Project 1 in ANN and Clustering by :

1. Mantzouranis Gewrgios (sdi1700076)

2. Mixopoulos Mixalis (sdi1700091)

To the point

Implementation of Lsh and Clustering in order to find **k-Nearest Neighbors** in dimension-d, using metric L2 of **Euclidean Distance,** between a dataset of vertices and some queries. For the implementation of ANN part, we use **k-NN, Brute Force Search and Range Search** so that we can print some statistics for analysis.

Download Repository

$ git clone https://github.com/mixo091/Emiris-Project1.git

Compilation and Execution

- **lsh:** make lsh
  - **./lsh -d test_files/<input_file> -q test_files<query_file> -o output_file -k <int> -L <int> -N <int>-R <int>**
    - **k:** number of Lsh functions
    - **L:** number of Hash Tables
    - **N:** number of neighbors
    - **R:** radius used for Range Search
- **hypercube:** make cube
  - **./cube -d test_files/<input_file> -q test_files<query_file> -o output_file -k <int> -N <int> -M <int> -probes <int> -R <int>**
    - **k:** dimension number
    - **N**: number of neighbors
    - **M:** max number of possible items to be checked

- **probes:** max number of neighbor vertices to be checked in hypercube
- **R:** radius used for Range Search

- **make clean** : to remove object files and targets lsh, cube

Lsh

Lsh is represented by a header file **./src/ile lsh.hpp,** in which we have implemented a templated class Lsh. Our Lsh class, has a numer of (**numberOfHashtables) ,** Hash Tables (**hash_tables**) in which some data (**Data)** are inserted with the use of hash functions (**numberOfHashFunction) .** When we get a query, we find in which bucket we should insert our data in every hash table. Finally, we insert in a **map** of **std::pair<double, int> (euclidean_distance, item_id)** the **k-best candidates,** coming from every possible hash table, so that we can keep the first **N-items** sorted, until we print our stats.

Lsh file path

- Our main function → **./src/main.cpp**
  1. Parse the arguments **lsh_parse_args (./src/Utilities/Utilities.hpp**)
  2. Get dimension and amount of vectors from input file **calc_dimensions (./src/Utilities/Utilities.hpp**)
  3. Create Lsh structure by calling Lsh constructor
  4. Ask the user for re-execution of the program by typing Y/N

- Header file → **./src/LSH/lsh.hpp**
  1. **numberOfHashTables (number of hash tables)**
  2. **BUCKET_DIVIDER (macro that can change before every compilation)**
  3. **ht_Size (size of our hash table which is total vectors / BUCKET_DIVIDER)**
  4. **vecDimension (current dimension)**
  5. **numberOfHashFunctions (number of hash functions)**
  6. **w (window size can change in every execution)**
  7. **HashTable<Data<T> *> (this is our hash table structure of Data<T>)**

  - **ANN(…)** → approximate nearest neighbor function that finds the k-best candidates from L-hash tables and print some stats.

Hypercube file path

- Our main function → **./src/cube_main.cpp**
  1. Parse the arguments **cube_parse_args (./src/Utilities/Utilities.hpp)**
  2. Get dimension and amount of vectors from input file **calc_dimensions (./src/Utilities/Utilities.hpp)**
  3. Create Hypercube structure by calling Hypercube constructor
  4. Ask the user for re-execution of the program by typing Y/N

- Header file → **./src/hypercube/hypercube.hpp (Derived class of Lsh)**
  1. **probes (number of neighbor buckets to check)**
  2. **M (number of total vectors to check)**
  3. **std::unordered_map<int, int> f_map (for every point p → f(h(p)) → 0 ? 1)**

  - **hypercube(…)** constructor calls **Lsh(…**) constructor in order to initialize the hash table size, number of hash function needed etc… For every vector, a bucket number is calculated and for every bucket number a 0 or 1 is created and stored in **f_map.**
  - **check_key(…)** → Returns 0 or 1 of bucket value exists, else new input is inserted in **f_map** and 0 or 1 is created for this value with **Coin Toss probability.**
  - **cube_hashing(…)** → Finds bucket of hypercube for the data given.
  - **get_neighbors_by_distance(…)** → Returns a vector which contains the neighbors of query point for every hamming distance from 1 to n, where n = power(2, max dimension).
  - **ANN(…)** → For every query set, we find it's neighbors. We check the bucket of hashed query at first and if **M** items are not searched yet, we move into the neighbors vector until we search all **probes** neighbors.

Hash table file path

- Header file → **./src/HashTable/HashTable.hpp**

  1. **buckets (number of buckets)**
  2. **table_size (size of hash table which is total vectors / BUCKET_DIVIDER = 4,8,16)**
  3. **struct Bucket<K> **hash_table (these are our buckets)**
  4. **h_fun (every hash table has it's own hash function)**

  - **insert(…)** → find hash value of item and then insert the item in the list of buckets (used for LSH).
  - **insertHyperCube(…)** → insertion for hypercube.
  - **search_NN_in_radius(…)** → helper function to find nearest neighbors in radius given by the user. This functions traverses the list of Data of each bucket, computes the euclidean distance between the query and stores the id's of distances which have eu_distance < radius. **We have 2 instances of these functions, one for LSH and the other for Hypercube. What changes is the definition of those functions, but the logic is the same.**
  - **search_NN(…)** → helper function to find the k-nearest neighbors and store them in a map. **We also have 2 instances of this function.** The difference is that for Hypercube, we have to stop searching when M items are searched or probes are reached.
  - **find_NN(...)** → find NN for **hypercube**, checking hashed bucket at first and if needed every other neighbor for this bucket, with the calculation of hamming distance from I = 1 ….. max_dimension = k (given by the user). For **LSH**, we calculate euclidean distance with the use of query_trick and we keep the best-k pair of <eu_distance, id> for every bucket in a map.
  - **range_search(...)** → For **hypercube** we call search_NN_in_radius(…) and until M or probes are reached we keep searching in vector of neighbors for distances < radius. For **LSH,** we return a vector of id's with eu_dist < radius.

In short, a **Bucket** of our Hash table is a **list** of **bucket entries**, where a bucket entry has a **query_trick (ID : used from theory as Query Trick to avoid calculating Euclidean Distance, between query and every point in Bucket)** and a key of type **Data<K>** where Data, is a structure that we use to store our data. **Data (./src/Data/Data.hpp)** is a template struct holding an id and a vector of items. (**vector<T> T = double, int).**

Hash function file path

- Header file→ **./src/HashFunction/HashFunction.hpp**
  1. **w (window size)**
  2. **t (normal distribution(0, w))**
  3. **num_of_hfun (number of hash functions)**
  4. **d (dimension)**
  5. **M (a constant for modulo operations)**
  6. **int *r (an array of r values for g calculation)**
  7. **int *h (h[p]= (r * v + t / w)**
  - **HashFunction(…) constructor** → initializes w, t, num_of_hfun, dim, M, int *r, int *h values.
  - **hfunction(...)**→ computes and returns **h[p]= (r * v + t / w)**
  - **hashValue(...)**→ computes and returns g(p) (amplified hashed function)