## 2.8.3 公共交通复合网络构建

### 2.8.3.1 数据预处理

南京（32°4’N，118°47’E）为中国东部地区重要的中心城市，江苏省省会，南京都市圈核心城市，常住人口850万人。公共交通网络（图1）包括地铁系统和公交系统，其地铁线路长度居中国第6位、世界第7位，10条线路覆盖全市13个市辖区县，包含站点348个；其公交线路1 304条（往返），包含站点32 660个，日均客流量约200万人次。南京市公共交通线路和站点数据（2020年数据），以及人口分布数据（425×425 m间隔的采样点，2020年数据）均来源于中国专业IT社区[CSDN（Chinese Software Developer Network）](https://www.csdn.net/)①的开放数据。其中公共交通（地铁和公交）的站点数据包括站点名称和ID、所属线路名称和ID、经纬度坐标等；线路数据包括线路名称和ID、几何对象等。人口分布数据包括采样点经纬度坐标、人口数等。根据南京市绿化园林局发布的《南京市绿地系统规划》中心城区规划综合公园一览表中确定保留和扩建的53个综合公园作为研究对象，在Google Earth中绘制各个公园范围边界。

参数管理使用AttrDict()方法（具体查看“Cityscapes数据集——参数管理”一节）。db子属性存储数据库信息；gi子属性存储地理坐标投影信息；data子属性存储数据文件路径。

from database import postSQL2gpd,gpd2postSQL,df2postSQL,postSQL2df  
from util\_misc import AttrDict  
import warnings  
warnings.filterwarnings('ignore')  
  
\_\_C=AttrDict()   
args=\_\_C  
  
\_\_C.db=AttrDict()   
\_\_C.db.UN='postgres'  
\_\_C.db.PW='123456'  
\_\_C.db.DB='public\_transportation'  
\_\_C.db.GC='geometry'   
\_\_C.db.db\_info=dict(geom\_col=args.db.GC,myusername=args.db.UN,mypassword=args.db.PW,mydatabase=args.db.DB)  
  
\_\_C.gi=AttrDict()  
\_\_C.gi.nanjing\_epsg=32650  
\_\_C.gi.beijing\_epsg=32750  
\_\_C.gi.epsg\_wgs84=4326  
  
\_\_C.data=AttrDict()  
\_\_C.data.bus\_routes='./data/nanjing\_bus\_route\_and\_station/bus\_routes.shp' # 公交站  
\_\_C.data.bus\_stations='./data/nanjing\_bus\_route\_and\_station/bus\_stations.shp' # 公交线路  
\_\_C.data.subway\_stations='./data/nanjing\_subway\_station\_and\_line/subway\_stations.shp' # 地铁站  
\_\_C.data.subway\_lines='./data/nanjing\_subway\_station\_and\_line/subway\_lines.shp' # 地铁线路  
\_\_C.data.region='./data/nanjing\_region/region.shp' # 南京市域边界  
\_\_C.data.administrative\_districts='./data/nanjing\_administrative\_districts/nanjing\_administrative\_districts.shp' # 南京行政区划  
\_\_C.data.population='./data/nanjing\_population/population.shp' # 南京人口分布  
\_\_C.data.comprehensive\_park='./data/NanjingParks.kml' # 南京公园  
\_\_C.data.geolife='G:\data\Geolife' # Geolife GPS trajectory 数据集（下载的原始数据）  
\_\_C.data.geolife\_gdf='E:\data\geolife.gpkg' # Geolife GPS trajectory 数据集（合并后存储的数据对象）

#### 1) 公交和地铁站点及路径数据读写

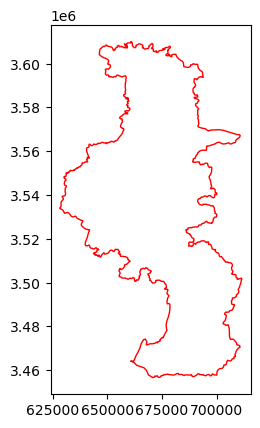
* 南京省市边界读写

将“不平等性和空间隔离”一章中定义的shp2gdf()函数置于模块database中调用读取SHP数据格式文件为GeoDataFrame格式。

from database import shp2gdf  
  
region=shp2gdf(args.data.region,epsg=args.gi.nanjing\_epsg)  
region.plot(color='none', edgecolor='red');  
region

original data info:(1, 3)  
dropna-how=all,result:(1, 3)  
dropna-several rows,result:(1, 3)  
EPSG:32650

|  | **NAME** | **KIND** | **geometry** |
| --- | --- | --- | --- |
| **0** | 南京市 | 0137 | POLYGON ((661383.381 3609433.909, 661397.682 3... |



将市域边界写入名为“public\_transportation”的数据库中，表名为’region’。

gpd2postSQL(region,table\_name='region',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The GeoDataFrame has been written to the PostgreSQL database.The table name is region.

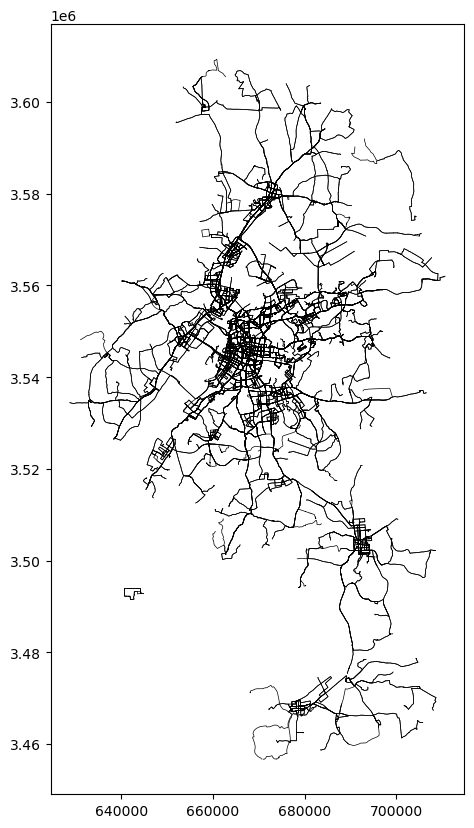
* 南京公交线路读写

读取SHP格式南京公交线路数据为GeoDataFrame格式，并写入数据库，表名为’bus\_routes’。读取数据时，指定了boundary参数配置裁切边界为市域边界矩形区域的包络线。同时中文字体为GBK编码。

bus\_routes=shp2gdf(args.data.bus\_routes,epsg=args.gi.nanjing\_epsg,boundary=region.envelope[0],encoding='GBK')

original data info:(1304, 9)  
dropna-how=all,result:(1304, 8)  
dropna-several rows,result:(1304, 8)  
EPSG:32650

import matplotlib.pyplot as plt  
fig, ax=plt.subplots(figsize=(10,10))  
bus\_routes.plot(ax=ax,edgecolor='black',linewidth=0.5,zorder=1)  
plt.show()



公交线路的字段（列名）包括线路名LineName，线路IDLineUid，运营开始时间StartTime，运营结束时间EndTime，工作日和周末运营时间WorkTimeDe，方向Direction，价格Price和Polyline几何对象geometry等。其中公交线路和公交站点各自数据中均有，用于链接的字段为LineUid。

bus\_routes.head(3)

|  | **LineName** | **LineUid** | **StartTime** | **EndTime** | **WorkTimeDe** | **Direction** | **Price** | **geometry** | **mask** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 100路(安德门-南医大二附院总站) | 079645c8210ef240754ff11b | 05:15 | 22:00 | 周一至周五 05:15-22:00 周六日 05:15-22:00 | 南医大二附院总站方向 | 2 | LINESTRING (666050.917 3541333.928, 666097.166... | True |
| **1** | 100路(南医大二附院总站-安德门) | 36ee56fea9cfa740cb09f01b | 06:00 | 22:45 | 周一至周五 06:00-22:45 周六日 06:00-22:45 | 安德门方向 | 2 | LINESTRING (664155.059 3551020.254, 664158.656... | True |
| **2** | 100路区间(下午班)(定坊工业园-安德门) | 0c835f11c853cb5cb5147fdf | 15:30 | 18:40 | 周一至周五 晚15:30-18:40 | 安德门方向 | 2 | LINESTRING (664034.596 3535151.892, 664093.486... | True |

gpd2postSQL(bus\_routes,table\_name='bus\_routes',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The GeoDataFrame has been written to the PostgreSQL database.The table name is bus\_routes.

* 南京公交站点读写

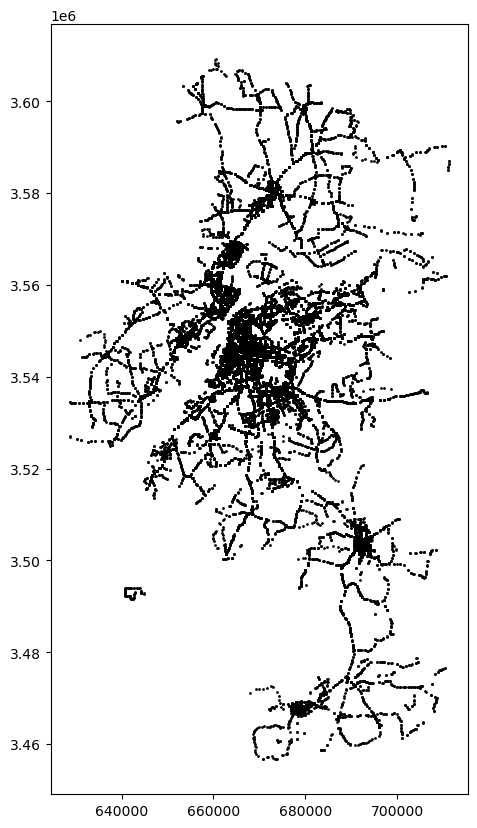
同公交线路的读写，公交站点字段包括站点名称PointName，站点（点）IDPointUid，经纬度Lng和Lat，所在公交线路名和IDLineName及LineUid，以及点（Point）几何对象geometry。写入数据库的表名为bus\_stations。

bus\_stations=shp2gdf(args.data.bus\_stations,epsg=args.gi.nanjing\_epsg,boundary=region.envelope[0],encoding='GBK')  
bus\_stations.head(3)

original data info:(32661, 7)  
dropna-how=all,result:(32661, 7)  
dropna-several rows,result:(32661, 7)  
EPSG:32650

|  | **PointName** | **PointUid** | **Lng** | **Lat** | **LineName** | **LineUid** | **geometry** | **mask** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 安德门 | aa03446d998e65a4b45f1e9f | 118.757782715597 | 31.9959552933912 | 100路(安德门-南医大二附院总站) | 079645c8210ef240754ff11b | POINT (666050.771 3541337.257) | True |
| **1** | 能仁里 | 78128091a7676747364aae39 | 118.76325592867 | 32.0017671207637 | 100路(安德门-南医大二附院总站) | 079645c8210ef240754ff11b | POINT (666557.365 3541990.011) | True |
| **2** | 雨花西路 | 03d556af603b624329fc48b1 | 118.766595750239 | 32.0060341836557 | 100路(安德门-南医大二附院总站) | 079645c8210ef240754ff11b | POINT (666865.156 3542468.234) | True |

fig, ax=plt.subplots(figsize=(10,10))  
bus\_stations.plot(ax=ax,color='black',markersize=1)  
plt.show()



gpd2postSQL(bus\_stations,table\_name='bus\_stations',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The GeoDataFrame has been written to the PostgreSQL database.The table name is bus\_stations.

* 地铁线路读写

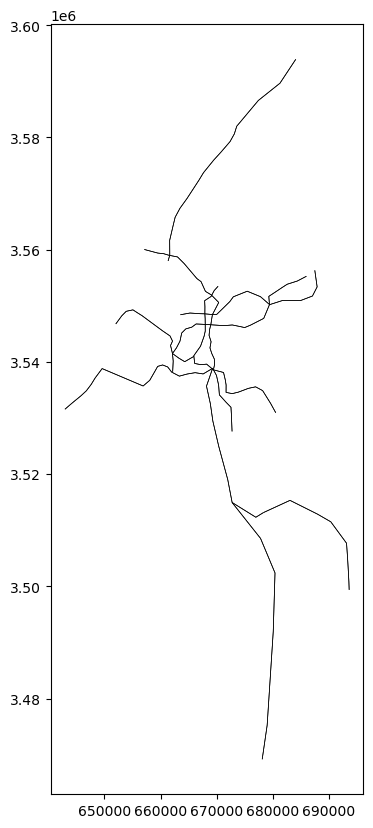
地铁线路字段名包括线路名LineName，线路IDLineName和Polyline几何对象geometry。写入数据库的表名为subway\_lines。

subway\_lines=shp2gdf(args.data.subway\_lines,epsg=args.gi.nanjing\_epsg,boundary=region.envelope[0],encoding='GBK')  
subway\_lines.head(3)

original data info:(20, 3)  
dropna-how=all,result:(20, 3)  
dropna-several rows,result:(20, 3)  
EPSG:32650

|  | **LineName** | **LineUid** | **geometry** | **mask** |
| --- | --- | --- | --- | --- |
| **0** | 地铁s9号线(高淳-翔宇路南) | 79743ab26a494368f8dc8594 | LINESTRING (678118.443 3469180.910, 679000.152... | True |
| **1** | 地铁s3号线(高家冲-南京南站) | d4a8b8c00653bd44c17da208 | LINESTRING (642965.556 3531558.066, 644261.635... | True |
| **2** | 地铁s7号线(无想山-空港新城江宁) | de67974fbfba8d7ca1d2309c | LINESTRING (693615.069 3499371.258, 693565.026... | True |

fig, ax=plt.subplots(figsize=(10,10))  
subway\_lines.plot(ax=ax,edgecolor='black',linewidth=0.5,zorder=1)  
plt.show()



gpd2postSQL(subway\_lines,table\_name='subway\_lines',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The GeoDataFrame has been written to the PostgreSQL database.The table name is subway\_lines.

* 地铁站点读写

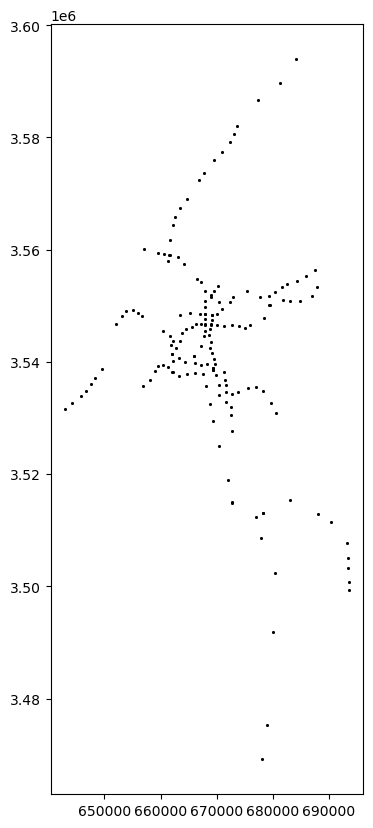
地铁站点的字段包括站点名PointName，站点IDPointUid，经纬度坐标Lng和Lat，所在地铁线路IDLineUid和点几何对象geometry。写入至数据库的表名为subway\_stations。

subway\_stations=shp2gdf(args.data.subway\_stations,epsg=args.gi.nanjing\_epsg,boundary=region.envelope[0],encoding='GBK')  
subway\_stations.head(3)

original data info:(348, 8)  
dropna-how=all,result:(348, 8)  
dropna-several rows,result:(348, 8)  
EPSG:32650

|  | **PointName** | **PointUid** | **Lng** | **Lat** | **IsPractica** | **LineName** | **LineUid** | **geometry** | **mask** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 高淳 | e4fe1091444ba694e311039c | 118.872368478559 | 31.3434577890014 | 0 | 地铁s9号线(高淳-翔宇路南) | 79743ab26a494368f8dc8594 | POINT (678118.443 3469180.910) | True |
| **1** | 团结圩 | 41396371183dc24fb9df8fea | 118.882721967772 | 31.3979546349684 | 1 | 地铁s9号线(高淳-翔宇路南) | 79743ab26a494368f8dc8594 | POINT (679000.152 3475239.180) | True |
| **2** | 明觉 | 2f7777a37adc2c0d88db732f | 118.897155970304 | 31.5482455693243 | 0 | 地铁s9号线(高淳-翔宇路南) | 79743ab26a494368f8dc8594 | POINT (680084.557 3491924.198) | True |

fig, ax=plt.subplots(figsize=(10,10))  
subway\_stations.plot(ax=ax,color='black',markersize=1)  
plt.show()



gpd2postSQL(subway\_stations,table\_name='subway\_stations',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The GeoDataFrame has been written to the PostgreSQL database.The table name is subway\_stations.

#### 2) 分析用数据预处理

* 城市公园数据读写

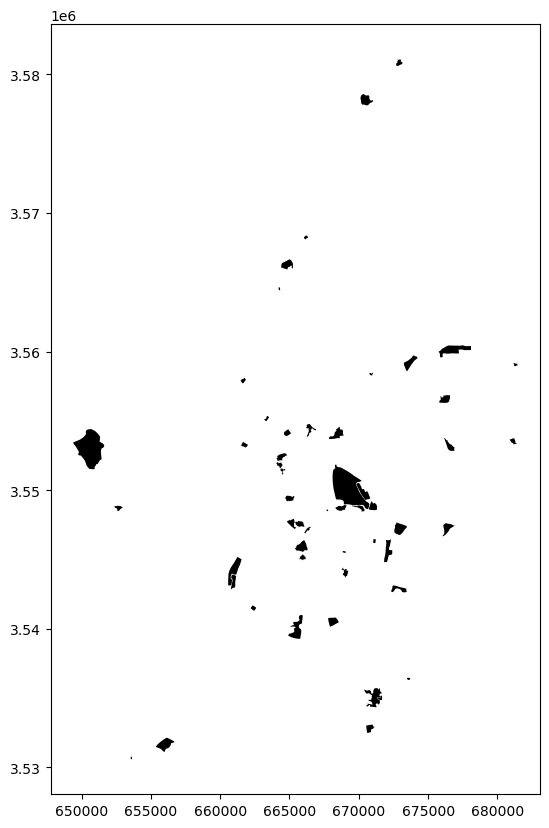
城市公园边界数据是根据《南京市绿地系统规划》中心城区规划综合公园一览表在Google Earth中绘制，再导出为KML格式数据。定义kml2gdf()函数读取KML格式数据为GeoDataFrame格式。并将其写入至数据库，表名为comprehensive\_park。

def kml2gdf(fn,epsg=None,boundary=None):   
 '''  
 读取KML格式数据为GeoDataFrame（GeoPandas）  
  
 Parameters  
 ----------  
 fn : string  
 .kml数据文件路径名.  
 epsg : pyproj.crs.crs.CRS[int], optional  
 投影epsg编号. The default is None.  
 boundary : shp, optional  
 配置裁切边界. The default is None.  
  
 Returns  
 -------  
 kml\_gdf\_proj : GeoDataFrame  
 读取KML格式为GeoDataFrame.  
  
 '''   
 import pandas as pd  
 import geopandas as gpd  
 import fiona,io  
 from tqdm import tqdm   
   
 from fiona.drvsupport import supported\_drivers  
 supported\_drivers['LIBKML'] = 'rw'  
 fiona.drvsupport.supported\_drivers['KML'] = 'rw'  
  
 kml\_gdf=gpd.GeoDataFrame()  
 for layer in tqdm(fiona.listlayers(fn)):  
 src=fiona.open(fn, layer=layer)  
 meta=src.meta  
 meta['driver']='KML'   
 with io.BytesIO() as buffer:  
 with fiona.open(buffer, 'w', \*\*meta) as dst:   
 for i, feature in enumerate(src):  
 if len(feature['geometry']['coordinates'][0]) > 1:  
 dst.write(feature)  
  
 buffer.seek(0)  
 one\_layer=gpd.read\_file(buffer,driver='KML')  
 one\_layer['group']=layer  
 kml\_gdf=kml\_gdf.append(one\_layer,ignore\_index=True)  
  
 if epsg is not None:  
 kml\_gdf\_proj=kml\_gdf.to\_crs(epsg=epsg)  
  
 if boundary:  
 kml\_gdf\_proj['mask']=kml\_gdf\_proj.geometry.apply(lambda row:row.within(boundary))  
 kml\_gdf\_proj.query('mask',inplace=True)   
  
 return kml\_gdf\_proj

comprehensive\_park=kml2gdf(args.data.comprehensive\_park,epsg=args.gi.nanjing\_epsg,boundary=None)

100%|██████████| 1/1 [00:00<00:00, 7.60it/s]

fig, ax=plt.subplots(figsize=(10,10))  
comprehensive\_park.plot(ax=ax,color='black')  
plt.show()  
  
comprehensive\_park.head(3)



|  | **Name** | **Description** | **geometry** | **group** |
| --- | --- | --- | --- | --- |
| **0** | 八字山公园 |  | POLYGON Z ((664351.229 3551540.582 0.000, 6643... | NanjingParks |
| **1** | 北崮山公园 |  | POLYGON Z ((667855.803 3553692.487 0.000, 6678... | NanjingParks |
| **2** | 大桥公园 |  | POLYGON Z ((664898.358 3554394.776 0.000, 6648... | NanjingParks |

同时翻译中文公园名为英文，建立映射字典（为可选项，根据后续需求确定）。增加字段Name\_EN存储英文名。

comprehensive\_park\_name\_mapping={  
 '八字山公园':'Bazi Mountain',  
 '北崮山公园':'Beigu Mountain',  
 '大桥公园':'Bridge park',  
 '古林公园':'Ancient Forest',  
 '鼓楼公园':'Drum Tower',  
 '老虎山公园':'Tiger Mountain',  
 '幕府山公园':'Mufu Mountain',  
 '清凉山公园':'Qingliang Mountain',  
 '狮子山公园':'Lion Rock Mountain',  
 '石头城公园':'Stone City Park',  
 '乌龙潭公园':'Wulong Pool',  
 '绣球公园':'Xiuqing Park',  
 '河西中央公园':'Hexi Central Park',  
 '绿博园':'Green EXPO Garden',  
 '滨江公园':'Riverside Park',  
 '莫愁湖公园':'Mochou Lake',  
 '南湖公园':'South Lake',  
 '百家湖公园':'Baijia Lake',  
 '凤凰公园':'Phoenix Park',  
 '九龙湖公园':'Jiulong Park',  
 '竹山公园':'Zhushan Park',  
 '九龙公园':'Jiulong Park',  
 '六合凤凰山公园':'Liuhe Phoenix Mountain',  
 '龙池公园':'Dragon Pool',  
 '平顶山公园':'Pingding Mountain',  
 '太子山公园':'Prince Mountain',  
 '宝塔山公园':'Pagoda Hill',  
 '北堡公园':'North Fort',  
 '凤凰山公园':'Phoenix Mountain',  
 '浦口公园':'Pukou Park',  
 '二桥公园':'Two Bridges Park',  
 '南炼公园':'Nanlian Park',  
 '三叶湖公园':'Three Leaf Lake',  
 '太平山公园':'Taiping Hill',  
 '乌龙山公园':'Wulong Mountain',  
 '燕子矶公园':'Swallow Rock',  
 '白鹭洲公园':'Egret Island',  
 '七桥瓮公园':'Qiqiaoweng Park',  
 '午朝门公园':'Wuchaomen Park',  
 '月牙湖公园':'Crescent Lake',  
 '郑和公园':'Zhenghe Park',  
 '白马公园':'White House Park',  
 '北极阁公园':'Arctic Pavilion',  
 '九华山公园':'Jiuhua Mountain',  
 '聚宝山公园':'Treasure Hill',  
 '梅花谷公园':'Plum Blossom Valley',  
 '情侣园':'Couples Garden',  
 '体育学院南公园':'South Park of Sports Institute',  
 '玄武湖公园':'Xuanwu Lake',  
 '花神湖公园':'Flora Lake',  
 '菊花台公园':'Chrysanthemums Terrace',  
 '莲花湖公园':'Lotus Lake',  
 '梅山公园':'Plum Blossom Hill',   
 }

comprehensive\_park['Name']=comprehensive\_park.Name.apply(lambda row:row.strip())  
comprehensive\_park['Name\_EN']=comprehensive\_park['Name'].map(comprehensive\_park\_name\_mapping)  
comprehensive\_park.head(3)

|  | **Name** | **Description** | **geometry** | **group** | **Name\_EN** |
| --- | --- | --- | --- | --- | --- |
| **0** | 八字山公园 |  | POLYGON Z ((664351.229 3551540.582 0.000, 6643... | NanjingParks | Bazi Mountain |
| **1** | 北崮山公园 |  | POLYGON Z ((667855.803 3553692.487 0.000, 6678... | NanjingParks | Beigu Mountain |
| **2** | 大桥公园 |  | POLYGON Z ((664898.358 3554394.776 0.000, 6648... | NanjingParks | Bridge park |

gpd2postSQL(comprehensive\_park,table\_name='comprehensive\_park',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The GeoDataFrame has been written to the PostgreSQL database.The table name is comprehensive\_park.

* 邻里站点提取

公共交通以站点为上下车位置，因此需要确定各个公园的邻近站点。计算每个公园5 min约400 m步行距离缓冲区，提取缓冲区内邻近公园的站点。定义adjacent\_pts()函数，可以根据输入的GeoDataFrame格式的Polygon对象polygons\_gdf根据配置的缓冲区大小buffer\_distance提取缓冲区内的Point点对象pts\_gdf。

读取已经存入到数据库中的公交和地铁站点，调用concat方法执行合并，统一考虑公园缓冲区内的公交和地铁站点。为了区别公交和地铁站点的PointUid标识，避免可能的重复，在地铁PointUid标识前增加s\_以示区别。

import pandas as pd  
  
bus\_stations=postSQL2gpd(table\_name='bus\_stations',\*\*args.db.db\_info)   
subway\_stations=postSQL2gpd(table\_name='subway\_stations',\*\*args.db.db\_info)   
subway\_stations['PointUid']=subway\_stations.PointUid.apply(lambda row:'s\_'+row)  
  
columns=['PointUid','geometry']  
stations=pd.concat([bus\_stations[columns],subway\_stations[columns]],ignore\_index=True)  
stations.head(3)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is bus\_stations.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is subway\_stations.

|  | **PointUid** | **geometry** |
| --- | --- | --- |
| **0** | aa03446d998e65a4b45f1e9f | POINT (666050.771 3541337.257) |
| **1** | 78128091a7676747364aae39 | POINT (666557.365 3541990.011) |
| **2** | 03d556af603b624329fc48b1 | POINT (666865.156 3542468.234) |

def adjacent\_pts\_within\_polygons(polygons\_gdf,pts\_gdf,buffer\_distance=200):  
 '''  
 提取位于Polygons几何对象缓冲区域内中的点几何对象  
  
 Parameters  
 ----------  
 polygons\_gdf : GeoDataFrame  
 含有Polygon对象的地理空间数据.  
 pts\_gdf : GeoDataFrame  
 含有Point点对象的地理空间数据.  
 buffer\_distance : float, optional  
 缓冲距离. The default is 200.  
  
 Returns  
 -------  
 adjacentpts\_gdf : GeoDataFrame  
 位于Polygons对象缓冲区域内的点对象.  
 buffer\_gdf : GeoDataFrame   
 缓冲区几何对象.  
 '''   
   
 from shapely.geometry import MultiPoint,Polygon,Point  
 from copy import deepcopy  
 import geopandas as gpd  
   
 adjacentpts\_gdf=deepcopy(polygons\_gdf)  
   
 def adjacent\_pts\_gdf\_single\_multipoint(geo\_polygon,return\_type='mp'):  
 buffer=geo\_polygon.buffer(buffer\_distance,join\_style=1)  
 pts\_gdf['within']=pts\_gdf.geometry.apply(lambda g:g.within(buffer))   
 pts\_gdf\_within=pts\_gdf[pts\_gdf['within'].values]  
   
 if return\_type=='mp':  
 return MultiPoint(pts\_gdf\_within['geometry'].to\_list())  
 elif return\_type=='pu':  
 return pts\_gdf\_within.PointUid.to\_list()  
   
 adjacentpts\_gdf['adjacent\_pts\_gdf']=adjacentpts\_gdf.geometry.apply(adjacent\_pts\_gdf\_single\_multipoint,args=('mp',))  
 adjacentpts\_gdf['adjacent\_PointUid']=adjacentpts\_gdf.geometry.apply(adjacent\_pts\_gdf\_single\_multipoint,args=('pu',))  
 adjacentpts\_gdf.rename(columns={'geometry':'geo\_park','adjacent\_pts\_gdf':'geometry'},inplace=True)   
 adjacentpts\_gdf['adjacent\_num']=adjacentpts\_gdf.adjacent\_PointUid.apply(lambda row:len(row))  
 adjacentpts\_gdf['park\_perimeter']=adjacentpts\_gdf.apply(lambda row:row.geo\_park.length,axis=1)  
 adjacentpts\_gdf['park\_area']=adjacentpts\_gdf.apply(lambda row:row.geo\_park.area,axis=1)  
 adjacentpts\_gdf['adjacent\_perimeterRatio']=adjacentpts\_gdf.apply(lambda row:len(row.adjacent\_PointUid)/row.geo\_park.length,axis=1)  
 adjacentpts\_gdf['adjacent\_areaRatio']=adjacentpts\_gdf.apply(lambda row:len(row.adjacent\_PointUid)/row.geo\_park.area,axis=1)   
   
 buffer\_gdf=deepcopy(adjacentpts\_gdf)  
 buffer\_gdf['buffer']=buffer\_gdf.geo\_park.apply(lambda row:row.buffer(buffer\_distance,join\_style=1))  
 buffer\_gdf.drop(['geometry','geo\_park'],axis=1,inplace=True)  
 buffer\_gdf.rename(columns={'buffer':'geometry'},inplace=True)  
   
 adjacentpts\_gdf.drop(['geo\_park'],axis=1,inplace=True)  
 return adjacentpts\_gdf,buffer\_gdf

comprehensive\_park=postSQL2gpd(table\_name='comprehensive\_park',\*\*args.db.db\_info)   
  
comprehensivePark\_adjacentStations,comprehensivePark\_buffer=adjacent\_pts\_within\_polygons(comprehensive\_park,stations,buffer\_distance=400)   
comprehensivePark\_buffer.head(3)

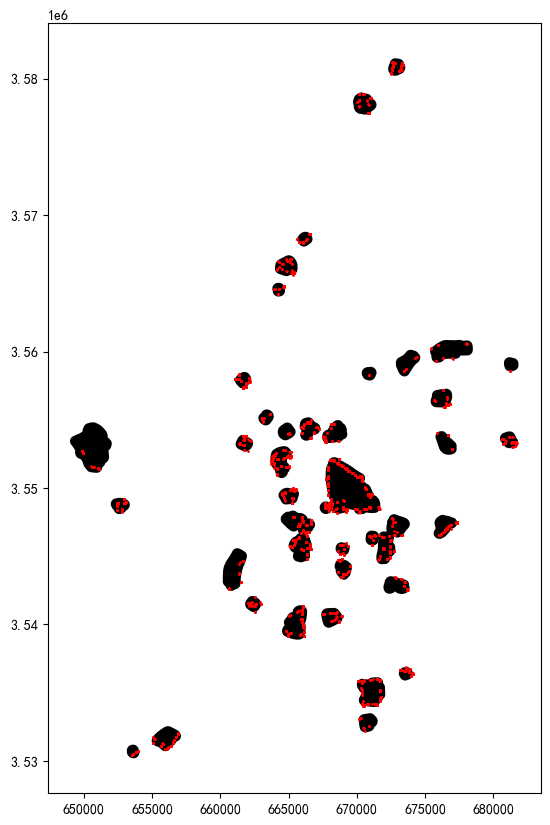
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is comprehensive\_park.

|  | **Name** | **Description** | **group** | **Name\_EN** | **adjacent\_PointUid** | **adjacent\_num** | **park\_perimeter** | **park\_area** | **adjacent\_perimeterRatio** | **adjacent\_areaRatio** | **geometry** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 八字山公园 | None | NanjingParks | Bazi Mountain | [3176ab8f267a522adc1e5249, 13627db9677fe868086... | 58 | 1498.063650 | 58555.986474 | 0.038717 | 0.000991 | POLYGON ((663972.964 3551670.642, 663979.695 3... |
| **1** | 北崮山公园 | None | NanjingParks | Beigu Mountain | [c7b453844859a109e6e96551, 3a35fc72daefc4b516a... | 46 | 3374.373940 | 429974.743870 | 0.013632 | 0.000107 | POLYGON ((667455.811 3553695.094, 667456.223 3... |
| **2** | 大桥公园 | None | NanjingParks | Bridge park | [e16200a462b5e86eaf080f13, 9bb91a554c924a9629f... | 14 | 1501.575973 | 152867.454776 | 0.009324 | 0.000092 | POLYGON ((664485.150 3553599.191, 664459.012 3... |

comprehensivePark\_adjacentStations.head(3)

|  | **Name** | **Description** | **group** | **Name\_EN** | **geometry** | **adjacent\_PointUid** | **adjacent\_num** | **park\_perimeter** | **park\_area** | **adjacent\_perimeterRatio** | **adjacent\_areaRatio** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 八字山公园 | None | NanjingParks | Bazi Mountain | MULTIPOINT (664855.439 3551547.768, 664432.335... | [3176ab8f267a522adc1e5249, 13627db9677fe868086... | 58 | 1498.063650 | 58555.986474 | 0.038717 | 0.000991 |
| **1** | 北崮山公园 | None | NanjingParks | Beigu Mountain | MULTIPOINT (668098.717 3554664.629, 668137.023... | [c7b453844859a109e6e96551, 3a35fc72daefc4b516a... | 46 | 3374.373940 | 429974.743870 | 0.013632 | 0.000107 |
| **2** | 大桥公园 | None | NanjingParks | Bridge park | MULTIPOINT (664957.333 3553981.379, 664934.772... | [e16200a462b5e86eaf080f13, 9bb91a554c924a9629f... | 14 | 1501.575973 | 152867.454776 | 0.009324 | 0.000092 |

fig, ax=plt.subplots(figsize=(10,10))  
comprehensivePark\_buffer.plot(ax=ax,color='black')  
comprehensivePark\_adjacentStations.plot(ax=ax,color='red',markersize=1)  
plt.show()



将公园邻里站点的返回结果邻里站点和缓冲区域均写入至数据库，表名分别为adjacent\_stations和park\_buffer。

gpd2postSQL(comprehensivePark\_adjacentStations,table\_name='adjacent\_stations',\*\*args.db.db\_info)   
gpd2postSQL(comprehensivePark\_buffer,table\_name='park\_buffer',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The GeoDataFrame has been written to the PostgreSQL database.The table name is adjacent\_stations.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The GeoDataFrame has been written to the PostgreSQL database.The table name is park\_buffer.

* 人口分布数据读写

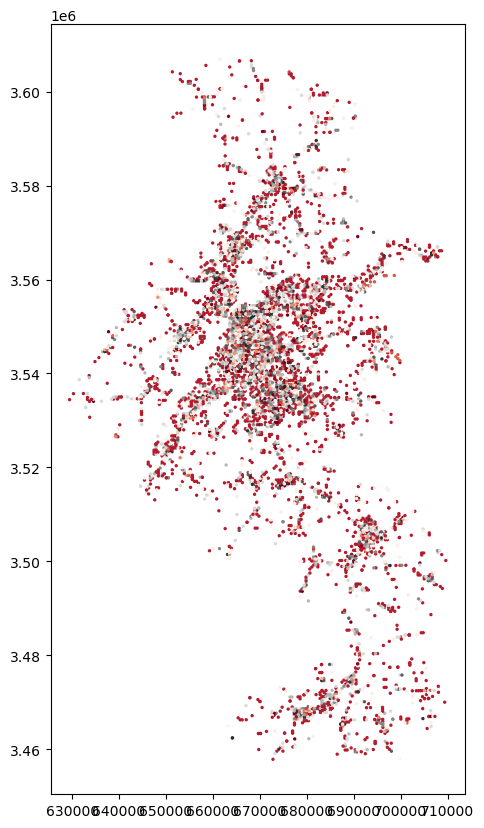
可以将人口数据按各个站点位置，指定缓冲区，将其内的总人口数据传入网络，以高斯函数等作为权重，根据实际路线衰减人口数至公园邻近站点，从而计算各个公园的人口服务压力。读取人口分布数据，该数据为给定经纬度地理点位的人口数统计。统一将数据写入至数据库，表名为population。

population=shp2gdf(args.data.population,epsg=args.gi.nanjing\_epsg)  
population.head(3)

original data info:(6673, 4)  
dropna-how=all,result:(6673, 4)  
dropna-several rows,result:(6673, 4)  
EPSG:32650

|  | **Lat** | **Lng** | **Population** | **geometry** |
| --- | --- | --- | --- | --- |
| **0** | 31.9497433580896 | 118.378427463393 | 129 | POINT (630276.381 3535694.515) |
| **1** | 31.9497433580896 | 118.400885345496 | 129 | POINT (632399.081 3535721.765) |
| **2** | 31.9230613713896 | 118.405376921916 | 386 | POINT (632862.009 3532769.406) |

fig, ax=plt.subplots(figsize=(10,10))  
population.plot(ax=ax,column='Population',markersize=2,cmap='RdGy')  
plt.show()



gpd2postSQL(population,table\_name='population',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The GeoDataFrame has been written to the PostgreSQL database.The table name is population.

* 行政区划读写

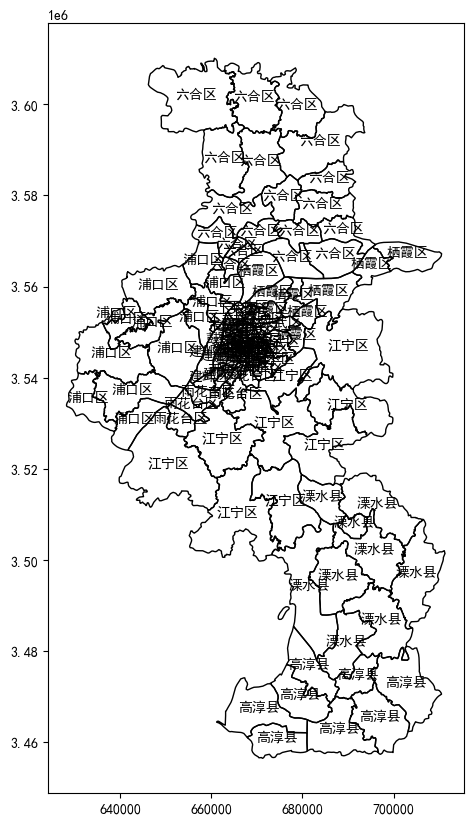
可以按行政区划分析公共交通复合网络对综合公园的影响，例如公园的供给服务和居民对公园服务的需求，及公园分布的均衡性或不均衡性等。读取的行政区划，每一区划为多个Polygon对象，需要按照区划字段countyname分组将同一组（相同值）空间相邻的几何对象聚合成一个，使用GeoDataFrame.dissolve(by=None, aggfunc=‘first’, as\_index=True, level=None, sort=True, observed=False, dropna=True)方法完成。在同组聚合时，要配合aggfunc参数，确定同组多个行对象所含属性字段计算的方式，例如对area和人口密[度]字段执行求和sum，对其它字段均选择第一行值first作为聚合后对应的字段值。最终聚合后，由117行聚合为13行，对应13个行政区划。

administrative\_districts=shp2gdf(args.data.administrative\_districts,epsg=args.gi.nanjing\_epsg,encoding='GBK')  
print(administrative\_districts.shape)  
administrative\_districts.head(3)

original data info:(117, 15)  
dropna-how=all,result:(117, 14)  
dropna-several rows,result:(117, 14)  
EPSG:32650  
(117, 14)

|  | **gml\_id** | **id** | **fldm** | **xzdm** | **Name** | **area** | **cityname** | **citycode** | **countyname** | **countycode** | **province** | **prvcode** | **人口密** | **geometry** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | layer\_township\_pg.32902 | 35004 | 1004 | 320102010 | 红山街道 | 7.158708 | 南京市 | 320100000 | 玄武区 | 320102000 | 江苏省 | 320000000 | 12323.034145 | POLYGON ((668836.064 3551847.179, 668869.968 3... |
| **1** | layer\_township\_pg.32903 | 35005 | 1004 | 320102008 | 孝陵卫街道 | 29.465389 | 南京市 | 320100000 | 玄武区 | 320102000 | 江苏省 | 320000000 | 3823.265387 | POLYGON ((670653.916 3548460.478, 671258.502 3... |
| **2** | layer\_township\_pg.32904 | 35006 | 1004 | 320102005 | 后宰门街道 | 3.574642 | 南京市 | 320100000 | 玄武区 | 320102000 | 江苏省 | 320000000 | 17341.598641 | POLYGON ((672016.060 3546395.591, 671962.790 3... |

plt.rcParams['font.sans-serif']=['SimHei']  
fig, ax=plt.subplots(figsize=(10,10))  
administrative\_districts.plot(ax=ax,color='none', edgecolor='k')  
administrative\_districts.apply(lambda x: ax.annotate(text=x['countyname'], xy=x.geometry.centroid.coords[0], ha='center'), axis=1);  
plt.show()



按分组执行空间几何对象的聚合。也增加了countyname\_EN字段存储行政区划翻译的英文名称（为可选，根据后续需要确定）。

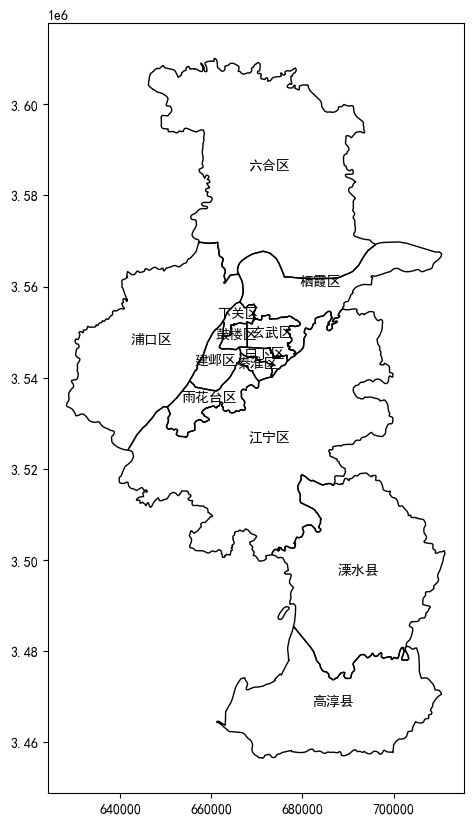
administrative\_districts\_name\_mapping={  
 '玄武区':'Xuanwu District',   
 '白下区':'Baixia District',   
 '建邺区':'Jianye District',   
 '下关区':'Xiaguan District',   
 '栖霞区':'Qixia District',   
 '六合区':'Liuhe District',   
 '鼓楼区':'Gulou District',   
 '秦淮区':'Qianhuai District',   
 '浦口区':'Pukou District',  
 '雨花台区':'Yuhuatai District',   
 '江宁区':'Jiangning District',   
 '溧水县':'Lishui Country',   
 '高淳县':'Gaochun Country',   
 }  
  
administrative\_districts.countyname=administrative\_districts.countyname.apply(lambda row:row.strip())  
administrative\_districts['countyname\_EN']=administrative\_districts['countyname'].map(administrative\_districts\_name\_mapping)  
administrative\_districts.drop(['gml\_id', 'id', 'xzdm', 'Name',],inplace=True,axis=1)  
aggregation\_functions={'fldm':'first', 'area':'sum', 'cityname':'first', 'citycode':'first','countyname':'first', 'countycode':'first', 'province':'first', 'prvcode':'first', '人口密':'sum', 'countyname\_EN':'first'}  
administrative\_districts\_dissolved=administrative\_districts.dissolve(by='countyname\_EN',aggfunc=aggregation\_functions)

print(administrative\_districts\_dissolved.shape)  
administrative\_districts\_dissolved.head(3)

(13, 11)

|  | **geometry** | **fldm** | **area** | **cityname** | **citycode** | **countyname** | **countycode** | **province** | **prvcode** | **人口密** | **countyname\_EN** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **countyname\_EN** |  |  |  |  |  |  |  |  |  |  |  |
| **Baixia District** | POLYGON ((673587.345 3544770.147, 673589.306 3... | 1004 | 25.823915 | 南京市 | 320100000 | 白下区 | 320103000 | 江苏省 | 320000000 | 469795.950111 | Baixia District |
| **Gaochun Country** | POLYGON ((680940.507 3458039.467, 680897.309 3... | 1004 | 792.063405 | 南京市 | 320100000 | 高淳县 | 320125000 | 江苏省 | 320000000 | 4365.512334 | Gaochun Country |
| **Gulou District** | POLYGON ((665909.067 3546529.362, 665845.406 3... | 1004 | 24.819821 | 南京市 | 320100000 | 鼓楼区 | 320106000 | 江苏省 | 320000000 | 310646.327962 | Gulou District |

plt.rcParams['font.sans-serif']=['SimHei']  
  
fig, ax=plt.subplots(figsize=(10,10))  
administrative\_districts\_dissolved.plot(ax=ax,color='none', edgecolor='k')  
administrative\_districts\_dissolved.apply(lambda x: ax.annotate(text=x['countyname'], xy=x.geometry.centroid.coords[0], ha='center'), axis=1);  
plt.show()



将行政区划聚合后的对象写入至数据库，表名为admin\_distr。

gpd2postSQL(administrative\_districts\_dissolved,table\_name='admin\_distr',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The GeoDataFrame has been written to the PostgreSQL database.The table name is admin\_distr.

#### 3）GeoLife数据与时速统计

公共交通复合网络的空间距离权重，在考虑实际路线长度的同时，需要考虑交通工具的速度，计算为时间成本距离。通过[GeoLife数据集](https://www.microsoft.com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide/)②计算公交速度、地铁速度及二者各自内部换乘和之间换乘的步行速度。因为该数据集中南京区域的数据极少，主要以北京为主，因此以北京区域的计算结果为参考。提取北京区域30万条数据，每条数据包括经纬度、时间等信息，GPS轨迹的精度为每2~5 s记录一次。通过记录有通行模式、起始和结束时间戳的数据，确定每一条数据的通行方式。最终获取880条连续路径，其中公交103条，地铁（含城铁）52条，步行368条。以各自速度中位数（公交均速13.54 km/h、地铁均速18.77 km/h、步行均速4.39 km/h）配置公共交通复合网络的3个子网络，包括公交系统子网络、地铁系统子网络和换乘子网络。

GPS轨迹数据是由微软亚洲研究院（Microsoft Research Asia）主导的GeoLife项目由178 个用户在四年多的时间里（从2007年4月到2011年10月）收集的。该数据集的GPS轨迹由一系列带有时间戳的点表示，每个点都包含纬度、经度和高度信息。 该数据集包含17,621条轨迹，总距离为1,251,654公里，总时长为48,203小时。 这些轨迹由不同的GPS记录器和GPS手机记录，并具有多种采样率。 91%的轨迹以密集表示形式记录，例如每 1~5 秒或每 5~10 米一点。该数据集记录了用户广泛的户外运动，不仅包括回家和上班等常规生活活动，还包括一些娱乐和体育活动，如购物、观光、餐饮、徒步旅行和骑自行车。该轨迹数据集可用于许多研究领域，例如移动模式挖掘、用户活动识别、基于位置的社交网络、位置隐私和位置推荐等。

下载的GPS轨迹数据（GeoLife数据集）包括181个子文件，从“000”编号至“181”，每个文件夹下含“Trajectory”子文件夹和一个labels.txt标签文件，“Trajectory”子文件夹下即包含后缀名为.plt的多个GPS轨迹数据，每行数据对应标签为纬度、经度、海拔、时间、格式化日期和格式化时间，单个文件抬头和部分数据如下：

.plt文件数据示例：

Geolife trajectory  
WGS 84  
Altitude is in Feet  
Reserved 3  
0,2,255,My Track,0,0,2,8421376  
0  
39.984702,116.318417,0,492,39744.1201851852,2008-10-23,02:53:04  
39.984683,116.31845,0,492,39744.1202546296,2008-10-23,02:53:10  
39.984686,116.318417,0,492,39744.1203125,2008-10-23,02:53:15  
39.984688,116.318385,0,492,39744.1203703704,2008-10-23,02:53:20  
39.984655,116.318263,0,492,39744.1204282407,2008-10-23,02:53:25  
39.984611,116.318026,0,493,39744.1204861111,2008-10-23,02:53:30

labels.txt标签文件用于标识起始Start Time和终止End Time时间段内乘坐的交通方式Transportation Mode。

labels.txt文件数据示例：

Start Time End Time Transportation Mode  
2007/05/09 14:48:28 2007/05/09 15:00:03 taxi  
2007/05/13 02:27:42 2007/05/13 03:33:15 car  
2007/05/13 03:47:05 2007/05/13 04:24:42 car  
2007/05/13 23:26:53 2007/05/14 00:03:05 car  
2007/05/15 00:46:24 2007/05/15 00:56:51 walk  
2007/05/15 13:41:41 2007/05/15 14:01:51 taxi  
2007/05/16 01:31:27 2007/05/16 01:33:05 walk  
2007/05/19 08:58:18 2007/05/19 09:01:36 walk  
2007/05/19 09:02:10 2007/05/19 09:49:54 bus

* 合并GeoLife数据集

定义geolife\_dataset\_preprocessing()函数，将单独标识有.plt后缀的文件合并为一个GeoDataFrame对象（如果文件较大，可以根据以时间命名的文件名拆分为按时间周期切分的多个文件）；同时合并每一子文件下的labels.txt标签数据，分别返回.plt和labels.txt合并后的数据。数据量比较大，为了减少数据量或者仅GPS轨迹数据（GeoLife数据集）GPS轨迹数据（GeoLife数据集）提取给定区域内的数据，提供boundary参数，根据左下角和右上角坐标范围提取数据。

def geolife\_dataset\_preprocessing(geolife\_fp,fileType,epsg=None,boundary=None):  
 '''  
 将分散的GPS轨迹数据（GeoLife数据集）合并为一个单独的GeoDataFrame格式数据对象  
  
 Parameters  
 ----------  
 geolife\_fp : string  
 GeoLife数据集根目录.  
 fileType : list[str]  
 文件类型，为.plt轨迹数据和.txt的标签数据.  
 epsg : pyproj.crs.crs.CRS[int], optional  
 投影epsg编号. The default is None.  
 boundary : list[float], optional  
 [右上角坐标(Lat,Lng), 左下角坐标(Lat,Lng)]，例如[41.095365,117.756161,39.399080,115.095937]. The default is None.  
  
 Returns  
 -------  
 plts\_gdf : GeoDataFrame  
 合并的GPS轨迹数据.  
 labels\_df : DataFrame  
 合并的标签数据.  
  
 '''   
   
 import util\_misc  
 import os  
 import pandas as pd  
 from tqdm import tqdm  
 from shapely.geometry import Point  
 import geopandas as gpd  
   
 plt\_columns=['Latitude','Longitude','code','altitude','date','date\_str','time\_str']  
 plt\_fns=util\_misc.filePath\_extraction(geolife\_fp,fileType[0])  
 plts\_df=pd.DataFrame(columns=plt\_columns)  
 for root, fns in tqdm(plt\_fns.items()):  
 for fn in fns:  
 fp=os.path.join(root,fn)  
 plt\_df=(pd.read\_csv(fp, names=plt\_columns, skiprows=6, sep=','))  
 plts\_df=pd.concat([plts\_df, plt\_df])  
  
 if boundary:  
 plts\_df['within']=plts\_df.apply(lambda row:1 if boundary[2]<row.Latitude<boundary[0] and boundary[3]<row.Longitude<boundary[1] else 0,axis=1)  
 plts\_df=plts\_df[plts\_df['within']==1]  
  
 plts\_df['geometry']=plts\_df.apply(lambda row:Point(row.Longitude,row.Latitude),axis=1)  
 wgs\_epsg=4326  
 plts\_gdf=gpd.GeoDataFrame(plts\_df,crs=wgs\_epsg)  
 if epsg is not None:  
 plts\_gdf=plts\_gdf.to\_crs(epsg=epsg)   
   
 labels\_fns=util\_misc.filePath\_extraction(geolife\_fp,fileType[1])  
 label\_columns=['start\_time','end\_time','transportation\_mode']  
 labels\_df=pd.DataFrame(columns=label\_columns)  
 for root, fns in tqdm(labels\_fns.items()):  
 for fn in fns:  
 fp=os.path.join(root,fn)  
 label\_df=(pd.read\_csv(fp, names=label\_columns, skiprows=1, sep='\t'))  
 labels\_df=pd.concat([labels\_df,label\_df],ignore\_index=True)  
  
 print("transportation\_mode:{}".format(labels\_df.transportation\_mode.unique()))   
   
 return plts\_gdf,labels\_df

boundary=[41.095365,117.756161,39.399080,115.095937] # right\_coordinate(Lat,Lng) left coordinate(Lat,Lng) ；Beijing region  
geolife\_gdf,geolife\_labels\_df=geolife\_dataset\_preprocessing(args.data.geolife,['plt','txt'],args.gi.beijing\_epsg,boundary)

100%|██████████| 182/182 [2:46:20<00:00, 54.84s/it]   
C:\Users\richi\anaconda3\envs\usda\lib\site-packages\pandas\core\dtypes\cast.py:127: ShapelyDeprecationWarning: The array interface is deprecated and will no longer work in Shapely 2.0. Convert the '.coords' to a numpy array instead.  
 arr = construct\_1d\_object\_array\_from\_listlike(values)  
100%|██████████| 69/69 [00:00<00:00, 200.33it/s]  
  
  
transportation\_mode:['bus' 'train' 'taxi' 'walk' 'subway' 'airplane' 'car' 'bike' 'boat' 'run'  
 'motorcycle']

geolife\_gdf.head(3)

|  | **Latitude** | **Longitude** | **code** | **altitude** | **date** | **date\_str** | **time\_str** | **within** | **geometry** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 39.984702 | 116.318417 | 0 | 492 | 39744.120185 | 2008-10-23 | 02:53:04 | 1 | POINT (441807.057 14426281.714) |
| **1** | 39.984683 | 116.318450 | 0 | 492 | 39744.120255 | 2008-10-23 | 02:53:10 | 1 | POINT (441809.858 14426279.584) |
| **2** | 39.984686 | 116.318417 | 0 | 492 | 39744.120313 | 2008-10-23 | 02:53:15 | 1 | POINT (441807.043 14426279.938) |

因为GPS轨迹数据合并后的数据量较大，并未将其存入本地数据库，而是存储到本地磁盘中（或外置磁盘中）。

geolife\_gdf.to\_file(args.data.geolife\_gdf)

C:\Users\richi\anaconda3\envs\usda\lib\site-packages\geopandas\io\file.py:299: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas in a future version. Use pandas.Index with the appropriate dtype instead.  
 pd.Int64Index,

geolife\_labels\_df.head(3)

|  | **start\_time** | **end\_time** | **transportation\_mode** |
| --- | --- | --- | --- |
| **0** | 2007/06/26 11:32:29 | 2007/06/26 11:40:29 | bus |
| **1** | 2008/03/28 14:52:54 | 2008/03/28 15:59:59 | train |
| **2** | 2008/03/28 16:00:00 | 2008/03/28 22:02:00 | train |

geolife标签数据直接写入到数据库，表名为geolife\_labels。

df2postSQL(geolife\_labels\_df,'geolife\_labels',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The GeoDataFrame has been written to the PostgreSQL database.The table name is geolife\_labels.

* 合并GPS轨迹和标签

.pltGPS轨迹数据和labels.txt标签数据是分离的，需要将其按照标签数据提供的始末时间对应的交通工具，对位到GPS轨迹数据上，才可以统计不同交通方式下的时速。因为数据量庞大，且仅仅统计交通工具的均速，因此并未使用全部数据统计，而是提取了300,000条数据进行计算。为了提高计算效率，使用并行计算的方式，对于并行计算需要在Spyder中完成，分别建立geolife\_data\_processing\_pool和geolife\_data\_processing模块。并将运行结果写入至数据库，表名为geolife\_tm\_beijing。

geolife\_data\_processing\_pool.py

import pandas as pd  
from tqdm import tqdm  
from multiprocessing import Pool  
import numpy as np  
from database import df2postSQL,postSQL2df  
from util\_misc import AttrDict  
\_\_C=AttrDict()   
args=\_\_C  
  
\_\_C.db=AttrDict()   
\_\_C.db.UN='postgres'  
\_\_C.db.PW='123456'  
\_\_C.db.DB='public\_transportation'  
\_\_C.db.GC='geometry'   
\_\_C.db.db\_info=dict(geom\_col=args.db.GC,myusername=args.db.UN,mypassword=args.db.PW,mydatabase=args.db.DB)  
  
geolife\_labels=postSQL2df('geolife\_labels',\*\*args.db.db\_info)  
geolife\_labels['st\_dt']=pd.to\_datetime(geolife\_labels['start\_time'])  
geolife\_labels['et\_dt']=pd.to\_datetime(geolife\_labels['end\_time'])  
geolife\_labels.sort\_values(by='st\_dt',inplace=True)  
geolife\_labels['label\_idx']=range(1, len(geolife\_labels)+1)  
  
def trans\_mode(row):  
 """根据轨迹点时间和标签时间段，确定轨迹点的交通模式"""  
 dt=row.datetime  
 t\_mode=None  
   
 for idx\_label,row\_label in geolife\_labels.iterrows():  
 transportation\_mode=row\_label.transportation\_mode  
 st\_dt=row\_label.st\_dt  
 et\_dt=row\_label.et\_dt  
 label\_idx=row\_label.label\_idx  
 if (dt>=st\_dt) & (dt<=et\_dt):  
 t\_mode=transportation\_mode  
 return pd.Series({"trans\_mode":t\_mode,"label\_idx":label\_idx})  
   
 return pd.Series({"trans\_mode":t\_mode,"label\_idx":None})  
  
def process(df):  
 results=df.apply(trans\_mode, axis=1)  
 return results

geolife\_data\_processing.py

import geopandas as gpd  
from database import postSQL2gpd,gpd2postSQL,df2postSQL,postSQL2df  
from util\_misc import AttrDict  
\_\_C=AttrDict()   
args=\_\_C  
  
\_\_C.db=AttrDict()   
\_\_C.db.UN='postgres'  
\_\_C.db.PW='123456'  
\_\_C.db.DB='public\_transportation'  
\_\_C.db.GC='geometry'   
\_\_C.db.db\_info=dict(geom\_col=args.db.GC,myusername=args.db.UN,mypassword=args.db.PW,mydatabase=args.db.DB)  
  
\_\_C.gi=AttrDict()  
\_\_C.gi.nanjing\_epsg=32650  
\_\_C.gi.beijing\_epsg=32750  
\_\_C.gi.epsg\_wgs84=4326  
  
\_\_C.data=AttrDict()  
\_\_C.data.geolife\_gdf='E:\data\geolife.gpkg'  
  
def geolife\_transportation\_mode(geolife\_gdf,geolife\_labels,epsg=None):  
 '''  
 合并GeoLife数据的GPS轨迹和交通工具时间段标签，可用于计算不同通行工具的均速  
  
 Parameters  
 ----------  
 geolife\_gdf : GeoDataFrame  
 合并的GPS轨迹数据.  
 geolife\_labels : DataFrame  
 合并后的标签数据.  
 epsg :pyproj.crs.crs.CRS[int], optional  
 投影epsg编号. The default is None.  
  
 Returns  
 -------  
 geolife\_tm\_gdf : GeoDataFrame  
 合并GPS轨迹和标签后（交通方式）的数据.  
 geolife\_labels : DataFrame  
 标签数据，将时间字符串转换为了时间格式数据.  
  
 '''   
 import pandas as pd  
 from tqdm import tqdm  
 from multiprocessing import Pool  
 import numpy as np  
 from geolife\_data\_processing\_pool import process  
 tqdm.pandas()  
   
 geolife\_gdf=geolife\_gdf.to\_crs(epsg=epsg)   
 geolife\_gdf['datetime']=pd.to\_datetime(geolife\_gdf['date\_str']+' '+geolife\_gdf['time\_str'])  
 geolife\_gdf.sort\_values(by='datetime',inplace=True)  
   
 geolife\_labels['st\_dt']=pd.to\_datetime(geolife\_labels['start\_time'])  
 geolife\_labels['et\_dt']=pd.to\_datetime(geolife\_labels['end\_time'])  
 geolife\_labels.sort\_values(by='st\_dt',inplace=True)  
   
 workers=8  
 with Pool(workers) as p:  
 pool\_results=p.map(process, tqdm(np.array\_split(geolife\_gdf,workers)))  
 t\_mode\_df=pd.concat(pool\_results, axis=0)  
 geolife\_tm\_gdf=pd.concat([geolife\_gdf, t\_mode\_df], axis=1)  
   
 return geolife\_tm\_gdf,geolife\_labels  
  
  
if \_\_name\_\_=="\_\_main\_\_":  
 geolife\_beijing\_gdf=gpd.read\_file(args.data.geolife\_gdf)  
 geolife\_beijing\_gdf\_part=geolife\_beijing\_gdf[:50000\*6]  
 print('-'\*50)  
 geolife\_labels\_df=postSQL2df('geolife\_labels',\*\*args.db.db\_info)  
 geolife\_tm\_beijing\_gdf,geolife\_labels=geolife\_transportation\_mode(geolife\_beijing\_gdf\_part,geolife\_labels\_df,args.gi.beijing\_epsg)   
 gpd2postSQL(geolife\_tm\_beijing\_gdf,table\_name='geolife\_tm\_beijing',myusername='postgres',mypassword='123456',mydatabase='public\_transport\_accessibility')

* 交通工具时速统计

统计不同交通方式下的通行时速，首先定义geolife\_traffic\_mode\_trajectory\_pts2linestring()函数，将GPS轨迹点按照所属，交通模式和时间顺序连为线段，计算线段（路径）的长度、持续时间，对应交通模式就可以计算交通工具的时速。

geolife\_tm\_beijing\_gdf=postSQL2gpd(table\_name='geolife\_tm\_beijing',myusername='postgres',mypassword='123456',mydatabase='public\_transport\_accessibility')

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is geolife\_tm\_beijing.

geolife\_tm\_beijing\_gdf

|  | **Latitude** | **Longitude** | **code** | **altitude** | **date** | **date\_str** | **time\_str** | **geometry** | **within** | **datetime** | **trans\_mode** | **label\_idx** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 39.988992 | 116.327023 | 0 | 128.937004593176 | 36526.966887 | 2000-01-01 | 23:12:19 | POINT (442545.429 14426752.285) | 1 | 2000-01-01 23:12:19 | None | NaN |
| **1** | 39.990964 | 116.327041 | 0 | 221.128615485564 | 36526.967604 | 2000-01-01 | 23:13:21 | POINT (442548.618 14426971.149) | 1 | 2000-01-01 23:13:21 | None | NaN |
| **2** | 39.993207 | 116.326827 | 0 | 217.191591207349 | 36526.969016 | 2000-01-01 | 23:15:23 | POINT (442532.229 14427220.241) | 1 | 2000-01-01 23:15:23 | None | NaN |
| **3** | 39.974233 | 116.330383 | 0 | 823.490813648294 | 39184.396898 | 2007-04-12 | 09:31:32 | POINT (442820.014 14425112.043) | 1 | 2007-04-12 09:31:32 | None | NaN |
| **4** | 39.974317 | 116.330450 | 0 | 823.490813648294 | 39184.402512 | 2007-04-12 | 09:39:37 | POINT (442825.777 14425121.250) | 1 | 2007-04-12 09:39:37 | None | NaN |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **299995** | 40.885653 | 117.069078 | 0 | 1360 | 39569.179329 | 2008-05-01 | 04:18:14 | POINT (505819.594 14526065.883) | 1 | 2008-05-01 04:18:14 | car | 2643.0 |
| **299996** | 39.893903 | 116.315097 | 0 | 216.5 | 39569.179329 | 2008-05-01 | 04:18:14 | POINT (441446.220 14416206.097) | 1 | 2008-05-01 04:18:14 | car | 2643.0 |
| **299997** | 40.006852 | 115.977567 | 0 | 1023.6 | 39569.179329 | 2008-05-01 | 04:18:14 | POINT (412733.302 14429018.274) | 1 | 2008-05-01 04:18:14 | car | 2643.0 |
| **299998** | 40.006852 | 115.977567 | 0 | 1023.6 | 39569.179329 | 2008-05-01 | 04:18:14 | POINT (412733.302 14429018.274) | 1 | 2008-05-01 04:18:14 | car | 2643.0 |
| **299999** | 39.893903 | 116.315097 | 0 | 216.5 | 39569.179329 | 2008-05-01 | 04:18:14 | POINT (441446.220 14416206.097) | 1 | 2008-05-01 04:18:14 | car | 2643.0 |

300000 rows × 12 columns

def geolife\_traffic\_mode\_trajectory\_pts2linestring(geolife\_tm\_gdf,epsg=None):  
 '''  
 将GeoLife轨迹和标签合并后数据中的轨迹点按照轨迹所属，交通模式和时间顺序连为线段  
 Parameters  
 ----------  
 geolife\_tm\_gdf : GeoDataFrame  
 GeoLife轨迹和标签合并后数据.  
 epsg : pyproj.crs.crs.CRS[int], optional  
 投影epsg编号. The default is None.  
  
 Returns  
 -------  
 duration\_gdf : GeoDataFrame  
 合并GPS轨迹点为折线段的地理空间数据，包含路径长度、持续时间、交通模式等字段.  
  
 '''   
 import copy  
 from shapely.geometry import LineString  
 from shapely.geometry import Point  
 import geopandas as gpd  
 from tqdm import tqdm  
   
 geolife\_tm\_gdf\_=copy.deepcopy(geolife\_tm\_gdf)  
 trans\_mode\_list=list(geolife\_tm\_gdf\_.trans\_mode.unique())  
   
 geolife\_tm\_gdf\_.dropna(inplace=True)  
 geolife\_tm\_gdf\_.set\_index(['label\_idx','trans\_mode'],inplace=True)   
 columns=['line\_length','duration','trans\_mode','geometry','label\_idx']  
 duration\_df=pd.DataFrame(columns=columns)  
 i=0  
 for label\_idx,df\_ in tqdm(geolife\_tm\_gdf\_.groupby(level=0)):  
 df=df\_[::]  
 df.sort\_values(by=['datetime'],inplace=True)  
 if len(list(df.geometry))>1:  
 line=LineString(list(df.geometry))  
 line\_len=line.length  
 else:   
 line\_len=None  
   
 dt\_list=df.datetime.to\_list()  
 duration=pd.Timedelta(dt\_list[-1]-dt\_list[0]).seconds/60.0 # min/60 hour/3600  
 t\_mode=list(df.index.get\_level\_values(1).unique())   
 duration\_df=duration\_df.append(pd.DataFrame(data={'line\_length':line\_len,  
 'duration':duration,  
 'trans\_mode':t\_mode,  
 'geometry':line,  
 'label\_idx':label\_idx}),ignore\_index=True)  
  
 duration\_gdf=gpd.GeoDataFrame(duration\_df,crs=epsg)  
   
 duration\_gdf['speed\_kmh']=duration\_gdf.apply(lambda row:(row.line\_length\*0.001)/(row.duration/60) if row.duration else None,axis=1)  
 duration\_gdf.dropna(inplace=True)  
   
 return duration\_gdf

duration\_beijing\_gdf=geolife\_traffic\_mode\_trajectory\_pts2linestring(geolife\_tm\_beijing\_gdf,args.gi.beijing\_epsg)

100%|██████████| 906/906 [00:08<00:00, 106.66it/s]

30万条GPS轨迹点数据，含905条轨迹路径，涉及到8种交通模式，有自行车（bike）、 汽车（car）、火车（train）、 出租（taxi）、 步行（walk）、公交（ bus）、地铁（ subway）和飞机（airplane）等。

print(duration\_beijing\_gdf.trans\_mode.unique())  
duration\_beijing\_gdf

['bike' 'car' 'train' 'taxi' 'walk' 'bus' 'subway' 'airplane']

|  | **line\_length** | **duration** | **trans\_mode** | **geometry** | **label\_idx** | **speed\_kmh** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | 23743.02579 | 275.666667 | bike | LINESTRING (442794.835 14425360.126, 442801.89... | 8.0 | 5.167769 |
| **1** | 21388.89143 | 385.433333 | car | LINESTRING (444065.109 14424298.068, 444086.81... | 15.0 | 3.329586 |
| **2** | 16124.881974 | 51.533333 | train | LINESTRING (442739.726 14425223.644, 442712.72... | 27.0 | 18.774119 |
| **3** | 7772.472847 | 10.533333 | taxi | LINESTRING (446221.177 14424236.293, 446315.10... | 31.0 | 44.273580 |
| **4** | 1173.303359 | 10.666667 | walk | LINESTRING (442685.941 14436392.333, 442666.03... | 32.0 | 6.599831 |
| **...** | ... | ... | ... | ... | ... | ... |
| **901** | 9810.752545 | 17.050000 | bike | LINESTRING (442144.251 14428472.684, 442139.67... | 2637.0 | 34.524642 |
| **902** | 12850.485243 | 1.750000 | walk | LINESTRING (442602.698 14425990.854, 442606.47... | 2639.0 | 440.588065 |
| **903** | 1728503.461217 | 14.016667 | train | LINESTRING (442936.982 14426445.207, 442940.23... | 2640.0 | 7399.063568 |
| **904** | 38347504.774009 | 38.483333 | bus | LINESTRING (468033.804 14418405.973, 444357.27... | 2642.0 | 59788.227452 |
| **905** | 1050753295.343658 | 186.300000 | car | LINESTRING (440082.702 14434359.278, 484599.52... | 2643.0 | 338406.858404 |

880 rows × 6 columns

按照交通模式字段trans\_mode分组计算时速均值。

trans\_mode\_speed\_mean=duration\_beijing\_gdf.groupby('trans\_mode').median()  
trans\_mode\_speed\_mean

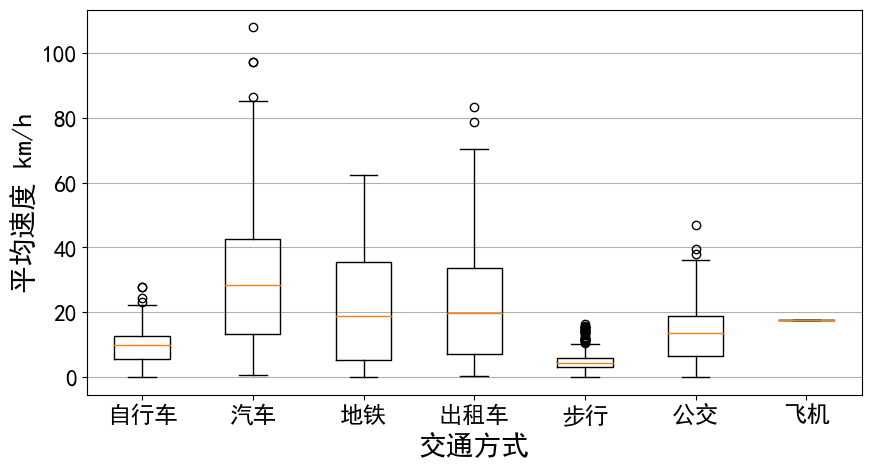
|  | **line\_length** | **duration** | **label\_idx** | **speed\_kmh** |
| --- | --- | --- | --- | --- |
| **trans\_mode** |  |  |  |  |
| **airplane** | 3223.830547 | 11.000000 | 953.0 | 17.584530 |
| **bike** | 2740.061339 | 15.466667 | 1018.0 | 11.284929 |
| **bus** | 3694.354100 | 13.850000 | 1436.0 | 15.570483 |
| **car** | 36721.506557 | 73.516667 | 416.0 | 33.920102 |
| **subway** | 5887.140773 | 15.725000 | 790.5 | 20.383329 |
| **taxi** | 7636.819993 | 14.666667 | 1174.5 | 22.150590 |
| **walk** | 1028.530414 | 11.316667 | 1263.5 | 5.009218 |

上述获取的时速中位数并未剔除异常值，因此定义geolife\_trans\_modes\_speed()函数，剔除异常值后，打印不同交通工具各个GPS轨迹时速中位数的箱型图，观察数据分布情况，并以字典形式按照交通模式返回所有数值。统计的不同交通模式的时速是实际生活上人们使用交通工具，受各种因素（例如天气、启动和停止时间、交通状况、行使状态等）影响下具有的速度，这个速度一般小于交通工具自身可以达到的速度。

def geolife\_trans\_modes\_speed(duration\_gdf,trans\_modes=None,figsize=(5,5)):  
 '''  
 配合geolife\_traffic\_mode\_trajectory\_pts2linestring()函数计算结果，移除异常值，获取不同交通模式时速的中位数，并打印箱型图观察数值分布  
  
 Parameters  
 ----------  
 duration\_gdf : GeoDataFrame  
 预处理后GeoLife数据集GPS轨迹路径数据.  
 trans\_modes : list[string], optional  
 选择用于打印箱型图的交通模式. The default is ["bus","subway","walk"].  
 figsize : tuple(int,int), optional  
 图表大小. The default is (5,5).  
  
 Returns  
 -------  
 t\_mode\_speed : dict[str:array]  
 不同交通模式各个轨迹对应的时速中位数.  
  
 '''   
 import matplotlib.pyplot as plt  
 import matplotlib  
 import numpy as np  
 import util\_misc  
   
 duration\_gdf.replace({"train":"subway"},inplace=True)  
 t\_mode=duration\_gdf.trans\_mode.unique()   
 trans\_mode\_group=duration\_gdf.groupby(by=['trans\_mode'])  
   
 t\_mode\_speed={}  
 for m in t\_mode:  
 group\_m=trans\_mode\_group.get\_group(name=m)   
 is\_outlier\_bool,data\_clean=util\_misc.is\_outlier(np.array(group\_m.speed\_kmh),threshold=3.5)  
 t\_mode\_speed[m]=np.array(data\_clean)  
   
 speed\_median={}  
 for k,v in t\_mode\_speed.items():  
 speed\_median[k]=np.median(v)  
 print(speed\_median)  
   
 matplotlib.rcParams['font.family'] = ['SimHei']  
 fig,ax=plt.subplots(figsize=figsize)   
   
 EN\_CH\_mapping={'bike': "自行车", 'car': "汽车", 'subway': "地铁", 'taxi': "出租车", 'walk':"步行", 'bus': "公交", 'airplane': "飞机"}  
 dict\_filter=lambda x, y: dict([(i,x[i]) for i in x if i in set(y)])  
 if trans\_modes:   
 t\_mode\_speed\_filter=dict\_filter(t\_mode\_speed,trans\_modes)   
 k,v=list(t\_mode\_speed\_filter.keys()),list(t\_mode\_speed\_filter.values())  
 else:  
 t\_mode\_speed\_filter=dict\_filter(t\_mode\_speed,EN\_CH\_mapping.keys())  
 k,v=list(t\_mode\_speed\_filter.keys()),list(t\_mode\_speed\_filter.values())  
 k\_CH=[EN\_CH\_mapping[i] for i in k]  
  
 ax.boxplot(v)  
 ax.yaxis.grid(True)  
 ax.set\_xticklabels(k\_CH,fontsize=17)  
 ax.set\_xlabel('交通方式',fontsize=20)  
 ax.set\_ylabel('平均速度 km/h',fontsize=20)  
 plt.yticks(fontsize=17)  
 plt.show()  
   
 return t\_mode\_speed

t\_mode\_speed=geolife\_trans\_modes\_speed(duration\_beijing\_gdf,figsize=(10,5))

{'bike': 9.993853342819856, 'car': 28.43731442778485, 'subway': 18.7683686611522, 'taxi': 19.838916253915173, 'walk': 4.386498871077352, 'bus': 13.544846491362806, 'airplane': 17.58453025807208}



### 2.8.3.2 构建公共交通复合网络

#### 1) 构建公交系统子网络

根据获取的公交站点和公交线路SHP格式数据，预处理为GeoDataFrame数据格式，用于构建公交系统子网络。复杂网络构建需要确定两两顶点之间的关系，对于公交系统子网络则需要确定同一公交线路下两两相邻站点的关系，对应构建边及增加顶点（站点）和边（线路）的有关属性。因为获取的公交线路为LINESTRING对象的空间几何对象，获取的公交站点数据站点位置不一定正好位于公交下线路上，因此根据LineUid列合并站点和线路数据后，需要找到站点到线路上的最近点，使用Shapely库的nearest\_points方法。每一条线路有多个站点，一个站点和下一个站点需要构建网络的边，抽象的边为两点间的直线，对应着实际上“曲折”的公交线路，因此需要增加forward\_length属性存储边的实际线路长度，并增加时间距离成本属性值time\_cost存储线路长度和时速的比值。定义subG\_bus()函数用于构建公交系统子网络。该函数如果用于类似城市的公交（地铁）系统网络的构建，需要将站点和线路数据处理为本次实验对应GeoDataFrame地理空间数据，且对应定义相同的列名。

bus\_stations=postSQL2gpd(table\_name='bus\_stations',\*\*args.db.db\_info)  
bus\_routes=postSQL2gpd(table\_name='bus\_routes',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is bus\_stations.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is bus\_routes.

def subG\_trans(bus\_stations\_,bus\_routes\_,speed):   
 '''  
 根据公共（地铁）站和公交（地铁）线路，及时速构建公交（地铁）系统自网络  
  
 Parameters  
 ----------  
 bus\_stations\_ : GeoDataFrame  
 公交站点地理空间数据  
 bus\_routes\_ : GeoDataFrame  
 公交线路地理空间数据  
 speed : float  
 时速 km/h.  
  
 Returns  
 -------  
 G : networkx.classes.graph.Graph  
 公交系统子网络.  
 s\_e\_nodes : list[(str,str)]  
 所有公交线路的起始点.  
 lines\_df4G : GeoDataFrame  
 用于生成公交子网络的GeoDataFrame对象.  
  
 '''   
 import copy  
 import pandas as pd  
 import networkx as nx  
 from shapely.ops import nearest\_points  
 from shapely.ops import substring  
 from tqdm import tqdm  
   
 # compute the distance between the site centroid and each bus station and get the nearest ones by given threshold  
 bus\_stations=copy.deepcopy(bus\_stations\_)  
   
 # build bus stations network  
 bus\_staions\_routes=pd.merge(bus\_stations,bus\_routes\_,on='LineUid')  
 bus\_staions\_routes\_idx\_LineUid=bus\_staions\_routes.set\_index('LineUid',append=True,drop=False)   
   
 lines\_group\_list=[]  
 s\_e\_nodes=[]  
 for LineUid,sub\_df in tqdm(bus\_staions\_routes\_idx\_LineUid.groupby(level=1)):  
 sub\_df['nearestPts']=sub\_df.apply(lambda row:nearest\_points(row.geometry\_y,row.geometry\_x)[0],axis=1)  
 sub\_df['project\_norm']=sub\_df.apply(lambda row:row.geometry\_y.project(row.nearestPts,normalized=True),axis=1)  
 sub\_df.sort\_values(by='project\_norm',inplace=True)  
 sub\_df['order\_idx']=range(1,len(sub\_df)+1)  
 project=sub\_df.project\_norm.to\_list()  
 sub\_df['second\_project']=project[1:]+project[:1]   
 PointName=sub\_df.PointName.to\_list()  
 sub\_df['second\_PointName']=PointName[1:]+PointName[:1]  
 PointUid=sub\_df.PointUid.to\_list()  
 sub\_df['second\_PointUid']= PointUid[1:]+ PointUid[:1]  
 sub\_df['substring']=sub\_df.apply(lambda row:substring(row.geometry\_y,row.project\_norm,row.second\_project,normalized=True),axis=1)  
 sub\_df['forward\_length']=sub\_df.apply(lambda row:row.substring.length,axis=1)  
 sub\_df['time\_cost']=sub\_df.apply(lambda row:row.forward\_length/(speed\*1000)\*60,axis=1)   
 sub\_df['edges']=sub\_df.apply(lambda row:[(row.PointUid,row.second\_PointUid),(row.second\_PointUid,row.PointUid)],axis=1)   
 lines\_group\_list.append(sub\_df)  
   
 if sub\_df.shape[0]>2:  
 s\_e\_nodes.append(sub\_df.edges.to\_list()[-1][0])  
  
 lines\_df4G=pd.concat(lines\_group\_list)  
  
 G=nx.from\_pandas\_edgelist(df=lines\_df4G,source='PointUid',target='second\_PointUid',edge\_attr=['PointName','second\_PointName','forward\_length','geometry\_x','time\_cost'])  
 G.remove\_edges\_from(s\_e\_nodes)  
   
 for idx,row in lines\_df4G.iterrows():  
 G.nodes[row['PointUid']]['position']=(row.geometry\_x.x,row.geometry\_x.y)  
 G.nodes[row['PointUid']]['station\_name']=row.PointName   
   
 return G,s\_e\_nodes,lines\_df4G

G\_bus\_stations,s\_e\_nodes,bus\_lines\_df4G=subG\_trans(bus\_stations,bus\_routes,speed=13.54)

100%|██████████| 1283/1283 [00:37<00:00, 34.42it/s]

查看用于构建复杂网络处理后的GeoDataFrame地理空间数据。

|  |  | **PointName** | **PointUid** | **Lng** | **Lat** | **LineName\_x** | **LineUid** | **geometry\_x** | **mask\_x** | **LineName\_y** | **StartTime** | **...** | **nearestPts** | **project\_norm** | **order\_idx** | **second\_project** | **second\_PointName** | **second\_PointUid** | **substring** | **forward\_length** | **time\_cost** | **edges** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **LineUid** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **2261** | **000ec915034f2e3f58bbbc5c** | 尧佳路东 | 098d2a5e146f358c250bd02d | 118.892640622249 | 32.1301950057802 | 140路(尧佳路东-白马公园) | 000ec915034f2e3f58bbbc5c | POINT (678531.157 3556435.574) | True | 140路(尧佳路东-白马公园) | 05:40 | ... | POINT (678531.157 3556435.574) | 0.000000 | 1 | 0.059152 | 新城金郡 | 85aca85907cc7b0c210262fb | LINESTRING (678531.157 3556435.574, 678531.402... | 863.631863 | 3.827025 | [(098d2a5e146f358c250bd02d, 85aca85907cc7b0c21... |
| **2262** | **000ec915034f2e3f58bbbc5c** | 新城金郡 | 85aca85907cc7b0c210262fb | 118.887909509396 | 32.1318136548581 | 140路(尧佳路东-白马公园) | 000ec915034f2e3f58bbbc5c | POINT (678081.658 3556607.200) | True | 140路(尧佳路东-白马公园) | 05:40 | ... | POINT (678081.658 3556607.200) | 0.059152 | 2 | 0.118442 | 前新塘 | 4b350e936a57861914c8f63b | LINESTRING (678081.658 3556607.200, 678020.195... | 865.657054 | 3.835999 | [(85aca85907cc7b0c210262fb, 4b350e936a57861914... |
| **2263** | **000ec915034f2e3f58bbbc5c** | 前新塘 | 4b350e936a57861914c8f63b | 118.884917011823 | 32.125994898654 | 140路(尧佳路东-白马公园) | 000ec915034f2e3f58bbbc5c | POINT (677810.631 3555957.124) | True | 140路(尧佳路东-白马公园) | 05:40 | ... | POINT (677810.631 3555957.124) | 0.118442 | 3 | 0.154839 | 太龙路东 | 2881003e7516d4f130169bc7 | LINESTRING (677810.631 3555957.124, 677808.033... | 531.406948 | 2.354831 | [(4b350e936a57861914c8f63b, 2881003e7516d4f130... |

3 rows × 26 columns

查看公交系统子网络顶点和边的属性内容。

print(G\_bus\_stations.nodes['098d2a5e146f358c250bd02d'])  
print(G\_bus\_stations.edges['098d2a5e146f358c250bd02d', '85aca85907cc7b0c210262fb'])

{'position': (678531.1572245015, 3556435.5737954923), 'station\_name': '尧佳路东'}  
{'PointName': '尧佳路东', 'second\_PointName': '新城金郡', 'forward\_length': 863.6318633109651, 'geometry\_x': <POINT (678531.157 3556435.574)>, 'time\_cost': 3.8270245050707468}

打印公交系统子网络。

import matplotlib.pyplot as plt  
plt.rcParams['font.sans-serif']=['DengXian'] # 指定默认字体 'KaiTi','SimHei'  
plt.rcParams['axes.unicode\_minus']=False # 解决保存图像是负号'-'显示为方块的问题  
  
fig, ax=plt.subplots(figsize=(30\*2\*2, 30\*4\*2))  
nx.draw(G\_bus\_stations,pos=nx.get\_node\_attributes(G\_bus\_stations,'position'),edge\_color='black',labels=nx.get\_node\_attributes(G\_bus\_stations,'station\_name'),font\_size=20,node\_size=50,ax=ax)



为方便后续应用公交系统子网络进行相关分析，将其存储到本地磁盘中。

import pickle  
\_\_C.data.G\_bus\_stations="./data/network/G\_bus\_stations.gpickle"  
with open(args.data.G\_bus\_stations,'wb') as f:  
 pickle.dump(G\_bus\_stations, f)

#### 2) 构建公交系统的换乘子网络

一个地点附近可能会有多个线路交会，存在不同线路的站点，因此如果存在换乘，则需要考虑站点之间的距离，因此增加公交系统的换乘子网络，定义transfer\_stations\_network()函数实现。换乘距离transfer\_distance参数，表述了换乘所能承受的最大距离，如果超过该距离，则不计入公交系统的换乘子网络。

实际上人们换乘所消耗的时间，不仅是路程距离的时间，还包括等待的时间；且这个距离可能并不是直线距离，因此公共交通复合网络的构建是一般情况的表述，为一个参照基础。

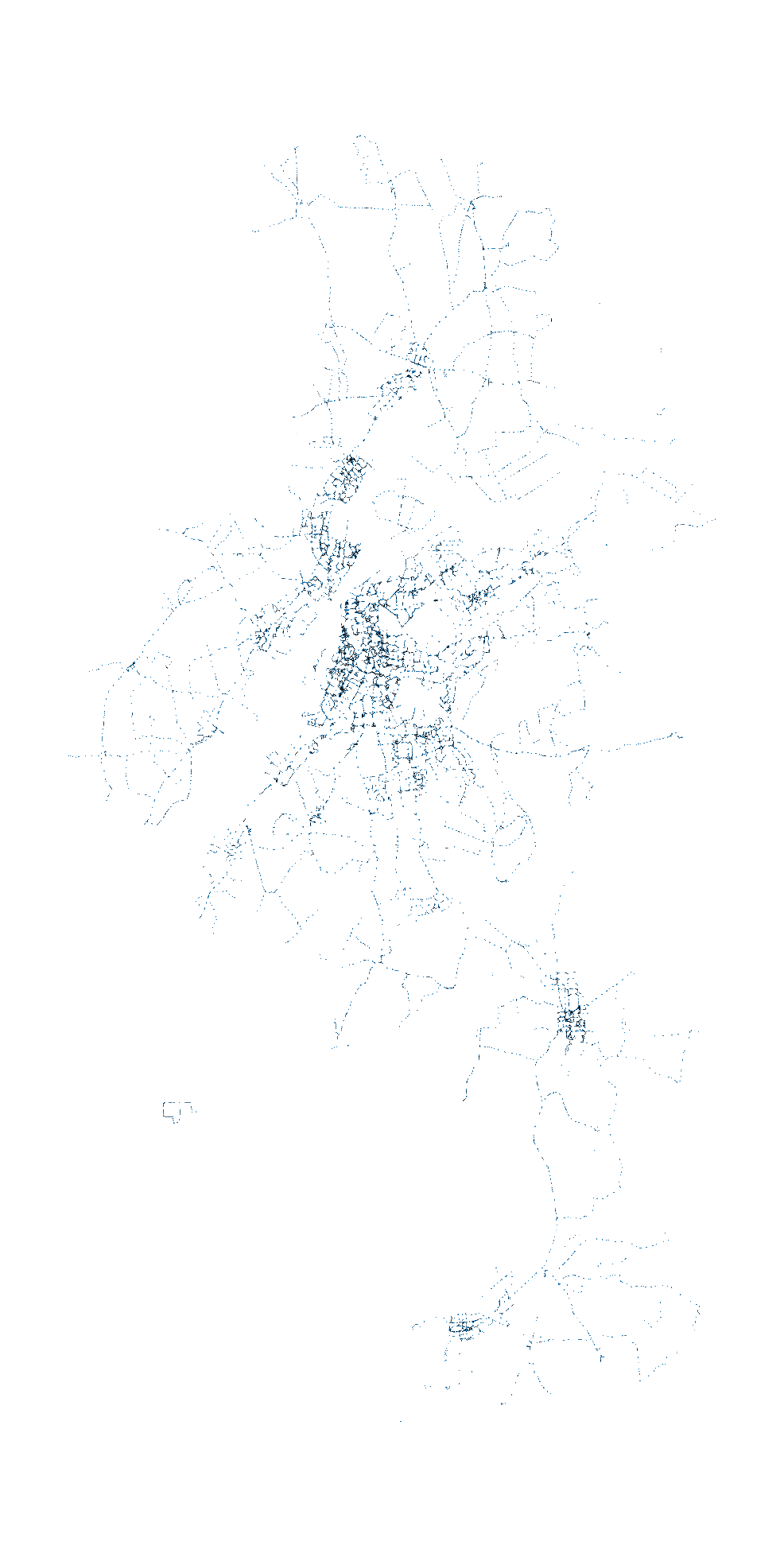
def transfer\_stations\_network(station\_geometries\_df,transfer\_distance,speed):   
 '''  
 构建公交系统的换乘子网络  
  
 Parameters  
 ----------  
 station\_geometries\_df : GeoDataFrame  
 subG\_trans函数（用于构建公共交通（公交或地铁）子网络）返回的结果.  
 transfer\_distance : float  
 换乘所能承受的最大距离.  
 speed : float  
 步行时速.  
  
 Returns  
 -------  
 G : networkx.classes.graph.Graph  
 公交系统的换乘子网络.  
 transfer\_df\_concat : DataFrame  
 用于生成公交换乘子网络的DataFrame对象.  
  
 '''   
 import copy  
 from tqdm import tqdm  
 import pandas as pd  
 import networkx as nx  
   
 station\_geometries\_dict=station\_geometries\_df.to\_dict('record')  
 transfer\_df\_list=[]  
 for pt in tqdm(station\_geometries\_dict):  
 station\_geometries\_df\_=copy.deepcopy(station\_geometries\_df)  
 station\_geometries\_df\_['dist']=station\_geometries\_df\_.geometry\_x.apply(lambda row:row.distance(pt['geometry\_x']))  
  
 transfer\_df=station\_geometries\_df\_[station\_geometries\_df\_.dist<=transfer\_distance]  
 transfer\_df=transfer\_df[transfer\_df.dist!=0]  
 transfer\_df.drop\_duplicates(subset='PointUid',keep='first',inplace=True)   
 transfer\_df['source\_station']=pt['PointUid']  
 transfer\_df['forward\_length']=transfer\_df.dist   
 transfer\_df=transfer\_df[transfer\_df.LineUid!=pt['LineUid']]   
 transfer\_df\_list.append(transfer\_df)  
   
 transfer\_df\_concat=pd.concat(transfer\_df\_list)   
 transfer\_df\_concat['time\_cost']=transfer\_df\_concat.apply(lambda row:row.forward\_length/(speed\*1000)\*60,axis=1)   
   
 G=nx.from\_pandas\_edgelist(df=transfer\_df\_concat,source='source\_station',target='PointUid',edge\_attr=['forward\_length','time\_cost'])  
   
 for idx,row in transfer\_df\_concat.iterrows():  
 G.nodes[row['PointUid']]['position']=(row.geometry\_x.x,row.geometry\_x.y)  
  
 return G,transfer\_df\_concat

station\_geometries\_df=lines\_df4G[['PointUid','geometry\_x','LineUid']]   
G\_bus\_transfer,transfer\_df\_concat=transfer\_stations\_network(station\_geometries\_df,transfer\_distance=400,speed=4.39)

100%|██████████| 31977/31977 [3:19:17<00:00, 2.67it/s]

打印公交系统的换乘子网络。

fig, ax=plt.subplots(figsize=(30\*2\*2, 30\*4\*2))  
nx.draw(G\_bus\_transfer,pos=nx.get\_node\_attributes(G\_bus\_transfer,'position'),edge\_color='black',node\_size=50,ax=ax)



将公交系统的换乘子网络存入本地磁盘。

\_\_C.data.G\_bus\_ransfer="./data/network/G\_bus\_transfer.gpickle"  
with open(args.data.G\_bus\_ransfer,'wb') as f:  
 pickle.dump(G\_bus\_transfer, f)

#### 3) 构建地铁系统子网络

同公交系统子网络的构建，直接调用已经定义的subG\_trans()函数，生成地铁系统子网络，并存储到本地磁盘。

subway\_stations=postSQL2gpd(table\_name='subway\_stations',\*\*args.db.db\_info)  
subway\_routes=postSQL2gpd(table\_name='subway\_lines',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is subway\_stations.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is subway\_lines.

地铁站点的PointUid字段ID标识和公交站点的PointUid字段ID标识可能存在相同的ID号（可能位于同一位置），为了区分地铁和公交的ID标识，将地铁的PointUid字段ID标识前增加字符s\_作为区分。

subway\_stations['PointUid']=subway\_stations.PointUid.apply(lambda row:'s\_'+row)

G\_subway\_stations,subway\_s\_e\_nodes,subway\_lines\_df4G=subG\_trans(subway\_stations,subway\_routes,speed=18.77)

100%|██████████| 20/20 [00:00<00:00, 43.89it/s]

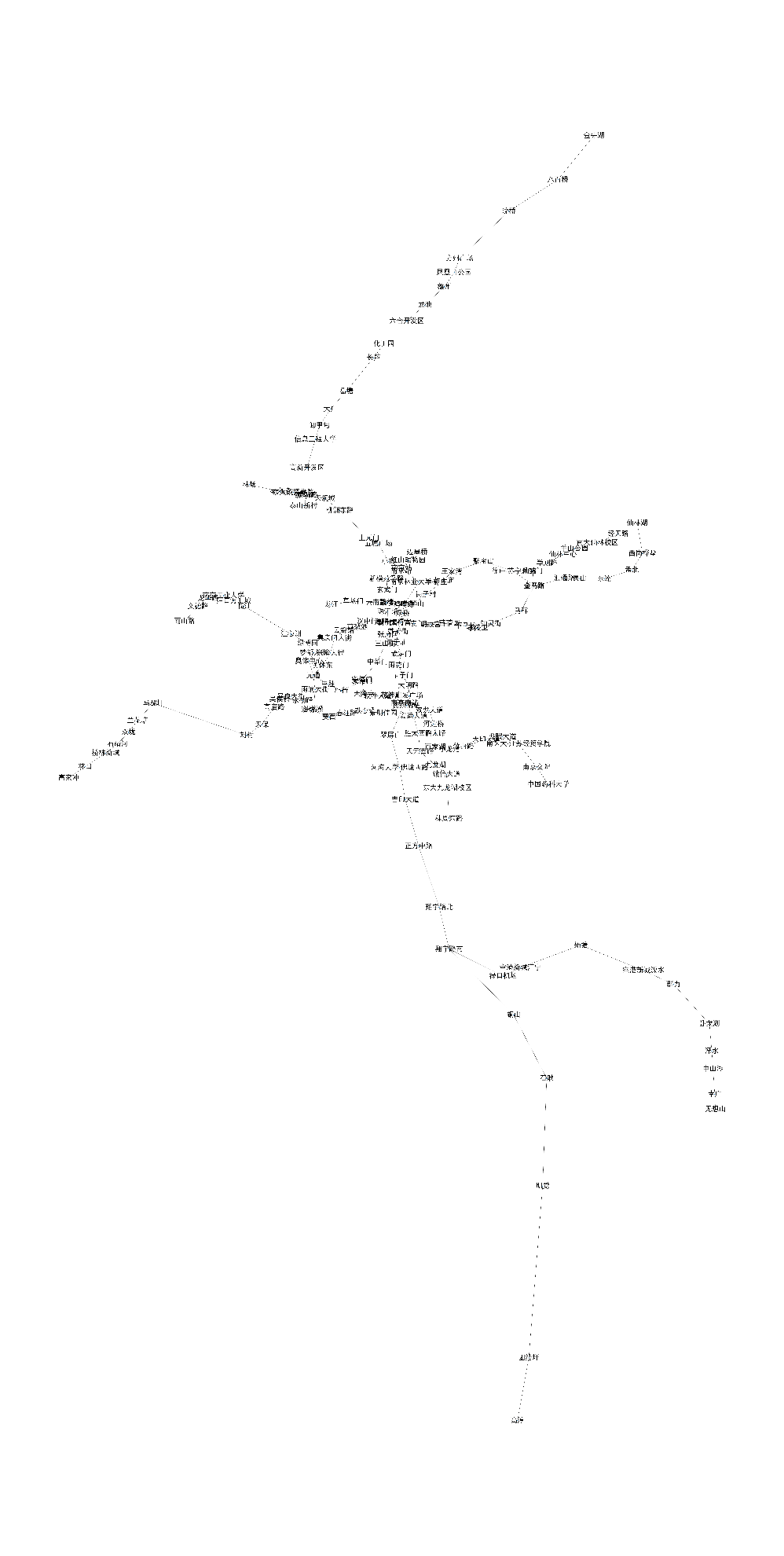
subway\_lines\_df4G.head(3)

|  |  | **PointName** | **PointUid** | **Lng** | **Lat** | **IsPractica** | **LineName\_x** | **LineUid** | **geometry\_x** | **mask\_x** | **LineName\_y** | **...** | **nearestPts** | **project\_norm** | **order\_idx** | **second\_project** | **second\_PointName** | **second\_PointUid** | **substring** | **forward\_length** | **time\_cost** | **edges** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **LineUid** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **315** | **031d4680f2fa3654e60738d4** | 安德门 | s\_26f430f612ec6cd121f01731 | 118.75688738084 | 31.992830651745 | 1 | 地铁10号线(安德门-雨山路) | 031d4680f2fa3654e60738d4 | POINT (665971.810 3540989.467) | True | 地铁10号线(安德门-雨山路) | ... | POINT (665971.810 3540989.467) | 0.000000 | 1 | 0.095246 | 小行 | s\_153f5f1301157bda4c3a595b | LINESTRING (665971.810 3540989.467, 664311.516... | 1917.170671 | 6.128409 | [(s\_26f430f612ec6cd121f01731, s\_153f5f1301157b... |
| **316** | **031d4680f2fa3654e60738d4** | 小行 | s\_153f5f1301157bda4c3a595b | 118.739156216275 | 31.9844281695514 | 0 | 地铁10号线(安德门-雨山路) | 031d4680f2fa3654e60738d4 | POINT (664311.516 3540030.840) | True | 地铁10号线(安德门-雨山路) | ... | POINT (664311.516 3540030.840) | 0.095246 | 2 | 0.155467 | 中胜 | s\_c67ae2d9d96e3b29df1e52e2 | LINESTRING (664311.516 3540030.840, 663273.829... | 1212.160321 | 3.874780 | [(s\_153f5f1301157bda4c3a595b, s\_c67ae2d9d96e3b... |
| **317** | **031d4680f2fa3654e60738d4** | 中胜 | s\_c67ae2d9d96e3b29df1e52e2 | 118.728283054112 | 31.9902280445245 | 0 | 地铁10号线(安德门-雨山路) | 031d4680f2fa3654e60738d4 | POINT (663273.829 3540657.368) | True | 地铁10号线(安德门-雨山路) | ... | POINT (663273.829 3540657.368) | 0.155467 | 3 | 0.225662 | 元通 | s\_918fd0373e1f728cf33cbad7 | LINESTRING (663273.829 3540657.368, 662128.887... | 1412.925204 | 4.516543 | [(s\_c67ae2d9d96e3b29df1e52e2, s\_918fd0373e1f72... |

3 rows × 22 columns

打印地铁系统子网络。

fig, ax=plt.subplots(figsize=(30\*2\*2, 30\*4\*2))  
nx.draw(G\_subway\_stations,pos=nx.get\_node\_attributes(G\_subway\_stations,'position'),edge\_color='black',labels=nx.get\_node\_attributes(G\_subway\_stations,'station\_name'),font\_size=60,node\_size=50,ax=ax)



\_\_C.data.G\_subway\_stations="./data/network/G\_subway\_stations.gpickle"  
with open(args.data.G\_subway\_stations,'wb') as f:  
 pickle.dump(G\_subway\_stations, f)

#### 4) 构建公交-地铁系统换乘子网络

除了构建公交系统内部的换乘子网络，公交系统和地铁系统之间同样存在换乘，因此构建公交-地铁系统换乘子网络，定义transfer\_stations\_network4subway\_bus()函数实现，该函数与定义的transfer\_stations\_network()函数基本类似。同样将计算获取的公交-地铁系统换乘子网络存至本地磁盘。

def transfer\_stations\_network4subway\_bus(bus\_df,subway\_df,transfer\_distance,speed):   
 '''  
 构建公交-地铁系统换乘子网络  
  
 Parameters  
 ----------  
 bus\_df : GeoDataFrame  
 subG\_trans函数（用于构建公共交通（公交或地铁）子网络）返回的结果——为公交系统返回结果.  
 subway\_df : GeoDataFrame  
 subG\_trans函数（用于构建公共交通（公交或地铁）子网络）返回的结果——为地铁系统返回结果.  
 transfer\_distance : float  
 换乘所能承受的最大距离.  
 speed : float  
 步行时速.  
  
 Returns  
 -------  
 G : networkx.classes.graph.Graph  
 公交-地铁系统换乘子网络.  
 transfer\_df\_concat : DataFrame  
 用于生成公交-地铁系统换乘子网络的DataFrame对象.  
  
 '''   
 import copy  
 from tqdm import tqdm  
 import pandas as pd  
 import networkx as nx  
   
 transfer\_df\_list=[]  
 bus\_stations\_=bus\_df[['PointUid','geometry\_x','LineUid']]  
 subway\_station\_=subway\_df[['PointUid','geometry\_x','LineUid']]  
 subway\_station=subway\_station\_.to\_dict('record')  
 for pt in tqdm(subway\_station):  
 bus\_stations=copy.deepcopy(bus\_stations\_)  
 bus\_stations['dist']=bus\_stations.geometry\_x.apply(lambda row:row.distance(pt['geometry\_x']))  
 transfer\_df=bus\_stations[bus\_stations.dist<=transfer\_distance]  
 transfer\_df=transfer\_df[transfer\_df.dist!=0]  
 transfer\_df.drop\_duplicates(subset='PointUid',keep='first',inplace=True)   
 transfer\_df['source\_station']=pt['PointUid']  
 transfer\_df['forward\_length']=transfer\_df.dist  
 transfer\_df\_list.append(transfer\_df)  
  
 transfer\_df\_concat=pd.concat(transfer\_df\_list)  
 transfer\_df\_concat['time\_cost']=transfer\_df\_concat.apply(lambda row:row.forward\_length/(speed\*1000)\*60,axis=1)  
 G=nx.from\_pandas\_edgelist(df=transfer\_df\_concat,source='source\_station',target='PointUid',edge\_attr=['forward\_length','time\_cost'])  
   
 for node in tqdm(G.nodes):   
 if node in bus\_stations\_.PointUid.to\_list():  
 pt=bus\_stations\_.loc[bus\_stations\_['PointUid']==node,'geometry\_x'].item()  
 G.nodes[node]['position']=(pt.x,pt.y)  
   
 elif node in subway\_station\_.PointUid.to\_list():  
 pt=subway\_station\_.loc[subway\_station\_['PointUid']==node,'geometry\_x'].item()  
 G.nodes[node]['position']=(pt.x,pt.y)   
   
 return G,transfer\_df\_concat

G\_subway\_bus\_transfer,subway\_bus\_transfer\_df\_concat=transfer\_stations\_network4subway\_bus(bus\_df=bus\_lines\_df4G,subway\_df=subway\_lines\_df4G,transfer\_distance=400,speed=4.39)

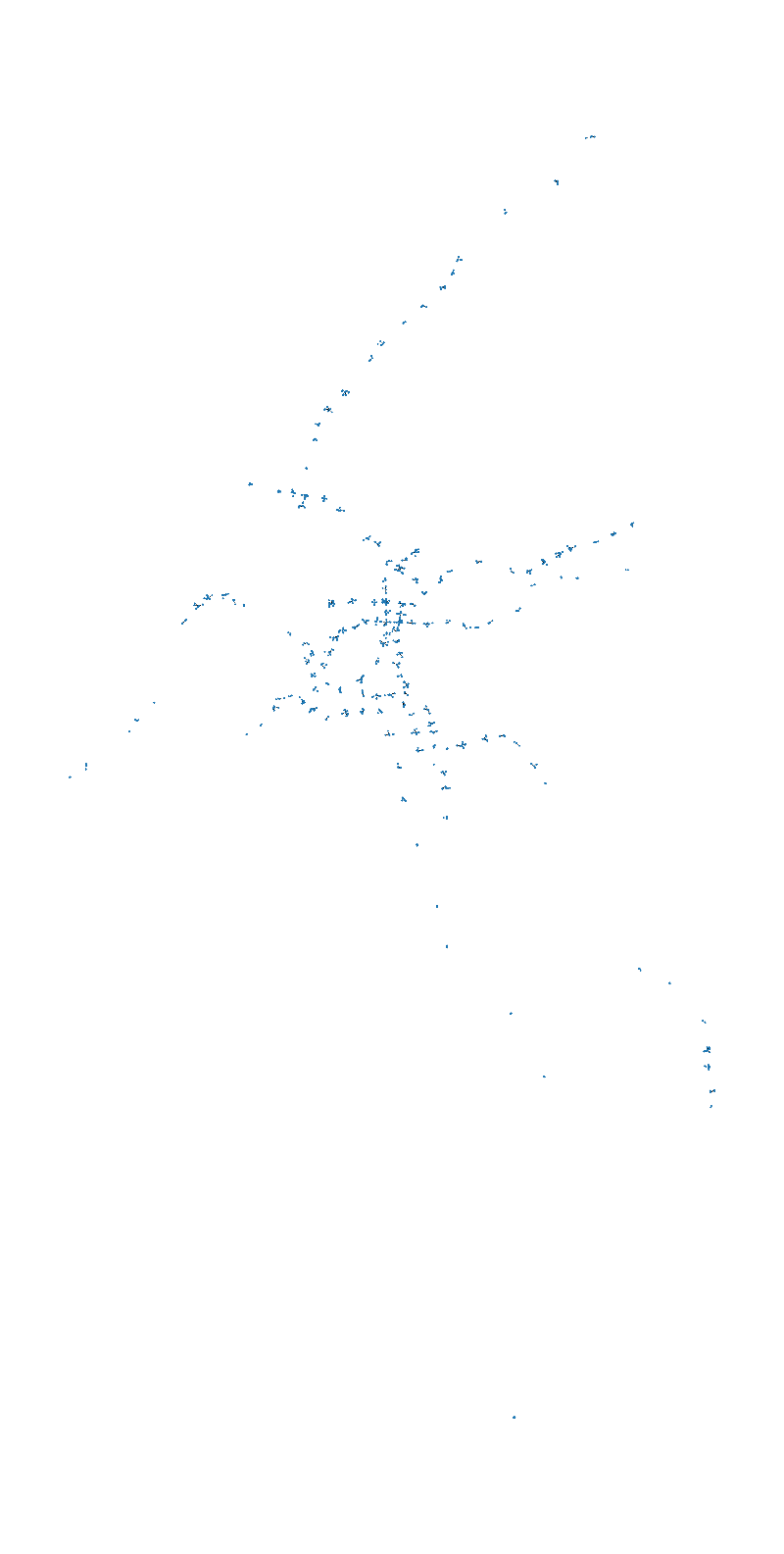
100%|██████████| 348/348 [02:42<00:00, 2.15it/s]  
100%|██████████| 4444/4444 [00:19<00:00, 222.55it/s]

subway\_bus\_transfer\_df\_concat.head(3)

|  |  | **PointUid** | **geometry\_x** | **LineUid** | **dist** | **source\_station** | **forward\_length** | **time\_cost** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **LineUid** |  |  |  |  |  |  |  |
| **22404** | **01000db975471b8cb64f54a7** | d3b6395653cc1c80601c67de | POINT (666026.030 3541291.205) | 01000db975471b8cb64f54a7 | 306.570722 | s\_26f430f612ec6cd121f01731 | 306.570722 | 4.190033 |
| **6506** | **01a59c056cf31073266e9810** | e9de6509923177582bb97c50 | POINT (665686.091 3540973.057) | 01a59c056cf31073266e9810 | 286.190597 | s\_26f430f612ec6cd121f01731 | 286.190597 | 3.911489 |
| **13809** | **01c0f536aa9099c130621272** | 6c9c9588865d300f9d82eac9 | POINT (666031.407 3541299.839) | 01c0f536aa9099c130621272 | 316.041467 | s\_26f430f612ec6cd121f01731 | 316.041467 | 4.319473 |

打印公交-地铁系统换乘子网络。

fig, ax=plt.subplots(figsize=(30\*2, 30\*4))  
nx.draw(G\_subway\_bus\_transfer,pos=nx.get\_node\_attributes(G\_subway\_bus\_transfer,'position'),edge\_color='black',node\_size=50,ax=ax)



\_\_C.data.G\_subway\_bus\_transfer="./data/network/G\_subway\_bus\_transfer.gpickle"  
with open(args.data.G\_subway\_bus\_transfer,'wb') as f:  
 pickle.dump(G\_subway\_bus\_transfer, f)

#### 5) 合并子网络，构建公共交通复合网络

通过前文已经构建了公交系统子网络，公交系统的换乘子网络，地铁系统子网络和公交-地铁系统换乘子网络，可以将各个子网络合并为一个公共交通复合网络。NetWorkX库提供的compose(G, H)方法每次只能合并两个子网络，因此先分别合并为G\_bus和G\_subway，再合并为最终的复合网络。将合并后的公共交通复合网络同样存储至本地磁盘，用于后续调用分析。

G\_bus=nx.compose(G\_bus\_stations,G\_bus\_transfer)   
G\_subway=nx.compose(G\_subway\_stations,G\_subway\_bus\_transfer)  
G\_SB=nx.compose(G\_subway,G\_bus)

\_\_C.data.G\_SB="./data/network/G\_SB.gpickle"  
with open(args.data.G\_SB,'wb') as f:  
 pickle.dump(G\_SB, f)

* 打印公共交通复合网络

总共建立了4个子网络，和一个合并4个子网络后的公共交通复合网络，均已存储至本地磁盘，因此可以读取。为了方便查看网络不同内容的分布情况，可以叠加打印不同部分，例如下述打印了复合网络后，又叠加打印了G\_bus\_transfer公交换乘子网络的边（红色），G\_subway\_stations地铁子网络的边，和公园邻近的站点G\_start\_stops。

G\_fns={'G\_SB':args.data.G\_SB,  
 'G\_bus\_stations':args.data.G\_bus\_stations,  
 'G\_bus\_transfer':args.data.G\_bus\_ransfer,  
 'G\_subway\_stations':args.data.G\_subway\_stations,  
 'G\_subway\_bus\_transfe':args.data.G\_subway\_bus\_transfer}  
  
Gs={}  
for k,v in G\_fns.items():  
 with open(v,'rb') as f:  
 Gs[k]=pickle.load(f)

flatten\_lst=lambda lst: [m for n\_lst in lst for m in flatten\_lst(n\_lst)] if type(lst) is list else [lst]   
  
comprehensivePark\_adjacentStations=postSQL2gpd(table\_name='adjacent\_stations',\*\*args.db.db\_info)  
start\_stops\_PointUid=flatten\_lst([eval(lst) for lst in comprehensivePark\_adjacentStations.adjacent\_PointUid])  
G\_bus\_stations=Gs['G\_bus\_stations']  
G\_start\_stops=G\_bus\_stations.subgraph(start\_stops\_PointUid)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is adjacent\_stations.

fig, ax=plt.subplots(figsize=(30\*2\*2, 30\*4\*2))  
G\_name='G\_SB'  
nx.draw(Gs['%s'%G\_name],  
 pos=nx.get\_node\_attributes(Gs['%s'%G\_name],'position'),  
 edge\_color='black',  
 font\_size=20,  
 node\_size=50,  
 ax=ax)   
nx.draw\_networkx\_edges(Gs['G\_bus\_transfer'],  
 pos=nx.get\_node\_attributes(Gs['G\_bus\_transfer'],'position'),  
 edgelist=Gs['G\_bus\_transfer'].edges,  
 edge\_color='red',  
 width=5,  
 ax=ax)   
nx.draw\_networkx\_edges(Gs['G\_subway\_stations'],  
 pos=nx.get\_node\_attributes(Gs['G\_subway\_stations'],'position'),  
 edgelist=Gs['G\_subway\_stations'].edges,  
 edge\_color='orange',  
 width=10,  
 ax=ax)  
nx.draw\_networkx\_nodes(G\_start\_stops,  
 pos=nx.get\_node\_attributes(G\_start\_stops,'position'),  
 node\_size=160,  
 ax=ax,  
 node\_color='green')  
plt.show()



### 2.8.3.3 最短路径及长度

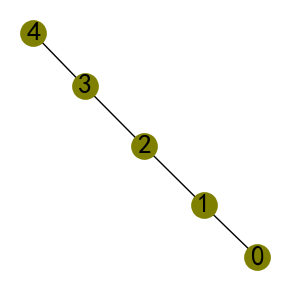
#### 1) [NetWorkX库的最短路径算法](https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html)③

* Shortest Paths（最短路径）

计算图中顶点之间的最短路径和路径长度（该部分算法适用于无向图和有向图）。

shortest\_path(G, source=None, target=None, weight=None, method=‘dijkstra’)：计算图中顶点间的最短路径。

import networkx as nx  
import util\_misc  
  
G = nx.path\_graph(5)  
util\_misc.G\_drawing(G)  
print(nx.shortest\_path(G, source=0, target=4))  
  
print(nx.shortest\_path(G, source=0)) # target not specified  
print(nx.shortest\_path(G, target=4)) # source not specified  
print(nx.shortest\_path(G)) # source, target not specified



[0, 1, 2, 3, 4]  
{0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 1, 2, 3], 4: [0, 1, 2, 3, 4]}  
{4: [4], 3: [3, 4], 2: [2, 3, 4], 1: [1, 2, 3, 4], 0: [0, 1, 2, 3, 4]}  
{0: {0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 1, 2, 3], 4: [0, 1, 2, 3, 4]}, 1: {1: [1], 0: [1, 0], 2: [1, 2], 3: [1, 2, 3], 4: [1, 2, 3, 4]}, 2: {2: [2], 1: [2, 1], 3: [2, 3], 0: [2, 1, 0], 4: [2, 3, 4]}, 3: {3: [3], 2: [3, 2], 4: [3, 4], 1: [3, 2, 1], 0: [3, 2, 1, 0]}, 4: {4: [4], 3: [4, 3], 2: [4, 3, 2], 1: [4, 3, 2, 1], 0: [4, 3, 2, 1, 0]}}

shortest\_path\_length(G, source=None, target=None, weight=None, method=‘dijkstra’)：计算图中顶点间的最短路径长度。其中method参数支持算法dijkstra和bellman-ford。

print(nx.shortest\_path\_length(G, source=0, target=4))  
  
print(nx.shortest\_path\_length(G, source=0)) # target not specified  
print(nx.shortest\_path\_length(G, target=4)) # source not specified  
print(dict(nx.shortest\_path\_length(G))) # source,target not specified

4  
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}  
{4: 0, 3: 1, 2: 2, 1: 3, 0: 4}  
{0: {0: 0, 1: 1, 2: 2, 3: 3, 4: 4}, 1: {1: 0, 0: 1, 2: 1, 3: 2, 4: 3}, 2: {2: 0, 1: 1, 3: 1, 0: 2, 4: 2}, 3: {3: 0, 2: 1, 4: 1, 1: 2, 0: 3}, 4: {4: 0, 3: 1, 2: 2, 1: 3, 0: 4}}

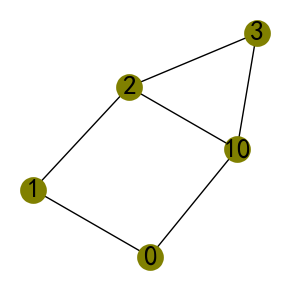
average\_shortest\_path\_length(G, weight=None, method=None:返回所有两点间最短路径的平均长度，计算公式为，式中，是图的顶点集；是顶点到的最短路径；是图的阶（顶点数量）。

nx.average\_shortest\_path\_length(G)

2.0

all\_shortest\_paths(G, source, target, weight=None, method=‘dijkstra’)：计算图中给定两个顶点间所有最短路径。

G = nx.Graph()  
nx.add\_path(G, [0, 1, 2])  
nx.add\_path(G, [0, 10, 2])  
nx.add\_path(G, [0, 10, 3,2])  
util\_misc.G\_drawing(G)  
print([p for p in nx.all\_shortest\_paths(G, source=0, target=2)])



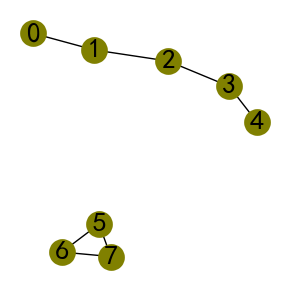
[[0, 1, 2], [0, 10, 2]]

* Advanced Interface（高级交互）

**无权重最短路径算法**

single\_source\_shortest\_path(G, source, cutoff=None)：计算从给定顶点到其它所有可达顶点之间的最短路径。

G=nx.path\_graph(5)  
G.add\_edges\_from([(5,6),(5,7),(6,7)])  
util\_misc.G\_drawing(G)  
print(nx.single\_source\_shortest\_path(G, 0))



{0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 1, 2, 3], 4: [0, 1, 2, 3, 4]}

single\_source\_shortest\_path\_length(G, source, cutoff=None)：计算从给定顶点到其它所有可达顶点之间的最短路径长度。

print(nx.single\_source\_shortest\_path\_length(G, 0))

{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}

single\_target\_shortest\_path(G, target, cutoff=None)：计算可到给定顶点所有其它顶点之间的最短路径。

print(nx.single\_target\_shortest\_path(G, 4))

{4: [4], 3: [3, 4], 2: [2, 3, 4], 1: [1, 2, 3, 4], 0: [0, 1, 2, 3, 4]}

single\_target\_shortest\_path\_length(G, target, cutoff=None)：计算可到给定顶点所有其它顶点之间的最短路径长度。

print(dict(nx.single\_target\_shortest\_path\_length(G, 4)))

{4: 0, 3: 1, 2: 2, 1: 3, 0: 4}

all\_pairs\_shortest\_path(G, cutoff=None)：计算所有顶点间最短路径。

print(dict(nx.all\_pairs\_shortest\_path(G)))

{0: {0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 1, 2, 3], 4: [0, 1, 2, 3, 4]}, 1: {1: [1], 0: [1, 0], 2: [1, 2], 3: [1, 2, 3], 4: [1, 2, 3, 4]}, 2: {2: [2], 1: [2, 1], 3: [2, 3], 0: [2, 1, 0], 4: [2, 3, 4]}, 3: {3: [3], 2: [3, 2], 4: [3, 4], 1: [3, 2, 1], 0: [3, 2, 1, 0]}, 4: {4: [4], 3: [4, 3], 2: [4, 3, 2], 1: [4, 3, 2, 1], 0: [4, 3, 2, 1, 0]}, 5: {5: [5], 6: [5, 6], 7: [5, 7]}, 6: {6: [6], 5: [6, 5], 7: [6, 7]}, 7: {7: [7], 5: [7, 5], 6: [7, 6]}}

all\_pairs\_shortest\_path\_length(G, cutoff=None)：计算所有顶点间最短路径长度。

print(dict(nx.all\_pairs\_shortest\_path\_length(G)))

{0: {0: 0, 1: 1, 2: 2, 3: 3, 4: 4}, 1: {1: 0, 0: 1, 2: 1, 3: 2, 4: 3}, 2: {2: 0, 1: 1, 3: 1, 0: 2, 4: 2}, 3: {3: 0, 2: 1, 4: 1, 1: 2, 0: 3}, 4: {4: 0, 3: 1, 2: 2, 1: 3, 0: 4}, 5: {5: 0, 6: 1, 7: 1}, 6: {6: 0, 5: 1, 7: 1}, 7: {7: 0, 5: 1, 6: 1}}

predecessor(G, source, target=None, cutoff=None, return\_seen=None)：返回给定顶点到所有顶点最短路径的前一顶点。如果指定参数return\_seen=True，同时返回以可达顶点为键，以到达该顶点的级（level）/跳（hops）数为值。

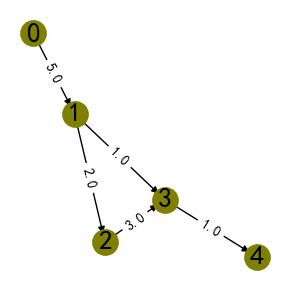
print(nx.predecessor(G, 0))  
print(nx.predecessor(G, 0, return\_seen=True))

{0: [], 1: [0], 2: [1], 3: [2], 4: [3]}  
({0: [], 1: [0], 2: [1], 3: [2], 4: [3]}, {0: 0, 1: 1, 2: 2, 3: 3, 4: 4})

**有权重最短路径算法**

dijkstra\_predecessor\_and\_distance(G, source, cutoff=None, weight=‘weight’)：计算加权最短路径长度，并返回前一顶点。

G = nx.path\_graph(5, create\_using=nx.DiGraph())  
G.add\_edge(1,3)  
nx.set\_edge\_attributes(G,  
 {  
 (0,1): {"weight": 5.0},  
 (1,2): {"weight": 2.0},  
 (2,3): {"weight": 3.0},  
 (3,4): {"weight": 1.0},  
 (1,3): {"weight": 1.0},  
 })  
util\_misc.G\_drawing(G,edge\_labels='weight')  
pred, dist = nx.dijkstra\_predecessor\_and\_distance(G, 0)   
print(sorted(pred.items()))  
print(sorted(dist.items()))



[(0, []), (1, [0]), (2, [1]), (3, [1]), (4, [3])]  
[(0, 0), (1, 5.0), (2, 7.0), (3, 6.0), (4, 7.0)]

dijkstra\_path(G, source, target, weight=‘weight’)：返回图G两点间最短加权路径。

print(nx.dijkstra\_path(G, 0, 4))

[0, 1, 3, 4]

dijkstra\_path\_length(G, source, target, weight=‘weight’)：返回图G两点间最短加权路径长度。

print(nx.dijkstra\_path\_length(G, 0, 4))

7.0

single\_source\_dijkstra(G, source, target=None, cutoff=None, weight=‘weight’：计算从给定顶点到其它所有可达顶点之间的最短加权路径和长度。

如果要单独返回最短加权路径和长度，则可以调用single\_source\_dijkstra\_path(G, source, cutoff=None, weight=‘weight’)和single\_source\_dijkstra\_path\_length(G, source, cutoff=None, weight=‘weight’)。如果要同时计算给定多个顶点（仅返回最短路径的顶点对应的路径和长度），则可以调用multi\_source\_dijkstra(G, sources, target=None, cutoff=None, weight=‘weight’)、multi\_source\_dijkstra\_path(G, sources, cutoff=None, weight=‘weight’)和multi\_source\_dijkstra\_path\_length(G, sources, cutoff=None, weight=‘weight’)方法。

print(nx.single\_source\_dijkstra(G, 0))

({0: 0, 1: 5.0, 3: 6.0, 2: 7.0, 4: 7.0}, {0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 1, 3], 4: [0, 1, 3, 4]})

all\_pairs\_dijkstra(G, cutoff=None, weight=‘weight’)：返回所有顶点间最短加权路径和长度。

如果只返回最短加权路径或长度，则可以调用all\_pairs\_dijkstra\_path(G, cutoff=None, weight=‘weight’)或all\_pairs\_dijkstra\_path\_length(G, cutoff=None, weight=‘weight’)。

sorted(nx.all\_pairs\_dijkstra(G))

[(0,  
 ({0: 0, 1: 5.0, 3: 6.0, 2: 7.0, 4: 7.0},  
 {0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 1, 3], 4: [0, 1, 3, 4]})),  
 (1,  
 ({1: 0, 3: 1.0, 2: 2.0, 4: 2.0},  
 {1: [1], 2: [1, 2], 3: [1, 3], 4: [1, 3, 4]})),  
 (2, ({2: 0, 3: 3.0, 4: 4.0}, {2: [2], 3: [2, 3], 4: [2, 3, 4]})),  
 (3, ({3: 0, 4: 1.0}, {3: [3], 4: [3, 4]})),  
 (4, ({4: 0}, {4: [4]}))]

NetWorkX库也提供了bellman\_ford、Johnson、Floyd和A\* (“A-star”)等最短（加权）路径算法的各类计算形式，可从官网查看。

#### 2) 公共交通复合网络最短路径和长度计算

分析站点和公园（邻近站点）之间的关系，如果以公园为对象，则计算每个公园邻近站点到所有站点的最短距离（时间成本距离），反映的是公园服务于整个分析区域的位置优势，具有相对平均更短的时间成本距离，会更容易吸引分析区域内的居民到达（就近原则）；如果以站点为对象，则计算每个站点到所有公园的最短时间成本距离和均值，反映的是站点位置到达各个公园的优先选择，更多的居民会优先选择距离站点近的公园，而总体的均值则揭示了各个站点可达公园的水平，均值低的站点有更多选择不同公园的机会，而均值高的站点更容易选择距离近的公园。因此对公共交通复合网络最短路径（为顶点，即站点列表）和长度（为时间成本距离值）的计算，需要从公园服务和站点可达性两个方向（对象）进行分析，表述了服务和供给的关系。

应用NetWorkX库提供的single\_source\_dijkstra和multi\_source\_dijkstra方法计算。

* 公园服务——计算每个公园各个邻近站点到所有站点的加权最短时间成本距离

import pickle  
comprehensivePark\_adjacentStations=postSQL2gpd(table\_name='adjacent\_stations',\*\*args.db.db\_info)  
with open(args.data.G\_SB,'rb') as f:  
 G\_SB=pickle.load(f)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is adjacent\_stations.

查看公共交通复合网络顶点数有3万点之多，而每个公园各个邻近站点总共有3266个，如果计算每个邻近站点到所有站点的最短路径，将有条，如果同时存储路径和时间成本距离，文件的大小可能比较大，因此采取每个公园所有的最短路径存储为单独一个文件，总共为53个文件。每个文件的大小从几百MB到约2.5GB不等，总共约20.6GB。

计算时，也同时计算了每个公园各个邻近站点到所有站点的最短路径均值park2station\_mean\_time\_cost\_all，存储到单独的文件中。

print(len(G\_SB.nodes))

32325

import numpy as np  
comprehensivePark\_adjacentStations\_unique\_list=np.unique(flatten\_lst([eval(i) for i in comprehensivePark\_adjacentStations.adjacent\_PointUid.to\_list()]))  
print(len(comprehensivePark\_adjacentStations\_unique\_list))

3266

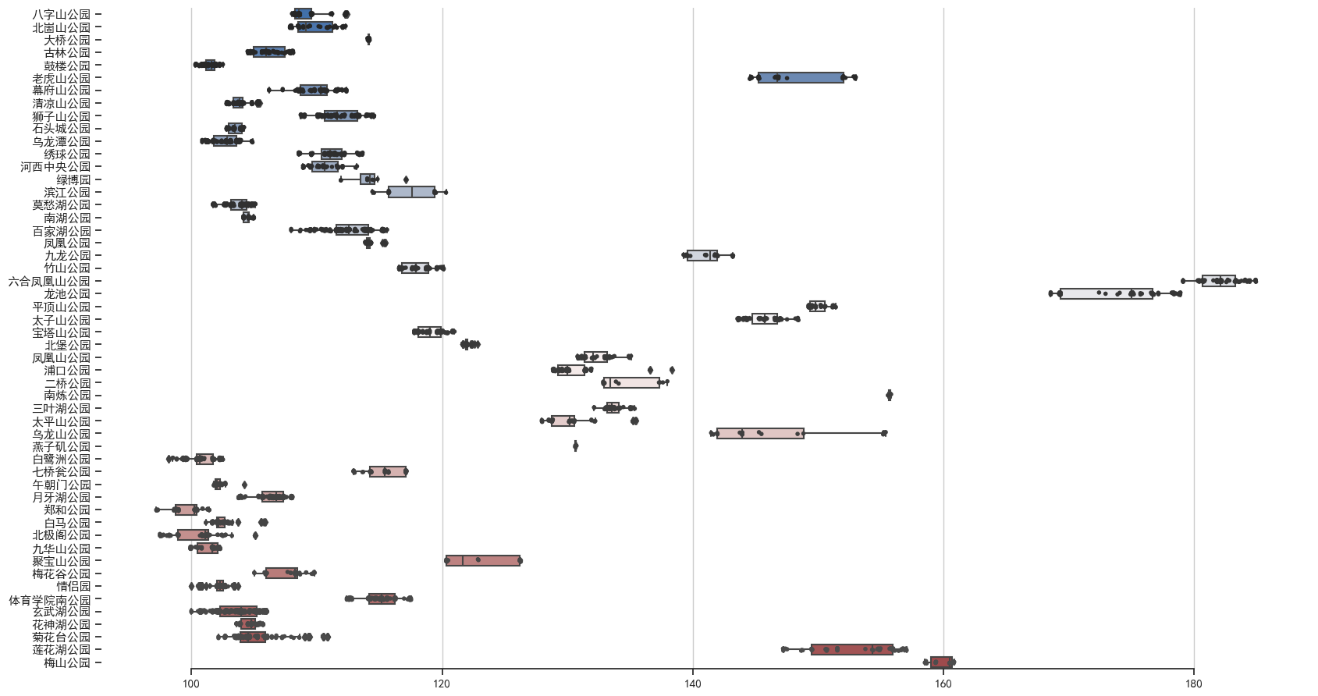
from tqdm import tqdm  
import pickle  
import os  
from statistics import mean  
\_\_C.data.park2staton\_sp\_root=r"E:\data\public\_trans\_network\park2station\_sp"  
\_\_C.data.park2station\_mean\_time\_cost\_all\_fn=r'E:\data\public\_trans\_network\park2station\_mean\_time\_cost\_all.pickle'  
  
park2station\_mean\_time\_cost\_all={}  
for idx,row in tqdm(comprehensivePark\_adjacentStations.iterrows(),total=comprehensivePark\_adjacentStations.shape[0]):  
 park2station\_sp\_all={}   
 park2station\_mean\_time\_cost\_all[row.Name\_EN]={}  
 save\_fn=os.path.join(args.data.park2staton\_sp\_root,f'{idx}\_{row.Name\_EN}.pickle')  
 for station in eval(row.adjacent\_PointUid):  
 park2station\_sp=nx.single\_source\_dijkstra(G\_SB, station,weight='time\_cost')  
 p2s\_mean\_time\_cost=mean(park2station\_sp[0].values())  
 park2station\_sp\_all[station]=park2station\_sp   
 park2station\_mean\_time\_cost\_all[row.Name\_EN][station]=p2s\_mean\_time\_cost  
  
 with open(save\_fn,'wb') as f:  
 pickle.dump(park2station\_sp\_all,f)   
  
with open(args.data.park2station\_mean\_time\_cost\_all\_fn,'wb') as f:  
 pickle.dump(park2station\_mean\_time\_cost\_all,f)

100%|██████████| 53/53 [44:45<00:00, 50.67s/it]

读取存储的公园邻近站点最短路径均值，并按公园打印为箱型图并叠加散点图，观察各个公园到分析区域所有站点最短路径时间成本距离的分布情况。较低值域分布通常位于分析区域中心，并有较多的站点可达；而较高值域分布通常位于分析区域偏离中心的位置，较为偏远。

with open(args.data.park2station\_mean\_time\_cost\_all\_fn,'rb') as f:  
 park2station\_mean\_time\_cost\_all=pickle.load(f)

import seaborn as sns  
import matplotlib.pyplot as plt  
sns.set\_style("ticks",{"font.sans-serif":['simhei', 'Arial']})   
  
f, ax=plt.subplots(figsize=(20, 11))  
mean\_time\_cost\_park=[list(i.values()) for i in park2station\_mean\_time\_cost\_all.values()]  
sns.boxplot(data=mean\_time\_cost\_park,palette="vlag",orient='h',)  
sns.stripplot(mean\_time\_cost\_park,size=4, color=".3", linewidth=0,orient='h')  
  
comprehensive\_park\_name\_mapping\_EN2CH={v:k for k,v in comprehensive\_park\_name\_mapping.items()}  
park\_names=[comprehensive\_park\_name\_mapping\_EN2CH[i] for i in park2station\_mean\_time\_cost\_all.keys()]  
ax.set\_yticklabels(park\_names)  
ax.xaxis.grid(True)  
sns.despine(trim=True, left=True)



* 站点可达性——计算每个站点到每个公园各个邻近站点的加权最短时间成本距离

已经获取每个公园各个邻近站点到分析区域内所有站点的最短距离，那么分析区域内各个站点到一个公园（包含多个邻近站点）的最短路径，应该是到该公园邻近站点中最短距离的那个邻近站点（即仅筛选出一条路径，对应一个邻近站点），因此定义station2park\_shortestPaths()函数，批量计算分析区域内所有站点到各个公园的最短路径。同样将计算结果存至本地磁盘。

def station2park\_shortestPaths(G\_SB,park2staton\_sp\_fns,epsg,save\_path):  
 '''  
 计算各个站点到一个公园（包含多个邻近站点）的最短路径，可以批量计算多个公园  
  
 Parameters  
 ----------  
 G\_SB : networkx.classes.graph.Graph  
 公共交通复合网络.  
 park2staton\_sp\_fns : list[string]  
 每个公园各个邻近站点到所有站点的加权最短时间成本距离文件路径.  
 epsg : pyproj.crs.crs.CRS[int]  
 投影epsg编号.  
 save\_path : string  
 计算结果文件保存路径.  
  
 Returns  
 -------  
 None.  
  
 '''   
 import networkx as nx  
 from tqdm import tqdm  
 import pickle  
 import pandas as pd  
 import numpy as np  
 from shapely.geometry import LineString  
 import geopandas as gpd  
 from pathlib import Path  
   
 station\_position=nx.get\_node\_attributes(G\_SB,'position')  
 park\_shortest\_path={}  
 for fn in tqdm(park2staton\_sp\_fns):  
 with open(fn,'rb') as f:  
 park2staton\_sp=pickle.load(f)  
 all\_shortest\_length\_df=pd.DataFrame.from\_dict({k:v[0] for k,v in park2staton\_sp.items()},orient='index')  
 all\_shortest\_length\_df.replace({0:np.NaN},inplace=True)  
 min\_length=all\_shortest\_length\_df.min()  
 min\_idx=all\_shortest\_length\_df.idxmin()  
 min\_concat=pd.concat([min\_idx,min\_length],axis=1).reset\_index().rename(columns={0:'idx\_adjacent',1:"time\_cost",'index':'stations'})   
 all\_shortest\_path\_dict={k:v[1] for k,v in park2staton\_sp.items()}  
 min\_concat['path']=min\_concat.apply(lambda row:all\_shortest\_path\_dict[row.idx\_adjacent][row.stations],axis=1)  
 min\_concat['geometry']=min\_concat.path.apply(lambda row:LineString([station\_position[PointUid] for PointUid in row]))  
 min\_concat\_gdf=gpd.GeoDataFrame(min\_concat,geometry='geometry',crs=epsg)  
 park\_shortest\_path[Path(fn).stem]=min\_concat\_gdf  
  
 with open(save\_path,'wb') as f:  
 pickle.dump(park\_shortest\_path,f)   
   
import os   
park2staton\_sp\_fns=[os.path.join(args.data.park2staton\_sp\_root,i) for i in os.listdir(args.data.park2staton\_sp\_root)]   
\_\_C.data.park\_shortest\_path\_fn=r"E:\data\public\_trans\_network\park\_shortest\_path.pkl"  
station2park\_shortestPaths(G\_SB,park2staton\_sp\_fns,args.gi.nanjing\_epsg,args.data.park\_shortest\_path\_fn)

100%|██████████| 53/53 [25:19<00:00, 28.68s/it]

读取并查看计算后存至本地磁盘的文件。每个公园对应一个GeoDataFrame格式数据，其中stations列为分析区域内所有站点，idx\_adjacent列为站点对应的邻近站点，是该站点到达公园的最短路径（为到公园诸多邻近站点路径最短的，time\_cost列为时间成本距离，path列为最短路径（路），geometry为最短路径空间几何对象。

with open(args.data.park\_shortest\_path\_fn,'rb') as f:  
 park\_shortest\_path=pickle.load(f)

print(park\_shortest\_path.keys())  
park\_shortest\_path['0\_Bazi Mountain']

dict\_keys(['0\_Bazi Mountain', '10\_Wulong Pool', '11\_Xiuqing Park', '12\_Hexi Central Park', '13\_Green EXPO Garden', '14\_Riverside Park', '15\_Mochou Lake', '16\_South Lake', '17\_Baijia Lake', '18\_Phoenix Park', '19\_Jiulong Park', '1\_Beigu Mountain', '20\_Zhushan Park', '21\_Jiulong Park', '22\_Liuhe Phoenix Mountain', '23\_Dragon Pool', '24\_Pingding Mountain', '25\_Prince Mountain', '26\_Pagoda Hill', '27\_North Fort', '28\_Phoenix Mountain', '29\_Pukou Park', '2\_Bridge park', '30\_Two Bridges Park', '31\_Nanlian Park', '32\_Three Leaf Lake', '33\_Taiping Hill', '34\_Wulong Mountain', '35\_Swallow Rock', '36\_Egret Island', '37\_Qiqiaoweng Park', '38\_Wuchaomen Park', '39\_Crescent Lake', '3\_Ancient Forest', '40\_Zhenghe Park', '41\_White House Park', '42\_Arctic Pavilion', '43\_Jiuhua Mountain', '44\_Treasure Hill', '45\_Plum Blossom Valley', '46\_Couples Garden', '47\_South Park of Sports Institute', '48\_Xuanwu Lake', '49\_Flora Lake', '4\_Drum Tower', '50\_Chrysanthemums Terrace', '51\_Lotus Lake', '52\_Plum Blossom Hill', '5\_Tiger Mountain', '6\_Mufu Mountain', '7\_Qingliang Mountain', '8\_Lion Rock Mountain', '9\_Stone City Park'])

|  | **stations** | **idx\_adjacent** | **time\_cost** | **path** | **geometry** |
| --- | --- | --- | --- | --- | --- |
| **0** | 3176ab8f267a522adc1e5249 | fe1f80a7631696d4ab175aed | 0.019408 | [fe1f80a7631696d4ab175aed, 3176ab8f267a522adc1... | LINESTRING (664855.936 3551546.438, 664855.439... |
| **1** | fe1f80a7631696d4ab175aed | 3176ab8f267a522adc1e5249 | 0.019408 | [3176ab8f267a522adc1e5249, fe1f80a7631696d4ab1... | LINESTRING (664855.439 3551547.768, 664855.936... |
| **2** | bdc72d7d81c5e9674d554732 | f9c911cc8443a54a58f6be3a | 0.009315 | [f9c911cc8443a54a58f6be3a, bdc72d7d81c5e9674d5... | LINESTRING (664849.553 3551548.860, 664850.145... |
| **3** | 9ae6f103dc1fa50f0aa8d4e7 | f9c911cc8443a54a58f6be3a | 0.033094 | [f9c911cc8443a54a58f6be3a, 9ae6f103dc1fa50f0aa... | LINESTRING (664849.553 3551548.860, 664850.745... |
| **4** | f9c911cc8443a54a58f6be3a | bdc72d7d81c5e9674d554732 | 0.009315 | [bdc72d7d81c5e9674d554732, f9c911cc8443a54a58f... | LINESTRING (664850.145 3551548.523, 664849.553... |
| **...** | ... | ... | ... | ... | ... |
| **32192** | 33192a863cc21818d8d891a8 | 2077fa870b9bbdb57adbd2c1 | 487.746560 | [2077fa870b9bbdb57adbd2c1, 141ded09336bdae46e2... | LINESTRING (665061.514 3551596.830, 665394.665... |
| **32193** | df1760ab01d707e48b2fb5c3 | 2077fa870b9bbdb57adbd2c1 | 488.808496 | [2077fa870b9bbdb57adbd2c1, 141ded09336bdae46e2... | LINESTRING (665061.514 3551596.830, 665394.665... |
| **32194** | c9d47676a7369a2f95f08c9e | 2077fa870b9bbdb57adbd2c1 | 488.949535 | [2077fa870b9bbdb57adbd2c1, 141ded09336bdae46e2... | LINESTRING (665061.514 3551596.830, 665394.665... |
| **32195** | d3648dce49b200d02efaa7a5 | 2077fa870b9bbdb57adbd2c1 | 491.759761 | [2077fa870b9bbdb57adbd2c1, 141ded09336bdae46e2... | LINESTRING (665061.514 3551596.830, 665394.665... |
| **32196** | f60b7ce35261670badb22aaf | 2077fa870b9bbdb57adbd2c1 | 491.854833 | [2077fa870b9bbdb57adbd2c1, 141ded09336bdae46e2... | LINESTRING (665061.514 3551596.830, 665394.665... |

32197 rows × 5 columns

按照stations列合并包含各个站点到公园最短路径的park\_shortest\_path数据，合并时仅保留最短路径的时间成本列time\_cost和stations。Pandas库提供的merge方法一次只能合并两个DataFrame格式数据，配合functools库提供的reduce批量计算。为了达到合并的目的，使用reduce时调用自定义函数pd\_merge()实现重命名列名，避免合并后的列名重复；并移除park列，由重命名的列名表明time\_cost值对应的公园名。

from functools import reduce  
import pandas as pd  
  
park\_shortest\_path\_list=[]  
for park\_name,df in park\_shortest\_path.items():  
 df\_=df[['stations','time\_cost']]  
 df\_['park']=park\_name  
 park\_shortest\_path\_list.append(df\_)   
   
def pd\_merge(x,y):  
 y.rename(columns={i:"{}-{}".format(i,y.park[0]) for i in y.columns if i!='stations'},inplace=True)   
 y.drop(columns=[i for i in y.columns if 'park' in i],inplace=True)  
 return pd.merge(x, y, on='stations')  
   
station2parks\_tc=reduce(pd\_merge,park\_shortest\_path\_list)   
station2parks\_tc.rename(columns={'time\_cost':"{}-{}".format('time\_cost',station2parks\_tc.park[0]),'park':"{}-{}".format('park',station2parks\_tc.park[0])},inplace=True)  
station2parks\_tc.drop(columns=[i for i in station2parks\_tc.columns if 'park' in i],inplace=True)

计算各个站点到达所有公园的最短路径时间成本距离均值，存储至列time\_cost\_average下。

time\_cost\_columns=[i for i in station2parks\_tc.columns if 'time\_cost' in i]  
station2parks\_tc['time\_cost\_average']=station2parks\_tc.apply(lambda row:row[time\_cost\_columns].mean(),axis=1)

station2parks\_tc

|  | **stations** | **time\_cost-0\_Bazi Mountain** | **time\_cost-10\_Wulong Pool** | **time\_cost-11\_Xiuqing Park** | **time\_cost-12\_Hexi Central Park** | **time\_cost-13\_Green EXPO Garden** | **time\_cost-14\_Riverside Park** | **time\_cost-15\_Mochou Lake** | **time\_cost-16\_South Lake** | **time\_cost-17\_Baijia Lake** | **...** | **time\_cost-4\_Drum Tower** | **time\_cost-50\_Chrysanthemums Terrace** | **time\_cost-51\_Lotus Lake** | **time\_cost-52\_Plum Blossom Hill** | **time\_cost-5\_Tiger Mountain** | **time\_cost-6\_Mufu Mountain** | **time\_cost-7\_Qingliang Mountain** | **time\_cost-8\_Lion Rock Mountain** | **time\_cost-9\_Stone City Park** | **time\_cost\_average** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 3176ab8f267a522adc1e5249 | 0.019408 | 22.950006 | 0.643352 | 46.856041 | 48.748149 | 50.986255 | 24.715803 | 32.920009 | 67.185653 | ... | 17.198855 | 46.180673 | 100.309090 | 110.793531 | 93.538042 | 18.425975 | 17.457294 | 4.405315 | 17.457294 | 50.073777 |
| **1** | fe1f80a7631696d4ab175aed | 0.019408 | 22.945966 | 0.650552 | 46.875450 | 48.767558 | 51.005663 | 24.711762 | 32.915968 | 67.205062 | ... | 17.218264 | 46.200081 | 100.328499 | 110.812940 | 93.557450 | 18.406567 | 17.453253 | 4.385907 | 17.453253 | 50.081661 |
| **2** | bdc72d7d81c5e9674d554732 | 0.009315 | 23.022326 | 0.570990 | 46.873608 | 48.772338 | 51.012818 | 24.788123 | 32.992329 | 67.235164 | ... | 17.225726 | 46.208000 | 100.326657 | 110.811098 | 93.534306 | 18.378164 | 17.529614 | 4.416776 | 17.529614 | 50.096912 |
| **3** | 9ae6f103dc1fa50f0aa8d4e7 | 0.033094 | 23.016033 | 0.580431 | 46.883053 | 48.781784 | 51.022263 | 24.784406 | 32.988612 | 67.211597 | ... | 17.204724 | 46.198019 | 100.336102 | 110.820543 | 93.543751 | 18.412571 | 17.525897 | 4.451183 | 17.525897 | 50.096371 |
| **4** | f9c911cc8443a54a58f6be3a | 0.009315 | 23.028336 | 0.562955 | 46.864804 | 48.763534 | 51.004014 | 24.796411 | 33.000616 | 67.225849 | ... | 17.216410 | 46.198685 | 100.317853 | 110.802294 | 93.525502 | 18.387479 | 17.537902 | 4.426091 | 17.537902 | 50.093797 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **32192** | 33192a863cc21818d8d891a8 | 487.746560 | 466.602987 | 488.359041 | 457.709650 | 468.442227 | 467.502121 | 465.143139 | 463.143003 | 418.375926 | ... | 467.682283 | 438.379531 | 472.670183 | 492.522058 | 519.658745 | 486.296449 | 471.567834 | 490.716023 | 471.567834 | 485.338623 |
| **32193** | df1760ab01d707e48b2fb5c3 | 488.808496 | 467.664923 | 489.420978 | 458.771587 | 469.504164 | 468.564057 | 466.205075 | 464.204940 | 419.437862 | ... | 468.744219 | 439.441467 | 473.732119 | 493.583994 | 520.720681 | 487.358385 | 472.629771 | 491.777959 | 472.629771 | 486.400560 |
| **32194** | c9d47676a7369a2f95f08c9e | 488.949535 | 467.805961 | 489.562016 | 458.912625 | 469.645202 | 468.705096 | 466.346114 | 464.345978 | 419.578901 | ... | 468.885258 | 439.582506 | 473.873158 | 493.725033 | 520.861720 | 487.499424 | 472.770809 | 491.918998 | 472.770809 | 486.541598 |
| **32195** | d3648dce49b200d02efaa7a5 | 491.759761 | 470.616188 | 492.372242 | 461.722852 | 472.455429 | 471.515322 | 469.156340 | 467.156205 | 422.389127 | ... | 471.695484 | 442.392732 | 476.683384 | 496.535259 | 523.671946 | 490.309650 | 475.581035 | 494.729224 | 475.581035 | 489.351825 |
| **32196** | f60b7ce35261670badb22aaf | 491.854833 | 470.711260 | 492.467314 | 461.817923 | 472.550501 | 471.610394 | 469.251412 | 467.251277 | 422.484199 | ... | 471.790556 | 442.487804 | 476.778456 | 496.630331 | 523.767018 | 490.404722 | 475.676107 | 494.824296 | 475.676107 | 489.446897 |

32197 rows × 54 columns

将站点到达所有公园的最短路径时间成本距离均值加入到图顶点属性下，属性名为station2parks\_tc，因为并不是所有站点都有到达公园的时间成本距离值，因此将空值部分替换为值0，避免图打印顶点颜色时出现错误。

for idx,row in tqdm(station2parks\_tc.iterrows(),total=station2parks\_tc.shape[0]):  
 G\_SB.nodes[row.stations]['station2parks\_tc']=row.time\_cost\_average

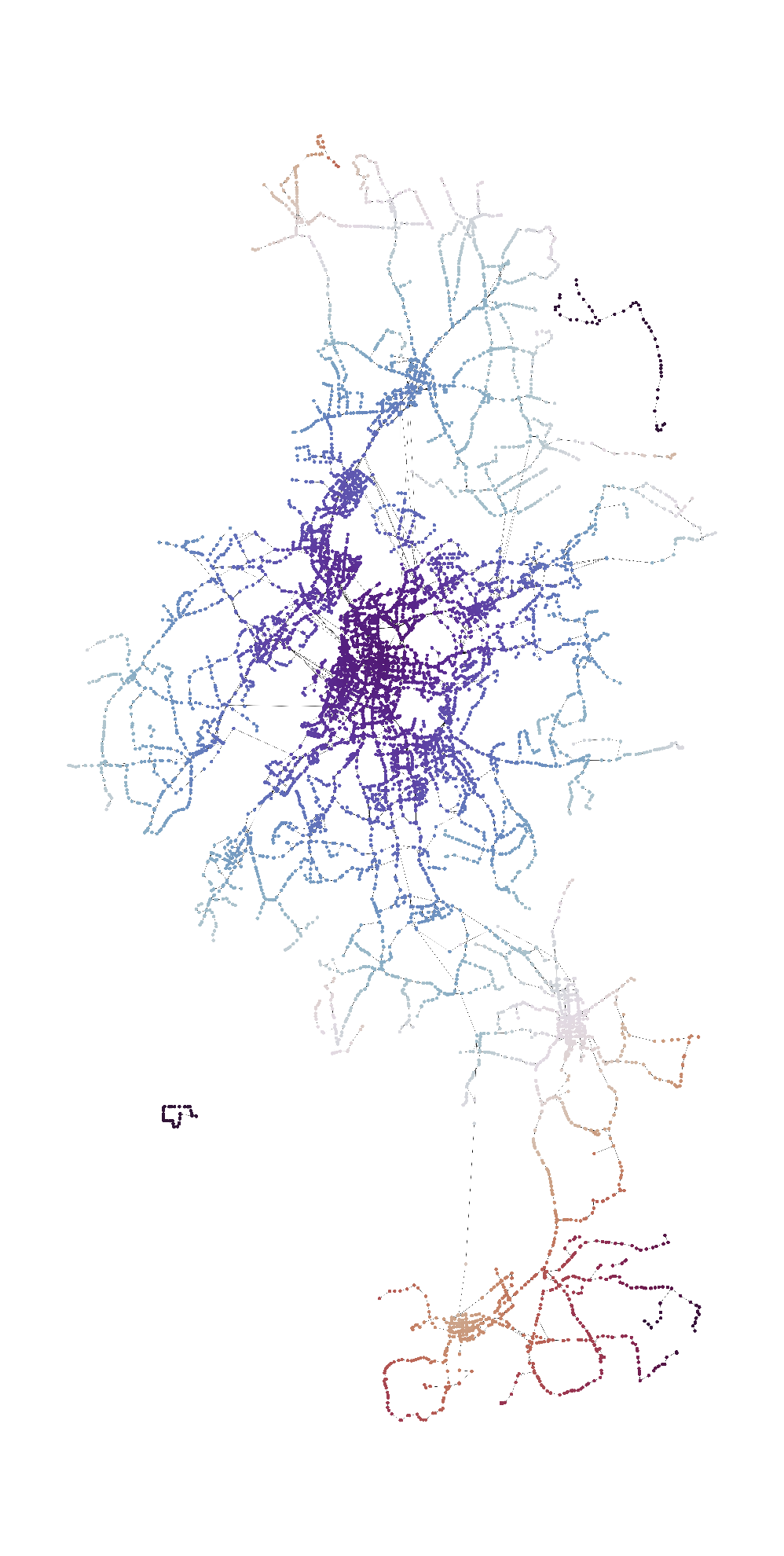
100%|██████████| 32197/32197 [00:02<00:00, 11943.34it/s]

for node in tqdm(G\_SB.nodes):  
 node\_dict=G\_SB.nodes[node]  
 if node\_dict.get('station2parks\_tc') is None:  
 node\_dict['station2parks\_tc']=0

100%|██████████| 32325/32325 [00:00<00:00, 1198854.71it/s]

从打印结果来看，具有明显中心性特征的城市，站点越趋近于城市中心，该站点到达各个公园的最短路径时间成本距离均值越小；如果远离城市中心，则均值越大。

fig, ax=plt.subplots(figsize=(30\*2\*2, 30\*4\*2))  
node\_color=[round(255\*i) for i in nx.get\_node\_attributes(G\_SB,'station2parks\_tc').values()]  
nx.draw(G\_SB,  
 pos=nx.get\_node\_attributes(G\_SB,'position'),  
 edge\_color='black',  
 font\_size=20,  
 node\_size=800,  
 ax=ax,  
 node\_color=node\_color,  
 cmap=plt.cm.twilight\_shifted  
 )



* 最短距离圈（服务范围等级）

以一个公园为对象，在公共交通复合网络下，该公园根据时间成本距离可以划分出多个距离圈，即服务范围等级（跳），这有助于研究公园服务的范围变化和城市生活圈相关的研究内容。

为了清晰的查看各个公园在不同时间成本距离下覆盖的范围，定义service\_area\_levels()和G\_draw\_paths\_composite()函数，可以打印各个公园不同距离（服务等级、圈）的子网络，并颜色标识区分满足不同距离的边。在函数定义时，将打印的子网络图存储至本地磁盘中。

def G\_draw\_paths\_composite(G,range\_paths,adjacent\_PointUid,title\_name,savefig\_root,figsize=(30, 30),level=3):  
 '''  
 计算单个公园的服务距离“圈”  
  
 Parameters  
 ----------  
 G : networkx.classes.graph.Graph  
 公共交通复合网络.  
 range\_paths : GeoDataFrame  
 公园邻近站点满足最短距离圈的划分级别.  
 adjacent\_PointUid : list  
 公园的邻里站点.  
 title\_name : string  
 图表名.  
 figsize : tuple(float,float), optional  
 图表大小. The default is (30, 30).  
 level : int, optional  
 最短距离圈（服务范围等级）. The default is 3.  
  
 Returns  
 -------  
 edges\_dict : dict  
 公园服务范围等级包含的所有边.  
 G\_p\_range : networkx.classes.graph.Graph  
 公园最短距离圈子网络.  
  
 '''   
 import networkx as nx  
 import matplotlib.pyplot as plt  
 import random  
 from collections import defaultdict  
 import matplotlib  
 import numpy as np  
 import matplotlib.colors as mcolors  
 import os   
 flatten\_lst=lambda lst: [m for n\_lst in lst for m in flatten\_lst(n\_lst)] if type(lst) is list else [lst]   
   
 fig, ax=plt.subplots(figsize=figsize)  
 ax.set\_title(title\_name,fontsize=300)  
   
 range\_paths.dropna(inplace=True)  
 ranges=range\_paths.range.unique()[:level]  
 # print('\nranges:',ranges)  
 path\_edges\_dict={}  
 for i in ranges:  
 range\_paths\_range=range\_paths[range\_paths.range==i]  
 path\_edges=range\_paths\_range.path.to\_list()   
 path\_edges\_dict[i]=path\_edges  
   
 G\_p=G.copy()  
 G\_p.remove\_edges\_from(list(G\_p.edges()))  
 G\_p\_range=nx.Graph(G\_p.subgraph(flatten\_lst(list(path\_edges\_dict.values()))))  
   
 edges\_dict=defaultdict(list)  
 for k in path\_edges\_dict.keys():  
 for r in path\_edges\_dict[k]:  
 path\_edges=[(r[n],r[n+1]) for n in range(len(r)-1)]  
 G\_p\_range.add\_edges\_from(path\_edges)  
 edges\_dict[k].append(path\_edges)  
 # print("Graph has %d nodes with %d paths(edges)" %(G\_p\_range.number\_of\_nodes(), G\_p\_range.number\_of\_edges()))   
  
 pos=nx.get\_node\_attributes(G\_p\_range,'position')  
 nx.draw(G\_p\_range,pos=pos,node\_size=100,ax=ax)  
  
 G\_start\_stops=G\_p\_range.subgraph(adjacent\_PointUid)  
 pos\_adjacent=nx.get\_node\_attributes(G\_start\_stops,'position')  
 nx.draw\_networkx\_nodes(G\_start\_stops,pos=pos\_adjacent,node\_size=1000\*2,ax=ax,node\_color='red')  
 linewidths=[5+i\*5 for i in range(len(ranges))] # [5,10,15,20,25]  
 linewidths.reverse()  
 cmap=matplotlib.cm.get\_cmap('tab20b') # 'tab20b'  
 colors=[cmap(i) for i in np.linspace(0,1,len(ranges))]  
 for j,k in enumerate(reversed(list(edges\_dict.keys()))):  
 for i, edgelist in enumerate(edges\_dict[k]):   
 nx.draw\_networkx\_edges(G\_p\_range,pos=pos,edgelist=edgelist,edge\_color=mcolors.to\_hex(colors[j]),width=linewidths[j],ax=ax)  
 plt.savefig(os.path.join(savefig\_root,'{}.png'.format(title\_name)))   
   
 return edges\_dict,G\_p\_range  
  
def service\_area\_levels(G\_SB,park\_shortest\_path,comprehensivePark\_adjacentStations,name\_mapping,savefig\_root,SES=[0,25,5],figsize=(30, 30),level=3):  
 '''  
 计算公园服务距离“圈”，调用单个计算函数G\_draw\_paths\_composite  
  
 Parameters  
 ----------  
 G\_SB : networkx.classes.graph.Graph  
 公共交通复合网络.  
 park\_shortest\_path : list[GeoDataFrame]  
 各个站点到一个公园（包含多个邻近站点）的最短路径.  
 comprehensivePark\_adjacentStations : GeoDataFrame  
 各个公园的邻近站点.  
 SES : list, optional  
 时间成本距离区间，为[start,stop,step]. The default is [0,25,5].  
 figsize : tuple, optional  
 图表大小. The default is (30, 30).  
 level : int, optional  
 最短距离圈（服务范围等级）. The default is 3.  
  
 Returns  
 -------  
 park\_serviceArea\_level: dict[string:GeoDataFrame]  
 各个公园邻近站点满足最短距离圈的划分级别.  
 park\_serviceArea\_level\_fre\_df: DataFrame  
 各个公园服务范围等级存有路径的频数.  
 G\_p\_range\_dict： dict[string,networkx.classes.graph.Graph]  
 各个公园最短距离圈子网络.  
 edges\_range\_dict: dict  
 各个公园服务范围等级包含的所有边.  
   
 '''   
 import pandas as pd  
 import numpy as np  
 from shapely.geometry import LineString  
 import geopandas as gpd  
 from pathlib import Path  
 from tqdm import tqdm  
 from IPython.display import clear\_output  
  
 start,end,step=SES   
 def partition(value,start,end,step):  
 import numpy as np  
 ranges\_list=list(range(start,end+step,step))  
 ranges=[(ranges\_list[i],ranges\_list[i+1]) for i in range(len(ranges\_list)-1)]  
 for r in ranges:  
 if r[0]<=value<r[1] :   
 return r[1]   
   
 park\_serviceArea\_level={}  
 park\_serviceArea\_level\_fre={}  
 G\_p\_range\_dict={}  
 edges\_range\_dict={}  
 for park,shortest\_path in tqdm(park\_shortest\_path.items()):  
 clear\_output(wait=True)  
 shortest\_path['range']=shortest\_path.time\_cost.apply(partition,args=(start,end,step))  
 ranges\_frequency=shortest\_path['range'].value\_counts()  
 park\_serviceArea\_level\_fre[park]=ranges\_frequency.to\_dict()   
 park\_serviceArea\_level[park]=shortest\_path  
 park\_serviceArea\_level\_fre\_df=pd.DataFrame.from\_dict(park\_serviceArea\_level\_fre,orient='index')   
 park\_name=park.split('\_')[-1]   
 adjacent\_PointUid=eval(comprehensivePark\_adjacentStations[comprehensivePark\_adjacentStations.Name\_EN==park\_name].adjacent\_PointUid.to\_list()[0])  
   
 name\_mapping\_=dict([(value, key) for key, value in name\_mapping.items()])  
 park\_name\_=name\_mapping\_[park\_name]  
 edges\_dict\_,G\_p\_range=G\_draw\_paths\_composite(G\_SB,shortest\_path,adjacent\_PointUid,savefig\_root=savefig\_root,figsize=figsize,level=level,title\_name=park\_name\_)   
 G\_p\_range\_dict[park]=G\_p\_range  
 edges\_range\_dict[park]=edges\_dict\_  
  
 park\_serviceArea\_level\_fre\_df=pd.DataFrame.from\_dict(park\_serviceArea\_level\_fre,orient='index')  
   
 return park\_serviceArea\_level,park\_serviceArea\_level\_fre\_df,G\_p\_range\_dict,edges\_range\_dict

\_\_C.data.service\_area\_levels\_root=r"E:\data\public\_trans\_network\service\_area\_levels"  
park\_serviceArea\_level,park\_serviceArea\_level\_fre,G\_p\_range\_dict,edges\_range\_dict=service\_area\_levels(  
 G\_SB,  
 park\_shortest\_path,  
 comprehensivePark\_adjacentStations,  
 name\_mapping=comprehensive\_park\_name\_mapping,  
 savefig\_root=args.data.service\_area\_levels\_root,  
 SES=[0,90,5],  
 figsize=(30\*2, 30\*2),  
 level=3)

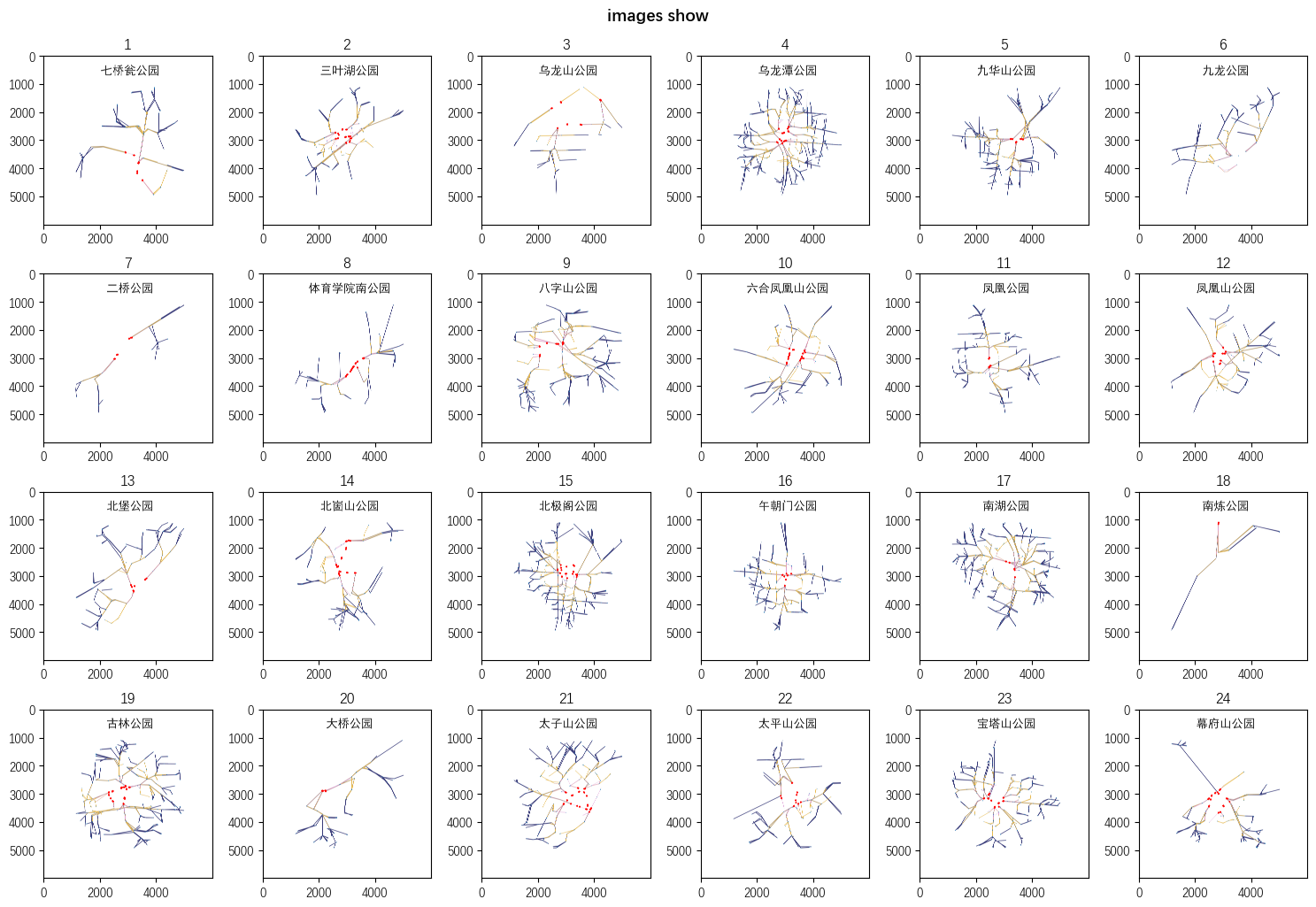
100%|██████████| 53/53 [23:24<00:00, 26.51s/it]

将G\_p\_range\_dict和edges\_range\_dict两个返回值存至本地磁盘，方便调用查看数据，而不必重新运行程序。

\_\_C.data.G\_p\_range\_dict\_fn=r"E:\data\public\_trans\_network\G\_p\_range\_dict.pkl"  
\_\_C.data.edges\_range\_dict\_fn=r"E:\data\public\_trans\_network\edges\_range\_dict.pkl"  
with open(args.data.G\_p\_range\_dict\_fn,'wb') as f:  
 pickle.dump(G\_p\_range\_dict,f)   
with open(args.data.edges\_range\_dict\_fn,'wb') as f:  
 pickle.dump(edges\_range\_dict,f)

在“调研图像与聚类城市色彩”一章定义有imgs\_layoutShow\_FPList()函数，将其置于util\_misc模块，直接调用显示给定文件夹下所有公园不同距离圈子网络的图片。

from util\_misc import imgs\_layoutShow\_FPList  
import glob  
import os  
  
imgs\_park\_serviceArea\_level\_fns=glob.glob(os.path.join(args.data.service\_area\_levels\_root,'\*.png'))  
columns=6;scale=1  
imgs\_layoutShow\_FPList(imgs\_park\_serviceArea\_level\_fns[:6\*4],columns,scale,figsize=(15,10))



### 2.8.3.5 公园潜在服务人口压力

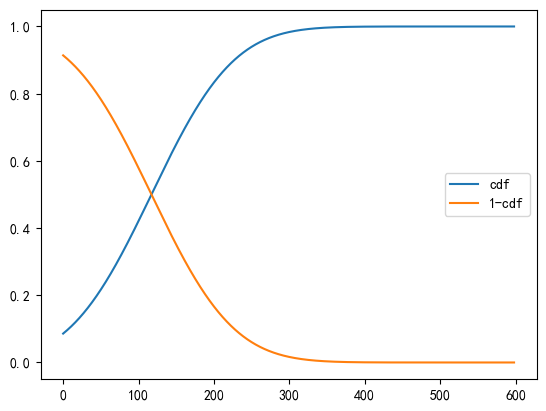
居民对公园绿地的需求会随距离的增加而下降，以高斯函数为阻抗函数反应距离衰减的影响，沿公共交通复合网络计算各个公园到所有站点的最短路径，将每一人口采样点的人口数延最近的站点传输到各个公园邻近站点，计算该公园潜在的人口服务压力。定义population\_flowOverNetwork()函数用于计算人口统计数据中，每个统计点到达各个公园潜在的人口流入量；定义populationWeighted\_adjacentStations()函数计算每个公园邻近站点传输至的潜在人口数（高斯权重加权后），及均值，表述分析区域各个公园潜在的人口压力。

population=postSQL2gpd(table\_name='population',\*\*args.db.db\_info)  
comprehensive\_park=postSQL2gpd(table\_name='comprehensive\_park',\*\*args.db.db\_info)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is population.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
The data has been read from PostgreSQL database. The table name is comprehensive\_park.

为了方便理解高斯（累积分布函数（Cumulative distribution function，CDF））权重对数值的变化影响，将population\_flowOverNetwork()函数中计算高斯权重的部分代码迁移出来，通过图表打印的方式，观察到累积分布函数曲线（CDF）和1-CDF曲线的变化。1-CDF曲线是随着值（时间成本距离）的增加而逐步降低，达到300左右几乎降至为0，并趋于平缓，因此时间成本距离越高，值越小，传入的人口数乘以权重值后越小；反之，时间成本距离越低，值越大，传入的人口数乘以权重值后越大，可以表述为距离公园越近区域的市民到公园的几率就大，越远则到公园的几率越小，当足够远时，去公园的意愿基本为0。

import statistics  
from scipy.stats import norm  
flatten\_lst=lambda lst: [m for n\_lst in lst for m in flatten\_lst(n\_lst)] if type(lst) is list else [lst]  
  
time\_cost=flatten\_lst([v.time\_cost.to\_list() for v in park\_shortest\_path.values()])  
tc\_max=max(time\_cost)  
tc\_min=min(time\_cost)  
tc\_mean=statistics.mean(time\_cost)  
tc\_std=statistics.stdev(time\_cost)   
  
time\_cost.sort()  
cdf\_tc=norm.cdf(time\_cost,loc=tc\_mean,scale=tc\_std)  
  
plt.plot(time\_cost, cdf\_tc, label='cdf')  
plt.plot(time\_cost, 1-cdf\_tc, label='1-cdf')  
plt.legend();



def population\_flowOverNetwork(G\_SB,population,park\_shortest\_path,save\_path):  
 '''  
 对于一个公园，获取人口分布统计点的最近站点，由该最近站点到这个公园的最短路径时间成本距离，计算高斯权重，乘以人口分布统计点的人口数作为可能到达公园的人口数。同样的方式批量计算所有公园的潜在人口流入量  
  
 Parameters  
 ----------  
 G\_SB : networkx.classes.graph.Graph  
 公共交通复合网络.  
 population : GeoDataFrame  
 人口分布数据.  
 park\_shortest\_path : list[GeoDataFrame]  
 各个站点到一个公园（包含多个邻近站点）的最短路径.  
 save\_path : string  
 计算结果文件保存路径.  
  
 Returns  
 -------  
 dict[GeoDataFrame]  
 各个人口统计点位到公园的潜在人口流入量.  
  
 '''   
 from shapely.ops import nearest\_points  
 import pandas as pd  
 from shapely.geometry import MultiPoint,Point  
 from tqdm import tqdm  
 import pickle  
 import copy  
 import statistics  
 from scipy.stats import norm  
 flatten\_lst=lambda lst: [m for n\_lst in lst for m in flatten\_lst(n\_lst)] if type(lst) is list else [lst]   
  
 stations\_pts={k:Point(v) for k,v in nx.get\_node\_attributes(G\_SB,"position").items()}  
 stations\_mpts=MultiPoint(list(stations\_pts.values()))  
   
 time\_cost=flatten\_lst([v.time\_cost.to\_list() for v in park\_shortest\_path.values()])  
 tc\_max=max(time\_cost)  
 tc\_min=min(time\_cost)  
 tc\_mean=statistics.mean(time\_cost)  
 tc\_std=statistics.stdev(time\_cost)   
   
 population\_flow\_dict={}  
 def row\_func(row):  
 nearest\_pt=nearest\_points(row,stations\_mpts)[1]  
 nearest\_pt\_row=shortest\_path\_df[shortest\_path\_df["station\_geometry"]==str(nearest\_pt)]  
   
 return [nearest\_pt,nearest\_pt\_row.idx\_adjacent.values[0],nearest\_pt\_row.time\_cost.values[0],nearest\_pt\_row.stations.values[0]]  
   
 for park,shortest\_path\_df in tqdm(park\_shortest\_path.items()):  
 population\_flow=copy.deepcopy(population)  
 shortest\_path\_df['station\_geometry']=shortest\_path\_df.stations.apply(lambda row:str(stations\_pts[row]))   
 nearest\_info=population\_flow.geometry.apply(row\_func)  
 col\_n=['nearest\_pt','idx\_adjacent','time\_cost','stations']  
 nearest\_info\_T=[list(i) for i in zip(\*nearest\_info.to\_list())]   
 for n,info in zip(col\_n,nearest\_info\_T):  
 population\_flow[n]=info  
   
 population\_flow['tc\_weight']=population\_flow.time\_cost.apply(lambda row:1-(row-tc\_min)/(tc\_max-tc\_min))  
 population\_flow['Gaussian\_weight']=population\_flow.time\_cost.apply(lambda row:1-norm.cdf(row,loc=tc\_mean,scale=tc\_std))   
 population\_flow['population\_Gweighted']=population\_flow.apply(lambda row:int(row.Population)\*row.Gaussian\_weight,axis=1)  
 population\_flow\_dict[park]=population\_flow  
   
 with open(save\_path,'wb') as f:  
 pickle.dump(population\_flow\_dict,f)   
 return population\_flow\_dict

\_\_C.data.G\_p\_range\_dict\_fn=r"E:\data\public\_trans\_network\population\_flow\_dict.pkl"  
population\_flow\_dict=population\_flowOverNetwork(G\_SB,population,park\_shortest\_path,args.data.G\_p\_range\_dict\_fn)

100%|██████████| 53/53 [46:33<00:00, 52.71s/it]

population\_flow\_dict['0\_Bazi Mountain']

|  | **Lat** | **Lng** | **Population** | **geometry** | **nearest\_pt** | **idx\_adjacent** | **time\_cost** | **stations** | **tc\_weight** | **Gaussian\_weight** | **population\_Gweighted** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 31.9497433580896 | 118.378427463393 | 129 | POINT (630276.381 3535694.515) | POINT (630806.0127756759 3534389.4983157325) | d9f65176dce0a6b6abf7292d | 198.790329 | 552fc62d8f0bc3133686f44f | 0.667461 | 0.169984 | 21.927988 |
| **1** | 31.9497433580896 | 118.400885345496 | 129 | POINT (632399.081 3535721.765) | POINT (633041.3670544036 3535551.691866558) | d9f65176dce0a6b6abf7292d | 185.991482 | 9142035b5e14c615c909f0c0 | 0.688871 | 0.210451 | 27.148125 |
| **2** | 31.9230613713896 | 118.405376921916 | 386 | POINT (632862.009 3532769.406) | POINT (633186.7269609962 3532671.776092457) | d9f65176dce0a6b6abf7292d | 197.886086 | 307d64433aec7b65e215f9c8 | 0.668974 | 0.172668 | 66.649900 |
| **3** | 31.9383091693281 | 118.369444310552 | 129 | POINT (629443.336 3534416.196) | POINT (629592.4638845846 3534237.0085329) | d9f65176dce0a6b6abf7292d | 204.295228 | 4677409768844f44a21de954 | 0.658252 | 0.154227 | 19.895280 |
| **4** | 31.976417597875 | 118.391902192655 | 386 | POINT (631511.965 3538667.820) | POINT (633159.4666844679 3538380.031112542) | d9f65176dce0a6b6abf7292d | 198.487712 | ac9653e99a0490c6b42655d4 | 0.667967 | 0.170880 | 65.959508 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **6668** | 32.2123364183777 | 119.209369101203 | 129 | POINT (708226.801 3566112.695) | POINT (707718.9550723871 3566095.829331054) | 2077fa870b9bbdb57adbd2c1 | 260.296110 | 7b882850378ca225c1f1cfd6 | 0.564573 | 0.047260 | 6.096488 |
| **6669** | 32.2199365524469 | 119.204877524783 | 386 | POINT (707786.094 3566946.714) | POINT (707718.9550723871 3566095.829331054) | 2077fa870b9bbdb57adbd2c1 | 260.296110 | 7b882850378ca225c1f1cfd6 | 0.564573 | 0.047260 | 18.242204 |
| **6670** | 32.2123364183777 | 119.213860677624 | 129 | POINT (708650.211 3566121.412) | POINT (707718.9550723871 3566095.829331054) | 2077fa870b9bbdb57adbd2c1 | 260.296110 | 7b882850378ca225c1f1cfd6 | 0.564573 | 0.047260 | 6.096488 |
| **6671** | 32.200935026272 | 119.204877524783 | 129 | POINT (707829.333 3564839.790) | POINT (706822.3294598485 3565145.8209223025) | 2077fa870b9bbdb57adbd2c1 | 254.448669 | 5cf67b908cd4a332157643b2 | 0.574355 | 0.054381 | 7.015130 |
| **6672** | 32.2199365524469 | 119.204877524783 | 386 | POINT (707786.094 3566946.714) | POINT (707718.9550723871 3566095.829331054) | 2077fa870b9bbdb57adbd2c1 | 260.296110 | 7b882850378ca225c1f1cfd6 | 0.564573 | 0.047260 | 18.242204 |

6673 rows × 11 columns

def populationWeighted\_adjacentStations(population\_flow\_dict,G\_SB,epsg,comprehensive\_park):  
 '''  
 计算各个公园邻近站点传入的潜在人口数，及均值，表述公园潜在人口压力  
  
 Parameters  
 ----------  
 population\_flow\_dict : dict[GeoDataFrame]  
 人口统计数据中，每个统计点到达各个公园潜在的人口流入量.  
 G\_SB : networkx.classes.graph.Graph  
 公共交通复合网络.  
 epsg : pyproj.crs.crs.CRS[int]  
 投影epsg编号.  
 comprehensive\_park : GeoDataFrame  
 公园地理空间数据.  
  
 Returns  
 -------  
 populationWeighted\_adjacentStations\_dict : dict(GeoDataFrame)  
 字典形式返回各个公园的潜在人口服务压力.  
 populationWeighted\_adjacentStations\_stack\_gdf : GeoDataFrame  
 合并各个公园的潜在人口服务压力为单独一个GeoDataFrame格式数据，以公园名为行索引.  
 comprehensive\_park : GeoDataFrame  
 各个公园的潜在人口服务压力均值.  
  
 '''   
 from tqdm import tqdm  
 import pandas as pd  
 import geopandas as gpd  
 from shapely.geometry import Point  
   
 stations\_pts={k:Point(v) for k,v in nx.get\_node\_attributes(G\_SB,"position").items()}  
 populationWeighted\_adjacentStations\_dict={}  
 for park,population\_flow in tqdm(population\_flow\_dict.items()):  
 populationGWeighted\_group\_sum=population\_flow[['idx\_adjacent','population\_Gweighted']].groupby(by=['idx\_adjacent']).sum()  
 populationGWeighted\_group\_sum.reset\_index(inplace=True)  
 populationGWeighted\_group\_sum.rename(columns={'index':'idx\_adjacent'})  
 populationGWeighted\_group\_sum['geometry']=populationGWeighted\_group\_sum.idx\_adjacent.apply(lambda row:stations\_pts[row])  
 populationWeighted\_adjacentStations\_dict[park]=populationGWeighted\_group\_sum   
   
 populationWeighted\_adjacentStations\_stack=pd.concat(populationWeighted\_adjacentStations\_dict.values(),keys=populationWeighted\_adjacentStations\_dict.keys())  
 populationWeighted\_adjacentStations\_stack['populationWeighted\_10thous']=populationWeighted\_adjacentStations\_stack.population\_Gweighted.apply(lambda row:row/10000)  
 populationWeighted\_adjacentStations\_stack\_gdf=gpd.GeoDataFrame(populationWeighted\_adjacentStations\_stack,crs="EPSG:{}".format(epsg))  
 populationWeighted\_parks={k.split("\_")[-1]:v.population\_Gweighted.sum() for k,v in populationWeighted\_adjacentStations\_dict.items()}   
   
 comprehensive\_park['populationWeighted']=comprehensive\_park['Name\_EN'].map(populationWeighted\_parks)   
 comprehensive\_park['populationWeighted\_10thous']=comprehensive\_park.populationWeighted.apply(lambda row:row/10000)  
   
 return populationWeighted\_adjacentStations\_dict,populationWeighted\_adjacentStations\_stack\_gdf,comprehensive\_park

populationWeighted\_adjacentStations\_dict,populationWeighted\_adjacentStations\_stack\_gdf,comprehensive\_park=populationWeighted\_adjacentStations(population\_flow\_dict,G\_SB,args.gi.nanjing\_epsg,comprehensive\_park)

100%|██████████| 53/53 [00:00<00:00, 231.02it/s]

populationWeighted\_adjacentStations\_stack\_gdf.head(3)

|  |  | **idx\_adjacent** | **population\_Gweighted** | **geometry** | **populationWeighted\_10thous** |
| --- | --- | --- | --- | --- | --- |
| **0\_Bazi Mountain** | **0** | 1585d76c5555a636f32d839c | 1.168231e+03 | POINT (664131.588 3551415.123) | 0.116823 |
| **1** | 2077fa870b9bbdb57adbd2c1 | 3.889698e+06 | POINT (665061.514 3551596.830) | 388.969841 |
| **2** | 3176ab8f267a522adc1e5249 | 1.494966e+04 | POINT (664855.439 3551547.768) | 1.494966 |

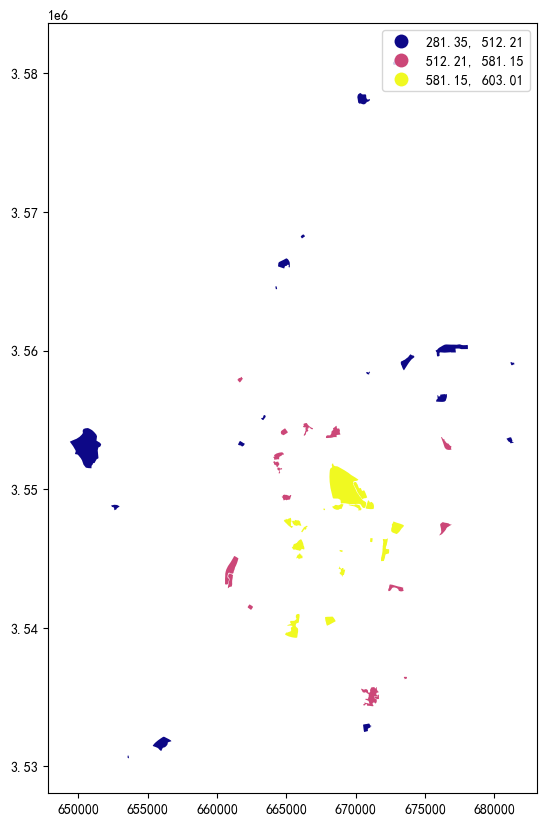
comprehensive\_park.head(3)

|  | **Name** | **Description** | **geometry** | **group** | **Name\_EN** | **populationWeighted** | **populationWeighted\_10thous** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 八字山公园 | None | POLYGON Z ((664351.229 3551540.582 0.000, 6643... | NanjingParks | Bazi Mountain | 5.662951e+06 | 566.295130 |
| **1** | 北崮山公园 | None | POLYGON Z ((667855.803 3553692.487 0.000, 6678... | NanjingParks | Beigu Mountain | 5.624582e+06 | 562.458167 |
| **2** | 大桥公园 | None | POLYGON Z ((664898.358 3554394.776 0.000, 6648... | NanjingParks | Bridge park | 5.411941e+06 | 541.194094 |

公园潜在的人口服务压力是公园可达性的一种表现，较大的值表明具有较好的可达性，反之可达性较弱。配置参数scheme='quantiles'，并配置k=3，将人口压力分成3个层级：第1个层级，具有较高的人口压力，位于公共交通复合网络的核心区域，即城市的核心区域；进而向四周扩散，形成第2个层级；第3个层级，较小的人口压力，位于网络的边缘，也是城市的边缘区域。

fig, ax=plt.subplots(figsize=(10,10))  
comprehensive\_park.plot(column='populationWeighted\_10thous',ax=ax,legend=True,cmap='plasma', scheme='quantiles',k=3)

<AxesSubplot: >



注释（Notes）：

① CSDN（Chinese Software Developer Network，（<https://www.csdn.net/>）。

② GeoLife数据集，（<https://www.microsoft.com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide/>）。

③ NetWorkX库的最短路径算法，（<https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html>）。