Α Ρ Ι Σ Τ Ο Τ Ε Λ Ε Ι Ο Π Α Ν Ε Π Ι Σ Τ Η Μ Ι Ο
Θ Ε Σ Σ Α Λ Ο Ν Ι Κ Η Σ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΉΜΑ ΠΛΗΡΟΦΟΡΙΚΉΣ

Π Τ Υ Χ Ι Α Κ Η / Δ Ι Π Λ Ω Μ Α Τ Ι Κ Η
Ε Ρ Γ Α Σ Ι Α

«ΥΠΟΛΟΓΙΣΜΟΣ ΚΟΡΥΦΟΓΡΑΜΜΗΣ ΜΕ ΑΝΑΖΗΤΗΣΗ ΠΛΗΣΙΕΣΤΕΡΟΥ ΓΕΙΤΟΝΑ ΣΕ R*-TREE»

Skyline computation by nearest neighbor search in R*-trees

«ΚΙΟΣΙΔΗΣ ΧΑΡΑΛΑΜΠΟΣ»

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ:
Παπαδόπουλος Απόστολος, Επίκουρος Καθηγητής

ΘΕΣΣΑΛΟΝΙΚΗ ΣΕΠΤΕΜΒΡΙΟΣ 2013

Περίληψη

Η παρούσα εργασία πραγματεύεται το θέμα του υπολογισμού της κορυφογραμμής. Η κορυφογραμμή έχει σαν στόχο να απομονώσει τα καλύτερα στοιχεία, από ένα μεγάλο σύνολο δεδομένων, ώστε να δοθεί στον χρήστη η δυνατότητα επιλογής από ένα πολύ μικρότερο σύνολο.

Η συνεχής συσσώρευση της πληροφορίας, η ανάγκη για αποδοτικότερη επεξεργασία και ανάλυση της με στόχο γρηγορότερες λειτουργίες και βελτιωμένο περιεχόμενο προς τους χρήστες αποτελεί αναγκαιότητα σε ένα ολοκληρωμένο σύστημα. Η κορυφογραμμή είναι μια από τις μεθόδους που δίνουν αυτή την δυνατότητα.

Η εργασία υλοποιήθηκε σε java, χρησιμοποιήθηκε μια δομή R*-tree, η οποία σαν σκοπό έχει την καλύτερη ομαδοποίηση των δεδομένων με βάση την θέση τους στον χώρο. Εκτός από ορισμένες αλλαγές στη βιβλιοθήκη της δομής για την προσαρμογή της στο πρόβλημα μας, υλοποιήθηκε ο κύριος αλγόριθμος για την εύρεση των σημείων, ένας αλγόριθμος πλησιέστερου γείτονα με δυνατότητα αναζήτηση σε περιοχές και ορισμένες βοηθητικές κλάσεις για την λειτουργία των παραπάνω.

Abstract

This paper addresses the issue of the Skyline Computation. The Skyline aims to isolate the best elements from a large set of data, in order to provide to the user a much smaller data set, thus an easier decision to make.

The constant accumulation of information, the need for efficient processing and analysis of this information, in terms of faster operations and improved user content is a necessity in a complete system. The Skyline is one of the methods that provide this possibility.

The project was implemented in java, an R*-tree spatial index was used to ensure better data clustering. Except for the changes in the java library of the index, the main algorithm for finding Skyline points was implemented as well as, a nearest neighbor algorithm that searches in regions, also certain auxiliary classes were implemented for the operation of the above.

Ευχαριστίες

Πριν την παρουσίαση των αποτελεσμάτων της παρούσας εργασίας, αισθάνομαι την υποχρέωση να ευχαριστήσω τον κ. Απόστολο Παπαδόπουλο, του οποίου η συμβολή έπαιξε πολύ σημαντικό ρόλο στην πραγματοποίησή της. Πιο συγκεκριμένα ευχαριστώ για τις υποδείξεις και παρατηρήσεις, καθώς και την υπομονή που δείξατε όλον αυτόν τον χρόνο. Επίσης θα ήθελα να ευχαριστήσω την οικογένεια μου για την συνεχής καθοδήγηση και υποστήριξη, καθώς και τους φίλους μου που με ανέχτηκαν όλο αυτό το διάστημα. Τα καλύτερα έρχονται.

9/29/2013

Κιοσίδης Χαράλαμπος

Περιεχόμενα

ΚΕΦΑΛΑΙΟ 1: Εισαγωγή

ΕΙΣΑΓΩΓΗ

Αντικείμενο της παρούσας εργασίας είναι η υλοποίηση του υπολογισμού της κορυφογραμμής με χρήση της δομής R*-tree και έναν αλγόριθμο πλησιέστερου γείτονα.

Η εργασία δομείται σε κεφάλαια ως εξής:

- Στο Κεφάλαιο 2 γίνεται μια επεξήγηση της έννοιας και των τρόπων υπολογισμού της κορυφογραμμής με παραδείγματα και εφαρμογές.
- Στο Κεφάλαιο 3 γίνεται αναφορά στις χωρικές βάσεις, ανάλυση των δομών R δέντρο και R* δέντρο καθώς και οι διαφορές τους.
- Στο Κεφάλαιο 4 αναλύεται ο τρόπος σύνδεσης των παραπάνω, ο αλγόριθμος που υλοποιήθηκε, παρατηρήσεις και προϋποθέσεις.
- Στο Κεφάλαιο 5 παραθέτονται τα αποτελέσματα από τα πειράματα.
- Στο Παράρτημα Ι παρουσιάζονται αλφαβητικά η βιβλιογραφία και οι δικτυακοί τόποι που αναφέρονται στην εργασία.
- Στο Παράρτημα ΙΙ παρουσιάζεται η υλοποιημένη εργασία.

ΚΕΦΑΛΑΙΟ 2: Κορυφογραμμή

ΚΟΡΥΦΟΓΡΑΜΜΗ (SKYLINE)

Το κεφάλαιο αυτό αποτελεί περιγραφή της κορυφογραμμής μέσα από παραδείγματα, ορισμένες εφαρμογές και τρόπου υπολογισμού της.

Αρχικά γίνεται επεξήγηση της έννοιας της κορυφογραμμής μέσω ενός παραδείγματος. Έπειτα αναφέρονται μερικά θέματα που μπορεί να έχει χρησιμότητα η εύρεση της κορυφογραμμής καθώς και τρόποι υπολογισμού της.

2 ΚΟΡΥΦΟΓΡΑΜΜΉ

Για την κατανόηση της έννοια της κορυφογραμμής θα χρησιμοποιήσουμε ένα παράδειγμα. Από τα πιο χαρακτηριστικά παραδείγματα υπολογισμού και χρησιμότητας της κορυφογραμμής αποτελεί το παράδειγμα των ξενοδοχείων.

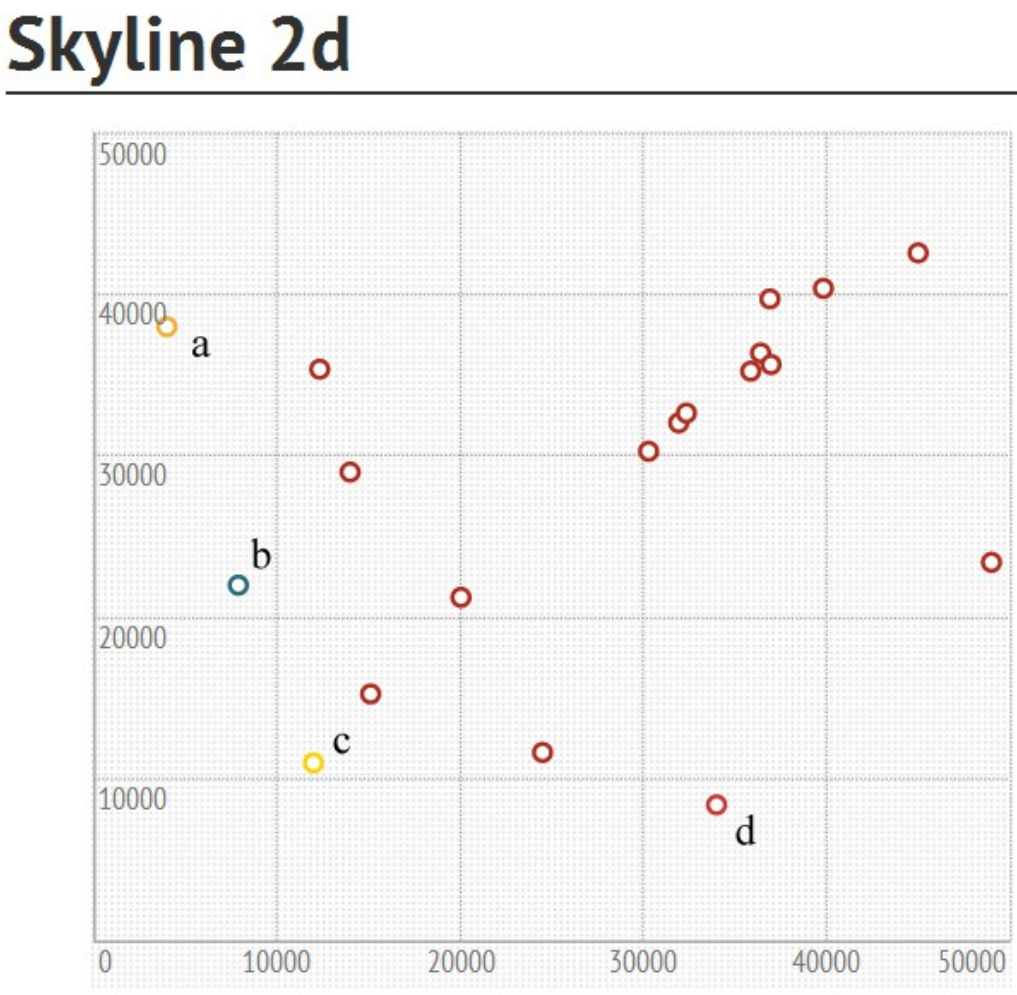
Έχουμε ένα σύνολο δεδομένων που περιέχει πληροφορίες για ξενοδοχεία. Τα δεδομένα αποτελούνται από 2 διαστάσεις , μια η απόσταση από την θάλασσα και δεύτερη η τιμή του ξενοδοχείου.

Σκεφτείτε έναν δισδιάστατο άξονα με την απόσταση και την τιμή να αναπαριστούν τους άξονες X και Y αντίστοιχα.

Στόχος της αναζήτησης είναι να βρεθεί το ξενοδοχείο που έχει ελάχιστη τιμή και απόσταση από την θάλασσα. Το σύνολο των δεδομένων μας μπορεί να περιέχει παραπάνω από ένα σημείο που ικανοποιούν τον περιορισμό.

Στο σχήμα 1 τα έντονα σημεία αποτελούν την κορυφογραμμή του συνόλου ενώ τα υπόλοιπα ξενοδοχεία δεν αποτελούν μέρος της κορυφογραμμής γιατί κυριαρχούνται είτε σε τιμή είτε σε απόσταση από τουλάχιστον ένα ξενοδοχείο που αποτελεί μέρος της κορυφογραμμής.

Στην περίπτωση μας τα σημεία a,b,c,d αποτελούν το σύνολο των ενδιαφέρων ξενοδοχείων, με το a να έχει την ελάχιστη απόσταση από την θάλασσα, το c την μικρότερη τιμή, ενώ τα b και d έχουν στη μια από της δυο διαστάσεις μικρότερη τιμή από τα a και c επομένως δεν κυριαρχούνται από αυτά και αποτελούν μέρος της κορυφογραμμής.



Αναπαράσταση του παραδείγματος

Η συγκεκριμένη κορυφογραμμή έχει μεγάλη χρησιμότητα σε ταξιδιωτικούς πράκτορες, αφού τους δίνει την δυνατότητα να περιορίσουν τα ενδιαφέροντα ξενοδοχεία, να βελτιώσουν τις προσφορές τους και να παρουσιάσουν καλύτερες επιλογές στους πελάτες.

2.1 Εφαρμογές της κορυφογραμμής

Τα ερωτήματα κορυφογραμμής δεν περιορίζονται σε 2 μόνο διαστάσεις ενώ η αυξανόμενη χρήση των εφαρμογών σε κινητά τηλέφωνα και η αύξηση των χρηστών τα καθιστούν αναγκαίο εργαλείο.

Μερικές εφαρμογές είναι σε υπηρεσίες προτάσεων ξενοδοχείων, χώρων αναψυχής γενικότερα, εστιατορίων ακόμη και δρομολογίων σε συνδυασμό με την χρήση της τρέχουσας θέσης του χρήστη.

Άλλες εφαρμογές είναι σε θέματα λήψεων αποφάσεων, σε θέματα προαγωγής μελών σε κοινωνικά δίκτυα, δραστηριότητες των μετοχών, πρόγνωση καιρού.

Επιπλέον η οπτικοποίηση δεδομένων γενικότερα και η βελτιστοποίηση κατανεμημένων ερωτημάτων αποκτούν νέες δυνατότητες.

Αναφορικά η κορυφογραμμή μπορεί να αναφερθεί και ως καμπύλη pareto ή μέγιστο διάνυσμα.

2.2 Τρόποι υπολογισμού

Ορισμένες τεχνικές για αξιολόγηση των ερωτημάτων κορυφογραμμής είναι:

Block Nested Loop: εκτελεί μια φωλιασμένη επανάληψη σε όλα τα μπλοκ των δεδομένων.

Divide and Conquer: χωρίζεται ο χώρος σε υποχώρους, λύνεται το πρόβλημα στους υποχώρους και στη συνέχεια συνδυάζονται οι λύσεις για όλο τον χώρο.

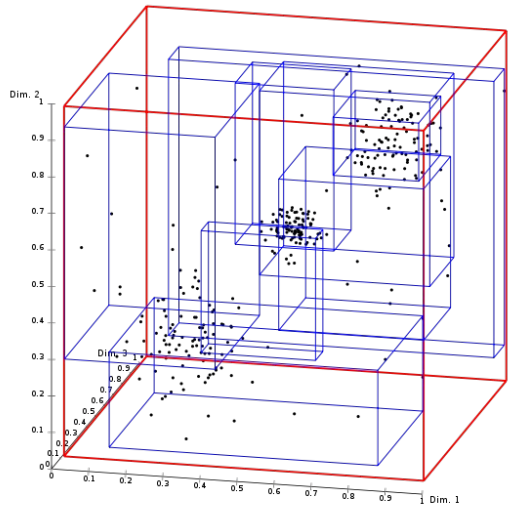
Plane-sweep: ο χώρος σαρώνεται ξεκινώντας από τους άξονες και αναζητεί με βάση μια νοητή γραμμή που μετατοπίζεται στον ευκλείδειο χώρο.

Nearest Neighbor Search: χρησιμοποιεί ένα R-tree κατάλογο και εκτελεί μια σειρά ερωτημάτων πλησιέστερου γείτονα για διαφορετικές περιοχές μέχρις ότου όλα τα αντικείμενα της κορυφογραμμής έχουν βρεθεί.

Branch and Bound Skyline: χρησιμοποιείτε πάλι ένα R-tree. Κατασκευάζεται μια σειρά προτεραιότητας που ταξινομεί τα αντικείμενα με βάση μιας τιμής MinDist και της απόστασης από την αρχή.

ΚΕΦΑΛΑΙΟ 3: Δομή R*-tree

ΔΟΜΗ R*-TREE



Η τεχνική αξιολόγησης Nearest Neighbor Search που υλοποιήσαμε σε java προϋποθέτει την αποθήκευση των δεδομένων με χρήση χωρικής δομής. Η δομή που χρησιμοποιήσαμε είναι το R*-tree. Το κεφάλαιο αυτό περιέχει μια αναφορά στις χωρικές βάσεις δεδομένων. Εκτενέστερη ανάλυση της δομής R-tree και των διαφορών της με την R*-tree που χρησιμοποιήθηκε.

3 Χωρική Βάση Δεδομένων

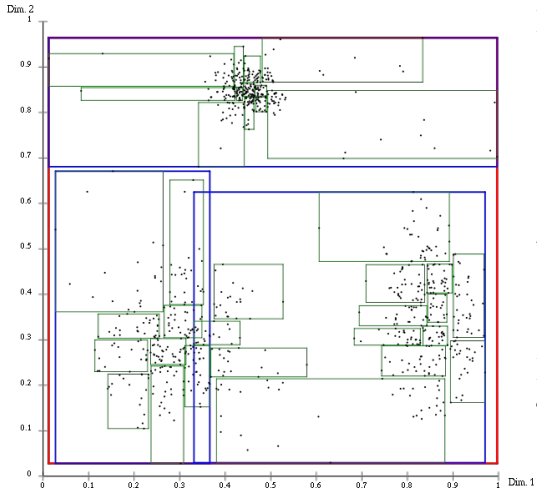
Η χωρική βάση δεδομένων είναι μια βάση δεδομένων που έχει βελτιστοποιηθεί για την αποθήκευση και αναζήτηση δεδομένων που αντιπροσωπεύουν αντικείμενα καθορισμένα σε ένα γεωμετρικό χώρο. Οι περισσότερες χωρικές βάσεις δεδομένων επιτρέπουν απλά γεωμετρικά αντικείμενα όπως σημεία, γραμμές και πολύγωνα.

Οι χωρικοί κατάλογοι χρησιμοποιούνται από χωρικές βάσεις δεδομένων για την βελτιστοποίηση χωρικών ερωτημάτων. Τα συμβατικά είδη πινάκων δεν χειρίζονται αποτελεσματικά χωρικά ερωτήματα όπως αν ορισμένα σημεία βρίσκονται στην ίδια χωρική περιοχή ενδιαφέροντος.

Πολλές εφαρμογές χρειάζονται την οργάνωση και διαχείριση πολυδιάστατων δεδομένων. Για την αποτελεσματική αναζήτηση των ερωτημάτων τα δεδομένα θα πρέπει να οργανώνονται με χρήση κάποιου συστήματος δεικτών ώστε να επιταχυνθεί η επεξεργασία τους.

3.1 R-tree

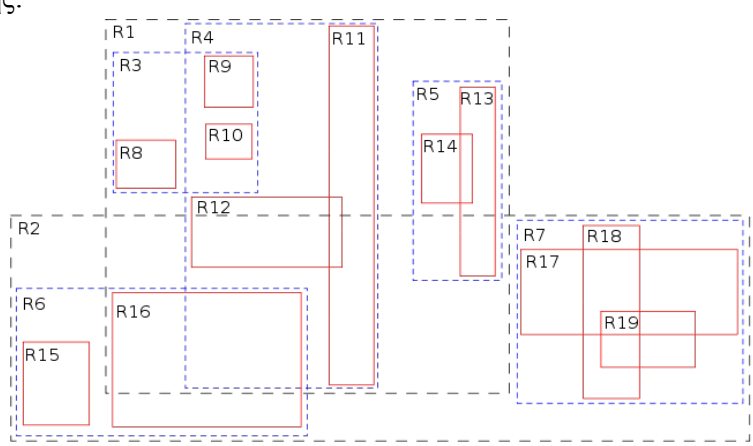
Τα R-δέντρα είναι δεντρικές δομές δεδομένων που χρησιμοποιούνται για χωρικές μεθόδους πρόσβασης, δηλαδή για εύρεση πολύ-διάστατης πληροφορίας, όπως γεωγραφικές συντεταγμένες, ορθογώνια ή πολύγωνα.

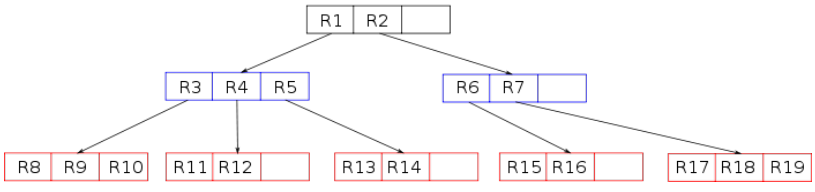


Η κύρια ιδέα της δομής είναι η ομαδοποίηση κοντινών αντικειμένων και η εκπροσώπηση τους από το ελάχιστο ορθογώνιο στο οποίο ανήκουν τα δεδομένα. Δεδομένου ότι τα αντικείμενα βρίσκονται μέσα στο ορθογώνιο, ένα ερώτημα που δεν τέμνει το ορθογώνιο οριοθέτησης, δεν μπορεί να τέμνει οποιοδήποτε από τα αντικείμενα που περιέχονται σε αυτό.

Πιο αναλυτικά η είσοδος της αναζήτηση γίνεται με δημιουργία ενός ορθογωνίου ερωτημάτων (Query Box). Η αναζήτηση ξεκινά από τον κόμβο ρίζα του δέντρου. Κάθε εσωτερικός κόμβος περιέχει ένα σύνολο από ορθογώνια και πληροφορίες για τους αντίστοιχους κόμβους παιδιά και κάθε κόμβος φύλο περιέχει τα ορθογώνια των χωρικών αντικειμένων.

Κάθε ορθογώνιο σε κόμβο ελέγχεται αν επικαλύπτει το ορθογώνιο της αναζήτησης . Αν το επικαλύπτει πρέπει να εξεταστεί και ο αντίστοιχος κόμβος παιδί. Η αναζήτηση συνεχίζεται με αυτόν τον αναδρομικό τρόπο έως ότου όλοι οι επικαλυπτόμενοι κόμβοι έχουν εξεταστεί. Όταν η διαδικασία συναντήσει κόμβο φύλο τα ορθογώνια που περιέχει ελέγχονται με το ορθογώνιο αναζήτησης και τα αντικείμενα αυτών (αν υπάρχουν) και εισάγονται στο σύνολο αποτελεσμάτων αν ανήκουν και στο ορθογώνιο αναζήτησης.





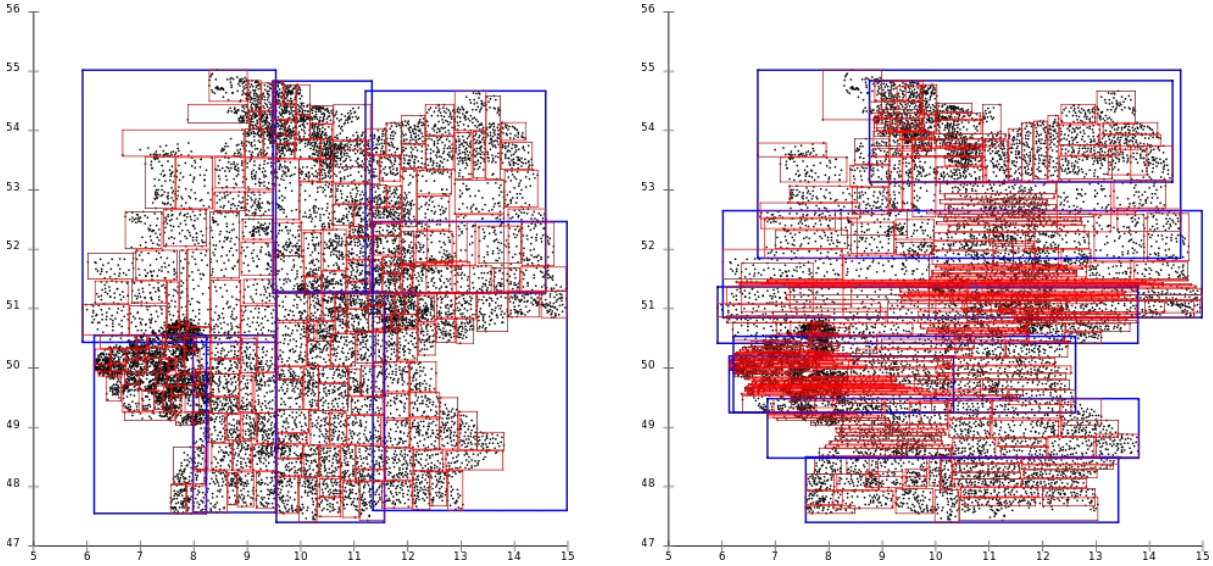
Παράδειγμα R-tree στον χώρο και δενδρική αναπαράσταση

3.2 R-tree*

Τα R* δέντρα αποτελούν μια παραλλαγή των R δέντρων, χρησιμοποιούν τους ίδιους αλγορίθμους για την αναζήτηση και διαγραφή. Η ελαχιστοποίηση της κάλυψης και επικάλυψης έχει κομβική σημασία για την απόδοση των R δέντρων. Η επικάλυψη σημαίνει ότι κατά την αναζήτηση ή την εισαγωγή δεδομένων περισσότερα από ένα κλαδιά του δέντρου πρέπει να επεκταθούν. Η ελαχιστοποίηση της κάλυψης βελτιώνει την απόδοση του κλαδέματος , επιτρέποντας να αποκλείονται ολόκληρες σελίδες από την διαδικασία αναζήτησης πιο συχνά.

Η δομή R* δέντρο επιχειρεί να μειώσει και τα δυο, χρησιμοποιώντας έναν συνδυασμό ενός αλγόριθμου διάσπασης κόμβου και την έννοια της αναγκαστικής επανένταξης κατά την υπερχείλιση κόμβων. Αυτό βασίζεται στην παρατήρηση ότι η δομή του R δέντρου εξαρτάται πολύ από την σειρά με την οποία γίνονται οι εισαγωγές, η διαδικασία διαγραφής και επανένταξης των αντικειμένων επιτρέπει στο σύστημα να βρει μια θέση στο δέντρο που πιθανώς να είναι πιο κατάλληλη από την αρχική.

Όταν ένας κόμβος υπερχειλίσει , ένα ποσοστό των στοιχείων του αφαιρείτε από τον κόμβο και επανεντάσσετε στο δέντρο, με αποτέλεσμα πιο συνεκτικά συγκεντρωμένες ομάδες και μεγάλη μείωση της επικάλυψης κόμβων, στρατηγική εμπνευσμένη από την έννοια της εξισορρόπησης ενός B δέντρου.



Επίδραση διαφορετικών αλγορίθμων τμηματοποίησης
Αριστερά τοπολογική τμηματοποίηση
Δεξιά quadratic Guttman τμηματοποίηση

ΚΕΦΑΛΑΙΟ 4: Αλγόριθμος

ΑΛΓΌΡΙΘΜΟΣ

Στο κεφάλαιο αυτό γίνεται ανάλυση του αλγορίθμου που υλοποιήθηκε και των περιφερειακών για την ορθή λειτουργία του, αναφέρονται παρατηρήσεις και προϋποθέσεις. Επιπλέον αναφέρονται προβλήματα και δυσκολίες στην εκπόνηση που προέκυψαν για την εκπόνηση του αλγορίθμου.

Τα ανοιχτά ζητήματα που θα μπορούσαν να διερευνηθούν από μια μελλοντική επέκταση της παρούσας εργασίας είναι +++

4 Προϋποθέσεις και Παρατηρήσεις

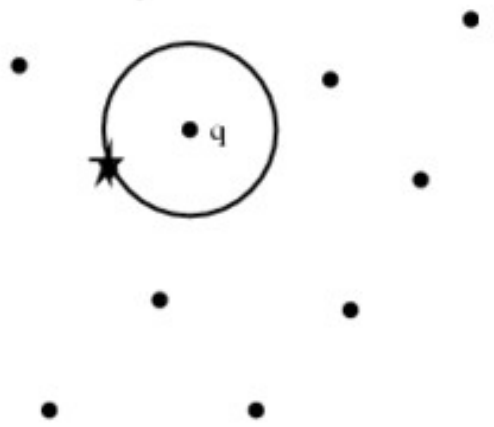
Παρακάτω θα γίνει περιγραφή του αλγορίθμου. Για ευκολία όλες οι τιμές είναι θετικοί αριθμοί, η f είναι μια αυθαίρετη μονότονη συνάρτηση απόστασης, το D είναι ένα δισδιάστατο σύνολο δεδομένων.

Παρατήρηση 1: Αν ένα σημείο n το οποίο ανήκει στο D είναι πλησιέστερο γείτονας στην αρχή των αξόνων σύμφωνα με την f, τότε το n ανήκει στην κορυφογραμμή του D.

Παρατήρηση 2: Όμοια με την παρατήρηση 1, αν ένα σημείο m που ανήκει σε ένα υποσύνολο του D, το Dm, είναι πλησιέστερος γείτονας με βάση την f, τότε ανήκει στην κορυφογραμμή του D.

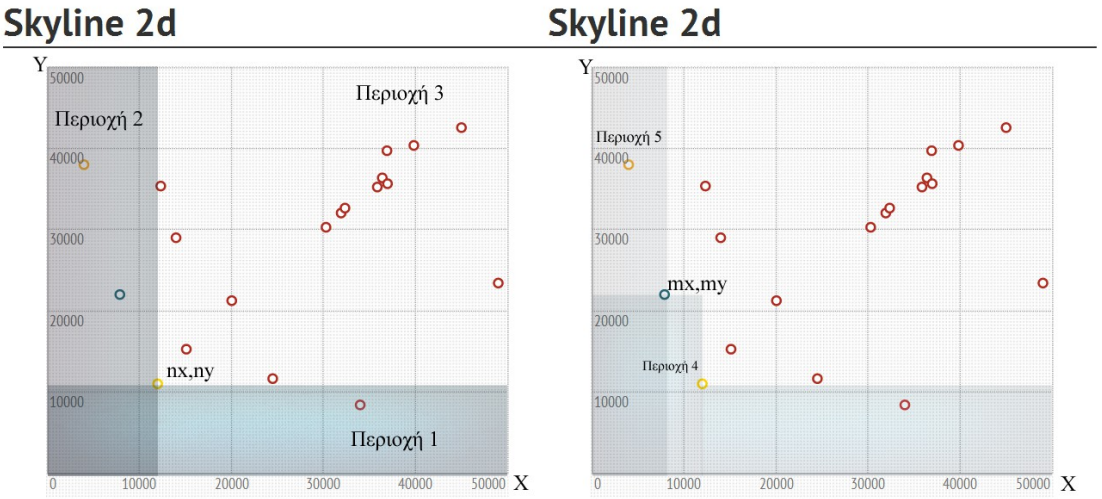
4.1 Αλγόριθμος Πλησιέστερου Γείτονα

Αναζήτηση του πλησιέστερου γείτονα (NNS), είναι ένα πρόβλημα βελτιστοποίησης για την εύρεση πλησιέστερων σημείων σε μετρικούς χώρους. Το πρόβλημα δίνεται ως εξής: δίνεται ένα σύνολο S σημείων στον χώρο M διαστάσεων και ένα σημείο p ανήκει M, να βρεθεί το πλησιέστερο σημείο S στο p. Το M λαμβάνεται ως d-διάστατος Ευκλείδειος χώρος και η απόσταση μετριέται είτε με χρήση της Ευκλείδειας απόστασης, απόσταση Manhattan είτε κάποια άλλη συνάρτηση μέτρησης απόστασης.



4.2 Παράδειγμα εκτέλεσης για 2 διαστάσεις

Έστω το παρακάτω σχήμα. Το παράδειγμα μας περιέχει ξενοδοχεία , ο x άξονας αντιπροσωπεύει την τιμή των ξενοδοχείων και ο y την απόσταση τους από την θάλασσα. Ο αλγόριθμος ξεκινά αναζητώντας τον πλησιέστερο γείτονα σε σχέση με την αρχή των αξόνων, χρησιμοποιούμε την Ευκλείδεια απόσταση στην διαδικασία αυτή. Το σημείο που βρέθηκε αποτελεί κομμάτι της κορυφογραμμής και μπορεί να επιστραφεί στον χρήστη απευθείας. Στο σχήμα 2 φαίνεται ο πρώτος εγγύτερος γείτονας (nx,ny). Επίσης βλέπουμε πως το σύνολο των δεδομένων μπορεί να χωριστεί σε 3 περιοχές.



Η περιοχή 1 περιέχει όλα τα στοιχεία του συνόλου που έχουν τιμή y μικρότερη από αυτήν του εγγύτερου γείτονα. Ονομάσαμε το σημείο (nx,ny) και το ορθογώνιο της περιοχής 1 στην περίπτωση μας οριοθετείτε ως εξής x:(0-∞), y:(0-ny).

Η περιοχή 2 όμοια περιέχει όλα τα στοιχεία του συνόλου που έχουν τιμή x μικρότερη από αυτήν του εγγύτερου γείτονα ενώ δεν υπάρχει περιορισμός στην τιμή y. Οπότε η περιοχή οριοθετείτε ως x:(0-nx), y:(0-∞).

Η περιοχή 3 περιέχει τα υπόλοιπα στοιχεία. Τα στοιχεία της περιοχής 3 κυριαρχούνται από τον πρώτο μας πλησιέστερο γείτονα και δεν χρειάζεται να εξετασθούν περαιτέρω.

Οι περιοχές 1 και 2 εξετάζονται για την εύρεση όλης της κορυφογραμμής , ο αλγόριθμος αναζητεί τον πλησιέστερο γείτονα, από την αρχή των αξόνων, που ανήκει στην περιοχή 1 και έπειτα στην περιοχή 2, ο αλγόριθμος συνεχίζει την ίδια διαδικασία μέχρι να τελειώσουν οι περιοχές και να μην υπάρχουν άλλα σημεία προς εξέταση.

Στο παραπάνω σχήμα φαίνεται η εύρεση και του πλησιέστερου γείτονα (mx,my) στην περιοχή 2. Η περιοχή που έγινε η αναζήτηση ήταν x:(0-nx), y:(0-∞) , οι νέες περιοχές που δημιουργούνται προς αναζήτηση είναι

Περιοχή 4 x:(0-nx), y:(0-my)

Περιοχή 5 x:(0-mx), y:(0-∞)

Η διαδικασία συνεχίζεται έως ότου να μην υπάρχουν περιοχές προς αναζήτηση.

4.3 Κλάσεις που δημιουργήθηκαν/μεταβλήθηκαν

Ακολουθεί μια αναφορά σε κομμάτια κώδικα που γράφτηκαν και μεταλλάχτηκαν για την εκπόνηση της εργασίας, καθώς και η χρησιμότητας τους.

4.3.1 NNRectangle

Για την υλοποίηση αυτής της διαδικασίας υλοποιήθηκε μια κλάση NNRectangle και αποθηκεύονται αντικείμενα της κλάσης αυτής σε μια λίστα, η οποία χρησιμοποιείται στην κύρια επανάληψη του αλγορίθμου.

Η κλάση NNRectangle αποτελεί επέκταση της κλάσης HyperRectangle που είναι υλοποιημένη στην βιβλιοθήκη που χρησιμοποιούμε για το R*-δέντρο.

Προστέθηκε η μεταβλητή τύπου float[] nearest, για την αποθήκευση του πλησιέστερου γείτονα σε κάθε περιοχή και ορισμένες συναρτήσεις για προσπέλαση και μετάλλαξη των μεταβλητών. Με σκοπό την ευκολότερη παραγωγή των επόμενων περιοχών.

Όταν επιστρέφεται ο πλησιέστερος γείτονας, ελέγχεται το σημείο αν ήδη έχει επιστραφεί σε προηγούμενο βήμα του αλγορίθμου, με χρήση ενός πίνακα κατακερματισμού. Αν ήδη υπάρχει στον πίνακα συνεχίζει ο αλγόριθμος με την επόμενη περιοχή προς εξέταση, αν όχι δημιουργούνται οι ανάλογες περιοχές, προστίθενται στην λίστα περιοχών, και τέλος ο γείτονας προστίθεται στην λίστα κατακερματισμού.

4.3.2 HyperRectangle

Αποτελεί την υλοποίηση του ορθογωνίου. Τα όρια του ορθογωνίου αποθηκεύονται στον δισδιάστατο πίνακα points. Στην συγκεκριμένη βιβλιοθήκη κάθε γραμμή στον πίνακα περιέχει 2 τιμές και αντιπροσωπεύει μια μεταβλητή, η τιμή στο πρώτο κελί του πίνακα αποτελεί το μέγιστο όριο του ορθογωνίου και στο δεύτερο κελί το ελάχιστο όριο. Οι περισσότερες συναρτήσεις χρησιμοποιούνται για την αποθήκευση και προσθήκη νέων σημείων στον πίνακα points.

Την σημαντικότερη συνάρτηση αποτελεί η getIntersection που δέχεται σαν παράμετρο ένα ορθογώνιο otherMBR και συγκρίνει το ορθογώνιο της κλάσης με το νέο ορθογώνιο και επιστρέφει αν υπάρχει ένα ορθογώνιο με την τομή των δύο.

4.3.3 RStar-Tree

RStar-Tree, αποτελεί την βασική κλάση του R* δέντρου. Περιέχει συναρτήσεις για εισαγωγή και επεξεργασία του δέντρου και των κόμβων του, συμπεριλαμβανομένου και συναρτήσεις για την διάσπαση των κόμβων. Επιπλέον έχουν ήδη υλοποιηθεί συναρτήσεις για αναζήτηση σημείου (pointSearch), αναζήτηση σημείων σε περιοχή (rangeSearch), αναζήτηση πλησιέστερων k γειτόνων (knnSearch) και τις αντίστοιχες αναδρομικές τους.

4.3.3.1 NNSearch

Η συνάρτηση που δημιουργήσαμε ονομάζεται NNSearch. Δέχεται σαν παραμέτρους ένα σημείο για και ένα αντικείμενο της κλάσης NNRectangle και επιστρέφει έναν πίνακα float που περιέχει τις συντεταγμένες του πλησιέστερου γείτονα.

Γίνονται οι απαραίτητες αρχικοποιήσεις όπως η φόρτωση τις ρίζας του δέντρου και αρχικοποίηση του πίνακα αποτελεσμάτων και του ορθογωνίου αναζήτησης και καλείτε η ήδη υλοποιημένη αναδρομική _rangeSearch όπου επιστρέφει όλα τα σημεία του δέντρου που βρίσκονται στην περιοχή αναζήτησης και τέλος καλείτε η nearestNeighbourSearch όπου βρίσκει και επιστρέφει το κοντινότερο από όλα τα σημεία.

4.3.3.2 _rangeSearch

Αναδρομική συνάρτηση που δέχεται ως παραμέτρους έναν κόμβο του δέντρου και ένα ορθογώνιο αναζήτησης. Ελέγχει τους κόμβους του δέντρου μέχρι να φτάσει στο επίπεδο των φύλων όπου προσθέτει τα σημεία του δέντρου που επικαλύπτονται από το ορθογώνιο αναζήτησης στον πίνακα _rangeSearchResult. Όταν ένας κόμβος δεν είναι στο επίπεδο των φύλων καλείτε αναδρομικά η συνάρτηση για τα παιδιά του κόμβου αυτού.

4.4 Διπλότυπα και αποφυγή τους

Για να αντιμετωπιστούν τα διπλότυπα εγγραφών οι περιοχές προς αναζήτηση πλησιέστερου γείτονα δημιουργούνται προσθέτοντας/αφαιρώντας μια ελάχιστη τιμή στα όρια

region[i][0] = nearest[i] - (float) 0.001;

που εγγυάται την αποφυγή των διπλοτύπων. Επιπλέον όταν μια περιοχή P έχει εξεταστεί και έχουνε δημιουργηθεί οι υποπεριοχές που προκύπτουν από αυτή, εισάγετε σε έναν πίνακα κατακερματισμού μια συμβολοσειρά που αποτελείτε από τα στοιχεία της περιοχής. Όταν μια περιοχή χρίζει εξέτασης, ελέγχεται από το σύστημα αν υπάρχει αναπαράσταση αυτής στον πίνακα κατακερματισμού. Αν υπάρχει αποφεύγεται η επεξεργασία της.

4.5 m-διαστάσεις

Ο αλγόριθμος φτιάχτηκε να λειτουργεί σε m-διαστάσεις. Το R*-δέντρο ήταν ήδη υλοποιημένο με τέτοιο τρόπο, χρειάστηκε μόνο να υλοποιηθεί το κομμάτι παραγωγής των περιοχών. Παρατηρούμε ότι μετά την εύρεση ενός πλησιέστερου γείτονα η νέες περιοχές που παράγονται είναι ίδιες με την περιοχή που εξετάστηκε, εκτός τα όρια του γείτονα σε κάθε διάσταση. Οπότε για την περιοχή D με x:(0-∞), y:(0-∞), z:(0-∞) με πλησιέστερο γείτονα τον n:(nx,ny,nz) η νέες περιοχές που παράγονται είναι:

Περιοχή 1 x:(0-nx), y:(0-∞), z(0-∞)

Περιοχή 2 x:(0-∞), y:(0-ny), z(0-∞)

Περιοχή 3 x:(0-∞), y:(0-∞), z(0-nz)

ΚΕΦΑΛΑΙΟ 5: Πειράματα και Αποδόσεις

ΠΕΙΡΑΜΑΤΑ ΚΑΙ ΑΠΟΔΟΣΕΙΣ

Σε αυτό το κεφάλαιο, μελετάμε την απόδοση του NN αλγορίθμου στον υπολογισμό της κορυφογραμμής. Χρησιμοποιούμε διάφορες συνθετικές βάσεις δεδομένων με διακυμάνσεις στο μέγεθος της βάσης (αριθμός σημείων), την αξία της κατανομής καθώς και τον αριθμό των διαστάσεων των σημείων στην βάση δεδομένων.

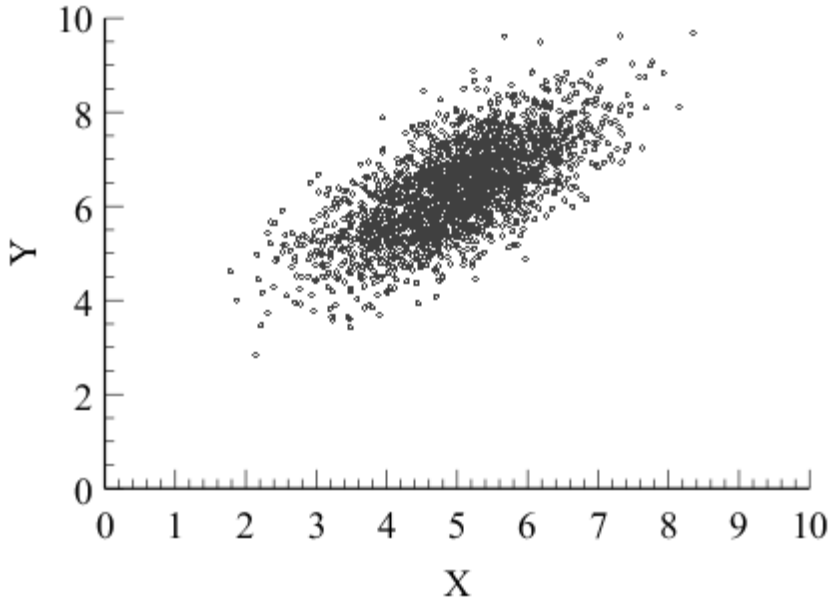
5 Περιβάλλον πειραμάτων

Τα πειράματα διεξήχθησαν σε ένα i5-3570k επεξεργαστή με ταχύτητα πυρήνα 3.4GHz και 8GB κύρια μνήμη ταχύτητας 1600MHz. Το λειτουργικό σύστημα είναι Windows 8. Η γλώσσα προγραμματισμού που εφαρμόστηκαν οι αλγόριθμοι είναι σε Java.

5.1 Σύνολα Δεδομένων

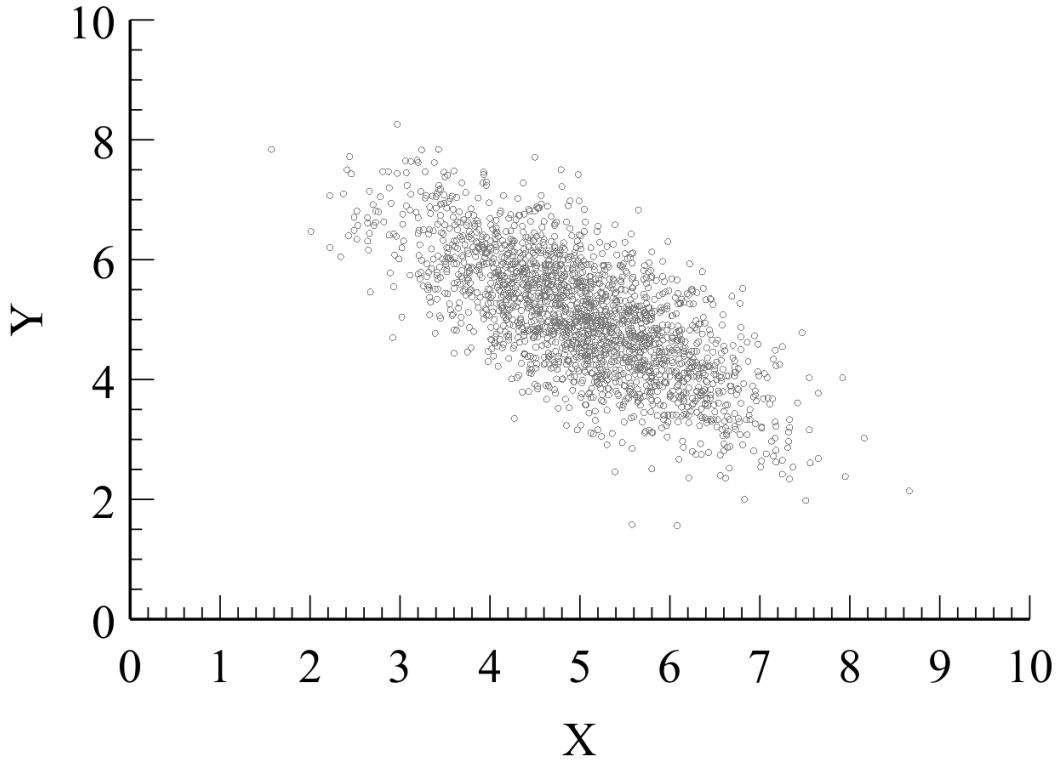
Τα σύνολα δεδομένων δημιουργήθηκαν χρησιμοποιώντας τις παρακάτω κατανομές:

- συσχετιζόμενη κατανομή (correlated)



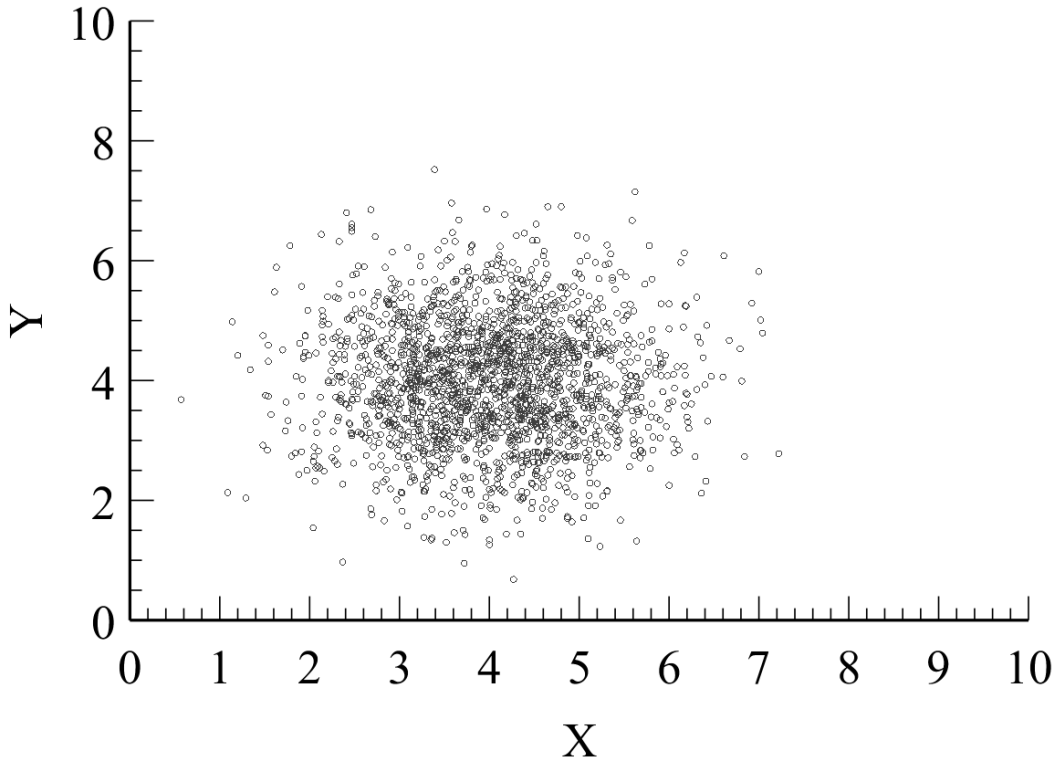
σε μια correlated κατανομή , τα σημεία τα οποία είναι καλά σε μία διάσταση, τείνουν να είναι καλά και σε άλλες διαστάσεις. Ως αποτέλεσμα, σχετικά λίγα σημεία κυριαρχούν πολλά άλλα σημεία, και η κορυφογραμμή του συνόλου αυτού είναι σχετικά μικρή.

- μη συχετιζόμενη κατανομή (anti-correlated)



σε μια μη συσχετιζόμενη κατανομή, τα σημεία τα οποία είναι καλά σε μια διάσταση, είναι κακά σε τουλάχιστον μια άλλη διάσταση. Ως αποτέλεσμα, η κορυφογραμμή ενός anti-correlated συνόλου δεδομένων αποτελείται από αρκετά στοιχεία.

- ανεξάρτητη κατανομή (indepedent)



σε μια indepedent database , τα σημεία παράγονται χρησιμοποιώντας την κανονική κατανομή. Το μέγεθος της κορυφογραμμής ενός ανεξάρτητου συνόλου δεδομένων είναι περίπου ενδιάμεσα από των correlated και anti-correlated.

Για την δημιουργία των παραπάνω, δημιουργήθηκε μια κατανομή με χρήση της βιβλιοθήκης Random και πιο συγκεκριμένα της συνάρτησης nextGaussian() που επιστρέφει μια τιμή από -1 έως 1. Μετατρέψαμε την τιμή αυτή σε θετική όπου χρειάστηκε και κρατήσαμε μέχρι και το δεύτερο δεκαδικό της. Τέλος μορφοποιήσαμε τα δεδομένα με βάση την εξίσωση για να παραχθεί η ζητούμενη κατανομή.

Y= Correlation*X + SQRT(1-Correlation*Correlation)

Ο παράγοντας correlation ήταν 0.7 για την παραγωγή των correlated δεδομένων ενώ -0.7 για τα anti-correlated.

Λεπτομερέστερες περιγραφές για τις τρεις αυτές περιγραφές μπορούν να βρεθούν στο [1].

5.2 Αποτελέσματα

Το πρόγραμμα αρχικά δημιουργεί το R* δέντρο και έπειτα τρέχει τον αλγόριθμο υπολογισμού της κορυφογραμμής. ρατήθηκαν χρόνοι και στατιστικά για τις εισαγωγές των δεδομένων, τον υπολογισμό κάθε σημείου της κορυφογραμμής και λεπτομέρειες κατά τον υπολογισμό αυτών όπως οι χρόνοι προσπέλασης στον δίσκο καθώς το R* δέντρο στην συγκεκριμένη βιβλιοθήκη αποθηκεύεται στο δίσκο. Ιδανικά το πρόγραμμα θα φόρτωνε το επίπεδο των φύλων του R* δέντρου στην μνήμη και οι χρόνοι θα ήταν πολύ μικρότεροι.

- correlated

	Insertion operations: (in milliseconds)	Skyline operations: (in milliseconds)
Total ops	1000000	7
Avg time	38.139	226.857
5th percentile	22	30
95th percentile	56	1247

Χαρακτηριστικά πειράματος:

- 1.000.000 στοιχεία κατανομής
- 7 στοιχεία αποτελούν την κορυφογραμμή

Neighbor	Neighbor Search Time (in milliseconds)	Disk Call Time (in milliseconds)
0.4 – 2.08	1247	1228
2.27 – 1.45	39	37
0.37 – 2.19	72	71
1.14 – 1.96	52	51
0.34 – 2.26	45	44
0.07 – 2.57	30	29

Οι σταθερές για τα ελάχιστα και μέγιστα παιδιά του κάθε κόμβου στο R*-δέντρο είναι 15 και 50 αντίστοιχα. Ενώ ο μέσος χρόνος υπολογισμού των σημείων της κορυφογραμμής είναι μόλις 0.2 δευτερόλεπτα.

- anti-correlated

	Insertion operations: (in milliseconds)	Skyline operations: (in milliseconds)
Total ops	1000000	31
Avg time	36.770	7902.483
5th percentile	20	82
95th percentile	61	20278

Χαρακτηριστικά πειράματος:

- 1.000.000 στοιχεία κατανομής
- 31 στοιχεία αποτελούν την κορυφογραμμή (10 παρουσιάζονται παρακάτω)

Neighbor	Neighbor Search Time (in milliseconds)	Disk Call Time (in milliseconds)
2.9 - 3.76	20278	20147
3.81 - 2.96	15294	15204
2.85 - 3.95	12087	11998
3.73 - 3.13	7783	7761
4.1 - 2.57	11249	11188
2.19 - 4.39	11747	11668

3.93 - 2.92	7532	7506
4.6 - 1.77	7999	7961
1.94 - 4.81	5868	5842
2.74 - 4.15	8040	8006
4.08 - 2.84	7451	7425
4.4 - 2.48	6113	6095
5.45 - 1.72	2103	2092
1.51 - 5.08	3809	3788

Οι πολλές κλήσεις στον δίσκο καθυστερούν αρκετά τον αλγόριθμο στα πρώτα σημεία που υπολογίζονται, καθώς βρίσκονται στο κέντρο της κατανομής, δηλαδή στην περιοχή με μεγαλύτερη πυκνότητα. Τα τελευταία 10 σημεία της κορυφογραμμής έχουν χρόνο υπολογισμού λιγότερο από ένα δευτερόλεπτο.

- Independent

	Insertion operations: (in milliseconds)	Skyline operations: (in milliseconds)
Total ops	1000000	7
Avg time	23	5405
5th percentile	13	37
95th percentile	29	22673

Neighbor	Neighbor Search Time (in milliseconds)	Disk Call Time (in milliseconds)
0.76 - 2.27	109	80
1.73 - 0.77	22673	22634
0.12 - 2.88	37	36
2.48 - 0.1	1597	1592
2.41 - 0.67	629	628
2.62 - 0.01	85	84

Παρόμοια συμπεριφορά παρατηρούμε και στην ανεξάρτητη κατανομή. Με ένα σημείο της κορυφογραμμής να χρειάζεται πάνω απο 22 δευτερόλεπτα, χρόνος αρκετά μεγαλύτερος συγκριτικά με τα υπόλοιπα.

- 10M anti-correlated

Ο υπολογισμός της κορυφογραμμής δεν αποδείχθηκε εφαρμόσιμος από άποψη χρόνου καθώς οι κλήσεις στον δίσκο μονάχα για τον υπολογισμό του πρώτου πλησιέστερου γείτονα διέρκησαν περισσότερο από εννενήντα λεπτά. Για την ακρίβεια ο πρώτος πλησιέστερος γείτονας υπολογίστηκε σε 5999277 milisecond και οι κλήσεις στον δίσκο διέρκησαν 5994388 milisecond. Ενώ λόγω της πυκνότητας σε ορισμένες περιοχές, υπήρχαν αρκετές διακυμάνσεις σε διαδοχικούς γείτονες. Ακολουθούν χρόνοι υπολογισμού τεσσάρων διαδοχικών γειτόνων.

Neighbor	Neighbor Search Time (in milliseconds)	Disk Call Time (in milliseconds)
1.91 - 5.11	599686	599491
5.7 - 1.58	8082	8055
1.75 - 5.17	350931	350836
6.02 - 1.48	4262	4249

5.3 Συμπέρασμα

Ο αλγόριθμος υπολογισμού της κορυφογραμμής λειτουργεί ορθά για m-διαστάσεις. Η ορθότητα των επιστρεφόμενων στοιχείων ως κομμάτια της κορυφογραμμής ελέγχθηκε με χρήση της μεθόδου ωμής βίας μετά την ολοκλήρωση κάθε πειράματος. Ενώ από τις χρονικές αποδόσεις των πειραμάτων καταλήγουμε στα εξής συμπεράσματα:

Γίνονται πολλές κλήσεις στον δίσκο και δεν κρατάται τίποτα στην μνήμη κατά την λειτουργία των αλγορίθμων αναζήτησης στο R-δέντρο, το οποίο προκαλεί αρκετές διακυμάνσεις σε θέματα απόδοσης από έναν πλησιέστερο γείτονα σε έναν άλλον ανάλογα με την πυκνότητα των σημείων. Σε ένα R-δέντρο του οποίου το επίπεδο των φύλλων είναι φορτωμένο στην μνήμη, ή αποφεύγονται οι άσκοπες κλήσεις ο μέσος χρόνος υπολογισμού των πλησιέστερων γειτόνων εμφανώς μειωμένος.

Για την σωστή εφαρμογή της κορυφογραμμής σε ένα online, από άποψη ροής δεδομένων, σύστημα το R*-δέντρο θα πρέπει να είναι ήδη υλοποιημένο και να φορτώνεται το επίπεδο των φύλλων στην μνήμη κατά την εκκίνηση του συστήματος. Ενώ η πλήρης υλοποίηση του δέντρου, όσον αφορά τις διαστάσεις, δίνει την δυνατότητα μεταβολής του αριθμού των διαστάσεων κατά την διαδικασία της αναζήτησης, απλά αλλάζοντας την περιοχή αναζήτησης.

ΠΑΡΑΡΤΗΜΑ Ι: Αναφορές

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Donald Kossmann, Frank Ramsak and Steffen Rost, Shooting Stars in the Sky: An Online Algorithm for Skyline Queries.

[2] Yannis Manolopoulos, Aleksandros Nanopoulos, Apostolos Papadopoulos, Yannis Theodoridis, R-Trees: Theory and Applications.

[3] Apostolos Papadopoulos, Yannis Manolopoulos, Performance of Nearest Neighbor Queries in R-trees

WEB SITES

[1] Rstar-Tree library by Lokesh Jangid https://github.com/lokeshj/RStar-Tree

ΠΑΡΑΡΤΗΜΑ ΙΙ: Κώδικας

Παραθέτονται μόνο τα κομμάτια που αλλάχθηκαν/δημιουργήθηκαν, το μεγαλύτερο μέρος που αποτελείται από την βιβλιοθήκη Rstar-Tree μπορεί να βρεθεί στο github, link παραπάνω.

Αλγόριθμος υπολογισμού κορυφογραμμής

```
protected void skyline() {


        long start, end;
        float[] region;
        float[] n;
        int next = 0;
        float[] center = new float[this.dimension];
                float[][] infPoints = new float[this.dimension][2];
        HashMap< String, float[]> map = new HashMap<>();
        String times="";


                ArrayList<NNRectangle> T = new ArrayList<NNRectangle>();
        NNRectangle m;


        for (int c = 0; c < this.dimension; c++) {
            center[c] = 0;
        }


        for (int k = 0; k < dimension; k++) {
            infPoints[k][0] = 1000000;
            infPoints[k][1] = 0;
        }
        NNRectangle inf = new NNRectangle(this.dimension);
        inf.setPoints(infPoints);
        T.add(inf);




        while (T.size() >= next + 1) {
            m = T.get(next++);


            start = System.currentTimeMillis();
            if(next!=1)
                n = tree.NNSearch(new SpatialPoint(center), m);
            else
                n = tree.knnSearch(new SpatialPoint(center), 1).get(0).getCords();


            if (n != null) {
                            String strKey="";
                for(int l=0;l<n.length;l++)
                    strKey= strKey + Float.toString(n[l])+ " ";


                if (!map.containsKey(strKey)) {
                    end = System.currentTimeMillis();


                    if(next!=1)
                    {
                        times+="\nNeighbour            : "+ n[0]+ " " + n[1];
                        times+="\nNeighbour Search Time: "+(end-start);
                               times+="\nDisk calls sum         : "+(tree.getSumTime()
+tree.getSumTime2());
                        times+="\n";
                    }


                    skylineTime.add(end - start);


                                        for (int i = 0; i < dimension; i++) {
```

```
                        NNRectangle nRegion = new NNRectangle(dimension);
                        float[][] nRegionPoints = new float[dimension][2];
                        for (int j = 0; j < dimension; j++) {
                            nRegionPoints[j][0] = m.getPoints()[j][0];
                            nRegionPoints[j][1] = m.getPoints()[j][1];




                        }
                        nRegionPoints[i][0] = n[i] - (float) 0.001;









                        nRegion.setPoints(nRegionPoints);
                        nRegion.setNearest(n);
                        map.put(strKey, n);
                        T.add(nRegion);


                    }
                }
            }
        }
        outcome("nearest neighbours",map);
        writeToFile("times", times);
}
```

ComputeSkyline

```
package ptyxiaki.rstar;


import ptyxiaki.rstar.rstar.RStarTree;
import ptyxiaki.rstar.rstar.spatial.SpatialPoint;
import ptyxiaki.rstar.util.Trace;
import ptyxiaki.rstar.util.Utils;


import java.io.*;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Random;




public class ComputeSkyline {


    private RStarTree tree;
    private int dimension;
    private String inputFile;
    private String resultFile;
    private List<Long> insertRunTime;
    private List<Long>  skylineTime;
    private Trace logger;


    public static void main(String[] args) {
        ComputeSkyline controller = new ComputeSkyline(args);


        System.out.println("Reading input file ...");
        controller.processInput();
        System.out.println("Finished Processing file ...");
                                controller.writeRuntimeToFile(controller.insertRunTime,
"Insertion_runtime.txt");
        controller.skyline();
        controller.writeRuntimeToFile(controller.skylineTime, "Skyline_runtime.txt");
        controller.printResults();


    }


    public ComputeSkyline (String[] args) {
        if(args.length >= 2){
        this.inputFile = args[0];
        this.dimension = Integer.parseInt(args[1]);
                if (args.length >= 3)
                                this.resultFile = args[2];
                        else
                                        this.resultFile = this.getClass().getSimpleName()+
"_Results.txt";


                } else {
                        this.printUsage();
                        System.exit(1);
                }
        tree = new RStarTree(dimension);
        this.insertRunTime = new ArrayList<Long>();
        this.skylineTime = new ArrayList<Long>();
        logger = Trace.getLogger(this.getClass().getSimpleName());



    }
```

```
    protected void processInput() {
        float  oid, k;
        double range;
        float[] point;
        long start, end;
        int lineNum = 0;


        try {
            BufferedReader input = new BufferedReader(new FileReader(this.inputFile));
            String line;
            String[] lineSplit;


            while ((line = input.readLine()) != null) {
                lineNum++;
                lineSplit = line.split(" ");
                try {
                            if (lineSplit.length != (this.dimension + 1)) {
                                throw new AssertionError();
                            }


                            oid = Float.parseFloat(lineSplit[0]);
                            point = extractPoint(lineSplit, 1);


                            start = System.currentTimeMillis();
                            tree.insert(new SpatialPoint(point, oid));
                            end = System.currentTimeMillis();


                                                                    insertRunTime.add((end -
start));


                            break;


                        } catch (Exception e) {
                                logger.traceError("Exception while processing line " +
lineNum
                                                  + ". Skipped Insertion. message: " +
e.getMessage());
                            break;
                        } catch (AssertionError error) {
                            logger.traceError("Error while processing line " + lineNum
                                              + ".Skipped Insertion. message: " +
error.getMessage());
                            break;
                        }
            }






            input.close();
            tree.save();
        } catch (Exception e) {
              logger.traceError("Error while reading input file. Line " + lineNum + "
Skipped\nError Details:");
        }


    }
```

```
          private   float[]   extractPoint(String[]   points,   int   startPos)   throws
NumberFormatException {
        float[] tmp = new float[this.dimension];
        for (int i = startPos, lineSplitLength = points.length;
                ((i < lineSplitLength) && (i < (startPos + this.dimension))); i++) {
            tmp[i - startPos] = Float.parseFloat(points[i]);
        }
        return tmp;
    }




    protected void printResults() {
        logger.trace("\nPerforming Run Time calculations..");


        List<Long> combined = new ArrayList<Long>();
        combined.addAll(insertRunTime);


        String result = "\n" + this.getClass().getSimpleName() + " --RESULTS--";


                  String  temp  =  "\n\nInsertion  operations:(in  milliseconds)  "  +
generateRuntimeReport(insertRunTime);
        logger.trace(temp);
        result += temp;
                        temp   =   "\n\nSkyline   operations:(in   milliseconds)   "   +
generateRuntimeReport(skylineTime);
        logger.trace(temp);
        result += temp;


        writeResultToFile(result);
    }


    protected String generateRuntimeReport(List<Long> runtime) {
        StringBuilder result = new StringBuilder();
        int size = runtime.size();


        if (size > 0) {
            Collections.sort(runtime);
            try {
                Long percent5th = runtime.get((int) (0.05 * size));
                Long percent95th = runtime.get((int) (0.95 * size));
                long sum = 0;
                for (Long aRuntime : runtime) {
                    sum += aRuntime;
                }
                double avg = sum / (double) size;


                result.append("\nTotal ops = ").append(size);
                result.append("\nAvg time: ").append(avg);
                result.append("\n5th percentile: ").append(percent5th);
                result.append("\n95th percentile: ").append(percent95th);


            } catch (Exception e) {
                logger.traceError("Exception while generating runtime results");
                e.printStackTrace();
            }
        }


        return result.toString();
    }
```

```
    protected void writeResultToFile(String result) {
        try {
            File outFile = new File(this.resultFile);
            if (outFile.exists()) {
                outFile.delete();
            }
            BufferedWriter outBW = new BufferedWriter(new FileWriter(outFile));
            try {
                logger.trace("\nWriting results to file .. ");
                outBW.write(result);
            } finally {
                outBW.close();
                logger.trace("done");
            }
        } catch (IOException e) {
            logger.traceError("IOException while writing results to " + resultFile);
        }
    }


    protected void writeRuntimeToFile(List<Long> runtime, String file) {
        try {
            File f = new File(file);
            if (f.exists()) {
                f.delete();
            }


            BufferedWriter bf = new BufferedWriter(new FileWriter(f));
            for (long i : runtime) {
                bf.write("" + i + "\n");
            }
            bf.close();
        } catch (IOException e) {
            logger.traceError("IOException while writing runtimes to file.");
//          e.printStackTrace();
        }
    }


    protected void printUsage() {
        System.err.println("Usage: " + this.getClass().getSimpleName()
                + " <path to input file> <dimension of points> [output file].\noutput
file is optional.\n");
    }


    protected void skyline() {


        long start, end;
        float[] region;
        float[] n;
        int next = 0;
        float[] center = new float[this.dimension];
                float[][] infPoints = new float[this.dimension][2];
        HashMap< String, float[]> map = new HashMap<>();
        String times="";


                ArrayList<NNRectangle> T = new ArrayList<NNRectangle>();
        NNRectangle m;


        for (int c = 0; c < this.dimension; c++) {
            center[c] = 0;
        }


        for (int k = 0; k < dimension; k++) {
            infPoints[k][0] = 1000000;
            infPoints[k][1] = 0;
        }
        NNRectangle inf = new NNRectangle(this.dimension);
```

```
        inf.setPoints(infPoints);
        T.add(inf);



        while (T.size() >= next + 1) {
            m = T.get(next++);


            start = System.currentTimeMillis();
            if(next!=1)
                n = tree.NNSearch(new SpatialPoint(center), m);
            else
                n = tree.knnSearch(new SpatialPoint(center), 1).get(0).getCords();


            if (n != null) {
                            String strKey="";
                for(int l=0;l<n.length;l++)
                    strKey= strKey + Float.toString(n[l])+ " ";


                if (!map.containsKey(strKey)) {
                    end = System.currentTimeMillis();


                    if(next!=1)
                    {
                        times+="\nNeighbour            : "+ n[0]+ " " + n[1];
                        times+="\nNeighbour Search Time: "+(end-start);
                               times+="\nDisk calls sum         : "+(tree.getSumTime()
+tree.getSumTime2());
                        times+="\n";
                    }


                    skylineTime.add(end - start);


                                        for (int i = 0; i < dimension; i++) {


                    NNRectangle nRegion = new NNRectangle(dimension);
                    float[][] nRegionPoints = new float[dimension][2];
                    for (int j = 0; j < dimension; j++) {
                        nRegionPoints[j][0] = m.getPoints()[j][0];
                        nRegionPoints[j][1] = m.getPoints()[j][1];




                    }
                    nRegionPoints[i][0] = n[i] - (float) 0.001;




                    nRegion.setPoints(nRegionPoints);
                    nRegion.setNearest(n);
                    map.put(strKey, n);
                    T.add(nRegion);


                    }
                }
            }
        }
        outcome("nearest neighbours",map);
        writeToFile("times", times);
    }
```

```
    public void outcome(String filename ,HashMap<String, float[]> map) {
        try {
            Random r = new Random();
            FileWriter fstream = new FileWriter(" "+filename+" "+this.inputFile);
            BufferedWriter out = new BufferedWriter(fstream);
            for (float[] f : map.values()) {
                for (int m = 0; m < f.length; m++) {
                    out.write(f[m] + " ");
                }
                out.write("\n");
            }
            out.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }


    public void writeToFile(String filename ,String text) {
        try {
            Random r = new Random();
            FileWriter fstream = new FileWriter(" "+filename+" "+this.inputFile);
            BufferedWriter out = new BufferedWriter(fstream);
                    out.write(text);


            out.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

NNRectangle

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package ptyxiaki.rstar;


import java.util.List;
import ptyxiaki.rstar.rstar.dto.MbrDTO;
import ptyxiaki.rstar.rstar.spatial.HyperRectangle;
import static ptyxiaki.rstar.rstar.spatial.HyperRectangle.MAX_CORD;
import static ptyxiaki.rstar.rstar.spatial.HyperRectangle.MIN_CORD;
import ptyxiaki.rstar.rstar.spatial.SpatialPoint;
import ptyxiaki.rstar.util.Constants;


/**
 *
 * @author Super User
 */
public class NNRectangle{
        private int _dimension;
    /**
     * points is a 2D double array containing
     * the max and min values for each dimension
     * in the rectangle.
     */
    private float[][] points;
    private float[] nearest;
    public static final int MAX_CORD = 0;
    public static final int MIN_CORD = 1;


     public NNRectangle(int dimension) {
        this._dimension = dimension;
        points = new float[dimension][2];
    }


    public NNRectangle(float[] cords) {
        this._dimension = cords.length;
        points = new float[_dimension][2];
        for (int i = 0; i < _dimension; i++) {
            points[i][MAX_CORD] = cords[i];
            points[i][MIN_CORD] = cords[i];
        }
    }


        public void setPoints(float[][] points) {
        this.points = points;
    }




    public NNRectangle(int dimension, SpatialPoint[] points) {
        this._dimension = dimension;
        this.points = new float[dimension][2];


        update(points);
    }


        private void update(SpatialPoint[] newPoints) {
        for (SpatialPoint newPoint : newPoints) {
            float[] cord = newPoint.getCords();
            assert cord.length == _dimension;
            for (int i = 0; i < cord.length; i++) {
```

```
                if (points[i][MAX_CORD] == 0 || points[i][MAX_CORD] < cord[i]) {
                    points[i][MAX_CORD] = cord[i];
                }
                if (points[i][MIN_CORD] == 0 || points[i][MIN_CORD] > cord[i]) {
                    points[i][MIN_CORD] = cord[i];
                }
            }
        }
    }



  /**
     * finds the intersecting region of this MBR with otherMBR
     * @param otherMBR the mbr with which this mbr's intersection
     *                 is to be calculated
     * @return the intersecting region, null if not intersecting
     */
    public HyperRectangle getIntersection(HyperRectangle otherMBR) {
        float[][] interPoints = new float[_dimension][2];
        float[][] newPoints = otherMBR.getPoints();
        assert newPoints.length == _dimension;


        boolean intersectExists = true;
        for (int i = 0; i < _dimension; i++) {
            if ((points[i][MAX_CORD] < newPoints[i][MIN_CORD]) || (points[i][MIN_CORD]
> newPoints[i][MAX_CORD])) {
                intersectExists = false;
                break;
            }
                interPoints[i][MAX_CORD] = Math.min(newPoints[i][MAX_CORD], points[i]
[MAX_CORD]);
                interPoints[i][MIN_CORD] = Math.max(newPoints[i][MIN_CORD], points[i]
[MIN_CORD]);
        }


        if (!intersectExists) {
            return null;
        }
        HyperRectangle intersect = new HyperRectangle(_dimension);
        intersect.setPoints(interPoints);
        return intersect;
    }


    public int getDimension()
    {
        return this._dimension;
    }


    public float[] getNearest()
    {
        return this.nearest;
    }


    public void setNearest(float[] nearest)
    {
           this.nearest=nearest;
    }


    public float[][] getPoints()
    {
        return this.points;
    }



    public void printPoints()
```

```
    {
        for(int i=0;i<this._dimension;i++)
        {
            for(int j=0;j<2;j++)
                System.out.print(i+" "+j+" "+points[i][j] +" ");
            System.out.println(" ");


        }
        System.out.print("nearest: ");
        for(int z=0;z<nearest.length;z++)
              System.out.print(nearest[z]+" ");
        System.out.println(" ");
        System.out.println(" ");



    }
}
```

Rstar-Tree

```
private void _rangeSearch(RStarNode start, HyperRectangle searchRegion) {
        HyperRectangle intersection = start.getMBR().getIntersection(searchRegion);


        if (intersection != null) {
            if (start.isLeaf()) {
                for (Long pointer : start.childPointers) {

                    long s = System.currentTimeMillis();
                    PointDTO dto = storage.loadPoint(pointer);
                    long e = System.currentTimeMillis();
                    this.diskCallsTime2.add(e-s);




                    SpatialPoint spoint = new SpatialPoint(dto);
                    HyperRectangle pointMbr = new HyperRectangle(dto.coords);


                    if(pointMbr.getIntersection(searchRegion) != null)
                    {
                        _rangeSearchResult.add(spoint);
                    }
                }
            }
            else {
                for (Long pointer : start.childPointers) {
                    try {

                        long s = System.currentTimeMillis();
                    RStarNode childNode = storage.loadNode(pointer);    //recurse down
                        long e = System.currentTimeMillis();
                        this.diskCallsTime.add(e-s);
                        _rangeSearch(childNode, searchRegion);


                    } catch (FileNotFoundException e) {
                        System.err.println("Exception while loading node from disk");
                    }
                }
            }
        }


    }


     /**
         *    searches for the nearest neighbour in a region of a center point
         *    @param center SpatialPoint
           *     @param m NNRectanglge a rectangle of the region that is going to be
examined
         *    @return float array of the points of the nearest neighbours.
         */
    public float[] NNSearch(SpatialPoint center, NNRectangle m)
    {
        this.diskCallsTime = new ArrayList<Long>();
        this.diskCallsTime.clear();
         this.diskCallsTime2 = new ArrayList<Long>();
        this.diskCallsTime2.clear();
        nearest= new RStarNearest(dimension);
        _rangeSearchResult = new ArrayList<SpatialPoint>();


                HyperRectangle searchRegion = new  HyperRectangle(m.getDimension());
        searchRegion.setPoints(m.getPoints());
        loadRoot();
```

```
        _rangeSearch(root,searchRegion);


        return nearestNeighbourSearch(center,_rangeSearchResult);
    }




    /**
     *    searches for the nearest neighbour of a center point
     *    @param center SpatialPoint
     *    @param k number of nearest neighbours required
     *    @return float array of the points of the nearest neighbours.
     */
     public float[] nearestNeighbourSearch(SpatialPoint center,ArrayList<SpatialPoint>
points )
    {
        float dist=Float.MAX_VALUE;
        int nearest=0;
        boolean a = false;
        for(int i=0;i<points.size();i++)
        {
              if(center.distance(points.get(i))<dist)
              {
                    dist=center.distance(points.get(i));
                    nearest=i;
                    a=true;
              }
        }
        if(a==true)
           return points.get(nearest).getCords();
        return null;
  }
```

Utilities

```
public void modifyDataSet(String filename,double correlation) {
        try {


            FileWriter fstream = new FileWriter(filename);
            BufferedWriter out = new BufferedWriter(fstream);
            double x;
            double y;
            for (int i = 0; i < 1000000; i++) {


                x= data[i][0];
                y= data[i][1];
               if(correlation!=0)
y=Math.round((correlation*x+ Math.sqrt(1- correlation*correlation)*y)*100)/100.0;
                if(y<0)
                    y=-1*y;


                data[i][0]=x;
                data[i][1]=y;
                 out.write(i+" "+x + " " + y + "\n");
            }
            out.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
  }
```