Ονοματεπώνυμα: ***Α.Μ***
Κωνσταντίνος Βερικάκης *1115201900311*
Κατερίνα Τάτση *1115201900185*
Πέτρος Σαρρής *1115201900169*

Εισαγωγή

Το παραπάνω project αναφέρεται στην υλοποίηση του αλγορίθμου vamana και των παραλλαγών της, για αναζήτηση σε μεγάλες βάσεις δεδομένων, με σκοπό την δημιουργία ενός βελτιστοποιημένου γραφου.

Η αρχική διαδικασία χωρίστηκε σε 3 βασικά μέρη.

1. Αρχικά υλοποιήθηκε η απλή vamana, όπου δέχεται σαν είσοδο τα δεδομένα, τα ερωτήματα και τους σωστούς γείτονες και μέσω των βασικών συναρτήσεων greedy search και robust prune έπρεπε να δημιουργήσουμε έναν βελτιστοποιημένο γραφο.

2. Στην συνέχεια το εύρος δεδομένων είχε μεγαλύτερη ποικιλία,αφού έπρεπε να διαχειριστούμε δεδομένα τα οποία περιείχαν φίλτρα. Έπρεπε λοιπόν να αναθεωρηθεί λίγο η λειτουργικότητα των συναρτήσεων. Αρχικά τώρα υπήρχαν 2 vamana συναρτήσεις, η filtered και η stitched (η οποία καλούσε μέσα της την παλιά αρχική vamana), η συνάρτηση medoid τώρα έπρεπε να επιστρέφει ένα map με τα φίλτρα, και να δηλώνει το εναρκτήριο σημείο στην greedy όπως έκανε και στο πρώτο μέρος. Στην point προσθέσαμε ένα νέο δεδομένο για την καλύτερη διαχείριση των φίλτρων. Ενώ η robust prune είναι αρκετά παρόμοια η greedy είναι αρκετά αλλαγμένη καθώς διαχειρίζεται σημεία και με φίλτρα και χωρίς. Εννοείται υπάρχουν παντού κατάλληλα τεστ στις βασικές συναρτήσεις για να ελέγχεται η εγκυρότητα των αποτελεσμάτων.

3. Στο τελευταίο μέρος της εργασίας λόγω των μεγάλων όγκων των δεδομένων έπρεπε να εστιάσουμε στις βελτιστοποιήσεις. Η μεγαλύτερη και σημαντικότερη βελτιστοποίηση αφορά την παραλληλία του προγράμματος. Εκεί βλέπουμε σημαντικές βελτιώσεις στον χρόνο οι οποίες θα αναλυθούν παρακάτω. Η επόμενη βελτίωση που υλοποιήσαμε είναι η αφαίρεση του σημείου medoid και η αντικατάσταση του με ένα τυχαίο σημείο. Εκεί υπάρχουν κάποιες βελτιώσεις αλλά σε πολύ μικρό βαθμό. Τέλος, η τελευταία απόπειρα βελτιστοποίησης (η οποία δεν είχε κάποια βελτίωση στα αποτελέσματα, η προσπάθεια αρχικοποίησης γράφου με τυχαίες ακμές, αντί για τη δημιουργία κενού γράφου. Είναι σημαντικό όμως να μελετηθούν και αυτά τα αποτελέσματα για την καλύτερη κατανόηση της εφαρμογής.

Περιγραφή Υλοποίησης

Η βασική ιδέα είναι ότι αυτό το project υλοποιεί τον αλγόριθμο Vamana και τις αντίστοιχες παραλλαγές του, για αναζήτηση σε μεγάλες βάσεις δεδομένων με τη χρήση των medoid(όπου χρειάζεται), robust prune και greedy search. Ο αλγόριθμος χρησιμοποιεί ταυτόχρονα πολλαπλά νήματα (threads) για την αποτελεσματική εκτέλεση του, βασισμένος σε παραλληλία.

Ας δούμε συνοπτικά τη λειτουργία τους:

1. vamanaWorker και vamanaIndexing

- vamanaWorker: Παρόμοια με την παραπάνω συνάρτηση, αυτή η συνάρτηση είναι υπεύθυνη για τον συγχρονισμό της αρχικής vamana.

- vamanaIndexing: Εκτελεί τη βασική διαδικασία του Vamana indexing, δημιουργεί το γράφημα και το επεξεργάζεται χρησιμοποιώντας τον medoid(όπου χρειάζεται) και την παραλληλία μέσω νημάτων.

2. filteredVamanaWorker και filteredVamanaIndexing

- filteredVamanaWorker: Αυτή η συνάρτηση είναι υπεύθυνη για συγχρονισμό της filtered vamana.

- filteredVamanaIndexing: Ο κύριος αλγόριθμος που δημιουργεί το graph και εκτελεί τις λειτουργίες του filtered Vamana χρησιμοποιώντας πολλαπλά νήματα. Χρησιμοποιεί random permutation για να κατανεμηθούν τα δεδομένα και εκτελεί την αναζήτηση σε διαφορετικά κομμάτια του γραφήματος.

3. stitchedVamanaWorker και stitchedVamanaIndexing

- stitchedVamanaWorker: Αυτή η συνάρτηση με την ίδια λογική με τις παραπάνω είναι υπεύθυνη για τον συγχρονισμό της stitched vamana.

- stitchedVamanaIndexing: Ολοκληρώνει την διαδικασία του stitched Vamana indexing, επεξεργάζοντας τα φίλτρα σε πολλαπλά νήματα και ενσωματώνοντας τα γραφήματα που έχουν δημιουργηθεί σε ένα τελικό γράφημα.

Κύρια ροή της εφαρμογής

Το κύριο πρόγραμμα αποτελείται από 2 main συναρτήσεις, την filtered_vamana_main και την stitched_vamana_main οι οποίες δημιουργούν 2 ανεξάρτητα εκτελέσιμα. Διαβάζουν τα dataset και τα query points,και μέσω της κλήσης των κατάλληλων συναρτήσεων εκκινούν την διαδικασία δημιουργίας του τελικού γράφου.

Χρήση Συγχρονισμού και Παραλληλίας

Χρησιμοποιούνται semaphores και mutexes για συγχρονισμό μεταξύ των νημάτων, εξασφαλίζοντας την σωστή πρόσβαση στις κοινές δομές δεδομένων. Το πρόγραμμα αξιοποιεί ταυτόχρονα πολλά νήματα για να εκτελέσει την αναζήτηση σε διάφορα κομμάτια του γραφήματος, επιταχύνοντας τη διαδικασία.

Κύριες Συναρτήσεις

1. Greedy Search: Υπολογίζει ποιοι είναι οι καλύτεροι γείτονες για κάθε σημείο, χρησιμοποιώντας τον αλγόριθμο Greedy.

2. Robust Prune: Αφαιρεί περιττούς γείτονες από το γράφημα για να μειώσει το μέγεθος του και να κάνει πιο αποδοτική την αναζήτηση.

3. Medoid: Εύρεση του medoid των σημείων, δηλαδή του σημείου που έχει την ελάχιστη απόσταση από όλα τα υπόλοιπα σημεία.Στην περίπτωση φίλτρων,επιστρέφει ενα map όπου αντιστοιχεί το κάθε φίλτρο σε όλα τα σημεία που ανήκουν σε αυτό.(Το κάθε σημείο αποτελείται από ένα μόνο φίλτρο)

Εισροές και Εξόδους

Εισροές: Διαβάζονται δύο αρχεία: το dataset και τα queries.(στην αρχική vamana και το groundtruth, στην περίπτωση με τα φίλτρα πρέπει να το υπολογίσουμε μόνοι μας, το οποίο γίνεται σε ξεχωριστή συνάρτηση την evaluate).

Εξόδους: Το αποτέλεσμα είναι το γράφημα filtered vamana, το γράφημα Stitched Vamana, το k-recall για την αξιολόγηση του αλγορίθμου αναζήτησης και ο αντίστοιχος χρόνος εκτέλεσης για κάθε αλγόριθμο.

<u>Συνοψίζοντας:</u>
Αυτός ο κώδικας υλοποιεί μια εξελιγμένη παραλλαγή του απλού Vamana indexing χρησιμοποιώντας παραλληλία για βελτίωση της απόδοσης σε μεγάλα δεδομένα, και συνδυάζει διάφορες τεχνικές όπως το greedy search, το robust pruning, και το medoid(όπου χρειάζεται) για τη δημιουργία γραφημάτων και την αναζήτηση σε δεδομένα.

Ανάλυση απόδοσης

1.Αρχικοποίηση γράφων με τυχαίες ακμές

```
root@LAPTOP-CI36P151:/mnt/c/Users/unban/Desktop/telikiergasia2# ./main
# Dataset points: 10000
# Query points: 5012
# Dataset filters: 129
# valid filters = 2495
# valid filters = 2495
Filtered Vamana
k-recall = 0.704379 (alpha = 1.2, R = 12, L = 80, k = 100)
-----------------------------------------------
Stitched Vamana
k-recall = 0.704343 (alpha = 1.2, R_small = 12, R_stitched = 12, L_small = 80, k = 100)
```

Με βελτιστοποίηση

```
sdi1900169@LAPTOP-8AK6FJ9D:~/c_programs/1PROJECT$ ./main
# Dataset points: 10000
# Query points: 5012
# Dataset filters: 129
# valid filters = 2495
# valid filters = 2495
Filtered Vamana
k-recall = 0.684937 (alpha = 1.2, R = 12, L = 80, k = 100)
-----------------------------------------------
Stitched Vamana
k-recall = 0.694598 (alpha = 1.2, R_small = 12, R_stitched = 12, L_small = 80, k = 100)
```

Δημιουργήσαμε μια νέα συνάρτηση για αρχικοποίηση, η οποία στην αρχή ακολουθούσε δημιουργούσε τα nodes και στη συνέχεια ένωνε με τη λογική του R-Regular όπως στην κλασική vamana των αρχικών ερωτημάτων, τις κορυφές με ίδιο φίλτρο, δημιουργώντας έτσι ανεξάρτητους, μεταξύ τους, υπογράφους. Στο τέλος ένωνε μεταξύ τους τις κορυφές με διαφορετικά φίλτρα από τους υπογράφους αυτούς και έτσι ολοκλήρωνε τον γράφο. Αυτό στο τέλος, ναι μεν κρατούσε ίδιους τους χρόνους, όμως μείωνε, για λίγο, το recall τόσο στον filtered vamana όσο και στον stitched vamana, όπως φαίνεται και στην εικόνα. Εφόσον δεν υπήρξε βελτιστοποίηση, δεν κρατήσαμε την υλοποίηση.

2.Αρχικοποίηση medoid στον Vamana με τυχαία σημεία

plain vamana

```
root@LAPTOP-CI36P151:/mnt/c/Users/unban/Desktop/alla/1PROJECT-master/1PROJECT-master# ./main
k-recall = 0.9976 (R = 20, L = 200, k = 100)
```

Με βελτιστοποίηση

```
root@LAPTOP-CI36P151:/mnt/c/Users/unban/Desktop/alla/1PROJECT-master/1PROJECT-master# ./main
k-recall = 0.9978 (R = 20, L = 200, k = 100)
```

Ξεκινώντας τη συγκεκριμένη βελτιστοποίηση από την απλή , αρχική βαμάνα , παρατηρούμε ότι με τα ίδια ακριβώς ορίσματα βλέπουμε μια μικρή βελτίωση στο recall. Αυτό που συμβαίνει στην πραγματικότητα είναι ότι για να χρησιμοποιηθεί το σημείο medoid ως σημείο έναρξης και ως όρισμα στην greedy search,τώρα παίρναμε ένα τυχαίο σημείο. Είναι λογικό λοιπόν και το recall να είναι είναι πάντα σταθερό,αλλά παρατήσουμε ότι κατά πλειοψηφία είναι ελάχιστα μεγαλύτερο,αλλά πολύ μικρή διαφορά.Η τόση μικρή μικρή διαφορά και η εναλλαγή του recall προφανώς ευθύνεται στην τυχαιότητα του σημείου που επιλέγεται.

stitched vamana

```
root@LAPTOP-CI36P151:/mnt/c/Users/unban/Desktop/kostis# ./stitched_vamana_main
# Dataset points: 10000
# Query points: 5012
# Dataset filters: 129
# valid filters = 2495
------------------------------------------------
Stitched Vamana Complete
Build time: 5.692s
k-recall = 0.860724 (a = 1.2, R_small = 12, R_stitched = 16, L_small = 80, k = 100)
```

Με βελτιστοποίηση

```
root@LAPTOP-CI36P151:/mnt/c/Users/unban/Desktop/kostis# ./stitched_vamana_main
# Dataset points: 10000
# Query points: 5012
# Dataset filters: 129
# valid filters = 2495
--------------------------------------------
Stitched Vamana Complete
Build time: 5.936s
k-recall = 0.860813 (a = 1.2, R_small = 12, R_stitched = 16, L_small = 80, k = 100)
```

Όσον αφορά την stitched vamana τώρα ,η οποία μέσα της καλεί την plain vamana,έχει και αυτή μικρή βελτιστοποίηση στο recall. Εδώ παρατηρούμε όμως και τα αποτελέσματα από το build Βλέπουμε λοιπόν ότι ο χρόνος αυξάνεται.Και αυτό είναι επόμενο αν σκεφτεί κανείς ότι λόγω του ότι δεν παίρνουμε το πιο κεντρικό σημείο,μπορεί να πάρουμε κάποιο σημείο από τις άκρες για παράδειγμα και επομένως αυτό να καθυστερήσει την διαδικασία. Παρολα αυτα ,ο χρόνος όπως παρατήσουμε αυξάνεται ελάχιστα.

3.Παραλληλοποίηση στις συναρτήσεις

filtered vamana

```
root@LAPTOP-CI36P151:/mnt/c/Users/unban/Desktop/2ergasia# ./filtered_main
# Dataset points: 10000
# Query points: 5012
# Dataset filters: 129
# valid filters = 2495
Filtered Vamana
Build time: 75.776s
k-recall = 0.704423 (alpha = 1.2, R = 12, L = 80, k = 100)
```

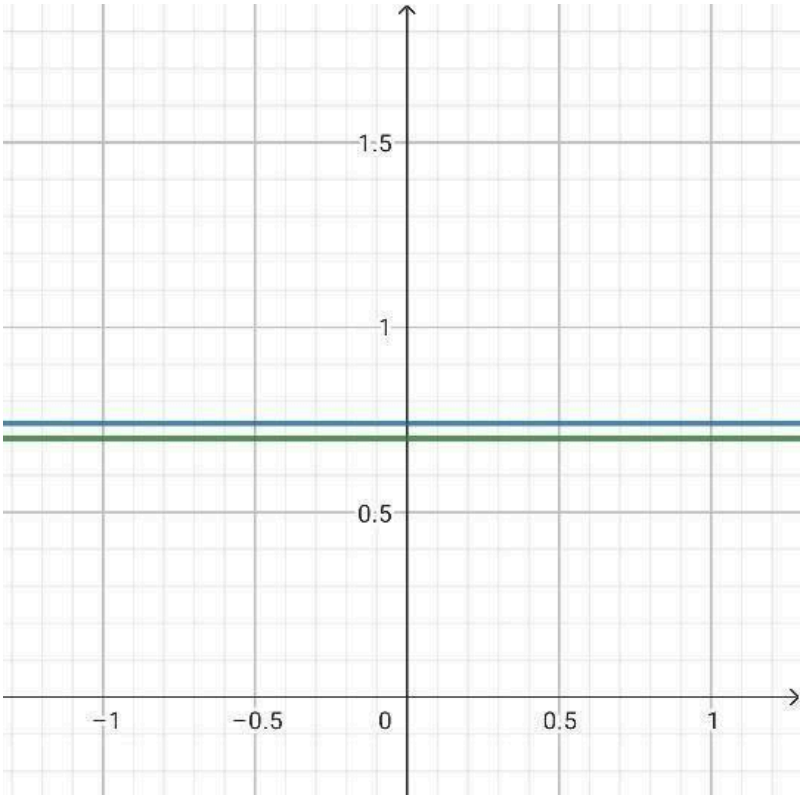
```
root@LAPTOP-CI36P151:/mnt/c/Users/unban/Desktop/2ergasia# ./filtered_main
# Dataset points: 1000000
# Query points: 5012
# Dataset filters: 1142
# valid filters = 2496
Filtered Vamana
Build time: 7323.63s
k-recall = 0.765311 (alpha = 1.2, R = 12, L = 80, k = 100)
```

Με βελτιστοποίηση

```
# Dataset points: 10000
# Query points: 5012
# Dataset filters: 129
# valid filters = 2495
------------------------------------------------
Filtered Vamana Complete
Build time: 6.58s
k-recall = 0.704315 (a = 1.2, R = 12, L = 80, k = 100)
```

```
# Dataset points: 1000000
# Query points: 5012
# Dataset filters: 1142
# valid filters = 2496
------------------------------------------------
Filtered Vamana Complete
Build time: 1288.39s
k-recall = 0.762318 (a = 1.2, R = 12, L = 80, k = 100)
```

Γραφική αναπαράσταση της βελτιστοποίησης



stitched vamana

```
root@LAPTOP-CI36P151:/mnt/c/Users/unban/Desktop/2ergasia# ./stitched_main
# Dataset points: 10000
# Query points: 5012
# Dataset filters: 129
# valid filters = 2495
Stitched Vamana
Build time: 74.317s
k-recall = 0.704439 (alpha = 1.2, R_small = 12, R_stitched = 12, L_small = 80, k = 100)
```

Με βελτιστοποίηση

```
# Dataset points: 10000
# Query points: 5012
# Dataset filters: 129
# valid filters = 2495
-----------------------------------------------
Stitched Vamana Complete
Build time: 5.082s
k-recall = 0.704383 (a = 1.2, R_small = 12, R_stitched = 12, L_small = 80, k = 100)
```

```
# Dataset points: 1000000
# Query points: 5012
# Dataset filters: 1142
# valid filters = 2496
-----------------------------------------------
Stitched Vamana Complete
Build time: 989.367s
k-recall = 0.759068 (a = 1.2, R_small = 12, R_stitched = 12, L_small = 80, k = 100)
```

Παραλληλοποίηση στον Κώδικα:
Χρησιμοποιούμε POSIX threads (pthread) για παράλληλη επεξεργασία
Ο αριθμός των threads καθορίζεται από το hardware_concurrency()
Κάθε thread επεξεργάζεται ένα διαφορετικό τμήμα (chunk) των δεδομένων
Χρησιμοποιείτε mutexes και semaphores για συγχρονισμό πρόσβασης στον γράφο.

Καταμερισμός Εργασίας:
Ο αλγόριθμος χωρίζει το σύνολο των σημείων σε chunks
Κάθε thread επεξεργάζεται ένα διαφορετικό chunk
Επιτρέπει παράλληλη επεξεργασία διαφορετικών τμημάτων του γράφου

Αποδοτική Διαχείριση Πόρων:
Χρήση όλων των διαθέσιμων πυρήνων του επεξεργαστή
Καλύτερη αξιοποίηση των πόρων του συστήματος

Παράλληλη Κατασκευή:
Παράλληλη εκτέλεση των λειτουργιών greedy search
Παράλληλο pruning των γειτόνων
Συγχρονισμένη ενημέρωση του γράφου

Σημαντική Παρατήρηση:
Στην αρχική έκδοση, η επεξεργασία κάθε σημείου γινόταν σειριακά, ένα-ένα
Με την παραλληλοποίηση, πολλαπλά σημεία επεξεργάζονται ταυτόχρονα
Παρατηρούμε ότι στα 1.000.000 σημεία ο σειριακός αλγόριθμος χρειάζεται πάνω από 2 ώρες.Αυτό δείχνει ότι ο σειριακός αλγόριθμος κλιμακώνεται πολύ άσχημα με την αύξηση των δεδομένων. Ο σειριακός κώδικας χρησιμοποιούσε μόνο έναν πυρήνα του επεξεργαστή.Η παραλληλοποίηση αξιοποιεί όλους τους διαθέσιμους πυρήνες, βελτιώνοντας δραματικά την απόδοση.Η παραλληλοποίηση αποδεικνύεται κρίσιμη για την πρακτική εφαρμογή του αλγορίθμου, μειώνοντας τους χρόνους εκτέλεσης από μη πρακτικούς (ώρες) σε διαχειρίσιμους (δευτερόλεπτα ή λεπτά).

4.Παραλληλοποίηση στις συναρτήσεις vamana

Δοκιμάσαμε να τρέξουμε παράλληλα και την stitched και την filtered vamana αλλά αυτό παρατηρήσαμε ότι δεν είχε καλύτερα αποτελέσματα ούτε στο recall ούτε στον χρόνο. Επιπλέον ήταν μια βελτιστοποίηση η οποία δεν μας επιτρέπει να χρόνο μετρηθεί καθαρά το building time.

Συμπεράσματα

Με βάση τα παραπάνω αποτελέσματα που πήραμε από τις βελτιώσεις που εφαρμόσαμε στον κώδικα μας , παρατηρούμε ότι η αρχικοποίηση γραφών με τυχαίες ακμές δεν οδήγησε σε βελτιστοποίηση του κώδικα.Η βελτιστοποίηση για την αρχικοποίηση medoid με τυχαία σημεία, παρατηρήσαμε ότι βελτίωσε το recall ακόμα και αν χρειάστηκε λίγο παραπάνω χρόνο.Όσον αφορά την τελευταία βελτιστοποίηση η οποία είχε και την μεγαλύτερη χρονική βελτίωση παρατηρούμε τεράστια διαφορά σε χρονικά αποτέλεσμα.Αυτο παρατηρούμε ότι διογκώνεται ακραία όσο μεγαλύτερα δεδομένα βάζουμε.Διάφορες μεταξύ ωρών μετατρέπονται σε μόλις ελάχιστα δευτερόλεπτα.Μια τελευταία βελτίωση που δοκιμάσαμε να υλοποιήσουμε αφορούσε την παράλληλη εκτέλεση της stitched και της filtered vamana.Την αφαιρέσαμε όμως όπως αναφέρθηκε παραπάνω λόγω του ότι δεν υπήρχε βελτίωση στο recall και δεν μας επέτρεπε να χρόνο μετρήσουμε καθαρά το building time.