HyperNetX Documentation

Release 2.3.5

Brenda Praggastis, Dustin Arendt

June 12, 2024
# CONTENTS

1 Why hypergraphs? 3
2 Our community 5
3 Our values 7
4 Contact us 9
5 Contents 11

Python Module Index 157

Index 159
HNX is a Python library for hypergraphs, the natural models for multi-dimensional network data. To get started, try the interactive COLAB tutorials. For a primer on hypergraphs, try this gentle introduction. To see hypergraphs at work in cutting-edge research, see our list of recent publications.
CHAPTER ONE

WHY HYPERGRAPHS?

Like graphs, hypergraphs capture important information about networks and relationships. But hypergraphs do more – they model multi-way relationships, where ordinary graphs only capture two-way relationships. This library serves as a repository of methods and algorithms that have proven useful over years of exploration into what hypergraphs can tell us.

As both vertex adjacency and edge incidence are generalized to be quantities, hypergraph paths and walks have both length and width because of these multiway connections. Most graph metrics have natural generalizations to hypergraphs, but since hypergraphs are basically set systems, they also admit to the powerful tools of algebraic topology, including simplicial complexes and simplicial homology, to study their structure.
OUR COMMUNITY

We have a growing community of users and contributors. For the latest software updates, and to learn about the development team, see the library overview. Have ideas to share? We'd love to hear from you! Our orientation for contributors can help you get started.
Our shared values as software developers guide us in our day-to-day interactions and decision-making. Our open source projects are no exception. Trust, respect, collaboration and transparency are core values we believe should live and breathe within our projects. Our community welcomes participants from around the world with different experiences, unique perspectives, and great ideas to share. See our code of conduct to learn more.
Questions and comments are welcome!

Add discussion topics and comments at
https://github.com/pnnl/HyperNetX/discussions

Contact us directly at
hypernetx@pnnl.gov
5.1 Overview

5.1.1 HyperNetX

The HyperNetX library provides classes and methods for the analysis and visualization of complex network data modeled as hypergraphs. The library generalizes traditional graph metrics.

HypernetX was developed by the Pacific Northwest National Laboratory for the Hypernets project as part of its High Performance Data Analytics (HPDA) program. PNNL is operated by Battelle Memorial Institute under Contract DE-AC05-76RL01830.

- Principal Developer and Designer: Brenda Praggastis
- Development Team: Audun Myers, Mark Bonicillo
- Visualization: Dustin Arendt, Ji Young Yun
- Principal Investigator: Cliff Joslyn
- Program Manager: Brian Kritzstein
- Principal Contributors (Design, Theory, Code): Sinan Aksoy, Dustin Arendt, Mark Bonicillo, Helen Jenne, Cliff Joslyn, Nicholas Landry, Audun Myers, Christopher Potvin, Brenda Praggastis, Emilie Purvine, Greg Roek, Mirah Shi, Francois Theberge, Ji Young Yun

The code in this repository is intended to support researchers modeling data as hypergraphs. We have a growing community of users and contributors. Documentation is available at: https://pnnl.github.io/HyperNetX
HyperNetX 2.3

HyperNetX 2.3 is the latest, stable release. The core library has been refactored to take better advantage of Pandas Dataframes, improve readability and maintainability, address bugs, and make it easier to change. New features have been added, most notably the ability to add and remove edges, nodes, and incidences. Updating is recommended.

Version 2.3 is not backwards compatible. Objects constructed using earlier versions can be imported using their incidence dictionaries and/or property dataframes.

What’s New

1. We’ve added new functionality to Hypergraphs; you can add and remove nodes, edges, and incidences on Hypergraph.
2. Arithmetic operations have also been added to Hypergraph: sum, difference, union, intersection.
3. We’ve also added a new tutorial on basic hypergraph arithmetic operations.
4. Under the hood, the EntitySet has been replaced by HypergraphView, new factory methods have been created to support the Hypergraph constructor, and internal classes such as IncidenceStore and PropertyStore help maintain the structure and attributes of a Hypergraph.

What’s Changed

1. Documentation has received a major update; the Glossary and docstrings of Hypergraph have been updated.
2. HNX now requires Python >=3.10,<4.0.0
3. We’ve upgraded all the underlying core libraries to the latest versions.

5.1.2 COLAB Tutorials

The following tutorials may be run in your browser using Google Colab. Additional tutorials are available on GitHub.

5.1.3 Notice

This material was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the United States Department of Energy, nor Battelle, nor any of their employees, nor any jurisdiction or organization that has cooperated in the development of these materials, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness or any information, apparatus, product, software, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
5.1.4 License

HyperNetX is released under the 3-Clause BSD license (see License).

5.2 Installing HyperNetX

The recommended installation method for most users is to create a virtual environment and install HyperNetX from PyPi.

HyperNetX may be cloned or forked from Github.

5.2.1 Prerequisites

HyperNetX officially supports Python ≥3.10,<4.0.0.

5.2.2 Create a virtual environment

Using Anaconda

```bash
conda create -n venv-hnx python=3.11 -y
conda activate venv-hnx
```

Using venv

```bash
python -m venv venv-hnx
source venv-hnx/bin/activate
```

Using virtualenv

```bash
virtualenv venv-hnx
source venv-hnx/bin/activate
```

For Windows Users

On both Windows PowerShell or Command Prompt, you can use the following command to activate your virtual environment:

```bash
.\env-hnx\Scripts\activate
```

To deactivate your environment, use:

```bash
.\env-hnx\Scripts\deactivate
```
5.2.3 Installation

After activating your virtual environment, install HyperNetX.

**Installing from PyPi**

```
pip install hypernetx
```

**Installing from Source**

The source code provides a Makefile to simplify the installation process. Ensure that you have `make` and `git` installed.

```
git clone https://github.com/pnnl/HyperNetX.git
cd HyperNetX
make venv
source venv-hnx/bin/activate
make install
```

5.2.4 Post-Installation Actions

**Interact with HyperNetX in a REPL**

Ensure that your environment is activated and that you run `python` on your terminal to open a REPL:

```
>>> import hypernetx as hnx
>>> H = hnx.Hypergraph(data)
>>> list(H.nodes)
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
>>> list(H.edges)
[0, 1, 2, 3]
>>> H.shape
(8, 4)
```

**Other Actions if installed from source**

If you have installed HyperNetX from source, you can perform additional actions such as viewing the provided Jupyter notebooks or building the documentation locally.

Ensure that you have activated your virtual environment and are at the root of the source directory before running any of the following commands:
Viewing jupyter notebooks

The following command will automatically open the notebooks in a browser.

```
make tutorials
```

Building documentation

The following commands will build and open a local version of the documentation in a browser:

```
cd docs
make html
open build/index.html
```

5.2.5 Using HyperNetX on Docker

As an alternative to installing HyperNetX, you can use the officially supported HyperNetX Docker image maintained at DockerHub. Use the image to quickly start HyperNetX in a Docker container. The container starts a Jupyter Notebook that has the latest version of HyperNetX and HNXWidget installed; it also contains all the HyperNetX tutorials.

Prerequisites

- Docker
- Docker-Compose

Steps

1. Run the container
   1. Using Docker CLI, run the container in the foreground:

   ```
docker run -it --rm -p 8888:8888 -v "$PWD":/home/jovyan/work hypernetx/
   "hypernetx:latest"
   ```

2. Alternatively, create a `docker-compose.yml` file with the following:

   ```
   version: '3'
   services:
     hypernetx:
       image: hypernetx/hypernetx:latest
       ports:
         - "8888:8888"
       tty: true
       stdin_open: true
       volumes:
         - "$PWD:/home/jovyan/work"
   ```

   Once `docker-compose.yml` is created, run the container:
2. Open Jupyter Notebook

After the container has started, access the HyperNetX Jupyter Notebooks by opening the following URL in a browser:

- http://localhost:8888/tree

5.3 HNX Data Structures

The HNX library centers around the idea of a hypergraph. There are many definitions of a hypergraph. In HNX a hypergraph is a tuple of three sets, \( H = (V, E, \mathcal{I}) \).

- \( V \) a set of nodes (aka hypernodes, vertices), distinguished by unique identifiers
- \( E \) a set of edges (aka hyperedges), distinguished by unique identifiers
- \( \mathcal{I} \), a set of incidences, which form a subset of \( E \times V \), distinguished by the pairing of unique identifiers of edges in \( E \) and nodes in \( V \)

The incidences \( \mathcal{I} \) can be described by a Boolean function, \( \mathcal{I}_B : E \times V \rightarrow \{0, 1\} \), indicating whether or not a pair is included in the hypergraph.

In HNX we instantiate \( H = (V, E, \mathcal{I}) \) using three hypergraph views. We can visualize this through a high level diagram of our current code structure shown in Fig. 1. Here we begin with data (e.g., data frame, dictionary, list of lists, etc.) that is digested via the appropriate factory method to construct property stores for nodes, edges, and incidences as well as an incidence store that captures the hypergraph structure. These four objects are then used to create three hypergraph views that the hypergraph object uses to access and analyze the hypergraph structure and attributes.

5.4 Glossary of HNX terms

**Note:** For all definitions below, assume \( H = (V, E, \mathcal{I}) \) is a hypergraph.

**degree**

Given a hypergraph \( H = (V, E, \mathcal{I}) \), the degree of a node in \( V \) is the number of edges in \( E \) to which the node is incident. See also: s-degree

**dual**

The dual of a hypergraph exchanges the roles of the edges and nodes in the hypergraph. For a hypergraph \( H = (V, E, \mathcal{I}) \) the dual is \( H_D = (E, V, \mathcal{I}^T) \) where the ordered pairs in \( \mathcal{I}^T \) are the transposes of the ordered pairs in \( \mathcal{I} \). The incidence matrix of \( H_D \) is the transpose of the incidence matrix of \( H \).
edges

hyperedges
A set of objects distinguished by unique identifiers (uids). Each edge has metadata associated with it. Edges are assigned a weight either by default or specified by the user. Edges contain nodes. Nodes are elements of edges.

elements
The elements of an edge is the set of nodes incident to the edge in the Hypergraph.

hypergraph
A hypergraph is a tuple of three sets, \( H = (V, E, \mathcal{I}) \).

- \( V \), a set of nodes (aka hypernodes, vertices), distinguished by unique identifiers
- \( E \) a set of edges (aka hyperedges), distinguished by unique identifiers
- \( \mathcal{I} \), a set of incidences, which form a subset of \( E \times V \), distinguished by the pairing of unique identifiers of edges in \( E \) and nodes in \( V \)

HypergraphView
Class in hyp_view.py tying the properties of hypergraph objects held in the PropertyStore class, which holds metadata, with their ids held in the IncidenceStore class, which holds the Hypergraph relationships. The PropertyStores are unaware of the IncidenceStore and vice versa.

incidence matrix
A rectangular matrix constructed from a hypergraph, \( H = (V, E, \mathcal{I}) \). The rows of the matrix are indexed by \( V \). The columns of the matrix are indexed by \( E \). An entry in the matrix at position \((v, e)\) for some \( v \in V \) and \( e \in E \) is nonzero if and only if \((e, v) \in \mathcal{I}\). A weighted incidence matrix uses the incidence weight associated with \((e, v)\) for the nonzero entry. An unweighted incidence matrix has the integer 1 in all nonzero entries.

incidences
The ordered pairs in \( \mathcal{I} \subset E \times V \), which define the relationships in the hypergraph. The incidences \( \mathcal{I} \) of a hypergraph provide the minimal amount of data required to instantiate the hypergraph. The Edges \( E \) and Nodes \( V \) of a Hypergraph can be inferred from the pairs \((e, v)\) in the Incidences.

Each incidence uniquely identifies a single edge and node. Each incidence has metadata assigned to it. Incidences in a hypergraph are assigned a weight either by default or specified by a user. If \((e, v) \in \mathcal{I}\) then \( e \) contains \( v \), \( v \) is an element of \( e \), and \( v \) has membership in \( e \).

IncidenceStore
Class in incidence_store.py holding the set of ordered pairs of Edges and Nodes belonging to the
hypergraph. The *elements* and *memberships* are inferred from the *incidences* held in the IncidenceStore.

**memberships**

The memberships of a node is the set of edges incident to the node in the Hypergraph.

**multihypergraph**

HNX hypergraphs may be multihypergraphs. A multihypergraph is a hypergraph that allows distinct edges to contain the same set of *elements* and distinct nodes to belong to the same set of edges (aka *memberships*). When collapsing a hypergraph, edges incident with the same set of nodes or nodes incident with the same set of edges are collapsed to single objects.

**nodes**

**vertices**

**hypernodes**

A set of objects distinguished by unique identifiers (uids). Each node has metadata associated with it. Nodes are assigned a weight either by default or specified by the user. Nodes belong to edges. Nodes have *memberships* in edges.

**PropertyStore**

Class in property_store.py. Each of the basic sets in a hypergraph, (Nodes, Edges, Incidences), have metadata stored in a PropertyStore. By storing the data and metadata in a single place, updates and references have a single source of truth.

**s-adjacency matrix**

For a positive integer s, a square matrix for a hypergraph, \( H = (V, E, I) \), indexed by \( V \) such that an entry \( (v_1, v_2) \) is nonzero if only if \( v_1, v_2 \in V \) are s-adjacent. An s-adjacency matrix can be weighted or unweighted, in which case all entries are 0’s and 1’s.

**s-adjacent**

For a hypergraph, \( H = (V, E, I) \), and positive integer s, two nodes in \( V \) are s-adjacent if there are at least s edges in \( E \), which contain both of them.

**s-auxiliary matrix**

**s-edge-auxiliary matrix**

For a hypergraph, \( H = (V, E, I) \), and positive integer s, the submatrix of the *s-adjacency matrix* or the *s-edge-adjacency matrix* obtained by removing all 0-rows and 0-columns.

**s-connected**

**s-node-connected**

**s-edge-connected**

A hypergraph is s-connected if it has one s-connected
component. Similarly for s-node-connected and s-edge-connected.

**s-connected component**
**s-node-connected component**
**s-edge-connected component**

For a hypergraph, \( H = (V, E, I) \), and positive integer \( s \), an s-connected component is a subhypergraph induced by a subset of \( V \) with the property that there exists an s-walk between every pair of nodes in this subset. An s-connected component is the maximal such subset in the sense that it is not properly contained in any other subset satisfying this property.

An s-node-connected component is an s-connected component. An s-edge-connected component is an s-connected component of the dual of \( H \).

**s-degree**

For a hypergraph, \( H = (V, E, I) \), and positive integer \( s \), the s-degree of a node, \( v \in V \) is the number of edges in \( E \) of size at least \( s \) to which \( v \) belongs. See also: degree

**s-diameter**

For a hypergraph, \( H = (V, E, I) \), and positive integer \( s \), the s-diameter is the maximum s-distance over all pairs of nodes in \( V \).

**s-distance**

For a hypergraph, \( H = (V, E, I) \), and positive integer \( s \), the s-distances between two nodes in \( V \) is the length of the shortest s-node-walk between them. If no s-node-walk between the pair of nodes exists, the s-distance between them is infinite.

**s-edge**

For a hypergraph, \( H = (V, E, I) \), and positive integer \( s \), an s-edge is any edge \( e \in E \) of size at least \( s \), where the size of \( e \) equals the number of nodes in \( V \) belonging to \( e \).

**s-edge-adjacency matrix**

An s-edge-adjacency matrix is the s-adjacency matrix for the dual of \( H \).

**s-edge-adjacent**

For a hypergraph, \( H = (V, E, I) \), and positive integer \( s \), two edges in \( E \) are s-edge-adjacent if they there are at least \( s \) nodes in \( V \) belonging to both of them. Another way of saying this is two edges are s-edge-adjacent if they are s-adjacent in the dual of \( H \).

**s-edge-distance**

The s-edge-distance between two edges in \( E \) is the length of the shortest s-edge-walk between them.

---

5.4. Glossary of HNX terms

19
If no s-edge-walk between the pair of edges exists, then s-distance between them is infinite.

**s-edge-walk**
For a hypergraph, $H = (V, E, I)$, and positive integer $s$, a sequence of edges in $E$ such that each successive pair of edges are s-edge-adjacent. The length of the s-edge-walk is the number of adjacent pairs in the sequence.

**s-linegraph**
For a hypergraph, $H = (V, E, I)$, and positive integer $s$, an s-linegraph $G$ is a graph representing the node to node or edge to edge connections defined by the s-adjacency matrices.

The node s-linegraph, $G_V$ is a graph on the set $V$. Two nodes in $V$ are incident in $G_V$ if they are s-adjacent.

The edge s-linegraph, $G_E$ is a graph on the set $E$. Two edges in $E$ are incident in $G_E$ if they are s-edge-adjacent.

**s-node-walk**
For a hypergraph, $H = (V, E, I)$, and positive integer $s$, a sequence of nodes in $V$ such that each successive pair of nodes are s-adjacent. The length of the s-node-walk is the number of adjacent pairs in the sequence.

**s-walk**
Either an s-node-walk or an s-edge-walk. The length of the s-walk is the number of adjacent pairs in the sequence.

**simple hypergraph**
A hypergraph for which no edge is completely contained in another.

**subhypergraph**
A subhypergraph of a hypergraph, $H = (V, E, I)$, is a hypergraph, $H' = (V', E', I')$ such that $(e', v') \in I'$ if and only if $e' \in E' \subseteq E$, $v' \in V' \subseteq V$ and $(e, v) \in I$.

**toplex**
A toplex in a hypergraph, $H = (V, E, I)$, is an edge $e \in E$ whose set of elements is not properly contained in any other edge in $E$. That is, if $f \in E$ and the elements of $e$ are all elements of $f$ then the elements of $f$ are all elements of $e$. 
5.5 HyperNetX Packages

5.5.1 classes

classes package

class classes.Hypergraph(
    setsystem=None, default_cell_weight=1, edge_col=0, node_col=1,
    cell_weight_col='weight', misc_cell_properties_col=None, aggregate_by='first',
    properties=None, misc_properties_col=None, weight_prop_col='weight',
    default_weight: float | int = 1, edge_properties=None,
    misc_edge_properties_col=None, edge_weight_prop_col='weight',
    default_edge_weight=1, node_properties=None, misc_node_properties_col=None,
    node_weight_prop_col='weight', default_node_weight=1, name=None, **kwargs)

Bases: object

Parameters

- **setsystem** (pandas.DataFrame, dict of iterables, dict of dicts, list of iterables, numpy.ndarray, optional, default=None) – See SetSystem below for additional setsystem requirements.

- **edge_col** (str | int, optional, default=0) – column index (or name) in pandas.DataFrame, used for (hyper)edge ids. Only used when setsystem is a pandas.DataFrame

- **node_col** (str | int, optional, default=1) – column index (or name) in pandas.dataframe, used for node ids. Only used when setsystem is a pandas.DataFrame

- **cell_weight_col** (str | int, optional, default="weight") – column index (or name) in pandas.DataFrame used for referencing cell weights. For a dict of dicts, it will be used as a key in the nested dictionary of properties. These are the same as edge dependent node weights and will populate the incidence matrix when weights=True.

- **default_cell_weight** (int | float, optional, default=1) – All incidence pairs in the Hypergraph are assigned a default weight if weight is not specified in the setsystem.

- **misc_cell_properties_col** (str | int, optional, default=None) – Used for Pandas Dataframe with one column containing dictionaries of properties. Useful if objects have diverse property sets. Ignored for other setsystem types.

- **properties** (pd.DataFrame | dict, optional, default=None) – Concatenation/union of edge_properties and node_properties. By default, the object id is used and should be the first column of the dataframe, or key in the dict. If there are nodes and edges with the same ids but distinct properties then separate them and use the edge_properties and node_properties keywords.

- **weight_prop_col** (str, optional, default=None) – Name of property in properties to use for weight

- **default_weight** (int | float, optional, default=1) – Used when weight property is missing or undefined

- **edge_properties** (pd.DataFrame | dict, optional, default=None) – Properties associated with edge ids. If a dataframe, the first column must be the names of the edges. First column of dataframe or keys of dict link to edge ids in setsystem.

- **edge_weight_prop_col** (str, optional, default=None) – Name of property in edge_properties to use for weight.
Hypergraphs in HNX 2.3

An hnx.Hypergraph \(H = (V,E)\) references a pair of disjoint sets: \(V\) = nodes (vertices) and \(E\) = (hyper)edges. HNX allows for multi-edges by distinguishing edges by their unique identifiers instead of their contents. For example, if \(V = \{1,2,3\}\) and \(E = \{e_1,e_2,e_3\}\), where \(e_1 = \{1,2\}\), \(e_2 = \{1,2\}\), and \(e_3 = \{1,2,3\}\), the edges \(e_1\) and \(e_2\) contain the same set of nodes and yet are distinct and are distinguishable within \(H = (V,E)\).

New as of version 2.3, HNX provides methods to easily store and access additional metadata such as cell, edge, and node weights. Metadata associated with all edges, nodes, and (edge,node) incidence pairs stored in the hypergraph are viewable using:

```python
>>> H.edges.to_dataframe
>>> H.nodes.to_dataframe
>>> H.incidences.to_dataframe
```

The fundamental object needed to create a hypergraph is a setsystem. The setsystem defines the many-to-many relationships between edges and nodes in the hypergraph. Properties for the incidence pairs are defined within the setsystem. Properties for the edges and nodes are defined with separate Pandas DataFrames or dictionaries.

A hypergraph is defined by its relationships. While the nodes and edges are distinct objects with their own properties, it is only when they are in a relationship (i.e. incidence pair) that nodes and edges are viewable within the hypergraph structure. Consequently, hypergraph metrics and combinatorics do not use “isolated” nodes or “empty” edges. For example, while node_properties could contain any number of node identifiers, only nodes belonging to an edge in the hypergraph are counted when computing the size and shape of the hypergraph.

SetSystems

There are five types of setsystems currently accepted by the library.

1. iterable of iterables: Barebones hypergraph, which uses Pandas default indexing to generate hyperedge ids. Elements must be hashable.

```python
>>> list_of_lists = [['book','candle','cat'],['book','coffee cup'],['coffee cup →','radio']]
>>> H = Hypergraph(list_of_lists)
```
2. **dictionary of iterables**: The most basic way to express many-to-many relationships providing edge ids. The elements of the iterables must be hashable:

```python
>>> scenes_dictionary = {
    0: ('FN', 'TH'),
    1: ('TH', 'JV'),
    2: ('BM', 'FN', 'JA'),
    3: ('JV', 'JU', 'CH', 'BM'),
    4: ('JU', 'CH', 'BR', 'CN', 'CC', 'JV', 'BM'),
    5: ('TH', 'GP'),
    6: ('GP', 'MP'),
    7: ('MA', 'GP'),
    8: ('FN', 'TH')
}
>>> H = hnx.Hypergraph(scenes_dictionary)
```

3. **dictionary of dictionaries**: allows cell properties to be assigned to a specific (edge, node) incidence. This is particularly useful when there are variable length dictionaries assigned to each pair:

```python
>>> nested_dictionary = {
    0: {'FN': { 'time': 'early', 'weight': 7, 'TH': { 'time': 'late' } },
    1: { 'TH': { 'subject': 'war', 'JV': { 'observed_by': 'someone' } },
    2: { 'BM': {}, 'FN': {}, 'JA': { 'role': 'policeman' } },
    3: { 'JV': { 'was_carrying': 'stick' }, 'JU': {}, 'CH': {}, 'BM': { 'state': { 'intoxicated': 'color': 'pinkish' } },
    4: { 'JU': { 'weight': 15 }, 'CH': {}, 'BR': { 'state': 'worried' }, 'CN': {}, 'CC': {} },
    5: { 'TH': {}, 'GP': {} },
    6: { 'GP': {}, 'MP': {} },
    7: { 'MA': {}, 'GP': { 'accompanied_by': 'dog', 'weight': 15, 'was_singing': '
Frère Jacques' } }
}
>>> H = hnx.Hypergraph(nested_dictionary)
```

4. **pandas.DataFrame**: For large datasets and for datasets with cell properties it is most efficient to construct a hypergraph directly from a pandas.DataFrame. Incidence pairs are in the first two columns. Cell properties shared by all incidence pairs can be placed in their own column of the dataframe. Variable length dictionaries of cell properties particular to only some of the incidence pairs may be placed in a single column of the dataframe. Representing the data above as a dataframe df:

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
<th>w</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>1</td>
<td>0.5</td>
<td>{'name': 'related_to'}</td>
</tr>
<tr>
<td>e1</td>
<td>2</td>
<td>0.1</td>
<td>{'name': 'related_to',</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;start-date&quot;: &quot;05.13.2020&quot;}</td>
</tr>
<tr>
<td>e2</td>
<td>1</td>
<td>0.52</td>
<td>{'name': 'owned_by',</td>
</tr>
</tbody>
</table>

The first row of the dataframe is used to reference each column.

```python
>>> import pandas as pd
>>> d = { 'col1': ['e1', 'e1', 'e2'], 'col2': [1, 2, 1], 'w': [0.5, 0.1, 0.52],
```
### Edge and Node Properties

Properties specific to a single edge or node are passed through the keywords: `edge_properties`, `node_properties`, or `properties`. Properties may be passed as dataframes or dictionaries. The first column or index of the dataframe or the keys of the dictionary correspond to the edge and/or node identifiers. If identifiers are shared among edges and nodes, or are distinct for edges and nodes, properties may be combined into a single object and passed to the `properties` keyword. For example:

```python
uid   weight properties
e1    5.0    {'type': 'event'}
e2    0.52   {'name': 'owned_by'}
...    ...    ...
1      1.2    {'color': 'red'}
2      .003   {'name': 'Fido', 'color': 'brown'}
3      1.0    {}
```

A properties dictionary should have the format:

```python
dp = {uid1: {prop1:val1, prop2:val2, ...},
      uid2: {...},
      ...}
```

### Weights

The default key for cell and object weights is “weight”. The default value is 1. Weights may be assigned from a column in the dataframe by specifying the column and/or a new default in the constructor using `cell_weight_col` and `default_cell_weight` for incidence pairs, and using `edge_weight_prop_col`, `default_edge_weight` for edges, `node_weight_prop_col`, `default_node_weight` for nodes, and `weight_prop_col`, `default_weight` for a shared property dataframe.

```python
add_edge(edge_uid, inplace=True, **attr)
```

Add a single edge with attributes to edge properties. Does not add an incidence to the hypergraph.
Parameters
- **edge_uid** (*int / str*) – edge_uid
- **inplace** (*bool, default=True*) – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.
- **attr** (*dict, optional*) – Properties to add to edges as key=value pairs.

Return type
- *Hypergraph*

``add_edges_from(edge_uids, inplace=True)``
Add a collection of edges with attributes to edge properties. Does not add an incidence to the hypergraph.

Parameters
- **edge_uids** (*list[int | str] | list[tuple[int | str, dict]], list[int | str | tuple[int | str, dict]]*) – edge_uids must be a list of uids and/or tuples of the form (uid, data) where data is dictionary
- **inplace** (*bool, default=True*) – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.

Return type
- *Hypergraph*

``add_incidence(edge_uid, node_uid, inplace=True, **attr)``
Add a single incidence with attributes to Hypergraph.

Parameters
- **edge_uid** (*int | str*) – edge_uid
- **node_uid** (*int | str*) – node_uid
- **inplace** (*bool, optional, default=True*) – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.
- **attr** (*dict, optional*) – Properties to add to incidences as key=value pairs.

Returns
- Hypergraph with incidences added.

Return type
- *Hypergraph*

``add_incidences_from(incidences, inplace=True)``
Adds a collection of incidences to Hypergraph

Parameters
- **incidences** (*list[str | int, str | int], list[tuple[str | int, str, dict], list[str | int, dict[str, Any]]]*) – Incidence pairs must be a list of uids of the form (edge_uid, node_uid) and/or tuples of the form (edge_uid, node_uid, data) where data is a dictionary.
- **inplace** (*bool, optional, default=True*) – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.

Returns
- Hypergraph with incidences added.

Return type
- *Hypergraph*
**add_node**(*node_uid*, *inplace=True, **attr*)

Add a single node with attributes to node properties. Does not add an incidence to the hypergraph.

**Parameters**

- **node_uid** ([int | str]) – node_uid
- **inplace** (bool, default=True) – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.
- **attr** (dict, optional) – Properties to add to edges as key=value pairs.

**Return type**

Hypergraph

**add_nodes_from**(*node_uids*, *inplace=True*)

Add a collection of nodes with attributes to nodes properties. Does not add an incidence to the hypergraph.

**Parameters**

- **node_uids** ([list[int | str] | list[tuple[int | str, dict]], list[int | str | tuple[int | str, dict]]]) – node_uids must be a list of uids and/or tuples of the form (uid, data) where data is dictionary
- **inplace** (bool, default=True) – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.

**Return type**

Hypergraph

**add_nodes_to_edges**(*edge_dict*, *inplace=True*)

Adds a collection of incidences to Hypergraph

**Parameters**

- **edge_dict** (dict[str, list[str | int] | dict[str, dict]]) – The edge dictionary must be a dictionary of edges as the keys and a list of nodes or a dictionary of nodes to properties as the values.
- **inplace** (bool, default=True) – If True, changes the existing. Otherwise, creates a new Hypergraph with the requested changes.

**Returns**

Hypergraph with the updated edges and their newly added nodes

**Return type**

Hypergraph

**adjacency_matrix**(*s=1, index=False*)

Returns the s-adjacency matrix for the hypergraph.

**Parameters**

- **s** (int, optional, default=1)
- **index** (boolean, optional, default=False) – If True, returns both the adjacency matrix and an array containing the row and column index of node_uids

**Returns**

- adjacency matrix (scipy.sparse.csr_matrix)
- node indexes (np.ndarray) – an np.ndarray containing the row and column index of node_uids.
The unweighted \( s \)-auxiliary matrix for hypergraph

**Parameters**

- \( s \) (int, optional, default=1)
- `node` (bool, optional, default=True) – whether to return based on node or edge adjacencies
- `index` (bool, optional, default=False) – If True, returns both the auxiliary matrix and an array containing the row and column index of node or edge uids

**Returns**

- \( s \)-auxiliary matrix (scipy.sparse.csr_matrix) – Node/Edge adjacency matrix with empty rows and columns removed
- `index` (np.ndarray) – row and column index of node or edge uids

`bipartite(keep_data=False, directed=False)`

Creates a bipartite NetworkX graph from hypergraph. The nodes and (hyper)edges of hypergraph become the nodes of bipartite graph. For every (hyper)edge \( e \) in the hypergraph and node \( v \) in \( e \) there is an edge \((e,v)\) in the graph.

**Parameters**

- `keep_data` (bool, optional, default = False) – If True the node and edge data from the hypergraph will be added to the NetworkX graph
- `directed` (bool, optional, default = False) – If True the graph edges will be directed so that the hypergraph edges are the sources and the hypergraph nodes are the targets

**Return type**

networkx.Graph or DiGraph

`clone(name=None)`

Create a deep copy of the hypergraph

**Parameters**

- `name` (str, optional, default = None)

**Return type**

Hypergraph

`collapse_edges(name=None, use_uids=None, use_counts=False, return_counts=True, return_equivalence_classes=False, aggregate_edges_by=None, aggregate_cells_by=None)`

Returns a new hypergraph by collapsing edges.

**Parameters**

- `name` (str, optional, default = None)
- `use_uids` (list, optional, default = None) – Specify the edge identifiers to use as representatives for a single equivalence class. If two identifiers occur in the same equivalence class, the first one found will be used.
- `use_counts` (boolean, optional, default = False) – Rename the equivalence class representatives as \(<uid>:<size of class>\)
• **return_counts** *(bool, optional, default = True)* — Add the size of the equivalence class to the properties associated to the representative in the collapsed hypergraph using keyword: `eclass_size`

• **return_equivalence_classes** *(bool, optional, default = False)* — Returns a dictionary of edge equivalence classes keyed by a representative from each class

• **aggregate_edges_by** *(dict, optional, default = {'weight': 'sum'})* — Dictionary of aggregation methods keyed by column names in the properties dataframes, does not apply to `misc_properties`. Defaults to ‘first’ on unlisted columns. See pandas.core.groupby.DataFrameGroupBy.agg for usage examples with dictionaries

• **aggregate_cells_by** *(dict, optional, default = {'weight': 'sum'})* — Dictionary of aggregation methods keyed by column names in the properties dataframes, does not apply to `misc_properties`. Defaults to ‘first’ on unlisted columns. See pandas.core.groupby.DataFrameGroupBy.agg for usage examples with dictionaries

Return type

*Hypergraph*

Notes

Collapses the edges of Hypergraph. Two edges are duplicates if their respective elements are the same. Using this as an equivalence relation, the uids of the edges are partitioned into equivalence classes. A single member of the equivalence class is chosen to represent the class.

Example

```python
>>> data = {'E1': ('a', 'b'), 'E2': ('a', 'b')}
>>> h = Hypergraph(data)
>>> h.incidence_dict
{'E1': ['a', 'b'], 'E2': ['a', 'b']}
>>> h.collapse_edges().incidence_dict
{'E1': ['a', 'b']}
>>> h.collapse_edges(use_counts=True).incidence_dict
{'E1:2': ['a', 'b']}
>>> h.collapse_edges().properties.to_dict(orient='records')
[{{'weight': 2.0, 'misc_properties': {}}}, {{'weight': 2.0, 'misc_properties': {}}}]```

collapse_nodes*(name=None, use_uids=None, use_counts=False, return_counts=True, return_equivalence_classes=False, aggregate_nodes_by=None, aggregate_cells_by=None)*

Returns a new hypergraph by collapsing nodes.

Parameters

• **name** *(str, optional, default = None)*

• **use_uids** *(list, optional, default = None)* — Specify the node identifiers to use as representatives for a single equivalence class. If two identifiers occur in the same equivalence class, the first one found will be used.

• **use_counts** *(bool, optional, default = False)* — Rename the equivalence class representatives as `<uid>:<size of class>`
• **return_counts** *(bool, optional, default = True)* – Add the size of the equivalence class to the properties associated to the representative in the collapsed hypergraph using keyword: `eclass_size`

• **return_equivalence_classes** *(boolean, optional, default = False)* – Returns a dictionary of edge equivalence classes keyed by a representative from each class

• **aggregate_nodes_by** *(dict, optional, default = {‘weight’:‘sum’})* – Dictionary of aggregation methods keyed by column names in the properties dataframes, does not apply to `misc_properties`. Defaults to ‘first’ on unlisted columns. See pandas.core.groupby.DataFrameGroupBy.agg for usage examples with dictionaries

• **aggregate_cells_by** *(dict, optional, default = {‘weight’:‘sum’})* – Dictionary of aggregation methods keyed by column names in the properties dataframes, does not apply to `misc_properties`. Defaults to ‘first’ on unlisted columns. See pandas.core.groupby.DataFrameGroupBy.agg for usage examples with dictionaries

**Return type**

Hypergraph

**Notes**

Collapses the nodes of Hypergraph. Two nodes are duplicates if their respective memberships are the same. Using this as an equivalence relation, the uids of the nodes are partitioned into equivalence classes. A single member of the equivalence class is chosen to represent the class.

**Example**

```python
>>> data = {'E1': ('a', 'b'), 'E2': ('a', 'b')}
>>> h = Hypergraph(data)
>>> h.incidence_dict
{'E1': ['a', 'b'], 'E2': ['a', 'b']}
>>> h.collapse_nodes().incidence_dict
{'E1': ['a'], 'E2': ['a']}
>>> h.collapse_nodes(use_counts=True).incidence_dict
{'E1: ['a:2'], 'E2: ['a:2']}
>>> h.collapse_nodes().properties.to_dict(orient='records')
[{{'weight': 2.0, 'misc_properties': {}}, {'weight': 2.0, 'misc_properties': {}}}]```

collapse_nodes_and_edges *(name=None, use_edge_uids=None, use_node_uids=None, use_counts=False, return_counts=True, return_equivalence_classes=False, aggregate_nodes_by=None, aggregate_edges_by=None, aggregate_cells_by=None)*

Returns a new hypergraph by collapsing nodes and edges.

**Parameters**

• **name** *(str, optional, default = None)*

• **return_equivalence_classes** *(boolean, optional, default = False)* – Returns a dictionary of edge equivalence classes keyed by a representative from each class

• **use_edge_uids** *(list, optional, default = None)* – Specify the edge and node identifiers to use as representatives for a single equivalence class. If two identifiers occur in the same equivalence class, the first one found will be used.
• **use_node_uids** *(list, optional, default = None)* – Specify the edge and node identifiers to use as representatives for a single equivalence class. If two identifiers occur in the same equivalence class, the first one found will be used.

• **use_counts** *(boolean, optional, default = False)* – Rename the equivalence class representatives as `<uid>:<size of class>`

• **return_counts** *(bool, optional, default = True)* – Add the size of the equivalence class to the properties associated to the representative in the collapsed hypergraph using keyword: *eclass_size*

• **aggregate_nodes_by** *(optional)* – default={'weight'='sum'}, all Method to combine duplicate rows of data for the same uids

• **aggregate_edges_by** *(optional)* – default={'weight'='sum'}, all Method to combine duplicate rows of data for the same uids

• **aggregate_cells_by** *(optional)* – default={'weight'='sum'}, all Method to combine duplicate rows of data for the same uids

Returns

• **new hypergraph** *(Hypergraph)*

• **node equivalence classes** *(dict)*

• **edge equivalence classes** *(dict)*

Notes

Collapses the Nodes and Edges of Hypergraph. Two nodes(edges) are duplicates if their respective memberships(elements) are the same. Using this as an equivalence relation, the uids of the nodes(edges) are partitioned into equivalence classes. A single member of the equivalence class is chosen to represent the class.

Example

```python
>>> data = {'E1': ('a', 'b'), 'E2': ('a', 'b')}
>>> h = Hypergraph(data)
>>> h.incidence_dict
{'E1': ['a', 'b'], 'E2': ['a', 'b']}
>>> h.collapse_nodes_and_edges().incidence_dict
{'E1': ['a']}
>>> h.collapse_nodes_and_edges(use_counts=True).incidence_dict
{'E1:2': ['a:2']}
```

**component_subgraphs** *(return_singletons=False, name=None)*

Same as `s_components_subgraphs()` with s=1. Returns iterator.

See also:

`s_component_subgraphs`

**components** *(edges=False)*

Same as `s_connected_components()` with s=1, but nodes are returned by default. Return iterator.

See also:

`s_connected_components`
`connected_component_subgraphs` *(return_singletons=True, name=None)*

Same as `s_component_subgraphs()` with s=1. Returns iterator

See also:

`s_component_subgraphs`

`connected_components` *(edges=False)*

Same as `s_connected_components()` with s=1, but nodes are returned by default. Return iterator.

See also:

`s_connected_components`

`property dataframe`

Returns dataframe of incidence properties as dataframe with edges and nodes in columns.

**Return type**

`pandas.DataFrame`

`degree` *(node_uid, s=1, max_size=None)*

The number of edges of size at least s and at most max_size that contain the node.

**Parameters**

- `node_uid` *(hashable)* – Identifier for the node.
- `s` *(int, optional, default=1)* – The smallest size (must be positive) of an edge to consider in degree.
- `max_size` *(int, optional, default=None)* – The largest size (must be positive) of edge to consider in degree.

**Returns**

The number of edges of size at least s and at most max_size that contain the node.

**Return type**

`int`

`diameter` *(s=1)*

Returns the length of the longest shortest s-walk between nodes in hypergraph

**Parameters**

- `s` *(int, optional, default=1)*

**Returns**

`diameter`

**Return type**

`int`

**Raises**

`HyperNetXError` – If hypergraph is not s-edge-connected
Notes

Two nodes are s-adjacent if they share s edges. Two nodes \(v_{\text{start}}\) and \(v_{\text{end}}\) are s-walk connected if there is a sequence of nodes \(v_{\text{start}}, v_1, v_2, \ldots, v_{n-1}, v_{\text{end}}\) such that consecutive nodes are s-adjacent. If the graph is not connected, an error will be raised.

difference \((\text{other}, \text{name}=\text{None})\)

Hypergraph obtained by restricting to incidences in self but not in other.

Parameters

- **other** (*Hypergraph*)
- **name** (*str*, optional, default = None)

Return type

*Hypergraph*

dim \((\text{edge})\)

Same as \(\text{size(\text{edge})} - 1\)

Parameters

- **edge** (*hashable*) – The uid of an edge in the hypergraph

Return type

*int*

distance \((\text{source}, \text{target}, s=1)\)

Returns the shortest s-walk distance between two nodes in the hypergraph.

Parameters

- **source** (*str | int*) – a node in the hypergraph
- **target** (*str | int*) – a node in the hypergraph
- **s** (*positive int*, optional, default=1) – the number of edges

Returns

s-walk distance

Return type

*int*

See also:

*edge_distance*

Notes

The s-distance is the shortest s-walk length between the nodes. An s-walk between nodes is a sequence of nodes that pairwise share at least s edges. The length of the shortest s-walk is 1 less than the number of nodes in the path sequence.

Uses the networkx shortest_path_length method on the graph generated by the s-adjacency matrix.

dual \((\text{name}=\text{None}, \text{share_properties}=\text{True})\)

Constructs a new hypergraph with roles of edges and nodes of hypergraph reversed.

Parameters

- **name** (*hashable, optional, default=None*)
• **share_properties**(bool, optional, default=True) – Whether to tie the edge and node properties of objects in the dual to objects in the hypergraph. If True, a change to edge and node properties in one will be reflected in the other.

**Return type**

Hypergraph

**edge_adjacency_matrix**(s=1, index=False)

Returns the **s-adjacency matrix** for the dual hypergraph.

**Parameters**

• **s**(int, optional, default=1)

• **index**(boolean, optional, default=False) – If True, returns both the adjacency matrix and an array containing the row and column index of edge_uids

**Returns**

• **edge adjacency matrix**(scipy.sparse.csr_matrix)

• **edge indexes**(np.ndarray) – an np.ndarray containing the row and column index of edge_uids.

**Notes**

This is also the adjacency matrix for the line graph. Two edges are s-adjacent if they share at least s nodes.

**edge_diameter**(s=1)

Returns the length of the longest shortest s-walk between edges in the hypergraph

**Parameters**

• **s**(int, optional, default=1)

**Returns**

edge_diameter

**Return type**

int

** Raises**

HyperNetXError – If hypergraph is not s-edge-connected

**Notes**

Two edges are s-adjacent if they share s nodes. Two nodes e_start and e_end are s-walk connected if there is a sequence of edges e_start, e_1, e_2, ... e_n-1, e_end such that consecutive edges are s-adjacent. If the graph is not connected, an error will be raised.

**edge_diameters**(s=1)

Returns the edge diameters of the s_edge_connected component subgraphs in the hypergraph.

**Parameters**

• **s**(int, optional, default=1)

**Returns**

maximum diameter, list of diameters, list of component – maximum diameter, list of diameters (List of edge_diameters for s-edge component subgraphs in hypergraph), list of component (List of the edge uids in the s-edge component subgraphs)
edge_distance(source, target, s=1)

Returns the shortest s-walk distance between two edges in the hypergraph.

**Parameters**

- **source** (str / int) – an edge in the hypergraph
- **target** (str / int) – an edge in the hypergraph
- **s** (positive int, optional, default=1) – the number of intersections between pairwise consecutive edges

**Returns**

s-walk distance – The shortest s-walk edge distance. A shortest s-walk is computed as a sequence of edges; the s-walk distance is the number of edges in the sequence minus 1. If no such path exists returns np.inf.

**Return type**

int | float

See also:

distance

**Notes**

The s-distance is the shortest s-walk length between the edges. An s-walk between edges is a sequence of edges such that consecutive pairwise edges intersect in at least s nodes. The length of the shortest s-walk is 1 less than the number of edges in the path sequence.

Uses the networkx shortest_path_length method on the graph generated by the s-edge_adjacency matrix.

edge_neighbors(edge, s=1)

The edges in hypergraph which share s nodes(s) with edge.

**Parameters**

- **edge** (hashable) – uid for an edge in hypergraph
- **s** (int, optional, default=1) – Minimum number of nodes shared by neighbors edge node.

**Returns**

a list of edge neighbors

**Return type**

list
property edges
Object associated with edges.

Return type
HypergraphView
equivalence_classes(edges=True)
Returns the equivalence classes created by collapsing edges or nodes.

Parameters
edges (bool, optional, default=True) – If True collapses edges, otherwise collapses nodes.

Returns
A list of sets of edges or nodes

Return type
list
See also:
collapse_edges, collapse_nodes, collapse_nodes_and_edges
classmethod from_bipartite(B, node_id=1, name=None, **kwargs)
Static method creates a Hypergraph from a NetworkX bipartite graph. Still to come: capturing edge and node properties from the graph for use in the hypergraph.

Parameters
• B (nx.Graph()) – A networkx bipartite graph. Each node in the graph has a property 'bipartite' taking the value of 0 or 1 indicating a 2-coloring of the graph.
• node_id (int) – bipartite value assigned to graph nodes that will be hypergraph edges
• name (hashable, optional)

Return type
Hypergraph
Notes
A partition for the nodes in a bipartite graph generates a hypergraph.

```python
>>> import networkx as nx
>>> B = nx.Graph()
>>> B.add_nodes_from([1, 2, 3, 4], bipartite=0)
>>> B.add_nodes_from(['a', 'b', 'c'], bipartite=1)
>>> B.add_edges_from([(1, 'a'), (1, 'b'), (2, 'b'), (2, 'c'), (3, 'c'), (4, 'a')])
>>> H = Hypergraph.from_bipartite(B, nodes=1)
>>> list(H.nodes), list(H.edges)
(['a', 'b', 'c'], [1, 2, 3, 4])
```
classmethod from_incidence_dataframe(df, name=None, fillna=0, key=None, return_only_dataframe=False, **kwargs)
Create a hypergraph from a Pandas Dataframe object, which has values equal to the incidence matrix of a hypergraph. Its index will identify the nodes and its columns will identify its edges.

Parameters
• **df** *(Pandas.DataFrame)* – a real valued dataframe with a single index

• **name** *(optional) string, default = None*

• **fillna** *(float, default = 0)* – a real value to place in empty cell, all-zero columns will not generate an edge.

• **key** *(optional) function, default = None* – boolean function to be applied to dataframe. will be applied to entire dataframe.

• **return_only_dataframe** *(optional) bool, default = False* – to use the incidence_dataframe with cell_properties or properties, set this to true and use it as the setsystem in the Hypergraph constructor.

**See also:**

*from_numpy_array*

**Return type**

Hypergraph | pd.DataFrame

**classmethod from_incidence_matrix** *(M, name=None, **kwargs)*

Accepts numpy.matrix or scipy.sparse matrix

**classmethod from_numpy_array** *(M, node_names=None, edge_names=None, name=None, key=None, **kwargs)*

Create a hypergraph from a real valued matrix represented as a 2 dimensional numpy array. The matrix is converted to a matrix of 0’s and 1’s so that any truthy cells are converted to 1’s and all others to 0’s.

**Parameters**

• **M** *(real valued array-like object, 2 dimensions)* – representing a real valued matrix with rows corresponding to nodes and columns to edges

• **node_names** *(object, array-like, default=None)* – List of node names must be the same length as M.shape[0]. If None then the node names correspond to row indices with ‘v’ prepended.

• **edge_names** *(object, array-like, default=None)* – List of edge names must have the same length as M.shape[1]. If None then the edge names correspond to column indices with ‘e’ prepended.

• **name** *(hashable)*

• **key** *(optional) function* – boolean function to be evaluated on each cell of the array, must be applicable to numpy.array

**Return type**

Hypergraph

**Note:** The constructor does not generate empty edges. All zero columns in M are removed and the names corresponding to these edges are discarded.

**get_cell_properties** *(edge_uid, node_uid, prop_name=None)*

Get cell properties on a specified edge and node

**Parameters**

• **edge_uid** *(str | int)* – edge_uid

• **node_uid** *(str | int)* – node_uid
• prop_name (str, optional, default=None) – name of a cell property; if None, all cell properties will be returned

Returns
cell property value if prop_name is provided, otherwise dict of all cell properties and values

Return type
Any

get_linegraph(*s=1, edges=True*)

Creates an s-linegraph for the Hypergraph. If edges=True, then the edges will be the vertices of the line graph. Two vertices are connected by an s-line-graph edge if the corresponding hypergraph edges intersect in at least s hypergraph nodes. If edges=False, the hypergraph nodes will be the vertices of the line graph. Two vertices are connected if the nodes they correspond to share at least s incident (hyper)edges.

Parameters
• s (int) – The width of the connections.
• edges (bool, optional, default = True) – Determine if edges or nodes will be the vertices in the linegraph.

Returns
A NetworkX graph.

Return type
nx.Graph

get_properties(uid, level=0, prop_name=None)

Returns an object’s specific property or all properties

Parameters
• uid (hashable) – edge or node id
• level (int | None , optional, default=0) – Enter 0 for edges and 1 for nodes.
• prop_name (str | None, optional, default=None) – if None then all properties associated with the object will be returned.

Returns
single property or dictionary of properties

Return type
Any

incidence_dataframe(weights=False)

property incidence_dict

Dictionary keyed by edge uids with values as the uids of nodes of each edge

Return type
dict

incidence_matrix(index=False, weights=False)

A sparse matrix indicating the existence of an incidence pair in the hypergraph. Each row corresponds to a node v and each column corresponds to an edge e. The entry corresponding to (row v, col e) is nonzero if v is an element of e. If weights = True then the value equals the weight given in the hypergraph incidence properties for the incidence pair (e,v). Otherwise, the value is 1.

Parameters
HyperNetX Documentation, Release 2.3.5

- **index** *(bool, optional, default = False)* – If `index=True`, returns a tuple containing the incidence matrix, an np.ndarray containing the row and column index of node_uids, and an np.ndarray containing the row and column index of edge_uids. Otherwise, returns the incidence matrix.

- **weights** *(bool, optional, default = False)* – If True, use the incidence weights corresponding to the row and column of the entry.

**Returns**

- **incidence matrix** *(scipy.sparse.csr_matrix)*
- **node indexes** *(np.ndarray)* – an np.ndarray containing the row and column index of node_uids
- **edge indexes** *(np.ndarray)* – an np.ndarray containing the row and column index of edge_uids

**property incidences**

Object associated with incidence pairs

**Return type**

HypergraphView

**intersection** *(other, name=None)*

Returns a hypergraph created by restricting to incidence pairs contained in both self and other. Properties inherited from self.

**Parameters**

- **other** *(Hypergraph)*
- **name** *(str, optional, default=None)*

**Return type**

Hypergraph

**is_connected** *(s=1, edges=False)*

Determines if hypergraph is *s-connected*.

**Parameters**

- **s** *(int, optional, default=1)*
- **edges** *(boolean, optional, default=False)* – If True, will determine if *s-edge-connected*. For s=1 s-edge-connected is the same as s-connected.

**Returns**

**is_connected**

**Return type**

boolean
Notes

A hypergraph is s node connected if for any two nodes \(v_0, v_n\) there exists a sequence of nodes \(v_0, v_1, v_2, \ldots, v(n-1), v_n\) such that every consecutive pair of nodes \(v(i), v(i+1)\) share at least s edges.

A hypergraph is s edge connected if for any two edges \(e_0, e_n\) there exists a sequence of edges \(e_0, e_1, e_2, \ldots, e(n-1), e_n\) such that every consecutive pair of edges \(e(i), e(i+1)\) share at least s nodes.

neighbors(\(node, s=1\))

The nodes in hypergraph which share s edge(s) with node.

Parameters

- \(node\) (hashable) – uid for a node in hypergraph
- \(s\) (int, optional, default=1) – Minimum number of edges shared by neighbors with node.

Returns

neighbors – s-neighbors share at least s edges in the hypergraph

Return type

list

node_diameters(s=1)

Returns the node diameters of the connected components in the hypergraph.

Parameters

- \(s\) (int, optional, default=1)

Returns

maximum diameter, list of diameters, list of component – maximum diameter, list of diameters (List of node_diameters for s-node component subgraphs in hypergraph), list of component (List of the node uids in the s-node component subgraphs)

Return type

tuple[int, list, list]

property nodes

Object associated with nodes.

Return type

HypergraphView

order()

The number of nodes in hypergraph.

Returns

order

Return type

int

property properties

Returns incidence properties

Return type

pandas.DataFrame

remove_edges(edge_uids, name=None, inplace=True)

Removes the edges from the Hypergraph. If inplace=True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.
Parameters

- **edge_uids** (str | int | list[str | int]) – edge_uids

- **name** (str, optional, default=None) – The name of the new Hypergraph. Used only when inplace=False; ignored if inplace=True.

- **inplace** (bool, optional, default=True) – Whether to replace the current hypergraph with a new one.

Return type

Hypergraph

**remove_incidence**s(incidence_uids, name=None, inplace=True)

Removes the incidences from the Hypergraph. If inplace=True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.

Parameters

- **incidence_uids** (tuple[str | int] | list[tuple[str | int]]) – incidence_uids

- **name** (str, optional, default=None) – The name of the new Hypergraph. Used only when inplace=False; ignored if inplace=True.

- **inplace** (bool, optional, default=True) – Whether to replace the current hypergraph with a new one.

Return type

Hypergraph

**remove_nodes**(node_uids, name=None, inplace=True)

Removes the nodes from the Hypergraph. If inplace=True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.

Parameters

- **node_uids** (str | int | list[str | int]) – node_uids

- **name** (str, optional, default=None) – The name of the new Hypergraph. Used only when inplace=False; ignored if inplace=True.

- **inplace** (bool, optional, default=True) – Whether to replace the current hypergraph with a new one.

Return type

Hypergraph

**remove_singletons**(name=None)

Constructs clone of hypergraph with singleton edges removed.

Parameters

- **name** (str, optional, default=None)

Return type

Hypergraph

**rename**(edges=None, nodes=None, name=None, inplace=True)

Rename the edges and/or nodes of the hypergraph.

Parameters

- **edges** (dict, optional, default = None) – dictionary of replacement edge_uids
- **nodes** (*dict, optional, default=None*) – dictionary of replacement node_uids
- **name** (*str, optional, default=None*)
- **inplace** (*bool, optional, default=True*)

**Return type**

*Hypergraph*

**restrict_to_edges** (*edges, name=None*)

New hypergraph gotten by restricting to edges

**Parameters**

- **edges** (*Iterable*) – edge identifiers to restrict to
- **name** (*str | int, optional, default=None*) – edge identifier

**Return type**

*Hypergraph*

**restrict_to_nodes** (*nodes, name=None*)

New hypergraph gotten by restricting to nodes

**Parameters**

- **nodes** (*Iterable*) – node identifiers to restrict to
- **name** (*str | int, optional, default=None*) – node identifier

**Return type**

*Hypergraph*

**s_component_subgraphs** (*s=1, edges=True, return_singletons=False, name=None*)

Returns a generator for the induced subgraphs of s_connected components. Removes singletons unless return_singletons is set to True. Computed using s-linegraph generated either by the hypergraph (edges=True) or its dual (edges = False)

**Parameters**

- **s** (*int, optional, default=1*)
- **edges** (*boolean, optional, default=False*) – Determines if edge or node components are desired. Returns subgraphs equal to the hypergraph restricted to each set of nodes(edges) in the s-connected components or s-edge-connected components
- **return_singletons** (*bool, optional, default=False*) – If True, keep singletons in subgraph. Otherwise, remove singletons.
- **name** (*str, optional, default=None*)

**Yields**

- **s_component_subgraphs** (*iterator*) – Iterator returns subgraphs generated by the edges (or nodes) in the s-edge(node) components of hypergraph.

**s_components** (*s=1, edges=True, return_singletons=True*)

Same as **s_connected_components()**

**See also:**

- **s_connected_components**
**s_connected_components**(*s=1, edges=True, return_singletons=False*)

Returns a generator for the *s*-edge-connected component or the *s*-node-connected component of the hypergraph.

**Parameters**

- **s** (int, optional, default=1)
- **edges** (boolean, optional, default=True) – If True, return edge components; otherwise, return node components
- **return_singletons** (bool, optional, default=False) – If True, keep singletons. Otherwise, remove singletons

**Notes**

If edges=True, this method returns the *s*-edge-connected components as lists of lists of edge uids. An *s*-edge-component has the property that for any two edges e1 and e2 there is a sequence of edges starting with e1 and ending with e2 such that pairwise adjacent edges in the sequence intersect in at least *s* nodes. If *s*=1 these are the path components of the hypergraph.

If edges=False this method returns *s*-node-connected components. A list of sets of uids of the nodes which are *s*-walk connected. Two nodes v1 and v2 are *s*-walk-connected if there is a sequence of nodes starting with v1 and ending with v2 such that pairwise adjacent nodes in the sequence share *s* edges. If *s*=1 these are the path components of the hypergraph.

**Example**

```python
>>> S = {'A':{1,2,3},'B':{2,3,4},'C':{5,6},'D':{6}}
>>> H = Hypergraph(S)

>>> list(H.s_connected_components(edges=True))
[['C', 'D'], ['A', 'B']]

>>> list(H.s_connected_components(edges=False))
[[1, 2, 3, 4], [5, 6]]
```

**Yields**

- **s_connected_components** (iterator) – Iterator returns sets of uids of the edges (or nodes) in the *s*-edge(node) components of hypergraph.

**set_state**(**kwargs**)

Allow state_dict updates from outside of class. Use with caution.

**Parameters**

- **kwargs** (dict, optional) – key-value pairs to save in state dictionary

**property shape**

The number of nodes, number of edges

**Returns**

- **number of nodes, number of edges**

**Return type**

tuple
**singletons()**

Returns a list of singleton edges. A singleton edge is an edge of size 1 with a node of degree 1.

- **Returns**
  - singles – A list of edge uids.

- **Return type**
  - list

**size(edge, nodeset=None)**

The number of nodes in nodeset that belong to edge. If nodeset is None then returns the size of edge

- **Parameters**
  - edge (hashable) – The uid of an edge in the hypergraph

- **Returns**
  - size

- **Return type**
  - int

**sum(other, name=None)**

Hypergraph obtained by joining incidences from self and other. Removes duplicates and uses properties of self.

- **Parameters**
  - other (Hypergraph)

- **Return type**
  - Hypergraph

**toplexes(return_hyp=False)**

Computes a maximal collection of toplexes for the hypergraph. A toplex is a hyperedge, which is not contained in any other hyperedge. If return_hyp=True, then returns the simple hypergraph created by restricting to the toplexes.

- **Parameters**
  - return_hyp (bool, optional, default=False)

- **Return type**
  - Hypergraph | list

**union(other, name=None)**

The hypergraph gotten by joining incidence pairs contained in self and other. Duplicates removed. Properties inherited from self. Same as sum()

- **Parameters**
  - other (Hypergraph)
  - name (str, optional, default=None)

- **Return type**
  - Hypergraph

**class HypergraphView**(incidence_store, level, property_store=None)

- **Bases:** object

  Wrapper for Property and Incidence Stores holding structural and metadata for hypergraph. Provides methods matching EntitySet methods in previous versions. Only nodes and edges in the Incidence Store will be seeable in this view.
property dataframe
   All properties for objects in the HypergraphView. Same as to_dataframe.
   
   **Return type**
   pd.DataFrame

property default_weight
   Default weight for an edge, node, or incidence
   
   **Return type**
   int | float

property elements
   See elements
   
   **Return type**
   dict

property incidence_dict
   incidence dictionary
   
   **Return type**
   dict | None

property incidence_store
   IncidenceStore
   
   **Return type**
   IncidenceStore

is_empty()
   Returns true if HypergraphView has no edges, nodes, or incidences depending on the level; otherwise, false
   
   **Return type**
   bool

property items
   If level 0 or 1, the list of edges or nodes, respectively. If level 2, the IncidenceStore
   
   **Return type**
   IncidenceStore | array

property level
   0 = Edges, 1 = Nodes, 2 = Incidences
   
   **Return type**
   int

   **Type**
   The type of store

property memberships
   See memberships
   
   **Return type**
   dict

property properties
   All properties for objects in the HypergraphView. Same as to_dataframe.
   
   **Return type**
   pd.DataFrame
property property_store
    PropertyStore
    
    Return type
    PropertyStore

set_defaults(defaults_dict)
    Creates or updates default values in PropertyStore associated with this HypergraphView. Does not over-write existing user-defined properties

    Parameters
    defaults_dict (dict) – Dictionary of prop_names to their default values

    Return type
    None

property to_dataframe
    Dataframe of properties (user defined and default) for all items in the HypergraphView. Creates a properties dataframe of non-user-defined items with default values. Combines user-defined and non-user-defined properties into one dataframe.

    Return type
    pd.DataFrame

property user_defined_properties
    User-defined properties. Does not include items in the HypergraphView that the user has not explicitly defined properties for.

    Return type
    pd.DataFrame

class classes.IncidenceStore(data)
    Bases: object

    Incidence store object that stores and accesses (multi) incidences with standard methods.

    Parameters
    data (Two column pandas dataframe of edges and nodes, respectively.)

    collapse_identical_elements(level, use_keys=None)

property data

property dimensions
    The number of distinct edges and nodes in that order

    Returns
    Tuple of size two of (number of unique edges, number of unique nodes).

    Return type
    tuple of ints

property edges
    Returns an array of edge names from the incidence pairs

    Returns
    Returns an array of edge names

    Return type
    array

5.5. HyperNetX Packages 45
property elements

equivalence_classes(level=0)

property memberships

neighbors(level, key)
    Returns elements or memberships depending on level.

    Parameters
    • level (int) – Level indicator for finding either elements or memberships. For level 0
      (elements), returns nodes in the edge. For level 1 (memberships), returns edges containing
      the node.
    • key (int or str) – Name of node or edge depending on level.

    Returns
    Elements or memberships (depending on level) of a given edge or node, respectively.

    Return type
    list

property nodes
    Returns an array of node names from the incidence pairs

    Returns
    Returns an array of node names

    Return type
    array

restrict_to(level, items, inplace=False)
    returns IncidenceStore of subset of incidence store restricted to pairs with items in the given level
    Will return with same data or deepcopy depending on inplace

    Parameters
    • level (int) – Level indicator for finding either elements or memberships. For level 0
      (elements), returns nodes in the edge. For level 1 (memberships), returns edges containing
      the node.
    • items (list) – List of uids to be removed from level
    • inplace (bool, optional) – whether to replace self, by default False

    Returns
    subset of incidence store given a restriction.

    Return type
    list

class classes.PropertyStore(data=None, default_weight=1)

    Bases: object
    Class for storing properties of a collection of edges, nodes, or incidences.
    Properties will be stored in a pandas dataframe.

    copy(deep=False)
    Create a copy of the PropertyStore. If deep=True, create a copy of the underlying data table. Otherwise,
    use the same underlying data table from the original PropertyStore
Parameters
  deep (bool, optional, default=False)

Return type
  PropertyStore

property default_properties: dict
  Returns copy of default dictionary of properties

Returns
  Dictionary of properties automatically given to objects either in the property store if no user
defined values have been assigned to them or objects that have not yet been added to the
Property Store.

Return type
  dict

get_properties(uid) \rightarrow dict
  Get all properties of an item

Parameters
  uid (Hashable) – uid is the index used to fetch all its properties

Returns
  Output dictionary containing all properties of the uid. {named property: property
  value, ..., properties: {property name: property value}}

Return type
  dict

See also:
  get_property, set_property

get_property(uid, prop_name) \rightarrow Any
  Get a property of an item

Parameters
  • uid (Hashable) – uid is the index used to fetch its property

  • prop_name (str | int) – name of the property to get

Returns
  • out (Any) – value of the property

  • None – if property not found

See also:
  get_properties, set_property

property properties: DataFrame
  Properties assigned to all items in the underlying data table

Returns
  a dataframe with the following columns:
  uid, weight, properties, <optional props> or level, id, weight, properties, <optional props>

Return type
  pandas.DataFrame
HyperNetX Documentation, Release 2.3.5

**set_defaults**(defaults) → None

Set default values for properties

Parameters
defaults (dict)

Return type
None

**set_properties**(uid, props) → None

Parameters

• uid (Hashable) – uid is the index used to set its property

• props (a dictionary containing user-defined properties)

Return type
None

See also:
get_property, get_properties, set_property

**set_property**(uid, prop_name, prop_val) → None

Set a property of an item in the ‘properties’ collection

Parameters

• uid (Hashable) – uid is the index used to set its property

• prop_name (str | int) – name of the property to set

• prop_val (any) – value of the property to set

Return type
None

See also:
get_property, get_properties, set_properties

Submodules

classes.factory module
classes.factory.create_df(DFP, uid_cols=None, level=0, use_index=False, weight_prop=None, default_weight=1.0, misc_properties_col=None, aggregation_methods=None)

classes.factory.dataframe_factory_method(DF, level, use_indices=False, uid_cols=None, misc_properties_col='misc_properties', weight_col='weight', default_weight=1.0, aggregate_by={})

This function creates a pandas dataframe in the correct format given a pandas dataframe of either cell, node, or edge properties.

Parameters

• DF (dataframe) – dataframe of properties for either incidences, edges, or nodes

• level (int) – Level to specify the type of data the dataframe is for: 0 for edges, 1 for nodes, and 2 for incidences (cells).
• **uid_cols** *(list of str or int)* – column index (or name) in pandas.dataframe used for (hyper)edge, node, or incidence (edge, node) IDs.

**misc_properties_col**
[(optional) int | str, default = None] Column of property dataframes with dtype=dict. Intended for variable length property dictionaries for the objects.

**weight_col**
[(optional) str, default = None] Name of property in edge_properties to use for weight.

**default_weight**
[(optional) int | float, default = 1] Used when edge weight property is missing or undefined.

**aggregate_by**
[(optional) dict, default = {}] By default duplicate incidences will be dropped unless specified with aggregation_methods. See pandas.DataFrame.agg() methods for additional syntax and usage information. An example aggregation method is `{‘weight’: ‘sum’}` to sum the weights of the aggregated duplicate rows.

**Return type**
Pandas Dataframe of the property store in the correct format for HNX.

```python
classes.factory.dict_to_incidence_store_df(D, level, use_indices=False, uid_cols=None, misc_properties_col='misc_properties', weight_col='weight', default_weight=1.0, aggregate_by={})
```

This function creates a pandas dataframe in the correct format given a dictionary of either cell, node, or edge properties.

**Parameters**
- **D** *(dictionary)* – dictionary of properties for either incidences, edges, or nodes
- **level** *(int)* – Level to specify the type of data the dataframe is for: 0 for edges, 1 for nodes, and 2 for incidences (cells).
- **uid_cols** *(list of str or int)* – column index (or name) in pandas.dataframe used for (hyper)edge, node, or incidence (edge, node) IDs.
- **misc_properties_col** *(optional) int | str, default = None* – Column of property dataframes with dtype=dict. Intended for variable length property dictionaries for the objects.
- **weight_col** *(optional) str, default = None,* – Name of property in edge_properties to use for weight.
- **default_weight** *(optional) int | float, default = 1* – Used when edge weight property is missing or undefined.
- **aggregate_by** *(optional) dict, default = {}* – By default duplicate incidences will be dropped unless specified with aggregation_methods. See pandas.DataFrame.agg() methods for additional syntax and usage information. An example aggregation method is `{‘weight’: ‘sum’}` to sum the weights of the aggregated duplicate rows.

**Return type**
Pandas Dataframe of the property store in the correct format for HNX.

```python
classes.factory.dict_to_incidence_store_df(D)
```
classes.factory.list_factory_method(L, level, use_indices=False, uid_cols=None, misc_properties_col='misc_properties', weight_col='weight', default_weight=1.0, aggregate_by={})

This function creates a pandas dataframe in the correct format given a list of lists to be used as the cell property store dataframe.

**Parameters**

- **L (list of lists)** – list of lists representing the nodes in each hyperedge.
- **level (int)** – Level to specify the type of data the dataframe is for: 0 for edges, 1 for nodes, and 2 for incidences (cells).
- **uid_cols (list of str or int)** – column index (or name) in pandas.dataframe used for (hyper)edge, node, or incidence (edge, node) IDs.
- **misc_properties_col ((optional) int | str, default = None)** – Column of property dataframes with dtype=dict. Intended for variable length property dictionaries for the objects.
- **weight_col ((optional) str, default = None)** – Name of property in edge_properties to use for weight.
- **default_weight ((optional) int | float, default = 1)** – Used when edge weight property is missing or undefined.
- **aggregate_by ((optional) dict, default = {})** – By default duplicate incidences will be dropped unless specified with *aggregation_methods*. See pandas.DataFrame.agg() methods for additional syntax and usage information. An example aggregation method is `{‘weight’: ‘sum’}` to sum the weights of the aggregated duplicate rows.

**Return type**
Pandas Dataframe of the property store in the correct format for HNX.

classes.factory.mkdict(x)

classes.factory.ndarray_factory_method(arr, level, *args, **kwargs)

**classes.hyp_view module**

class classes.hyp_view.HypergraphView(incidence_store, level, property_store=None)

**Bases:** object

Wrapper for Property and Incidence Stores holding structural and metadata for hypergraph. Provides methods matching EntitySet methods in previous versions. Only nodes and edges in the Incidence Store will be seeable in this view.

**property dataframe**

All properties for objects in the HypergraphView. Same as to_dataframe.

**Return type**
pd.DataFrame

**property default_weight**

Default weight for an edge, node, or incidence

**Return type**
int | float
property elements
  See elements
  Return type
dict

property incidence_dict
  incidence dictionary
  Return type
dict | None

property incidence_store
  IncidenceStore
  Return type
  IncidenceStore

is_empty()
  Returns true if HypergraphView has no edges, nodes, or incidences depending on the level; otherwise, false
  Return type
  bool

property items
  If level 0 or 1, the list of edges or nodes, respectively. If level 2, the IncidenceStore
  Return type
  IncidenceStore | array

property level
  0 = Edges, 1 = Nodes, 2 = Incidences
  Return type
  int
  Type
  The type of store

property memberships
  See memberships
  Return type
dict

property properties
  All properties for objects in the HypergraphView. Same as to_dataframe.
  Return type
  pd.DataFrame

property property_store
  PropertyStore
  Return type
  PropertyStore

set_defaults(defaults_dict)
  Creates or updates default values in PropertyStore associated with this HypergraphView. Does not over-
  write existing user-defined properties
Parameters

**defaults_dict** *(dict)* – Dictionary of prop_names to their default values

Return type

None

**property to_dataframe**

Dataframe of properties (user defined and default) for all items in the HypergraphView. Creates a properties dataframe of non-user-defined items with default values. Combines user-defined and non-user-defined properties into one dataframe.

Return type

pd.DataFrame

**property user_defined_properties**

User-defined properties. Does not include items in the HypergraphView that the user has not explicitly defined properties for.

Return type

pd.DataFrame

**classes.hypergraph module**

class classes.hypergraph.Hypergraph*(setsystem=None, default_cell_weight=1, edge_col=0, node_col=1, cell_weight_col='weight', misc_cell_properties_col=None, aggregate_by='first', properties=None, misc_properties_col=None, weight_prop_col='weight', default_weight: float | int = 1, edge_properties=None, misc_edge_properties_col=None, edge_weight_prop_col='weight', default_edge_weight=1, node_properties=None, misc_node_properties_col=None, node_weight_prop_col='weight', default_node_weight=1, name=None, **kwargs)*

Bases: object

Parameters

- **setsystem** *(pandas.DataFrame, dict of iterables, dict of dicts, list of iterables, numpy.ndarray, optional, default=None)* – See SetSystem below for additional setsystem requirements.
- **edge_col** *(str / int, optional, default=0)* – column index (or name) in pandas.DataFrame, used for (hyper)edge ids. Only used when setsystem is a pandas.DataFrame
- **node_col** *(str / int, optional, default=1)* – column index (or name) in pandas.DataFrame, used for node ids. Only used when setsystem is a pandas.DataFrame
- **cell_weight_col** *(str | int, optional, default="weight")* – column index (or name) in pandas.DataFrame used for referencing cell weights. For a dict of dicts, it will be used as a key in the nested dictionary of properties. These are the same as edge dependent node weights and will populate the incidence matrix when weights=True.
- **default_cell_weight** *(int / float, optional, default=1)* – All incidence pairs in the Hypergraph are assigned a default weight if weight is not specified in the setsystem.
- **misc_cell_properties_col** *(str | int, optional, default=None)* – Used for Pandas Dataframe with one column containing dictionaries of properties. Useful if objects have diverse property sets. Ignored for other setsystem types.
• **properties** (*pd.DataFrame | dict, optional, default=None*) – Concatenation/union of edge_properties and node_properties. By default, the object id is used and should be the first column of the dataframe, or key in the dict. If there are nodes and edges with the same ids but distinct properties then separate them and use the edge_properties and node_properties keywords.

• **weight_prop_col** (*str, optional, default=None*) – Name of property in properties to use for weight

• **default_weight** (*int | float, optional, default=1*) – Used when weight property is missing or undefined

• **edge_properties** (*pd.DataFrame | dict, optional, default=None*) – Properties associated with edge ids. If a dataframe, the first column must be the names of the edges. First column of dataframe or keys of dict link to edge ids in setsystem.

• **edge_weight_prop_col** (*str, optional, default=None*) – Name of property in edge_properties to use for weight.

• **default_edge_weight** (*int | float, optional, default=1*) – Used when edge weight property is missing or undefined.

• **node_properties** (*pd.DataFrame | dict, optional, default=None*) – Properties associated with node ids. If a dataframe, the first column must be the names of the nodes. First column of dataframe or keys of dict link to nodes ids in setsystem.

• **node_weight_prop_col** (*str, optional, default=None*) – Name of property in node_properties to use for weight.

• **default_node_weight** (*int | float, optional, default=1*) – Used when node weight property is missing or undefined.

• **misc_properties_col** (*str | int, optional, default=None*) – Used for properties, edge_properties, and node_properties Pandas Dataframes with one column containing dictionaries of properties. Useful if objects have diverse property sets. Ignored for other setsystem types.

• **name** (*str, optional, default=None*) – Name assigned to hypergraph

### Hypergraphs in HNX 2.3

An hnx.Hypergraph \( H = (V,E) \) references a pair of disjoint sets: \( V = \) nodes (vertices) and \( E = \) (hyper)edges.

HNX allows for multi-edges by distinguishing edges by their unique identifiers instead of their contents. For example, if \( V = \{1,2,3\} \) and \( E = \{e_1,e_2,e_3\} \), where \( e_1 = \{1,2\} \), \( e_2 = \{1,2\} \), and \( e_3 = \{1,2,3\} \), the edges \( e_1 \) and \( e_2 \) contain the same set of nodes and yet are distinct and are distinguishable within \( H = (V,E) \).

New as of version 2.3, HNX provides methods to easily store and access additional metadata such as cell, edge, and node weights. Metadata associated with all edges, nodes, and (edge,node) incidence pairs stored in the hypergraph are viewable using:

```python
>>> H.edges.to_dataframe
>>> H.nodes.to_dataframe
>>> H.incidences.to_dataframe
```

The fundamental object needed to create a hypergraph is a **setsystem**. The setsystem defines the many-to-many relationships between edges and nodes in the hypergraph. Properties for the incidence pairs are defined within the setsystem. Properties for the edges and nodes are defined with separate Pandas DataFrames or dictionaries.
A hypergraph is defined by its relationships. While the nodes and edges are distinct objects with their own properties, it is only when they are in a relationship (i.e. incidence pair) that nodes and edges are viewable within the hypergraph structure. Consequently, hypergraph metrics and combinatorics do not use “isolated” nodes or “empty” edges. For example, while node properties could contain any number of node identifiers, only nodes belonging to an edge in the hypergraph are counted when computing the size and shape of the hypergraph.

**SetSystems**

There are five types of setsystems currently accepted by the library.

1. **iterable of iterables**: Barebones hypergraph, which uses Pandas default indexing to generate hyperedge ids. Elements must be hashable.

   ```python
code
>>> H = hnx.Hypergraph(list_of_lists)
```

2. **dictionary of iterables**: The most basic way to express many-to-many relationships providing edge ids. The elements of the iterables must be hashable.

   ```python
code
>>> scenes_dictionary = {
    0: {'FN': 'TH'},
    1: {'TH': 'JV'},
    2: {'BM': 'FN', 'JA'},
    3: {'JV': 'JU', 'CH', 'BM'},
    4: {'JU': 'CH', 'BR', 'CN', 'CC', 'JV', 'BM'},
    5: {'TH': 'GP'},
    6: {'GP', 'MP'},
    7: {'MA', 'GP'},
    8: {'FN', 'TH'}
}
>>> H = hnx.Hypergraph(scenes_dictionary)
```

3. **dictionary of dictionaries**: allows cell properties to be assigned to a specific (edge, node) incidence. This is particularly useful when there are variable length dictionaries assigned to each pair.

   ```python
code
>>> nested_dictionary = {
    0: {'FN': {'time': 'early', 'weight': 7}, 'TH': {'time': 'late'}},
    1: {'TH': {'subject': 'war'}, 'JV': {'observed_by': 'someone'}},
    2: {'BM': {}, 'FN': {}, 'JA': {'role': 'policeman'}},
    3: {'JV': {'was_carrying': 'stick'}, 'JU': {}, 'CH': {}, 'BM': {'state': 'intoxicated', 'color': 'pinkish'}},
    4: {'JU': {'weight': 15}, 'CH': {}, 'BR': {'state': 'worried'}, 'CN': {}, 'CC': {}, 'JV': {}, 'BM': {}},
    5: {'TH': {}, 'GP': {}},
    6: {'GP': {}, 'MP': {}},
    7: {'MA': {}, 'GP': {'accompanied_by': 'dog', 'weight': 15, 'was_singing': 'Frère Jacques'}}
}
>>> H = hnx.Hypergraph(nested_dictionary)
```

4. **pandas.DataFrame** For large datasets and for datasets with cell properties it is most efficient to construct a hypergraph directly from a pandas.DataFrame. Incidence pairs are in the first two columns. Cell properties shared by all incidence pairs can be placed in their own column of the dataframe. Variable length dictionaries of cell properties particular to only some of the incidence pairs may be placed in a single column of the dataframe. Representing the data above as a dataframe df:
The first row of the dataframe is used to reference each column.

```python
>>> import pandas as pd
>>> d = {
...    'col1': [e1, e1, e2],
...    'col2': [1, 2, 1],
...    'w': [0.5, 0.1, 0.52],
...    'col3': [{'name': 'related_to'},
...              {'name': 'related_to', 'startdate': '05.13.2020'},
...              {'name': 'owned_by'}]
...}
>>> df = pd.DataFrame(d)
>>> H = hnx.Hypergraph(df, edge_col="col1", node_col="col2",
                     cell_weight_col="w", misc_cell_properties_col="col3")
```

5. `numpy.ndarray` For homogeneous datasets given in a $n \times 2$ ndarray a pandas dataframe is generated. In this case, the constructor will only accept properties for the edges and nodes using the edge and node uids listed in the array, although incidence properties can be added after construction:

```python
>>> import numpy as np
>>> np_array = np.array([[A, a], [A, b], [A, c], [B, a], [B, d], [C, c], [C, d]])
>>> H = hnx.Hypergraph(np_array)
>>> H.incidences[('A', 'a')].color = 'red'
>>> H.dataframe
```

### Edge and Node Properties

Properties specific to a single edge or node are passed through the keywords: `edge_properties`, `node_properties`, or `properties`. Properties may be passed as dataframes or dictionaries. The first column or index of the dataframe or the keys of the dictionary correspond to the edge and/or node identifiers. If identifiers are shared among edges and nodes, or are distinct for edges and nodes, properties may be combined into a single object and passed to the `properties` keyword. For example:

<table>
<thead>
<tr>
<th>uid</th>
<th>weight</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>5.0</td>
<td>{'type': 'event'}</td>
</tr>
<tr>
<td>e2</td>
<td>0.52</td>
<td>{'name': 'owned_by'}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1.2</td>
<td>{'color': 'red'}</td>
</tr>
<tr>
<td>2</td>
<td>.003</td>
<td>{'name': 'Fido', 'color': 'brown'}</td>
</tr>
<tr>
<td>3</td>
<td>1.0</td>
<td>{}</td>
</tr>
</tbody>
</table>

A properties dictionary should have the format:
dp = {uid1 : {prop1:val1, prop2:val2, ...},
      uid2 : {...},
      ...
}

Weights

The default key for cell and object weights is “weight”. The default value is 1. Weights may be assigned from a column in the dataframe by specifying the column and/or a new default in the constructor using `cell_weight_col` and `default_cell_weight` for incidence pairs, and using `edge_weight_prop_col`, `default_edge_weight` for edges, `node_weight_prop_col`, `default_node_weight` for nodes, and `weight_prop_col`, `default_weight` for a shared property dataframe.

`add_edge(edge_uid, inplace=True, **attr)`

Add a single edge with attributes to edge properties. Does not add an incidence to the hypergraph.

Parameters

- **edge_uid** *(int / str)* – edge_uid
- **inplace** *(bool, default=True)* – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.
- **attr** *(dict, optional)* – Properties to add to edges as key=value pairs.

Return type

`Hypergraph`

`add_edges_from(edge_uids, inplace=True)`

Add a collection of edges with attributes to edge properties. Does not add an incidence to the hypergraph.

Parameters

- **edge_uids** *(list[int | str] | list[tuple[int | str, dict]] | list[int | str | tuple[int | str, dict]])* – edge_uids must be a list of uids and/or tuples of the form (uid, data) where data is dictionary
- **inplace** *(bool, default=True)* – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.

Return type

`Hypergraph`

`add_incidence(edge_uid, node_uid, inplace=True, **attr)`

Add a single incidence with attributes to Hypergraph.

Parameters

- **edge_uid** *(int | str)* – edge_uid
- **node_uid** *(int | str)* – node_uid
- **inplace** *(bool, optional, default=True)* – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.
- **attr** *(dict, optional)* – Properties to add to incidences as key=value pairs.

Returns

Hypergraph with incidences added.

Return type

`Hypergraph`
add_incidences_from(incidences, inplace=True)

Adds a collection of incidences to Hypergraph

Parameters
- incidences (list[str | int, str | int], list[tuple[str | int, str | int], dict[str, Any]]) – Incidence pairs must be a list of uids of the form (edge_uid, node_uid) and/or tuples of the form (edge_uid, node_uid, data) where data is a dictionary.
- inplace (bool, optional, default=True) – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.

Returns
Hypergraph with incidences added.

Return type
Hypergraph

add_node(node_uid, inplace=True, **attr)

Add a single node with attributes to node properties. Does not add an incidence to the hypergraph.

Parameters
- node_uid (int | str) – node_uid
- inplace (bool, default=True) – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.
- **attr (dict, optional) – Properties to add to edges as key=value pairs.

Return type
Hypergraph

add_nodes_from(node_uids, inplace=True)

Add a collection of nodes with attributes to nodes properties. Does not add an incidence to the hypergraph.

Parameters
- node_uids (list[int | str | tuple[int | str, dict]], list[int | str | tuple[int | str, dict]]) – node_uids must be a list of uids and/or tuples of the form (uid, data) where data is dictionary.
- inplace (bool, default=True) – If True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.

Return type
Hypergraph

add_nodes_to_edges(edge_dict, inplace=True)

Adds a collection of incidences to Hypergraph

Parameters
- edge_dict (dict[str, list[str | int | dict[str, dict]]) – The edge dictionary must be a dictionary of edges as the keys and a list of nodes or a dictionary of nodes to properties as the values.
- inplace (bool, default=True) – If True, changes the existing. Otherwise, creates a new Hypergraph with the requested changes.

Returns
Hypergraph with the updated edges and their newly added nodes
Return type
Hypergraph

adjacency_matrix(s=1, index=False)
Returns the $s$-adjacency matrix for the hypergraph.

Parameters
- $s$ (int, optional, default=1)
- index (boolean, optional, default=False) – If True, returns both the adjacency matrix and an array containing the row and column index of node_uids

Returns
- adjacency matrix (scipy.sparse.csr_matrix)
- node indexes (np.ndarray) – an np.ndarray containing the row and column index of node_uids.

auxiliary_matrix(s=1, node=True, index=False)
The unweighted $s$-auxiliary matrix for hypergraph

Parameters
- $s$ (int, optional, default=1)
- node (bool, optional, default=True) – whether to return based on node or edge adjacencies
- index (bool, optional, default=False) – If True, returns both the auxiliary matrix and an array containing the row and column index of node or edge_uids

Returns
- auxiliary matrix (scipy.sparse.csr_matrix) – Node/Edge adjacency matrix with empty rows and columns removed
- index (np.ndarray) – row and column index of node or edge uids

bipartite(keep_data=False, directed=False)
Creates a bipartite NetworkX graph from hypergraph. The nodes and (hyper)edges of hypergraph become the nodes of bipartite graph. For every (hyper)edge $e$ in the hypergraph and node $v$ in $e$ there is an edge $(e,v)$ in the graph.

Parameters
- keep_data (bool, optional, default = False) – If True the node and edge data from the hypergraph will be added to the NetworkX graph
- directed (bool, optional, default = False) – If True the graph edges will be directed so that the hypergraph edges are the sources and the hypergraph nodes are the targets

Return type
networkx.Graph or DiGraph

clone(name=None)
Create a deep copy of the hypergraph

Parameters
- name (str, optional, default = None)

Return type
Hypergraph
collapse_edges(name=None, use_uids=None, use_counts=False, return_counts=True,
  return_equivalence_classes=False, aggregate_edges_by=None,
  aggregate_cells_by=None)

Returns a new hypergraph by collapsing edges.

Parameters

- **name** (str, optional, default = None)
- **use_uids** (list, optional, default = None) – Specify the edge identifiers to use as representatives for a single equivalence class. If two identifiers occur in the same equivalence class, the first one found will be used.
- **use_counts** (boolean, optional, default = False) – Rename the equivalence class representatives as <uid>:<size of class>
- **return_counts** (bool, optional, default = True) – Add the size of the equivalence class to the properties associated to the representative in the collapsed hypergraph using keyword: eclass_size
- **return_equivalence_classes** (boolean, optional, default = False) – Returns a dictionary of edge equivalence classes keyed by a representative from each class
- **aggregate_edges_by** (dict, optional, default = {'weight':'sum'}) – dictionary of aggregation methods keyed by column names in the properties dataframes, does not apply to misc_properties. Defaults to ‘first’ on unlisted columns. See pandas.core.groupby.DataFrameGroupBy.agg for usage examples with dictionaries
- **aggregate_cells_by** (dict, optional, default = {'weight':'sum'}) – dictionary of aggregation methods keyed by column names in the properties dataframes, does not apply to misc_properties. Defaults to ‘first’ on unlisted columns. See pandas.core.groupby.DataFrameGroupBy.agg for usage examples with dictionaries

Return type

Hypergraph

Notes

Collapses the edges of Hypergraph. Two edges are duplicates if their respective elements are the same. Using this as an equivalence relation, the uids of the edges are partitioned into equivalence classes. A single member of the equivalence class is chosen to represent the class.

Example

```python
>>> data = {'E1': ('a', 'b'), 'E2': ('a', 'b')}
>>> h = Hypergraph(data)
>>> h.incidence_dict
{'E1': ['a', 'b'], 'E2': ['a', 'b']}
>>> h.collapse_edges().incidence_dict
{'E1': ['a', 'b']}
>>> h.collapse_edges(use_counts=True).incidence_dict
{'E1': ['a', 'b']}
>>> h.collapse_edges().properties.to_dict(orient='records')
[[{'weight': 2.0, 'misc_properties': {}}, {'weight': 2.0, 'misc_properties': {}}]]
```
collapse_nodes

(name=None, use_uids=None, use_counts=False, return_counts=True,
  return_equivalence_classes=False, aggregate_nodes_by=None,
  aggregate_cells_by=None)

Returns a new hypergraph by collapsing nodes.

Parameters

- **name** *(str, optional, default = None)*
- **use_uids** *(list, optional, default = None)* – Specify the node identifiers to use as representatives for a single equivalence class. If two identifiers occur in the same equivalence class, the first one found will be used.
- **use_counts** *(boolean, optional, default = False)* – Rename the equivalence class representatives as <uid>:<size of class>
- **return_counts** *(bool, optional, default = True)* – Add the size of the equivalence class to the properties associated to the representative in the collapsed hypergraph using keyword: eclass_size
- **return_equivalence_classes** *(boolean, optional, default = False)* – Returns a dictionary of edge equivalence classes keyed by a representative from each class
- **aggregate_nodes_by** *(dict, optional, default = {"weight": 'sum'})* – Dictionary of aggregation methods keyed by column names in the properties dataframes, does not apply to misc_properties. Defaults to ‘first’ on unlisted columns. See pandas.core.groupby.DataFrameGroupBy.agg for usage examples with dictionaries
- **aggregate_cells_by** *(dict, optional, default = {"weight": 'sum'})* – Dictionary of aggregation methods keyed by column names in the properties dataframes, does not apply to misc_properties. Defaults to ‘first’ on unlisted columns. See pandas.core.groupby.DataFrameGroupBy.agg for usage examples with dictionaries

Return type

Hypergraph

Notes

Collapses the nodes of Hypergraph. Two nodes are duplicates if their respective memberships are the same. Using this as an equivalence relation, the uids of the nodes are partitioned into equivalence classes. A single member of the equivalence class is chosen to represent the class.

Example

```python
>>> data = {'E1': ('a', 'b'), 'E2': ('a', 'b')}
>>> h = Hypergraph(data)
>>> h.incidence_dict
{'E1': ['a', 'b'], 'E2': ['a', 'b']}
>>> h.collapse_nodes().incidence_dict
{'E1': ['a'], 'E2': ['a']}
>>> h.collapse_nodes(use_counts=True).incidence_dict
{'E1: ['a:2'], 'E2: ['a:2']}
>>> h.collapse_nodes().properties.to_dict(orient='records')
[{{'weight': 2.0, 'misc_properties': {}}}, {{'weight': 2.0, 'misc_properties': {}}}
```
collapse_nodes_and_edges(name=None, use_edge_uids=None, use_node_uids=None, use_counts=False, return_counts=True, return_equivalence_classes=False, aggregate_nodes_by=None, aggregate_edges_by=None, aggregate_cells_by=None)

Returns a new hypergraph by collapsing nodes and edges.

Parameters

- **name** (str, optional, default = None)
- **return_equivalence_classes** (boolean, optional, default = False) – Returns a dictionary of edge equivalence classes keyed by a representative from each class
- **use_edge_uids** (list, optional, default = None) – Specify the edge and node identifiers to use as representatives for a single equivalence class. If two identifiers occur in the same equivalence class, the first one found will be used.
- **use_node_uids** (list, optional, default = None) – Specify the edge and node identifiers to use as representatives for a single equivalence class. If two identifiers occur in the same equivalence class, the first one found will be used.
- **use_counts** (boolean, optional, default = False) – Rename the equivalence class representatives as <uid>:<size of class>
- **return_counts** (bool, optional, default = True) – Add the size of the equivalence class to the properties associated to the representative in the collapsed hypergraph using keyword: eclass_size
- **aggregate_nodes_by** (optional) – default = {'weight'='sum'}, all Method to combine duplicate rows of data for the same uids
- **aggregate_edges_by** (optional) – default = {'weight'='sum'}, all Method to combine duplicate rows of data for the same uids
- **aggregate_cells_by** (optional) – default = {'weight'='sum'}, all Method to combine duplicate rows of data for the same uids

Returns

- **new hypergraph** (Hypergraph)
- **node equivalence classes** (dict)
- **edge equivalence classes** (dict)

Notes

Collapses the Nodes and Edges of Hypergraph. Two nodes(edges) are duplicates if their respective memberships(elements) are the same. Using this as an equivalence relation, the uids of the nodes(edges) are partitioned into equivalence classes. A single member of the equivalence class is chosen to represent the class.
Example

```python
>>> data = {'E1': ('a', 'b'), 'E2': ('a', 'b')}
>>> h = Hypergraph(data)
>>> h.incidence_dict
{'E1': ['a', 'b'], 'E2': ['a', 'b']}
>>> h.collapse_nodes_and_edges().incidence_dict
{'E1:2': ['a:2']}
```

component_subgraphs

```python
(returnsingletons=False, name=None)
```

Same as `s_components_subgraphs()` with `s=1`. Returns iterator.

See also:

`s_component_subgraphs`

components

```python
(edges=False)
```

Same as `s_connected_components()` with `s=1`, but nodes are returned by default. Return iterator.

See also:

`s_connected_components`

connected_component_subgraphs

```python
(returnsingletons=True, name=None)
```

Same as `s_component_subgraphs()` with `s=1`. Returns iterator.

See also:

`s_component_subgraphs`

connected_components

```python
(edges=False)
```

Same as `s_connected_components()` with `s=1`, but nodes are returned by default. Return iterator.

See also:

`s_connected_components`

property dataframe

Returns dataframe of incidence properties as dataframe with edges and nodes in columns.

Return type
pandas.DataFrame

degree

```python
(node_uid, s=1, max_size=None)
```

The number of edges of size at least `s` and at most `max_size` that contain the node.

Parameters

- `node_uid` (hashable) – Identifier for the node.
- `s` (int, optional, default=1) – The smallest size (must be positive) of an edge to consider in degree.
- `max_size` (int, optional, default=None) – The largest size (must be positive) of edge to consider in degree.

Returns

The number of edges of size at least `s` and at most `max_size` that contain the node.
Return type
int
diameter\((s=1)\)
Returns the length of the longest shortest s-walk between nodes in hypergraph

Parameters
\(s\) (int, optional, default=1)

Returns
diameter

Return type
int

Raises
HyperNetXError – If hypergraph is not s-edge-connected

Notes
Two nodes are s-adjacent if they share s edges. Two nodes \(v_{\text{start}}\) and \(v_{\text{end}}\) are s-walk connected if there is a sequence of nodes \(v_{\text{start}}, v_1, v_2, \ldots v_{n-1}, v_{\text{end}}\) such that consecutive nodes are s-adjacent. If the graph is not connected, an error will be raised.
difference\((other, name=None)\)
Hypergraph obtained by restricting to incidences in self but not in other.

Parameters
- other (Hypergraph)
- name (str, optional, default = None)

Return type
Hypergraph
dim\((edge)\)
Same as size(edge) - 1()

Parameters
edge (hashable) – The uid of an edge in the hypergraph

Return type
int
distance\((source, target, s=1)\)
Returns the shortest s-walk distance between two nodes in the hypergraph.

Parameters
- source (str | int) – a node in the hypergraph
- target (str | int) – a node in the hypergraph
- s (positive int, optional, default=1) – the number of edges

Returns
s-walk distance

Return type
int
See also:

edge_distance

Notes

The s-distance is the shortest s-walk length between the nodes. An s-walk between nodes is a sequence of nodes that pairwise share at least s edges. The length of the shortest s-walk is 1 less than the number of nodes in the path sequence.

Uses the networkx shortest_path_length method on the graph generated by the s-adjacency matrix.

dual\((name=None, share\_properties=True)\)

Constructs a new hypergraph with roles of edges and nodes of hypergraph reversed.

Parameters

- **name** (hashable, optional, default=None)
- **share\_properties** (bool, optional, default=True) – Whether to tie the edge and node properties of objects in the dual to objects in the hypergraph. If True, a change to edge and node properties in one will be reflected in the other.

Return type

Hypergraph

edge_adjacency_matrix\((s=1, index=False)\)

Returns the s-adjacency matrix for the dual hypergraph.

Parameters

- **s** (int, optional, default=1)
- **index** (boolean, optional, default=False) – If True, returns both the adjacency matrix and an array containing the row and column index of edge_uids

Returns

- **edge adjacency matrix** (scipy.sparse.csr_matrix)
- **edge indexes** (np.ndarray) – an np.ndarray containing the row and column index of edge_uids.

Notes

This is also the adjacency matrix for the line graph. Two edges are s-adjacent if they share at least s nodes.

edge_diameter\((s=1)\)

Returns the length of the longest shortest s-walk between edges in the hypergraph

Parameters

- **s** (int, optional, default=1)

Returns

edge_diameter

Return type

int

Raises

HyperNetXError – If hypergraph is not s-edge-connected
Notes

Two edges are s-adjacent if they share s nodes. Two nodes e_start and e_end are s-walk connected if there is a sequence of edges e_start, e_1, e_2, ... e_n-1, e_end such that consecutive edges are s-adjacent. If the graph is not connected, an error will be raised.

`edge_diameters(s=1)`

Returns the edge diameters of the s_edge_connected component subgraphs in the hypergraph.

Parameters

- `s` (int, optional, default=1)

Returns

- maximum diameter, list of diameters, list of component — maximum diameter, list of diameters (List of edge_diameters for s-edge component subgraphs in hypergraph), list of component (List of the edge uids in the s-edge component subgraphs)

Return type

tuple[int, list, list]

`edge_distance(source, target, s=1)`

Returns the shortest s-walk distance between two edges in the hypergraph.

Parameters

- `source` (str | int) — an edge in the hypergraph
- `target` (str | int) — an edge in the hypergraph
- `s` (positive int, optional, default=1) — the number of intersections between pairwise consecutive edges

Returns

- s-walk distance — The shortest s-walk edge distance. A shortest s-walk is computed as a sequence of edges; the s-walk distance is the number of edges in the sequence minus 1. If no such path exists returns np.inf.

Return type

tuple[int, list, list]

See also:

distance

Notes

The s-distance is the shortest s-walk length between the edges. An s-walk between edges is a sequence of edges such that consecutive pairwise edges intersect in at least s nodes. The length of the shortest s-walk is 1 less than the number of edges in the path sequence.

Uses the networkx shortest_path_length method on the graph generated by the s-edge_adjacency matrix.

`edge_neighbors(edge, s=1)`

The edges in hypergraph which share s nodes(s) with edge.

Parameters

- `edge` (hashable) — uid for an edge in hypergraph
- `s` (int, optional, default=1) — Minimum number of nodes shared by neighbors edge node.
Returns
a list of edge neighbors

Return type
list

edge_size_dist()
Returns the size for each edge.

Returns
a list of sizes of each edge.

Return type
list

property edges
Object associated with edges.

Return type
HypergraphView

equivalence_classes(edges=True)
Returns the equivalence classes created by collapsing edges or nodes.

Parameters
edges (bool, optional, default=True) – If True collapses edges, otherwise collapses nodes.

Returns
A list of sets of edges or nodes

Return type
list

See also:
collapse_edges, collapse_nodes, collapse_nodes_and_edges

classmethod from_bipartite(B, node_id=1, name=None, **kwargs)
Static method creates a Hypergraph from a NetworkX bipartite graph. Still to come: capturing edge and node properties from the graph for use in the hypergraph.

Parameters
• B (nx.Graph()) – A networkx bipartite graph. Each node in the graph has a property ‘bipartite’ taking the value of 0 or 1 indicating a 2-coloring of the graph.
• node_id (int) – bipartite value assigned to graph nodes that will be hypergraph edges
• name (hashable, optional)

Return type
Hypergraph
Notes

A partition for the nodes in a bipartite graph generates a hypergraph.

```python
>>> import networkx as nx
>>> B = nx.Graph()
>>> B.add_nodes_from([1, 2, 3, 4], bipartite=0)
>>> B.add_nodes_from(['a', 'b', 'c'], bipartite=1)
>>> B.add_edges_from([(1, 'a'), (1, 'b'), (2, 'b'), (2, 'c'), (3, 'c'), (4, 'a')])
>>> H = Hypergraph.from_bipartite(B, nodes=1)
>>> list(H.nodes), list(H.edges)
(['a', 'b', 'c'], [1, 2, 3, 4])
```

```python
classmethod from_incidence_dataframe(df, name=None, fillna=0, key=None, return_only_dataframe=False, **kwargs)
```

Create a hypergraph from a Pandas Dataframe object, which has values equal to the incidence matrix of a hypergraph. Its index will identify the nodes and its columns will identify its edges.

**Parameters**

- `df` *(Pandas.DataFrame)* – a real valued dataframe with a single index
- `name` *(str, optional, default=None)*
- `fillna` *(float, default=0)* – a real value to place in empty cell, all-zero columns will not generate an edge.
- `key` *(function, optional, default=None)* – boolean function to be applied to dataframe. will be applied to entire dataframe.
- `return_only_dataframe` *(bool, default=False)* – to use the incidence_dataframe with cell_properties or properties, set this to true and use it as the setsystem in the Hypergraph constructor.

See also:

`from_numpy_array`

**Return type**

`Hypergraph` | `pd.DataFrame`

```python
classmethod from_incidence_matrix(M, name=None, **kwargs)
```

Accepts numpy.matrix or scipy.sparse matrix

```python
classmethod from_numpy_array(M, node_names=None, edge_names=None, name=None, key=None, **kwargs)
```

Create a hypergraph from a real valued matrix represented as a 2 dimensional numpy array. The matrix is converted to a matrix of 0’s and 1’s so that any truthy cells are converted to 1’s and all others to 0’s.

**Parameters**

- `M` *(real valued array-like object, 2 dimensions)* – representing a real valued matrix with rows corresponding to nodes and columns to edges
- `node_names` *(object, array-like, default=None)* – List of node names must be the same length as M.shape[0]. If None then the node names correspond to row indices with ‘v’ prepended.
HyperNetX Documentation, Release 2.3.5

- **edge_names**(object, array-like, default=None) – List of edge names must have the same length as M.shape[1]. If None then the edge names correspond to column indices with ‘e’ prepended.

- **name**(hashable)

- **key**(optional function) – boolean function to be evaluated on each cell of the array, must be applicable to numpy.array

**Return type**

*Hypergraph*

**Note:** The constructor does not generate empty edges. All zero columns in M are removed and the names corresponding to these edges are discarded.

**get_cell_properties**(edge_uid, node_uid, prop_name=None)

Get cell properties on a specified edge and node

**Parameters**

- **edge_uid**(str | int) – edge_uid
- **node_uid**(str | int) – node_uid
- **prop_name**(str, optional, default=None) – name of a cell property; if None, all cell properties will be returned

**Returns**

cell property value if prop_name is provided, otherwise dict of all cell properties and values

**Return type**

Any

**get_linegraph**(s=1, edges=True)

Creates an s-linegraph for the Hypergraph. If edges=True, then the edges will be the vertices of the line graph. Two vertices are connected by an s-line-graph edge if the corresponding hypergraph edges intersect in at least s hypergraph nodes. If edges=False, the hypergraph nodes will be the vertices of the line graph. Two vertices are connected if the nodes they correspond to share at least s incident (hyper)edges.

**Parameters**

- **s**(int) – The width of the connections.
- **edges**(bool, optional, default = True) – Determine if edges or nodes will be the vertices in the linegraph.

**Returns**

A NetworkX graph.

**Return type**

*nx.Graph*

**get_properties**(uid, level=0, prop_name=None)

Returns an object’s specific property or all properties

**Parameters**

- **uid**(hashable) – edge or node id
- **level**(int | None , optional, default=0) – Enter 0 for edges and 1 for nodes.
• **prop_name** *(str | None, optional, default=None)* – if None then all properties associated with the object will be returned.

**Returns**

single property or dictionary of properties

**Return type**

Any

**incidence_dataframe** *(weights=False)*

**property incidence_dict**

Dictionary keyed by edge uids with values as the uids of nodes of each edge

**Return type**

dict

**incidence_matrix** *(index=False, weights=False)*

A sparse matrix indicating the existence of an incidence pair in the hypergraph. Each row corresponds to a node v and each column corresponds to an edge e. The entry corresponding to (row v, col e) is nonzero if v is an element of e. If weights = True then the value equals the weight given in the hypergraph incidence properties for the incidence pair (e,v). Otherwise, the value is 1.

**Parameters**

• **index** *(bool, optional, default = False)* – If index=True, returns a tuple containing the incidence matrix, an np.ndarray containing the row and column index of node_uids, and an np.ndarray containing the row and column index of edge_uids. Otherwise, returns the incidence matrix.

• **weights** *(bool, optional, default = False)* – If True, use the incidence weights corresponding to the row and column of the entry.

**Returns**

• **incidence matrix** *(scipy.sparse.csr_matrix)*

• **node indexes** *(np.ndarray)* – an np.ndarray containing the row and column index of node_uids

• **edge indexes** *(np.ndarray)* – an np.ndarray containing the row and column index of edge_uids

**property incidences**

Object associated with incidence pairs

**Return type**

HypergraphView

**intersection** *(other, name=None)*

Returns a hypergraph created by restricting to incidence pairs contained in both self and other. Properties inherited from self.

**Parameters**

• **other** *(Hypergraph)*

• **name** *(str, optional, default=None)*

**Return type**

Hypergraph
is_connected\(s=1, edges=False\)
Determines if hypergraph is \(s\)-connected.

Parameters
- \(s\) \((\text{int}, \text{optional}, \text{default}=1)\)
- \(edges\) \((\text{boolean}, \text{optional}, \text{default}=False)\) – If True, will determine if \(s\)-edge-connected. For \(s=1\) \(s\)-edge-connected is the same as \(s\)-connected.

Returns
- is_connected

Return type
- boolean

Notes
A hypergraph is \(s\) node connected if for any two nodes \(v_0,v_n\) there exists a sequence of nodes \(v_0,v_1,v_2,...,v_{(n-1)},v_n\) such that every consecutive pair of nodes \(v(i),v(i+1)\) share at least \(s\) edges.

A hypergraph is \(s\) edge connected if for any two edges \(e_0,e_n\) there exists a sequence of edges \(e_0,e_1,e_2,...,e_{(n-1)},e_n\) such that every consecutive pair of edges \(e(i),e(i+1)\) share at least \(s\) nodes.

neighbors\((node, s=1)\)
The nodes in hypergraph which share \(s\) edge(s) with node.

Parameters
- \(node\) \((\text{hashable})\) – uid for a node in hypergraph
- \(s\) \((\text{int}, \text{optional}, \text{default}=1)\) – Minimum number of edges shared by neighbors with node.

Returns
- neighbors – \(s\)-neighbors share at least \(s\) edges in the hypergraph

Return type
- list

node_diameters\((s=1)\)
Returns the node diameters of the connected components in the hypergraph.

Parameters
- \(s\) \((\text{int}, \text{optional}, \text{default}=1)\)

Returns
- maximum diameter, list of diameters, list of component – maximum diameter, list of diameters (List of node_diameters for \(s\)-node component subgraphs in hypergraph), list of component (List of the node uids in the \(s\)-node component subgraphs)

Return type
- tuple[int, list, list]

property nodes
Object associated with nodes.

Return type
- HypergraphView
order()

The number of nodes in hypergraph.

- **Returns**
  - order

- **Return type**
  - int

property properties

Returns incidence properties

- **Return type**
  - pandas.DataFrame

remove_edges(edge_uids, name=None, inplace=True)

Removes the edges from the Hypergraph. If inplace=True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.

- **Parameters**
  - edge_uids (str | int | list[str | int]) – edge_uids
  - name (str, optional, default=None) – The name of the new Hypergraph. Used only when inplace=False; ignored if inplace=True.
  - inplace (bool, optional, default=True) – Whether to replace the current hypergraph with a new one.

- **Return type**
  - Hypergraph

remove_incidences(incidence_uids, name=None, inplace=True)

Removes the incidences from the Hypergraph. If inplace=True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.

- **Parameters**
  - incidence_uids (tuple[str | int] | list[tuple[str | int]]) – incidence_uids
  - name (str, optional, default=None) – The name of the new Hypergraph. Used only when inplace=False; ignored if inplace=True.
  - inplace (bool, optional, default=True) – Whether to replace the current hypergraph with a new one.

- **Return type**
  - Hypergraph

remove_nodes(node_uids, name=None, inplace=True)

Removes the nodes from the Hypergraph. If inplace=True, changes the existing Hypergraph. Otherwise, creates a new Hypergraph with the requested changes.

- **Parameters**
  - node_uids (str | int | list[str | int]) – node_uids
  - name (str, optional, default=None) – The name of the new Hypergraph. Used only when inplace=False; ignored if inplace=True.
  - inplace (bool, optional, default=True) – Whether to replace the current hypergraph with a new one.
remove_singletons(name=None)

Constructs clone of hypergraph with singleton edges removed.

Parameters

- name (str, optional, default=None)

Return type

Hypergraph

rename(edges=None, nodes=None, name=None, inplace=True)

Rename the edges and/or nodes of the hypergraph.

Parameters

- edges (dict, optional, default=None) – dictionary of replacement edge_uids
- nodes (dict, optional, default=None) – dictionary of replacement node_uids
- name (str, optional, default=None)
- inplace (bool, optional, default=True)

Return type

Hypergraph

restrict_to_edges(edges, name=None)

New hypergraph gotten by restricting to edges

Parameters

- edges (Iterable) – edge identifiers to restrict to
- name (str | int, optional, default=None) – edge identifier

Return type

Hypergraph

restrict_to_nodes(nodes, name=None)

New hypergraph gotten by restricting to nodes

Parameters

- nodes (Iterable) – node identifiers to restrict to
- name (str | int, optional, default=None) – node identifier

Return type

Hypergraph

s_component_subgraphs(s=1, edges=True, return_singletons=False, name=None)

Returns a generator for the induced subgraphs of s_connected components. Removes singletons unless return_singletons is set to True. Computed using s-linegraph generated either by the hypergraph (edges=True) or its dual (edges = False)

Parameters

- s (int, optional, default=1)
- edges (boolean, optional, default=False) – Determines if edge or node components are desired. Returns subgraphs equal to the hypergraph restricted to each set of nodes(edges) in the s-connected components or s-edge-connected components
- **return_singletons** (bool, optional, default=False) – If True, keep singletons in subgraph. Otherwise, remove singletons.

- **name** (str, optional, default=None)

Yields

- **s_component_subgraphs** (iterator) – Iterator returns subgraphs generated by the edges (or nodes) in the s-edge(node) components of hypergraph.

**s_components** (s=1, edges=True, return_singletons=True)

Same as **s_connected_components()**

See also: **s_connected_components**

**s_connected_components** (s=1, edges=True, return_singletons=False)

Returns a generator for the **s-edge-connected component** or the **s-node-connected component** of the hypergraph.

Parameters

- **s** (int, optional, default=1)
- **edges** (boolean, optional, default=True) – If True, return edge components; otherwise, return node components
- **return_singletons** (bool, optional, default=False) – If True, keep singletons. Otherwise, remove singletons

Notes

If edges=True, this method returns the s-edge-connected components as lists of lists of edge uids. An s-edge-component has the property that for any two edges e1 and e2 there is a sequence of edges starting with e1 and ending with e2 such that pairwise adjacent edges in the sequence intersect in at least s nodes. If s=1 these are the path components of the hypergraph.

If edges=False this method returns s-node-connected components. A list of sets of uids of the nodes which are s-walk connected. Two nodes v1 and v2 are s-walk-connected if there is a sequence of nodes starting with v1 and ending with v2 such that pairwise adjacent nodes in the sequence share s edges. If s=1 these are the path components of the hypergraph.

Example

```python
>>> S = {'A':{1,2,3}, 'B':{2,3,4}, 'C':{5,6}, 'D':{}}
>>> H = Hypergraph(S)

>>> list(H.s_connected_components(edges=True))
[['C', 'D'], {'A', 'B'}]

>>> list(H.s_connected_components(edges=False))
[[1, 2, 3, 4], {5, 6}]
```

Yields

- **s_connected_components** (iterator) – Iterator returns sets of uids of the edges (or nodes) in the s-edge(node) components of hypergraph.
**set_state(**kwargs)**
Allow state_dict updates from outside of class. Use with caution.

Parameters
**kwargs (dict, optional) – key-value pairs to save in state dictionary

**property shape**
The number of nodes, number of edges

Returns
number of nodes, number of edges

Return type
tuple

**singleton**s()  
Returns a list of singleton edges. A singleton edge is an edge of size 1 with a node of degree 1.

Returns
singles – A list of edge uids.

Return type
list

**size**(edge, nodeset=None)
The number of nodes in nodeset that belong to edge. If nodeset is None then returns the size of edge

Parameters
edge (hashable) – The uid of an edge in the hypergraph

Returns
size

Return type
int

**sum**(other, name=None)
Hypergraph obtained by joining incidences from self and other. Removes duplicates and uses properties of self.

Parameters
other (Hypergraph)

Return type
Hypergraph

**toplexes**(return_hyp=False)
Computes a maximal collection of toplexes for the hypergraph. A toplex is a hyperedge, which is not contained in any other hyperedge. If return_hyp=True, then returns the simple hypergraph created by restricting to the toplexes.

Parameters
return_hyp (bool, optional, default=False)

Return type
Hypergraph | list

**union**(other, name=None)
The hypergraph gotten by joining incidence pairs contained in self and other. Duplicates removed. Properties inherited from self. Same as **sum**()
• other (Hypergraph)
• name (str, optional, default=None)

Return type
Hypergraph

classes.incidence_store module

class classes.incidence_store.IncidenceStore(data)
    Bases: object
    Incidence store object that stores and accesses (multi) incidences with standard methods.

    Parameters
    data (Two column pandas dataframe of edges and nodes, respectively.)

    collapse_identical_elements (level, use_keys=None)

    property data

    property dimensions
    The number of distinct edges and nodes in that order

    Returns
    Tuple of size two of (number of unique edges, number of unique nodes).

    Return type
tuple of ints

    property edges
    Returns an array of edge names from the incidence pairs

    Returns
    Returns an array of edge names

    Return type
array

    property elements

    equivalence_classes (level=0)

    property memberships

    neighbors (level, key)
    Returns elements or memberships depending on level.

    Parameters
    • level (int) – Level indicator for finding either elements or memberships. For level 0 (elements), returns nodes in the edge. For level 1 (memberships), returns edges containing the node.

    • key (int or str) – Name of node or edge depending on level.

    Returns
    Elements or memberships (depending on level) of a given edge or node, respectively.

    Return type
list
**property nodes**

Returns an array of node names from the incidence pairs

- **Returns**
  - Returns an array of node names

- **Return type**
  - array

**restrict_to**(level, items, inplace=False)

returns IncidenceStore of subset of incidence store restricted to pairs with items in the given level Will return with same data or deepcopy depending on inplace

- **Parameters**
  - level (int) – Level indicator for finding either elements or memberships. For level 0 (elements), returns nodes in the edge. For level 1 (memberships), returns edges containing the node.
  - items (list) – List of uids to be removed from level
  - inplace (bool, optional) – whether to replace self, by default False

- **Returns**
  - subset of incidence store given a restriction.

- **Return type**
  - list

---

**classes.property_store module**

**class** classes.property_store.PropertyStore(data=None, default_weight=1)

- **Bases:** object

  Class for storing properties of a collection of edges, nodes, or incidences.

  Properties will be stored in a pandas dataframe.

- **copy**(deep=False)

  Create a copy of the PropertyStore. If deep=True, create a copy of the underlying data table. Otherwise, use the same underlying data table from the original PropertyStore

  - **Parameters**
    - deep (bool, optional, default=False)

  - **Return type**
    - PropertyStore

- **property default_properties**: dict

  Returns copy of default dictionary of properties

  - **Returns**
    - Dictionary of properties automatically given to objects either in the property store if no user defined values have been assigned to them or objects that have not yet been added to the Property Store.

  - **Return type**
    - dict
get_properties\( (uid) \rightarrow \text{dict} \)
Get all properties of an item

Parameters
uid (Hashable) – uid is the index used to fetch all its properties

Returns
Output dictionary containing all properties of the uid. \{\text{name}: \text{property value}, \ldots, \text{properties}: \{\text{property name}: \text{property value}\}\}

Return type
dict

See also:
get_property, set_property

get_property\( (uid, prop\_name) \rightarrow \text{Any} \)
Get a property of an item

Parameters
• uid (Hashable) – uid is the index used to fetch its property
  • prop\_name (str | int) – name of the property to get

Returns
• out (Any) – value of the property
  • None – if property not found

See also:
get_properties, set_property

property properties: DataFrame
Properties assigned to all items in the underlying data table

Returns
a dataframe with the following columns:
uid, weight, properties, \langle\text{optional props}\rangle or level, id, weight, properties, \langle\text{optional props}\rangle

Return type
pandas.DataFrame

set_defaults\( (defaults) \rightarrow \text{None} \)
Set default values for properties

Parameters
defaults (dict)

Return type
None

set_properties\( (uid, props) \rightarrow \text{None} \)

Parameters
• uid (Hashable) – uid is the index used to set its property
  • props (a dictionary containing user-defined properties)

Return type
None
See also:

get_property, get_properties, set_property

set_property(uid, prop_name, prop_val) → None
Set a property of an item in the ‘properties’ collection

Parameters

• uid (Hashable) – uid is the index used to set its property
• prop_name (str / int) – name of the property to set
• prop_val (any) – value of the property to set

Return type

None

See also:

get_property, get_properties, set_properties

classes.property_store.flatten(my_dict)
Recursive method to flatten dictionary for returning properties as a dictionary instead of a Series, from https://stackoverflow.com/a/71952620

5.5.2 algorithms

algorithms package

Submodules

algorithms.contagion module

algorithms.contagion.Gillespie_SIR(H, tau, gamma, transmission_function=<function threshold>,
  initial_infecteds=None, initial_recovereds=None, rho=None, tmin=0, tmax=inf, **args)


Parameters

• H (HyperNetX Hypergraph object)
• tau (dictionary) – Edge sizes as keys (must account for all edge sizes present) and rates of infection for each size (float)
• gamma (float) – The healing rate
• transmission_function (lambda function, default: threshold) – A lambda function that has required arguments (node, status, edge) and optional arguments
• initial_infecteds(list or numpy array, default: None) – Iterable of initially infected node uids
• initial_recovereds (list or numpy array, default: None) – An iterable of initially recovered node uids
• **rho** *(float from 0 to 1, default: None)* – The fraction of initially infected individuals. Both rho and initially infected cannot be specified.

• **tmin** *(float, default: 0)* – Time at the start of the simulation

• **tmax** *(float, default: float('Inf'))* – Time at which the simulation should be terminated if it hasn’t already.

• **return_full_data** *(bool, default: False)* – This returns all the infection and recovery events at each time if True.

• ****args** *(Optional arguments to transmission function)* – This allows user-defined transmission functions with extra parameters.

**Returns**

- *t, S, I, R* – time (t), number of susceptible (S), infected (I), and recovered (R) at each time.

**Return type**

- numpy arrays

**Notes**

Example:

```python
>>> import hypernetx.algorithms.contagion as contagion
>>> import random
>>> import hypernetx as hnx

>>> n = 1000
>>> m = 10000
>>> hyperedgeList = [random.sample(range(n), k=random.choice([2,3])) for i in range(m)]
>>> H = hnx.Hypergraph(hyperedgeList)
>>> tau = {2:0.1, 3:0.1}
>>> gamma = 0.1
>>> tmax = 100
>>> t, S, I, R = contagion.Gillespie_SIR(H, tau, gamma, rho=0.1, tmin=0, tmax=tmax)
```


**Parameters**

- **H** *(HyperNetX Hypergraph object)*

- **tau** *(dictionary)* – Edge sizes as keys (must account for all edge sizes present) and rates of infection for each size (float)

- **gamma** *(float)* – The healing rate

- **transmission_function** *(lambda function, default: threshold)* – A lambda function that has required arguments (node, status, edge) and optional arguments

- **initial_infecteds** *(list or numpy array, default: None)* – Iterable of initially infected node uids
- **rho** *(float from 0 to 1, default: None)* – The fraction of initially infected individuals. Both rho and initially infected cannot be specified.
- **tmin** *(float, default: 0)* – Time at the start of the simulation
- **tmax** *(float, default: 100)* – Time at which the simulation should be terminated if it hasn’t already.
- **return_full_data** *(bool, default: False)* – This returns all the infection and recovery events at each time if True.
- ****args** *(Optional arguments to transmission function)* – This allows user-defined transmission functions with extra parameters.

Returns
- `t, S, I` – time (t), number of susceptible (S), and infected (I) at each time.

Return type
- `numpy arrays`

Notes

Example:

```python
>>> import hypernetx.algorithms.contagion as contagion
>>> import random
>>> import hypernetx as hnx

>>> n = 1000
>>> m = 10000
>>> hyperedgeList = [random.sample(range(n), k=random.choice([2,3])) for i in range(m)]
>>> H = hnx.Hypergraph(hyperedgeList)
>>> tau = {2:0.1, 3:0.1}
>>> gamma = 0.1
>>> tmax = 100
>>> t, S, I = contagion.Gillespie_SIS(H, tau, gamma, rho=0.1, tmin=0, tmax=tmax)
```

`algorithms.contagion.collective_contagion(node, status, edge)`

The collective contagion mechanism described in “The effect of heterogeneity on hypergraph contagion models” by Landry and Restrepo https://doi.org/10.1063/5.0020034

Parameters
- **node** *(hashable)* – the node uid to infect (If it doesn’t have status “S”, it will automatically return False)
- **status** *(dictionary)* – The nodes are keys and the values are statuses (The infected state denoted with “I”)
- **edge** *(iterable)* – Iterable of node ids (node must be in the edge or it will automatically return False)

Returns
- False if there is no potential to infect and True if there is.

Return type
- `bool`
Notes

Example:

```python
>>> status = {0:"S", 1:"I", 2:"I", 3:"S", 4:"R"}
>>> collective_contagion(0, status, (0, 1, 2))
True
>>> collective_contagion(1, status, (0, 1, 2))
False
>>> collective_contagion(3, status, (0, 1, 2))
False
```

```python
algorithms.contagion.contagion_animation(fig, H, transition_events, node_state_color_dict, edge_state_color_dict, node_radius=1, fps=1)
```

A function to animate discrete-time contagion models for hypergraphs. Currently only supports a circular layout.

Parameters

- `fig` (*matplotlib Figure object*)
- `H` (*HyperNetX Hypergraph object*)
- `transition_events` (*dictionary*) – The dictionary that is output from the discrete_SIS and discrete_SIR functions with return_full_data=True
- `node_state_color_dict` (*dictionary*) – Dictionary which specifies the colors of each node state. All node states must be specified.
- `edge_state_color_dict` (*dictionary*) – Dictionary with keys that are edge states and values which specify the colors of each edge state (can specify an alpha parameter). All edge-dependent transition states must be specified (most common is “I”) and there must be a default “OFF” setting.
- `node_radius` (*float, default: 1*) – The radius of the nodes to draw
- `fps` (*int > 0, default: 1*) – Frames per second of the animation

Return type

*matplotlib Animation object*

Notes

Example:

```python
>>> import hypernetx.algorithms.contagion as contagion
>>> import random
>>> import hypernetx as hnx
>>> import matplotlib.pyplot as plt
>>> from IPython.display import HTML

>>> n = 1000
>>> m = 10000
>>> hyperedgeList = [random.sample(range(n), k=random.choice([2,3])) for i in range(m)]
>>> H = hnx.Hypergraph(hyperedgeList)
>>> tau = {2:0.1, 3:0.1}
>>> gamma = 0.1
>>> tmax = 100
```
dt = 0.1

transition_events = contagion.discrete_SIS(H, tau, gamma, rho=0.1, tmin=0, tmax=tmax, dt=dt, return_full_data=True)

node_state_color_dict = {"S":"green", "I":"red", "R":"blue"}

edge_state_color_dict = {"S":(0, 1, 0, 0.3), "I":(1, 0, 0, 0.3), "R":(0, 0, 1, 0.3), "OFF": (1, 1, 1, 0)}

d = 1

animation = contagion.contagion_animation(fig, H, transition_events, node_state_color_dict, edge_state_color_dict, node_radius=1, fps=fps)

HTML(animation.to_jshtml())

algorithms.contagion.discrete_SIR(H, tau, gamma, transmission_function=<function threshold>, initial_infecteds=None, initial_recovereds=None, rho=None, tmin=0, tmax=inf, dt=1.0, return_full_data=False, **args)

A discrete-time SIR model for hypergraphs similar to the construction described in “The effect of heterogeneity on hypergraph contagion models” by Landry and Restrepo https://doi.org/10.1063/5.0020034 and “Simplicial models of social contagion” by Iacopini et al. https://doi.org/10.1038/s41467-019-10431-6

Parameters

- **H** (*HyperNetX Hypergraph object*)
- **tau** (*dictionary*) – Edge sizes as keys (must account for all edge sizes present) and rates of infection for each size (float)
- **gamma** (*float*) – The healing rate
- **transmission_function** (*lambda function, default: threshold*) – A lambda function that has required arguments (node, status, edge) and optional arguments
- **initial_infecteds** (*list or numpy array, default: None*) – Iterable of initially infected node uids
- **initial_recovereds** (*list or numpy array, default: None*) – An iterable of initially recovered node uids
- **rho** (*float from 0 to 1, default: None*) – The fraction of initially infected individuals. Both rho and initially infected cannot be specified.
- **tmin** (*float, default: 0*) – Time at the start of the simulation
- **tmax** (*float, default: float('Inf')*) – Time at which the simulation should be terminated if it hasn’t already.
- **dt** (*float > 0, default: 1.0*) – Step forward in time that the simulation takes at each step.
- **return_full_data** (*bool, default: False*) – This returns all the infection and recovery events at each time if True.
- ****args** (*Optional arguments to transmission function*) – This allows user-defined transmission functions with extra parameters.

Returns

- if **return_full_data** –
  - **dictionary**
    - Time as the keys and events that happen as the values.
• else –

\[ t, S, I, R \]

[numpy arrays] time (t), number of susceptible (S), infected (I), and recovered (R) at each time.

**Notes**

Example:

```python
>>> import hypernetx.algorithms.contagion as contagion
>>> import random
>>> import hypernetx as hnx

>>> n = 1000
>>> m = 10000
>>> hyperedgeList = [random.sample(range(n), k=random.choice([2,3])) for i in range(m)]
>>> H = hnx.Hypergraph(hyperedgeList)
>>> tau = {2:0.1, 3:0.1}
>>> gamma = 0.1
>>> tmax = 100
>>> dt = 0.1
>>> t, S, I, R = contagion.discrete_SIR(H, tau, gamma, rho=0.1, tmin=0, tmax=tmax, dt=dt)
```

algorithms.contagion.discrete_SIS(H, tau, gamma, transmission_function=<function threshold>, initial_infecteds=None, rho=None, tmin=0, tmax=100, dt=1.0, return_full_data=False, **args)

A discrete-time SIS model for hypergraphs as implemented in “The effect of heterogeneity on hypergraph contagion models” by Landry and Restrepo https://doi.org/10.1063/5.0020034 and “Simplicial models of social contagion” by Iacopini et al. https://doi.org/10.1038/s41467-019-10431-6

**Parameters**

- \( H \) *(HyperNetX Hypergraph object)*
- \( \text{tau} \) *(dictionary)* – Edge sizes as keys (must account for all edge sizes present) and rates of infection for each size (float)
- \( \text{gamma} \) *(float)* – The healing rate
- \( \text{transmission_function} \) *(lambda function, default: threshold)* – A lambda function that has required arguments (node, status, edge) and optional arguments
- \( \text{initial_infecteds} \) *(list or numpy array, default: None)* – Iterable of initially infected node uids
- \( \text{rho} \) *(float from 0 to 1, default: None)* – The fraction of initially infected individuals. Both rho and initially infected cannot be specified.
- \( \text{tmin} \) *(float, default: 0)* – Time at the start of the simulation
- \( \text{tmax} \) *(float, default: 100)* – Time at which the simulation should be terminated if it hasn’t already.
- \( \text{dt} \) *(float > 0, default: 1.0)* – Step forward in time that the simulation takes at each step.
• **return_full_data** (bool, **default**: False) – This returns all the infection and recovery events at each time if True.

• **args** (Optional arguments to transmission function) – This allows user-defined transmission functions with extra parameters.

Returns

• if return_full_data –
  
  dictionary
  Time as the keys and events that happen as the values.

• else –

  t, S, I
  [numpy arrays] time (t), number of susceptible (S), and infected (I) at each time.

Notes

Example:

```python
>>> import hypernetx.algorithms.contagion as contagion
>>> import random
>>> import hypernetx as hnx

>>> n = 1000
>>> m = 10000
>>> hyperedgeList = [random.sample(range(n), k=random.choice([2,3])) for i in range(m)]
>>> H = hnx.Hypergraph(hyperedgeList)
>>> tau = {2:0.1, 3:0.1}
>>> gamma = 0.1
>>> tmax = 100
>>> dt = 0.1
>>> t, S, I = contagion.discrete_SIS(H, tau, gamma, rho=0.1, tmin=0, tmax=tmax, dt=dt)
```

algorithms.contagion.individual_contagion(node, status, edge)

The individual contagion mechanism described in “The effect of heterogeneity on hypergraph contagion models” by Landry and Restrepo https://doi.org/10.1063/5.0020034

Parameters

• **node** (hashable) – The node uid to infect (If it doesn’t have status “S”, it will automatically return False)

• **status** (dictionary) – The nodes are keys and the values are statuses (The infected state denoted with “I”)

• **edge** (iterable) – Iterable of node ids (node must be in the edge or it will automatically return False)

Returns

False if there is no potential to infect and True if there is.

Return type

bool
**Notes**

Example:

```python
>>> status = {0:"S", 1:"I", 2:"I", 3:"S", 4:"R"}
>>> individual_contagion(0, status, (0, 1, 3))
True
>>> individual_contagion(1, status, (0, 1, 2))
False
>>> collective_contagion(3, status, (0, 3, 4))
False
```

`algorithms.contagion.majority_vote(node, status, edge)`

The majority vote contagion mechanism. If a majority of neighbors are contagious, it is possible for an individual to change their opinion. If opinions are divided equally, choose randomly.

**Parameters**

- `node` *(hashable)* – The node uid to infect (If it doesn’t have status “S”, it will automatically return False)
- `status` *(dictionary)* – The nodes are keys and the values are statuses (The infected state denoted with “I”)
- `edge` *(iterable)* – Iterable of node ids (node must be in the edge or it will automatically return False

**Returns**

False if there is no potential to infect and True if there is.

**Return type**

`bool`

**Notes**

Example:

```python
>>> status = {0:"S", 1:"I", 2:"I", 3:"S", 4:"R"}
>>> majority_vote(0, status, (0, 1, 2))
True
>>> majority_vote(0, status, (0, 1, 2, 3))
True
>>> majority_vote(1, status, (0, 1, 2))
False
>>> majority_vote(3, status, (0, 1, 2))
False
```

`algorithms.contagion.threshold(node, status, edge, tau=0.1)`

The threshold contagion mechanism

**Parameters**

- `node` *(hashable)* – The node uid to infect (If it doesn’t have status “S”, it will automatically return False)
- `status` *(dictionary)* – The nodes are keys and the values are statuses (The infected state denoted with “I”)

5.5. HyperNetX Packages
**edge** (iterable) – Iterable of node ids (node must be in the edge or it will automatically return False)

**tau** (float between 0 and 1, default: 0.1) – The fraction of nodes in an edge that must be infected for the edge to be able to transmit to the node

Returns
False if there is no potential to infect and True if there is.

Return type
bool

Notes

Example:
```python
>>> status = {0:"S", 1:"I", 2:"I", 3:"S", 4:"R"}
>>> threshold(0, status, (0, 2, 3, 4), tau=0.2)
True
>>> threshold(0, status, (0, 2, 3, 4), tau=0.5)
False
>>> threshold(3, status, (1, 2, 3), tau=1)
False
```

---

**algorithms.generative_models module**

*algorithms.generative_models.chung_lu_hypergraph*(k1, k2)

A function to generate an extension of Chung-Lu hypergraph as implemented by Mirah Shi and described for bipartite networks by Aksoy et al. in [https://doi.org/10.1093/comnet/cnx001](https://doi.org/10.1093/comnet/cnx001)

Parameters

- **k1** (dictionary) – This a dictionary where the keys are node ids and the values are node degrees.
- **k2** (dictionary) – This a dictionary where the keys are edge ids and the values are edge degrees also known as edge sizes.

Return type
HyperNetX Hypergraph object

Notes

The sums of k1 and k2 should be roughly the same. If they are not the same, this function returns a warning but still runs. The output currently is a static Hypergraph object. Dynamic hypergraphs are not currently supported.

Example:
```python
>>> import hypernetx.algorithms.generative_models as gm
>>> 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 = gm.chung_lu_hypergraph(k1, k2)
```
algorithms.generative_models.dcsbm_hypergraph(k1, k2, g1, g2, omega)

A function to generate an extension of DCSBM hypergraph as implemented by Mirah Shi and described for bipartite networks by Larremore et al. in https://doi.org/10.1103/PhysRevE.90.012805

Parameters

- **k1** (dictionary) – This a dictionary where the keys are node ids and the values are node degrees.
- **k2** (dictionary) – This a dictionary where the keys are edge ids and the values are edge degrees also known as edge sizes.
- **g1** (dictionary) – 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** (dictionary) – 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.

Return type

HyperNetX Hypergraph object

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.

The output currently is a static Hypergraph object. Dynamic hypergraphs are not currently supported.

Example:

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

algorithms.generative_models.erdos_renyi_hypergraph(n, m, p, node_labels=None, edge_labels=None)

A function to generate an Erdos-Renyi hypergraph as implemented by Mirah Shi and described for bipartite networks by Aksoy et al. in https://doi.org/10.1093/comnet/cnx001

Parameters

- **n** (int) – Number of nodes
- **m** (int) – Number of edges
- **p** (float) – The probability that a bipartite edge is created
- **node_labels** (list, default=None) – Vertex labels
- **edge_labels** (list, default=None) – Hyperedge labels
Return type
HyperNetX Hypergraph object

Example:

```python
>>> import hypernetx.algorithms.generative_models as gm
>>> n = 1000
>>> m = n
>>> p = 0.01
>>> H = gm.erdos_renyi_hypergraph(n, m, p)
```

**algorithms.homology_mod2** module

**Homology and Smith Normal Form**

The purpose of computing the Homology groups for data generated hypergraphs is to identify data sources that correspond to interesting features in the topology of the hypergraph.

The elements of one of these Homology groups are generated by $k$-dimensional cycles of relationships in the original data that are not bound together by higher order relationships. Ideally, we want the briefest description of these cycles; we want a minimal set of relationships exhibiting interesting cyclic behavior. This minimal set will be a bases for the Homology group.

The cyclic relationships in the data are discovered using a **boundary map** represented as a matrix. To discover the bases we compute the **Smith Normal Form** of the boundary map.

**Homology Mod2**

This module computes the homology groups for data represented as an abstract simplicial complex with chain groups $\{C_k\}$ and $\mathbb{Z}_2$ additions. The boundary matrices are represented as rectangular matrices over $\mathbb{Z}_2$. These matrices are diagonalized and represented in Smith Normal Form. The kernel and image bases are computed and the Betti numbers and homology bases are returned.


**algorithms.homology_mod2.add_to_column(M, i, j)**

Replaces column $i$ (of $M$) with logical xor between column $i$ and $j$

**Parameters**

- $M$ (*np.array*) – matrix
- $i$ (*int*) – index of column being altered
- $j$ (*int*) – index of column being added to altered

**Returns**

$N$

**Return type**

*np.array*

**algorithms.homology_mod2.add_to_row(M, i, j)**

Replaces row $i$ with logical xor between row $i$ and $j$

**Parameters**

- $M$ (*np.array*) – matrix
- $i$ (*int*) – index of row being altered
- $j$ (*int*) – index of row being added to altered

**Returns**

$N$

**Return type**

*np.array*
• \( M (\text{np.array}) \)
• \( i (\text{int}) \) – index of row being altered
• \( j (\text{int}) \) – index of row being added to altered

Returns
\( N \)

Return type
\( \text{np.array} \)

\texttt{algorithms.homology_mod2.betti} (\( bd, k=\text{None} \))

Generate the \( k \)-th betti numbers for a chain complex with boundary matrices given by \( bd \)

Parameters
• \( bd (\text{dict of } k\text{-boundary matrices keyed on dimension of domain}) \)
• \( k (\text{int, list or tuple, optional, default=\text{None}}) \) – list must be min value and max value of \( k \) values inclusive if None, then all betti numbers for dimensions of existing cells will be computed.

Returns
\( \text{betti} \) – Description

Return type
\( \text{dict} \)

\texttt{algorithms.homology_mod2.betti_numbers} (\( h, k=\text{None} \))

Return the \( k \)-th betti numbers for the simplicial homology of the ASC associated to \( h \)

Parameters
• \( h (\text{hnx.Hypergraph}) \) – Hypergraph to compute the betti numbers from
• \( k (\text{int or list, optional, default=\text{None}}) \) – list must be min value and max value of \( k \) values inclusive if None, then all betti numbers for dimensions of existing cells will be computed.

Returns
\( \text{betti} \) – A dictionary of betti numbers keyed by dimension

Return type
\( \text{dict} \)

\texttt{algorithms.homology_mod2.bkMatrix} (\( km1basis, kbasis \))

Compute the boundary map from \( C_{k-1} \)-basis to \( C_k \) basis with respect to \( \mathbb{Z}_2 \)

Parameters
• \( km1basis (\text{indexable iterable}) \) – Ordered list of \( k-1 \) dimensional cell
• \( kbasis (\text{indexable iterable}) \) – Ordered list of \( k \) dimensional cells

Returns
\( \text{bk} \) – boundary matrix in \( \mathbb{Z}_2 \) stored as boolean

Return type
\( \text{np.array} \)

\texttt{algorithms.homology_mod2.boundary_group} (\( image\_basis \))

Returns a \texttt{csr_matrix} with rows corresponding to the elements of the group generated by image basis over \( \mathbb{Z}_2 \)
Parameters

- `image_basis` *(numpy.ndarray or scipy.sparse.csr_matrix)* – 2d-array of basis elements

Return type

scipy.sparse.csr_matrix

`algorithms.homology_mod2.chain_complex(h, k=None)`

Compute the k-chains and k-boundary maps required to compute homology for all values in k

Parameters

- `h` *(hnx.Hypergraph)*
- `k` *(int or list of length 2, optional, default=None)* – k must be an integer greater than 0 or a list of length 2 indicating min and max dimensions to be computed. eg. if k = [1,2] then 0,1,2,3-chains and boundary maps for k=1,2,3 will be returned, if None then k = [1,max dimension of edge in h]

Returns

- `C, bd` – C is a dictionary of lists bd is a dictionary of numpy arrays

Return type

dict

`algorithms.homology_mod2.homology_basis(bd, k=None, boundary=False, **kwargs)`

Compute a basis for the kth-simplicial homology group, $\mathbb{SH}_k$, defined by a chain complex $\mathbb{CS}$ with boundary maps given by bd $=$ \{k:partial_k \}$

Parameters

- `bd` *(dict)* – dict of boundary matrices on k-chains to k-1 chains keyed on k if krange is a tuple then all boundary matrices k in [krange[0],..,krange[1]] inclusive must be in the dictionary
- `k` *(int or list of ints, optional, default=None)* – k must be a positive integer or a list of 2 integers indicating min and max dimensions to be computed, if none given all homology groups will be computed from available boundary matrices in bd
- `boundary` *(bool)* – option to return a basis for the boundary group from each dimension. Needed to compute the shortest generators in the homology group.

Returns

- `basis` *(dict)* – dict of generators as 0-1 tuples keyed by dim basis for dimension k will be returned only if bd[k] and bd[k+1] have been provided.
- `im` *(dict)* – dict of boundary group generators keyed by dim

`algorithms.homology_mod2.hypergraph_homology_basis(h, k=None, shortest=False, interpreted=True)`

Computes the kth-homology groups mod 2 for the ASC associated with the hypergraph h for k in krange inclusive

Parameters

- `h` *(hnx.Hypergraph)*
- `k` *(int or list of length 2, optional, default=None)* – k must be an integer greater than 0 or a list of length 2 indicating min and max dimensions to be computed
- `shortest` *(bool, optional, default=False)* – option to look for shortest representative for each coset in the homology group, only good for relatively small examples
- `interpreted` *(bool, optional, default=True)* – if True will return an explicit basis in terms of the k-chains
Returns

- **basis (list)** – list of generators as k-chains as boolean vectors
- **interpreted_basis** – lists of k-chains in basis

`algorithms.homology_mod2.interpret(Ck, arr, labels=None)`

Returns the data as represented in Ck associated with the arr

Parameters

- **Ck (list)** – a list of k-cells being referenced by arr
- **arr (np.array)** – array of 0-1 vectors
- **labels (dict, optional)** – dictionary of labels to associate to the nodes in the cells

Returns

list of k-cells referenced by data in Ck

Return type

list

`algorithms.homology_mod2.kchainbasis(h, k)`

Compute the set of k dimensional cells in the abstract simplicial complex associated with the hypergraph.

Parameters

- **h** (*hnx.Hypergraph*)
- **k** (*int*) – dimension of cell

Returns

an ordered list of k-chains represented as tuples of length k+1

Return type

list

See also:

`hnx.hypergraph.toplexes`

Notes

- Method works best if h is simple [Berge], i.e. no edge contains another and there are no duplicate edges (toplexes).
- Hypergraph node uids must be sortable.

`algorithms.homology_mod2.logical_dot(ar1, ar2)`

Returns the boolean equivalent of the dot product mod 2 on two 1-d arrays of the same length.

Parameters

- **ar1 (numpy.ndarray)** – 1-d array
- **ar2 (numpy.ndarray)** – 1-d array

Returns

boolean value associated with dot product mod 2

Return type

bool
Raises

HyperNetXError – If arrays are not of the same length an error will be raised.

algorithms.homology_mod2.logical_matadd(mat1, mat2)

Returns the boolean equivalent of matrix addition mod 2 on two binary arrays stored as type boolean

Parameters

• mat1 (np.ndarray) – 2-d array of boolean values
• mat2 (np.ndarray) – 2-d array of boolean values

Returns

mat – boolean matrix equivalent to the mod 2 matrix addition of the matrices as matrices over \( \mathbb{Z}/2\mathbb{Z} \)

Return type

np.ndarray

Raises

HyperNetXError – If dimensions are not equal an error will be raised.

algorithms.homology_mod2.logical_matmul(mat1, mat2)

Returns the boolean equivalent of matrix multiplication mod 2 on two binary arrays stored as type boolean

Parameters

• mat1 (np.ndarray) – 2-d array of boolean values
• mat2 (np.ndarray) – 2-d array of boolean values

Returns

mat – boolean matrix equivalent to the mod 2 matrix multiplication of the matrices as matrices over \( \mathbb{Z}/2\mathbb{Z} \)

Return type

np.ndarray

Raises

HyperNetXError – If inner dimensions are not equal an error will be raised.

algorithms.homology_mod2.matmulreduce(arr, reverse=False)

Recursively applies a ‘logical multiplication’ to a list of boolean arrays.

For arr = [arr[0], arr[1], arr[2]...arr[n]] returns product arr[0]arr[1]...arr[n] If reverse = True, returns product arr[n]arr[n-1]...arr[0]

Parameters

• arr (list of np.array) – list of nxm matrices represented as np.array
• reverse (bool, optional) – order to multiply the matrices

Returns

P – Product of matrices in the list

Return type

np.array

algorithms.homology_mod2.reduced_row_echelon_form_mod2(M)

Computes the invertible transformation matrices needed to compute the reduced row echelon form of M modulo 2

Parameters

• M (np.array) – a rectangular matrix with elements in $\mathbb{Z}/2$
Returns
L, S, Linv – LM = S where S is the reduced echelon form of M and M = LinvS

Return type
np.arrays

algorithms.homology_mod2.smith_normal_form_mod2(M)
Computes the invertible transformation matrices needed to compute the Smith Normal Form of M modulo 2

Parameters
• M (np.array) – a rectangular matrix with data type bool
• track (bool) – if track=True will print out the transformation as \(Z/2Z\) matrix as it discovers L[i] and R[j]

Returns
L, R, S, Linv – LMR = S is the Smith Normal Form of the matrix M.

Return type
np.arrays

Note: Given a mxn matrix $M$ with entries in $Z_2$ we start with the equation: $L M R = S$, where $L = I_m$, and $R=I_n$ are identity matrices and $S = M$. We repeatedly apply actions to the left and right side of the equation to transform S into a diagonal matrix. For each action applied to the left side we apply its inverse action to the right side of $I_m$ to generate $L^{-1}$. Finally we verify: $SL R = S$ and $L^{-1} = I_m$.

algorithms.homology_mod2.swap_columns(i, j, *args)
Swaps ith and jth column of each matrix in args Returns a list of new matrices

Parameters
• i (int)
• j (int)
• args (np.arrays)

Returns
list of copies of args with ith and jth row swapped

Return type
list

algorithms.homology_mod2.swap_rows(i, j, *args)
Swaps ith and jth row of each matrix in args Returns a list of new matrices

Parameters
• i (int)
• j (int)
• args (np.arrays)

Returns
list of copies of args with ith and jth row swapped

Return type
list
algorithms.hypergraph_modularity module

Hypergraph_Modularity

Modularity and clustering for hypergraphs using HyperNetX. Adapted from F. Théberge’s GitHub repository: Hypergraph Clustering. See Tutorial 13 in the tutorials folder for library usage.

References

algorithms.hypergraph_modularity.conductance(H, A)

Computes conductance \([4]\) of hypergraph \(HG\) with respect to partition \(A\).

**Parameters**
- \(H\) (\texttt{hnx.Hypergraph}) – The hypergraph
- \(A\) (\texttt{set}) – Partition of the vertices in \(H\)

**Returns**
The conductance function for partition \(A\) on \(H\)

**Return type**
float

algorithms.hypergraph_modularity.dict2part(D)

Given a dictionary mapping the part for each vertex, return a partition as a list of sets; inverse function to part2dict

**Parameters**
- \(D\) (\texttt{dict}) – Dictionary keyed by vertices with values equal to integer index of the partition the vertex belongs to

**Returns**
List of sets; one set for each part in the partition

**Return type**
list

algorithms.hypergraph_modularity.kumar(HG, delta=0.01, verbose=False)

Compute a partition of the vertices in hypergraph \(HG\) as per Kumar’s algorithm\(^1\)

**Parameters**
- \(HG\) (\texttt{hnx.Hypergraph})
- \(\text{delta}\) (\texttt{float}, \textit{optional}) – convergence stopping criterion

**Returns**
A partition of the vertices in \(HG\)

**Return type**
list of sets

---

Given some initial partition $L$, compute a new partition of the vertices in $HG$ as per Last-Step algorithm.

Note: This is a very simple algorithm that tries moving nodes between communities to improve hypergraph modularity. It requires an initial non-trivial partition which can be obtained for example via graph clustering on the 2-section of $HG$, or via Kumar’s algorithm.

Parameters
- $HG$ ($hnx.Hypergraph$)
- $A$ (list of sets) – some initial partition of the vertices in $HG$
- $wdc$ (func, optional) – Hyperparameter for hypergraph modularity
- $delta$ (float, optional) – convergence stopping criterion
- $verbose$ (boolean, optional) – If set, also returns progress after each pass through the vertices

Returns
A new partition for the vertices in $HG$

Return type
list of sets

$\text{algorithms.hypergraph_modularity.last_step(HG, A, wdc=<function linear>, delta=0.01, verbose=False)}$

---

$\text{algorithms.hypergraph_modularity.linear(d, c)}$

Hyperparameter for hypergraph modularity for $d$-edge with $c$ vertices in the majority class. This is the default choice for \text{modularity()} and \text{last_step()} functions.

Parameters
- $d$ (int) – Number of vertices in an edge
- $c$ (int) – Number of vertices in the majority class

Returns
$c/d$ if $c>d/2$ else 0

Return type
float

$\text{algorithms.hypergraph_modularity.majority(d, c)}$

Hyperparameter for hypergraph modularity for $d$-edge with $c$ vertices in the majority class. This corresponds to the majority rule.

Parameters
- $d$ (int) – Number of vertices in an edge
- $c$ (int) – Number of vertices in the majority class

Returns
1 if $c>d/2$ else 0

---


Return type
  bool

algorithms.hypergraph_modularity.modularity(HG, A, wdc=<function linear>)
  Computes modularity of hypergraph HG with respect to partition A.

Parameters
  • HG (hnx.Hypergraph) – The hypergraph with some precomputed attributes via: precompute_attributes(HG)
  • A (list of sets) – Partition of the vertices in HG
  • wdc (func, optional) – Hyperparameter for hypergraph modularity

Note: For ‘wdc’, any function of the format w(d,c) that returns 0 when c <= d/2 and value in [0,1] otherwise can be used. Default is ‘linear’; other supplied choices are ‘majority’ and ‘strict’.

Returns
  The modularity function for partition A on HG

Return type
  float

algorithms.hypergraph_modularity.part2dict(A)
  Given a partition (list of sets), returns a dictionary mapping the part for each vertex; inverse function to dict2part

Parameters
  A (list of sets) – a partition of the vertices

Returns
  a dictionary with {vertex: partition index}

Return type
  dict

algorithms.hypergraph_modularity.strict(d, c)
  Hyperparameter for hypergraph modularity for d-edge with c vertices in the majority class. This corresponds to the strict rule for d-edge with c vertices in the majority class. This corresponds to the strict rule

Parameters
  • d (int) – Number of vertices in an edge
  • c (int) – Number of vertices in the majority class

Returns
  1 if c==d else 0

Return type
  bool

algorithms.hypergraph_modularity.two_section(HG)
  Creates a random walk based 2-section igraph Graph with transition weights defined by the weights of the hyperedges.

Parameters
  HG (hnx.Hypergraph)

Returns
  The 2-section graph built from HG
Return type
igraph.Graph

algorithms.laplacians_clustering module

Hypergraph Probability Transition Matrices, Laplacians, and Clustering

We construct hypergraph random walks utilizing optional “edge-dependent vertex weights”, which are weights associated with each vertex-hyperedge pair (i.e., cell weights on the incidence matrix). The probability transition matrix of this random walk is used to construct a normalized Laplacian matrix for the hypergraph. That normalized Laplacian then serves as the input for a spectral clustering algorithm. This spectral clustering algorithm, as well as the normalized Laplacian and other details of this methodology are described in


Please direct any inquiries concerning the clustering module to Sinan Aksoy, sinan.aksoy@pnnl.gov

algorithms.laplacians_clustering.get_pi(P)

Returns the eigenvector corresponding to the largest eigenvalue (in magnitude), normalized so its entries sum to 1. Intended for the probability transition matrix of a random walk on a (connected) hypergraph, in which case the output can be interpreted as the stationary distribution.

Parameters
\( P \) (csr matrix) – Probability transition matrix

Returns
\( \pi \) – Stationary distribution of random walk defined by \( P \)

Return type
numpy.ndarray

algorithms.laplacians_clustering.norm_lap(H, weights=False, index=True)

Normalized Laplacian matrix of the hypergraph. Symmetrizes the probability transition matrix of a hypergraph random walk using the stationary distribution, using the digraph Laplacian defined in:


and studied in the context of hypergraphs in:


Parameters

- \( H \) (hnx.Hypergraph) – The hypergraph must be connected, meaning there is a path linking any two vertices
- \( \text{weight} \) (bool, optional, default: False) – Uses cell_weights, if False, uniform weights are utilized.
- \( \text{index} \) (bool, optional) – Whether to return matrix-index to vertex-label mapping

Returns

- \( P \) (scipy.sparse.csr.csr_matrix) – Probability transition matrix of the random walk on the hypergraph
- \( \text{id} \) (list) – contains list of index of node ids for rows

5.5. HyperNetX Packages
algorithms.laplacians_clustering.prob_trans($H$, $weights=False$, $index=True$, $check_connected=True$)

The probability transition matrix of a random walk on the vertices of a hypergraph. At each step in the walk, the next vertex is chosen by:

1. Selecting a hyperedge $e$ containing the vertex with probability proportional to $w(e)$
2. Selecting a vertex $v$ within $e$ with probability proportional to $\gamma(v,e)$

If weights are not specified, then all weights are uniform and the walk is equivalent to a simple random walk. If weights are specified, the hyperedge weights $w(e)$ are determined from the weights $\gamma(v,e)$.

**Parameters**

- $H$ (*hnx.Hypergraph*) – The hypergraph must be connected, meaning there is a path linking any two vertices
- $weights$ (*bool, optional, default : False*) – Use the cell_weights associated with the hypergraph. If False, uniform weights are utilized.
- $index$ (*bool, optional*) – Whether to return matrix index to vertex label mapping

**Returns**

- $P$ (*scipy.sparse.csr.csr_matrix*) – Probability transition matrix of the random walk on the hypergraph
- $index$ (*list*) – contains list of index of node ids for rows

algorithms.laplacians_clustering.spec_clus($H$, $k$, $existing_lap=None$, $weights=False$)

Hypergraph spectral clustering of the vertex set into $k$ disjoint clusters using the normalized hypergraph Laplacian. Equivalent to the “RDC-Spec” Algorithm 1 in:


**Parameters**

- $H$ (*hnx.Hypergraph*) – The hypergraph must be connected, meaning there is a path linking any two vertices
- $k$ (*int*) – Number of clusters
- $existing_lap$ (*csr matrix, optional*) – Whether to use an existing Laplacian; otherwise, normalized hypergraph Laplacian will be utilized
- $weights$ (*bool, optional*) – Use the cell_weights of the hypergraph. If False uniform weights are used.

**Returns**

- $clusters$ – Vertex cluster dictionary, keyed by integers $0,...,k-1$, with lists of vertices as values.

**Return type**

- dict
### Algorithms.s.centrality_measures Module

#### S-Centrality Measures

We generalize graph metrics to s-metrics for a hypergraph by using its s-connected components. This is accomplished by computing the s-edge-adjacency matrix and constructing the corresponding graph of the matrix. We then use existing graph metrics on this representation of the hypergraph. In essence we construct an s-line graph corresponding to the hypergraph on which to apply our methods.


```python
algorithms.s.centrality_measures.s_betweenness_centrality(H, s=1, edges=True, normalized=True, return_singletons=True)
```

A centrality measure for an s-edge(node) subgraph of H based on shortest paths. Equals the betweenness centrality of vertices in the edge(node) s-linegraph.

In a graph (2-uniform hypergraph) the betweenness centrality of a vertex $\nu$ is the ratio of the number of non-trivial shortest paths between any pair of vertices in the graph that pass through $\nu$ divided by the total number of non-trivial shortest paths in the graph.

The centrality of edge to all shortest s-edge paths $V = \text{the set of vertices in the linegraph}$. $n = |V|$ $d = \text{shortest path distance}$

$$c_B(\nu) = \sum_{s \neq t \in V} \frac{\sigma(s, t|\nu)}{\sigma(s, t)}$$

**Parameters**

- **H** *(hnx.Hypergraph)*
- **s** *(int)* – s connectedness requirement
- **edges** *(bool, optional)* – determines if edge or node linegraph
- **normalized** – bool, default=False. If true the betweenness values are normalized by $2/(n-1)(n-2)$, where $n$ is the number of edges in H
- **return_singletons** *(bool, optional)* – if False will ignore singleton components of linegraph

**Returns**

A dictionary of s-betweenness centrality value of the edges.

**Return type**

dict

```python
algorithms.s.centrality_measures.s_closeness_centrality(H, s=1, edges=True, return_singletons=True, source=None)
```

In a connected component the reciprocal of the sum of the distance between an edge(node) and all other edges(nodes) in the component times the number of edges(nodes) in the component minus 1.

$V = \text{the set of vertices in the linegraph}$. $n = |V|$ $d = \text{shortest path distance}$

$$C(\nu) = \frac{n-1}{\sum_{v \neq u \in V} d(v, u)}$$

**Parameters**

5.5. HyperNetX Packages
HyperNetX Documentation, Release 2.3.5

- **H** (*hnx.Hypergraph*)
- **s** (*int*, *optional*)
- **edges** (*bool*, *optional*) – Indicates if method should compute edge linegraph (default) or node linegraph.
- **return_singletons** (*bool*, *optional*) – Indicates if method should return values for singleton components.
- **source** (*str*, *optional*) – Identifier of node or edge of interest for computing centrality

**Returns**

Returns the s-closeness centrality value of the edges(nodes). If source=None a dictionary of values for each s-edge in H is returned. If source then a single value is returned.

**Return type**
dict or float

`algorithms.s_centrality_measures.s_eccentricity(H, s=1, edges=True, source=None, return_singletons=True)`

The length of the longest shortest path from a vertex $u$ to every other vertex in the s-linegraph. $V$ = set of vertices in the s-linegraph $d$ = shortest path distance

\[
s\text{-ecc}(u) = \max\{d(u, v) : v \in V\}
\]

**Parameters**

- **H** (*hnx.Hypergraph*)
- **s** (*int*, *optional*)
- **edges** (*bool*, *optional*) – Indicates if method should compute edge linegraph (default) or node linegraph.
- **return_singletons** (*bool*, *optional*) – Indicates if method should return values for singleton components.
- **source** (*str*, *optional*) – Identifier of node or edge of interest for computing centrality

**Returns**

Returns the s-eccentricity value of the edges(nodes). If source=None a dictionary of values for each s-edge in H is returned. If source then a single value is returned. If the s-linegraph is disconnected, np.inf is returned.

**Return type**
dict or float

`algorithms.s_centrality_measures.s_harmonic_centrality(H, s=1, edges=True, source=None, normalized=False, return_singletons=True)`

A centrality measure for an s-edge subgraph of H. A value equal to 1 means the s-edge intersects every other s-edge in H. All values range between 0 and 1. Edges of size less than s return 0. If H contains only one s-edge a 0 is returned.

The denormalized reciprocal of the harmonic mean of all distances from $u$ to all other vertices. $V$ = the set of vertices in the linegraph. $d$ = shortest path distance

\[
C(u) = \sum_{v \neq u \in V} \frac{1}{d(v, u)}
\]

Normalized this becomes: $C(u) = \sum_{v \neq u \in V} \frac{1}{d(v, u)} \cdot \frac{1}{(n-1)(n-2)}$ where $n$ is the number vertices.
Parameters

- **H** *(hnx.Hypergraph)*
- **s** *(int, optional)*
- **edges** *(bool, optional)* – Indicates if method should compute edge linegraph (default) or node linegraph.
- **return_singletons** *(bool, optional)* – Indicates if method should return values for singleton components.
- **source** *(str, optional)* – Identifier of node or edge of interest for computing centrality

Returns

returns the s-harmonic closeness centrality value of the edges, a number between 0 and 1 inclusive. If source=None a dictionary of values for each s-edge in H is returned. If source then a single value is returned.

Return type
dict or float

algorithms.s_centrality_measures.s_harmonic_closeness_centrality(H, s=1, edge=None)

Module contents

algorithms.Gillespie_SIR(H, tau, gamma, transmission_function=<function threshold>,
initial_infecteds=None, initial_recovereds=None, rho=None, tmin=0, tmax=inf,
**args)


Parameters

- **H** *(HyperNetX Hypergraph object)*
- **tau** *(dictionary)* – Edge sizes as keys (must account for all edge sizes present) and rates of infection for each size (float)
- **gamma** *(float)* – The healing rate
- **transmission_function** *(lambda function, default: threshold)* – A lambda function that has required arguments (node, status, edge) and optional arguments
- **initial_infecteds** *(list or numpy array, default: None)* –Iterable of initially infected node uids
- **initial_recovereds** *(list or numpy array, default: None)* – An iterable of initially recovered node uids
- **rho** *(float from 0 to 1, default: None)* – The fraction of initially infected individuals. Both rho and initially infected cannot be specified.
- **tmin** *(float, default: 0)* – Time at the start of the simulation
- **tmax** *(float, default: float('Inf'))* – Time at which the simulation should be terminated if it hasn’t already.
- **return_full_data** *(bool, default: False)* – This returns all the infection and recovery events at each time if True.
• **args (Optional arguments to transmission function) – This allows user-defined transmission functions with extra parameters.

**Returns**

\(t, S, I, R\) – time (t), number of susceptible (S), infected (I), and recovered (R) at each time.

**Return type**

numpy arrays

**Notes**

Example:

```python
>>> import hypernetx.algorithms.contagion as contagion
>>> import random
>>> import hypernetx as hnx

>>> n = 1000
>>> m = 10000
>>> hyperedgeList = [random.sample(range(n), k=random.choice([2,3])) for i in range(m)]
>>> H = hnx.Hypergraph(hyperedgeList)
>>> tau = {2:0.1, 3:0.1}
>>> gamma = 0.1
>>> tmax = 100
>>> t, S, I, R = contagion.Gillespie_SIR(H, tau, gamma, rho=0.1, tmin=0, tmax=tmax)
```


**Parameters**

- \(H\) (HyperNetX Hypergraph object)
- \(\tau\) (dictionary) – Edge sizes as keys (must account for all edge sizes present) and rates of infection for each size (float)
- \(\gamma\) (float) – The healing rate
- \(\text{transmission\_function}\) (lambda function, default: \text{threshold}) – A lambda function that has required arguments (node, status, edge) and optional arguments
- \(\text{initial\_infecteds}\) (list or numpy array, default: \text{None}) – Iterable of initially infected node uids
- \(\rho\) (float from 0 to 1, default: \text{None}) – The fraction of initially infected individuals. Both \(\rho\) and initially infected cannot be specified.
- \(\text{tmin}\) (float, default: 0) – Time at the start of the simulation
- \(\text{tmax}\) (float, default: 100) – Time at which the simulation should be terminated if it hasn’t already.
- \(\text{return\_full\_data}\) (bool, default: False) – This returns all the infection and recovery events at each time if True.
**args (Optional arguments to transmission function) – This allows user-defined transmission functions with extra parameters.

Returns

- \( t, S, I \) – time (t), number of susceptible (S), and infected (I) at each time.

Return type

- numpy arrays

Notes

Example:

```python
>>> import hypernetx.algorithms.contagion as contagion
>>> import random
>>> import hypernetx as hnx

>>> n = 1000
>>> m = 10000
>>> hyperedgeList = [random.sample(range(n), k=random.choice([2,3])) for i in range(m)]
>>> H = hnx.Hypergraph(hyperedgeList)
>>> tau = {2:0.1, 3:0.1}
>>> gamma = 0.1
>>> tmax = 100
>>> t, S, I = contagion.Gillespie_SIS(H, tau, gamma, rho=0.1, tmin=0, tmax=tmax)
```

algorithms.add_to_column(M, i, j)

Replaces column \( i \) (of M) with logical xor between column \( i \) and \( j \)

Parameters

- \( M \) (np.array) – matrix
- \( i \) (int) – index of column being altered
- \( j \) (int) – index of column being added to altered

Returns

- \( N \)

Return type

- np.array

algorithms.add_to_row(M, i, j)

Replaces row \( i \) with logical xor between row \( i \) and \( j \)

Parameters

- \( M \) (np.array)
- \( i \) (int) – index of row being altered
- \( j \) (int) – index of row being added to altered

Returns

- \( N \)

Return type

- np.array
algorithms.betti(bd, k=None)

Generate the kth-betti numbers for a chain complex with boundary matrices given by bd

Parameters
- **bd** *(dict of k-boundary matrices keyed on dimension of domain)*
- **k** *(int, list or tuple, optional, default=None)* – list must be min value and max value of k values inclusive if None, then all betti numbers for dimensions of existing cells will be computed.

Returns
- **betti** – Description

Return type
dict

algorithms.betti_numbers(h, k=None)

Return the kth betti numbers for the simplicial homology of the ASC associated to h

Parameters
- **h** *(hnx.Hypergraph)* – Hypergraph to compute the betti numbers from
- **k** *(int or list, optional, default=None)* – list must be min value and max value of k values inclusive if None, then all betti numbers for dimensions of existing cells will be computed.

Returns
- **betti** – A dictionary of betti numbers keyed by dimension

Return type
dict

algorithms.bkMatrix(km1basis, kbasis)

Compute the boundary map from $C_{k-1}$-basis to $C_k$ basis with respect to $\mathbb{Z}_2$

Parameters
- **km1basis** *(indexable iterable)* – Ordered list of $k-1$ dimensional cell
- **kbasis** *(indexable iterable)* – Ordered list of $k$ dimensional cells

Returns
- **bk** – boundary matrix in $\mathbb{Z}_2$ stored as boolean

Return type
np.array

algorithms.boundary_group(image_basis)

Returns a csr_matrix with rows corresponding to the elements of the group generated by image basis over $\mathbb{Z}_2$

Parameters
- **image_basis** *(numpy.ndarray or scipy.sparse.csr_matrix)* – 2d-array of basis elements

Return type
scipy.sparse.csr_matrix

algorithms.chain_complex(h, k=None)

Compute the k-chains and k-boundary maps required to compute homology for all values in k

Parameters
• **h** (*hnx.Hypergraph*)

• **k** (*int or list of length 2, optional, default=None*) – k must be an integer greater than 0 or a list of length 2 indicating min and max dimensions to be computed. eg. if k = [1, 2] then 0, 1, 2, 3-chains and boundary maps for k=1, 2, 3 will be returned, if None then k = [1, max dimension of edge in h]

**Returns**

C, bd – C is a dictionary of lists bd is a dictionary of numpy arrays

**Return type**

dict

---

**algorithms.chung_lu_hypergraph(k1, k2)**

A function to generate an extension of Chung-Lu hypergraph as implemented by Mirah Shi and described for bipartite networks by Aksoy et al. in https://doi.org/10.1093/comnet/cnx001

**Parameters**

• **k1** (*dictionary*) – This a dictionary where the keys are node ids and the values are node degrees.

• **k2** (*dictionary*) – This a dictionary where the keys are edge ids and the values are edge degrees also known as edge sizes.

**Return type**

HyperNetX Hypergraph object

---

**Notes**

The sums of k1 and k2 should be roughly the same. If they are not the same, this function returns a warning but still runs. The output currently is a static Hypergraph object. Dynamic hypergraphs are not currently supported.

**Example:**

```python
>>> import hypernetx.algorithms.generative_models as gm
>>> 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 = gm.chung_lu_hypergraph(k1, k2)
```

---

**algorithms.collective_contagion(node, status, edge)**

The collective contagion mechanism described in “The effect of heterogeneity on hypergraph contagion models” by Landry and Restrepo https://doi.org/10.1063/5.0020034

**Parameters**

• **node** (*hashable*) – the node uid to infect (If it doesn’t have status “S”, it will automatically return False)

• **status** (*dictionary*) – The nodes are keys and the values are statuses (The infected state denoted with “I”)

• **edge** (*iterable*) – Iterable of node ids (node must be in the edge or it will automatically return False)

**Returns**

False if there is no potential to infect and True if there is.
Return type

bool

Notes

Example:

```python
>>> status = {0:"S", 1:"I", 2:"I", 3:"S", 4:"R"}
>>> collective_contagion(0, status, (0, 1, 2))
True
>>> collective_contagion(1, status, (0, 1, 2))
False
>>> collective_contagion(3, status, (0, 1, 2))
False
```

`algorithms.contagion_animation(fig, H, transition_events, node_state_color_dict, edge_state_color_dict, node_radius=1, fps=1)`

A function to animate discrete-time contagion models for hypergraphs. Currently only supports a circular layout.

Parameters

- **fig** (*matplotlib Figure object*)
- **H** (*HyperNetX Hypergraph object*)
- **transition_events** (*dictionary*) – The dictionary that is output from the discrete_SIS and discrete_SIR functions with return_full_data=True
- **node_state_color_dict** (*dictionary*) – Dictionary which specifies the colors of each node state. All node states must be specified.
- **edge_state_color_dict** (*dictionary*) – Dictionary with keys that are edge states and values which specify the colors of each edge state (can specify an alpha parameter). All edge-dependent transition states must be specified (most common is “I”) and there must be a default “OFF” setting.
- **node_radius** (*float, default: 1*) – The radius of the nodes to draw
- **fps** (*int > 0, default: 1*) – Frames per second of the animation

Return type

*matplotlib Animation object*

Notes

Example:

```python
>>> import hypernetx.algorithms.contagion as contagion
>>> import random
>>> import hypernetx as hnx
>>> import matplotlib.pyplot as plt
>>> from IPython.display import HTML

>>> n = 1000
>>> m = 10000
>>> hyperedgeList = [random.sample(range(n), k=random.choice([2,3])) for i in range(m)]
```

(continues on next page)
HyperNetX Documentation, Release 2.3.5

algorithms.dcsbm_hypergraph(k1, k2, g1, g2, omega)

A function to generate an extension of DCSBM hypergraph as implemented by Mirah Shi and described for
bipartite networks by Larremore et al. in https://doi.org/10.1103/PhysRevE.90.012805

Parameters

- **k1 (dictionary)** – This a dictionary where the keys are node ids and the values are node
degrees.
- **k2 (dictionary)** – This a dictionary where the keys are edge ids and the values are edge
degrees also known as edge sizes.
- **g1 (dictionary)** – 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 (dictionary)** – 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.

Return type

HyperNetX Hypergraph object

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.

The output currently is a static Hypergraph object. Dynamic hypergraphs are not currently supported.

Example:

```python
>>> n = 100
>>> k1 = {i : random.randint(1, 100) for i in range(n)}
>>> k2 = {i : sorted(k1.values())[i] for i in range(n)}
>>> g1 = {i : random.choice([0, 1]) for i in range(n)}
```
HyperNetX Documentation, Release 2.3.5

>>> g2 = {i : random.choice([0, 1]) for i in range(n)}
>>> omega = np.array([[100, 10], [10, 100]])
>>> H = gm.dcsbm_hypergraph(k1, k2, g1, g2, omega)

g2 = {i : random.choice([0, 1]) for i in range(n)}
omega = np.array([[100, 10], [10, 100]])
H = gm.dcsbm_hypergraph(k1, k2, g1, g2, omega)

algorithms.dict2part(D)

Given a dictionary mapping the part for each vertex, return a partition as a list of sets; inverse function to part2dict

Parameters

D (dict) – Dictionary keyed by vertices with values equal to integer index of the partition the vertex belongs to

Returns

List of sets; one set for each part in the partition

Return type

list

algorithms.discrete_SIR(H, tau, gamma, transmission_function=<function threshold>, initial_infecteds=None, initial_recovereds=None, rho=None, tmin=0, tmax=inf, dt=1.0, return_full_data=False, **args)

A discrete-time SIR model for hypergraphs similar to the construction described in “The effect of heterogeneity on hypergraph contagion models” by Landry and Restrepo https://doi.org/10.1063/5.0020034 and “Simplicial models of social contagion” by Iacopini et al. https://doi.org/10.1038/s41467-019-10431-6

Parameters

• H (HyperNetX Hypergraph object)
• tau (dictionary) – Edge sizes as keys (must account for all edge sizes present) and rates of infection for each size (float)
• gamma (float) – The healing rate
• transmission_function (lambda function, default: threshold) – A lambda function that has required arguments (node, status, edge) and optional arguments
• initial_infecteds (list or numpy array, default: None) – Iterable of initially infected node uids
• initial_recovereds (list or numpy array, default: None) – An iterable of initially recovered node uids
• rho (float from 0 to 1, default: None) – The fraction of initially infected individuals. Both rho and initially infected cannot be specified.
• tmin (float, default: 0) – Time at the start of the simulation
• tmax (float, default: float('Inf')) – Time at which the simulation should be terminated if it hasn’t already.
• dt (float > 0, default: 1.0) – Step forward in time that the simulation takes at each step.
• return_full_data (bool, default: False) – This returns all the infection and recovery events at each time if True.
• **args (Optional arguments to transmission function) – This allows user-defined transmission functions with extra parameters.

Returns

• if return_full_data –
dictionary
Time as the keys and events that happen as the values.

• else –
  t, S, I, R
  [numpy arrays] time (t), number of susceptible (S), infected (I), and recovered (R) at each
time.

Notes
Example:

```python
>>> import hypernetx.algorithms.contagion as contagion
>>> import random
>>> import hypernetx as hnx

>>> n = 1000
>>> m = 10000
>>> hyperedgeList = [random.sample(range(n), k=random.choice([2,3])) for i in
˓→range(m)]
>>> H = hnx.Hypergraph(hyperedgeList)
>>> tau = {2:0.1, 3:0.1}
>>> gamma = 0.1
>>> tmax = 100
>>> dt = 0.1
>>> t, S, I, R = contagion.discrete_SIR(H, tau, gamma, rho=0.1, tmin=0, tmax=tmax,␣
˓→dt=dt)
```

`algorithms.discrete_SIS(H, tau, gamma, transmission_function=<function threshold>,
  initial_infecteds=None, rho=None, tmin=0, tmax=100, dt=1.0,
  return_full_data=False, **args)`

A discrete-time SIS model for hypergraphs as implemented in “The effect of heterogeneity on hypergraph con-
tagion models” by Landry and Restrepo https://doi.org/10.1063/5.0020034 and “Simplicial models of social
contagion” by Iacopini et al. https://doi.org/10.1038/s41467-019-10431-6

Parameters

• H ([HyperNetX Hypergraph object]

• tau (dictionary) – Edge sizes as keys (must account for all edge sizes present) and rates
  of infection for each size (float)

• gamma (float) – The healing rate

• transmission_function (lambda function, default: threshold) – A lambda function that has required arguments (node, status, edge) and optional arguments

• initial_infecteds (list or numpy array, default: None) – Iterable of initially
  infected node uids

• rho (float from 0 to 1, default: None) – The fraction of initially infected indi-
 viduals. Both rho and initially infected cannot be specified.

• tmin (float, default: 0) – Time at the start of the simulation

• tmax (float, default: 100) – Time at which the simulation should be terminated if it
  hasn’t already.
dt (float > 0, default: 1.0) – Step forward in time that the simulation takes at each step.

return_full_data (bool, default: False) – This returns all the infection and recovery events at each time if True.

**args (Optional arguments to transmission function) – This allows user-defined transmission functions with extra parameters.

Returns

if return_full_data –

dictionary
    Time as the keys and events that happen as the values.

else –

t, S, I
    [numpy arrays] time (t), number of susceptible (S), and infected (I) at each time.

Notes

Example:

```python
>>> import hypernetx.algorithms.contagion as contagion
>>> import random
>>> import hypernetx as hnx

>>> n = 1000
>>> m = 10000
>>> hyperedgeList = [random.sample(range(n), k=random.choice([2,3])) for i in range(m)]
>>> H = hnx.Hypergraph(hyperedgeList)
>>> tau = {2:0.1, 3:0.1}
>>> gamma = 0.1
>>> tmax = 100
>>> dt = 0.1
>>> t, S, I = contagion.discrete_SIS(H, tau, gamma, rho=0.1, tmin=0, tmax=tmax, dt=dt)
```

algorithms.erdos_renyi_hypergraph(n, m, p, node_labels=None, edge_labels=None)

A function to generate an Erdos-Renyi hypergraph as implemented by Mirah Shi and described for bipartite networks by Aksoy et al. in https://doi.org/10.1093/comnet/cnx001

Parameters

- n (int) – Number of nodes
- m (int) – Number of edges
- p (float) – The probability that a bipartite edge is created
- node_labels (list, default=None) – Vertex labels
- edge_labels (list, default=None) – Hyperedge labels

Return type

HyperNetX Hypergraph object

Example:
>>> import hypernetx.algorithms.generative_models as gm
>>> n = 1000
>>> m = n
>>> p = 0.01
>>> H = gm.erdos_renyi_hypergraph(n, m, p)

algorithms.get_pi(P)

Returns the eigenvector corresponding to the largest eigenvalue (in magnitude), normalized so its entries sum to 1. Intended for the probability transition matrix of a random walk on a (connected) hypergraph, in which case the output can be interpreted as the stationary distribution.

Parameters

P (csr matrix) – Probability transition matrix

Returns

pi – Stationary distribution of random walk defined by P

Return type
	numpy.ndarray

algorithms.homology_basis(bd, k=None, boundary=False, **kwargs)

Compute a basis for the kth-simplicial homology group, $H_k$, defined by a chain complex $CS$ with boundary maps given by $bd =$ {...}

Parameters

- bd (dict) – dict of boundary matrices on k-chains to k-1 chains keyed on k if krangle is a tuple then all boundary matrices k in [krange[0]...krange[1]] inclusive must be in the dictionary
- k (int or list of ints, optional, default=None) – k must be a positive integer or a list of 2 integers indicating min and max dimensions to be computed, if none given all homology groups will be computed from available boundary matrices in bd
- boundary (bool) – option to return a basis for the boundary group from each dimension. Needed to compute the shortest generators in the homology group.

Returns

- basis (dict) – dict of generators as 0-1 tuples keyed by dim basis for dimension k will be returned only if bd[k] and bd[k+1] have been provided.
- im (dict) – dict of boundary group generators keyed by dim

algorithms.hypergraph_homology_basis(h, k=None, shortest=False, interpreted=True)

Computes the kth-homology groups mod 2 for the ASC associated with the hypergraph h for k in krange inclusive

Parameters

- h (hnx.Hypergraph)
- k (int or list of length 2, optional, default=None) – k must be an integer greater than 0 or a list of length 2 indicating min and max dimensions to be computed
- shortest (bool, optional, default=False) – option to look for shortest representative for each coset in the homology group, only good for relatively small examples
- interpreted (bool, optional, default=True) – if True will return an explicit basis in terms of the k-chains

Returns

- basis (list) – list of generators as k-chains as boolean vectors
• interpreted_basis – lists of kchains in basis

**algorithms.individual_contagion**(node, status, edge)

The individual contagion mechanism described in “The effect of heterogeneity on hypergraph contagion models” by Landry and Restrepo [https://doi.org/10.1063/5.0020034](https://doi.org/10.1063/5.0020034)

**Parameters**

- **node** (hashable) – The node uid to infect (If it doesn’t have status “S”, it will automatically return False)
- **status** (dictionary) – The nodes are keys and the values are statuses (The infected state denoted with “I”)
- **edge** (iterable) – Iterable of node ids (node must be in the edge or it will automatically return False)

**Returns**

False if there is no potential to infect and True if there is.

**Return type**

bool

**Notes**

Example:

```python
>>> status = {0:"S", 1:"I", 2:"I", 3:"S", 4:"R"}
>>> individual_contagion(0, status, (0, 1, 3))
True
>>> individual_contagion(1, status, (0, 1, 2))
False
>>> collective_contagion(3, status, (0, 3, 4))
False
```

**algorithms.interpret**(Ck, arr, labels=None)

Returns the data as represented in Ck associated with the arr

**Parameters**

- **Ck** (list) – a list of k-cells being referenced by arr
- **arr** (np.array) – array of 0-1 vectors
- **labels** (dict, optional) – dictionary of labels to associate to the nodes in the cells

**Returns**

list of k-cells referenced by data in Ck

**Return type**

list

**algorithms.kchainbasis**(h, k)

Compute the set of k dimensional cells in the abstract simplicial complex associated with the hypergraph.

**Parameters**

- **h** (*hnx.Hypergraph*)
- **k** (int) – dimension of cell
Returns

an ordered list of kchains represented as tuples of length k+1

Return type

list

See also:

hnx.hypergraph.toplexes

Notes

- Method works best if h is simple [Berge], i.e. no edge contains another and there are no duplicate edges (toplexes).
- Hypergraph node uids must be sortable.

algorithms.kumar(HG, delta=0.01, verbose=False)

Compute a partition of the vertices in hypergraph HG as per Kumar's algorithm

Parameters

• HG (hnx.Hypergraph)
• delta (float, optional) – convergence stopping criterion

Returns

A partition of the vertices in HG

Return type

list of sets

algorithms.last_step(HG, A, wdc=<function linear>, delta=0.01, verbose=False)

Given some initial partition L, compute a new partition of the vertices in HG as per Last-Step algorithm

Note: This is a very simple algorithm that tries moving nodes between communities to improve hypergraph modularity. It requires an initial non-trivial partition which can be obtained for example via graph clustering on the 2-section of HG, or via Kumar’s algorithm.

Parameters

• HG (hnx.Hypergraph)
• A (list of sets) – some initial partition of the vertices in HG
• wdc (func, optional) – Hyperparameter for hypergraph modularity
• delta (float, optional) – convergence stopping criterion
• verbose (boolean, optional) – If set, also returns progress after each pass through the vertices

Returns

A new partition for the vertices in HG

Return type

list of sets
algorithms.linear($d, c$)

Hyperparameter for hypergraph modularity\textsuperscript{Page 95, 2} for d-edge with c vertices in the majority class. This is the default choice for modularity() and last_step() functions.

Parameters

- $d$ (int) – Number of vertices in an edge
- $c$ (int) – Number of vertices in the majority class

Returns

$c/d$ if $c > d/2$ else $0$

Return type

float

algorithms.logical_dot($ar1, ar2$)

Returns the boolean equivalent of the dot product mod 2 on two 1-d arrays of the same length.

Parameters

- $ar1$ (numpy.ndarray) – 1-d array
- $ar2$ (numpy.ndarray) – 1-d array

Returns

boolean value associated with dot product mod 2

Return type

bool

Raises

HyperNetXError – If arrays are not of the same length an error will be raised.

algorithms.logical_matadd($mat1, mat2$)

Returns the boolean equivalent of matrix addition mod 2 on two binary arrays stored as type boolean

Parameters

- $mat1$ (np.ndarray) – 2-d array of boolean values
- $mat2$ (np.ndarray) – 2-d array of boolean values

Returns

mat – boolean matrix equivalent to the mod 2 matrix addition of the matrices as matrices over $\mathbb{Z}/2\mathbb{Z}$

Return type

np.ndarray

Raises

HyperNetXError – If dimensions are not equal an error will be raised.

algorithms.logical_matmul($mat1, mat2$)

Returns the boolean equivalent of matrix multiplication mod 2 on two binary arrays stored as type boolean

Parameters

- $mat1$ (np.ndarray) – 2-d array of boolean values
- $mat2$ (np.ndarray) – 2-d array of boolean values

Returns

mat – boolean matrix equivalent to the mod 2 matrix multiplication of the matrices as matrices over $\mathbb{Z}/2\mathbb{Z}$
Return type
np.ndarray

Raises
HyperNetXError – If inner dimensions are not equal an error will be raised.

algorithms.majority(d, c)
Hyperparameter for hypergraph modularity\textsuperscript{Page 95, 2} for d-edge with c vertices in the majority class. This corresponds to the majority rule\textsuperscript{Page 95, 3}

Parameters
- d (int) – Number of vertices in an edge
- c (int) – Number of vertices in the majority class

Returns
1 if c>d/2 else 0

Return type
bool

algorithms.majority_vote(node, status, edge)
The majority vote contagion mechanism. If a majority of neighbors are contagious, it is possible for an individual to change their opinion. If opinions are divided equally, choose randomly.

Parameters
- node (hashable) – The node uid to infect (If it doesn’t have status “S”, it will automatically return False)
- status (dictionary) – The nodes are keys and the values are statuses (The infected state denoted with “I”)
- edge (iterable) – Iterable of node ids (node must be in the edge or it will automatically return False

Returns
False if there is no potential to infect and True if there is.

Return type
bool

Notes
Example:

`>>> status = {0:"S", 1:"I", 2:"I", 3:"S", 4:"R"}
>>> majority_vote(0, status, (0, 1, 2))
True
>>> majority_vote(0, status, (0, 1, 2, 3))
True
>>> majority_vote(1, status, (0, 1, 2))
False
>>> majority_vote(3, status, (0, 1, 2))
False`
algorithms.matmulreduce(arr, reverse=False)

Recursively applies a ‘logical multiplication’ to a list of boolean arrays.
For arr = [arr[0], arr[1], arr[2]…arr[n]] returns product arr[0]arr[1]…arr[n] If reverse = True, returns product arr[n]arr[n-1]…arr[0]

Parameters
- arr (list of np.array) – list of nxm matrices represented as np.array
- reverse (bool, optional) – order to multiply the matrices

Returns
P – Product of matrices in the list

Return type
np.array

algorithms.modularity(HG, A, wdc=<function linear>)

Computes modularity of hypergraph HG with respect to partition A.

Parameters
- HG (hnx.Hypergraph) – The hypergraph with some precomputed attributes via: precompute_attributes(HG)
- A (list of sets) – Partition of the vertices in HG
- wdc (func, optional) – Hyperparameter for hypergraph modularity

Note: For ‘wdc’, any function of the format w(d,c) that returns 0 when c <= d/2 and value in [0,1] otherwise can be used. Default is ‘linear’; other supplied choices are ‘majority’ and ‘strict’.

Returns
The modularity function for partition A on HG

Return type
float

algorithms.norm_lap(H, weights=False, index=True)

Normalized Laplacian matrix of the hypergraph. Symmetrizes the probability transition matrix of a hypergraph random walk using the stationary distribution, using the digraph Laplacian defined in:


and studied in the context of hypergraphs in:


Parameters
- H (hnx.Hypergraph) – The hypergraph must be connected, meaning there is a path linking any two vertices
- weight (bool, optional, default : False) – Uses cell_weights, if False, uniform weights are utilized.
- index (bool, optional) – Whether to return matrix-index to vertex-label mapping

Returns
• **P** (*scipy.sparse.csr.csr_matrix*) – Probability transition matrix of the random walk on the hypergraph

• **id** (*list*) – contains list of index of node ids for rows

**algorithms.part2dict(A)**

Given a partition (list of sets), returns a dictionary mapping the part for each vertex; inverse function to dict2part

**Parameters**

- **A** (*list of sets*) – a partition of the vertices

**Returns**

- a dictionary with {vertex: partition index}

**Return type**

*dict*

**algorithms.prob_trans(H, weights=False, index=True, check_connected=True)**

The probability transition matrix of a random walk on the vertices of a hypergraph. At each step in the walk, the next vertex is chosen by:

1. Selecting a hyperedge e containing the vertex with probability proportional to w(e)

2. Selecting a vertex v within e with probability proportional to a gamma(v,e)

If weights are not specified, then all weights are uniform and the walk is equivalent to a simple random walk. If weights are specified, the hyperedge weights w(e) are determined from the weights gamma(v,e).

**Parameters**

- **H** (*hnx.Hypergraph*) – The hypergraph must be connected, meaning there is a path linking any two vertices

- **weights** (*bool, optional, default : False*) – Use the cell_weights associated with the hypergraph If False, uniform weights are utilized.

- **index** (*bool, optional*) – Whether to return matrix index to vertex label mapping

**Returns**

- **P** (*scipy.sparse.csr.csr_matrix*) – Probability transition matrix of the random walk on the hypergraph

- **index** (*list*) – contains list of index of node ids for rows

**algorithms.reduced_row_echelon_form_mod2(M)**

Computes the invertible transformation matrices needed to compute the reduced row echelon form of M modulo 2

**Parameters**

- **M** (*np.array*) – a rectangular matrix with elements in $Z_2$

**Returns**

- **L, S, Linv** – LM = S where S is the reduced echelon form of M and M = LinvS

**Return type**

*np.arrays*

**algorithms.s_betweenness_centrality(H, s=1, edges=True, normalized=True, return_singletons=True)**

A centrality measure for an s-edge(node) subgraph of H based on shortest paths. Equals the betweenness centrality of vertices in the edge(node) s-linegraph.
In a graph (2-uniform hypergraph) the betweenness centrality of a vertex $v$ is the ratio of the number of non-trivial shortest paths between any pair of vertices in the graph that pass through $v$ divided by the total number of non-trivial shortest paths in the graph.

$$c_B(v) = \sum_{s \neq t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)}$$

Parameters

- $H$ (hnx.Hypergraph)
- $s$ (int) – $s$ connectedness requirement
- $edges$ (bool, optional) – determines if edge or node linegraph
- $normalized$ – bool, default=False. If true the betweenness values are normalized by $2/(l((n-l)/(n-2)))$, where $n$ is the number of edges in $H$
- $return\_singletons$ (bool, optional) – if False will ignore singleton components of linegraph

Returns

A dictionary of $s$-betweenness centrality value of the edges.

Return type

dict

`algorithms.s_closeness_centrality(H, s=1, edges=True, return_singletons=True, source=None)`

In a connected component the reciprocal of the sum of the distance between an edge(node) and all other edges(nodes) in the component times the number of edges(nodes) in the component minus 1.

$V$ = the set of vertices in the linegraph. $n = |V|$ $d$ = shortest path distance

$$C(u) = \frac{n - 1}{\sum_{v \neq u \in V} d(v, u)}$$

Parameters

- $H$ (hnx.Hypergraph)
- $s$ (int)
- $edges$ (bool, optional) – Indicates if method should compute edge linegraph (default) or node linegraph.
- $return\_singletons$ (bool, optional) – Indicates if method should return values for singleton components.
- $source$ (str, optional) – Identifier of node or edge of interest for computing centrality

Returns

returns the $s$-closeness centrality value of the edges(nodes). If source=None a dictionary of values for each $s$-edge in $H$ is returned. If source then a single value is returned.

Return type

dict or float
algorithms.s_eccentricity(H, s=1, edges=True, source=None, return_singletons=True)

The length of the longest shortest path from a vertex $u$ to every other vertex in the s-linegraph. $V$ = set of vertices in the s-linegraph $d$ = shortest path distance

$$s\text{-ecc}(u) = \max\{d(u, v) : v \in V\}$$

Parameters
- H (hnx.Hypergraph)
- s (int, optional)
- edges (bool, optional) – Indicates if method should compute edge linegraph (default) or node linegraph.
- return_singletons (bool, optional) – Indicates if method should return values for singleton components.
- source (str, optional) – Identifier of node or edge of interest for computing centrality

Returns
returns the s-eccentricity value of the edges(nodes). If source=None a dictionary of values for each s-edge in H is returned. If source then a single value is returned. If the s-linegraph is disconnected, np.inf is returned.

Return type
dict or float

algorithms.s_harmonic_centrality(H, s=1, edges=True, source=None, normalized=False, return_singletons=True)

A centrality measure for an s-edge subgraph of H. A value equal to 1 means the s-edge intersects every other s-edge in H. All values range between 0 and 1. Edges of size less than s return 0. If H contains only one s-edge a 0 is returned.

The denormalized reciprocal of the harmonic mean of all distances from $u$ to all other vertices. $V$ = the set of vertices in the linegraph. $d$ = shortest path distance

$$C(u) = \sum_{v \neq u \in V} \frac{1}{d(v, u)}$$

Normalized this becomes: $SC(u) = \sum_{v \neq u \in V} \frac{1}{d(v, u)} \cdot \frac{2}{(n-1)(n-2)}$ where $n$ is the number vertices.

Parameters
- H (hnx.Hypergraph)
- s (int, optional)
- edges (bool, optional) – Indicates if method should compute edge linegraph (default) or node linegraph.
- return_singletons (bool, optional) – Indicates if method should return values for singleton components.
- source (str, optional) – Identifier of node or edge of interest for computing centrality

Returns
returns the s-harmonic closeness centrality value of the edges, a number between 0 and 1 inclusive. If source=None a dictionary of values for each s-edge in H is returned. If source then a single value is returned.
Return type

dict or float

algorithms.s_harmonic_closeness_centrality($H, s=1, edge=None$)

Computes the \( s \) harmonic closeness centrality

\[ \text{algorithms.smith_normal_form_mod2}(M) \]

Computes the invertible transformation matrices needed to compute the Smith Normal Form of \( M \) modulo 2

Parameters

- \( M \) (np.array) – a rectangular matrix with data type bool
- \( \text{track} \) (bool) – if track=True will print out the transformation as \( \mathbb{Z}/2\mathbb{Z} \) matrix as it discovers \( L[i] \) and \( R[j] \)

Returns

- \( L, R, S, \text{Linv} \) – \( LMR = S \) is the Smith Normal Form of the matrix \( M \).

Return type

np.arrays

Note: Given a \( mxn \) matrix \( M \) with entries in \( \mathbb{Z}_2 \) we start with the equation: \( LMR = S \), where \( L = I_m \), and \( SR = I_n \) are identity matrices and \( S = M \). We repeatedly apply actions to the left and right side of the equation to transform \( S \) into a diagonal matrix. For each action applied to the left side we apply its inverse action to the right side of \( I_m \) to generate \( SL^{-1} \). Finally we verify: \( LMR = S \) and \( LLinv = I_m \).

algorithms.spec_clus($H, k, \text{existing_lap=None, weights=False}$)

Hypergraph spectral clustering of the vertex set into \( k \) disjoint clusters using the normalized hypergraph Laplacian. Equivalent to the “RDC-Spec” Algorithm 1 in:


Parameters

- \( H \) (hnx.Hypergraph) – The hypergraph must be connected, meaning there is a path linking any two vertices
- \( k \) (int) – Number of clusters
- \( \text{existing_lap} \) (csr matrix, optional) – Whether to use an existing Laplacian; otherwise, normalized hypergraph Laplacian will be utilized
- \( \text{weights} \) (bool, optional) – Use the cell_weights of the hypergraph. If False uniform weights are used.

Returns

- \( \text{clusters} \) – Vertex cluster dictionary, keyed by integers 0,...,\( k-1 \), with lists of vertices as values.

Return type

dict

algorithms.strict($d, c$)

Hyperparameter for hypergraph modularity\(^{Page 95.2}\) for \( d \)-edge with \( c \) vertices in the majority class. This corresponds to the strict rule\(^{Page 95.3}\)

Parameters

- \( d \) (int) – Number of vertices in an edge
- \( c \) (int) – Number of vertices in the majority class
Returns
1 if c==d else 0

Return type
bool

`algorithms.swap_columns(i, j, *args)`
Swaps ith and jth column of each matrix in args Returns a list of new matrices

Parameters
- **i** (int)
- **j** (int)
- **args** (np.arrays)

Returns
list of copies of args with ith and jth column swapped

Return type
list

`algorithms.swap_rows(i, j, *args)`
Swaps ith and jth row of each matrix in args Returns a list of new matrices

Parameters
- **i** (int)
- **j** (int)
- **args** (np.arrays)

Returns
list of copies of args with ith and jth row swapped

Return type
list

`algorithms.threshold(node, status, edge, tau=0.1)`
The threshold contagion mechanism

Parameters
- **node** (hashable) – The node uid to infect (If it doesn’t have status “S”, it will automatically return False)
- **status** (dictionary) – The nodes are keys and the values are statuses (The infected state denoted with “I”)
- **edge** (iterable) – Iterable of node ids (node must be in the edge or it will automatically return False)
- **tau** (float between 0 and 1, default: 0.1) – The fraction of nodes in an edge that must be infected for the edge to be able to transmit to the node

Returns
False if there is no potential to infect and True if there is.

Return type
bool
Notes

Example:

```python
>>> status = {0:"S", 1:"I", 2:"I", 3:"S", 4:"R"}
>>> threshold(0, status, (0, 2, 3, 4), tau=0.2)
True
>>> threshold(0, status, (0, 2, 3, 4), tau=0.5)
False
>>> threshold(3, status, (1, 2, 3), tau=1)
False
```

`algorithms.two_section(HG)`

Creates a random walk based 2-section igraph Graph with transition weights defined by the weights of the hyperedges.

**Parameters**

HG ([hnx.Hypergraph](https://example.com/))

**Returns**

The 2-section graph built from HG

**Return type**

`igraph.Graph`

5.5.3 drawing

drawing package

Submodules

drawing.rubber_band module

drawing.rubber_band.draw(H, pos=None, with_color=True, with_node_counts=False, with_edge_counts=False, layout=<function spring_layout>, layout_kwargs={}, ax=None, node_radius=None, edges_kwargs={}, nodes_kwargs={}, edge_labels_on_edge=True, edge_labels={}, edge_labels_kwargs={}, node_labels={}, node_labels_kwargs={}, with_edge_labels=True, with_node_labels=True, node_label_alpha=0.35, edge_label_alpha=0.35, with_additional_edges=None, contain_hyper_edges=False, additional_edges_kwargs={}, return_pos=False)

Draw a hypergraph as a Matplotlib figure

By default this will draw a colorful “rubber band” like hypergraph, where convex hulls represent edges and are drawn around the nodes they contain.

This is a convenience function that wraps calls with sensible parameters to the following lower-level drawing functions:

- `draw_hyper_edges`,
- `draw_hyper_edge_labels`,
- `draw_hyper_labels`, and
- `draw_hyper_nodes`
The default layout algorithm is `nx.spring_layout`, but other layouts can be passed in. The Hypergraph is converted to a bipartite graph, and the layout algorithm is passed the bipartite graph.

If you have a pre-determined layout, you can pass in a “pos” dictionary. This is a dictionary mapping from node id’s to x-y coordinates. For example:

```python
>>> pos = {
>>>    'A': (0, 0),
>>>    'B': (1, 2),
>>>    'C': (5, -3)
>>> }
```

will position the nodes {A, B, C} manually at the locations specified. The coordinate system is in Matplotlib “data coordinates”, and the figure will be centered within the figure.

By default, this will draw in a new figure, but the axis to render in can be specified using `ax`.

This approach works well for small hypergraphs, and does not guarantee a rigorously “correct” drawing. Overlapping of sets in the drawing generally implies that the sets intersect, but sometimes sets overlap if there is no intersection. It is not possible, in general, to draw a “correct” hypergraph this way for an arbitrary hypergraph, in the same way that not all graphs have planar drawings.

**Parameters**

- `H (hnx.Hypergraph) – the entity to be drawn`
- `pos (dict) – mapping of node and edge positions to R^2`
- `with_color (bool) – set to False to disable color cycling of edges`
- `with_node_counts (bool) – set to True to replace the label for collapsed nodes with the number of elements`
- `with_edge_counts (bool) – set to True to label collapsed edges with number of elements`
- `layout (function) – layout algorithm to compute`
- `layout_kwargs (dict) – keyword arguments passed to layout function`
- `ax (Axis) – matplotlib axis on which the plot is rendered`
- `edges_kwargs (dict) – keyword arguments passed to matplotlib.collections.PolyCollection for edges`
- `node_radius (None, int, float, or dict) – radius of all nodes, or dictionary of node:value; the default (None) calculates radius based on number of collapsed nodes; reasonable values range between 1 and 3`
- `nodes_kwargs (dict) – keyword arguments passed to matplotlib.collections.PolyCollection for nodes`
- `edge_labels_on_edge (bool) – whether to draw edge labels on the edge (rubber band) or inside`
- `edge_labels_kwargs (dict) – keyword arguments passed to matplotlib.annotate for edge labels`
- `node_labels_kwargs (dict) – keyword arguments passed to matplotlib.annotate for node labels`
- `with_edge_labels (bool) – set to False to make edge labels invisible`
- `with_node_labels (bool) – set to False to make node labels invisible`
• **node_label_alpha** *(float)* – the transparency (alpha) of the box behind text drawn in the figure for node labels
• **edge_label_alpha** *(float)* – the transparency (alpha) of the box behind text drawn in the figure for edge labels
• **with_additional_edges** *(networkx.Graph)* – ...
• **contain_hyper_edges** *(bool)* – whether the rubber band should be drawn around the location of the edge in the bipartite graph. This may be invisible unless “with_additional_edges” contains this information.

drawing.rubber_band.draw_hyper_edge_labels(H, pos, polys, labels={}, edge_labels_on_edge=True, ax=None, **kwargs)

Draws a label on the hyper edge boundary.

Should be passed Matplotlib PolyCollection representing the hyper-edges, see the return value of draw_hyper_edges.

The label will be draw on the least curvy part of the polygon, and will be aligned parallel to the orientation of the polygon where it is drawn.

**Parameters**

- **H** *(hnx.Hypergraph)* – the entity to be drawn
- **polys** *(PolyCollection)* – collection of polygons returned by draw_hyper_edges
- **labels** *(dict)* – mapping of node id to string label
- **ax** *(Axis)* – matplotlib axis on which the plot is rendered
- **kwargs** *(dict)* – Keyword arguments are passed through to Matplotlib’s annotate function.

drawing.rubber_band.draw_hyper_edges(H, pos, ax=None, node_radius={}, contain_hyper_edges=False, dr=None, **kwargs)

Draws a convex hull around the nodes contained within each edge in H

**Parameters**

- **H** *(hnx.Hypergraph)* – the entity to be drawn
- **pos** *(dict)* – mapping of node and edge positions to R^2
- **node_radius** *(dict)* – mapping of node to R^1 (radius of each node)
- **dr** *(float)* – the spacing between concentric rings
- **ax** *(Axis)* – matplotlib axis on which the plot is rendered
- **kwargs** *(dict)* – keyword arguments, e.g., linewidth, facecolors, are passed through to the PolyCollection constructor

**Returns**

a Matplotlib PolyCollection that can be further styled

**Return type**

PolyCollection

drawing.rubber_band.draw_hyper_labels(H, pos, node_radius={}, ax=None, labels={}, **kwargs)

Draws text labels for the hypergraph nodes.

The label is drawn to the right of the node. The node radius is needed (see draw_hyper_nodes) so the text can be offset appropriately as the node size changes.
The text label can be customized by passing in a dictionary, labels, mapping a node to its custom label. By
default, the label is the string representation of the node.

Keyword arguments are passed through to Matplotlib’s annotate function.

**Parameters**

- H (hnx.Hypergraph) – the entity to be drawn
- pos (dict) – mapping of node and edge positions to R^2
- node_radius (dict) – mapping of node to R^1 (radius of each node)
- ax (Axis) – matplotlib axis on which the plot is rendered
- labels (dict) – mapping of node to text label
- kwargs (dict) – keyword arguments passed to matplotlib.annotate

**drawing.rubber_band.draw_hyper_nodes**(H, pos, node_radius={}, r0=None, ax=None, **kwargs)

Draws a circle for each node in H.

The position of each node is specified by a dictionary/list-like, pos, where pos[v] is the xy-coordinate for the vertex. The radius of each node can be specified as a dictionary where node_radius[v] is the radius. If a node is missing from this dictionary, or the node_radius is not specified at all, a sensible default radius is chosen based on distances between nodes given by pos.

**Parameters**

- H (hnx.Hypergraph) – the entity to be drawn
- pos (dict) – mapping of node and edge positions to R^2
- node_radius (dict) – mapping of node to R^1 (radius of each node)
- r0 (float) – minimum distance that concentric rings start from the node position
- ax (Axis) – matplotlib axis on which the plot is rendered
- kwargs (dict) – keyword arguments, e.g., linewidth, facecolors, are passed through to the PolyCollection constructor

**Returns**

a Matplotlib PolyCollection that can be further styled

**Return type**

PolyCollection

**drawing.rubber_band.get_default_radius**(H, pos)

Calculate a reasonable default node radius

This function iterates over the hyper edges and finds the most distant pair of points given the positions provided. Then, the node radius is a fraction of the median of this distance take across all hyper-edges.

**Parameters**

- H (hnx.Hypergraph) – the entity to be drawn
- pos (dict) – mapping of node and edge positions to R^2

**Returns**

the recommended radius

**Return type**

float
drawing.rubber_band.layout_hyper_edges(H, pos, node_radius={}, dr=None, contain_hyper_edges=False)

Draws a convex hull for each edge in H.

Position of the nodes in the graph is specified by the position dictionary, pos. Convex hulls are spaced out such that if one set contains another, the convex hull will surround the contained set. The amount of spacing added between hulls is specified by the parameter, dr.

Parameters
- H (hnx.Hypergraph) – the entity to be drawn
- pos (dict) – mapping of node and edge positions to R^2
- node_radius (dict) – mapping of node to R^1 (radius of each node)
- dr (float) – the spacing between concentric rings
- ax (Axis) – matplotlib axis on which the plot is rendered

Returns
A mapping from hyper edge ids to paths (Nx2 numpy matrices)

Return type
dict

drawing.rubber_band.layout_node_link(H, G=None, layout=<function spring_layout>, **kwargs)

Helper function to use a NetworkX-like graph layout algorithm on a Hypergraph.

The hypergraph is converted to a bipartite graph, allowing the usual graph layout techniques to be applied.

Parameters
- H (hnx.Hypergraph) – the entity to be drawn
- G (Graph) – an additional set of links to consider during the layout process
- layout (function) – the layout algorithm which accepts a NetworkX graph and keyword arguments
- kwargs (dict) – Keyword arguments are passed through to the layout algorithm

Returns
mapping of node and edge positions to R^2

Return type
dict

drawing.two_column module

drawing.two_column.draw(H, with_node_labels=True, with_edge_labels=True, with_node_counts=False, with_edge_counts=False, with_color=True, edge_kwargs=None, ax=None)

Draw a hypergraph using a two-column layout.

This is intended reproduce an illustrative technique for bipartite graphs and hypergraphs that is typically used in papers and textbooks.

The left column is reserved for nodes and the right column is reserved for edges. A line is drawn between a node an an edge

The order of nodes and edges is optimized to reduce line crossings between the two columns. Spacing between disconnected components is adjusted to make the diagram easier to read, by reducing the angle of the lines.

Parameters
• H (hnx.Hypergraph) – the entity to be drawn
• with_node_labels (bool) – False to disable node labels
• with_edge_labels (bool) – False to disable edge labels
• with_node_counts (bool) – set to True to label collapsed nodes with number of elements
• with_edge_counts (bool) – set to True to label collapsed edges with number of elements
• with_color (bool) – set to False to disable color cycling of hyper edges
• edge_kwargs (dict) – keyword arguments to pass to matplotlib.LineCollection
  • ax (Axis) – matplotlib axis on which the plot is rendered

drawing.two_column.draw_hyper_edges(H, pos, ax=None, **kwargs)

Renders hyper edges for the two column layout.

Each node-hyper edge membership is rendered as a line connecting the node in the left column to the edge in the right column.

Parameters

• H (hnx.Hypergraph) – the entity to be drawn
• pos (dict) – mapping of node and edge positions to R^2
• ax (Axis) – matplotlib axis on which the plot is rendered
• kwargs (dict) – keyword arguments passed to matplotlib.LineCollection

Returns

the hyper edges

Return type

LineCollection

drawing.two_column.draw_hyper_labels(H, pos, labels={}, with_node_labels=True, with_edge_labels=True, ax=None)

Renders hyper labels (nodes and edges) for the two column layout.

Parameters

• H (hnx.Hypergraph) – the entity to be drawn
• pos (dict) – mapping of node and edge positions to R^2
• labels (dict) – custom labels for nodes and edges can be supplied
• with_node_labels (bool) – False to disable node labels
• with_edge_labels (bool) – False to disable edge labels
• ax (Axis) – matplotlib axis on which the plot is rendered
• kwargs (dict) – keyword arguments passed to matplotlib.LineCollection

drawing.two_column.layout_two_column(H, spacing=2)

Two column (bipartite) layout algorithm.

This algorithm first converts the hypergraph into a bipartite graph and then computes connected components. Disconnected components are handled independently and then stacked together.

Within a connected component, the spectral ordering of the bipartite graph provides a quick and dirty ordering that minimizes edge crossings in the diagram.

Parameters
HyperNetX Documentation, Release 2.3.5

- \( H (\text{hnx.Hypergraph}) \) – the entity to be drawn
- \( \text{spacing} (\text{float}) \) – amount of whitespace between disconnected components

**drawing.util module**

drawing.util.get_collapsed_size(v)

drawing.util.get_frozenset_label(S, count=False, override={})

Helper function for rendering the labels of possibly collapsed nodes and edges

**Parameters**

- \( S (\text{iterable}) \) – list of entities to be labeled
- \( \text{count} (\text{bool}) \) – True if labels should be counts of entities instead of list

**Returns**

mapping of entity to its string representation

**Return type**

dict
drawing.util.get_line_graph(H, collapse=True)

Computes the line graph, a directed graph, where a directed edge \((u, v)\) exists if the edge \(u\) is a subset of the edge \(v\) in the hypergraph.

**Parameters**

- \( H (\text{hnx.Hypergraph}) \) – the entity to be drawn
- \( \text{collapse} (\text{bool}) \) – True if edges should be added if hyper edges are identical

**Returns**

A directed graph

**Return type**

networkx.DiGraph
drawing.util.get_set_layering(H, collapse=True)

Computes a layering of the edges in the hyper graph.

In this layering, each edge is assigned a level. An edge \(u\) will be above (e.g., have a smaller level value) another edge \(v\) if \(v\) is a subset of \(u\).

**Parameters**

- \( H (\text{hnx.Hypergraph}) \) – the entity to be drawn
- \( \text{collapse} (\text{bool}) \) – True if edges should be added if hyper edges are identical

**Returns**

a mapping of vertices in \(H\) to integer levels

**Return type**

dict
drawing.util.inflate(items, v)
drawing.util.inflate_kwargs(items, kwargs)

Helper function to expand keyword arguments.

**Parameters**
• \(n\) (int) – length of resulting list if argument is expanded
• \(kwargs\) (dict) – keyword arguments to be expanded

Returns
dictionary with same keys as \(kwargs\) and whose values are lists of length \(n\)

Return type
dict
drawing.util.transpose_inflated_kwargs(inflated)

Module contents
drawing.draw(H, pos=None, with_color=True, with_node_counts=False, with_edge_counts=False, layout=<function spring_layout>, layout_kwargs={}, ax=None, node_radius=None, edges_kwargs={}, nodes_kwargs={}, edge_labels_on_edge=True, edge_labels={}, with_edge_labels=True, with_node_labels=True, node_label_alpha=0.35, edge_label_alpha=0.35, with_additional_edges=None, contain_hyper_edges=False, additional_edges_kwargs={}, return_pos=False)

Draw a hypergraph as a Matplotlib figure
By default this will draw a colorful “rubber band” like hypergraph, where convex hulls represent edges and are drawn around the nodes they contain.

This is a convenience function that wraps calls with sensible parameters to the following lower-level drawing functions:
• draw_hyper_edges,
• draw_hyper_edge_labels,
• draw_hyper_labels, and
• draw_hyper_nodes

The default layout algorithm is \(nx.spring_layout\), but other layouts can be passed in. The Hypergraph is converted to a bipartite graph, and the layout algorithm is passed the bipartite graph.

If you have a pre-determined layout, you can pass in a “pos” dictionary. This is a dictionary mapping from node id’s to x-y coordinates. For example:

```python
>>> pos = {
>>>    'A': (0, 0),
>>>    'B': (1, 2),
>>>    'C': (5, -3)
>>> }
```

will position the nodes \{A, B, C\} manually at the locations specified. The coordinate system is in Matplotlib “data coordinates”, and the figure will be centered within the figure.

By default, this will draw in a new figure, but the axis to render in can be specified using \(ax\).

This approach works well for small hypergraphs, and does not guarantee a rigorously “correct” drawing. Overlapping of sets in the drawing generally implies that the sets intersect, but sometimes sets overlap if there is no intersection. It is not possible, in general, to draw a “correct” hypergraph this way for an arbitrary hypergraph, in the same way that not all graphs have planar drawings.

Parameters
• H (hnx.Hypergraph) – the entity to be drawn
• pos (dict) – mapping of node and edge positions to R^2
• with_color (bool) – set to False to disable color cycling of edges
• with_node_counts (bool) – set to True to replace the label for collapsed nodes with the number of elements
• with_edge_counts (bool) – set to True to label collapsed edges with number of elements
• layout (function) – layout algorithm to compute
• layout_kwargs (dict) – keyword arguments passed to layout function
• ax (Axis) – matplotlib axis on which the plot is rendered
• edges_kwargs (dict) – keyword arguments passed to matplotlib.collections.PolyCollection for edges
• node_radius (None, int, float, or dict) – radius of all nodes, or dictionary of node:value; the default (None) calculates radius based on number of collapsed nodes; reasonable values range between 1 and 3
• nodes_kwargs (dict) – keyword arguments passed to matplotlib.collections.PolyCollection for nodes
• edge_labels_on_edge (bool) – whether to draw edge labels on the edge (rubber band) or inside
• edge_labels_kwargs (dict) – keyword arguments passed to matplotlib.annotate for edge labels
• node_labels_kwargs (dict) – keyword arguments passed to matplotlib.annotate for node labels
• with_edge_labels (bool) – set to False to make edge labels invisible
• with_node_labels (bool) – set to False to make node labels invisible
• node_label_alpha (float) – the transparency (alpha) of the box behind text drawn in the figure for node labels
• edge_label_alpha (float) – the transparency (alpha) of the box behind text drawn in the figure for edge labels
• with_additional_edges (networkx.Graph) – ...
• contain_hyper_edges (bool) – whether the rubber band should be drawn around the location of the edge in the bipartite graph. This may be invisible unless “with_additional_edges” contains this information.

```python
drawing.draw_two_column(H, with_node_labels=True, with_edge_labels=True, with_node_counts=False, with_edge_counts=False, with_color=True, edge_kwargs=None, ax=None)
```

Draw a hypergraph using a two-column layout.

This is intended to reproduce an illustrative technique for bipartite graphs and hypergraphs that is typically used in papers and textbooks.

The left column is reserved for nodes and the right column is reserved for edges. A line is drawn between a node and an edge

The order of nodes and edges is optimized to reduce line crossings between the two columns. Spacing between disconnected components is adjusted to make the diagram easier to read, by reducing the angle of the lines.

Parameters
HyperNetX Documentation, Release 2.3.5

- \( H (\text{hnx.Hypergraph}) \) – the entity to be drawn
- \texttt{with_node_labels} (bool) – False to disable node labels
- \texttt{with_edge_labels} (bool) – False to disable edge labels
- \texttt{with_node_counts} (bool) – set to True to label collapsed nodes with number of elements
- \texttt{with_edge_counts} (bool) – set to True to label collapsed edges with number of elements
- \texttt{with_color} (bool) – set to False to disable color cycling of hyper edges
- \texttt{edge_kwargs} (dict) – keyword arguments to pass to matplotlib.LineCollection
- \texttt{ax} (Axes) – matplotlib axis on which the plot is rendered

5.5.4 reports

reports package

Submodules

reports.descriptive_stats module

This module contains methods which compute various distributions for hypergraphs:
- Edge size distribution
- Node degree distribution
- Component size distribution
- Toplex size distribution
- Diameter

Also computes general hypergraph information: number of nodes, edges, cells, aspect ratio, incidence matrix density

\texttt{reports.descriptive_stats.centrality_stats}(X)

Computes basic centrality statistics for \( X \)

**Parameters**
- \( X \) – an iterable of numbers

**Returns**
- \([\text{min}, \text{max}, \text{mean}, \text{median}, \text{standard deviation}]\) – List of centrality statistics for \( X \)

**Return type**
- list

\texttt{reports.descriptive_stats.comp_dist}(H, aggregated=False)

Computes component sizes, number of nodes.

**Parameters**
- \( H (\text{hnx.Hypergraph}) \)
- \texttt{aggregated} – If aggregated is True, returns a dictionary of component sizes (number of nodes) and counts. If aggregated is False, returns a list of components sizes in \( H \).

**Returns**
- \texttt{comp_dist} – List of component sizes or dictionary of component size distribution
**Return type**

list or dictionary

**See also:**

`s_comp_dist`

`reports.descriptive_stats.degree_dist(H, aggregated=False)`

Computes degrees of nodes of a hypergraph.

**Parameters**

- `H` (*hnx.Hypergraph*)
- `aggregated` – If aggregated is True, returns a dictionary of degrees and counts. If aggregated is False, returns a list of degrees in `H`.

**Returns**

`degree_dist` – List of degrees or dictionary of degree distribution

**Return type**

list or dict

`reports.descriptive_stats.dist_stats(H)`

Computes many basic hypergraph stats and puts them all into a single dictionary object

- `nrows` = number of nodes (rows in the incidence matrix)
- `ncols` = number of edges (columns in the incidence matrix)
- `aspect ratio` = `nrows/ncols`
- `ncells` = number of filled cells in incidence matrix
- `density` = `ncells/(nrows*ncols)`
- `node degree list` = `degree_dist(H)`
- `node degree dist` = `centrality_stats(degree_dist(H))`
- `node degree hist` = `Counter(degree_dist(H))`
- `max node degree` = `max(degree_dist(H))`
- `edge size list` = `edge_size_dist(H)`
- `edge size dist` = `centrality_stats(edge_size_dist(H))`
- `edge size hist` = `Counter(edge_size_dist(H))`
- `max edge size` = `max(edge_size_dist(H))`
- `comp nodes list` = `s_comp_dist(H, s=1, edges=False)`
- `comp nodes dist` = `centrality_stats(s_comp_dist(H, s=1, edges=False))`
- `comp nodes hist` = `Counter(s_comp_dist(H, s=1, edges=False))`
- `comp edges list` = `s_comp_dist(H, s=1, edges=True)`
- `comp edges dist` = `centrality_stats(s_comp_dist(H, s=1, edges=True))`
- `comp edges hist` = `Counter(s_comp_dist(H, s=1, edges=True))`
- `num comps` = `len(s_comp_dist(H))`

**Parameters**

- `H` (*hnx.Hypergraph*)
Returns

dist_stats – Dictionary which keeps track of each of the above items (e.g., basic[‘nrows’] = the number of nodes in H)

Return type
dict

reports.descriptive_stats.edge_size_dist(H, aggregated=False)
Computes edge sizes of a hypergraph.

Parameters

• H (hnx.Hypergraph)
• aggregated – If aggregated is True, returns a dictionary of edge sizes and counts. If aggregated is False, returns a list of edge sizes in H.

Returns

dist_stats – List of edge sizes or dictionary of edge size distribution.

Return type
list or dict

reports.descriptive_stats.info(H, node=None, edge=None)
Print a summary of simple statistics for H

Parameters

• H (hnx.Hypergraph)
• obj (optional) – either a node or edge uid from the hypergraph
• dictionary (optional) – If True then returns the info as a dictionary rather than a string. If False (default) returns the info as a string

Returns

info – Returns a string of statistics of the size, aspect ratio, and density of the hypergraph. Print the string to see it formatted.

Return type
string

reports.descriptive_stats.info_dict(H, node=None, edge=None)
Create a summary of simple statistics for H

Parameters

• H (hnx.Hypergraph)
• obj (optional) – either a node or edge uid from the hypergraph

Returns

info_dict – Returns a dictionary of statistics of the size, aspect ratio, and density of the hypergraph.

Return type
dict

reports.descriptive_stats.s_comp_dist(H, s=1, aggregated=False, edges=True, return_singletons=True)
Computes s-component sizes, counting nodes or edges.

Parameters

• H (hnx.Hypergraph)
• **s** (*positive integer, default is 1*)

• **aggregated** – If aggregated is True, returns a dictionary of s-component sizes and counts in \( H \). If aggregated is False, returns a list of s-component sizes in \( H \).

• **edges** – If edges is True, the component size is number of edges. If edges is False, the component size is number of nodes.

• **return_singletons** (*bool, optional, default= True*)

**Returns**

\[ s \text{\_comp\_dist} \] – List of component sizes or dictionary of component size distribution in \( H \)

**Return type**

list or dictionary

**See also:**

\[ comp\_dist \]

\[ \text{reports.descriptive_stats.} s\_edge\_diameter\_dist(H) \]

**Parameters**

\[ H (\text{hnx.Hypergraph}) \]

**Returns**

\[ s\_edge\_diameter\_dist \] – List of s-edge-diameters for hypergraph \( H \) starting with \( s=1 \) and going up as long as the hypergraph is s-edge-connected

**Return type**

list

\[ \text{reports.descriptive_stats.} s\_node\_diameter\_dist(H) \]

**Parameters**

\[ H (\text{hnx.Hypergraph}) \]

**Returns**

\[ s\_node\_diameter\_dist \] – List of s-node-diameters for hypergraph \( H \) starting with \( s=1 \) and going up as long as the hypergraph is s-node-connected

**Return type**

list

\[ \text{reports.descriptive_stats.} toplex\_dist(H, \text{aggregated}=False) \]

Computes toplex sizes for hypergraph \( H \).

**Parameters**

• **H** (*hnx.Hypergraph*)

• **aggregated** – If aggregated is True, returns a dictionary of toplex sizes and counts in \( H \). If aggregated is False, returns a list of toplex sizes in \( H \).

**Returns**

\[ toplex\_dist \] – List of toplex sizes or dictionary of toplex size distribution in \( H \)

**Return type**

list or dictionary
Module contents

reports.centrvity_stats(X)

Computes basic centrality statistics for X

Parameters

X – an iterable of numbers

Returns

[min, max, mean, median, standard deviation] – List of centrality statistics for X

Return type

list

reports.comp_dist(H, aggregated=False)

Computes component sizes, number of nodes.

Parameters

• H (hnx.Hypergraph)

• aggregated – If aggregated is True, returns a dictionary of component sizes (number of nodes) and counts. If aggregated is False, returns a list of components sizes in H.

Returns

comp_dist – List of component sizes or dictionary of component size distribution

Return type

list or dictionary

See also:

s_comp_dist

reports.degree_dist(H, aggregated=False)

Computes degrees of nodes of a hypergraph.

Parameters

• H (hnx.Hypergraph)

• aggregated – If aggregated is True, returns a dictionary of degrees and counts. If aggregated is False, returns a list of degrees in H.

Returns

degree_dist – List of degrees or dictionary of degree distribution

Return type

list or dict

reports.dist_stats(H)

Computes many basic hypergraph stats and puts them all into a single dictionary object

• nrows = number of nodes (rows in the incidence matrix)

• ncols = number of edges (columns in the incidence matrix)

• aspect ratio = nrows/ncols

• ncells = number of filled cells in incidence matrix

• density = ncells/(nrows*ncols)

• node degree list = degree_dist(H)
HyperNetX Documentation, Release 2.3.5

- node degree dist = centrality_stats(degree_dist(H))
- node degree hist = Counter(degree_dist(H))
- max node degree = max(degree_dist(H))
- edge size list = edge_size_dist(H)
- edge size dist = centrality_stats(edge_size_dist(H))
- edge size hist = Counter(edge_size_dist(H))
- max edge size = max(edge_size_dist(H))
- comp nodes list = s_comp_dist(H, s=1, edges=False)
- comp nodes dist = centrality_stats(s_comp_dist(H, s=1, edges=False))
- comp nodes hist = Counter(s_comp_dist(H, s=1, edges=False))
- comp edges list = s_comp_dist(H, s=1, edges=True)
- comp edges dist = centrality_stats(s_comp_dist(H, s=1, edges=True))
- comp edges hist = Counter(s_comp_dist(H, s=1, edges=True))
- num comps = len(s_comp_dist(H))

Parameters

- H (hnx.Hypergraph)

Returns

- dist_stats – Dictionary which keeps track of each of the above items (e.g., basic['nrows'] = the number of nodes in H)

Return type
dict

reports.edge_size_dist(H, aggregated=False)
Computes edge sizes of a hypergraph.

Parameters

- H (hnx.Hypergraph)
- aggregated – If aggregated is True, returns a dictionary of edge sizes and counts. If aggregated is False, returns a list of edge sizes in H.

Returns

- edge_size_dist – List of edge sizes or dictionary of edge size distribution.

Return type
list or dict

reports.info(H, node=None, edge=None)
Print a summary of simple statistics for H

Parameters

- H (hnx.Hypergraph)
- obj (optional) – either a node or edge uid from the hypergraph
- dictionary (optional) – If True then returns the info as a dictionary rather than a string
  If False (default) returns the info as a string
Returns
info – Returns a string of statistics of the size, aspect ratio, and density of the hypergraph. Print the string to see it formatted.

Return type
string

reports.info_dict(H, node=None, edge=None)
Create a summary of simple statistics for H

Parameters
- H (hnx.Hypergraph)
- obj (optional) – either a node or edge uid from the hypergraph

Returns
info_dict – Returns a dictionary of statistics of the size, aspect ratio, and density of the hypergraph.

Return type
dict

reports.s_comp_dist(H, s=1, aggregated=False, edges=True, return_singletons=True)
Computes s-component sizes, counting nodes or edges.

Parameters
- H (hnx.Hypergraph)
- s (positive integer, default is 1)
- aggregated – If aggregated is True, returns a dictionary of s-component sizes and counts in H. If aggregated is False, returns a list of s-component sizes in H.
- edges – If edges is True, the component size is number of edges. If edges is False, the component size is number of nodes.
- return_singletons (bool, optional, default=True)

Returns
s_comp_dist – List of component sizes or dictionary of component size distribution in H

Return type
list or dictionary

See also:
comp_dist

reports.s_edge_diameter_dist(H)

Parameters
H (hnx.Hypergraph)

Returns
s_edge_diameter_dist – List of s-edge-diameters for hypergraph H starting with s=1 and going up as long as the hypergraph is s-edge-connected

Return type
list
HyperNetX Documentation, Release 2.3.5

reports.s_node_diameter_dist(H)

Parameters
H (hnx.Hypergraph)

Returns
s_node_diameter_dist – List of s-node-diameters for hypergraph H starting with s=1 and going up as long as the hypergraph is s-node-connected

Return type
list

reports.toplex_dist(H, aggregated=False)

Computes toplex sizes for hypergraph H.

Parameters
• H (hnx.Hypergraph)
• aggregated – If aggregated is True, returns a dictionary of toplex sizes and counts in H. If aggregated is False, returns a list of toplex sizes in H.

Returns
toplex_dist – List of toplex sizes or dictionary of toplex size distribution in H

Return type
list or dictionary

5.6 A Gentle Introduction to Hypergraph Mathematics

Here we gently introduce some of the basic concepts in hypergraph modeling. We note that in order to maintain this "gentleness", we will be mostly avoiding the very important and legitimate issues in the proper mathematical foundations of hypergraphs and closely related structures, which can be very complicated. Rather we will be focusing on only the most common cases used in most real modeling, and call a graph or hypergraph gentle when they are loopless, simple, finite, connected, and lacking empty hyperedges, isolated vertices, labels, weights, or attributes. Additionally, the deep connections between hypergraphs and other critical mathematical objects like partial orders, finite topologies, and topological complexes will also be treated elsewhere. When it comes up, below we will sometimes refer to the added complexities which would attend if we weren’t being so “gentle”. In general the reader is referred to [1,2] for a less gentle and more comprehensive treatment.

5.6.1 Graphs and Hypergraphs

Network science is based on the concept of a graph $G = (V, E)$ as a system of connections between entities. $V$ is a (typically finite) set of elements, nodes, or objects, which we formally call “vertices”, and $E$ is a set of pairs of vertices. Given that, then for two vertices $u, v \in V$, an edge is a set $e = \{u, v\}$ in $E$, indicating that there is a connection between $u$ and $v$. It is then common to represent $G$ as either a Boolean adjacency matrix $A_{n \times n}$ where $n = |V|$, where an $i, j$ entry in $A$ is 1 if $v_i, v_j$ are connected in $G$; or as an incidence matrix $I_{n \times m}$, where now also $m = |E|$, and an $i, j$ entry in $I$ is now 1 if the vertex $v_i$ is in edge $e_j$. 
Notice that in the incidence matrix $I$ of a gentle graph $G$, it is necessarily the case that every column must have precisely two 1 entries, reflecting that every edge connects exactly two vertices. The move to a hypergraph $H = \langle V, E \rangle$ relaxes this requirement, in that now a hyperedge (although we will still say edge when clear from context) $e \in E$ is a subset $e = \{v_1, v_2, \ldots, v_k\} \subseteq V$ of vertices of arbitrary size. We call $e$ a $k$-edge when $|e| = k$. Note that thereby a 2-edge is a graph edge, while both a singleton $e = \{v\}$ and a 3-edge $e = \{v_1, v_2, v_3\}$, 4-edge $e = \{v_1, v_2, v_3, v_4\}$, etc., are all hypergraph edges. In this way, if every edge in a hypergraph $H$ happens to be a 2-edge, then $H$ is a graph. We call such a hypergraph 2-uniform.

Our incidence matrix $I$ is now very much like that for a graph, but the requirement that each column have exactly two 1 entries is relaxed: the column for edge $e$ with size $k$ will have $k$ 1’s. Thus $I$ is now a general Boolean matrix (although with some restrictions when $H$ is gentle).
Notice also that in the examples we’re showing in the figures, the graph is closely related to the hypergraph. In fact, this particular graph is the 2-section or underlying graph of the hypergraph. It is the graph \( G \) recorded when only the pairwise connections in the hypergraph \( H \) are recognized. Note that while the 2-section is always determined by the hypergraph, and is frequently used as a simplified representation, it almost never has enough information to be able to recover the hypergraph from it.

5.6.2 Important Things About Hypergraphs

While all graphs \( G \) are (2-uniform) hypergraphs \( H \), since they’re very special cases, general hypergraphs have some important properties which really stand out in distinction, especially to those already conversant with graphs. The following issues are critical for hypergraphs, but “disappear” when considering the special case of 2-uniform hypergraphs which are graphs.

All Hypergraphs Come in Dual Pairs

If our incidence matrix \( I \) is a general \( n \times m \) Boolean matrix, then its transpose \( I^T \) is an \( m \times n \) Boolean matrix. In fact, \( I^T \) is also the incidence matrix of a different hypergraph called the dual hypergraph \( H^* \) of \( H \). In the dual \( H^* \), it’s just that vertices and edges are swapped: we now have \( H^* = (E, V) \) where it’s \( E \) that is a set of vertices, and the now edges \( v \in V, v \subseteq E \) are subsets of those vertices.

Just like the “primal” hypergraph \( H \) has a 2-section, so does the dual. This is called the line graph, and it is an important structure which records all of the incident hyperedges. Line graphs are also used extensively in graph theory.

Note that it follows that since every graph \( G \) is a (2-uniform) hypergraph \( H \), so therefore we can form the dual hypergraph \( G^* \) of \( G \). If a graph \( G \) is a 2-uniform hypergraph, is its dual \( G^* \) also a 2-uniform hypergraph? In general, no, only in the case where \( G \) is a single cycle or a union of cycles would that be true. Also note that in order to calculate the line graph of a graph \( G \), one needs to work through its dual hypergraph \( G^* \).
**Edge Intersections Have Size**

As we’ve already seen, in a graph all the edges are size 2, whereas in a hypergraph edges can be arbitrary size 1, 2, . . . , n. Our example shows a singleton, three “graph edge” pairs, and a 2-edge.

In a gentle graph G consider two edges \( e = \{u, v\}, f = \{w, z\} \in E \) and their intersection \( g = e \cap f \). If \( g \neq \emptyset \) then \( e \) and \( f \) are non-disjoint, and we call them incident. Let \( s(e, f) = |g| \) be the size of that intersection. If \( G \) is gentle and \( e \) and \( f \) are incident, then \( s(e, f) = 1 \), in that one of \( u, v \) must be equal to one of \( w, z \), and \( g \) will be that singleton. But in a hypergraph, the intersection \( g = e \cap f \) of two incident edges can be any size \( s(e, f) \in [1, \min(|e|, |f|)] \). This aspect, the size of the intersection of two incident edges, is critical to understanding hypergraph structure and properties.

**Edges Can Be Nested**

While in a gentle graph \( G \) two edges \( e \) and \( f \) can be incident or not, in a hypergraph \( H \) there’s another case: two edges \( e \) and \( f \) may be nested or included, in that \( e \subseteq f \) or \( f \subseteq e \). That’s exactly the condition above where \( s(e, f) = \min(|e|, |f|) \), which is the size of the edge included within the including edge. In our example, we have that edge 1 is included in edge 2 is included in edge 3.

**Walks Have Length and Width**

A walk is a sequence \( W = \langle e_0, e_1, \ldots, e_N \rangle \) of edges where each pair \( e_i, e_{i+1} \), \( 0 \leq i \leq N - 1 \) in the sequence are incident. We call \( N \) the length of the walk. Walks are the raison d'être of both graphs and hypergraphs, in that in a graph \( G \) a walk \( W \) establishes the connectivity of all the \( e_i \) to each other, and a way to “travel” between the ends \( e_0 \) and \( e_N \). Naturally in a walk for each such pair we can also measure the size of the intersection \( s_i = s(e_i, e_{i+1}), 0 \leq i \leq N \). While in a gentle graph \( G \), all the \( s_i = 1 \), as we’ve seen in a hypergraph \( H \) all these \( s_i \) can vary widely. So for any walk \( W \) we can not only talk about its length \( N \), but also define its width \( s(W) = \min_{0 \leq i \leq N} s_i \) as the size of the smallest such intersection. When a walk \( W \) has width \( s \), we call it an \( s \)-walk. It follows that all walks in a graph are
1-walks with width 1. In Fig. 5.6 we see two walks in a hypergraph. While both have length 2 (counting edgewise, and recalling origin zero), the one on the left has width 1, and that on the right width 3.

![Fig. 5.6: Two hypergraph walks of length 2: (Left) A 1-walk. (Right) A 3-walk.](image)

### 5.6.3 Towards Less Gentle Things

We close with just brief mentions of more advanced issues.

#### s-Walks and Hypernetwork Science

Network science has become a dominant force in data analytics in recent years, including a range of methods measuring distance, connectivity, reachability, centrality, modularity, and related things. Most all of these concepts generalize to hypergraphs using “s-versions” of them. For example, the s-distance between two vertices or hyperedges is the length of the shortest s-walk between them, so that as s goes up, requiring wider connections, the distance will also tend to grow, so that ultimately perhaps vertices may not be s-reachable at all. See [2] for more details.

#### Hypergraphs in Mathematics

Hypergraphs are very general objects mathematically, and are deeply connected to a range of other essential objects and structures mostly in discrete science.

Most obviously, perhaps, is that there is a one-to-one relationship between a hypergraph $H = \langle V, E \rangle$ and a corresponding bipartite graph $B = \langle V \cup E, I \rangle$. $B$ is a new graph (not a hypergraph) with vertices being both the vertices and the hyperedges from the hypergraph $H$, and a connection being a pair $\{v, e\} \in I$ if and only if $v \in e$ in $H$. That you can go the other way to define a hypergraph $H$ for every bipartite graph $G$ is evident, but not all operations carry over unambiguously between hypergraphs and their bipartite versions.

Even more generally, the Boolean incidence matrix $I$ of a hypergraph $H$ can be taken as the characteristic matrix of a binary relation. When $H$ is gentle this is somewhat restricted, but in general we can see that there are one-to-one relations now between hypergraphs, binary relations, as well as bipartite graphs from above.

Additionally, we know that every hypergraph implies a hierarchical structure via the fact that for every pair of incident hyperedges either one is included in the other, or their intersection is included in both. This creates a partial order, establishing a further one-to-one mapping to a variety of lattice structures and dual lattice structures relating how groups of vertices are included in groups of edges, and vice versa. Fig. 5.8 shows the concept lattice [3], perhaps the most important of these structures, determined by our example.
Fig. 5.7: Bipartite graph.

Fig. 5.8: The concept lattice of the example hypergraph $H$. 
Finally, the strength of hypergraphs is their ability to model multi-way interactions. Similarly, mathematical topology is concerned with how multi-dimensional objects can be attached to each other, not only in continuous spaces but also with discrete objects. In fact, a finite topological space is a special kind of gentle hypergraph closed under both union and intersection, and there are deep connections between these structures and the lattices referred to above.

In this context also an abstract simplicial complex (ASC) is a kind of hypergraph where all possible included edges are present. Each hypergraph determines such an ASC by “closing it down” by subset. ASCs have a natural topological structure which can reveal hidden structures measurable by homology, and are used extensively as the workhorse of topological methods such as persistent homology. In this way hypergraphs form a perfect bridge from network science to computational topology in general.

![Diagram of ASC](image)

**Fig. 5.9:** A diagram of the ASC implied by our example. Numbers here indicate the actual hyper-edges in the original hypergraph $H$, where now additionally all sub-edges, including singletons, are in the ASC.

### Non-Gentle Graphs and Hypergraphs

Above we described our use of “gentle” graphs and hypergraphs as finite, loopless, simple, connected, and lacking empty hyperedges, isolated vertices, labels, weights, or attributes. But at a higher level of generality we can also have:

- **Empty Hyperedges:**
  - If a column of $I$ has all zero entries.

- **Isolated Vertices:**
  - If a row of $I$ has all zero entries.

- **Multihypergraphs:**
  - We may choose to allow duplicated hyperedges, resulting in duplicate columns in the incidence matrix $I$.

- **Self-Loops:**
  - In a graph allowing an edge to connect to itself.

- **Direction:**
  - In an edge, where some vertices are recognized as “inputs” which point to others recognized as “outputs”.

- **Order:**
  - In a hyperedge, where the vertices carry a particular (total) order. In a graph, this is equivalent to being directed, but not in a hypergraph.

- **Attributes:**
  - In general we use graphs and hypergraphs to model data, and thus carrying attributes of different types, including weights, labels, identifiers, types, strings, or really in principle any data object. These attributes could be on
vertices (rows of \( I \)), edges (columns of \( I \)) or what we call “incidences”, related to a particular appearance of a particular vertex in a particular edge (cells of \( I \)).


5.7 Hypergraph Constructors

An hnx.Hypergraph \( H = (V,E) \) references a pair of disjoint sets: \( V \) = nodes (vertices) and \( E \) = (hyper)edges.

HNX allows for multi-edges by distinguishing edges by their unique identifiers instead of their contents. For example, if \( V = \{1,2,3\} \) and \( E = \{e_1,e_2,e_3\} \), where \( e_1 = \{1,2\} \), \( e_2 = \{1,2\} \), and \( e_3 = \{1,2,3\} \), the edges \( e_1 \) and \( e_2 \) contain the same set of nodes and yet are distinct and are distinguishable within \( H = (V,E) \).

HNX provides methods to easily store and access additional metadata such as cell, edge, and node weights. Metadata associated with (edge,node) incidences are referenced as `cell_properties`. Metadata associated with a single edge or node is referenced as its `properties`.

The fundamental object needed to create a hypergraph is a `setsystem`. The setsystem defines the many-to-many relationships between edges and nodes in the hypergraph. Cell properties for the incidence pairs can be defined within the setsystem or in a separate pandas.DataFrame or dict. Edge and node properties are defined with a pandas.DataFrame or dict.

A hypergraph is defined by its relationships. While the nodes and edges are distinct objects with their own properties, it is only when they are in a relationship (i.e. incidence pair) that nodes and edges are viewable within the hypergraph structure. Consequently, hypergraph metrics and combinatorics do not use “isolated” nodes or “empty” edges. For example, while `node_properties` could contain any number of node identifiers, only nodes belonging to an edge in the hypergraph are counted when computing the size and shape of the hypergraph.

5.7.1 SetSystems

There are five types of setsystems currently accepted by the library.

1. **iterable of iterables**: Barebones hypergraph, which uses Pandas default indexing to generate hyperedge ids. Elements must be hashable.:

```python
>>> list_of_lists = [['book','candle','cat'],['book','coffee cup'],['coffee cup','radio']]
>>> H = Hypergraph(list_of_lists)
```

2. **dictionary of iterables**: The most basic way to express many-to-many relationships providing edge ids. The elements of the iterables must be hashable):

```python
>>> scenes_dictionary = {
...   0: ('FN', 'TH'),
...   1: ('TH', 'JV'),
...   2: ('BM', 'FN', 'JA'),
...     }
```

(continues on next page)
3. **dictionary of dictionaries**: allows cell properties to be assigned to a specific (edge, node) incidence. This is particularly useful when there are variable length dictionaries assigned to each pair:

```python
>>> nested_dictionary = {
    0: {'FN': {'time': 'early', 'weight': 7}, 'TH': {'time': 'late'}},
    1: {'TH': {'subject': 'war'}, 'JV': {'observed_by': 'someone'}},
    2: {'BM': {}, 'FN': {}, 'JA': {'role': 'policeman'}},
    3: {'JV': {'was_carrying': 'stick'}, 'JU': {}, 'CH': {}, 'BM': {'state':
        'intoxicated', 'color': 'pinkish'}},
    4: {'JV': {'weight': 15}, 'CH': {}, 'BR': {'state': 'worried'}, 'CN': {}, 'CC' 
        {'time': '05.13.2020'}},
    5: {'TH': {}, 'GP': {}},
    6: {'GP': {}, 'MP': {}},
    7: {'MA': {}, 'GP': {'accompanied_by': 'dog', 'weight': 15, 'was_singing':
        'Frère Jacques'}}
}
>>> H = hnx.Hypergraph(nested_dictionary)
```

4. **pandas.DataFrame** For large datasets and for datasets with cell properties it is most efficient to construct a hypergraph directly from a pandas.DataFrame. Incidence pairs are in the first two columns. Cell properties shared by all incidence pairs can be placed in a single column of the dataframe. Variable length dictionaries of cell properties particular to only some of the incidence pairs may be placed in a single column of the dataframe. Representing the data above as a dataframe `df`:

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
<th>w</th>
<th>col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>1</td>
<td>0.5</td>
<td>{'name': 'related_to'}</td>
</tr>
<tr>
<td>e1</td>
<td>2</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{&quot;name&quot;: &quot;related_to&quot;, &quot;start-date&quot;: &quot;05.13.2020&quot;}</td>
</tr>
<tr>
<td>e2</td>
<td>1</td>
<td>0.52</td>
<td>{'name': &quot;owned_by&quot;}</td>
</tr>
</tbody>
</table>

The first row of the dataframe is used to reference each column.

```python
>>> import pandas as pd
>>> d = {'col1': ['e1', 'e1', 'e2'],
    'col2': [1, 2, 1],
    'w': [0.5, 0.1, 0.52],
    'col3': [{'name': 'related_to'}, {'name': 'related_to', 'start_date': '05.13.2020'},
             {'name': 'owned_by'}]
>>> df = pd.DataFrame(d)
>>> H = hnx.Hypergraph(df, edge_col="col1", node_col="col2",
                     cell_weight_col="w", misc_cell_properties_col="col3")
```

5. **numpy.ndarray** For homogeneous datasets given in a n x 2 ndarray a pandas dataframe is generated. In this
case, the constructor will only accept properties for the edges and nodes using the edge and node uids listed in
the array, although incidence properties can be added after construction:

```python
>>> import numpy as np
>>> np_array = np.array([[['A','a'], ['A','b'], ['A','c'], ['B','a'], ['B','d'], ['C','c
→'], ['C','d']]])
>>> H = hnx.Hypergraph(np_array)
>>> H.includes[('A','a')].color = 'red'
>>> H.dataframe
```

### 5.7.2 Edge and Node Properties

Properties specific to edges and/or node can be passed through the keywords: `edge_properties`, `node_properties`, `properties`. Properties may be passed as dataframes or dicts. The first column or index of the dataframe or keys of the
dict keys correspond to the edge and/or node identifiers. If properties are specific to a unique id, they may be stored in
a single object and passed to the `properties` keyword. For example:

<table>
<thead>
<tr>
<th>uid</th>
<th>weight</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>5.0</td>
<td>{'type': 'event'}</td>
</tr>
<tr>
<td>e2</td>
<td>0.52</td>
<td>{'name': 'owned_by'}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>[...]</td>
</tr>
<tr>
<td>1</td>
<td>1.2</td>
<td>{'color': 'red'}</td>
</tr>
<tr>
<td>2</td>
<td>0.003</td>
<td>{'name': 'Fido', 'color': 'brown'}</td>
</tr>
<tr>
<td>3</td>
<td>1.0</td>
<td>{}</td>
</tr>
</tbody>
</table>

A properties dictionary should have the format:

```python
dp = {uid1 : {prop1:val1, prop2,val2,...}, uid2 : ... }
```

### 5.7.3 Weights

The default key for cell and object weights is “weight”. The default value is 1. Weights may be assigned from a
column in the dataframe by specifying the column and/or a new default in the constructor using `cell_weight_col` and `default_cell_weight` for incidence pairs, and using `edge_weight_prop_col`, `default_edge_weight` for edges, `node_weight_prop_col`, `default_node_weight` for nodes, and `weight_prop_col`, `default_weight` for a shared property
dataframe.
5.8 Hypernetx-Widget

5.8.1 Overview

The HyperNetXWidget is an addon for HNX, which extends the built-in visualization capabilities of HNX to a JavaScript based interactive visualization. The tool has two main interfaces, the hypergraph visualization and the nodes & edges panel. You may demo the widget here.

The HypernetxWidget is open source and available on GitHub It is also published on PyPI

The HyperNetX widget is currently in beta with limitations on the Jupyter environment in which it may be used. It is being actively worked on. Look for improvements and an expanded list of usable environments in a future release.

5.8.2 Installation

HyperNetXWidget is currently in beta and will only work on Jupyter Notebook 6.5.x. It is not supported on Jupyter Lab, but support for Jupyter Lab is in planning.

In addition, HyperNetXWidget must be installed using the Anaconda platform so that the widget can render on Jupyter notebook.

For users with inexperience with Jupyter and Anaconda, it is highly recommended to use the base environment of Anaconda so that the widget works seamlessly and out-of-the box on Jupyter Notebook. The widget does not work on Jupyter Lab.

If users want to create a custom environment instead of using the base environment provided by Anaconda, then users will need to do additional configuration on Jupyter and the kernel to ensure that the widget works. Specifically, users will need to set the Kernel to use a custom environment. For a guide on how to do this, please read and follow this guide: How to add your Conda environment to your jupyter notebook in just 4 steps.

It is highly recommended to use Anaconda to setup your virtual environment. Using python’s built-in venv module or virtualenv to create your virtual environment may result in the widget will not rendering on Jupyter notebook.
Prerequisites

- conda 23.11.x
- python 3.11.x
- jupyter notebook 6.5.4
- ipywidgets 7.6.5

Installation Steps

Open a new shell and run the following commands:

```bash
# update conda
conda update conda

# activate the base environment
conda activate

# install hypernetx and hnxwidget
pip install hypernetx hnxwidget

# install jupyter notebook and extensions
conda install -y -c anaconda notebook
conda install -y -c conda-forge jupyter_contrib_nbextensions

# install and enable the hnxwidget on jupyter
jupyter nbextension install --py --symlink --sys-prefix hnxwidget
jupyter nbextension enable --py --sys-prefix hnxwidget

# install ipykernel and use it to add the base environment to jupyter notebook
conda install -y -c anaconda ipykernel
python -m ipykernel install --user --name=base

# start the notebook
jupyter-notebook
```

Conda Environment

If the notebook runs into a *ModuleNotFoundError* for the HyperNetX or HyperNetXWidget packages, ensure that you set your kernel to the conda base environment (i.e. *base*). This will ensure that your notebook has the right environment to run the widget.

On the notebook, click the “New” drop-down button and select “base” as the environment for your notebook. See the following screenshot as an example:
5.8.3 Using the Tool

Layout

The hypergraph visualization is an Euler diagram that shows nodes as circles and hyper edges as outlines containing the nodes/circles they contain. The visualization uses a force directed optimization to perform the layout. This algorithm is not perfect and sometimes gives results that the user might want to improve upon. The visualization allows the user to drag nodes and position them directly at any time. The algorithm will re-position any nodes that are not specified by the user. Ctrl (Windows) or Command (Mac) clicking a node will release a pinned node it to be re-positioned by the algorithm.

Selection

Nodes and edges can be selected by clicking them. Nodes and edges can be selected independently of each other, i.e., it is possible to select an edge without selecting the nodes it contains. Multiple nodes and edges can be selected, by holding down Shift while clicking. Shift clicking an already selected node will de-select it. Clicking the background will de-select all nodes and edges. Dragging a selected node will drag all selected nodes, keeping their relative placement. Selected nodes can be hidden (having their appearance minimized) or removed completely from the visualization. Hiding a node or edge will not cause a change in the layout, whereas removing a node or edge will. The selection can also be expanded. Buttons in the toolbar allow for selecting all nodes contained within selected edges, and selecting all edges containing any selected nodes. The toolbar also contains buttons to select all nodes (or edges), un-select all nodes (or edges), or reverse the selected nodes (or edges). An advanced user might:

- **Select all nodes not in an edge** by: select an edge, select all nodes in that edge, then reverse the selected nodes to select every node not in that edge.

- **Traverse the graph** by: selecting a start node, then alternating select all edges containing selected nodes and selecting all nodes within selected edges

- **Pin Everything** by: hitting the button to select all nodes, then drag any node slightly to activate the pinning for all nodes.

Side Panel

Details on nodes and edges are visible in the side panel. For both nodes and edges, a table shows the node name, degree (or size for edges), its selection state, removed state, and color. These properties can also be controlled directly from this panel. The color of nodes and edges can be set in bulk here as well, for example, coloring by degree.

Other Features

Nodes with identical edge membership can be collapsed into a super node, which can be helpful for larger hypergraphs. Dragging any node in a super node will drag the entire super node. This feature is available as a toggle in the nodes panel.

The hypergraph can also be visualized as a bipartite graph (similar to a traditional node-link diagram). Toggling this feature will preserve the locations of the nodes between the bipartite and the Euler diagrams.
5.9 Modularity and Clustering

5.9.1 Overview

The hypergraph_modularity submodule in HNX provides functions to compute hypergraph modularity for a given partition of the vertices in a hypergraph. In general, higher modularity indicates a better partitioning of the vertices into dense communities.

Two functions to generate such hypergraph partitions are provided: Kumar’s algorithm, and the simple last-step refinement algorithm.

The submodule also provides a function to generate the two-section graph for a given hypergraph which can then be used to find vertex partitions via graph-based algorithms.

5.9.2 Installation

Since it is part of HNX, no extra installation is required. The submodule can be imported as follows:

```python
import hypernetx.algorithms.hypergraph_modularity as hmod
```

5.9.3 Using the Tool

Modularity

Given hypergraph HG and a partition A of its vertices, hypergraph modularity is a measure of the quality of this partition. Random partitions typically yield modularity near zero (it can be negative) while positive modularity is indicative of the presence of dense communities or modules. There are several variations for the definition of hypergraph modularity, and the main difference lies in the weight given to different edges given their size $d$ and purity $c$. Modularity is computed via:
\[ q = \text{hmod.modularity}(HG, A, \text{wdc}=\text{hmod.linear}) \]

where the \textit{wdc} parameter points to a function that controls the weights (details below).

In a graph, an edge only links two nodes; so given partition \( A \), an edge is either within a community (which increases the modularity) or between communities. With hypergraphs, we consider edges of size \( d = 2 \) or more. Given some vertex partition \( A \) and some \( d \)-edge \( e \), let \( c \) be the number of nodes that belong to the most represented part in \( e \); if \( c > d/2 \), we consider this edge to be within the part. Hyper-parameters \( 0 \leq w(d, c) \leq 1 \) control the weight given to such edges. Three functions are supplied in this submodule, namely:

**linear**
\[
w(d, c) = \begin{cases} 
  c/d & \text{if } c > d/2, \\
  0 & \text{else.}
\end{cases}
\]

**majority**
\[
w(d, c) = \begin{cases} 
  1 & \text{if } c > d/2, \\
  0 & \text{else.}
\end{cases}
\]

**strict**
\[
w(d, c) = \begin{cases} 
  1 & \text{iff } c = d, \\
  0 & \text{else.}
\end{cases}
\]

The ‘linear’ function is used by default. Other functions \( w(d, c) \) can be supplied as long as \( 0 \leq w(d, c) \leq 1 \) and \( w(d, c) = 0 \) when \( c \leq d \). More details can be found in [2].

**Two-section graph**

There are several good partitioning algorithms for graphs such as the Louvain algorithm, Leiden and ECG, a consensus clustering algorithm. One way to obtain a partition for hypergraph \( HG \) is to build its corresponding two-section graph \( G \) and run a graph clustering algorithm. Code is provided to build such a graph via:

\[ G = \text{hmod.two_section}(HG) \]

which returns an igraph.Graph object.

**Clustering Algorithms**

Two clustering (vertex partitioning) algorithms are supplied. The first one is a hybrid method proposed by Kumar et al. (see [1]) that uses the Louvain algorithm on the two-section graph, but re-weights the edges according to the distribution of vertices from each part inside each edge. Given hypergraph \( HG \), this is called as:

\[ K = \text{hmod.kumar}(HG) \]

The other supplied algorithm is a simple method to improve hypergraph modularity directly. Given some initial partition of the vertices (for example via Louvain on the two-section graph), we move vertices between parts in order to improve hypergraph modularity. Given hypergraph \( HG \) and an initial partition \( A \), it is called as follows:

\[ L = \text{hmod.last_step}(HG, A, \text{wdc}=\text{linear}) \]

where the ‘wdc’ parameter is the same as in the modularity function.
Other Features

We represent a vertex partition $A$ as a list of sets, but another useful representation is via a dictionary. We provide two utility functions to switch representation, namely:

$$A = \text{dict2part}(D)$$
$$D = \text{part2dict}(A)$$

References


5.10 Publications

Praggastis, Brenda; Aksoy, Sinan; Arendt, Dustin; Bonicillo, Mark; Joslyn, Cliff; Purvine, Emilie; Shapiro, Madelyn; Yun, Ji Young: (2024) “HyperNetX: A Python package for modeling complex network data as hypergraphs”, Journal of Open Source Software, 9(95), 6016, https://doi.org/10.21105/joss.06016


Feng, Song; Heath, Emily; Jefferson, Brett; Joslyn, CA; Kvinge, Henry; McDermott, Jason E; Mitchell, Hugh D; Praggastis, Brenda; Eisfeld, Amie J; Sims, Amy C; Thackray, Larissa B; Fan, Shufang; Walters, Kevin B; Halfmann, Peter J; Westhoff-Smith, Danielle; Tan, Qing; Menachery, Vineet D; Sheahan, Timothy P; Cockrell, Adam S; Kocher, Jacob F; Stratton, Kelly G; Heller, Natalie C; Bramer, Lisa M; Diamond, Michael S; Baric, Ralph S; Waters, Katrina M; Kawaoka, Yoshihiro; Purvine, Emilie: (2021) “Hypergraph Models of Biological Networks to Identify Genes Critical to Pathogenic Viral Response”, in: BMC Bioinformatics, v. 22:287, https://doi.org/10.1186/s12859-021-04197-2


Joslyn, Cliff A; Aksoy, Sinan; Arendt, Dustin; Firoz, J; Jenkins, Louis; Praggastis, Brenda; Purvine, Emilie AH; Zalewski, Marcin: (2020) “Hypergraph Analytics of Domain Name System Relationships”, in: 17th Wshop. on
5.11 License

HyperNetX

Copyright 2018, 2023, 2024, Battelle Memorial Institute

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
5.12 Indices and tables

- genindex
- modindex
- search
PYTHON MODULE INDEX

a
algorithms, 101
algorithms.contagion, 78
algorithms.generative_models, 86
algorithms.homology_mod2, 88
algorithms.hypergraph_modularity, 94
algorithms.laplacians_clustering, 97
algorithms.s_centrality_measures, 99

c
classes, 21
classes.factory, 48
classes.hyp_view, 50
classes.hypergraph, 52
classes.incidence_store, 75
classes.property_store, 76

d
drawing, 129
drawing.rubber_band, 122
drawing.two_column, 126
drawing.util, 128

r
reports, 135
reports.descriptive_stats, 131
INDEX

A

add_edge() (classes.Hypergraph method), 24
add_edge() (classes.hypergraph.Hypergraph method), 56
add_edges_from() (classes.Hypergraph method), 25
add_edges_from() (classes.hypergraph.Hypergraph method), 56
add_incidence() (classes.Hypergraph method), 25
add_incidence() (classes.hypergraph.Hypergraph method), 56
add_incidences_from() (classes.Hypergraph method), 25
add_incidences_from() (classes.hypergraph.Hypergraph method), 57
add_node() (classes.Hypergraph method), 26
add_node() (classes.hypergraph.Hypergraph method), 57
add_nodes_from() (classes.Hypergraph method), 26
add_nodes_from() (classes.hypergraph.Hypergraph method), 57
add_to_column() (in module algorithms), 103
add_to_column() (in module algorithms.homology_mod2), 88
add_to_row() (in module algorithms), 103
add_to_row() (in module algorithms.homology_mod2), 88
adjacency_matrix() (classes.Hypergraph method), 26
adjacency_matrix() (classes.hypergraph.Hypergraph method), 58

B

betti() (in module algorithms), 103
betti() (in module algorithms.homology_mod2), 89
betti_numbers() (in module algorithms), 104
betti_numbers() (in module algorithms.homology_mod2), 89
bipartite() (classes.Hypergraph method), 27
bipartite() (classes.hypergraph.Hypergraph method), 58
bkMatrix() (in module algorithms), 104
bkMatrix() (in module algorithms.homology_mod2), 89
boundary_group() (in module algorithms), 104
boundary_group() (in module algorithms.homology_mod2), 89

C

centrality_stats() (in module reports), 135
centrality_stats() (in module reports.descriptive_stats), 131
chain_complex() (in module algorithms), 104
chain_complex() (in module algorithms.homology_mod2), 90
chung_lu_hypergraph() (in module algorithms), 105
chung_lu_hypergraph() (in module algorithms.homology_mod2), 86
classes
module, 21
classes.factory
module, 48
classes.hyp_view
module, 50
class.Hypergraph method), 27
collapse_edges() (classes.Hypergraph method), 27
collapse_edges() (classes.hypergraph.Hypergraph method), 59
collapse_identical_elements() (classes.incidence_store.IncidenceStore method), 75
collapse_identical_elements() (classes.IncidenceStore method), 45
collapse_nodes() (classes.Hypergraph method), 28
collapse_nodes() (classes.hypergraph.Hypergraph method), 59
collapse_nodes_and_edges() (classes.Hypergraph method), 29
collapse_nodes_and_edges() (classes.hypergraph.Hypergraph method), 60
collective_contagion() in module algorithms, 105
collective_contagion() (in module algorithms.contagion), 80
comp_dist() (in module reports), 135
comp_dist() (in module reports.descriptive_stats), 131
comp_dist() (in module reports.contagion), 135
comp_dist() (in module reports.descriptive_stats), 132
diameter() (classes.Hypergraph method), 31
diameter() (classes.hypergraph.Hypergraph method), 63
dict2part() (in module algorithms), 108
dict2part() (in module algorithms.hypergraph_modularity), 94
dict_factory_method() (in module classes.factory), 49
dict_to_incidence_store_df() (in module classes.factory), 49
diff() (classes.Hypergraph method), 32
diff() (classes.hypergraph.Hypergraph method), 63
dim() (classes.Hypergraph method), 32
dim() (classes.hypergraph.Hypergraph method), 63
dimensions (classes.incidence_store.IncidenceStore property), 75
dimensions (classes.IncidenceStore property), 45
discrete_SIR() (in module algorithms), 108
discrete_SIR() (in module algorithms.contagion), 82
discrete_SIS() (in module algorithms), 109
discrete_SIS() (in module algorithms.contagion), 83
dist_stats() (in module reports), 135
dist_stats() (in module reports.descriptive_stats), 132
distance() (classes.Hypergraph method), 32
distance() (classes.hypergraph.Hypergraph method), 63
draw() (in module drawing), 129
draw() (in module drawing.rubber_band), 122
draw() (in module drawing.two_column), 126
draw_hyper_edge_labels() (in module drawing.rubber_band), 124
draw_hyper_edges() (in module drawing.rubber_band), 124
draw_hyper_edges() (in module drawing.two_column), 127
draw_hyper_labels() (in module drawing.rubber_band), 124
draw_hyper_labels() (in module drawing.two_column), 127
draw_hyper_nodes() (in module drawing.rubber_band), 125
draw_two_column() (in module drawing), 130
drawing module, 129
drawing.rubber_band module, 122
drawing.two_column module, 126
drawing.util module, 128
dual, 16
dual() (classes.Hypergraph method), 32
dual() (classes.hypergraph.Hypergraph method), 64
edge_adjacency_matrix() (classes.Hypergraph method), 33
equivalence_classes() (classes.Hypergraph method), 35
equivalence_classes() (classes.hypergraph.Hypergraph method), 66
equivalence_classes() (classes.incidence_store.IncidenceStore method), 75
equivalence_classes() (classes.IncidenceStore property), 45
equivalence_classes() (classes.IncidenceStore property), 46
erdos_renyi_hypergraph() (in module algorithms), 110
erdos_renyi_hypergraph() (in module algorithms.generative_models), 87
f
flatten() (in module classes.property_store), 78
from_bipartite() (classes.Hypergraph class method), 35
from_bipartite() (classes.hypergraph.Hypergraph class method), 66
from_incidence_dataframe() (classes.Hypergraph class method), 35
from_incidence_dataframe() (classes.hypergraph.Hypergraph class method), 67
get_cell_properties() (classes.Hypergraph method), 64
Index 161
get_cell_properties() (classes.hypergraph.Hypergraph method), 68
getCollapsedSize() (in module drawing.util), 128
get_default_radius() (in module drawing.rubber_band), 125
get_frozenset_label() (in module drawing.util), 128
get_line_graph() (in module drawing.util), 128
get_linegraph() (classes.Hypergraph method), 37
get_pi() (classes.hypergraph.Hypergraph method), 68
get_properties() (classes.Hypergraph method), 68
get_properties() (classes.Hypergraph method), 68
get_properties() (classes.property_store.PropertyStore method), 76
get_properties() (classes.PropertyStore method), 47
get_property() (classes.property_store.PropertyStore method), 77
get_set_layering() (classes.PropertyStore method), 77
Gillespie_SIRC() (in module algorithms), 101
Gillespie_SIRC() (in module algorithms.contagion), 78
Gillespie_SIS() (in module algorithms), 102
Gillespie_SIS() (in module algorithms.contagion), 79

homology_basis() (in module algorithms), 111
homology_basis() (in module algorithms.homology_mod2), 90
hyperedges, 17
hypergraph, 17
Hypergraph (class in classes), 21
Hypergraph (class in classes.hypergraph), 52
hypergraph_homology_basis() (in module algorithms), 111
hypergraph_homology_basis() (in module algorithms.homology_mod2), 90
HypergraphView, 17
HypergraphView (class in classes), 43
HypergraphView (class in classes.hyp_view), 50
hypernodes, 18

incidence_matrix, 17
incidence_dataframe() (classes.Hypergraph method), 37
incidence_dataframe() (classes.hypergraph.Hypergraph method), 69

kchainbasis() (in module algorithms), 112
kchainbasis() (in module algorithms.homology_mod2), 91
kumar() (in module algorithms), 113
kumar() (in module algorithms.hypergraph_modularity), 94
<table>
<thead>
<tr>
<th>Function/Property</th>
<th>Module/Class/Property</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>last_step()</td>
<td>algorithms</td>
<td>113</td>
</tr>
<tr>
<td>last_step()</td>
<td>algorithms.hypergraph_modularity</td>
<td>94</td>
</tr>
<tr>
<td>layout_hyper_edges()</td>
<td>drawing.rubber_band</td>
<td>125</td>
</tr>
<tr>
<td>layout_node_link()</td>
<td>drawing.rubber_band</td>
<td>126</td>
</tr>
<tr>
<td>layout_two_column()</td>
<td>drawing.two_column</td>
<td>127</td>
</tr>
<tr>
<td>level</td>
<td>classes.hyp_view.HypergraphView property</td>
<td>51</td>
</tr>
<tr>
<td>level</td>
<td>classes.HypergraphView property</td>
<td>44</td>
</tr>
<tr>
<td>linear()</td>
<td>algorithms</td>
<td>113</td>
</tr>
<tr>
<td>linear()</td>
<td>algorithms.hypergraph_modularity</td>
<td>95</td>
</tr>
<tr>
<td>list_factory_method()</td>
<td>classes.factory</td>
<td>49</td>
</tr>
<tr>
<td>logical_dot()</td>
<td>algorithms</td>
<td>114</td>
</tr>
<tr>
<td>logical_dot()</td>
<td>algorithms.homology_mod2</td>
<td>91</td>
</tr>
<tr>
<td>logical_matadd()</td>
<td>algorithms</td>
<td>114</td>
</tr>
<tr>
<td>logical_matadd()</td>
<td>algorithms.homology_mod2</td>
<td>92</td>
</tr>
<tr>
<td>logical_matmul()</td>
<td>algorithms</td>
<td>114</td>
</tr>
<tr>
<td>logical_matmul()</td>
<td>algorithms.homology_mod2</td>
<td>92</td>
</tr>
<tr>
<td>majority()</td>
<td>algorithms</td>
<td>115</td>
</tr>
<tr>
<td>majority()</td>
<td>algorithms.hypergraph_modularity</td>
<td>95</td>
</tr>
<tr>
<td>majority()</td>
<td>algorithms.contagion</td>
<td>85</td>
</tr>
<tr>
<td>matmulreduce()</td>
<td>algorithms</td>
<td>115</td>
</tr>
<tr>
<td>matmulreduce()</td>
<td>algorithms.homology_mod2</td>
<td>92</td>
</tr>
<tr>
<td>memberships</td>
<td>classes.hyp_view.HypergraphView property</td>
<td>51</td>
</tr>
<tr>
<td>memberships</td>
<td>classes.HypergraphView property</td>
<td>44</td>
</tr>
<tr>
<td>memberships</td>
<td>classes.incidence_store.IncidenceStore property</td>
<td>75</td>
</tr>
<tr>
<td>mdict()</td>
<td>classes.IncidenceStore property</td>
<td>46</td>
</tr>
<tr>
<td>modularity()</td>
<td>algorithms</td>
<td>116</td>
</tr>
<tr>
<td>modularity()</td>
<td>algorithms.hypergraph_modularity</td>
<td>96</td>
</tr>
<tr>
<td>module</td>
<td>algorithms</td>
<td>101</td>
</tr>
<tr>
<td>module</td>
<td>algorithms.contagion</td>
<td>78</td>
</tr>
<tr>
<td>module</td>
<td>algorithms.generative_models</td>
<td>86</td>
</tr>
<tr>
<td>module</td>
<td>algorithms.homology_mod2</td>
<td>88</td>
</tr>
<tr>
<td>module</td>
<td>algorithms.hypergraph_modularity</td>
<td>94</td>
</tr>
<tr>
<td>module</td>
<td>algorithms.laplacians_clustering</td>
<td>97</td>
</tr>
<tr>
<td>algorithms.s_centrality_measures</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>classes</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>classes.factory</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>classes.hyp_view</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>classes.hypergraph</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>classes.incidence_store</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>classes.property_store</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>drawing</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>drawing.rubber_band</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>drawing.two_column</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>drawing.util</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>reports</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>reports.descriptive_stats</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>multihypergraph</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>ndarray_factory_method()</td>
<td>classes.factory</td>
<td>50</td>
</tr>
<tr>
<td>neighbors()</td>
<td>classes.Hypergraph method</td>
<td>39</td>
</tr>
<tr>
<td>neighbors()</td>
<td>classes.hypergraph.Hypergraph method</td>
<td>70</td>
</tr>
<tr>
<td>neighbors()</td>
<td>classes.incidence_store.IncidenceStore method</td>
<td>75</td>
</tr>
<tr>
<td>neighbors()</td>
<td>classes.IncidenceStore method</td>
<td>46</td>
</tr>
<tr>
<td>node_diameters()</td>
<td>classes.Hypergraph method</td>
<td>39</td>
</tr>
<tr>
<td>node_diameters()</td>
<td>classes.hypergraph.Hypergraph method</td>
<td>70</td>
</tr>
<tr>
<td>nodes</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>nodes</td>
<td>classes.Hypergraph method</td>
<td>39</td>
</tr>
<tr>
<td>nodes</td>
<td>classes.hypergraph.Hypergraph property</td>
<td>70</td>
</tr>
<tr>
<td>nodes</td>
<td>classes.incidence_store.IncidenceStore property</td>
<td>75</td>
</tr>
<tr>
<td>nodes</td>
<td>classes.IncidenceStore property</td>
<td>46</td>
</tr>
<tr>
<td>norm_lap()</td>
<td>algorithms</td>
<td>116</td>
</tr>
<tr>
<td>norm_lap()</td>
<td>algorithms.laplacians_clustering</td>
<td>97</td>
</tr>
<tr>
<td>order()</td>
<td>classes.Hypergraph method</td>
<td>39</td>
</tr>
<tr>
<td>order()</td>
<td>classes.hypergraph.Hypergraph method</td>
<td>70</td>
</tr>
<tr>
<td>part2dict()</td>
<td>algorithms</td>
<td>117</td>
</tr>
<tr>
<td>part2dict()</td>
<td>algorithms.hypergraph_modularity</td>
<td>96</td>
</tr>
<tr>
<td>prob_trans()</td>
<td>algorithms</td>
<td>117</td>
</tr>
<tr>
<td>prob_trans()</td>
<td>algorithms.laplacians_clustering</td>
<td>98</td>
</tr>
<tr>
<td>properties</td>
<td>classes.hyp_view.HypergraphView property</td>
<td>51</td>
</tr>
<tr>
<td>properties</td>
<td>classes.Hypergraph property</td>
<td>39</td>
</tr>
<tr>
<td>properties</td>
<td>classes.hypergraph.Hypergraph property</td>
<td>71</td>
</tr>
<tr>
<td>properties</td>
<td>classes.HypergraphView property</td>
<td>44</td>
</tr>
</tbody>
</table>
properties (classes.property_store.PropertyStore property), 77
properties (classes.PropertyStore property), 47
property_store (classes.hyp_view.HypergraphView property), 51
property_store (classes.HypergraphView property), 45
PropertyStore, 18
PropertyStore (class in classes), 46
PropertyStore (class in classes.property_store), 76

R
reduced_row_echelon_form_mod2() (in module algorithms), 117
reduced_row_echelon_form_mod2() (in module algorithms.homology_mod2), 92
remove_edges() (classes.Hypergraph method), 39
remove_edges() (classes.hypergraph.Hypergraph method), 71
remove_incidences() (classes.Hypergraph method), 40
remove_incidences() (classes.hypergraph.Hypergraph method), 71
remove_nodes() (classes.Hypergraph method), 40
remove_nodes() (classes.hypergraph.Hypergraph method), 71
remove_singletons() (classes.Hypergraph method), 40
remove_singletons() (classes.hypergraph.Hypergraph method), 72
rename() (classes.Hypergraph method), 40
rename() (classes.hypergraph.Hypergraph method), 72
reports
  module, 135
reports.descriptive_stats
  module, 131
restrict_to() (classes.incidence_store.IncidenceStore method), 76
restrict_to() (classes.IncidenceStore method), 46
restrict_to_edges() (classes.Hypergraph method), 41
restrict_to_edges() (classes.hypergraph.Hypergraph method), 72
restrict_to_nodes() (classes.Hypergraph method), 41
restrict_to_nodes() (classes.hypergraph.Hypergraph method), 72

S
s_betweenness_centrality() (in module algorithms), 117
s_betweenness_centrality() (in module algorithms.s_centrality_measures), 99
s_closeness_centrality() (in module algorithms), 118
s_closeness_centrality() (in module algorithms.s_centrality_measures), 99
s_comp_dist() (in module reports), 137
s_comp_dist() (in module reports.descriptive_stats), 133
s_component_subgraphs() (classes.Hypergraph method), 41
s_component_subgraphs() (classes.Hypergraph method), 72
s_components() (classes.Hypergraph method), 41
s_components() (classes.hypergraph.Hypergraph method), 73
s_connected_components() (classes.Hypergraph method), 41
s_connected_components() (classes.hypergraph.Hypergraph method), 73
s_eccentricity() (in module algorithms), 118
s_eccentricity() (in module algorithms.s_centrality_measures), 100
s_edge_diameter_dist() (in module reports), 137
s_edge_diameter_dist() (in module reports.descriptive_stats), 134
s_harmonic_centrality() (in module algorithms), 119
s_harmonic_centrality() (in module algorithms.s_centrality_measures), 100
s_harmonic_closeness_centrality() (in module algorithms), 120
s_harmonic_closeness_centrality() (in module algorithms.s_centrality_measures), 101
s_node_diameter_dist() (in module reports), 137
s_node_diameter_dist() (in module reports.descriptive_stats), 134
s-adjacency matrix, 18
s-adjacent, 18
s-auxiliary matrix, 18
s-connected, 18
s-connected component, 19
s-degree, 19
s-diameter, 19
s-distance, 19
s-edge, 19
s-edge-adjacency matrix, 19
s-edge-adjacent, 19
s-edge-auxiliary matrix, 18
s-edge-connected, 18
s-edge-connected component, 19