UNIVERSITY OF SCIENCE,
HO CHI MINH

Query big data in JSON file with Python using OOP to optimize the handling process
Using GEOJSON to make a whole view of the bus's path in HCM city
Functions in Shapely, optimization with Rtree, and further improvements with LLMs

TECHNICAL REPORT
W06

submitted for the Solo Project in Semester 2 – First Year

IT DEPARTMENT

Computer Science

by

Phan Tuan Kiet

Full name: Phan Tuan Kiet
Class: 23APCS2
ID: 23125062
Task achieved: 06/06

Lecturer:
Professor Dinh Ba Tien (PhD)
Teaching Assistants:
Mr. Ho Tuan Thanh (MSc)
Mr. Nguyen Le Hoang Dung (MSc)

2024

Contents

I) Structure

- *There are four py files in the program:*

+ main.py: Create User Interface and task handling.

+ path.py: Create class Path and PathQuery.

+ function.py: Display the user interface and make functions to handle queries.

+ changeAttitudes.py: Change coordinates from EPSG:4326 to EPSG:3405.

+ llm.py: Use to run prompt with LLM using Pytorch framework and transformer library (additional).

- *Drawio.png:*

+ This is the class diagram for Path's class and PathQuery's class. They are connected by a straight line, which stands for having a connection with each other but not necessarily something specific.

II) Comments and Initial Approach

- With the flow of last week's process, we know that this is the path of the bus map in Ho Chi Minh City (may be enlarged to another area outside the city).

- In the paths.json, each RouteVar-RouteVarId will have a list of latitude and longitude, which stands for the path of this route. The bus will pass these coordinates and we see that we can make a GEOJSON file to make a clear view of the bus route.

- Another notice is that the latitude and longitude of each route are not in pairs, therefore we have to make a pair using zip in Python.

- For the process of handling data, we see that we can make each [Lat-Lng] or point in the route become the object of the Path class, each object will have for attitudes: Lat, Lng, RouteId, RouteVarId. Another way to do this is to make each Route an object of the class Path, but this will be hard for the query process so we will do it with the initial approach.

- The interesting thing about this week is that we can make a GEOJSON file to put it in the real map so that we can have a broader view of the bus's path.

- The coordinates given are in latitude and longitude, therefore we can use pyproj to convert those into x and y so that we can count the distance easily.

III) Process of handling data and making class

*** Library use in the program: JSON, CSV, pandas, pyproj**

A) Path class

- Name of class: Path.

- Constructor:

```
class Path:
    def __init__(self, lat, lng, RouteId, RouteVarId):
        self._lat = lat
        self._lng = lng
        self._RouteId = RouteId
        self._RouteVarId = RouteVarId
```

+ Each object of Path's class will have four attributes: lat, lng, RouteId, and RouteVarId. Similarly, like var and stop, we will protect these attributes so that they can be only used by their subclasses and inside the class, making the class safe.

- Property:

```
@property
def lat(self):
    return self._lat
@lat.setter
def lat(self, lat):
    self._lat = lat
```

+ Make get and set method by using @property decoration.

B) PathQuery class

- Constructor:

```
def __init__(self):
    self.paths = []
    self.loadJson()
```

+ Initially, we will make a list (paths) to hold path objects, and we will load our paths.json to get the data and then handle it.

```
def __iter__(self):
    return iter(self.paths)
```

+ Similarly like var and stop, we will make the paths iterable by using iter(self. paths)

```
def loadJson(self):
    try:
        with open('jsonFile/paths.json', 'r', encoding='utf-8') as file:
            for line in file:
                data = json.loads(line)
                self.paths.append(Path(data['lat'], data['lng'], data['RouteId'], data['RouteVarId']))
    except Exception as e:
        print(f"Error loading paths.json: {str(e)}")
```

+ Function to load our JSON file.

```
def searchSiteCoordinate(self, latIn, lngIn):
    queryResult = []

    for path in self.paths:
        latCheck = False
        lngCheck = False
        for latVa in path.lat:
            if latIn == str(latVa):
                latCheck = True
                break
        for lngVa in path.lng:
            if lngIn == str(lngVa):
                lngCheck = True
                break
        if latCheck and lngCheck:
            queryResult.append(path)

    if len(queryResult) == 0:
        print("No site found!")
    else:
        return queryResult
```

+ This function is used to query latitude and longitude queries. If we find a suitable point, we will add it to the query result and then return it.

```
def searchByABC(self, **kwargs):
    queryList = ["RouteId", "RouteVarId"];
    queryResult = []

    for path in self.paths:
        match = True
        for key, value in kwargs.items():
            pathAttr = str(getattr(path, key, "Wrong key")).lower()

            if (pathAttr == "wrong key"):
                print(f"The key '{key}' is not in the list. \nPlease use one of the following: {queryL
                return
            if pathAttr != value:
                match = False
                break
        if (match):
            queryResult.append(path)

    if len(queryResult) == 0:
        print("No route found!")
    else:
        return queryResult
```

+ This function is used to search for RouteId and RouteVarId queries. We will still take advantage of the **kwargs to unpack the queries for easier handling. If the values in our queries are the same as those in paths, we will add those points to the query result and then return them.

```
def outputAsCSV(self, list):
    try:
        with open("output/outputCSV.csv", "w", newline="", encoding="utf-8") as file:
            writer = csv.writer(file)
            writer.writerow(["lat", "lng", "RouteId", "RouteVarId"])
            for path in list:
                writer.writerow([path.lat, path.lng, path.RouteId, path.RouteVarId])
        print("CSV file created successfully.")
    except Exception as e:
        print(f"Error creating CSV file: {str(e)}")
```

+ Similarly to stop and var, this function outputs CSV files.

```
def outputAsCSVByPandas(self, list):
    try:
        df = pd.DataFrame([vars(path) for path in list])
        df.columns = df.columns.str.replace('_', '')
        df.to_csv("output/outputCSVPandas.csv", index=False, encoding="utf-8")
        print("CSV file created by pandas successfully.")
    except Exception as e:
        print(f"Error creating CSV file: {str(e)}")
```

+ Still output CSV file but by panda library.

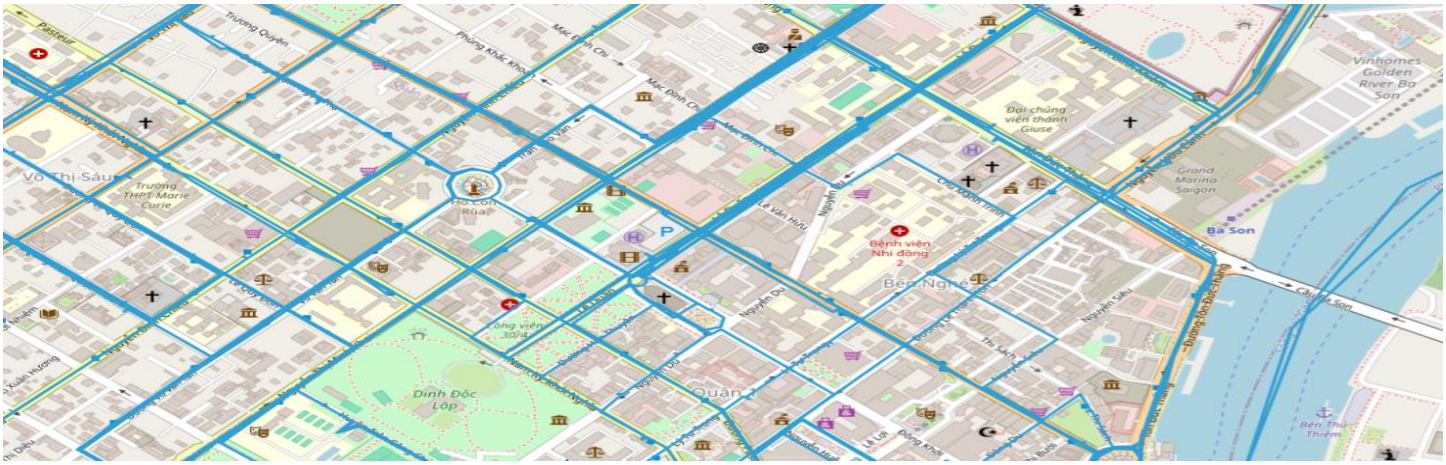
```
def outputAsJson(self, list):
    try:
        with open("output/outputJson.json", "w", encoding="utf-8") as file:
            for path in list:
                json.dump(vars(path), file, ensure_ascii = False)
                file.write('\n')
        print("JSON file created successfully.")
    except Exception as e:
        print(f"Error creating JSON file: {str(e)}")
```

+ Output the query result as a JSON file.

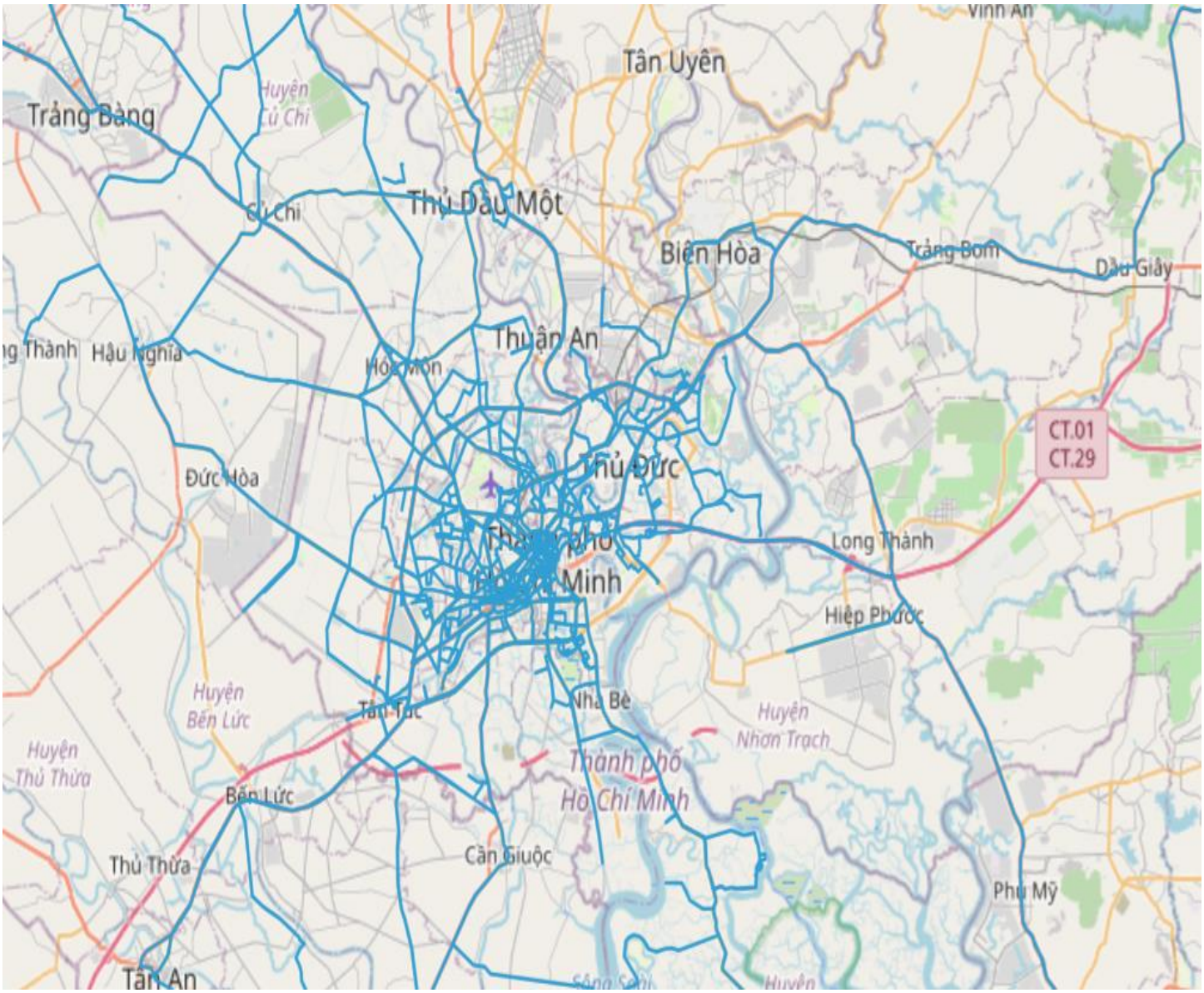
```
def outputAsGeoJson(self, list):
    try:
        with open("output/outputGeoJson.json", "w", encoding="utf-8") as file:
            file.write('{\n')
            file.write('    "type": "FeatureCollection",\n')
            file.write('    "features": [\n')
            for path in list:
                file.write('    {\n')
                file.write('      "type": "Feature",\n')
                file.write('      "properties": {},\n')
                file.write('      "geometry": {\n')
                file.write('        "coordinates": [')
                for i in range(len(path.lng)):
                    file.write('[' + str(path.lng[i]) + ',' + str(path.lat[i]) + ']')
                    if i != len(path.lng) - 1:
                        file.write(',')
                file.write('],\n')
                file.write('        "type": "LineString"\n')
                file.write('      }\n')
                file.write('    }')
                if (list.index(path) != len(list) - 1):
                    file.write(', \n')
                else:
                    file.write('\n')
            file.write('            ]\n')
            file.write('}\n')
        print("GeoJSON file created successfully.")
    except Exception as e:
        print(f"Error creating GeoJSON file: {str(e)}")
```

+ This is the special thing of this week's project. We will output the paths.json data into a GEOJSON file so that we can see the whole picture of the bus route.

C) Using GEOJSON to see the whole picture of the bus:



+ For the whole view:




```
def changeToPairs(self, data_list):
    try:
        with open("output/outputPairs.json", "w", encoding="utf-8") as file:
            for path in data_list:
                file.write('[')
                for i in range(len(path.lat)):
                    file.write('[' + str(path.lat[i]) + ',' + str(path.lng[i]) + ']')
                    if i != len(path.lat) - 1:
                        file.write(',')
                file.write(']')
                if (data_list.index(path) != len(data_list) - 1):
                    file.write(', \n')
        print("Pairs file created successfully.")
    except Exception as e:
        print(f"Error creating Pairs file: {str(e)}")
```

+ This function is used to make the list of lat and the list of lng becomes pair [Lat-Lng] for convenience.

```
[[10.77678967,106.70585632],[10.77678967,106.70585632],[10.77680874,106.70593262],[10.77631283,106.70611572],[10.77581787,106.7062912],[10.77576
[[10.89288044,106.67342377],[10.89335346,106.67333984],[10.89455509,106.67724609],[10.89550304,106.68105316],[10.89676762,106.68466187],[10.898
[[10.75125313,106.652565],[10.7512598,106.65241241],[10.75124836,106.65229797],[10.7512064,106.65213776],[10.75111198,106.65200806],[10.7510223
[[10.76767635,106.68936157],[10.76767635,106.68936157],[10.76767635,106.68936157],[10.7672224,106.68955994],[10.76747322,106.69023132],[10.7677
[[10.76767635,106.68936157],[10.76767635,106.68936157],[10.76721764,106.68955994],[10.76747322,106.69022369],[10.76772881,106.69089508],[10.767
[[10.75125313,106.652565],[10.7512598,106.65234375],[10.7512064,106.6521225],[10.75111675,106.65200806],[10.75101185,106.65193939],[10.75097466,
[[10.8437376,106.6133728],[10.84418106,106.61418152],[10.84297943,106.61500549],[10.84253693,106.61504364],[10.84233665,106.61538696],[10.840312
[[10.82361126,106.69189453],[10.82378483,106.69211578],[10.82388973,106.6920166],[10.82388973,106.6920166],[10.82637215,106.68930817],[10.825792
[[10.75125313,106.652565],[10.75123787,106.65223694],[10.75102711,106.65193176],[10.75098515,106.6509552],[10.75103283,106.65048981],[10.7515544
```


IV) User interface in the terminal and main handling

A) For the main.py

```python
if __name__ == "__main__":
    view()
    pathQuery = PathQuery()

    choice = input("Enter your choice: ")
    while True:
        if choice == "1":
            searchForSite(pathQuery)
        elif choice == "2":
            searchByABC(pathQuery)
        elif choice == "3":
            seeAllPaths(pathQuery)
        elif choice == "4":
            lat = float(input("Enter lat: "))
            lng = float(input("Enter lng: "))
            change(lat, lng)
        elif choice == "5":
            seeAllPathsInPairs(pathQuery)
        elif choice == "6":
            print("Exiting...")
            break
        else:
            print("Invalid choice")
        view()
        choice = input("Enter your choice: ")
```

+ This is used to make queries and create a user interface in the terminal

```
------------------------------API-----------------------------
Welcome to the Path API, please choose the following options:
1. Search for site coordinates (and out put as CSV, JSON, GeoJSON)
2. Search by ABC for RouteId and RouteVarId (and output as CSV, JSON, GeoJSON)
3. See all the routes (and output as CSV, JSON, GeoJSON)
4. Change lat and lng to x and y
5. See all paths in pairs and output as JSON
6. Exit
```

B) For the function.py:

```
def view():
    print("----------------------------API----------------------------")
    print("Welcome to the Path API, please choose the following options: ")
    print("1. Search for site coordinates (and out put as CSV, JSON, GeoJSON)")
    print("2. Search by ABC for RouteId and RouteVarId (and output as CSV, JSON, GeoJSON)")
    print("3. See all the routes (and output as CSV, JSON, GeoJSON)")
    print("4. Change lat and lng to x and y")
    print("5. See all paths in pairs and output as JSON")
    print("6. Exit")
```

+ Make the view of our program

```
def searchForSite(pathQuery):
    lat = str(input("Enter lat: "))
    lng = str(input("Enter lng: "))
    result = pathQuery.searchSiteCoordinate(lat, lng)
    pathQuery.outputAsCSV(result)
    pathQuery.outputAsCSVByPandas(result)
    pathQuery.outputAsJson(result)
    pathQuery.outputAsGeoJson(result)
```

+ Function to search for the site (point with the lat – long)

```
def searchByABC(pathQuery):
    args = {}
    firstTime = False
    while True:
        arg_key = input("Enter the argument key (or q to exit): ")
        if arg_key == "q":
            print("Exiting...")
            break
        firstTime = True
        arg_value = input("Enter the value for {}: ".format(arg_key))
        args[arg_key] = str(arg_value).lower()

    if (firstTime):
        print("Querying the routes with the following arguments: {}".format(args))
        result = pathQuery.searchByABC(**args)
        if (result != None):
            pathQuery.outputAsCSV(result)
            pathQuery.outputAsCSVByPandas(result)
            pathQuery.outputAsJson(result)
            pathQuery.outputAsGeoJson(result)
```

+ Function to search for the RouteId and RouteVarId

```
def seeAllPaths(pathQuery):
    result = pathQuery.paths
    pathQuery.outputAsCSV(result)
    pathQuery.outputAsCSVByPandas(result)
    pathQuery.outputAsJson(result)
    pathQuery.outputAsGeoJson(result)

def seeAllPathsInPairs(pathQuery):
    pathQuery.changeToPairs(pathQuery.paths)
```

+ See all paths in the paths.json file and make the output as CSV, JSON, or GEOJSON.

C) Example 1

```
Enter your choice: 1
Enter lat: 10.77678967
Enter lng: 106.70585632
CSV file created successfully.
CSV file created by pandas successfully.
JSON file created successfully.
GeoJSON file created successfully.
```

- Results:

```
lat,lng,RouteId,RouteVarId
"[10.77678967, 10.77678967, 10.77680874, 10.77631283, 10.77581787, 10.77576542, 10.77569103, 10.77552223, 10.77534389, 10.77518559, 10.775016
"[10.75125313, 10.7512598, 10.7512064, 10.75111675, 10.75101185, 10.75097466, 10.75017357, 10.75009441, 10.74996853, 10.75027943, 10.7502317
```


```
{
    "type": "FeatureCollection",
    "features": [
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "coordinates": [[106.70585632,10.77678967],[106.70585632,10.77678967],[106.70593262,10.77680874],[106.70611572,10.77631283],[106.706291
        "type": "LineString"
      }
    },
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "coordinates": [[106.652565,10.75125313],[106.65234375,10.7512598],[106.6521225,10.7512064],[106.65200806,10.75111675],[106.65193939,10.
        "type": "LineString"
      }
    }
                ]
}
```



```
{"_lat": [10.77678967, 10.77678967, 10.77680874, 10.77631283, 10.77581787, 10.77576542, 10.77569103, 10.77552223, 10.77534389, 10.77518559,
{"_lat": [10.75125313, 10.7512598, 10.7512064, 10.75111675, 10.75101185, 10.75097466, 10.75017357, 10.75009441, 10.74996853, 10.75027943, 10
```

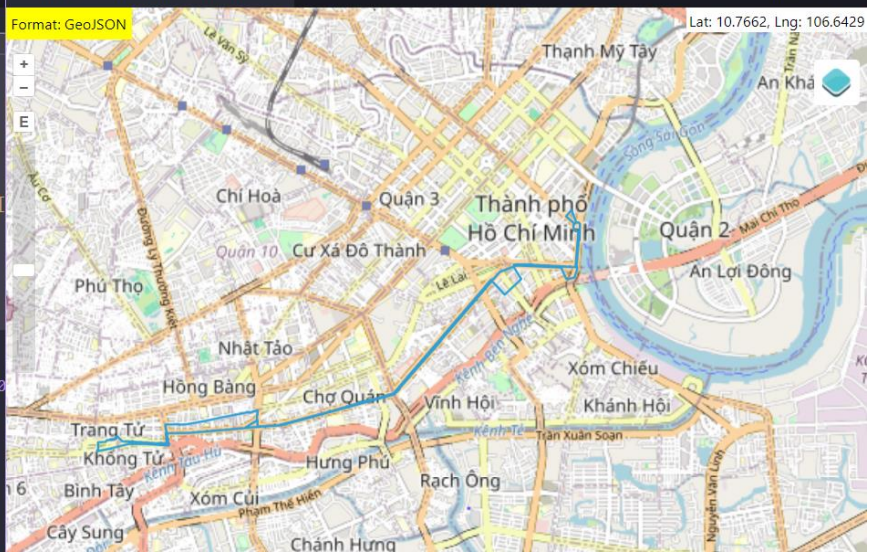
main.py | outputGeoJson.json | path.py | Map Preview: outputGeoJson.json

SourceCode > output > { } outputGeoJson.json > ...


```
{
    "type": "FeatureCollection",
    "features": [
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "coordinates": [[106.70585632,10.77678967],[
        "type": "LineString"
      }
    },
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "coordinates": [[106.652565,10.75125313],[10
        "type": "LineString"
      }
    }
                ]
}
```


Format: GeoJSON

Lat: 10.7662, Lng: 106.6429



D) Example 2

```
Enter your choice: 2
Enter the argument key (or q to exit): RouteId
Enter the value for RouteId: 1
Enter the argument key (or q to exit): RouteVarId
Enter the value for RouteVarId: 2
Enter the argument key (or q to exit): q
Exiting...
Querying the routes with the following arguments: {'RouteId': '1', 'RouteVarId': '2'}
CSV file created successfully.
CSV file created by pandas successfully.
```

- Results:

eCode > output > outputCSV.csv

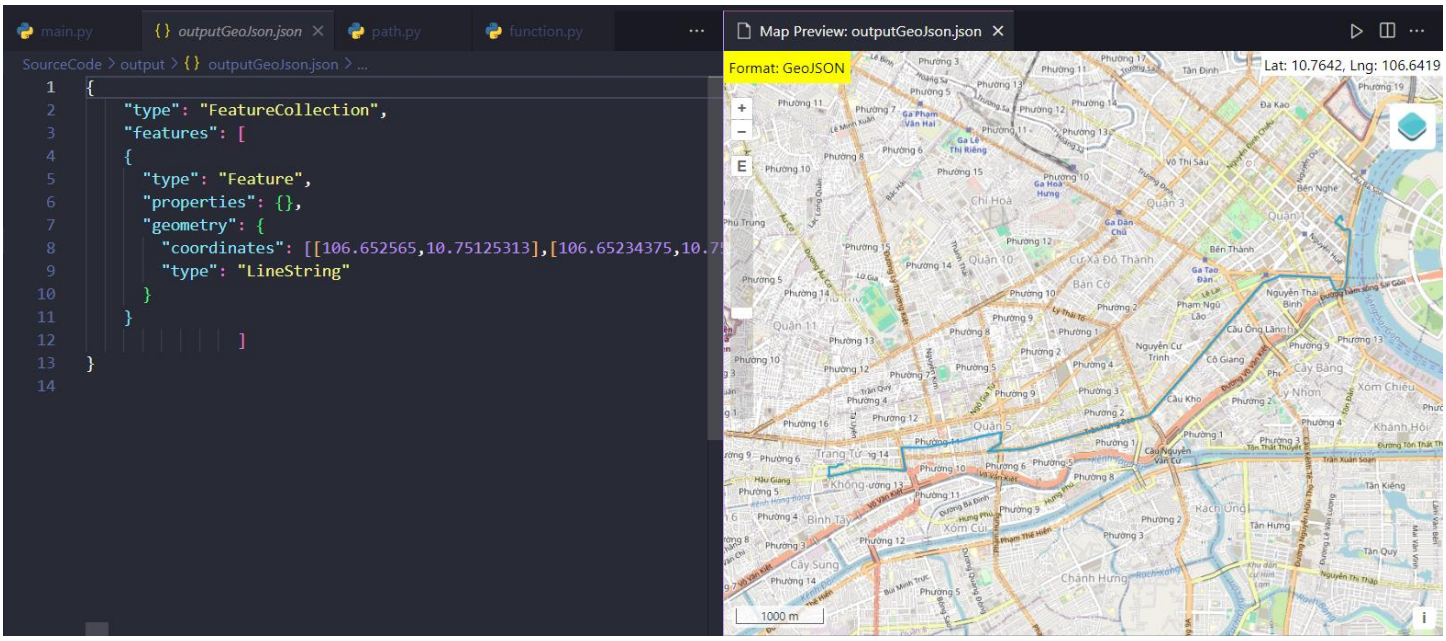

```
lat,lng,RouteId,RouteVarId
"[10.75125313, 10.7512598, 10.7512064, 10.75111675, 10.75101185, 10.75097466, 10.75017357, 10.75009441, 10.74996853, 10.75027943, 10.7502317
```


rceCode > output > { } outputGeoJson.json > ...


```
{
    "type": "FeatureCollection",
    "features": [
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "coordinates": [[106.652565,10.75125313],[106.65234375,10.7512598],[106.6521225,10.7512064],[106.65200806,10.75111675],[106.65193939,10
        "type": "LineString"
      }
    }
                ]
}
```


Code > output > { } outputJson.json > ...


```
{"_lat": [10.75125313, 10.7512598, 10.7512064, 10.75111675, 10.75101185, 10.75097466, 10.75017357, 10.75009441, 10.74996853, 10.75027943, 10.75
```



E) For the changeAttitudes.py:

```python
from pyproj import Transformer
import math

def change(lat, lng):
    lat_lng_crs = "EPSG:4326"
    target_crs = "EPSG:3405"

    transformer = Transformer.from_crs(lat_lng_crs, target_crs)

    x, y = transformer.transform(lat, lng)

    print(f"X: {x}, Y: {y}")
```

+ We will use pyproj and math to handle this task of converting coordinates.

+ We need to change from EPSG:4326 to EPSG:3405. Using Transformer.from_crs will help us do this task, and then we will need to return x and y by transformer.transform(lat,lng)

V) Functions in shapely

- Shapely is a popular Python library for geometric operations. It provides various functions for creating, manipulating, and analyzing geometric objects.

- Some of the popular functions in Shapely:

1) Creating Geometric Objects:

+ Point: Creates a point with given coordinates.

+ LineString: Creates a line string from a sequence of points.

+ Polygon: Creates a polygon from a sequence of points.

2) Geometric Operations:

+ union: Returns the union of two or more geometric objects.

+ intersection: Returns the intersection of two geometric objects.

+ difference: Returns the difference of two geometric objects.

+ buffer: Returns a buffer around the geometry.

3) Geometric Analysis:

+ area: Returns the area of a polygon.

+ length: Returns the length of a line string.

+ centroid: Returns the centroid of a geometric object.

+ boundary: Returns the boundary of a geometry.

4) Geometric Relationships:

+ contains: Checks if one geometry contains another.

+ within: Checks if one geometry is within another.

+ touches: Checks if two geometries touch each other.

+ intersects: Checks if two geometries intersect.

5) Geometric Manipulations:

+ translate: Translates a geometry by given distances in x and y directions.

+ rotate: Rotates a geometry around a specified point.

+ scale: Scales a geometry by given factors in x and y directions.

6) Constructive Methods:

+ affinity: Allows modifying geometries using affine transformations.

+ subdivide: Subdivides a geometry into smaller parts.

7) Predicates:

+ is_valid: Checks if a geometry is valid.

+ is_empty: Checks if a geometry is empty.

+ is_simple: Checks if a geometry is simple (non-self-intersecting).

8) Spatial Analysis:

+ nearest_points: Finds the nearest points between two geometries.

+ cascaded_union: Computes the union of a collection of geometries.

9) Conversion:

+ wkt: Converts a geometry to Well-Known Text (WKT) format.

+ wkb: Converts a geometry to Well-Known Binary (WKB) format.

+ geojson: Converts a geometry to GeoJSON format.

VI) Rtree in Python

- Rtree is a Python library for spatial indexing that enables efficient spatial queries on large datasets of geometric objects. It provides an interface for creating, storing, and querying spatial data using various spatial indexing techniques, such as R-tree.

- Some of the features of Rtree:

1) Spatial Indexing:

+ Rtree implements the R-tree data structure, which is optimized for spatial queries.

+ R-tree organizes spatial objects into a hierarchical structure of bounding boxes, enabling fast spatial search operations.

2) Supported Geometric Objects:

+ Rtree supports indexing of various geometric objects, including points, rectangles, lines, and polygons.

+ It allows indexing of both two-dimensional and higher-dimensional geometries.

3) Efficient Querying:

+ Rtree provides efficient spatial querying capabilities, such as range queries, nearest neighbor searches, and spatial joins.

+ Spatial indexing significantly reduces the search space, leading to faster query performance, especially for large datasets.

4) Flexible API:

+ Rtree offers a user-friendly API for creating, updating, and querying spatial indexes.

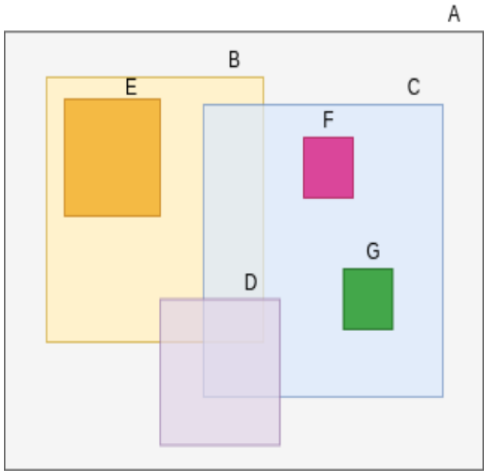
+ It supports both low-level operations for index manipulation and high-level functions for common spatial operations.

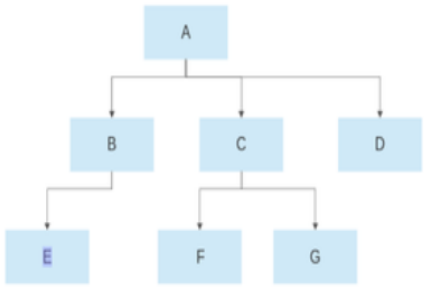
5) Integration with Other Libraries:

+ Rtree seamlessly integrates with other Python libraries commonly used in geospatial analysis, such as Shapely and GeoPandas.

+ This integration enables easy manipulation and analysis of spatial data using Rtree's indexing capabilities.

- On the example of Rtree's usage is to store spatial data indexes efficiently:





R Tree Representation

6) Some of the properties of R-tree:

+ Consists of a single root, internals nodes, and leaf nodes.

+ The root contains the pointer to the largest region in the spatial domain.

+ Parent nodes contain pointers to their child nodes where the region of child nodes completely overlaps the regions of parent nodes.

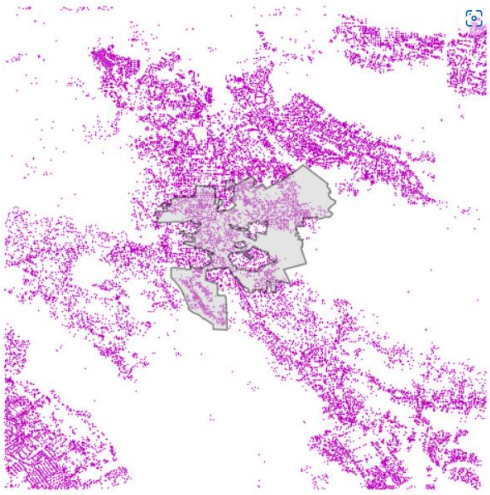
+ Leaf nodes contain data about the MBR to the current objects.

+ MBR-Minimum bounding region refers to the minimal bounding box parameter surrounding the region/object under consideration.

7) Another Spatial Indexing example using Python:

+ We have a polygon representing the city boundary of Walnut Creek, California:

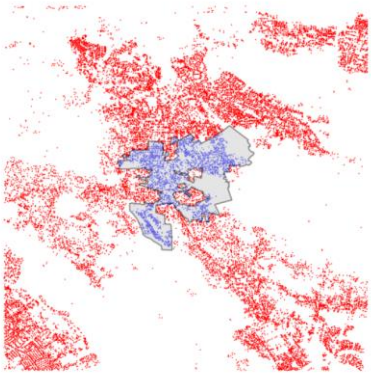
+ We also have a geopandas GeoDataFrame of lat-long points representing street intersections in the vicinity of this city. Some of these points are within the city's borders, but others are outside of them:



+ If we use the R-tree spatial index, we can find which street intersections lie within the boundaries of the city.

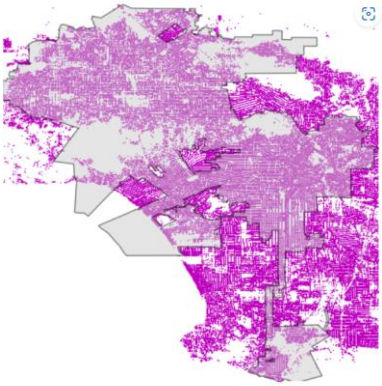
```
spatial_index = gdf.sindex
possible_matches_index = list(spatial_index.intersection(polygon.bounds))
possible_matches = gdf.iloc[possible_matches_index]
precise_matches = possible_matches[possible_matches.intersects(polygon)]
```

+ Now, we can see all of the street intersections within the city of Walnut Creek in blue and all those outside of it in red.

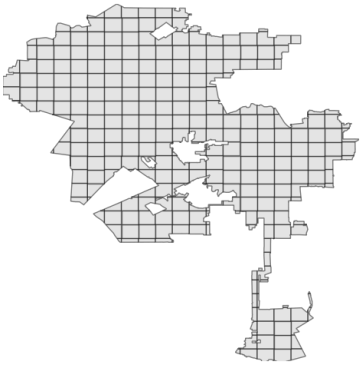


8) Advanced Spatial Indexing example using Python:

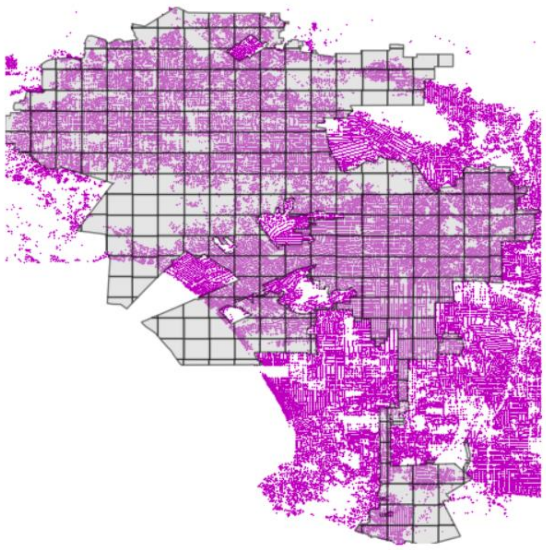
+ The R-tree index is effective in scenarios where the bounding boxes of the two feature sets being intersected or joined are distinct, as seen in the previous example involving polygons and points. Nonetheless, its performance does not improve when the bounding boxes of the features are identical. In such cases, every point is flagged as a potential match since the polygon's bounding box intersects with every nested rectangle within the index. How to solve this problem?



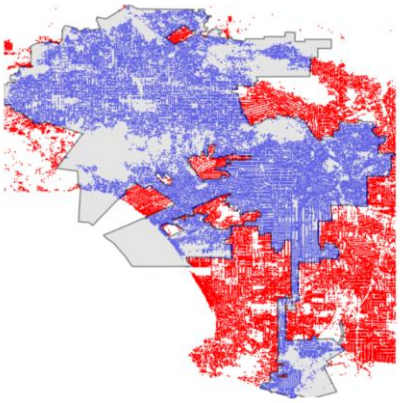
+ Fortunately, there is a way to do that using shapely. We could subdivide our polygon into smaller sub-polygons with smaller minimum bounding boxes.



+ Now, we can efficiently traverse these smaller sub-polygons to swiftly determine which points are contained within each one, utilizing the R-tree spatial index, as showcased in the preceding code snippet.



+ With the utilization of the R-tree index, the spatial intersection can now fully optimize computation time, slashing it from over 20 minutes to just a few seconds. Here, we depict all street intersections within Los Angeles in blue, while those outside the city are highlighted in red.



VII) LLM library

A) Common library

- Large Language Models (LLMs) are a type of artificial neural network trained on massive amounts of text data. This training enables them to generate human-quality text, translate languages, write different kinds of creative content, and answer your questions in an informative way. This report will focus on the capabilities of LLMs through Python libraries and frameworks.

- Some of the popular Python libraries for LLMs in python:

1) TensorFlow.js (https://www.tensorflow.org/js): While not purely a Python library, TensorFlow.js offers pre-trained LLM models like GPT-2 and allows you to run them in a web browser environment using JavaScript. This enables easy integration of LLMs into web applications developed with Python frameworks like Flask or Django.

2) Transformers (https://huggingface.co/docs/transformers/en/index): This popular library provides access to a vast collection of pre-trained LLM models from various architectures like GPT-2, BERT, and T5. It offers functionalities for fine-tuning these models on specific tasks and generating text, translating languages, and performing question answering.

3) OpenAI API (https://openai.com/): OpenAI offers access to their powerful LLM models through an API. Python libraries like gym-openai and stable-baselines3 can be used to interact with the OpenAI API for reinforcement learning tasks using LLMs.

- Some of the examples using LLM to create creative texts:

```python
from transformers import GPT2Tokenizer, GPT2LMHeadModel

model_name = "gpt2"
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)

prompt = "Once upon a time"

input_ids = tokenizer.encode(prompt, return_tensors="pt")

attention_mask = input_ids.clone().fill_(1)

output = model.generate(input_ids, attention_mask=attention_mask, max_length=200, temperature=0.7)

generated_text = tokenizer.decode(output[0], skip_special_tokens=True)

print(prompt + generated_text)
```

```
Once upon a time, the world was a place of great beauty and great danger.
```

This code snippet first loads the pre-trained GPT-2 model and tokenizer from the Transformers library. It then defines a starting prompt and encodes it using the tokenizer. The generating function of the model is used to create a continuation of the prompt text with a specified maximum length. Finally, the generated text is decoded and printed.

→ LLMs represent a significant advancement in the field of natural language processing. Python libraries and frameworks provide powerful tools for interacting with these models and exploring their capabilities. As research continues, LLMs are poised to play an increasingly important role in diverse applications.

B) Langchain

- Another library we can use to make queries based on LLM support is Langchain. This library can interact with our CSV file and therefore can answer questions about our data!

- How to implement it?

1) Install Langchain

$$pip\ install\ langchain - openai$$

2) Get the model through openai_api_key:

$$from\ langchain\_openai\ import\ ChatOpenAI$$

$$llm\ =\ ChatOpenAI(openai\_api\_key\ =\ "\dots")$$

3) Give the model some prompts:

$$from\ langchain\_core.prompts\ import\ ChatPromptTemplate$$

$$prompt\ =\ ChatPromptTemplate.from\_messages([("system", "StopID", "Code", "name"), ("user", "\{input\}")\ ])$$

4) Using Retrieval Chain for better interacting

```
from langchain_openai import ChatOpenAI

from langchain import hub

from langchain.agents import create_openai_functions_agent

from langchain.agents import AgentExecutor

# Need to set OPENAI_API_KEY environment variable or pass it as argument `openai_api_key`.

llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=.0)

agent = create_pandas_dataframe_agent(

        stops_df[['StopId', 'Code', 'Zone']],

        verbose = True

)

# Using invoke function to make LLm do its work

agent.invoke("List 5 stops have code 04")
```

VIII) Conclusion

- In this week's technical report, I successfully built a program to handle bus route data in Ho Chi Minh City.

- The program utilizes various functionalities including:

+ Data processing and class creation for storing and manipulating route information.

+ User interface for querying the data through the terminal.

+ Conversion of latitude and longitude coordinates for distance calculations.

+ Generation of output files in different formats (CSV, JSON, GEOJSON) for data visualization and analysis.

- Find information about the concept of spatial indexing with libraries like Shapely and Rtree for future implementation with larger datasets. Additionally, we introduced Large Language Models (LLMs) and their potential applications in this project, such as route description generation.