Fall 2024 EC504 Final Report:

Hashing and Trees for Fast Approximate Nearest Neighbor *Geospatial* Search

Chris Krenz, ckrenz@bu.edu, U19402301

Table of Contents

General Description

This project is based on the Suggested Project 5: Hashing and Trees for Fast Approximate Nearest Neighbor Search. I implemented multiple algorithms/data structures to perform approximate nearest neighbor search of geospatial data and compared their relative speed and accuracy. Specifically, given a pair of coordinates, the program can identify zip codes near those coordinates. I successfully implemented search algorithms based on:

- Multi-Table Locality-Sensitive Hashing (LSH)[1]
- Approximate K-D Trees with Priority Search[2,3]
- R-Trees[4]

I experimented with a few datasets (such as Open Street Map[5]), including data on time-zone, population, and elevation, but these additional dimensions did not prove useful. They were likely too sparse, as OSM is missing a lot of elevation data, and the other dimensions would only be loosely correlated with zip codes and coordinates. So the primary data source I relied on was:

- SimpleMaps[6] (US zip code and longitude/latitude data consolidated from USPS, Census Bureau, etc.)

As a final goal, I also implemented a simple interactive application that allows users to perform these searches in real time. This app allows the user to:

- Enter coordinates and select one of the 3 algorithms
- Click 'Run' to generate a map that displays markers for the target coordinates and suggested nearby zip codes

(My presentation slides contain more info:
https://docs.google.com/presentation/d/1cgaXUtRxTxCplw3CdPCHHPpMtw8CO0HeHZIf2TlUmZg/edit#slide=id.p)

The Data

I had some challenges accessing and merging data. Initially, the OSM data was not accessible (for several weeks, downloads were simply failing, as others also reported on forums, though more recently they started working again), and even when accessible, the data can be difficult to effectively merge, as some datasets do not share coordinate or zip code data and come in a variety of different formats (.gpkg, .pbf, .csv, etc.). I did experiment with a few additional dimensions, such as timezone, but this did not improve the algorithms. Given time constraints, I ultimately decided to forego the higher dimensional data.

This is unfortunately as I would have expected LSH to perform especially well with higher dimensional data, though the number of additional dimensions that would be required to make a substantial difference would likely be prohibitively large anyway. I was also hesitant to use excessively large datasets, as it would make it more difficult for you to run the code independently (OSM data alone would be on the order of 10s of GB). As such, I relied on the lower-dimensional SimpleMaps data, which provides data on US zip codes and coordinates.

Critically, this will also allow me to package the data directly with the code to simplify the instructions as much as possible.

The Algorithms/Data Structures and Other Program Features

- Search based on multi-table Locality-Sensitive Hashing
    - To implement Multi-Table LSH, I created multiple hash tables where each table uses a different set of hash functions. For each query, I combined the results from these tables to find candidate nearest neighbors, thereby improving the chance of capturing similar items across different hash “views” of the data. The main intuition here is that while a single hash function might miss some close neighbors (leading to false negatives), using multiple hash functions and tables can help recover them. This approach reduces reliance on any single hashing scheme and thus can boost overall recall.
    - Although our data is low-dimensional (e.g., just latitude and longitude), LSH methods are often more beneficial in high-dimensional spaces where more traditional tree-based methods degrade in performance. In very high dimensions, exact nearest neighbor search becomes difficult and approximate methods like LSH can efficiently find points close to the query, even if they are spread out in a large and complex search space. Our LSH implementation therefore serves as a proof-of-concept that could scale better with higher-dimensional data.

- Search based on K-D Trees with Priority Search
    - I first construct a K-D tree, recursively partitioning the search space along alternating dimensions (e.g., splitting on latitude at one level, longitude at the next; see the diagram in the slides…), resulting in a binary tree structure that can be pruned during search.

  - During the search phase, I maintain a priority queue where nodes are prioritized by their distance from the query coordinates. By always expanding the most promising nodes first, the algorithm can quickly narrow in on the nearest neighbors. This priority-based approach allows the K-D tree to significantly cut down on the amount of data examined, improving lookup speeds while maintaining decent accuracy, especially in low-dimensional scenarios like ours.

- Implement R-Trees
  - To implement R-trees, I created a tree structure where each node represents a bounding rectangle that encloses a set of spatial objects. This structure dynamically adjusts as data is inserted, splitting nodes to keep bounding boxes minimally overlapping. Reduced overlap between bounding boxes means that during querying, fewer branches of the tree need to be explored.
  - The logic behind R-trees is that by encapsulating data points into nested rectangles, we can quickly discard large areas of space that are too far from the query point without examining each data point individually. Although our tree construction is relatively simple, the R-tree design proves effective for spatial data.

- Benchmarking algorithm to measure the speed and accuracy of these search algorithms
  - I measure the speed by tracking the average query time per search and the time required to build the data structure initially. This includes both the insertion time and the search time.
  - For accuracy, I compare the retrieved neighbors against a brute-force baseline, which enumerates all points and finds the actual closest ones. If the suggested neighbors align well with these “true” neighbors, we consider the algorithm accurate. Methods that return at least one of the top actual nearest neighbors are credited as ‘correct’.

- Interactive App
  - I implemented a GUI that displays a map and auto-focuses on the target region, placing a pin at the user-specified coordinates and marking the zip codes predicted by the chosen algorithm. This provides a visual, intuitive way to compare methods: when points cluster tightly around the query location, it indicates higher accuracy.
  - The interactive component was built with Python’s tkinter for the GUI and folium for rendering interactive map data. Tkinter simplifies creating windows, buttons, and text fields, while folium integrates seamlessly with map services (like Leaflet) to produce rich, interactive maps directly from Python. The user can thus select an algorithm, run a search, and instantly see the geographic distribution of returned neighbors.

What the program does

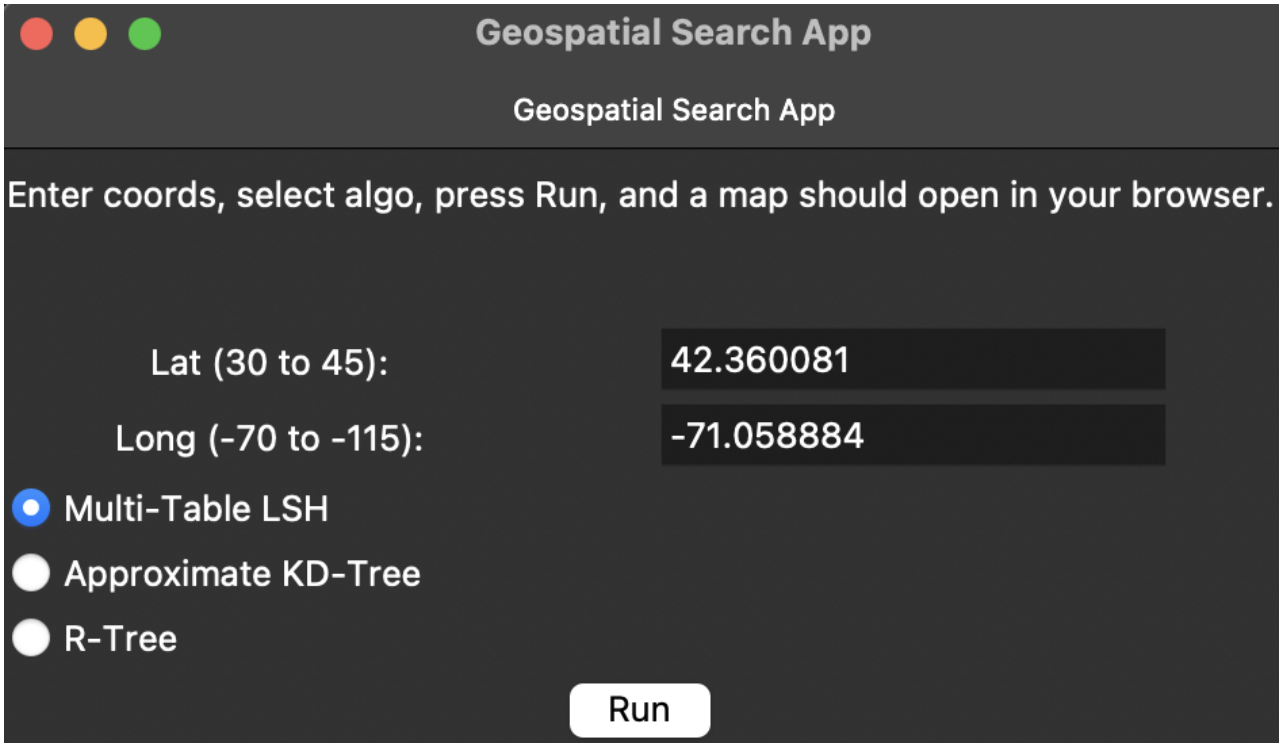
This program has scripts to perform the following:

- Importing from either a .pbf (from OSM, using the osmium package) or .csv file.
- Running either LSH-based, KD-Tree-based, or R-Tree-based search for nearby zip-codes given coordinate inputs.
    - e.g. coordinate inputs of latitude=18.34, longitude=-64.92 would yield 5 results like the following:
      `Zip Code: 00985, Location: (18.40628, -65.94767)`
- Performing a benchmarking of the three algos, reporting their accuracy (compared to brute-force) and run times.
- An interactive app that opens a window, allowing you to enter coordinates and select an algorithm. Clicking 'Run' will then create a map with the target coordinates and resulting zip codes marked. The map should open automatically in a browser.

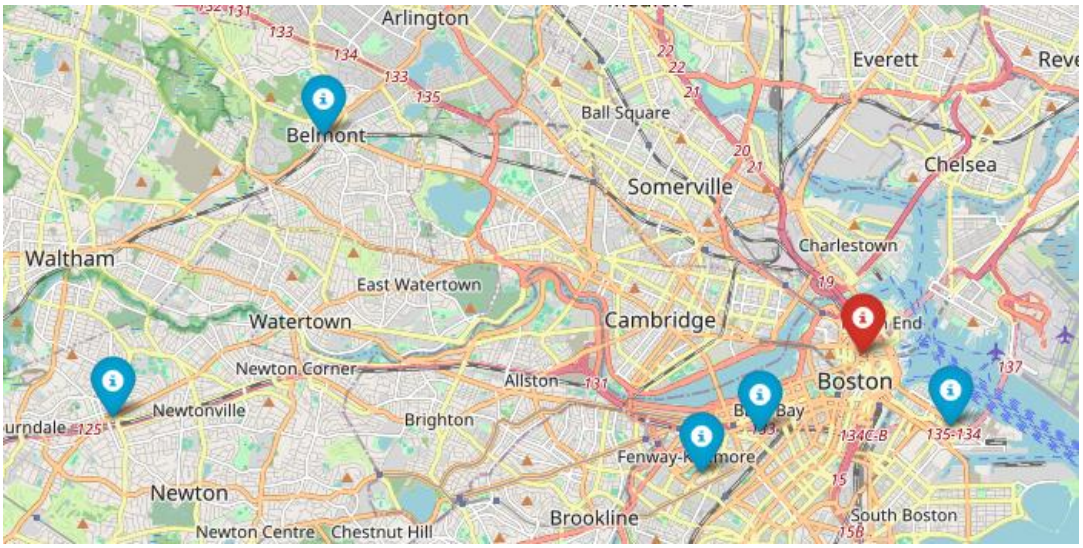
The primary point of entry is main.py. Running this algorithm will run both the benchmarker.py and app.py, resulting in an output similar to the following:

```
(venv) $ python main.py
Multi-Table LSH - Time: 0.04073s, Accuracy: 0.58
Approximate KD Tree - Time: 0.00011s, Accuracy: 0.06
```

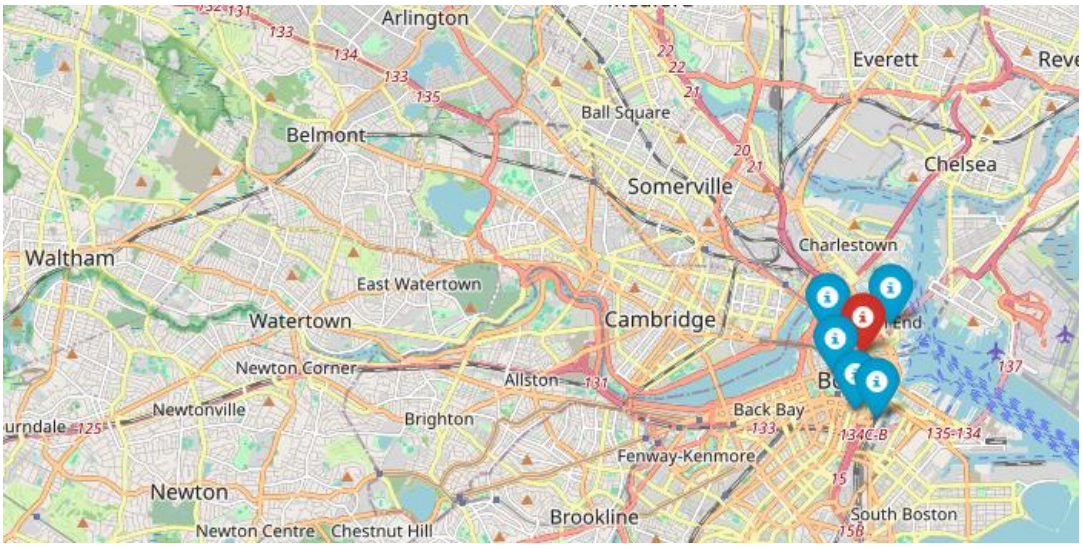
This result makes sense, as there is a clear tradeoff between accuracy and speed. LSH provides a reasonably high accuracy result but takes a ~couple order of magnitudes longer to run. Approximate KD Tree, however, is much less accurate. R-Trees offer the best accuracy and a reasonable speed, even when the max_children (the max num of children the node can hold before needing to split) is set relatively high, like the default of 64.



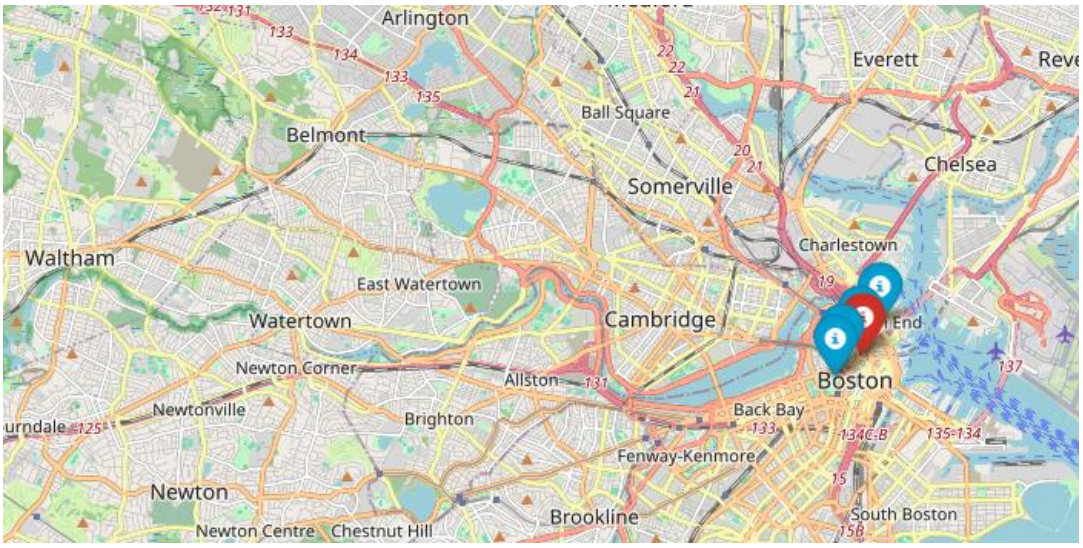
K-D Tree Search (general accuracy of ~0.07):



LSH Search (general accuracy of 0.59):



R-Tree Search (general accuracy of 0.94 with max_children at 64):



Summary of Results

	K-D Trees	LSH	R-Trees
Accuracy	0.07	0.59	0.94
Runtime (s)	0.00016	0.04088	0.00149
Notes	Fastest, but *very* low accuracy Simplest to implement	Decent accuracy, but many times slower than K-D Trees	Longer/more accurate w/ > 64 nodes (and vice-versa)

How to run the program

The code is accessible at my public GitHub repo: https://github.com/chris-krenz/504-geospatial-project. This is a Python project. Instructions for installing Python can be found here: https://wiki.python.org/moin/BeginnersGuide/Download.

You will want to perform the following steps:

1. Download the code: `git clone git@github.com:chris-krenz/504-geospatial-project.git`
2. `cd` to the root directory
3. Run the following commands to create a virtual environment, install the packages, and run the main script:

```
python -m venv venv

source venv/bin/activate          OR for Windows          venv\Scripts\activate.bat

pip install -r requirements.txt

python src/main.py
```

4. (Optionally 'deactivate' to exit the virtual environment)

Alternatively, you can build and run the benchmarker with **Docker** using the following command:

```
docker-compose up --build
```

After a little time (about a minute on an M2 mac), you should see a console output similar to that shown above. Note, the Docker version does NOT run the interactive app, as getting Tkinter to work in Docker is messy.

Challenges

The primary challenge I was dealing with was to do with downloading and merging datasets. Large data files (10s or 100s of GB) associated with global–or even national–regions are failing partway through the download, making the data a bit more difficult to access than I had initially anticipated. I mostly tried to pull from OSM, as that source has a variety of different data types that could increase dimensions substantially.

I was able to test a few additional dimensions, including timezone, population, and elevation, but they did not prove useful, so I chose instead rely on the coordinate and zip code data. While I had initially hoped to provide global search, I ended up limiting this to just the US so that the data could be easily packaged with the code.

Conclusion

Although I faced some unexpected setbacks, the basic algorithms function correctly. This allowed me to properly benchmark the different algorithms to see clearly their competing speeds and accuracies. Some clear lessons emerged:

- K-D Trees were technically the fastest but with prohibitively low accuracy
- R-Trees were the most accurate by far (given they the max_children was set sufficiently high (about 4 or greater)
- R-Trees are very well suited to geospatial data
    - The dynamically sized/positioned bounding boxes effectively mimic the spatial distribution of zip codes
- The R-Trees max_children parameter (max num of children the node can hold before needing to split…) can be easily scaled up or down to optimize for accuracy or speed, respectively
- LSH would likely perform better with higher dimensional data (given the constant lookup time for hash functions), though I was unfortunately unable to confirm this.
- K-D Trees and LSH were quite easy and quite to implement. If ease of implementation is a factor, LSH is probably the best option. R-Trees, implemented manually, were a fair bit trickier, though libraries do exist to simplify the process.

Lastly, I was able to complete the interactive app to effectively showcase some of the above principles. The images above clearly show the respective accuracies of each algorithm, with the suggested zip codes clustered closely around the target coordinates for R-Trees (and somewhat for LSH), indicating high accuracy, and spread out farther away from the target coordinates for K-D Trees, indicating low accuracy. While I did not accomplish all of my stretch goals, I did accomplish my primary goals (benchmarking K-D Trees and LSH) as well as R-Trees and the interactive app, learning a great deal throughout the process.

References

1. Datar, M., Immorlica, N., Indyk, P. & Mirrokni, V. S. Locality-sensitive hashing scheme based on p-stable distributions. in *Proceedings of the twentieth annual symposium on Computational geometry* 253–262 (Association for Computing Machinery, New York, NY, USA, 2004). doi:10.1145/997817.997857.
2. A comparison of k-nearest neighbour algorithms with performance results on speech data - KU Leuven. https://kuleuven.limo.libis.be/discovery/fulldisplay/lirias1945818/32KUL_KUL:Lirias.
3. Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R. & Wu, A. Y. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J ACM* **45**, 891–923 (1998).
4. Hadjieleftheriou, M., Manolopoulos, Y., Theodoridis, Y. & Tsotras, V. J. R-Trees – A Dynamic Index Structure for Spatial Searching. in *Encyclopedia of GIS* (eds. Shekhar, S. & Xiong, H.) 993–1002 (Springer US, Boston, MA, 2008). doi:10.1007/978-0-387-35973-1_1151.
5. Geofabrik Download Server. *GeoFabrik: OpenStreetMap Data* https://download.geofabrik.de/ (2018).
6. US Zip Codes Database | Simplemaps.com. *SimpleMaps* https://simplemaps.com/data/us-zips.
7. Earth Science Data Systems, N. Find Data | Earthdata. https://www.earthdata.nasa.gov/learn/find-data (2021).
8. GADM. *Database of Global Administrative Areas* https://gadm.org/data.html (2022).