GEOG 4/590: Geospatial Data Science

Lecture 3: Network data analysis

Email: jryan4@uoregon.edu
Office: 163A Condon Hall
Office hours: Monday 15:00-16:00 and Tuesday 14:00-15:00

In this lecture we will use the Python library `OSMnx` to download data from **OpenStreetMap (OSM)** and format it into a **graph** model. We will then use the `NetworkX` library to conduct network analysis.

- Which is the closest cafe?
- How many cafes can we walk to in 15 minutes?

Just remember that while we will be using network analysis on transport infrastructure, these principles apply to many other types of relational data such as international trade, character dialogue in films, or social media.

In this lecture we will use the Python library `OSMnx` to download data from **OpenStreetMap (OSM)** and format it into a **graph** model. We will then use the `NetworkX` library to conduct network analysis.

- Which is the closest cafe?
- How many cafes can we walk to in 15 minutes?

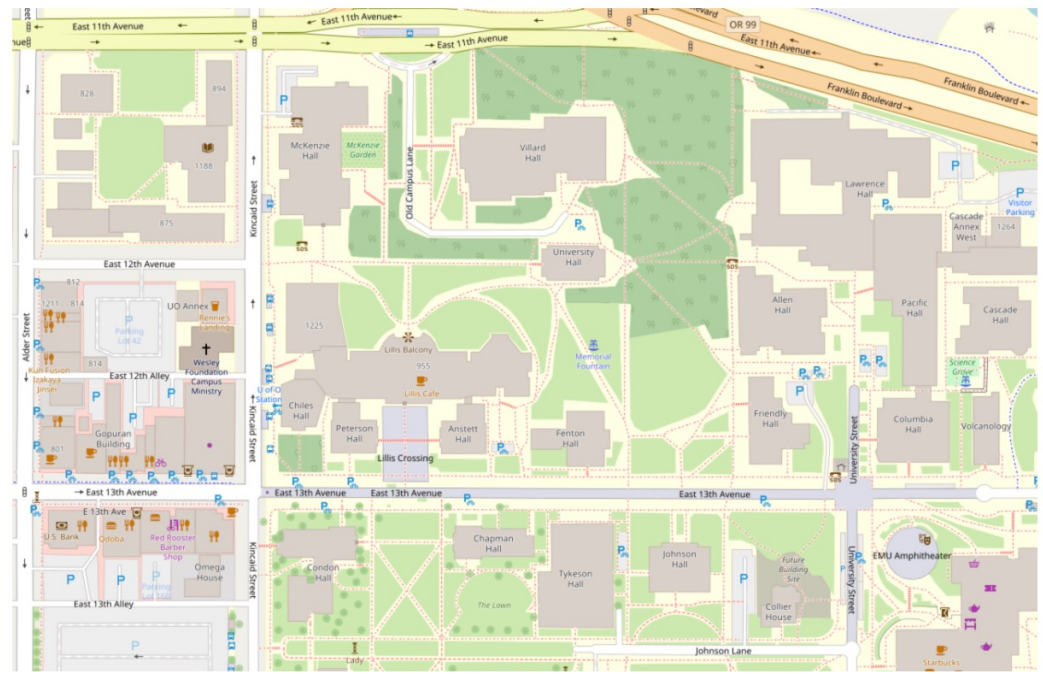
Just remember that while we will be using network analysis on transport infrastructure, these principles apply to many other types of relational data such as international trade, character dialogue in films, or social media.

```
import numpy as np
import geopandas as gpd

import osmnx as ox
import networkx as nx

import os
os.environ['USE_PYGEOS'] = '0'
from shapely.geometry import Point, LineString, Polygon, MultiPolygon, MultiLineString
from descartes import PolygonPatch

import matplotlib.pyplot as plt
import folium
```

OpenStreetMap

OpenStreetMap (OSM) is a project that is building a free map of the whole world. Thousands of members are teamed to create an accurate, detailed and up-to-date map that is as good or better than commercial products. Like Wikipedia, OpenStreetMap benefits from the ongoing input of thousands of people around the world.

- Everyone can contribute and add objects.
- More than eight million participant accounts with more than 16,000 (and rapidly growing) of those highly active members in multiple languages.
- Anyone can access the OSM map data for free, and it already is being used in many applications.
- There already is full coverage in most urban areas and data is used in many commercial applications.



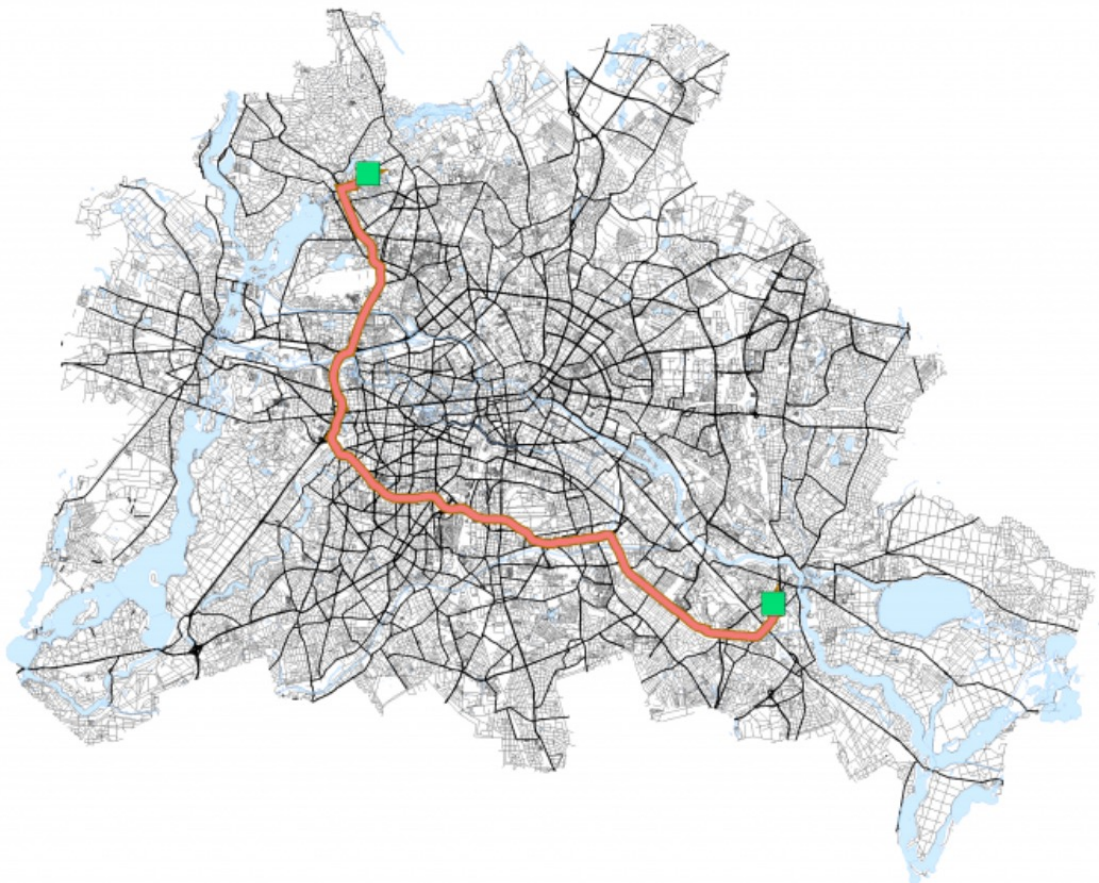
Network data

Most geospatial data is represented in one of two spatial models, **vector-based**, i.e. points, lines and polygons, or **raster-based**, i.e. cells of a continuous grid surface. The vector data model is **feature-oriented**, as it represents space as a series of discrete entities.

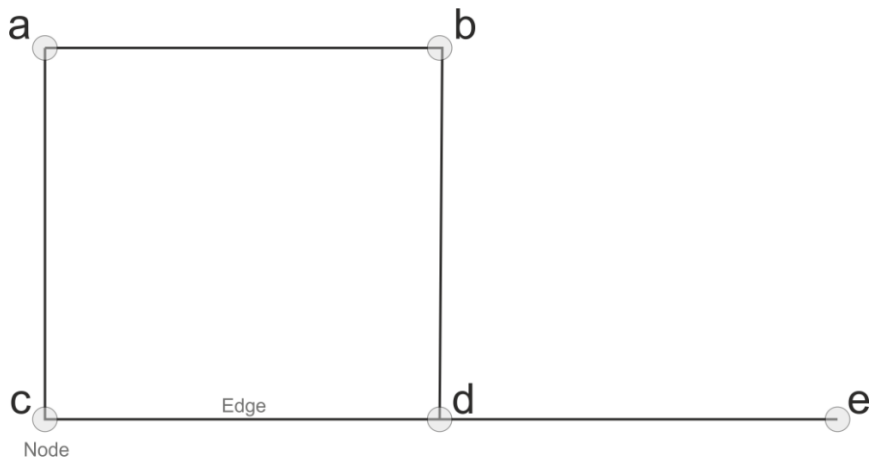
Network data represents **interconnections** between a set of features or entities. Almost everyone has needed a type of network analysis in their life.

For example:

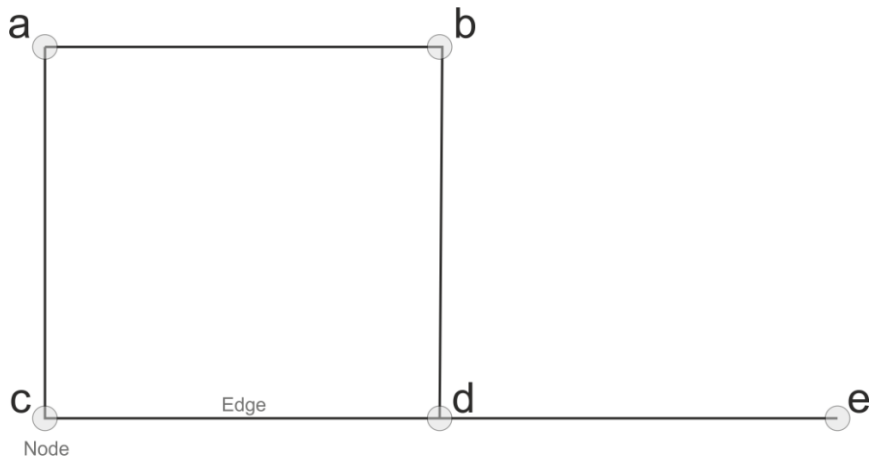
- What is the shortest route to the beach?
- Where should we build a hospital to best serve a community?
- How can I optimize a package delivery route?



Types of network data



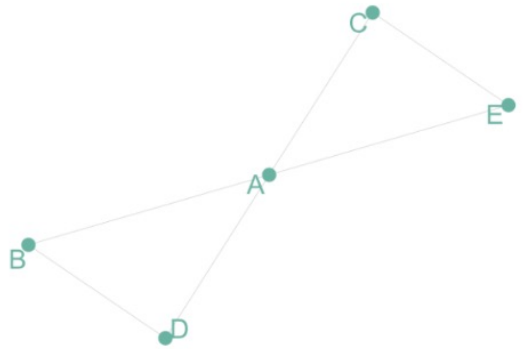
Types of network data



`Undirected and Unweighted`

Tom, Cherelle and Melanie live in the same house. They are connected but no direction and no weight.

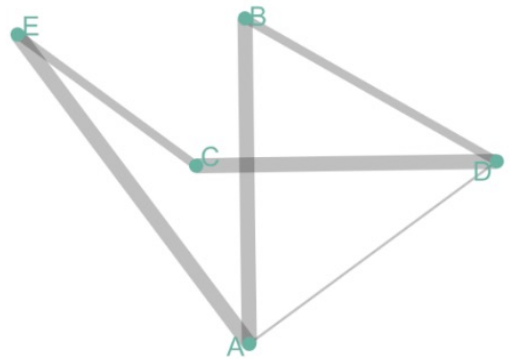
CODE



`Undirected and Weighted`

In the previous co-authors network, people are connected if they published a scientific paper together. The weight is the number of time it happend.

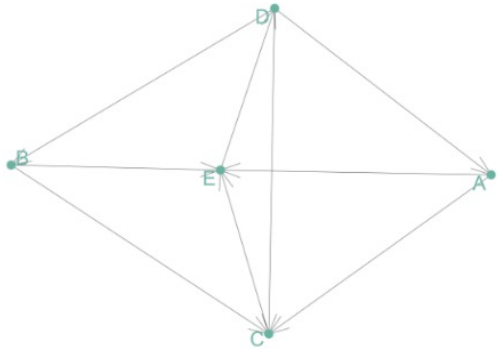
CODE



`Directed and Unweighted`

Tom follows Shirley on twitter, but the opposite is not necessarily true. The connection is unweighted: just connected or not.

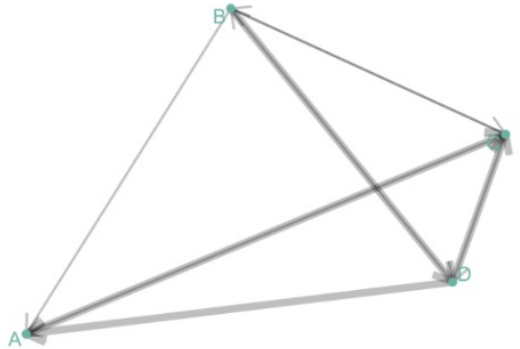
CODE



`Directed and Weighted`

People migrate from a country to another: the weight is the number of people, the direction is the destination.

CODE



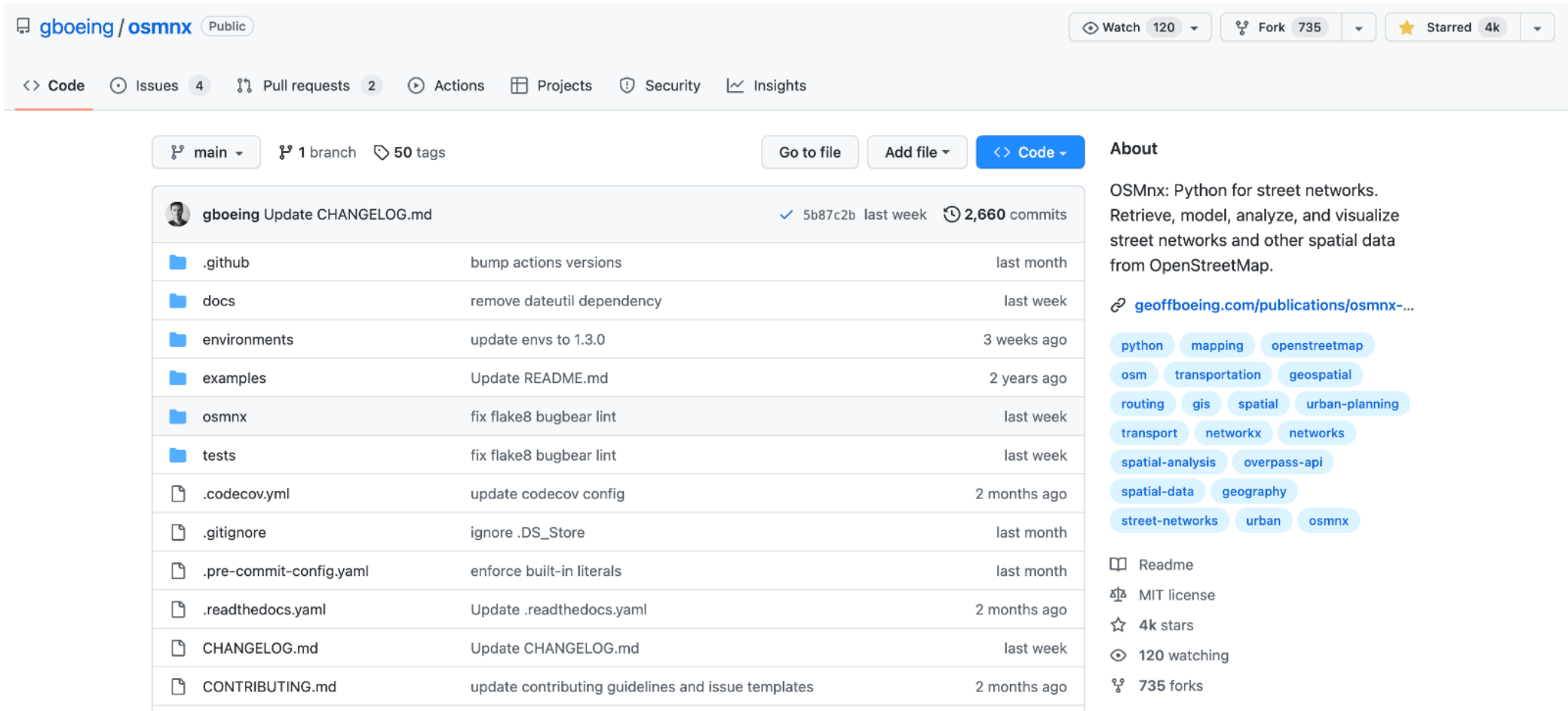
NetworkX

NetworkX
Network Analysis in Python

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

OSMnx

OSMnx is a Python package to retrieve, model, analyze, and visualize street **networks** from **OpenStreetMap**. Users can download and model walkable, drivable, or bikeable urban networks with a single line of Python code, and then easily analyze and visualize them.



Retrieve OSM data

First we will retrieve all features labeled (or tagged) as **buildings** in Eugene from OSM. Don't worry if you're not familiar with OSM or `OSMnx`, we will cover this topic next week in our data access lecture.

```
# Specify type of data
tags = {'building': True}

# Download building geometries from OSM
gdf = ox.geometries_from_place('Eugene, Oregon, USA', tags)
```

Retrieve OSM data

First we will retrieve all features labeled (or tagged) as **buildings** in Eugene from OSM. Don't worry if you're not familiar with OSM or `OSMnx`, we will cover this topic next week in our data access lecture.

```
# Specify type of data
tags = {'building': True}

# Download building geometries from OSM
gdf = ox.geometries_from_place('Eugene, Oregon, USA', tags)
```

This produces a large `GeoDataFrame` containing over 56,000 buildings.

```
print(gdf.shape)
```

```
(56025, 246)
```

Retrieve OSM data

For the purposes of this assignment, we are only interested in buildings that are tagged as **cafes**. Cafes are usually tagged as an **amenity** in OSM so we can filter them using a string comparison.

```
# Filter cafes
cafes = gdf[gdf['amenity'] == 'cafe'].reset_index()
print(cafes.shape)
```

```
(35, 248)
```

We find that there are 35 cafes in Eugene. This could well be an underestimate since there could be cafes that were not tagged as amenities when mapped. But let's keep moving forward and plot them.

Retrieve OSM data

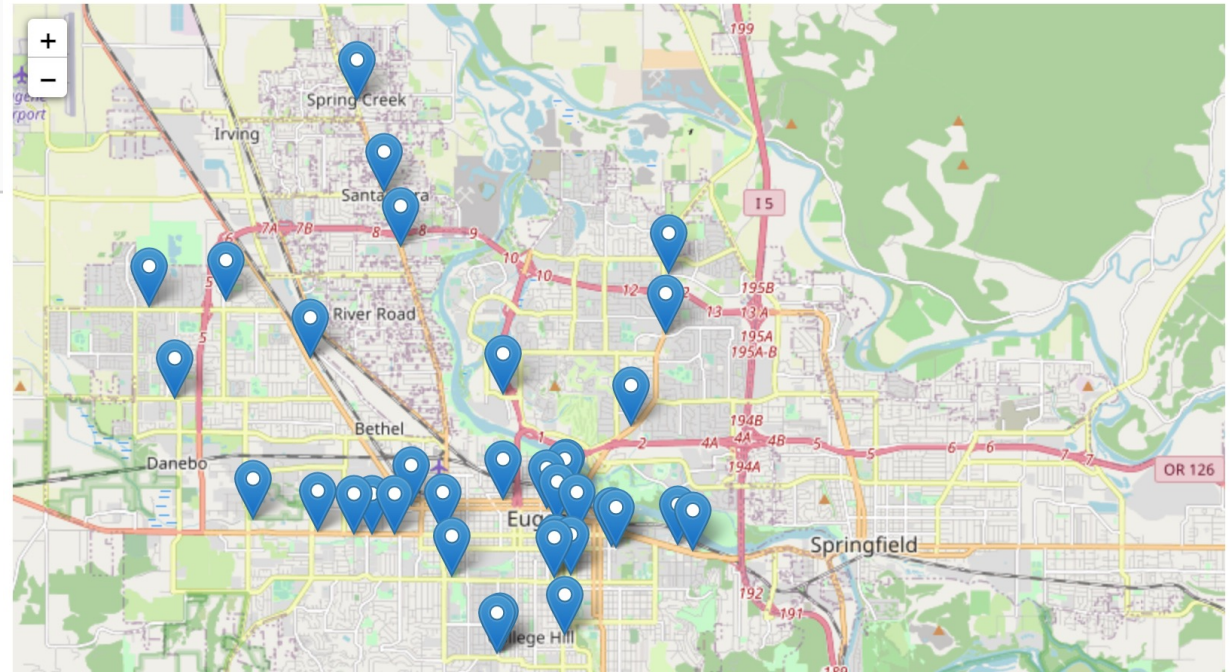
The cafes are acually **polygons** so we will compute their **centroids** to make it simpler to plot.

```
# Get cafe centroids
cafes['centroid'] = cafes['geometry'].apply(
  lambda x: x.centroid if type(x) == Polygon else (
  x.centroid if type(x) == MultiPolygon else x))
```

Retrieve OSM data

We can visulize interactively using `folium`. Again don't be too worried if you haven't used this library, we will cover it in a future lecture.

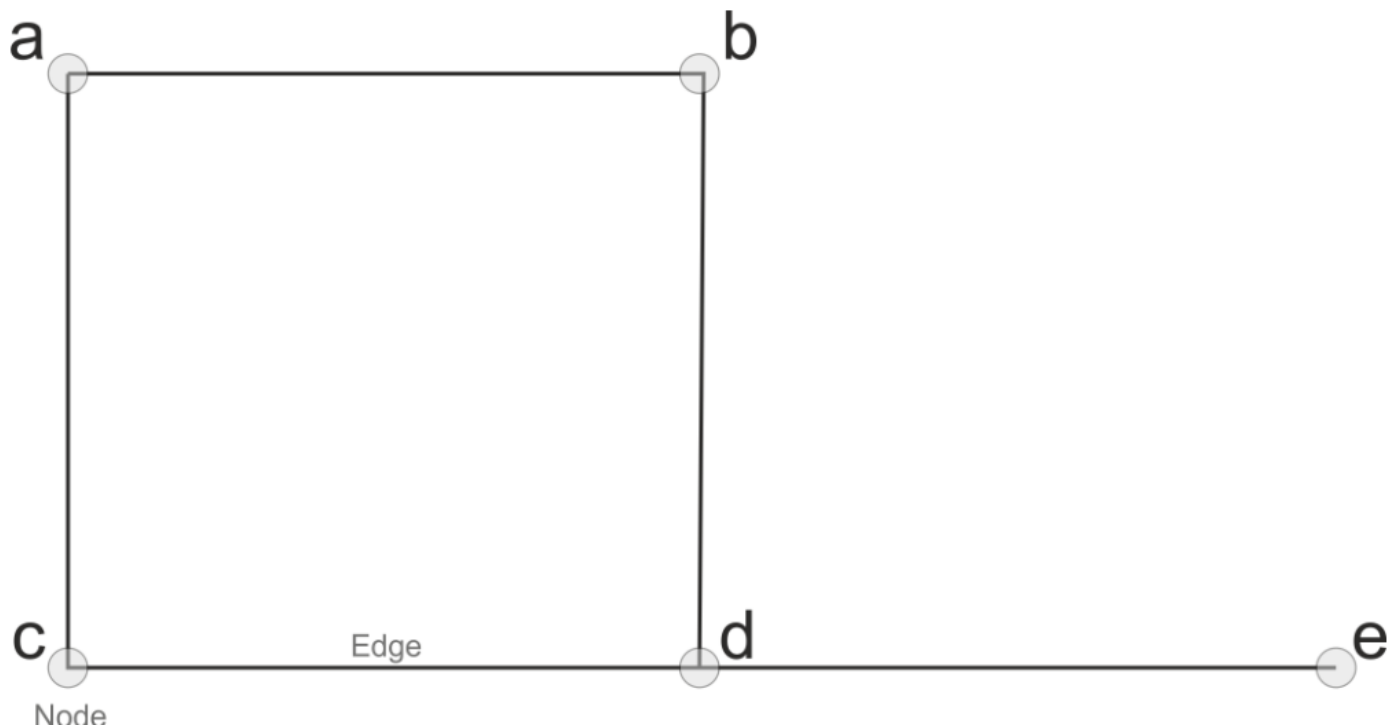
```
# Define center of map (i.e. Condon Hall) and initial zoom level
lat_lon = [44.0751, -123.0781]
m = folium.Map(location=lat_lon, zoom_start=12)

for i in range(0, cafes.shape[0]):
    my_string = cafes.iloc[i]['name']
    folium.Marker([cafes.iloc[i]['centroid'].y, cafes.iloc[i]['centroid'].x],
                  popup=my_string).add_to(m)

# Display map
m
```



Produce a graph (or network)

We will now produce a network using roads, sidewalks, and trails features in OSM. The `graph` module in `OSMnx` automatically process a network topology from raw OpenStreetMap data.

> **Note**
>
> A **network** is also known as a **graph** in mathematics.



Produce a graph (or network)

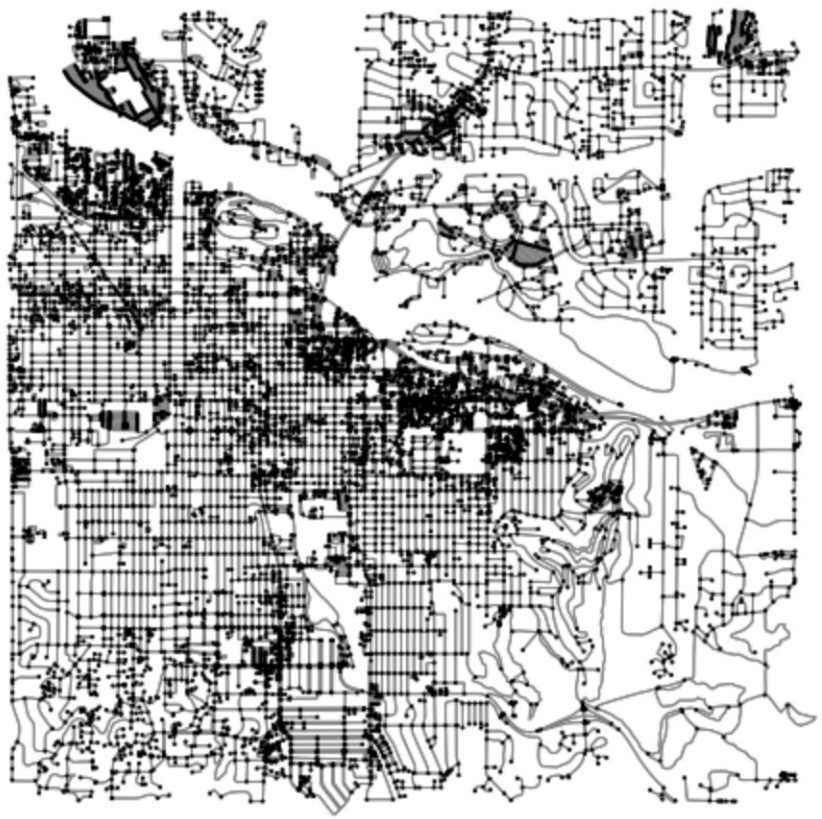
We will use the `graph_from_point` function which accepts a **point** (as lat/lon), a **distance** (in meters), and a **network type**. The options for **network type** are available in documentation and include `"all_private"`, `"all"`, `"bike"`, `"drive"`, `"drive_service"`, `"walk"`. We choose a distance of **2 miles** and the `walk` option since we are interested in cafes that are walkable.

```
# Define coordinates of Condon Hall
lat_lon = (44.0451, -123.0781)

# Define walkable street network 3.2 km around Condon Hall
g = ox.graph_from_point(lat_lon, dist=3200, network_type='walk')
```

Produce a graph (or network)

Let's see what the graph looks like...

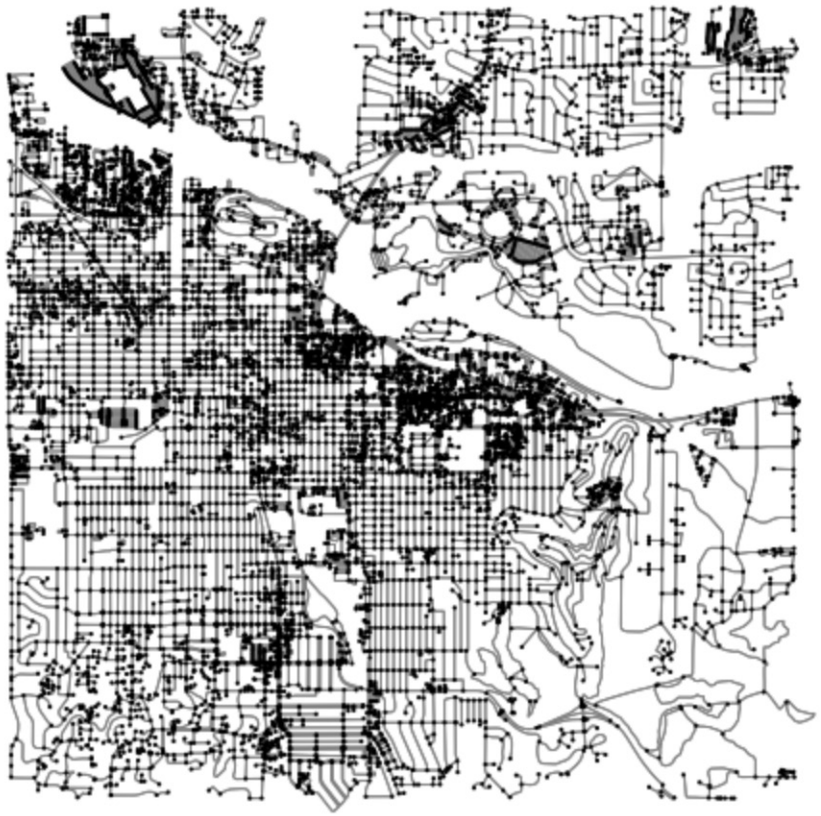
```
# Plot map
fig, ax = ox.plot_graph(g, bgcolor='white', node_color='black', edge_color='grey', node_size=5
```

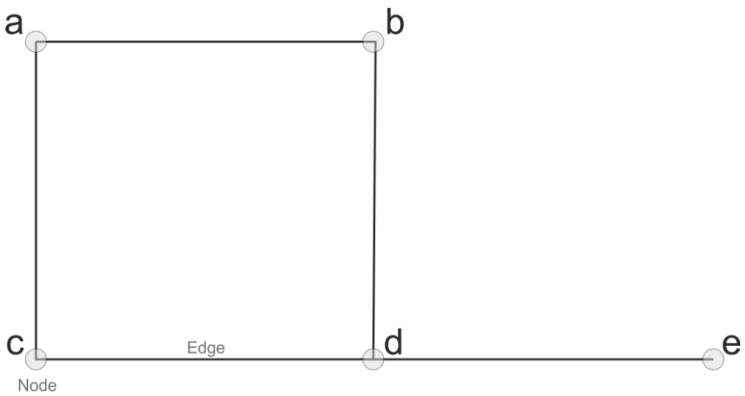


Produce a graph (or network)

The first thing to note is that th network structure consists of **nodes** (dots) and **edges** (lines). Nodes represent objects while edges represent the relationship between those objects.

Since our network represents transport infrastructure, nodes represent intersections/dead-ends and edges represent the street segments that link them. If we were studying social relationships between Facebook users, nodes would represent users and edges would represent relationships between them (e.g. friendships or group memberships).





Produce a graph (or network)

Also note that the graph is `MultiDiGraph` `NetworkX` object. `Multi` means that multiple edges are allowed between any pair of nodes. `Di` stands for **directed** which means that all our edges are directional. Bidirectional streets are therefore represented with **two edges** (with identical geometries): one from node 1 to node 2 and another from 2 to 1, to represent both possible directions of flow.

```
type(g)
```

```
networkx.classes.multidigraph.MultiDiGraph
```

Reproject network

Once we have produced our graph (or network, it is good practice to **reproject** it to UTM coorindates so we can work in SI units (i.e. meters) instead of degrees. The `graph_project` function can be used to reproject the graph. The docs note that if the `to_crs` argument is set to `None`, the graph is projected to the UTM coordinate system for the UTM zone in which the graph's centroid lies.

```
# Convert to graph
graph_proj = ox.project_graph(g, to_crs=None)
```

Define points of interest

Next we define the centroid locations of Condon Hall and our cafes, making sure that they are in the same projection as our graph (i.e. UTM Zone 10N or EPSG:32610).

Define points of interest

Next we define the centroid locations of Condon Hall and our cafes, making sure that they are in the same projection as our graph (i.e. UTM Zone 10N or EPSG:32610).

We can find Condon Hall in our original OSM building data using string matching.

```
# Get coordinates of Condon Hall
condon_hall = gdf[gdf['name'] == 'Condon Hall'].reset_index()

# Reproject to UTM Zone 10N
condon_hall = condon_hall.to_crs('EPSG:32610')
cafes = cafes.to_crs('EPSG:32610')
```

Define points of interest

Note

To begin with we will only compute the shortest distance between Condon Hall and the **first** cafe in our `GeoDataFrame`.

```
# Get x and y coordinates of Condon Hall
orig_xy = [condon_hall['centroid'].y.values[0], condon_hall['centroid'].x.values[0]]

# Get x and y coordinates of the first cafe
target_xy = [cafes['centroid'].y.values[0], cafes['centroid'].x.values[0]]
```

Find the shortest path between points

The **`distance`** module in `OSMnx` contains functions for calculating distances, shortest paths, and finding nearest node/edge(s) to point(s). Since the coordinates of our points are unlikely to exactly align with one of the nodes, we first have to find the nearest node to our points using the `nearest_nodes` function.

```
# Find the node in the graph that is closest to the origin point
orig_node = ox.distance.nearest_nodes(graph_proj, X=orig_xy[1], Y=orig_xy[0], return_dist=Fals

# Find the node in the graph that is closest to the target point
target_node = ox.distance.nearest_nodes(graph_proj, X=target_xy[1], Y=target_xy[0], return_dis
```

Find the shortest path between points

Now we can compute the shortest distance between our two points using the **`shortest_path`** functions available from `NetworkX`. The **`shortest_path`** function returns a list of nodes along the shortest path, and the **`shortest_path_length`** function returns the length of this path.

Find the shortest path between points

Now we can compute the shortest distance between our two points using the **`shortest_path`** functions available from `NetworkX`. The **`shortest_path`** function returns a list of nodes along the shortest path, and the **`shortest_path_length`** function returns the length of this path.

> **Note**
>
> The `weight` argument defines the edge attribute to minimize when solving shortest path. In our case, we would like the **shortest distance** so we choose `length`.

```
# Calculate the shortest path
route = nx.shortest_path(G=graph_proj, source=orig_node, target=target_node, weight='length')
length = nx.shortest_path_length(G=graph_proj, source=orig_node, target=target_node, weight='l
```

Find the shortest path between points

Now we can compute the shortest distance between our two points using the **`shortest_path`** functions available from `NetworkX`. The **`shortest_path`** function returns a list of nodes along the shortest path, and the **`shortest_path_length`** function returns the length of this path.

> **Note**
>
> The `weight` argument defines the edge attribute to minimize when solving shortest path. In our case, we would like the **shortest distance** so we choose `length`.

```
# Calculate the shortest path
route = nx.shortest_path(G=graph_proj, source=orig_node, target=target_node, weight='length')
length = nx.shortest_path_length(G=graph_proj, source=orig_node, target=target_node, weight='l
```

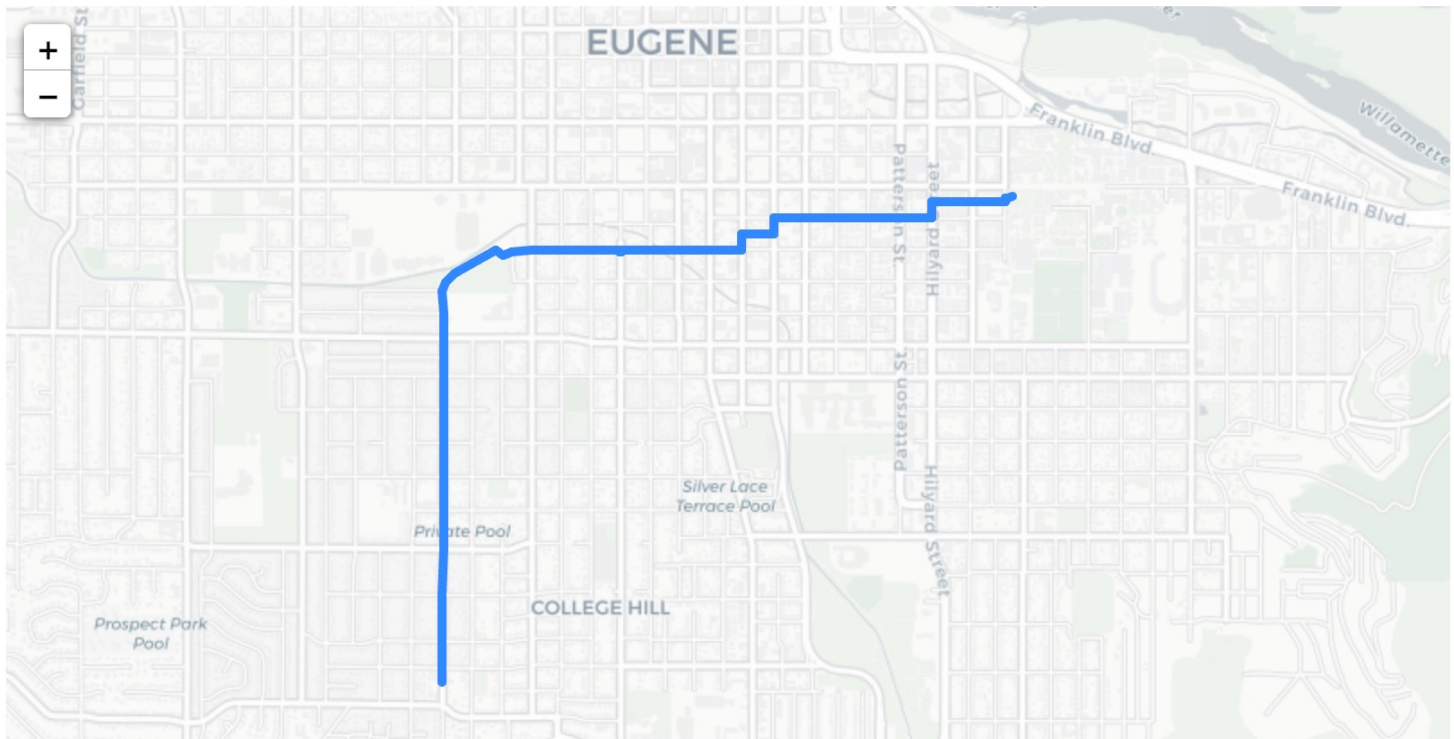
```
print("Shortest path distance = {t:.1f} km.".format(t=length/1000))
```

```
Shortest path distance = 4.0 km.
```

Find the shortest path between points

The `route` variable contains a list of the nodes constituting the shortest path between the two points. It can plotted using the `plot_route_folium` function.

```
# Plot the shortest path using folium
m = ox.plot_route_folium(g, route, weight=5)
m
```



Isochrones

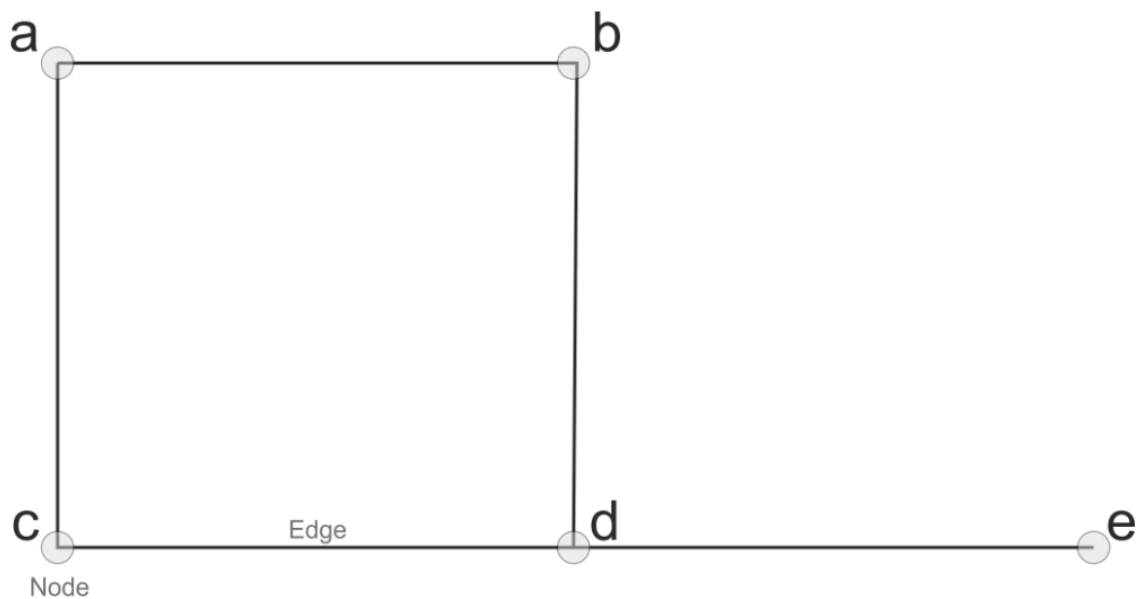
Another way to approach this problem would be to find all the **buildings** that we would be prepared to walk to in a **given time period**, and then find all buildings labeled as cafes. This may be a more logical approach since we don't have to rely on finding all the cafes at the start of the analysis - and we know some are likely to be missing. We can also visualize our results using an **isochrone map**, a popular type of map where each color represents an area accessible from a point within a certain time threshold.

Edge and node attributes

To conduct this analysis, it is useful to spend a little more time thinking about the attributes of **nodes** and **edges** of the `MultiDiGraph` object that is produced by `OSMnx`.

```
type(graph_proj)
```

```
networkx.classes.multidigraph.MultiDiGraph
```



Edge and node attributes

To conduct this analysis, it is useful to spend a little more time thinking about the attributes of **nodes** and **edges** of the `MultiDiGraph` object that is produced by `OSMnx`.

```
type(graph_proj)
```

```
networkx.classes.multidigraph.MultiDiGraph
```

We can explore the **edge** and **node** attributes easier if we convert the `MultiDiGraph` to a `GeoDataFrame`. We'll have a look at the nodes first since they are a bit more intuitive. In our infrastructure network, nodes represent intersections/dead-ends,

```
nodes = ox.graph_to_gdfs(graph_proj, nodes=True, edges=False)
edges = ox.graph_to_gdfs(graph_proj, nodes=False, edges=True)
```

Nodes

```
nodes.head()
```

osmid	y	x	street_count	lon	lat	highway	geometry
39649644	4.878377e+06	492202.469421	3	-123.097353	44.058520	NaN	POINT (492202.469 4878377.157)
9127143342	4.878355e+06	492209.299667	3	-123.097268	44.058322	NaN	POINT (492209.300 4878355.202)
7391423647	4.878469e+06	492202.385688	3	-123.097356	44.059347	NaN	POINT (492202.386 4878468.978)
39649646	4.878275e+06	492203.254169	3	-123.097342	44.057603	NaN	POINT (492203.254 4878275.306)
7391423890	4.878276e+06	492156.438857	3	-123.097926	44.057611	NaN	POINT (492156.439 4878276.283)

osmid	y	x	street_count	lon	lat	highway	geometry
39649644	4.878377e+06	492202.469421	3	-123.097353	44.058520	NaN	POINT (492202.469 4878377.157)

- The **nodes** `GeoDataFrame` is indexed by an `osmid` which provides a unique identifier for each node.
- We then have `y` and `x` columns that represent location in **UTM Zone 10 N** coordinates (since we reprojected earlier). The `geometry` column also looks to be in UTM Zone 10 N so we know we are working in meters, not degrees.

osmid	y	x	street_count	lon	lat	highway	geometry
39649644	4.878377e+06	492202.469421	3	-123.097353	44.058520	NaN	POINT (492202.469 4878377.157)

- The **nodes** `GeoDataFrame` is indexed by an `osmid` which provides a unique identifier for each node.
- We then have `y` and `x` columns that represent location in **UTM Zone 10 N** coordinates (since we reprojected earlier). The `geometry` column also looks to be in UTM Zone 10 N so we know we are working in meters, not degrees.
- The `street_count` column provides the number of street segments connected to each node. The `highway` column provides the type of intersection.

```
nodes['highway'].unique()
```

```
array([nan, 'traffic_signals', 'crossing', 'turning_circle',
       'motorway_junction', 'stop', 'turning_loop'], dtype=object)
```

Edges

Edges represent street segments and there are a few more columns to interpret.

```
edges.head()
```

u	v	key	osmid	lanes	name	highway	oneway	length	geometry	service	access	ref	maxspeed	bridge	width	junction	tunnel
39649644	**9127143342**	**0**	713160022	2	Skinners Butte Loop	unclassified	False	23.012	LINESTRING (492202.469 4878377.157, 492209.300...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	7391423647	**0**	713160022	2	Skinners Butte Loop	unclassified	False	92.181	LINESTRING (492202.469 4878377.157, 492199.636...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	39649646	**0**	1054883373	NaN	Lincoln Street	residential	False	102.018	LINESTRING (492202.469 4878377.157, 492201.283...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9127143342	**39649644**	**0**	713160022	2	Skinners Butte Loop	unclassified	False	23.012	LINESTRING (492209.300 4878355.202, 492202.469...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	9127143339	**0**	987440284	NaN	NaN	service	False	13.919	LINESTRING (492209.300 4878355.202, 492223.103...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Edges

u	v	key	osmid	lanes	name	highway	oneway	length	geometry	service	access	ref	maxspeed	bridge	width	junction	tunnel
39649644	**9127143342**	**0**	713160022	2	Skinners Butte Loop	unclassified	False	23.012	LINESTRING (492202.469 4878377.157, 492209.300...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The **edges** `GeoDataFrame` is indexed by `u` and `v` which represent the start and end nodes for the edge. The `key` attribute differentiates between parallel edges. In other words, two edges that are between the same two nodes but differ in other attributes.
- Each edge has an `osmid` but, in this framework, it's more convenient to work with the nodes at either end of the edge.
- The columns `name`, `oneway`, `length` (in meters), `geometry`, `bridge`, `width`, `junction`, and `tunnel` are self-explanatory.
- The `highway` column is provides the type of street segment (e.g. residential, footway, path).

Edges

u	v	key	osmid	lanes	name	highway	oneway	length	geometry	service	access	ref	maxspeed	bridge	width	junction	tunnel
39649644	**9127143342**	**0**	713160022	2	Skinners Butte Loop	unclassified	False	23.012	LINESTRING (492202.469 4878377.157, 492209.300...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- The `access` column is represents accessibility of the street.

```
edges['access'].unique()
```

```
array([nan, 'yes', 'customers', 'destination', 'no', 'agricultural',
       'permissive'], dtype=object)
```

Add an edge attribute

To compute the distance we can travel in a given time period, we need to know the time taken to walk along each street segment or edge. Since we know the length of each edge and our travel speed, we can compute this attribute like so:

```
travel_speed = 5
meters_per_minute = travel_speed * 1000 / 60  # km per hour to m per minute

for u, v, data in graph_proj.edges.data():
    data['time'] = data['length'] / meters_per_minute
```

Add an edge attribute

Now when we print a single value from our edges dataset, we have a new attribute called `time` which is the number of minutes to travel along the edge.

```
list(graph_proj.edges.data())[0]
```

```
(39649644,
 9127143342,
 {'osmid': 713160022,
  'lanes': '2',
  'name': 'Skinners Butte Loop',
  'highway': 'unclassified',
  'oneway': False,
  'length': 23.012,
  'geometry': <shapely.geometry.linestring.LineString at 0x7fcf7153d520>,
  'time': 0.276144})
```

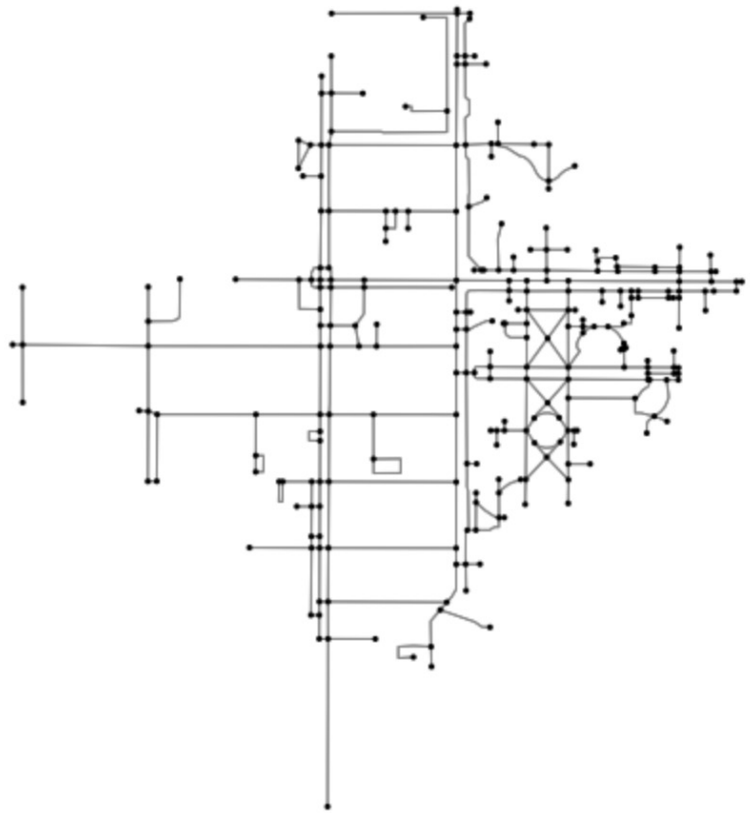
Find all nodes within given radius

We can find all the nodes within a given radius using the `ego_graph` function in `NetworkX`. The first two arguments include the original graph (i.e. our Eugene network) and a single node which, in our case, is Condon Hall. The last two arguments are the `radius` within which to include neighboring nodes and `distance` which detemines the edge attribute to use. We will use the `time` attribute we just produced but note that we could also use `length` if we were interested only in distance.

Find all nodes within given radius

As a test we will set `radius` to **10** which is equivalent to 10 **minutes** of traveling by foot. But note that if `distance` was set to `length`, this would be 10 **meters** of traveling by foot (i.e. not very far).

```
subgraph = nx.ego_graph(graph_proj, orig_node, radius=10, distance="time")
fig, ax = ox.plot_graph(subgraph, bgcolor='white', node_color='black', edge_color='grey', node
```



Produce isochrone map

We now have all the components to calculate the nodes that are within certain walking times from Condon Hall. We will use walking times of 5 to 45 minutes in 5-minute intervals.

```
trip_times = [5, 10, 15, 20, 25, 30, 35, 40, 45]  # in minutes
```

Next, we will define a color for each trip time (or isochrone).

```
# get one color for each isochrone
iso_colors = ox.plot.get_colors(n=len(trip_times), cmap="plasma", start=0, return_hex=True)
```

Produce isochrone map

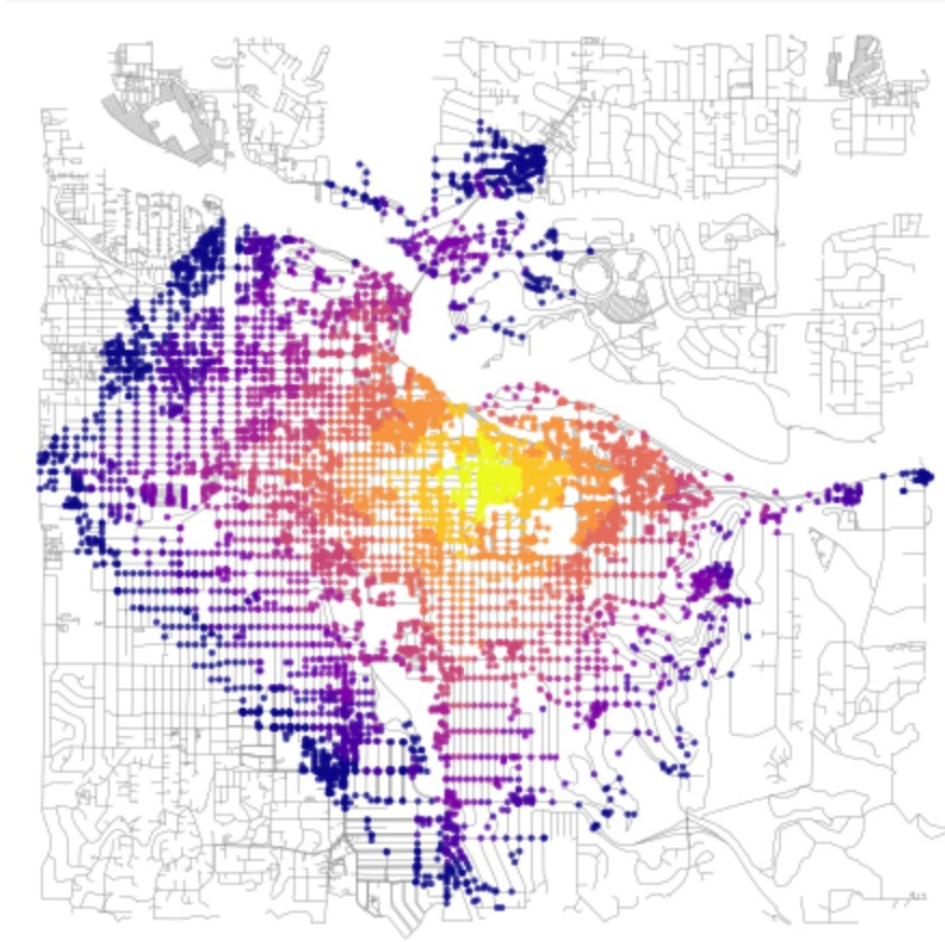
This next bit gets a little hairy but basically we will assign each node a color based on the travel time from Condon Hall. We found this code from here. We're not sure we could have written this ourselves but it's perfectly OK to appropriate someone else's code if it works.

```
# color the nodes according to isochrone then plot the street network
node_colors = {}
for trip_time, color in zip(sorted(trip_times, reverse=True), iso_colors):
    subgraph = nx.ego_graph(graph_proj, orig_node, radius=trip_time, distance="time")
    for node in subgraph.nodes():
        node_colors[node] = color

nc = [node_colors[node] if node in node_colors else "none" for node in graph_proj.nodes()]
ns = [10 if node in node_colors else 0 for node in graph_proj.nodes()]
```

Produce isochrone map

Now we can plot our colored nodes onto the original graph.

```
fig, ax = ox.plot_graph(graph_proj, node_color=nc, node_size=ns, node_alpha=0.8,
    edge_linewidth=0.2, edge_color="grey", bgcolor='white')
```



Isochrone polygons

Again, we found this code from here, don't worry if you don't understand it. We first define a function to make the isochrone polygons from positions of the nodes.

```
def make_iso_polys(G, edge_buff=25, node_buff=50, infill=False):
    isochrone_polys = []
    for trip_time in sorted(trip_times, reverse=True):
        subgraph = nx.ego_graph(G, orig_node, radius=trip_time, distance='time')

        node_points = [Point((data['x'], data['y'])) for node, data in subgraph.nodes(data=Tru
        nodes_gdf = gpd.GeoDataFrame({'id': subgraph.nodes()}, geometry=node_points)
        nodes_gdf = nodes_gdf.set_index('id')

        edge_lines = []
        for n_fr, n_to in subgraph.edges():
            f = nodes_gdf.loc[n_fr].geometry
            t = nodes_gdf.loc[n_to].geometry
            edge_lookup = G.get_edge_data(n_fr, n_to)[0].get('geometry',  LineString([f,t]))
            edge_lines.append(edge_lookup)

        n = nodes_gdf.buffer(node_buff).geometry
        e = gpd.GeoSeries(edge_lines).buffer(edge_buff).geometry
        all_gs = list(n) + list(e)
        new_iso = gpd.GeoSeries(all_gs).unary_union

        # try to fill in surrounded areas so shapes will appear solid and blocks without white
        if infill:
            new_iso = Polygon(new_iso.exterior)
        isochrone_polys.append(new_iso)
    return isochrone_polys
```

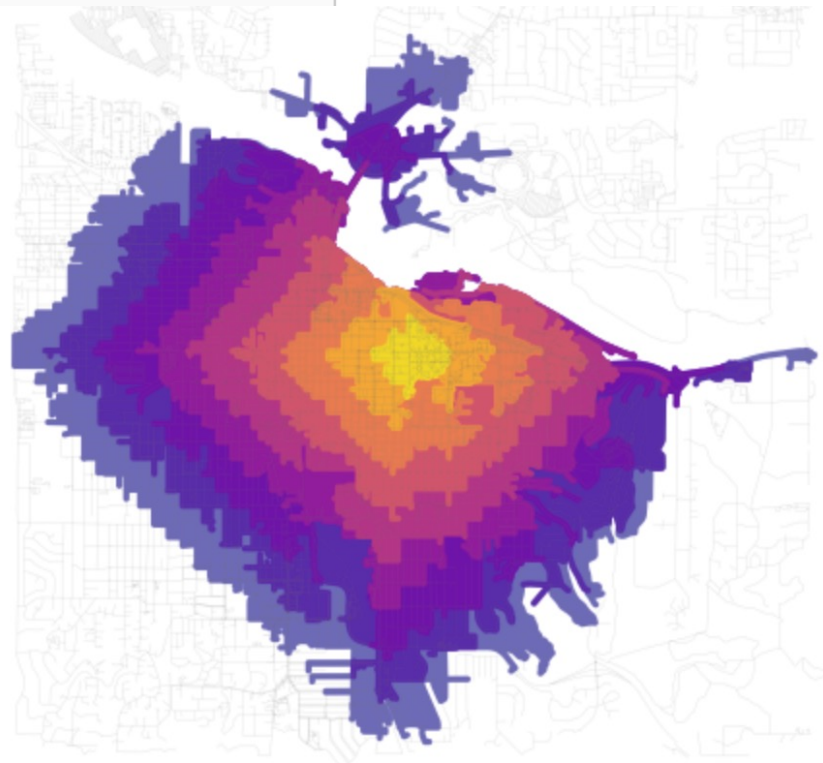
Isochrone polygons

Next we will call the function...

```
isochrone_polys = make_iso_polys(graph_proj, edge_buff=25, node_buff=0, infill=True)
```

And plot!

```
fig, ax = ox.plot_graph(graph_proj, show=False, close=False, edge_linewidth=0.2,
                        edge_color="grey", bgcolor='white', edge_alpha=0.2, node_size=0)
for polygon, fc in zip(isochrone_polys, iso_colors):
    patch = PolygonPatch(polygon, fc=fc, ec='none', alpha=0.6, zorder=-1)
    ax.add_patch(patch)
plt.show()
```

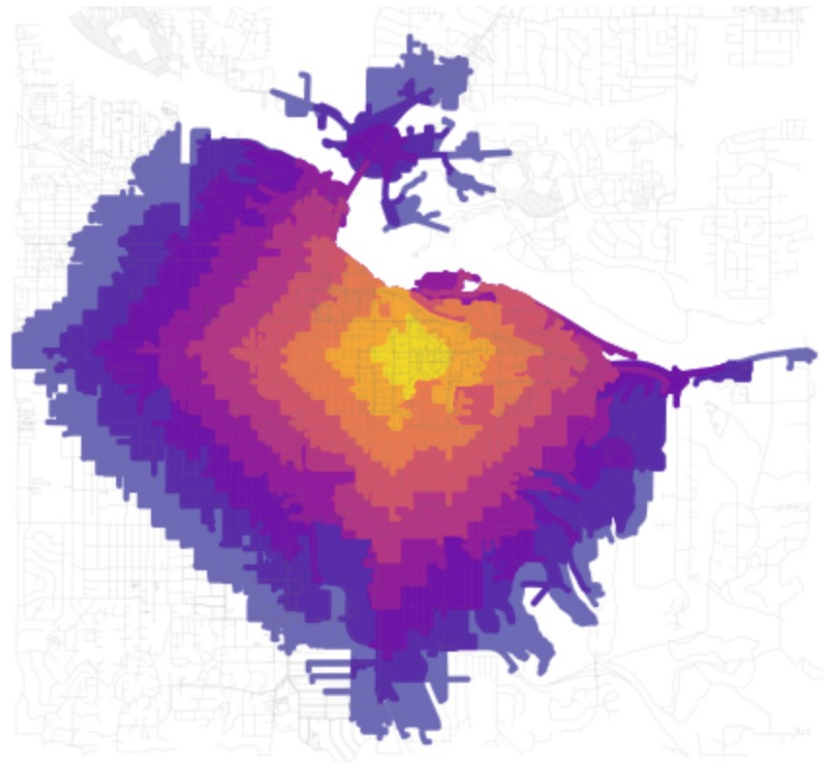


Isochrone polygons

Finally, let's intersect these polygons with our cafes to see which are within a given travel time. The `isochrone_polys` are actually in reverse order so the first polygon represents the longest travel time (i.e. 45 minutes) and the last polygon represents the shortest travel time (i.e. 5 minutes).

So to find which cafes are within 20 minutes of walking we would use the **fourth last** item in the index (i.e. 5, 10, 15, **20**). Remember that if `-1` is the last item in the index, `-4` is the fourth last item.

```
cafes['20-minutes'] = cafes.intersects(isochrone_polys[-4])
```



Isochrone polygons

Finally, let's intersect these polygons with our cafes to see which are within a given travel time. The `isochrone_polys` are actually in reverse order so the first polygon represents the longest travel time (i.e. 45 minutes) and the last polygon represents the shortest travel time (i.e. 5 minutes).

So to find which cafes are within 20 minutes of walking we would use the **fourth last** item in the index (i.e. 5, 10, 15, **20**). Remember that if `-1` is the last item in the index, `-4` is the fourth last item.

```
cafes['20-minutes'] = cafes.intersects(isochrone_polys[-4])
```

```
cafes[['name', 'time_minutes', 'distance_km']][cafes['20-minutes'] == True]
```

	name	time_minutes	distance_km
4	Starbucks	16.232267	1.217420
5	Espresso Roma	2.155133	0.161635
6	Dutch Bros. Coffee	20.103613	1.507771
7	Greenleaf Juicing Company	3.298507	0.247388
10	Dutch Bros. Coffee	14.359920	1.076994

Next time: Gridded data analysis

Email: jryan4@uoregon.edu
Office: 163A Condon Hall
Office hours: Monday 15:00-16:00 and Tuesday 14:00-15:00