Independent Study

Desheng Zheng

Zhihan Fang

Diana Kris Navarro

Olaolu Emmanuel

Mapping Cellphone Tower Usage in Shenzhen

This past semester Biggie and I worked on the problem of visualizing data for different regions and districts of Shenzhen, China. Given to us was the data to form the polygons for the different regions and districts. We would also be given the data of cell tower coordinates and data associated with each cell tower. Our task was to aggregate the coordinates of each cell tower for each region in Shenzhen and visualize the data given to us.

Our original approach was:

1. Run through the cell-tower coordinates
2. For each cell-tower coordinate
    a. Run pnpoly() to figure out which region in Shenzhen the cell-tower data point was in
    b. Append the cell-tower data to the region's data

Although this original approach would have worked, this approach was not the best we could do. Running this algorithm (given n cellular tower coordinates and m regions) we would be running a worst case time complexity of O(nm).

Our new approach:

We decided that instead of running pnpoly for every cellular tower coordinate, we would pre-process the region/district polygons into a 2D matrix. So instead of every cellular coordinate

being checked against every polygon/region, the cellular coordinate would be mapped to a box in the 2D matrix that represented the map. Every box in the 2D matrix would hold a list of the index of the regions that overlapped the box.

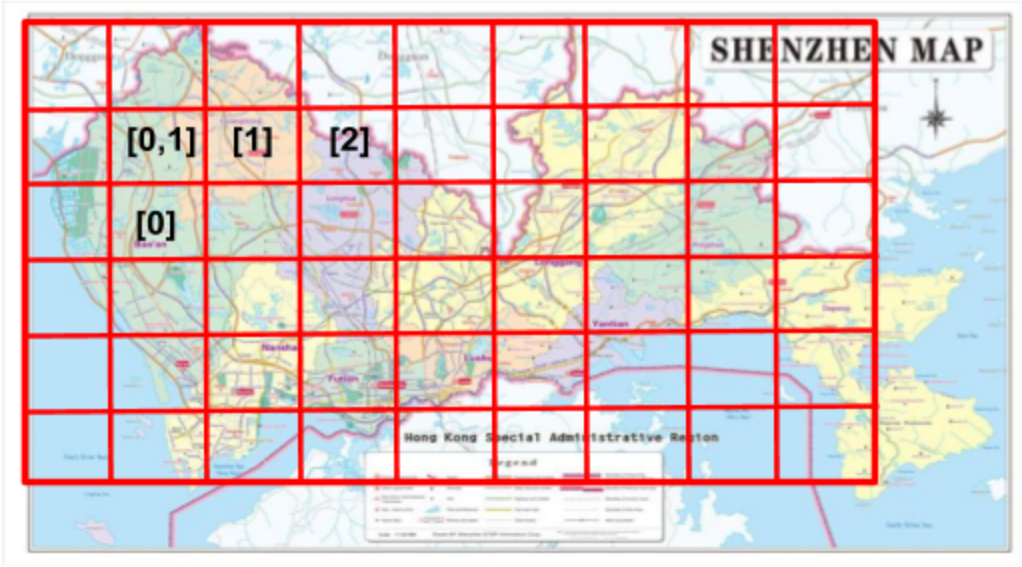
In the example below, in the 2D Matrix box [2,2] is an array of the indices [0,1]. This is referring to the index of the green region and the orange region in the region array. This way is more efficient because when we ran through the cell-tower coordinates, we would greatly reduce the regions to check if the point was in. We would reduce running the pnpoly algorithm from checking all the regions for each coordinate, to 1 or 2 regions.

EXAMPLE

Region array is:

```
[green_region, orange_region, purple_region, yell_region]
```

2D Matrix (in red):



The Pre-Processing:

To first initialize the 2D grid:

1. Run through every polygon

2. For every polygon, run through it's latitude and longitude coordinates
3. For every coordinate:
    a. i value in the 2D matrix = [(maxX - minX) - (latidude - minX)] / xStep
    b. j value in 2D matrix = (longitude-minY) / yStep
    c. append the region index to the list inside 2Dmatrix[i][j]
4. Run through the empty boxes inside the 2D matrix, picking a random latitude, longitude point in that box
5. Run that random latitude longitude point through pnpoly
6. Appending any regions associated with that point to the list inside the 2D matrix

Example: Given 3 random polygons, the 10 by 10 2D matrix would look like this:

NOTE: The latitude, longitude point [0,0] would correspond to the 2D matrix [i,j] points in this matrix as [10,0] because longitudinal points ascend going up, whilst in a 2D matrix the i value ascends going down.

```
{
    "type": "FeatureCollection",
    "features": [
        {
            "type": "Feature",
            "geometry": {
                "type": "Polygon",
                "coordinates":
                       [[  [10,10],
                           [10, 20],
                           [0, 10],
                           [0, 20]] ]
            }
        },
        {
            "type": "Feature",
            "geometry": {
                "type": "Polygon",
                "coordinates":
                       [[  [7,0],
                           [0, 0],
                           [7, 20],
                           [0, 20]] ]
            }
        },
        {
           "type": "Feature",
            "geometry": {
                "type": "Polygon",
                "coordinates":
                       [[  [0, 0],
                           [0, 2],
                           [15, 0],
                           [20, 0],
                           [15, 2],
                           [20, 2]] ]
            }
        }
    ]
}
```

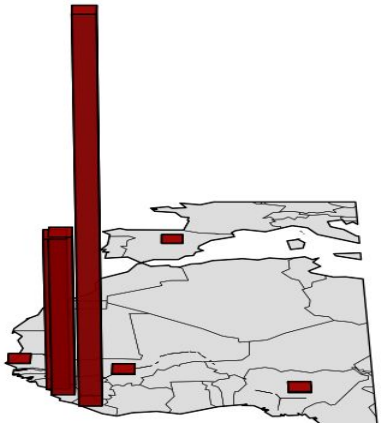
```
⚡ python pnpoly.py
(0, [[2], [2], [], [], [], [], [], [], [], [], []])

(1, [[2], [2], [], [], [], [], [], [], [], [], []])

(2, [[2], [2], [], [], [], [], [], [], [], [], []])

(3, [[2], [2], [], [], [], [], [], [], [], [], []])

(4, [[2], [2], [], [], [], [], [], [], [], [], []])

(5, [[2], [2], [], [], [], [0], [0], [0], [0], [0], [0]])

(6, [[1], [2], [], [], [], [0], [0], [0], [0], [0], [1]])

(7, [[1, 2], [1, 2], [1], [1], [1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1]])

(8, [[1, 2], [1, 2], [1], [1], [1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1]])

(9, [[1, 2], [1, 2], [1], [1], [1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1]])

(10, [[1, 2], [2], [1], [1], [1], [0], [0, 1], [0, 1], [0, 1], [0, 1], [0, 1]])
```

Choosing a Visualization

When deciding how to visualize our data, we opted to go with a 3D bar plot where the height of each bar would reflect the usage of each cellphone tower. We chose to do so for a few reasons:

1. Bars seemed visually appropriate as they could represent the cellphone towers themselves.
2. From a glance, the user would be able to identify hot spots by looking for the highest bars. Additionally, by placing the bars in the middle of each region, the user could also identify which regions in particular received the most activity and this data seemed like it may be very useful for drawing insightful conclusions.
3. We also felt a 3D bar plot would be novel because after discussing with Zhihan, we found that no maps like this had previously been made. In the spirit of research we felt that the type of visualization itself could give us important information such as whether or not a plot like this would be viable for the given dataset.

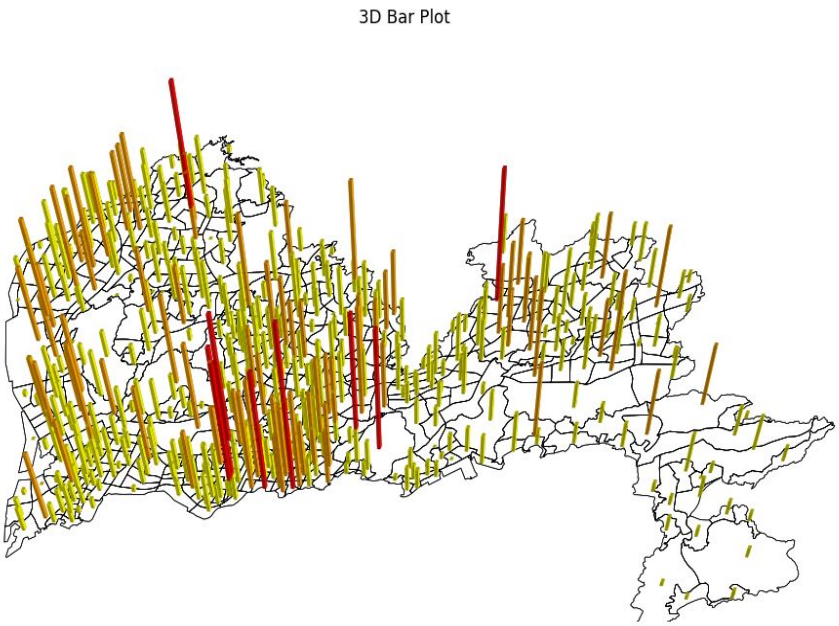
We ended up with a plot that looked similar to this:



This was a decent approach, but as the amount of bars increased, the graph soon became very cluttered and difficult to comprehend. We addressed this issue by introducing a color coding to the bars. Before applying the colors to the bars, we first normalized our data by feature scaling it, such that all bar values would lie between 0 and 1. Now, we could apply colors to certain ranges of values. In the end, we chose to do the following:

YELLOW	0 - 33%
ORANGE	34% - 66%
RED	67% - 1

This felt appropriate because the highest values would have the most intense color, thus making it very easy to spot high activity. In addition, the initial glance at the map would now be more insightful. In the end, we had a visualization that looked like the following:



Looking at this plot, we concluded that towers that were being used most frequently were located in the southern part of Shenzhen. We could also, see that usage seems to be quite sparse as you move more towards the East. This however, could be due to not having enough cell data.

Building the project

We spent most of our initial time, writing out on paper what wanted to accomplish before diving into the code. Once we had our plan we started building the map. Finding an appropriate library to create a 3D bar plot proved to be very difficult. There was not a lot of resources available to us and the few that we could find either did not provide an API or required us to pay money in order to use their platform. Eventually, we came across BaseMap3D, an extension of matplotlib which provides additional tooling for rendering 3D plots. The documentation was sparse however, and not many people have used it so we spent a lot of time reading.

Our next hurdle, was gathering all of our dependencies. BaseMap3D depended on a lot of other python modules and other libraries which took a while to find and install. This posed a problem because we wanted others to be able with run our code without having to spend a lot of time installing dependencies. For this reason, we provided a setup script in our project which gathers all of the necessary dependencies without requiring any input from the user. We also have a setup section in the README describing how to run said script.

With all of our dependencies in order, we now had to actually write the code. We knew the pnpoly algorithm would be complicated since we planned to do some pre-processing (described above) so we exported that logic to its own module. Once, we wrote and tested the code for it, our next step was to write the logic for rendering the plot. We didn't want the plot to only run on a single dataset so we provided command line arguments for users to supply their

own data and described how to use them in our README. We were sure to add comments to our code and use descriptive variable/function names so that future students could extend upon our work, if need be. We also, tried to have the code gracefully exit whenever possible and provide helpful hints on how to fix such errors. For example, we validate the command-line arguments and ensure all files exist before proceeding in our script. We alert the user of any files that don't exist so that they can fix their paths.

Links

1. github repository - where all our code/assets are located

Resources

1. BaseMap3D - creating the 3D bars
2. Matplotlib - the underlying library which render the visualization
3. Mapshaper.org - used to convert geojsons to shp files so that BaseMap could read them
4. Shapely - used for point/polygon geometric operations i.e. point in polygon

Improvements

Though we did always strive to choose the most efficient methods, there are some places we could improve, given more time.

1. Unfortunately, due to the sparse documentation and lack of use of the BaseMap3D library, we were not able to figure out how to add a legend to the map. This would help to explain what the colors are indicating. In the meantime, we added a description of the color coding to the README located on the github repo.
2. The point in polygon (pnpoly) algorithm is notorious for being a bottleneck in the visualizations of geospatial data. We knew this before starting and found many ways to speed up this calculation. We did some pre-processing of the data (described above) to significantly cut down on the total run time. However, another optimization that could have contributed to an even faster runtime would be to perform the pnpoly algorithm to a faster, lower-level, language such as Rust or C. While python's high level abstractions did help us to iterate quickly on our work, this comes at the cost of a higher runtime. Also, python's Global Interpreter Lock (GIL) prevent the concurrent execution of a CPU-intensive routines i.e. the pnpoly algorithm. Performing this in a lower level language would not only be faster, but would allow for true concurrency which would further cut our runtime. This guide from a book on Rust would be a good place to look into if this needs to be implemented in the future: link. Despite that, we are still pleased with the runtime we have now as it seems to fit our needs for the time being.