ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών

Project

Μέρος 1ο

Όνομα
Κυλάφη Χριστίνα-Θεανώ

ΑΜ
1115201200077

Αθήνα, 2018

Η εργασία έχει υλοποιηθεί σε **γλώσσα C**, στο πρόγραμμα " **sublime text** " ενώ παράλληλα δοκιμαζόταν σε περιβάλλον **linux** σε Virtual Machine(**VirtualBox**).

Στο φάκελο, περιλαμβάνονται τα εξής αρχεία εκτός από το **readme** :
makefile, **lsh.c**, **lsh.h**, **cube.c**, **cubefuns.c**, **cubefuns.h**, **hash.c**, **hash.h**, **structfuns.c**, **structs.h**, **extras.c**, **extras.h**, **defs.h**, **final_input_small**, **final_query_small**, **output** .

Με την εντολή "**make**" παράγονται τα εκτελέσιμα "**lsh**" και "**cube**".

Η σύνταξη για την εκτέλεσή των παραπάνω, είναι:
./lsh –d <input_points_file> **–q** <query_points_file> **–k** <# of h functions> **-L** <# of hashtables> **-o** <output _file>

./cube –d <input_points_file> **–q** <query_points_file> **–k** <dimension> **-M** <max # of points to check> **-probes** <max # of vertices to check> **-o** <output_file>

(Έχω προσθέσει **ελέγχους** για μη ορθή σύνταξη - εαν δοθούν λάθος - εκτός ορίων τιμές, το πρόγραμμα εκτελείται κανονικά, με **default** τιμές - και δυνατότητα **επανεκτέλεσης** του εκάστοτε αλγορίθμου, μέχρι ο χρήστης να πληκτρολογήσει " **exit** ").

> Λίγες λεπτομέρειες για τα αρχεία:
—lsh.c: Το αρχείο που καλεί διαδοχικά όλες τις απαραίτητες συναρτήσεις ώστε να δημιουργηθούν οι δομές και να εκτελεστεί ο αλγόριθμος **LSH**
—lsh.h: Όλες οι βασικές επικεφαλίδες που χρειάζονται.
—cube.c: Αντίστοιχα με το lsh.c, το αρχείο αυτό, χειρίζεται τις συναρτήσεις για τη δημιουργία δομών και γενικά τις διαδικασίες ώστε να "έχει στο τέλος χτιστεί" βήμα βημα ο αλγόριθμος **HYPERCUBE**
—cubefuns.c - cubefuns.h: Συναρτήσεις και δομές σχετικές με τον αλγόριθμο του υπερκύβου (δημιουργία, εκτέλεση αλγορίθμου, εκτύπωση κύβου για έλεγχο, κλπ)
—hash.c - hash.h: Τα structs / δομές/ συναρτήσεις που χτίζουν τον αλγόριθμο **LSH**.
—structfuns.c - structs.h: Οι συναρτήσεις και η δομές που σχετίζονται με τα points και την αποθήκευσή τους
—extras.c: Οι βοηθητικές συναρτήσεις (**readfilemode( )** - διαβάζει από το input file και αποθηκεύει τα points στη δομή, **my_log( )** - log με βάση που θέλουμε κάθε φορά, **str_isdigit( )** - ελέγχει ολόκληρη συμβολοσειρά εαν είναι αριθμός)
—defs.h: Τα defines που χρειάζονται οι αλγόριθμοι (d: dimension (των vectors που δίνονται στο input) και w: μεταβλητή που χρησιμοποιείται για τη δημιουργία των h functions για την ευκλείδεια μετρική και στους δύο αλγορίθμους)

Το πρόγραμμα έχει ως εξής:
> Έχουν παραχθεί δύο εκτελέσιμα, το **lsh** και **cube**.
LSH: Χρησιμοποιώντας τον αλγόριθμο **LSH**, βρίσκει τον **προσεγγιστικά κοντινότερο γείτονα** (όταν είναι επιτυχημένος ο αλγόριθμος) και όλους εκείνους τους R - γείτονες με range <= δοσμένου radius, εαν είναι πάνω απο 0.0 .
CUBE: Πρόκειται για τον Hamming Cube αλγόριθμο, ο οποίος όπως και ο παραπάνω, ανάλογα με τις μετρικές και τα ζητούμενα, παράγει ό,τι και ο LSH, έχοντας όμως δεσμεύσει διαφορετικό μέρος μνήμης και έχοντας ακολουθήσει διαφορετικές τεχνικές και λογική.

> Παρακάτω αναλύονται οι **αλγόριθμοι** (υλοποίηση - συναρτήσεις - διαδικασίες) και παρουσιάζονται κάποιες ενδεικτικές εκτελέσεις.

> **Κύριες δομές :**

—> List of Input Points (struct multipoints* Mpointlist) στην οποία αποθηκεύονται όλα τα points που διαβάζουμε από το input_points_file που έχει επιλέξει ο χρήστης. Το διάβασμα και η αποθήκευσή τους, γίνεται από τη συνάρτηση **readfilemode( )** (στο αρχείο **extras.c**), η οποία αποθηκεύει τις συντεταγμένες (μία μία με τη συνάρτηση **multipoint_insertcoord( )**) του κάθε σημείου στον αντίστοιχο κόμβο (κάθε σημείο το εισάγει στη λίστα με την **multipoint_insertpoint( )**), το όνομά του και τον επόμενό του (σε προηγούμενο στάδιο υλοποίησης, είχα και ένα id κάθε σημείου- σε περίπτωση που έπρεπε να το δημιουργήσω εγώ εσωτερικά στο πρόγραμμα).

—> Array of HashTables (L in size - L hashtables στο πλήθος - struct bucket *hashtablesarray[L])(LSH), το οποίο υλοποιείται από ένα 2-διάστατο πίνακα από κελιά τύπου **struct bucket** . Το κάθε hashtable, έχει **tablesize** κελιά, το καθένα από τα οποία έχει αρχικό point (**firstpoint**), τελικό (**lastpoint** - αξιοποιείται στη δημιουργία της δομής) και έναν **int** που δηλώνει πόσα σημεία έχουν πέσει στο συγκεκριμένο **bucket**. Επίσης, επειδή χρειάζεται στη συνέχεια του αλγορίθμου **LSH**, αφού στη συνέχεια δημιουργήσουμε για κάθε point ένα διάνυσμα g, το αποθηκεύουμε και αυτό στη δομή **hashtablesarray** (**char* gfunvector**), για τις συγκρίσεις που θ' ακολουθήσουν.

—> Array of the H (L * k στο πλήθος) family functions (struct hfun* hfarray[ ][ ]) η οποία δομή, περιέχει το **διάνυσμα v** και την **πραγματική float τιμή t** κάθε h, για κάθε G (επιλέγονται όπως ακριβώς ζητείται στην εκφώνηση, με τη συνάρτηση **create_hfun( )**), για τη δημιουργία αργότερα των hi(p) για τη δημιουργία της αντίστοιχης G hashfunction του κάθε hashtable.

—> Cube (with dimension: log2(total_input_points) if not given as argument) που αποτελεί την αναπαράσταση του d'-dimensional hypercube του αλγορίθμου **HYPERCUBE**. Πρόκειται για **dimension** * **struct bucket*** κελιά με την ίδια δομή όπως ακριβώς χτίζεται και το κάθε HashTable στον αλγόριθμο **LSH**(εδώ L = 1). Σε αυτόν, αποθηκεύονται οι δείκτες των σημείων που γίνονται hash σύμφωνα με τη δεκαδική αναπαράσταση της G που έχει σχηματιστεί για το κάθε σημείο (όπως στη μετρική cosine του **LSH**). Εαν έχω euclidean μετρική, τότε περνώ τις G από μία διαδικασία **mapping** κάθε τιμής hi(p) με ομοιόμορφη κατανομή στο {0,1}(τυχαία ρίψη νομίσματος - coin toss). Αυτά τα mappings τα αποθηκεύω σε λίστες, οι οποίες πρέπει να έχουν για την ίδια τιμή του hi(p) ίδιο αποτέλεσμα mapping - να αποτελεί συνάρτηση(αν παράδειγμα στην κάθε hi έχω πολλές τιμές = 9 , κάθε φορά πρέπει να γίνεται map ή με 1 ή με 0 - το πρώτο που θα φέρει το coin toss για το 9). Τελικά, παράγω ένα g διάνυσμα με συντεταγμένες στο {0,1}. Από αυτό το σημείο και μετά στην ευκλείδεια ή αν έχω cosine μετρική, η διαδικασία hashing στον cube, είναι η ίδια με του LSH. Παίρνω το διάνυσμα g που έχω παράξει, βρίσκω τη δεκαδική αναπαράστασή του και τοποθετώ το στοιχείο στο αντίστοιχο bucket (δείκτη σε αυτό το στοιχείο).

—> List of R-Neighbours (struct simple_list) είναι μια απλή λίστα από γείτονες εντός ακτίνας, για την εκτύπωσή τους χωρίς διπλότυπα, σε περίπτωση που υπάρχουν ίδια points μέσα σε buckets των διαφορετικών **hashTables** - **LSH** / **γειτονικών κορυφών** - **CUBE** .

—> List of mappings (struct h_to_fmap) που κρατάει τα mappings που ανέφερα πιο πάνω, από **hi(p) -> { 0, 1 }** . Κάθε φορά που διατρέχω τις συντεταγμένες ενός σημείου στην ευκλείδεια μετρική, αφού δημιουργήσω την hi(x), όπου x συντεταγμένη του p, ελέγχω αν αυτή η τιμή υπάρχει στη λίστα με κάποιο mapping σε 0 ή 1, αλλιώς προσθέτω έναν κόμβο τύπου **struct h_to_fmap** και αντιστοιχίζω τη συγκεκριμένη τιμή με 0 ή 1, με τυχαίο τρόπο (coin flip - uniform distribution).

> **Συναρτήσεις :**

—> readfilemode( ) : Διαβάζει από το input αρχείο που επέλεξε ο χρήστης, τα input points, και τα αποθηκεύει στη δομή **struct multipoints* Mpointlist**
—> struct functions : Συναρτήσεις για τη σταδιακή αποθήκευση των points στη δομή
—> my_log( ) : Λογάριθμος με customised βάση κάθε φορά(εδώ χρειάστηκε η βάση 2)
—> str_digit( ) : Ελέγχει αν ένα ολόκληρο string είναι αριθμός (η απλή isdigit( ) ελέγχει χαρακτήρα- χαρακτήρα)
—> create_vector( ) : Δημιουργεί τυχαία (με κανονική κατανομή οι τυχαίες μεταβλητές) το διάνυσμα v για κάθε συνάρτηση h που θα χρειάζεται ο αλγόριθμος **LSH**
—> create_hfun( ) : Επιλέγει τυχαία vector και t , και τ' αποθηκεύει στην αντίστοιχη θέση του πίνακα των h functions που έχω δημιουργήσει για τη συνέχεια του αλγορίθμου όπου και θα αξιοποιηθούν. Δημιουργεί τελικά k * L h functions της οικογένειας H που έχουμε στην εκφώνηση
—> h( ) : Δέχεται ένα point και τις κατάλληλες δομές και τιμες, ώστε να παράξει τελικά το hi(p) (ανάλογα με τη μετρική - euclidean/cosine, εκτελεί την κατάλληλη συνάρτηση)

—> hashing( ) : Εδώ αξιοποιούνται τα παραπάνω ώστε να γίνει το hashing των input points σε κάθε hashtable. Κάθε G παράγει για κάθε point ένα διάνυσμα, το οποίο στη συνέχεια μετασχηματίζεται με τις κατάλληλες μαθηματικές πράξεις σε τιμή μεταξύ [0 … tablesize-1], με την οποία αντιστοιχίζεται σε κάποιο bucket του αντίστοιχου πίνακα. Για να γίνουν όμως αργότερα οι συγκρίσεις μόνο με τα στοιχεία που μας ενδιαφέρουν και όχι με όλα εκείνα που έπεσαν στο ίδιο bucket στην **ευκλείδεια μετρική**, κρατάμε και τα διανύσματα G που είχαμε δημιουργήσει στο προηγούμενο βήμα του hashing, και το αποθηκεύουμε μέσα στη δομή που κρατάμε το point που μόλις τοποθετήθηκε στον πίνακα (**struct points_hashed** -> **gfunvector**).
—> true_dist( ) : Εξαντλητική αναζήτηση κοντινότερου γείτονα (βρίσκει τον πραγματικά κοντινότερο γείτονα και επιστρέφει την απόστασή του από το σημείο για το οποίο αναζητούμε NN - Nearest Neighbour)
—> cos_v( ) : Συνάρτηση δοσμένη για την εύρεση της απόστασης στην cosine μετρική
—> LSH_algo( ) : Η ουσία του αλγορίθμου **LSH**. Διαβάζει ένα ένα τα query points από το αρχείο που επιλέγει ο χρήστης, και κάθε ένα, το περνά από τις ίδιες hash functions που πέρασε και όλα τα input points. Αυτό, θα έχει ως αποτέλεσμα να βρεθεί το κάθε query point σε L buckets (ένα για κάθε hashtable). Σε αυτά τα buckets, θα παίρνει ένα ένα κάθε στοιχείο που βρίσκεται στο αντίστοιχο bucket κάθε φορά και θα ελέγχει εαν το διάνυσμα G του σημείου είναι ίδιο με το G του query point του οποίου ψάχνουμε τον **NN** (στην ευκλείδεια μετρική μόνο). Εαν είναι ίδιο, τότε μόνο μπαίνει στη διαδικασία σύγκρισης των 2 σημείων με την ευκλείδεια απόσταση. Για την cosine, συγκρίνει όλα τα σημεία που βρίσκονται μέσα στο συγκεκριμένο bucket με το query point. Σε κάθε σύγκριση, γίνεται έλεγχος για **R-γείτονες**(εαν το **radius > 0.0**) και σύγκριση με το μέχρι τότε min distance, ώστε να κρατάμε πάντα τη μικρότερη απόσταση και το όνομα του σημείου αυτής. Τελικά, αφού εκτελέσουμε όλες τις συγκρίσεις με τα κατάλληλα στοιχεία του συγκεκριμένου bucket κάθε φορά(L φορές), εκτελούμε εξαντλητική αναζήτηση για να βρούμε τον πραγματικά κοντινότερο γείτονα(με τη συνάρτηση **true_dist( )**) . Παράλληλα, κάνω και τις μετρήσεις χρόνου εκτέλεσης των 2 αναζητήσεων(**προσεγγιστικά** / **πραγματικά** κοντινότερου γείτονα), ώστε να τους τυπώσω στο output_file όπως ζητείται και να δημιουργήσω τα **στατιστικά** του αλγορίθμου στο τέλος(εκτύπωση στο terminal - stdout).

—> ham_dist_neighbour( ) : Επιστρέφει τον επόμενο γείτονα σε απόσταση 1 ή 2 (δίνεται σαν όρισμα), αλλάζοντας κάθε φορά τις θέσεις του bitstring που δείχνουν οι δείκτες που περνώ στη συνάρτηση. Εαν έχω απόσταση 1, χρησιμοποιώ έναν δείκτη που ξεκινά από τη θέση 0 και φτάνει στη θέση k-1, ενώ αν έχω απόσταση 2, τότε έχω 2 δείκτες που ο πρώτος ξεκινά από το 0 και φτάνει k-2 και ο δεύτερος σε κάθε επανάληψη ξεκινάει από μία θέση μετά τον πρώτο δείκτη και φτάνει μέχρι k-1 θέση του string. Για την υλοποίηση αυτών των αλλαγών bit και μετατροπή σε

δεκαδικό, ώστε να επιστραφεί η δεκαδική τιμή του γείτονα, χρησιμοποιείται πίνακας k- διάστασης για κάθε γείτονα, στον οποίο αποθηκεύεται η δυαδική αναπαράσταση του αρχικού bucket, στο οποίο εκτελούμαι τις αλλαγές που περιγράφηκαν παραπάνω για να βρούμε τους γείτονές του στην αντίστοιχη απόσταση.
—> h_cube( ) : Πρόκειται για τη δημιουργία της G function του Cube(στην ευκλείδεια μόνο μετρική). Έχοντας δημιουργήσει k h συναρτήσεις, κάνω hash την κάθε συντεταγμένη του κάθε point με την κλασική **h( )** όπως στον **LSH** αλγόριθμο και στη συνέχεια, περνάω αυτό το **hi(p)** από την **h_cube( )** ώστε να κάνω mapping του συγκεκριμένου value του **hi(p)** στο {0,1} ομοιόμορφα, όπως αναφέρω παραπάνω, όταν αναλύω τη δομή **Cube**. Τα mappings αυτά, τα κρατάω σε λίστα, ώστε κάθε hi, να έχει τα ίδια mappings για τις ίδες values (να αποτελεί συνάρτηση).
—> cube_create( ) : Δημιουργία του κύβου διάστασης **k** (**2^k** κορυφές / **buckets**) , όπου **k** δίνεται σαν όρισμα, αλλιώς παίρνει τιμή ίση με **log2(total_input_points)**. Όπως και στον **LSH**, έτσι κι εδώ για κάθε input point δημιουργούμε την g του (**g = [h1(p) h2(p) h3(p) … hk(p)]**). Εαν πρόκειται για ευκλείδεια μετρική, την περνάμε από την **h_cube( )** ώστε να καταλήξει να έχει συντεταγμένες 0 ή 1, αλλίως (cosine μετρική) μένει ως έχει. Έπειτα, όπως και στην cosine μετρική του **LSH**, με βάση τη δεκαδική αναπαράσταση του αριθμού που σχηματίζει η g του κάθε σημείου, τοποθετούμε τα points στα buckets του **cube** (στη δομή που περιγράφω παραπάνω).
—> hypercube_algo( ) : Αυτός είναι ο **Hamming Cube αλγόριθμος**. Για κάθε query point από το query_file που έχει δώσει ο χρήστης, κάνω ακριβώς την ίδια διαδικασία όπως στον **LSH cosine** ή αν έχω **ευκλείδεια** μετρική, όπως και παραπάνω στο hashing, έτσι κι εδώ, δημιουργώ μια g την οποία θα μετατρέψω σε **“ f “**, κάνοντας **mapping** το κάθε **hi -> { 0, 1 }** αμέσως μετά. Έτσι και πάλι, παίρνω τη δεκαδική αναπαράσταση του αριθμού που σχηματίζει το διάνυσμα **f** (mapped g). Πηγαίνω στο αντίστοιχο bucket, εκτελώ τους γνωστούς ελέγχους απόστασης ανάλογα με τη μετρική και τυπώνω/αποθηκεύω τα αντίστοιχα αποτελέσματα για χρήση και παραγωγή των στατιστικών αργότερα. Εδώ, έχουμε επίσης τα **probes** και τα **M** που πρέπει να ελέγχουμε . Συγκεκριμένα, τα probes δηλώνουν το μέγιστο πλήθος των κορυφών που θα ελεγχθούν και το M το μέγιστο πλήθος points. Έχοντας λοιπόν εξετάσει το αρχικό bucket/κορυφή του κύβου, επεκτείνεται η αναζήτηση στους γείτονες **μέχρι και hamming_distance = 2** (γίνεται έλεγχος όταν δίνονται οι τιμές του probes και k , ώστε εαν το hamming_distance ξεπερνά την τιμή 2, να εκτελείται το πρόγραμμα με τις default τιμές που δίνονται στην εκφώνηση). Αυτό που κάνει η συνάρτηση είναι να ελέγχει εαν οι κορυφές που έχω ήδη επισκεφθεί είναι λιγότερες από probes, ώστε να συνεχίζει να ψάχνει στους γείτονες, ή να τερματίσει την αναζήτηση για το συγκεκριμένο point και να προχωρήσει στο επόμενο(αντίστοιχα με το M). Όσο για την απόσταση και την εύρεση γειτόνων, έχω αναλύσει πώς γίνεται η διαδικασία, με τη χρήση της συνάρτησης **ham_dist_neighbour( )** πιο πάνω. Στην αναζήτηση γειτόνων, έχω προσθέσει και μια επιπλέον συνθήκη, η μεταβλητή **points_hashed** του γείτονα που θα επιλεγεί να είναι **θετική**, ώστε να αυξάνω την πιθανότητα να βρει τον κοντινότερο γείτονα με καλύτερα στατιστικά, **αποφεύγοντας** γείτονες με **μηδενικό πλήθος hashed_points**.

—> print functions : Οικογένεια συναρτήσεων που εκτυπώνουν κάποιες δομές (για έλεγχο τιμών κατά τη διάρκεια της ανάπτυξης).

—> delete functions : Οικογένεια συναρτήσεων για απελευθέρωση μνήμης (διαγραφή δομών που χρησιμοποιήθηκαν κατά τη διάρκεια εκτέλεσης του εκάστοτε αλγορίθμου).

—> calculate_memory_cube/LSH( ) : Μέτρηση της μνήμης που χρησιμοποιούν οι κυριότερες και μεγαλύτερες δομές σε κάθε αλγόριθμο / εκτελέσιμο.

> Παρατηρήσεις :

—> Έχουμε **δύο αλγορίθμους και δύο μετρικές**. Ο αλγόριθμος **LSH** στην **euclidean** μετρική, αφού περάσει το query point από την εκάστοτε hash function, χρησιμοποιεί ευκλείδεια απόσταση μεταξύ των σημείων που βρίσκονται στο ίδιο bucket, όμως όχι με όλα, αλλά μόνο με εκείνα των οποίων τα διανύσματα G είναι ιδια με του query point.
Στην **cosine** μετρική στον LSH, οι h(p) βγάζουν αποτέλεσμα στο {0,1}, οπότε η θέση τους στον hashtable θα είναι η δεκαδική αναπαράσταση αυτού του bitstring G (τυχαίος συνδυασμός των hi(p)).
Στον **hypercube**, αντίστοιχα, όταν έχουμε **cosine** μετρική, δε χρειάζεται να κάνουμε κάποιο μετασχηματισμό, γιατί ο **LSH** παράγει όπως είδαμε h στο {0,1}, άρα προχωράμε κατευθείαν σε hashing στις κορυφές του κύβου, πλήθους 2^d' (cube ~ hashtable) , ενώ στην **euclidean**, έχουμε να περάσουμε πρώτα τις G από τυχαίο , ομοιόμορφο hashing στο {0,1} (για κάθε hi, κρατάω τις τιμές που προκύπτουν από ίδιες τιμές h(p) σε λίστα) .

```
Algorithm: LSH

Metric: Euclidean

W: 400

# of Input Points: 10000

# of Query Points: 99

Dimension of points: 128

# of HashTables( size:2500 ): 8

Size of H function Family: 7

Max Approximation: 2.3567

Mean LSH time(of 99 points): 3.4820msec



~ Memory allocated: 7818343 bytes
```

Άρα παρατηρούμε ότι όπως είχαμε k πληθος h functions στον LSH, αντίστοιχα έχουμε d' = k διάσταση υπερκύβου στον hypercube και όμοια, όπως είχαμε L hashtables στον LSH, μεταβλητού πλήθους, εδώ στον υπερκύβο, έχουμε L = 1.

—> Εαν η **Radius** τιμή στο query_file είναι > 0.0, τότε υλοποιείται και **range search** με αυτή την τιμή radius, ώστε εαν βρίσκονται μέσα σε αυτό το range να τυπώνει στο output_file τα ονόματα των points αυτών (σε κάθε bucket, όσα points αποτελούν R-neighbours, τα εισάγω σε μια λίστα, ελέγχοντας παράλληλα αν ήδη υπάρχουν σε αυτήν, δηλαδή εαν ήδη έχω βρει αυτό τον R-γείτονα σε άλλο bucket / HashTable).

Ακολουθούν συγκρίσεις μεταξύ **LSH** - **HYPERCUBE** πάνω στο **final_input_small** input dataset και **final_query_small** query dataset (siftsmall με προσθήκη id's στην πρώτη στήλη), για την **euclidean** μετρική.

- LSH -

Parametres:
W: 400
k: 7
L: 8
Tablesize: 2500

Results:
> **Max Approximation:** 2.3567

> **Mean LSH time:** 3.4820 msec

> **~ Memory Allocated:** 7,818,343 bytes

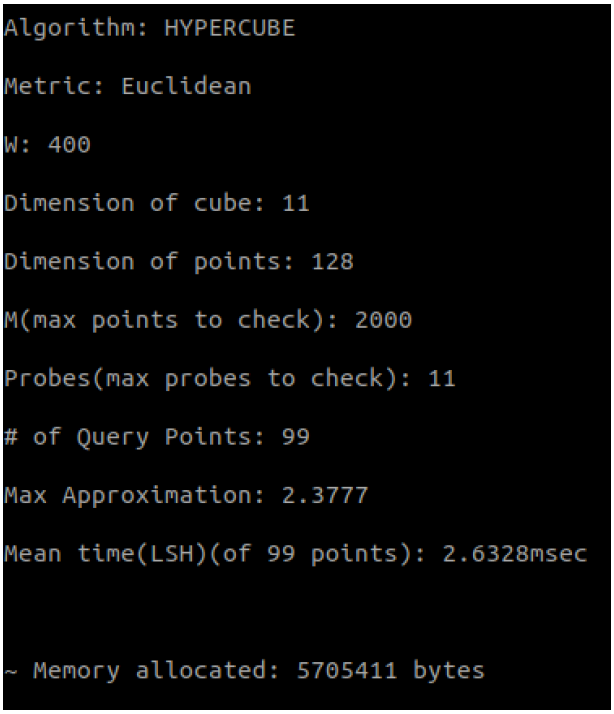
- HYPERCUBE -

Parametres:
W: 400
d'(= k): 11
Max probes(probes): 11
Max points(M): 2000

Results:
> **Max Approximation:** 2.3777

> **Mean LSH time:** 2.6328 msec

> **~ Memory Allocated:** 5,705,411 bytes



Παρατηρούμε πως για να πετύχουμε το **ίδιο max approximation** στην ευκλείδεια μετρική, έχουμε χρήση **μικρότερου μέρους της μνήμης** από τον αλγόριθμο **HYPERCUBE** , όπως και **μικρότερο μέσο χρόνο αναζήτησης** του προσεγγιστικά κοντινότερου γείτονα. Παράλληλα, μεγαλώσαμε

αρκετά τη διάσταση σε 11 , τις κορυφές προς εξέταση σε 11 και τα σημεία προς εξέταση σε 2000, αυξάνοντας έτσι το εύρος της αναζήτησης στον κύβο.
Ο **LSH**, χρησιμοποιεί **περισσότερη μνήμη** για τις δομές του, και για το συγκεκριμένο max approximation έχει **μεγαλύτερο μέσο χρόνο αναζήτησης** κοντινότερου γείτονα, με 8 HashTables, διάστασης 7.

Το **window**(**W**) που χρησιμοποιήθηκε για την οικογένεια **H functions**, ήταν **400**.