XGI Documentation

Release 0.6

May 15, 2023

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.

- Repository: https://github.com/xgi-org/xgi
- PyPI: https://pypi.org/project/xgi/
- Documentation: https://xgi.readthedocs.io/

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 3.8-3.11 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 code of conduct and the contributing guidelines.

The best way to contribute to XGI is by submitting a bug or request a new feature by opening a new issue.

To get more actively involved, you are invited to browse the issues page and choose one that you can work on. The core developers will be happy to help you understand the codebase and any other doubts you may have while working on your contribution.

If you are interested in the daily goings-on of XGI, you are invited to join our Zulip channel.

CHAPTER

FOUR

CONTRIBUTORS

The core XGI team members:

- Nicholas Landry
- Leo Torres
- Maxime Lucas
- Iacopo Iacopini
- Giovanni Petri
- Alice Patania
- Alice Schwarze

Other contributors:

- Martina Contisciani
- Tim LaRock
- Sabina Adhikari
- Marco Nurisso

CHAPTER

FIVE

FUNDING

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

CHAPTER

SIX

LICENSE

This project is licensed under the BSD 3-Clause License.

CHAPTER

SEVEN

XGI.CLASSES.HYPERGRAPH.HYPERGRAPH

class xgi.classes.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.

Multiedges and self-loops 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)
  - Scipy/Numpy incidence matrix
  - Hypergraph 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.

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.
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.
cleanup	Removes potentially undesirable artifacts from the hypergraph.

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, *get_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, *get_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*, *get_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*, *get_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*, *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.
- **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 attribues, 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.

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)
>>> xgi.set_edge_attributes(H, 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")
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

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

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

EIGHT

XGI.CLASSES.SIMPLICIALCOMPLEX.SIMPLICIALCOMPLEX

class xgi.classes.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)
  - Scipy/Numpy incidence matrix
  - SimplicialComplex 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.

Methods

add_edge	Deprecated in SimplicialComplex.
add_edges_from	Deprecated in SimplicialComplex.
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_edges_from	Deprecated in SimplicialComplex.
add_weighted_simplices_from	Add weighted simplices in *ebunch_to_add* with specified weight attr
close	Adds all missing subfaces to the complex.
has_simplex	Whether a simplex appears in the simplicial complex.

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.
remove_simplex_id	Remove a simplex with a given id.
remove_simplex_ids_from	Remove all simplicies specified in ebunch.

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})]
>>> 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. Attribues 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}
```

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.

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
>>> H = xgi.SimplicialComplex([[1, 2], [2, 3, 4]])
>>> H.has_simplex([1, 2])
True
>>> H.has_simplex({1, 3})
False
```

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*, *strong=False*)

remove_node is not implemented in SimplicialComplex.

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

NINE

CLASSES PACKAGE

Modules

hypergraph	Base class for undirected hypergraphs.
simplicialcomplex	Base class for undirected simplicial complexes.
reportviews	View classes for hypergraphs.
hypergraphviews	View of Hypergraphs as a subhypergraph or read-only.
function	Functional interface to hypergraph methods and assorted utilities.

9.1 xgi.classes.hypergraph

Base class for undirected hypergraphs.

Classes

Hypergraph	A hypergraph is a collection of subsets of a set of *nodes* or *vertices*.

9.2 xgi.classes.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.

9.3 xgi.classes.reportviews

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, edges, degree, and edge size 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.

9.3.1 xgi.classes.reportviews.IDView

class xgi.classes.reportviews.**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

- **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.
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*) – *NodeStat* object, or name of a *NodeStat*.
- **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, 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*.

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))
>>> 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, optional*) – Comparison mode. Valid options are 'eq' (default), 'neq', 'lt', 'gt', 'leq', 'geq', or 'between'.
- **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 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
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*)

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.

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}
```

9.3.2 xgi.classes.reportviews.NodeView

class xgi.classes.reportviews.**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.

9.3.3 xgi.classes.reportviews.EdgeView

class xgi.classes.reportviews.**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()

9.4 xgi.classes.hypergraphviews

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 hypergraphviews 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.classes.hypergraphviews.**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

9.5 xgi.classes.function

Functional interface to hypergraph methods and assorted utilities.

Functions

xgi.classes.function.**convert_labels_to_integers**(*H*, *label_attribute='label'*)

Relabel node and edge IDs to be sequential integers.

Parameters

- **H** (Hypergraph) – The hypergraph of interest
- **label_attribute** (*string, default: "label"*) – The attribute name that stores the old node and edge labels

Returns
A new hypergraph 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

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.classes.function.**create_empty_copy**(*H*, *with_data=True*)

Create a new hypergraph with the nodes (and data) of a specified hypergraph.

Parameters

- **H** (*Hypergraph object*) – The hypergraph to copy

- **with_data** (*bool, optional*) – Whether to keep the node and hypergraph data, by default True.

Returns
A hypergraph with the same nodes but without edges

Return type
Hypergraph object

See also:

is_empty, empty_hypergraph

Examples

```
>>> import xgi
>>> hyperedge_list = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(hyperedge_list)
>>> H_copy = xgi.create_empty_copy(H)
>>> H_copy.nodes
NodeView((1, 2, 3, 4))
>>> H_copy.edges
EdgeView(())
```

xgi.classes.function.**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.classes.function.**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.classes.function.**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.classes.function.**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}]
```

(continues on next page)

(continued from previous page)

```
>>> xgi.edge_neighborhood(H, 3, include_self=True)
[{1, 2, 3}, {3, 4}]
```

xgi.classes.function.**freeze**(*H*)

Method for freezing a hypergraph which prevents it from being modified

Parameters
H (*Hypergraph object*) – The hypergraph to freeze

Returns
The hypergraph with all the functions that can modify the hypergraph set to the frozen method

Return type
Hypergraph object

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
>>> hyperedge_list = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(hyperedge_list)
>>> xgi.freeze(H)
<xgi.classes.hypergraph.Hypergraph object at 0x...>
>>> H.add_node(5)
Traceback (most recent call last):
xgi.exception.XGIError: Frozen hypergraph can't be modified
```

xgi.classes.function.**frozen**(**args*, ***kwargs*)

Dummy method that raises an error when trying to modify frozen hypergraphs

Raises
XGIError – Raises error when user tries to modify the hypergraph

Examples

```
>>> import xgi
>>> hyperedge_list = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(hyperedge_list)
>>> xgi.freeze(H)
<xgi.classes.hypergraph.Hypergraph object at 0x...>
>>> H.add_node(5)
Traceback (most recent call last):
xgi.exception.XGIError: Frozen hypergraph can't be modified
```

xgi.classes.function.**get_edge_attributes**(*H*, *name=None*)

Get the edge attributes of the hypergraph

Parameters

- **H** (*Hypergraph object*) – The hypergraph to get edge attributes from
- **name** (*string, optional*) – Attribute name. If None (default), then return the entire attribute dictionary.

Returns
Dictionary of attributes keyed by edge ID.

Return type
dict

See also:

set_node_attributes, *get_node_attributes*, *set_edge_attributes*

xgi.classes.function.**get_node_attributes**(*H*, *name=None*)

Get the node attributes for a hypergraph

Parameters

- **H** (*Hypergraph object*) – The hypergraph to get node attributes from
- **name** (*string, optional*) – Attribute name. If None, then return the entire attribute dictionary.

Returns
Dictionary of attributes keyed by node.

Return type
dict of dict

See also:

set_node_attributes, *get_edge_attributes*

xgi.classes.function.**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 nad 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.classes.function.**is_empty**(*H*)

Returns True if *H* has no edges.

Parameters
H (*Hypergraph object*) – Hypergraph of interest

Returns
True if *H* has no edges, and False otherwise.

Return type
bool

See also:

create_empty_copy, empty_hypergraph

Examples

```
>>> import xgi
>>> hyperedge_list = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(hyperedge_list)
>>> xgi.is_empty(H)
False
```

xgi.classes.function.**is_frozen**(*H*)

Checks whether a hypergraph is frozen

Parameters
H (*Hypergraph object*) – The hypergraph to check

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
>>> hyperedge_list = [[1, 2], [2, 3, 4]]
>>> H = xgi.Hypergraph(hyperedge_list)
>>> xgi.freeze(H)
<xgi.classes.hypergraph.Hypergraph object at 0x...>
>>> xgi.is_frozen(H)
True
```

xgi.classes.function.**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.classes.function.**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.classes.function.**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.classes.function.**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.classes.function.**set_edge_attributes**(*H*, *values*, *name=None*)

Set the edge attributes from a value or a dictionary of values.

Parameters

- **H** (*Hypergraph object*) – The hypergraph to set edge attributes
- **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, *get_edge_attributes*, *add_edge*, *add_edges_from*

Notes

Note that if the dict contains edge IDs that are not in *H*, they are silently ignored.

xgi.classes.function.**set_node_attributes**(*H*, *values*, *name=None*)

Sets node attributes from a given value or dictionary of values.

Parameters

- **H** (*Hypergraph object*) – The hypergraph to set node attributes
- **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:

get_node_attributes, *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.

xgi.classes.function.**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)]
```

(continues on next page)

(continued from previous page)

```
>>> xgi.subfaces(edges, order=2)
[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4), (3, 4, 5)]
```

xgi.classes.function.**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

TEN

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.

Modules

nodestats	Node statistics.
edgestats	Edge statistics.

10.1 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}
```

10.2 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},
 '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

Classes

NodeStat	An arbitrary node-quantity mapping.
EdgeStat	An arbitrary edge-quantity mapping.
MultiNodeStat	Multiple node-quantity mappings.
MultiEdgeStat	Multiple edge-quantity mappings.

10.3 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.
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.

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.

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.

10.4 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.
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.

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.

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.

10.5 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.

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}}
```

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/python3.9/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.

10.6 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.

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}}
```

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/python3.9/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.

Decorators

nodestat_func	Decorate arbitrary functions to behave like *NodeStat* objects.
edgestat_func	Decorate arbitrary functions to behave like *EdgeStat* objects.

10.7 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
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}
```

10.8 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()

CHAPTER

ELEVEN

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	
connected	Algorithms related to connected components of a hypergraph.

11.1 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

11.2 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, x^2.
- **g** (*lambda function, optional*) – The function g as described in Tudisco and Higham. Must accept a numpy array. By default, $sqrt{x}$.
- **phi** (*lambda function, optional*) – The function phi as described in Tudisco and Higham. Must accept a numpy array. By default x^2.
- **psi** (*lambda function, optional*) – The function psi as described in Tudisco and Higham. Must accept a numpy array. By default: $sqrt{x}$.

- **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. :param H: The hypergraph of interest :type H: Hypergraph

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

11.3 xgi.algorithms.clustering

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., *num / denom*, where *num* equals $A^3_{i,i}$ and *denom* equals $inom{k}{2}$. Here 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}
```

11.4 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
```

CHAPTER

TWELVE

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.

12.1 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* node 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.

12.2 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.

12.3 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.

12.4 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*)

12.5 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*, *rho*, *k*, *epsilon*, *seed=None*)

Construct the m-uniform hypergraph planted partition model (m-HPPM)

Parameters

- **n** (*int > 0*) – Number of nodes
- **m** (*int > 0*) – Hyperedge size
- **rho** (*float between 0 and 1*) – The fraction of nodes in community 1
- **k** (*float > 0*) – Mean degree
- **epsilon** (*float > 0*) – Imbalance parameter
- **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

12.6 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])
```

CHAPTER

THIRTEEN

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.

13.1 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.
- **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

13.2 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],
       [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.**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)

13.3 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],
       [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

FOURTEEN

READWRITE PACKAGE

Modules

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.

14.1 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=",")
```

14.2 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=",")
```

14.3 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=",")
```

14.4 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

14.5 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, default: None*) – Dataset name. Valid options are the top-level tags of the index.json file in the xgi-data repository. If None, prints the list of available datasets.
- **cache** (*bool, optional*) – Whether to cache the input data
- **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.
- **path** (*str, optional*) – Path to a local copy of the data set
- **nodetype** (*type, optional*) – Type to cast the node ID to
- **edgetype** (*type, optional*) – Type to cast the edge ID to
- **max_order** (*int, optional*) – Maximum order of edges to add to the hypergraph

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

FIFTEEN

DYNAMICS PACKAGE

Modules

synchronization	Simulation of the Kuramoto model.

15.1 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

SIXTEEN

DRAWING PACKAGE

Modules

layout	Algorithms to compute node positions for drawing.
draw(H[, pos, ax, dyad_color, dyad_lw, ...])	Draw hypergraph or simplicial complex.

16.1 xgi.drawing.layout

Algorithms to compute node positions for drawing.

Functions

xgi.drawing.layout.**random_layout**(*H*, *center=None*, *dim=2*, *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.
- **dim** (*int, optional*) – Dimension of layout, by default 2.
- **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*)

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.

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*)

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.

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.**weighted_barycenter_spring_layout**(*H*, *return_phantom_graph=False*, *seed=None*)

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.

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*)

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).

Returns
pos – A dictionary of positions keyed by node

Return type
dict

16.2 xgi.drawing.draw

Draw hypergraphs and simplicial complexes with matplotlib.

Functions

xgi.drawing.draw.**draw**(*H*, *pos=None*, *ax=None*, *dyad_color='black'*, *dyad_lw=1.5*, *edge_fc=None*, *node_fc='white'*, *node_ec='black'*, *node_lw=1*, *node_size=15*, *max_order=None*, *node_labels=False*, *hyperedge_labels=False*, ***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.
- **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.
- **edge_fc** (*str, dict, iterable, or* EdgeStat*, optional*) – Color of the hyperedges. If str, use the same color for all nodes. If a dict, must contain (edge_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 EdgeStat, use the colormap specified with edge_fc_cmap. If None (default), use the H.edges.size.
- **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.
- **max_order** (*int, optional*) – Maximum of hyperedges to plot. If None (default), plots all orders.
- **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.
- ****kwargs** (*optional args*) – Alternate default values. Values that can be overwritten are the following: * min_node_size * max_node_size * min_node_lw * max_node_lw * min_dyad_lw * max_dyad_lw * node_fc_cmap * node_ec_cmap * dyad_color_cmap * edge_fc_cmap

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_hypergraph_hull**(*H*, *pos=None*, *ax=None*, *dyad_color='black'*, *edge_fc=None*, *edge_ec=None*, *node_fc='tab:blue'*, *node_ec='black'*, *node_lw=1*, *node_size=7*, *max_order=None*, *node_labels=False*, *hyperedge_labels=False*, *radius=0.05*, ***kwargs*)

Draw hypergraphs displaying the hyperedges of order k>1 as convex hulls

Parameters

- **H** (Hypergraph) –
- **pos** (*dict, optional*) – If passed, this dictionary of positions node_id:(x,y) is used for placing the nodes. 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".
- **edge_fc** (*str, dict, iterable, or* EdgeStat*, optional*) – Color of the hyperedges of order k>1. If str, use the same color for all hyperedges of order k>1. If a dict, must contain (edge_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 EdgeStat, use the colormap specified with edge_fc_cmap. If None (default), use the H.edges.size.
- **edge_ec** (*str, dict, iterable, or* EdgeStat*, optional*) – Color of the borders of the hyperdges of order k>1. 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 edge_ec_cmap) associated to it. If None (default), use the H.edges.size.
- **node_fc** (*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, "tab:blue".
- **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* EdgeStat*, 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.
- **max_order** (*int, optional*) – Maximum of hyperedges to plot. If None (default), plots all orders.
- **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.
- **radius** (*float, optional*) – Radius of the convex hull in the vicinity of the nodes, by default 0.05.
- ****kwargs** (*optional args*) – Alternate default values. Values that can be overwritten are the following: * min_node_size * max_node_size * min_node_lw * max_node_lw * node_fc_cmap * node_ec_cmap * dyad_color_cmap * edge_fc_cmap * edge_ec_cmap * alpha

Returns
ax

Return type
matplotlib.pyplot.axes

See also:

draw

xgi.drawing.draw.**draw_nodes**(*H*, *pos=None*, *ax=None*, *node_fc='white'*, *node_ec='black'*, *node_lw=1*, *node_size=15*, *zorder=None*, *settings=None*, *node_labels=False*, ***kwargs*)

Draw the nodes of a hypergraph

Parameters

- **H** (Hypergraph *or* SimplicialComplex) – Higher-order network to plot.
- **ax** (*matplotlib.pyplot.axes, optional*) – Axis to draw on. If None (default), get the current axes.
- **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.
- **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* EdgeStat*, 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.
- **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.
- **settings** (*dict*) – Default parameters. Keys that may be useful to override default settings: * min_node_size * max_node_size * min_node_lw * max_node_lw * node_fc_cmap * node_ec_cmap
- **kwargs** (*optional keywords*) – See *draw_node_labels* for a description of optional keywords.

See also:

draw, *draw_hyperedges*, *draw_simplices*, *draw_node_labels*, *draw_hyperedge_labels*

xgi.drawing.draw.**draw_hyperedges**(*H*, *pos=None*, *ax=None*, *dyad_color='black'*, *dyad_lw=1.5*, *edge_fc=None*, *max_order=None*, *settings=None*, *hyperedge_labels=False*, ***kwargs*)

Draw hyperedges.

Parameters

- **H** (Hypergraph) –
- **ax** (*matplotlib.pyplot.axes, optional*) – Axis to draw on. If None (default), get the current axes.
- **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.
- **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.
- **edge_fc** (*str, dict, iterable, or* EdgeStat*, optional*) – Color of the hyperedges. If str, use the same color for all nodes. If a dict, must contain (edge_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 EdgeStat, use the colormap specified with edge_fc_cmap. If None (default), color by edge size.

- **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.
- **settings** (*dict*) – Default parameters. Keys that may be useful to override default settings: * min_dyad_lw * max_dyad_lw * dyad_color_cmap * edge_fc_cmap
- **kwargs** (*optional keywords*) – See *draw_hyperedge_labels* for a description of optional keywords.

Raises

XGIError – If a SimplicialComplex is passed.

See also:

draw, *draw_nodes*, *draw_simplices*, *draw_node_labels*, *draw_hyperedge_labels*

xgi.drawing.draw.**draw_simplices**(*SC*, *pos=None*, *ax=None*, *dyad_color='black'*, *dyad_lw=1.5*, *edge_fc=None*, *max_order=None*, *settings=None*, *hyperedge_labels=False*, ***kwargs*)

Draw maximal simplices and pairwise faces.

Parameters

- **SC** (SimplicialComplex) – Simplicial complex to draw
- **ax** (*matplotlib.pyplot.axes, optional*) – Axis to draw on. If None (default), get the current axes.
- **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.
- **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.
- **edge_fc** (*str, dict, iterable, or* EdgeStat*, optional*) – Color of the hyperedges. If str, use the same color for all nodes. If a dict, must contain (edge_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 EdgeStat, use the colormap specified with edge_fc_cmap. If None (default), color by simplex size.
- **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. Note, we plot only the maximal simplices so if you pass a dict be careful to match its keys with the new edge ids in the converted SimplicialComplex. These may differ from the edge ids in the given SC. By default, False.
- **settings** (*dict*) – Default parameters. Keys that may be useful to override default settings: * min_dyad_lw * max_dyad_lw * dyad_color_cmap * edge_fc_cmap

- **kwargs** (*optional keywords*) – See *draw_hyperedge_labels* for a description of optional keywords.

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*)

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.

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

SEVENTEEN

CONVERTING TO AND FROM OTHER DATA FORMATS

xgi.convert.**convert_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

xgi.convert.**convert_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

xgi.convert.**convert_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

xgi.convert.**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`

xgi.convert.**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.**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.**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.**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.**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.**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.**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)
```

xgi.convert.**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

xgi.convert.**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.**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

xgi.convert.**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*

xgi.convert.**to_line_graph**(*H*, *s=1*)

The s-line graph of the hypergraph.

The line graph of the hypergraph H is the graph whose nodes correspond to each hyperedge in H, linked together if they share at least one vertex.

Parameters

- **H** (Hypergraph) – The hypergraph of interest
- **s** (*int*) – The intersection size to consider edges as connected, by default 1.

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

CHAPTER

EIGHTEEN

UTILS PACKAGE

Modules

utilities	General utilities.

18.1 xgi.utils.utilities

General utilities.

Classes

IDDict	A dict that holds (node or edge) IDs.

18.1.1 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*)

Returns all possible subsets of the elements in iterable, with options to include the empty set and the set containing all elements.

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

Return type
itertools.chain

Notes

include_empty overrides include_singletons if True: singletons will always be included if the empty set is.

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)

CHAPTER

NINETEEN

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.

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

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

CHAPTER

TWENTY

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 3.8-3.11 on Mac OS, Windows, and Ubuntu.

CHAPTER

TWENTYONE

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

TWENTYTWO

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

TWENTYTHREE

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 XGI in your work. Currently, the best way to cite XGI is to go to our repository page and click the "cite this repository" button on the right sidebar. This will generate a citation in your preferred format, and will also integrate well with citation managers.

CHAPTER

TWENTYFOUR

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

TWENTYFIVE

CONTRIBUTORS

The core XGI team members:

- Nicholas Landry
- Leo Torres
- Maxime Lucas
- Iacopo Iacopini
- Giovanni Petri
- Alice Patania
- Alice Schwarze

Other contributors:

- Martina Contisciani
- Tim LaRock
- Sabina Adhikari
- Marco Nurisso
- Alexis Arnaudon
- Thomas Robiglio
- Gonzalo Contreras Aso

CHAPTER

TWENTYSIX

FUNDING

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

CHAPTER

TWENTYSEVEN

LICENSE

PYTHON MODULE INDEX

X

INDEX

D

E

F

G

H

N

O

P

R

S

T

U

V

W

X