Ανάπτυξη Λογισμικού για Αλγοριθμικά Προβλήματα– Χειμερινό '23

2η Άσκηση - Readme

Α) Τίτλος Προγράμματος : Γραφοθεωρητική αναζήτηση γειτόνων στη $C++$.
Το πρόγραμμα υλοποιεί την αναζήτηση πλησιέστερων γειτόνων μέσω *GNN* και *MRNG*. Για να το πετύχει αυτό κατασκευάζει τους αντίστοιχος γράφους βάσει του *dataset* και μετά αναζητά τον πλησιέστερο γείτονα ενός *query point* μέσω κατάλληλων αλγορίθμων αναζήτησης σε γράφο. Το πρόγραμμα βασίζεται στην πρώτη εργασία του μαθήματος Ανάπτυξης Λογισμικού για Αλγοριθμικά Προβλήματα. Αυτό που χρειάζεται να ξέρει κανείς για να κατανοήσει την παρούσα εργασία είναι ότι χρησιμοποιεί συναρτήσεις από τα αρχεία *ReadTrainData.cpp* και *ReadQueryData.cpp* για να μπορέσει να διαβάσει τα προφανή $inputs$. Ακόμη χρησιμοποιεί τον *LSH* σε κάποιες περιπτώσεις αντί για την εξαντλητική αναζήτηση για να επιταχυνθεί η κατασκευή του γράφου. Για να μπορέσει κανείς να καταλάβει αυτή την εργασία, αρκεί να ξέρει ότι ο *constructor* του *LSH* παίρνει σαν ορίσματα K, L, οι λεπτομέρειες για αυτές τις παραμέτρους έχουν συζητηθεί στην προηγούμενη εργασία. Αφού δημιουργηθεί μια μεταβλητή τύπου *LSH*, καλείται η συνάρτηση $train()$ για την εκπαίδευση στο *dataset*, και έπειτα μπορούμε να πάρουμε τους K *approximate* γείτονες μέσω της $KNN(k, QueryRepresentation, PositionOfQuery)$. Αξίζει να σημειωθεί εδώ ότι η συνάρτηση *KNN* του *LSH* είναι υπερφορτωμένη : δημιουργήσαμε σε αυτήν την εργασία μια ακόμη *KNN* που παίρνει τρια ορίσματα, όπως περιγράφηκαν παραπάνω, σε αντίθεση με την πρώτυπη *KNN* που έπαιρνε μόνο τα πρώτα δύο ορίσματα. Ο λόγος είναι ότι επειδή στην περίπτωση μας, το *Query*, ανήκει στο *Dataset*, δεν θέλουμε η *LSH* να επιστρέφει τον ίδιο το *Query* σαν κοντινότερο γείτονα. Θα μπορούσαμε απλά να ζητάμε $k+1$ γείτονες την φορά και να διαγράφουμε τον πρώτο, αλλά για λόγους ομοιομορφίας επιλέξαμε αυτή την στρατηγική (στην πραγματικότητα αρχικά υπήρχε και ο λόγος ότι θέλαμε αυτή η εκδοχή του *KNN* να επιτρέπει την παραλληλία για ταχύτερες εκτελέσεις, ωστόσο αποδείχθηκε πως απλά αν συμπεριλάβουμε το $-O3$ κατά την μεταγλώττιση, η εκτέλεση είναι ταχύτερη από την παράλληλη εκτέλεση και άρα δεν χρειαζόταν παραλληλία). Σε κάθε περίπτωση αυτό που χρειάζεται να γνωρίζει κανείς είναι ότι ο *KNN* επιστέφει έναν *vector* από *doubles* όπου η i θέση είναι η απόσταση του $i/2$ γείτονα από το *query* και η $i+1$ θέση η θέση του γείτονα αυτού στο *dataset*, δηλαδή ο πίνακα είναι μεγέθους $2k$. Η ιδέα πίσω από τον διαχωρισμό των πηγαίων αρχείων είναι ότι η *main* που βρίσκεται στο *graph_search.cpp* χρησιμοποιεί τα $interfaces$ που παρέχονται από τις κεφαλίδες *gnn.h* και *mrng.h* για να δημιουργήσει τους γράφους. Έπειτα χρησιμοποιεί τις *GNNS* και *GenericGraphSearch* με τα κατάλληλα ορίσματα (τα ορίσματα είναι βάσει των διαφανειών) για να λάβει τους N κοντινότερους γείτονες. Σε κάθε περίπτωση επιστρέφεται στην *main* ένας *vector* από $double*$ που αναπαριστά τους γείτονες όπου κάθε στοιχείο τύπου $double*$ είναι ένας πίνακα δύο θέσεων από *double*, όπου σε κάθε τέτοιο πίνακα το στοιχείο στην θέση *POSITION* δείχνει την θέση του γείτονα στο *dataset* και το στοιχείο στην θέση *DISTANCE* δείχνει την απόσταση από του γείτονα από το *Query* (τα *POSITION, DISTANCE*) έχουν οριστεί με $\#define$.
Β) Κατάλογος με τα αρχεία κώδικα (συμπεριλάβαμε και αυτά της πρώτης άσκησης, χωρίς το *clustering* σε περίπτωση που κάποιο σημείο του κώδικα δεν είναι καταναητό) :

- *graph_search.cpp* : Εδώ βρίσκεται η main συνάρτηση για τον αλγορίθμους GNN και MRNG .Ο κώδικας αρχικοποιεί με προεπιλεγμένες τιμές τις μεταβλητές k, l, N, R, E, m και στη συνέχεια πραγματοποιεί loop για τα ορίσματα της γραμμής εντολών (εάν υπάρχουν) και τα επεξεργάζεται.Αναζητά συγκεκριμένα flags όπως -d, -q, -k, -L, -o, -N, -E, -m και -R και προσαρμόζει τις αντίστοιχες μεταβλητές ανάλογα με βάση τις τιμές που παρέχονται στα ορίσματα της γραμμής εντολών. Στην συνέχεια καλείται η συνάρτηση ReadTrainData για το inputfile η οποία είναι υπεύθυνη για το διάβασμα των εικόνων byte-per-byte (έχει υλοποιηθεί και αναλυθεί απο την προηγούμενη εργασία). Ανάλογα το m που δόθηκε, (1 για GNN , 2 για MRNG), δημιουργούνται οι κατάλληλοι γράφοι μέσω των συναρτήσεων CreateGnn και CreateMrng αντίστοιχα. Διαβάζεται το query file μεσω της ReadQueryData και για κάθε query εκτελούνται οι συναρτήσεις GNNS και GenericGraphSearch (πάλι βαση του m που δόθηκε) και FindTrue ,καταγράφοντας τον χρόνο που απαιτείται και γράφοντας τα αποτελέσματα σε ένα αρχείο εξόδου outputfileName. Με βάση αυτά που επέστρεψαν οι δύο αυτές συναρτήσεις (αριθμός του N-οστού προσεγγιστικά πλησιέστερου γείτονα που βρέθηκε και την απόπσταση του απο το query , απόσταση από το αληθινά N-οστό πλησιέστερο γείτονα αντίστοιχα) γίνονται οι ανάλογες εκτύπωσεις στο αρχείο εξόδου. Τέλος εκτυπώνονται ακόμη ο μέσος χρόνος αναζήτησης των αλγορίθμων και το μέγιστο κλάσμα προσέγγισης ενώ το πρόγραμμα ρωτά τον χρήστη εάν θέλει να τερματίσει και αν όχι, ζητά από τον χρήστη ένα νέο queryfile . Αυτός ο βρόχος συνεχίζεται όσο ο χρήστης δεν εισάγει "y" για τερματισμό.

- *gnn.cpp* : Στην κεφαλίδα του αρχείου ορίζονται μερικές χρήσιμες μεταβλητές και διάφορες συναρτήσεις που είναι χρήσιμες για τον αλγόριθμο.Για την αρχικοποίηση και κατασκευή του γράφου χρησιμοποιούνται οι συναρτήσεις: createGraphGNN, CreateGnn και addEdge.Ο γράφος ουσιαστικά έχει υλοποιηθεί σαν ενα array απο pointers σε adjacency lists όπου το κάθε adjacency list είναι ένας πίνακας που αποθηκεύει την θέση του πλησιέστερου γείτονα του σημείου που θέλουμε.Αυτό γίνεται μέσω της συνάρτησης CreateGnn όπου για όλα τα images του dataset παίρνουμε του πλησιέστερους γείτονες μέσω της KNN του lsh και τους αποθηκεύουμε μεσω της addEdge στο adjacency list του εκάστοτε image.Η συνάρτηση GNNS λειτουργεί όπως περιγράφουν οι διαφάνειες του μαθήματος.Για αρχή επιλέγεται ενα τυχαίο σημείο (Yt) του γράφου και μέσω της συνάρτησης NearestNeighbor επιλέγεται το νεο Yt το οποίο είναι το κοντινότερο σημείο στο query από τους nearest neighbors του προηγούμενου Yt.Παράλληλα δημιουργείται και το S που περιέχει τους κοντινότερους γείτονες του query όπου ταξινομείται και επιστρέφεται. Τέλος, στο αρχείο βρίσκονται συναρτήσεις που ελευθερώνουν τις μεταβλητές που χρησιμοποιήσαμε κατα την διάρκεια του αλγορίθμου.(Ο αλγόριθμος GNNS επιστρέφει τους προσεγγιστικά πλησιέστερους γείτονες ενώ η συνάρτηση FindTrue τους αληθινά πλησιέστερους.)

  Σημείωση:
  Το T του αλγορίθμου των διαφανειών γίνεται defined = 50 (GREEDY STEPS) αφού δεν ζητείται να δίνεται στην γραμμή εντολών.

- *mrng.cpp* : Ο γράφος εδώ αναπρίσταται μέσω το πλήθος των κόμβων και μιας *adjList* που είναι ένας *vector* από *double∗* αναπαράσταση για τους γείτονες, όπου κάθε *double∗* είναι ένας είναι ένας πίνακας 2 στοιχείων : ένα για τη θέση του γείτονα στο *dataset*, την απόσταση του από τον κόμβο (χρησιμοποιείται για να μην επαναυπολογίζεται η απόσταση όταν κοιτάμε τις ακμές του εκάστοτε τριγώνου). Υπάρχουν τετριμμένες συναρτήσεις για την κατασκευή και την προσπέλαση του γράφου με τις συνήθεις λειτουργίες. Για την συνάρτηση *GenericGraphSearch* ακολουθείται η λογική των διαφανειών. Το σύνολο *R* των διαφανειών που αναπαριστά τους *candidates*, υλοποιείται μέσω ενός *vector* από *double∗* όπου το κάθε στοιχείο ακολουθεί την λογική του *MRNG* γράφου όπως περιγράφηκε παραπάνω συν το οτι έχει προστεθεί ένα στοιχείο στον πίνακα από *double∗* που είναι το *CHECKED* που μας βοηθάει στο *markings as checked* των διαφανειών. Έπειτα απλά ο αλγόριθμος είναι αυτός των διαφανειών, μπορεί κανείς να διαβάσει τον κώδικα με την βοήθεια των σχολίων. Για την κατασκευή του γράφου *MRNG* χρησιμοποιείται η συνάρτηση *CreateMrng*, όπου για τους πλησιέστερους γείτονες δεν χρησιμοποιούμε *exhaustive search* αλλά *LSH*, αυτό γιατί η ταχύτητα αυξάνεται σημαντικά και η απόδοση παραμένει αρκετά καλή (ο κ. Εμίρης σε διάλεξη του είχε προτείνει αυτή την τροποποίηση του αλγορίθμου). Η υλοποίηση δεν παρεκκλίνει καθόλου από τον αλγόριθμο των διαφανειών πέραν του παραπάνω. Για τον υπολογισμό των ακμών του τριγώνου χρησιμοποιούνται τα *DISTANCES* που έχουν αποθηκευτεί στον γράφο και στο σύνολο *R*. Κατά τα άλλα, μια ανάγνωση των σχολιών ξεκαθαρίζει το τοπίο. Υπάρχει ακόμη μια συνάρτηση *FindNavigating* που είναι τεττριμένη όπως και η συνάρτηση διαγραφής του γράφου.

- *StoreTrainData.cpp* : Στην κεφαλίδα του αρχείο υπάρχουν δύο κλάσεις : η κλάση για τις εικόνες και η κλάση που είναι υπεύθυνη για την αποθήκευση των εικόνων που ανήκουν στο training set . Η αναπαράσταση τις εικόνας γίνεται μέσω ενός πίνακα από bytes όπου το κάθε pixel είναι και ένα byte , ενώ υπάρχουν και δύο έξτρα πεδία checked, cluster με τους αντίστοιχους setters, getters που θα μας χρησιμεύσουν στην αναζήτηση και την συσταδοποίησης. Η κάθε εικόνα δημιουργείται byte per byte μέσω της συνάρτησης Insert , όπου κάθε φορά εισάγει ένα byte στον πίνακα των bytes της αναπαράστασης, και τον μεγαλώνει κατά 1. Όσον αφορά την αποθήκευση των εικόνων χρησιμοποιείται η κλάση TrainStore όπου στην ουσία οι εικόνες αποθηκεύονται μέσω ενός πίνακα δεικτών σε εικόνες. Η λογική είναι ότι οι εικόνες εισάγονται μια προς μια μέσω της συνάρτησης Insert . Έπειτα, μπορεί να γίνει προσπέλαση της κάθε εικόνας μέσω ενός integer index , όπου αντιστοιχεί στην θέση της εικόνας κατά την εισαγωγή ξεκινώντας την αρίθμηση από το 0. Δηλαδή, η πρώτη εικόνα που θα εισαχθεί θα έχει *index* 0, η δεύτερη 1 κ.ο.κ. Στην πραγματικότητα είναι η θέση στον πίνακα των δεικτών, καθώς αυτός μεγαλώνει με την εισαγωγή της κάθε εικόνας. Οπότε μέσω αυτού του index , παρέχεται πρόσβαση στους setters, getters της κλάσης της εικόνας.Σημειώνουμε εδώ, ότι ο χρήστης δεν έχει πρόσβαση σε αυτές τις κλάσεις, το interface που του παρέχεται βρίσκεται στην RedTrainData.cpp .

- *ReadTrainData.cpp* : Εδώ βρίσκεται η υλοποίηση της συνάρτησης που είναι υπεύθυνη για το άνοιγμα του κατάλληλου αρχείου και διάβασμα του train set, την δημιουργία της κλάσης που περιγράφηκε παραπάνω για την αποθήκευση του, την εισαγωγή των εικόνων στην εν λόγων κλάση και την παροχή του interface προς τον χρήστη. Για το διάβασμα και την αποθήκευση, χρησιμοποιείται η συνάρτηση ReadTrainData που σαν όρισμα παίρνει το μονοπάτι, ως string, στο αρχείο που περιλαμβάνει το dataset . Αφού αρχικά διαβάσει με κατάλληλη τρόπο την κεφαλίδα του αρχείο, δηλαδή τα πρώτα, 16 bytes και τα αποθηκεύσει σε κατάλληλες μεταβλητές όπως αυτές ορίζονται στο MNIST manual , συνεχίζει με το διάβασμα των εικόνων. Για το διάβασμα των εικόνων αρχικά δημιουργείται μια νέα εικόνα, και έπειτα byte per byte διαβάζονται τα pixels της και αποθηκεύονται καταλλήλως μέσω συνάρτησης που παρέχει η κλάση της εικόνας, το σύνολο των pixels προκύπτει από τον πολλαπλασιασμό $columns \cdot rows$, η διαδικασία αυτή επαναλμβάνεται τόσες φορές όσο και το πλήθος των εικόνων στο αρχείο. Το interface με κάθε εικόνα από τον χρήστη γίνεται μέσω του index που υπολογίζεται όπως περιγράφηκε παραπάνω, από εδώ και στο εξής πρόσβαση και οιονεί αναφορά σε κάθε εικόνα γίνεται μέσω index το οποίο παραμένει σταθερό καθ' όλη την εκτέλεση του προγράμματος.

- *ReadQueryData.cpp* : Ίδια λογική με το ReadTrainData.cpp μόνο που αφορά του test set, και δεν χρειάζεται interface για τους setters,getters των πεδίων checked, cluster των αποθηκευμένων εικόνων.

- *StoreQueryData.cpp* : Ίδια λογική με το StoreTrainData.cpp

- *hFunc.cpp* : Εδώ βρίσκεται η υλοποίηση της συνάρτησης H του LSH καθώς και η συνάρτηση που υπολογίζει την Ευκλείδεια Απόσταση μεταξύ δύο διανυσμάτων byte . Απλά τα πράγματα , ακολουθούνται ακριβώς οι ορισμοί των διαφανειών. Η συνάρτηση H παίρνει σαν όρισμα το w και το μέγεθος των διανυσμάτων. Σημειώνουμε εδώ, ότι επειδή θέλουμε οι επιστρεφόμενες τιμές της g να είναι θετικές, γίνεται μια κανονικοποίηση στο εσωτερικό γινόμενο, για να είναι πάντα θετικό, μέσω τετραγωνισμού. Ορίζεται για κάθε H ένας τυχαίος πίνακας v και μέσω της συνάρτησης operator παίρνεις το αποτέλεσμα του εσωτερικού γινομένου με αυτόν τον πίανακα.

- *lsh.cpp* : Στην κεφαλίδα του αρχείο υπάρχουν δύο κλάσεις LSH και Hashtable .Η μία αφορά τους πίνακες κατακερματισμού ενώ η άλλη είναι υπεύθυνη για την κατασκευή των πινάκων κατακερματισμού και αποθήκευσης των σημείων σε buckets αλλά και γενικά για όλη την διαδικασία εύρεσης πλησιέστερων γειτόνων. Κατά την δημιουργία ενός αντικειμένου LSH , κατασκευάζεται ένας πίνακας L θέσεων από πίνακες κατακερματισμού (class Hashbucket). Κάθε πίνακας κατακερματισμού δημιουργεί με την σειρά του ένα πίνακα από buckets (class Bucket) και μία g function (class gFunction) η οποία κατασκευάζει K h functions και τυχαία r και μέσω της συνάρτησης FindPosition παίρνεις το αποτέλεσμα της g όπως ακριβώς περιγράφεται στις διαφάνειες του μαθήματος. Ουσιαστικά κάθε hashtable είναι ένας πίνακας απο pointers σε linked list , οπότε καθε bucket λειτουργεί σαν ένα node αυτής της λίστας όπου περιέχει ένα σήμειο. Με την κλήση της συνάρτησης Train , χρησιμοποιείται η συνάρτηση GetRepresentation όπου επιστρέφει τα pixels-bytes κάθε εικόνας.Έπειτα κάθε σήμειο αποθηκεύεται για κάθε hastable σε κάποιο bucket ανάλογα με την g function . Η συνάρτηση KNN επιστέφει ένα πίνακα ο οποίος περιέχει τους κατα προσέγγιση πλησιέστερους γείτονες του query και τις αποστάσεις τους απο αυτο. Αυτό επιδιώκεται με την κλήση μίας συνάρτησης NearestNeighbour της κλάσης LSH έπειτα της κλάσης Hashtables κ.ο.κ ώσπου να φτάσει σε κάποιο bucket και μέσω της ευκλείδιας απόστασης να πάρει το επιθυμητό αποτέλεσμα. Για κάθε γείτονα που επιλέγεται υπάρχει η συνάρτηση SetChecked ώστε να μην λαμβάνεται ο γείτονας αυτός υπόψη στις επόμενες επαναλήψεις . Με παρόμοιο τρόπο έχουν κατασκευαστεί και οι συναρτήσεις AKNN και RangeSearch.

- *lshmain.cpp* : Εδώ βρίσκεται η main συνάρτηση για τον αλγόριθμο LSH.Ο κώδικας αρχικοποιεί με προεπιλεγμένες τιμές τις μεταβλητές K, L, N, R και στη συνέχεια πραγματοποιεί loop για τα ορίσματα της γραμμής εντολών (εάν υπάρχουν) και τα επεξεργάζεται. Αναζητά συγκεκριμένα flags όπως -d, -q, -k, -L, -o, -N και -R και προσαρμόζει τις αντίστοιχες μεταβλητές ανάλογα με βάση τις τιμές που παρέχονται στα ορίσματα της γραμμής εντολών. Στην συνέχεια καλείται η συνάρτηση ReadTrainData για το inputfile και η συνάρτηση ReadQueryData για το queryfile όπου αναλύθηκαν παραπάνω. Έπειτα δημιουργείται ενα αντικείμενο LSH και εκτελείται η συνάρτηση Train όπως αναλύθηκε παραπανω. Για κάθε query εκτελούνται οι συναρτήσεις KNN και AccurateKNN καταγράφοντας τον χρόνο που απαιτείται και γράφοντας

τα αποτελέσματα σε ένα αρχείο εξόδου outputfileName. Με βάση αυτά που επέστρεψαν οι δύο αυτές συναρτήσεις (αριθμός του N-οστού προσεγγιστικά πλησιέστερου γείτονα που βρέθηκε και την απόσταση του απο το query , απόσταση από το αληθινά N-οστό πλησιέστερο γείτονα αντίστοιχα) γίνονται οι ανάλογες εκτύπωσεις στο αρχείο εξόδου. Καλείται η συνάρτηση RangeSearch βρίσκοντας και εκτυπόνωντας τους γείτονες εντός μιας ακτίνας R . Τέλος το πρόγραμμα ρωτά τον χρήστη εάν θέλει να τερματίσει το πρόγραμμα και αν όχι, ζητά από τον χρήστη ένα νέο queryfile . Αυτός ο βρόχος συνεχίζεται όσο ο χρήστης δεν εισάγει "y" για τερματισμό.

- *RandomProjection.cpp* : Εδώ βρίσκεται η υλοποίηση της προβολής στον υπερκύβο, το interface προς τον χρήστη είναι το ίδιο με αυτό του LSH , δηλαδή πρώτα αρχικοποιεί ένα αντικείμενο τύπου RandomProjection , και στην συνέχεια μπορεί να καλέσει τις συναρτήσεις που ζητούνται βάζοντας ως input τα queries . Τα ορίσματα για τον constructor είναι τα k, m, probes όπως περιγράφονται στην εκφώνηση. Επειδή ο υπερκύβος περιλαμβάνει ένα σύνολο F συναρτήσεων που η τιμή τους αποφασίζεται τυχαία αλλά παραμένει ίδια για όλο το πρόγραμμα, έχουμε δημιουργήσει μια κλάση FValues , η οποία έχει δύο πεδία, ένα για το input ένα για την τιμή του κάθε input (f(5) = 3)καθώς και τους κατάλληλους setters, getters . Με την σειρά της, η κλάση FunctionF , κρατά μια συνδεδεμένη λίστα από FValues , έτσι καθώς δέχεται σαν όρισμα ένα point , παίρνει την τιμή από την κατάλληλη LSH , και έπειτα ελέγχει αν αυτή η τιμή βρίσκεται σε κάποιο κόμβο της συνδεδεμένης λίστας, αν ναι επιστρέφει αν όχι δημιουργεί νέο κόμβο. Όλα τα υπόλοιπα είναι απλές υλοποιήσεις αυτών που ζητά η εργασία στην λογική του LSH
- *cube.cpp* : Τίποτα δύσκολο εδώ, ό,τι ζητάει η εκφώνηση πάλι στην λογική της LSH main.

Γ) Για την εκτέλεση του μέρους της εργασίας αυτής, αρκεί να γίνει χρήση της *make graph* ενώ τα *.o* και εκτέλέσιμα διαγράφονται μέσω της *make clean*.
Δ) Το πρόγραμμα εκτελείται όπως ζητείται από την εκφώνηση επιτρέποντας κάποια $flags$ να λείπουν. Για να τρέξει κανείς το πρόγραμμα με τις $default$ τιμές, ο πιο απλός τρόπος είναι μέσω της :
./graph -d dataset.dat -o test.txt -q query.dat -m 1
για τον GNN.
Ε) **Στοιχεία φοιτητών**

- Απόστολος Κουκουβίνης, ΑΜ : 1115202000098
- Γιώργος Μητρόπουλος, ΑΜ : 1115202000128

Ακολουθεί ένας πίνακας με κάποιες εκτελέσεις για διάφορες υπερπαραμέτρους, και σχολιασμός τους, αυτό το μέρος αφορά το ερώτημα Γ της εργασίας.

Index	Method	k	L	M	Probes	E	R	I	MAF	Time
1	LSH	4	5	-	-	-	-	-	1.50	0.0001426
2	LSH	4	10	-	-	-	-	-	1.47	0.00026
3	LSH	10	5	-	-	-	-	-	1.48	0.000130
4	LSH	10	10	-	-	-	-	-	1.45	0.000300
5	Hypercube	14	-	10	2	-	-	-	2.8	0.00154
6	Hypercube	14	-	5	2	-	-	-	2.3	0.00090
7	Hypercube	14	-	20	2	-	-	-	2.12500	0.0028
8	Hypercube	10	-	10	2	-	-	-	3	0.0009
9	Hypercube	10	-	5	2	-	-	-	3.1	0.0006
10	Hypercube	10	-	20	2	-	-	-	2.5	0.0019
11	Hypercube	14	-	10	5	-	-	-	2.7	0.0015
12	Hypercube	14	-	5	5	-	-	-	3	0.00095
13	Hypercube	14	-	20	5	-	-	-	2.1	0.0027
14	Hypercube	10	-	10	5	-	-	-	2.6	0.014
15	Hypercube	10	-	5	5	-	-	-	2.8	0.001
16	Hypercube	10	-	20	5	-	-	-	2.2	0.00185
17	Hypercube	10	-	200	2	-	-	-	1.35	0.014
18	Hypercube	6	-	100	2	-	-	-	1.45	0.00531
19	GNN	50	-	-	-	5	1	-	1.2311	0.0001418
20	GNN	50	-	-	-	15	1	-	1.07007	0.0004172
21	GNN	50	-	-	-	30	1	-	1.00983	0.0009664
22	GNN	100	-	-	-	30	1	-	1	0.0009868
23	GNN	50	-	-	-	50	1	-	1.10505	0.0017812
24	GNN	100	-	-	-	50	1	-	1	0.0018506
25	GNN	50	-	-	-	30	3	-	1	0.0033204
26	GNN	100	-	-	-	30	3	-	1	0.0034202
27	GNN	50	-	-	-	50	3	-	1	0.0052134
28	GNN	100	-	-	-	50	3	-	1	0.0064774
29	MRNG	-	-	-	-	-	-	20	1.39166	0.0001126
30	MRNG	-	-	-	-	-	-	50	1.24378	0.0004288
31	MRNG	-	-	-	-	-	-	100	1.00521	0.0015288
32	MRNG	-	-	-	-	-	-	400	1	0.0185454

Πίνακας 1: Trials

Για το GNN παρατηρούμε πως καθώς αυξάνεται ο αριθμός των επεκτάσεων και των τυχαίων επανακινήσεων αυξάνεται ο χρόνος αναζήτησης πράγμα λογικό αφού αυξάνεται ο αριθμός των επαναλήψεων. Παρατηρούμε επίσης πως για πολύ μικρά E το κλάσμα προσέγγισης δεν είναι 1 που σημαίνει πως υπάρχει απόκλιση του προσεγγιστικά πλησιέστερου γείτονα με τον αληθινά πλησιέστερο.Ωστόσο αυξάνοντας λιγο το E δηλαδή τον αριθμό των επεκτάσεων βλέπουμε πως προσεγγίζεται το 1 πράγμα που σημαίνει οτι ο αλγόριθμος βρίσκει ακριβώς τον πλησιέστερο γειτονα σε πολλές περιπτώσεις. Η μεταβολή της μεταβλητής k παρατηρούμε πως δεν επηρεάζει σημαντικά κάπου παρα μόνο στην κατασκευή του γράφου εφόσον ο αλγόριθμος επιλέγει κάθε φορά τους πρώτους E γείτονες.

Για το MRNG παρατηρούμε όπως ειναι λογικό πως καθώς αυξάνεται ο αριθμός υποψηφίων μειώνεται το κλάσμα προσέγγισης προσεγγίζοντας σταδιακά το 1 αφού ο αλγόριθμος κοιτάει μεγαλύτερο αριθμό υποψηφίων και επιλέγει τον κοντινότερο, ωστόσο αυξάνεται ο χρόνος αναζήτησης.

Συγκρίνοντας τους αλγορίθμους MRNG και GNN παρατηρούμε πως ο GNN είναι πιο αποδοτικός αν θέλουμε να έχουμε μεγαλύτερη ακρίβεια στην λύση μας ενώ Ο $MRNG$ είναι αρκετά ταχύτερος, που θα μας ήταν πιο

χρήσιμος αν θέλαμε να έχουμε γρήγορες εκτελέσεις. Για τον GNN μάλλον οι καλύτεροι παράμετροι είναι το $k = 100, E = 30, R = 1$, και για τον $MRNG$ $I = 20$ αφού στην πρώτη περίπτωση, ο GNN επιτυγχάνει στο να βρίσκει αρκετά συχνά τον κοντινότερο γείτονα παραμένοντας αρκετά γρήγορος, ενώ ο $MRNG$ για αυτή την παράμετρο είναι ταχύτατος ενώ μας παρέχει ένα άνω φράγμα στο MAF, μπορούμε να πούμε ότι πειραματικά ισχύει $MAF < 1.7$, αφού στις εκτελέσεις μας σπάνια είχαμε $MAF > 1.55$. Παρατηρούμε ακόμη ότι ο υπερκύβος, ενώ είναι αρκετά γρήγορος υστερεί στο MAF, παράγει καλά αποτελέσματα μόνο αν αυξήσουμε πολύ το M, δηλαδή κοιτάει πολλούς υποψήφιους ή αν μειώσουμε αρκετά το k δηλαδή προβάλλεται σε μικρή διάσταση. Σε κάθε περίπτωση θα προτιμούσαμε τους GNN και $MRNG$ έναντι του υπερκύβου. Ο LSH παραμένει αρκετά ανταγωνιστικός αφού παρέχει ένα καλό άνω φράγμα στο MAF, κοντά στο 1.6 με αρκετά καλή ταχύτητα. Ωστόσο, οι αλγόριθμοι αναζήτησης σε γράφο παραμένουν καλύτεροι, αφού όπως βλέπουμε από τον πίνακα, για τις πρώτες τιμές που δώσαμε στις παραμέτρους των δύο αυτών αλγορίθμων, πήραμε καλύτερο MAF, σε πολύ μικρότερο χρόνο. Καταλήγουμε ότι οι γραφειοθεωρητικοί αλγόριθμοι είναι οι αποτελεσματικότεροι εξ όσων υλοποιήσαμε. Αξίζει να σημειωθεί, ότι οι συγκρίσεις έγιναν για μικρό *dataset*, η δύναμη αυτών των αλγορίθμων έναντι του *exhaustive search* αναδεικνύεται για μεγαλύτερα *datasets*. Ενδεικτικά για *dataset* με 20.000 εικόνες παίρνουμε τα εξής αποτελέσματα για τις πρώτες τιμές των παραμέτρων του GNN στον πίνακα
$tapproximate = 0.0013082, ttrue = 0.01174, maf = 1.3863$. Δηλαδή η αναζήτηση με GNN είναι δέκα φορές ταχύτερη και το maf αρκετά καλό. Με τον $MRNG$ με $L = 300$ παίρνουμε $tapproximate = 0.0097, ttrue = 0.011$ και το maf τείνει στο 1, και αρκετές φορές μάλιστα είναι και 1 ακριβώς. Δηλαδή ο $MRNG$ έχει σχεδόν την απόδοση του *exhaustive* όσον αφορά την ακρίβεια στους γείτονες και είναι πάνω από 10 φορές ταχύτερος.