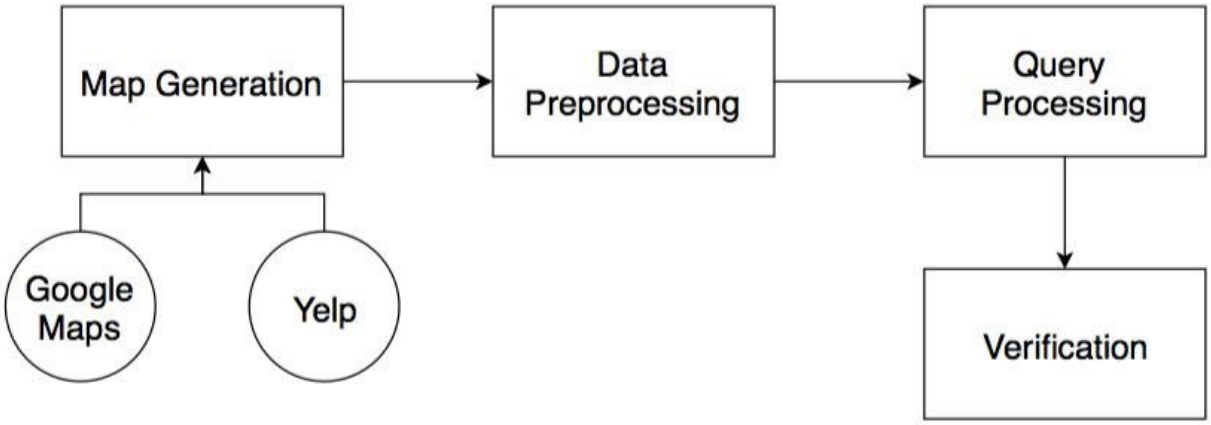


BACKGROUND & MOTIVE:

Recently, multiple Location-Based Service Providers (e.g. Yelp) have been outsourcing POI (points of interests) datasets to third-party cloud service providers (e.g. Amazon), which process queries on their behalf. These cloud service providers may return fake query results. In our project, we propose an algorithm to efficiently verify the result returned by a location-based service provider.

OUTLINE OF THE PROJECT

(1) **PREPARATION ALGORITHMS**

- Map Generation
- Shortest Path Algorithm (shortest path between any two points in any road segments)
- Data Crawl (you can rely on existed API eg. yelp fusion API)

(2). **DATA PREPROCESSING:** From Data owner to cloud servers, we need to compute extra information for verification. The algorithms are listed as follows

- 1d Skyline Algorithm (pre-compute all neighboring set and query ranges)
- Graph Partition and Subgraph-->Binary tree (one node in the tree represents one subgraph, the root node is the whole graph, leaf nodes include all road segments) I will send the pseudocode of graph partition later.
- Compute the local skyline union (I can describe with your guys later and it is easy. This is also extra information for verification)
- Dataset-->HashTree(verification) Actually, there are two hash trees. In one hash tree, the leaf node is a POI or restaurant. Another hash tree's leaf node represents a specific possible subgraph.

(3) **QUERY PROCESSING:** Cloud Server to users. Given any location of query from users, find the corresponding extra information and return to users.

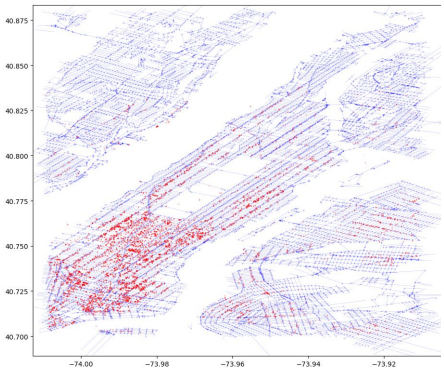
(4) **USER VERIFICATION**.

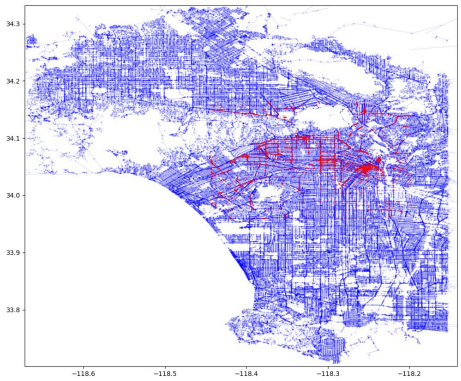
Dynamic dataset generation

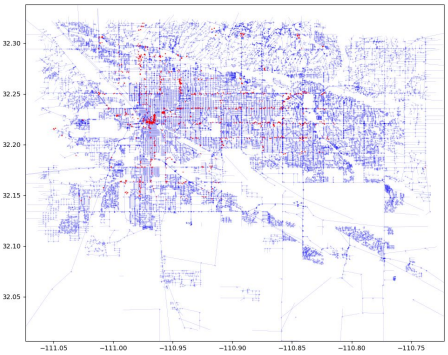
All python scripts related to road data generation are within the /map/ folder. These files Will either use a bounding box or location to specify what area to query. These specifications can be changed in **bBoxConfig.txt.** Any files generated by these scripts will be saved in /generated_map_data/ for CSV's or /map_data_images/ for images.

- **intersectionGen.py**
  - This file gets all intersections within a specified bounding box and writes them to a CSV file
    - CSV is of form: Name, Latitude, Longitude, ID
  - How it works
    - Using the OpenMaps API, we query for all "ways" within a bounding box. Each way has an array of Nodes that are used to define a road's form. Ways/Nodes also have additional data such as the Name and ID within the OpenMaps system.
    - After getting all ways, we extract all Nodes and Node ID's and store them into a dictionary {ID : Node} . Further, we extract store all node ID's inside an array. We use that array to find ID's that appear more than once. These ID's represent intersections, since two or more roads must share that node.
    - We can then use these ID's to get any Node information we want using the dictionary we created earlier.
- **roadSegmentGen.py**
  - This file gets all road segments within a specified bounding box and writes them to a CSV file
    - CSV is of form: startNodeID, endNodeID, startLat, startLong, endLat, endLong, length, name
  - How it works
    - Once again, we query for all ways within a bounding box, and store them in an array. Then, for each way, we compare every Node, searching for the two Nodes who have the maximum distance between each other. These Nodes will represent the start Node and end Node of that road segment.
    - Once we have the start and end Nodes for every way, we use them to write any relevant data to a CSV
- **poiGenYelp.py**
  - This file is an adaptation of the POI generation file in the /yelp/ folder. It is adapted to calculate which road segment a POI lies on, writing the startID and endID of its road segment to a CSV; in addition to other relevant information.
    - CSV is of form: address, latitude, longitude, rating, startNodeID, endNodeID, price

    - NOTE: This file requires a corresponding roadSegmentsXXX.csv file to work, so please generate one before attempting to use this script.
- **displayGeneratedDataset.py**
    - This file will take generated roadSegmentXXX.csv, intersectionsXXX.csv, poiYelpXXX.csv and create a plot of the data; resulting in images similar to these:







Graph Structure Generation

- **/map/graphgen.py**
    - This file converts the csv containing road segment data and the csv containing poi data and maps it to a networkx undirected graph, and then serializes this to a 'pickle' file called **graph_pickle**. This allows the structure to be saved as a file and read in later by another program through use of the python pickle library. Serializing the data in this way allows us to generate large datasets only once. We can also dynamically add or delete parts of the dataset later without needing to regenerate the entire dataset.
    - The graph generated stores the "weight" in the edges, which is equal to the distance in kilometers, and the nodes hold all other data (location, node names, type, etc.)
    - **The two arguments are hardcoded filenames:**
        - filein = "road_segment.csv" . This is a csv file representing all the road segments in the bounding box.
            - format: [node1,node2,lat1,long1,lat2,long2,weight]
                - The two nodes are unique node identifiers that correspond to node numbers in the other input file. The lat and long values correspond to each node's latitude and longitude. The weight is the linear distance of the edge measured in kilometers.
        - poifilein = "res_mapping_road_based.csv" . This is a csv file with data on all the points of interest in the bounding box
            - **The generated poi datasets from openmaps don't match this format exactly yet. Some more preprocessing needs to be done to reach this point.**
                - **In the /map/generated_map_data/ folder**:
                    - poiYelpManhattan.csv and roadSegmentsManhattan.csv are examples of correctly formatted files.
                    - In the map folder, there are two example files using the hardcoded csv names described which also match this format.
            - Format: [name,lat,long,maplat,maplong,rating,startnode,endnode]
                - Name of the point of interest (usually the address)
                - The lat and long are the actual latitude and longitude, which might not be along the road segment.
                - The maplat and maplong are the "mapping coordinates" which are where the pois are mapped to along the various road segments. This is used to extrapolate the distance between each node of the segment. There is some variability in this distance, especially with very long segments or when the road is aggressively curved. In most

situations, however (dense cities with straight roads), the accuracy is over 99.9%.
    - The rating is the 5 start floating point yelp rating, but could also be replaced by some other metric for the LBSQ.
    - The start node and end node are used to find the road segment to map the poi along. They need to be the exact names of a pair of nodes with a segment between them already in the graph.

HashTree Verification

- Hashtree.java: The hashtree class includes the following methods for data verification purposes:
    - Constructor: given a set of items (e.g., POI), construct a hash tree
    - FindAuthentication4Item: takes one item as input and returns the set of internal nodes that are needed to compute the root of the Merkle hash tree. Every item or internal node needs to contain the information about its position in the tree to allow computing the root when needed.
    - VerifyItem: takes one item and the set of internal nodes as input and returns true if they can compute the correct Merkle Hash tree root.
    - FindAuthentication4Set: takes a subset of items as input and returns the set of internal nodes that are needed to compute the root of the Merkle hash tree for every input item. This is a generalization of the FindAuthentication4Item method.
    - VerifySet: takes a subset of items and the set of internal nodes as input and returns true if we can compute the correct Merkle Hash tree root for EVERY input item. This is a generalization of the VerifyItem method.

Query Processing

- Skyline1d.py
    - Is not designed to be run as a standalone script, but instead imported to calculate 1D skyline queries and neighbor queries
    - Calculates 1D skyline query for a single road segment, returning a set of POIs via sky1D()
    - sky1D() takes as input road, representing an edge from graph_pickle, poi_set, representing the POIs on that road segment, and road_pos, representing the position of the ‘user’ on the road as a percentage from left to right
    - Skyline query functions by splitting the poi_set into a left and right set, where the divider is the location of the user. It then performs a standard skyline query on the left set and right set, where the only POIs left in result_set are those not dominated by any other POI, and returns the result set
    - sky1D() may need changing as the formatting of the graph and it’s POIs changes as well

        - Currently, the graph representing the road segments and the list of all POIs are separate entities. This makes using sky1D() difficult, as both road and poi_set must be constructed for each call to sky1D()
    - result_set_to_neighbors() takes in a result_set (usually from sky1D(), but technically could be any poi_set) and adds verification POIs to the result_set, setting the verification POIs to distance infinity away with price 0