deepgraph Documentation

Release 0.2.1

Dominik Traxl

Sep 23, 2018
# Contents

1 Contents
   1.1 What is DeepGraph ................................................. 3
   1.2 Installation ......................................................... 4
   1.3 Tutorials ........................................................... 8
   1.4 API Reference ....................................................... 65
   1.5 Contact ............................................................. 103

2 Indices and tables ....................................................... 105
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 and cite the use of this software and its authors when results are used in publications or published elsewhere. You can use the following BibTex entry
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 requests, 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

```bash
$ 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

```bash
$ sudo pip install deepgraph
```

Alternatively, if you’re using Conda, install with
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 repository
   
   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

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

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

# the usual
import numpy as np
import pandas as pd

import deepgraph as dg

%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

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

```
  time    x         y    ball_id
0  0.0  1692.000000  0.000000    0
1  0.0  8681.000000  0.000000    1
2  0.0   490.000000  0.000000    2
3  0.0   7439.000000  0.000000    3
4  0.0   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    4
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

```python
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)
```
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:

```python
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`

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

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

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

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

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

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

Let’s compare the efficiency

```bash
%timeit -n 3 -r 3 g.create_edges(connectors=x_dist, selectors=x_dist_selector)
```

3 loops, best of 3: 557 ms per loop

```bash
%timeit -n 3 -r 3 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

```python
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
```

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

1.3. Tutorials
We can now plot the flying balls and the edges we just created with the `plot_2d` method:

```python
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]

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

```python
# 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
```python
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.

```python
# 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])})

# 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
```
Let's collect the computed correlation values and store them in an hdf file.

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

Let's have a quick look at the correlations.

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

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>-0.006066</td>
</tr>
<tr>
<td>2</td>
<td>0.094063</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-0.025529</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.074080</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.035490</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.005221</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.032064</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.000378</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>-0.049318</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>-0.084853</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0.026407</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0.028543</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>-0.013347</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>-0.180113</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0.151164</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>-0.094398</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>-0.124582</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>-0.000781</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>-0.044138</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>-0.193609</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>0.003877</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>0.048305</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>0.006477</td>
<td></td>
</tr>
</tbody>
</table>
And finally, let's see where most of the computation time is spent.

```python
>>> 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 memmap.py:335(__setitem__)  
```
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

```python
# 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
```

(continues on next page)
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.

- 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`

```python
# choose "wet times" threshold
r = .1

# choose "extreme" precipitation threshold
p = .9

v_list = []
```

(continues on next page)
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['longitude'] = vt['longitude'].astype(np.uint16)
        vt['latitude'] = vt['latitude'].astype(np.float16)
        vt['longitude'] = vt['longitude'].astype(np.float16)
        vt['latitude'] = vt['latitude'].astype(np.uint16)
        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()
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

```
  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
...  ...          ...        ...  ...  ...  ...  ...  ...  ...  ...
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
```

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,}  
```
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 $t_i1$ and $t_i + 1$).

For that matter, we pass the following connectors

```python
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

```python
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

```python
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)
```

To rename fast track relation

```python
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)
```

```
   s  t   dx  dy  dt
0  1362  0   1  False
1  1432  1   0  False
2  1433  1   1  False
3  1696  1   0   True
```

(continues on next page)
Logfile Plot

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

```python
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`
```
g.append_cp(consolidate_singles=True)
del g.e
```

the node table now has a component membership column appended

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

<table>
<thead>
<tr>
<th>lat</th>
<th>lon</th>
<th>dtime</th>
<th>r</th>
<th>x</th>
<th>y</th>
<th>area</th>
<th>vol</th>
<th>g_id</th>
<th>time</th>
<th>cp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15.125</td>
<td>2004-08-20</td>
<td>1084</td>
<td>28</td>
<td>101</td>
<td>743</td>
<td>24174</td>
<td>5652</td>
<td>0</td>
<td>865</td>
</tr>
<tr>
<td>1</td>
<td>44.875</td>
<td>2004-08-20</td>
<td>392</td>
<td>378</td>
<td>220</td>
<td>545</td>
<td>6433</td>
<td>85341</td>
<td>0</td>
<td>5079</td>
</tr>
<tr>
<td>2</td>
<td>45.125</td>
<td>2004-08-20</td>
<td>454</td>
<td>378</td>
<td>221</td>
<td>543</td>
<td>7416</td>
<td>85342</td>
<td>0</td>
<td>5079</td>
</tr>
<tr>
<td>3</td>
<td>45.375</td>
<td>2004-08-20</td>
<td>909</td>
<td>378</td>
<td>222</td>
<td>540</td>
<td>14767</td>
<td>85343</td>
<td>0</td>
<td>5079</td>
</tr>
<tr>
<td>4</td>
<td>45.625</td>
<td>2004-08-20</td>
<td>907</td>
<td>378</td>
<td>223</td>
<td>538</td>
<td>14669</td>
<td>85344</td>
<td>0</td>
<td>5079</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>382307</td>
<td>38.375</td>
<td>2004-09-27</td>
<td>453</td>
<td>352</td>
<td>194</td>
<td>603</td>
<td>8222</td>
<td>79095</td>
<td>304</td>
<td>0</td>
</tr>
<tr>
<td>382308</td>
<td>8.125</td>
<td>2004-09-27</td>
<td>509</td>
<td>80</td>
<td>73</td>
<td>762</td>
<td>11663</td>
<td>17007</td>
<td>304</td>
<td>174</td>
</tr>
<tr>
<td>382310</td>
<td>6.625</td>
<td>2004-09-27</td>
<td>192</td>
<td>56</td>
<td>67</td>
<td>764</td>
<td>4428</td>
<td>11790</td>
<td>304</td>
<td>15610</td>
</tr>
</tbody>
</table>

[382311 rows x 11 columns]

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

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

33169

and how many nodes there are in the components

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

<table>
<thead>
<tr>
<th>cp</th>
<th>value_counts()</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64678</td>
</tr>
<tr>
<td>1</td>
<td>16460</td>
</tr>
<tr>
<td>2</td>
<td>8519</td>
</tr>
<tr>
<td>3</td>
<td>6381</td>
</tr>
<tr>
<td>4</td>
<td>3403</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>29601</td>
<td>2</td>
</tr>
<tr>
<td>27554</td>
<td>2</td>
</tr>
<tr>
<td>25507</td>
<td>2</td>
</tr>
<tr>
<td>23460</td>
<td>2</td>
</tr>
<tr>
<td>20159</td>
<td>2</td>
</tr>
</tbody>
</table>

Name: cp, dtype: int64

**Partition the Nodes Into a Component Supernode Table**

In order to aggregate and compute some information about the precipitation 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

```python
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
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

```python
print(cpv)
```

<table>
<thead>
<tr>
<th>n_nodes</th>
<th>dtime_amin</th>
<th>dtime_amax</th>
<th>time_amin</th>
<th>time_amax</th>
<th>lat_0_mean</th>
<th>vol_sum</th>
<th>lon_mean</th>
<th>g_ids</th>
<th>n_unique_g_ids</th>
<th>dt</th>
<th>area</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp</td>
<td>64678</td>
<td>2004-08-20</td>
<td>2004-09-27</td>
<td>00:00:00</td>
<td>0</td>
<td>304</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>609375</td>
<td>2004-09-01</td>
<td>2004-09-17</td>
<td>00:00:00</td>
<td>98</td>
<td>230</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>21250</td>
<td>2004-09-17</td>
<td>2004-09-24</td>
<td>15:00:00</td>
<td>225</td>
<td>285</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>906250</td>
<td>2004-09-26</td>
<td>2004-09-26</td>
<td>03:00:00</td>
<td>51</td>
<td>137</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>625000</td>
<td>2004-08-24</td>
<td>2004-08-24</td>
<td>12:00:00</td>
<td>2242</td>
<td>1749673</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>578125</td>
<td>2004-08-21</td>
<td>2004-08-21</td>
<td>12:00:00</td>
<td>15</td>
<td>36</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>33165</td>
<td>2004-08-23</td>
<td>2004-08-23</td>
<td>18:00:00</td>
<td>134</td>
<td>143</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5000000</td>
<td>20212</td>
<td>103.875000</td>
<td>18:00:00</td>
<td>18115</td>
<td>18116</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>00:00:00</td>
<td>1483</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>33166</td>
<td>2</td>
<td>134</td>
<td>27</td>
<td>2868</td>
<td>2687</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>33167</td>
<td>2</td>
<td>1368</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2500000</td>
<td>43509</td>
<td>6.525000</td>
<td>85</td>
<td>85</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>00:00:00</td>
<td>1519</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(continues on next page)
Plot the Largest Component

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

```python
# 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)
```

(continues on next page)
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
code_time=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">
    </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**

```python
# 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

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

(continues on next page)
# 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 \( cpg.\text{n} \times (cpg.\text{n}-1)/2 \)

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

```
print(cpg.e)

<table>
<thead>
<tr>
<th>s</th>
<th>t</th>
<th>intsec_strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.018499</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.002457</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0.000000</td>
</tr>
<tr>
<td>4997</td>
<td>4999</td>
<td>0.000000</td>
</tr>
<tr>
<td>5000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4999</td>
<td>5000</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

[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

... 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.

```python
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

# 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

```python
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

```python
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**

```python
# 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)

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

[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)

(continues on next 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

```python
# 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

```python
# 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

```python
print(fgv)
```
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 {0}'.format(F))
Geographical Locations and Components

```python
# 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_func)

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

print(gcpv)
```
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))
```

[287301 rows x 4 columns]
1.3.4 From Multilayer Networks to Deep Graphs

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 correspond 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
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.


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

```python
# 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

```python
# 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%20Codebook%20ora%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)
# 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&Pajek%29.paj?
           "attredirects=0&amp;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():
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
epaper = e.copy()

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

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

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

Let’s have a look at the node table first

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

<table>
<thead>
<tr>
<th>V_N</th>
<th>layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Business</td>
</tr>
<tr>
<td>1</td>
<td>Business</td>
</tr>
<tr>
<td>2</td>
<td>Business</td>
</tr>
<tr>
<td>3</td>
<td>Business</td>
</tr>
<tr>
<td>4</td>
<td>Business</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>785</td>
<td>T Soulmates</td>
</tr>
<tr>
<td>786</td>
<td>T Soulmates</td>
</tr>
<tr>
<td>787</td>
<td>T Soulmates</td>
</tr>
<tr>
<td>788</td>
<td>T Soulmates</td>
</tr>
<tr>
<td>789</td>
<td>T Soulmates</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[790 rows x 2 columns]

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

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

```plaintext`
```

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,

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

<table>
<thead>
<tr>
<th>type</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>t</td>
</tr>
<tr>
<td>9</td>
<td>67</td>
</tr>
<tr>
<td>69</td>
<td>Business</td>
</tr>
<tr>
<td>77</td>
<td>Business</td>
</tr>
<tr>
<td>11</td>
<td>61</td>
</tr>
<tr>
<td>20</td>
<td>59</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>733</td>
<td>769</td>
</tr>
<tr>
<td>755</td>
<td>769</td>
</tr>
<tr>
<td>787</td>
<td>T Soulmates</td>
</tr>
</tbody>
</table>

(continues on next page)
is comprised of 1014 edges between the nodes in \( v \). Each edge has two relations. The first relation (of type \( \text{type} \)) is determined by the tuple of features \(( \text{layer}_i, \text{layer}_j )\) of the adjacent nodes \( V_i \) and \( V_j \). The second relation (of type \( \text{weight} \)) 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

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

```
```

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

```python
print(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

```python
# 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
```
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:

```python
# 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 \textit{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 \textit{layer} leads to the supergraph \( G^L = (V^L, E^L) \), where every supernode \( V^L_i \in V^L \) corresponds to a distinct layer, encompassing all its respective nodes. Superedges \( E^L_{i,j} \in E^L \) 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.

```python
# 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>
```

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

```
+----------------+--------+
| layer          | 79     |
| Business        | 79     |
| Communication   | 79     |
| O Logistics     | 79     |
| O Meetings      | 79     |
| O Operations    | 79     |
```

(continues on next page)
print(lg.e)

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

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
```
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',
```
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 $V^N_i \in V^N$ corresponds to a node of the MLN. Superedges $E^N_{ij} \in E^N$ with $i^N = j^N$ correspond to the coupling edges of a MLN.

```python
# 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
```

1.3. Tutorials
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

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 NaN 1.0 NaN NaN
1 4 1 NaN NaN NaN NaN NaN
→ NaN 1.0 NaN NaN NaN NaN NaN
5 1 NaN NaN NaN NaN NaN NaN
→ NaN 1.0 NaN NaN NaN NaN NaN
16 1 NaN NaN NaN NaN 2.0 NaN
→ NaN NaN NaN NaN NaN NaN NaN
21 1 NaN NaN NaN NaN NaN NaN
→ NaN 1.0 NaN NaN NaN NaN NaN
... ... ... ... ... ... ... ...
72 73 4 NaN 1.0 NaN NaN 2.0 NaN
→ 2.0 NaN NaN NaN NaN 1.0 NaN
76 6 NaN 1.0 NaN 2.0 2.0 NaN
→ 2.0 1.0 1.0 NaN NaN NaN
77 2 NaN NaN NaN 2.0 NaN NaN
→ NaN NaN NaN NaN NaN NaN
73 76 2 NaN NaN NaN 1.0 NaN NaN
→ 2.0 NaN NaN NaN NaN NaN
75 78 2 NaN NaN NaN NaN NaN NaN
→ 2.0 NaN 1.0 NaN NaN NaN
... ... ... ... ... ... ... ...
Let’s plot the graph \( g \) grouped by \( V_N \).

```python
# create graph_tool graph object
g.v['V_N'] = g.v['V_N'].astype(np.int32)  # sfdp only takes int32
g_tmp = dg.DeepGraph(g)
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)
```

1.3. Tutorials
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.

```python
# 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)
# 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 $G^{NL} = (V^{NL}, E^{NL})$ 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 $G^{NL}$.

```python
# 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>

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

<table>
<thead>
<tr>
<th>n_nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_N layer</td>
</tr>
<tr>
<td>Business</td>
</tr>
<tr>
<td>Communication</td>
</tr>
<tr>
<td>O Logistics</td>
</tr>
<tr>
<td>O Meetings</td>
</tr>
<tr>
<td>O Operations</td>
</tr>
<tr>
<td>... ...</td>
</tr>
<tr>
<td>O Training</td>
</tr>
<tr>
<td>T Classmates</td>
</tr>
<tr>
<td>T Friendship</td>
</tr>
<tr>
<td>T Kinship</td>
</tr>
<tr>
<td>T Soulmates</td>
</tr>
</tbody>
</table>

[790 rows x 1 columns]

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

<table>
<thead>
<tr>
<th>n_edges</th>
<th>weight</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_N_s layer_s</td>
<td>V_N_t layer_t</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Communication</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>O Logistics</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>T Kinship</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>O Operations</td>
<td>16</td>
</tr>
<tr>
<td>22</td>
<td>O Operations</td>
<td>1</td>
</tr>
<tr>
<td>... ... ... ... ... ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>T Soulmates</td>
<td>77</td>
</tr>
<tr>
<td>73</td>
<td>O Operations</td>
<td>76</td>
</tr>
<tr>
<td>0</td>
<td>O Training</td>
<td>76</td>
</tr>
<tr>
<td>75</td>
<td>O Training</td>
<td>78</td>
</tr>
<tr>
<td>T Friendship</td>
<td>78</td>
<td>T Friendship</td>
</tr>
</tbody>
</table>

[1014 rows x 3 columns]

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

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

<table>
<thead>
<tr>
<th>n_edges</th>
<th>weight</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_N_s layer_s</td>
<td>V_N_t layer_t</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Communication</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>Communication</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>Communication</td>
<td>1</td>
</tr>
<tr>
<td>58</td>
<td>Communication</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

1.3. Tutorials 55
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:

```python
print(vinfo)
```

<table>
<thead>
<tr>
<th>Education Level</th>
<th>Contact with People</th>
<th>Military Training</th>
<th>Nationality</th>
<th>Current Status (ICG Article)</th>
<th>Role</th>
<th>Primary Group</th>
<th>Affiliation</th>
<th>Noordin’s Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abdul Malik</td>
<td>0 1 7 5 0 3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abdul Rauf</td>
<td>1 2 10 3 0 3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abdul Rohim</td>
<td>2 0 1 9 0 3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abdullah Sunata</td>
<td>3 2 1 1 5 3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abdullah Sungkar</td>
<td>4 0 1 3 3 0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Umar Patek</td>
<td>785 2 12 5 3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Umar Wayan</td>
<td>786 2 1 7 3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Umar Urwah</td>
<td>787 2 4 3 3 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Usman bin Sef</td>
<td>788 2 10 11 3 3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zulkarnaen</td>
<td>789 2 1 7 4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

```python
# 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 positions

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.

```python
# 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)
# 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.

```python
# partition ng by "Education Level"
relation_funcs = {l: lambda x: x.null().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)
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)

<table>
<thead>
<tr>
<th>n_nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Education Level</td>
</tr>
<tr>
<td>0 25</td>
</tr>
<tr>
<td>1 1</td>
</tr>
<tr>
<td>2 39</td>
</tr>
<tr>
<td>3 5</td>
</tr>
<tr>
<td>4 5</td>
</tr>
<tr>
<td>5 1</td>
</tr>
<tr>
<td>6 2</td>
</tr>
<tr>
<td>8 1</td>
</tr>
</tbody>
</table>

print(ELng.e)

<table>
<thead>
<tr>
<th>n_edges dominant_type Business Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ 0 Logistics O Meetings O Operations O Training T Classmates T Friendship T Kinship T Soulmates</td>
</tr>
<tr>
<td>→ O Operations 0 7</td>
</tr>
<tr>
<td>→ 2 0 1 16 15 1 1</td>
</tr>
<tr>
<td>→ 3 0 2 1 0 0</td>
</tr>
<tr>
<td>→ 0 0 2 146 0 Operations 1 31</td>
</tr>
<tr>
<td>→ 3 7 43 32 9 16</td>
</tr>
<tr>
<td>→ 4 0 3 60 0 Operations 1 11</td>
</tr>
<tr>
<td>→ 1 0 2 14 19 2 9</td>
</tr>
<tr>
<td>→ 0 0 4 16 0 Operations 0 0</td>
</tr>
<tr>
<td>→ 0 0 0 6 9 1 0</td>
</tr>
<tr>
<td>... ... ... ... ... ... ... ... ...</td>
</tr>
</tbody>
</table>

1.3. Tutorials
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.

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

# get sfdp layout positions
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.

```python
# 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']))
```

```
<table>
<thead>
<tr>
<th>n_edges</th>
<th>Nationality_s</th>
</tr>
</thead>
</table>
```

(continues on next page)
```
print(g.partition_edges(source_features=['Nationality'], target_features=['Military Training']))
```

```
<table>
<thead>
<tr>
<th>n_edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nationality_s Military Training_t</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
```

```
[26 rows x 1 columns]
```

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

```
<table>
<thead>
<tr>
<th>n_edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>type Nationality_s Military Training_t</td>
</tr>
<tr>
<td>Business</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
| ...     | ...     | ...
| T Soulmates | 3   | 9 |
|            | 10     | 1 |
|            | 4     | 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)
```

```
<table>
<thead>
<tr>
<th>Business</th>
<th>Communication</th>
<th>O Logistics</th>
<th>O Meetings</th>
<th>O Operations</th>
<th>O Training</th>
<th>T Classmates</th>
<th>T Friendship</th>
<th>T Kinship</th>
<th>T Soulmates</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>67</td>
<td>2.0</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
```

(continues on next page)
As you can see, the edge table is also comprised of 1014 edges between the nodes in \( v \). However, every type of connection gets 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

\[
\text{DeepGraph}([v, e, supernode_labels_by, \ldots]) \quad \text{The core class of DeepGraph (dg).}
\]

**deepgraph.deepgraph.DeepGraph**

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**
  
  `pandas.DataFrame` – See Parameters.

- **e**
  
  `pandas.DataFrame` – See Parameters.

- **n**
  
  `int` – Property: Number of nodes.

- **m**
  
  `int` – Property: Number of edges.

- **f**
  
  `pd.DataFrame` – Property: types of features and number of features of corresponding type.

- **r**
  
  `pd.DataFrame` – Property: types of relations and number of relations of corresponding type.

```python
__init__(v=None, e=None, supernode_labels_by=None, superedge_labels_by=None)
```

`x.__init__(...) initializes x; see help(type(x)) for signature`
### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code></td>
<td>(<code>v, e, supernode_labels_by, …</code>)</td>
</tr>
<tr>
<td><code>append_binning_labels_v</code></td>
<td>(<code>col, col_name[, …]</code>)</td>
</tr>
<tr>
<td><code>append_cp</code></td>
<td>(<code>(directed, connection, col_name, …)</code>)</td>
</tr>
<tr>
<td><code>append_datetime_categories_v</code></td>
<td>(<code>[col, …]</code>)</td>
</tr>
<tr>
<td><code>create_edges</code></td>
<td>(<code>[connectors, selectors, …]</code>)</td>
</tr>
<tr>
<td><code>create_edges_ft</code></td>
<td>(<code>ft_feature[, connectors, …]</code>)</td>
</tr>
<tr>
<td><code>filter_by_interval_e</code></td>
<td>(<code>col, interval[, end-point]</code>)</td>
</tr>
<tr>
<td><code>filter_by_interval_v</code></td>
<td>(<code>col, interval[, end-point]</code>)</td>
</tr>
<tr>
<td><code>filter_by_values_e</code></td>
<td>(<code>col, values</code>)</td>
</tr>
<tr>
<td><code>filter_by_values_v</code></td>
<td>(<code>col, values</code>)</td>
</tr>
<tr>
<td><code>partition_edges</code></td>
<td>(<code>[relations, …]</code>)</td>
</tr>
<tr>
<td><code>partition_graph</code></td>
<td>(<code>features[, feature_funcs, …]</code>)</td>
</tr>
<tr>
<td><code>partition_nodes</code></td>
<td>(<code>features[, feature_funcs, …]</code>)</td>
</tr>
<tr>
<td><code>plot_2d</code></td>
<td>(<code>x, y[, edges, C, C_split_0, …]</code>)</td>
</tr>
<tr>
<td><code>plot_2d_generator</code></td>
<td>(<code>x, y, by[, edges, C, …]</code>)</td>
</tr>
<tr>
<td><code>plot_3d</code></td>
<td>(<code>x, y, z[, edges, kwds_scatter, …]</code>)</td>
</tr>
<tr>
<td><code>plot_hist</code></td>
<td>(<code>x[, bins, log_bins, density, …]</code>)</td>
</tr>
<tr>
<td><code>plot_logfile</code></td>
<td>(logfile)</td>
</tr>
<tr>
<td><code>plot_map</code></td>
<td>(<code>lon, lat[, edges, C, C_split_0, …]</code>)</td>
</tr>
<tr>
<td><code>plot_map_generator</code></td>
<td>(<code>lon, lat, by[, edges, C, …]</code>)</td>
</tr>
<tr>
<td><code>plot_raster</code></td>
<td>(<code>label[, time, ax]</code>)</td>
</tr>
<tr>
<td><code>plot_rects_label_numeric</code></td>
<td>(<code>label, xl, xr[, …]</code>)</td>
</tr>
<tr>
<td><code>plot_rects_numeric_numeric</code></td>
<td>(<code>yb, yt, xl, xr[, …]</code>)</td>
</tr>
<tr>
<td><code>return_cs_graph</code></td>
<td>(<code>[relations, dropna]</code>)</td>
</tr>
<tr>
<td><code>return_gt_graph</code></td>
<td>(<code>[features, relations, …]</code>)</td>
</tr>
<tr>
<td><code>return_motif_graph</code></td>
<td>(<code>[thresholds, …]</code>)</td>
</tr>
<tr>
<td><code>return_nx_graph</code></td>
<td>(<code>[features, relations, dropna]</code>)</td>
</tr>
<tr>
<td><code>return_nx_multigraph</code></td>
<td>(<code>[features, relations, …]</code>)</td>
</tr>
<tr>
<td><code>return_sparse_tensor</code></td>
<td>(<code>relations[, dropna]</code>)</td>
</tr>
<tr>
<td><code>update_edges</code></td>
<td>()</td>
</tr>
</tbody>
</table>

### Attributes

1.4. API Reference
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

```python
>>> 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

```python
>>> 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

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

```python
>>> g.e
dt
   s  t  dt
0  1  2
```
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

```python
>>> 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

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

```plaintext
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

```python
>>> 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

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

```plaintext
g.e
   dt
  s  t
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 \( e \)

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

```python
>>> 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 remaining 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
initiate a DeepGraph instance with the store

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

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

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

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

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
```

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:

```python
>>> 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 conjunction 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 measurements 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 output 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.

```python
>>> 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., \( f(s, t) \neq f(t, s) \)), one can always create an additional connector (or output variable) that computes the mirrored values of that function.

```python

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

```python
>>> v = pd.DataFrame({'time': [-3.6, -1.1, 1.4, 4., 6.3],
                     'x': [-3., 3., 1., 12., 7.1]})
```

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

```python
>>> 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

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

```python
>>> g.e
   ft_r
data  
s t
0  1  2.5
  2  5.0
  3  2.6
  4  4.9
  5  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. Its 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 ($n = \text{min\_chunk\_size}$, 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 $d_{ftf} \leq ftt$, $n$ 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
we can compute all relations of the source nodes in $[1,3]$ by

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

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

```python
>>> 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

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

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

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

```
  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 one 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 exemplary 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 conjunction 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 conjunction 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 measurements 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 capabilities 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):

```python
>>> 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

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

one stores it as recommended

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

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

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

and then sorts it on disc with

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

(continues on next page)
2. To speed up the internal queries on the fast-track feature

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


### 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: 

```python
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*.


#### 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](http://stackoverflow.com/questions/18429491/groupby-columns-with-nan-missing-values) and [https://github.com/pydata/pandas/issues/3729](https://github.com/pydata/pandas/issues/3729).

**Examples**

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

```python
>>> 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’:

```python
>>> 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):

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

Partition by ‘color’ and collect x values:
Partition by ‘color’ and aggregate with different functions:

```python
g.partition_nodes('color', {'time': lambda x: list(x)},
  'x': [np.mean, np.sum, np.std])
```

<table>
<thead>
<tr>
<th>n_nodes</th>
<th>x_mean</th>
<th>x_sum</th>
<th>x_std</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>-1.10000</td>
<td>-1.1</td>
<td>NaN</td>
</tr>
<tr>
<td>g</td>
<td>-0.13333</td>
<td>-0.4</td>
<td>2.891943</td>
</tr>
<tr>
<td>r</td>
<td>2.30000</td>
<td>2.3</td>
<td>NaN</td>
</tr>
</tbody>
</table>

**deepgraph.deepgraph.DeepGraph.partition_edges**

DeepGraph.partition_edges

The `DeepGraph.partition_edges` 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 one 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`.


**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](http://stackoverflow.com/questions/18429491/groupby-columns-with-nan-missing-values) and [https://github.com/pydata/pandas/issues/3729](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:

```python
>>> 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)
```

```
     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:

```python
>>> 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
```

(continues on next page)
... return dx, v, same_color, larger_than

```python
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
```

```plaintext
>>> 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’:

```plaintext
>>> g.partition_edges(relations='larger_than',
...                        relation_funcs={'dx': ['mean', 'std'],
...                                                    'same_color': 'sum'})
```

```plaintext
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’:

```plaintext
>>> g.partition_edges(relations=['larger_than', 'same_color'],
...                        relation_funcs={'dx': ['mean', 'std'],
...                                                    'dt': lambda x: tuple(x)})
```

```plaintext
n_edges dt_<lambda> dx_mean dx_std
larger_than same_color
False False  2 (2, 3)  2.150000  0.212132
True  True  3 (1, 5, 4)  2.866667  3.572581
False  True  2 (1, 4) -0.900000  3.252691
```

Partitioning by the type of source feature ‘color’:

```plaintext
>>> g.partition_edges(source_features='color',
...                        relation_funcs={'same_color': 'sum'})
```

```plaintext
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:

```plaintext
>>> g.e
```

```plaintext
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
1 2 -3.2  1  True  False -3.200000 (1, 2)  g
3 -1.2  4  False  True -0.300000 (1, 3)  g
```

(continues on next page)
A further refinement of the last partition by the type of source feature ‘size’:

```python
>>> g.partition_edges(source_features=['color', 'size'],
                      relation_funcs={'same_color': 'sum',
                                      'inds': lambda x: tuple(x)})
```

<table>
<thead>
<tr>
<th>color_s</th>
<th>size_s</th>
<th>n_edges</th>
<th>same_color</th>
<th>inds</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>((2, 3),)</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>((0, 1), (0, 2), (0, 3))</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>((1, 2), (1, 3), (3, 4))</td>
</tr>
</tbody>
</table>

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

```python
g.partition_edges(target_features=['color', 'size'],
                   relation_funcs={'same_color': 'sum',
                                    'inds': lambda x: tuple(x)})
```

<table>
<thead>
<tr>
<th>color_t</th>
<th>size_t</th>
<th>n_edges</th>
<th>same_color</th>
<th>inds</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>((0, 2), (1, 2))</td>
</tr>
<tr>
<td>g</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>((0, 1), (0, 3), (1, 3), (2, 3))</td>
</tr>
<tr>
<td>r</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>((3, 4),)</td>
</tr>
</tbody>
</table>

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

```python
>>> g.partition_edges(source_features='color', target_features='size',
                      relation_funcs={'same_color': 'sum',
                                      'inds': lambda x: tuple(x)})
```

<table>
<thead>
<tr>
<th>color_s</th>
<th>size_t</th>
<th>n_edges</th>
<th>same_color</th>
<th>inds</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>((2, 3),)</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>((3, 4),)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>((0, 2), (1, 2))</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>((0, 1), (0, 3), (1, 3))</td>
</tr>
</tbody>
</table>

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

```python
>>> g.partition_edges(relations='larger_than',
                      source_features='color', target_features='size',
                      relation_funcs={'inds': lambda x: tuple(x)})
```

<table>
<thead>
<tr>
<th>larger_than</th>
<th>color_s</th>
<th>size_t</th>
<th>n_edges</th>
<th>inds</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>b</td>
<td>3</td>
<td>1</td>
<td>((2, 3),)</td>
</tr>
<tr>
<td></td>
<td>g</td>
<td>2</td>
<td>1</td>
<td>((0, 2),)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>((0, 1), (0, 3), (1, 3))</td>
</tr>
<tr>
<td>True</td>
<td>g</td>
<td>1</td>
<td>1</td>
<td>((3, 4),)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>((1, 2),)</td>
</tr>
</tbody>
</table>

depthgraph.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 s_v and s_e.
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 \( 
\text{help(g.v.groupby)}, \text{and/or inspect the available methods of gv.} 
\)


### 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: \( \text{help(gv.agg)} \).

- **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: \( \text{help(ge.agg)} \).

- **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 aggregated 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, \( e.groupby(features_i+features_j) \).

See also:

- \( \text{partition_nodes()}, \text{partition_edges()} \)
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:

```python
>>> 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)
```

Create an edge table:

```python
>>> 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
```

Create a supergraph by partitioning by the type of feature ‘color’:

```python
>>> sv, se = g.partition_graph('color')
```
Create intersection partitions by the types of features ‘color’ and ‘size’ (which are further refinements of the last partitions):

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

Partition by ‘color’ and aggregate some properties:

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

depdeepgraph.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 \( e[\text{relations}] \) (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()`, `return_sparse_tensor()`

#### 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 \( e[\text{relations}] \) where any NA values are present. If 'all', drop all edges (rows) in \( e[\text{relations}] \) 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 node indices.
• `edge_indices` (bool, optional (default=False)) – If True, internalize edge propertysmaps `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()`, `return_sparse_tensor()`

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DeepGraph.plot_2d(x, y[, edges, C, ...])</code></td>
<td>Plot nodes and corresponding edges in 2 dimensions.</td>
</tr>
<tr>
<td><code>DeepGraph.plot_2d_generator(x, y, by[, ...])</code></td>
<td>Plot nodes and corresponding edges by groups.</td>
</tr>
<tr>
<td><code>DeepGraph.plot_map(lon, lat[, edges, C, ...])</code></td>
<td>Plot nodes and corresponding edges on a basemap.</td>
</tr>
<tr>
<td><code>DeepGraph.plot_map_generator(lon, lat, by[, ...])</code></td>
<td>Plot nodes and corresponding edges by groups, on basemaps.</td>
</tr>
<tr>
<td><code>DeepGraph.plot_hist(x[, bins, log_bins, ...])</code></td>
<td>Plot a histogram (or pdf) of <code>x</code>.</td>
</tr>
<tr>
<td><code>DeepGraph.plot_logfile(logfile)</code></td>
<td>Plot a logfile.</td>
</tr>
</tbody>
</table>

`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 as 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 as 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

```
Dee...e as 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 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()

deeppgraph.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 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 as `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 as `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].

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_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.
- **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:

```python
>>> import pandas as pd
>>> import deepgraph as dg

v = pd.DataFrame({'time': [1, 2, 12, 105, 899]})

g = dg.DeepGraph(v)
```

Binning time values with default arguments:

```python
>>> 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.        ])
```

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

Binning time values with logarithmically spaced bins:

```python
>>> 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.        ])
```

```python
>>> 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 (floored):

```python
>>> 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.        ])
```

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

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

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

depthgraph.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 under 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 Documentation, Release 0.2.1

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 \( \text{endpoint} \) is not set to False.

Parameters

• \( \text{col} (\text{str or int}) \) – A column name of \( e \), indicating the type of relation used in the filtering.

• \( \text{interval} (\text{tuple}) \) – A tuple of two values, (value, larger_value). All edges outside the interval are removed.

• \( \text{endpoint} (\text{bool}, \text{optional (default=True)}) \) – 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

Auxiliary connector and selector functions to create edges.

Connector Functions

great_circle_dist (lat_s, lat_t, lon_s, lon_t)

Return the great circle distance between nodes.

The latitude and longitude values in the node table have to be in signed decimal degrees without compass direction (the sign indicates west/south). The great circle distance is calculated using the spherical law of
cosines.

**deepgraph.functions.cp_node_intersection**

```python
cp_node_intersection(supernode_ids, sources, targets)
```
Work in progress!

**deepgraph.functions.cp_intersection_strength**

```python
cp_intersection_strength(n_unique_nodes, intsec_card, sources, targets)
```
Work in progress!

**deepgraph.functions.hypergeometric_p_value**

```python
hypergeometric_p_value(n_unique_nodes, intsec_card, sources, targets)
```
Work in progress!

**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
## Symbols

<table>
<thead>
<tr>
<th><strong>init</strong>() (DeepGraph method), 66</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>append_binning_labels_v() (DeepGraph method), 98</td>
</tr>
<tr>
<td>append_cp() (DeepGraph method), 100</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>cp_intersection_strength() (in module deepgraph.functions), 103</td>
</tr>
<tr>
<td>cp_node_intersection() (in module deepgraph.functions), 103</td>
</tr>
<tr>
<td>create_edges() (DeepGraph method), 68</td>
</tr>
<tr>
<td>create_edges_ft() (DeepGraph method), 73</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>DeepGraph (class in deepgraph.deepgraph), 65</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>e (DeepGraph attribute), 66</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>f (DeepGraph attribute), 66</td>
</tr>
<tr>
<td>filter_by_interval_e() (DeepGraph method), 102</td>
</tr>
<tr>
<td>filter_by_interval_v() (DeepGraph method), 101</td>
</tr>
<tr>
<td>filter_by_values_e() (DeepGraph method), 101</td>
</tr>
<tr>
<td>filter_by_values_v() (DeepGraph method), 101</td>
</tr>
<tr>
<td>G</td>
</tr>
<tr>
<td>great_circle_dist() (in module deepgraph.functions), 102</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>hypergeometric_p_value() (in module deepgraph.functions), 103</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>m (DeepGraph attribute), 66</td>
</tr>
</tbody>
</table>

## N

| n (DeepGraph attribute), 66 |

## P

<table>
<thead>
<tr>
<th>partition_edges() (DeepGraph method), 81</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition_graph() (DeepGraph method), 84</td>
</tr>
<tr>
<td>partition_nodes() (DeepGraph method), 79</td>
</tr>
<tr>
<td>plot_2d() (DeepGraph method), 91</td>
</tr>
<tr>
<td>plot_2d_generator() (DeepGraph method), 92</td>
</tr>
<tr>
<td>plot_hist() (DeepGraph static method), 97</td>
</tr>
<tr>
<td>plot_logfile() (DeepGraph static method), 97</td>
</tr>
<tr>
<td>plot_map() (DeepGraph method), 94</td>
</tr>
<tr>
<td>plot_map_generator() (DeepGraph method), 95</td>
</tr>
</tbody>
</table>

## R

<table>
<thead>
<tr>
<th>r (DeepGraph attribute), 66</th>
</tr>
</thead>
<tbody>
<tr>
<td>return_cs_graph() (DeepGraph method), 88</td>
</tr>
<tr>
<td>return_gt_graph() (DeepGraph method), 90</td>
</tr>
<tr>
<td>return_nx_graph() (DeepGraph method), 89</td>
</tr>
<tr>
<td>return_nx_multigraph() (DeepGraph method), 89</td>
</tr>
</tbody>
</table>

## U

| update_edges() (DeepGraph method), 102 |

## V

| v (DeepGraph attribute), 66 |