Ανάπτυξη Λογισμικού Για Αλγοριθμικά Προβλήματα

Χειμερινό Εξάμηνο, Ακαδημαϊκό Έτος 2021-2022

Χρήστος Τσούχλαρης 1115201800204

Ιωάννης Ροβιθάκης 1115201800164

Github link: https://github.com/rvthak/AlgoProject1

Project Structure

- Args.h/cpp : Structures that hold the arguments for each task provided by the user
- List.h/cpp : A simplified version of the ShortedList used for Range Search
- Vector.h/cpp : This file holds all the structures for Vectors, Centroids & structures that contain them both and their relationships
- bucket.h/cpp : The bucket struct that holds vector points and is used for the searching algorithms
- distributions.h/cpp : Functions for uniform & normal distributions
- hash_cube.h/cpp : The Hypercube struct & it’s helper methods
- hash_fun.h/cpp : This file has the H, G & F hash function structs
- hash_lsh.h/cpp : The LSH struct & it’s helper methods
- main_Cluster.h/cpp : Main function for clustering analysis
- main_Cube.h/cpp : Main function for Hypercube analysis
- main_LSH.h/cpp : Main function for LSH analysis
- shortedList.h/cpp : The ShortedList structure that is sorts itself when you insert vectors and is used for K-Nearest-Neighbors & Range Search
- timer.h/cpp : Timer functions for statistical analysis
- utils.h/cpp : Utility functions for all the other files like numerical methods, file manipulation, printing functions

LSH

Our implementation of LSH works based on the **MultiHash** struct. We create an array of Hash Tables, each one with its own **unique G hash function**. (G functions are based on H functions in line with the theory and are randomized following the needed distributions).
We insert every input Vector to every HashTable to take advantage of the randomness of the hash functions and be able to get good search results with very high probability. When searching a query in the **MultiHash,** we check for it on every HashTable and keep the best results across all the hashtables.

When it comes to the hash function arguments (M, W, ri), we played around with the values and through trial and error we ended up with the values you see. We ended up fine tuning them on a trade-off between better accuracy and execution time.

Hypercube

To index the vectors from the input files, we use an F **hash function** struct that is based on the **H hash function** struct for the LSH algorithm. This generates a bit array that is converted to its equivalent decimal value. This number serves as the hash key for the main hash table.

For each query vector we search all the neighbors and fill a ShortedList list that has every element sorted. We search through the hypercube hash table and add elements to the list until we reach the search limitations. More specifically the search stops when we reach the maximum amount of buckets searched or the maximum amount of vectors to check. After that we return this shorted list. All the vectors in this list are sorted by their distance to the query vector (minimum to maximum).

- For K - Nearest neighbors we take the K - first elements.
- For Range Search we take the elements that have a distance that is equal to that of the range specified.

Cluster

We implemented Lloyds algorithm using cluster average for the update stage.
For the Assignment stage we implemented three methods:
Classic > Following the exact methodology, for each Vector, we calculate its distance with every existing Centroid and assign it in the nearest one.
LSH > We add all the Vectors in the LSH structs and we use Centroids to reverse search for Vectors to add to their cluster by continuously increasing the range-search range.
Hypercube > The same logic with LSH but using the Hypercube struct
In the end we calculate the Silhouette in order to evaluate how good are our clusters.

Silhouette

We calculate the result’s silhouette following the formula from the slides, using the current assignment of Vectors to Centroids. Silhouette takes a substantial amount of time to run because of the need to find the second nearest Centroid for each calculation.

Compilation & Execution

After extracting the project, start by running: **make init**
(Make sure you add the input/query file paths in the makefile variables)
Then you can use any of the options below:

LSH:

make lsh

Cube:

make cube

Cluster Classic:

make cluster

Cluster LSH:

make cluster_lsh

Cluster Cube:

make cluster_cube

Notes:

1) The makefile contains all the console arguments (He have our own fine tuned arguments). You can edit them from there
2) Valgrind is integrated in the makefile if you need it
3) We could have improved out range search times even more. We currently search for already existing items in the search results list in O(n). Using a map or a hash table we could have reduced it to O(logn) or O(1) - with a memory tradeoff - respectively. On range search we do a lot of searches in the list so a small complexity improvement should improve our times significantly.
4) In the clustering problem, in the initialization phase, our random init seemed to perform better than our implementation of initialize++ so we left the random init in the final build. (initialize++ function still exists and you can uncomment it on main to try it) In our tests, Initialize++ generally improves the convergence speed but reduces the Silhouette relatively to the random approach.

For any further questions/suggestions, feel free to contact us at any time at: sdi1800164@di.uoa.g