Τελική Αναφορά της Εφαρμογής στο Μάθημα ‘Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα’

Χειμερινό Εξάμηνο 2024-2025

Μέλη ομάδας:

Κοντοχρήστος Χρήστος 1115202000090
Νικέλλη Εμμανουέλα 1115202000152

Πίνακας Περιεχομένων

1. Εισαγωγή

1.1 Γενικά

Στην αναφορά αυτή θα περιγράψουμε τον τρόπο με τον οποίο υλοποιήσαμε τον αλγόριθμο Vamana, για την εύρεση των Κ-κοντινότερων γειτόνων. Θα εξηγήσουμε τις σχεδιαστικές μας επιλογές, θα συγκρίνουμε την απόδοση σε χρόνο και σε μνήμη διαφορετικών προσεγγίσεων και θα εξετάσουμε βελτιστοποιήσεις με τη χρήση παραλληλοποίησης.

1.2 Η αναζήτηση Κ-κοντινότερων Γειτόνων (KNN)

Η αναζήτηση των Κ-κοντινότερων γειτόνων είναι ένας θεμελιώδης αλγόριθμος στους τομείς του data science, του machine learning αλλά έχει χρήση και σε λειτουργίες που αφορούν συστήματα συστάσεων. Ο λόγος που είναι τόσο σημαντικός, έγκειται στην ικανότητά του να εντοπίζει σχέσεις και μοτίβα μεταξύ στοιχείων ενός συνόλου δεδομένων. Στην αρχή, αυτή η μέθοδος αναζήτησης ήταν εξαντλητική, γεγονός που την καθιστούσε υπολογιστικά δαπανηρή για μεγάλα σύνολα δεδομένων. Ωστόσο, καθώς η τεχνολογία εξελίχθηκε και το μέγεθος των δεδομένων αυξήθηκε δραστικά, η ανάγκη εύρεσης μίας πιο αποδοτικής μεθόδου έγινε επιτακτική.

Έτσι, η μέθοδος αναζήτησης των Κ-κοντινότερων γειτόνων έχει εξελιχθεί σε ένα σημαντικό εργαλείο κατάλληλο να εφαρμοστεί σε μεγάλη κλίμακα δεδομένων. Σήμερα χρησιμοποιείται σε τομείς όπως η αναγνώριση εικόνας, η επεξεργασία φυσικής γλώσσας, τα συστήματα προτάσεων κ.α., στα οποία η ταχύτητα και ακρίβεια της αναζήτησης είναι μέγιστης σημασίας.

1.3 Ο Αλγόριθμος Vamana

Με βάση τα παραπάνω προκύπτει πως είναι απαραίτητο να δημιουργηθεί ένας κατάλληλος αλγόριθμος, ο οποίος να υλοποιεί την αναζήτηση των Κ-κοντινότερων γειτόνων με αποδοτικότητα. Αυτό ακριβώς πραγματοποιείται με τον αλγόριθμο Vamana, ο οποίος κάνει μία προσεγγιστική αναζήτηση των Κ-κοντινότερων γειτόνων, αντιμετωπίζοντας τις προαναφερόμενες υπολογιστικές προκλήσεις που προκύπτουν από τον μεγάλο όγκο δεδομένων. Ο αλγόριθμος Vamana, χρησιμοποιεί μία ιεραρχική δομή βασισμένη σε γράφους, γεγονός που επιταχύνει σημαντικά την αναζήτηση γειτόνων. Η βασική ιδέα του αλγορίθμου πηγάζει από την κατασκευή ενός πλοηγήσιμου γράφου, του οποίου οι κόμβοι είναι σημεία δεδομένων τα οποία συνδέονται μεταξύ τους με βάση την εγγύτητά τους στον χώρο χαρακτηριστικών. Με τον τρόπο αυτό επιτυγχάνονται υψηλά ποσοστά ανάκλησης (recall) με μικρό ελάχιστο υπολογιστικό κόστος και γενικά μεγάλη αποδοτικότητα.

Η ικανότητα του αλγορίθμου Vamana να εξισορροπεί ταχύτητα και απόδοση, τον καθιστά κατάλληλο για εφαρμογές πραγματικού χρόνου, όπως μηχανές αναζήτησης, συστήματα συστάσεων και την παράδοση δυναμικού περιεχομένου. Αποτελεί μία κρίσιμη καινοτομία στον τομέα της προσεγγιστικής αναζήτησης κοντινότερων γειτόνων, η οποία επιτρέπει την εξέλιξη τομέων, όπου οι εξαντλητικοί μέθοδοι αναζήτησης θα αποτύγχαναν λόγω της πολυπλοκότητας των δεδομένων.

2. Περιγραφή Αρχικής Υλοποίησης

Στην παράγραφο αυτή θα εξηγήσουμε την αρχική υλοποίηση του αλγορίθμου που πραγματοποιήσαμε (εργασίες 1 και 2). Θα περιγράψουμε την γενικότερη δομή του repository, τις βασικές δομές του προγράμματος, καθώς και τις σχεδιαστικές μας επιλογές (δομές δεδομένων και λογική) για την εκάστοτε συνάρτηση.

2.1 Δομή Repository

Η δομή του GitHub Repository μας έχει ως εξής:

- Αρχεία dataset.cpp/dataset.h:
  Τα δύο αυτά αρχεία αφορούν την κλάση dataset. Σκοπός της είναι να χειρίζεται τα input αρχεία, δηλαδή διαβάζει αρχεία τύπου .fvecs .ivecs .bvecs και τα αποθηκεύει σε ένα vector που αποτελείται από vectors με templated type για να μπορεί να αποθηκεύει τιμές int, float και unsigned char. Η συνάρτηση read_dataset() αναγνωρίζει το format του αρχείου, δηλαδή αν είναι fvecs, bvecs ή ivecs και καλεί τις read_fvecs, read_bvecs, read_ivecs ανάλογα τι αρχείο είναι. Χρησιμοποιείται φτιάχνοντας ένα object της κλάσης δηλώνοντας τι τύπου θέλουμε να είναι δηλαδή: Datase< float > data; και για να διαβαστεί το αρχείο καλούμε, data.read_dataset();.

- Αρχεία filtered_dataset.cpp/ filtered_dataset.h:
  Τα αρχεία αυτά επιτελούν αντίστοιχη λειτουργικότητα με τα dataset αρχεία, με διαφορά ότι το dataset που παράγουν αφορά τα δεδομένα με φίλτρα και χρησιμοποιούνται μόνο για την Filtered και Stitched Vamana.

- Αρχεία Graph.cpp/Graph.h:
  Η κλάση RRGraph δημιουργεί και αποθηκεύει ένα τυχαίο R-regular graph σε adjacency list representation. Το adjacency list είναι ένας vector που αποτελείται από pointers σε ένα struct object Node, το οποίο περιέχει το id του node και ένα vector σε int οπού αποθηκεύει τις εξερχόμενες ακμές κάθε node.

- Αρχεία Vamana.cpp/Vamana.h:
  Τα αρχεία αυτά αφορούν την υλοποίηση του αλγόριθμου Vamana και περιέχουν συναρτήσεις δημιουργίας του Vamana index, τις συναρτήσεις greedy search, pruning, εύρεσης medoid, εύρεσης recall και ευκλείδειας απόστασης, καθώς και τις αντίστοιχες συναρτήσεις για την filtered και stitched Vamana.

Γενικά οι περισσότερες συναρτήσεις δέχονται το dataset/filtered_dataset για να μπορούν να πάρουν τις τιμές των vectors και για αυτό τον λόγο έχουν χρησιμοποιηθεί templates ώστε να δέχονται και τους 3 τύπους (ints, floats, unsigned chars).

2.2 Σχεδιαστικές Επιλογές

Στο σημείο αυτό, θα περιγράψουμε αναλυτικότερα τις σχεδιαστικές μας επιλογές για κάθε μία από τις βασικές συναρτήσεις του προγράμματος.

- Συνάρτηση find_medoid/ Filtered_Find_Medoid:

Η συνάρτηση αυτή βρίσκει το medoid από τα στοιχεία του dataset, διατρέχοντας όλα τα δεδομένα και υπολογίζοντας την ευκλείδεια απόσταση κάθε σημείου από τα υπόλοιπα, κρατώντας στο τέλος αυτό με την μικρότερη συνολικά. Στην περίπτωση των filtered δεδομένων, κατασκευάζεται ένα map που εν τέλη συνδέει φίλτρα με τα αντίστοιχα medoids. Για να το κάνει αυτό, δημιουργεί ένα δεύτερο map στο οποίο κρατά τις εμφανίσεις κάθε στοιχείου. Στη συνέχεια, για κάθε φίλτρο βρίσκει τα στοιχεία του dataset που αντιστοιχούν σε αυτό και βρίσκει υποψήφια medoids. Στο τέλος κρατάει τα σημεία εκείνα με τις λιγότερες εμφανίσεις και τα επιστρέφει με το αρχικό map για κάθε φίλτρο.

- Συνάρτηση GreedySearch/ FilteredGreedySearch:

Η συνάρτηση GreedySearch αναζήτα τους κ κοντινότερους γείτονες ενός δοσμένου query σημείου. Χρησιμοποιούμε ένα priority queue με min-heap για να αποθηκεύσουμε την απόσταση κάθε σημείου από το query, δύο unordered sets για να κρατάμε τους visited κόμβους και κόμβους ήδη στο min-heap και σε ένα vector τοποθετούμε το αποτέλεσμα. Διατρέχουμε το min-heap, παίρνοντας πάντα στοιχείο στην κορυφή (αφού είναι το μικρότερο) και βάζουμε τους γείτονες αυτού στο min-heap(εφόσον δεν είναι ήδη). Ακολουθούμε τη διαδικασία αυτή μέχρι να εξαντληθούν οι γείτονες και στο τέλος συμπληρώνουμε το vector με το αποτέλεσμα κρατώντας τα κ πρώτα στοιχεία του min-heap.

Για την περίπτωση των δεδομένων με φίλτρα, ακολουθήσαμε παρόμοια υλοποίηση, με διαφορά ότι αρχικά φτιάχνουμε ένα vector με ένα ζευγάρι τιμών, το οποίο vector αποθηκεύει σημεία δεδομένων με κοινό φίλτρο με το ζητούμενο και την απόστασή τους από αυτό. Στη συνέχεια ακολουθείται η ίδια διαδικασία με πριν, απλά στο vector που σχηματίσαμε και όχι σε ολόκληρο το σύνολο δεδομένων.

- Συνάρτηση RobustPruning/ FilteredRobustPruning:

Η συνάρτηση αυτή δέχεται ένα σύνολο με γείτονες ενός δοσμένου σημείου και κλαδεύει τις κορυφές αυτές που δεν χρειάζονται, επιστρέφοντας τους τελικούς γείτονες σε ένα vector. Σε ένα min-heap τοποθετούνται γείτονες από το αρχικό set με βάση την απόστασή τους από το query. Από το min-heap αυτό αφαιρείται το στοιχείο στην κορυφή και τοποθετείται στο vector με το τελικό αποτέλεσμα, αν δεν έχει μπει ήδη. Η διαδικασία συνεχίζεται μέχρι το τελικό αποτέλεσμα να έχει R στοιχεία και ταυτόχρονα ελέγχεται και η δοσμένη συνθήκη $(\alpha \cdot d(p^*,p') \leq d(p,p'))$.

Η FilteredRobustPruning ακολουθεί την ίδια ακριβώς διαδικασία, με μοναδική διαφορά ότι ελέγχει μία ακόμα συνθήκη πριν τοποθετήσει ένα στοιχείο στο τελικό αποτέλεσμα, η οποία είναι το στοιχείο να έχει το ίδιο φίλτρο με το query στοιχείο.

- Συνάρτηση Vamana_Index/ Filtered_Vamana_Index/ StitchedVamana:

  Η συνάρτηση Vamana_Index είναι υπεύθυνη για την κατασκευή του γράφου. Αρχικά κατασκευάζει έναν R-regular γράφο με όλα τα δεδομένα. Στην συνέχεια βρίσκει το medoid στοιχείο. Δημιουργεί ένα τυχαίο permutation στοιχείων και για κάθε στοιχείο αυτού, τρέχει τις συναρτήσεις greedy search και robust pruning ώστε να βελτιωθεί ο γράφος. Στη συνέχεια ανανεώνει τις συνδέσεις του γράφου με βάση τα αποτελέσματα των προηγούμενων αποτελεσμάτων. Τέλος, τρέχει ξανά το robust pruning για να μην υπάρχουν πάνω από R γείτονες για κάθε κόμβο και αυτό εξακριβωθεί, επιστρέφει τον τελικό γράφο.

  Η συνάρτηση Filtered_Vamana_Index αρχικά δημιουργεί έναν κενό γράφο. Στη συνέχεια κρατά το σύνολο φίλτρων σε ένα unordered set, δημιουργεί ένα τυχαίο permutation των στοιχείων του dataset και σε ένα map τοποθετεί συνδυασμούς φίλτρων με τα medoids τους. Έπειτα για κάθε στοιχείο του permutation τρέχει την filtered greedy search, ανανεώνει τι λίστα από visited κόμβους και τρέχει την filtered robust pruning. Μετά ελέγχει την ορθότητα των γειτόνων και τρέχει ξανά την filtered robust pruning εάν χρειάζεται. Τέλος επιστρέφει τον γράφο.

  Η συνάρτηση StitchedVamana δημιουργεί έναν υπογράφο για κάθε υπάρχον φίλτρο, και επιστρέφει ένα vector με όλους του γράφους. Αρχικά, σε ένα set τοποθετεί όλα τα πιθανά φίλτρα. Στη συνέχεια διατρέχει τα φίλτρα και για κάθε ένα τοποθετεί σε ένα προσωρινό vector όλα τα στοιχεία του dataset που έχουν το φίλτρο αυτό. Επάνω στο σύνολο αυτό τρέχει τον κλασικό αλγόριθμο Vamana και τον γράφο που δημιουργεί τον προσθέτει στη συλλογή από γράφους.

3. Αλγοριθμικές Βελτιστοποιήσεις

Καθ' όλη τη διάρκεια της υλοποίησης του προγράμματος, κάναμε διάφορες βελτιστοποιήσεις σε σημεία του κώδικα. Στο σημείο αυτό θα αναφέρουμε τις πιο σημαντικές (η χρήση παράλληλου προγραμματισμού θα αναφερθεί ξεχωριστά σε επόμενη παράγραφο).

Μία από τις βελτιστοποιήσεις που πραγματοποιήσαμε, ήταν η χρήση min-heap σε διάφορα σημεία του κώδικα, όπως οι συναρτήσεις robust pruning και greedy search. Η χρήση min-heap μείωσε σημαντικά τον χρόνο εύρεσης της ελάχιστης τιμής, καθώς δεν χρειαζόταν κάποιος περεταίρω υπολογισμός αυτού(το min στοιχείο σε ένα min-heap είναι πάντα το πρώτο) όπως για παράδειγμα ταξινόμηση.

Άλλη μία βελτιστοποίηση που υλοποιήσαμε είναι το early stopping στην ευκλείδεια απόσταση. Το early stopping στη μέτρηση της Ευκλείδειας απόστασης βοηθάει να μειωθεί ο υπολογιστικός φόρτος και να γίνει πιο αποδοτική η αναζήτηση, καθώς αποφεύγονται περιττοί υπολογισμοί όταν μια απόσταση ήδη ξεπερνάει το τρέχον ελάχιστο όριο. Επειδή τα διανύσματα είναι πολυδιάστατα ο υπολογισμός της Ευκλείδειας απόστασης μεταξύ δύο διανυσμάτων γίνεται σε ένα for loop που υπολογίζεται σταδιακά το άθροισμα των τετραγώνων των διαφορών. Σε κάθε βήμα του βρόχου, ελέγχεται αν η ενδιάμεση τιμή της απόστασης έχει ξεπεράσει την τιμή της μικρότερης απόστασης τότε σταματάει να υπολογίζει ολόκληρη την απόσταση, επιστρέφοντας μία μεγάλη τιμή και υπολογισμός σταματάει πρόωρα. Αποφυγή πλήρους υπολογισμού: Αν μια απόσταση είναι ήδη μεγαλύτερη από το ελάχιστο όριο που έχουμε βρει, δεν υπάρχει λόγος να υπολογίσουμε την απόσταση μέχρι τέλους, καθώς αυτή δεν μπορεί να είναι υποψήφια για τα αποτελέσματα. Αυτό επιφέρει εξοικονόμηση χρόνου σε συναρτήσεις όπως στη Greedy Search/Filtered Greedy Search και

Robust Pruning/ Filtered Robust Pruning που χρειάζεται σε κάποια στάδια να υπολογίσουν τη μικρότερη απόσταση μεταξύ κάποιων nodes. Το early stopping μειώνει τον συνολικό αριθμό πράξεων που απαιτούνται για αυτές τις συγκρίσεις, βελτιώνοντας την ταχύτητα των αλγορίθμων και επομένως των Vamana/Filtered Vamana αλγόριθμων χωρίς να επηρεάζει την ορθότητα των αποτελεσμάτων (recall).

Τέλος, για να αυξηθεί το recall στα unfiltered queries, βάζουμε στο τελικό search στην Filtered Greedy Search L = πλήθος των medoids * 10, το οποίο επιφέρει καλύτερη αλλά πιο χρονοβόρα αναζήτηση.

4. Χρήση Παραλληλοποίησης

Η παραλλληλοποίηση έγινε στις Filtered Vamana, Stitched Vamana και Find Medoid. Για την Παραλλληλοποίηση έχει χρησιμοποιηθεί OpenMP.

- Find Medoid: Η παραλληλοποιημένη συνάρτηση Filtered_Find_Medoid επιλέγει το "medoid" (αρχικό σημείο) για κάθε κατηγορία (filter) στο dataset. Χρησιμοποιεί την OpenMP για να εκμεταλλευτεί πολλούς πυρήνες του επεξεργαστή, καθιστώντας την πιο αποδοτική σε μεγάλα datasets. Η εξωτερική επανάληψη (που διατρέχει τις κατηγορίες filters) είναι παραλληλοποιημένη με: #pragma omp parallel for schedule(runtime). Χρησιμοποιείται η οδηγία schedule(runtime) που επιτρέπει δυναμική κατανομή των επαναλήψεων στα threads, εξισορροπώντας το φορτίο. Κάθε νήμα επεξεργάζεται μία ή περισσότερες κατηγορίες ανεξάρτητα. Αυτό μειώνει τον συνολικό χρόνο, καθώς οι κατηγορίες είναι ανεξάρτητες μεταξύ τους. Τα σημεία για κάθε filter υπολογίζονται τοπικά από κάθε νήμα, μειώνοντας την ανάγκη για συγχρονισμό. Χρησιμοποιούνται περιοχές #pragma omp critical για αποφυγή συγκρούσεων όταν: Διαβάζουμε και γράφουμε στον map T_map. Ενημερώνουμε τον map M_map (τελικά medoids). Η παραλληλοποίηση μειώνει τον συνολικό χρόνο εκτέλεσης.

- Filtered Vamana: Με τη χρήση της οδηγίας #pragma omp parallel, ο κώδικας δημιουργεί νήματα (threads) που εκτελούν ανεξάρτητα τις εργασίες τους. Το κύριο μέρος της εργασίας (δηλαδή το βρόχο for που επεξεργάζεται κάθε κόμβο i του dataset) διαμερίζεται στα threads μέσω της οδηγίας #pragma omp for. Οι μεταβλητές όπως local_visited είναι τοπικές για κάθε νήμα, ώστε να αποφεύγεται η σύγκρουση δεδομένων. Στην παραλληλοποιημένη έκδοση (Filtered_Vamana_Index_Parallel), πολλοί κόμβοι επεξεργάζονται ταυτόχρονα από διαφορετικά νήματα, μειώνοντας τον συνολικό χρόνο επεξεργασίας κατά τον αριθμό των νημάτων που χρησιμοποιούνται. Κάθε νήμα διαχειρίζεται το δικό του υποσύνολο κόμβων, και μόνο σε συγκεκριμένα σημεία γίνεται συγχρονισμός (π.χ., κατά την ενημέρωση των γειτόνων στον γράφο). Ο παραλληλοποιημένος αλγόριθμος Filtered Vamana μειώνει σημαντικά τον χρόνο εκτέλεσης χωρίς να επηρεάζει την ορθότητα των αποτελεσμάτων (recall).

- Stitched Vamana: Ο παραλληλοποιημένος αλγόριθμος Stitched Vamana δημιουργεί πολλαπλά subgraphs παράλληλα για διαφορετικά filters του dataset και τα επιστρέφει ως έναν πίνακα γραφημάτων. Χρησιμοποιείται το #pragma omp parallel for για την επανάληψη που επεξεργάζεται κάθε filter και δημιουργεί τον υπογράφο για αυτό το filter. Κάθε νήμα επεξεργάζεται μία κατηγορία (filter)

ανεξάρτητα από τις υπόλοιπες. Τα filters είναι ανεξάρτητα, άρα μπορούν να υπολογιστούν ταυτόχρονα χωρίς αλληλεπιδράσεις.

5. Στατιστικά Απόδοσης

Παραθέτουμε κάποια αποτελέσματα του προγράμματός μας για τις διαφορετικές εκδοχές του Vamana για το dataset των 10000 στοιχείων:

- Vamana:

Για L = 100, a = 1, R = 13, k = 100: 102 seconds average recall = 98.2%
Για L = 150, a = 1.1, R = 13, k = 100: 214 seconds average recall = 99.1%

- Filtered Vamana:

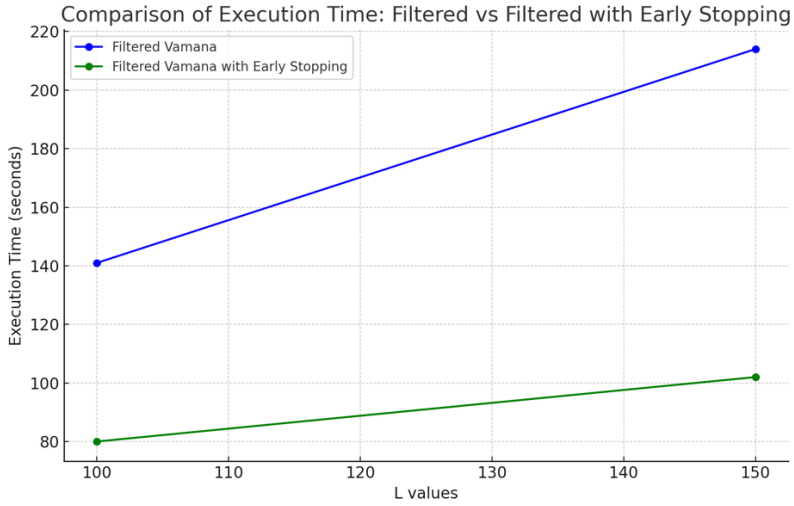
Για L = 100, a = 1, R = 13, k = 100: 141 seconds average recall (filtered) = 99.01% average recall (unfiltered queries) = 75.04%
Για L = 150, a = 1.1, R = 13, k = 100: 214 seconds average recall (filtered) = 99.1% average recall (unfiltered queries) = 75.1%

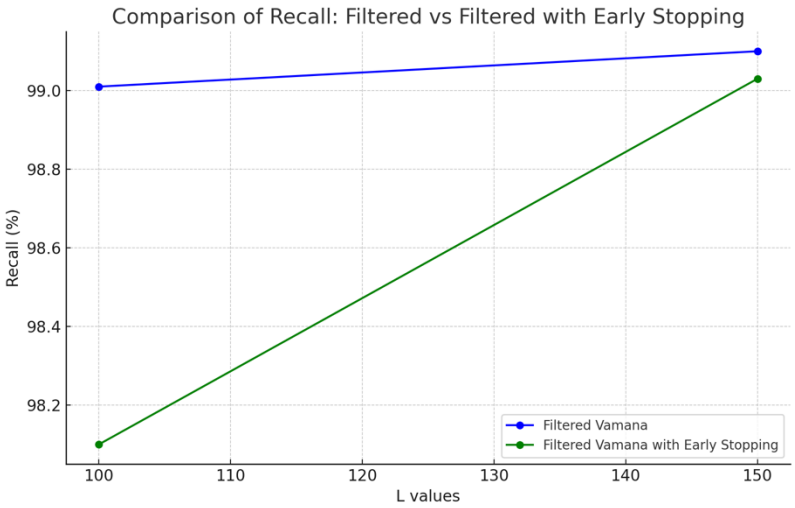
- Filtered Vamana με early stopping στην ευκλείδεια:

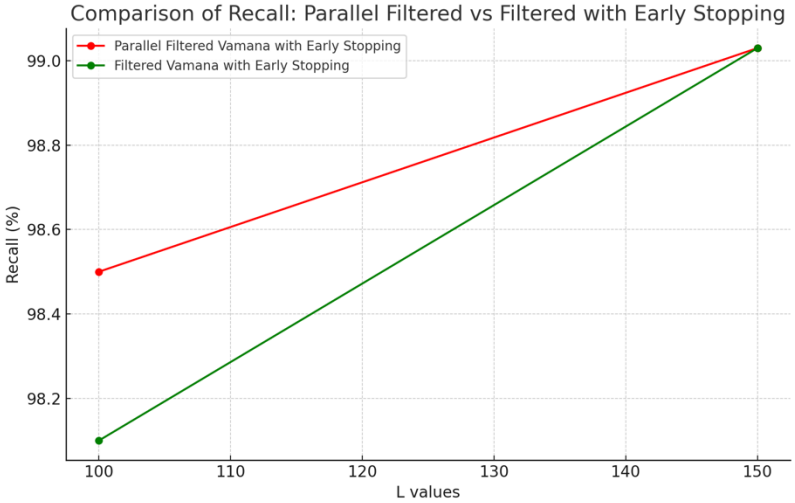
Για L = 100, a = 1, R = 13, k = 100: 80 seconds average recall (filtered) = 98.1 average recall (unfiltered queries) = 70.2%
Για L = 150, a = 1.1, R = 13, k = 100: 102 seconds average recall (filtered) = 99.03% average recall (unfiltered queries) = 71.5%

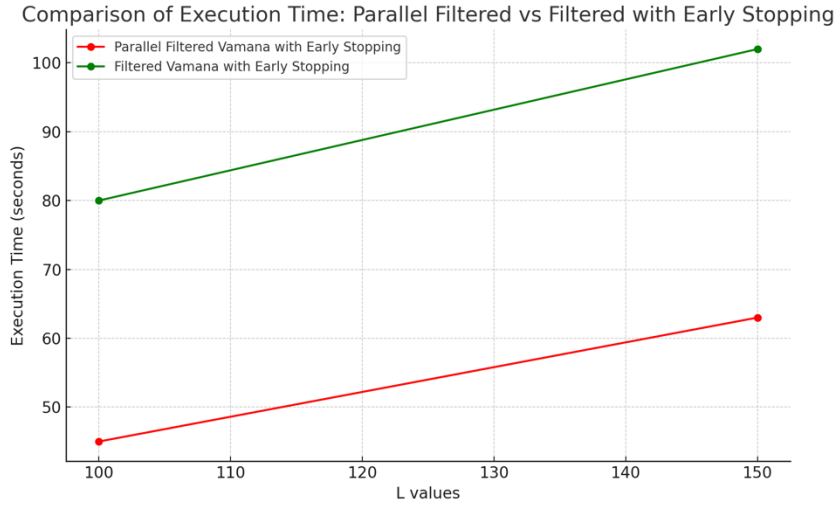
- Parallel Filtered Vamana with early stopping in Euclidean distances (in Greedy search):

Για L = 100, a = 1, R = 13, k = 100: 45 seconds average recall (filtered) = 98.5%. average recall (unfiltered queries) = 72.03%
Για L = 150, a = 1.1, R = 13, k = 100: 63 seconds average recall (filtered) = 99.03% average recall (unfiltered queries) = 72.4%




Comparison of Recall: Filtered vs Filtered with Early Stopping
Recall (%)
99.0
98.8
98.6
98.4
98.2
100
110
120
130
140
150
L values
Filtered Vamana
Filtered Vamana with Early Stopping


Comparison of Recall: Parallel Filtered vs Filtered with Early Stopping
Recall (%)
99.0
98.8
98.6
98.4
98.2
100
110
120
130
140
150
L values
Parallel Filtered Vamana with Early Stopping
Filtered Vamana with Early Stopping


Comparison of Execution Time: Parallel Filtered vs Filtered with Early Stopping
Execution Time (seconds)
100
90
80
70
60
50
100
110
120
130
140
150
L values
Parallel Filtered Vamana with Early Stopping
Filtered Vamana with Early Stopping

6. Βελτιστοποιήσεις που δεν δούλεψαν

Σε κάποια σημεία του κώδικα, δοκιμάζοντας κάποιες πιθανές βελτιστοποιήσεις και ιδέες, παρατηρήσαμε πως δεν υπήρχε κάποια σημαντική βελτίωση στην αποδοτικότητα του προγράμματος ή ακόμα η απόδοση μειώθηκε αρκετά.

Μία από αυτές τις ιδέες ήταν η αρχικοποίηση του γράφου με τυχαίες ακμές. Κατά την υλοποίηση αυτού όμως, παρατηρήσαμε σημαντική πτώση στην ταχύτητα εκτέλεσης του προγράμματος καθώς όλες οι συναρτήσεις βασίζονται στο γεγονός πως ο γράφος είναι R-regular και ότι υπάρχει συνοχή μεταξύ των γειτόνων. Κατ' επέκταση, χρειάζεται να γίνουν περισσότερα iterations ώστε ο γράφος να έρθει σε σωστή μορφή. Ειδικά στην περίπτωση που υπάρχουν φίλτρα, η τυχαία αρχικοποίηση καταρρίπτει τελείως την ιδέα διαχωρισμού των στοιχείων με βάση τα φίλτρα.

Μία ακόμα βελτιστοποίηση που προσπαθήσαμε να υλοποιήσουμε, ήταν η τυχαία αρχικοποίηση του medoid, η οποία επίσης έκανα το πρόγραμμα ακόμα λιγότερο αποδοτικό χρονικά. Το σημείο medoid έχει μεγάλη σημασία, καθώς η κατάλληλη επιλογή αυτού μειώνει σημαντικά το σύνολο αποστάσεων που ελέγχει ο αλγόριθμος. Έτσι, ένα τυχαίο σημείο εκκίνησης, δεν είναι αντιπροσωπευτικό για το σύνολο δεδομένων, οδηγώντας σε πολύ κακούς χρόνους εκτέλεσης, λόγω των εκτεταμένων υπολογισμών αποστάσεων.

7. Συμπεράσματα

Συνοψίζοντας, ο αλγόριθμος Vamana προσεγγιστικής αναζήτηση K-κοντινότερων γειτόνων αποτελεί ένα αποτελεσματικό εργαλείο για αρκετές εφαρμογές. Κατά την εκπόρευση της εργασίας αυτής, υλοποιήσαμε τον αλγόριθμο αυτό, τον βελτιστοποιήσαμε τόσο με απλές προγραμματιστικές μεθόδους όσο και με παράλληλο προγραμματισμό και διαπιστώσαμε πως οι αλλαγές αυτές επηρέασαν το συνολικό αποτέλεσμα σχετικά με την αποδοτικότητα. Τελικά, έγινε φανερό πόσο βελτιώθηκε σε θέμα χρόνου ο αλγόριθμος μας με τη χρήση παραλληλοποίησης και καταλήξαμε σε ένα αποτέλεσμα με αρκετά καλό recall και σε αρκετά μικρό χρόνο εκτέλεσης.

8. Αναφορές

1. http://dl.acm.org/citation.cfm?id=313559.313768
2. https://dl.acm.org/doi/abs/10.5555/3454287.3455520
3. https://transactional.blog/sigmod-contest/2024
4. https://doi.org/10.1145/3543507.3583552

Κώδικας python για τα plots:

```
import matplotlib.pyplot as plt

import numpy as np



# Data for plotting

L_values = [100, 150]
```

```
filtered_time = [141, 214]

filtered_recall = [99.01, 99.1]



filtered_early_time = [80, 102]

filtered_early_recall = [98.1, 99.03]



parallel_filtered_early_time = [45, 63]

parallel_filtered_early_recall = [98.5, 99.03]



# Plotting recall comparison between filtered vs filtered with early stopping

plt.figure(figsize=(10, 6))

plt.plot(L_values, filtered_recall, label='Filtered Vamana', marker='o', color='blue')

plt.plot(L_values, filtered_early_recall, label='Filtered Vamana with Early Stopping',
marker='o', color='green')

plt.xlabel('L values')

plt.ylabel('Recall (%)')

plt.title('Comparison of Recall: Filtered vs Filtered with Early Stopping')

plt.legend()

plt.grid(True)

plt.savefig("/mnt/data/recall_comparison_filtered_vs_early_stopping.png")



# Plotting time comparison between filtered vs filtered with early stopping

plt.figure(figsize=(10, 6))

plt.plot(L_values, filtered_time, label='Filtered Vamana', marker='o', color='blue')

plt.plot(L_values, filtered_early_time, label='Filtered Vamana with Early Stopping',
marker='o', color='green')

plt.xlabel('L values')

plt.ylabel('Execution Time (seconds)')

plt.title('Comparison of Execution Time: Filtered vs Filtered with Early Stopping')

plt.legend()

plt.grid(True)

plt.savefig("/mnt/data/time_comparison_filtered_vs_early_stopping.png")
```

```
# Plotting recall comparison between parallel filtered with early stopping vs filtered with early stopping

plt.figure(figsize=(10, 6))

plt.plot(L_values, parallel_filtered_early_recall, label='Parallel Filtered Vamana with Early Stopping', marker='o', color='red')

plt.plot(L_values, filtered_early_recall, label='Filtered Vamana with Early Stopping', marker='o', color='green')

plt.xlabel('L values')

plt.ylabel('Recall (%)')

plt.title('Comparison of Recall: Parallel Filtered vs Filtered with Early Stopping')

plt.legend()

plt.grid(True)

plt.savefig("/mnt/data/recall_comparison_parallel_vs_filtered_early_stopping.png")



# Plotting time comparison between parallel filtered with early stopping vs filtered with early stopping

plt.figure(figsize=(10, 6))

plt.plot(L_values, parallel_filtered_early_time, label='Parallel Filtered Vamana with Early Stopping', marker='o', color='red')

plt.plot(L_values, filtered_early_time, label='Filtered Vamana with Early Stopping', marker='o', color='green')

plt.xlabel('L values')

plt.ylabel('Execution Time (seconds)')

plt.title('Comparison of Execution Time: Parallel Filtered vs Filtered with Early Stopping')

plt.legend()

plt.grid(True)

plt.savefig("/mnt/data/time_comparison_parallel_vs_filtered_early_stopping.png")



plt.show()
```