deepgraph Documentation

Release 0.2.3

Dominik Traxl

Jul 19, 2023

Contents

Release 0.2.3

Date Jul 19, 2023

Code GitHub

CHAPTER 1

Contents

1.1 What is DeepGraph

DeepGraph is an open source Python implementation of a new network representation introduced here. Its purpose is to facilitate data analysis by interpreting data in terms of network theory.

The basis of this software package is Pandas, a fast and flexible data analysis tool for the Python programming language. Utilizing one of its primary data structures, the DataFrame, we represent objects (i.e. the nodes of a network) by one DataFrame, and their pairwise relations (i.e. the edges of a network) by another DataFrame.

One of the main features of DeepGraph is an efficient and scalable creation of edges. Given a set of nodes in the form of a DataFrame (or an on disc HDFStore), DeepGraph's `core class` provides *methods* to iteratively compute pairwise relations between the nodes (e.g. similarity/distance measures) using arbitrary, user-defined functions on the nodes' features. These methods provide arguments to parallelize the computation and control memory consumption, making them suitable for very large data-sets and adjustable to whatever hardware you have at hand (from netbooks to cluster architectures).

Furthermore, once a graph is constructed, DeepGraph allows you to partition its `nodes`, `edges` or the entire `graph` by the graph's properties and labels, enabling the aggregation, computation and allocation of information on and between arbitrary *groups* of nodes. These methods also let you express elaborate queries on the information contained in a deep graph.

DeepGraph is not meant to replace or compete with already existing Python network libraries, such as NetworkX or graph_tool, but rather to combine and extend their capabilities with the merits of Pandas. For that matter, the core class of DeepGraph provides *interfacing methods* to convert to common network representations and graph objects of popular Python network packages.

Deepgraph also implements a number of useful plotting methods, including drawings on geographical map projections.

It's also possible to represent *multilayer networks* by deep graphs. We're thinking of implementing an interface to a suitable package dedicated to the analysis of multilayer networks.

Note: Please acknowledge the authors and cite the use of this software when results are used in publications or published elsewhere. Various citation formats are available here: https://aip.scitation.org/action/showCitFormats?type=show&doi=10.1063%2F1.4952963

For your convenience, you can find the BibTex entry below:

```
@Article{traxl-2016-deep,
    author      = {Dominik Traxl AND Niklas Boers AND J\"urgen Kurths},
    title       = {Deep Graphs - A general framework to represent and analyze
                   heterogeneous complex systems across scales},
    journal     = {Chaos},
    year        = {2016},
    volume      = {26},
    number      = {6},
    eid         = {065303},
    doi         = {http://dx.doi.org/10.1063/1.4952963},
    eprinttype  = {arxiv},
    eprintclass = {physics.data-an, cs.SI, physics.ao-ph, physics.soc-ph},
    eprint      = {http://arxiv.org/abs/1604.00971v1},
    version     = {1},
    date        = {2016-04-04},
    url         = {http://arxiv.org/abs/1604.00971v1}
}
```

To get started, have a look at

- *Installation of DeepGraph*
- *DeepGraph's Tutorials*
- *API Reference*

Want to share feedback, or contribute?

So far the package has only been developed by *me*, a fact that I would like to change very much. So if you feel like contributing in any way, shape or form, please feel free to contact me, report bugs, create pull requestes, milestones, etc. You can contact me via email: dominik.traxl@posteo.org

Note: This documentation assumes general familiarity with NumPy and Pandas. If you haven't used these packages, do invest some time in learning about them first.

Note: DeepGraph is free software; you can redistribute it and/or modify it under the terms of the BSD License. We highly welcome contributions from the community.

1.2 Installation

1.2.1 Quick Install

DeepGraph can be installed via pip from PyPI

```
$ pip install deepgraph
```

Depending on your system, you may need root privileges. On UNIX-based operating systems (Linux, Mac OS X etc.) this is achieved with sudo

```
$ sudo pip install deepgraph
```

Alternatively, if you're using Conda, install with

```
$ conda install -c conda-forge deepgraph
```

1.2.2 Installing from Source

Alternatively, you can install DeepGraph from source by downloading a source archive file (tar.gz or zip).

Source Archive File

1. Download the source (tar.gz or zip file) from https://pypi.python.org/pypi/deepgraph/ or https://github.com/deepgraph/deepgraph/
2. Unpack and change directory to the source directory (it should have the files README.rst and setup.py).
3. Run `python setup.py install` to build and install. As a developer, you may want to install using cython: `python setup.py install --use-cython`.
4. (Optional) Run `py.test` to execute the tests if you have pytest installed.

GitHub

1. Clone the deepgraph repostitory

   git clone https://github.com/deepgraph/deepgraph.git
2. Change directory to `deepgraph`
3. Run `python setup.py install` to build and install. As a developer, you may want to install using cython: `python setup.py install --use-cython`.
4. (Optional) Run `py.test` to execute the tests if you have pytest installed.

Installing without Root Privileges

If you don't have permission to install software on your system, you can install into another directory using the `--user`, `--prefix`, or `--home` flags to setup.py.

For example

```
$ python setup.py install --prefix=/home/username/python
```

or

```
$ python setup.py install --home=~
```

or

```
$ python setup.py install --user
```

Note: If you didn't install in the standard Python site-packages directory you will need to set your PYTHONPATH variable to the alternate location. See here for further details.

1.2.3 Requirements

The easiest way to get Python and the required/optional packages is to use Conda (or Miniconda), a cross-platform (Linux, Mac OS X, Windows) Python distribution for data analytics and scientific computing.

Python

To use DeepGraph you need Python 2.7, 3.4 or later.

Pandas

Pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

Pandas is the core dependency of DeepGraph, and it is highly recommended to install the recommended and optional dependencies of Pandas as well.

NumPy

NumPy is the fundamental package for scientific computing with Python.

Needed for internal operations.

1.2.4 Recommended Packages

The following are recommended packages that DeepGraph can use to provide additional functionality.

Matplotlib

Matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

Allows you to use the *plotting methods* of DeepGraph.

Matplotlib Basemap Toolkit

basemap is an add-on toolkit for matplotlib that lets you plot data on map projections with coastlines, lakes, rivers and political boundaries. See the basemap tutorial for documentation and examples of what it can do.

Used by `plot_map` and `plot_map_generator` to plot networks on map projections.

PyTables

PyTables is a package for managing hierarchical datasets and designed to efficiently and easily cope with extremely large amounts of data.

Necessary for HDF5-based storage of pandas DataFrames. DeepGraph's `core class` may be initialized with a HDFStore containing a node table in order to iteratively create edges directly from disc (see `create_edges` and `create_edges_ft`).

SciPy

SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering.

Allows you to convert from DeepGraph's network representation to sparse adjacency matrices (see *return_cs_graph*).

NetworkX

NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

Allows you to convert from DeepGraph's network representation to NetworkX's network representation (see *return_nx_graph*).

Graph-Tool

graph_tool is an efficient Python module for manipulation and statistical analysis of graphs (a.k.a. networks).

Allows you to convert from DeepGraph's network representation to Graph-Tool's network representation (see *return_gt_graph*).

Conda users can install graph_tool by adding the following channels to their ~/.condarc

```
$ conda config --add channels conda-forge
$ conda config --add channels ostrokach-forge
```

Then, install graph-tool

```
$ conda install graph-tool
```

You can test your graph-tool installation by

```
$ python -c "from graph_tool.all import *"
```

1.2.5 Optional Packages

The following packages are considered to provide very useful tools and methods.

Scikit-Learn

sklearn is a Python module integrating classical machine learning algorithms in the tightly-knit world of scientific Python packages (numpy, scipy, matplotlib).

Sklearn-pandas

sklearn-pandas provides a bridge between Scikit-Learn's machine learning methods and pandas-style Data Frames.

1.3 Tutorials

1.3.1 10 Minutes to DeepGraph

[ipython notebook][python script][data]

This is a short introduction to DeepGraph. In the following, we demonstrate DeepGraph's core functionalities by a toy data-set, "flying balls".

First of all, we need to import some packages

```
# for plots
import matplotlib.pyplot as plt

# the usual
import numpy as np
import pandas as pd

import deepgraph as dg

# notebook display
%matplotlib inline
plt.rcParams['figure.figsize'] = 8, 6
pd.options.display.max_rows = 10
pd.set_option('expand_frame_repr', False)
```

Loading Toy Data

Then, we need data in the form of a pandas DataFrame, representing the nodes of our graph

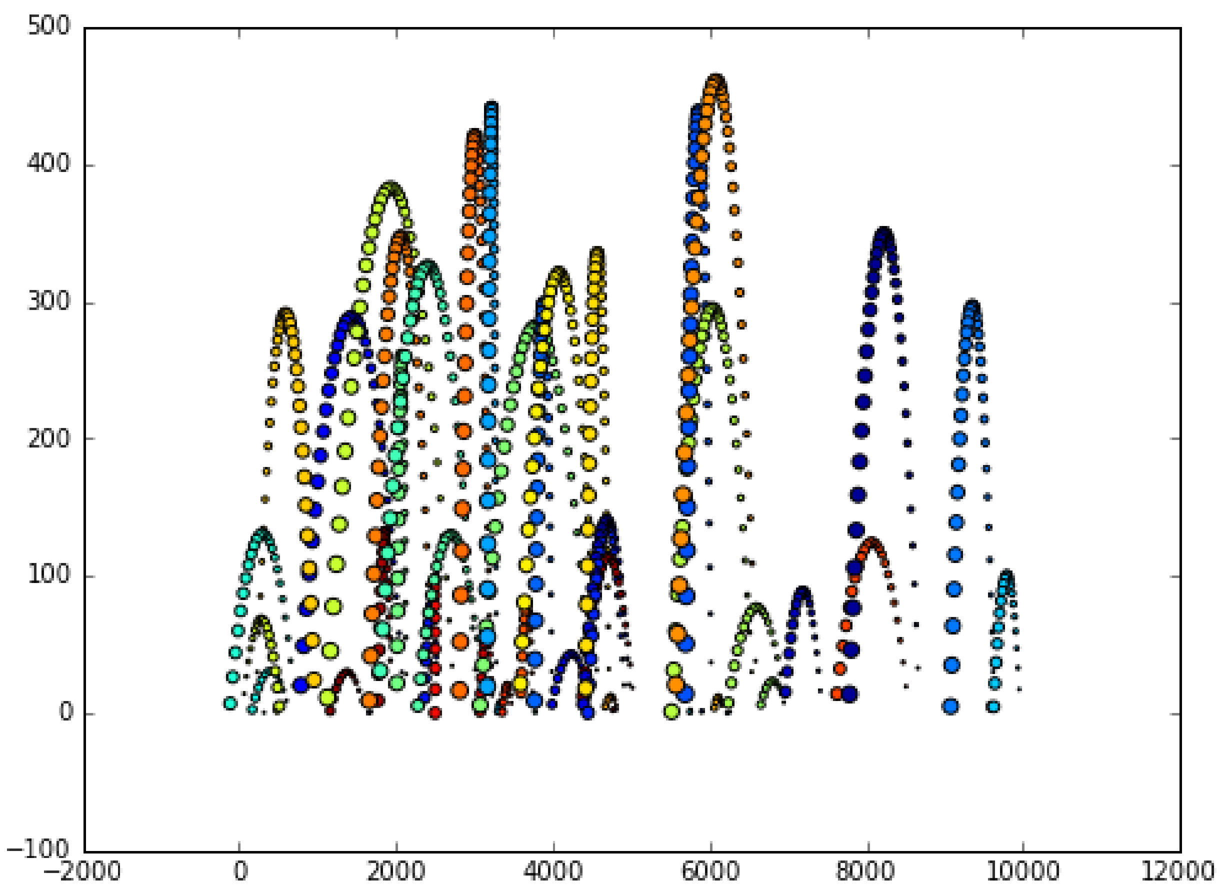
```
v = pd.read_csv('flying_balls.csv', index_col=0)
print(v)
```

```
      time            x          y  ball_id
0        0  1692.000000   0.000000        0
1        0  8681.000000   0.000000        1
2        0   490.000000   0.000000        2
3        0  7439.000000   0.000000        3
4        0  4998.000000   0.000000        4
...    ...          ...        ...      ...
1163    45  2812.552734  16.503178       39
1164    46  5686.915998  14.161693       10
1165    46  3161.729086  19.381823       14
1166    46  5594.233413  57.701712       37
1167    47  5572.216748  20.588750       37

[1168 rows x 4 columns]
```

The data consists of 1168 space-time measurements of 50 different toy balls in two-dimensional space. Each space-time measurement (i.e. row of v) represents a **node**.

Let's plot the data such that each ball has it's own color

```
plt.scatter(v.x, v.y, s=v.time, c=v.ball_id)
```



Creating Edges

In order to create edges between these nodes, we now initiate a *dg.DeepGraph* instance

```
g = dg.DeepGraph(v)
g
```

```
<DeepGraph object, with n=1168 node(s) and m=0 edge(s) at 0x7facf3b35dd8>
```

and use it to create edges between the nodes given by *g.v*. For that matter, we may define a **connector** function

```
def x_dist(x_s, x_t):
    dx = x_t - x_s
    return dx
```

and pass it to *g.create_edges* in order to compute the distance in the x-coordinate of each pair of nodes

```
g.create_edges(connectors=x_dist)
g
```

```
<DeepGraph object, with n=1168 node(s) and m=681528 edge(s) at 0x7facf3b35dd8>
```

```
print(g.e)
```

```
                      dx
s    t
0    1     6989.000000
     2    -1202.000000
     3     5747.000000
     4     3306.000000
     5     2812.000000
...                ...
1164 1166   -92.682585
     1167  -114.699250
1165 1166  2432.504327
     1167  2410.487662
1166 1167   -22.016665

[681528 rows x 1 columns]
```

Let's say we're only interested in creating edges between nodes with a x-distance smaller than 1000. Then we may additionally define a **selector**

```
def x_dist_selector(dx, sources, targets):
    dxa = np.abs(dx)
    sources = sources[dxa <= 1000]
    targets = targets[dxa <= 1000]
    return sources, targets
```

and pass both the **connector** and **selector** to *g.create_edges*

```
g.create_edges(connectors=x_dist, selectors=x_dist_selector)
g
```

```
<DeepGraph object, with n=1168 node(s) and m=156938 edge(s) at 0x7facf3b35dd8>
```

```
print(g.e)
```

```
                    dx
s    t
0    6     416.000000
     7     848.000000
     19   -973.000000
     24    437.000000
     38    778.000000
...               ...
1162 1167  -44.033330
1163 1165  349.176351
1164 1166  -92.682585
     1167 -114.699250
1166 1167  -22.016665

[156938 rows x 1 columns]
```

There is, however, a much more efficient way of creating edges that involve a simple distance threshold such as the one above

Creating Edges on a FastTrack

In order to efficiently create edges including a selection of edges via a simple distance threshold as above, one should use the `create_edges_ft` method. It relies on a sorted DataFrame, so we need to sort `g.v` first

```
g.v.sort_values('x', inplace=True)
```

```
g.create_edges_ft(ft_feature=('x', 1000))
g
```

```
<DeepGraph object, with n=1168 node(s) and m=156938 edge(s) at 0x7facf3b35dd8>
```

Let's compare the efficiency

```
%timeit -n3 -r3 g.create_edges(connectors=x_dist, selectors=x_dist_selector)
```

```
3 loops, best of 3: 557 ms per loop
```

```
%timeit -n3 -r3 g.create_edges_ft(ft_feature=('x', 1000))
```

```
3 loops, best of 3: 167 ms per loop
```

The `create_edges_ft` method also accepts **connectors** and **selectors** as input. Let's connect only those measurements that are close in space and time

```
def y_dist(y_s, y_t):
    dy = y_t - y_s
    return dy

def time_dist(time_t, time_s):
    dt = time_t - time_s
    return dt

def y_dist_selector(dy, sources, targets):
    dya = np.abs(dy)
    sources = sources[dya <= 100]
    targets = targets[dya <= 100]
    return sources, targets

def time_dist_selector(dt, sources, targets):
    dta = np.abs(dt)
    sources = sources[dta <= 1]
    targets = targets[dta <= 1]
    return sources, targets
```

```
g.create_edges_ft(ft_feature=('x', 100),
                  connectors=[y_dist, time_dist],
                  selectors=[y_dist_selector, time_dist_selector])
g
```

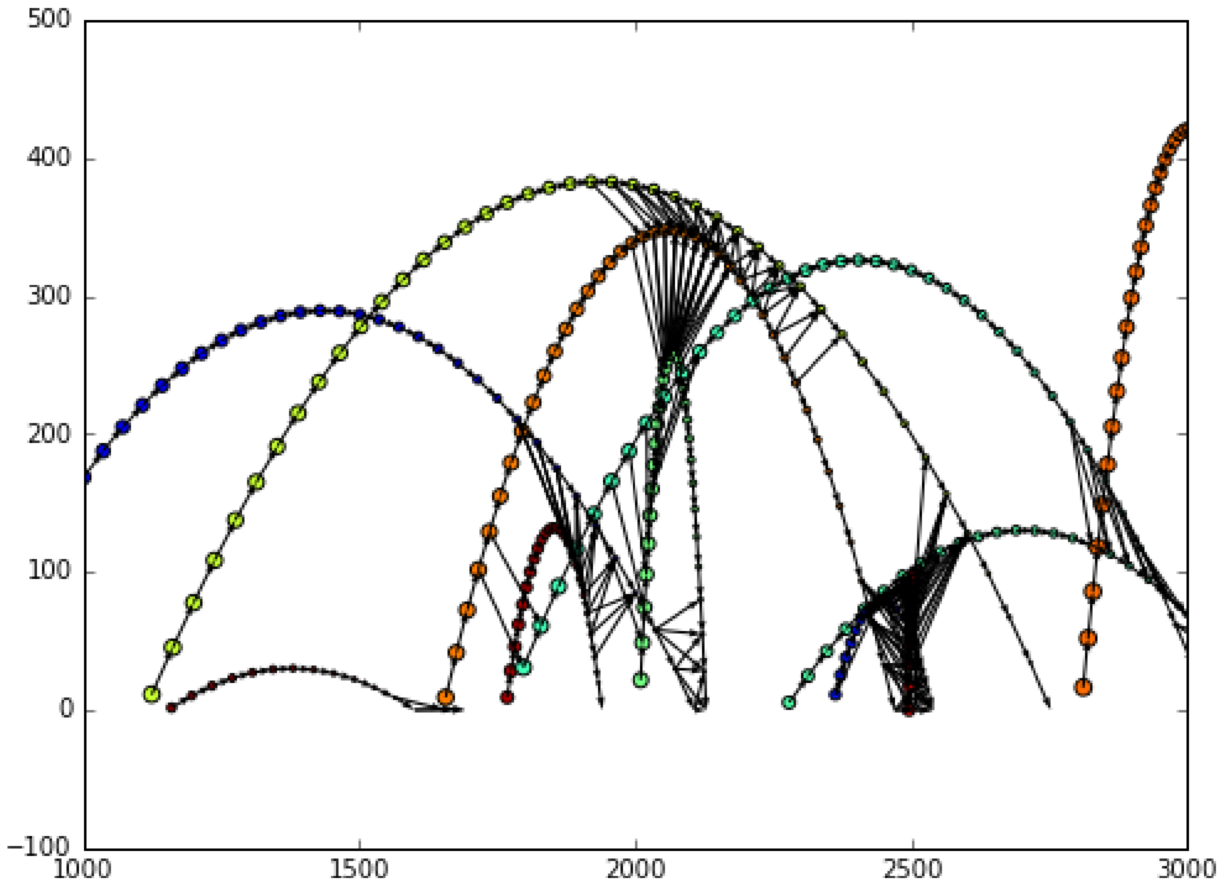
```
<DeepGraph object, with n=1168 node(s) and m=1899 edge(s) at 0x7facf3b35dd8>
```

```
print(g.e)
```

```
         dt         dy       ft_r
s   t
890 867  -1  19.311136  33.415831
867 843  -1  17.678482  33.415831
843 818  -1  16.045829  33.415831
818 792  -1  14.413176  33.415831
792 766  -1  12.780523  33.415831
...      ..        ...        ...
244 203  -1 -10.825226  15.455612
203 159  -1 -12.457879  15.455612
159 114  -1 -14.090532  15.455612
114 65   -1 -15.723185  15.455612
65  16   -1 -17.355838  15.455612

[1899 rows x 3 columns]
```

We can now plot the flying balls and the edges we just created with the *plot_2d* method

```
obj = g.plot_2d('x', 'y', edges=True,
                kwds_scatter={'c': g.v.ball_id, 's': g.v.time})
obj['ax'].set_xlim(1000,3000)
```



Graph Partitioning

The `DeepGraph` class also offers methods to partition `nodes`, `edges` and an entire `graph`. See the docstrings and the other tutorials for details and examples.

Graph Interfaces

Furthermore, you may inspect the docstrings of `return_cs_graph`, `return_nx_graph` and `return_gt_graph` to see how to convert from DeepGraph's DataFrame representation of a network to sparse adjacency matrices, NetworkX's network representation and graph_tool's network representation.

Plotting Methods

DeepGraph also offers a number of useful Plotting methods. See *plotting methods* for details and have a look at the other tutorials for examples.

1.3.2 Computing Very Large Correlation Matrices in Parallel

`[ipython notebook]` `[python script]`

Note: Please acknowledge the authors and cite the use of this software when results are used in publications or published elsewhere. Various citation formats are available here: https://aip.scitation.org/action/showCitFormats?type=show&doi=10.1063%2F1.4952963

For your convenience, you can find the BibTex entry below:

```
@Article{traxl-2016-deep,
    author       = {Dominik Traxl AND Niklas Boers AND J\"urgen Kurths},
    title        = {Deep Graphs - A general framework to represent and analyze
                    heterogeneous complex systems across scales},
    journal      = {Chaos},
    year         = {2016},
    volume       = {26},
    number       = {6},
    eid          = {065303},
    doi          = {http://dx.doi.org/10.1063/1.4952963},
    eprinttype   = {arxiv},
    eprintclass  = {physics.data-an, cs.SI, physics.ao-ph, physics.soc-ph},
    eprint       = {http://arxiv.org/abs/1604.00971v1},
    version      = {1},
    date         = {2016-04-04},
    url          = {http://arxiv.org/abs/1604.00971v1}
}
```

In this short tutorial, we'll demonstrate how DeepGraph can be used to efficiently compute very large correlation matrices in parallel, with full control over RAM usage.

Assume you have a set of `n_samples` samples, each comprised of `n_features` features and you want to compute the Pearson correlation coefficients between all pairs of features (for the Spearman's rank correlation coefficients, see the *Note*-box below). If your data is small enough, you may use scipy.stats.pearsonr or numpy.corrcoef, but for large data, neither of these methods is feasible. Scipy's pearsonr would be very slow, since you'd have to compute pair-wise correlations in a double loop, and numpy's corrcoef would most likely blow your RAM.

Using DeepGraph's *create_edges* method, you can compute all pair-wise correlations efficiently. In this tutorial, the data is stored on disc and only the relevant subset of features for each iteration will be loaded into memory by the computing nodes. Parallelization is achieved by using python's standard library multiprocessing, but it should be straight-forward to modify the code to accommodate other parallelization libraries. It should also be straight-forward to modify the code in order to compute other correlation/distance/similarity-measures between a set of features.

First of all, we need to import some packages

```
# data i/o
import os

# compute in parallel
from multiprocessing import Pool

# the usual
import numpy as np
import pandas as pd

import deepgraph as dg
```

Let's create a set of variables and store it as a 2d-matrix `X` (`shape=(n_features, n_samples)`) on disc. To speed up the computation of the correlation coefficients later on, we whiten each variable.

```
# create observations
from numpy.random import RandomState
prng = RandomState(0)
n_features = int(5e3)
n_samples = int(1e2)
X = prng.randint(100, size=(n_features, n_samples)).astype(np.float64)

# uncomment the next line to compute ranked variables for Spearman's correlation␣
↪coefficients
# X = X.argsort(axis=1).argsort(axis=1)

# whiten variables for fast parallel computation later on
X = (X - X.mean(axis=1, keepdims=True)) / X.std(axis=1, keepdims=True)

# save in binary format
np.save('samples', X)
```

Note: On the computation of the Spearman's rank correlation coefficients: Since the Spearman correlation coefficient is defined as the Pearson correlation coefficient between the ranked variables, it suffices to uncomment the indicated line in the above code-block in order to compute the Spearman's rank correlation coefficients in the following.

Now we can compute the pair-wise correlations using DeepGraph's *create_edges* method. Note that the node table *v* only stores references to the mem-mapped array containing the samples.

```
# parameters (change these to control RAM usage)
step_size = 1e5
n_processes = 100

# load samples as memory-map
X = np.load('samples.npy', mmap_mode='r')

# create node table that stores references to the mem-mapped samples
v = pd.DataFrame({'index': range(X.shape[0])})
```

(continues on next page)

(continued from previous page)

```
# connector function to compute pairwise pearson correlations
def corr(index_s, index_t):
    features_s = X[index_s]
    features_t = X[index_t]
    corr = np.einsum('ij,ij->i', features_s, features_t) / n_samples
    return corr

# index array for parallelization
pos_array = np.array(np.linspace(0, n_features*(n_features-1)//2, n_processes),␣
↪dtype=int)

# parallel computation
def create_ei(i):

    from_pos = pos_array[i]
    to_pos = pos_array[i+1]

    # initiate DeepGraph
    g = dg.DeepGraph(v)

    # create edges
    g.create_edges(connectors=corr, step_size=step_size,
                   from_pos=from_pos, to_pos=to_pos)

    # store edge table
    g.e.to_pickle('tmp/correlations/{}.pickle'.format(str(i).zfill(3)))

# computation
if __name__ == '__main__':
    os.makedirs("tmp/correlations", exist_ok=True)
    indices = np.arange(0, n_processes - 1)
    p = Pool()
    for _ in p.imap_unordered(create_ei, indices):
        pass
```

Let’s collect the computed correlation values and store them in an hdf file.

```
# store correlation values
files = os.listdir('tmp/correlations/')
files.sort()
store = pd.HDFStore('e.h5', mode='w')
for f in files:
    et = pd.read_pickle('tmp/correlations/{}'.format(f))
    store.append('e', et, format='t', data_columns=True, index=False)
store.close()
```

Let’s have a quick look at the correlations.

```
# load correlation table
e = pd.read_hdf('e.h5')
print(e)
```

```
               corr
s    t
0    1    -0.006066
```

(continues on next page)

(continued from previous page)

```
     2     0.094063
     3    -0.025529
     4     0.074080
     5     0.035490
     6     0.005221
     7     0.032064
     8     0.000378
     9    -0.049318
     10   -0.084853
     11    0.026407
     12    0.028543
     13   -0.013347
     14   -0.180113
     15    0.151164
     16   -0.094398
     17   -0.124582
     18   -0.000781
     19   -0.044138
     20   -0.193609
     21    0.003877
     22    0.048305
     23    0.006477
     24   -0.021291
     25   -0.070756
     26   -0.014906
     27   -0.197605
     28   -0.103509
     29    0.071503
     30    0.120718
...                ...
4991 4998 -0.012007
     4999 -0.252836
4992 4993  0.202024
     4994 -0.046088
     4995 -0.028314
     4996 -0.052319
     4997 -0.010797
     4998 -0.025321
     4999 -0.093721
4993 4994 -0.027568
     4995  0.045602
     4996 -0.102075
     4997  0.035370
     4998 -0.069946
     4999 -0.031208
4994 4995  0.108063
     4996  0.144441
     4997  0.078353
     4998 -0.024799
     4999 -0.026432
4995 4996 -0.019991
     4997 -0.178458
     4998 -0.162406
     4999  0.102835
4996 4997  0.115812
     4998 -0.061167
     4999  0.018606
```

(continues on next page)

(continued from previous page)

```
4997 4998 -0.151932
     4999 -0.271358
4998 4999  0.106453

[12497500 rows x 1 columns]
```

And finally, let's see where most of the computation time is spent.

```
g = dg.DeepGraph(v)
p = %prun -r g.create_edges(connectors=corr, step_size=step_size)
```

```
p.print_stats(20)
```

```
         244867 function calls (239629 primitive calls) in 14.193 seconds

   Ordered by: internal time
   List reduced from 541 to 20 due to restriction <20>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      250    9.355    0.037    9.361    0.037 memmap.py:334(__getitem__)
      125    1.584    0.013    1.584    0.013 {built-in method numpy.core.multiarray.
↪c_einsum}
      125    1.012    0.008   12.013    0.096 deepgraph.py:4558(map)
        2    0.581    0.290    0.581    0.290 {method 'get_labels' of 'pandas._libs.
↪hashtable.Int64HashTable' objects}
        1    0.301    0.301    0.414    0.414 multi.py:795(_engine)
        5    0.157    0.031    0.157    0.031 {built-in method numpy.core.multiarray.
↪concatenate}
      250    0.157    0.001    0.170    0.001 internals.py:5017(_stack_arrays)
        2    0.105    0.053    0.105    0.053 {pandas._libs.algos.take_1d_int64_int64}
      889    0.094    0.000    0.094    0.000 {method 'reduce' of 'numpy.ufunc'␣
↪objects}
      125    0.089    0.001   12.489    0.100 deepgraph.py:5294(_select_and_return)
      125    0.074    0.001    0.074    0.001 {deepgraph._triu_indices._reduce_triu_
↪indices}
      125    0.066    0.001    0.066    0.001 {built-in method deepgraph._triu_
↪indices._triu_indices}
        4    0.038    0.009    0.038    0.009 {built-in method pandas._libs.algos.
↪ensure_int16}
      125    0.033    0.000   10.979    0.088 <ipython-input-3-26c4f59cd911>:12(corr)
        2    0.028    0.014    0.028    0.014 function_base.py:4703(delete)
        1    0.027    0.027   14.163   14.163 deepgraph.py:4788(_matrix_iterator)
        1    0.027    0.027    0.113    0.113 multi.py:56(_codes_to_ints)
45771/45222    0.020    0.000    0.043    0.000 {built-in method builtins.isinstance}
        1    0.019    0.019   14.193   14.193 deepgraph.py:183(create_edges)
        2    0.012    0.006    0.700    0.350 algorithms.py:576(factorize)
```

As you can see, most of the time is spent by getting the requested features in the corr-function, followed by computing the correlation values themselves.

1.3.3 Building a DeepGraph of Extreme Precipitation

[ipython notebook][python script][data]

In the following we build a deep graph of a high-resolution dataset of precipitation measurements.

The goal is to first detect spatiotemporal clusters of extreme precipitation events and then to create families of these clusters based on a spatial correlation measure. Finally, we create and plot some informative (intersection) partitions of the deep graph.

For further details see section V of the original paper: https://arxiv.org/abs/1604.00971

First of all, we need to import some packages

```
# data i/o
import os
import xarray

# for plots
import matplotlib.pyplot as plt

# the usual
import numpy as np
import pandas as pd

import deepgraph as dg

# notebook display
from IPython.display import HTML
%matplotlib inline
plt.rcParams['figure.figsize'] = 8, 6
pd.options.display.max_rows = 10
pd.set_option('expand_frame_repr', False)
```

Selecting and Preprocessing the Precipitation Data

Selection

If you want to select your own spatiotemporal box of precipitation events, you may follow the instructions below and change the filename in the next box of code.

- Go to https://disc.gsfc.nasa.gov/datasets/TRMM_3B42_V7/summary?keywords=TRMM_3B42_V7
- click on “Simple Subset Wizard”
- select the “Date Range” (and if desired a “Spatial Bounding Box”) you’re interested in
- click on “Search for Data Sets”
- expand the list by clicking on the “+” symbol
- mark the check box “precipitation”
- (optional, but recommended) click on the selector to change from “netCDF” to “gzipped netCDF”
- click on “Subset Selected Data Sets”
- click on “View Subset Results”
- right click on the “Get list of URLs for this subset in a file” link, and choose “Save Link As. . . ”
- the downloaded file will have a name similar to “SSW_download_2016-05-03T20_19_28_23621_2oIe06xp.inp”. Note which directory the downloaded file is saved to, and in your Unix shell, set your current working directory to that directory.
- register an account to get authentication credentials using these instructions: https://disc.gsfc.nasa.gov/information/howto/5761bc6a5ad5a18811681bae?keywords=wget

- get the files via

```
os.system("wget --content-disposition --directory-prefix=tmp --load-cookies ~/.urs_
↪cookies --save-cookies ~/.urs_cookies --auth-no-challenge=on --keep-session-cookies␣
↪-i SSW_download_2016-05-03T20_19_28_23621_2oIe06xp.inp")
```

Preprocessing

Next, we need to convert the downloaded netCDF files to a pandas DataFrame, which we can then use to initiate a *dg.DeepGraph*

```
# choose "wet times" threshold
r = .1
# choose "extreme" precipitation threshold
p = .9

v_list = []
for file in os.listdir('tmp'):
    if file.startswith('3B42.'):

        # open the downloaded netCDF file
        # unfortunately, we have to decode times ourselves, since
        # the format of the downloaded files doesn't work
        # see also: https://github.com/pydata/xarray/issues/521
        f = xarray.open_dataset('tmp/{}'.format(file), decode_times=False)

        # create integer-based (x,y) coordinates
        f['x'] = (('longitude'), np.arange(len(f.longitude)))
        f['y'] = (('latitude'), np.arange(len(f.latitude)))

        # convert to pd.DataFrame
        vt = f.to_dataframe()

        # we only consider "wet times", pcp >= 0.1mm/h
        vt = vt[vt.pcp >= r]

        # reset index
        vt.reset_index(inplace=True)

        # add correct times
        ftime = f.time.units.split()[2:]
        year, month, day = ftime[0].split('-')
        hour = ftime[1]
        time = pd.datetime(int(year), int(month), int(day), int(hour))
        vt['time'] = time

        # compute "area" for each event
        vt['area'] = 111**2 * .25**2 * np.cos(2*np.pi*vt.latitude / 360.)

        # compute "volume of water precipitated" for each event
        vt['vol'] = vt.pcp * 3 * vt.area

        # set dtypes -> economize ram
        vt['pcp'] = vt['pcp'] * 100
        vt['pcp'] = vt['pcp'].astype(np.uint16)
        vt['latitude'] = vt['latitude'].astype(np.float16)
```

(continues on next page)

(continued from previous page)

```
        vt['longitude'] = vt['longitude'].astype(np.float16)
        vt['area'] = vt['area'].astype(np.uint16)
        vt['vol'] = vt['vol'].astype(np.uint32)
        vt['x'] = vt['x'].astype(np.uint16)
        vt['y'] = vt['y'].astype(np.uint16)

        # append to list
        v_list.append(vt)
        f.close()

# concatenate the DataFrames
v = pd.concat(v_list)

# append a column indicating geographical locations (i.e., supernode labels)
v['g_id'] = v.groupby(['longitude', 'latitude']).grouper.group_info[0]
v['g_id'] = v['g_id'].astype(np.uint32)

# select `p`th percentile of precipitation events for each geographical location
v = v.groupby('g_id').apply(lambda x: x[x.pcp >= x.pcp.quantile(p)])

# append integer-based time
dtimes = pd.date_range(v.time.min(), v.time.max(), freq='3H')
dtdic = {dtime: itime for itime, dtime in enumerate(dtimes)}
v['itime'] = v.time.apply(lambda x: dtdic[x])
v['itime'] = v['itime'].astype(np.uint16)

# sort by time
v.sort_values('time', inplace=True)

# set unique node index
v.set_index(np.arange(len(v)), inplace=True)

# shorten column names
v.rename(columns={'pcp': 'r',
                  'latitude': 'lat',
                  'longitude': 'lon',
                  'time': 'dtime',
                  'itime': 'time'},
         inplace=True)
```

The created DataFrame of extreme precipitation measurements looks like this

```
print(v)
```

```
            lat      lon      dtime     r    x    y  area    vol   g_id  time
0        15.125 -118.125 2004-08-20  1084   28  101   743  24174   5652     0
1        44.875  -30.625 2004-08-20   392  378  220   545   6433  85341     0
2        45.125  -30.625 2004-08-20   454  378  221   543   7416  85342     0
3        45.375  -30.625 2004-08-20   909  378  222   540  14767  85343     0
4        45.625  -30.625 2004-08-20   907  378  223   538  14669  85344     0
...         ...      ...        ...   ...  ...  ...   ...    ...    ...   ...
382306   26.875  -46.625 2004-09-27   503  314  148   686  10385  70380   304
382307   38.375  -37.125 2004-09-27   453  352  194   603   8222  79095   304
382308    8.125 -105.125 2004-09-27   509   80   73   762  11663  17007   304
382309   21.875  -42.875 2004-09-27   260  329  128   714   5595  73875   304
382310    6.625 -111.125 2004-09-27   192   56   67   764   4428  11790   304
```

(continues on next page)

(continued from previous page)

```

[382311 rows x 10 columns]
```

We identify each row of this table as a node of our `DeepGraph`

```
g = dg.DeepGraph(v)
```

Plot the Data

Let's take a look at the data by creating a video of the time-evolution of precipitation measurements. Using the `plot_map_generator` method, this is straight forward.

```
# configure map projection
kwds_basemap = {'llcrnrlon': v.lon.min() - 1,
                'urcrnrlon': v.lon.max() + 1,
                'llcrnrlat': v.lat.min() - 1,
                'urcrnrlat': v.lat.max() + 1,
                'resolution': 'i'}

# configure scatter plots
kwds_scatter = {'s': 1.5,
                'c': g.v.r.values / 100.,
                'edgecolors': 'none',
                'cmap': 'viridis_r'}

# create generator of scatter plots on map
objs = g.plot_map_generator('lon', 'lat', 'dtime',
                            kwds_basemap=kwds_basemap,
                            kwds_scatter=kwds_scatter)

# plot and store frames
for i, obj in enumerate(objs):

    # configure plots
    cb = obj['fig'].colorbar(obj['pc'], fraction=0.025, pad=0.01)
    cb.set_label('[mm/h]')
    obj['m'].fillcontinents(color='0.2', zorder=0, alpha=.4)
    obj['ax'].set_title('{}'.format(obj['group']))

    # store and close
    obj['fig'].savefig('tmp/pcp_{:03d}.png'.format(i),
                       dpi=300, bbox_inches='tight')
    plt.close(obj['fig'])
```

```
# create video with ffmpeg
cmd = "ffmpeg -y -r 5 -i tmp/pcp_%03d.png -c:v libx264 -r 20 -vf scale=2052:1004 {}.
↪mp4"
os.system(cmd.format('precipitation_files/pcp'))
```

```
# embed video
HTML("""
<video width="700" height="350" controls>
  <source src="precipitation_files/pcp.mp4" type="video/mp4">
```

(continues on next page)

(continued from previous page)

```
</video>
""")
```

[download video]

Detecting SpatioTemporal Clusters of Extreme Precipitation

In this tutorial, we're interested in local formations of spatiotemporal clusters of extreme precipitation events. For that matter, we now use DeepGraph to identify such clusters and track their temporal evolution.

Create Edges

We now use DeepGraph to create edges between the nodes given by `g.v`.

The edges of `g` will be utilized to detect spatiotemporal clusters in the data, or in more technical terms: to partition the set of all nodes into subsets of connected grid points. One can imagine the nodes to be elements of a 3 dimensional grid box (x,y,time), where we allow every node to have 26 possible neighbours (8 neighbours in the time slice of the measurement, t_i, and 9 neighbours in each the time slice $t_i 1$ and $t_i + 1$).

For that matter, we pass the following **connectors**

```
def grid_2d_dx(x_s, x_t):
    dx = x_t - x_s
    return dx

def grid_2d_dy(y_s, y_t):
    dy = y_t - y_s
    return dy
```

and **selectors**

```
def s_grid_2d_dx(dx, sources, targets):
    dxa = np.abs(dx)
    sources = sources[dxa <= 1]
    targets = targets[dxa <= 1]
    return sources, targets

def s_grid_2d_dy(dy, sources, targets):
    dya = np.abs(dy)
    sources = sources[dya <= 1]
    targets = targets[dya <= 1]
    return sources, targets
```

to the *`create_edges_ft`* method

```
g.create_edges_ft(ft_feature=('time', 1),
                  connectors=[grid_2d_dx, grid_2d_dy],
                  selectors=[s_grid_2d_dx, s_grid_2d_dy],
                  r_dtype_dic={'ft_r': np.bool,
                               'dx': np.int8,
                               'dy': np.int8},
                  logfile='create_e',
                  max_pairs=1e7)
```

(continues on next page)

(continued from previous page)

```
# rename fast track relation
g.e.rename(columns={'ft_r': 'dt'}, inplace=True)
```

To see how many nodes and edges our graph's comprised of, one may simply type

```
g
```

```
<DeepGraph object, with n=382311 node(s) and m=567225 edge(s) at 0x7f7a4c3de160>
```

The edges we just created look like this

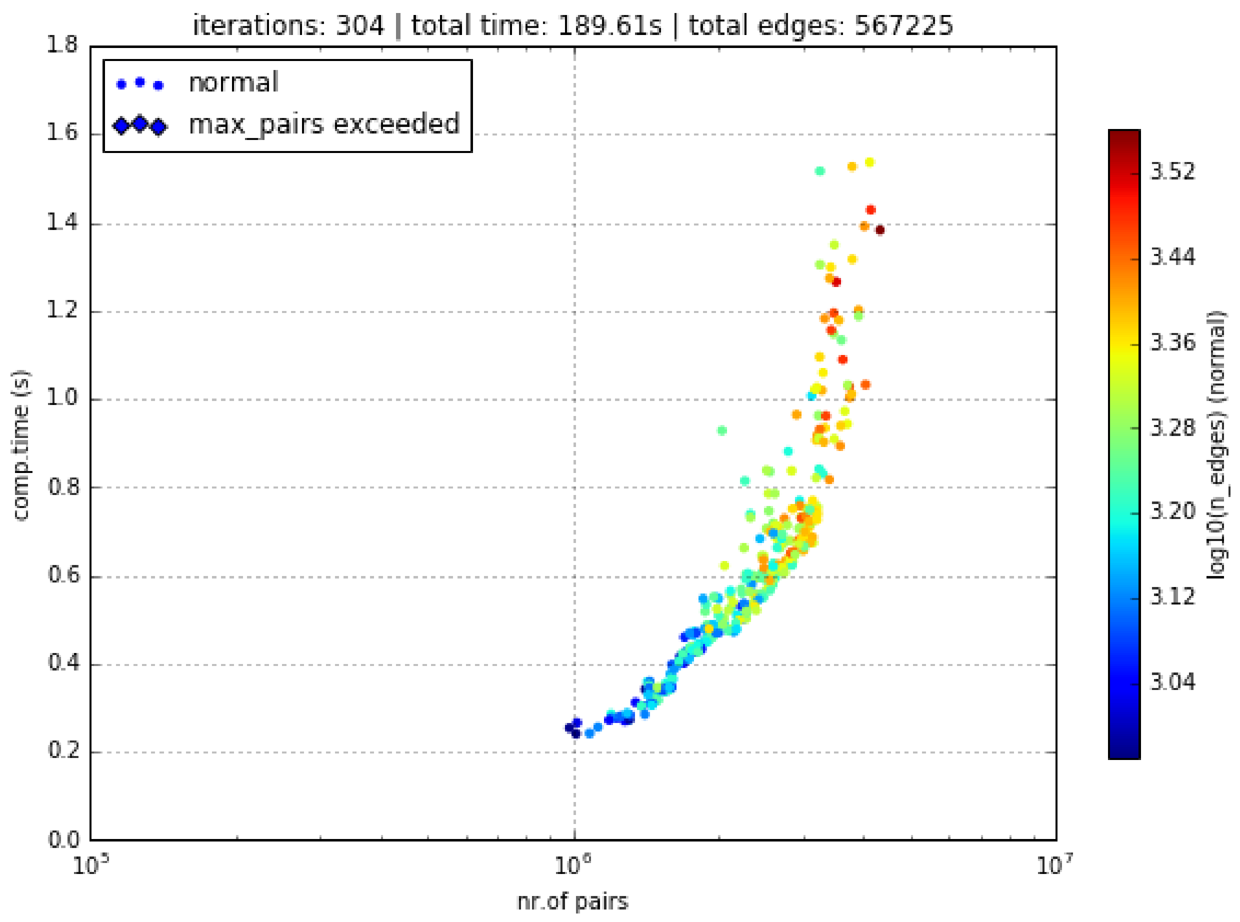
```
print(g.e)
```

```
               dx  dy     dt
s      t
0      1362     0   1  False
       1432     1   0  False
       1433     1   1  False
       1696     1   0   True
       1699     1   1   True
...            ..  ..    ...
382284 382291   0   1  False
382295 382296   0   1  False
382296 382299   0   1  False
382299 382309   0   1  False
382304 382306   0   1  False

[567225 rows x 3 columns]
```

Logfile Plot

To see how long it took to create the edges, one may use the `plot_logfile` method

```
g.plot_logfile('create_e')
```



Find the Connected Components

Having linked all neighbouring nodes, the spatiotemporal clusters can be identified as the connected components of the graph. For practical reasons, *DeepGraph* directly implements a method to find the connected components of a graph, *append_cp*

```
# all singular components (components comprised of one node only)
# are consolidated under the label 0
g.append_cp(consolidate_singles=True)
# we don't need the edges any more
del g.e
```

the node table now has a component membership column appended

```
print(g.v)
```

```
           lat      lon      dtime     r    x    y  area    vol   g_id  time     cp
0       15.125 -118.125 2004-08-20  1084   28  101   743  24174   5652     0    865
1       44.875  -30.625 2004-08-20   392  378  220   545   6433  85341     0   5079
2       45.125  -30.625 2004-08-20   454  378  221   543   7416  85342     0   5079
3       45.375  -30.625 2004-08-20   909  378  222   540  14767  85343     0   5079
4       45.625  -30.625 2004-08-20   907  378  223   538  14669  85344     0   5079
...        ...      ...        ...   ...  ...  ...   ...    ...    ...   ...    ...
```

(continues on next page)

(continued from previous page)

```
382306  26.875  -46.625 2004-09-27   503  314  148   686  10385  70380   304    609
382307  38.375  -37.125 2004-09-27   453  352  194   603   8222  79095   304      0
382308   8.125 -105.125 2004-09-27   509   80   73   762  11663  17007   304    174
382309  21.875  -42.875 2004-09-27   260  329  128   714   5595  73875   304      8
382310   6.625 -111.125 2004-09-27   192   56   67   764   4428  11790   304  15610

[382311 rows x 11 columns]
```

Let's see how many spatiotemporal clusters `g` is comprised of (discarding singular components)

```
g.v.cp.max()
```

```
33169
```

and how many nodes there are in the components

```
print(g.v.cp.value_counts())
```

```
0        64678
1        16460
2         8519
3         6381
4         3403
         ...
29601        2
27554        2
25507        2
23460        2
20159        2
Name: cp, dtype: int64
```

Partition the Nodes Into a Component Supernode Table

In order to aggregate and compute some information about the precipitiation clusters, we now partition the nodes by the type of feature `cp`, the component membership labels of the nodes just created. This can be done with the *partition_nodes* method

```
# feature functions, will be applied to each component of g
feature_funcs = {'dtime': [np.min, np.max],
                 'time': [np.min, np.max],
                 'vol': [np.sum],
                 'lat': [np.mean],
                 'lon': [np.mean]}

# partition the node table
cpv, gv = g.partition_nodes('cp', feature_funcs, return_gv=True)

# append geographical id sets
cpv['g_ids'] = gv['g_id'].apply(set)

# append cardinality of g_id sets
cpv['n_unique_g_ids'] = cpv['g_ids'].apply(len)

# append time spans
```

(continues on next page)

(continued from previous page)

```
cpv['dt'] = cpv['dtime_amax'] - cpv['dtime_amin']

# append spatial coverage
def area(group):
    return group.drop_duplicates('g_id').area.sum()
cpv['area'] = gv.apply(area)
```

The clusters look like this

```
print(cpv)
```

```
       n_nodes          dtime_amin          dtime_amax  time_amin  time_amax   lat_
↪mean     vol_sum   lon_mean                                                 g_ids  n_
↪unique_g_ids                dt       area
cp
0        64678 2004-08-20 00:00:00 2004-09-27 00:00:00          0         304  17.
↪609375  627097323  -63.40625  {0, 1, 2, 6, 7, 10, 12, 13, 14, 22, 23, 24, 25...    ␣
↪      49808 38 days 00:00:00  34781178
1        16460 2004-09-01 06:00:00 2004-09-17 18:00:00         98         230  17.
↪281250  351187150  -65.12500  {65536, 65537, 65538, 65539, 65540, 65541, 655...    ␣
↪       6629 16 days 12:00:00   4803624
2         8519 2004-09-17 03:00:00 2004-09-24 15:00:00        225         285  26.
↪906250  133698579  -44.62500  {73728, 73729, 73730, 73731, 73732, 73733, 737...    ␣
↪       3730  7 days 12:00:00   2507350
3         6381 2004-08-26 09:00:00 2004-09-06 03:00:00         51         137  21.
↪062500  113782748  -64.12500  {65555, 65556, 65557, 65558, 65559, 65560, 655...    ␣
↪       2442 10 days 18:00:00   1749673
4         3403 2004-08-21 21:00:00 2004-08-24 12:00:00         15          36  10.
↪578125   66675326 -111.93750  {8141, 14654, 11805, 16363, 8142, 11806, 20490...    ␣
↪       1294  2 days 15:00:00    978604
...        ...                 ...                 ...        ...         ...    ..
↪.        ...        ...                                                   ...    ␣
↪    ...                 ...       ...
33165        2 2004-08-23 18:00:00 2004-08-23 18:00:00         30          30  15.
↪500000      20212 -103.87500                                      {18115, 18116}    ␣
↪          2  0 days 00:00:00      1483
33166        2 2004-09-05 18:00:00 2004-09-05 18:00:00        134         134  27.
↪250000       9366 -121.87500                                        {2688, 2687}    ␣
↪          2  0 days 00:00:00      1368
33167        2 2004-08-30 15:00:00 2004-08-30 15:00:00         85          85   9.
↪250000      43096    0.62500                                    {112116, 112117}    ␣
↪          2  0 days 00:00:00      1519
33168        2 2004-09-09 03:00:00 2004-09-09 03:00:00        161         161   6.
↪750000      24156  -13.62500                                    {100613, 100614}    ␣
↪          2  0 days 00:00:00      1528
33169        2 2004-09-11 03:00:00 2004-09-11 03:00:00        177         177  15.
↪500000      46798  -16.12500                                      {98523, 98524}    ␣
↪          2  0 days 00:00:00      1483

[33170 rows x 12 columns]
```

Plot the Largest Component

Let's see how the largest cluster of extreme precipitation evolves over time, again using the *plot_map_generator* method

```
# temporary DeepGraph instance containing
# only the largest component
gt = dg.DeepGraph(g.v)
gt.filter_by_values_v('cp', 1)

# configure map projection
from mpl_toolkits.basemap import Basemap
m1 = Basemap(projection='ortho',
             lon_0=cpv.loc[1].lon_mean + 12,
             lat_0=cpv.loc[1].lat_mean + 8,
             resolution=None)
width = (m1.urcrnrx - m1.llcrnrx) * .65
height = (m1.urcrnry - m1.llcrnry) * .45

kwds_basemap = {'projection': 'ortho',
                'lon_0': cpv.loc[1].lon_mean + 12,
                'lat_0': cpv.loc[1].lat_mean + 8,
                'llcrnrx': -0.5 * width,
                'llcrnry': -0.5 * height,
                'urcrnrx': 0.5 * width,
                'urcrnry': 0.5 * height,
                'resolution': 'i'}

# configure scatter plots
kwds_scatter = {'s': 2,
                'c': np.log(gt.v.r.values / 100.),
                'edgecolors': 'none',
                'cmap': 'viridis_r'}

# create generator of scatter plots on map
objs = gt.plot_map_generator('lon', 'lat', 'dtime',
                             kwds_basemap=kwds_basemap,
                             kwds_scatter=kwds_scatter)

# plot and store frames
for i, obj in enumerate(objs):

    # configure plots
    obj['m'].fillcontinents(color='0.2', zorder=0, alpha=.4)
    obj['m'].drawparallels(range(-50, 50, 20), linewidth=.2)
    obj['m'].drawmeridians(range(0, 360, 20), linewidth=.2)
    obj['ax'].set_title('{}'.format(obj['group']))

    # store and close
    obj['fig'].savefig('tmp/cp1_ortho_{:03d}.png'.format(i),
                       dpi=300, bbox_inches='tight')
    plt.close(obj['fig'])
```

```
# create video with ffmpeg
cmd = "ffmpeg -y -r 5 -i tmp/cp1_ortho_%03d.png -c:v libx264 -r 20 -vf␣
↪scale=1919:1406 {}.mp4"
os.system(cmd.format('precipitation_files/cp1_ortho'))
```

```
# embed video
HTML("""
<video width="700" height="500" controls>
  <source src="precipitation_files/cp1_ortho.mp4" type="video/mp4">
```

(continues on next page)

(continued from previous page)

```
</video>
""")
```

[download video]

Detecting Families of Spatially Related Clusters

Create SuperEdges between the Components

We now create superedges between the spatiotemporal clusters in order to find families of clusters that have a strong regional overlap. Passing the following **connectors** and **selector**

```
# compute intersection of geographical locations
def cp_node_intersection(g_ids_s, g_ids_t):
    intsec = np.zeros(len(g_ids_s), dtype=object)
    intsec_card = np.zeros(len(g_ids_s), dtype=np.int)
    for i in range(len(g_ids_s)):
        intsec[i] = g_ids_s[i].intersection(g_ids_t[i])
        intsec_card[i] = len(intsec[i])
    return intsec_card

# compute a spatial overlap measure between clusters
def cp_intersection_strength(n_unique_g_ids_s, n_unique_g_ids_t, intsec_card):
    min_card = np.array(np.vstack((n_unique_g_ids_s, n_unique_g_ids_t)).min(axis=0),
                        dtype=np.float64)
    intsec_strength = intsec_card / min_card
    return intsec_strength

# compute temporal distance between clusters
def time_dist(dtime_amin_s, dtime_amin_t):
    dt = dtime_amin_t - dtime_amin_s
    return dt
```

to the *create_edges* method will provide the information necessary for this task

```
# discard singular components
cpv.drop(0, inplace=True)

# we only consider the largest 5000 clusters
cpv = cpv.iloc[:5000]

# initiate DeepGraph
cpg = dg.DeepGraph(cpv)

# create edges
cpg.create_edges(connectors=[cp_node_intersection,
                             cp_intersection_strength],
                 no_transfer_rs=['intsec_card'],
                 logfile='create_cpe',
                 step_size=1e7)
```

Since no selection of edges has taken place, the number of edges should be `cpg.n`*(`cpg.n`-1)/2

```
cpg
```

```
<DeepGraph object, with n=5000 node(s) and m=12497500 edge(s) at 0x7f7a00aec128>
```

```
print(cpg.e)
```

```
           intsec_strength
s    t
1    2            0.018499
     3            0.002457
     4            0.000000
     5            0.000000
     6            0.000000
...                    ...
4997 4999         0.000000
     5000         0.000000
4998 4999         0.000000
     5000         0.000000
4999 5000         0.000000

[12497500 rows x 1 columns]
```

```
print(cpg.e.intsec_strength.value_counts())
```

```
0.000000    12481941
1.000000         787
0.111111         488
0.333333         481
0.500000         462
              ...
0.012346           1
0.158537           1
0.178082           1
0.658537           1
0.018809           1
Name: intsec_strength, dtype: int64
```

Hierarchically Agglomerate Clusters into Families

Based on the above measure of spatial overlap between clusters, we now perform an agglomerative, hierarchical clustering of the spatio-temporal clusters into regionally coherent families.

```
from scipy.cluster.hierarchy import linkage, fcluster

# create condensed distance matrix
dv = 1 - cpg.e.intsec_strength.values
del cpg.e

# create linkage matrix
lm = linkage(dv, method='average', metric='euclidean')
del dv

# form flat clusters and append their labels to cpv
cpv['F'] = fcluster(lm, 1000, criterion='maxclust')
del lm

```

(continues on next page)

(continued from previous page)

```
# relabel families by size
f = cpv['F'].value_counts().index.values
fdic = {j: i for i, j in enumerate(f)}
cpv['F'] = cpv['F'].apply(lambda x: fdic[x])
```

Let's see how many clusters there are in the families

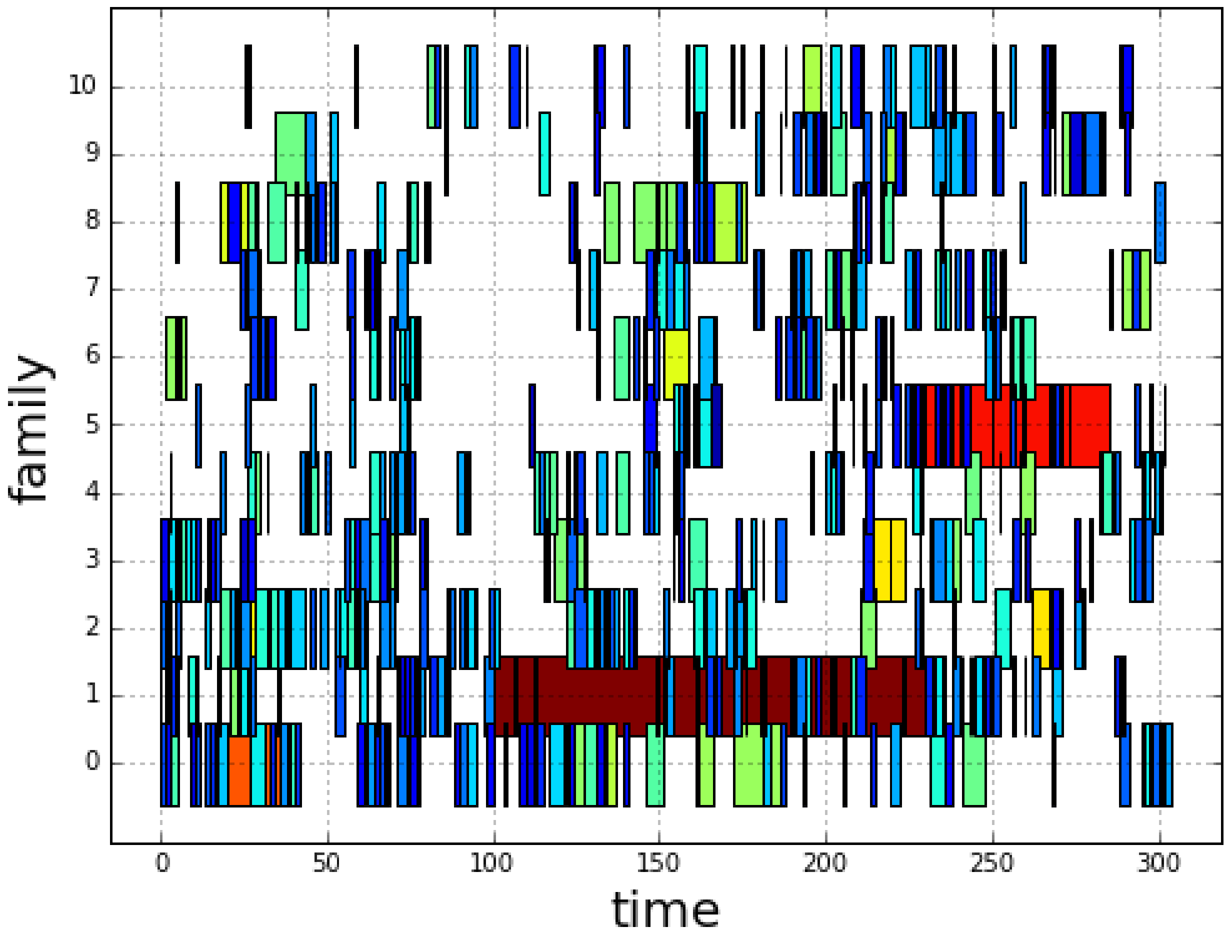
```
print(cpv['F'].value_counts())
```

```
0      79
1      76
2      74
3      56
4      52
       ..
502     1
498     1
494     1
490     1
997     1
Name: F, dtype: int64
```

Create a "Raster Plot" of Families

Let's plot the clusters of the largest 10 families in a raster-like boxplot, by means of the `plot_rects_label_numeric` method

```
cpgt = dg.DeepGraph(cpg.v[cpg.v.F <= 10])
obj = cpgt.plot_rects_label_numeric('F', 'time_amin', 'time_amax',
                                    colors=np.log(cpgt.v.vol_sum.values))
obj['ax'].set_xlabel('time', fontsize=20)
obj['ax'].set_ylabel('family', fontsize=20)
obj['ax'].grid()
```



Create and Plot Informative (Intersection) Partitions

In this last section, we create some useful (intersection) partitions of the deep graph, which we then use to create some plots.

Geographical Locations

```
# how many components have hit a certain
# geographical location (discarding singular cps)
def count(cp):
    return len(set(cp[cp != 0]))

# feature functions, will be applied to each g_id
feature_funcs = {'cp': [count],
                 'vol': [np.sum],
                 'lat': np.min,
                 'lon': np.min}

gv = g.partition_nodes('g_id', feature_funcs)
gv.rename(columns={'lat_amin': 'lat',
                   'lon_amin': 'lon'}, inplace=True)
```

```
print(gv)
```

```
        n_nodes  cp_count     lat  vol_sum      lon
g_id
0             2         1 -10.125    10142 -125.125
1             2         1  -9.875     8716 -125.125
2             2         0  -9.625     4372 -125.125
3             2         2  -9.375     5310 -125.125
4             2         2  -9.125     6409 -125.125
...         ...       ...     ...      ...      ...
115618        2         1  48.875    14319    5.125
115619        1         1  49.125    10129    5.125
115620        2         1  49.375    12826    5.125
115621        2         2  49.625     9117    5.125
115622        2         1  49.875    12101    5.125

[115623 rows x 5 columns]
```

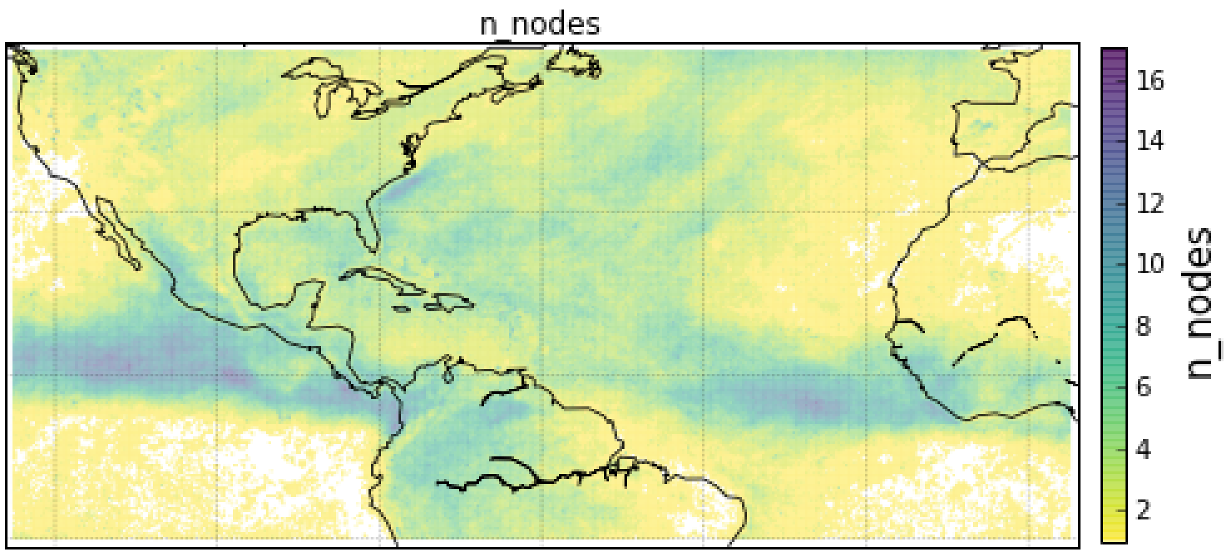
Plot GeoLocational Information

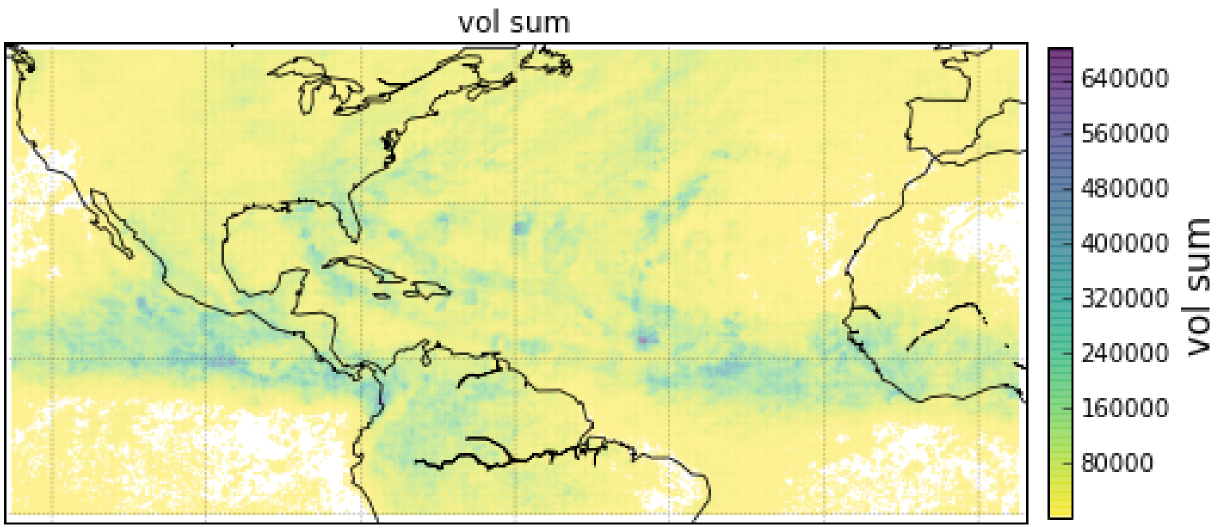
```
cols = {'n_nodes': gv.n_nodes,
        'vol sum': gv.vol_sum,
        'cp count': gv.cp_count}

for name, col in cols.items():

    # for easy filtering, we create a new DeepGraph instance for
    # each component
    gt = dg.DeepGraph(gv)

    # configure map projection
    kwds_basemap = {'llcrnrlon': v.lon.min() - 1,
                    'urcrnrlon': v.lon.max() + 1,
                    'llcrnrlat': v.lat.min() - 1,
                    'urcrnrlat': v.lat.max() + 1}

    # configure scatter plots
    kwds_scatter = {'s': 1,
                    'c': col.values,
                    'cmap': 'viridis_r',
                    'alpha': .5,
                    'edgecolors': 'none'}

    # create scatter plot on map
    obj = gt.plot_map(lon='lon', lat='lat',
                      kwds_basemap=kwds_basemap,
                      kwds_scatter=kwds_scatter)

    # configure plots
    obj['m'].drawcoastlines(linewidth=.8)
    obj['m'].drawparallels(range(-50, 50, 20), linewidth=.2)
    obj['m'].drawmeridians(range(0, 360, 20), linewidth=.2)
    obj['ax'].set_title(name)

    # colorbar
```

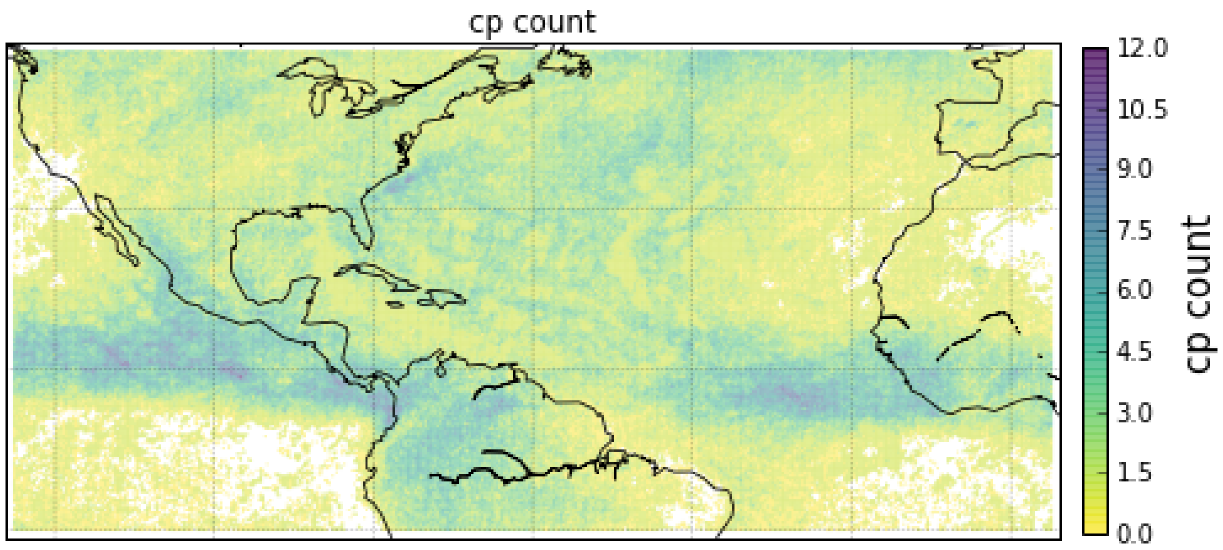
(continues on next page)

(continued from previous page)

```
    cb = obj['fig'].colorbar(obj['pc'], fraction=.022, pad=.02)
    cb.set_label('{}'.format(name), fontsize=15)
```







Geographical Locations and Families

In order to create the intersection partition of geographical locations and families, we first need to append a family membership column to v

```
# create F col
v['F'] = np.ones(len(v), dtype=int) * -1
gcpv = cpv.groupby('F')
it = gcpv.apply(lambda x: x.index.values)

for F in range(len(it)):
    cp_index = v.cp.isin(it.iloc[F])
    v.loc[cp_index, 'F'] = F
```

Then we create the intersection partition

```
# feature funcs
def n_cp_nodes(cp):
    return len(cp.unique())

feature_funcs = {'vol': [np.sum],
                 'lat': np.min,
                 'lon': np.min,
                 'cp': n_cp_nodes}

# create family-g_id intersection graph
fgv = g.partition_nodes(['F', 'g_id'], feature_funcs=feature_funcs)
fgv.rename(columns={'lat_amin': 'lat',
                    'lon_amin': 'lon',
                    'cp_n_cp_nodes': 'n_cp_nodes'}, inplace=True)
```

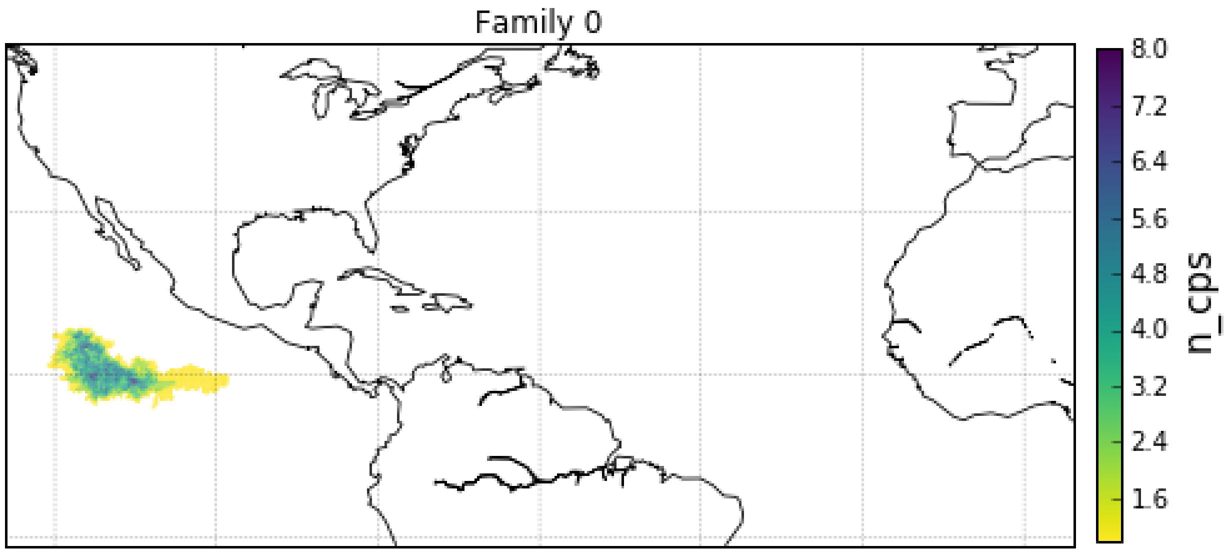
which looks like this

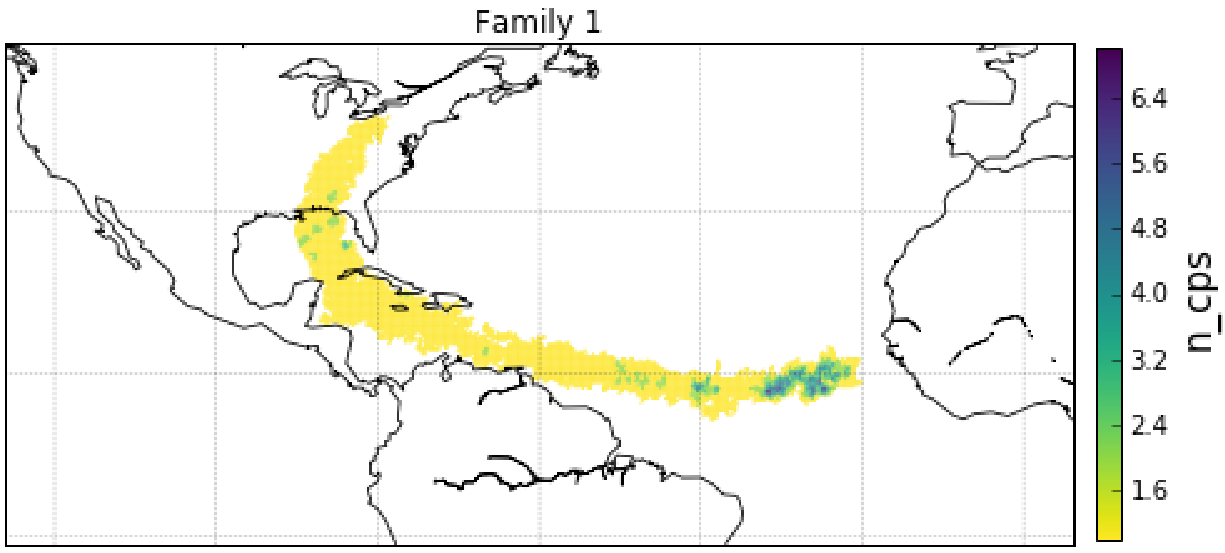
```
print(fgv)
```

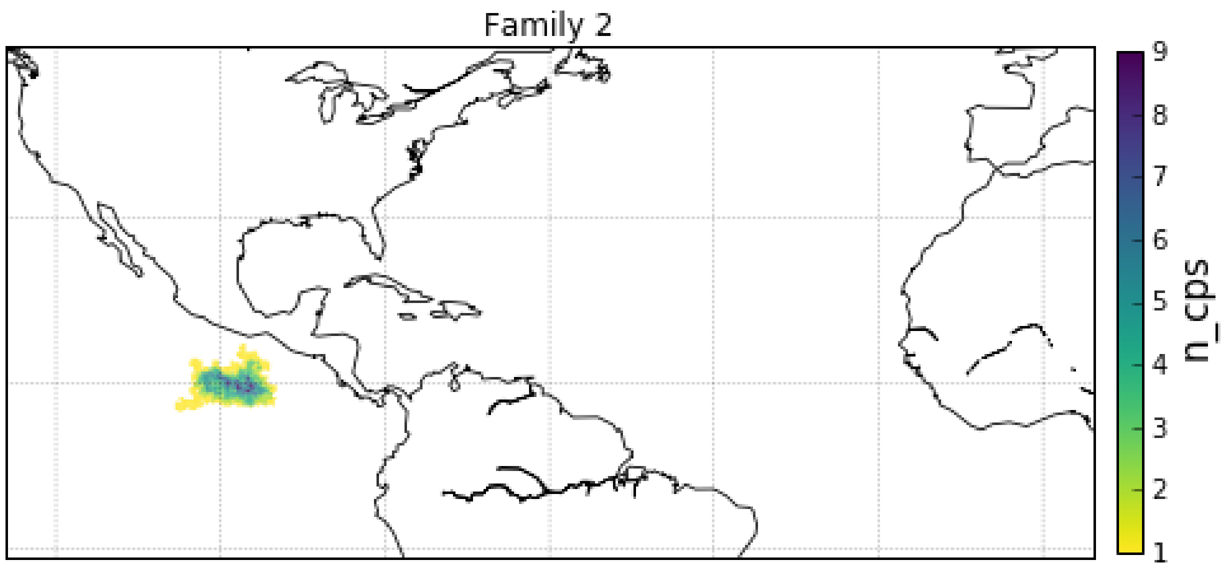
```
             n_nodes  n_cp_nodes     lat  vol_sum      lon
F    g_id
-1   0             2           2 -10.125    10142 -125.125
     1             2           2  -9.875     8716 -125.125
     2             2           1  -9.625     4372 -125.125
     3             2           2  -9.375     5310 -125.125
     4             2           2  -9.125     6409 -125.125
...              ...         ...     ...      ...      ...
 998 26685         1           1  -8.875      593  -93.625
     26686         1           1  -8.625      411  -93.625
     26887         1           1  -9.375      364  -93.375
     26888         1           1  -9.125      478  -93.375
     26889         1           1  -8.875      456  -93.375

[186903 rows x 5 columns]
```

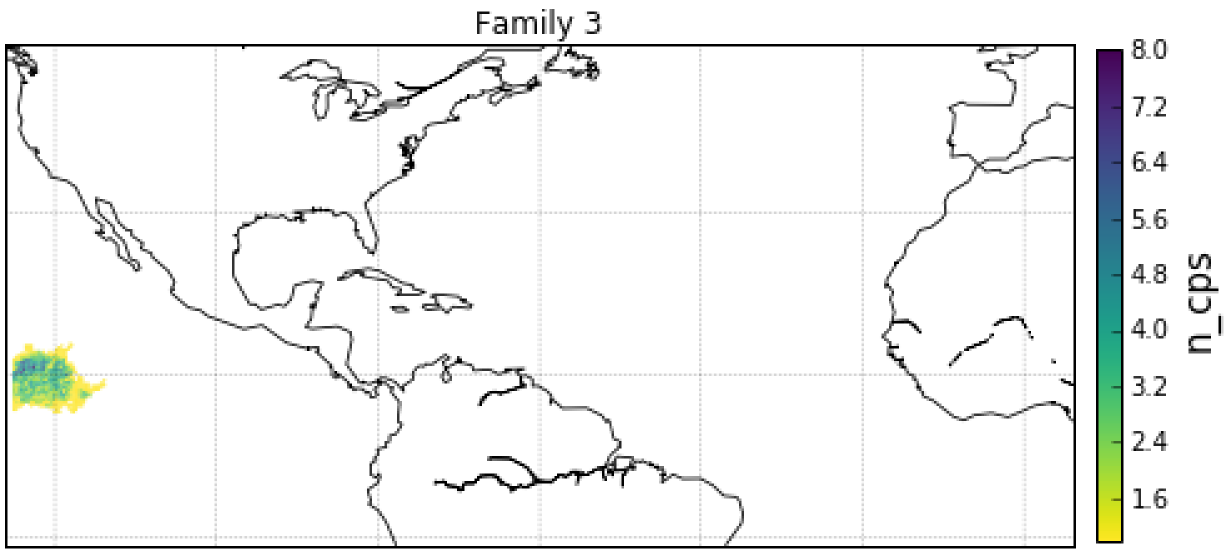
Plot Family Information

```
families = [0,1,2,3]

for F in families:

    # for easy filtering, we create a new DeepGraph instance for
    # each component
    gt = dg.DeepGraph(fgv.loc[F])

    # configure map projection
    kwds_basemap = {'llcrnrlon': v.lon.min() - 1,
                    'urcrnrlon': v.lon.max() + 1,
                    'llcrnrlat': v.lat.min() - 1,
                    'urcrnrlat': v.lat.max() + 1}

    # configure scatter plots
    kwds_scatter = {'s': 1,
                    'c': gt.v.n_cp_nodes.values,
                    'cmap': 'viridis_r',
                    'edgecolors': 'none'}

    # create scatter plot on map
    obj = gt.plot_map(
        lat='lat', lon='lon',
        kwds_basemap=kwds_basemap, kwds_scatter=kwds_scatter)

    # configure plots
    obj['m'].drawcoastlines(linewidth=.8)
    obj['m'].drawparallels(range(-50, 50, 20), linewidth=.2)
    obj['m'].drawmeridians(range(0, 360, 20), linewidth=.2)
    cb = obj['fig'].colorbar(obj['pc'], fraction=.022, pad=.02)
    cb.set_label('n_cps', fontsize=15)
    obj['ax'].set_title('Family {}'.format(F))
```


Family 0
8.0
7.2
6.4
5.6
4.8
4.0
3.2
2.4
1.6
n_cps


Family 1
6.4
5.6
4.8
4.0
3.2
2.4
1.6
n_cps





Geographical Locations and Components

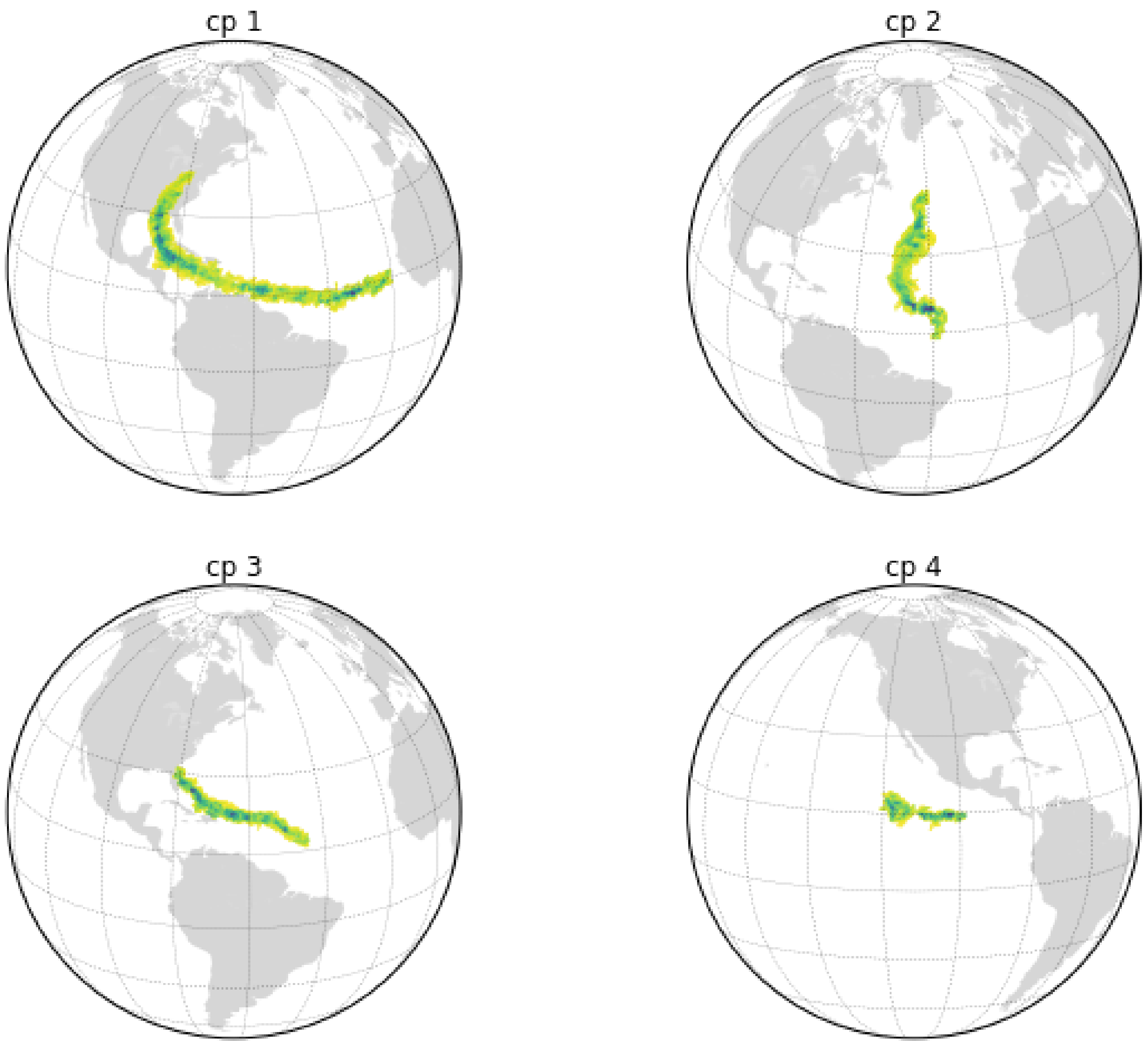
```
# feature functions, will be applied on each [g_id, cp] group of g
feature_funcs = {'vol': [np.sum],
                 'lat': np.min,
                 'lon': np.min}

# create gcpv
gcpv = g.partition_nodes(['cp', 'g_id'], feature_funcs)

gcpv.rename(columns={'lat_amin': 'lat',
                     'lon_amin': 'lon'}, inplace=True)
```

```
print(gcpv)
```

```
              n_nodes     lat  vol_sum      lon
cp    g_id
0     0             1 -10.125     5071 -125.125
      1             1  -9.875     4415 -125.125
      2             2  -9.625     4372 -125.125
      6             3  -8.375     1026 -125.125
      7             1  -8.125      594 -125.125
...               ...     ...      ...      ...
33167 112117        1   9.375    24618    0.625
33168 100613        1   6.625    11450  -13.625
      100614        1   6.875    12706  -13.625
33169 98523         1  15.375    31057  -16.125
      98524         1  15.625    15741  -16.125

[287301 rows x 4 columns]
```

Plot Component Information

```python
# select the components to plot
comps = [1, 2, 3, 4]

fig, axs = plt.subplots(2, 2, figsize=[10,8])
axs = axs.flatten()

for comp, ax in zip(comps, axs):

    # for easy filtering, we create a new DeepGraph instance for
    # each component
    gt = dg.DeepGraph(gcpv[gcpv.index.get_level_values('cp') == comp])

    # configure map projection
    kwds_basemap = {'projection': 'ortho',
                    'lon_0': cpv.loc[comp].lon_mean,
                    'lat_0': cpv.loc[comp].lat_mean,
                    'resolution': 'c'}

    # configure scatter plots
    kwds_scatter = {'s': .5,
                    'c': gt.v.vol_sum.values,
                    'cmap': 'viridis_r',
                    'edgecolors': 'none'}

    # create scatter plot on map
    obj = gt.plot_map(lon='lon', lat='lat',
                      kwds_basemap=kwds_basemap,
                      kwds_scatter=kwds_scatter,
                      ax=ax)

    # configure plots
    obj['m'].fillcontinents(color='0.2', zorder=0, alpha=.2)
    obj['m'].drawparallels(range(-50, 50, 20), linewidth=.2)
    obj['m'].drawmeridians(range(0, 360, 20), linewidth=.2)
    obj['ax'].set_title('cp {}'.format(comp))
```



1.3.4 From Multilayer Networks to Deep Graphs

`[ipython notebook][python script]`

In this tutorial we exemplify the representation of multilayer networks (MLNs) by deep graphs and demonstrate some of the advantages of deepgraph's network representation.

We start by converting the Noordin Top Terrorist MLN into a graph *g* - comprised of two DataFrames, a node table *g.v* and an edge table *g.e* - that corresponds to the supra-graph representation of the multilayer network.

We then partition the graph `g` by the information attributed to its layers, resulting in different supergraphs on the partition lattice of `g` that correpsond to different representations of a MLN (including its tensor representation).

In the next part, we demonstrate how additional information that might be at hand or computed during the analysis can be used to induce further supergraphs, or metaphorically speaking, how additional information corresponds to "hidden layers" of a MLN.

Finally, we briefly show how to use the nodes' properties to partition the edges of a MLN.

** **References** **

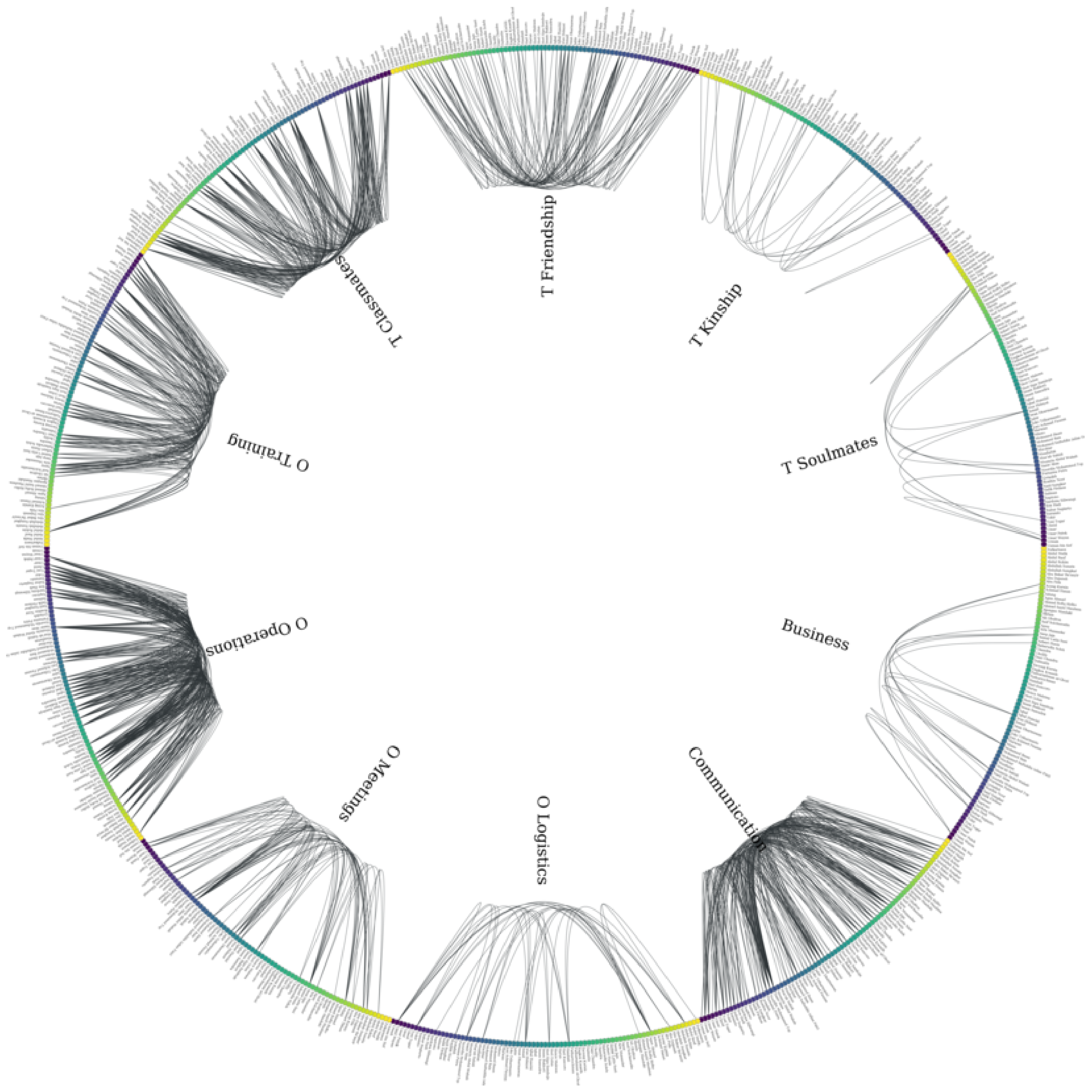
For a short summary of the multilayer network representation, see Appendix C of the Deep Graphs paper.

For a more in-depth introduction to MLNs, I recommend the following papers:

- Multilayer Networks (review paper of MLNs)
- The Structure and Dynamics of Multilayer Networks (review paper of MLNs)
- Mathematical Formulation of Multilayer Networks (tensor formalism for MLNs)

For a discussion of how Deep Graphs relates to the multilayer network representation, see Sec. IV B and Appendix D of the Deep Graphs paper.

The Noordin Top Terrorist Data



[high-res version][python plot script]

The data we use in this tutorial is the Noordin Top Terrorist Network, which has previously been represented as a multilayer network (e.g., http://arxiv.org/abs/1308.3182)

It includes relational data on 79 Indonesian terrorists belonging to the so-called Noordin Top Terrorist Network.

For information about the individual's attributes and their relations, see http://www.thearda.com/archive/files/codebooks/origCB/Noordin%20Subset%20Codebook.pdf and http://arxiv.org/pdf/1308.3182v3.pdf.

Preprocessing

We download the data from here, and process it into two pandas DataFrames, a node table and an edge table. The preprocessing is quite lengthy, so you might want to proceed directly to the *next section*.

First of all, we need to import some packages

```
# data i/o
import os
import subprocess
import zipfile

# for plots
import matplotlib.pyplot as plt

# the usual
import numpy as np
import pandas as pd

import deepgraph as dg

# notebook display
%matplotlib inline
pd.options.display.max_rows = 10
pd.set_option('expand_frame_repr', False)
```

Preprocessing the Nodes

```
# zip file containing node attributes
os.makedirs("tmp", exist_ok=True)
get_nodes_zip = ("wget -O tmp/terrorist_nodes.zip "
                 "https://sites.google.com/site/sfeverton18/"
                 "research/appendix-1/Noordin%20Subset%20%28ORA%29.zip?"
                 "attredirects=0&d=1")
subprocess.call(get_nodes_zip.split())

# unzip
zf = zipfile.ZipFile('tmp/terrorist_nodes.zip')
zf.extract('Attributes.csv', path='tmp/')
zf.close()

# create node table
v = pd.read_csv('tmp/Attributes.csv')
v.rename(columns={'Unnamed: 0': 'Name'}, inplace=True)

# create a copy of all nodes for each layer (i.e., create "node-layers")
```

(continues on next page)

(continued from previous page)

```
# there are 10 layers and 79 nodes on each layer
v = pd.concat(10*[v])

# add "aspect" as column to v
layer_names = ['Business', 'Communication', 'O Logistics', 'O Meetings',
               'O Operations', 'O Training', 'T Classmates', 'T Friendship',
               'T Kinship', 'T Soulmates']
layers = [[name]*79 for name in layer_names]
layers = [item for sublist in layers for item in sublist]
v['layer'] = layers

# set unique node index
v.reset_index(inplace=True)
v.rename(columns={'index': 'V_N'}, inplace=True)

# swap columns
cols = list(v)
cols[1], cols[10] = cols[10], cols[1]
v = v[cols]

# get rid of the attribute columns for demonstrational purposes,
# will be inserted again later
v, vinfo = v.iloc[:, :2], v.iloc[:, 2:]
```

Preprocessing the Edges

```
# paj file containing edges for different layers
get_paj = ("wget -O tmp/terrorists.paj "
           "https://sites.google.com/site/sfeverton18/"
           "research/appendix-1/Noordin%20Subset%20%28Pajek%29.paj?"
           "attredirects=0&d=1")
subprocess.call(get_paj.split())

# get data blocks from paj file
with open('tmp/terrorists.paj') as txtfile:
    comments = []
    data = []
    part = []
    for line in txtfile:
        if line.startswith('*'):
            # comment lines
            comment = line
            comments.append(comment)
            if part:
                data.append(part)
                part = []
        else:
            # vertices
            if comment.startswith('*Vertices') and len(line.split()) > 1:
                sublist = line.split('"')
                sublist = sublist[:2] + sublist[-1].split()
                part.append(sublist)
            # edges or partitions
            elif not line.isspace():
```

(continues on next page)

(continued from previous page)

```
                part.append(line.split())
    # append last block
    data.append(part)

# extract edge tables from data blocks
ecomments = []
eparts = []
for i, c in enumerate(comments):
    if c.startswith('*Network'):
        del data[0]
    elif c.startswith('*Partition'):
        del data[0]
    elif c.startswith('*Vector'):
        del data[0]
    elif c.startswith('*Arcs') or c.startswith('*Edges'):
        ecomments.append(c)
        eparts.append(data.pop(0))

# layer data parts (indices found manually via comments)
inds = [11, 10, 5, 6, 7, 8, 0, 1, 2, 3]
eparts = [eparts[ind] for ind in inds]

# convert to DataFrames
layer_frames = []
for name, epart in zip(layer_names, eparts):
    frame = pd.DataFrame(epart, dtype=np.int16)
    # get rid of self-loops, bidirectional edges
    frame = frame[frame[0] < frame[1]]
    # rename columns
    frame.rename(columns={0: 's', 1: 't', 2: name}, inplace=True)
    frame['s'] -= 1
    frame['t'] -= 1
    layer_frames.append(frame)

# set indices
for i, e in enumerate(layer_frames):
    e['s'] += i*79
    e['t'] += i*79
    e.set_index(['s', 't'], inplace=True)

# concat the layers
e = pd.concat(layer_frames)

# edge table as described in the paper
e_paper = e.copy()
```

```
# alternative representation of e
e['type'] = 0
e['weight'] = 0
for layer in layer_names:
    where = e[layer].notnull()
    e.loc[where, 'type'] = layer
    e.loc[where, 'weight'] = e.loc[where, layer]
e = e[['type', 'weight']]
```

DeepGraph's Supra-Graph Representation of a MLN, $G = (V, E)$

Above, we have processed the downloaded data into a node table v and an edge table e, that correspond to the supra-graph representation of a multilayer network. This is the preferred representation of a MLN by a deep graph, since all other representations are entailed in the supra-graph's partition lattice, as we will demonstrate below.

```
g = dg.DeepGraph(v, e)
print(g)
```

```
<DeepGraph object, with n=790 node(s) and m=1014 edge(s) at 0x7fb8e13499e8>
```

Let's have a look at the node table first

```
print(g.v)
```

```
     V_N       layer
0      0    Business
1      1    Business
2      2    Business
3      3    Business
4      4    Business
..   ...         ...
785   74  T Soulmates
786   75  T Soulmates
787   76  T Soulmates
788   77  T Soulmates
789   78  T Soulmates

[790 rows x 2 columns]
```

As you can see, there are 790 nodes in total. Each of the 10 layers,

```
print(g.v.layer.unique())
```

```
['Business' 'Communication' 'O Logistics' 'O Meetings' 'O Operations'
 'O Training' 'T Classmates' 'T Friendship' 'T Kinship' 'T Soulmates']
```

is comprised of 79 nodes. Every node has a feature of type V_N, indicating the individual the node belongs to, and a feature of type layer, corresponding to the layer the node belongs to. Each of the 790 nodes corresponds to a node-layer of the MLN representation of this data.

The edge table,

```
print(g.e)
```

```
                 type  weight
s   t
9   67       Business     2.0
    69       Business     1.0
    77       Business     1.0
11  61       Business     1.0
20  59       Business     1.0
...               ...     ...
733 769   T Soulmates     1.0
755 769   T Soulmates     1.0
    787   T Soulmates     1.0
```

(continues on next page)

(continued from previous page)

```
771 788   T Soulmates     1.0
783 788   T Soulmates     1.0

[1014 rows x 2 columns]
```

is comprised of 1014 edges between the nodes in `v`. Each edge has two relations. The first relation (of type `type`) is determined by the tuple of features $(layer_i, layer_j)$ of the adjacent nodes V_i and V_j. The second relation (of type `weight`) indicates the "weight" of the connection.

This representation of the edges of a MLN deviates from the one you can find in the paper, which is described in the *last section*.

There are 10 types of relations in the above edge table

```
g.e['type'].unique()
```

```
array(['Business', 'Communication', 'O Logistics', 'O Meetings',
       'O Operations', 'O Training', 'T Classmates', 'T Friendship',
       'T Kinship', 'T Soulmates'], dtype=object)
```

which - in the case of this data set - correspond to the layers of the nodes. This is due to the fact that there are no inter-layer connections in the Noordin Top Terrorist Network (such as, e.g., an edge from layer `Business` to layer `Communication` would be). The edges here are all (undirected) intra-layer edges (e.g., Business $\rightarrow$ Business, Operations $\rightarrow$ Operations).

To see how the edges are distributed among the different types, you can simply type

```
g.e['type'].value_counts()
```

```
O Operations     267
Communication    200
T Classmates     175
O Training       147
T Friendship      91
O Meetings        63
O Logistics       29
T Kinship         16
Business          15
T Soulmates       11
Name: type, dtype: int64
```

Let's have a look at how many "actors" (nodes with at least one connection) there are within each layer

```
# append degree
gtg = g.return_gt_graph()
g.v['deg'] = gtg.degree_property_map('total').a

# how many "actors" are there per layer?
g.v[g.v.deg != 0].groupby('layer').size()
```

```
layer
Business         13
Communication    74
O Logistics      16
O Meetings       26
O Operations     39
```

(continues on next page)

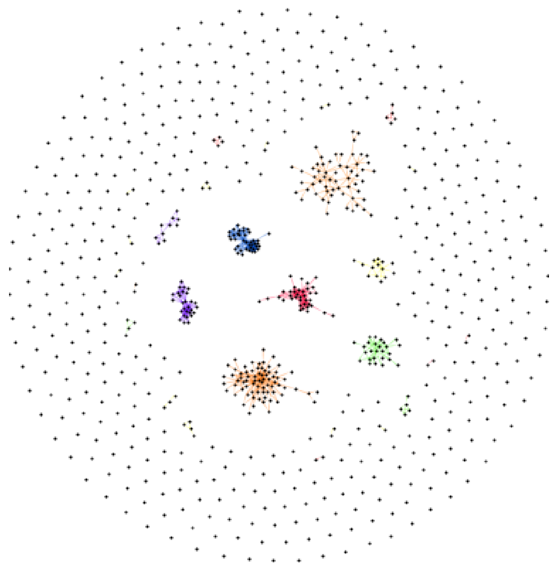
(continued from previous page)

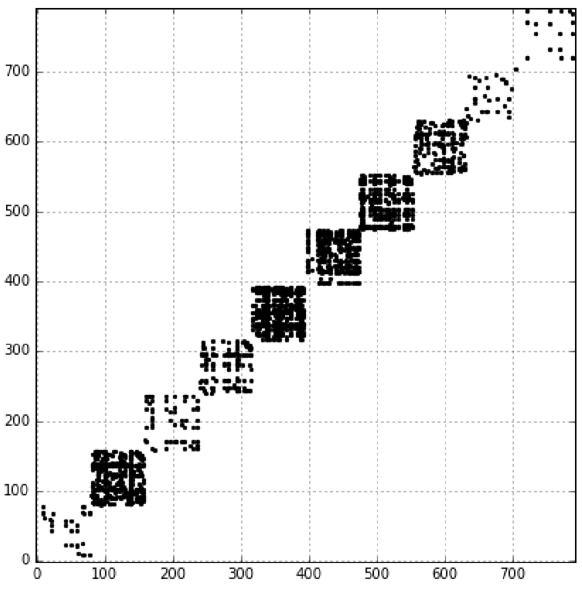
```
O Training       38
T Classmates     39
T Friendship     61
T Kinship        24
T Soulmates       9
dtype: int64
```

For the purpose of this tutorial, the fact that the Noordin Top Terrorist Network is a MLN with only one aspect, and without inter-layer edges, is of little importance. The generalization of what we're showing in the following to more general MLNs is straight-forward (and explained in detail in Appendix D of the paper).

Let's illustrate the supra-graph representation of this MLN by a plot

```
# create graph_tool graph for layout
import graph_tool.draw as gtd
gtg = g.return_gt_graph()
gtg.set_directed(False)

# get sfdp layout postitions
pos = gtd.sfdp_layout(gtg, gamma=.5)
pos = pos.get_2d_array([0, 1])
g.v['x'] = pos[0]
g.v['y'] = pos[1]

# configure nodes
kwds_scatter = {'s': 1,
                'c': 'k'}

# configure edges
kwds_quiver = {'headwidth': 1,
               'alpha': .3,
               'cmap': 'prism'}
# color by type
C = g.e.groupby('type').grouper.group_info[0]

# plot
fig, ax = plt.subplots(1, 2, figsize=(15, 7))
g.plot_2d('x', 'y', edges=True, C=C,
          kwds_scatter=kwds_scatter,
          kwds_quiver=kwds_quiver, ax=ax[0])

# turn axis off, set x/y-lim
ax[0].axis('off')
ax[0].set_xlim((g.v.x.min() - 1, g.v.x.max() + 1))
ax[0].set_ylim((g.v.y.min() - 1, g.v.y.max() + 1))

# plot adjacency matrix
adj = g.return_cs_graph().todense()
adj = adj + adj.T
inds = np.where(adj != 0)
ax[1].scatter(inds[0], inds[1], c='k', marker='.')
ax[1].grid()
ax[1].set_xlim(-1, 791)
ax[1].set_ylim(-1,791)
```





The supra-graph representation of a MLN is by itself a powerful representation and exploitable in various ways (see, e.g., section 2.3 of this paper). However, in the following, we will demonstrate how to use the additional information attributed to the layers of the MLN, in order to "structure" and partition the MLN into different representations.

Redistributing Information on the Partition Lattice of the MLN

Based on the types of features `V_N` and `layer`, we can now redistribute the information contained in the supra-graph `g`. This redistribution allows for several representations of the graph, which we will demonstrate in the following.

The SuperGraph $G^L = (V^L, E^L)$

Partitioning by the type of feature `layer` leads to the supergraph $G^L = (V^L, E^L)$, where every supernode $V^L_{i^L} \in V^L$ corresponds to a distinct layer, encompassing all its respective nodes. Superedges $E^L_{i^L,j^L} \in E^L$ with either $i^L = j^L$ or $i^L \neq j^L$ correspond to collections of intra- and inter-layer edges of the MLN, respectively.

```
# partition the graph
lv, le = g.partition_graph('layer',
                           relation_funcs={'weight': ['sum', 'mean', 'std']})
lg = dg.DeepGraph(lv, le)
print(lg)
```

```
<DeepGraph object, with n=10 node(s) and m=10 edge(s) at 0x7fb8e1349c50>
```

```
print(lg.v)
```

```
               n_nodes
layer
Business            79
Communication       79
O Logistics         79
O Meetings          79
O Operations        79
```

(continues on next page)

(continued from previous page)

```
O Training          79
T Classmates        79
T Friendship        79
T Kinship           79
T Soulmates         79
```

```
print(lg.e)
```

```
                             n_edges  weight_sum  weight_mean  weight_std
layer_s       layer_t
Business      Business            15        16.0     1.066667    0.258199
Communication Communication      200       200.0     1.000000    0.000000
O Logistics   O Logistics         29        58.0     2.000000    0.000000
O Meetings    O Meetings          63       170.0     2.698413    1.612801
O Operations  O Operations       267       574.0     2.149813    0.699107
O Training    O Training         147       334.0     2.272109    0.763534
T Classmates  T Classmates       175       175.0     1.000000    0.000000
T Friendship  T Friendship        91        91.0     1.000000    0.000000
T Kinship     T Kinship           16        16.0     1.000000    0.000000
T Soulmates   T Soulmates         11        11.0     1.000000    0.000000
```

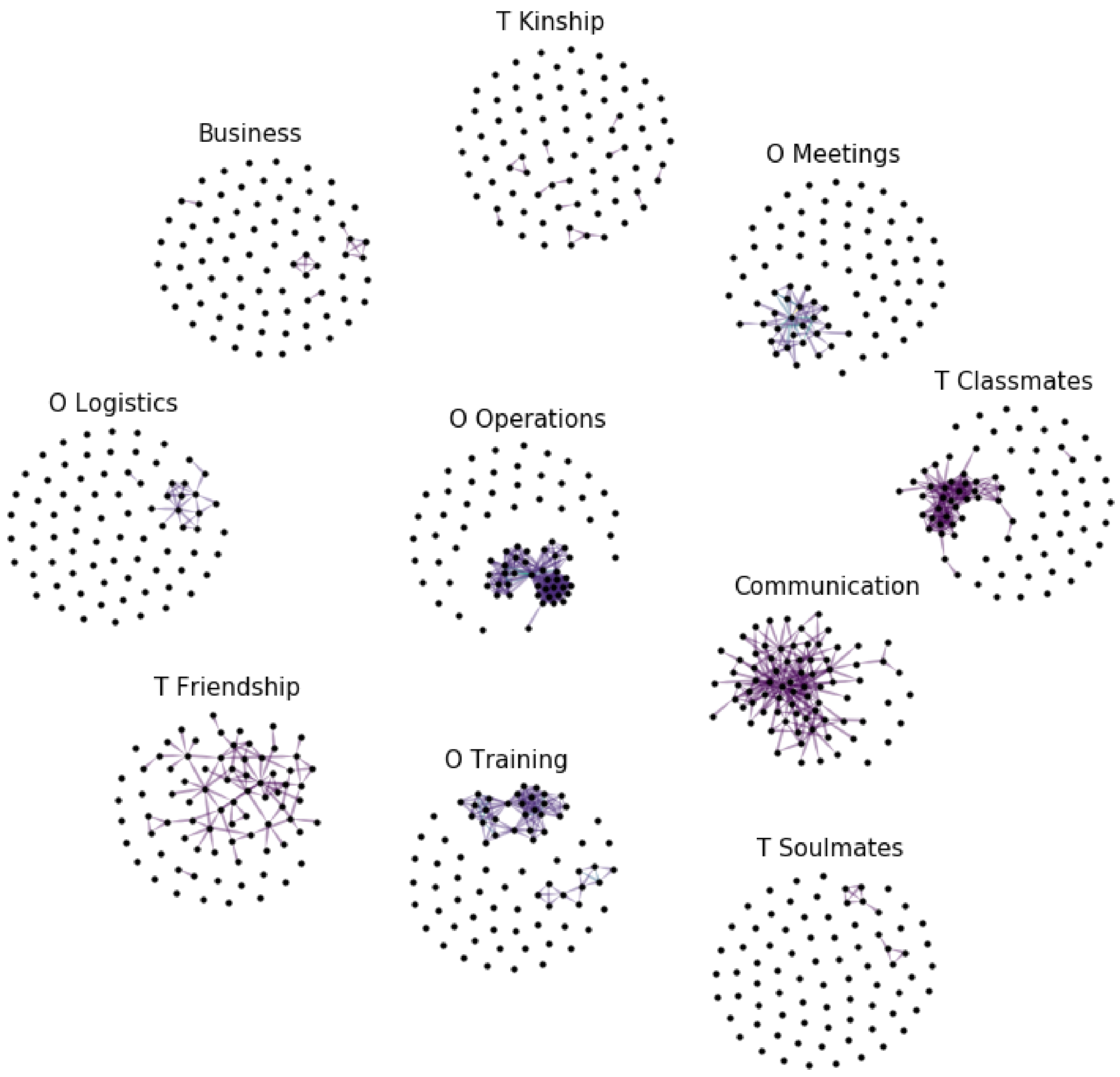
Let's plot the graph `g` grouped by its layers.

```
# append layer_id to group nodes by layers
g.v['layer_id'] = g.v.groupby('layer').grouper.group_info[0].astype(np.int32)

# create graph_tool graph object
gtg = g.return_gt_graph(features=['layer_id'])
gtg.set_directed(False)

# get sfdp layout postitions
pos = gtd.sfdp_layout(gtg, groups=gtg.vp['layer_id'], mu=.15)
pos = pos.get_2d_array([0, 1])
g.v['x'] = pos[0]
g.v['y'] = pos[1]

# configure nodes
kwds_scatter = {'s': 10,
                'c': 'k'}

# configure edges
kwds_quiver = {'headwidth': 1,
               'alpha': .4,
               'cmap': 'viridis'}
# color by weight
C = g.e.weight.values

# plot
fig, ax = plt.subplots(figsize=(12, 12))
obj = g.plot_2d('x', 'y', edges=True, C=C,
          kwds_scatter=kwds_scatter,
          kwds_quiver=kwds_quiver, ax=ax)

# turn axis off, set x/y-lim and name layers
ax.axis('off')
margin = 10
```

(continues on next page)

(continued from previous page)

```
ax.set_xlim((g.v.x.min() - margin, g.v.x.max() + margin))
ax.set_ylim((g.v.y.min() - margin, g.v.y.max() + margin))
for layer in layer_names:
    plt.text(g.v[g.v['layer'] == layer].x.mean() - margin * 3,
             g.v[g.v['layer'] == layer].y.max() + margin,
             layer, fontsize=15)
```



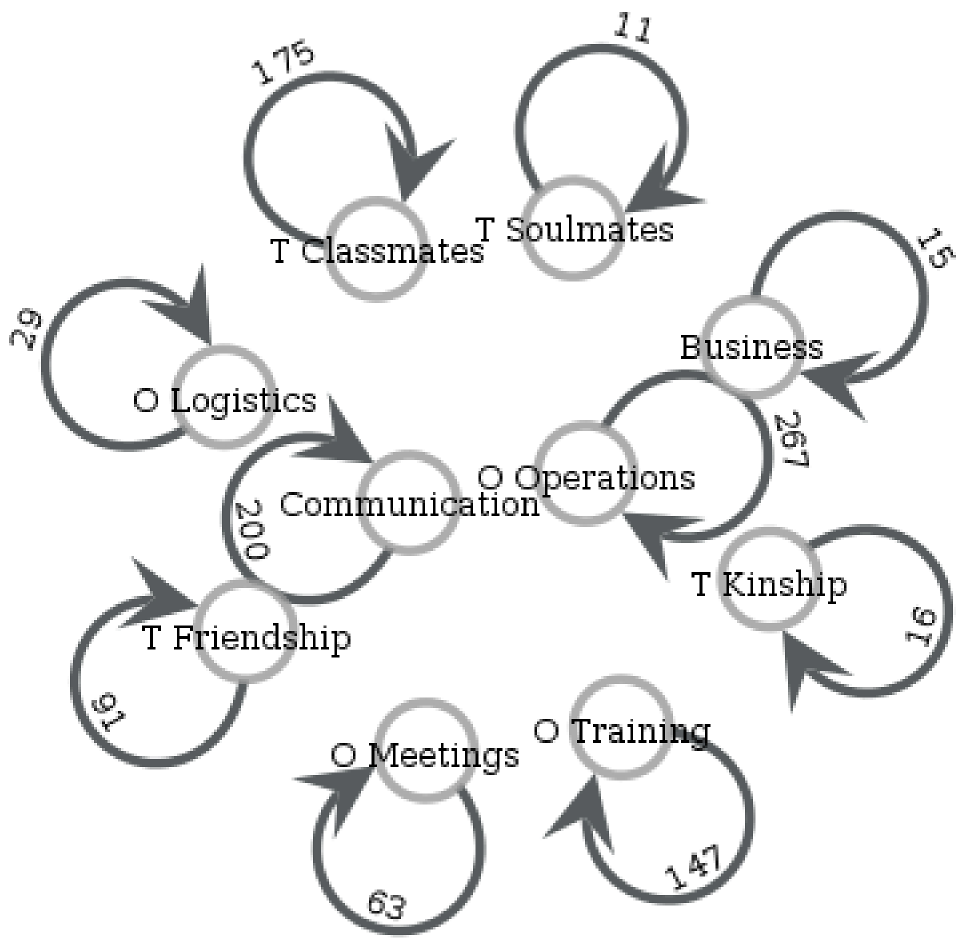
We can also plot the supergraph $G^L = (V^L, E^L)$

```
# create graph_tool graph of lg
gtg = lg.return_gt_graph(relations=True, node_indices=True, edge_indices=True)

# create plot
gtd.graph_draw(gtg,
               vertex_text=gtg.vp['i'], vertex_text_position=-2,
               vertex_fill_color='w',
               vertex_text_color='k',
```

(continues on next page)

(continued from previous page)

```
                 edge_text=gtg.ep['n_edges'],
                 inline=True, fit_view=.8,
                 output_size=(400,400))
```



The SuperGraph $G^N = (V^N, E^N)$

Partitioning by the type of feature V_N leads to the supergraph $G^N = (V^N, E^N)$, where each supernode $V^N_{i^N} \in V^N$ corresponds to a node of the MLN. Superedges $E^N_{i^N j^N} \in E^N$ with $i^N = j^N$ correspond to the coupling edges of a MLN.

```
# partition by MLN's node indices
nv, ne, gv, ge = g.partition_graph('V_N', return_gve=True)

# for each superedge, get types of edges and their weights
def type_weights(group):
    index = group['type'].values
    data = group['weight'].values
```

(continues on next page)

(continued from previous page)

```
    return pd.Series(data=data, index=index)
ne_weights = ge.apply(type_weights).unstack()
ne = pd.concat((ne, ne_weights), axis=1)

# create graph
ng = dg.DeepGraph(nv, ne)
ng
```

```
<DeepGraph object, with n=79 node(s) and m=623 edge(s) at 0x7fb8d1da8b70>
```

```
print(ng.v)
```

```
     n_nodes
V_N
0         10
1         10
2         10
3         10
4         10
..       ...
74        10
75        10
76        10
77        10
78        10

[79 rows x 1 columns]
```

```
print(ng.e)
```

```
             n_edges  Business  Communication  O Logistics  O Meetings  O Operations ␣
↪O Training  T Classmates  T Friendship  T Kinship  T Soulmates
V_N_s V_N_t
0     15           3       NaN            1.0          2.0         NaN           NaN ␣
↪       NaN           NaN           NaN        1.0          NaN
1     4            1       NaN            NaN          NaN         NaN           NaN ␣
↪       NaN           1.0           NaN        NaN          NaN
      5            1       NaN            NaN          NaN         NaN           NaN ␣
↪       NaN           1.0           NaN        NaN          NaN
      16           1       NaN            NaN          NaN         NaN           2.0 ␣
↪       NaN           NaN           NaN        NaN          NaN
      21           1       NaN            NaN          NaN         NaN           NaN ␣
↪       NaN           1.0           NaN        NaN          NaN
...              ...       ...            ...          ...         ...           ... ␣
↪       ...           ...           ...        ...          ...
72    73           4       NaN            1.0          NaN         NaN           2.0 ␣
↪       2.0           NaN           NaN        1.0          NaN
      76           6       NaN            1.0          NaN         2.0           2.0 ␣
↪       2.0           1.0           1.0        NaN          NaN
      77           2       NaN            NaN          2.0         NaN           NaN ␣
↪       NaN           NaN           NaN        NaN          1.0
73    76           2       NaN            NaN          NaN         NaN           2.0 ␣
↪       2.0           NaN           NaN        NaN          NaN
75    78           2       NaN            NaN          NaN         NaN           NaN ␣
↪       2.0           NaN           1.0        NaN          NaN
```

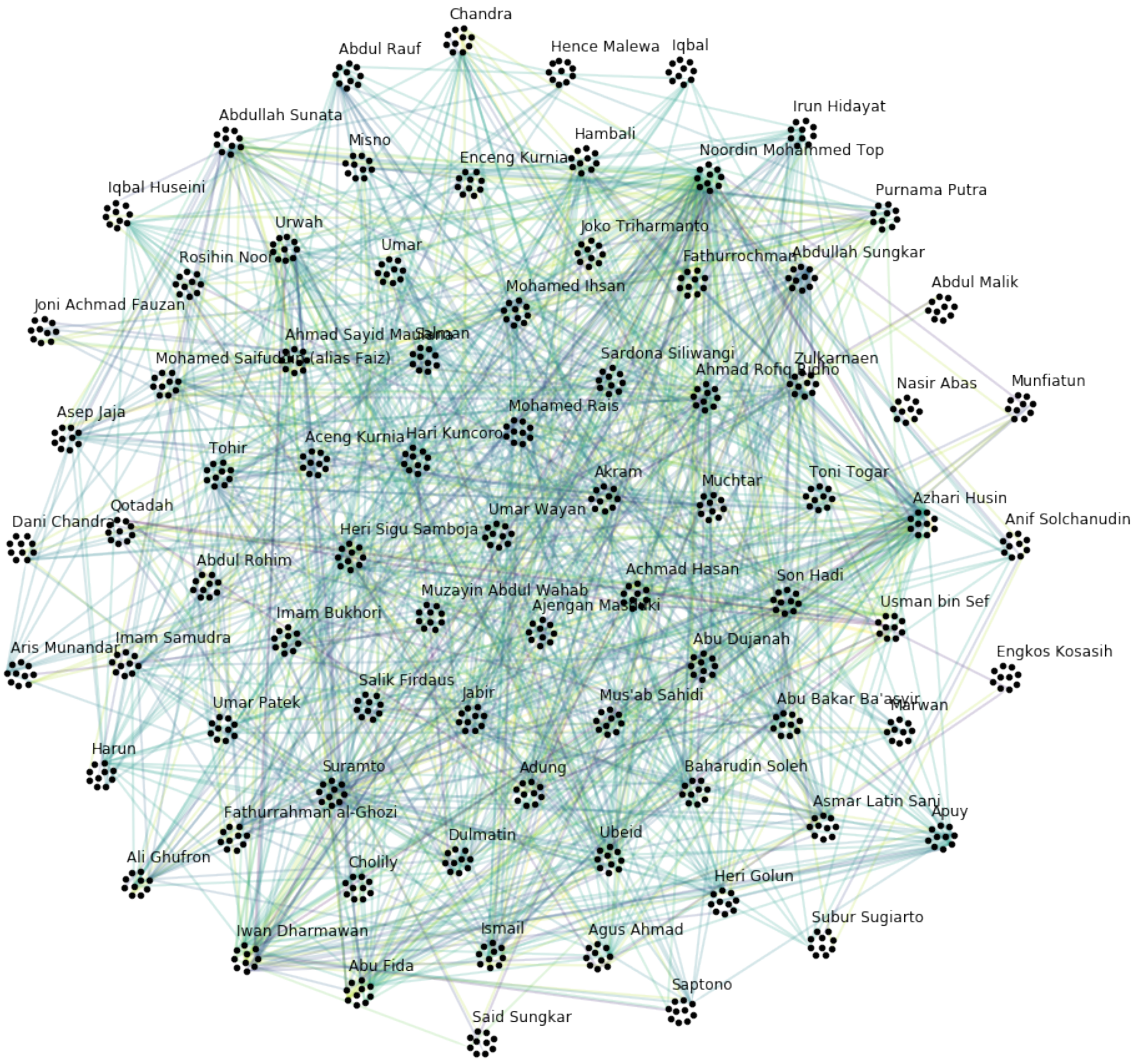
(continues on next page)

(continued from previous page)

```
[623 rows x 11 columns]
```

Let's plot the graph g grouped by V_N.

```
# create graph_tool graph object
g.v['V_N'] = g.v['V_N'].astype(np.int32)  # sfpd only takes int32
g_tmp = dg.DeepGraph(v)
gtg = g_tmp.return_gt_graph(features='V_N')
gtg.set_directed(False)

# get sfdp layout postitions
pos = gtd.sfdp_layout(gtg, groups=gtg.vp['V_N'], mu=.3, gamma=.01)
pos = pos.get_2d_array([0, 1])
g.v['x'] = pos[0]
g.v['y'] = pos[1]

# configure nodes
kwds_scatter = {'c': 'k'}

# configure edges
kwds_quiver = {'headwidth': 1,
               'alpha': .2,
               'cmap': 'viridis_r'}
# color by type
C = g.e.groupby('type').grouper.group_info[0]

# plot
fig, ax = plt.subplots(figsize=(15,15))
g.plot_2d('x', 'y', edges=True,
          kwds_scatter=kwds_scatter, C=C,
          kwds_quiver=kwds_quiver, ax=ax)

# turn axis off, set x/y-lim and name nodes
name_dic = {i: name for i, name in enumerate(vinfo.iloc[:79].Name)}
ax.axis('off')
ax.set_xlim((g.v.x.min() - 1, g.v.x.max() + 1))
ax.set_ylim((g.v.y.min() - 1, g.v.y.max() + 1))
for node in g.v['V_N'].unique():
    plt.text(g.v[g.v['V_N'] == node].x.mean() - 1,
             g.v[g.v['V_N'] == node].y.max() + 1,
             name_dic[node], fontsize=12)
```



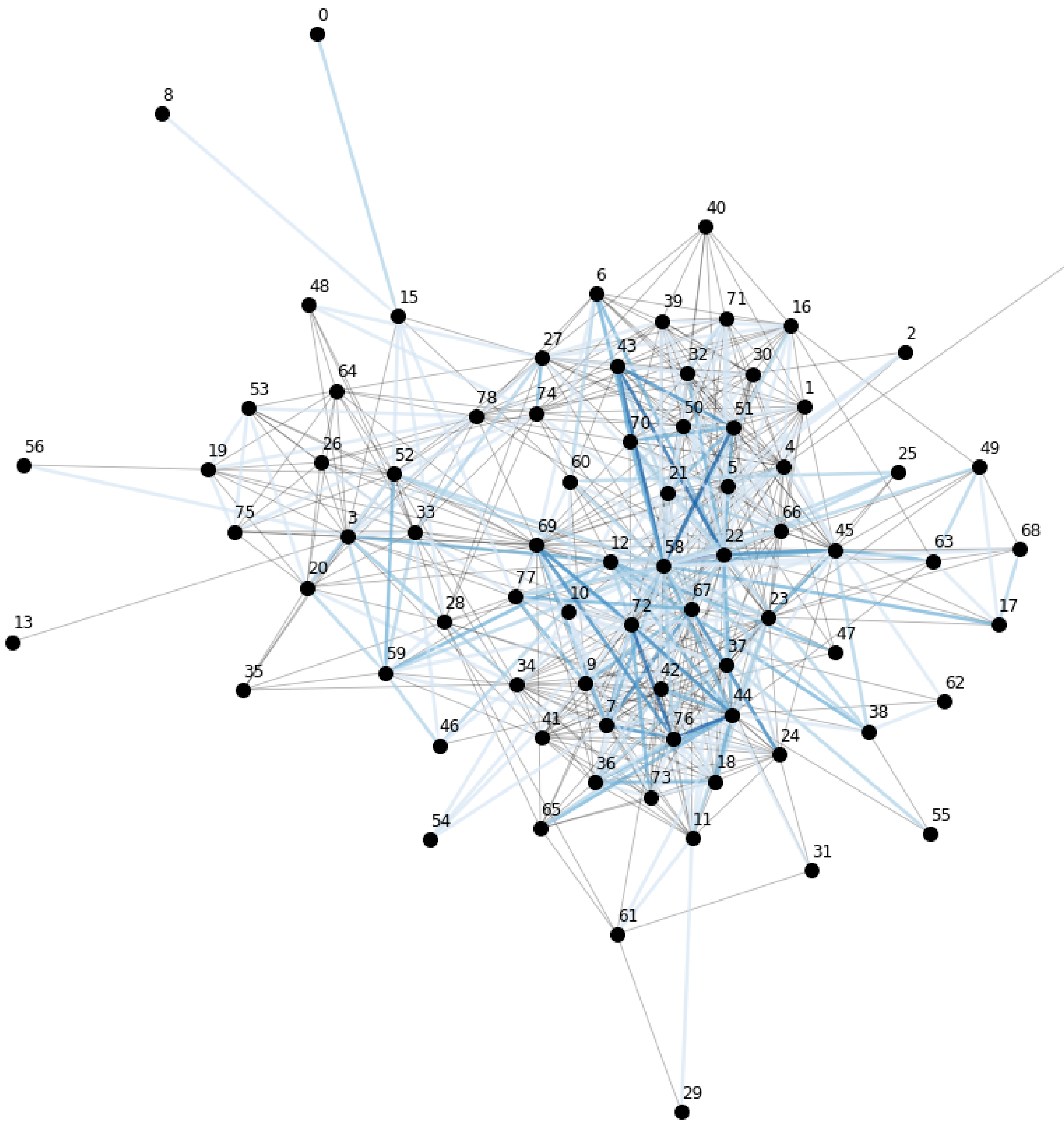
Let's also plot the supergraph $G^N = (V^N, E^N)$, where the color of the superedges corresponds to the number of edges within the respective superedge.

```
# get rid of isolated node for nicer layout
ng.v.drop(57, inplace=True, errors='ignore')

# create graph_tool graph object
gtg = ng.return_gt_graph(features=True, relations='n_edges')
gtg.set_directed(False)

# get sfdp layout postitions
pos = gtd.sfdp_layout(gtg)
pos = pos.get_2d_array([0, 1])
ng.v['x'] = pos[0]
ng.v['y'] = pos[1]

# configure nodes
kwds_scatter = {'s': 100,
```

(continues on next page)

(continued from previous page)

```
                'c': 'k'}

# configure edges
# split edges with only one type of connection
C_split_0 = ng.e['n_edges'].values.copy()
C_split_0[C_split_0 == 1] = 0

# edges with one type of connection
kwds_quiver_0 = {'alpha': .3,
                 'width': .001}

# edges with more than one type
kwds_quiver = {'headwidth': 1,
               'width': .003,
               'alpha': .7,
               'cmap': 'Blues',
               'clim': (1, ng.e.n_edges.max())}

# create plot
fig, ax = plt.subplots(figsize=(15,15))
ng.plot_2d('x', 'y', edges=True, C_split_0=C_split_0,
           kwds_scatter=kwds_scatter, kwds_quiver_0=kwds_quiver_0,
           kwds_quiver=kwds_quiver, ax=ax)

# turn axis off, set x/y-lim and name nodes
ax.axis('off')
ax.set_xlim(ng.v.x.min() - 1, ng.v.x.max() + 1)
ax.set_ylim(ng.v.y.min() - 1, ng.v.y.max() + 1)
for i in ng.v.index:
    plt.text(ng.v.at[i, 'x'], ng.v.at[i, 'y'] + .3, i, fontsize=12)
```



The Tensor-Like Representation $G^{NL} = (V^{NL}, E^{NL})$

Considering only the information attributed to the layers of the MLN, and the fact that this MLN has just one aspect, there is only one more supergraph we can create of `g`. It is given by creating the intersection partition (see section III E of the Deep Graphs paper) of the types of features `V_N` and `layer`. The resulting supergraph $G^{N \cdot L} = (V^{N \cdot L}, E^{N \cdot L})$ corresponds one to one to the graph $G = (V, E)$, and therefore to the supra-graph representation of the MLN. The only difference is the indexing, which is tensor-like for the supergraph $G^{N \cdot L}$.

```
# partition the graph
relation_funcs = {'type': 'sum', 'weight': 'sum'}  # just to transfer relations
nlv, nle = g.partition_graph(['V_N', 'layer'], relation_funcs=relation_funcs)
nlg = dg.DeepGraph(nlv, nle)
nlg
```

```
<DeepGraph object, with n=790 node(s) and m=1014 edge(s) at 0x7fb8d5325550>
```

```
print(nlg.v)
```

```
                   n_nodes
V_N layer
0   Business             1
    Communication        1
    O Logistics          1
    O Meetings           1
    O Operations         1
...                    ...
78  O Training           1
    T Classmates         1
    T Friendship         1
    T Kinship            1
    T Soulmates          1

[790 rows x 1 columns]
```

```
print(nlg.e)
```

```
                                                n_edges  weight           type
V_N_s layer_s       V_N_t layer_t
0     Communication 15    Communication               1     1.0  Communication
      O Logistics   15    O Logistics                 1     2.0    O Logistics
      T Kinship     15    T Kinship                   1     1.0      T Kinship
1     O Operations  16    O Operations                1     2.0   O Operations
                    22    O Operations                1     2.0   O Operations
...                                                 ...     ...            ...
72    T Soulmates   77    T Soulmates                 1     1.0    T Soulmates
73    O Operations  76    O Operations                1     2.0   O Operations
      O Training    76    O Training                  1     2.0     O Training
75    O Training    78    O Training                  1     2.0     O Training
      T Friendship  78    T Friendship                1     1.0   T Friendship

[1014 rows x 3 columns]
```

This tensor-like index allows you to use the advanced indexing features of pandas.

```
print(nlg.e.loc[2, 'Communication', :, 'Communication'])
```

```
                                                n_edges  weight           type
V_N_s layer_s       V_N_t layer_t
2     Communication 5     Communication               1     1.0  Communication
                    12    Communication               1     1.0  Communication
                    30    Communication               1     1.0  Communication
                    58    Communication               1     1.0  Communication
```

In the future, we might implement a method to convert this tensor-representation of a MLN to some sparse-tensor data structure (e.g., https://github.com/mnick/scikit-tensor). Another idea is to create an interface to a suitable multilayer network package that implements the measures and models developed particularly for MLNs.

The “Hidden Layers” of a MLN

Partitioning a multilayer network solely based on the information attributed to its layers only gets us this far. If there is more information available, or computed during the analysis [e.g., by statistical measures, network measures or similarity/distance measures (see *g.create_edges*)], it can be used to induce further supergraphs and reach other elements of the partition lattice of g.

This is what we’ll demonstrate here, based on the additional information available about the individual’s attributes:

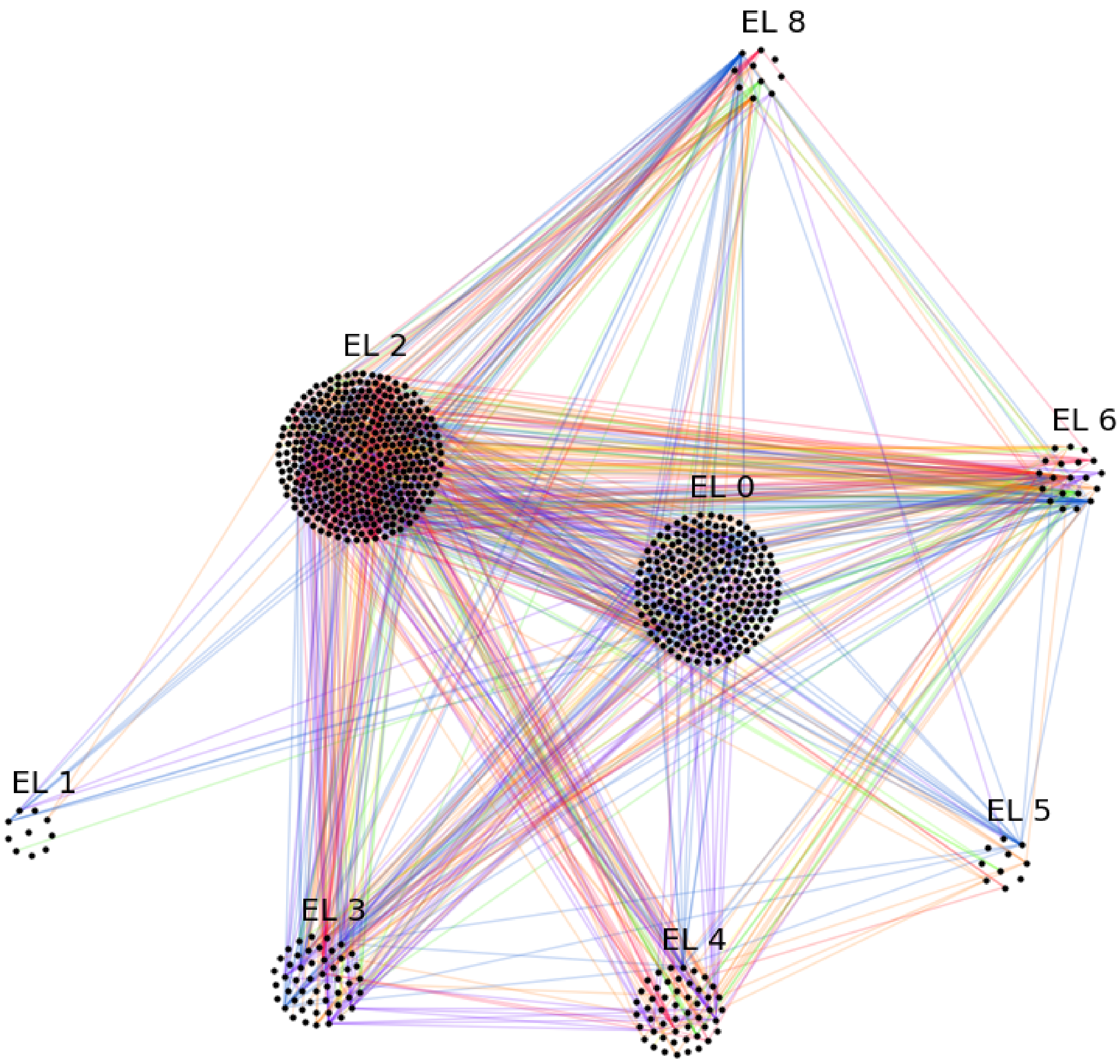
```
print(vinfo)
```

```
     Education Level  Contact with People  Military Training  Nationality  Current␣
↪Status (ICG Article)  Role  Primary Group Affiliation  Noordin's Network         ␣
↪  Name
0                   0                      5                   0            3         ␣
↪                   1     7                          1                   0       Abdul␣
↪Malik
1                   2                      3                   0            3          ␣
↪                   2    10                          1                   0        Abdul␣
↪Rauf
2                   0                     10                   0            3         ␣
↪                   1     9                          0                   0       Abdul␣
↪Rohim
3                   3                      5                   3            3         ␣
↪                   2     1                          2                   0    Abdullah␣
↪Sunata
4                   2                      3                   0            3         ␣
↪                   0     1                          3                   0   Abdullah␣
↪Sungkar
..                ...                    ...                 ...          ...         ␣
↪                 ...   ...                        ...                 ...            ␣
↪ ...
785                 2                     12                   5            3         ␣
↪                   1     3                          3                   1        Umar␣
↪Patek
786                 2                      1                   7            3         ␣
↪                   2     4                          3                   0       Umar␣
↪Wayan
787                 2                      3                   3            3         ␣
↪                   2     7                          3                   1           ␣
↪Urwah
788                 2                     11                   3            3         ␣
↪                   2    10                          3                   1     Usman␣
↪bin Sef
789                 2                      1                   7            4         ␣
↪                   1     1                          3                   0       ␣
↪Zulkarnaen

[790 rows x 9 columns]
```

As you can see, there are 9 different attributes associated with each individual, such as their military training, nationality, education level, etc. Let’s append this information to the node table, and plot the nodes grouped by their education level.

```
# append node information to g
v = pd.concat((v, vinfo), axis=1)
g = dg.DeepGraph(v, e)
```

```
# create graph_tool graph object
g.v['Education Level'] = g.v['Education Level'].astype(np.int32)
g_tmp = dg.DeepGraph(g.v)
gtg = g_tmp.return_gt_graph(features=['Education Level'])
gtg.set_directed(False)

# get sfdp layout postitions
pos = gtd.sfdp_layout(gtg, groups=gtg.vp['Education Level'], mu=.3, gamma=.1)
pos = pos.get_2d_array([0, 1])
g.v['x'] = pos[0]
g.v['y'] = pos[1]

# configure nodes
kwds_scatter = {'s': 10,
                'c': 'k'}

# configure edges
kwds_quiver = {'width': 0.002,
               'headwidth': 1,
               'alpha': .2,
               'cmap': 'prism'}
# color by type
C = g.e.groupby('type').grouper.group_info[0]

# plot
fig, ax = plt.subplots(figsize=(13,12))
obj = g.plot_2d('x', 'y', edges=True,
          kwds_scatter=kwds_scatter, C=C,
          kwds_quiver=kwds_quiver, ax=ax)

# turn axis off, set x/y-lim and name layers
ax.axis('off')
ax.set_xlim((g.v.x.min() - 1, g.v.x.max() + 1))
ax.set_ylim((g.v.y.min() - 1, g.v.y.max() + 1))
for el in g.v['Education Level'].unique():
    plt.text(g.v[g.v['Education Level'] == el].x.mean() - 1,
             g.v[g.v['Education Level'] == el].y.max() + 1,
             'EL {}'.format(el), fontsize=20)
```



Let's also append the information to the supergraph G^N, and plot this supergraph grouped by education level.

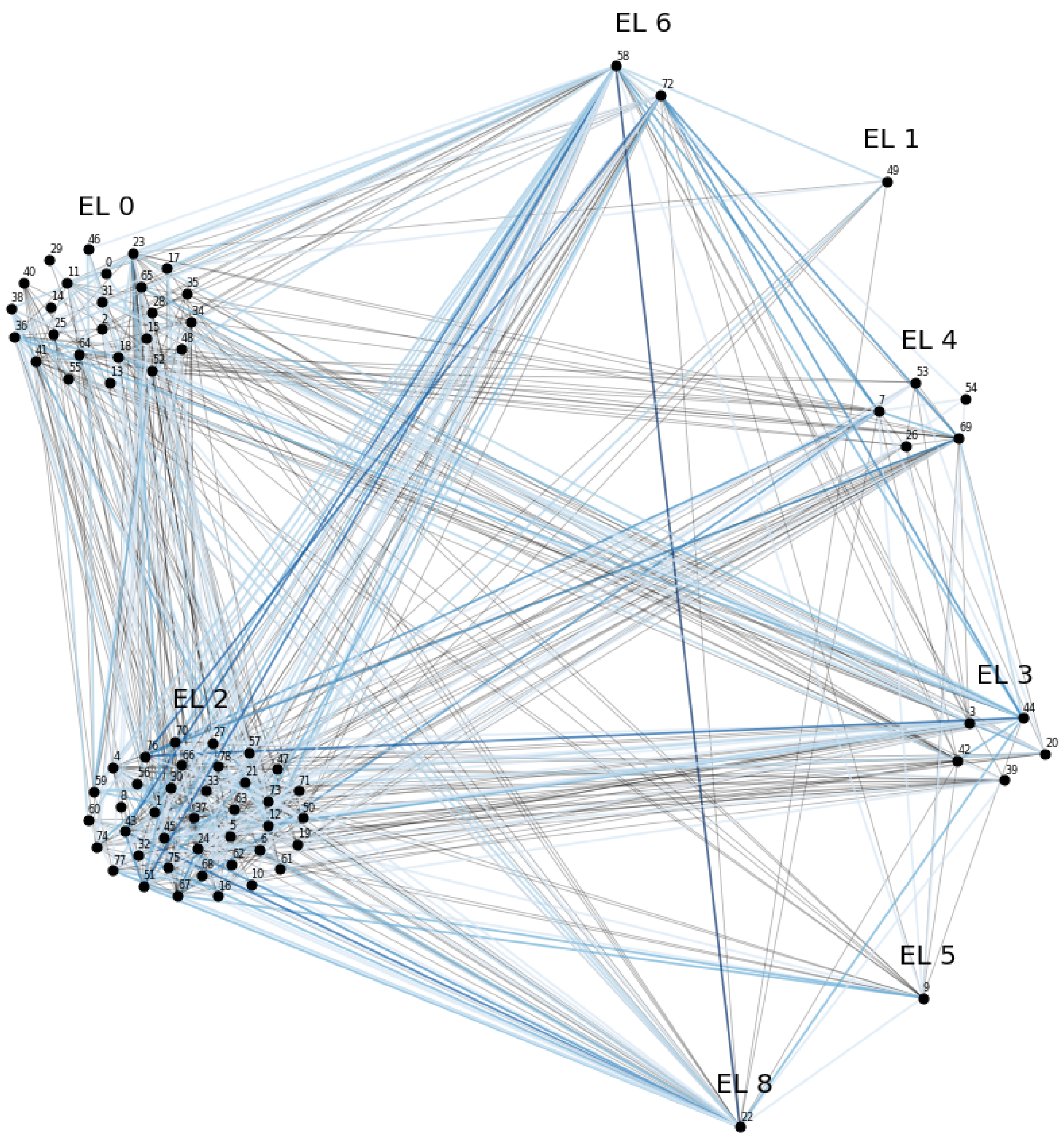
```
# append info to ng.v
ng.v = pd.concat((ng.v, vinfo[:79]), axis=1)
```

```
# create graph_tool graph object
ng.v['Education Level'] = ng.v['Education Level'].astype(np.int32)
g_tmp = dg.DeepGraph(ng.v)
gtg = g_tmp.return_gt_graph(features=['Education Level'])
gtg.set_directed(False)

# get sfdp layout postitions
pos = gtd.sfdp_layout(gtg, groups=gtg.vp['Education Level'], mu=.3, gamma=.01)
pos = pos.get_2d_array([0, 1])
ng.v['x'] = pos[0]
ng.v['y'] = pos[1]
```

(continues on next page)

(continued from previous page)

```
# configure nodes
kwds_scatter = {'s': 50,
                'c': 'k'}

# configure edges
# split edges with only one type of connection
C_split_0 = ng.e['n_edges'].values.copy()
C_split_0[C_split_0 == 1] = 0

# edges with one type of connection
kwds_quiver_0 = {'alpha': .3,
                 'width': .001}

# edges with more than one type
kwds_quiver = {'headwidth': 1,
               'width': .002,
               'alpha': .7,
               'cmap': 'Blues',
               'clim': (1, ng.e.n_edges.max())}

# create plot
fig, ax = plt.subplots(figsize=(15,15))
obj = ng.plot_2d('x', 'y', edges=True, C_split_0=C_split_0,
                 kwds_scatter=kwds_scatter, kwds_quiver_0=kwds_quiver_0,
                 kwds_quiver=kwds_quiver, ax=ax)

# turn axis off, set x/y-lim and name nodes
ax.axis('off')
ax.set_xlim(ng.v.x.min() - 1, ng.v.x.max() + 1)
ax.set_ylim(ng.v.y.min() - 1, ng.v.y.max() + 1)
for i in ng.v.index:
    plt.text(ng.v.at[i, 'x'],
             ng.v.at[i, 'y'] + .2,
             i, fontsize=8)

for el in ng.v['Education Level'].unique():
    plt.text(ng.v[ng.v['Education Level'] == el].x.mean() - .5,
             ng.v[ng.v['Education Level'] == el].y.max() + 1,
             'EL {}'.format(el), fontsize=20)
```



We can now further partition the supergraph G^N into groups with the same education level.

```
# partition ng by "Education Level"
relation_funcs = {l: lambda x: x.notnull().sum() for l in layer_names}
relation_funcs['n_edges'] = 'sum'
ELnv, ELne = ng.partition_graph('Education Level',
                                relation_funcs=relation_funcs,
                                n_edges=False)

# compute "undirected" weights
s = ELne.index.get_level_values(0)
t = ELne.index.get_level_values(1)
df1 = ELne[s <= t]
df2 = ELne[s > t].swaplevel(0,1)
```

(continues on next page)

(continued from previous page)

```
df2.index.names = df2.index.names[::-1]
ELne = df1.add(df2, fill_value=0)

# set dtypes
for col in ELne.columns:
    ELne[col] = ELne[col].astype(int)

# find the type of connection most dominant between supernodes
ELne['dominant_type'] = ELne[layer_names].idxmax(axis=1)

# change column order
ELne = ELne[['n_edges'] + ['dominant_type'] + layer_names]

# create graph
ELng = dg.DeepGraph(ELnv, ELne)
ELng
```

```
<DeepGraph object, with n=8 node(s) and m=30 edge(s) at 0x7fb8d1d245c0>
```

```
print(ELng.v)
```

```
                 n_nodes
Education Level
0                     25
1                      1
2                     39
3                      5
4                      5
5                      1
6                      2
8                      1
```

```
print(ELng.e)
```

```
                                    n_edges  dominant_type  Business  Communication ␣
↪O Logistics  O Meetings  O Operations  O Training  T Classmates  T Friendship  T␣
↪Kinship  T Soulmates
Education Level_s Education Level_t
0                 0                      45   O Operations         0              7 ␣
↪           2           1            16           15             1             1  ␣
↪   2           0
                  1                       3   O Operations         0              0 ␣
↪           0           0             2            1             0             0  ␣
↪   0           0
                  2                     146   O Operations         1             31 ␣
↪           3           7            43           32             9            16  ␣
↪   4           0
                  3                      60     O Training         0             11 ␣
↪           2           2            14           19             2             9  ␣
↪   1           0
                  4                      16     O Training         0              0 ␣
↪           0           0             6            9             1             0  ␣
↪   0           0
...                                     ...            ...       ...            ... ␣
↪         ...         ...           ...          ...           ...           ...  ␣
↪  ...         ...
```

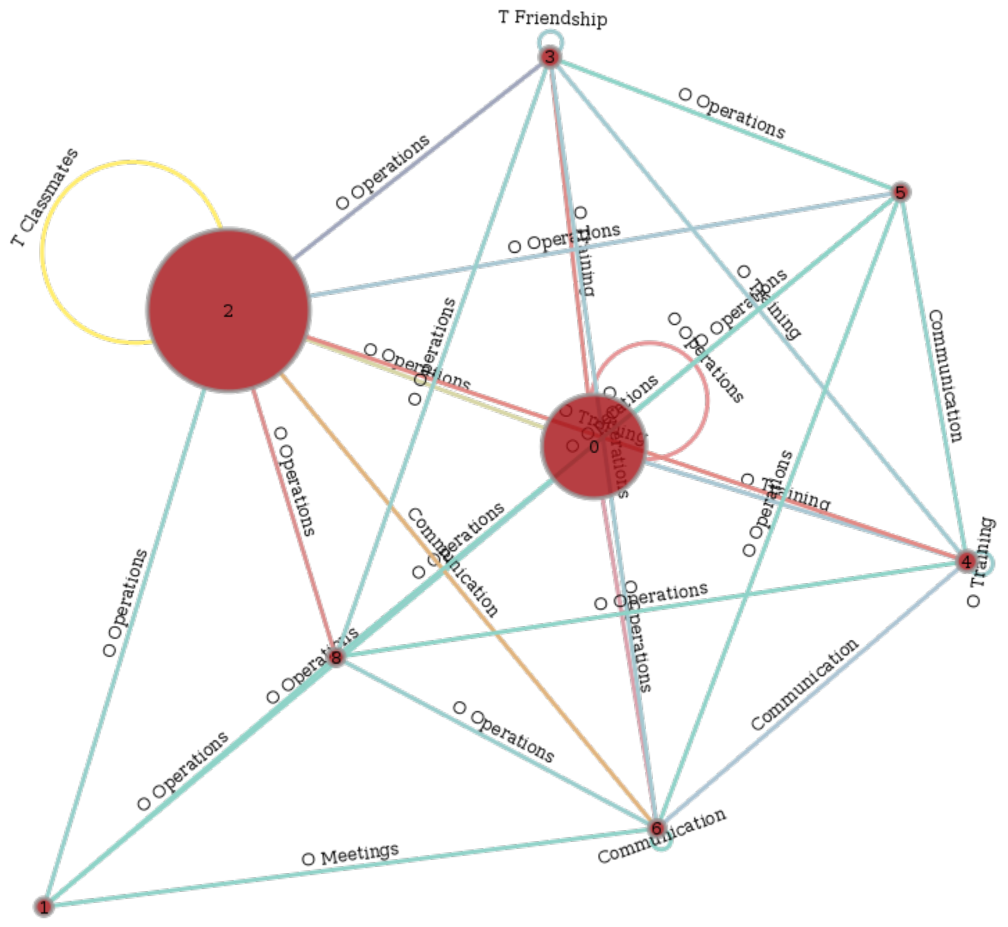
(continues on next page)

(continued from previous page)

```
4                 8                      1   O Operations         0              0 ␣
↪           0           0             1           0             0             0     ␣
↪    0            0
5                 6                      3   O Operations         0              1 ␣
↪           0           0             2           0             0             0     ␣
↪    0            0
                  8                      2   O Operations         0              0 ␣
↪           0           0             1           1             0             0     ␣
↪    0            0
6                 6                      3  Communication         0              1 ␣
↪           0           1             1           0             0             0     ␣
↪    0            0
                  8                      8   O Operations         1              1 ␣
↪           0           1             2           0             1             1     ␣
↪    0            1

[30 rows x 12 columns]
```

Let's plot the supergraph of education levels, where the node size relates to the number of individuals, edge colors correspond to the number of edges, and edge labels correspond to the most dominant type of connection between nodes.

```
# create graph_tool graph object
gtg = ELng.return_gt_graph(features=True, relations=True, node_indices=True)
gtg.set_directed(False)

# get sfdp layout postitions
pos = gtd.sfdp_layout(gtg, vweight=gtg.vp['n_nodes'], eweight=gtg.ep['n_edges'])
pos = pos.get_2d_array([0, 1])

# create plot
gtg.vp['n_nodes'].a *= 3
gtd.graph_draw(gtg,
               vertex_text=gtg.vp['i'],
               vertex_text_color='k', vertex_size=gtg.vp['n_nodes'],
               edge_text=gtg.ep['dominant_type'],
               edge_color=gtg.ep['n_edges'],
               inline=True, output_size=(900,900), fit_view=True)
```



Partitioning Edges Based on Node Properties

Here, we demonstrate very briefly how to use the additional information of the nodes to perform queries on the edges.

```
# create "undirected" edge table (swap-copy all edges)
g.e = pd.concat((e, e.swaplevel(0,1)))
g.e.sort_index(inplace=True)
```

```
print(g.partition_edges(source_features=['Nationality']))
```

```
               n_edges
Nationality_s
```

(continues on next page)

(continued from previous page)

```
3                 1655
4                  351
5                   22
```

```
print(g.partition_edges(source_features=['Nationality'], target_features=['Military␣
↪Training']))
```

```
                                           n_edges
Nationality_s Military Training_t
3             0                                185
              1                                 51
              3                                847
              4                                 60
              5                                115
...                                            ...
5             4                                  3
              5                                  1
              7                                  1
              9                                  1
              10                                 1

[26 rows x 1 columns]
```

```
print(g.partition_edges(source_features=['Nationality'],
                        target_features=['Military Training'],
                        relations='type'))
```

```
                                                       n_edges
type        Nationality_s Military Training_t
Business    3             0                                  3
                          3                                 16
                          4                                  1
                          9                                  2
                          10                                 2
...                                                        ...
T Soulmates 3             9                                  1
                          10                                 2
            4             3                                  3
                          9                                  1
                          10                                 3

[138 rows x 1 columns]
```

Alternative Representation of the MLN Edges

The edges of the supra-graph representation as presented in the paper look like this

```
print(e_paper)
```

```
         Business  Communication  O Logistics  O Meetings  O Operations  O Training ␣
↪T Classmates  T Friendship  T Kinship  T Soulmates
s   t
9   67        2.0            NaN          NaN         NaN           NaN         NaN  ␣
↪        NaN           NaN        NaN          NaN
```

(continues on next page)

(continued from previous page)

```
    69        1.0           NaN          NaN         NaN          NaN         NaN ␣
↪        NaN          NaN         NaN          NaN
    77        1.0           NaN          NaN         NaN          NaN         NaN ␣
↪        NaN          NaN         NaN          NaN
11  61        1.0           NaN          NaN         NaN          NaN         NaN ␣
↪        NaN          NaN         NaN          NaN
20  59        1.0           NaN          NaN         NaN          NaN         NaN ␣
↪        NaN          NaN         NaN          NaN
...           ...           ...          ...         ...          ...         ... ␣
↪        ...          ...         ...          ...
733 769       NaN           NaN          NaN         NaN          NaN         NaN ␣
↪        NaN          NaN         NaN          1.0
755 769       NaN           NaN          NaN         NaN          NaN         NaN ␣
↪        NaN          NaN         NaN          1.0
    787       NaN           NaN          NaN         NaN          NaN         NaN ␣
↪        NaN          NaN         NaN          1.0
771 788       NaN           NaN          NaN         NaN          NaN         NaN ␣
↪        NaN          NaN         NaN          1.0
783 788       NaN           NaN          NaN         NaN          NaN         NaN ␣
↪        NaN          NaN         NaN          1.0

[1014 rows x 10 columns]
```

As you can see, the edge table is also comprised of 1014 edges between the nodes in `v`. However, every type of connection get's its own column, where a "nan" value means that an edge does not have a relation of the corresponding type.

What Next

Now that you have an idea of what the DeepGraph package provides, you should investigate the parts of the package most useful for you. See *API Reference* for details.

1.4 API Reference

The API reference summarizes DeepGraph's core class, its methods and the functions subpackage.

1.4.1 The DeepGraph class

`DeepGraph`([v, e, supernode_labels_by, . . .])	The core class of DeepGraph (dg).

deepgraph.deepgraph.DeepGraph

class DeepGraph(*v=None*, *e=None*, *supernode_labels_by=None*, *superedge_labels_by=None*)

The core class of DeepGraph (dg).

This class encapsulates the graph representation as `pandas.DataFrame` objects in its attributes `v` and `e`. It can be initialized with a node table `v`, whose rows represent the nodes of the graph, as well as an edge table `e`, whose rows represent edges between the nodes.

Given a node table `v`, it provides methods to iteratively compute pairwise relations between the nodes using arbitrary, user-defined functions. These methods provide arguments to parallelize the computation and control memory consumption (see `create_edges` and `create_edges_ft`).

Also provides methods to partition nodes, edges or an entire graph by the graph's properties and labels, and to create common network representations and graph objects of popular Python network packages.

Furthermore, it provides methods to visualize graphs and their properties and to benchmark the graph construction parameters.

Optionally, the convenience parameter `supernode_labels_by` can be passed, creating supernode labels by enumerating all distinct (tuples of) values of a (multiple) column(s) of `v` . Superedge labels can be created analogously, by passing the parameter `superedge_labels_by`.

Parameters

- **v** (*pandas.DataFrame or pandas.HDFStore, optional (default=None)*) – The node table, a table representation of the nodes of a graph. The index of `v` must be unique and represents the node indices. The column names of `v` represent the types of features of the nodes, and each cell represents a feature of a node. Only a reference to the input DataFrame is created, not a copy. May also be a `pandas.HDFStore`, but only `create_edges` and `create_edges_ft` may then be used (so far).
- **e** (*pandas.DataFrame, optional (default=None)*) – The edge table, a table representation of the edges between the nodes given by `v`. Its index has to be a `pandas.core.index.MultiIndex`, whose first level contains the indices of the source nodes, and the second level contains the indices of the target nodes. Each row of `e` represents an edge, column names of `e` represent the types of relations of the edges, and each cell in `e` represents a relation of an edge. Only a reference to the input DataFrame is created, not a copy.
- **supernode_labels_by** (*dict, optional (default=None)*) – A dictionary whose keys are strings and their values are (lists of) column names of `v`. Appends a column to `v` for each key, whose values correspond to supernode labels, enumerating all distinct (tuples of) values of the column(s) given by the dict's value.
- **superedge_labels_by** (*dict, optional (default=None)*) – A dictionary whose keys are strings and their values are (lists of) column names of `e`. Appends a column to `e` for each key, whose values correspond to superedge labels enumerating all distinct (tuples of) values of the column(s) given by the dict's value.

v

See Parameters.

Type pandas.DataFrame

e

See Parameters.

Type pandas.DataFrame

n

Property: Number of nodes.

Type int

m

Property: Number of edges.

Type int

f

Property: types of features and number of features of corresponding type.

Type pd.DataFrame

r

Property: types of relations and number of relations of corresponding type.

Type pd.DataFrame

__init__(*v=None*, *e=None*, *supernode_labels_by=None*, *superedge_labels_by=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([v, e, supernode_labels_by, . . .])	Initialize self.
append_binning_labels_v(col, col_name[, . . .])	Append a column with binning labels of the values in `v[col]`.
append_cp([directed, connection, col_name, . . .])	Append a component membership column to `v`.
`append_datetime_categories_v`([col, . . .])	Append datetime categories to `v`.
create_edges([connectors, selectors, . . .])	Create an edge table `e` linking the nodes in `v`.
create_edges_ft(ft_feature[, connectors, . . .])	Create (ft) an edge table `e` linking the nodes in `v`.
filter_by_interval_e(col, interval[, endpoint])	Keep only edges in `e` with relations of type `col` in `interval`.
filter_by_interval_v(col, interval[, endpoint])	Keep only nodes in `v` with features of type `col` in `interval`.
filter_by_values_e(col, values)	Keep only edges in `e` with relations of type `col` in `values`.
filter_by_values_v(col, values)	Keep only nodes in `v` with features of type `col` in `values`.
partition_edges([relations, . . .])	Return a superedge DataFrame `se`.
partition_graph(features[, feature_funcs, . . .])	Return supergraph DataFrames `sv` and `se`.
partition_nodes(features[, feature_funcs, . . .])	Return a supernode DataFrame `sv`.
plot_2d(x, y[, edges, C, C_split_0, . . .])	Plot nodes and corresponding edges in 2 dimensions.
plot_2d_generator(x, y, by[, edges, C, . . .])	Plot nodes and corresponding edges by groups.
`plot_3d`(x, y, z[, edges, kwds_scatter, . . .])	Work in progress!
plot_hist(x[, bins, log_bins, density, . . .])	Plot a histogram (or pdf) of x.
plot_logfile(logfile)	Plot a logfile.
plot_map(lon, lat[, edges, C, C_split_0, . . .])	Plot nodes and corresponding edges on a basemap.
plot_map_generator(lon, lat, by[, edges, C, . . .])	Plot nodes and corresponding edges by groups, on basemaps.
`plot_raster`(label[, time, ax])	Work in progress!
`plot_rects_label_numeric`(label, xl, xr[, . . .])	Work in progress!
`plot_rects_numeric_numeric`(yb, yt, xl, xr[, . . .])	Work in progress!
return_cs_graph([relations, dropna])	Return `scipy.sparse.coo_matrix` representation(s).
return_gt_graph([features, relations, . . .])	Return a `graph_tool.Graph` representation.
return_nx_graph([features, relations, dropna])	Return a `networkx.DiGraph` representation.
return_nx_multigraph([features, relations, . . .])	Return a `networkx.MultiDiGraph` representation.
update_edges()	After removing nodes in `v`, update `e`.

Attributes

f	Types of features and number of features of corresponding type.
m	The number of edges
n	The number of nodes
r	Types of relations and number of relations of corresponding type.

Creating Edges

DeepGraph.create_edges([connectors, ...])	Create an edge table e linking the nodes in v.
DeepGraph.create_edges_ft(ft_feature[, ...])	Create (ft) an edge table e linking the nodes in v.

deepgraph.deepgraph.DeepGraph.create_edges

DeepGraph.**create_edges**(*connectors=None*, *selectors=None*, *transfer_features=None*, *r_dtype_dic=None*, *no_transfer_rs=None*, *step_size=10000000*, *from_pos=0*, *to_pos=None*, *hdf_key=None*, *verbose=False*, *logfile=None*)

Create an edge table e linking the nodes in v.

This method enables an iterative computation of pairwise relations (edges) between the nodes represented by v. It does so in a flexible, efficient and vectorized fashion, easily parallelizable and with full control over RAM usage.

1. Connectors

The simplest use-case is to define a single connector function acting on a single column of the node table v. For instance, given a node table v

```
>>> import pandas as pd
>>> import deepgraph as dg
>>> v = pd.DataFrame({'time': [0.,2.,9.], 'x': [3.,1.,12.]})
>>> g = dg.DeepGraph(v)
```

```
>>> g.v
   time   x
0     0   3
1     2   1
2     9  12
```

one may define a function

```
>>> def time_difference(time_s, time_t):
...     dt = time_t - time_s
...     return dt
```

and pass it to create_edges, in order to compute the time difference of each pair of nodes

```
>>> g.create_edges(connectors=time_difference)
```

```
>>> g.e
     dt
s t
```

(continues on next page)

(continued from previous page)

```
0 1   2
  2   9
1 2   7
```

As one can see, the connector function takes column names of `v` with additional '_s' and '_t' endings (indicating source node values and target node values, respectively) as input, and returns a variable with the computed values. The resulting edge table `g.e` is indexed by the node indices ('s' and 't', representing source and target node indices, respectively), and has one column ('dt', the name of the returned variable) with the computed values of the given connector. Note that only the upper triangle adjacency matrix is computed, which is always the case. See Notes for further information.

One may also pass a list of functions to `connectors`, which are then computed in the list's order. Generally, a connector function can take multiple column names of `v` (with '_s' and/or '_t' appended) as input, as well as already computed relations of former connectors. Also, any connector function may have multiple output variables. Every output variable has to be a 1-dimensional `np.ndarray` (with arbitrary dtype, including `object`). The return statement may not contain any operators, only references to each computed relation.

For instance, considering the above example, one may define an additional connector

```
>>> def velocity(dt, x_s, x_t):
...     dx = x_t - x_s
...     v = dx / dt
...     return v, dx
```

and then apply both connectors on `v`, resulting in

```
>>> g.create_edges(connectors=[time_difference, velocity])
```

```
>>> g.e
     dt  dx         v
s t
0 1   2  -2 -1.000000
  2   9   9  1.000000
1 2   7  11  1.571429
```

2. Selectors

However, one is often only interested in a subset of all possible edges. In order to select edges during the iteration process - based on some conditions on the node's features and their computed relations - one may pass a (list of) selector function(s) to `create_edges`. For instance, given the above example, one may define a selector

```
>>> def dt_thresh(dt, sources, targets):
...     sources = sources[dt > 5]
...     targets = targets[dt > 5]
...     return sources, targets
```

and apply it in conjunction with the `time_difference` connector

```
>>> g.create_edges(connectors=time_difference, selectors=dt_thresh)
```

```
>>> g.e
     dt
s t
```

(continues on next page)

(continued from previous page)

```
0 2   9
1 2   7
```

leaving only edges with a time difference larger than 5.

Every selector function must have `sources` and `targets` as input arguments as well as in the return statement. Most generally, they may depend on column names of `v` (with '_s' and/or '_t' appended) and/or computed relations of connector functions, and/or computed relations of former selector functions. Apart from `sources` and `targets`, they may additionally return computed relations. Given this input/output flexibility of selectors, one could in fact compute all required relations, and select any desired subset of edges, with a single selector function. The purpose of splitting connectors and/or selectors, however, is to control the iteration's performance by consecutively computing relations and selecting edges: **hierarchical selection**.

3. Hierarchical Selection

As the algorithm iterates through the chunks of all possible source and target node indices ([0, g.n*(g.n-1)/2]), it goes through the list of `selectors` at each step. If a selector has a relation as input, it must have either been computed by a former selector, or the selector requests its computation by the corresponding connector function in `connectors` (this connector may not depend on any other not yet computed relations). Once the input relations are computed (if requested), the selector is applied and returns updated indices, which are then passed to the next selector. Hence, with each selector, the indices are reduced and consecutive computation of relations only consider the remaining indices. After all selectors have been applied, the connector functions that have not been requested by any selector are computed (on the final, reduced chunk of node and target indices).

4. Transferring Features

The argument `transfer_features`, which takes a (list of) column name(s) of `v`, makes it possible to transfer features of `v` to the created edge table `e`

```
>>> g.create_edges(connectors=time_difference,
...                transfer_features=['x', 'time'])
```

```
>>> g.e
     dt  time_s  time_t  x_s  x_t
s t
0 1   2       0       2    3    1
  2   9       0       9    3   12
1 2   7       2       9    1   12
```

If computation time and memory consumption are of no concern, one might skip the remaing paragraphs.

5. Logging

Clearly, the order of the hierarchical selection as described in 3. influences the computation's efficiency. The complexity of a relation's computation and the (expected average) number of deleted edges of a selector should be considered primarily. In order to track and benchmark the iteration process, the progress and time measurements are printed for each iteration step, if `verbose` is set to True. Furthermore, one may create a logfile (which can also be plot by `dg.DeepGraph.plot_logfile`) by setting the argument `logfile` to a string, indicating the file name of the created logfile.

6. Parallelization and Memory Control

The arguments `from_pos`, `to_pos` and `step_size` control the range of processed pairs of nodes and the number of pairs of nodes to process at each iteration step. They may be used for parallel computation and to control RAM usage. See Parameters for details.

It is also possible to initiate `dg.DeepGraph` with a `pandas.HDFStore` containing the DataFrame representing the node table. Only the data requested by `transfer_features` and the user- defined connectors and selectors at each iteration step is then pulled from the store, which is particularly useful for large node tables and parallel computation. The only requirement is that the node table contained in the store is in table(t) format, not fixed(f) format. For instance, considering the above created node table, one may store it in a hdf file

```
>>> vstore = pd.HDFStore('vstore.h5')
>>> vstore.put('node_table', v, format='t', index=False)
```

initiate a DeepGraph instance with the store

```
>>> g = dg.DeepGraph(vstore)
```

```
>>> g.v
<class 'pandas.io.pytables.HDFStore'>
File path: vstore.h5
/node_table            frame_table  (typ->appendable,nrows->3,ncols->2,
indexers->[index])
```

and then create edges the same way as if g.v were a DataFrame

```
>>> g.create_edges(connectors=time_difference)
```

```
>>> g.e
     dt
s t
0 1   2
  2   9
1 2   7
```

In case the store has multiple nodes, hdf_key has to be set to the node corresponding to the node table of the graph.

Also, one may pass a (list of) name(s) of computed relations, no_transfer_rs, which should not be transferred to the created edge table e. This can be advantageous, for instance, if a selector depends on computed relations that are of no further interest.

Furthermore, it is possible to force the dtype of computed relations with the argument r_dtype_dic. The dtype of a relation is then set at each iteration step, but **after** all selectors and connectors were processed.

7. Creating Edges on a Fast Track

If the selection of edges includes a simple distance threshold, i.e. a selector function defined as follows:

```
>>> def ft_selector(x_s, x_t, threshold, sources, targets):
...     dx = x_t - x_s
...     sources = sources[dx <= threshold]
...     targets = targets[dx <= threshold]
...     return sources, targets, dx
```

the method create_edges_ft should be considered, since it provides a much faster iteration algorithm.

Parameters

- **connectors** (*function or array_like, optional (default=None)*) – User defined connector function(s) that compute pairwise relations between the nodes in v. A connector accepts multiple column names of v (with '_s' and/or '_t' appended, indicating source node values and target node values, respectively) as input, as well as already computed relations of former connectors. A connector function may have multiple output variables. Every output variable has to be a 1-dimensional np.ndarray (with arbitrary dtype, including `object`). See above and `dg.functions` for examplary connector functions.

- **selectors** (*function or array_like, optional (default=None)*) – User defined selector function(s) that select edges during the iteration process, based on some conditions on the node's features and their computed relations. Every selector function must have `sources` and `targets` as input arguments as well as in the return statement. A selector may depend on column names of `v` (with '_s' and/or '_t' appended) and/or computed relations of connector functions, and/or computed relations of former selector functions. Apart from `sources` and `targets`, they may also return computed relations (see connectors). See above, and `dg.functions` for exemplary selector functions.
- **transfer_features** (*str, int or array_like, optional (default=None)*) – A (list of) column name(s) of `v`, indicating which features of `v` to transfer to `e` (appending '_s' and '_t' to the column names of `e`, indicating source and target node features, respectively).
- **r_dtype_dic** (*dict, optional (default=None)*) – A dictionary with names of computed relations of connectors and/or selectors as keys and dtypes as values. Forces the data types of the computed relations in `e` during the iteration (but **after** all selectors and connectors were processed), otherwise infers them.
- **no_transfer_rs** (*str or array_like, optional (default=None)*) – Name(s) of computed relations that are not to be transferred to the created edge table `e`. Can be used to save memory, e.g., if a selector depends on computed relations that are of no interest otherwise.
- **step_size** (*int, optional (default=1e6)*) – The number of pairs of nodes to process at each iteration step. Must be in [1, g.n*(g.n-1)/2]. Its value determines computation speed and memory consumption.
- **from_pos** (*int, optional (default=0)*) – Determines from which pair of nodes to start the iteration process. Must be in [0, g.n*(g.n-1)/2 [. May be used in conjuction with `to_pos` for parallel computation.
- **to_pos** (*positive integer, optional (default=None)*) – Determines at which pair of nodes to stop the iteration process (the endpoint is excluded). Must be in [1, g.n*(g.n-1)/2] and larger than `from_pos`. Defaults to None, which translates to the last pair of nodes, g.n*(g.n-1)/2. May be used in conjunction with `from_pos` for parallel computation.
- **hdf_key** (*str, optional (default=None)*) – If you initialized `dg.DeepGraph` with a `pandas.HDFStore` and the store has multiple nodes, you must pass the key to the node in the store that corresponds to the node table.
- **verbose** (*bool, optional (default=False)*) – Whether to print information at each step of the iteration process.
- **logfile** (*str, optional (default=None)*) – Create a log-file named by `logfile`. Contains the time and date of the method's call, the input arguments and time mesaurements for each iteration step. A plot of `logfile` can be created by `dg.DeepGraph.plot_logfile`.

Returns **e** – Set the created edge table `e` as attribute of `dg.DeepGraph`.

Return type pd.DataFrame

See also:

`create_edges_ft()`

Notes

1. Input and output data types

Since connectors (and selectors) take columns of a pandas DataFrame as input, there are no restrictions on the data types of which pairwise relations are computed. In the most general case, a DataFrame's column has `object` as dtype, and its values may then be arbitrary Python objects. The same goes for the output variables of connectors (and selectors). The only requirement is that each ouput variable is 1-dimensional.

However, it is also possible to use the values of a column of `v` as references to arbitrary objects, which may sometimes be more convenient. In case a connector (or selector) needs the node's original indices as input, one may simply copy them to a column, e.g.

```
>>> v['indices'] = v.index
```

and then define the connector's (or selector's) input arguments accordingly.

2. Connectors and selectors

The only requirement on connectors and selectors is that their input arguments and return statements are consistent with the column names of `v` and the passing of computed relations (see above, 3. Hierarchical Selection).

Whatever happens inside the functions is entirely up to the user. This means, for instance, that one may wrap arbitrary functions within a connector (selector), such as optimized C functions or existing functions whose input/output is not consistent with the `create_edges` method (see, e.g., the methods provided in `dg.functions`, `scipy` or scikit learn's `sklearn.metrics` and `sklearn.neighbors.DistanceMetric`). One could also store a connector's (selector's) computations directly within the function, or let the function print out any desired information during iteration.

3. Why not compute the full adjacency matrix?

This is due to efficiency. For any asymmetric function (i.e., f(s, t) != f(t, s)), one can always create an additional connector (or output variable) that computes the mirrored values of that function.

deepgraph.deepgraph.DeepGraph.create_edges_ft

`DeepGraph.`**`create_edges_ft`**(*ft_feature*, *connectors=None*, *selectors=None*, *transfer_features=None*, *r_dtype_dic=None*, *no_transfer_rs=None*, *min_chunk_size=1000*, *max_pairs=10000000*, *from_pos=0*, *to_pos=None*, *hdf_key=None*, *verbose=False*, *logfile=None*)

Create (ft) an edge table `e` linking the nodes in `v`.

This method implements the same functionalities as `create_edges`, with the difference of providing a much quicker iteration algorithm based on a so-called fast-track feature. It is advised to read the docstring of `create_edges` before this one, since only the differences are explained in the following.

Apart from the hierarchical selection through `connectors` and `selectors` as described in the method `create_edges` (see 1.-3.), this method necessarily includes the (internal) selector function

```
>>> def ft_selector(ftf_s, ftf_t, ftt, sources, targets):
...     ft_r = ftf_t - ftf_s
...     sources = sources[ft_r <= ftt]
...     targets = targets[ft_r <= ftt]
...     return sources, targets, ft_r
```

where `ftf` is the fast-track feature (a column name of `v`), `ftt` the fast-track threshold (a positive number), and ft_r the computed fast-track relation. The argument `ft_feature`, which has to be a tuple (`ftf`, `ftt`), determines these variables.

1. The Fast-Track Feature

The simplest use-case, therefore, is to only pass `ft_feature`. For instance, given a node table

```
>>> import pandas as pd
>>> import deepgraph as dg
>>> v = pd.DataFrame({'time': [-3.6,-1.1,1.4,4., 6.3],
...                   'x': [-3.,3.,1.,12.,7.]})
>>> g = dg.DeepGraph(v)
```

```
>>> g.v
   time   x
0  -3.6  -3
1  -1.1   3
2   1.4   1
3   4.0  12
4   6.3   7
```

one may create and select edges by

```
>>> g.create_edges_ft(ft_feature=('time', 5))
```

```
>>> g.e
     ft_r
s t
0 1   2.5
  2   5.0
1 2   2.5
2 3   2.6
  4   4.9
3 4   2.3
```

leaving only edges with a time difference smaller than (or equal to) `ftt` = 5. Note that the node table always has to be sorted by the fast-track feature. This is due to the fact that the algorithm only processes pairs of nodes whose fast-track relation is smaller than (or equal to) the fast-track threshold, and the (pre)determination of these pairs relies on a sorted DataFrame.

2. Hierarchical Selection

Additionally, one may define `connectors` and `selectors` as described in `create_edges` (see 1.-3.). Per default, the (internal) fast-track selector is applied first. It's order of application, however, may be determined by inserting the string 'ft_selector' in the desired position of the list of `selectors`.

The remaining arguments are as described in `create_edges`, apart from `min_chunk_size`, `max_pairs`, `from_pos` and `to_pos`. If computation time and/or memory consumption are a concern, one may therefore read the remaining paragraph.

3. Parallelization and Memory Control on a FastTrack

At each iteration step, the algorithm takes a number of nodes (n = `min_chunk_size`, per default n=1000) and computes the fast track relation (distance) between the last node and the first node, d_ftf = ftf_last - ftf_first. In case d_ftf > `ftt`, all nodes with a fast- track feature < ftf_last - `ftt` are considered source nodes, and their relations with all n nodes are computed (hierarchical selection). In case d_ftf <= `ftt`, n is increased, s.t. d_ftf > `ftt`. This might lead to a large number of pairs of nodes to process at a given iteration step. In order to control memory consumption, one might therefore set `max_pairs` to a suitable value, triggering a subiteration if this value is exceeded.

In order to parallelize the iterative computation, one may pass the arguments `from_pos` and `to_pos`. They determine the range of **source nodes** to process (endpoint excluded). Hence, `from_pos` has to be in [0, g.n[, and `to_pos` in [1,g.n]. For instance, given the node table above

```
>>> g.v
   time   x
0  -3.6  -3
1  -1.1   3
2   1.4   1
3   4.0  12
4   6.3   7
```

we can compute all relations of the source nodes in [1,3[by

```
>>> g.create_edges_ft(ft_feature=('time', 5), from_pos=1, to_pos=3)
```

```
>>> g.e
     ft_r
s t
1 2   2.5
2 3   2.6
  4   4.9
```

Like `create_edges`, this method also works with a `pd.HDFStore` containing the DataFrame representing the node table. Only the data requested by `ft_feature`, `transfer_features` and the user-defined `connectors` and `selectors` at each iteration step is then pulled from the store. The node table in the store has to be in table(t) format, and additionally, the fast_track feature has to be a data column. For instance, storing the above node table

```
>>> vstore = pd.HDFStore('vstore.h5')
>>> vstore.put('node_table', v, format='t', data_columns=True,
...            index=False)
```

one may initiate a DeepGraph instance with the store

```
>>> g = dg.DeepGraph(vstore)
```

```
>>> g.v
<class 'pandas.io.pytables.HDFStore'>
File path: vstore.h5
/node_table            frame_table  (typ->appendable,nrows->5,ncols->2,
indexers->[index],dc->[time,x])
```

and then create edges the same way as if `g.v` were a DataFrame

```
>>> g.create_edges_ft(ft_feature=('time', 5), from_pos=1, to_pos=3)
```

```
>>> g.e
     ft_r
s t
1 2   2.5
2 3   2.6
  4   4.9
```

Warning: There is no assertion whether the node table in a store is sorted by the fast-track feature! The result of an unsorted table is unpredictable, and generally not correct.

Parameters

- **ft_feature** (*tuple*) – A tuple (ftf, ftt), where ftf is a column name of `v` (the fast-track feature) and ftt a positive number (the fast-track threshold). The fast-track feature may contain integers or floats, but datetime-like values are also accepted. In that case, `ft_feature` has to be a tuple of length 3, (ftf, ftt, dt_unit), where dt_unit is on of {'D','h','m','s','ms','us','ns'}:

  - *D*: days
  - *h*: hours
  - *m*: minutes
  - *s*: seconds
  - *ms*: milliseconds
  - *us*: microseconds
  - *ns*: nanoseconds

  determining the unit in which the temporal distance is measured. The variable name of the fast-track relation transferred to `e` is `ft_r`.

- **connectors** (*function or array_like, optional (default=None)*) – User defined connector function(s) that compute pairwise relations between the nodes in `v`. A connector accepts multiple column names of `v` (with '_s' and/or '_t' appended, indicating source node values and target node values, respectively) as input, as well as already computed relations of former connectors. A connector function may have multiple output variables. Every output variable has to be a 1-dimensional `np.ndarray` (with arbitrary dtype, including `object`). A connector may also depend on the fast- track relations ('ft_r'). See `dg.functions` for examplary connector functions.

- **selectors** (*function or array_like, optional (default=None)*) – User defined selector function(s) that select edges during the iteration process, based on some conditions on the node's features and their computed relations. Every selector function must have `sources` and `targets` as input arguments as well as in the return statement. A selector may depend on column names of `v` (with '_s' and/or '_t' appended) and/or computed relations of connector functions, and/or computed relations of former selector functions. Apart from `sources` and `targets`, they may also return computed relations (see connectors). A selector may also depend on the fast-track relations ('ft_r'). See `dg.functions` for exemplary selector functions.

  Note: To specify the hierarchical order of the selection by the fast-track selector, insert the string 'ft_selector' in the corresponding position of the `selectors` list. Otherwise, computation of ft_r and selection by the fast-track selector is carried out first.

- **transfer_features** (*str, int or array_like, optional (default=None)*) – A (list of) column name(s) of `v`, indicating which features of `v` to transfer to `e` (appending '_s' and '_t' to the column names of `e`, indicating source and target node features, respectively).

- **r_dtype_dic** (*dict, optional (default=None)*) – A dictionary with names of computed relations of connectors and/or selectors as keys and dtypes as values. Forces the data types of the computed relations in `e` during the iteration (but **after** all selectors and connectors were processed), otherwise infers them.

- **no_transfer_rs** (*str or array_like, optional (default=None)*) – Name(s) of computed relations that are not to be transferred to the created edge table `e`.

Can be used to save memory, e.g., if a selector depends on computed relations that are of no interest otherwise.

- **min_chunk_size** (*int, optional (default=1000)*) – The minimum number of nodes to form pairs of at each iteration step. See above for details.
- **max_pairs** (*positive integer, optional (default=1e6)*) – The maximum number of pairs of nodes to process at any given iteration step. If the number is exceeded, a memory saving subiteration is applied.
- **from_pos** (*int, optional (default=0)*) – The locational index (.iloc) of `v` to start the iteration. Determines the range of **source nodes** to process, in conjuction with `to_pos`. Has to be in [0, g.n[, and smaller than `to_pos`. See above for details and an example.
- **to_pos** (*int, optional (default=None)*) – The locational index (.iloc) of `v` to end the iteration (excluded). Determines the range of **source nodes** to process, in conjuction with `from_pos`. Has to be in [1, g.n], and larger than `from_pos`. Defaults to None, which translates to the last node of `v`, to_pos=g.n. See above for details and an example.
- **hdf_key** (*str, optional (default=None)*) – If you initialized `dg.DeepGraph` with a `pandas.HDFStore` and the store has multiple nodes, you must pass the key to the node in the store that corresponds to the node table.
- **verbose** (*bool, optional (default=False)*) – Whether to print information at each step of the iteration process.
- **logfile** (*str, optional (default=None)*) – Create a log-file named by `logfile`. Contains the time and date of the method's call, the input arguments and time mesaurements for each iteration step. A plot of `logfile` can be created by `dg.DeepGraph.plot_logfile`.

Returns **e** – Set the created edge table `e` as attribute of `dg.DeepGraph`.

Return type pd.DataFrame

See also:

create_edges()

Notes

The parameter `min_chunk_size` enforces a vectorized iteration and changing its value can both accelerate or slow down computation time. This depends mostly on the distribution of values of the fast track feature, and the complexity of the given `connectors` and `selectors`. Use the logging capabilites to determine a good value.

When using a `pd.HDFStore` for the computation, the following advice might be considered. Recall that the only requirements on the node in the store are: the format is table(t), not fixed(t); the node is sorted by the fast-track feature; and the fast-track feature is a data column.

The recommended procedure of storing a given node table `v` in a store is the following (using the above node table):

```
>>> vstore = pd.HDFStore('vstore.h5')
>>> vstore.put('node_table', v, format='t', data_columns=True,
...             index=False)
```

Setting index=False significantly decreases the time to construct the node in the store, and also reduces the resulting file size. It has no impact, however, on the capability of querying the store (with the pd.HDFStore.select* methods).

However, there are two reasons one might want to create a pytables index of the fast-track feature:

1. The node table might be too large to be sorted in memory. To sort it on disc, one may proceed as follows. Assuming an unsorted (large) node table

```
>>> v = pd.DataFrame({'time': [6.3,-3.6,4.,-1.1,1.4],
...                    'x': [-3.,3.,1.,12.,7.]})
```

```
>>> v
   time   x
0   6.3  -3
1  -3.6   3
2   4.0   1
3  -1.1  12
4   1.4   7
```

one stores it as recommended

```
>>> vstore = pd.HDFStore('vstore.h5')
>>> vstore.put('node_table', v, format='t', data_columns=True,
...            index=False)
>>> vstore.get_storer('node_table').group.table
/node_table/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "time": Float64Col(shape=(), dflt=0.0, pos=1),
  "x": Float64Col(shape=(), dflt=0.0, pos=2)}
  byteorder := 'little'
  chunkshape := (2730,)
```

creates a (full) pytables index of the fast-track feature

```
>>> vstore.create_table_index('node_table', columns=['time'],
...                           kind='full')
>>> vstore.get_storer('node_table').group.table
/node_table/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "time": Float64Col(shape=(), dflt=0.0, pos=1),
  "x": Float64Col(shape=(), dflt=0.0, pos=2)}
  byteorder := 'little'
  chunkshape := (2730,)
  autoindex := True
  colindexes := {
    "time": Index(6, full, shuffle, zlib(1)).is_csi=True}
```

and then sorts it on disc with

```
>>> vstore.close()
>>> !ptrepack --chunkshape=auto --sortby=time vstore.h5 s_vstore.h5
>>> s_vstore = pd.HDFStore('s_vstore.h5')
```

```
>>> s_vstore.node_table
   time   x
```

(continues on next page)

(continued from previous page)

```
1  -3.6   3
3  -1.1  12
4   1.4   7
2   4.0   1
0   6.3  -3
```

2. To speed up the internal queries on the fast-track feature

```
>>> s_vstore.create_table_index('node_table', columns=['time'],
...                             kind='full')
```

See http://stackoverflow.com/questions/17893370/ptrepack-sortby-needs-full-index and https://gist.github.com/michaelaye/810bd0720bb1732067ff for details, benchmarks, and the effects of compressing the store.

Graph Partitioning

DeepGraph.partition_nodes(features[, …])	Return a supernode DataFrame `sv`.
DeepGraph.partition_edges([relations, …])	Return a superedge DataFrame `se`.
DeepGraph.partition_graph(features[, …])	Return supergraph DataFrames `sv` and `se`.

deepgraph.deepgraph.DeepGraph.partition_nodes

DeepGraph.**partition_nodes**(*features*, *feature_funcs=None*, *n_nodes=True*, *return_gv=False*)

Return a supernode DataFrame `sv`.

This is essentially a wrapper around the pandas groupby method: `sv` = `v`.groupby(`features`).agg(`feature_funcs`). It creates a (intersection) partition of the nodes in `v` by the type(s) of feature(s) `features`, resulting in a supernode DataFrame `sv`. By passing a dictionary of functions on the features of `v`, `feature_funcs`, one may aggregate user-defined values of the partition's elements, the supernodes' features. If `n_nodes` is True, create a column with the number of each supernode's constituent nodes. If `return_gv` is True, return the created groupby object to facilitate additional operations, such as `gv`.apply(func, *args, **kwargs).

For details, type help(`v`.groupby), and/or inspect the available methods of `gv`.

For examples, see below. For an in-depth description and mathematical details of graph partitioning, see https://arxiv.org/pdf/1604.00971v1.pdf, in particular Sec. III A, E and F.

Parameters

- **features** (*str, int or array_like*) – Column name(s) of `v`, indicating the type(s) of feature(s) used to induce a (intersection) partition. Creates a pandas groupby object, `gv` = `v`.groupby(`features`).
- **feature_funcs** (*dict, optional (default=None)*) – Each key must be a column name of `v`, each value either a function, or a list of functions, working when passed a `pandas.DataFrame` or when passed to `pandas.DataFrame.apply`. See the docstring of `gv`.agg for details: help(`gv`.agg).
- **n_nodes** (*bool, optional (default=True)*) – Whether to create a `n_nodes` column in `sv`, indicating the number of nodes in each supernode.
- **return_gv** (*bool, optional (default=False)*) – If True, also return the `v`.groupby(`features`) object, `gv`.

Returns

- **sv** (*pd.DataFrame*) – The aggreated DataFrame of supernodes, `sv`.
- **gv** (*pandas.core.groupby.DataFrameGroupBy*) – The pandas groupby object, `v`.groupby(`features`).

See also:

`partition_edges()`, `partition_graph()`

Notes

Currently, NA groups in GroupBy are automatically excluded (silently). One workaround is to use a placeholder (e.g., -1, 'none') for NA values before doing the groupby (calling this method). See http://stackoverflow.com/questions/18429491/groupby-columns-with-nan-missing-values and https://github.com/pydata/pandas/issues/3729.

Examples

First, we need a node table, in order to demonstrate its partitioning:

```
>>> import pandas as pd
>>> import deepgraph as dg
>>> v = pd.DataFrame({'x': [-3.4,2.1,-1.1,0.9,2.3],
...                   'time': [0,0,2,2,9],
...                   'color': ['g','g','b','g','r'],
...                   'size': [1,3,2,3,1]})
>>> g = dg.DeepGraph(v)
>>> g.v
  color  size  time    x
0     g     1     0 -3.4
1     g     3     0  2.1
2     b     2     2 -1.1
3     g     3     2  0.9
4     r     1     9  2.3
```

Create a partition by the type of feature 'color':

```
>>> g.partition_nodes('color')
       n_nodes
color
b            1
g            3
r            1
```

Create an intersection partition by the types of features 'color' and 'size' (which is a further refinement of the last partition):

```
>>> g.partition_nodes(['color', 'size'])
            n_nodes
color size
b     2           1
g     1           1
      3           2
r     1           1
```

Partition by 'color' and collect x values:

```
>>> g.partition_nodes('color', {'time': lambda x: list(x)})
       n_nodes       time
color
b            1        [2]
g            3  [0, 0, 2]
r            1        [9]
```

Partition by 'color' and aggregate with different functions:

```
>>> g.partition_nodes('color', {'time': [lambda x: list(x), np.max],
...                            'x': [np.mean, np.sum, np.std]})
       n_nodes    x_mean  x_sum     x_std time_<lambda>  time_amax
color
b            1 -1.100000   -1.1       NaN           [2]          2
g            3 -0.133333   -0.4  2.891943     [0, 0, 2]          2
r            1  2.300000    2.3       NaN           [9]          9
```

deepgraph.deepgraph.DeepGraph.partition_edges

DeepGraph.**partition_edges**(*relations=None*, *source_features=None*, *target_features=None*, *relation_funcs=None*, *n_edges=True*, *return_ge=False*)

Return a superedge DataFrame se.

This method allows you to partition the edges in e by their types of relations, but also by the types of features of their incident source and target nodes, and any combination of the three.

Essentially, this method is a wrapper around the pandas groupby method: se = e.groupby(relations + features_s + features_t).agg(relation_funcs), where relations are column names of e, and in order to group e by features_s and/or features_t, the features of type source_features and/or target_features (column names of v) are transferred to e, appending '_s' and/or '_t' to the corresponding column names of e (if they are not already present). The only requirement on the combination of relations, source_features and target_features is that at least on of the lists has to be of length >= 1.

By passing a dictionary of functions on the relations of e, relation_funcs, one may aggregate user-defined values of the partition's elements, the superedges' relations. If n_edges is True, create a column with the number of each superedge's constituent edges. If return_ge is True, return the created groupby object to facilitate additional operations, such as ge.apply(func, *args, **kwargs).

For details, type help(g.e.groupby), and/or inspect the available methods of ge.

For examples, see below. For an in-depth description and mathematical details of graph partitioning, see https://arxiv.org/pdf/1604.00971v1.pdf, in particular Sec. III B, E and F.

Parameters

- **relations** (*str, int or array_like, optional (default=None)*) – Column name(s) of e, indicating the type(s) of relation(s) used to induce a (intersection) partition of e (in conjunction with source_features and target_features).
- **source_features** (*str, int or array_like, optional (default=None)*) – Column name(s) of v, indicating the type(s) of feature(s) of the edges' incident source nodes used to induce a (intersection) partition of e (in conjunction with relations and target_features).
- **target_features** (*str, int or array_like, optional (default=None)*) – Column name(s) of v, indicating the type(s) of feature(s) of

the edges' incident target nodes used to induce a (intersection) partition of `e` (in conjunction with `relations` and `source_features`).

- **`relation_funcs`** (*`dict, optional (default=None)`*) – Each key must be a column name of `e`, each value a (list of) function(s), working when passed a `pandas.DataFrame` or when passed to `pandas.DataFrame.apply`. See the docstring of `ge`.agg for details: help(`ge`.agg).
- **`n_edges`** (*`bool, optional (default=True)`*) – Whether to create a `n_edges` column in `se`, indicating the number of edges in each superedge.
- **`return_ge`** (*`bool, optional (default=False)`*) – If True, also return the pandas groupby object, `ge`.

Returns

- **se** (*pd.DataFrame*) – The aggreated DataFrame of superedges, `se`.
- **ge** (*pandas.core.groupby.DataFrameGroupBy*) – The pandas groupby object, `ge`.

See also:

`partition_nodes()`, *`partition_graph()`*

Notes

Currently, NA groups in GroupBy are automatically excluded (silently). One workaround is to use a placeholder (e.g., -1, 'none') for NA values before doing the groupby (calling this method). See http://stackoverflow.com/questions/18429491/groupby-columns-with-nan-missing-values and https://github.com/pydata/pandas/issues/3729.

Examples

First, we need to create a graph in order to demonstrate how to partition its edge set.

Create a node table:

```
>>> import pandas as pd
>>> import deepgraph as dg
>>> v = pd.DataFrame({'x': [-3.4,2.1,-1.1,0.9,2.3],
...                   'time': [0,1,2,5,9],
...                   'color': ['g','g','b','g','r'],
...                   'size': [1,3,2,3,1]})
>>> g = dg.DeepGraph(v)
```

```
>>> g.v
  color  size  time    x
0     g     1     0 -3.4
1     g     3     1  2.1
2     b     2     2 -1.1
3     g     3     5  0.9
4     r     1     9  2.3
```

Create an edge table:

```
>>> def some_relations(ft_r, x_s,x_t,color_s,color_t,size_s,size_t):
...     dx = x_t - x_s
...     v = dx / ft_r
```

(continues on next page)

(continued from previous page)

```
...     same_color = color_s == color_t
...     larger_than = size_s > size_t
...     return dx, v, same_color, larger_than
>>> g.create_edges_ft(('time', 5), connectors=some_relations)
>>> g.e.rename(columns={'ft_r': 'dt'}, inplace=True)
>>> g.e['inds'] = g.e.index.values  # to ease the eyes
```

```
>>> g.e
      dx  dt  larger_than  same_color         v    inds
s t
0 1  5.5   1        False        True  5.500000  (0, 1)
  2  2.3   2        False       False  1.150000  (0, 2)
  3  4.3   5        False        True  0.860000  (0, 3)
1 2 -3.2   1         True       False -3.200000  (1, 2)
  3 -1.2   4        False        True -0.300000  (1, 3)
2 3  2.0   3        False       False  0.666667  (2, 3)
3 4  1.4   4         True       False  0.350000  (3, 4)
```

Partitioning by the type of relation 'larger_than':

```
>>> g.partition_edges(relations='larger_than',
...                   relation_funcs={'dx': ['mean', 'std'],
...                                   'same_color': 'sum'})
             n_edges  same_color_sum  dx_mean    dx_std
larger_than
False              5               3     2.58  2.558711
True               2               0    -0.90  3.252691
```

A refinement of the last partition by the type of relation 'same_color':

```
>>> g.partition_edges(relations=['larger_than', 'same_color'],
...                   relation_funcs={'dx': ['mean', 'std'],
...                                   'dt': lambda x: tuple(x)})
                        n_edges dt_<lambda>   dx_mean    dx_std
larger_than same_color
False       False             2      (2, 3)  2.150000  0.212132
            True              3   (1, 5, 4)  2.866667  3.572581
True        False             2      (1, 4) -0.900000  3.252691
```

Partitioning by the type of source feature 'color':

```
>>> g.partition_edges(source_features='color',
...                   relation_funcs={'same_color': 'sum'})
         n_edges  same_color
color_s
b              1           0
g              6           3
```

As one can see, the type of feature 'color' of the source nodes has been transferred to `e`:

```
>>> g.e
      dx  dt  larger_than  same_color         v    inds color_s
s t
0 1  5.5   1        False        True  5.500000  (0, 1)       g
  2  2.3   2        False       False  1.150000  (0, 2)       g
  3  4.3   5        False        True  0.860000  (0, 3)       g
```

(continues on next page)

(continued from previous page)

```
1 2 -3.2    1         True      False -3.200000  (1, 2)       g
  3 -1.2    4        False       True -0.300000  (1, 3)       g
2 3  2.0    3        False      False  0.666667  (2, 3)       b
3 4  1.4    4         True      False  0.350000  (3, 4)       g
```

A further refinement of the last partition by the type of source feature ‘size’:

```
>>> g.partition_edges(source_features=['color', 'size'],
...                   relation_funcs={'same_color': 'sum',
...                                   'inds': lambda x: tuple(x)})
                n_edges  same_color                      inds
color_s size_s
b       2             1           0                 ((2, 3),)
g       1             3           2  ((0, 1), (0, 2), (0, 3))
        3             3           1  ((1, 2), (1, 3), (3, 4))
```

Partitioning by the types of target features (‘color’, ‘size’):

```
>>> g.partition_edges(target_features=['color', 'size'],
...                   relation_funcs={'same_color': 'sum',
...                                   'inds': lambda x: tuple(x)})
                n_edges  same_color                              inds
color_t size_t
b       2             2           0                  ((0, 2), (1, 2))
g       3             4           3  ((0, 1), (0, 3), (1, 3), (2, 3))
r       1             1           0                         ((3, 4),)
```

Partitioning by the type of source feature ‘color’ and the type of target feature ‘size’:

```
>>> g.partition_edges(source_features='color', target_features='size',
...                   relation_funcs={'same_color': 'sum',
...                                   'inds': lambda x: tuple(x)})
                n_edges  same_color                      inds
color_s size_t
b       3             1           0                 ((2, 3),)
g       1             1           0                 ((3, 4),)
        2             2           0          ((0, 2), (1, 2))
        3             3           3  ((0, 1), (0, 3), (1, 3))
```

A further refinement of the last partition by the type of relation ‘larger_than’:

```
>>> g.partition_edges(relations='larger_than',
...                   source_features='color', target_features='size',
...                   relation_funcs={'inds': lambda x: tuple(x)})
                            n_edges                      inds
larger_than color_s size_t
False       b       3             1                 ((2, 3),)
            g       2             1                 ((0, 2),)
                    3             3  ((0, 1), (0, 3), (1, 3))
True        g       1             1                 ((3, 4),)
                    2             1                 ((1, 2),)
```

deepgraph.deepgraph.DeepGraph.partition_graph

`DeepGraph.`**`partition_graph`**(*features*, *feature_funcs=None*, *relation_funcs=None*, *n_nodes=True*, *n_edges=True*, *return_gve=False*)

Return supergraph DataFrames `sv` and `se`.

This method allows partitioning of the graph represented by `v` and `e` into a supergraph, `sv` and `se`. It creates a (intersection) partition of the nodes in `v` by the type(s) of feature(s) `features`, together with the (intersection) partition's **corresponding** partition of the edges in `e`.

Essentially, this method is a wrapper around pandas groupby methods: `sv` = `v`.groupby(`features`).agg(`feature_funcs`) and `se` = `e`.groupby(features_s+features_t).agg(`relation_funcs`). In order to group `e` by features_s and features_t, the features of type `features` are transferred to `e`, appending '_s' and '_t' to the corresponding column names of `e`, indicating source and target features, respectively (if they are not already present).

By passing a dictionary of functions on the features (relations) of `v` (`e`), `feature_funcs` (`relation_funcs`), one may aggregate user-defined values of the partition's elements, the supernodes' (superedges') features (relations). If `n_nodes` (`n_edges`) is True, create a column with the number of each supernode's (superedge's) constituent nodes (edges).

If `return_gve` is True, return the created groupby objects to facilitate additional operations, such as `gv`.apply(func, *args, **kwargs) or `ge`.apply(func, *args, **kwargs).

For details, type help(`g.v`.groupby), and/or inspect the available methods of `gv`.

For examples, see below. For an in-depth description and mathematical details of graph partitioning, see https://arxiv.org/pdf/1604.00971v1.pdf, in particular Sec. III C, E and F.

Parameters

- **`features`** (*`str, int or array_like`*) – Column name(s) of `v`, indicating the type(s) of feature(s) used to induce a (intersection) partition of `v`, and its **corresponding** partition of the edges in `e`. Creates pandas groupby objects, `gv` and `ge`.
- **`feature_funcs`** (*`dict, optional (default=None)`*) – Each key must be a column name of `v`, each value either a function, or a list of functions, working when passed a `pandas.DataFrame` or when passed to `pandas.DataFrame.apply`. See the docstring of `gv`.agg for details: help(`gv`.agg).
- **`relation_funcs`** (*`dict, optional (default=None)`*) – Each key must be a column name of `e`, each value either a function, or a list of functions, working when passed a `pandas.DataFrame` or when passed to `pandas.DataFrame.apply`. See the docstring of `ge`.agg for details: help(`ge`.agg).
- **`n_nodes`** (*`bool, optional (default=True)`*) – Whether to create a `n_nodes` column in `sv`, indicating the number of nodes in each supernode.
- **`n_edges`** (*`bool, optional (default=True)`*) – Whether to create a `n_edges` column in `se`, indicating the number of edges in each superedge.
- **`return_gve`** (*`bool, optional (default=False)`*) – If True, also return the pandas groupby objects, `gv` and `ge`.

Returns

- **sv** (*pd.DataFrame*) – The aggreated DataFrame of supernodes, `sv`.
- **se** (*pd.DataFrame*) – The aggregated DataFrame of superedges, `se`.
- **gv** (*pandas.core.groupby.DataFrameGroupBy*) – The pandas groupby object, `v`.groupby(`features`).

- **ge** (*pandas.core.groupby.DataFrameGroupBy*) – The pandas groupby object, e.groupby(features_i+feaures_j).

See also:

partition_nodes(), *partition_edges()*

Notes

Currently, NA groups in GroupBy are automatically excluded (silently). One workaround is to use a placeholder (e.g., -1, 'none') for NA values before doing the groupby (calling this method). See http://stackoverflow.com/questions/18429491/groupby-columns-with-nan-missing-values and https://github.com/pydata/pandas/issues/3729.

Examples

First, we need to create a graph in order to demonstrate its partitioning into a supergraph.

Create a node table:

```
>>> import pandas as pd
>>> import deepgraph as dg
>>> v = pd.DataFrame({'x': [-3.4,2.1,-1.1,0.9,2.3],
...                   'time': [0,1,2,5,9],
...                   'color': ['g','g','b','g','r'],
...                   'size': [1,3,2,3,1]})
>>> g = dg.DeepGraph(v)
```

```
>>> g.v
  color  size  time    x
0     g     1     0 -3.4
1     g     3     1  2.1
2     b     2     2 -1.1
3     g     3     5  0.9
4     r     1     9  2.3
```

Create an edge table:

```
>>> def some_relations(ft_r, x_s,x_t,color_s,color_t,size_s,size_t):
...     dx = x_t - x_s
...     v = dx / ft_r
...     same_color = color_s == color_t
...     larger_than = size_s > size_t
...     return dx, v, same_color, larger_than
>>> g.create_edges_ft(('time', 5), connectors=some_relations)
>>> g.e.rename(columns={'ft_r': 'dt'}, inplace=True)
>>> g.e['inds'] = g.e.index.values  # to ease the eyes
```

```
>>> g.e
      dx  dt larger_than same_color         v    inds
s t
0 1  5.5   1       False       True  5.500000  (0, 1)
  2  2.3   2       False      False  1.150000  (0, 2)
  3  4.3   5       False       True  0.860000  (0, 3)
1 2 -3.2   1        True      False -3.200000  (1, 2)
  3 -1.2   4       False       True -0.300000  (1, 3)
```

(continues on next page)

(continued from previous page)

```
2 3  2.0   3        False      False  0.666667  (2, 3)
3 4  1.4   4         True      False  0.350000  (3, 4)
```

Create a supergraph by partitioning by the type of feature 'color':

```
>>> sv, se = g.partition_graph('color')
```

```
>>> sv
       n_nodes
color
b            1
g            3
r            1
```

```
>>> se
                 n_edges
color_s color_t
b       g              1
g       b              2
        g              3
        r              1
```

Create intersection partitions by the types of features 'color' and 'size' (which are further refinements of the last partitions):

```
>>> sv, se = g.partition_graph(
...     ['color', 'size'],
...     relation_funcs={'inds': lambda x: tuple(x)})
```

```
>>> sv
            n_nodes
color size
b     2           1
g     1           1
      3           2
r     1           1
```

```
>>> se
                               n_edges               inds
color_s size_s color_t size_t
b       2      g       3             1          ((2, 3),)
g       1      b       2             1          ((0, 2),)
               g       3             2  ((0, 1), (0, 3))
        3      b       2             1          ((1, 2),)
               g       3             1          ((1, 3),)
               r       1             1          ((3, 4),)
```

Partition by 'color' and aggregate some properties:

```
>>> sv, se = g.partition_graph('color',
...     feature_funcs={'time': lambda x: list(x)},
...     relation_funcs={'larger_than': 'sum', 'same_color': 'sum'})
```

```
>>> sv
       n_nodes       time
```

(continues on next page)

(continued from previous page)

```
color
b            1        [2]
g            3  [0, 1, 5]
r            1        [9]
```

```
>>> se
                 n_edges larger_than  same_color
color_s color_t
b       g              1       False           0
g       b              2        True           0
        g              3       False           3
        r              1        True           0
```

Graph Interfaces

DeepGraph.return_cs_graph([relations, dropna])	Return `scipy.sparse.coo_matrix` representation(s).
DeepGraph.return_nx_graph([features, ...])	Return a `networkx.DiGraph` representation.
DeepGraph.return_nx_multigraph([features, ...])	Return a `networkx.MultiDiGraph` representation.
DeepGraph.return_gt_graph([features, ...])	Return a `graph_tool.Graph` representation.

deepgraph.deepgraph.DeepGraph.return_cs_graph

`DeepGraph.`**`return_cs_graph`**(*relations=False*, *dropna=True*)

Return `scipy.sparse.coo_matrix` representation(s).

Create a compressed sparse graph representation for each type of relation given by `relations`. `relations` can either be False, True, or a (list of) column name(s) of `e`. If `relations` is False (default), return a single csgraph entailing all edges in `e.index`, each with a weight of 1 (in that case, `dropna` is discarded). If `relations` is True, create one csgraph for each column of `e`, where the weights are given by the columns' values. If only a subset of columns is to be mapped to csgraphs, `relations` has to be a (list of) column name(s) of `e`.

The argument `dropna` indicates whether to discard edges with NA values or not. If `dropna` is True or False, it applies to all types of relations given by `relations`. However, `dropna` can also be array_like with the same shape as `relations` (or with the same shape as `e.columns`, if `relations` is True).

Parameters

- **`relations`** (*`bool, str or array_like, optional (default=False)`*) – The types of relations to be mapped to scipy csgraphs. Can be False, True, or a (list of) column name(s) of `e`.
- **`dropna`** (*`bool or array_like, optional (default=True)`*) – Whether to drop edges with NA values. If True or False, applies to all relations given by `relations`. Otherwise, must be the same shape as `relations`. If `relations` is False, `dropna` is discarded.

Returns **csgraph** – A dictionary, where keys are column names of `e`, and values are the corresponding `scipy.sparse.coo_matrix` instance(s). If only one csgraph is created, return it directly.

Return type scipy.sparse.coo_matrix or dict

See also:

return_nx_graph(), *return_nx_multigraph()*, *return_gt_graph()*

deepgraph.deepgraph.DeepGraph.return_nx_graph

DeepGraph.**return_nx_graph**(*features=False*, *relations=False*, *dropna='none'*)

Return a `networkx.DiGraph` representation.

Create a `networkx.DiGraph` representation of the graph given by `v` and `e`. Node and edge properties to transfer can be indicated by the `features` and `relations` input arguments. Whether to drop edges with NA values in the subset of types of relations given by `relations` can be controlled by `dropna`.

Needs pandas >= 0.17.0.

Parameters

- **features** (*bool, str, or array_like, optional (default=False)*) – Indicates which types of features to transfer as node attributes. Can be column name(s) of `v`, False or True. If False, create no node attributes. If True, create node attributes for every column in `v`. If str or array_like, must be column name(s) of `v` indicating which types of features to transfer.
- **relations** (*bool, str, or array_like, optional (default=False)*) – Indicates which types of relations to transfer as edge attributes. Can be column name(s) of `e`, False or True. If False, create no edge attributes (all edges in `e.index` are transferred, regardless of `dropna`). If True, create edge attributes for every column in `e` (all edges in `e.index` are transferred, regardless of `dropna`). If str or array_like, must be column name(s) of `e` indicating which types of relations to transfer (which edges are transferred can be controlled by `dropna`).
- **dropna** (*str, optional (default='none')*) – One of {'none','any','all'}. If 'none', all edges in `e.index` are transferred. If 'any', drop all edges (rows) in `e[relations]` where any NA values are present. If 'all', drop all edges (rows) in `e[relations]` where all values are NA. Only has an effect if `relations` is str or array_like.

Returns nx_g

Return type networkx.DiGraph

See also:

return_nx_multigraph(), *return_cs_graph()*, *return_gt_graph()*

deepgraph.deepgraph.DeepGraph.return_nx_multigraph

DeepGraph.**return_nx_multigraph**(*features=False*, *relations=False*, *dropna=True*)

Return a `networkx.MultiDiGraph` representation.

Create a `networkx.MultiDiGraph` representation of the graph given by `v` and `e`. As opposed to `return_nx_graph`, where every row of `e` is treated as one edge, this method treats every cell of `e` as one edge. The input argument `features` indicates which node properties to transfer. `relations` indicates which edges to transfer. Whether to drop edges with NA values can be controlled by `dropna`.

Needs pandas >= 0.17.0.

Parameters

- **features** (*bool, str, or array_like, optional (default=False)*) – Indicates which types of features to transfer as node attributes. Can be column name(s) of `v`, False or True. If False, create no node attributes. If True, create node attributes for every column in `v`. If str or array_like, must be column name(s) of `v` indicating which types of features to transfer.
- **relations** (*bool, str, or array_like, optional (default=False)*) – Indicates which cells of `e` to transfer as edges. Can be False, True, or a (list of) column name(s) of `e`. If False (default), all cells of `e` are translated to edges, but their values are not transferred as edge attributes. If True, all cells of `e` are translated, and their values are transferred as edge attributes. If str or array_like, must be column name(s) of `e`, restricting the translation of cells to edges to `e[relations]` (values are transferred as edge attributes).
- **dropna** (*bool, optional (default=True)*) – Whether to drop edges with NA values. Cells in `e` with NA values are not translated to edges.

Returns **nx_g**

Return type networkx.MultiDiGraph

See also:

return_nx_graph(), *return_cs_graph()*, *return_gt_graph()*

deepgraph.deepgraph.DeepGraph.return_gt_graph

`DeepGraph.`**`return_gt_graph`**(*features=False*, *relations=False*, *dropna='none'*, *node_indices=False*, *edge_indices=False*)

Return a `graph_tool.Graph` representation.

Create a `graph_tool.Graph` (directed) representation of the graph given by `v` and `e`. Node and edge properties to transfer can be indicated by the `features` and `relations` input arguments. Whether to drop edges with NA values in the subset of types of relations given by `relations` can be controlled by `dropna`. If the nodes in `v` are not indexed by consecutive integers starting from 0, one may internalize the original node and edge indices as propertymaps by setting `node_indices` and/or `edge_indices` to True.

Parameters

- **features** (*bool, str, or array_like, optional (default=False)*) – Indicates which types of features to internalize as `graph_tool.PropertyMap`. Can be column name(s) of `v`, False or True. If False, create no propertymaps. If True, create propertymaps for every column in `v`. If str or array_like, must be column name(s) of `v` indicating which types of features to internalize.
- **relations** (*bool, str, or array_like, optional (default=False)*) – Indicates which types of relations to internalize as `graph_tool.PropertyMap`. Can be column name(s) of `e`, False or True. If False, create no propertymaps (all edges in `e.index` are transferred, regardless of `dropna`). If True, create propertymaps for every column in `e` (all edges in `e.index` are transferred, regardless of `dropna`). If str or array_like, must be column name(s) of `e` indicating which types of relations to internalize (which edges are transferred can be controlled by `dropna`).
- **dropna** (*str, optional (default='none')*) – One of {'none','any','all'}. If 'none', all edges in `e.index` are transferred. If 'any', drop all edges (rows) in `e[relations]` where any NA values are present. If 'all', drop all edges (rows) in `e[relations]` where all values are NA. Only has an effect if `relations` is str or array_like.

- **node_indices** (*bool, optional (default=False)*) – If True, internalize a vertex propertymap `i` with the original node indices.
- **edge_indices** (*bool, optional (default=False)*) – If True, internalize edge propertymaps `s` and `t` with the original source and target node indices of the edges, respectively.

Returns gt_g

Return type graph_tool.Graph

See also:

return_cs_graph(), *return_nx_graph()*, *return_nx_multigraph()*

Notes

If the index of `v` is not pd.RangeIndex(start=0,stop=len(`v`), step=1), the indices will be enumerated, which is expensive for large graphs.

Plotting Methods

DeepGraph.plot_2d(x, y[, edges, C, . . .])	Plot nodes and corresponding edges in 2 dimensions.
DeepGraph.plot_2d_generator(x, y, by[, . . .])	Plot nodes and corresponding edges by groups.
DeepGraph.plot_map(lon, lat[, edges, C, . . .])	Plot nodes and corresponding edges on a basemap.
DeepGraph.plot_map_generator(lon, lat, by[, . . .])	Plot nodes and corresponding edges by groups, on basemaps.
DeepGraph.plot_hist(x[, bins, log_bins, . . .])	Plot a histogram (or pdf) of x.
DeepGraph.plot_logfile(logfile)	Plot a logfile.

deepgraph.deepgraph.DeepGraph.plot_2d

DeepGraph.**plot_2d**(*x*, *y*, *edges=False*, *C=None*, *C_split_0=None*, *kwds_scatter=None*, *kwds_quiver=None*, *kwds_quiver_0=None*, *ax=None*)

Plot nodes and corresponding edges in 2 dimensions.

Create a scatter plot of the nodes in `v`, and optionally a quiver plot of the corresponding edges in `e`.

The xy-coordinates of the scatter plot are determined by the values of `v[x]` and `v[y]`, where `x` and `y` are column names of `v` (the arrow's coordinates are determined automatically).

In order to map colors to the arrows, either `C` or `C_split_0` can be be passed, an array of the same length as `e`. Passing `C` creates a single quiver plot (qu). Passing `C_split_0` creates two separate quiver plots, one for all edges where `C_split_0` == 0 (qu_0), and one for all other edges (qu). By default, the arrows of qu_0 have no head, indicating "undirected" edges. This can be useful, for instance, when `C_split_0` represents an array of temporal distances.

In order to control the plotting parameters of the scatter, quiver and/or quiver_0 plots, one may pass keyword arguments by setting `kwds_scatter`, `kwds_quiver` and/or `kwds_quiver_0`.

Can be used iteratively by passing `ax`.

Parameters

- **x** (*int or str*) – A column name of `v`, determining the x-coordinates of the scatter plot of nodes.

- **y** (*int or str*) – A column name of `v`, determining the y-coordinates of the scatter plot of nodes.
- **edges** (*bool, optional (default=True)*) – Whether to create a quiver plot (2-D field of arrows) of the edges between the nodes.
- **C** (*array_like, optional (default=None)*) – An optional array used to map colors to the arrows. Must have the same length es `e`. Has no effect if `C_split_0` is passed as an argument.
- **C_split_0** (*array_like, optional (default=None)*) – An optional array used to map colors to the arrows. Must have the same length es `e`. If this parameter is passed, `C` has no effect, and two separate quiver plots are created (qu and qu_0).
- **kwds_scatter** (*dict, optional (default=None)*) – kwargs to be passed to scatter.
- **kwds_quiver** (*dict, optional (default=None)*) – kwargs to be passed to quiver (qu).
- **kwds_quiver_0** (*dict, optional (default=None)*) – kwargs to be passed to quiver (qu_0). Only has an effect if `C_split_0` has been set.
- **ax** (*matplotlib axes object, optional (default=None)*) – An axes instance to use.

Returns **obj** – If `C_split_0` has been passed, return a dict of matplotlib objects with the following keys: ['fig', 'ax', 'pc', 'qu', 'qu_0']. Otherwise, return a dict with keys: ['fig', 'ax', 'pc', 'qu'].

Return type dict

Notes

When passing `C_split_0`, the color of the arrows in qu_0 can be set by passing the keyword argument *color* to `kwds_quiver_0`. The color of the arrows in qu, however, are determined by `C_split_0`.

The default drawing order is set to: 1. quiver_0 (zorder=1) 2. quiver (zorder=2) 3. scatter (zorder=3) This order can be changed by setting the `zorder` in `kwds_quiver_0`, `kwds_quiver` and/or `kwds_scatter`. See also http://matplotlib.org/examples/pylab_examples/zorder_demo.html

See also:

plot_2d_generator(), `plot_3d()`, *plot_map()*, *plot_map_generator()*

deepgraph.deepgraph.DeepGraph.plot_2d_generator

`DeepGraph.`**plot_2d_generator**(*x, y, by, edges=False, C=None, C_split_0=None, kwds_scatter=None, kwds_quiver=None, kwds_quiver_0=None, passable_ax=False*)

Plot nodes and corresponding edges by groups.

Create a generator of scatter plots of the nodes in `v`, split in groups by `v`.groupby(`by`). If edges is set True, also create a quiver plot of each group's corresponding edges.

The xy-coordinates of the scatter plots are determined by the values of `v[x]` and `v[y]`, where `x` and `y` are column names of `v` (the arrow's coordinates are determined automatically).

In order to map colors to the arrows, either `C` or `C_split_0` can be be passed, an array of the same length as `e`. Passing `C` creates a single quiver plot (qu). Passing `C_split_0` creates two separate quiver plots, one for all edges where `C_split_0` == 0 (qu_0), and one for all other edges (qu). By default, the arrows of qu_0 have no head, indicating "undirected" edges. This can be useful, for instance, when `C_split_0` represents an array of temporal distances.

When mapping colors to arrows by setting `C` (or `C_split_0`), *clim* is automatically set to the min and max values of the entire array. In case one wants clim to be set to min and max values for each group's colors, one may explicitly pass *clim* = None to `kwds_quiver`.

The same behaviour occurs when passing a sequence of `g.n` Numbers as colors *c* to `kwds_scatter`. In that case, *vmin* and *vmax* are automatically set to *c*.min() and *c*.max() of all nodes. Explicitly setting *vmin* and *vmax* to *None*, the min and max values of the groups' color arrays are used.

In order to control the plotting parameters of the scatter, quiver and/or quiver_0 plots, one may pass keyword arguments by setting `kwds_scatter`, `kwds_quiver` and/or `kwds_quiver_0`.

If `passable_ax` is True, create a generator of functions. Each function takes a matplotlib axes object as input, and returns a scatter/quiver plot.

Parameters

- **x** (*int or str*) – A column name of `v`, determining the x-coordinates of the scatter plot of nodes.
- **y** (*int or str*) – A column name of `v`, determining the y-coordinates of the scatter plot of nodes.
- **by** (*array_like*) – Column name(s) of `v`, determining the groups to create plots of.
- **edges** (*bool, optional (default=True)*) – Whether to create a quiver plot (2-D field of arrows) of the edges between the nodes.
- **C** (*array_like, optional (default=None)*) – An optional array used to map colors to the arrows. Must have the same length es `e`. Has no effect if `C_split_0` is passed as an argument.
- **C_split_0** (*array_like, optional (default=None)*) – An optional array used to map colors to the arrows. Must have the same length es `e`. If this parameter is passed, `C` has no effect, and two separate quiver plots are created (qu and qu_0).
- **kwds_scatter** (*dict, optional (default=None)*) – kwargs to be passed to scatter.
- **kwds_quiver** (*dict, optional (default=None)*) – kwargs to be passed to quiver (qu).
- **kwds_quiver_0** (*dict, optional (default=None)*) – kwargs to be passed to quiver (qu_0). Only has an effect if `C_split_0` has been set.
- **passable_ax** (*bool, optional (default=False)*) – If True, return a generator of functions. Each function takes a matplotlib axes object as input, and returns a dict of matplotlib objects.

Returns **obj** – If `C_split_0` has been passed, return a generator of dicts of matplotlib objects with the following keys: ['fig', 'ax', 'pc', 'qu', 'qu_0', 'group']. Otherwise, return a generator of dicts with keys: ['fig', 'ax', 'pc', 'qu', 'group']. If `passable_ax` is True, return a generator of functions. Each function takes a matplotlib axes object as input, and returns a dict as described above.

Return type generator

Notes

When passing `C_split_0`, the color of the arrows in qu_0 can be set by passing the keyword argument *color* to `kwds_quiver_0`. The color of the arrows in qu, however, are determined by `C_split_0`.

The default drawing order is set to: 1. quiver_0 (zorder=1) 2. quiver (zorder=2) 3. scatter (zorder=3) This order can be changed by setting the `zorder` in `kwds_quiver_0`, `kwds_quiver` and/or `kwds_scatter`. See also http://matplotlib.org/examples/pylab_examples/zorder_demo.html

See also:

`append_binning_labels_v()`, *`plot_2d()`*, `plot_3d()`, *`plot_map()`*, *`plot_map_generator()`*

deepgraph.deepgraph.DeepGraph.plot_map

`DeepGraph.`**`plot_map`**(*lon, lat, edges=False, C=None, C_split_0=None, kwds_basemap=None, kwds_scatter=None, kwds_quiver=None, kwds_quiver_0=None, ax=None, m=None*)

Plot nodes and corresponding edges on a basemap.

Create a scatter plot of the nodes in `v` and optionally a quiver plot of the corresponding edges in `e` on a `mpl_toolkits.basemap.Basemap` instance.

The coordinates of the scatter plot are determined by the node's longitudes and latitudes (in degrees): `v[lon]` and `v[lat]`, where `lon` and `lat` are column names of `v` (the arrow's coordinates are determined automatically).

In order to map colors to the arrows, either `C` or `C_split_0` can be be passed, an array of the same length as `e`. Passing `C` creates a single quiver plot (qu). Passing `C_split_0` creates two separate quiver plots, one for all edges where `C_split_0` == 0 (qu_0), and one for all other edges (qu). By default, the arrows of qu_0 have no head, indicating "undirected" edges. This can be useful, for instance, when `C_split_0` represents an array of temporal distances.

In order to control the parameters of the basemap, scatter, quiver and/or quiver_0 plots, one may pass keyword arguments by setting `kwds_basemap`, `kwds_scatter`, `kwds_quiver` and/or `kwds_quiver_0`.

Can be used iteratively by passing `ax` and/or `m`.

Parameters

- **`lon`** (*`int or str`*) – A column name of `v`. The corresponding values must be longitudes in degrees.
- **`lat`** (*`int or str`*) – A column name of `v`. The corresponding values must be latitudes in degrees.
- **`edges`** (*`bool, optional (default=True)`*) – Whether to create a quiver plot (2-D field of arrows) of the edges between the nodes.
- **`C`** (*`array_like, optional (default=None)`*) – An optional array used to map colors to the arrows. Must have the same length es `e`. Has no effect if `C_split_0` is passed as an argument.
- **`C_split_0`** (*`array_like, optional (default=None)`*) – An optional array used to map colors to the arrows. Must have the same length es `e`. If this parameter is passed, `C` has no effect, and two separate quiver plots are created (qu and qu_0).
- **`kwds_basemap`** (*`dict, optional (default=None)`*) – kwargs passed to basemap.

- **kwds_scatter** (*dict, optional (default=None)*) – kwargs to be passed to scatter.
- **kwds_quiver** (*dict, optional (default=None)*) – kwargs to be passed to quiver (qu).
- **kwds_quiver_0** (*dict, optional (default=None)*) – kwargs to be passed to quiver (qu_0). Only has an effect if `C_split_0` has been set.
- **ax** (*matplotlib axes object, optional (default=None)*) – An axes instance to use.
- **m** (*Basemap object, optional (default=None)*) – A mpl_toolkits.basemap.Basemap instance to use.

Returns **obj** – If `C_split_0` has been passed, return a dict of matplotlib objects with the following keys: ['fig', 'ax', 'm', 'pc', 'qu', 'qu_0']. Otherwise, return a dict with keys: ['fig', 'ax', 'm', 'pc', 'qu'].

Return type dict

Notes

When passing `C_split_0`, the color of the arrows in qu_0 can be set by passing the keyword argument *color* to `kwds_quiver_0`. The color of the arrows in qu, however, are determined by `C_split_0`.

The default drawing order is set to: 1. quiver_0 (zorder=1) 2. quiver (zorder=2) 3. scatter (zorder=3) This order can be changed by setting the `zorder` in `kwds_quiver_0`, `kwds_quiver` and/or `kwds_scatter`. See also http://matplotlib.org/examples/pylab_examples/zorder_demo.html

See also:

plot_map_generator(), *plot_2d()*, *plot_2d_generator()*, `plot_3d()`

deepgraph.deepgraph.DeepGraph.plot_map_generator

DeepGraph.**plot_map_generator**(*lon, lat, by, edges=False, C=None, C_split_0=None, kwds_basemap=None, kwds_scatter=None, kwds_quiver=None, kwds_quiver_0=None, passable_ax=False*)

Plot nodes and corresponding edges by groups, on basemaps.

Create a generator of scatter plots of the nodes in `v`, split in groups by v.groupby(by), on a `mpl_toolkits.basemap.Basemap` instance. If edges is set True, also create a quiver plot of each group's corresponding edges.

The coordinates of the scatter plots are determined by the node's longitudes and latitudes (in degrees): `v[lon]` and `v[lat]`, where `lon` and `lat` are column names of `v` (the arrow's coordinates are determined automatically).

In order to map colors to the arrows, either `C` or `C_split_0` can be be passed, an array of the same length as `e`. Passing `C` creates a single quiver plot (qu). Passing `C_split_0` creates two separate quiver plots, one for all edges where `C_split_0` == 0 (qu_0), and one for all other edges (qu). By default, the arrows of qu_0 have no head, indicating "undirected" edges. This can be useful, for instance, when `C_split_0` represents an array of temporal distances.

When mapping colors to arrows by setting `C` (or `C_split_0`), *clim* is automatically set to the min and max values of the entire array. In case one wants clim to be set to min and max values for each group's colors, one may explicitly pass *clim* = None to `kwds_quiver`.

The same behaviour occurs when passing a sequence of `g.n` Numbers as colors *c* to `kwds_scatter`. In that case, *vmin* and *vmax* are automatically set to *c*.min() and *c*.max() of all nodes. Explicitly setting *vmin* and *vmax* to *None*, the min and max values of the groups' color arrays are used.

In order to control the parameters of the basemap, scatter, quiver and/or quiver_0 plots, one may pass keyword arguments by setting `kwds_basemap`, `kwds_scatter`, `kwds_quiver` and/or `kwds_quiver_0`.

If `passable_ax` is True, create a generator of functions. Each function takes a matplotlib axes object (and/or a Basemap object) as input, and returns a scatter/quiver plot.

Parameters

- **`lon`** (*`int or str`*) – A column name of `v`. The corresponding values must be longitudes in degrees.
- **`lat`** (*`int or str`*) – A column name of `v`. The corresponding values must be latitudes in degrees.
- **`by`** (*`array_like`*) – Column name(s) of `v`, determining the groups to create plots of.
- **`edges`** (*`bool, optional (default=True)`*) – Whether to create a quiver plot (2-D field of arrows) of the edges between the nodes.
- **`C`** (*`array_like, optional (default=None)`*) – An optional array used to map colors to the arrows. Must have the same length es `e`. Has no effect if `C_split_0` is passed as an argument.
- **`C_split_0`** (*`array_like, optional (default=None)`*) – An optional array used to map colors to the arrows. Must have the same length es `e`. If this parameter is passed, `C` has no effect, and two separate quiver plots are created (qu and qu_0).
- **`kwds_basemap`** (*`dict, optional (default=None)`*) – kwargs passed to basemap.
- **`kwds_scatter`** (*`dict, optional (default=None)`*) – kwargs to be passed to scatter.
- **`kwds_quiver`** (*`dict, optional (default=None)`*) – kwargs to be passed to quiver (qu).
- **`kwds_quiver_0`** (*`dict, optional (default=None)`*) – kwargs to be passed to quiver (qu_0). Only has an effect if `C_split_0` has been set.
- **`passable_ax`** (*`bool, optional (default=False)`*) – If True, return a generator of functions. Each function takes a matplotlib axes object (and/or a Basemap object) as input, and returns a dict of matplotlib objects.

Returns **obj** – If `C_split_0` has been passed, return a generator of dicts of matplotlib objects with the following keys: ['fig', 'ax', 'm', 'pc', 'qu', 'qu_0', 'group']. Otherwise, return a generator of dicts with keys: ['fig', 'ax', 'm', 'pc', 'qu', 'group']. If `passable_ax` is True, return a generator of functions. Each function takes a matplotlib axes object (and/or a Basemap object) as input, and returns a dict as described above.

Return type generator

Notes

When passing `C_split_0`, the color of the arrows in qu_0 can be set by passing the keyword argument *color* to `kwds_quiver_0`. The color of the arrows in qu, however, are determined by `C_split_0`.

The default drawing order is set to: 1. quiver_0 (zorder=1) 2. quiver (zorder=2) 3. scatter (zorder=3) This order can be changed by setting the `zorder` in `kwds_quiver_0`, `kwds_quiver` and/or `kwds_scatter`. See also http://matplotlib.org/examples/pylab_examples/zorder_demo.html

See also:

`append_binning_labels_v()`, *`plot_map()`*, *`plot_2d()`*, *`plot_2d_generator()`*, `plot_3d()`

deepgraph.deepgraph.DeepGraph.plot_hist

static `DeepGraph.`**`plot_hist`**(*x*, *bins=10*, *log_bins=False*, *density=False*, *floor=False*, *ax=None*, ***kwargs*)

Plot a histogram (or pdf) of x.

Compute and plot the histogram (or probability density) of x. Keyword arguments are passed to plt.plot. See parameters and `np.histogram` for details.

Parameters

- **`x`** (*`array_like`*) – The data from which a frequency distribution is plot.
- **`bins`** (*`int or array_like, optional (default=10)`*) – If `bins` is an int, it determines the number of bins to create. If `log_bins` is True, this number determines the (approximate) number of bins to create for each magnitude. For linear bins, it is the number of bins for the whole range of values. If `bins` is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.
- **`log_bins`** (*`bool, optional (default=False)`*) – Whether to use logarithmically or linearly spaced bins.
- **`density`** (*`bool, optional (default=False)`*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability *density* function at the bin, normalized such that the *integral* over the range is 1. Note that the sum of the histogram values will not be equal to 1 unless bins of unity width are chosen; it is not a probability *mass* function.
- **`floor`** (*`bool, optional (default=False)`*) – Whether to floor the bin edges to the closest integers. Only has an effect if `bins` is an int.
- **`ax`** (*`matplotlib axes object, optional (default=None)`*) – An axes instance to use.

Returns

- **ax** (*matplotlib axes object*) – A matplotlib axes instance.
- **hist** (*np.ndarray*) – The values of the histogram. See `density`.
- **bin_edges** (*np.ndarray*) – The edges of the bins.

deepgraph.deepgraph.DeepGraph.plot_logfile

static `DeepGraph.`**`plot_logfile`**(*logfile*)

Plot a logfile.

Plot a benchmark logfile created by `create_edges` or `create_edges_ft`.

Parameters **`logfile`** (*`str`*) – The filename of the logfile.

Returns obj – Depending on the logfile, return a dict of matplotlib objects with a subset of the following keys: ['fig', 'ax', 'pc_n', 'pc_e', 'cb_n', 'cb_e']

Return type dict

Other Methods

DeepGraph.append_binning_labels_v(col, col_name)	Append a column with binning labels of the values in `v[col]`.
DeepGraph.append_cp([directed, connection, . . .])	Append a component membership column to `v`.
DeepGraph.filter_by_values_v(col, values)	Keep only nodes in `v` with features of type `col` in `values`.
DeepGraph.filter_by_values_e(col, values)	Keep only edges in `e` with relations of type `col` in `values`.
DeepGraph.filter_by_interval_v(col, interval)	Keep only nodes in `v` with features of type `col` in `interval`.
DeepGraph.filter_by_interval_e(col, interval)	Keep only edges in `e` with relations of type `col` in `interval`.
DeepGraph.update_edges()	After removing nodes in `v`, update `e`.

deepgraph.deepgraph.DeepGraph.append_binning_labels_v

`DeepGraph.`**`append_binning_labels_v`**(*col*, *col_name*, *bins=10*, *log_bins=False*, *floor=False*, *return_bin_edges=False*)

Append a column with binning labels of the values in `v[col]`.

Append a column `col_name` to `v` with the indices of the bins to which each value in `v[col]` belongs to.

If `bins` is an int, it determines the number of bins to create. If `log_bins` is True, this number determines the (approximate) number of bins to create for each magnitude. For linear bins, it is the number of bins for the whole range of values. If `floor` is set True, the bin edges are floored to the closest integer. If `return_bin_edges` is set True, the created bin edges are returned.

If `bins` is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.

See `np.digitize` for details.

Parameters

- **`col`** (*`int or str`*) – A column name of `v`, whose corresponding values are binned and labelled.
- **`col_name`** (*`str`*) – The column name for the created labels.
- **`bins`** (*`int or array_lke, optional (default=10)`*) – If `bins` is an int, it determines the number of bins to create. If `log_bins` is True, this number determines the (approximate) number of bins to create for each magnitude. For linear bins, it is the number of bins for the whole range of values. If `bins` is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.
- **`log_bins`** (*`bool, optional (default=False)`*) – Whether to use logarithmically or linearly spaced bins.
- **`floor`** (*`bool, optional (default=False)`*) – Whether to floor the bin edges to the closest integers.

- **return_bin_edges** (*bool, optional (default=False)*) – Whether to return the bin edges.

Returns

- **v** (*pd.DataFrame*) – Appends an extra column `col_name` to `v` with the binning labels.
- **bin_edges** (*np.ndarray*) – Optionally, return the created bin edges.

Examples

First, we need a node table:

```
>>> import pandas as pd
>>> import deepgraph as dg
>>> v = pd.DataFrame({'time': [1,2,12,105,899]})
>>> g = dg.DeepGraph(v)
```

```
>>> g.v
   time
0     1
1     2
2    12
3   105
4   899
```

Binning time values with default arguments:

```
>>> bin_edges = g.append_binning_labels_v('time', 'time_l',
...                                        return_bin_edges=True)
```

```
>>> bin_edges
array([   1.        ,  100.77777778,  200.55555556,  300.33333333,
        400.11111111,  499.88888889,  599.66666667,  699.44444444,
        799.22222222,  899.        ])
```

```
>>> g.v
   time  time_l
0     1       1
1     2       1
2    12       1
3   105       2
4   899      10
```

Binning time values with logarithmically spaced bins:

```
>>> bin_edges = g.append_binning_labels_v('time', 'time_l', bins=5,
...                                        log_bins=True,
...                                        return_bin_edges=True)
```

```
>>> bin_edges
array([   1.        ,    1.62548451,    2.64219989,    4.29485499,
          6.98122026,   11.34786539,   18.44577941,   29.9833287 ,
         48.73743635,   79.22194781,  128.77404899,  209.32022185,
        340.24677814,  553.06586728,  899.        ])
```

```
>>> g.v
   time  time_l
0     1       1
1     2       2
2    12       6
3   105      10
4   899      15
```

Binning time values with logarithmically spaced bins (floored):

```
>>> bin_edges = g.append_binning_labels_v('time', 'time_l', bins=5,
...                                       log_bins=True, floor=True,
...                                       return_bin_edges=True)
```

```
>>> bin_edges
array([   1.,    2.,    4.,    6.,   11.,   18.,   29.,   48.,   79.,
        128.,  209.,  340.,  553.,  899.])
```

```
>>> g.v
   time  time_l
0     1       1
1     2       2
2    12       5
3   105       9
4   899      14
```

deepgraph.deepgraph.DeepGraph.append_cp

`DeepGraph.`**`append_cp`**(*directed=False*, *connection='weak'*, *col_name='cp'*, *label_by_size=True*, *consolidate_singles=False*)

Append a component membership column to `v`.

Append a column to `v` indicating the component membership of each node. Requires scipy.

Parameters

- **`directed`** (*`bool, optional (default=False)`*) – If True , then operate on a directed graph: only move from point i to point j along paths csgraph[i, j]. If False, then find the shortest path on an undirected graph: the algorithm can progress from point i to j along csgraph[i, j] or csgraph[j, i].
- **`connection`** (*`str, optional (default='weak')`*) – One of {'weak','strong'}. For directed graphs, the type of connection to use. Nodes i and j are strongly connected if a path exists both from i to j and from j to i. Nodes i and j are weakly connected if only one of these paths exists. Only has an effect if `directed` is True
- **`col_name`** (*`str, optional (default='cp')`*) – The name of the appended column of component labels.
- **`label_by_size`** (*`bool, optional (default=True)`*) – Whether to rename component membership labels to reflect component sizes. If True, the smallest component corresponds to the largest label, and the largest component corresponds to the label 0 (or 1 if `consolidate_singles` is True). If False, pass on labels given by scipy's connected_components method directly (faster and uses less memory).
- **`consolidate_singles`** (*`bool, optional (default=False)`*) – If True, all singular components (components comprised of one node only) are consolidated un-

der the label 0. Also, all other labels are renamed to reflect component sizes, see `label_by_size`.

Returns **v** – appends an extra column to `v` indicating component membership.

Return type pd.DataFrame

deepgraph.deepgraph.DeepGraph.filter_by_values_v

`DeepGraph.`**`filter_by_values_v`**`(`*col*, *values*`)`

Keep only nodes in `v` with features of type `col` in `values`.

Remove all nodes from `v` (and their corresponding edges in `e`) with feature(s) of type `col` not in the list of features given by `values`.

Parameters

- **`col`** (*`str or int`*) – A column name of `v`, indicating the type of feature used in the filtering.
- **`values`** (*`object or array_like`*) – The value(s) indicating which nodes to keep.

Returns

- **v** (*pd.DataFrame*) – update `v`
- **e** (*pd.DataFrame*) – update `e`

deepgraph.deepgraph.DeepGraph.filter_by_values_e

`DeepGraph.`**`filter_by_values_e`**`(`*col*, *values*`)`

Keep only edges in `e` with relations of type `col` in `values`.

Remove all edges from `e` with relation(s) of type `col` not in the list of relations given by `values`.

Parameters

- **`col`** (*`str or int`*) – A column name of `e`, indicating the type of relation used in the filtering.
- **`values`** (*`object or array_like`*) – The value(s) indicating which edges to keep.

Returns **e** – update `e`

Return type pd.DataFrame

deepgraph.deepgraph.DeepGraph.filter_by_interval_v

`DeepGraph.`**`filter_by_interval_v`**`(`*col*, *interval*, *endpoint=True*`)`

Keep only nodes in `v` with features of type `col` in `interval`.

Remove all nodes from `v` (and their corresponding edges in `e`) with features of type `col` outside the interval given by a tuple of values. The endpoint is included, if `endpoint` is not set to False.

Parameters

- **`col`** (*`str or int`*) – A column name of `v`, indicating the type of feature used in the filtering.

- **interval** (*tuple*) – A tuple of two values, (value, larger_value). All nodes outside the interval are removed.
- **endpoint** (*bool, optional (default=True)*) – False excludes the endpoint.

Returns

- **v** (*pd.DataFrame*) – update v
- **e** (*pd.DataFrame*) – update e

deepgraph.deepgraph.DeepGraph.filter_by_interval_e

DeepGraph.**filter_by_interval_e**(*col*, *interval*, *endpoint=True*)

Keep only edges in e with relations of type col in interval.

Remove all edges from e with relations of type col outside the interval given by a tuple of values. The endpoint is included, if endpoint is not set to False.

Parameters

- **col** (*str or int*) – A column name of e, indicating the type of relation used in the filtering.
- **interval** (*tuple*) – A tuple of two values, (value, larger_value). All edges outside the interval are removed.
- **endpoint** (*bool, optional (default=True)*) – False excludes the endpoint.

Returns **e** – update e

Return type pd.DataFrame

deepgraph.deepgraph.DeepGraph.update_edges

DeepGraph.**update_edges**()

After removing nodes in v, update e.

If you deleted rows from v, you can remove all edges associated with the deleted nodes in e by calling this method.

Returns **e** – update e

Return type pd.DataFrame

1.4.2 The Functions Module

deepgraph.functions

Connector Functions

great_circle_dist
cp_node_intersection
cp_intersection_strength
hypergeometric_p_value

Selector Functions

1.5 Contact

1.5.1 Email

Please feel free to contact me if you have questions or suggestions regarding DeepGraph:

```
Dominik Traxl <dominik.traxl@posteo.org>
```

1.5.2 Authors

Deepgraph was written as part of a PhD thesis in physics by Dominik Traxl at Humboldt University Berlin, the Berstein Center for Computational Neuroscience and the Potsdam Institute for Climate Impact Research.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index