Τελική αναφορά project 2024-2025

Οικονόμου Νεφέλη - 1115202100127
Κυριακάκος Μιχαήλ - 1115202100074

Εισαγωγή:

Η εργασία αυτή αφορούσε στην βελτιστοποίηση του προβλήματος αναζήτησης του εγγύτερου γείτονα (Nearest Neighbour Search - NNS) με εστίαση στην εύρεση των k κοντινότερων γειτόνων ενός σημείου με βάση κάποιο γράφο-ευρετήριο.

Για το πρώτο παραδοτέο ακολουθήθηκε μία προσεγγιστική εύρεση των KNN, όπου κατά τη δημιουργία του ευρετηρίου κατασκευάζεται ένας γράφος G, ο οποίος μετά διασχίζεται από έναν φυσικά άπληστο αλγόριθμο (GreedySearch) για να καταλήξουμε στο εκάστοτε ερώτημα. Για την βελτιστοποίηση των γράφων προς όφελος της αναζήτησης και της κατασκευής του ευρετηρίου χρησιμοποιείται και ένας ακόμα αλγόριθμος (RobustPrune) για να καθορίζει τους εξερχόμενους γείτονες του κάθε σημείου του γράφου. Ο επαναληπτικός αλγόριθμος που συνδυάζει τους δύο προηγούμενους (Vamana) είναι υπεύθυνος για την δημιουργία του τελικού κατευθυνόμενου γράφου. Ξεκινώντας από έναν τυχαία αρχικοποιημένο γράφο G ο αλγόριθμος, με αφετηρία το μέσο του συνόλου των σημείων, επισκέπτεται τυχαία κάθε σημείο, ενημερώνοντας κάθε φορά τον γράφο καλώντας πρώτα την GreedySearch και μετά την RobustPrune. Έπειτα προσθέτει απαραίτητες εισερχόμενες ακμές για τους εξερχόμενους γείτονες διασφαλίζοντας σωστές συνδέσεις που ταιριάζουν καλύτερα με τα αποτελέσματα της GreedySearch που εξασφαλίζουν τη σύγκλιση. Τέλος διασφαλίζει ότι τυχόν προσθέσεις δεν θα αυξήσουν πολύ τον βαθμό των εξερχόμενων ακμών καλώντας ξανά την RobustPrune. Σταδιακά ο αλγόριθμος γίνεται καλύτερος για την GreedySearch, μέχρις ότου να τερματιστεί, όποτε έχουμε και το τελικό ευρετήριο πάνω στο οποίο μπορούμε να κάνουμε τα ερωτήματά μας.

Το δεύτερο παραδοτέο αφορούσε στην περαιτέρω επέκταση της λογικής του πρώτου με την επιπλέον προσθήκη φίλτρων στα σημεία. Αυτό είχε ως αποτέλεσμα προφανώς την αλλαγή των αλγορίθμων προκειμένου να υποστηρίζουν τη χρήση φίλτρων, να ξεχωρίζουν και να διαχειρίζονται τα διάφορα σημεία με βάση αυτά. Έτσι είδαμε μια πιο συλλογική αντιμετώπιση των σημείων με βάση την ομαδοποίησή τους σε φίλτρα.

Τέλος, το τρίτο μέρος αφορούσε σε βελτιστοποιήσεις και πειράματα πάνω στους προϋπάρχοντες αλγορίθμους, ώστε να μελετηθούν οι διαφορές πάνω στην εκτέλεσή τους.

Παρακάτω θα εξετάσουμε αναλυτικά τυχόν σχεδιαστικές αποφάσεις και παρατηρήσεις πάνω στην υλοποίησή μας, καθώς και θα παρουσιάσουμε τα αποτελέσματα των πειραμάτων μας πάνω στις διάφορες βελτιστοποιήσεις, όπως και τα συμπεράσματα που αντλήσαμε από αυτά.

1ο Παραδοτέο:

Επιλέξαμε να χρησιμοποιήσουμε C οπότε υλοποιήσαμε μόνοι μας ότι τυχόν δομές αναφερθούν στην συνέχεια.
Για να μπορέσουμε να επεξεργαστούμε τα δεδομένα που μας δόθηκαν από το texmex τα αποθηκεύσαμε τοπικά και έπειτα τα ανακτήσαμε σε μορφή δισδιάστατου πίνακα όπου κάθε στήλη αντιστοιχεί σε ένα σημείο του συνόλου (vector), ενώ οι γραμμές αντιστοιχούν στα components του vector.

Όσον αφορά στον Vamana μια σημαντική παρατήρηση είναι το πως κατασκευάσαμε το ευρετήριο. Αποφασισαμε να υλοποιήσουμε τον γράφο G ως ένα array από vectors όπου κάθε θέση (index) του πίνακα αντιστοιχεί στο σημείο του dataset με το ίδιο index και οι γραμμές περιέχουν τα indices των γειτόνων του εκάστοτε σημείου.

Για διευκόλυνση και για να αποφύγουμε όσο το δυνατόν περισσότερο χρόνο με το να επαναλαμβάνουμε υπολογισμούς, δημιουργήσαμε ένα δισδιάστατο πίνακα αποστάσεων (dist_matrix), τον οποίο γεμίσαμε κατά τη διάρκεια υπολογισμού του medoid και περιέχει τις αποστάσεις μεταξύ όλων των σημείων του dataset.

Στα πλαίσια του GreedySearch υλοποιήσαμε το σύνολο των k κοντινότερων γειτόνων ως ένα priority queue και όχι σαν set όπως είχαμε αρχικά σκεφτεί γεγονός που βελτίωσε δραματικά την απόδοση από άποψη χρόνου, διότι ήταν ανά πάσα στιγμή ταξινομημένη με βάση την απόσταση από το εκάστοτε query (η οποία αποθηκεύεται κιόλας στη δομή εφόσον αυτό είναι αναγκαίο). Γεγονός που έκανε την διαχείρισή της και ιδιαίτερα την διαγραφή από αυτή (πράγμα που γίνεται συχνά) πολύ πιο γρήγορη. Το σύνολο των κόμβων που επισκεφτήκαμε διατηρήθηκε ως set καθώς δεν παρουσίαζε ιδιαίτερα μεγάλες λειτουργικές απαιτήσεις. Αξίζει να σημειωθεί, πως στην GreedySearch ακολουθήθηκε αντίστοιχη λογική για των υπολογισμό τυχόν αναγκαίων αποστάσεων όπως αναφέρθηκε και παραπάνω. Δεν χρησιμοποιήθηκαν οι αποστάσεις του ήδη υπάρχοντος dist_matrix διότι θέλαμε να διατηρήσουμε την γενικότητα του κώδικα, ώστε να μπορεί να χρησιμοποιηθεί και για την τελικά αναζήτηση του query στον ολοκληρωμένο γράφο, ακόμα και αν αυτό το query δεν είναι μέρος του dataset.

Δεν έχουμε να αναφέρουμε κάτι ιδιαίτερο για τον RobustPrune, καθώς χρησιμοποιήσαμε τις δομές που αναλύσαμε παραπάνω και τα τυπικά του αλγορίθμου ακολουθήθηκαν πιστά από την εκφώνηση.

2ο Παραδοτέο:

Στο δεύτερο παραδοτέο, όπως και στο πρώτο, αποθηκεύσαμε τοπικά όποια αρχεία με datasets χρειαζόμασταν και έπειτα ανακτήσαμε τα περιεχόμενά τους με τη μορφή δισδιάστατου πίνακα αντίστοιχης φιλοσοφίας (στήλες → vectors, γραμμές → components) με την επιπλέον προσθήκη κάποιον χαρακτηριστικών όπως filters, timestamps, κ.α.

Δεδομένου του ότι εμείς εργαζόμασταν μόνο με datapoints με ένα μόνο φίλτρο, ενώ κανονικά οι αλγόριθμοι είναι προσαρμοσμένοι για να διαχειρίζονται δεδομένα με ένα ή περισσότερα φίλτρα, στο σημείο αυτό θα αναφέρουμε διάφορες παραδοχές που κάναμε στους αλγορίθμους και στη σχεδίαση των δομών μας.

Πρώτα θα αναλύσουμε τον Filtered Vamana index:

- ➔ Αρχικά χρησιμοποιήσαμε ξανά μία συνάρτηση για την εύρεση του medoid του συνόλου, καθώς, ακόμα και αν αυτό δεν έχει ιδιαίτερη σημασία για την συγκεκριμένη προσέγγιση, είναι ένας αποδοτικός τρόπος να υπολογίσουμε τις αποστάσεις μεταξύ όλων των σημείων του dataset μας.
- ➔ Έπειτα, δημιουργήσαμε ένα map (filtered_data) το οποίο αντοιστοιχίζει κάθε φίλτρο με τα αντίστοιχα σημεία του dataset. Για τον σκοπό αυτό υλοποιήσαμε μία δομή hashmap, όπου τα φίλτρα είναι τα κλειδιά, και το αντίστοιχο array είναι σχεδιασμένο ώστε να είναι αρκετά μεγάλο προκειμένου κάθε φίλτρο να αντιστοιχίζεται σε μία διαφορετική θέση του. Άρα ουσιαστικά κάθε θέση του πίνακα “κρατάει” τα σημεία ενός συγκεκριμένου φίλτρου σε ένα σύνολο (set). Η αντιστοίχιση γίνεται με βάση το παρακάτω hash function: (filter-min)%table_size.
- ➔ Βασιζόμενοι πλέον στην οργάνωση των σημείων ανά φίλτρο μέσω του map filtered_data υλοποιήσαμε την Find Medoid, για την οποία ακολουθήσαμε τον αρχικό αλγόριθμο. Έτσι κατασκευάσαμε το map filtered_medoids όπου αντιστοιχίζει σε κάθε φίλτρο ένα τυχαίο επιλεγμένο medoid.
- ➔ Όσον αφορά στον Filtered Greedy Search, μία μικρή διαφορά είναι ότι για την εύρεση του starting point της αναζήτησης, χρησιμοποιήσαμε κατευθείαν το medoid που αντιστοιχεί στο εκάστοτε φίλτρο του σημείου που μας ενδιαφέρει, από το filtered_medoids, εφόσον πάντα μιλάμε για σημεία με ένα μοναδικό φίλτρο. Αντίστοιχα και για την επιλογή τον εξερχόμενων γειτόνων στη συνέχεια, απλά απορρίπταμε όσα σημεία είχαν διαφορετικό φίλτρο από το επιθυμητό.
- ➔ Για τον Filtered Robust Prune αυτό που αξίζει να αναφέρουμε στην περίπτωση μοναδικών φίλτρων είναι η συνθήκη σύμφωνα με την οποία αφαιρούμε τους γειτονικούς κόμβους, όπου στην προκειμένη περίπτωση γίνεται αφαίρεση γειτόνων μόνο όταν τα σημεία είναι του ίδιου φίλτρου.

Ο τελικός γράφος διατήρησε την ίδια δομή με το προηγούμενο παραδοτέο στην περίπτωση του Filtered Vamana index.

Όσον αφορά τώρα στον Stitched Vamana Index, λαμβάνοντας υπόψη τη διαφορετική φιλοσοφία του αλγορίθμου, ο οποίος δημιουργεί ένα ξεχωριστό γράφο για τα σημεία του κάθε φίλτρου, αποφασίσαμε να τον δομήσουμε σαν έναν πίνακα από γράφους. Δηλαδή το αποτέλεσμα που επιστρέφει ο Stitched Vamana είναι τελικά ένας πίνακας filter θέσεων, όπου filter ο αριθμός όλως των διαφορετικών φίλτρων του dataset, ο οποίος σε κάθε θέση περιέχει τον αντίστοιχο γράφο του συγκεκριμένου φίλτρου, που όπως αναφέρθηκε προηγουμένως είναι ένα array από vectors.

Ειδικότερα, κάποιες από τις σχεδιαστικές επιλογές που κάναμε για την υλοποίηση του Stitched Vamana είναι οι εξής:

- ➔ Όπως αναφέρθηκε παραπάνω για τον Filtered Vamana έτσι και για τον Stitched αξιοποιήσαμε σε μεγάλο βαθμό την κατηγοριοποίηση των δεδομένων σε φίλτρα με το map filtered_data.
- ➔ Αφού λοιπόν δημιουργήσαμε αυτόν, όπως και τον filter_medoids μέσω του Find Medoid, αποφασίσαμε να κατασκευάσουμε ένα array από vectors (per) το οποίο να αποθηκεύει το αρχικό permutation των indices των σημείων του εκάστοτε φίλτρου.
  Αυτό μας είναι απαραίτητο γιατί στη συνέχεια, για κάθε φίλτρο, δημιουργούμε εκ νέου ένα προσωρινό dataset (data_f) το οποίο να είναι διαχειρίσιμο από τον αρχικό αλγόριθμο Vamana.
- ➔ Έχοντας λοιπόν το data_f για κάθε φίλτρο, καλούμε τον Vamana προκειμένου να κατασκευάσει τον γράφο του αντίστοιχου φίλτρου, τον οποίο μετά αποθηκεύουμε στο ευρύτερο Stitched Vamana Index στη θέση που αντιστοιχεί στο φίλτρο.
- ➔ Δεν έχουμε κάτι ιδιαίτερο να αναφέρουμε για τον Vamana καθώς είναι ίδιος λειτουργικά με το πρώτο παραδοτέο.
- ➔ Εφόσον κάθε σημείο έχει ένα μόνο φίλτρο, δεν δημιουργούνται ακμές οι οποίες να ενώνουν τους διαφορετικούς υπογράφους μεταξύ τους, έτσι αποφασίσαμε να παραλείψουμε το βήμα του Pruning στο τέλος του Stitched Vamana, ενώ παρατηρήσαμε ότι αυτό δεν είχε αρνητικές επιπτώσεις στο accuracy μας.

3ο Παραδοτέο:

Το τρίτο παραδοτέο αφορούσε στην βελτιστοποίηση των ήδη υπάρχων αλγορίθμων με σκοπό να πειραματιστούμε με τα όρια της εφαρμογής μας, καθώς και στην σύνταξη της αναφοράς αυτής.

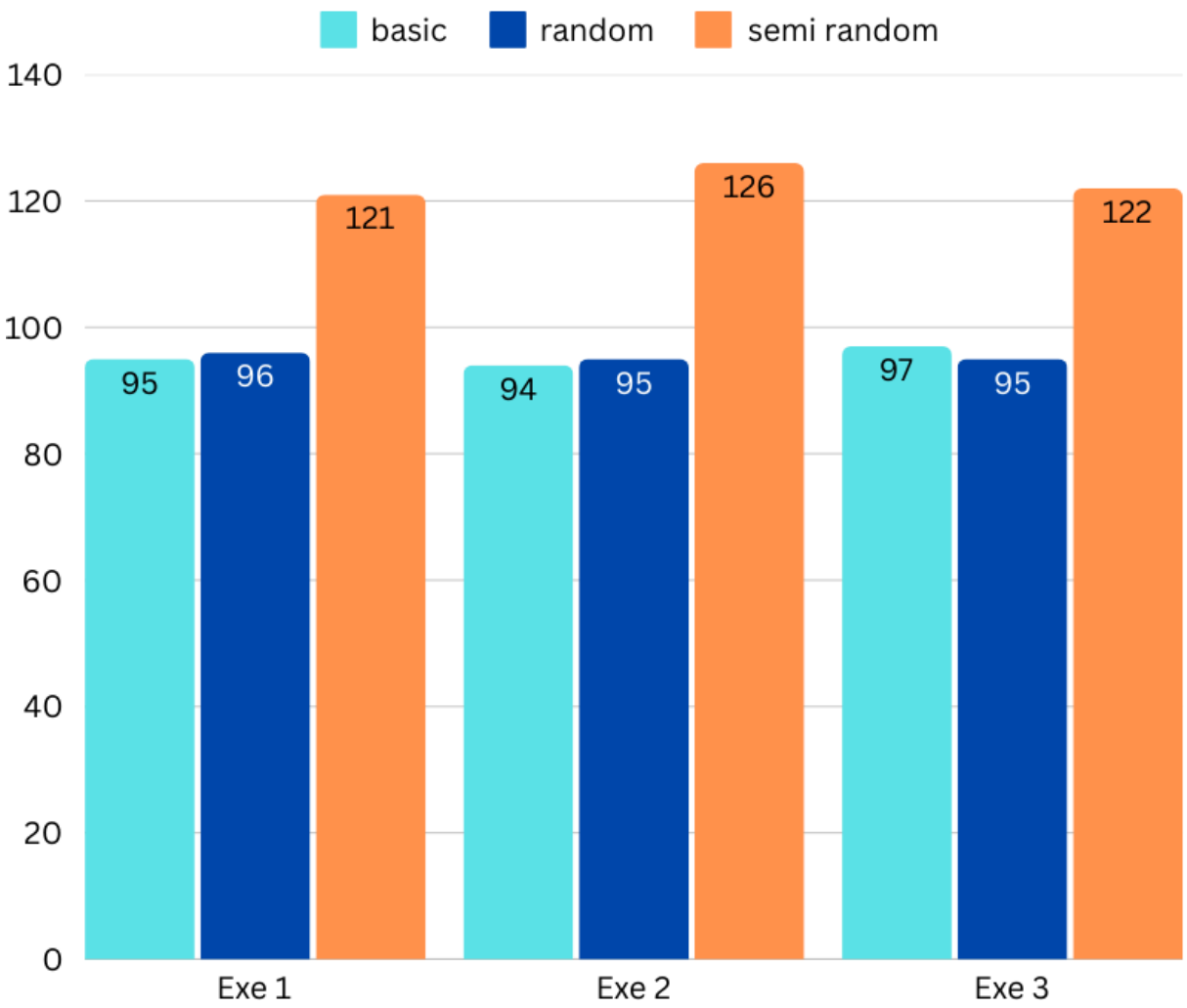
Στο σημείο θα παρουσιάσουμε τυχόν βελτιστοποιήσεις που υλοποιήσαμε, καθώς και τα ευρήματά μας από την εκπόνηση πειραμάτων πάνω στα διάφορα στάδια υλοποίησης των αλγορίθμων όλων των παραδοτέων.

Αρχικοποίηση medoid στον Vamana με τυχαία σημεία

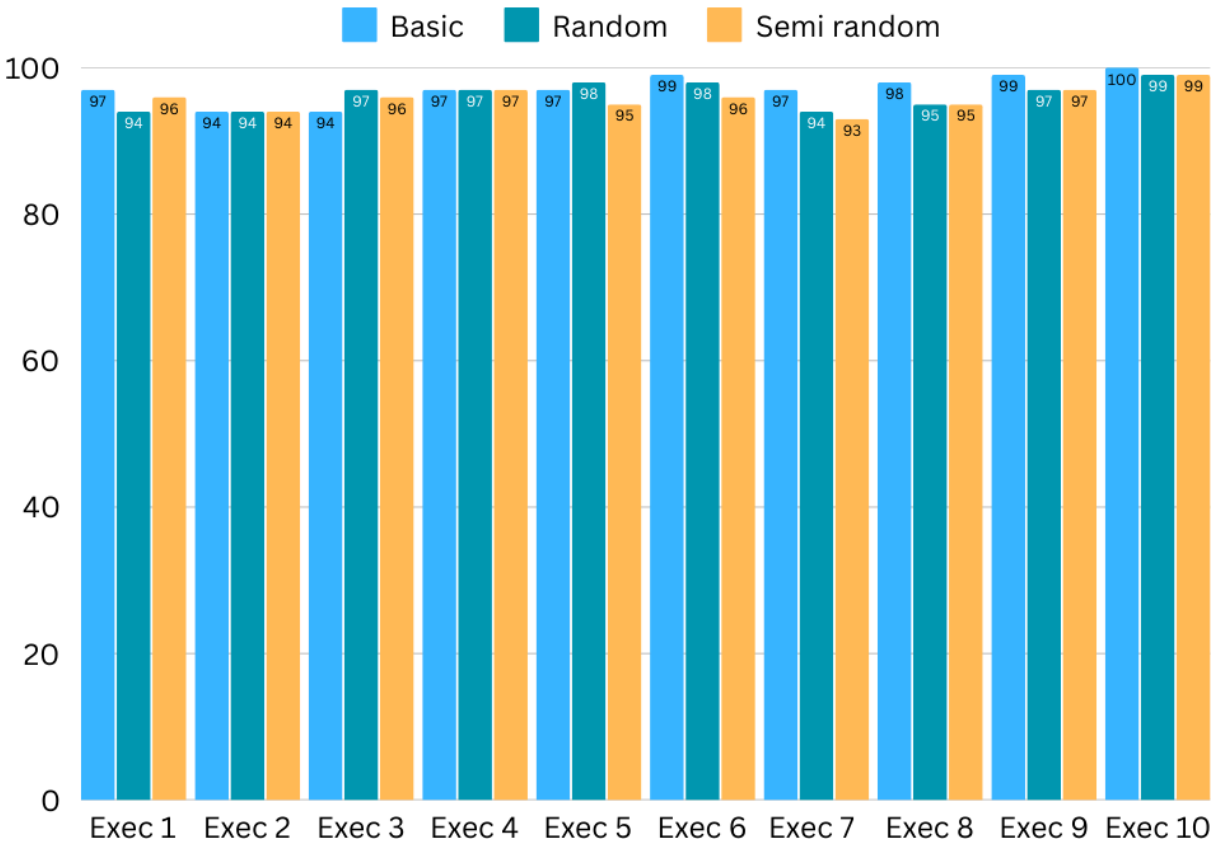
Υλοποιήσαμε δύο επιπλέον εκδόσεις του αρχικού αλγορίθμου Vamana:

- Vamana_random: όπου απλά επιλέξαμε ένα τυχαίο σημείο σαν το μέσο (medoid).
- Vamana_semi_random: όπου επιλέξαμε ένα σύνολο τυχαίων σημείων του dataset και στην συνέχεια ορίσαμε σαν medoid το μέσο αυτών.

Το πιο σημαντικό πράγμα που παρατηρήσαμε είναι ότι ο μέσος όρος των recall και των τριών υλοποιήσεων είναι ικανοποιητικός. Έπειτα, διαπιστώνουμε ότι ο χρόνος υλοποίησης του basic και του random είναι πολύ κοντά, ωστόσο, ο semi random είναι πολύ κοστοφόρος. Παρακάτω επισυνάπουμε έναν γράφο ο οποίος δείχνει αυτά που προαναφέρθηκαν.



Επιπλέον, έχουμε δημιουργήσει και έναν αλγόριθμο για το accuracy του κάθε αλγορίθμου:



Αρχικοποίηση γράφων με τυχαίες ακμές

Υλοποιήσαμε μία επιπλέον έκδοση για τον Filtered και μία για τον Stitched Vamana αντίστοιχα:

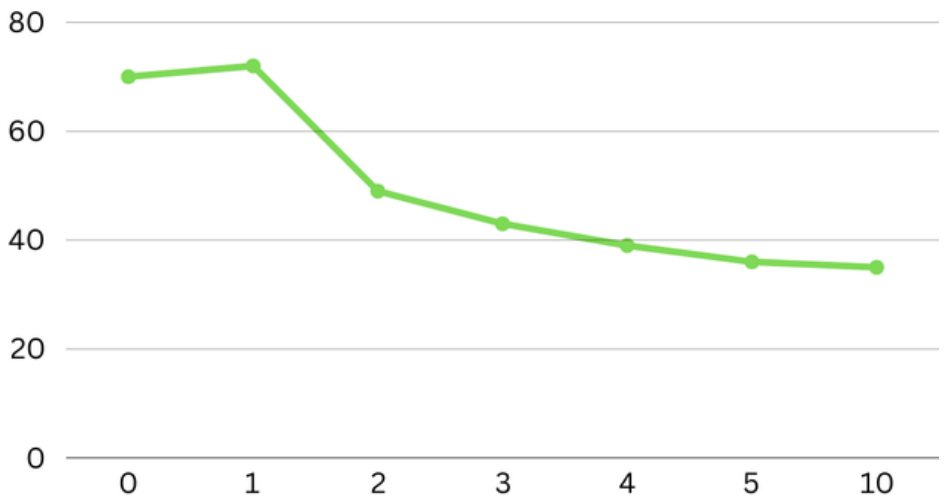
- FilteredVamanaIndexing_randomG: όπου αρχικοποιήσαμε τον γράφο G με κάποιους τυχαία επιλεγμένους γείτονες, αντί να δοθεί κενός στην κατασκευή του index.
- StitchedVamanaIndexing_randomG: όπου αντίστοιχα πριν καλέσουμε τον Vamana σε κάθε για την κατασκευή των υπογράφων αρχικοποιήσαμε τον γράφο G με κάποιους τυχαία επιλεγμένους γείτονες του ίδιου, κάθε φορά, φίλτρου. Σε αυτήν υλοποίηση βλέπουμε ότι τρέχει με μέσο όρο περίπου 30 δευτερόλεπτα.

Παράλληλη εκτέλεση

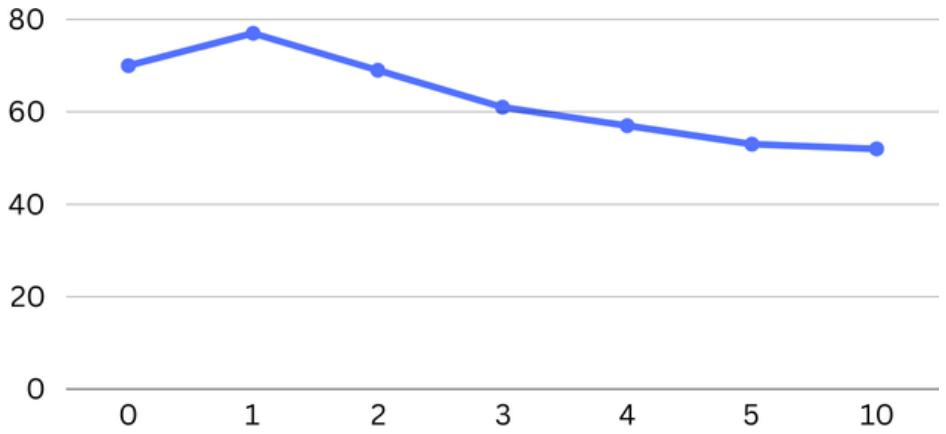
Για τη παράλληλη εκτέλεση αποφασίσαμε να παραλληλοποιήσουμε κάποια σημεία του Vamana και του Stitched Vamana:

- Για τον Vamana εκμεταλλευτήκαμε τα threads για τον παράλληλο υπολογισμό αποστάσεων κατά την κατασκευή του distance matrix στον υπολογισμό του medoid. Για τον σκοπό αυτό δημιουργήσαμε μία δομή με όλα τα δεδομένα που θα πρέπει να διαχειρίζονται τα διάφορα threads αναμεταξύ τους (thread data), καθώς και ένα mutex για να εξασφαλίσουμε τον ασφαλή και σωστό συγχρονισμό τους. Οι εργασίες που καλούνταν να κάνει το κάθε thread φαίνονται από τις συναρτήσεις calculate_distances, calculate_distances_simple, calculate_distances_random (κάθε μία από αυτές αντιστοιχεί σε μία διαφορετική εκδοχή του Vamana).
- Για τον Stitched Vamana χρησιμοποιήσαμε threads για την κατασκευή των επιμέρους υπογράφων, αφού αυτοί είναι ανεξάρτητοι, δεδομένου ότι οι διάφοροι υπογράφοι δεν συνδέονται μεταξύ τους λόγω των διαφορετικών φίλτρων. Έτσι, αντίστοιχα με προηγουμένος αυτό δημιουργήσαμε μία δομή με όλα τα δεδομένα που θα πρέπει να διαχειρίζονται τα διάφορα threads (thread data stitch), καθώς και δύο mutexes, αυτή τη φορά, τον συγχρονισμό τους. Οι εργασίες που καλούνταν να κάνει το κάθε thread φαίνονται από τις συναρτήσεις build_vamana_index και build_vamana_index_random (κάθε μία από αυτές αντιστοιχεί σε μία διαφορετική εκδοχή του Stitched Vamana).

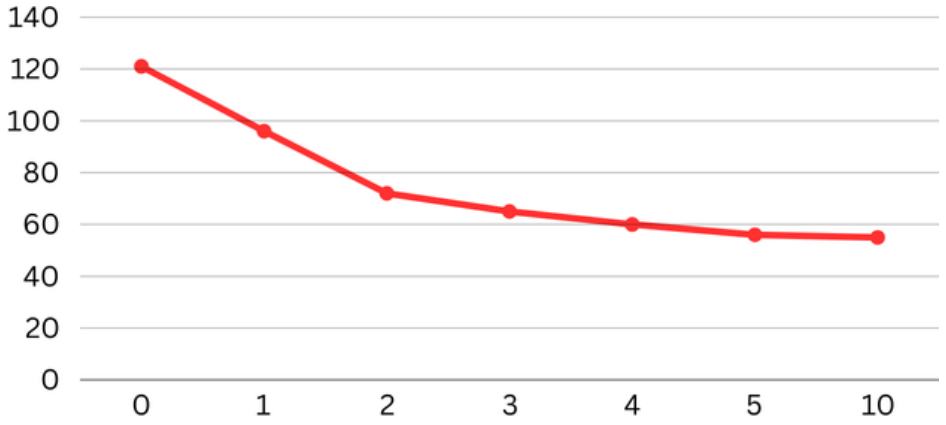
Εδώ πρέπει να δώσουμε ιδιαίτερη προσοχή καθώς, με την χρήση των threads καταφέραμε να βελτιώσουμε αισθητά την απόδοση των προγραμμάτων μας.
Παρακάτω έχουμε δημιουργήσει κάποια διαγράμματα αναπαριστούν την απόδοση των threads. Σε κάθε εκτέλεση αυξάναμε κατά ένα το πλήθος τους.



1. Το παραπάνω σχήμα μας δείχνει την μείωση του χρόνου εκτέλεσης του προγράμματος filtered vamana καθώς αυξάνουμε το πλήθος των threads.



2. Το παραπάνω σχήμα μας δείχνει την μείωση του χρόνου εκτέλεσης του προγράμματος basic vamana καθώς αυξάνουμε το πλήθος των threads.



3. Το παραπάνω σχήμα μας δείχνει την μείωση του χρόνου εκτέλεσης του προγράμματος vamana semi random καθώς αυξάνουμε το πλήθος των threads.

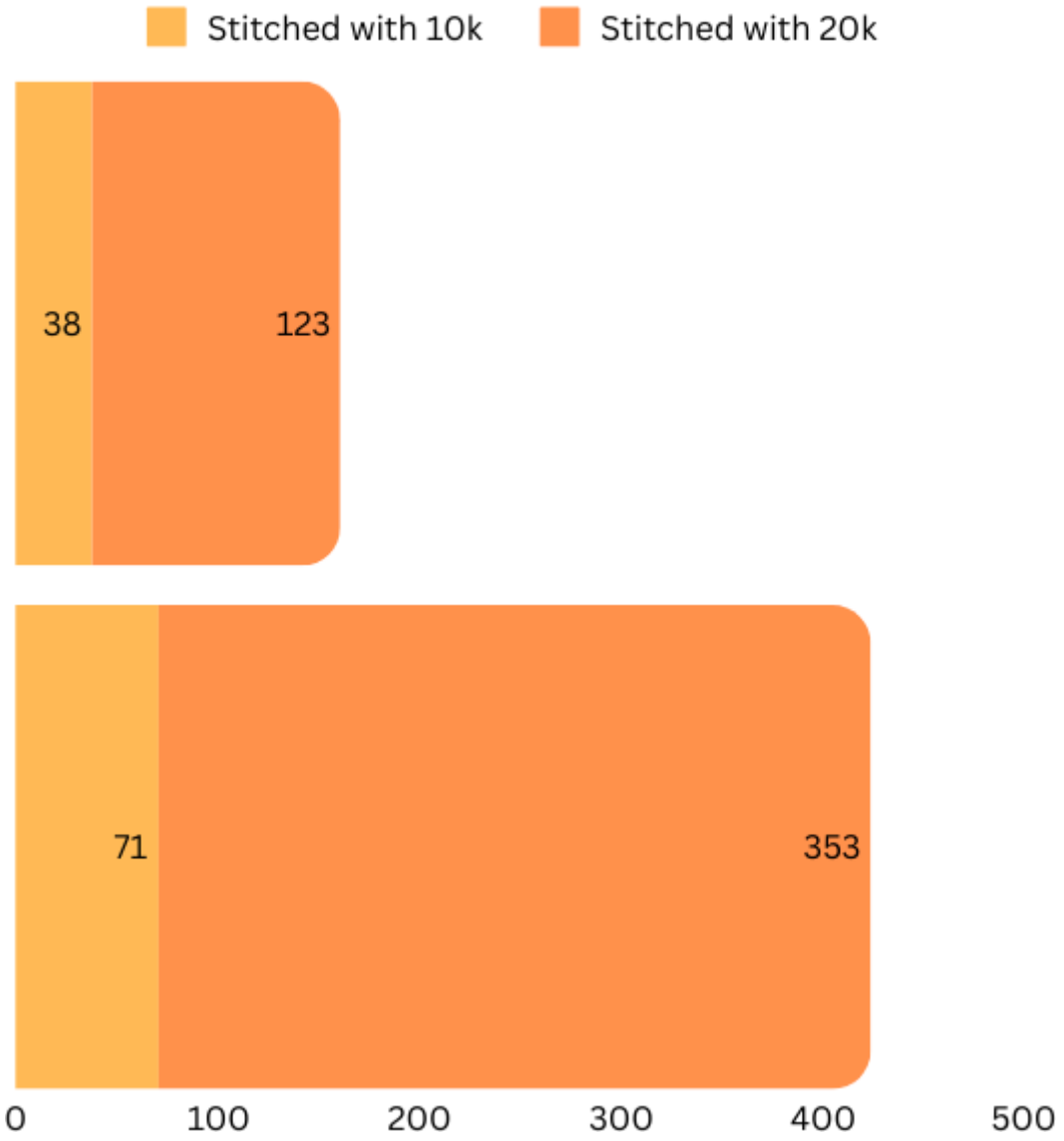
Όπως μπορείτε να παρατηρήσετε ότι καθώς αυξάνουμε τα threads τόσο μειώνεται και ο χρόνος. Παρόλα αυτά υπάρχουν δύο πολύ κρίσιμα σημεία.

1. Στις περιπτώσεις ,του basic vamana και του filtered vamana, βλέπουμε όταν είχαμε 1 thread το πρόγραμμα τελείωνε ελάχιστα πιο αργά από ότι αν έτρεχε χωρίς threads.

2. Παρατηρήσαμε ότι αφού προσθέσουμε τον 5ο thread , έπειτα ο χρόνος μειώνεται ελάχιστα (περίπου 1 δευτερόλεπτο).

Τέλος, πρέπει να σημειωθεί ότι στο πρόγραμμα vamana semi random τα threads είναι ιδιαίτερα χρήσιμα αφού υποδιπλασιάζουν τον χρόνο εκτέλεσης.

Καταληκτικά, θα θέλαμε να σχολιάσουμε ότι δυστυχώς δεν είχαμε τον χρόνο που θα επιθυμούσαμε για να αφιερώσουμε στην αύξηση των δεδομένων. Προσπαθήσαμε, ωστώσο, καταφέραμε να φτάσουμε μέχρι και τα 20 χιλιάδες δεδομένα και να πειραματιστούμε πάνω τους. Συγκεκριμένα παρακάτω επισυνάπτουμε ένα γράφημα στο όποιο τρέχουμε την filtered vamana με 10 χιλιάδες και 20 αντίστοιχα.



Το παραπάνω σχήμα μας παρέχει μία εικόνα της εκτέλεσης της filtered vamana για 10 και 20 χιλιαδες. Η πάνω εκτέλεση έγινε με την χρήση threads συγκεκριμένα 5, ενώ η κάτω πραγματοποιήθηκε χωρίς threads. Παρατηρούμε ότι ο χρόνος και στις δύο περιπτώσεις σχεδόν διπλασιάζεται κάτι που μας δείχνει πόσο χρήσιμα και αναγκαία είναι τα threads για την βελτιστοποίηση των αλγορίθμων μας. Πολύ σημαντικό είναι το γεγονός ότι η εκτέλεση των 10 χιλιάδων είναι δραστικά συντομότερη από ότι των 20 χιλιάδων.