Project Algorithms 1

Σκορδούλης Κωνσταντίνος 1115 2016 00155

Main Programs: lsh.cpp, cube.cpp, curve_grid_lsh.cpp, curve_grid_hypercube.cpp, curve_projection_lsh.cpp, curve_projection_hypercube.cpp (6 in total)

Source Programs (with their header files): Curve.cpp, Curve_LSH.cpp, HyperCube.cpp, InfoCurve.cpp, InfoItem.cpp, LSH_HT.cpp, Relevant_List.cpp, Relevant_Node.cpp, Relevant_Table.cpp, Utilities.cpp, Utilities2.cpp, Vector.cpp, Vector_Node.cpp

Compilation: Υπάρχει διαθέσιμο **Makefile**, οπότε με την εντολή **make**, γίνεται γρήγορα το compilation.

ΣΗΜΑΝΤΙΚΟ: Το πρόγραμμα υλοποιήθηκε με C++ ➔ **(C++11)**. Χρησιμοποιήθηκαν αρκετές συναρτήσεις/δομές της STL, όπως **vectors** (είναι σαν arrays , αλλά με πολλές παραπάνω λειτουργίες και μεταβλητό μέγεθος).

ΣΗΜΑΝΤΙΚΟ2: Επειδή η αγγλική μετάφραση του διανύσματος είναι Vector, καταλαβαίνω μπορεί να γίνει πονοκέφαλος με τους vectors(STL). Στο κώδικα μου όταν μιλάω για **vectors ➔ STL**, ενώ όταν λέω **Vectors➔Object ➔ Διάνυσμα**. Διαφορά αν είναι uppercase ή lowercase.

ΣΗΜΑΝΤΙΚΟ3: Για όλες τις κατανομές, χρησιμοποιήθηκαν έτοιμες βιβλιοθήκες της C++ ➔ std::normal_real_distribution, std::uniform_real_distribution, std::uniform_int_distribution.

ΣΗΜΑΝΤΙΚΟ4: Σε μερικές συναρτήσεις έχω και **function()** και **function1()**. Συνήθως έχουν ίδια λειτουργικότητα, απλά η **function ➔ Vectors**, ενώ η **function1() ➔ Curves.**

Περιγραφή main files

Δεν υπάρχουν πολλά να πω. Σε κάθε main υπάρχει συνάρτηση που κάνει parse το input και query file. Φτιάχνει τις δομές και τα βάζει μέσα σε vectors.

Κάθε φορά **εκτελείται πρώτα η brute force**, για τον υπολογισμό των πραγματικών αποστάσεων. Μετά υπολογίζονται οι παράμετροι (που μπορούμε να υπολογίσουμε). Στη συνέχεια, εισάγονται τα Vectors/Curves στις δομές και ακολουθούν τα queries. Μόλις τελειώσει αυτό, τυπώνονται τα **Στατιστικά** (more on this later), και το **memory cleanup**.

Θα πάρω κάθε μια main, και θα δείξω ποια source files χρησιμοποιεί.

Περιγραφή source files

1. **Curve.cpp**: Αυτή είναι η δομή των καμπυλών μας. Όσον αφορά τα σημεία των καμπυλών, αποφάσισα να χρησιμοποιήσω **vector< vector<double>* >** , δηλαδή έναν vector που περιέχει δείκτες σε vector<double>. Καλύτερη εξήγηση ➔ δείτε το Curves.hpp.
2. **Curves_LSH**: Εδώ έχουμε την δομή του **LSH_grid**. Για κάθε grid δημιουργείται ένα **LSH_HT ➔ LSH hashTable** ή **Hypercube**(**htc**).
    a. Τα grid points που προκύπτουν (για το input) αποθηκεύονται (στο εσωτερικό του αντικειμένου) σε vectors<>, και μόλις γίνει το **padding ➔ (max_coord + 1) of grid points** ,εισάγονται στην αντίστοιχη δομή για να γίνει το hashing.
    b. Επιπλέον, οι συναρτήσεις που μετατρέπουν **Curve ➔ Grid Curve** και **Grid Curve ➔ Grid points**, είναι η **Curve2Gcurve()** και **Gcurve2Curve() .**
    c. Η συνάρτηση που κάνει το padding λέγεται **Update**.
3. **Hypercube**: Εδώ έχουμε την δομή του **Υπερκύβου**.
    a. Όπως και στην **LSH**, παράγουμε το uniform real διάνυσμα s. Μόνο που εδώ χρειαζόμαστε **d' * g(x) ➔ d' * k * h(x)**. Άρα έχουμε **double*** s ➔ 3D table**, διαστάσεων (**d' x k x d-1**), όπου **k** το πλήθος των h(x), **d** η διάσταση των σημείων και **d'**(ή bh_d ➔binaryHypercube_d όπως το γράφω) είναι η διάσταση του υπερκύβου (κάθε κελί παράγεται από uniform_real_distribution, κατά τη δημιουργία του υπερκύβου).
    b. Κάθε φορά που παράγεται μια g(x) αποδίδω **0 ή 1**, με την συνάρτηση **binary()** (χρησιμοποιεί την uniform_int_distribution, που αρχικοποιήθηκε κατά τη δημιουργία του αντικειμένου).
    c. Επειδή όμως θέλω να θυμάμαι τι τιμές έχω αποδώσει , φτιάχνω ένα **ευρετήριο ➔ std::unordered_map<>**, στο οποίο κάθε element είναι ένα ζευγάρι **(key,value) == (string, int).** Το string είναι η g(x) (don't worry I will explain in a bit), και το int είναι το 0 ή 1.
    d. **Hash()**, παράγει το g(x) κατά τα γνωστά και ελέγχει αν υπάρχει ήδη στο ευρετηριο μας. Αν όχι ➔ κάλεσε την binary(), αλλιώς πάρε την τιμή που είναι αποθηκευμένη.
    e. **Search()**, πριν αρχίσει να ψάχνει τον κοντινότερο γείτονα, παράγει αναδρομικά όλα τα binary string με διαφορετικές Hamming Distance (αρχίζοντας από Hamming Distance = 1, έως Hamming Distance = length of original ➔ όλα τα bytes αλλαγμένα). Η συνάρτηση που κάνει αυτούς τους υπολογισμούς λέγεται **Hamming_Distance()**. Αυτά τα binary string (μαζί με το original αποθηκεύονται σε ένα vector. Και ξεκινάμε επιτέλους την αναζήτηση , ψάχνοντας στο current bucket και επισκεπτόμαστε και τα υπόλοιπα.
    f. **Search1()**, ίδια με την Search() απλά απευθύνεται σε **Curves ➔ DTW**.
4. **InfoCurve**: ➔**Info for Curves.** Το συγκεκριμένο class, εξυπηρετεί απλά την αποθήκευση των **Στατιστικών/Output** που απαιτούνται για την έξοδο των προγραμμάτων του Β' μέρους. Περιέχει πεδία όπως **πραγματικός γείτονας**, **πραγματική απόσταση**, **approximate απόσταση** κλπ.
5. **InfoItem**: ➔ **Info for Vectors**. Όπως και στην InfoCurve, πάλι για **Στατιστικούς/Output** λόγους.

6. **LSH_HT**: ➔ **LSH_HashTable**. Η αγαπημένη μας μέθοδο LSH, εδώ έχουμε το HashTable της μεθόδου. Τα **number of buckets**
    a. Παράγουμε το **double** s** s ➔** (k x d). Τα στοιχεία του προέρχονται από την uniform_real_distribution. Το s φτιάχνεται κατά τη δημιουργία του Hash_Table
    b. **Hash()**. Εδώ παράγουμε τα **ai**, σύμφωνα με την Manhattan metric. Το **m = 2^32** . Για το mod() χρησιμοποίησα δικιά μου συνάρτηση ➔**Mod()** (στη **Utilities.cpp**). Επιπλέον για το **modular exponentiation**, έφτιαξα την **alt_pow()**. Μόλις παραχθούν όλα τα h(x), τα μετατόπιζα στην κατάλληλη θέση χρησιμοποιώντας std::**sll() ➔ shift left logical.** Τον παραγόμενο αριθμό **(32 bit)** τον έκανα **string**.
        i. **ΣΗΜΑΝΤΙΚΟ**: τα g(x) (αφού τα υπολόγιζα με τον παραπάνω τρόπο σε 32 bit int) τα αποθηκεύω σαν **STRING ➔** δεν αλλάζει καθόλου το περιεχόμενο του αλγορίθμου (απλά συγκρίνω με strings αντί για integers)
    c. **Insert()**. Δέχομαι τα **input Vectors**. Βρίσκω την g(x), και φτιάχνω για κάθε Vector μια νέα δομή **Vector_Node**(more on this later). Αυτή η δομή θα είχε αποθηκευμένη την τιμή του g(x) για να γίνουν οι απαραίτητες συγκρίσεις αργότερα. Εισάγεται ως Vector_Node στο κατάλληλο bucket.
    d. **Search()**. Αφού βρούμε το g(x) , και μας οδηγήσουν στο κατάλληλο bucket, ψάχνουμε 1-1 τα Vector_Nodes του bucket. Αν έχουμε **ίδιο g(x)**, υπολογίζουμε την **Manhattan Distance**, και συγκρίνουμε κάθε φορά με τον προ-υπολογισμένο κοντινότερο ➔ if we found a new approximate nearest neighbor. Έχω βάλει και **άνω φράγμα για τον έλεγχο στοιχείων στο bucket**, το οποίο μπορείται να τροποποιήσετε αν θέλετε.
7. **Curve_Projection:** Αφορά 3 αρχεία, **Relevant_Table.cpp, Relevant_List.cpp, Relevant_Node.cpp.** Με αυτή τη σειρά μιλάνε για:
    a. τον πίνακα από **δείκτες σε Relevant_Lists ➔** Τον έχω φτιάξε σαν class.
        i. Ο πραγματικός πίνακας (**table**) είναι διαστάσεων MxM και περιέχει δείκτες στα **Relevant_List**.
        ii. Ο normal_real_distribution πίνακας **G,** παράγεται στον constructor και μεταφέρεται σε κάθε **Relevant_Node** (**Relevant_table ➔ Relevant_List ➔ Relevant_Node**).
        iii. **Insert()** και **Insert1()**. Η πρώτη απευθύνεται για **LSH_HT**, ενώ η δεύτερη σε **HyperCube**. Χρησιμοποιούμε την **Categorise()**, η οποία παράγει ένα **πίνακα από vector< Curve* >**, οπού ταξινομεί τα Curves ανάλογα με το μήκος τους. Πχ το cell(1) περιέχει **Curves** μήκους 1 κλπ. Μόλις τελειώσουμε, στέλνουμε τα Curves στα κατάλληλα Relevant_Lists.
        iv. **Σημείωση:** Εφάρμοσα τις προτεινόμενες αλλαγές:
            1. μόνο 7 διαγώνιοι μας ενδιαφέρουν (διαγώνιοι = 2*window+1, η window μπορεί να πειραχτεί στη main).
            2. Δεν μας ενδιαφέρουν τα relevant traversals κάτω από τη πορτοκαλί διαγώνιο.
        v. **Search().** Απλά προωθούμε το Curve στα κατάλληλα Relevant_Lists (εφαρμόζουμε το σχετικό παραθυράκι πάνω και κάτω του (I,j)).
    b. λίστα/**vector<>** όπου κάθε node είναι ένα **Relevant_Node**

i. **<u>Σημείωση:</u>** Όλα τα Relevant traversals, είναι μια λίστα από ζευγάρια. Για την αναπαράσταση αυτών των ζευγαριών χρησιμοποιώ **std::pair (STL)**.
ii. Καταρχάς υπολογίζουμε την πορτοκαλί διαγώνιο με τη βοήθεια της **Main_RT()**.
    1. Η πορτοκαλί διαγώνιος είναι ουσιαστικά σαν να έχουμε μια ευθεία **y = λ * x**, και αφού φτιάξουμε το σχετικό grid, να χρωματίσουμε τα σημεία από όπου περναέι.
    2. Βρίσκουμε λοιπόν το **λ** (κλίση ευθείας **λ = y/x ➔ double λ = j_length/ i_length**)
    3. Υπολογίζουμε αναδρομικά το πορτοκαλί grid.
    4. Αρχίζοντας με (i,j) = (0,0) ➔ (xi,yj) = (1,1). Σε κάθε βήμα συγκρίνουμε το **y της ευθείας**, με το **yj**. Αν είναι **(=)** τότε κατευθυνόμαστε **διαγώνια ➔ i+1 και j+1**. Αν είναι **(<)** τότε πήγαινε **δεξιά ➔ i+1**. Αν είναι **(>)** τότε πήγαινε **πάνω ➔ j+1.**
iii. Βρίσκουμε την πορτοκαλί διαγώνιο, και ορίζουμε για κάθε j (y συντεταγμένη) το όριο της i (x συντεταγμένη). Αποθηκεύονται τα όρια σε ένα **vector<int>* limits**. Όλο αυτό για να κόψουμε το κάτω μέρος από τη πορτοκαλή διαγώνιο.
iv. Βρίσκουμε όλα τα **Relevant Traversals** με τη βοήθεια της **ALL_RT()**, η οποία δοκιμάζει όλους τους συνδυασμούς (που πληρούν τις προϋποθέσεις).
v. Για καθένα από αυτά φτιάχνουμε το αντίστοιχο Relevant Node.

c. Κάθε **Relevant Node** ,
i. Έχει ως "**ID**" ένα από τα πιθανά **Relevant Traversals**
ii. Έχει δείκτες και για **LSH_HT** και για **HyperCube** (μόνο ένας από τους 2 θα λειτουργεί, ο άλλος NULL).
iii. **Insert()** και **Insert1()**. Η πρώτη αφορά τη δημιουργία και εισαγωγή σε πολλά **LSH_HTs**, ενώ το 2^{o} στο Hypercube. Για να φτιάξουμε τα vectors, χρειαζόμαστε την **CreateX()**.
iv. **CreateX()**. Σύμφωνα με το Relevant Traversal του node, επιλέγονται τα σημεία τα οποία πολλαπλασιάζονται με τον **G ➔ πολλαπλασιασμός πινάκων** (επιτυγχάνεται με την **Multiply_arrays()**). Αφού βγουν τα καινούργια σημεία (after multiply) <u>ενώνουμε όλες τις συντεταγμένες</u> και **insert them to HashTable.**
v. Η **CreateY()** είναι ίδια απλά απευθύνεται σε queries.
vi. Η **Search()**, χρησιμοποιεί την **CreateY()** για να εισάγει το query στο hash table.

8. **<u>Utilities.cpp</u>**: Εδώ περιέχονται διάφορες χρήσιμες συναρτήσεις για το 1^{o} κομμάτι της εργασίας που αφορά **Vectors**. ➔ **Mod() , alt_pow()** κλπ
9. **<u>Utilities2.cpp</u>**: Εδώ περιέχονται διάφορες χρήσιμες συναρτήσεις για το 2^{o} κομμάτι της εργασίας που αφορά **Curves**.
10. **<u>Vector.cpp:</u>** Η δομή μας για τα **διανύσματα**
11. **<u>Vector_Node:</u>** Ουσιαστικά είναι **Vectors**, τα οποία έχουν extra πληροφορίες:
    a. **g(x)** , που χρησιμεύει σε οτιδήποτε έχει σχέση με **LSH σημείων**.

b. **Curve* Real**, δείκτη στην πραγματική καμπύλη, για να μπορέσουμε να κάνουμε **DTW**.
c. **Curve* Grid** (scrapped Idea). Δείκτη στην Grid καμπύλη που προκύπτει από την Curve_LSH. Η original ιδέα είναι ότι στην Search() που κάνεις στη curve_grid_lsh, να συγκρίνεις αυτά που έχουν ίδιο g(x) αλλά και ίδιο Grid Curve.

Σχολιασμός Αποτελεσμάτων

Σε τέτοια προβλήματα πολύ σημαντικό ρόλο παίζουν και οι τιμές των παραμέτρων (parameter sensitive problems). Παρατήρησα λοιπόν τα εξής πράγματα:

1. **Σύγκριση LSH και HyperCube**: Η **Hypercube** είναι πιο γρήγορη στην αναζήτηση κοντινότερων γειτόνων, αλλά τρώει ένα μικρό **penalty** στην ακρίβεια.
2. **Σύγκριση Grid και Projection**: Και οι 2 βγάζουν σχετικά παρόμοια ακρίβεια. Το **Grid_Curve** μπορεί να δουλέψει για καμπύλες μεγάλων διαστάσεων, ενώ το **Projection** (για φυσιολογικούς υπολογιστές) έχει θέματα μνήμης. Για μικρό πλήθος μήκος καμπυλών (για να τρέχουν και οι 2) η **Projection** είναι λίγο πιο γρήγορη , βγάζοντας παρόμοια ακρίβεια με το Grid.
3. **Τι επιλέγουμε από τις 4?**: Επιλέγουμε ανάλογα με τις ανάγκες μας.
    a. Καταρχάς το **πλήθος των διαστάσεων**.
        i. **Grid_Curve**, αν έχουμε πολλές διαστάσεις (> 10)
        ii. **Projection**, αν έχουμε λίγες διαστάσεις (μέχρι 10)
    b. Τι κάνουμε **prioritize ➔ ακρίβεια ή ταχύτητα**
        i. **LSH**, αν θέλουμε ακρίβεια
        ii. **HyperCube,** αν θέλουμε ταχύτητα.