Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα

Διαμαντή Χριστίνα 1115201800046
Γιέβτοβιτς Άντρια 1115202000033
Σταμάτη Μαριάννα 1115202000183

Τμήμα Πληροφορικής και Τηλεπικοινωνιών
Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Ιανουάριος 2025

Filtered Approximate Nearest Neighbor Search (Filtered ANNS)

Βελτιστοποιήσεις και συμπεράσματα

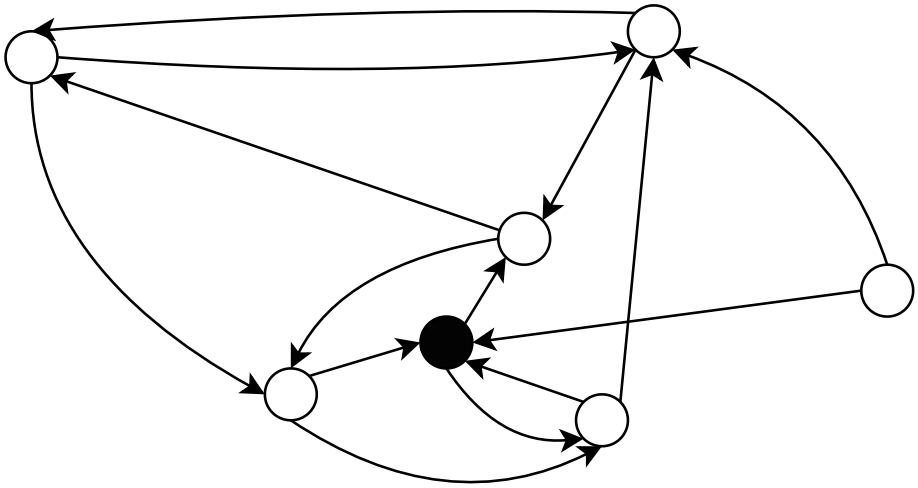
Περιεχόμενα

1 Εισαγωγή

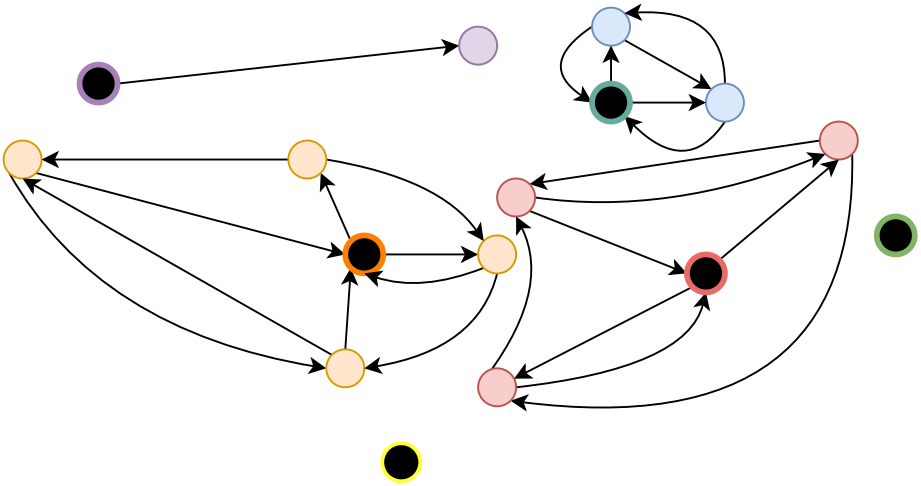
Οι αλγόριθμοι filtered vamana και stitched vamana, έχουν ως στόχο την κατασκευή γράφων (ευρετηρίων) τα οποία θα χρησιμοποιηθούν για την απάντηση ερωτημάτων (queries) για την εύρεση των k πιθανών κοντινότερων γειτόνων ενός ερωτήματος. Η κατασκευή αυτών των γράφων, στην προκειμένη περίπτωση, βασίζεται σε datasets με σημεία που περιέχουν φίλτρα (συγκεκριμένα 1 μόνο φίλτρο ανά σημείο). Η παραπάνω υλοποίηση, που χρησιμοποιείται στο δεύτερο παραδοτέο, καταφέρνει να δημιουργεί γράφους-ευρετήρια τα οποία χρησιμοποιούνται και απαντούν με μεγάλη επιτυχία σε ερωτήματα που αφορούν φίλτρα (filtered queries). Δίνουν δηλαδή συνολικό recall πάνω από 95% για filtered queries. Αντίστοιχα δοκιμάστηκαν οι ίδιοι γράφοι ως ευρετήρια για την απάντηση ερωτημάτων που δεν φέρουν φίλτρα (unfiltered queries). Το συνολικό recall για την συγκεκριμένη κατηγορία ερωτημάτων ήταν υπερβολικά χαμηλό (κάτω από 50%).

Συμπεράσματα

Με βάση τα παραπάνω, συμπεραίνουμε πως οι αλγόριθμοι filtered και stitched vamana δημιουργούν γράφους οι οποίοι είναι κατάλληλα ευρετήρια για την απάντηση μόνο σε ερωτήματα με φίλτρα.
Αντίθετα, ο αλγόριθμος του απλού vamana (ο αλγόριθμος που χρησιμοποιήθηκε στο πρώτο παραδοτέο) κατασκευάζει γράφο ο οποίος απαντάει με πολύ μεγάλη επιτυχία σε ερωτήματα χωρίς φίλτρο (ολικό recall πάνω από 95%). Συγκρίνοντας τις δύο μεθόδους παρατηρείται πως η κατασκευή των ευρετηρίων παίζει πολύ βασικό ρόλο στην εύρεση των σωστών κοντινότερων γειτόνων. Παρακάτω απεικονίζονται σχηματικά οι γράφοι που παράγονται με την μέθοδο του απλού vamana και την μέθοδο filtered ή stitched vamana (οι δύο τελευταίοι παράγουν γράφο ίδιας μορφής).



Εικόνα 1: Γράφος που κατασκευάζεται με τη μέθοδο του απλού vamana.



Εικόνα 2: Γράφος που κατασκευάζεται με τη μέθοδο filtered ή stitched vamana.

Όπως φαίνεται και στα παραπάνω σχήματα ο απλός vamana παράγει έναν κατευθυνόμενο γράφο με σταθερό βαθμό εξόδου (max out-degree) σε κάθε κόμβο ίσο με R και ένα κέντρο, medoid, από το οποίο ξεκινάει πάντα την αναζήτηση των κοντινότερων γειτόνων. Αντίθετα, οι αλγόριθμοι filtered και stitched παράγουν έναν μη συνεκτικό κατευθυνόμενο γράφο, όπου κάθε συνεκτική συνιστώσα αντιστοιχεί σε ένα φίλτρο (Εικόνα 2 - διαφορετικό χρώμα κόμβων) και έχει ένα τοπικό κέντρο (medoid) όπως φαίνεται στο σχήμα.

Μπορούμε εύκολα να καταλάβουμε το λόγο που ο δεύτερος γράφος δεν είναι αποδοτικός στην εύρεση ερωτημάτων χωρίς φίλτρα, καθώς οι συνδέσεις μεταξύ των κόμβων είναι ομαδοποιημένες και οι συνδέσεις μεταξύ των φίλτρων ανύπαρκτες. Επίσης η αναζήτηση για κοντινότερους γείτονες σε unfiltered queries πρέπει να γίνει σε ολόκληρο τον γράφο και όχι μόνο σε ένα υποσύνολο αυτού, όπως συμβαίνει στα filtered queries. Εδώ η ύπαρξη πολλών medoids δυσκολεύει αυτήν την διαδικασία και εννοείται και οι ασθενείς συνδέσεις μεταξύ των κόμβων.

Παρακάτω παρουσιάζονται ορισμένες βελτιώσεις που θα συμβάλουν στην κατασκευή γράφου, με την χρήση των αλγόριθμων filtered και stitched vamana, ο οποίος θα μπορεί να βελτιώσει την εύρεση κοντινότερων γειτόνων για unfiltered queries αλλά ταυτόχρονα δεν θα επηρεάζει αρνητικά την αναζήτηση κοντινότερων γειτόνων για filtered queries.

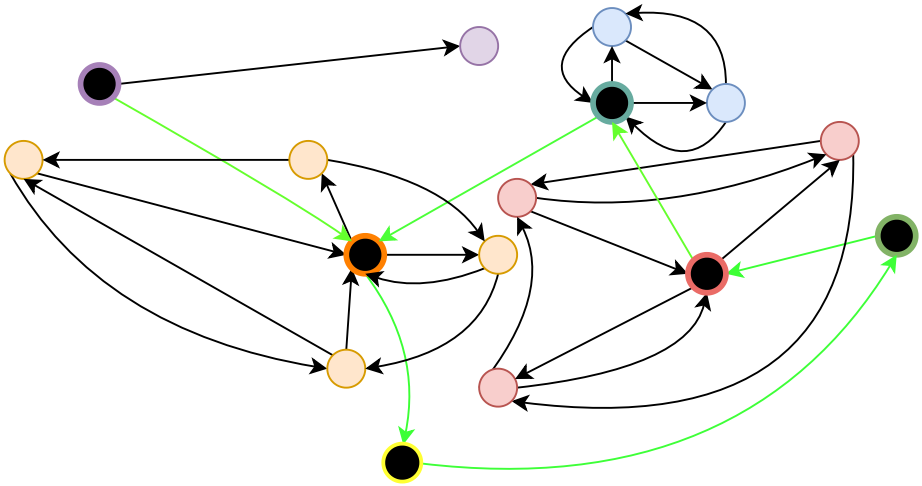
2 Βελτιώσεις στα αποτελέσματα (Recall)

2.1 Ακμές και συνδέσεις

Σκοπός της συγκεκριμένης βελτίωσης είναι να φέρουμε την μορφή του γράφου, που παράγεται από τους αλγόριθμους filtered και stitched vamana, σε όσο πιο κοντινή μορφή γίνεται με τον γράφο που παράγει ο αλγόριθμος της απλής vamana. Αυτό είναι σημαντικό διότι παρατηρούμε πως οι συνδέσεις του απλού vamana εξυπηρετούν στην απάντηση unfiltered ερωτημάτων. Για να επιτύχει το παραπάνω προσθέτουμε μερικές έξτρα ακμές στον γραφο για να κάνουμε τις συνδέσεις μεταξύ των κόμβων πιο πυκνές. Οι έξτρα ακμές ωστόσο προστίθονται μονο μεταξύ των κόμβων που είναι medoids ώστε να μην επιβαρύνονται όλοι οι κόμβοι με περισσότερους γείτονες.

Με αυτόν τον τρόπο δεν αλλοιώνουμε πολύ την αρχική μορφή του γράφου που παράγεται από τους αλγορίθμους αλλά ταυτόχρονα διευκολύνουμε την εύρεση κοντινότερων γειτόνων για unfiltered queries , καθώς οι έξτρα ακμές μεταξύ των medoids αυξάνουν την πιθανότητα να επισκεφτούμε περισσότερους κόμβους διαφορετικών φίλτρων, πιο γρήγορα, μεταπηδώντας από ένα φίλτρο σε ένα άλλο μέσω των medoids.

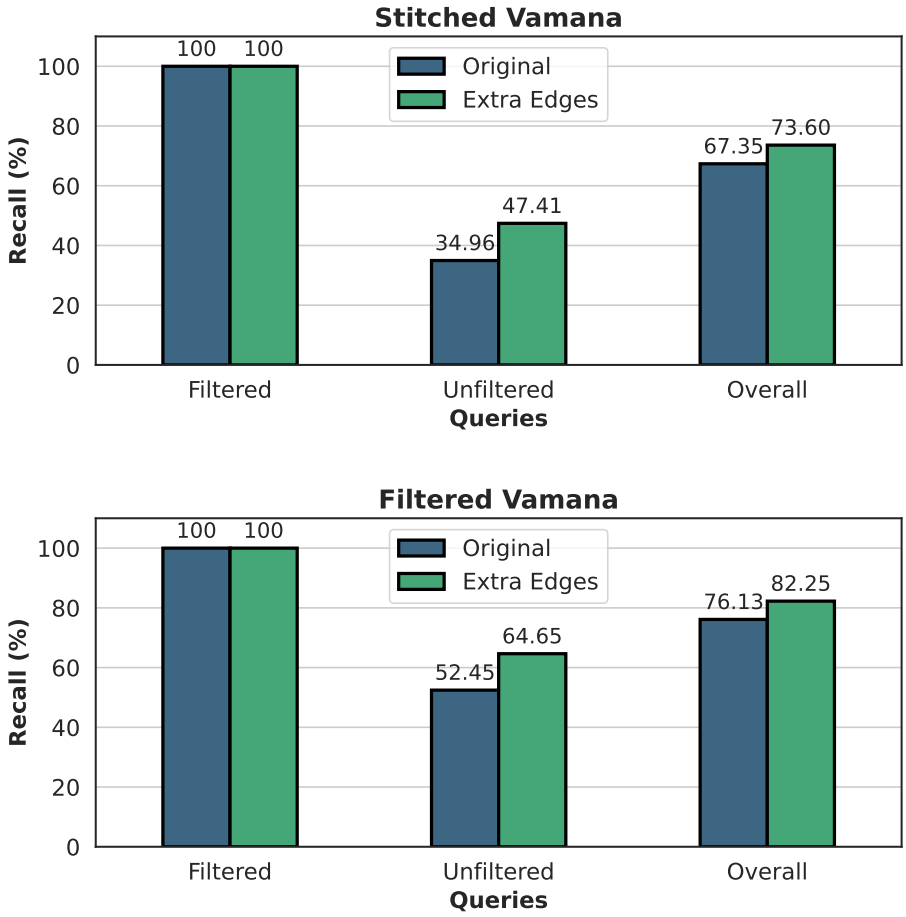
Παρακάτω παρουσιάζεται η μορφή του γράφου με τις έξτρα ακμές που προστίθενται (πράσινες ακμές). Πρώτα κατασκευάζεται ο γράφος και ύστερα ενώνουμε τα medoids με έξτρα ακμές. Ως πλήθος έξτρα ακμών επιλέγουμε ένα μικρό νούμερο (στο μικρό dataset διαλέγουμε το 5% του πλήθους των φίλτρων).



Εικόνα 3: Γράφος με την προσθήκη έξτρα ακμών (γέφυρες μεταξύ medoids).

Αυτή η μέθοδος αυξάνει το recall πάνω από 10% περισσότερο για τα unfiltered queries ενώ ταυτόχρονα δεν μειώνει την απόδοση για τα filtered queries. Το διάγραμμα στη συνέχεια μας δείχνει αυτή την διαφορά στο απο-

τέλεσμα (recall), πριν και μετά την προσθήκη των ακμών.



Εικόνα 4: Αποτελέσματα recall για stitched και filtered vamana γράφους

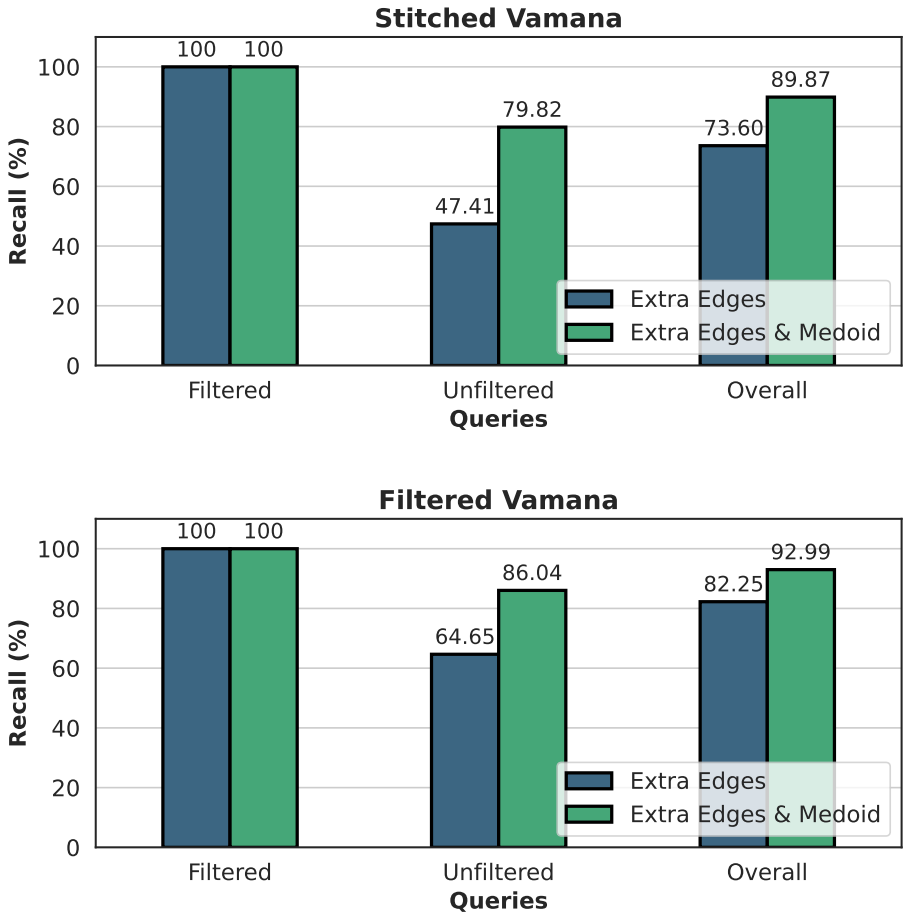
2.2 Find medoid

Η συνάρτηση Find Medoid που χρησιμοποιείται για την εύρεση των medoids (στο δεύτερο παραδοτέο) χρησιμοποιεί μια μέθοδο τυχαίας επιλογής κόμβων ως medoids για κάθε φίλτρο. Αυτή η μέθοδος σαφώς επηρεάζει σημαντικά τόσο την κατασκευή του αρχικού γράφου, όσο και την αναζήτηση γειτόνων για την απάντηση των queries στη συνέχεια. Τα medoids που επιλέγονται αποτελούν τα σημεία από τα οποία ο γράφος θα ξεκινήσει να χτίζεται (αλγόριθμος filtered vamana) και τα σημεία από τα οποία η greedy search θα ξεκινάει να ψάχνει για κοντινότερους γείτονες (αλγόριθμος stitched vamana

και greedy search για queries).

Αφού η επιλογή των medoids γίνεται τυχαία, τότε υπάρχει περίπτωση να επιλεχθούν ως κέντρα κόμβοι πολύ απομακρυσμένοι από την κάθε συστάδα (κάθε συστάδα αντιπροσωπεύει ένα φίλτρο). Αυτό έχει ως αποτέλεσμα να αυξήσει τον χρόνο αναζήτησης κοντινότερων γειτόνων, μιας και η έρευνα ξεκινά από πολύ μακρινό σημείο, αλλά και να χτιστούν γράφοι-ευρετήρια με λάθος προδιαγραφές.

Για την αντιμετώπιση των παραπάνω χρησιμοποιείται η κλασική μέθοδος εύρεσης medoid με την προσθήκη δειγματοληψίας. Αναλυτικότερα, επειδή τα κέντρα παίζουν καθοριστικό ρόλο στην αναζήτηση, χρησιμοποιείται η μέθοδος brute force σε μικρές συστάδες, ενώ σε μεγαλύτερες συστάδες (άνω των 30.000 κόμβων) επιλέγεται ένα ικανοποιητικό μέρος των κόμβων τυχαία ως δείγμα και ύστερα υπολογίζεται το medoid με τον κλασικό τρόπο, ανάμεσα από τους επιλεγμένους κόμβους. Με αυτόν τον τρόπο βρίσκουμε σταθερά κέντρα (χωρίς να έχουμε σοβαρή επιβάρυνση στον χρόνο εκτέλεσης) και οι γράφοι που παράγονται και χρησιμοποιούνται ως ευρετήρια δεν αλλάζουν μορφή με κάθε εκτέλεση του προγράμματος. Επίσης η σημασία των σωστών medoids φαίνεται στα παρακάτω διαγράμματα, καθώς το recall για τα unfiltered queries αυξήθηκε αρκετά (συγκρίνουμε τα αποτελέσματα της προηγούμενης βελτίωσης, με τα αποτελέσματα του συνδυασμού των έξτρα ακμών και των σωστών κέντρων).



Εικόνα 5: Αποτελέσματα recall για stitched και filtered vamana πριν και μετά την προσθήκη πραγματικών medoids

2.3 Πειράματα και αποτυχημένες δοκιμές

Δοκιμή 1 - Αρχικοποίηση γράφου με τυχαίους κόμβους

Η αρχικοποίηση του γράφου με τυχαία σημεία πριν την εκτέλεση του filtered vamana (όπως γίνεται στην κατασκευή γράφου με τον απλό vamana) δεν είναι ικανοποιητική, διότι οι συναρτήσεις που κατασκευάζουν τον γράφο (filtered greedy search και filtered robust prune) ξεχωρίζουν και κόβουν τους γείτονες με "άσχετα" φίλτρα και καταλήγουμε να έχουμε ως αποτέλεσμα έναν γράφο παρόμοιας μορφής με την Εικόνα 2 και χωρίς σοβαρή αλλαγή στο recall.

Δοκιμή 2 - Επιλογή πραγματικού medoid για unfiltered queries

Η επιλογή, ως medoid, του πραγματικού κεντρικού κόμβου για την αναζήτηση (filtered greedy search) σε unfiltered queries δεν μπορεί να χρησιμοποιηθεί, διότι το αληθινό medoid του dataset μπορεί να είναι κάποιος κόμβος αδιάφορος στον γράφο που κατασκευάζουν οι filtered μέθοδοι και να μην έχει καλές συνδέσεις ώστε να επεκταθεί η έρευνα στους υπόλοιπους κόμβους. (Στο μικρό dataset το πραγματικό medoid είναι ο κόμβος 5234). Αυτή η δοκιμή έριξε το recall πολύ πιο χαμηλά ακόμα και από την κλασική υλοποίηση του δεύτερου παραδοτέου.

Δοκιμή 3 - Επιλογή medoid από τα τυχαία medoids

Η αρχική ιδέα έχει ως εξής: Κατασκευάζουμε τον χάρτη με τις αντιστοιχίες filter - start node με την χρήση της παλιάς συνάρτησης τυχαίας επιλογής ''Find Medoid''. Ύστερα, από αυτά τα start nodes βρίσκουμε με brute force το πιο κεντρικό σημείο και χρησιμοποιούμε αυτό ως medoid στην filtered greedy search για την εύρεση γειτόνων για unfiltered queries. Αυτή η διαδικασία έριξε το recall πολύ πιο χαμηλά ακόμα και από την κλασική υλοποίηση του δεύτερου παραδοτέου, διότι η φύση του γράφου με τις αραιές συνδέσεις δεν επιτρέπει την εύκολη αναζήτηση γειτόνων σε όλο τον γράφο ξεκινώντας από ένα μόνο σημείο.

Δοκιμή 4 - Έξτρα N ακμές, N=10% του πλήθους των φίλτρων

Η επιλογή έξτρα ακμών για καλύτερη σύνδεση μεταξύ των medoids δεν δίνει πάντα καλύτερα αποτελέσματα. Η προσθήκη πολλών επιπλέον ακμών σε έναν κόμβο έριχνε το recall για unfiltered queries. Δοκιμάστηκε η προσθήκη 12 επιπλέον ακμών για κάθε κόμβο medoid (το 12 προέρχεται από το 10% του πλήθους των φίλτρων του μικρού dataset). Η παραπάνω δοκιμή επιβάρυνε χρονικά την αναζήτηση, παρόλο που έκανε πιο πυκνές τις συνδέσεις, και επηρέασε το recall αρνητικά, σε αντίθεση με την επιλογή του 5% του πλήθους των φίλτρων το οποίο δεν επηρέασε τον χρόνο αλλά αύξησε και το recall.

3 Χρόνος εκτέλεσης

3.1 Εισαγωγή

Παρατηρώντας τον χρόνο εκτέλεσης τόσο της κλασικής υλοποίησης όσο και αυτής με το βελτιωμένο recall, πρέπει να επισημάνουμε πόσο σημαντικό είναι να γίνει μείωση στον χρόνο εκτέλεσης. Υπάρχουν αρκετά σημεία στον κώδικα τα οποία επαναλαμβάνονται ή συναρτήσεις που καλούνται συχνά με αποτέλεσμα την αύξηση του χρόνου. Παρακάτω χρησιμοποιούμε δύο μεθόδους, οι οποίες σε συνδυασμό, καταφέρνουν να μειώσουν τον χρόνο εκτέλεσης σημαντικά.

3.2 Χρονικές βελτιώσεις

3.2.1 SIMD

Ένα πολύ κρίσιμο σημείο που αυξάνει τον χρόνο εκτέλεσης του προγράμματος αποτελούν οι πράξεις για τον υπολογισμό της ευκλείδειας απόστασης μεταξύ των κόμβων (μήκος ακμών). Η ευκλείδεια απόσταση χρησιμοποιείται σχεδόν παντού μέσα στο πρόγραμμα και σχεδόν όλα καθορίζονται από αυτή (υπολογισμός medoid, groundtruth αρχείο, ταξινόμηση λιστών στις greedy search και filtered greedy).

Ο υπολογισμός και η αποθήκευση όλων των αποστάσεων εξ' αρχής σε έναν πίνακα είναι μια λύση αδύνατη για το συγκεκριμένο πρόγραμμα, καθώς αυτή η λύση δεν μπορεί να υποστηρίξει μεγάλα δεδομένα λόγω περιορισμένης μνήμης. Άρα η μοναδική λύση είναι η βελτιστοποίηση των πράξεων της ευκλείδειας απόστασης με την χρήση SIMD (Single Instruction Multiple Data) μέσω AVX/AVX2 στη συνάρτηση υπολογισμού της.

Με αυτή την τεχνική καταφέρνουμε να κάνουμε πράξεις με πολλούς αριθμούς ταυτόχρονα, στην περίπτωσή μας, για το datasize που είχαμε (100 διαστάσεις), επιλέγουμε παραλληλία 8 αριθμών μαζί, καθώς τόσο μας επιτρέπουν στο μέγιστο οι registers του AVX2 για floating point αριθμούς των 32-bit. Ύστερα με τη χρήση του Fused Multiply Accumulate (FMA) συνδυάζουμε την πράξη του πολλαπλασιασμού και της πρόσθεσης σε ένα μόνο instruction, με αποτέλεσμα να υπάρχει περαιτέρω μείωση στον χρόνο εκτέλεσης ακόμα και στα datasets μικρών δεδομένων (τα οποία ήδη είχαν ικανοποιητικό χρόνο εκτέλεσης).

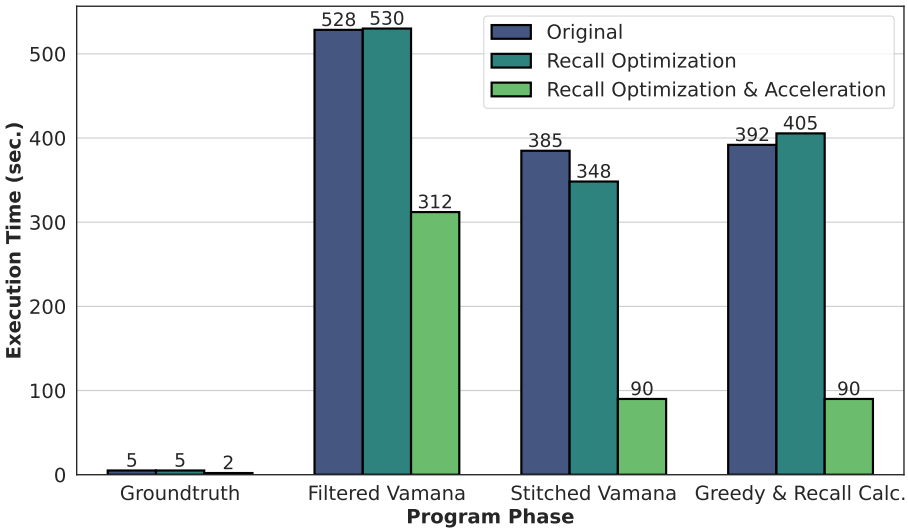
3.2.2 Multithreading

Άλλη μια βελτιστοποίηση που χρησιμοποιείται για την μείωση του χρόνου εκτέλεσης είναι το multithreading. Χρησιμοποιήθηκε OpenMP για μεγαλύτερη ευκολία μέσω των ντιρεκτίβων (pragmas) που προσφέρει. Η τεχνική αυτή τοποθετήθηκε κυρίως σε loops με μεγάλο αριθμό επαναλήψεων και σε σημεία όπου επαναλαμβάνονται συχνά με αποτέλεσμα την αύξηση του χρόνου. Η προσθήκη της δεν βοήθησε σε όλα τα σημεία διότι πολλές φορές η

συνεχής δημιουργία των threads ήταν πιο χρονοβόρα από την απλή εκτέλεση των επαναλήψεων, οπότε περιορίστηκε σε όσα θεωρούνται πιο σημαντικά.

Πιο συγκεκριμένα χρησιμοποιήθηκε multithreading στην συνάρτηση create graph, η οποία αρχικοποιεί κάθε κόμβο από το dataset με R τυχαίους γείτονες. Η παραλληλοποίηση στο συγκεκριμένο σημείο ενεργοποιείται μόνο όταν έχουμε πολύ μεγάλο datastet (πάνω από 30.000 κόμβους) ώστε να μειώσει τον χρόνο δημιουργίας του τυχαίου γράφου. Δοκιμάστηκε σε πλήθος κόμβων ίσο με 58.000 και η μείωση του χρόνου ήταν σημαντική, από 40 δευτερόλεπτα σε 10 δευτερόλεπτα για την δημιουργία τυχαίου γράφου. Η create graph χρησιμοποιείται από τον αλγόριθμο του απλού vamana, ο οποίος χρησιμοποιείται από στον stitched vamana για την κατασκευή υπογράφου για κάθε φίλτρο. Επίσης στη συνάρτηση medoid, που καλείται από την Find Medoid για την εύρεση του κεντρικού σημείου κάθε φίλτρου, χρησιμοποιήθηκε multithreading καθώς τα loops της έχουν πολλές επαναλήψεις.

Στη συνέχεια φαίνονται οι χρόνοι εκτέλεσης του προγράμματος σε τρείς φάσεις. Αρχικά το πρόγραμμα χωρίς καμία βελτίωση (ούτε χρονική, ούτε recall), ύστερα ο χρόνος εκτέλεσης με τις βελτιώσεις recall και τέλος ο χρόνος εκτέλεσης του ολοκληρωμένου προγράμματος με τις βελτιώσεις recall και χρόνου.



Εικόνα 6: Χρόνοι εκτέλεσης για διαφορετικές υλοποιήσεις.

Παρατηρούμε πως ο χρόνος εκτέλεσης μειώθηκε σημαντικά για όλες τις συναρτήσεις, ακόμα και για την παραγωγή του groundtruth αρχείου το οποίο επηρεάζεται μόνο από το SIMD.

4 Βασικές πληροφορίες

Χαρακτηριστικά υπολογιστή που έγιναν οι δοκιμές για τα δείγματα:

Χαρακτηριστικό	**Περιγραφή**
Επεξεργαστής	Intel Core i5-9400F @ 4.1 GHz (6C/6T)
Μνήμη RAM	32GB DDR4
Λειτουργικό Σύστημα	Ubuntu 22.04
Έκδοση GCC	gcc 11.4.0
Έκδοση Πυρήνα	Linux 6.8.0-49-generic

Αναφορές

[1] S. Arya and D. M. Mount, "Approximate nearest neighbor queries in fixed dimensions.," in *SODA*, vol. 93, pp. 271–280, Citeseer, 1993.

[2] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[3] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, S. Raz, Y. Lin, Y. Zhang, N. Mahapatro, P. Srinivasan, *et al.*, "Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters," in *Proceedings of the ACM Web Conference 2023*, pp. 3406–3416, 2023.

[4] "SIGMOD Programming Contest Archive: Hybrid Vector Search (2024) — transactional.blog." https://transactional.blog/sigmod-contest/2024.