or False

Examples

This function can be used as a boolean check:

```
>>> import xgi
>>> H = xgi.Hypergraph([(0, 1, 2), (1, 2, 3), (2, 3, 4)])
>>> xgi.is_uniform(H)
2
>>> if xgi.is_uniform(H): print('H is uniform!')
H is uniform!
```

xgi.algorithms.properties.**max_edge_order**(*H*)

The maximum order of edges in the hypergraph.

Parameters
H (Hypergraph) – The hypergraph of interest.

Returns
Maximum order of edges in hypergraph.

Return type
int

See also:

num_edges_order

xgi.algorithms.properties.**num_edges_order**(*H*, *d=None*)

The number of edges of order d.

Parameters

- **H** (Hypergraph) – The hypergraph of interest.
- **d** (*int, optional*) – The order of edges to count. If None (default), counts for all orders.

Returns
The number of edges of order d

Return type
int

See also:

max_edge_order

xgi.algorithms.properties.**unique_edge_sizes**(*H*)

A function that returns the unique edge sizes.

Parameters
H (*Hypergraph object*) – The hypergraph of interest

Returns
The unique edge sizes in ascending order by size.

Return type
list()

CHAPTER

NINETEEN

GENERATORS PACKAGE

Modules

classic	Generators for some classic hypergraphs.
simple	Generators for some simple hypergraphs.
lattice	Generators for some lattice hypergraphs.
random	Generate random (non-uniform) hypergraphs.
uniform	Generate random uniform hypergraphs.
simplicial_complexes	Generators for some simplicial complexes.
randomizing	Functions to randomize hypergaphs

19.1 xgi.generators.classic

Generators for some classic hypergraphs.

All the functions in this module return a Hypergraph class (i.e. a simple, undirected hypergraph).

Functions

xgi.generators.classic.**empty_hypergraph**(*create_using=None*, *default=None*)

Returns the empty hypergraph with zero nodes and edges.

Parameters

- **create_using** (*Hypergraph Instance, Constructor or None*) – If None, use the *default* constructor. If a constructor, call it to create an empty hypergraph.
- **default** (*Hypergraph constructor (default None)*) – The constructor to use if create_using is None. If None, then xgi.Hypergraph is used.

Returns

An empty hypergraph

Return type

Hypergraph object

See also:

empty_simplicial_complex, *trivial_hypergraph*

Examples

```
>>> import xgi
>>> H = xgi.empty_hypergraph()
>>> H.num_nodes, H.num_edges
(0, 0)
```

xgi.generators.classic.**empty_simplicial_complex**(*create_using=None*, *default=None*)

Returns the empty simplicial complex with zero nodes and simplices.

Parameters

- **create_using** (*SimplicialComplex Instance, Constructor or None*) – If None, use the *default* constructor. If a constructor, call it to create an empty simplicial complex.
- **default** (*SimplicialComplex constructor (default None)*) – The constructor to use if create_using is None. If None, then xgi.SimplicialComplex is used.

Returns

An empty simplicial complex.

Return type

SimplicialComplex

See also:

empty_hypergraph, *trivial_hypergraph*

Examples

```
>>> import xgi
>>> H = xgi.empty_simplicial_complex()
>>> H.num_nodes, H.num_edges
(0, 0)
```

xgi.generators.classic.**trivial_hypergraph**(*n=1*, *create_using=None*, *default=None*)

Returns a hypergraph with *n* nodes and zero edges.

Parameters

- **n** (*int, optional*) – Number of nodes (default is 1)
- **create_using** (*Hypergraph Instance, Constructor or None*) – If None, use the *default* constructor. If a constructor, call it to create an empty hypergraph.
- **default** (*Hypergraph constructor (default None)*) – The constructor to use if create_using is None. If None, then xgi.Hypergraph is used.

Returns

A trivial hypergraph with *n* nodes

Return type

Hypergraph object

See also:

empty_hypergraph, *empty_simplicial_complex*

Examples

```
>>> import xgi
>>> H = xgi.trivial_hypergraph()
>>> H.num_nodes, H.num_edges
(1, 0)
```

xgi.generators.classic.**complete_hypergraph**(*N*, *order=None*, *max_order=None*, *include_singletons=False*)

Generate a complete hypergraph, i.e. one that contains all possible hyperdges at a given *order* or up to a *max_order*.

Parameters

- **N** (*int*) – Number of nodes
- **order** (*int or None*) – If not None (default), specifies the single order for which to generate hyperedges
- **max_order** (*int or None*) – If not None (default), specifies the maximum order for which to generate hyperedges
- **include_singletons** (*bool*) – Whether to include singleton edges (default: False). This argument is discarded if max_order is None.

Returns
A complete hypergraph with *N* nodes

Return type
Hypergraph object

Notes

Only one of *order* and *max_order* can be specified by and int (not None). Additionally, at least one of either must be specified.

The number of possible edges grows exponentially as 2^N for large *N* and quickly becomes impractically long to compute, especially when using *max_order*. For example, *N=100* and *max_order=5* already yields 10^8 edges. Increasing *N=1000* makes it 10^{13}. *N=100* and with a larger *max_order=6* yields 10^9 edges.

xgi.generators.classic.**complement**(*H*)

Returns the complement of hypergraph H.

The complement Hc of a hypergraph H has the same nodes (same indexes) as H and contains every possible hyperedge linking these nodes, except those present in H. Also, the maximum hyperedge size in Hc is the same as in H. Hyperedges of size one are taken into account.

Parameters
H (*xgi.Hypergraph*) – Hypergraph to complement.

Returns
Hc – Complement of H.

Return type
xgi.Hypergraph

19.2 xgi.generators.simple

Generators for some simple hypergraphs.

All the functions in this module return a Hypergraph class (i.e. a simple, undirected hypergraph).

Functions

xgi.generators.simple.**star_clique**(*n_star*, *n_clique*, *d_max*)

Generate a star-clique structure

That is a star network and a clique network, connected by one pairwise edge connecting the centre of the star to the clique. network, the each clique is promoted to a hyperedge up to order d_max.

Parameters

- **n_star** (*int*) – Number of legs of the star
- **n_clique** (*int*) – Number of nodes in the clique
- **d_max** (*int*) – Maximum order up to which to promote cliques to hyperedges

Returns
H

Return type
Hypergraph

Examples

```
>>> import xgi
>>> H = xgi.star_clique(6, 7, 2)
```

Notes

The total number of nodes is n_star + n_clique.

xgi.generators.simple.**sunflower**(*l*, *c*, *m*)

Create a sunflower hypergraph.

This creates an m-uniform hypergraph according to the sunflower model.

Parameters

- **l** (*int*) – Number of petals
- **c** (*int*) – Size of the core
- **m** (*int*) – Size of each edge

Raises
XGIError – If the edge size is smaller than the core.

19.3 xgi.generators.lattice

Generators for some lattice hypergraphs.

All the functions in this module return a Hypergraph class (i.e. a simple, undirected hypergraph).

Functions

xgi.generators.lattice.**ring_lattice**(*n*, *d*, *k*, *l*)

A ring lattice hypergraph.

A d-uniform hypergraph on n nodes where each node is part of k edges and the overlap between consecutive edges is d-l.

Parameters

- **n** (*int*) – Number of nodes
- **d** (*int*) – Edge size
- **k** (*int*) – Number of edges of which a node is a part. Should be a multiple of 2.
- **l** (*int*) – Overlap between edges

Returns
The generated hypergraph

Return type
Hypergraph

Raises
XGIError – If k is negative.

Notes

ring_lattice(n, 2, k, 0) is a ring lattice graph where each node has k//2 edges on either side.

19.4 xgi.generators.random

Generate random (non-uniform) hypergraphs.

Functions

xgi.generators.random.**chung_lu_hypergraph**(*k1*, *k2*, *seed=None*)

A function to generate a Chung-Lu hypergraph

Parameters

- **k1** (*dictionary*) – Dictionary where the keys are node ids and the values are node degrees.
- **k2** (*dictionary*) – Dictionary where the keys are edge ids and the values are edge sizes.
- **seed** (*integer or None (default)*) – The seed for the random number generator.

Returns
The generated hypergraph

Return type
Hypergraph object

Warns
warnings.warn – If the sums of the edge sizes and node degrees are not equal, the algorithm still runs, but raises a warning.

Notes

The sums of k1 and k2 should be the same. If they are not the same, this function returns a warning but still runs.

References

Implemented by Mirah Shi in HyperNetX and described for bipartite networks by Aksoy et al. in https://doi.org/10.1093/comnet/cnx001

Example

```
>>> import xgi
>>> import random
>>> n = 100
>>> k1 = {i : random.randint(1, 100) for i in range(n)}
>>> k2 = {i : sorted(k1.values())[i] for i in range(n)}
>>> H = xgi.chung_lu_hypergraph(k1, k2)
```

xgi.generators.random.**dcsbm_hypergraph**(*k1*, *k2*, *g1*, *g2*, *omega*, *seed=None*)

A function to generate a Degree-Corrected Stochastic Block Model (DCSBM) hypergraph.

Parameters

- **k1** (*dict*) – This is a dictionary where the keys are node ids and the values are node degrees.
- **k2** (*dict*) – This is a dictionary where the keys are edge ids and the values are edge sizes.
- **g1** (*dict*) – This a dictionary where the keys are node ids and the values are the group ids to which the node belongs. The keys must match the keys of k1.
- **g2** (*dict*) – This a dictionary where the keys are edge ids and the values are the group ids to which the edge belongs. The keys must match the keys of k2.
- **omega** (*2D numpy array*) – This is a matrix with entries which specify the number of edges between a given node community and edge community. The number of rows must match the number of node communities and the number of columns must match the number of edge communities.
- **seed** (*int or None (default)*) – Seed for the random number generator.

Return type
Hypergraph

Warns
warnings.warn – If the sums of the edge sizes and node degrees are not equal, the algorithm still runs, but raises a warning. Also if the sum of the omega matrix does not match the sum of degrees, a warning is raised.

Notes

The sums of k1 and k2 should be the same. If they are not the same, this function returns a warning but still runs. The sum of k1 (and k2) and omega should be the same. If they are not the same, this function returns a warning but still runs and the number of entries in the incidence matrix is determined by the omega matrix.

References

Implemented by Mirah Shi in HyperNetX and described for bipartite networks by Larremore et al. in https://doi.org/10.1103/PhysRevE.90.012805

Examples

```
>>> import xgi; import random; import numpy as np
>>> n = 50
>>> k1 = {i : random.randint(1, n) for i in range(n)}
>>> k2 = {i : sorted(k1.values())[i] for i in range(n)}
>>> g1 = {i : random.choice([0, 1]) for i in range(n)}
>>> g2 = {i : random.choice([0, 1]) for i in range(n)}
>>> omega = np.array([[n//2, 10], [10, n//2]])
>>> # H = xgi.dcsbm_hypergraph(k1, k2, g1, g2, omega)
```

xgi.generators.random.**random_hypergraph**(*N*, *ps*, *order=None*, *seed=None*)

Generates a random hypergraph

Generate N nodes, and connect any d+1 nodes by a hyperedge with probability ps[d-1].

Parameters

- **N** (*int*) – Number of nodes
- **ps** (*list of float*) – List of probabilities (between 0 and 1) to create a hyperedge at each order d between any d+1 nodes. For example, ps[0] is the wiring probability of any edge (2 nodes), ps[1] of any triangles (3 nodes).
- **order** (*int of None (default)*) – If None, ignore. If int, generates a uniform hypergraph with edges of order *order* (ps must have only one element).
- **seed** (*integer or None (default)*) – Seed for the random number generator.

Returns

The generated hypergraph

Return type

Hypergraph object

References

Described as 'random hypergraph' by M. Dewar et al. in https://arxiv.org/abs/1703.07686

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(50, [0.1, 0.01])
```

xgi.generators.random.**watts_strogatz_hypergraph**(*n*, *d*, *k*, *l*, *p*, *seed=None*)

19.5 xgi.generators.uniform

Generate random uniform hypergraphs.

Functions

xgi.generators.uniform.**uniform_hypergraph_configuration_model**(*k*, *m*, *seed=None*)

A function to generate an m-uniform configuration model

Parameters

- **k** (*dictionary*) – This is a dictionary where the keys are node ids and the values are node degrees.
- **m** (*int*) – specifies the hyperedge size
- **seed** (*integer or None (default)*) – The seed for the random number generator

Returns
The generated hypergraph

Return type
Hypergraph object

Warns
warnings.warn – If the sums of the degrees are not divisible by m, the algorithm still runs, but raises a warning and adds an additional connection to random nodes to satisfy this condition.

Notes

This algorithm normally creates multi-edges and loopy hyperedges. We remove the loopy hyperedges.

References

"The effect of heterogeneity on hypergraph contagion models" by Nicholas W. Landry and Juan G. Restrepo https://doi.org/10.1063/5.0020034

Example

```
>>> import xgi
>>> import random
>>> n = 1000
>>> m = 3
>>> k = {1: 1, 2: 2, 3: 3, 4: 3}
>>> H = xgi.uniform_hypergraph_configuration_model(k, m)
```

xgi.generators.uniform.**uniform_erdos_renyi_hypergraph**(*n*, *m*, *p*, *p_type='degree'*, *seed=None*)

Generate an m-uniform Erdős–Rényi hypergraph

This creates a hypergraph with *n* nodes where hyperedges of size *m* are created at random to obtain a mean degree of *k*.

Parameters

- **n** (*int > 0*) – Number of nodes
- **m** (*int > 0*) – Hyperedge size
- **p** (*float or int > 0*) – Mean expected degree if p_type="degree" and probability of an m-hyperedge if p_type="prob"
- **p_type** (*str*) – "degree" or "prob", by default "degree"
- **seed** (*integer or None (default)*) – The seed for the random number generator

Returns
The Erdos Renyi hypergraph

Return type
Hypergraph

See also:

random_hypergraph

xgi.generators.uniform.**uniform_HSBM**(*n*, *m*, *p*, *sizes*, *seed=None*)

Create a uniform hypergraph stochastic block model (HSBM).

Parameters

- **n** (*int*) – The number of nodes
- **m** (*int*) – The hyperedge size
- **p** (*m-dimensional numpy array*) – tensor of probabilities between communities
- **sizes** (*list or 1D numpy array*) – The sizes of the community blocks in order
- **seed** (*integer or None (default)*) – The seed for the random number generator

Returns
The constructed SBM hypergraph

Return type
Hypergraph

Raises

- **XGIError** –
  - If the length of sizes and p do not match. - If p is not a tensor with every dimension equal - If p is not m-dimensional - If the entries of p are not in the range [0, 1] - If the sum of the vector of sizes does not equal the number of nodes.
- **Exception** – If there is an integer overflow error

See also:

uniform_HPPM

References

Nicholas W. Landry and Juan G. Restrepo. "Polarization in hypergraphs with community structure." Preprint, 2023. https://doi.org/10.48550/arXiv.2302.13967

xgi.generators.uniform.**uniform_HPPM**(*n*, *m*, *k*, *epsilon*, *rho=0.5*, *seed=None*)

Construct the m-uniform hypergraph planted partition model (m-HPPM)

Parameters

- **n** (*int > 0*) – Number of nodes
- **m** (*int > 0*) – Hyperedge size
- **k** (*float > 0*) – Mean degree
- **epsilon** (*float > 0*) – Imbalance parameter
- **rho** (*float between 0 and 1, optional*) – The fraction of nodes in community 1, default 0.5
- **seed** (*integer or None (default)*) – The seed for the random number generator

Returns
The constructed m-HPPM hypergraph.

Return type
Hypergraph

Raises
XGIError –

- If rho is not between 0 and 1 - If the mean degree is negative. - If epsilon is not between 0 and 1

See also:

uniform_HSBM

References

Nicholas W. Landry and Juan G. Restrepo. "Polarization in hypergraphs with community structure." Preprint, 2023. https://doi.org/10.48550/arXiv.2302.13967

19.6 xgi.generators.simplicial_complexes

Generators for some simplicial complexes.

All the functions in this module return a SimplicialComplex class.

Functions

xgi.generators.simplicial_complexes.**flag_complex**(*G*, *max_order=2*, *ps=None*, *seed=None*)

Generate a flag (or clique) complex from a NetworkX graph by filling all cliques up to dimension max_order.

Parameters

- **G** (*Networkx Graph*) –
- **max_order** (*int*) – maximal dimension of simplices to add to the output simplicial complex
- **ps** (*list of float*) – List of probabilities (between 0 and 1) to create a hyperedge from a clique, at each order d. For example, ps[0] is the probability of promoting any 3-node clique (triangle) to a 3-hyperedge.
- **seed** (*int or None (default)*) – The seed for the random number generator

Returns
S

Return type
SimplicialComplex

Notes

Computing all cliques quickly becomes heavy for large networks. *flag_complex_d2* is faster to compute up to order 2.

See also:

flag_complex_d2

xgi.generators.simplicial_complexes.**flag_complex_d2**(*G*, *p2=None*, *seed=None*)

Generate a flag (or clique) complex from a NetworkX graph by filling all cliques up to dimension 2.

Parameters

- **G** (*networkx Graph*) – Graph to consider
- **p2** (*float*) – Probability (between 0 and 1) of filling empty triangles in graph G
- **seed** (*int or None (default)*) – The seed for the random number generator

Returns
S

Return type
xgi.SimplicialComplex

Notes

Computing all cliques quickly becomes heavy for large networks. This is faster than *flag_complex* to compute up to order 2.

See also:

flag_complex

xgi.generators.simplicial_complexes.**random_flag_complex**(*N*, *p*, *max_order=2*, *seed=None*)

Generate a flag (or clique) complex from a $G_{N,p}$ Erdős-Rényi random graph.

This proceeds by filling all cliques up to dimension max_order.

Parameters

- **N** (*int*) – Number of nodes
- **p** (*float*) – Probability (between 0 and 1) to create an edge between any 2 nodes
- **max_order** (*int*) – maximal dimension of simplices to add to the output simplicial complex
- **seed** (*int or None (default)*) – The seed for the random number generator

Return type

SimplicialComplex

Notes

Computing all cliques quickly becomes heavy for large networks.

xgi.generators.simplicial_complexes.**random_flag_complex_d2**(*N*, *p*, *seed=None*)

Generate a maximal simplicial complex (up to order 2) from a $G_{N,p}$ Erdős-Rényi random graph.

This proceeds by filling all empty triangles in the graph with 2-simplices.

Parameters

- **N** (*int*) – Number of nodes
- **p** (*float*) – Probabilities (between 0 and 1) to create an edge between any 2 nodes
- **seed** (*int or None (default)*) – The seed for the random number generator

Return type

SimplicialComplex

Notes

Computing all cliques quickly becomes heavy for large networks.

xgi.generators.simplicial_complexes.**random_simplicial_complex**(*N*, *ps*, *seed=None*)

Generates a random hypergraph

Generate N nodes, and connect any d+1 nodes by a simplex with probability ps[d-1]. For each simplex, add all its subfaces if they do not already exist.

Parameters

- **N** (*int*) – Number of nodes

- **ps** (*list of float*) – List of probabilities (between 0 and 1) to create a hyperedge at each order d between any d+1 nodes. For example, ps[0] is the wiring probability of any edge (2 nodes), ps[1] of any triangles (3 nodes).
- **seed** (*int or None (default)*) – The seed for the random number generator

Returns
The generated simplicial complex

Return type
Simplicialcomplex object

References

Described as 'random simplicial complex' in "Simplicial Models of Social Contagion", Nature Communications 10(1), 2485, by I. Iacopini, G. Petri, A. Barrat & V. Latora (2019). https://doi.org/10.1038/s41467-019-10431-6

Example

```
>>> import xgi
>>> H = xgi.random_simplicial_complex(20, [0.1, 0.01])
```

19.7 xgi.generators.randomizing

Functions to randomize hypergaphs

All the functions in this module return a Hypergraph class (i.e. a simple, undirected hypergraph).

Functions

xgi.generators.randomizing.**shuffle_hyperedges**(*S*, *order*, *p*)

Shuffle existing hyperdeges of order *order* with probablity *p*.

Parameters

- **S** (*xgi.HyperGraph*) – Hypergraph
- **order** (*int*) – Order of hyperedges to shuffle
- **p** (*float*) – Probability of shuffling each hyperedge

Returns
H – Hypergraph with edges of order d shuffled

Return type
xgi.HyperGraph

Note: By shuffling hyperedges in a simplicial complex, it will in general lose its "simpliciality" and become a hypergraph.

Zhang, Y.*, Lucas, M.* and Battiston, F., 2023. "Higher-order interactions shape collective dynamics differently in hypergraphs and simplicial complexes." Nature Communications, 14(1), p.1605. https://doi.org/10.1038/s41467-023-37190-9

Example

```
>>> S = xgi.random_simplicial_complex(50, [0.1, 0.01, 0.001], seed=1)
>>> H = xgi.shuffle_hyperedges(S, order=2, p=0.5)
```

xgi.generators.randomizing.**node_swap**(*H*, *nid1*, *nid2*, *id_temp=-1*, *order=None*)

Swap nodes *nid1* and node *nid2* in all edges of order *order*.

Parameters

- **H** (*HyperGraph*) – Hypergraph to consider
- **nid1** (*node ID*) – ID of first node to swap
- **nid2** (*node ID*) – ID of second node to swap
- **id_temp** (*node ID*) – Temporary ID given to nodes when swapping
- **order** (*{int, None}, default: None*) – If None, consider all orders. If an integer, consider edges of that order.

Returns

- **HH** (*HyperGraph*)
- *Reference*
- ———
- *Zhang, Y.*, Lucas, M.* and Battiston, F., 2023.*
- *"Higher-order interactions shape collective dynamics differently*
- *in hypergraphs and simplicial complexes."*
- *Nature Communications, 14(1), p.1605.*
- **https** (*//doi.org/10.1038/s41467-023-37190-9*)

CHAPTER

TWENTY

LINALG PACKAGE

Modules

hypergraph_matrix	General matrices associated to hypergraphs.
laplacian_matrix	Laplacian matrices associated to hypergraphs.
hodge_matrix	Hodge theory matrices associated to hypergraphs.

20.1 xgi.linalg.hypergraph_matrix

General matrices associated to hypergraphs.

Note that the order of the rows and columns of the matrices in this module correspond to the order in which nodes/edges are added to the hypergraph or simplicial complex. If the node and edge IDs are able to be sorted, the following is an example to sort by the node and edge IDs.

```
>>> import xgi
>>> import pandas as pd
>>> H = xgi.Hypergraph([[1, 2, 3, 7], [4], [5, 6, 7]])
>>> I, nodedict, edgedict = xgi.incidence_matrix(H, sparse=False, index=True)
>>> # Sorting the resulting numpy array:
>>> sortedI = I.copy()
>>> sortedI = sortedI[sorted(nodedict, key=nodedict.get), :]
>>> sortedI = sortedI[:, sorted(edgedict, key=edgedict.get)]
>>> sortedI
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [0, 1, 0],
       [0, 0, 1],
       [0, 0, 1],
       [1, 0, 1]])
>>> # Indexing a Pandas dataframe by the node/edge IDs
>>> df = pd.DataFrame(I, index=nodedict.values(), columns=edgedict.values())
```

If the nodes are already sorted, this order can be preserved by adding the nodes to the hypergraph prior to adding edges. For example,

```
>>> import xgi
>>> H = xgi.Hypergraph()
```

(continues on next page)

(continued from previous page)

```
>>> H.add_nodes_from(range(1, 8))
>>> H.add_edges_from([[1, 2, 3, 7], [4], [5, 6, 7]])
>>> xgi.incidence_matrix(H, sparse=False)
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [0, 1, 0],
       [0, 0, 1],
       [0, 0, 1],
       [1, 0, 1]])
```

Functions

xgi.linalg.hypergraph_matrix.**adjacency_matrix**(*H*, *order=None*, *sparse=True*, *s=1*, *weighted=False*, *index=False*)

A function to generate an adjacency matrix (N,N) from a Hypergraph object.

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **order** (*int, optional*) – Order of interactions to use. If None (default), all orders are used. If int, must be >= 1.
- **sparse** (*bool, default: True*) – Specifies whether the output matrix is a scipy sparse matrix or a numpy matrix
- **s** (*int, default: 1*) – Specifies the number of overlapping edges to be considered connected.
- **weighted** (*bool*) – If True, entry (i, j) [and (j, i)] is the number of edges that connect i and j. If False, entry (i, j) [and (j, i] is 1 if i and j share at least one edge and 0 otherwise. By default, False.
- **index** (*bool, default: False*) – Specifies whether to output disctionaries mapping the node IDs to indices

Returns

- *if index is True* – return A, rowdict
- *else* – return A

Warns

warn – If there are isolated nodes and the matrix is sparse.

xgi.linalg.hypergraph_matrix.**clique_motif_matrix**(*H*, *sparse=True*, *index=False*)

A function to generate a weighted clique motif matrix from a Hypergraph object.

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **sparse** (*bool, default: True*) – Specifies whether the output matrix is a scipy sparse matrix or a numpy matrix
- **index** (*bool, default: False*) – Specifies whether to output dictionaries mapping the node and edge IDs to indices

Returns

- *if index is True* – return W, rowdict
- *else* – return W

References

"Higher-order organization of complex networks" by Austin Benson, David Gleich, and Jure Leskovic https://doi.org/10.1126/science.aad9029

xgi.linalg.hypergraph_matrix.**degree_matrix**(*H*, *order=None*, *index=False*)

Returns the degree of each node as an array

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **order** (*int, optional*) – Order of interactions to use. If None (default), all orders are used. If int, must be >= 1.
- **index** (*bool, default: False*) – Specifies whether to output disctionaries mapping the node and edge IDs to indices.

Returns

- *if index is True* – return K, rowdict
- *else* – return K

xgi.linalg.hypergraph_matrix.**incidence_matrix**(*H*, *order=None*, *sparse=True*, *index=False*, *weight=<function <lambda>>*)

A function to generate a weighted incidence matrix from a Hypergraph object, where the rows correspond to nodes and the columns correspond to edges.

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **order** (*int, optional*) – Order of interactions to use. If None (default), all orders are used. If int, must be >= 1.
- **sparse** (*bool, default: True*) – Specifies whether the output matrix is a scipy sparse matrix or a numpy matrix
- **index** (*bool, default: False*) – Specifies whether to output dictionaries mapping the node and edge IDs to indices.
- **weight** (*lambda function, default=lambda function outputting 1*) – A function specifying the weight, given a node and edge

Returns

- **I** (*numpy.ndarray or scipy csr_array*) – The incidence matrix, has dimension (n_nodes, n_edges)
- **rowdict** (*dict*) – The dictionary mapping indices to node IDs, if index is True
- **coldict** (*dict*) – The dictionary mapping indices to edge IDs, if index is True

xgi.linalg.hypergraph_matrix.**intersection_profile**(*H*, *order=None*, *sparse=True*, *index=False*)

A function to generate an intersection profile from a Hypergraph object.

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **order** (*int, optional*) – Order of interactions to use. If None (default), all orders are used. If int, must be >= 1.
- **sparse** (*bool, default: True*) – Specifies whether the output matrix is a scipy sparse matrix or a numpy matrix
- **index** (*bool, default: False*) – Specifies whether to output dictionaries mapping the edge IDs to indices

Returns

- *if index is True* – return P, rowdict, coldict
- *else* – return P

20.2 xgi.linalg.laplacian_matrix

Laplacian matrices associated to hypergraphs.

Note that the order of the rows and columns of the matrices in this module correspond to the order in which nodes/edges are added to the hypergraph or simplicial complex. If the node and edge IDs are able to be sorted, the following is an example to sort by the node and edge IDs.

```
>>> import xgi
>>> import pandas as pd
>>> H = xgi.Hypergraph([[1, 2, 3, 7], [4], [5, 6, 7]])
>>> I, nodedict, edgedict = xgi.incidence_matrix(H, sparse=False, index=True)
>>> # Sorting the resulting numpy array:
>>> sortedI = I.copy()
>>> sortedI = sortedI[sorted(nodedict, key=nodedict.get), :]
>>> sortedI = sortedI[:, sorted(edgedict, key=edgedict.get)]
>>> sortedI
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [0, 1, 0],
       [0, 0, 1],
       [0, 0, 1],
       [1, 0, 1]])
>>> # Indexing a Pandas dataframe by the node/edge IDs
>>> df = pd.DataFrame(I, index=nodedict.values(), columns=edgedict.values())
```

If the nodes are already sorted, this order can be preserved by adding the nodes to the hypergraph prior to adding edges. For example,

```
>>> import xgi
>>> H = xgi.Hypergraph()
>>> H.add_nodes_from(range(1, 8))
>>> H.add_edges_from([[1, 2, 3, 7], [4], [5, 6, 7]])
>>> xgi.incidence_matrix(H, sparse=False)
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
```

(continues on next page)

(continued from previous page)

```
       [0, 1, 0],
       [0, 0, 1],
       [0, 0, 1],
       [1, 0, 1]])
```

Functions

xgi.linalg.laplacian_matrix.**laplacian**(*H*, *order=1*, *sparse=False*, *rescale_per_node=False*, *index=False*)

Laplacian matrix of order d, see [1].

Parameters

- **HG** (Hypergraph) – Hypergraph
- **order** (*int*) – Order of interactions to consider. If order=1 (default), returns the usual graph Laplacian.
- **sparse** (*bool, default: False*) – Specifies whether the output matrix is a scipy sparse matrix or a numpy matrix.
- **index** (*bool, default: False*) – Specifies whether to output disctionaries mapping the node and edge IDs to indices.

Returns

- **L_d** (*numpy array*) – Array of dim (N, N)
- *if index is True* – return rowdict

See also:

multiorder_laplacian

References

xgi.linalg.laplacian_matrix.**multiorder_laplacian**(*H*, *orders*, *weights*, *sparse=False*, *rescale_per_node=False*, *index=False*)

Multiorder Laplacian matrix, see [1].

Parameters

- **HG** (Hypergraph) – Hypergraph
- **orders** (*list of int*) – Orders of interactions to consider.
- **weights** (*list of float*) – Weights associated to each order, i.e coupling strengths gamma_i in [1].
- **sparse** (*bool, default: False*) – Specifies whether the output matrix is a scipy sparse matrix or a numpy matrix
- **rescale_per_node** (*bool, (default=False)*) – Whether to rescale each Laplacian of order d by d (per node).
- **index** (*bool, default: False*) – Specifies whether to output dictionaries mapping the node and edge IDs to indices.

Returns

- **L_multi** (*numpy array*) – Array of dim (N, N)

- *if index is True* – return rowdict

See also:

laplacian

References

xgi.linalg.laplacian_matrix.**normalized_hypergraph_laplacian**(*H*, *sparse=True*, *index=False*)

Compute the normalized Laplacian.

Parameters

- **H** (Hypergraph) – Hypergraph
- **sparse** (*bool, optional*) – whether or not the laplacian is sparse, by default True
- **index** (*bool, optional*) – whether to return a dictionary mapping IDs to rows, by default False

Returns

- *array* – csr_array if sparse and if not, a numpy ndarray
- *dict* – a dictionary mapping node IDs to rows and columns if index is True.

Raises

XGIError – If there are isolated nodes.

References

"Learning with Hypergraphs: Clustering, Classification, and Embedding" by Dengyong Zhou, Jiayuan Huang, Bernhard Schölkopf Advances in Neural Information Processing Systems (2006)

20.3 xgi.linalg.hodge_matrix

Hodge theory matrices associated to hypergraphs.

Note that the order of the rows and columns of the matrices in this module correspond to the order in which nodes/edges are added to the hypergraph or simplicial complex. If the node and edge IDs are able to be sorted, the following is an example to sort by the node and edge IDs.

```
>>> import xgi
>>> import pandas as pd
>>> H = xgi.Hypergraph([[1, 2, 3, 7], [4], [5, 6, 7]])
>>> I, nodedict, edgedict = xgi.incidence_matrix(H, sparse=False, index=True)
>>> # Sorting the resulting numpy array:
>>> sortedI = I.copy()
>>> sortedI = sortedI[sorted(nodedict, key=nodedict.get), :]
>>> sortedI = sortedI[:, sorted(edgedict, key=edgedict.get)]
>>> sortedI
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [0, 1, 0],
```

(continues on next page)

(continued from previous page)

```
       [0, 0, 1],
       [0, 0, 1],
       [1, 0, 1]])
>>> # Indexing a Pandas dataframe by the node/edge IDs
>>> df = pd.DataFrame(I, index=nodedict.values(), columns=edgedict.values())
```

If the nodes are already sorted, this order can be preserved by adding the nodes to the hypergraph prior to adding edges. For example,

```
>>> import xgi
>>> H = xgi.Hypergraph()
>>> H.add_nodes_from(range(1, 8))
>>> H.add_edges_from([[1, 2, 3, 7], [4], [5, 6, 7]])
>>> xgi.incidence_matrix(H, sparse=False)
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [0, 1, 0],
       [0, 0, 1],
       [0, 0, 1],
       [1, 0, 1]])
```

Functions

xgi.linalg.hodge_matrix.**boundary_matrix**(*S*, *order=1*, *orientations=None*, *index=False*)

Generate the boundary matrices of an oriented simplicial complex.

The rows correspond to the (order-1)-simplices and the columns to the (order)-simplices.

Parameters

- **S** (*simplicial complex object*) – The simplicial complex of interest
- **order** (*int, default: 1*) – Specifies the order of the boundary matrix to compute
- **orientations** (*dict, default: None*) – Dictionary mapping non-singleton simplices IDs to their boolean orientation
- **index** (*bool, default: False*) – Specifies whether to output dictionaries mapping the simplices IDs to indices

Returns

- **B** (*numpy.ndarray*) – The boundary matrix of the chosen order, has dimension (n_simplices of given order - 1, n_simplices of given order)
- **rowdict** (*dict*) – The dictionary mapping indices to (order-1)-simplices IDs, if index is True
- **coldict** (*dict*) – The dictionary mapping indices to (order)-simplices IDs, if index is True

References

"Discrete Calculus" by Leo J. Grady and Jonathan R. Polimeni https://doi.org/10.1007/978-1-84996-290-2

xgi.linalg.hodge_matrix.**hodge_laplacian**(*S*, *order=1*, *orientations=None*, *index=False*)

A function to compute the Hodge Laplacians of an oriented simplicial complex.

Parameters

- **S** (*simplicial complex object*) – The simplicial complex of interest
- **order** (*int, default: 1*) – Specifies the order of the Hodge Laplacian matrix to be computed
- **orientations** (*dict, default: None*) – Dictionary mapping non-singleton simplices IDs to their boolean orientation
- **index** (*bool, default: False*) – Specifies whether to output dictionaries mapping the simplices IDs to indices

Returns

- **L_o** (*numpy.ndarray*) – The Hodge Laplacian matrix of the chosen order, has dimension (n_simplices of given order, n_simplices of given order)
- **matdict** (*dict*) – The dictionary mapping indices to (order)-simplices IDs, if index is True

CHAPTER

TWENTYONE

READWRITE PACKAGE

Modules

bigg_data	Load a metabolic network from the BiGG models database.
bipartite	Read from and write to bipartite formats.
edgelist	Read from and write to edgelists.
incidence	Read from and write to incidece matrices.
json	Read from and write to JSON.
xgi_data	Load a data set from the xgi-data repository or a local file.

21.1 xgi.readwrite.bigg_data

Load a metabolic network from the BiGG models database.

Functions

xgi.readwrite.bigg_data.**load_bigg_data**(*dataset=None*, *cache=True*)

Load a metabolic network from the BiGG models database.

The Biochemical, Genetic and Genomic (BiGG) knowledge base is hosted at http://bigg.ucsd.edu/. It contains metabolic reaction networks at the genome scale.

We represent metabolites as nodes and metabolic reactions as directed edges where reactants are the tail of the directed edge and the products are the head of the directed edge.

Parameters

- **dataset** (*str, default: None*) – Dataset name. Valid options are the "bigg_id" tags in http://bigg.ucsd.edu/api/v2/models. If None, prints the list of available datasets.
- **cache** (*bool, optional*) – Whether to cache the input data

Returns

The loaded dihypergraph.

Return type

DiHypergraph

Raises

XGIError – The specified dataset does not exist.

References

Zachary A. King, Justin Lu, Andreas Dräger, Philip Miller, Stephen Federowicz, Joshua A. Lerman, Ali Ebrahim, Bernhard O. Palsson, Nathan E. Lewis Nucleic Acids Research, Volume 44, Issue D1, 4 January 2016, Pages D515–D522, https://doi.org/10.1093/nar/gkv1049

21.2 xgi.readwrite.bipartite

Read from and write to bipartite formats.

Functions

xgi.readwrite.bipartite.**generate_bipartite_edgelist**(*H*, *delimiter=' '*)

A helper function to generate a bipartite edge list from a Hypergraph object.

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **delimiter** (*char, default: space (" ")*) – Specifies the delimiter between hyperedge members

Yields

A iterator of strings – Each entry is a line to be written to the output file.

xgi.readwrite.bipartite.**parse_bipartite_edgelist**(*lines*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *edgetype=None*, *dual=False*)

A helper function to read a iterable of strings containing a bipartite edge list and convert it to a Hypergraph object.

Reads the first two entries of each line and assumes that the first entry is a node ID and that the second entry is an edge ID. Raises error if there are fewer than two entries.

Parameters

- **lines** (*iterable of strings*) – Lines where each line is a bipartite edge
- **comments** (*string, default: "#"*) – The token that denotes comments to ignore
- **delimiter** (*char, default: space (" ")*) – Specifies the delimiter between hyperedge members
- **create_using** (*Hypergraph constructor, optional*) – The hypergraph object to add the data to, by default None
- **nodetype** (*type*) – type that the node labels will be cast to
- **edgetype** (*type*) – type that the edge labels will be cast to
- **data** (*bool, default: False*) – Specifies whether there is a dictionary of data at the end of the line.

Raises

- **XGIError** – If a line contains fewer than two entries
- **TypeError** – If node types fail to be converted

Returns
The loaded hypergraph.

Return type
Hypergraph

xgi.readwrite.bipartite.**read_bipartite_edgelist**(*path*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *edgetype=None*, *dual=False*, *encoding='utf-8'*)

Read a file containing a bipartite edge list and convert it to a Hypergraph object.

Parameters

- **path** (*string*) – The path of the file to read from
- **comments** (*string, default: "#"*) – The token that denotes comments in the file
- **delimiter** (*char, default: space (" ")*) – Specifies the delimiter between hyperedge members
- **create_using** (*Hypergraph constructor, optional*) – The hypergraph object to add the data to, by default None
- **nodetype** (*type*) – type that the node labels will be cast to
- **edgetype** (*type*) – type that the edge labels will be cast to
- **dual** (*bool, default: False*) – Specifies whether the node IDs are in the second column. If False, the node IDs are in the first column.
- **encoding** (*string, default: "utf-8"*) – Encoding of the file

Returns
The loaded hypergraph

Return type
A Hypergraph object

See also:

write_bipartite_edgelist

Example

```
>>> import xgi
>>> # H = xgi.read_bipartite_edgelist("test.csv", delimiter=",")
```

xgi.readwrite.bipartite.**write_bipartite_edgelist**(*H*, *path*, *delimiter=' '*, *encoding='utf-8'*)

Write a Hypergraph object to a file as a bipartite edgelist.

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **path** (*string*) – The path of the file to write to
- **delimiter** (*char, default: space (" ")*) – Specifies the delimiter between hyperedge members
- **encoding** (*string, default: "utf-8"*) – Encoding of the file

See also:

read_bipartite_edgelist

Example

```
>>> import xgi
>>> H = xgi.random_hypergraph(50, [0.01, 0.001])
>>> # xgi.write_bipartite_edgelist(H, "test.csv", delimiter=",")
```

21.3 xgi.readwrite.edgelist

Read from and write to edgelists.

Functions

xgi.readwrite.edgelist.**generate_edgelist**(*H*, *delimiter=' '*)

A helper function to generate a hyperedge list from a Hypergraph object.

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **delimiter** (*char, default: space (" ")*) – Specifies the delimiter between hyperedge members

Yields

iterator of strings – Each entry is a line for the file to write.

xgi.readwrite.edgelist.**parse_edgelist**(*lines*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*)

A helper function to read a iterable of strings containing a hyperedge list and convert it to a Hypergraph object.

Parameters

- **lines** (*iterable of strings*) – Lines where each line is an edge
- **comments** (*string, default: "#"*) – The token that denotes comments to ignore
- **delimiter** (*char, default: space (" ")*) – Specifies the delimiter between hyperedge members
- **create_using** (*Hypergraph constructor, optional*) – The hypergraph object to add the data to, by default None
- **nodetype** (*type*) – type that the node labels will be cast to

Returns

The loaded hypergraph

Return type

Hypergraph object

xgi.readwrite.edgelist.**read_edgelist**(*path*, *comments='#'*, *delimiter=None*, *create_using=None*, *nodetype=None*, *encoding='utf-8'*)

Read a file containing a hyperedge list and convert it to a Hypergraph object.

Parameters

- **path** (*string*) – The path of the file to read from
- **comments** (*string, default: "#"*) – The token that denotes comments in the file

- **delimiter** (*char, default: space (" ")*) – Specifies the delimiter between hyperedge members
- **create_using** (*Hypergraph constructor, optional*) – The hypergraph object to add the data to, by default None
- **nodetype** (*type*) – type that the node labels will be cast to
- **encoding** (*string, default: "utf-8"*) – Encoding of the file

Returns
The loaded hypergraph

Return type
Hypergraph object

See also:

`read_weighted_edgelist`

Examples

```
>>> import xgi
>>> # H = xgi.read_edgelist("test.csv", delimiter=",")
```

`xgi.readwrite.edgelist.`**write_edgelist**(*H*, *path*, *delimiter=' '*, *encoding='utf-8'*)

Create a file containing a hyperedge list from a Hypergraph object.

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **path** (*string*) – The path of the file to write to
- **delimiter** (*char, default: space (" ")*) – Specifies the delimiter between hyperedge members
- **encoding** (*string, default: "utf-8"*) – Encoding of the file

Examples

```
>>> import xgi
>>> H = xgi.random_hypergraph(50, [0.01, 0.001])
>>> # xgi.write_edgelist(H, "test.csv", delimiter=",")
```

21.4 xgi.readwrite.incidence

Read from and write to incidece matrices.

Functions

xgi.readwrite.incidence.**read_incidence_matrix**(*path*, *comments='#'*, *delimiter=None*, *create_using=None*, *encoding='utf-8'*)

Read a file containing an incidence matrix and convert it to a Hypergraph object.

Parameters

- **path** (*string*) – The path of the file to read from
- **comments** (*string, default: "#"*) – The token that denotes comments in the file
- **delimiter** (*char, default: space (" ")*) – Specifies the delimiter between hyperedge members
- **create_using** (*Hypergraph constructor, optional*) – The hypergraph object to add the data to, by default None
- **nodetype** (*type*) – type that the node labels will be cast to
- **encoding** (*string, default: "utf-8"*) – Encoding of the file

Returns
The loaded hypergraph

Return type
A Hypergraph object

See also:

write_incidence_matrix

Examples

```
>>> import xgi
>>> # H = xgi.read_incidence_matrix("test.csv", delimiter=",")
```

xgi.readwrite.incidence.**write_incidence_matrix**(*H*, *path*, *delimiter=' '*, *encoding='utf-8'*)

Write a Hypergraph object to a file as an incidence matrix.

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **path** (*string*) – The path of the file to write to
- **delimiter** (*char, default: space (" ")*) – Specifies the delimiter between hyperedge members
- **encoding** (*string, default: "utf-8"*) – Encoding of the file

See also:

read_incidence_matrix

Examples

```
>>> import xgi
>>> H = xgi.random_hypergraph(50, [0.01, 0.001])
>>> # xgi.write_incidence_matrix(H, "test.csv", delimiter=",")
```

21.5 xgi.readwrite.json

Read from and write to JSON.

Functions

xgi.readwrite.json.**read_json**(*path*, *nodetype=None*, *edgetype=None*)

A function to read a file in a standardized JSON format.

Parameters

- **data** (*dict*) – A dictionary in the hypergraph JSON format
- **nodetype** (*type, optional*) – type that the node IDs will be cast to
- **edgetype** (*type, optional*) – type that the edge IDs will be cast to

Returns
The loaded hypergraph

Return type
A Hypergraph object

Raises
XGIError – If the JSON is not in a format that can be loaded.

xgi.readwrite.json.**write_json**(*H*, *path*)

A function to write a file in a standardized JSON format.

Parameters

- **H** (*Hypergraph object*) – The specified hypergraph object
- **path** (*string*) – The path of the file to read from

21.6 xgi.readwrite.xgi_data

Load a data set from the xgi-data repository or a local file.

Functions

xgi.readwrite.xgi_data.**load_xgi_data**(*dataset=None*, *cache=True*, *read=False*, *path=''*, *nodetype=None*, *edgetype=None*, *max_order=None*)

Load a data set from the xgi-data repository or a local file.

Parameters

- **dataset** (*str, optional*) – Dataset name. Valid options are the top-level tags of the index.json file in the xgi-data repository. If None (default), prints the list of available datasets.
- **cache** (*bool, optional*) – Whether to cache the input data, by default True.
- **read** (*bool, optional*) – If read==True, search for a local copy of the data set. Use the local copy if it exists, otherwise use the xgi-data repository. By default, False.
- **path** (*str, optional*) – Path to a local copy of the data set
- **nodetype** (*type, optional*) – Type to cast the node ID to, by default None.
- **edgetype** (*type, optional*) – Type to cast the edge ID to, by default None.
- **max_order** (*int, optional*) – Maximum order of edges to add to the hypergraph, by default None.

Returns

The loaded hypergraph.

Return type

Hypergraph

Raises

XGIError – The specified dataset does not exist.

xgi.readwrite.xgi_data.**download_xgi_data**(*dataset*, *path=''*)

Make a local copy of a dataset in the xgi-data repository.

Parameters

- **dataset** (*str*) – Dataset name. Valid options are the top-level tags of the index.json file in the xgi-data repository.
- **path** (*str, optional*) – Path to where the local copy should be saved. If none is given, save file to local directory.

CHAPTER

TWENTYTWO

DYNAMICS PACKAGE

Modules

synchronization	Simulation of the Kuramoto model.

22.1 xgi.dynamics.synchronization

Simulation of the Kuramoto model.

Functions

xgi.dynamics.synchronization.**simulate_kuramoto**(*H*, *k2*, *k3*, *omega=None*, *theta=None*, *timesteps=10000*, *dt=0.002*)

Simulates the Kuramoto model on hypergraphs. This solves the Kuramoto model ODE on hypergraphs with edges of sizes 2 and 3 using the Euler Method. It returns timeseries of the phases.

Parameters

- **H** (*Hypergraph object*) – The hypergraph on which you run the Kuramoto model
- **k2** (*float*) – The coupling strength for links
- **k3** (*float*) – The coupling strength for triangles
- **omega** (*numpy array of real values*) – The natural frequency of the nodes. If None (default), randomly drawn from a normal distribution
- **theta** (*numpy array of real values*) – The initial phase distribution of nodes. If None (default), drawn from a random uniform distribution on [0, 2pi[.
- **timesteps** (*int greater than 1, default: 10000*) – The number of timesteps for Euler Method.
- **dt** (*float greater than 0, default: 0.002*) – The size of timesteps for Euler Method.

Returns

- **theta_time** (*numpy array of floats*) – Timeseries of phases from the Kuramoto model, of dimension (T, N)
- **times** (*numpy array of floats*) – Times corresponding to the simulate phases

References

"Synchronization of phase oscillators on complex hypergraphs" by Sabina Adhikari, Juan G. Restrepo and Per Sebastian Skardal https://doi.org/10.48550/arXiv.2208.00909

Examples

```
>>> import numpy as np
>>> import xgi
>>> n = 50
>>> H = xgi.random_hypergraph(n, [0.05, 0.001], seed=None)
>>> omega = 2*np.ones(n)
>>> theta = np.linspace(0, 2*np.pi, n)
>>> theta_time, times = simulate_kuramoto(H, k2=2, k3=3, omega=omega, theta=theta)
```

xgi.dynamics.synchronization.**compute_kuramoto_order_parameter**(*theta_time*)

Calculate the order parameter for the Kuramoto model on hypergraphs.

Calculation proceeds from time series, and the output is a measure of synchrony.

Parameters

theta_time (*numpy array of floats*) – Timeseries of phases from the Kuramoto model, of dimension (T, N)

Returns

r_time – Timeseries for Kuramoto model order parameter

Return type

numpy array of floats

xgi.dynamics.synchronization.**simulate_simplicial_kuramoto**(*S*, *orientations=None*, *order=1*, *omega=[]*, *sigma=1*, *theta0=[]*, *T=10*, *n_steps=10000*, *index=False*)

Simulate the simplicial Kuramoto model's dynamics on an oriented simplicial complex using explicit Euler numerical integration scheme.

Parameters

- **S** (*simplicial complex object*) – The simplicial complex on which you run the simplicial Kuramoto model
- **orientations** (*dict, Default : None*) – Dictionary mapping non-singleton simplices IDs to their boolean orientation
- **order** (*integer*) – The order of the oscillating simplices
- **omega** (*numpy.ndarray*) – The simplicial oscillators' natural frequencies, has dimension (n_simplices of given order, 1)
- **sigma** (*positive real value*) – The coupling strength
- **theta0** (*numpy.ndarray*) – The initial phase distribution, has dimension (n_simplices of given order, 1)
- **T** (*positive real value*) – The final simulation time.
- **n_steps** (*integer greater than 1*) – The number of integration timesteps for the explicit Euler method.

- **index** (*bool, default: False*) – Specifies whether to output dictionaries mapping the node and edge IDs to indices.

Returns

- **theta** (*numpy.ndarray*) – Timeseries of the simplicial oscillators' phases, has dimension (n_simplices of given order, n_steps)
- **theta_minus** (*numpy array of floats*) – Timeseries of the projection of the phases onto lower order simplices, has dimension (n_simplices of given order - 1, n_steps)
- **theta_plus** (*numpy array of floats*) – Timeseries of the projection of the phases onto higher order simplices, has dimension (n_simplices of given order + 1, n_steps)
- **om1_dict** (*dict*) – The dictionary mapping indices to (order-1)-simplices IDs, if index is True
- **o_dict** (*dict*) – The dictionary mapping indices to (order)-simplices IDs, if index is True
- **op1_dict** (*dict*) – The dictionary mapping indices to (order+1)-simplices IDs, if index is True

References

"Explosive Higher-Order Kuramoto Dynamics on Simplicial Complexes" by Ana P. Millán, Joaquín J. Torres, and Ginestra Bianconi https://doi.org/10.1103/PhysRevLett.124.218301

xgi.dynamics.synchronization.**compute_simplicial_order_parameter**(*theta_minus*, *theta_plus*)

This function computes the simplicial order parameter of a simplicial Kuramoto dynamics simulation.

Parameters

- **theta_minus** (*numpy.ndarray*) – Timeseries of the projection of the phases onto lower order simplices, has dimension (n_simplices of given order - 1, n_steps)
- **theta_plus** (*numpy.ndarray*) – Timeseries of the projection of the phases onto higher order simplices, has dimension (n_simplices of given order + 1, n_steps)

Returns

R – Timeseries of the simplicial order parameter, has dimension (1, n_steps)

Return type

numpy.ndarray

References

"Connecting Hodge and Sakaguchi-Kuramoto through a mathematical framework for coupled oscillators on simplicial complexes" by Alexis Arnaudon, Robert L. Peach, Giovanni Petri, and Paul Expert https://doi.org/10.1038/s42005-022-00963-7

CHAPTER

TWENTYTHREE

DRAWING PACKAGE

Modules

layout	Algorithms to compute node positions for drawing.
draw(H[, pos, ax, node_fc, node_ec, ...])	Draw hypergraph or simplicial complex.

23.1 xgi.drawing.layout

Algorithms to compute node positions for drawing.

Functions

xgi.drawing.layout.**random_layout**(*H*, *center=None*, *seed=None*)

Position nodes uniformly at random in the unit square.

For every node, a position is generated by choosing each of dim coordinates uniformly at random on the interval [0.0, 1.0). NumPy (http://scipy.org) is required for this function.

Parameters

- **H** (*Hypergraph or SimplicialComplex*) – A position will be assigned to every node in HG.
- **center** (*array-like, optional*) – Coordinate pair around which to center the layout. If None (default), does not center the positions.
- **seed** (*int, optional*) – Set the random state for deterministic node layouts. If int, *seed* is the seed used by the random number generator, If None (default), random numbers are sampled from the numpy random number generator without initialization.

Returns

pos – A dictionary of positions keyed by node

Return type

dict

See also:

pairwise_spring_layout, *barycenter_spring_layout*, *weighted_barycenter_spring_layout*

Notes

This function proceeds exactly as NetworkX does.

Examples

```
>>> import xgi
>>> N = 50
>>> ps = [0.1, 0.01]
>>> H = xgi.random_hypergraph(N, ps)
>>> pos = xgi.random_layout(H)
```

xgi.drawing.layout.**pairwise_spring_layout**(*H*, *seed=None*, *k=None*, ***kwargs*)

Position the nodes using Fruchterman-Reingold force-directed algorithm using the graph projection of the hypergraph or the hypergraph constructed from the simplicial complex.

Parameters

- **H** (Hypergraph *or* SimplicialComplex) – A position will be assigned to every node in H.
- **seed** (*int, optional*) – Set the random state for deterministic node layouts. If int, *seed* is the seed used by the random number generator, If None (default), random numbers are sampled from the numpy random number generator without initialization.
- **k** (*float*) – The spring constant of the links. When k=None (default), k = 1/sqrt(N). For more information, see the documentation for the NetworkX spring_layout() function.
- **kwargs** – Optional arguments for the NetworkX spring_layout() function. See https://networkx.org/documentation/stable/reference/generated/networkx.drawing.layout.spring_layout.html

Returns

pos – A dictionary of positions keyed by node

Return type

dict

See also:

random_layout, *barycenter_spring_layout*, *weighted_barycenter_spring_layout*

Notes

If a simplicial complex is provided the results will be based on the hypergraph constructed from its maximal simplices.

Examples

```
>>> import xgi
>>> N = 50
>>> ps = [0.1, 0.01]
>>> H = xgi.random_hypergraph(N, ps)
>>> pos = xgi.pairwise_spring_layout(H)
```

xgi.drawing.layout.**barycenter_spring_layout**(*H*, *return_phantom_graph=False*, *seed=None*, *k=None*, ***kwargs*)

Position the nodes using Fruchterman-Reingold force-directed algorithm using an augmented version of the the graph projection of the hypergraph (or simplicial complex), where phantom nodes (barycenters) are created for each edge composed by more than two nodes. If a simplicial complex is provided the results will be based on the hypergraph constructed from its maximal simplices.

Parameters

- **H** (*xgi Hypergraph or SimplicialComplex*) – A position will be assigned to every node in H.
- **return_phantom_graph** (*bool (default=False)*) – If True the function returns also the augmented version of the the graph projection of the hypergraph (or simplicial complex).
- **seed** (*int, RandomState instance or None optional (default=None)*) – Set the random state for deterministic node layouts. If int, *seed* is the seed used by the random number generator, If None (default), random numbers are sampled from the numpy random number generator without initialization.
- **k** (*float*) – The spring constant of the links. When k=None (default), k = 1/sqrt(N). For more information, see the documentation for the NetworkX spring_layout() function.
- **kwargs** – Optional arguments for the NetworkX spring_layout() function. See https://networkx.org/documentation/stable/reference/generated/networkx.drawing.layout.spring_layout.html

Returns
pos – A dictionary of positions keyed by node

Return type
dict

See also:

random_layout, *pairwise_spring_layout*, *weighted_barycenter_spring_layout*

Examples

```
>>> import xgi
>>> N = 50
>>> ps = [0.1, 0.01]
>>> H = xgi.random_hypergraph(N, ps)
>>> pos = xgi.barycenter_spring_layout(H)
```

xgi.drawing.layout.**edge_positions_from_barycenters**(*H*, *node_pos*)

Given a higher-order network and node positions, assigns edge marker positions to be the barycenters of its member nodes.

Parameters

- **H** (Hypergraph, SimplicialComplex, *or* DiHypergraph) – A position will be assigned to every node and edge in H.
- **node_pos** (*dict*) – Nodal positions where keys are node ids and values are Numpy arrays of the positions.

Returns
edge_pos – a dictionary of positions keyed by edge

Return type
dict

See also:

bipartite_spring_layout

Examples

```
>>> import xgi
>>> N = 50
>>> ps = [0.1, 0.01]
>>> H = xgi.random_hypergraph(N, ps)
>>> node_pos = xgi.pairwise_spring_layout(H)
>>> edge_pos = xgi.edge_positions_from_barycenters(H, node_pos)
```

xgi.drawing.layout.**weighted_barycenter_spring_layout**(*H*, *return_phantom_graph=False*, *seed=None*, *k=None*, ***kwargs*)

Position the nodes using Fruchterman-Reingold force-directed algorithm.

This uses an augmented version of the the graph projection of the hypergraph (or simplicial complex), where phantom nodes (barycenters) are created for each edge of order d>1 (composed by more than two nodes). Weights are assigned to all hyperedges of order 1 (links) and to all connections to phantom nodes within each hyperedge to keep them together. Weights scale as the order d. If a simplicial complex is provided the results will be based on the hypergraph constructed from its maximal simplices.

Parameters

- **H** (Hypergraph *or* SimplicialComplex) – A position will be assigned to every node in H.
- **return_phantom_graph** (*bool (default=False)*) – If True the function returns also the augmented version of the the graph projection of the hypergraph (or simplicial complex).
- **seed** (*int, RandomState instance or None optional (default=None)*) – Set the random state for deterministic node layouts. If int, *seed* is the seed used by the random number generator, If None (default), random numbers are sampled from the numpy random number generator without initialization.
- **k** (*float*) – The spring constant of the links. When k=None (default), k = 1/sqrt(N). For more information, see the documentation for the NetworkX spring_layout() function.
- **kwargs** – Optional arguments for the NetworkX spring_layout() function. See https://networkx.org/documentation/stable/reference/generated/networkx.drawing.layout.spring_layout.html

Returns
pos – A dictionary of positions keyed by node

Return type
dict

See also:

random_layout, *pairwise_spring_layout*, *barycenter_spring_layout*

Examples

```
>>> import xgi
>>> N = 50
>>> ps = [0.1, 0.01]
>>> H = xgi.random_hypergraph(N, ps)
>>> pos = xgi.weighted_barycenter_spring_layout(H)
```

xgi.drawing.layout.**pca_transform**(*pos*, *theta=0*, *degrees=True*)

Transforms the positions of the nodes based on the principal components.

Parameters

- **pos** (*dict of numpy arrays*) – The output from any layout function
- **theta** (*float, optional*) – The angle between the horizontal axis and the principal axis measured counterclockwise, by default 0.
- **degrees** (*bool, optional*) – Whether the angle specified is in degrees (True) or in radians (False), by default True.

Returns
The transformed positions.

Return type
dict of numpy arrays

See also:

random_layout, *pairwise_spring_layout*, *barycenter_spring_layout*, *weighted_barycenter_spring_layout*

xgi.drawing.layout.**circular_layout**(*H*, *center=None*, *radius=None*)

Position nodes on a circle.

Parameters

- **H** (Hypergraph *or* SimplicialComplex) – A position will be assigned to every node in H.
- **center** (*array-like or None*) – Coordinate pair around which to center the layout. If None set to [0,0]
- **radius** (*float or None (default=None)*) – Radius of the circle on which to draw the nodes, if None set to 1.0.

Returns
pos – A dictionary of positions keyed by node

Return type
dict

xgi.drawing.layout.**spiral_layout**(*H*, *center=None*, *resolution=0.35*, *equidistant=False*)

Position nodes in a spiral layout.

Parameters

- **H** (Hypergraph *or* SimplicialComplex) – A position will be assigned to every node in H.
- **center** (*array-like or None*) – Coordinate pair around which to center the layout. If None set to [0,0]
- **resolution** (*float, default=0.35*) – The compactness of the spiral layout returned. Lower values result in more compressed spiral layouts.
- **equidistant** (*bool, default=False*) – If True, nodes will be positioned equidistant from each other by decreasing angle further from center. If False, nodes will be positioned at equal angles from each other by increasing separation further from center.

Returns
pos – A dictionary of positions keyed by node

Return type
dict

xgi.drawing.layout.**barycenter_kamada_kawai_layout**(*H*, *return_phantom_graph=False*, ***kwargs*)

Position nodes using Kamada-Kawai path-length cost-function using an augmented version of the the graph projection of the hypergraph (or simplicial complex), where phantom nodes (barycenters) are created for each edge composed by more than two nodes. If a simplicial complex is provided the results will be based on the hypergraph constructed from its maximal simplices.

Parameters

- **H** (*xgi Hypergraph or* SimplicialComplex) – A position will be assigned to every node in H.
- **return_phantom_graph** (*bool (default=False)*) – If True the function returns also the augmented version of the the graph projection of the hypergraph (or simplicial complex).
- **kwargs** – Optional arguments for the NetworkX spring_layout() function. See https://networkx.org/documentation/stable/reference/generated/networkx.drawing.layout.kamada_kawai_layout.html

Returns
pos – A dictionary of positions keyed by node

Return type
dict

23.2 xgi.drawing.draw

Draw hypergraphs and simplicial complexes with matplotlib.

Functions

xgi.drawing.draw.**draw**(*H*, *pos=None*, *ax=None*, *node_fc='white'*, *node_ec='black'*, *node_lw=1*, *node_size=7*, *node_shape='o'*, *node_fc_cmap='Reds'*, *vmin=None*, *vmax=None*, *max_order=None*, *dyad_color='black'*, *dyad_lw=1.5*, *dyad_style='solid'*, *dyad_color_cmap='Greys'*, *dyad_vmin=None*, *dyad_vmax=None*, *edge_fc=None*, *edge_fc_cmap='crest_r'*, *edge_vmin=None*, *edge_vmax=None*, *alpha=0.4*, *hull=False*, *radius=0.05*, *node_labels=False*, *hyperedge_labels=False*, *rescale_sizes=True*, *aspect='equal'*, ***kwargs*)

Draw hypergraph or simplicial complex.

Parameters

- **H** (Hypergraph *or SimplicialComplex.*) – Hypergraph to draw
- **pos** (*dict, optional*) – If passed, this dictionary of positions node_id:(x,y) is used for placing the 0-simplices. If None (default), use the *barycenter_spring_layout* to compute the positions.
- **ax** (*matplotlib.pyplot.axes, optional*) – Axis to draw on. If None (default), get the current axes.
- **node_fc** (*str, dict, iterable, or* NodeStat*, optional*) – Color of the nodes. If str, use the same color for all nodes. If a dict, must contain (node_id: color_str) pairs. If other iterable, assume the colors are specified in the same order as the nodes are found in H.nodes. If NodeStat, use the colormap specified with node_fc_cmap. By default, "white".
- **node_ec** (*str, dict, iterable, or* NodeStat*, optional*) – Color of node borders. If str, use the same color for all nodes. If a dict, must contain (node_id: color_str) pairs. If other iterable, assume the colors are specified in the same order as the nodes are found in H.nodes. If NodeStat, use the colormap specified with node_ec_cmap. By default, "black".
- **node_lw** (*int, float, dict, iterable, or* NodeStat*, optional*) – Line width of the node borders in pixels. If int or float, use the same width for all node borders. If a dict, must contain (node_id: width) pairs. If iterable, assume the widths are specified in the same order as the nodes are found in H.nodes. If NodeStat, use a monotonic linear interpolation defined between min_node_lw and max_node_lw. By default, 1.
- **node_size** (*int, float, dict, iterable, or* NodeStat*, optional*) – Radius of the nodes in pixels. If int or float, use the same radius for all nodes. If a dict, must contain (node_id: radius) pairs. If iterable, assume the radiuses are specified in the same order as the nodes are found in H.nodes. If NodeStat, use a monotonic linear interpolation defined between min_node_size and max_node_size. By default, 15.
- **node_shape** (*string, optional*) – The shape of the node. Specification is as matplotlib.scatter marker. Default is "o".
- **node_fc_cmap** (*colormap*) – Colormap for mapping node colors. By default, "Reds". Ignored, if *node_fc* is a str (single color).
- **vmin** (*float or None*) – Minimum for the node_fc_cmap scaling. By default, None.
- **vmax** (*float or None*) – Maximum for the node_fc_cmap scaling. By default, None.
- **max_order** (*int, optional*) – Maximum of hyperedges to plot. If None (default), plots all orders.
- **dyad_color** (*str, dict, iterable, or* EdgeStat*, optional*) – Color of the dyadic links. If str, use the same color for all edges. If a dict, must contain (edge_id: color_str) pairs. If iterable, assume the colors are specified in the same order as the edges are found in H.edges. If EdgeStat, use a colormap (specified with dyad_color_cmap) associated to it. By default, "black".
- **dyad_lw** (*int, float, dict, iterable, or* EdgeStat*, optional*) – Line width of edges of order 1 (dyadic links). If int or float, use the same width for all edges. If a dict, must contain (edge_id: width) pairs. If iterable, assume the widths are specified in the same order as the edges are found in H.edges. If EdgeStat, use a monotonic linear interpolation defined between min_dyad_lw and max_dyad_lw. By default, 1.5.
- **dyad_style** (*str or list of strings, optional*) – Line style of the dyads, e.g. '-', '–', '-.', ':' or words like 'solid' or 'dashed'. See matplotlib's documentation for all accepted values. By default, "solid".

- **dyad_color_cmap** (*matplotlib colormap*) – Colormap used to map the dyad colors. By default, "Greys".
- **dyad_vmin** (*float, optional*) – Minimum and maximum for dyad colormap scaling. By default, None.
- **dyad_vmax** (*float, optional*) – Minimum and maximum for dyad colormap scaling. By default, None.
- **edge_fc** (*color or list of colors or array-like or dict or* EdgeStat*, optional*) – Color of the hyperedges. The accepted formats are the same as matplotlib's scatter, with the addition of dict and IDStat. Those with colors: * single color as a string * single color as 3- or 4-tuple * list of colors of length len(ids) * dict of colors containing the *ids* as keys

  Those with numerical values (will be mapped to colors): * array of floats * dict of floats containing the *ids* as keys * IDStat containing the *ids* as keys

  If None (default), color by edge size.
- **edge_fc_cmap** (*matplotlib colormap*) – Colormap used to map the edge colors. By default, "cres_r".
- **edge_vmin** (*float, optional*) – Minimum and maximum for edge colormap scaling. By default, None.
- **edge_vmax** (*float, optional*) – Minimum and maximum for edge colormap scaling. By default, None.
- **alpha** (*float, optional*) – The edge transparency. By default, 0.4.
- **hull** (*bool, optional*) – Wether to draw hyperedes as convex hulls. By default, False.
- **radius** (*float, optional*) – Radius margin around the nodes when drawing convex hulls. Ignored if *hull is False*. Default is 0.05.
- **node_labels** (*bool or dict, optional*) – If True, draw ids on the nodes. If a dict, must contain (node_id: label) pairs. By default, False. The default node_size (7) is too small to display the default labels well. The user may need to set it to a size of a least 15.
- **hyperedge_labels** (*bool or dict, optional*) – If True, draw ids on the hyperedges. If a dict, must contain (edge_id: label) pairs. By default, False.
- **aspect** (*{"auto", "equal"} or float, optional*) – Set the aspect ratio of the axes scaling, i.e. y/x-scale. *aspect* is passed directly to matplotlib's *ax.set_aspect()*. Default is *equal*. See full Set the aspect ratio of the axes scaling, i.e. y/x-scale. *aspect* is passed directly to matplotlib's *ax.set_aspect()*. Default is *equal*. See full description at https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.set_aspect.html
- **rescale_sizes** (*bool, optional*) – If True, linearly interpolate *node_size*, *node_lw* and *dyad_lw* between min/max values that can be changed in the other argument *params*. If those are single values, *interpolate_sizes* is ignored for it. By default, True.
- ****kwargs** (*optional args*) – Alternate default values. Values that can be overwritten are the following: * "min_node_size" (default: 5) * "max_node_size" (default: 30) * "min_node_lw" (default: 0) * "max_node_lw" (default: 5) * "min_dyad_lw" (default: 1) * "max_dyad_lw" (default: 10)

Returns

- **ax** (*matplotlib Axes*) – Axes plotted on
- **collections** (*a tuple of 3 collections:*) –

- **node_collection**
  [matplotlib PathCollection] Collection containing the nodes
- **dyad_collection**
  [matplotlib LineCollection] Collection containing the dyads
- **edge_collection**
  [matplotlib PathCollection] Collection containing the edges

Examples

```
>>> import xgi
>>> H = xgi.Hypergraph()
>>> H.add_edges_from([[1,2,3],[3,4],[4,5,6,7],[7,8,9,10,11]])
>>> ax = xgi.draw(H, pos=xgi.barycenter_spring_layout(H))
```

See also:

draw_nodes, *draw_hyperedges*, *draw_simplices*, *draw_node_labels*, *draw_hyperedge_labels*

xgi.drawing.draw.**draw_bipartite**(*H*, *pos=None*, *ax=None*, *node_fc='white'*, *node_ec='black'*, *node_lw=1*, *node_size=7*, *node_shape='o'*, *node_fc_cmap='Reds'*, *edge_marker_fc=None*, *edge_marker_ec='black'*, *edge_marker_lw=1*, *edge_marker_size=7*, *edge_marker_shape='s'*, *edge_marker_fc_cmap='crest_r'*, *max_order=None*, *dyad_color=None*, *dyad_lw=1*, *dyad_style='solid'*, *dyad_color_cmap='crest_r'*, *node_labels=None*, *hyperedge_labels=None*, *arrowsize=10*, *arrowstyle='->'*, *connectionstyle='arc3'*, *rescale_sizes=True*, *aspect='equal'*, ***kwargs*)

Draw a hypergraph as a bipartite network.

Parameters

- **H** (*Hypergraph or DiHypergraph*) – The hypergraph to draw.
- **pos** (*tuple of two dicts, optional*) – The tuple should contain a (1) dictionary of positions node_id:(x,y) for placing node markers, and (2) a dictionary of positions edge_id:(x,y) for placing the edge markers. If None (default), use the *bipartite_spring_layout* to compute the positions.
- **ax** (*matplotlib.pyplot.axes, optional*) – Axis to draw on. If None (default), get the current axes.
- **node_fc** (*str, dict, iterable, or NodeStat, optional*) – Color of the nodes. If str, use the same color for all nodes. If a dict, must contain (node_id: color_str) pairs. If other iterable, assume the colors are specified in the same order as the nodes are found in H.nodes. If NodeStat, use the colormap specified with node_fc_cmap. By default, "white".
- **node_ec** (*str, dict, iterable, or NodeStat, optional*) – Color of node borders. If str, use the same color for all nodes. If a dict, must contain (node_id: color_str) pairs. If other iterable, assume the colors are specified in the same order as the nodes are found in H.nodes. If NodeStat, use the colormap specified with node_ec_cmap. By default, "black".
- **node_lw** (*int, float, dict, iterable, or NodeStat, optional*) – Line width of the node borders in pixels. If int or float, use the same width for all node borders. If a dict, must contain (node_id: width) pairs. If iterable, assume the widths are specified in the same order as the nodes are found in H.nodes. If NodeStat, use a monotonic linear interpolation defined between min_node_lw and max_node_lw. By default, 1.

- **node_size** (*int, float, dict, iterable, or* NodeStat*, optional*) – Radius of the nodes in pixels. If int or float, use the same radius for all nodes. If a dict, must contain (node_id: radius) pairs. If iterable, assume the radiuses are specified in the same order as the nodes are found in H.nodes. If NodeStat, use a monotonic linear interpolation defined between min_node_size and max_node_size. By default, 7.
- **node_shape** (*str, optional*) – Marker used for the nodes. By default 'o' (circle marker).
- **node_fc_cmap** (*colormap*) – Colormap for mapping node colors. By default, "Reds". Ignored, if *node_fc* is a str (single color).
- **edge_marker_fc** (*str, dict, iterable, optional*) – Filling color of the hyperedges (markers). If str, use the same color for all hyperedges. If a dict, must contain (hyperedge_id: color_str) pairs. If other iterable, assume the colors are specified in the same order as the hyperedges are found in H.edges. If None, colors markers by edge size. By default, None.
- **edge_marker_ec** (*str, dict, iterable, optional*) – Edge color of the hyperedges (markers). If str, use the same color for all hyperedges. If a dict, must contain (hyperedge_id: color_str) pairs. If other iterable, assume the colors are specified in the same order as the hyperedges are found in H.edges. By default, "black".
- **edge_marker_lw** (*int, float, dict, iterable, or* EdgeStat*, optional*) – Line width of the edge marker borders in pixels. If int or float, use the same width for all edge marker borders. If a dict, must contain (edge_id: width) pairs. If iterable, assume the widths are specified in the same order as the nodes are found in H.edges. If EdgeStat, use a monotonic linear interpolation defined between min_edge_marker_lw and max_edge_marker_lw. By default, 1.
- **edge_marker_size** (*int, float, dict, iterable, or* EdgeStat*, optional*) – Radius of the edge markers in pixels. If int or float, use the same radius for all edge markers. If a dict, must contain (edge_id: radius) pairs. If iterable, assume the radii are specified in the same order as the edges are found in H.edges. If EdgeStat, use a monotonic linear interpolation defined between min_edge_marker_size and max_edge_marker_size. By default, 7.
- **edge_marker_shape** (*str, optional*) – Marker used for the hyperedges. By default 's' (square marker). If "", no marker is displayed.
- **edge_marker_fc_cmap** (*colormap*) – Colormap for mapping edge marker colors. By default, "Blues". Ignored, if *edge_marker_fc* is a str (single color) or an iterable of colors.
- **max_order** (*int, optional*) – Maximum of hyperedges to plot. If None (default), plots all orders.
- **dyad_color** (*str, dict, iterable, optional*) – Color of the bipartite edges. If str, use the same color for all edges. If a dict, must contain (hyperedge_id: color_str) pairs. If other iterable, assume the colors are specified in the same order as the hyperedges are found in H.edges. By default, "black".
- **dyad_lw** (*int, float, dict, iterable, optional*) – Line width of the bipartite edges. If int or float, use the same width for all hyperedges. If a dict, must contain (hyperedge_id: width) pairs. If other iterable, assume the widths are specified in the same order as the hyperedges are found in H.edges. By default, 1.
- **dyad_style** (*str or list of strings, optional*) – Line style of the dyads, e.g. '-', '–', '-.', ':' or words like 'solid' or 'dashed'. See matplotlib's documentation for all accepted values. By default, "solid".
- **dyad_color_cmap** (*colormap*) – Colormap for mapping bipartite edge colors. By default, "Greys". Ignored, if *dyad_color* is a str (single color) or an iterable of colors.

- **node_labels** (*bool or dict, optional*) – If True, draw ids on the nodes. If a dict, must contain (node_id: label) pairs. By default, False.
- **hyperedge_labels** (*bool or dict, optional*) – If True, draw ids on the hyperedges. If a dict, must contain (edge_id: label) pairs. By default, False.
- **arrowsize** (*int (default=10)*) – Size of the arrow head's length and width. See *matplotlib.patches.FancyArrowPatch* for attribute *mutation_scale* for more info. Only used if the higher-order network is a *DiHypergraph*.
- **arrowstyle** (*str, optional*) – By default: '->'. See *matplotlib.patches.ArrowStyle* for more options. Only used if the higher-order network is a *DiHypergraph*.
- **connectionstyle** (*string (default="arc3")*) – Pass the connectionstyle parameter to create curved arc of rounding radius rad. For example, connectionstyle='arc3,rad=0.2'. See *matplotlib.patches.ConnectionStyle* and *matplotlib.patches.FancyArrowPatch* for more info. Only used if the higher-order network is a *DiHypergraph*.
- **rescale_sizes** (*bool, optional*) – If True, linearly interpolate *node_size* and between min/max values that can be changed in the other argument *params*. If *node_size* is a single value, this is ignored. By default, True.
- **aspect** (*{"auto", "equal"} or float, optional*) – Set the aspect ratio of the axes scaling, i.e. y/x-scale. *aspect* is passed directly to matplotlib's *ax.set_aspect()*. Default is *equal*. See full description at https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.set_aspect.html
- ****kwargs** (*optional args*) – Alternate default values. Values that can be overwritten are the following: * min_node_size (default: 5) * max_node_size (default: 30) * min_node_lw (default: 0) * max_node_lw (default: 5) * min_edge_marker_size (default: 5) * max_edge_marker_size (default: 30) * min_edge_marker_lw (default: 0) * max_edge_marker_lw (default: 5) * min_dyad_lw (default: 1) * max_dyad_lw (default: 10) * min_source_margin (default: 0) * min_target_margin (default: 0)

Returns

- **ax** (*matplotlib.pyplot.axes*) – The axes corresponding the visualization
- **collections** (*a tuple of 3 collections:*) –
  - **node_collection**
    [matplotlib PathCollection] Collection containing the nodes
  - **edge_marker_collection**
    [matplotlib PathCollection] Collection containing the edge markers
  - **dyad_collection**
    [matplotlib LineCollection if undirected,]

    list of FancyArrowPatches if not

    Collection containing the edges

Raises

XGIError – If the network is not a Hypergraph, SimplicialComplex or a DiHypergraph.

See also:

draw, *draw_multilayer*

xgi.drawing.draw.**draw_multilayer**(*H*, *pos=None*, *ax=None*, *node_fc='white'*, *node_ec='black'*, *node_lw=1*, *node_size=5*, *node_shape='o'*, *node_fc_cmap='Reds'*, *vmin=None*, *vmax=None*, *dyad_color='grey'*, *dyad_lw=1.5*, *dyad_style='solid'*, *edge_fc=None*, *edge_fc_cmap='crest_r'*, *edge_vmin=None*, *edge_vmax=None*, *alpha=0.4*, *layer_color='grey'*, *layer_cmap='crest_r'*, *max_order=None*, *conn_lines=True*, *conn_lines_style='dotted'*, *h_angle=10*, *v_angle=20*, *sep=0.4*, *rescale_sizes=True*, ***kwargs*)

Draw a hypergraph or simplicial complex visualized in 3D showing hyperedges/simplices of different orders on superimposed layers.

Parameters

- **H** (*Hypergraph or SimplicialComplex.*) – Higher-order network to plot.
- **pos** (*dict or None, optional*) – The positions of the nodes in the multilayer network. If None, a default layout will be computed using xgi.barycenter_spring_layout(). Default is None.
- **ax** (*matplotlib Axes3DSubplot or None, optional*) – The subplot to draw the visualization on. If None, a new subplot will be created. Default is None.
- **node_fc** (*str, iterable, or NodeStat, optional*) – Color of the nodes. If str, use the same color for all nodes. If other iterable, or NodeStat, assume the colors are specified in the same order as the nodes are found in H.nodes. By default, "white".
- **node_ec** (*color or sequence of colors, optional*) – Color of node borders. If color, use the same color for all nodes. If sequence of colors, assume the colors are specified in the same order as the nodes are found in H.nodes. By default, "black".
- **node_lw** (*int, float, iterable, or NodeStat, optional*) – Line width of the node borders in pixels. If int or float, use the same width for all node borders. If iterable or NodeStat, assume the widths are specified in the same order as the nodes are found in H.nodes. Values are clipped below and above by min_node_lw and max_node_lw, respectively. By default, 1.
- **node_size** (*int, float, iterable, or NodeStat, optional*) – Radius of the nodes in pixels. If int or float, use the same radius for all nodes. If iterable or NodeStat, assume the radiuses are specified in the same order as the nodes are found in H.nodes. Values are clipped below and above by min_node_size and max_node_size, respectively. By default, 5.
- **node_shape** (*string, optional*) – The shape of the node. Specification is as matplotlib.scatter marker. Default is "o".
- **node_fc_cmap** (*colormap*) – Colormap for mapping node colors. By default, "Reds". Ignored, if *node_fc* is a str (single color).
- **vmin** (*float or None*) – Minimum for the node_fc_cmap scaling. By default, None.
- **vmax** (*float or None*) – Maximum for the node_fc_cmap scaling. By default, None.
- **dyad_color** (*color or list of colors*) – Color of the dyadic links. If str, use the same color for all edges. If iterable, assume the colors are specified in the same order as the edges are found in H.edges. By default, "black".
- **dyad_lw** (*int, float, dict, iterable, or EdgeStat, optional*) – Line width of edges of order 1 (dyadic links). If int or float, use the same width for all edges. If a dict, must contain (edge_id: width) pairs. If iterable, assume the widths are specified in the same order as the edges are found in H.edges. If EdgeStat, use a monotonic linear interpolation defined between min_dyad_lw and max_dyad_lw. By default, 1.5.

- **dyad_style** (*str or list of strings, optional*) – Line style of the dyads, e.g. '-', '–', '-.', ':' or words like 'solid' or 'dashed'. See matplotlib's documentation for all accepted values. By default, "solid".
- **edge_fc** (*color or list of colors or array-like or dict or* EdgeStat*, optional*) – Color of the hyperedges. The accepted formats are the same as matplotlib's scatter, with the addition of dict and IDStat. Those with colors: * single color as a string * single color as 3- or 4-tuple * list of colors of length len(ids) * dict of colors containing the *ids* as keys Those with numerical values (will be mapped to colors): * array of floats * dict of floats containing the *ids* as keys * IDStat containing the *ids* as keys If None (default), color by edge size.
- **edge_fc_cmap** (*matplotlib colormap, optional*) – Colormap used to map the edge colors. By default, "crest_r".
- **edge_vmin** (*float, optional*) – Minimum and maximum for edge colormap scaling. By default, None.
- **edge_vmax** (*float, optional*) – Minimum and maximum for edge colormap scaling. By default, None.
- **alpha** (*float, optional*) – The edge transparency. By default, 0.4.
- **layer_color** (*color or list of colors, optional*) – Color of layers. By default, "grey".
- **layer_cmap** (*colormap, optional*) – Colormap to use in case of numerical values in layer_color. Ignored if layer_color does not contain numerical values to be mapped. By default, "crest_r", but ignored.
- **max_order** (*int, optional*) – Maximum of hyperedges to plot. If None (default), plots all orders.
- **conn_lines** (*bool, optional*) – Whether to draw connections between layers. Default is True.
- **conn_lines_style** (*str, optional*) – The linestyle of the connections between layers. Default is 'dotted'.
- **h_angle** (*float, optional*) – The rotation angle around the horizontal axis in degrees. Default is 10.
- **v_angle** (*float, optional*) – The rotation angle around the vertical axis in degrees. Default is 0.
- **sep** (*float, optional*) – The separation between layers. Default is 0.4.
- ****kwargs** (*optional args*) – Alternate default values. Values that can be overwritten are the following: * "min_node_size" (default: 10) * "max_node_size" (default: 30) * "min_node_lw" (default: 2) * "max_node_lw" (default: 10) * "min_dyad_lw" (default: 1) * "max_dyad_lw" (default: 5)

Returns

- **ax** (*matplotlib Axes3DSubplot*) – The subplot with the multilayer network visualization.
- **collections** (*a tuple of 2 collections:*) –
  - **node_collection**
    [matplotlib PathCollection] Collection containing the nodes one the top layer
  - **edge_collection**
    [matplotlib PathCollection] Collection containing the edges of size > 2

xgi.drawing.draw.**draw_nodes**(*H*, *pos=None*, *ax=None*, *node_fc='white'*, *node_ec='black'*, *node_lw=1*, *node_size=7*, *node_shape='o'*, *node_fc_cmap='Reds'*, *vmin=None*, *vmax=None*, *zorder=None*, *params={}*, *node_labels=False*, *rescale_sizes=True*, ***kwargs*)

Draw the nodes of a hypergraph

Parameters

- **H** (*Hypergraph or SimplicialComplex*) – Higher-order network to plot.
- **pos** (*dict, optional*) – If passed, this dictionary of positions node_id:(x,y) is used for placing the 0-simplices. If None (default), use the *barycenter_spring_layout* to compute the positions.
- **ax** (*matplotlib.pyplot.axes, optional*) – Axis to draw on. If None (default), get the current axes.
- **node_fc** (*str, iterable, or NodeStat, optional*) – Color of the nodes. If str, use the same color for all nodes. If other iterable, or NodeStat, assume the colors are specified in the same order as the nodes are found in H.nodes. By default, "white".
- **node_ec** (*color or sequence of colors, optional*) – Color of node borders. If color, use the same color for all nodes. If sequence of colors, assume the colors are specified in the same order as the nodes are found in H.nodes. By default, "black".
- **node_lw** (*int, float, iterable, or NodeStat, optional*) – Line width of the node borders in pixels. If int or float, use the same width for all node borders. If iterable or NodeStat, assume the widths are specified in the same order as the nodes are found in H.nodes. Values are clipped below and above by min_node_lw and max_node_lw, respectively. By default, 1.
- **node_size** (*int, float, iterable, or NodeStat, optional*) – Radius of the nodes in pixels. If int or float, use the same radius for all nodes. If iterable or NodeStat, assume the radiuses are specified in the same order as the nodes are found in H.nodes. Values are clipped below and above by min_node_size and max_node_size, respectively. By default, 15.
- **node_shape** (*string, optional*) – The shape of the node. Specification is as matplotlib.scatter marker. Default is "o".
- **node_fc_cmap** (*colormap*) – Colormap for mapping node colors. By default, "Reds". Ignored, if *node_fc* is a str (single color).
- **vmin** (*float or None*) – Minimum for the node_fc_cmap scaling. By default, None.
- **vmax** (*float or None*) – Maximum for the node_fc_cmap scaling. By default, None.
- **zorder** (*int*) – The layer on which to draw the nodes.
- **node_labels** (*bool or dict*) – If True, draw ids on the nodes. If a dict, must contain (node_id: label) pairs. The default node_size (7) is too small to display the default labels well. The user may need to set it to a size of a least 15.
- **rescale_sizes** (*bool, optional*) – If True, linearly interpolate *node_size* and *node_lw* between min/max values (5/30 for size, 0/5 for lw) that can be changed in the other argument *params*. If *node_size* (*node_lw*) is a single value, *interpolate_sizes* is ignored for it. By default, True.
- **params** (*dict*) – Default parameters used if *rescale_sizes* is True. Keys to override default settings: * "min_node_size" (default: 5) * "max_node_size" (default: 30) * "min_node_lw" (default: 0) * "max_node_lw" (default: 5)

- **kwargs** (*optional keywords*) – See *draw_node_labels* for a description of optional keywords.

Returns

- **ax** (*matplotlib Axes*) – Axes plotted on
- **node_collection** (*matplotlib PathCollection*) – Collection containing the nodes

See also:

draw, *draw_hyperedges*, *draw_simplices*, *draw_node_labels*, *draw_hyperedge_labels*

Notes

If nodes are colored with a cmap, the *node_collection* returned can be used to easily plot a colorbar corresponding to the node colors. Simply do *plt.colorbar(node_collection)*.

Nodes with nonfinite *node_fc* (i.e. *inf*, *-inf* or *nan* are drawn with the bad colormap color (see *plotnonfinitebool* in *plt.scatter* and Colormap.set_bad from Matplotlib).

xgi.drawing.draw.**draw_hyperedges**(*H*, *pos=None*, *ax=None*, *dyad_color='black'*, *dyad_lw=1.5*, *dyad_style='solid'*, *dyad_color_cmap='Greys'*, *dyad_vmin=None*, *dyad_vmax=None*, *edge_fc=None*, *edge_fc_cmap='crest_r'*, *edge_vmin=None*, *edge_vmax=None*, *alpha=0.4*, *max_order=None*, *params={}*, *hyperedge_labels=False*, *hull=False*, *radius=0.05*, *rescale_sizes=True*, ***kwargs*)

Draw hyperedges.

Parameters

- **H** (*Hypergraph*) – Hypergraph to plot
- **pos** (*dict, optional*) – If passed, this dictionary of positions node_id:(x,y) is used for placing the 0-simplices. If None (default), use the *barycenter_spring_layout* to compute the positions.
- **ax** (*matplotlib.pyplot.axes, optional*) – Axis to draw on. If None (default), get the current axes.
- **dyad_color** (*str, dict, iterable, or EdgeStat, optional*) – Color of the dyadic links. If str, use the same color for all edges. If a dict, must contain (edge_id: color_str) pairs. If iterable, assume the colors are specified in the same order as the edges are found in H.edges. If EdgeStat, use a colormap (specified with dyad_color_cmap) associated to it. By default, "black".
- **dyad_lw** (*int, float, dict, iterable, or EdgeStat, optional*) – Line width of edges of order 1 (dyadic links). If int or float, use the same width for all edges. If a dict, must contain (edge_id: width) pairs. If iterable, assume the widths are specified in the same order as the edges are found in H.edges. If EdgeStat, use a monotonic linear interpolation defined between min_dyad_lw and max_dyad_lw. By default, 1.5.
- **dyad_style** (*str or list of strings, optional*) – Line style of the dyads, e.g. '-', '–', '-.', ':' or words like 'solid' or 'dashed'. See matplotlib's documentation for all accepted values. By default, "solid".
- **dyad_color_cmap** (*matplotlib colormap*) – Colormap used to map the dyad colors. By default, "Greys".
- **dyad_vmin** (*float, optional*) – Minimum and maximum for dyad colormap scaling. By default, None.

- **dyad_vmax** (*float, optional*) – Minimum and maximum for dyad colormap scaling. By default, None.
- **edge_fc** (*color or list of colors or array-like or dict or* EdgeStat*, optional*) – Color of the hyperedges. The accepted formats are the same as matplotlib's scatter, with the addition of dict and IDStat. Those with colors: * single color as a string * single color as 3- or 4-tuple * list of colors of length len(ids) * dict of colors containing the *ids* as keys

  Those with numerical values (will be mapped to colors): * array of floats * dict of floats containing the *ids* as keys * IDStat containing the *ids* as keys

  If None (default), color by edge size.
- **edge_fc_cmap** (*matplotlib colormap*) – Colormap used to map the edge colors. By default, "crest_r".
- **edge_vmin** (*float, optional*) – Minimum and maximum for edge colormap scaling. By default, None.
- **edge_vmax** (*float, optional*) – Minimum and maximum for edge colormap scaling. By default, None.
- **alpha** (*float, optional*) – The edge transparency. By default, 0.4.
- **max_order** (*int, optional*) – Maximum of hyperedges to plot. By default, None.
- **hyperedge_labels** (*bool or dict, optional*) – If True, draw ids on the hyperedges. If a dict, must contain (edge_id: label) pairs. By default, None.
- **hull** (*bool, optional*) – Wether to draw hyperedes as convex hulls. By default, False.
- **radius** (*float, optional*) – Radius margin around the nodes when drawing convex hulls. Ignored if *hull is False*. Default is 0.05.
- **rescale_sizes** (*bool, optional*) – If True, linearly interpolate *dyad_lw* and between min/max values (1/10) that can be changed in the other argument *params*. If *dyad_lw* is a single value, *interpolate_sizes* is ignored for it. By default, True.
- **params** (*dict*) – Default parameters. Keys that may be useful to override default settings: * "min_dyad_lw" (default: 1) * "max_dyad_lw" (default: 10)
- **kwargs** (*optional keywords*) – See *draw_hyperedge_labels* for a description of optional keywords.

Returns

- **ax** (*matplotlib Axes*) – Axes plotted on
- **collections** (*a tuple of 2 collections:*) –
  - **dyad_collection**
    [matplotlib LineCollection] Collection containing the dyads
  - **edge_collection**
    [matplotlib PathCollection] Collection containing the edges

Raises

XGIError – If a SimplicialComplex is passed.

See also:

draw, *draw_nodes*, *draw_simplices*, *draw_node_labels*, *draw_hyperedge_labels*

xgi.drawing.draw.**draw_undirected_dyads**(*H*, *pos=None*, *ax=None*, *max_order=None*, *dyad_color=None*, *dyad_lw=1*, *dyad_style='solid'*, *dyad_color_cmap='crest_r'*, *rescale_sizes=True*, ***kwargs*)

Draw the bipartite edges of an undirected hypergraph.

Parameters

- **H** (Hypergraph) – The hypergraph to draw.
- **pos** (*tuple of two dicts, optional*) – The tuple should contains a (1) dictionary of positions node_id:(x,y) for placing node markers, and (2) a dictionary of positions edge_id:(x,y) for placing the edge markers. If None (default), use the *bipartite_spring_layout* to compute the positions.
- **ax** (*matplotlib.pyplot.axes, optional*) – Axis to draw on. If None (default), get the current axes.
- **max_order** (*int, optional*) – Maximum of hyperedges to plot. If None (default), plots all orders.
- **dyad_color** (*str, dict, iterable, optional*) – Color of the bipartite edges. If str, use the same color for all edges. If a dict, must contain (hyperedge_id: color_str) pairs. If other iterable, assume the colors are specified in the same order as the hyperedges are found in H.edges. By default, "black".
- **dyad_lw** (*int, float, dict, iterable, optional*) – Line width of the bipartite edges. If int or float, use the same width for all hyperedges. If a dict, must contain (hyperedge_id: width) pairs. If other iterable, assume the widths are specified in the same order as the hyperedges are found in H.edges. By default, 1.
- **dyad_style** (*str or list of strings, optional*) – Line style of the dyads, e.g. '-', '–', '-.', ':' or words like 'solid' or 'dashed'. See matplotlib's documentation for all accepted values. By default, "solid".
- **dyad_color_cmap** (*colormap*) – Colormap for mapping bipartite edge colors. By default, "Greys". Ignored, if *dyad_color* is a str (single color) or an iterable of colors.
- **rescale_sizes** (*bool, optional*) – If True, linearly interpolate *node_size* and between min/max values that can be changed in the other argument *params*. If *node_size* is a single value, this is ignored. By default, True.
- ****kwargs** (*optional args*) – Alternate default values. Values that can be overwritten are the following: * min_dyad_lw (default: 1) * max_dyad_lw (default: 10)

Returns

- **ax** (*matplotlib.pyplot.axes*) – The axes corresponding the visualization
- *** dyad_collection** (*matplotlib LineCollection*) – of bipartite edges

Raises

XGIError – If DiHypergraph is passed.

See also:

draw_bipartite, *draw_directed_dyads*

xgi.drawing.draw.**draw_directed_dyads**(*H*, *pos=None*, *ax=None*, *max_order=None*, *dyad_color=None*, *dyad_lw=1*, *dyad_style='solid'*, *dyad_color_cmap='crest_r'*, *arrowsize=10*, *arrowstyle='->'*, *connectionstyle='arc3'*, *node_size=5*, *node_shape='o'*, *edge_marker_size=5*, *edge_marker_shape='s'*, *rescale_sizes=True*, ***kwargs*)

Draw the bipartite edges of a directed hypergraph.

Parameters

- **H** (DiHypergraph) – The hypergraph to draw.
- **pos** (*tuple of two dicts, optional*) – The tuple should contains a (1) dictionary of positions node_id:(x,y) for placing node markers, and (2) a dictionary of positions edge_id:(x,y) for placing the edge markers. If None (default), use the *bipartite_spring_layout* to compute the positions.
- **ax** (*matplotlib.pyplot.axes, optional*) – Axis to draw on. If None (default), get the current axes.
- **max_order** (*int, optional*) – Maximum of hyperedges to plot. If None (default), plots all orders.
- **dyad_color** (*str, dict, iterable, optional*) – Color of the bipartite edges. If str, use the same color for all edges. If a dict, must contain (hyperedge_id: color_str) pairs. If other iterable, assume the colors are specified in the same order as the hyperedges are found in H.edges. By default, "black".
- **dyad_lw** (*int, float, dict, iterable, optional*) – Line width of the bipartite edges. If int or float, use the same width for all hyperedges. If a dict, must contain (hyperedge_id: width) pairs. If other iterable, assume the widths are specified in the same order as the hyperedges are found in H.edges. By default, 1.
- **dyad_style** (*str or list of strings, optional*) – Line style of the dyads, e.g. '-', '–', '-.', ':' or words like 'solid' or 'dashed'. See matplotlib's documentation for all accepted values. By default, "solid".
- **dyad_color_cmap** (*colormap*) – Colormap for mapping bipartite edge colors. By default, "Greys". Ignored, if *dyad_color* is a str (single color) or an iterable of colors.
- **arrowsize** (*int (default=10)*) – Size of the arrow head's length and width. See *matplotlib.patches.FancyArrowPatch* for attribute *mutation_scale* for more info. Only used if the higher-order network is a *DiHypergraph*.
- **arrowstyle** (*str, optional*) – By default: '->'. See *matplotlib.patches.ArrowStyle* for more options. Only used if the higher-order network is a *DiHypergraph*.
- **connectionstyle** (*string (default="arc3")*) – Pass the connectionstyle parameter to create curved arc of rounding radius rad. For example, connectionstyle='arc3,rad=0.2'. See *matplotlib.patches.ConnectionStyle* and *matplotlib.patches.FancyArrowPatch* for more info. Only used if the higher-order network is a *DiHypergraph*.
- **rescale_sizes** (*bool, optional*) – If True, linearly interpolate *node_size* and between min/max values that can be changed in the other argument *params*. If *node_size* is a single value, this is ignored. By default, True.
- **node_size** (*int, float, dict, iterable, or* NodeStat*, optional*) – Radius of the nodes in pixels. If int or float, use the same radius for all nodes. If a dict, must contain (node_id: radius) pairs. If iterable, assume the radiuses are specified in the same order as the nodes are found in H.nodes. If NodeStat, use a monotonic linear interpolation defined between min_node_size and max_node_size. Used for arrow spacing. By default, 7.
- **node_shape** (*str, optional*) – Marker used for the nodes. Used for arrow spacing. By default 'o' (circle marker).

- **edge_marker_size** (*int, float, dict, iterable, or* EdgeStat, *optional*) – Radius of the edge markers in pixels. If int or float, use the same radius for all edge markers. If a dict, must contain (edge_id: radius) pairs. If iterable, assume the radii are specified in the same order as the edges are found in H.edges. If EdgeStat, use a monotonic linear interpolation defined between min_edge_marker_size and max_edge_marker_size. Used for arrow spacing. By default, 7.
- **edge_marker_shape** (*str, optional*) – Marker used for the hyperedges. If "", no marker is displayed. Used for arrow spacing. By default 's' (square marker).
- ****kwargs** (*optional args*) – Alternate default values. Values that can be overwritten are the following: * min_dyad_lw (default: 1) * max_dyad_lw (default: 10)

Returns

- **ax** (*matplotlib.pyplot.axes*) – The axes corresponding the visualization
- **dyad_collection** (*list of FancyArrowPatches*) – representing directed bipartite edges

Raises

XGIError – If something different than a DiHypergraph is passed.

See also:

draw_bipartite, *draw_directed_dyads*

xgi.drawing.draw.**draw_simplices**(*SC*, *pos=None*, *ax=None*, *dyad_color='black'*, *dyad_lw=1.5*, *dyad_style='solid'*, *dyad_color_cmap='Greys'*, *dyad_vmin=None*, *dyad_vmax=None*, *edge_fc=None*, *edge_fc_cmap='crest_r'*, *edge_vmin=None*, *edge_vmax=None*, *alpha=0.4*, *max_order=None*, *params={}*, *hyperedge_labels=False*, *rescale_sizes=True*, ***kwargs*)

Draw maximal simplices and pairwise faces.

Parameters

- **SC** (SimplicialComplex) – Simplicial complex to draw
- **pos** (*dict, optional*) – If passed, this dictionary of positions node_id:(x,y) is used for placing the 0-simplices. If None (default), use the *barycenter_spring_layout* to compute the positions.
- **ax** (*matplotlib.pyplot.axes, optional*) – Axis to draw on. If None (default), get the current axes.
- **dyad_color** (*str, dict, iterable, or* EdgeStat, *optional*) – Color of the dyadic links. If str, use the same color for all edges. If a dict, must contain (edge_id: color_str) pairs. If iterable, assume the colors are specified in the same order as the edges are found in H.edges. If EdgeStat, use a colormap (specified with dyad_color_cmap) associated to it. By default, "black".
- **dyad_lw** (*int, float, dict, iterable, or* EdgeStat, *optional*) – Line width of edges of order 1 (dyadic links). If int or float, use the same width for all edges. If a dict, must contain (edge_id: width) pairs. If iterable, assume the widths are specified in the same order as the edges are found in H.edges. If EdgeStat, use a monotonic linear interpolation defined between min_dyad_lw and max_dyad_lw. By default, 1.5.
- **dyad_style** (*str or list of strings, optional*) – Line style of the dyads, e.g. '-', '–', '-.', ':' or words like 'solid' or 'dashed'. See matplotlib's documentation for all accepted values. By default, "solid".
- **dyad_color_cmap** (*matplotlib colormap*) – Colormap used to map the dyad colors. By default, "Greys".

- **dyad_vmin** (*float, optional*) – Minimum and maximum for dyad colormap scaling. By default, None.
- **dyad_vmax** (*float, optional*) – Minimum and maximum for dyad colormap scaling. By default, None.
- **edge_fc** (*color or list of colors or array-like or dict or* EdgeStat*, optional*) – Color of the hyperedges. The accepted formats are the same as matplotlib's scatter, with the addition of dict and IDStat. Those with colors: * single color as a string * single color as 3- or 4-tuple * list of colors of length len(ids) * dict of colors containing the *ids* as keys

  Those with numerical values (will be mapped to colors): * array of floats * dict of floats containing the *ids* as keys * IDStat containing the *ids* as keys

  If None (default), color by edge size.
- **edge_fc_cmap** (*matplotlib colormap*) – Colormap used to map the edge colors. By default, "crest_r".
- **edge_vmin** (*float, optional*) – Minimum and maximum for edge colormap scaling. By default, None.
- **edge_vmax** (*float, optional*) – Minimum and maximum for edge colormap scaling. By default, None.
- **alpha** (*float, optional*) – The edge transparency. By default, 0.4.
- **max_order** (*int, optional*) – Maximum of hyperedges to plot. By default, None.
- **hyperedge_labels** (*bool or dict, optional*) – If True, draw ids on the hyperedges. If a dict, must contain (edge_id: label) pairs. By default, None.
- **rescale_sizes** (*bool, optional*) – If True, linearly interpolate *dyad_lw* and between min/max values (1/10) that can be changed in the other argument *params*. If *dyad_lw* is a single value, *interpolate_sizes* is ignored for it. By default, True.
- **params** (*dict*) – Default parameters. Keys that may be useful to override default settings: * "min_dyad_lw" (default: 1) * "max_dyad_lw" (default: 10)
- **kwargs** (*optional keywords*) – See *draw_hyperedge_labels* for a description of optional keywords.

Returns

- *ax*
- **collections** (*a tuple of 2 collections:*) –
  - **dyad_collection**
    [matplotlib LineCollection] Collection containing the dyads
  - **edge_collection**
    [matplotlib PathCollection] Collection containing the edges

Raises

XGIError – If a Hypergraph is passed.

See also:

draw, *draw_nodes*, *draw_hyperedges*, *draw_node_labels*, *draw_hyperedge_labels*

xgi.drawing.draw.**draw_node_labels**(*H*, *pos*, *node_labels=False*, *font_size_nodes=10*, *font_color_nodes='black'*, *font_family_nodes='sans-serif'*, *font_weight_nodes='normal'*, *alpha_nodes=None*, *bbox_nodes=None*, *horizontalalignment_nodes='center'*, *verticalalignment_nodes='center'*, *ax_nodes=None*, *clip_on_nodes=True*, *zorder=None*)

Draw node labels on the hypergraph or simplicial complex.

Parameters

- **H** (*Hypergraph or SimplicialComplex.*) –
- **pos** (*dict*) – Dictionary of positions node_id:(x,y).
- **node_labels** (*bool or dict, optional*) – If True, draw ids on the nodes. If a dict, must contain (node_id: label) pairs. By default, False.
- **font_size_nodes** (*int, optional*) – Font size for text labels, by default 10.
- **font_color_nodes** (*str, optional*) – Font color string, by default "black".
- **font_family_nodes** (*str, optional*) – Font family, by default "sans-serif".
- **font_weight_nodes** (*str (default='normal')*) – Font weight.
- **alpha_nodes** (*float, optional*) – The text transparency, by default None.
- **bbox_nodes** (*Matplotlib bbox, optional*) – Specify text box properties (e.g. shape, color etc.) for node labels. When it is None (default), use Matplotlib's ax.text default
- **horizontalalignment_nodes** (*str, optional*) – Horizontal alignment {'center', 'right', 'left'}. By default, "center".
- **verticalalignment_nodes** (*str, optional*) – Vertical alignment {'center', 'top', 'bottom', 'baseline', 'center_baseline'}. By default, "center".
- **ax_nodes** (*matplotlib.pyplot.axes, optional*) – Draw the graph in the specified Matplotlib axes. By default, None.
- **clip_on_nodes** (*bool, optional*) – Turn on clipping of node labels at axis boundaries. By default, True.
- **zorder** (*int, optional*) – The vertical order on which to draw the labels. By default, None, in which case it is plotted above the last plotted object.

Returns

dict of labels keyed by node id.

Return type

dict

See also:

draw, *draw_nodes*, *draw_hyperedges*, *draw_simplices*, *draw_hyperedge_labels*

xgi.drawing.draw.**draw_hyperedge_labels**(*H*, *pos*, *hyperedge_labels=False*, *font_size_edges=10*, *font_color_edges='black'*, *font_family_edges='sans-serif'*, *font_weight_edges='normal'*, *alpha_edges=None*, *bbox_edges=None*, *horizontalalignment_edges='center'*, *verticalalignment_edges='center'*, *ax_edges=None*, *rotate_edges=False*, *clip_on_edges=True*)

Draw hyperedge labels on the hypegraph or simplicial complex.

Parameters

- **H** (*Hypergraph.*) –
- **pos** (*dict*) – Dictionary of positions node_id:(x,y).
- **hyperedge_labels** (*bool or dict, optional*) – If True, draw ids on the hyperedges. If a dict, must contain (edge_id: label) pairs. By default, False.
- **font_size_edges** (*int, optional*) – Font size for text labels, by default 10.
- **font_color_edges** (*str, optional*) – Font color string, by default "black".
- **font_family_edges** (*str (default='sans-serif')*) – Font family.
- **font_weight_edges** (*str (default='normal')*) – Font weight.
- **alpha_edges** (*float, optional*) – The text transparency, by default None.
- **bbox_edges** (*Matplotlib bbox, optional*) – Specify text box properties (e.g. shape, color etc.) for edge labels. By default, {boxstyle='round', ec=(1.0, 1.0, 1.0), fc=(1.0, 1.0, 1.0)}
- **horizontalalignment_edges** (*str, optional*) – Horizontal alignment {'center', 'right', 'left'}. By default, "center".
- **verticalalignment_edges** (*str, optional*) – Vertical alignment {'center', 'top', 'bottom', 'baseline', 'center_baseline'}. By default, "center".
- **ax_edges** (*matplotlib.pyplot.axes, optional*) – Draw the graph in the specified Matplotlib axes. By default, None.
- **rotate_edges** (*bool, optional*) – Rotate edge labels for dyadic links to lie parallel to edges, by default False.
- **clip_on_edges** (*bool, optional*) – Turn on clipping of hyperedge labels at axis boundaries, by default True.

Returns
dict of labels keyed by hyperedge id.

Return type
dict

See also:

draw, *draw_nodes*, *draw_hyperedges*, *draw_simplices*, *draw_node_labels*

CHAPTER

TWENTYFOUR

CONVERT PACKAGE

Modules

bipartite_edges	Methods for converting to and from bipartite edgelists.
bipartite_graph	Methods for converting to and from bipartite graphs.
encapsulation_dag	
graph	Method for projecting a hypergraph to a graph.
higher_order_network	Methods for converting to higher-order network objects.
hyperedges	Methods for converting to and from hyperedge lists.
hypergraph_dict	Method for converting from a standardized dictionary.
incidence	Methods for converting to and from an incidence matrix.
line_graph	Method for converting to a line graph.
pandas	Methods for converting to and from a Pandas dataframe.
simplex	Methods for simplicial complexes.

24.1 xgi.convert.bipartite_edges

Methods for converting to and from bipartite edgelists.

Functions

xgi.convert.bipartite_edges.**from_bipartite_edgelist**(*edges*, *create_using=None*)

Generate a hypergraph from a list of lists.

Parameters

- **e** (*tuple, list, or array of tuples, lists, or arrays, each of size 2*) – A bipartite edgelist
- **create_using** (*Hypergraph constructor, optional*) – The hypergraph to add the edges to, by default None

Returns

The constructed hypergraph object

Return type

Hypergraph object

See also:

`to_hyperedge_list`

xgi.convert.bipartite_edges.**to_bipartite_edgelist**(*H*)

Generate a hyperedge list from a hypergraph.

Parameters
H (*Hypergraph object*) – The hypergraph of interest

Returns
The hyperedge list

Return type
list of sets

See also:

`from_hyperedge_list`

24.2 xgi.convert.bipartite_graph

Methods for converting to and from bipartite graphs.

Functions

xgi.convert.bipartite_graph.**from_bipartite_graph**(*G*, *create_using=None*, *dual=False*)

Create a Hypergraph from a NetworkX bipartite graph.

Any hypergraph may be represented as a bipartite graph where nodes in the first layer are nodes and nodes in the second layer are hyperedges.

The default behavior is to create nodes in the hypergraph from the nodes in the bipartite graph where the attribute bipartite=0 and hyperedges in the hypergraph from the nodes in the bipartite graph with attribute bipartite=1. Setting the keyword *dual* reverses this behavior.

Parameters

- **G** (*nx.Graph*) – A networkx bipartite graph. Each node in the graph has a property 'bipartite' taking the value of 0 or 1 indicating the type of node.
- **create_using** (*Hypergraph constructor, optional*) – The hypergraph object to add the data to, by default None
- **dual** (*bool, default : False*) – If True, get edges from bipartite=0 and nodes from bipartite=1

Returns
The equivalent hypergraph

Return type
Hypergraph

References

The Why, How, and When of Representations for Complex Systems, Leo Torres, Ann S. Blevins, Danielle Bassett, and Tina Eliassi-Rad, https://doi.org/10.1137/20M1355896

Examples

```
>>> import networkx as nx
>>> import xgi
>>> G = nx.Graph()
>>> G.add_nodes_from([1, 2, 3, 4], bipartite=0)
>>> G.add_nodes_from(['a', 'b', 'c'], bipartite=1)
>>> G.add_edges_from([(1, 'a'), (1, 'b'), (2, 'b'), (2, 'c'), (3, 'c'), (4, 'a')])
>>> H = xgi.from_bipartite_graph(G)
```

xgi.convert.bipartite_graph.**to_bipartite_graph**(*H*, *index=False*)

Create a NetworkX bipartite network from a hypergraph.

Parameters

- **H** (*xgi.Hypergraph*) – The XGI hypergraph object of interest
- **index** (*bool (default False)*) – If False (default), return only the graph. If True, additionally return the index-to-node and index-to-edge mappings.

Returns

The resulting equivalent bipartite graph, and optionally the index-to-unit mappings.

Return type

nx.Graph[, dict, dict]

References

The Why, How, and When of Representations for Complex Systems, Leo Torres, Ann S. Blevins, Danielle Bassett, and Tina Eliassi-Rad, https://doi.org/10.1137/20M1355896

Examples

```
>>> import xgi
>>> hyperedge_list = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(hyperedge_list)
>>> G = xgi.to_bipartite_graph(H)
>>> G, itn, ite = xgi.to_bipartite_graph(H, index=True)
```

24.3 xgi.encapsulation_dag

Functions

xgi.encapsulation_dag.**empirical_subsets_filter**(*H*, *dag*)

Filters encapsulation DAG of H in place to only include edges between hyperedges of size k and the maximum existing k'.

Parameters

- **H** (Hypergraph) – The hypergraph of interest
- **dag** (*nx.DiGraph*) – The encapsulation dag of H constructed with to_encapsulation_dag(H, subset_types="all")

Returns
dag – The filtered line graph (also modified in place)

Return type
networkx.DiGraph

References

"Encapsulation Structure and Dynamics in Hypergraphs", by Timothy LaRock & Renaud Lambiotte. https://arxiv.org/abs/2307.04613

xgi.encapsulation_dag.**to_encapsulation_dag**(*H*, *subset_types='all'*)

The encapsulation DAG (Directed Acyclic Graph) of the hypergraph H.

An encapsulation DAG is a directed line graph where the nodes are hyperedges in H and a directed edge exists from a larger hyperedge to a smaller hyperedge if the smaller is a subset of the larger.

Parameters

- **H** (Hypergraph) – The hypergraph of interest
- **subset_types** (*str, optional*) –

  Type of relations to include. Options are:
  "all" : all subset relationships "immediate" : only subset relationships between hyperedges of

  adjacent sizes (i.e., edges from k to k-1)

  "empirical"
  [A relaxation of the "immediate" option where only] subset relationships between hyperedges of size k and subsets of maximum size k'<k existing in the hypergraph are included. For example, a hyperedge of size 5 may have no immediate encapsulation relationships with hyperedges of size 4, but may encapsulate hyperedegs of size 3, which will be included if using this setting (whereas relationships with subsets of size 2 would not be included).

Returns
LG – The line graph associated to the Hypergraph

Return type
networkx.DiGraph

Examples

```
>>> import xgi
>>> from xgi.convert import to_encapsulation_dag, empirical_subsets_filter
>>> H = xgi.Hypergraph([["a","b","c"], ["b","c","f"], ["a","b"], ["c", "e"], ["a"],␣
↪["f"]])
>>> dag = to_encapsulation_dag(H)
>>> dag.edges()
OutEdgeView([(0, 2), (0, 4), (2, 4), (1, 5)])
>>> dag = to_encapsulation_dag(H, subset_types="immediate")
>>> dag.edges()
OutEdgeView([(0, 2), (2, 4)])
>>> dag = to_encapsulation_dag(H, subset_types="empirical")
>>> dag.edges()
OutEdgeView([(0, 2), (2, 4), (1, 5)])
```

References

"Encapsulation Structure and Dynamics in Hypergraphs", by Timothy LaRock & Renaud Lambiotte. https://arxiv.org/abs/2307.04613

24.4 xgi.convert.graph

Method for projecting a hypergraph to a graph.

Functions

xgi.convert.graph.**to_graph**(*H*)

Graph projection (1-skeleton) of the hypergraph H. Weights are not considered.

Parameters
H (*Hypergraph object*) – The hypergraph of interest

Returns
G – The graph projection

Return type
networkx.Graph

24.5 xgi.convert.higher_order_network

Methods for converting to higher-order network objects.

Functions

xgi.convert.higher_order_network.**to_hypergraph**(*data*, *create_using=None*)

Make a hypergraph from a known data structure.

The preferred way to call this is automatically from the class constructor.

Parameters

- **data** (*object to be converted*) –

  Current known types are:

  - a Hypergraph object
  - a SimplicialComplex object
  - list-of-iterables
  - dict-of-iterables
  - Pandas DataFrame (bipartite edgelist)
  - numpy matrix
  - numpy ndarray
  - scipy sparse matrix

- **create_using** (*Hypergraph constructor, optional (default=Hypergraph)*) – Hypergraph type to create. If hypergraph instance, then cleared before populated.

Returns
A hypergraph constructed from the data

Return type
Hypergraph object

See also:

from_max_simplices
Constructs a hypergraph from the maximal simplices of a simplicial complex.

xgi.convert.higher_order_network.**to_dihypergraph**(*data*, *create_using=None*)

Make a dihypergraph from a known data structure.

The preferred way to call this is automatically from the class constructor.

Parameters

- **data** (*object to be converted*) –

  Current known types are:

  - a DiHypergraph object
  - a SimplicialComplex object
  - list-of-iterables
  - dict-of-iterables
  - Pandas DataFrame (bipartite edgelist)
  - numpy matrix
  - numpy ndarray

- scipy sparse matrix

- **create_using** (*Hypergraph constructor, optional (default=Hypergraph)*) – Hypergraph type to create. If hypergraph instance, then cleared before populated.

Returns
A hypergraph constructed from the data

Return type
Hypergraph object

xgi.convert.higher_order_network.**to_simplicial_complex**(*data*, *create_using=None*)

Make a hypergraph from a known data structure. The preferred way to call this is automatically from the class constructor.

Parameters

- **data** (*object to be converted*) –

  Current known types are:

  - a SimplicialComplex object
  - a Hypergraph object
  - list-of-iterables
  - dict-of-iterables
  - Pandas DataFrame (bipartite edgelist)
  - numpy matrix
  - numpy ndarray
  - scipy sparse matrix

- **create_using** (*Hypergraph graph constructor, optional (default=Hypergraph)*) – Hypergraph type to create. If hypergraph instance, then cleared before populated.

Returns
A hypergraph constructed from the data

Return type
Hypergraph object

24.6 xgi.convert.hyperedges

Methods for converting to and from hyperedge lists.

Functions

xgi.convert.hyperedges.**from_hyperedge_dict**(*d*, *create_using=None*)

Creates a hypergraph from a dictionary of hyperedges

Parameters

- **d** (*dict*) – A dictionary where the keys are edge IDs and the values are containers of nodes specifying the edges.
- **create_using** (*Hypergraph constructor, optional*) – The hypergraph object to add the data to, by default None

Returns
The constructed hypergraph object

Return type
Hypergraph object

See also:

to_hyperedge_dict

xgi.convert.hyperedges.**to_hyperedge_dict**(*H*)

Outputs a hyperedge dictionary

Parameters
H (*Hypergraph object*) – The hypergraph of interest

Returns
A dictionary where the keys are edge IDs and the values are sets of nodes specifying the edges.

Return type
dict

See also:

from_hyperedge_dict

xgi.convert.hyperedges.**from_hyperedge_list**(*d*, *create_using=None*, *max_order=None*)

Generate a hypergraph from a list of lists.

Parameters

- **d** (*list of iterables*) – A hyperedge list
- **create_using** (*Hypergraph constructor, optional*) – The hypergraph to add the edges to, by default None

Returns
The constructed hypergraph object

Return type
Hypergraph object

See also:

to_hyperedge_list

xgi.convert.hyperedges.**to_hyperedge_list**(*H*)

Generate a hyperedge list from a hypergraph.

Parameters
H (*Hypergraph object*) – The hypergraph of interest

Returns
The hyperedge list

Return type
list of sets

See also:

from_hyperedge_list

24.7 xgi.convert.hypergraph_dict

Method for converting from a standardized dictionary.

Functions

xgi.convert.hypergraph_dict.**dict_to_hypergraph**(*data*, *nodetype=None*, *edgetype=None*, *max_order=None*)

A function to read a file in a standardized JSON format.

Parameters

- **data** (*dict*) – A dictionary in the hypergraph JSON format
- **nodetype** (*type, optional*) – Type that the node IDs will be cast to
- **edgetype** (*type, optional*) – Type that the edge IDs will be cast to
- **max_order** (*int, optional*) – Maximum order of edges to add to the hypergraph

Returns
The loaded hypergraph

Return type
A Hypergraph object

Raises
XGIError – If the JSON is not in a format that can be loaded.

See also:

read_json

24.8 xgi.convert.incidence

Methods for converting to and from an incidence matrix.

Functions

xgi.convert.incidence.**from_incidence_matrix**(*d*, *create_using=None*, *nodelabels=None*, *edgelabels=None*)

Create a hypergraph from an incidence matrix

Parameters

- **d** (*numpy array or a scipy sparse arrary*) – The incidence matrix where rows specify nodes and columns specify edges.
- **create_using** (*Hypergraph constructor, optional*) – The hypergraph object to add the data to, by default None
- **nodelabels** (*list or 1D numpy array, optional*) – The ordered list of node IDs to map the indices of the incidence matrix to, by default None
- **edgelabels** (*list or 1D numpy array, optional*) – The ordered list of edge IDs to map the indices of the incidence matrix to, by default None

Returns

The constructed hypergraph

Return type

Hypergraph object

Raises

XGIError – Raises an error if the specified labels are the wrong dimensions

See also:

incidence_matrix, *to_incidence_matrix*

xgi.convert.incidence.**to_incidence_matrix**(*H*, *sparse=True*, *index=False*)

Convert a hypergraph to an incidence matrix.

Parameters

- **H** (*Hypergraph object*) – The hypergraph of interest
- **sparse** (*bool, optional*) – Whether the constructed incidence matrix should be sparse, by default True
- **index** (*bool, optional*) – Whether to return the corresponding node and edge labels, by default False

Returns

- *numpy.ndarray or scipy csr_array* – The incidence matrix
- *dict* – The dictionary mapping indices to node IDs, if index is True
- *dict* – The dictionary mapping indices to edge IDs, if index is True

See also:

incidence_matrix, *from_incidence_matrix*

24.9 xgi.convert.line_graph

Method for converting to a line graph.

Functions

xgi.convert.line_graph.**to_line_graph**(*H*, *s=1*, *weights=None*)

The s-line graph of the hypergraph.

The s-line graph of the hypergraph H is the graph whose nodes correspond to each hyperedge in H, linked together if they share at least s vertices.

Optional edge weights correspond to the size of the intersection between the hyperedges, optionally normalized by the size of the smaller hyperedge.

Parameters

- **H** (Hypergraph) – The hypergraph of interest
- **s** (*int*) – The intersection size to consider edges as connected, by default 1.
- **weights** (*str or None*) – Specify whether to return a weighted line graph. If None, returns an unweighted line graph. If 'absolute', includes edge weights corresponding to the size of intersection between hyperedges. If 'normalized', includes edge weights normalized by the size of the smaller hyperedge.

Returns

LG – The line graph associated to the Hypergraph

Return type

networkx.Graph

References

"Hypernetwork science via high-order hypergraph walks", by Sinan G. Aksoy, Cliff Joslyn, Carlos Ortiz Marrero, Brenda Praggastis & Emilie Purvine. https://doi.org/10.1140/epjds/s13688-020-00231-0

24.10 xgi.convert.pandas

Methods for converting to and from a Pandas dataframe.

Functions

xgi.convert.pandas.**from_bipartite_pandas_dataframe**(*df*, *create_using=None*, *node_column=0*, *edge_column=1*)

Create a hypergraph from a pandas dataframe given specified node and edge columns.

Parameters

- **df** (*Pandas dataframe*) – A dataframe where specified columns list the node IDs and the associated edge IDs
- **create_using** (*Hypergraph constructor, optional*) – The hypergraph object to add the data to, by default None

- **node_column** (*hashable, optional*) – The column with the node IDs, by default 0 Can specify names or indices
- **edge_column** (*hashable, optional*) – The column with the edge IDs, by default 1 Can specify names or indices

Returns
The constructed hypergraph

Return type
Hypergraph object

Raises
XGIError – Raises an error if the user specifies invalid column names

xgi.convert.pandas.**to_bipartite_pandas_dataframe**(*H*)

Create a two column dataframe from a hypergraph.

Parameters
H (Hypergraph *or Simplicial Complex*) – A dataframe where specified columns list the node IDs and the associated edge IDs

Returns
A two column dataframe

Return type
Pandas Dataframe object

Raises
XGIError – Raises an error if the user specifies invalid column names

24.11 xgi.convert.simplex

Methods for simplicial complexes.

Functions

xgi.convert.simplex.**from_max_simplices**(*SC*)

Returns a hypergraph constructed from the maximal simplices of the provided simplicial complex.

Parameters
SC (SimplicialComplex) –

Return type
Hypergraph

xgi.convert.simplex.**from_simplex_dict**(*d*, *create_using=None*)

Creates a Simplicial Complex from a dictionary of simplices, if the subfaces of existing simplices are not given in the dict then the function adds them with integer IDs.

Parameters

- **d** (*dict*) – A dictionary where the keys are simplex IDs and the values are containers of nodes specifying the simplices.
- **create_using** (*SimplicialComplex constructor, optional*) – The simplicial complex object to add the data to, by default None

Returns
The constructed simplicial complex object

Return type
SimplicialComplex object

CHAPTER

TWENTYFIVE

UTILS PACKAGE

Modules

utilities	General utilities.

25.1 xgi.utils.utilities

General utilities.

Classes

IDDict	A dict that holds (node or edge) IDs.

25.1.1 xgi.utils.utilities.IDDict

class xgi.utils.utilities.**IDDict**

Bases: `dict`

A dict that holds (node or edge) IDs.

For internal use only. Adds input validation functionality to the internal dicts that hold nodes and edges in a network.

Methods

Functions

xgi.utils.utilities.**dual_dict**(*edge_dict*)

Given a dictionary with IDs as keys and sets as values, return the dual.

Parameters
edge_dict (*dict*) – A dictionary where the keys are IDs and the values are sets of hashables

Returns
A dictionary with IDs as keys and sets as values, but the reverse of the original dict.

Return type
dict

Examples

```
>>> import xgi
>>> xgi.dual_dict({0 : [1, 2, 3], 1 : [0, 2]})
{1: {0}, 2: {0, 1}, 3: {0}, 0: {1}}
```

xgi.utils.utilities.**powerset**(*iterable*, *include_empty=False*, *include_full=False*, *include_singletons=True*, *max_size=None*)

Returns all possible subsets of the elements in iterable, with options to include the empty set and the set containing all elements, and to set the maximum subset size.

Parameters

- **iterable** (*list-like*) – List of elements
- **include_empty** (*bool, default: False*) – Whether to include the empty set
- **include_singletons** (*bool, default: True*) – Whether to include singletons
- **include_full** (*bool, default: False*) – Whether to include the set containing all elements of iterable
- **max_size** (*int, default: None*) – Maximum size of the returned subsets.

Return type
itertools.chain

Notes

include_empty overrides include_singletons if True: singletons will always be included if the empty set is. Likewise, max_size will override other arguments: if set to -1, no subset will be returned.

Examples

```
>>> import xgi
>>> list(xgi.powerset([1,2,3,4]))
[(1,), (2,), (3,), (4,), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4),
 (1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
```

xgi.utils.utilities.**update_uid_counter**(*H*, *new_id*)

Helper function to make sure the uid counter is set correctly after adding an edge with a user-provided ID.

If we don't set the start of self._edge_uid correctly, it will start at 0, which will overwrite any existing edges when calling add_edge(). First, we use the somewhat convoluted float(e).is_integer() instead of using isinstance(e, int) because there exist integer-like numeric types (such as np.int32) which fail the isinstance() check.

Parameters

- **H** (*xgi.Hypergraph*) – Hypergraph of which to update the uid counter
- **id** (*any hashable type*) – User-provided ID.

xgi.utils.utilities.**find_triangles**(*G*)

Returns list of 3-node cliques present in a graph

Parameters
G (*networkx Graph*) – Graph to consider

Return type
list of 3-node cliques (triangles)

xgi.utils.utilities.**min_where**(*dicty*, *where*)

Finds the minimum value of a dictonary *dicty*. The dictonary *where* indicates which keys to take into account. The minimum is eventualy infinite.

Parameters

- **dicty** (*dict*) – Dictionary of values (int, float...) from which to find the minimum.
- **where** (*dict*) – Dictionary of booleans that has the same keys as *dicty*. The minimum will be searched among the values for which *where[key]* is TRUE.

Returns
min_val – Minimum value found in *dicty*. Is set to np.infty if *where* indicated nowhere or if all values are *np.infty*.

Return type
float or np.Inf

xgi.utils.utilities.**request_json_from_url**(*url*)

HTTP request json file and return as dict.

Parameters
url (*str*) – The url where the json file is located.

Returns
A dictionary of the JSON requested.

Return type
dict

Raises
XGIError – If the connection fails or if there is a bad HTTP request.

xgi.utils.utilities.**request_json_from_url_cached**(*url*)

HTTP request json file and return as dict.

Parameters
url (*str*) – The url where the json file is located.

Returns
A dictionary of the JSON requested.

Return type
dict

Raises
XGIError – If the connection fails or if there is a bad HTTP request.

xgi.utils.utilities.**subfaces**(*edges*, *order=None*)

Returns the subfaces of a list of hyperedges

Parameters

- **edges** (*list of edges*) – Edges to consider, as tuples of nodes
- **order** (*{None, -1, int}, optional*) – If None, compute subfaces recursively down to nodes. If -1, compute subfaces the order below (e.g. edges for a triangle). If d > 0, compute the subfaces of order d. By default, None.

Returns
faces – List of hyperedges that are subfaces of input hyperedges.

Return type
list of sets

Raises
XGIError – Raises error when order is larger than the max order of input edges

Notes

Hyperedges in the returned list are not unique, they may appear more than once if they are subfaces or more than one edge from the input edges.

Examples

```
>>> import xgi
>>> edges = [{1,2,3,4}, {3,4,5}]
>>> xgi.subfaces(edges)
[(1,), (2,), (3,), (4,), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4), (1, 2, 3),
 (1, 2, 4), (1, 3, 4), (2, 3, 4), (3,), (4,), (5,), (3, 4), (3, 5), (4, 5)]
>>> xgi.subfaces(edges, order=-1)
[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4), (3, 4), (3, 5), (4, 5)]
>>> xgi.subfaces(edges, order=2)
[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4), (3, 4, 5)]
```

xgi.utils.utilities.**convert_labels_to_integers**(*net*, *label_attribute='label'*)

Relabel node and edge IDs to be sequential integers.

Parameters

- **net** (Hypergraph, DiHypergraph, *or* SimplicialComplex) – The higher-order network of interest
- **label_attribute** (*string, default: "label"*) – The attribute name that stores the old node and edge labels

Returns
A new higher-order network with nodes and edges with sequential IDs starting at 0. The old IDs are stored in the "label" attribute for both nodes and edges.

Return type
Hypergraph, *DiHypergraph*, or *SimplicialComplex*

Notes

The "relabeling" will occur even if the node/edge IDs are sequential. Because the old IDs are stored in the "label" attribute for both nodes and edges, the old "label" values (if they exist) will be overwritten.

`xgi.utils.utilities.hist`(*vals*, *bins=10*, *bin_edges=False*, *density=False*, *log_binning=False*)

Return the distribution of a numpy array.

Parameters

- **vals** (*Numpy array*) – The array of values
- **bins** (*int, list, or Numpy array*) – The number of bins or the bin edges. By default, 10.
- **bin_edges** (*bool*) – Whether to also output the min and max of each bin, by default, False.
- **density** (*bool*) – Whether to normalize the resulting distribution. By default, False.
- **log_binning** (*bool*) – Whether to bin the values with log-sized bins. By default, False.

Returns
A two-column table with "bin_center" and "value" columns, where "value" is a count or a probability. If *bin_edges* is True, outputs two additional columns, *bin_lo* and *bin_hi*, which outputs the left and right bin edges respectively.

Return type
Pandas DataFrame

Notes

Originally from https://github.com/jkbren/networks-and-dataviz

CHAPTER

TWENTYSIX

ABOUT

The Comple**X** **G**roup **I**nteractions (XGI) library provides data structures and algorithms for modeling and analyzing complex systems with group (higher-order) interactions. For more information about what higher-order interactions are, see a brief overview.

Many datasets can be represented as graphs, where pairs of entities (or nodes) are related via links (or edges). Examples are road networks, energy grids, social networks, neural networks, etc. However, in many other datasets, more than two entities can be related at a time. For example, many scientists (entities) can collaborate on a scientific article together (links), and multiple email accounts (entities) can all participate on the same email thread (links). In this latter case, graphs no longer present a viable alternative to represent such datasets. It is for this kind of datasets, where the interactions are given among groups of more than two entities (also called higher-order interactions), that XGI was designed for.

XGI is implemented in pure Python and is designed to seamlessly interoperate with the rest of the Python scientific stack (numpy, scipy, pandas, matplotlib, etc). XGI is designed and developed by network scientists with the needs of network scientists in mind.

- Repository: https://github.com/xgi-org/xgi
- PyPI: latest release
- Twitter: @xginets
- List of Contributors
- Projects Using XGI

Sign up for our mailing list and follow XGI on Twitter or Mastodon!

CHAPTER

TWENTYSEVEN

INSTALLATION

To install and use XGI as an end user, execute

```
pip install xgi
```

To install for development purposes, first clone the repository and then execute

```
pip install -e .['all']
```

If that command does not work, you may try the following instead

```
pip install -e .\[all\]
```

XGI was developed and tested for Python 3.8-3.12 on Mac OS, Windows, and Ubuntu.

CHAPTER

TWENTYEIGHT

CORRESPONDING DATA

A number of higher-order datasets are available in the XGI-DATA repository and can be easily accessed with the `load_xgi_data()` function.

CHAPTER

TWENTYNINE

CONTRIBUTING

If you want to contribute to this project, please make sure to read the contributing guidelines. We expect respectful and kind interactions by all contributors and users as laid out in our code of conduct.

The XGI community always welcomes contributions, no matter how small. We're happy to help troubleshoot XGI issues you run into, assist you if you would like to add functionality or fixes to the codebase, or answer any questions you may have.

Some concrete ways that you can get involved:

- **Get XGI updates** by following the XGI Twitter account, signing up for our mailing list, or starring this repository.
- **Spread the word** when you use XGI by sharing with your colleagues and friends.
- **Request a new feature or report a bug** by raising a new issue.
- **Create a Pull Request (PR)** to address an open issue or add a feature.
- **Join our Zulip channel** to be a part of the daily goings-on of XGI.

CHAPTER

THIRTY

HOW TO CITE

We acknowledge the importance of good software to support research, and we note that research becomes more valuable when it is communicated effectively. To demonstrate the value of XGI, we ask that you cite the XGI paper in your work. You can cite XGI either by going to our repository page repository page and clicking the "cite this repository" button on the right sidebar (which will generate a citation in your preferred format) or by copying the following BibTeX entry:

```
@article{Landry_XGI_2023,
author = {Landry, Nicholas W. and Lucas, Maxime and Iacopini, Iacopo and Petri, Giovanni␣
↪and Schwarze, Alice and Patania, Alice and Torres, Leo},
title = {{XGI: A Python package for higher-order interaction networks}},
doi = {10.21105/joss.05162},
journal = {Journal of Open Source Software},
publisher = {The Open Journal},
year = {2023},
month = may,
volume = {8},
number = {85},
pages = {5162},
url = {https://doi.org/10.21105/joss.05162},
}
```

CHAPTER

THIRTYONE

ACADEMIC REFERENCES

- The Why, How, and When of Representations for Complex Systems, Leo Torres, Ann S. Blevins, Danielle Bassett, and Tina Eliassi-Rad.
- Networks beyond pairwise interactions: Structure and dynamics, Federico Battiston, Giulia Cencetti, Iacopo Iacopini, Vito Latora, Maxime Lucas, Alice Patania, Jean-Gabriel Young, and Giovanni Petri.
- What are higher-order networks?, Christian Bick, Elizabeth Gross, Heather A. Harrington, Michael T. Schaub.
- From networks to optimal higher-order models of complex systems, Renaud Lambiotte, Martin Rosvall, and Ingo Scholtes.

CHAPTER

THIRTYTWO

FUNDING

The XGI package has been supported by NSF Grant 2121905, HNDS-I: Using Hypergraphs to Study Spreading Processes in Complex Social Networks.

CHAPTER

THIRTYTHREE

LICENSE

PYTHON MODULE INDEX

X

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X