### 简介

充电站选址规划算法将使用城市各类数据及仿真结果进行城市充电站的选址，包括（1）根据城市多源数据包括，充电站基础设施如充电桩数、功率、充电需求热力数据如车流量、人流量、充电站地理位置信息如POI等数据进行选址规划。（2）根据MR热力数据及POI数据进行选址。（3）根据仿真结果，建立选址优化模型，进行选址。

### 算法输入输出

1. 输入：城市栅格级数据或仿真结果
2. 输出：城市栅格级充电选址评级及推荐点位

### 技术细节

（1）Grid.ipynb为根据广州、重庆、上海等城市的充电站信息及POI信息进行选址。具体说明如下：

1. read\_data(filename: str, select\_city: str) -> pd.DataFrame

作用:

从指定的 Excel 文件中读取数据，并根据所选城市名称过滤出该城市的相关数据。

输入:

• filename: 包含数据的 Excel 文件路径。

• select\_city: 需要选择的城市名称。

输出:

• 返回过滤后的 pd.DataFrame，包含选定城市的数据。

2. calculate\_scores(df: pd.DataFrame, col: pd.Index) -> pd.DataFrame

作用:

对指定列进行归一化处理（使用 MinMaxScaler），将结果转换为百分制评分，并计算所有评分列的总分。

输入:

• df: 原始数据集 pd.DataFrame。

• col: 需要进行归一化和评分的列名列表。

输出:

• 包含新的评分列以及总评分列的 pd.DataFrame。

3. extract\_coordinates(df: pd.DataFrame) -> pd.DataFrame

作用:

从 "栅格编号" 列中提取经纬度信息，将其分为 "lon" 和 "lat" 列。

输入:

• df: 包含 "栅格编号" 列的 pd.DataFrame。

输出:

• 包含经纬度信息（lon 和 lat 列）的更新 pd.DataFrame。

4. create\_geodataframe(df: pd.DataFrame) -> gpd.GeoDataFrame

作用:

将普通的 pd.DataFrame 转换为 GeoDataFrame，其中包含几何点信息（经纬度）。

输入:

• df: 包含经纬度信息的 pd.DataFrame。

输出:

• gpd.GeoDataFrame，包含几何信息（点）并与原始数据集相关联。

5. score\_grid(df: pd.DataFrame, bounds: tuple) -> pd.DataFrame

作用:

基于输入的区域边界 (bounds) 生成格网，并根据栅格编号对每个格网计算总得分。

输入:

• df: 包含经纬度和总得分的 pd.DataFrame。

• bounds: 用于定义区域边界的坐标范围，格式为元组（tuple）。

输出:

• 返回包含总得分的格网 pd.DataFrame，只保留有得分的格网。

6. classify\_grid(grid: pd.DataFrame) -> pd.DataFrame

作用:

根据格网的总得分对其进行降序排序，并将格网划分为 7 个评分等级。

输入:

• grid: 包含总得分的格网 pd.DataFrame。

输出:

• 按总得分排序并分类后的 pd.DataFrame，增加了评分等级列 num。

7. select\_best\_sites(df: pd.DataFrame, grid: pd.DataFrame) -> pd.DataFrame

作用:

根据分类后的格网数据，从每个栅格中选择评分最高的站点。

输入:

• df: 原始数据集，包含坐标和总得分。

• grid: 已分类的格网数据集，包含评分等级。

输出:

• 返回按评分等级和总得分筛选出的最佳站点 pd.DataFrame。

8. main(filename: str, select\_city: str, bounds: tuple) -> pd.DataFrame

作用:

执行完整的站点评分流程，从读取数据到筛选出最佳站点，并返回最终结果。

输入:

• filename: 数据文件路径。

• select\_city: 选择的城市名称。

• bounds: 定义的区域边界坐标范围，格式为元组。

输出:

• 最终筛选出的最佳站点 pd.DataFrame，包含每个栅格中的最佳站点信息。

具体代码如下：

import pandas as pd

import geopandas as gpd

from sklearn.preprocessing import MinMaxScaler

import transbigdata as tbd

# 读取数据文件，选择城市

def read\_data(filename, select\_city):

"""

输入:

filename (str): 文件路径

select\_city (str): 选择的城市名称

输出:

pd.DataFrame: 过滤后的数据集，仅包含选定城市的数据

"""

df = pd.read\_excel(filename)

return df[df["市"] == select\_city]

# 对每个列进行MinMax标准化，并计算得分

def calculate\_scores(df, col):

"""

输入:

df (pd.DataFrame): 输入的数据集

col (pd.Index): 需要进行标准化的列

输出:

pd.DataFrame: 返回包含标准化得分的新数据集

"""

for c in col:

scaler = MinMaxScaler()

new\_col = c + "\_score"

df[new\_col] = scaler.fit\_transform(df[[c]]) \* 100

df["total\_score"] = df[[c for c in df.columns if "score" in c]].sum(axis=1)

return df

# 提取经纬度坐标信息

def extract\_coordinates(df):

"""

输入:

df (pd.DataFrame): 输入的数据集，包含栅格编号

输出:

pd.DataFrame: 返回提取了经纬度坐标的更新数据集

"""

df["lon"] = df["栅格编号"].str.split("-", expand=True)[1].astype("float")

df["lat"] = df["栅格编号"].str.split("-", expand=True)[2].astype("float")

return df

# 将数据转换为GeoDataFrame

def create\_geodataframe(df):

"""

输入:

df (pd.DataFrame): 包含经纬度的普通DataFrame

输出:

gpd.GeoDataFrame: 包含几何信息的GeoDataFrame

"""

return gpd.GeoDataFrame(df, geometry=gpd.points\_from\_xy(df["lon"], df["lat"]))

# 对栅格评分，并生成格网

def score\_grid(df, bounds):

"""

输入:

df (pd.DataFrame): 包含经纬度及评分的DataFrame

bounds (tuple): 定义区域边界的坐标

输出:

pd.DataFrame: 包含总评分的格网数据

"""

p = tbd.area\_to\_params(bounds, accuracy=5000)

grid, p = tbd.area\_to\_grid(bounds, params=p)

df["LONCOL"], df["LATCOL"] = tbd.GPS\_to\_grid(df["lon"], df["lat"], params=p)

score = df.groupby(["LONCOL", "LATCOL"])["total\_score"].sum().reset\_index()

score.columns = ["LONCOL", "LATCOL", "total\_score"]

grid = pd.merge(grid, score, on=["LONCOL", "LATCOL"], how="left")

return grid[grid["total\_score"].notna()]

# 根据评分将格网进行分类

def classify\_grid(grid):

"""

输入:

grid (pd.DataFrame): 包含格网和评分的数据

输出:

pd.DataFrame: 排序后的格网数据，并按评分分类

"""

grid.sort\_values(by="total\_score", ascending=False, inplace=True)

grid["num"] = pd.cut(grid["total\_score"], bins=7, labels=["0", "1", "2", "3", "4", "5", "6"])

return grid

# 选择最佳站点

def select\_best\_sites(df, grid):

"""

输入:

df (pd.DataFrame): 原始数据集，包含坐标和评分

grid (pd.DataFrame): 已经分类的格网数据

输出:

pd.DataFrame: 每个栅格中的最佳站点选择

"""

select = pd.merge(df[["geometry", "LONCOL", "LATCOL", "total\_score"]],

grid[["LONCOL", "LATCOL", "num"]])

select["num"] = select["num"].astype(int)

select = select.sort\_values(by=["LONCOL", "LATCOL", "total\_score"], ascending=[True, True, False])

f = select.groupby(["LONCOL", "LATCOL"]).apply(lambda x: x.head(x["num"].iloc[0])).reset\_index(drop=True)

return f

# 主函数，用于调用所有步骤

def main(filename, select\_city, bounds):

"""

输入:

filename (str): 文件路径

select\_city (str): 选择的城市

bounds (tuple): 定义区域边界的坐标

输出:

pd.DataFrame: 最终的站点选择结果

"""

df\_gz = read\_data(filename, select\_city)

cols = df\_gz.columns[4:]

df\_gz = calculate\_scores(df\_gz, cols)

df\_gz = extract\_coordinates(df\_gz)

df\_gz = create\_geodataframe(df\_gz)

grid = score\_grid(df\_gz, bounds)

grid = classify\_grid(grid)

best\_sites = select\_best\_sites(df\_gz, grid)

return best\_sites

# 调用主函数

# best\_sites = main(filename, select\_city, bounds)

（2）该工具的目的基于MR及POI数据进行选址评分。通过处理和分析与充电站、地理网格及兴趣点（POI）相关的空间数据。处理特定城市（如海口或其他城市）的数据，将其与移动无线电（MR）信号网格和POI分布信息进行合并，最终根据各类指标对不同网格区域进行评分并对其进行排名。

函数说明：

1. process\_haikou\_data(filename: str) -> pd.DataFrame：

* + 功能：处理海口市的数据，读取Excel文件，填充缺失值，筛选包含特定地理信息的行，并从数据中提取纬度和经度信息。
  + 参数：

filename：包含海口市数据的Excel文件的路径。

* + 返回：处理过的DataFrame，包含海口市的纬度和经度数据。

2. process\_other\_data(filename: str) -> pd.DataFrame：

* 功能：读取并返回非海口市的其他城市数据。
* 参数：

filename：其他城市数据的Excel文件路径。

* 返回：从Excel文件中读取的数据DataFrame。

3. to\_grid(bounds: list, df: pd.DataFrame) -> tuple：

* 功能：将输入数据中的纬度和经度转换为网格坐标。使用预定义的地理边界和500米的网格精度进行转换。
* 参数：

bounds：定义区域边界的列表，[最小经度，最小纬度，最大经度，最大纬度]。

df：包含经纬度信息的DataFrame。

* 返回：返回一个包含以下内容的元组：

更新后的包含网格坐标（loncol 和 latcol）的DataFrame。

网格数据。

用于网格转换的参数。

4. process\_mrgrid(mrfile: str, p) -> pd.DataFrame：

* 功能：处理移动无线电（MR）信号数据，将数据中的经纬度信息转换为与其他数据一致的网格坐标系统。
* 参数：

mrfile：包含MR数据的CSV文件路径。

p：to\_grid函数中的网格参数。

* 返回：带有网格坐标的MR数据DataFrame。

5. process\_poi(poifile: str, city: str, p) -> pd.DataFrame：

* 功能：处理特定城市的兴趣点（POI）数据。根据指定的城市名称筛选数据，处理地理信息，并按网格聚合不同类别的POI数据。
* 参数：

poifile：包含POI数据CSV文件的目录。

city：需要筛选的城市名称。

p：来自to\_grid函数的网格参数。

* 返回：包含按网格坐标和POI类别统计的透视表DataFrame。

6. trancol(df: pd.DataFrame, col: str) -> pd.DataFrame：

* 功能：使用MinMaxScaler对指定的列进行归一化处理，计算得分（0到100分）。
* 参数：

df：包含待归一化列的DataFrame。

col：需要进行归一化处理的列名。

* 返回：添加了新得分列的DataFrame。

7. calculate\_scores(grid\_score: pd.DataFrame, num: int) -> pd.DataFrame：

* 功能：基于多个属性（如MR数据和POI数量）计算每个网格的总分，并将其进行排名。总分按10个等级进行划分，计算每个等级的比例，并根据这些比例进行排序。
* 参数：

grid\_score：包含网格得分数据的DataFrame。

num：要计算的最终数值（例如网格数量）。

* 返回：带有最终得分、排名、比例和分类的DataFrame。

8. main\_process(filename: str, bounds: list, mrfile: str, poifile: str, city: str, num: int, is\_haikou: bool)：

* 功能：该函数执行主要的数据处理任务。根据是否为海口市的数据，分别调用不同的数据处理函数，并将数据转换为网格，处理MR信号和POI数据，计算最终的网格得分并对其进行排序。
* 参数：

filename：Excel文件的路径，包含主要的城市数据。

bounds：定义区域边界的列表（[最小经度，最小纬度，最大经度，最大纬度]）。

mrfile：MR信号数据的CSV文件路径。

poifile：POI数据文件夹路径。

city：城市名称，用于筛选POI数据。

num：用于最终评分计算的参数值（例如网格的数量或需求）。

is\_haikou：布尔值，指示是否为海口市数据处理。

* 返回：包含最终评分、排名和比例的DataFrame。

通过该流程，可以从多个数据源（包括经纬度、MR信号、POI等）中提取信息，生成各网格的总评分，并用于后续的分析或决策。

具体代码如下：

import pandas as pd

import geopandas as gpd

import transbigdata as tbd

from sklearn.preprocessing import MinMaxScaler

import os

# Function to process Haikou city data

def process\_haikou\_data(filename: str) -> pd.DataFrame:

df = pd.read\_excel(filename)

df = df.fillna(method="ffill")

df = df[df["84坐标系经纬度信息"].str.contains("N")]

df["84坐标系经纬度信息"] = df["84坐标系经纬度信息"].str.replace("T", "1")

df["latitude"] = df["84坐标系经纬度信息"].str.extract(r"(\d+\.\d+)[°N]")

df["longitude"] = df["84坐标系经纬度信息"].str.extract(r"[，,.\s](\d+\.\d+)[°E]")

df["longitude"] = df["longitude"].astype("float")

df["latitude"] = df["latitude"].astype("float")

return df

# Function to process data for other cities

def process\_other\_data(filename: str) -> pd.DataFrame:

df = pd.read\_excel(filename)

return df

# Function to convert longitude and latitude to grid

def to\_grid(bounds: list, df: pd.DataFrame) -> tuple:

p = tbd.area\_to\_params(bounds, accuracy=500)

grid, p = tbd.area\_to\_grid(bounds, accuracy=500)

df["loncol"], df["latcol"] = tbd.GPS\_to\_grid(df["longitude"], df["latitude"], params=p)

grid.columns = ["loncol", "latcol", "geometry"]

return df, grid, p

# Function to process MR grid data

def process\_mrgrid(mrfile: str, p) -> pd.DataFrame:

mr = pd.read\_csv(mrfile)

mr["loncol"], mr["latcol"] = tbd.GPS\_to\_grid(mr["longitude"], mr["latitude"], params=p)

return mr

# Function to process POI data

def process\_poi(poifile: str, city: str, p) -> pd.DataFrame:

merge\_df = pd.DataFrame()

for file in os.listdir(poifile):

file\_path = os.path.join(poifile, file)

if os.path.isfile(file\_path):

df = pd.read\_csv(file\_path, encoding\_errors="ignore", encoding="gbk", skiprows=[618095])

df = df[df["cityname"] == city]

merge\_df = pd.concat([merge\_df, df])

merge\_df = merge\_df[merge\_df["page\_publish\_time"].str.contains("2018")]

merge\_df["ttype"] = merge\_df["type"].str.split(";").str.get(0)

merge\_df["lon"] = merge\_df["location"].str.split("，").str.get(0).astype("float")

merge\_df["lat"] = merge\_df["location"].str.split("，").str.get(1).astype("float")

merge\_df["loncol"], merge\_df["latcol"] = tbd.GPS\_to\_grid(merge\_df["lon"], merge\_df["lat"], params=p)

poi = merge\_df.groupby(["ttype", "loncol", "latcol"]).size().reset\_index()

poi.columns = ["ttype", "loncol", "latcol", "count"]

poi\_pivot = poi.pivot\_table(index=["loncol", "latcol"], columns="ttype", values="count", fill\_value=0).reset\_index()

return poi\_pivot

# Function to scale columns and calculate total scores

def trancol(df: pd.DataFrame, col: str) -> pd.DataFrame:

scaler = MinMaxScaler()

new\_col\_name = f"{col}\_score"

df[new\_col\_name] = scaler.fit\_transform(df[[col]]) \* 100

return df

# Function to calculate final scores

def calculate\_scores(grid\_score: pd.DataFrame, num: int) -> pd.DataFrame:

c = grid\_score.columns[3:]

for col in c:

grid\_score = trancol(grid\_score, col)

grid\_score["total\_score"] = grid\_score.filter(like="score").sum(axis=1)

grid\_score["num"] = pd.cut(grid\_score["total\_score"], bins=10, labels=[str(i) for i in range(1, 11)]).astype("int")

grid\_score["ratio"] = grid\_score["num"] / grid\_score["num"].sum()

grid\_score = grid\_score.sort\_values(by="total\_score", ascending=False)

grid\_score["rank"] = grid\_score["total\_score"].rank(ascending=False, method="dense").astype(int)

grid\_score.reset\_index(drop=True, inplace=True)

grid\_score["c\_num"] = grid\_score["ratio"] \* (num \* grid\_score["ratio"].sum())

return grid\_score

# Main process

def main\_process(filename: str, bounds: list, mrfile: str, poifile: str, city: str, num: int, is\_haikou: bool):

# Data reading and processing

df = process\_haikou\_data(filename) if is\_haikou else process\_other\_data(filename)

# Convert to grid

df, grid, p = to\_grid(bounds, df)

# Process MR grid data

mr = process\_mrgrid(mrfile, p)

df = pd.merge(df, mr, on=["loncol", "latcol"], how="left")

# Calculate grid scores

score = df[["loncol", "latcol", "总平均"]]

grid\_score = pd.merge(grid, score, on=["loncol", "latcol"])

# Process POI data

poi\_pivot = process\_poi(poifile, city, p)

grid\_score = pd.merge(grid\_score, poi\_pivot, on=["loncol", "latcol"], how="left")

grid\_score.fillna(0, inplace=True)

# Calculate final scores

final\_scores = calculate\_scores(grid\_score, num)

return final\_scores

filename = "/path/to/data.xlsx"

bounds = [110.197204, 19, 111, 21]

mrfile = "/path/to/mrfile.csv"

poifile = "/path/to/poifile/"

city = "乌鲁木齐市"

num = 10

is\_haikou = False # or True if processing Haikou

result = main\_process(filename, bounds, mrfile, poifile, city, num, is\_haikou)

（3）根据选址模型进行优化确定选址栅格

1、 处理充电站数据的函数

def process\_charging\_orders(station\_info, step\_length, params):

* 输入：

station\_info: 包含充电站信息的DataFrame，至少应包含 ['station\_id', 'lon', 'lat', 'max\_capacity', 'charge\_speed\_station', 'num\_current\_car', 'current\_car', 'waiting\_car', 'time']

step\_length: 整数，表示时间步长，用于确定充电订单的时间间隔

params: 字典，包含栅格化参数，用于将GPS坐标转换为栅格坐标

* 输出：

station\_info\_table: 处理后的充电站信息DataFrame，包含充电站的基本信息

current\_car\_infos: 处理后的当前车辆信息DataFrame，包含每个充电站当前车辆的详细信息

waiting\_car\_infos: 处理后的等待车辆信息DataFrame，包含每个充电站等待车辆的详细信息

* 目的：

清洗和处理充电站数据，提取当前车辆和等待车辆的信息，并将其转换为用于后续分析的格式。

2、 处理充电订单的函数

def get\_charging\_order(station\_info\_table, current\_car\_infos, step\_length):

* 输入：

station\_info\_table: 包含充电站基本信息的DataFrame

current\_car\_infos: 包含每个充电站当前车辆详细信息的DataFrame

step\_length: 整数，表示时间步长，用于确定充电订单的时间间隔

* 输出：

charging\_orders: 包含充电订单详细信息的DataFrame，包含每个充电订单的开始时间、结束时间和持续时间等信息

* 目的：

根据当前车辆信息计算充电订单，确定每个充电订单的开始和结束时间以及持续时间。

3、计算充电订单函数

def get\_order(charging\_orders, params, grid):

* 输入：

charging\_orders: 包含充电订单详细信息的DataFrame

params: 字典，包含栅格化参数

grid: 栅格数据，用于进行GPS坐标转换和聚合

* 输出：

charging\_order\_agg: 包含每个栅格内充电订单数量的DataFrame

* 目的：

将充电订单数据聚合到栅格级别，为后续的空间分析提供基础数据。

4、 处理潜在充电需求的函数

def process\_potential\_demand(car\_infos, step\_length, params):

* 输入：

car\_infos: 包含车辆信息的DataFrame

step\_length: 整数，表示时间步长

params: 字典，包含栅格化参数

* 输出：

stay\_order\_agg: 包含每个栅格内潜在充电需求的DataFrame

* 目的：

识别潜在的充电需求，并将这些需求聚合到栅格级别，为充电站选址提供依据。

5、处理POI数据的函数

def process\_poi(poi\_data, grid, params, poi\_types=['停车场', '加油站']):

* 输入：

poi\_data: 包含POI信息的DataFrame

grid: 栅格数据，用于进行GPS坐标转换和聚合

params: 字典，包含栅格化参数

poi\_types: 列表，包含感兴趣的POI类型，默认为['停车场', '加油站']

* 输出：

poi\_agg: 包含每个栅格内POI数量的DataFrame

* 目的：

处理POI数据，提取特定类型的POI，并将其聚合到栅格级别，为充电站选址提供参考。

6、 处理充电站利用率的函数

def calculate\_utilization(charging\_orders, grid, params):

* 输入：

charging\_orders: 包含充电订单详细信息的DataFrame

grid: 栅格数据，用于进行GPS坐标转换和聚合

params: 字典，包含栅格化参数

* 输出：

station\_chargetime\_agg: 包含每个栅格内充电站利用率的DataFrame

* 目的：

计算充电站的利用率，为充电站的运营效率分析和充电站选址提供依据。

7、 计算建站成本的函数

def process\_station\_cost(price\_data, grid, params):

* 输入：

price\_data: 包含建站成本信息的DataFrame

grid: 栅格数据，用于进行GPS坐标转换和聚合

params: 字典，包含栅格化参数

* 输出：

pricegrid: 包含每个栅格内建站成本的DataFrame

* 目的：

计算每个栅格内的建站成本，为充电站的经济性分析和选址提供依据。

8、主函数

def main():

* 输出：

gridsum: 包含充电站选址分析结果的DataFrame

* 目的：

整合上述所有函数，进行充电站选址的完整数据分析流程，输出选址分析结果。

9、数据预处理函数

def preprocess\_data(gridsum):

* 输入：

gridsum: DataFrame，包含充电站选址分析的数据。

* 输出：

df: DataFrame，预处理后的数据，满足利用率 <= 0.5 且停车位数 >= 1。

* 目的：

对原始数据进行筛选，保留满足特定条件的数据，为后续分析做准备。

10、目标函数：只考虑充电需求和潜在充电需求

def objective\_function(individual, df):

* 输入：

individual: 表示个体的列表，其中1表示选中的栅格。

df: DataFrame，预处理后的数据。

* 输出：

total\_score: 浮点数，表示选中栅格的充电需求和潜在充电需求之和。

* 目的：

计算选中栅格的总充电需求和潜在充电需求，作为优化的目标。

11、约束条件检查函数

def satisfies\_constraints(individual, df, max\_cost, max\_sites):

* 输入：

individual: 表示个体的列表，其中1表示选中的栅格。

df: DataFrame，预处理后的数据。

max\_cost: 浮点数，建站成本的上限。

max\_sites: 整数，最大选中站点数量。

* 输出：

是否满足所有约束条件的布尔值。

* 目的：

检查选中的栅格是否满足用地、充电需求满足度和建站成本的约束。

12、评价函数

def evaluate(individual, df, max\_cost, max\_sites):

* 输入：

individual: 表示个体的列表，其中1表示选中的栅格。

df: DataFrame，预处理后的数据。

max\_cost: 浮点数，建站成本的上限。

max\_sites: 整数，最大选中站点数量。

* 输出：

个体适应度值，如果满足所有约束条件则为 objective\_function 的输出，否则为0.0。

* 目的：

根据个体是否满足约束条件，计算其适应度值，用于遗传算法的选择过程。

13、初始化个体的函数

def init\_individual(icls, df, size, num\_ones, target\_cost):

* 输入：

icls: 用于创建个体的类。

df: DataFrame，预处理后的数据。

size: 整数，个体的大小。

num\_ones: 整数，个体中1的数量。

target\_cost: 浮点数，建站成本的目标值。

* 输出：

一个初始化的个体，其中包含固定数量的1。

* 目的：

初始化一个个体，确保其中包含固定数量的1，且选中的栅格满足成本约束。

14、 自定义变异函数，确保变异后仍然有固定数量的1

def mut\_shuffle\_indexes(individual, indpb):

* 输入：

individual: 表示个体的列表，其中1表示选中的栅格。

indpb: 浮点数，变异概率。

* 输出：

变异后的个体。

* 目的：

自定义变异操作，保证变异后的个体中1的数量不变。

15、初始化遗传算法工具

def init\_toolbox(df, cost, max\_sites):

* 输入：

df: DataFrame，预处理后的数据。

cost: 浮点数，建站成本的上限。

max\_sites: 整数，最大选中站点数量。

* 输出：

toolbox: 包含遗传算法所需工具的对象。

* 目的：

初始化遗传算法所需的工具箱，包括个体的创建、交叉、变异和选择方法。

16、遗传算法主函数

def genetic\_algorithm(toolbox, population\_size, generations, cxpb, mutpb):

* 输入：

toolbox: 包含遗传算法所需工具的对象。

population\_size: 整数，种群大小。

generations: 整数，遗传算法的迭代代数。

cxpb: 浮点数，交叉概率。

mutpb: 浮点数，变异概率。

* 输出：

optimal\_sites: 列表，表示最优站点的索引。

* 目的：

运行遗传算法，找到满足约束条件的最优站点选择。

17、主调用函数

def run\_site\_selection(gridsum, cost=120\*10000, population\_size=900, generations=200, cxpb=0.5, mutpb=0.2, max\_sites=100):

* 输入：

gridsum: DataFrame，包含充电站选址分析的数据。

cost: 浮点数，建站成本的上限，默认为120\*10000。

population\_size: 整数，种群大小，默认为900。

generations: 整数，遗传算法的迭代代数，默认为200。

cxpb: 浮点数，交叉概率，默认为0.5。

mutpb: 浮点数，变异概率，默认为0.2。

max\_sites: 整数，最大选中站点数量，默认为100。

* 输出：

optimal\_grid\_data: DataFrame，包含最优站点的数据。

* 目的：

运行充电站选址的完整流程，包括数据预处理、遗传算法的初始化和执行，最终输出最优站点的选择结果。

具体代码如下：

import pandas as pd

import geopandas as gpd

import ast

import transbigdata as tbd # 假设tbd库有必要的GPS\_to\_grid和area\_to\_grid函数

# 定义处理充电站数据的函数

def process\_charging\_orders(station\_info, step\_length, params):

station\_info\_table = station\_info[['station\_id','lon','lat','max\_capacity','charge\_speed\_station']].drop\_duplicates().copy()

station\_info = station\_info[station\_info['num\_current\_car'] > 0]

station\_info['time'] = pd.to\_datetime(station\_info['time'])

station\_info['current\_car'] = station\_info['current\_car'].apply(lambda a: ast.literal\_eval(a))

station\_info['waiting\_car'] = station\_info['waiting\_car'].apply(lambda a: ast.literal\_eval(a))

station\_info.sort\_values(by=['station\_id', 'time'], inplace=True)

current\_car\_infos = station\_info[['station\_id', 'time', 'current\_car']].explode('current\_car')

current\_car\_infos = current\_car\_infos[~current\_car\_infos['current\_car'].isnull()]

current\_car\_infos = current\_car\_infos.sort\_values(by=['current\_car', 'time'])[['current\_car', 'time', 'station\_id']]

waiting\_car\_infos = station\_info[['station\_id', 'time', 'waiting\_car']].explode('waiting\_car')

waiting\_car\_infos = waiting\_car\_infos[~waiting\_car\_infos['waiting\_car'].isnull()]

waiting\_car\_infos = waiting\_car\_infos.sort\_values(by=['waiting\_car', 'time'])[['waiting\_car', 'time', 'station\_id']]

return station\_info\_table , current\_car\_infos, waiting\_car\_infos

# 计算充电订单

def get\_charging\_order(station\_info\_table,current\_car\_infos, step\_length):

current\_car\_infos['timegap'] = current\_car\_infos['time'].diff().dt.total\_seconds().fillna(1000000).astype(int)

current\_car\_infos['order\_id'] = (current\_car\_infos['timegap'] > step\_length).cumsum()

charge\_info\_s = current\_car\_infos.groupby(['current\_car', 'order\_id']).first().reset\_index()

charge\_info\_e = current\_car\_infos.groupby(['current\_car', 'order\_id']).last().reset\_index()

charging\_order = pd.merge(charge\_info\_s, charge\_info\_e, on=['current\_car', 'order\_id', 'station\_id'])

charging\_order = charging\_order[['current\_car', 'order\_id', 'time\_x', 'time\_y', 'station\_id']]

charging\_order.columns = ['carid', 'order\_id', 'stime', 'etime', 'station\_id']

charging\_order['duration'] = (charging\_order['etime'] - charging\_order['stime']).dt.total\_seconds()

charging\_order = charging\_order[charging\_order['duration'] > 0]

charging\_orders = pd.merge(charging\_order,station\_info\_table)

# 计算充电时长

charging\_orders['duration'] = (charging\_orders['etime']-charging\_orders['stime']).dt.total\_seconds()

return charging\_orders

def get\_order(charging\_orders,params,grid):

charging\_order = charging\_orders

charging\_order['LONCOL'],charging\_order['LATCOL'] = tbd.GPS\_to\_grid(charging\_order['lon'],charging\_order['lat'],params=params)

charging\_order\_agg = charging\_order.groupby(['LONCOL','LATCOL'])['carid'].count().reset\_index()

charging\_order\_agg = pd.merge(grid[['LONCOL','LATCOL']],charging\_order\_agg,on=['LONCOL','LATCOL'],how="left")[['LONCOL','LATCOL','carid']]

charging\_order\_agg.fillna(0,inplace=True)

return charging\_order\_agg

# 处理潜在充电需求

def process\_potential\_demand(car\_infos, step\_length, params):

car\_infos['time'] = pd.to\_datetime(car\_infos['time'])

stay\_infos = car\_infos.sort\_values(by=['carid', 'time'])

stay\_infos['timegap'] = (-stay\_infos['time'].diff(-1).dt.total\_seconds()).fillna(1000000).astype(int)

stay\_infos['etime'] = stay\_infos['time'].shift(-1)

stay\_order = stay\_infos[stay\_infos['timegap'] > step\_length][['carid', 'time', 'etime', 'soc', 'lon', 'lat']]

stay\_order.columns = ['carid', 'stime', 'etime', 'soc', 'lon', 'lat']

stay\_order['duration'] = (stay\_order['etime'] - stay\_order['stime']).dt.total\_seconds()

stay\_order = stay\_order[(stay\_order["soc"] <= 50) & (stay\_order["duration"] >= 60 \* 5)]

stay\_order['LONCOL'], stay\_order['LATCOL'] = tbd.GPS\_to\_grid(stay\_order['lon'], stay\_order['lat'], params=params)

stay\_order\_agg = stay\_order.groupby(['LONCOL', 'LATCOL']).size().reset\_index()

stay\_order\_agg.columns=['LONCOL','LATCOL',"pdemand"]

return stay\_order\_agg

# 处理 POI 数据

def process\_poi(poi\_data, grid, params, poi\_types=['停车场', '加油站']):

poi\_data = poi\_data[poi\_data["pname"] == "上海市"]

poi\_data['ttype'] = poi\_data['type'].apply(lambda x: x.split(';')[0])

poi\_data["lon"] = poi\_data['location'].apply(lambda x: x.split(',')[0])

poi\_data["lat"] = poi\_data['location'].apply(lambda x: x.split(',')[1])

poi\_data = poi\_data[poi\_data["type"].str.contains('|'.join(poi\_types))]

poi\_data["lon"] = poi\_data["lon"].astype("float")

poi\_data["lat"] = poi\_data["lat"].astype("float")

poi\_data['LONCOL'], poi\_data['LATCOL'] = tbd.GPS\_to\_grid(poi\_data['lon'], poi\_data['lat'], params=params)

poi\_agg = poi\_data.groupby(['LONCOL', 'LATCOL']).size().reset\_index()

poi\_agg = pd.merge(poi\_agg, grid[['LONCOL', 'LATCOL']], on=['LONCOL', 'LATCOL'], how="right")

poi\_agg.columns = ['LONCOL', 'LATCOL', "park"]

poi\_agg.fillna(0, inplace=True)

return poi\_agg

# 处理充电站的利用率

def calculate\_utilization(charging\_orders, grid, params):

"""

计算充电站的充电需求满足度（利用率）。

参数:

- charging\_orders: 包含充电订单的DataFrame，至少应包含 ['stime', 'station\_id', 'lon', 'lat', 'max\_capacity', 'duration']

- grid: 栅格数据，用于进行GPS坐标转换和聚合。

- params: 栅格化参数。

返回:

- station\_chargetime\_agg: DataFrame, 包含聚合后的利用率数据，按栅格坐标（LONCOL, LATCOL）。

"""

# 确定充电订单中的最早和最晚时间

start\_time = charging\_orders["stime"].min()

end\_time = charging\_orders["stime"].max()

duration = end\_time - start\_time

# 计算每个充电站的充电时长（duration）

station\_chargetime = charging\_orders.groupby(["station\_id", "lon", "lat", "max\_capacity"])["duration"].sum().reset\_index()

# 转换为timedelta格式

station\_chargetime["duration"] = pd.to\_timedelta(station\_chargetime["duration"], unit='s')

# 计算充电站的利用率

station\_chargetime["uti"] = station\_chargetime["duration"] / (duration \* station\_chargetime["max\_capacity"])

# 将站点的经纬度转换为栅格坐标

station\_chargetime['LONCOL'], station\_chargetime['LATCOL'] = tbd.GPS\_to\_grid(station\_chargetime['lon'], station\_chargetime['lat'], params=params)

# 按照栅格（LONCOL, LATCOL）聚合充电站利用率

station\_chargetime\_agg = station\_chargetime.groupby(['LONCOL', 'LATCOL']).mean().reset\_index()

# 将聚合后的利用率数据与栅格进行合并

station\_chargetime\_agg = pd.merge(station\_chargetime\_agg, grid[['LONCOL', 'LATCOL']], on=['LONCOL', 'LATCOL'], how="right")[['LONCOL', 'LATCOL', 'uti']]

# 处理空值情况

station\_chargetime\_agg.fillna(0, inplace=True)

return station\_chargetime\_agg

# 计算建站成本

def process\_station\_cost(price\_data, grid, params):

price\_data["geometry"] = gpd.points\_from\_xy(price\_data["lon"], price\_data["lat"])

price\_data = gpd.GeoDataFrame(price\_data, geometry=price\_data["geometry"])

price\_data.crs = "EPSG:4326"

price\_data = price\_data.to\_crs("EPSG:32651")

buffer = price\_data.buffer(1000)

price\_data = gpd.GeoDataFrame(price\_data, geometry=buffer)

pricegrid = grid.to\_crs("EPSG:32651")

pricegrid = gpd.sjoin(pricegrid, price\_data)

pricegrid = pricegrid.groupby(["LONCOL", "LATCOL"])["price"].mean().reset\_index()

pricegrid["price"] = pricegrid["price"] \* 100 \* 0.02 \* 0.1 \* 20 + 200000 + 400000 + 0.4 \* (pricegrid["price"] \* 100 \* 0.02 \* 0.1 \* 20 + 200000 + 400000)

pricegrid = pd.merge(pricegrid, grid, how="right", on=['LONCOL', 'LATCOL'])

pricegrid.fillna(pricegrid["price"].min(), inplace=True)

pricegrid = pricegrid[['LONCOL', 'LATCOL', "price"]]

return pricegrid

# 主函数

def main():

# 加载数据

station\_info = pd.read\_csv(r'station\_infos.csv')

taz = gpd.read\_file(r" taz.geojson")

car\_infos = pd.read\_csv('car\_infos.csv')

poi = pd.read\_excel(r"gd\_310000\_poi.xlsx")

price = pd.read\_csv(r"上海房价.csv")

gridfile = r" gridsum.csv"

gridgejson = r" grid.geojson"

# 设置栅格化参数

paramssh = {'slon': 120.88125, 'slat': 30.7125, 'deltalon': 0.0125, 'deltalat': 0.008333, 'theta': 0, 'method': 'rect', 'gridsize': 1000}

grid, paramssh = tbd.area\_to\_grid(taz, params=paramssh)

# 处理订单数据

step\_length = 5 \* 60

station\_info\_table ,current\_car\_infos, waiting\_car\_infos = process\_charging\_orders(station\_info, step\_length, paramssh)

charging\_orders = get\_charging\_order(station\_info\_table,current\_car\_infos, step\_length)

#处理充电需求数据

charging\_order\_agg=get\_order(charging\_orders,paramssh,grid)

# 处理潜在充电需求

stay\_order\_agg = process\_potential\_demand(car\_infos, step\_length, paramssh)

# 处理POI

poi\_agg = process\_poi(poi, grid, paramssh)

# 处理充电站利用率

station\_chargetime\_agg = calculate\_utilization(charging\_orders, grid, paramssh)

# 计算建站成本

pricegrid = process\_station\_cost(price, grid, paramssh)

gridsum=pd.merge(charging\_order\_agg,stay\_order\_agg,on=['LONCOL','LATCOL'])

gridsum=pd.merge(gridsum,poi\_agg,on=['LONCOL','LATCOL'])

gridsum=pd.merge(gridsum,station\_chargetime\_agg,on=['LONCOL','LATCOL'])

gridsum=pd.merge(gridsum,pricegrid,on=['LONCOL','LATCOL'])

gridsum.columns=['LONCOL','LATCOL',"demand","pdemand","park","uti","price"]

gridsum.to\_csv(gridfile,index=False)

grid.to\_file(gridgejson)

# 数据汇总（如果需要后续处理，可以在这里继续处理或保存）

return gridsum

# 运行主函数

if \_\_name\_\_ == "\_\_main\_\_":

grid = main()