ΑΜ : 1115201400024
Δημήτριος Γάγγας

Ανάπτυξη Λογισμικού για Αλγοριθμικά Προβλήματα

Project 1
Lsh / HyperCube

Υλοποιήθηκε το lsh και το HyperCube επιτυχώς
σε C++11 με STL library σε λειτουργικό Linux.

- **[Optimality Note]** Σχεδόν όλες οι παράμετροι στις συναρτήσεις γίνονται pass by reference για εξοικονόμηση χώρου και βελτιστοποίηση της ταχύτητας.

- Ελέγχθηκαν επιτυχώς με Valgrind .Συνεπώς δεν υπάρχουν memory leaks.

- Για την μετατροπή των input αρχείων σε μορφή κατάλληλη σύμφωνα με τα δοθεντα πρότυπα χρησιμοποιήθηκε εντολή τερματικού με ανακατεύθυνση σε αρχείο .
  Συγκεκριμένα : **$ awk '{print NR,$0}' input_small >> counted_input_small**

- Χρησιμοποιήθηκε το git και το gitkraken για οπτικοποίηση των εκάστοτε αλλαγών αλλά και για την ευκολία επιστροφής σε προηγούμενα στάδια ανάπτυξης του κώδικα.

- Δημιουργήθηκαν 2 φάκελοι ξεχωριστοί.Ο ένας περιέχει όλα τα αρχεία που χρειάστηκαν για την υλοποίηση του lsh και ο δεύτερος τα αρχεία για τον υπερκύβο.

- Καθένα απ' αυτά τα αρχεία περιέχει δικό του ξεχωριστό makefile που δίνει τα εκτελέσιμα **lsh** και **cube** αντιστοιχα.

- Έχουν υλοποιηθεί όλες οι απαιτήσεις της εκφώνησης για τα ορίσματα με όποια σειρά και να δοθούν τα flags. Μπορώντας φυσικά να παραληφθούν και κάποια.
  1. $./lsh –d <input file> –q <query file> –k <int> -L <int> -o <output file>
  2. $./cube –d <input file> –q <query file> –k <int> -M <int> -probes <int> -o <output file>

- Περιέχεται σε κάθε φάκελο και ένα ενδεικτικό αρχείο output με αποτέλεσματα που έκανε generate το πρόγραμμα.

- Για την καλύτερο δυνατό έλεγχο του κώδικα,διασπάστηκε σε πλήθος αρχείων .cpp/.h ,κλάσεων και συναρτήσεων. Αναλύονται διεξοδικότερα παρακάτω.

Κοινές δομές και αρχεία
για lsh/hypercube

MyVector_Class_Functions.cpp/h :

Περιέχει τη **class** myVector
Ουσιαστικά η κλάση αντιπροσωπεύει όλα τα διανύσματα d-διαστατου χωρου απο το input και query dataset.
Συγκεκριμένα, η κλάση αποτελείται από ένα <vector> απο double αριθμούς που αντιπροσωπεύει τις συντεταγμένες των διανυσμάτων.
Τέλος , υπάρχουν υλοποιημένες και οι μέθοδοι που χρειάστηκαν για τα διανύσματα,όπως είναι το εσωτερικό γινόμενο και η ευκλείδεια νόρμα.

Metrics.cpp/h :

Περιέχει την **abstract class** DistanceMetrics , η οποία περιέχει τη μέθοδο
virtual double distance(myVector& p,myVector& q) = 0;

Η μέθοδος αυτή υλοποιείται απο τις 2 επιμέρους κλάσεις που την κληρονομούν

- ★ **class** EuclideanMetric final
- ★ **class** CosineMetric final

Και αφορά για την ευκλείδεια ή την cosine similarity μετρική απόστασης.

HashTablesStructures.cpp/h : (~Περίπου ίδια)

Εδώ αξίζει να αναφερθεί το εξης:
Στις δομές του HashTable αποθηκεύονται items_ids και οχι δείκτες σε διανύσματα!!
Ουσιαστικά κατα το preprocessing τα input και query διανυσματα αποθηκεύονται σε δομες **unordered maps** της cpp ή κοινώς σε dictionaries όπως γνωρίζουμε απο τη python.
Τα unordered maps έχουν την μορφή : std::unordered_map <std::string, myVector > ,
όπου *string* αφορά το id του vector (ή πιό σωστά τη γραμμή στην οποία διαβάστηκε)
και *myVector* το ίδιο το διάνυσμα.

Περιέχει τις δομές *from bottom up* :

- ★ **struct** bucket_chain
  Αποτελείται απο ενα string vector που περιέχει τα id των δοθέντων p διανυσμάτων.

- ★ **struct** hash_table
  Αποτελείται από έναν vector απο pointers μεγέθους HT_TableSize και τύπου bucket chain, αλλά καθώς επίσης και εναν δείκτη τυπου LSH * / HyperCube *
  (εδώ έγκειται και η μεγάλη διαφορά μεταξύ lsh και υπερκύβου).

  Για το lsh και cube ο δείκτης χρειάζεται για να γνωρίζουν τον τρόπο υπολογισμου του index μέσω της get_HT_index(myVector& p) .
  [Για το lsh χρειάζεται επίσης για να γνωρίζει ανα hash table την συνάρτηση g()]

★ **struct** multi_hash_table
Δομή που απλά περιέχει vector απο L pointers τυπου hash table.Για cube **L=1**.

SearchingAlgorithms.cpp/h :

Ουσιαστικά η κλάση αντιπροσωπεύει όλα τα διανύσματα d-διαστατου χωρου απο το input και query dataset.
Περιέχει όλους του ζητούμενους αλγορίθμους αναζήτησης.
Συγκεκριμένα, περιεχει τους εξης:

- ➔ NN_search
- ➔ RangeN_search
- ➔ TrueNN_search (*Brute Force*)

Για το LSH

LSH.h :

Η διεπαφή αυτή περιέχει την **abstract** class **class** LSH ,την οποία την κληρονομούν οι κλάσεις
class EuclideanSpaceLSH και η **class** CosineSimilarityLSH .
Επί της ουσίας περιέχει 2 virtual functions ,οι οποίες αξιοποιούνται από τις 2 προαναφερθέντες κλάσεις.Οι συναρτήσεις αυτες ειναι οι εξης :

- ➔ **virtual unsigned int** get_HT_index(myVector& p) **const** = 0 ;
Ουσιαστικά δοθέντος ενος διανύσματος p επιστρέφει το index του HashTable.

- ➔ **virtual** string g(myVector& p) =0;
Η hash function g για πολυδιάστατο χώρο όπως αναφέρεται και στις διαφάνειες.

LshFunctions.cpp/h :

Αφορά κυρίως το preprocessing.
Συγκεκριμένα οι πιο σημαντικές εξ αυτών είναι:

- ➔ **void** ReadHandleArgms(**int**& argc, **char****& argv , string &inFileName , string &QueryFileName , **unsigned int** &k , **unsigned int** &L , string &OutFileName);

  Διαβάζει τα ορίσματα οπως ζητείται στην εκφώνηση.

- ➔ **void** ReadInFile_save2umap(string inFileName, unordered_map<string, myVector > &umap , **unsigned int**& d ,**int**& flag);
- ➔ **void** ReadQueryFile_save2umap(string QueryFileName ,unordered_map<string, myVector > &query_umap , **unsigned int**& d , **int**& Radius);

  Αφορούν κυρίως το **preprocessing** κατα το οποιο διαβάζουν το input και το query file και τα αποθηκεύουν στις κατάλληλες δομές.

➔ **void** Write2_OutFile(std::ofstream& outFile ,string &OutFileName ,string &query , vector<string> &Rnn_list , string &ApproxNN , **double** &distAproxNN_from_q , **double** &distTrueNN_from_q , **double** &elapsed_secs_for_AproxNN , **double** &elapsed_secs_for_TrueNN);

Γράφει τα αποτελέσματα στο δοθέν output file.

EuclideanSpaceLSH.cpp/h :

Περιέχει τις δομές για τον ευκλείδειο χώρο όπως έχουμε διδαχθεί στο μάθημα.
Συγκεκριμένα, οι κλάσεις που υλοποιήθηκαν είναι οι εξης :

★ **class** H_Class , **class** F_HF_Obj

Περιέχουν τη δομή h και φ αντίστοιχα οπως αναφέρονται στις διαφάνειες.

★ **class** EuclideanSpaceLSH :

Περιέχει <vector> απο H_Class και 1 object τυπου F_HF_Obj
Ουσιαστικά η κλάση αφορά όλο τον ευκλειδειο χωρο.

CosineSimilarityLSH.cpp/h :

Περιέχει τις αντίστοιχες δομές για την ομοιότητα συνημιτόνου όπως έχουμε διδαχθεί στο μάθημα.
Συγκεκριμένα, οι κλάσεις που υλοποιήθηκαν είναι οι εξης :

★ **class** H_Class_Cos

Περιέχουν τη δομή h για την ομοιότητα συνημιτόνου οπως αναφέρεται στις διαφάνειες.

★ **class** CosineSimilarityLSH

Περιέχει τις απαραίτητες λειτουργίες για την μέθοδο ομοιότητα συνημιτόνου.

Για το HYPERCUBE

HyperCube.h :

Επι της ουσίας είναι η ίδια διαδικασία οπως στο LSH χωρις τον υπολογισμο της g().

CubeFunctions.cpp/h :

Αρκετά όμοια με τα αρχεία LshFunctions.cpp/.h .
Κύριες διαφορές είναι ότι έχουν αλλαχτεί τα ορίσματα ωστε να συμβαδίζει με τα πρότυπα της εκφώνησης.
Επίσης έχουν προστεθεί οι 2 εξης συναρτήσεις :

- ➔ `vector <unsigned int > getAllNeigborsIndexes( unsigned int index , unsigned int k );`
  Επιστρέφει ολα τα indexes του HashTable των γειτονικών κορυφών που έχουν Hamming distance = 1.

- ➔ `vector <string> HammingDist1(const string &str );`
  Για ένα δοθέν bitstring που αντιπροσωπεύει το index του HashTable
  Επιστρέφει όλα τα δυνατά substrings που έχουν Hamming Distance 1.
  Μετέπειτα , αυτά θα μετατραπούν σε indexes του Hashtable μέσω της getAllNeigborsIndexes().

EuclideanSpaceCube.cpp/h :

Παρόμοια με το Lsh περιέχει τις κλάσεις που χρειάστηκαν να υλοποιηθούν.
Συγκεκριμένες διαφορες ειναι οι εξης:

1. Δεν υπάρχει πλέον η **class** F_HF_Obj καθώς δεν χρησιμοποιείται η φ() στον υπερκύβο.
2. Δημιουργήθηκε συνάρτηση string f**();** , η οποία δωθέντος μια ακέραιας τιμής από το εκάστοτε h αποδίδει 0 ή 1 .
3. Για την ακριβώς από πάνω περίπτωση για κάθε h-i κρατείται και ένα unordered_map Τυπου unordered_map< **long long int**, string > , όπου σαν κλειδί δίνεται η ακέραια τιμή που επιστρέφει η h-i και σαν τιμη ενα string literal που είναι είται “1” είτε “0”.
   Συνεπώς ,αν στην f() δοθεί σαν όρισμα μια τιμή της εκάστοτε h που έχει ήδη “ξανάρθει” στο παρελθόν τότε η f επιστρέφει τη τιμη unordered_map_i [h_i value].

CosineSimilarityCube.cpp/h :

Καμια απολύτως διαφορά με το CosineSimilarityLSH.
<u>Νοητικά</u> η μόνη τους διαφορά είναι ότι στο lsh εχουμε L Hash Tables ενώ στο HyperCube έχουμε μόλις 1.

Μετρήσεις-Πειράματα-Σχόλια

- Στις δομές του HashTable απο
  Πειραματιζόμενος με τις παραμέτρους και στις 2 υλοποιήσεις στο μικρο δοθέν dataset
  ,παρατήρησα οτι πετυχαίνω περίπου σχεδόν ίσο κλάσμα προσέγγισης με τιμη 1.6 - 1.7 για τις συγκεκριμένες τιμές :

Για LSH :
K =3
L =3
W = 300
TableSize = n/2

Ενώ για το HyperCube :
K = 11
Probes = 2 [μια γειτονική κορυφή]
M = ∞ [δε λαμβάνεται υπόψη]

- Η μέτρηση του χώρου έγινε με το Valgrind .
  Όμως όπως αναφέρει το documentation
  "It measures how much heap memory your program uses. This includes both the useful space, and the extra bytes allocated for book-keeping and alignment purposes."

  Δε μας καθιστά ένα σαφές κριτήριο μέτρησης του πραγματικού χώρου που καταναλώνουν οι δομές του προγράμματος.
  Παρόλ' αυτά ενδεικτικές εκτελέσεις έβγαλαν αυτά τα αποτελέσματα :

  Για το lsh με το μικρο dataset και τις προαναφερθέντες παραμέτρους :

```
==15691== HEAP SUMMARY:
==15691==     in use at exit: 0 bytes in 0 blocks
==15691==   total heap usage: 163,677 allocs, 163,677 frees, 15,247,298 bytes allocated
==15691==
==15691== All heap blocks were freed -- no leaks are possible
==15691==
==15691== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==15691== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
dimitrisgan@dimitrisgan-MS-7918:~/Desktop/project1/project1/LSH$
```

  Αντίστοιχα για το hypercube με το μικρο dataset και τις προαναφερθέντες παραμέτρους :

```
==16020== HEAP SUMMARY:
==16020==     in use at exit: 0 bytes in 0 blocks
==16020==   total heap usage: 146,543 allocs, 146,543 frees, 18,376,398 bytes allocated
==16020==
==16020== All heap blocks were freed -- no leaks are possible
==16020==
==16020== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==16020== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
dimitrisgan@dimitrisgan-MS-7918:~/Desktop/project1/project1/HyperCube$
```