HELLENIC REPUBLIC
National and Kapodistrian University of Athens
EST. 1837

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Κ23α

Προσεγγιστική Επίλυση του προβλήματος Κ-Εγγύτερων Γειτόνων (K-Nearest Neighbors) μέσω του Αλγορίθμου Vamana Indexing Algorithm (VIA)

Τελική Αναφορά

Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Αντώνης Ζήκας 1115202100038
Σταύρος Κώτσιλας 1115201700292
Ευανθία Χασιώτη 1115202100289

Αθήνα, Ιανουάριος 2025

Περιεχόμενα

1 Εισαγωγή

1.1 Γενική Επισκόπηση

Το παρόν έργο επικεντρώνεται στην ανάπτυξη λογισμικού για πληροφοριακά συστήματα, με ειδική έμφαση στην προσέγγιση του προβλήματος των Προσεγγιστικών Εγγύτερων Γειτόνων (**Approximate Nearest Neighbors - ANN**). Συγκεκριμένα, υλοποιείται ο Αλγόριθμος Δεικτοδότησης **Vamana** (**Vamana Indexing Algorithm - VIA**), ο οποίος αποτελεί σύγχρονη μέθοδο για την αποδοτική επίλυση του προβλήματος **ANN** σε μεγάλες διαστάσεις δεδομένων.

Ο στόχος του έργου είναι η δημιουργία ενός αποδοτικού και ευέλικτου συστήματος που θα επιτρέπει την ταχεία αναζήτηση και ανάκτηση πληροφορίας σε μεγάλες βάσεις δεδομένων, αξιοποιώντας τις δυνατότητες του **VIA**. Η υλοποίηση αυτή στοχεύει στη βελτίωση της απόδοσης και της ακρίβειας σε εφαρμογές που απαιτούν γρήγορη και αξιόπιστη εύρεση εγγύτερων γειτόνων.

1.2 Στόχοι

Οι κύριοι στόχοι του έργου περιλαμβάνουν:

- Υλοποίηση του Αλγορίθμου **Vamana**: Ανάπτυξη μιας πλήρους και λειτουργικής έκδοσης του **VIA** , προσαρμοσμένης στις ανάγκες του προβλήματος **ANN**.
- Βελτιστοποίηση Απόδοσης: Εφαρμογή τεχνικών που μειώνουν τον χρόνο εκτέλεσης και τις απαιτήσεις σε πόρους, διασφαλίζοντας την αποδοτικότητα του συστήματος.
- Αξιολόγηση Ακρίβειας: Διεξαγωγή πειραμάτων για την αξιολόγηση της ακρίβειας και της αποτελεσματικότητας του υλοποιημένου αλγορίθμου σε σύγκριση με υπάρχουσες μεθόδους.

2 Υλοποίηση εφαρμογής

Η διαδικασία ξεκινά από τη διαχείριση των δεδομένων, τα οποία αποτελούν τη βάση της λειτουργικότητας του συστήματος. Τα δεδομένα αποθηκεύονται σε αρχεία που περιέχουν διανύσματα υψηλών διαστάσεων, κάθε ένα εκ των οποίων αναπαριστά ένα σημείο δεδομένων (data point). Η φόρτωση των δεδομένων πραγματοποιείται μέσω εξειδικευμένων συναρτήσεων, όπως η loaddata, η οποία ενσωματώνεται στις υποστηρικτικές κλάσεις του έργου. Τα δεδομένα μεταφέρονται από τα αρχεία σε δομές δεδομένων όπως πίνακες (arrays) ή διανύσματα (vectors), τα οποία οργανώνονται ώστε να επιτρέπουν την αποδοτική πρόσβαση και επεξεργασία.

Μετά τη φόρτωση των δεδομένων, το επόμενο βήμα είναι η κατασκευή του γράφου. Ο γράφος υλοποιείται με βάση μια λίστα γειτνίασης adjacency list, η οποία αποθηκεύει τις σχέσεις εγγύτητας μεταξύ των σημείων. Οι κόμβοι (nodes) του γράφου αντιστοιχούν στα δεδομένα, ενώ οι ακμές (edges) εκφράζουν τη συνάφεια μεταξύ αυτών, όπως υπολογίζεται από συγκεκριμένα μέτρα απόστασης, συνήθως την ευκλείδεια απόσταση (Euclidean distance). Η κατασκευή του γράφου γίνεται μέσω της μεθόδου buildgraph.

Εντούτοις, για την αποδοτική αναζήτηση στον γράφο, η διαδικασία αναζήτησης βασίζεται σε αλγόριθμους όπως ο **Greedy Search**και η στρατηγική **Robust Prune**.Ο αλγόριθμος **Greedy Search**,αναλαμβάνει την αποδοτική αναζήτηση κοντινών γειτόνων. Στη διαδικασία αυτή, επιλέγεται ο πλησιέστερος γείτονας ανά πάσα στιγμή και η αναζήτηση συνεχίζεται εστιάζοντας στους πιο υποσχόμενους κόμβους, αποφεύγοντας μη χρήσιμους. Αυτή η στρατηγική μειώνει την υπολογιστική πολυπλοκότητα, επιτυγχάνοντας ταχύτερη εκτέλεση ακόμα και σε μεγάλα σύνολα δεδομένων.

Επιπλέον, η στρατηγική **Robust Prune** εφαρμόζεται για τη βελτιστοποίηση της αναζήτησης, μειώνοντας τον αριθμό των μη χρήσιμων ακμών στον γράφο και περιορίζοντας τις απαιτήσεις μνήμης και υπολογιστικών πόρων. Η στρατηγική αυτή έχει σημαντικό ρόλο στη μείωση της χρονικής επιβάρυνσης, επιτρέποντας στο σύστημα να διαχειρίζεται μεγάλους και πυκνούς γράφους αποτελεσματικά.

Αφού οι βασικές αρχές αναζήτησης έχουν εφαρμοστεί, προχωρούμε στη κατασκευή του γράφου μέσω της μεθόδου buildgraph,η οποία περιλαμβάνεται στην κλάση **Vamana**.Ο αλγόριθμος κατασκευής του γράφου βασίζεται σε μια greedy στρατηγική, κατά την οποία κάθε κόμβος συνδέεται με έναν πεπερασμένο αριθμό M πλησιέστερων γειτόνων, οι οποίοι επιλέγονται δυναμικά βάσει αποστάσεων.

Η κλάση **Vamana** υλοποιεί τον ομώνυμο αλγόριθμο, ο οποίος είναι σχεδιασμένος για αποδοτική πλοήγηση στον γράφο και αναζήτηση K-Εγγύτερων Γειτόνων. Η μέθοδος **Greedy search**,που αποτελεί τον πυρήνα της λειτουργικότητας της κλάσης, λαμβάνει ως είσοδο ένα ερώτημα (query vector) και επιστρέφει τους δείκτες των K πλησιέστερων γειτόνων. Η διαδικασία αναζήτησης ξεκινά από έναν αρχικό κόμβο, ο οποίος επιλέγεται τυχαία ή βάσει συγκεκριμένων κριτηρίων, και συνεχίζεται με τη χρήση μιας τοπικής greedy στρατηγικής, η οποία εξασφαλίζει ότι το ερώτημα προσεγγίζει σταδιακά την περιοχή του χώρου όπου βρίσκονται οι πραγματικοί πλησιέστεροι γείτονες. Αυτή η στρατηγική μειώνει σημαντικά τον χρόνο αναζήτησης, καθιστώντας τον αλγόριθμο ιδιαίτερα αποδοτικό για μεγάλα σύνολα δεδομένων.

Η επεκτασιμότητα του συστήματος επιτυγχάνεται μέσω υποκλάσεων της κλάσης **Vamana**,όπως οι **FilteredVamana** και **StitchedVamana**. Η **FilteredVamana** προσφέρει τη δυνατότητα προσθήκης φίλτρων στις συνδέσεις του γράφου, τα οποία καθορίζουν ποιοι κόμβοι μπορούν να συνδεθούν μεταξύ τους, με βάση κριτήρια όπως η απόσταση ή τα χαρακτηριστικά των κόμβων. Αυτή η λειτουργικότητα επιτρέπει την ευέλικτη διαχείριση της σχέσης των γειτονικών σημείων, διασφαλίζοντας ότι μόνο οι πιο σχετικοί κόμβοι συνδέονται μεταξύ τους. Αντίστοιχα, η **StitchedVamana** επιτρέπει τη συγχώνευση (stitching) πολλαπλών γράφων σε έναν ενιαίο γράφο, διατηρώντας τη συνοχή των συνδέσεων και τη δομή των αρχικών γράφων. Αυτή η δυνατότητα είναι εξαιρετικά χρήσιμη όταν τα δεδομένα προέρχονται από διαφορετικές πηγές και απαιτείται η ενοποίησή τους σε μια κοινή βάση.

Η διαδικασία εκτέλεσης του κώδικα περιλαμβάνει την ενσωμάτωση όλων των παραπάνω στοιχείων σε μια ενιαία ροή εργασίας. Αρχικά, τα δεδομένα φορτώνονται και οργανώνονται στις απαιτούμενες δομές. Στη συνέχεια, κατασκευάζεται ο γράφος, ενώ η λειτουργία αναζήτησης ενεργοποιείται η οποία εκτελείται για κάθε ερώτημα. Τέλος, τα αποτελέσματα επιστρέφουν ως λίστες δεικτών, αντιπροσωπεύοντας τους πλησιέστερους γείτονες κάθε ερωτήματος.

3 Ανάλυση Αλγορίθμων

Εξετάσαμε τους συγκεκριμένους αλγορίθμους με διάφορες τιμές του L και του R και λάβαμε διάφορα αποτελέσματα τόσο στο **χρόνο κατασκευής** του ευρετηρίου, όσο και στην **ακρίβεια** (recall) των αποτελεσμάτων.

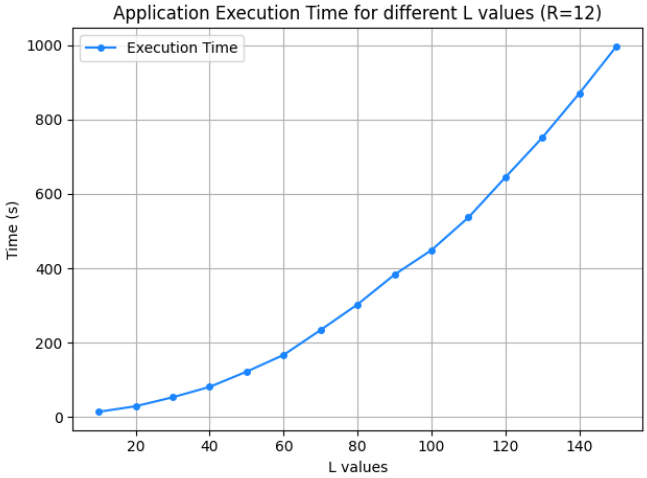
3.1 Χρόνος Κατασκευής Ευρετηρίου

3.1.1 Simple Vamana

Αρχικά θα αναλύσουμε τη συμπεριφορά που έχει ο χρόνος κατασκευής του ευρετηρίου για διάφορες τιμές τόσο του **L**, όσο και του **R**. Θα ξεκινήσουμε αναλύοντας τον **Απλό Vamana** (Simple Vamana) , ο οποίος χρησιμοποιεί δεδομένα από το σετ **Siftsmall** . Εκτελέσαμε τον **απλό αλγόριθμο Vamana** για τις παρακάτω τιμές του L:

$$L \in \{10, 20, 30, \ldots, 150\}$$

με το R να είναι σταθερό και ίσο με $R = 12$. Τα αποτελέσματα που λάβαμε φαίνονται στο παρακάτω διάγραμμα.

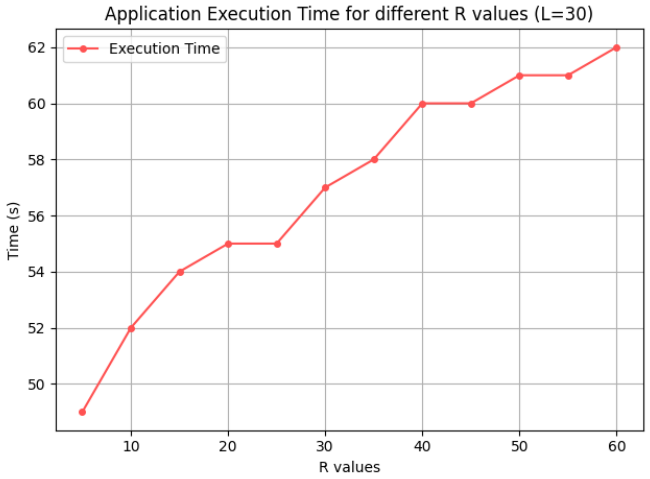


Όπως παρατηρούμε από το παραπάνω διάγραμμα, όσο η **τιμή του L** αυξάνεται, αυξάνεται επίσης και ο **χρόνος** ο οποίος απαιτείται για την κατασκευή του ευρετηρίου. Αυτό είναι λογικό αφού το L καθορίζει κάθε φορά τη **μέγιστη χωρητικότητα** που μπορεί να έχει η λίστα με την οποία γίνεται η **αναζήτηση** στο γράφο με τον αλγόριθμο **Greedy Search**.

Ας δοκιμάσουμε τώρα να δημιουργήσουμε κάποια ευρετήρια με διαφορετικές τιμές για το R. Συγκεκριμένα θα χρησιμοποιήσουμε για L το $L = 30$ και για τιμές του R θα έχουμε:

$$R \in \{5, 10, 15, \ldots, 60\}$$

Οι χρόνοι κατασκευής των ευρετηρίων με τις παραπάνω τιμές για το R φαίνονται παρακάτω.



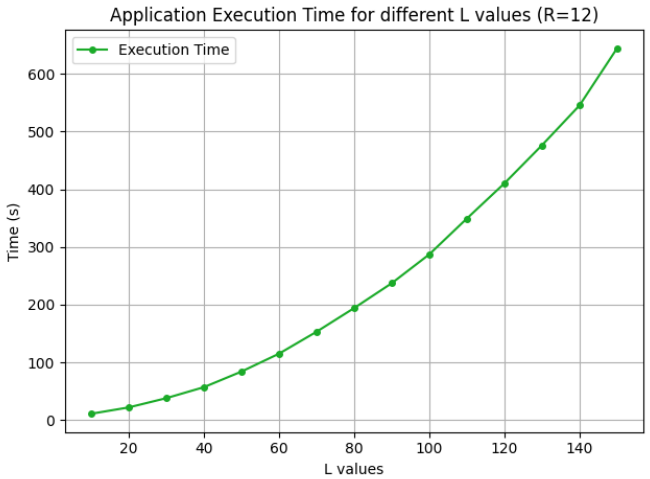
Όπως παρατηρούμε όσο αυξάνεται η **τιμή του R** τόσο αυξάνεται και ο **χρόνος** κατασκευής του ευρετηρίου ξανά. Αυτό είναι επίσης λογικό καθώς το R δείχνει το μέγιστο αριθμό ακμών που θα έχει ο κάθε κόμβος στο γράφο του ευρετηρίου.

Η διαφορά με το L είναι ότι καθώς αυξάνεται η τιμή του L ο ρυθμός αύξησης του χρόνου κατασκευής γίνεται όλο και πιο **μεγάλος**, σε αντίθεση με την αύξηση του R που ο ρυθμός αύξησης του χρόνου κατασκευής γίνεται όλο και πιο **μικρός**.

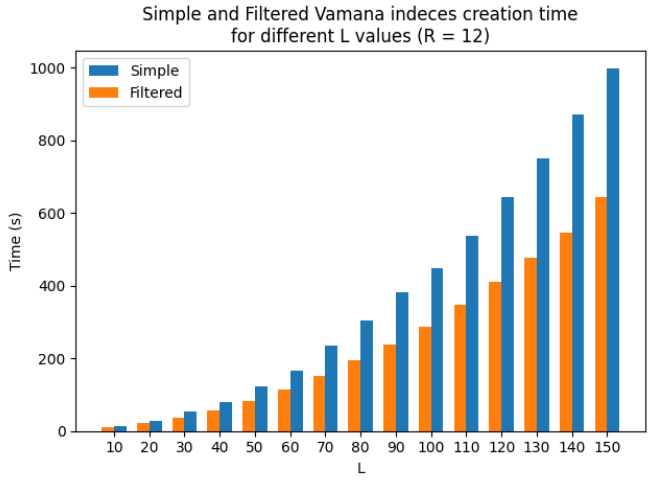
3.1.2 Filtered Vamana

Στη συνέχεια θα μελετήσουμε το **Vamana με φίλτρα** (Filtered Vamana), ο οποίος χρησιμοποιεί δεδομένα από τα σετ δεδομένων **Dummy** και **Contest_1m**.

Συγκεκριμένα θα αναλύσουμε τον αλγόριθμο χρησιμοποιώντας το σετ δεδομένων **Dummy** καθώς περιέχει λιγότερα δεδομένα σε σύγκριση με το **Contest_1m**. Όπως και με τον απλό Vamana, έτσι και στον Filtered εκτελέσαμε τον αλγόριθμο με σταθερό $R = 12$ και $L \in \{10, 20, 30, \ldots, 150\}$, και τα αποτελέσματα που λάβαμε φαίνονται στο παρακάτω διάγραμμα.

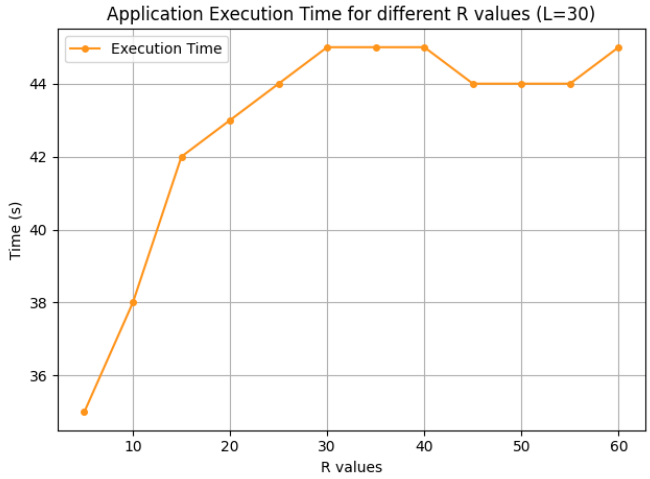


Όπως βλέπουμε όσο αυξάνουμε την τιμή του L , τόσο αυξάνεται και ο χρόνος κατασκευής του ευρετηρίου για το ίδιο λόγο που εξηγήσαμε προηγουμένως. Παρατηρούμε παρόλα αυτά πως οι χρόνοι είναι **καλύτεροι** από ότι αυτοί στον απλό Vamana και αυτό μπορούμε να το δούμε καλύτερα στο παρακάτω διάγραμμα.

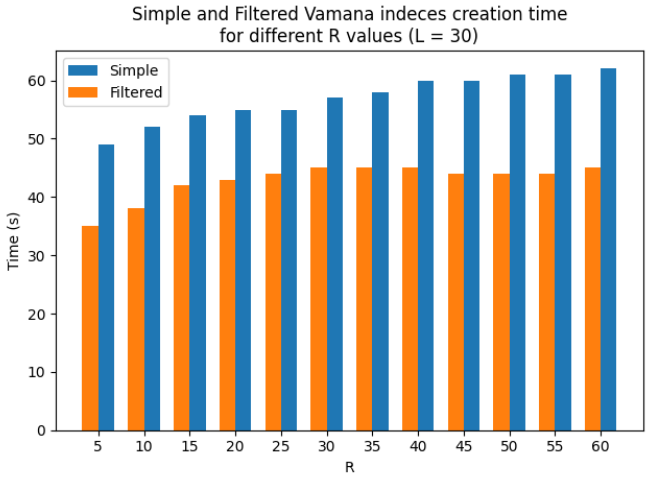


Από το διάγραμμα μπορούμε να συμπεράνουμε πως η κατασκευή ενός **filtered index** παίρνει λιγότερο από ότι η κατασκευή ενός **simple index**. Για παράδειγμα παρατηρούμε πως όταν το L έχει την τιμή $L = 150$, τότε η κατασκευή ενός simple index διαρκεί κάπου στα 1000 δευτερόλεπτα, το οποίο αντιστοιχεί σε περίπου **16 λεπτά**, ενώ η κατασκευή ενός filtered index διαρκεί γύρω στα 600 δευτερόλεπτα, το οποίο σημαίνει περίπου **10 λεπτά**.

Τέλος θα αναλύσουμε επίσης και τους χρόνους κατασκευής με βάση διαφορετικές τιμές του R. Όπως και στον απλό vamana εκτελέσαμε τον αλγόριθμο για $R \in \{5, 10, 15, \ldots, 60\}$ και $L = 30$ και τα αποτελέσματα που πήραμε είναι τα εξής:

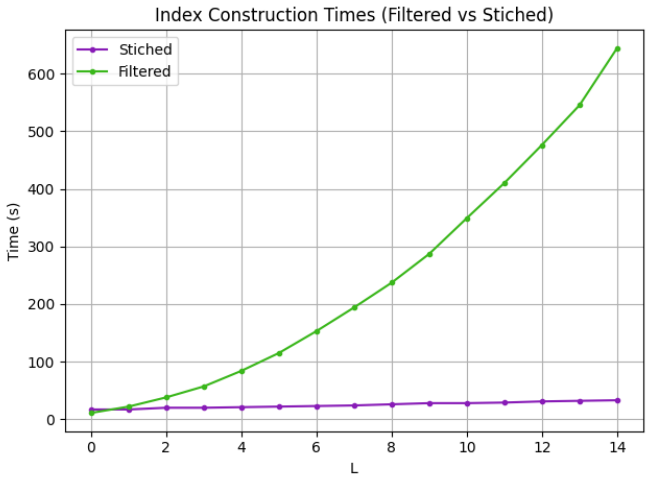


Από το παραπάνω διάγραμμα παρατηρούμε πως ο χρόνος σε σχέση με τον απλό vamana έχει μειωθεί και πάλι καθώς οι τιμές αυτή τη φορά κυμαίνονται στο διάστημα (34, 46) ενώ στον απλό vamana οι τιμές κυμαίνονταν στο διάστημα (48, 62). Αυτή η διαφορά στους χρόνους φαίνεται και παρακάτω.



3.1.3 Stiched Vamana

Τέλος θα μελετήσουμε την απόδοση του αλγορίθμου **Stiched Vamana**, πάνω στο σετ δεδομένων **Dummy**. Όπως και στους προηγούμενους δύο αλγορίθμους, εκτελέσαμε τον stiched αλγόριθμο για $L\_small \in \{10, 20, 30, \ldots, 150\}$, $R\_small = 12$, $R\_stiched = 20$ και τα αποτελέσματα που λάβαμε ήταν ενδιαφέροντα. Συγκεκριμένα συγκρίναμε τον εκάστοτε αλγόριθμο με το filtered και παρατηρήσαμε ακραίες διαφορές στους χρόνους κατασκευής των ευρετηρίων. Πιο συγκεκριμένα τα αποτελέσματα που πήραμε φαίνονται παρακάτω.



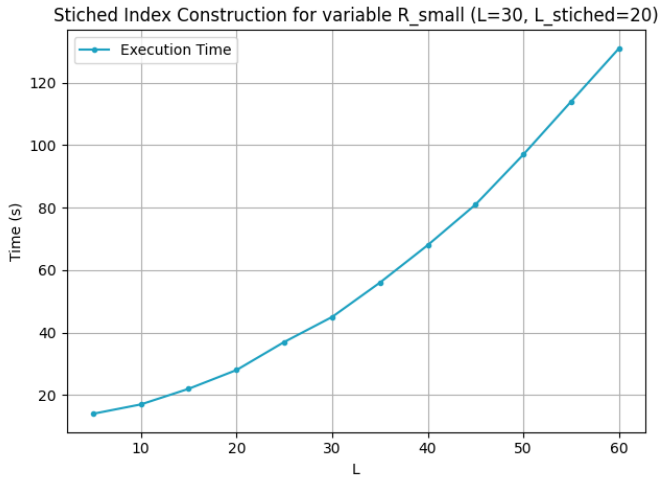
Όπως παρατηρούμε από το παραπάνω διάγραμμα οι χρόνοι κατασκευής του αλγορίθμου stiched είναι **πολύ πιο καλοί** από εκείνους του αλγορίθμου filtered. Συγκεκριμένα βλέπουμε πως όσο αυξάνεται η τιμή του L ο χρόνος κατασκευής ενός filtered index αυξάνεται όλο και περισσότερο,

ενώ ο χρόνος κατασκευής ενός stiched index, οριακά μένει σταθερός, ή τουλάχιστον έχει πολύ μικρό ρυθμό αύξησης.

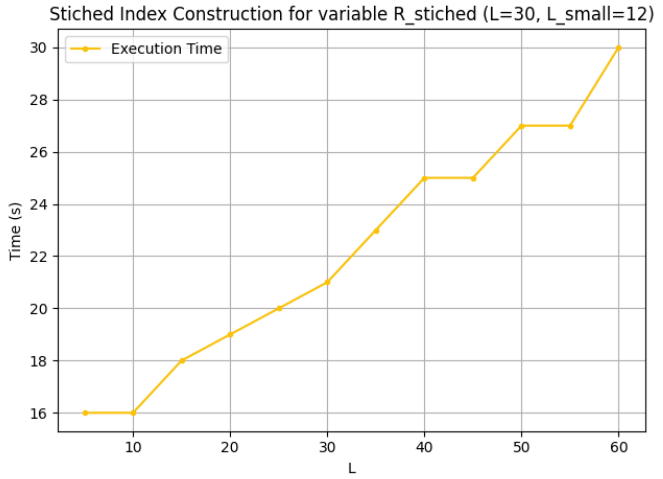
Ας αναλύσουμε όμως και τον stiched αλγόριθμο ως προς την παράμετρο R . Η διαφορά με τους δύο προηγούμενους αλγορίθμους είναι πως αυτή τη φορά έχουμε δύο παραμέτρους R, το R_small και το $R\_stiched$. Αρχικά θα αναλύσουμε τον αλγόριθμο με βάση την παράμετρο

$$R\_small \in \{5, 10, 15, \ldots, 60\}$$

και $L\_small = 30$, $R\_stiched = 20$. Τα αποτελέσματα που λαμβάνουμε είναι τα εξής:



Από το παραπάνω διάγραμμα παρατηρούμε πως αυτή τη φορά ο χρόνος κατασκευής αυξάνεται σημαντικά καθώς το R_small αυξάνεται. Ας παρατηρήσουμε και την συμπεριφορά του χρόνου καθώς αυξάνουμε την παράμετρο $R\_stiched$.



Σε αυτή τη περίπτωση παρατηρούμε πως η απόδοση είναι καλύτερη καθώς ο ρυθμός αύξησης του χρόνου κατασκευής, όσο αυξάνεται το $R\_stiched$ είναι **μικρότερος** από εκείνον που αυξάνεται όσο αυξάνεται το R_small.

3.2 Ακρίβεια Αποτελεσμάτων (Recall)

Αφού αναλύσαμε τους αλγορίθμούς μας ως προς το χρόνο κατασκευής των ευρετηρίων, θα πρέπει επίσης να αναλύσουμε τα **παραγόμενα ευρετήρια** αυτών των αλγορίθμων, ως προς την **ακρίβεια των αποτελεσμάτων τους**. Με πιο απλά λόγια θα αναλύσουμε το **recall** που δεχόμαστε από κάθε ευρετήριο, το οποίο έχει παραχθεί με **διαφορετικές παραμέτρους** κάθε φορά.

3.2.1 Simple Vamana

Αρχικά θα αναλύσουμε τα αποτελέσματα που παράγει ένα ευρετήριο κατασκευασμένο από τον **Απλό Vamana** (Simple Vamana). Στην προηγούμενη ενότητα αναλύσαμε τους χρόνους κατασκευής τέτοιων ευρετηρίων, κάθε φορά με διαφορετικές τιμές για τις παραμέτρους **L** και **R** . Συγκεκριμένα αναλύσαμε τον αλγόριθμο για:

$$L \in \{10, 20, 30, \ldots, 150\},\ R = 12$$

και

$$R \in \{5, 10, 15, \ldots, 60\},\ L = 30$$

Χρησιμοποιώντας αυτές τις παραμέτρους, παράξαμε κάποια ευρετήρια και τα αποθηκεύσαμε τοπικά στα μηχανήματά μας. Σε αυτήν την ενότητα θα χρησιμοποιήσουμε εκείνα τα παραγόμενα ευρετήρια και θα αναλύσουμε την ακρίβεια των αποτελεσμάτων με τις ίδιες τιμές των L και R.

Θα ξεκινήσουμε με τα αποτελέσματα που λάβαμε αφού εκτελέσαμε τον απλό Vamana με **διαφορετικές τιμές** του L και **σταθερό** R. Το **μέσο recall** που λάβαμε για **100 query vectors** περιγράφεται στον παρακάτω πίνακα.

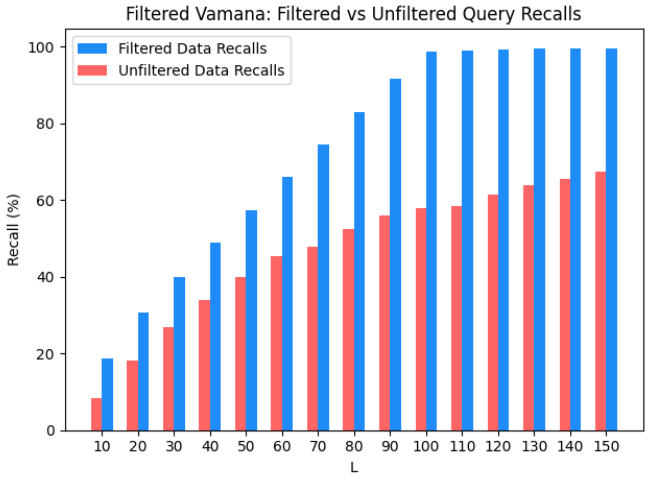
L values	Mean Recall	Mean Search Time
10	10%	0.00s
20	20%	0.00s
30	30%	0.00s
40	40%	0.00s
50	50%	0.00s
60	60%	0.01s
70	70%	0.013s
80	80%	0.025s
90	90%	0.031s
100	97.2%	0.04s
110	98.34%	0.06s
120	99.45%	0.075s
130	99.82%	0.08s
140	99.91%	0.09s
150	99.97%	0.097s

Από τον παραπάνω πίνακα παρατηρούμε πως το **μέσο recall** στην αρχή αυξάνεται γραμμικά σε σχέση με την παράμετρο L , ενώ στη συνέχεια δείχνει να συγκλίνει προς το 100%, το οποίο είναι πολύ θετικό. Βλέπουμε επίσης στην τρίτη στήλη του πίνακα το **χρόνο** που διήρκησε η αναζήτηση μέσα στο ευρετήριο, ο οποίος δείχνει να είναι λιγότερος από 0.1 δευτερόλεπτα.

Μπορούμε επομένως να πούμε πως για $L > 100$ ο αλγόριθμος **Simple Vamana** είναι **αποτελεσματικός** και

αποδοτικός.

> **Παρατήρηση**
>
> Με την αλλαγή της παραμέτρου **R** δεν παρατηρήσαμε κάποια σημαντική διαφορά στα recalls των αποτελεσμάτων. Υπενθυμίζουμε πως δημιουργήσαμε διάφορα ευρετήρια για διαφορετικές τιμές του R με το L να είναι σταθερό και ίσο με $L = 30$. Το αποτέλεσμα που λάβαμε δοκιμάζοντας τα ευρετήρια με τιμές $R \in \{5, 10, 15, \ldots, 150\}$ ήταν σταθερό και ίσο με
>
> $$Recall = 30$$



3.2.2 Filtered Vamana

Τα δεδομένα που χρησιμοποιήθηκαν για το **Filtered Vamana** ήταν τα **Dummy** και **Contest_1m**. Τα συγκεκριμένα σετ περιέχουν δεδομένα τα οποία χωρίζονται στις κατηγορίες που περιγράψαμε σε προηγούμενη ενότητα. Σε αυτήν την εργασία κληθήκαμε να χρησιμοποιήσουμε δεδομένα με συγκεκριμένα Categorical Attributes. Πιο συγκεκριμένα:

$$C \in \{0, 1\}$$

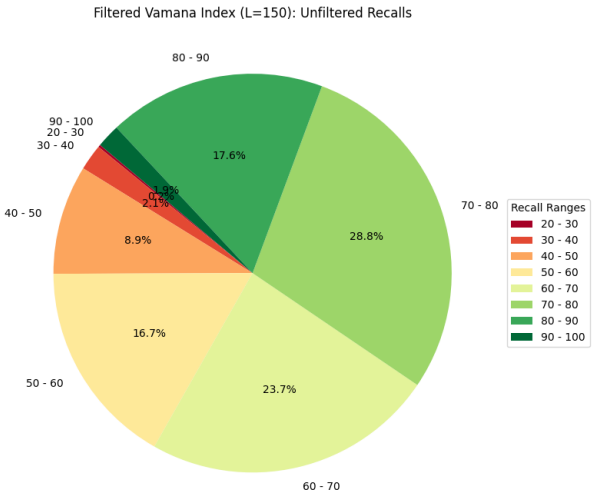
όπου το **0** υποδείχνει ότι το εκάστοτε base vector δεν έχει φίλτρο, ενώ το **1** σημαίνει πως έχει. Μελετήσαμε το συγκεκριμένο αλγόριθμο για κάθε τιμή του $L \in \{10, 20, 30, \ldots, 150\}$ και εξάγαμε αποτελέσματα τόσο για τα **unfiltered** $(C = 0)$, όσο και για τα filtered $(C = 1)$ query vectors. Συγκεκριμένα εξάγαμε το **μέσο recall** και για τις δύο κατηγορίες ερωτημάτων (queries) και τα αποτελέσματα που λάβαμε ήταν τα εξής:

L values	Unfiltered Recalls	Filtered Recalls
10	8.44%	18.63%
20	18.02%	30.55%
30	26.83%	39.98%
40	34.00%	48.88%
50	40.04%	57.45%
60	45.45%	65.98%
70	47.79%	74.51%
80	52.33%	83.02%
90	55.97%	91.49%
100	57.88%	98.60%
110	58.41%	98.94%
120	61.36%	99.34%
130	63.96%	99.43%
140	65.41%	99.53%
150	67.38%	99.61%

Από τον παραπάνω πίνακα παρατηρούμε πως το **recall rate** για τα **filtered** query vectors, είναι συγκριτικά **καλύτερο** από εκείνο των **unfiltered** query vectors. Αυτό μπορούμε να το δούμε καλύτερα και στο παρακάτω διάγραμμα.

Από το διάγραμμα παρατηρούμε πως όχι μόνο το recall rate των filtered query vectors είναι καλύτερο από εκείνο των unfiltered query vectors αλλά και ότι το πρώτο τείνει προς το **100%** ενώ το δεύτερο τείνει προς το **60%** με **70%**.

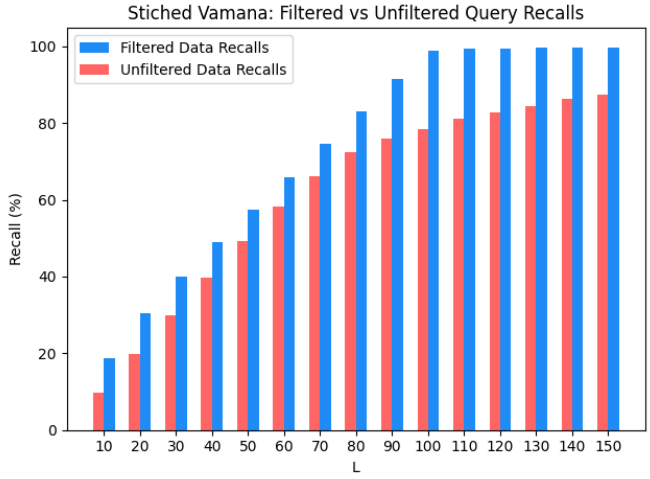
Ας αναλύσουμε λίγο περισσότερο το ευρετήριο το οποίο έχει κατασκευαστεί με $L = 150$ όπου δείχνει να είναι το καλύτερο. Αρχικά για τα **filtered** query vectors , το mean recall rate φαίνεται να είναι **99.61%**. Μελετώντας λίγο παραπάνω το συγκεκριμένο ευερετήριο παρατηρήσαμε πως για όλα τα query vectors μέσα στο δετ δεδομένων, το recall rate τους ήταν πάνω από 90%. Παρόλα αυτά το ενδιαφέρον μας εστιάζει στα **unfiltered** query vectors όπου το recall rate τους δεν ήταν σταθερό όπως πριν. Συγκερκιμένα τα ποσοστά που εξάγαμε φαίνονται στο παρακάτω διάγραμμα.



Το παραπάνω διάγραμμα δείχνει το πλήθος των query vectors με recall rate σε ένα συγκεκριμένο διάστημα. Με **κόκκινο** χρώμα αναπαρίστανται τα **χαμηλά** recalls και με **πράσινο** χρώμα αναπαρίστανται τα **υψηλά** recalls. Παρατηρούμε επομένως πως τα περισσότερα query vectors πέτυχαν recall στο διάστημα **70 - 80%**.

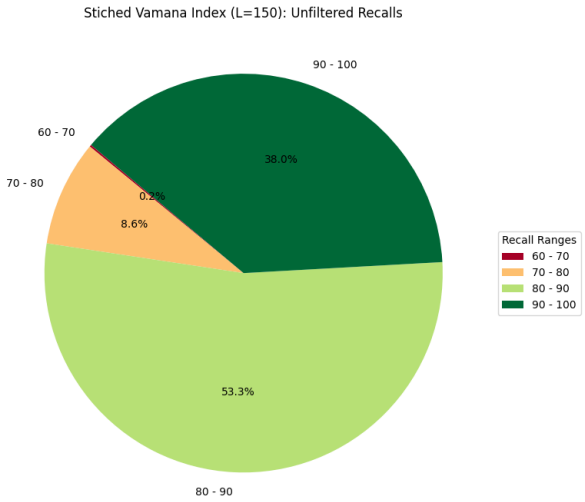
3.2.3 Stiched Vamana

Όπως και στο Filtered Vamana έτσι και στο Stiched Vamana χρησιμοποιήθηκαν τα **Dummy** και **Contest_1m** σετ δεδομένων. Τα αποτελέσματα που λάβαμε για το συγκεκριμένο αλγόριθμο ήταν τα εξής:



Με την παραπάνω οπτικοποίηση των αποτελεσμάτων, παρατηρούμε πως το **mean recall rate** των **unfiltered query vectors** έχει υποστεί ραγδαία αύξηση. Συγκεκριμένα παρατηρούμε στη περίπτωση του stiched index το recall των unfiltered queries τείνει προς το **80%** με **90%**, σε αντίθεση με τη περίπτωση του filtered index όπου έτεινε προς το **60%** με **70%**. Αυτό φυσικά είναι πολύ θετικό αν λάβουμε υπόψη και το χρόνο που χρειάζεται για την κατασκευή ενός stiched ευρετηρίου, το οποίο όπως είδαμε παίρνει πολύ λιγότερο συγκριτικά με ένα filtered ευρετήριο.

Ας δούμε όμως και αναλυτικά τι ισχύει για τα unfiltered query vectors όπως και πριν.



Όπως παρατηρούμε από το διάγραμμα παραπάνω, τα αποτελέσματα για τα unfiltered queries είναι πράγματι αρκετά καλύτερα σε σχέση με το filtered index. Καταρχάς το μικρότερο recall rate που λαμβάνουμε κυμαίνεται στο **60%** με **70%**, και κατά δεύτερον το ποσοστό των query vectors που λαμβάνουν τέτοιο recall είναι **πολύ μικρό** και συγκεκριμένα 0.2%. Το μεγαλύτερο ποσοστό των queries λαμβάνει recall rate στο διάστημα **80%** με **90%**, το οποίο είναι επίσης πολύ θετικό.

> **Σημείωση**
>
> Τόσο για το **filtered index**, όσο και για το **stiched index**, μελετήσαμε τα recall rates που λαμβάνουν τα filtered query vectors, για $L = 150$, και τα αποτελέσματα που πήραμε ήταν **όλα πάνω από 90%**.

3.3 Σχολιασμός Αποτελεσμάτων

Μελετώντας και αναλύοντας τους παραπάνω αλγορίθμους, καταλήξαμε σε κάποια συμπεράσματα τα οποία περιγράφονται παρακάτω:

1. Ο αλγόριθμος **Simple Vamana** αν και φέρνει καλά αποτελέσματα είναι πολύ αργός.

2. Τα αποτελέσματα που φέρνει ο **Filtered Vamana** συγκριτικά με αυτά που φέρνει ο **Simple Vamana** είναι λίγο χειρότερα, αλλά η κατασκευή του filtered ευρετηρίου διαρκεί πολύ λιγότερο. Αυτό αμέσως καθιστά τον αλγόριθμο Filtered Vamana **καλύτερο** από τον Simple Vamana.

3. Ο αλγόριθμος Stiched Vamana ο οποίος λειτουργεί πάνω στα ίδια δεδομένα με τον Filtered Vamana φαίνεται όχι μόνο να κατασκευάζεται πολύ πιο γρήγορα από τον Filtered Vamana αλλά παράγει επίσης και καλύτερα αποτελέσματα. Αυτό αμέσως τον καθιστά τον καλύτερο αλγόριθμο από τους τρεις.

4 Βελτιστοποιήσεις

Έχοντας αναλύσει πλήρως τους αλγορίθμους ως προς την απόδοση και την αποτελεσματικότητά τους, επιχειρήσαμε να βελτιστοποιήσουμε επίσης την εφαρμογή με διάφορους τρόπους. Οι προτεινόμενες αλλά και δικές μας μέθοδοι που ακολουθήσαμε προκειμένου να βελτιώσουμε τους αλγορίθμους και ως προς το χρόνο αλλά και ως προς την ακρίβεια των αποτελεσμάτων, ήταν οι εξής:

1. Αποθήκευση αποστάσεων σε πίνακα
2. Αρχικοποίηση γράφων με τυχαίες ακμές
3. Αρχικοποίηση medoid στον Vamana με τυχαία σημεία
4. Παράλληλη Εκτέλεση

Ας δούμε αναλυτικά τα αποτελέσματα των παραπάνω μεθόδων.

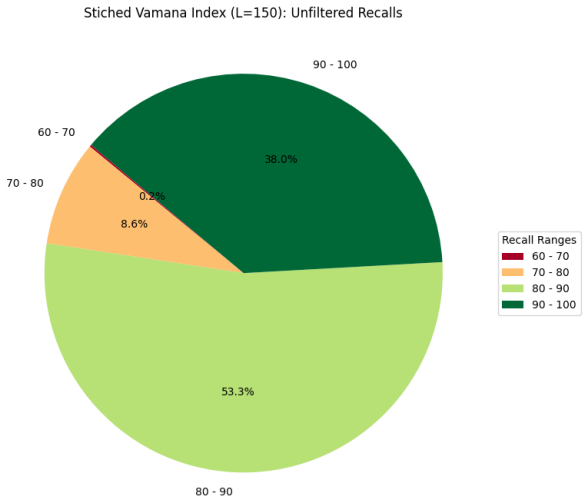
4.1 Αποθήκευση Αποστάσεων

Μία αρχική ιδέα η οποία αργότερα εξελίχθηκε σε γενικό κανόνα για μείωση του χρόνου κατασκευής των ευρετηρίων, με κάποιους περιορισμούς, ήταν η **αποθήκευση των αποστάσεων** (Distance Cache).

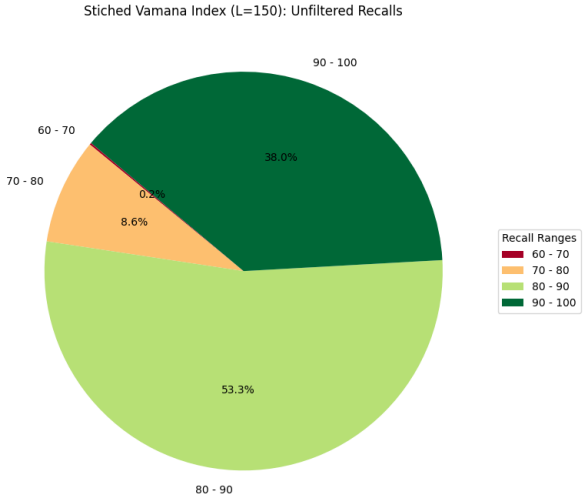
Βασική Ιδέα

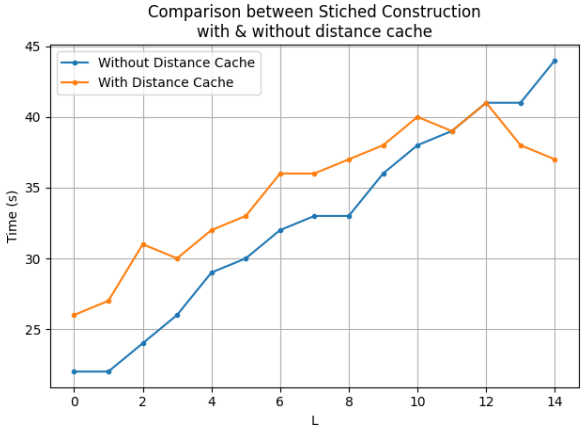
Η βασική ιδέα ήταν πως η επαναχρησιμοποίηση κάποιων αποστάσεων θα έφερνε σαν αποτέλεσμα καλύτερο χρόνο ως προς την κατασκευή του ευρετηρίου, σε αντίθεση με τον επαναυπολογισμό τους. Πράγματι μετά από κάποια πειράματα συνειδητοποιήσαμε πως ο υπολογισμός των αποστάσεων μεταξύ διανυσμάτων μακροπρόθεσμα θέτει την δημιουργία ενός ευρετηρίου ως μια πολύ ακριβή διαδικασία όσον αφορά το χρόνο. Μία λύση σε αυτό το πρόβλημα ήταν ο υπολογισμός **όλων των αποστάσεων** και η αποθήκευσή τους σε μία προσωρινή δομή με στόχο την ανάκτησή τους σε σταθερό χρόνο $O(1)$. Η δομή η οποία χρησιμοποιήθηκε για αυτή τη βελτιστοποίηση ήταν ο **πίνακας** (array).

Ακολουθώντας λοιπόν αυτή τη μέθοδο παρατηρήσαμε σημαντικές διαφορές στους χρόνους κατασκευής των ευρετηρίων και πιο συγκεκριμένα παρατηρήσαμε μείωση των χρόνων περίπου στο **μισό**. Για παράδειγμα μία τυπική εκτέλεση του Simple Vamana αλγορίθμου, χωρίς κάποια βελτιστοποίηση έπαιρνε κάπου στα **14 λεπτά**. Με την προσθήκη της προηγούμενης μεθόδου, ο χρόνος κατασκευής του ευρετηρίου μειώθηκε στα **7 με 8 λεπτά**. Το φαινόμενο αυτό φαίνεται και καλύτερα στο παρακάτω διάγραμμα.



Βελτίωση του χρόνου παρατηρήσαμε επίσης και στους αλγορίθμους Filtered και Stiched Vamana και τα αποτελέσματα φαίνονται παρακάτω.





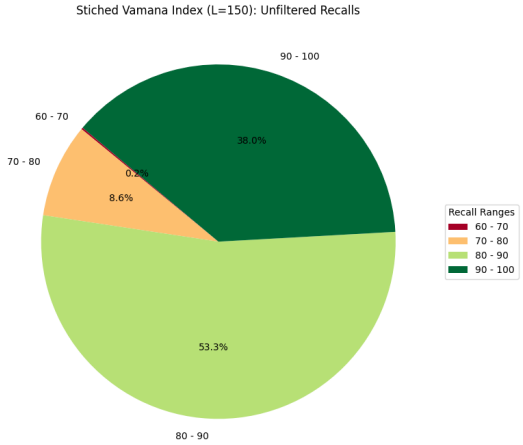
Προσοχή

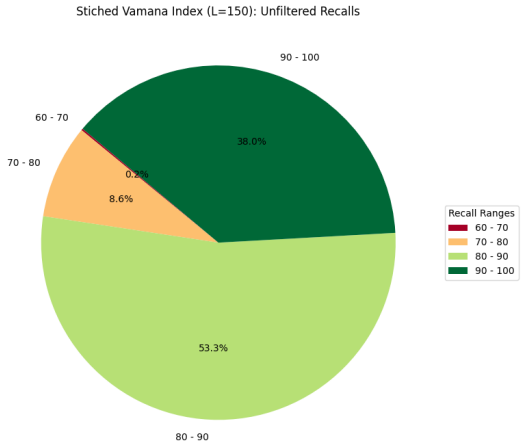
Η συγκεκριμένη μέθοδος, αν και βελτιστοποιεί την εφαρμογή, όσον αφορά το χρόνο κατασκευής. Παρόλα αυτά αυξάνει σημαντικά το χώρο μνήμης του προγράμματος. Για παράδειγμα για ένα σετ δεδομένων με 10.000 διανύσματα, το πρόγραμμα θα δεσμεύσει μνήμη για έναν πίνακα 10.000×10.000, όπου από αυτόν τον πίνακα είναι πολύ λογικό να μην χρησιμοποιούνται όλες οι αποστάσεις μέσα στο πρόγραμμα. Σε αυτή τη περίπτωση δεν υπάρχει κάποιο θέμα. Τα προβλήματα αρχίζουν όταν δουλεύουμε με σετ δεδομένων με περισσότερα διανύσματα.

4.2 Αρχικοποίηση με τυχαίες Ακμές

Η δεύτερη μέθοδος είναι η αρχικοποίηση του γράφου με τυχαίες ακμές. Τη μέθοδο αυτή την εφαρμόσαμε στους αλγορίθμους Filtered Vamana και Stiched Vamana, μιας και οι αρχικοί αλγόριθμοι προτείνουν οι γράφοι να δημιουργούνται **κενοί**.

Η προσθήκη αυτής της μεθόδου βελτίωσε σημαντικά τα recalls των αποτελεσμάτων, χωρίς καμία διαφορά στο χρόνο κατασκευής των ευρετηρίων. Πιο αναλυτικά τα recalls φαίνονται στα παρακάτω διαγράμματα.





4.3 Αρχικοποίηση με τυχαία Medoid

Η τρίτη μέθοδος αφορά την αρχικοποίηση του medoid με τυχαία σημεία. Στην προσπάθειά μας να εφαρμόσουμε αυτή τη μέθοδο δεν παρατηρήσαμε κάποια βελτίωση αλλά ούτε χειροτέρευση στο χρόνο κατασκευής και στην ακρίβεια των αποτελεσμάτων.

4.4 Παράλληλη Εκτέλεση

Η τελευταία μέθοδος για βελτιστοποίηση της εφαρμογής είναι η **παράλληλη εκτέλεση** για την οποία αξιοποιήσαμε τα **threads** της **C++**.

Συγκεκριμένα προσθέσαμε παράλληλη εκτέλεση σε διάφορα σημεία του προγράμματος:

1. Υπολογισμός Αποστάσεων
2. Υπολογισμός Filtered Medoid
3. Υπολογισμός Stiched Vamana

4.4.1 Υπολογισμός Αποστάσεων

Αρχικά προσθέσαμε **multithreading** στον υπολογισμό των αποστάσεων. Η ιδέα ήταν να μην υπολογίζουμε όλες τις αποστάσεις σειριακά μιας και δεν υπάρχει κάποια σχέση μεταξύ τους. Αυτό φυσικά **μείωσε** το χρόνο των υπολογισμών τους αλλά όχι την κατασκευή του ευρετηρίου.

4.4.2 Υπολογισμός Filtered Medoid

Για τον υπολογισμό των filtered medoids προσθέσαμε επίσης μυλτιτηρεαδινγ μιας και ο προτεινόμενος αλγόριθμος

5 References

1. Acutest GitHub Repository
2. Embedding Graph in Euclidean Space - Stack Overflow
3. Google Test Documentation
4. FAISS Demo: SIFT1M
5. YouTube Video: How to Build a KNN Search Engine
6. Vector Databases - The Data Quarry
7. Medoid - Wikipedia
8. Compiler Optimizations - Medium Article
9. Divide and Conquer KNN Paper