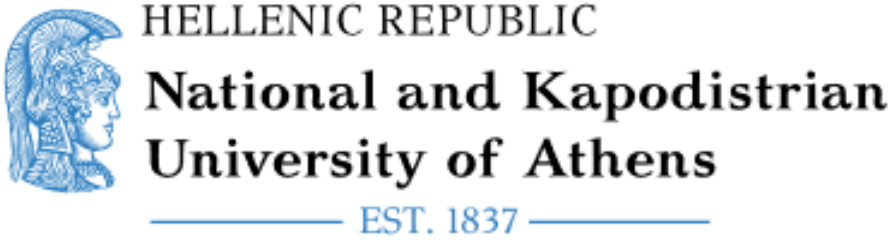
Locality Sensitive Hashing & Hypercube 2017-2018

Author: Nemanja Nedic
A.M. 1115201400124



Contents

Intro

Compile

Makefile is porvided for compilation.
LSH: make
HyperCube: make cube
Other: make clean, make count

Run

Run LSH:
./lsh -d input_file_path -q query_file_path -o output_file_path -k int -L int

Run Hypercube:
./cube -d input_file_path -q query_file_path -o output_file_path -k int -M int -probes int -M int

NOTE: Only the -d argument is necessary in each case. All other are either optional and will be set to default values (printed at start) or can be provided in the program from cin

Organization of directories and files

Version Control: Git/Github

The project is organized in 3 directories:
Shared: .hpp/.cpp files that are shared between cube and lsh
HypercubeLSH: cube .hpp/.cpp files
LSH: lsh .hpp/.cpp files

Inside each you will find the following:
main.cpp: general outline of the program
ErrorCodes.hpp: error codes used to handle irregular termination
myvector: a std::vector with a std::string id CosineSimilarity: Class CosineSimilarity
Euclidean: Class Euclidean
Metric: Abstract Class Metric

HashTable: Class HashTable
ReadInput: Functions to pare argument and read data sets
LSH: Implementations of LSH for lsh or cube
utility: Various functions and templates (ie Exhaustive Search, Hamming Neighbors, Comparing vectors, Modulo operation, Vector Norms, progress bar, ...)

Design details

You can change the coordinates from double to int by changin the typdef double coord to typdef int coord in Shared/myvector.hpp line:8

The program will first preprocess the dataset and read it to a list and afterwards initialize HashTables from that list. If we knew the number of vectors before we read the dataset we could skip the list and read directly to the HashTable.

HashTable

The HashTable consists of a abstract class pointer (Metric*) that hold the metric (Euclidean or Cosine) related to that HashTable.

The table's buckets are nothing more than a $std :: vector < myvector >$. The reason i used a $std :: vector$ here instead or $std :: list$ or $std :: forward\_list$ is that after testing for both $std :: vector$ seemed to be same (sometime more efficient) even for insertion. That might be due to the fact that the objects ($< myvector >$) I'm inserting are not that big, so the re-allocations for the bucket are not as costly. If the objects were bigger then definitely the bucket should be changed to a list in order to take advantage of $O(1)$ insert time and avoid re-allocating the whole bucket, you can to that in Shared/HashTable.hpp line:10

Metrics

All metrics (euclidean,cosine) inherit from abstract class Metric. That way we can write abstract code independent of the actual metric used by the

program. The most important member function is Metric::Hash(). It always returns an int that hashes to a bucket in the hashtable.
But metric can also be used to call vectorDistance, which takes as parameters two vectors and return their distance based on the metric (euclidean distance for euclidean, cosine distance for cosine)

LSH Euclidean vs Hypercube Euclidean

$gfunc$: concat of all $hfunc$

When searching for a q with LSH Euclidean we must make sure that we only compare vectors with the same $gfunc$ (because we reduce the $gfunc$ in order to hash it to the table there is a chance 2 different $g's$ will end up in the same bucket). That is done by filtering the bucket and returning only those vectors that have the same $gfunc$.

Hypercube Euclidean directly returns a $k - bits$ bitstring that hashes to the table and therefore there is no need to reduce the value or check for same $gfunc$.
But, there is the issue of how to map $hfunc$ to a bit. One solution would be to use $h_i mod 2$ but that introduces a bias regarding even/odd h_i. So the solution I used was the following: Every time a new $hfunc$ value is found get a random bit for it (uniform distribution), and save that to to a map (key:h_i,$rand_bit$). The next time the same h_i is produced we'll look it up in the map and hash according to the mapped value.

LSH vs HypercubeLSH comparison

Comparing for LSH with default parameters for Euclidean metric.
Dataset is $siftsmall/input$ and queryset is $siftsmall/query$.
For this dataset w=300.
Approximated distance ratio LSH/True 2.16518

LSH:
Creating $n/4 = 10000/4 = 2500$ buckets for this input seems suboptimal as the large w value reduces the vectors $hfunc$ and creates clusters of hash values. As a result many buckets are left empty and there is a huge portion of memory unused.

Creating less buckets $i.en/16$ helps with this problem or having a bigger value for K would spread out the vectors in the table.

Hypercube:
Here the memory utilization is much more efficient. The number of *probes* directly affects the $distance - ratio$ but the cost reflect on the search time. In order to reduce search time you should increase K, that way there are less vectors per bucket and therefore less comparisons. Incrasing K by 1 will halve the num of vectors per bucket. Increasing K should be mirrored with in increase in *probes* in order to keep the $distance - ratio$ constant:
$k = 7$ - $probes = 5$
$k = 7$ - $probes = 10$
$k = 9$ - $probes = 20$
$k = 11$ - $probes = 75$
Now all these are biased because of the set, but the general case seems to be that: When doubling the buckets ($newK = K + 1$) the probes needed to keep the $dist - ratio$ constant are (in the worse case) less than double. That leads to the conclusion that we should strive for as big K as possible to reduce comparisons per probe and therefore time, but not too big because of memory and because of probe overhead.

Distance ratio parameterization

Distance approximation succeded for: LSH with default parameters
Hypercube with $K = 7$,$probes = 10$

Mean times

(default parameters, small dataset)
LSH: 0.0143389 sec
Hypercube: 0.0155365 sec

Memory Usage

Not taking into account the list used to load the vectors from the file. Explained its not necessary in "Design details"

Assumed:
-coords are double (8bytes)
-data set size 10000 vectors, dimension 128

Sizeof() for every class in bytes:
myvector: 56
HashTable: 32
Bucket: 24
Metric: 40
Euclidean: 152
Cosine: 72

LSH

Default L=5.
L hashtables with n/4 buckets each.
Each vector is of dimension 128.
Size without taking into account the metric type:

$$5 * (32 + 40 + (10000/4) * 24 + 10000 * (56 + 128 * 8)) = 54300360 bytes$$

if you want to include metric types size then add 5*152 for euclidean. For cosine add 5*72 and num of buckets is 2^K.

Hypercube

One hashtable with 2^K buckets.
K=7.

$$32 + 40 + 2^7 * 24 + 10000 * (56 + 128 * 8) = 10803144 bytes$$