## 



Ανάπτυξη λογισμικού για πληροφοριακά συστήματα

09.01.2025

Anastasia Orfanoudaki 1115202100129

Ioulia Osadtsiv 1115202100130

Alexandros Koutras 1115202100073

# 

# Περιεχόμενα

[**Περιεχόμενα 2**](#_17kpml5yj34e)

[**Greedy Search Overview 3**](#_au51mny0sx6)

[Filtered Greedy Search Version 4](#_ea7qdqnjb3ft)

[**Robust Prune Overview 4**](#_2d0gibtt0rwh)

[Filtered Robust Prune Version 5](#_io3ldrlhqp1z)

[**Vamana Indexing overview 5**](#_gxopy5l4tfg)

[1. Αρχικοποιηση γράφου 5](#_n0jb9hd0jefu)

[2. Εύρεση medoid και K-means clustering 5](#_5p675l9tfxjd)

[3. Fisher-yates algorithm 9](#_e0bq9n7j0hz5)

[**Filtered Vamana algorithm overview 10**](#_wv4xoj4evmn2)

[**Συγκριση Vamana-Filtered Vamana** 11](#_bi64kxsjtsz8)

[**Συγκριση Vamana-StitchedVamana** 13](#_carqbe4ojpjh)

[**Σύγκριση FilteredVamana –StitchedVamana** 14](#_6a8yy7dgmvxl)

[**Find Medoid for filters 14**](#_uo9lwuz0l86s)

[Ανάλυση Λειτουργίας 14](#_8pcb5e39sb1j)

[Παρατηρήσεις 16](#_x5py61vndvdo)

[**Stitched Vamana overview 16**](#_wjbk2f29knem)

[**Threads in Stitched Vamana 16**](#_vlnhqa6osf53)

[**Threads in main function 17**](#_5pfy5b3g6li)

[**Threads for distance calculations 17**](#_3at9u9s4e0vp)

[1. Υπολογισμός Αποστάσεων με Threads 17](#_wpf37p809sc9)

[2. Πειραματικά Αποτελέσματα 18](#_xyhf8nohwmtc)

[3. Πλεονεκτήματα Παραλληλοποίησης 18](#_rrvrcdrqrv30)

[4. Συμπέρασμα 18](#_afumxxsu9c9u)

[**Ιστόγραμμα χρόνου εκτέλεσης με threads 19**](#_378xxxt3m6gh)

[**Αρχικοποίηση γράφων με τυχαίες ακμές 19**](#_5uqvjbmecldd)

[Βελτιστοποίηση FilteredVamana **19**](#_ovxxsz20npv)

[Βελτιστοποίηση StichedVamana 20](#_xi0p9i9warfh)

[**Αρχικοποίηση medoid στον Vamana με τυχαία σημεία** 21](#_6gtm64g6rb5j)

# Greedy Search Overview

Για την υλοποίηση της Greedy Search , χειριζόμαστε τρία σύνολα δεδομένων. Ένα για την την υλοποίηση του L, το οποίο χρησιμοποιούμε για την διατήρηση των κοντινότερων κόμβων (και αρχικοποιείται με τον αρχικό κόμβο s), ένα σύνολο για την αποθήκευση των κόμβων που έχουμε επισκεφθεί ( V ) και τέλος ένα σύνολο για την αποθήκευση των αποστάσεων των σημείων ( distances ). Το L επιλέξαμε να το υλοποιήσουμε με vector<Node\*>. Το V το υλοποιήσαμε με unordered\_set<node\*>, έτσι ώστε να αποφύγουμε τις διπλότυπες καταγραφές ενός κόμβου. Τέλος για την αποθήκευση των αποστάσεων αρχικά επιλέξαμε priority queue αλλά παρατηρήσαμε ότι με το map η εκτέλεση ήταν αποδοτικότερη. Οπότε ο υπολογισμός των αποστάσεων γίνεται με τη χρήση της ευκλείδειας και η αποθήκευση τους στο map με κλειδί το node\* και τιμή την απόσταση.

Αυτό που κάνει λοιπόν ο αλγόριθμος τελικά είναι:  
1. Εκτελείται ένας επαναληπτικός βρόχος όσο υπάρχουν στοιχεία στο σύνολο L, τα οποία δεν έχουμε επισκεφθεί. Για τον βρόχο αυτό χρησιμοποιούμε τη συνάρτηση any\_of(L.begin(), L.end(), [&](Node\* p) { return V.find(p) == V.end(); })

Της c++ , η οποία καθορίζει εάν ακόμη και ένα στοιχείο σε μία δεδομένη περιοχή ικανοποιεί ένα καθορισμένο κριτήριο.

2. Βρίσκουμε το πλησιέστερο σημείο στο query ( p\_star ) από τα σημεία του L που δεν έχουμε ήδη επισκεφθεί. Για τον σκοπό αυτό, για κάθε κόμβο αν δεν υπάρχει στο map, τότε υπολογίζουμε την απόσταση με την ευκλείδεια αλλιώς χρησιμοποιούμε την ήδη υπολογισμένη τιμή που έχουμε αποθηκεύσει στον map και συγκρίνοντας με την εκάστοτε minimum τιμή καταλήγουμε στο καθολικά πλησιέστερο σημείο.

3. Αφού πλέον έχουμε πλέον βρει το p\_star το επισημαίνουμε ως επισκέψιμο (εισαγωγή στο V) και προσθέτουμε όλους τους γείτονες του τελευταίου, οι οποίοι όμως δεν ανήκουν στο σύνολο V, στο σύνολο L.

4. Αν το σύνολο L έχει μέγεθος μεγαλύτερο της παραμέτρου l της συνάρτησης τότε ταξινομούμε το σύνολο L με τη χρήση της συνάρτησης

nth\_element(L.begin(), L.begin() + list\_size, L.end(), [&](Node\* a, Node\* b) {

Return euclidea(a, x\_q) < euclidean(b, x\_q);

});

, η οποία αναδιατάσσει τη λίστα με τέτοιο τρόπο ώστε τα στοιχεία μέχρι την νιοστή θέση να είναι αυτά που θα έπρεπε εαν ταξινομούσαμε τη λίστα. Έπειτα κάνουμε resize το σύνολο L ώστε να κρατήσουμε τα l στοιχεία. Και κλείνουμε τον επαναληπτικό βρόχο

5. Έξω από την επανάληψη τώρα, άμα το σύνολο L είναι μεγαλύτερο από τη παράμετρο k της συνάρτηση με ίδιο τρόπο με το βήμα 4 διατηρούμε τα k στοιχεία του συνόλου , τα οποία είναι τελικά και τα k κοντινότερα σημεία στο query και τα οποία θα επιστρέψει η συνάρτηση.

Τελικά με αυτή την εκδοχή της greedy search η αναζήτηση είχε 1.5s χρόνο εκτέλεσης.

### **Filtered Greedy Search Version**

Στην filtered εκδοχή της συνάρτησης οι τροποποιήσεις που χρειάστηκαν ήταν δύο. Αρχικά η προσθήκη ενός επαναληπτικού βρόχου στον οποίο εκτελούμε έλεγχο για το αν οι αρχικοί κόμβοι έχουν ίδια φίλτρα με τον κόμβο ερώτησης. Και δεύτερον, η προσθήκη ελέγχου για τα φίλτρα στον έλεγχο του βήματος 3 πριν την προσθήκη των γειτόνων του p\_star στο σύνολο L.

# Robust Prune Overview

Για την υλοποίηση της Robust Prune χρησιμοποιήσαμε ένα set για να αποθηκεύουμε τα IDs των κόμβων, έτσι ώστε να μην υπάρχουν διπλότυπα. Στην αρχή, προσθέτουμε όλους τους πιθανούς γείτονες που μας δίνει η greedy search στο set και μετά προσθέτουμε τους ήδη γείτονες του κόμβου που εξετάζουμε στους πιθανούς γείτονες, αποφεύγοντας την εισαγωγή κάποιου ήδη υπάρχοντος με τη χρήση του set.

Στη συνέχεια, κάνουμε έναν έλεγχο για να βεβαιωθούμε ότι το ίδιο το node δεν είναι μέσα στους πιθανούς γείτονες. Μετά, διαγράφουμε όλους τους γείτονες του κόμβου και υπολογίζουμε την απόσταση του κάθε πιθανού γείτονα από τον κόμβο. Τα ταξινομούμε από το μικρότερο προς το μεγαλύτερο και πλέον είμαστε έτοιμοι να ξεκινήσουμε τη διαδικασία pruning.

Επαναλαμβάνουμε τη διαδικασία όσο υπάρχουν ακόμα κόμβοι μέσα στους πιθανούς γείτονες και παίρνουμε τον πιο κοντινό κάθε φορά. Τον προσθέτουμε στους γείτονες του κόμβου και τον αφαιρούμε από τους πιθανούς γείτονες. Ύστερα, ελέγχουμε ότι δεν έχουμε φτάσει το όριο των γειτόνων που μπορεί να έχει ο κόμβος. Αν το έχουμε φτάσει, τερματίζεται η διαδικασία αλλιώς συνεχίζουμε. Διατρέχουμε όλους τους πιθανούς γείτονες, υπολογίζοντας για τον καθένα την απόσταση του κοντινότερου που πήραμε από την πρώτη λούπα με τον κόμβο (από τους πιθανούς που διατρέχουμε), πολλαπλασιασμένη με τον συντελεστή α.

Εάν αυτός ο αριθμός είναι μικρότερος από την απόσταση του πιθανού γείτονα από τον κόμβο, τότε τον διαγράφουμε από τους πιθανούς γείτονες. Αν όχι, προχωράμε στον επόμενο πιθανό γείτονα.

### **Filtered Robust Prune Version**

Όσον αφορά τη Filtered Robust Prune, η μόνη αλλαγή που χρειάστηκε να γίνει είναι ότι, πριν ξεκινήσει η διαδικασία pruning, προστέθηκε ένας έλεγχος ώστε να βεβαιωθούμε ότι ο κόμβος στον οποίο θα εφαρμόσουμε τη διαδικασία pruning έχει τουλάχιστον ένα κοινό φίλτρο είτε με τον κοντινότερο κόμβο που επιλέξαμε από την προηγούμενη επανάληψη είτε με τον κόμβο που εξετάζουμε. Ωστόσο, για τις απαιτήσεις της εργασίας, αυτός ο έλεγχος δεν είναι απαραίτητος, καθώς κάθε κόμβος έχει μόνο ένα φίλτρο.

# Vamana Indexing overview

### **1. Αρχικοποιηση γράφου**

Πριν την εκτέλεση του αλγορίθμου Vamana απαιτείται η αρχικοποίηση ενός τυχαίου κατευθυνόμενου γράφου G με R γείτονες για κάθε κόμβο. Για την κατασκευή του υλοποιήσαμε βοηθητική συνάρτηση

void initializeRandomGraph(vector<Node\*>& nodes, unsigned int R)

, η οποία εξασφαλίζει ότι οι γείτονες ενός γράφου δεν είναι διπλότυποι και δεν ταυτίζονται με τον εαυτό του μέσω συνθήκης

if (random\_index != node->id && neighbors.find(random\_index) == neighbors.end())

### **2. Εύρεση medoid και K-means clustering**

Ως σημείο αφετηρίας του Vamana επιλέγουμε τον ενδιάμεσο s, ο οποίος είναι το πιο κεντρικό σημείο, μέσω κλήσης της

int medoid = approximateMedoid(nodes, k) ;

Μετά από προσπάθεια εύρεσης του medoid με εξέταση κάθε κόμβου, καταλήγαμε σε ακρίβεια, αλλά η διαδικασία ήταν αρκετα χρονοβόρα με πολυπλοκότητα χρόνου Ο(n^2). Γι'αυτό καταλήξαμε στη παρακάτω λογική, η οποία στηρίζεται στο k-means clustering [K-means\_clustering](https://en.wikipedia.org/wiki/K-means_clustering), περιορίζοντας αρκετά την αναζήτηση του medoid και έτσι βελτιώνοντας πια την πολυπλοκότητα και άρα την αποδοτικότητα. Η

int approximateMedoid( const vector<Node\*>& nodes, int k)

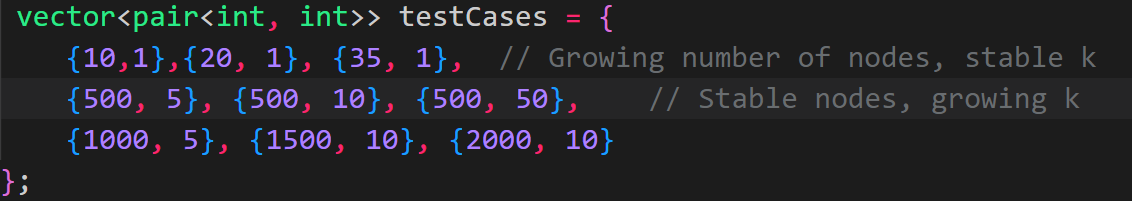
σπάει τα nodes σε k clusters

auto clusters = kMeansClustering (nodes, k);

και επιλέγει το μεγαλύτερο cluster. Έπειτα υπολογίζει το centroid του μεγαλύτερου cluster

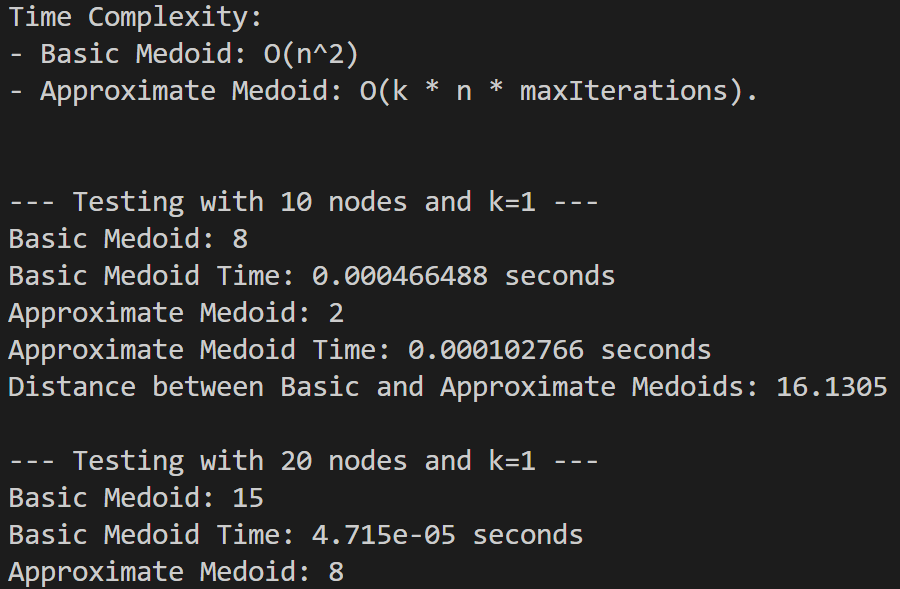
Node\* centroid = findCentroid(clusters[largestClusterIndex]);

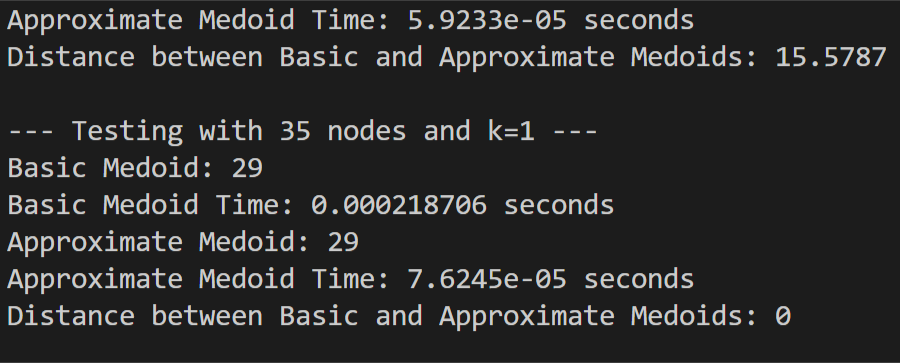
το οποίο θα αποτελέσει reference point για την medoid. Και τελικά βρίσκει τον κόμβο από το συγκεκριμένο cluster που έχει την μικρότερη απόσταση από το centroid και το ορίζει ως medoid. Συνεπώς με εύρεση του medoid μέσω centroid, αποφύγαμε το κόστος υπολογισμού απόστασης για κάθε κόμβο με κάθε γείτονά του, το οποίο αποτελούσε χρονοβόρα διαδικασία, αλλά χάνουμε πιθανόν σε ακρίβεια αφού παραβλέπουμε την αναζήτηση medoid για άλλα clusters.(Πιθανόν πρόβλημα σε clusters με ίδια μεγέθη).

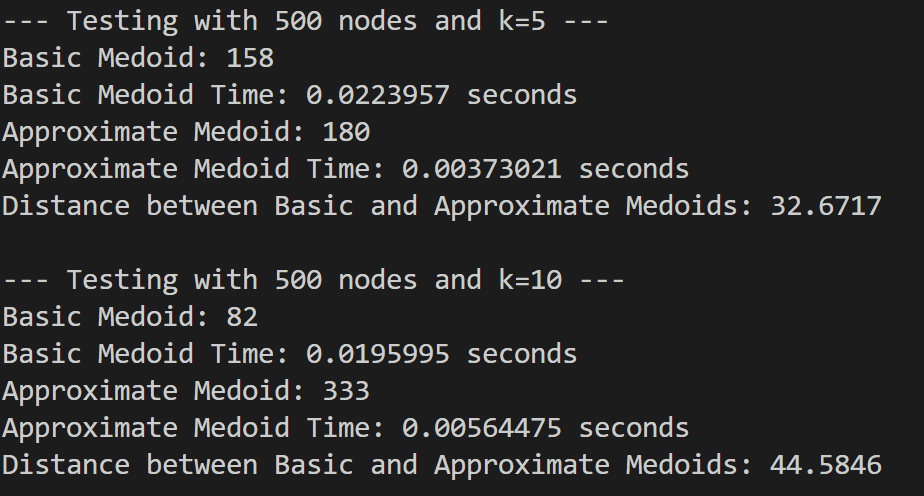
Για τη σύγκριση των δύο αυτών αλγορίθμων (basic και approximate medoid) δημιουργήσαμε αρχείο medoid\_data.cpp το οποίο για διαφορετικά test cases

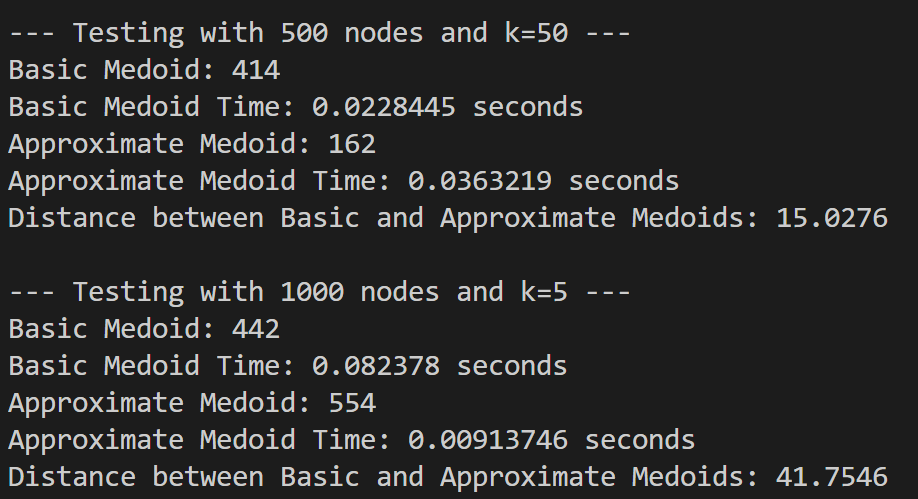
τρέχει τους δύο αλγορίθμους και υπολογίζει πόσο χρόνο απαιτεί ο καθένας, τα medoid και την απόκλιση της approximate medoid από την basic.

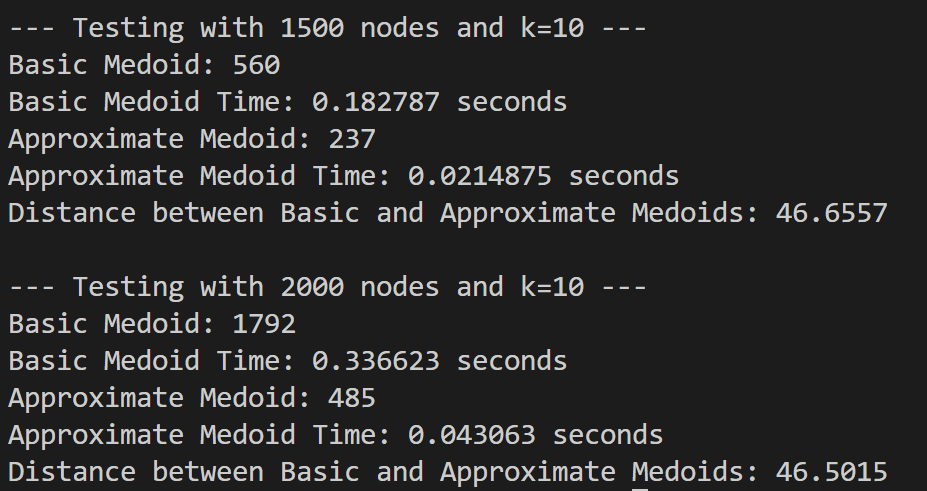
**Αποτελέσματα:**









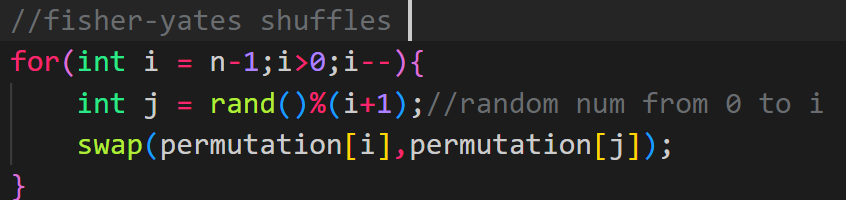


Παρατηρούμε στο χρόνο εκτέλεσηςότι το clustering είναι ταχύτερο για μικρα k ( μικρό k σχετικό με τον αριθμό των κόμβων) και η διαφορά γίνεται αρκετά μεγάλη όσο μεγαλώνει το n, το οποίο είναι λογικό αφού ο πρώτος αλγόριθμος είναι Ο(n^2). Επίσης παρατηρούμε ότι για μεγαλύτερα data sets αυξάνεται και η απόσταση από το ακριβές medoid, κάτι που περιμέναμε αφού η ταχύτητα με την ακρίβεια για την approximate είναι αντιστρόφως ανάλογα.

### **3. Fisher-yates algorithm**

Κατά την εκτέλεση της Vamana έχουμε επανάληψη του αλγορίθμου σε τυχαία σημεία. Για αυτό δημιουργούμε ένα permutation vector

vector<int> permutation(n);

το οποίο περιέχει όλα τα indices των κόμβων. Για να εξασφαλίζουμε την τυχαιότητα εκτελούμε ανακάτεμα δεικτών στο permutation μέσω αλγορίθμου fisher-yates

[FisherYates\_shuffle.](https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle)

Ο λόγος που τον επιλέξαμε είναι επειδή κάθε ανακάτεμα που κάνει είναι εξίσου πιθανό.

# Filtered Vamana algorithm overview

Για την διευκόλυνσή μας και καθώς οι αλγόριθμοι Vamana εκτελούνται σε γράφους δημιουργήσαμε το τύπο

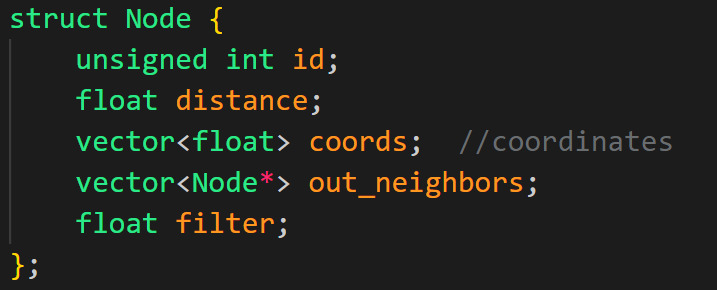
struct DirectedGraph{

unorderd\_map<Node\*, unordered\_set<Node\*>> adjacency\_list;

size\_t size() const{

Return adjacency\_list.size();  
}

};

o οποίος στηρίζεται πάνω στο struct Node 

η από το προηγούμενο κομμάτι της εργασίας με την προσθήκη όμως του μέλους filter. Αφού οι κόμβοι περιέχουν φίλτρα αυτό αναιρεί την μοναδική εύρεση medoid όπως στην Vamana. Συνεπώς βρίσκουμε ένα medoid, δηλαδή ένα αντιπροσωπευτικό κόμβο, για κάθε φίλτρο μέσω ανάλογης συνάρτησης findmedoid

unordered\_map<float, unsigned int> medoids = findmedoid(databasePoints, tau);

Για την κατασκευή του γράφου και αφού ανακατέψουμε τα δεδομένα μας ,όμοια με την Vamana, για κάθε κόμβο αναγνωρίζουμε το φίλτρο του και θέτουμε το medoid αυτού του φίλτρου ως σημείο εκκίνησης S\_Fx.

Η αποδοτικότητα αυτού του αλγορίθμου στηρίζεται ακριβώς πάνω στην χρήση των medoids καθώς έτσι περιορίζεται ο χώρος αναζήτησης, μειώνοντας την υπολογιστική επιβάρυνση για μεγάλα δεδομένα.

Επιπλέον μέσω της εντολής

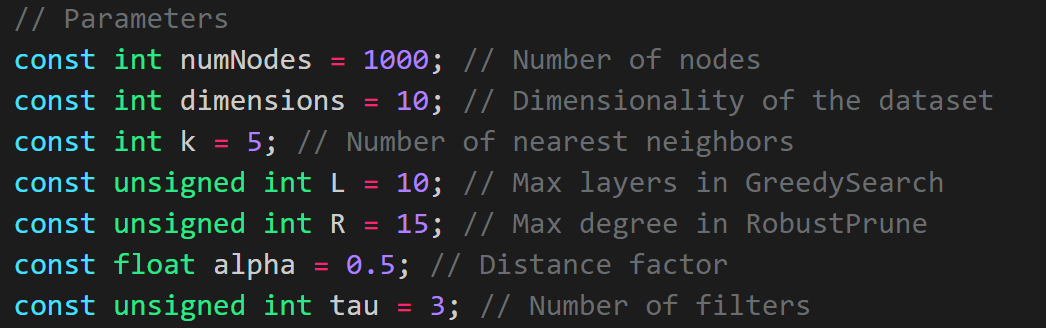
for(Node\* neighbor : G.adjacency\_list[point]){

G.adjacency\_list[neighbor].insert(point);

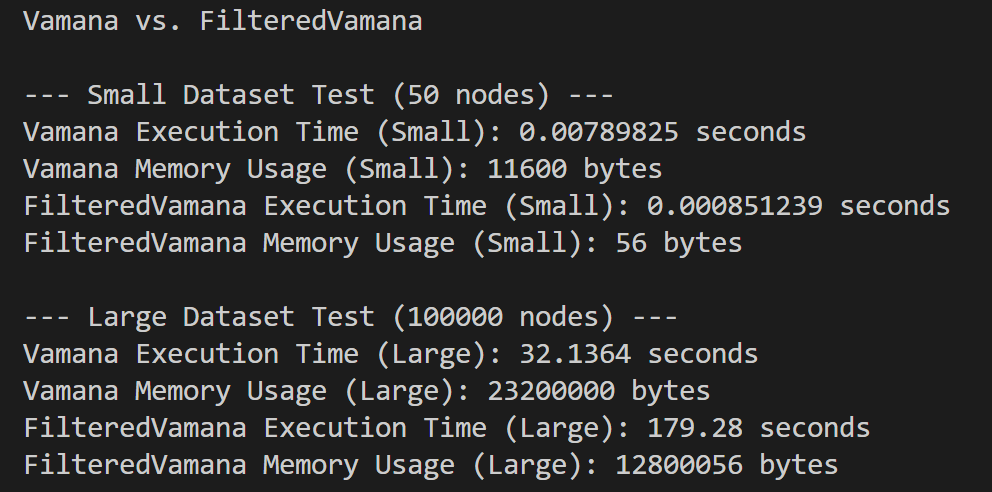
επιτυγχάνουμε τακτική ενημέρωση της λίστας γειτνίασης για την εξασφάλιση της συντήρησης συμμετρίας γειτόνων μεταξύ κόμβου και γειτόνων του.

# Συγκριση Vamana-Filtered Vamana

* Φίλτρα
* Η Filtered Vamana αξιοποιεί τα φίλτρων για τη μείωση του χώρου αναζήτησης, οδηγώντας σε ταχύτερη εκτέλεση για μεγάλα σύνολα δεδομένων.
* Η Vamana Indexing Algorithm εκτελείται σε όλο το σύνολο δεδομένων χωρίς φίλτρα, οδηγώντας σε πιο εξαντλητικές συγκρίσεις.
* Μedoid
* Και οι δύο χρησιμοποιούν medoids ως σημεία εκκίνησης, αλλά η Filtered Vamana προσδιορίζει medoids για κάθε φίλτρο, καθιστώντας τον αλγόριθμο πιο εστιασμένο σε δεδομένα με ομαδοποιήσεις.
* Αναπαράσταση γραφήματος:
* Η Vamana χρησιμοποιεί μια λίστα γειτνίασης για κάθε κόμβο, ενώ η FilteredVamana δημιουργεί ένα κατευθυνόμενο γράφο.
* Χρήση:
* Χρήση της Vamana Indexing Algorithm για μικρότερα σύνολα δεδομένων ή σύνολα χωρίς φίλτρα.
* Χρήση της Filtered Vamana για μεγάλα, σύνολα δεδομένων με φίλτρα

Για τη σύγκριση των δύο αυτών αλγορίθμων (Vamana Indexing Algorithm και Filtered Vamana) δημιουργήσαμε αρχείο vamanaVSfiltered.cpp το οποίο για παραμέτρους

τρέχει τους δύο αλγορίθμους και υπολογίζει πόσο χρόνο απαιτεί ο καθένας και πόση μνήμη καταναλώνει για case1: 50 Nodes και case2:100.000 nodes.

Αποτελέσματα:

Παρατηρούμε ότι για μεγάλα σύνολα δεδομένων (100.000 κόμβοι), η Filtered Vamana διαρκεί σημαντικά περισσότερο από τη Vamana. Αυτό υποδηλώνει ότι τα επιπλέον βήματα στη FilteredVamana, δηλαδή το φιλτράρισμα και η διατήρηση της δομής κατευθυνόμενου γράφου, αντισταθμίζουν τα οφέλη του φιλτραρίσματος. Αυτό αναμένεται σε κάποιο βαθμό, αλλά η απόκλιση (179,28 δευτερόλεπτα έναντι 32,14 δευτερολέπτων) υποδηλώνει ότι η γενική υλοποίηση του αλγορίθμου είναι αναποτελεσματική.

Αναφορικά με την μνήμη, η χρήση μνήμης της Vamana φαίνεται συνεπής με την εκτιμώμενη. Η χρήση μνήμης περιλαμβάνει το διάνυσμα coords, out\_neighbors διάνυσμα και τυχόν πρόσθετα μεταδεδομένα για κάθε κόμβο. Συγκεκριμένα 11600 bytes για 50 κόμβους φαίνεται να είναι σωστό, υποθέτοντας ότι ο κάθε κόμβος καταλαμβάνει 200−250 byte.

Για την Filtered Vamana τωρα, για μικρά σύνολα δεδομένων, η χρήση μνήμης που αναφέρεται (56 byte για 50 κόμβους) φαίνεται πολύ χαμηλή. Αυτό μπορεί να υποδηλώνει ότι είτε η λίστα γειτνίασης είναι ως επί το πλείστον κενή.

Ενώ για μεγάλα σύνολα δεδομένων, η χρήση μνήμης (12,8 MB) είναι χαμηλότερη από τη Vamana (23,2 MB), το οποίο είναι μη αναμενόμενο. Η κατευθυνόμενη δομή γράφου απαιτεί περισσότερη μνήμη από τη λίστα απλών out\_neighbors στη Vamana, ειδικά με τα πρόσθετα μεταδεδομένα φίλτρων.

# Συγκριση Vamana-StitchedVamana

* Φίλτρα
* Η Vamana δεν έχει φίλτρα
* Η Stitched οργανώνει τους κόμβους με κοινά φίλτρα, τους επεξεργάζεται ανεξάρτητα και στη συνέχεια ενώνει τους γράφους
* Medoid
* Η Vamana βρίσκει μια medoid
* Η Stitched χειρίζεται φιλτραρισμένα υποσύνολα, επομένως τα medoids αντιπροσωπεύουν κάθε ξεχωριστό φίλτρο
* Κλάδεμα
* Η Vamana εκτελεί την RobustPrune μετά από εισαγωγή κόμβων στους γείτονες ενός κόμβου
* Η Stitched εκτελεί την Filtered Robust Prune κατά τη διάρκεια της ένωσης των υπογράφων
* Προσθήκη αντίστροφων ακμών
* Η Vamana προσθέτει αντίστροφες ακμές κατά την κατασκευή του γραφήματος
* Η Stitched προσθέτει αντίστροφες ακμές κατά τη διάρκεια του Filtered Robust Prune
* Χρόνος
* Η Vamana έχει υψηλότερη χρονική πολυπλοκότητα, καθώς εκτελείται απευθείας στο πλήρες σύνολο δεδομένων. Επαναλαμβάνεται σε όλους τους κόμβους, εκτελεί το GreedySearch και εκτελεί κλάδεμα σε ένα μόνο γράφημα
* Η Stitched έχει χαμηλότερη χρονική πολυπλοκότητα για υποσύνολα, καθώς οι μικρότερες φιλτραρισμένες ομάδες οδηγούν σε λιγότερες συγκρίσεις και ταχύτερη αναζήτηση
* Μνήμη
* Η χρήση μνήμης της Vamana αυξάνεται με το μέγεθος των δεδομένων και τον αριθμό των γειτόνων
* Η κατανάλωση μνήμης της Stitched διανέμεται καλύτερα, καθώς οι μικρότερους γράφοι κατασκευάζονται ανεξάρτητα ο ένας από τον άλλον.
* Επεκτασιμότητα
* Η Vamana αποδίδει καλά για μικρά σύνολα δεδομένων ή για εκείνα χωρίς φίλτρα.
* Η Stitched είναι κατάλληλη για σύνολα δεδομένων με φίλτρα ή γενικότερα κάποιο είδος διαμερισμού.

# 

# Σύγκριση FilteredVamana –StitchedVamana

* Κλάδεμα
* Η FilteredVamana εκτελεί το κλάδεμα κατά την εισαγωγή κόμβων στους γείτονες ενός κόμβου
* Η Stitched εκτελεί την Filtered Robust Prune κατά τη διάρκεια της ένωσης των υπογράφων
* Αποδοτικότητα
* Η FilteredVamana επιτυγχάνει μέγιστη βελτιστοποίηση
* Η Stitched έχει υψηλότερο γενικά κόστος λόγω επαναλαμβανόμενων βημάτων

* Επεκτασιμότητα
* Η FilteredVamana έχει καλύτερη επεκτασιμότητα λόγω medoid-filtering
* Η Stitched είναι κατάλληλη για σύνολα δεδομένων με φίλτρα ή γενικότερα κάποιο είδος διαμερισμού, αλλά δεν χειρίζεται καλά δεδομένα με πολλά διαφορετικά φίλτρα.

# Find Medoid for filters

Η συνάρτηση findmedoid υλοποιεί έναν αλγόριθμο για την επιλογή αντιπροσωπευτικών κόμβων (medoids) από ένα σύνολο δεδομένων, ταξινομημένων με βάση φίλτρα. Κάθε φίλτρο (*filter*) αντιστοιχεί σε μια ομάδα σημείων, και ο στόχος είναι να επιλεγεί ένα σημείο από κάθε φίλτρο που θα λειτουργεί ως αντιπρόσωπος. Η επιλογή γίνεται λαμβάνοντας υπόψη έναν περιορισμό δείγματος (τ) και τη συχνότητα επιλογών κάθε σημείου.

### **Ανάλυση Λειτουργίας**

**Είσοδοι:**

* P: Ένα διάνυσμα από δείκτες προς κόμβους (*nodes*), κάθε κόμβος έχει ένα μοναδικό αναγνωριστικό (id) και ένα φίλτρο (filter).
* tau: Ο μέγιστος αριθμός δειγμάτων που μπορούν να επιλεγούν από κάθε φίλτρο.

**Έξοδοι:**

* M: Ένας unordered\_map που αντιστοιχεί κάθε φίλτρο στο medoid του.

### 

1. **Αρχικοποίηση**:
   * Δημιουργούνται:
     + Ο χάρτης M για την αποθήκευση των medoids.
     + Ο χάρτης T για την καταμέτρηση του πλήθους επιλογών κάθε κόμβου.
     + Το σύνολο all\_filters που περιέχει όλα τα μοναδικά φίλτρα από το σύνολο κόμβων P.
2. **Ομαδοποίηση κατά Φίλτρα**:
   * Για κάθε φίλτρο f:
     + Συλλέγονται τα id όλων των κόμβων που ανήκουν στο φίλτρο f και αποθηκεύονται στο διάνυσμα P\_f.
     + Αν δεν υπάρχουν κόμβοι για το φίλτρο, εμφανίζεται μήνυμα προειδοποίησης.
3. **Δειγματοληψία**:
   * Αν το πλήθος των κόμβων|P\_f| είναι μικρότερο ή ίσο με τ, επιλέγονται όλοι οι κόμβοι.
   * Αν είναι μεγαλύτερο από τ\tauτ, εφαρμόζεται τυχαία επιλογή tau κόμβων χρησιμοποιώντας τον αλγόριθμο shuffle.
4. **Επιλογή Medoid**:
   * Από τα τυχαία επιλεγμένα δείγματα R\_f, επιλέγεται ο κόμβος με τη μικρότερη συχνότητα εμφάνισης στο T (δηλ., ο κόμβος που έχει επιλεγεί λιγότερες φορές ως medoid στο παρελθόν).
   * Ο χάρτης M ενημερώνεται ώστε το medoid του φίλτρου f να είναι ο επιλεγμένος κόμβος.
   * Ο μετρητής του επιλεγμένου κόμβου στο T αυξάνεται κατά 1.
5. **Επιστροφή Αποτελεσμάτων**:
   * Επιστρέφεται ο χάρτης M, ο οποίος αντιστοιχεί κάθε φίλτρο στον αντιπροσωπευτικό του κόμβο (medoid).

### **Παρατηρήσεις**

1. Χρήση Τυχαιότητας: Η χρήση του shuffle και του γεννήτορα τυχαίων αριθμών (mt19937) εξασφαλίζει τυχαία επιλογή κόμβων, μειώνοντας την πιθανότητα μεροληψίας στην επιλογή.
2. Αποφυγή Σύγκρουσης Κόμβων: Με τη χρήση του T, η συνάρτηση επιδιώκει να εξισορροπήσει τη συχνότητα επιλογής των κόμβων, διασφαλίζοντας ότι κανένας κόμβος δεν επιλέγεται δυσανάλογα συχνά.
3. Αποδοτικότητα: Η πολυπλοκότητα επηρεάζεται από το πλήθος των φίλτρων και των κόμβων. Η χρήση δομών δεδομένων όπως unordered\_map και unordered\_set εξασφαλίζει γρήγορη πρόσβαση και αναζήτηση.

# Stitched Vamana overview

1. Η συνάρτηση Stitched Vamana ξεκινάει με τη συλλογή όλων των μοναδικών φίλτρων που υπάρχουν μεταξύ των κόμβων. Αυτά τα φίλτρα αποθηκεύονται σε ένα set, το οποίο διασφαλίζει ότι κάθε φίλτρο εμφανίζεται μόνο μία φορά.
2. Στη συνέχεια, οργανώνονται σε έναν map, όπου κάθε φίλτρο αποτελεί το κλειδί, και οι κόμβοι που έχουν το ίδιο φίλτρο αποθηκεύονται ως τιμές αυτού του χάρτη. Έτσι, κάθε φίλτρο αντιστοιχεί σε μία λίστα κόμβων που το μοιράζονται.
3. Αφού οργανωθούν οι κόμβοι, η συνάρτηση καλεί τη Vamana Indexing Algorithm για όλους τους κόμβους που έχουν κοινό φίλτρο, προκειμένου να δημιουργηθεί μια δομή αναζήτησης για κάθε ομάδα. Τέλος, για κάθε κόμβο καλείται η Filtered Robust Prune.

# 

# Threads in Stitched Vamana

Η συνάρτηση Stitched Vamana αναβαθμίστηκε με την προσθήκη πολυνηματικής εκτέλεσης (threads), με σκοπό την ταχύτερη εκτέλεση της διαδικασίας. Δημιουργούμε ένα νήμα για κάθε μοναδικό φίλτρο που εντοπίζεται στους κόμβους και το κάθε νήμα εκτελεί την VamanaIndexingAlgorithm για την ομάδα κόμβων που μοιράζονται το ίδιο φίλτρο. Με αυτόν τον τρόπο, οι κόμβοι με διαφορετικά φίλτρα επεξεργάζονται ανεξάρτητα. Επίσης δημιουργούμε νήματα ίσα με τον αριθμό των κόμβων και το κάθε νήμα εκτελεί τη διαδικασία Filtered Robust Prune για έναν συγκεκριμένο κόμβο. Με αυτές τις παραλληλοποιήσεις βελτιώνετε η συνολική απόδοση της Stitched Vamana. Τέλος, για να αποφευχθούν συγκρούσεις δεδομένων ή προβλήματα ασυνεπών καταστάσεων, κάθε ομάδα νημάτων συγχρονίζεται χρησιμοποιώντας τη μέθοδο join(). Με αυτόν τον τρόπο, διασφαλίζεται ότι όλα τα νήματα μιας ομάδας ολοκληρώνονται πριν προχωρήσει η εκτέλεση στο επόμενο στάδιο. Με αυτή την βελτίωση παρατηρήσαμε μείωση 45% στον χρόνο.

# Threads in main function

Επίσης αναβαθμίσαμε και την main συνάρτηση με την προσθήκη πολυνηματικής εκτέλεσης (threads). Στο σημείο που καλούμε την Greedy Search ή την Filtered Greedy Search, ανάλογα με τον τύπο του query, βάλαμε threads, όπου στο κάθε thread δίνουμε έναν αριθμό από queries για να βρουν τα k πιο κοντινά nodes. Με αυτή την αλλαγή παρατηρήσαμε μείωση 70% του χρόνου και αύξηση του recall κατά 50%.

# Threads for distance calculations

Η παραλληλοποιημένη έκδοση του αλγορίθμου Greedy Search και Robust Prune βασίζεται στη χρήση πολλαπλών threads για την επιτάχυνση του υπολογισμού αποστάσεων μεταξύ των κόμβων. Παρακάτω περιγράφονται οι βασικές διαφοροποιήσεις και λειτουργίες αυτής της υλοποίησης:

### **1. Υπολογισμός Αποστάσεων με Threads**

Στην παραλληλοποιημένη εκδοχή, ο υπολογισμός των αποστάσεων πραγματοποιείται από τη συνάρτηση calculate\_distances\_parallel, η οποία κατανέμει το σύνολο των κόμβων nodes σε τμήματα (chunks), καθένα από τα οποία επεξεργάζεται ένα ξεχωριστό thread.

* **Χρήση std::thread**: Δημιουργούμε ένα σύνολο από threads, ανάλογα με τον αριθμό των διαθέσιμων πυρήνων (thread::hardware\_concurrency()), για την κατανομή του φορτίου.
* **Ασφαλής Ενημέρωση Δομών Δεδομένων**: Οι αποστάσεις αποθηκεύονται σε έναν unordered\_map<Node\*, double> distances, με την πρόσβαση σε αυτόν να προστατεύεται μέσω ενός std::mutex για αποφυγή προβλημάτων συγχρονισμού.
* **Κατανομή Εργασίας**: Οι κόμβοι κατανέμονται ισομερώς σε κάθε thread, ενώ ο τελευταίος αναλαμβάνει και τυχόν υπόλοιπους κόμβους.

### **2. Πειραματικά Αποτελέσματα**

Η παραλληλοποίηση δεν επέφερε σημαντική βελτίωση στο χρόνο εκτέλεσης του υπολογισμού αποστάσεων. Καθώς ο χρόνος αναζήτησης τόσο χωρίς threads, όσο και με threads είναι 1.5 sec.

| **Υλοποίηση** | **Χρόνος Εκτέλεσης** |
| --- | --- |
| Χωρίς Threads | 1.5s |
| Με Threads | 1.5s |

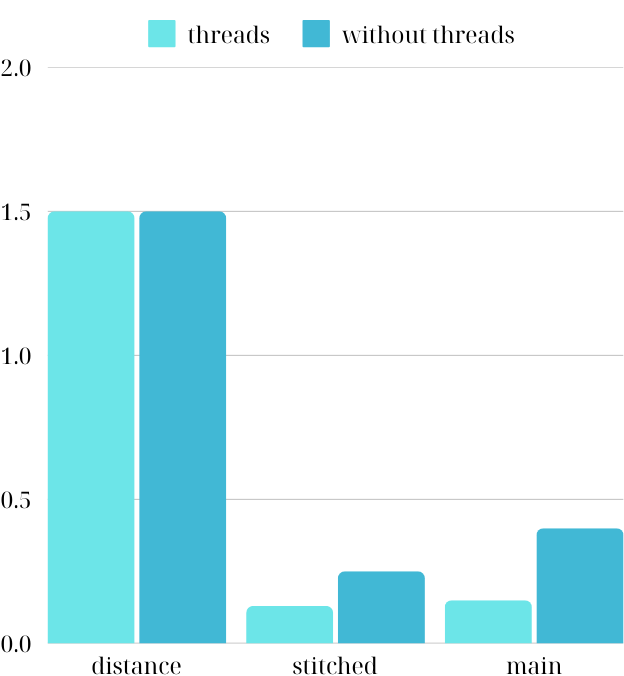
### **3. Πλεονεκτήματα Παραλληλοποίησης**

* **Εκμετάλλευση Πολλαπλών Πυρήνων**: Ο υπολογισμός των αποστάσεων γίνεται ταυτόχρονα, μειώνοντας το χρόνο για μεγάλα datasets.
* **Αποδοτικότερη Επεξεργασία**: Η κατανομή των κόμβων ανά thread διατηρεί την απόδοση κοντά στο θεωρητικό όριο επιτάχυνσης.

### **4. Συμπέρασμα**

Η παραλληλοποίηση για την υλοποίηση των υπολογισμών αποστάσεων στον αλγόριθμο της Greedy Search και του Robust Prune δεν βελτίωσε την απόδοση του αλγορίθμου.

# Ιστόγραμμα χρόνου εκτέλεσης με threads



# Αρχικοποίηση γράφων με τυχαίες ακμές

## Βελτιστοποίηση FilteredVamana

**Βελτίωση πυκνότητας**

* Πριν: Oι αραιοί γράφοι με πιθανούς απο συνδεδεμένους κόμβους ήταν ένα πρόβλημα, ειδικά όταν τα medoids δεν εντοπίζοντας σωστά ή όταν το FilteredGreedySearch αποτύγχανε να βρει γείτονες.
* Μετά: Η τυχαία αρχικοποίηση κόμβων διασφαλίζει ότι κάθε κόμβος έχει τουλάχιστον κάποια συνδεσιμότητα στο γράφο, αποτρέποντας εντελώς απομονωμένους κόμβους. Αυτό αυξάνει την πυκνότητα του γραφήματος και εξασφαλίζει καλύτερη χρησιμότητα .

**Ενισχυμένη στιβαρότητα**

* Πριν: Η διαδικασία δημιουργίας γράφου εξαρτιόταν σε μεγάλο βαθμό από την ορθότητα της επιλογής medoid (findmedoid) και την ακρίβεια του FilteredGreedySearch. Εάν κάποιο από τα δύο αποτύχει επηρεάζεται ολόκληρος ο γράφος
* Μετά: Οι τυχαίες άκρες λειτουργούν ως δίχτυ ασφαλείας. Ακόμα κι αν η επιλογή medoid ή το FilteredGreedySearch δεν αποδίδει βέλτιστα, το γράφημα θα εξακολουθεί να έχει ένα βασικό επίπεδο συνδεσιμότητας

**Ταχύτερη αρχικοποίηση γραφήματος**

* Πριν: Η δομή του γράφου κατασκευαζόταν εξ ολοκλήρου από το μηδέν, απαιτώντας πλήρη εκτέλεση αλγορίθμων αναζήτησης και κλαδέματος για κάθε κόμβο. Αυτό ήταν υπολογιστικά ακριβό, ειδικά για μεγάλα σύνολα δεδομένων.
* Μετά: Η τυχαία προετοιμασία άκρων επιταχύνει τη διαδικασία , μειώνοντας τον φόρτο εργασίας των επόμενων λειτουργιών όπως της FilteredGreedySearch και του Filtered Robust Prune.

## Βελτιστοποίηση StichedVamana

**Συνδεσιμότητα μεταξύ φίλτρων**

* Πριν: Οι κόμβοι που άνηκαν σε διαφορετικά φίλτρα ήταν εντελώς απομονωμένοι. Ο αλγόριθμος βασίστηκε αποκλειστικά στα FilteredRobustPrune και VamanaIndexingAlgorithm για μεμονωμένα φίλτρα για την κατασκευή του γραφήματος.
* Μετά: Οι τυχαίες συνδέσεις μεταξύ φίλτρων παρέχουν ένα βασικό επίπεδο συνδεσιμότητας. Αυτό βοηθά στην αποτροπή απομονωμένων υπογράφων. Επιτρέπει την αποτελεσματικότερη πλοήγηση μεταξύ φίλτρων κατά τη διάρκεια αναζητήσεων.

**Αυξημένη ευρωστία**

* Πριν: Τα προβλήματα συνδεσιμότητας ήταν συνηθισμένο σε περίπτωση που ο αλγόριθμος VamanaIndexingAlgorithm αποτύγχανε να συνδέσει αποτελεσματικά κόμβους μέσα σε μια ομάδα φίλτρων, καθώς δεν υπήρχε εναλλακτικός μηχανισμός για συνδέσεις μεταξύ φίλτρων.
* Μετά: Η προσθήκη τυχαίων συνδέσεων μεταξύ φίλτρων χρησιμεύει ώστε ακόμα κι αν ένα φίλτρο δεν είναι καλά συνδεδεμένο , οι άκρες του φίλτρου να μπορούν να αντισταθμίσουν και να διατηρήσουν την ευρωστία του γράφου.

**Βελτιωμένη πυκνότητα γράφου**

* Πριν: Ο γράφος ήταν αραιός και αποσυνδεδεμένος μεταξύ φίλτρων
* Μετά: Οι τυχαίες ακμές εξασφαλίζουν ένα ελάχιστο επίπεδο πυκνότητας, καθιστώντας τη διέλευση των κόμβων πιο αποτελεσματική και τις λειτουργίες γράφου ταχύτερες.

**Απλοποιημένο φιλτράρισμα**

* Πριν: Όλοι οι γείτονες, ανεξάρτητα από τα φίλτρα τους, διατηρήθηκαν μετά το κλάδεμα, δημιουργώντας ενδεχομένως ασυνέπειες στους περιορισμούς που βασίζονται σε φίλτρα.
* Μετά: Οι γείτονες με διαφορετικά φίλτρα αφαιρούνται μετά το κλάδεμα, διασφαλίζοντας ότι το γράφημα τηρεί αυστηρά τους περιορισμούς που βασίζονται σε φίλτρα

.

**Οφέλη τυχαιοποίησης**

Η εισαγωγή τυχαιότητας στην επεξεργασία φίλτρων και την προσθήκη άκρων

επιτυγχάνει την εξισορρόπηση φορτίου μεταξύ φίλτρων

# Αρχικοποίηση medoid στον Vamana με τυχαία σημεία

**1. Μειωμένο υπολογιστικό κόστος**

* Πριν: Το medoid υπολογίστηκε χρησιμοποιώντας όλους τους κόμβους στο σύνολο δεδομένων. Για μεγάλα σύνολα δεδομένων με n κόμβους, αυτό απαιτεί O(n^2 )
* Μετά: Η εισαγωγή της περίπτωσης 0 και της περίπτωσης 1 μειώνει σημαντικά το κόστος: Η περίπτωση 0 αποφεύγει εντελώς τον υπολογισμό του medoid, απαιτώντας μόνο O(1) για την επιλογή ενός τυχαίου κόμβου. Η περίπτωση 1 περιορίζει τους υπολογισμούς απόστασης κατά ζεύγη στο μέγεθος του υποσυνόλου O(subsetSize^2 )), προσφέροντας ένα είδος συμβιβασμού μεταξύ ακρίβειας και απόδοσης.

**2. Βελτιωμένη ευελιξία**

Η παράμετρος medoidCase επιτρέπει στο χρήστη να προσαρμόσει το πρόγραμμα

με βάση το μέγεθος δεδομένων. Πχ για μεγάλα σύνολα δεδομένων, η περίπτωση

0 ή η περίπτωση 1 μπορεί να επιταχύνει την διαδικασία. Ενώ εάν επιθυμούμε

ακρίβεια η περίπτωση 2 παρέχει την πιο ακριβή επιλογή medoid αλλά με

υψηλότερο υπολογιστικό κόστος. Η περίπτωση 1 χρησιμεύει ως μέση λύση

.Τέλος εάν αποζητάμε μικρή χρήση πόρων η περίπτωση 0 είναι η ταχύτερη και

λιγότερο απαιτητική σε πόρους.

**3. Επεκτασιμότητα**

Η νέα έκδοση είναι πιο κατάλληλη για σύνολα δεδομένων μεγάλης έκτασης. Η

δυνατότητα χρήσης επιλογής medoid βάσει υποσυνόλου ή με τυχαιότητα

διασφαλίζει ότι ο αλγόριθμος παραμένει υπολογιστικά εφικτός καθώς το

σύνολο δεδομένων μεγαλώνει.

**4. Ευρωστία αλγορίθμου**

Με την εισαγωγή πολλαπλών στρατηγικών επιλογής medoid, ο αλγόριθμος

γίνεται πιο ευέλικτος:

**-Περίπτωση 0:** Αποτελεσματική όταν το σύνολο δεδομένων έχει ομοιόμορφη

κατανομή και χωρίς ομαδοποίηση.

**-Περίπτωση 1:** Κατάλληλο για δεδομένα όπου ένα υποσύνολο μπορεί να είναι

αντιπροσωπευτικό, πχ σε φίλτρα .

**-Περίπτωση 2:** Εξασφαλίζει βέλτιστα αποτελέσματα για δεδομένα που απαιτούν

ακριβή υπολογισμό medoid.