VIETNAM NATIONAL UNIVERSITY, HCMC

UNIVERSITY OF SCIENCE

FALCULTY OF INFORMATICS TECHNOLOGY

ADVANCED PROGRAM IN COMPUTER SCIENCE

TECHNICAL REPORT

Project: BUSMAP 0.5 - Full report

Subject: CS162 - Introduction to Computer Science II

Student:

Hieu Cao Thanh (23125034)

Lecturers:

Thanh Ho Tuan, M.Sc.

Dung Nguyen Le Hoang, M.Sc

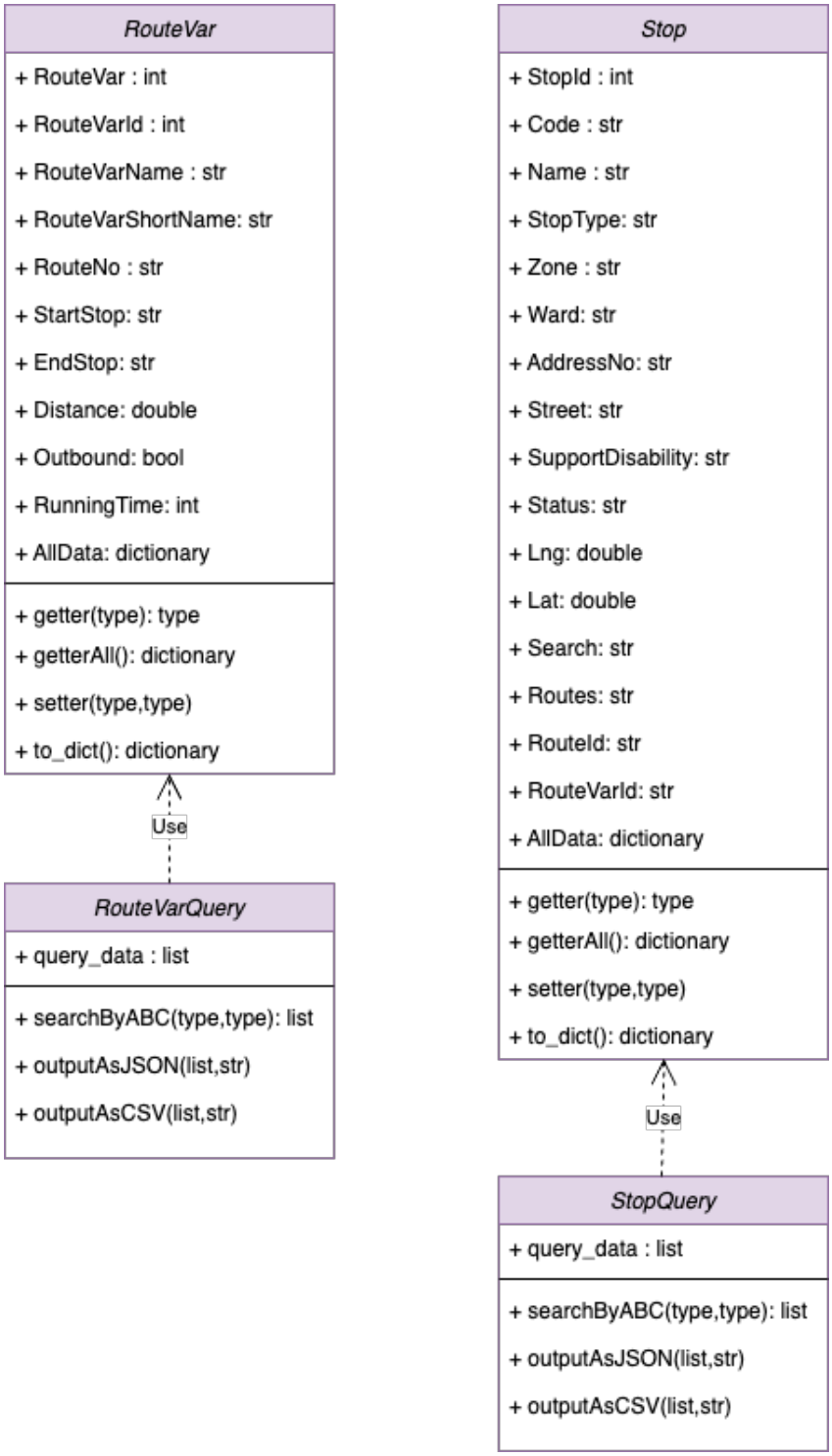
May 15, 2024

Contents

1 Week 5 - Bus Routes, Stops Query

1.1 Class Diagram

Class diagram for classes RouteVar, RouteVarQuery and Stop, StopQuery



1.2 General functions

These functions applied for both RouteVar and Stop classes.

1.2.1 Read JSON file

In order to read from JSON file, in this project, I use

```
import json
variables = json.loads()
```

With `vars.json`, each line of data contains a list of 2 route information. Therefore, I firstly read the list on each line, than break the list into dictionaries and set them to be RouteVar's objects.

```
with open("vars.json") as fileRoute:
    for eachline in fileRoute:
        content = json.loads(eachline)
        for eachdict in content:
            p = RouteVar(eachdict)
            data.append(p)
```

With `stops.json`, each line of data is a dictionary which has 3 properties `Stop, RouteId, RouteVarId`. The `Stop` property contains list of stops that need to set to be Stop's objects. I firstly read the whole dictionaries, then each of elements in the Stop's list, I add 2 properties `RouteId, RouteVarId` to make them identical.

```
with open("stops.json") as filestops:
    for eachline in filestops:
        data = json.loads(eachline)
        datalist = data['Stops']
        for eachdata in data['Stops']:
            mydict = eachdata
            mydict.update({'RouteId':data['RouteId']})
            mydict.update({'RouteVarId':data['RouteVarId']})
            s = Stop(mydict)
            data_stop.append(s)
```

1.2.2 Getter, setter

I use the same function of getter, setter for both `RouteVar, Stop` classes.

- getter: 2 functions (get object due to properties, get all data of object)
- setter: 1 functions (set properties to value)

```
    #Get functions
    def getter(self, properties):
        return self.__AllData[properties]
    def getterAll(self):
        return self.__AllData

    #Set functions
    def setter(self, properties, val):
        self.__AllData[properties] = val
        self.__RouteId = self.__AllData['RouteId']
        self.__RouteVarId = self.__AllData['RouteVarId']
        self.__RouteVarName = self.__AllData['RouteVarName']
        self.__RouteVarShortName = self.__AllData['RouteVarShortName']
        self.__RouteNo = self.__AllData['RouteNo']
        self.__StartStop = self.__AllData['StartStop']
        self.__EndStop = self.__AllData['EndStop']
        self.__Distance = self.__AllData['Distance']
        self.__Outbound = self.__AllData['Outbound']
        self.__RunningTime = self.__AllData['RunningTime']
```

1.2.3 Search by properties

The get function from above is used to implement this searchByABC() function.

```
def searchByABC(self,properties,value):
    SearchResult = []
    for eachRouteVar in self.__query_data:
        if eachRouteVar.getter(properties) == value: SearchResult.append(
   eachRouteVar)
    return SearchResult
```

1.2.4 Write to JSON/CSV file

Because variables `eachlist`'s type is RouteVar or Stop object (cannot use json.dumps or .writerow), a function to convert object to dictionary need to be implemented to use the syntax.

```
def to_dict(self):
        typedict = {'RouteId':self.__RouteId, 'RouteVarId':self.__RouteVarId,'
    RouteVarName':self.__RouteVarName,'RouteVarShortName':self.
    __RouteVarShortName,'RouteNo':self.__RouteNo,'StartStop':self.__StartStop,'
    EndStop':self.__EndStop,'Distance':self.__Distance,'Outbound':self.__Outbound
    ,'RunningTime':self.__RunningTime}
        return typedict
```

`json.dumps(dictionaries)` helps write dictionaries into JSON file.

```
def outputAsJSON(self, list, filename):
    with open(filename, "w", encoding='utf-8') as fileJSON:
        fileJSON.write( '[' + ',\n'.join (json.dumps(eachlist.to_dict(),
    ensure_ascii=False) for eachlist in list) + ']\n')
```

`.writeheader, .writerow` helps write names of fields and dictionaries into CSV file respectively.

```
def outputAsCSV(self,list,filename):
    field = ["RouteId", "RouteVarId", "RouteVarName","RouteVarShortName","
    RouteNo","StartStop","EndStop","Distance","Outbound","RunningTime"]
    with open(filename,"w",encoding='utf-8') as fileCSV:
        writer = csv.DictWriter(fileCSV, fieldnames=field)
        writer.writeheader()
        for eachlist in list:
            writer.writerow(eachlist.to_dict())
```

2 Week 6 - Convert latitude, longitude to x,y-coordinate

2.1 Coordinate Reference Systems (CRS)

`Coordinate reference system (CRS)` is a coordinate-based local, regional or global system used to locate geographical entities.

A particular CRS can be referenced by its EPSG code (i.e., EPSG:4121). The `EPSG - European Petroleum Survey Group` is a structured dataset of CRS and Coordinate Transformations, will be later used to convert lat, lng values to x,y-coordinate values.

Common-used values of EPSG:

- `EPSG:4326 - WGS 84`, lat/lng coordinate system based on the Earth's center of mass.
- `EPSG:3405 - WGS 84`, projected coordinate system for Vietnam - onshore west of 108E

2.2 Conversion with pyproj.Transformer

pyproj.Transformer has the capabilities of performing 2D, 3D, and 4D (time) transformations. It can do anything that the PROJ command line programs `proj`, `cs2cs`, and `cct` can do.

However, I chose `Transformer` over `Proj` to perform conversion from lat,lng-coordinate to x,y-coordinate because pyproj.Proj is limited to converting between geographic and projection coordinates within one datum, which means it is not a generic latitude/longitude to projection converter.

As recommended, I use pyproj.Transformer for this query:

```
from pyproj import Transformer

source_proj = pyproj.CRS.from_epsg(4326) #Earth's center of mass
target_proj = pyproj.CRS.from_epsg(3405) #Vietnam
transformer = pyproj.Transformer.from_crs(source_proj, target_proj, always_xy=
    True)

def latlng_to_xy(lat, lng):
    x, y = transformer.transform(lng, lat)
    return x, y
```

3 Week 6 - GeoJSON

3.1 Overall research

GeoJSON is a geospatial data interchange format based on JavaScript Object Notation (JSON). It defines several types of JSON objects to represent data about geographic features, their properties, and their spatial extents. GeoJSON uses a geographic coordinate reference system, World Geodetic System 1984, and units of decimal degrees.
GeoJSON structure includes `type`, `geometry`, and `properties`.

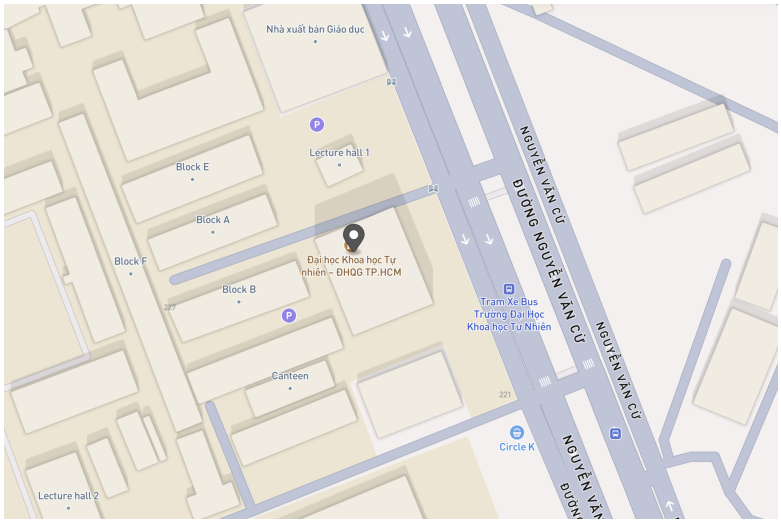
- `type`: Data type of the GeoJSON file.
  - `Feature`: represent a geometry object.
  - `FeatureCollection`: represent a set of geometry objects.
- `geometry`: objects' coordinates.
  - `Point`: a point, often used to display marker .
  - `MultiPoint`: a set of points.
  - `LineString`: represent a line.
  - `MultiLineString`: set of lines.
  - `Polygon`: a polygon, the starting point must be the same as the ending point.
  - `MultiPolygon`: set of polygons.
  - `GeometryCollection`: set of different types of geometry.
- `properties`: list of objects' property.

3.2 Write GeoJSON file

3.2.1 Examples

```
{
  "type": "FeatureCollection",
  "features": [ {
      "type": "Feature",
```

```
      "properties": {
        "name":"HCMUS" },
      "geometry": {
        "type": "Point",
        "coordinates": [
          106.68223702747679,
          10.76263209536674 ] } } ]
}
```

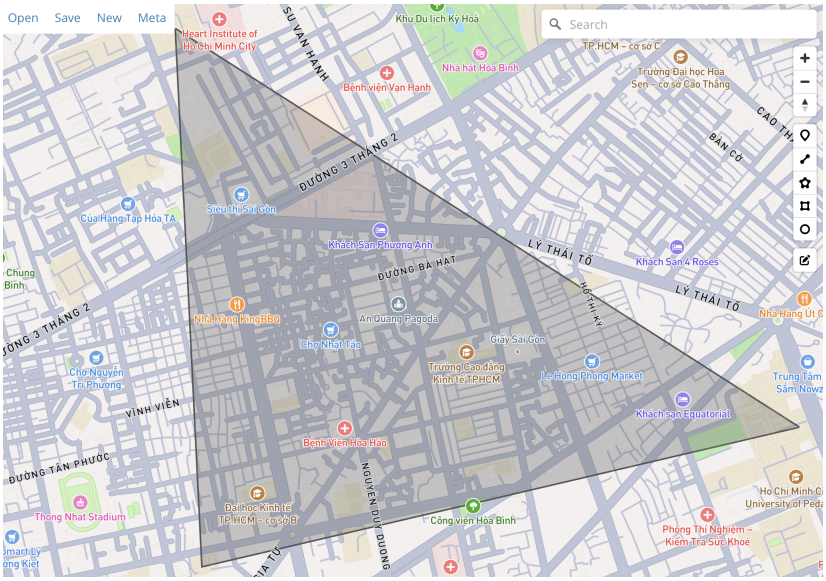


GeoJSON file for Point type

```
{
  "type": "FeatureCollection",
  "features": [ {
      "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [ 105.85120728709535, 21.034841146715365 ],
          [ 106.70345956059191, 10.772507562941314] ] },
      "properties": {
        "name": "HaNoi - HCMC" } } ]
}
```

GeoJSON file for LineString type

```
{
  "type": "FeatureCollection",
  "features": [ {
      "type": "Feature",
      "geometry": {
        "type": "Polygon",
        "coordinates": [ [
            [ 106.66649556604773, 10.759030271567084],
            [ 106.68224097988735, 10.762647462035432],
            [ 106.66579331070648, 10.7729692630718],
            [ 106.66649556604773, 10.759030271567084] ] ] },
      "properties": {
        "name": "3 schools" } } ]
}
```



GeoJSON file for Polygon type

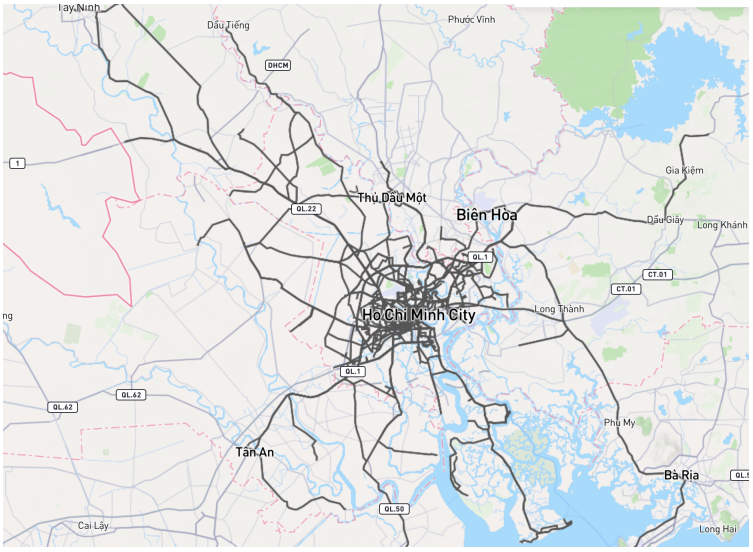
3.2.2 Visualizing bus paths from paths.json

By reading the examples and following the structure of GeoJSON, I was able to write a GeoJSON file from normal JSON file by coding in Python.

I must keep in mind that, in `paths.json`, latitude values placed before longitude values; in contrast, to display as GeoJSON file, **longitude values must be placed in front of latitude values.**

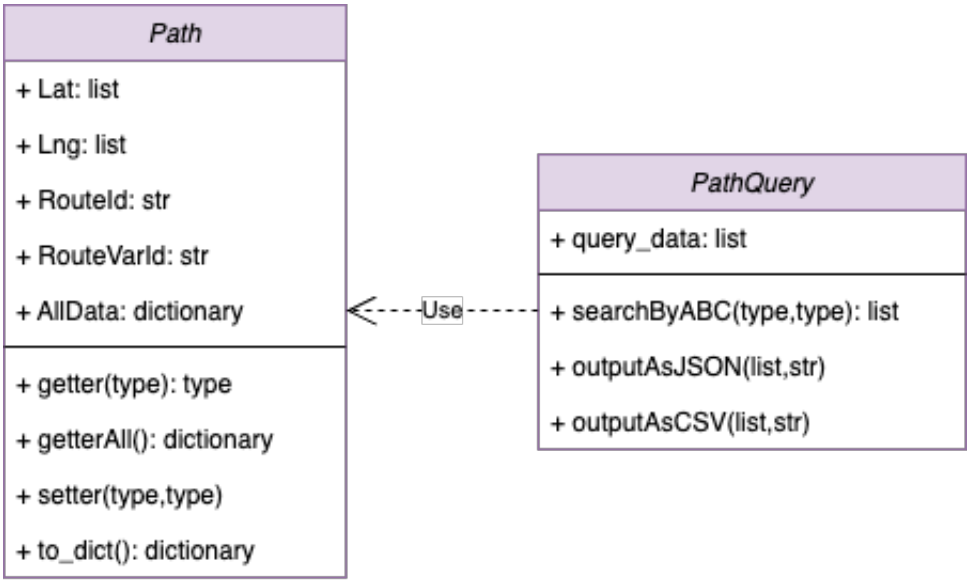
Here is my code:

```
import json
path = {"type": "FeatureCollection","features":[]}
with open("paths.json") as file:
    for eachline in file:
        coordi = []
        data = json.loads(eachline)
        datalat = data['lat']
        datalng = data['lng']
        for i in range(0,len(datalat)):
            coordi.append([datalng[i],datalat[i]])
        tmpdict = {  "type": "Feature", "geometry": { "type": "LineString",
     "coordinates": coordi}, "properties": { "RouteId": data['RouteId']," 
   RouteVarId": data['RouteVarId'] } }
        path["features"].append(tmpdict)
with open("geo.json", "w", encoding='utf-8') as fileJSON:
    json.dump(path,fileJSON,indent=2)
```



4 Week6 - Path Query

4.1 Class Diagram



4.2 Functions

The structure of Path and PathQuery class is the same as RouteVar/Stop and RouteVarQuery/StopQuery class in Week 5 respectively. Therefore, I decided not to mention it again.

Here is how I read `paths.json`

```
data_paths = []

with open("paths.json") as filepaths:
    for eachline in filepaths:
        data = json.loads(eachline)
        mypath = Path(data)
        data_paths.append(mypath)

Pathdata = PathQuery(data_paths)
```

5 Week 6 - Further Tool Research

5.1 Shapely Library

Shapely is widely used for creating, accessing, manipulating and analysing geometric objects, because of its useful operations on geometries, and provision of attributes of geometries.

Using terminal to install `shapely` with the command

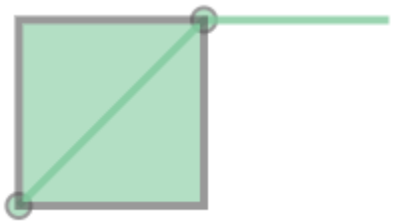
```
pip install shapely
```

Below are some common functions of `shapely`

5.1.1 Create Geometric Objects

`Shapely` provides classes for representing geometric objects such as Point, LineString, LinearRing, Polygon, and MultiPolygon. These objects can be created using coordinates and manipulated using various operations.

```
from shapely.geometry import Polygon,LineString,Point,GeometryCollection
#Create a point with coordinates x,y
point1 = Point(0, 0)
point2 = Point(1, 1)

#Create a line passing list of points
line = LineString([(0, 0), (1, 1), (2, 1)])

#Create a closed polygon with list of points
polygon = Polygon([(0, 0), (0, 1), (1, 1), (1, 0)])

#Construct a collection of geometric objects
geocollection = GeometryCollection([point1, point2, line,polygon])
```



Visualizing `geocollection` object

5.1.2 Perform Geometric Operations

`Shapely` offers a wide range of geometric operations such as union, intersection, difference, and buffering. These operations can be performed between different geometric objects to create new geometries or analyze spatial relationships.

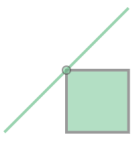
- `union()`: Combines the geometries of two or more objects.

```
from shapely.geometry import Polygon

# Create two overlapping polygons
polygon1 = Polygon([(0, 0), (0, 1), (1, 1), (1, 0)])
polygon2 = Polygon([(0.5, 0.5), (0.5, 1.5), (1.5, 1.5), (1.5, 0.5)])

# Compute the union of the polygons
union_polygon = polygon1.union(polygon2)

```

Visualizing `union_polygon` object

- `intersection()`: Finds the common region between two geometries.

```
from shapely.geometry import LineString, Polygon

# Create a LineString and a Polygon
line = LineString([(0, 0), (2, 2)])
polygon = Polygon([(1, 0), (1, 1), (2, 1), (2, 0)])

# Compute the intersection of the LineString and the Polygon
intersection = line.intersection(polygon)

```



Visualizing `line, polygon, intersection` objects

- `difference()`: Computes the region of the first geometry not covered by the second geometry.

```
from shapely.geometry import Polygon

# Create two polygons
polygon1 = Polygon([(0, 0), (0, 2), (2, 2), (2, 0)])
polygon2 = Polygon([(1, 1), (1, 3), (3, 3), (3, 1)])

# Compute the difference between polygon1 and polygon2
difference_polygon = polygon1.difference(polygon2)

```

Visualizing `difference_polygon` object

- `buffer()`: Creates a buffer region around a geometry.

```
from shapely.geometry import Point,GeometryCollection

# Create a Point
point = Point(1, 1)

# Create a buffer around the Point with a radius of 0.5
buffered_point = point.buffer(0.5)

```

Visualizing `point,buffered_point` object

Above are some examples among a huge number of Shapely's function to perform geometric operations.

5.1.3 Geometric Analysis

`Shapely` provides functionalities for geometric analysis such as computing area, length, perimeter, centroid, bounding box, and convex hull of geometric objects.
Some common-use functions to analyse geometric objects using Shapely

- `point.coords, point.x, point.y`: Show the coordinates, x-coordinate, y-coordinate of the point
- `line.length`: Calculate the length of the LineString
- `polygon.area, polygon.length, polygon.centroid`: Calculate area, perimeter and centroid of polygons
- `polygon.bounds`: Bounding Box of the Polygon (Min X, Min Y, Max X, Max Y)
- `contains(geometry)`: Check if one geometric object contains another
- `intersects(geometry)`: Check if one geometric object intersects with another

5.1.4 Other functions

- `as_polygon()`: Convert a geometric object into a polygon
- `project(crs)`: Change the coordinate system of a geometric object

5.2 RTree

A `R-Tree` is a dynamic index structure for spatial searching (Spatial Data: Objects cover areas in multidimensional spaces. Data can be points, lines and rectangles), and a height balanced tree. It is used in GeoMaps/RoadMaps, VLSI Design, Image Replica Detection (For example: Fingerprint), Astronomie Data Indexing (For example: NASA's earth observing system EOSDIS)
In Python, `RTree` is a ctypes wrapper of libspatialindex that provides a number of advanced spatial indexing features. RTree can be installed via pip.

```
pip install rtree
```

5.2.1 Creating RTree Index

To add objects to an R-Tree, the object need to be in correct format (x_min, y_min, x_max, y_max)

```
from rtree import index

# Create a new R-tree index
idx = index.Index()

# Add some rectangles to the index
idx.insert(0, (0, 0, 1, 1))
idx.insert(1, (1, 1, 2, 2))
idx.insert(2, (2, 2, 3, 3))
```

5.2.2 Querying the index

The `Intersection` function to check if objects in the RTree intersect or are contained by the given object.

```
result = list(idx.intersection((0.5, 0.5, 2.5, 2.5)))
```

The `Nearest Neighbors` function finds the 1 nearest item to the given bounds. If multiple items are of equal distance to the bounds, both are returned.

```
result = list(idx.nearest((1.0000001, 1.0000001, 2.0, 2.0), 1))
```

5.3 LLM (Large Language Models) Library for Function Calling

`Large Language Models (LLMs)` are advanced AI models trained on vast text datasets using deep learning techniques like Transformers. LLMs excel in natural language understanding and generation tasks, offering promising applications in diverse domains such as machine translation, summarization, and conversational agents. To handle such problems like choosing function from query, there are some LLMs library that are commonly use, such as:

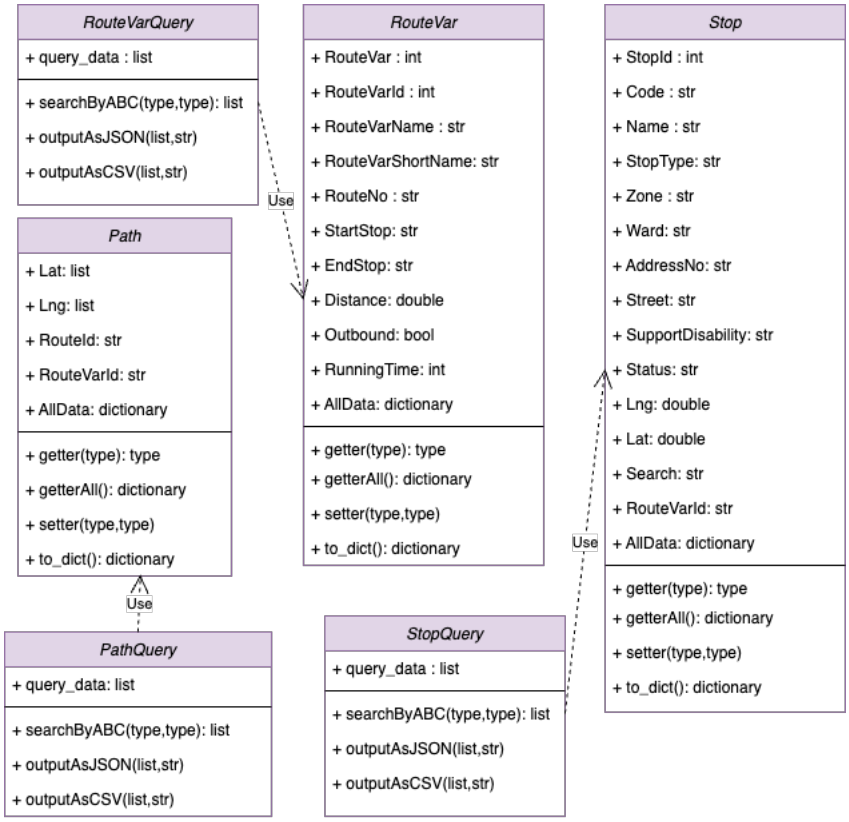
- OpenAI API: describe functions and have the model intelligently choose to output a JSON object containing arguments to call one or many functions
- Mistral AI: By integrating Mistral models with external tools such as user defined functions or APIs, users can easily build applications catering to specific use cases and practical problems.

6 Week 7 - Building graph

6.1 Data preparation

6.1.1 Class Diagram

Since there is no class newly created, the in-use classes are the same as Week 5 and Week 6



6.1.2 Routes data

Firstly, I created a dictionary to **store distance and time for calculating average speed**.

```
Route_pos = {}

with open("vars.json","r",encoding='utf-8') as fileroute:
    for eachline in fileroute:
        content = json.loads(eachline)
        for eachdict in content:
            p = RouteVar(eachdict)
            Route_pos.update({combine_var_id(p.RouteId,p.RouteVarId):{'Distance'
   :p.Distance,'Time':p.RunningTime}})
```

After that, I receive a dictionary `Route_pos` in the memory with the format

`Route_pos ={"RouteId,RouteVarId": {"Distance":float, "Time":float} }`

6.1.3 Stops data

Subsequently, I created another dictionary to **store the position of each stop for usage of arrays and the information of each stop for further query**.

```
Stop_pos = {}
cntStop = 0
track_stop_from_pos = [] #For track StopId from its position in array
with open("stops.json") as filestops:
    for eachline in filestops:
        data = json.loads(eachline)
        datalist = data['Stops']
        for eachdata in data['Stops']:
            mydict = eachdata
            s = Stop(mydict)
            if Stop_pos.get(str(s.StopId)) is None:
                Stop_pos.update({str(s.StopId):{'Pos':cntStop,'Stop':mydict}})
                track_stop_from_pos.append(s.StopId)
                cntStop = cntStop + 1
```

After that, I receive a dictionary `Stop_pos` in the memory with the format

`Stop_pos ={"StopId": {"Pos":int, "Stop":dict} }`

6.1.4 Paths data

Finally, I create a dictionary to **store the path of each route for creating Adjacency list**

```
Path_pos = {}
with open("paths.json") as filepaths:
    for eachline in filepaths:
        data = json.loads(eachline)
        mypath = Path(data)
        Path_pos.update({combine_var_id(mypath.RouteId,mypath.RouteVarId):{"Lat"
   :mypath.lat,"Lng":mypath.lng}})
```

After that, I receive a dictionary `Path_pos` in the memory with the format

`Path_pos ={"RouteId,RouteVarId": {"Lat":[], "Lng":[]} }`

6.2 Build Adjacency List

To calculate distance and time between two stops, I decided to perform it on `stops.json`. Each route in `stops.json`, I travel to each element in `"Lat"` and `"Lng"` list taken from the dictionary `Path_pos`, find the nearest coordinates from the stop and take it as where the bus stops (because as observation, not all the stops have their nearest points with the distance less than 5 meters)

In order to calculate the distance, I use prefix sum for easier implement (but costs memory)

```
distance_path = [0] * 2000
for x in range(1,len_latlng): #PrefixSum
    x1,y1 = latlng_to_xy(tmp_lat_list[x],tmp_lng_list[x])
    x2,y2 = latlng_to_xy(tmp_lat_list[x-1],tmp_lng_list[x-1])
    distance_path[x] = distance_path[x-1] + calc_distance(x1,y1,x2,y2)
```

The main structure of my code is shown below

```
visit each stop of the route:
    for list of lat,lng
        chosen point = nearest coordinates from stop
    time = distance from previous stop to chosen point / average speed (
   calculate from Route_pos['RouteId,RouteVarId']['Distance','Time']
    if not the first stop of the route
        if the stop appears in adjacency list:
            update the smaller value
        else update its value
```

After that, I receive a adjacency dictionary `adlist` in the memory with the format

`adlist ={"Stop_pos": {"Near_stop" : tuple} }`

The tuple stores **time and distance from stop to stop, RouteId which paths taken, position of the path beginning, position of the path ending**.

```
adlist[str(prev_pos_stop_ord)].update({str(pos_stop_ord):(time,
   distance_from_prev,combine_var_id(data['RouteId'],data['RouteVarId']),
   prev_stop_from_path_ID,chosen_point)})
```

7 Week 7 - Dijkstra performance

7.1 heapq structure

Heap data structure is mainly used to represent a priority queue. The heap[0] element returns the smallest element each time.
There are numbers of function used with heapq, but in Dijkstra Algorithm, I use only functions that perform appending and popping items.

- `heappop(heap)`: This function is used to remove and return the smallest element from the heap. The order is adjusted, so that heap structure is maintained.
- `heappush(heap, element)`: This function is used to insert the element mentioned in its arguments into a heap. The order is adjusted, so that heap structure is maintained.

```
pq = [(0,source_node)] #Create the heap queue
current_time, current_node = heapq.heappop(pq) #Use of heappop
heapq.heappush(pq,(dp[source_node][int(node)],int(node))) #Use of heappush
```

7.2 Dijkstra implement

Dijkstra is a common algorithms for finding shortest path from a source to all nodes without negative path. Its idea is generalizing breadth-first search to weighted graphs.

```
Dijkstra(source_node)
    min_weight of each node = infinity
    set min_weight[source_node] to 0 (from source_node to source_node)
    push the (weight,source_node) to the queue
    while (queue is not empty)
       pop the first items out of queue
       compare the weight to minweight, if weight is bigger -> skip
       for (all near node of the considering node)
           if its weight smaller than min_weight
               push the (weight, node) to the queue
```

Translate the idea to Python language

```
for i in range(0, cntStop):
    dp[source_node].append(1000000000000)
    trace[source_node].append(0)

dp[source_node][source_node] = 0

pq = [(0,source_node)] #Queue

while pq:
    current_time, current_node = heapq.heappop(pq)
    if current_time > dp[source_node][int(current_node)]:
            continue
    for node, value in adlist[str(current_node)].items():
        cost, distance,name,startid,endid = value
        if dp[source_node][node] > dp[source_node][int(current_node)] + cost:
            trace[source_node][int(node)] = (current_node,name,startid,endid)
            dp[source_node][node] = dp[source_node][int(current_node)] + cost
            heapq.heappush(pq,(dp[source_node][node],node))
```

7.3 All pairs shortest

Perform Dijkstra on all pairs

```
for i in range(0,cntStop):
    dijkstra(i)
```

Output the answer to `dijkstra.json`

```
with open ("dijkstra.json", "w", encoding = 'utf8') as jsonfile:
    for i in range(0, cntStop):
        for j in range(0, cntStop):
            data = {"From_Stop_Id" : track_stop_from_pos[i], "To Stop Id" :
   track_stop_from_pos[j], "Total time" : dp[i][j]}
            json.dump(data, jsonfile, ensure_ascii=False)
            jsonfile.write('\n')
```

To calculate time consuming of all pairs shortest, I use `datetime` and `timeit` library

```
from timeit import default_timer as timer
from datetime import timedelta
```

```
start = timer()
#Code
end = timer()
print(timedelta(seconds=end-start))
```

Time to perform Dijkstra on all pairs is approximately 30 seconds

```
hiukao@192 23125034_Week07_4 % /usr/local/bin/python3 /Users/hiukao/Documents/PrjCS162/23125034_Week07_4/main.py
0:00:27.540969
```

Time to write answers to json file is approximately 5 minutes 30 seconds

```
hiukao@192 23125034_Week07_4 % /usr/local/bin/python3 /Users/hiukao/Documents/PrjCS162/23125034_Week07_4/main.py
0:05:35.490765
```

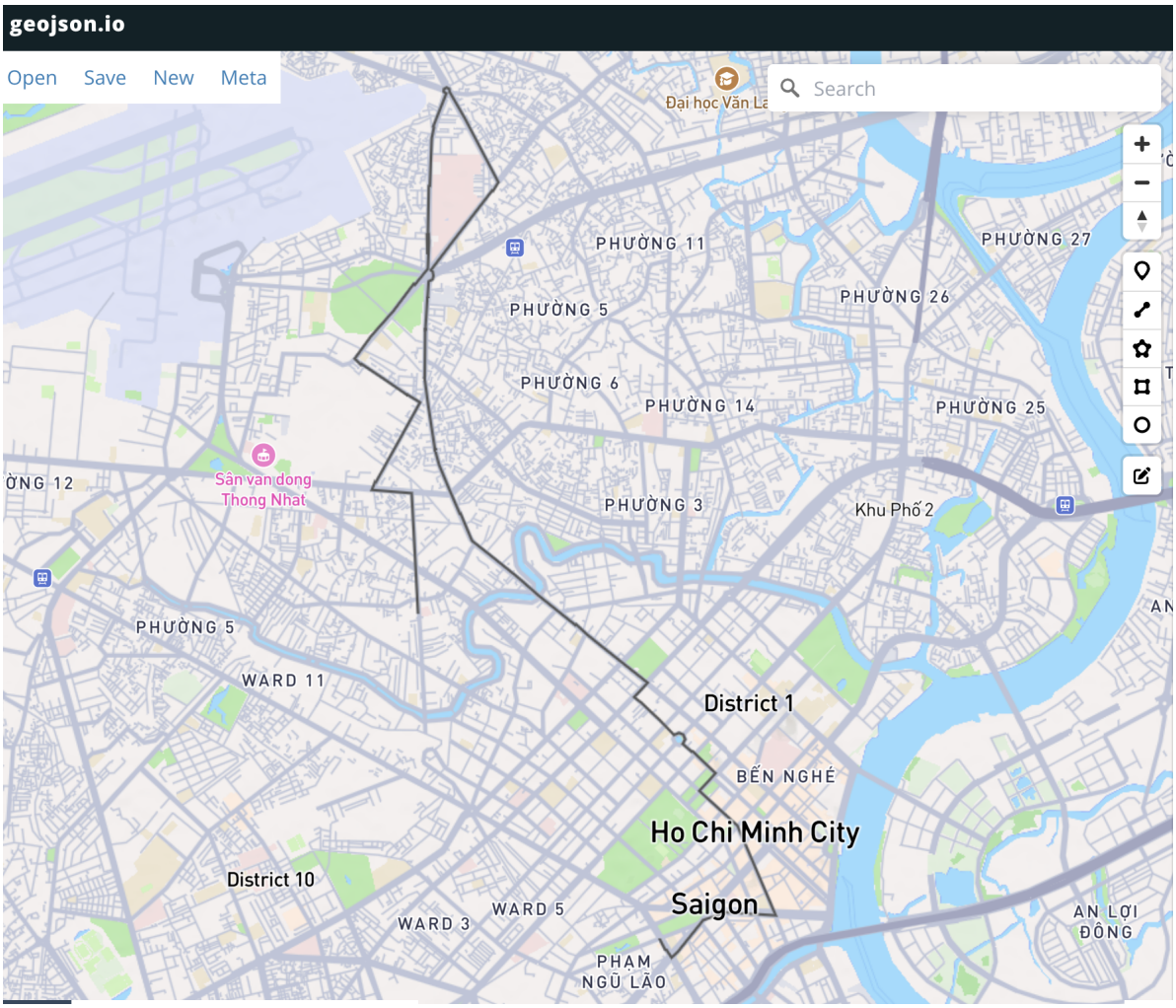
I store my `dijkstra.json` file via Google Drive

8 Week 7 - (start_stop, end_stop) Query

As being stored in the adjacency list, I can easily pull out data contains route and path between 2 nodes. Besides, the `trace` list can also be used to track stops.

```
def QuerytoFile(start_stop,end_stop,filename): #input StopId
    start_stop = Stop_pos[str(start_stop)]['Pos']
    end_stop = Stop_pos[str(end_stop)]['Pos']
    dict = {}
    dict.update({"Time": dp[start_stop][end_stop]})
    ans_Stop = []
    ans_Lat = []
    ans_Lng = []
    tmp_name = ""
    tmp_startidx = tmp_endidx = 0
    pos = end_stop
    while 1:
        if pos == start_stop:
            break
        ans_Stop.append(track_stop_from_pos[pos])
        tmp_pos, tmp_name, tmp_startidx, tmp_endidx = trace[start_stop][pos]
        for x in range(tmp_endidx+1,tmp_startidx+1,-1):
            ans_Lat.append(Path_pos[tmp_name]['Lat'][x])
            ans_Lng.append(Path_pos[tmp_name]['Lng'][x])
        pos = tmp_pos
    ans_Stop.append(track_stop_from_pos[start_stop])
    ans_Stop.reverse()
    ans_Lat.reverse()
    ans_Lng.reverse()
    dict.update({"Stop":ans_Stop,"lat":ans_Lat,"lng":ans_Lng})
    with open(filename, 'w', encoding = 'utf8') as fileJSON:
        fileJSON.write(json.dumps(dict))
```

For example, I perform the function with StopId 35 and 552 from `QuerytoFile(35,552,"ans.json")`, the result turns out like the following figure.



9 Week 7 - K most important Stops

Firstly, I create a list to count the importance of each stop with `trace` list.

```
importance = [0] * 5000
for u in range(0, cntStop):
    for v in range(0, cntStop):
        if dp[u][v] == 1000000000000:
            continue
        pos = v
        while 1:
            importance[pos] = importance[pos] + 1
            if pos == u:
                break
            pos, tmp_name, tmp_startidx, tmp_endidx = trace[u][pos]
```

Then, I use heapq library to find the top K stops, push negative value of importance to query from largest to smallest, store the result in `K_stop.json`

```
def k_importance(k, filename):
    pq = []
    for i in range(0, cntStop):
        vt = track_stop_from_pos[i]
        heapq.heappush(pq, (-importance[i], vt))
    with open(filename, 'w', encoding = 'utf8') as jsonfile:
        for i in range(0, k):
            u, v = heapq.heappop(pq)
            ans = {"Importance no." : i + 1, "Stop": Stop_pos[str(v)]['Stop']}
            json.dump(ans, jsonfile, ensure_ascii=False)
            jsonfile.write('\n')
```

10 Week 10 - Dijkstra's Improvement

I used the `timeit` library to calculate time between lines of code.

```
All pairs:  29.371463003801182
Shortest time with stopId  7511  for  0.00021058297716081142
Longest time with stopId  874  for  0.056441582972183824
Time to find the most important_stop (K=1):  243.3992402079748
```

11 Week 10 - Shortest Path Improvement

11.1 Floyd-Warshall

After trying to use Floyd-Warshall algorithm instead of Dijkstra, I observed that, Floyd-Warshall is easier to implement (3 for-loop) and can be used for all pairs shortest path (APSP) problems; however, because of using 3 loops, Floyd-Warshall is only useful for problems with few hundreds nodes. Therefore, applying in this project, which has 4397 stops, Floyd-Warshall is not effective.

```
Floyd-Warshall timing:  12547.344298750046
```

Time querying Floyd-Warshall APSP

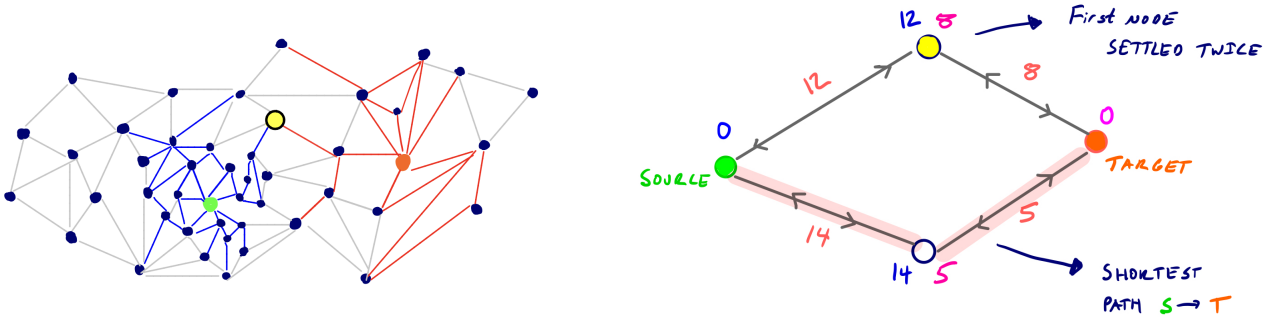
```
#BUILD ADJACENCY ARRAY
#[...] find time between 2 stops like Dijkstra
if stop_ord>0:
    pos_stop_ord = Stop_pos[str(curr_StopId)]['Pos']
    prev_pos_stop_ord = Stop_pos[str(prev_StopId)]['Pos']
    floyd_weight[prev_pos_stop_ord][pos_stop_ord] = min(floyd_weight[
   prev_pos_stop_ord][pos_stop_ord],timing)
#[...]

def floyd_warshall():
    for k in range(0,cntStop):
        for i in range(0,cntStop):
            for j in range(0,cntStop):
                floyd_weight[i][j] = min(floyd_weight[i][j],floyd_weight[i][k] +
    floyd_weight[k][j])
```

11.2 Contraction Hierarchy

11.2.1 Bidirectional Dijkstra APSP

Bidirectional Search can be use in the improvement of APSP problems. Rather than only run a Dijkstra query that settles nodes outward from the source, we simultaneously run Dijkstra's algorithm forward from the source and backward from the target.

To run two "simultaneous" rounds of Dijkstra's algorithm, we maintain two priority queues and at each iteration either settle a node in the forward search, or settle a node in the backward search.

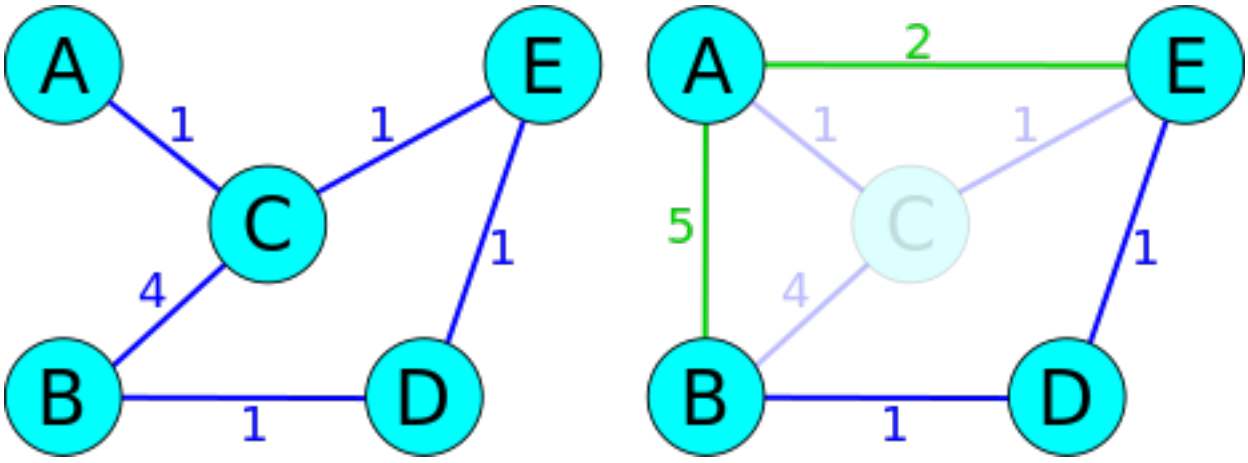


Visualizing Bidirectional Search, courtesy of John Lazarsfeld on Github

In the worst cases, time complexity remains basically the same as normal Dijkstra; however, in practical cases like finding a route on a road network, bidirectional approach is similar to growing a disc around each end and stop (nearly) as soon as both discs meet, while a single-direction approach would require to grow a disc from the start until you reach the end. Moreover, if R is the straight distance between start and end, the cost of a normal straight search would be $O(R^2)$, while the bidirectional approach would be in $O(2(\frac{R}{2})^2)$, i.e. 2 times faster.

11.2.2 Contraction Hierarchy algorithms

The most general idea of Contraction Hierarchy is **node contraction**. The condition to reduce a node is if the contracted node existed on the shortest path between two of its neighbors before contraction, we add a shortcut edge between the two neighbors such that all shortest path lengths are still preserved.



Before and after a node is contracted, courtesy of Mjt

In conclusion, Contraction Hierarchy includes 3 main stages:

- **Node contraction**: The query is correct no matter the order in which nodes were contracted, but our nodes ordered from least costly to most costly would be more effective.
- **Shortcut Creation**: These shortcuts represent direct connections between nodes that were previously indirectly connected through the contracted nodes. Only add a shortcut edge `uw` if the path `uvw` is the shortest path from `u` to `w`.
- **Querying**: Queries for shortest paths between pairs of nodes can be efficiently answered using bidirectional search or other techniques.

Contraction Hierarchy is used in various scenarios where finding shortest paths in large-scale graphs is a common task. For example, it can be used in Transportation Networks, Logistics and Supply Chain Management, Geographical Information Systems (GIS),i.e.

CH algorithms takes the idea of zooming out a map, where all small streets are eliminated

12 Week 11 - Further research on LLMs

12.1 Gorilla's OpenFunctions: Overview

Gorilla is a large language model developed by Microsoft and University of California, Berkeley. Their OpenFunctions is proposed to be developed similar to function calling in OpenAI's GPT-4. Besides, instructions from the developers is pretty straightforward, which is easy to understand. OpenFunctions v2 supports multi-coding-language with numbers of data types.

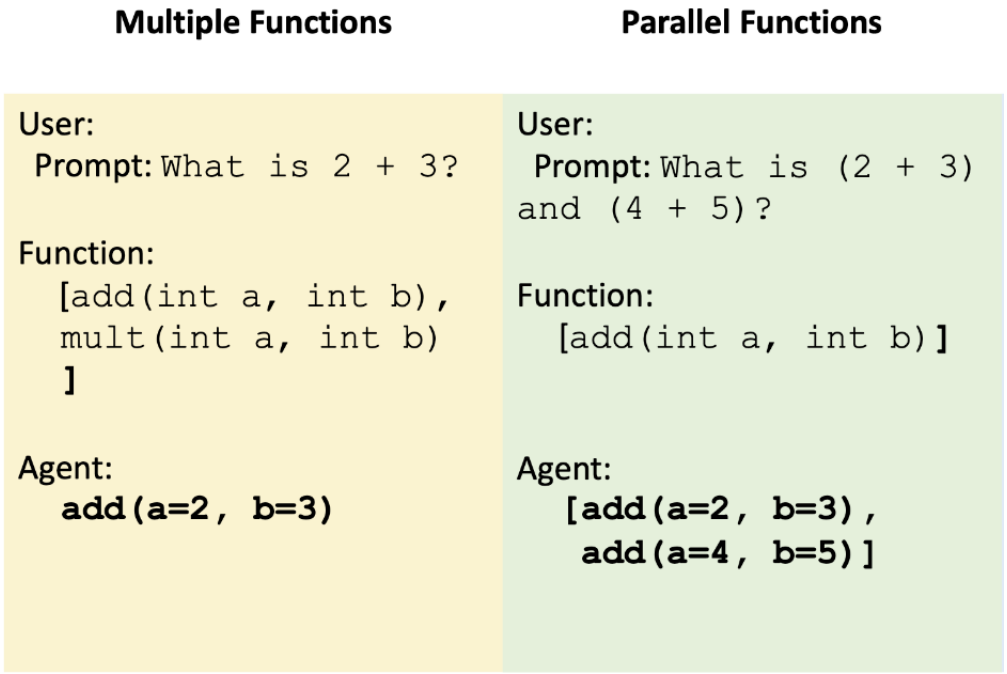
- **Python** - supports `string, number, boolean, list, tuple, dict` parameter datatypes and `Any` for those not natively supported.
- **JAVA** - support for `byte, short, int, float, double, long, boolean, char, Array, ArrayList, Set, HashMap, Hashtable, Queue, Stack`, and `Any` datatypes.
- **JavaScript** - support for `String, Number, Bigint, Boolean, dict (object), Array, Date`, and `Any` datatypes.
- **REST** - native REST support

OpenFunctions can be used in several cases

- **Chatbots with API calls**: Create chatbots that answer questions by calling external APIs (e.g. like ChatGPT Plugins)
- **Natural Language to API Conversion**: Convert natural language into API calls
- **Data Extraction**: Extract structured data from text

According to the developers, OpenFunctions version 2 provides Multiple Functions and Parallel Functions for users.

- **Multiple Functions**: Users can input multiples functions when they are not sure which exact function is best to service the prompt. In this scenario, the Gorilla model picks one or more (or none) of the functions provided to respond to the user's requests.
- **Parallel Functions**: The user's prompt could be serviced by multiple calls to the same function.



Comparison between two types of function calling

12.2 Function calling from natural language - English

12.2.1 Small update

For the purpose of performing function calling for Week 11, I reduce the time finding adjacency list for Dijkstra by write the whole dictionary in a json file called `edge.json`, so that the next time I run the program, I only have to read the file instead of step through every single stop.

```
hiukao@192 23125034_Week10_2 % /usr/local/bin/python3
Time creating adlist 3.0232380838133395
```

Former codes to create adjacency list

```
hiukao@192 23125034_Week07_4 % /usr/local/bin/python3
Time reading adlist:  0.008746417006477714
```

Update leads to run 3 seconds faster

```
#READ FILE EDGE.JSON
with open("edge.json") as filejson1:
    adlist = json.load(filejson1)
```

```json
{
    "0": {
        "1": [
            29.02373008237039,
            218.76291028990462,
            "115,231",
            0,
            2
        ],
        "1102": [
            436.7811265719646,
            1565.1323702162067,
            "198,1",
            0,
            6
        ],
        "1411": [
            307.23186261046857,
            2343.155005509173,
            "24,47",
            0,
            14
        ],
        "2368": [
            110.13405606324362,
            453.808789983673,
```


Some first lines of the attached `edge.json`

12.2.2 Applying OpenFunctions into this project

Following the instructions, first of all, we must install openai because Gorilla's OpenFunctions is compatible with OpenAI Functions.

```
pip install openai==0.28.1
```

After that, invoking the hosted Gorilla Openfunctions-v2 model is necessary. I use the following given code provided from the developers.

```
def get_gorilla_response(prompt="", model="gorilla-openfunctions-v2", functions
    =[]):
  openai.api_key = "EMPTY"
  openai.api_base = "http://luigi.millennium.berkeley.edu:8000/v1"
  try:
    completion = openai.ChatCompletion.create(
      model="gorilla-openfunctions-v2",
```

```
      temperature=0.0,
      messages=[{"role": "user", "content": prompt}],
      functions=functions,
    )
    return completion.choices[0]
  except:
    print("ERROR OCCURRED.")
```

Consequently, a set of functions needs to be defined in the type of JSON dictionaries. Each function should encompass fields: name, description, and parameters. All the functions of mine are stored in `function_doc.json`, which is attached along in the submitted folder.

```
{"name":"fastest_route","description":"Find fastest route from start stop to end
    stop","parameters":{"type":"object","properties":{"start_stop":{"type":"int"
   ,"description":"Initial bus stop"},"end_stop":{"type":"int","description":"
   Destination bus stop"},"required":["start_stop","end_stop"]}}}
```

Finally, put the query in a string variable and get the model response via the syntax

```
get_gorilla_response(prompt=query, functions=[function_documentation]))
```


```
{
  "index": 0,
  "message": {
    "role": "assistant",
    "content": "fastest_route(start_stop=3289, end_stop=3934), fastest_route(start_stop=35, end_stop=481)",
    "function_call": [
      {
        "name": "fastest_route",
        "arguments": {
          "start_stop": 3289,
          "end_stop": 3934
        }
      },
      {
        "name": "fastest_route",
        "arguments": {
          "start_stop": 35,
          "end_stop": 481
        }
      }
    ]
  },
  "finish_reason": "stop"
}
```


Result response from OpenFunctions v2, parallel functions

To call the functions, use `exec()` to execute OpenFunctions answer.

```
exec(get_gorilla_response(prompt=query, functions=[function_documentation]).
   message.content)
```