Nicholas Shvelidze and Nicholas Taweel
CMPSC 463
Project - 2: Blitzortung Lightning Clusters
Submitted to: Professor Yang

Project Objective

The objective of this project is to create a system that collects and decodes real-time lightning strike data from the Blitzortung network. Blitzortung does not provide a simple or officially supported API, so the system uses the raw data stream. It interprets the encoded messages, and converts them into structured geographic information. This is so that lightning strike data can be captured continuously and in a proper format.

The project also performs geographic computations like clustering, hotspot detection, and MST-based grouping to show patterns in lightning activity. To have real-world use and reliability, our project also includes process supervision, automated restarts, and frontend build integration, which are meant to maintain stability.

Project Significance

This project is significant because it creates an accessible way to work with real-time lightning strike data, which is usually difficult to obtain in a structured format. The Blitzortung network provides accurate crowdsourced lightning detection, but it does not offer a straightforward public API. This project makes lightning data available for further applications such as weather visualization, storm tracking, scientific research, and hazard monitoring.

The project is unique in that it combines data collection and geographic analysis into a single pipeline. The system can accomplish tasks like clustering, minimum spanning tree grouping, and hotspot detection to reveal patterns in lightning behavior that are not directly available from the raw stream. This is more efficient than just logging the data. The analytical end of our project sets it apart from basic scrapers and makes it useful for predictive systems.

Code Structure

Our project utilizes three different backend scripts. Two scripts, blitzortung_api.py and blitzortung_parser, are used to collect information from the Blitzortung network. The script "blitzortung_api.py" algorithmically processes the data derived from the network and applies Breadth-First Search, Greedy Hotspot Selection, and Prim's Minimum Spanning Tree. It also creates and analyzes clusters by finding local densities and calculating the supposed "centers" of particular clusters. The "blitzortung_parser.py" script is used to parse and clean data from the Blitzortung network, connected through a headless web browser. This script also allows the raw and processed data to be recorded to a JSON file. The main script of our program “app.py” has many functions to support the collection process and general system build. It serves as the backend API for the system and also helps operate the frontend. To check that the collection process is stable, it includes a “supervisor” function that makes sure that the collector subprocess is running. If it shuts down, the supervisor function will restart the page collection process. The “main()” function of this script connects the frontend and backend by making sure the frontend is built, starting the collector supervisor, and starting Flask.

Blitzortung Network

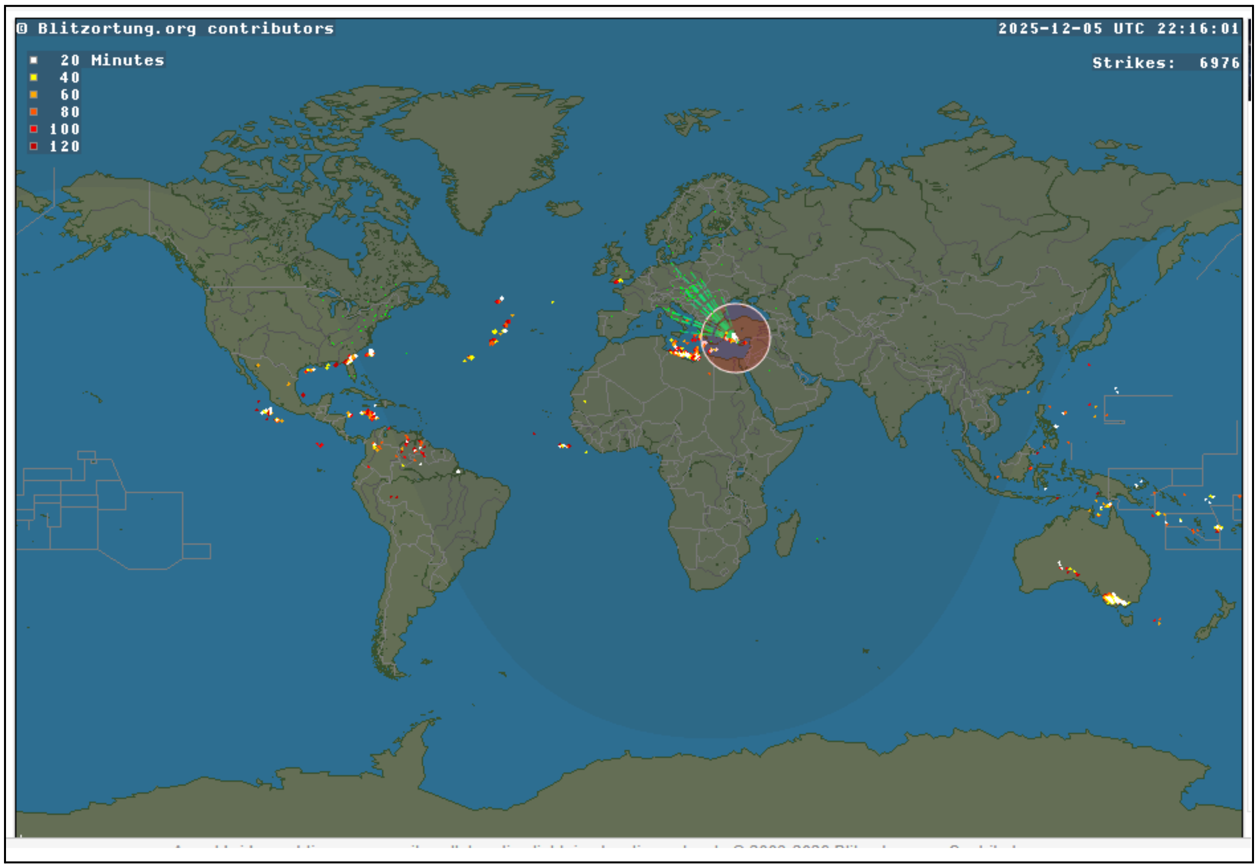
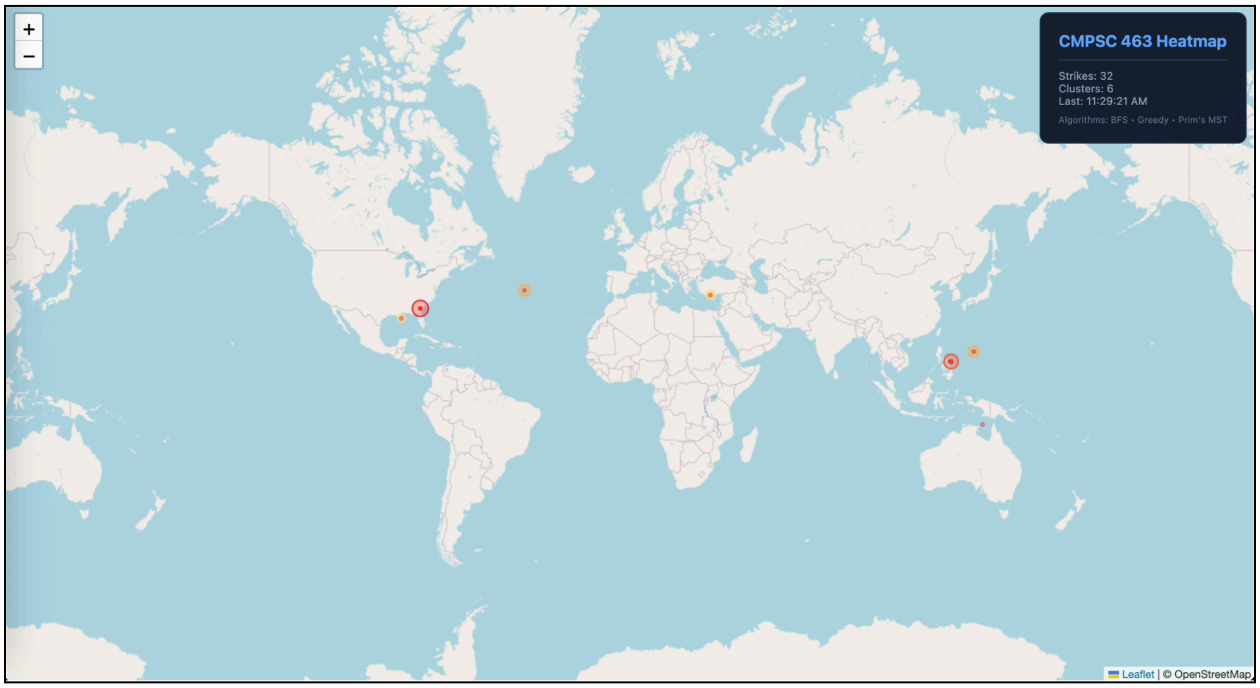


Figure 1 - Lightning strike visualization from Blitzortung.org

The Blitzortung network is a crowdsourced lightning detection system made of sensors that capture electromagnetic signals produced by lightning strikes. These sensors send timing and signal data to Blitzortung servers, which triangulate the strike locations. In our project, we use Blitzortung’s live data feed to collect lightning event messages through an automated browser connection. This data is then decoded, cleaned, and analyzed using clustering algorithms and graph-based methods. Blitzortung’s real-time global lightning observations help our system with identifying activity hotspots, forming clusters, and showing patterns in lightning behavior.

Description of Algorithms


CMPSC 463 Heatmap

Strikes: 32
Clusters: 6
Last: 11:29:21 AM

Algorithms: BFS • Greedy • Prim's MST

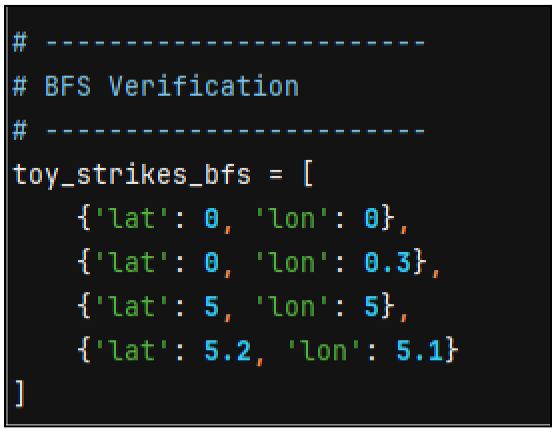
Figure 2 - CMPSC 463 Heatmap web application UI

This project uses several algorithms from CMPSC 463 to analyze lightning strike data. The first algorithm is Breadth-First Search (BFS), which is applied to find connected components of lightning strikes. Each strike is treated as a node in a graph, and edges are formed between strikes that occur within a certain geographic radius. BFS explores these nodes level by level to find clusters of nearby lightning activity. This results with a set of groups that represent storm regions where many strikes are happening in close proximity.

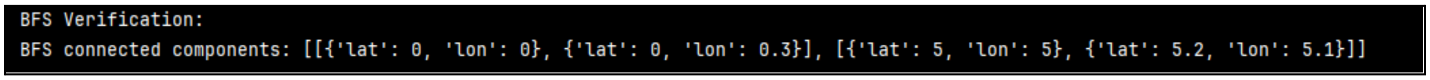
A second algorithm used is a Greedy Hotspot Selection method. In this method, each lightning strike is assigned a weight based on how many other strikes occur within a certain radius. After computing these densities, the algorithm sorts all strikes by weight and selects the top k as the primary hotspots. This greedy method is efficient and highlights regions with unusually high concentrations of lightning activity, making it useful for heatmaps and identifying storm centers.

The project also implements Prim's Minimum Spanning Tree (MST) algorithm for accurate clustering. Prim's algorithm constructs an MST across all lightning strikes using pairwise geographic distances as edge weights. Long edges, representing unrelated strikes, are removed, and the remaining connected components form clusters. Unlike BFS, which uses a fixed distance threshold, MST clustering adapts to the structure of the data and can find clusters of varying sizes.

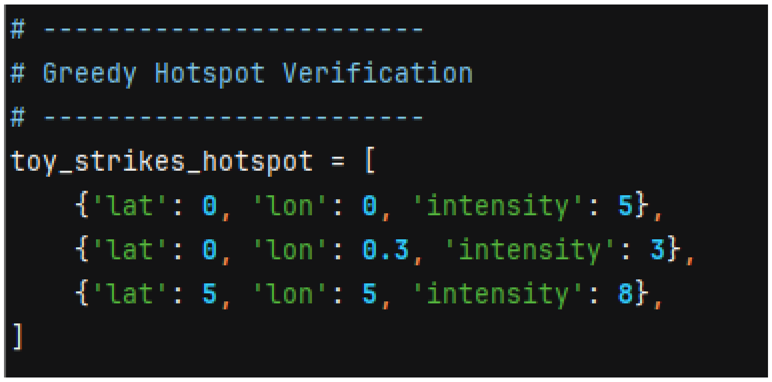
Algorithm Toy Examples



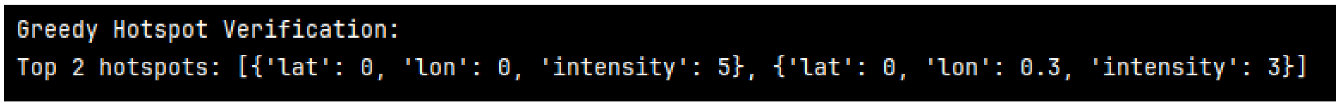
```
# ------------------------
# BFS Verification
# ------------------------
toy_strikes_bfs = [
    {'lat': 0, 'lon': 0},
    {'lat': 0, 'lon': 0.3},
    {'lat': 5, 'lon': 5},
    {'lat': 5.2, 'lon': 5.1}
]
```



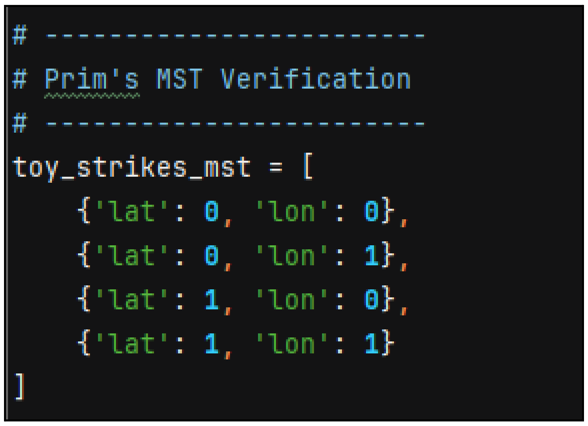
```
BFS Verification:
BFS connected components: [[{'lat': 0, 'lon': 0}, {'lat': 0, 'lon': 0.3}], [{'lat': 5, 'lon': 5}, {'lat': 5.2, 'lon': 5.1}]]
```



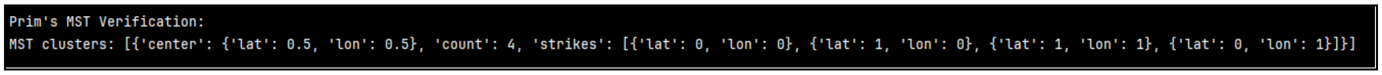
```
# ------------------------
# Greedy Hotspot Verification
# ------------------------
toy_strikes_hotspot = [
    {'lat': 0, 'lon': 0, 'intensity': 5},
    {'lat': 0, 'lon': 0.3, 'intensity': 3},
    {'lat': 5, 'lon': 5, 'intensity': 8},
]
```



```
Greedy Hotspot Verification:
Top 2 hotspots: [{'lat': 0, 'lon': 0, 'intensity': 5}, {'lat': 0, 'lon': 0.3, 'intensity': 3}]
```



```
# ------------------------
# Prim's MST Verification
# ------------------------
toy_strikes_mst = [
    {'lat': 0, 'lon': 0},
    {'lat': 0, 'lon': 1},
    {'lat': 1, 'lon': 0},
    {'lat': 1, 'lon': 1}
]
```



```
Prim's MST Verification:
MST clusters: [{'center': {'lat': 0.5, 'lon': 0.5}, 'count': 4, 'strikes': [{'lat': 0, 'lon': 0}, {'lat': 1, 'lon': 0}, {'lat': 1, 'lon': 1}, {'lat': 0, 'lon': 1}]}]
```

Figure 3 - Algorithm test examples for BFS, Greedy, and Prim's algorithms

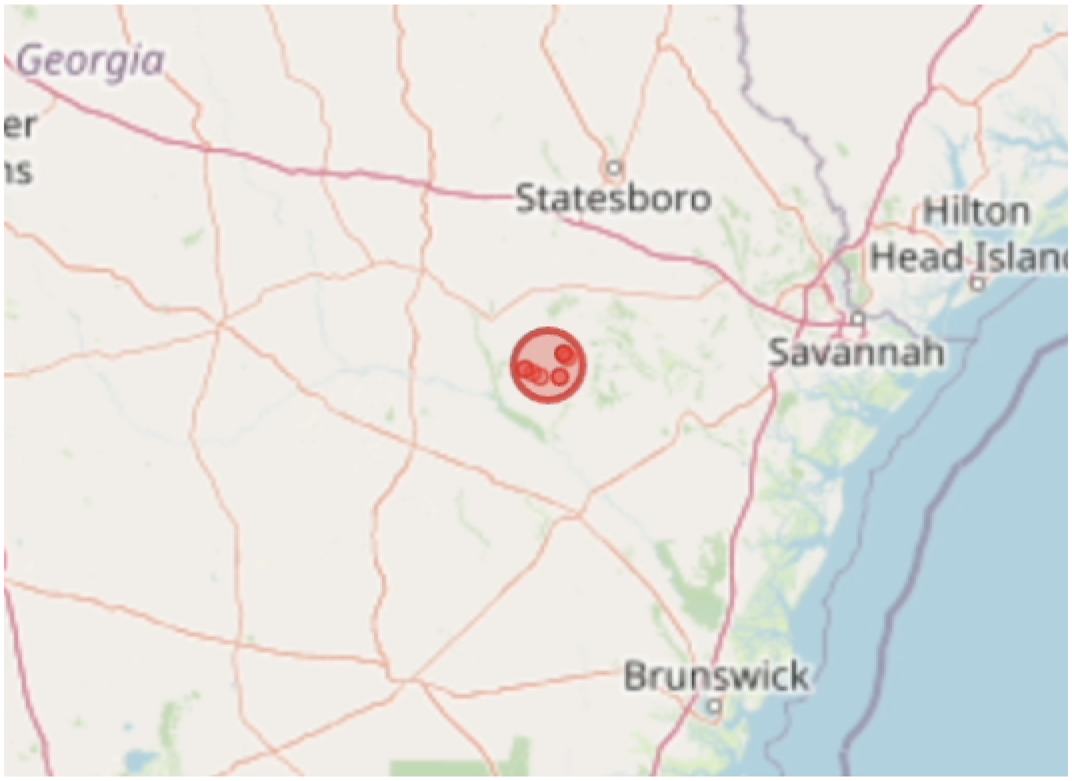


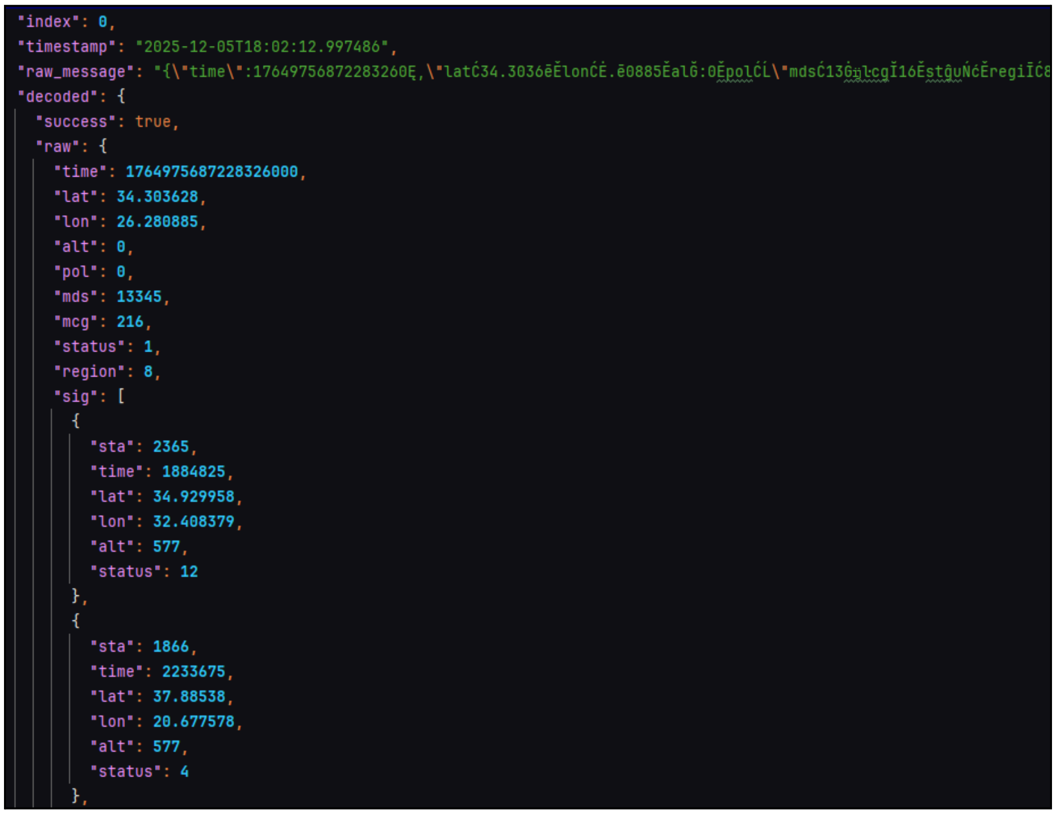
Figure 4 - Lightning strike cluster found by the algorithms

Functionalities

The program provides several functionalities that work together to collect and analyze the lightning data. First, it connects directly to Blitzortung's live WebSocket feed and captures incoming lightning strike messages. These messages are often compressed or encoded, so the program automatically handles decompression and uses a LZW-style decoder to translate the data into JSON. After decoding messages, the system cleans the output by converting binary values into geographic and numeric information, so that lightning strikes are shown with accurate information.

The system also saves both the raw and decoded messages into a structured JSON file, which lets the dataset be easily reused for analysis or visualization. The program also monitors the WebSocket connection in real time during collection, and reports the number of decoded messages and any failures. The program helps with geographic analysis by using clustering tools like BFS, hotspot detection, and MST-based grouping. The algorithms used were chosen to be efficient for the identification of storm regions and strike patterns. Playwright browser automation allows the entire collection process to run automatically. It opens the Blitzortung site, detects its WebSocket, and records data for a specified length.

JSON Data Storage



```
"index": 0,
"timestamp": "2025-12-05T18:02:12.997486",
"raw_message": "{\"time\":17649756872283260Ę,\"latĆ34.3036ēĚlonĆĖ.ē0885ĚalĞ:0ĚpolĆĹ\"mdsĆ13Ġŀlcgĩ16ĚstĝuŃćĔregiĨĆ8
"decoded": {
  "success": true,
  "raw": {
    "time": 1764975687228326000,
    "lat": 34.303628,
    "lon": 26.280885,
    "alt": 0,
    "pol": 0,
    "mds": 13345,
    "mcg": 216,
    "status": 1,
    "region": 8,
    "sig": [
      {
        "sta": 2365,
        "time": 1884825,
        "lat": 34.929958,
        "lon": 32.408379,
        "alt": 577,
        "status": 12
      },
      {
        "sta": 1866,
        "time": 2233675,
        "lat": 37.88538,
        "lon": 20.677578,
        "alt": 577,
        "status": 4
      },
```

Figure 5 - Lightning strike data parsed by Blitzortungs websockets and decoded with an LZW style string unobfuscator

The JSON file in our project serves as the structured output container for all decoded lightning data collected from the Blitzortung WebSocket stream. Each entry in the file represents an individual message that has been decoded, validated, and formatted so it can be easily analyzed or processed later. Storing the data in JSON makes it straightforward to load into Python scripts, visualization tools, or external applications, so that data access is consistent across the entire workflow.

Conclusion

In conclusion, this project successfully implemented a system to collect and decode real-time lightning data using web automation and data parsing techniques, and then make heatmap clusters based on the data provided. The findings show that the system is capable of retrieving live lightning messages and storing them in a structured format. However, some issues we encountered were with network timeouts and the asynchronous browser automation reliability. This project made use of important course concepts, especially in our methods that use graph searching algorithms, greedy algorithms, and spanning trees.