



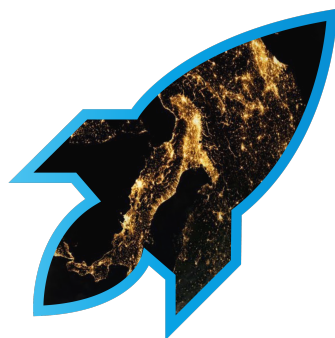
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
— ΙΔΡΥΘΕΝ ΤΟ 1837 —

ΑΝΑΠΤΥΞΗ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ ΑΛΓΟΡΙΘΜΙΚΑ ΠΡΟΒΛΗΜΑΤΑ

ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ 2020

1η ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΝΑΖΗΤΗΣΗ ΚΑΙ ΣΥΣΤΑΔΟΠΟΙΗΣΗ ΔΙΑΝΥΣΜΑΤΩΝ ΣΤΗ C/C++



Αριθμός Μητρώου(ΑΜ):

1115201700217

1115201700203

Ονοματεπώνυμο:

Ορέστης ΣΤΕΦΑΝΟΥ

Λεωνίδας ΕΦΡΑΙΜ

ΑΚΑΔΗΜΑΪΚΗ ΧΡΟΝΙΑ 2020-2021

ΠΕΡΙΕΧΟΜΕΝΑ

1	ΕΙΣΑΓΩΓΗ	3
2	ΜΕΤΑΓΛΩΤΤΙΣΗ-ΕΚΤΕΛΕΣΗ	4
3	ΥΛΟΠΟΙΗΣΗ	5
3.1	ΕΙΣΟΔΟΣ ΔΕΔΟΜΕΝΩΝ	5
3.2	ΜΕΤΡΙΚΕΣ	6
3.3	HASH TABLE	6
3.4	LSH	7
3.5	HYPER CUBE	9
3.6	CLUSTERING	11
3.6.1	Lloyd's (A)	11
3.6.2	LSH Range Search (B)	11
3.6.3	ΤΥΧΑΙΑ ΠΡΟΒΟΛΗ (Γ)	12
3.6.4	Silhouette	12

ΕΙΣΑΓΩΓΗ

Στο πλαίσιο αυτής της εργασίας, είχαμε να υλοποιήσουμε τον αλγόριθμο LSH για διανύσματα στον D -διάστατο χώρο, καθώς και τον αλγόριθμο τυχαίας προβολής στον υπερκύβο βάσει της μετρικής Μανχάταν $L1$. Στη συνέχεια έπρεπε να εκτελέσουμε κάποια queries στο dataset που μας δόθηκε, έτσι ώστε να επαληθεύσουμε τη σωστή λειτουργία των αλγορίθμων. Τέλος, κληθήκαμε να υλοποιήσουμε τους αλγόριθμους για τη συσταδοποίηση διανυσμάτων βάσει της μετρικής Μανχάταν όπου η ανάθεση θα έπρεπε να γίνει με τον αλγόριθμο του Lloyd's ή με αντίστροφη ανάθεση μέσω Range Search με LSH. Η υλοποίηση της εργασίας έχει γίνει σε C++

ΜΕΤΑΓΛΩΤΤΙΣΗ-ΕΚΤΕΛΕΣΗ

Για τις ανάγκες της εργασίας δημιουργήσαμε τρεις main συναρτήσεις, όπου οι δύο είναι υπεύθυνες για τους αλγόριθμους LSH και Hypercube, ενώ η τρίτη είναι υπεύθυνη για το Clustering

Η μεταγλώττιση γίνεται με τις παρακάτω εντολές

- **make lsh**
- **make cube**
- **make cluster**

Ενώ η εκτέλεση των προγραμμάτων γίνεται με τις εντολές που μας δόθηκαν στην εκφώνηση της εργασίας, δηλαδή:

- **LSH**

```
./lsh -d <input file> -q <query file> -k <int> -L <int> -o <output file> -  
N<number of nearest> -R <radius>
```

- **HYPER CUBE**

```
./cube -d <input file> -q <query file> -k <int> -M <int> -probes <int>  
-o<output file> -N <number of nearest> -R <radius>
```

- **CLUSTERING**

```
./cluster -i <input file> -c <configuration file> -o <output file> -complete  
<optional> -m <method: Classic OR LSH or Hypercube>
```

ΥΛΟΠΟΙΗΣΗ

3.1 ΕΙΣΟΔΟΣ ΔΕΔΟΜΕΝΩΝ

Για την εισαγωγή των δεδομένων έχουμε δημιουργήσει μια συνάρτηση με το όνομα **ReadData** η οποία δέχεται σαν όρισμα το path με το αρχείο εικόνων και ένα vector όπου στη συνέχεια το γεμίζει με τις εικόνες.

Η συνάρτηση αφού ανοίξει το αρχείο διαβάζει διαδοχικά 4 integers όπου αντιπροσωπεύουν αντίστοιχα

- Το magic number
- Το ύψος της εικόνας
- Το πλάτος της εικόνας
- Τον αριθμό των εικόνων που υπάρχουν στο αρχείο

Αφού ξέρουμε τις διαστάσεις των εικόνων, τώρα μπορούμε να διαβάζουμε $N*N$ chars και να τους αποθηκεύουμε σε μια γραμμή του vector διαδοχικά.

Για την υλοποίηση της συνάρτησης ReadData χρειαστήκαμε να υλοποιήσουμε ακόμα μια συνάρτηση με το όνομα **NumReverse** η οποία παίρνει ένα integer και του αλλάζει το endian του με μερικά shifts γιατί ο αριθμός που υπάρχει στο αρχείο είναι ανάποδα οπότε πρέπει να αντιστραφεί.

3.2 ΜΕΤΡΙΚΕΣ

Για τις μετρικές δημιουργήσαμε μια κλάση με το όνομα **Metrics** η οποία έχει μια συνάρτηση με το όνομα **get_distance** η οποία δέχεται σαν όρισμά της δύο εικόνες που θέλουμε να βρούμε τις αποστάσεις τους, καθώς και ακόμα ένα όρισμα το οποίο είναι το όνομα της μετρικής π.χ. L1 για την μετρική Μανχάταν. Εδώ μπορούν να υλοποιηθούν και άλλες μετρικές αλλά στην εργασία μάς ζητήθηκε μόνο η μετρική Μανχάταν.

Για την υλοποίηση της Μανχάταν μετρικής πήραμε το άθροισμα της απόλυτης τιμής των σημείων των δύο εικόνων

3.3 HASH TABLE

Για την υλοποίηση του Lsh χρειαστήκαμε ένα hashtable οπότε δημιουργήσαμε μια κλάση με το όνομα hashtable. Η κλάση αυτή αποτελείται από τα buckets που είναι ένας πίνακας με vectors, το μέγεθος του πίνακα, τις σταθερές K και W, μια μεταβλητή sRandInit η οποία αρχικοποιεί την rand για να έχουμε τυχαία s κάθε φορά καθώς και ένα vector με vectors το οποίο περιέχει τα s. Στον **constructor** της hashtable αρχικοποιούμε όλες τις μεταβλητές καθώς και δημιουργούμε τα τυχαία s όπου τα βάζουμε στο vector. Εκτός από τον constructor το hashtable έχει και τις παρακάτω συναρτήσεις

- **hash_function**

Η συνάρτηση αυτή δημιουργεί τη συνάρτηση $g(p)$ σύμφωνα με τον αλγόριθμο lsh. Αυτό το κάνει φτιάχνοντας μια διαφορετική $h(p)$ κάθε φορά βάσει του παρακάτω τύπου

$$h(p) = a_{d-1} + m \cdot a_{d-2} + \dots + m^{d-1} \cdot a_0 \bmod M \in \mathbb{N},$$

Στη συνέχεια ενώνει όλες τις $h(p)$ για να δημιουργήσει την $g(p)$

$$g(p) = [h_1(p)|h_2(p)|\cdots|h_k(p)] \in \mathbb{N}.$$

- **insert**

Η συνάρτηση αυτή δέχεται ως όρισμα μια εικόνα καθώς και τον αριθμό του bucket που πρέπει να μπει με σκόπο να εισαγάγει την εικόνα αυτή στο κατάλληλο bucket του hashtable

- **get_bucket_imgs**

Η συνάρτηση αυτή παίρνει ως όρισμα τον αριθμό κάποιου bucket και επιστρέφει ένα vector με τα στοιχεία αυτού του bucket

3.4 LSH

Ο αλγόριθμος LSH υλοποιείται μέσω μιας κλάσης με το αντίστοιχο όνομα. Η κλάση αυτή περιέχει τις σταθερές K, L, r , ένα hash table και ένα vector με όλα τα δεδομένα των εικόνων. Στον constructor αρχικοποιούνται όλες οι μεταβλητές. Επίσης δημιουργούνται όλα τα hashfunction και μπαίνει η κάθε εικόνα στο bucket που της αντιστοιχεί. Οι συναρτήσεις που υλοποιούνται στην κλάση LSH είναι οι εξής.

- **nearest_neighbor**

Αυτή η συνάρτηση δέχεται ως όρισμα ένα vector με το query και μας επιστρέφει τον ένα pair που περιέχει τον κοντινότερο γείτονα μαζί με την απόσταση που έχει από αυτό τον γείτονα. Η διαδικασία αυτή γίνεται υπολογίζοντας αρχικά το bucket που αντιστοιχεί στο query σε κάθε hashtable και στη συνέχεια παίρνουμε όλα τα στοιχεία που βρίσκονται σε αυτό το bucket στο vector image_indexes. Αφού αποθηκεύσουμε στο img_indexes προσωρινά του κοντινούς γείτονες βρίσκουμε την Μανχάταν απόσταση, μεταξύ αυτών και του query. Τέλος, παίρνουμε την πιο κοντινή απόσταση από όλα και την επιστρέφουμε

- **knn**

Η συνάρτηση αυτή λειτουργεί με παρόμοιο τρόπο με την `nearest_neighbor` με τη μόνη διαφορά αντί να επιστρέψει ένα κοντινό γείτονα επιστρέφει τους K κοντινούς γείτονες. Οπότε εδώ δέχεται ως όρισμα το `query` και το K που μας προσδιορίζει τον αριθμό των γειτόνων που θέλουμε να επιστρέψουμε, με αποτέλεσμα να επιστρέφει ένα `vector` με K pairs που περιέχουν την απόσταση και τον N κοντινότερο γείτονα του `query`

- **range_search**

Η συνάρτηση `range_search` βρίσκει τους γείτονες του `query` απόσταση r . Δέχεται ως όρισμα το `query`, την ακτίνα του κύκλου όπου θα γίνει το `range search` και μια σταθερά c , όπου αν δεν δώσουμε όρισμα, παίρνει default τιμή 1. Στη συνέχεια όπως και οι προηγούμενες συναρτήσεις, έτσι και η `range search` βρίσκει το bucket που αντιστοιχεί στο `query` σε κάθε hashtable και στη συνέχεια παίρνουμε όλα τα στοιχεία που βρίσκονται σε αυτό το bucket στο `vector image_indexes`. Τώρα για κάθε εικόνα ελέγχει βάσει της μετρικής Μανχάταν αν βρίσκεται εντός της ακτίνας r . Σε περίπτωση που βρίσκεται τότε προσθέτει την εικόνα στο `results` έτσι ώστε στο τέλος να τις επιστρέψει

- **exact_nearest_neighbor**

Αυτή η συνάρτηση έχει σκοπό να μας επιστρέψει τους ακριβείς πιο κοντινούς γείτονες για να ελέξουμε ότι τα αποτελέσματα των παραπάνω συναρτήσεων είναι σωστά. Η διαδικασία αυτή γίνεται με τη μέθοδο του `brute force`, δηλαδή ελέγχουμε όλες τις εικόνες του dataset και επιστρέφουμε τις K εικόνες με τη μικρότερη Μανχάταν απόσταση από το `query`. Εδώ η συνάρτηση αυτή παίρνει σαν όρισμα το `query`, το K και επιστρέφει ένα `vector` με pairs όπου το κάθε ζευγάρι αποτελείται από την απόσταση και την εικόνα που είναι πιο κόντα στο `query`

3.5 HYPER CUBE

Για την υλοποίηση του Hyper Cube δημιουργήσαμε μια κλάση με το όνομα `BinaryHyperCube` η οποία έχει σαν σταθερές το `d`, `M`, `probes`, `R` καθώς και τρία `vectors` τα οποία είναι τα δεδομένα των εικόνων, οι τιμές των `s` και μια δομή για τον υπερκύβο. Στον **constructor** του υπερκύβου αρχικοποιούνται όλες οι μεταβλητές και μπαίνουν τα δεδομένα στο `data vector`. Στη συνέχεια δημιουργούνται με τυχαίο τρόπο τα `s` και μπαίνουν στο αντίστοιχο `vector`. Τέλος τα δεδομένα hasharονται και μπαίνουν στο ανάλογο `bucket` της δομής του υπερκύβου. Η κλάση `BinaryHyperCube` υλοποιεί και τις παρακάτω συναρτήσεις.

- **f**

Η συνάρτηση `F` σύμφωνα με τη θεωρία για την υλοποίηση του αλγόριθμου του υπερκύβου πρέπει να επιστρέφει 0 ή 1 με ομοιόμορφη κατανομή. Έτσι λοιπόν αποφασίσαμε η συνάρτηση `f` να δέχεται ένα `integer`, να παίρνει τη δυαδική του μορφή και να μετράει πόσα μηδενικά και πόσους άσσους έχει. Αν οι άσσοι είναι περισσότεροι από τα μηδενικά τότε επιστρέφει 1, αλλιώς επιστρέφει 0.

- **h**

Η συνάρτηση `h` είναι η `hashfunction` που χρησιμοποιά ο υπερκύβος αλλά είναι ίδια με τη συνάρτηση `h(p)` του `Lsh` που αναφέραμε πιο πάνω. Οπότε η υλοποίηση είναι η ίδια

- **get_number_from_bits**

Η συνάρτηση αυτή παίρνει ως όρισμα ένα `vector` και επιστρέφει τον αριθμό των `bits` με σκοπό να γνωρίζουμε τον αριθμό του `bucket` που ζητούμε

- **hamming_distance**

Αυτή η συνάρτηση μετρά την απόσταση `hamming` μεταξύ δύο αριθμών. Δηλαδή βρίσκει τον ελάχιστο αριθμό αντικαταστάσεων που χρειάζονται, ώστε να μετατραπεί

ο ένας αριθμός στον άλλο. Οπότε η συνάρτηση δέχεται δυο integers τους σπάζει σε bits και τους βάζει σε δύο arrays αντίστοιχα. Στη συνέχεια συγκρίνει τα bits και μετρά πόσα είναι διαφορετικά. Τέλος, επιστρέφει την απόσταση hamming των δύο αριθμών

- **knn**

Η συνάρτηση αυτή υπολογίζει τους K κοντινότερους γείτονες μιας εικόνας. Για αυτό τον υπολογισμό χρειάζεται ως όρισμα το query και τον αριθμό K για να επιστρέψει ένα vector με τα K κοντινότερα ζευγάρια απόστασης-εικόνας. Η υλοποίηση είναι παρόμοια με τον knn του Lsh με τη μόνη διαφορά ότι είναι ο υπολογισμός της απόστασης των σημείων που βρίσκονται στο ίδιο bucket, όπου πρώτα υπολογίζονται οι κοντινοί γείτονες βάσει της απόστασης hamming

- **range_search**

Η συνάρτηση range_search βρίσκει τους γείτονες του query βάσει της απόστασης r. Δέχεται ως όρισμα το query, την ακτίνα του κύκλου όπου θα γίνει το range search και μια σταθερά c, όπου αν δεν δώσουμε παίρνει default τιμή 1. Όπως η συνάρτηση knn έτσι και αυτή η συνάρτηση έχει παρόμοια υλοποίηση με το range_search του Lsh με τη μόνη διαφορά ότι προσεγγίζουμε τους γείτονες με την μικρότερη απόσταση hamming

- **exact_nearest_neighbor**

Η συνάρτηση exact_nearest_neighbor έχει ακριβώς την ίδια υλοποίηση με τη συνάρτηση exact_nearest_neighbor της κλάσης Lsh όπου εξηγείται πιο πάνω

- **get_bucket_imgs**

Η συνάρτηση αυτή παίρνει ως όρισμα τον αριθμό κάποιου bucket και επιστρέφει ένα vector με τα στοιχεία αυτού του bucket

3.6 CLUSTERING

Για την υλοποίηση του clustering δημιουργήσαμε μια ξεχωριστή κλάση με το όνομα `Clustering` και μια καινούρια `main`. Σε αυτή την κλάση υλοποιούνται οι συνάρτησεις `loyds`, `lsh`, `hypercube` οι οποίες είναι υπεύθυνες στο βήμα της ανάθεσης για το clustering. Επίσης υλοποιείται η συνάρτηση `silhouette_score` για την αξιολόγηση των αποτελεσμάτων.

3.6.1 LLOYD'S (A)

Το clustering με τον ακριβή αλγόριθμο του Lloyd's υλοποιείται στη συνάρτηση **loyds**. Η συνάρτηση δέχεται ως όρισμα την παράμετρο K που αντιστοιχεί στο πλήθος των clusters. Στη συνέχεια δημιουργεί ένα vector με τα κεντρικά σημεία των clusters (centroids), ένα vector για να κρατά προσωρινά τα νέα κεντρικά σημεία και ένα vector που περιέχει τα K clusters. Τα κεντρικά σημεία αρχικοποιούνται και στη συνέχεια γίνεται ο υπολογισμός των καινούριων κεντρικών σημείων βάσει της μέσης απόστασης όλων των σημείων του cluster. Η διαδικασία αυτή συνεχίζεται μέχρι η απόσταση των νέων κεντρικών σημείων από τα παλιά να είναι πολύ μικρή (καθορίζεται από τη μεταβλητή `centroids_difference`)

3.6.2 LSH RANGE SEARCH (B)

Το clustering με αντίστροφη ανάθεση (reverse) μέσω Range search με `Lsh` γίνεται στη συνάρτηση **lsh** με τη μόνη διαφορά (σε σχέση με τη συνάρτηση `loyds`) στο βήμα της ανάθεσης των σημείων όπου αναθέτουνται με διαφορετικό τρόπο. Η διαφορά είναι ότι αφού έχουν επιλεγεί τα κεντρικά σημεία τότε κάνουμε range search γύρω από το κάθε κεντρικό σημείο και όσα σημεία είναι σε αυτή την ακτίνα, τότε μπαίνουν στο cluster του σημείου. Στη συνέχεια διπλασιάζουμε την ακτίνα μέχρι να καλύψουμε αρκετή επιφάνεια των κοντινών σημείων από το κεντρικό σημείο και συνεχίζουμε την ανανέωση των κεντρικών κανονικά βάσει της απόστασης. Σε περίπτωση που ένα σημείο διεκδικείται από δύο cluster, πηγαίνει στο cluster με την κοντινότερη

Μανχάταν απόσταση. Για την υλοποίηση αυτού του σημείου χρησιμοποιήσαμε ένα vector με δύο στήλες όπου στην πρώτη στήλη βάλαμε ένα flag και στη δεύτερη το cluster στο οποίο ανήκει. Όταν ένα σημείο μπει σε ένα cluster τότε το μαρκάρουμε και σημειώνουμε το cluster που ανήκει, έτσι ώστε να γνωρίζουμε όταν πάει να το διεκδικήσει κάποιο άλλο cluster ότι είναι ήδη καταχωρημένο

3.6.3 ΤΥΧΑΙΑ ΠΡΟΒΟΛΗ (Γ)

Το clustering με τυχαία προβολή υλοποιείται ακριβώς με τον ίδιο τρόπο με την LSH range search με τη μόνη διαφορά ότι στον αλγόριθμο ανάθεσης χρησιμοποιούμε την κλάση BinaryHyperCube και στην αναζήτηση των κοντινών σημείων τη συνάρτηση `cube.range_search`

3.6.4 SILHOUETTE

Η συνάρτηση Silhouette μας βοηθά να δημιουργήσουμε τον δείκτη εσωτερικής αξιολόγησης της συσταδοποίησης έτσι ώστε να ξέρουμε περίπου πόσο ακριβές ήταν το clustering μας. Οι τιμές που μας επιστρέφει ο δείκτης χωρίζονται ως εξής

- **1**

Που σημενει ότι το cluster είναι ορθό

- **Κοντά στο 0**

Που σημενει ότι το cluster είναι αμφισβητούμενο αλλά δεν χρειάζεται να αλλάξει

- **-1**

Που σημενη ότι είναι λαθος το cluster

Σύμφωνα με τον τύπο ορίσαμε δύο διανύσματα το α και το β για να υλοποιήσουμε την συνάρτηση μας

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} = \begin{cases} 1 - a(i)/b(i), & \text{if } a(i) < b(i) \\ 0, & \text{if } a(i) = b(i) \\ b(i)/a(i) - 1, & \text{if } a(i) > b(i) \end{cases} \in [-1, 1].$$

Για κάθε cluster υπολογίσαμε για κάθε σημείο του, την απόσταση από κάθε άλλο σημείο του cluster και βρήκαμε την μέση απόσταση. Έτσι υπολογίσαμε το α . Στην συνέχεια η συνάρτηση βρίσκει το 2ο κοντινότερο cluster του σημείου (όχι δηλαδή το cluster που βρίσκετε είδη αλλά το επομενο) και υπολογίζει την αποστάση. Έτσι βρίσκουμε και το β . Οπότε τώρα αυτό που μένει στην συνάρτηση είναι να υπολογίσει το s τις πιο πάνω εξίσωσης και να το βάλει στο πίνακα score ο οποίος κρατά όλα τα s για να τα επιστρέψει η συνάρτηση