ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΘΝΙΚΌ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΌ ΠΑΝΕΠΙΣΤΉΜΙΟ ΑΘΗΝΏΝ
ΤΜΉΜΑ ΠΛΗΡΟΦΟΡΙΚΉΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΏΝ

Τίτλος Εργασίας

1η Προγραμματιστική Εργασία

Αναζήτηση και συσταδοποίηση διανυσμάτων στη C/C++

Μάθημα

Κ23γ: Ανάπτυξη Λογισμικού για Αλγοριθμικά Προβλήματα

Χειμερινό εξάμηνο 2020-21

Ονοματεπώνυμα φοιτητών:

- Λάκης Κωνσταντίνος (Α.Μ.: 11152017 00069)
- Μαυραπίδης Νικόλαος (Α.Μ.: 11152017 00082)

Github link:

https://github.com/KonstantinosLakis/ApproximateVectorSearchAndClusterization

Αθήνα, 2020

ΚΑΤΆΛΟΓΟΣ ΑΡΧΕΊΩΝ ΚΏΔΙΚΑ ΚΑΙ ΕΠΙΚΕΦΑΛΊΔΩΝ

- **Aux.cpp**: Περιέχει βοηθητικές συναρτήσεις που χρησιμοποιούμε κατά την διάρκεια του προγράμματος.

- **LSHmain.cpp**: Περιέχει την main συνάρτηση για την υλοποίηση του αλγορίθμου LSH. Συνοπτικά, παίρνει τα ορίσματα και τις παραμέτρους απο την εντολή που δίνει ο χρήστης, ανοίγει το αρχείο που αντιστοιχεί στο dataset MNIST που περιέχει τις εικόνες καθώς και το αρχείο με το σύνολο αναζήτησης. Έπειτα, δημιουργείται η δομή LSH και καλούνται οι ζητούμενοι αλγόριθμοι για την εκτύπωση των αποτελεσμάτων στο αρχείο εξόδου.

- **Cubemain.cpp**: Ομοίως με το παραπάνω αρχείο, διαβάζονται ορίσματα και παράμετροι, ανοίγουν τα ζητούμενα αρχεία, δημιουργείται δομή HyperCube, καλούνται οι ζητούμενοι αλγόριθμοι και δημιουργείται το ζητούμενο αρχείο εξόδου.

- **Clusteringmain.cpp**: Ομοίως με τα παραπάνω αλλά για τα ζητούμενα του Β. κομματιού της εργασίας.

- **LSH.cpp**: Περιέχει τους ορισμούς των συναρτήσεων για την κλάση LSH όπως τις συνάρτησεις κατακερματισμού καθώς και τους ζητούμενους αλγορίθμους NearestNeighbor, kNearestNeighbors, rangeSearch.

- **Hashtable.cpp**: Ομοίως περιέχει τους ορισμούς των σχετικών συναρτήσεων για την κλάση HashTable.

- **Bucket.cpp**: Ομοιως περιέχει τους ορισμούς των σχετικών συναρτήσεων για την κλάση Bucket.

- **Image.cpp**: Ομοιως περιέχει τους ορισμούς των σχετικών συναρτήσεων για την κλάση Image.

- **HyperCube.cpp**: Περιέχει τους ορισμούς των συναρτήσεων για την κλάση HyperCube όπως τις συναρτήσεις κατακερματισμού καθώς και τους ζητούμενους αλγορίθμους NearestNeighbor, kNearestNeighbors, rangeSearch.

- **Aux.h**: Περιέχει τις δηλώσεις των συναρτήσεων που χρησιμοποιούνται στο αντίστοιχο .cpp καθώς και κατά την διάρκεια ολόκληρου του προγράμματος.

- **LSH.h**: Περιέχει τη δήλωση της κλάσσης LSH και των συναρτήσεων μελών της.

- **HashTable.h**: Περιέχει τη δήλωση της κλάσσης HashTable και των συναρτήσεων μελών της.

- **Bucket.h**: Περιέχει τη δήλωση της κλάσσης Bucket και των συναρτήσεων μελών της.

- **Image.h**: Περιέχει τη δήλωση της κλάσσης Image και των συναρτήσεων μελών της.

- **HyperCube.h**: Περιέχει τη δήλωση της κλάσσης HyperCube και των συναρτήσεων μελών της.

ΕΚΤΈΛΕΣΗ ΚΑΙ ΧΡΉΣΗ ΠΡΟΓΡΆΜΜΑΤΟΣ

Ο παρεχόμενος φάκελος συμπεριλαμβάνει ένα makefile για την μεταγλώττιση όλων των εκτελέσιμων. Ανάλογα με το εκτελέσιμο που επιθυμεί να τρέξει, ο χρήστης οφείλει να εισάγει τα σωστά ορίσματα τα οποία δηλώνονται ρητά στην εκφώνηση. Στο πρόγραμμα έχουμε ενσωματώσει ελέγχους για την σωστή εκχώρηση των ορισμάτων και των παραμέτρων σε περίπτωση που δεν δοθούν οι τιμές τους.

Όσο για την χρήση του προγράμματος, έχουν ακολουθηθεί πιστά οι συμβάσεις της εκφώνησης που δομούν την άσκηση. Επομένως ο χρήστης δεν έχει παρά να ακολουθήσει τις ίδιες οδηγίες για να μπορέσει να εκτελέσει σωστά το πρόγραμμα.

ΜΕΡΙΚΈΣ ΛΕΠΤΟΜΈΡΕΙΕΣ ΥΛΟΠΟΊΗΣΗΣ

1) Το BFS πάνω στον υπερκύβο γίνεται χωρίς την χρήση δεικτών από τον έναν γείτονα στον άλλον. Αντιθέτως, εκμεταλλευόμαστε την “ταυτότητα” των κορυφών, δηλαδή την δυαδική τους αναπαράσταση. Έτσι, έχοντας έναν πίνακα με Buckets, για να βρούμε τους γείτονες ενός bucket στην θέση i, δεν έχουμε παρά να υπολογίσουμε όλους τους αριθμούς με hamming distance 1 από τον i, αλλάζοντας απλά ένα bit την φορά, χρησιμοποιόντας unsigned int προφανώς.

2) Ο υπολογισμός των K κοντινότερων γειτόνων γίνεται με την χρήση ουράς προτεραιότητας. Όμως, δεν βάζουμε **όλους** τους γείτονες στην ουρά και εξάγουμε στο τέλος. Αντιθέτως, έχουμε ένα max-heap, το οποίο ανα πάσα στιγμή κρατά τον K-**μακρύτερο** γείτονα στην κορυφή. Κάθε φορά απλά συγκρίνουμε με την κορυφή δηλαδή, και κανουμε pop και insert αν χρειάζεται. Προφανώς, όταν η ουρά έχει λιγότερα από K στοιχεία, εισάγουμε ανεξαιρέτως.

3) Στην αντίστροφη ανάθεση, για την επίλυση των conflicts μεταξύ clusters για κάθε εικόνα, χρησιμοποιούμε λογική παρόμοια της εισαγωγής σε σχολή στις Πανελλήνιες. Δηλαδή, μία εικόνα αρχικά εισάγεται στο cluster που θα την εντοπίσει πρώτο. Κάθε επόμενο cluster που θα την εντοπίσει, θα την πάρει από το προηγούμενο cluster αν η καινούργια απόσταση είναι μικρότερη. Για να αποφύγουμε το βαλε-βγαλε σε sets (δομή που κρατάμε τα cluster), έχουμε ένα βοηθητικό struct για κάθε εικόνα, που μας λέει σε ποιό κεντροειδές είναι αντιστοιχισμένη ανά πάσα στιγμή η εικόνα. Το ίδιο struct μας βοηθάει να αγνοούμε εικόνες σε επόμενα loops (με μεγαλύτερο radius), καθώς άπαξ και βρεθεί ο καλύτερος cluster για μία εικόνα με κάποιο radius, αυτό δεν θα αλλάξει για μεγαλύτερο radius. Στο τέλος της επανάληψης, μέσω των struct αυτών, βρίσκονται οι εικόνες που δεν έχουν ανατεθεί σε cluster και προωθούνται στον αλγόριθμο Lloyd. Τα αποτελέσματα του reverse assignment και του Lloyd συγκεράζονται με την χρήση της set_union.

4) Έχει γίνει όσο το δυνατόν πιο ενδελεχής έλεγχος για την αποφυγή memory leaks με την χρήση του valgrind, και σε όλες τις εκτελέσεις που δοκιμάσαμε υπήρχε **μηδενικό** leak, με την χρήση της μνήμης να μην ξεπερνά τα 15MB για τα προγράμματα lsh και cube, χωρίς να υπολογίζονται τα αρχεία που φορτώνονται στην μνήμη.

ΒΕΛΤΙΣΤΟΠΟΙΉΣΕΙΣ

Στην υλοποίηση μας έχουν χρησιμοποιηθεί αρκετές βελτιστοποιήσεις. Κάποιες από τις σημαντικότερες είναι:

1) Για τις τιμές m^k, κρατούμε έναν στατικό πίνακα που υπολογίζεται μόνο μία φορά κατά την εκτέλεση του προγράμματος. Ο υπολογισμός του γίνεται με τον εξής τρόπο. Στην πρώτη θέση του πίνακα θέτουμε $m^0 = 1$. Έπειτα, υπολογίζουμε την τιμή mMult = m mod M **μία** φορά. Επαναληπτικά θέτουμε arr[i] = (arr[i – 1] * mMult) mod M. Εκμεταλλευόμαστε δηλαδή την σχέση m^i mod M = (m * m^{i-1}) mod M = (m mod M) * (m^{i-1} mod M) mod M.

2) Για τον υπολογισμό των συναρτήσεων h_i, χρησιμοποιούμε το εξής τέχνασμα. Θέτουμε hashSum = 0. Επαναληπτικά, θέτουμε hashSum = (hashSum + term) mod M. Εδώ, term = (a_i * m^{d-i-1}) mod M. Προφανώς και ο υπολογισμός του term γίνεται χρησιμοποιώντας την σχέση των διαφανειών καθώς και τον πίνακα του (1). Παρακάτω υπάρχει μία μικρή απόδειξη πως στο τέλος της επανάληψης θα ισχύει hashSum = h(x).

Επαγωγή ως προς το k (πλήθος όρων του αθροίσματος). Για k = 1, θα γίνει μόνο ένας υπολογισμός, hashSum = (0 + (a_0 * m^{d-1}) mod M) mod M = (a_0 * m^{d-1}) mod M. Προφανώς ισχύει hashSum = (Σ(a_i * m^{d-i-1}) i = 0..0) mod M. Για k >= 2, ισχύει το εξής: (Σ(a_i * m^{d-i-1}) i = 0..k - 1) mod M = ((Σ(a_i * m^{d-i-1}) i = 0..k - 2) mod M + a_(k-1) * m^{d-k} mod M) mod M (από σχέση διαφανειών). Στο k – 1 βήμα του αλγορίθμου, ισχύει (από επαγωγή) hashSum = (Σ(a_i * m^{d-i-1}) i = 0..k - 2) mod M. Στην συνέχεια, τίθεται hashSum = (hashSum + term) mod M. Το δεξί μέρος της ανάθεσης ισούται μαθηματικά με: ((Σ(a_i * m^{d-i-1}) i = 0..k - 2) mod M + a_(k-1) * m^{d-k} mod M) mod M = (Σ(a_i * m^{d-i-1}) i = 0..k – 1) mod M. Έτσι, στο τελευταίο βήμα έχουμε hashSum = h(x).

3) Δεδομένου πως M είναι δύναμη του 2, όπου mod M παραπάνω κάνουμε & με μία μάσκα που περιέχει την τιμή M – 1, αντί για % με το M.

4) Για την αντίστροφη ανάθεση, δίνεται ένα set στο rangeSearch με τις εικόνες που έχουν ήδη ανατεθεί σε clusters, ώστε αυτές να αγνοηθούν και να μην υπολογίζονται αποστάσεις χωρίς λόγο.