National and Kapodistrian University of Athens
Department of Informatics and Telecommunications

SOFTWARE DEVELOPMENT FOR ALGORITHMIC PROBLEMS

Searching & Clustering of Vectors & Curves
December 2021

Team No. 59
Maraziaris Charalampos - 1115 2018 00105
Chalkias Spyridon - 1115 2018 00209

Table of Contents

0. Abstract

0.1 Main goal

The purpose of this project is to demonstrate a few techniques designed for vector and curve searching and clustering, in order to highlight the need to use efficient algorithms for big data management.

0.2 Project's Structure

The folder structure of the project is as follows:

- `/data/:` Includes all files that are necessary for the project, such as input, query and configuration files.
- `/cont_frechet_repo/:` Files included from the FRED repository.
- `/src/:` Includes all project's source files.
- `/tests/:` Contains the main test function for each executable and the unit-tests.
- `/CMakeLists.txt:` Instructions for CMake, in order to fully build the project.

- `/src/bruteforce/:` Includes all source files in which bruteforce method has been implemented for K-NN.
- `/src/lsh/:` Folder containing Locality Sensitive Hashing implementation.
- `/src/hypercube/:`Folder containing the implementation of Hypercube projection.
- `/src/clustering/:` Fully functional clustering module. Also contains the implementation of a *Complete Binary Tree*.
- `/src/util/:` Utilities module.
- `/src/curve/:` Contains the *Curve*, *Grid* and *Point* class representations.
- `/src/lsh/hash_functions/:` Module in which hash families and amplified hash functions are implemented.
- `/src/lsh/hashtable/:` Custom constant size hash table using separate chaining method.
- `/src/util/metrics/:` Contains the Euclidean, Discrete Frechet and Continuous Frechet distance metrics and also the Optimal DFD Traversal generator function.
- `/src/util/dataset/:` Wrapper class for the project's dataset.
- `/src/util/files/:` Folder providing reading/writing functionality for the project's files.

1. Usage

1.1 Build

This project uses CMake in order to be built. In order to do so, while being in the project's root directory, simply type:

```
cmake .
```

There are five (5) executables available in the project, **euclidean-lsh**, **discr-frechet**, **cont-frechet**, **cluster** and **unit-tests**.
In order to produce them, type:

```
make <executable_name>
```

where *executable_name* is either one of the aforementioned executables.

1.2 Run

Each executable receives its own parameters, as shown below:

1.2.1 Vector and Curve Searching (A1-A2-A3)

```
$ ./search -i <input file> -q <query file> -k <int> -L <int> -M <int> -probes <int> -o <output file> -algorithm <LSH or Hypercube or Frechet> -metric <discrete or continuous | only for -algorithm Frechet> -delta <double>
```

1.2.2 Clustering (B)

```
$ ./cluster -i <input file> -c <configuration file> -o <output file> -update <Mean Frechet or Mean Vector> -assignment <Classic or LSH or Hypercube or LSH_Frechet> -complete <optional> -silhouette <optional>
```

Where:

- `-i` : Relative path of the input file.
- `-q` : Relative path of the query file.
- `-o` : Relative path of the output file.

- `-c :` Relative path of the configuration file.
- `-k :` Number of hash functions for each hash family.
- `-M :` Number of hash tables for LSH / Hypercube.
- `-L : //TODO ?`
- `-probes :` Maximum permissible number of hypercube vertices to be checked.
- `-complete :` Be verbose when presenting clustering results.
- `-delta : // TODO`
- `-silhouette :` Also calculate Silhouette score after clustering.

1.3 Unit Tests

Run:

```
$ ./unit_tests
```

In our Unit Tests we chose to include 3 deterministic methods our program utilizes, namely:

- The **Complete Binary Tree** module, (test name: "test_cbt") where we test the exact same *post-order* functionality we employ in the *Mean Curve* task, with a sum of integers.
- The **Discrete Frechet Distance** metric, (test name: "test_dfd") where we compare the results of our *Dynamic Programming* implementation with the results of a *brute-force* implementation.
- The **Optimal Traversal Generator** of 2 curves, (test name: "test_opt_traversal") where we compare the result value and other various elements, such as that every point is included from both curves.

The tests also helped us monitor and resolve memory leaks in our deployed implementations. We used the simple acutest testing library to implement our unit tests.

2. Searching (A1, A2, A3)

General implementation notes :

- In the Grid class, noise represented by '**t**' in theory, is being chosen uniformly in **[0, grid interval(δ)]**.
- Grid interval **(δ-delta)**, is being calculated by computing the *average* points of all curves in the input dataset and returning approximately:
  *10e-4 * data_dimensions * the_above_average.*
- The distance between each input curve and a given query curve is being calculated on the initial **raw** curves and **not** the preprocessed ones.
- Padding value is being set to a value: **1e9**, as it performed well experimentally. Also, padding limit is being set **dynamically** by choosing the maximum curve length amongst all input curves.
- Window size is being set by hand to value: **1000**, as it performed well experimentally.
- When reducing the size of the curves by filtering, the filtering threshold is being set by hand to the value: **10**, as again it performed well experimentally.
- When **projecting** the curves to a grid (1 or 2-dimensional), the following repetitive formula is being applied to every coordinate:
  *floor((q - t)/δ + 1/2)*δ + t,*
  where q is a vector query, t is the noise and δ is the grid interval.

2.1 A1

All input and query curves are being processed in order to trim the x-axis coordinate and then they are being given as input to LSH algorithm, with Euclidean distance as a distance metric.

2.2 A2

When using Discrete Frechet as a distance metric, all input and query curves go through the following pipeline:

1. They are being fit into a 2-dimensional grid, by applying the already introduced repetitive formula to each curve's coordinate.
2. Then, every consecutive duplicate in the grid is being removed.
3. The curves are being flattened to 1-dimensional vectors.
4. Padding is being applied to the resulting vectors in order for every input vector to have the same size for LSH algorithm.
5. LSH algorithm is being run.

2.3 A3

When using Continuous Frechet as a distance metric, the input and query curves are being preprocessed as follows:

1. The size of the curves is being reduced by applying filtering to all the curves.
2. The x-axis coordinates are being deleted, in order to end up with 1-dimensional input vectors.
3. All the aforementioned vectors are being projected into a 1-dimensional grid, by applying the already introduced repetitive formula.

4. In each vector, min-max filtering is being applied in order to further reduce the size of the input vectors and keep as much useful information as possible.
5. Padding is being applied to all vectors.
6. LSH algorithm is being run.

3. Clustering (B)

The cluster module in this assignment has been upgraded to a ***template class*** that takes as parameters the *ItemType* to cluster (either *Curves* or *FlattenedCurves* (Points)) and a *DistanceFunction* (either *Euclidean* for Points or *Discrete Frechet* distance for Curves).
We also employed a ***wrapper class*** (named *cluster*) that utilizes our template class (named *internal_cluster*).

3.1 K-Means++

Clustering is being implemented, using the K-Means++ method. More specifically K-Means++ combines the classic K-Means algorithm with the **Initialization++** procedure, as presented in the following algorithm:

```
INITIALIZATION++ : K-MEANS++ / K-MEDOIDS++:
  (1) Choose a centroid uniformly at random; $t \leftarrow 1$.
  (2) $\forall$ non-centroid point $i = 1, \ldots, n - t$,
      let $D(i) \leftarrow \min$ distance to some centroid, among $t$ chosen centroids.
  (3) Choose new centroid: $r$ chosen with probability proportional to $D(r)^2$:
          $\text{prob}[\text{choose } r] = D(r)^2 / \sum_{i=1}^{n-t} D(i)^2.$
  Let $t \leftarrow t + 1$.
  (4) Go to (2) until $t = k =$ given $\#$centroids.
```

Note:

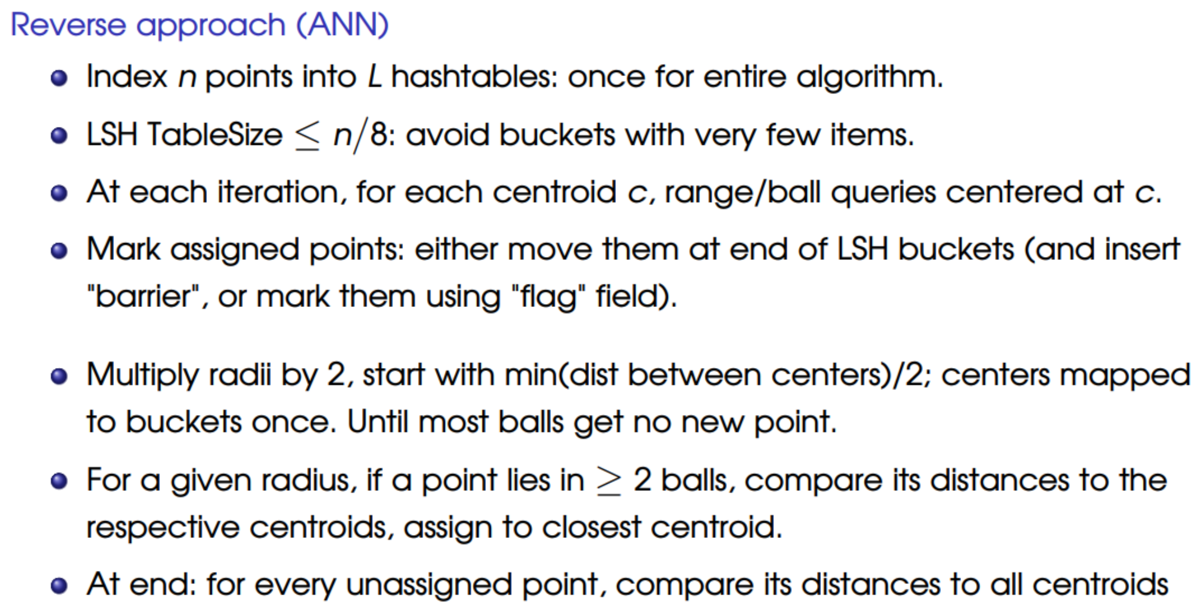
- When the matrix of probabilities is filled up, a number is picked **uniformly** in [0, 100], indicating the element to be selected as the next centroid. Subsequently, an iteration is performed through the probabilities matrix until the cumulative probability of the visited elements is greater than the uniformly selected number. The stopping index corresponds to the datapoint that will be selected as the next centroid.

3.2 Assignment

3.2.1 Exact Lloyd's

Exact Lloyd's assignment method is implemented by calculating **for every datapoint** its distance to **every centroid** and thus assigning each datapoint to its nearest centroid.

3.2.2 Reverse Assignment



Reverse approach (ANN)

- Index n points into L hashtables: once for entire algorithm.
- LSH TableSize $\leq n/8$: avoid buckets with very few items.
- At each iteration, for each centroid c, range/ball queries centered at c.
- Mark assigned points: either move them at end of LSH buckets (and insert "barrier", or mark them using "flag" field).
- Multiply radii by 2, start with min(dist between centers)/2; centers mapped to buckets once. Until most balls get no new point.
- For a given radius, if a point lies in ≥ 2 balls, compare its distances to the respective centroids, assign to closest centroid.
- At end: for every unassigned point, compare its distances to all centroids

We implement the reverse assignment algorithm exactly as described above.
A few implementation details worth noting are the following:

- We use a **set** to keep every **unassigned** point.
- For each fixed radius (i.e. in each iteration) we use a **map**, mapping **Points** to **(Centroid, Distance)** tuples, denoting the **local** assignment of each point to a centroid, and the respective distance between them.
- For each fixed radius, if a point appears in the range search results of **two** different centroids, we compare the distances from the point to each one of the centroids and keep the minimum, updating the previously described map accordingly.
- For each fixed radius, after the range searches are over, we propagate the **local** assignments (stored in the map described above) to the **global** assignments, which are stored in a **map** mapping **Centroids** to a **list of Points**.
- In order to avoid duplicate source code we use **C++ templates** to differentiate between Reverse Assignment using **LSH** and **Hypercube**, as the underlying containers to perform the range search queries are subject to change, but every other aspect of the algorithm remains the same in code.

3.3 Update

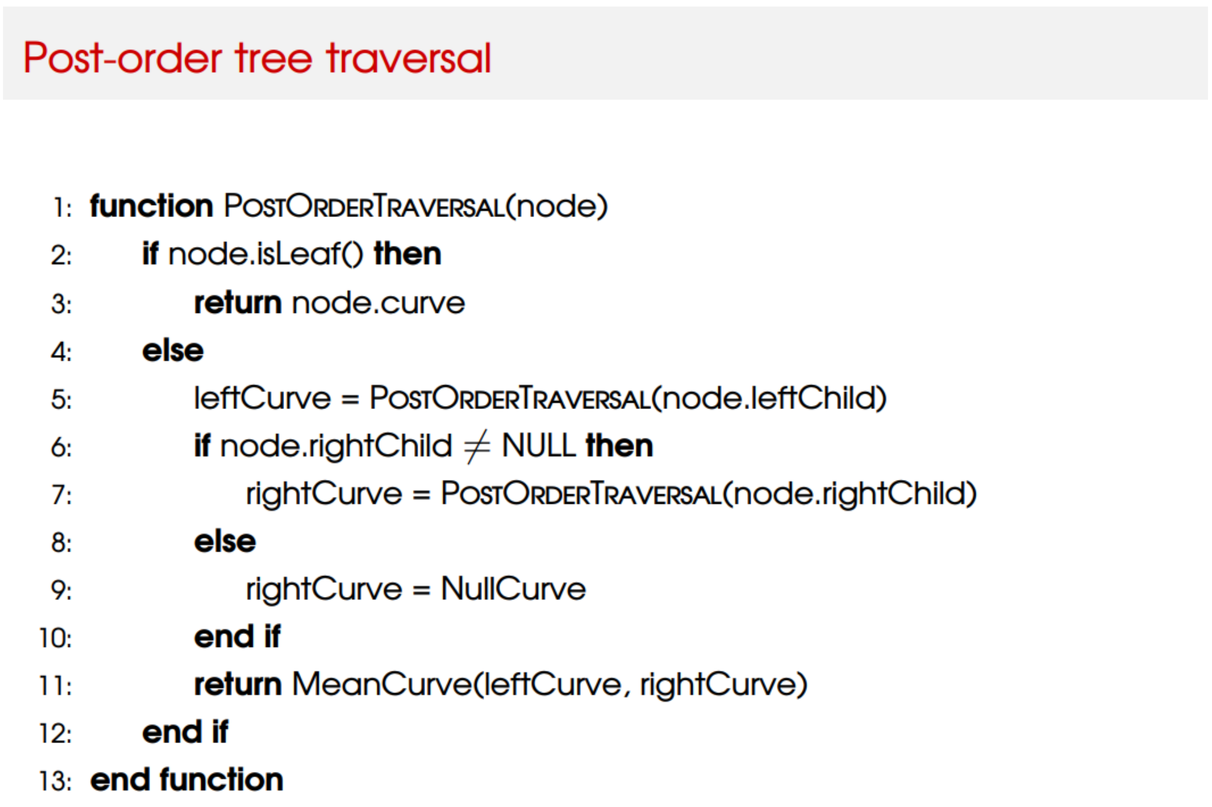
3.3.1 Mean Vector

The classic *Mean Vector* algorithm is implemented, just like in the first assignment.

3.3.2 Mean Curve

In order to implement the *Mean Curve* update algorithm, we make use of the *Complete Binary Tree* module.

The **Complete Binary Tree** is implemented using a **vector** as the underlying container. Its size is the exact minimum required to store the given leaves and the internal nodes.
The algorithm implemented to find the *Mean Curve of the n curves* in a single cluster is the following, from the course's slides:

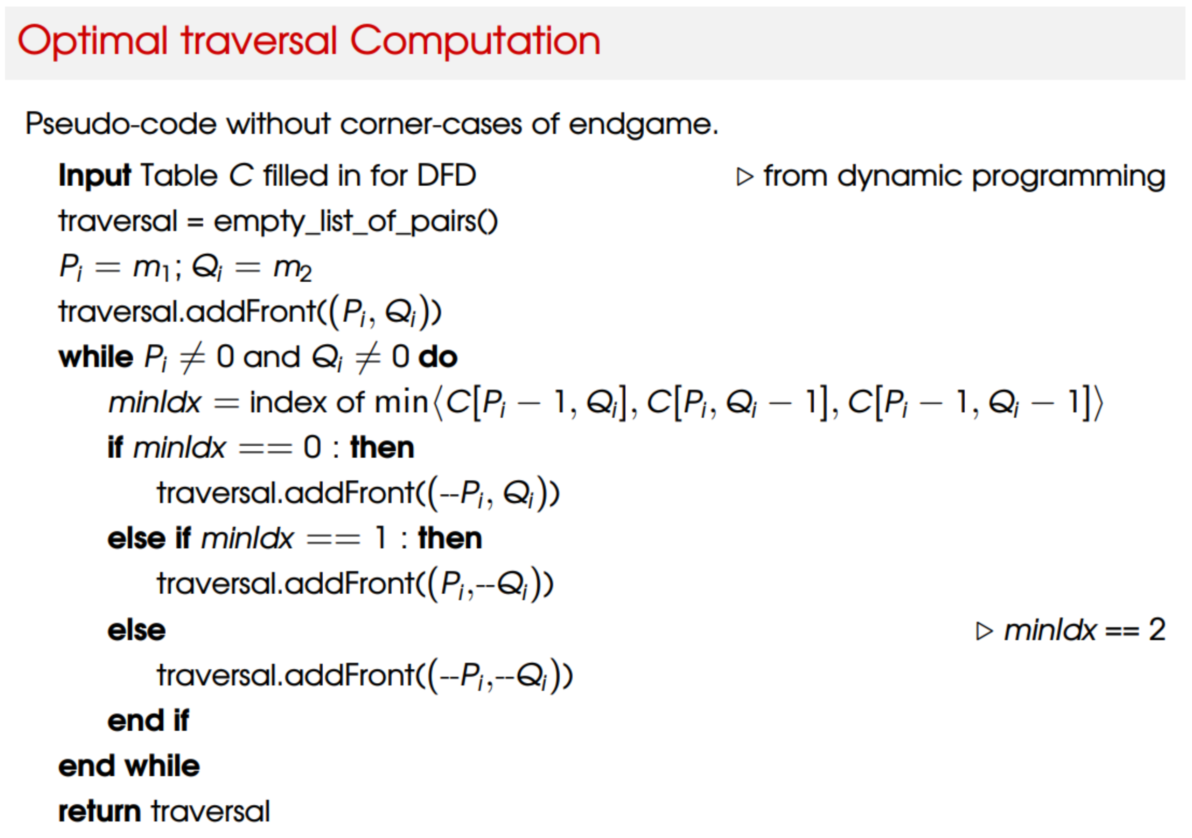


Post-order tree traversal

```
function PostOrderTraversal(node)
    if node.isLeaf() then
        return node.curve
    else
        leftCurve = PostOrderTraversal(node.leftChild)
        if node.rightChild $\neq$ NULL then
            rightCurve = PostOrderTraversal(node.rightChild)
        else
            rightCurve = NullCurve
        end if
        return MeanCurve(leftCurve, rightCurve)
    end if
end function
```

The *Mean Curve* of **2** curves is computed by finding an *Optimal Traversal* between the 2 curves and then taking the average point of the matched coordinates in the Optimal Traversal (using Euclidean distance).

To find the *Optimal Traversal of 2 curves*, we also followed implemented the algorithm of the course's slides:



Optimal traversal Computation

Pseudo-code without corner-cases of endgame.

```
Input Table $C$ filled in for DFD                    ▷ from dynamic programming
traversal = empty_list_of_pairs()
$P_i = m_1; Q_i = m_2$
traversal.addFront($(P_i, Q_i)$)
while $P_i \neq 0$ and $Q_i \neq 0$ do
    $minIdx$ = index of $\min\langle C[P_i - 1, Q_i], C[P_i, Q_i - 1], C[P_i - 1, Q_i - 1]\rangle$
    if $minIdx == 0$ : then
        traversal.addFront($(--P_i, Q_i)$)
    else if $minIdx == 1$ : then
        traversal.addFront($(P_i, --Q_i)$)
    else                                             ▷ $minIdx == 2$
        traversal.addFront($(--P_i, --Q_i)$)
    end if
end while
return traversal
```

It is worth noting that we **filter** the final Mean Curve of the n curves to reduce its complexity and thus speed up our computations, until it becomes lower or equal to the complexity of the input curves. This *filtering* operation is performed just before we assign the mean curve as our new **centroid**.

3.3.3 Stopping Condition

Each time all the centroids are updated with new ones, the difference of each new individual centroid from its old position is being computed. Then, if the **maximum** calculated difference is below a preset threshold, the stopping condition is fulfilled and the clustering stops. The aforementioned threshold has been experimentally set to **10**.

3.4 Evaluation: Silhouette

We implement the Silhouette evaluation metric by *brute-force* computing the average total distance:

1) Between the **point (or curve)** and the other points in the **same** cluster (**a(i)**).
2) Between the **point (or curve)** and points residing in the **closest** external cluster (**b(i)**).

We then compute $s(i) = (b - a) / max\{a, b\}$ while also keeping track of the **overall** Silhouette value.
We attribute a Silhouette value of **0** in edge-cases where 0 or 1 points are assigned to a cluster.

END OF DOCUMENTATION