CSCI 587 • Geospatial Information Management
Assignment 1

1 Overview of the Assignment

The goal of this assignment is to explore and compare different methods for executing spatial queries. You will implement three approaches: a brute-force **linear search**, a space-driven **grid index**, and a data-driven **KD-Tree index**. By evaluating their performance on *k-Nearest Neighbors* (KNN) and *range queries*, you will learn how different indexing techniques can impact query efficiency and understand the trade-offs between them in terms of speed and adaptability to different data distributions.

2 Programming Requirements and Environment Settings

This assignment is designed to be implemented in Python. If you choose to implement the indexes in C++ you will get extra points. Please adhere to the following requirements for your chosen programming language:

- Python: Use Python version 3.6 or higher. Include a requirements.txt file with your submission that lists all the necessary libraries and dependencies (such as scipy, numpy, matplotlib, etc.). To do so you can use the following commands:

  ```
  pip install pipreqs
  pipreqs /path/to/project
  ```

- C++: Use standard C++11 or higher for your implementation. Include a README file that describes the compilation instructions and any dependencies required for running your code.

Regardless of the programming language you choose, your code should be well documented and accompanied by clear instructions on how to compile and run it in your submission.

3 Dataset

For this assignment, we will use the New York Points of Interest (POI) data from OpenStreetMap (OSM), a free and open geographic database maintained by a community of volunteers [2]. We will use *Overpass Turbo*, a web-based tool for querying and downloading custom OSM datasets [1]. To obtain the New York POI data, you will need to follow these steps:

1. **Access Overpass Turbo**: Go to the Overpass Turbo dashboard at `https://overpass-turbo.eu/`.

2. **Run the Query**: Use the following query in the Overpass Turbo interface to retrieve POIs from several categories (e.g., `amenity`, `shop`, `leisure`, etc.) within the New York bounding box:

   ```
   [out:csv(::id,::lat,::lon,'name'; true; ',')][timeout:3000];
   (
     node['amenity'](40.4774, -79.7624, 45.0159, -71.8562);
     node['shop'](40.4774, -79.7624, 45.0159, -71.8562);
     node['leisure'](40.4774, -79.7624, 45.0159, -71.8562);
     node['tourism'](40.4774, -79.7624, 45.0159, -71.8562);
     node['craft'](40.4774, -79.7624, 45.0159, -71.8562);
     node['office'](40.4774, -79.7624, 45.0159, -71.8562);
     node['healthcare'](40.4774, -79.7624, 45.0159, -71.8562);
     node['natural'](40.4774, -79.7624, 45.0159, -71.8562);
     node['man_made'](40.4774, -79.7624, 45.0159, -71.8562);
     node['historic'](40.4774, -79.7624, 45.0159, -71.8562);
     node['sport'](40.4774, -79.7624, 45.0159, -71.8562);
     node['place'](40.4774, -79.7624, 45.0159, -71.8562);
     node['railway'](40.4774, -79.7624, 45.0159, -71.8562);
   ```

```
    node['public_transport'](40.4774, -79.7624, 45.0159, -71.8562);
    node['highway'](40.4774, -79.7624, 45.0159, -71.8562);
  );
  out;
```

This query retrieves all nodes within the specified bounding box that belong to various POI categories. The output is formatted as CSV and includes POI ID, latitude, longitude, and name.

3. **Export the Data**: After running the query, click on the `Export` button, and select `raw data directly from Overpass API` to download the data as a CSV file.

4 Deliverables and Tasks

4.1 Part A: Data Download and Preprocessing (10 pts)

In this part of the assignment, you will download and preprocess a dataset of Points of Interest (POIs) obtained from OpenStreetMap (OSM). The goal is to prepare a clean dataset that can be used for further analysis in subsequent parts of the assignment:

1. **Download the POI Dataset:** Begin by downloading the POI dataset from OpenStreetMap (OSM) using the instructions provided in Section 3. The dataset contains a total of `825,171` POIs, including information such as unique identifiers, geographic coordinates (latitude and longitude), and attributes like the type of amenity and name.
2. **Identify and Remove Invalid Entries:** After downloading the dataset, some of the entries may contain invalid or missing values. Your task is to clean the dataset by:
   - Removing entries with missing or malformed data, such as POIs that lack the following attributes: unique identifiers (`@id`), latitude (`@lat`), and longitude (`@lon`).
   - Ensuring that the unique identifier (`@id`) consists only of numeric characters.
   - Ensuring that the unique identifier (`@id`) is unique to each POI.

4.2 Part B: Query Implementation: Brute Force Approach (30 pts)

In this part, you will implement two fundamental spatial queries — **k-nearest neighbors (kNN)** and **range queries** — using a **brute force** linear search approach. This method involves sequentially checking each entry in the dataset to find the results.

1. **k-Nearest Neighbors (kNN) Query Using Linear Search:** Implement a function to find the `k` closest POIs to a target POI using linear search.
   - Write a helper function, `euclidean_distance(point_x, point_y)`, to compute the Euclidean distance between two POIs.
   - Create a function, `knn_linear_search(dataset, target_id, k)`, that:
     - Takes as input the dataset, a target POI ID, and the number of neighbors (`k`).
     - Finds the target POI and computes its distance to all other POIs.
     - Returns a list of the `k`-nearest neighbors and their distances.
   - Test your function by randomly selecting different target IDs and varying `k` and dataset sizes (`N`):
     - `k` $\in$ {1, 5, 10, 50, 100, 500}, `N` $\in$ {1,000, 10,000, 100,000, 825,171}.
   - Plot the query execution times and write your observations.
2. **Range Query Using Linear Search:** Implement a function to find all POIs within a specified distance `r` from a target POI using linear search.
   - Create a function, `range_query_linear_search(dataset, target_id, r)`, that:
     - Takes as input the dataset, a target POI ID, and a radius `r`.
     - Returns a list of POIs within the radius `r` from the target.
   - Test your function by randomly selecting target IDs and varying `r` and dataset sizes (`N`):
     - `r` $\in$ {0.01, 0.05, 0.1, 0.2, 0.5}, `N` $\in$ {1,000, 10,000, 100,000, 825,171}.
   - Plot the query execution times and write your observations.

4.3 Part C: Implementing Spatial Indexes: Grid Index and KD-Tree (60 pts)

In this part, you will implement two types of spatial indexes — a **grid index** and a **KD-Tree index** — to optimize the performance of k-nearest neighbors (kNN) and range queries. After implementing these indexes, you will run the same queries as in the brute force approach, compare the performance of these methods, and analyze the results. You should not use built-in implementations of these indexes such as `scipy.kdTree`!

1. **Implement a Grid Index:** A grid index divides the space into uniform cells and assigns each point to a cell based on its coordinates, restricting the search to relevant cells near the target point.
   - Implement a function, `build_grid_index(dataset, cell_size)`, that:
     - Takes as input the dataset of POIs and the size of each cell (`cell_size`).
     - Assigns each POI to the appropriate grid cell based on its latitude and longitude.
     - Returns the grid index.
   - Experiment with different cell sizes (`cell_size` $\in \{0.01, 0.05, 0.1, 0.2\}$) and evaluate their impact on query performance.
2. **Implement a KD-Tree Index:** A KD-Tree is a binary search tree that recursively partitions the space into half-spaces, adapting to the data distribution.
   - Implement a function, `build_kd_tree(dataset)`, that:
     - Takes as input the dataset of POIs.
     - Constructs a KD-Tree from scratch using the coordinates (latitude, longitude) of the POIs.
     - Returns the root node of the constructed KD-Tree.
   - Use your KD-Tree implementation to perform spatial queries by traversing the tree and comparing distances to find the nearest neighbors or points within a specified range.
3. **Run Queries and Compare Performance:**
   - Use your implemented Grid Index and KD-Tree to run the same kNN and range queries with the same configurations (values of `k`, `r`, and dataset sizes) as in the brute force approach.
   - Verify the correctness of your results by comparing the outcomes from the brute force approach with those from the indexing methods.
   - Measure and report the query execution times for each indexing method.
   - Compare the performance of the Grid Index and KD-Tree against each other and against the brute force approach.
   - Provide observations on the impact of different indexing methods on query performance, including any differences observed with varying data distributions, grid cell sizes, and dataset sizes.

5 Submission Guidelines

You should submit a single zip file named `YourName_USC_ID.zip` on Brightspace. This file must contain a folder with your code, a `ReadMe` file with clear instructions on how to run your code, and a pdf file named `Results` that includes your plots, comparisons, and observations.

References

[1] Olbricht, R. Overpass turbo. `https://overpass-turbo.eu/`.

[2] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . `https://www.openstreetmap.org`, 2017.