ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ

ΥΠΟΛΟΓΙΣΤΩΝ

# ΕΠΕΞΕΡΓΑΣΙΑ ΕΡΩΤΗΜΑΤΩΝ Κ-ΚΟΝΤΙΝΟΤΕΡΩΝ ΓΕΙΤΟΝΩΝ ΓΙΑ ΟΛΟΥΣ

# PROCESSING OF ALL K-NEAREST NEIGHBOR QUERIES

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΩΠΑΣ ΝΙΚΟΛΑΟΣ

ΕΠΙΒΛΕΠΩΝ:

ΒΑΣΙΛΑΚΟΠΟΥΛΟΣ ΜΙΧΑΗΛ, ΑΝΑΠΛΗΡΩΤΗΣ ΚΑΘΗΓΗΤΗΣ

Βόλος, 2017

# ΕΥΧΑΡΙΣΤΙΕΣ

Πρώτα απ’ όλα, θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα της διπλωματικής εργασίας μου, Αναπληρωτή Καθηγητή κ. Βασιλακόπουλο Μιχαήλ, για την πολύτιμη βοήθεια και καθοδήγησή του σε όλη τη διάρκεια της διπλωματικής εργασίας μου, αλλά και γενικότερα στην πορεία μου στη σχολή, συμβουλεύοντάς με ώστε να πάρω τις σωστές αποφάσεις. Επίσης, ευχαριστώ και όλα τα υπόλοιπα μέλη της επιτροπής για το χρόνο που μου διέθεσαν.

Έπειτα, θα ήθελα να ευχαριστήσω τον Αναπληρωτή Καθηγητή κ. Corral Liria, Antonio Leopoldo, τον οποίο γνώρισα στα πλαίσια του προγράμματος Erasmus+, και με τις συμβουλές και υποδείξεις του, μου έδωσε τροφή για σκέψη για μια μελλοντική επέκταση της εργασίας μου.

Τέλος, θα ήθελα να ευχαριστήσω τον αδερφό μου Νιώπα Δημήτρη για τη στήριξή του και την πίστη του σε μένα, καθώς και τους γονείς μου Νιώπα Ιωάννη και Γαβριέλη Χρυσή για την στήριξη τους και για τις αξίες που μου μετέδωσαν.

# ΠΕΡΙΛΗΨΗ

Με την πάροδο των χρόνων, ο όγκος των δεδομένων που πρέπει να υποβληθούν σε επεξεργασία ολοένα και αυξάνεται. Κατά συνέπεια, χρειάζεται να βρεθούν αποδοτικοί τρόποι της διαχείρισης αυτού του μεγάλου όγκου δεδομένων. Ένα από τα θεμελιώδη ερωτήματα, το οποίο είναι υπολογιστικά ακριβό, είναι το ερώτημα όλων των k-κοντινότερων γειτόνων (AkNN). Το ερώτημα AkNN βρίσκει για κάθε σημείο που ανήκει σε ένα dataset A, τους k κοντινότερους γείτονές του από ένα άλλο dataset B. Το AkNN είναι ένα από τα βασικά ερωτήματα που εμφανίζεται σε πολλές εφαρμογές (π.χ. τα GIS και η επεξεργασία εικόνας) που απαιτούν λειτουργίες εξόρυξης δεδομένων, όπως την κατηγοριοποίηση και την παλινδρόμηση. Σε αυτή την εργασία θα προσπαθήσουμε να βελτιώσουμε τον υπολογισμό των AkNN ερωτημάτων με αντικείμενα 2 διαστάσεων και θα παρουσιάσουμε 3 νέες παραλλαγές αλγορίθμων για το Join και Self-Join AkNN. Ο πρώτος αλγόριθμος αναζητά γείτονες εναλλάξ αριστερά και δεξιά από κάθε σημείο του οποίου ψάχνουμε τους γείτονες, ενώ οι άλλοι 2 βασίζονται στην αποικοδόμηση του χώρου. Ο ένας χωρίζει το χώρο σε λωρίδες, έτσι ώστε κάθε λωρίδα να έχει το ίδιο ύψος και ο άλλος χωρίζει το χώρο σε λωρίδες με ίδιο αριθμό αντικειμένων. Τέλος, αξιολογούμε τους αλγορίθμους αυτούς μέσω πειραμάτων με πραγματικά δεδομένα και στη συνέχεια παρουσιάζουμε συγκριτικά αποτελέσματα επιδόσεων.

# ABSTRACT

During the last years, the amount of data that needs to be processed is vastly increasing. Thus, we need to find efficient ways of manipulating such amount of data. One of the basic queries, which is computationally expensive, is the All-k-Nearest-Neighbor query (AkNN). An AkNN query finds from a given object set A, k nearest neighbors for each object in a specified query set B. The AkNN is a basic query that is common in many applications (e.g. in GIS and image analysis) that require data mining functions such as classification and regression. In this paper we will try to improve the computation of the AkNN queries with 2-dimensional objects and we will present the implementation of 3 new variations of algorithms for the Join and Self-Join AkNN. The first one is searching for neighbors alternately on the right and on the left side of each object that we are searching for its neighbors, and the other 2 are based on space decomposition. The one divides the area in strips, in order that each strip has the same height and the second one divides the area in strips that have the same amount of objects. Finally, the implementations are evaluated through a process of experiments using real data sets, and then we present the comparative performance results.

# ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

# ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

***Σχήμα 6.6.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Free Strips με χωρισμό σε λωρίδες 3 σημείων. (6)

# ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

***Πίνακας 1***. Παράδειγμα αποθήκευσης των 3-κοντινότερων γειτόνων self-join.

***Πίνακας 2.*** Παράδειγμα αποθήκευσης των 3-κοντινότερων γειτόνων join

# ΚΑΤΑΛΟΓΟΣ ΓΡΑΦΗΜΑΤΩΝ

# 1.ΕΙΣΑΓΩΓΗ

Στην εποχή μας οι βάσεις δεδομένων είναι ιδιαίτερα χρήσιμες και αναγκαίες για τη διαχείριση δεδομένων με σκοπό την επίλυση πολλών προβλημάτων. Χρησιμοποιούνται για να αποθηκεύουν διάφορα δεδομένα όπως σελίδες ιστού, συλλογές γονιδιωμάτων, σχέδια από τσιπάκια, βίντεο, δορυφορικές εικόνες, μουσική ή χάρτες [1]. Με την ευρεία χρήση τους, υπάρχει η ανάγκη για περισσότερες τεχνικές αποδοτικής διαχείρισής τους, συνδυάζοντας κλασικές αλλά και εξειδικευμένες προσεγγίσεις στα διάφορα είδη δεδομένων. Παρ' όλα αυτά οι βασικές λειτουργίες των Βάσεων Δεδομένων, δηλαδή η αποδοτική αποθήκευση δεδομένων και οι ποικίλες αναζητήσεις αυτών δεν έχουν αλλάξει. Συνεπώς, χρήζουν εύρεσης νέοι βελτιστοποιημένοι αλγόριθμοι για πιο γρήγορη απόκριση στις αναζητήσεις, καθώς όσο αυξάνεται η πολυπλοκότητα και το μέγεθος των δεδομένων, τόσο περισσότερο αυξάνεται και ο χρόνος αναζήτησης.

## 1.1 ΧΩΡΙΚΕΣ ΒΑΣΕΙΣ ΔΕΔΟΜΕΝΩΝ

Οι χωρικές βάσεις δεδομένων προσφέρουν την αποθήκευση χωρικών δεδομένων (όπως σημεία, γραμμές, σχήματα, περιοχές), χρησιμοποιώντας μια γλώσσα αναζήτησης για ευρήματα χωρικών συντεταγμένων, τεχνικές κατάταξης των δεδομένων σε πίνακες καθώς και αποδοτική επεξεργασία χωρικών αναζητήσεων [2]. Με τα χρόνια, οι χωρικές βάσεις δεδομένων έχουν εξελιχθεί αισθητά, καταλήγοντας να έχουν πολύ περισσότερες δυνατότητες σήμερα ώστε να διαχειρίζονται πιο πολύπλοκους τύπους χωρικών δεδομένων και να εκτελούν πολλούς νέους αλγόριθμους.

Τα συστήματα διαχείρισης βάσεων δεδομένων αποθηκεύουν και διατηρούν μεγάλες συλλογές από πολυδιάστατα δεδομένα. Παλαιότερες έρευνες επάνω στις χωρικές βάσεις δεδομένων εστίασαν στην αξιολόγηση των συνηθισμένων τύπων αναζητήσεων, όπως τα range queries [3] (π.χ. βρείτε όλα τα αντικείμενα που τέμνουν ή εμπεριέχονται σε μία χωρική περιοχή), τις spatial joins [4] (π.χ. βρείτε όλα τα ζευγάρια αντικειμένων από 2 σετ δεδομένων που ικανοποιούν ένα χωρικό εύρημα) και τα ερωτήματα κοντινότερων γειτόνων [5] (π.χ. βρείτε το πιο κοντινό αντικείμενο από ένα σημείο αναφοράς ή από ένα άλλο αντικείμενο). Τα αποτελέσματα έδειξαν ότι υπάρχει χώρος για έρευνα σε αυτά τα κομμάτια, καθώς υπάρχει ανάγκη για πιο γρήγορους και αποδοτικούς αλγόριθμους που απαντούν σε τέτοιους είδους ερωτήματα, ιδιαίτερα εξαιτίας του μεγάλου ρυθμού αύξησης της πολυπλοκότητας των χωρικών δεδομένων.

## 1.2 ΕΦΑΡΜΟΓΕΣ ΤΩΝ ΧΩΡΙΚΩΝ ΒΑΣΕΩΝ ΔΕΔΟΜΕΝΩΝ

Από όλων των ειδών τα εξειδικευμένα δεδομένα που διαχειρίζονται οι σημερινές βάσεις δεδομένων, τα χωρικά δεδομένα εμφανίζονται σε πολλές εφαρμογές. Σε αυτές συμπεριλαμβάνονται τα συστήματα γεωγραφικών πληροφοριών (GIS), ο σχεδιασμός με τη βοήθεια υπολογιστή (CAD), η ρομποτική, η επεξεργασία εικόνας και τα VLSI. Για όλες αυτές τις εφαρμογές, πυρήνας τους είναι τα χωρικά αντικείμενα τα οποία πρέπει να ταξινομηθούν, να αναζητηθούν και να απεικονιστούν.

Σύμφωνα με το βιβλίο [1], υπάρχει μια ολοένα αυξανόμενη ανάγκη για εξειδικευμένες τεχνικές για τη διαχείριση των χωρικών αυτών αντικειμένων και αυτό δημιουργεί ένα ερευνητικό κενό στην περιοχή των χωρικών βάσεων δεδομένων. Σε σχετικά σύντομο χρονικό διάστημα οι χωρικές βάσεις δεδομένων ανέπτυξαν περιεκτικές τεχνολογικές μεθόδους, συμπεριλαμβανομένων των αναπαραστάσεων χωρικών αντικειμένων, χωρική πρόσβαση σε μεθόδους για γρήγορη ανάκτηση, ειδικές γλώσσες αναζήτησης και αλγόριθμους προσαρμοσμένους σε γειτονικές επιστημονικές περιοχές, όπως η υπολογιστική γεωμετρία [1].

## 1.3 ΑΝΑΖΗΤΗΣΕΙΣ Κ-ΚΟΝΤΙΝΟΤΕΡΩΝ ΓΕΙΤΟΝΩΝ

Οι αναζητήσεις κ-κοντινότερων γειτόνων (kNN) είναι μη παραμετρικές μέθοδοι που χρησιμοποιούνται για ταξινόμηση (classification) και παλινδρόμηση (regression) [7]. Στην ταξινόμηση, το αντικείμενο κατατάσσεται σύμφωνα με την πλειονότητα των ψήφων των κ-κοντινότερων γειτόνων του, με αποτέλεσμα να ενσωματωθεί στην κλάση στην οποία ανήκουν οι περισσότεροι γείτονές του (η παράμετρος κ είναι ένας θετικός ακέραιος αριθμός, συνήθως μικρός). Στην περίπτωση που το κ = 1, τότε το αντικείμενο ενσωματώνεται στην κλάση που ανήκει το πιο κοντινό αντικείμενο προς αυτό. Στην οπισθοδρόμηση, το output είναι μια τιμή βάρους για το αντικείμενο. Η τιμή αυτή είναι ο μέσος όρος των τιμών των κ-κοντινότερων γειτόνων του.

Οι κύριοι αλγόριθμοι που θα εξετάσουμε σε αυτή την εργασία είναι α) η αναζήτηση όλων των κ-κοντινότερων γειτόνων από ένα dataset (self-join) και β) η αναζήτηση όλων των κ-κοντινότερων γειτόνων από 2 διαφορετικά datasets (join). Δεδομένων 2 datasets A και B τα οποία περιέχουν χωρικά αντικείμενα, η αναζήτηση όλων των κ-κοντινότερων γειτόνων (AkNN) επιστρέφει για κάθε αντικείμενο στο dataset A τους κ-κοντινότερους γείτονες του από το dataset B [8]. Υπάρχει επίσης και η self-join τροποποίηση, κατά την οποία το dataset είναι ένα και για κάθε αντικείμενο επιστρέφεται ένα group αντικειμένων από τι ίδιο dataset.

## 1.4 ΕΦΑΡΜΟΓΕΣ ΑΝΑΖΗΤΗΣΕΩΝ Κ-ΚΟΝΤΙΝΟΤΕΡΩΝ ΓΕΙΤΟΝΩΝ

Υπάρχουν πολυάριθμες εφαρμογές των αλγορίθμων τύπου k-κοντινότερων γειτόνων και κάθε μέρα υπάρχουν νέα προβλήματα που εμφανίζονται και μπορούν να αντιμετωπιστούν με τέτοιου είδους αλγόριθμους. Κατ' αρχάς χρησιμοποιούνται ευρέως σε συστήματα που βασίζονται σε τοποθεσία όπως το GIS, παρέχοντας έναν γρήγορο και αποδοτικό τρόπο εύρεσης τοποθεσιών που βρίσκονται κοντά σε ένα σημείο ή αντικείμενο. Ιδίως στις εφαρμογές πραγματικού χρόνου, ο αλγόριθμος πρέπει να είναι όσο γρηγορότερος γίνεται και συνεπώς, ακόμα και μια μικρή βελτίωση στον χρόνο εκτέλεσης του ήδη υπάρχοντος αλγορίθμου είναι πολύ σημαντική.

Επίσης, οι αλγόριθμοι τύπου k-κοντινότερων γειτόνων μπορούν να εφαρμοστούν επιτυχώς στα προβλήματα ιατρικής διάγνωσης. Για παράδειγμα μπορούν να χρησιμοποιηθούν για τη διάγνωση των καρκινικών κυττάρων του στήθους εντοπίζοντας παρόμοια καρκινικά κύτταρα [9]. Επίσης έχει αποδειχτεί ότι στην ιατρική εξόρυξη δεδομένων ο kNN αλγόριθμος αυξάνει την ακρίβεια της διάγνωσης και βρίσκει τις κρυφές πληροφορίες από τις ιατρικές καταγραφές πιο αποδοτικά. Αυτού του είδους τα δεδομένα περιέχουν κρυμμένα πρότυπα και σχέσεις, τα οποία μπορούν να οδηγήσουν σε πιο σωστή διάγνωση και συνεπώς την πιο αποτελεσματική θεραπεία [10].

Τέλος, σε όποιο πρόβλημα χρειάζεται classification, οι αλγόριθμοι k-κοντινότερων γειτόνων μπορούν να λύσουν το πρόβλημα. Άλλες χρήσεις που έχουν είναι στην υπολογιστική γεωμετρία, για data reduction, στα high dimensional spaces, κ.ά.

## 1.5 ΔΟΜΗ ΤΗΣ ΕΡΓΑΣΙΑΣ

Σε αυτή την εργασία εξετάζονται αλγόριθμοι που απαντούν στα ερωτήματα τύπου k-κοντινότερων γειτόνων. Οι 9 αλγόριθμοι που υλοποιήθηκαν συνολικά είναι οι εξής:

- Brute-force για αναζήτηση k-κοντινότερων γειτόνων από όλα τα αντικείμενα που ανήκουν σε ένα dataset.
- Plane-sweep για αναζήτηση k-κοντινότερων γειτόνων από όλα αντικείμενα που ανήκουν σε ένα dataset.
- Plane-sweep με αναζήτηση εναλλάξ για αναζήτηση k-κοντινότερων γειτόνων από όλα τα αντικείμενα που ανήκουν σε ένα dataset.
- Plane-sweep με χωρισμό σε λωρίδες σταθερού ύψους για αναζήτηση k-κοντινότερων γειτόνων από όλα τα αντικείμενα που ανήκουν σε ένα dataset.

- Plane-sweep με χωρισμό σε λωρίδες σταθερού αριθμού αντικειμένων για αναζήτηση k-κοντινότερων γειτόνων από όλα τα αντικείμενα που ανήκουν σε ένα dataset.
- Brute-force για αναζήτηση των k-κοντινότερων γειτόνων ενός dataset από όλα τα αντικείμενα ενός άλλου dataset.
- Plane-sweep με αναζήτηση εναλλάξ για αναζήτηση των k-κοντινότερων γειτόνων ενός dataset από όλα τα αντικείμενα ενός άλλου dataset.
- Plane-sweep με χωρισμό σε λωρίδες σταθερού ύψους για αναζήτηση των k-κοντινότερων γειτόνων ενός dataset από όλα τα αντικείμενα ενός άλλου dataset.
- Plane-sweep με χωρισμό σε λωρίδες σταθερού αριθμού αντικειμένων για αναζήτηση των k-κοντινότερων γειτόνων ενός dataset από όλα τα αντικείμενα ενός άλλου dataset.

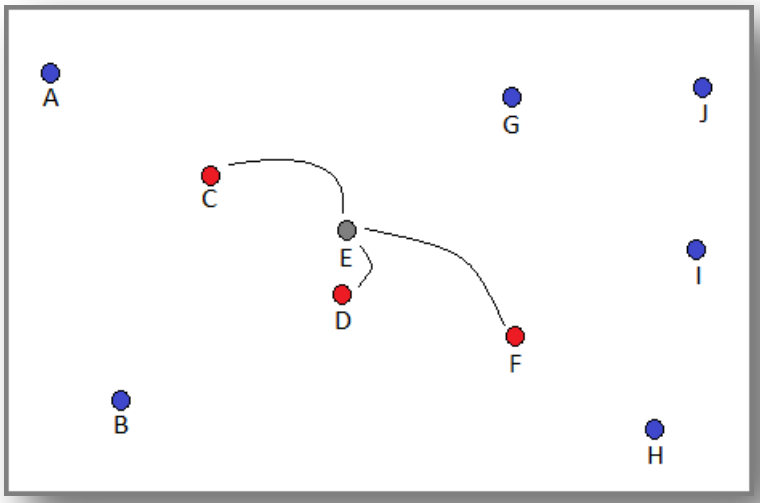
Ο κώδικας γράφτηκε στο ολοκληρωμένο περιβάλλον προγραμματισμού Code::Blocks[11] σε windows, με γλώσσα προγραμματισμού την C και με compiler τον GNU GCC[12] για καλύτερη συμβατότητα με το linux περιβάλλον. Οι παραπάνω αλγόριθμοι θα περιγραφούν εκτενώς στο $2^{ο}$ κεφάλαιο. Στο $3^{ο}$ κεφάλαιο θα δούμε αναλυτικά την υλοποίησή τους. Έπειτα θα παρουσιάσουμε τα πειραματικά αποτελέσματα των αλγορίθμων αυτών στο $4^{ο}$ κεφάλαιο, δοκιμάζοντας διαφορετικές παραμέτρους, ώστε να φανούν τα προτερήματα και τα μειονεκτήματα καθενός από αυτούς. Τέλος, στο $5^{ο}$ κεφάλαιο έχουμε τη σύγκριση όλων αυτών των αλγορίθμων μαζί τα εξαγόμενα συμπεράσματα, αλλά και κάποιους στόχους για τη μελλοντική συνέχιση αυτής της δουλειάς.

# 2. ΠΕΡΙΓΡΑΦΗ ΤΩΝ ΕΡΩΤΗΜΑΤΩΝ ΚΑΙ ΤΩΝ ΥΛΟΠΟΙΗΜΕΝΩΝ ΑΛΓΟΡΙΘΜΩΝ

Σε αυτό το κεφάλαιο θα εξηγήσουμε το είδος των ερωτημάτων που θέλουμε να επιλύσουμε και έπειτα θα περιγράψουμε τους αλγόριθμους που υλοποιήθηκαν.

## 2.1 ΠΕΡΙΓΡΑΦΗ SELF-JOIN K-ΚΟΝΤΙΝΟΤΕΡΩΝ ΓΕΙΤΟΝΩΝ

Το πρώτο ερώτημα που θέλουμε να απαντήσουμε είναι το AkNN self-join, δηλαδή θέλουμε να βρούμε όλους τους k-κοντινότερους γείτονες για κάθε ένα σημείο από ένα σύνολο σημείων που εμπεριέχονται μέσα σε ένα dataset. Στο Σχήμα 1. βλέπουμε ένα παράδειγμα όπου k=3, θέλουμε δηλαδή για κάθε σημείο να βρούμε τα 3 πιο κοντινά σημεία προς αυτό. Έστω ότι έχουμε ήδη βρει τους 3 πιο κοντινούς γείτονες για τα σημεία A,B,C και D και τώρα ελέγχουμε το σημείο E. Παρατηρούμε ότι τα 3 πιο κοντινά σημεία ως προς το σημείο E, είναι το D, C και το F. Κάνουμε αυτή τη διαδικασία για όλα τα σημεία και καταλήγουμε στο τέλος στον πίνακα 1. Σημειώνουμε ότι εάν ένα σημείο είναι ένας από τους 3 κοντινότερους γείτονες ενός άλλου σημείου, αυτό δε σημαίνει και το αντίστροφο. Για παράδειγμα το D είναι ένας από τους 3 κοντινότερους γείτονες του B, το B όμως δεν είναι για το D.



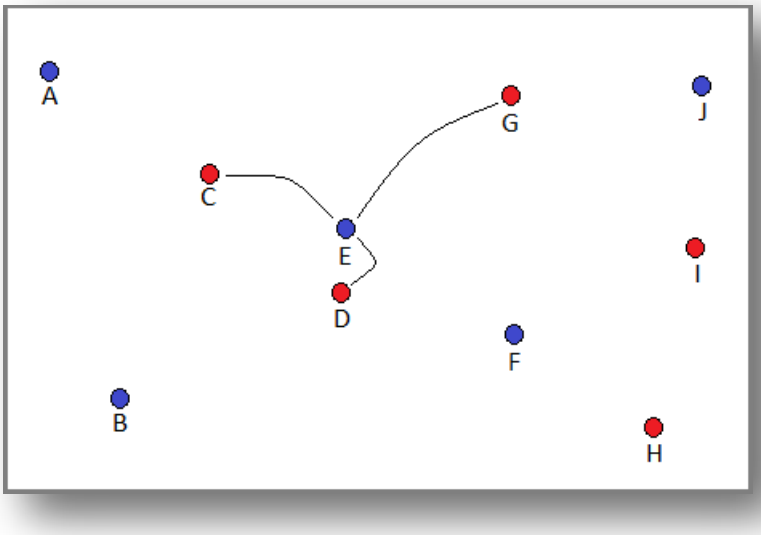

***Σχήμα 1.*** Παράδειγμα εύρεσης 3-κοντινότερων γειτόνων self-join.

| **Σημείο** | **Κοντινότεροι γείτονες** |
|---|---|
| A | B C E |
| B | C D E |
| C | A D E |
| D | C E F |
| E | C D F |
| F | D E H |
| G | E I J |
| H | D F I |
| I | F H J |
| J | G I F |

***Πίνακας 1***. Παράδειγμα αποθήκευσης των 3-κοντινότερων γειτόνων self-join.

## 2.2 ΠΕΡΙΓΡΑΦΗ JOIN K-ΚΟΝΤΙΝΟΤΕΡΩΝ ΓΕΙΤΟΝΩΝ

Το δεύτερο ερώτημα που θέλουμε να απαντήσουμε είναι το AkNN join, δηλαδή για κάθε σημείο από ένα dataset A θέλουμε να βρούμε τα k πιο κοντινά σημεία ως προς αυτό, τα οποία ανήκουν σε ένα dataset B. Στο Σχήμα 2. βλέπουμε ένα παράδειγμα όπου και πάλι k=3, θέλουμε δηλαδή για κάθε μπλε σημείο (dataset A) να βρούμε τα 3 πιο κοντινά κόκκινα σημεία (dataset B) προς αυτό. Έστω ότι έχουμε ήδη βρει τους 3 πιο κοντινούς γείτονες των μπλε σημείων A και B και ψάχνουμε του E. Φαίνεται ότι τα κόκκινα σημεία που βρίσκονται πιο κοντά στο E είναι το C, B και G. Οπότε, κάνοντας την ίδια διαδικασία για όλα τα μπλε σημεία καταλήγουμε στον Πίνακα 2. Σημειώνουμε ότι τα dataset A και B είναι ανεξάρτητα μεταξύ τους. Μπορεί το ένα να έχει διαφορετικό αριθμό σημείων από το άλλο.




***Σχήμα 2.*** Παράδειγμα εύρεσης 3-κοντινότερων γειτόνων join

| **Σημείο** | **Κοντινότεροι γείτονες** |
|---|---|
| A | C D G |
| B | C D G |
| E | C D G |
| F | D H I |
| J | G H I |

***Πίνακας 2.*** Παράδειγμα αποθήκευσης των 3-κοντινότερων γειτόνων join

## 2.3 ΠΕΡΙΓΡΑΦΗ BRUTE-FORCE (SELF-JOIN & JOIN)

Ο αλγόριθμος brute-force υλοποιήθηκε για self-join και για join ερωτήματα. Είναι ο πιο απλός αλγόριθμος προγραμματιστικά αλλά ο πιο χρονοβόρος εκτελεστικά καθώς κάνει πολλούς περιττούς ελέγχους. Πιο αναλυτικά, για κάθε σημείο ψάχνει όλα τα υπόλοιπα σημεία και κάθε φορά κρατάει τα k πιο κοντινά μέχρι να τα ελέγξει όλα. Επομένως για self-join ερωτήματα όπου στο dataset υπάρχουν n σημεία, ο brute-force θα κάνει $n^2$ ελέγχους. Αντίστοιχα, στα ερωτήματα τύπου join με dataset A που περιέχει n σημεία και dataset B που περιέχει m σημεία οι έλεγχοι θα είναι n*m. Εύλογα καταλαβαίνει κανείς ότι ο brute-force αλγόριθμος δεν είναι καθόλου αποδοτικός, όμως στα πλαίσια της εργασίας αυτής ήταν ιδιαίτερα χρήσιμος καθώς είμαστε σίγουροι ότι βγάζει τα σωστά αποτελέσματα (αφού κάνει όλες τις συγκρίσεις που μπορούν να γίνουν). Έτσι χρησιμοποιήθηκε για τον έλεγχο όλων των υπόλοιπων αλγορίθμων.

Ας γυρίσουμε πάλι στον Πίνακα 1. Έστω λοιπόν ότι ελέγχουμε το σημείο E. Θα αρχίσουμε να εξετάζουμε τα σημεία του dataset ξεκινώντας από το σημείο A. Αφού το E δεν έχει ακόμα κανέναν κοντινότερο γείτονα, το A θα αποθηκευτεί. Έπειτα συνεχίζουμε με το B και το C τα οποία αποθηκεύονται και αυτά. Τώρα, έχουμε βρει ήδη 3 γείτονες του E, οπότε για κάθε νέο σημείο που ελέγχουμε πρέπει να εξετάζουμε αν η απόστασή του είναι μικρότερη από τον πιο μακρινό – ήδη αποθηκευμένο – γείτονα και αν ναι τον αντικαθιστούμε, ενώ αν όχι προχωράμε στον έλεγχο του επόμενου σημείου. Συνεχίζουμε, λοιπόν, με τον έλεγχο του σημείου D. Παρατηρούμε ότι η απόστασή του από το σημείο E είναι μικρότερη από αυτή του μακρινότερου αποθηκευμένου γείτονα του E, δηλαδή το A. Συνεπώς θα γίνει αντικατάσταση του σημείου A με το σημείο D. Συνεχίζοντας με αυτή τη λογική, το F θα αντικαταστήσει το B, και μετά όλοι οι επόμενοι έλεγχοι των σημείων G, H, I και J δε θα προκαλέσουν κάποια αντικατάσταση, καθώς οι αποστάσεις τους από το σημείο E, δεν είναι μικρότερες από την απόσταση του F από το E.

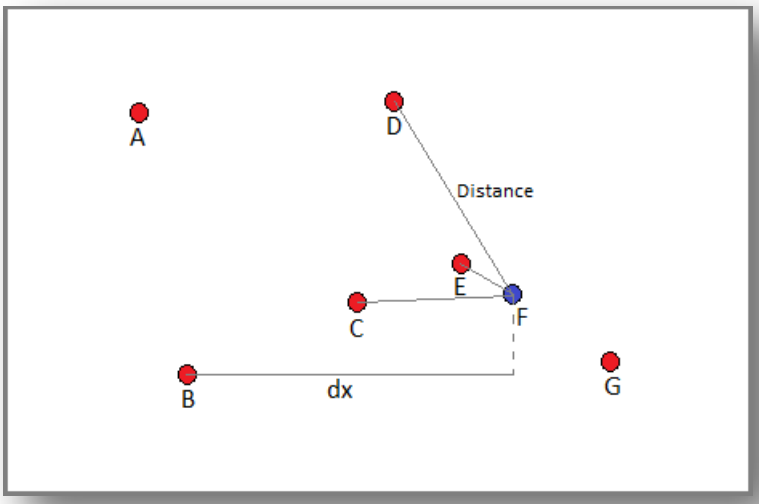
Η ίδια λογική ακολουθείται και στον brute-force join αλγόριθμο. Η διαφορά εδώ είναι ότι για κάθε σημείο του dataset A ελέγχουμε όλα τα σημεία του dataset B.

## 2.4 ΠΕΡΙΓΡΑΦΗ PLANE-SWEEP (SELF-JOIN)

Ο αλγόριθμος Plane-Sweep υλοποιήθηκε μόνο για self-join ερωτήματα, καθώς στην πορεία φάνηκε ότι μπορεί να βελτιωθεί περαιτέρω, όπως και έγινε με την variation του, Plane-Sweep Alternately. Η λογική του απλού Plane-Sweep αλγορίθμου είναι σχετικά απλή. Αρχικά, κάνουμε μια ταξινόμηση όλων των σημείων ως προς το x. Έπειτα για κάθε σημείο που ελέγχουμε, εξετάζουμε ως πιθανούς γείτονες τα σημεία που βρίσκονται πιο κοντά του ως προς το x. Η ταξινόμηση που κάναμε μας βοηθάει στο να βρούμε τα πιο κοντινά – ως προς το x – σημεία.

Πιο συγκεκριμένα, στην αρχή για κάθε σημείο ελέγχουμε τα σημεία που βρίσκονται στα αριστερά του. Αφού βρούμε k γείτονες, συνεχίζουμε το ψάξιμο ελέγχοντας κάθε φορά εάν η απόσταση dx του νέου σημείου που βρήκαμε σε σχέση με το σημείο που ελέγχουμε, είναι μεγαλύτερη από την απόσταση του μέχρι τώρα πιο μακρινού αποθηκευμένου γείτονα. Εάν είναι όντως μεγαλύτερη, δε χρειάζεται να ψάξουμε για πιο μακρινά x, γιατί είμαστε σίγουροι ότι έχουμε ήδη βρει τους πιο κοντινούς γείτονες από την αριστερή μεριά. Έπειτα κάνουμε την ίδια διαδικασία και από τη δεξιά μεριά. Με αυτόν τον τρόπο αποφεύγουμε ελέγχους με σημεία που βρίσκονται μακριά από το σημείο που εξετάζουμε, πράγμα που μας κάνει τον αλγόριθμο σαφώς πιο αποδοτικό από τον brute-force.

Ας δούμε το παράδειγμα στον Πίνακα 3. Έχει γίνει ταξινόμηση ως προς το x οπότε έχουμε τα σημεία μας αποθηκευμένα ως (A, B, C, D, E, F, G). Έστω, λοιπόν, ότι θέλουμε να βρούμε τους 3 κοντινότερους γείτονες του F. Ξεκινάμε πρώτα από τα αριστερά και ελέγχουμε τον κοντινότερο ως προς x γείτονά του, δηλαδή το E. Επειδή ακόμα δεν έχει βρεθεί κάποιος γείτονας αποθηκεύουμε το E χωρίς επιπλέον ελέγχους. Το ίδιο κάνουμε και για τα σημεία D και C. Έπειτα ελέγχουμε το σημείο B. Θα εξετάσουμε αν το dx μεταξύ B και F, είναι μεγαλύτερο από την απόσταση του D και F (Distance). Παρατηρούμε ότι το dx είναι όντως μεγαλύτερο από το distance, οπότε σταματάμε την αναζήτηση από τα αριστερά. Είμαστε σίγουροι ότι όλα τα υπόλοιπα σημεία (όπως το A) θα απέχουν περισσότερο από τους ήδη αποθηκευμένους γείτονες, καθώς το dx τους σε σχέση με το F συνεχώς θα αυξάνεται. Έπειτα ξεκινάμε τον έλεγχο από τα δεξιά. Παρατηρούμε ότι το σημείο G έχει μικρότερο dx από το distance, οπότε θα γίνει ο έλεγχος κανονικά.



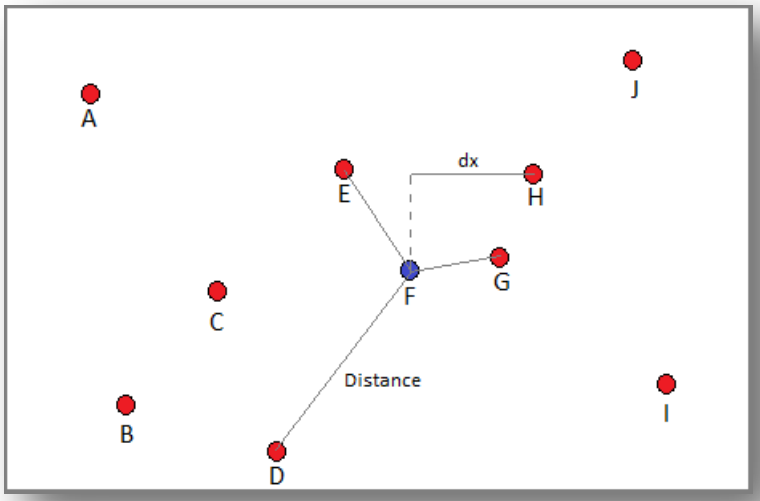

***Σχήμα 3.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με Plane-Sweep.

Στο παράδειγμα αυτό, καταφέραμε να αποφύγουμε μια σύγκριση. Δεν ελέγξαμε καθόλου εάν το σημείο A είναι πιθανός κοντινότερος γείτονας για το F. Σε προβλήματα μεγάλων δεδομένων οι συγκρίσεις ελαττώνονται σε πολύ μεγαλύτερο βαθμό και αυτό έχει όφελος στην αποδοτικότητα. Ωστόσο, υπάρχουν ακόμα πιο αποδοτικές λύσεις και γι αυτό υλοποιήθηκε και ο αλγόριθμος Plane Sweep Alternately.

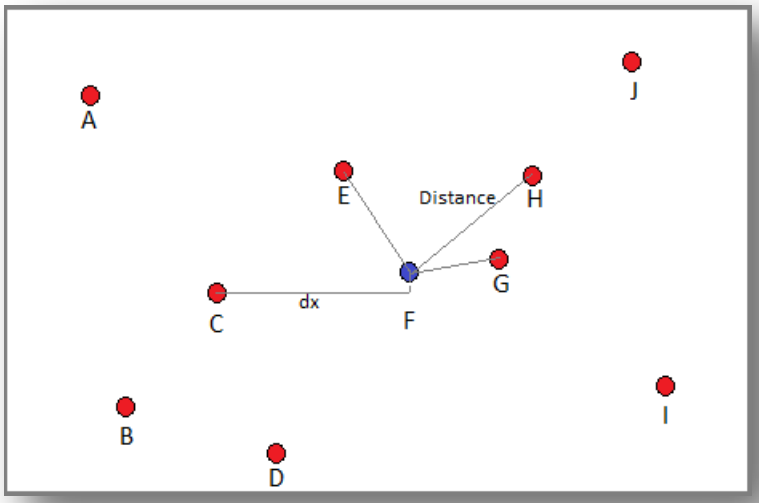
## 2.5 ΠΕΡΙΓΡΑΦΗ PLANE-SWEEP ALTERNATELY (SELF-JOIN & JOIN)

Ο αλγόριθμος Plane-Sweep Alternately υλοποιήθηκε για self-join και join ερωτήματα. Είναι μια παραλλαγή του απλού Plane-Sweep μόνο που εδώ αντί να ελέγχουμε πρώτα στο αριστερό και όταν τελειώσει εκεί η αναζήτηση να ελέγχουμε στο δεξί κομμάτι, ελέγχουμε ένα σημείο από τα αριστερά και ένα σημείο από τα δεξιά εναλλάξ. Κατά τον ίδιο τρόπο ξεκινάμε με μια ταξινόμηση ως προς x, και στη συνέχεια ελέγχουμε εναλλάξ τα σημεία, που είναι υποψήφια για κοντινότεροι γείτονες.

Για να γίνει κατανοητή η λογική του αλγόριθμου, ας δούμε το παράδειγμα αναζήτησης 3 κοντινότερων γειτόνων self-join του Σχήματος 4.1. Μετά την ταξινόμηση ως προς x έχουμε τα σημεία μας αποθηκευμένα ως (A, B, C, D, E, F, G, H, I, J) όπως φαίνονται στο σχήμα. Για να βρούμε τους 3 κοντινότερους γείτονες του σημείου F ξεκινάμε από τα αριστερά και προσθέτουμε το σημείο E, αφού η λίστα με τους κοντινότερους γείτονες είναι άδεια. Έπειτα, ελέγχουμε στα δεξιά και με την ίδια λογική αποθηκεύουμε το σημείο G. Κατόπιν, επιστρέφουμε στην αριστερή μεριά και αποθηκεύουμε το σημείο D. Τώρα έχουμε ήδη βρει 3 κοντινούς γείτονες (E, G, D), οπότε για τα υπόλοιπα σημεία θα κάνουμε ελέγχους απόστασης. Επιστρέφουμε, λοιπόν, στη δεξιά μεριά και ελέγχουμε το σημείο H. Παρατηρούμε ότι το dx του H με το F είναι μικρότερο από το distance του πιο μακρινού αποθηκευμένου κοντινότερου σημείου (F), οπότε δε σταματάει την αναζήτηση. Αφού βρεθεί το distance του H με το F, ελέγχουμε εάν είναι μικρότερη από το distance του D με το F. Πράγματι είναι, οπότε αντικαθιστούμε το D με το H. Η λίστα των αποθηκευμένων γειτόνων του F έχει διαμορφωθεί πλέον στην (E, G, H) (Σχήμα 4.2.). Συνεχίζουμε τον έλεγχο εναλλάξ και πάλι, οπότε πάμε στην αριστερή μεριά. Το σημείο C έχει dx μεγαλύτερο από το distance του H με το F. Συνεπώς σταματάει η αναζήτηση από την αριστερή μεριά. Έπειτα ελέγχουμε στη δεξιά μεριά το σημείο J. Επίσης το dx του J είναι μεγαλύτερο από το distance του H, οπότε η αναζήτηση τελειώνει και από τα δεξιά. Συνεπώς για το σημείο F οι 3 κοντινότεροι γείτονές του είναι οι (E, G, H).




***Σχήμα 4.1.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep Alternately. (1)




***Σχήμα 4.2.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep Alternately. (2)

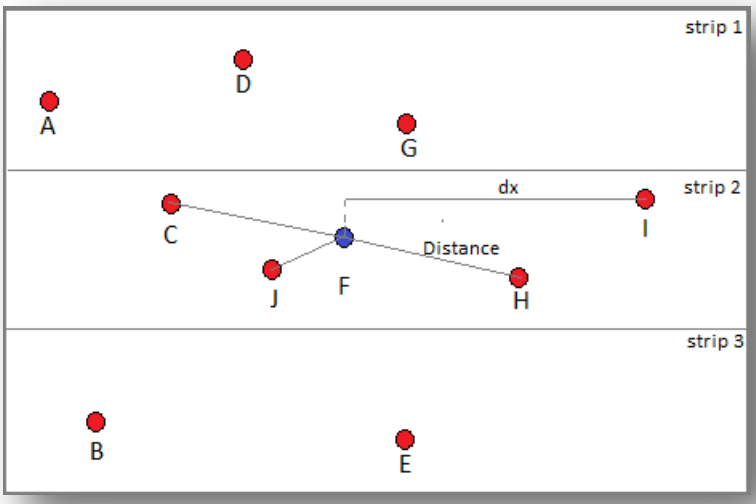
Εάν για αυτό το παράδειγμα χρησιμοποιούσαμε τον απλό Plane Sweep αλγόριθμο θα ελέγχαμε πρώτα τα (E, D, C) τα οποία και θα αποθηκεύαμε, έπειτα στον έλεγχο θα διακόπταμε την αναζήτηση στο B από τα αριστερά και στη συνέχεια θα ελέγχαμε και τα G και H. Συνολικά δηλαδή θα εξετάζαμε 5 σημεία, ενώ στον Plane Sweep Alternately εξετάσαμε μόνο 4, τα (E, G, D, H). Συνεπώς, φαίνεται ότι η παραλλαγή που έγινε, είναι πιο αποδοτική.

## 2.6 ΠΕΡΙΓΡΑΦΗ PLANE-SWEEP WITH FIXED STRIPS (SELF-JOIN & JOIN)

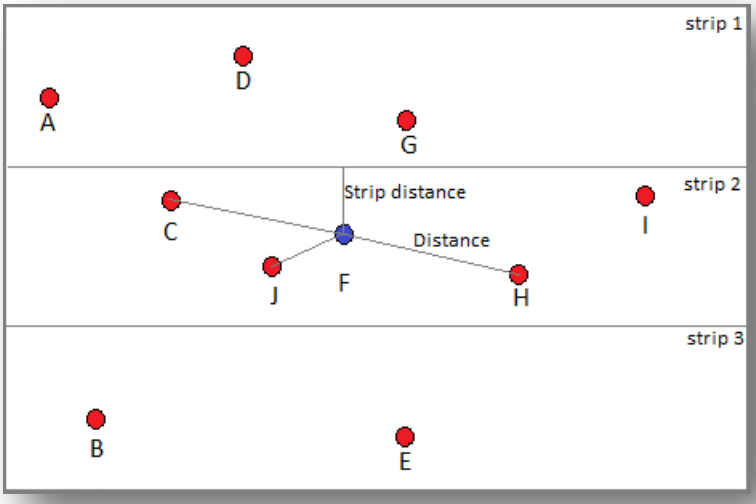
Ο αλγόριθμος Plane-Sweep with Fixed Strips υλοποιήθηκε για self-join και join ερωτήματα. Εδώ εντάσσουμε τη νέα έννοια των λωρίδων. Χωρίζουμε δηλαδή το χώρο σε ένα συγκεκριμένο αριθμό λωρίδων και εργαζόμαστε σε κάθε λωρίδα ξεχωριστά. Αυτή η λογική, εκτός από το γεγονός ότι μειώνει τους ελέγχους που θα κάνουμε, ανοίγει το δρόμο για παράλληλη υλοποίηση, αφού κάθε λωρίδα μπορεί να διαχειριστεί αυτόνομα. Βέβαια, χρειάζεται προσεκτική επιλογή των παραμέτρων και συγκεκριμένα του αριθμού λωρίδων, για να ελαχιστοποιηθεί ο χρόνος εκτέλεσης. Να σημειωθεί εδώ ότι ο βέλτιστος αριθμός λωρίδων εξαρτάται από πολλές παραμέτρους, όπως το πλήθος των σημείων και η διασπορά τους, με αποτέλεσμα για διαφορετικά προβλήματα να έχουμε και διαφορετικό βέλτιστο αριθμό λωρίδων.

Η λογική για τον Plane-Sweep με σταθερό αριθμό λωρίδων ξεκινά με τον χωρισμό του χώρου μας δε λωρίδες ίσου ύψους. Έπειτα, κατατάσσουμε κάθε σημείο στη λωρίδα που ανήκει και κάνουμε από μία ταξινόμηση ως προς x σε κάθε λωρίδα. Στη συνέχεια, για κάθε σημείο ελέγχουμε πρώτα για κοντινότερους γείτονες στη λωρίδα που ανήκει με τη λογική Plane-Sweep Alternately και στη συνέχεια εάν χρειαστεί, ελέγχουμε και στις γειτονικές του λωρίδες, μέχρις ότου να βρεθεί σημείο που το dx του είναι μεγαλύτερο από την απόσταση του πιο μακρινού αποθηκευμένου γείτονα.

Ας δούμε όμως το παράδειγμα του Σχήματος 5.1. για να κατανοήσουμε καλύτερα τον αλγόριθμο. Ψάχνουμε να βρούμε τους 3 κοντινότερους γείτονες για όλα τα σημεία σε αυτό το self-join ερώτημα. Έχουμε χωρίσει τα σημεία σε 3 λωρίδες ίσου ύψους και έστω ότι σε αυτό το στιγμιότυπο ψάχνουμε τους 3 κοντινότερους γείτονες του σημείου F. Αρχικά, έχει γίνει η ταξινόμηση κάθε λωρίδας ως προς x, με αποτέλεσμα να έχουμε: λωρίδα 1 (A, D, G), λωρίδα 2 (C, J, F, H, I), λωρίδα 3 (B, E). Έπειτα, με τη λογική του Plane-Sweep Alternately αποθηκεύουμε ως κοντινούς γείτονες τα 3 πρώτα σημεία που βρίσκουμε ξεκινώντας από αριστερά του F και εναλλάξ δηλαδή τα (J, H, C). Στη συνέχεια, ελέγχουμε το σημείο I. Το dx του I όμως είναι μεγαλύτερο από το distance του χειρότερου γείτονα (H), οπότε σταματάμε την αναζήτηση στα δεξιά. Επίσης, σταματάει και η αναζήτηση στα αριστερά, καθώς δεν υπάρχει άλλο σημείο προς έλεγχο, έτσι ξεκινάμε το ψάξιμο σε νέες λωρίδες. Ξεκινάμε από την επάνω λωρίδα (No.1) και ελέγχουμε αν η απόσταση του F από τη λωρίδα 1 είναι μεγαλύτερη από την πιο μακρινή αποθηκευμένη απόσταση γείτονα (Σχήμα 5.2). Παρατηρούμε ότι δεν είναι οπότε ξεκινάμε την αναζήτηση στη λωρίδα 1.




***Σχήμα 5.1.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Fixed Strips με χωρισμό σε 3 λωρίδες. (1)

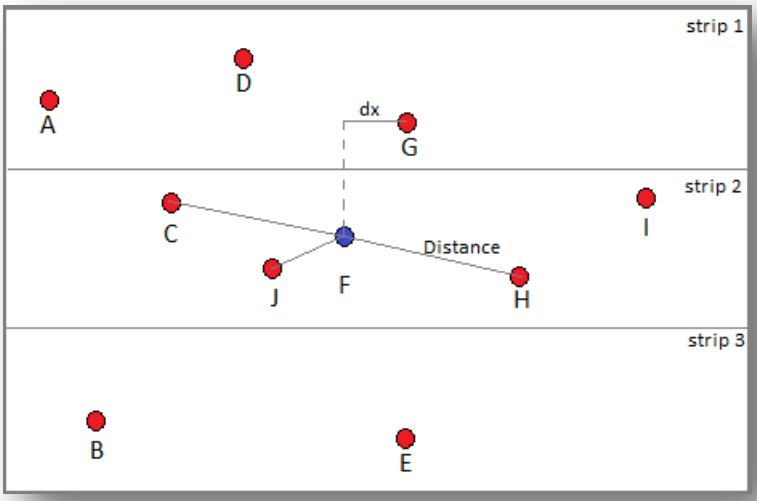



***Σχήμα 5.2.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Fixed Strips με χωρισμό σε 3 λωρίδες. (2)

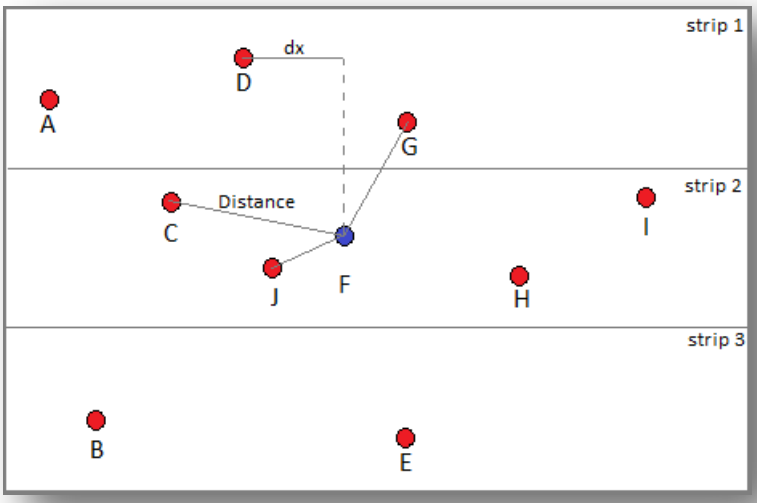
Βρισκόμαστε, λοιπόν, στην λωρίδα 1 και βρίσκουμε το σημείο που είναι πιο κοντά ως προς το x με το σημείο F, του οποίου ψάχνουμε τους πιο κοντινούς γείτονές του. Το σημείο αυτό είναι το σημείο G, οπότε ξεκινάμε από αυτό όπως φαίνεται και στο Σχήμα 5.3. Το dx του G είναι μικρότερο από το distance του πιο μακρινού αποθηκευμένου γείτονα (H), οπότε ξεκινάμε τον έλεγχο του distance του G. Είναι εμφανώς πιο μικρό από το distance του H, οπότε αντικαθιστούμε το H με το G. Συνεχίζουμε τον έλεγχο στη λωρίδα 1 με τη λογική Plane-Sweep Alternately ελέγχοντας από αριστερά το επόμενο σημείο, το D (Σχήμα 5.4.). Το dx του είναι μικρότερο από το χειρότερο distance, οπότε ελέγχουμε το distance του D από το F. Παρατηρούμε ότι η απόστασή του από το F είναι μεγαλύτερη από τη χειρότερη αποθηκευμένη απόσταση οπότε δεν το συμπεριλαμβάνουμε στους γείτονες. Συνεχίζουμε με τον έλεγχο του σημείου A. Επειδή το dx του είναι μεγαλύτερο από το χειρότερο distance, σταματάμε τον έλεγχο εδώ και τελειώνουμε την αναζήτηση στη λωρίδα 1.

Στη συνέχεια, θα ελέγξουμε τη λωρίδα 3. Αρχικά, εξετάζουμε εάν η απόσταση του σημείου F από τη λωρίδα 3 είναι μικρότερη από την απόσταση του πιο μακρινού αποθηκευμένου γείτονα (Σχήμα 5.5.) Όντως είναι, οπότε ξεκινά η αναζήτηση στη λωρίδα 3. Βρίσκουμε το σημείο του οποίου το x είναι πιο κοντά στο x του F και ξεκινάμε από αυτό (E). Το E, λοιπόν, έχει dx μικρότερο από το χειρότερο distance, οπότε το εξετάζουμε (Σχήμα 5.6.). Παρατηρούμε, όμως, ότι η απόστασή του είναι μεγαλύτερη από τη χειρότερη αποθηκευμένη απόσταση, οπότε δεν το συμπεριλαμβάνουμε στους κοντινότερους γείτονες. Συνεχίζουμε με τη λογική Plane-Sweep Alternately από αριστερά στο σημείο A. Είναι εμφανές ότι το dx του A είναι μεγαλύτερο από το χειρότερο distance, οπότε δεν εξετάζουμε το A και σταματάμε την αναζήτηση εδώ, έχοντας βρει τους 3 πιο κοντινούς γείτονες του F, δηλαδή τους (C, J, G).

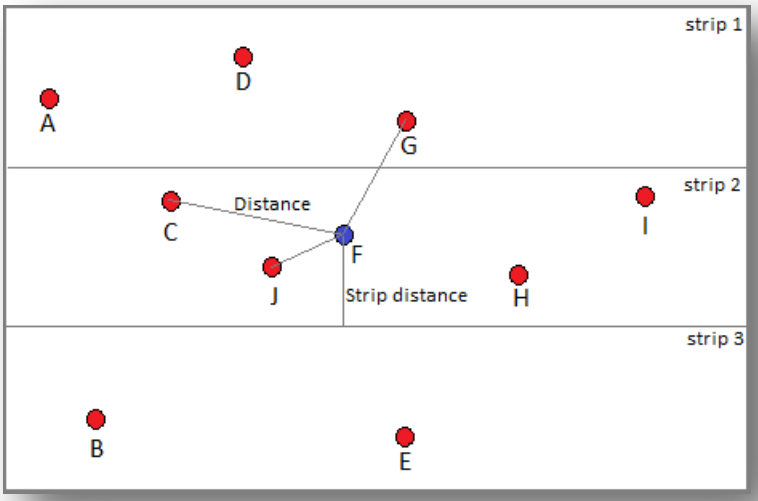
Στο παραπάνω παράδειγμα είδαμε μια ειδική περίπτωση για να γίνει κατανοητό πώς λειτουργεί ο αλγόριθμος σε όλες τις περιπτώσεις. Στη μέση περίπτωση, και με σωστά επιλεγμένο αριθμό λωρίδων, οι αναζητήσεις σε άλλες λωρίδες μειώνονται αισθητά, και γλιτώνουμε πολλούς ελέγχους και υπολογισμούς.



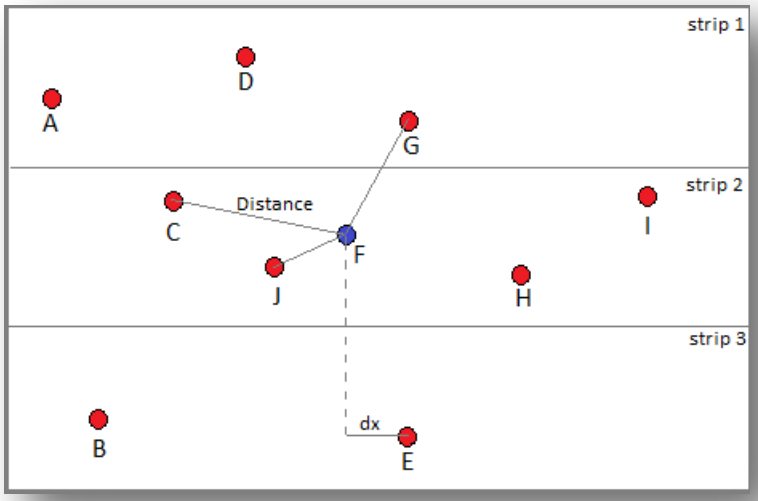

***Σχήμα 5.3.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Fixed Strips με χωρισμό σε 3 λωρίδες. (3)




***Σχήμα 5.4.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Fixed Strips με χωρισμό σε 3 λωρίδες. (4)




***Σχήμα 5.5.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Fixed Strips με χωρισμό σε 3 λωρίδες. (5)



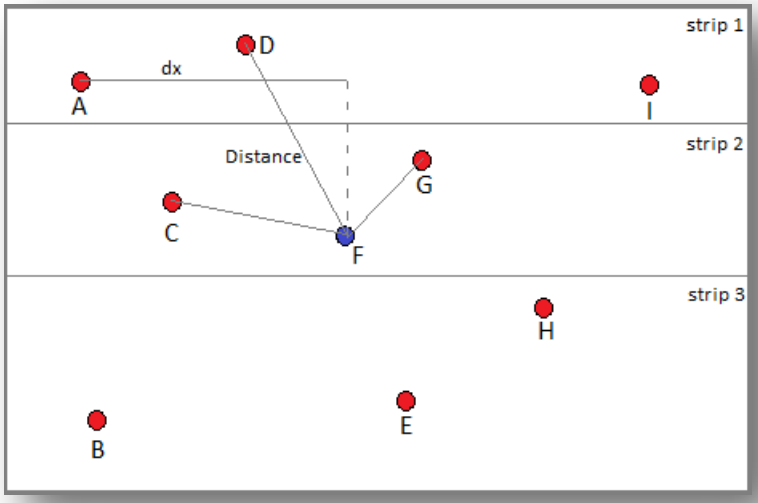

***Σχήμα 5.6.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Fixed Strips με χωρισμό σε 3 λωρίδες. (6)

## 2.7 ΠΕΡΙΓΡΑΦΗ PLANE-SWEEP WITH FREE STRIPS (SELF-JOIN & JOIN)

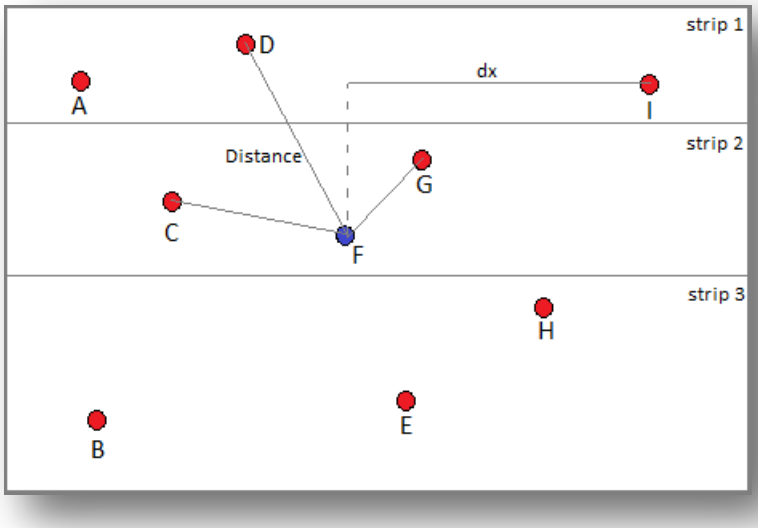
Ο αλγόριθμος Plane-Sweep with Free Strips υλοποιήθηκε για self-join και join ερωτήματα. Παραμένουμε στη λογική με τις λωρίδες, αλλά εδώ εξετάζουμε μία νέα παράμετρο. Αντί η κάθε λωρίδα να έχει σταθερό ύψος, τροποποιούμε τον προηγούμενο αλγόριθμο ώστε η κάθε λωρίδα να έχει ίδιο αριθμό σημείων. Αυτή η τροποποίηση μας βοηθάει να διαχειριστούμε καλύτερα τα δεδομένα που δεν είναι ομοιόμορφα κατανεμημένα, αφού όσο πιο πυκνά είναι τα σημεία έχουμε περισσότερες λωρίδες σε εκείνα τα σημεία, ενώ όσο πιο αραιά είναι, χρειαζόμαστε λιγότερες.

Αρχικά, η λογική του Plane-Sweep με λωρίδες ίσου αριθμού σημείων είναι η ταξινόμηση όλων των σημείων ως προς y. Αφού τα έχουμε όλα ταξινομημένα, για κάθε m σημεία δημιουργούμε μία λωρίδα. Συνεπώς, ο αριθμός των λωρίδων θα είναι [n/m], όπου n ο συνολικός αριθμός των σημείων και m το πλήθος των σημείων σε κάθε λωρίδα. Εδώ να σημειώσουμε ότι η τελευταία λωρίδα συνήθως θα έχει διαφορετικό αριθμό σημείων, εκτός αν η διαίρεση n/m έχει αποτέλεσμα ακέραιο. Αφού έχουμε δημιουργήσει τις λωρίδες μας λοιπόν, κατά τα γνωστά, ψάχνουμε για κοντινούς γείτονες με λογική Plane-Sweep Alternately πρώτα στη λωρίδα όπου ανήκει το σημείο αναζήτησης και έπειτα – εάν χρειαστεί – και σε άλλες λωρίδες.

Εδώ, θα εξετάσουμε το παράδειγμα του Σχήματος 6.1. Έστω ότι το m = 3, θέλουμε δηλαδή κάθε λωρίδα να έχει 3 σημεία και k = 3, θέλουμε δηλαδή να βρούμε τους 3 κοντινότερους γείτονες κάθε σημείου. Αρχικά, ταξινομούμε τα σημεία ως προς y, και τα ταξινομούμε στις λωρίδες. Για τα 9 σημεία που έχουμε εδώ θα χρειαστούμε 3 λωρίδες. Έπειτα ταξινομούμε τα σημεία σε κάθε λωρίδα ως προς x με αποτέλεσμα να έχουμε: λωρίδα 1 (A, D, I), λωρίδα 2 (C, F, G), λωρίδα 3 (B, E, H). Στο στιγμιότυπο που βλέπουμε ψάχνουμε τους 3 κοντινότερους γείτονες του σημείου F. Η πρώτη αναζήτηση θα γίνει μέσα στη λωρίδα που ανήκει το F, δηλαδή τη 2. Προσθέτουμε τα σημεία C και G, αφού η λίστα κοντινότερων γειτόνων έχει άδειες θέσεις. Έπειτα ψάχνουμε στην πιο πάνω λωρίδα (1) και βρίσκουμε το σημείο που έχει το πιο κοντινό x σε σχέση με το F. Αυτό το σημείο είναι το D, το οποίο και προσθέτουμε αφού η λίστα των γειτόνων έχει κενές θέσεις και συνεχίζουμε την αναζήτηση στη λωρίδα 1. Σύμφωνα με τη λογική Plane-Sweep Alternately ελέγχουμε το σημείο A. Το dx του είναι μεγαλύτερο από το distance τού χειρότερου γείτονα οπότε σταματάμε την αναζήτηση από τα αριστερά. Το ίδιο κάνουμε και με το I από τα δεξιά (Σχήμα 6.2.). Επειδή το dx του είναι μεγαλύτερο από την απόσταση του χειρότερου γείτονα διακόπτουμε την αναζήτηση και από τα δεξιά, οπότε σταματάει συνολικά η αναζήτηση στη λωρίδα 1.



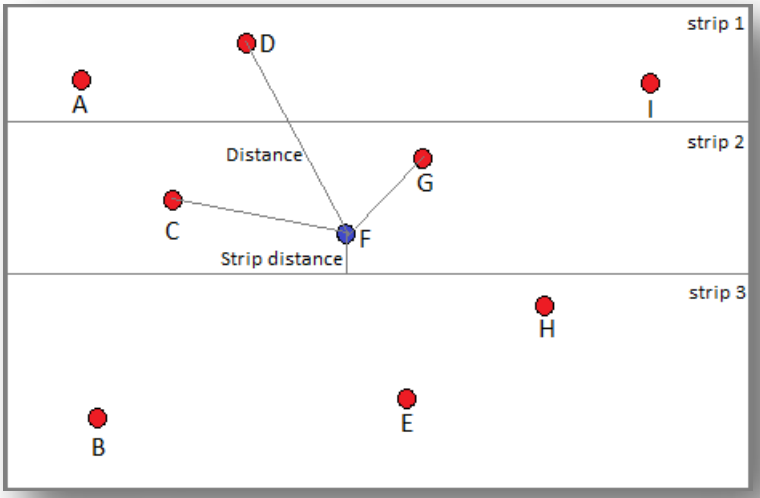

***Σχήμα 6.1.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Free Strips με χωρισμό σε λωρίδες 3 σημείων. (1)



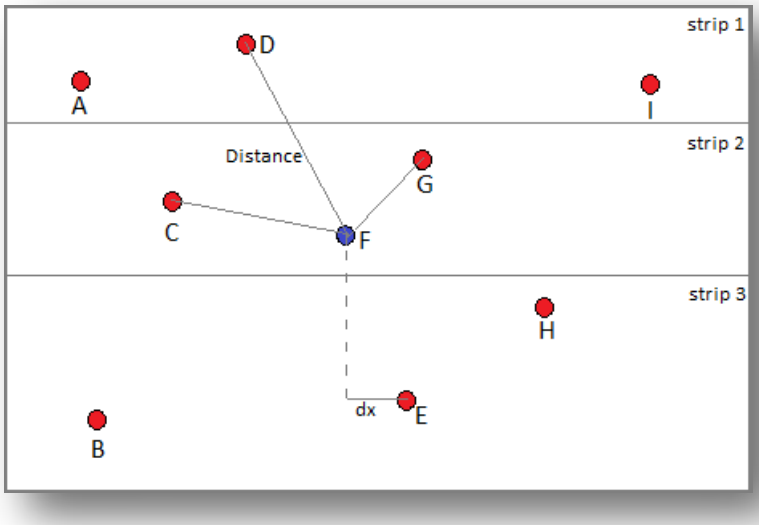

***Σχήμα 6.2.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Free Strips με χωρισμό σε λωρίδες 3 σημείων. (2)

Συνεχίζουμε λοιπόν με την αναζήτηση στην κάτω λωρίδα (3). Πρώτα εξετάζουμε εάν χρειάζεται να αναζητήσουμε εκεί, ελέγχοντας εάν η απόσταση του F από τη λωρίδα είναι μικρότερη από την απόσταση του χειρότερου αποθηκευμένου γείτονα (Σχήμα 6.3.). Πράγματι, είναι μικρότερη, οπότε ξεκινάμε να εξετάζουμε τα σημεία της $3^{ης}$ λωρίδας. Βρίσκουμε αυτό που έχει κοντινότερο x ως προς το x του F. Το σημείο αυτό είναι το E, οπότε ελέγχουμε το dx του σε σχέση με το distance του χειρότερου αποθηκευμένου γείτονα (Σχήμα 6.4.). Το dx του E σε σχέση με το F είναι όντως μικρότερο, οπότε αφού ελέγξουμε το distance του E, παρατηρούμε ότι είναι μικρότερο από αυτό του D. Αντικαθιστούμε, λοιπόν, το E με το D. Έπειτα κατά τη λογική Plane-Sweep Alternately ελέγχουμε το σημείο B (Σχήμα 6.5.). Το dx του B είναι μεγαλύτερο από το χειρότερο distance, οπότε σταματάμε την αναζήτηση στα αριστερά της λωρίδας 3. Έπειτα ελέγχουμε στα δεξιά το σημείο H (Σχήμα 6.6.). Παρατηρούμε ότι και αυτό έχει μεγαλύτερο dx, από την απόσταση του F με τον μακρινότερο αποθηκευμένο γείτονά του (E), οπότε η αναζήτηση σταματάει και στα δεξιά. Συνεπώς, έχουμε τελειώσει με όλες τις αναζητήσεις και έχουν βρεθεί οι 3 κοντινότεροι γείτονες για το F, ο οποίοι είναι οι (C, G, E).

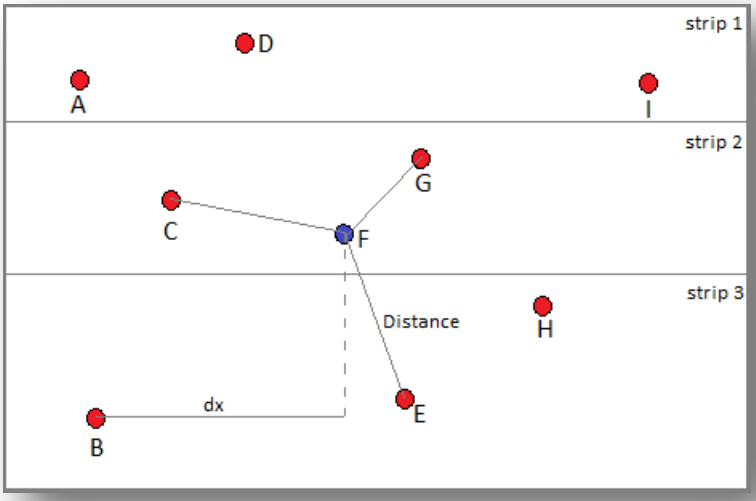
Όπως και στον αλγόριθμο Plane-Sweep with Fixed Strips, έτσι και εδώ, πρέπει να γίνει προσεκτική επιλογή του m, του αριθμού δηλαδή των σημείων που θα ανήκουν σε μια λωρίδα. Με το βέλτιστο m, θα πετύχουμε καλύτερη απόδοση μειώνοντας το χρόνο εκτέλεσης. Να σημειώσουμε όμως, ότι και εδώ, το βέλτιστο m εξαρτάται από άλλες παραμέτρους και δεν είναι σταθερό για τη λύση όλων των προβλημάτων.



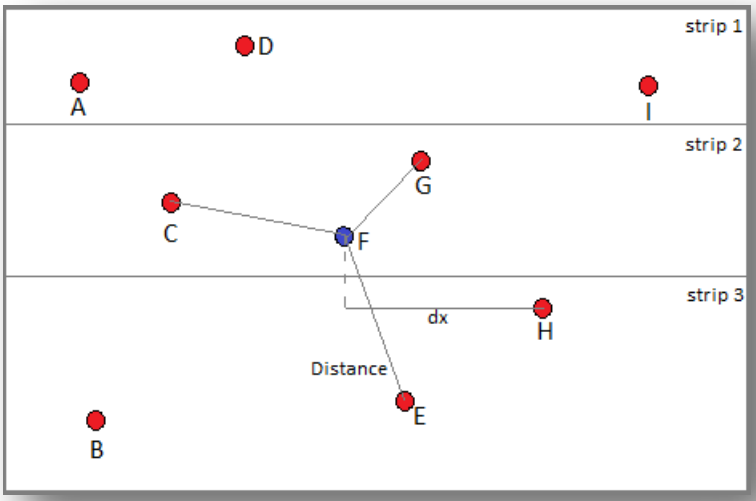

***Σχήμα 6.3.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Free Strips με χωρισμό σε λωρίδες 3 σημείων. (3)




***Σχήμα 6.4.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Free Strips με χωρισμό σε λωρίδες 3 σημείων. (4)




***Σχήμα 6.5.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Free Strips με χωρισμό σε λωρίδες 3 σημείων. (5)




***Σχήμα 6.6.*** Παράδειγμα εύρεσης 3 κοντινότερων γειτόνων με τον Plane-Sweep with Free Strips με χωρισμό σε λωρίδες 3 σημείων. (6)

# 3. ΥΛΟΠΟΙΗΣΗ ΚΩΔΙΚΑ

Σε αυτό το κεφάλαιο θα δούμε την υλοποίηση των αλγορίθμων που εξετάζουμε σε γλώσσα C παραθέτοντας τα αντίστοιχα κομμάτια κώδικα. Θα εξετάσουμε την περίπτωση των join, αλλά με την ίδια λογική έχουν υλοποιηθεί και οι περιπτώσεις self-join.

## 3.1 ΥΛΟΠΟΙΗΣΗ BRUTE-FORCE (JOIN)

Παρακάτω φαίνεται ο κώδικας με τον οποίον υλοποιήθηκε ο brute-force για ερωτήματα τύπου join. Ξεκινώντας από τα αριστερά, ελέγχουμε την απόσταση κάθε σημείου. Σημείωση πως δε χρησιμοποιούμε την πραγματική απόσταση των σημείων, αλλά το τετράγωνο της απόστασης για να γλιτώσουμε την απαιτητική πράξη υπολογισμού της ρίζας από τον τύπο distance = $\sqrt{dx^2 + dy^2}$. Συνεπώς υπολογίζουμε το distance$^2$ αθροίζοντας τα dx$^2$ + dy$^2$. Αφού, λοιπόν, ελέγξουμε κάποιο σημείο και υπολογίσουμε την απόστασή του από το σημείο που ψάχνουμε τους γείτονές του, κοιτάμε αρχικά εάν η λίστα γειτόνων είναι άδεια όπως φαίνεται στη γραμμή 24. Εάν υπάρχουν κενές θέσεις, προσθέτουμε το νέο γείτονα. Σε περίπτωση που η λίστα γειτόνων δεν έχει κενές θέσεις, ελέγχουμε εάν η απόσταση του νέου σημείου είναι μικρότερη από την απόσταση του πιο μακρινού γείτονα, που είναι ήδη αποθηκευμένος στη λίστα. Εάν είναι όντως μικρότερη, προσθέτουμε το νέο γείτονα, και βρίσκουμε τον νέο πιο μακρινό γείτονα τον οποίο βάζουμε στη θέση 1 της λίστας για να κάνουμε τους νέους ελέγχους (γραμμή 36). Συνεχίζουμε αυτή τη διαδικασία μέχρι να τελειώσουν όλα τα σημεία. Στο τέλος, θα έχουμε βρει όλους τους k κοντινότερους γείτονες για κάθε σημείο.

```
for (i=0; i<dataset_points_num; i++) {
    // dataset_point is the point that we are searching for its neighbors
    brute_force_neighbors[i].p = &dataset_point[i];
    for (c=0; c<neighbors_num; c++)
        // initialization
        brute_force_neighbors[i].neighbor[c].distance = 0;
    for (j=0; j<points_num; j++) {
        dx = dataset_point[i].x - point[j].x;
        dy = dataset_point[i].y - point[j].y;
        // in case of the same point, do not add it as a neighbor
        if (dx == 0 && dy == 0)
            continue;
        // new_distance is the distance between the 2 points (i and j) we are checking
        new_distance = dx*dx + dy*dy;
        // max_distance is the saved distance of the point[i] and its max-
distance neighbor
        max_distance = brute_force_neighbors[i].neighbor[0].distance;
        // if no neighbor found yet
        if (max_distance == 0) {
            add_neighbor(brute_force_neighbors[i], point[j]);
        }
        // if the new distance found is lower than the max_distance, add that point as nei
ghbor
        else if (max_distance > new_distance) {
            // not all neighbor slots taken
            if (brute_force_neighbors[i].neighbor[neighbors_num-1].distance == 0) {
                // find the next free slot and save the new neighbor
                for (c=neighbors_num-2; c>=0; c--) {
                    if (brute_force_neighbors[i].neighbor[c].distance != 0) {
                        brute_force_neighbors[i].neighbor[c+1].nearest_neighbor = &point[j
];
                        brute_force_neighbors[i].neighbor[c+1].distance = new_distance;
                        break;
                    }
                }
            }
            // find the max-distance neighbor to replace the old max-
distance neighbor (using array)
            else {
                replace_max_distance_neibhbor_and_add_point(point[i], brute_force_neighbor
s[i]);
            }
        }
        else if (max_distance <= new_distance) {
            // if not all neighbor slots taken add the new neighbor
            if (brute_force_neighbors[i].neighbor[neighbors_num-1].distance == 0) {
                add_neighbor(brute_force_neighbors[i], point[j]);
            }
        }
    }
}
```

## 3.2 ΥΛΟΠΟΙΗΣΗ SIMPLE PLANE SWEEP ALTERNATELY (JOIN)

Παρακάτω φαίνεται ο κώδικας με τον οποίο υλοποιήθηκε ο Simple Plane Sweep Alternately. Εδώ αρχικά ταξινομούμε τα σημεία και των 2 dataset ως προς x και ξεκινάμε την αναζήτηση γειτόνων για καθένα από τα σημεία του input set. Αρχικά, βρίσκουμε το σημείο του training set που είναι πιο κοντά – ως προς τον άξονα των x – στο σημείο που εξετάζουμε του input set (γραμμή 9). Έπειτα, ξεκινάμε την αναζήτηση από εκείνο το σημείο ελέγχοντας κάθε φορά ένα σημείο από τα αριστερά του και ένα σημείο από τα δεξιά του. Το βήμα μας εδώ είναι το j όπως φαίνεται στη γραμμή 21, από την οποία και ξεκινάμε την αναζήτηση. Αρχικά μέχρι να γεμίσει η λίστα γειτόνων αποθηκεύουμε όσα σημεία βρίσκουμε. Εάν γεμίσει η λίστα, αυξάνουμε το βήμα (γραμμή 61) και πλέον αντικαθιστούμε τους νέους γείτονες εάν πληρούν τις προϋποθέσεις απόστασης. Κατά τα γνωστά, ελέγχουμε αριστερά και δεξιά και εάν η απόσταση ενός σημείου είναι μικρότερη από την απόσταση του πιο μακρινού αποθηκευμένου γείτονα της λίστας, αντικαθίσταται από το νέο σημείο και στο heap δέντρο γίνονται οι πράξεις ισορρόπησης. Έτσι, στην κορυφή του δέντρου έχουμε τον πιο μακρινό αποθηκευμένο γείτονα που θα μας βοηθήσει στους ελέγχους. Αφού ελέγξουμε όλα τα σημεία, τελειώνει ο αλγόριθμος, έχοντας βρει όλους τους k κοντινότερους γείτονες για κάθε σημείο του input set.

```
1.  for (i=0; i<dataset_points_num; i++) {
2.      array_length = 0;
```

```
3.      // dataset_point[i] is the point that we are searching for its neighbors
4.      plane_sweep_neighbors[i].p = &dataset_point[i];
5.      // initialization
6.      plane_sweep_neighbors[i].neighbor[0].distance = 0;
7.
8.      // find the closest-to-x point of training set to the input set
9.      pos = binary_search (point, 0, points_num-1, dataset_point[i].x);
10.
11.     dx = dataset_point[i].x - point[pos].x;
12.     dy = dataset_point[i].y - point[pos].y;
13.     // if we have 2 identical points, do not add as neighbor
14.     if (dx!=0 || dy!=0) {
15.         // new_distance is the distance between the 2 points (i and j) we are checking
16.         new_distance = dx*dx + dy*dy;
17.         // add the first point as neighbor
18.         array_length = heap_add(new_distance, point, pos, array_length, plane_sweep_neighb
    ors, i);
19.     }
20.
21.     for (j=1; ; j++) {
22.         // check on the left
23.         if (pos-j>=0) {
24.             dx = dataset_point[i].x - point[pos-j].x;
25.             dy = dataset_point[i].y - point[pos-j].y;
26.             new_distance = dx*dx + dy*dy;
27.             // if we do not have 2 identical points
28.             if (new_distance != 0) {
29.                 // the neighbors list is not full, so add the new point
30.                 array_length = heap_add(new_distance, point, pos-
    j, array_length, plane_sweep_neighbors, i);
31.                 // if after the now all slots are full, check on the right before break
32.                 if (array_length == neighbors_num) {
33.                     if (pos+j<points_num) {
34.                         dx = dataset_point[i].x - point[pos+j].x;
35.                         dy = dataset_point[i].y - point[pos+j].y;
36.                         new_distance = dx*dx + dy*dy;
37.                         max_distance = plane_sweep_neighbors[i].neighbor[0].distance;
38.                         // if the new point's distance is less than the max-
    saved distance
39.                         if ((max_distance > new_distance)) {
40.                             // add the new point and replace the root with the point that
    has the greater distance
41.                             heap_add_and_replace_root(new_distance, point, pos+j, plane_sw
    eep_neighbors, i);
42.                         }
43.                     }
44.                     break;
45.         }}}
46.         // check on the right
47.         if (pos+j<points_num) {
48.             dx = dataset_point[i].x - point[pos+j].x;
49.             dy = dataset_point[i].y - point[pos+j].y;
50.             new_distance = dx*dx + dy*dy;
51.             if (new_distance != 0) {
52.                 // add the new point
53.                 array_length = heap_add(new_distance, point, pos+j, array_length, plane_sw
    eep_neighbors, i);
54.                 if (array_length == neighbors_num)
55.                     break;
56.             }
57.         }
58.     }
59.     max_distance = plane_sweep_neighbors[i].neighbor[0].distance;

60. // now the list is full, so if we find a new neighbor we will have to replace old ones
61.     for (j=j+1; ; j++) {
```

```
62.         // check on the left
63.         if (pos-j>=0) {
64.             dx = dataset_point[i].x - point[pos-j].x;
65.             // if the rest of the points on the right are far away (dx>max_distance), chec
    k only on the left and then break
66.             if ((max_distance < dx*dx)) {
67.                 for ( ;pos+j<points_num; j++) {
68.                     dx = dataset_point[i].x - point[pos+j].x;
69.                     if ((max_distance < dx*dx))
70.                         break;
71.                     dy = dataset_point[i].y - point[pos+j].y;
72.                     new_distance = dx*dx + dy*dy;
73.                     if ((max_distance > new_distance)) {
74.                         heap_add_and_replace_root(new_distance, point, pos+j, plane_sweep_
    neighbors, i);
75.                         max_distance = plane_sweep_neighbors[i].neighbor[0].distance;
76.                     }
77.                 }
78.                 break;
79.             }
80.             dy = dataset_point[i].y - point[pos-j].y;
81.             new_distance = dx*dx + dy*dy;
82.             if ((max_distance > new_distance)) {
83.                 heap_add_and_replace_root(new_distance, point, pos-
    j, plane_sweep_neighbors, i);
84.                 max_distance = plane_sweep_neighbors[i].neighbor[0].distance;
85.             }
86.
87.         }
88.         // check on the right
89.         if (pos+j<points_num) {
90.             dx = dataset_point[i].x - point[pos+j].x;
91.             // if the rest of the points on the left are far away (dx>max_distance), check
     only on the right and then break
92.             if ((max_distance < dx*dx)) {
93.                 for (j=j+1; pos-j>=0; j++) {
94.                     dx = dataset_point[i].x - point[pos-j].x;
95.                     if ((max_distance < dx*dx))
96.                         break;
97.                     dy = dataset_point[i].y - point[pos-j].y;
98.                     new_distance = dx*dx + dy*dy;
99.                     if ((max_distance > new_distance)) {
100.                        heap_add_and_replace_root(new_distance, point, pos-
    j, plane_sweep_neighbors, i);
101.                        max_distance = plane_sweep_neighbors[i].neighbor[0].distance;
102.                    }
103.                }
104.                break;
105.            }
106.            dy = dataset_point[i].y - point[pos+j].y;
107.            new_distance = dx*dx + dy*dy;
108.            if ((max_distance > new_distance)) {
109.                heap_add_and_replace_root(new_distance, point, pos+j, plane_sweep_neighbor
    s, i);
110.                max_distance = plane_sweep_neighbors[i].neighbor[0].distance;
111.            }
112.        }
113.    }
114.}
```

## 3.3 ΥΛΟΠΟΙΗΣΗ SIMPLE PLANE SWEEP ΜΕ ΛΩΡΙΔΕΣ ΙΣΟΥ ΥΨΟΥΣ (JOIN)

Παρακάτω έχουμε τον κώδικα για την υλοποίηση του Simple Plane-Sweep με λωρίδες ίσου ύψους. Αρχικά καλούμε την create_strips() για να ορίσουμε τα όρια από τις λωρίδες διαιρώντας τον αριθμό λωριδών με τον χώρο μας (Γραμμή 12). Έπειτα διατρέχουμε κάθε σημείο και το εισάγουμε στη λωρίδα που του αντιστοιχεί, σύμφωνα με την y συντενταγμένη του (Γραμμή 16). Κάθε φορά που εισάγεται ένα νέο σημείο σε μια λωρίδα, αυξάνουμε τη μεταβλητή points_at_strip για εκείνη τη λωρίδα. Οπότε έχουμε όλα τα σημεία στην αντίστοιχη λωρίδα είτε είναι στο training set είτε στο input set. Τέλος για κάθε dataset και για κάθε λωρίδα ταξινομούμε τα σημεία ως προς το x.

```
1.  void create_strips() {
2.      double threshold;
3.      int i, j;
4.
5.      for (i=0; i<strips_num; i++) {
6.          points_at_strip[i] = 0;
7.          datapoints_at_strip[i]=0;
8.      }
9.
10.     /*Saving points for strips*/
11.     //threshold is the height of each strip
12.     threshold = 1/(double)strips_num;
13.     //categorize each point to the suitable strip
14.     for (i=0; i<points_num; i++) {
15.         for (j=0; j<strips_num; j++) {
16.             if (point[i].y <= (j+1)*threshold) {
17.                 strip_point[j][points_at_strip[j]] = point[i];
18.                 strip_point[j][points_at_strip[j]].id = points_at_strip[j];
19.                 points_at_strip[j]++;
20.                 break;
21.     }    }    }
22.     /*Saving datapoints for strips*/
23.     //threshold is the height of each strip
24.     threshold = 1/(double)strips_num;
25.     //categorize each point to the suitable strip
26.     for (i=0; i<dataset_points_num; i++) {
27.         for (j=0; j<strips_num; j++) {
28.             if (dataset_point[i].y <= (j+1)*threshold) {
29.                 datastrip_point[j][datapoints_at_strip[j]] = dataset_point[i];
30.                 datastrip_point[j][datapoints_at_strip[j]].id = datapoints_at_strip[j];
31.                 datapoints_at_strip[j]++;
32.                 break;
33.             }
34.         }
35.     }
36.     /* quicksort each strip */
37.     for (i=0; i<strips_num; i++) {
38.         quicksort_points(0, points_at_strip[i]-1, strip_point[i]);
39.         quicksort_points(0, datapoints_at_strip[i]-1, datastrip_point[i]);
40.     }
41. }
```

Έπειτα για κάθε λωρίδα ξεκινάμε την αναζήτηση για τους γείτονες. Αρχικά βρίσκουμε το σημείο του training set που βρίσκεται πιο κοντά – ως προς το x – στο σημείο του input set (Γραμμή 15). Εδώ θα πρέπει να σημειώσουμε ότι μπορεί η λωρίδα που ανήκει το σημείο του Input set να μην έχει καθόλου σημεία από το training set. Σε αυτή την περίπτωση ψάχνουμε και σε άλλες λωρίδες (Γραμμή 20). Έπειτα βρίσκουμε τα όρια στα οποία θα οποία θα κάνουμε την αναζήτηση (Γραμμή 37).

```
1.  void find_fixed_strip_neighbors(Point *point, int strip) {
2.      int i,j, max_j, strip_points_num, array_length, pos, inputpoint_strip;
3.      double dx,dy, max_distance, new_distance;
4.
5.      strip_points_num = datapoints_at_strip[strip];
6.
7.      for (i=0; i<datapoints_at_strip[strip]; i++) {
8.          // initialization
9.          array_length = 0;
10.         strip_neighbors[strip][i].p = &point[i];
11.         strip_neighbors[strip][i].neighbor[0].distance = 0;
12.
13.         // if the strip has some elements, find the closest to x point
14.         if (points_at_strip[strip] != 0) {
15.             pos = binary_search(strip_point[strip], 0, points_at_strip[strip]-
    1, strip_neighbors[strip][i].p->x);
16.             // inputpoint_strip is the strip that the closest to x point of the training s
    et belongs
17.             inputpoint_strip = strip;
18.         }
19.         // if the strip is empty, search in other strips
20.         else {
21.             for (j=1; j<strips_num; j++){
22.                 if (strip-j>=0 && points_at_strip[strip-j]!=0) {
23.                     pos = binary_search(strip_point[strip-j], 0, points_at_strip[strip-j]-
    1, strip_neighbors[strip][i].p->x);
24.                     // inputpoint_strip is the strip that the closest to x point of the tr
    aining set belongs
25.                     inputpoint_strip = strip-j;
26.                     break;
27.                 }
28.                 if (strip+j<strips_num && points_at_strip[strip-j]!=0) {
29.                     pos = binary_search(strip_point[strip+j], 0, points_at_strip[strip+j]-
    1, strip_neighbors[strip][i].p->x);
30.                     // inputpoint_strip is the strip that the closest to x point of the tr
    aining set belongs
31.                     inputpoint_strip = strip-j;
32.                     break;
33.         }    }    }
34.         strip_points_num = points_at_strip[inputpoint_strip];
35.          /* max_j to calculate the max steps while searching alternately depends
    weather the
36.         max steps have to be done on the left or on the right before reaching the boundari
    es */
37.         if (pos <= (strip_points_num/2))
38.             max_j = strip_points_num-pos;
39.         else
40.             max_j = pos;
```

Στη συνέχεια, μέχρι να γεμίσει η λίστα των γειτόνων ελέγχουμε τα σημεία με τη λογική Plane Sweep Alternately και αποθηκεύουμε τα σημεία που βρίσκουμε. Εάν ελέγξουμε όλα τα σημεία στη λωρίδα, αλλά δεν έχει γεμίσει ακόμα η λίστα γειτόνων ψάχνουμε σε νέα λωρίδα (Γραμμή 50).

```
1.    dx = point[i].x - strip_point[inputpoint_strip][pos].x;
2.          dy = point[i].y - strip_point[inputpoint_strip][pos].y;
3.          new_distance = dx*dx + dy*dy;
4.          if (new_distance != 0) {
5.              array_length = heap_add(new_distance, strip_point[inputpoint_strip], pos, arra
    y_length, strip_neighbors[strip], i);
6.          }
7.
8.      /* Code for filling the empty array of neighbors. Working Alternately */
9.      // j is the step counter which indicates how many steps we are making (either left of
    right)
10.         for (j=1; j<=max_j; j++) {
11.             // check if we are inside the left boundaries
12.             if (pos-j>=0) {
13.                 dx = point[i].x - strip_point[inputpoint_strip][pos-j].x;
14.                 dy = point[i].y - strip_point[inputpoint_strip][pos-j].y;
15.                 new_distance = dx*dx + dy*dy;
16.                 if (new_distance != 0) {
17.                     array_length = heap_add(new_distance, strip_point[inputpoint_strip], p
    os-j, array_length, strip_neighbors[strip], i);
18.                     // if after the j on the left all slots are full, check the j on the r
    ight before break
19.                     if (array_length == neighbors_num) {
20.                         // check if we are inside the right boundaries
21.                         if (pos+j<strip_points_num) {
22.                             dx = point[i].x - strip_point[inputpoint_strip][pos+j].x;
23.                             dy = point[i].y - strip_point[inputpoint_strip][pos+j].y;
24.                             new_distance = dx*dx + dy*dy;
25.                             max_distance = strip_neighbors[strip][i].neighbor[0].distance;

26.                             if ((max_distance > new_distance)) {
27.                                 heap_add_and_replace_root(new_distance, strip_point[inputp
    oint_strip], pos+j, strip_neighbors[strip], i);
28.                             }
29.                         }
30.                         break;
31.                     }
32.                 }
33.             }
34.             // check if we are inside the right boundaries
35.             if (pos+j<strip_points_num) {
36.                 dx = point[i].x - strip_point[inputpoint_strip][pos+j].x;
37.                 dy = point[i].y - strip_point[inputpoint_strip][pos+j].y;
38.                 new_distance = dx*dx + dy*dy;
39.                 if (new_distance != 0) {
40.                     array_length = heap_add(new_distance, strip_point[inputpoint_strip], p
    os+j, array_length, strip_neighbors[strip], i);
41.                     if (array_length == neighbors_num)
42.                         break;
43.                 }
44.             }
45.         }
46.
47.         // if not sufficient neighbors found
48.         if (j == max_j+1) {
49.             // check in other strip
50.             find_new_fixed_strips(strip, i, array_length, inputpoint_strip);
51.         }
```

Κατόπιν, βρισκόμαστε στο σημείο όπου η λίστα των γειτόνων έχει γεμίσει και τώρα για κάθε νέο σημείο πρέπει να ελέγξουμε την απόστασή του. Εάν η απόστασή του είναι μικρότερη από αυτή του χειρότερου αποθηκευμένου γείτονα, τότε θα γίνει αντικατάσταση του σημείου αυτού. Συνεχίζουμε να εργαζόμαστε εναλλάξ αριστερά και δεξιά αντικαθιστώντας γείτονες. Στο τέλος, ελέγχουμε την απόσταση του σημείου του οποίου ψάχνουμε τους κοντινότερους γείτονές του και εάν είναι μεγαλύτερη από την απόστασή του από κάποια άλλη λωρίδα, ελέγχουμε και τις επικαλυπτόμενες λωρίδες, αλλιώς τερματίζουμε (Γραμμή 61).

```
1.  /* code for replacing the full array of neighbors with closer ones */
2.          else {
3.              max_distance = strip_neighbors[strip][i].neighbor[0].distance;
4.
5.              // j is the step counter which indicates how many steps we are making (either
    left of right)
6.              for (j=j+1; j<=max_j; j++) {
7.                  // check if we are inside the left boundaries
8.                  if (pos-j>=0) {
9.                      dx = point[i].x - strip_point[inputpoint_strip][pos-j].x;
10.                     // if the rest of the points on the right are far away, check only on
    the left and then break
11.                     if ((max_distance < dx*dx)) {
12.                         for ( ;pos+j<strip_points_num; j++) {
13.                             dx = point[i].x - strip_point[inputpoint_strip][pos+j].x;
14.                             if ((max_distance < dx*dx))
15.                                 break;
16.                             dy = point[i].y - strip_point[inputpoint_strip][pos+j].y;
17.                             new_distance = dx*dx + dy*dy;
18.                             // if the new point is closer than the worst neighbor replace
    the second with that point
19.                             if ((max_distance > new_distance)) {
20.                                 heap_add_and_replace_root(new_distance, strip_point[inputp
    oint_strip], pos+j, strip_neighbors[strip], i);
21.                                 max_distance = strip_neighbors[strip][i].neighbor[0].dista
    nce;
22.                             }
23.                         }
24.                         break;
25.                     }
26.                     dy = point[i].y - strip_point[inputpoint_strip][pos-j].y;
27.                     new_distance = dx*dx + dy*dy;
28.                     // if the new point is closer than the worst neighbor replace the seco
    nd with that point
29.                     if ((max_distance > new_distance)) {
30.                         heap_add_and_replace_root(new_distance, strip_point[inputpoint_str
    ip], pos-j, strip_neighbors[strip], i);
31.                         max_distance = strip_neighbors[strip][i].neighbor[0].distance;
32.                     }
33.                 }
34.                 // check if we are inside the right boundaries
35.                 if (pos+j<strip_points_num) {
36.                     dx = point[i].x - strip_point[inputpoint_strip][pos+j].x;
37.                     // if the rest of the points on the left are far away, check only on t
    he right and then break
38.                     if ((max_distance < dx*dx)) {
39.                         for (j=j+1; pos-j>=0; j++) {
```

```
                            dx = point[i].x - strip_point[inputpoint_strip][pos-j].x;
                            if ((max_distance < dx*dx))
                                break;
                            dy = point[i].y - strip_point[inputpoint_strip][pos-j].y;
                            new_distance = dx*dx + dy*dy;
                            if ((max_distance > new_distance)) {
                                heap_add_and_replace_root(new_distance, strip_point[inputpoint_strip], pos-j, strip_neighbors[strip], i);
                                max_distance = strip_neighbors[strip][i].neighbor[0].distance;
                            }
                        }
                        break;
                    }
                    dy = point[i].y - strip_point[inputpoint_strip][pos+j].y;
                    new_distance = dx*dx + dy*dy;
                    if ((max_distance > new_distance)) {
                        heap_add_and_replace_root(new_distance, strip_point[inputpoint_strip], pos+j, strip_neighbors[strip], i);
                        max_distance = strip_neighbors[strip][i].neighbor[0].distance;
                    }
                }
            }
            // check if we need to search in a new strip
            if ((inputpoint_strip1>=0 && (strip_neighbors[strip][i].neighbor[0].distance > distance_point_from_strip(strip_neighbors[strip][i].p, inputpoint_strip-1)))
|| (inputpoint_strip+1<strips_num && (strip_neighbors[strip][i].neighbor[0].distance > distance_point_from_strip(strip_neighbors[strip][i].p, inputpoint_strip+1)))) {
                find_new_fixed_strips(strip, i, array_length, inputpoint_strip);
            }
        }
    }
}
```

Τώρα βρισκόμαστε στην επιλογή της νέας λωρίδας που θέλουμε να ψάξουμε. Αρχικά, δεδομένης της λωρίδας που βρισκόμαστε, βρίσκουμε το μέγιστο βήμα που μπορούμε να κάνουμε αριστερά ή δεξιά (Γραμμή 8). Έπειτα, ελέγχουμε ανάλογα με το βήμα i αριστερά και δεξιά. Εάν δεν έχουμε ελέγξει τη λωρίδα και υπάρχουν σημεία του training set μέσα, υπολογίζουμε τη νέα απόσταση του σημείου μας με τα όρια της λωρίδας και αν δεν υπερβαίνει αυτή του χειρότερου αποθηκευμένου γείτονα, ξεκινούμε αναζήτηση σε αυτή (από αριστερά Γραμμή 17, από δεξιά Γραμμή 21). Τέλος, αν χρειαστεί να αναζητήσουμε σε άλλη λωρίδα καλούμε την αντίστοιχη συνάρτηση, όπου με τη λογική που είδαμε στο προηγούμενο βήμα (αναζήτηση σε μία λωρίδα) βρίσκουμε τους γείτονες και αντικαθιστούμε τους παλιούς. Εάν το βήμα μεγαλώσει τόσο που η απόσταση του σημείου από τις λωρίδες είναι μεγαλύτερη από την απόσταση του χειρότερου γείτονα, τότε διακόπτεται η αναζήτηση (Γραμμή 46). Συνεπώς, έχουμε βρει όλους τους k κοντινότερους γείτονες για κάθε σημείο.

```
1.  void find_new_fixed_strips (int strip, int point_pos, int array_length, int input_strip) {
2.
3.      int i, max_i, closest_pos;
4.      // initializations set to a value far greater than the real distances which vary betwe
    en [0.0 - 1.0]
5.      double distance_down = 10, distance_up = 10;
6.
7.      // max_i to calculate the max steps towards the strips
8.      if (strip <= (strips_num/2))
9.          // depends weather the max steps have to be done on the left or on the right
10.         max_i = strips_num-strip;
11.     else
12.         max_i = strip;
13.
14.     // i is the step counter which indicates how many steps we are making (either left of
    right)
15.     for (i=1; i<=max_i; i++) {
16.         // if we are inside the left boundaries and the strip has some elements
17.         if (strip-i >=0 && points_at_strip[strip-i] >0) {
18.             // find the distance between the point we are searching its neighbors and the
    lower strip we want to search for
19.             distance_down = distance_point_from_strip(strip_neighbors[strip][point_pos].p,
     strip-i);
20.             // if we haven't searched in that strip
21.             if (((strip-input_strip>=0 && strip-input_strip-i<0) || (abs(strip-
    input_strip)-i<0))
22.                 // if distance from strip < max neighbor distance or not sufficient neighb
    ors found
23.                 && (distance_down < strip_neighbors[strip][point_pos].neighbor[0].distance
     || array_length < neighbors_num)) {
24.                 // find the point-position that has the closest x-
    coordinate to the point we are searching its neighbors
25.                 closest_pos = binary_search(strip_point[strip-
    i], 0, points_at_strip[strip-i]-1, strip_neighbors[strip][point_pos].p->x);
26.                 // find the neighbors
```

```
27.                 array_length = find_other_fixed_strip_neighbors(strip_point[strip-
    i], strip-i, point_pos, closest_pos, strip, array_length);
28.             }
29.         }
30.         // if we are inside the right boundaries and the strip has some elements
31.         if (strip+i < strips_num && points_at_strip[strip+i] > 0) {
32.             // find the distance between the point we are searching its neighbors and the
    upper strip we want to search for
33.             distance_up = distance_point_from_strip(strip_neighbors[strip][point_pos].p, s
    trip+i);
34.             // if we haven't searched in that strip
35.             if (((strip-input_strip<=0 && input_strip-strip-i<0) || (abs(strip-
    input_strip)-i<=0 && input_strip-strip-i<0))
36.                 // if distance from strip < max neighbor distance or not sufficient neighb
    ors found
37.                 && (distance_up < strip_neighbors[strip][point_pos].neighbor[0].distance |
    | array_length < neighbors_num)) {
38.                 // find the point-position that has the closest x-
    coordinate to the point we are searching its neighbors
39.                 closest_pos = binary_search(strip_point[strip+i], 0, points_at_strip[strip
    +i]-1, strip_neighbors[strip][point_pos].p->x);
40.                 // find the neighbors
41.                 array_length = find_other_fixed_strip_neighbors(strip_point[strip+i], stri
    p+i, point_pos, closest_pos, strip, array_length);
42.             }
43.
44.         }
45.         // if distance from lower strip AND distance from upper strip are greater than max
    -neighbor-distance and there are no empty neighbors-> end
46.         if (strip_neighbors[strip][point_pos].neighbor[0].distance < distance_up && strip_
    neighbors[strip][point_pos].neighbor[0].distance < distance_down
47.             && array_length == neighbors_num)
48.             break;
49.     }
50. }
```

## 3.4 ΥΛΟΠΟΙΗΣΗ SIMPLE PLANE SWEEP ΜΕ ΛΩΡΙΔΕΣ ΙΣΟΥ ΑΡΙΘΜΟΥ ΣΗΜΕΙΩΝ (JOIN)

Παρακάτω έχουμε τον κώδικα για την υλοποίηση του Simple Plane-Sweep με λωρίδες ίσου αριθμού σημείων. Αρχικά ταξινομούμε τα σημεία του training set ως προς y και για κάθε points_per_strip σημεία ορίζουμε μια νέα λωρίδα (Γραμμή 10). Έπειτα, βρίσκουμε τα όρια κάθε λωρίδας, όπου το πάνω όριο είναι το y του 1$^{ου}$ σημείου και το κάτω το y του τελευταίου σημείου (Γραμμή 25). Αφού έχουμε οριοθετήσει τις λωρίδες, κατατάσσουμε τα σημεία του input set ανάλογα με το y τους στη λωρίδα που ανήκουν και ξεκινούμε την αναζήτηση γειτόνων (Γραμμή 38).

```
1.  void create_free_strips() {
2.      int i, j;
3.
4.      strips_num = points_num/points_per_strip +1;
5.      points_of_last_strip = points_num - (strips_num-1)*points_per_strip;
6.
7.      quicksort_points_y(0, points_num-1, point);
8.
9.      /*Saving points for strips*/
10.     for (i=0; i<strips_num-1; i++) {
11.         //categorize each point to the suitable strip
12.         for (j=0; j<points_per_strip; j++) {
13.             strip_point[i][j] = point[i*(points_per_strip) + j];
14.             strip_point[i][j].id = j;
15.             datapoints_at_strip[i]=0;
16.         }
17.     }
18.     //categorize each point to the suitable strip
19.     for (j=0; j<points_of_last_strip; j++) {
20.         strip_point[i][j] = point[i*(points_per_strip) + j];
21.         strip_point[i][j].id = j;
22.         datapoints_at_strip[i]=0;
23.     }
24.     /* quicksort each strip */
25.     for (i=0; i<strips_num-1; i++) {
26.         // setting the strip boundaries
27.         //(upper boundary: strip_point[i][0], lower boundary: strip_point[i][1]
28.         strip_boundaries[i][0] = strip_point[i][0].y;
29.         strip_boundaries[i][1] = strip_point[i][points_per_strip-1].y;
30.         quicksort_points(0, points_per_strip-1, strip_point[i]);
31.     }
32.     strip_boundaries[i][0] = strip_point[i][0].y;
33.     strip_boundaries[i][1] = 1;
34.     strip_boundaries[0][0] = 0;
35.     quicksort_points(0, points_of_last_strip-1, strip_point[i]);
36.
37.     //categorize each point of input set to the suitable strip
38.     for (i=0; i<dataset_points_num; i++) {
39.         for (j=0; j<strips_num; j++) {
40.             if (dataset_point[i].y <= strip_boundaries[j][1]) {
41.                 datastrip_point[j][datapoints_at_strip[j]] = dataset_point[i];
42.                 datastrip_point[j][datapoints_at_strip[j]].id = datapoints_at_strip[j];
43.                 datapoints_at_strip[j]++;
44.                 break;
45.             }
46. }   }   }
```

Αφού χωρίσαμε το χώρο σε λωρίδες ίσου πλήθους σημείων προχωράμε στην συνάρτηση που αναζητά γείτονες σε μία λωρίδα. Αρχικά βρίσκουμε το σημείο του training set το οποίο είναι πιο κοντά – ως προς x – στο σημείο του input set, του οποίου ψάχνουμε τους k κοντινότερους γείτονες (Γραμμή 19). Είμαστε σίγουροι ότι δεν υπάρχουν κενές λωρίδες, αφού εμείς τις κατασκευάσαμε, οπότε προσθέτουμε αυτό το πρώτο σημείο, γνωρίζοντας ότι η λίστα γειτόνων είναι ακόμα άδεια. Έπειτα, ξεκινούμε τη διαδικασία εύρεσης των υπόλοιπων γειτόνων στη λωρίδα.

```
1.  void find_free_strip_neighbors(Point *point, int strip) {
2.
3.      int i,j, max_j, array_length, strip_points_num, inputpoint_strip, pos;
4.      double dx,dy, max_distance, new_distance;
5.
6.      for (i=0; i<datapoints_at_strip[strip]; i++) {
7.          // initialization
8.          array_length = 0;
9.          strip_neighbors[strip][i].p = &point[i];
10.         strip_neighbors[strip][i].neighbor[0].distance = 0;
11.
12.         // check if we are in the last strip with less points
13.         if (strip!=strips_num-1)
14.             strip_points_num=points_per_strip;
15.         else
16.             strip_points_num=points_of_last_strip;
17.
18.         // find the closest-to-
    x point with the one that we are searching for its neighbors
19.         pos = binary_search(strip_point[strip], 0, strip_points_num-
    1, strip_neighbors[strip][i].p->x);
20.         inputpoint_strip = strip;
21.
22.         // max_j to calculate the max steps while searching alternately
23.         if (pos <= (strip_points_num/2))
24.             max_j = strip_points_num-pos;
25.         else
26.             max_j = pos;
27.
28.         dx = point[i].x - strip_point[inputpoint_strip][pos].x;
29.         dy = point[i].y - strip_point[inputpoint_strip][pos].y;
30.         // new_distance is the distance between the 2 points (i and j) we are checking

31.         new_distance = dx*dx + dy*dy;
32.         if (new_distance != 0) {
33.             array_length = heap_add(new_distance, strip_point[inputpoint_strip], pos,
    array_length, strip_neighbors[strip], i);
34.         }
```

Συνεχίζουμε με το γέμισμα της λίστας των γειτόνων με κάθε σημείο που βρίσκουμε. Ελέγχουμε αριστερά και δεξιά εναλλάξ με βήμα j. Και πάλι, εάν βρούμε σημείο με ακριβώς ίδιες συντεταγμένες, δεν το προσθέτουμε ως γείτονα (Γραμμή 43). Με αυτό τον τρόπο μπορούμε να έχουμε training και input set πανομοιότυπο, οπότε να εκτελέσουμε self-join k-κοντινότερων γειτόνων. Τέλος, εάν τελείωσαν τα σημεία στη λωρίδα και δεν γέμισε η λίστα γειτόνων, η αναζήτηση συνεχίζεται σε νέα λωρίδα (Γραμμή 75).

```
35. /* Code for filling the empty array of neighbors. Working Alternately */
36.         // j is the step counter which indicates how many steps we are making (either
    left of right)
37.         for (j=1; j<=max_j; j++) {
38.             // check if we are inside the left boundaries
39.             if (pos-j>=0) {
40.                 dx = point[i].x - strip_point[inputpoint_strip][pos-j].x;
41.                 dy = point[i].y - strip_point[inputpoint_strip][pos-j].y;
42.                 new_distance = dx*dx + dy*dy;
43.                 if (new_distance != 0) {
44.                     array_length = heap_add(new_distance, strip_point[inputpoint_strip
    ], pos-j, array_length, strip_neighbors[strip], i);
45.                     // if after the j on the left all slots are full, check the j on t
    he right before break
46.                     if (array_length == neighbors_num) {
47.                         // check if we are inside the right boundaries
48.                         if (pos+j<strip_points_num) {
49.                             dx = point[i].x -
     strip_point[inputpoint_strip][pos+j].x;
50.                             dy = point[i].y -
     strip_point[inputpoint_strip][pos+j].y;
51.                             new_distance = dx*dx + dy*dy;
52.                             max_distance = strip_neighbors[strip][i].neighbor[0].dista
    nce;
53.                             // if the point we found has less distance than the worst
    neighbor, add it
54.                             if ((max_distance > new_distance)) {
55.                                 heap_add_and_replace_root(new_distance, strip_point[in
    putpoint_strip], pos+j, strip_neighbors[strip], i);
56.                             }
57.                         }
58.                         break;
59.                     }
60.                 }
61.             }
62.             // check if we are inside the right boundaries
63.             if (pos+j<strip_points_num) {
64.                 dx = point[i].x - strip_point[inputpoint_strip][pos+j].x;
65.                 dy = point[i].y - strip_point[inputpoint_strip][pos+j].y;
66.                 new_distance = dx*dx + dy*dy;
67.                 if (new_distance != 0) {
68.                     // fill with points until the neighbor list is full
69.                     array_length = heap_add(new_distance, strip_point[inputpoint_strip
    ], pos+j, array_length, strip_neighbors[strip], i);
70.                     if (array_length == neighbors_num)
71.                         break;
72.         }   }   }
73.         // if not sufficient neighbors found
74.         if (j == max_j+1) {
75.             // check in other strip
76.             find_new_free_strip(strip, i, array_length, inputpoint_strip);
77.         }
```

Στη συνέχεια ψάχνουμε για τους υπόλοιπους γείτονες, παραμένοντας πάντα στην ίδια λωρίδα. Αφού η λίστα γειτόνων έχει γεμίσει, πρέπει να ελέγξουμε και την απόσταση του κάθε νέου σημείου και εάν είναι μικρότερη του χειρότερου αποθηκευμένου γείτονα, να γίνεται η αντικατάσταση. Κατά την ίδια λογική ελέγχουμε αριστερά και δεξιά εναλλάξ μέχρι το dx του νέου σημείου που θα ελέγξουμε να είναι μεγαλύτερο από τη χειρότερη – μέχρι στιγμής – απόσταση ή μέχρι να τελειώσουν τα σημεία στη λωρίδα (Γραμμές 90 και 115). Αφού γίνουν οι απαραίτητες αντικαταστάσεις, πρέπει να γίνει ο έλεγχος για πιθανή αναζήτηση σε νέες λωρίδες. Στη Γραμμή 135, ελέγχουμε εάν η απόσταση του σημείου από την επόμενη λωρίδα (πάνω και κάτω) είναι μικρότερη από την απόσταση του χειρότερου γείτονα. Εάν όντως είναι, τότε πρέπει να γίνει και εκεί αναζήτηση (Γραμμή 137).

```
78. /* code for replacing the full array of neighbors with closer ones */
79.         else {
80.             max_distance = strip_neighbors[strip][i].neighbor[0].distance;
81.             // j is the step counter which indicates how many steps we are making (either
    left of right)
82.             for (j=j+1; j<=max_j; j++) {
83.                 // check if we are inside the left boundaries
84.                 if (pos-j>=0) {
85.                     dx = point[i].x - strip_point[inputpoint_strip][pos-j].x;
86.                     // if the rest of the points on the right are far away, check only on
    the left and then break
87.                     if ((max_distance < dx*dx)) {
88.                         for ( ;pos+j<strip_points_num; j++) {
89.                             dx = point[i].x - strip_point[inputpoint_strip][pos+j].x;
90.                             if ((max_distance < dx*dx))
91.                                 break;
92.                             dy = point[i].y - strip_point[inputpoint_strip][pos+j].y;
93.                             new_distance = dx*dx + dy*dy;
94.                             if ((max_distance > new_distance) && new_distance !=0) {
95.                                 heap_add_and_replace_root(new_distance, strip_point[inputp
    oint_strip], pos+j, strip_neighbors[strip], i);
96.                                 max_distance = strip_neighbors[strip][i].neighbor[0].dista
    nce;
97.                             }
98.                         }
99.                         break;
100.                    }
101.                    dy = point[i].y - strip_point[inputpoint_strip][pos-j].y;
102.                    new_distance = dx*dx + dy*dy;
103.                    if ((max_distance > new_distance) && new_distance!=0) {
104.                        heap_add_and_replace_root(new_distance, strip_point[inputpoint_str
    ip], pos-j, strip_neighbors[strip], i);
105.                        max_distance = strip_neighbors[strip][i].neighbor[0].distance;
106.                    }
107.                }
108.                // check if we are inside the right boundaries
109.                if (pos+j<strip_points_num) {
110.                    dx = point[i].x - strip_point[inputpoint_strip][pos+j].x;
111.                    // if the rest of the points on the left are far away, check only on t
    he right and then break
112.                    if ((max_distance < dx*dx)) {
113.                        for (j=j+1; pos-j>=0; j++) {
114.                            dx = point[i].x - strip_point[inputpoint_strip][pos-j].x;
```

```
115.                             if ((max_distance < dx*dx))
116.                                 break;
117.                             dy = point[i].y - strip_point[inputpoint_strip][pos-j].y;
118.                             new_distance = dx*dx + dy*dy;
119.                             if ((max_distance > new_distance) && new_distance!=0) {
120.                                 heap_add_and_replace_root(new_distance, strip_point[inputpoint_strip], pos-j, strip_neighbors[strip], i);
121.                                 max_distance = strip_neighbors[strip][i].neighbor[0].distance;
122.                             }
123.                         }
124.                         break;
125.                     }
126.                     dy = point[i].y - strip_point[inputpoint_strip][pos+j].y;
127.                     new_distance = dx*dx + dy*dy;
128.                     if ((max_distance > new_distance) && new_distance!=0) {
129.                         heap_add_and_replace_root(new_distance, strip_point[inputpoint_strip], pos+j, strip_neighbors[strip], i);
130.                         max_distance = strip_neighbors[strip][i].neighbor[0].distance;
131.                     }
132.                 }
133.             }
134.             // if distance from point to a strip is lesser than the distance of the worst neighbor, search in that strip too.
135.             if (((inputpoint_strip-1>=0) && (strip_neighbors[strip][i].neighbor[0].distance > distance_point_from_free_strip(strip_neighbors[strip][i].p, strip_boundaries[inputpoint_strip-1][1])))
136.                 || (inputpoint_strip+1<strips_num && (strip_neighbors[strip][i].neighbor[0].distance > distance_point_from_free_strip(strip_neighbors[strip][i].p, strip_boundaries[inputpoint_strip+1][0])))) {
137.                 find_new_free_strip(strip, i, array_length, inputpoint_strip);
138.             }
139.         }
140.     }
141. }
```

Στη συνέχεια καλείται η συνάρτηση `find_new_free_strip (strip, i, array_length, inputpoint_strip)` για την αναζήτηση της επόμενης λωρίδας για ψάξιμο. Με την ίδια λογική, όπως στον αλγόριθμο με λωρίδες ίσου ύψους, ελέγχουμε τις πάνω και κάτω λωρίδες. Εάν η απόσταση του σημείου που ψάχνουμε από μία λωρίδα είναι μικρότερη από την απόσταση του χειρότερου γείτονα, τότε πρέπει να ελέγξουμε για γείτονες και σε αυτή (Γραμμές 22 και 40). Βρίσκουμε λοιπόν το σημείο στη νέα λωρίδα που είναι πιο κοντά – ως προς x – στο αρχικό μας σημείο και ξεκινούμε την αναζήτηση στη νέα λωρίδα από αυτό. Αφού βρούμε τη νέα λωρίδα για αναζήτηση χρησιμοποιούμε τον παραπάνω κώδικα για αναζήτηση γειτόνων μέσα σε μία λωρίδα. Στο τέλος, έχουμε βρει όλους τους γείτονες για κάθε σημείο του input set στην κάθε λωρίδα.

```
1.  void find_new_free_strip (int strip, int point_pos, int array_length, int input_strip) {
2.
3.      int i, max_i, closest_pos, pointsnum;
4.      //initializations set to a value far greater than the real distances which vary betwee
    n [0.0 - 1.0]
5.      double distance_down = 10, distance_up = 10;
6.
7.      // max_i to calculate the max steps towards the strips
8.      if (strip <= (strips_num/2))
9.          // depends weather the max steps have to be done on the left or on the right
10.         max_i = strips_num-strip;
11.     else
12.         max_i = strip;
13.
14.     // i is the step counter which indicates how many steps we are making (either left of
    right)
15.     for (i=1; i<=max_i; i++) {
16.         // if we are inside the left boundaries
17.         if (strip-i >=0) {
18.             pointsnum = points_per_strip;
19.             // find the distance between the point we are searching its neighbors and the
    lower strip we want to search for
20.             distance_down = distance_point_from_free_strip(strip_neighbors[strip][point_po
    s].p, strip_boundaries[strip-i][1]);
21.             // if distance from strip < max neighbor distance or not sufficient neighbors
    found
22.             if (distance_down < strip_neighbors[strip][point_pos].neighbor[0].distance ||
    array_length < neighbors_num) {
23.                 // find the point-position that has the closest x-
    coordinate to the point we are searching its neighbors
24.                 closest_pos = binary_search(strip_point[strip-i], 0, pointsnum-
    1, strip_neighbors[strip][point_pos].p->x);
25.                 // find the neighbors
26.                 array_length = find_other_free_strip_neighbors(strip_point[strip-
    i], strip-i, point_pos, closest_pos, strip, array_length);
27.             }
28.         }
29.         // if we are inside the right boundaries
30.         if (strip+i< strips_num) {
31.             // if we are in the last strip, points num is not fixed
32.             if (strip+i != strips_num-1)
33.                 pointsnum = points_per_strip;
34.             else
35.                 pointsnum = points_of_last_strip;
36.
37.             // find the distance between the point we are searching its neighbors and the
    upper strip we want to search for
38.             distance_up = distance_point_from_free_strip(strip_neighbors[strip][point_pos]
    .p, strip_boundaries[strip+i][0]);
39.             // if distance from strip < max neighbor distance or not sufficient neighbors
    found
40.             if (distance_up < strip_neighbors[strip][point_pos].neighbor[0].distance || ar
    ray_length < neighbors_num) {
41.                 // find the point-position that has the closest x-
    coordinate to the point we are searching its neighbors
42.                 closest_pos = binary_search(strip_point[strip+i], 0, pointsnum-
    1, strip_neighbors[strip][point_pos].p->x);
43.                 // find the neighbors
44.                 array_length = find_other_free_strip_neighbors(strip_point[strip+i], strip
    +i, point_pos, closest_pos, strip, array_length);
45.         }    }
46.         if (strip_neighbors[strip][point_pos].neighbor[0].distance < distance_up && strip_
    neighbors[strip][point_pos].neighbor[0].distance < distance_down
47.             && array_length == neighbors_num)
48.             break;
49. }    }
```

# 4. ΠΕΙΡΑΜΑΤΙΚΗ ΔΙΑΔΙΚΑΣΙΑ

Στο κεφάλαιο αυτό θα παρουσιάσουμε το αποτέλεσμα των πειραμάτων που διεξήχθησαν για τον έλεγχο της αποδοτικότητας των αλγορίθμων που υλοποιήθηκαν για τα ερωτήματα k-κοντινότερων γειτόνων. Αρχικά, θα εξετάσουμε τα self-join k-κοντινότερων γειτόνων και έπειτα τα join. Έγιναν εκτενή πειράματα, δοκιμάζοντας διάφορες τιμές για τις εξής παραμέτρους:

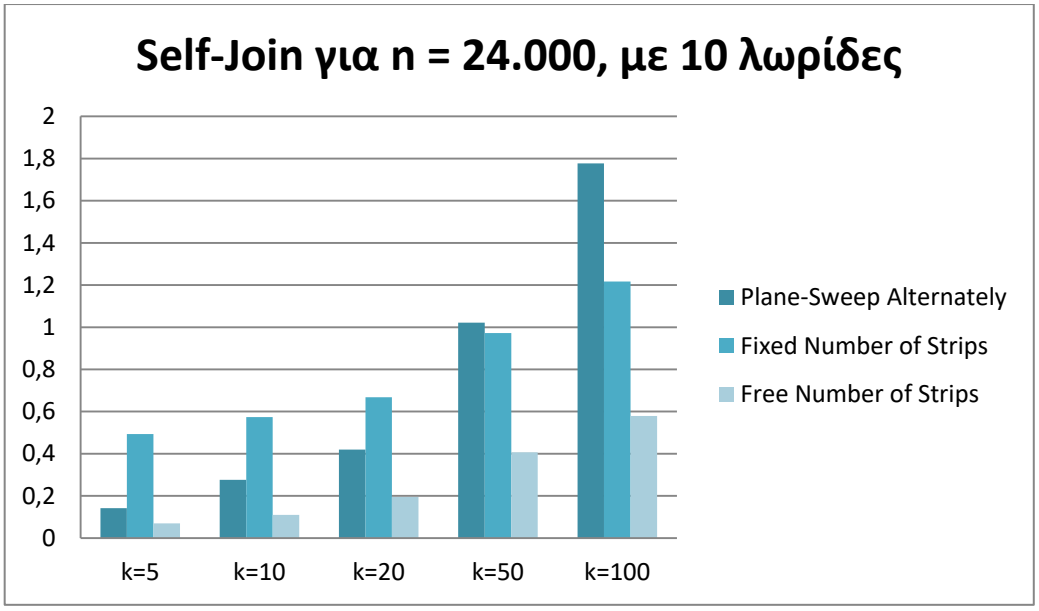
- *k*, τον αριθμό των κοντινότερων γειτόνων που ψάχνουμε,
- *n*, τον αριθμό των σημείων του training set,
- *m*, τον αριθμό των σημείων του input set,
- *l*, τον αριθμό λωρίδων, και
- *q*, τον αριθμό των σημείων που περιέχονται σε κάθε λωρίδα.

Παρακάτω θα παραθέσουμε τα αποτελέσματα για self-join αναζητησεις, που είναι εκ των πραγμάτων join, απλώς το input set και το training set είναι το ίδιο. Αρχικά πειραματιστήκαμε για n = 24.000, 19.000, 14.000, 9.000, από βάσεις δεδομένων πραγματικών σημείων. Επίσης, το k πήρε τιμές από 5 έως 100 και συγκεκριμενα 5, 10, 20, 50, 100. Όσον αφορά στις λωρίδες, κάναμε ελέγχους για 10, 20, 50, 100 και 200. Εάν ελέγχαμε τον αλγόριθμο με τις λωρίδες σταθερού ύψους ορίζαμε τον αντίστοιχο αριθμό λωριδών, ενώ στον αλγόριθμο με τις λωρίδες ίσου αριθμού σημείων, βρίσκαμε τις λωρίδες διαιρώντας το πλήθος σημείων, με τον αριθμό σημείων σε κάθε λωρίδα l=n/q. Οπότε, αλλάζοντας το q, επιλέγαμε πόσες λωρίδες θα σχηματιστούν.

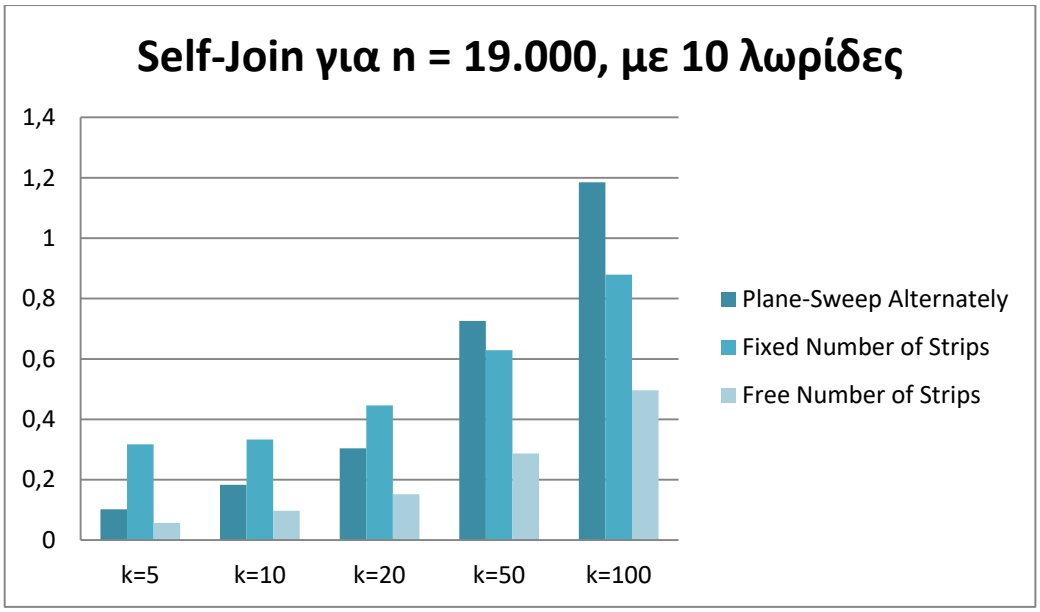
Στα παρακάτω γραφήματα βλέπουμε τον χρόνο εκτέλεσης των 3 ταχύτερων αλγορίθμων: Plane-Sweep Alternately, Plane-Sweep με σταθερό αριθμό λωρίδων (Fixed Strips) και Plane-Sweep με λωρίδες ίσου αριθμού σημείων (with Free Strips). Θα παρατηρήσουμε, λοιπόν, τη συμπεριφορά στις παραπάνω παραμέτρους για να ελέγξουμε την αποδοτικότητά τους.

## 4.1 ΠΕΙΡΑΜΑΤΑ ΜΕ ΑΡΙΘΜΟ ΛΩΡΙΔΩΝ ΙΣΟ ΜΕ 10

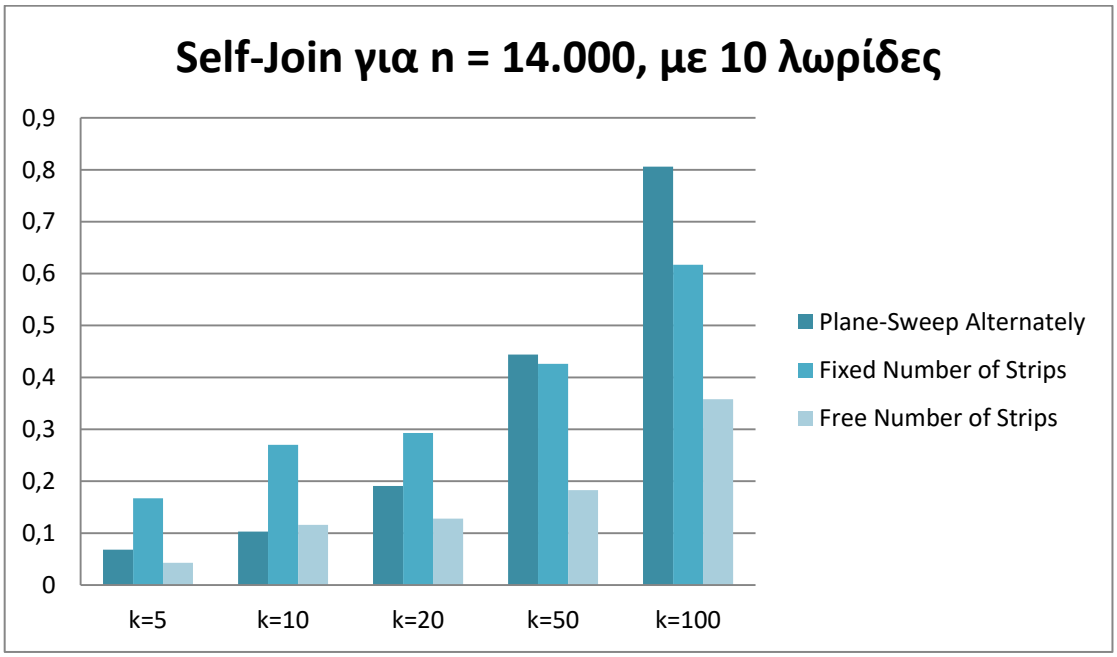
Στα 4 παρακάτω γραφήματα βλέπουμε το Self-Join για n = 24.000, 19.000, 14.000 και 9.000. Οι λωρίδες που χρησιμοποιούμε είναι 10. Το k παίρνει τιμές 5, 10, 20, 50, 100. Παρατηρούμε ότι ταχύτερος ειναι ο αλγόριθμος με ίσο αριθμό σημείων σε κάθε λωρίδα. Ο αλγόριθμος με λωρίδες ίσου ύψους είναι σχετικά αργός για μικρά k, έχει όμως καλή κλιμάκωση, σε αντίθεση με τον απλό Plane-Sweep που για μικρά k τρέχει γρήγορα, αλλά δεν έχει καθόλου καλή κλιμάκωση.



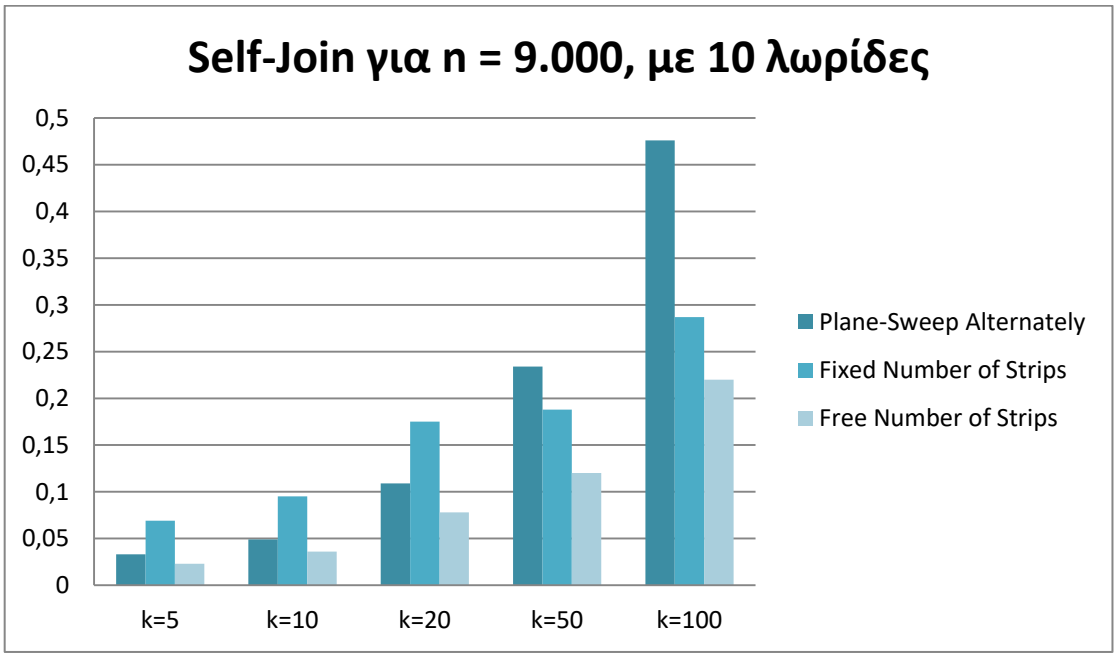

***Γράφημα 1.*** Self-Join για n = 24.000, με 10 λωρίδες.




***Γράφημα 2.*** Self-Join για n = 19.000, με 10 λωρίδες.



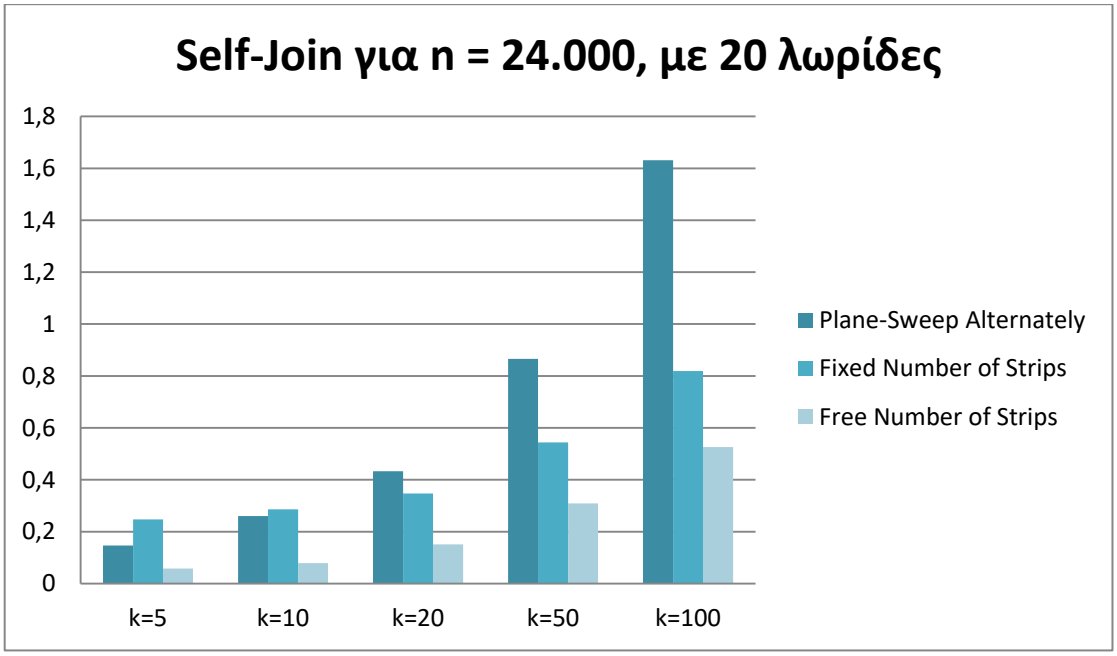

***Γράφημα 3.*** Self-Join για n = 14.000, με 10 λωρίδες.



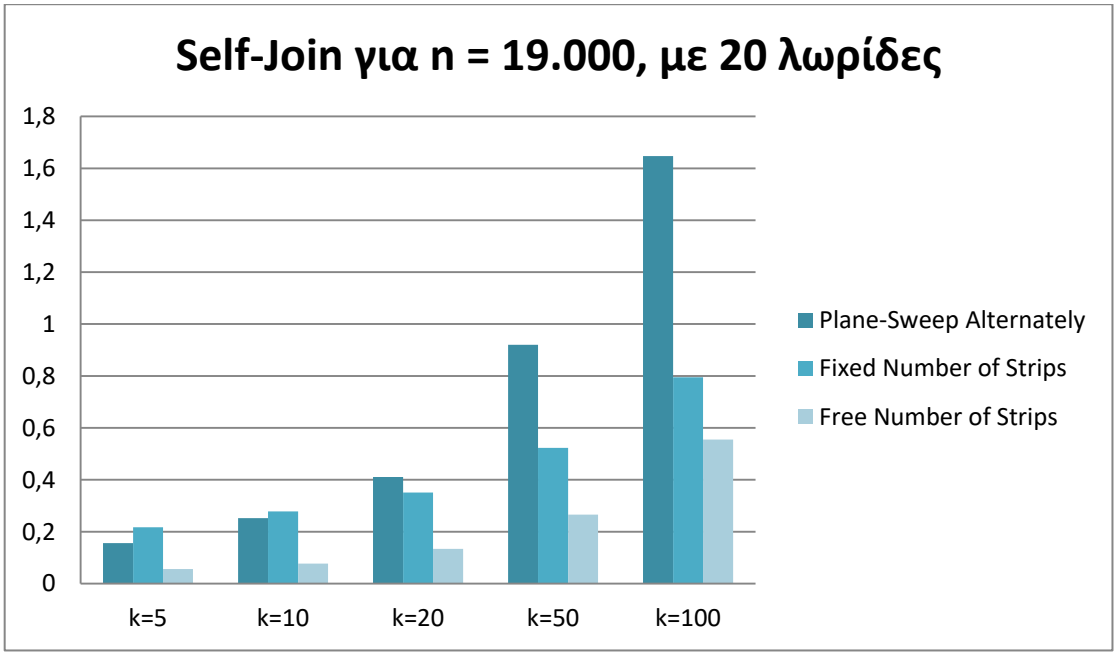

***Γράφημα 4.*** Self-Join για n = 9.000, με 10 λωρίδες.

## 4.2 ΠΕΙΡΑΜΑΤΑ ΜΕ ΑΡΙΘΜΟ ΛΩΡΙΔΩΝ ΙΣΟ ΜΕ 20

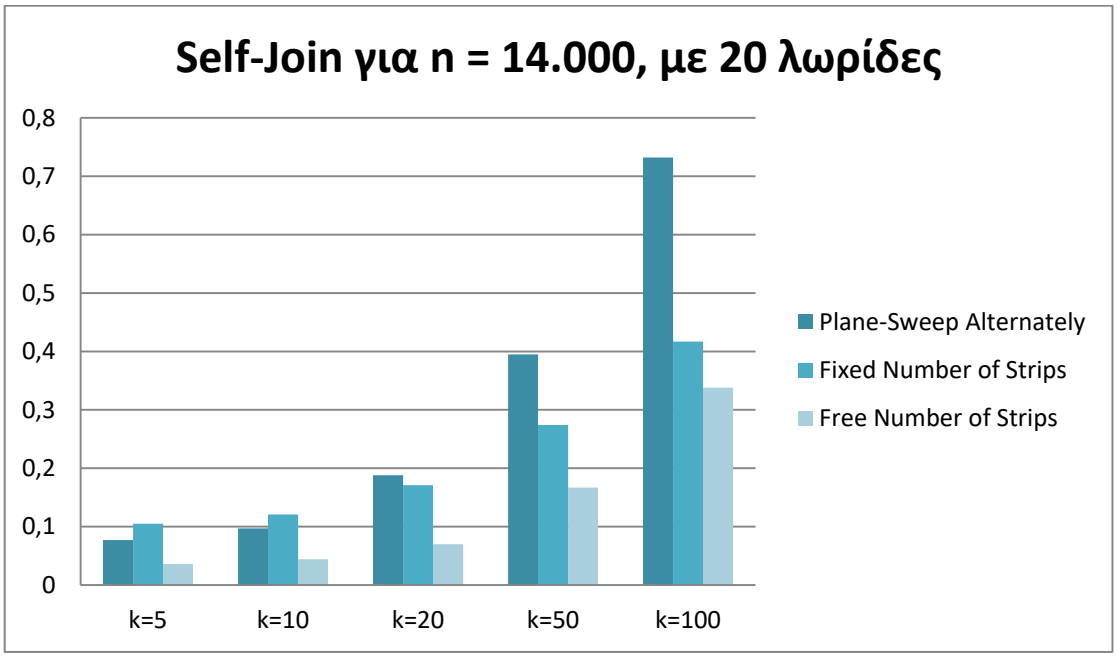
Εδώ κάνουμε τα ίδια πειράματα με αριθμό λωρίδων 20. Τα αποτελέσματα είναι παρόμοια με πριν. Η κλιμάκωση του αλγορίθμου με τις λωρίδες ίσου ύψους είναι καλή, ενώ ο αλγόριθμος με λωρίδες ίσου πλήθους σημείων είναι σταθερά ο ταχύτερος.



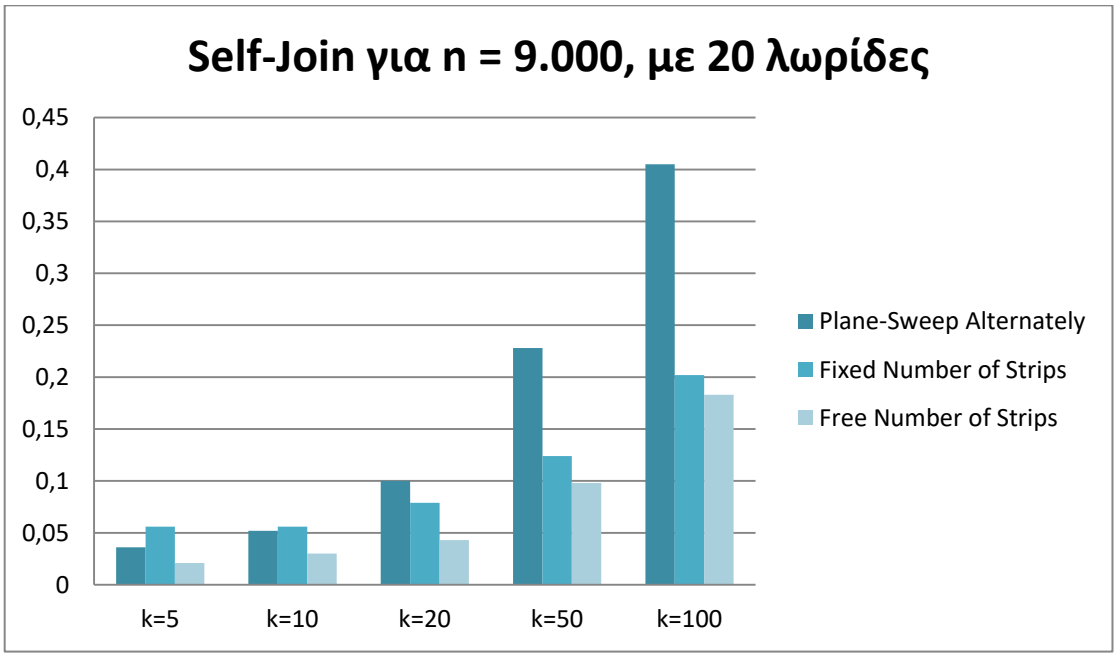

***Γράφημα 5.*** Self-Join για n = 24.000, με 20 λωρίδες.




***Γράφημα 6.*** Self-Join για n = 19.000, με 20 λωρίδες.



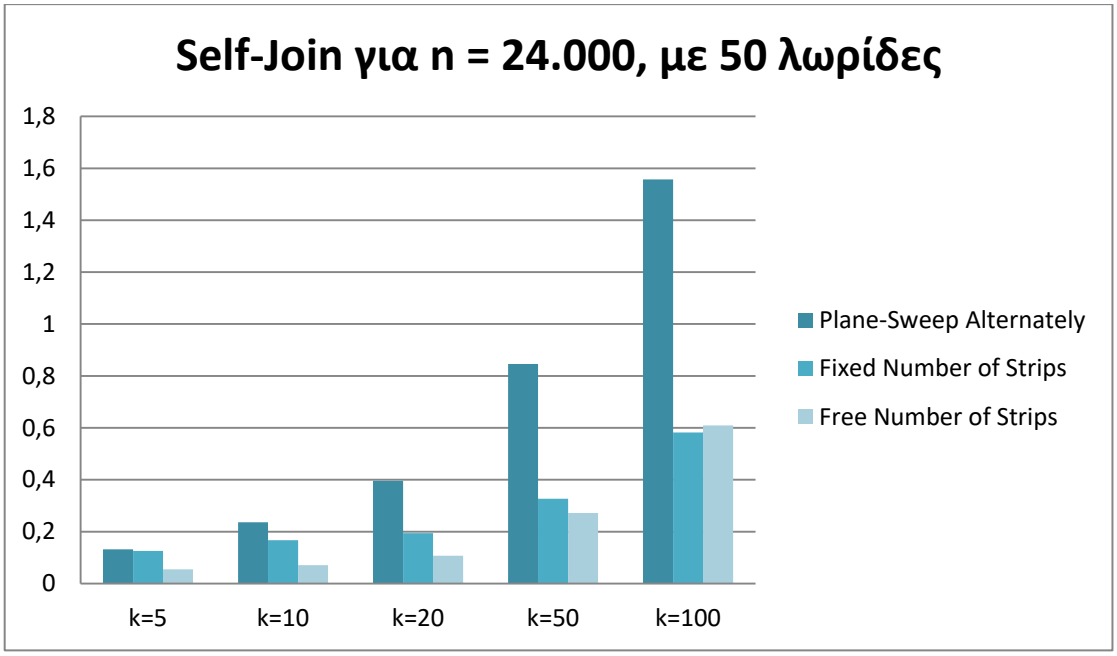

***Γράφημα 7.*** Self-Join για n = 14.000, με 20 λωρίδες.



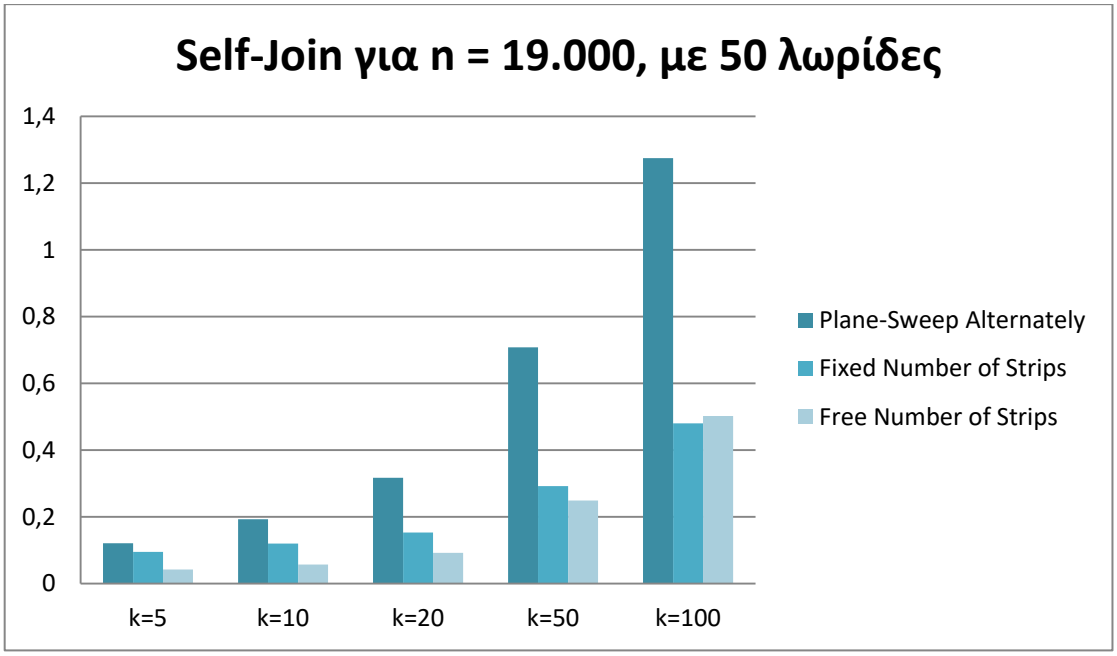

***Γράφημα 8.*** Self-Join για n = 9.000, με 20 λωρίδες.

## 4.3 ΠΕΙΡΑΜΑΤΑ ΜΕ ΑΡΙΘΜΟ ΛΩΡΙΔΩΝ ΙΣΟ ΜΕ 50

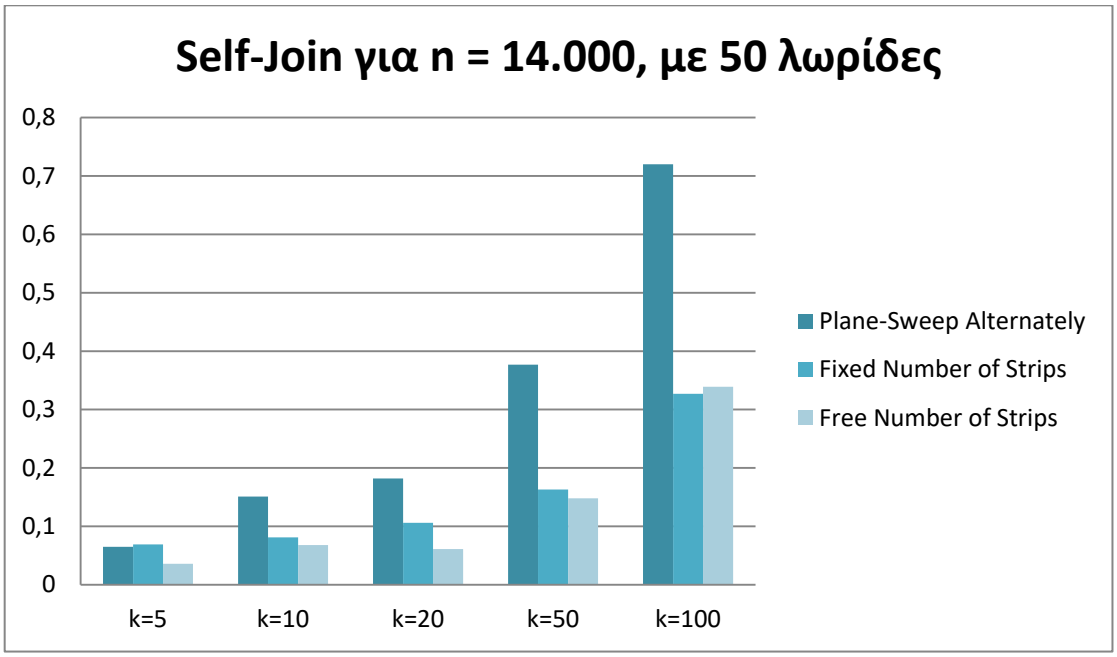
Σε αυτή τη σειρά πειραμάτων αυξάνουμε τις λωρίδες σε 50. Τα αποτελέσματα είναι παρόμοια με τα προηγούμενα. Εδώ όμως παρατηρούμε ότι η κλιμάκωση του αλγορίθμου με λωρίδες ίσου ύψους έχει καλύτερη κλιμάκωση σε σύγκριση με τον αλγόριθμο ίσου πλήθους σημείων σε κάθε λωρίδα, με αποτέλεσμα για k = 100 να είναι ταχύτερος ο πρώτος.



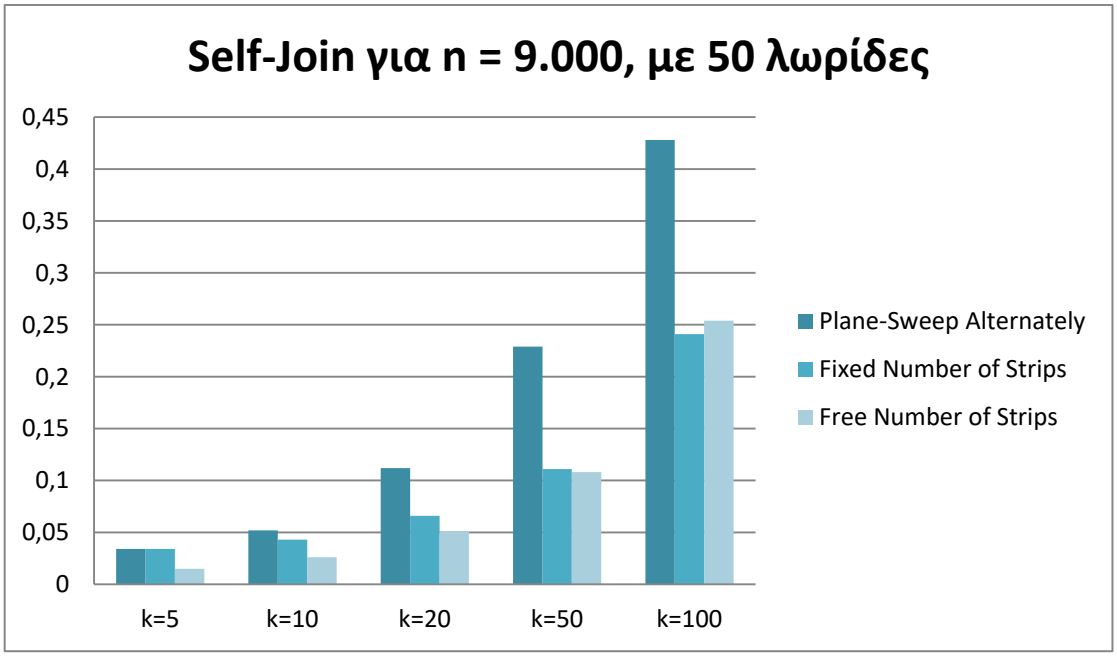

***Γράφημα 9.*** Self-Join για n = 24.000, με 50 λωρίδες.




***Γράφημα 10.*** Self-Join για n = 19.000, με 50 λωρίδες.



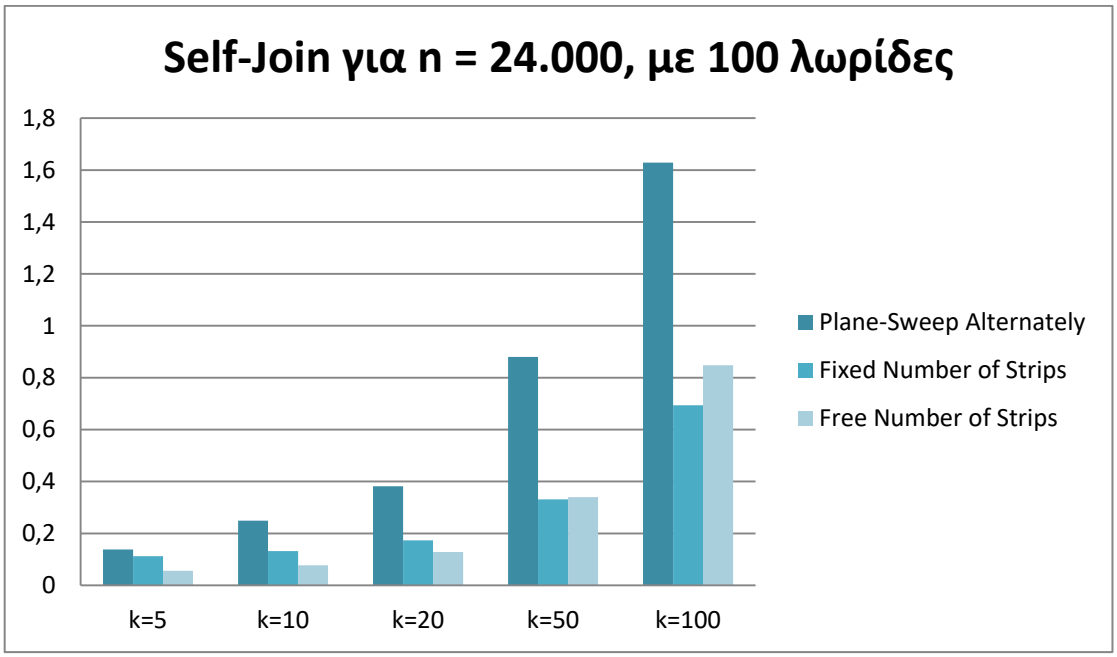

***Γράφημα 11.*** Self-Join για n = 14.000, με 50 λωρίδες.



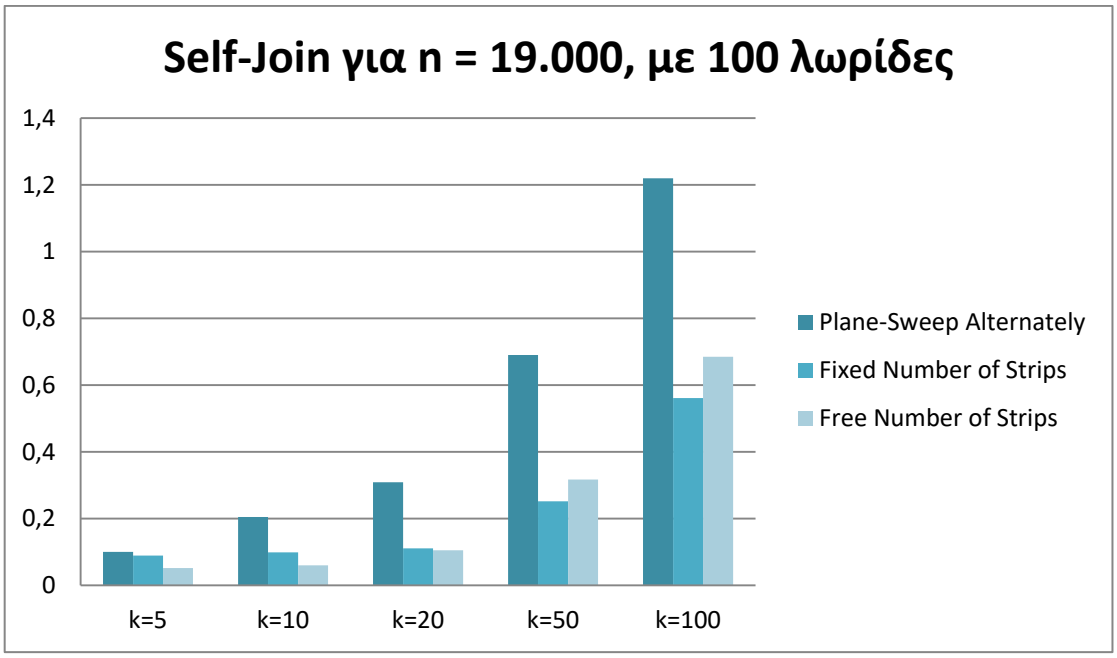

***Γράφημα 12.*** Self-Join για n = 9.000, με 50 λωρίδες.

## 4.4 ΠΕΙΡΑΜΑΤΑ ΜΕ ΑΡΙΘΜΟ ΛΩΡΙΔΩΝ ΙΣΟ ΜΕ 100

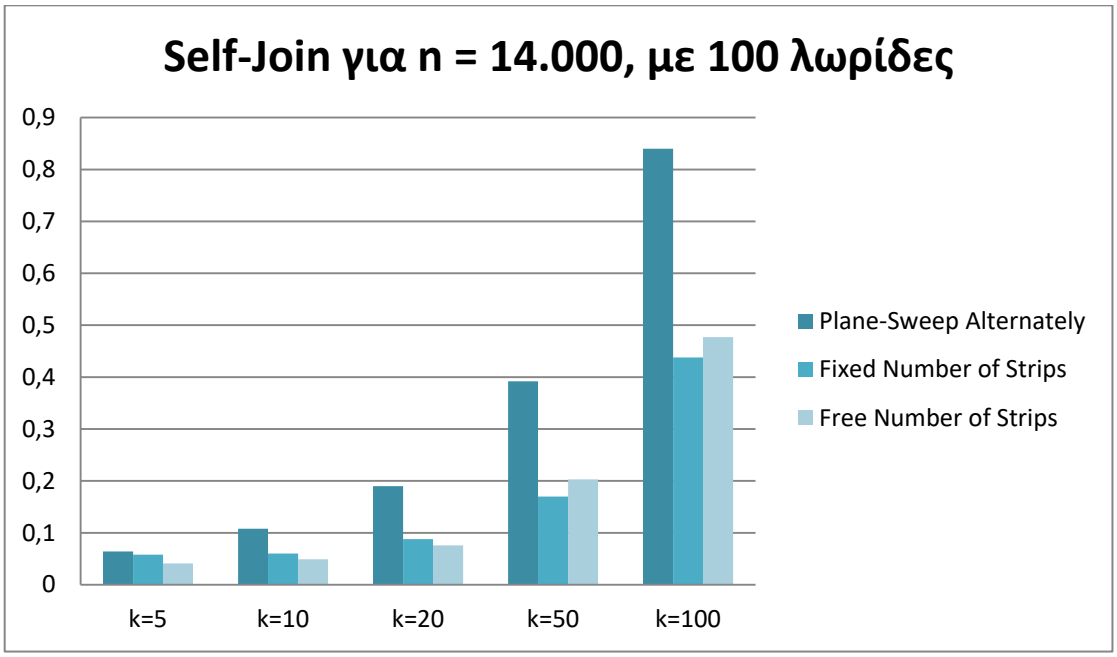
Εδώ, αυξάνουμε τις λωρίδες σε 100 και παρατηρούμε ότι τώρα πια, για k>50 ο αλγόριθμος με λωρίδες ίσου ύψους είναι ταχύτερος από τον αλγόριθμο με λωρίδες που έχουν ίσο πλήθος στοιχείων.



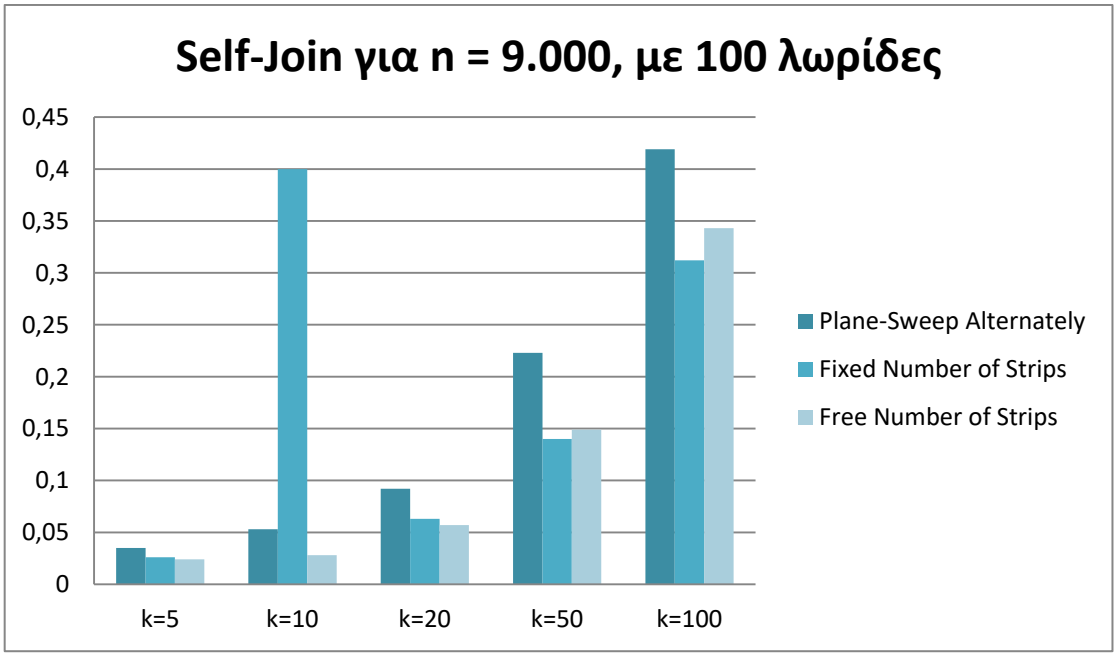

***Γράφημα 13.*** Self-Join για n = 24.000, με 100 λωρίδες.




***Γράφημα 14.*** Self-Join για n = 19.000, με 100 λωρίδες.



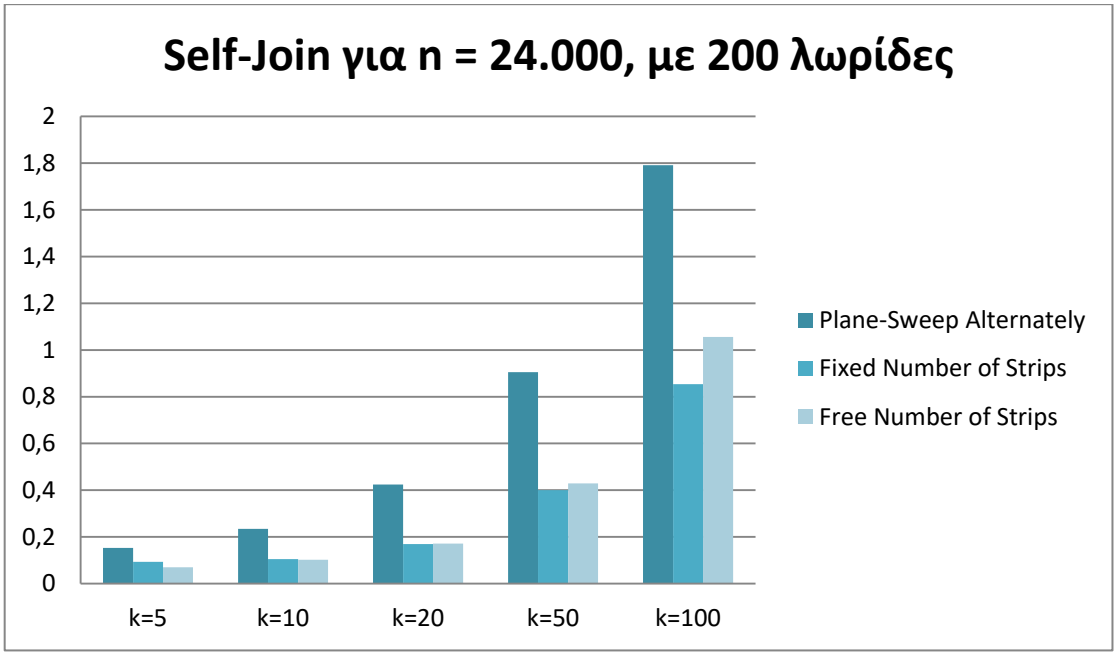

***Γράφημα 15.*** Self-Join για n = 14.000, με 100 λωρίδες.



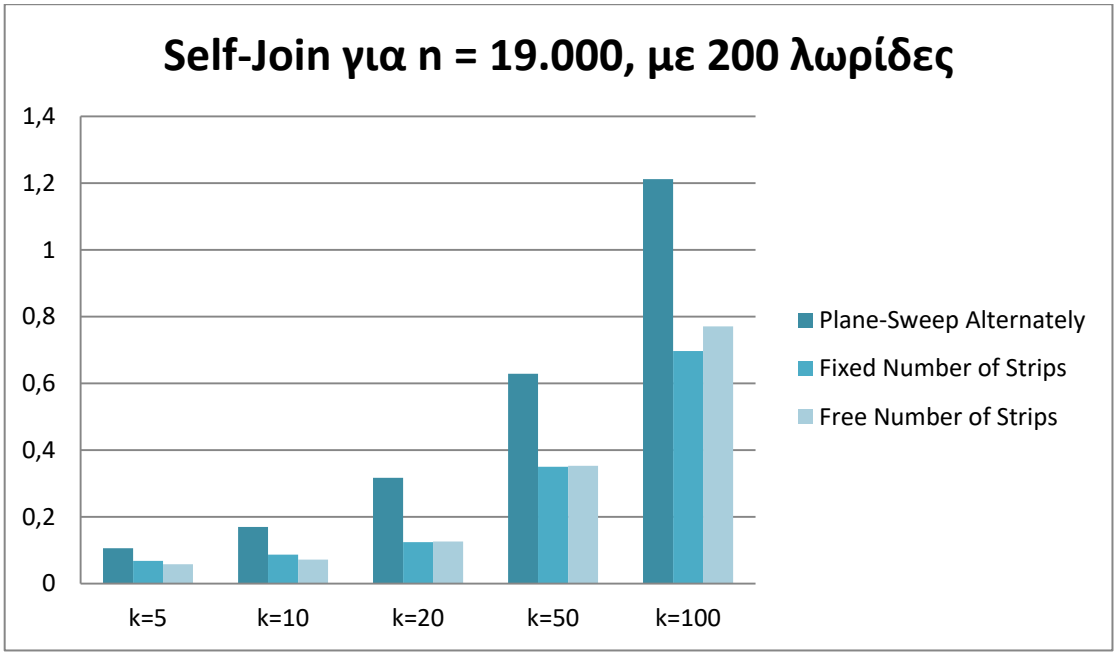

***Γράφημα 16.*** Self-Join για n = 9.000, με 100 λωρίδες.

## 4.5 ΠΕΙΡΑΜΑΤΑ ΜΕ ΑΡΙΘΜΟ ΛΩΡΙΔΩΝ ΙΣΟ ΜΕ 200

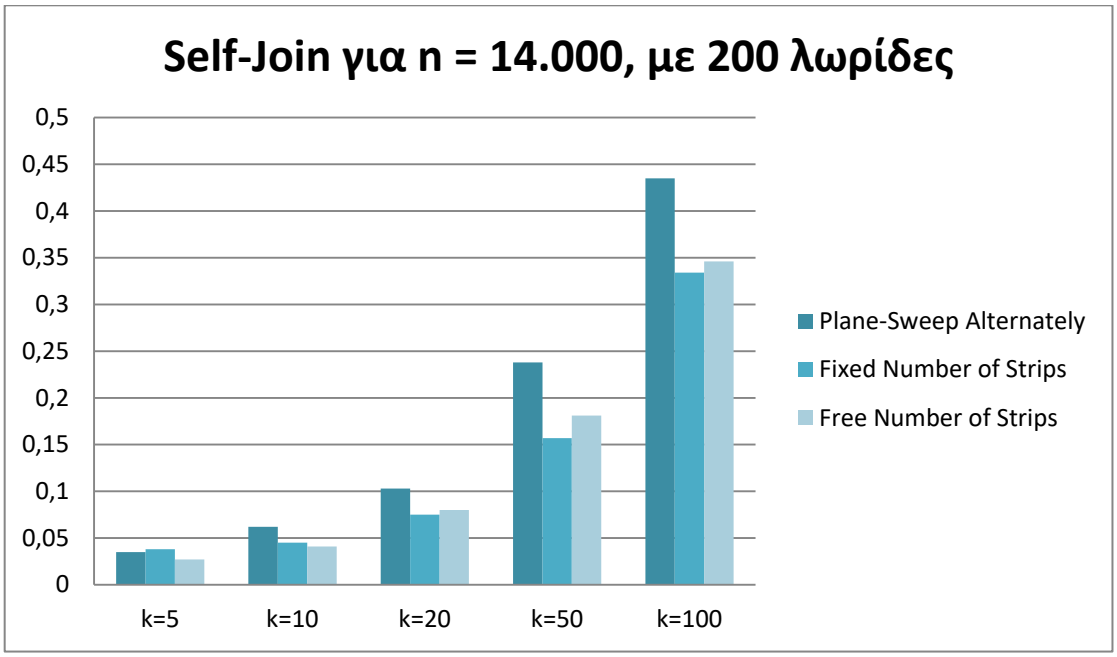
Στην τελευταία σειρά πειραμάτων αυτού του είδους, αυξάνουμε τις λωρίδες σε 200. Κατ' αντιστοιχία με τα προηγούμενα αποτελέσματα και εδώ βλέπουμε ότι η κλιμάκωση του αλγορίθμου με λωρίδες σταθερού ύψους είναι καλύτερη και μάλιστα για k>20 είναι ο ταχύτερος.



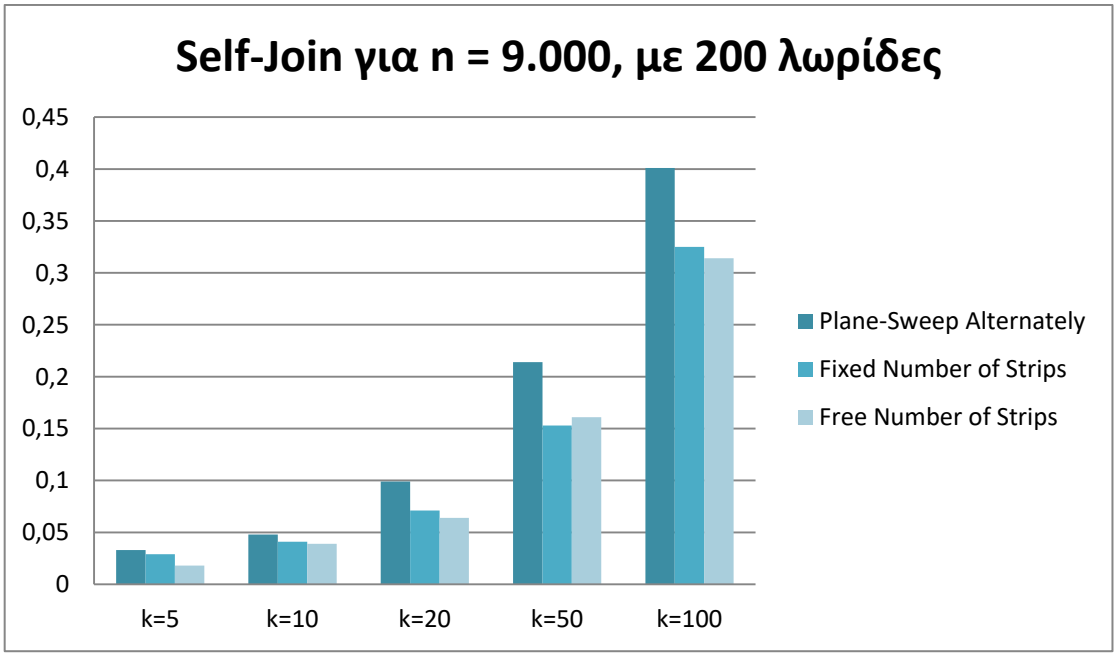

***Γράφημα 17.*** Self-Join για n = 24.000, με 200 λωρίδες.




***Γράφημα 18.*** Self-Join για n = 19.000, με 200 λωρίδες.



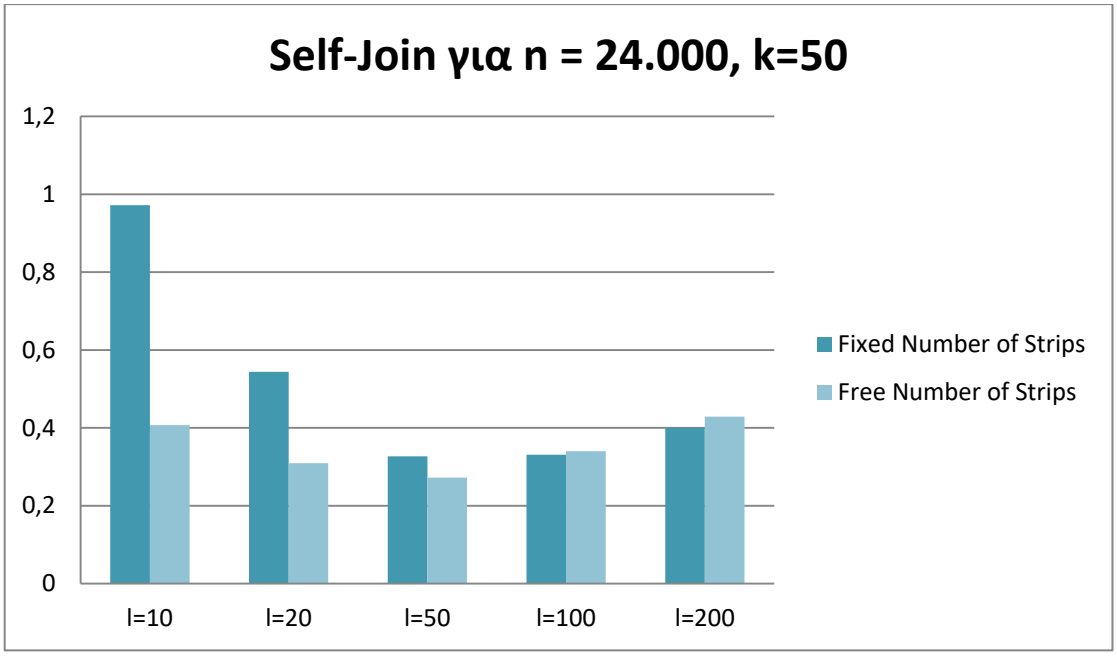

***Γράφημα 19.*** Self-Join για n = 14.000, με 200 λωρίδες.



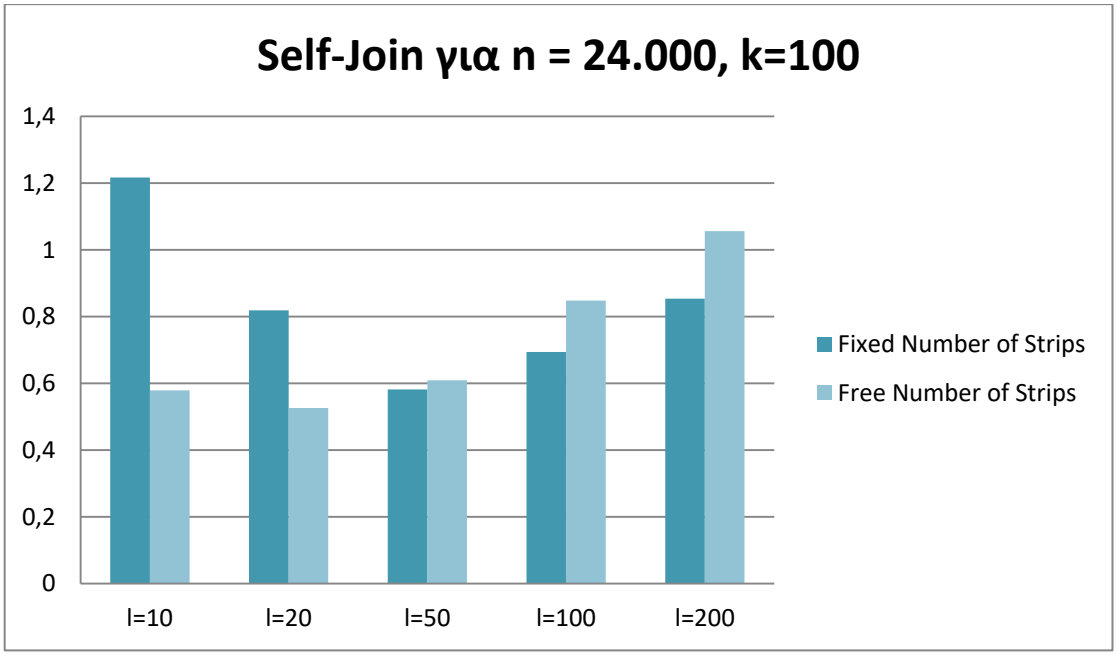

***Γράφημα 20.*** Self-Join για n = 9.000, με 200 λωρίδες.

## 4.6 ΑΝΑΖΗΤΗΣΗ ΒΕΛΤΙΣΤΟΥ ΑΡΙΘΜΟΥ ΛΩΡΙΔΩΝ

Στα ακόλουθα πειράματα αναζητούσαμε το βέλτιστο αριθμό λωρίδων για την ταχύτερη εκτέλεση. Έτσι, συγκρίναμε τους αλγόριθμους με λωρίδες για αναζήτηση 50 και 100 κοντινότερων γειτόνων. Για παραμέτρους n=24.000 και k=50, ο βέλτιστος αριθμός λωρίδων προσεγγίζει τις 50, ενώ για παραμέτρους n=24.000 και k=100, είναι ανάμεσα στις 20 με 50.

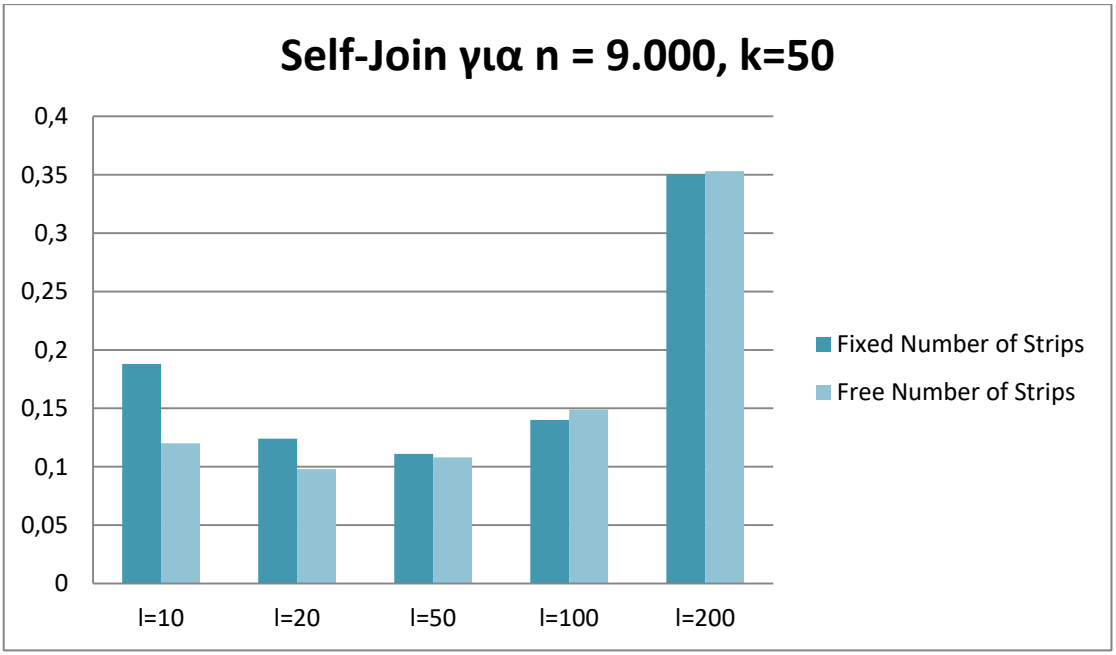



***Γράφημα 21.*** Self-Join για n = 24.000, k=50.

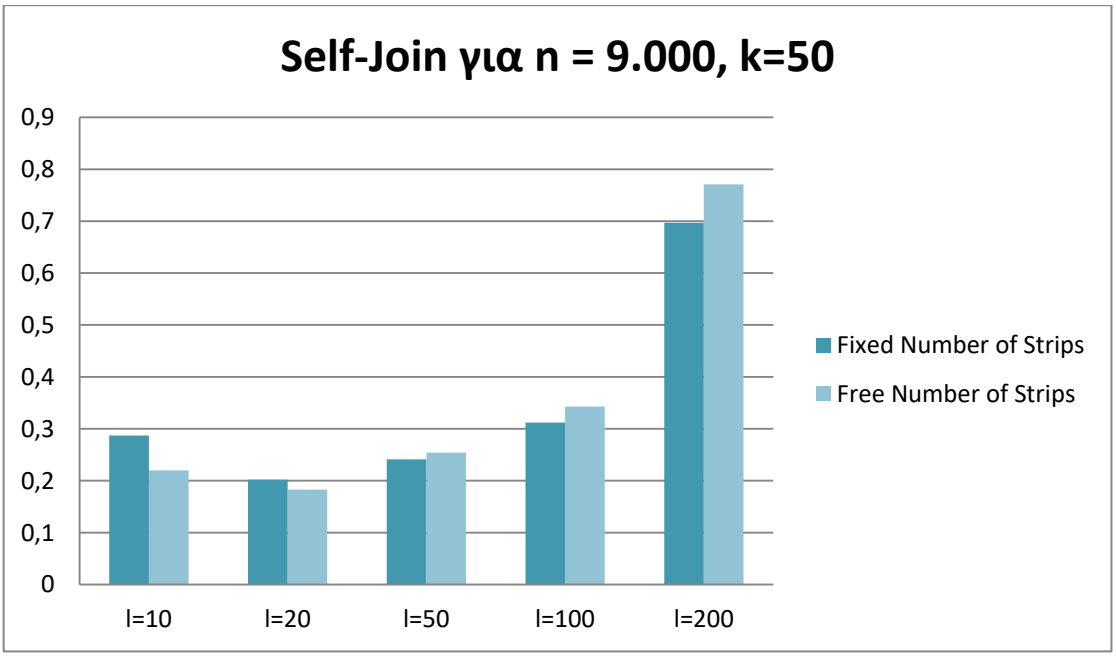



***Γράφημα 22.*** Self-Join για n = 24.000, k=100.

Αντίστοιχα και εδώ, κάνουμε τους ίδιους ελέγχους, για πολύ μικρότερο dataset που αποτελείται από 9.000 σημεία. Τα αποτελέσματα είναι παρόμοια με τα προηγούμενα, αναδεικνύοντας τους αλγόριθμους βέλτιστους για l που κυμαίνεται μεταξύ 20-50.




***Γράφημα 23.*** Self-Join για n = 0.000, k=50.




***Γράφημα 24.*** Self-Join για n = 24.000, k=100.

# 5. ΣΥΜΠΕΡΑΣΜΑΤΑ ΚΑΙ ΕΠΕΚΤΑΣΕΙΣ

Στο κεφάλαιο αυτό θα παρουσιάσουμε τα συνολικά συμπεράσματα που προέκυψαν μετά την πειραματική διαδικασία καθώς και κάποιες προτάσεις για μια μελλοντική επέκταση της εργασίας αυτής.

## 5.1 ΣΥΝΟΛΙΚΑ ΣΥΜΠΕΡΑΣΜΑΤΑ ΤΗΣ ΕΡΓΑΣΙΑΣ

Στην παρούσα διπλωματική εργασία παρουσιάσαμε 3 νέους αλγόριθμους για Join και Self-Join όλων των k-κοντινότερων γειτόνων: Τον Plane-Sweep Alternately, Plane-Sweep με λωρίδες ίσου ύψους και Plane-Sweep με λωρίδες ίσου αριθμού σημείων. Εκμεταλλευτήκαμε το γεγονός ότι εάν μια απόσταση είναι μεγαλύτερη από την άλλη, τότε και το τετράγωνο αυτής θα είναι επίσης μεγαλύτερο, με αποτέλεσμα να μη χρειάζεται να κάνουμε ακριβές υπολογιστικές πράξεις με ρίζες. Επίσης, με βάση το αντικείμενο που ψάχνουμε τους γείτονές του, ελέγχουμε εναλλάξ αριστερά και δεξιά του, αφού εκεί είναι το πιθανότερο να βρίσκονται όλοι οι γείτονες. Τέλος, ορίσαμε την έννοια του χωρισμού του χώρου σε λωρίδες, για καλύτερη και αυτόνομη διαχείριση κάθε λωρίδας ξεχωριστά.

Τα αποτελέσματα των πειραμάτων έδειξαν διαφορετικούς αλγόριθμους ως καλύτερους, ανάλογα την αναζήτηση που θέλουμε να κάνουμε. Ο Plane-Sweep Alternately είναι καλός για μικρό όγκο δεδομένων, καθώς γλιτώνει τις πράξεις χωρισμού σε λωρίδες και ξεκινάει κατευθείαν με τον υπολογισμό. Καθώς όμως τα δεδομένα αυξάνονται, δεν έχει καλή κλιμάκωση, και εκεί καλύτερη επίδοση έδειξαν οι άλλοι 2 αλγόριθμοι. Ο Plane-Sweep με λωρίδες σταθερού ύψους δεν αποδείχτηκε καλός για μικρό όγκο δεδομένων καθώς είναι πολύ πιθανό να υπάρχουν λωρίδες χωρίς καθόλου σημεία, πράγμα που τον αναγκάζει να ψάχνει σε άλλες λωρίδες και να ξοδεύει υπολογιστικό χρόνο. Ωστόσο, έχει πολύ καλή κλιμάκωση, με αποτέλεσμα καθώς τα δεδομένα αυξάνονται, γίνεται αναλογικά καλύτερος. Τέλος, ο Plane-Sweep με λωρίδες ίσου πλήθους αντικειμένων είναι ο ταχύτερος για μικρό και μεσαίο όγκο δεδομένων και έχει αρκετά καλή κλιμάκωση. Για μεγάλο όγκο δεδομένων όμως, είναι ελάχιστα πιο αργός από τον προηγούμενο αλγόριθμο. Τα αποτελέσματα αυτά βέβαια έγιναν με βάσεις δεδομένων πραγματικών σημείων, χωρίς να εξετάσουμε τη συμμετρία ή τη διασπορά τους. Σε άλλες βάσεις δεδομένων είναι πιθανό να έχουν διαφορετική συμπεριφορά.

Έπειτα ελέγξαμε ποιος είναι ο βέλτιστος αριθμός λωρίδων για πιο ταχύ υπολογισμό. Τα πειράματα έδειξαν ότι για δεδομένα μεταξύ 9.000 και 24.000 και με

αναζήτηση 50 και 100 κοντινότερων γειτόνων, ο βέλτιστος αριθμός λωρίδων κυμαίνεται μεταξύ 20 και 50 και στους 2 αλγόριθμους.

## 5.2 ΠΡΟΤΑΣΕΙΣ ΓΙΑ ΠΙΘΑΝΕΣ ΕΠΕΚΤΑΣΕΙΣ

Στα πλαίσια αυτής της εργασίας είδαμε ότι οι αλγόριθμοι αυτοί είναι αποδοτικοί για σχετικά μικρό όγκο δεδομένων. Οι περιορισμοί της μνήμης RAM δεν μας επιτρέπει να διαχειριστούμε μεγάλο όγκο δεδομένων με τις υλοποιήσεις που κάναμε. Μια επέκταση που θα μπορούσε να υλοποιηθεί είναι να γίνει παράλληλη υλοποίηση της λογικής των αλγορίθμων αυτών. Κάθε λωρίδα θα μπορεί να υπολογίζεται ανεξάρτητα, και στο τέλος θα συλλέγονται και θα ελέγχονται τα αποτελέσματα. Έτσι, η κάθε λωρίδα θα μπορεί να εκτελείται παράλληλα, μειώνοντας δραματικά το χρόνο υπολογισμού. Μια άλλη επέκταση θα μπορούσε να είναι η διαχείριση των δεδομένων από το σκληρό δίσκο. Να παίρνουμε δηλαδή μέρος των δεδομένων στη RAM κάθε φορά, να βρίσκουμε τα αντίστοιχα αποτελέσματα και έπειτα να το αποθηκεύουμε πίσω στο σκληρό δίσκο. Κατόπιν, να εισάγουμε νέο μέρος των δεδομένων και να επαναλαμβάνουμε. Τέλος, θα μπορούσαμε να κάνουμε μια κατανεμημένη υλοποίηση σε περισσότερους από έναν υπολογιστικούς κόμβους με το μοντέλο MapReduce και τη χρήση του Hadoop. Με αυτό τον τρόπο μπορούσαμε να υπολογίσουμε Self-Join και Join ερωτήματα για big data.

# ΒΙΒΛΙΟΓΡΑΦΙΑ


1. Rigaux, Philippe, Michel Scholl, and Agnès Voisard. Spatial databases: with application to GIS. San Francisco: Morgan Kaufmann, 2011.
2. Roumelis, George, Michael Vassilakopoulos, Antonio Corral, and Yannis Manolopoulos. "A New Plane-Sweep Algorithm for the K-Closest-Pairs Query." SOFSEM 2014: Theory and Practice of Computer Science Lecture Notes in Computer Science (2014): 478-90.
3. A. Guttman, R-trees: A Dynamic Index Structure for Spatial Searching, ACM SIGMOD Intl. Conference on Management of Data, 1984.
4. T. Brinkhoff, H.P. Kriegel, B. Seeger, Efficient Processing of Spatial Joins Using R-trees, ACM SIGMOD Intl. Conference on Management of Data, 1993.
5. G. R. Hjaltason, H. Samet, Distance Browsing in Spatial Databases. ACM Transactions On Database Systems (TODS), 24(2): 265-318, 1999.
6. Zhang, Jun, N. Mamoulis, D. Papadias, and Yufei Tao. "All-nearest-neighbors queries in spatial databases." Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.
7. Altman, N. S. "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression." The American Statistician 46.3 (1992): 175-85.
8. Chen, Hue-Ling, and Ye-In Chang. "All-nearest-neighbors finding based on the Hilbert curve." Expert Systems with Applications 38.6 (2011): 7462-475.
9. Kamath, Surekha. "Fuzzy Logic for Breast Cancer Diagnosis Using Medical Thermogram Images." Fuzzy Expert Systems for Disease Diagnosis (n.d.): 168-99.
10. Sanchez-Morillo, D., M. A. Fernandez-Granero, and A. Leon-Jimenez. "Use of predictive algorithms in-home monitoring of chronic obstructive pulmonary disease and asthma: A systematic review." Chronic Respiratory Disease 13.3 (2016): 264-83.
11. “Code::Blocks.” Code::Blocks IDE, www.codeblocks.org/.
12. MortenMacFly. “Download binary.” Code::Blocks IDE, www.codeblocks.org/downloads/26.