spaghetti Documentation

Release 1.3.1

pysal developers

Aug 04, 2019

CONTENTS:

CHAPTER

ONE

SPATIAL GRAPHS: NETWORKS, TOPOLOGY, & INFERENCE

Spaghetti is an open-source python library for the analysis of network-based spatial data. Originating from the `network` module in PySAL (Python Spatial Analysis Library), it is under active development for the inclusion of newly proposed methods for building graph-theoretic networks and the analysis of network events.

CHAPTER

TWO

CITING SPAGHETTI

If you use PySAL-spaghetti in a scientific publication, we would appreciate using the following citation:

Bibtex entry:

```
@misc{Gaboardi2018,
  author = {Gaboardi, James D. and Laura, Jay and Rey, Sergio and Wolf, Levi␣
↪John and Folch, David C. and Kang, Wei and Stephens, Philip and Schmidt,␣
↪Charles},
  month = {oct},
  year = {2018},
  title = {pysal/spaghetti},
  url = {https://github.com/pysal/spaghetti},
  keywords = {graph-theory,network-analysis,python,spatial-networks,topology}
}
```

As of version 1.3, spaghetti supports Python 3.6 and 3.7 only. Please make sure that you are operating in a Python 3 environment.

2.1 Installation

2.1.1 Installing with `conda` via spaghetti-feedstock (highly recommended)

To install `spaghetti` and all its dependencies, we recommend using the conda manager, specifically with the conda-forge channel. This can be obtained by installing the Anaconda Distribution (a free Python distribution for data science), or through miniconda (minimal distribution only containing Python and the `conda` package manager).

Using *conda*, *spaghetti* can be installed as follows:

```
$ conda config --set channel_priority strict
$ conda install --channel conda-forge spaghetti
```

2.1.2 Installing with Python Package Index

```
$ pip install spaghetti
```

or download the source distribution (`.tar.gz`) and decompress it to your selected destination. Open a command shell and navigate to the decompressed folder.

```
$ pip install .
```

Warning

When installing via *pip*, you have to ensure that the required dependencies for *spaghetti* are installed on your operating system. Details on how to install these packages are linked here. Using *conda* (above) avoids having to install the dependencies separately.

Install the most current development version of *spaghetti* by running:

```
$ pip install git+https://github.com/pysal/spaghetti
```

2.1.3 Development Version

Install the most current development version of spaghetti by running:

```
$ pip install git+https://github.com/pysal/spaghetti
```

You can also fork the pysal/spaghetti repo and create a local clone of your fork. By making changes to your local clone and submitting a pull request to pysal/spaghetti, you can contribute to the spaghetti development.

2.2 API reference

2.2.1 spaghetti.Network

spaghetti.Network.extract_components(self, w)	Extract connected component information from a `libpysal.weights.weights.W` object
spaghetti.Network.extractgraph(self)	Using the existing network representation, create a graph-theoretic representation by removing all vertices with a neighbor incidence of two (non-articulation points).
spaghetti.Network.contiguityweights(self[, ...])	Create a contiguity-based libpysal W object.
spaghetti.Network.distancebandweights(self, ...)	Create distance based weights.
spaghetti.Network.snapobservations(self, ...)	Snap a point pattern shapefile to network object.
spaghetti.Network.compute_distance_to_vertices(...)	Given an observation on a network arc, return the distance to the two vertices that bound that end.
spaghetti.Network.compute_snap_dist(self, ...)	Given an observation snapped to a network arc, calculate the distance from the original location to the snapped location.
spaghetti.Network.count_per_link(self, obs_on)	Compute the counts per arc or edge (link).
spaghetti.Network.simulate_observations(...)	Generate a simulated point pattern on the network.
spaghetti.Network.enum_links_vertex(self, v0)	Returns the arcs (links) around vertices.

Continued on next page

Table 1 – continued from previous page

spaghetti.Network.full_distance_matrix(self, ...)	All vertex-to-vertex distances on a network.
spaghetti.Network.allneighbordistances(self, ...)	Compute either all distances between `i` and `j` in a single point pattern or all distances between each `i` from a source pattern and all `j` from a destination pattern.
spaghetti.Network.nearestneighbordistances(...)	Compute the interpattern nearest neighbor distances or the intrapattern nearest neighbor distances between a source pattern and a destination pattern.
spaghetti.Network.split_arcs(self, distance)	Split all of the arcs in the network at either a
spaghetti.Network.savenetwork(self, filename)	Save a network to disk as a binary file.
spaghetti.Network.loadnetwork(filename)	Load a network from a binary file saved on disk.
spaghetti.Network.NetworkF(self, pointpattern)	Computes a network constrained F-Function
spaghetti.Network.NetworkG(self, pointpattern)	Computes a network constrained G-Function
spaghetti.Network.NetworkK(self, pointpattern)	Computes a network constrained K-Function
spaghetti.Network._evaluate_napts(self, ...)	Evaluate one connected component in a network for non-articulation points (napts) and return an updated set of napts and unvisted vertices.
spaghetti.Network._extractnetwork(self)	Used internally to extract a network from a polyline shapefile of a `geopandas.GeoDataFrame`.
spaghetti.Network._newpoint_coords(self, ...)	Used internally to compute new point coordinates during snapping.
spaghetti.Network._round_sig(self, v)	Used internally to round the vertex to a set number of significant digits.
spaghetti.Network._snap_to_link(self, ...)	Used internally to snap point observations to network arcs.
spaghetti.Network._yield_napts(self)	Find all nodes with degree 2 that are not in an isolated island ring (loop) component.
spaghetti.Network._yieldneighbor(self, vtx, ...)	Used internally, this method traverses a bridge arc to find the source and destination nodes.

spaghetti.Network.extract_components

`Network.`**`extract_components`**(*self*, *w*, *graph=False*)

Extract connected component information from a `libpysal.weights.weights.W` object

Parameters

w [libpysal.weights.weights.W] Weights object created from the network segments (either raw or graph-theoretic)

graph [bool] Flag for raw network [False] or graph-theoretic network `True`. Default is `False`.

spaghetti.Network.extractgraph

`Network.`**`extractgraph`**(*self*)

Using the existing network representation, create a graph-theoretic representation by removing all vertices with a neighbor incidence of two (non-articulation points). That is, we assume these vertices are bridges between vertices with higher or lower incidence.

spaghetti.Network.contiguityweights

`Network.`**`contiguityweights`**(*self*, *graph=True*, *weightings=None*)

Create a contiguity-based libpysal W object.

Parameters

graph [bool] `{True, False}` controls whether the W is generated using the spatial representation or the graph representation. Default is `True`.

weightings [dict] dictionary of lists of weightings for each arc/edge.

Returns

W [libpysal.weights.weights.W] A `pysal` W Object representing the binary adjacency of the network.

Examples

Instantiate an instance of a network.

```
>>> import spaghetti as spgh
>>> from libpysal import examples
>>> import esda
>>> import numpy as np
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
```

Snap point observations to the network with attribute information.

```
>>> ntw.snapobservations(examples.get_path('crimes.shp'),
...                      'crimes', attribute=True)
```

Find counts per network arc.

```
>>> counts = ntw.count_per_link(ntw.pointpatterns['crimes']
...                             .obs_to_arc, graph=False)
>>> counts[(50, 165)]
4
```

Create a contiguity based W object.

```
>>> w = ntw.contiguityweights(graph=False)
```

Using the W object, access to ESDA functionality is provided. First, a vector of attributes is created for all edges with observations.

```
>>> w = ntw.contiguityweights(graph=False)
>>> arcs = w.neighbors.keys()
>>> y = np.zeros(len(arcs))
>>> for i, e in enumerate(arcs):
...     if e in counts.keys():
...         y[i] = counts[e]
>>> y[3]
3.0
```

Next, a standard call ot Moran is made and the result placed into `res`.

```
>>> res = esda.moran.Moran(y, w, permutations=99)
>>> type(res)
<class 'esda.moran.Moran'>
```

spaghetti.Network.distancebandweights

`Network.`**`distancebandweights`**(*self*, *threshold*, *n_proccess=None*, *gen_tree=False*)

Create distance based weights.

Parameters

threshold [float] Distance threshold value.

n_processes [{int, str}] (Optional) Specify the number of cores to utilize. Default is 1 core. Use `int` to specify an exact number or cores. Use `"all"` to request all available cores.

gen_tree [bool] Rebuild shortest path with `True`, or skip with `False`.

Returns

w [libpysal.weights.weights.W] A `pysal` W Object representing the binary adjacency of the network.

Examples

```
>>> import spaghetti as spgh
>>> streets_file = examples.get_path('streets.shp')
>>> ntw = spgh.Network(in_data=streets_file)
>>> w = ntw.distancebandweights(threshold=500)
>>> w.n
230
>>> w.histogram[-1]
(8, 3)
```

spaghetti.Network.snapobservations

`Network.`**`snapobservations`**(*self*, *in_data*, *name*, *idvariable=None*, *attribute=None*)

Snap a point pattern shapefile to network object. The point pattern is stored in the `network.pointpattern['key']` attribute of the network object.

Parameters

in_data [{geopandas.GeoDataFrame, str}] The input geographic data. Either (1) a path to a shapefile (`str`); or (2) a `geopandas.GeoDataFrame`.

name [str] Name to be assigned to the point dataset.

idvariable [str] Column name to be used as ID variable.

attribute [bool] Defines whether attributes should be extracted. `True` for attribute extraction. `False` for no attribute extraction.

Examples

```
>>> import spaghetti as spgh
>>> streets_file = examples.get_path('streets.shp')
>>> ntw = spgh.Network(in_data=streets_file)
>>> pt_str = 'crimes'
>>> in_data = examples.get_path('{}.shp'.format(pt_str))
>>> ntw.snapobservations(in_data, pt_str, attribute=True)
>>> ntw.pointpatterns[pt_str].npoints
287
```

spaghetti.Network.compute_distance_to_vertices

Network.**compute_distance_to_vertices**(*self*, *x*, *y*, *arc*)

Given an observation on a network arc, return the distance to the two vertices that bound that end.

Parameters

x [float] x-coordinate of the snapped point.

y [float] y-coordinate of the snapped point.

arc [tuple] (vtx0, vtx1) representation of the network arc.

Returns

d1 [float] The distance to vtx0. Always the vertex with the lesser id.

d2 [float] The distance to vtx1. Always the vertex with the greater id.

spaghetti.Network.compute_snap_dist

Network.**compute_snap_dist**(*self*, *pattern*, *idx*)

Given an observation snapped to a network arc, calculate the distance from the original location to the snapped location.

Parameters

pattern [spaghetti.network.PointPattern] point pattern object

idx [int] point id

Returns

dist [float] euclidean distance from original location to snapped location.

spaghetti.Network.count_per_link

Network.**count_per_link**(*self*, *obs_on*, *graph=True*)

Compute the counts per arc or edge (link).

Parameters

obs_on_network [dict] Dictionary of observations on the network. Either {(link):{pt_id:(coords)}} or {link:[(coord),(coord),(coord)]}

Returns

counts [dict] {(link):count}

Examples

Note that this passes the obs_to_arc or obs_to_edge attribute of a point pattern snapped to the network.

```
>>> import spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> ntw.snapobservations(examples.get_path('crimes.shp'),
...                                         'crimes',
...                                          attribute=True)
```

```
>>> counts = ntw.count_per_link(ntw.pointpatterns['crimes']
...                                .obs_to_arc, graph=False)
>>> counts[(140, 142)]
10
```

```
>>> s = sum([v for v in list(counts.values())])
>>> s
287
```

spaghetti.Network.simulate_observations

Network.**simulate_observations**(*self*, *count*, *distribution='uniform'*)

Generate a simulated point pattern on the network.

Parameters

count [int] The number of points to create or mean of the distribution if not 'uniform'.

distribution [str] {'uniform', 'poisson'} distribution of random points. If "poisson", the distribution is calculated from half the total network length.

Returns

random_pts [dict] Keys are the edge tuple. Values are lists of new point coordinates.

Examples

```
>>> import spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> ntw.snapobservations(examples.get_path('crimes.shp'),
...                                         'crimes',
...                                          attribute=True)
```

```
>>> npts = ntw.pointpatterns['crimes'].npoints
>>> sim = ntw.simulate_observations(npts)
>>> isinstance(sim, spgh.network.SimulatedPointPattern)
True
```

spaghetti.Network.enum_links_vertex

Network.**enum_links_vertex**(*self*, *v0*)

Returns the arcs (links) around vertices.

Parameters

v0 [int] vertex id

Returns

links [list] List of tuple arcs adjacent to the vertex.

Examples

```
>>> import spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> ntw.enum_links_vertex(24)
[(24, 48), (24, 25), (24, 26)]
```

spaghetti.Network.full_distance_matrix

`Network.`**`full_distance_matrix`**(*self*, *n_processes*, *gen_tree=False*)

All vertex-to-vertex distances on a network. This function is called from within `allneighbordistances()`, `nearestneighbordistances()`, and `distancebandweights()`.

Parameters

n_processes [int] Cpu cores for multiprocessing.

gen_tree [bool] Rebuild shortest path `True`, or skip `False`.

Notes

Based on [Dij59].

spaghetti.Network.allneighbordistances

`Network.`**`allneighbordistances`**(*self*, *sourcepattern*, *destpattern=None*, *fill_diagonal=None*, *n_processes=None*, *gen_tree=False*, *snap_dist=False*)

Compute either all distances between `i` and `j` in a single point pattern or all distances between each `i` from a source pattern and all `j` from a destination pattern.

Parameters

sourcepattern [{str, spaghetti.network.PointPattern}] The key of a point pattern snapped to the network OR the full `spaghetti.network.PointPattern` object.

destpattern [str] (Optional) The key of a point pattern snapped to the network OR the full `spaghetti.network.PointPattern` object.

fill_diagonal [{float, int}] (Optional) Fill the diagonal of the cost matrix. Default is `None` and will populate the diagonal with `numpy.nan`. Do not declare a `destpattern` for a custom `fill_diagonal`.

n_processes [{int, str}] (Optional) Specify the number of cores to utilize. Default is 1 core. Use `int` to specify an exact number or cores. Use `"all"` to request all available cores.

gen_tree [bool] Rebuild shortest path `True`, or skip `False`.

snap_dist [bool] Flag as `True` to include the distance from the original location to the snapped location along the network. Default is `False`.

Returns

nearest [numpy.ndarray] An array of shape (n,n) storing distances between all points.

tree_nearest [dict] Nearest network node to point pattern vertex shortest path lookup. The values of the dictionary are a `tuple` of the nearest source vertex and the near destination vertex to query the lookup tree. If two observations are snapped to the same network arc a flag of -.1 is set for both the source and destination network vertex indicating the same arc is used while also raising an `IndexError` when rebuilding the path.

Examples

```
>>> import spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> ntw.snapobservations(examples.get_path('crimes.shp'),
...                                         'crimes',
...                                          attribute=True)
```

```
>>> s2s_dist = ntw.allneighbordistances('crimes')
>>> s2s_dist[0,0], s2s_dist[1,0]
(nan, 3105.189475447081)
```

```
>>> ntw.snapobservations(examples.get_path('schools.shp'),
...                                         'schools',
...                                         attribute=False)
```

```
>>> s2d_dist = ntw.allneighbordistances('crimes',
...                                     destpattern='schools')
>>> s2d_dist[0,0], s2d_dist[1,0]
(4520.72353741989, 6340.422971967316)
```

```
>>> s2d_dist, tree = ntw.allneighbordistances('schools',
...                                           gen_tree=True)
>>> tree[(6, 7)]
(173, 64)
```

spaghetti.Network.nearestneighbordistances

Network.**nearestneighbordistances**(*self*, *sourcepattern*, *destpattern=None*, *n_processes=None*, *gen_tree=False*, *all_dists=None*, *snap_dist=False*, *keep_zero_dist=True*)

Compute the interpattern nearest neighbor distances or the intrapattern nearest neighbor distances between a source pattern and a destination pattern.

Parameters

sourcepattern [str] The key of a point pattern snapped to the network.

destpattern [str] (Optional) The key of a point pattern snapped to the network.

n_processes [{int, str}] (Optional) Specify the number of cores to utilize. Default is 1 core. Use `int` to specify an exact number or cores. Use `"all"` to request all available cores.

gen_tree [bool] Rebuild shortest path `True`, or skip `False`.

all_dists [numpy.ndarray] An array of shape (n,n) storing distances between all points.

snap_dist [bool] Flag as `True` to include the distance from the original location to the snapped location along the network. Default is `False`.

keep_zero_dist [bool] Include zero values in minimum distance `True` or exclude `False`. Default is `True`. If the source pattern is the same as the destination pattern the diagonal is filled with `numpy.nan`.

Returns

nearest [dict] key is source point id, value is tuple of list containing nearest destination point ids and distance.

Examples

```
>>> import spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> ntw.snapobservations(examples.get_path('crimes.shp'),
...                      'crimes')
>>> nn = ntw.nearestneighbordistances('crimes',
...                                   keep_zero_dist=True)
>>> nn[11], nn[18]
(([18, 19], 165.33982412719126), ([19], 0.0))
```

```
>>> nn = ntw.nearestneighbordistances('crimes',
...                                   keep_zero_dist=False)
>>> nn[11], nn[18]
(([18, 19], 165.33982412719126), ([11], 165.33982412719126))
```

spaghetti.Network.split_arcs

`Network.`**`split_arcs`**(*self*, *distance*)

Split all of the arcs in the network at either a fixed distance or a fixed number of arcs.

Parameters

distance [float] The distance at which arcs are split.

Returns

split_network [spaghetti.Network] newly instantiated `spaghetti.Network` object.

Examples

```
>>> import spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> n200 = ntw.split_arcs(200.0)
>>> len(n200.arcs)
688
```

spaghetti.Network.savenetwork

`Network.`**`savenetwork`**(*self*, *filename*)

Save a network to disk as a binary file.

Parameters

filename [str] The filename where the network should be saved. This should be a full path or it will be save in the current directory.

Examples

```
>>> import spaghetti as spgh
>>> ntw = spgh.Network(examples.get_path('streets.shp'))
>>> ntw.savenetwork('mynetwork.pkl')
```

spaghetti.Network.loadnetwork

static Network.**loadnetwork**(*filename*)

Load a network from a binary file saved on disk.

Parameters

filename [str] The filename where the network should be saved.

Returns

self [spaghetti.Network] spaghetti Network object

spaghetti.Network.NetworkF

Network.**NetworkF**(*self*, *pointpattern*, *nsteps=10*, *permutations=99*, *threshold=0.2*, *distribution='uniform'*, *lowerbound=None*, *upperbound=None*)

Computes a network constrained F-Function

Parameters

pointpattern [spaghetti.network.PointPattern] A spaghetti point pattern object.

nsteps [int] The number of steps at which the count of the nearest neighbors is computed.

permutations [int] The number of permutations to perform. Default 99.

threshold [float] The level at which significance is computed. (0.5 would be 97.5% and 2.5%).

distribution [str] The distribution from which random points are sampled. Either `"uniform"` or `"poisson"`.

lowerbound [float] The lower bound at which the F-function is computed. Default 0.

upperbound [float] The upper bound at which the F-function is computed. Defaults to the maximum observed nearest neighbor distance.

Returns

NetworkF [spaghetti.analysis.NetworkF] A network F class instance.

Examples

```
>>> import spaghetti as spgh
>>> ntw = spgh.Network(in_data=examples.get_path('streets.shp'))
>>> pt_str = 'crimes'
>>> in_data = examples.get_path('{}.shp'.format(pt_str))
>>> ntw.snapobservations(in_data, pt_str, attribute=True)
```

(continues on next page)

(continued from previous page)

```
>>> crimes = ntw.pointpatterns['crimes']
>>> sim = ntw.simulate_observations(crimes.npoints)
>>> fres = ntw.NetworkF(crimes, permutations=5, nsteps=10)
>>> fres.lowerenvelope.shape[0]
10
```

spaghetti.Network.NetworkG

Network.**NetworkG**(*self*, *pointpattern*, *nsteps=10*, *permutations=99*, *threshold=0.5*, *distribution='uniform'*, *lowerbound=None*, *upperbound=None*)

Computes a network constrained G-Function

Parameters

pointpattern [spaghetti.network.PointPattern] A spaghetti point pattern object.

nsteps [int] The number of steps at which the count of the nearest neighbors is computed.

permutations [int] The number of permutations to perform. Default 99.

threshold [float] The level at which significance is computed. (0.5 would be 97.5% and 2.5%).

distribution [str] The distribution from which random points are sampled Either `"uniform"` or `"poisson"`.

lowerbound [float] The lower bound at which the G-function is computed. Default 0.

upperbound [float] The upper bound at which the G-function is computed. Defaults to the maximum observed nearest neighbor distance.

Returns

NetworkG [spaghetti.analysis.NetworkG] A network G class instance.

Examples

```
>>> import spaghetti as spgh
>>> ntw = spgh.Network(in_data=examples.get_path('streets.shp'))
>>> pt_str = 'crimes'
>>> in_data = examples.get_path('{}.shp'.format(pt_str))
>>> ntw.snapobservations(in_data, pt_str, attribute=True)
>>> crimes = ntw.pointpatterns['crimes']
>>> sim = ntw.simulate_observations(crimes.npoints)
>>> gres = ntw.NetworkG(crimes, permutations=5, nsteps=10)
>>> gres.lowerenvelope.shape[0]
10
```

spaghetti.Network.NetworkK

Network.**NetworkK**(*self*, *pointpattern*, *nsteps=10*, *permutations=99*, *threshold=0.5*, *distribution='uniform'*, *lowerbound=None*, *upperbound=None*)

Computes a network constrained K-Function

Parameters

pointpattern [spaghetti.network.PointPattern] A spaghetti point pattern object.

nsteps [int] The number of steps at which the count of the nearest neighbors is computed.

permutations [int] The number of permutations to perform. Default is 99.

threshold [float] The level at which significance is computed. (0.5 would be 97.5% and 2.5%).

distribution [str] The distribution from which random points are sampled Either `"uniform"` or `"poisson"`.

lowerbound [float] The lower bound at which the K-function is computed. Default is 0.

upperbound [float] The upper bound at which the K-function is computed. Defaults to the maximum observed nearest neighbor distance.

Returns

NetworkK [spaghetti.analysis.NetworkK] A network K class instance.

Notes

Based on [OY01].

Examples

```
>>> import spaghetti as spgh
>>> ntw = spgh.Network(in_data=examples.get_path('streets.shp'))
>>> pt_str = 'crimes'
>>> in_data = examples.get_path('{}.shp'.format(pt_str))
>>> ntw.snapobservations(in_data, pt_str, attribute=True)
>>> crimes = ntw.pointpatterns['crimes']
>>> sim = ntw.simulate_observations(crimes.npoints)
>>> kres = ntw.NetworkK(crimes, permutations=5, nsteps=10)
>>> kres.lowerenvelope.shape[0]
10
```

spaghetti.Network._evaluate_napts

`Network.`**`_evaluate_napts`**(*self*, *napts*, *unvisited*, *component_id*, *ring*)

Evaluate one connected component in a network for non-articulation points (napts) and return an updated set of napts and unvisted vertices.

Parameters

napts [set] Non-articulation points (napts) in the network. The 'napts' here do not include those within an isolated loop island.

unvisited [set] Vertices left to evaluate in the network.

component_id [int] ID for the network connected component for the current iteration of the algorithm.

ring [bool] Network component is isolated island loop `True` or not `False`.

Returns

napts [set] Updated 'napts' object.

unvisited [set] Updated 'napts' object.

spaghetti.Network._extractnetwork

Network.**_extractnetwork**(*self*)
Used internally to extract a network from a polyline shapefile of a `geopandas.GeoDataFrame`.

spaghetti.Network._newpoint_coords

Network.**_newpoint_coords**(*self*, *arc*, *distance*)
Used internally to compute new point coordinates during snapping.

spaghetti.Network._round_sig

Network.**_round_sig**(*self*, *v*)
Used internally to round the vertex to a set number of significant digits. If `sig` is set to 4, then the following are some possible results for a coordinate are as follows. (1) 0.0xxxx, (2) 0.xxxx, (3) x.xxx, (4) xx.xx, (5) xxx.x, (6) xxxx.0, (7) xxxx0.0

Parameters

v [tuple] X,Y coordinate of the vertex

spaghetti.Network._snap_to_link

Network.**_snap_to_link**(*self*, *pointpattern*)
Used internally to snap point observations to network arcs.

Parameters

pointpattern [spaghetti.network.PointPattern] point pattern object

Returns

obs_to_arc [dict] Dictionary with arcs as keys and lists of points as values.

arc_to_obs [dict] Dictionary with point ids as keys and arc tuples as values.

dist_to_vertex [dict] Dictionary with point ids as keys and values as dicts with keys for vertex ids and values as distances from point to vertex.

dist_snapped [dict] Dictionary with point ids as keys and distance from point to the network arc which it is snapped.

spaghetti.Network._yield_napts

Network.**_yield_napts**(*self*)
Find all nodes with degree 2 that are not in an isolated island ring (loop) component. These are non-articulation points on the graph representation.

Returns

napts [list] non-articulation points on a graph representation

spaghetti.Network._yieldneighbor

`Network._yieldneighbor`(*self*, *vtx*, *arc_vertices*, *bridge*)

Used internally, this method traverses a bridge arc to find the source and destination nodes.

Parameters

vtx [int] vertex id

arc_vertices [list] All non-articulation points in the network. These are referred to as degree-2 vertices.

bridge [list] Inital bridge list containing only `vtx`.

Returns

nodes [list] Vertices to keep (articulation points). These elements are referred to as nodes.

2.2.2 spaghetti.PointPattern

spaghetti.PointPattern([in_data, ...])	A stub point pattern class used to store a point pattern.

spaghetti.PointPattern

class spaghetti.**PointPattern**(*in_data=None*, *idvariable=None*, *attribute=False*)

A stub point pattern class used to store a point pattern. This class is monkey patched with network specific attributes when the points are snapped to a network. In the future this class may be replaced with a generic point pattern class.

Parameters

in_data [{geopandas.GeoDataFrame, str}] The input geographic data. Either (1) a path to a shapefile `str`; or (2) a geopandas.GeoDataFrame.

idvariable [str] Field in the shapefile to use as an id variable.

attribute [bool] A flag to indicate whether all attributes are tagged to this class (`True`) or excluded (`False`). Default is `False`.

Attributes

points [dict] Keys are the point ids (int). Values are the x,y coordinates (tuple).

npoints [int] The number of points.

obs_to_arc [dict] Keys are arc ids (tuple). Values are snapped point information (`dict`). Within the snapped point information (`dict`) keys are observation ids (`int`), and values are snapped coordinates.

obs_to_vertex [list] List of incident network vertices to snapped observation points converted from a `default_dict`. Originally in the form of paired left/right nearest network vertices {netvtx1: obs_id1, netvtx2: obs_id1, netvtx1: obs_id2... netvtx1: obs_idn}, then simplified to a list in the form [netvtx1, netvtx2, netvtx1, netvtx2, ...].

dist_to_vertex [dict] Keys are observations ids (`int`). Values are distance lookup (`dict`). Within distance lookup (`dict`) keys are the two incident vertices of the arc and values are distance to each of those arcs.

snapped_coordinates [dict] Keys are the point ids (`int`). Values are the snapped x,y coordinates (tuple).

snap_dist [bool] Flag as `True` to include the distance from the original location to the snapped location along the network. Default is `False`.

__init__(*self*, *in_data=None*, *idvariable=None*, *attribute=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(self[, in_data, idvariable, attribute])	Initialize self.

2.2.3 spaghetti.SimulatedPointPattern

spaghetti.SimulatedPointPattern()	Struct style class to mirror the `PointPattern` class.

spaghetti.SimulatedPointPattern

class spaghetti.**SimulatedPointPattern**
Struct style class to mirror the `PointPattern` class. If the `PointPattern` class has methods, it might make sense to make this a child of that class. This class is not intended to be used by the external user.

Attributes

npoints [int] The number of points.

obs_to_arc [dict] Keys are arc ids (tuple). Values are snapped point information (`dict`). Within the snapped point information (`dict`) keys are observation ids (`int`), and values are snapped coordinates.

obs_to_vertex [list] List of incident network vertices to snapped observation points converted from a `default_dict`. Originally in the form of paired left/right nearest network vertices {netvtx1: obs_id1, netvtx2: obs_id1, netvtx1: obs_id2... netvtx1: obs_idn}, then simplified to a list in the form [netvtx1, netvtx2, netvtx1, netvtx2, ...].

dist_to_vertex [dict] Keys are observations ids (`int`). Values are distance lookup (`dict`). Within distance lookup (`dict`) keys are the two incident vertices of the arc and values are distance to each of those arcs.

snapped_coordinates [dict] Keys are the point ids (`int`). Values are the snapped x,y coordinates (tuple).

snap_dist [bool] Flag as `True` to include the distance from the original location to the snapped location along the network. Default is `False`.

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

Methods

__init__(self)	Initialize self.

2.2.4 spaghetti

spaghetti.element_as_gdf(net[, vertices, ...])	Return a geopandas.GeoDataFrame of network elements.This can be (a) the vertices of a network; (b) the arcs of a network; (c) both the vertices and arcs of the network; (d) raw point pattern associated with the network; or (e) snapped point pattern of (d)..

spaghetti.element_as_gdf

spaghetti.**element_as_gdf**(*net*, *vertices=False*, *arcs=False*, *pp_name=None*, *snapped=False*, *id_col='id'*, *geom_col='geometry'*)

Return a geopandas.GeoDataFrame of network elements. This can be (a) the vertices of a network; (b) the arcs of a network; (c) both the vertices and arcs of the network; (d) raw point pattern associated with the network; or (e) snapped point pattern of (d).

Parameters

net [spaghetti.Network] network object

vertices [bool] Extract the network vertices. Default is `False`.

arcs [bool] Extract the network arcs. Default is `False`.

pp_name [str] Name of the network `PointPattern` to extract. Default is `None`.

snapped [bool] If extracting a network `PointPattern`, set to `True` for snapped point locations along the network. Default is `False`.

id_col [str] GeoDataFrame column name for IDs. Default is `'id'`.

geom_col [str] GeoDataFrame column name for geometry. Default is `'geometry'`.

Returns

points [geopandas.GeoDataFrame] Network point elements (either vertices or `PointPattern` points) as a *geopandas.GeoDataFrame* of `shapely.Point` objects with an `id` column and `geometry` column.

lines [geopandas.GeoDataFrame] Network arc elements as a `geopandas.GeoDataFrame` of `shapely.LineString` objects with an `id` column and `geometry` column.

Raises

KeyError In order to extract a `PointPattern` it must already be a part of the `spaghetti.Network` object. This exception is raised when a `PointPattern` is being extracted that does not exist within the `spaghetti.Network` object.

Notes

This function requires geopandas.

2.3 References

BIBLIOGRAPHY

[Dij59] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959. doi:10.1007/BF01386390.

[OY01] Atsuyuki Okabe and Ikuho Yamada. The K-Function Method on a Network and Its Computational Implementation. *Geographical Analysis*, 33(3):271–290, 2001.

INDEX

Symbols

A

C

D

E

F

L

N

P

S