Πανεπιστήμιο Θεσσαλίας

Τμήμα Μηχανικών Η/Υ , Τηλεπικοινωνιών & Δικτύων

Τίτλος Διπλωματικής Εργασίας:

Μελέτη της Δομής Ευρετηρίου RR*

Μπολτσή Αγγελική

ΑΕΜ 580

agboltsi@uth.gr

Επιβλέπων Καθηγητής : Μποζάνης Παναγιώτης

Βόλος 2011

Περιεχόμενα

ΠΕΡΙΛΗΨΗ

Ο RR*Tree αλγόριθμος αποτελεί μια βελτιωμένη μορφή του R* Tree που χρησιμοποιείται για αναπαράσταση και την ανάπτυξη στοιχείων σε Συστήματα Διαχείρισης Βάσεων Δεδομένων(DBMS). Στην περιγραφή του αλγορίθμου που θα ακολουθήσει στα παρακάτω κεφάλαια, εκτός από την περιγραφή βασικών δομών όπως είναι η εισαγωγή ενός στοιχείου στο δέντρο(insertion), η διαγραφή (deletion), θα δούμε μια επανασχεδιασμένη δομή για την επιλογή του κατάλληλου υποδέντρου (subtree choice) για την εισαγωγή του στοιχείου και για τη διαίρεση του δέντρου σε υποδένω σε περίπτωση υπερχείλισης (split) ενός κόμβου. Αρχικά λοιπόν, θα γίνει μια αναφορά στους αλγορίθμους που αποτελούν την βάση της δημιουργίας του αναθεωρημένου αλγορίθμου R*Tree ,όπως είναι ο R-tree και ο R*Tree, και στη συνέχεια θα εισάγουμε τις επανασχεδιασμένες δομές και θα δούμε που διαφέρουν, με αυτές των προαναφερθέντων αλγορίθμων. Τέλος, θα παρουσιαστούν τα αποτελέσματα των πειραμάτων που έγιναν μετά από αντιπροσωπευτικά τρεξίματα του αλγορίθμου.

1.ΕΙΣΑΓΩΓΗ

Τα R-trees είναι από τις πιο δημοφιλείς μεθόδους για την προσπέλαση και ευρετηριοποίηση διδιάστατων χωρικών δεδομένων αλλά και πολυδιάστατων σημειακών δεδομένων σε μια βάση δεδομένων. Χάρη στην εννοιολογική τους απλότητα τα δέντρα αυτά μπορούν να υλοποιούνται πάνω σε εμπορικά συστήματα βάσεων δεδομένων όπως είναι IBM Informix, Oracle, Sun MySQL , καθώς ακόμη και σε open source συστήματα όπως στο PostgreSQL. Επίσης, R-tree εφαρμογές είναι διαθέσιμες και σε γεωγραφικά πληροφοριακά συστήματα όπως είναι το MapInfo.

Τις τελευταίες δύο δεκαετίες έχουν προταθεί πολλές διαφοροποιήσεις πάνω στα R-trees , που έχουν να κάνουν κυρίως με τον αλγόριθμο διαίρεσης των κόμβων σε περίπτωση υπερχείλισης (split algorithm) και τον αλγόριθμο εισαγωγής στοιχείων σε έναν κόμβο δηλαδή το πως θα προσπελαύνονται οι διάφοροι κόμβοι και ποια υποδέντρα θα επιλέγουμε μέχρι να τοποθετήσουμε σε κάποιον κόμβο το δεδομένο-στοιχείο (ChooseSubtree algorithm).

Σε κεφάλαιο που ακολουθεί θα δούμε αναλυτικά τους αλγορίθμους που απαρτίζουν μία R-tree δομή.

Συνεχίζουμε με τα R*-trees , τα οποία , έχει αποδειχθεί , ότι αποτελούν την πιο αποτελεσματική δομή για την παρουσίαση και υποβολή ερωτημάτων. Η συγκεκριμένη δομή αποτέλεσε οδηγό για διάφορες εφαρμογές με R-trees και έχει επηρεάσει την σχεδίαση και πολλών μεθόδων προσπέλασης , ειδικά αυτών που έχουν να κάνουν με την ευρετηριοποίηση πολυδιάστατων σημειακών δεδομένων.

Βέβαια, υπάρχουν κάποιες ελλείψεις στις συναρτήσεις που απαρτίζουν μία R*-tree δομή. Αρχικά, όσον αφορά την επανεισαγωγή ενός στοιχείου στο δέντρο, η οποία είναι η πιο σημαντική συνεισφορά του αλγορίθμου στην παρουσίαση των ερωτημάτων (επερωτήσεων) , αποτελεί ένα «δυσκίνητο» σχέδιο σε ένα σύστημα διαχείρισης βάσεων δεδομένων, και αυτό γιατί απαιτεί συγχρονισμό για τον έλεγχο των δεδομένων και επιπλέον στην προσπάθεια βελτιστοποίησης της εισαγωγής των δεδομένων γίνεται επικάλυψη των προηγούμενων δεδομένων που ήδη υπάρχουν στο δέντρο.

Επίσης, το CPUκόστος πρέπει πάντα να κυμαίνεται μέσα σε αποδεκτά όρια, αυτή η παράμετρος όμως είναι δύσκολο να παρακολουθείται εφόσον υπάρχει πληθώρα από σύνολα δεδομένων τα οποία είναι και πολυδιάστατα.

Μία ακόμη έλλειψη των R*-trees , είναι η παράμετρος που ορίζει πόσα δεδομένα μπορούν να εισέλθουν σε έναν κόμβο χωρίς να γίνει υπερχείλιση και η οποία δεν επηρεάζεται από τον αριθμό της σελίδας. Αυτό το ελάττωμα δεν παρατηρείται μόνο στην συγκεκριμένη δομή αλλά σε όλη την οικογένεια των R-tree αλγορίθμων.

Τέλος , ένα ακόμη πρόβλημα δημιουργείται από το γεγονός ότι τα R-trees δεν είναι αποτελεσματικά σχεδιασμένα να υποστηρίζουν πολυδιάστατα δεδομένα. Οι στρατηγικές που χρησιμοποιούν πάνω στην διαίρεση των κόμβων είναι βασισμένες στον όγκο των τετραγώνων και είναι αναποτελεσματικές.

Στα επόμενα κεφάλαια θα παρουσιαστεί μια εκσυγχρονισμένη μέθοδος των αλγορίθμων του R*-tree , που βέβαια ακολουθεί κάποιες βασικές αρχές αυτού , όπως για παράδειγμα , τη λογική της διαίρεσης των κόμβων , την βελτιστοποίηση κατά την εισαγωγή και την προσπέλαση του δέντρου. Αλλά εγκαταλείπεται η έννοια της εισαγωγής του ίδιου στοιχείου μία ή παραπάνω φορές όταν αυτό υπάρχει ήδη στην δομή, το οποίο, όπως τονίσαμε και παραπάνω, δημιουργούσε πρόβλημα στο σύστημα διαχείρισης των βάσεων δεδομένων.

Έτσι οι αλγόριθμοι όπως ο ChooseSubTree επανασχεδιάζονται ώστε οι όχι σχετικές καταχωρήσεις που δεν μπορούν να φιλοξενήσουν το καινούριο αντικείμενο να κλαδεύονται. Αυτό έχει ως αποτέλεσμα την βελτιστοποίηση σε όλα τα επίπεδα του ευρετηρίου και συνεπώς μείωση του χρόνου αναζήτησης στην CPU.

Επίσης ο αλγόριθμος Split δεν είναι περιορισμένος στο να βελτιστοποιεί κάποια αποδεκτά σε όλους κριτήρια ,όπως όγκος,περίμετρος,επικάλυψη, αλλά λαμβάνει υπόψη την ισορροπία. Δηλαδή, προτιμάμε στη διαίρεση τα μισά από τα δεδομένα του παλιού κόμβου να μεταφέρονται στον καινούριο. Παρόλα αυτά , ο παραπάνω τρόπος διαίρεσης δεν είναι πρώτος στις επιλογές μας όταν τα δεδομένα δεν καταχωρούνται τυχαία. Κρατώντας μία στοίβα από τροποποιήσεις στα ελάχιστα δεσμευμένα τετράγωνα (minimum bounding box , MBB) του κάθε κόμβου είμαστε σε θέση να εκτιμήσουμε την μελλοντική θέση των δεδομένων, έτσι οδηγούμαστε σε ένα καινούριο αλγόριθμο για την διαίρεση των κόμβων, τον οποίο θα αναλύσουμε παρακάτω.

Και οι δύο αλγόριθμοι που αναφέρθηκαν ChooseSubTree και Split όταν δεν είναι αποτελεσματικοί στην βελτιστοποίηση που κάνουν με βάση τον όγκο των τετραγώνων , βελτιστοποιούν με βάση την περίμετρο αυτών.

Γενικά ο αναθεωρημένος R*-tree (RR*-tree) αλγόριθμος θα πρέπει να ικανοποιεί την εξής απαίτηση: πρέπει να είναι πιο αποδοτικός από τον R*-tree για μεγάλο όγκο δεδομένων και ανεξαρτήτως των διαστάσεων αυτών και επίσης η επίδοση της CPU να είναι εντός αποδεκτών ορίων όταν τα δεδομένα οργανώνονται στην κύρια μνήμη.

Θα δούμε την επίδοση του RR*-tree εισάγοντας διάφορα σύνολα δεδομένων και με διάφορες διαστάσεις.

2.ΠΡΟΓΕΝΕΣΤΕΡΟΙ ΑΛΓΟΡΙΘΜΟΙ

Το R-tree είναι ένα ισορροπημένο ,από πλευρά ύψους, δέντρο, που σχεδιάστηκε για να οργανώνει σύνολα από διδιάστατα τετράγωνα δυναμικά. Ο πρωταρχικός στόχος ενός τέτοιου δέντρου είναι να εκχωρήσει τα τετράγωνα στα φύλλα ανάλογα την χωρική απόσταση μεταξύ τους. Οι εσωτερικοί κόμβοι περιέχουν εκχωρήσεις που σε ενημερώνουν για τα υποδέντρα Τ που βρίσκονται στο επόμενο επίπεδο , καθώς και για το MBB που σχηματίζουν τα τετράγωνα που περιέχονται στο κάθε Τ. Τα φύλλα , περιέχουν τα δεδομένα-στοιχεία και τα MBBs που περιέχονται τα δεδομένα-στοιχεία Ένας κόμβος στο R-tree αντιστοιχεί σε μια σελίδα με καθορισμένο μέγεθος . Ορίζουμε το Μ να αντιπροσωπεύει την χωρητικότητα ενός κόμβου και m την ελάχιστη κατοχή ενός κόμβου , όπου m <= [M/2].

Μία **εισαγωγή (insertion)** ενός νέου τετραγώνου στο R-tree ξεκινάει πάντα από την ρίζα. Με την ρουτίνα ChooseSubTree αποφασίζουμε ποιες διακλαδώσεις θα ακολουθήσει το τετράγωνο ώσπου τελικά μέσα από αναδρομικές κλήσεις της ρουτίνας αυτής να καταλήξει και να εισαχθεί στο κατάλληλο φύλλο και να προσαρμοστεί πάλι το MBB έπειτα από την εισαγωγή αυτή. Αν το φύλλο περιέχει Μ αντικείμενα (χωρητικότητα κόμβου) καλείται η ρουτίνα Split.Η ρουτίνα αυτή διαιρεί τον κόμβο σε 2 , όπου κάποια από τα τετράγωνα παραμένουν στον παλιό (αυτός που υπήρχε πριν την διαίρεση) και τα υπόλοιπα μετακινούνται στον καινούριο κόμβο. Στην συνέχεια μετά την διαίρεση ενημερώνουμε και τους κόμβους που βρίσκονται στα παραπάνω επίπεδα, τοποθετώντας μία ενημερωμένη με την καινούρια τιμή αναφορά στον παλιό κόμβο και μία καινούρια αναφορά για τον κόμβο που δημιουργήθηκε μετά την διαίρεση, αν χρειαστεί γίνεται διαίρεση με ανάλογο τρόπο και σε αυτούς τους κόμβους των παραπάνω επιπέδων , με τον ίδιο τρόπο , όπως και με τα φύλλα. Η διαδικασία είναι και αυτή αναδρομική όπως και η ChooseSubTree.

2.1 Σχετικές Λειτουργίες

Υπάρχουν διάφορες μορφές R-tree δομών οι οποίες ακολουθούν διαφορετικές στρατηγικές στον τρόπο που αντιμετωπίζουν την διαίρεση των κόμβων σε περίπτωση υπερχείλισης. Ο Guttman προτείνει δύο άπληστες στρατηγικές για το Split, η μία γίνεται σε γραμμικό χρόνο και η άλλη σε τετραγωνικό. Πειράματα έδειξαν πως ο αλγόριθμος που τρέχει σε τετραγωνικό χρόνο έχει καλύτερη επίδοση

ως προς την αναζήτηση. Πολλά συστήματα διαχείρισης βάσεων δεδομένων (DBMSs) , όπως η PosteSQL κάνουν χρήση της τετραγωνικής αυτής στρατηγικής.

Στα R*-trees το Split γίνεται με μια διαφορετική προσέγγιση όπου εκεί λαμβάνονται υπόψη τα διάφορα κριτήρια που ισχύουν στα MBBs του κάθε κόμβου. Αυτή η διαφορετική προσέγγιση δικαιώθηκε αργότερα από ένα μοντέλο σταθερού κόστους , όπου ο όγκος και η περίμετρος των boxes ήταν οι κύριοι παράμετροι. Επίσης , κάτι άλλο που ισχύει στις δομές αυτές είναι ότι οι διαιρέσεις αναβάλλονται από την επανεισαγωγή στοιχείων μακριά από το κέντρο του MBB. Αυτό , βελτιώνει την επίδοση και αυξάνει την χρήση αποθήκευσης.

Το δέντρο Hilbert αποτελεί μια ακόμη μορφή του R-tree που συμπεριφέρεται σαν ένα B+-tree κατά την εισαγωγή ενός στοιχείου. Η κύρια ιδέα είναι να εισάγουμε δεδομένα στους κόμβους χρησιμοποιώντας την τιμή Hilbert του κέντρου των τετραγώνων. Αλλά τα πειράματα έδειξαν ότι η ποιότητα της αναζήτησης δεν είναι καλή καθώς βασίζεται σε μονοδιάστατη στρατηγική εισαγωγής ενός στοιχείου. Επίσης, χρειάζεται έναν προκαθορισμένο χώρο δεδομένων σε αντίθεση με τα γνήσια R-trees που δεν χρειάζεται να προκαθορίσουμε και που δεν χρειάζεται κατά την εισαγωγή να επανασχεδιάσουμε όλο το δέντρο άλλα να μεταβάλλουμε μόνο κάποια όρια.

Έχει υπολογιστεί πως το πρόβλημα της διαίρεσης ενός κόμβου σε δύο (splitting) είναι βέλτιστο με χρόνο $O(M^{dim})$ (όπου dim, οι διαστάσεις του τετραγώνου) ,και επίσης η βέλτιστη αυτή μέθοδος μόνο οριακό βελτιώνει το όριο της επίδοσης του R-tree αλγορίθμου. Μια βελτίωση στην επίδοση μπορεί να επιτευχθεί και από καθολικές βελτιστοποιήσεις και όχι μόνο στον αλγόριθμο της διαίρεσης, αλλά και αυτές οι τεχνικές απαιτούν επανασχεδίαση του δέντρου σε μεγάλο βαθμό. Έτσι τίθεται το ερώτημα πότε θα καταφέρουμε μέσα από μεθόδους συγχρονισμού να προσαρμόσουμε πιο αποτελεσματικές τεχνικές στις υλοποιήσεις των R-trees οι οποίες θα είναι διαθέσιμες στα DBMSs.

Σημαντική προσοχή δόθηκε και στην μαζική φόρτωση των στοιχείων. Έχουν προταθεί διάφορες τεχνικές μαζικής φόρτωσης που μειώνουν το κόστος σε σχέση με τις απλές μεθόδους που μόνο μια πλειάδα εισέρχεται κάθε φορά. Οι τεχνικές αυτές δημιουργούν worst-case βέλτιστα R-trees , όπου ο χρόνος φόρτωσης αντιστοιχεί με μια ταξινόμηση. Η μαζική όμως φόρτωση είναι πέρα από τα πεδίο που θέλουμε να καλύψουμε, οπότε τα τετράγωνα θα εισέρχονται ένα τη φορά.

Ενώ άλλοι αλγόριθμοι προσπαθούν να προσαρμόσουν και να μελετήσουν το πρόβλημα της εισαγωγής διδιάστατων και τρισδιάστατων τετραγώνων, τα R-trees εφαρμόζονται στην οργάνωση πολυδιάστατων σημειακών δεδομένων . Έρευνες ,βέβαια , έδειξαν ότι τα R-trees δεν είναι κατάλληλα γι αυτό τον σκοπό, και ως συνέπεια αυτού έχουν προταθεί διάφορες άλλες μέθοδοι που προσπαθούν να βελτιώσουν τις ελλείψεις των R-trees. Παράδειγμα αποτελούν τα X-trees , που είναι

μια γενίκευση των R*-trees , όπου οι «άσχημες» διαιρέσεις αποφεύγονται αυξάνοντας δυναμικά το μέγεθος των σελίδων. Όμως, ο χειρισμός του μεταβλητού μεγέθους της σελίδας δεν υποστηρίζεται στα συστήματα βάσεων δεδομένων, καθώς εκεί είναι πιο πολύπλοκη η διαχείριση του ελεύθερου χώρου και της μνήμης. Απλά να αναφέρουμε ότι κάποιες άλλες δομές που έχουν προταθεί είναι τα SR-trees , όπου εκεί υπάρχουν και δείκτες εκτός του MBB και Minimum Bounding Spheres, τα A-trees , που προσπαθούν να μειώσουν την χωρητικότητα των κόμβων με κάποιες στρατηγικές συμπύκνωσης, και τα hybrid-trees , που αποτελούν έναν έξυπνο συνδυασμό των kd-trees και R-trees.

2.2 Συμβολισμοί

Ορίζουμε το R= $[min_1 , max_1]$ *…..*$[min_{dim} , max_{dim}]$, όπου συμβολίζει ένα τετράγωνο dim διαστάσεων.

Το χ_d (R) είναι το κέντρο του $[min_d , max_d]$ και το λ_d (R) το μήκος , 1 ≤ d ≤ dim.

Η περίμετρος και ο όγκος του τετραγώνου R δίνονται από τους παρακάτω τύπους:

perim (R) = $\sum_{d=1}^{dim} \lambda\, d(R)$ και vol (R) = $\prod_{d=1}^{dim} \lambda d(R)$

2.3 Ανάλυση των προγενέστερων δομών

Στο υποκεφάλαιο αυτό θα αναλύσουμε καλύτερα και θα παρουσιάσουμε με περισσότερες λεπτομέρειες τις μεθόδους προσπέλασης χωρικών δεδομένων (SAMs-spatial access methods) πάνω στις οποίες βασίστηκε η δημιουργία των δομών του RR*-tree. Οι μέθοδοι αυτοί βασίζονται στην προσέγγιση ενός συνόλου από χωρικά στοιχεία με ελάχιστα τετράγωνα, με τις πλευρές των τετραγώνων αυτών παράλληλες στους άξονες του χώρου δεδομένων.

Το R-tree ,αποτελεί μία από τις πιο γνωστές μεθόδους πρόσβασης για τετράγωνα, η οποία βασίζεται σε με ευρεστική βελτιστοποίηση της περιοχής που καλύπτει κάθε τετράγωνο μέσα στον κόμβο που ανήκει. Έπειτα από πολλές δόκιμες διάφορων δεδομένων σχεδιάστηκε το R*-tree , το οποίο ενσωματώνει έναν συνδυασμό από βελτιστοποιήσεις στην περιοχή , στα όρια και στην επικάλυψη των τετραγώνων που βρίσκονται σε έναν κατάλογο. Αποδείχθηκε ότι το R*-tree υπερέχει των διάφορων παραλλαγών R-tree που υπάρχουν (Guttmann γραμμική και τετραγωνική και Greene).

Το R*-tree είναι ελκυστικό για δύο λόγους, πρώτον, μπορεί να υποστηρίξει σημειακά και χωρικά δεδομένα ταυτόχρονα και δεύτερον, οι υλοποιήσεις είναι ελάχιστα πιο ακριβές από αυτές του R-tree.

Η πιο σημαντική ιδιότητα αυτής της προσέγγισης είναι ότι ένα σύνθετο στοιχείο μπορεί να αναπαρασταθεί από έναν ορισμένο αριθμό bytes. Παρόλο που πολλή πληροφορία χάνεται , τα ελάχιστα τετράγωνα των χωρικών αυτών στοιχείων διατηρούν ουσιαστικές ιδιότητες αυτών , όπως για παράδειγμα , την τοποθεσία του στοιχείου και την έκτασή του σε κάθε άξονα.

Γνωστές SAMs οργανώνουν τα ελάχιστα τετράγωνα βασισμένες στις μεθόδους πρόσβασης σημειακών δεδομένων (PAM - point access method) χρησιμοποιώντας τρεις από τις ακόλουθες τεχνικές, αποκοπή (ψαλίδισμα), μετασχηματισμό και επικάλυψη περιοχών.

Η πιο γνωστή SAM για αποθήκευση τετραγώνων είναι το R-tree [3]. Το R-tree βασίζεται στην PAM B+ -tree [20] χρησιμοποιώντας την τεχνική της επικάλυψης περιοχών.

2.3.1 B-δέντρα και B+ -δέντρα

Αξίζει σε αυτό το σημείο να αναφερθούμε στα B-trees και B+-trees για να κατανοήσουμε κάποιες βασικές έννοιες πριν προχωρήσουμε στην περεταίρω ανάλυση των SAMs.

B-δέντρα

Πολλές φορές έχουμε περισσότερα δεδομένα απ' ότι μπορούμε να κρατήσουμε στη μνήμη και χχρειαζόμαστε μία δομή δεδομένων που να διατηρείται στο δίσκο**.** Ο αριθμός των προσπελάσεων στο δίσκο γίνεται σημαντικός όταν ο όγκος των δεδομένων γίνεται αρκετά μεγάλος για να διατηρηθεί στη μνήμη. Γι αυτό τον λόγο δημιουργήθηκαν M-δρόμων δέντρα αναζήτησης (αύξηση των διακλαδώσεων - μείωση βάθους - Ύψος : $\log_2 N$ > Ύψος M-δρόμων : $\log_M N$)
Τα B-δέντρα είναι η πιο δημοφιλής δομή δεδομένων για αναζήτηση σε δίσκο,ένα B-δέντρο τάξης n είναι ένα n-δέντρο με συγκεκριμένες ιδιότητες. Δηλαδή:

Ορισμός B-δέντρου *n*-τάξης

Κάθε μονοπάτι από τη ρίζα προς το φύλλο έχει το ίδιο μήκος h >= 0
Κάθε κόμβος εκτός από τη ρίζα και τα φύλλα έχει τουλάχιστον υποδέντρα και κλειδιά

Κάθε κόμβος έχει το πολύ n-υποδέντρα και n-1 κλειδιά
Η ρίζα έχει τουλάχιστον δύο υποδέντρα ή είναι φύλλο

Για την εισαγωγή ενός στοιχείου στο Β-δέντρο ακολουθούμε την παρακάτω διαδικασία:

Βρες το φύλλο για να εισάγεις νέο κόμβο

Εάν δεν είναι πλήρες, εισήγαγε το κλειδί στην κατάλληλη θέση

Αλλιώς δημιούργησε έναν νέο κόμβο και μοίρασε τα περιεχόμενα του παλιού κόμβου.

Αριστερός και δεξιός κόμβος
Τα n/2 μικρότερα κλειδιά, πηγαίνουν στον αριστερό κόμβο
Τα n/2 μεγαλύτερα κλειδιά πηγαίνουν στο δεξιό κόμβο
Το κλειδί διαχωριστής (separator) ανεβαίνει στον κόμβο πατέρα.
Εάν το *n είναι άρτιος*:
Ένα επιπλέον κλειδί στον αριστερό (left bias) ή δεξί κόμβο (right bias).

Εάν ο κόμβος πατέρας είναι πλήρης
διαίρεσε τον και προχώρησε με τον ίδιο τρόπο, ο νέος πατέρας μπορεί να διαιρεθεί πάλι εάν είναι πλήρης.

Οι διαιρέσεις μπορούν να φτάσουν στη ρίζα, η οποία μπορεί επίσης να διαιρεθεί δημιουργώντας μία νέα ρίζα, οι αλλαγές προωθούνται από τα φύλλα προς τη ρίζα και το δέντρο παραμένει ισοζυγισμένο. Τα πλεομεκτήματα της χρήσης μιας τέτοιας δομής είναι τα εξής:
Το δέντρο παραμένει ισοζυγισμένο
Καλή χρήση του χώρου αποθήκευσης: περίπου 50%
Χαμηλού ύψους δέντρο
Ίσο ύψος για όλα τα φύλλα συνεπώς, ίδιος αριθμός προσπελάσεων δίσκου για όλα τα φύλλα

Η διαγραφή ενός στοιχείου στο Β-δέντρο γίνεται με τα παρακάτω βήματα:

‘Εχουμε δύο περιπτώσεις:

Συγχώνευση: Εάν είναι λιγότερα από n/2 κλειδιά στο αριστερό και δεξί κόμβο τότε, σύνολο κλειδιών+κλειδί διαχωριστή στο κόμβο πατέρα < n
οι δύο κόμβοι συγχωνεύονται,
ένας κόμβος ελευθερώνεται
Δανεισμός: εάν μετά τη διαγραφή ο κόμβος έχει λιγότερα από n/2 κλειδιά και ο αριστερός (ή δεξιός) κόμβος έχει περισσότερα από n/2 κλειδιά
βρες τον διάδοχο του διαχωριστή
βάλε το διαχωριστή στη θέση του διαγραμμένου κλειδιού
βάλε τον διάδοχο σαν κλειδί διαχωριστή

B+-δέντρα

Για τα B+-δέντρα ισχύουν οι παρακάτω ιδιότητες:

Κάθε κόμβος εκτός από τη ρίζα και τα φύλλα έχει τουλάχιστον $\lceil n/2 \rceil$ υποδέντρα και $\lceil n/2 \rceil$ - 1 κλειδιά
Κάθε κόμβος έχει το πολύ n υποδέντρα και n -1 κλειδιά
Η ρίζα έχει τουλάχιστον δύο υποδέντρα και 1 κλειδί
Τα δεδομένα είναι αποθηκευμένα στα φύλλα

B+-δέντρο κύριος κατάλογος: τα φύλλα περιέχουν εγγραφές δεδομένων
B+-δέντρο δευτερεύων κατάλογος: τα φύλλα περιέχουν κλειδιά και δείκτες προς τις εγγραφές
Όλα τα φύλλα βρίσκονται στο ίδιο επίπεδο

Σύγκριση B- και B+-δέντρων

Στα B+ -δέντρα τα δεδομένα βρίσκονται μόνο στα φύλλα ,οι εσωτερικοί κόμβοι περιέχουν μόνο κλειδιά και διευθύνσεις.

Τα B+-δέντρα είναι ρηχά και αρκετά πλατιά συνεπώς αποτελεσματικά κατά την αναζήτηση και είναι καλύτερα για ερωτήσεις διαστήματος

Εισαγωγή στοιχείου στο B+-δέντρο

Βρες το κατάλληλο κόμβο φύλλο για να εισάγεις το κλειδί
Εάν είναι πλήρες:
Δημιούργησε έναν νέο κόμβο
Διαίρεσε τα περιεχόμενα του, και
Εισήγαγε το κλειδί διαχωριστή στον κόμβο πατέρα
Εάν ο πατέρας είναι πλήρης
Δημιούργησε έναν νέο κόμβο και διαίρεσε με τον ίδιο τρόπο
Συνέχισε προς τα πάνω εάν είναι απαραίτητο
Εάν η ρίζα διαιρείται δημιούργησε έναν νέο κόμβο με δύο υπό-δέντρα

Διαγραφή στοιχείου στο B+-δέντρο

Βρες και διέγραψε το κλειδί από το φύλλο

Εάν το φύλλο έχει < n/2 κλειδιά
Δανεισμός εάν οι κόμβοι γείτονες έχουν περισσότερα από n/2 κλειδιά
(ο διαχωριστής κλειδί μπορεί να αλλάξει) ή

Συγχώνευση με γείτονα εάν και οι δύο έχουν < n κλειδιά, προκαλεί διαγραφή του διαχωριστή στον κόμβο πατέρα

Ενημέρωση του κόμβου πατέρα

Συνέχισε προς τα πάνω εάν ο κόμβος πατέρας δεν είναι ρίζα και έχει λιγότερα από n/2 κλειδιά

B-δέντρα και οι παραλλαγές τους

Τα B-δέντρα και οι παραλλαγές τους είναι οι πιο επιτυχής οργάνωση δεδομένων για δευτερεύουσα αποθήκευση (με βάση το πρωτεύον κλειδί)
Πολύ καλή απόδοση, συγκρίσιμη με το κατακερματισμό
Πλήρως δυναμική
Καλή χρήση του χώρου

B-δέντρα vs B+-δέντρα

B-δέντρα: μη επανάληψη κλειδιού, καλύτερα σε τυχαίες προσπελάσεις (δεν χρειάζεται πάντα να φτάσουν σε φύλλο), σελίδες δεδομένων σε κάθε κόμβο.
B+- δέντρα : επανάληψη κλειδιού, σελίδες δεδομένων μόνο στους κόμβους φύλλα, καλύτερα για ερωτήσεις διαστήματος, ευκολότερα στην υλοποίηση.

2.3.2 R-trees – Αρχές και Πιθανά Κριτήρια Βελτιστοποίησης.

Αφού είδαμε τις ιδιότητες των B-δέντρων και B+-δέντρων , συνεχίζουμε με τα R-trees , τις βασικές αρχές τους και τα πιθανά κριτήρια βελτιστοποίησής τους.

Για έναν εσωτερικό κόμβο του δέντρου ισχύει ότι περιέχει εκχωρήσεις της μορφής (cp,Rectangle) όπου το cp είναι η διεύθυνση ενός κόμβου παιδιού στο δέντρο και Rectangle είναι το ελάχιστο τετράγωνο στο οποίο περιέχονται όλα τα τετράγωνα που βρίσκονται τοποθετημένα σε αυτόν τον κόμβο παιδί.

Ένα φύλλο περιέχει εκχωρήσεις της μορφής (Oid,Rectangle) ,όπου το Oid αναφέρεται σε μια εγγραφή στην βάση δεδομένων, που περιγράφει ένα χωρικό δεδομένο και το Rectangle είναι το τετράγωνο που επισυνάπτεται με αυτό το δεδομένο. Επίσης, τα φύλλα περιέχουν κόμβους που οι εκχωρήσεις μπορεί να είναι της μορφής (dataobject,Rectangle) , αλλά στην προσέγγιση που κάνουμε παρακάτω δε θα εξετάσουμε τέτοιους κόμβους μιας και η εισαγωγή αυτών δεν επηρεάζει την βασική δομή του R-tree.

Θεωρούμε το M τον μέγιστο αριθμό εκχωρήσεων που μπορούν να εισαχθούν σε έναν κόμβο και m τον ελάχιστο αριθμό εκχωρήσεων ($2 \leq m \leq M/2$).Ένα R-tree ικανοποιεί τις ακόλουθες ιδιότητες:

Η ρίζα έχει τουλάχιστον δύο παιδιά , εκτός και αν είναι φύλλο.

Κάθε εσωτερικός κόμβος έχει παιδιά από m μέχρι και M , εκτός και αν είναι η ρίζα.

Κάθε φύλλο έχει παιδιά από m μέχρι και M , εκτός και αν είναι η ρίζα.

Όλα τα φύλλα βρίσκονται στο ίδιο επίπεδο.

Ένα R-tree (R*-tree) είναι απόλυτα δυναμικό, οι εισαγωγές και οι διαγραφές αναμειγνύονται με τις επερωτήσεις και δεν απαιτείται περιοδική ανασύνταξη. Προφανώς, η δομή θα πρέπει να επιτρέπει την επικάλυψη τετραγώνων , αλλά αυτό δεν μπορεί να εγγυηθεί ότι μόνο ένα μονοπάτι αναζήτησης απαιτείται για να ταιριαστεί μία επερώτηση. Θα δούμε ότι οι τεχνικές που θα χρησιμοποιηθούν για την επικάλυψη περιοχών δεν συνεπάγονται άσχημη μέση βελτίωση στην επίδοση. Παρακάτω χρησιμοποιείται ο όρος ορθογώνιο καταλόγου, που είναι γεωμετρικά το ελάχιστο τετράγωνο.

Το κύριο μέλημα στα R-trees είναι πως θα καταφέρουν μετά από διάφορες αλλαγές και εισαγωγές αυθαίρετων τετραγώνων να επιφέρουν την καλύτερη δυνατή επιδιόρθωση από πλευράς επίδοσης. Στην προσπάθειά μας να βελτιστοποιήσουμε κάποιες μεμονωμένες παραμέτρους χωρίς να επηρεάσουν τις υπόλοιπες μπορεί να οδηγηθούμε στην αλλοίωση της συνολικής επίδοσης. Επιπλέον, καθώς τα τετράγωνα δεδομένων δεν έχουν καθορισμένο μέγεθος και σχήμα και τα τετράγωνα καταλόγου αυξομειώνονται δυναμικά , η επιτυχία των μεθόδων που βελτιστοποιούν μία παράμετρο είναι πολύ αβέβαιη.

Στο υποκεφάλαιο αυτό θα μελετήσουμε σημαντικές παραμέτρους για την βελτίωση της επίδοσης και θα δούμε τις αλληλεξαρτήσεις αυτών καθώς θα αναλυθούν και κριτήρια βελτιστοποίησης.

(O1) Η περιοχή που καλύπτεται από τετράγωνα καταλόγου πρέπει να ελαχιστοποιηθεί, δεν θα πρέπει απλά να υπάρχει ένα τετράγωνο που απλά καλύπτει όλα τα υπόλοιπα που βρίσκονται στο παρακάτω επίπεδο, θα πρέπει να είναι και το ελάχιστο. Αυτό βελτιώνει την επίδοση διότι μειώνει τα μονοπάτια.

(O2) Η επικάλυψη μεταξύ των τετραγώνων καταλόγου πρέπει να ελαχιστοποιηθεί, και αυτό επίσης μειώνει το σύνολο των μονοπατιών στο δέντρο.

(O3) Θα ήταν συνετό να ομαδοποιούμε τα τετράγωνα με τέτοιο τρόπο έτσι ώστε τα μήκη των πλευρών τους να μην έχουν μεγάλη διαφορά από αυτή που έχουν τα μήκη των πλευρών του τετραγώνου καταλόγου που τα καλύπτει.

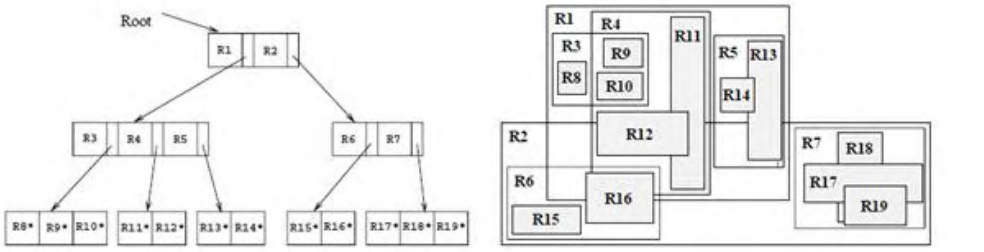
(O4) Όσο το δέντρο είναι χαμηλό η χρήση της μνήμης είναι μειωμένη. Διότι αν έχουμε ένα μεγάλο σύνολο από επερωτήσεις και το ύψος είναι μεγάλο θα πρέπει να διατρέξουμε πολλές σελίδες και ψηλά στο δέντρο με αποτέλεσμα να κάνουμε περισσότερο χρήσης της αποθήκης για να επαναφέρουμε σελίδες από την μνήμη.

Κρατώντας την περιοχή του τετραγώνου καταλόγου μικρή έχουμε περισσότερη ελευθερία στον αριθμό των τετραγώνων που αποθηκεύονται σε έναν κόμβο. Ελαχιστοποιώντας τις παραμέτρους αυτές θα καταβληθεί λιγότερη χρήση της μνήμης. Όπως και σε κάθε γεωμετρική βελτιστοποίηση, ελαχιστοποιώντας τα περιθώρια θα μπορέσουμε να μειώσουμε την χρήση της μνήμης.

2.3.3 Παραλλαγές R-tree

Το R-tree είναι μια δυναμική δομή, γι αυτό όλες οι προσεγγίσεις για την βελτιστοποίηση της επίδοσης πρέπει να εφαρμοστούν κατά τη διάρκεια μιας εισαγωγής ενός νέου τετραγώνου δεδομένων. Ο αλγόριθμος της εισαγωγής καλεί δύο ακόμη αλγορίθμους μέσω των οποίων παίρνονται οι πιο κύριες αποφάσεις για την βελτίωση της απόδοσης. Ο πρώτος είναι ο αλγόριθμος ChooseSubTree, ξεκινώντας από τη ρίζα και καταλήγοντας σε φύλλο, βρίσκει σε κάθε επίπεδο το καταλληλότερο υποδέντρο να εισάγει τη νέα εκχώρηση. Ο δεύτερος είναι ο αλγόριθμος Split ,καλείται όταν ο ChooseSubTree καταλήξει σε κόμβο που έχει το μέγιστο M των εκχωρήσεων που μπορεί να φιλοξενήσει , και θα πρέπει να διαιρέσει τα M+1 τετράγωνα σε δύο κόμβους με τον καταλληλότερο τρόπο.

Η εικόνα 1 δείχνει πως μπορούμε να αναπαραστήσουμε τα τετράγωνα σε μορφή R-δέντρου.



Εικόνα 1

Θα εξετάσουμε πρώτα το γνήσιο R-tree όπως προτείνεται από τον Guttmann στο [3].

Αλγόριθμος ChooseSubtree

CS1 Ορίζουμε το N ρίζα του δέντρου

CS2 Αν N είναι φύλλο

επέστρεψε N

αλλιώς

επέλεξε την εκχώρηση στο N της οποίας το τετράγωνο χρειάζεται την μικρότερη επιμήκυνση για να εισαχτεί σ αυτό το νέο στοιχείο. Αν υπάρχουν παραπάνω από ένα τέτοια τετράγωνα επιλέγεται αυτό που καλύπτει μικρότερη περιοχή

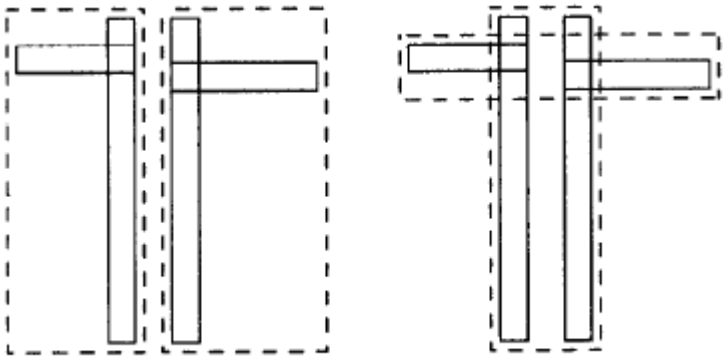
CS3 Θέτουμε το N να είναι πλέον δείχτης στο παραπάνω τετράγωνο και επαναλαμβάνουμε το CS2

Η μέθοδος βελτιστοποίησης στοχεύει στην ελαχιστοποίηση της περιοχής που καλύπτει το τετράγωνο του καταλόγου, για παράδειγμα το (O1) που μπορεί να μειώσει την επικάλυψη και το κόστος cpu είναι σχετικά μικρό.

Ο Guttmann εξετάζει αλγόριθμους split με εκθετικό, τετραγωνικό και γραμμικό κόστος δίνοντας πάντα προσοχή στον αριθμό των εκχωρήσεων σε ένα κόμβο. Αυτές οι τρεις περιπτώσεις έχουν σχεδιαστεί για να ελαχιστοποιούν την περιοχή που καλύπτεται από τα δύο τετράγωνα που είναι αποτέλεσμα του splitt. Το εκθετικό split βρίσκει την περιοχή αυτή έχοντας ως στόχο να ελαχιστοποιήσει την συνολική περιοχή , αλλά αυτό έχει ως αποτέλεσμα μεγάλο cpu κόστος. Οι άλλες δύο περιπτώσεις split προσπαθούν να βρουν άλλες προσεγγίσεις. Στα πειράματά του ο Guttman δείχνει ότι η επίδοση του αλγορίθμου είναι παρόμοια για το γραμμικό και το τετραγωνικό split. Παρακάτω θα δούμε αναλυτικά και τους δύο αυτούς τρόπους. Όμως, σε πειράματα που έγιναν με διαφορετικές τιμές σε παραμέτρους , όπως στην επικάλυψη , και στα M, m καθώς και με διάφορες τιμές στα τετράγωνα που εισήχθησαν , φαίνεται πως το τετραγωνικό split έχει καλύτερη επίδοση απ' ότι το γραμμικό.

Να τονίσουμε πως σημαντικό κατά τη διαίρεση ενός κόμβου είναι να ελαχιστοποιήσουμε και την περιοχή που καλύπτει τους καινούριους κόμβους.

Το παρακάτω σχήμα (εικόνα 2) δείχνει πως τα ίδια τετράγωνα μπορούν διαιρεθούν και να πάρουμε δύο τελείως διαφορετικά αποτελέσματα.



Εικόνα 2

<u>Αλγόριθμος Split (τετραγωνικό)</u>

[Διαίρεση ένα σύνολο από M+1 εκχωρήσεις σε δύο ομάδες]

QS1 Καλούμε την *Pickseeds* για να διαλέξουμε την εκχώρηση που θα μπει πρώτη στον καινούριο κόμβο.

QS2 Επανάλαβε

DistributeEntry

μέχρι

όλες οι εισαγωγές να διανεμηθούν ή ένας από τους δύο κόμβους να έχει M-m+1 στοιχεία

QS3 Εάν απομένουν εκχωρήσεις, εισάγονται στον κόμβο εκείνο με τον ελάχιστο αριθμό εκχωρήσεων

Αλγόριθμος PickSeeds

PS1 Για κάθε ζευγάρι εκχωρήσεων E1 και E2, δημιούργησε ένα τετράγωνο R που να περιέχει τα E1 και E2.

Υπολόγισε το d= area(R) – area(E1) – area(E2)

PS2 Επέλεξε το ζευγάρι με το μεγαλύτερο d.

Αλγόριθμος DistributeEntry

DE1 Κάλεσε την *PickNext* για την επόμενη εκχώρηση

DE2 Πρόσθεσε την εκχώρηση αυτή στον κόμβο που το τετράγωνο που τον καλύπτει πρέπει να επιμηκυνθεί λιγότερο. Αν υπάρχουν περισσότερα από ένα τέτοια τετράγωνα για να επιλύσουμε τις συγκρούσεις η προσθήκη της εκχώρησης γίνεται σε κείνο το τετράγωνο που καλύπτει την μικρότερη περιοχή ή μετά αυτό που έχει τις λιγότερες εισαγωγές.

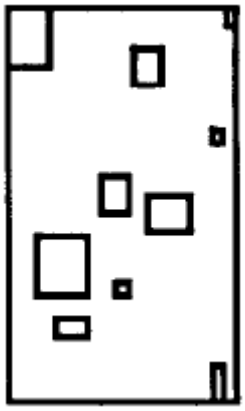
Αλγόριθμος PickNext

PN1 Για κάθε εκχώρηση Ε που δεν έχει εισαχθεί ακόμη σε κάποιο από τους δύο κόμβους υπολογίζουμε το d1= το πόσο θα μεγαλώσει η περιοχή του τετραγώνου που καλύπτει τον κόμβο1 αν σ αυτόν τοποθετήσουμε το Ε και αναλόγως υπολογίζουμε το d2 για το κόμβο2.

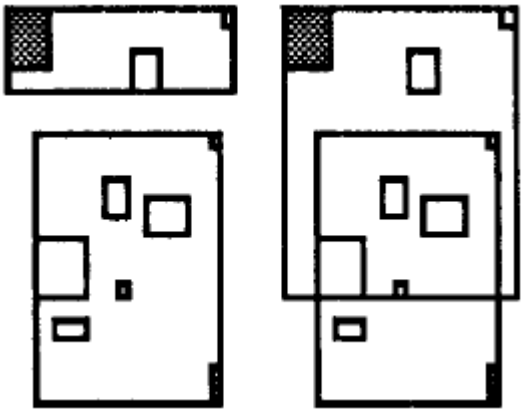
PN2 Επιλέγουμε την εκχώρηση εκείνη με την μέγιστη διαφορά ανάμεσα στα d1 και d2.

Ο αλγόριθμος PickSeeds βρίσκει ποια δύο τετράγωνα καταναλώνουν μεγαλύτερη περιοχή αν τοποθετηθούν σε μια ομάδα. Με αυτή τη λογική τα δύο αυτά τετράγωνα είναι τα πιο απομακρυσμένα. Ακόμη και τα τετράγωνα (seeds) που υπολογίζονται στην PickSeeds θα τείνουν να είναι μικρότερα , αν τα τετράγωνα που περιμένουν να διανεμηθούν είναι πολύ διαφορετικού μεγέθους και η επικάλυψη μεταξύ αυτών είναι μεγάλη. Ο αλγόριθμος DistributeEntry προσδιορίζει τις εκχωρήσεις που παραμένουν με το κριτήριο της ελάχιστης περιοχής και ο PickNext διαλέγει την εκχώρηση με την καλύτερη τιμή ως προς την περιοχή για κάθε περίπτωση.

Αν όμως, ο αλγόριθμος ξεκινήσει με μικρά τετράγωνα (small seeds) μπορεί να δημιουργηθούν προβλήματα , αν στους d-1 από τους d άξονες ένα μακρινό τετράγωνο έχει περίπου τις ίδιες συντεταγμένες με ένα από τα seeds, θα διανεμηθεί πρώτο. Αυτό θα έχει ως αποτέλεσμα η περιοχή να μεγαλώσει λίγο για να χωρέσει το καινούριο τετράγωνο αλλά η απόσταση θα είναι πολύ μεγάλη. Έτσι θα χουμε από την αρχή ένα άσχημο split. Ο αλγόριθμος προτιμά να επιλέγει ως το επικαλύπτον τετράγωνο αυτό που σχηματίζεται από το τετράγωνο της πρώτης εκχώρησης και στη συνέχεια να μεγαλώνει .Αυτό απαιτεί λιγότερη επιμήκυνση της περιοχής(σχήμα 1α). Ένα άλλο πρόβλημα είναι πως αν μια ομάδα(από τις δύο που δημιουργούνται κατά την διαίρεση) έχει φτάσει στο μέγιστο των εκχωρήσεων M-m+1, όλες οι υπόλοιπες εισάγονται στην δεύτερη ομάδα ανεξαρτήτως των γεωμετρικών ιδιοτήτων (σχήμα 1β).



Σχήμα 1α : Κόμβος που πρέπει να διαιρεθεί.



Σχήμα 1β Σχήμα 1γ

Το αποτέλεσμα είναι μια διαίρεση με μεγαλύτερη επικάλυψη (σχήμα1γ) ή μια διαίρεση με άνιση διανομή των εκχωρήσεων μειώνοντας την χρήση μνήμης.

Εξετάζοντας την τετραγωνική διαίρεση παίρνοντας για το m διάφορες τιμές , για παράδειγμα 20%,30% (σχήμα 1β) ,35%,40% (σχήμα 1γ) και 45% σε σχέση με το M αποδείχθηκε πως η καλύτερη επίδοση του αλγορίθμου είναι θέτοντας το m =40%.

Ο Greene πρότεινε έναν εναλλακτικό αλγόριθμο για τον αλγόριθμο Split , ο οποίος χρησιμοποιεί τον αλγόριθμο ChooseSubTree του Guttmann για να αποφασίσει τον κατάλληλο κόμβο και μονοπάτι που θα εισαχθεί ένα καινούριο τετράγωνο.

Αλγόριθμος Split του Greene

[διαιρεί ένα σύνολο M+1 εκχωρήσεων σε δύο ομάδες]

GS1 Καλούμε την *ChooseAxis* για να διαλέξουμε άξονα που θα εκτελεστεί η διαίρεση

GS2 Καλούμε την *Distribute*

Αλγόριθμος ChooseAxis

CA1 Καλούμε την *PickSeeds* για να βρούμε τα δύο πιο απομακρυσμένα τετράγωνα του τρέχοντος κόμβου.

CA2 Για κάθε άξονα καταγράφουμε το πως χωρίζονται τα δύο τετράγωνα (seeds)

CA3 Κανονικοποιούμε τις διαιρέσεις αυτές διαιρώντας τες με το μήκος των κόμβων

CA4 Επιστρέφουμε τον άξονα με την μεγαλύτερη κανονικοποιημένη διαίρεση

Αλγόριθμος Distribute

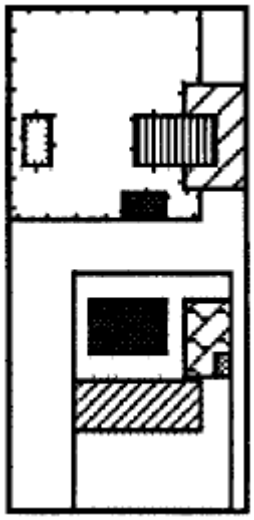
D1 Ταξινομούμε τις εκχωρήσεις με αύξουσα σειρά με βάση την τιμή των τετραγώνων τους στον άξονα που εξετάζουμε

D2 Τοποθετούμε τις πρώτες (M+1) div 2 εκχωρήσεις σε μια ομάδα και τις υπόλοιπες (M+1) div 2 στην άλλη.

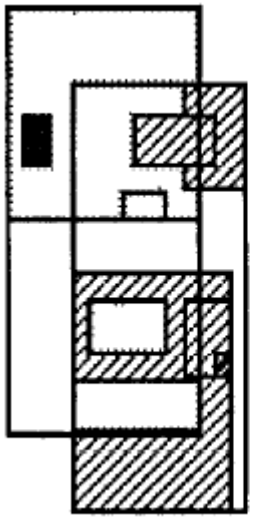
D3 Αν το M+1 είναι περιττός αριθμός, τότε τοποθετούμε την εκχώρηση που περισσεύει στην ομάδα που το επικαλύπτον τετράγωνο θα αυξηθεί λιγότερο από την εισαγωγή της.

Ο αλγόριθμος split του Greene χρησιμοποιεί ως μόνο γεωμετρικό κριτήριο την επιλογή του άξονα για την διαίρεση των τετραγώνων και πολλές φορές παρά την καλή ομαδοποίηση που επιτυγχάνει δεν μπορεί να βρει τον “σωστό” άξονα και ένας

κακός διαχωρισμός φαίνεται στο σχήμα 2β. Επίσης , πειράματα έδειξαν πως περισσότερα κριτήρια θα πρέπει να ληφθούν υπόψη για να πετύχουμε καλύτερη επίδοση στα R-trees.

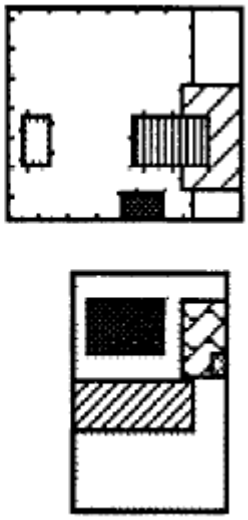


Σχήμα 2α : Κόμβος που πρέπει να διαιρεθεί.



Σχήμα 2β

Διαίρεση ως προς τον οριζόντιο άξονα.



Σχήμα 2γ

Διαίρεση ως προς τον κάθετο άξονα.

2.4 R*-TREE

2.4.1 Αλγόριθμος ChooseSubTree

Για να λυθεί το πρόβλημα της επιλογής κατάλληλου μονοπατιού κατά την εισαγωγή ενός νέου τετράγωνου ,οι προηγούμενες εκδόσεις του R-tree λάμβαναν υπόψη τους μόνο την παράμετρο της περιοχής. Στον R*-tree λαμβάνονται υπόψη η περιοχή , το περιθώριο και η επικάλυψη , σε διάφορους συνδυασμούς μεταξύ τους , όπου η επικάλυψη ορίζεται ως εξής :

Θέτουμε E1,...Ep τις εκχωρήσεις στον τρέχοντα κόμβο , η επικάλυψη θα είναι overlap(Ek) = Σ(i=1,i≠k ως p) area(Ek Rectangle ∩ Ei Rectangle) , 1≤k≤p.

CS1 Θέτουμε N την ρίζα

CS2 Αν N είναι φύλλο

επέστρεψε N

αλλιώς

αν οι δείχτες των παιδιών του N είναι σε φύλλα [αποφάσισε το ελάχιστο κόστος επικαλύψεις],

επέλεξε την εκχώρηση εκείνη του N που το τετράγωνό της χρειάζεται να αυξήσει λιγότερο την επικάλυψή του έτσι ώστε να εισαχθεί σ'αυτό το νέο τετράγωνο-δεδομένο, και λύνουμε τυχόν συγκρούσεις επιλέγοντας μετά την εκχώρηση με την μικρότερη αύξηση της περιοχής.

διαφορετικά,επέλεξε την εκχώρηση με την μικρότερη περιοχή.

αν οι δείχτες των παιδιών του N δεν είναι σε φύλλα

[αποφασίζουμε με βάση την ελάχιστη περιοχή] επέλεξε εκείνη την εκχώρηση στο N που το τετράγωνό της χρειάζεται να αυξήσει λιγότερο την περιοχή του για να εισαχθεί το νέο τετράγωνο,και λύνουμε τις συγκρούσεις επιλέγοντας το

τετράγωνο με την μικρότερη περιοχή.

CS3 Θέτουμε το N να είναι πλέον δείκτης στο παραπάνω τετράγωνο(παιδί του παλιού N και επαναλαμβάνουμε το CS2

Για την επιλογή του καλύτερου εσωτερικού κόμβου , οι εναλλακτικές μέθοδοι δεν υπερέχουν του γνήσιου αλγορίθμου του Guttman. Για τα φύλλα ,η ελαχιστοποίηση της επικάλυψης εκτελείται ελαφρώς καλύτερα.

Σε αυτή την έκδοση, το cpu κόστος καθορισμού της επικάλυψης είναι τετραγωνικό σε σχέση με τον αριθμό των εκχωρήσεων, διότι πρέπει να υπολογιστεί η επικάλυψη ενός τετραγώνου με όλα τα υπόλοιπα του κόμβου. Βέβαια , μπορούμε να μειώσουμε το μέγεθος του κόστους αν τα τετράγωνα σε έναν κόμβο είναι πολλά , αν δεν υπολογίσουμε κάποιες επικαλύψεις που δεν χρειάζονται , καθώς για απομακρυσμένα τετράγωνα η πιθανότητα να έχουν την ελάχιστη επικάλυψη με το τετράγωνο που θέλουμε να εισάγουμε είναι μικρή.Έτσι , εκείνο το κομμάτι του αλγορίθμου[CS2 στο αλλιώς το πρώτο αν] μπορεί να μετασχηματιστεί ως εξής:

[αποφάσισε την κοντινότερη ελάχιστη επικάλυψη] Ταξινόμησε τα τετράγωνα στον N με αύξουσα σειρά ως προς την μεγέθυνση που πρόκειται να συμβεί στις περιοχές τους με την εισαγωγή του νέου τετραγώνου.Θέτουμε A την ομάδα με τις πρώτες ρ εκχωρήσεις, και από το A,επιλέγουμε εκείνη που η επικάλυψη θα μεγαλώσει λιγότερο,επιλύουμε τις συγκρούσεις όπως παραπάνω.

Για δύο διαστάσεις αν θέσουμε το ρ=32 η βελτίωση της επίδοσης είναι σχετικά μικρή, σε περισσότερες διαστάσεις πρέπει να γίνουν πειράματα. Ωστόσο, το cpu κόστος παραμένει υψηλό σε σχέση με την γνήσια έκδοση ChooseSubTree , αλλά ο αριθμός των προσπελάσεων στον δίσκο έχει μειωθεί, καθώς μειώνεται ο αριθμός τον εκχωρήσεων που ελέγχουμε.

Τα πειράματα έδειξαν πως ο ChooseSubTree βελτιώνει την επίδοση ιδιαίτερα όταν έχουμε επερωτήσεις με μικρά τετράγωνα - δεδομένα με μη ομοιόμορφα διανεμημένα σημεία.Στις υπόλοιπες περιπτώσεις η επίδοση είναι σχεδόν ίδια με αυτή του αλγορίθμου του Guttman.

2.4.2 Αλγόριθμος Split

Το R*-tree χρησιμοποιεί την παρακάτω μέθοδο για να βρει τις κατάλληλες διαιρέσεις.Ταξινομεί τους άξονες με αύξουσα σειρά και τα τετράγωνα σε φθίνουσα. Για κάθε M-2m+2 ταξινομήσεις , αποφασίζονται οι διανομές των M+1 εκχωρήσεων σε δύο ομάδες, όπου η κ-οστή διανομή περιγράφεται ως εξής: (k= 1,...,(M-2m+2)) , η πρώτη ομάδα περιέχει τις πρώτες (m-1) + k εκχωρήσεις και η δεύτερη τις υπόλοιπες.

Για κάθε διανομή αποφασίζονται οι καλές τιμές και βασιζόμενη στις καλές αυτές τιμές αποφασίζεται η τελική διανομή των εκχωρήσεων.Αυτές οι καλές τιμές και οι διάφορες προσεγγίσεις της χρήσης αυτών με διάφορους συνδυασμούς εξετάζονται πειραματικά.

(i) area-value = area[boundingbox(first group)]+ area[bb(second group)]

(ii) margin-value = margin[bb(first group)]+ area[bb(second group)]

(iii) overlap-value = area[bb(first group)]∩ bb(second group)]

Πιθανές μέθοδοι της διαδικασίας είναι να αποφασίσουμε:

-το ελάχιστο των αξόνων ή μιας ταξινόμησης

-το ελάχιστο της πρόσθεσης μιας καλής τιμής σε έναν άξονα ή μια ταξινόμηση

-το ολικό ελάχιστο

Ακολουθεί ο αλγόριθμος με τα καλύτερα αποτελέσματα ως προς την επίδοση:

Αλγόριθμος Split

S1 Καλούμε την *ChooseSplitAxis* για να αποφασίσουμε σε ποιον κάθετο άξονα θα εκτελεστεί η διαίρεση.

S2 Καλούμε την *ChooseSplitIndex* για να αποφασίσουμε την καλύτερη διανομή σε δύο ομάδες κατά μήκος του άξονα του S1.

S3 Οι εκχωρήσεις μοιράζονται στις δύο ομάδες

Αλγόριθμος ChooseSplitAxis

CSA1 Για κάθε άξονα

ταξινομούμε τις εκχωρήσεις σε αύξουσα σειρά ως προς τις τιμές των τετραγώνων τους και υπολογίζουμε το άθροισμα S των margin-values των διανομών.

CSA2 Κρατάμε τον άξονα με την μικρότερη τιμή S

Αλγόριθμος ChooseSplitIndex

CSI1 Αφού έχουμε επιλέξει τον άξονα,επιλέγουμε την διανομή με την ελάχιστη overlap-value και επιλύουμε τυχόν συγκρούσεις επιλέγοντας την διανομή με την ελάχιστη area-value.

Ο αλγόριθμος split δοκιμάστηκε με m= 20%,30%,40% και 45% της τιμής M που κρατάει το μέγιστο των εκχωρήσεων(με το M να παίρνει διάφορες τιμές) και τα πειράματα έδειξαν πως για m= 40% παίρνουμε την καλύτερη επίδοση. Επιπρόσθετα , για την επιλογή του m λαμβανόταν υπόψη και η χρήση της μνήμης με τις γεωμετρικές παραμέτρους. Υπολογίζαμε το split με m1=30% του M και μετά με m2=40% και αν το split(m2) έχει την επιθυμητή επικάλυψη και split(m1) όχι τότε παίρνουμε split(m2), διαφορετικά split(m1).

Το κόστος του αλγορίθμου split είναι:

Για κάθε άξονα (διάσταση) οι εκχωρήσεις πρέπει να ταξινομηθούν δύο φορές το οποίο απαιτεί O(M log(M)) χρόνο. Η πειραματική ανάλυση του κόστους έδειξε ότι αυτό είναι το μισό του κόστους του αλγορίθμου split. Το υπόλοιπο υπολογίζεται ως εξής: Για κάθε άξονα θα πρέπει να υπολογιστούν τα περιθώρια 2*(2(M -2m+2)) τετραγώνων και οι επικαλύψεις 2*(M-2m+2) διανομών.

2.5 Αναγκαστική επανεισαγωγή

Και οι δύο δομές R-tree και R*-tree είναι μη ντετερμινιστικές στο να διανείμουν τις εκχωρήσεις μέσα στους κόμβους ,για παράδειγμα διαφορετικές ακολουθίες από εισαγωγές θα δημιουργήσουν διαφορετικά δέντρα. Γι αυτό τον λόγο , το R-tree υποφέρει από τις παλιές εκχωρήσεις. Τα τετράγωνα-δεδομένα που έχουν εισαχθεί νωρίς στο δέντρο μπορεί να έχουν εισαχτεί σε τετράγωνα καταλόγου, και που στην τρέχουσα κατάσταση δεν είναι κατάλληλα να εγγυηθούν μία καλή επίδοση. Κατά τη διάρκεια που γίνεται split επιτυγχάνεται μια τοπική αναδιοργάνωση του τετραγώνου καταλόγου αλλά αυτό είναι φτωχό και απαιτείται μια πιο κατάλληλη δομή και λιγότερο τοπική.

Το πρόβλημα επιδεινώνεται αν κάποιοι κόμβοι περιέχουν εκχωρήσεις λιγότερες από το κάτω όριο που πρέπει να περιέχουν (αποτέλεσμα διαγραφών) και που πρέπει να συγχωνευτούν κάτω από τον παλιό πατέρα. Η προσέγγιση που ακολουθείται στο R-tree είναι ο κόμβος αυτός να διαγράφεται και οι ορφανές εκχωρήσεις να εισέρχονται στους κατάλληλους κόμβους που υπάρχουν[3]. Έτσι ο αλγόριθμος ChooseSubTree έχει μια ευκαιρία ακόμη να διανείμει ξανά τις ορφανές αυτές εκχωρήσεις σε διαφορετικούς κόμβους.

Όπως ήταν αναμενόμενο , αυτή η διαγραφή και η επανεισαγωγή των παλιών τετραγώνων βελτιώνει την επίδοση, ακολουθεί ένα παράδειγμα της πειραματικής διαδικασίας που το αποδεικνύει.

Παράδειγμα: Έπειτα από εισαγωγή 20000 ομοιόμορφα κατανεμημένων τετραγώνων ,διαγραφή των πρώτων 10000 και εκχώρηση αυτών ξανά. Η επίδοση αυξήθηκε από 20% ως 50%, ανάλογα με το τύπο των επερωτήσεων.

Η διαγραφή των μισών δεδομένων και η επανεισαγωγή τους φαίνεται να είμαι μια απλή διαδικασία συγχρονισμού των αρχείων δεδομένων ενός R-tree αλλά αυτό είναι μια στατική διαδικασία και για σχεδόν στατικά αρχεία δεδομένων υπάρχει ο pack αλγόριθμος [6] όπου είναι πιο σύγχρονη προσέγγιση και αναφερόμαστε με λίγα λόγια σ'αυτόν στο υποκεφάλαιο που ακολουθεί.

Για να επιτύχουμε δυναμικές αναδιοργανώσεις , το R*-tree αναγκάζει τις εκχωρήσεις να επανεισάγονται κατά τη διάρκεια της ρουτίνας insertion. Ο ακόλουθος αλγόριθμος βασίζεται στην ικανότητα της ρουτίνας insert να εισάγει δεδομένα σε κάθε επίπεδο του δέντρου όπως απαιτεί ο αλγόριθμος της διαγραφής [3]. Εκτός από την διαδικασία που γίνεται κατά την υπερχείλιση , είναι ίδια όπως περιγράφεται στον γνήσιο αλγόριθμο του Guttman και γι αυτό είναι η μόνη που περιγράφεται εδώ.

Αλγόριθμος InsertData

ID1 Καλούμε την *Insert* που ξεκινάει από το επίπεδο των φύλλων σαν παράμετρο για να εισάγουμε δεδομένα.

Αλγόριθμος Insert

I1 Καλούμε την *ChooseSubtree* με το επίπεδο σαν παράμετρο, για να βρούμε τον κατάλληλο κόμβο N, στον οποίο θα εισάγουμε την νέα εκχώρηση E.

I2 Αν το N έχει λιγότερες από N εκχωρήσεις εισάγουμε το E στον N

Αν το N έχει M εκχωρήσεις , καλούμε την *OverflowTreatment* με το επίπεδο του N σαν παράμετρο [για reinsertion ή split]

I3 Αν κληθεί η *OverflowTreatment* και πραγματοποιηθεί split , μεταδίδουμε την *OverflowTreatment* αν είναι αναγκαίο και στα παραπάνω επίπεδα.

I4 Προσαρμόζουμε όλα τα επικαλύπτοντα τετράγωνα στο μονοπάτι εισαγωγής τα οποία είναι τα MMBs και περιέχουν τους κόμβους παιδιά.

Αλγόριθμος OverflowTreatment

OT1 Αν το επίπεδο δεν είναι το επίπεδο της ρίζας και αυτή είναι η πρώτη φορά που καλείται η *OverflowTreatment* στο επίπεδο αυτό κατά της διάρκεια της εισαγωγής του τετραγώνου-δεδομένου τότε

Κάλεσε *ReInsert*

else

κάλεσε *Split*

Αλγόριθμος ReInsert

RI1 Για όλες τις M+1 εκχωρήσεις στον κόμβο N , υπολόγισε την απόσταση μεταξύ των κέντρων των τετραγώνων τους και του κέντρου του επικαλύπτοντος τετραγώνου του N.

RI2 Ταξινομήσεις τις εκχωρήσεις σε φθίνουσα σειρά με βάση τις αποστάσεις που υπολογίστηκαν στο RI1.

RI3 Μετακίνησε τις πρώτες ρ εκχωρήσεις από το N και προσάρμοσε το επικαλύπτον τετράγωνο.

RI4 Ξεκίνα με την μέγιστη ή την ελάχιστη τιμή που ορίστηκε στο RI2 και κάλεσε την *ReInser*t για να επανεισάγεις τις εκχωρήσεις.

Αν ένα νέο τετράγωνο-δεδομένο εισαχθεί , κάθε μία υπερχείλιση σε κάθε επίπεδο μπορεί να προκαλέσει την επανεισαγωγή ρ εκχωρήσεων. Αυτό μπορεί να προκαλέσει την διαίρεση σε έναν κόμβο η οποία με τη σειρά της μπορεί να προκαλέσει εκ νέου μια υπερχείλιση αν όλες οι εκχωρήσεις εισαχθούν ξανά στην ίδια τοποθεσία. Σε άλλες περιπτώσεις, διαιρέσεις μπορεί να συμβούν σε έναν ή περισσότερους κόμβους, αλλά σε πολλές περιπτώσεις εμποδίζονται. Η παράμετρος ρ , είναι ανεξάρτητη των κόμβων (φύλλων και εσωτερικών) , και εξετάζεται μέσω πειραμάτων ποια τιμή είναι καταλληλότερη για την βελτιστοποίηση της επίδοσης. Οι έρευνες έχουν δείξει πως το ρ=30% του M των κόμβων μπορεί να επιφέρει τα καλύτερα αποτελέσματα ως προς την επίδοση.

Συνοψίζοντας μπορούμε να πούμε ότι:

-Οι αναγκαστικές επανεισαγωγές αλλάζουν τις εκχωρήσεις ανάμεσα σε γειτονικούς κόμβους και αυτό μπορεί να προκαλέσει μείωση της επικάλυψης.

-Ως επακόλουθο αυτού, η χρήση μνήμης βελτιώνεται.

-Λόγω της περισσότερης αναδόμησης δεν συμβαίνουν πολλές διαιρέσεις.

- Αφού το εξωτερικό τετράγωνο επανεισάγεται , το σχήμα του τετραγώνου καταλόγου γίνεται πιο τετραγωνικό.

Προφανώς, το cpu κόστος θα αυξηθεί καθώς η ρουτίνα insertion καλείται πιο συχνά. Βέβαια αυτό μετριάζεται καθώς πλέον συμβαίνουν λιγότερες διαιρέσεις κόμβων. Τα πειράματα έδειξαν πως ο μέσος αριθμός των προσπελάσεων στο δίσκο αυξάνεται μόνο κατά 4% (και παραμένει το χαμηλότερο απ' όλες τις παραλλαγές του R-tree) , αν η αναγκαστική επανεισαγωγή εφαρμοστεί στο R*-tree, και αυτό γίνεται γιατί η δομή της ReInsertion βασίζεται πάνω στον αλγόριθμο Insertion.

2.5.1 Αλγόριθμος PACK

Σε αυτό το υποκεφάλαιο θα παρουσιάσουμε μια τεχνική αρχικού στοιβάγματος για την δημιουργία R-tree δομών για να ευρετηριοποιήσουμε χωρικά αντικείμενα. Επειδή οι εικονογραφημένες βάσεις δεδομένων δεν ενημερώνονται εντατικά αλλά είναι περισσότερο στατικές, τα οφέλη αυτής της τεχνικής είναι σημαντικά.

Θα δούμε πως αυτή η τεχνική συμπιέζει τον αχρησιμοποίητο χώρο των R-trees με μια καλύτερη χρήση του ευρετηρίου των χωρικών αντικειμένων που έχει ως αποτέλεσμα βελτίωση της επίδοσης. Η αρχική κατασκευή του ευρετηρίου δεν έρχεται σε σύγκρουση με την δυναμική φύση των R-trees, τα οποία μπορούν να ενημερωθούν στη συνέχεια, όμως όπως και οι περισσότερες εικονογραφημένες βάσεις δεδομένων είναι σχετικά στατικές.

Τα "στοιβαγμένα" R-trees έχουν εφαρμοστεί στο πλαίσιο της PSQL(PostgreSQL) που είναι μία γλώσσα αναζήτησης για εικονογραφημένες βάσεις δεδομένων.

2.5.1.1 PSQL – Μία γλώσσα επερωτήσεων σε εικονογραφημένες βάσεις δεδομένων.

Η PSQL είναι μία σχεσιακή γλώσσα για την ανάκτηση πληροφοριών από μια βάση δεδομένων. Επεκτείνει τη δύναμη της SQL , η οποία χρησιμοποιείται για την ανάκτηση αλφαριθμητικών δεδομένων και επιτρέπει τη άμεση χωρική αναζήτηση. Οι εικονογραφημένες βάσεις επιτρέπουν την ένωση μεταξύ χωρικών και αλφαριθμητικών δεδομένων.

Το επόμενο παράδειγμα είναι μία τυπική επερώτηση σε PSQL:

select city,state,populatlon,loc

from cities

on us-map

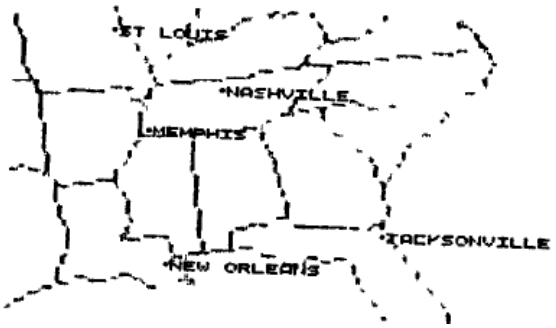
at lot covered-by {4+-4,11+-9}

where population > 450,000

η οποία επιλέγει όλες τις πόλεις στην περιοχή {4+-4,11+-9} (ανατολικές πολιτείες των ΗΠΑ που έχουν εισαχθεί από συντεταγμένες ή με το ποντίκι) που έχουν πληθυσμό άνω των 450.000. Η εικόνα 3α απεικονίζει το αλφαριθμητικό αποτέλεσμα της επερώτησης και η 3β την εικονογραφημένη έξοδο που εμφανίζεται στην οθόνη. Στις εικόνες σημειώνονται τα ονόματα των αντικειμένων για να καταλάβει ο χρήστης τις αντιστοιχίες.

char	char	real
city	state	population
ct	st	pop
JACKSONVILLE	FLORIDA	540,920
NEW ORLEANS	LOUISIANA	557,515
ST LOUIS	MISOURI	453,085
MEMPHIS	TENNESSEE	646,356
NASHVILLE	TENNESSEE	455,651

Εικόνα 3α



Εικόνα 3β

2.5.1.2 R-trees – θεμέλιο της PSQL και η συνάρτηση Pack

Η PSQL βασίζεται πάνω στην SQL και στη γνωστή δομή R-tree όπως ορίζεται από τον Guttman.

Η συνάρτηση αυτή επιδιώκει να ελαχιστοποιήσει την επικάλυψη καθώς και το ίδιο το μέγεθος των τετραγώνων. Δέχεται σαν είσοδο της δεδομένα και παράγει σαν έξοδο σχεδόν-βέλτιστα στοιβαγμένα R-tree , αλλά απαιτεί όχι ειδική περιστροφή ή προσανατολισμό του πλαισίου αναφοράς της βάσης δεδομένων.

Θεωρούμε ότι η χωρική βάση δεδομένων παραμένει σχετικά στατική (για παράδειγμα οι χάρτες των γεωγραφικών περιοχών , δεν απαιτούν συχνές επανεισαγωγές ή ενημερώσεις). Θα ενισχύσουμε την προϋπόθεση του Guttman που ορίζει ένα m ως ελάχιστο αριθμό εκχωρήσεων σε ένα κόμβο , ορίζοντας ότι όλοι οι κόμβοι θα πρέπει να στοιβάζονται όσο το δυνατόν πιο πλήρεις. Για ευκολία θεωρούμε τον παράγοντα διακλάδωσης ίσο με 4 και σε οποιοδήποτε επίπεδο το σύνολο των αντικειμένων είναι πολλαπλάσιο του 4. Αυτό θα ήταν εξαιρετικά απίθανο σε οποιαδήποτε ρεαλιστική εφαρμογή , αλλά η παραδοχή των «πολλαπλάσιων του τέσσερα» μας απαλλάσσει από τετριμμένες ειδικές περιπτώσεις για ένα μερικώς γεμάτο κόμβο και εναπομείνασες εκχωρήσεις ανά επίπεδο.

Ο αλγόριθμος PACK μπορεί να γραφτεί σαν μια αναδρομική συνάρτηση , μοναδικό στοιχείο η DLIST, μία λίστα με αντικείμενα που θα στοιβαχτούν , NN είναι η συνάρτηση του κοντινότερου γείτονα που παίρνει δύο παραμέτρους NN(DLIST,I) επιστρέφει το στοιχείο μέσα στην DLIST που είναι χωρικά πιο κοντά στο στοιχείο I και έχει και την επιπλέον ιδιότητα της διαγραφής του στοιχείου αυτού από την λίστα. Η περιγραφή του αλγορίθμου είναι η εξής:

Αναδρομική Συνάρτηση PACK (DLIST)

{επιστρέφει ένα δείκτη στη ρίζα ενός στοιβαγμένου R-tree και έχει όλα τα δεδομένα στην DLIST}

Αν η DLIST περιέχει τέσσερα δεδομένα

Τοποθέτησε έναν δείκτη στο νέο R-tree κόμβο, N0 , return(N0)

Αλλιώς,

Ταξινόμησε τα στοιχεία της DLIST με κάποιο κριτήριο {για παράδειγμα αύξουσα χ-συντεταγμένη }

NLIST = () , {αρχικοποίηση της κενής λίστας }

Όσο η DLIST δεν είναι κενή

I1 = $1^{ο}$ στοιχείο από την DLST,

DLIST = tail (DLIST) , {διαγραφή του $1^{ου}$ στοιχείου}

I2= NN(DLIST,I1) ,

I3= NN(DLIST,I2),

I4= NN(DLIST,I1)

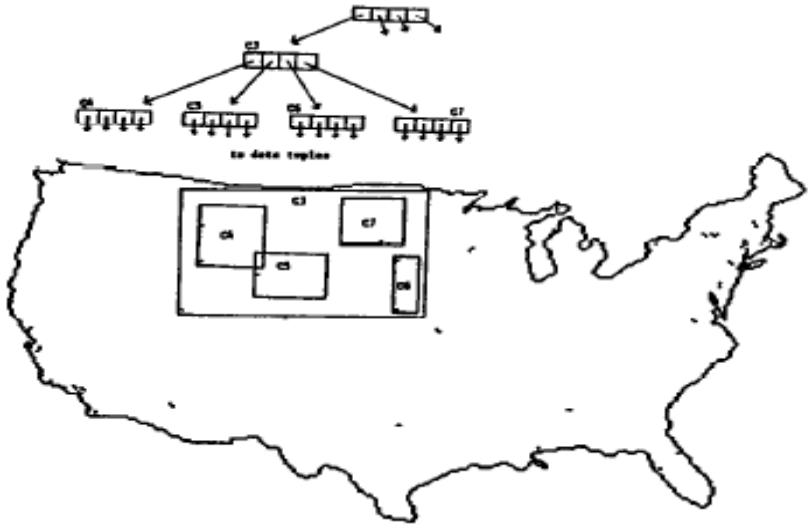
Τοποθέτησε έναν νέο R-tree κόμβο, N1

Δείκτες του N1 δείχνουν τα I1,I2,I3,I4

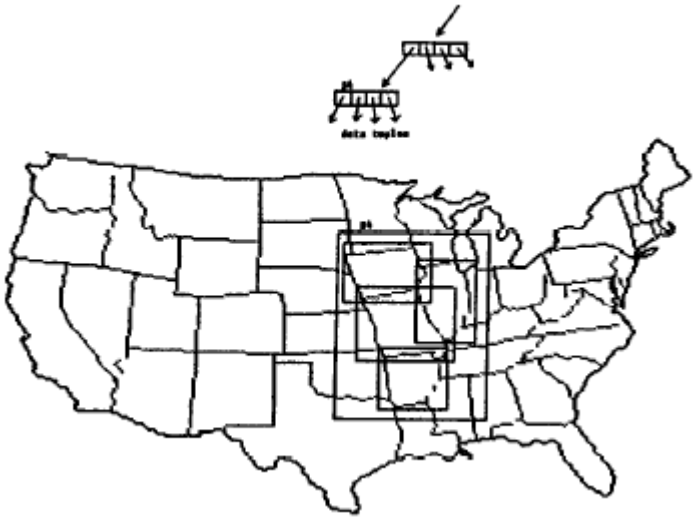
LIST = append (NLIST,N1) , {προσθήκη νέου κόμβου},

Return(PACK(NLIST))

Ένα απλό παράδειγμα που απεικονίζει τις αρχές του αλγορίθμου PACK είναι η εικόνα 4.Το οποίο δείχνει έναν χάρτη πόλεων στις ΗΠΑ, στην πρώτη κλήση η DLIST περιέχει την λίστα των πόλεων που είναι αποθηκευμένες σαν ζευγάρι συντεταγμένων [ίσως γεωγραφικό πλάτος και μήκος]. Εφόσον υπάρχουν πάνω από 4 ζευγάρια εκτελείται η περίπτωση else , και οι πόλεις ομαδοποιούνται ως προς τον κοντινότερο γείτονα (εικόνα 4α) .Η συνάρτηση καλείται αναδρομικά χρησιμοποιώντας την λίστα του φύλλου MBR σαν δεδομένο , και αυτή η διαδικασία συνεχίζει να εκτελείται προς τα πίσω μέχρι να φτάσουμε στη ρίζα. Όπως βλέπουμε , ο αλγόριθμος δεν διαχωρίζει τους κόμβους σε εσωτερικούς ή φύλλα , αλλά ένας τέτοιος διαχωρισμός είναι κρίσιμος και πρέπει να γίνεται στις πρακτικές εφαρμογές. Τα αποτελέσματα έδειξαν πως ο αλγόριθμος κατασκευάζει πολύ στενά στοιβαγμένα R-trees τα οποία είναι πρόσφορα για αποτελεσματική αναζήτηση.



Εικόνα 4



Εικόνα 4α

Εικόνα 5α

Εικόνα 5β

3. ΕΠΙΛΟΓΗ ΥΠΟΔΕΝΤΡΟΥ (ChooseSubTree)

Ορίζουμε το Ω σαν ένα στοιχείο που αντιπροσωπεύει ή ένα σημείο ή ένα τετράγωνο που πρόκειται να εισαχθεί στο R*-tree. Ξεκινάμε περνώντας στην ChooseSubTree έναν κόμβο N , αρχικά αυτός ο κόμβος είναι ο root (ρίζα) , και μετά στόχος της συνάρτησης είναι να αποφασίσει σε παιδί του N θα κατευθύνει το στοιχείο Ω. Ο αλγόριθμος αυτός είναι σημαντικός για την ποιότητα του δέντρου , ιδιαίτερα όταν ένα MBB στον κόμβο N χρειάζεται να μεγαλώσει για να καλύψει το Ω.

Πριν παρουσιάσουμε το νέο αλγόριθμο ChooseSubTree πρώτα θα εισάγουμε κάποιες σημαντικούς ορισμούς και θα κάνουμε μια ανασκόπηση του αλγορίθμου ChooseSubTree του R*-tree.

Πρόταση 1

Θέτουμε ως {E1,...,Εκ} τις εκχωρήσεις που υπάρχουν σε έναν κόμβο N και το Ri να αντιπροσωπεύει το MBB του Ei, όπου 1≤i ≤ k .

Υπάρχει f ανήκει {vol,perim} ,όπου vol και perim είναι ο όγκος και η περίμετρος ενός τετραγώνου , έννοιες που ορίστηκαν παραπάνω.

Η εκχώρηση του στοιχείου Ω στο Et αυξάνει :

- το f(Et) γίνεται Δft = f (MBB (Rt ∪ Ω)) - f(Rt) ,
- η κοινή επικάλυψη της Et και Ej , 1≤j≤k είναι

Δpvlpf t,j = f (MBB (Rt ∪ Ω) ∩ Rj) - f(Rt ∩ Rj),

η κοινή επικάλυψη της Et με τις εκχωρήσεις {Ep,....Eq}, 1≤p≤q≤k, θα είναι Δovlpf t,[p,q] = $\sum_{j=p,j\neq t}^{q}$ Δovlpf t, j

Επομένως ,πλέον μπορούμε παρακάτω να αναφερθούμε στην συνάρτηση ChooseSubTree του γνήσιου R*-tree (CSOrig).

Index CSOriq (Node N, Object Ω)

//επιστρέφει τον δείχτη της εκχώρησης E όπου το Ω θα εισαχθεί

//κ ο αριθμός όλων των εκχωρήσεων E στον κόμβο N

//r όριο

COV = { i| MBB (Ri ∩ Ω) = Ω , 1≤i≤k }
if (COV != 0)

return argmin { vol(Ri) | I ∈ COV} ;

//αν το COV είναι διάφορο του μηδενός τότε ένα MBB πρέπει να μεγαλώσει

if (οι εκχωρήσεις του N δεν αναφέρονται σε φύλλα)

return argmin1<=i<=k {Δvoli} ;

else

{ ταξινομούμε τις {E1,...Eκ} του κόμβου N σε αύξουσα σειρά ως προς Δvol

if (υπάρχει δείκτης j όπου Δovlp vol ,j[1,k] = 0)

return τον μικρότερο δείκτη j όπου Δovlp vol ,j[1,k] = 0 ;

else

//λαμβάνουμε υπόψη μόνο τις πρώτες r εκχωρήσεις

return argmini=1,...,r{Δovlp vol ,i[1,k] }

end CSOrig;

Πειράματα έδειξαν ότι η CSOrig συγκρινόμενη με άλλες μεθόδους ,αν και κάνει βελτιστοποίηση με επικάλυψη, είναι πολύ ακριβή στρατηγική και χρησιμοποιείται μόνο στα κατώτερα επίπεδα του καταλόγου. Η παράμετρος r που χρησιμοποιείται στον κώδικα μειώνει το κόστος CPU σε O(MlogM + Mr). Ο ορισμός του r είναι πολύ σημαντικός, διότι αν το r είναι πολύ μικρό , η πιθανότητα να χάσουμε την εκχώρηση εκείνη που θα μεγάλωνε λιγότερο για να επικαλύψει το νέο τετράγωνο , είναι

μεγαλύτερη. Αν το r ήταν από την άλλη , πολύ μεγάλο ,το κόστος της CPU γίνεται καθοριστικός παράγοντας. Τέλος , αν και στα πειράματα το r =32 ήταν κατάλληλα ορισμένο δεν μπορεί να χρησιμοποιηθεί σαν γενικός κανόνας το νούμερο αυτό. Η πυκνότητα των δεδομένων καθώς και ο αριθμός των διαστάσεων παίζουν σημαντικό ρόλο στον καθορισμό του r.

Επίσης, ένα άλλο σοβαρό πρόβλημα του αλγορίθμου είναι η αμιγώς βασισμένη στον όγκο στρατηγική βελτιστοποίησης που ακολουθείται. Η οποία, γίνεται αναποτελεσματική για πολυδιάστατα δεδομένα. Ισχύει ότι , τα bounding boxes με μηδενικό όγκο δεν είναι τόσο ασυνήθιστα όσο μπορεί να είναι σε υποχώρους με λιγότερο δηλαδή αριθμό διαστάσεων. Να τονίσουμε όμως ότι αυτό δεν είναι ελάττωμα μόνο των R*-trees αλλά παρατηρείται σε όλες τις δομές της οικογενείας των R-trees.

Ο αναθεωρημένος αλγόριθμος ChooseSubtree, ο CSRevised, αναδεικνύει τα παραπάνω αναφερόμενα προβλήματα. Αρχικά, μία απλή προσαρμοστική στρατηγική προτείνει τον υπολογισμό ενός υποψήφιου συνόλου από εκχωρήσεις τα οποία ελέγχονται για τον αν μεγαλώνουν ώστε να επικαλύψουν το νέο τετράγωνο. Όμοια με τον γνήσιο αλγόριθμο η χειρότερη επίδοση είναι ακόμη $O(M^2)$, αλλά η μέση επίδοση είναι καλύτερη στην πράξη. Λόγω της μείωσης αυτής, η βελτιστοποίηση με βάση την επικάλυψη εφαρμόζεται τώρα σε όλα τα επίπεδα του καταλόγου. Επίσης, ο αλγόριθμος παρουσιάζει και περιπτώσεις όπου τα κριτήρια με βάση τον όγκο που χρησιμοποιούνται στον γνήσιο αλγόριθμο γίνονται και αποτελεσματικά. Σε αυτές τις περιπτώσεις αντί για στρατηγική βασισμένη στον όγκο χρησιμοποιούμε στρατηγικές με βάση την περίμετρο.

Index CSRevised(Node N, Object Ω)

//επιστρέφει τον δείκτη της θέσης που θα εισαχθεί το Ω

//k = ο αριθμός των εκχωρήσεων στον κόμβο N

1. COV ← { i | MBB($R_i \cap \Omega$) = Ω, 1≤ i ≤ k };

   //ψάχνουμε να βρούμε ένα σύνολο που θα περιέχει όσα τετράγωνα καλύπτουν εξολοκλήρου το Ω , εφόσον η τομή των τετραγώνων αυτών με το Ω μας δίνει το Ω.

2. if (COV ≠ 0)

3. if (there is i ε COV with vol(Ri) = 0

4. return argmin { perim(Ri) | i ∈ COV};

5. else

6. return argmin { vol(Ri) | i ∈ COV};

   // αν το σύνολο COV δεν είναι κενό και αν υπάρχει τέτοια καταχώρηση i μέσα στο σύνολο αυτό που ο όγκος να είναι ίσο με μηδέν επέστρεψε εκείνο το τετράγωνο που έχει την ελάχιστη περίμετρο, διαφορετικά αν δεν υπάρχει στοιχείο με μηδενικό όγκο, επέστρεψε εκείνο με το μικρότερο όγκο.

7. // προσοχή! αν το COV είναι άδειο αναγκαστικά κάποιο από τα MBBs πρέπει να μεγαλώσει, αφού το Ω δεν θα καλύπτεται εξολοκλήρου από κάποιο τετράγωνο.

8. let E1,…..,Ek be the sequence of entries of N, sorted in ascending order of

9. their Δperim;

   //Δperim είναι η περίμετρος των τετραγώνων

10. if ($\Delta ovlp^{perim}$ 1,[1,k])= 0

    // αν η συνολική επικάλυψη του E1 με τα υπόλοιπα τετράγωνα είναι μηδέν

11. return 1;

12. else

13. p ← $argmax_i$ = 2,….k,{ $\Delta ovlp^{perim}$ 1,i | $\Delta ovlp^{perim}$ 1,i ≠ 0 }

14. // παίρνουμε υπόψη μας μόνο τις πρώτες p εκχωρήσεις

15. CAND←0; success ←false

16. if (there is an index i with vol(MBB(Ri ∪Ω)) = 0)

17. CheckComb(1,perim);

18. else

19. CheckComb(1,vol);

20.Procedure CheckComp (Index t, AggeragateFunction f)

21. CAND ← CAND ∪{t} ;

22. Δovlp[t]←0; // ξεκινάμε τον υπολογισμό του $\Delta ovlp^f t,[1,p]$

23.for(j=1,….p, j≠t)

24. $\Delta ovlp[t] \leftarrow \Delta ovlp[t] + \Delta ovlp^f t,j$;

25. if ($\Delta ovlp^f t,j \neq 0$ and j not in CAND)

26. CheckComp(j,f);

27. if (success) break;

28. end

29.end

30. if (Δovlp [t]= 0) //για παράδειγμα $\Delta ovlp^f t[1,p]=0$

31. c←t; success←true;

32. end CheckComp

//επιστροφή στην αρχική συνάρτηση

33.if(success)

34. return c;

35.else

36. //o Δovlp[i] , i ε CAND υπολογίστηκε ήδη στη CheckComp

37. return argmin {Δovlp[i] i ε CAND } ;

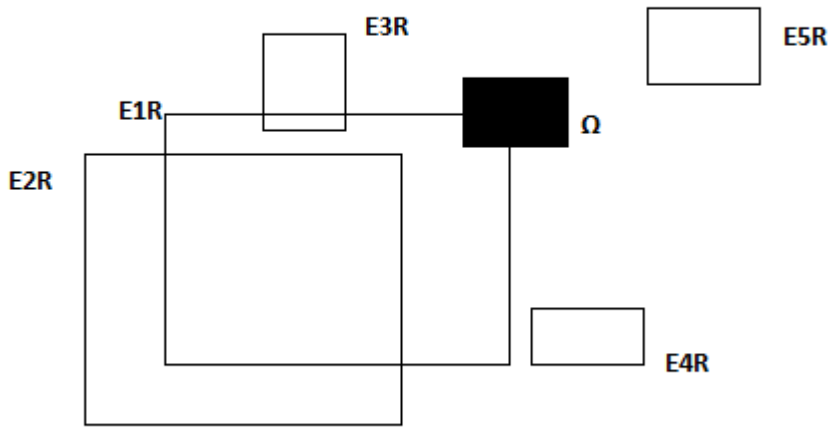
38.end CSRevised;

Ο παραπάνω αλγόριθμος ξεκινάει με τον υπολογισμό του COV, με το σύνολο των εκχωρήσεων που καλύπτουν το Ω. Εάν το COV δεν είναι άδειο, η καταχώρηση με τον ελάχιστο όγκο (περίμετρο) επιλέγεται από το COV.Διαφορετικά, το μέγεθος της περιμέτρου Δperim υπολογίζεται εκ νέου για κάθε καταχώρηση , με την λογική ότι το καινούριο στοιχείο εισέρχεται σε αυτή. Στη συνέχεια ταξινομούνται οι καταχωρήσεις με βάση το καινούριο Δperim.

Το Ω εκχωρείται στο E1 αν καμία από τις υπόλοιπες εκχωρήσεις δεν θα μεγαλώσουν (περίμετρος) από αυτή την εισαγωγή του Ω. Διαφορετικά, ο αλγόριθμος προσπαθεί να πετύχει μια βελτιστοποιημένη επικάλυψη με κάποιο εναλλακτικό τρόπο, όπου το κόστος υπολογισμού μειώνεται μέσω δύο φίλτρα. Το πρώτο φίλτρο θέτει ένα όριο και επικεντρώνεται στις πρώτες p εκχωρήσεις από την ταξινομημένη ακολουθία , όπου η Ep είναι η τελευταία εκχώρηση όπου η περίμετρος του τετραγώνου στην προσπάθεια της E1 να επικαλύψει το Ω αυξάνεται. Η CheckComp είναι βασισμένη στην στρατηγική βάσει όγκου (περιμέτρου), με depth-first διαπέραση στο δέντρο βρίσκει τον δείκτη t , $1 \le t \le p$, όπου η εισαγωγή του Ω στην Et δεν αυξάνει την επικάλυψη του Et με τις υπόλοιπες p-1 εκχωρήσεις. Κατά την διάρκεια της παραπάνω διαπέρασης , όλοι υπολογισμοί που γίνονται αποθηκεύονται σε ένα πίνακα CAND , ο οποίος χρησιμοποιείται σαν δεύτερο φίλτρο, σε περίπτωση που η CheckComp αποτύχει στην εύρεση κάποιας εκχώρησης χωρίς πρόσθετες επικαλύψεις. Σε αυτήν την περίπτωση, επιλέγεται ο μικρότερος δείκτης του CAND με την μικρότερη αύξηση στην επικάλυψη. Οι θέσεις q του CAND μπορεί να είναι το πολύ p, αλλά γενικά είναι μικρότερες.

Στην εικόνα 6 απεικονίζονται τα βήματα του αλγορίθμου CSRevised που αναλύσαμε παραπάνω. Η εκχώρηση του Ω στο E1 προκαλεί αύξηση στην επικαλύψει με τις E3 και E4. Το p=4 διότι το E4 είναι το τελευταίο με αύξηση στην επικάλυψη. Η κλήση της CheckComp βασισμένη στην στρατηγική βάσει όγκου εξετάζει τους δείκτες 1,3 και 4 αλλά αποτυγχάνει στο να βρει μία εκχώρηση που να μην αυξάνει την συνολική επικάλυψη αυτής με τις υπόλοιπες εκχωρήσεις. Στο τελικό στάδιο επιστρέφεται ο

δείκτης 3 μιας και με την εισαγωγή του Ω στην E3 είναι προκαλείται η μικρότερη επικάλυψη με τις υπόλοιπες. Σημειώνεται ότι ο δείκτης 2 δεν συμμετέχει καθώς δεν είναι καν στο πίνακα CAND.



Εικόνα 6

Θα κλείσουμε , με μία απλή ανάλυση κόστους. Το κόστος CPU του νέου αλγορίθμου είναι O(MlogM + qp) , όπου το q συμβολίζει το μέγεθος του πίνακα CAND.Ο πρώτος όρος περιγράφει το κόστος της ταξινόμησης και ο δεύτερος το κόστος της CheckComp. Παίρνουμε την χειρότερη επίδοση $O(M^2)$ όταν το q<=p<=M , αν και κατά μέσο όρο τα p,q είναι πολύ μικρότερα του του M. Το χειρότερο κόστος CPU είναι O(MlogM+Mr) , r<M (r=32) , αλλά η βελτιστοποίηση για την επικάλυψη εκτελείται μόνο στο χαμηλότερο επίπεδο του καταλόγου για παράδειγμα σε κάθε εκχώρηση.Ωστόσο, βελτιστοποιώντας αποτελεσματικότερα τα πεδία της CSRevised παίρνουμε μια ουσιαστική μείωση του μέσου κόστους της CPU για κάθε εισαγωγή.

4.ΔΙΑΙΡΕΣΗ ΚΟΜΒΩΝ (Split)

Για τον αλγόριθμο Split ακολουθούμε την ίδια λογική όπως και με τον γνήσιο R*-tree ,όπου προσπαθούμε να ελαχιστοποιήσουμε μια συνάρτηση (goal function). Η σχεδίαση ωστόσο διαφέρει καθώς στην goal function λαμβάνονται υπόψη, εκτός από τα κριτήρια επικάλυψης βάσει όγκου και περιμέτρου , και ο βαθμός της ισορροπίας που θέλουμε κατά τη διαίρεση (ο τρόπος δηλαδή που μοιράζονται τα στοιχεία κατά τη διαίρεση). Όταν τα δεδομένα εισέρχονται τυχαία , γενικά προτιμούμε διαιρέσεις όπου τα στοιχεία μοιράζονται εξίσου και στους δύο κόμβους. Για μη τυχαίες εισαγωγές ο αλγόριθμος προτιμά μία όχι τόσο ισορροπημένη διαίρεση μεταξύ των κόμβων παρακολουθώντας πόσο ανόμοια μεγαλώνουν τα MBBs.

Ο νέος μας αλγόριθμος ακολουθεί μια στρατηγική που είναι υποκατάστατο της επανεισαγωγής ενός στοιχείου, η οποία θεωρείται ότι συνεισφέρει ευεργετικά στην βελτίωση της επίδοσης της αναζήτησης που έχουμε στα R*-trees.Γενικά, η επανεισαγωγή είναι μια αποτελεσματική λύση για να εμποδίσει την πτώση της επίδοσης της αναζήτησης για μην τυχαίες εισαγωγές.

Στις παρακάτω παραγράφους περιγράφουμε με λεπτομέρειες για τον καινούριο αυτό αλγόριθμο. Αρχικά παρουσιάζουμε κάποιες σημαντικές προτάσεις, μετά κάνουμε μια μικρή ανασκόπηση στον αλγόριθμο split του R* -tree και τέλος παρουσιάζουμε την νέα goal function και την εφαρμογή της στους κόμβους.

4.1 Ανασκόπηση του Split αλγορίθμου του R*-tree.

Η στρατηγική της διαίρεσης ενός κόμβου σε δύο βασίζεται στην ταξινόμηση των τετραγώνων δίνοντας προσοχή στις προβολές αυτών σε όλες τις διαστάσεις, και αποφασίζεται για την κάθε ταξινομημένη ακολουθία σε πιο i-οστό ($m \le i \le M+1-m$) στοιχείο θα γίνει η διαίρεση, όπου οι πρώτες i εκχωρήσεις μένουν στον παλιό κόμβο και οι M+1-i καταχωρούνται στον καινούριο κόμβο. Υπάρχουν δύο διαφορετικοί τρόποι να διατάξουμε τα τετράγωνα. Ο πρώτος είναι ως προς το ελάχιστο σύνορο σε κάθε διάσταση και ο άλλος ως προς το μέγιστο σύνορο, γενικά για dim διαστάσεις υπάρχουν 2dim τρόποι. Για χάρη σαφήνειας προχωρούμε στην ανάλυση

του αλγορίθμου παίρνοντας ένα από τους δύο διαφορετικούς τρόπους διάταξης, έστω αυτή που κάνει χρήση του μικρότερου ορίου.

Πρόταση 2.

Έχουμε το $SC_{i,d} = (F_{i,d}, S_{i,d})$, όπου είναι ο i-στός υποψήφιος , με i=m,…,M+1-m , όπου το d συμβολίζει την διάταξη της ακολουθίας στην d διάσταση , το $F_{i,d}$ αντιπροσωπεύει τις πρώτες i εκχωρήσεις και $S_{i,d}$ τις υπόλοιπες M+1-i.

- Η περίμετρος και ο όγκος του υποψήφιου $SC_{i,d}$ δίνονται από τον εξής τύπο

  $f(SC_{i,d)} = f(MBB(F_{i,d})) + f(MBB(S_{i,d}))$, με $f \in (perim, vol)$
- Η επικάλυψη ενός υποψήφιου είναι $ovlp^f(SC_{i,d}) = f(MBB(F_{i,d})) \cap f(MBB(S_{i,d}))$
- Αν $f(MBB(F_{i,d})) \cap f(MBB(S_{i,d})) = 0$,το $SC_{i,d}$ ορίζεται ως overlap-free του split στην d-οστή διάσταση.

Η διαίρεση ενός κόμβου αντιστοιχεί στον υπολογισμό της βέλτιστης διαίρεσης του υποψήφιου $SC_{u,a}$, όπου το u είναι ο δείκτης split στην εκχώρηση του άξονα α ,ως προς τον οποία γίνεται το split.Ο γνήσιος αλγόριθμος για το split του R*-tree ενεργεί σε δύο βήματα:

1. // υπολογισμός του άξονα α που θα γίνει το split α= argmin $_{1\le d\le dim}$ $\{\sum_{i=m}^{M+1-m} \mathrm{perim}(SCi, d)\}$
2. if (αν υπάρχουν overlap-free split υποψήφιοι στον split άξονα α)

   //διάλεξε τον υποψήφιο με την μικρότερη περίμετρο

   u = argmin $_{m\le i\le M+1-m}$ {perim $(SC_{i,a})$ | $(SC_{i,a})$ o.f. }

   //το γνήσιο R*tree χρησιμοποιεί σαν κριτήριο τον όγκο

   else

   //διάλεξε τον υποψήφιο με την μικρότερη επικάλυψη

   u = argmin $_{m\le i\le M+1-m}$ {$ovlp^{vol}(SC_{i,a})$ }

Ο αναθεωρημένος αλγόριθμος διαφέρει στο δεύτερο βήμα, όπου πλέον γίνεται διαχωρισμός των κόμβων σε φύλλα και εσωτερικούς κόμβους, όσον αφορά την επιλογή του άξονα α. Για ένα φύλλο , η διαίρεση γίνεται όπως περιγράφεται παραπάνω στο βήμα 1. Σε περίπτωση όμως, που η διαίρεση γίνεται σε έναν εσωτερικό κόμβο, το βήμα 1 δεν λαμβάνεται καθόλου υπόψη και στο βήμα 2 γίνονται οι υπολογισμοί για όλες τις διαστάσεις. Πλέον θα έχουμε να επιλέξουμε ανάμεσα σε dim*(M+2-2m) υποψήφιους δείκτες τον ένα που ελαχιστοποιεί την αντίστοιχη συνάρτηση (goal function) του δεύτερου βήματος. Έμμεσα μέσα από την παραπάνω επιλογή του υποψηφίου επιλέγεται και ο άξονας α. Τέλος, ο αναθεωρημένος αλγόριθμος εναλλάσσει την στρατηγική που βασίζεται στον όγκο σε στρατηγική βασισμένη στην περίμετρο (τελευταία γραμμή του παραπάνω ψευτοκώδικα) σε περίπτωση που επιτευχθεί vol(MBB ($F_{m,a}$)) = 0 ή vol(MBB($S_{M+1-m,a}$)) = 0 Η μη επίτευξη του παραπάνω σημαίνει ότι ο όγκος σε κάθε διαίρεση είναι μεγαλύτερη του 0.

4.2 Σχεδίαση της συνάρτησης για την διαίρεση

Ο αναθεωρημένος αλγόριθμος χρησιμοποιεί μια νέα συνάρτηση w:[m,...,M+1-m] -> R η οποία επιστρέφει μία πραγματική τιμή για κάθε υποψήφιο για διαίρεση Sci,α. Η ελάχιστη τιμή της συνάρτησης αυτής μας δείχνει και που θα γίνει το split. Η βασική ιδέα της συνάρτησης w είναι να ελαχιστοποιήσει μια σύνθεση , της γνήσιας συνάρτησης wg του αλγορίθμου R*-tree (βασισμένη στα κριτήρια όγκος, περίμετρος και επικάλυψη) και μιας συνάρτησης wf την οποία θα αναλύσουμε στο παρακάτω κεφάλαιο και η οποία εξυπηρετεί στο να προσαρμόσουμε το split τοπικά και να λάβουμε υπόψη την ισορροπία του δέντρου, μια στρατηγική που δεν λαμβάνεται υπόψη στις προηγούμενες δομές. Η σύνθετη αυτή συνάρτηση w θα αποτελεί λοιπόν αποτέλεσμα των 2 άλλων συναρτήσεων της wg και της wf. Η ανάλυση βρίσκεται στο υποκεφάλαιο 4.2.4

4.2.1 Σχεδίαση της wf (weighting function)

Η wf έχει σχεδιαστεί για να δείχνει την αλλαγή στο MBB του υπερχειλισμένου κόμβου N από την στιγμή που ο N δημιουργήθηκε μέχρι τη στιγμή που συμβαίνει το split. Για να απλουστεύσουμε την διαδικασία ορίζουμε την wf να ανήκει στο συνεχές διάστημα [-1,1].Το κέντρο x του γνήσιου MBB (Οbox) ορίζεται την στιγμή που ο κόμβος N δημιουργείται . Όταν το N πρέπει αργότερα να διαιρεθεί κρατάμε

και το κέντρο του υπερχειλισμένου κόμβου για τον υπολογισμό της παραμέτρου asym στο [-1,1].

asym = [2 (xα(MBB(N)) – xα(OBox)] / λα(MBB(N))

Αν asym = 0 διαιρούμε τον N στη μέση ,με μεγαλύτερη πιθανότητα , διότι ένα μη ισορροπημένο δέντρο είναι λιγότερο πιθανό να συμβεί. Σε περίπτωση που ισχύει το δεύτερο θεωρούμε το μ = (1- 2m / (M+1)) * asym σαν το σημείο στο [-1, 1] που λαμβάνει την μεγαλύτερη πιθανότητα.

Η συνάρτηση wf:[-1,1] -> [0,1] σχεδιάζεται με τα ακόλουθα κριτήρια:

1.wf(μ) =1

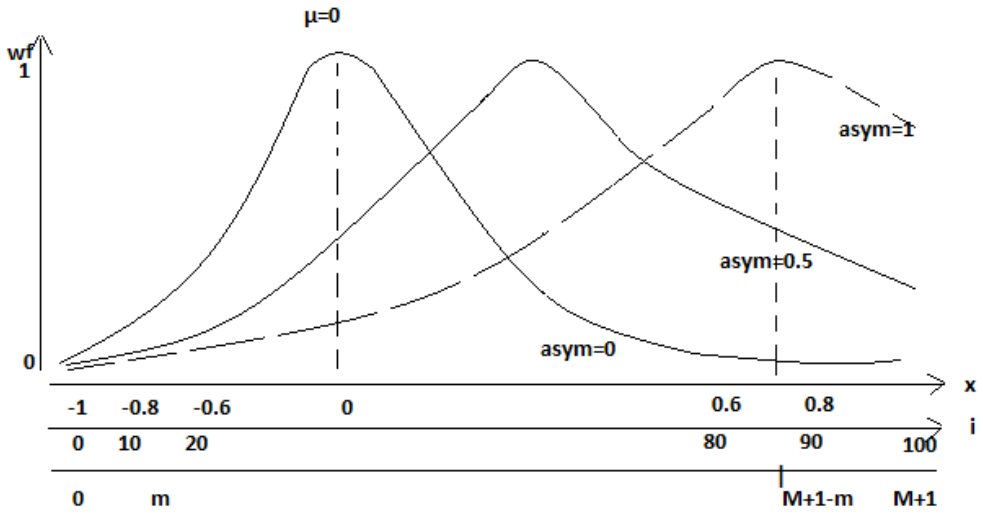
2.αυξάνεται και μειώνεται μονοτονικά μόνο μέσα στο εύρος τιμών [-1,μ] και [μ,1]

3. wf(-1) = 0 αν asym >=0 και wf(1) = 0 αν asym<=0

Για να δούμε γραφικά τι ισχύει παίρνουμε την συνάρτηση του Gauss $\exp(- ((x-\mu)/\sigma)^2)$ που μας εξασφαλίζει τα καλύτερα αποτελέσματα , και όπου σ = s (1 + |μ|) και s είναι μία σταθερά που ελέγχει το σχήμα της καμπύλης. Για s να τείνει στο 0 η wf παράγει μια καμπύλη με στενή κορυφή, για s μεγαλύτερο από 3 η καμπύλη μοιάζει με την παραβολή $1 – x^2$.

Μέχρι εδώ η συνάρτηση wf πληρεί την πρόταση 2. Επιπλέον, wf(μ) = max(wf) και wf(-1)=c αν asym>=0 και wf(1) = c αν asym<=0 (το c εξαρτάται μόνο από το s). Το τελευταίο είναι λόγω το υπολογισμού που δόθηκε για το σ. Για να ικανοποιήσουμε τα κριτήρια 1 και 3, η συνάρτηση Gauss απαιτεί μια ολίσθηση κατά $y1=\exp(-1 / s^2)$ και μία κλιμάκωση με τον συντελεστή ys = 1 / (1-y1). Τέλος, μετασχηματίζουμε με $x_i=2i / (M+1) -1$ για να καθορίσουμε κάθε δείκτη i , 0<=i<=M+1 , σε μια τιμή μεταξύ [-1,1] , το οποίο θα χρησιμοποιηθεί σαν είσοδος στην wf. Συνεπώς, μπορούμε να θεωρήσουμε την wf ως μια συνάρτηση μια διακεκριμένο πεδίο ορισμού {0,.....,M+1}. Γενικά, η wf αποτελεί μια τροποποιημένη συνάρτηση Gauss που δίνεται από τον τύπο:

$wf(i) = wf(i , asym , s , M , m) = y_s * \{ \exp [- ((x_i-\mu) / \sigma)^2] – y1 \}$



Εικόνα 7 (wf συναρτήσει του x,i για asym = 0 , 0.5, 1 και s=0.5 M=100 m=15)

4.2.2 Το κίνητρο της δημιουργίας μιας wf

Μετά την περιγραφή της wf παραπάνω θα εξηγήσουμε το κίνητρο της σχεδίασης αυτής αναφερόμενοι στην γραφική παράσταση του σχήματος 2.

Η wf είναι κατασκευασμένη με τέτοιο τρόπο ώστε το μ μετακινηθεί προς τα δεξιά , ο αριστερός root ορίζεται 0, όσο ο δεξιός root μετακινείται προς τα δεξιά. Η μέγιστη τιμή για τον δεξιό root της wf είναι (M+1-m) για asym=1. Το διάστημα ανάμεσα στους 2 roots της wf αντιστοιχεί στον άξονα split ενός εικονικού κόμβου. Αν το μ>0, αυτός ο κόμβος δεν περιέχει μόνο τα M+1 τετράγωνα του N , αλλά και τα τετράγωνα δεξιά του N και αναμένεται να εισαχθούν στο N στο κοντινό μέλλον. Από την πλευρά του εικονικού αυτού κόμβου το μ είναι πάντα στο κέντρο , αντίθετα στον κόμβο N μπορεί να είναι τοποθετημένο ασυμετρικά.

4.2.3 Κόστος της wf

Το κόστος CPU της wf είναι χαμηλό. Τα περισσότερα μέρη της συνάρτησης υπολογίζονται μόνο μια φορά κατά τη διάρκεια του split, ή είναι σταθερά για όλο τον κύκλο ζωής του δέντρου. Μόνο οι παράμετροι x_i και w_f υπολογίζονται για κάθε υποψήφιο που είναι για διαίρεση δείκτη i , για παράδειγμα 2(M+2-2m) φορές (για

δύο ταξινομήσεις) για την διαίρεση ενός φύλλου και 2dim(M+2-2m) φορές για τους εσωτερικούς κόμβους. Να θυμηθούμε ότι για τους εσωτερικούς κόμβους δεν υπάρχει επανυπολογισμός για τον άξονα ως προς τον οποίο θα γίνει η διαίρεση. Επίσης, για τον υπολογισμό της wf μόνο μία εκτίμηση της εκθετικής συνάρτησης είναι απαραίτητη.

Υπάρχει και μια επιπρόσθετη αποθήκευση στην μνήμη τον dim-διάστατων σημείων για κάθε κόμβο, αυτό μειώνει την χωρητικότητα του κόμβου σε μία το πολύ εκχώρηση.

4.2.4 Εφαρμογή της wf

Σε αυτό το κεφάλαιο παρουσιάζεται ο συνδυασμός της συνάρτησης wf με την συνάρτηση wg του R*-tree. Πιο πάνω διακρίναμε δύο διαφορετικές καταστάσεις. Αν υπάρχει τουλάχιστον ένας overlap-free υποψήφιος για διαίρεση στον δεδομένο άξονα ,πρέπει να περιορίσουμε την μελέτη στο ταξινομημένο σύνολο για το οποίο ισχύει η ιδιότητα αυτή. Διαφορετικά, θα πρέπει να εξεταστεί ολόκληρη η ταξινομημένη ακολουθία. Και στις 2 περιπτώσεις, η συνάρτηση wg (goal function) θα πρέπει να επιστρέφει μια καλή τιμή για κάθε υποψήφιο για διαίρεση.

Για να δημιουργηθεί μια μοναδική goal function w , έγιναν πολλά πειράματα παίρνοντας κάθε φορά διάφορους συνδυασμούς των wf και wg , ώσπου τελικά να πάρουμε την μορφή που θα δείξουμε στο τέλος αυτού του κεφαλαίου.

Η τιμή που θα πάρει η w απαιτεί ότι κάθε συνάρτηση παραδίδει μια τιμή σε ένα ορισμένο διάστημα. Η wf έχει εύρος [0,1] και έτσι ικανοποιεί το κριτήριο. Το λήμμα μας δείχνει πως και goal function του R*-tree μπορεί να οριστεί σε ένα διάστημα.

Λήμμα 1:

Η goal function του R*-tree υπολογίζει μία τιμή εντός κάποιου ορίου.

Απόδειξη:

Η goal function του R*-tree ελαχιστοποιεί την επικάλυψη. Αν η επικάλυψη είναι μηδέν , χρησιμοποιείται η περίμετρος ως εναλλακτικό κριτήριο. Το ελάχιστο και για

τα δύο είναι βέβαια το μηδέν. Τ άνω όριο για το κριτήριο της επικάλυψη μπορεί να υπολογιστεί με τον παρακάτω τρόπο:

Έστω N ο υπερχειλισμένος κόμβος ,θέτουμε ως R_N=MBB(N) . Η μέγιστη δυνατή επικάλυψη για έναν υποψήφιο για διαίρεση SC_i= (F_i , S_i) είναι

$ovlp^f max = max (f (MBB(F_i) \cap MBB(S_i))) , f \in \{perim,vol \}$

$= f (R_N \cap R_N) = f (R_N)$

Υποθέτουμε ότι υπάρχει overlap-free υποψήφια διαίρεση . Έχουμε $\lambda_{min} (R_N) = min \{ \lambda1(R_N) , ….., \lambda_{dim}(R_N) \}$, όπου κρατάει το ελάχιστο μήκος του τετραγώνου R εξετάζοντας τις προβολές του σε όλες τις διαστάσεις.

Η μέγιστη έτσι πιθανή περίμετρος της υποψήφιας διαίρεσης SC_i είναι $perim_{max} = 2(\sum_{d=1}^{dim} \lambda d(RN)) - \lambda_{min}(R_N)$, που είναι το άθροισμα των περιμέτρων των δύο τετραγώνων, το οποίο λαμβάνεται με μια τομή κάθετα στην μικρότερη ακμή του R_N.

Βάσει του λήμματος 1, η goal function του R*-tree μπορεί να μετατραπεί στην παρακάτω μορφή:

$$wg(i) = \begin{cases} perim(SC_i) - perim_{max} & \text{αν } ovlp^f(SC_i) = 0 \\ ovlp^f(SC_i) & \text{διαφορετικά} \end{cases}$$

Στην overlap-free περίπτωση , δεν γίνεται έλεγχος για τον υποψήφιο με την μικρότερη περίμετρο , αλλά για αυτόν με την μεγαλύτερη “free perimeter”. Η συνάρτηση wg τοποθετεί κάθε δείκτη i στο διάστημα $[- perim_{max} , ovlp^f_{max}]$, όπου η πρώτη επιλογή επιστρέφει αρνητικές τιμές και η δεύτερη θετικές. Η συνάρτηση κατοπτρίζει το δεύτερο βήμα της διαίρεσης του αλγορίθμου του R*-tree του οποίου η στρατηγική για την επιλογή του κατάλληλου υποψηφίου για διαίρεση μπορεί να πάρει την εξής μορφή: u_a <- argmin $_{m \le i \le M+1-m}$ {wg(i) }.

Τώρα πλέον μπορούμε να διατυπώσουμε την σύνθετη συνάρτηση

$$w(i) = \begin{cases} wg(i) * wf(i) & \text{αν } ovlp^f(SC_i) = 0 \\ wg(i) / wf(i) & \text{διαφορετικά} \end{cases}$$

Να σημειωθεί ότι wf(i) >0 όσο $0<m\leq i\leq M+1-m<M+1$. Η συνάρτηση w τοποθετεί κάθε δείκτη i μέσα στο διάστημα $[- perim_{max} , \infty)$. Ως wg , η w επιστρέφει μηδέν στην περίπτωση που το SC_i είναι overlap-free , και $MBB(F_i) \cup MBB(S_i)=MBB(N)$

Το δεύτερο βήμα του καινούριου αλγορίθμου διαίρεσης είναι:

$u_a \leftarrow \text{argmin}_{m\leq i\leq M+1-m} \{w(i)\}$.

4.3 Ρυθμίσεις παραμέτρων

Στη δομή R*-tree για τον αλγόριθμο Split, μόνο μια σταθερά πρέπει να προσδιοριστεί, αυτή που μας δείχνει την ελάχιστη κατοχή σε στοιχεία m που μπορεί να έχει ένας κόμβος, όπου $m \leq \lceil M/2 \rceil$.Κατά τη διάρκεια των πειραμάτων του νέου αλγορίθμου που παρουσιάστηκε παραπάνω το m μειώθηκε στο 20% του M, το οποίο αποδίδει την καλύτερη εκτέλεση της αναζήτησης στα πειράματά μας. Επίσης, αυτή η τιμή αυξάνει τις πιθανότητες να βρούμε ένα υποψήφιο για διαίρεση overlap-free, αλλά αυξάνει και το κόστος για την διαίρεση ενός κόμβου, μιας και περισσότεροι υποψήφιοι εξετάζονται. Μία ακόμη σταθερά που πρέπει να οριστεί καταλλήλως στον αναθεωρημένο αλγόριθμο είναι η s . Τα πειράματα έδειξαν πως το s μπορεί να κυμαίνεται στο εύρος [0.47, ..., 0.54] για να επιτύχουμε την καλύτερη μέση επίδοση της αναζήτησης. Τελικά επιλογή για την τιμή του s ορίστηκε το 0.5.

5. ΠΕΙΡΑΜΑΤΑ

Στο κεφάλαιο αυτό θα δοθεί μια λεπτομερής περιγραφή των πειραμάτων που κάναμε.

Αρχικά θα παρουσιάσουμε τη φύση των τετραγώνων που έλαβαν χώρα στην έρευνα , τα οποία σύνολα των τετραγώνων που δημιουργήσαμε είναι διαθέσιμα στο http://www.mathematic.uni-marburg.de/~seeger/rrstar/,2009. Έπειτα θα αναφέρουμε την διαδικασία εγκατάστασης των πειραμάτων και τέλος στα αποτελέσματα αυτών.

5.1 Διανομές των δεδομένων

Τα πειράματα περιέχουν 28 σύνολα δεδομένων , 21 των οποίων είναι τεχνητά δημιουργημένα και τα υπόλοιπα αποτελούν σενάρια από πραγματικές εφαρμογές. Τα τεχνητά σύνολα δεδομένων ανήκουν σε εφτά ομάδες διανομής και κάθε ομάδα περιέχει δεδομένα 2D, 3D και 9D . Τα πραγματικά σύνολα ανταποκρίνονται σε 2D,3D,5D,9D,16D,22D και 26D διανομές.

5.1.1Τεχνητές διανομές

Οι εφτά διανομές που επιλέχτηκαν μέσα από ένα μεγάλο αριθμό διανομών δεν επιλέχτηκαν τυχαία , καθώς χαρακτηρίζονται από την ικανότητά τους να αποκαλύπτουν συχνές αδυναμίες των R-trees. Η διανομή Uniform (ομοιόμορφη κατανομή δεδομένων) δεν αποτελεί εξαίρεση.

Τα τεχνητά αυτά σύνολα δημιουργήθηκαν από αλγόριθμους που παράγουν συχνά διανομές με αυθαίρετες διαστάσεις. Παρόλ' αυτά τα θεμελιώδη χαρακτηριστικά των διανομών δεν επηρεάζονται από την διαφορετική διάσταση. Τα σύνολα των δεδομένων που παράγονται περιέχουν τουλάχιστον 1 εκατομμύριο στοιχεία , ο ακριβής αριθμός έχει να κάνει με τον κάθε αλγόριθμο. Παρουσιάζουμε παρακάτω μια σύντομη περιγραφή των τεχνητών αυτών διανομών.

Αρχικά πριν μπούμε στις λεπτομέρειες των διανομών ορίζουμε τους όρους dithering vector και density. Ένας τέτοιος πίνακας D=(d1, ...ddim) όπου το di έχει

τυχαία εκτείνεται στο [-ci,+ci] , όπου |ci| είναι η ένταση ίδια για όλα τα I.Ένας πίνακας X ταλαντεύεται απλά προσθέτοντας ένα πίνακα D. Η πυκνότητα (density) είναι το άθροισμα των όγκων ενός σετ από τετράγωνα, διαιρεμένο με τον όγκο του αντίστοιχου MBB

-*Absolute* περιέχει τετράγωνα που απέχουν μεταξύ τους ίση απόσταση και είναι ίδιου μεγέθους (density 0.7) ελαφρώς ταλαντεύεται ως προς τη θέση και την επέκταση. Διάταξη: σε σειρά και από αριστερά προς δεξιά.

-*Bit* είναι μια διανομή από σημεία με τυχαία διάταξη.

-Diagonal προέρχεται από ένας σύνολο τετραγώνων , οργανωμένα με ίση απόσταση από την κύρια διαγώνιο του χώρου δεδομένων. Οι θέσεις και οι εκτάσεις ταλαντεύονται και η διάταξη γίνεται κατά μήκος της διαγωνίου.

-*Parsel* προέρχεται από μια διανομή τετραγώνου , όπου αναδρομικά διαιρούμε τον χώρο δεδομένων σε δύο μέρη για τυχαίο μέγεθος, ο άξονας που γίνεται κάθε φορά η διαίρεση εναλλάσσεται. Από την ταλάντωση στις θέσεις , μεγάλα τετράγωνα καλύπτουν τα μικρότερα. Η πυκνότητα είναι ίση με 0.5 και η διάταξη ακολουθεί μία z-διάταξη στα κέντρα των τετραγώνων.

-*P-edges* μια διανομή από μια λεπτή γραμμή σημείων και τα παράγωγά της προέρχονται απ'αυτά της Parsel , η διάταξη είναι τυχαία.

-*P-haze* είναι μια διανομή από ένα σύμπλεγμα σημείων σε ελλειπτικό σχήμα και σχετίζεται πολύ με την Parsel. Διάταξη: η διανομή μεγαλώνει μαζί με πολλαπλά σημεία.

-*Uniform* είναι ομοιόμορφη κατανομή ενός συνόλου σημείων και η διάταξη είναι τυχαία.

5.1.2 Πραγματικές διανομές

Στο υποκεφάλαιο αυτό παρουσιάζουμε μία σύντομη περιγραφή των πραγματικών συνόλων δεδομένων που χρησιμοποιήσαμε στα πειράματά μας.

-*CaliforniaStreets* είναι μια διδιάστατη διανομή από 1,888,012 τετράγωνα και σημεία που παράχθηκαν από τους δρόμους στην Καλιφόρνια [18]. Διάταξη: υποπεριοχές των 20,000 στοιχείων και είναι παρόμοια με την διάταξη σε σειρά.

-BioData είναι μια τρισδιάστατη διανομή των 11,958,999 σημείων που εξάγονται από βιολογικά δεδομένα και η διάταξη είναι τυχαία.

-*CoverType* είναι μια πενταδιάστατη διανομή των 581,012 σημείων από ένα αρχείο του US Forest Service [19] και η διάταξη είναι όχι τυχαία και αποτελείται από ευδιάκριτα κατανεμημένα τα στοιχεία.

-ColorMoments είναι μια εννιαδιάστατη διανομή των 68,040 σημείων από χαρακτηριστικά εικόνας μιας συλλογής Corel [7] και η διάταξη των σημείων είναι τυχαία.

-*Fourier* είναι μια δεκαεξαδιάστατη διανομή από 1,312,173 σημεία από συντελεστές Fourier από CAD δεδομένα. Αυτά τα δεδομένα χρησιμοποιούνται στο [17] και η διάταξη που φαίνεται να ακολουθούν σε μια διδιάστατη άποψη είναι καμπύλη.

-*ActivityReport* είναι μια εικοσιδιδιάστατη διανομή από 500,000 σημεία που αναπαριστούν στιγμιότυπα από μια αναφορά δραστηριότητας που δημιουργείται από ένας Unix πυρήνα και η διάταξη είναι χρονολογική.

-*OccurenceCount* είναι μια εικοσιεξαδική διανομή από 427,060 σημεία καθένα από τα οποία αναπαριστά την ύπαρξη των γραμμάτων ‘α’.....‘z’ σε ένα αρχείο κειμένου. Ο αριθμός των διπλότυπων είναι πολύ συχνός σε αυτή την διανομή και η διάταξη δεν είναι τυχαία.

5.2 Αποτελέσματα των πειραμάτων

Τα αποτελέσματα που διεξάγαμε είναι μια σειρά από τρεξίματα του αναθεωρημένου R*-tree αλγορίθμου σε λειτουργικό OpenSuse και διανομή 11.4.

Είναι σημαντικό εδώ να αναφερθούμε στα χαρακτηριστικά του υπολογιστή που έγιναν τα πειράματα και να τονίσουμε ότι λόγω του ότι έγιναν σε 32-bito σύστημα* και ότι ο κώδικας δεν προγραμματίστηκε με την λογική της δημιουργίας σελίδων οι οποίες θα επαναφέρονται από τον σκληρό δίσκο στην μνήμη και αντιστρόφως για να εξασφαλιστεί κάποια οικονομία χώρου, μας περιόρισε στο να πάρουμε μόνο ένα ποσοστό τετραγώνων από κάθε διανομή των δεδομένων. Έτσι από κάθε τεχνητή διανομή περιοριστήκαμε στα 100.000 πρώτα τετράγωνα καθώς από κει και πάνω είχαμε overflow.

<u>χαρακτηριστικά μηχανήματος</u>

Intel Core 2 CPU 1.67GHz

RAM 3,00 GB

32-bit Operating System

*Το συμπέρασμα αυτό έγινε ύστερα και από τρεξίματα των πειραμάτων σε υπολογιστή (32-bit)[8GB DDR3 Kingston 1333MHz ,Asus M4A77TD Pro μητρική

Corsair 128GB SSD σκληρός δίσκος] με περισσότερη μνήμη που όμως το πρόγραμμα δεν χρησιμοποίησε.

Παρακάτω ακολουθεί πίνακας που μας δείχνει:

- πόσες διαιρέσεις έγιναν στο δέντρο για την εισαγωγή των 100.000 τετραγώνων για την κάθε διανομή

- πόσους κόμβους περιέχει το δέντρο μετά την εισαγωγή των τετραγώνων

- πόσοι συνολικά κόμβοι διαπεράστηκαν για την εισαγωγή και των 100.000 τετραγώνων.

- συνολικός χρόνος που έτρεξε το πρόγραμμα για την εισαγωγή των τετραγώνων αυτών.

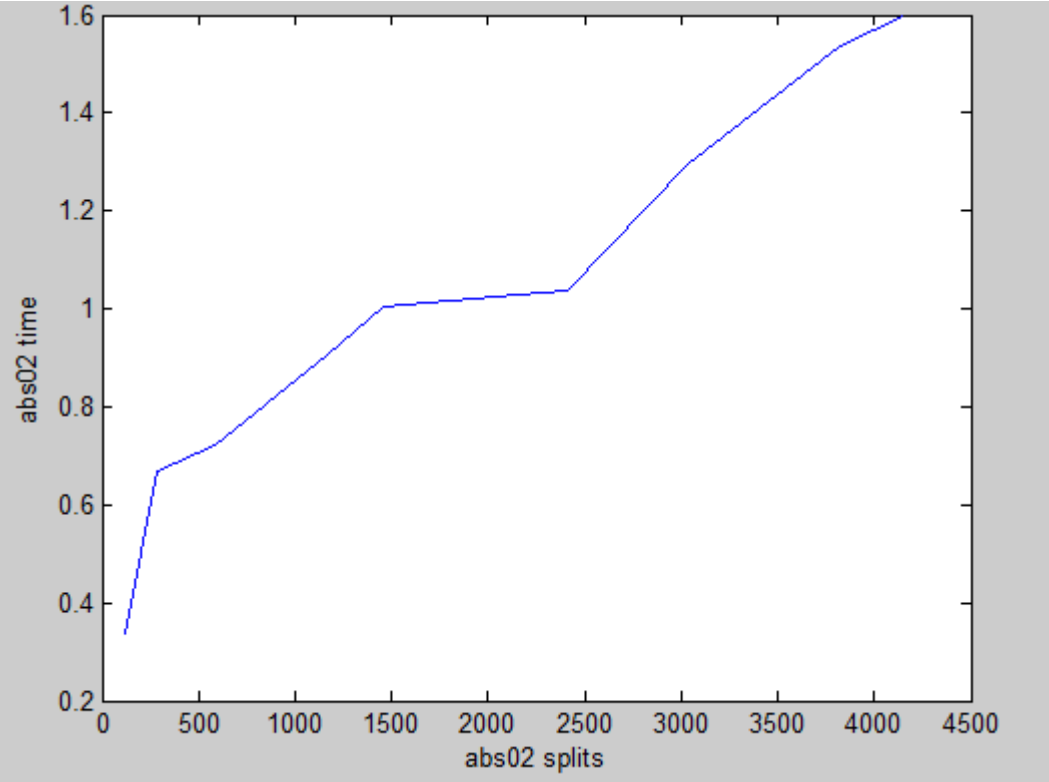
διανομή #διαστάσεων	abs 02	abs 03	abs 09	bit 02	bit 03	bit 09	dia 02	dia 03	dia 09
#διαιρέσεων	4153	4157	4048	4174	4194	4194	4190	4198	4207
#κόμβων	4632	4636	4202	4636	4678	4678	4674	4683	4692
#διαπεράσεων	95846	95932	97200	95932	96804	96805	96720	96894	97092
χρόνος	1.59744	1.598881	1.620007	1.618210	1.613402	1.713485	1.605431	1.614850	1.618210

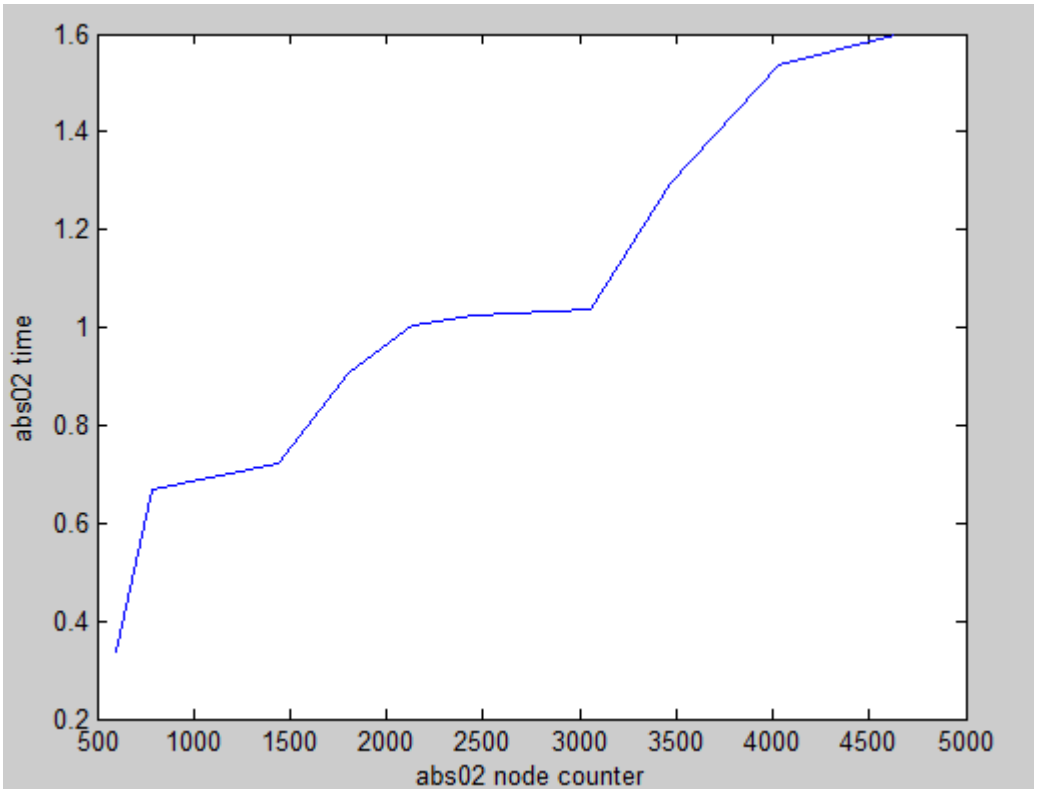
διανομή #διαστάσεων	par 02	par 03	par 09	ped 02	ped 03	ped 09	pha 02	pha 03	pha 09
#διαιρέσεων	4355	4356	4360	4297	4278	4276	4258	4288	4200
# κόμβων	4857	4861	4863	4792	4782	4769	4750	4783	4684
#διαπεράσεων	100502	100513	100629	99164	98944	98681	98280	98966	96929
χρόνος	1.675041	1.675047	1.677156	1.652749	1.649078	1.644698	1.638014	1.649447	1.615495

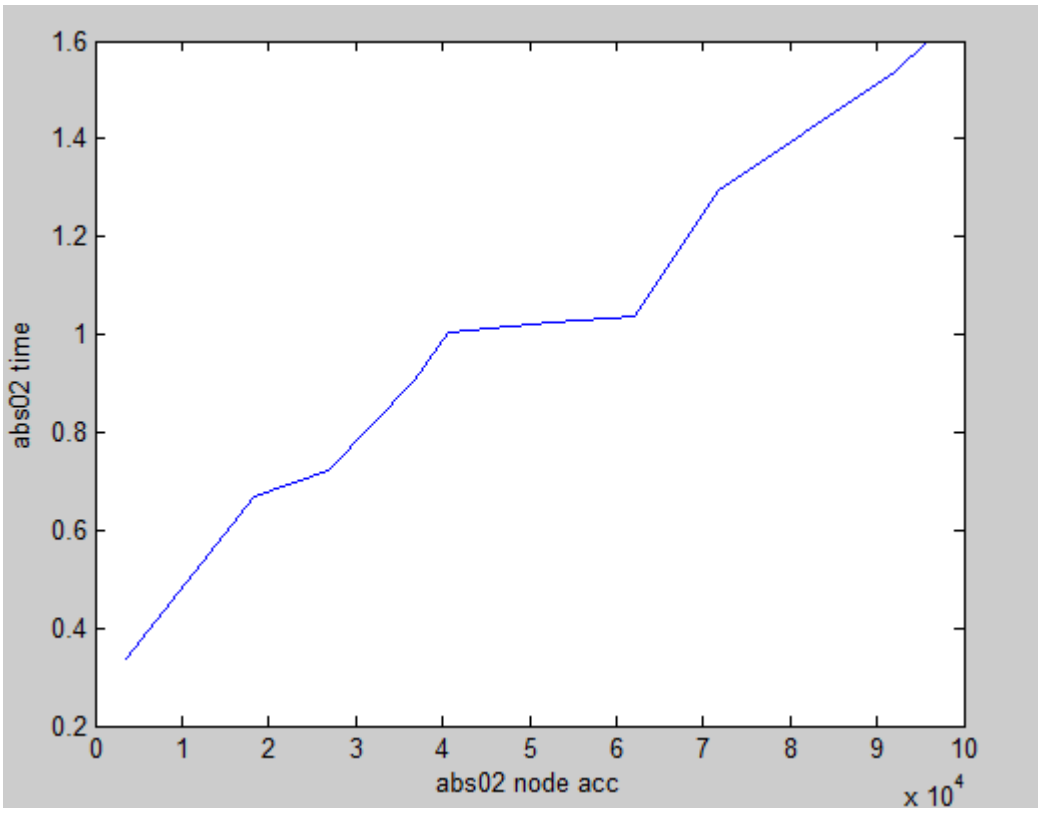
διανομή #διαστάσεων	uni 02	uni 03	uni 09	rea 02	rea 03	rea 05	rea 09	rea 16	rea 22	rea 26
#διαιρέσεων	4157	4158	4161	7841	4966	6566	6976	5449	6230	5927
#διαπεράσεις	4637	4637	4641	8746	5540	7324	7784	6078	6948	6610
#κόμβων	95942	95941	96037	180959	114622	151534	161060	125767	143769	136778
χρόνος	1.599041	1.599025	2.600623	3.015993	1.91038	2.525578	2.684000	2.096123	2.796166	3.076123

Επίσης, στα παρακάτω διαγράμματα φαίνεται για κάθε διανομή πόσος χρόνος περνάει,πόσοι κόμβοι δημιουργήθηκαν , πόσες διαιρέσεις των κόμβων και πόσες διαπεράσεις κόμβων έχουμε για κάθε 10000 εισαγωγές τετραγώνων.

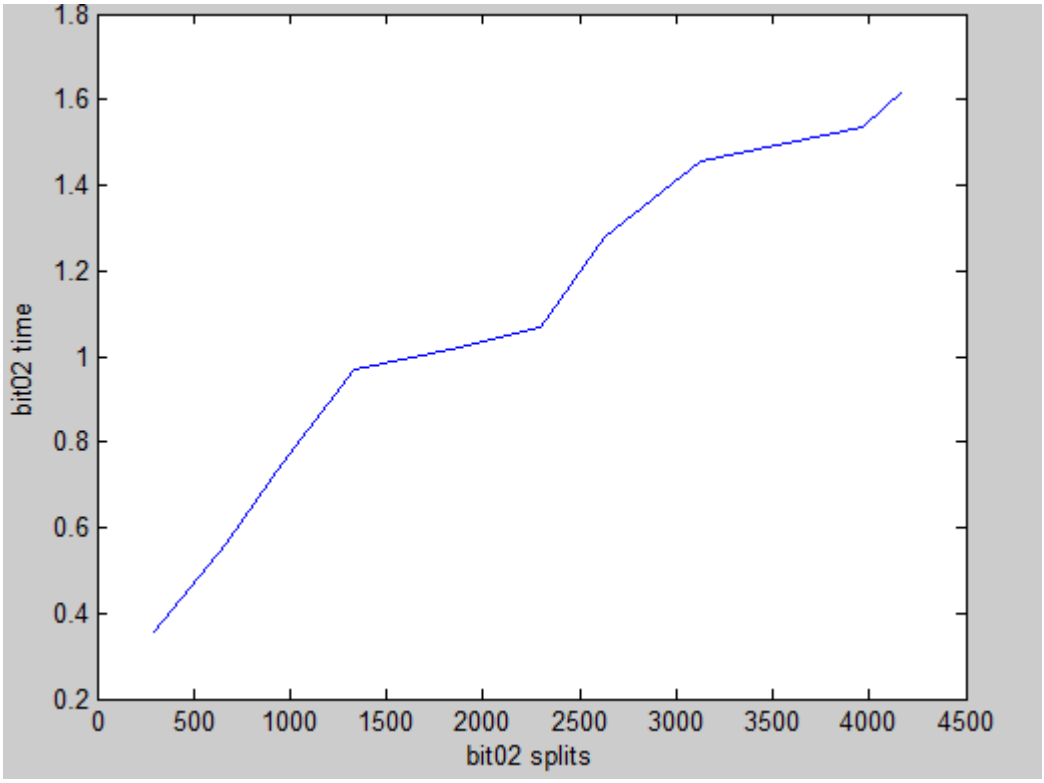
-*Διανομή Absolute δύο διαστάσεων*

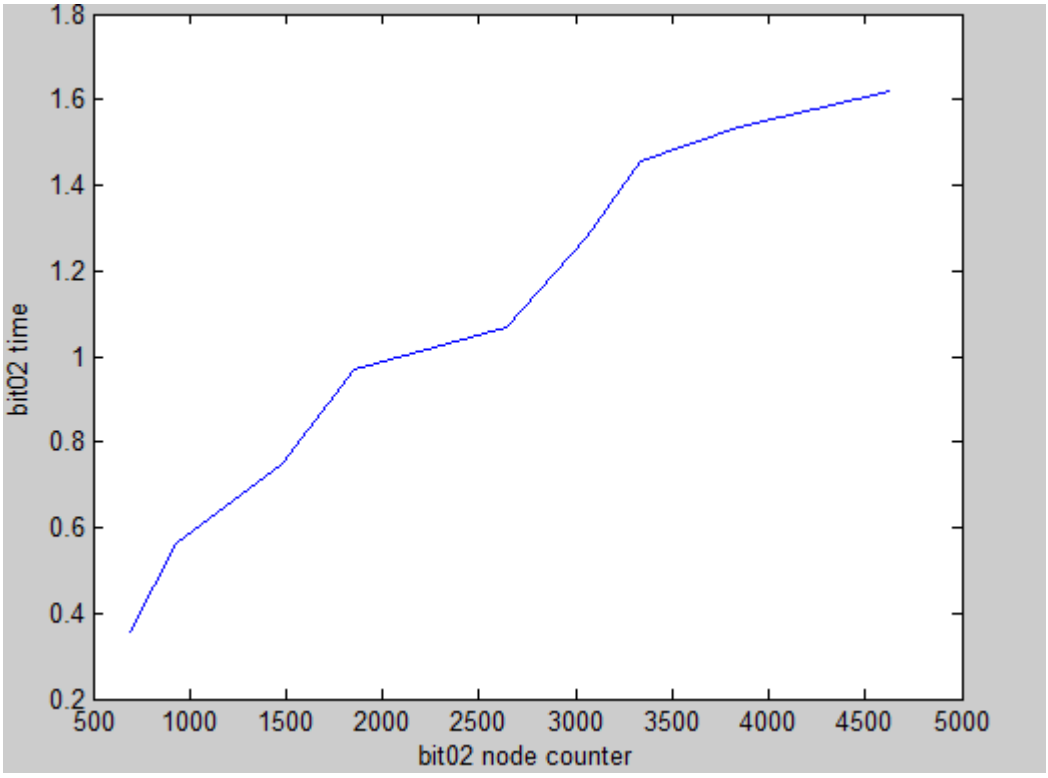


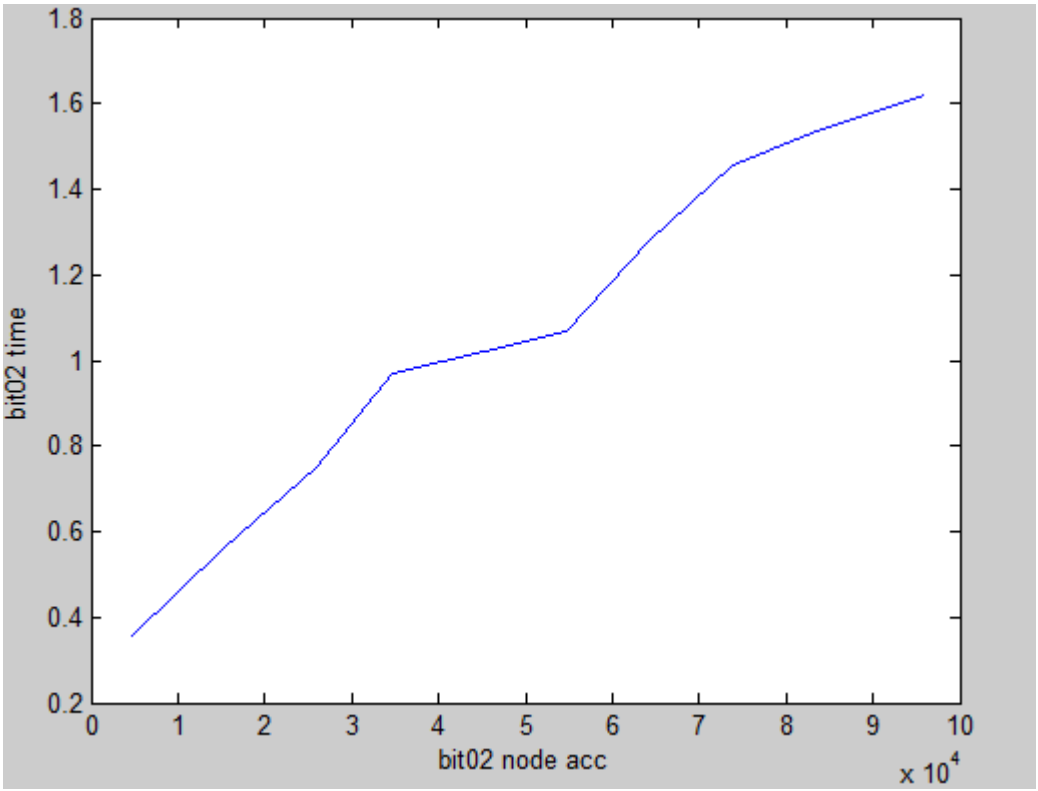

1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
abs02 time
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
abs02 node counter


1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
abs02 time
0
1
2
3
4
5
6
7
8
9
10
abs02 node acc
x 10^4

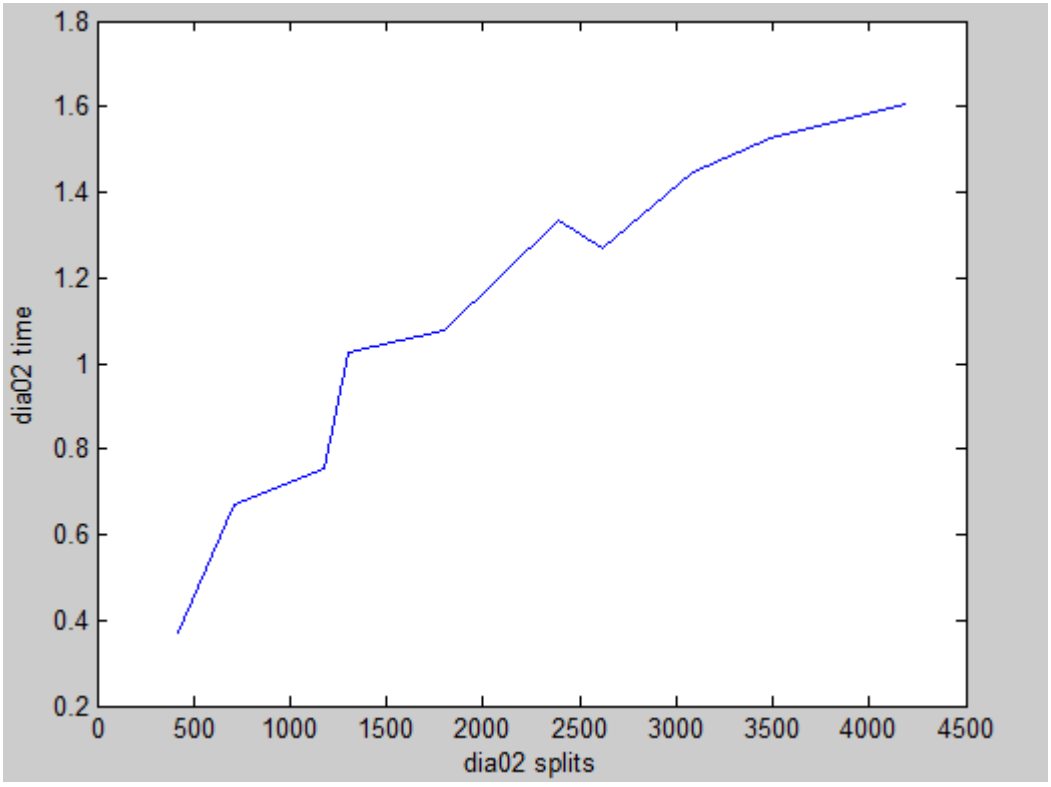
-*Διανομή Bit δύο διαστάσεων*

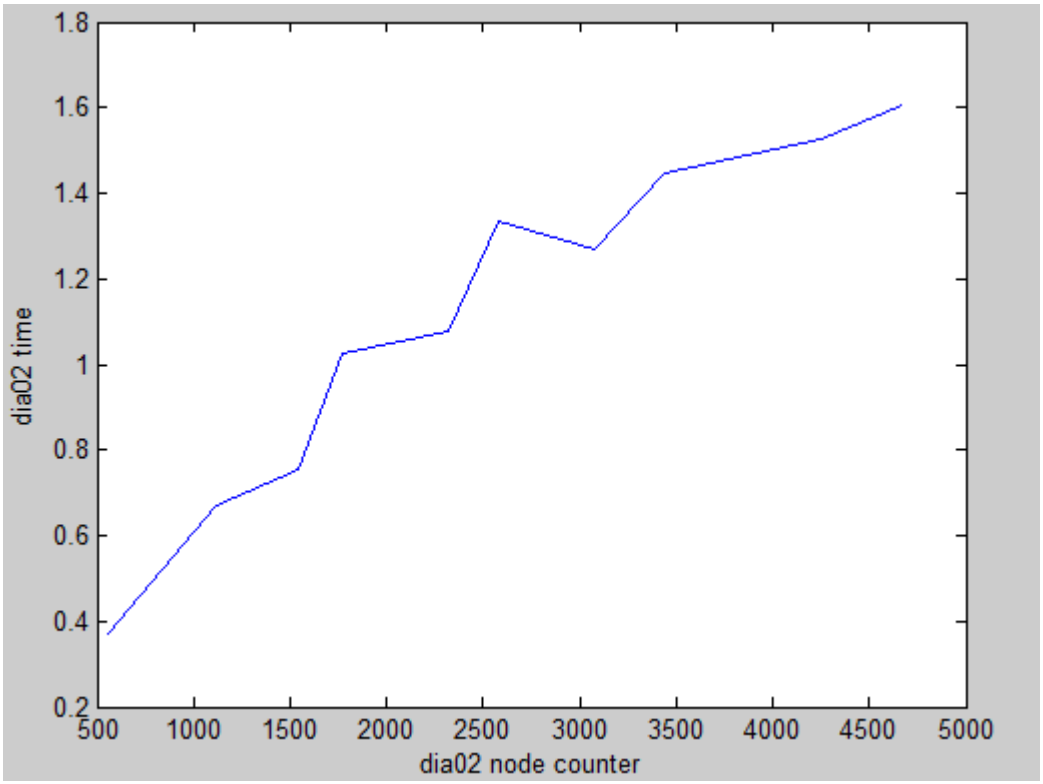


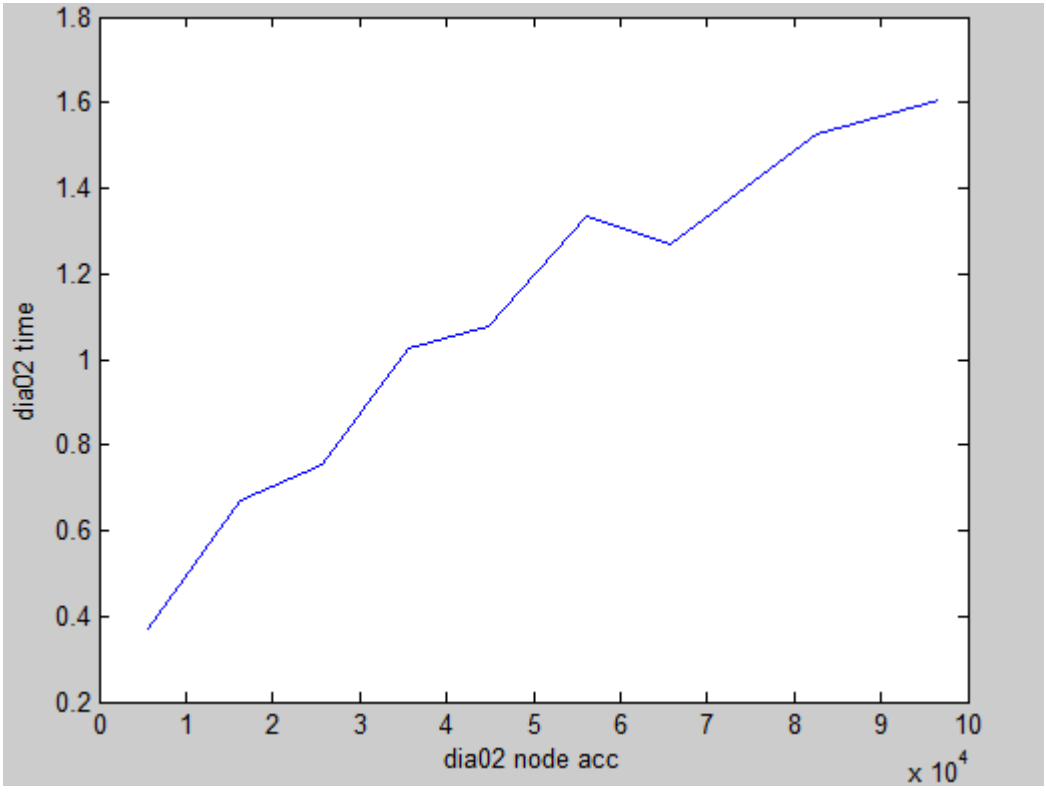




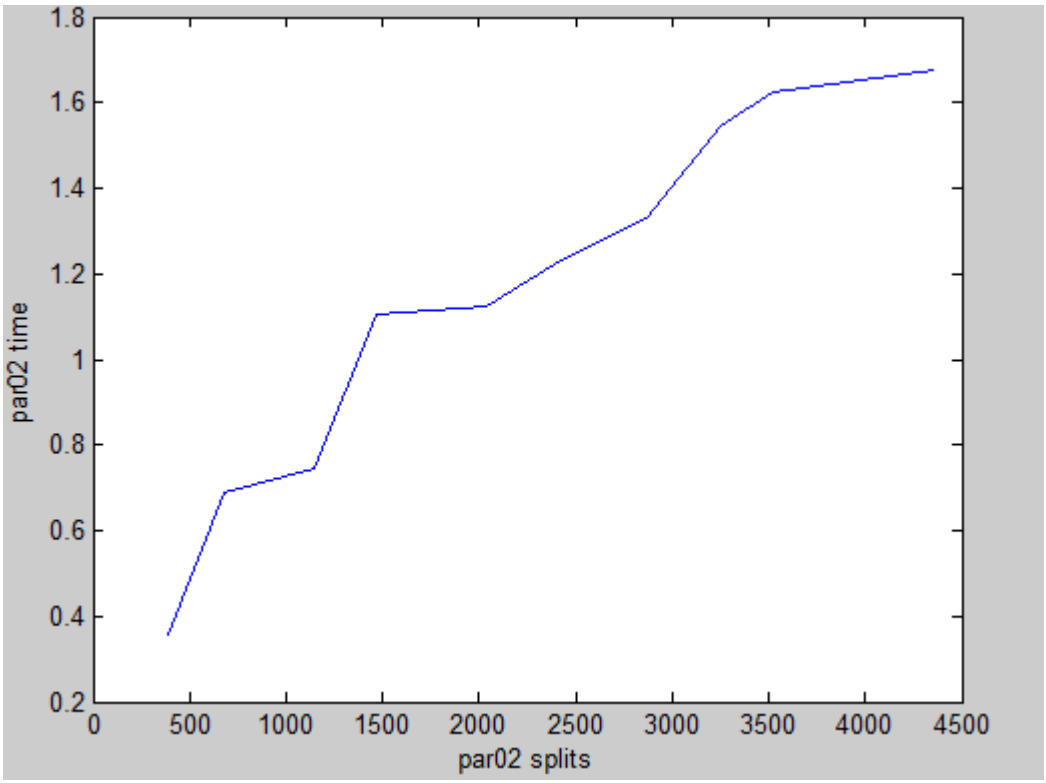
-*<u>Διανομή Diagonal δύο διαστάσεων</u>*

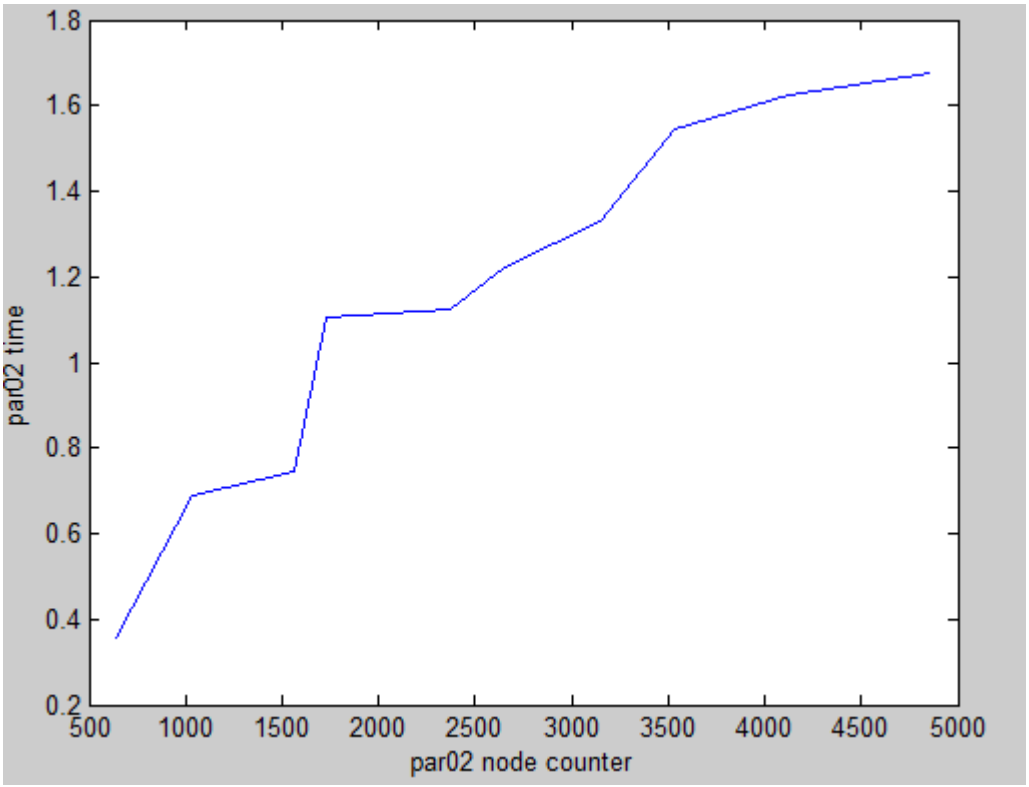


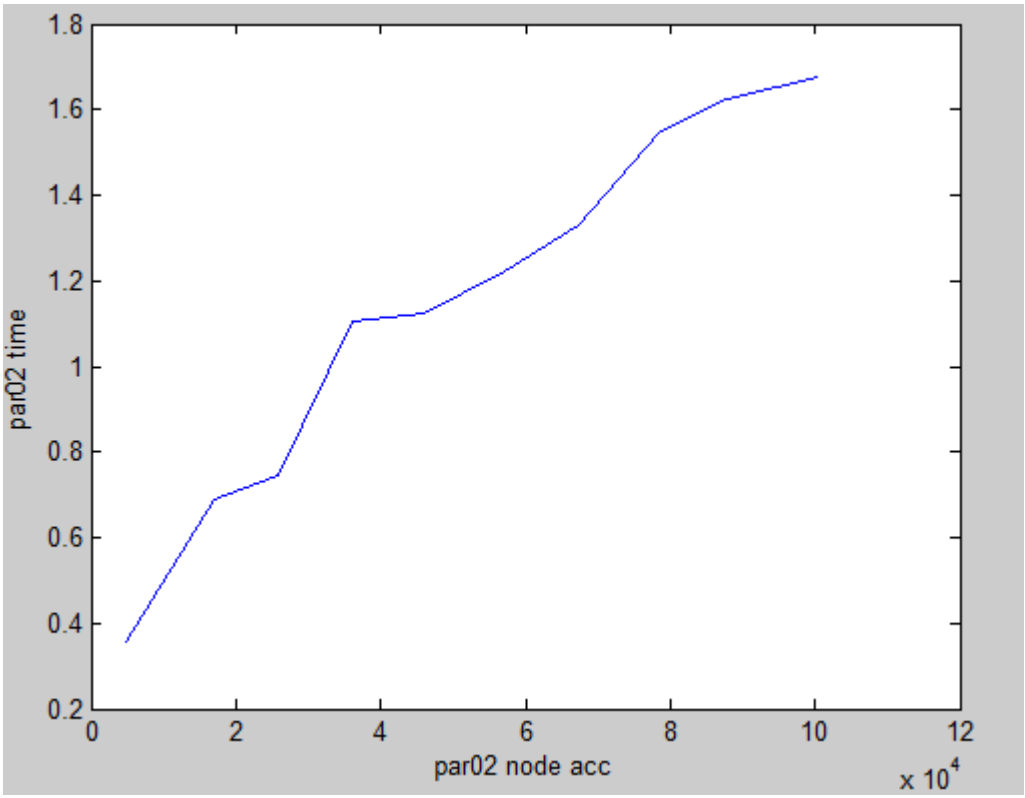

dia02 time
dia02 node counter
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
500
1000
1500
2000
2500
3000
3500
4000
4500
5000


dia02 time
dia02 node acc
x 10^4
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
9
10

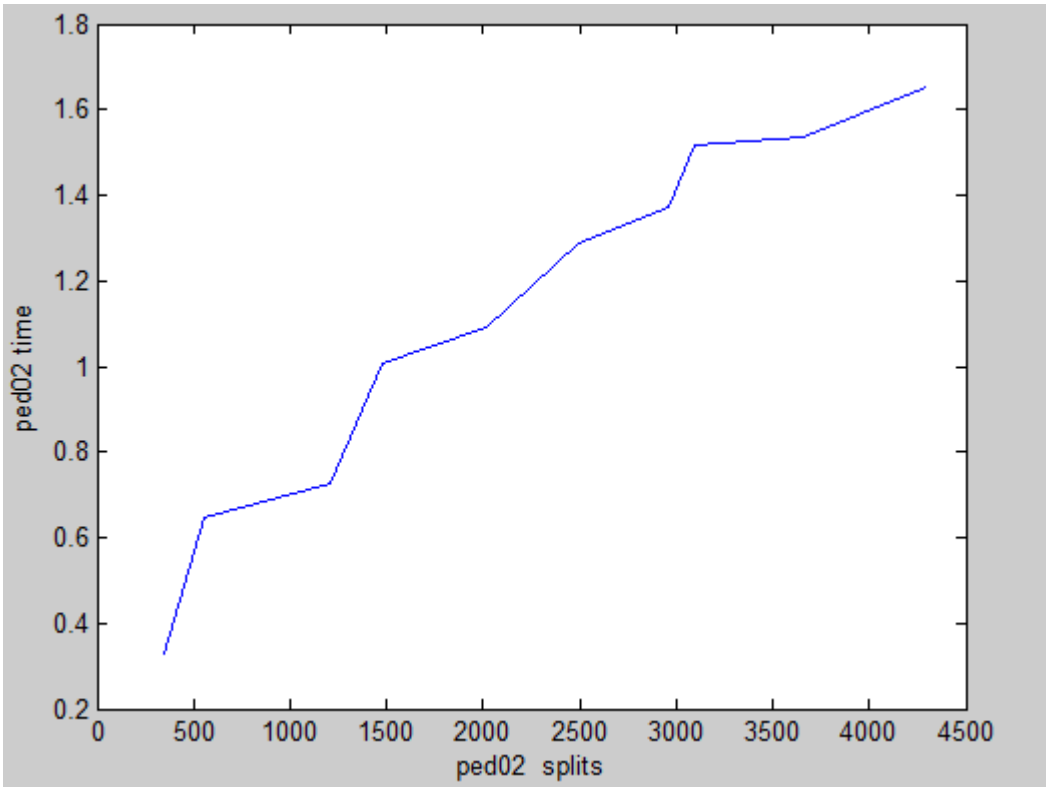
-*<u>Διανομή Parsel δύο διαστάσεων</u>*

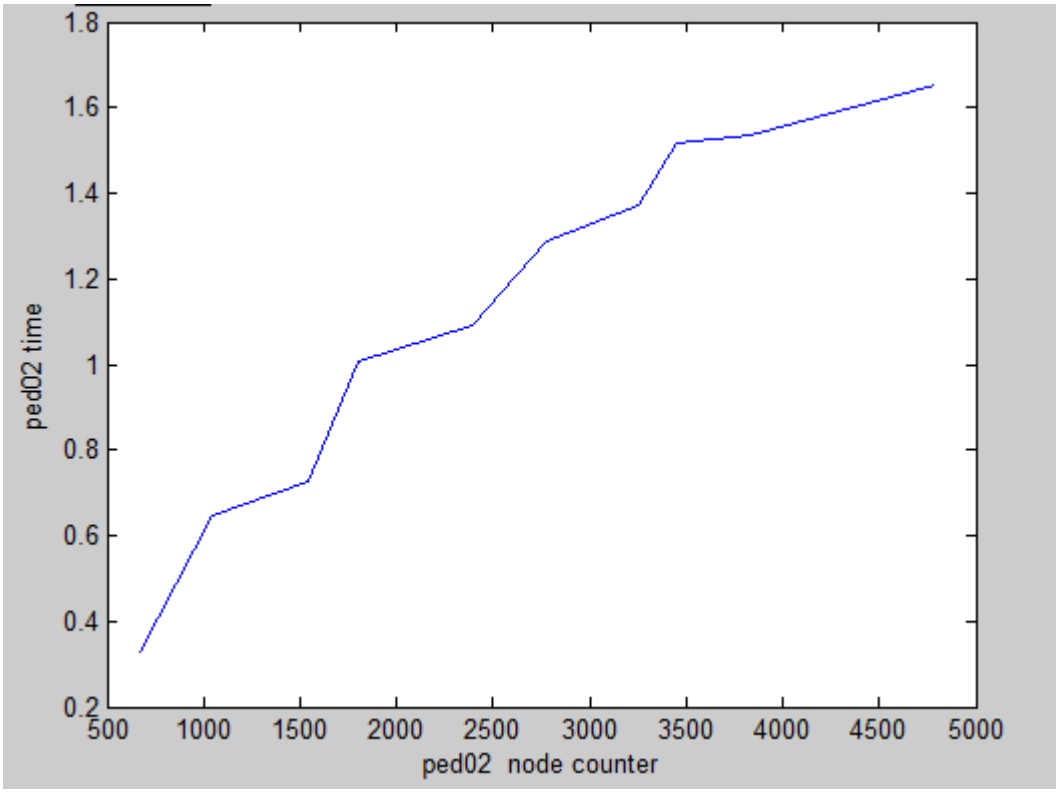


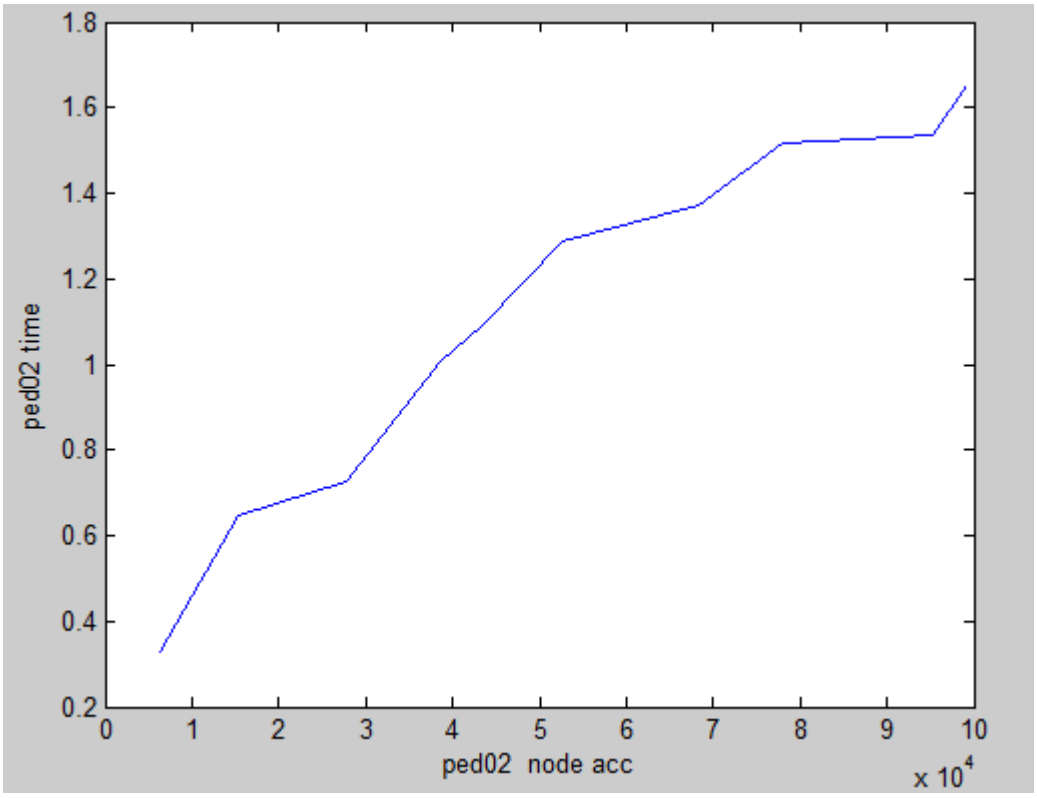




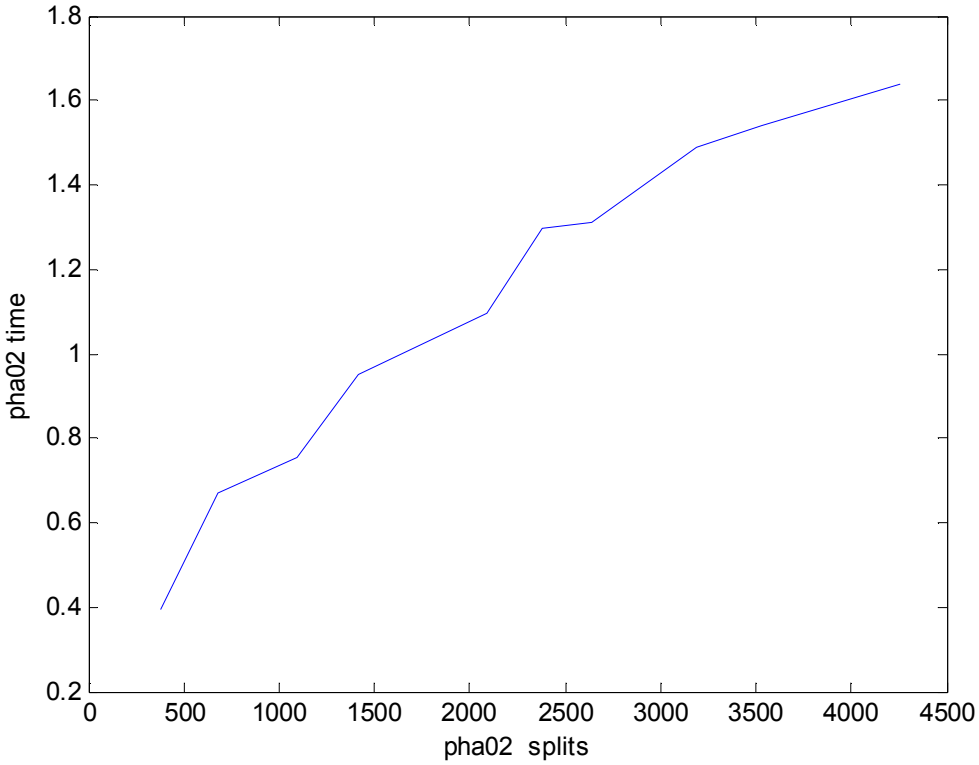
-*<u>Διανομή P-edges δύο διαστάσεων</u>*

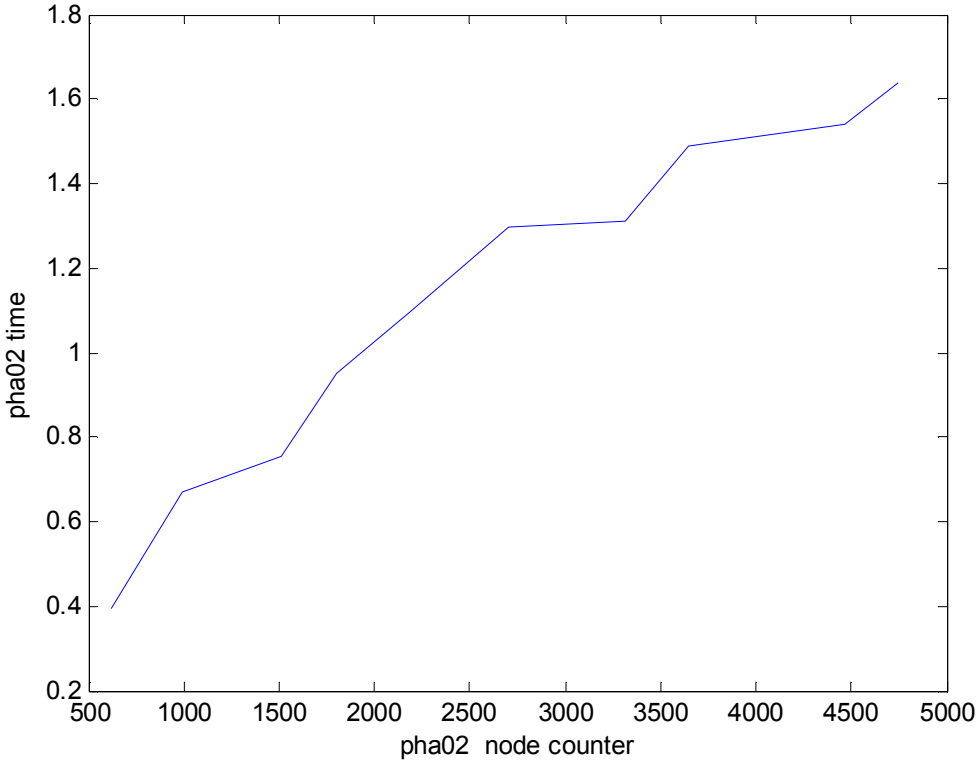


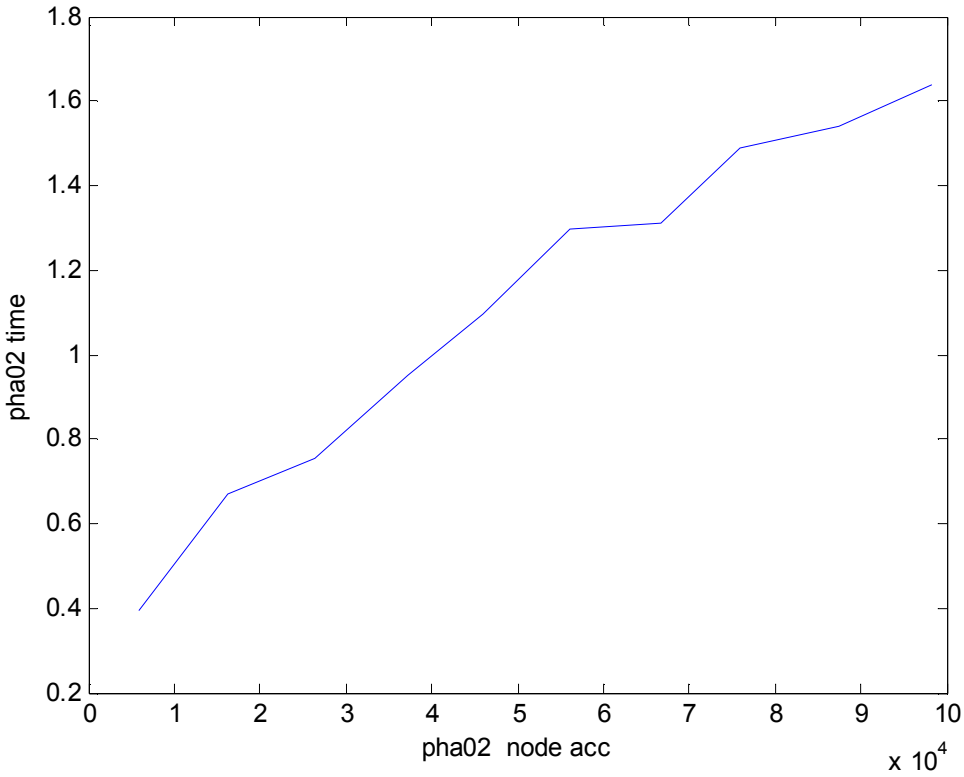

ped02 time
ped02 node counter
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
500
1000
1500
2000
2500
3000
3500
4000
4500
5000


ped02 time
ped02 node acc
x 10^4
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
9
10

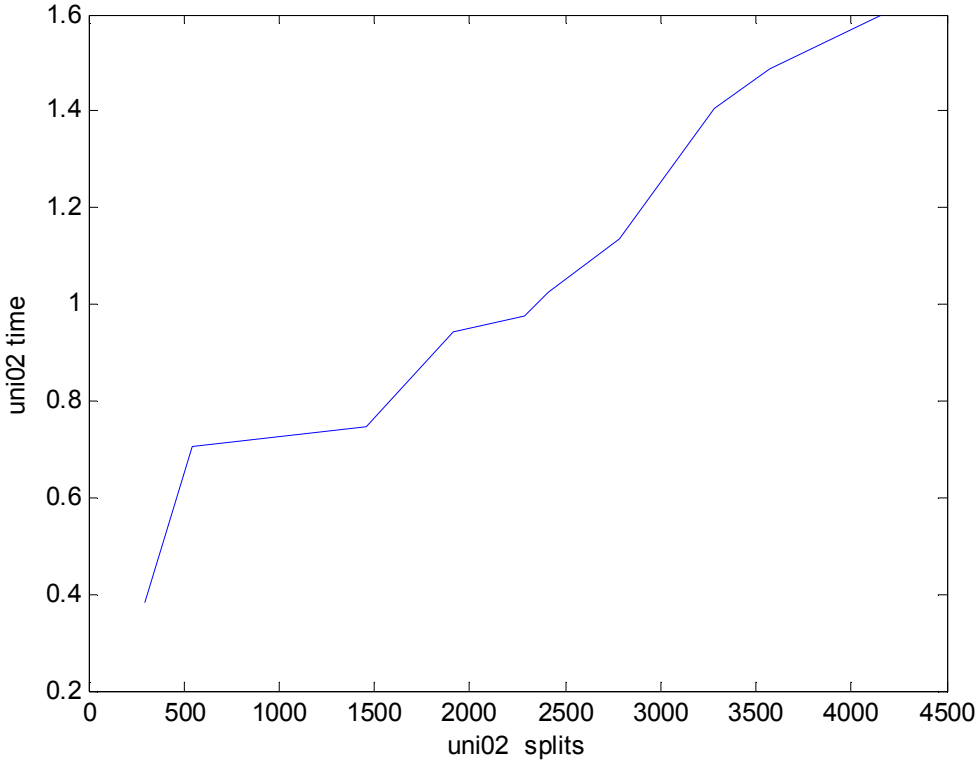
-*Διανομή P-haze δύο διαστάσεων*

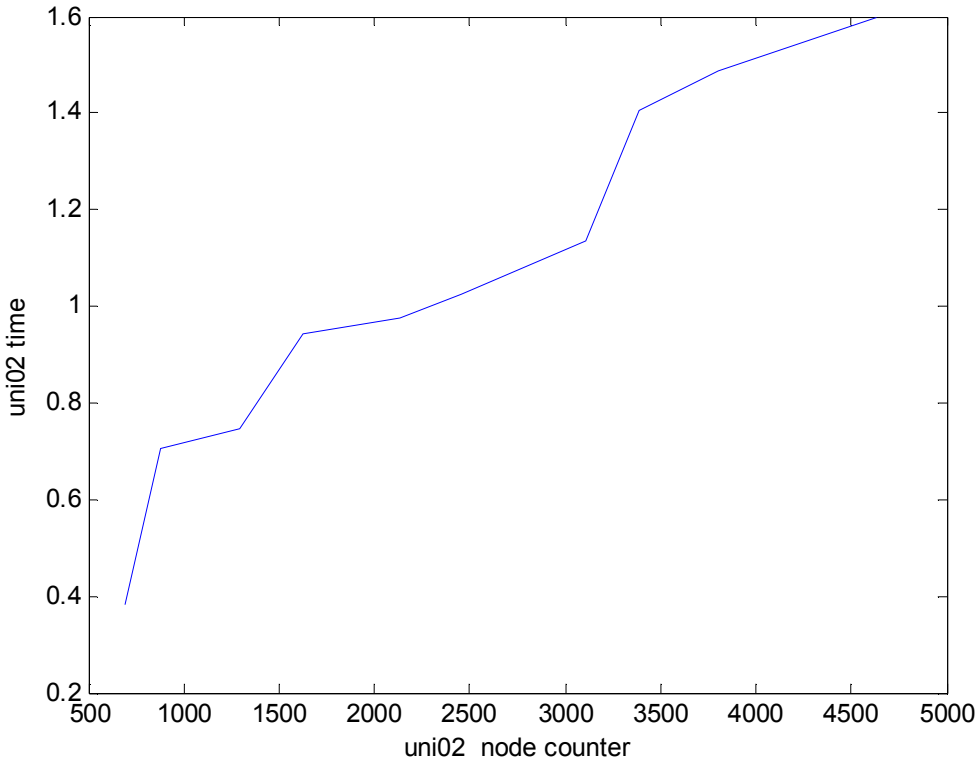


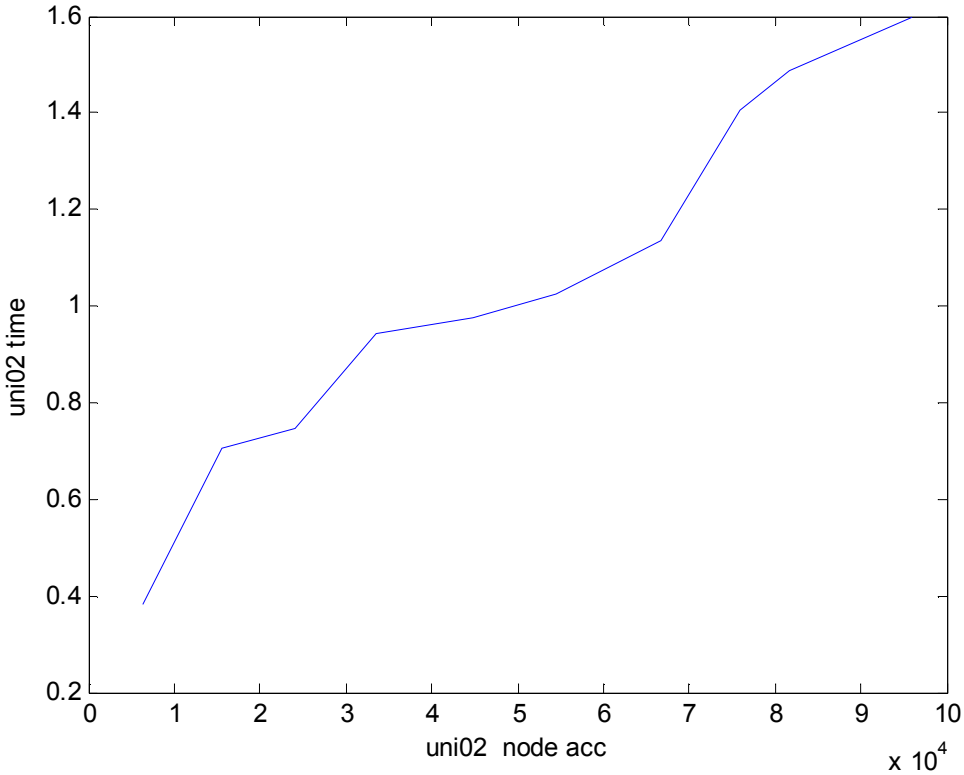




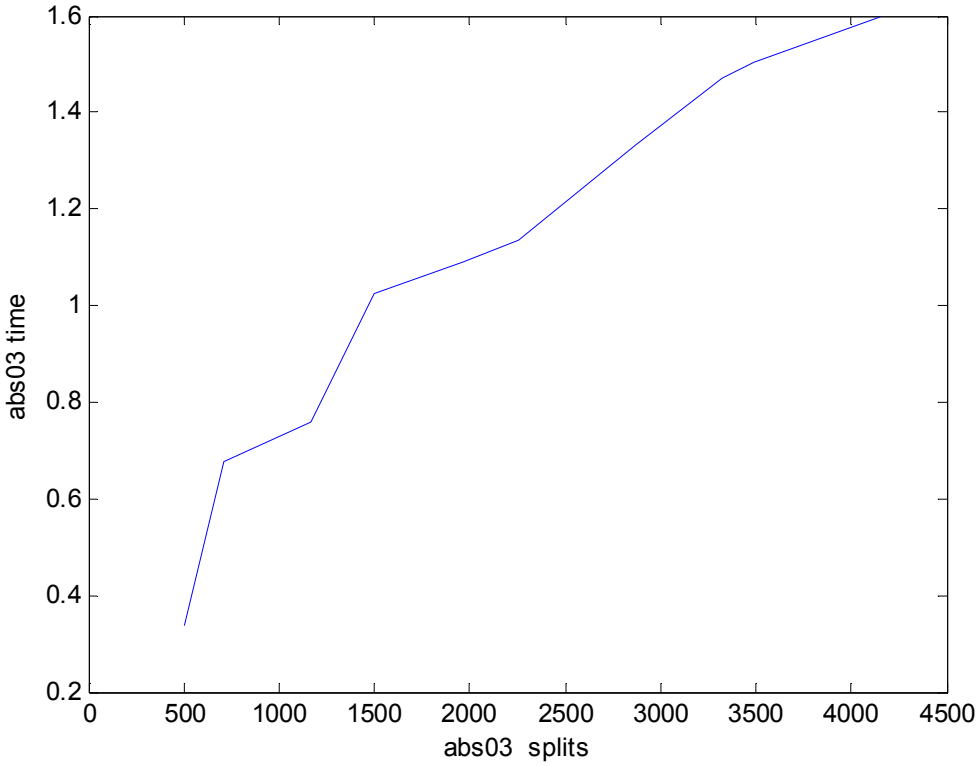
-*<u>Διανομή Uniform δύο διαστάσεων</u>*

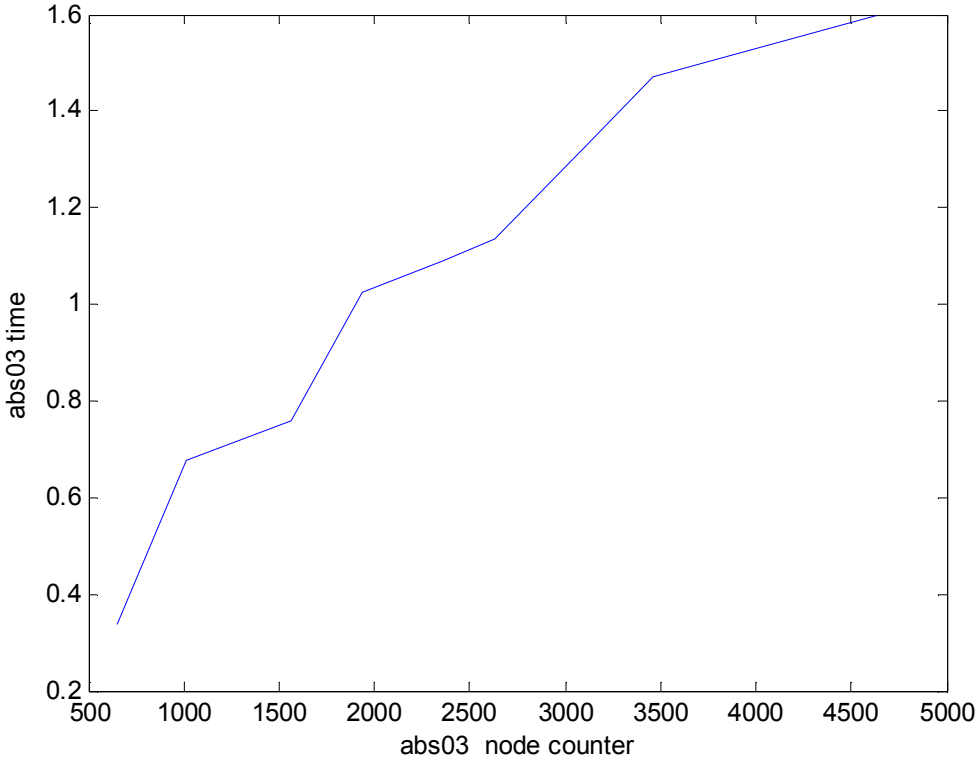


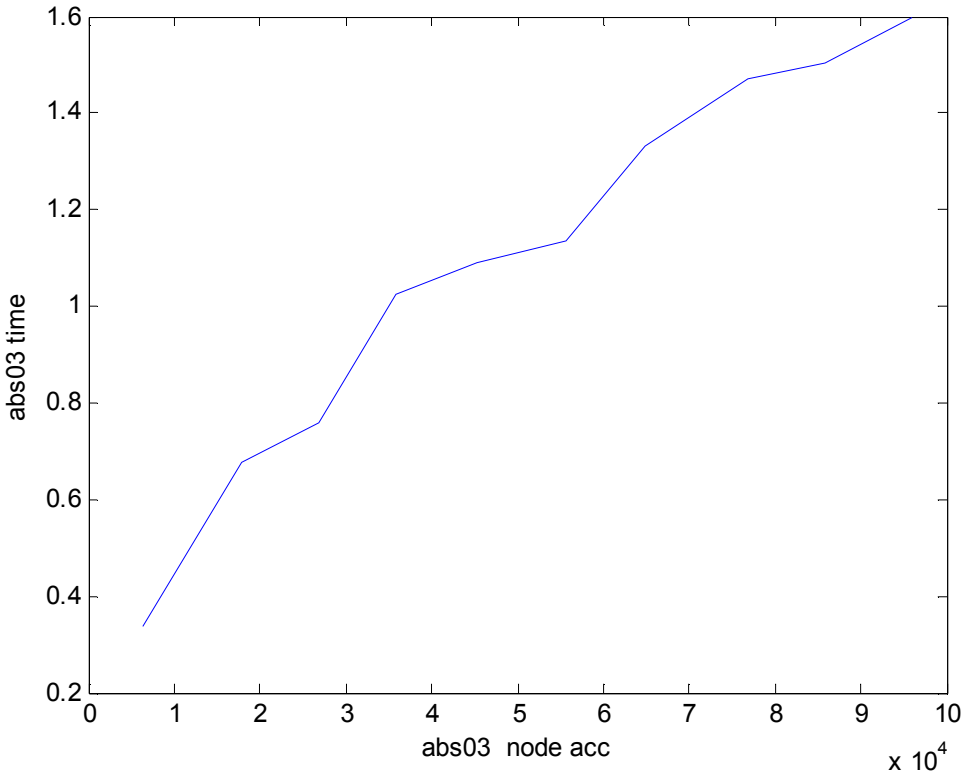

1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
uni02 time
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
uni02 node counter


1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
uni02 time
0
1
2
3
4
5
6
7
8
9
10
uni02 node acc
x 10^4

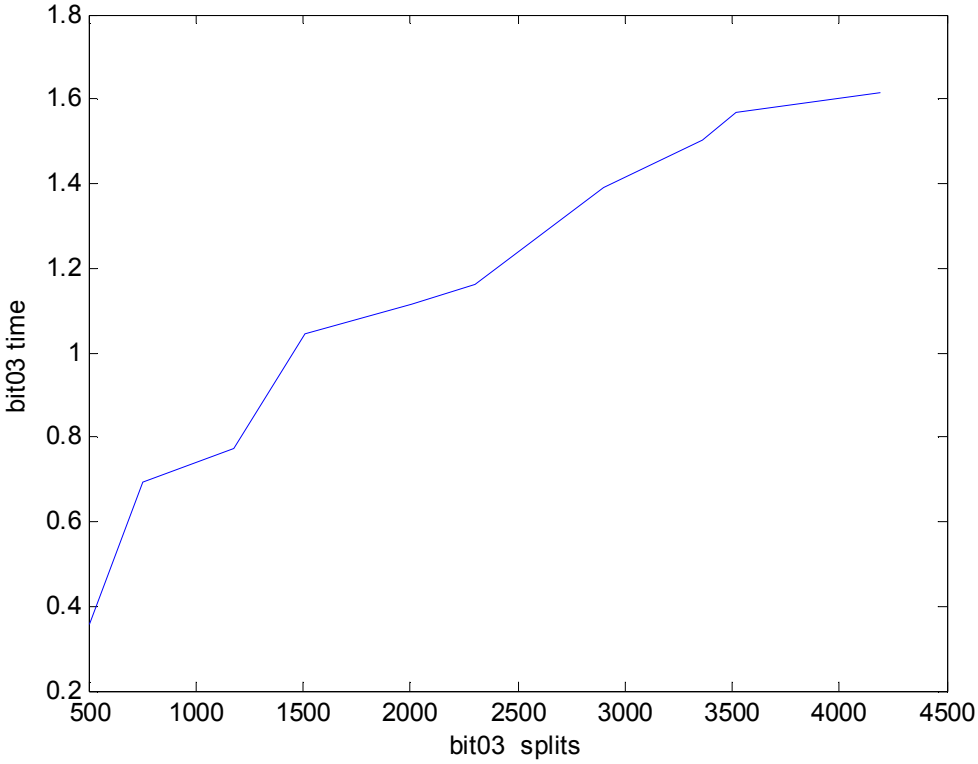
-*<u>Διανομή Absolute τριών διαστάσεων</u>*

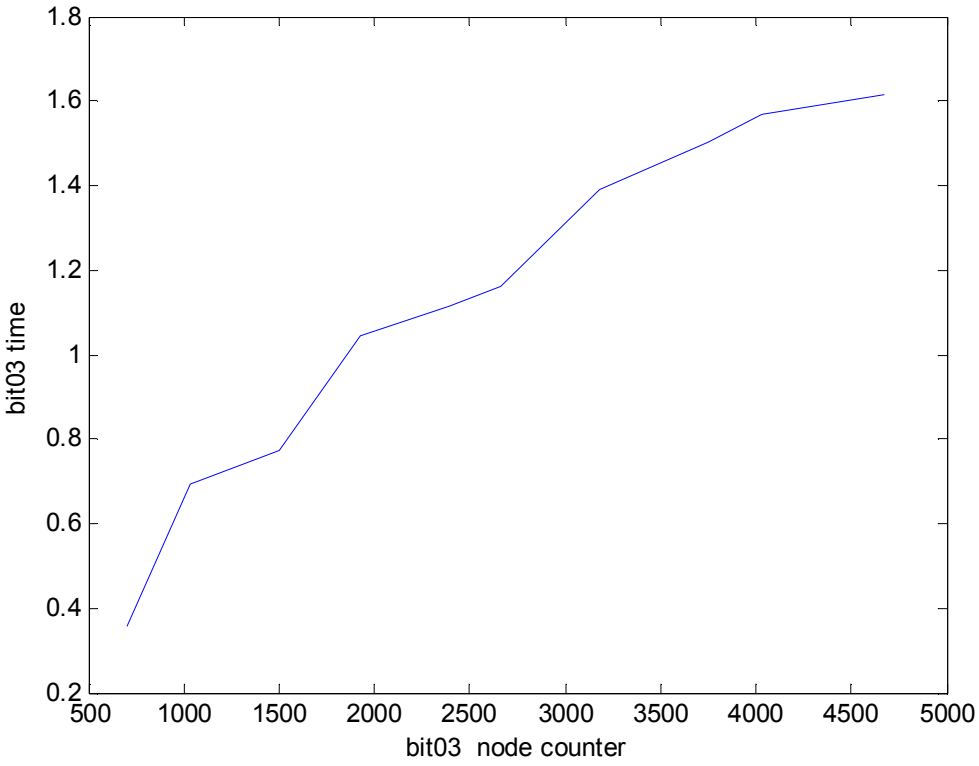


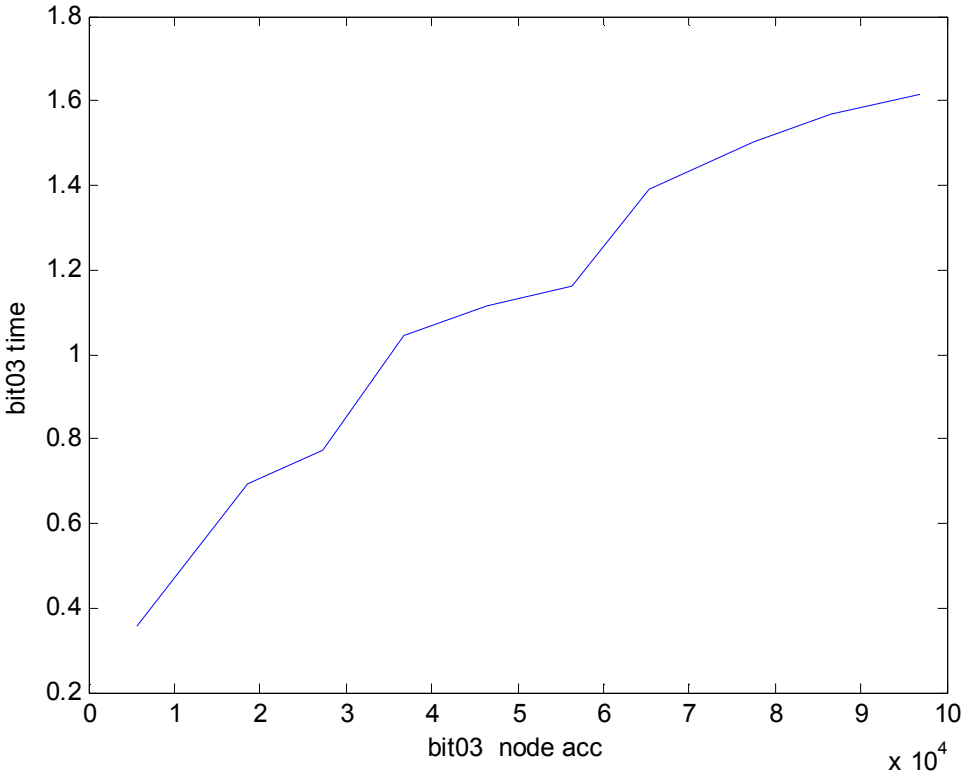




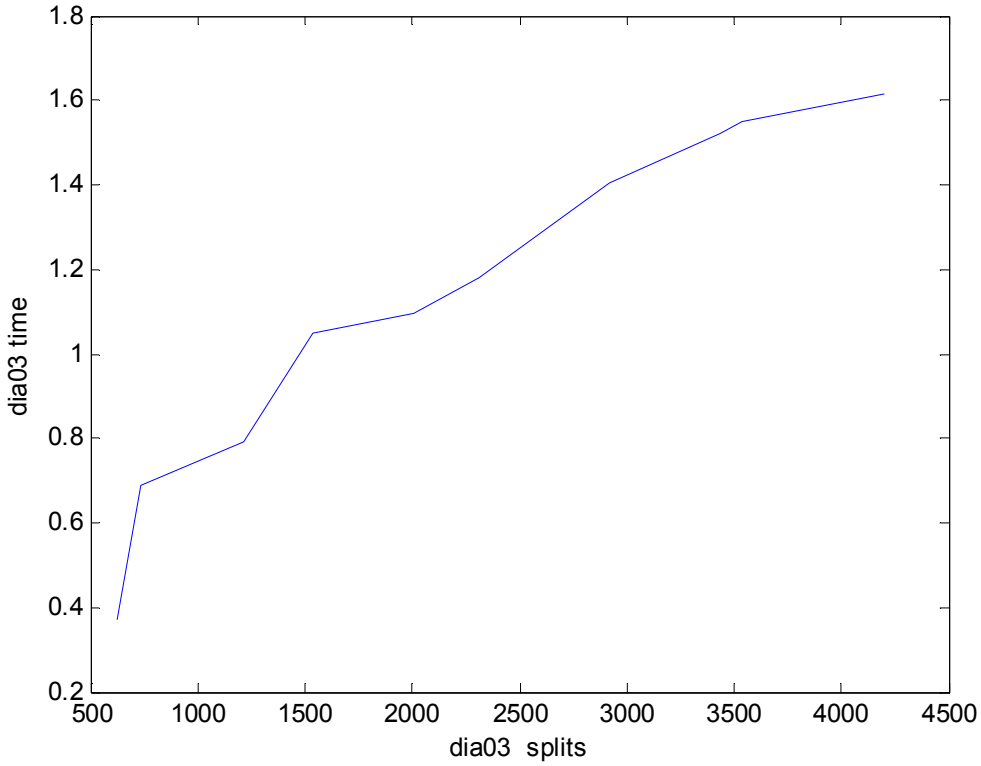
-*<u>Διανομή Bit τριών διαστάσεων</u>*

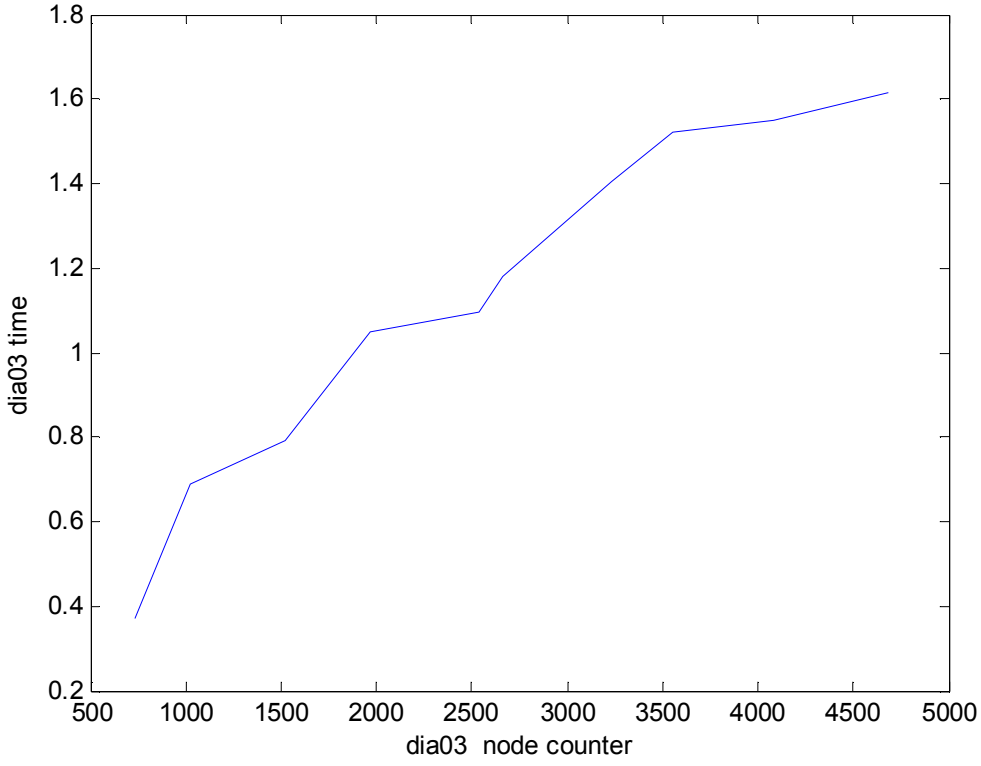


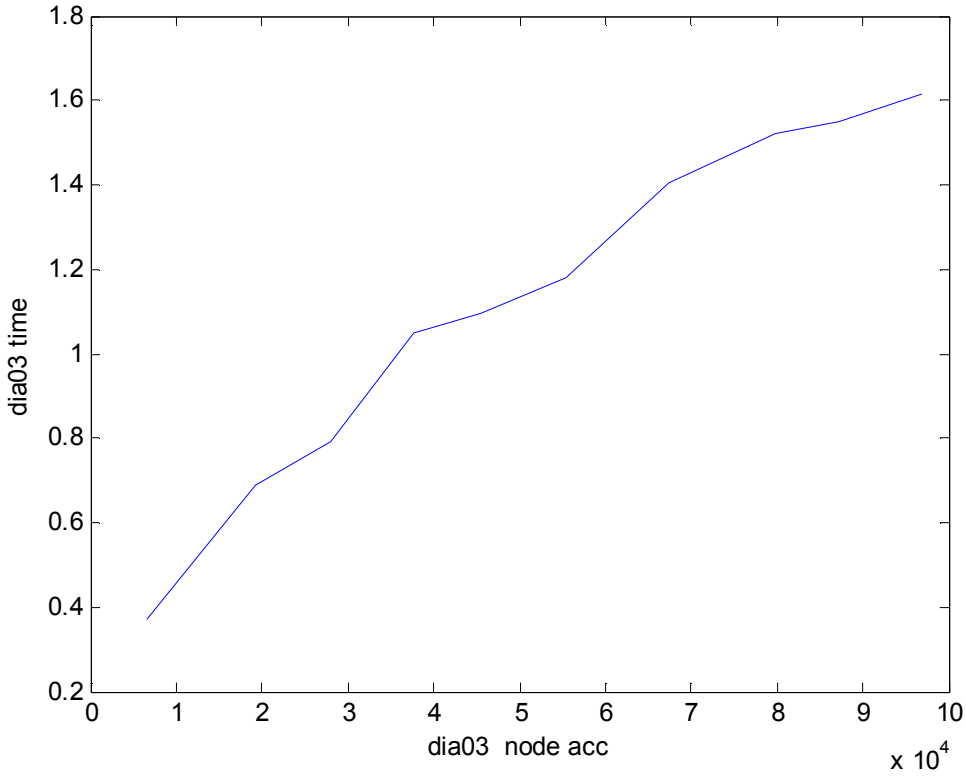

1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
bit03 time
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
bit03 node counter


1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
bit03 time
0
1
2
3
4
5
6
7
8
9
10
bit03 node acc
x 10^4

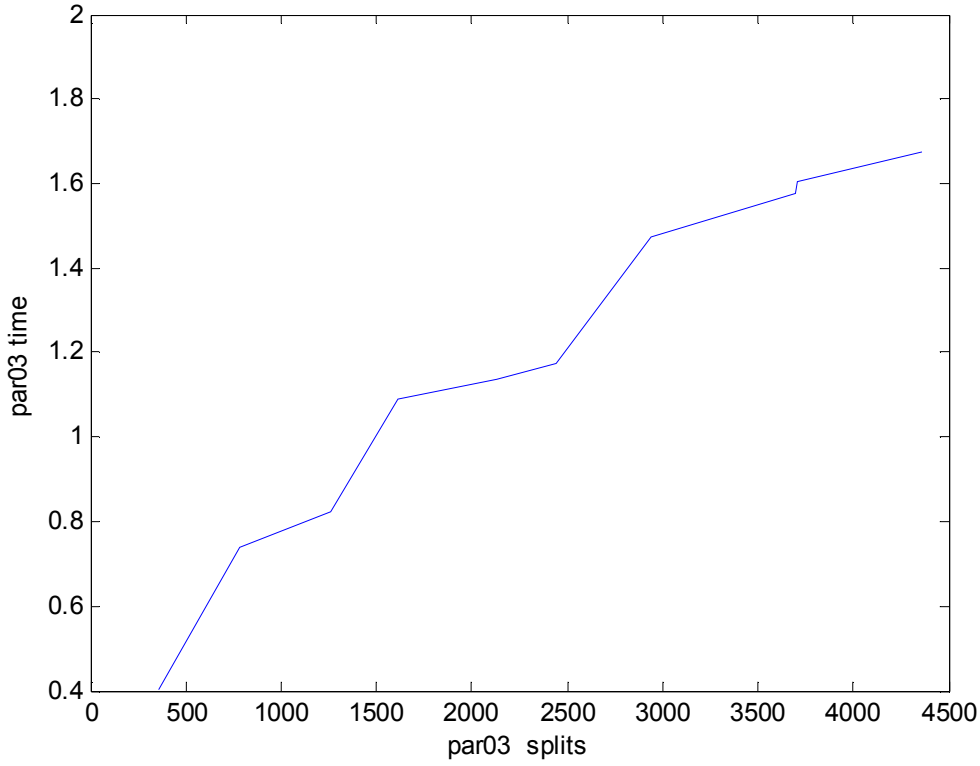
-*Διανομή Diagonal τριών διαστάσεων*

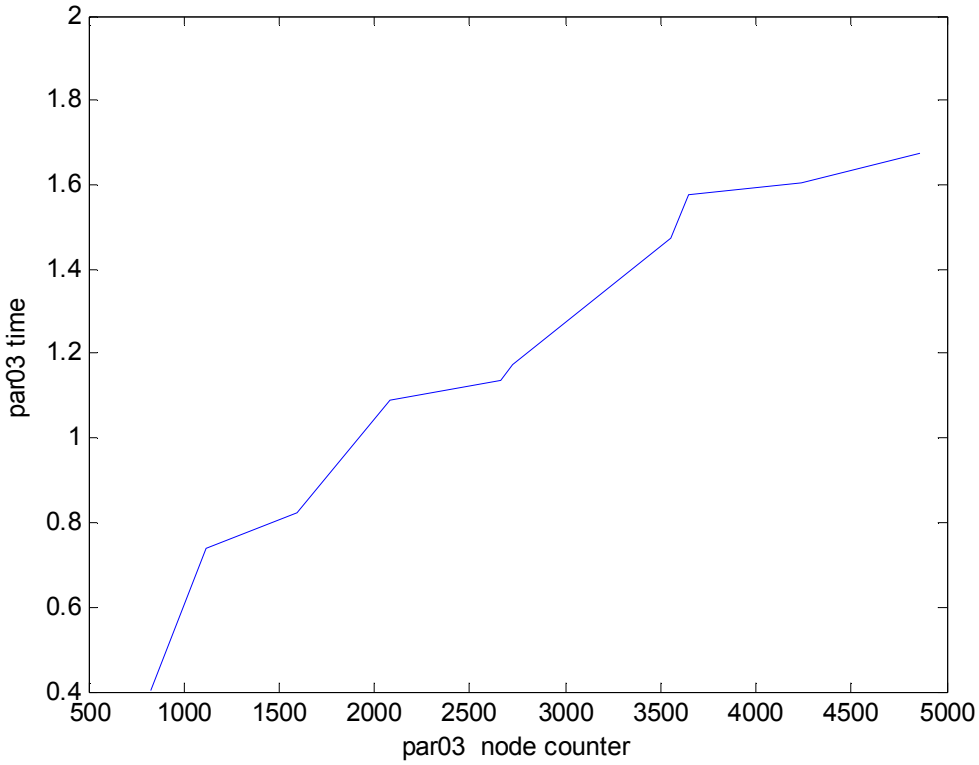


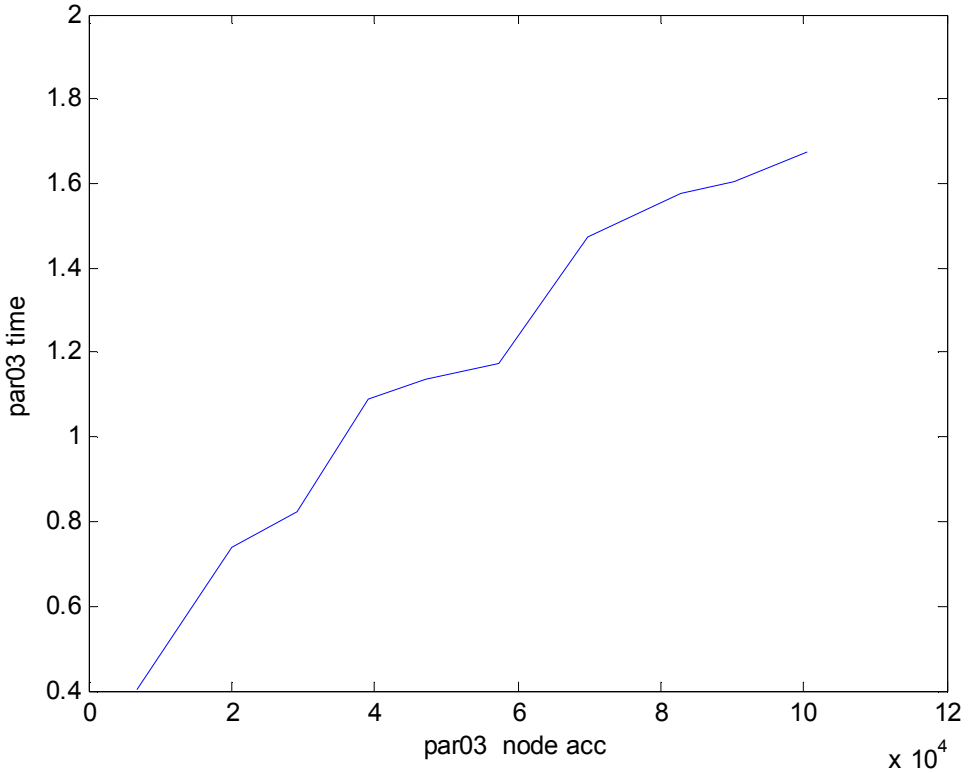




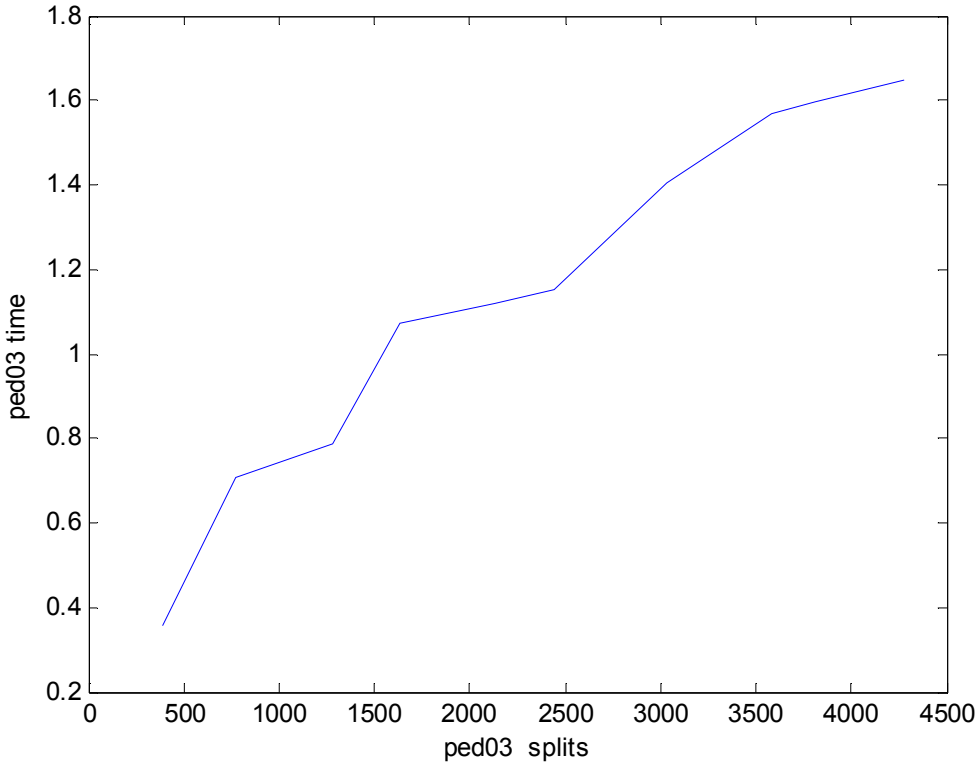
-*<u>Διανομή Parsel τριών διαστάσεων</u>*

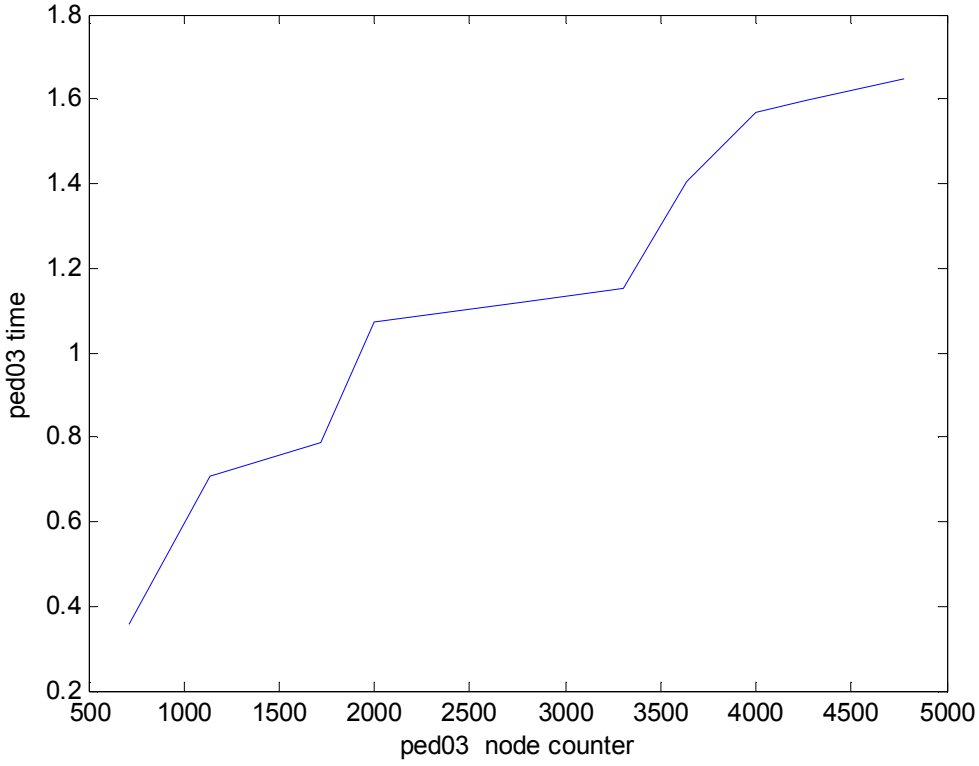


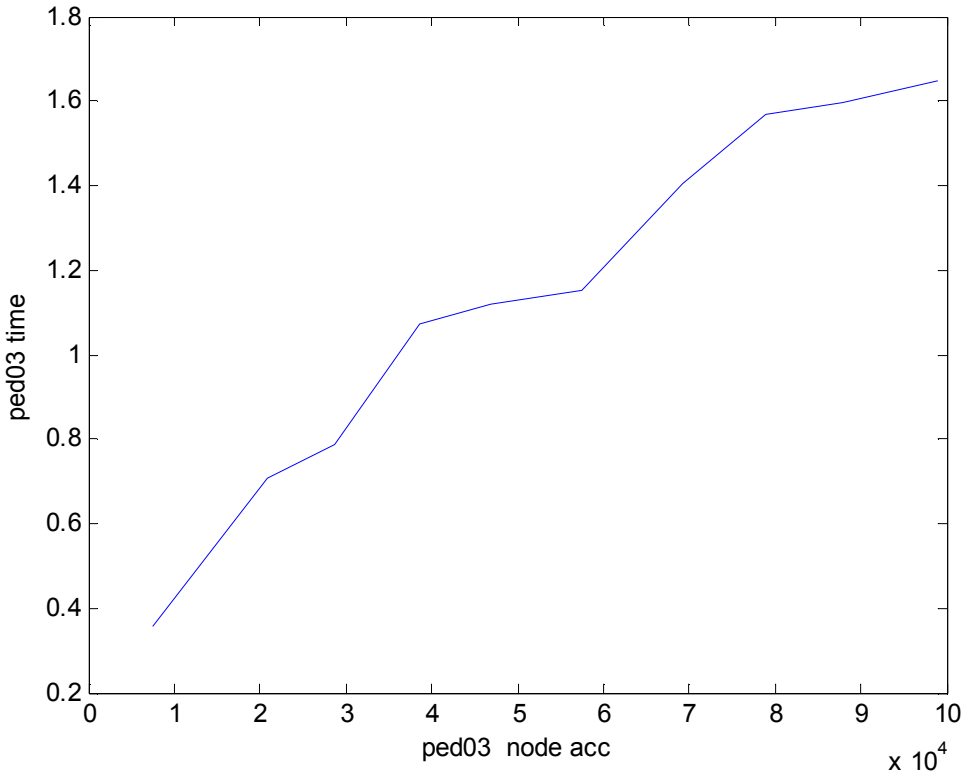

par03 time
par03 node counter
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
500
1000
1500
2000
2500
3000
3500
4000
4500
5000


par03 time
par03 node acc
x 10^4
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0
2
4
6
8
10
12

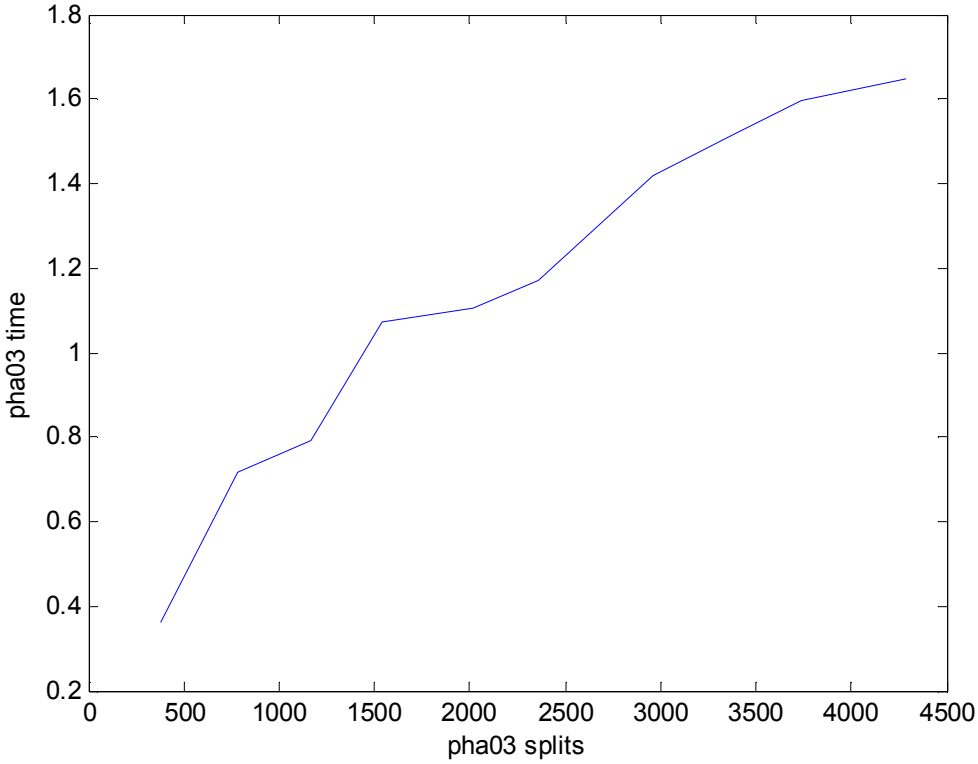
-*Διανομή P-edges τριών διαστάσεων*

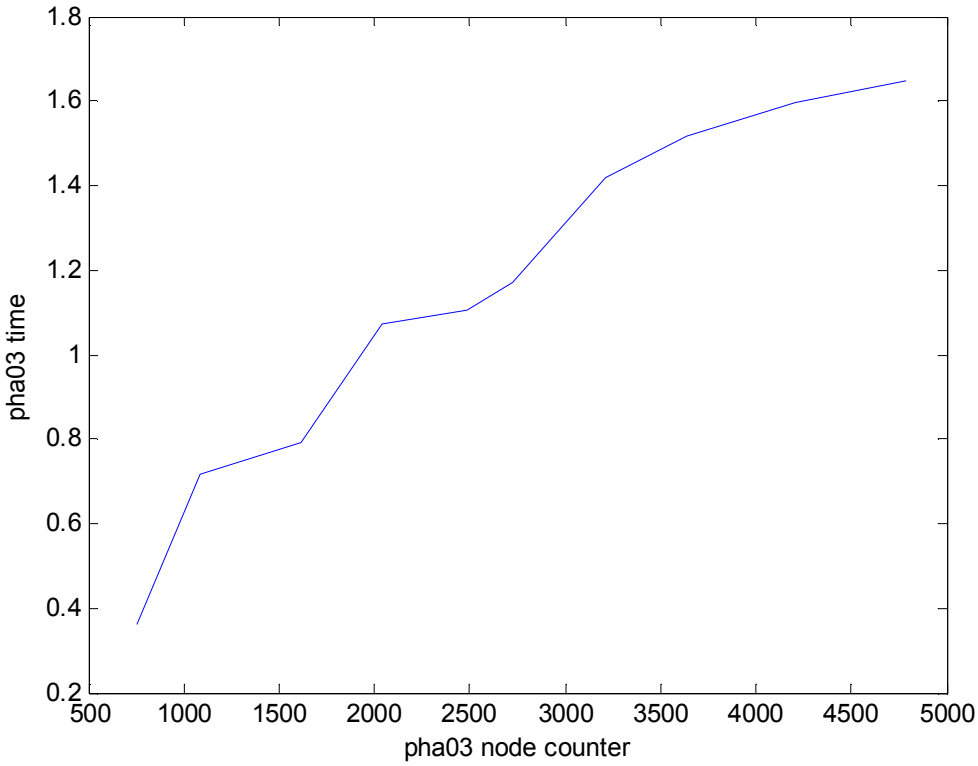


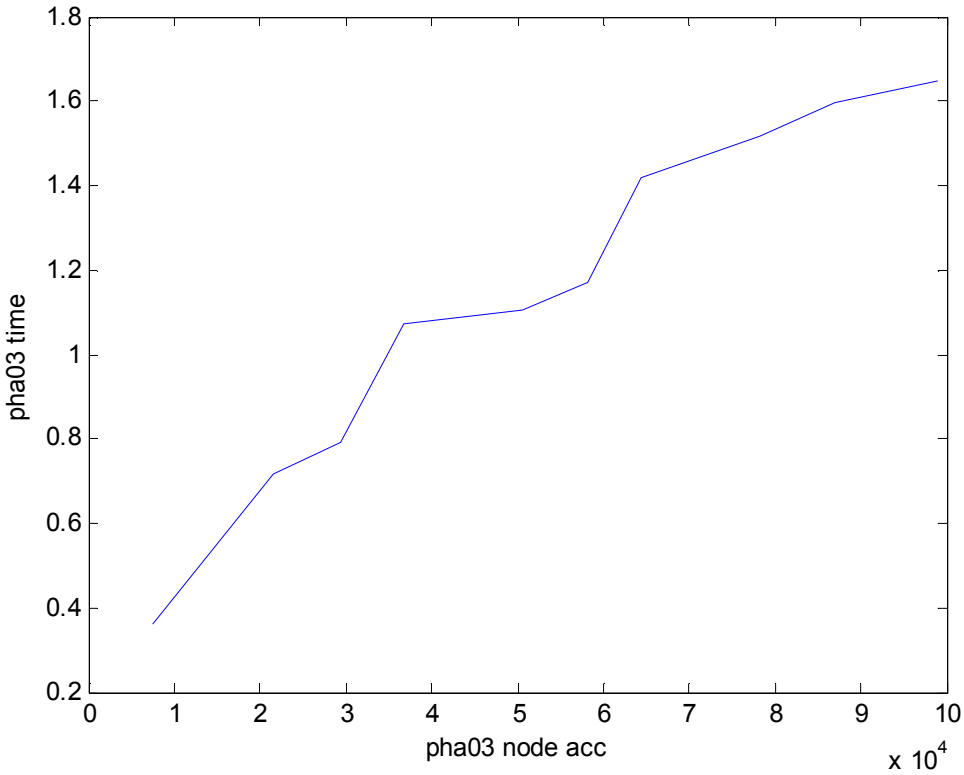




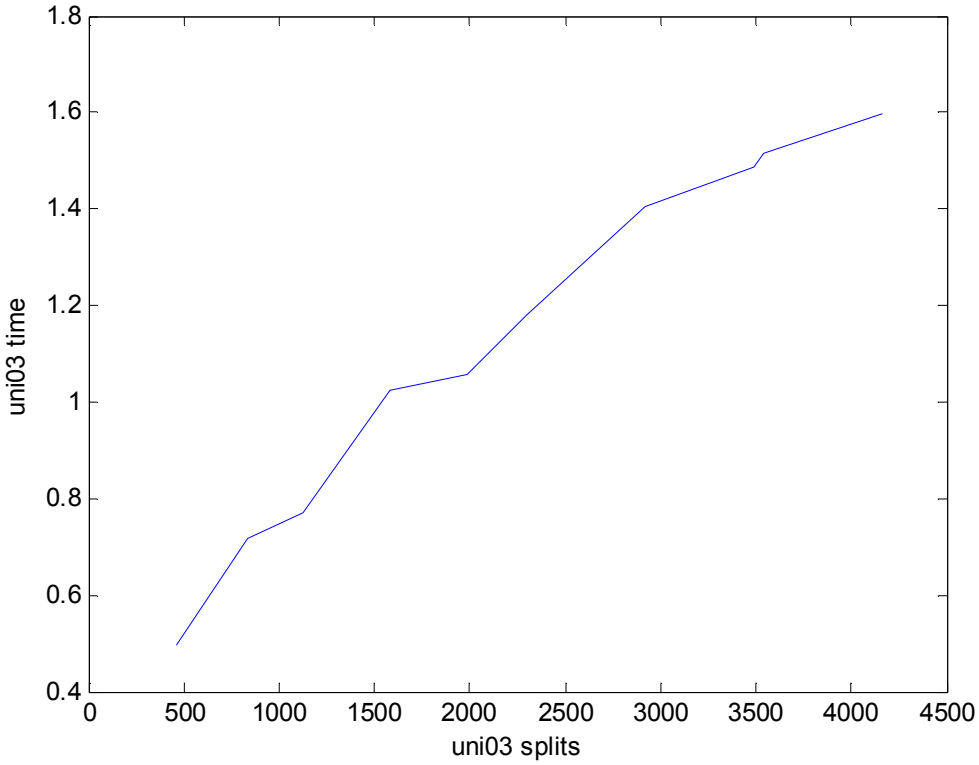
-*<u>Διανομή P-haze τριών διαστάσεων</u>*

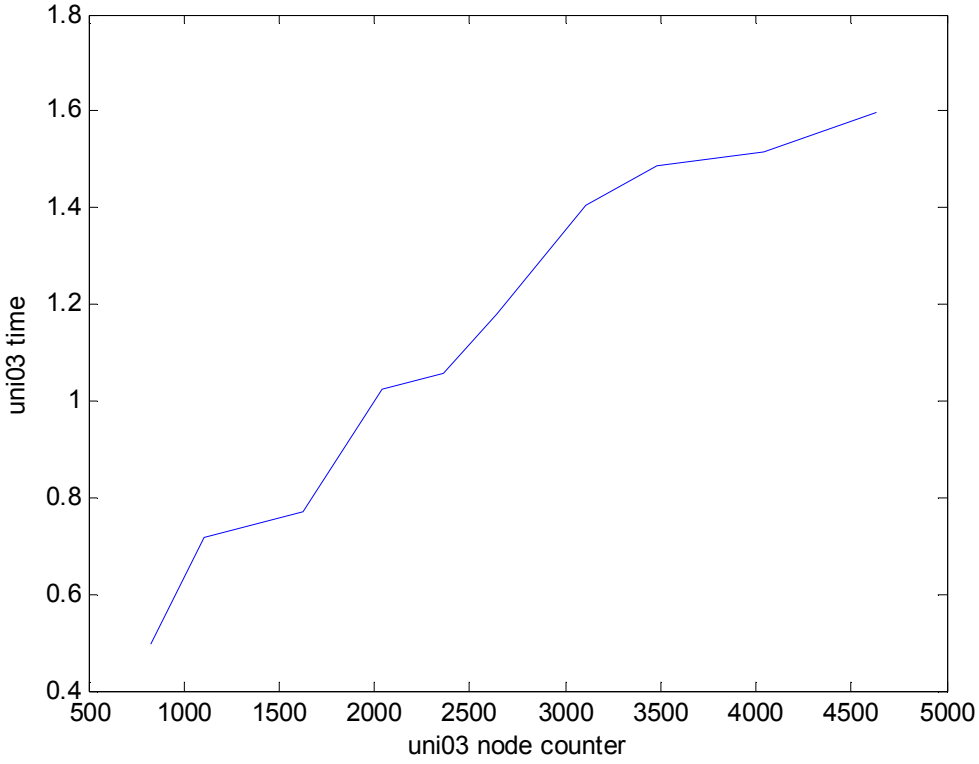


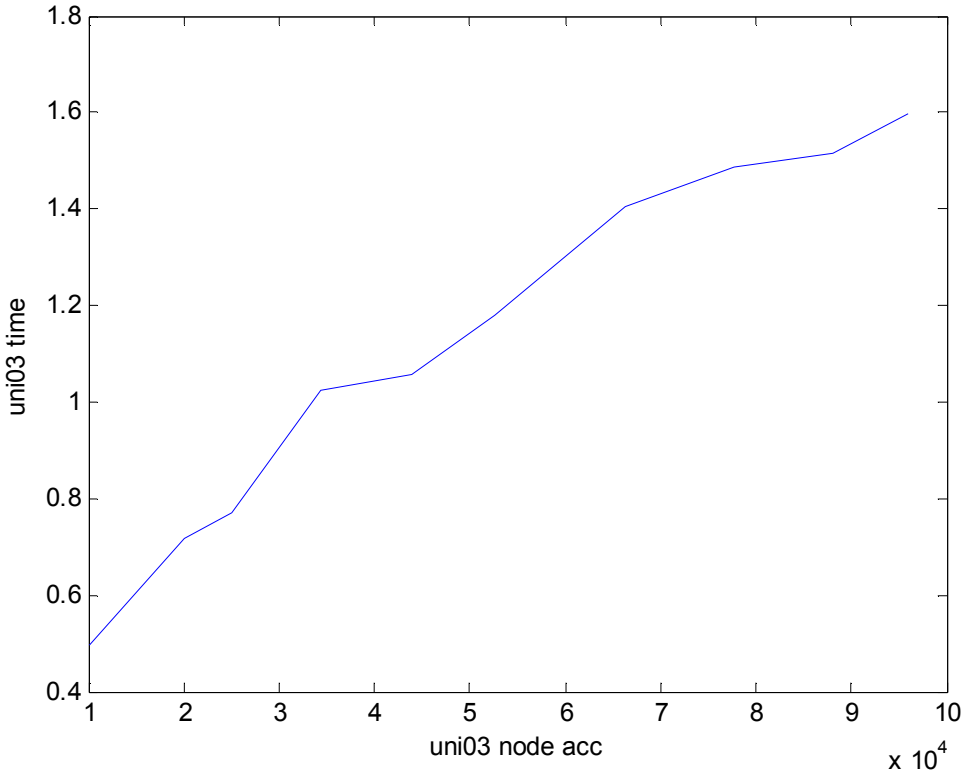

1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
pha03 time
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
pha03 node counter


1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
pha03 time
0
1
2
3
4
5
6
7
8
9
10
pha03 node acc
x 10^4

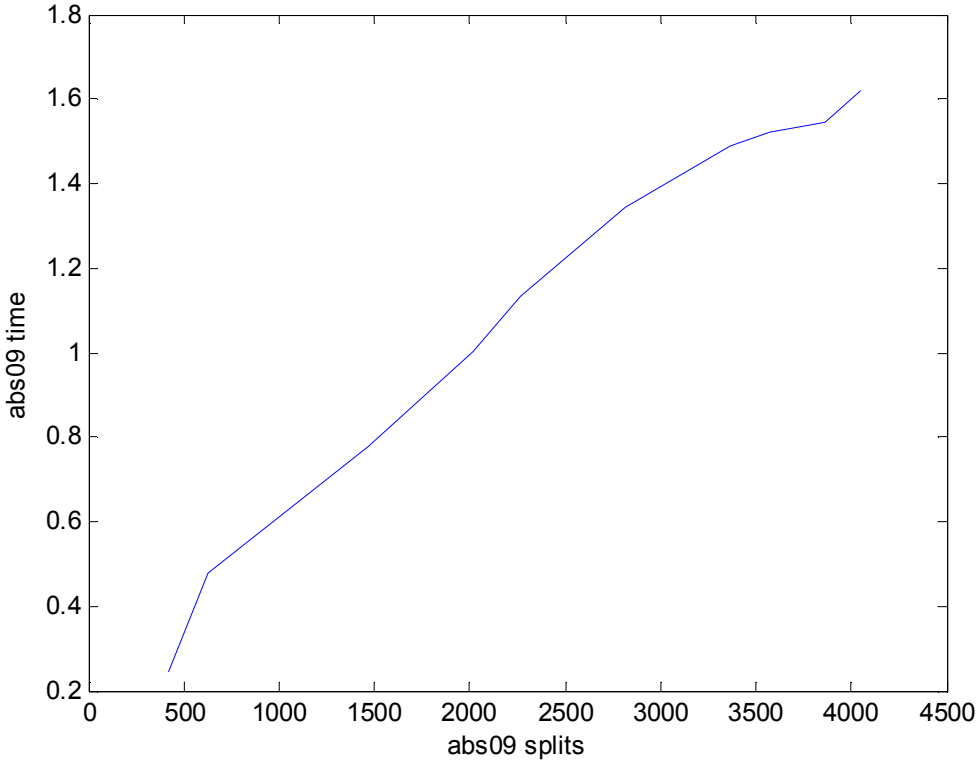
-*<u>Διανομή Uniform τριών διαστάσεων</u>*

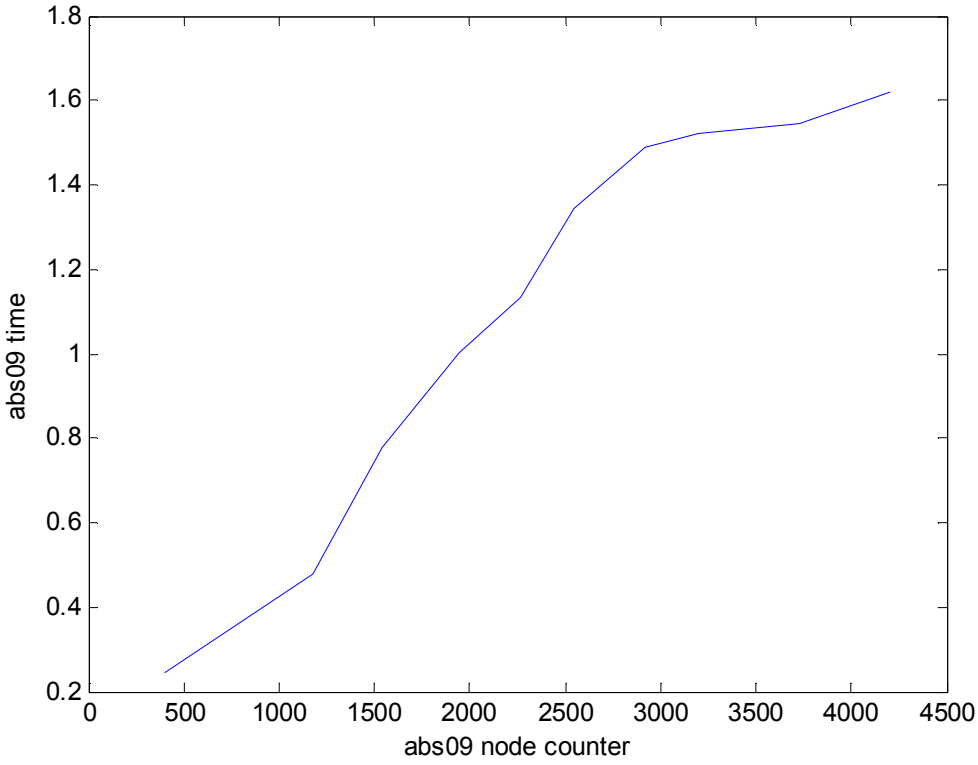


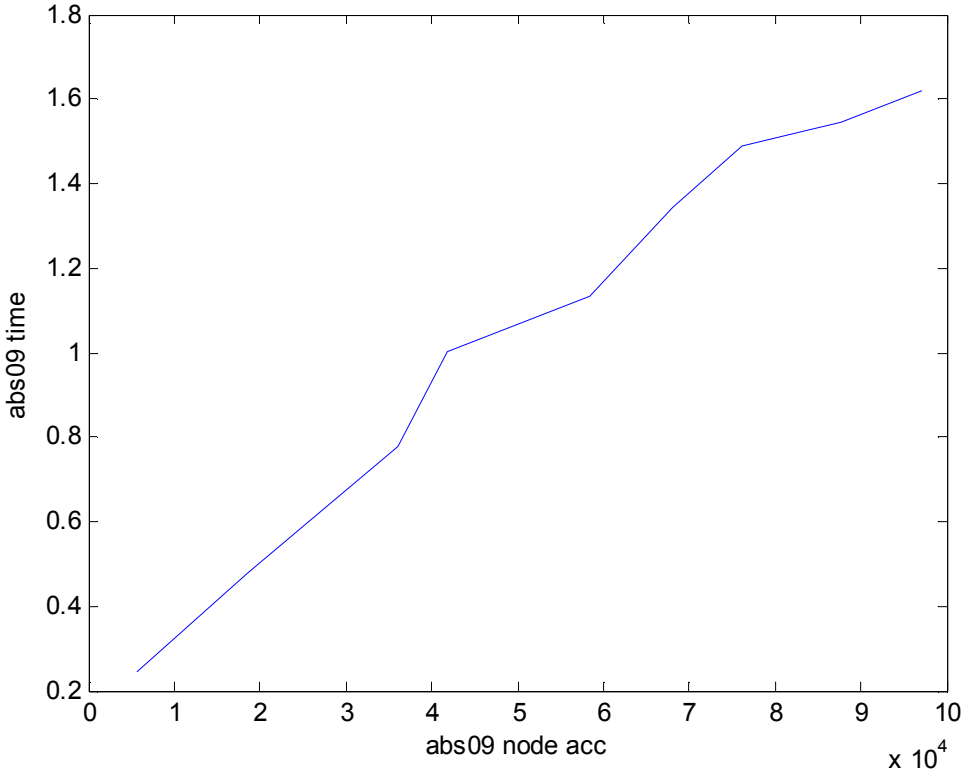




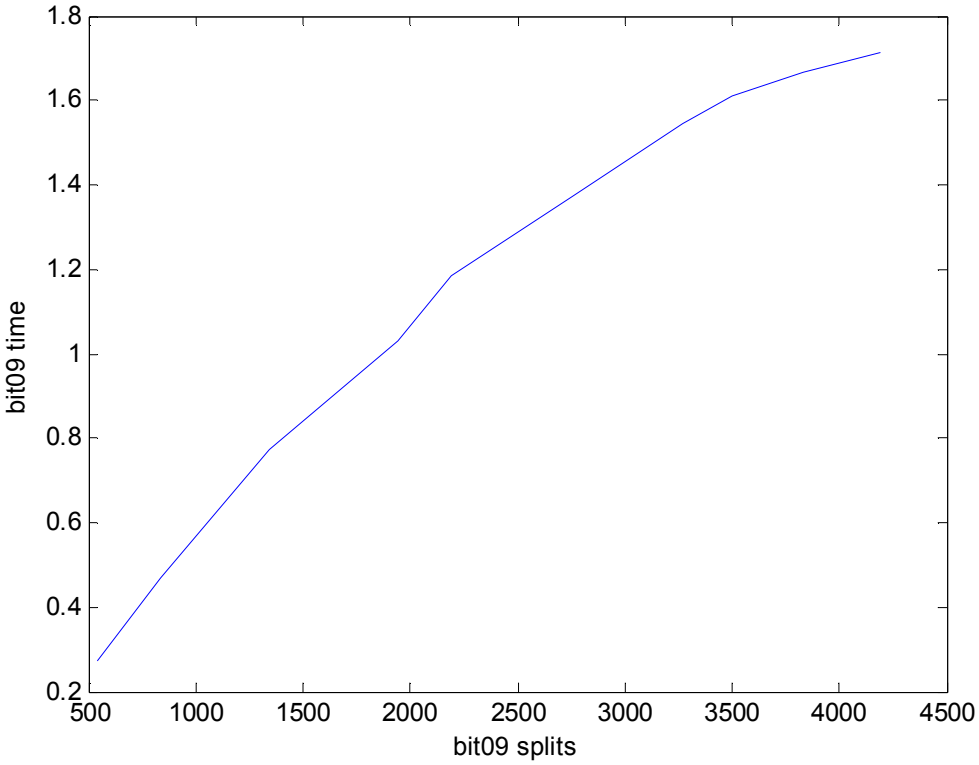
-*<u>Διανομή Absolute εννέα διαστάσεων</u>*

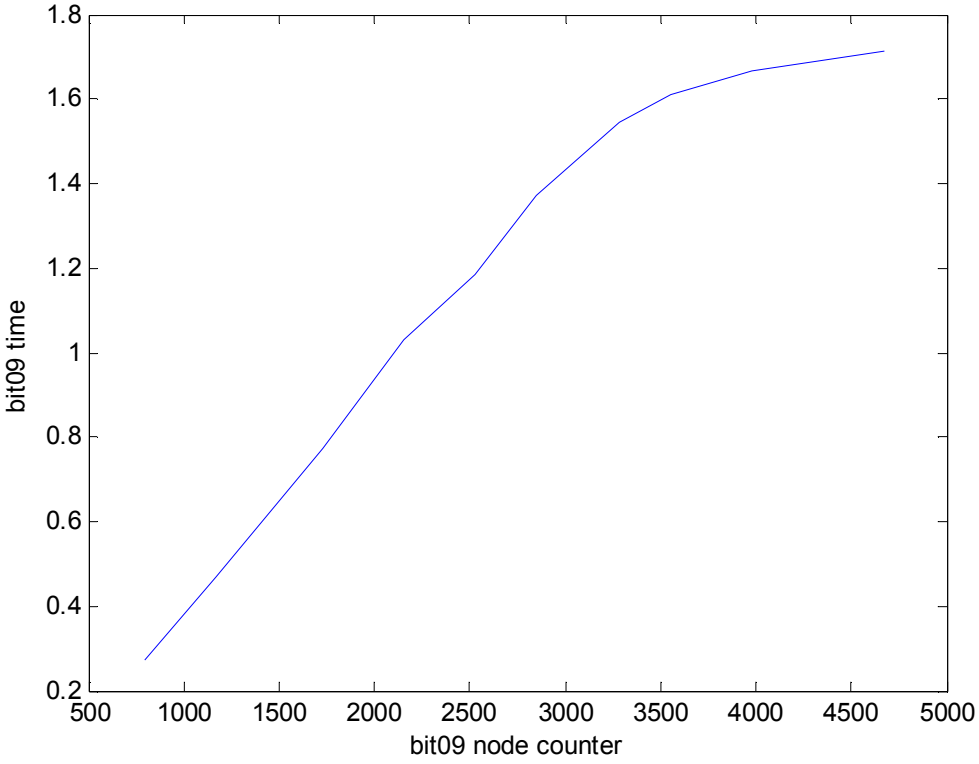


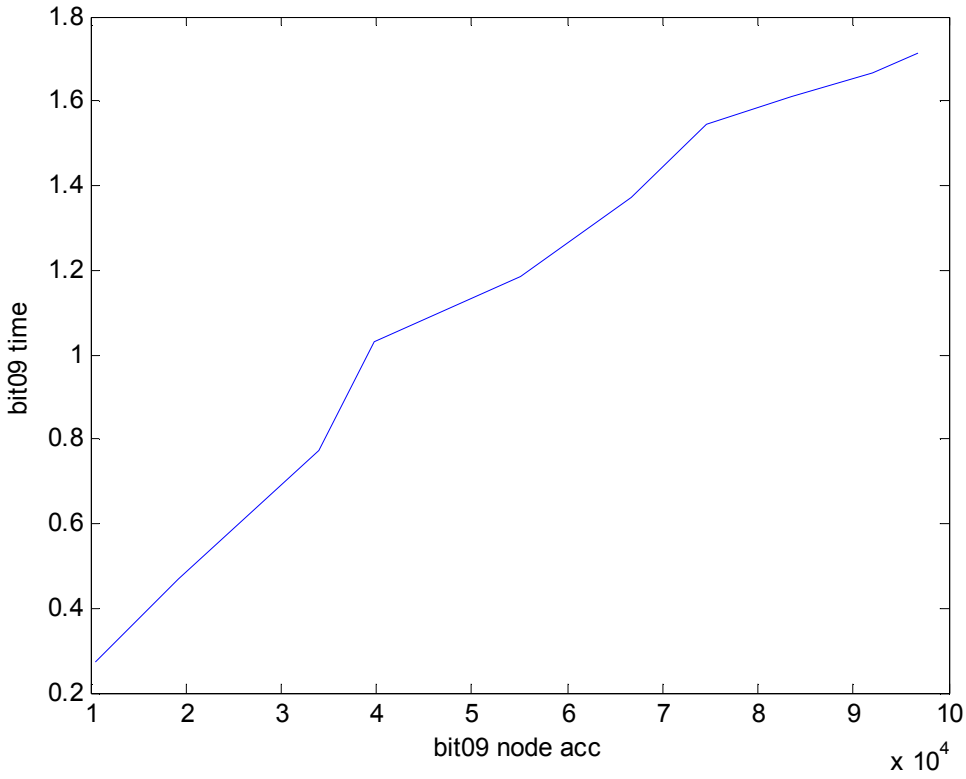

abs09 time
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
500
1000
1500
2000
2500
3000
3500
4000
4500
abs09 node counter


abs09 time
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
9
10
abs09 node acc
x 10^4

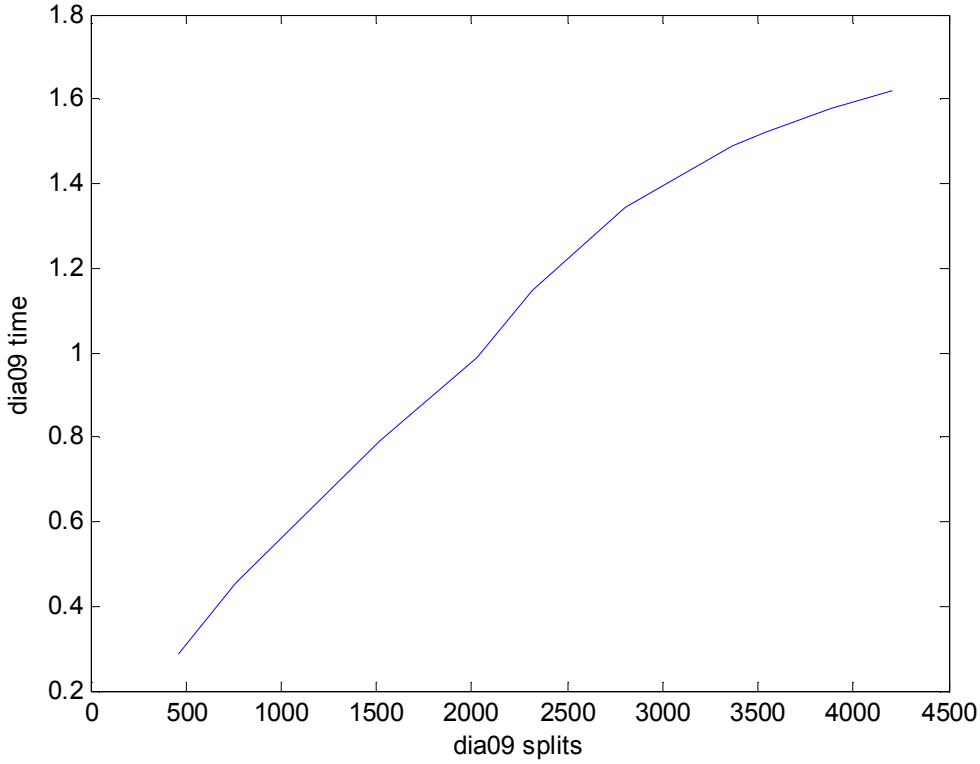
-*Διανομή Bit εννέα διαστάσεων*

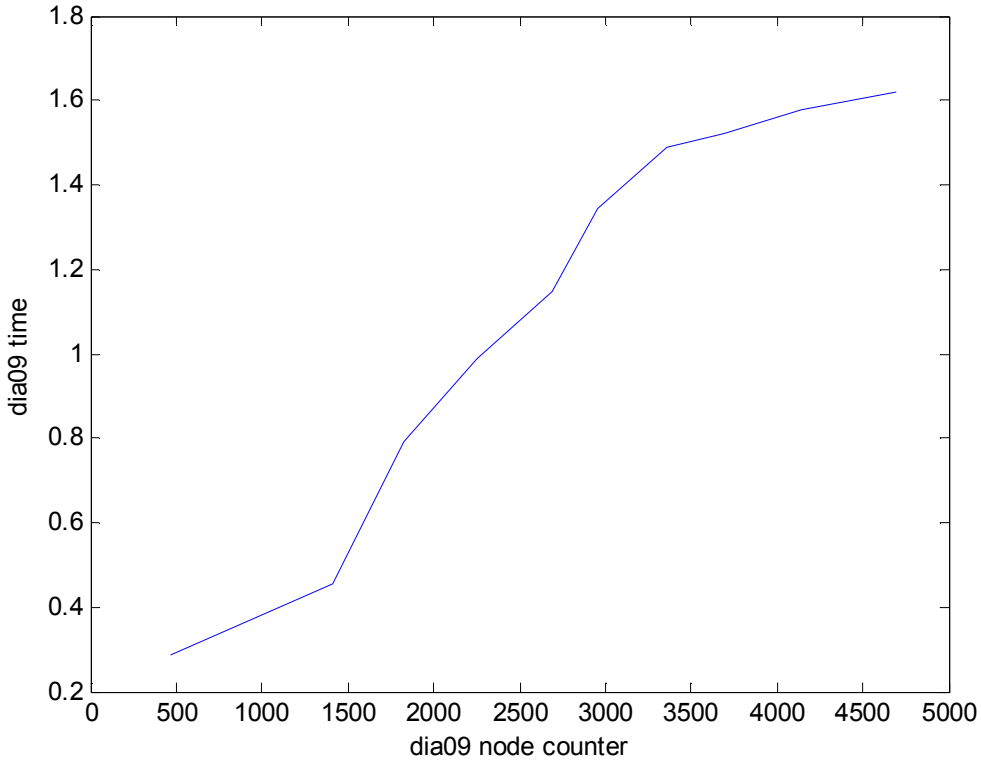


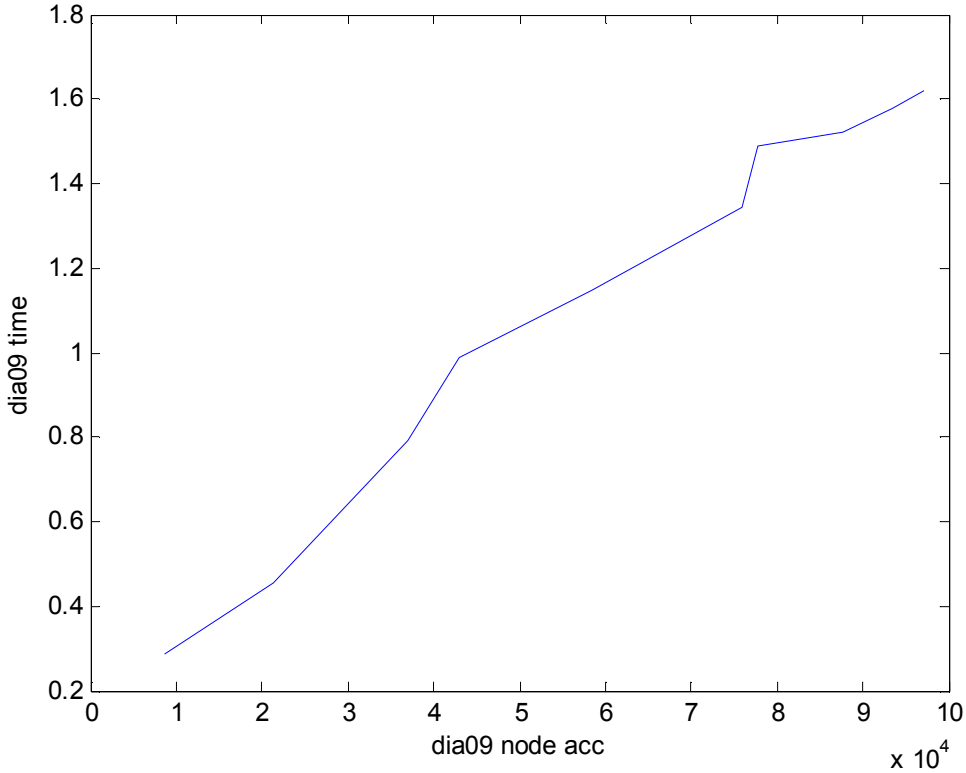




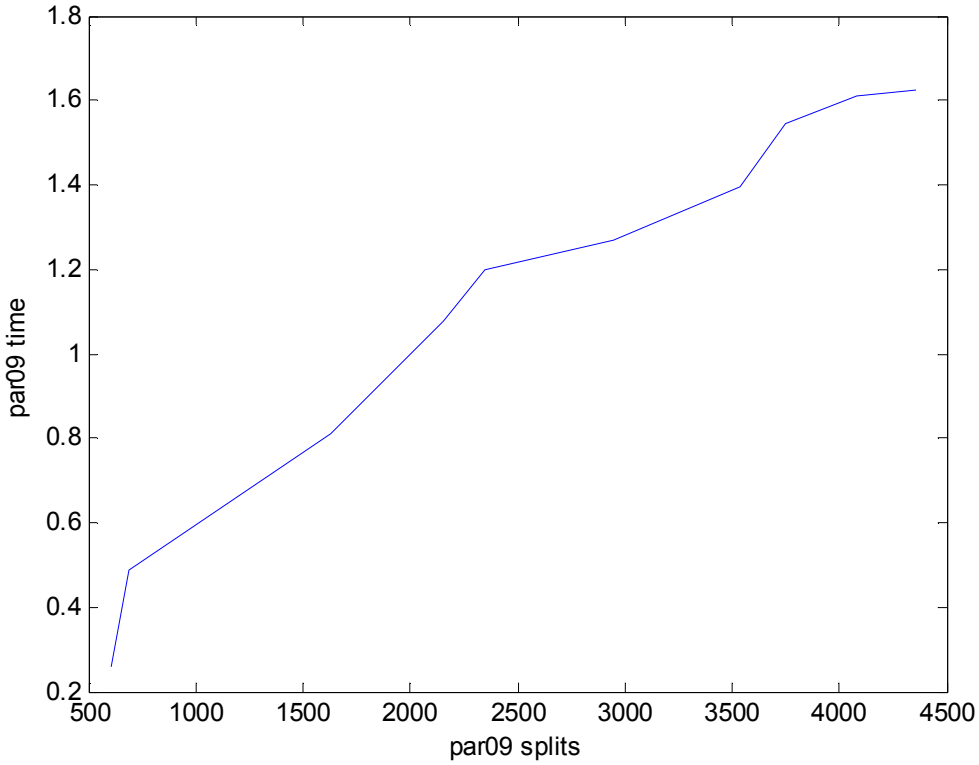
-*<u>Διανομή Diagonal εννέα διαστάσεων</u>*

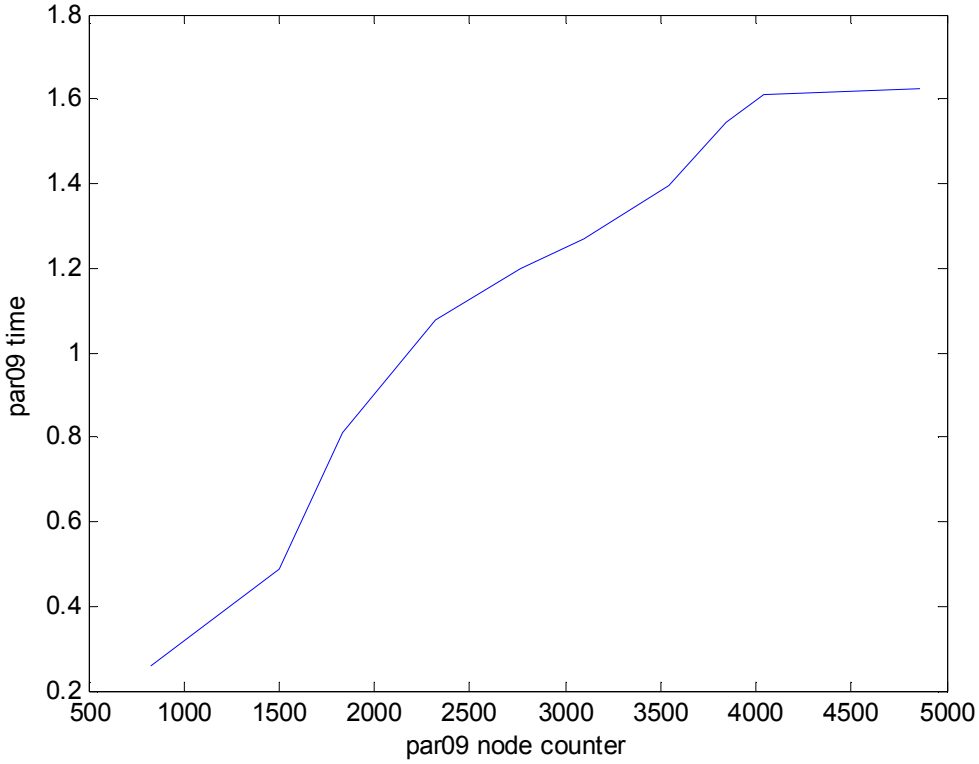


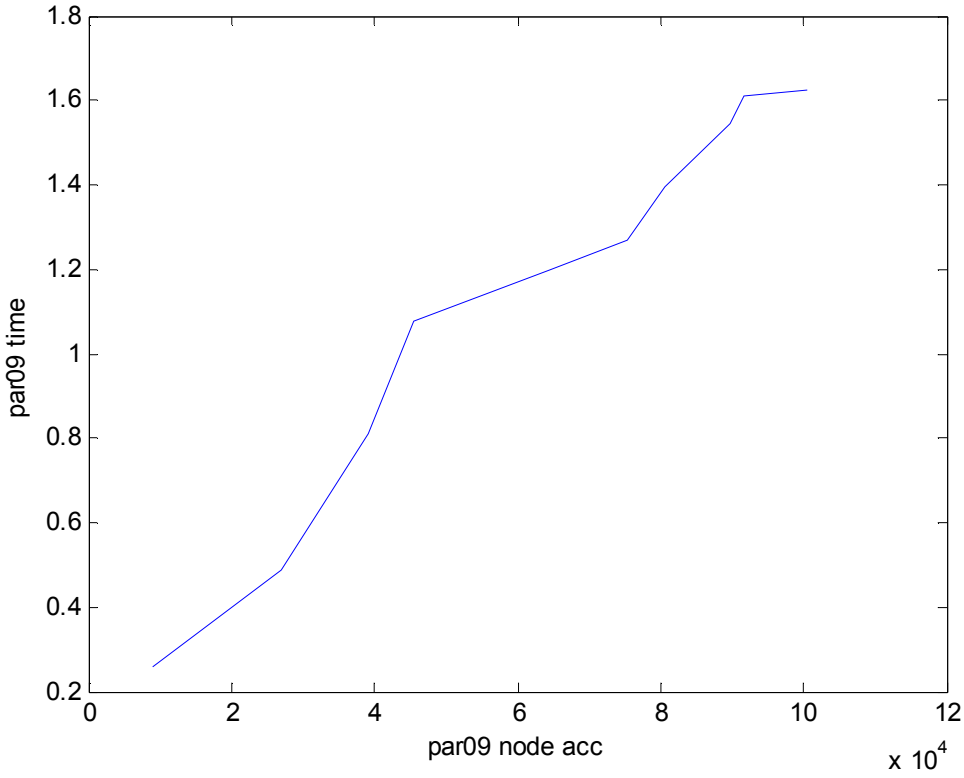

dia09 time
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
dia09 node counter


dia09 time
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
9
10
dia09 node acc
x 10^4

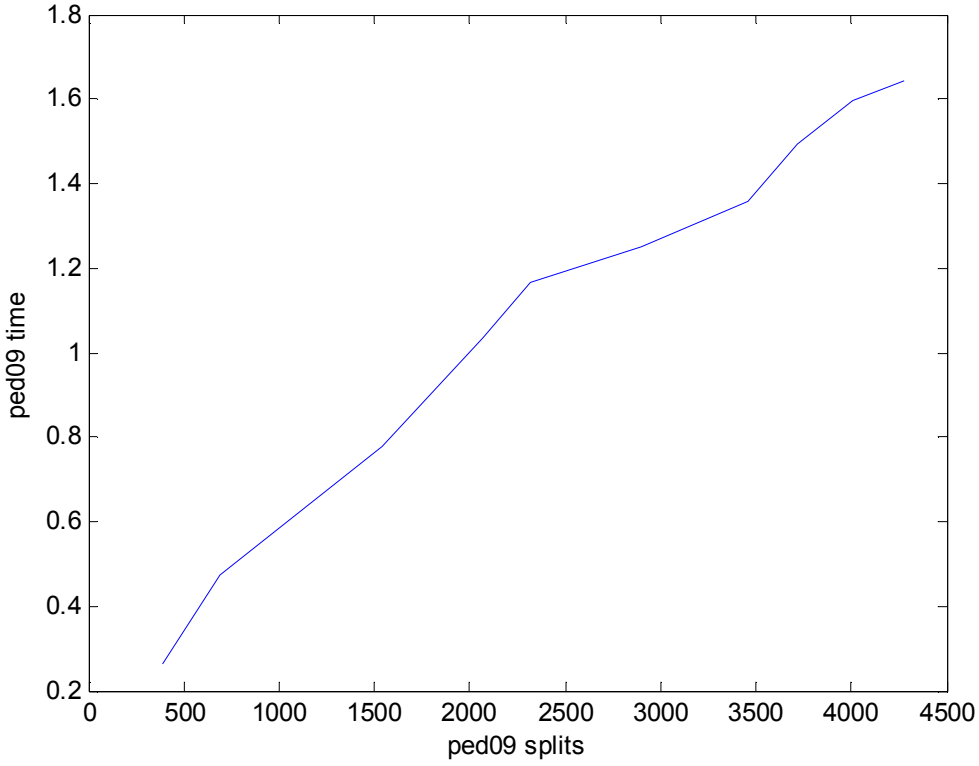
-*Διανομή Parsel εννέα διαστάσεων*

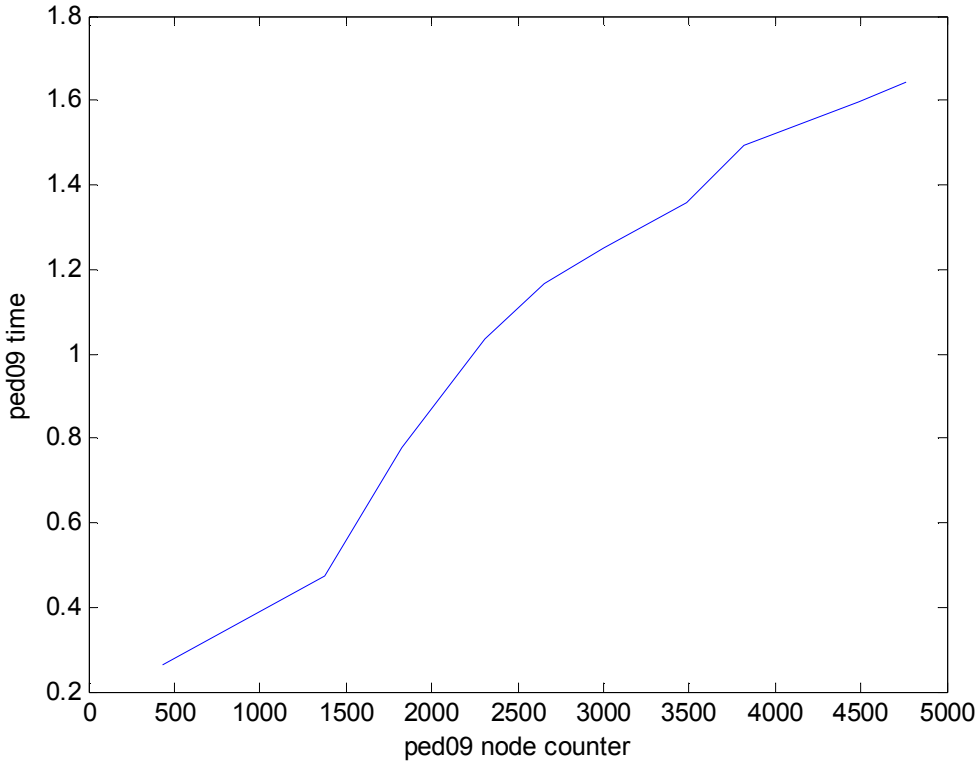


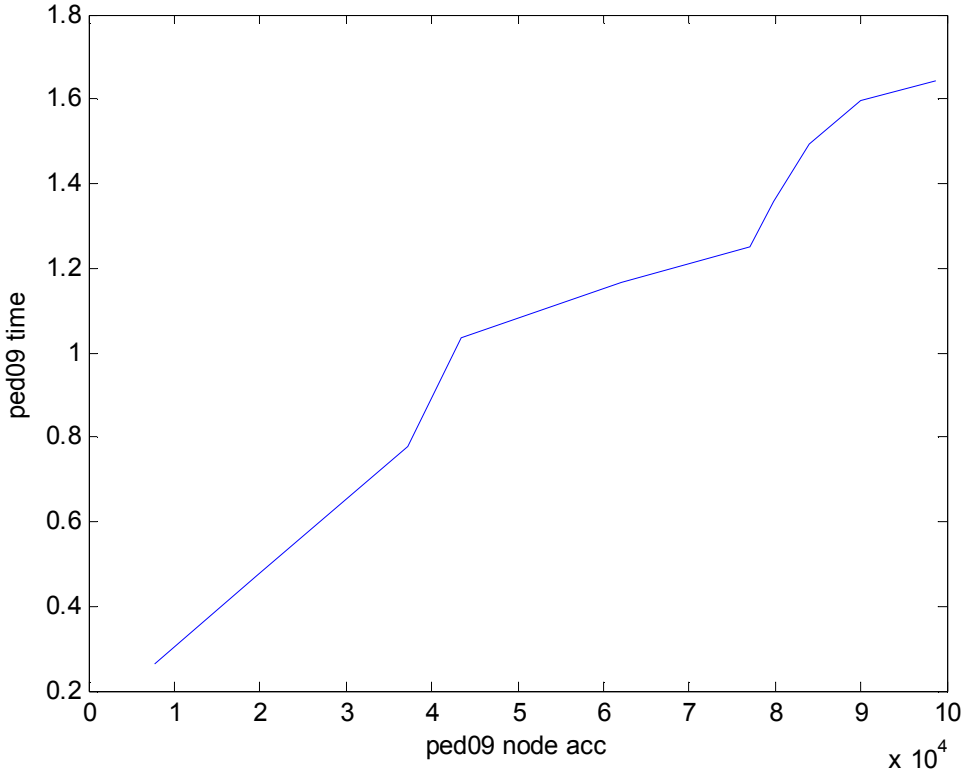




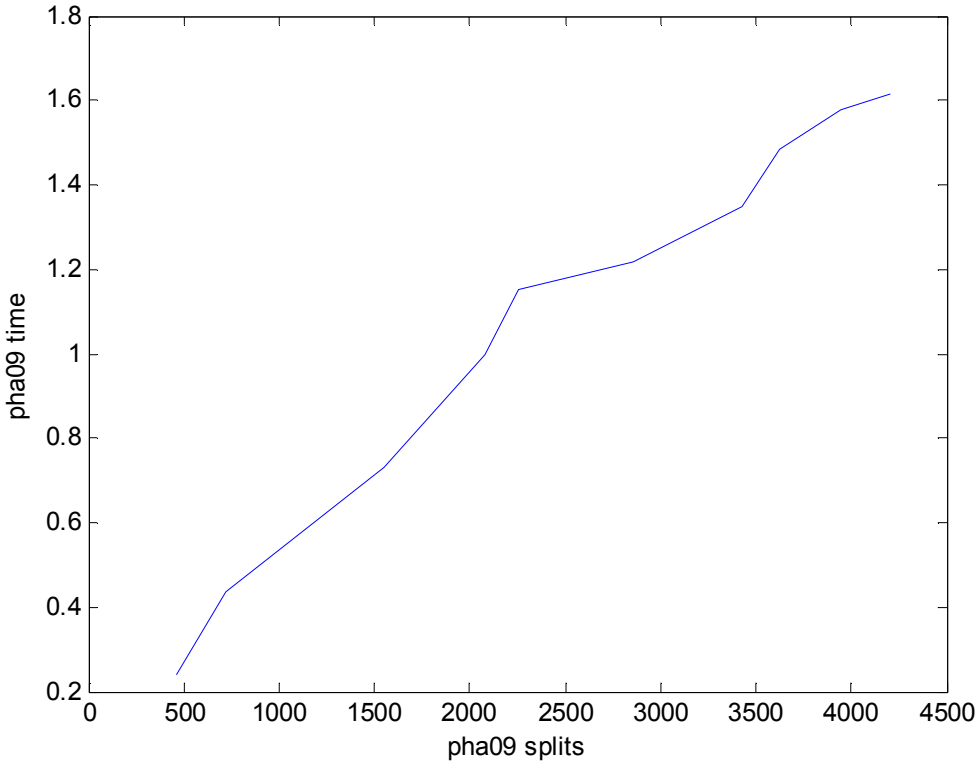
-*Διανομή P-edges εννέα διαστάσεων*

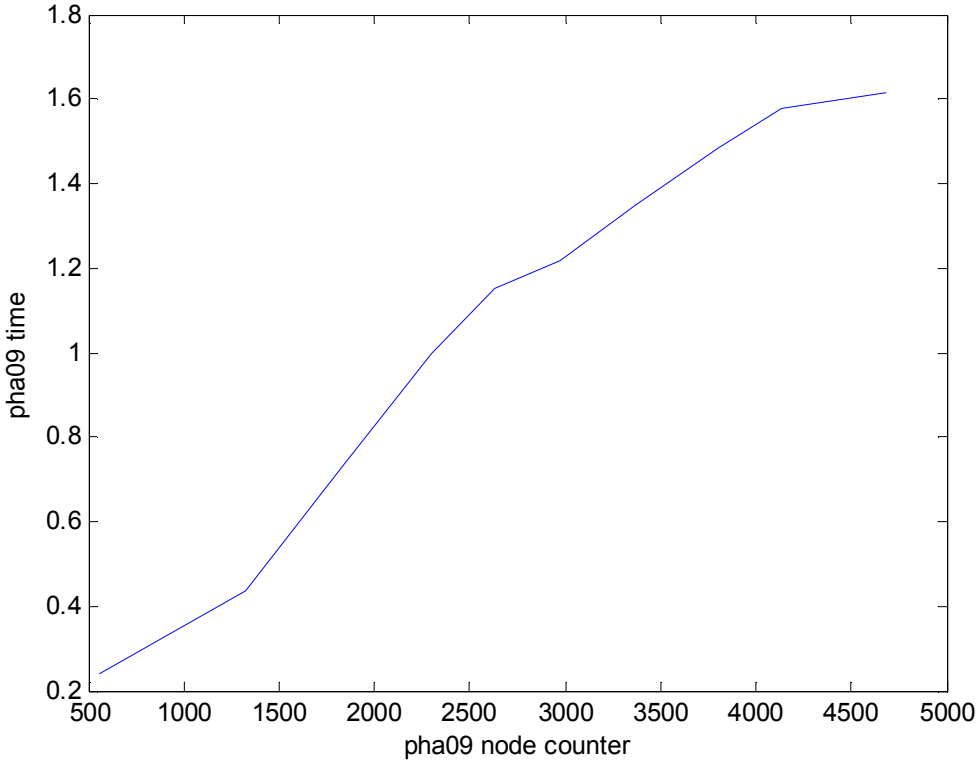


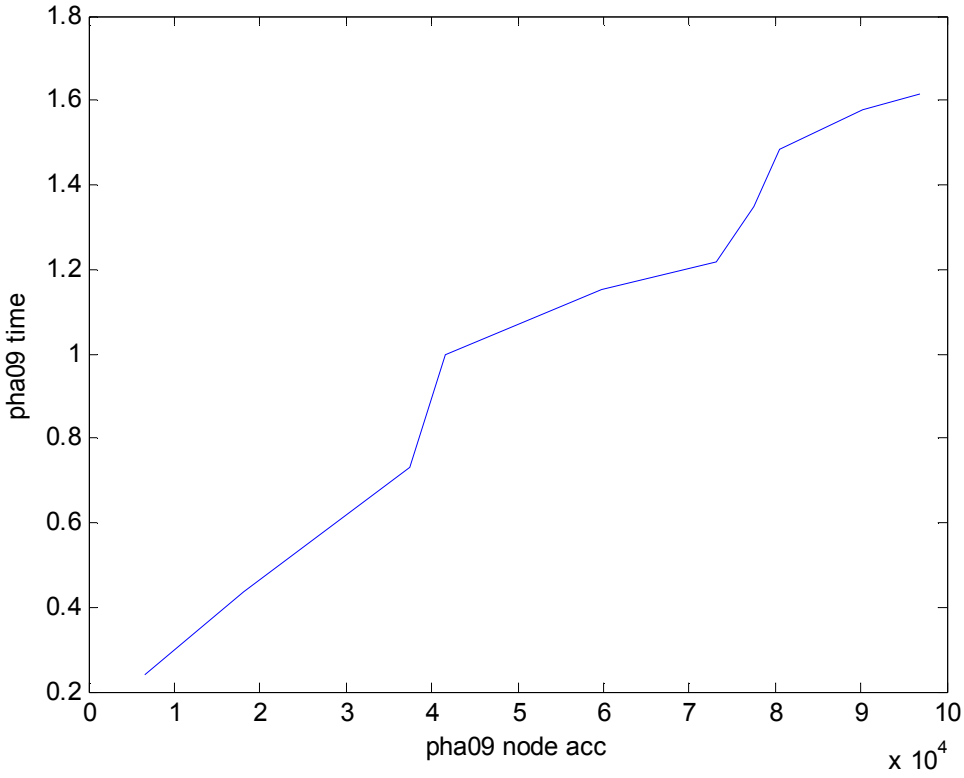

ped09 time
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
ped09 node counter


ped09 time
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
9
10
ped09 node acc
x 10^4

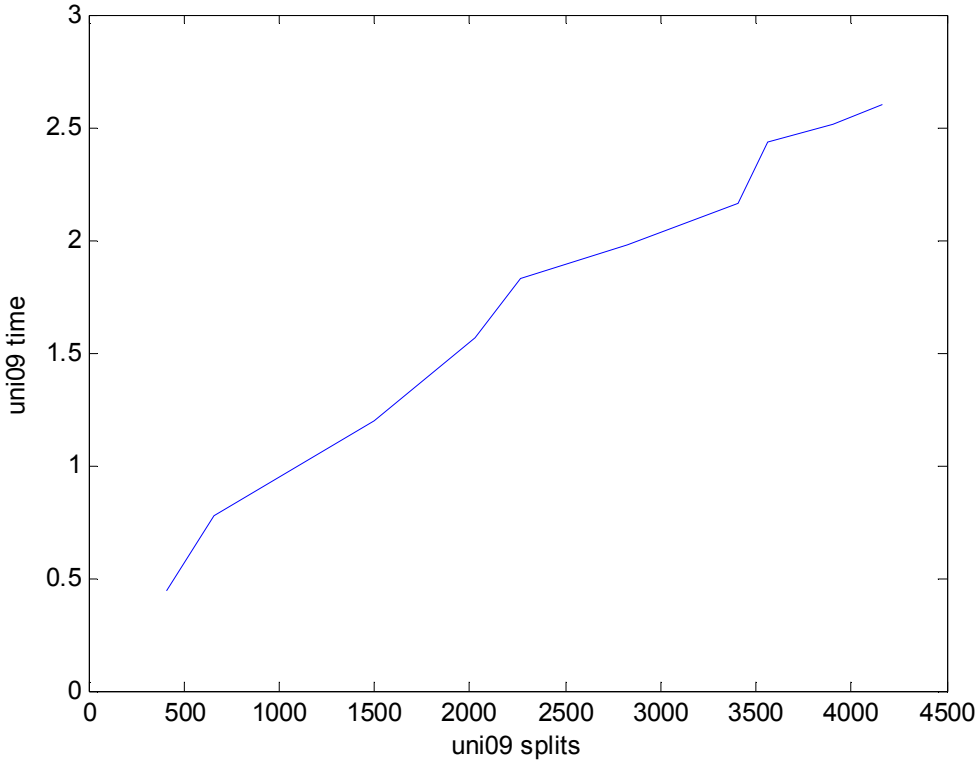
-*Διανομή P-haze εννέα διαστάσεων*

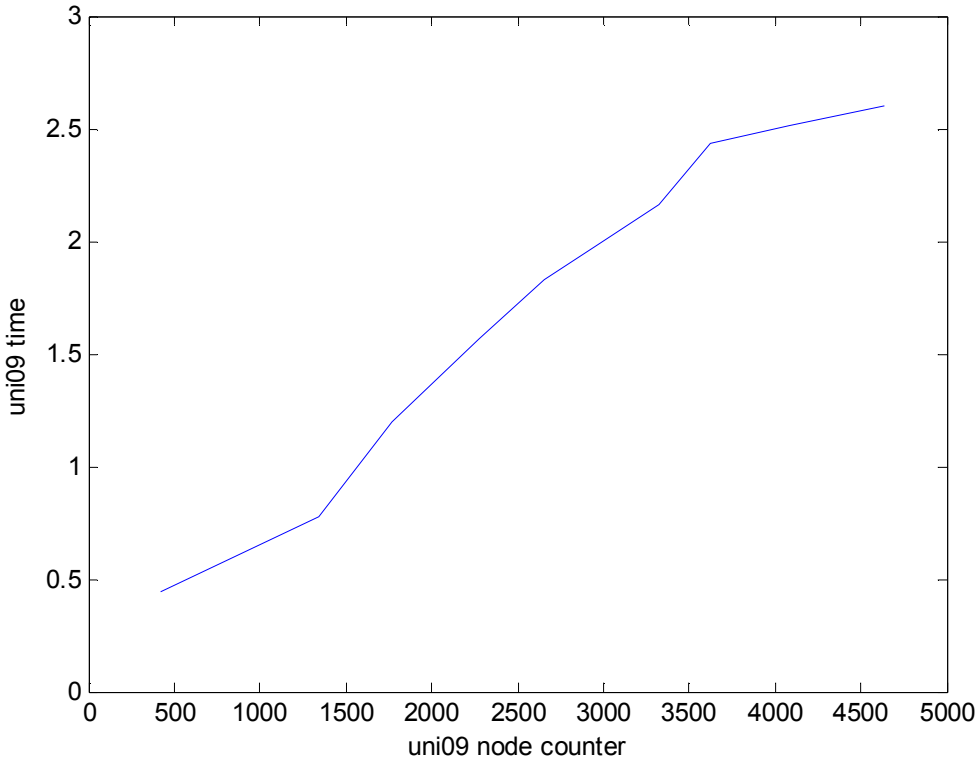


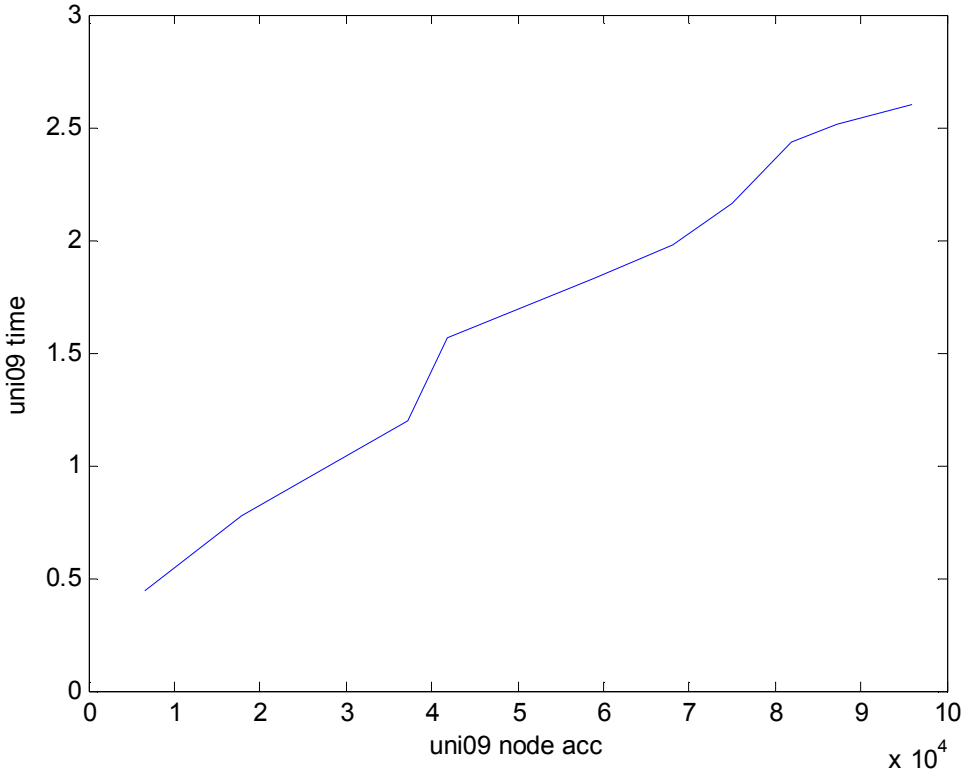




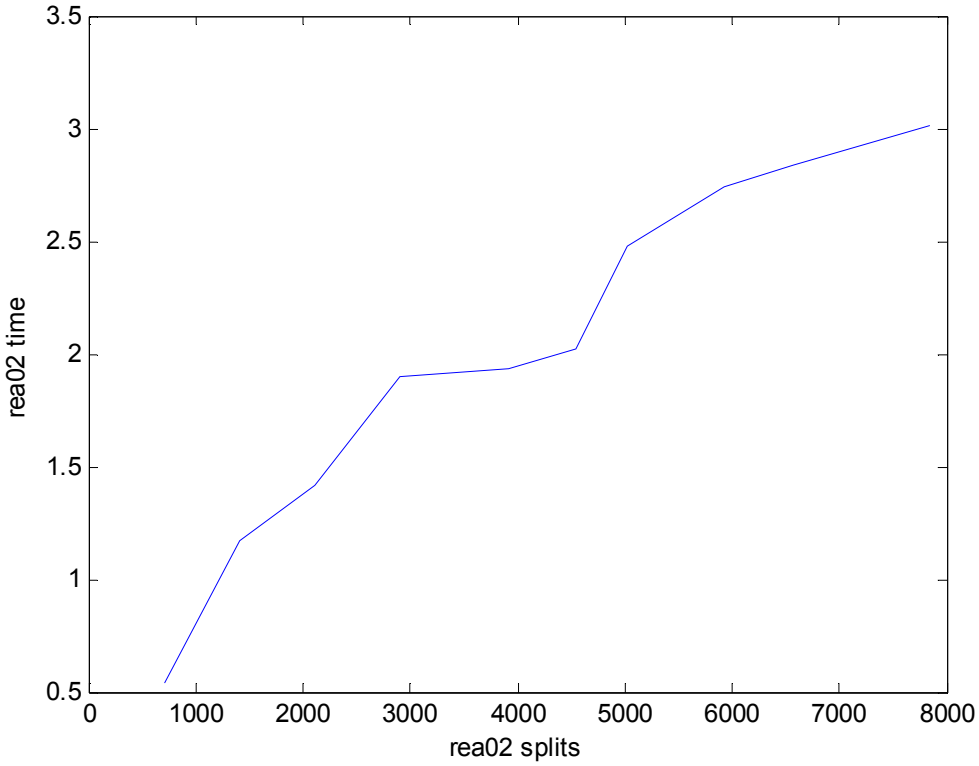
-*<u>Διανομή Uniform εννέα διαστάσεων</u>*

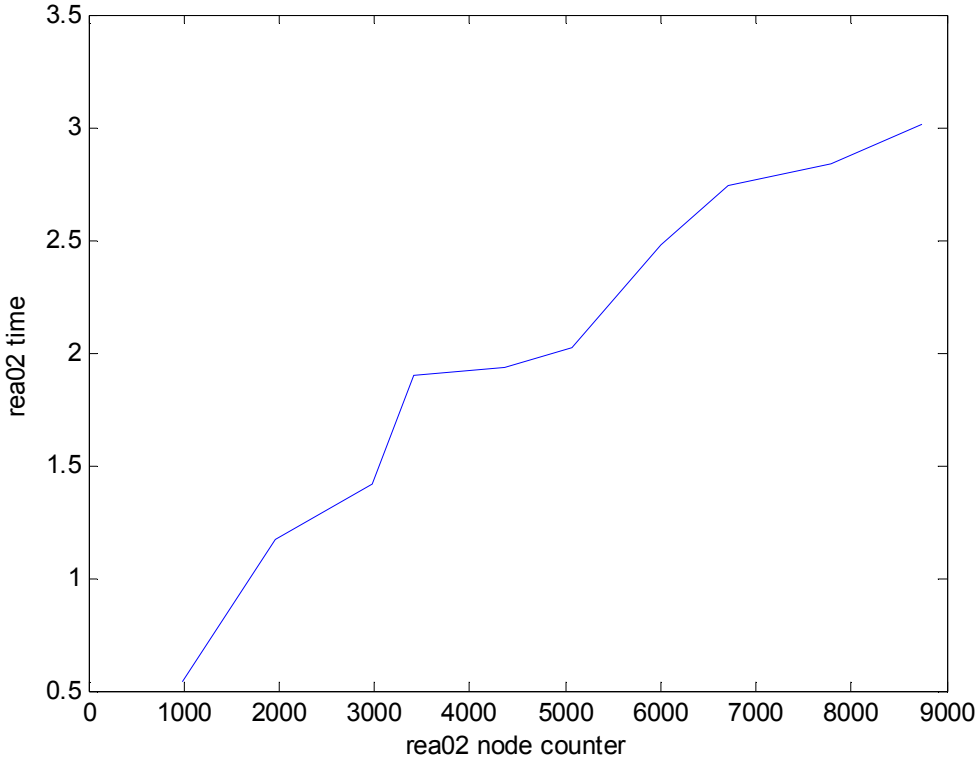


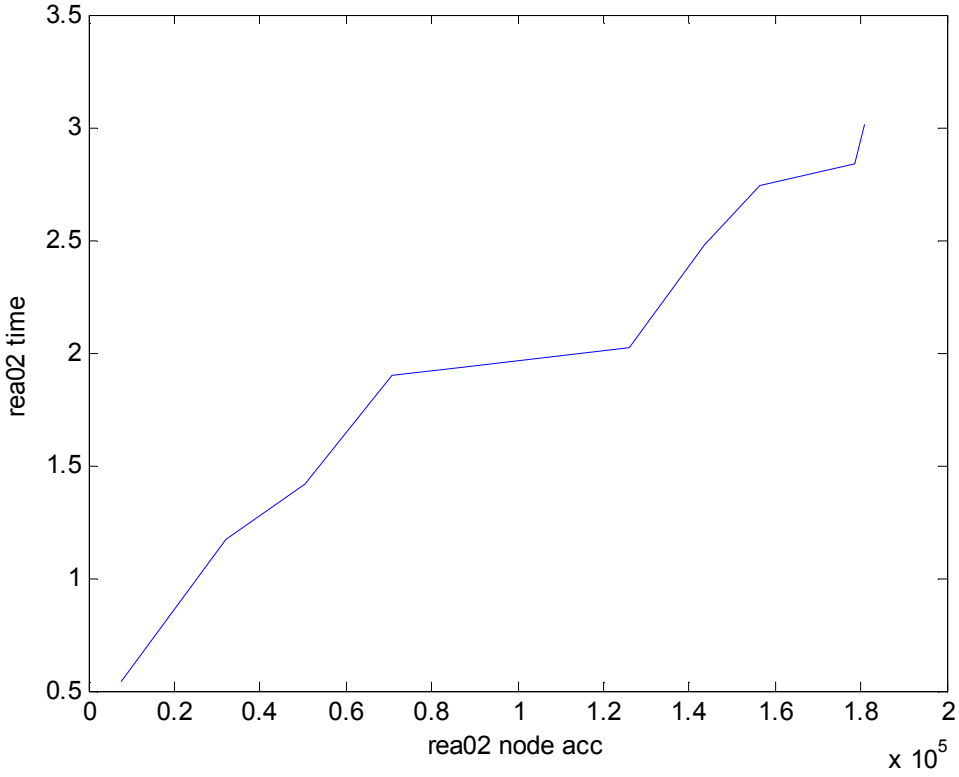

uni09 time
uni09 node counter
0
0.5
1
1.5
2
2.5
3
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000


uni09 time
uni09 node acc
x 10^4
0
0.5
1
1.5
2
2.5
3
0
1
2
3
4
5
6
7
8
9
10

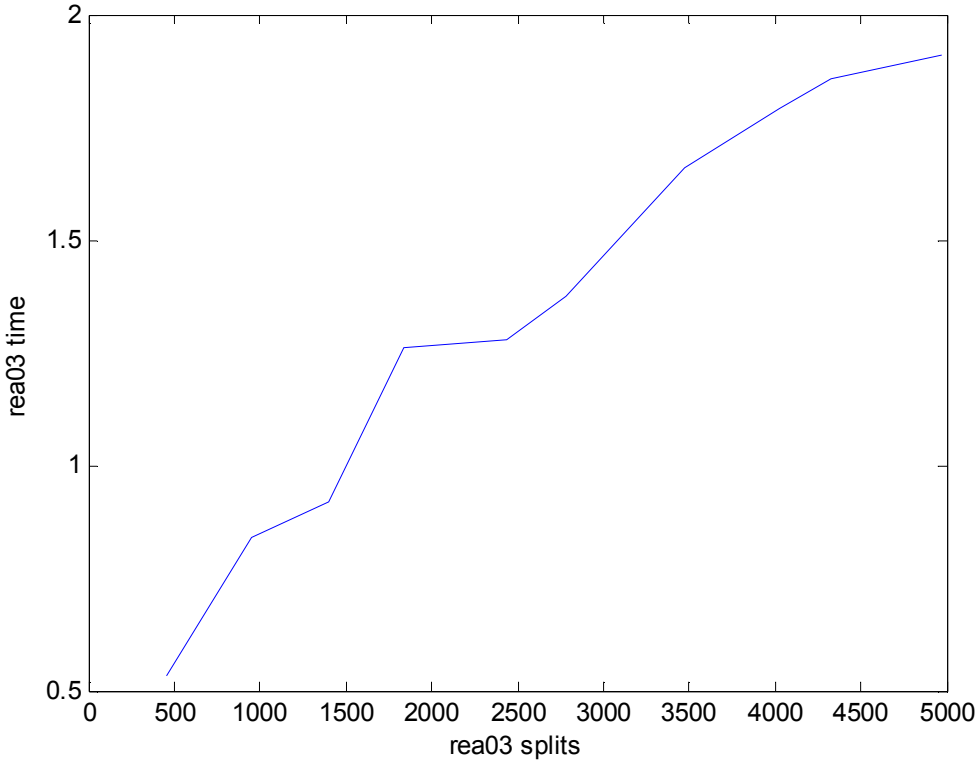
-*Διανομή Real δύο διαστάσεων*

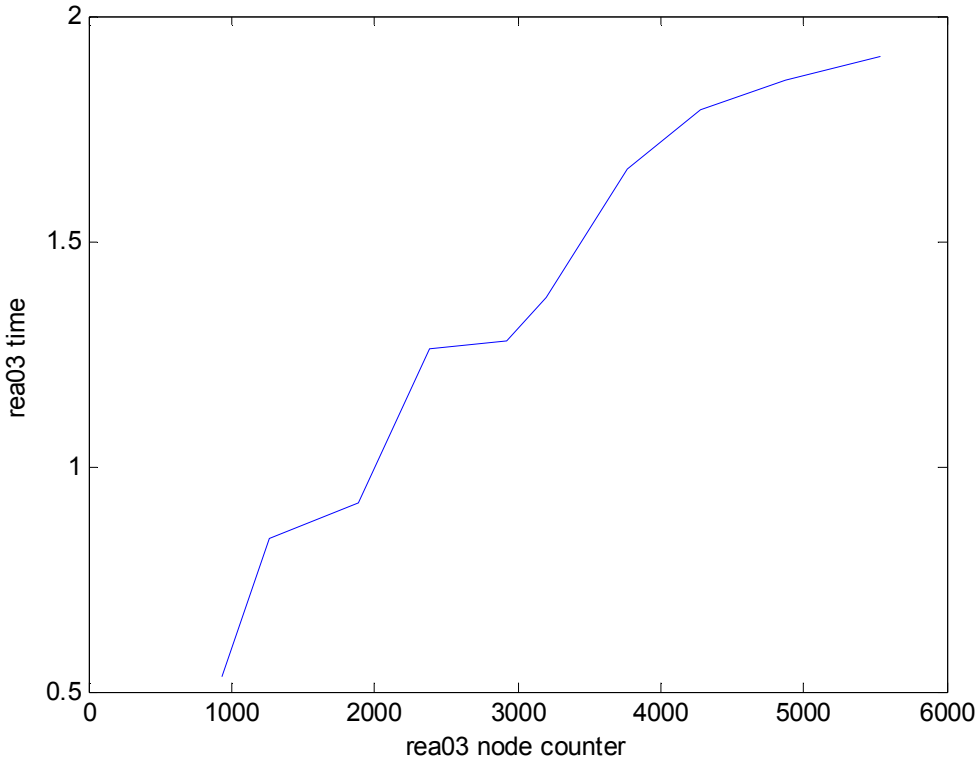


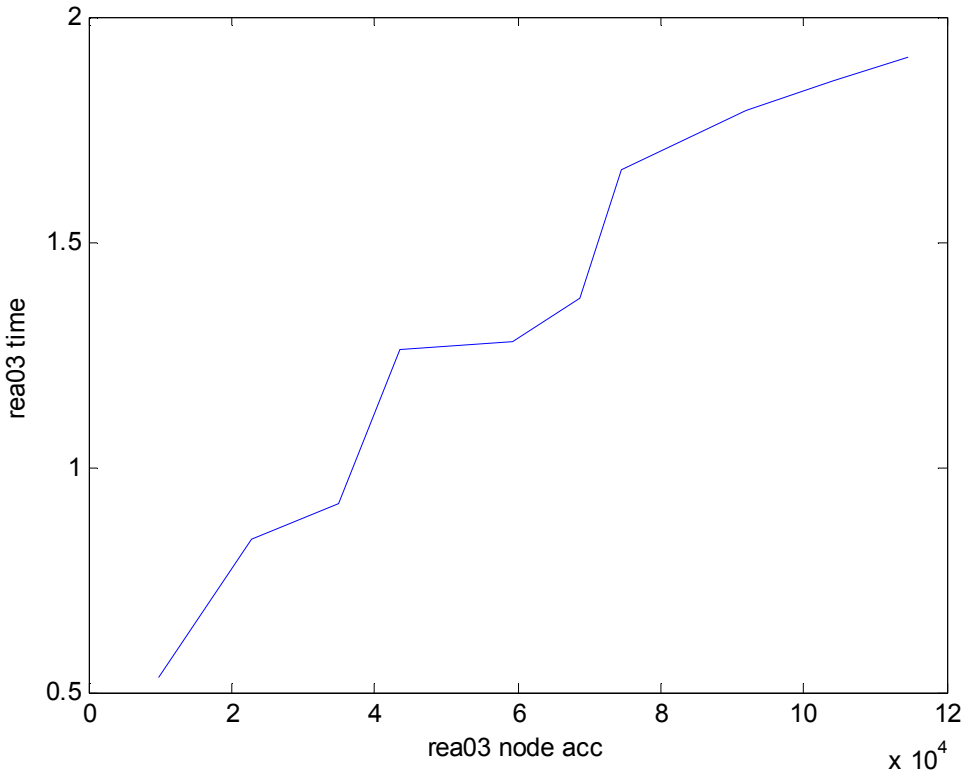




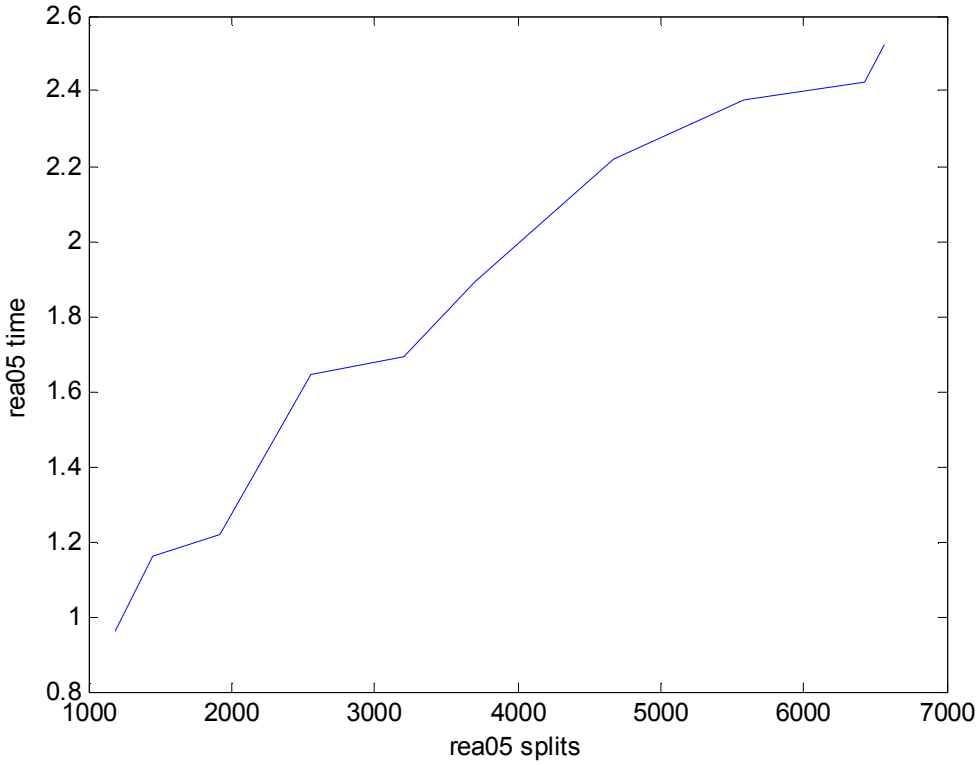
-*<u>Διανομή Real τριών διαστάσεων</u>*

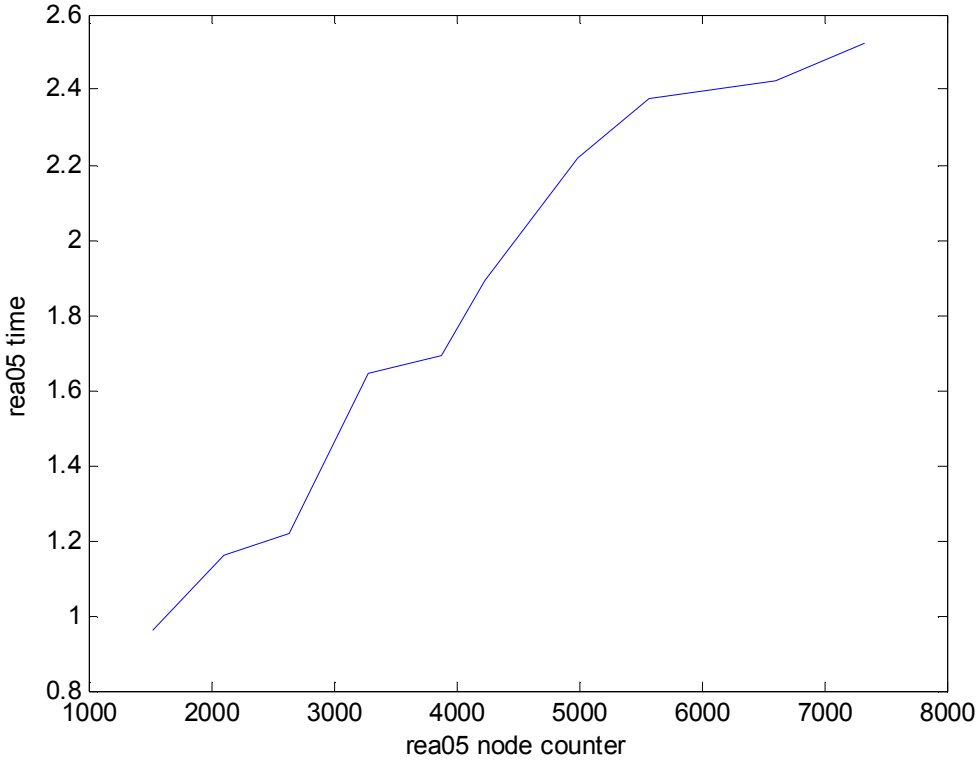


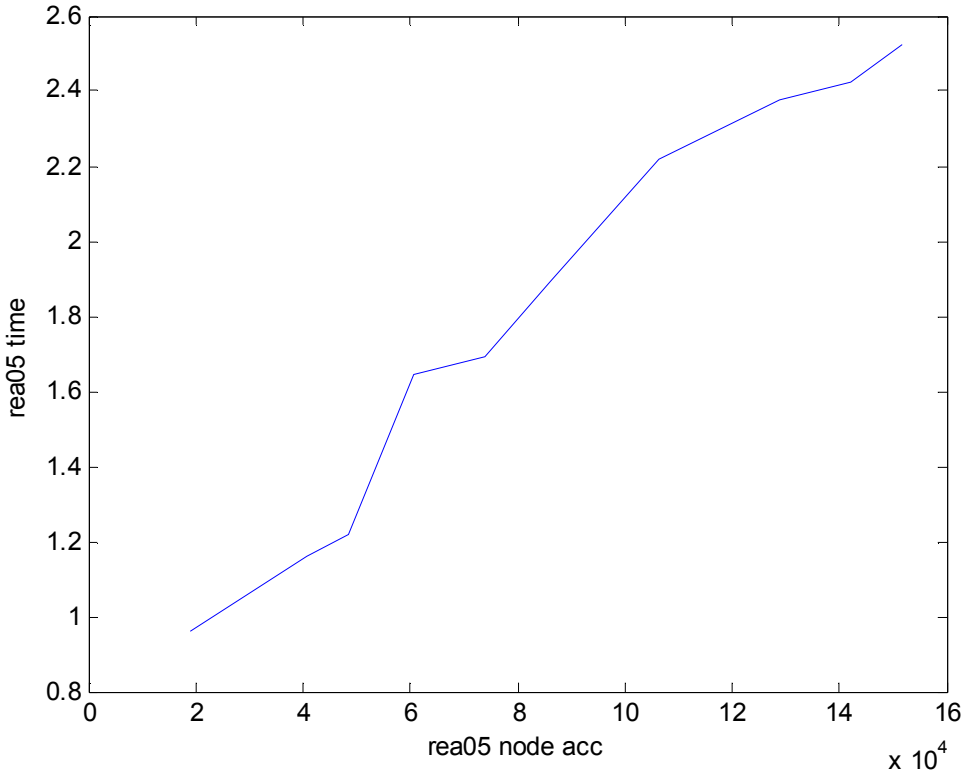

2
1.5
1
0.5
rea03 time
0
1000
2000
3000
4000
5000
6000
rea03 node counter


2
1.5
1
0.5
rea03 time
0
2
4
6
8
10
12
rea03 node acc
x 10^4

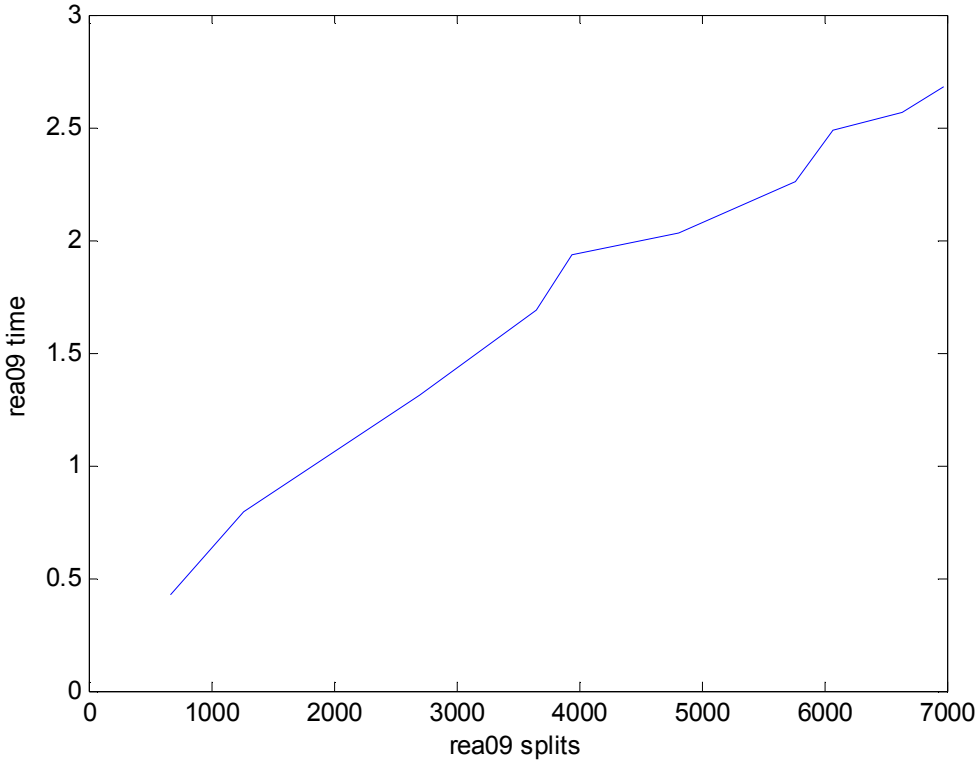
-*Διανομή Real πέντε διαστάσεων*

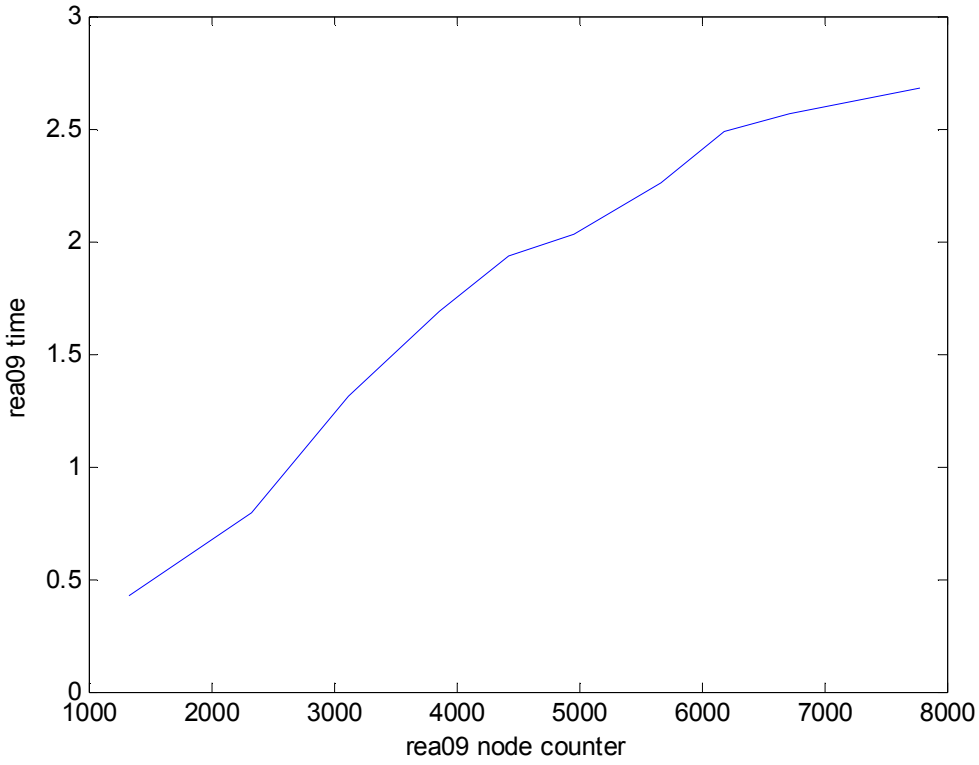


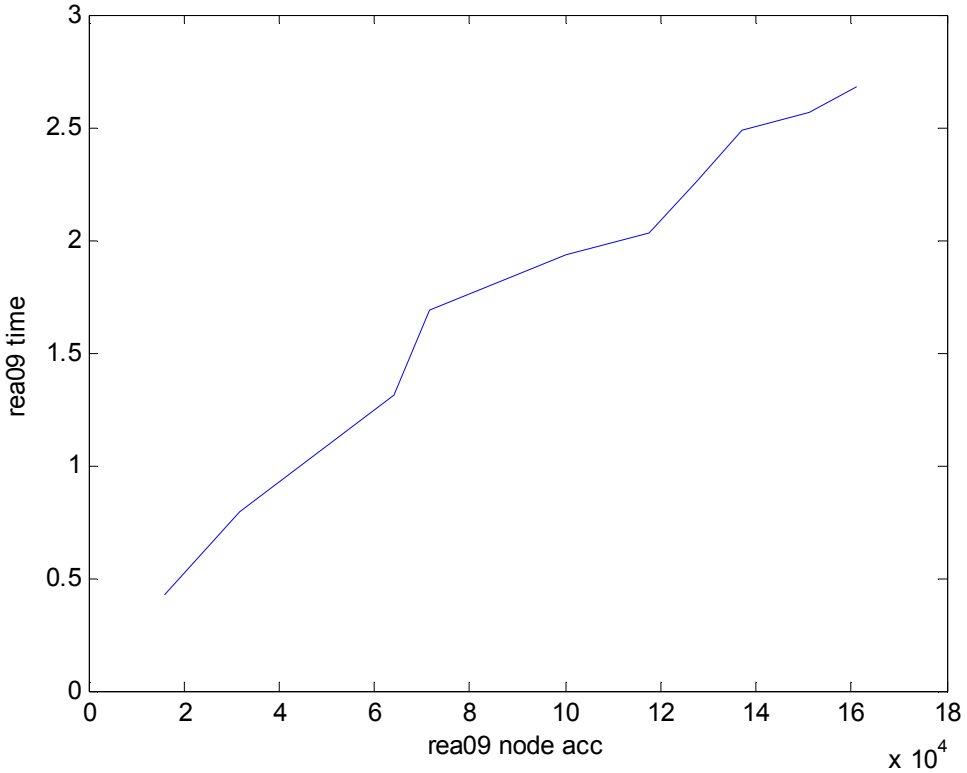




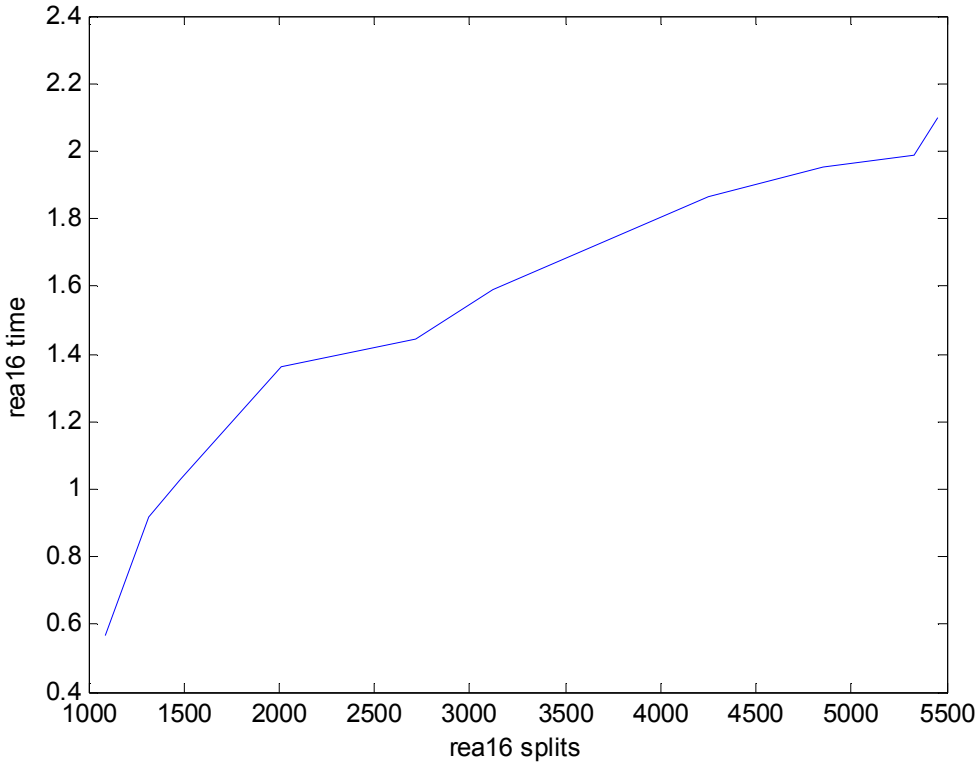
-*Διανομή Real εννέα διαστάσεων*

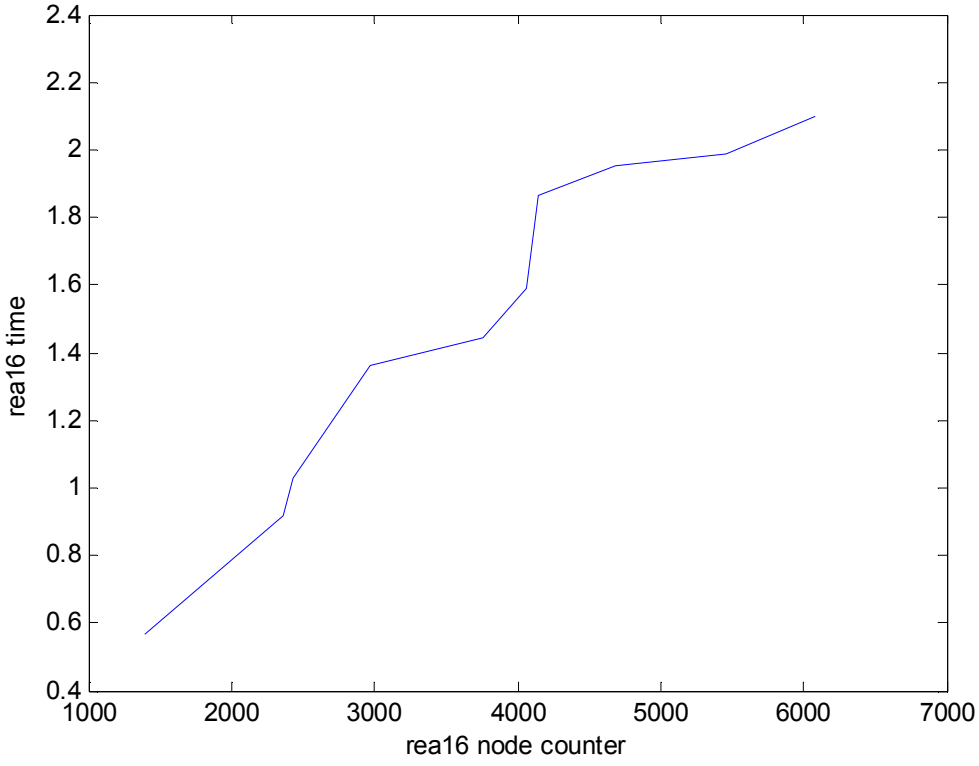


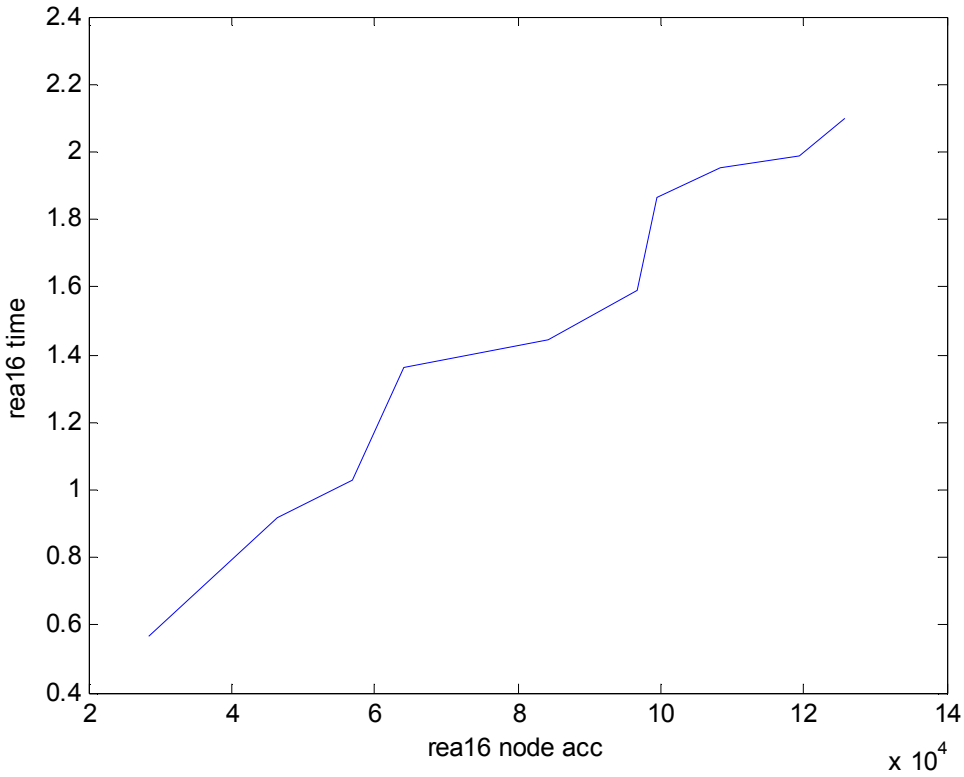

rea09 time
rea09 node counter
0
0.5
1
1.5
2
2.5
3
1000
2000
3000
4000
5000
6000
7000
8000


rea09 time
rea09 node acc
x 10^4
0
0.5
1
1.5
2
2.5
3
0
2
4
6
8
10
12
14
16
18

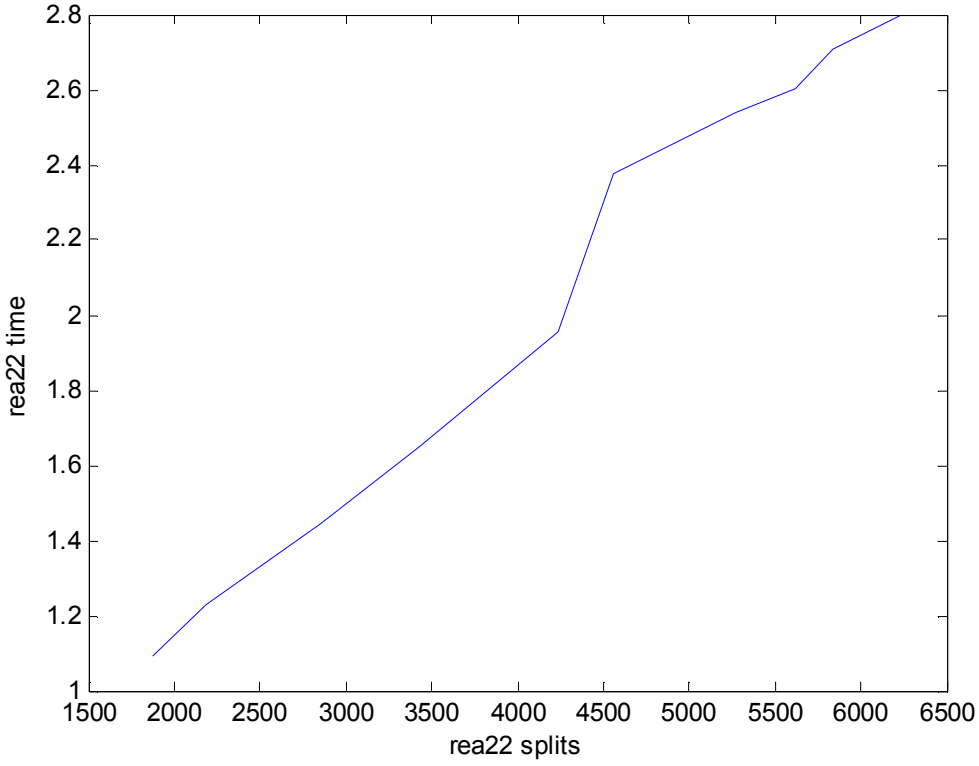
-*Διανομή Real δεκαέξι διαστάσεων*

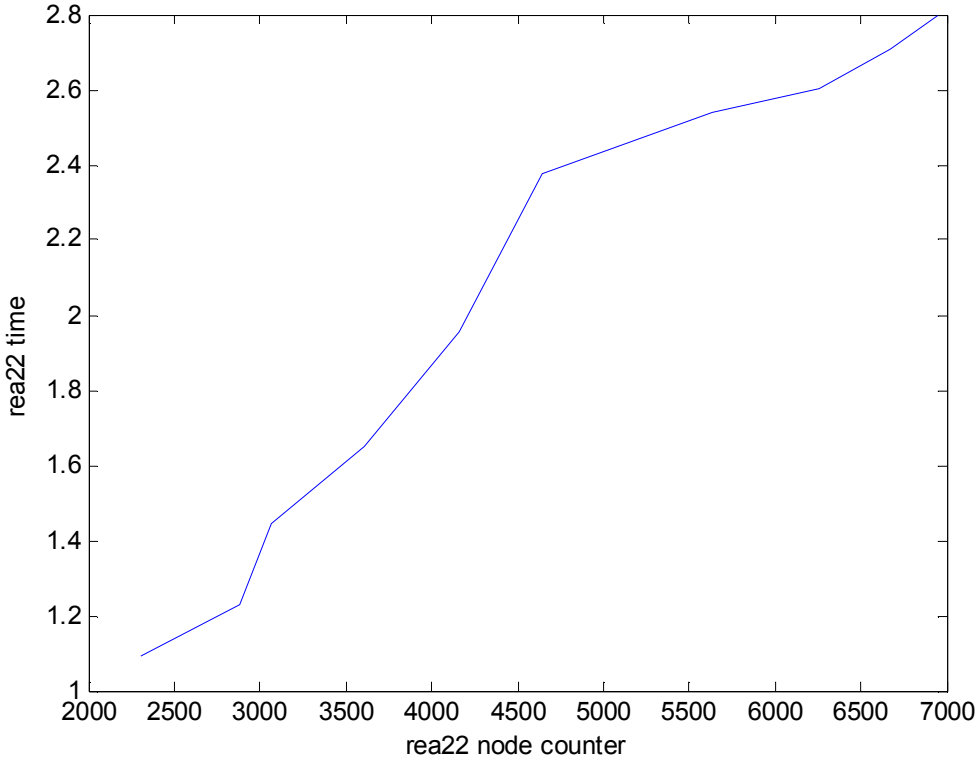


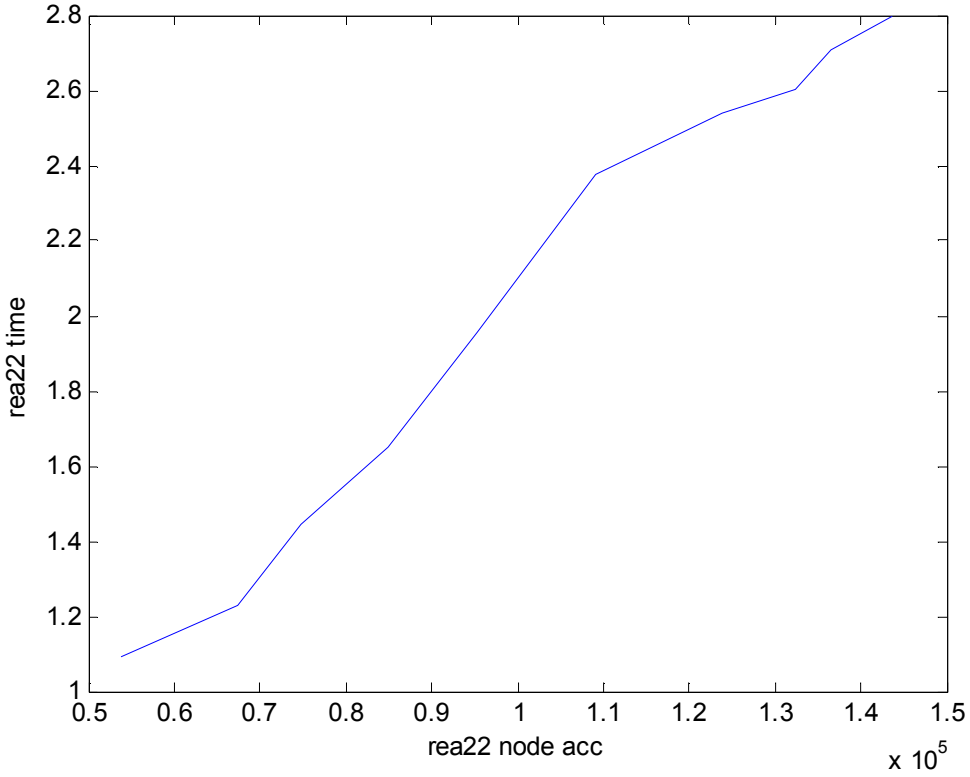




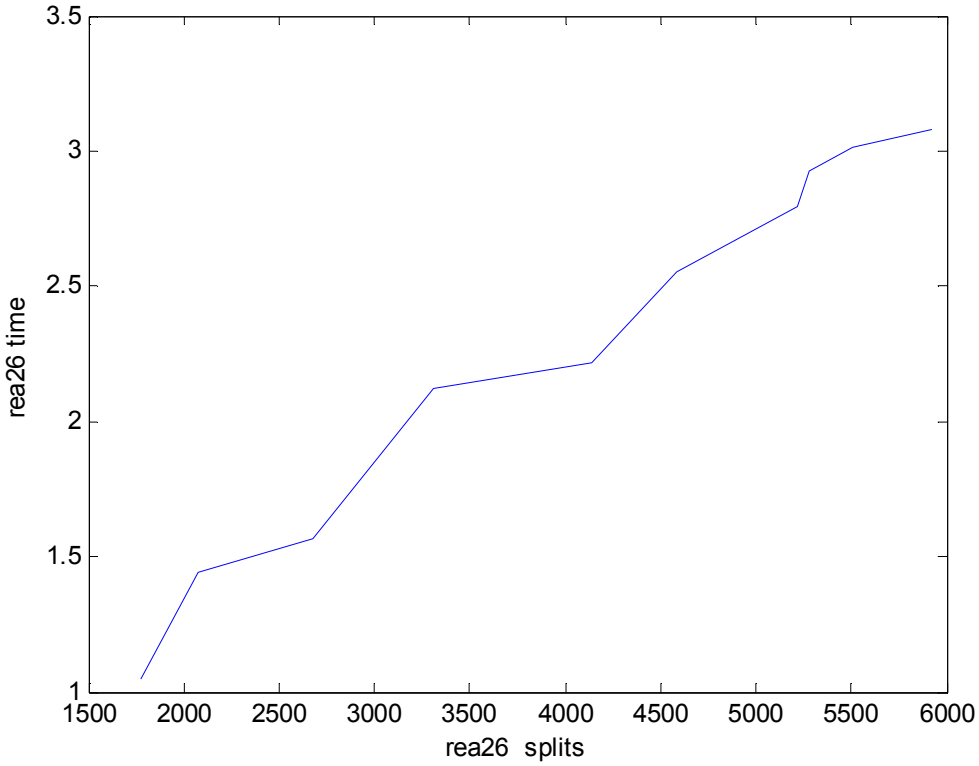
-*<u>Διανομή Real εικοσιδύο διαστάσεων</u>*

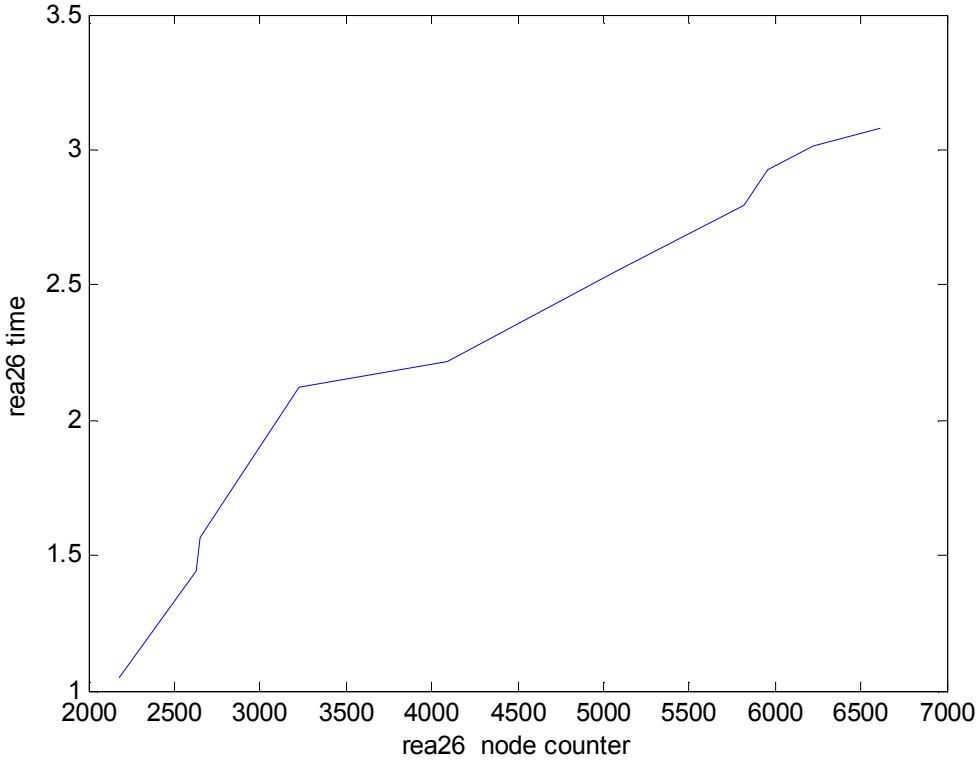


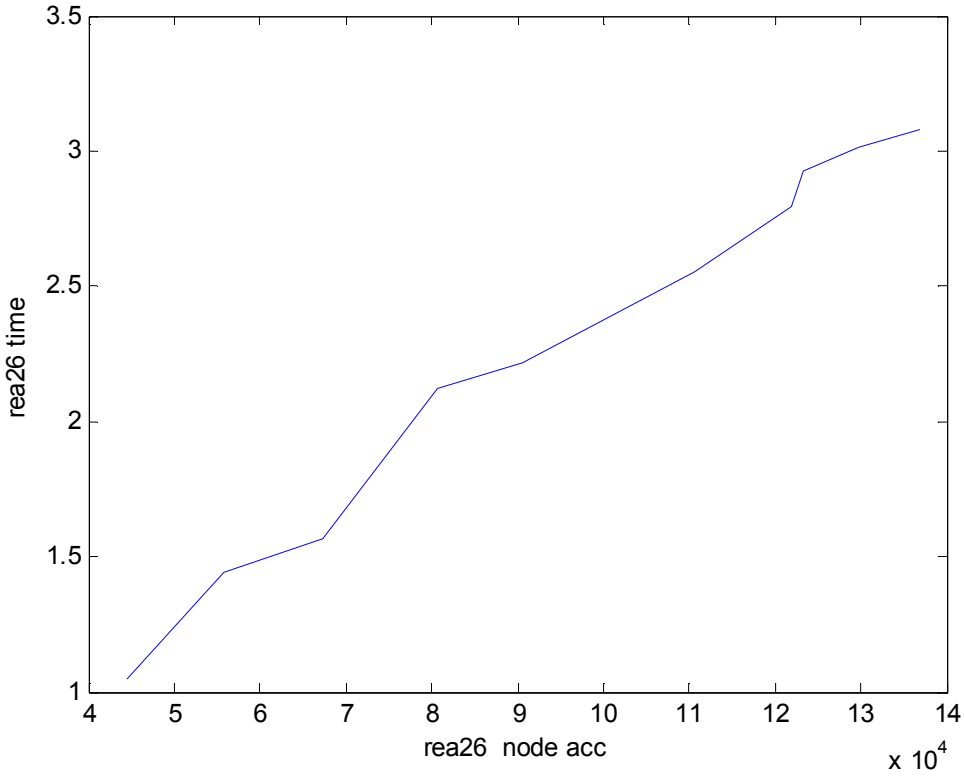

rea22 time
rea22 node counter
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
2000
2500
3000
3500
4000
4500
5000
5500
6000
6500
7000


rea22 time
rea22 node acc
x 10^5
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.5
0.6
0.7
0.8
0.9
1
1.1
1.2
1.3
1.4
1.5

-*Διανομή Real εικοσιέξι διαστάσεων*







Τα διαγράμματα έρχονται να επιβεβαιώσουν ότι ,εκτός από ελάχιστες εξαιρέσεις, ο αλγόριθμος συμπεριφέρεται παρόμοια ανεξαρτήτως διάστασης και ότι επίσης για κάθε διάσταση το ποσοστό των διαπεράσεων των κόμβων, για κάθε 10.000 τετράγωνα που εισάγονται στο δέντρο ,είναι παρόμοιο με μια μεταβολή περίπου 4%.

6. ΣΥΝΟΨΗ

Η εργασία αυτή αποκάλυψε ελλείψεις στον σχεδιασμό των προγενέστερων δομών R-trees που είχαν προταθεί και κυρίως των R*-trees. Σε κανένα από τα δέντρα της οικογενείας των R-trees δεν λαμβάνεται υπόψη η ανισορροπία που δημιουργεί η διαίρεση ενός κόμβου ως κριτήριο βελτιστοποίησης ή το γεγονός ότι η βελτιστοποίηση βασισμένη στο όγκο γίνεται αναποτελεσματική όταν τα επικαλύπτοντα τετράγωνα είναι σε υποχώρους χαμηλών διαστάσεων. Και επίσης , μια σύνθετη βελτιστοποίηση, αυτή της επανεισαγωγής δεν μπορεί να χρησιμοποιηθεί σε μια DBMS.

Για να καταπολεμηθούν αυτές οι ελλείψεις , προτάθηκε λοιπόν ένας αναθεωρημένος R*-tree αλγόριθμος (RR*-tree) .Κατά την εισαγωγή πλέον, περιοριζόμαστε σε ένα και μοναδικό μονοπάτι και εφαρμόζονται και πρότυπα πρωτόκολλα. Η εκτέλεση της αναζήτησης βελτιώθηκε μέσα από καινούριες στρατηγικές που ακολουθήθηκαν. Επίσης, με τον αναθεωρημένο αλγόριθμο μπορέσαμε να επεξεργαστούμε και πολυδιάστατα δεδομένα, αλλάζοντας την στρατηγική από βασισμένη στον όγκο σε στρατηγική βάσει περιμέτρου εκεί που η πρώτη δεν μπορούσε να επιτευχθεί. Τα δεδομένα που χρησιμοποιήθηκαν για την διεξαγωγή των πειραμάτων είναι πρωτοφανή, καθώς είναι συνδυασμός από έναν μεγάλο όγκο πραγματικών και τεχνητών διανομών. Τα πειράματα έδειξαν πως ο αλγόριθμος RR*-tree σε σύγκριση με τις υπόλοιπες δομές ευρετηριοποίησης είναι ο πιο κατάλληλος για να εφαρμοστεί σε μια DBMS καθώς παρέχει εξαιρετική επίδοση και δεν συγκρούονται με τα πρωτόκολλα συγχρονισμού που ισχύουν σε μια βάση δεδομένων.

Συμπερασματικά να τονίσουμε ότι ύστερα από τα παραπάνω πειράματα που κάναμε, καταλήξαμε στο ότι ο αναθεωρημένος αλγόριθμος R*-tree δεν επηρεάζεται σχεδόν καθόλου από τον αριθμό των διαστάσεων των τετραγώνων που εισάγουμε κάθε φορά και δεν υπάρχουν επίσης μεγάλες μεταβολές στον χρόνο που τρέχει ο αλγόριθμος για κάθε διαφορετική διανομή δεδομένων. Παρατηρώντας και τα διαγράμματα είδαμε πως και εκεί υπάρχει μια σταθερότητα στον τρόπο που αυξάνονται οι διαιρέσεις των κόμβων και οι διαπεράσεις κατά την εισαγωγή ενός νέου τετραγώνου, καθώς και στον χρόνο που τρέχει ο αλγόριθμος για κάθε 10.000 εισαγωγές τετραγώνων.

Σ’αυτό το σημείο θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Παναγιώτη Μποζάνη και τον συμφοιτητή μου Αναστάσιο Μ. Χιδερίδη ,που ήταν ο κάτοχος του δεύτερου μηχανήματος που έτρεξα τα πειράματά μου.

ΠΑΡΑΡΤΗΜΑ Α

Περιγραφή του κώδικα.

Ο κώδικας υλοποιήθηκε σε γλώσσα C και η δομή του είναι η εξής:

Αρχικά διαβάζουμε από τη σελίδα http://www.mathematic.uni-marburg.de/~seegeer/rrstar/ τα τετράγωνα και τα δημιουργούμε με την βοήθεια του αρχείου demo.c που είναι το εξής:

demo.c

```
1.typedef typinterval  typrect[2];


2.int main() {
 3.int datafile;
 4. typrect rectangle;
  5.int ferr, length, numbRects, nbytes, i, j;
  6.struct stat status;

 7.datafile= open("abs02",O_RDONLY);

  8.ferr= fstat(datafile,&status);
  9.length= status.st_size;
  10.numbRects= length / sizeof(typrect);

  11.printf("Number of rectangles: %10d\n",numbRects);
  12.printf(" Size of a rectangle: %10d\n",sizeof(typrect));
  13.printf("\n");
```

```
14.i= 0;
15.do {
16.nbytes= read(datafile,rectangle,sizeof(typrect));
 17.if (nbytes != sizeof(typrect)) {
 18. printf("Error during read!\n");
 19   return 1;
 20. }

 21. for (j= 0; j < 2; j++) {
 22.  printf("% .15e % .15e\n",rectangle[j].l,rectangle[j].h);
 23. }
 24. printf("\n");

 25. i++;
26.} while (i < numbRects);

 27.close(datafile);
28.  return 0;
}
```

Και το output του παραπάνω κώδικα θα είναι αρχεία που θα έχουν την μορφή:

(βλέπουμε 3 διδιάστατα τετράγωνα,σε κάθε γραμμή έχουμε το ελάχιστο και το μέγιστο που έχει το κάθε τερτάγωνο σε κάθε διάσταση)

```
Number of rectangles:    1000000
 Size of a rectangle:        32


 7.912761233509085e-05  8.278824376602774e-04
 1.435067319239531e-04  9.470766944464349e-04
```

1.113108637611602e-03 1.967621643689478e-03

-7.695952312052518e-05 7.557696818529785e-04

1.934997468065997e-03 2.901217393498510e-03

3.392188558990890e-04 1.022420132030345e-03

Ο κώδικας αλλάζει στις γραμμές 1,7,21 ανάλογα με τις διαστάσεις και το αρχείο που θέλουμε να δουλέψουμε.

Το αρχείο rrstartree διαβάζει το αρχείο με τα αποθηκευμένα τετράγωνα της κάθε διανομής και εκτελεί την δομή του αλγορίθμου RR*-tree που περιγράφηκε στα παραπάνω κεφάλαια.

rrstartree.c (για 2 διαστάσεις και κάνουμε τις ανάλογες αλλαγές για την υλοποίηση των άλλων διαστάσεων)

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <float.h>
#include <math.h>
#include <sys/times.h>
#include <stdint.h>

#include <time.h>
#include "assert.h"
#include "Index.h"
#include "card.h"
#include "Split.h"
```

```
#define BIG_NUM (FLT_MAX/4.0)



#define Undefined(x) ((x)->boundary[0] > (x)->boundary[NUMDIMS])
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define NUMBER_OF_RECTANGLES 100000

#define SYNTETAGMENES 4
//init
struct Rect rects[] = {
        {0, 0, 0, 0}, // xmin, ymin, xmax, ymax (for 2 dimensional RTree)
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        ……
        };

int nrects = sizeof(rects) / sizeof(rects[0]);




void initial() {

FILE *f; char s[22]; char c; int i=0;
float integer[NUMBER_OF_RECTANGLES*SYNTETAGMENES ];
int k=0; int j=0;
int inte[NUMBER_OF_RECTANGLES*SYNTETAGMENES ];
```

```
for(i=0; i<22; i++) { s[i]='a';}

f=fopen("abs02.txt","r+");
i=0;
   while (!feof(f)) {
         c= fgetc(f);
         if((c!=' ')&&(c!='\n')){
                  for(i=0; i<22; i++){
                  s[i]=c;
                  c=fgetc(f);
                  }

                  integer[j] = atoi (s);
                  j++;
                   }

             }
fclose(f);




         struct Rect arects[]= {{integer[0],integer[1],integer[2],integer[3]},
                                {integer[4],integer[5],integer[6],integer[7]},
                                {integer[8],integer[9],integer[10],integer[11]},
                                ……
         };
for(i=0; i<nrects; i++) { rects[i]=arects[i];}
```

```
}



struct Rect search_rect = {
        {1, 2, 1, 2}, // search will find above rects that this one overlaps
};


//-----------------------------------------------------------------------------------------------


RectReal rect[NUMDIMS][MAXCARD+1]; //arxikopoiisi
int axis;

int mini=0;


int p=0;


int splits_counter=1;

/*-----------------------------------------------------------------------
| Initialize a rectangle to have all 0 coordinates.
-----------------------------------------------------------------------*/
void RTreeInitRect(struct Rect *R)
{
         struct Rect *r = R;
        int i;
        for (i=0; i<NUMSIDES; i++)
```

```
                r->boundary[i] = (RectReal)0;

}


/*-----------------------------------------------------------------------
| Return a rect whose first low side is higher than its opposite side -
| interpreted as an undefined rect.
-----------------------------------------------------------------------*/
struct Rect RTreeNullRect()
{
        struct Rect r;
        int i;

        r.boundary[0] = (RectReal)1;
        r.boundary[NUMDIMS] = (RectReal)-1;
        for (i=1; i<NUMSIDES; i=i+2)
                r.boundary[i] = r.boundary[i+1] = (RectReal)0;
        return r;
}



/*-----------------------------------------------------------------------
| Fills in random coordinates in a rectangle.
| The low side is guaranteed to be less than the high side.
-----------------------------------------------------------------------*/
void RTreeRandomRect(struct Rect *R)
{
        struct Rect *r = R;
        int i;
         RectReal width;
```

```
        for (i = 0; i < NUMSIDES; i=i+2)
        {
                /* width from 1 to 1000 / 4, more small ones
                */
                width = drand48() * (1000 / 4) + 1;

                /* sprinkle a given size evenly but so they stay in [0,100]
                */
                r->boundary[i] = drand48() * (1000-width); /* low side */
                r->boundary[i + 1] = r->boundary[i] + width; // high side
        }
}


/*-----------------------------------------------------------------------------
| Fill in the boundaries for a random search rectangle.
| Pass in a pointer to a rect that contains all the data,
| and a pointer to the rect to be filled in.
| Generated rect is centered randomly anywhere in the data area,
| and has size from 0 to the size of the data area in each dimension,
| i.e. search rect can stick out beyond data area.
-----------------------------------------------------------------------------*/
void RTreeSearchRect(struct Rect *Search, struct Rect *Data)
{
         struct Rect *search = Search, *data = Data;
         int i, j;
         RectReal size, center;

        assert(search);
        assert(data);

        for (i=0; i<NUMSIDES; i=i+2)
```

```
        {
                j = i + 1;  // index for high side boundary
                if (data->boundary[i] > -BIG_NUM &&
                   data->boundary[j] < BIG_NUM)
                {
                        size = (drand48() * (data->boundary[j] -
                                        data->boundary[i] + 1)) / 2;
                        center = data->boundary[i] + drand48() *
                                (data->boundary[j] - data->boundary[i] + 1);
                        search->boundary[i] = center - size/2;
                        search->boundary[j] = center + size/2;
                }
                else  // some open boundary, search entire dimension
                {
                        search->boundary[i] = -BIG_NUM;
                        search->boundary[j] = BIG_NUM;
                }
        }
}
```

```
/*-----------------------------------------------------------------------
| Print out the data for a rectangle.
-----------------------------------------------------------------------*/
void RTreePrintRect(struct Rect *R, int depth)
{
         struct Rect *r = R;
        int i;
```

```
        assert(r);

        RTreeTabIn(depth);
        printf("rect:\n");
        for (i = 0; i < NUMSIDES; i=i+2) {
                RTreeTabIn(depth+1);
                printf("%f\t%f\n", r->boundary[i], r->boundary[i + 1]);
        }
}




/*-----------------------------------------------------------------------
| Calculate the n-dimensional volume of a rectangle
-----------------------------------------------------------------------*/
RectReal RTreeRectVolume(struct Rect *R)
{
         struct Rect *r = R;
         int i;
         RectReal volume = (RectReal)1;

        assert(r);
        if (Undefined(r))
                return (RectReal)0;

        for(i=0; i<NUMSIDES; i=i+2)
                volume *= r->boundary[i+1] - r->boundary[i];
        assert(volume >= 0.0);
        return volume;
}
```

```
/*-----------------------------------------------------------------------
| Calculate the n-dimensional perimeter of a rectangle-------------------------
------------------------------------------------------------------------------*/
RectReal RTreeRectPerimeter(struct Rect *R)
{
          struct Rect *r = R;
          int i;
          RectReal perimeter = (RectReal)0;

         assert(r);
         if (Undefined(r))
                  return (RectReal)0;

         for(i=0; i<NUMSIDES; i=i+2)
                  perimeter += 2*(r->boundary[i+1] - r->boundary[i]); //r-
>boundary[i+1] - r->boundary[i];  xmax-xmin=length
         if(perimeter < 0.0){ return FALSE;}
         return perimeter;
}
```

```
/*-----------------------------------------------------------------------
| Combine two rectangles, make one that includes both.
-----------------------------------------------------------------------*/
struct Rect  RTreeCombineRect(struct Rect *R, struct Rect *Rr)
{
        struct Rect *r = R, *rr = Rr;
         int i, j;
        struct Rect new_rect;
        assert(r && rr);

        if (Undefined(r))
                return *rr;

        if (Undefined(rr))
                return *r;

        for (i = 0; i < NUMSIDES; i=i+2)
        {
                new_rect.boundary[i] = MIN(r->boundary[i], rr->boundary[i]);
                j = i +1; //NUMDIMS;
                new_rect.boundary[j] = MAX(r->boundary[j], rr->boundary[j]);
        }
        return new_rect;
}


/*-----------------------------------------------------------------------
```

```
| Decide whether two rectangles overlap.
--------------------------------------------------------------------------*/
int RTreeOverlap(struct Rect *R, struct Rect *S)
{
        struct Rect *r = R, *s = S;
        int i, j;
       assert(r && s);
       struct Rect new_rect;


       for (i = 0; i < NUMSIDES; i=i+2)
       {


                       new_rect.boundary[i] = MAX(r->boundary[i], s->boundary[i]);
                       j = i + 1;
                       new_rect.boundary[j] = MIN(r->boundary[j], s->boundary[j]);

       }

       RectReal c=RTreeRectPerimeter(&new_rect);
       if(c==0)  return FALSE;
       else   return TRUE;


}


/*--------------------------------------------------------------------------
| Decide whether rectangle r is contained in rectangle s.
--------------------------------------------------------------------------*/
int RTreeContained(struct Rect *R, struct Rect *S)
```

```
{
        struct Rect *r = R, *s = S;
        int i, j;
       assert(r && s);
       struct Rect new_rect;


       for (i = 0; i < NUMSIDES; i=i+2)
       {


                        new_rect.boundary[i] = MAX(r->boundary[i], s->boundary[i]);
                        j = i + 1;
                        new_rect.boundary[j] = MIN(r->boundary[j], s->boundary[j]);

       }

       RectReal a=RTreeRectPerimeter(r);
       RectReal b=RTreeRectPerimeter(s);
       RectReal c=RTreeRectPerimeter(&new_rect);

       if(c==b)   return TRUE; //if r contained on s => intersect = r
       else return FALSE;


}
```

```
//=============================================================
//---------------------------------INTERSECT-----------------
//=============================================================
=

struct Rect RTreeIntersectRect(struct Rect *R, struct Rect *Rr)
{
        struct Rect *r = R, *rr = Rr;
        int i, j;
        struct Rect new_rect;
        assert(r && rr);

        if (Undefined(r))
                return *rr;

        if (Undefined(rr))
                return *r;

        for (i = 0; i < NUMSIDES; i=i+2)
        {


                        new_rect.boundary[i] = MAX(r->boundary[i], rr->boundary[i]);
                        j = i + 1;
                        new_rect.boundary[j] = MIN(r->boundary[j], rr->boundary[j]);
```

```
        }
        return new_rect;
}




int NODECARD = MAXCARD;
int LEAFCARD = MAXCARD;




/*-----------------------------------------------------------------------------
| Load branch buffer with branches from full node plus the extra branch.
-----------------------------------------------------------------------------*/
void RTreeGetBranches(struct Node *N, struct Branch *B)
{
         struct Node *n = N;
         struct Branch *b = B;
         int i;

        assert(n);
        assert(b);

        /* load the branch buffer */
        for (i=0; i<MAXKIDS(n); i++)
        {
                assert(n->branch[i].child);  /* every entry should be full */
```

```
            BranchBuf[i] = n->branch[i];
        }
        BranchBuf[MAXKIDS(n)] = *b;             //the extra branch
        BranchCount = MAXKIDS(n) + 1;

        /* calculate rect containing all in the set */
        CoverSplit = BranchBuf[0].rect;
        for (i=1; i<MAXKIDS(n)+1; i++)
        {
            CoverSplit = RTreeCombineRect(&CoverSplit, &BranchBuf[i].rect);
//put them all in one rectangle
        }

        RTreeInitNode(n);      //all the rects which were in the old node + the extra rect
are now in buffer(coversplit) and the old node initialized

}

/*-----------------------------------------------------------------------------
| Initialize a PartitionVars structure.
-----------------------------------------------------------------------------*/
void RTreeInitPVars(struct PartitionVars *P, int maxrects, int minfill)
{
        struct PartitionVars *p = P;
        int i;
        assert(p);
```

```
        p->count[0] = p->count[1] = 0;
        p->total = maxrects;
        p->minfill = minfill;                   //p->minnfill=m ,
LEAFCARD,NODECARD=M
        for (i=0; i<maxrects; i++)
        {
                p->taken[i] = FALSE;
                p->partition[i] = -1;
        }
}
```

```
/*-----------------------------------------------------------------------------
| Put a branch in one of the groups.
-----------------------------------------------------------------------------*/
void RTreeClassify(int i, int group, struct PartitionVars *p)
{
        assert(p);
        assert(!p->taken[i]);

        p->partition[i] = group;
        p->taken[i] = TRUE;

        if (p->count[group] == 0)
                p->cover[group] = BranchBuf[i].rect;
        else
                p->cover[group] = RTreeCombineRect(&BranchBuf[i].rect,&p-
>cover[group]);
```

```
        p->count[group]++;
}




void swap (RectReal *a, RectReal *b)
{
        int tmp;
        tmp=*a; *a=*b; *b=tmp;
}


//sort for line 9 (pg 802 N.Beckmann,B.Seeger)
void sorting(RectReal t[], int len,RectReal r[])
{
        int i,j;
        for(i=0; i<len; i++) {
                for(j=i; j<len; j++) {
                        if(t[i]>t[j]) { swap(&t[i],&t[j]);  swap(&r[i],&r[j]);}
                }
        }
}
```

```
/*-----------------------------------------------------------------------------
| Calculate the perim(MBB(F i,d) )  +   perim(MBB(S i,d)) (in the i-th dimension
| i=m,...,M+1-m, with respect to the ordering of the d-th dimension.
|Fi,d represents the first i entries of the node , whereas the remaining  M+1-i
| constitute Si,d.
-----------------------------------------------------------------------------*/

RectReal find_perim(int i,int dimension) {
                int j;
                struct Rect temp_rect1,temp_rect2;
                RectReal perim1;
                RectReal perim2;
                struct Rect r;
                int k;



                temp_rect1=BranchBuf[0].rect;



                for (j=0; j<i; j++)                                             //
#define MinNodeFill (NODECARD / 2) CARD.h
                        {
```

```
                    k=rect[dimension][j];

                    temp_rect1 =
RTreeCombineRect(&temp_rect1,&BranchBuf[k].rect);      //MBB(F i,d)

                }

                perim1=RTreeRectPerimeter(&temp_rect1);
        //perim(MBB(F i,d) )









                int b=j;
                k=rect[dimension][b];
                temp_rect2=BranchBuf[k].rect;

                for (j=b; j<=NODECARD; j++)
                    {

                    k=rect[dimension][j];



                    temp_rect2 =
RTreeCombineRect(&temp_rect2,&BranchBuf[k].rect);    //MBB(S i,d)
                    }

                perim2=RTreeRectPerimeter(&temp_rect2);          //perim(MBB(S
i,d) )
```

```
                return (perim1+perim2);
        //perim(MBB(F i,d) )  +   perim(MBB(S i,d) )

}

/*-------------------------------------------------------------------------
| Calculate the overlap_volume = volume(MBB(F i,d) )  intersect   volume(MBB(S i,d))
-------------------------------------------------------------------------*/
RectReal find_ovlp_volume(int i) {
                 int j,k,c;

                struct Rect temp_rect3,temp_rect4,rect_intersect;
                RectReal volume1;

                k=rect[axis][0];
                temp_rect3=BranchBuf[k].rect;

                for (j=0; j<i; j++)                //   #define MinNodeFill (NODECARD / 2) CARD.h
                        {

                        k=rect[axis][j];

                        temp_rect3 = RTreeCombineRect(&temp_rect3,&BranchBuf[k].rect);       //MBB(F i,d)

                        }


                k=rect[axis][j];
```

```
                temp_rect4=BranchBuf[k].rect;
                c=j;
                for (j=c; j<=NODECARD; j++)
                        {
                        k=rect[axis][j];

                        temp_rect4=
RTreeCombineRect(&temp_rect4,&BranchBuf[k].rect);    //MBB(S i,d)

                }

                if(!RTreeOverlap(&temp_rect3,&temp_rect4))
        //OVERLAP FREE
                {
                        return 0;
                }
                else
                {
        //otherwise calculate their intersect
                        rect_intersect =
RTreeIntersectRect(&temp_rect3,&temp_rect4);              // (MBB(Fi,d)intersect
MBB(Si,d))
                        volume1=RTreeRectVolume(&rect_intersect);                 ////
volume(MBB(Fi,d)intersect MBB(Si,d))
                        return volume1;
                }
}
```

```
/*----------------------------------------------------------------------------------------------
| Calculate the overlap_perimeter = perimeter(MBB(F i,d) )  intersect
perimeter(MBB(S i,d))
----------------------------------------------------------------------------------------------*/
RectReal find_ovlp_perimeter(int i) {

                struct Rect temp_rect5,temp_rect6,rect_intersect;
                RectReal perimeter1;
                  int j,k,c;




                k=rect[axis][0];
                temp_rect5=BranchBuf[k].rect;
                for (j=0; j<i; j++)
                        {
                        k=rect[axis][j];

                        temp_rect5 =
RTreeCombineRect(&temp_rect5,&BranchBuf[k].rect);       //MBB(F i,d)
                }



                k=rect[axis][j];
                temp_rect6=BranchBuf[k].rect;
```

```
		c=j;

		for (j=c; j<=NODECARD; j++)
			{
			k=rect[axis][j];

			temp_rect6 =
RTreeCombineRect(&temp_rect6,&BranchBuf[k].rect);    //MBB(S i,d)
		}


		if(RTreeOverlap(&temp_rect5,&temp_rect6))         //OVERLAP
FREE
		{
			return 0;
		}
		else
		{                                                                    //
otherwise calculate their intersect
			rect_intersect =
RTreeIntersectRect(&temp_rect5,&temp_rect6);      // (MBB(Fi,d)intersect
MBB(Si,d))
			perimeter1=RTreeRectPerimeter(&rect_intersect);
		//// perimeter(MBB(Fi,d)intersect MBB(Si,d))
			return perimeter1;
		}
}
```

```
/*==========================================================
==========
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\\\\\\\\\\\\\\weighting function \\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
wf(i) = wf(i , asym , s , M , m ) = ys * { exp[-((xi-Î¼) / Ïƒ ) 2 ] â€“y1}
==========================================================
==========*/


RectReal weighting_function (int i , int asym , int s ,int M ,int m)
{       RectReal mi,sigma,xi;

         mi=((1-2*m)/(M+1))*asym;
         sigma=s*(1+abs(mi));   //absolute value

         xi=(2*i)/((M+1)-1);


       return xi;
}


/*---------------------------------------------------
| Calculate the minimum length for each dimension
|of the overfilled node which is in buffer(Coversplit)
```

```
|length_min(RectN)=min{length1(RectN),....length_dim(RectN) }
|(pg 804 N.Beckmann,B.Seeger)
-----------------------------------------------------*/
RectReal find_min_length (struct Rect *R)
{
        struct Rect *r = R;
        int i;
        RectReal length[NUMDIMS];
        RectReal min_mikos;

        /*assert(r && rr);

        if (Undefined(r))
                return *rr;

        if (Undefined(rr))
                return *r; */

        int l=0;
        for (i = 0; i < NUMSIDES; i=i+2)
        {
                length[l] = r->boundary[i+1]-r->boundary[i];
                l++;

        }

        min_mikos=length[0];

        for(i=1; i<NUMDIMS; i++) {
                if(min_mikos>length[i]){
                        min_mikos=length[i];
                }
```

```
        }

        return min_mikos;
}



/*-------------------------------------------------------
| Calculate the sum of lengthsfor each dimension
|of the overfilled node which is in buffer(Coversplit)
|sum_length(RectN)=length1(RectN),....length_dim(RectN) }
-------------------------------------------------------*/
RectReal find_sum_length(struct Rect *R)
{
         struct Rect *r = R;
        int i;
        RectReal sum_length=0;



        assert(r);




        for (i = 0; i < NUMSIDES; i=i+2)
        {
                sum_length =sum_length + ( r->boundary[i+1]-r->boundary[i]);

        }

        return sum_length;
}
```

```
/*-----------------------------------------------------------------
                perim(SCi) â€“ perimmax          if ovlpf(SCi) = 0
wg(i) = {
                ovlpf(SCi)                     otherwise
-----------------------------------------------------------------*/
RectReal wg(int i,RectReal per_max)
{
        RectReal result_wgi;

        if(!find_ovlp_perimeter(i))            //overlap free because returns 0
                {
                result_wgi=find_perim(i,axis)-per_max;
                }

        else
                {
                result_wgi=find_ovlp_perimeter(i);
                }

        return result_wgi;
}
```

```
/*----------------------------------------------------------------

Our composed weighting function given by

                wg(i) * wf(i)           if ovlpf(SCi) = 0
w(i) = {
                wg(i) / wf(i)           otherwise

----------------------------------------------------------------*/

RectReal wf(int ki,RectReal max_perim,RectReal asym,int s,int M,int m)
//arguments-> i,asym,s,NODECARD,p->minfill
{
        RectReal result;


        if(!find_ovlp_perimeter(ki))                    //overlap free because
returns 0
        {
                result=wg(ki,max_perim)*weighting_function(ki,asym,s,M,m);
        }


        else
        {
                result=wg(ki,max_perim)/weighting_function(ki,asym,s,M,m);
        }
        return result;
}
```

```
/*-----------------------------------------------------
| Calculate the center of rect R in i-th dimesion
-----------------------------------------------------*/
RectReal find_center_a(struct Rect *R,int i) {
        struct Rect *r = R;
        RectReal center=( r->boundary[i + 1] - r->boundary[i])/2;
        return center;
}

/*-----------------------------------------------------
| Calculate the length of rect R in i-th dimesion
-----------------------------------------------------*/
RectReal find_length_a(struct Rect *R,int i) {
        struct Rect *r = R;
        RectReal length_a= r->boundary[i + 1] - r->boundary[i];
        return length_a;
}



/*-----------------------------------------------------
| Conider the split at the i-th element , where the first
|i elements are assign to the first node and the remaining
|M+1-i entries assigned to the second node.

|Our revised algorithm distinguishes between splitting
```

```
|internal nodes and leaves,concerning the choics of the
|split axis
-----------------------------------------------------*/

void RTreePickSeeds(struct PartitionVars *p,struct Node *n,int level)
{




     int i, dim, high;
     struct Rect *r, *rlow, *rhigh;
    float w, separation, bestSep;
    RectReal width[NUMDIMS];
    int leastUpper[NUMDIMS], greatestLower[NUMDIMS];
    int seed0, seed1;
    assert(p);



    int sum=0;

    RectReal asym;
    RectReal perim_max;




    RectReal min_length=find_min_length(&CoverSplit);
    RectReal sum_length=find_sum_length(&CoverSplit);
```

```
int low;

//for spliting a leaf
RectReal sort_array[NUMDIMS][NODECARD+1];

RectReal length;
RectReal sort_temp[NODECARD+1];
RectReal rects_temp[NODECARD+1];

RectReal sum_ar[NUMDIMS];

int z;
int j=0;

RectReal ovlp_vol[NODECARD+1-2*(p->minfill)];

RectReal ovlp_perim[NODECARD+1-2*(p->minfill)];

RectReal w1[NODECARD+1-2*(p->minfill)+1];

RectReal ua;
```

```
int ii;

if(level==0) {

        //the split axis is still determined as described in step

        for (dim=0; dim<NUMDIMS; dim++)          //for each dimension sort
rects  according the min boundary
        {
                low=dim*NUMDIMS;                 // find the min for each
dimension

                for (i=0; i<=NODECARD; i++)
                {

                r = &BranchBuf[i].rect;
                sort_array[dim][i]=r->boundary[low];

                rect[dim][i]=i;

                }
        }

       //sorting
        length=i-1;

        for (dim=0; dim<NUMDIMS; dim++)
        {
```

```
                for (i=0; i<=NODECARD; i++)
                {
                        sort_temp[i]=sort_array[dim][i];
                        rects_temp[i]=rect[dim][i];
                }


                sorting(sort_temp,NODECARD+1,rects_temp);


                 for (i=0; i<=NODECARD; i++)
                {
                sort_array[dim][i]=sort_temp[i];
                rect[dim][i]=rects_temp[i];
                }


}


for (i=0; i<NUMDIMS; i++)
        {
        sum_ar[i]=0;
        }




for (dim=0; dim<NUMDIMS; dim++)
{
```

```
                    for (i=p->minfill; i<=NODECARD+1-(p->minfill); i++)
//   #define MinNodeFill (NODECARD / 2) CARD.h
                    {

                    sum_ar[dim]=sum_ar[dim]+find_perim(i,dim);      //
Sum( i=m , M+1-m) [perim(SCi,d)]
                    }

          }




          axis=sum_ar[0];
          int counter=0;
                                        //axis = min 1<=d<=dim { S
(i=m,i=M+1-m) [perim(SCi,d)] }
          for(z=1; z<NUMDIMS; z++)
          {
               if(axis>sum_ar[z]){
                    axis=sum_ar[z];
                    counter=z;
               }
          }

          axis=counter;
```

```
//------calculate OVLPvolume-------
                for (i=p->minfill; i<=NODECARD+1-(p->minfill); i++)
                {
                        ovlp_vol[j]= find_ovlp_volume(i);

                        if (ovlp_vol[j]==0) {  break ; }          // OVERLAP FREE
                        j++;
                }

//-----------------------------------




//------upologismos tou OVLPperimeter----------
                j=0;

                for (i=p->minfill; i<=NODECARD+1-(p->minfill); i++)
                {
                        ovlp_perim[j]= find_ovlp_perimeter(i);

                        if (ovlp_perim[j]==0) { break; }  //  OVERLAP FREE
                        j++;
                }
//-------------------------------------------


//init w1


        for(ii=0; ii<(NODECARD+1-2*(p->minfill)+1); ii++ ) {w1[ii]=1000.0; }
```

```
//=================================================================
====
//------------------design of the weigthing function--------------
//=================================================================
====



//------------------perim max-----------
        perim_max=2*sum_length-min_length;
//----------------------------------------



//asym is the axis for the perim(SCi,a)



                asym= 2*(find_center_a(&CoverSplit,axis) - find_center_a(n-
>init,axis) )/ (find_length_a(&CoverSplit,axis));



// THE SECOND STEP  of our new algorithm is then:

                j=0;
```

```
        for (i=p->minfill; i<=NODECARD+1-(p->minfill); i++)
        {
                w1[j]= wf(i,perim_max,axis,1,NODECARD,p->minfill);
                j++;

        }



    ua=w1[0];
                                                        //axis = min 1<=d<=dim
S(i=m,i=M+1-m) [perim(SCi,d)]

        for(z=1; z<NODECARD+1-2*(p->minfill); z++) {

                if(ua>w1[z]){
                        ua=w1[z];
                        mini=z;
                }
        }

    }
```

```
RectReal ua_level1[NUMDIMS];


        if(level==1)                //for internal node the first step is left out and all
dimensions are considered in the computation of the second step
        {                           // we choose the axis that minimizes the goal function
//init ua_level1
                for(ii=0; ii<NUMDIMS; ii++ ) {ua_level1[ii]=1000.0; }
                for (axis=0; axis<NUMDIMS; axis++)
                {




//----------upologismos tou OVLPvolume----------


                for (i=p->minfill; i<NODECARD+1-(p->minfill); i++)
                        {
                                ovlp_vol[j]= find_ovlp_volume(i);
                                if (ovlp_vol[j]==0) {break ; }  // OVERLAP FREE
                                j++;
                        }
```

```
//-------upologismos tou OVLPperimeter-------
//=============================================================
=============================================================
====================

                        j=0;

                        for (i=p->minfill; i<NODECARD+1-(p->minfill); i++)
//   #define MinNodeFill (NODECARD / 2) CARD.h
                        {
                                ovlp_perim[j]= find_ovlp_perimeter(i);
                                if (ovlp_perim[j]==0) {break; }  // exoume OVERLAP
FREE
                                j++;
                        }




//=======================================================
//------------design of the weigthing function---------
//=======================================================

//------------------perim max-----------

                        perim_max=2*sum_length-min_length;
//----------------------------------------
```

```
                    asym= 2*(find_center_a(&CoverSplit,axis) - find_center_a(n->init,axis) )/ (find_length_a(&CoverSplit,axis));




                for(ii=0; ii<(NODECARD+1-2*(p->minfill)+1); ii++ ) {w1[ii]=1000.0; }



                // THE SECOND STEP  of our new algorithm is then:

                j=0;



                for (i=p->minfill; i<=NODECARD+1-(p->minfill); i++)
                {
                w1[j]= wf(i,perim_max,axis,1,NODECARD,p->minfill);
                j++;
                }



                //kai vriskoume to min ua

                 ua_level1[axis]=w1[0];
                                                                //f = min 1<=d<=dim { S(i=m,i=M+1-m) perim(SCi,d) }
                for(z=1; z<NODECARD+1-2*(p->minfill); z++) {
                        if(ua_level1[axis]>w1[z]){
```

```
                                ua_level1[axis]=w1[z];
                                //mini=z;
                        }
                }
//-----------------------------------------------------------------------------------------------------
-------------------------------------------


                }


        int minaxis=ua_level1[0];
        mini=0;
        for(i=1; i<NUMDIMS; i++) {
           if(minaxis>ua_level1[i]){
                        minaxis=ua_level1[i];
                           mini=i;
                                        }
                   }


        }


printf("pickseeds min i = %d\n",mini);

}
```

```
/*-----------------------------------------------
| Also update the covers for both groups.
-----------------------------------------------*/
void RTreePigeonhole(struct PartitionVars *P)
{
        struct PartitionVars *p = P;
        int i;


      for (i=0; i<NODECARD+1; i++)
      {
            if (!p->taken[i])
            {
                  if(i<=mini){

                        RTreeClassify(i, 0, p);

                  }
                  else {

                        RTreeClassify(i, 1, p);
                  }
            }
      }
      assert(p->count[0] + p->count[1] == NODECARD + 1);
```

```
}



/*-----------------------------------------------------------------------
| Method 0 for finding a partition:
| First find two seeds, one for each group, well separated.
| Then put other rects in whichever group will be smallest after addition.
-----------------------------------------------------------------------*/
void RTreeMethodZero(struct PartitionVars *p, int minfill,struct Node *n,int level)
{

        RTreeInitPVars(p, BranchCount, minfill);

        RTreePickSeeds(p,n,level);


        RTreePigeonhole(p);


}




/*-----------------------------------------------------------------------
| Copy branches from the buffer into two nodes according to the partition.
-----------------------------------------------------------------------*/
void RTreeLoadNodes(struct Node *N, struct Node *Q,struct PartitionVars *P)
{
```

```
 struct Node *n = N, *q = Q;
struct PartitionVars *p = P;
int i;
assert(n);
assert(q);
assert(p);

for (i=0; i<NODECARD+1; i++)
{
        if (p->partition[i] == 0)               // p->partition is 0 or 1 and inform
us in which group the rect inserted
        {       RTreeAddBranch(&BranchBuf[i], n, NULL);


                printf("BranchBuf[i].rect =
{ %f , %f , %f , %f }\n",BranchBuf[i].rect.boundary[0],BranchBuf[i].rect.boundary[1]
,BranchBuf[i].rect.boundary[2],BranchBuf[i].rect.boundary[3]);



}
        else if (p->partition[i] == 1)
        {       RTreeAddBranch(&BranchBuf[i], q, NULL);


                printf("BranchBuf[i].rect =
{ %f , %f , %f , %f }\n",BranchBuf[i].rect.boundary[0],BranchBuf[i].rect.boundary[1]
,BranchBuf[i].rect.boundary[2],BranchBuf[i].rect.boundary[3]);


         }
    else
                assert(FALSE);
    }
}
```

```
/*-----------------------------------------------------------------------------
| Split a node.
| Divides the nodes branches and the extra one between two nodes.
| Old node is one of the new ones, and one really new one is created.
-----------------------------------------------------------------------------*/
void RTreeSplitNode(struct Node *n, struct Branch *b, struct Node **nn)
{
        struct PartitionVars *p;
        int level;


      assert(n);
      assert(b);

      /* load all the branches into a buffer, initialize old node */
      level = n->level;

      RTreeGetBranches(n, b);

      /* find partition */
      p = &Partitions[0];




      /* Note: can't use MINFILL(n) below since n was cleared  -> (init node) by
GetBranches() */
```

```
        RTreeMethodZero(p, level>0 ? MinNodeFill : MinLeafFill,n,level);   //if
level>0 then use MinNodeFill




        /* put branches from buffer in 2 nodes according to chosen partition */
        *nn = RTreeNewNode();
        (*nn)->level = n->level = level;
        RTreeLoadNodes(n, *nn, p);
        assert(n->count + (*nn)->count == NODECARD+1);

}







/*-----------------------------------------------------------------------------
| Print out data for a partition from PartitionVars struct.
-----------------------------------------------------------------------------*/
void RTreePrintPVars(struct PartitionVars *p)
{
        int i;
        assert(p);

        printf("\npartition:\n");
        for (i=0; i<NODECARD+1; i++)
        {
                printf("%3d\t", i);
        }
```

```
        printf("\n");
        for (i=0; i<NODECARD+1; i++)
        {
                if (p->taken[i])
                        printf("  t\t");
                else
                        printf("\t");
        }
        printf("\n");
        for (i=0; i<NODECARD+1; i++)
        {
                printf("%3d\t", p->partition[i]);
        }
        printf("\n");



printf("count[0] = %d\n", p->count[0]);
printf("count[1] = %d\n", p->count[1]);



        printf("cover[0]:\n");
        RTreePrintRect(&p->cover[0], 0);

        printf("cover[1]:\n");
        RTreePrintRect(&p->cover[1], 0);
}
```

```
// Initialize one branch cell in a node.

void RTreeInitBranch(struct Branch *b)
{
        RTreeInitRect(&(b->rect));
        b->child = NULL;

}



// Initialize a Node structure.

void RTreeInitNode(struct Node *N)
{
         struct Node *n = N;
        int i;
        n->count = 0;
        n->level = -1;


        if(n->init==NULL) {
                n->init= (struct Rect*)malloc(sizeof(struct Rect));
                for(i=0; i<NUMDIMS; i++) {
                        n->init->boundary[i]=0;
                        }
        }
        for (i = 0; i < MAXCARD; i++)
```

```
                    RTreeInitBranch(&(n->branch[i]));
}



// Make a new node and initialize to have all branch cells empty.
//
struct Node * RTreeNewNode()
{
          struct Node *n;


          n = (struct Node*)malloc(sizeof(struct Node));
          assert(n);
          RTreeInitNode(n);
          return n;
}


void RTreeFreeNode(struct Node *p)
{
          assert(p);

          free(p);
}



void RTreePrintBranch(struct Branch *b, int depth)
{         printf("print branch\n");
          RTreePrintRect(&(b->rect), depth);
          RTreePrintNode(b->child, depth);
```

```
}


void RTreeTabIn(int depth)
{
        int i;
        for(i=0; i<depth; i++)
                putchar('\t');
}


// Print out the data in a node.
void RTreePrintNode(struct Node *n, int depth)
{
        int i;
        assert(n);

        RTreeTabIn(depth);
        printf("node");
        if (n->level == 0)
                printf(" LEAF");
        else if (n->level > 0)
                printf(" NONLEAF");
        else
                printf(" TYPE=?");

        printf("  level=%d  count=%d  address=%o\n", n->level, n->count, n);

        for (i=0; i<n->count; i++)
        {
                if(n->level == 0) {
                        printf("n->level=0\n");
```

```
                        RTreeTabIn(depth);
                        RTreePrintRect(&n->branch[i].rect, n->level);
                         printf("\t%d: data = %d\n", i, n->branch[i].child);
                }
                else {
                        RTreeTabIn(depth);
                        printf("branch %d\n", i);
                        RTreePrintBranch(&n->branch[i], depth+3);
                }
        }
}
```

```
// Find the smallest rectangle that includes all rectangles in
// branches of a node.
struct Rect RTreeNodeCover(struct Node *N)  //me to combine einai to mikrotero
tetragwno giati enwnei ta mesa tetragwna simfwna me ta max
{                                   //kai min twn mesa tetragvnvn
         struct Node *n = N;
         int i, first_time=1;
        struct Rect r;
        assert(n);

        RTreeInitRect(&r);
        for (i = 0; i < MAXKIDS(n); i++)
                if (n->branch[i].child)
                {
                        if (first_time)
                        {
                                r = n->branch[i].rect;
```

```
                    first_time = 0;
                }
                else
                    r = RTreeCombineRect(&r, &(n->branch[i].rect));  // to r pou dhmiourgeitai to syndyazei kathe fora me to epomeno paidi
            }
        return r;
}
```

```
// Pick a branch.  Pick the one that ----------will need the smallest increase----------
// in area to accomodate the new rectangle.  This will result in the
// least total area for the covering rectangles in the current node.
// In case of a tie, pick the one which was smaller before, to get
// the best resolution when searching.
//
```

```
void sort(RectReal t[], int len)
{printf("sort%f  %d\n",t[0],len);
        int i,j;
        for(i=0; i<=len; i++) {
                for(j=i; j<=len; j++) {
                        if(t[i]>t[j]) { printf("sort%f\n",t[i]);swap(&t[i],&t[j]); }
                }
        }
}



int perimeter1;
int perimeter2;


int CheckComp(int t , int f,RectReal Doverlapperimeter[][p],RectReal
Doverlapvolume[][p],int cand[],struct Rect *r,int p1,int success,int c)
{
        struct Rect tmp_rect;


        struct Rect rect1,rect2;
        RTreeInitRect(&tmp_rect);
        RTreeInitRect(&rect1);
        RTreeInitRect(&rect2);


        struct Node *n;
        static int i =0;
        cand[i]=t;                      //line 21
        i++;
```

```
 struct Rect *rt,*rr;



static RectReal sum2=0.0,sum3=0.0;
int j,m;



RectReal perim2,volume1,volume2;



if (f==1) //perim
{


 if (n->branch[t].child)
          {
                   rt = &n->branch[t].rect;


      tmp_rect = RTreeCombineRect(r,rt);              //MBB(r1Uomega)
    }


 for (j=0; j<p1; j++)
{
          if(j==t) {
                   continue;
          }
          if (n->branch[j].child)
```

```
{
        rr = &n->branch[j].rect;        // rr ->  rj

        if(RTreeOverlap(&tmp_rect,rr))
        {
                rect1 = RTreeIntersectRect(&tmp_rect,rr);   //
(MBB(r1Uomega)intersect rj)
                perimeter1=RTreeRectPerimeter(&rect1);
//perim((MBB(r1Uomega)intersect rj))
        }
        else
        {
                perimeter1=0.0;
        }

        if(RTreeOverlap(rt,rr))                 //r1 intersect rj
        {
                rect2 = RTreeIntersectRect(rt, rr);//
(MBB(r1Uomega)intersect rj)
                perimeter2=RTreeRectPerimeter(&rect2);
//perim((MBB(r1Uomega)intersect rj))
        }
        else
        {
                perimeter2=0.0;
        }

        Doverlapperimeter[t][j]=perimeter1-perimeter2;
    sum2= sum2 + Doverlapperimeter[t][j];

}
```

```
            if(Doverlapperimeter[t][j]!=0.0)
            {
                  for (m=0; m<MAXKIDS(n); m++)
                  {
                        if (cand[m]!=j) {continue; }
                        else
                         {

      CheckComp(j,1,Doverlapperimeter,Doverlapvolume,cand,r,p1,success,c);
                              if(success)  break;
                         }
                  }
            }
      }
       if(sum2==0.0)                    //line 30
       {
            c=t;
           printf("success=TRUE;\n");
            return success=TRUE;
       }
      else return success=FALSE;


      }

// volume

      else //if (f==2)
      {




            if (n->branch[t].child)
```

```
{
        rt = &n->branch[t].rect;


         tmp_rect = RTreeCombineRect(r,rt); //MBB(r1Uomega)
}


 for (j=0; j<p1; j++)
{
        if(j==t) {
        continue;
        }

        if (n->branch[j].child)
        {
        rr = &n->branch[j].rect;        //to rr einai to rj

                if(RTreeOverlap(&tmp_rect,rr))
                {
                rect1 = RTreeIntersectRect(&tmp_rect, rr);//
(MBB(r1Uomega)intersect rj)
                volume1=RTreeRectVolume(&rect1);
//volume((MBB(r1Uomega)intersect rj))
                }
                else
                {
                volume1=0.0;
                }

                if(RTreeOverlap(rt,rr))                 //r1 intersect rj
                {
```

```
                    rect2 = RTreeIntersectRect(rt, rr);//
(MBB(r1Uomega)intersect rj)
                    perim2=RTreeRectVolume(&rect2);
//perim((MBB(r1Uomega)intersect rj))
                    }
                    else
                    {
                    volume2=0.0;
                    }

                    Doverlapvolume[t][j]=volume1-volume2;
                     sum3= sum3 + Doverlapvolume[t][j];

                }


                if(Doverlapvolume[t][j]!=0.0)
                {
                    for (m=0; m<MAXKIDS(n); i++)
                     {
                        if (cand[m]==j)
                        {
                            continue;
                        }
                        else
                        {

    CheckComp(j,1,Doverlapperimeter,Doverlapvolume,cand,r,p1,success,c);
                            if(success)
                            {
                            break;
                            }
                        }
```

```
                                }
                        }
                }
                 if(sum3==0.0)                  //line 30
                 {
                        c=t;
                        return success=TRUE;
                 }
                else return success=FALSE;

        }
}
```

```
RectReal RTreePickBranch(struct Rect *R, struct Node *N) {
        struct Rect *r = R;
```

```
 struct Node *n = N;
 struct Rect *rr;
int i, first_time=1;
RectReal increase, bestIncr=(RectReal)-1, area, bestArea;
int best;
struct Rect tmp_rect;
assert(r && n);



RectReal voli[MAXKIDS(n)];
RectReal perimi[MAXKIDS(n)];
RectReal keepi[MAXKIDS(n)];//keep i with volume=0

int z;



int flag1=0; //=1 if cov=omega
int m=0;
int flag2=0; //=1 if volume[i]=0
RectReal min;

RectReal perimeter1[MAXKIDS(n)];        //perim(MBB(rrUr)  line 8
RectReal perimeter2[MAXKIDS(n)];        //perim(r)
RectReal Dperim[MAXKIDS(n)];            //Dperim



int length;



RectReal sum=0.0;
```

```
struct Rect *r1;
struct Rect *r2;
RectReal perim1,perim2;
RectReal Doverlapperim[MAXKIDS(n)];     //Dperim




int cand[MAXKIDS(n)];




struct Rect rect1,rect2,temp_rect;
RTreeInitRect(&temp_rect);
RTreeInitRect(&rect1);
RTreeInitRect(&rect2);


for(z=0; z<MAXKIDS(n); z++) { voli[z]=10000; perimi[z]=10000;
keepi[z]=0; }//we want min so we put sth big in  vectors









for (i=0; i<MAXKIDS(n); i++)
```

```
        {
                if (n->branch[i].child)
                {

                        rr = &n->branch[i].rect;

                        if(RTreeContained(rr,r)) {
        //cov!=o calculate vol kai perim

                                flag1=1;
                                voli[i]=RTreeRectVolume(rr);
                                perimi[i]=RTreeRectPerimeter(rr);

                        }

                }
        }


        if(flag1==1) {
                for(z=0; z<MAXKIDS(n); z++) {
                        if(voli[z]==0)  {
                                flag2=1;
                                keepi[m]=perimi[z];
                                m++;
                        }               //keep  i with volume 0 and find this i with min
perim
                }
        }
```

```
if (flag2==1) {
                min=keepi[0];                                   //line 3
        for(z=1; z<m-1; z++) {
                if(min>keepi[z]){
                        min=keepi[z];
                }
        }
        return min;
}
if ((flag2==0) &(flag1==1)) {


                                                        //line 5-6
        min=voli[0];

                                                        //line 3
        for(z=1; z<MAXKIDS(n); z++) {
                if(min>keepi[z]){
                        min=keepi[z];
                }
        }
        return min;

}



                                                                //Dperim=
perim(MBB(rrUr)  - perim(rr)

        for (i=0; i<MAXKIDS(n); i++)
        {
```

```
            if (n->branch[i].child)
            {
                    rr = &n->branch[i].rect;

                    tmp_rect = RTreeCombineRect(r, rr);


                    perimeter1[i]=RTreeRectPerimeter(&tmp_rect);

                    perimeter2[i]=RTreeRectPerimeter(r);

                    Dperim[i]= perimeter1[i] - perimeter2[i];


            }
    }

    length=i-1;  //sorting line 9

    sort(Dperim,length);


                                                                    //line 10



i=0;
     if (n->branch[i].child)
            {
                    r1 = &n->branch[i].rect;
```

```
        }

    tmp_rect = RTreeCombineRect(r, r1); //MBB(r1Uomega)


     for (i=1; i<MAXKIDS(n); i++)
    {
        if (n->branch[i].child)
        {
            rr = &n->branch[i].rect;        // rr -> rj

            if(RTreeOverlap(&tmp_rect,rr))
            {
                rect1 = RTreeIntersectRect(&tmp_rect, rr);//
(MBB(r1Uomega)intersect rj)
                perim1=RTreeRectPerimeter(&rect1);
//perim((MBB(r1Uomega)intersect rj))
            }
            else
            {
                perim1=0;
            }

            if(RTreeOverlap(r1,rr))                 //r1 intersect rj
            {
                rect2 = RTreeIntersectRect(r1, rr);//
(MBB(r1Uomega)intersect rj)
                perim2=RTreeRectPerimeter(&rect2);
//perim((MBB(r1Uomega)intersect rj))
            }
            else
            {
                perim2=0;
```

```
                }

                Doverlapperim[i]=perim1-perim2;
                sum= sum + Doverlapperim[i];

        }
}


int j;
 if(sum==0.0) {  return TRUE; }                    //line 11-12
 else                                                 //line 13
 {
        for(j=0; j<=i-1; j++)
        {
                if(Doverlapperim[j]>0) { p++; }
        }
 }

 //line 14


 for (i=0; i<MAXKIDS(n); i++)
  {
                cand[i]=0;

 }
 int success = FALSE;            //line 15
RectReal volume;

 int c;
```

```
for (i=0; i<MAXKIDS(n); i++)
{
        if (n->branch[i].child)
        {
                rr = &n->branch[i].rect;        // rr -> ri        line 16


                tmp_rect = RTreeCombineRect(r, rr);
                volume=RTreeRectVolume(&temp_rect);
                if(volume==0.0) { break; }

        }
        if(volume==0.0) {break;}
}
```

```
RectReal Doverlapperimeter[MAXKIDS(n)][p];//Doverlapperim[][] keep t = 1 to (p-1)

RectReal Doverlapvol[MAXKIDS(n)][p];


        //line 22
        for (i=0; i<MAXKIDS(n); i++)
    {
                for (j=0; j<p; j++)
                {
                Doverlapperimeter[i][j]=0.0;
              Doverlapvol[i][j]=0.0;
                }
        }
```

```
        if(volume==0.0)
{ CheckComp(1,1,Doverlapperimeter,Doverlapvol,cand,r,p,success,c); }
//line 16-19  1 for perimeter
        else { CheckComp(1,2,Doverlapperimeter,Doverlapvol,cand,r,p,success,c); }
                // 2 for volume




        if(success) {return c; }    //line 33-34
        else
        {
        int minimum=cand[0];
        for (i=1; i<MAXKIDS(n); i++)
          {
                        if(cand[i]!=0)
                        {
                                if(minimum>cand[i])
                                {
                                        minimum=cand[i];
                                }
                        }
                }

        return(minimum);}
}
```

```
// Add a branch to a node.  Split the node if necessary.
// Returns 0 if node not split.  Old node updated.
// Returns 1 if node split, sets *new_node to address of new node.
// Old node updated, becomes one of two.
int RTreeAddBranch(struct Branch *B, struct Node *N, struct Node **New_node)
{
        struct Branch *b = B;
         struct Node *n = N;
         struct Node **new_node = New_node;
         int i;

        assert(b);
        assert(n);

printf("addbranch\n");
        if (n->count < MAXKIDS(n))  /* split won't be necessary */
        {
                printf("split won't be necessary\n");
                for (i = 0; i < MAXKIDS(n); i++)  /* find empty branch */
                {
                        if (n->branch[i].child == NULL)
                        {
                                n->branch[i] = *b;
                                n->count++;

                                break;
                        }

                }
```

```
            printf("addbranch n_count %d\n",n->count);
            return 0;
        }
        else
        {

            printf("rtreesplitnode\n");

            printf("# of splits=%d\n",splits_counter);
            splits_counter++;
            RTreeSplitNode(n, b, new_node);

            return 1;
        }


}



// Disconnect a dependent node.
//
void RTreeDisconnectBranch(struct Node *n, int i)
{
        assert(n && i>=0 && i<MAXKIDS(n));
        assert(n->branch[i].child);

        RTreeInitBranch(&(n->branch[i]));
        n->count--;
        printf("disconect\n");
}
```

```
// Make a new index, empty.  Consists of a single node.
struct Node * RTreeNewIndex()
{
        struct Node *x;
        x = RTreeNewNode();
        x->level = 0; /* leaf */
        return x;
}



// Search in an index tree or subtree for all data retangles that
// overlap the argument rectangle.
// Return the number of qualifying data rects.
int RTreeSearch(struct Node *N, struct Rect *R)
{
        struct Node *n = N;
        struct Rect *r = R;
        int hitCount = 0;
        int i;

        assert(n);
```

```
        assert(n->level >= 0);
        assert(r);

        if (n->level > 0) /* this is an internal node in the tree */
        {printf("n->level>0 %d\n",n->level);
                for (i=0; i<NODECARD; i++)
                        if (n->branch[i].child &&
                           RTreeOverlap(r,&n->branch[i].rect))
                        {       printf("jhh\n");
                                hitCount += RTreeSearch(n->branch[i].child, R);
                        }
        }
        else /* this is a leaf node */
        {       printf("n->level=0 leaf node %d\n",n->level);

                for (i=0; i<LEAFCARD; i++)
                        if (n->branch[i].child &&
                           RTreeOverlap(r,&n->branch[i].rect))
                        {
                                hitCount++;

                        }
        }

        return hitCount;
}



// Inserts a new data rectangle into the index structure.
// Recursively descends tree, propagates splits back up.
// Returns 0 if node was not split.  Old node updated.
```

```
// If node was split, returns 1 and sets the pointer pointed to by
// new_node to point to the new node.  Old node updated to become one of two.
// The level argument specifies the number of steps up from the leaf

// level to insert; e.g. a data rectangle goes in at level = 0.

int RTreeInsertRect2(struct Rect *R,int Tid, struct Node *N, struct Node
**New_node, int Level)
{

        struct Rect *r = R;
        int tid = Tid;
        struct Node *n = N, **new_node = New_node;
        int level = Level;

        int i;
        struct Branch b;
        struct Node *n2;

        //assert(r && n && new_node);
        assert(level >= 0 && level <= n->level);

        // Still above level for insertion, go down tree recursively

        // in the beggining n->level = root->level  because the 3rd argument in
RTreeInsertRect2 is root
        // => level=0


        if (n->level > level)
        {
```

```
        i = RTreePickBranch(r, n);


        if (!RTreeInsertRect2(r, tid, n->branch[i].child, &n2, level))
        {

                printf("child was not split\n");

                n->branch[i].rect =
                        RTreeCombineRect(r,&(n->branch[i].rect));
                return 0;
        }
        else    // child was split
        {       printf("child was  split\n");
                n->branch[i].rect = RTreeNodeCover(n->branch[i].child);
                b.child = n2;
                b.rect = RTreeNodeCover(n2);
                return RTreeAddBranch(&b, n, new_node);
        }
}

// Have reached level for insertion. Add rect, split if necessary
//
else if (n->level == level)
{

        b.rect = *r;
        b.child = (struct Node *) tid;


        return RTreeAddBranch(&b, n, new_node);    //return 1 if split
```

```
        }
        else
        {
                printf("Not supposed to happen");

                /* Not supposed to happen */
                assert (FALSE);
                return 0;
        }
}



// Insert a data rectangle into an index structure.
// RTreeInsertRect provides for splitting the root;
// returns 1 if root was split, 0 if it was not.
// The level argument specifies the number of steps up from the leaf
// level to insert; e.g. a data rectangle goes in at level = 0.
// RTreeInsertRect2 does the recursion.
//

// parameter 4 is always zero which means to add from the root.
//  id  = id +1 for each new rectangle's insertion

void RTreeInsertRect(struct Rect *R, int Tid, struct Node **Root, int Level)
{
         struct Rect *r = R;
         int tid = Tid;
        struct Node **root = Root;
        int level = Level;
        int i;
        struct Node *newroot;
```

```
	struct Node *newnode;
	struct Branch b;
	int result;

	assert(r && root);
	assert(level >= 0 && level <= (*root)->level);


	if((*root)->init->boundary[0]==0)  {   (*root)->init=r; }


	//for (i=0; i<NUMSIDES; i=i+2)
		//assert(r->boundary[i] <= r->boundary[i+1]);

	if (RTreeInsertRect2(r, tid, *root, &newnode, level))  /* root split */
	{

		printf("root split\n");


		newroot = RTreeNewNode();                                   /* grow a
new root, & tree taller */
		newroot->level = (*root)->level + 1;                //the level = level +1
because of the root split
		b.rect = RTreeNodeCover(*root);                     //root in a rect
		b.child = *root;
//old root is child of new root
		RTreeAddBranch(&b, newroot, NULL);
//create a new branch
		b.rect = RTreeNodeCover(newnode);
		b.child = newnode;

		RTreeAddBranch(&b, newroot, NULL);
```

```
                *root = newroot;
        //new rect is now the root

                (*root)->init=r;

                result = 1;
        }
        else
                result = 0;

//      return result;
}




// Allocate space for a node in the list used in DeletRect to
// store Nodes that are too empty.
struct ListNode * RTreeNewListNode()
{
        return (struct ListNode *) malloc(sizeof(struct ListNode));
        //return new ListNode;
}




void RTreeFreeListNode(struct ListNode *p)
{//n->countn->count
        free(p);
        //delete(p);
```

```
}



// Add a node to the reinsertion list.  All its branches will later
// be reinserted into the index structure.
void RTreeReInsert(struct Node *n, struct ListNode **ee)
{
        struct ListNode *l;

        l = RTreeNewListNode();
        l->node = n;
        l->next = *ee;
        *ee = l;
}



// Delete a rectangle from non-root part of an index structure.
// Called by RTreeDeleteRect.  Descends tree recursively,
// merges branches on the way back up.
// Returns 1 if record not found, 0 if success.
//
int RTreeDeleteRect2(struct Rect *R, int Tid, struct Node *N, struct ListNode **Ee)
{
        struct Rect *r = R;
        int tid = Tid;
        struct Node *n = N;
        struct ListNode **ee = Ee;
        int i;

        assert(r && n && ee);
        assert(tid >= 0);
```

```
assert(n->level >= 0);

if (n->level > 0)  // not a leaf node
{printf("delete2 n->level>0 %d\n",n->level);
   for (i = 0; i < NODECARD; i++)
   {
        if (n->branch[i].child && RTreeOverlap(r, &(n->branch[i].rect)))
        {
                if (!RTreeDeleteRect2(r, tid, n->branch[i].child, ee))
                {
                        if (n->branch[i].child->count >= MinNodeFill)
                                n->branch[i].rect = RTreeNodeCover(
                                        n->branch[i].child);
                        else
                        {
                                // not enough entries in child,
                                // eliminate child node
                                RTreeReInsert(n->branch[i].child, ee);
                                RTreeDisconnectBranch(n, i);
                        }
                        return 0;
                }
        }
   }
   return 1;
}
else  // a leaf node
{printf("delete n->level=0 %d\n",n->level);
        for (i = 0; i < LEAFCARD; i++)
        {
                if (n->branch[i].child &&
                   n->branch[i].child == (struct Node *) tid)
```

```
                        {       printf("RTreeDisconnectBranch(n, i);\n");
                                RTreeDisconnectBranch(n, i);

                                return 0;
                        }
                }
                return 1;
        }
}



// Delete a data rectangle from an index structure.
// Pass in a pointer to a Rect, the tid of the record, ptr to ptr to root node.
// Returns 1 if record not found, 0 if success.
// RTreeDeleteRect provides for eliminating the root.
//
int RTreeDeleteRect(struct Rect *R, int Tid, struct Node **Nn)
{
        struct Rect *r = R;
        int tid = Tid;
        struct Node **nn = Nn;
        int i;
        struct Node *tmp_nptr;
        struct ListNode *reInsertList = NULL;
        struct ListNode *e;

        assert(r && nn);
        assert(*nn);
        assert(tid >= 0);

        if (!RTreeDeleteRect2(r, tid, *nn, &reInsertList))
```

```
{
        /* found and deleted a data item */

        /* reinsert any branches from eliminated nodes */
        printf("if (!RTreeDeleteRect2(r, tid, *nn, &reInsertList))\n");
        while (reInsertList)
        {       printf("while (reInsertList)\n");
                tmp_nptr = reInsertList->node;
                for (i = 0; i < MAXKIDS(tmp_nptr); i++)
                {
                        if (tmp_nptr->branch[i].child)
                        {
                                RTreeInsertRect(
                                        &(tmp_nptr->branch[i].rect),
                                        (int)tmp_nptr->branch[i].child,
                                        nn,
                                        tmp_nptr->level);
                        }
                }
                e = reInsertList;
                reInsertList = reInsertList->next;
                RTreeFreeNode(e->node);
                RTreeFreeListNode(e);
        }

        /* check for redundant root (not leaf, 1 child) and eliminate
        */
        if ((*nn)->count == 1 && (*nn)->level > 0)
        {
                for (i = 0; i < NODECARD; i++)
                {
                        tmp_nptr = (*nn)->branch[i].child;
```

```
                                if(tmp_nptr)
                                        break;
                        }
                        assert(tmp_nptr);
                        RTreeFreeNode(*nn);
                        *nn = tmp_nptr;
                }
                return 0;
        }
        else
        {
                return 1;
        }
}
```

```
int timeval_subtract(struct timeval *result, struct timeval *t2, struct timeval *t1)
{
   long int diff = (t2->tv_usec + 1000000 * t2->tv_sec) - (t1->tv_usec + 1000000 * t1->tv_sec);
   result->tv_sec = diff / 1000000;
   result->tv_usec = diff % 1000000;

   return (diff<0);
}
```

```
void timeval_print(struct timeval *tv)
{
    char buffer[30];
    time_t curtime;

    printf("%ld.%06ld", tv->tv_sec, tv->tv_usec);
    curtime = tv->tv_sec;
    strftime(buffer, 30, "%m-%d-%Y  %T", localtime(&curtime));
    printf(" = %s.%06ld\n", buffer, tv->tv_usec);
}




int main() {


//          clock_t start, stop;


            /* Start timer */
//          assert((start = clock())!=-1);

initial();
struct timeval tvBegin, tvEnd, tvDiff;

    // begin
    gettimeofday(&tvBegin, NULL);
    timeval_print(&tvBegin);
```

```
struct Node* root = RTreeNewIndex();
int i, nhits;



printf("nrects = %d\n", nrects);
/*
 * Insert all the data rects.
 * Notes about the arguments:
 * parameter 1 is the rect being inserted,
 * parameter 2 is its ID. NOTE: *** ID MUST NEVER BE ZERO ***, hence the +1,
 * parameter 3 is the root of the tree. Note: its address is passed
 * because it can change as a result of this call, therefore no other parts
 * of this code should stash its address since it could change undernieth.
 * parameter 4 is always zero which means to add from the root.
 */

int ft=0;

for(i=0; i<nrects; i++) {

        RTreeInsertRect(&rects[i], i+1, &root, 0); // i+1 is rect ID. Note: root can change
 printf("rect{ %f %f %f %f } inserted\n\n\n",rects[i].boundary[0],rects[i].boundary[1],rects[i].boundary[2],rects[i].boundary[3]);
ft++;
if(ft==10000) { ft=0;
                gettimeofday(&tvEnd, NULL);
               timeval_print(&tvEnd);

                // diff
```

```
                        timeval_subtract(&tvDiff, &tvEnd, &tvBegin);
                        printf("hi %ld.%06ld\n", tvDiff.tv_sec, tvDiff.tv_usec);
                        printf("# of splits=%d  node_counter=%d
nodeAcc=%d\n",splits_counter-1,node_counter,nodeAcc);
                        //printCurrentTimeNsec();
                        //clock_gettime(CLOCK_REALTIME, &ts);
                        //elapsed = clock_gettime();    //(double)(clock() - start);
                        // printf("time per 10000 entries: %f\n", elapsed);
                }
        }

    nhits = RTreeSearch(root, &search_rect);
        printf("Search resulted in %d hits\n", nhits);
//
//
//      i=0;
//      int del_res=RTreeDeleteRect(&rects[i], i+1, &root);
//      printf("delete result is %d\n",del_res);
//
        RTreePrintNode(root, 2);
        printf("%d\n",MAXCARD);
        printf("# of splits=%d  node_counter=%d nodeAcc=%d\n",splits_counter-
1,node_counter,nodeAcc);



        //printCurrentTimeNsec();
        /* Stop timer */
        //stop = clock();
        //t = (float) (stop-start)/CLOCKS_PER_SEC;

        //printf("Run time: %f\n", t);
```

```
        //end
    gettimeofday(&tvEnd, NULL);
    timeval_print(&tvEnd);

    // diff
    timeval_subtract(&tvDiff, &tvEnd, &tvBegin);
    printf("%ld.%06ld\n", tvDiff.tv_sec, tvDiff.tv_usec);
}
```

end of rrstratree.c

Τα αρχεία headers που χρησιμοποιήθηκαν είναι τα εξής:

/* assert.h

||

|| This header file contains "assertion" macros for debug purposes.

*/

```
#ifndef NDEBUG
#define _assert(ex) {if (!(ex)){fflush(stdout);fprintf(stderr,"Assertion failed: file %s, line %d\n", __FILE__, __LINE__);abort();}}
#define assert(ex) {if (!(ex)){fflush(stdout);fprintf(stderr,"Assertion failed: file %s, line %d\n", __FILE__, __LINE__);abort();}}
#else
#define _assert(ex) ;
#define assert(ex) ;
#endif
```

//card.h

```
#ifndef __CARD__
#define __CARD__

extern int NODECARD;
extern int LEAFCARD;

/* balance criteria for node splitting */
/* NOTE: can be changed if needed. */
#define MinNodeFill (NODECARD / 2)
#define MinLeafFill (LEAFCARD / 2)

#define MAXKIDS(n) ((n)->level > 0 ? NODECARD : LEAFCARD)
#define MINFILL(n) ((n)->level > 0 ? MinNodeFill : MinLeafFill)

#endif
```

//Index.h

```
#ifndef _INDEX_
#define _INDEX_

/* PGSIZE is normally the natural page size of the machine */
#define PGSIZE         512
#define NUMDIMS    2         /* number of dimensions */
#define NDEBUG

typedef float RectReal;



/*-----------------------------------------------------------------------------
| Global definitions.
-----------------------------------------------------------------------------*/

#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

#define NUMSIDES 2*NUMDIMS

struct Rect
{
        RectReal boundary[NUMSIDES]; /* xmin,ymin,...,xmax,ymax,... */
};
```

```
struct Node;

struct Branch
{
        struct Rect rect;
        struct Node *child;
};

/* max branching factor of a node */
#define MAXCARD (int)((PGSIZE-(2*sizeof(int))) / sizeof(struct Branch))

struct Node
{
        int count;
        int level; /* 0 is leaf, others positive */
        struct Rect *init;
        struct Branch branch[MAXCARD];
};

struct ListNode
{
        struct ListNode *next;
        struct Node *node;
};

/*
 * If passed to a tree search, this callback function will be called
 * with the ID of each data rect that overlaps the search rect
 * plus whatever user specific pointer was passed to the search.
 * It can terminate the search early by returning 0 in which case
 * the search will return the number of hits found up to that point.
```

```
 */
//typedef int (*SearchHitCallback)(int id, void* arg);


extern int RTreeSearch(struct Node*, struct Rect*); //edw eixe parametro
*SearchHitCallback
extern void RTreeInsertRect(struct Rect*, int, struct Node**, int depth);
extern int RTreeDeleteRect(struct Rect*, int, struct Node**);
extern struct Node * RTreeNewIndex();
extern struct Node * RTreeNewNode();
extern void RTreeInitNode(struct Node*);
extern void RTreeFreeNode(struct Node *);
extern void RTreePrintNode(struct Node *, int);
extern void RTreeTabIn(int);
extern struct Rect RTreeNodeCover(struct Node *);
extern void RTreeInitRect(struct Rect*);
extern struct Rect RTreeNullRect();
extern RectReal RTreeRectVolume(struct Rect *R);
extern RectReal RTreeRectPerimeter(struct Rect *R);
extern struct Rect RTreeCombineRect(struct Rect*, struct Rect*);
extern struct Rect RTreeIntersectRect(struct Rect*, struct Rect*);
extern int RTreeOverlap(struct Rect*, struct Rect*);
extern void RTreePrintRect(struct Rect*, int);
extern int RTreeAddBranch(struct Branch *, struct Node *, struct Node **);
extern RectReal RTreePickBranch(struct Rect *, struct Node *);
extern void RTreeDisconnectBranch(struct Node *, int);
extern void RTreeSplitNode(struct Node*, struct Branch*, struct Node**);

extern int RTreeSetNodeMax(int);
extern int RTreeSetLeafMax(int);
extern int RTreeGetNodeMax();
extern int RTreeGetLeafMax();
```

```
#endif /* _INDEX_ */
```

//Split.h

```
/*-------------------------------------------------------------------------
| Definitions and global variables used in linear split code.
-------------------------------------------------------------------------*/

#define METHODS 1

struct Branch BranchBuf[MAXCARD+1];
int BranchCount;
struct Rect CoverSplit;

/* variables for finding a partition */
struct PartitionVars
{
        int partition[MAXCARD+1];
        int total, minfill;
        int taken[MAXCARD+1];
        int count[2];
        struct Rect cover[2];
        } Partitions[METHODS];
```

7.ΑΝΑΦΟΡΕΣ

[1] N. Beckmann, B. Seeger : ARevised R*-tree in Comparison with Related Index Structures.

[2] N. Beckmann, H. –P. Kriegel , R. Schneider, B.seegeer : The R*-Tree : An Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference 1990: 322-331

[3]A. Guttman. R-trees: A dynamic index structure for spatial searching. SIGMOD Conference 1984: 47-57, 1984

[4] http://www.mathematic.uni-marburg.de/~seegeer/rrstar/, 2009

[5] N. Beckmann, B. Seeger : A Benchmark for Multidimensional Index Structures

[6] N. Rousopoulos, D. Leifker : Direct Spatial Search on Pictorial Databases Using Packed R-trees.

[7] I. Kamel, C. Faloutsos : Hilbert R-tree: An improved R-tree using fractals

[8] A.Kemper , L. Neumann : Algorithms and Data Structures for Persistent Data : The R-tree.

[9] C. Wu, L. Chang, T. Kuo : An Efficient R-Tree Implementation over Flash-Memory Storage Systems.

[10] R.Elmari , S.B. Navathe : Θεμελιώδης Αρχές Βάσεων Δεδομένων

[11] Π. Μποζάνης : Δομές Δεδομένων

[12] Παν.Πειραιά : Β-δέντρα (lecture b_trees.pdf)

[13] C.Bohm, S. Berchtold, D. Keim : Searching in high-dimesional spaces: Index structures for improving the performance of multimedia databases.ACM Computing Surveys 33(3):322-373 (2001)

[14] A. Guttman, M.Stonebraker : R-Trees: A Dynamic Index Structure For Spatial Searching (Original Algorithm and test)

[15] P.Brooke :A C implementation of the R-tree data structure using the volume of the bounding sphere as the area metric for nodes

[16] Κ.Μαϊστρέλης : Γνωριμία με την PostGreSQL

[17] S.Berchtold, D.Keim. H.-P. Kriegel: The X-tree: An Index Structure for High-Dimensional Data. VLDB Conference 1996: 28-39

[18] J.Van den Bercken, B.seeger, P.Widmayer: AGeneric Approach to Bulk Loading Multidimensional Index Structures.VLDB 1997: 406:415

[19] UC Irvine KDD Archive, http://kdd.ics.uci.edu/,2002

[20] D. Knuth “The art of computer programming” Vol 3 sorting and searching, Addison-Wesley Publ Co, Reading , Mass , 1973