{ Ελευθερία Εκατομμάτη sdi2000048, Θεοδώρα Αρχοντάκη sdi2000014 }

Για το LSH:

Random.h:

Περιέχει τους ορισμούς δύο συναρτήσεων “Random_Num_ND” και “Random_Num_Uniform”.

Random.cpp:

Έχει την υλοποίηση των δύο παραπάνω συναρτήσεων. Η “Random_Num_ND” δημιουργεί έναν τυχαίο αριθμό από μια κανονική κατανομή με μέση τιμή 0 και τυπική απόκλιση 1 και έπειτα στρογγυλοποιεί το αποτέλεσμα στον πλησιέστερο ακέραιο. Η “Random_Num_Uniform” δημιουργεί έναν τυχαίο αριθμό από μια ομοιόμορφη κατανομή.

Image_vector.h:

Έχουμε ένα struct που αναπαριστά την εικόνα MNIST και αποτελείται από έναν πίνακα διάστασης 784 και σε αυτό θα αποθηκεύουμε τα pixels ώστε να φτιάξουμε το διάνυσμα της εικόνας.

Έχουμε ορίσει έξι ακεραίους ως extern οι οποίοι είναι ο αριθμός των εικόνων που έχει το input αρχείο, το k (αριθμός των g-functions), το L (αριθμός των hash tables), το w το «παράθυρο» που χρησιμοποιούμε για την δημιουργία των συναρτήσεων, το N (αριθμός των πλησιέστερων γειτόνων) και R (ακτίνα αναζήτησης).

Έχουμε δύο vectors τύπου struct MNIST_Image. Στο ένα αποθηκεύουμε «Images» τα διανύσματα από τις εικόνες που πήραμε από το αρχείο εισόδου (πρακτικά θα περιέχει διανύσματα (με 784 στοιχεία) όσα ο αριθμός των εικόνων που πήραμε για είσοδο). Και ο άλλος vector «Q_Image» θα περιέχει ένα διάνυσμα με 784 το οποίο θα αντιστοιχεί στο διάνυσμα της εικόνας-query. Έχουμε και ένα string το όποιο είναι το όνομα του output αρχείου σε περίπτωση που αυτό δεν δίνεται από τον χρήστη. Και έχουμε και το output αρχείο.

Έχουμε και τον ορισμό δύο συναρτήσεων “Input”, “create_image”.

Create_image_vector.cpp:

Αρχικοποιούμε με default τιμές το k,L,N,R οπότε αν δεν δοθούν τιμές από την γραμμή εντολών θα χρησιμοποιηθούν οι default τιμές.

Η συνάρτηση “create_image” παίρνει σαν όρισμα μία αναφορά σε string το οποίο είναι το όνομα ενός αρχείου και επιστρέφει ένα vector τύπου MNIST_Image. Χρησιμοποιούμε αυτή τη συνάρτηση για να μετατρέψουμε από το binary αρχείο τις πληροφορίες και να φτιάξουμε το διάνυσμα για την κάθε εικόνα. Αρχικά ανοίγουμε το αρχείο και ελέγχουμε αν όντως έχει ανοίξει. Διαβάζουμε από το αρχείο τα πρώτα 16bytes που είναι γενικές πληροφορίες(magic_number, number_of_images, number_rows, number_columns). Διαβάζουμε από το αρχείο μία μία τις εικόνες και για κάθε εικόνα ένα ένα το pixel και το τοποθετούμε στο τέλος του vector. Κλείνουμε το αρχείο και επιστρέφουμε τον vector με τα διανύσματα των εικόνων.

Η συνάρτηση “Input” παίρνει σαν όρισμα «int argc, char* argv[]» Ελέγχουμε τα ορίσματα που παίρνουμε από την γραμμή εντολών. Αν ο χρήστης τρέξει το πρόγραμμα με ένα μόνο argument τότε το πρόγραμμα του ζητά τα αρχεία και τα k,L,N,R. Αν ο χρήστης δώσει arguments θα πρέπει να δοθούν με τον παρακάτω τρόπο.

Στις περιπτώσεις που τα k, L, N, R δεν δίνονται θα έχουν τις default τιμές τους.

$ ./lsh –d <input_file> –q <query_file> -k <int> -L <int> -o <output_file> -N <int> -R <radius>

Επίσης αν ο χρήστης δεν δώσει στη γραμμή εντολών το όνομα του αρχείου εξόδου τότε το πρόγραμμα παίρνει σαν default τιμή του ονόματος του αρχείου το «output.txt»

Hash.h

Έχουμε ένα vector τύπου double που είναι το διάνυσμα t και ένα διπλό vector v που χρησιμοποιούμε για την δημιουργία των h,g συναρτήσεων. Έχουμε ένα 2-dimensional vector HashFunction_Amplified που φτιάχνουμε τις hash-functions. Έχουμε και ένα 3-dimensional vector που αποθηκεύουμε τα hash_tables. Έχουμε δύο vectors τύπου int τον Points_Range και τον N_Points, στον πρώτο αποθηκεύουμε τα σημεία που βρίσκει ο αλγόριθμος κάνοντας το Range Search και στον δεύτερο αποθηκεύουμε τα σημεία που βρίσκουμε εκτελώντας τον αλγόριθμο για την εύρεση N πλησιέστερων γειτόνων.

Ορίζουμε στο .h αρχείο τις παρακάτω συναρτήσεις: «Hash_initialize» αρχικοποιούμε τα hash tables, «HashFunctions_initialize» αρχικοποιούμε τις hash-functions, «Euclidian_hash» που υπολογίζουμε τις h-functions, «Euclidian_dist» υπολογίζουμε μέσω Ευκλείδειας μετρικής την απόσταση μεταξύ δύο διανυσμάτων διάστασης 784. Η «Hash_clear» που πρακτικά απελευθερώνει μνήμη από τα Hash Tables, η «Euclidian_Nearest_N» βρίσκει τον πιο κοντινό γείτονα με βάση την ευκλείδεια μετρική. Η «Query_Hash» βρίσκει τους κοντινούς γείτονες ενός συγκεκριμένου query. Η «LSH_Nearest_N» βρίσκει τον πλησιέστερο γείτονα ενός query με τον LSH αλγόριθμο. Η «LSH_RangeSearch» βρίσκει τους πλησιέστερους γείτονες που βρίσκονται σε εύρος ακτίνας R από το query. Η «N_Nearestneighbors_LSH» βρίσκει τους N πλησιέστερους γείτονες ενός query με τον LSH αλγόριθμο. Και η «LSH_OutFile» καλεί τις τέσσερις προηγούμενες συναρτήσεις και κάνει τις κατάλληλες εκτυπώσεις στο αρχείο εξόδου και υπολογίζει και τον χρόνο εκτέλεσης του αλγορίθμου. Η συνάρτηση «Euclidian_NN» που υπολογίζει με τον exhaustive αλγόριθμο τους N κοντινότερους γείτονες από το query.

Hash.cpp

Για την «Hash_initialize» αρχικοποιούμε τους πίνακες κατακερματισμού που είναι μια τρισδιάστατη δομή. Έχουμε L πίνακες το κάθε slot έχει μέγεθος NUMBER_IMAGES/8 και η τρίτη διάσταση περιγράφει πόσες εικόνες μπορούμε να έχουμε σε κάθε slot κάθε πίνακα.

Για την «HashFunctions_initialize» αρχικοποιούμε τον δυσδιάστατο vector v η μία διάσταση θα είναι ο αριθμός των h-functions που θα έχουμε και η άλλη διάσταση θα είναι ίση με τη διάσταση του διανύσματος της εικόνας άρα 784. Σβήνουμε με την συνάρτηση clear τα «σκουπίδια» που έχουν δημιουργηθεί στον v και μετά καλούμε την συνάρτηση που φτιάχνει τυχαίους αριθμούς με βάση την κατανομή Gauss και γεμίζουμε τον v. Ύστερα αρχικοποιούμε τις συναρτήσεις g. Καλούμε την resize, θα έχουμε L συναρτήσεις g και η κάθε g θα έχει μέγεθος k. Στη συνέχεια θα ορίσουμε για κάθε g την σειρά με την οποία καλεί της h_i functions.

Φτιάχνουμε έναν τυχαίο αριθμό f_number που θα είναι από 0-(k-1) και κοιτάμε για κάθε g αν υπάρχει αυτός ο αριθμός αν υπάρχει τον κάνουμε push και έτσι κάθε συνάρτηση g θα έχει με τυχαία σειρά τις ίδιες συναρτήσεις h_i (I μικρότερο ίσο του k). Τέλος αρχικοποιούμε τον vector t με τυχαίους αριθμούς που βασίζονται στην Uniform-distribution.

Για την «Euclidian_Hash» παίρνει σαν όρισμα το number_image που πρακτικά είναι ο αριθμός της εικόνας (δηλαδή αν το number_image είναι τρία τότε η συνάρτηση θα πάει στον πίνακα Images που έχουμε αποθηκεύσει όλα τα διανύσματα των εικόνων και θα πάρει το τρίτο διάνυσμα δηλαδή το Images[2]), το L και το k. Λέμε για όλες τις g συναρτήσεις και για κάθε h συνάρτηση της κάθε g υπολογίζουμε το εσωτερικό γινόμενο ανάμεσα στο διάνυσμα της εικόνας και του διανύσματος v. Αποθηκεύουμε το αποτέλεσμα σε μία μεταβλητή Res. Αφού έχουμε υπολογίσει το εσωτερικό γινόμενο σε ολόκληρη την διάσταση του διανύσματος (784) τότε προσθέτουμε το διάνυσμα t. Και τέλος υπολογίζουμε και αποθηκεύουμε στο Res το τελικό αποτέλεσμα που είναι το κάτω ακέραιο μέρος της διαίρεσης του Res με το w (window). Αφού έχουμε βρει τα h_i(p)*r_i για μία συγκεκριμένη g-function τα προσθέτουμε και έχουμε ένα άθροισμα Sum_2 και υπολογίζουμε το Modulo ανάμεσα σε αυτό το Sum_2 και ενός μεγάλου ακεραίου (M = 2^32-5) και αυτό το αποτέλεσμα (Pos) το παίρνουμε και ξανακάνουμε Modulo με το μέγεθος του Table Size το οποίο είναι NUMBER_IMAGES/8. Και στο τέλος για όλες τις g-functions θα κάνουμε push στη θέση Pos τον αριθμό της εικόνας. Οπότε πρακτικά όταν κληθεί η συνάρτηση για μία συγκεκριμένη εικόνα η δομή Hash_Tables θα έχει στην θέση Pos για όλες τις g-functions τον αριθμό της εικόνας.

Για την «Euclidian_dist» δεχόμαστε σαν ορίσματα τις συνάρτησης δύο ακεραίους, ο πρώτος είναι ο αριθμός της εικόνας που θα διαλέξουμε από το σύνολο των εικόνων και αντίστοιχα ο δεύτερος για το query. Και μετά απλά εφαρμόζουμε τον τύπο τις αποστάσεις για τις 784 διαστάσεις.

Για την «Hash_clear» για τα Hash tables έχουμε εφαρμόσει αρχικά την συνάρτηση resize αυτό σημαίνει ότι μπορεί να έχουν δημιουργηθεί στοιχεία άχρηστα που θα έχουν τιμή μηδέν. Με την συνάρτηση αυτή σβήνουμε πρακτικά από τα hash tables τα άχρηστα μηδενικά.

Για την «Euclidian_Nearest_NN» παίρνουμε ένα query και σύμφωνα με αυτό βρίσκουμε ποια εικόνα είναι πιο «κοντά» σε αυτό. Υπολογίζουμε σε μία λούπα όλες τις αποστάσεις ανάμεσα στο query και τα διανύσματα των εικόνων και στο τέλος κρατάμε την εικόνα που έχει την μικρότερη απόσταση από το query. Και εκτυπώνουμε τις κατάλληλες πληροφορίες στο αρχείο εξόδου.

Για την «Query_Hash» η συνάρτηση αυτή είναι παρόμοια με την Euclidian_Hash. Η συνάρτηση αυτή επιστρέφει την θέση του κοντινότερου γείτονα ενός συγκεκριμένου query. Για όλες τις h συναρτήσεις ξανά υπολογίζουμε το εσωτερικό γινόμενο του διανύσματος της εικόνας query και του διανύσματος v και υπολογίζουμε όπως και πριν το Sum_2 και κάνοντας τα κατάλληλα modulo υπολογίζουμε την θέση.

Για την «LSH_Nearest_N» βρίσκουμε τον πλησιέστερο γείτονα ενός Query με τον αλγόριθμο LSH. Η συνάρτηση έχει για ορίσματα τον αριθμό του Q_number και το L, k. Αρχικοποιούμε την μικρότερη απόσταση με το άπειρο. Για κάθε hash table αρχικά υπολογίζουμε την θέση του πλησιέστερου γείτονα του query με την Query_Hash.

Και ψάχνουμε ολόκληρο το hash table για να βρούμε το διάνυσμα της εικόνας που θα έχει την μικρότερη απόσταση από το query. Έχουμε και μία ακέραια μεταβλητή η οποία αρχικοποιείται όταν αλλάζουμε bucket και όταν αυτή η μεταβλητή μηδενιστεί στη λούπα καθώς είμαστε σε κάποιο bucket σημαίνει πως ο αλγόριθμος έχει βρει το βέλτιστο διάνυσμα με την μικρότερη απόσταση από το query άρα σταματάμε να ψάχνουμε στο συγκεκριμένο bucket.

Για την «LSH_RangeSearch» βρίσκει τους κοντινότερους γείτονες που είναι μέσα σε μία εμβέλεια ακτίνας R από το query. Για κάθε hash table υπολογίζουμε την θέση του query και ψάχνουμε ένα ένα τα σημεία μέσα σε αυτό. Για κάθε σημείο ελέγχουμε αν η απόσταση του από το query είναι μικρότερη από την ακτίνα R. Αν αυτό ισχύει και αν το συγκεκριμένο στοιχείο δεν το έχουμε ήδη βρει τότε κάνουμε ένα push_back στον vector Points_range που έχουμε φτιάξει. Οπότε στο τέλος αφού έχουμε ψάξει όλα τα hash tables το «Points_range» θα έχει αποθηκευμένα τα σημεία που βρίσκονται σε εμβέλεια μικρότερη από R από το query. Και κάνουμε και τις κατάλληλες εκτυπώσεις στο αρχείο εξόδου.

Για την «N_Nearestneighbors_LSH» υπολογίζουμε τους N πλησιέστερους γείτονες ενός query. Ψάχνουμε όλα τα hash tables και υπολογίζουμε την θέση του κοντινότερου γείτονα του query με την συνάρτηση Query_Hash. Έχουμε και έναν πίνακα μονοδιάστατο στον οποίο αποθηκεύουμε τις Ν-καλύτερες αποστάσεις. Ψάχνουμε ολόκληρο το hash table αρχικά και γεμίζουμε τον πίνακα των αποστάσεων με τις Ν πρώτες αποστάσεις. Όταν γεμίσει αυτός ο πίνακας κάθε φορά που θα έρχεται μία νέα απόσταση θα εκτελούμε τον παρακάτω αλγόριθμο. Θα βρίσκουμε με ένα for την μέγιστη απόσταση που υπάρχει στον πίνακα. Και θα λέμε αν η απόσταση που έχει έρθει είναι μικρότερη από την μέγιστη απόσταση στον πίνακα και αν αυτή η εικόνα δεν υπάρχει στον vector “N_points” ήδη τότε αντικατέστησε την μέγιστη απόσταση με αυτήν που έχει έρθει. Και αντίστοιχα στον vector που αποθηκεύουμε τα Ν στοιχεία κάνε την κατάλληλη αντικατάσταση. Στο τέλος της συνάρτησης έχουμε τους Ν κοντινότερους γείτονες του query αποθηκευμένους στον vector: “N_points” και κάνουμε τις κατάλληλες εκτυπώσεις στο αρχείο εξόδου.

Για την «Euclidian_NN» υπολογίζουμε με τον exhaustive αλγόριθμο τους Ν κοντινότερους γείτονες από το query. Για κάθε εικόνα υπολογίζουμε την απόσταση της από το query. Έχουμε και δύο vectors στον έναν αποθηκεύουμε τις Ν καλύτερες αποστάσεις και στον άλλον τους αριθμούς των εικόνων με αυτές τις αποστάσεις. Αρχικά γεμίζουμε τον πίνακα με τις αποστάσεις. Και μετά όταν μας έρχεται μία απόσταση θα την βάζουμε σε αυτόν τον πίνακα μόνο αν είναι μικρότερη από την μεγαλύτερη απόσταση που υπάρχει στον πίνακα. Αν ισχύει αυτό γίνονται οι κατάλληλες αντικαταστάσεις και στους δύο πίνακες. Και στο τέλος εκτυπώνουμε στο αρχείο εξόδου την απόσταση του Ν-οστού γείτονα.

Για την «LSH_OutFile» δέχεται σαν όρισμα το L, k, Q_number, N, R και εκτελεί τους αλγορίθμους: LSH για την εύρεση του κοντινότερου γείτονα, LSH για την εύρεση Ν-κοντινότερων γειτόνων ,Range Search και τον exhaustive αλγόριθμο για την εύρεση του πλησιέστερου γείτονα και των Ν πλησιέστερων γειτόνων. Υπολογίζει τους χρόνους εκτέλεσης των αλγορίθμων μέσω της βιβλιοθήκης της C++ “Chono” και τους εκτυπώνει στο αρχείο εξόδου. Ο χρόνος tTrue είναι ο χρόνος που χρειάζεται ο exhaustive αλγόριθμος για την εύρεση του πλησιέστερου γείτονα και των Ν πλησιέστερων γειτόνων. Και ο tLsh είναι το άθροισμα των χρόνων που χρειάζονται οι αλγόριθμοι LSH για την εύρεση πλησιέστερου γείτονα, LSH για την εύρεση Ν-πλησιέστερων γειτόνων, και LSH Range Search.

Για την τυχαία προβολή στον υπερκύβο:

Image.h:

Έχουμε ένα struct που αναπαριστά την εικόνα MNIST και αποτελείται από έναν πίνακα διάστασης 784 και σε αυτό θα αποθηκεύουμε τα pixels ώστε να φτιάξουμε το διάνυσμα της εικόνας.

Έχουμε ορίσει έξι ακεραίους ως extern οι οποίοι είναι ο αριθμός των εικόνων που έχει το input αρχείο, το k, το M, το probes(μέγιστο επιτρεπόμενο πλήθος κορυφών που θα ελεγχθούν), το N (αριθμός των πλησιέστερων γειτόνων) και R (ακτίνα αναζήτησης).

Έχουμε δύο vectors τύπου struct MNIST_Image. Στο ένα αποθηκεύουμε «Images» τα διανύσματα από τις εικόνες που πήραμε από το αρχείο εισόδου (πρακτικά θα περιέχει διανύσματα (με 784 στοιχεία) όσα ο αριθμός των εικόνων που πήραμε για είσοδο). Και ο άλλος vector «Q_Image» θα περιέχει ένα διάνυσμα με 784 το οποίο θα αντιστοιχεί στο διάνυσμα της εικόνας-query. Έχουμε και ένα string το όποιο είναι το όνομα του output αρχείου σε περίπτωση που αυτό δεν δίνεται από τον χρήστη. Και έχουμε και το output αρχείο.

Έχουμε και τον ορισμό δύο συναρτήσεων “Input”, “create_image”.

Image.cpp:

Η “create_image” είναι ίδια με αυτή που χρησιμοποιούμε στον LSH.

Η “Input” επίσης ακολουθεί την ίδια λογική με αυτή που χρησιμοποιήσαμε στον LSH.

Euclidian.h

Περιέχει τους ορισμούς τριών συναρτήσεων. «Euclidian_dist» υπολογίζει την απόσταση μεταξύ δύο διανυσμάτων με βάση την ευκλείδεια μετρική. «Euclidian_Nearest_N» επιστρέφει τον κοντινότερο γείτονα αλλά με τον εξαντλητικό αλγόριθμο. «Euclidian_NNN» επιστρέφει τους N κοντινότερους γείτονες αλλά με τον εξαντλητικό αλγόριθμο.

Euclidian.cpp

Έχει την υλοποίηση των παραπάνω συναρτήσεων η οποία είναι ίδια με τις αντίστοιχες συναρτήσεις που χρησιμοποιούνται στον LSH στο το αρχείο «Hash.h».

Cube.h

Έχουμε δύο πίνακες “Points_Range”, “N_Points” στους οποίους θα αποθηκεύουμε τους αριθμούς των εικόνων μετά από την εκτέλεση του Range Search και την εύρεση N πλησιέστερων γειτόνων αντίστοιχα. Έχουμε ένα διπλό διάνυσμα “searchSpace” στο οποίο θα αποθηκεύουμε τις εικόνες τις οποίες θα ψάχνουμε. Θα περιέχει M διανύσματα των k στοιχείων (0 ή 1). Έχουμε το “BinaryImages” που θα περιέχει διανύσματα μεγέθους k πλήθους όσες και οι εικόνες στο αρχείο εισόδου. Έχουμε αντίστοιχα και το “BianaryQuery” που θα αποθηκεύουμε ένα διάνυσμα μεγέθους k (θα περιέχει 0 ή 1). Και ένα vector “Probe” που θα περιέχει ένα τυχαίο binary διάνυσμα μεγέθους k.

Έχουμε τους ορισμούς συναρτήσεων. Η «HammingDistance» υπολογίζουμε την hamming απόσταση μεταξύ δύο δυαδικών διανυσμάτων. Η «Images_To_Binary» που φτιάχνει το binary διάνυσμα για τις εικόνες και το query. Η «GenerateRandomBinaryProbe» φτιάχνει ένα τυχαίο δυαδικό διάνυσμα μεγέθους k. Η «FIndSearchSpace» βρίσκει τις εικόνες στις οποίες θα εκτελέσουμε τον αλγόριθμο και κρατάει Μ από αυτές. Η «RandomlySelectMPoints» διαλέγει τυχαία Μ από ένα σύνολο εικόνων. Η «Random_projection_to_HyperCube» εκτελεί τον αλγόριθμο του υπερκύβου για την εύρεση του πλησιέστερου γείτονα. Η «Random_projection_to_HyperCube_N» εκτελεί τον αλγόριθμο του υπερκύβου για την εύρεση Ν πλησιέστερων γειτόνων. Η «Random_projection_to_HyperCube_RangeSearch» εκτελεί τον αλγόριθμο του υπερκύβου με Range search. Και η «Cube_OutFile» καλεί τις παραπάνω συναρτήσεις για τη εκτέλεση των αλγορίθμων, κάνει τις κατάλληλες εκτυπώσεις και μέτρηση χρόνου.

Cube_projection.cpp

Για την «HammingDistance» δεχόμαστε σαν όρισμα δύο διανύσματα με μηδέν και ένα και υπολογίζουμε την hamming απόσταση. Δηλαδή βρίσκουμε σε πόσα bits διαφέρουν.

Για την «Images_To_Binary» μετατρέπουμε το διάνυσμα της εικόνας από διάσταση 784 σε ένα δυαδικό διάνυσμα διάστασης k. Αντιστοιχούμε κάθε αριθμό pixel (0-255) στο 0 ή το ένα τυχαία. Αν ξανά έρθει ο ίδιος αριθμός του pixel τότε θυμόμαστε με έναν πίνακα ποια τιμή είχε φέρει η συνάρτηση rand. Το ίδιο κάνουμε και για το query.

Για την «GenerateRandomBinaryProbe» επιστρέφει ένα τυχαίο δυαδικό διάνυσμα μεγέθους k.

Για την «FIndSearchSpace» για κάθε εικόνα βρίσκουμε ένα τυχαίο διάνυσμα probe μεγέθους k. Και αν η hamming απόσταση του σημείου από το query είναι μικρότερη από την απόσταση του query από το συγκεκριμένο probe τότε κρατάμε την εικόνα αυτή. Και αποθηκεύουμε όλα τα διανύσματα των εικόνες που ικανοποιούν αυτή την συνθήκη στο searchSpace και μετά θα διαλέξουμε Μ από αυτές τις εικόνες. Άρα στο τέλος της συνάρτησης θα έχουμε στο searchSpace Μ διανύσματα εικόνων.

Για την «Random_projection_to_HyperCube» για κάθε κορυφή και για όλα τα διανύσματα των εικόνων που υπάρχουν στο searchSpace υπολογίζουμε την hamming απόσταση από το query. Και κρατάμε την μικρότερη απόσταση. Έτσι βρήκαμε τον κοντινότερο γείτονα και κάνουμε τις κατάλληλες εκτυπώσεις στο αρχείο εξόδου.

Για την «Random_projection_to_HyperCube_N» βρίσκουμε τους Ν κοντινότερους γείτονες ψάχνουμε για όλα τα διανύσματα που είναι στο searchSpace έχουμε ένα πίνακα που θα αποθηκεύουμε τις Ν καλύτερες αποστάσεις και άλλον έναν που θα αποθηκεύομε τους αριθμούς των εικόνων με τις καλύτερες αποστάσεις. Αρχικά γεμίζουμε τον πίνακα με τις πρώτες Ν αποστάσεις. Αφού γεμίσει ο πίνακας θα υπολογίζουμε την μέγιστη απόσταση και να υπάρχει μικρότερη απόσταση από αυτή θα κάνει τις κατάλληλες αντικαταστάσεις στα στοιχεία των πινάκων. Στο τέλος θα έχουμε κρατήσει τις Ν καλύτερες αποστάσεις και τους αριθμούς των γειτόνων με αυτές. Και θα κάνουμε τις εκτυπώσεις στο αρχείο εξόδου.

Για την «Random_projection_to_HyperCube_RangeSearch» εκτελούμε τον αλγόριθμο του υπερκύβου με αναζήτηση περιοχής. Για κάθε εικόνα στο searchSpace ελέγχουμε αν η απόσταση της από το query είναι μικρότερη από την ακτίνα αναζήτησης αν ισχύει και αν δεν την έχουμε ήδη βρει τότε την αποθηκεύουμε στον αντίστοιχο πίνακα. Και κάνουμε τις εκτυπώσεις στο αρχείο εξόδου.

Για την «Cube_OutFile» δέχεται σαν όρισμα το p, k, Q_number, N, R και εκτελεί τους αλγορίθμους: Cube για την εύρεση του κοντινότερου γείτονα, Cube για την εύρεση N-κοντινότερων γειτόνων ,Range Search, τον exhaustive αλγόριθμο για την εύρεση του πλησιέστερου γείτονα και τον exhaustive αλγόριθμο για την εύρεση N πλησιέστερων γειτόνων. Υπολογίζει τους χρόνους εκτέλεσης των αλγορίθμων μέσω της βιβλιοθήκης της C++ “Chono” και τους εκτυπώνει στο αρχείο εξόδου. Ο χρόνος tTrue είναι ο χρόνος που χρειάζεται ο exhaustive αλγόριθμος για την εύρεση του πλησιέστερου γείτονα και των N πλησιέστερων γειτόνων. Και ο tCube είναι το άθροισμα των χρόνων που χρειάζονται οι αλγόριθμοι Cube για την εύρεση πλησιέστερου γείτονα, Cube για την εύρεση N-πλησιέστερων γειτόνων, και Cube Range Search.

Για την συσταδοποίση:

Image.h:

Έχουμε ένα struct που αναπαριστά την εικόνα MNIST και αποτελείται από έναν πίνακα διάστασης 784 και σε αυτό θα αποθηκεύουμε τα pixels ώστε να φτιάξουμε το διάνυσμα της εικόνας.

Έχουμε ορίσει έξι ακεραίους ως extern οι οποίοι είναι ο αριθμός των εικόνων που έχει το input αρχείο, το k_LSH, M, probes, K, k_Cube και L. Έχουμε το CompFlag το οποίο θα είναι ένα όταν ο χρήστης δίνει το -complete και μηδέν αλλιώς.

Έχουμε ένα vector τύπου struct MNIST_Image. Στο οποίο αποθηκεύουμε τα διανύσματα από τις εικόνες που πήραμε από το αρχείο εισόδου (πρακτικά θα περιέχει διανύσματα (με 784 στοιχεία) όσα ο αριθμός των εικόνων που πήραμε για είσοδο). Έχουμε και ένα string το όποιο είναι το όνομα του output αρχείου σε περίπτωση που αυτό δεν δίνεται από τον χρήστη. Και έχουμε και το output αρχείο. Έχουμε και μία extern μεταβλητή τύπου string στην οποία θα αποθηκεύουμε ποια μέθοδο θέλει ο χρήστης να τρέξει το πρόγραμμα.

Έχουμε και τον ορισμό συναρτήσεων “Input”, “create_image”, “readFromFile”.

Image.cpp:

Η “create_image” είναι ίδια με αυτή που χρησιμοποιούμε στον LSH.

Η “Input” επίσης ακολουθεί την ίδια λογική με αυτή που χρησιμοποιήσαμε στα προηγούμενα αρχεία.

Η «readFromFile» έχει σαν όρισμα το όνομα ενός αρχείου ελέγχει αρχικά αν το αρχείο ανοίγει σωστά. Χρησιμοποιούμε την συνάρτηση αυτή για να διαβάσουμε από το αρχείο cluster.conf τις κατάλληλες μεταβλητές. Αν για κάποια μεταβλητή ο χρήστης δεν έχει δώσει τιμή τότε το πρόγραμμα χρησιμοποιεί τις default τιμές των μεταβλητών.

Hash.h:

Έχουμε τους ορισμούς των συναρτήσεων που θα χρησιμοποιούσουμε για να εκτελέσουμε τους αλγορίθμους του LSH range search και για την τυχαία προβολή στον υπερκύβο. Μαζί με τις συναρτήσεις έχουμε και κάποιες μεταβλητές. Όλα αυτά είναι ίδια με τις συναρτήσεις στα αρχεία Hash.h και Cube.h.

Hash.cpp:

Έχουμε την υλοποίηση των συναρτήσεων που υπάρχουν στο .h αρχείο. Οι συναρτήσεις είναι ίδιες με την υλοποίηση που ήδη έχουμε κάνει στους άλλους φακέλους. Η μόνη διαφορά είναι πως στη θέση του Query πλέον έχουμε το centroid. Γιατί υπολογίζουμε τις αποστάσεις των εικόνων από κάποιο centroid.

Cluster.h:

Έχουμε την δομή centroids στην οποία θα αποθηκεύουμε τις συντεταγμένες όλων των centroids. Για να κρατάμε αυτές τις πληροφορίες πρέπει το centroids να είναι ένας διπλός vector.

Έχουμε και έναν απλό vector clusters στον οποίο αποθηκεύουμε στην θέση i τον αριθμό του cluster στο οποίο ανήκει η εικόνα με αριθμό i (για παράδειγμα clusters[2] = 3 σημαίνει ότι η εικόνα με αριθμό δύο ανήκει στο cluster τρία).

Έχουμε και τους ορισμούς των παρακάτω συναρτήσεων. Η «Euclidian_dist» υπολογίζει την απόσταση μίας εικόνας από ένα centroid. Η «k_MeansPlusPlus_Init» κάνει την αρχικοποίηση των centroids. Η «AssignToClusters» βάζει όλες τις εικόνες σε ένα cluster. Η «updateCentroids» βρίσκει τις νέες συντεταγμένες των centroids. Η «AssignToClusters_LSH» εκτελεί τον αλγόριθμο με range search LSH. Η «AssignToClusters_Cube» εκτελεί τον αλγόριθμο με range search HyperCube. Η «macQueen» καλεί τις συναρτήσεις κάνοντας την εκτέλεση του αλγορίθμου που ζητάει ο χρήστης. «Euclidian_dist_between_images» επιστρέφει την ευκλείδεια απόσταση μεταξύ δύο εικόνων. Η «CalculateA», «CalculateB» που υπολογίζει αντίστοιχα τα a, b για κάποιο cluster. Η «OutFile» καλεί τις συναρτήσεις για την εκτέλεση των αλγορίθμων και υπολογίζει και τον χρόνο.

Cluster.cpp:

Η «Euclidian_dist» δέχεται σαν όρισμα δύο ακεραίους ο ένας είναι ο αριθμός μίας εικόνας και ο άλλος ενός cluster. Και υπολογίζει την απόσταση μεταξύ των δύο με την ευκλείδεια μετρική.

Η «k_MeansPlusPlus_Init» αρχικοποιούμε το πρώτο centroid με το διάνυσμα μίας τυχαίας εικόνας από αυτές που έχουμε από το input αρχείο.
Για τα υπόλοιπα κέντρα (από i = 1 έως k - 1, όπου το k είναι ο επιθυμητός αριθμός κέντρων), ακολουθούμε τα παρακάτω βήματα:
Υπολογίζουμε την ελάχιστη απόσταση από κάθε εικόνα προς το πλησιέστερο ήδη επιλεγμένο κέντρο και αποθηκεύουμε αυτές τις αποστάσεις στο διάνυσμα minDistances. Με αυτό το βήμα εξασφαλίζουμε ότι το επόμενο κέντρο είναι πιθανό να βρίσκεται αρκετά μακριά από τα υπάρχοντα κέντρα. Από το διάνυσμα minDistances υπολογίζουμε το totalDistance, που αντιπροσωπεύει στο άθροισμα των αποστάσεων προς τα πλησιέστερα υπάρχοντα κέντρα για όλες τις εικόνες. Δημιουργούμε μια τυχαία απόσταση ανάμεσα στο 0 και το totalDistance. Ο στόχος είναι να επιλέξουμε το επόμενο κέντρο να είναι όσο το δυνατόν πιο μακριά από τα υπάρχοντα κέντρα. Τέλος διατρέχουμε όλες τις εικόνες και αφαιρούμε τις αντίστοιχες minDistances από την πιθανή απόσταση μέχρι να γίνει μηδενική ή αρνητική. Η εικόνα στην οποία η πιθανή απόσταση γίνεται μηδενική ή αρνητική επιλέγεται ως το επόμενο κέντρο. Και η διαδικασία επαναλαμβάνεται μέχρι να επιλεγούν k κέντρα. Τα τελικά κέντρα αποθηκεύονται στο διάνυσμα centroids.

Η «AssignToClusters» για κάθε εικόνα η συνάρτηση υπολογίζει την απόσταση της από κάθε ένα από τα centroids καλώντας την συνάρτηση «Euclidian_dist». Συγκρίνει αυτές τις αποστάσεις για να βρει το κοντινότερο centroid. Η εικόνα ανατίθεται στη συστάδα που αντιστοιχεί στο πλησιέστερο κέντρο. Ο αριθμός της συστάδας στην οποία ανατίθεται η εικόνα αποθηκεύεται στο διάνυσμα “clusters”. Άρα τελικά το διάνυσμα αυτό περιέχει τις αναθέσεις των εικόνων στις αντίστοιχες συστάδες, με βάση τα κέντρα που έχουν υπολογιστεί.

Η «updateCentroids» δημιουργούμε δύο νέα διανύσματα «clusterSizes», «newCentroids». Το πρώτο περιέχει τον αριθμό των εικόνων που έχουν ανατεθεί στην κάθε συστάδα και το δεύτερο διάνυσμα θα χρησιμοποιηθεί για τον υπολογισμό των νέων κέντρων. Στη συνέχεια για κάθε εικόνα η συνάρτηση εντοπίζει σε ποια συστάδα ανήκει η εικόνα.

Το διάνυσμα της εικόνας θα προστεθεί στο αντίστοιχο νέο κέντρο στο διάνυσμα newCentroids. Και αυξάνεται ο αριθμός των εικόνων που ανήκουν σε κάθε συστάδα καταλλήλως στο διάνυσμα clusterSizes. Μετά κάνουμε έλεγχο για κάθε συστάδα. Αν η συστάδα έχει εικόνες το κέντρο της συστάδας ενημερώνεται με τον μέσο όρο των συντεταγμένων των σημείων που της ανήκουν. Στο τέλος το διάνυσμα centroids ενημερώνεται με τα νέα κέντρα.

Η «AssignToClusters_LSH» Πραγματοποιεί την αρχικοποίηση των centroids χρησιμοποιώντας την μέθοδο K-Means++. Υπολογίζει την ακτίνα ως το μισό της Ευκλείδειας απόστασης μεταξύ των δύο πρώτων centroids. Καλεί τις συναρτήσεις Hash_initialize(L) και HashFunctions_initialize(k_lsh, L) για την αρχικοποίηση του LSH και την Hash_clear(L) για να αδειάσει τις δομές του LSH. Υπολογίζει τις τιμές για τη συνάρτηση Euclidian_Hash για όλες τις εικόνες. Επαναλαμβάνουμε την βασική διαδικασία του αλγορίθμου για ένα συγκεκριμένο αριθμό επαναλήψεων. Για κάθε centroid καλεί την συνάρτηση LSH_RangeSearch για την αναζήτηση σημείων στο εύρος του τρέχοντος centroid. Ενημερώνει τον πίνακα clusters με τις ανατεθειμένες εικόνες. Υπολογίζει τις συντεταγμένες του νέο κέντρου για το cluster. Διπλασιάζει την τιμή του R για τον επόμενο κύκλο της επανάληψης. Στο τελευταίο for ελέγχει κάθε εικόνα αν ανήκει στον πίνακα Points_Range αν έχει ανατεθεί δηλαδή σε κάποιο cluster. Αν δεν ανήκει, υπολογίζει το κοντινότερο centroid (ανάλογα με την ευκλείδεια απόσταση) και αναθέτει την εικόνα στο κατάλληλο cluster.

Η «AssignToClusters_Cube» κάνει ότι ακριβώς και η προηγούμενη συνάρτηση απλώς αντί να καλεί την συνάρτηση που για range search με LSH, καλεί την συνάρτηση για range search με προβολή στον υπερκύβο.

Η «macQueen» ανάλογα με τον αλγόριθμο που ζητάει ο χρήστης καλεί και την κατάλληλη συνάρτηση και κάνει τις εκτυπώσεις για τις συντεταγμένες των κέντρων στο αρχείο εξόδου.

Η «Euclidian_dist_between_images» επιστρέφει την ευκλείδεια απόσταση μεταξύ δύο εικόνων.

Η «CalculateA» δέχεται σαν όρισμα τον αριθμό μίας εικόνας και υπολογίζει το α(Number_Image) υπολογίζει την μέση απόσταση των ενός σημείου από όλα τα άλλα σημεία που ανήκουν στο ίδιο cluster. Για κάθε εικόνα αν αυτή είναι διαφορετική από αυτή που δώσαμε σαν όρισμα και ανήκει όμως στο ίδιο cluster με την εικόνα του ορίσματος της συνάρτησης υπολόγισε την απόσταση. Προσθέτουμε τις αποστάσεις και επιστρέφουμε την μέση.

Η «CalculateB» δέχεται σαν όρισμα τον αριθμό των clusters και τον αριθμό μίας εικόνας. Για όλα τα cluster (εκτός από αυτό που ανήκει η εικόνα που δεχόμαστε σαν όρισμα της συνάρτησης υπολογίζουμε την μέση απόσταση της εικόνας από όλες τις εικόνες που ανήκουν στα άλλα clusters. Βρίσκουμε την μικρότερη από αυτές τις αποστάσεις και την επιστρέφουμε.

Η «OutFile» καλεί την παραπάνω συνάρτηση για την εκτέλεση του κατάλληλου αλγορίθμου. Και υπολογίζει και τους κατάλληλους χρόνους. Αν ο χρήστης έχει δώσει το «-complete» τότε πρέπει να εκτυπώσουμε για κάθε cluster τους αριθμούς των εικόνων που έχει το κάθε ένα. Φτιάχνουμε ένα διπλό διάνυσμα στο οποίο θα αποθηκεύουμε τους αριθμούς των εικόνων που θα έχει το κάθε cluster.

Αρχικά πρέπει να σβήσουμε τα μηδενικά τα οποία έχουν δημιουργηθεί με την συνάρτηση resize. Αφού πια το διάνυσμα είναι άδειο πάμε και βάζουμε στο κάθε cluster τον αριθμό των εικόνων που περιέχει. Και στο τέλος κάνουμε την εκτύπωση στο αρχείο εξόδου. Επίσης υπολογίζουμε την Silhouette. Για να την υπολογίσουμε πρέπει να ξέρουμε μία εικόνα από το κάθε cluster. Οπότε έχουμε ένα for στο οποίο αποθηκεύουμε στο διάνυσμα "ImagesCluster" Κ τον αριθμό εικόνες, η κάθε μία από αυτές ανήκει σε διαφορετικό cluster. (Αν για παράδειγμα έχουμε: ImagesCluster = [22, 34, 90] σημαίνει ότι έχουμε τρία διαφορετικά clusters και η εικόνα με αριθμό 22 ανήκει στο πρώτο, η εικόνα 34 στο δεύτερο και η εικόνα 90 στο τρίτο). Υπολογίζουμε τα a_i, b_i για κάθε cluster i καλώντας τις κατάλληλες συναρτήσεις υπολογίζουμε έτσι τα s_i, s_total και κάνουμε την εκτύπωση στο αρχείο εξόδου.

Οδηγίες μεταγλώττισης:

- Για τον αλγόριθμο LSH. Βρισκόμαστε στον φάκελο LSH και καλούμε: make lsh έτσι δημιουργείται το εκτελέσιμο αρχείο lsh. Μπορούμε να το εκτελέσουμε με τους παρακάτω δύο τρόπους:
    - ./lsh σε αυτήν την περίπτωση το πρόγραμμα θα ζητήσει από τον χρήστη τις διαδρομές για τα κατάλληλα αρχεία και τις τιμές για τα k,L,N,R.
    - ./lsh –d <input_file> –q <query_file> -k <int> -L <int> -o <output_file> -N <int> -R <radius>

    Στην πρώτη περίπτωση αν ο χρήστης δεν θέλει να δώσει κάποια τιμή σε μία μεταβλητή τότε απλά γράφει "-" και σε αυτήν την περίπτωση χρησιμοποιούνται οι default τιμές.

- Για τον αλγόριθμο προβολής στον υπερκύβο. Βρισκόμαστε στον φάκελο Cube και καλούμε: make cube έτσι δημιουργείται το εκτελέσιμο αρχείο cube. Μπορούμε να το εκτελέσουμε με τους παρακάτω δύο τρόπους:
    - ./cube σε αυτήν την περίπτωση το πρόγραμμα θα ζητήσει από τον χρήστη τις διαδρομές για τα κατάλληλα αρχεία και τις τιμές για τα k,M,N,R, probes.
    - ./cube –d <input_file> –q <query_file> –k <int> -M <int> -probes <int> -o <output_file> -N <int> -R <int>

    Στην πρώτη περίπτωση αν ο χρήστης δεν θέλει να δώσει κάποια τιμή σε μία μεταβλητή τότε απλά γράφει "-" και σε αυτήν την περίπτωση χρησιμοποιούνται οι default τιμές.

- Για την συσταδοποίηση. Βρισκόμαστε στο φάκελο Cluster και καλούμε make cluster έτσι δημιουργείται το εκτελέσιμο αρχείο cluster. Μπορούμε να το εκτελέσουμε με τον παρακάτω τρόπο:
    - $./cluster –i <input_file> –c <configuration_file> -o <output_file> -complete -m <method>
      Για το method έχουμε τρεις εναλλακτικές: Classic, LSH ή Hypercube.
      Και το «-complete» δεν είναι υποχρεωτικό να το δώσει ο χρήστης.

Εξήγηση κάθε προγράμματος:

Για τον φάκελο **LSH** έχουμε τα παρακάτω αρχεία:

- **Random.h**, **Random.cpp** περιέχει τους ορισμούς και την υλοποίηση των συναρτήσεων που θα χρησιμοποιήσουμε για να φτιάξουμε τυχαίους αριθμούς με την Normal, Uniform distribution.
- **Image_vector.h**, **Create_image_vector.cpp** έχουμε την συνάρτηση που διαβάζει τα δεδομένα από το binary αρχείο και αποθηκεύουμε τα διανύσματα των εικόνων και του query σε μία extern μεταβλητή. Επίσης έχουμε και την συνάρτηση που διαχειρίζεται και τα δεδομένα που δίνει ο χρήστης.
- **Hash.h**, **Hash.cpp** φτιάχνουμε και αρχικοποιούμε τις δομές των hash tables, hash functions και ότι χρειαζόμαστε για την εκτέλεση των αλγορίθμων. Έχουμε επίσης τις συναρτήσεις για την εκτέλεση των αλγορίθμων εύρεσης πλησιέστερου γείτονα, εύρεση N πλησιέστερων γειτόνων και range search με LSH.

- Τέλος έχουμε και την συνάρτηση που καλεί όλες τις προηγούμενες συναρτήσεις για εκτέλεση των αλγορίθμων και υπολογίζει και τον χρόνο.
- **lsh.cpp** περιέχει την main συνάρτηση η οποία αρχικά καλεί την συνάρτηση για να φτιάξουμε τα διανύσματα των εικόνων και του query και διαβάζει και τα δεδομένα που έχει δώσει ο χρήστης. Κάνει αρχικοποίηση των δομών για τον LSH και μετά καλεί την συνάρτηση για την εκτέλεση των αλγορίθμων.
  Μετά από την πρώτη εκτέλεση των αλγορίθμων το πρόγραμμα ρωτάει τον χρήστη αν θέλει να τερματίσει ή αν θέλει να συνεχίσει με άλλα δεδομένα αν ο χρήστης γράψει "END" τότε τερματίζει αν γράψει "continue" θα του ζητηθούν να ξανά δώσει τις παραμέτρους και θα ξανά εκτελεστούν οι αλγόριθμοι.

Για τον φάκελο **Cube** έχουμε τα παρακάτω αρχεία:

- **Image.h**, **Image.cpp** είναι ίδια με τα προηγούμενα φτιάχνουμε τα διανύσματα των εικόνων και διαβάζουμε τα δεδομένα από τον χρήστη.
- **Euclidian.h**, **Euclidian.cpp** έχουμε την υλοποίηση για τις συναρτήσεις που:
  1. Υπολογίζει την ευκλείδεια απόσταση μεταξύ μίας εικόνας και του query.
  2. Εκτελεί τον exhaustive αλγόριθμο για την εύρεση του πλησιέστερου γείτονα.
  3. Εκτελεί τον exhaustive αλγόριθμο για την εύρεση των N πλησιέστερων γειτόνων.
- **Cube.h**, **Cube_projection.cpp** έχουμε τις συναρτήσεις για την αρχικοποίηση των δομών για τον αλγόριθμο του υπερκύβου και την εκτέλεση του. Έχουμε και την συνάρτηση που καλεί τις συναρτήσεις για την εκτέλεση του αλγορίθμου εύρεσης του πλησιέστερου γείτονα, των N πλησιέστερων γειτόνων και το range search.
- **cube.cpp** περιέχει την main συνάρτηση η οποία αρχικά καλεί την συνάρτηση για να φτιάξουμε τα διανύσματα των εικόνων και του query και διαβάζει και τα δεδομένα που έχει δώσει ο χρήστης. Κάνει αρχικοποίηση των δομών για τον hypercube και μετά καλεί την συνάρτηση για την εκτέλεση των αλγορίθμων.
  Μετά από την πρώτη εκτέλεση των αλγορίθμων το πρόγραμμα ρωτάει τον χρήστη αν θέλει να τερματίσει ή αν θέλει να συνεχίσει με άλλα δεδομένα αν ο χρήστης γράψει "END" τότε τερματίζει αν γράψει "continue" θα του ζητηθούν να ξανά δώσει τις παραμέτρους και θα ξανά εκτελεστούν οι αλγόριθμοι.

Για τον φάκελο **Cluster** έχουμε τα παρακάτω αρχεία:

- **Image.h**, **Image.cpp** είναι παρόμοια με τα προηγούμενα φτιάχνουμε τα διανύσματα των εικόνων και διαβάζουμε τα δεδομένα από τον χρήστη. Σε αυτή την περίπτωση έχουμε και μία συνάρτηση που διαβάζει τα δεδομένα από το αρχείο που δίνει ο χρήσης.
- **Hash.h**, **Hash.cpp** περιέχει τις δομές και τις συναρτήσεις που χρησιμοποιούμε για τον αλγόριθμο του LSH range search και του HyperCube range search είναι ίδιες συναρτήσεις με αυτές που χρησιμοποιούνται και στα αρχεία στου άλλους φακέλους. Η μόνη διαφορά που υπάρχει είναι ότι αντί να υπολογίζουμε την απόσταση μεταξύ μίας εικόνας και του query. Υπολογίζουμε την απόσταση της εικόνας από ένα κέντρο.
- **Cluster.h, Cluster.cpp** έχουμε την υλοποίηση των συναρτήσεων για την εκτέλεση των αλγορίθμων clustering. Έχουμε την συνάρτηση "macQueen" η οποία ανάλογα με το ποια μέθοδο ζητάει ο χρήστης καλεί τις κατάλληλες συναρτήσεις για την εκτέλεση των αλγορίθμων.

Και υπάρχει και η συνάρτηση που καλείται από την main και καλεί την προηγούμενη συνάρτηση, υπολογίζει τον χρόνο εκτέλεσης και το silhouette και αν έχει δοθεί και σαν παράμετρος το “-complete” κάνει και τις αντίστοιχες εκτυπώσεις.

- **main.cpp** περιέχει την main συνάρτηση η οποία αρχικά καλεί την συνάρτηση για να δημιουργήσει τα διανύσματα των εικόνων και να διαβάσει τα δεδομένα που δίνει ο χρήστης τόσο από την γραμμή εντολών όσο και από το αντίστοιχο αρχείο. Και μετά καλεί την συνάρτηση για την εκτέλεση των αλγορίθμων και κάνει και τις κατάλληλες εκτυπώσεις στο αρχείο εξόδου.