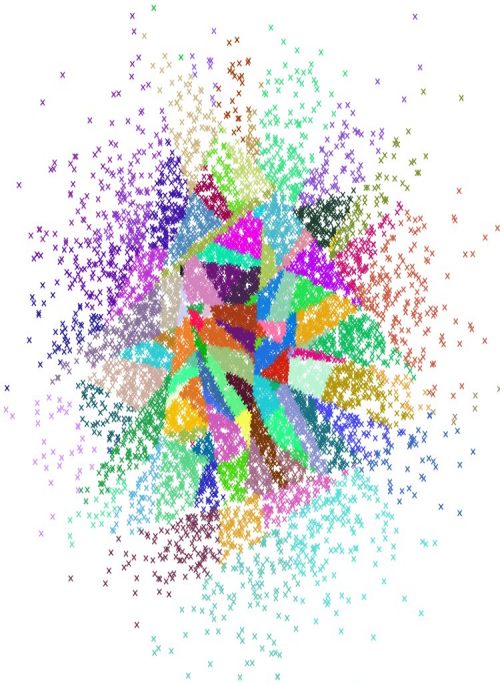
Ανάπτυξη Λογισμικού για Αλγοριθμικά Προβλήματα

2019-2020

Ανάπτυξη Λογισμικού για το Πρόβλημα Nearest Neighbours

Υλοποίηση Αλγορίθμου LSH βάσει Manhattan distance &
Υλοποίηση Μεθόδου Τυχαίας Προβολής σε Υπερκύβο
για διανύσματα στον d-διάστατο χώρο &
για πολυγωνικές καμπύλες

των φοιτητών
Δημήτρη Αλεξανδρή & Ιωάννας Ζαπαλίδη
1115201400006 & 1115201400044



Περιεχόμενα

Εισαγωγή

Η μέθοδος εύρεσης κοντινότερων γειτόνων πρόκειται για μία εξαιρετικά χρήσιμη διαδικασία που χρησιμοποιείται ευρέως στον κλάδο του Data Science. Πιο συγκεκριμένα, εξίσου σε διαδικασίες classification αλλά και clustering, η nearest neighbors αποτελεί μία από τις πλέον κατανοητές μεθόδους, που επιπλέον επιδέχεται πληθώρα επεκτάσεων και βελτιστοποιήσεων. Δύο μέθοδοι που έχουν αναπτυχθεί με σκοπό τη βελτίωση αυτού είναι η LSH & η μέθοδος τυχαίας προβολής στον υπερκύβο.

Αναλυτικότερα, η μέθοδος LSH βασίζεται στην ιδέα ότι αντί να αναζητούμε τον κοντινότερο γείτονα ενός διανύσματος από όλον τον “κόσμο” με τα διανύσματα που έχουμε στο χώρο μας, αντ’αυτού ομαδοποιούμε τα διανύσματα αυτά σε hashtables. Επιπλέον, κάνουμε αυτό τον χωρισμό αρκετές φορές (L), ώστε να έχουμε μικρότερη πιθανότητα λάθους. Κάθε hashtable έχει μέσα buckets, στα οποία καταλήγει κάποιο διάνυσμα μέσω μιας συνάρτησης g, στην οποία φτάνουμε περνώντας κάθε διάνυσμα μέσα από κάποιες συναρτήσεις h. Οι συναρτήσεις h αυτές είναι πο κάνουν αυτήν την διαδικασία ξεχωριστή. Η οικογένεια συναρτήσεων h είναι τέτοια ώστε να έχουμε καλό διαμοιρασμό των διανυσμάτων του “κόσμου” μας στα διάφορα buckets των hashtables.

Έπειτα, η μέθοδος υπερκύβου έχει κάποια κοινά με την μέθοδο LSH. Εδώ, δεν έχουμε πολλά αντίγραφα “κόσμου” σε πολλά hashtables, αλλά μόνο ένα. Η αντίστοιχη συνάρτηση g επιστρέφει ένα συνδυασμό bits, τα οποία αντιστοιχίζουν σε κάποια κορυφή του υπερκύβου στις διαστάσεις που βρισκόμαστε. Λαμβάνοντας υπόψιν την απόσταση Hamming των συνδυασμών bits, και άρα των κορυφών του κύβου, βρίσκουμε τον κοντινότερο γείτονα κάποιου διανύσματος αλλά και τους κοντινότερούς του σε κάποια έκταση.

Στην συνέχεια, εκτελούμε τους ανωτέρω αλγορίθμους, όχι σε διανύσματα d διαστάσεων, αλλά σε διαδομές, δηλαδή πολυγωνικές καμπύλες στον δισδιάστατο χώρο. Μετατρέπουμε κάθε καμπύλη σε διάνυσμα με την βοήθεια του αλγορίθμου DTW, αφότου φτιάξουμε L grids, όπου εφαρμόζουμε κάθε καμπύλη για να βρούμε την ισοδύναμή της grid curve. Τις τελευταίες, τις μετατρέπουμε σε διανύσματα και έπειτα επανεκτελούμε είτε LSH είτε hypercube για διανύσματα.

Τέλος, εκτελούμε την μέθοδο τυχαίας προβολής για καμπύλες, με τα παραγόμενα διανύσματα να υφίστανται, όπως παραπάνω, LSH & hypercube.

Ανάπτυξη και Δομή Κώδικα

Χρησιμοποιησαμε git και GitHub για το version control κατά την εκπόνηση της εργασίας μας.

Τα αρχεία μας:

1. `Makefile` – *με οδηγίες μεταγλώττισης και εκτέλεσης*
2. `lsh.cpp` – *η εφαρμογή της μεθόδου*
3. `cube.cpp` – *η εφαρμογή της μεθόδου*
4. `curve_grid_hypercube.cpp` – *η εφαρμογή της μεθόδου*
5. *`curve_grid_lsh.cpp` – η εφαρμογή της μεθόδου*
6. `curve_projection_hypercube.cpp` – *η εφαρμογή της μεθόδου*
7. `curve_projection_hypercube.cpp` – *η εφαρμογή της μεθόδου*
8. `h_funs.h/h_funs.cpp` – *οι συναρτήσεις* h_i
9. `g_funs.h/g_funs.cpp` – *οι συναρτήσεις* g_i
10. `ht.h/ht.cpp` – *δική μας κλάση hash table (για LSH - διανύσματα)*
11. `cube_ht.h/cube_ht.cpp` – *δική μας κλάση hash table (για υπερκύβο – διανύσματα)*
12. `curve_ht.h/curve_ht.cpp` – *δική μας κλάση hash table (για υπερκύβο - καμπύλες)*
13. `curve_cube_ht.h/curve_cube_ht.cpp` – *δική μας κλάση hash table (για υπερκύβο - καμπύλες)*
14. `my_vector.h/my_vector.cpp` – *δική μας κλάση διανυσμάτων*
15. `NNpair.h/NNpair.cpp` – *ζεύγη διανυσμάτων που είναι κοντινότεροι γείτονες & η απόστασή τους*
16. `curve_points.h/curve_points.cpp` – *κλάση με τα σημεία των καμπυλών*
17. `curve.h/curve.cpp` – *κλάση των καμπυλών*
18. `pan_traversal.hpp` – *κλάση των traversals & υλοποίηση DTW*
19. `grid.h/grid.cpp` – *κλάση grid (LSH)*
20. `cube_grid.h/cube_grid.cpp` – *κλάση grid (Hypercube)*
21. `readme.pdf` – *αυτό το αρχείο*
22. `utils.h` – *συμπληρωματικές συναρτήσεις που χρησιμοποιούμε*

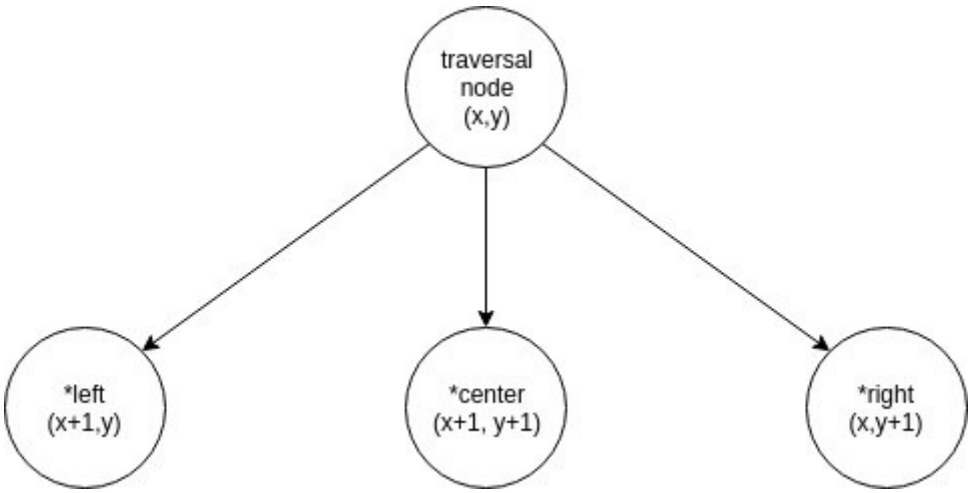
Καθολικές Σχεδιαστικές Επιλογές:

- Έχουμε υλοποιήσει μια δομή (NNpair) που διατηρεί τα ids των διανυσμάτων που αποτελούν το ένα nearest neighbour του άλλου (είτε για χρήση στον actual NN είτε στους άλλους αλγορίθμους που υλοποιήσαμε).
- Επιλέξαμε οι αποστάσεις (L1 - Manhattan, εδώ) να είναι πάντοτε τύπου double χάριν ευχρησίας του κώδικα.
- Η μέτρηση του εκάστοτε χρόνου εκτέλεσης γίνεται με χρήση της βιβλιοθήκης `<chrono>` της C++, με ακρίβεια microseconds.

- Χρησιμοποιήσαμε templates της C++ για να προσδώσουμε επεκτασιμότητα στον κώδικά μας.
- Αξιοποιήσαμε τις predefined optimization parameters του g++.
- Η κλάση my_vector αποτελείται από ένα string id (ώστε είτε είναι αριθμός είτε string, να το χειριζόμαστε με τον ίδιο τρόπο) και ένα std::vector με τις τιμές των συντεταγμένων του διανύσματος. Έχει απλές συναρτήσεις constructor, destructor, getters, setters και την `get_id_as_int` που επιστρέφει την int τιμή του id του διανύσματος εν πάσει περιπτώσι.
- Η κλάση των συναρτήσεων h έχει την τιμή του m (hard coded σε κάθε εκτέλεση, σε μετέπειτα χρόνο αν χρειαστεί προστίθεται συνάρτηση εύρεσης της τιμής βάσει του τύπου που υπάρχει στις διαφάνειες), μία μεταβλητή για την τιμή του w, μία για το M, ένα διάνυσμα από s_i και τον αριθμό των διαστάσεων. Χρησιμοποιήσαμε δύο συναρτήσεις, μία βοηθητική και μία που αποτελεί την συνάρτηση h, επιστρέφοντας την τιμή της για κάποιο διάνυσμα. Η `individual_comp` συνάρτηση επιστρέφει κάθε $a_i \cdot m^{d-1-i}$ και η `actual_h_function` επιστρέφει την τιμή της h.
- Η κλάση των συναρτήσεων g περιέχει τον ακέραιο k και ένα διάνυσμα από h συναρτήσεις, από μεθόδους έχει constructors, destructor & την συνάρτηση g, που δοθέντος διανύσματος επιστρέφει τον ακέραιο που ζητείται.
- Η κλάση των hash tables για το ερώτημα αυτό περιέχει έναν ακέραιο που δηλώνει το μέγεθος κάθε table, ένα διάνυσμα από διανύσματα από ζεύγη που περιέχουν έναν δείκτη σε διάνυσμα και την αντίστοιχη τιμή που επιστρέφει η g για αυτό (γεγονός που χρειάζεται αφού για την εισαγωγή των vectors στα hash tables κάθε φορά κάνουμε την τιμή της g mod το μέγεθος του hash table), και μία συνάρτηση g. Έχουμε δύο συναρτήσεις, η μία αποθηκεύει σε κάποιο hash table, στο κατάλληλο bucket ένα input vector, και η άλλη επιστρέφει απλώς σε ποιο bucket θα κατέληγσε ένα query vector (ουσιαστικά, μία συνάρτηση αποθήκευσης και μία εύρεσης θέσης, όπως οφείλει να έχει κάθε hash table, με την διαφοροποίηση ότι η hash query επιστρέφει τα ids των πιθανών γειτόνων του `query_vector` σε αυτό το bucket και όχι το σε ποιο bucket θα αποθηκευόταν το query vector, χάριν ταχύτητας).
- Υλοποιήσαμε δική μας συνάρτηση mod, η οποία επιστρέφει το υπόλοιπο ενός αριθμού mod ενός άλλου, άρα ένα θετικό ακέραιο, όπως ζητήθηκε.
- Για να χωρούν στην μνήμη οι πράξεις που οφείλουν να πραγματοποιηθούν, χρησιμοποιήσαμε, όπως συζητήθηκε στην τάξη, modular arithmetic, φτιάχνοντας μία συνάρτηση υπολογισμού modular exponentiation με ύστατο στόχο την μείωση του χώρου που χρησιμοποιούμε.
- Σε κάθε ζητούμενη επανεκτέλεση του κώδικα (αν δηλαδή ο χρήστης ξαναζητήσει να τρέξει η κάθε μέθοδος), ζητούμε από τον χρήστη να δίνει εκ νέου κάθε παράμετρο που χρειάζεται κάθε μέθοδος. Σε περίπτωση λάθους, λήγει η επανεκτέλεση.
- Στο ερώτημα Β, σε ό,τι αφορά τις καμπύλες, η διάρθωση του κώδικα έχει ως εξής: η κλάση curve points αποτελείται από `std::pair` από δύο

συντεταγμένες (παραμετροποιημένες βάσει template). Έγινε η επιλογή αυτής της δομής ώστε σε τυχόν επανάχρηση του κώδικα, με αλλαγή της δομής αυτής (π.χ. σε vector από points ή σε ένα 1D point) να μην χρειάζεται να αλλάζει ολόκληρη η δομή της κλάσης των καμπυλών. Η τελευταία αποτελείται από ένα vector από curve points, έναν ακέραιο που αποθηκεύει το πλήθος αυτών, και ένα string id. Ο constructor της curves που λαμβάνει ως όρισμα string έχει παραμετροποιηθεί ακριβώς για τα δεδομένα της άσκησης και σε περίπτωση που αυτά αλλάξουν, οφείλει να ενημερωθεί η υλοποίησή του.

- Για τις τυχαίες προβολές, έχουμε υλοποιήσει δομές γεννητριών traversal μεταξύ δύο καμπυλών. Αυτές λειτουργούν βάση δενδρικής δομής, όπου κάθε κόμβος έχει ένα ζεύγος ακεραίων τιμών (που υποδεικνύουν τους δείκτες του traversal – το πρώτο στοιχείο δείχνει σε ποιο σημείο της καμπύλης 1 αναφερόμεαστε, και το δεύτερο σε ποιο σημείο της καμπύλης 2), και τρεις δείκτες σε δενδρικούς κόμβους: ο αριστερός (υποδηλώνει ότι έχει μετατοπιστεί κατά 1 ο δείκτης που αναφέρεται στην καμπύλη 1), ο δεξιός (υποδηλώνει ότι έχει μετατοπιστεί κατά 1 ο δείκτης που αναφέρεται στην καμπύλη 2) και ο κεντρικός (υποδηλώνει ότι έχουν μετατοπιστεί κατά 1 και οι 2 δείκτες). Ακολουθεί επεξηγηματικό σχήμα:



- Κάθε traversal tree αποτελείται από traversal nodes όπως αναφέρθηκε ανωτέρω. Κάθε traversal tree γεννά ένα δένδρο που παρουσιάζει κάθε πιθανό traversal μεταξύ δύο καμπυλών.
- Με αναδρομική αναζήτηση σε αυτό το δένδρο μπορούν να επιστραφούν όλα τα traversals ως αντικείμενα της κλάσης traversal, όπου κρατώνται ένα ζεύγος από string (τα id των δύο καμπυλών), και ένας vector από ζεύγη ακεραίων, που αφορούν στην αλληλουχία των ζεύυγών indices του εκάστοτε traversal. Υποσημειώνεται ότι αυτή η λειτουργία είναι εκτενέστατη και απαιτητική σε επεξεργαστική ισχύ.

LSH για διανύσματα

Χρησιμοποιούμε τα αρχεία:

- `Makefile`
- `lsh.cpp`
- `h_funs.h/h_funs.cpp`
- `g_funs.h/g_funs.cpp`
- `ht.h/ht.cpp`
- `my_vector.h/my_vector.cpp`
- `NNpair.h/NNpair.cpp`
- `utils.h`

Οδηγίες μεταγλωττισης:
Στον φάκελο της εργασίας εκτελούμε τα εξής:

- `$make`
  Παράγει το εκτελέσιμο αρχείο (και τα .o αρχεία)
- `$make clean`
  Διαγράφει το εκτελέσιμο αρχείο (και τα .o αρχεία)

Αρχικά διαβάζουμε κατάλληλα την γραμμή εντολών, αναθέτουμε τιμές όπου αυτές δίνονται και όπου όχι είτε λαμβάνουν τις default τιμές τους (π.χ. k, L) είτε ζητούμε από τον χρήστη να γράψει τις τιμές που επιθυμεί (π.χ. paths για τα αρχεία).

Έπειτα διαβάζουμε τα αρχεία εισόδου (input) και αναζήτησης (query) και, αφότου κρατήσουμε το πλήθος των vectors σε κάθε περίπτωση, βρίσκουμε το distance matrix αυτών και με brute force τον εκάστοτε πλησιέστερο γείτονα κάθε query vector.

Ακόμη, υπολογίζουμε για τα input vectors τις αποστάσεις τους από τους πλησιέστερους γείτονές τους αναμεταξύ τους, βρίσκουμε τη μέση τιμή αυτών των αποστάσεων ώστε να ορίσουμε την μεταβλητή w όπως ειπώθηκε στην τάξη.

Σε αυτό το στάδιο αρχίζει η εκτέλεση του αλγορίθμου LSH. Δημιουργούμε ένα vector με τα L hashtables μας, όπου βάζουμε όλα τα input vectors με τον κατάλληλο τρόπο. Έπειτα για κάθε query vector, βρίσκουμε τους πιθανούς γείτονές του (αρκεί να βρίσκονται στο ίδιο bucket σε κάποιο από τα L hash tables, και να έχουν την ίδια τιμή από την συνάρτηση g για αυτό το hash table). Παίρνουμε τους πιθανούς αυτούς γείτονες και εκτελούμε συγκρίσεις των αποστάσεών τους με το query vector, κρατάμε τον aproximate NN σε δομή NNpair και συνεχίζουμε στο επόμενο query vector. Να σημειωθεί ότι έχουμε θέσει ως όριο να εκτελούνται μέχρι 3·L τέτοιοι υπολογισμοί, όπως συζητήθηκε στην τάξη.

Σχεδιαστικές επιλογές:

- Για να διευκολύνουμε την εκτέλεση του προγράμματός μας, δημιουργούμε έναν distance matrix, δηλαδή έναν πίνακα από αποστάσεις μεταξύ των inpt & query vectors, όπου κάθε θέση του πίνακα `[i][j]` έχει αποθηκευμένη μέσα την απόσταση μεταξύ του query vector με id i-1 και του input vector με id j-1. Έτσι διευκολύνουμε την αναζήτηση actual NN μεταξύ των διανυσμάτων, αλλά και μπορούμε να συγκρίνουμε ταχύτατα (O(1)) την απόσταση του actual NN με την ευρεθείσα απόσταση του aproximate NN του εκάστοτε διανύσματος.

Hypercube για διανύσματα

Χρησιμοποιούμε τα αρχεία:

- `Makefile`
- `cube.cpp`
- `h_funs.h/h_funs.cpp`
- `g_funs.h/g_funs.cpp`
- `cube_ht.h/cube_ht.cpp`
- `my_vector.h/my_vector.cpp`
- `NNpair.h/NNpair.cpp`
- `utils.h`

Οδηγίες μεταγλωττισης:
Στον φάκελο της εργασίας εκτελούμε τα εξής:

- `$make a1`
  Παράγει το εκτελέσιμο αρχείο (και τα .o αρχεία)
- `$make clean`
  Διαγράφει το εκτελέσιμο αρχείο (και τα .o αρχεία)

Αρχικά διαβάζουμε κατάλληλα την γραμμή εντολών, αναθέτουμε τιμές όπου αυτές δίνονται και όπου όχι είτε λαμβάνουν τις default τιμές τους (π.χ. probes, M) είτε ζητούμε από τον χρήστη να γράψει τις τιμές που επιθυμεί (π.χ. paths για τα αρχεία).

Έπειτα διαβάζουμε τα αρχεία εισόδου (input) και αναζήτησης (query) και, αφότου κρατήσουμε το πλήθος των vectors σε κάθε περίπτωση, βρίσκουμε το distance matrix αυτών και με brute force τον εκάστοτε πλησιέστερο γείτονα κάθε query vector, όπως παραπάνω.

Ακόμη, υπολογίζουμε για τα input vectors τις αποστάσεις τους από τους πλησιέστερους γείτονές τους αναμεταξύ τους, βρίσκουμε τη μέση τιμή αυτών των αποστάσεων ώστε να ορίσουμε την μεταβλητή w όπως ειπώθηκε στην τάξη.

Σε αυτό το στάδιο αρχίζει η εκτέλεση του αλγορίθμου του Υπερκύβου. Εδώ έχουμε ένα hash table, που αναπαριστά τον υπερκύβο μας, και κάθε κορυφή του έχει μία τιμή βάσει ενός bitstring. Έχουμε υλοποιήσει την κλάση `cube_ht` που παρέχει την δομή αυτή, που σε αντίθεση με το hash table που χρησιμοποιούμε στο LSH, εδώ περνάει τις g από μία τυχαιοκρατική ανάθεση 0 ή 1 ώστε να δωθεί το κατάλληλο bit στην αντίστοιχη θέση του bitstring. Οι αναζητήσεις των approximate NNs γίνονται με αντίστοιχο τρόπο (βρίσκονται οι “συγκάτοικοι” ενός query vector στο bucket, ελέγχεται ο κοντινότερος & επιστρέφεται αυτός).

Σχεδιαστικές επιλογές:

- Ο distance matrix πραγματοποιείται όπως και στο προηγούμενο ερώτημα.

- Το hash table μας τώρα αποτελείται από vectors από buckets που διατηρούν δείκτες σε δομές `my_vector`. Ο hash table υλοποιείται με την χρήση unordered map, ώστε να επιλύεται κατ'αυτόν τον τρόπο το πρόβλημα τεράστιων πινάκων με άδειες θέσεις (δεδομένου ότι μέγεθος πίνακα = $2^{d'}$)
- Χρησιμοποιούμε ένα vector από `fi_seeds`, δηλαδή από seeds που χαρακτηρίζουν την τυχαία απόδοση 0 ή 1 στα αποτελέσματα των g.
- Η μεταβλητή `verticier` της κλάσης `cube_ht` είναι ένας πίνακας από string, βοηθητικής φύσης, που μας βοηθάει στην αναζήτηση και καταγραφή των κατά hamming distance διαφορετικών bitstrings που αντιστοιχούν σε άλλες κορυφές του υπερκύβου που θα αναζητηθούν βάσει της μεταβλητής `probes`.
- Εφ'όσον εδώ ελέγχουμε `probes` το πλήθος κορυφές, όταν ελέγξουμε την κορυφή στην οποία βρίσκεται ήδη το query vector, ελέγχουμε `probes-1` γειτονικές κορυφές, βρίκοντας αρχικά κάθε πιθανό bitstring με hamming distance>0, τα περνάμε από την μεταβλητή `verticier` σειριακά (οπότε πρώτα θα πάνε αυτά με απόσταση 1, μετά με 2 κοκ), και έτσι διατηρούμε τα bitstrings με απόσταση 1 εν τέλει, τα οποία και ελέγχουμε ανάλογα με την μεταβλητή `M` (αν οι πιθανοί γείτονες είναι λιγότεροι από `M` ή ίσοι με `M`, ελέγχονται όλοι τους, ειδάλλως ελέγχουμε μέχρι `M` πιθανούς γείτονες).
- Η ανελαστική φύση των f_i και ο συμβιβασμός στο πρότυπο της uniform distribution που ζητήθηκε μπορεί, κατά την εκτέλεση του υπερκύβου, να συμβάλλουν στον μεγάλο συνωστισμό πολλών διανυσμάτων σε ίδια bucket του υπερκύβου, με αποτέλεσμα ένα query vector που θα βρεθεί σε απομονωμένο bucket να μην εμφανίσει γείτονα για πολύ μικρές τιμές των παραμέτρων M και probes.

LSH για πολυγωνικές καμπύλες (LSH στα διανύσματα που προκύπτουν)

Χρησιμοποιούμε τα αρχεία:

- `Makefile`
- `curve_grid_lsh.cpp`
- `curve_point.h/curve_point.cpp`
- `curve.h/curve.cpp`
- `pan_traversal.hpp`
- `grid.h/grid.cpp`
- `h_funs.h/h_funs.cpp`
- `g_funs.h/g_funs.cpp`
- `my_vector.h/my_vector.cpp`
- `NNpair.h/NNpair.cpp`
- `utils.h`

Οδηγίες μεταγλωττισης:
Στον φάκελο της εργασίας εκτελούμε τα εξής:

- `$make a3`
  Παράγει το εκτελέσιμο αρχείο (και τα .o αρχεία)
- `$make clean`
  Διαγράφει το εκτελέσιμο αρχείο (και τα .o αρχεία)

Αρχικά διαβάζουμε κατάλληλα την γραμμή εντολών, αναθέτουμε τιμές όπου αυτές δίνονται και όπου όχι είτε λαμβάνουν τις default τιμές τους (π.χ. k_vec, L_grid) είτε ζητούμε από τον χρήστη να γράψει τις τιμές που επιθυμεί (π.χ. paths για τα αρχεία).

Έπειτα κρατάμε σε ένα vector από καμπύλες τις input καμπύλες και διατηρούμε τις 86 τελευταίες ως query καμπύλες.

Κατά την μετατροπή μιας καμπύλης σε grid καμπύλη, τις περνάμε μέσω μιας συνάρτησης του grid, που λέγεται gridify. Έπειτα αυτές περνώνται από την συνάρτηση vectorify, που τις μετατρέπει, όπως ειπώθηκε στην τάξη, σε ένα διάνυσμα. Σε κάθε τέτοιο βήμα διατηρούμε το id της αρχικής καμπύλης ώστε να διατηρούμε επαφή με την αρχική καμπύλη. Αυτό το διάνυσμα έπειτα το συμπληρώνουμε (pad) με τιμή που υπολογίζεται από τη main μέσω των καμπυλών του dataset ως μεγαλύτερη από την μέγιστη συντεταγμένη οποιουδήποτε στοιχείου του dataset. Αυτό γίνεται με σκοπό να μην αλλοιώνονται τα αποτελέσματα των αλγορίθμων.

Τα padded διανύσματα αυτά περνώνται από τον αλγόριθμο LSH όπως στο ερώτημα Α, αλλά τώρα στο hash table αποθηκεύονται οι αρχικές καμπύλες που αντιστοιχούν σε αυτό το διάνυσμα. Έπειτα οι συγκρίσεις των καμπυλών για την εύρεση κάθε γείτονα γίνεται με χρήση της DTW συνάρτησης, κατ’ αντιστοιχίαν με

την μετρική που είχαμε ανωτέρω (Η DTW *δεν* αποτελεί μετρική βάσει ορισμού, απλώς λειτουργεί ως τέτοια).

Σχεδιαστικές επιλογές:

- Δεν διατηρούμε αυτό καθαυτό το grid στη μνήμη, μόνο τις παραμέτρους του (t, δ).
- Έχουμε υλοποιήσει την μετρική DTW βάσει του δοθέντος αλγορίθμου, στο αρχείο `pan_traversal.hpp`, η οποία υπολογίζει την απόσταση DTW μεταξύ δύο καμπυλών.
- Έχουμε υλοποιήσει την συνάρτηση εύρεσης ευκλίδειας απόστασης μεταξύ δύο curve points συγκεκριμένα.

LSH για πολυγωνικές καμπύλες (Hypercube στα διανύσματα που προκύπτουν)

Χρησιμοποιούμε τα αρχεία:

- `Makefile`
- `curve_grid_hypercube.cpp`
- `curve_point.h/curve_point.cpp`
- `curve.h/curve.cpp`
- `pan_traversal.hpp`
- `cube_grid.h/cube_grid.cpp`
- `h_funs.h/h_funs.cpp`
- `g_funs.h/g_funs.cpp`
- `my_vector.h/my_vector.cpp`
- `NNpair.h/NNpair.cpp`
- `utils.h`

Οδηγίες μεταγλωττισης:
Στον φάκελο της εργασίας εκτελούμε τα εξής:

- `$make a2`
  Παράγει το εκτελέσιμο αρχείο (και τα .o αρχεία)
- `$make clean`
  Διαγράφει το εκτελέσιμο αρχείο (και τα .o αρχεία)

Αρχικά διαβάζουμε κατάλληλα την γραμμή εντολών, αναθέτουμε τιμές όπου αυτές δίνονται και όπου όχι είτε λαμβάνουν τις default τιμές τους (π.χ. k_vec, L_grid) είτε ζητούμε από τον χρήστη να γράψει τις τιμές που επιθυμεί (π.χ. paths για τα αρχεία).

Έπειτα κρατάμε σε ένα vector από καμπύλες τις input καμπύλες και διατηρούμε τις 86 τελευταίες ως query καμπύλες.

Η μετατροπή μιας καμπύλης σε grid καμπύλη, και τέλος σε ένα διάνυσμα γίνεται όπως και πριν.

Τα padded διανύσματα περνώνται από τον αλγόριθμο hypercube όπως στο ερώτημα Α, αλλά τώρα στο hash table/unordered map αποθηκεύονται οι αρχικές καμπύλες που αντιστοιχούν σε αυτό το διάνυσμα. Έπειτα οι συγκρίσεις των καμπυλών για την εύρεση κάθε γείτονα γίνεται με χρήση της DTW συνάρτησης, κατ' αντιστοιχίαν με την μετρική που είχαμε ανωτέρω (Η DTW *δεν* αποτελεί μετρική βάσει ορισμού, απλώς λειτουργεί ως τέτοια).

Τυχαία προβολή για πολυγωνικές καμπύλες (LSH στα διανύσματα που προκύπτουν)

Δεν υλοποιήθηκε πλήρως.

Τυχαία προβολή για πολυγωνικές καμπύλες (Hypercube στα διανύσματα που προκύπτουν)

Δεν υλοποιήθηκε πλήρως.

Μετρήσεις & Πειράματα

Απουσία υλοποίησης ολόκληρων των μεθόδων τυχαίων προβολών (εξίσου με LSH & με υπερκύβο) δεν έγιναν μετρήσεις.

	LSH καμπυλών με LSH L1	**LSH καμπυλών με Hypercube**
*Μέγιστο κλάσμα προσέγγισης**	0.148226	0.313206
Μέσος χρόνος εύρεσης approximate NN	0.00157474	0.0398965

** = μέση απόσταση approximate NNs προς μέση απόσταση actual NNs*

Παρατηρήσεις – Συμπεράσματα

Ειναι σημαντικό να σημειωθεί ότι το πρόγραμμά μας έχει φτιαχτεί με απώτερο σκοπό την μελέτη των αλγορίθμων για big data, οπότε όταν δεν δίνεται μεγάλο μέγεθος στοιχείων, η ακρίβεια κυμαίνεται πιο έντονα από ότι σε μεγαλύτερου μεγέθους δεδομένα.

Εν γένει λόγω των δοθέντων τιμών των παραμέτρων, ο αλγόριθμος του υπερκύβου επιστρέφει λιγότερο επιτυχώς αντιστοιχισμένα ζεύγη nearest neighbours, συγκρινόμενος με την μέθοδο LSH.

Σε εξαιρετικά μικρά datasets οι brute ευρέσεις γειτόνων είναι πιο γρήγορες, ενώ στα datasets που δώθηκαν για μελέτη η διαφορά είναι εμφανής, καθώς προφανώς οι μέθοδοι που υλοποιήσαμε είναι πολύ πιο αποδοτικοί χρονικά, εφόσον έχουν δημιουργηθεί για big data.

Βιβλιογραφία

- http://www.cplusplus.com/
- https://en.cppreference.com/
- Σημειώσεις του μαθήματος
- https://www.geeksforgeeks.org/
- DTW Tutorial