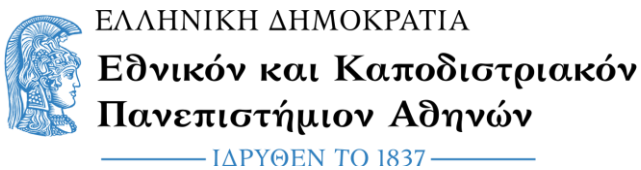
Κ23α – Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Χειμερινό Εξάμηνο 2024–2025



Κινδύνης Μάριος (sdi2000223) και Κορρές Μιχαήλ (sdi2000092)

1. Εισαγωγή

1.1 Σύντομη Περιγραφή του Project

Η **Αναζήτηση Εγγύτερου Γείτονα** (Nearest Neighbor Search - NNS) αποτελεί ένα θεμελιώδες πρόβλημα στον τομέα της Πληροφορικής και των Πληροφοριακών Συστημάτων. Στόχος της είναι η εύρεση του σημείου σε ένα δεδομένο σύνολο δεδομένων που είναι πιο κοντά ή πιο παρόμοιο με ένα συγκεκριμένο ερώτημα, σύμφωνα με μια προκαθορισμένη μετρική απόστασης. Η εφαρμογή της NNS εκτείνεται σε ποικίλους τομείς όπως η μηχανική μάθηση, η αναγνώριση προτύπων, η αναζήτηση εικόνων και δεδομένων, καθώς και στην ανάλυση μεγάλων δεδομένων.

Το παρόν project αφορά το μάθημα **"Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα"** και επικεντρώνεται στον σχεδιασμό, την υλοποίηση και τη βελτιστοποίηση αλγορίθμων αναζήτησης εγγύτερων γειτόνων. Συγκεκριμένα, οι αλγόριθμοι **Vamana, Filtered Vamana** και **Stitched Vamana** υλοποιήθηκαν χρησιμοποιώντας τις γλώσσες προγραμματισμού **C++**, στοχεύοντας στην αποτελεσματική διαχείριση μεγάλων και

πολυδιάστατων συνόλων δεδομένων.

1.2 Στάδια εργασία

1. **Άσκηση 1**: Θεμελίωση της βασικής δομής του αλγορίθμου Vamana και υλοποίηση της αναζήτησης εγγύτερων γειτόνων (GreedySearch, RobustPrune, κ.λπ.).
2. **Άσκηση 2**: Προσθήκη και υποστήριξη φιλτραρισμένων ερωτημάτων μέσω του Filtered Vamana και του Stitched Vamana.
3. **Άσκηση 3**: Βελτιστοποιήσεις και παραλληλοποίηση του κώδικα, με έμφαση στη μείωση χρόνου εκτέλεσης, μνήμης και αύξηση της ταχύτητας απόκρισης.

Η παρούσα αναφορά περιλαμβάνει μια ολοκληρωμένη παρουσίαση της διαδικασίας υλοποίησης και βελτιστοποίησης, καθώς και τα πειραματικά αποτελέσματα που αποδεικνύουν την αποτελεσματικότητα των προτεινόμενων βελτιώσεων.

2. Θεωρητικό Υπόβαθρο & Επισκόπηση

2.1 Θεωρία της Αναζήτησης Εγγύτερων Γειτόνων

Η αναζήτηση εγγύτερου γείτονα (Nearest Neighbor Search - NNS) είναι ένα βασικό πρόβλημα στην επιστήμη των υπολογιστών και τη μηχανική μάθηση, όπου στόχος είναι να εντοπιστούν τα πιο κοντινά σημεία σε ένα δεδομένο σημείο-ερώτημα q μέσα σε ένα σύνολο δεδομένων P. Το πρόβλημα αυτό εφαρμόζεται σε εφαρμογές όπως συστήματα συστάσεων, ανάκτηση εικόνων, ανίχνευση ανωμαλιών και βιομετρική αναγνώριση.

Με την αύξηση της διάστασης και του όγκου των δεδομένων, η παραδοσιακή μέθοδος γραμμικής σάρωσης γίνεται αναποτελεσματική, γεγονός που οδήγησε στην ανάπτυξη προσεγγιστικών αλγορίθμων (Approximate Nearest Neighbor Search - ANNS). Αυτοί οι αλγόριθμοι επιτυγχάνουν σημαντική μείωση του χρόνου αναζήτησης θυσιάζοντας με κάποιο βαθμό την ακρίβεια.

3. Υλοποίηση Άσκησης 1 & 2 (Implementation Details)

3.1 Δομή Κώδικα

Ο κώδικας οργανώθηκε σε επιμέρους αρχεία (cpp/h) και κλάσεις, εξυπηρετώντας τη δημιουργία ενός ευέλικτου συστήματος για Approximate Nearest Neighbor Search. Η διαδικασία εκτέλεσης αυτοματοποιείται μέσω Makefile (π.χ., στόχοι ann, filterann, groundtruthcalc) και προσαρμοσμένων ρυθμίσεων (<TARGET>.mk.config).

3.2 Αλγόριθμος Vamana (Άσκηση 1)

1. *Αρχικοποίηση*: Προσθήκη τυχαίων ακμών, επιλογή ή υπολογισμός ενός medoid ως σημείο εκκίνησης.
2. *GreedySearch*: Ξεκινώντας από το medoid, αναζητά υποψήφιους κόμβους σε κάθε βήμα, μετακινείται στον πλησιέστερο και επαναλαμβάνει.
3. *RobustPrune*: Για κάθε κόμβο, κλαδεύει ακμές ώστε να παραμένουν μόνο οι πιο σχετικές, περιορίζοντας τον βαθμό R.

3.3 Filtered Vamana (Άσκηση 2)

1. *Λογική Φίλτρων*: Κάθε σημείο φέρει ετικέτες ο γράφος και οι αναζητήσεις λαμβάνουν υπόψη μόνο τους κόμβους με κοινές ετικέτες.
2. *FilteredGreedySearch*: Εξετάζει μόνο κόμβους που πληρούν το φίλτρο, περνώντας από τον γράφο με τρόπο παρόμοιο με τον κλασικό GreedySearch.
3. *FilteredRobustPrune*: Κλαδεύει ακμές μεταξύ κόμβων που δεν μοιράζονται τις ίδιες ετικέτες, διατηρώντας ικανοποιητική συνδεσιμότητα για κάθε φίλτρο.
4. *Stitched Vamana*: Εναλλακτική υλοποίηση που συναρμόζει (stitches) πολλαπλούς επιμέρους γραφήματα (ένα ανά φίλτρο) σε ένα ενιαίο ευρετήριο.

3.4 Διαφορές με τον Κλασικό Vamana

- Κατά το indexing, οι ακμές συνδέουν μόνο κόμβους με κοινές ετικέτες.
- Ο αλγόριθμος αναζήτησης φιλτράρει τους γείτονες, επιταχύνοντας την εύρεση κατάλληλων σημείων.

3.5 Ορθότητας

- *Test Datasets*: Συγκρίθηκε η απόδοση του Vamana με brute force σε μικρά υποσύνολα.
- *Unit Tests*: Για κάθε βασική λειτουργία δημιουργήθηκαν αυτόνομα tests με τη βιβλιοθήκη acutest.h.

- *Benchmarking*: Τρέξαμε το σύστημα σε datasets με γνωστό ground truth, μετρώντας recall για διαφορετικές παραμέτρους.
- *Manual Verification*: Περαιτέρω επιβεβαίωση σε σενάρια με φίλτρα, ώστε να διαπιστωθεί ότι ο γράφος και η αναζήτηση ανταποκρίνονται σωστά.

3.6 Δομές που χρησιμοποιήθηκαν

3.6.1 FilterGraph

Η κλάση αυτή αποτελεί τον πυρήνα της υλοποίησης του συστήματος.

Τα βασικά πεδία της κλάσης είναι:

1. **numOfThreads**: Ο αριθμός των νημάτων που θα χρησιμοποιηθούν για την εκτέλεση παράλληλων εργασιών.
2. **numOfDatapoints**: Ο συνολικός αριθμός των δεδομένων (σημείων) στον γράφο.
3. **vertexMap**: Map που συνδέει κάθε id με το σημείο δεδομένων (id ->Datapoint)
4. **g**: Map που αναπαριστά τις ακμές του γράφου, συνδέοντας τα σημεία με τις αντίστοιχες ακμές (id -> [Edges]).
5. **L**: Η μέγιστη χωρητικότητα του container που χρησιμοποιείται από τους αλγορίθμους.
6. **R**: Ο μέγιστος αριθμός εξερχόμενων ακμών από κάθε κόμβο.
7. **k**: Ο αριθμός των γειτόνων που θα βρεθούν σε αλγορίθμους αναζήτησης.
8. **a**: Παράμετρος για το αλγόριθμο **RobustPrune** (κατώφλι απόστασης).
9. **tau**: Παράμετρος για τη δειγματοληψία στον γράφο.
10. **filters**: Λίστα φίλτρων (π.χ., κατηγορίες ή κλάσεις) που χρησιμοποιούνται για την επιλογή των δεδομένων.
11. **numOfFilters**: Ο αριθμός των φίλτρων που υπάρχουν στον γράφο.
12. **ids**: Λίστα με τα IDs των σημείων δεδομένων.

3.6.2 VamanaContainer

Χρησιμοποιείται για την αποθήκευση και διαχείριση ενός συνόλου από ζεύγη (ID, Απόσταση), με περιορισμένο μέγεθος. Η κλάση αυτή παρέχει τις εξής βασικές λειτουργίες:

1. **insert(const pair<int,double>& newItem)**: Εισάγει ένα νέο στοιχείο (ζεύγος ID και Απόστασης) στην κοντέινερ. Αν το στοιχείο υπάρχει ήδη ή αν η απόσταση του νέου στοιχείου είναι μικρότερη από την μεγαλύτερη αποθηκευμένη απόσταση (και ο χώρος είναι γεμάτος), το στοιχείο δεν προστίθεται. Τα στοιχεία

ταξινομούνται πάντα κατά αύξουσα απόσταση και, αν το μέγεθος υπερβεί το μέγιστο, το μεγαλύτερο στοιχείο αφαιρείται.

2. **contains(int id) const**: Ελέγχει αν υπάρχει ήδη το συγκεκριμένο ID στο σύνολο.
3. **subset(int k)**: Επιστρέφει τα πρώτα k στοιχεία (ή όλα τα στοιχεία αν k είναι -1), δηλαδή τα IDs των πιο κοντινών στοιχείων.
4. **print() const**: Εκτυπώνει τα IDs όλων των αποθηκευμένων στοιχείων στην κοντέινερ.

3.6.3 Dataset – FilterDataset – FilterQuerySet

Πρόκειται για απλές υλοποιήσεις που χρησιμοποιούνται για την είσοδο και αποθήκευση των δεδομένων από τα αντίστοιχα αρχεία εισόδου.

3.6.4 Datapoint – Query

Πρόκειται για δομές που μοντελοποιούν τις εγγραφές (σημεία δεδομένων) του ευρετηρίου, καθώς επίσης και τα αντίστοιχα ερωτήματα που δέχεται.

4. Βελτιστοποιήσεις & Παραλληλοποίηση (Άσκηση 3)

4.1 Περιγραφή Βελτιστοποιήσεων

4.1.1 Αρχικοποίηση Γράφου και τυχαία medoids

Για τη βελτίωση της ταχύτητας κατασκευής του γράφου, εισάγαμε την αρχικοποίηση με **τυχαίες ακμές** αντί για κενή αρχική κατάσταση. Αυτό επιτάχυνε τη σύγκλιση του GreedySearch, ειδικά σε περιπτώσεις όπου υπήρχαν ανεξάρτητοι υπογράφοι. Επιπλέον, εφαρμόσαμε **τυχαία medoids** αντί για τον υπολογισμό του πραγματικού medoid, μειώνοντας έτσι τον χρόνο αρχικοποίησης χωρίς σημαντική απώλεια ακρίβειας.

4.1.2 Ελαχιστοποίηση Υπολογισμών

Εφαρμόσαμε **caching αποστάσεων** για την αποθήκευση των ήδη υπολογισμένων αποστάσεων μεταξύ σημείων, μειώνοντας έτσι τον αριθμό επαναλαμβανόμενων υπολογισμών. Επιπλέον, βελτιώσαμε τη λειτουργία RobustPrune ώστε να μειώνεται η πολυπλοκότητα κατά τη διάρκεια της κατασκευής του γράφου, εφαρμόζοντας πιο αποδοτικά κριτήρια κλαδέματος.

4.2 Παραλληλοποίηση

Σε αυτό το στάδιο, εφαρμόσαμε τεχνικές παράλληλης εκτέλεσης με τη χρήση του **OpenMP**. Εστιάσαμε κυρίως στην παράλληλη κατασκευή του ευρετηρίου (index build) και στην ταυτόχρονη επεξεργασία πολλαπλών ερωτημάτων. Η χρήση οδηγιών τύπου #pragma omp parallel for επέτρεψε τη διανομή του φόρτου εργασίας σε πολλά νήματα, συμβάλλοντας στη μείωση του συνολικού χρόνου. Ωστόσο, απαιτήθηκε προσεκτική διαχείριση του **συγχρονισμού** σε κοινές δομές δεδομένων, ώστε να αποφευχθούν φαινόμενα race conditions και να διατηρηθεί η ορθότητα του αλγορίθμου.

5. Πειραματικά Αποτελέσματα (Experiments & Results)

5.1 Αρχικοποίηση Γράφου και Τυχαία Medoids

Στο πλαίσιο της βελτιστοποίησης της αρχικοποίησης του γράφου, πραγματοποιήθηκαν διάφορες δοκιμές για την αξιολόγηση της επίδρασης διαφορετικών μεθόδων αρχικοποίησης στην απόδοση του αλγορίθμου **Filtered Vamana**. Συγκεκριμένα, εξετάστηκαν οι εξής μέθοδοι, πάντα με τις ίδιες παραμέτρους ($k=100$, $L=250$, $R=60$, $a=1.2$):

1. **Βασική Υλοποίηση (Χωρίς Βελτιστοποιήσεις)**
2. **Προσθήκη Τυχαίων Ακμών στην Αρχικοποίηση**
3. **Αρχικοποίηση με Τυχαίο Medoid**
4. **Συνδυασμός Τυχαίων Ακμών και Τυχαίου Medoid**

5.1.1 Ανάλυση Αποτελεσμάτων

Μέθοδος Αρχικοποίησης	**Χρόνος Κατασκευής (ms)**	**Χρόνος Queries (ms)**	**Filtered k-recall@k (%)**
Βασική Υλοποίηση	464,003	60,736	98.92%
Τυχαίες Ακμές	632,135	58,387	98.87%
Τυχαίο Medoid	503,511	67,353	98.37%
Συνδυασμός Τυχαίων Ακμών και Medoid	646,074	67,697	98.43%

5.1.2 Συμπεράσματα

1. **Προσθήκη Τυχαίων Ακμών**: Αν και παρατηρήθηκε σημαντική αύξηση στον χρόνο κατασκευής του γράφου, ο χρόνος εκτέλεσης των queries μειώθηκε ελαφρώς, χωρίς αισθητή απώλεια στην ακρίβεια.
2. **Αρχικοποίηση με Τυχαίο Medoid:** Η μέθοδος αυτή μείωσε τον χρόνο κατασκευής σε σχέση με την προσθήκη τυχαίων ακμών, αλλά αύξησε τον χρόνο εκτέλεσης των queries και ελάττωσε ελαφρώς την ακρίβεια των filtered queries.
3. **Συνδυασμός Τυχαίων Ακμών και Τυχαίου Medoid**: Ο συνδυασμός των δύο βελτιστοποιήσεων οδήγησε σε περαιτέρω αύξηση του χρόνου κατασκευής και των queries, ενώ η ακρίβεια παρέμεινε σχεδόν αμετάβλητη.

Συνολικά, οι βελτιστοποιήσεις στην αρχικοποίηση του γράφου έδειξαν μικρές βελτιώσεις στον χρόνο εκτέλεσης των queries, αλλά με αντίστοιχη αύξηση του χρόνου κατασκευής και ελαφρά μείωση της ακρίβειας. Ο συνδυασμός των δύο μεθόδων δεν απέδωσε τα αναμενόμενα οφέλη, γεγονός που υποδεικνύει ότι η επιλογή της κατάλληλης τεχνικής αρχικοποίησης εξαρτάται από τις συγκεκριμένες ανάγκες της εκάστοτε εφαρμογής.

5.2 Παραλληλοποίηση

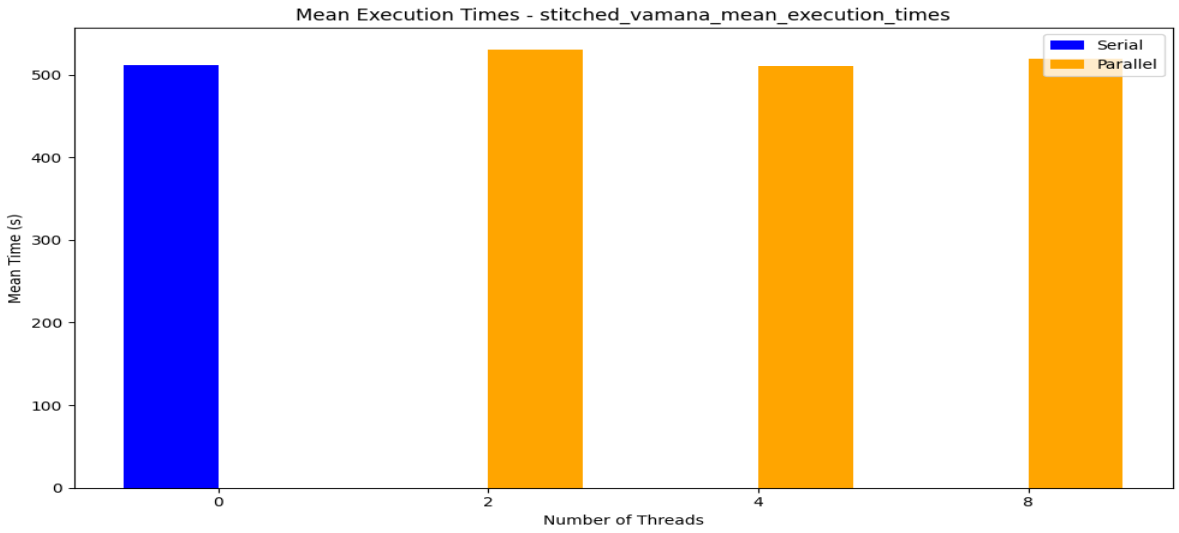
5.2.1 Περί Υλοποίησης

Χρησιμοποιήθηκε το δηλωτικό API του προτύπου OpenMP για 2,4 και 8 threads. Οι μετρήσεις αφορούν το χρόνο κατασκευής του ευρετηρίου.

5.2.2 Αποτελέσματα

Παραλληλοποιώντας τον αλγόριθμο Stitched Vamana,λαμβάνουμε τα εξής αποτελέσματα:

NUM_OF_THREADS	SERIAL_ALGORITHM_TIME	PARALLEL_ALGORITHM_TIME
σειριακός αλγόριθμος	511.5	0
2	0	530
4	0	510.5
8	0	519.5



5.2.3 Συμπεράσματα

Από τα αποτελέσματα φαίνεται ότι η παράλληλη υλοποίηση του Stitched Vamana δεν οδηγεί απαραίτητα σε σταθερή βελτίωση του χρόνου κατασκευής του ευρετηρίου συγκριτικά με την σειριακή εκτέλεση. Ενώ με 4 νήματα παρατηρείται ελαφρώς καλύτερη επίδοση (510.5 ms) σε σχέση με το σειριακό σενάριο (511.5 ms), η χρήση 2 ή 8 νημάτων δεν προσφέρει βελτίωση — και μάλιστα με 2 νήματα ο χρόνος αυξάνεται (530 ms). Αυτό υποδηλώνει ότι ο βαθμός επιτάχυνσης εξαρτάται έντονα από την ισορροπία μεταξύ του κόστους συγχρονισμού των νημάτων και της κατανομής του φόρτου εργασίας. Σε ορισμένες περιπτώσεις, το overhead της πολυνημάτωσης (π.χ. μηχανισμοί συγχρονισμού, κατανομή δεδομένων) μπορεί να αντισταθμίσει ή και να ξεπεράσει το όφελος της ταυτόχρονης εκτέλεσης. Ως εκ τούτου, η βέλτιστη επιλογή του αριθμού νημάτων θα πρέπει να αξιολογείται με βάση τις εκάστοτε ανάγκες και την αρχιτεκτονική του συστήματος.