ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Τελική Αναφορά Project

"Βελτιστοποιημένη Υλοποίηση Αλγορίθμων Vamana για Αναζήτηση και Εύρεση κ-Κοντινότερων Γειτόνων"

Στοιχεία Φοιτητών:

Παπαδήμα Θεοδοσία 1115202000162

Κορύλλος Γιώργος 1115202100069

Τσαντήλα Βασιλική 1115201800199

Δεκέμβριος 2024

Πίνακας Περιεχομένων

Σύνοψη - Abstract

Η παρούσα εργασία επικεντρώνεται στην διαδικασία εύρεσης κ-κοντινότερων γειτόνων, μέσω των αλγορίθμων FilteredVamana και StitchedVamana, οι οποίοι υλοποιήθηκαν και αξιολογήθηκαν.

Βασικές βελτιώσεις περιλαμβάνουν παραλληλοποίηση μέσω OpenMP και pthreads, καθώς και απλοποιήσεις για τη μείωση της υπολογιστικής πολυπλοκότητας. Τα πειραματικά αποτελέσματα έδειξαν ότι η παραλληλοποίηση στον StitchedVamana μειώνει τον χρόνο εκτέλεσης κατά 35% σε σχέση με τη σειριακή υλοποίηση, ενώ η χρήση εξειδικευμένων δομών δεδομένων βελτιώνει σημαντικά την απόδοση στις διαδικασίες αναζήτησης. Αντίθετα, η παραλληλοποίηση του FilteredVamana δεν οδήγησε σε ουσιαστική βελτίωση, υποδεικνύοντας ότι η αρχιτεκτονική του περιορίζεται από τον συγχρονισμό.

Η εφαρμογή παρουσιάζει εξαιρετική απόδοση σε πειραματικά datasets, με ακρίβεια που ξεπερνά το 97% για unfiltered ερωτήματα. Η μελέτη αυτή παρέχει κατευθύνσεις για περαιτέρω βελτιώσεις σε παρόμοιες δομές γραφημάτων και παραλληλοποιημένων αλγορίθμων.

Εισαγωγή

Η εύρεση των k-κοντινότερων γειτόνων (k-NN) αποτελεί θεμελιώδες πρόβλημα για πολλές εφαρμογές ανάλυσης δεδομένων. Στόχος της παρούσας εργασίας είναι η βελτιστοποίηση της διαδικασίας αυτής μέσω της υλοποίησης των αλγορίθμων filteredVamana και stitchedVamana, οι οποίοι συνδυάζουν αποδοτικές τεχνικές κατασκευής και βελτιστοποίησης γράφων.

Οι αλγόριθμοι αυτοί χρησιμοποιούν διαδικασίες όπως το "κλάδεμα" ακμών (pruning) και την αναζήτηση γειτόνων (greedy search) για να κατασκευάσουν κατευθυνόμενους γράφους περιορισμένου βαθμού (out-degree). Επιπλέον, ο αλγόριθμος FindMedoid εξασφαλίζει τη δίκαιη επιλογή κεντρικών σημείων (medoids), αξιοποιώντας τυχαία δείγματα από τα δεδομένα.

Βασικές παράμετροι όπως ο αριθμός των γειτόνων (R), η ακρίβεια αναζήτησης (α), και το μέγιστο πλήθος ακμών (L) καθορίζουν τη συμπεριφορά των αλγορίθμων και την αποτελεσματικότητα των αποτελεσμάτων τους.

Για την υποστήριξη των παραπάνω, αναπτύχθηκε μια δομή γράφου, στην οποία οι κόμβοι αντιπροσωπεύουν σημεία στον d-διάστατο χώρο με χαρακτηριστικά (π.χ., συντεταγμένες, ετικέτα), ενώ οι ακμές αντιπροσωπεύουν τη γειτνίαση μεταξύ δεδομένων. Η κατασκευή του γράφου γίνεται με στόχο τη βελτιστοποίηση της αναζήτησης γειτόνων. Σημαντικό είναι, επίσης, να σημειωθεί ότι οι δοθείσαντες αλγόριθμοι αναφέρονται σε σημεία τα οποία χαρακτηρίζει ένα σύνολο ετικετών, ενώ στην παρούσα εφαρμογή θεωρήθηκε ότι το κάθε σημείο έχει μία μοναδική ετικέτα.

Στα πλαίσια της εργασίας, η εφαρμογή έχει υλοποιηθεί και με παραλληλία και όλα της τα μέρη έχουν δοκιμαστεί με διάφορες παραμέτρους και σε διαφορετικά συστήματα. Τα αποτελέσματα και ο σχολιασμός των πειραμάτων αυτών, παρουσιάζονται παρακάτω.

Βασικά Στοιχεία

Η συνάρτηση **GreedySearch** πρόκειται για την υλοποίηση ενός άπληστου αλγορίθμου αναζήτησης. Με δεδομένο ένα διάνυσμα $\square_{\square}$ η αναζήτηση ξεκινά από ένα ορισμένο σημείο s (starting point), και σταδιακά διασχίζει το γράφο όλο και πιο κοντά στο $\square_{\square}$. Με το πέρας της συνάρτησης καταφέρνουμε να βρούμε τους (προσεγγιστικά) εγγύτερους γείτονες για κάθε ερώτηση $\square_{\square}$.

Η συνάρτηση **FilteredGreedySearch** αποτελεί εμπλουτισμένη έκδοση της GreedySearch, με τη νέα απαίτηση ότι κάθε σημείο της εξόδου θέλουμε να μοιράζεται τουλάχιστον μία ετικέτα με τις ετικέτες του ερωτήματος $\square_{\square}$. Δεδομένου ότι στη δική μας εφαρμογή, κάθε σημείο έχει μοναδική ετικέτα, η νέα απαίτηση της FilteredGreedySearch ανάγεται στον εντοπισμό των k εγγύτερων γειτόνων που έχουν την ίδια ετικέτα με το ερώτημα $\square_{\square}$.

Ο **RobustPrune** ξεκινά με έναν κόμβο p και ένα σύνολο υποψήφιων γειτόνων V. Σταδιακά, προσθέτει τους εγγύτερους γείτονες στον κόμβο p, με βάση την ευκλείδεια απόσταση, μέχρι ο βαθμός του κόμβου να φτάσει το R. Επιπλέον, φιλτράρει κόμβους που βρίσκονται μακριά, χρησιμοποιώντας μια αναλογία απόστασης που καθορίζεται από την παράμετρο a. Έτσι, "κλαδεύει" τους κόμβους και παράγει έναν κατευθυνόμενο γράφο περιορισμένου βαθμού εξερχόμενων γειτόνων R.

Ο **FilteredRobustPrune** εκτός από την ευκλείδεια απόσταση και την παράμετρο a, λαμβάνεται υπόψη και η "συμβατότητα" των ετικετών των κόμβων. Κατά τη διάρκεια της επιλογής των γειτόνων, επιτρέπονται μόνο εκείνοι που έχουν την ίδια ετικέτα με τον κόμβο-στόχο. Οπότε, ο γράφος που προκύπτει είναι πιο εξειδικευμένος για δεδομένα με ετικέτες.

Η συνάρτηση **Vamana** ξεκινά από ένα αρχικό σημείο (start node) και προσθέτει κόμβους στον γράφο μέσω επαναλαμβανόμενων βημάτων GreedySearch και RobustPrune. Ως αποτέλεσμα, επιστρέφει έναν κατευθυνόμενο γράφο, του οποίου οι κόμβοι δείχνουν στους κ-κοντινότερους σε αυτούς κόμβους του γράφου.

Ο **FilteredVamana** κατασκευάζει έναν κατευθυνόμενο γράφο με περιορισμένο βαθμό εξερχόμενων γειτόνων (R) χρησιμοποιώντας φίλτρα (label-sets) για την αναζήτηση. Κάθε κόμβος συσχετίζεται με ένα σύνολο ετικετών (στην παρούσα εργασία με μία μόνο ετικέτα), και η διαδικασία ξεκινά με την τυχαία αναδιάταξη (random permutation) των κόμβων. Στη συνέχεια, για κάθε κόμβο:

1. Εντοπίζονται οι αρχικοί κόμβοι (start nodes) που αντιστοιχούν στις ετικέτες του (εδώ, *στην* ετικέτα του).
2. Εφαρμόζεται η διαδικασία FilteredGreedySearch για να βρεθούν υποψήφιοι γείτονες.
3. Οι γείτονες "κλαδεύονται" (pruned) μέσω της συνάρτησης FilteredRobustPrune για να εξασφαλιστεί ότι ο βαθμός εξερχόμενων γειτόνων δεν υπερβαίνει το R.

Ο **StitchedVamana** συνδυάζει υπογράφους, έναν για κάθε ετικέτα, καθένας από τους οποίους δημιουργείται χρησιμοποιώντας τη συνάρτηση Vamana για το σύνολο των κόμβων που ανήκουν στη συγκεκριμένη ετικέτα. Μετά τη δημιουργία όλων των υπογράφων, οι κόμβοι κλαδεύονται ώστε να εξασφαλιστεί ότι η τελικός βαθμός R δεν υπερβαίνει το □□□□□□□□□ .

Η συνάρτηση **FindMedoid** εξυπηρετεί την εύρεση των medoids για κάθε φίλτρο. Για κάθε ετικέτα:

1. Δειγματίζεται τυχαία ένα υποσύνολο κόμβων.
2. Ο κόμβος με τη μικρότερη τιμή σε έναν μετρητή χρήσης (counter T) επιλέγεται ως medoid.

Αυτός ο αλγόριθμος διασφαλίζει ότι τα medoids κατανέμονται δίκαια μεταξύ των φίλτρων.

Σχεδιαστικές Επιλογές και Παρατηρήσεις

Παρατίθενται οι σχεδιαστικές επιλογές, συνοδευόμενες από την αντίστοιχη δικαιολόγηση και εξήγηση.

Διευκρινίσεις

Για την παραλληλοποίηση της εφαρμογής, χρησιμοποιήθηκαν pthreads και OpenMP. Οι διαφορετικές εκδοχές των συναρτήσεων (σειριακές - παράλληλες) βρίσκονται σε διαφορετικά αρχεία με αντίστοιχη ονομασία (π.χ. filteredVamanaParallel).

Για όσες απλοποιήσεις έγιναν, βεβαιώθηκε ότι η βασισμένη στον αρχικό ψευδοκώδικα εκδοχή είναι λειτουργική και υπάρχει σε σχόλιο στο ίδιο αρχείο.

Για την ασφαλή παραλληλοποίηση, προστέθηκε στη δομή του κόμβου του γράφου ένα mutex, το οποίο θα κλειδώνει την λίστα γειτόνων του κόμβου όπου χρειάζεται, για να αποφευχθεί η ταυτόχρονη πρόσβαση.

Στο αρχείο utility βρίσκεται η findMedoid, δηλαδή η "κλασσική" μέθοδος έυρεσης medoid, η οποία καλείται στον Vamana, ενώ στο αρχείο FindMedoid, βρίσκεται η ομώνυμη συνάρτηση που περιγράφηκε παραπάνω σύμφωνα με τον ψευδοκώδικα, η οποία καλείται στον filteredVamana.

Αρχικοποίηση γράφων με τυχαίες ακμές

Στους αλγορίθμους FilteredVamana και StitchedVamana περιγράφεται διαδικασία με αρχικά κενό γράφο, στον οποίο σταδιακά προστίθενται κορυφές και ακμές. Το αποτέλεσμα αυτού είναι ένας γράφος, ο οποίος, στην ουσία, αποτελείται από ανεξάρτητους υπογράφους, λόγω της ύπαρξης μιας μόνο ετικέτας ανά κόμβο. Αυτό οδήγησε στην δοκιμή αρχικοποίησης των γράφων με τυχαίες ακμές μεταξύ κορυφών, ώστε να παρατηρηθεί αν στο τέλος υπάρχει διασύνδεση μεταξύ των υπογράφων που αναφέρθηκαν και αντίστοιχα ποια θα είναι η απόδοση στα unfiltered ερωτήματα.

Για τον σκοπό αυτό δημιουργήθηκαν οι συναρτήσεις generate_random_edges για τον filteredVamana και connect_subgraphs για τον stitchedVamana στο αρχείο generate_graph.cpp.

Η συνάρτηση generate_random_edges, δέχεται έναν γράφο με μη-συνδεδεμένους κόμβους και μια μεταβλητή maxEdgesPerNode και δημιουργεί μεταξύ των κόμβων τυχαίες διασυνδέσεις, το πολύ maxEdgesPerNode ανά κόμβο. Πράγματι, στον γράφο που δημιουργεί ο filteredVamana, με αυτή την αρχικοποίηση, υπάρχει διασύνδεση μεταξύ των υπογράφων με διαφορετικές ετικέτες, ακόμα και για maxEdgesPerNode = 1.

Η συνάρτηση connect_subgraphs, εγγυάται ότι όλα τα υπογραφήματα συνδέονται ως εξής:

- Δημιουργεί μια λίστα με τις ετικέτες των υπογράφων, την οποία "ανακατεύει" τυχαία.
- Για κάθε υπογράφο της αντίστοιχης ετικέτας της λίστας, συνδέει έναν τυχαία επιλεγμένο κόμβο από αυτόν (Gf[i]) σε έναν άλλο τυχαία επιλεγμένο κόμβο στο επόμενο υπογράφο της λίστας (Gf[i + 1]).

Έτσι, δημιουργείται μια "αλυσίδα" συνδέσεων και επαληθεύεται ότι όλοι οι υπογράφοι είναι συνδεδεμένοι με κάποιον άλλο υπογράφο. Η συνάρτηση καλείται στο τέλος του stitchedVamana και πράγματι, δημιουργεί τις ζητούμενες διασυνδέσεις.

Ωστόσο, δεν παρατηρήθηκε διαφορά στην απόδοση όσον αφορά τα unfiltered ερωτήματα, τόσο στον filteredVamana όσο και στον stitchedVamana, δεδομένου ότι, λόγω της παραδοχής που υλοποιήθηκε στην main (βλ. παρακάτω: Δυνατότητες της main), η απόδοση ήταν ήδη υψηλή (τουλάχιστον 97% για οποιοδήποτε input).

Αρχικοποίηση γράφων με τυχαίες ακμές σε κόμβους με την ίδια ετικέτα

Η υλοποίηση παρέχει και την εναλλακτική αρχικοποίησης του filteredVamana με γράφο, ο οποίος έχει κόμβους με το πολύ μία ακμή προς έναν τυχαίο κόμβο με την *ίδια ετικέτα* (generate_label_based_graph). Τα αποτελέσματα παραμένουν ίδια στην περίπτωση αυτή, για αυτό και δεν έχει συμπεριληφθεί στα πειράματα που ακολουθούν.

Αρχικοποίηση medoid στον Vamana με τυχαία σημεία

Ο αρχικός αλγόριθμος Vamana υπολόγιζε στην αρχή το medoid του συνόλου των σημείων, το οποίο θα είναι και το σημείο από το οποίο ξεκινούν όλα τα ερωτήματα. Δοκιμάστηκε η αντικατάσταση αυτού του βήματος με την επιλογή

ενός τυχαίου σημείου. Η απόδοση του χρόνου βελτιώθηκε κατά 2%, ενώ τα recalls παρέμειναν τα ίδια.

(Filtered)GreedySearch

Τοπική βελτιστοποίηση

Παρατηρούμε ότι όσο το σύνολο □ \ □ δεν είναι κενό, ο υπολογισμός της απόστασης των σημείων που περιέχει το σύνολο □ \ □ από το ερώτημα □□ επαναλαμβάνεται σε κάθε επανάληψη του while loop. Σκεφτόμαστε ότι ενδεχομένως να είναι πιο αποτελεσματικό να υπολογίζουμε αυτές τις αποστάσεις 1 φορά (as they show up), και όποτε τις ξανά χρειαστούμε να τις κάνουμε retrieve από τη δομή στην οποία τις έχουμε αποθηκεύσει (η δομή αυτή είναι η nodeMap).

Θέλουμε, επίσης, το min των υπολογισμένων αποστάσεων να το βρίσκουμε σε O(1), έτσι ορίζουμε τη δομή sortedSet.

Με τις παραπάνω δύο δομές, καταφέρνουμε επιπλέον η ανανέωση του συνόλου □ (to retain closest L points to □□), αλλά και η εύρεση των closest k points from □ να υπολογίζεται κατευθείαν από το sortedSet (αντί να κάναμε πρώτα sorting, και μετά retrieve).

Τα παραπάνω υλοποιήθηκαν τόσο στον GreedySearch όσο και στον FilteredGreedySearch.

Απλοποίηση FindMedoid

Σύμφωνα με τον δοθέντα ψευδοκώδικα, η FindMedoid χρησιμοποιεί έναν χάρτη-μετρητή T. Ο αρχικός σκοπός του χάρτη T ήταν να παρακολουθεί πόσες φορές κάθε κόμβος επιλέχθηκε ως medoid. Η απλοποιημένη συνάρτηση FindMedoid εξαλείφει τον T και επιλέγει απευθείας έναν κόμβο από το τυχαία ανακατεμένο σύνολο κόμβων ίδιας ετικέτας (f). Έτσι, μειώνεται η υπολογιστική επιβάρυνση.

FilteredVamana

Γενικές Παρατηρήσεις

Ο γράφος αρχικοποιείται με κόμβους από τις συντεταγμένες που δίνονται, αλλά χωρίς ακμές.

Απλοποίηση

Στην παρούσα υλοποίηση, όπως έχει ήδη διευκρινιστεί, το κάθε σημείο έχει μόνο μία ετικέτα. Επομένων ήταν δυνατόν να πραγματοποιηθούν και σε αυτόν τον αλγόριθμο κάποιες απλοποιήσεις σε σχέση με τον ψευδοκώδικα, ώστε να αποφευχθούν επιπλέον υπολογισμοί και να μειωθεί η πολυπλοκότητα και ο χρόνος εκτέλεσης. Εκτός αυτού, λόγω της συγκεκριμένης δομής, κάποιοι υπολογισμού ήταν περιττοί. Οι αλλαγές είναι οι εξής:

- Παραλείπεται ο υπολογισμός του Fx ως "label-set" και αρχικοποιείται απευθείας το F με την ετικέτα του τρέχοντος κόμβου σε κάθε επανάληψη.
- Το $□_{□□□(□)}$ (S_set) αρχικοποιείται με το start node που αντιστοιχεί στην ετικέτα του εκάστοτε κόμβου, χωρίς να διατρέχονται όλες τις ετικέτες ενός συνόλου Fx και να ανακτώνται τα αντίστοιχα start nodes, όπως θα γινόταν για πολλές ετικέτες.
- Αγνοείται το βήμα 7, δηλαδή η διατήρηση ενός συνόλου V. Το βήμα αυτό είναι περιττό, αφού το σύνολο δεν χρησιμοποιείται αργότερα κάπου.
- Δεν ελέγχεται ξανά η διπλοτυπία, αφού η συνάρτηση addEdge δεν θα προσθέσει έναν γείτονα που ήδη υπάρχει.
- Αποφεύγονται επιπλέον έλεγχοι για null κόμβους, που γίνονται ήδη στις συναρτήσεις που καλούνται στον filteredVamana.

Έτσι, επιτυγχάνεται μια περιεκτικότερη και αποδοτικότερη υλοποίηση, καθώς μειώνεται, επίσης, ο χρόνος.

Παραλληλοποίηση

Για την παραλληλοποίηση του filteredVamana (με pthreads), οι κόμβοι χωρίζονται σε τμήματα και κάθε νήμα αναλαμβάνει την διαδικασία του αλγορίθμου για ένα τμήμα. Οι διαμοιραζόμενοι πόροι προστατεύονται με mutex.

Η παραλληλοποίηση του αλγορίθμου filteredVamana επιτυγχάνεται με την κατανομή των κόμβων του γράφου σε τμήματα, κάθε ένα από τα οποία επεξεργάζεται ένα νήμα. Συνολικά χρησιμοποιούνται 10 νήματα (NUM_THREADS), ενώ η επικοινωνία και συγχρονισμός μεταξύ τους γίνεται μέσω ενός κοινόχρηστου mutex.

Μετά από δοκιμές (βλ. παρακάτω: Αποτελέσματα Εκτέλεσης>FilteredVamana), παρατηρήθηκε ότι από τα 10 νήματα και μετά, δεν υπήρχε διαφορά στον χρόνο, παρά μόνο αύξηση της μνήμης, γι' αυτό και επιλέχθηκαν τόσα νήματα.

Το mutex αυτό είναι απαραίτητο για την προστασία των διαμοιραζόμενων τμημάτων. Αν τα τμήματα αυτά δεν έκαναν χρήση των ίδιων συναρτήσεων (π.χ. η

addEdge χρησιμοποιείται και στον κώδικα του κάθε νήματος και στην συνάρτηση FilteredRobustPrune, η οποία καλείται σε δύο φάσεις στον κώδικα αυτό), θα μπορούσαν να είχαν οριστεί διαφορετικά mutex, ώστε όταν εκτελείται το ένα τμήμα να μην μπλοκάρει και το άλλο, εφόσον δεν θα επεξεργάζονταν τα ίδια δεδομένα. Όμως, αφού πράγματι επεξεργάζονται τα ίδια, το ένα mutex είναι αναγκαίο, ώστε να μπλοκάρονται όσα νήματα θέλουν να αποκτήσουν πρόσβαση στα ίδια τμήματα και να αποφευχθεί, έτσι, η επικάλυψη. Όμως, στην περίπτωση αυτή, λόγω της υποχρεωτικής αυτής αναμονής των νημάτων, ενδέχεται στο τέλος η εκτέλεση να είναι, στην ουσία, σειριακή.

Δοκιμάστηκε, επίσης, η παραλληλοποίηση της συνάρτησης initialize_graph, η οποία καλείται στην αρχή της filteredVamana, αλλά δεν παρατηρήθηκε βελτίωση. Η συνάρτηση υπάρχει στο αρχείο utility.cpp με το όνομα initialize_graph_parallel.

Για την παραλληλοποίηση της συγκεκριμένης διαδικασίας, υπάρχει και το αρχείο filteredVamanaParallelDistances, όπου συνδυάζεται η προαναφερθείσα παραλληλία με τον παράλληλο υπολογισμό των αποστάσεων (βλ. Σχεδιαστικές Επιλογές και Παρατηρήσεις>Υπολογισμός Αποστάσεων).

StitchedVamana

Γενικές Παρατηρήσεις

Όπου χρειάστηκε η χρήση map, χρησιμοποιήθηκε ο τύπος unordered_map, αφού:

1. δεν χρειάστηκε τα στοιχεία να είναι κατανεμημένα και
2. έχει την λιγότερη πολυπλοκότητα (O(1)).

Ο γράφος αρχικοποιείται με κόμβους από τις συντεταγμένες που δίνονται, αλλά χωρίς ακμές.

Απλοποίηση

Το βήμα 5, δηλαδή το κλάδεμα όλου του γράφου με περιορισμό $\square_{\square\square\square\square\square\square\square\square\square}$, παραλείπεται, επομένως και η παράμετρος $\square_{\square\square\square\square\square\square\square\square\square}$ δεν χρησιμοποιείται, αλλά διατηρείται για λόγους πληρότητας. Η επιλογή αυτή βασίζεται στο γεγονός ότι το κλάδεμα εκτελείται ήδη τοπικά σε κάθε υπογράφο Gf κατά τον Vamana. Δεδομένου ότι οι περιορισμοί ($\square_{\square\square\square\square\square\square}$) επιβάλλονται για κάθε υπογράφο και συνήθως $\square_{\square\square\square\square\square\square\square\square\square} \geq \square_{\square\square\square\square\square\square}$, το ολικό κλάδεμα στο τέλος δεν θα άλλαζε τον γράφο.

Παραλληλοποίηση

Στον stitchedVamana, γίνεται μια επανάληψη για κάθε ετικέτα στο σύνολο F. Για την παραλληλοποίησή του (με pthreads), η δουλειά της κάθε επανάληψης έχει δοθεί σε ένα νήμα. Οι συναρτήσεις graphUnion και store_medoid προστατεύονται από mutex, αφού κατά την εκτέλεση πολλά νήματα θα επιχειρήσουν ταυτόχρονη πρόσβαση σε αυτές. Για μεγαλύτερη απόδοση, προστατεύονται από διαφορετικά mutex, αφού στο περιεχόμενό τους, δεν επεξεργάζονται ίδια δεδομένα ούτε καλούν ίδιες συναρτήσεις. Με αυτόν τον τρόπο, όταν ένα νήμα καλεί την μία από αυτές, η άλλη δεν μπλοκάρεται και μπορεί να εκτελεστεί ταυτόχρονα.

Δοκιμάστηκε, επίσης, η παραλληλοποίηση της συνάρτησης generate_graph, η οποία καλείται στον Vamana (που αντίστοιχα, καλείται στον stitchedVamana), ωστόσο δεν παρατηρήθηκε βελτίωση. Η συνάρτηση υπάρχει στο αρχείο generate_graph.cpp με το όνομα generate_graph_parallel.

Υπολογισμός αποστάσεων

Ιδέα και παραλληλοποίηση

Παρατηρούμε ότι ο filteredVamana καλεί την FilteredGreedySearch για κάθε σημείο του dataset. Με τη σειρά της η greedy θα χρειαστεί να υπολογίσει τις αποστάσεις του συγκεκριμένου σημείου προς άλλα σημεία ίδιας ετικέτας με το εξετάζον σημείο.

Επειδή, στο dataset υπάρχουν παραπάνω από 1 σημεία με το ίδιο φίλτρο, συνειδητοποιούμε ότι κλήσεις της greedy για σημεία με ίδια ετικέτα, θα υπολογίσουν *ξανά* τις ίδιες αποστάσεις.

Αποφασίζουμε, λοιπόν, στην filteredVamana να αποθηκεύουμε σε μία δομή τις αποστάσεις κάθε σημείου προς κάθε άλλο σημείο που έχει την ίδια ετικέτα με αυτό. Εν συνέχεια, κάθε φορά που καλούμε την greedy της δίνουμε τη δομή αυτή.

Σκεφτόμαστε πως η συμπλήρωση αυτής της δομής θα μπορούσε να γίνει παράλληλα. Να αναλάβει, δηλαδή, κάθε thread τη συμπλήρωση ενός μέρους της δομής αυτής. Συνειδητοποιούμε, όμως, ότι στη συγκεκριμένη περίπτωση θα χρειαστεί να ορίσουμε ένα mutex, ώστε αυτός να γίνεται locked κάθε φορά που ένα thread, πάει να προσθέσει ένα νέο entry στη δομή μας.

Παρατηρούμε ότι η ύπαρξη του mutex, καθυστερεί σημαντικά την ολοκλήρωση της συμπλήρωσης της δομής (βλ. branch). Έτσι, αποφασίζουμε το κάθε thread να

συμπληρώνει μια δική του δομή, και όταν ολοκληρώσουν όλα τα threads, να ενώσουμε όλες τις δομές σε μία.

Η παραπάνω βελτιστοποίηση υλοποιήθηκε και στον stitchedVamana, συγκεκριμένα στον Vamana του 1ου παραδοτέου.

Δυνατότητες της main

Η main παρέχει δύο βασικές επιλογές:

1. Να εκτελεστούν από την αρχή οι αλγόριθμοι Vamana, οι οποίοι δημιουργούν τον γράφο και τον χάρτη των medoids, αποθηκεύοντάς τους σε ένα δυαδικό αρχείο.
2. Να υπολογιστούν τα queries χρησιμοποιώντας τον έτοιμο γράφο και τον χάρτη από τα αποθηκευμένα δυαδικά αρχεία, χωρίς να χρειαστεί νέα εκτέλεση των αλγορίθμων Vamana.

Για κάθε διαφορετικό συνδυασμό υλοποιήσεων υπάρχει και διαφορετική main συνάρτηση (π.χ., για τον filteredVamanaParallel με παραλληλοποίηση στον υπολογισμό των queries, έχουμε την parallel_filtered_main_parallel).

Όσον αφορά τα unfiltered queries, οι filteredVamana και stitchedVamana στη ουσία παράγουν έναν υπογράφο για κάθε φίλτρο. Αυτοί οι υπογράφοι δεν συνδέονται μεταξύ τους. Αν θέλουμε να υπολογίσουμε τους k-ANNS ενός unfiltered query, τότε καλούμε την filteredGreedy με S = {starting nodes για κάθε φίλτρο} και F = {κάθε φίλτρο}. Η greedySearch θα ξεκινήσει την αναζήτηση με το starting node που βρίσκεται πιο κοντά στο query. Η greedy θα αλλάξει υπογράφο μόνο όταν δεν υπάρχει γείτονας του starting node πιο κοντά στο query από τα άλλα starting nodes.

Αντιλαμβανόμαστε ότι το παραπάνω δίνει χαμηλά ποσοστά επιτυχίας στα unfiltered queries. Έτσι, ο τρόπος που το διορθώνουμε αυτό είναι: τρέχουμε την filteredGreedy με S = {points που προέκυψαν τρέχοντας τον greedy για k = 1 για κάθε φίλτρο με starting point το αντίστοιχο starting point του φίλτρου} και F = {κάθε φίλτρο}.

Ο παραπάνω τροποποιημένος υπολογισμός γίνεται είτε ο γράφος έχει φτιαχτεί από τον filteredVamana, είτε ο γράφος έχει φτιαχτεί από τον stitchedVamana.

Παραλληλοποίηση queries

Υπάρχουν 2 διαφορετικές κύριες εκδοχές των main (οι άλλες απλά χρησιμοποιούν άλλες υλοποιήσεις των Vamana), οι filtered_main/stitched_main και οι parallel_filtered_main/parallel_stitched_main. Η διαφορά ανάμεσα στις κανονικές και τις parallel είναι στα queries. Στις parallel έχει παραλληλοποιηθεί το searching των γειτόνων για τα queries με χρήση OpenMP. Λόγω αυτού του approach, η "μπάρα" που είχαμε ("Calculating Query xxxx/5012") δεν γίνεται να υπάρχει διότι θα είχε πολλαπλά threads να προσπαθούν να γράψουν στο terminal ταυτόχρονα, και η χρήση critical section δεν βοηθάει. Οι μετρήσεις που κάναμε για τον ολικό χρόνο που έπαιρνε ο υπολογισμός του FilteredGreedySearch για όλα/filtered/unfiltered queries δεν δουλέυει διότι προσθέταμε κάθε χρόνο ξεχωριστά, όμως εδώ καταλήγουμε με αποτελέσματα του τύπου όλο το πρόγραμμα να έτρεξε για 30 δευτερόλεπτα και ο υπολογισμός του FilteredGreedySearch για όλα τα queries να πήρε 250 δευτερόλεπτα.

Γι'αυτό τώρα μετράμε τον συνολικό χρόνο που παίρνει η διαδικασία εύρεση των γειτόνων για τα queries. Δυστυχώς δεν μπορούμε να κάνουμε αυτή την μέτρηση μεμονωμένα για filtered/unfiltered queries, όμως μπορούμε να έχουμε το average time ο FilteredGreedySearch πήρε για όλα/filtered/unfiltered queries. Παρατηρούμε ότι παρόλο που ο μέσος όρος είναι μεγαλύτερος (σχεδόν 110% χειρότερο χρόνο, οπότε περίπου x2.3 φορές μεγαλύτερο), αναμενόμενο λόγω των επικοινωνιών ανάμεσα στα threads, ο συνολικός χρόνος μειώνεται απίστευτα (4 φορές πιο γρηγορο σχεδόν, σε PC με 4 cores).

Αποτελέσματα Εκτέλεσης

Παρακάτω παρουσιάζονται οι δομές που έδωσαν τον καλύτερο χρόνο εκτέλεσης, μαζί με τον χρόνο αυτό και τη μνήμη που καταναλώθηκε. Τα πειράματα πραγματοποιήθηκαν σε τρία διαφορετικά συστήματα. Κάθε πείραμα εκτελέστηκε επαναλαμβανόμενα μέσω ενός script (testing.sh), το οποίο για κάθε επανάληψη παρήγαγε στατιστικά δεδομένα που παρουσιάζονται στους πίνακες. Τα αναλυτικά αποτελέσματα είναι διαθέσιμα στα αρχεία αποτελεσμάτων (results.txt).

Προδιαγραφές μηχανημάτων:

- PC 1: Linux με WSL. CPU: Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz 2.40 GHz.
- PC 2: Linux με WSL. CPU: Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
- PC 3: Native Linux. CPU: 11th Gen Intel® Core™ i7-1165G7 @ 2.80GHz × 8

Καλύτερες υλοποιήσεις χρονικά

FilteredVamana: Σειριακή υλοποίηση (filteredVamana)
Χρόνος (Average) Index για k = 100, L = 110, R = 96, a = 1.2, t = 55:
PC1: 165 sec (+14.72800% faster Average Index Time than Parallel and +101.54200% faster Average Index Time than Parallel Distances)
PC2: 124 sec (+9.49800% faster Average Index Time than Parallel and +89.43900% faster Average Index Time than Parallel Distances)
PC3: 46 sec (+11.54% faster Average Index Time than Parallel and +35.21% faster Average Index Time than Parallel Distances)
29.6875 MB (2.97 MB% less memory than parallel, and 1227.35 MB% less memory than parallel distances)

StitchedVamana: Παράλληλη υλοποίηση (stitchedVamanaParallel)
Χρόνος (Average) Index για k = 100, L = 110, R = 96, a = 1.2, □□□□□□□□□ = 98:
PC1: 130 sec (+35.17400% faster Average Index Time than Serial)
PC2: 94 sec (+37.00000% faster Average Index Time than Serial)
PC3: 34 sec (+39.29% faster Average Index Time than Serial, and + 12.82% faster Average Index Time than ParallelDistances)
172.336 MB (+25.0 MB% more memory than parallel, and 1143.244 MB% less memory than parallel distances)

Query Calculation (για filteredVamana): Παράλληλη υλοποίηση (στις συναρτήσεις parallel_filtered_main)
Χρόνος (Total Query Time) για k = 100, L = 110, R = 96, a = 1.2, t = 55:
PC1: 59.215 sec (65.58500% better)
PC2: 32 sec (73.62300% better)
PC3: 8 secs (85.519% better)

Query Calculation (για stitchedVamana): Παράλληλη υλοποίηση (στις συναρτήσεις parallel_stitched_main)
Χρόνος (Total Query Time) για k = 100, L = 110, R = 96, a = 1.2, □□□□□□□□□= 98:
PC1: 59.694 sec (63.01200% better)
PC2: 31 sec (73.55700% better)
PC3: 7 secs (85.262% better)

Κάποια αξιοσημείωτα αποτελέσματα

Τοπική βελτιστοποίηση (Filtered)GreedySearch

Τα συγκεκριμένα πειράματα εκτελέστηκαν στο PC3. Συγκρίνουμε το παραδοτέο 2 (στο οποίο δεν είχε γίνει αυτή η βελτιστοποίηση) με το παραδοτέο 3 (έχοντας βάλει μόνο αυτή την βελτιστοποίηση). Τρέχουμε τα προγράμματα με παραμέτρους: k = 20, L = 40, R = 80, a = 1.2, t = 2

filteredVamana		
	Παραδοτέο 2	**Παραδοτέο 3**
Χρόνος εκτέλεσης (secs)	40	24
Κατανάλωση μνήμης (MB)	22.5	22.5

stitchedVamana		
	Παραδοτέο 2	**Παραδοτέο 3**
Χρόνος εκτέλεσης (secs)	57	52
Κατανάλωση μνήμης (MB)	143.969	144.105

Υπολογισμός Αποστάσεων

Τα συγκεκριμένα πειράματα εκτελέστηκαν στο PC3.
Και οι δύο εκτελέσεις περιέχουν την τοπική βελτιστοποίηση στον (Filtered)GreedySearch. Η 1η εκτέλεση υπολογίζει τις αποστάσεις χωρίς threads, ενώ η 2η εκτέλεση με 10 threads (ή 4 threads, αντίστοιχα).
Τρέχουμε τα προγράμματα με παραμέτρους: k = 20, L = 40, R = 80, a = 1.2, t = 2

filteredVamana		
	No threads	**10 threads**
Χρόνος εκτέλεσης (secs)	71	34
Κατανάλωση μνήμης	620.25	1246.08

(MB)		

stitchedVamana		
	No threads	**4 threads**
Χρόνος εκτέλεσης (secs)	52	48
Κατανάλωση μνήμης (MB)	469.695	908.266

FilteredVamana

Οι παρακάτω μετρήσεις αφορούν την δημιουργία του index, δηλαδή την συνάρτηση filteredVamana, όχι όλο το πρόγραμμα. Οι μετρήσεις με τον τίτλο "πριν την παραλληλοποίηση" είναι αποτελέσματα εκτέλεσης του παραδοτέου 2. Τα συγκεκριμένα πειράματα για τον filteredVamana εκτελέσθηκαν στο PC 1.

Για: k = 20, L = 40, R = 80, a = 1.2, t = 2		
Πριν την παραλληλοποίηση		
Χρόνος εκτέλεσης (δευτερόλεπτα)		Κατανάλωση μνήμης (MB)
38.2		22.9
Με χρήση νημάτων για την διαδικασία του filteredVamana		
Αριθμός νημάτων (NUM_THREADS)	Χρόνος εκτέλεσης (δευτερόλεπτα)	Κατανάλωση μνήμης (MB)
1	35	23
2	42	23
4	39	23
8	46.5	24.9
10	39	23

20	39	26	
Με χρήση νημάτων για την διαδικασία του filteredVamana και τον υπολογισμό των αποστάσεων			
Αριθμός νημάτων (NUM_THREADS)	Αριθμός νημάτων αποστάσεων (THREADS_NO)	Χρόνος εκτέλεσης (δευτερόλεπτα)	Κατανάλωση μνήμης (MB)
1	10	70	1291
2	10	67.9	1292
4	10	69	1292
8	10	66	1293
10	10	66	1292
20	10	66	1292

Για: k = 100, L = 110, R = 96, a = 1.2, t = 55			
Πριν την παραλληλοποίηση			
Χρόνος εκτέλεσης (δευτερόλεπτα)		Κατανάλωση μνήμης (MB)	
154		29	
Με χρήση νημάτων για την διαδικασία του filteredVamana			
Αριθμός νημάτων (NUM_THREADS)	Χρόνος εκτέλεσης (δευτερόλεπτα)	Κατανάλωση μνήμης (MB)	Αριθμός νημάτων (NUM_THREADS)
1	179	29.6	1
2	172	29.7	2
4	171	30	4
8	173	31	8

10	171	31	10
Με χρήση νημάτων για την διαδικασία του filteredVamana και τον υπολογισμό των αποστάσεων			
Αριθμός νημάτων (NUM_THREADS)	Αριθμός νημάτων αποστάσεων (THREADS_NO)	Χρόνος εκτέλεσης (δευτερόλεπτα)	Κατανάλωση μνήμης (MB)
1	10	418	3854
2	10	418	3854
4	10	421	3855
8	10	416	3855
10	10	416	3855

Συμπεράσματα

Παρατηρείται ότι η απόδοση του filteredVamana τόσο στον χρόνο εκτέλεσης όσο και στην μνήμη, όχι μόνο δεν βελτιώνεται, αλλά επιβαρύνεται με τη χρήση νημάτων. Όσον αφορά τον χρόνο, έχοντας μόνο την παραλληλοποίηση σε τμήματα (chunks), ο χρόνος παραμένει ίδιος, γεγονός πολύ λογικό, αφού η συγκεκριμένη χρήση του mutex οδηγεί ουσιαστικά σε σειριακή υλοποίηση. Με την παραλληλοποίηση των αποστάσεων και την χρήση του FilteredGreedySearchIndex, ο χρόνος χειροτερεύει, δηλαδή αυξάνεται αρκετά, το ίδιο και η μνήμη.

Η απόδοση του stitchedVamana, ωστόσο, είναι σημαντικά βελτιωμένη, με τον χρόνο εκτέλεσης να έχει μειωθεί κατά 35%. Αυτό είναι αναμενόμενο, καθώς η παραλληλοποίηση, σε αντίθεση με τον filteredVamana, ήταν λειτουργική, δηλαδή, πράγματι, η συνάρτηση εκτελέσθηκε παράλληλα.

Επιπλέον, όπως ήδη αναφέρθηκε, η παραλληλοποίηση των queries μείωσε σημαντικά τον χρόνο εκτέλεσης της εφαρμογής, σχεδόν έως 4 φορές λιγότερο.