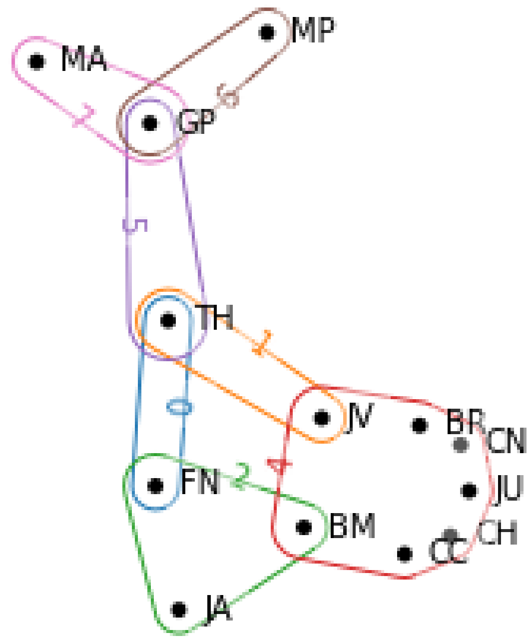
HyperNetX Documentation

Release 2.2.0

Brenda Praggastis, Dustin Arendt

March 01, 2024

CONTENTS



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.

CHAPTER

TWO

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.

CHAPTER THREE

OUR VALUES

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.

CHAPTER

FOUR

CONTACT US

Questions and comments are welcome! Contact us at

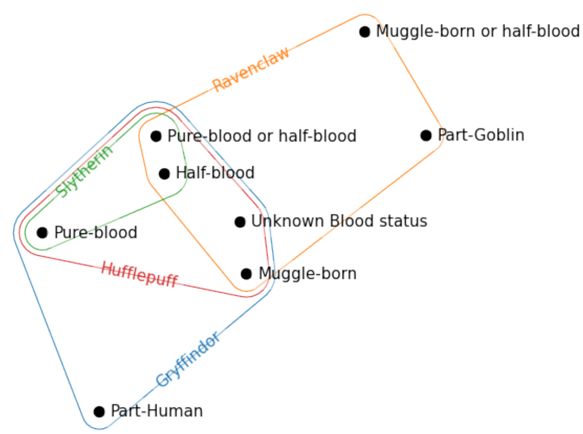
hypernetx@pnnl.gov

CHAPTER

FIVE

CONTENTS

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-ACO5-76RL01830.

- 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

For questions and comments contact the developers directly at: hypernetx@pnnl.gov

New Features in Version 2.0

HNX 2.0 now accepts metadata as core attributes of the edges and nodes of a hypergraph. While the library continues to accept lists, dictionaries and dataframes as basic inputs for hypergraph constructions, both cell properties and edge and node properties can now be easily added for retrieval as object attributes.

The core library has been rebuilt to take advantage of the flexibility and speed of Pandas Dataframes. Dataframes offer the ability to store and easily access hypergraph metadata. Metadata can be used for filtering objects, and characterize their distributions by their attributes.

Version 2.0 is not backwards compatible. Objects constructed using version 1.x can be imported from their incidence dictionaries.

What's New

1. The Hypergraph constructor now accepts nested dictionaries with incidence cell properties, pandas.DataFrames, and 2-column Numpy arrays.
2. Additional constructors accept incidence matrices and incidence dataframes.
3. Hypergraph constructors accept cell, edge, and node metadata.
4. Metadata available as attributes on the cells, edges, and nodes.
5. User-defined cell weights and default weights available to incidence matrix.
6. Meta data persists with restrictions and removals.
7. Meta data persists onto s-linegraphs as node attributes of Networkx graphs.
8. New hnxwidget available using *pip install hnxwidget*.

What's Changed

1. The *static* and *dynamic* distinctions no longer exist. All hypergraphs use the same underlying data structure, supported by Pandas dataFrames. All hypergraphs maintain a *state_dict* to avoid repeating computations.
2. Methods for adding nodes and hyperedges are currently not supported.
3. The *nwhy* optimizations are no longer supported.
4. Entity and EntitySet classes are being moved to the background. The Hypergraph constructor does not accept either.

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

5.2.1 Installation

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.

5.2.2 Prerequisites

HyperNetX officially supports Python 3.8, 3.9, 3.10 and 3.11.

5.2.3 Create a virtual environment

Using Anaconda

```
>>> conda create -n venv-hnx python=3.8 -y
>>> conda activate venv-hnx
```

Using venv

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

Using virtualenv

```
>>> 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:

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

To deactivate your environment, use:

```
>>> .\env-hnx\Scripts\deactivate
```

5.2.4 Installing Hypernetx

Regardless of how you install HyperNetX, ensure that your environment is activated and that you are running Python >=3.8.

Installing from PyPi

```
>>> pip install hypernetx
```

If you want to use supported applications built upon HyperNetX (e.g. `hypernetx.algorithms.hypergraph_modularity` or `hypernetx.algorithms.contagion`), you can install HyperNetX with those supported applications by using the following command:

```
>>> pip install hypernetx[all]
```

If you are using zsh as your shell, use single quotation marks around the square brackets:

```
>>> pip install hypernetx'[all]'
```

Installing from Source

Ensure that you have `git` installed.

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

5.2.5 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
>>> data = { 0: ('A', 'B'), 1: ('B', 'C'), 2: ('D', 'A', 'E'), 3: ('F', 'G', 'H', 'D') }
>>> H = hnx.Hypergraph(data)
>>> list(H.nodes)
['G', 'F', 'D', 'A', 'B', 'H', 'C', 'E']
>>> 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 tutorial-deps
>>> make tutorials
```

Building documentation

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

```
>>> make docs-deps
>>> cd docs
>>> make html
>>> open build/index.html
```

5.3 Glossary of HNX terms

The HNX library centers around the idea of a *hypergraph*. This glossary provides a few key terms and definitions.

degree
: Given a hypergraph (Nodes, Edges, Incidence), the degree of a node in Nodes is the number of edges in Edges to which the node is incident. See also: s-degree

dual
The dual of a hypergraph (Nodes, Edges, Incidence) switches the roles of Nodes and Edges. More precisely, it is the hypergraph (Edges, Nodes, Incidence'), where Incidence' is the function that assigns Incidence(n,e) to each pair (e,n). The *incidence matrix* of the dual hypergraph is the transpose of the incidence matrix of (Nodes, Edges, Incidence).

edge nodes (aka edge elements)
The nodes (or elements) of an edge e in a hypergraph (Nodes, Edges, Incidence) are the nodes that are incident to e.

Entity and Entity set
Class in entity.py. HNX stores many of its data structures inside objects of type Entity. Entities help to insure safe behavior, but their use is primarily technical, not mathematical.

hypergraph
The term *hypergraph* can have many different meanings. In HNX, it means a tuple (Nodes, Edges, Incidence), where Nodes and Edges are sets, and Incidence is a function that assigns a value of True or False to every pair (n,e) in the Cartesian product Nodes x Edges. We call - Nodes the set of nodes - Edges the set of edges - Incidence the incidence function *Note* Another term for this type of object is a *multihypergraph*. The ability to work with multihypergraphs efficiently is a distinguishing feature of HNX!

incidence
A node n is incident to an edge e in a hypergraph (Nodes, Edges, Incidence) if Incidence(n,e) = True. !!! – give the line of code that would allow you to evaluate

incidence matrix
A rectangular matrix constructed from a hypergraph (Nodes, Edges, Incidence) where the elements of Nodes index the matrix rows, and the elements of Edges index the matrix columns. Entry (n,e) in the incidence matrix is 1 if n and e are incident, and is 0 otherwise.

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

subhypergraph
A subhypergraph of a hypergraph (Nodes, Edges, Incidence) is a hypergraph (Nodes', Edges', Incidence') such that Nodes' is a subset of Nodes, Edges' is a subset of Edges, and every incident pair (n,e) in (Nodes', Edges', Incidence') is also incident in (Nodes, Edges, Incidence)

subhypergraph induced by a set of nodes
An induced subhypergraph of a hypergraph (Nodes, Edges, Incidence) is a subhypergraph (Nodes', Edges', Incidence') where a pair (n,e) is incident if and only if it is incident in (Nodes, Edges, Incidence)

toplex
A toplex in a hypergraph (Nodes, Edges, Incidence) is an edge e whose node set isn't properly contained in the node set of any other edge. That is, if f is another edge and ever node incident to e is also incident to f, then the node sets of e and f are identical.

5.3.1 S-line graphs

HNX offers a variety of tool sets for network analysis, including s-line graphs.

s-adjacency matrix
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, a square matrix where the elements of Nodes index both rows and columns. The matrix can be weighted or unweighted. Entry (i,j) is nonzero if and only if node i and node j are incident to at least s edges in common. If it is nonzero, then it is equal to the number of shared edges (if weighted) or 1 (if unweighted).

s-edge-adjacency matrix
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, a square matrix where the elements

of Edges index both rows and columns. The matrix can be weighted or unweighted. Entry (i,j) is nonzero if and only if edge i and edge j share to at least s nodes, and is equal to the number of shared nodes (if weighted) or 1 (if unweighted).

s-auxiliary matrix
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, the submatrix of the s-edge-adjacency matrix obtained by restricting to rows and columns corresponding to edges of size at least s.

s-node-walk
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, a sequence of nodes in Nodes such that each successive pair of nodes share at least s edges in Edges.

s-edge-walk
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, a sequence of edges in Edges such that each successive pair of edges intersects in at least s nodes in Nodes.

s-walk
Either an s-node-walk or an s-edge-walk.

s-connected component, s-node-connected component
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, an s-connected component is a *subhypergraph* induced by a subset of Nodes 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.

s-edge-connected component
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, an s-edge-connected component is a *subhypergraph* induced by a subset of Edges with the property that there exists an s-edge-walk between every pair of edges in this subset. An s-edge-connected component is the maximal such subset in the sense that it is not properly contained in any other subset satisfying this property.

s-connected, s-node-connected
A hypergraph is s-connected if it has one s-connected component.

s-edge-connected
A hypergraph is s-edge-connected if it has one s-edge-connected component.

s-distance
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, the s-distances between two nodes in Nodes is the length of the shortest s-node-walk between them. If no s-node-walks between the pair of nodes exists, the s-distance between them is infinite. The s-distance between edges is the length of the shortest s-edge-walk between them. If no s-edge-walks between the pair of edges exist, then s-distance between them is infinite.

s-diameter
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, the s-diameter is the maximum s-Distance over all pairs of nodes in Nodes.

s-degree
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, the s-degree of a node is the number of edges in Edges of size at least s to which node belongs. See also: *degree*

s-edge
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, an s-edge is any edge of size at least s.

s-linegraph
For a hypergraph (Nodes, Edges, Incidence) and positive integer s, an s-linegraph is a graph representing the node to node or edge to edge connections according to the *width* s of the connections. The

node s-linegraph is a graph on the set Nodes. Two nodes in Nodes are incident in the node s-linegraph if they share at lease s incident edges in Edges; that is, there are at least s elements of Edges to which they both belong. The edge s-linegraph is a graph on the set Edges. Two edges in Edges are incident in the edge s-linegraph if they share at least s incident nodes in Nodes; that is, the edges intersect in at least s nodes in Nodes.

5.4 HyperNetX Packages

5.4.1 classes

classes package

Submodules

classes.entityset module

class classes.entityset.**EntitySet**(*entity: DataFrame | Mapping[T, Iterable[T]] | Iterable[Iterable[T]] | Mapping[T, Mapping[T, Any]] | None = None*, *data_cols: Sequence[T] = (0, 1)*, *data: ndarray | None = None*, *static: bool = True*, *labels: OrderedDict[T, Sequence[T]] | None = None*, *uid: Hashable | None = None*, *weight_col: str | int | None = 'cell_weights'*, *weights: Sequence[float] | float | int | str | None = 1*, *aggregateby: str | dict | None = 'sum'*, *properties: DataFrame | dict[int, dict[T, dict[Any, Any]]] | None = None*, *misc_props_col: str | None = None*, *level_col: str = 'level'*, *id_col: str = 'id'*, *cell_properties: Sequence[T] | DataFrame | dict[T, dict[T, dict[Any, Any]]] | None = None*, *misc_cell_props_col: str | None = None*)

Bases: `object`

Base class for handling N-dimensional data when building network-like models, i.e., `Hypergraph`

Parameters

- **entity** (*pandas.DataFrame, dict of lists or sets, dict of dicts, list of lists or sets, optional*) – If a `DataFrame` with N columns, represents N-dimensional entity data (data table). Otherwise, represents 2-dimensional entity data (system of sets).
- **data_cols** (*sequence of ints or strings, default=(0,1)*) –
- **level1** (*str or int, default = 0*) –
- **level2** (*str or int, default = 1*) –
- **data** (*numpy.ndarray, optional*) – 2D M x N `ndarray` of `ints` (data table); sparse representation of an N-dimensional incidence tensor with M nonzero cells. Ignored if *entity* is provided.
- **static** (*bool, default=True*) – If `True`, entity data may not be altered, and the `state_dict` will never be cleared. Otherwise, rows may be added to and removed from the data table, and updates will clear the `state_dict`.
- **labels** (*collections.OrderedDict of lists, optional*) – User-specified labels in corresponding order to `ints` in *data*. Ignored if *entity* is provided or *data* is not provided.
- **uid** (*hashable, optional*) – A unique identifier for the object

- **weight_col** (*string or int, default="cell_weights"*) –
- **weights** (*sequence of float, float, int, str, default=1*) – User-specified cell weights corresponding to entity data. If sequence of `floats` and *entity* or *data* defines a data table,

  length must equal the number of rows.

  If sequence of `floats` and *entity* defines a system of sets,
  length must equal the total sum of the sizes of all sets.

  If `str` and *entity* is a `DataFrame`,
  must be the name of a column in *entity*.

  Otherwise, weight for all cells is assumed to be 1.
- **aggregateby** (*{'sum', 'last', count', 'mean','median', max', 'min', 'first', None}, default="sum"*) – Name of function to use for aggregating cell weights of duplicate rows when *entity* or *data* defines a data table. If None, duplicate rows will be dropped without aggregating cell weights. Ignored if *entity* defines a system of sets.
- **properties** (*pandas.DataFrame or doubly-nested dict, optional*) – User-specified properties to be assigned to individual items in the data, i.e., cell entries in a data table; sets or set elements in a system of sets. See Notes for detailed explanation. If `DataFrame`, each row gives `[optional item level, item label, optional named properties, {property name: property value}]` (order of columns does not matter; see Notes for an example). If doubly-nested dict, `{item level: {item label: {property name: property value}}}`.
- **misc_props_col** (*str, default="properties"*) – Column names for miscellaneous properties, level index, and item name in `properties`; see Notes for explanation.
- **level_col** (*str, default="level"*) –
- **id_col** (*str, default="id"*) –
- **cell_properties** (*sequence of int or str, pandas.DataFrame, or doubly-nested dict, optional*) –
- **misc_cell_props_col** (*str, default="cell_properties"*) –

Notes

A property is a named attribute assigned to a single item in the data.

You can pass a **table of properties** to *properties* as a `DataFrame`:

Level (optional)	ID	[explicit property type]	[. . .]	misc. properties
0	level 0 item	property value	. . .	{property name: property value}
1	level 1 item	property value	. . .	{property name: property value}
. . .	. . .	. . .	. . .	. . .
N	level N item	property value	. . .	{property name: property value}

The Level column is optional. If not provided, properties will be assigned by ID (i.e., if an ID appears at multiple levels, the same properties will be assigned to all occurrences).

The names of the Level (if provided) and ID columns must be specified by *level_col* and *id_col*. *misc_props_col* can be used to specify the name of the column to be used for miscellaneous properties; if no column by that name is found, a new column will be created and populated with empty `dicts`. All other columns will be considered explicit property types. The order of the columns does not matter.

This method assumes that there are no rows with the same (Level, ID); if duplicates are found, all but the first occurrence will be dropped.

add(**args*) → Self

Updates the underlying data table with new entity data from multiple sources

Parameters
***args** – variable length argument list of Entity and/or representations of entity data

Returns
self

Return type
EntitySet

> **Warning:** Adding an element directly to an Entity will not add the element to any Hypergraphs constructed from that Entity, and will cause an error. Use `Hypergraph.add_edge` or `Hypergraph.add_node_to_edge` instead.

See also:

add_element
update from a single source

`Hypergraph.add_edge`, `Hypergraph.add_node_to_edge`

add_element(*data: DataFrame | Mapping[T, Iterable[T]] | Iterable[Iterable[T]] | Mapping[T, Mapping[T, Any]]*) → Self

Updates the underlying data table with new entity data

Supports adding from either an existing EntitySet or a representation of entity (data table or labeled system of sets are both supported representations)

Parameters
data (*pandas.DataFrame*, dict of lists or sets, lists of lists, or nested dict) –

Returns
self

Return type
EntitySet

> **Warning:** Adding an element directly to an Entity will not add the element to any Hypergraphs constructed from that Entity, and will cause an error. Use *Hypergraph.add_edge* or *Hypergraph.add_node_to_edge* instead.

See also:

add
takes multiple sources of new entity data as variable length argument list

`Hypergraph.add_edge`, `Hypergraph.add_node_to_edge`

add_elements_from(*arg_set*) → Self

Adds arguments from an iterable to the data table one at a time

DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]
Duplicates *add*

Parameters
arg_set (*iterable*) – list of Entity and/or representations of entity data

Returns
self

Return type
EntitySet

assign_cell_properties(*cell_props: DataFrame | dict[T, dict[T, dict[Any, Any]]]*, *misc_col: str | None = None*, *replace: bool = False*) → None

Assign new properties to cells of the incidence matrix and update `properties`

Parameters

- **cell_props** (*pandas.DataFrame, dict of iterables, or doubly-nested dict, optional*) – See documentation of the *cell_properties* parameter in `EntitySet`
- **misc_col** (*str, optional*) – name of column to be used for miscellaneous cell property dicts
- **replace** (*bool, default=False*) – If True, replace existing `cell_properties` with result; otherwise update with new values from result

Raises
AttributeError – Not supported for :attr:`dimsize`=1

assign_properties(*props: DataFrame | dict[int, dict[T, dict[Any, Any]]]*, *misc_col: str | None = None*, *level_col=0*, *id_col=1*) → None

Assign new properties to items in the data table, update `properties`

Parameters

- **props** (*pandas.DataFrame or doubly-nested dict*) – See documentation of the *properties* parameter in `EntitySet`
- **level_col** (*str, optional*) – column names corresponding to the levels, items, and misc. properties; if None, default to `_level_col`, `_id_col`, `_misc_props_col`, respectively.
- **id_col** (*str, optional*) – column names corresponding to the levels, items, and misc. properties; if None, default to `_level_col`, `_id_col`, `_misc_props_col`, respectively.
- **misc_col** (*str, optional*) – column names corresponding to the levels, items, and misc. properties; if None, default to `_level_col`, `_id_col`, `_misc_props_col`, respectively.

See also:

`properties`

property cell_properties: DataFrame | None

Properties assigned to cells of the incidence matrix

Returns
Returns None if `dimsize` < 2

Return type
pandas.DataFrame, optional

property cell_weights: dict[str, tuple[T]]

Cell weights corresponding to each row of the underlying data table

Returns
dict of {tuple – Keyed by row of data table (as a tuple)

Return type
int or float}

property children: set

Labels of all items in level 1 (second column) of the underlying data table

Return type
set

See also:

uidset
Labels of all items in level 0 (first column)

uidset_by_level, *uidset_by_column*

collapse_identical_elements(*return_equivalence_classes: bool = False*, ***kwargs*) → *EntitySet* | tuple[*classes.entityset.EntitySet*, dict[str, list[str]]]

Create a new `EntitySet` by collapsing sets with the same set elements

Each item in level 0 (first column) defines a set containing all the level 1 (second column) items with which it appears in the same row of the underlying data table.

Parameters

- **return_equivalence_classes** (*bool, default=False*) – If True, return a dictionary of equivalence classes keyed by new edge names
- ****kwargs** – Extra arguments to `EntitySet` constructor

Returns

- **new_entity** (*EntitySet*) – new `EntitySet` with identical sets collapsed; if all sets are unique, the system of sets will be the same as the original.
- **equivalence_classes** (*dict of lists, optional*) – if *return_equivalence_classes`=True, ``{collapsed set label: [level 0 item labels]}`*.

property data: ndarray

Sparse representation of the data table as an incidence tensor

This can also be thought of as an encoding of *dataframe*, where items in each column of the data table are translated to their int position in the *self.labels[column]* list :returns: 2D array of ints representing rows of the underlying data table as indices in an incidence tensor :rtype: numpy.ndarray

See also:

labels, *dataframe*

property dataframe: DataFrame

The underlying data table stored by the Entity

Return type
pandas.DataFrame

property dimensions: tuple[int]

Dimensions of data i.e., the number of distinct items in each level (column) of the underlying data table

Returns
Length and order corresponds to columns of *self.dataframe* (excluding cell weight column)

Return type
tuple of ints

property dimsize: int

Number of levels (columns) in the underlying data table

Returns
Equal to length of *self.dimensions*

Return type
int

property elements: dict[Any, hypernetx.classes.helpers.AttrList]

System of sets representation of the first two levels (columns) of the underlying data table

Each item in level 0 (first column) defines a set containing all the level 1 (second column) items with which it appears in the same row of the underlying data table

Returns
System of sets representation as dict of {level 0 item : AttrList(level 1 items)}

Return type
dict of *AttrList*

See also:

incidence_dict
same data as dict of list

memberships
dual of this representation, i.e., each item in level 1 (second column) defines a set

elements_by_level, *elements_by_column*

elements_by_column(*col1: Hashable*, *col2: Hashable*) → dict[Any, hypernetx.classes.helpers.AttrList]

System of sets representation of two columns (levels) of the underlying data table

Each item in col1 defines a set containing all the col2 items with which it appears in the same row of the underlying data table

Properties can be accessed and assigned to items in col1

Parameters

- **col1** (*Hashable*) – name of column whose items define sets
- **col2** (*Hashable*) – name of column whose items are elements in the system of sets

Returns
System of sets representation as dict of {col1 item : AttrList(col2 items)}

Return type
dict of *AttrList*

See also:

elements, *memberships*

elements_by_level
same functionality, takes level indices instead of column names

elements_by_level(*level1: int*, *level2: int*) → dict[Any, hypernetx.classes.helpers.AttrList]

System of sets representation of two levels (columns) of the underlying data table

Each item in level1 defines a set containing all the level2 items with which it appears in the same row of the underlying data table

Properties can be accessed and assigned to items in level1

Parameters

- **level1** (*int*) – index of level whose items define sets
- **level2** (*int*) – index of level whose items are elements in the system of sets

Returns
System of sets representation as dict of {level1 item : AttrList(level2 items)}

Return type
dict of *AttrList*

See also:

elements, *memberships*

elements_by_column
same functionality, takes column names instead of level indices

property empty: bool

Whether the underlying data table is empty or not

Return type
bool

See also:

is_empty
for checking whether a specified level (column) is empty

dimsize
0 if empty

encode(*data: DataFrame*) → array

Encode dataframe to numpy array

Parameters
data (*dataframe, dataframe columns must have dtype set to 'category'*) –

Return type
numpy.array

get_cell_properties(*item1: T*, *item2: T*) → dict[Any, Any] | None

Get all properties of a cell, i.e., incidence between items of different levels

Parameters

- **item1** (*hashable*) – name of an item in level 0
- **item2** (*hashable*) – name of an item in level 1

Returns

- *dict* – {named cell property: cell property value, ..., misc. cell property column name: {cell property name: cell property value}}
- *None* – If properties do not exist

See also:

get_cell_property, *set_cell_property*

get_cell_property(*item1: T*, *item2: T*, *prop_name: Any*) → Any

Get a property of a cell i.e., incidence between items of different levels

Parameters

- **item1** (*hashable*) – name of an item in level 0
- **item2** (*hashable*) – name of an item in level 1
- **prop_name** (*hashable*) – name of the cell property to get

Returns

- **prop_val** (*any*) – value of the cell property
- *None* – If prop_name not found

Raises

`KeyError` – If *(item1, item2)* is not in *cell_properties*

See also:

get_cell_properties, *set_cell_property*

get_properties(*item: T*, *level: int | None = None*) → dict[Any, Any]

Get all properties of an item

Parameters

- **item** (*hashable*) – name of an item
- **level** (*int, optional*) – level index of the item

Returns

prop_vals – {named property: property value, ..., misc. property column name: {property name: property value}}

Return type

dict

Raises

`KeyError` – if (*level*, *item*) is not in *properties*, or if *level* is not provided and *item* is not in *properties*

Warns

UserWarning – If *level* is not provided and *item* appears in multiple levels, assumes the first (closest to 0)

See also:

get_property, *set_property*

get_property(*item: T*, *prop_name: Any*, *level: int | None = None*) → Any

Get a property of an item

Parameters

- **item** (*hashable*) – name of an item

- **prop_name** (*hashable*) – name of the property to get
- **level** (*int, optional*) – level index of the item

Returns

- **prop_val** (*any*) – value of the property
- *None* – if property not found

Raises
: **KeyError** – if (*level*, *item*) is not in `properties`, or if *level* is not provided and *item* is not in `properties`

Warns
: **UserWarning** – If *level* is not provided and *item* appears in multiple levels, assumes the first (closest to 0)

See also:

`get_properties`, `set_property`

property incidence_dict: dict[T, list[T]]

System of sets representation of the first two levels (columns) of the underlying data table

Returns
: System of sets representation as dict of {level 0 item : AttrList(level 1 items)}

Return type
: dict of list

See also:

`elements`
: same data as dict of AttrList

incidence_matrix(*level1: int = 0*, *level2: int = 1*, *weights: bool | dict = False*, *aggregateby: str = 'count'*) → csr_matrix | None

Incidence matrix representation for two levels (columns) of the underlying data table

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

If *level1* and *level2* contain N and M distinct items, respectively, the incidence matrix will be M x N. In other words, the items in *level1* and *level2* correspond to the columns and rows of the incidence matrix, respectively, in the order in which they appear in *self.labels[column1]* and *self.labels[column2]* (*column1* and *column2* are the column labels of *level1* and *level2*)

Parameters

- **level1** (*int, default=0*) – index of first level (column)
- **level2** (*int, default=1*) – index of second level
- **weights** (*bool or dict, default=False*) – If False all nonzero entries are 1. If True all nonzero entries are filled by self.cell_weight dictionary values, use `aggregateby` to specify how duplicate entries should have weights aggregated. If dict of {(level1 item, level2 item): weight value} form; only nonzero cells in the incidence matrix will be updated by dictionary, i.e., *level1 item* and *level2 item* must appear in the same row at least once in the underlying data table
- **aggregateby** (*{'last', count', 'sum', 'mean','median', max', 'min', 'first', 'last', None}, default='count'*) –

Method to aggregate weights of duplicate rows in data table.
If None, then all cell weights will be set to 1.

- **index** (*bool, optional*) – Not used

Returns
sparse representation of incidence matrix (i.e. Compressed Sparse Row matrix)

Return type
scipy.sparse.csr.csr_matrix

Note: In the context of Hypergraphs, think *level1 = edges, level2 = nodes*

---

index(*column: str*, *value: str | None = None*) → int | tuple[int, int]

Get level index corresponding to a column and (optionally) the index of a value in that column

The index of `value` is its position in the list given by `self.labels[column]`, which is used in the integer encoding of the data table `self.data`

Parameters

- **column** (*str*) – name of a column in self.dataframe
- **value** (*str, optional*) – label of an item in the specified column

Returns
level index corresponding to column, index of value if provided

Return type
int or (int, int)

See also:

indices
for finding indices of multiple values in a column

level
same functionality, search for the value without specifying column

indices(*column: str*, *values: str | Iterable[str]*) → list[int]

Get indices of one or more value(s) in a column

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters

- **column** (*str*) –
- **values** (*str or iterable of str*) –

Returns
indices of values

Return type
list of int

See also:

index
for finding level index of a column and index of a single value

is_empty(*level: int = 0*) → bool

Whether a specified level (column) of the underlying data table is empty or not

Parameters
`level` (*int*) – the level of a column in the underlying data table

Return type
bool

See also:

empty
for checking whether the underlying data table is empty

size
number of items in a level (columns); 0 if level is empty

property isstatic: bool

Whether to treat the underlying data as static or not

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE] If True, the underlying data may not be altered, and the state_dict will never be cleared Otherwise, rows may be added to and removed from the data table, and updates will clear the state_dict

Return type
bool

property labels: dict[str, list]

Labels of all items in each column of the underlying data table

Returns
dict of {column name: [item labels]} The order of [item labels] corresponds to the int encoding of each item in *self.data*.

Return type
dict of lists

See also:

data, *dataframe*

level(*item: str*, *min_level: int = 0*, *max_level: int | None = None*, *return_index: bool = True*) → int | tuple[int, int] | None

First level containing the given item label

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Order of levels corresponds to order of columns in *self.dataframe*

Parameters

- **item** (*str*) –
- **min_level** (*int, default=0*) – minimum inclusive bound on range of levels to search for item
- **max_level** (*int, optional*) – maximum inclusive bound on range of levels to search for item
- **return_index** (*bool, default=True*) – If True, return index of item within the level

Returns
index of first level containing the item, index of item if *return_index=True* returns None if item is not found

Return type
int, (int, int), or None

See also:

index, *indices*

property memberships: dict[Any, hypernetx.classes.helpers.AttrList]

System of sets representation of the first two levels (columns) of the underlying data table

Each item in level 1 (second column) defines a set containing all the level 0 (first column) items with which it appears in the same row of the underlying data table

Returns
System of sets representation as dict of {level 1 item : AttrList(level 0 items)}

Return type
dict of *AttrList*

See also:

elements
dual of this representation i.e., each item in level 0 (first column) defines a set

elements_by_level, *elements_by_column*

property properties: DataFrame

Properties assigned to items in the underlying data table

Returns
pandas.DataFrame a dataframe with the following columns

Return type
level/(edge|node), uid, weight, properties

remove(**args: T*) → *EntitySet*

Removes all rows containing specified item(s) from the underlying data table

Parameters
***args** – variable length argument list of items which are of type string or int

Returns
self

Return type
EntitySet

See also:

remove_element
remove all rows containing a single specified item

remove_element(*item: T*) → None

Removes all rows containing a specified item from the underlying data table

Parameters
item (*Union[str, int]*) – the label of an edge

See also:

remove
same functionality, accepts variable length argument list of item labels

remove_elements_from(*arg_set*)

Removes all rows containing specified item(s) from the underlying data table

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Duplicates *remove*

Parameters
arg_set (*iterable*) – list of item labels

Returns
self

Return type
EntitySet

restrict_to(*indices: int | Iterable[int]*, ***kwargs*) → *EntitySet*

Alias of `restrict_to_indices()` with default parameter `level`=0

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters

- **indices** (*array_like of int*) – indices of item label(s) in *level* to restrict to
- ****kwargs** – Extra arguments to `EntitySet` constructor

Return type
EntitySet

See also:

restrict_to_indices

restrict_to_indices(*indices: int | Iterable[int]*, *level: int = 0*, ***kwargs*) → *EntitySet*

Create a new EntitySet by restricting the data table to rows containing specific items in a given level

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters

- **indices** (*int or iterable of int*) – indices of item label(s) in *level* to restrict to
- **level** (*int, default=0*) – level index
- ****kwargs** – Extra arguments to *EntitySet* constructor

Return type
EntitySet

restrict_to_levels(*levels: int | Iterable[int]*, *weights: bool = False*, *aggregateby: str | None = 'sum'*, *keep_memberships: bool = True*, ***kwargs*) → *EntitySet*

Create a new EntitySet by restricting to a subset of levels (columns) in the underlying data table

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters

- **levels** (*array-like of int*) – indices of a subset of levels (columns) of data

- **weights** (*bool, default=False*) – If True, aggregate existing cell weights to get new cell weights. Otherwise, all new cell weights will be 1.
- **aggregateby** (*{'sum', 'first', 'last', 'count', 'mean', 'median', 'max', 'min', None}, optional*) – Method to aggregate weights of duplicate rows in data table If None or `weights`=False then all new cell weights will be 1
- **keep_memberships** (*bool, default=True*) – Whether to preserve membership information for the discarded level when the new `EntitySet` is restricted to a single level
- ****kwargs** – Extra arguments to `EntitySet` constructor

Return type
EntitySet

Raises
`KeyError` – If *levels* contains any invalid values

set_cell_property(*item1: T, item2: T, prop_name: Any, prop_val: Any*) → None

Set a property of a cell i.e., incidence between items of different levels

Parameters

- **item1** (*hashable*) – name of an item in level 0
- **item2** (*hashable*) – name of an item in level 1
- **prop_name** (*hashable*) – name of the cell property to set
- **prop_val** (*any*) – value of the cell property to set

See also:

get_cell_property, *get_cell_properties*

set_property(*item: T, prop_name: Any, prop_val: Any, level: int | None = None*) → None

Set a property of an item

Parameters

- **item** (*hashable*) – name of an item
- **prop_name** (*hashable*) – name of the property to set
- **prop_val** (*any*) – value of the property to set
- **level** (*int, optional*) – level index of the item; required if *item* is not already in `properties`

Raises
`ValueError` – If *level* is not provided and *item* is not in `properties`

Warns
UserWarning – If *level* is not provided and *item* appears in multiple levels, assumes the first (closest to 0)

See also:

get_property, *get_properties*

size(*level: int = 0*) → int

The number of items in a level of the underlying data table

Equivalent to `self.dimensions[level]`

Parameters
`level` (*int, default=0*) –

Return type
int

See also:

dimensions

`translate`(*level: int, index: int | list[int]*) → str | list[str]

Given indices of a level and value(s), return the corresponding value label(s)

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters

- **level** (*int*) – the index of the level
- **index** (*int or list of int*) – value index or indices

Returns
label(s) corresponding to value index or indices

Return type
str or list of str

See also:

translate_arr
translate a full row of value indices across all levels (columns)

`translate_arr`(*coords: tuple[int, int]*) → list[str]

Translate a full encoded row of the data table e.g., a row of `self.data`

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters
`coords` (*tuple of ints*) – encoded value indices, with one value index for each level of the data

Returns
full row of translated value labels

Return type
list of str

`property uid: Hashable`

User-defined unique identifier for the *Entity*

Return type
Hashable

`property uidset: set`

Labels of all items in level 0 (first column) of the underlying data table

Return type
set

See also:

children
Labels of all items in level 1 (second column)

uidset_by_level, *uidset_by_column*

uidset_by_column(*column: Hashable*) → set

Labels of all items in a particular column (level) of the underlying data table

Parameters
column (*Hashable*) – Name of a column in *self.dataframe*

Return type
set

See also:

uidset
Labels of all items in level 0 (first column)

children
Labels of all items in level 1 (second column)

uidset_by_level
Same functionality, takes the level index instead of column name

uidset_by_level(*level: int*) → set

Labels of all items in a particular level (column) of the underlying data table

Parameters
level (*int*) –

Return type
set

See also:

uidset
Labels of all items in level 0 (first column)

children
Labels of all items in level 1 (second column)

uidset_by_column
Same functionality, takes the column name instead of level index

classes.entityset.**build_dataframe_from_entity**(*entity: DataFrame | Mapping[str | int, Iterable[str | int]] | Iterable[Iterable[str | int]] | Mapping[T, Mapping[T, Mapping[T, Any]]], data_cols: Sequence[str | int]*) → DataFrame

classes.helpers module

class classes.helpers.**AttrList**(*entity: EntitySet, key: tuple[int, str | int], initlist: list | None = None*)

Bases: UserList

Custom list wrapper for integrated property storage in Entity

Parameters

- **entity** (*hypernetx.EntitySet*) –
- **key** (*tuple of (int, str or int)*) – (level, item)

- **initlist** (*list, optional*) – list of elements, passed to UserList constructor

classes.helpers.**assign_weights**(*df: DataFrame*, *weights: list | tuple | ndarray | Hashable = 1*, *weight_col: Hashable = 'cell_weights'*)

Parameters

- **df** (*pandas.DataFrame*) – A DataFrame to assign a weight column to
- **weights** (*array-like or Hashable, optional*) – If numpy.ndarray with the same length as df, create a new weight column with these values. If Hashable, must be the name of a column of df to assign as the weight column Otherwise, create a new weight column assigning a weight of 1 to every row
- **weight_col** (*Hashable*) – Name for new column if one is created (not used if the name of an existing column is passed as weights)

Returns

- **df** (*pandas.DataFrame*) – The original DataFrame with a new column added if needed
- **weight_col** (*str*) – Name of the column assigned to hold weights

Note: TODO: move logic for default weights inside this method

classes.helpers.**create_dataframe**(*data: Mapping[str | int, Iterable[str | int]]*) → DataFrame

Create a valid pandas Dataframe that can be used for the 'entity' param in EntitySet

classes.helpers.**create_properties**(*props: DataFrame | dict[str | int, collections.abc.Iterable[str | int]] | dict[str | int, dict[str | int, dict[Any, Any]]] | None*, *index_cols: list[str]*, *misc_col: str*) → DataFrame

Helper function for initializing properties and cell properties

Parameters

- **props** (*pandas.DataFrame, dict of iterables, doubly-nested dict, or None*) – See documentation of the *properties* parameter in Entity, *cell_properties* parameter in EntitySet
- **index_cols** (*list of str*) – names of columns to be used as levels of the MultiIndex
- **misc_col** (*str*) – name of column to be used for miscellaneous property dicts

Returns

with MultiIndex on *index_cols*; each entry of the miscellaneous column holds dict of {property name: property value}

Return type

pandas.DataFrame

classes.helpers.**dict_depth**(*dic*, *level=0*)

classes.helpers.**encode**(*data: DataFrame*)

Encode dataframe to numpy array

Parameters

data (*dataframe*) –

Return type

numpy.array

classes.helpers.**merge_nested_dicts**(*a*, *b*, *path=None*)

merges b into a

classes.helpers.**remove_row_duplicates**(*df*, *data_cols*, *weights=1*, *weight_col='cell_weights'*, *aggregateby=None*)

Removes and aggregates duplicate rows of a DataFrame using groupby Also sets the dtype of entity data columns to categorical (simplifies encoding, etc.)

Parameters

- **df** (*pandas.DataFrame*) – A DataFrame to remove or aggregate duplicate rows from
- **data_cols** (*list*) – A list of column names in df to perform the groupby on / remove duplicates from
- **weights** (*array-like or Hashable, optional*) – Argument passed to assign_weights
- **aggregateby** (*str, optional, default='sum'*) – A valid aggregation method for pandas groupby If None, drop duplicates without aggregating weights

Returns

- **df** (*pandas.DataFrame*) – The DataFrame with duplicate rows removed or aggregated
- **weight_col** (*Hashable*) – The name of the column holding aggregated weights, or None if aggregateby=None

classes.helpers.**validate_mapping_for_dataframe**(*data: Mapping[str | int, Iterable[str | int]]*) → None

classes.hypergraph module

class classes.hypergraph.**Hypergraph**(*setsystem: DataFrame | ndarray | Mapping[T, Iterable[T]] | Iterable[Iterable[T]] | Mapping[T, Mapping[T, Mapping[str, Any]]] | None = None*, *edge_col: str | int = 0*, *node_col: str | int = 1*, *cell_weight_col: str | int | None = 'cell_weights'*, *cell_weights: Sequence[float] | float = 1.0*, *cell_properties: Sequence[str | int] | Mapping[T, Mapping[T, Mapping[str, Any]]] | None = None*, *misc_cell_properties_col: str | int | None = None*, *aggregateby: str | dict[str, str] = 'first'*, *edge_properties: DataFrame | dict[T, dict[Any, Any]] | None = None*, *node_properties: DataFrame | dict[T, dict[Any, Any]] | None = None*, *properties: DataFrame | dict[T, dict[Any, Any]] | dict[T, dict[T, dict[Any, Any]]] | None = None*, *misc_properties_col: str | int | None = None*, *edge_weight_prop_col: str | int = 'weight'*, *node_weight_prop_col: str | int = 'weight'*, *weight_prop_col: str | int = 'weight'*, *default_edge_weight: float | None = None*, *default_node_weight: float | None = None*, *default_weight: float = 1.0*, *name: str | None = None*, ***kwargs*)

Bases: object

Parameters

- **setsystem** (*(optional) dict of iterables, dict of dicts,iterable of iterables,*) – pandas.DataFrame, numpy.ndarray, default = None See SetSystem above for additional setsystem requirements.
- **edge_col** (*(optional) str | int, default = 0*) – column index (or name) in pandas.dataframe or numpy.ndarray, used for (hyper)edge ids. Will be used to reference edgeids for all set systems.

- **node_col** (*(optional) str | int, default = 1*) – column index (or name) in pandas.dataframe or numpy.ndarray, used for node ids. Will be used to reference nodeids for all set systems.
- **cell_weight_col** (*(optional) str | int, default = None*) – column index (or name) in pandas.dataframe or numpy.ndarray used for referencing cell weights. For a dict of dicts references key in cell property dicts.
- **cell_weights** (*(optional) Sequence[float,int] | int | float , default = 1.0*) – User specified cell_weights or default cell weight. Sequential values are only used if setsystem is a dataframe or ndarray in which case the sequence must have the same length and order as these objects. Sequential values are ignored for dataframes if cell_weight_col is already a column in the data frame. If cell_weights is assigned a single value then it will be used as default for missing values or when no cell_weight_col is given.
- **cell_properties** (*(optional) Sequence[int | str] | Mapping[T, Mapping[T,Mapping[str,Any]]],*) – default = None Column names from pd.DataFrame to use as cell properties or a dict assigning cell_property to incidence pairs of edges and nodes. Will generate a misc_cell_properties, which may have variable lengths per cell.
- **misc_cell_properties** (*(optional) str | int, default = None*) – Column name of dataframe corresponding to a column of variable length property dictionaries for the cell. Ignored for other setsystem types.
- **aggregateby** (*(optional) str, dict, default = 'first'*) – By default duplicate edge,node incidences will be dropped unless specified with *aggregateby*. See pandas.DataFrame.agg() methods for additional syntax and usage information.
- **edge_properties** (*(optional) pd.DataFrame | dict, default = None*) – Properties associated with edge ids. First column of dataframe or keys of dict link to edge ids in setsystem.
- **node_properties** (*(optional) pd.DataFrame | dict, default = None*) – Properties associated with node ids. First column of dataframe or keys of dict link to node ids in setsystem.
- **properties** (*(optional) pd.DataFrame | dict, 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 and different properties then use the edge_properties and node_properties keywords.
- **misc_properties** (*(optional) int | str, default = None*) – Column of property dataframes with dtype=dict. Intended for variable length property dictionaries for the objects.
- **edge_weight_prop** (*(optional) str, default = None,*) – Name of property in edge_properties to use for weight.
- **node_weight_prop** (*(optional) str, default = None,*) – Name of property in node_properties to use for weight.
- **weight_prop** (*(optional) str, default = None*) – Name of property in properties to use for 'weight'
- **default_edge_weight** (*(optional) int | float, default = 1*) – Used when edge weight property is missing or undefined.
- **default_node_weight** (*(optional) int | float, default = 1*) – Used when node weight property is missing or undefined

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

Hypergraphs in HNX 2.0

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 identifiers instead of their contents. For example, if V = {1,2,3} and E = {e1,e2,e3}, where e1 = {1,2}, e2 = {1,2}, and e3 = {1,2,3}, the edges e1 and e2 contain the same set of nodes and yet are distinct and are distinguishable within H = (V,E).

New as of version 2.0, 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.

SetSystems

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

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

```
>>> H = Hypergraph([{1,2},{1,2},{1,2,3}])
```

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):

```
>>> H = Hypergraph({'e1':[1,2],'e2':[1,2],'e3':[1,2,3]})
```

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:

```
>>> d = {'e1':{ 1: {'w':0.5, 'name': 'related_to'},
>>>             2: {'w':0.1, 'name': 'related_to',
>>>                 'startdate': '05.13.2020'}},
>>>      'e2':{ 1: {'w':0.52, 'name': 'owned_by'},
>>>             2: {'w':0.2}},
>>>      'e3':{ 1: {'w':0.5, 'name': 'related_to'},
>>>             2: {'w':0.2, 'name': 'owner_of'},
>>>             3: {'w':1, 'type': 'relationship'}}
```

```
>>> H = Hypergraph(d, cell_weight_col='w')
```

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:

col1	col2	w	col3
e1	1	0.5	{'name':'related_to'}
e1	2	0.1	**{"name":"related_to",** "start-date":"05.13.2020"}
e2	1	0.52	{"name":"owned_by"}
e2	2	0.2	
...	...	...	{...}

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

```
>>> H = Hypergraph(df,edge_col="col1",node_col="col2",
>>>                 cell_weight_col="w",misc_cell_properties="col3")
```

5. **numpy.ndarray** For homogeneous datasets given in an ndarray a pandas dataframe is generated and column names are added from the edge_col and node_col arguments. Cell properties containing multiple data types are added with a separate dataframe or dict and passed through the cell_properties keyword.

```
>>> arr = np.array([['e1','1'],['e1','2'],
>>>                 ['e2','1'],['e2','2'],
>>>                 ['e3','1'],['e3','2'],['e3','3']])
>>> H = hnx.Hypergraph(arr, column_names=['col1','col2'])
```

Edge and Node Properties

Properties specific to a single edge or node are 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 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:

id	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:

```
dp = {id1 : {prop1:val1, prop2,val2,...}, id2 : ... }
```

A properties dataframe may be used for nodes and edges sharing ids but differing in cell properties by adding a level index using 0 for edges and 1 for nodes:

level	id	weight	properties
0	e1	5.0	{'type':'event'}
0	e2	0.52	{"name":"owned_by"}
…	…	…	{…}
1	1.2	{'color':'red'}	
2	.003	{'name':'Fido','color':'brown'}	
…	…	…	{…}

Weights

The default key for cell and object weights is "weight". The default value is 1. Weights may be assigned and/or a new default prescribed in the constructor using **cell_weight_col** and **cell_weights** for incidence pairs, and using **edge_weight_prop, node_weight_prop, weight_prop, default_edge_weight,** and **default_node_weight** for node and edge weights.

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

The s-adjacency matrix for the hypergraph.

Parameters

- `s` (*int, optional, default = 1*) –
- `index` (*boolean, optional, default = False*) – if True, will return the index of ids for rows and columns
- `remove_empty_rows` (*boolean, optional, default = False*) –

Returns

- **adjacency_matrix** (*scipy.sparse.csr.csr_matrix*)
- **node_index** (*list*) – index of ids for rows and columns

`auxiliary_matrix`(*s=1*, *node=True*, *index=False*)

The unweighted s-edge or node auxiliary matrix for hypergraph

Parameters

- `s` (*int, optional, default = 1*) –
- `node` (*bool, optional, default = True*) – whether to return based on node or edge adjacencies

Returns

- **auxiliary_matrix** (*scipy.sparse.csr.csr_matrix*) – Node/Edge adjacency matrix with empty rows and columns removed
- **index** (*np.array*) – row and column index of userids

`bipartite`()

Constructs the networkX bipartite graph associated to hypergraph.

Returns
bipartite

Return type
nx.Graph()

Notes

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 n in e there is an edge (n,e) in the graph.

collapse_edges(*name=None*, *return_equivalence_classes=False*, *use_reps=None*, *return_counts=None*)

Constructs a new hypergraph gotten by identifying edges containing the same nodes

Parameters

- **name** (*hashable, optional, default = None*) –
- **return_equivalence_classes** (*boolean, optional, default = False*) – Returns a dictionary of edge equivalence classes keyed by frozen sets of nodes

Returns

- **new hypergraph** (*Hypergraph*) – Equivalent edges are collapsed to a single edge named by a representative of the equivalent edges followed by a colon and the number of edges it represents.
- **equivalence_classes** (*dict*) – A dictionary keyed by representative edge names with values equal to the edges in its equivalence class

Notes

Two edges are identified 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 edge from the collapsed edges followed by a colon and the number of elements in its equivalence class as uid for the new edge

collapse_nodes(*name=None*, *return_equivalence_classes=False*, *use_reps=None*, *return_counts=None*) → *Hypergraph*

Constructs a new hypergraph gotten by identifying nodes contained by the same edges

Parameters

- **name** (*str, optional, default = None*) –
- **return_equivalence_classes** (*boolean, optional, default = False*) – Returns a dictionary of node equivalence classes keyed by frozen sets of edges
- **use_reps** (*boolean, optional, default = None*) – [DEPRECATED; WILL BE REMOVED IN NEXT RELEASE] Choose a single element from the collapsed nodes as uid for the new node, otherwise uses a frozen set of the uids of nodes in the equivalence class. If use_reps is True the new nodes have uids given by a tuple of the rep and the count
- **return_counts** (*boolean, optional, default = None*) – [DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Returns

new hypergraph

Return type

Hypergraph

Notes

Two nodes are identified 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 followed by the number of members of the class.

Example

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

`collapse_nodes_and_edges`(*name=None*, *return_equivalence_classes=False*, *use_reps=None*, *return_counts=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 frozen sets of nodes
- **use_reps** (*boolean, optional, default = None*) – [DEPRECATED; WILL BE REMOVED IN NEXT RELEASE] Choose a single element from the collapsed elements as a representative. If use_reps is True, the new elements are keyed by a tuple of the rep and the count.
- **return_counts** (*boolean, optional, default = None*) – [DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Returns

new hypergraph

Return type

Hypergraph

Notes

Collapses the Nodes and Edges of EntitySets. 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 followed by the number of members of the class.

Example

```
>>> 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(*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 pairs and their properties.

Return type
pd.DataFrame

degree(*node*, *s=1*, *max_size=None*)

The number of edges of size s that contain node.

Parameters

- **node** (*hashable*) – identifier for the node.
- **s** (*positive integer, optional, default 1*) – smallest size of edge to consider in degree
- **max_size** (*positive integer or None, optional, default = None*) – largest size of edge to consider in degree

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_start and v_end are s-walk connected if there is a sequence of nodes v_start, v_1, v_2, … v_n-1, v_end such that consecutive nodes are s-adjacent. If the graph is not connected, an error will be raised.

`dim`(*edge*)

Same as size(edge)-1.

`distance`(*source*, *target*, *s=1*)

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

Parameters

- `source` (*node.uid or node*) – a node in the hypergraph
- `target` (*node.uid or node*) – a node in the hypergraph
- `s` (*positive integer*) – 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*, *switch_names=True*)

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

Parameters

- `name` (*hashable, optional*) –
- `switch_names` (*bool, optional, default = True*) – reverses edge_col and node_col names unless edge_col = 'edges' and node_col = 'nodes'

Return type
hypergraph

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

The s-adjacency matrix for the dual hypergraph.

Parameters

- **s** (*int, optional, default 1*) –
- **index** (*boolean, optional, default = False*) – if True, will return the index of ids for rows and columns

Returns

- **edge_adjacency_matrix** (*scipy.sparse.csr.csr_matrix*)
- **edge_index** (*list*) – index of ids for rows and columns

Notes

This is also the adjacency matrix for the line graph. Two edges are s-adjacent if they share at least s nodes. If remove_zeros is True will return the auxillary matrix

`edge_diameter`(*s=1*)

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

Parameters
s (*int, optional, default 1*) –

Returns

- **maximum diameter** (*int*)
- **list of diameters** (*list*) – List of edge_diameters for s-edge component subgraphs in hypergraph
- **list of component** (*list*) – List of the edge uids in the s-edge component subgraphs.

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

XX TODO: still need to return path and translate into user defined nodes and edges Returns the shortest s-walk distance between two edges in the hypergraph.

Parameters

- **source** (*edge.uid or edge*) – an edge in the hypergraph
- **target** (*edge.uid or edge*) – an edge in the hypergraph
- **s** (*positive integer*) – the number of intersections between pairwise consecutive edges
- **TODO** (*add edge weights*) –
- **weight** (*None or string, optional, default = None*) – if None then all edges have weight 1. If string then edge attribute string is used if available.

Returns

s- walk 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

the shortest s-walk edge distance

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 or* EntitySet) – uid for a edge in hypergraph or the edge Entity
- **s** (*int, list, optional, default = 1*) – Minimum number of nodes shared by neighbors edge node.

Returns

List of edge neighbors

Return type

list

property edge_props

Dataframe of edge properties indexed on edge ids

Return type

pd.DataFrame

edge_size_dist()

Returns the size for each edge

Return type

np.array

property edges

Object associated with self._edges.

Return type
EntitySet

classmethod from_bipartite(*B*, *set_names=('edges', 'nodes')*, *name=None*, ***kwargs*)

Static method creates a Hypergraph from a bipartite graph.

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.
- **set_names** (*iterable of length 2, optional, default = ['edges','nodes']*) – Category names assigned to the graph nodes associated to each bipartite set
- **name** (*hashable, optional*) –

Return type
Hypergraph

Notes

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

```
>>> 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)
>>> H.nodes, H.edges
# output: (EntitySet(_:Nodes,[1, 2, 3, 4],{}), /
# EntitySet(_:Edges,['b', 'c', 'a'],{}))
```

classmethod from_incidence_dataframe(*df*, *columns=None*, *rows=None*, *edge_col: str = 'edges'*, *node_col: str = 'nodes'*, *name=None*, *fillna=0*, *transpose=False*, *transforms=[ ]*, *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
- **columns** (*(optional) list, default = None*) – restricts df to the columns with headers in this list.
- **rows** (*(optional) list, default = None*) – restricts df to the rows indexed by the elements in this list.
- **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.

- **transpose** (*(optional) bool, default = False*) – option to transpose the dataframe, in this case df.Index will identify the edges and df.columns will identify the nodes, transpose is applied before transforms and key
- **transforms** (*(optional) list, default = []*) – optional list of transformations to apply to each column, of the dataframe using pd.DataFrame.apply(). Transformations are applied in the order they are given (ex. abs). To apply transforms to rows or for additional functionality, consider transforming df using pandas.DataFrame methods prior to generating the hypergraph.
- **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

classmethod from_incidence_matrix(*M*, *node_names=None*, *edge_names=None*, *node_label='nodes'*, *edge_label='edges'*, *name=None*, *key=None*, ***kwargs*)

Same as from_numpy_array.

classmethod from_numpy_array(*M*, *node_names=None*, *edge_names=None*, *node_label='nodes'*, *edge_label='edges'*, *name=None*, *key=None*, ***kwargs*)

Create a hypergraph from a real valued matrix represented as a 2 dimensionsl 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: str*, *node: str*, *prop_name: str | None = None*) → Any | dict[str, Any]

Get cell properties on a specified edge and node

Parameters

- **edge** (*str*) – edgeid
- **node** (*str*) – nodeid
- **prop_name** (*str, optional*) – 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

int or str or dict of {str: any}

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

Creates an ::term::s-linegraph for the Hypergraph. If edges=True (default)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(*id*, *level=None*, *prop_name=None*)

Returns an object's specific property or all properties

Parameters

- **id** (*hashable*) – edge or node id
- **level** (*int | None , optional, default = None*) – if separate edge and node properties then 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

str or dict

incidence_dataframe(*sort_rows=False*, *sort_columns=False*, *cell_weights=True*)

Returns a pandas dataframe for hypergraph indexed by the nodes and with column headers given by the edge names.

Parameters

- **sort_rows** (*bool, optional, default =True*) – sort rows based on hashable node names

- **sort_columns** (*bool, optional, default =True*) – sort columns based on hashable edge names
- **cell_weights** (*bool, optional, default =True*) –

property incidence_dict

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

Return type
dict

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

An incidence matrix for the hypergraph indexed by nodes x edges.

Parameters

- **weights** (*bool, default =False*) – If False all nonzero entries are 1. If True and self.static all nonzero entries are filled by self.edges.cell_weights dictionary values.
- **index** (*boolean, optional, default = False*) – If True return will include a dictionary of node uid : row number and edge uid : column number

Returns

- **incidence_matrix** (*scipy.sparse.csr.csr_matrix or np.ndarray*)
- **row_index** (*list*) – index of node ids for rows
- **col_index** (*list*) – index of edge ids for columns

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 v0,vn there exists a sequence of nodes v0,v1,v2,…,v(n-1),vn 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 e0,en there exists a sequence of edges e0,e1,e2,…,e(n-1),en 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 or EntitySet*) – uid for a node in hypergraph or the node Entity
- **s** (*int, list, 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 hypergraph.

Parameters

- **and** (*list of the diameters of the s-components*) –
- **nodes** (*list of the s-component*) –

`property node_props`

Dataframe of node properties indexed on node ids

Return type
pd.DataFrame

`property nodes`

Object associated with self._nodes.

Return type
EntitySet

`number_of_edges`(*edgeset=None*)

The number of edges in edgeset belonging to hypergraph.

Parameters
edgeset (*an iterable of Entities, optional, default = None*) – If None, then return the number of edges in hypergraph.

Returns
number_of_edges

Return type
int

`number_of_nodes`(*nodeset=None*)

The number of nodes in nodeset belonging to hypergraph.

Parameters
nodeset (*an interable of Entities, optional, default = None*) – If None, then return the number of nodes in hypergraph.

Returns
number_of_nodes

Return type
int

`order`()

The number of nodes in hypergraph.

Returns
order

Return type
int

property properties

Returns dataframe of edge and node properties.

Return type
pd.DataFrame

remove(*keys*, *level=None*, *name=None*)

Creates a new hypergraph with nodes and/or edges indexed by keys removed. More efficient for creating a restricted hypergraph if the restricted set is greater than what is being removed.

Parameters

- **keys** (*list | tuple | set | Hashable*) – node and/or edge id(s) to restrict to
- **level** (*None, optional*) – Enter 0 to remove edges with ids in keys. Enter 1 to remove nodes with ids in keys. If None then all objects in nodes and edges with the id will be removed.
- **name** (*str, optional*) – Name of new hypergraph

Return type
hnx.Hypergraph

remove_edges(*keys*, *name=None*)

remove_nodes(*keys*, *name=None*)

remove_singletons(*name=None*)

Constructs clone of hypergraph with singleton edges removed.

Returns
new hypergraph

Return type
Hypergraph

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

New hypergraph gotten by restricting to edges

Parameters
edges (*Iterable*) – edgeids to restrict to

Return type
hnx.Hypergraph

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

New hypergraph gotten by restricting to nodes

Parameters
nodes (*Iterable*) – nodeids to restrict to

Return type
hnx. 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, edges=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*) –

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 components or the s-node-connected components of the hypergraph.

Parameters

- **s** (*int, optional, default 1*) –
- **edges** (*boolean, optional, default = True*) – If True will return edge components, if False will return node components
- **return_singletons** (*bool, optional, default = False*) –

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

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

```
>>> list(H.s_components(edges=True))
[{'C', 'D'}, {'A', 'B'}]
>>> list(H.s_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` – key=value pairs to save in state dictionary

property shape

(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

toplexes(*name=None*)

Returns a *simple hypergraph* corresponding to self.

> **Warning:** Collapsing is no longer supported inside the toplexes method. Instead generate a new collapsed hypergraph and compute the toplexes of the new hypergraph.

Parameters
name (*str, optional, default = None*) –

Module contents

class classes.**EntitySet**(*entity: DataFrame | Mapping[T, Iterable[T]] | Iterable[Iterable[T]] | Mapping[T, Mapping[T, Any]] | None = None, data_cols: Sequence[T] = (0, 1), data: ndarray | None = None, static: bool = True, labels: OrderedDict[T, Sequence[T]] | None = None, uid: Hashable | None = None, weight_col: str | int | None = 'cell_weights', weights: Sequence[float] | float | int | str | None = 1, aggregateby: str | dict | None = 'sum', properties: DataFrame | dict[int, dict[T, dict[Any, Any]]] | None = None, misc_props_col: str | None = None, level_col: str = 'level', id_col: str = 'id', cell_properties: Sequence[T] | DataFrame | dict[T, dict[T, dict[Any, Any]]] | None = None, misc_cell_props_col: str | None = None*)

Bases: `object`

Base class for handling N-dimensional data when building network-like models, i.e., `Hypergraph`

Parameters

- **entity** (*pandas.DataFrame, dict of lists or sets, dict of dicts, list of lists or sets, optional*) – If a DataFrame with N columns, represents N-dimensional entity data (data table). Otherwise, represents 2-dimensional entity data (system of sets).
- **data_cols** (*sequence of ints or strings, default=(0,1)*) –
- **level1** (*str or int, default = 0*) –
- **level2** (*str or int, default = 1*) –
- **data** (*numpy.ndarray, optional*) – 2D M x N ndarray of ints (data table); sparse representation of an N-dimensional incidence tensor with M nonzero cells. Ignored if *entity* is provided.
- **static** (*bool, default=True*) – If True, entity data may not be altered, and the state_dict will never be cleared. Otherwise, rows may be added to and removed from the data table, and updates will clear the state_dict.
- **labels** (*collections.OrderedDict of lists, optional*) – User-specified labels in corresponding order to ints in *data*. Ignored if *entity* is provided or *data* is not provided.
- **uid** (*hashable, optional*) – A unique identifier for the object
- **weight_col** (*string or int, default="cell_weights"*) –
- **weights** (*sequence of float, float, int, str, default=1*) – User-specified cell weights corresponding to entity data. If sequence of floats and *entity* or *data* defines a data table,

  length must equal the number of rows.

  If sequence of floats and *entity* defines a system of sets,
  length must equal the total sum of the sizes of all sets.

  If str and *entity* is a DataFrame,
  must be the name of a column in *entity*.

  Otherwise, weight for all cells is assumed to be 1.
- **aggregateby** (*{'sum', 'last', count', 'mean','median', max', 'min', 'first', None}, default="sum"*) – Name of function to use for aggregating cell weights of duplicate rows when *entity* or *data* defines a data table. If None, duplicate rows will be dropped without aggregating cell weights. Ignored if *entity* defines a system of sets.
- **properties** (*pandas.DataFrame or doubly-nested dict, optional*) – User-specified properties to be assigned to individual items in the data, i.e., cell entries in a data table; sets or set elements in a system of sets. See Notes for detailed explanation. If DataFrame, each row gives [optional item level, item label, optional named properties, {property name: property value}] (order of columns does not matter; see Notes for an example). If doubly-nested dict, {item level: {item label: {property name: property value}}}.
- **misc_props_col** (*str, default="properties"*) – Column names for miscellaneous properties, level index, and item name in properties; see Notes for explanation.
- **level_col** (*str, default="level"*) –
- **id_col** (*str, default="id"*) –

- **cell_properties** (*sequence of int or str, pandas.DataFrame, or doubly-nested dict, optional*) –
- **misc_cell_props_col** (*str, default="cell_properties"*) –

Notes

A property is a named attribute assigned to a single item in the data.

You can pass a **table of properties** to *properties* as a `DataFrame`:

Level (optional)	ID	[explicit property type]	[...]	misc. properties
0	level 0 item	property value	...	{property name: property value}
1	level 1 item	property value	...	{property name: property value}
...	...	...	...	...
N	level N item	property value	...	{property name: property value}

The Level column is optional. If not provided, properties will be assigned by ID (i.e., if an ID appears at multiple levels, the same properties will be assigned to all occurrences).

The names of the Level (if provided) and ID columns must be specified by *level_col* and *id_col*. *misc_props_col* can be used to specify the name of the column to be used for miscellaneous properties; if no column by that name is found, a new column will be created and populated with empty `dicts`. All other columns will be considered explicit property types. The order of the columns does not matter.

This method assumes that there are no rows with the same (Level, ID); if duplicates are found, all but the first occurrence will be dropped.

add(**args*) → Self

Updates the underlying data table with new entity data from multiple sources

Parameters
***args** – variable length argument list of Entity and/or representations of entity data

Returns
self

Return type
EntitySet

> **Warning:** Adding an element directly to an Entity will not add the element to any Hypergraphs constructed from that Entity, and will cause an error. Use `Hypergraph.add_edge` or `Hypergraph.add_node_to_edge` instead.

See also:

add_element
update from a single source

`Hypergraph.add_edge`, `Hypergraph.add_node_to_edge`

add_element(*data: DataFrame | Mapping[T, Iterable[T]] | Iterable[Iterable[T]] | Mapping[T, Mapping[T, Any]]*) → Self

Updates the underlying data table with new entity data

Supports adding from either an existing EntitySet or a representation of entity (data table or labeled system of sets are both supported representations)

Parameters
`data` (*pandas.DataFrame*, dict of lists or sets, lists of lists, or nested dict) –

Returns
self

Return type
EntitySet

> **Warning:** Adding an element directly to an Entity will not add the element to any Hypergraphs constructed from that Entity, and will cause an error. Use *Hypergraph.add_edge* or *Hypergraph.add_node_to_edge* instead.

See also:

add
takes multiple sources of new entity data as variable length argument list

`Hypergraph.add_edge`, `Hypergraph.add_node_to_edge`

add_elements_from(*arg_set*) → Self

Adds arguments from an iterable to the data table one at a time

DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]
Duplicates *add*

Parameters
`arg_set` (*iterable*) – list of Entity and/or representations of entity data

Returns
self

Return type
EntitySet

assign_cell_properties(*cell_props: DataFrame | dict[T, dict[T, dict[Any, Any]]], misc_col: str | None = None, replace: bool = False*) → None

Assign new properties to cells of the incidence matrix and update `properties`

Parameters

- `cell_props` (*pandas.DataFrame, dict of iterables, or doubly-nested dict, optional*) – See documentation of the *cell_properties* parameter in `EntitySet`
- `misc_col` (*str, optional*) – name of column to be used for miscellaneous cell property dicts
- `replace` (*bool, default=False*) – If True, replace existing `cell_properties` with result; otherwise update with new values from result

Raises
`AttributeError` – Not supported for :attr:`dimsize`=1

assign_properties(*props: DataFrame | dict[int, dict[T, dict[Any, Any]]], misc_col: str | None = None, level_col=0, id_col=1*) → None

Assign new properties to items in the data table, update `properties`

Parameters

- **props** (`pandas.DataFrame or doubly-nested dict`) – See documentation of the *properties* parameter in `EntitySet`
- **level_col** (`str, optional`) – column names corresponding to the levels, items, and misc. properties; if None, default to `_level_col`, `_id_col`, `_misc_props_col`, respectively.
- **id_col** (`str, optional`) – column names corresponding to the levels, items, and misc. properties; if None, default to `_level_col`, `_id_col`, `_misc_props_col`, respectively.
- **misc_col** (`str, optional`) – column names corresponding to the levels, items, and misc. properties; if None, default to `_level_col`, `_id_col`, `_misc_props_col`, respectively.

See also:

`properties`

property cell_properties: DataFrame | None

Properties assigned to cells of the incidence matrix

Returns
Returns None if `dimsize` < 2

Return type
pandas.DataFrame, optional

property cell_weights: dict[str, tuple[T]]

Cell weights corresponding to each row of the underlying data table

Returns
dict of {tuple – Keyed by row of data table (as a tuple)

Return type
int or float}

property children: set

Labels of all items in level 1 (second column) of the underlying data table

Return type
set

See also:

`uidset`
Labels of all items in level 0 (first column)

`uidset_by_level`, `uidset_by_column`

collapse_identical_elements(*return_equivalence_classes: bool = False, **kwargs*) → *EntitySet* | tuple[*hypernetx.classes.entityset.EntitySet*, dict[str, list[str]]]

Create a new `EntitySet` by collapsing sets with the same set elements

Each item in level 0 (first column) defines a set containing all the level 1 (second column) items with which it appears in the same row of the underlying data table.

Parameters

- **return_equivalence_classes** (*bool, default=False*) – If True, return a dictionary of equivalence classes keyed by new edge names
- **kwargs** – Extra arguments to `EntitySet` constructor

Returns

- **new_entity** (*EntitySet*) – new `EntitySet` with identical sets collapsed; if all sets are unique, the system of sets will be the same as the original.
- **equivalence_classes** (*dict of lists, optional*) – if *return_equivalence_classes`=True, ``{collapsed set label: [level 0 item labels]}`.*

property data: ndarray

Sparse representation of the data table as an incidence tensor

This can also be thought of as an encoding of *dataframe*, where items in each column of the data table are translated to their int position in the *self.labels[column]* list :returns: 2D array of ints representing rows of the underlying data table as indices in an incidence tensor :rtype: numpy.ndarray

See also:

labels, *dataframe*

property dataframe: DataFrame

The underlying data table stored by the Entity

Return type
pandas.DataFrame

property dimensions: tuple[int]

Dimensions of data i.e., the number of distinct items in each level (column) of the underlying data table

Returns
Length and order corresponds to columns of *self.dataframe* (excluding cell weight column)

Return type
tuple of ints

property dimsize: int

Number of levels (columns) in the underlying data table

Returns
Equal to length of *self.dimensions*

Return type
int

property elements: dict[Any, hypernetx.classes.helpers.AttrList]

System of sets representation of the first two levels (columns) of the underlying data table

Each item in level 0 (first column) defines a set containing all the level 1 (second column) items with which it appears in the same row of the underlying data table

Returns
System of sets representation as dict of {level 0 item : AttrList(level 1 items)}

Return type
dict of *AttrList*

See also:

incidence_dict
: same data as dict of list

memberships
: dual of this representation, i.e., each item in level 1 (second column) defines a set

elements_by_level, *elements_by_column*

elements_by_column(*col1: Hashable*, *col2: Hashable*) → dict[Any, hypernetx.classes.helpers.AttrList]

System of sets representation of two columns (levels) of the underlying data table

Each item in col1 defines a set containing all the col2 items with which it appears in the same row of the underlying data table

Properties can be accessed and assigned to items in col1

Parameters

- **col1** (*Hashable*) – name of column whose items define sets
- **col2** (*Hashable*) – name of column whose items are elements in the system of sets

Returns
System of sets representation as dict of {col1 item : AttrList(col2 items)}

Return type
dict of *AttrList*

See also:

elements, *memberships*

elements_by_level
: same functionality, takes level indices instead of column names

elements_by_level(*level1: int*, *level2: int*) → dict[Any, hypernetx.classes.helpers.AttrList]

System of sets representation of two levels (columns) of the underlying data table

Each item in level1 defines a set containing all the level2 items with which it appears in the same row of the underlying data table

Properties can be accessed and assigned to items in level1

Parameters

- **level1** (*int*) – index of level whose items define sets
- **level2** (*int*) – index of level whose items are elements in the system of sets

Returns
System of sets representation as dict of {level1 item : AttrList(level2 items)}

Return type
dict of *AttrList*

See also:

elements, *memberships*

elements_by_column
: same functionality, takes column names instead of level indices

property empty: bool

Whether the underlying data table is empty or not

Return type
bool

See also:

is_empty
for checking whether a specified level (column) is empty

dimsize
0 if empty

encode(*data: DataFrame*) → array

Encode dataframe to numpy array

Parameters
data (*dataframe, dataframe columns must have dtype set to 'category'*) –

Return type
numpy.array

get_cell_properties(*item1: T*, *item2: T*) → dict[Any, Any] | None

Get all properties of a cell, i.e., incidence between items of different levels

Parameters

- **item1** (*hashable*) – name of an item in level 0
- **item2** (*hashable*) – name of an item in level 1

Returns

- *dict* – `{named cell property: cell property value, ..., misc. cell property column name: {cell property name: cell property value}}`
- *None* – If properties do not exist

See also:

get_cell_property, *set_cell_property*

get_cell_property(*item1: T*, *item2: T*, *prop_name: Any*) → Any

Get a property of a cell i.e., incidence between items of different levels

Parameters

- **item1** (*hashable*) – name of an item in level 0
- **item2** (*hashable*) – name of an item in level 1
- **prop_name** (*hashable*) – name of the cell property to get

Returns

- **prop_val** (*any*) – value of the cell property
- *None* – If prop_name not found

Raises
KeyError – If *(item1, item2)* is not in *cell_properties*

See also:

get_cell_properties, *set_cell_property*

get_properties(*item: T*, *level: int | None = None*) → dict[Any, Any]

Get all properties of an item

Parameters

- **item** (*hashable*) – name of an item
- **level** (*int, optional*) – level index of the item

Returns

prop_vals – `{named property: property value, ..., misc. property column name: {property name: property value}}`

Return type

dict

Raises

KeyError – if (*level*, *item*) is not in *properties*, or if *level* is not provided and *item* is not in *properties*

Warns

UserWarning – If *level* is not provided and *item* appears in multiple levels, assumes the first (closest to 0)

See also:

get_property, *set_property*

get_property(*item: T*, *prop_name: Any*, *level: int | None = None*) → Any

Get a property of an item

Parameters

- **item** (*hashable*) – name of an item
- **prop_name** (*hashable*) – name of the property to get
- **level** (*int, optional*) – level index of the item

Returns

- **prop_val** (*any*) – value of the property
- *None* – if property not found

Raises

KeyError – if (*level*, *item*) is not in *properties*, or if *level* is not provided and *item* is not in *properties*

Warns

UserWarning – If *level* is not provided and *item* appears in multiple levels, assumes the first (closest to 0)

See also:

get_properties, *set_property*

property incidence_dict: dict[T, list[T]]

System of sets representation of the first two levels (columns) of the underlying data table

Returns

System of sets representation as dict of {level 0 item : AttrList(level 1 items)}

Return type

dict of list

See also:

elements
same data as dict of AttrList

incidence_matrix(*level1: int = 0*, *level2: int = 1*, *weights: bool | dict = False*, *aggregateby: str = 'count'*) → csr_matrix | None

Incidence matrix representation for two levels (columns) of the underlying data table

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

If *level1* and *level2* contain N and M distinct items, respectively, the incidence matrix will be M x N. In other words, the items in *level1* and *level2* correspond to the columns and rows of the incidence matrix, respectively, in the order in which they appear in *self.labels[column1]* and *self.labels[column2]* (*column1* and *column2* are the column labels of *level1* and *level2*)

Parameters

- **level1** (*int, default=0*) – index of first level (column)
- **level2** (*int, default=1*) – index of second level
- **weights** (*bool or dict, default=False*) – If False all nonzero entries are 1. If True all nonzero entries are filled by self.cell_weight dictionary values, use `aggregateby` to specify how duplicate entries should have weights aggregated. If dict of {(level1 item, level2 item): weight value} form; only nonzero cells in the incidence matrix will be updated by dictionary, i.e., *level1 item* and *level2 item* must appear in the same row at least once in the underlying data table
- **aggregateby** (*{'last', count', 'sum', 'mean','median', max', 'min', 'first', 'last', None}, default='count'*) –

  Method to aggregate weights of duplicate rows in data table.
  If None, then all cell weights will be set to 1.
- **index** (*bool, optional*) – Not used

Returns
sparse representation of incidence matrix (i.e. Compressed Sparse Row matrix)

Return type
scipy.sparse.csr.csr_matrix

Note: In the context of Hypergraphs, think *level1 = edges, level2 = nodes*

index(*column: str*, *value: str | None = None*) → int | tuple[int, int]

Get level index corresponding to a column and (optionally) the index of a value in that column

The index of `value` is its position in the list given by `self.labels[column]`, which is used in the integer encoding of the data table `self.data`

Parameters

- **column** (*str*) – name of a column in self.dataframe
- **value** (*str, optional*) – label of an item in the specified column

Returns
level index corresponding to column, index of value if provided

Return type
int or (int, int)

See also:

indices
for finding indices of multiple values in a column

level
same functionality, search for the value without specifying column

`indices`(*column: str*, *values: str | Iterable[str]*) → list[int]
Get indices of one or more value(s) in a column

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters

- **column** (*str*) –
- **values** (*str or iterable of str*) –

Returns
indices of values

Return type
list of int

See also:

index
for finding level index of a column and index of a single value

`is_empty`(*level: int = 0*) → bool
Whether a specified level (column) of the underlying data table is empty or not

Parameters
level (*int*) – the level of a column in the underlying data table

Return type
bool

See also:

empty
for checking whether the underlying data table is empty

size
number of items in a level (columns); 0 if level is empty

property isstatic: bool
Whether to treat the underlying data as static or not

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE] If True, the underlying data may not be altered, and the state_dict will never be cleared Otherwise, rows may be added to and removed from the data table, and updates will clear the state_dict

Return type
bool

property labels: dict[str, list]

Labels of all items in each column of the underlying data table

Returns

dict of {column name: [item labels]} The order of [item labels] corresponds to the int encoding of each item in *self.data*.

Return type

dict of lists

See also:

`data`, `dataframe`

level(*item: str*, *min_level: int = 0*, *max_level: int | None = None*, *return_index: bool = True*) → int | tuple[int, int] | None

First level containing the given item label

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Order of levels corresponds to order of columns in *self.dataframe*

Parameters

- **item** (*str*) –
- **min_level** (*int, default=0*) – minimum inclusive bound on range of levels to search for item
- **max_level** (*int, optional*) – maximum inclusive bound on range of levels to search for item
- **return_index** (*bool, default=True*) – If True, return index of item within the level

Returns

index of first level containing the item, index of item if *return_index=True* returns None if item is not found

Return type

int, (int, int), or None

See also:

`index`, `indices`

property memberships: dict[Any, hypernetx.classes.helpers.AttrList]

System of sets representation of the first two levels (columns) of the underlying data table

Each item in level 1 (second column) defines a set containing all the level 0 (first column) items with which it appears in the same row of the underlying data table

Returns

System of sets representation as dict of {level 1 item : AttrList(level 0 items)}

Return type

dict of *AttrList*

See also:

`elements`

dual of this representation i.e., each item in level 0 (first column) defines a set

`elements_by_level`, `elements_by_column`

property properties: DataFrame

Properties assigned to items in the underlying data table

Returns

pandas.DataFrame a dataframe with the following columns

Return type

level/(edge|node), uid, weight, properties

remove(**args: T*) → *EntitySet*

Removes all rows containing specified item(s) from the underlying data table

Parameters

***args** – variable length argument list of items which are of type string or int

Returns

self

Return type

EntitySet

See also:

remove_element

remove all rows containing a single specified item

remove_element(*item: T*) → None

Removes all rows containing a specified item from the underlying data table

Parameters

item (*Union[str, int]*) – the label of an edge

See also:

remove

same functionality, accepts variable length argument list of item labels

remove_elements_from(*arg_set*)

Removes all rows containing specified item(s) from the underlying data table

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Duplicates *remove*

Parameters

arg_set (*iterable*) – list of item labels

Returns

self

Return type

EntitySet

restrict_to(*indices: int | Iterable[int], **kwargs*) → *EntitySet*

Alias of `restrict_to_indices()` with default parameter `level`=0

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters

- **indices** (*array_like of int*) – indices of item label(s) in *level* to restrict to

- **`**kwargs`** – Extra arguments to `EntitySet` constructor

Return type
EntitySet

See also:

`restrict_to_indices`

`restrict_to_indices`(*indices: int | Iterable[int]*, *level: int = 0*, ***kwargs*) → *EntitySet*

Create a new EntitySet by restricting the data table to rows containing specific items in a given level

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters

- **`indices`** (*`int or iterable of int`*) – indices of item label(s) in *level* to restrict to
- **`level`** (*`int, default=0`*) – level index
- **`**kwargs`** – Extra arguments to *EntitySet* constructor

Return type
EntitySet

`restrict_to_levels`(*levels: int | Iterable[int]*, *weights: bool = False*, *aggregateby: str | None = 'sum'*, *keep_memberships: bool = True*, ***kwargs*) → *EntitySet*

Create a new EntitySet by restricting to a subset of levels (columns) in the underlying data table

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters

- **`levels`** (*`array-like of int`*) – indices of a subset of levels (columns) of data
- **`weights`** (*`bool, default=False`*) – If True, aggregate existing cell weights to get new cell weights. Otherwise, all new cell weights will be 1.
- **`aggregateby`** (*`{'sum', 'first', 'last', 'count', 'mean', 'median', 'max', 'min', None}, optional`*) – Method to aggregate weights of duplicate rows in data table If None or `weights`=False then all new cell weights will be 1
- **`keep_memberships`** (*`bool, default=True`*) – Whether to preserve membership information for the discarded level when the new `EntitySet` is restricted to a single level
- **`**kwargs`** – Extra arguments to `EntitySet` constructor

Return type
EntitySet

Raises
`KeyError` – If *levels* contains any invalid values

`set_cell_property`(*item1: T*, *item2: T*, *prop_name: Any*, *prop_val: Any*) → None

Set a property of a cell i.e., incidence between items of different levels

Parameters

- **`item1`** (*`hashable`*) – name of an item in level 0
- **`item2`** (*`hashable`*) – name of an item in level 1
- **`prop_name`** (*`hashable`*) – name of the cell property to set
- **`prop_val`** (*`any`*) – value of the cell property to set

See also:

get_cell_property, *get_cell_properties*

set_property(*item: T*, *prop_name: Any*, *prop_val: Any*, *level: int | None = None*) → None

Set a property of an item

Parameters

- **item** (*hashable*) – name of an item
- **prop_name** (*hashable*) – name of the property to set
- **prop_val** (*any*) – value of the property to set
- **level** (*int, optional*) – level index of the item; required if *item* is not already in *properties*

Raises

ValueError – If *level* is not provided and *item* is not in *properties*

Warns

UserWarning – If *level* is not provided and *item* appears in multiple levels, assumes the first (closest to 0)

See also:

get_property, *get_properties*

size(*level: int = 0*) → int

The number of items in a level of the underlying data table

Equivalent to `self.dimensions[level]`

Parameters

level (*int, default=0*) –

Return type

int

See also:

dimensions

translate(*level: int*, *index: int | list[int]*) → str | list[str]

Given indices of a level and value(s), return the corresponding value label(s)

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters

- **level** (*int*) – the index of the level
- **index** (*int or list of int*) – value index or indices

Returns

label(s) corresponding to value index or indices

Return type

str or list of str

See also:

translate_arr

translate a full row of value indices across all levels (columns)

translate_arr(*coords: tuple[int, int]*) → list[str]

Translate a full encoded row of the data table e.g., a row of `self.data`

[DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Parameters
coords (*tuple of ints*) – encoded value indices, with one value index for each level of the data

Returns
full row of translated value labels

Return type
list of str

property uid: Hashable

User-defined unique identifier for the *Entity*

Return type
Hashable

property uidset: set

Labels of all items in level 0 (first column) of the underlying data table

Return type
set

See also:

children
Labels of all items in level 1 (second column)

uidset_by_level, *uidset_by_column*

uidset_by_column(*column: Hashable*) → set

Labels of all items in a particular column (level) of the underlying data table

Parameters
column (*Hashable*) – Name of a column in *self.dataframe*

Return type
set

See also:

uidset
Labels of all items in level 0 (first column)

children
Labels of all items in level 1 (second column)

uidset_by_level
Same functionality, takes the level index instead of column name

uidset_by_level(*level: int*) → set

Labels of all items in a particular level (column) of the underlying data table

Parameters
level (*int*) –

Return type
set

See also:

`uidset`
Labels of all items in level 0 (first column)

`children`
Labels of all items in level 1 (second column)

`uidset_by_column`
Same functionality, takes the column name instead of level index

class classes.**Hypergraph**(*setsystem: DataFrame | ndarray | Mapping[T, Iterable[T]] | Iterable[Iterable[T]] | Mapping[T, Mapping[T, Mapping[str, Any]]] | None = None, edge_col: str | int = 0, node_col: str | int = 1, cell_weight_col: str | int | None = 'cell_weights', cell_weights: Sequence[float] | float = 1.0, cell_properties: Sequence[str | int] | Mapping[T, Mapping[T, Mapping[str, Any]]] | None = None, misc_cell_properties_col: str | int | None = None, aggregateby: str | dict[str, str] = 'first', edge_properties: DataFrame | dict[T, dict[Any, Any]] | None = None, node_properties: DataFrame | dict[T, dict[Any, Any]] | None = None, properties: DataFrame | dict[T, dict[Any, Any]] | dict[T, dict[T, dict[Any, Any]]] | None = None, misc_properties_col: str | int | None = None, edge_weight_prop_col: str | int = 'weight', node_weight_prop_col: str | int = 'weight', weight_prop_col: str | int = 'weight', default_edge_weight: float | None = None, default_node_weight: float | None = None, default_weight: float = 1.0, name: str | None = None, **kwargs*)

Bases: `object`

Parameters

- **setsystem** (*(optional) dict of iterables, dict of dicts,iterable of iterables,*) – pandas.DataFrame, numpy.ndarray, default = None See SetSystem above for additional setsystem requirements.
- **edge_col** (*(optional) str | int, default = 0*) – column index (or name) in pandas.dataframe or numpy.ndarray, used for (hyper)edge ids. Will be used to reference edgeids for all set systems.
- **node_col** (*(optional) str | int, default = 1*) – column index (or name) in pandas.dataframe or numpy.ndarray, used for node ids. Will be used to reference nodeids for all set systems.
- **cell_weight_col** (*(optional) str | int, default = None*) – column index (or name) in pandas.dataframe or numpy.ndarray used for referencing cell weights. For a dict of dicts references key in cell property dicts.
- **cell_weights** (*(optional) Sequence[float,int] | int | float , default = 1.0*) – User specified cell_weights or default cell weight. Sequential values are only used if setsystem is a dataframe or ndarray in which case the sequence must have the same length and order as these objects. Sequential values are ignored for dataframes if cell_weight_col is already a column in the data frame. If cell_weights is assigned a single value then it will be used as default for missing values or when no cell_weight_col is given.
- **cell_properties** (*(optional) Sequence[int | str] | Mapping[T, Mapping[T,Mapping[str,Any]]],*) – default = None Column names from pd.DataFrame to use as cell properties or a dict assigning cell_property to incidence pairs of edges and nodes. Will generate a misc_cell_properties, which may have variable lengths per cell.

- **misc_cell_properties** (*(optional) str | int, default = None*) – Column name of dataframe corresponding to a column of variable length property dictionaries for the cell. Ignored for other setsystem types.
- **aggregateby** (*(optional) str, dict, default = 'first'*) – By default duplicate edge,node incidences will be dropped unless specified with *aggregateby*. See pandas.DataFrame.agg() methods for additional syntax and usage information.
- **edge_properties** (*(optional) pd.DataFrame | dict, default = None*) – Properties associated with edge ids. First column of dataframe or keys of dict link to edge ids in setsystem.
- **node_properties** (*(optional) pd.DataFrame | dict, default = None*) – Properties associated with node ids. First column of dataframe or keys of dict link to node ids in setsystem.
- **properties** (*(optional) pd.DataFrame | dict, 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 and different properties then use the edge_properties and node_properties keywords.
- **misc_properties** (*(optional) int | str, default = None*) – Column of property dataframes with dtype=dict. Intended for variable length property dictionaries for the objects.
- **edge_weight_prop** (*(optional) str, default = None,*) – Name of property in edge_properties to use for weight.
- **node_weight_prop** (*(optional) str, default = None,*) – Name of property in node_properties to use for weight.
- **weight_prop** (*(optional) str, default = None*) – Name of property in properties to use for 'weight'
- **default_edge_weight** (*(optional) int | float, default = 1*) – Used when edge weight property is missing or undefined.
- **default_node_weight** (*(optional) int | float, default = 1*) – Used when node weight property is missing or undefined
- **name** (*(optional) str, default = None*) – Name assigned to hypergraph

Hypergraphs in HNX 2.0

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 identifiers instead of their contents. For example, if V = {1,2,3} and E = {e1,e2,e3}, where e1 = {1,2}, e2 = {1,2}, and e3 = {1,2,3}, the edges e1 and e2 contain the same set of nodes and yet are distinct and are distinguishable within H = (V,E).

New as of version 2.0, 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.

SetSystems

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

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

```
>>> H = Hypergraph([{1,2},{1,2},{1,2,3}])
```

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):

```
>>> H = Hypergraph({'e1':[1,2],'e2':[1,2],'e3':[1,2,3]})
```

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:

```
>>> d = {'e1':{ 1: {'w':0.5, 'name': 'related_to'},
>>>             2: {'w':0.1, 'name': 'related_to',
>>>                 'startdate': '05.13.2020'}},
>>>      'e2':{ 1: {'w':0.52, 'name': 'owned_by'},
>>>             2: {'w':0.2}},
>>>      'e3':{ 1: {'w':0.5, 'name': 'related_to'},
>>>             2: {'w':0.2, 'name': 'owner_of'},
>>>             3: {'w':1, 'type': 'relationship'}}
```

```
>>> H = Hypergraph(d, cell_weight_col='w')
```

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:

col1	col2	w	col3
e1	1	0.5	{'name':'related_to'}
e1	2	0.1	**{"name":"related_to",** "start-date":"05.13.2020"}
e2	1	0.52	{"name":"owned_by"}
e2	2	0.2	
…	…	…	{…}

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

```
>>> H = Hypergraph(df,edge_col="col1",node_col="col2",
>>>                 cell_weight_col="w",misc_cell_properties="col3")
```

5. **numpy.ndarray** For homogeneous datasets given in an ndarray a pandas dataframe is generated and column names are added from the edge_col and node_col arguments. Cell properties containing multiple data types are added with a separate dataframe or dict and passed through the cell_properties keyword.

```
>>> arr = np.array([['e1','1'],['e1','2'],
>>>                 ['e2','1'],['e2','2'],
>>>                 ['e3','1'],['e3','2'],['e3','3']])
>>> H = hnx.Hypergraph(arr, column_names=['col1','col2'])
```

Edge and Node Properties

Properties specific to a single edge or node are 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 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:

id	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:

```
dp = {id1 : {prop1:val1, prop2,val2,...}, id2 : ... }
```

A properties dataframe may be used for nodes and edges sharing ids but differing in cell properties by adding a level index using 0 for edges and 1 for nodes:

level	id	weight	properties
0	e1	5.0	{'type':'event'}
0	e2	0.52	{"name":"owned_by"}
…	…	…	{…}
1	1.2	{'color':'red'}	
2	.003	{'name':'Fido','color':'brown'}	
…	…	…	{…}

Weights

The default key for cell and object weights is "weight". The default value is 1. Weights may be assigned and/or a new default prescribed in the constructor using **cell_weight_col** and **cell_weights** for incidence pairs, and using **edge_weight_prop, node_weight_prop, weight_prop, default_edge_weight,** and **default_node_weight** for node and edge weights.

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

The s-adjacency matrix for the hypergraph.

Parameters

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

- **index** (*boolean, optional, default = False*) – if True, will return the index of ids for rows and columns
- **remove_empty_rows** (*boolean, optional, default = False*) –

Returns

- **adjacency_matrix** (*scipy.sparse.csr.csr_matrix*)
- **node_index** (*list*) – index of ids for rows and columns

`auxiliary_matrix`(*s=1*, *node=True*, *index=False*)

The unweighted s-edge or node auxiliary matrix for hypergraph

Parameters

- **s** (*int, optional, default = 1*) –
- **node** (*bool, optional, default = True*) – whether to return based on node or edge adjacencies

Returns

- **auxiliary_matrix** (*scipy.sparse.csr.csr_matrix*) – Node/Edge adjacency matrix with empty rows and columns removed
- **index** (*np.array*) – row and column index of userids

`bipartite`()

Constructs the networkX bipartite graph associated to hypergraph.

Returns
bipartite

Return type
nx.Graph()

Notes

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 n in e there is an edge (n,e) in the graph.

`collapse_edges`(*name=None*, *return_equivalence_classes=False*, *use_reps=None*, *return_counts=None*)

Constructs a new hypergraph gotten by identifying edges containing the same nodes

Parameters

- **name** (*hashable, optional, default = None*) –
- **return_equivalence_classes** (*boolean, optional, default = False*) – Returns a dictionary of edge equivalence classes keyed by frozen sets of nodes

Returns

- **new hypergraph** (*Hypergraph*) – Equivalent edges are collapsed to a single edge named by a representative of the equivalent edges followed by a colon and the number of edges it represents.
- **equivalence_classes** (*dict*) – A dictionary keyed by representative edge names with values equal to the edges in its equivalence class

Notes

Two edges are identified 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 edge from the collapsed edges followed by a colon and the number of elements in its equivalence class as uid for the new edge

collapse_nodes(*name=None*, *return_equivalence_classes=False*, *use_reps=None*, *return_counts=None*) → *Hypergraph*

Constructs a new hypergraph gotten by identifying nodes contained by the same edges

Parameters

- **name** (*str, optional, default = None*) –
- **return_equivalence_classes** (*boolean, optional, default = False*) – Returns a dictionary of node equivalence classes keyed by frozen sets of edges
- **use_reps** (*boolean, optional, default = None*) – [DEPRECATED; WILL BE REMOVED IN NEXT RELEASE] Choose a single element from the collapsed nodes as uid for the new node, otherwise uses a frozen set of the uids of nodes in the equivalence class. If use_reps is True the new nodes have uids given by a tuple of the rep and the count
- **return_counts** (*boolean, optional, default = None*) – [DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Returns
new hypergraph

Return type
Hypergraph

Notes

Two nodes are identified 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 followed by the number of members of the class.

Example

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

collapse_nodes_and_edges(*name=None*, *return_equivalence_classes=False*, *use_reps=None*, *return_counts=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 frozen sets of nodes

- **use_reps** (*boolean, optional, default = None*) – [DEPRECATED; WILL BE REMOVED IN NEXT RELEASE] Choose a single element from the collapsed elements as a representative. If use_reps is True, the new elements are keyed by a tuple of the rep and the count.
- **return_counts** (*boolean, optional, default = None*) – [DEPRECATED; WILL BE REMOVED IN NEXT RELEASE]

Returns
new hypergraph

Return type
Hypergraph

Notes

Collapses the Nodes and Edges of EntitySets. 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 followed by the number of members of the class.

Example

```
>>> 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(*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 pairs and their properties.

Return type

pd.DataFrame

degree(*node*, *s=1*, *max_size=None*)

The number of edges of size s that contain node.

Parameters

- **node** (*hashable*) – identifier for the node.
- **s** (*positive integer, optional, default 1*) – smallest size of edge to consider in degree
- **max_size** (*positive integer or None, optional, default = None*) – largest size of edge to consider in degree

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_start and v_end are s-walk connected if there is a sequence of nodes v_start, v_1, v_2, … v_n-1, v_end such that consecutive nodes are s-adjacent. If the graph is not connected, an error will be raised.

dim(*edge*)

Same as size(edge)-1.

distance(*source*, *target*, *s=1*)

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

Parameters

- **source** (*node.uid or node*) – a node in the hypergraph
- **target** (*node.uid or node*) – a node in the hypergraph
- **s** (*positive integer*) – 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*, *switch_names=True*)

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

Parameters

- **name** (*hashable, optional*) –
- **switch_names** (*bool, optional, default = True*) – reverses edge_col and node_col names unless edge_col = 'edges' and node_col = 'nodes'

Return type

hypergraph

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

The s-adjacency matrix for the dual hypergraph.

Parameters

- **s** (*int, optional, default 1*) –
- **index** (*boolean, optional, default = False*) – if True, will return the index of ids for rows and columns

Returns

- **edge_adjacency_matrix** (*scipy.sparse.csr.csr_matrix*)
- **edge_index** (*list*) – index of ids for rows and columns

Notes

This is also the adjacency matrix for the line graph. Two edges are s-adjacent if they share at least s nodes. If remove_zeros is True will return the auxillary matrix

edge_diameter(*s=1*)

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

Parameters

`s` (*int, optional, default 1*) –

Returns

- **maximum diameter** (*int*)
- **list of diameters** (*list*) – List of edge_diameters for s-edge component subgraphs in hypergraph
- **list of component** (*list*) – List of the edge uids in the s-edge component subgraphs.

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

XX TODO: still need to return path and translate into user defined nodes and edges Returns the shortest s-walk distance between two edges in the hypergraph.

Parameters

- **source** (*edge.uid or edge*) – an edge in the hypergraph
- **target** (*edge.uid or edge*) – an edge in the hypergraph
- **s** (*positive integer*) – the number of intersections between pairwise consecutive edges
- **TODO** (*add edge weights*) –
- **weight** (*None or string, optional, default = None*) – if None then all edges have weight 1. If string then edge attribute string is used if available.

Returns

s- walk 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

the shortest s-walk edge distance

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 or EntitySet*) – uid for a edge in hypergraph or the edge Entity

- **s** (*int, list, optional, default = 1*) – Minimum number of nodes shared by neighbors edge node.

Returns
List of edge neighbors

Return type
list

property edge_props

Dataframe of edge properties indexed on edge ids

Return type
pd.DataFrame

edge_size_dist()

Returns the size for each edge

Return type
np.array

property edges

Object associated with self._edges.

Return type
EntitySet

classmethod from_bipartite(*B*, *set_names=('edges', 'nodes')*, *name=None*, ***kwargs*)

Static method creates a Hypergraph from a bipartite graph.

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.
- **set_names** (*iterable of length 2, optional, default = ['edges','nodes']*) – Category names assigned to the graph nodes associated to each bipartite set
- **name** (*hashable, optional*) –

Return type
Hypergraph

Notes

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

```
>>> 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)
>>> H.nodes, H.edges
# output: (EntitySet(_:Nodes,[1, 2, 3, 4],{}), /
# EntitySet(_:Edges,['b', 'c', 'a'],{}))
```

classmethod from_incidence_dataframe(*df*, *columns=None*, *rows=None*, *edge_col: str = 'edges'*, *node_col: str = 'nodes'*, *name=None*, *fillna=0*, *transpose=False*, *transforms=[ ]*, *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
- **columns** (*(optional) list, default = None*) – restricts df to the columns with headers in this list.
- **rows** (*(optional) list, default = None*) – restricts df to the rows indexed by the elements in this list.
- **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.
- **transpose** (*(optional) bool, default = False*) – option to transpose the dataframe, in this case df.Index will identify the edges and df.columns will identify the nodes, transpose is applied before transforms and key
- **transforms** (*(optional) list, default = []*) – optional list of transformations to apply to each column, of the dataframe using pd.DataFrame.apply(). Transformations are applied in the order they are given (ex. abs). To apply transforms to rows or for additional functionality, consider transforming df using pandas.DataFrame methods prior to generating the hypergraph.
- **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

classmethod from_incidence_matrix(*M*, *node_names=None*, *edge_names=None*, *node_label='nodes'*, *edge_label='edges'*, *name=None*, *key=None*, ***kwargs*)

Same as from_numpy_array.

classmethod from_numpy_array(*M*, *node_names=None*, *edge_names=None*, *node_label='nodes'*, *edge_label='edges'*, *name=None*, *key=None*, ***kwargs*)

Create a hypergraph from a real valued matrix represented as a 2 dimensionsl 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: str*, *node: str*, *prop_name: str | None = None*) → Any | dict[str, Any]

Get cell properties on a specified edge and node

Parameters

- **edge** (*str*) – edgeid
- **node** (*str*) – nodeid
- **prop_name** (*str, optional*) – 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
int or str or dict of {str: any}

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

Creates an ::term::s-linegraph for the Hypergraph. If edges=True (default)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`(*id*, *level=None*, *prop_name=None*)

Returns an object's specific property or all properties

Parameters

- **id** (*hashable*) – edge or node id

- **level** (*int | None , optional, default = None*) – if separate edge and node properties then 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
str or dict

`incidence_dataframe`(*sort_rows=False*, *sort_columns=False*, *cell_weights=True*)

Returns a pandas dataframe for hypergraph indexed by the nodes and with column headers given by the edge names.

Parameters

- **sort_rows** (*bool, optional, default =True*) – sort rows based on hashable node names
- **sort_columns** (*bool, optional, default =True*) – sort columns based on hashable edge names
- **cell_weights** (*bool, optional, default =True*) –

`property incidence_dict`

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

Return type
dict

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

An incidence matrix for the hypergraph indexed by nodes x edges.

Parameters

- **weights** (*bool, default =False*) – If False all nonzero entries are 1. If True and self.static all nonzero entries are filled by self.edges.cell_weights dictionary values.
- **index** (*boolean, optional, default = False*) – If True return will include a dictionary of node uid : row number and edge uid : column number

Returns

- **incidence_matrix** (*scipy.sparse.csr.csr_matrix or np.ndarray*)
- **row_index** (*list*) – index of node ids for rows
- **col_index** (*list*) – index of edge ids for columns

`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 v0,vn there exists a sequence of nodes v0,v1,v2,…,v(n-1),vn 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 e0,en there exists a sequence of edges e0,e1,e2,…,e(n-1),en 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 or* EntitySet) – uid for a node in hypergraph or the node Entity
- **s** (*int, list, 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 hypergraph.

Parameters

- **and** (*list of the diameters of the s-components*) –
- **nodes** (*list of the s-component*) –

`property node_props`

Dataframe of node properties indexed on node ids

Return type
pd.DataFrame

`property nodes`

Object associated with self._nodes.

Return type
EntitySet

`number_of_edges`(*edgeset=None*)

The number of edges in edgeset belonging to hypergraph.

Parameters
edgeset (*an iterable of Entities, optional, default = None*) – If None, then return the number of edges in hypergraph.

Returns
number_of_edges

Return type
int

`number_of_nodes`(*nodeset=None*)

The number of nodes in nodeset belonging to hypergraph.

Parameters

`nodeset` (*an interable of Entities, optional, default = None*) – If None, then return the number of nodes in hypergraph.

Returns

number_of_nodes

Return type

int

`order`()

The number of nodes in hypergraph.

Returns

order

Return type

int

`property properties`

Returns dataframe of edge and node properties.

Return type

pd.DataFrame

`remove`(*keys*, *level=None*, *name=None*)

Creates a new hypergraph with nodes and/or edges indexed by keys removed. More efficient for creating a restricted hypergraph if the restricted set is greater than what is being removed.

Parameters

- `keys` (*list | tuple | set | Hashable*) – node and/or edge id(s) to restrict to
- `level` (*None, optional*) – Enter 0 to remove edges with ids in keys. Enter 1 to remove nodes with ids in keys. If None then all objects in nodes and edges with the id will be removed.
- `name` (*str, optional*) – Name of new hypergraph

Return type

hnx.Hypergraph

`remove_edges`(*keys*, *name=None*)

`remove_nodes`(*keys*, *name=None*)

`remove_singletons`(*name=None*)

Constructs clone of hypergraph with singleton edges removed.

Returns

new hypergraph

Return type

Hypergraph

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

New hypergraph gotten by restricting to edges

Parameters

`edges` (*Iterable*) – edgeids to restrict to

Return type
hnx.Hypergraph

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

New hypergraph gotten by restricting to nodes

Parameters
nodes (*Iterable*) – nodeids to restrict to

Return type
hnx. 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, edges=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*) –

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 components or the s-node-connected components of the hypergraph.

Parameters

- **s** (*int, optional, default 1*) –
- **edges** (*boolean, optional, default = True*) – If True will return edge components, if False will return node components
- **return_singletons** (*bool, optional, default = False*) –

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

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

```
>>> list(H.s_components(edges=True))
[{'C', 'D'}, {'A', 'B'}]
>>> list(H.s_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` – key=value pairs to save in state dictionary

`property shape`

(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

`toplexes(name=None)`

Returns a *simple hypergraph* corresponding to self.

> **Warning:** Collapsing is no longer supported inside the toplexes method. Instead generate a new collapsed hypergraph and compute the toplexes of the new hypergraph.

Parameters
name (`str, optional, default = None`) –

5.4.2 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*)

A continuous-time SIR model for hypergraphs similar to the model in "The effect of heterogeneity on hypergraph contagion models" by Landry and Restrepo https://doi.org/10.1063/5.0020034 and implemented for networks in the EoN package by Joel C. Miller https://epidemicsonnetworks.readthedocs.io/en/latest/

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:

```
>>> 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)
```

`algorithms.contagion.`**`Gillespie_SIS`**(*H*, *tau*, *gamma*, *transmission_function=<function threshold>*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=inf*, *return_full_data=False*, *sim_kwargs=None*, ***args*)

A continuous-time SIS model for hypergraphs similar to the model in "The effect of heterogeneity on hypergraph contagion models" by Landry and Restrepo https://doi.org/10.1063/5.0020034 and implemented for networks in the EoN package by Joel C. Miller https://epidemicsonnetworks.readthedocs.io/en/latest/

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:

```
>>> 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:

```
>>> 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.**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 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:

```
>>> 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)}
>>> fps = 1
>>> fig = plt.figure()
>>> 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:

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> 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*) –
- **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.
- **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:

```
>>> 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:

```
>>> 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:

```
>>> 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")
- **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:

```
>>> 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

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:

```
>>> 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:

```
>>> 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:

```
>>> 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 Z_2 additions. The boundary matrices are represented as rectangular matrices over 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.

Methods for obtaining SNF for Z/2Z are based on Ferrario's work: http://www.dlfer.xyz/post/2016-10-27-smith-normal-form/

`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*) –
- **i** (*int*) – index of row being altered
- **j** (*int*) – index of row being added to altered

Returns
N

Return type
np.array

`algorithms.homology_mod2.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.homology_mod2.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.homology_mod2.bkMatrix(km1basis, kbasis)`

Compute the boundary map from C_{k-1}-basis to C_k basis with respect to 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 Z_2 stored as boolean

Return type
np.array

`algorithms.homology_mod2.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.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 than 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, H_k, defined by a chain complex C 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 kchains 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 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.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 Z/2Z

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 Z/2Z

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 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: $L M R = S$ and $LLinv = I_m$.

`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** (Hypergraph) – The hypergraph
- **A** (*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 (*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** (Hypergraph) –
- **delta** (*float, optional*) – convergence stopping criterion

Returns
A partition of the vertices in HG

Return type
list of sets

`algorithms.hypergraph_modularity.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[2]

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

[1] Kumar T., Vaidyanathan S., Ananthapadmanabhan H., Parthasarathy S. and Ravindran B. "A New Measure of Modularity in Hypergraphs: Theoretical Insights and Implications for Effective Clustering". In: Cherifi H., Gaito S., Mendes J., Moro E., Rocha L. (eds) Complex Networks and Their Applications VIII. COMPLEX NETWORKS 2019. Studies in Computational Intelligence, vol 881. Springer, Cham. https://doi.org/10.1007/978-3-030-36687-2_24

[2] Kamiński B., Prałat P. and Théberge F. "Community Detection Algorithm Using Hypergraph Modularity". In: Benito R.M., Cherifi C., Cherifi H., Moro E., Rocha L.M., Sales-Pardo M. (eds) Complex Networks & Their Applications IX. COMPLEX NETWORKS 2020. Studies in Computational Intelligence, vol 943. Springer, Cham. https://doi.org/10.1007/978-3-030-65347-7_13

- **HG** (Hypergraph) –
- **L** (*list of sets*) – some initial partition of the vertices in HG
- **wdc** (*func, optional*) – Hyperparameter for hypergraph modularity[Page 103, 2]
- **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.hypergraph_modularity.`**`linear`**`(d, c)`

Hyperparameter for hypergraph modularity[Page 103, 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.hypergraph_modularity.`**`majority`**`(d, c)`

Hyperparameter for hypergraph modularity[Page 103, 2] for d-edge with c vertices in the majority class. This corresponds to the majority rule[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.hypergraph_modularity.`**`modularity`**`(HG, A, wdc=<function linear>)`

Computes modularity of hypergraph HG with respect to partition A.

Parameters

- **HG** (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[Page 103, 2]

[3] Kamiński B., Poulin V., Prałat P., Szufel P. and Théberge F. "Clustering via hypergraph modularity", Plos ONE 2019, https://doi.org/10.1371/journal.pone.0224307

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[Page 103, 2] for d-edge with c vertices in the majority class. This corresponds to the strict rule[Page 104, 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.hypergraph_modularity.two_section(HG)`

Creates a random walk based[Page 103, 1] 2-section igraph Graph with transition weights defined by the weights of the hyperedges.

Parameters
HG (*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 contruct 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

K. Hayashi, S. Aksoy, C. Park, H. Park, "Hypergraph random walks, Laplacians, and clustering", Proceedings of the 29th ACM International Conference on Information & Knowledge Management. 2020. https://doi.org/10.1145/3340531.3412034

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:

Chung, Fan. "Laplacians and the Cheeger inequality for directed graphs." Annals of Combinatorics 9.1 (2005): 1-19.

and studied in the context of hypergraphs in:

Hayashi, K., Aksoy, S. G., Park, C. H., & Park, H. Hypergraph random walks, laplacians, and clustering. In Proceedings of CIKM 2020, (2020): 495-504.

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.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 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.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:

Hayashi, K., Aksoy, S. G., Park, C. H., & Park, H. Hypergraph random walks, laplacians, and clustering. In Proceedings of CIKM 2020, (2020): 495-504.

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.

S-Metrics for hypergraphs are discussed in depth in: *Aksoy, S.G., Joslyn, C., Ortiz Marrero, C. et al. Hypernetwork science via high-order hypergraph walks. EPJ Data Sci. 9, 16 (2020). https://doi.org/10.1140/epjds/s13688-020-00231-0* 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 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.

The centrality of edge to all shortest s-edge paths V = the set of vertices in the linegraph. $sigma(s,t)$ = the number of shortest paths between vertices s and t. $sigma(s,t|v)$ = the number of those paths that pass through vertex v.

$$c_B(v) = \sum_{s \neq t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

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/((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

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 = 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, 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

$$\text{s-ecc}(u) = \max\{d(u,v) : v \in V\}$$

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: $$C(u) = sum_{v neq u in V} frac{1}{d(v, u)}cdotfrac{2}{(n-1)(n-2)}$$ 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_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*`)`

A continuous-time SIR model for hypergraphs similar to the model in "The effect of heterogeneity on hypergraph contagion models" by Landry and Restrepo https://doi.org/10.1063/5.0020034 and implemented for networks in the EoN package by Joel C. Miller https://epidemicsonnetworks.readthedocs.io/en/latest/

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:

```
>>> 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)
```

algorithms.**Gillespie_SIS**(*H*, *tau*, *gamma*, *transmission_function=<function threshold>*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=inf*, *return_full_data=False*, *sim_kwargs=None*, ***args*)

A continuous-time SIS model for hypergraphs similar to the model in "The effect of heterogeneity on hypergraph contagion models" by Landry and Restrepo https://doi.org/10.1063/5.0020034 and implemented for networks in the EoN package by Joel C. Miller https://epidemicsonnetworks.readthedocs.io/en/latest/

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:

```
>>> 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 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 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 than 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:

```
>>> 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:

```
>>> 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 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:

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> 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)}
>>> fps = 1
>>> fig = plt.figure()
>>> 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.**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:

```
>>> 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.**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:

```
>>> 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 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
- **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.
- **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:

```
>>> 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 C 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.**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

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"}
>>> 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[Page 103, 1]

Parameters

- **HG** (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[Page 103, 2]

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** (Hypergraph) –
- **L** (*list of sets*) – some initial partition of the vertices in HG
- **wdc** (*func, optional*) – Hyperparameter for hypergraph modularity[Page 103, 2]
- **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[Page 103, 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 Z/2Z

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 Z/2Z

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[Page 103, 2] for d-edge with c vertices in the majority class. This corresponds to the majority rule[Page 104, 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** (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[Page 103, 2]

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:

Chung, Fan. "Laplacians and the Cheeger inequality for directed graphs." Annals of Combinatorics 9.1 (2005): 1-19.

and studied in the context of hypergraphs in:

Hayashi, K., Aksoy, S. G., Park, C. H., & Park, H. Hypergraph random walks, laplacians, and clustering. In Proceedings of CIKM 2020, (2020): 495-504.

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.

The centrality of edge to all shortest s-edge paths V = the set of vertices in the linegraph. $sigma(s,t)$ = the number of shortest paths between vertices s and t. $sigma(s,t|v)$ = the number of those paths that pass through vertex v.

$$c_B(v) = \sum_{s \neq t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

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/((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

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, 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_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

$$\text{s-ecc}(u) = \max\{d(u,v) : v \in V\}$$

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: $$C(u) = sum_{v neq u in V} frac{1}{d(v, u)}cdotfrac{2}{(n-1)(n-2)}$$ 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*)

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
- **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: $L M R = S$ and $LLinv = I_m$.

algorithms.**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:

Hayashi, K., Aksoy, S. G., Park, C. H., & Park, H. Hypergraph random walks, laplacians, and clustering. In Proceedings of CIKM 2020, (2020): 495-504.

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.**strict**(*d*, *c*)

Hyperparameter for hypergraph modularity[Page 103, 2] for d-edge with c vertices in the majority class. This corresponds to the strict rule[Page 104, 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 row 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:

```
>>> 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[Page 103, 1] 2-section igraph Graph with transition weights defined by the weights of the hyperedges.

Parameters
HG (Hypergraph) –

Returns
The 2-section graph built from HG

Return type
igraph.Graph

5.4.3 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*, *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:

```
>>> 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** (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 argumetns 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

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** (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={}*, *dr=None*, ***kwargs*)

Draws a convex hull around the nodes contained within each edge in H

Parameters

- **H** (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** (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 the 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** (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** (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*)

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** (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 NetwrokX-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** (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-collumn 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** (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** (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** (`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. Disonneccted 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

- **H** (`Hypergraph`) – the entity to be drawn
- **spacing** (*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** (*iterable*) – list of entities to be labeled
- **count** (*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** (Hypergraph) – the entity to be drawn
- **collapse** (*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** (Hypergraph) – the entity to be drawn
- **collapse** (*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={}*, *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*, *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:

```
>>> 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** (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 argumetns 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

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-collumn 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** (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

5.4.4 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

reports.descriptive_stats.**centrality_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.descriptive_stats.**comp_dist**(*H*, *aggregated=False*)

Computes component sizes, number of nodes.

Parameters

- **H** (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.descriptive_stats.**degree_dist**(*H*, *aggregated=False*)

Computes degrees of nodes of a hypergraph.

Parameters

- **H** (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 (`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** (`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.descriptive_stats.info(H, node=None, edge=None)`

Print a summary of simple statistics for H

Parameters

- **H** (`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** (`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** (`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.descriptive_stats.`**`s_edge_diameter_dist`**`(`*H*`)`

Parameters
H (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

`reports.descriptive_stats.`**`s_node_diameter_dist`**`(`*H*`)`

Parameters
H (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.descriptive_stats.`**`toplex_dist`**`(`*H*, *aggregated=False*`)`

Computes toplex sizes for hypergraph H.

Parameters

- **H** (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.`**`centrality_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** (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** (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)
- 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` (`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`** (`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`** (`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`** (`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** (`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 (`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

`reports.`**`s_node_diameter_dist`**`(`*H*`)`

Parameters
H (`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** (`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

5.5 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.

5.5.1 Graphs and Hypergraphs

Network science is based on the concept of a **graph** $G = \langle V, E \rangle$ 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.

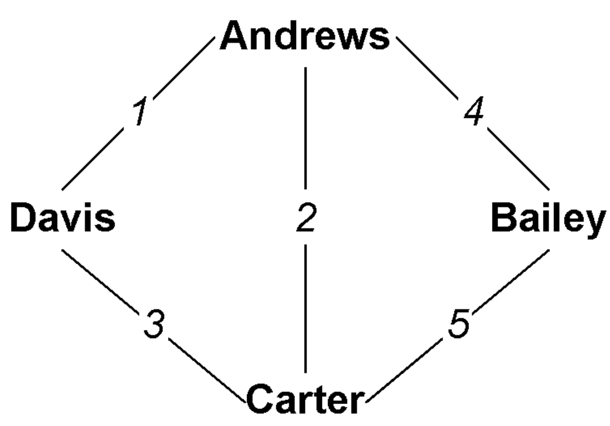


Fig. 5.1: An example graph, where the numbers are edge IDs.

Table 5.1: Adjacency matrix A of a graph.

	Andrews	Bailey	Carter	Davis
Andrews	0	1	1	1
Bailey	1	0	1	0
Carter	1	1	0	1
Davis	1	0	1	1

Table 5.2: Incidence matrix I of a graph.

	1	2	3	4	5
Andrews	1	1	0	1	0
Bailey	0	0	0	1	1
Carter	0	1	1	0	1
Davis	1	0	1	0	0

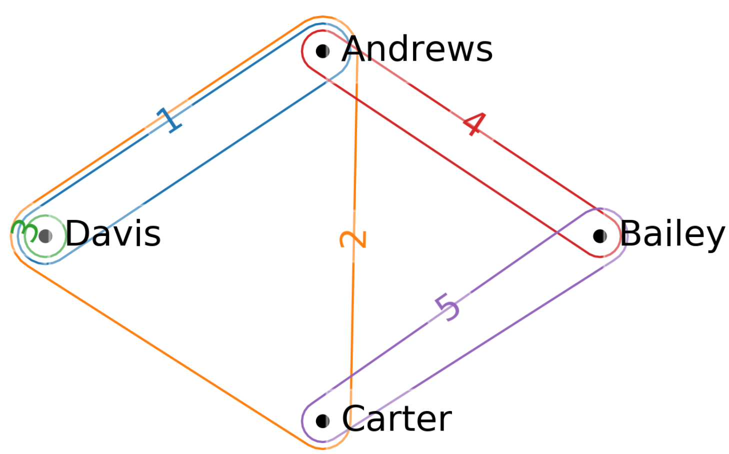


Fig. 5.2: An example hypergraph, where similarly now the hyperedges are shown with numeric IDs.

Table 5.3: Incidence matrix I of a hypergraph.

	1	2	3	4	5
Andrews	1	1	0	1	0
Bailey	0	0	0	1	1
Carter	0	1	0	0	1
Davis	1	1	1	0	0

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.

5.5.2 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 $H^* = \langle E, V \rangle$ 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.

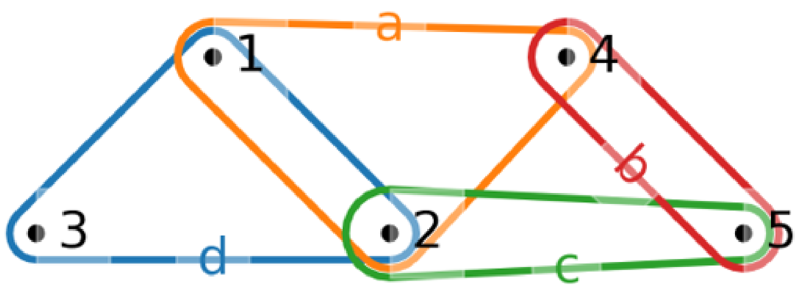


Fig. 5.3: The dual hypergraph H^*.

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^*.

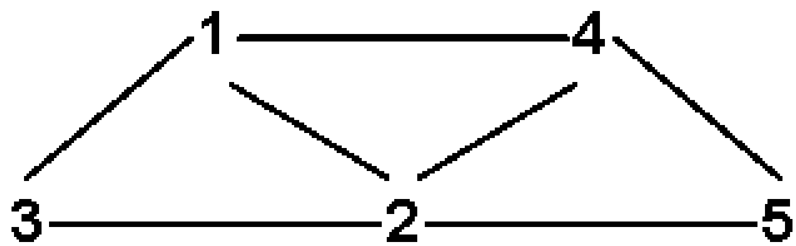


Fig. 5.4: The line graph of H, which is the 2-section of the dual H^*.

Edge Intersections Have Size

As we've already seen, in a graph all the edges are size 2, whereas in a hypergarph edges can be arbitrary size $1, 2, \ldots, n$. 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* 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.

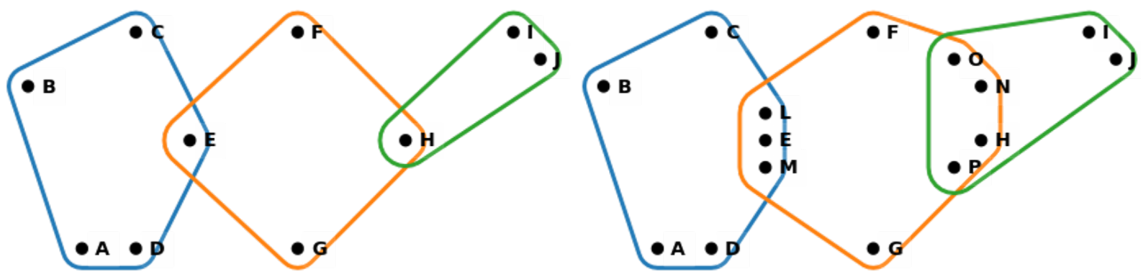


Fig. 5.5: Two hypergraph walks of length 2: (Left) A 1-walk. (Right) A 3-walk.

5.5.3 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 $B = \langle V \sqcup E, I \rangle$. 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. refex shows the **concept lattice** [3], perhaps the most important of these structures, determined by our example.

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

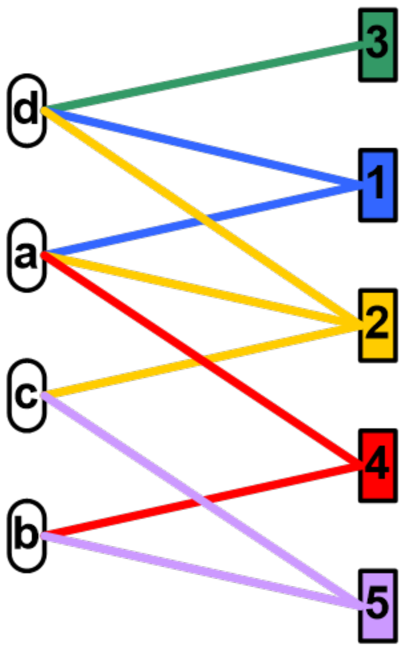


Fig. 5.6: Bipartite graph.

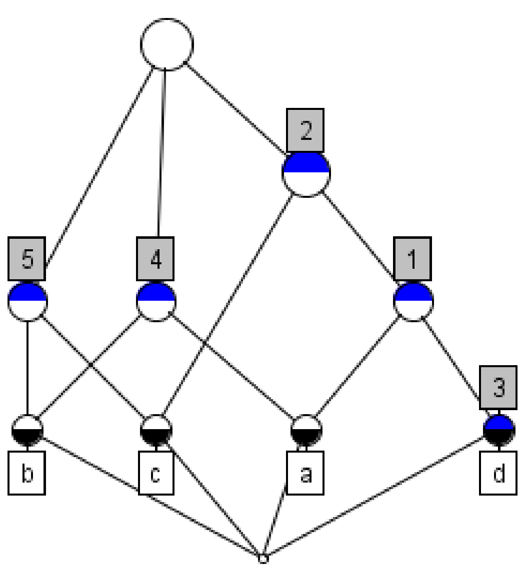


Fig. 5.7: The concept lattice of the example hypergraph H.

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.

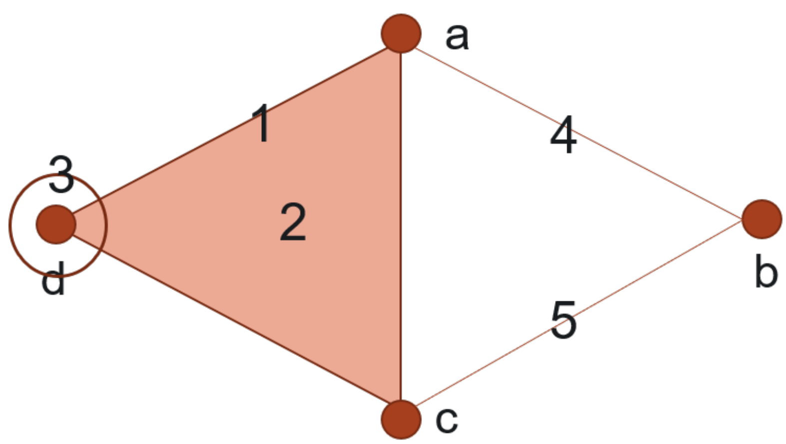


Fig. 5.8: A diagram of the ASC implied by our example. Numbers here indicate the actual hyper-edges in the original hypergraph H, where now additionally all sub-edges, including singletons, are in the ASC.

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 appearnace of a particular vertex in a particular edge (cells of I).

[1] Joslyn, Cliff A; Aksoy, Sinan; Callahan, Tiffany J; Hunter, LE; Jefferson, Brett; Praggastis, Brenda; Purvine, Emilie AH; Tripodi, Ignacio J: (2021) "Hypernetwork Science: From Multidimensional Networks to Computational Topology", in: *Unifying Themes in Complex systems X: Proc. 10th Int. Conf. Complex Systems*, ed. D. Braha et al., pp. 377-392, Springer, `https://doi.org/10.1007/978-3-030-67318-5_25`

[2] Aksoy, Sinan G; Joslyn, Cliff A; Marrero, Carlos O; Praggastis, B; Purvine, Emilie AH: (2020) "Hypernetwork Science via High-Order Hypergraph Walks", *EPJ Data Science*, v. **9**:16, `https://doi.org/10.1140/epjds/s13688-020-00231-0`

[3] Ganter, Bernhard and Wille, Rudolf: (1999) *Formal Concept Analysis*, Springer-Verlag

5.6 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 identifiers instead of their contents. For example, if V = {1,2,3} and E = {e1,e2,e3}, where e1 = {1,2}, e2 = {1,2}, and e3 = {1,2,3}, the edges e1 and e2 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.

5.6.1 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.:

```
>>> H = Hypergraph([{1,2},{1,2},{1,2,3}])
```

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):

```
>>> H = Hypergraph({'e1':[1,2],'e2':[1,2],'e3':[1,2,3]})
```

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:

```
>>> d = {'e1':{ 1: {'w':0.5, 'name': 'related_to'},
>>>             2: {'w':0.1, 'name': 'related_to',
>>>                 'startdate': '05.13.2020'}},
>>>      'e2':{ 1: {'w':0.52, 'name': 'owned_by'},
>>>             2: {'w':0.2}},
>>>      'e3':{ 1: {'w':0.5, 'name': 'related_to'},
>>>             2: {'w':0.2, 'name': 'owner_of'},
>>>             3: {'w':1, 'type': 'relationship'}}
```

```
>>> H = Hypergraph(d, cell_weight_col='w')
```

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:

col1	col2	w	col3
e1	1	0.5	{'name':'related_to'}
e1	2	0.1	**{"name":"related_to",** "start-date":"05.13.2020"}
e2	1	0.52	{"name":"owned_by"}
e2	2	0.2	
…	…	…	{…}

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

```
>>> H = Hypergraph(df,edge_col="col1",node_col="col2",
>>>                 cell_weight_col="w",misc_cell_properties="col3")
```

5. **numpy.ndarray** For homogeneous datasets given in a *n x 2* ndarray a pandas dataframe is generated and column names are added from the edge_col and node_col arguments. Cell properties containing multiple data types are added with a separate dataframe or dict and passed through the cell_properties keyword.

```
>>> arr = np.array([['e1','1'],['e1','2'],
>>>                 ['e2','1'],['e2','2'],
>>>                 ['e3','1'],['e3','2'],['e3','3']])
>>> H = hnx.Hypergraph(arr, column_names=['col1','col2'])
```

5.6.2 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 an id, they may be stored in a single object and passed to the **properties** keyword. For example:

id	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:

```
dp = {id1 : {prop1:val1, prop2,val2,...}, id2 : ... }
```

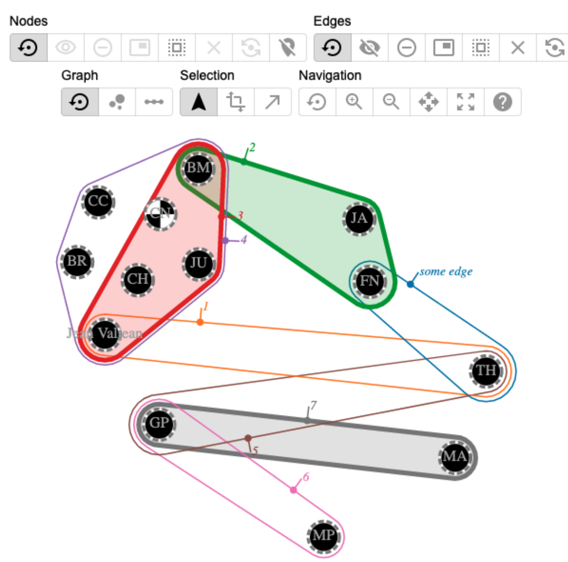
A properties dataframe may be used for nodes and edges sharing ids but differing in cell properties by adding a level index using 0 for edges and 1 for nodes:

level	id	weight	properties
0	e1	5.0	{'type':'event'}
0	e2	0.52	{"name":"owned_by"}
…	…	…	{…}
1	1.2	{'color':'red'}	
2	.003	{'name':'Fido','color':'brown'}	
…	…	…	{…}

5.6.3 Weights

The default key for cell and object weights is "weight". The default value is 1. Weights may be assigned and/or a new default prescribed in the constructor using **cell_weight_col** and **cell_weights** for incidence pairs, and using **edge_weight_prop, node_weight_prop, weight_prop, default_edge_weight,** and **default_node_weight** for node and edge weights.

5.7 Hypernetx-Widget



5.7.1 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.

5.7.2 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.

Do not use python's built-in venv module or virtualenv to create a virtual environment; the widget will not render 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:

```
# update conda
conda update conda

# activate the base environment
conda activate

# install hypernetx and hnxwidget
pip install hypernetx hnxwidget

# install jupyter notebook and extensions
```

(continues on next page)

(continued from previous page)

```
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:

5.7.3 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, wheras 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

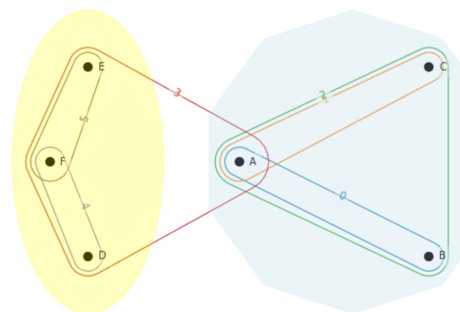
Details on nodes and edges are visible in the side panel. For both nodes and edges, a table shows the node name, degree (or size for edges), its selection state, removed state, and color. These properties can also be controlled directly from this panel. The color of nodes and edges can be set in bulk 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.

5.8 Modularity and Clustering



5.8.1 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.

5.8.2 Installation

Since it is part of HNX, no extra installation is required. The submodule can be imported as follows:

```
import hypernetx.algorithms.hypergraph_modularity as hmod
```

5.8.3 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 = hmod.modularity(HG, A, wdc=hmod.linear)
```

where the 'wdc' parameter points to a function that controls the weights (details below).

In a graph, an edge only links 2 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 $0 le w(d,c) le 1$ control the weight given to such edges. Three functions are supplied in this submodule, namely:

linear
: $w(d,c) = c/d$ if $c > d/2$, else 0.

majority
: $w(d,c) = 1$ if $c > d/2$, else 0.

strict
: $w(d,c) = 1$ iff $c = d$, else 0.

The 'linear' function is used by default. Other functions $w(d,c)$ can be supplied as long as $0 le w(d,c) le 1$ and $w(d,c)=0$ when $c le d$. 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 = 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 distibution of vertices from each part inside each edge. Given hypergraph HG, this is called as:

```
K = hmod.kumar(HG)
```

The other supplied algorithm is a simple method to improve hypergraph modularity directely. 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 = hmod.last_step(HG, A, wdc=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 = dict2part(D)
D = part2dict(A)
```

References

[1] Kumar T., Vaidyanathan S., Ananthapadmanabhan H., Parthasarathy S. and Ravindran B. "A New Measure of Modularity in Hypergraphs: Theoretical Insights and Implications for Effective Clustering". In: Cherifi H., Gaito S., Mendes J., Moro E., Rocha L. (eds) Complex Networks and Their Applications VIII. COMPLEX NETWORKS 2019. Studies in Computational Intelligence, vol 881. Springer, Cham. https://doi.org/10.1007/978-3-030-36687-2_24

[2] Kamiński B., Prałat P. and Théberge F. "Community Detection Algorithm Using Hypergraph Modularity". In: Benito R.M., Cherifi C., Cherifi H., Moro E., Rocha L.M., Sales-Pardo M. (eds) Complex Networks & Their Applications IX. COMPLEX NETWORKS 2020. Studies in Computational Intelligence, vol 943. Springer, Cham. https://doi.org/10.1007/978-3-030-65347-7_13

5.9 Publications

Joslyn, Cliff A; Aksoy, Sinan; Callahan, Tiffany J; Hunter, LE; Jefferson, Brett; Praggastis, Brenda; Purvine, Emilie AH; Tripodi, Ignacio J: (2021) Hypernetwork Science: From Multidimensional Networks to Computational Topology, in: `Unifying Themes in Complex systems X: Proc. 10th Int. Conf. Complex Systems*, ed. D. Braha et al., pp. 377-392, Springer, https://doi.org/10.1007/978-3-030-67318-5_25

Aksoy, Sinan G; Joslyn, Cliff A; Marrero, Carlos O; Praggastis, B; Purvine, Emilie AH: (2020) "Hypernetwork Science via High-Order Hypergraph Walks" , *EPJ Data Science*, v. **9**:16, https://doi.org/10.1140/epjds/s13688-020-00231-0

Aksoy, Sinan G; Hagberg, Aric; Joslyn, Cliff A; Kay, Bill; Purvine, Emilie; Young, Stephen J: (2022) "Models and Methods for Sparse (Hyper)Network Science in Business, Industry, and Government", *Notices of the AMS*, v. **69**:2, pp. 287-291, https://doi.org/10.1090/noti2424

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

Myers, Audun; Joslyn, Cliff A; Kay, Bill; Purvine, EAH; Roek, Gregory; Shapiro, Madelyn: (2023) "Topological Analysis of Temporal Hypergraphs", in: *Proc. Wshop. on Analysis of the Web Graph (WAW 2023)* https://arxiv.org/abs/2302.02857 and *2022 SIAM Conf. on Mathematics of Data Science*, https://www.siam.org/Portals/0/Conferences/MDS/MDS22/MDS22_ABSTRACTS.pdf

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 Algorithms and Models for the Web Graph (WAW 2020), Lecture Notes in Computer Science*, v. **12901**, ed. Kaminski, B et al., pp. 1-15, Springer, https://doi.org/10.1007/978-3-030-48478-1_1

Hayashi, Koby; Aksoy, Sinan G; Park, CH; and Park, Haesun: (2020) "Hypergraph Random Walks, Laplacians, and Clustering", in:**Proc. 29th ACM Int. Conf. Information and Knowledge Management (CIKM 2020)**, pp. 495-504, ACM, New York,** https://doi.org/10.1145/3340531.3412034

Kay, WW; Aksoy, Sinan G; Baird, Molly; Best, DM; Jenne, Helen; Joslyn, CA; Potvin, CD; Roek, Greg; Seppala, Garrett; Young, Stephen; Purvine, Emilie: (2022) "Hypergraph Topological Features for Autoencoder-Based Intrusion Detection for Cybersecurity Data", *ML4Cyber Wshop., Int. Conf. Machine Learning 2022*, https://icml.cc/Conferences/2022/ScheduleMultitrack?event=13458#collapse20252

Liu, Xu T; Firoz, Jesun; Lumsdaine, Andrew; Joslyn, CA; Aksoy, Sinan; Amburg, Ilya; Praggastis, Brenda; Gebremedhin, Assefaw: (2022) "High-Order Line Graphs of Non-Uniform Hypergraphs: Algorithms, Applications, and Experimental Analysis", *36th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS 22)*, https://ieeexplore.ieee.org/document/9820632

Liu, Xu T; Firoz, Jesun; Lumsdaine, Andrew; Joslyn, CA; Aksoy, Sinan; Praggastis, Brenda; Gebremedhin, Assefaw: (2021) "Parallel Algorithms for Efficient Computation of High-Order Line Graphs of Hypergraphs", in: *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC 2021)*, https://doi.ieeecomputersociety.org/10.1109/HiPC53243.2021.00045

5.10 License

5.11 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V