# GeoNetworkX Documentation

***Release 0.5.0***

**Artelys - HC**

**May 25, 2020**

# CONTENTS:

GeoNetworkX is a project to handle geospatial graphs. GeoNetworkX extends the NetworkX package to allow spatial operations on geospatial graphs and benefit from the data structures and algorithm defined in NetworkX. Moreover, it allows to use GeoPandas library tools on nodes and edges.

CHAPTER

ONE

# DESCRIPTION

The goal of GeoNetworkX is to embed a set of tools to handle geospatial graphs easily. It combines the capabilities of networkx, geopandas and shapely, providing geospatial operations in networkx high-level interface.

GeoNetworkX provides data structures that extends the networkx classes with this inheritance scheme:

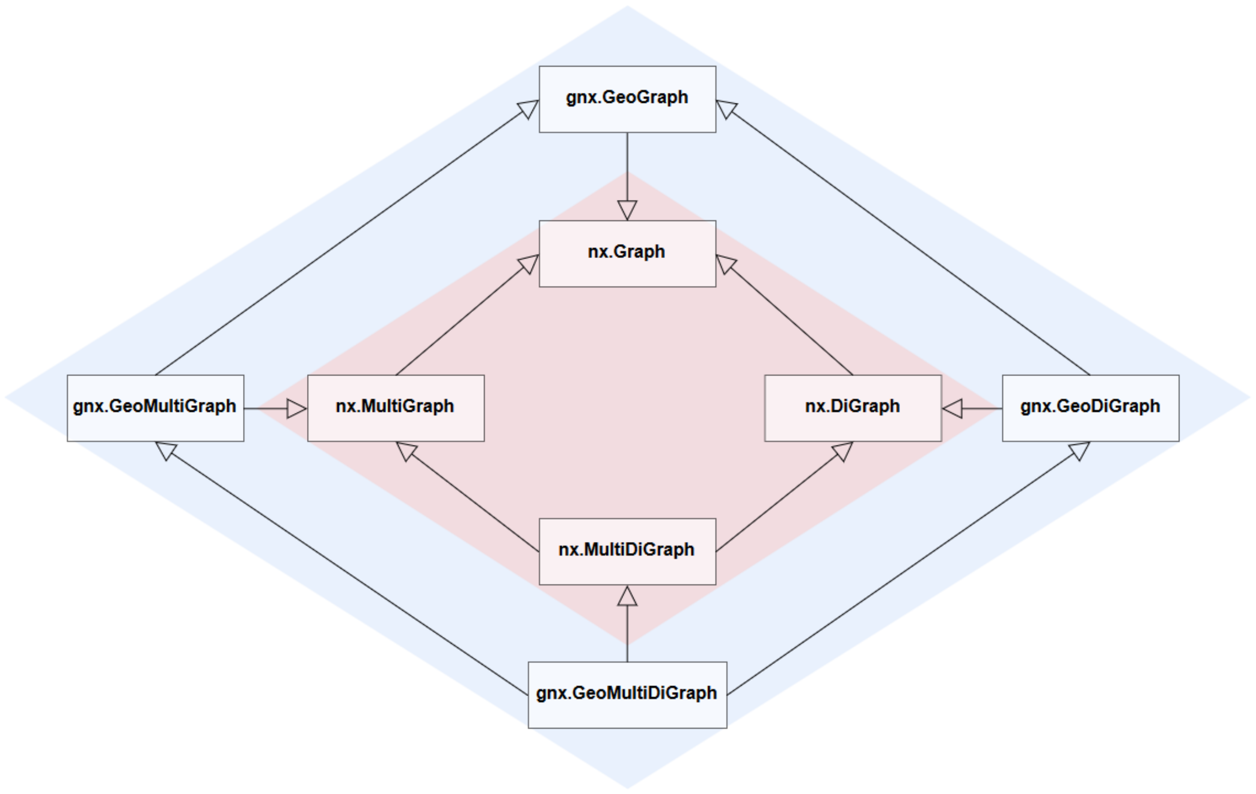

Fig. 1: GeoNetworkX inheritance graph

## 1.1 Getting started

### 1.1.1 Installation

GeoNetworkX can be installed with pip with the following command:

```
pip install geonetworkx
```

**Warning:** GeoNetworkX needs packages that have C dependencies that may need to be compiled and installed manually (*shapely*, *fiona*, *pyproj* and *rtree*). For Windows users, wheels can be found at Christopher Gohlke's website.

### 1.1.2 What's a GeoGraph ?

A geograph is an object extending classical graph definition with topological space. For example, it can be a road network where edges represent streets and nodes represent their intersection. Other applications can be found in electrical networks, railway networks, etc. Mathematically, a geograph is defined with the following elements:

- A topological space $S$ with a distance measurement application $d : S \times S \to \mathbb{R}^+$
- A graph $G(N, E)$ with $N$ a finite set of vertices and $E \subset N^2$ at set of pairs of vertices.
- $P := \bigcup_{n \in N} p_n$ with $p_n \in S$ the coordinates of the node $n$.
- $L := \bigcup_{(u,v) \in E} l_{u,v}$ with $l_{u,v} \subset S$ a topological curve starting at $p_u$ and ending at $p_v$.

The space $S$ is usually here considered to be $\mathbb{R}^2$ with the euclidian distance, or the WGS84 spheroid with the great-circle distance (or Vincenty distance).

### 1.1.3 Closest edge rule

The implementation uses the closest edge rule to connect a topological point $p \in S$ to a geograph. This rule define a connection point $i_p$ :

$$i_p := \text{proj}_L(p) = \text{argmin}\{d(p, x) | x \in L\}$$

This rule allows to connect any point of the topological space to the geograph. In the street network example, it means finding the closest street for starting a trip.

### 1.1.4 Implementation details

The implementation of geographs in GeoNetworkX is based on the following hypothesis:

- All nodes have coordinates stored in a `shapely.geometry.Point` object.
- Edges may have geometry stored in a `shapely.geometry.LineString` object.
- A geograph may have a Coordinate Reference System (CRS) using GeoPandas implementation.

An edge may not have a geometry but it is supposed that it can be deduced by a simple "straight" line between the two nodes.

## 1.2 Reading and Writing Files

### 1.2.1 Reading Spatial Data

As GeoNetworkX provides an interface to *geopandas* for the nodes and edges, it is possible to read data from any vector-based spatial data supported by *geopandas* (including ESRI Shapefile and GeoJSON).

Nodes and edges can be added to a given graph with the following methods:

```
import geonetworkx as gnx
# Adding nodes and edges to an existing graph
g = gnx.GeoGraph()
g.add_nodes_from_gdf("copenhagen_streets_net_nodes.geojson")
g.add_edges_from_gdf("copenhagen_streets_net_edges.geojson")
gnx.read_geofiles(nodes_path, edges_path, directed=True, multigraph=True)

# Creating a graph from existing files
g = gnx.read_geofiles("copenhagen_streets_net_nodes.geojson",
                      "copenhagen_streets_net_edges.geojson",
                      edges_path, directed=True, multigraph=True)
```

### 1.2.2 Writing Spatial Data

Geographs can be exported to same file formats as *geopandas*. Two files are used to write a GeoGraph: one for nodes and one for edges. All the attributes of the nodes and of the edges will be added in the files. If an attribute type is not handled by *fiona* drivers, an attempt is made to cast it (see *gnx.write_geofile* for more details).

```
g.name = "streets_graph"
gnx.write_geofile(g, "test/path/", driver="GeoJSON")
```

The above code will write two GeoJSON files: `test/path/streets_graph_nodes.geojson` and `test/path/streets_graph_edges.geojson` that can be directly read with GIS software.

## 1.3 Supplement data

### 1.3.1 Computing distances

GeoNetworkX provides methods to compute distances within the coordinate reference system of the graph. Typically, a method is given to add a length attribute on edges. Different methods are available: euclidian distance but also geodesic, great-circle or Vincenty distance (wrapped from *geopy*). All available distances are stored within the dictionary *DISTANCE_MEASUREMENT_METHODS*.

```
>>> import geonetworkx as gnx
>>> g = gnx.GeoGraph(crs=gnx.WGS84_CRS)
>>> g.add_edge(1, 2, geometry=gnx.LineString([(-73.614, 45.504), (-73.632, 45.506)]))
>>> gnx.fill_length_attribute(g)  # using geodesic distance
>>> print(g.edges[(1, 2)]["length"])
1424.174413518016
>>> g.to_utm(inplace=True)
>>> gnx.fill_length_attribute(g, only_missing=False)
>>> print(g.edges[(1, 2)]["length"])  # using euclidian distance in UTM
1423.8073619096585
```

A custom distance measurement method can be used by defining the appropriate method in the settings. Here is an example implementing the Manhattan distance:

```
>>> def manhattan(p1, p2):
...     return abs(p1.x - p2.x) + abs(p1.y - p2.y)
>>> gnx.settings.DISTANCE_MEASUREMENT_METHODS["manhattan"] = manhattan
>>> gnx.fill_length_attribute(g, only_missing=False, method="manhattan")  # using␣
↪manhattan distance
>>> print(g.edges[(1, 2)]["length"])
1608.0440213837428
```

### 1.3.2 Getting elevation data

The elevation of nodes points can be filled as an attribute through the SRTM package.

```
>>> import geonetworkx as gnx
>>> g = gnx.GeoGraph(crs=gnx.WGS84_CRS)
>>> g.add_node(1, gnx.Point(5.145, 45.213))
>>> gnx.fill_elevation_attribute(g)
>>> print(g.nodes[1]["elevation[m]"])
473
```

## 1.4 Spatial merge tools

GeoNetworkX implements methods for map-matching points to the geograph. That is to say finding, for a query point, the closest edge or node in a geograph. Mathematically, it means solving the following optimization problem for a query point $p \in S$:

$$\min_{x \in L} d(p, x)$$

A frequently encountered problem is finding the closest edge in a geograph for a set of points $P$. If done naively, a nested loop on points and edges is performed to compute all distances. This introduces a high computational cost ($o(|P| \times |E|)$) that can be avoided by using the right data structure. GeoNetworkX uses kd-trees to efficiently solve this problem. Theses allow to find the optimal solution without having to compute all distances ($o((|P| + |E|) \log |E|)$).

To compute the closest edge, all edge geometries are discretized within a tolerance distance $\epsilon > 0$. This method is not exact if the coordinate of the geograph are not unprojected (using latitude and longitude angles), but produces fairly good results. The implemented method uses kd-trees implemented in the Scipy Spatial package (see cKDTree).

### 1.4.1 Spatial points merge

GeoNetworkX implements a method to add a set of point to a geograph as nodes using the closest edge rule (`gnx.spatial_points_merge`). This method not only find the closest edge but generate the new edges to connect the new nodes to the geograph:

Here is an example that merge a set of bicycle station to a street network.

```
import geopandas as gpd
import geonetworkx as gnx
import osmnx as ox
```

(continues on next page)

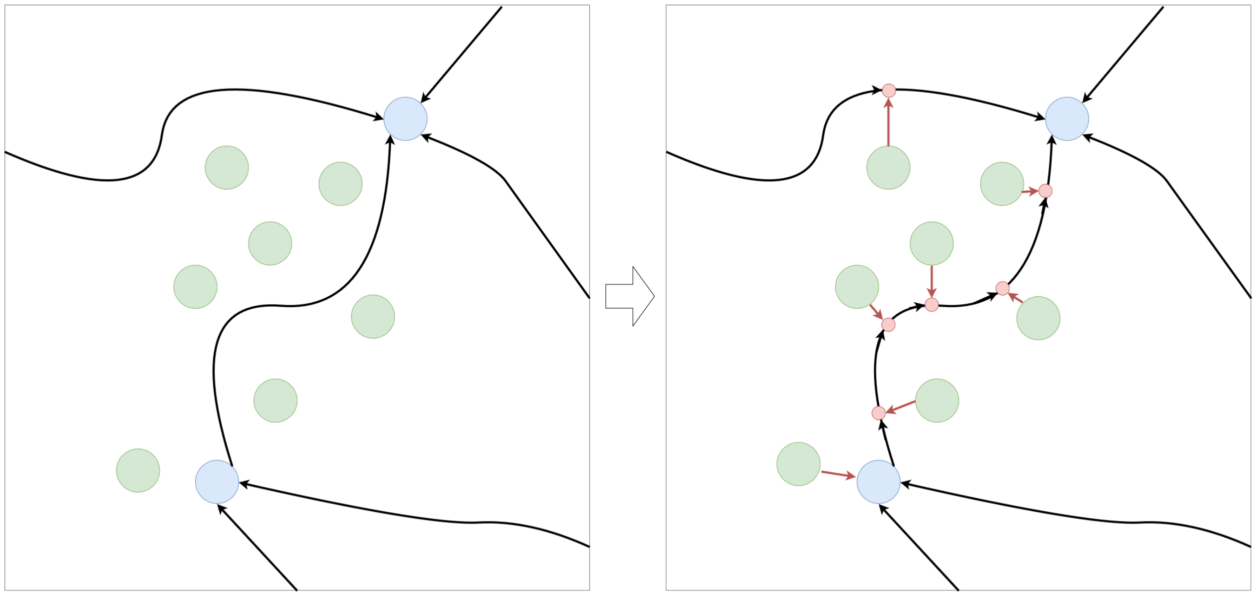

Fig. 2: Illustration of work done in the `spatial_points_merge` method. Initial geograph edges are in black, nodes in blue , new nodes are in green, new intersection nodes in red.

(continued from previous page)

```
# Download and set up the street network (main streets_graph only)
streets_graph = ox.graph_from_address("Rennes, France", distance=2500,
                                      infrastructure='way["highway"~
↪"primary|secondary|tertiary"]')
streets_graph = gnx.read_geograph_with_coordinates_attributes(streets_graph)

# Getting the bicycle stations
bicycle_stations = gpd.read_file("geonetworkx/tests/datasets/"
                                 "rennes_bicycle_stations_velo_star.geojson")

# Merging the stations to the street network
gnx.spatial_points_merge(streets_graph, bicycle_stations, inplace=True)
```

### 1.4.2 Spatial graph merge

An additional useful feature that provides GeoNetworkX is geographs merge. That is to say, from a base graph, adding another graph on top of it and setting the right edges connect both. This feature may be very useful for multimodal transport routing. For example, a use case is to merge a street graph with a subway system graph to find an optimal route combining walk and subway transportation. To do so, the closest street of each subway station has to be found and an edge has to be added to link them. This is what is done in the `gnx.spatial_graph_merge` method.

Practically, this can be useful for merging two independent networks by specifying connecting nodes from one graph and reachable edges from the other graph. For instance, it can be used to build a multi-modal network combining streets network and subway network by connecting subway stations (represented as nodes) to their closest street (represented as edges).

```
# streets: GeoMultiDiGraph
# subway: GeoMultiDiGraph
```

(continues on next page)

(continued from previous page)

```
subway_node_is_station = lambda n: subway.nodes[n].get("name", None) is None
streets_and_subway = gnx.spatial_graph_merge(streets,
                                             subway,
                                             node_filter=subway_node_is_station)
```

Here is a visualization of the result:

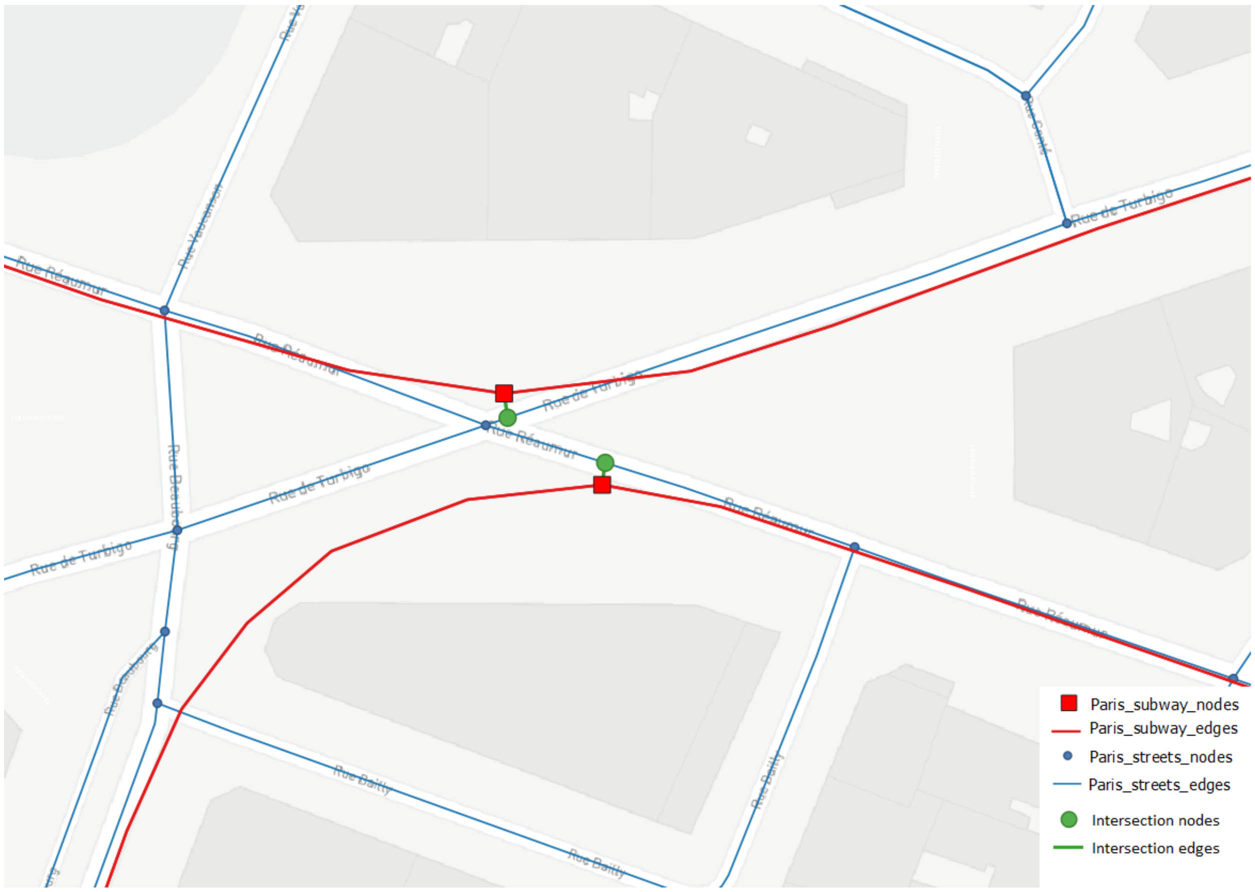

Fig. 3: Illustration of work done in the `spatial_graph_merge` method. Street network is in blue, subway network in red, and merging elements in green.

## 1.5 Isochrones

Isochrones defines the reach of a location within a distance limit. An isochrone polygon is defined by all reachable points from a source node trough a given geograph.

Using an additional distance function $d_G : N \times N \rightarrow \mathbb{R}^+$ corresponding to a shortest path distance between two nodes in the graph, the isochrone polygon with source node $n$ and a distance $\epsilon$ can be defined as:

$$I_n^\epsilon := \{x \in S : d_G(n, i_x) + d(i_x, x) \leq \epsilon\}$$

with $i_x := \text{proj}_L(x)$

The core method is based on Shortest Path Tree generation (SPT) (or ego-graph as in NetworkX). This tree contains all nodes reachable within the $\epsilon$ distance from the source node. To get an isochrone polygon approximation, this tree has to be "buffered" to represent the boundaries of the SPT. Such polygons can be approximate by various methods (see Isochrones OSM wiki). Two methods have been implemented in GeoNetworkX:

- $\alpha$-shapes for fast approximation
- Natural neighborhood with edges Voronoi cells computation for a precise approximation

The $\alpha$-shape method produces most of the time good results at city scale. In contrast, edges Voronoi cells produce a more faithful representation of isochrone polygons that can be interpreted at street scale. Here is an example foreach method:

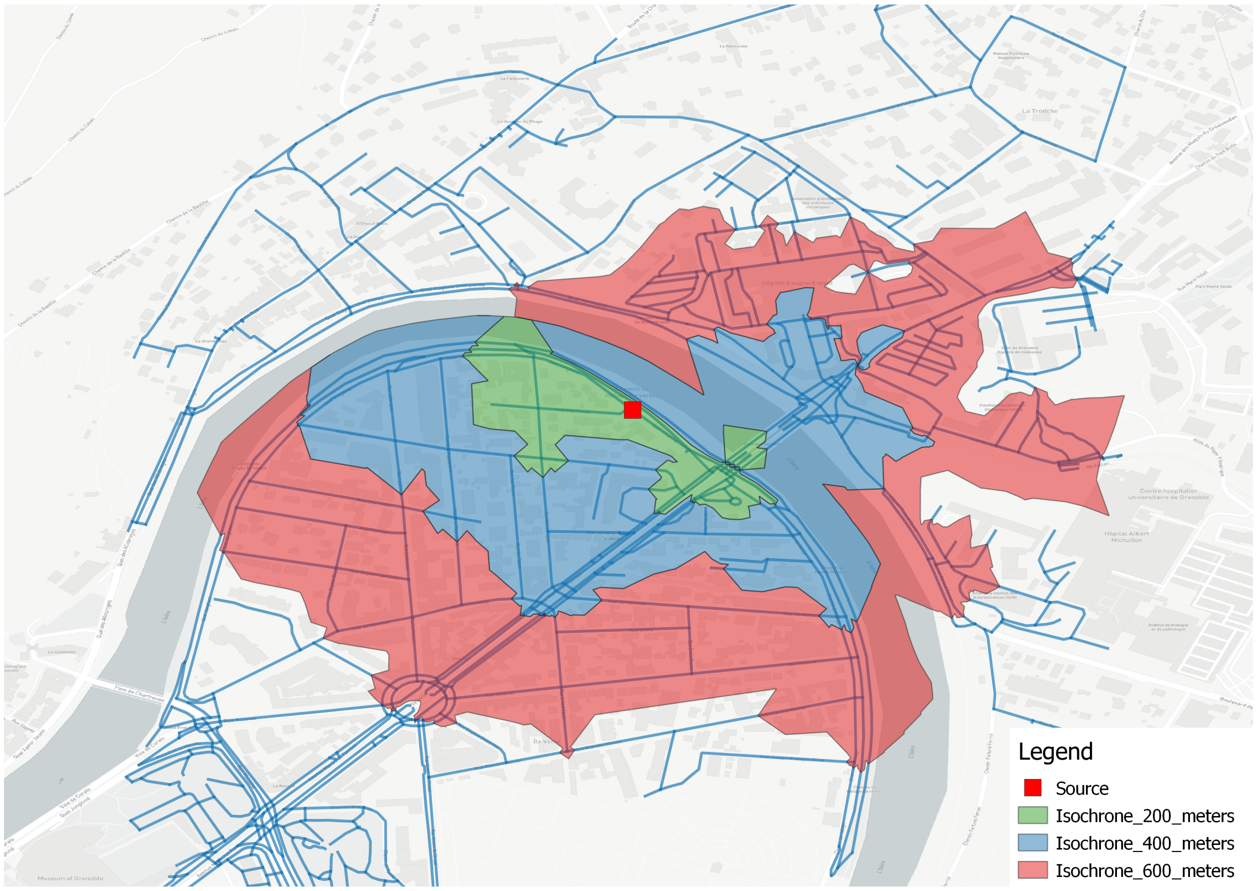

Fig. 4: Isochrone polygon example using natural neighborhood (Grenoble, France)

Details about each method are provided below.

### 1.5.1 Shortest Path Tree

To compute an isochrone, GeoNetworkX uses shortest path tree computation that is implemented in NetworkX `ego_graph` method. An extended version is proposed to compute precisely the boundaries of a graph by using spatial information. The proposed algorithm adds boundary nodes on edges leaving the ego-graph to represent the exact point where the cutoff value is reach in the SPT. The coordinates of the boundary nodes are computed using a linear interpolation with edge geometry length (see `geonetworkx.generators.extended_ego_graph` for more details). Here is an illustration of an extended shortest path tree for a street network:

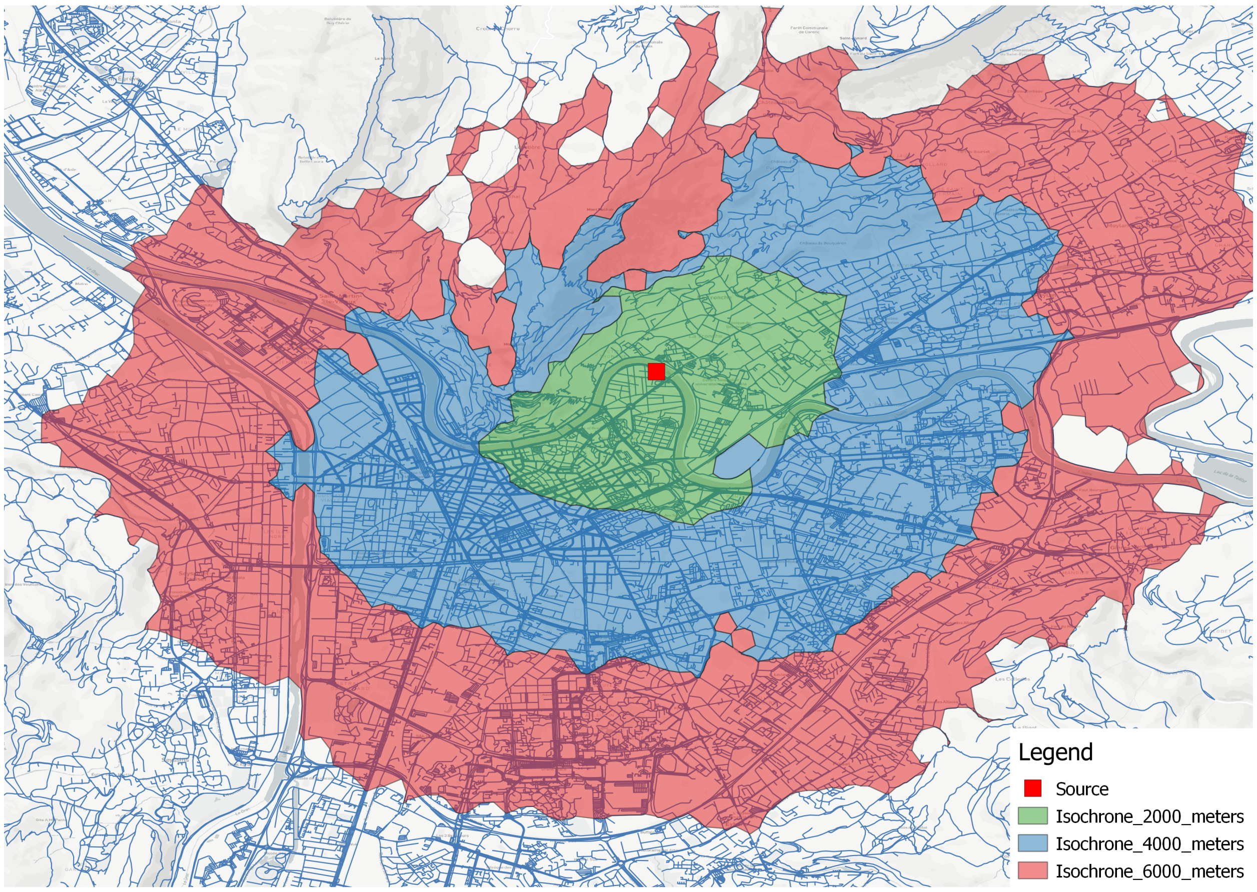

Fig. 5: Isochrone polygon example using alpha-shape (Grenoble, France)

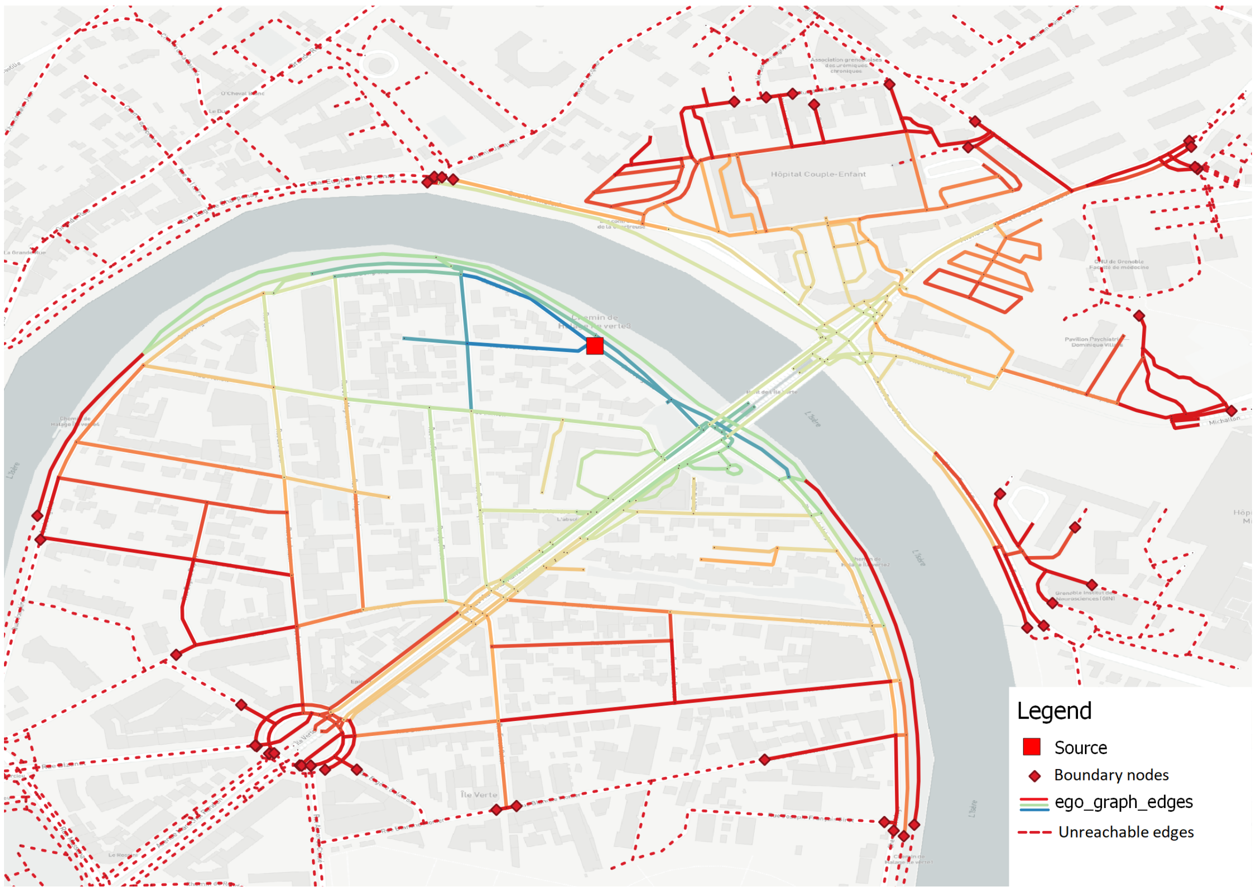

Fig. 6: Ego-graph (or SPT) example from a source node and 600 meters limit. Edges colors represent total length to the source.

### 1.5.2 Edges Voronoi cells

For a given graph, this method computes voronoi cells of each edge. That is to say for each edge, all points closer to this edge than any other edge. Formally, the voronoi cell $V_e$ of the edge $e$ is:

$$V_e := \{x \in S : d(x, \mathrm{proj}_{l_e}(x)) \leq d(x, \mathrm{proj}_{l_u}(x)), \forall u \in E \backslash \{e\}\}$$

From a shortest path tree $T \subset G$, we define its influence polygon $V_T$ by all points closer to any edge of $T$ than any edge of $G$ that is not in $T$. That is to say:

$$V_T := \bigcup_{e \in E_T} V_e$$

Edges Voronoi cells are computed in GeoNetworkX thanks to the PyVoronoi package that provides an interface to the boost Voronoi library. This code allows to compute cells for points and disjoint segments. A work has been done in GeoNetworkX to generalize this work to generic *linestrings*. For instance, here is an example of the edges voronoi cells of a street graph:

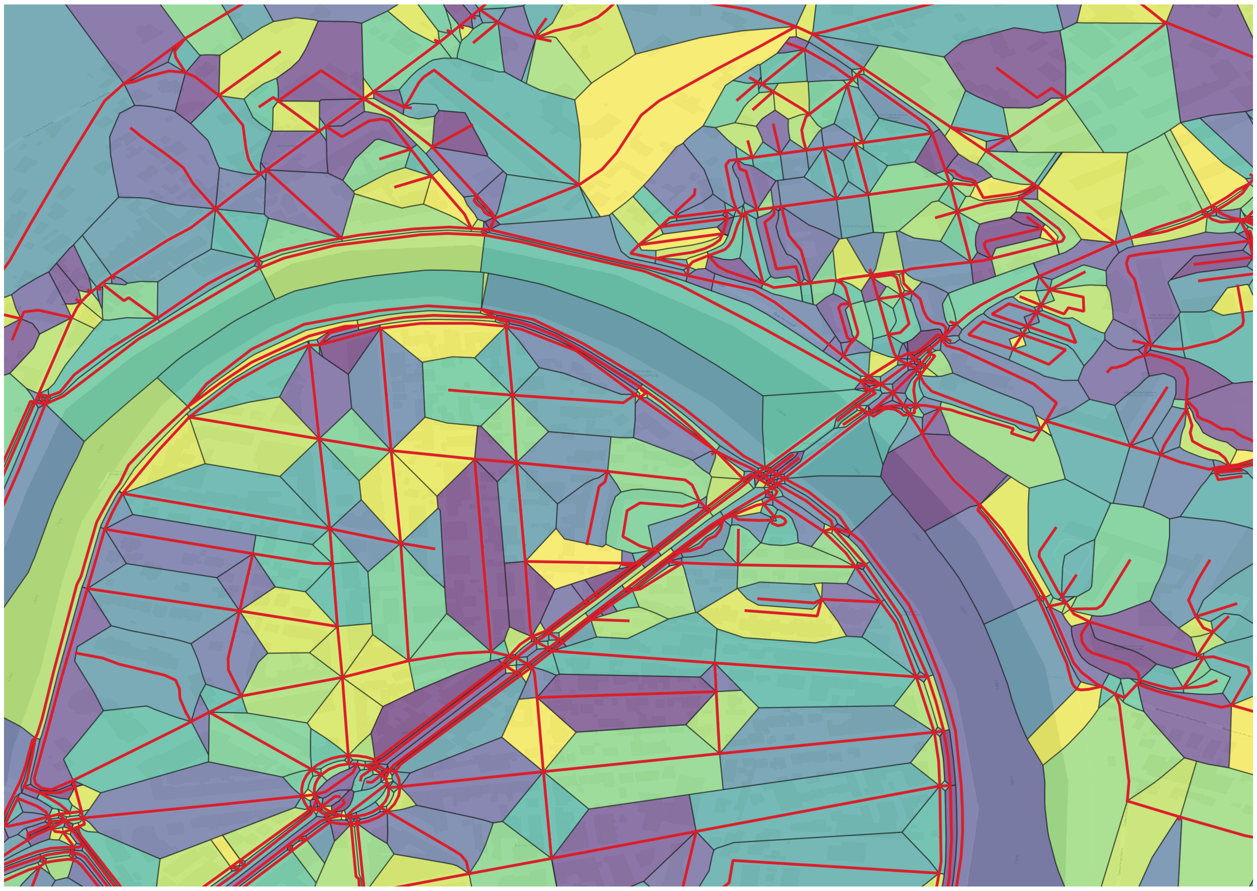

Fig. 7: Edges Voronoi cells of a street graph.

### 1.5.3 $\alpha$-shape

$\alpha$-shapes are a generalization of the concept of convex hull. For a finite set of points, it defines a piecewise linear curve that forms a polygon that contains all points. It can be computed with the Delaunay triangulation of the point set $P$ and with the circumradius of the triangles:

$$A_\alpha := \bigcup_{t \in \text{Delaunay}(P) : r(t) \leq 1/\alpha} t$$

With $r(t)$ the circumradius of the triangle $t$.

Using this definition, $A_0$ represents the convex hull of $P$ and $A_{\text{inf}}$ is the empty set (it can be seen has the minimum spanning tree of the points).

Theses shapes can be used to approximate an isochrone polygon to "buffer" the SPT geometry. To do this, GeoNetworkx compute an $\alpha$-shape on a discretized edges of the SPT.

Here is an example of the Delaunay triangulation used to compute an $\alpha$-shape.

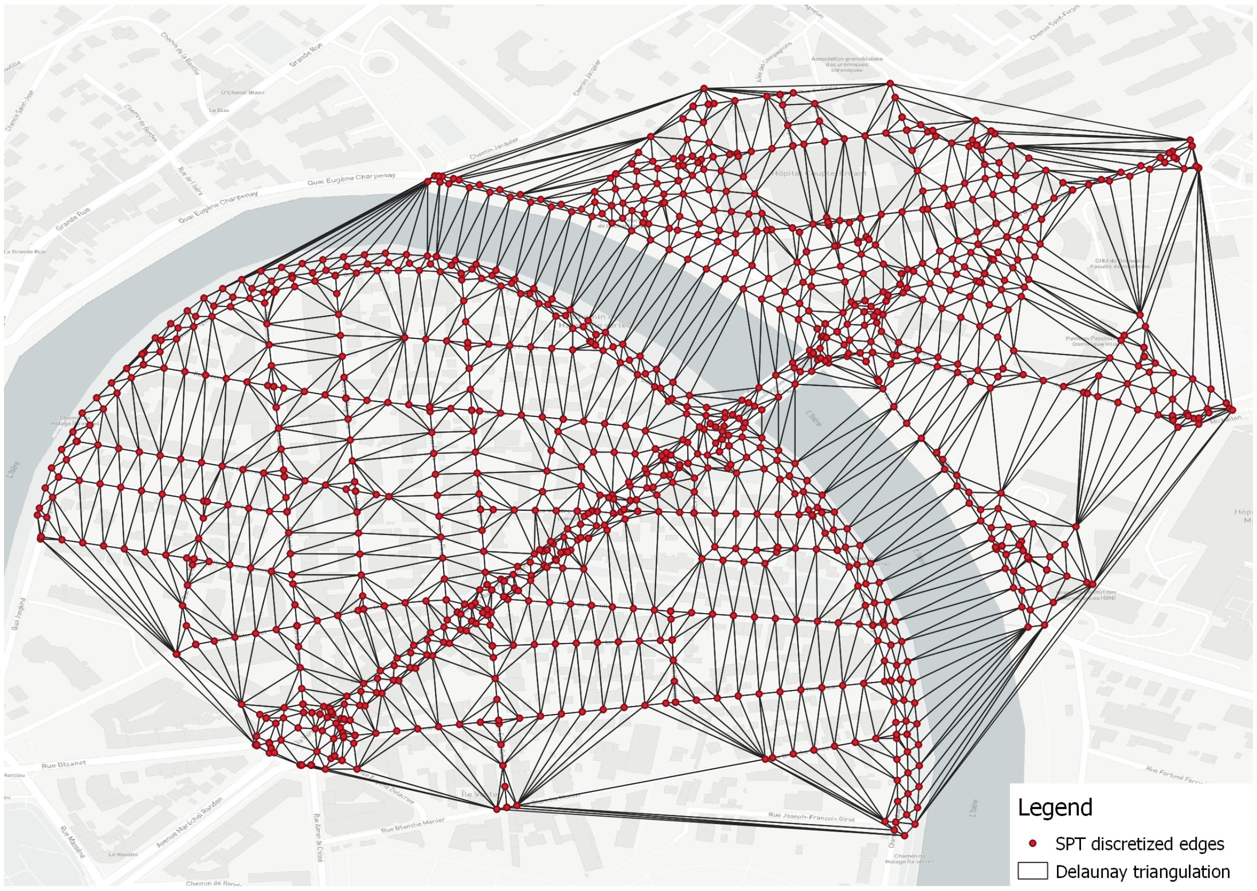

Fig. 8: Delaunay triangulation of a discretization of SPT edges.

The parameter $\alpha$ is computed by taking a percentile of the circumradius of all triangles. For the same example as above, here is the distribution of circumradius of triangles.

A "good" choice for $1/\alpha$ is the 99-percentile of the circumradius to exclude only outliers.

## 1.6 Reference

**Release** 0.5.0

**Date** May 25, 2020

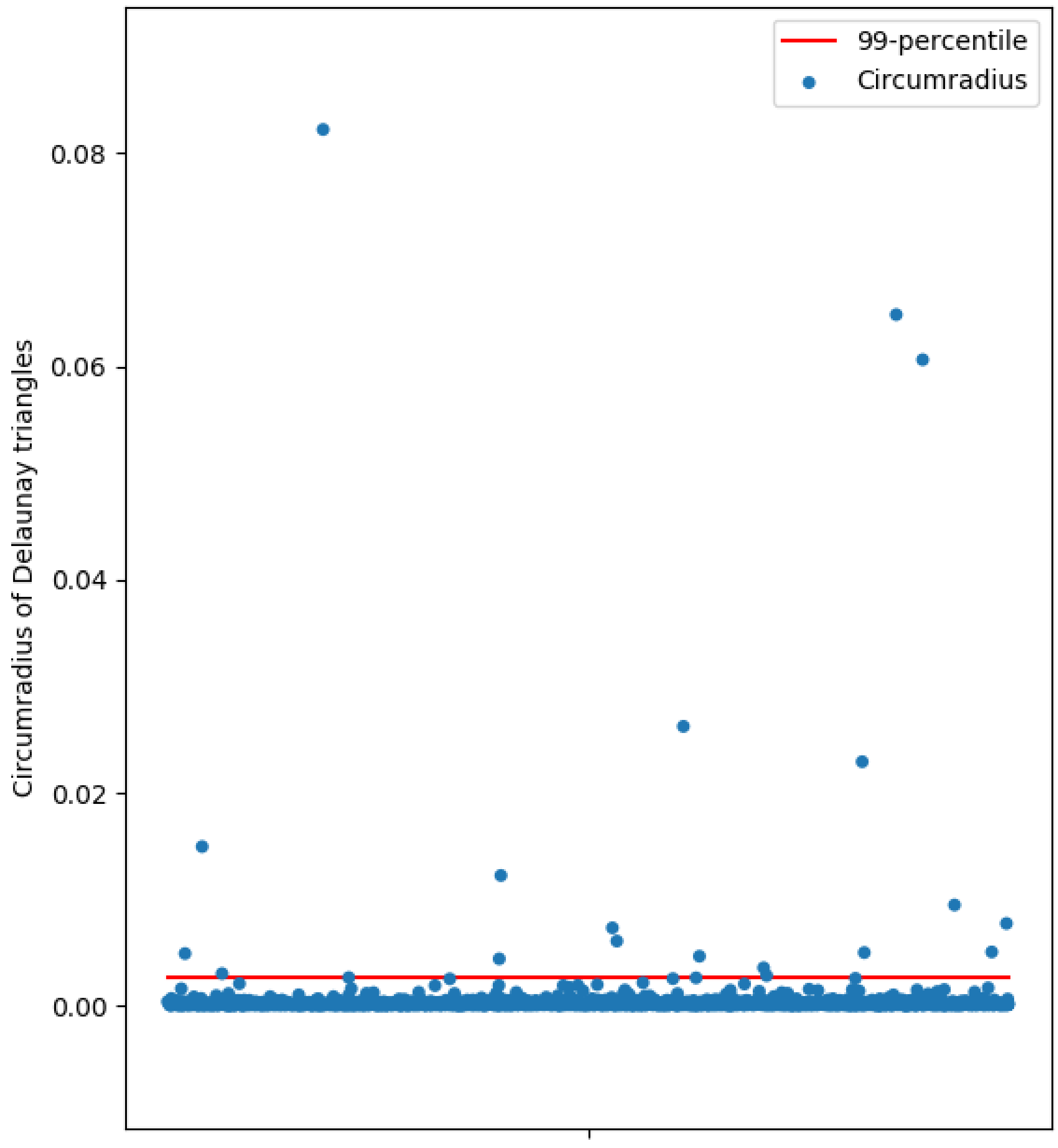

Fig. 9: Scatter plot of the circumradius of all triangles.

### 1.6.1 Main classes

#### GeoGraph

**class GeoGraph**(*incoming_graph_data=None*, ***attr*)

Bases: `networkx.classes.graph.Graph`

This class extends the `networkx.Graph` to represent a graph that have a geographical meaning. Nodes are located with their coordinates (x, y) (using `shapely.geometry.Point` objects) and edges can be represented with a given broken line (using `shapely.geometry.LineString` objects). Each graph has its own keys for naming nodes and edges geometry (`nodes_geometry_key`, `edges_geometry_key`). A coordinate reference system (CRS) can be defined for a graph and will be used for some methods managing earth coordinates (especially for distances). All nodes must have defined coordinates, otherwise a default coordinates are used.

**Raises** `ValueError` – If the all nodes don't have valid coordinates.

**See also:**

`networkx.Graph`, `GeoDiGraph`, `GeoMultiGraph`, `GeoMultiDiGraph`

Initialize a graph with edges, name, or graph attributes.

**Parameters**

- **incoming_graph_data** (*input graph (optional, default: None)*) – Data to initialize graph. If None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Attributes to add to graph as key=value pairs.

**See also:**

`convert`

#### Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1, 2), (2, 3), (3, 4)]  # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

```
>>> G = nx.Graph(e, day="Friday")
>>> G.graph
{'day': 'Friday'}
```

**_get_nodes_geometries_from_edge_geometry**(*u*, *v*, *geometry*)

For each node of the edge, return the node geometry deduced from the linestring if it not already present.

**_get_nodes_geometries_to_set_for_edges_adding**(*ebunch_to_add*, *attr*)

Return a dictionary of nodes geometries to set when adding a set of edges.

**add_edge**(*u_of_edge*, *v_of_edge*, ***attr*)

Add an edge between u and v.

If one of the node is not already in the graph and a geometry is provided, the node geometry is deduced from the first or last point of the linestring.

### Examples

```
>>> import geonetworkx as gnx
>>> g = gnx.GeoGraph()
>>> g.add_edge(1, 2, geometry=gnx.LineString([(0, 0), (1, 1)]))
>>> print(g.nodes[2]["geometry"])
POINT (1 1)
```

**add_edges_from**(*ebunch_to_add*, ***attr*)
Add all the edges in ebunch_to_add and add nodes geometry if they are not present.

If one of the node is not already in the graph and a geometry is provided, the node geometry is deduced from the first or last point of the linestring.

### Examples

```
>>> import geonetworkx as gnx
>>> g = gnx.GeoGraph()
>>> g.add_edges_from([(0, 1, dict(geometry=gnx.LineString([(0, 0), (1, 1)]))),
...                   (1, 2, dict(geometry=gnx.LineString([(1, 1), (2, 2)])))])
>>> print(g.nodes[2]["geometry"])
POINT (2 2)
```

```
>>> g = gnx.GeoMultiGraph()
>>> g.add_edges_from([(0, 1, 7, dict(geometry=gnx.LineString([(-1, 0), (1,␣
↪1)]))),
...                   (1, 2, 8, dict(geometry=gnx.LineString([(1, 1), (2,␣
↪2)])))])
[7, 8]
>>> print(g.nodes[1]["geometry"])
POINT (1 1)
```

**See also:**

*add_edge*, `nx.Graph.add_edges_from`

**add_edges_from_gdf**(*gdf*, *edge_first_node_attr=None*, *edge_second_node_attr=None*)
Add edges with the given *GeoDataFrame*. If no dataframe columns are specified for first and second node, the dataframe index must be a multi-index *(u, v)*.

**Parameters**

- **gdf** – GeoDataFrame representing edges to add (one row for one edge).
- **edge_first_node_attr** – Edge first node attribute. If `None`, the dataframe index is used, else the given column is used. Must be used with `edge_second_node_attr`. (Default value = None)
- **edge_second_node_attr** – Edge second node attribute. If `None`, the dataframe index is used, else the given column is used. Must be used with `edge_first_node_attr`. (Default value = None)

**See also:**

*add_nodes_from_gdf*

**add_node** (*node_for_adding*, *geometry=None*, ***attr*)

Add a single node *node_for_adding* with its given geometry.

**See also:**

`nx.Graph.add_node`

### Examples

```
>>> import geonetworkx as gnx
>>> g = gnx.GeoGraph()
>>> g.add_node(1, gnx.Point(2, 3))
>>> print(g.nodes[1]["geometry"])
POINT (2 3)
```

**add_nodes_from** (*nodes_for_adding*, ***attr*)

Add multiple nodes with potentially given geometries.

If no geometry is provided, behaviour is same as the `nx.Graph.add_nodes_from` method.

**See also:**

`nx.Graph.add_nodes_from`

### Examples

```
>>> import geonetworkx as gnx
>>> g = gnx.GeoGraph()
>>> g.add_nodes_from([(1, gnx.Point(1, 1)),
...                   (2, gnx.Point(2, 1)),
...                   (3, gnx.Point(3, 1))])
>>> print(g.nodes[2]["geometry"])
POINT (2 1)
```

**add_nodes_from_gdf** (*gdf*, *node_index_attr=None*)

Add nodes with the given *GeoDataFrame* and fill nodes attributes with the geodataframe columns.

**Parameters**

- **gdf** – GeoDataFrame representing nodes to add (one row for one node).
- **node_index_attr** – Node index attribute for labeling nodes. If `None`, the dataframe index is used, else the given column is used. (Default value = None)

**See also:**

*add_edges_from_gdf*

**check_nodes_validity** ()

Check that all nodes have geometries.

**copy** (*as_view=False*)

Return a copy of the graph (see `networkx.Graph.copy`).

**property crs**

Coordinate Reference System of the graph. This graph attribute appears in the attribute dict *G.graph* keyed by the string `"crs"` as well as an attribute `G.crs`

**property edges_geometry_key**
Attribute name for the edges geometry attributes. This graph attribute appears in the attribute dict *G.graph* keyed by the string `"edges_geometry_key"` as well as an attribute `G.edges_geometry_key`

**edges_to_gdf**()
Create a `gpd.GeoDataFrame` from edges of the current graph. The column representing the geometry is named after the current `edges_geometry_key` attribute.

**Returns** **gdf_edges** – The resulting GeoDataFrame : one row is an edge

**Return type** geopandas.GeoDataFrame

**See also:**

*get_edges_as_line_series*, *nodes_to_gdf*

**Return type** `GeoDataFrame`

**get_default_node_dict**()
Return the default node attribute dictionary.

**get_edges_as_line_series**()
Return the edges as a `geopandas.GeoSeries` of `shapely.geometry.LineString`.

**Returns** Series containing all edges geometries. Its CRS is the graph CRS.

**Return type** gpd.GeoSeries

**See also:**

*edges_to_gdf*, *get_nodes_as_point_series*

**Return type** `GeoSeries`

**get_node_as_point**(*node_name*)
Return a node as a `shapely.geometry.Point` object.

**Parameters** **node_name** – Name of the node on which the geometry is browsed.

**Returns** The point representing the located node.

**Return type** shapely.geometry.Point

**See also:**

*get_node_coordinates*, *get_node_coordinates*, *get_nodes_as_points*

**get_node_coordinates**(*node_name*)
Return the coordinates of the given node.

**Parameters** **node_name** – Name of the node on which the coordinates are browsed.

**Returns** A two-element list containing (x,y) coordinates of the given node.

**Return type** list

**See also:**

*get_nodes_coordinates*, *get_node_as_point*, *get_nodes_as_points*

**Return type** `list`

**get_nodes_as_multipoint**()
Return nodes geometries as a `shapely.geometry.MultiPoint`.

**Returns** MutltiPoint containing all nodes geometries.

**Return type** MultiPoint

**Return type** `MultiPoint`

**`get_nodes_as_point_series`**`()`
Return the nodes as a `geopandas.GeoSeries` of `shapely.geometry.Point`.

**Returns** Series containing all nodes geometries. Its CRS is the graph CRS.

**Return type** gpd.GeoSeries

**See also:**

*`nodes_to_gdf`*, *`get_edges_as_line_series`*

**Return type** `GeoSeries`

**`get_nodes_as_points`**`()`
Return all nodes as `shapely.geometry.Point` objects within a dictionary.

**Returns** Dictionary containing the geometry of each node of the graph.

**Return type** dict

**See also:**

*`get_node_coordinates`*, *`get_node_coordinates`*, *`get_node_as_point`*

**Return type** `dict`

**`get_nodes_coordinates`**`()`
Return all nodes coordinates within a dictionary.

**Returns** Dictionary containing the coordinates of the each node of the graph.

**Return type** dict

**See also:**

*`get_node_coordinates`*, *`get_node_as_point`*, *`get_nodes_as_points`*

**Return type** `dict`

**`get_spatial_keys`**`()`
Return the current graph spatial keys.

**Returns** Dictionary containing spatial keys (nodes and edges geometry keys and crs).

**Return type** dict

**Return type** `dict`

**`node_attr_dict_check`**`(`*attr*`)`
Check that the given attribute dictionary contains mandatory fields for a node.

**`property nodes_geometry_key`**
Attribute name for the edges geometry attributes. This graph attribute appears in the attribute dict *G.graph* keyed by the string `"edges_geometry_key"` as well as an attribute `G.nodes_geometry_key`

**`nodes_to_gdf`**`()`
Create a `geopandas.GeoDataFrame` from nodes of the current graph. The column representing the geometry is named after the current `nodes_geometry_key` attribute.

**Returns** The resulting GeoDataFrame : one row is a node

**Return type** gpd.GeoDataFrame

**See also:**

*get_nodes_as_point_series*, *edges_to_gdf*

**Return type** `GeoDataFrame`

**set_nodes_coordinates**(*coordinates*)
Set nodes coordinates with a given dictionary of coordinates (can be used for a subset of all nodes).

**Parameters** **coordinates** (*dict :*) – Dictionary mapping node names and two-element list of coordinates.

**to_crs**(*crs=None*, *epsg=None*, *inplace=False*)
Transform nodes and edges geometries to a new coordinate reference system.

**Parameters**

- **crs** (*dict or str*) – Output projection parameters as string or in dictionary form (Default value = None).
- **epsg** (*int*) – EPSG code specifying output projection.
- **inplace** (*bool*) – If True, the modification is done inplace, otherwise a new graph is created (Default value = False).

**Returns** Nothing is returned if the transformation is inplace, a new GeoGraph is returned otherwise.

**Return type** None or *GeoGraph*

**See also:**

`geopandas.GeoSeries.to_crs`

**to_directed**(*as_view=False*)
Return a directed representation of the graph (see `networkx.Graph.to_directed`).

**to_directed_class**()
Returns the class to use for empty directed copies (see `networkx.Graph.to_directed_class`).

**to_nx_class**()
Return the closest networkx class (in the inheritance graph).

**to_undirected**(*as_view=False*)
Return an undirected copy of the graph (see `networkx.Graph.to_undirected`).

**to_undirected_class**()
Returns the class to use for empty undirected copies (see `networkx.Graph.to_undirected_class`).

**to_utm**(*inplace=False*)
Project graph coordinates to the corresponding UTM (Universal Transverse Mercator)

**Parameters** **inplace** (*bool*) – If True, the modification is done inplace, otherwise a new graph is returned (Default value = False).

### Example

```
>>> import geonetworkx as gnx
>>> from shapely.geometry import Point
>>> g = gnx.GeoGraph(crs=gnx.WGS84_CRS)
>>> g.add_edge(1, 2, geometry=gnx.LineString([(4.28, 45.5), (4.32, 45.48)]))
>>> g.to_utm(inplace=True)
>>> print(g.crs)
+proj=utm +zone=31 +ellps=WGS84 +datum=WGS84 +units=m +no_defs +type=crs
>>> print(g.nodes[1]["geometry"])
POINT (600002.1723317318 5039293.296216004)
```

**See also:**

*`to_crs`*

## GeoMultiGraph

**`class GeoMultiGraph`**(*incoming_graph_data=None*, ***attr*)

Bases: *`geonetworkx.geograph.GeoGraph`*, `networkx.classes.multigraph.MultiGraph`

A undirected geographic graph class that can store multiedges.

Initialize a graph with edges, name, or graph attributes.

**Parameters**

- **`incoming_graph_data`** (*`input graph (optional, default: None)`*) – Data to initialize graph. If None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.
- **`attr`** (*`keyword arguments, optional (default= no attributes)`*) – Attributes to add to graph as key=value pairs.

**See also:**

`convert`

### Examples

```
>>> G = nx.Graph()  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1, 2), (2, 3), (3, 4)]  # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

```
>>> G = nx.Graph(e, day="Friday")
>>> G.graph
{'day': 'Friday'}
```

**`add_edge`**(*u_for_edge*, *v_for_edge*, *key=None*, ***attr*)

Add a single edge.

This method exists only for reflecting nx.MultiGraph method so that the multiple inheritance scheme works.

#### Examples

```
>>> import geonetworkx as gnx
>>> g = gnx.GeoMultiGraph()
>>> g.add_edge(1, 2, 0, geometry=gnx.LineString([(5, 4), (2, 7)]))
0
>>> print(g.nodes[1]["geometry"])
POINT (5 4)
```

**to_directed**(*as_view=False*)
Return a directed representation of the graph (see `networkx.MultiGraph.to_directed`).

**to_directed_class**()
Returns the class to use for empty directed copies (see `networkx.MultiGraph.to_directed_class`).

**to_nx_class**()
Return the closest networkx class (in the inheritance graph).

**to_undirected**(*as_view=False*)
Return an undirected copy of the graph (see `networkx.MultiGraph.to_undirected`).

**to_undirected_class**()
Returns the class to use for empty undirected copies (see `networkx.MultiGraph.to_undirected_class`)..

### GeoDiGraph

**class GeoDiGraph**(*incoming_graph_data=None*, ***attr*)
Bases: `geonetworkx.geograph.GeoGraph`, `networkx.classes.digraph.DiGraph`

Base class for directed geographic graphs.

Because edges are directed, it supposes that the edges lines are well-ordered. Namely, that the first point of the line matches with the coordinates of the first vertex of the edge (or is at least close) and vice versa with the last point of the line and the second. If this is not the case, the method `order_well_lines` can be useful to make sure of that.

Initialize a graph with edges, name, or graph attributes.

**Parameters**

- **incoming_graph_data** (*input graph (optional, default: None)*) – Data to initialize graph. If None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.
- **attr** (*keyword arguments, optional (default= no attributes)*) – Attributes to add to graph as key=value pairs.

**See also:**

`convert`

#### Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1, 2), (2, 3), (3, 4)]  # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

```
>>> G = nx.Graph(e, day="Friday")
>>> G.graph
{'day': 'Friday'}
```

**`to_directed`**(*as_view=False*)
Return a directed representation of the graph (see `networkx.DiGraph.to_directed`).

**`to_directed_class`**()
Returns the class to use for empty directed copies (see `networkx.DiGraph.to_directed_class`).

**`to_nx_class`**()

**`to_undirected`**(*reciprocal=False*, *as_view=False*)
Return an undirected copy of the graph (see `networkx.DiGraph.to_undirected`).

**`to_undirected_class`**()
Returns the class to use for empty undirected copies (see `networkx.DiGraph.to_undirected_class`).

### GeoMultiDiGraph

**`class GeoMultiDiGraph`**(*incoming_graph_data=None*, ***attr*)
Bases: *`geonetworkx.geomultigraph.GeoMultiGraph`*, *`geonetworkx.geodigraph.GeoDiGraph`*, `networkx.classes.multidigraph.MultiDiGraph`

A directed geographic graph class that can store multiedges.

Initialize a graph with edges, name, or graph attributes.

**Parameters**

- **`incoming_graph_data`** (*`input graph (optional, default: None)`*) – Data to initialize graph. If None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.
- **`attr`** (*`keyword arguments, optional (default= no attributes)`*) – Attributes to add to graph as key=value pairs.

**See also:**

`convert`

#### Examples

```
>>> G = nx.Graph()   # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1, 2), (2, 3), (3, 4)]  # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

```
>>> G = nx.Graph(e, day="Friday")
>>> G.graph
{'day': 'Friday'}
```

**`to_directed`**(*as_view=False*)
Return a directed representation of the graph (see `networkx.MultiDiGraph.to_directed`).

**`to_directed_class`**()
Returns the class to use for empty directed copies (see `networkx.MultiDiGraph.to_directed_class`).

**`to_nx_class`**()
Return the closest networkx class (in the inheritance graph).

**`to_undirected`**(*reciprocal=False*, *as_view=False*)
Return an undirected copy of the graph (see `networkx.MultiDiGraph.to_undirected`).

**`to_undirected_class`**()
Returns the class to use for empty undirected copies (see `networkx.MultiDiGraph.to_undirected_class`).

## 1.6.2 Tools

### Spatial Merge

**`spatial_graph_merge`**(*base_graph*, *other_graph*, *inplace=False*, *merge_direction='both'*, *node_filter=None*, *intersection_nodes_attr=None*, *discretization_tol=None*)
Operates spatial merge between two graphs. Spatial edge projection is used on merging nodes (see `spatial_points_merge`). The `base_graph` attributes have higher priority than the `other_graph` attributes ( i.e. if graphs have common graph attributes, nodes or edges, the `base_graph` attributes will be kept).

**Parameters**

- **`base_graph`** (`GeoGraph`, `GeoDiGraph`, `GeoMultiGraph` *or* `GeoMultiDiGraph`) – Base graph on which the merge operation is done.
- **`other_graph`** (`GeoGraph`, `GeoDiGraph`, `GeoMultiGraph` *or* `GeoMultiDiGraph`) – Input graph to merge. Modified graph if operation is done inplace.
- **`inplace`** (*`bool`*) – If True, do operation inplace and return None. (Default value = False)
- **`merge_direction`** (*`str`*) – See `spatial_points_merge` (Default value = "both")
- **`node_filter`** – Lambda returning if a given node (from the `other_graph` graph) has to be merged. (Default value = None)
- **`intersection_nodes_attr`** (*`str`*) – A dictionary of attributes (constant for all added intersection nodes). (Default value = None)

- **discretization_tol** (*float*) – A custom discretization tolerance for lines. If None, tolerance with the right order of magnitude is pre-defined for some CRS. For more details, see `gnx.get_default_discretization_tolerance` method. (Default value = None)

**Returns** A new graph with the same type as `base_graph` if not inplace.

**Return type** None or *GeoGraph*

**See also:**

*spatial_points_merge*

**spatial_points_merge**(*graph*, *points_gdf*, *inplace=False*, *merge_direction='both'*, *node_filter=<function no_filter>*, *edge_filter=<function no_filter>*, *intersection_nodes_attr=None*, *discretization_tol=None*)

Merge given points as node with a spatial merge. Points are projected on the closest edge of the graph and an intersection node is added if necessary. If two nodes a given point and a node have the same name, with equal coordinates, then the node is considered as already in the graph. A discretization tolerance is used for indexing edges lines. New nodes created from the geodataframe have attributes described by other columns (except if an attribute value is *nan*). When a point is projected on an edge, this edge is removed and replaced by two others that connect the extremities to the intersection node. A reference to the original edge is kept on theses new edges with the attribute `settings.ORIGINAL_EDGE_KEY`. The original edge is the oldest parent of the new edge, to have the direct parent, the attribute has to be cleant first.

**Parameters**

- **graph** (`GeoGraph`, `GeoDiGraph`, `GeoMultiGraph` *or* `GeoMultiDiGraph`) – A GeoGraph or derived class describing a spatial graph.
- **points_gdf** (*gpd.GeoDataFrame*) – A list of point describing new nodes to add.
- **inplace** (*bool*) – If True, do operation inplace and return None. (Default value = False)
- **merge_direction** (*str*) – For directed graphs only:
  - `'both'`: 2 edges are added: graph -> new node and new node -> graph
  - `'in'`: 1 edge is added: new_node -> graph
  - `'out'`: 1 edge is added: graph -> new_node (Default value = "both")
- **node_filter** – A node filter (lambda) to exclude nodes (and by the way all concerned edges) from the projection operation. (Default value = no_filter)
- **edge_filter** – An edge filter (lambda) to exclude edges on which the projection will not take place. (Default value = no_filter)
- **intersection_nodes_attr** (*dict*) – A dictionary of attributes (constant for all added intersection nodes). (Default value = None)
- **discretization_tol** (*float*) – A custom discretization tolerance for lines. If None, tolerance with the right order of magnitude is pre-defined for some CRS. For more details, see `gnx.get_default_discretization_tolerance` method. (Default value = None)

**Returns** If not inplace, the created graph.

**Return type** None or *GeoGraph*

**See also:**

*spatial_graph_merge*

**Return type** `GeoGraph`

### Isochrones

**`boundary_edge_buffer`**(*line*)

Return the edge buffer polygon on the oriented line. This represented the area where all points are reachable starting from the line first extremity and using the closest edge projection rule.

**Return type** `Union[Polygon, MultiPolygon]`

**`get_alpha_shape_polygon`**(*points*, *quantile*)

Return the alpha-shape polygon formed by the given points. Alpha parameter is determined using a quantile of circumradius of Delaunay triangles.

**Parameters**

- **`points`** (*`list`*) – List of input points (2D)
- **`quantile`** (*`float`*) – Quantile on circumradius to determine alpha (100 returns the convex hull, 0 returns an empty polygon). `0 <= quantile <= 100`.

**Returns**

The polygon formed by all triangles having a circumradius inferior or equal to $1/\alpha$.

Note that this does not return the exhaustive alpha-shape for low quantiles, the minimum spanning tree LineString should be added to the returned polygon. This is adapted from Sean Gillies code.

**Return type** Polygon or MultiPolygon

**Return type** `Union[Polygon, MultiPolygon]`

**`get_edges_voronoi_cells`**(*graph*, *tolerance=1e-07*)

Return edge voronoi cells as *GeoSeries*.

**Return type** `GeoSeries`

**`get_point_boundary_buffer_polygon`**(*point_coords*, *radius*, *segment_direction*, *resolution=16*)

Returns a half-disk centered on the given point, with the given radius and having the boundary edge orthogonal to the given segment direction. See `boundary_edge_buffer`.

**Return type** `Polygon`

**`get_segment_boundary_buffer_polygon`**(*segment_coords*, *radius*, *residual_radius*)

Return a segment boundary polygon using given radius. It represents all reachable points from the first extremity of the segment. The returned polygon is a trapeze. See `boundary_edge_buffer`.

**Return type** `Polygon`

**`isochrone_polygon`**(*graph*, *source*, *limit*, *weight='length'*, *tolerance=1e-07*)

Return a polygon approximating the isochrone set in the geograph.

**Parameters**

- **`graph`** (*`Geograph`*) – Graph representing possible routes.
- **`source`** – Source node from where distance is computed
- **`limit`** (*`float or int`*) – Isochrone limit (e.g. 100 meters, 5 minutes, depending on `weight` unit).
- **`weight`** (*`str`*) – Weight attribute on edges to compute distances (edge weights should be non-negative). (Default value = “length”)
- **`tolerance`** (*`float`*) – Tolerance to compute Voronoi cells. (Default value = 1e-7)

**Returns** A polygon representing all reachable points within the given limit from the source node.

**Return type** Polygon or MultiPolygon

**Return type** `Union[Polygon, MultiPolygon]`

**`isochrone_polygon_with_alpha_shape`**(*graph*, *source*, *limit*, *weight='length'*, *alpha_quantile=99.0*, *remove_holes=True*, *tolerance=1e-07*)

Returns an approximation of the isochrone polygon using an alpha-shape of the Shortest Path Tree.

**Parameters**

- **`graph`** (`GeoGraph`) – GeoGraph to browse
- **`source`** – Source node from where distance is computed
- **`limit`** (*`float or int`*) – Isochrone limit (e.g. 100 meters, 5 minutes, depending on `weight` unit).
- **`weight`** (*`str`*) – Weight attribute on edges to compute distances (edge weights should be non-negative). (Default value = "length")
- **`alpha_quantile`** (*`float`*) – Quantile on circumradius to determine alpha (100 returns the convex hull, 0 returns an empty polygon). `0 <= quantile <= 100`. (Default value = 99.0)
- **`remove_holes`** (*`bool`*) – If `True` remove holes in the returned polygon. (Default value = True)
- **`tolerance`** (*`float`*) – Buffering tolerance on polygon for rendering (Default value = 1e-7)

**Returns** A polygon approximating the isochrone.

**Return type** Polygon or MultiPolygon

**Return type** `Union[Polygon, MultiPolygon]`

### 1.6.3 Geometry operations

**`class Extremity`**(*shape_id*, *position*, *coords*)

Bases: `object`

Represents an extremity of a line. It's useful to parse and deal with lines given as input.

**`almost_equally_located`**(*p1*, *p2*, *tolerance=1e-08*)

Test if two point are loacated at the same place within a tolerance.

**Parameters**

- **`p1`** (*`Point`*) – First point to compare
- **`p2`** (*`Point`*) – Second point to compare
- **`tolerance`** – Comparison tolerance (Default value = 1e-8)

**Returns** True if the two points have the same coordinates.

**Return type** bool

**Return type** `bool`

**`convert_multilinestring_to_linestring`**(*gdf*)

**Convert all geometry attribute being a 'MultiLineString' to a 'LineString'. The created line is a merge of all**
sublines.

**Parameters** **gdf** (*gpd.GeoDataFrame*) – A GeoDataFrame with a 'geometry' column to modify

**Returns** The number of converted 'MultiLineString'

**Return type** int

**Raises** **RuntimeError** – If an input shape is not a LineString or a MultiLineString

**Return type** `int`

**coordinates_almost_equal**(*c1*, *c2*, *tolerance=1e-08*)

Returns true if the two given list of coordinates equals within a given tolerance.

**Parameters**

- **c1** (*Iterable*) – First point coordinates
- **c2** (*Iterable*) – Second point coordinates
- **tolerance** (*float*) – Tolerance comparison (Default value = 1e-8)

**Returns** True if the coordinates almost equal, false otherwise.

**Return type** bool

**Return type** `bool`

**discretize_line**(*line*, *discretization_tol*)

Takes a shapely LineString and discretize it into a list of shapely Points. Each point is at most at the discretization tolerance distance of the following point.

**Parameters**

- **line** (*LineString*) – Line to discretize
- **discretization_tol** (*float*) – Maximum distance between two points on the line.

**Returns** An ordered list of shapely Point

**Return type** list

**See also:**

*discretize_lines*

**Return type** `list`

**discretize_lines**(*lines*, *discretization_tol*)

Discretize some line into points.

**Parameters**

- **lines** (*Iterable[LineString] :*) – Lines to discretize
- **discretization_tol** (*float*) – Maximum distance between two points on the line.

**Returns** Return all the discretized points as a shapely MultiPoint and a dictionary to map the discretized points for each line.

**Return type** MultiPoint and defaultdict

**See also:**

*discretize_line*

**get_closest_line_from_point**(*point_from*, *lines_to=None*, *discretization_tol=None*, *kd_tree=None*, *points_line_association=None*)

Find the closest line from a given point.

**Parameters**

- **`point_from`** (*`PointCoordinatesLike`*) – Point coordinate to find the closest line.
- **`lines_to`** (*`list`*) – Group of lines among which the closest has to be found (optional if `kdtree` and `points_line_association` are given). (Default value = None)
- **`discretization_tol`** (*`float`*) – Maximum distance between discretized points (optional if `kdtree` and `points_line_association` are given). (Default value = None)
- **`kd_tree`** (*`cKDTree`*) – An optional pre-computed kd_tree of discretized lines. (Default value = None)
- **`points_line_association`** (*`dict`*) – An optional pre-computed dictionary matching lines and discretized points. (Default value = None)

**Returns**

- *float* – distance
- *int* – index of the closest line

**`get_closest_line_from_points`** (*points_from*, *lines_to*, *discretization_tol*)
Find the closest line for each given points.

**Parameters**

- **`points_from`** (*`list`*) – Points coordinates.
- **`lines_to`** (*`list`*) – Group of lines among which the closest has to be found.
- **`discretization_tol`** (*`float`*) – Maximum distance between discretized points

**Returns** A list of closest lines indexes.

**Return type** list

**`get_closest_point_from_line`** (*line_from*, *discretization_tol*, *points_to=None*, *kd_tree=None*)
Return the closest point from a given line and its distance.

**Parameters**

- **`line_from`** (*`LineString`*) – A shapely LineString (Default value = None)
- **`discretization_tol`** (*`float`*) – Maximum distance between two discretized points on the line.
- **`points_to`** (*`list`*) – A list of points among which the closest to the line has to be found (optional is `kdtree` is given)
- **`kd_tree`** (*`cKDTree`*) – A kd-tree representing the points among which the closest to the line has to be found (optional if `points_to` is given) (Default value = None)

**Returns**

- *float* – closest distance
- *int* – index of the closest point

**`get_closest_point_from_multi_shape`** (*multi_shape*, *points_to=None*, *kd_tree=None*)
Computes the closest point to the multi shape (i.e. the point that has the smallest projection distance on the entire multi shape object.

**Parameters**

- **`multi_shape`** (*`MultiPoint or MultiLineString`*) – The multi shape object can be any shapely object among: MultiPoint, MultiLineString

- **points_to** (*list*) – A list of points among which to find the closest to the multi shape (Default value = None)
- **kd_tree** (*cKDTree*) – A kdtree representing the points among which the closest to the multishape has to be found (optional if 'points_to' is given) (Default value = None)

**Returns**

- *float* – distance
- *int* – index of the closest point

**get_closest_point_from_points**(*points_from*, *points_to=None*, *kd_tree=None*)

Compute the closest point among the `points_from` list for each point in the `points_to` list.

**Parameters**

- **points_from** (*PointsCoordinatesLike*) – Iterable of points coordinates
- **points_to** (*list or None*) – Iterable of points coordinates (Default value = None)
- **kd_tree** (*cKDTree*) – a constructed kd tree representing `points_from` (Default value = None)

**Returns**

- *array of floats* – distances
- *ndarray of ints* – indexes

**get_closest_point_from_shape**(*shape*, *points_to=None*, *kd_tree=None*)

Compute the closest point to the given shape.

**Parameters**

- **shape** (*Point, MultiPoint, LineString or MultiLineString*) – Any shapely shape
- **points_to** (*list*) – A list of points among which to find the closest to the multi shape (Default value = None)
- **kd_tree** (*cKDTree*) – A kdtree representing the points among which the closest to the shape has to be found (optional if 'points_to' is given) (Default value = None)

**Returns**

- *float* – distance
- *int* – index of the closest point

**get_closest_point_from_shapes**(*shapes_from*, *points_to*)

Compute the closest point for each given shape.

**Parameters**

- **shapes_from** – An iterable of shapes (Point, MultiPoint, LineString, MultiLineString)
- **points_to** (*list*) – A list of points among which to find the closest to the multi shape

**Returns**

- *float* – distance
- *int* – index of the closest point

**get_default_discretization_tolerance**(*crs*)

Return a discretization tolerance with the right order of magnitude for the given crs.

#### Examples

```
>>> import geonetworkx as gnx
>>> print(gnx.get_default_discretization_tolerance("epsg:3857"))
3.0
```

**`get_polygons_neighborhood`**(*polygons*)
Returns for each polygon a set of intersecting polygons.

**`get_shape_extremities`**(*shape*, *shape_id*)
Return the extremities of a shape in the network_shapes_gdf.

**`insert_point_in_line`**(*line*, *point_coords*, *position*)
Insert a new point in a line given its coordinates.

**Return type** `LineString`

**`merge_two_lines_with_closest_extremities`**(*first_line*, *second_line*)
Merge two lines with their closest extremities. Euclidian distance is used here.

**Return type** `LineString`

**`merge_two_shape`**(*e1*, *e2*, *line1*, *line2*)
Merge two lines (line1 and line2) with the given extremities (e1 and e2).

**Return type** `LineString`

**`split_line`**(*line*, *distance*)
Cuts a line in two at a distance from its starting point.

## 1.6.4 Utils

### Geograph utils

**`approx_map_unit_factor`**(*point*, *tolerance=1e-07*, *method='geodesic'*)
Compute a linear approximation of the map unit factor `u` for 1 meter:

$$d(p_1, p_2) pprox ||p1 - p2||_2 imesu$$

This can be useful to not change the CRS of geograph. The approximation can be very wrong for long distances.

**Parameters**

- **`point`** – Point where the approximation is computed.
- **`tolerance`** – Precision for the iterative method
- **`method`** – Distance method (geodesic, great_circle, vincenty)

**Returns**

**Return type** The linear approximation unit factor.

#### Examples

```
>>> import geonetworkx as gnx
>>> p1 = gnx.Point(-73.614, 45.504)
>>> u = gnx.approx_map_unit_factor(p1)
```

(continues on next page)

(continued from previous page)

```
>>> p2 = gnx.Point(-73.613, 45.502)
>>> "%.2f" % gnx.geodesic_distance(p1, p2)
'235.62'
>>> "%.2f" % (gnx.euclidian_distance(p1, p2) / u)
'214.83'
```

**Return type** `float`

**`compose`**(*G*, *H*)
Return a new graph of G composed with H. Makes sure the returned graph is consistent with respect to the spatial keys. (See `nx.compose`). According to the priority rule in networkX, attributes from `H` take precedent over attributes from G.

**Return type** *`GeoGraph`*

**`crs_equals`**(*crs1*, *crs2*)
Compare CRS using `pyproj.Proj` objects.

**Return type** `bool`

**`euclidian_distance`**(*p1*, *p2*)
Return the euclidian distance between the two points

**Parameters**

- **p1** (*`Point`*) – The first shapely Point
- **p2** (*`Point`*) – The second shapely Point

**Returns** The euclidian distance

**Return type** float

**Return type** `float`

**`euclidian_distance_coordinates`**(*c1*, *c2*)
Return the euclidian distance between the two sets of coordinates.

**Return type** `float`

**`fill_edges_missing_geometry_attributes`**(*graph*)
Add a geometry attribute to the edges that don't have any. The created geometry is a straight line between the two nodes.

**Parameters** **graph** (`GeoGraph`) – graph to fill

**`fill_elevation_attribute`**(*graph*, *attribute_name='elevation[m]'*, *only_missing=True*)
Fill the `elevation[m]` attribute on nodes of the given geograph. The elevation is found with the *srtm* package. Graph crs has to be WGS84 standard, otherwise elevation data won't be consistent.

**Parameters**

- **graph** (`GeoGraph`) – GeoGraph to modify
- **attribute_name** (*`str`*) – Attribute to fill (Default value = "elevation[m]")
- **only_missing** (*`bool`*) – Get the elevation and set it only if the node attribute is missing. (Default value = True)

### Examples

```
>>> import geonetworkx as gnx
>>> g = gnx.GeoGraph(crs=gnx.WGS84_CRS)
>>> g.add_edge(1, 2, geometry=gnx.LineString([(5.15, 45.504), (5.167, 45.506)]))
>>> gnx.fill_elevation_attribute(g)
>>> print(g.nodes[1]["elevation[m]"])
393
```

**fill_length_attribute**(*graph*, *attribute_name='length'*, *only_missing=True*, *method=None*)

Fill the `'length'` attribute of the given networkX Graph.

**Parameters**

- **graph** (`GeoGraph`) – graph to fill
- **attribute_name** (*str*) – The length attribute name to set (Default value = "length")
- **only_missing** (*bool*) – Compute the length only if the attribute is missing (Default value = True)
- **method** (*str*) – Method to compute the distance

### Examples

```
>>> import geonetworkx as gnx
>>> g = gnx.GeoGraph(crs=gnx.WGS84_CRS)
>>> g.add_edge(1, 2, geometry=gnx.LineString([(-73.614, 45.504), (-73.632, 45.
↪506)]))
>>> gnx.fill_length_attribute(g)  # using geodesic distance
>>> "%.2f" % g.edges[(1, 2)]["length"]
'1424.17'
>>> g.to_utm(inplace=True)
>>> gnx.fill_length_attribute(g, only_missing=False)
>>> "%.2f" % g.edges[(1, 2)]["length"]  # using euclidian distance in UTM
'1423.81'
```

**geodesic_distance**(*p1*, *p2*)

**Return geopy geodesic distance with two given point in the** long/lat format.

**Parameters**

- **p1** – First 2D point
- **p2** – Second 2D point

**Returns**

**Return type** The geodesic distance in meters.

### Examples

```
>>> import geonetworkx as gnx
>>> p1 = gnx.Point(-73.614, 45.504)  # long/lat format
>>> p2 = gnx.Point(-73.632, 45.506)
>>> "%.2f" % gnx.geodesic_distance(p1, p2)
'1424.17'
```

**Return type** `float`

**`geographical_distance`**(*graph*, *node1*, *node2*, *method='great_circle'*)

Return the geographical distance between the two given nodes.

**Parameters**

- **`graph`** (`Geograph`) – Geograph
- **`node1`** – First node label
- **`node2`** – Second node label
- **`method`** (`str`) – "vincenty", "euclidian", "great_circle" (Default value = "great_circle")

**Returns** Distance between nodes, unit depends on the method.

**Return type** float

**Return type** `float`

**`get_closest_nodes`**(*graph*, *point*, *k*, ***kwargs*)

Return the `k` closest nodes from the given point.

Euclidian distance is used here by default.

**Parameters**

- **`graph`** – Geograph on which nodes are browsed
- **`point`** – Query point on which the distances from nodes are computed.
- **`k`** – Number of nodes to return.
- **`kwargs`** – Additional parameters to send to *scipy.spatial.cKDTree.query* method.

**Returns** A list containing closest nodes labels.

**Return type** list

## Examples

```
>>> import geonetworkx as gnx
>>> g = gnx.GeoGraph()
>>> g.add_nodes_from([(1, gnx.Point(1, 1)),
...                   (2, gnx.Point(-1, 3)),
...                   (3, gnx.Point(-1, -4)),
...                   (4, gnx.Point(-1, -1)),
...                   (5, gnx.Point(-10, 10))])
>>> cns = gnx.get_closest_nodes(g, gnx.Point(0, 0), 3)
>>> print(cns)
[1, 4, 2]
```

**Return type** `list`

**`get_crs_as_str`**(*crs*)

Return the given CRS as string `pyproj.Proj` methods.

**Return type** `str`

**`get_default_distance_method_from_crs`**(*crs*)

Return the default method for computing distances for the given CRS.

**Parameters** **`crs`** – Coordinate Reference System

**Returns**

**Return type** String representing the distance calculation method.

**Return type** `str`

**`get_distance`**(*p1*, *p2*, *method*)
Return the distance between the two given points with the given method.

**Return type** `float`

**`get_graph_bounding_box`**(*graph*)
Return the bounding box coordinates of the given GeoGraph. It takes into account nodes and edges geometries.

**`get_line_ordered_edge`**(*graph*, *e*, *line*)
Return the given edge with the first node of the edge representing the first line point and the second node the last edge point. The closest node rule is applied.

**`get_line_start`**(*graph*, *e*, *line*)
For a given edge, return the node constituting the line start with a closest node rule.

**`get_new_node_unique_name`**(*graph*, *name*)
Return a new unique node name from an initial node name. A counter suffix is added at the end if the node name is already used.

**Parameters**

- **`graph`** (*`nx.Graph`*) – A given graph
- **`name`** (*`str`*) – A initial node name

**Returns** A unique name not in `graph.nodes()`.

**Return type** str

**`get_surrounding_nodes`**(*graph*, *point*, *r*, ***kwargs*)
Return all nodes that are within distance `r` of given point.

Euclidian distance is used here by default.

**Parameters**

- **`graph`** – Geograph on which nodes are browsed
- **`point`** – Query point on which the distances from nodes are computed.
- **`r`** – Maximum distance between point and nodes to return.
- **`kwargs`** – Additional parameters to send to scipy.spatial.cKDTree.query_ball_point method.

**Returns** A list containing nodes labels that are within the distance.

**Return type** list

### Examples

```
>>> import geonetworkx as gnx
>>> g = gnx.GeoGraph()
>>> g.add_nodes_from([(1, gnx.Point(1, 1)),
...                   (2, gnx.Point(-1, 3)),
...                   (3, gnx.Point(-1, -4)),
```

(continues on next page)

(continued from previous page)

```
...                         (4, gnx.Point(-1, -1)),
...                         (5, gnx.Point(-10, 10))])
>>> sns = gnx.get_surrounding_nodes(g, gnx.Point(0, 0), 1.5)
>>> print(sns)
[1, 4]
```

**Return type** `list`

**get_utm_crs**(*p*)
Return the Universal Transverse Mercator CRS with a given in point in long-lat format.

**great_circle_distance**(*p1*, *p2*)

**Return geopy great circle distance with two given point in the** long/lat format.

**Parameters**

- **p1** – First 2D point
- **p2** – Second 2D point

**Returns**

**Return type** The great circle distance in meters.

## Examples

```
>>> import geonetworkx as gnx
>>> p1 = gnx.Point(-73.614, 45.504)  # long/lat format
>>> p2 = gnx.Point(-73.632, 45.506)
>>> "%.2f" % gnx.great_circle_distance(p1, p2)
'1420.27'
```

**Return type** `float`

**hard_write_spatial_keys**(*graph*)
Write spatial keys in the graph attribute, so that if the default keys are used, they are propagated for special operations (e.g. composing graphs).

**is_null_crs**(*crs*)
Test for null crs values.

**Return type** `bool`

**join_lines_extremity_to_nodes_coordinates**(*graph*)
Modify the edges geometry attribute so that lines extremities match with nodes coordinates.

**Parameters** **graph** (`GeoGraph`) – A geograph to modify

**measure_line_distance**(*line*, *method*)
Measure the length of a shapely LineString object using the vincenty distance.

**Parameters**

- **line** – Linestring to measure. Coordinates have to be (in the WGS-84 ellipsoid model)
- **method** – Method to compute the distance

**Returns** distance in meters of the linestring.

**Return type** float

**Return type** `float`

**`order_well_lines`**(*graph*)

Try to order well each geometry attribute of edges so that the first coordinates of the line string are the coordinates of the first vertex of the edge. The closest node rule is applied. If the graph is not oriented, the modification will be inconsistent (nodes declaration in edges views are not ordered). Euclidian distance is used here.

**Parameters** **`graph`** (`GeoGraph`) – Graph on which to apply the ordering step. Modification is inplace.

**`rename_edges_attribute`**(*graph*, *old_name*, *new_name*)

Rename edges attribute defined by its old name to a new name.

**`rename_nodes_attribute`**(*graph*, *old_name*, *new_name*)

Rename nodes attribute defined by its old name to a new name.

**`stringify_nodes`**(*graph*, *copy=True*)

Modify the graph node names into strings.

**`vincenty_distance`**(*p1*, *p2*)

**Return geopy great circle distance with two given point in the** long/lat format.

**Parameters**

- **`p1`** – First 2D point
- **`p2`** – Second 2D point

**Returns**

**Return type** The vincenty distance in meters.

### Examples

```
>>> import geonetworkx as gnx
>>> p1 = gnx.Point(-73.614, 45.504)  # long/lat format
>>> p2 = gnx.Point(-73.632, 45.506)
>>> "%.2f" % gnx.vincenty_distance(p1, p2)
'1424.17'
```

**Return type** `float`

**`vincenty_distance_coordinates`**(*p1*, *p2*)

Returns the vincenty distance in meters with given coordinates.

**Return type** `float`

## Voronoi utils

**class `PyVoronoiHelper`**(*points*, *segments*, *bounding_box_coords*, *scaling_factor=100000.0*)

Bases: `object`

Add-on for the pyvoronoi (boost voronoi) tool. It computes the voronoi cells within a bounding box.

**static `add_polygon_coordinates`**(*coordinates*, *point*)

Add given point to given coordinates list if is not the equal to the last coordinates.

**clip_infinite_edge**(*cell_coords*, *edge*, *eta*)
Fill infinite edge coordinate by placing the infinite vertex to a `eta` distance of the known vertex.

**get_cells_as_gdf**()
Returns the voronoi cells in *geodataframe* with a column named *id* referencing the index of the associated input geometry.

**Return type** `GeoDataFrame`

**get_cells_as_polygons**()
Return the voronoi cells as polygons trimmed with the bounding box.

**Return type** `dict`

**get_cells_coordiates**(*eta=1.0*, *discretization_tolerance=0.05*)
“Parse the results of `pyvoronoi` to compute the voronoi cells coordinates. The infinite ridges are projected at a `eta` distance in the ridge direction.

**Parameters**

- **eta** (*float*) – Distance for infinite ridges projection. (Default value = 1.0)
- **discretization_tolerance** (*float*) – Discretization distance for curved edges. (Default value = 0.05)

**Returns** A dictionary mapping the cells ids and their coordinates.

**Return type** dict

**Return type** `dict`

**static repair_bowtie_polygon**(*polygon*)
Repair an invalid polygon for the ‘bowtie’ case.

**Return type** `MultiPolygon`

**static repair_polygon**(*polygon*)
Repair an invalid polygon. It works in most cases but it has no guarantee of success.

**Return type** `Union`[`Polygon`, `MultiPolygon`]

**compute_voronoi_cells_from_lines**(*lines*, *tolerance=1e-07*)
Compute the voronoi cells of given generic lines. Input linestrings can be not simple.

**Parameters**

- **lines** (*list*) – List of `LineString`
- **tolerance** (*float*) – Tolerance for the voronoi cells computation (Two points will be considered equal if their coordinates are equal when rounded at `tolerance`). (Default value = 1e-7)

**Returns** A list of cells geometries.

**Return type** list

**Return type** `list`

**split_as_simple_segments**(*lines*, *tol=1e-07*)
Split a list of lines to simple segments (linestring composed by two points). All returned segments do not cross themselves except at extremities.

**Parameters**

- **lines** (*list*) – List of lines to split
- **tol** (*float*) – Tolerance to test if a line is a sub line of another one. (Default value = 1e-7)

**Returns** A dictionary mapping for each input line index, the list of simple segments.

**Return type** defaultdict

**Return type** `defaultdict`

**`split_linestring_as_simple_linestrings`**(*line*)
Split a linestring if it is not simple (i.e. it crosses itself).

**Return type** `list`

## Generators utils

**`_get_ego_boundaries`**(*graph*, *ego_graph*, *n*, *radius*, *distance=None*)
Retrieve all information to build an extended ego-graph. See `gnx.extended_ego_graph` for more info.

**Return type** `tuple`

**`add_ego_boundary_nodes`**(*graph*, *n*, *radius*, *distance*, *undirected=False*)
Modify the given graph to add boundary nodes at exact radius distance. An edge $(u, v)$ is a boundary edge if $(u, v)$ if $d(n, u) < radius < d(n, v)$. A boundary node is added on the edge to represent the ego- graph limit. See `gnx.extended_ego_graph` for more info.

**Parameters**

- **`graph`** (`GeoGraph, GeoDiGraph, GeoMultiGraph or GeoMultiDiGraph`) – Input graph to modify
- **`n`** – A single source node
- **`radius`** (*`float or int`*) – Include all neighbors of distance<=radius from n.
- **`distance`** (*`str`*) – Use specified edge data key as distance. For example, setting distance='weight' will use the edge weight to measure the distance from the node n.
- **`undirected`** (*`bool`*) – If True use both in- and out-neighbors of directed graphs. (Default value = False)

**See also:**

*`extended_ego_graph`*

**`extended_ego_graph`**(*graph*, *n*, *radius=1*, *center=True*, *undirected=False*, *distance=None*)
Returns induced subgraph of neighbors centered at node n within a given radius extended by interpolated nodes on boundary edges.

Note that the returned graph is not a subgraph of the input graph because it will have boundary nodes in addition. It means that a node is added on each edge leaving the ego-graph to represent the furthest reachable point on this edge. The boundary node is added at given node using a linear interpolation. A boundary node $b$ will be added on the edge $(u, v)$ if $d(n, u) < radius < d(n, v)$. The boundary will be placed along the edge geometry at the following distance:

$$d(u, b) = \frac{\text{radius} - d(n, u)}{d(u, v)}$$

Furthermore, the attribute `distance` is filled with the value $d(u, b)$.

**Parameters**

- **`graph`** (`GeoGraph, GeoDiGraph, GeoMultiGraph or GeoMultiDiGraph`) – A Geograph or subclass
- **`n`** – A single source node

- **radius** (*float or int*) – Include all neighbors of distance<=radius from n. (Default value = 1)
- **center** (*bool*) – If False, do not include center node in graph (Default value = True)
- **undirected** (*bool*) – If True use both in- and out-neighbors of directed graphs. (Default value = False)
- **distance** (*str*) – Use specified edge data key as distance. For example, setting distance='weight' will use the edge weight to measure the distance from the node n. (Default value = None)

**Returns** The resulting graph with node, edge, and graph attributes copied.

**Return type** *GeoGraph*, *GeoDiGraph*, *GeoMultiGraph* or *GeoMultiDiGraph*

**See also:**

`add_ego_boundary_nodes`

**Return type** `GeoGraph`

### 1.6.5 Read and write

**cast_for_fiona** (*gdf*)

Transform elements so that attributes can be writable by fiona.

**Parameters gdf** (*gpd.GeoDataFrame :*) – GeoDataFrame to modify

**get_graph_with_wkt_geometry** (*geograph*)

Modify the edges geometry attribute to a well-known text format to make the graph writable is some text formats. The returned graph is not as operational as the given one (edge geometries has been removed).

**Parameters geograph** (*GeoGraph :*) – Geograph to transform

**Returns** A networkx graph with WKT geometries instead of shapely objects.

**Return type** nx.Graph

**See also:**

`parse_nodes_attribute_as_wkt`

**Return type** `Graph`

**graph_edges_to_gdf** (*graph*)

**Create and fill a GeoDataFrame (geopandas) from edges of a networkX graph. The `'geometry'` attribute is used** for shapes.

**Parameters graph** (*nx.Graph*) – Graph to parse

**Returns** The resulting GeoDataFrame : one row is an edge

**Return type** gpd.GeoDataFrame

**Return type** `GeoDataFrame`

**graph_nodes_to_gdf** (*graph*)

Create and fill a GeoDataFrame (geopandas) from nodes of a networkX graph. The `'geometry'` attribute is used for shapes.

**Parameters graph** (`GeoGraph`) – Graph to parse

**Returns** The resulting GeoDataFrame : one row is a node

**Return type** gpd.GeoDataFrame

**Return type** `GeoDataFrame`

**`parse_bool_columns_as_int`**(*gdf*)

Transform bool columns into integer columns.

**Parameters** **`gdf`** (*`gpd.GeoDataFrame :`*) – GeoDataFrame to modify

**`parse_edges_attribute_as_wkt`**(*graph*, *attribute_name*)

Transform a graph edges attribute from WKT to shapely objects. Attribute is replaced.

**Parameters**

- **`graph`** (*`nx.Graph :`*) – Graph to modify and parse
- **`attribute_name`** (*`str :`*) – Attribute to parse the edges geometries

**See also:**

*`get_graph_with_wkt_geometry`*, *`parse_nodes_attribute_as_wkt`*

**`parse_graph_as_geograph`**(*graph*, ***attr*)

Parse a `networkx.Graph` as a `geonetworkx.GeoGraph` with the closest geonetworkx graph type.

**Parameters**

- **`graph`** (*`nx.Graph, nx.DiGraph, nx.MultiGraph or nx.MultiDiGraph`*) –
- **`**attr`** – Potential spatial keys.

**Returns** Depending the orientation and multi edges properties.

**Return type** *GeoGraph*, *GeoDiGraph*, *GeoMultiGraph* or *GeoMultiDiGraph*

**`parse_nodes_attribute_as_wkt`**(*graph*, *attribute_name*)

Transform a graph nodes attribute from WKT to shapely objects. Attribute is replaced.

**Parameters**

- **`graph`** (*`nx.Graph :`*) – Graph to modify and parse
- **`attribute_name`** (*`str :`*) – Attribute to parse the nodes geometries

**See also:**

*`get_graph_with_wkt_geometry`*, *`parse_edges_attribute_as_wkt`*

**`parse_numpy_types`**(*gdf*)

Transform numpy types as scalar types.

**Parameters** **`gdf`** (*`gpd.GeoDataFrame :`*) – GeoDataFrame to modify

**`read_geofiles`**(*nodes_file_path*, *edges_file_path*, *directed=True*, *multigraph=False*, *node_index_attr='id'*, *edge_first_node_attr='u'*, *edge_second_node_attr='v'*)

Read geofiles to create a `GeoGraph`. Geofiles are read with `geopandas.read_file` method.

**Parameters**

- **`nodes_file_path`** (*`str`*) – File path of nodes.
- **`edges_file_path`** (*`str`*) – File path of edges.
- **`directed`** (*`bool`*) – If `True`, returns a directed graph. (Default value = True)
- **`multigraph`** (*`bool`*) – If `True`, returns a multigraph. (Default value = False)

- **node_index_attr** (*str*) – Node id attribute in the geofile for nodes labeling. (Default value = settings.NODE_ID_COLUMN_NAME)
- **edge_first_node_attr** (*str*) – Edge first node attribute in the geofile. (Default value = settings.EDGE_FIRST_NODE_COLUMN_NAME)
- **edge_second_node_attr** (*str*) – Edge second node attribute in the geofile. (Default value = settings.EDGE_SECOND_NODE_COLUMN_NAME)

**Returns** A parsed `Geograph`.

**Return type** *GeoGraph*, *GeoDiGraph*, *GeoMultiGraph*, *GeoMultiDiGraph*

**See also:**

`GeoGraph.add_nodes_from_gdf`, `GeoGraph.add_edges_from_gdf`, `geopandas.read_file`

**read_geograph_with_coordinates_attributes** (*graph*, *x_key='x'*, *y_key='y'*, ***attr*)
Parse a *networkx* graph which have node's coordinates as attribute. This method can be useful to parse an output graph of the *osmnx* package.

**Parameters**

- **graph** (*nx.Graph*) – Given graph to parse. All nodes must have the `x_key` and `y_key` attributes.
- **x_key** – x-coordinates attribute to parse (Default value = 'x')
- **y_key** – y-coordinates attribute to parse (Default value = 'y')
- ****attr** – Optional geograph spatial keys.

**Returns** The parsed geograph (shallow copy of the input graph).

**Return type** *GeoGraph*, *GeoDiGraph*, *GeoMultiGraph*, *GeoMultiDiGraph*

**Return type** `GeoGraph`

**read_gpickle** (*path*, ***attr*)
Read geograph object in Python pickle format.

**Parameters**

- **path** (*str*) – Path where to read a graph object.
- ****attr** – Potential spatial keys.

**Returns** The parsed geograph.

**Return type** *GeoGraph*, *GeoDiGraph*, *GeoMultiGraph* or *GeoMultiDiGraph*

**See also:**

*write_gpickle*, `nx.read_gpickle`, `nx.write_gpickle`

**read_graphml** (*path*, *node_type=<class 'str'>*, *edge_key_type=<class 'int'>*, ***attr*)
Read graph in GraphML format from path.

**Parameters**

- **path** – File path to the graphml file.
- **node_type** – See `nx.read_graphml` (Default value = str)
- **edge_key_type** – See `nx.read_graphml` (Default value = int)
- ****attr** – Potential spatial keys

**Returns** Parsed Geograph

**Return type** *GeoGraph*, *GeoDiGraph*, *GeoMultiGraph*, *GeoMultiDiGraph*

**See also:**

`write_graphml`, `nx.read_graphml`, `nx.write_graphml`

**Return type** `GeoGraph`

**`stringify_crs`**(*graph*)
Write the CRS attribute as a string.

**`stringify_unwritable_columns`**(*gdf*)
Transform elements which have type bool or list to string

**Parameters** **`gdf`** (`gpd.GeoDataFrame :`) – GeoDataFrame to modify

**`write_edges_to_geofile`**(*graph*, *file_name*, *driver='GPKG'*, *fiona_cast=True*)
Writes the edges of a geograph as a geographic file.

**Parameters**

- **`graph`** (`GeoGraph, GeoDiGraph, GeoMultiGraph, GeoMultiDiGraph`) – Graph to export
- **`file_name`** – File name (with path)
- **`driver`** – driver for export file format (GPKG, geojson, etc: can be found from `fiona.supported_drivers`) (Default value = "GPKG")
- **`fiona_cast`** – If true, methods for casting types to writable fiona types are used (Default value = True)

**See also:**

`write_geofile`, `write_nodes_to_geofile`

**`write_geofile`**(*graph*, *path='./'*, *nodes=True*, *edges=True*, *driver='GPKG'*, *fiona_cast=True*)
Export a networkx graph as a geographic files. Two files are generated: one for the nodes and one for the edges. The files names will be prefixed by the graph name and suffixed by "_edges" or "_nodes".

**Parameters**

- **`graph`** – Graph to export
- **`path`** – export directory (Default value = './')
- **`nodes`** – boolean to indicate whether export nodes or not. (Default value = True)
- **`edges`** – boolean to indicate whether export edges or not. (Default value = True)
- **`driver`** –

  **driver for export file format (GPKG, geojson, etc: can be found from `fiona.supported_drivers`)**
  (Default value = "GPKG")
- **`fiona_cast`** – If true, methods for casting types to writable fiona types are used (Default value = True)

**See also:**

`write_nodes_to_geofile`, `write_edges_to_geofile`

**`write_gpickle`**(*geograph*, *path*, *protocol=4*)
Write geograph object in Python pickle format.

**Parameters**

- **`geograph`** (`GeoGraph`, `GeoDiGraph`, `GeoMultiGraph`, `GeoMultiDiGraph`) – Geograph to write
- **`path`** – Path where to right the pickle file.
- **`protocol`** – See pickle protocols (Default value = pickle.HIGHEST_PROTOCOL).

**See also:**

*`read_gpickle`*, `nx.read_gpickle`, `nx.write_gpickle`

**`write_graphml`** (*geograph*, *path*, *encoding='utf-8'*, *prettyprint=True*, *infer_numeric_types=False*)
Generate GraphML file for the given geograph.

**Parameters**

- **`geograph`** (`GeoGraph`, `GeoDiGraph`, `GeoMultiGraph`, `GeoMultiDiGraph`) – Geograph to write
- **`path`** (*`str`*) – Writing file path
- **`encoding`** – See `nx.write_graphml` (Default value = 'utf-8')
- **`prettyprint`** – See `nx.write_graphml` (Default value = True)
- **`infer_numeric_types`** – See `nx.write_graphml` (Default value = False)

**See also:**

*`read_graphml`*, `nx.read_graphml`, `nx.write_graphml`

**`write_nodes_to_geofile`** (*graph*, *file_name*, *driver='GPKG'*, *fiona_cast=True*)
Writes the nodes of a geograph as a geographic file.

**Parameters**

- **`graph`** (`GeoGraph`, `GeoDiGraph`, `GeoMultiGraph`, `GeoMultiDiGraph`) – Graph to export
- **`file_name`** – File name (with path)
- **`driver`** – driver for export file format (GPKG, geojson, etc: can be found from `fiona.supported_drivers`) (Default value = "GPKG")
- **`fiona_cast`** – If true, methods for casting types to writable fiona types are used (Default value = True)

**See also:**

*`write_geofile`*, *`write_edges_to_geofile`*

### 1.6.6 Simplify

**`_clean_merge_mapping`** (*edge_mapping*, *new_edge*, *old_edges*, *directed*)
For the two-degree node merge operation, it cleans the new-old edges mapping dictionary by reporting original edges to the newest edge. It makes sure that all edges in the mapping dictionary dict are in the resulting graph.

**`get_dead_ends`** (*graph*, *node_filter=<function no_filter>*, *only_strict=False*)
Return the list of dead end in the given graph. A dead end is defined as a node having only one neighbor. For directed graphs, a strict dead end is a node having a unique predecessor and no successors. A weak dead end is a node having a unique predecessor that is also its unique successor.

**Parameters**

- **graph** (*nx.Graph*) – Graph to parse.
- **node_filter** – Evaluates to true if a node can be considered as dead end, false otherwise. (Default value = no_filter)
- **only_strict** – If true, remove only strict dead ends. Used only for directed graphs. (Default value = False)

**Returns** List of node name that are dead ends.

**Return type** list

**Return type** `list`

**remove_dead_ends**(*graph*, *node_filter=<function no_filter>*, *only_strict=False*)
Remove dead ends from a given graph. A dead end is defined as a node having only one neighbor. For directed graphs, a strict dead end is a node having a unique predecessor and no successors. A weak dead end is a node having a unique predecessor that is also its unique successor.

**Parameters**

- **graph** (*nx.Graph*) – Graph to simplify
- **node_filter** – Evaluates to true if a node can be removed, false otherwise. (Default value = no_filter)
- **only_strict** – If true, remove only strict dead ends. Used only for directed graphs. (Default value = False)

**remove_isolates**(*graph*)
Removes all isolates nodes in the given graph.

**Parameters** **graph** (*nx.Graph*) – A graph on which to remove all isolates

**Returns** Number of removed isolates

**Return type** int

**Return type** `int`

**remove_nan_attributes**(*graph*, *remove_nan=True*, *remove_none=True*, *copy=False*)
Remove the *nan* and *None* values from nodes and edges attributes.

**Parameters**

- **graph** (*nx.Graph*) – Graph (or subclass)
- **remove_nan** – If true, remove the *nan* values (test is `val is np.nan`) (Default value = True)
- **remove_none** – If true, remove the `None` values (test is `val is None`) (Default value = True)
- **copy** – If True, a copy of the graph is returned, otherwise the graph is modified inplace. (Default value = False)

**Returns** The modified graph if `copy` is true.

**Return type** None or nx.Graph

**remove_self_loop_edges**(*graph*)
Remove self loop edges on nodes of the given graph.

**Parameters** **graph** (*nx.Graph*) – A graph on which to remove all self loops.

**Returns** The number of removed self loops

**Return type** int

**Return type** `int`

**`remove_small_connected_components`**(*graph*, *minimum_allowed_size*)
Remove all connected components having strictly less than `minimum_allowed_size`.

**Parameters**

- **`graph`** (*`nx.Graph`*) – The graph on which to remove connected components
- **`minimum_allowed_size`** (*`int`*) – The minimum number of nodes where a connected component is kept.

**Returns** The number of removed connected components

**Return type** int

**Return type** `int`

**`trim_graph_with_polygon`**(*graph*, *polygon*, *as_view=False*, *method='intersects'*)
Trim a graph with a given polygon. Keep only the nodes that intersect (or are within) the polygon.

**Parameters**

- **`graph`** (`GeoGraph, GeoDiGraph, GeoMultiGraph` *`or`* `GeoMultiDiGraph`) – A GeoGraph (or subclass)
- **`polygon`** (*`Polygon or MultiPolygon`*) – A `shapely.Polygon` describing the area to keep
- **`as_view`** (*`bool`*) – If `True`, a view of the given graph is returned
- **`method`** (*`str`*) – If set to `"intersects"`, the `shapely.intersects` is used (keeps nodes and edges that intersects the polygon). If set to `"within"`, the `shapely.within` is used (keep nodes and edges that are strictly into the polygon). (Default value = "intersects")

**Returns** The modified graph if `as_view` is `True`.

**Return type** None or *GeoGraph*

**`two_degree_node_merge`**(*graph*, *node_filter=<function no_filter>*)
Merge edges that connects two nodes with a unique third node.

**Parameters**

- **`graph`** (`GeoGraph, GeoDiGraph, GeoMultiGraph` *`or`* `GeoMultiDiGraph`) – Graph to modify
- **`node_filter`** – Evaluates to true if a given node can be merged. (Default value = no_filter)

**Returns** Dictionary indicating for each new edge the merged ones.

**Return type** dict

**See also:**

*`two_degree_node_merge_for_directed_graphs`*, *`two_degree_node_merge_for_undirected_graphs`*

**Return type** `dict`

**`two_degree_node_merge_for_directed_graphs`**(*graph*, *node_filter=<function no_filter>*)
Merge edges that connects two nodes with a unique third node. A potential node to merge *n* must have exactly two different neighbors *u* and *v* with one of the following set of edges:

- *(u, n)* and *(n, v)*
- *(u, n)*, *(n, u)*, *(n, v)* and *(v, n)*

For the first case, a merging edge *(u, v)* is added. Under the latter, two edges *(u, v)* and *(v, u)* are added. The added edges will have a geometry corresponding to concatenation of the two replaced edges. If a replaced edge doesn't have a geometry, the added edge will not have a geometry as well. Edges geometries must be well ordered (first node must match with line's first extremity), otherwise lines concatenation may not be consistent (see `order_well_lines`).

**Parameters**

- **`graph`** (`GeoDiGraph` *or* `GeoMultiDiGraph`) – Given graph to modify
- **`node_filter`** – Evaluates to true if a given node can be merged. (Default value = no_filter)

**Returns** **merged_edges** – Dictionary indicating for each new edge the merged ones.

**Return type** dict

**`two_degree_node_merge_for_undirected_graphs`**(*graph*, *node_filter=<function no_filter>*)

Merge edges that connects two nodes with a unique third node for undirected graphs. Potential nodes to merge are nodes with two edges connecting two different nodes. If a replaced edge doesn't have a geometry, the added edge will not have a geometry as well.

**Parameters**

- **`graph`** (`GeoGraph` *or* `GeoMultiGraph`) – Graph to modify
- **`node_filter`** – Evaluates to true if a given node can be merged. (Default value = no_filter)

**Returns** Dictionary indicating for each new edge the merged ones.

**Return type** dict

**Return type** `dict`

CHAPTER

TWO

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX