Πανεπιστήμιο Πειραιώς - Τμήμα Πληροφορικής
Πρόγραμμα Μεταπτυχιακών Σπουδών
«Προηγμένα Συστήματα Πληροφορικής»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Ερωτήματα Οπτικού Πεδίου**
Ονοματεπώνυμο Φοιτήτριας	**'Αννα Γιαννούτσου**
Πατρώνυμο	**Κωνσταντίνος**
Αριθμός Μητρώου	**ΜΠΣΠ/08011**
Επιβλέπων	**Ιωάννης Θεοδωρίδης, Αν. Καθηγητής**

Ημερομηνία Παράδοσης **Σεπτέμβριος 2011**

Τριμελής Εξεταστική Επιτροπή

Πρόλογος

Τα ερωτήματα οπτικού πεδίου (line of sight - LOS - queries) αποτελούν μια προηγμένη εφαρμογή στα συστήματα Spatial DBMS και GIS (Geographical Information Systems). Στόχος της εργασίας είναι η επισκόπηση σύγχρονων πρακτικών σχετικών με τα ερωτήματα οπτικού πεδίου, η επέκταση κάποιων από αυτές και η πειραματική μελέτη των πρακτικών που θα επεκταθούν.

Ερωτήματα οπτικού πεδίου (Line Of Sight) ονομάζονται τα ερωτήματα ορατότητας ανάμεσα σε δύο οντότητες με βάση κάποια περιοχή και πιθανόν άλλες οντότητες-εμπόδια. Παρουσιάζουν ιδιαίτερο ενδιαφέρον στις μέρες μας εξαιτίας της αυξημένης δημοτικότητας των έξυπνων κινητών συσκευών και τη ραγδαία ανάπτυξη της ασύρματης τεχνολογίας. Τα ερωτήματα οπτικού πεδίου είναι ένα θέμα με πολλές εφαρμογές. Τα συναντάμε στα συστήματα GPS, όπου ενδιαφερόμαστε για παράδειγμα για την κοντινότερη τράπεζα ή το κοντινότερο βενζινάδικο στην κατεύθυνση που πηγαίνουμε. Τα ερωτήματα οπτικού πεδίου δίνουν απαντήσεις σε θέματα όπως από ποια θέση είναι ορατό κάθε σημείο ενός συγκεκριμένου χώρου. Επιπλέον ερωτήσεις όπως που μπορεί να τοποθετηθεί μια αφίσα ώστε να είναι ορατή σε κάθε σημείο του διαδρόμου στο εμπορικό κέντρο, που μπορεί να τοποθετηθεί μια κάμερα ώστε να ελέγχει όσο το δυνατόν μεγαλύτερο μέρος ενός χώρου. Επίσης τουρίστες που ενδιαφέρονται για περιοχές όπου τα αξιοθέατα είναι ορατά σε αυτούς. Σε πεδία μαχών όπου άνθρωποι και όπλα είναι σε κίνηση ενδιαφέρουν ερωτήσεις όπως που βρίσκονται οι εχθροί που είναι ορατοί από μια συγκεκριμένη θέση. Άλλοι τομείς που βρίσκουν εφαρμογή τα ερωτήματα οπτικού πεδίου είναι πληροφορίες κυκλοφοριακής συμφόρησης, πληροφορίες για τον καιρό τοπικά, συστήματα ανίχνευσης, κινητή διαφήμιση, κινητές υπηρεσίες εντοπισμού, πλοήγηση.

Κλείνοντας, θα ήθελα να ευχαριστήσω τον επιβλέποντα Αναπληρωτή Καθηγητή Θεοδωρίδη Ιωάννη για την συνεχή του υποστήριξη και ενθάρρυνση στην εκπόνηση της παρούσας εργασίας. Επίσης ευχαριστώ τα μέλη της εξεταστικής επιτροπής για το ενδιαφέρον που έδειξαν για τη δουλειά μου.

Περιεχόμενα

1

Εισαγωγή

Οι χωρικές ερωτήσεις μπορούν να διακριθούν στις ακόλουθες κατηγορίες σύμφωνα με τα [7] και [8].

- *Ερωτήσεις σημείου*: Βρες όλα αντικείμενα που περικλείουν ένα δοθέν σημείο.
- *Ερωτήσεις εύρους*: Βρες όλα τα αντικείμενα μέσα σε μια (συνήθως ορθογώνια) περιοχή.
- *Ερωτήσεις κοντινότερου γείτονα*: Βρες το πλησιέστερο αντικείμενο σε σχέση με ένα σημείο.
- *Ερωτήματα χωρικής σύνδεσης* (μεταξύ δύο συνόλων αντικειμένων): Βρες όλα τα ζεύγη αντικειμένων (ένα από κάθε σύνολο) που ικανοποιούν μια χωρική συνθήκη.

Οι πρακτικές LOS που έχουν αναπτυχθεί μέχρι σήμερα, επεξεργάζονται διαφορετικού τύπου ερωτήματα οπτικού πεδίου τα οποία καλύπτουν μεγάλο εύρος εφαρμογών και προσεγγίζουν όσο γίνεται περισσότερο τον πραγματικό κόσμο με τα οπτικά εμπόδια, τα αντικείμενα σε κίνηση, τα μη σημειακά αντικείμενα δεδομένων κτλ. Επίσης κάποια από αυτά τα ερωτήματα αποτελούν τη βάση ανάπτυξης άλλων ερωτημάτων που επεκτείνουν τα υπάρχοντα.

Στην παρούσα εργασία επιλέγονται δύο τύποι ερωτημάτων από τις τεχνικές που μελετήθηκαν και με τις κατάλληλες τροποποιήσεις επεκτείνονται έτσι ώστε η μία να καλύπτει τις ιδιότητες του χώρου τις οποίες ικανοποιεί η άλλη. Τελικά προκύπτουν δύο όμοιοι τύποι ερωτημάτων οπτικού πεδίου, τους οποίους στη συνέχεια μπορούμε να τους συγκρίνουμε.

Τα ερωτήματα οπτικού πεδίου διαχειρίζονται χωρικά δεδομένα, δηλαδή όχι απλούς τύπους τιμών, όπως αριθμοί και συμβολοσειρές. Τα αντικείμενα του χώρου προσεγγίζονται με σημεία, γραμμές και πολύγωνα. Σε αυτό το πλαίσιο χρησιμοποιείται συχνά το MBR (Minimum Bounding Rectangle) δηλαδή το μικρότερο ορθογώνιο παραλληλόγραμμο που περιβάλλει ένα αντικείμενο.

Το πρόβλημα με τα χωρικά δεδομένα είναι ότι δεν έχουν κάποια φυσική διάταξη και έτσι δεν μπορούν να χρησιμοποιηθούν τα συνηθισμένα ευρετήρια.

Πολλές γνωστές μέθοδοι αποθήκευσης, ευρετηριοποίησης και αναζήτησης δεδομένων βασίζονται στην έννοια της διάταξης. Για να επιλυθεί αυτό το πρόβλημα χρησιμοποιούνται οι γραμμές διάσχισης του χώρου και τα χωρικά ευρετήρια. Τα πιο συνηθισμένα χωρικά ευρετήρια είναι το R-tree [11], [12] και το Quadtree. Τα χωρικά ευρετήρια παρέχουν καλύτερες αποδόσεις στους υπολογισμούς.

Στις πρακτικές LOS ως χωρικό ευρετήριο χρησιμοποιείται συνήθως το R-tree, επειδή είναι ευρέως διαδεδομένο ευρετήριο για τα χωρικά δεδομένα [13] αλλά και εξαιτίας της αποδοτικότητάς του [14]. Έτσι και στις πρακτικές που επιλέχθηκαν για να επεκταθούν χρησιμοποιείται το R-tree. Το R-tree αποτελείται από μια ιεραρχία από MBRs, καθένα από τα οποία αντιστοιχεί σε έναν κόμβο του δένδρου και περικλείει όλα τα MBRs που βρίσκονται στο υποδένδρο του. Τα δεδομένα βρίσκονται αποθηκευμένα στα φύλλα του δένδρου και είναι χωρισμένα με βάση κάποια ευρετική μέθοδο που έχει σκοπό να ελαχιστοποιήσει το κόστος εισόδου - εξόδου στο ευρετήριο.

Τα δεδομένα πάνω στα οποία εκτελούνται οι τεχνικές ερωτημάτων οπτικού πεδίου που επεκτείνονται και επομένως και αυτά με βάση τα οποία εκτελούνται τα πειράματα είναι χωρικά δεδομένα. Πρόκειται για αντικείμενα που αναπαρίστανται με MBRs χρησιμοποιώντας τις συντεταγμένες των κορυφών από τις οποίες αποτελούνται. Για τα πειράματα, επομένως και τις συγκρίσεις των αλγορίθμων που θα επεκταθούν, χρησιμοποιούνται δεδομένα με αντικείμενα από τον πραγματικό κόσμο αλλά και τεχνητά δεδομένα. Τα δεδομένα αυτά οργανώνονται σε ευρετήρια τύπου R-tree και στη συνέχεια εκτελούνται οι αλγόριθμοι εύρεσης κοντινότερων γειτόνων πάνω σε αυτά τα ευρετήρια.

Συνοψίζοντας η εργασία χωρίζεται σε τρία θεματικά μέρη. Το πρώτο μέρος αποτελείται από την επισκόπηση των σχετικών εργασιών. Το δεύτερο μέρος περιλαμβάνει την συγκριτική παρουσίαση των εργασιών που μελετήθηκαν και την επιλογή δύο πρακτικών από αυτές που μελετήθηκαν, ώστε να επεκταθούν. Στη συνέχεια υλοποιείται το δύσκολο κομμάτι της επέκτασης αυτών των υπαρχόντων αλγορίθμων. Το τρίτο μέρος αφορά στην πειραματική μελέτη των τεχνικών που επεκτάθηκαν και στην εξαγωγή συμπερασμάτων.

2

Επισκόπηση Σύγχρονων Πρακτικών

Τα ερωτήματα οπτικού πεδίου χωρίζονται σε διάφορες κατηγορίες ανάλογα με τα διαφορετικού τύπου ερωτήματα που επιδιώκουν να επιλύσουν. Κάποια αναφέρονται σε κινούμενα αντικείμενα, κάποια λαμβάνουν υπόψη τα οπτικά εμπόδια [15] ή/και η κατεύθυνση των αντικειμένων, σε κάποια το αντικείμενο που θέτει το ερώτημα αναπαριστάνεται με σημείο και σε κάποια άλλα με γραμμή ή πολύγωνο.

Nearest Surrounder (NS): Δοθέντος ενός συνόλου αντικειμένων O και ενός σημείου ερωτήματος q ένα ερώτημα NS ανακτά από το O τους κοντινότερους γείτονες σε διακριτές γωνίες ως προς το q. Τα ερωτήματα αυτά εφαρμόζονται και σε περιβάλλοντα όπου τα αντικείμενα και το σημείο ερωτήματος κινούνται. Ερωτήματα αυτού του τύπου εξετάζει το [3].

Continuous Nearest Neighbor (CNN): Δοθέντος ενός συνόλου δεδομένων στον πολυδιάστατο χώρο και μιας γραμμής ερωτήματος, το CNN ερώτημα ανακτά τους κοντινότερους γείτονες από το σύνολο δεδομένων για κάθε σημείο της γραμμής ερωτήματος. Τα ερωτήματα αυτού του τύπου δεν λαμβάνουν υπόψη τα εμπόδια του χώρου. Άρθρα αυτής της κατηγορίας ερωτημάτων είναι τα [22], [24], [25], [28]-[30].

Μια γενίκευση του CNN ερωτήματος είναι το ερώτημα εύρους (*Range Query*) [36], όπου το αντικείμενο που θέτει το ερώτημα δεν ανήκει στο μονοδιάστατο χώρο, όπως το σημείο ή η γραμμή ερωτήματος, αλλά έχει ίσες ή περισσότερες από δύο διαστάσεις. Δηλαδή είναι υπερκύβοι για τους οποίους αναζητούνται κοντινότεροι γείτονες.

Visible Nearest Neighbor (VNN): Το ερώτημα αυτό βρίσκει το κοντινότερο αντικείμενο που είναι ορατό από το σημείο του ερωτήματος. Ο αλγόριθμος είναι φτιαγμένος για ένα σταθερό σημείο και όχι για ένα ευθύγραμμο τμήμα που περιέχει πολλαπλά σημεία ερωτήματος. Η βασική ιδέα είναι να γίνει αναζήτηση κοντινότερων γειτόνων και κατόπιν να εξεταστεί η συνθήκη ορατότητας με τρόπο σταδιακό. Τα άρθρα [1] και [6] εξετάζουν ερωτήματα VNN.

Ορισμός - VNNQ: Δοθέντος ενός συνόλου δεδομένων S και ενός σημείου ερωτήματος Q, βρες ένα αντικείμενο $O \in S$, έτσι ώστε (1) το O είναι ορατό στο Q (2) $\forall O' \in S$, εάν το O' είναι ορατό στο Q, τότε $dist(O, Q) < dist(O', Q)$,

όπου dist είναι μια συνάρτηση που επιστρέφει την απόσταση ανάμεσα σε ένα σημείο ερωτήματος και σε ένα αντικείμενο. Το O ονομάζεται κοντινότερος ορατός γείτονας του Q.

Υπάρχουν διάφορες παραλλαγές του VNN ερωτήματος. Μια τέτοια είναι το ερώτημα των k κοντινότερων ορατών γειτόνων (VkNN), το οποίο δεν βρίσκει μόνο έναν γείτονα (τον κοντινότερο ορατό) αλλά ανακτά σταδιακά τους k κοντινότερους ορατούς γείτονες. Υπάρχει ακόμα και το ερώτημα του αντίστροφου ορατού κοντινότερου γείτονα (*Visible Reverse Nearest Neighbor - VRNN*) [5] το οποίο ανακτά τα αντικείμενα τα οποία έχουν το σημείο ερωτήματος ως τον κοντινότερο ορατό γείτονα. Άρθρα που εξετάζουν RNN ερωτήματα είναι τα [37]-[41].

Μια γενίκευση του VkNN ερωτήματος είναι το συνολικό VkNN (*Aggregate Visible k Nearest Neighbor - AVkNN*) [6] ερώτημα το οποίο βρίσκει k αντικείμενα με τις μικρότερες συνολικές αποστάσεις σε ένα σύνολο Q από σημεία ερωτήματος. Ως συνολική συνάρτηση χρησιμοποιείται είτε το άθροισμα, είτε το ελάχιστο, είτε το μέγιστο, δηλαδή μια από τις συναρτήσεις SUMMINVIDIST, MAXMINVIDIST, MINMINVIDIST. Ερωτήματα ANN συναντάμε στα άρθρα [32]-[35].

Continuous Visible Nearest Neighbor (CVNN): Σε αυτή την περίπτωση το ερώτημα δεν αφορά ένα σημείο αλλά μια γραμμή στο χώρο και επίσης λαμβάνονται υπόψη τα εμπόδια. Με άλλα λόγια δεν ενδιαφερόμαστε για τα κοντινότερα αντικείμενα που είναι ορατά σε ένα σημείο αλλά για αυτά που είναι ορατά σε ένα διάστημα. Τα ερωτήματα συνεχούς κοντινότερου ορατού γείτονα επιστρέφουν ένα σύνολο από δυάδες (p, R) τέτοια ώστε το p είναι ο κοντινότερος γείτονας σε ένα υποδιάστημα R της γραμμής ερωτήματος. Εργασίες που απαντούν σε CVNN ερωτήματα βλέπουμε στα [2], [27], [31].

Αξιοσημείωτη ερευνητική προσπάθεια έχει γίνει σε *περιβάλλοντα με κινούμενα αντικείμενα* ή/και κινούμενες θέσεις ερωτήματος, [4], [16]-[26], [37].

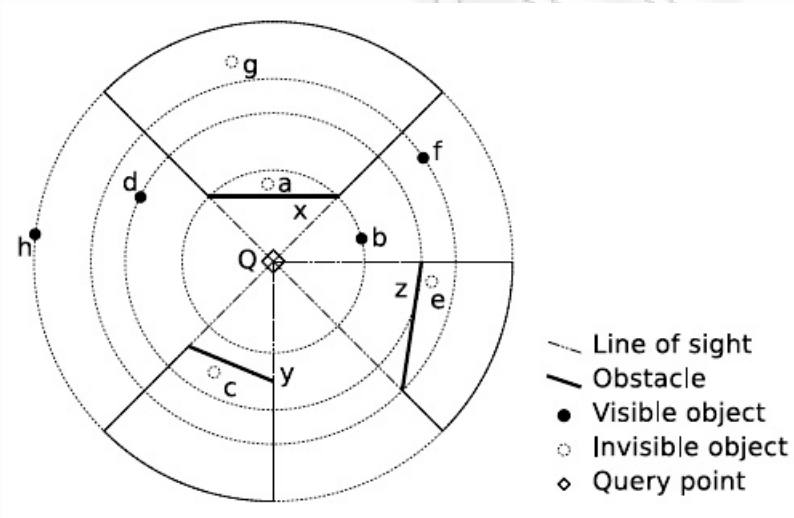
Μια άλλη κατηγόρια ερωτημάτων ονομάζεται "*All Nearest Neighbors*" (ANN) και αναφέρεται σε σύνολα πολυδιάστατων αντικειμένων. Δοθέντων δύο συνόλων A και B πολυδιάστατων αντικειμένων, το ερώτημα ANN ανακτά για κάθε αντικείμενο στο σύνολο A τον κοντινότερο γείτονά του στο σύνολο B. Ερωτήματα ANN συναντούμε στα [42], [43].

Το άρθρο [44] αναλύει το ερώτημα των k κοντινότερων ζευγαριών (*k closest pairs*). Το ερώτημα αυτό αναζητάει τα k κοντινότερα ζευγάρια ανάμεσα σε δύο χωρικά σύνολα δεδομένων.

Στη συνέχεια αναλύονται διεξοδικά έξι σύγχρονες πρακτικές που έχουν αναπτυχθεί για να αντιμετωπίσουν έξι διαφορετικούς τύπους ερωτημάτων οπτικού πεδίου. Πρόκειται για ερωτήματα κοντινότερου ορατού γείτονα (Visible Nearest Neighbor) των άρθρων [1] και [6], συνολικά κοντινότερου ορατού γείτονα (Aggregate Visible Nearest Neighbor) [6], συνεχούς κοντινότερου ορατού γείτονα (Continuous Visible Nearest Neighbor) [2], κοντινότερου περιβάλλοντος (Nearest Surrounder) [3], κοντινότερου περιβάλλοντος σε κινούμενα αντικείμενα [4] και αντίστροφου κοντινότερου ορατού γείτονα (Visible Reverse Nearest Neighbor) [5].

2.1 Ερωτήματα Ορατού Κοντινότερου Γείτονα

Στο άρθρο [1] αναλύεται το ερώτημα αναζήτησης των κοντινότερων ορατών γειτόνων (Visible k Nearest Neighbor - VkNN), το οποίο βρίσκει τα k κοντινότερα αντικείμενα που είναι ορατά σε ένα σημείο ερωτήματος. Η εικόνα 2.1 δείχνει ένα παράδειγμα του ερωτήματος VNN. Το σύνολο δεδομένων αποτελείται από αντικείμενα δεδομένων (μαύρες τελείες και κύκλοι) και εμπόδια (γραμμές). Το Q είναι το σημείο ερωτήματος. Από το Q κάποια αντικείμενα είναι ορατά (τα b, d, f, h που αναπαρίστανται από τις μαύρες τελείες), ενώ η θέαση κάποιων αντικειμένων μπλοκάρεται από εμπόδια, είναι δηλαδή μη ορατά (τα a, c, e, g που αναπαρίστανται από κύκλους). Το a, ανάμεσα σε όλα τα αντικείμενα, έχει τη μικρότερη απόσταση από το Q, άρα κατά την παραδοσιακή έννοια είναι ο κοντινότερος γείτονας. Όμως το a δεν είναι ο κοντινότερος ορατός γείτονας του Q. Ανάμεσα σε όλα τα ορατά αντικείμενα το b είναι το κοντινότερο στο Q, άρα είναι ο κοντινότερος ορατός γείτονας του Q.



Εικόνα 2.1. Το ερώτημα VNN.

Στο άρθρο [1] προτείνεται ένας αποδοτικός αλγόριθμος για την επεξεργασία αυτών των ερωτημάτων. Οι ορατοί γείτονες υπολογίζονται σταδιακά καθώς μεγαλώνει ο χώρος αναζήτησης. Το προτέρημα αυτού του αλγορίθμου είναι ότι μειώνει σημαντικά το υπολογιστικό κόστος διότι δεν απαιτεί τον υπολογισμό της ορατότητας όλων των αντικειμένων εκ των προτέρων όπως κάνουν οι υπάρχουσες μέθοδοι.

Ορισμός ερωτημάτων κοντινότερων ορατών γειτόνων (VNN): Δοθέντος ενός συνόλου δεδομένων S και ενός σημείου ερωτήματος Q, βρίσκουμε ένα αντικείμενο $O \in S$ έτσι ώστε: (1) το O είναι ορατό στο Q και (2) $\forall O' \in S$, αν

το O' είναι ορατό στο Q τότε $dist(O,Q) \leq dist(O',Q)$, όπου dist() είναι μια συνάρτηση που επιστρέφει την απόσταση ανάμεσα στο σημείο ερωτήματος και σε ένα αντικείμενο. Το O ονομάζεται ο κοντινότερος ορατός γείτονας του Q.

Ορισμός ερωτημάτων k κοντινότερων ορατών γειτόνων (VkNN): Δοθέντος ενός συνόλου δεδομένων S και ενός σημείου ερωτήματος Q, βρίσκουμε ένα σύνολο αντικειμένων A, έτσι ώστε: (1) το A να περιέχει k αντικείμενα από το S και (2) $\forall O \in A$, το O είναι ορατό στο Q και (3) $\forall O' \in S - A$, αν το O' είναι ορατό στο Q, τότε $dist(O,Q) \leq dist(O',Q)$, όπου dist() είναι μια συνάρτηση που επιστρέφει την απόσταση ανάμεσα στο σημείο ερωτήματος και ένα αντικείμενο. Το A ονομάζεται σύνολο k κοντινότερων ορατών γειτόνων του Q.

Ο αλγόριθμος σε αυτό το άρθρο [1] βασίζεται στην παρατήρηση ότι ένα απομακρυσμένο αντικείμενο δεν μπορεί να επηρεάσει την ορατότητα ενός κοντινότερου αντικειμένου. Επομένως μπορεί ο αλγόριθμος να ξεκινήσει από την ανάκτηση των κοντινότερων αντικειμένων και σταδιακά να αποκτήσει τη γνώση της ορατότητας, καθώς βρίσκει τους ορατούς γείτονες. Έτσι το κόστος για τη διερεύνηση της ορατότητας των αντικειμένων ελαχιστοποιείται.

Για λόγους απλοποίησης τα αντικείμενα που επιστρέφονται σαν αποτέλεσμα και τα εμπόδια που δημιουργούν αόρατες περιοχές ονομάζονται και τα δύο αντικείμενα. Τα αντικείμενα αναπαριστάνονται σαν πολύγωνα και επομένως μπορεί να είναι μερικώς ορατά. Εισάγεται έτσι μια νέα συνάρτηση απόστασης MinViDist που επιστρέφει την απόσταση ανάμεσα στο σημείο ερωτήματος και στο κοντινότερο ορατό σημείο ενός αντικειμένου με βάση μια δοθείσα συνθήκη ορατότητας. Αυτή η συνάρτηση είναι διαφορετική από την κοινώς χρησιμοποιούμενη συνάρτηση MinDist. Η ελάχιστη απόσταση MinDist ανάμεσα σε ένα πολύγωνο και σε ένα σημείο είναι η απόσταση ανάμεσα στο σημείο αυτό και στο κοντινότερο σημείο του πολυγώνου.

Ορισμός MinViDist: Δοθέντος ενός πολυγώνου P και ενός σημείου ερωτήματος Q, MinViDist είναι η απόσταση ανάμεσα στο Q και σε ένα σημείο $T \in P$, έτσι ώστε (1) το T είναι ορατό στο Q και (2) $\forall T' \in P$, αν το T είναι ορατό στο Q, τότε $dist(T,Q) \leq dist(T',Q)$ όπου dist() είναι μια συνάρτηση που επιστρέφει την απόσταση ανάμεσα σε δύο σημεία.

Το σύνολο των αντικειμένων καθώς και η σειρά με την οποία τα επιστρέφει η MinDist() είναι πιθανόν να διαφέρει από τις απαντήσεις που αναζητούνται με το ερώτημα VkNN. Συνεπώς η συνάρτηση MinDist() δεν είναι κατάλληλη για έναν VkNN αλγόριθμο σε αντίθεση με τη MinViDist().

Στον αλγόριθμο που προτείνεται στο άρθρο [1] θεωρείται ότι όλα τα αντικείμενα είναι ταξινομημένα σε χωρικό ευρετήριο R-tree (θα μπορούσαν όμως να χρησιμοποιηθούν και άλλες δομές ευρετηρίου). Προτείνονται τρεις παραλλαγές του αλγορίθμου οι οποίες διαφέρουν στο εάν το κλάδεμα των αντικειμένων με βάση την ορατότητα, γίνεται πριν ή μετά την ανάκτηση ενός φύλλου του R-tree. Διαφέρουν επίσης στη συνάρτηση απόστασης που χρησιμοποιείται για την ταξινόμηση των αντικειμένων στην ουρά προτεραιότητας η οποία διατηρεί τα υποψήφια αντικείμενα. Τα αποτελέσματα ταξινομούνται σύμφωνα με τη συ-

νάρτηση MinViDist() και για τις τρεις παραλλαγές του αλγορίθμου. Οι τρεις αυτές παραλλαγές περιγράφονται στη συνέχεια.

PostPruning: Για την αναζήτηση k κοντινότερων γειτόνων χρησιμοποιείται ένας υπάρχον αλγόριθμος. Η συνάρτηση MinDist() χρησιμοποιείται για να ταξινομήσει τους υποψήφιους στην ουρά προτεραιότητας. Επιπλέον σε αυτόν τον αλγόριθμο τα αντικείμενα ελέγχονται ως προς τη συνθήκη ορατότητας αφού ανακτηθούν από την ουρά και έτσι αγνοούνται τα μη ορατά αντικείμενα. Τα αποτελέσματα ταξινομούνται τελικά με τη συνάρτηση MinViDist().

PrePruning - MinDist: Όπως και στον αλγόριθμο PostPruning χρησιμοποιείται και εδώ η συνάρτηση MinDist() για την ταξινόμηση των υποψηφίων στην ουρά προτεραιότητας αλλά εδώ τα αντικείμενα (είτε κόμβοι είτε αντικείμενα αποτελέσματος) ελέγχονται ως προς την ορατότητα πριν ανακτηθούν. Έτσι αποφεύγεται ο άσκοπος έλεγχος αόρατων περιοχών οι οποίες δεν περιέχουν αποτελέσματα. Τελικά και εδώ τα αποτελέσματα ταξινομούνται με βάση τη συνάρτηση MinViDist().

PrePruning - MinViDist: Αυτός ο αλγόριθμος διαφέρει από τον PrePruning - MinDist επειδή χρησιμοποιεί τη συνάρτηση MinViDist() για να ταξινομήσει τα υποψήφια αντικείμενα στην ουρά προτεραιότητας. Η MinViDist είναι πιο ακριβής στον υπολογισμό αλλά αυξάνει το υπολογιστικό κόστος.

Συμπερασματικά η συνάρτηση MinViDist() χρησιμοποιείται για την ταξινόμηση των τελικών αποτελεσμάτων και επίσης στον αλγόριθμο PrePruning - MinViDist για να ταξινομήσει τα προ - διερεύνηση αντικείμενα. Και οι τρεις παραλλαγές αλγορίθμου αποκτούν τη γνώση της ορατότητας σταδιακά καθώς ανακτούνται τα κοντινότερα ορατά αντικείμενα και έτσι το υπολογιστικό κόστος για την απόφαση των ορατών περιοχών μειώνεται. Από τα πειραματικά αποτελέσματα του άρθρου [1] διαπιστώνεται ότι οι τελευταίες δύο παραλλαγές (PrePruning - MinDist και PrePruning - MinViDist) έχουν περισσότερο κόστος στο κλάδεμα με βάση την ορατότητα προκειμένου να μειώσουν το κόστος ανάκτησης δεδομένων. Αυτό θα μπορούσε να είναι χρήσιμο σε εφαρμογές που βασίζονται στην αποθήκευση δεδομένων σε δίσκο ή σε δίκτυο, όπου το κόστος ανάκτησης δεδομένων είναι πιο κρίσιμο από το κόστος επεξεργασίας. Αυτές οι δύο παραλλαγές είναι πιο αποδοτικές από την πρώτη (PostPruning). Στα πειράματα φαίνεται ότι ο χρόνος απόκρισης στην επεξεργασία του ερωτήματος VkNN βελτιώνεται κατά 35%.

2.2 Σταδιακή Αξιολόγηση Ερωτημάτων Ορατών Κοντινότερων Γειτόνων

Το άρθρο [6] αποτελεί μια νέα επεκταμένη έκδοση του προηγούμενου αρθρου VkNN [1]. Στο άρθρο [1] είχαν προτείνει το VkNN ερώτημα και δύο προσεγγίσεις για την επεξεργασία του. Τους αλγορίθμους PostPruning και PrePruning. Σε αυτό το νέο άρθρο [6] προτείνουν ένα νέο PrePruning αλγόριθμο ο οποίος είναι βέλτιστος σε όρους κόστους εισόδου-εξόδου. Επιπλέον γενικεύουν το VkNN ερώτημα σε μια έκδοση με πολλαπλά σημεία ερωτήματος, το συνολικό

VkNN (Aggregate VkNN - AVkNN) ερώτημα. Το AVkNN ερώτημα βρίσκει k αντικείμενα με τις συνολικά μικρότερες ορατές αποστάσεις σε ένα δοθέν σύνολο από σημεία ερωτήματος και όχι μόνο σε ένα σημείο ερωτήματος.

2.2.1 Ερωτήματα VkNN

Στο άρθρο [6] περιγράφονται κάποιες τεχνικές που φωτίζουν περισσότερο την επεξεργασία του VkNN ερωτήματος. Ανάμεσα στις τεχνικές αυτές αναφέρεται ότι ο αλγόριθμος VkNN που επεξεργάζονται βασίζεται στον αλγόριθμο Best-First σε ότι αφορά τη διάσχιση του δένδρου - ευρετηρίου. Ο αλγόριθμος Best-First έχει τρία πλεονεκτήματα: (i) η τιμή του k δεν χρειάζεται να καθοριστεί εκ των προτέρων, (ii) τα αποτελέσματα εξ' ορισμού ταξινομούνται σύμφωνα με τις αποστάσεις τους, (iii) το πλήθος των κόμβων που επισκέπτεται είναι ελάχιστο και γι' αυτό ο αλγόριθμος αυτός θεωρείται βέλτιστος.

Στο άρθρο [6] στο περιβάλλον στο οποίο επιλύεται το VkNN πρόβλημα (i) τα αντικείμενα αναπαρίστανται με πολύγωνα και (ii) κάθε αντικείμενο είναι επίσης και ένα εμπόδιο. Δηλαδή κάθε αντικείμενο αποτελεί εμπόδιο ορατότητας των πίσω από αυτό αντικειμένων. Επίσης όπως τονίστηκε και στο προηγούμενο άρθρο τους [1] χρησιμοποιείται η ελάχιστη ορατή απόσταση (MinViDist) για την ταξινόμηση των VkNN αποτελεσμάτων που σημαίνει ότι τα μη ορατά αντικείμενα, που έχουν άπειρη απόσταση, αγνοούνται. Διευκρινίζεται ότι για τον υπολογισμό της ελάχιστης ορατής απόστασης ανάμεσα σε ένα αντικείμενο X και ένα σημείο ερωτήματος q δεν απαιτείται ολόκληρο το σύνολο των αντικειμένων S. Το ακόλουθο λήμμα χρησιμοποιείται για να αποφασιστεί το υποσύνολο B του S έτσι ώστε η συνάρτηση MinViDist(q, X, S) να έχει το ίδιο αποτέλεσμα με τη συνάρτηση MinViDist(q, X, B).

Λήμμα: Αν η MinViDist(q, Z, S) είναι μεγαλύτερη από τη MinViDist(q, X, S), τότε η MinViDist(q, X, S) είναι ίση με τη MinViDist(q, X, S-Z). Το λήμμα αυτό υπονοεί ότι μόνο αντικείμενα με MinViDist μεγαλύτερη από αυτή του X μπορούν να αγνοηθούν με ασφάλεια ως εμπόδια, όταν υπολογίζεται η ελάχιστη ορατή απόσταση ανάμεσα στο q και στο X. Αυτό το λήμμα επιτρέπει τη σταδιακή ανάκτηση των κοντινότερων ορατών γειτόνων και την κατασκευή της ορατής περιοχής ταυτόχρονα. Συνεπώς η απαιτούμενη «ποσότητα» ορατότητας ελαχιστοποιείται. Για την εύρεση των αντικειμένων με ελάχιστη ορατή απόσταση μεγαλύτερη από αυτή του προς εξέταση αντικειμένου χρησιμοποιείται ένας αισιόδοξος εκτιμητής, η ελάχιστη απόσταση. Δηλαδή αν η MinDist του X είναι μεγαλύτερη από c, τότε η MinViDist του X πρέπει να είναι μεγαλύτερη από c επίσης.

Στο άρθρο [6] περιγράφονται οι αλγόριθμοι PostPruning και PrePruning οι οποίοι διαφέρουν στη συνάρτηση απόστασης που χρησιμοποιείται για να ταξινομήσει τις εισόδους στην ουρά προτεραιότητας αλλά παράγουν τα ίδια αποτελέσματα. Ο αλγόριθμος PrePruning αποτελεί μια βελτιστοποίηση του αλγορίθμου PostPruning σε όρους κόστους εισόδου - εξόδου.

2.2.2 Γενίκευση VkNN Ερώτηματος

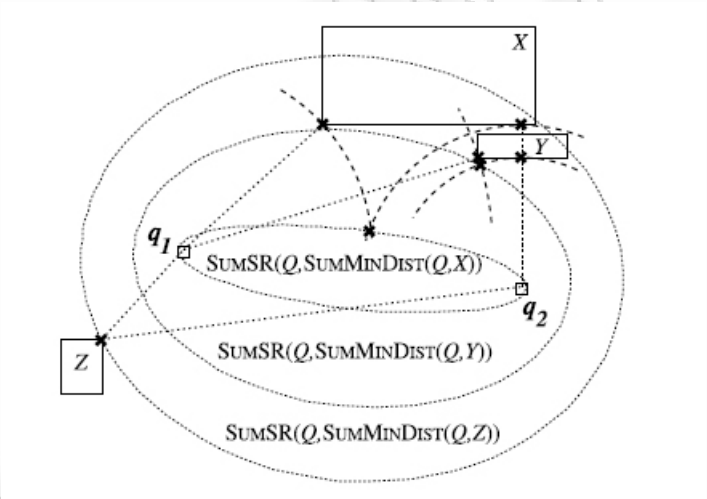
Ένα συνολικό kNN (Aggregate k Nearest Neighbor - AkNN) ερώτημα βρίσκει k αντικείμενα με τις συνολικά μικρότερες αποστάσεις σε ένα σύνολο Q από σημεία ερωτήματος. Ο επίσημος ορισμός είναι ο ακόλουθος.

Ορισμός συνολικού kNN ερωτήματος: Δοθέντος ενός συνόλου Q από σημεία ερωτήματος και ενός συνόλου S από αντικείμενα, οι συνολικοί kNN του Q είναι ένα σύνολο A από αντικείμενα τέτοια ώστε (i) το A περιέχει k αντικείμενα από το S (ii) για κάθε δοθέν X που είναι στο A και Y στο (S-A), η συνολική MINDIST ανάμεσα στο Q και το X, AGGMINDIST(Q, X), είναι ίση ή μικρότερη από την AGGMINDIST(Q, Y).

Η συνάρτηση AGGMINDIST ορίζεται ως ακολούθως.

Ορισμός συνολικής ελάχιστης απόστασης: Δοθέντος ενός συνόλου Q από σημεία ερωτήματος και μιας επιλογής για τη συνολική συνάρτηση, η AGGMINDIST(Q, X) επιστρέφει είτε την ελάχιστη (MINMINDIST(Q, X)), είτε τη μέγιστη (MAXMINDIST(Q, X)) ή το άθροισμα (SUMMINDIST(Q, X)) της MINDIST(Q, X) για όλα τα σημεία ερωτήματος στο σύνολο Q.

Ένα παράδειγμα του ερωτήματος AkNN δίνεται στην εικόνα 2.2.



Εικόνα 2.2. Παράδειγμα συνολικού ερωτήματος με σύνολο ερωτημάτων $Q = q_1, q_2$ και αντικείμενα X, Y και Z. Οι ελλείψεις δείχνουν τα όρια της περιοχής αναζήτησης.

Σύμφωνα με τη συνολική απόσταση του αθροίσματος (SUMMINDIST), οι συνολικοί 3 NNs των q1 και q2 είναι X, Y και Z, σύμφωνα με τη διάταξη που δίνει η απόσταση SUMMINDIST.

Περιοχή Αναζήτησης

Χρησιμοποιώντας την ιδέα της συνολικής περιοχής αναζήτησης είναι δυνατόν να εφαρμοστεί μια στρατηγική σταδιακής ανάκτησης στο AVkNN ερώτημα. Για κάθε κοντινότερο γείτονα που ανακτάται από την ουρά προτεραιότητας υπάρχει μια αντίστοιχη περιοχή αναζήτησης (Search Region - SR) η οποία περιορίζει την αναζήτηση.

Ορισμός συνολικής περιοχής αναζήτησης: Δοθέντος ενός συνόλου Q από σημεία ερωτήματος, περιοχή αναζήτησης AGGSR(Q, c) είναι ένα σύνολο από σημεία p τέτοια ώστε η AGGMINDIST(Q, p) είναι ίση ή μικρότερη από το c, δηλαδή, $AGGSR(Q, c) = p : AGGMINDIST(Q, p) \leq c$.

Αφού θεωρούμε τρεις συνολικές συναρτήσεις SUM, MAX, και MIN υπάρχουν τρεις τύποι συνολικής περιοχής αναζήτησης, SUMSR, MAXSR και MINSR αντίστοιχα.

Λήμμα: Η περιοχή αναζήτησης SUMSR ενός ερωτήματος SUM-AkNN είναι κυρτή.

Το ίδιο ισχύει και για τη συνάρτηση MAX. Χρησιμοποιώντας την κυρτότητα των SUMSRs και MAXSRs περιοχών αναζήτησης, μπορεί να αποφασιστεί αν υπάρχουν αρκετά εμπόδια για να υπολογιστεί η συνολική MINVIDIST (AGGMINVIDIST) ενός αντικειμένου. Συνεπώς η ανάκτηση των αντικειμένων και η κατασκευή της ορατής περιοχής μπορεί να γίνει με τρόπο σταδιακό. Η MINSR δεν έχει την ιδιότητα της κυρτότητας.

Ερώτημα Συνολικού Ορατού Κοντινότερου Γείτονα

Το ερώτημα συνολικών k κοντινότερων γειτόνων είναι μια γενίκευση του ερωτήματος VkNN για πολλά σημεία ερωτήματος.

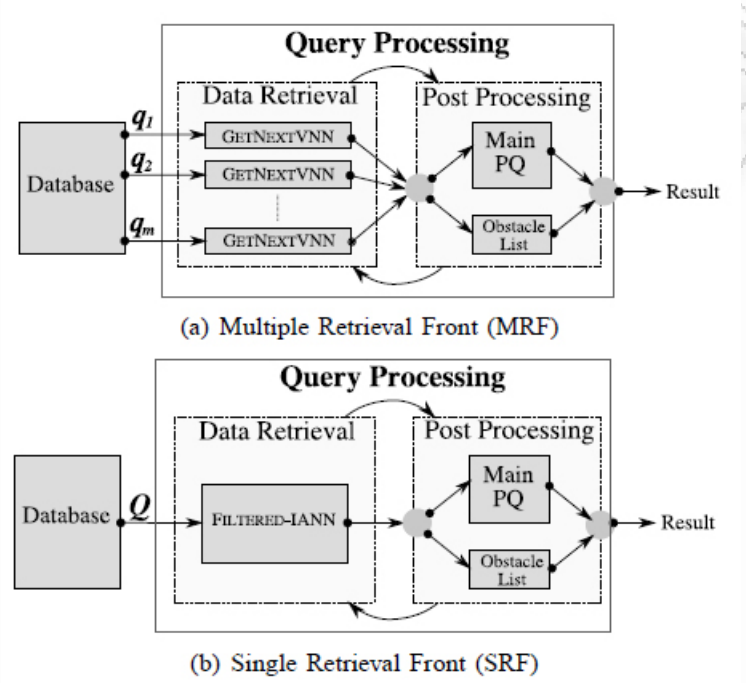
Ορισμός συνολικού VkNN ερωτήματος: Δοθέντος ενός συνόλου S από αντικείμενα και ενός συνόλου Q από σημεία ερωτήματος, οι συνολικοί ορατοί kNNs του Q είναι ένα σύνολο A από αντικείμενα τέτοια ώστε (i) το A περιέχει k αντικείμενα από το S, τα οποία είναι ορατά στο Q (ii) για κάθε X στο A και Y στο (S-A), η AGGMINVIDIST(Q, X, S) είναι ίση ή μικρότερη από την AGGMINVIDIST(Q, Y, S).

Ορισμός συνολικής MINVIDIST: Δοθέντος ενός συνόλου Q από σημεία ερωτήματος, η συνάρτηση απόστασης AGGMINVIDIST(Q, X, S) είναι η συνολική απόσταση των MINVIDIST(q, X, S) για όλα τα q στο Q.

Ιδιότητες: (i) Για τις συναρτήσεις SUMMINVIDIST και MAXMINVIDIST, το X είναι αόρατο στο Q, αν και μόνο αν υπάρχει ένα σημείο ερωτήματος q στο Q τέτοιο ώστε η MINVIDIST(q, X, S) είναι άπειρη. (ii) Για τη συνάρτηση MINMINVIDIST, το X είναι αόρατο στο Q αν και μόνο αν η MINVIDIST(q, X, S) είναι άπειρη για όλα τα σημεία ερωτήματος q στο Q.

Το πρόβλημα του ερωτήματος AVkNN δεν μπορεί να επιλυθεί με τους συμβατικούς συνολικούς kNN (AkNN) αλγορίθμους, καθώς κάθε σημείο ερωτήματος έχει ένα διαφορετικό σύνολο από ορατά αντικείμενα και κάθε ορατό

αντικείμενο μπορεί να έχει ένα διαφορετικό ορατό μέρος για κάθε σημείο ερωτήματος. Έτσι προτείνονται δύο σταδιακές προσεγγίσεις για την επεξεργασία των AVkNN ερωτημάτων, η πολλαπλή ανάκτηση μετώπου (Multiple Retrieval Front - MRF) και η ενιαία ανάκτηση μετώπου (Single Retrieval Front - SRF) για τις τρεις συνολικές συναρτήσεις (ελάχιστη, μέγιστη και άθροισμα). Η ανάκτηση μετώπου είναι ένα υποερώτημα που χρησιμοποιείται για να προσπελαύνει τη βάση δεδομένων. Η εικόνα 2.3 δείχνει πως οι δύο προσεγγίσεις διαφέρουν στον τρόπο πρόσβασης στη βάση δεδομένων.



Εικόνα 2.3. Δομική σύγκριση ανάμεσα στους αλγορίθμους MRF και SRF.

Ο MRF αλγόριθμος εκτελεί πολλαπλά στιγμιότυπα του αλγορίθμου GETNEXTVNN, ο οποίος ανακτά σταδιακά τους VNNs βασισμένος στον αλγόριθμο PrePruning για κάθε σημείο ερωτήματος. Τα αποτελέσματα από τα διαφορετικά σημεία ερωτήματος συνδυάζονται σε μια ουρά προτεραιότητας. Από την άλλη πλευρά η προσέγγιση SRF προσπελαύνει τη βάση δεδομένων μέσω ενός ενιαίου ερωτήματος που κάνει φιλτράρισμα των αντικειμένων ανάλογα με την ορατότητά τους, υιοθετώντας μια στρατηγική όμοια με αυτή του Best-First kNN αλγορίθμου.

Οι δύο προσεγγίσεις MRF και SRF έχουν μια διαδικασία μεταεπεξεργασίας. Για την προσέγγιση MRF, το κομμάτι μεταεπεξεργασίας χρησιμοποιείται για την αναδιάταξη των αντικειμένων που ανακτήθηκαν για κάθε σημείο ερωτή-

ματος σύμφωνα με την απόσταση AGGMINVIDIST στο Q. Για τον αλγόριθμο SRF η μεταεπεξεργασία χρησιμοποιείται για την αναδιάταξη των αντικειμένων που ανακτήθηκαν από τον αλγόριθμο φιλτραρίσματος, σύμφωνα με την απόσταση AGGMINVIDIST.

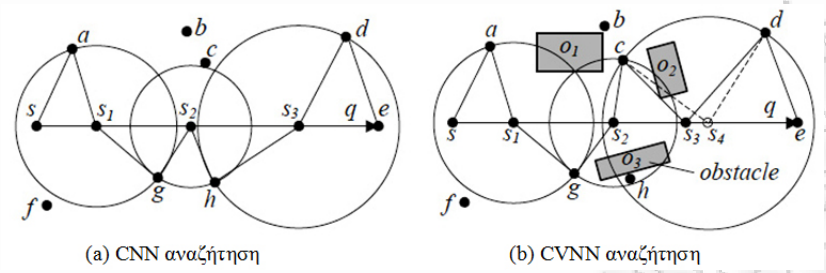
Όλα τα ανακτηθέντα αντικείμενα και από τις δύο προσεγγίσεις διατηρούνται ως εμπόδια για τον υπολογισμό της AGGMINVIDIST των αντικειμένων στην ουρά προτεραιότητας (MainPQ). Η ουρά MainPQ χρησιμοποιεί την AGGMINDIST ως ένα αισιόδοξο εκτιμητή και την AGGMINVIDIST σαν το πραγματικό μέτρο ταξινόμησης. Επομένως τα αντικείμενα που ανακτούνται από την κορυφή της MainPQ είναι σε αύξουσα σειρά σύμφωνα με την απόσταση AGGMINVIDIST. Ως αποτέλεσμα και οι δύο προσεγγίσεις χρησιμοποιούνται για τη σταδιακή ανάκτηση συνολικών VkNNs από τη βάση δεδομένων.

2.3 Ερωτήματα Συνεχούς Ορατού Κοντινότερου Γείτονα

Η τεχνική που αναπτύσσεται στο άρθρο [2] απαντάει σε ερωτήματα συνεχούς, κοντινότερου, ορατού γείτονα (Continuous Visible Nearest Neighbor - CVNN). Σε αυτή την περίπτωση το ερώτημα δεν αφορά ένα σημείο αλλά μια γραμμή στο χώρο και επίσης λαμβάνονται υπόψη τα εμπόδια. Με άλλα λόγια δεν ενδιαφερόμαστε για τα κοντινότερα αντικείμενα που είναι ορατά σε ένα σημείο αλλά για αυτά που είναι ορατά σε ένα διάστημα. Τα ερωτήματα συνεχούς κοντινότερου ορατού γείτονα επιστρέφουν ένα σύνολο από δυάδες (p, R) τέτοια ώστε το p είναι ο κοντινότερος γείτονας σε ένα υποδιάστημα R της γραμμής ερωτήματος.

Ένα παράδειγμα CNN ερωτήματος φαίνεται στην εικόνα 2.4a με σύνολο δεδομένων P=a, b, c, d, f, g, h και q=[s, e] το ευθύγραμμο τμήμα ερωτήματος. Το σύνολο των αποτελεσμάτων είναι $CNN = \{\langle \alpha, [s, s1]\rangle, \langle g, [s1, s2]\rangle, \langle h, [s2, s3]\rangle, \langle d, [s3, e]\rangle\}$, το οποίο δείχνει ότι το σημείο a είναι ο κοντινότερος γείτονας για κάθε σημείο στο ευθύγραμμο τμήμα [s, s1] και ούτω καθεξής. Τα σημεία s1, s2, s3 ονομάζονται σημεία διαίρεσης καθώς ο κοντινότερος γείτονας αλλάζει σε αυτά τα σημεία. Στον πραγματικό κόσμο όμως με τα εμπόδια χρειάζεται να απαντηθούν ερωτήματα CVNN. Ένα παράδειγμα τέτοιου ερωτήματος φαίνεται στην εικόνα 2.4b. Το ερώτημα κοντινότερου ορατού γείτονα τίθεται στο σημείο s4 και η απάντηση είναι το σημείο d. Παρόλο που το h είναι κοντινότερο στο s4 από το d, το τελευταίο μπλοκάρεται από το εμπόδιο o3 και έτσι εξαιρείται από το σύνολο των αποτελεσμάτων.

Ο αλγόριθμος σε αυτό το άρθρο [2] σκοπεύει να δώσει αποτελέσματα διατρέχοντας μια και μοναδική φορά το σύνολο των αντικειμένων. Αρχικά το σύνολο των αποτελεσμάτων είναι κενό για το διάστημα του ερωτήματος και κατά την αναζήτηση ανανεώνεται. Σε κάθε βήμα το σύνολο των απαντήσεων έχει το τρέχον αποτέλεσμα με βάση όλα τα αντικείμενα που έχουν εξεταστεί μέχρι στιγμής. Το αποτέλεσμα τελικά περιέχει δυάδες που αποτελούνται από το αντικείμενο και το διάστημα κυριαρχίας του επάνω στη γραμμή ερωτήματος. Το σύνολο των αντικειμένων και το σύνολο των εμποδίων δομούνται από δύο ξε-



Εικόνα 2.4. Παράδειγμα ερωτημάτων CNN και CVNN.

χωριστά R-trees. Οι μέθοδοι κλαδέματος αντικειμένων και εμποδίων αυξάνουν την ταχύτητα των αλγορίθμων.

Ο αλγόριθμος αυτός [2] διατρέχει τα δένδρα - ευρετήρια αντικειμένων και εμποδίων το πολύ μια φορά. Ο αλγόριθμος ανακτά ακριβώς τους κοντινότερους ορατούς γείτονες κάθε σημείου σε μια γραμμή ερωτήματος, δηλαδή δεν χάνει σημεία και δεν κάνει λάθη. Αυτό συμβαίνει διότι ο αλγόριθμος εξετάζει μόνο εκείνα τα αντικείμενα που μπορούν να συμμετάσχουν στο σύνολο των απαντήσεων και αγνοεί όλα τα αντικείμενα που δεν είναι ικανά να συμμετέχουν στο σύνολο των λύσεων.

Προτού αναλυθούν οι μέθοδοι κλαδέματος του συνόλου των αντικειμένων και των εμποδίων ορίζονται οι έννοιες της «κυριαρχίας» και του «κύκλου γειτνίασης».

Κυριαρχία: Δοθέντος ενός σημείου p, ενός τμήματος R (=[R.l, R.r]) και ενός συνόλου δεδομένων P, ως κυριαρχία του p πάνω στο R ορίζεται όταν ισχύουν δύο συνθήκες: $p' \in P - \{p\}$, (i) $dist(p', R.l) > dist(p, R.l)$ και (ii) $dist(p', R.r) > dist(p, R.r)$.

Κύκλος γειτνίασης: Με κέντρο ένα από τα άκρα ενός τμήματος R και ακτίνα ένα σημείο από το σύνολο των αντικειμένων ορίζεται ο κύκλος γειτνίασης.

Για να επιτευχθεί από τον αλγόριθμο, η απόδοση αποτελεσμάτων, διατρέχοντας το σύνολο των αντικειμένων μια και μοναδική φορά χρησιμοποιούνται τα ακόλουθα λήμματα.

Λήμμα: Υποθέτουμε ότι το p κυριαρχεί στο τμήμα R=[s, e]. Ένα νέο σημείο p' παραβιάζει την κυριαρχία του p πάνω στο R ανν το p' είναι στον κύκλο γειτνίασης του s (VC(s)) ή/ και του e (VC(e)).

Λήμμα: Δοθέντος ενός τμήματος R=[s, e] και ενός νέου σημείου δεδομένων p, το p δεν θα είναι VNN σε κανένα σημείο του R, αν το p είναι αόρατο σε κάθε σημείο του R.

Λήμμα: Ορίζουμε VR_p την ορατή περιοχή ενός σημείου δεδομένων p και DR_p την περιοχή κυριαρχίας του p. Το σημείο p πρέπει να είναι VNN για το τμήμα $R = VR_p \cap DR_p$.

Παρατηρούνται ακόμα δύο σημαντικές ιδιότητες που είναι μοναδικές για τη CVNN αναζήτηση.

Ιδιότητα ασυνέχειας: Ένα σημείο αποτελέσματος μπορεί να VNN σε πολλαπλά τμήματα τα οποία δεν είναι γειτονικά.

Ιδιότητα αόρατου τμήματος: Η λίστα των αποτελεσμάτων ενός ερωτήματος CVNN μπορεί να έχει $k(\geq 1)$ αόρατα τμήματα $\langle \emptyset, R \rangle$, όπου κανένα σημείο στο δοθέν σύνολο αντικειμένων P δεν είναι ορατό σε κανένα σημείο του R.

2.3.1 Κλάδεμα του Συνόλου Αντικειμένων

Οι ακόλουθες τεχνικές μειώνουν το χώρο αναζήτησης, κλαδεύοντας το σύνολο των αντικειμένων.

Μέθοδος 1: Ένας ενδιάμεσος (όχι φύλλο) κόμβος E του R-tree μπορεί να περιέχει απαντήσεις μόνο εάν η $mindist(E, q) < RLmaxD$. Όπου q η γραμμή ερωτήματος και $RLmaxD$ η μέγιστη απόσταση κάποιου αντικειμένου του αποτελέσματος, είτε από το δεξί είτε από το αριστερό άκρο του τμήματος που κυριαρχεί.

Μέθοδος 2: Η μέθοδος αυτή εφαρμόζεται μετά τη μέθοδο 1 επειδή προκαλεί επιπλέον υπολογιστικό κόστος. Ένας ενδιάμεσος κόμβος E μπορεί να περιέχει απαντήσεις μόνο όταν υπάρχει τουλάχιστον ένα διάστημα R στο σύνολο των αποτελεσμάτων τέτοιων ώστε το R να κυριαρχείται από το E.

Μέθοδος 3: Εφαρμόζεται μετά τις μεθόδους 2 και 3. Επιπλέον σε αυτά που προϋποθέτει η μέθοδος 2 το E πρέπει να είναι ορατό στο R.

Μέθοδος 4: Αφορά στη σειρά επισκεψιμότητας των κόμβων έτσι ώστε να βελτιστοποιηθεί το κλάδεμα. Οι κόμβοι ταξινομούνται σε αύξουσα σειρά με βάση την ελάχιστη απόσταση από τη γραμμή ερωτήματος.

2.3.2 Κλάδεμα του Συνόλου Εμποδίων

Με τις ακόλουθες τεχνικές αποκόπτονται τα εμπόδια που δεν μπορούν να εμποδίσουν την ορατότητα ενός αντικειμένου. Έτσι δεν εξετάζονται και μειώνεται το υπολογιστικό κόστος της αναζήτησης CVNN.

Μέθοδος 5: Ο χώρος χωρίζεται από τη γραμμή ερωτήματος σε δύο μέρη. Αυτόν που είναι πάνω από τη γραμμή ερωτήματος και αυτόν που είναι κάτω. Με βάση αυτόν τον διαχωρισμό ένα εμπόδιο επηρεάζει την ορατότητα του προς εξέταση αντικειμένου ως προς τη γραμμή ερωτήματος όταν ένα τουλάχιστον σημείο του εμποδίου βρίσκεται στην ίδια πλευρά της γραμμής ερωτήματος με το αντικείμενο.

Μέθοδος 6: Τα εμπόδια επηρεάζουν σίγουρα την ορατότητα ενός αντικειμένου στη γραμμή ερωτήματος αν και μόνο αν αλληλεπιδρούν ή πέφτουν μέσα στο τρίγωνο που σχηματίζεται από το αντικείμενο και τη γραμμή ερωτήματος.

Μέθοδος 7: Το εμπόδιο o επηρεάζει την ορατότητα ενός σημείου p στη γραμμή ερωτήματος q μόνο αν $min(o, p) < min(p, q)$.

2.3.3 Επιμέρους Συναρτήσεις Αλγορίθμου CVNN

Στη συνέχεια παρουσιάζονται οι επιμέρους συναρτήσεις από τις οποίες αποτελείται ο αλγόριθμος CVNN.

Συνάρτηση GetObstacle: Προκειμένου να βρούμε την ορατή περιοχή ενός σημείου δεδομένων p πάνω σε μια γραμμή ερωτήματος q παρουσία εμποδίων, πρέπει να βρούμε όλα τα εμπόδια που μπορούν να επηρεάσουν την ορατότητα του p πάνω στο q. Η συνάρτηση GetObstacle δίνει τη λύση. Εξετάζει τα εμπόδια σε αύξουσα σειρά με βάση την απόστασή τους από το q και τερματίζει όταν το εμπόδιο έχει απόσταση από το q μεγαλύτερη από $mindist(p, q)$. Επιστρέφεται μια συνδεδεμένη λίστα με τα εμπόδια που επηρεάζουν την ορατότητα του p.

Συνάρτηση Visible Region Computation (VRC): Με τη βοήθεια αυτής της συνάρτησης υπολογίζονται οι αόρατες περιοχές του αντικειμένου με βάση το κάθε εμπόδιο που το επηρεάζει και αφαιρούνται από τη γραμμή ερωτήματος.

Συνάρτηση Result List Update (RLU): Αυτή η συνάρτηση προτείνεται για να ανανεώνει τη λίστα των αποτελεσμάτων σε μια CVNN αναζήτηση, όταν εκτιμάται ένα νέο σημείο δεδομένων. Παίρνει σαν είσοδο την τρέχουσα λίστα αποτελεσμάτων, ένα νέο σημείο δεδομένων p και την ορατή περιοχή του p, έτσι ώστε να εκτιμήσει την επιρροή του p στη λίστα των αποτελεσμάτων. Ειδικότερα για κάθε περιοχή R στη λίστα των αποτελεσμάτων, η συνάρτηση διακρίνει δύο περιπτώσεις: (i) Εάν το p είναι ορατό στο R η συνάρτηση πρώτα υπολογίζει την επικάλυψη $R_{int}(= R \cap VR_p)$ και τη διαφορά $R_{dif}(= R - VR_p)$ ανάμεσα στο R και VR_p, όπου VR_p είναι η ορατή περιοχή του p. Στη συνέχεια αν δεν υπάρχει κοντινότερος ορατός γείτονας στην περιοχή R (εφεξής R.VNN), τότε τα $\langle p, R_{int}\rangle$ και $\langle \emptyset, R_{dif}\rangle$ (εφόσον $R_{dif} \neq \emptyset$) εισάγονται στο τρέχον σύνολο των αποτελεσμάτων. Αλλιώς, αν υπάρχει R.VNN, η συνάρτηση RLU εισάγει το $\langle R.VNN, R_{dif}\rangle$ στο τρέχον αποτέλεσμα (εφόσον $R_{dif} \neq \emptyset$). Τότε καλείται η συνάρτηση RS - CVNN για να αποφασίσει αν ο R.VNN μπορεί να αντικατασταθεί πλήρως ή μερικώς από το p για την περιοχή R_{int}. (ii) Αν το p είναι μη ορατό στο R, σημαίνει ότι το p δεν έχει καμιά επιρροή στη περιοχή R και έτσι η συνάρτηση RLU εισάγει το $\langle R.VNN, R\rangle$ στο τρέχον αποτέλεσμα.

Συνάρτηση Region Split for CVNN (RS - CVNN): Αυτή η συνάρτηση χρησιμοποιείται για να ελέγξει την εγκυρότητα του R.VNN σε σχέση με το p και να αντικαταστήσει το R.VNN πλήρως ή μερικώς με το p, εάν χρειάζεται. Όταν καλείται αυτή η συνάρτηση η περιοχή R είναι ορατή στο p και επομένως το μόνο που χρειάζεται να ελέγχει είναι η σχέση κυριαρχίας. Η συνάρτηση RS-CVNN διακρίνει τέσσερεις περιπτώσεις, (i) Αν το p δεν κυριαρχεί στο R η αρχική δυάδα $\langle R.VNN, R\rangle$ παραμένει έγκυρη και προστίθεται στην προσωρινή λίστα των αποτελεσμάτων, (ii) Αν το p κυριαρχεί σε ολόκληρο το R ο αλγόριθμος αντικαθιστά το R.VNN με το p και εισάγει τη δυάδα $\langle p, R\rangle$ στην προσωρινή λίστα αποτελεσμάτων, (iii) Αν το p είναι στον κύκλο γειτνίασης του δεξιού άκρου του R, τότε ο αλγόριθμος υπολογίζει το σημείο τομής s1 που σχηματίζεται από τη γραμμή ερωτήματος και την κάθετη διχοτόμο του ευθύγραμμου τμήματος [v, p] με το v να είναι R.VNN και προσθέτει τις δυάδες $\langle p, [R.l, s1]\rangle$ και $\langle v, [s1, R.r]\rangle$ στην προσωρινή λίστα αποτελεσμάτων, (iv) Αντίστοιχα με την περίπτωση (iii) εάν το p είναι πιο κοντά στον κύκλο γειτνίασης του αριστερού άκρου του R.

2.3.4 Αλγόριθμος CVNN

Υπάρχουν δύο σωροί, ο σωρός των αντικειμένων (Hp) και ο σωρός των εμποδίων (Ho) που διατηρούν τα δεδομένα σε αύξουσα σειρά της απόστασής τους από τη γραμμή ερωτήματος. Σε κάθε βήμα ο αλγόριθμος επισκέπτεται το αντικείμενο του Hp με τη μικρότερη απόσταση (την κορυφή e του σωρού) και κάνει τα εξής:

1. Ελέγχει αν $mindist(e, q) \geq RLmaxD$. Αν ισχύει ο αλγόριθμος τερματίζει διότι σύμφωνα με τη μέθοδο 1 τα υπόλοιπα αντικείμενα τα σωρού δεν περιέχουν απαντήσεις.
2. Διαφορετικά εξετάζεται αν το e είναι αντικείμενο καλείται η GetObs για να βρει όλα τα εμπόδια που μπορεί να επηρεάσουν την ορατότητα του. Κατόπιν καλείται η συνάρτηση Visible Region Computation για να υπολογιστεί η ορατή περιοχή του e στο q και κατόπιν καλείται η συνάρτηση Result List Update για την ανανέωση του συνόλου των αποτελεσμάτων.
3. Αν το e είναι ενδιάμεσος κόμβος και όχι αντικείμενο ο αλγόριθμος CVNN επισκέπτεται το υποδένδρο του μόνο αν τα δεδομένα του μπορούν να αποτελέσουν λύσεις. Αυτό ελέγχεται σύμφωνα με τις μεθόδους 2 και 3.

2.3.5 Ανάλυση Αλγορίθμου CVNN

Ο αλγόριθμος CVNN διατρέχει τα δένδρα Tp (αντικειμένων) και To (εμποδίων) το πολύ μια φορά. Η πολυπλοκότητα του αλγορίθμου είναι $O(Nlog \mid T_p \mid \times(log \mid T_o \mid + \mid L_o \mid + \mid RL \mid))$. Ο αλγόριθμος ανακτά ακριβώς τους κοντινότερους ορατούς γείτονες κάθε σημείου σε μια γραμμή ερωτήματος, δηλαδή δεν χάνει σημεία και δεν κάνει λάθη. Αυτό συμβαίνει διότι ο αλγόριθμος εξετάζει μόνο εκείνα τα αντικείμενα που μπορούν να συμμετάσχουν στο σύνολο των απαντήσεων και αγνοεί όλα τα αντικείμενα που δεν είναι ικανά να συμμετέχουν στο σύνολο των λύσεων σύμφωνα με τις μεθόδους 1 έως 4.

Σε αντίθεση με το να διατηρούμε δύο R-trees μπορούμε να έχουμε μόνο ένα με όλα τα αντικείμενα και τα εμπόδια και συνεπώς μόνο έναν σωρό με όλα τα δεδομένα σε αύξουσα σειρά σύμφωνα με την απόστασή τους από το q. Με αυτόν τον τρόπο όλα τα εμπόδια που επηρεάζουν ένα αντικείμενο έχουν προσπελαστεί πριν αυτό και δεν χρειάζεται να κληθεί η συνάρτηση GetObs.

Ο παραπάνω αλγόριθμος μπορεί να επεκταθεί για να υποστηρίξει και κάποιες παραλλαγές των CVNN ερωτημάτων όπως CVkNN και δ-CVNN. Ο CVkNN βρίσκει τους k κοντινότερους ορατούς γείτονες σε μια γραμμή ερωτήματος. Ο δ-CVNN βασίζεται στον περιορισμό που θέτει ο χρήστης ώστε οι απαντήσεις που θα επιστρέψει ο αλγόριθμος να βρίσκονται το πολύ σε απόσταση δ.

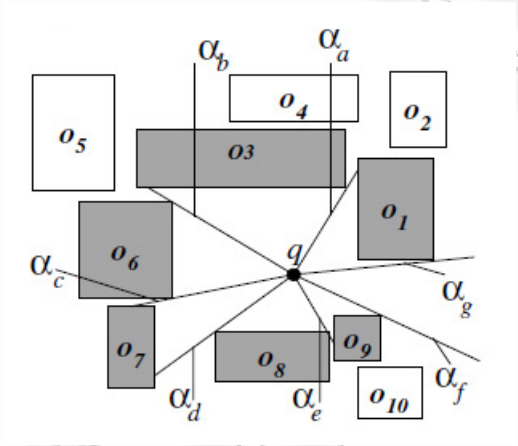
2.4 Ερωτήματα Κοντινότερου Περιβάλλοντος

Σε ορισμένες περιπτώσεις όπως στο πεδίο μάχης ο αλγόριθμος που επιστρέφει όλους τους περιβάλλοντες εχθρούς είναι πιο χρήσιμος από αυτόν που επιστρέφει

απλώς τους κοντινότερους εχθρούς. Έτσι δημιουργήθηκε η ανάγκη για ένα νέο ερώτημα, αυτό του κοντινότερου περιβάλλοντος αντικειμένου.

Τα ερωτήματα κοντινότερου περιβάλλοντος NS (Nearest Surrounder) σε αντίθεση με αυτά των κοντινότερων γειτόνων λαμβάνουν υπόψη τους και την κατεύθυνση. Γι' αυτό έχουν ιδιότητες που βασίζονται στη γωνία και την απόσταση.

Στην εικόνα 2.5 φαίνεται ένα NS σημείο ερωτήματος, q, το οποίο περιβάλλεται από δέκα αντικείμενα $o_1, o_2, ...o_10$. Το σύνολο των αποτελεσμάτων είναι $NS(q) = \{\langle o_1, [\alpha_g, \alpha_\alpha]\rangle, \langle o_3, [\alpha_\alpha, \alpha_b]\rangle, \langle o_6, [\alpha_b, \alpha_c]\rangle, \langle o_7, [\alpha_c, \alpha_d]\rangle, \langle o_8, [\alpha_d, \alpha_e]\rangle, \langle o_9, [\alpha_e, \alpha_f]\rangle\}$, όπου α_α έως α_g είναι διακριτές γωνίες. Τα αντικείμενα του συνόλου των αποτελεσμάτων είναι τα κοντινότερα περιβάλλοντα αντικείμενα του q στις αντίστοιχες γωνίες, καθώς κανένα άλλο αντικείμενο δεν παρεμβάλλεται ανάμεσα σε αυτά και το σημείο ερωτήματος σε αυτές τις γωνίες.



Εικόνα 2.5. Ερώτημα κοντινότερου περιβάλλοντος.

Το άρθρο [3] προτείνει δύο αλγορίθμους τον Sweep και τον Ripple. Ο Sweep αναζητά τα περιβάλλοντα αντικείμενα σύμφωνα με την κατεύθυνση τους ενώ ο Ripple σύμφωνα με την απόσταση τους από το σημείο ερωτήματος. Και οι δύο αλγόριθμοι ελέγχουν το σύνολο των αντικειμένων μόνο μια φορά και παραδίδουν τα αποτελέσματα σταδιακά μόλις είναι διαθέσιμα.

Ορισμός NS: Δοθέντος ενός συνόλου αντικειμένων και ενός σημείου ερωτήματος, ο αλγόριθμος Nearest Surrounder ανακτά τους κοντινότερους γείτονες ενός σημείου ερωτήματος σε διαφορετικές γωνίες. Το αποτέλεσμα είναι ένα σύνολο δυάδων της μορφής $\langle o, [\alpha, \beta]\rangle$, όπου ο είναι το αντικείμενο και [α, β] είναι μια γωνία στην οποία το ο είναι ο κοντινότερος γείτονας του σημείου ερωτήματος.

2.4.1 Προεργασία

Χρησιμοποιούνται οι γωνιακές ιδιότητες του Minimum Bounding Rectangle (MBR) ώστε να αποκομισθούν ευρετικές μέθοδοι για αποτελεσματική αναζήτηση NS.

Ως *γωνιακό εύρος* ενός MBR ορίζεται η ελάχιστη και η μέγιστη γωνία του MBR από το σημείο ερωτήματος q. Αν υποθέσουμε ότι $\theta^{\vdash}_{q,R}$ είναι η ελάχιστη γωνία του MBR του αντικειμένου R και $\theta^{\dashv}_{q,R}$ η μέγιστη, τότε ισχύει $\theta^{\vdash}_{q,R} \leq \theta^{\dashv}_{q,R}$. Το γωνιακό εύρος κάθε MBR - κόμβου στο R-tree πρέπει να είναι αρκετά ευρύ ώστε να περιέχει τα MBRs των παιδιών του. Οι κόμβοι του R-tree επισκέπτονται σύμφωνα με τις ελάχιστες γωνίες των MBRs.

Ως *ελάχιστη γωνιακή απόσταση* ενός MBR R ορίζεται η ευκλείδεια απόσταση από το σημείο ερωτήματος q έως το κοντινότερο σημείο του MBR στο εύρος μιας γωνίας α και συμβολίζεται με madist(q, R, α). Η σχέση ελάχιστης γωνιακής απόστασης ενός MBR Rp με οποιοδήποτε από τα παιδιά που αυτό περικλείει Rc είναι, $madist(q, R_p, \alpha) \leq madist(q, R_c, \alpha)$.

Όριο ελάχιστης γωνιακής απόστασης: Δοθέντος ενός γωνιακού εύρους $[\theta^{\vdash}, \theta^{\dashv}]$ αποφασίζεται ένα εύρος ελάχιστων γωνιακών αποστάσεων, το $max\_madist(q, R, [\theta^{\vdash}, \theta^{\dashv}])$ και το $min\_madist(q, R, [\theta^{\vdash}, \theta^{\dashv}])$ αναφέρονται αντίστοιχα στη μέγιστη και την ελάχιστη απόσταση ανάμεσα σε όλες τις ελάχιστες γωνιακές αποστάσεις για τη συγκεκριμένη γωνία.

Μια μέθοδος κλαδέματος του χώρου αναζήτησης που απορρέει είναι ότι εάν η min_madist ενός κόμβου είναι μεγαλύτερη από τη max_madist ενός NS για το ίδιο γωνιακό εύρος, όλα τα παιδιά του κόμβου αυτού δεν περιέχουν NS και έτσι αγνοούνται.

2.4.2 Σύγκριση Αντικειμένων

Ένα βασικό θέμα στην επεξεργασία των NS ερωτημάτων είναι να αποφασιστούν ποια αντικείμενα ανάμεσα σε όλα τα υποψήφια είναι NS σε ένα γωνιακό εύρος. Η σύγκριση αντικειμένων είναι μια δυαδική συνάρτηση, η οποία συγκρίνει δύο αντικείμενα MBRs και επιστρέφει το πιο κοντινό σε ολόκληρο το γωνιακό εύρος ή και τα δύο αντικείμενα με την αντίστοιχη διαιρεμένη γωνία στην οποία κάθε αντικείμενο είναι κοντινότερο.

Κάθε MBR έχει το πολύ δύο πλευρές απέναντι από το σημείο ερωτήματος, μια οριζόντια και μια κάθετη. Στη συνέχεια σχεδιάζονται οι καμπύλες των γωνιακών αποστάσεων από το σημείο ερωτήματος στις πλευρές αυτές. Από τις καμπύλες αυτές μπορεί να αποφασιστεί το κοντινότερο αντικείμενο για μια συγκεκριμένη γωνία. Υπάρχουν τέσσερεις περιπτώσεις για την αλληλεπίδραση μεταξύ των ακμών των αντικειμένων:

- Παράλληλες ακμές χωρίς αλληλεπίδραση.
- Παράλληλες ακμές με αλληλεπίδραση.
- Ορθογώνιες ακμές χωρίς αλληλεπίδραση.
- Ορθογώνιες ακμές με αλληλεπίδραση.

Στη συνέχεια παρουσιάζονται οι αλγόριθμοι για την NS αναζήτηση.

2.4.3 Sweep

Ο αλγόριθμος Sweep διατηρεί μια ουρά προτεραιότητας με τους κόμβους και τα αντικείμενα ταξινομημένα σύμφωνα με τις γωνιές έναρξης. Αρχικά το αποτέλεσμα είναι $\{\langle \perp : [0, 2\pi]\rangle\}$ όπου $\perp$ είναι το κενό και η ελάχιστη γωνιακή απόσταση είναι ∞. Αρχικά η ρίζα του R-tree εισάγετε στην ουρά προτεραιότητας και στη συνέχεια παίρνουμε την είσοδο από την κορυφή της ουράς. Αν η είσοδος είναι κόμβος τα παιδιά του τοποθετούνται πάλι στην ουρά ενώ αν είναι αντικείμενο συγκρίνεται με το τρέχον αποτέλεσμα NS για την ίδια γωνία, χρησιμοποιώντας τη συνάρτηση σύγκρισης αντικειμένων. Το αποτέλεσμα που προκύπτει προστίθεται στο NS αποτέλεσμα. Αυτή η διαδικασία συνεχίζεται μέχρι να εξεταστούν όλες οι είσοδοι της ουράς.

Μέθοδοι Καλύτερης Απόδοσης του Sweep

Επειδή δεν συνεισφέρουν όλες οι είσοδοι της ουράς προτεραιότητας στο NS αποτέλεσμα, έχουν επινοηθεί διάφορες ευρετικές μέθοδοι για το «κλάδεμα» του χώρου αναζήτησης. Αυτές οι μέθοδοι περιγράφονται στη συνέχεια.

Μέθοδος 1: Δοθείσας της γωνίας ενός MBR R, $[\theta^{\vdash}_{q,R}, \theta^{\dashv}_{q,R}]$, όλα τα NS αντικείμενα του αποτελέσματος των οποίων τα γωνιακά όρια αλληλεπιδρούν με αυτό επιλέγονται και το μέγιστο από τις max_madist υπολογίζεται και ονομάζεται πάνω όριο. Εάν η $min\_madist(q, R, [\theta^{\vdash}_{q,R}, \theta^{\dashv}_{q,R}])$ είναι μεγαλύτερη από το πάνω όριο τότε το R είτε σαν αντικείμενο είτε σαν κόμβος αντικειμένων μπορεί να αγνοηθεί.

Μέθοδος 2: Υποθέτοντας ότι υπάρχει μια γωνία $[\theta^{\vdash}, \theta^{\dashv}]$ που καλύπτει όλα τα NS αντικείμενα του αποτελέσματος, ένα MBR R είναι πιθανότερο να περιέχει NS αντικείμενο αν έχει τη μικρότερη γωνιακή απόσταση στην κοινή γωνία $[\theta^{\vdash}, \theta^{\dashv}] \cap [\theta^{\vdash}_{q,R}, \theta^{\dashv}_{q,R}]$ ανάμεσα σε όλες τις άλλες εισόδους της ουράς.

Μέθοδος 3: Αν η γωνία έναρξης της εισόδου στην κορυφή της ουράς προτεραιότητας είναι α, όλες οι υπόλοιπες είσοδοι της ουράς δεν επηρεάζουν το NS αποτέλεσμα εάν έχουν γωνίες μικρότερες από α.

Αλγόριθμος Sweep

1. Αρχικοποίηση μεταβλητών ως εξής:
   - Στην ουρά προτεραιότητας εισάγεται η ρίζα του R-tree.
   - Γωνία α είναι η μέγιστη γωνία των αντικειμένων που έχουν εξεταστεί και αρχικοποιείται στο 0.
   - Το αποτέλεσμα NS τίθεται στο $\{\langle \perp : [0, 2\pi]\rangle\}$
2. Η ουρά προτεραιότητας εξετάζεται αναδρομικά μέχρι να αδειάσει οπότε και τελειώνει ο αλγόριθμος. Καλείται η συνάρτηση Lookahead για να επιλέξει την καλύτερη είσοδο της ουράς. Η συνάρτηση Lookahead επιλέγει κατά

προτεραιότητα τις εισόδους που καλύπτονται πλήρως από την παράμετρο της γωνίας. Στη συνέχεια επιλέγει εισόδους με τη μικρότερη min_madist στην κοινή γωνία.

3. Για την είσοδο που ανακτήθηκε από την ουρά εφαρμόζεται η ευρετική μέθοδος 1.
4. Εφόσον η είσοδος δεν αγνοηθεί εξετάζουμε εάν πρόκειται για ενδιάμεσο κόμβο ή αντικείμενο. Αν είναι ενδιάμεσος κόμβος τα παιδιά του εισάγονται στην ουρά, αν είναι αντικείμενο καλείται η συνάρτηση NSIncorporate ώστε να αναθεωρήσει το αποτέλεσμα NS. Η συνάρτηση NSIncorporate παίρνει σαν είσοδο ένα αντικείμενο MBR o, το αποτέλεσμα NS και το σημείο ερωτήματος. Εξάγει από το NS σύνολο ένα αντικείμενο του οποίου η γωνία αλληλεπιδρά με αυτή του o και καλεί την ObjectCompare ώστε να βρει το κοντινότερο αντικείμενο ανάμεσα στα δύο για τη συγκεκριμένη γωνία.

Ο αλγόριθμος Sweep μπορεί να επεκταθεί και για πολυεπίπεδη NS αναζήτηση. Πολυεπίπεδη NS αναζήτηση σημαίνει να βρεθούν τα NS αντικείμενα στα m πρώτα επίπεδα από το σημείο ερωτήματος. Για την αναπαράσταση των NS αντικειμένων σε κάθε γωνιακό εύρος χρησιμοποιούνται m οπές αντικειμένων. Για παράδειγμα για 2-επίπεδη NS αναζήτηση το αποτέλεσμα αρχικοποιείται ως εξής: $\{\langle(\perp,\perp) : [0, 2\pi)\rangle\}$. Η λογική του αλγορίθμου Sweep για την πολυεπίπεδη αναζήτηση είναι η ίδια με αυτήν που περιγράφτηκε για το πρώτο επίπεδο.

2.4.4 Ripple

Σε αντίθεση με τον Sweep ο αλγόριθμος Ripple διατηρεί μια ουρά προτεραιότητας των αντικειμένων σε αύξουσα απόσταση από το σημείο ερωτήματος. Έτσι ο Ripple ξεκινάει από το σημείο ερωτήματος προς τα έξω και μπορεί να τερματιστεί πριν εξεταστεί όλη η ουρά σύμφωνα με την ευρετική μέθοδο 4.

Μέθοδος 4: Η εξέταση της ουράς προτεραιότητας μπορεί να παραληφθεί εφόσον ικανοποιούνται δύο συνθήκες. Πρώτον το NS αποτέλεσμα δεν περιέχει το αρχικό κενό αντικείμενο και δεύτερον όλες οι υπόλοιπες είσοδοι έχουν αποστάσεις από το σημείο ερωτήματος, μεγαλύτερες από το πάνω όριο ολόκληρου του NS αποτελέσματος.

Με άλλα λόγια ο αλγόριθμος μπορεί να τερματιστεί όταν η ουρά προτεραιότητας εξαντληθεί ή όταν ικανοποιούνται οι ακόλουθες συνθήκες τερματισμού: (i) σε κάθε κατεύθυνση υπάρχει ένα NS, (ii) όλα τα αντικείμενα στην ουρά προτεραιότητας είναι έξω από το μικρότερο κύκλο (με κέντρο το σημείο ερωτήματος) που περικλείει όλες τις NS απαντήσεις. Η συνθήκη αυτή τερματισμού ονομάζεται αλλιώς *NS - TC* (Nearest Surrounder - Termination Condition).

Ο αλγόριθμος Ripple μπορεί να επεκταθεί και για πολυεπίπεδη αρχιτεκτονική. Η ιδέα της πολυεπίπεδης αρχιτεκτονικής του Ripple είναι διαφορετική από αυτή του Sweep. Αντί να διατηρούνται m NS οπές σε κάθε δυάδα NS αποτελέσματος, διατηρείται μια μήτρα από m NS αποτελέσματα. Η δομή του πολυεπίπεδου NS αποτελέσματος μοιάζει με αυτή των δαχτυλιδιών του κρεμμυδιού. Τα NS αποτελέσματα των χαμηλότερων στοιβάδων συμπληρώνονται νωρίτερα από

αυτά των υψηλότερων στοιβάδων. Όταν ένα αντικείμενο εξετάζεται μελετάται πρώτα πως συνεισφέρει στα χαμηλότερα επίπεδα. Αν το αντικείμενο δεν είναι κοντά στο σημείο ερωτήματος ως προς το τρέχον επίπεδο εξετάζεται μήπως συνεισφέρει σε κάποιο άλλο μεγαλύτερο επίπεδο. Διαφορετικά αν το αντικείμενο είναι πιο κοντά από κάποιο μέρος των NS αντικειμένων που ίσως υπάρχουν, τότε εξετάζουμε όλα τα υπόλοιπα NS αντικείμενα και τα αντικείμενα που θα αντικατασταθούν στο αμέσως επόμενο επίπεδο.

Η μέθοδος 4 επεκτείνεται στη μέθοδο 5 διατηρώντας την πρώτη συνθήκη της μεθόδου 4 και διαμορφώνοντας τη δεύτερη ως εξής:

Μέθοδος 5: Αν η κεφαλή της ουράς προτεραιότητας είναι πιο μακριά από το άνω όριο του NS αποτελέσματος στο τρέχον επίπεδο, τότε οι υπόλοιπες είσοδοι δεν επηρεάζουν το NS αποτέλεσμα σε αυτό το επίπεδο.

Με τη μέθοδο 5 ελέγχεται αν το NS αποτέλεσμα του τρέχοντος επιπέδου έχει ολοκληρωθεί. Εάν ναι, παραδίδεται ένα μερικό σύνολο.

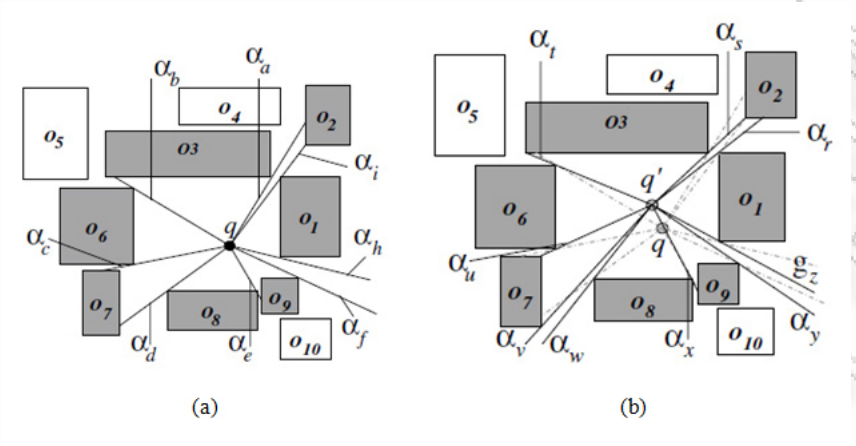
2.4.5 Σύγκριση Ripple με Sweep

Ο αλγόριθμος Ripple φαίνεται πιο γρήγορος από τον Sweep επειδή η συνθήκη τερματισμού κάνει τον Ripple να τερματίζει χωρίς να εξετάζει την υπόλοιπη ουρά. Παρόλα αυτά ο συντομότερος χρόνος τερματισμού του Ripple δεν συνεπάγεται και καλύτερη απόδοση. Δηλαδή εάν δεν υπάρχει NS σε συγκεκριμένες γωνίες ο Ripple θα συνεχίζει να εξετάζει τυφλά όλες τις εισόδους της ουράς. Εάν τα κενά είναι διασπαρμένα ο Ripple εξετάζει συνεχόμενα εισόδους που αλληλεπιδρούν με τα κενά αυτά με αποτέλεσμα να αυξάνεται το υπολογιστικό κόστος. Στην ίδια περίπτωση ο Sweep θα λειτουργούσε καλύτερα επειδή εξετάζει τα αντικείμενα σύμφωνα με την αύξουσα γωνία τους και έτσι μπορεί να προσπεράσει τα κενά.

2.5 Ερωτήματα Κοντινότερου Περιβάλλοντος σε Κινούμενα Αντικείμενα

Το άρθρο [4] περιγράφει ένα σύστημα για τον εντοπισμό NS αντικειμένων σε περιβάλλοντα όπου τα αντικείμενα είναι σε κίνηση. Σε μεγάλης κλίμακας εφαρμογές όπου πολλά αντικείμενα αλλάζουν θέσεις είναι κρίσιμος για την επιτυχία του συστήματος ο αποδοτικός τρόπος επικαιροποίησης των NS ερωτημάτων και η ανανέωση του NS αποτελέσματος.

Ας υποθέσουμε ότι το αντικείμενο o_1 της εικόνας 2.5 μετακινείται προς τα κάτω, όπως φαίνεται στην εικόνα 2.6a, ενώ τα άλλα αντικείμενα παραμένουν σταθερά. Συνεπώς το o_2 που προηγουμένως ήταν κρυμμένο από το o_1, γίνεται νέο μέλος του συνόλου αποτελεσμάτων με τα κοντινότερα περιβάλλοντα αντικείμενα του q και η γωνία $[a_f, a_g)$ που ήταν μια κενή περιοχή καλύπτεται μερικώς από το o_1. Το νέο NS αποτέλεσμα γίνεται:
$NS(q) = \{\langle o_1, [\alpha_h, \alpha_i]\rangle, \langle o_2, [\alpha_i, \alpha_\alpha]\rangle, \langle o_3, [\alpha_\alpha, \alpha_b]\rangle, \langle o_6, [\alpha_b, \alpha_c]\rangle, \langle o_7, [\alpha_c, \alpha_d]\rangle,$
$\langle o_8, [\alpha_d, \alpha_e]\rangle, \langle o_9, [\alpha_e, \alpha_f]\rangle\}$



Εικόνα 2.6. (a) Αλλαγή της θέσης του o1 και (b) αλλαγή θέσης του σημείου ερωτήματος από το q στο q'.

Ο απλός τρόπος θα ήταν να εξεταστούν όλα τα NS ερωτήματα και να αποφασιστούν αυτά που δεν είναι έγκυρα μέσω ερωτημάτων και μέσω αναθεώρησης ερωτημάτων. Προκειμένου να μειωθεί το υπολογιστικό κόστος χρησιμοποιείται η ιδέα της ασφαλούς περιοχής (safe region), ώστε να ελαχιστοποιηθεί το κόστος αναζήτησης. Επίσης χρησιμοποιείται η ιδέα της μερικής αναθεώρησης του ερωτήματος ώστε να επανεξεταστούν μόνο τα μη έγκυρα μέρη του αποτελέσματος. Στη συνέχεια θα αναλυθεί κάθε μια από αυτές τις δύο μεθόδους μείωσης του υπολογιστικού κόστους, το οποίο προκαλείται από την κίνηση των αντικειμένων.

Ασφαλής περιοχή είναι μια περιοχή που οριοθετεί το NS αποτέλεσμα. Δοθείσας μιας ασφαλούς περιοχής, ένα NS ερώτημα χρειάζεται επανεκτίμηση μόνο όταν (i) ένα αντικείμενο κινείται μέσα στην ασφαλή περιοχή, (ii) ένα αντικείμενο φεύγει από την ασφαλή περιοχή (iii) ένα αντικείμενο μπαίνει στην ασφαλή περιοχή. Με άλλα λόγια όλες οι ανανεώσεις έξω από την ασφαλή περιοχή αγνοούνται εφόσον δεν επηρεάζουν το NS αποτέλεσμα. Η μερική αναθεώρηση ερωτήματος αποφασίζει μόνο τις απαραίτητες αλλαγές του αποτελέσματος από το να εκτελεί τα ερωτήματα εξαρχής και έτσι μειώνεται το κόστος της επεξεργασίας.

Ένα NS αποτέλεσμα γίνεται ανεπίκαιρο όχι μόνο όταν μετακινούνται τα NS αντικείμενα αλλά και όταν μετακινείται το σημείο ερωτήματος σε μια νέα θέση. Συνεπώς το ερώτημα πρέπει να επανατεθεί με συνέπεια να αυξάνεται το υπολογιστικό κόστος. Το θέμα γίνεται πιο σοβαρό εάν το πλήθος των ερωτημάτων που τίθενται είναι πολύ μεγάλο και το σημείο ερωτήματος ανανεώνει τη θέση του συχνά. Στην πραγματικότητα όταν η νέα θέση του σημείου ερωτήματος δεν είναι μακριά από την παλιά, το NS αποτέλεσμα δεν είναι και τόσο διαφορετικό. Αναπτύσσεται λοιπόν η μέθοδος της «αυξητικής ανανέωσης του αποτελέσματος» για τη μείωση του κόστους που προκαλείται από την αλλαγή

θέσης του σημείου ερωτήματος. Η μέθοδος αυτή αποφασίζει τις απαραίτητες αλλαγές με βάση τα επιπλέον αντικείμενα που προστίθενται στο αποτέλεσμα. Έτσι μεγιστοποιείται η επαναχρησιμοποίηση των υπαρχόντων αποτελεσμάτων με βάση τις προηγούμενες θέσεις του σημείου ερωτήματος.

Στην εικόνα 2.6b υποτίθεται ότι q' είναι η νέα θέση του σημείου ερωτήματος, όχι πολύ μακριά από την προηγούμενη θέση του q. Στη θέση q' τα ίδια NS αντικείμενα $o_1, o_2, o_3, o_6, o_7, o_8$ ανήκουν στο σύνολο των αποτελεσμάτων και δεν προστίθενται νέα. Δηλαδή NSq = NSq'. Όμως οι αντίστοιχες γωνίες των NS αντικειμένων έχουν αλλάξει και το νέο αποτέλεσμα γίνεται:
$NS(q) = \{\langle o_1, [\alpha_z, \alpha_r]\rangle, \langle o_2, [\alpha_r, \alpha_s]\rangle, \langle o_3, [\alpha_s, \alpha_t]\rangle, \langle o_6, [\alpha_t, \alpha_u]\rangle, \langle \perp, [\alpha_u, \alpha_v]\rangle,$
$\langle o_7, [\alpha_v, \alpha_w]\rangle, \langle o_8, [\alpha_w, \alpha_x]\rangle, \langle o_9, [\alpha_x, \alpha_y]\rangle, \langle \perp, [\alpha_y, \alpha_z]\rangle\}$

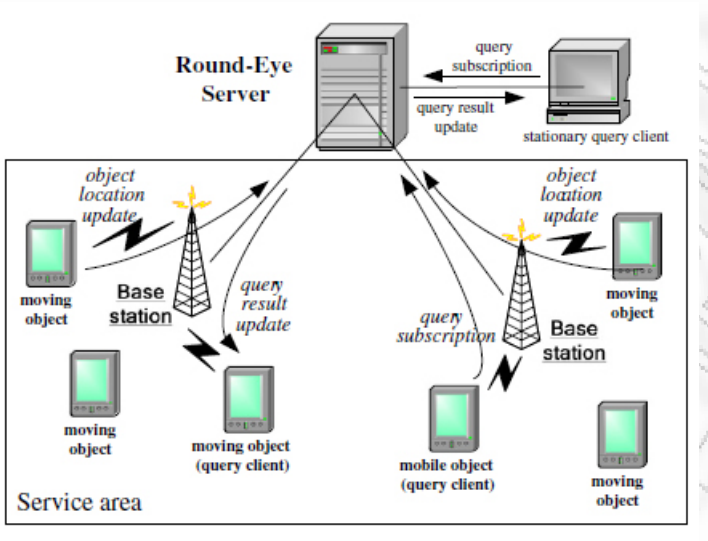
Η δουλειά που παρουσιάζεται στο άρθρο αυτό του Round Eye [4] προτίθεται να δώσει μια νέα διάσταση στην αναζήτηση των κοντινότερων γειτόνων. Δηλαδή τη σχετική θέση του αντικειμένου ως προς τη γωνία του σημείου ερωτήματος με σκοπό την αναζήτηση των κοντινότερων γειτόνων και εστιάζοντας στην ανανέωση του αποτελέσματος. Επιπλέον η μέθοδος αυτή δεν κάνει υποθέσεις για την τροχιά των αντικειμένων όπως γίνεται σε πολλές μεθόδους του τύπου CNNQ.

2.5.1 Αρχιτεκτονική Συστήματος Round-Eye

Ο πιο σημαντικός παράγοντας που πρέπει κανείς να λάβει υπόψη του στο σχεδιασμό του Round - Eye εξυπηρετητή είναι η διάρκεια ανανέωσης του αποτελέσματος. Δηλαδή ο χρόνος από τη στιγμή που ένα ερώτημα γίνεται ανεπίκαιρο (εξαιτίας της αλλαγής της θέσης των αντικειμένων ή του σημείου ερωτήματος) έως την ανανέωση όλων των NS αποτελεσμάτων που επηρεάζονται από την αλλαγή. Προκειμένου να μειωθεί αυτός ο χρόνος προτείνονται διάφορες τεχνικές όπως η «ασφαλής περιοχή», η «μερική αναθεώρηση του ερωτήματος» και η «αυξητική ανανέωση του αποτελέσματος».

Η αρχιτεκτονική του συστήματος Round - Eye αποτελείται από τέσσερα μέλη: (i) ένα σύνολο κινούμενων αντικειμένων (ii) πολλαπλούς σταθμούς βάσης (iii) έναν Round - Eye εξυπηρετητή και (iv) πελάτες ερωτημάτων.

Κάθε κινούμενο αντικείμενο διαθέτει μια συσκευή εντοπισμού θέσης και περιοδικά ελέγχει τη θέση του. Αν αυτή είναι διαφορετική από την προηγούμενη ενημερώνει τον σταθμό βάσης μέσω ασύρματης δικτύωσης. Στη συνέχεια αυτός ο σταθμός βάσης στέλνει αυτές τις αναφορές των αντικειμένων στον κεντρικό εξυπηρετητή μέσω ενσύρματης δικτύωσης. Ο κεντρικός εξυπηρετητής καταγράφει τις θέσεις των κινούμενων αντικειμένων και είναι υπεύθυνος για την επεξεργασία των NS ερωτημάτων. Οι πελάτες που θέτουν τα ερωτήματα αποθηκεύουν τα δικά τους αποτελέσματα και μπορούν να συγκρίνουν τα νέα αποτελέσματα με αυτά που έχουν. Έτσι είναι δυνατόν να γίνεται μερική αναθεώρηση του ερωτήματος και να παραδίδονται μόνο τα αποτελέσματα που είναι διαφορετικά από τα προηγούμενα.



Εικόνα 2.7. Αρχιτεκτονική συστήματος Round-Eye.

2.5.2 Δομή Round-Eye Εξυπηρετητή

Ο Round - Eye εξυπηρετητής αποτελείται από πέντε στοιχεία:

1. την ουρά αναμονής των αιτημάτων,
2. το ευρετήριο των αντικειμένων,
3. τη βάση των ερωτημάτων,
4. τον επεξεργαστή ανανέωσης της θέσης των αντικειμένων,
5. και τον επεξεργαστή των ερωτημάτων.

Όλες οι ερωτήσεις που έρχονται (είτε ερωτήματα, είτε αλλαγή θέσης για αντικείμενα ή σημεία ερωτημάτων) εισάγονται στην ουρά των αιτήσεων. Η βάση των ερωτημάτων διατηρεί πληροφορίες για τα ερωτήματα που υποβάλλονται. Τέτοιες πληροφορίες είναι το id του ερωτήματος, η θέση του, το αποτέλεσμα και πληροφορίες επικοινωνίας (όπως η διεύθυνση IP) με τον πελάτη που θέτει το ερώτημα. Ο επεξεργαστής των ερωτημάτων αναλαμβάνει τα ερωτήματα που υποβάλλονται, την ανανέωση της θέσης ερωτήματος, την ακύρωση ενός ερωτήματος. Ο επεξεργαστής ανανέωσης θέσης πυροδοτείται όταν υποβάλλεται ένα αίτημα ανανέωσης θέσης αντικειμένου.

2.5.3 Ασφαλείς Περιοχές

Οι ασφαλείς περιοχές διακρίνονται στις ασφαλείς περιοχές κλειστών γωνιών και στις ασφαλείς περιοχές ανοιχτών γωνιών, δηλαδή τα δύο είδη γωνιών που υπάρχουν στα NS σύνολα αποτελεσμάτων. Οι πρώτες παριστάνουν γωνίες όπου

τα αντικείμενα είναι γνωστά ενώ στις δεύτερες δεν υπάρχουν γνωστά αντικείμενα. Ακολούθως περιγράφεται πως οργανώνονται οι ασφαλείς περιοχές έτσι ώστε να επιταχυνθεί η διαδικασία των ερωτημάτων αναζήτησης.

Ασφαλής Περιοχή Κλειστού Γωνιακού Εύρους

Στις κλειστές γωνίες ενός NS ερωτήματος ενδιαφέρει εάν ένα υπάρχον NS αντικείμενο διαγράφεται ή ένα νέο αντικείμενο τοποθετείται πιο κοντά στο σημείο ερωτήματος από όλα τα άλλα υπάρχοντα NS αντικείμενα. Δοθέντος ενός σημείου ερωτήματος σχηματίζεται ένα κοίλο πολύγωνο από τις αντικριστές πλευρές όλων των NS αντικειμένων που περιβάλουν αυτό το σημείο. Το κάθε αντικείμενο συγκρίνεται ως προς τη θέση του με το πολύγωνο. Επειδή όμως ο υπολογισμός της επικάλυψης ανάμεσα στο αντικείμενο και το πολύγωνο είναι πολύπλοκος χρησιμοποιείται ο κύκλος σαν ασφαλής περιοχή. Ο κύκλος αυτός περικλείει το πολύγωνο και έτσι μειώνει το υπολογιστικό κόστος.

Κύκλος: Σχηματίζεται ένας κύκλος με κέντρο το σημείο ερωτήματος q και ακτίνα ίση με τη μέγιστη απόσταση από το q προς όλα τα NS αντικείμενα για όλες τις κλειστές γωνίες. Έτσι ο κύκλος περικλείει το κοίλο πολύγωνο. Συνεπώς ο επαναπροσδιορισμός του NS αποτελέσματος γίνεται μόνο όταν η περιοχή του αντικειμένου - όταν αυτό διαγράφεται, εισάγεται ή ανανεώνει τη θέση του - αλληλεπιδρά με τον κύκλο. Δηλαδή μόνο όταν η ελάχιστη απόσταση ανάμεσα στο αντικείμενο και το σημείο ερωτήματος δεν ξεπερνά την ακτίνα του αντίστοιχου κύκλου.

Χρησιμοποιώντας τον κύκλο για την προσέγγιση του κοίλου πολυγώνου μειώνεται το υπολογιστικό κόστος αλλά προκύπτουν λανθασμένα αποτελέσματα επειδή ένα αντικείμενο που εξετάζεται μπορεί να είναι μέσα στον κύκλο αλλά δεν είναι απαραίτητα μέσα στο κοίλο πολύγωνο. Συνεπώς προτείνεται ένας άλλος τύπος ασφαλούς περιοχής το τεταρτοκύκλιο σύνορο.

Τεταρτοκύκλιο παραλληλόγραμμο σύνορο: Ο χώρος με κέντρο το σημείο ερωτήματος χωρίζεται σε τέσσερα τεταρτημόρια. Για κάθε τεταρτημόριο σχηματίζεται ένα παραλληλόγραμμο που καλύπτει στενά τον κενό χώρο για όλες τις κλειστές γωνίες στο τεταρτημόριο. Με το τεταρτοκύκλιο παραλληλόγραμμο σύνορο επιχειρείται να ελαχιστοποιηθεί η διαφορά ανάμεσα στην περιοχή που καλύπτει και εκείνη του πολυγώνου. Έτσι το τεταρτοκύκλιο παραλληλόγραμμο οριοθετεί τον κενό χώρο πιο στενά από τον κύκλο.

Το πολύγωνο για κάθε NS ερώτημα αναπαριστάται από τέσσερα παραλληλόγραμμα με αποτέλεσμα να χρησιμοποιείται περισσότερη μνήμη και να αυξάνεται το πλήθος των δεικτών. Παρόλα αυτά μειώνονται σημαντικά οι μη απαραίτητες επανεκτιμήσεις εξαιτίας της υψηλότερης ακρίβειας στην αναπαράσταση, γεγονός που αιτιολογεί και την προσέγγιση.

Ασφαλής Περιοχή Ανοιχτού Γωνιακού Εύρους

Στις ανοιχτές γωνίες ενός NS ερωτήματος ενδιαφέρει εάν ένα νέο αντικείμενο εισάγεται στην ανοιχτή γωνία και γίνεται νέο μέλος στο NS αποτέλεσμα. Για

να αυξηθεί η απόδοση των ερωτημάτων αναζήτησης ορίζονται ασφαλείς περιοχές ανοιχτού γωνιακού εύρους. Μια ανοιχτή γωνία καλύπτει έναν κενό χώρο. Βασισμένοι στην υπόθεση ότι η περιοχή αναζήτησης είναι οριοθετημένη, η κενή περιοχή είναι ένα τρίγωνο (ή πολλαπλά τρίγωνα, εάν η ανοιχτή γωνία αγγίζει περισσότερες από μια πλευρές της περιοχής αναζήτησης), το οποίο σχηματίζεται από τις δύο ακτίνες της ανοιχτής γωνίας και την πλευρά της περιοχής αναζήτησης. Εφόσον κανένα αντικείμενο δεν βρίσκεται μέσα στο τρίγωνο, θα αναφέρεται σαν κοίλο τρίγωνο.

Εάν χρησιμοποιήσουμε ένα απλό ορθογώνιο παραλληλόγραμμο για να ορίσουμε το τρίγωνο, η ασφαλή περιοχή τουλάχιστον διπλασιάζει τον χώρο που καλύπτεται από το κοίλο τρίγωνο. Έτσι ένα αντικείμενο που αλληλεπιδρά με την ασφαλή περιοχή μπορεί με μεγάλη πιθανότητα να μην πέφτει μέσα στο τρίγωνο με συνέπεια να προκύπτουν λανθασμένα αποτελέσματα.

Προκειμένου να μειωθούν τα λανθασμένα αποτελέσματα προτείνεται ένας απλός και αποδοτικός τρόπος ορισμού του τριγώνου με παραλληλόγραμμα. Το τρίγωνο κόβεται κάθετα και κάθε τμήμα του ορίζεται με ένα ορθογώνιο παραλληλόγραμμο. Όσο περισσότερα παραλληλόγραμμα χρησιμοποιηθούν τόσο περισσότερο προσεγγίζεται το σχήμα του τριγώνου και τόσο περισσότερο αυξάνεται αναλογικά το κόστος αναπαράστασης. Για να εξισορροπηθεί η ακρίβεια της ασφαλούς περιοχής με το κόστος αποθήκευσης, ορίζεται ένα κατώφλι.

2.5.4 Επανεξέταση NS ερωτήματος Λόγω Αλλαγής Θέσης του Αντικειμένου

Όταν ένα κινούμενο αντικείμενο αλλάζει τη θέση του, το NS ερώτημα πρέπει να επανατεθεί ώστε να αντικατοπτρίζει τα νέα αποτελέσματα. Κάθε τερματικό που θέτει ένα ερώτημα έχει τη δυνατότητα να ενσωματώνει τα αναθεωρημένα αποτελέσματα στο αρχικό σύνολο αποτελεσμάτων. Έτσι υιοθετείται η προσέγγιση της μερικής αναθεώρησης του ερωτήματος, για να εκτιμηθούν μόνο τα ανεπίκαιρα μέρη του αποτελέσματος και να παραδοθούν οι αλλαγές του αποτελέσματος μόνο για αυτά τα μέρη. Κατά συνέπεια μειώνεται το υπολογιστικό κόστος και η κατανάλωση του bandwidth - για την παράδοση των αποτελεσμάτων - σε σύγκριση με τον επαναπροσδιορισμό του ερωτήματος από την αρχή.

Μερική Αναθεώρηση Ερωτήματος

Στις περισσότερες περιπτώσεις η ανανέωση της θέσης ενός αντικειμένου αναθεωρεί μόνο ένα μέρος του NS αποτελέσματος. Έτσι η αναθεώρηση των ανεπίκαιρων αποτελεσμάτων θα ήταν αποδοτική ενώ τα ανεπηρέαστα μέρη του αποτελέσματος μπορούν να διατηρηθούν.

Αναθεώρηση του αποτελέσματος προκύπτει όταν ένα νέο αντικείμενο μπαίνει στην ασφαλή περιοχή ή ένα άλλο βγαίνει από αυτή. Χωρίς παραβίαση της γενικότητας αντιμετωπίζουμε την κίνηση ενός αντικειμένου εντός της ασφαλούς περιοχής σαν τη διαγραφή και ακολούθως την είσοδο του αντικειμένου.

Όταν διαγράφεται ένα αντικείμενο ο όλα τα NS ερωτήματα που περιέχουν το ο στα αποτελέσματά τους γίνονται ανεπίκαιρα. Κάθε NS αποτέλεσμα περιέχει δυάδες της μορφής $(o, [\alpha, b]) \in NS(q)$. Με τη διαγραφή του ο η δυάδα αυτή πρέπει να αναθεωρηθεί με τα καινούργια o' NS αντικείμενα που βρίσκονται στη γωνία [α, b]. Εάν δεν υπάρχει κανένα NS αντικείμενο στη γωνία αυτή, τότε η δυάδα γράφεται $(\perp, [\alpha, b])$. Παρατηρούμε ότι η ένωση των γωνιών από όλες τις διαγραφείσες δυάδες ισούται με τις γωνίες όλων των εισαχθέντων δυάδων.

Όπως η διαγραφή αντικειμένου έτσι και η εισαγωγή ενός νέου αντικειμένου καθιστά ανεπίκαιρο μέρος του NS αποτελέσματος. Σε αυτή την περίπτωση επανεξετάζεται μόνο εάν το αντικείμενο που εισάγεται είναι πιο κοντά στο σημείο ερωτήματος από οποιοδήποτε άλλο NS αντικείμενο.

Παράδοση Αναθεωρημένων Αποτελεσμάτων στο Τερματικό

Επειδή η αλλαγή της θέσης ενός αντικειμένου αντιμετωπίζεται σαν διαγραφή του αντικειμένου και εισαγωγή του σε μια νέα θέση, θα δημιουργούνταν λανθασμένη εικόνα εάν τα αποτελέσματα παραδίδονταν πριν την εισαγωγή του αντικειμένου στη νέα θέση. Για να αντιμετωπιστεί αυτό το πρόβλημα οι σύνθετες διαγραφές - εισαγωγές θα πρέπει να παραδίδονται ενιαία. Στην περίπτωση ανεξάρτητων αλλαγών θέσης οι αλλαγές αποτελέσματος συσσωρεύονται πριν παραδοθούν. Αυτό συμβαίνει διότι δύο δυάδες που έχουν ίδιο αντικείμενο και γωνία αλλά η μια αφορά διαγραφή και η άλλη εισαγωγή, αλληλοαναιρούνται και συνεπώς μπορούν να παραληφθούν.

2.5.5 Επανεξέταση NS Ερωτήματος Λόγω Αλλαγής Θέσης του Σημείου Ερωτήματος

Κάθε φορά που ένα σημείο ερωτήματος μετακινείται σε μια νέα θέση πυροδοτεί την ανανέωση του NS αποτελέσματος στέλνοντας το id του και τη νέα του θέση στο εξυπηρετητή των ερωτημάτων. Εξαιτίας της χωρικής θέσης τα αποτελέσματα των NS ερωτημάτων, που λαμβάνουν χώρα κοντά το ένα στο άλλο, θα περιέχουν παρόμοια σύνολα αντικειμένων. Η ομοιότητα των συνόλων αποτελεσμάτων εξαρτάται από την απόσταση ανάμεσα στα σημεία ερωτήματος και την κατανομή των NS αντικειμένων γύρω από αυτά.

Το κυρτό πολύγωνο παρέχει χρήσιμη πληροφορία ώστε να ληφθεί η απόφαση της επεξεργασίας του ερωτήματος. Το κυρτό πολύγωνο είναι ένας κενός χώρος που οριοθετείται από όλα τα τρέχοντα NS αντικείμενα. Αν το σημείο ερωτήματος παραμένει μέσα στον κενό χώρο είναι πολύ πιθανόν να διατηρεί μερικά, αν όχι όλα, από τα υπάρχοντα NS αντικείμενα στο νέο σύνολο αποτελεσμάτων, παρόλο που οι γωνίες για όλα τα NS αντικείμενα έχουν αλλάξει. Από την άλλη για ένα σημείο έξω από τον NS χώρο δεν μπορεί να διαβεβαιωθεί πόσα NS αντικείμενα θα επαναχρησιμοποιήσει και έτσι πρέπει να εξεταστεί από την αρχή. Για λόγους απλοποίησης αφαιρείται το παλιό ερώτημα και εισάγεται

ένα άλλο σε μια νέα θέση. Μετά την επανεξέταση τα προηγούμενα αποτελέσματα που ήταν αποθηκευμένα στο τερματικό αντικαθίστανται πλήρως από τα νέα αποτελέσματα.

Το μοντέλο για την ανανέωση του NS αποτελέσματος, όσο το σημείο ερωτήματος παραμένει μέσα στον κενό χώρο, δουλεύει ως εξής:

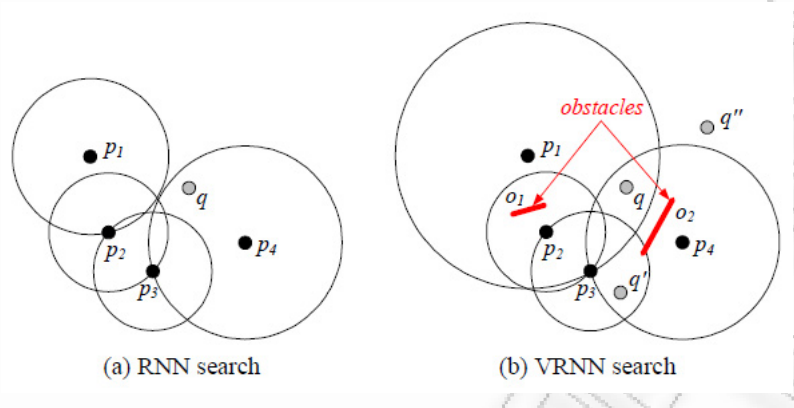
1. Εκτέλεση αναζήτησης βασισμένη στις αντικριστές πλευρές των υπαρχόντων NS αντικειμένων.
2. Αναζήτηση των δεικτών των αντικειμένων για την εύρεση επιπλέον αντικειμένων εάν υπάρχουν.
3. Παράδοση των επιπλέον αντικειμένων στα τερματικά που θέτουν τα ερωτήματα.
4. Ανανέωση του αποτελέσματος που διατηρείται στο τερματικό και στη βάση δεδομένων των ερωτημάτων στον εξυπηρετητή.

2.6 Ερωτήματα Ορατού Αντίστροφου Κοντινότερου Γείτονα

Τα ερωτήματα αντίστροφου κοντινότερου γείτονα (Reverse Nearest Neighbor) έχουν ευρεία εφαρμογή σε συστήματα όπως η υποστήριξη αποφάσεων, το μάρκετινγκ με βάση το προφίλ, η κατανομή πόρων και η εξόρυξη δεδομένων. Ένας αλγόριθμος που επεξεργάζεται ερωτήματα ορατού αντίστροφου κοντινότερου γείτονα (Visible Reverse k Nearest Neighbor - VRkNN) παρουσιάζεται στο άρθρο [5]. Δοθέντος ενός συνόλου δεδομένων P και ενός σημείου ερωτήματος q, ένα ερώτημα VRNN ανακτά τα σημεία του P τα οποία έχουν το q ως τον κοντινότερο ορατό γείτονα. Μια φυσική γενίκευση είναι τα ερωτήματα VRkNN, που βρίσκουν όλα τα σημεία p που ανήκουν στο P τα οποία έχουν το q ως έναν από τους k κοντινότερους ορατούς γείτονες.

Στην εικόνα 2.8a βλέπουμε ένα παράδειγμα με τέσσερα σημεία δεδομένων, τα p_1, p_2, p_3, p_4. Καθένα από αυτά τα σημεία περιβάλλεται από ένα κύκλο με κέντρο το ίδιο το σημείο και ακτίνα την απόσταση του σημείου αυτού από τον κοντινότερο γείτονα του. Με άλλα λόγια ο κύκλος του p_i περικλείει τους κοντινότερους γείτονες του p_i. Αν το σημείο ερωτήματος είναι το q τότε το σύνολο των αποτελεσμάτων είναι $RNN(q) = p_4$, διότι το q βρίσκεται μόνο μέσα στον κύκλο του p_4. Στην εικόνα 2.8b όπου υπάρχουν εμπόδια το σύνολο των αποτελεσμάτων είναι διαφορετικό από αυτό του ερωτήματος RNN και είναι $VRNN(q) = p_1$.

Όλοι οι αλγόριθμοι RNN/ RkNN που έχουν προταθεί μπορούν να διακριθούν σε τρεις βασικές κατηγορίες: (i) σε αυτούς που βασίζονται στην προεπεξεργασία (ii) τους δυναμικούς αλγορίθμους και (iii) σε αλγορίθμους για την επεξεργασία των διαφόρων τύπων RNN/ RkNN ερωτημάτων. Το καινοτόμο που προτείνεται στο άρθρο [5] σε σχέση με τους υπόλοιπους αλγορίθμους είναι ότι λαμβάνονται υπόψη τα φυσικά εμπόδια που υπάρχουν στον πραγματικό κόσμο και μπορεί να επηρεάζουν την ορατότητα/απόσταση ανάμεσα σε δύο αντικείμενα. Ο αλγόριθμος στο άρθρο αυτό δομεί τα σύνολα δεδομένων και



Εικόνα 2.8. Παράδειγμα ερωτημάτων RNN και VRNN.

εμποδίων σε R-trees ευρετήρια. Η μέθοδος τους δεν απαιτεί προεπεξεργασία, εκμεταλλεύεται τις ιδιότητες του επιπέδου και κάνει έλεγχο ορατότητας προκειμένου να κλαδέψει το χώρο αναζήτησης. Επίσης ακολουθείται η διαδικασία του φιλτραρίσματος και της εκλέπτυνσης. Η μέθοδος αυτή εγγυάται την ορθότητα του αποτελέσματος καθώς οι εναπομείναντες κόμβοι και σημεία επανεξετάζονται προκειμένου να εξαλειφθούν τα σφάλματα.

3

Ανάλυση Απαιτήσεων

Στο παρόν κεφάλαιο συγκρίνονται όλες οι πρακτικές που μελετήθηκαν διεξοδικά στο κεφάλαιο της Επισκόπησης των Σύγχρονων Πρακτικών με στόχο με γίνουν περισσότερο ευδιάκριτες οι ομοιότητες και οι διαφορές αυτών των πρακτικών. Μέσα από αυτή τη διαδικασία θα επιλεγούν δύο πρακτικές ώστε να επεκταθούν και τελικά να εξομοιωθούν. Στη συνέχεια του κεφαλαίου συγκρίνονται αναλυτικά οι αλγόριθμοι των ερωτημάτων που επιλέχθηκαν να επεκταθούν και έτσι γίνονται πιο σαφείς οι τροποποιήσεις που απαιτούνται σε κάθε αλγόριθμο.

3.1 Συγκριτική Παρουσίαση των Πρακτικών LOS

Στον πίνακα 3.1 βλέπουμε συγκριτικά κάθε μια από τις μεθόδους που μελετήθηκαν στο προηγούμενο κεφάλαιο ανάλογα με τα ερωτήματα που εξετάζουν.

Πίνακας 3.1. Σύγκριση τεχνικών ως προς κάποια βασικά στοιχεία του πραγματικού χώρου.

	Οπτικά εμπόδια - Ορατότητα	Αντικείμενα σε κίνηση	Γωνία	Επιλογή πλήθους κοντινότερων γειτόνων
VkNN [1], [6]	X			X
CVNN [2]	X			
NS [3]	X		X	
Round-Eye [4]	X	X	X	
RVkNN [5]	X			X
AVkNN [6]	X			X

Στον Πίνακα 3.1 βλέπουμε ότι όλες οι πρακτικές που μελετήθηκαν λαμβάνουν υπόψη τα οπτικά εμπόδια που περιορίζουν την ορατότητα των αντικειμένων.

Η τεχνική CVNN [2] δεν εφαρμόζεται ακριβώς για τα αντικείμενα σε κίνηση, αλλά μπορεί δυνητικά να εφαρμοστεί σε αντικείμενα που κινούνται θεωρώντας ότι αυτά έχουν μια προκαθορισμένη τροχιά. Εφόσον τα ερωτήματα CVNN δεν αφορούν ένα σημείο ερωτήματος αλλά μια γραμμή ερωτήματος. Η τεχνική Round-Eye [4] εφαρμόζεται σε αντικείμενα που βρίσκονται σε κίνηση και δεν παίρνει σαν δεδομένο ότι αυτά τα αντικείμενα έχουν προκαθορισμένες τροχιές. Η κατεύθυνση - γωνία των αντικειμένων είναι θέμα που εξετάζουν τα NS [3] ερωτήματα. Ακόμη επειδή η τεχνική Round - Eye βρίσκει τα NS κινούμενα αντικείμενα, λαμβάνει και αυτή υπόψη την κατεύθυνση των αντικειμένων. Η τεχνική VkNN [1], [6] και η αντίστροφή της RVkNN [5] δίνουν τη δυνατότητα στον χρήστη να επιλέξει το πλήθος των κοντινότερων γειτόνων που επιθυμεί να αναζητήσει. Αυτό δηλώνεται με το στοιχείο k. Αν το k που θα δώσει ο χρήστης είναι πολύ μεγάλο τότε ο αλγόριθμος μπορεί να επιστρέψει όλους τους κοντινότερους γείτονες. Το AVkNN [6] ερώτημα το οποίο αποτελεί γενίκευση του VkNN ερωτήματος λαμβάνει υπόψη του ότι και το VkNN, διότι η γενίκευση έγκειται στο ότι χρησιμοποιούνται πολλαπλά σημεία ερωτήματος και όχι μόνο ένα όπως στο VkNN ερώτημα.

Πίνακας 3.2. Σύγκριση τεχνικών ως προς την αποδοτικότητα.

	Μείωση χρόνου απόκρισης	Μείωση υπολογιστικού κόστους	Μείωση κόστους μνήμης	Ορθότητα αποτελέσματος
VkNN [1], [6]	X	X	X	
CVNN [2]	X	X	X	X
NS [3]	X	X	X	
Round-Eye [4]	X	X	X	
RVkNN [5]	X	X	X	X
AVkNN [6]	X	X	X	

Όπως φαίνεται από τον Πίνακα 3.2 όλες οι τεχνικές έχουν διάφορες ευρετικές μεθόδους με τις οποίες επιχειρούν να μειώσουν τον χρόνο απόκρισης, το υπολογιστικό κόστος και το κόστος κατανάλωσης σε μνήμη, αλλά μόνο οι τεχνικές CVNN [2] και RVkNN [5] εγγυώνται την ορθότητα του αποτελέσματος. Διότι ο αλγόριθμος CVNN [2] εξετάζει μόνο τα αντικείμενα που είναι ικανά να συμμετάσχουν στο σύνολο των απαντήσεων και αγνοεί όλα τα υπόλοιπα και ο αλγόριθμος RVkNN [5] εφαρμόζει τεχνικές που εξαλείφουν τα σφάλματα.

3.2 Επιλογή Πρακτικών προς Επέκταση

Από τις τεχνικές που μελετήθηκαν η [1] και η νέα έκδοση της [6] που εξετάζουν τα VkNN ερωτήματα, σχετίζεται περισσότερο με την [3] καθώς αφορούν

και σε θέσεις ερωτημάτων που είναι σταθερές και σημειακές. Η [2] σχετίζεται περισσότερο με την [4]. Στην [4] η θέση ερωτήματος είναι σημειακή αλλά τόσο το σημείο ερωτήματος όσο και τα αντικείμενα προς αναζήτηση βρίσκονται σε κίνηση. Δηλαδή το σημείο ερωτήματος διαγράφει μια μη καθορισμένη τροχιά. Στην τεχνική [2] η θέση ερωτήματος δεν είναι σημειακή αλλά αφορά ένα ευθύγραμμο τμήμα σε κάθε σημείο του οποίου αναζητούμε κοντινότερους γείτονες. Εναλλακτικά στην τεχνική [2] η θέση ερωτήματος θα μπορούσε να θεωρηθεί σημειακή και επιπλέον ότι το σημείο ερωτήματος κινείται στην τροχιά ενός ευθύγραμμου τμήματος σε κάθε σημείο της οποίας αναζητούμε τους κοντινότερους γείτονες. Η τεχνική [5], όπως φαίνεται και από το όνομα της, είναι η αντίστροφη των VkNN ερωτημάτων [1], [6]. Δηλαδή αναζητούμε όχι τα αντικείμενα αλλά το σημείο του χώρου στο οποίο τα αντικείμενα αυτά είναι οι κοντινότεροι ορατοί γείτονες. Η θέση ερωτήματος, για αυτή την πρακτική, είναι σημειακή και σταθερή, όπως και στην ευθεία πρακτική [1], [6]. Όπως προαναφέρθηκε το AVkNN ερώτημα αφορά πολλαπλά σταθερά σημεία ερωτήματος, συνεπώς ως προς τη θέση ερωτήματος δεν μοιάζει με κανένα άλλο από τα ερωτήματα που εξετάστηκαν.

Σχηματίζεται ο πίνακας 3.3 σύγκρισης των τεχνικών των ερωτημάτων οπτικού πεδίου, όπου συγκρίνονται με βάση τις ιδιότητες της θέσης ερωτήματος.

Πίνακας 3.3. Σύγκριση τεχνικών ως προς τη θέση ερωτήματος.

	Ένα σημείο ερωτήματος	Πολλαπλά σημεία ερωτήματος	Γραμμή ερωτήματος	Σταθερή θέση	Κινούμενη θέση
VkNN [1], [6]	X			X	
CVNN [2]			X	X	
NS [3]	X			X	
Round-Eye [4]	X				X
RVkNN [5]	X			X	
AVkNN [6]		X		X	

Από τους πίνακες 3.1 και 3.3 βλέπουμε ότι οι τεχνικές [1], [6] και [3] είναι όμοιες ως προς τη θέση ερωτήματος και διαφέρουν ως προς κάποια χαρακτηριστικά του πίνακα 3.1. Συγκεκριμένα η τεχνική των VkNN ερωτημάτων δίνει στον χρήστη τη δυνατότητα να επιλέξει το πλήθος των κοντινότερων γειτόνων προς αναζήτηση και η τεχνική των NS ερωτημάτων επιστρέφει την αντίστοιχη γωνία των κοντινότερων γειτόνων. Επιλέγονται λοιπόν οι δύο πρακτικές για να εξομοιωθούν με την επέκταση των αλγορίθμων τους έτσι ώστε η τεχνική [6] (νέα έκδοση της [1]) να επιστρέφει επιπλέον την αντίστοιχη γωνία των κοντινότερων γειτόνων που ανακτά και η τεχνική [3] να δίνει την επιλογή στον χρήστη να καθορίζει το πλήθος των κοντινότερων γειτόνων. Οι νέες πρακτικές θα απαντούν στα ερωτήματα των k ορατών κοντινότερων γειτόνων με τις

αντίστοιχες γωνίες τους. Εν συντομία θα ονομάζεται ερώτημα VkNS (Visible k Nearest Surrounder).

3.3 Συγκριτική Επισκόπηση των Αλγορίθμων προς Επέκταση

Προκειμένου ο κώδικας που θα αναπτυχθεί να είναι συνεπής με αυτόν που περιγράφεται στα αντίστοιχα άρθρα αναζητήθηκαν οι υλοποιήσεις από τους ίδιους τους συγγραφείς. Επιπλέον για την κατανόηση και την επέκταση αυτών των πρακτικών χρησιμοποιείται ο ψευδοκώδικας που περιγράφεται στα αντίστοιχα άρθρα.

Κατέστη δυνατό να βρεθεί μόνο ο κώδικας που επεξεργάζεται τα NS ερωτήματα του άρθρου [3]. Για την υλοποίηση και επέκταση των VkNN [6] ερωτημάτων επιλέχτηκε να χρησιμοποιηθεί ως βάση ο κώδικας των NS ερωτημάτων. Κάτι τέτοιο είναι δυνατό να γίνει καθώς το υποστηρίζει και η προσέγγιση [6] των VkNN ερωτημάτων που μελετήθηκε. Ο τρόπος με τον οποίο μπορεί να γίνει αυτό περιγράφεται στην επόμενη ενότητα 3.3.1.

Στο άρθρο [6] περιγράφονται οι αλγόριθμοι PostPruning και PrePruning οι οποίοι διαφέρουν στο εάν το κλάδεμα των αντικειμένων με βάση την ορατότητα, γίνεται πριν ή μετά την ανάκτηση ενός φύλλου του R-tree. Επίσης διαφέρουν στη συνάρτηση απόστασης που χρησιμοποιείται για να ταξινομήσει τις εισόδους στην ουρά προτεραιότητας, αλλά παράγουν τα ίδια αποτελέσματα. Στη συνέχεια παρουσιάζονται με ψευδοκώδικα οι δύο αυτοί αλγόριθμοι.

Στον αλγόριθμο PostPruning οι είσοδοι εισάγονται στην ουρά προτεραιότητας σύμφωνα με την ελάχιστη απόσταση τους από το σημείο ερωτήματος. Αυτό φαίνεται στις γραμμές του κώδικα 14 - 17. Να παρατηρηθεί ότι όταν η είσοδος που ανακτάται από την ουρά είναι αντικείμενο υπολογίζεται η ελάχιστη ορατή απόστασή του από το σημείο ερωτήματος. Στη συνέχεια συγκρίνεται αυτή η ελάχιστη ορατή απόσταση με την ελάχιστη απόσταση της εισόδου στην κεφαλή της ουράς προτεραιότητας. Αν η απόσταση του κεφαλιού της ουράς είναι μικρότερη και η απόσταση του προς εξέταση αντικειμένου δεν είναι άπειρη τότε το αντικείμενο εισάγεται πάλι στην ουρά (γραμμή 10) με απόσταση την ελάχιστη ορατή απόσταση.

Ο αλγόριθμος PrePruning αποτελεί μια βελτιστοποίηση του αλγορίθμου PostPruning σε όρους κόστους εισόδου - εξόδου. Σε αντίθεση με τον PostPruning ο PrePruning εφαρμόζει την ελάχιστη ορατή απόσταση τόσο σε αντικείμενα όσο και σε κόμβους. Επομένως οι κόμβοι κλαδεύονται σύμφωνα με την ορατότητα τους πριν τους επισκεφτούμε.

Έχοντας τη γνώση του τρόπου επεξεργασίας των ερωτημάτων VkNN και NS, μπορούμε να δούμε συγκριτικά τις διαφορές στην υλοποίηση των δύο πρακτικών και έτσι να μπορέσουμε τελικά να τις επεκτείνουμε ώστε ο αλγόριθμος VkNN να καλύπτει επιπλέον τη γωνία και ο αλγόριθμος NS να καλύπτει το k - πλήθος των αντικειμένων.

Αλγόριθμος 3.1 POSTPRUNING(Tree, q, k)

```
1: Create PQ with Tree.ROOT as the first entry
2: Create an empty set A of answers
3: while PQ is not empty and | A | is less than k do
4:         E ← PQ.POPHEAD()
5:         if E contains an object then
6:             E.DST ← Calculate MinViDist(q, E, A)
7:             if E.DST ≤ PQ.HEAD().DST then
8:                 Insert E.OBJ into A
9:             else if E.DST is not infinity then
10:                Insert E back into PQ
11:            end if
12:        else if E contains a node then
13:            Children ← Tree.GETCHILDNODES(E.NODE)
14:            for all C in Children do
15:                D ← Calculate MINDIST(q, C)
16:                Create NewEntry from C and D
17:                Insert NewEntry into PQ
18:            end for
19:        end if
20: end while
21: return A
```

Αλγόριθμος 3.2 PREPRUNING(Tree, q, k)

```
1: Create PQ with Tree.ROOT as the first entry
2: Create an empty set A of answers
3: while PQ is not empty and | A | is less than k do
4:         E ← PQ.POPHEAD()
5:         E.DST ← Calculate MinViDist(q, E, A)
6:         if E.DST > PQ.HEAD().DST then
7:             if E.DST is not infinity then
8:                 insert E back into PQ
9:             end if
10:        else if E contains an object then
11:            Insert E.OBJ into A
12:        else if E contains a node then
13:            Children ← Tree.GETCHILDNODES(E.NODE)
14:            for all C in Children do
15:                D ← Calculate MINDIST(q, C)
16:                Create NewEntry from C and D
17:                Insert NewEntry into PQ
18:            end for
19:        end if
20: end while
21: return A
```

3.3.1 Υλοποίηση του VkNN από το NS

Στο άρθρο [6] συγκρίνεται το πρόβλημα των VkNN ερωτημάτων με τον αλγόριθμο επίλυσης των NS ερωτημάτων του άρθρου [3], δηλαδή οι τεχνικές των άρθρων οι οποίες επιδιώκεται να εξομοιωθούν. Τονίζεται λοιπόν, ότι η κύρια διαφορά ανάμεσα στο ερώτημα NS και στο VkNN είναι ότι το NS βρίσκει όλα τα ορατά αντικείμενα γύρω από το σημείο ερωτήματος ενώ ο αριθμός των ορατών αντικειμένων για το VkNN ερώτημα καθορίζεται από τον χρήστη. Επίσης πιο κάτω στο άρθρο [6], μετά την περιγραφή του αλγορίθμου PostPruning, περιγράφεται εν συντομία πως τροποποιώντας τον αλγόριθμο Ripple των ερωτημάτων NS προκύπτει ο αλγόριθμος PostPruning-NS-TC (NS - Termination Condition), δηλαδή ο αλγόριθμος PostPruning με τη συνθήκη τερματισμού του αλγορίθμου NS.

Στην αυθεντική εκδοχή του αλγόριθμου NS Ripple τα αντικείμενα ανακτούνται από την ουρά προτεραιότητας σύμφωνα με την ελάχιστη απόστασή τους από το σημείο ερωτήματος. Σύμφωνα με το [6] ο αλγόριθμος NS Ripple μπορεί να τροποποιηθεί ώστε να ανακτά σταδιακά τους ορατούς κοντινότερους γείτονες και να σταματά αφού ανακτήσει k κοντινότερους ορατούς γείτονες. Αυτή η τροποποίηση γίνεται εφαρμόζοντας το μέτρο της ελάχιστης ορατής απόστασης και επανεισάγοντας τα αντικείμενα που μπορεί να μην είναι ο επόμενος VNN στην ουρά προτεραιότητας. Αυτή η τροποποίηση καταλήγει στη δημιουργία του αλγορίθμου PostPruning-NS-TC. Επιπλέον αν αυτή η τροποποίηση γίνει αμέσως μόλις ανακτούνται οι είσοδοι από την ουρά προτεραιότητας και επομένως εφαρμοστεί όχι μόνο σε αντικείμενα, όπως στον PostPruning, αλλά και σε κόμβους, τότε θα προκύψει ο αλγόριθμος PrePruning-NS-TC.

Έχοντας τον κώδικα υλοποίησης του αλγορίθμου NS [3] και λαμβάνοντας υπόψη αυτές τις τροποποιήσεις θα δημιουργηθούν οι αλγόριθμοι PostPruning-NS-TC και PrePruning-NS-TC του άρθρου [6], διατηρώντας παράλληλα την ιδιότητα της γωνίας του αλγορίθμου NS. Έτσι θα έχει υλοποιηθεί ο επεκταμένος αλγόριθμος VkNN που θα λαμβάνει υπόψη τη γωνία. Απομένει η δημιουργία του επεκταμένου αλγορίθμου NS στον οποίο θα προστεθεί η παράμετρος k. Αυτή η προσθήκη μπορεί να γίνει είτε στον αλγόριθμο NS - Ripple είτε στον αλγόριθμο NS - Sweep.

Όπως σημειώνεται στο άρθρο [6] ο αλγόριθμος PostPruning-NS-TC έχει την ίδια απόδοση με τον αλγόριθμο PostPruning όταν το k είναι μικρότερο από τον αριθμό των κοντινότερων ορατών γειτόνων. Όταν χρησιμοποιούνται οι αλγόριθμοι αυτοί για την ταξινόμηση όλων των κοντινότερων ορατών γειτόνων στο σύνολο δεδομένων, ο αλγόριθμος PostPruning πάντα επισκέπτεται όλους τους κόμβους του R-tree εξαιτίας της απουσίας συνθήκης τερματισμού. Ο αλγόριθμος PostPruning-NS-TC από την άλλη πλευρά τερματίζει (i) όταν το σημείο ερωτήματος περιβάλλεται εντελώς από κοντινότερους ορατούς γείτονες και (ii) η επόμενη είσοδος της ουράς προτεραιότητας είναι έξω από τον ελάχιστο κύκλο που περικλείει όλους τους τρέχοντες υποψήφιους κοντινότερους ορατούς γείτονες. Έτσι όταν το σημείο ερωτήματος περιβάλλεται από γείτονες και δεν υπάρχει καμία κενή γωνία τότε σχηματίζεται αυτός ο κλειστός κύκλος και

ο αλγόριθμος PostPruning-NS-TC αποδίδει καλύτερα από τον PostPruning. Όταν υπάρχει κενό στους κοντινότερους ορατούς γείτονες οι δύο παραλλαγές του ίδιου αλγορίθμου συμπεριφέρονται το ίδιο. Και στις δύο περιπτώσεις, είτε σχηματίζεται κλειστός κύκλος, είτε υπάρχει κενή γωνία, ο αλγόριθμος PrePruning επισκέπτεται μόνο τους κόμβους που επικαλύπτουν την ορατή περιοχή. Επομένως ο PrePruning έχει μικρότερο κόστος εισόδου - εξόδου και από τις δύο παραλλαγές του αλγορίθμου PostPruning.

Συνεπώς χωρίς βλάβη στην απόδοση θα υλοποιηθεί ο αλγόριθμος PostPruning-NS-TC ο οποίος θα καλύπτει επιπλέον τη γωνία αντί να υλοποιηθεί απλώς ο αλγόριθμος PostPruning.

4

Σχεδιασμός

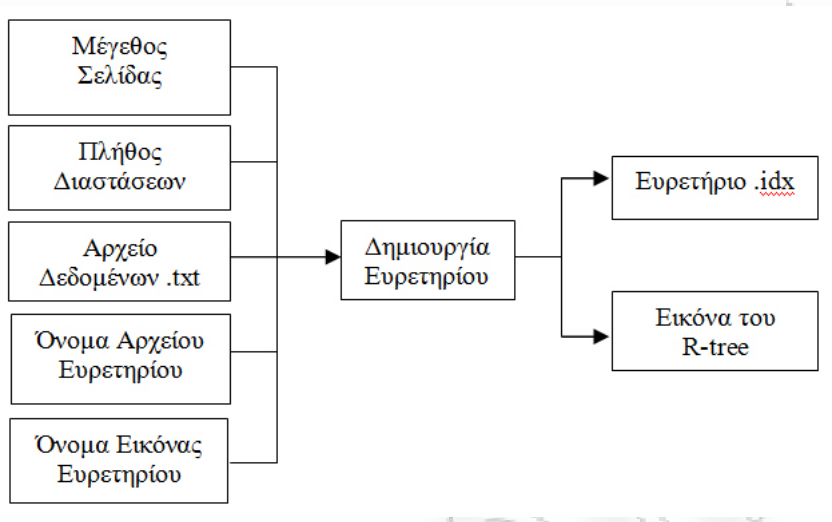
Όπως αναλύθηκε ο κώδικας που θα χρησιμοποιηθεί για την επέκταση των ερωτημάτων NS και VkNN είναι ο κώδικας για την επεξεργασία των ερωτημάτων NS. Στη συνέχεια θα περιγραφεί η αρχιτεκτονική του συστήματος επεξεργασίας των NS ερωτημάτων, ώστε να γίνουν σαφή τα σημεία του κώδικα στα οποία πρέπει να γίνουν οι επεμβάσεις - τροποποιήσεις, προκειμένου να εξομοιωθούν τα ερωτήματα NS και VkNN.

4.1 Αρχιτεκτονική Επεξεργασίας NS Ερωτημάτων

Όπως είδαμε το πιο ευρέως διαδεδομένο ευρετήριο για τα χωρικά δεδομένα είναι το R-tree. Έτσι και για την επεξεργασία των NS ερωτημάτων τα χωρικά δεδομένα θα δομηθούν σε ένα R-tree. Πριν εκτελέσουμε κάποιον από τους αλγορίθμους για την επεξεργασία των NS ερωτημάτων, τον Sweep ή τον Ripple, θα εκτελέσουμε τον αλγόριθμο - που στον κώδικα που δόθηκε από τους συγγραφείς ονομάζεται rtreetest - ο οποίος θα δημιουργήσει ένα αρχείο τύπου ευρετηρίου, με την κατάληξη .idx. Αυτό το αρχείο στη συνέχεια θα χρησιμοποιηθεί σαν είσοδος στον αλγόριθμο Sweep ή Ripple.

Στην εικόνα 4.1 βλέπουμε σε υψηλό επίπεδο την αρχιτεκτονική του κώδικα δόμησης των χωρικών δεδομένων σε R-tree.

Η είσοδος για τη δημιουργία του R-tree είναι βασικά το αρχείο των χωρικών δεδομένων. Πρόκειται για ένα αρχείο τύπου κειμένου όπου τα αντικείμενα που περιέχει συντάσσονται στη μορφή "id, xmin, ymin, [zmin], xmax, ymax, [zmax]". Επίσης χρησιμοποιούνται σαν είσοδοι το μέγεθος της σελίδας εισόδου - εξόδου το οποίο τυπικά είναι 4096 bytes, το πλήθος διαστάσεων του χώρου στον οποίο ορίζονται τα πολύγωνα - αντικείμενα, το όνομα του ευρετηρίου που θα δημιουργηθεί και το όνομα της εικόνας του ευρετηρίου. Ως έξοδοι του αλγορίθμου δημιουργίας R-tree ευρετηρίου θα είναι το αρχείο .idx και η εικόνα του ευρετηρίου αυτού, για να φαίνεται και οπτικά η δομή του ευρετηρίου με τα αντικείμενα στο χώρο. Δεν θα αναλυθεί περισσότερο ο τρόπος δόμησης των



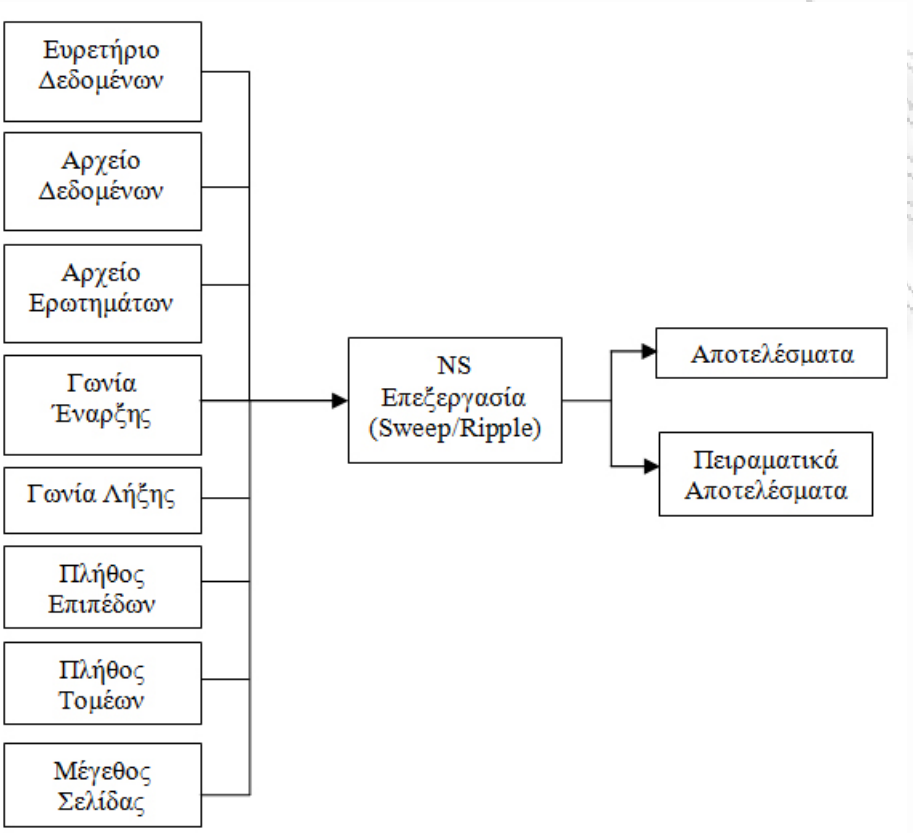
Εικόνα 4.1. Δόμηση των χωρικών δεδομένων σε R-tree.

αντικειμένων σε R-tree και οι λειτουργίες του R-tree γενικότερα καθώς δεν είναι αυτό που ενδιαφέρει στην παρούσα εργασία.

Στην εικόνα 4.2 βλέπουμε σε υψηλό επίπεδο τις εισόδους που απαιτούνται από τους αλγορίθμους Sweep και Ripple για την επεξεργασία των NS ερωτημάτων και τις εξόδους που παράγονται.

Οι είσοδοι που απαιτούνται για την επεξεργασία των NS ερωτημάτων είναι:

- Το ευρετήριο δεδομένων, είναι το αρχείο .idx που παρήγαγε ο αλγόριθμος δημιουργίας του R-tree ευρετηρίου που είδαμε στην εικόνα 4.1.
- Το αρχείο δεδομένων, είναι ένα αρχείο τύπου κειμένου που περιέχει τα αντικείμενα του χώρου ανάμεσα στα οποία θα αναζητηθούν τα NS. Τα αντικείμενα αυτά συντάσσονται στη μορφή "id #vertices x0 y0 x1 y1 ... xn yn". Όπως βλέπουμε δηλώνεται επίσης το πλήθος των κορυφών του πολυγώνου έτσι ώστε ο NS αλγόριθμος να ξέρει πόσες πλευρές θα αναγνώσει. Τα δεδομένα που θα χρησιμοποιηθούν για την εκτέλεση των πειραμάτων θα αφορούν σε ορθογώνια παραλληλόγραμμα άρα θα είναι πάντα τέσσερεις κορυφές.
- Το αρχείο των ερωτημάτων είναι ένα αρχείο κειμένου που περιέχει σε κάθε γραμμή ένα ερώτημα στη μορφή "id x y", όπου x, y είναι οι συντεταγμένες του σημείου ερωτήματος.
- Τη γωνία έναρξης και τη γωνία λήξης. Εξ' ορισμού αυτές οι γωνίες ορίζονται 0 και 360 μοίρες αλλά μπορούν να προσδιοριστούν από τον χρήστη στην περίπτωση που θέλει να ορίσει ένα μικρότερο γωνιακό εύρος για την αναζήτηση NS αντικειμένων.
- Το πλήθος των επιπέδων. Επειδή ο αλγόριθμος NS μπορεί να επεκταθεί για να δίνει αποτελέσματα σε περισσότερα του ενός επίπεδα, δίνει τη δυνατότητα στον χρήστη να επιλέξει το πλήθος των επιπέδων στα οποία επιθυμεί να



Εικόνα 4.2. Είσοδοι και έξοδοι κατά την επεξεργασία των NS ερωτημάτων.

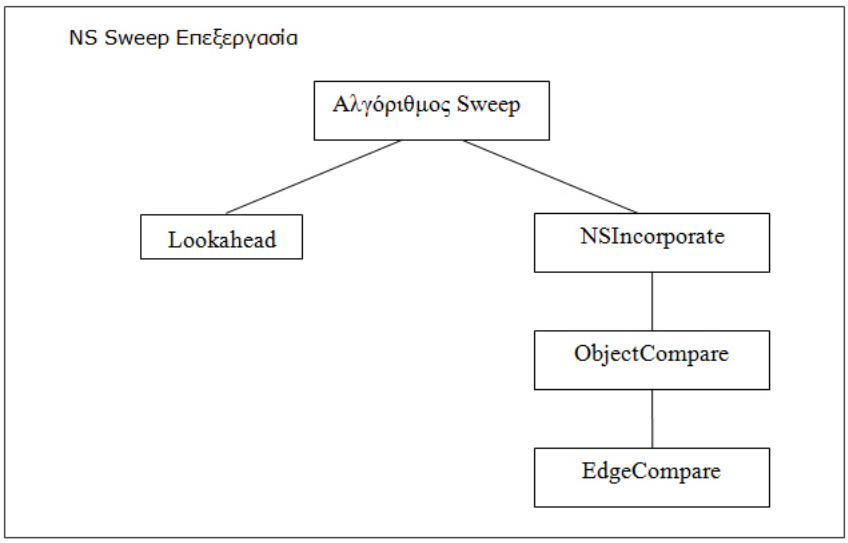
αναζητήσει NS αντικείμενα. Στο πρώτο επίπεδο αναζητάει τα αντικείμενα που είναι ορατά στο σημείο ερωτήματος. Στα επόμενα επίπεδα αναζητάει τα αντικείμενα που βρίσκονται πίσω από τα ορατά. Στην επέκταση των αλγορίθμων που θα επιχειρηθεί θα εξετάζεται μόνο το πρώτο επίπεδο των ορατών αντικειμένων και έτσι το πλήθος των επιπέδων θα είναι πάντα ένα.

- Το πλήθος των τομέων. Ο χρήστης μπορεί να ορίσει και το πλήθος των τομέων έτσι ώστε η αναζήτηση των NS αντικειμένων να γίνεται διαιρεμένη σε τομείς. Στην επέκταση των αλγορίθμων θα χρησιμοποιείται πάντα ένας τομέας που θα περιλαμβάνει όλο γωνιακό εύρος αναζήτησης.
- Το μέγεθος σελίδας μπορεί επίσης να οριστεί από τον χρήστη αλλά επειδή τυπικά είναι 4096 bytes, στα πειράματα που θα εκτελεστούν θα χρησιμοποιείται αυτό.

Ως έξοδο παίρνουμε ένα σύνολο δυάδων της μορφής $\langle o, [\alpha, \beta] \rangle$, όπου ο είναι το κοντινότερο αντικείμενο στη γωνία $[\alpha, \beta]$. Επίσης στην έξοδο παίρνουμε κάποια στοιχεία απόδοσης που μπορούν να χρησιμοποιηθούν για τη διεξαγωγή

των πειραμάτων. Αυτά είναι, ο χρόνος επεξεργασίας του ερωτήματος, ο αριθμός των αντικειμένων που εξετάστηκαν, ο αριθμός των εισόδων στην ουρά προτεραιότητας που μπορεί να χρησιμοποιηθεί για τη μέτρηση του κόστους μνήμης, το πλήθος των αντικειμένων του αποτελέσματος, το κόστος εισόδου - εξόδου.

Οι αλλαγές που θα κάνουμε θα πραγματοποιηθούν εντός της NS επεξεργασίας που στην εικόνα 4.2 αναπαρίσταται με το τετράγωνο «NS επεξεργασία (Sweep/Ripple)». Στις εικόνες 4.3 και 4.4 βλέπουμε πιο αναλυτικά τη διεργασία που πραγματοποιείται εντός αυτού του τετραγώνου για καθέναν από τους αλγορίθμους Sweep και Ripple. Συγκεκριμένα στην εικόνα 4.3 βλέπουμε τις βασικές συναρτήσεις οι οποίες αλληλεπιδρούν με τον αλγόριθμο Sweep.



Εικόνα 4.3. Αρχιτεκτονική συναρτήσεων NS Sweep επεξεργασίας.

Η συνάρτηση EdgeCompare κάνει σύγκριση μεταξύ δύο πλευρών αντικειμένων σε μια κοινή γωνία, ώστε να αποφασίσει ποια πλευρά είναι κοντινότερη στο σημείο ερωτήματος και την αντίστοιχη γωνία στην οποία αυτό συμβαίνει. Η συνάρτηση αυτή καλείται από τη συνάρτηση ObjectCompare. Η τελευταία αναλύει την εγγύτητα μεταξύ δύο αντικειμένων και επιστρέφει το κοντινότερο αντικείμενο με την αντίστοιχη γωνία του. Στην αρχή ορίζεται μια κοινή γωνία έξω από την οποία τα ανεξάρτητα αντικείμενα σε μη επικαλυπτόμενες γωνίες είναι μέρη του αποτελέσματος, εφόσον είναι ασύγκριτα. Στη συνέχεια καλείται η συνάρτηση EdgeCompare για να συγκρίνει τις πλευρές των αντικειμένων που έχουν κοινή γωνία. Η συνάρτηση NSIncorporate ανανεώνει το NS αποτέλεσμα. Εξάγει από το NS αποτέλεσμα τα αντικείμενα εκείνα των οποίων οι γωνίες διασταυρώνονται με το γωνιακό όριο του προς εξέταση αντικειμένου. Στη συνέχεια καλεί τη συνάρτηση ObjectCompare για να αποφασίσει τα κοντινότερα

αντικείμενα στις κατάλληλες γωνίες. Τελικά η NSIncorporate ανανεώνει το NS αποτέλεσμα με τα αντικείμενα που έχει βρει και τις αντίστοιχες γωνίες τους.

Η συνάρτηση Lookahead καλείται από τον αλγόριθμο Sweep, για να επιλέξει με βάση κάποιες ευρετικές μεθόδους την καλύτερη είσοδο στην ουρά προτεραιότητας. Σημειώνεται ότι στην ουρά προτεραιότητας, για τον αλγόριθμο Sweep, διατηρούνται κόμβοι και αντικείμενα του R-tree προς εξέταση, ταξινομημένα σύμφωνα με την αύξουσα γωνία έναρξης. Μια είσοδος της ουράς μπορεί να αγνοηθεί με ασφάλεια, όταν η ελάχιστη γωνιακή απόστασή της είναι μεγαλύτερη από τη μέγιστη γωνιακή απόσταση των NS αντικειμένων μέσα στην ίδια γωνία. Με βάση λοιπόν αυτή την ευρετική μέθοδο 1 και τη μέθοδο 2[1] που είδαμε στην ενότητα 2.4.3. η συνάρτηση Lookahead αναθεωρεί τη σειρά με την οποία ανακτούνται οι είσοδοι από την ουρά προτεραιότητας. Δεν αλλάζει τις θέσεις των εισόδων στην ουρά αλλά κάθε φορά που καλείται, εξετάζει το μπροστινό μέρος της ουράς για να επιλέξει μια είσοδο σύμφωνα με την αναθεωρημένη προτεραιότητα. Στη συνέχεια παρουσιάζεται ο ψευδοκώδικας του αλγορίθμου Sweep, όπου στη γραμμή 6 βλέπουμε να καλείται η συνάρτηση Lookahead.

Αλγόριθμος 4.1 SWEEP(q, root)

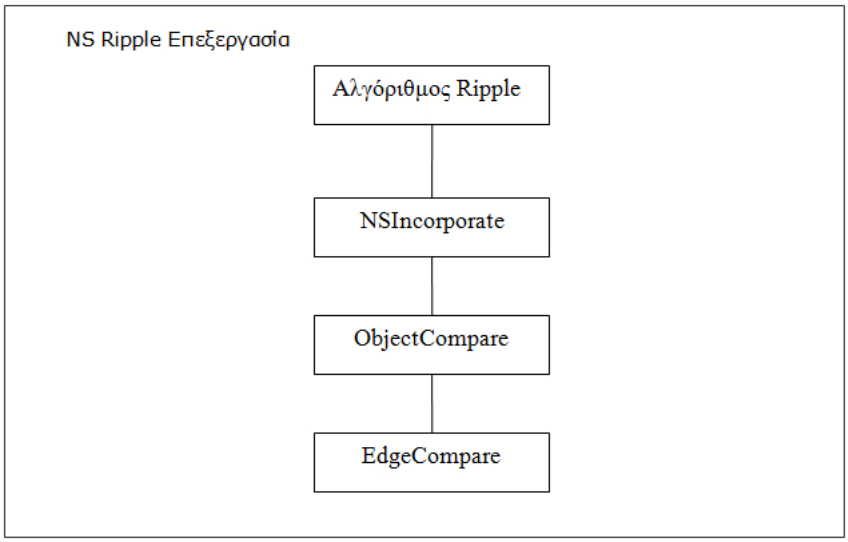
Input. a query point (q), and an R-tree root index node (root).
Output. An NS result, i.e. a set of ⟨object:angular range⟩ tuples.
1: Let Q be priority queue and be initialized with root;
2: Let α be max. ending angle of examined objects, initialized to 0;
3: Let N be the NS result and be initialized as $\{\langle \perp : [0, 2\pi] \rangle\}$;
4: **while** (Q is not empty)
5: Let ε be an element (it could be either a node or an object);
6: ε ← Lookahead(q, Q, [0, α]);
7: **if**$(min\_madist(q, \varepsilon, [\theta^{\vdash}_{q,\varepsilon}, \theta^{\dashv}_{q,\varepsilon}] > conservativeupperbound \wedge [\theta^{\vdash}_{q,\varepsilon}, \theta^{\dashv}_{q,\varepsilon}] \subseteq [0, \alpha])$
then skip;
8: **else**
9: **if** (ε is node) **then** explore ε and put all its child entries to Q;
10: **else**
11: N ← NSIncorporate(q, N, ε);
12: $\alpha \leftarrow max(\alpha, \theta^{\dashv}_{q,\varepsilon})$;
13: Output N;

Στον ψευδοκώδικα του Sweep στις γραμμές 1-3 βλέπουμε την αρχικοποίηση κάποιων παραμέτρων. Στην ουρά προτεραιότητας εισάγεται η ρίζα του R-tree. Η γωνία α αναπαριστά τη μέγιστη γωνία στην οποία τελειώνουν όλα τα αντικείμενα που έχουν εξεταστεί και είναι αρχικά 0. Το σύνολο των αποτελεσμά-

[1] Η μέθοδος 2 λέει ότι στη γωνία $[\theta^{\vdash}, \theta^{\dashv}]$ που καλύπτει όλα τα NS αντικείμενα του αποτελέσματος, ένα MBR R είναι πιθανότερο να περιέχει NS αντικείμενο αν έχει τη μικρότερη γωνιακή απόσταση στην κοινή γωνία $[\theta^{\vdash}, \theta^{\dashv}] \cap [\theta^{\vdash}_{q,R}, \theta^{\dashv}_{q,R}]$ ανάμεσα σε όλες τις άλλες εισόδους της ουράς.

των αρχικοποιείται στο $\{\langle \perp : [0, 2\pi] \rangle\}$. Στη γραμμή 4 η ουρά προτεραιότητας εξετάζεται αναδρομικά μέχρι να αδειάσει. Στη συνέχεια καλείται η συνάρτηση Lookahead για να επιστρέψει την καλύτερη είσοδο της ουράς. Η είσοδος η οποία έχει min_madist (όπου madist είναι η μικρότερη γωνιακή απόσταση) μεγαλύτερη από το συντηρητικό άνω όριο μπορεί να αγνοηθεί από την εξέταση. Το συντηρητικό άνω όριο ορίζεται ως το μέγιστο ανάμεσα στις max_madists από όλα τα αλληλεπικαλυπτόμενα NS αντικείμενα του μέχρι τώρα αποτελέσματος. Στη γραμμή 9 όταν η είσοδος είναι κόμβος τα παιδιά του εισάγονται πάλι στην ουρά για μετέπειτα εξέταση. Όταν η είσοδος είναι αντικείμενο καλείται η συνάρτηση NSIncorporate για να ανανεώσει το NS αποτέλεσμα. Η NSIncorporate εξετάζει το σύνολο των NS αποτελεσμάτων και εξάγει τα αντικείμενα εκείνα των οποίων οι γωνίες αλληλεπικαλύπτονται με το προς εξέταση αντικείμενο. Κατόπιν καλεί τη συνάρτηση ObjectCompare για να αποφασίσει το κοντινότερο αντικείμενο στην κοινή γωνία ή να επιστρέψει και τα δύο αντικείμενα με την αντίστοιχη διαιρεμένη γωνία.

Στην εικόνα 4.4 βλέπουμε τις βασικές συναρτήσεις οι οποίες αλληλεπιδρούν με τον αλγόριθμο Ripple.



Εικόνα 4.4. Αρχιτεκτονική συναρτήσεων NS Ripple επεξεργασίας.

Η διαφορά της NS Ripple επεξεργασίας είναι ότι δεν καλείται η συνάρτηση Lookahead για να επιστρέψει την καλύτερη είσοδο της ουράς σύμφωνα με κάποιες ευρετικές μεθόδους. Αυτό συμβαίνει διότι ο αλγόριθμος Ripple διατηρεί στην ουρά προτεραιότητας τις εισόδους, κόμβους ή αντικείμενα, σύμφωνα με την αύξουσα απόστασή τους από το σημείο ερωτήματος. Έτσι δεν χρειάζεται να αναθεωρηθεί η σειρά ανάκτησης των εισόδων από την ουρά προτεραιότητας.

Ταξινομώντας τις εισόδους σε αύξουσα απόσταση, η εξερεύνηση του Ripple πηγαίνει από το σημείο ερωτήματος προς τα έξω. Ο Ripple μπορεί να τερματίσει προτού η ουρά προτεραιότητας εξεταστεί πλήρως σύμφωνα με τη συνθήκη NS - TC που περιγράφτηκε στην ενότητα 2.2.4. Ο ψευδοκώδικας του αλγορίθμου Ripple είναι όμοιος με αυτόν του Sweep με τη διαφορά ότι δεν περιλαμβάνει την κλήση της συνάρτησης Lookahead[2].

Αλγόριθμος 4.2 RIPPLE(q, root)

Input. a query point (q), and an R-tree root index node (root).
Output. An NS result, i.e. a set of ⟨object:angular range⟩ tuples.
1: Let Q be priority queue and be initialized with root;
2: Let ε be entry of Q;
3: Let N be the NS result and be initialized as $\{\langle \perp : [0, 2\pi] \rangle\}$;
4: **while** (Q is not empty)
5: ε ← dequeue(Q);
6: *check_NS_TC*(ε);
7: **if**(ε is node) **then** explore ε and put all its child entries to Q;
8: **else**
9: N ← NSIncorporate(q, N, ε);
10: Output N;

Στη συνέχεια θα εντοπιστούν τα σημεία του κώδικα NS στα οποία πρέπει να επέμβουμε ώστε να πραγματοποιήσουμε την επέκταση του αλγορίθμου Ripple σε PostPruningkNS, δηλαδή τον PostPruning που επιστρέφει k κοντινότερους γείτονες με τις αντίστοιχες γωνίες τους. Επιπλέον θα εντοπίσουμε τα σημεία για την επέκταση του αλγόριθμου Ripple σε PrePruningkNS, δηλαδή ο βελτιστοποιημένος αλγόριθμος PostPruning, που επιστρέφει k κοντινότερους γείτονες με τις αντίστοιχες γωνίες τους. Επίσης θα δούμε την επέκταση του αλγόριθμου Sweep σε SweepkNS, δηλαδή των αλγόριθμο Sweep που επιστρέφει k NS.

4.2 Επέκταση Αλγορίθμου Sweep

Όπως διευκρινίστηκε η τροποποίηση που θα γίνει στον αλγόριθμο NS ώστε να επεκταθεί είναι να προστεθεί η παράμετρος k, δηλαδή το πλήθος των NS αντικειμένων που επιθυμεί να αναζητήσει ο χρήστης. Από τους δύο αλγορίθμους επιλέγεται τυχαία ο αλγόριθμος Sweep, ώστε να επεκτείνουμε τα NS ερωτήματα σε kNS. Αυτό συνεπάγεται ότι στην επεξεργασία των NS ερωτημάτων όπως

[2] Όπως έχει αναφερθεί η κύρια διαφορά του αλγορίθμου Ripple από το Sweep είναι ότι ο Ripple εισάγει τα δεδομένα (αντικείμενα ή κόμβοι) του R-tree, στην ουρά προτεραιότητας σύμφωνα με την αύξουσα απόστασή τους από το σημείο ερωτήματος, ενώ ο Sweep σύμφωνα με την αύξουσα γωνία τους.

φαίνεται στην εικόνα 4.2 θα πρέπει να προστεθεί άλλη μια είσοδος, η παράμετρος k. Επιπλέον αλλαγές θα πρέπει να γίνουν και μέσα στην επεξεργασία των NS ερωτημάτων.

Το πρόβλημα με τον αλγόριθμο Sweep ως προς την επιλογή k NS, είναι ότι η ουρά προτεραιότητας διατηρεί τις εισόδους του R-tree σύμφωνα με την αύξουσα γωνία έναρξης και έτσι οι είσοδοι ανακτούνται με αυτή την ταξινόμηση. Συνεπώς αν θέλουμε να αναζητήσουμε τους k κοντινότερους - στο σημείο ερωτήματος - γείτονες με τις αντίστοιχες γωνίες τους, δεν μπορούμε να επέμβουμε εντός του αλγορίθμου Sweep και σταδιακά να σταματήσουμε την αναζήτηση όταν έχουν ανακτηθεί k NS. Αν γινόταν κάτι τέτοιο ο αλγόριθμος θα επέστρεφε σαν αποτέλεσμα τους k NS σε ένα συνεχόμενο γωνιακό εύρος, από τη γωνία έναρξης που ορίζει ο χρήστης και μέχρι τη γωνία που θα βρει k αντικείμενα, χωρίς να εξετάσει το υπόλοιπο γωνιακό εύρος στο οποίο θα μπορούσαν να βρεθούν κοντινότεροι γείτονες. Συνεπώς για να επιστρέψει ο Sweep k NS, θα πρέπει να τρέξει ο αλγόριθμος Sweep, να δώσει αποτελέσματα και στη συνέχεια να ταξινομήσουμε αυτά τα αποτελέσματα σε αύξουσα απόσταση από το σημείο ερωτήματος και να πάρουμε τα k αντικείμενα ως τελικό αποτέλεσμα.

4.3 Επέκταση Αλγορίθμου Ripple

Στην ενότητα 3.3.1 είδαμε ποιες τροποποιήσεις απαιτούνται για να δημιουργηθεί ο αλγόριθμος PostPruning NS - TC από τον αλγόριθμο NS Ripple. Στη συνέχεια θα δούμε σε ποια σημεία του κώδικα NS Ripple θα πρέπει να επέμβουμε ώστε να πραγματοποιήσουμε αυτές τις αλλαγές.

'Οπως βλέπουμε από τον ψευδοκώδικα (γραμμή 5 - 6) του αλγορίθμου PostPruning, όταν η είσοδος που ανακτάται από την ουρά προτεραιότητας είναι αντικείμενο υπολογίζεται η ελάχιστη ορατή απόστασή του. 'Ετσι επεμβαίνουμε στον αλγόριθμο Ripple, όταν η είσοδος που ανακτάται είναι αντικείμενο και πριν κληθεί η συνάρτηση που ανανεώνει το NS αποτέλεσμα (συνάρτηση NSIncorporate), δηλαδή ανάμεσα στις γραμμές 8 και 9 του ψευδοκώδικα του Ripple. Σε αυτό το σημείο θα πρέπει να υλοποιηθεί και να κληθεί μια συνάρτηση που να υπολογίζει την ελάχιστη ορατή απόσταση του αντικειμένου. Κατόπιν αυτή η ορατή απόσταση που θα επιστρέφεται θα συγκρίνεται με την απόσταση της εισόδου στην κορυφή της ουράς. Σημειώνεται ότι οι είσοδοι εισάγονται στην ουρά σε αύξουσα απόσταση από το σημείο ερωτήματος. Αν το αντικείμενο έχει μικρότερη ορατή απόσταση από την κορυφή της ουράς, τότε καλείται η συνάρτηση NSIncorporate καθώς έχει βρεθεί ένα νέο NS αντικείμενο. Διαφορετικά αν το προς εξέταση αντικείμενο έχει μεγαλύτερη ορατή απόσταση και όχι άπειρη, δηλαδή δεν είναι μη ορατό, επανεισάγεται στην ουρά προτεραιότητας με απόσταση την ελάχιστη ορατή που υπολογίστηκε.

Θα πρέπει ακόμα να εισάγουμε στον Ripple και την παράμετρο k. Μπορεί να προστεθεί στον κώδικα ένας έλεγχος ώστε να καλείται η συνάρτηση NSIncorporate μόνο εφόσον το πλήθος των αντικειμένων στο NS αποτέλεσμα είναι μικρότερο ή ίσο του k.

Η διαδικασία που περιγράφηκε είναι ο κώδικας που πρέπει να υλοποιηθεί στο σημείο του Ripple πριν κληθεί η συνάρτηση NSIncorporate, ώστε να δημιουργηθεί ο αλγόριθμος PostPruning NS - TC.

Η διαφορά του αλγορίθμου PostPruning από τον αλγόριθμο PrePruning είναι ότι ο τελευταίος ελέγχει την ελάχιστη ορατή απόσταση όχι μόνο των αντικειμένων αλλά και των κόμβων του R-tree. Δηλαδή οι είσοδοι κλαδεύονται σύμφωνα με την ορατότητα αμέσως μόλις ανακτούνται από την ουρά προτεραιότητας και όχι όταν εντοπιστεί κάποιο αντικείμενο, όπως συμβαίνει στον PostPruning. Αυτό φαίνεται και από τη γραμμή 5 του ψευδοκώδικα PrePruning που είδαμε στο κεφάλαιο 3. Στη συνέχεια, όπως και στον αλγόριθμο PostPruning, ελέγχεται αν αυτή η ελάχιστη ορατή απόσταση που υπολογίστηκε είναι μεγαλύτερη από την απόσταση της εισόδου στην κορυφή της ουράς. Αν αυτό ισχύει και επιπλέον η είσοδος της ουράς που εξετάζεται δεν έχει άπειρη ορατή απόσταση, τότε επανεισάγεται στην ουρά με απόσταση την ελάχιστη ορατή που υπολογίστηκε (γραμμές 6-8 του ψευδοκώδικα PrePruning). Αλλιώς αν η ελάχιστη ορατή απόσταση της εισόδου που εξετάζεται είναι μικρότερη από την απόσταση της κορυφής της ουράς, τότε συνεχίζεται ο έλεγχος του αλγορίθμου για το αν η είσοδος είναι κόμβος ή αντικείμενο κλπ., δηλαδή σχεδόν ο ίδιος έλεγχος που κάνει και ο Ripple.

Έτσι για να υλοποιήσουμε τον αλγόριθμο PrePruningNS θα πρέπει να επέμβουμε ανάμεσα στις γραμμές 6 και 7 του ψευδοκώδικα του Ripple. Δηλαδή ακριβώς πριν γίνει ο έλεγχος για το εάν η είσοδος που αφαιρέθηκε από την ουρά προτεραιότητας είναι κόμβος ή αντικείμενο.

Για να προστεθεί η παράμετρος k θα γίνει ο ίδιος έλεγχος με αυτόν που έγινε και για τον αλγόριθμος PostPruningkNS. Με αυτές τις τροποποιήσεις και διατηρώντας την ιδιότητα της γωνίας και τη συνθήκη τερματισμού NS-TC που έχει ο αλγόριθμος Ripple, θα προκύψει ο αλγόριθμος PrePruningkNS.

5

Ανάλυση των Νέων Υποσυστημάτων

Από τις τροποποιήσεις στον NS κώδικα που περιγράφτηκαν στο προηγούμενο κεφάλαιο θα δημιουργηθεί ο αλγόριθμος PostPruning-NS-TC, ο οποίος θα επιστρέφει k NS με τις αντίστοιχες γωνίες τους και εν συντομία θα ονομάζεται PostPruningkNS. Είναι εύκολο να δημιουργηθεί και η εκδοχή του αλγορίθμου PostPruning που επιστρέφει όλους του NS και όχι μόνο τους k, εν συντομία PostPruningNS. Αλλάζοντας την θέση που γίνεται ο έλεγχος της ελάχιστης ορατής απόστασης, δημιουργείται ο αλγόριθμος PrePruning-NS-TC, που επιστρέφει k NS με τις αντίστοιχες γωνίες τους ή αλλιώς ο PrePruningkNS. Ομοίως δημιουργείται και η εκδοχή που επιστρέφει όλους τους NS γείτονες, δηλαδή ο PrePruningNS. Επιπλέον είπαμε ότι θα επεκτείνουμε τον αλγόριθμο Sweep ώστε να επιστρέφει k NS, ο οποίος θα ονομάζεται SweepkNS, ενώ ο αλγόριθμος Sweep που επιστρέφει όλους τους κοντινότερους γείτονες με τις αντίστοιχες γωνίες τους υπάρχει έτοιμος από τους συγγραφείς που μας έδωσαν τον κώδικα. Έτσι καταλήγουμε με πέντε αλγορίθμους οι οποίοι δημιουργήθηκαν επεκτείνοντας τους υπάρχοντες κώδικες των αλγορίθμων Ripple και Sweep. Συνοπτικά οι αλγόριθμοι αυτοί είναι:

- PostPruningkNS
- PostPruningNS
- PrePruningkNS
- PrePruningNS
- SweepkNS

5.1 Επέκταση του Ripple σε PostPruning(k)NS

Στο κεφάλαιο της αρχιτεκτονικής κατά την NS Ripple επεξεργασία των ερωτημάτων είδαμε ποιες συναρτήσεις καλεί ο αλγόριθμος Ripple και τι κάνει κάθε μια από αυτές τις συναρτήσεις. Σε αυτό το σημείο θα εστιάσουμε στον αλγόριθμο Ripple καθαυτό. Ο κώδικας που μας έδωσαν οι συγγραφείς του άρθρου

[3] είναι γραμμένος σε C++, έτσι οι επεκτάσεις έγιναν σε αυτή τη γλώσσα προγραμματισμού.

Ο αλγόριθμος του Ripple ξεκινάει εξάγοντας από την ουρά προτεραιότητας τη ρίζα του R-tree, εφόσον η ουρά αρχικοποιείται με τη ρίζα του R-tree ευρετηρίου. Στη συνέχεια κάνει βασικά δύο πράγματα, ελέγχει αν η είσοδος που εξήχθη από την ουρά είναι κόμβος ή αντικείμενο. Ότι δηλαδή κάνει και ο αλγόριθμος PostPruning που θέλουμε να δημιουργήσουμε. Αν είναι κόμβος τότε και οι δύο αλγόριθμοι (Ripple και PostPruning) εξερευνούν τα παιδιά του και τα εισάγουν στην ουρά. Έτσι η ουρά προτεραιότητας γεμίζει σταδιακά με κόμβους και τελικά αντικείμενα. Αν όμως η είσοδος είναι αντικείμενο τότε ο αλγόριθμος PostPruning κάνει κάτι διαφορετικό από τον Ripple. Ο Ripple θεωρεί ότι έχει βρει έναν υποψήφιο κοντινότερο γείτονα και καλεί την συνάρτηση που ανανεώνει το NS αποτέλεσμα. Ο PostPruning από την άλλη πλευρά κάνει έναν επιπλέον έλεγχο πριν θεωρήσει το αντικείμενο υποψήφιο για να μπει στο σύνολο των αποτελεσμάτων. Ελέγχει την ελάχιστη ορατή απόσταση του. Μια έννοια που δεν υπάρχει στον αλγόριθμο Ripple. Επομένως θα πρέπει πριν ο Ripple καλέσει τη συνάρτηση που ανανεώνει το NS αποτέλεσμα, να δημιουργηθεί και να κληθεί μια συνάρτηση που υπολογίζει την ελάχιστη ορατή απόσταση.

Πριν αναλυθεί η λογική με την οποία υπολογίζεται η ελάχιστη ορατή απόσταση να διευκρινιστεί ότι ο Ripple ανανεώνει το NS αποτέλεσμα με βάση τις ακμές των αντικειμένων και όχι τα αντικείμενα καθαυτά. Αυτό σημαίνει ότι όταν καθώς διατρέχει το R-tree φτάσει σε κάποιο αντικείμενο τότε εισάγει τις ακμές του αντικειμένου στην ουρά προτεραιότητας. Όταν ανασύρει από την ουρά κάποια ακμή τότε καλεί τη συνάρτηση NSIncorporate η οποία καλεί την ObjectCompare και αυτή με τη σειρά της την EdgeCompare, δηλαδή τη συνάρτηση που συγκρίνει τις ακμές ανά δύο, για να αποφασίσει ποια είναι η κοντινότερη. Από την άλλη πλευρά στο άρθρο [6] για τον υπολογισμό της ελάχιστης ορατής απόστασης των αλγορίθμων PostPruning και PrePruning, στην ενότητα «προεισαγωγικά», διευκρινίζεται ότι χρησιμοποιείται η συνάρτηση CLIP που βασίζεται στο άρθρο [9]. Σε αυτό το άρθρο το εξέταση δύο πολυγώνων γίνεται διχοτομώντας το χώρο σύμφωνα με τις y συντεταγμένες των κορυφών των πολυγώνων. Για κάθε μέρος του χώρου σχηματίζεται ένα μερικό περίγραμμα του αποτελέσματος, ελέγχοντας για πιθανές τομές ανάμεσα στα δύο πολύγωνα. Εφόσον έχουν επεξεργαστεί όλα τα μέρη του χώρου, ανακτάται ολόκληρο το πολύγωνο του αποτελέσματος, χωρίς να απαιτείται μεταεπεξεργασία, όπως για παράδειγμα η ταξινόμηση των ακμών. Στην περίπτωσή μας για την υλοποίηση του αλγορίθμου PostPruningkNS αλλά και των συναφών PostPruningNS, PrePruningkNS, PrePruning, θα χρησιμοποιήσουμε την ταξινόμηση των ακμών όπως γίνεται και στον αλγόριθμο Ripple.

Είναι εύκολο τώρα να περιγραφεί η λογική της υλοποίησης της συνάρτησης calcMinViDist της οποίας ο κώδικας φαίνεται στο Παράρτημα Α. Η συνάρτηση αυτή παίρνει σαν ορίσματα το σύνολο του τρέχοντος αποτελέσματος, το σημείο ερωτήματος, το id του πολυγώνου, την ακμή της οποίας την ελάχιστη ορατή απόσταση θέλουμε να υπολογίσουμε και το γωνιακό εύρος αυτής της

ακμής. Τελικά επιστρέφεται η ελάχιστη ορατή απόσταση αυτής της ακμής και το αντίστοιχο γωνιακό εύρος στο οποίο είναι ορατή από το σημείο ερωτήματος.

Η λογική της συνάρτησης calcMinViDist βασίζεται στην παρατήρηση ότι μόνο τα αντικείμενα που είναι κοντινότερα στο σημείο ερωτήματος μπορούν να επηρεάσουν την ορατότητα του προς εξέταση αντικειμένου. Επιπλέον τα αντικείμενα που είναι κοντινότερα στο σημείο ερωτήματος από το προς εξέταση αντικείμενο βρίσκονται ήδη στο τρέχον σύνολο των αποτελεσμάτων. Έτσι δεν απαιτείται να συγκρίνουμε με τις αποστάσεις των αντικειμένων όλου του συνόλου των πολυγώνων, αλλά μόνο αυτών που βρίσκονται στο τρέχον αποτέλεσμα.

Περιγράφοντας τη συνάρτηση με λόγια θα λέγαμε τα εξής: Αν το αντικείμενο που εξετάζεται είναι το πρώτο του αποτελέσματος, τότε έχει ελάχιστη ορατή απόσταση ίση την ελάχιστη απόστασή του από το σημείο ερωτήματος, διότι κανένα άλλο αντικείμενο δεν έχει μικρότερη απόσταση από αυτό. Έτσι το εισάγουμε στο σύνολο των αποτελεσμάτων. Διαφορετικά αν δεν είναι το πρώτο υποψήφιο αντικείμενο για το αποτέλεσμα, τότε για όλα τα μέχρι τώρα αντικείμενα του αποτελέσματος ελέγχω αν έχουν κοινή γωνία. Αν ναι, αφαιρώ την κοινή γωνία από το γωνιακό εύρος του προς εξέταση αντικειμένου. Διότι στην κοινή τους γωνία το αντικείμενο που βρίσκεται ήδη στο αποτέλεσμα είναι κοντινότερο και επομένως σε αυτή τη γωνία κρύβει το προς εξέταση αντικείμενο. Επαναλαμβάνοντας αυτή τη διαδικασία για όλα τα αντικείμενα του τρέχοντος αποτελέσματος μένει στο τέλος το ορατό γωνιακό εύρος του προς εξέταση αντικειμένου. Μπορούμε τώρα να υπολογίσουμε την ελάχιστη απόσταση του αντικειμένου σε αυτό το γωνιακό εύρος. Με αυτό τον τρόπο θα προκύψει η ελάχιστη ορατή απόσταση.

Επιστρέφουμε από τη συνάρτηση calcMinViDist και σύμφωνα με τον αλγόριθμο PostPruning, θα πρέπει να συγκρίνουμε αυτή την ελάχιστη ορατή απόσταση με την απόσταση της εισόδου που βρίσκεται στην κορυφή της ουράς προτεραιότητας. Αν το αντικείμενο που εξετάζεται έχει ελάχιστη ορατή απόσταση μικρότερη από την είσοδο στην κεφαλή της ουράς τότε το αντικείμενο αυτό εισάγεται στο σύνολο των αποτελεσμάτων. Διαφορετικά εφόσον το αντικείμενο που εξετάζεται δεν έχει άπειρη ελάχιστη ορατή απόσταση, δηλαδή δεν είναι μη ορατό, τότε επανεισάγεται στην ουρά με απόσταση την ελάχιστη ορατή που υπολογίστηκε. Συνεπώς μετά την κλήση της συνάρτησης calcMinViDist και πριν κληθεί η συνάρτηση NSIncorporate υλοποιείται ο παραπάνω έλεγχος.

Με όλες αυτές τις αλλαγές έχουμε δημιουργήσει τον αλγόριθμο PostPruningNS ο οποίος στον κώδικα υλοποιείται με τη συνάρτηση postpruning της κλάσης nspostpruning.cc, όπως φαίνεται και στο Παράρτημα Α. Απομένει να δοθεί η δυνατότητα στον χρήστη να επιλέγει το πλήθος των κοντινότερων γειτόνων. Εισάγεται στη συνάρτηση postpruning ένα ακόμη όρισμα, ο ακέραιος k, ο οποίος είναι ένας αριθμός που έχει αναγνωστεί από τον κώδικα ως είσοδος από τον χρήστη. Για να υλοποιηθεί ο έλεγχος του k ελέγχουμε το πλήθος των αντικειμένων που είναι στο αποτέλεσμα. Είναι δυνατόν να γίνει κάτι τέτοιο εφόσον ο αλγόριθμος Ripple που τροποποιούμε ανακτά τους κοντινότερους γείτονες σταδιακά, κατά αύξουσα απόσταση από το σημείο ερωτήματος. Επομένως μπορούμε να ἐπούμεη στον Ripple να σταματήσει όταν θα έχει ανακτήσει k γεί-

τονες. Ο έλεγχος αυτός προστέθηκε πριν κληθεί η συνάρτηση calcMinViDist, ο έλεγχος της ελάχιστης ορατής απόστασης σε σχέση με την ελάχιστη απόσταση της κεφαλής της ουράς και η ανανέωση του αποτελέσματος.

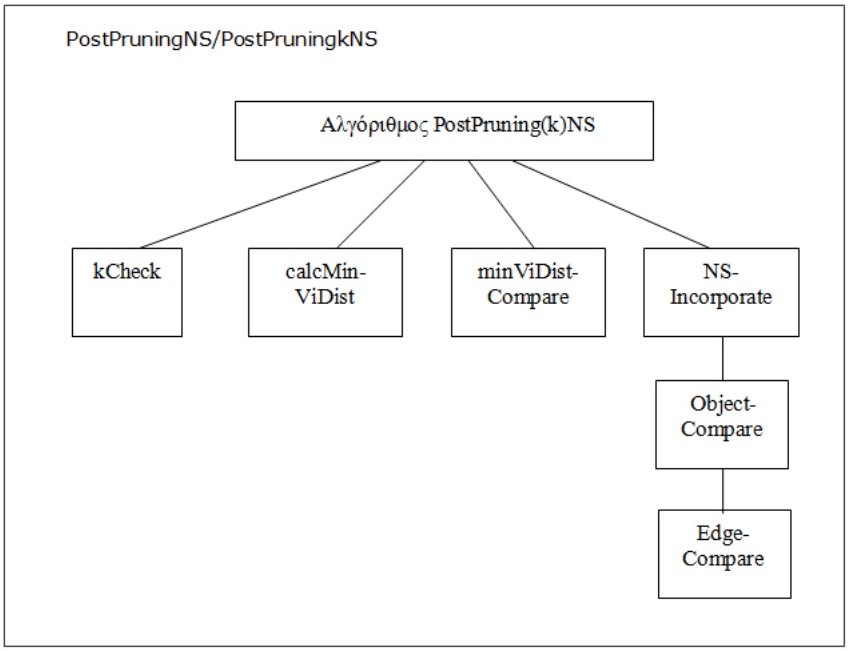
Όπως είδαμε και πιο πάνω σε αυτή την ενότητα στο αποτέλεσμα αποθηκεύονται ακμές πολυγώνων και όχι ολόκληρα πολύγωνα. Για αυτό το λόγο δεν μπορούμε να πούμε στον αλγόριθμο να σταματήσει εφόσον έχει k ακμές στο αποτέλεσμα. Διότι στο αποτέλεσμα μπορεί να περιέχονται δύο ακμές του ίδιου παραλληλογράμμου (ενώ ο χρήστης έχει ζητήσει k παραλληλόγραμμα). Εφόσον μέχρι δύο ακμές ενός ορθογωνίου παραλληλογράμμου μπορεί να είναι ορατές από ένα σημείο ερωτήματος. Έτσι δημιουργήθηκε η ανάγκη να υλοποιηθούν δύο ακόμα μικρές συναρτήσεις που ελέγχουν τα αντικείμενα του αποτελέσματος. Η συνάρτηση existsInResult ελέγχει αν ένα αντικείμενο υπάρχει ήδη στο τρέχον αποτέλεσμα. Η συνάρτηση noOfObjsInResult επιστρέφει τον αριθμό των αντικειμένων που βρίσκεται στο τρέχον αποτέλεσμα. Η συνάρτηση noOfObjsInResult δεν αρκεί διότι στο τελευταίο k αντικείμενο που επιχειρείται να προστεθεί στο αποτέλεσμα και ενώ το σύνολο αποτελεσμάτων έχει ακμές από k διαφορετικά αντικείμενα, μπορεί ο αλγόριθμος να ελέγχει μια άλλη ακμή ενός από τα αντικείμενα που βρίσκονται ήδη στο αποτέλεσμα. Σε αυτή την περίπτωση δεν μιλάμε για το k+1 αντικείμενο αλλά για το k, και επομένως χρησιμοποιείται η συνάρτηση existsInResult ώστε να ελέγξει αν αυτό το αντικείμενο έχει ήδη ακμή μέσα στο σύνολο των αποτελεσμάτων. Συνεπώς προκύπτει ο αλγόριθμος PostPruningkNS.

Στην εικόνα 5.1 βλέπουμε την αρχιτεκτονική της PostPruning kNS επεξεργασίας που υλοποιήσαμε.

Παρατηρούμε ότι σε αντίθεση με τις αρχιτεκτονικές των πρακτικών Sweep και Ripple καλείται από τον αλγόριθμο PostPruningkNS μια ακόμα βασική συνάρτηση, η calcMinViDist, ανεξάρτητη από τις υπόλοιπες βασικές συναρτήσεις. Ως minViDistCompare αναφέρεται η διαδικασία που συγκρίνει την ελάχιστη ορατή απόσταση ενός αντικειμένου με την ελάχιστη απόσταση της κεφαλής της ουράς και επανεισάγει το αντικείμενο στην ουρά, εάν η τελευταία απόσταση είναι μικρότερη. Για αυτή τη διαδικασία δεν έχει δημιουργηθεί ξεχωριστή συνάρτηση, αλλά έχει ενσωματωθεί στον αλγόριθμο PostPruning(k)NS. Παρουσιάζεται στην εικόνα 5.1 διότι είναι μια σημαντική διαδικασία που δεν υπάρχει στους αλγορίθμους Ripple και Sweep. Ούτε και η διαδικασία που ελέγχει αν υπάρχουν k αντικείμενα στο αποτέλεσμα υπάρχει στους αλγορίθμους Ripple και Sweep. Αυτή η διαδικασία ονομάζεται στην εικόνα 5.1 kCheck και χρησιμοποιείται μόνο από τον αλγόριθμο PostPruningkNS.

Η διαδικασία minViDistCompare υλοποιείται αμέσως μετά την κλήση της συνάρτησης calcMinViDist και μετά ανάλογα με το αποτέλεσμα της διαδικασίας minViDistCompare καλείται η συνάρτηση NSIncorporate και οι ακόλουθες σε αυτή συναρτήσεις. Όλα αυτά γίνονται εφόσον η διαδικασία kCheck έχει βρει λιγότερα ή ίσα με k αντικείμενα στο αποτέλεσμα.

Μέχρι αυτό το σημείο έχουμε επεκτείνει τη μια εκδοχή, τον αλγόριθμο PostPruning, των VkNN ερωτημάτων, ώστε να καλύπτει επιπλέον την κατεύ-



Εικόνα 5.1. Αρχιτεκτονική συναρτήσεων PostPruning(k)NS επεξεργασίας.

θυνση. Στην επόμενη ενότητα θα δούμε και την επέκταση της άλλης εκδοχής, του αλγορίθμου PrePruning.

5.2 Επέκταση του Ripple σε PrePruning(k)NS

Όπως είδαμε σε προηγούμενο κεφάλαιο η διαφορά του PrePruning από τον PostPruning είναι ότι ο PrePruning ελέγχει την ελάχιστη ορατή απόσταση όχι μόνο των αντικειμένων αλλά και των κόμβων. Αυτό επιτυγχάνεται όταν ο έλεγχος αυτός γίνεται μόλις εξάγεται ένα στοιχείο της ουράς προτεραιότητας, είτε αυτό είναι κόμβος είτε αντικείμενο. Δηλαδή αλλάζοντας μέσα στον αλγόριθμο του PostPruningNS την θέση στην οποία γίνεται ο έλεγχος της ελάχιστης ορατής απόστασης και την επανείσοδο στην ουρά αν αυτή η απόσταση είναι μεγαλύτερη από την απόσταση της κεφαλής της ουράς, επιτυγχάνουμε τη δημιουργία του αλγορίθμου PrePruningNS.

Για την υλοποίηση του αλγορίθμου PostPruningNS δημιουργήθηκε μια συνάρτηση που υπολογίζει την ελάχιστη ορατή απόσταση των ακμών. Καθώς διευκρινίστηκε ότι γίνεται σύγκριση ακμών και όχι πολυγώνων και το NS αποτέλεσμα ανανεώνεται με τις ακμές των αντικειμένων. Στην περίπτωση του PrePruningNS όταν εξάγεται ένα στοιχείο από την ουρά αυτό μπορεί να είναι ακμή ή κόμβος. Αν είναι κόμβος, δηλαδή ουσιαστικά είναι ένα ελάχιστο ορθογώ-

νιο οριοθέτησης (MBR), το οποίο περικλείει αντικείμενα ή/και άλλα ελάχιστα ορθογώνια οριοθέτησης. Συνεπώς επιδιώκουμε να υπολογίσουμε την ελάχιστη ορατή απόσταση για ένα ορθογώνιο που αποτελείται από τέσσερεις ακμές, σε αντίθεση με τον αλγόριθμο PostPruningNS που είχαμε να υπολογίσουμε την ελάχιστη ορατή απόσταση για μια μόνο ακμή.

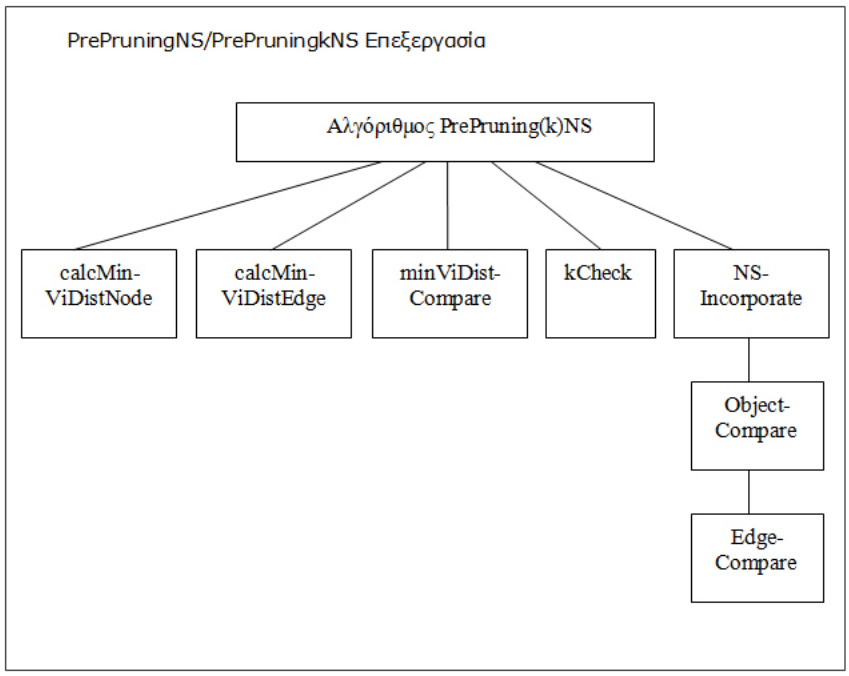
Υλοποιείται μια νέα συνάρτηση, η calcMinViDistNode που βλέπουμε στο Παράρτημα Α, που υπολογίζει την ελάχιστη ορατή απόσταση ενός κόμβου. Η συνάρτηση αυτή στα ορίσματά της αντί για μια ακμή, παίρνει έναν υπερκύβο ή αλλιώς κόμβο του R-tree. Η λογική με την οποία αυτή η συνάρτηση υπολογίζει την ελάχιστη ορατή απόσταση είναι ίδια με την λογική της συνάρτησης calcMinViDist που είδαμε στην προηγούμενη ενότητα. Δηλαδή ο στόχος είναι να αφαιρεθούν από το γωνιακό εύρος του κόμβου οι γωνίες εκείνες στις οποίες είναι μη ορατό, ούτως ώστε στο τέλος να απομείνει μόνο το γωνιακό εύρος του κόμβου στο οποίο είναι ορατό από το σημείο ερωτήματος. Αυτό γίνεται ελέγχοντας τις τομές της γωνίας του κόμβου που εξετάζεται με τις γωνίες όλων των αντικείμενων του τρέχοντος αποτελέσματος. Διότι μόνο τα αντικείμενα του τρέχοντος αποτελέσματος είναι πιο κοντά στο σημείο ερωτήματος από τον κόμβο που εξετάζεται και επομένως μόνο αυτά μπορούν να επηρεάσουν την ορατότητά του. Εφόσον υπολογιστεί η ορατή γωνία, υπολογίζεται κατόπιν η ελάχιστη απόσταση του κόμβου σε αυτή την ορατή γωνία.

Όπως, όμως, αναφέρθηκε ο κόμβος αποτελείται από τέσσερεις ακμές. Προέκυψε η ανάγκη δημιουργίας μιας συνάρτησης που να υπολογίζει την ελάχιστη απόσταση ενός κόμβου σε συγκεκριμένη γωνία. Στον κώδικα NS υπήρχε η συνάρτηση που υπολογίζει την ελάχιστη απόσταση μιας ακμής σε συγκεκριμένο γωνιακό εύρος, όμως δεν υπήρχε αντίστοιχη συνάρτηση για τον κόμβο. Από την συνάρτηση calcMinViDistNode, όταν έχει υπολογιστεί η ορατή γωνία, καλούμε τη συνάρτηση mindistHC που υπολογίζει την ελάχιστη απόσταση ενός υπερκύβου σε κάποια γωνία. Η συνάρτηση αυτή εκμεταλλεύεται την υπάρχουσα συνάρτηση που υπολογίζει την ελάχιστη απόσταση μιας ακμής σε κάποια γωνία. Η mindistHC παίρνει τις συντεταγμένες των κορυφών του κόμβου και για κάθε δύο σημεία δημιουργεί μια ακμή. Στη συνέχεια καλεί τη συνάρτηση που υπολογίζει την ελάχιστη απόσταση της ακμής στην ορατή γωνία που της έχει δοθεί ως όρισμα. Αυτό γίνεται για όλες τις ακμές του κόμβου και στο τέλος επιστρέφεται η ελάχιστη απόσταση του κόμβου στο συγκεκριμένο γωνιακό εύρος που δόθηκε ως όρισμα.

Με τις παραπάνω αλλαγές - προσθήκες στον κώδικα Ripple ή αλλιώς παραλλάσσοντας τον συγγενή κώδικα PostPruningNS έχουμε δημιουργήσει τον αλγόριθμο PrePruningNS. Για την παραλλαγή PrePruningkNS ακολουθείται όμοια διαδικασία όπως στον PostPruningkNS. Για να έχει τη δυνατότητα ο χρήστης να επιλέξει το πλήθος των κοντινότερων γειτόνων που επιθυμεί να του επιστρέψει ο αλγόριθμος, ελέγχουμε το πλήθος των διαφορετικών αντικειμένων στο αποτέλεσμα. Αυτό γίνεται ακριβώς πριν κληθεί η συνάρτηση που ανανεώνει το NS αποτέλεσμα. Έτσι ανακτώνται σταδιακά k κοντινότεροι γείτονες. Όπως και στον αλγόριθμο PostPruningkNS επειδή στο αποτέλεσμα αποθηκεύονται ακμές και όχι πολύγωνα, χρησιμοποιούνται οι συναρτήσεις noOfObjsInResult

και existsInResult για να υπολογιστεί το πλήθος των διαφορετικών αντικειμένων στο αποτέλεσμα.

Η αρχιτεκτονική της PrePruning kNS επεξεργασίας που υλοποιήσαμε φαίνεται στην εικόνα 5.2.



Εικόνα 5.2. Αρχιτεκτονική συναρτήσεων PrePruning(k)NS επεξεργασίας.

Στην αρχιτεκτονική της PrePruning(k)NS επεξεργασίας χρειάστηκε να διακρίνουμε τη συνάρτηση υπολογισμού της ελάχιστης ορατής απόστασης σε δύο συναρτήσεις. Γι' αυτό στην εικόνα 5.2 υπάρχουν δύο ανεξάρτητες συναρτήσεις για τον υπολογισμό της ελάχιστης ορατής απόστασης. Η μία αναφέρεται στις ακμές και η άλλη στους κόμβους. Επιπλέον υπάρχει η διαδικασία minViDistCompare που συγκρίνει την ελάχιστη ορατή απόσταση των κόμβων ή των ακμών με την ελάχιστη απόσταση της κεφαλής της ουράς. Η διαδικασία minViDistCompare εφαρμόζεται αμέσως μετά τις κλήσεις των συναρτήσεων calcMinViDistNode και calcMinViDistEdge. Ο έλεγχος για το πλήθος των αντικειμένων στο τρέχον σύνολο των αποτελεσμάτων, kCheck, εφαρμόζεται πριν κληθεί η συνάρτηση NSIncorporate.

5.3 Επέκταση του Sweep σε SweepkNS

Σχετικά με την επέκταση του αλγορίθμου Sweep σε SweepkNS, υπάρχει ένα βασικό πρόβλημα που έχει να κάνει με την σταδιακή ανάκτηση των k κοντινότερων γειτόνων.

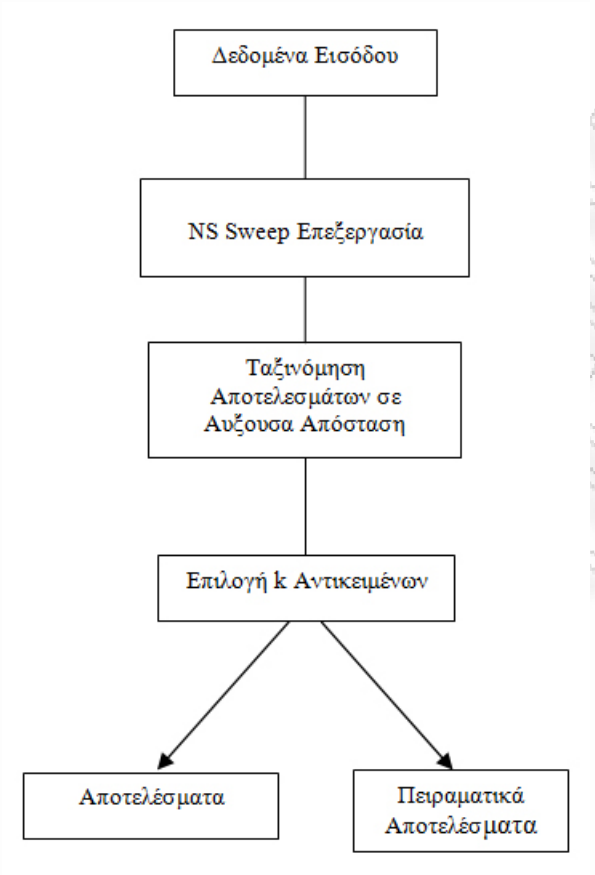
Εξ' ορισμού ο αλγόριθμος Sweep βάζει αντικείμενα και κόμβους στην ουρά προτεραιότητας σύμφωνα με την αύξουσα γωνία έναρξής τους και έτσι αναζητάει τα αντικείμενα σύμφωνα με την κατεύθυνσή τους. Όταν ο χρήστης θέλει να αναζητήσει τους k κοντινότερους γείτονες, τον ενδιαφέρουν οι k κοντινότεροι σε οποιαδήποτε κατεύθυνση μέσα στο γωνιακό εύρος που έχει ορίσει. Ο αλγόριθμος Sweep αναζητάει σταδιακά κοντινότερους ορατούς γείτονες σύμφωνα με την αύξουσα γωνιά τους. Έτσι είναι αδύνατο να αναζητήσουμε σταδιακά τους k κοντινότερους ορατούς γείτονες προς όλες τις κατευθύνσεις μέσα στο γωνιακό εύρος που ορίζει ο χρήστης.

Προκειμένου να επιτευχθεί η αναζήτηση των k κοντινότερων ορατών γειτόνων θα πρέπει πρώτα να εκτελεστεί, ο αλγόριθμος Sweep, και να δώσει το τελικό σύνολο των αποτελεσμάτων που περιέχει όλους τους κοντινότερους ορατούς γείτονες. Μετά την ολοκλήρωση του αλγορίθμου θα πρέπει να ταξινομήσουμε τα αντικείμενα του αποτελέσματος σε αύξουσα απόσταση και να πάρουμε τα k πρώτα από αυτά. Συνεπώς δεν γίνεται καμία επέμβαση στο εσωτερικό της NS Sweep επεξεργασίας. Η αρχιτεκτονική της Sweep πρακτικής αλλάζει στο προηγούμενο επίπεδο, από την NS Sweep επεξεργασία και γίνεται όπως στην εικόνα 5.3.

Στην εικόνα 5.3 παρατηρούμε ότι εκτός της NS Sweep επεξεργασίας, δηλαδή όταν τελειώσει την εκτέλεση ο αλγόριθμος Sweep έχει υλοποιηθεί μια διαδικασία που ταξινομεί τα αποτελέσματα σε αύξουσα απόσταση από το σημείο ερωτήματος. Κατόπιν μια άλλη διαδικασία επιλέγει τα k κοντινότερα αντικείμενα πριν πάρουμε τα τελικά αποτελέσματα. Οι διαδικασίες της ταξινόμησης και της επιλογής αντικειμένων έχουν υλοποιηθεί εκτός της κλάσης που κάνει την καθαυτό Sweep επεξεργασία, δηλαδή εκτός της κλάσης nssweep.cc και μέσα στην κλάση sweepkns.cc (βλ. Παράρτημα Α). Η τελευταία κλάση διαβάζει τα στοιχεία εισόδου που δίνει ο χρήστης, καλεί τον αλγόριθμο Sweep, ταξινομεί τα αποτελέσματα του Sweep, επιλέγει τα k κοντινότερα αντικείμενα και τυπώνει τα αποτελέσματα στην έξοδο, με απλά λόγια διαχειρίζεται την NS Sweep πρακτική και περιέχει τη συνάρτηση main.

5.4 Υλοποίηση Επιπρόσθετων Συναρτήσεων

Πέρα από τις επεκτάσεις των αλγορίθμων που περιγράφηκαν παραπάνω υλοποιήθηκαν και κάποιες επιπρόσθετες συναρτήσεις για την εκτύπωση των αποτελεσμάτων σε εικόνα, για την ευδιάκριτη εμφάνιση των αποτελεσμάτων στην οθόνη εξόδου και επιπλέον διορθώθηκαν κάποια σφάλματα του αρχικού κώδικα που παραλήφθηκε από τους συγγραφείς του άρθρου [3].



Εικόνα 5.3. Αρχιτεκτονική SweepkNS πρακτικής.

Σε όλους τους αλγόριθμους που δημιουργήθηκαν (PostPruningNS, PostPruningkNS, PrePruningNS, PrePruningkNS, SweepkNS) και στον αλγόριθμο Sweep προστέθηκε μια συνάρτηση που εκτυπώνει τα αποτελέσματα του V(k)NS ερωτήματος στο οποίο απαντούν. Συγκεκριμένα τυπώνει σε εικόνα όλα τα αντικείμενα του συνόλου δεδομένων, τυπώνει τη θέση του σημείου ερωτήματος και χρωματίζει διαφορετικά, από τα υπόλοιπα, τα αντικείμενα εκείνα που επιστρέφονται ως αποτέλεσμα. Η συνάρτηση αυτή ονομάζεται drawResult και είναι υλοποιημένη μέσα στην κλάση που περιέχει τη main συνάρτηση κάθε αλγορίθμου. Η συνάρτηση αυτή δημιουργεί ένα αρχείο εικόνας με κατάληξη .eps. Το αρχείο αυτό χρησιμοποιείται για την επαλήθευση των αποτελεσμάτων - αντικειμένων που επιστρέφει το κάθε ερώτημα που τίθεται.

Στον αρχικό κώδικα υπήρχαν κάποια σφάλματα που έδιναν null τιμές στα αποτελέσματα. Ένας από τους λόγους που γίνονταν αυτό είναι ότι δεν δούλευε σωστά η συνάρτηση που χωρίζει ένα αντικείμενο κατά μήκος του άξονα των x.

Όπως είδαμε στο κεφάλαιο 2 στον ορισμό του γωνιακού εύρους ενός MBR, η ελάχιστη γωνία του MBR είναι μικρότερη από τη μέγιστη γωνία του. Αυτό όμως δεν ισχύει στην περίπτωση που το MBR τέμνεται από τον άξονα των x που χαράσσεται με αρχή το σημείο ερωτήματος. Σε αυτή την περίπτωση το MBR διαχωρίζεται οριζόντια σε δύο ορθογώνια παραλληλόγραμμα έτσι ώστε το ένα να βρίσκεται από τον άξονα των x και πάνω και το άλλο να είναι κάτω από τον άξονα των x. Αυτή τη διαδικασία καλείται να την πραγματοποιήσει η συνάρτηση acrossX που βρίσκεται μέσα στην κλάση polygon.cc. Η συνάρτηση αυτή δεν έκανε σωστό διαχωρισμό του MBR και επιπλέον επέστρεφε τιμές null όταν το σημείο ερωτήματος και το MBR είχαν την ίδια y συντεταγμένη. Έτσι η συνάρτηση αυτή τροποποιήθηκε.

Η άλλη αιτία για την οποία ο κώδικας επέστρεφε λανθασμένα αποτελέσματα είναι η συνάρτηση angle της κλάσης point.cc. Η συνάρτηση αυτή καλείται από ένα σημείο του χώρου με όρισμα κάποιο άλλο σημείο, ας πούμε το σημείο ερωτήματος και επιστρέφει τη γωνία του σημείου που την καλεί ως προς το σημείο ερωτήματος. Για να το κάνει αυτό ελέγχει σε ποιο τεταρτημόριο ως προς το σημείο ερωτήματος βρίσκεται το σημείο που την καλεί και κατόπιν υπολογίζει τη γωνία. Εδώ επίσης δεν είχε προβλεφθεί η περίπτωση όπου το σημείο που καλεί τη συνάρτηση εφάπτεται σε έναν από τους άξονες συντεταγμένων. Επιπλέον δεν γίνονταν η σωστή επιλογή των τεταρτημορίων με αποτέλεσμα να επιστρέφονται null τιμές. Η συνάρτηση angle διορθώθηκε.

Υλοποιήθηκε ακόμα μέσα στη συνάρτηση main του κάθε αλγορίθμου ένα κομμάτι κώδικα για την ομαλή εμφάνιση των αποτελεσμάτων στην οθόνη εξόδου. Ο κώδικας έτσι όπως παραλήφθηκε αρχικά για να τυπώσει τα αποτελέσματα χώριζε το γωνιακό εύρος σε ίσα και μικρά διαστήματα μοιρών και εμφάνιζε ένα διάστημα και ένα αντικείμενο το οποίο ήταν NS σε αυτό το διάστημα. Το αποτέλεσμα είχε πολλαπλές εμφανίσεις του ίδιου αντικειμένου, σε περίπτωση που το αντικείμενο αυτό κάλυπτε συνεχόμενα πολλά τέτοια μικρά διαστήματα. Ο κώδικας που υλοποιήθηκε μέσα στην main εμφανίζει μια φορά το NS αντικείμενο με όλο το γωνιακό εύρος που καλύπτει συνεχόμενα. Αν το αντικείμενο είναι NS και σε κάποιο άλλο γωνιακό εύρος που δεν είναι συνέχεια του προηγούμενου γωνιακού εύρους, τότε μόνο το αντικείμενο θα ξαναεμφανιστεί στο αποτέλεσμα.

Με την διόρθωση αυτών των σφαλμάτων και όλες τις τροποποιήσεις των αλγορίθμων που περιγράφτηκαν παραπάνω σε αυτό το κεφάλαιο έχουν δημιουργηθεί ολοκληρωμένες πρακτικές που απαντούν στα ερωτήματα κοντινότερου ορατού γείτονα σε συγκεκριμένες γωνίες δίνοντας τη δυνατότητα του χρήστη να επιλέγει το πλήθος αυτών των γειτόνων.

6

Υλοποίηση

Ο κώδικας που μοιράστηκαν μαζί μας οι συγγραφείς του άρθρου [3] είναι γραμμένος σε C++. Η έκδοση του πακέτου του κώδικα μπορεί να μεταγλωττίζεται και να εκτελείται σε περιβάλλον Unix/Linux με μεταγλωττιστή gcc. Χρησιμοποιώντας αυτόν τον κώδικα όλοι οι αλγόριθμοι των ερωτημάτων οπτικού πεδίου που υλοποιήθηκαν είναι γραμμένοι σε C++. Πιο συγκεκριμένα το πακέτο κώδικα μεταγλωττίστηκε και εκτελέστηκε σε περιβάλλον Ubuntu 11.04 με μεταγλωττιστή gcc 4.5.2. Όλα τα εκτελέσιμα βρίσκονται μέσα στον φάκελο bin. Όλες οι εντολές εκτελούνται μέσα από την κονσόλα του Linux.

6.1 Εκτέλεση των Αλγορίθμων

Πριν από την εκτέλεση οποιουδήποτε αλγορίθμου θα πρέπει να δημιουργηθεί το R-tree ευρετήριο των χωρικών δεδομένων, με βάση το οποίο θα γίνονται οι αναζητήσεις. Στο πακέτο κώδικα που μας δόθηκε παρέχεται η υπηρεσία για τη δημιουργία ενός R-tree, εισαγωγής/διαγραφής στοιχείων, περιήγηση και αναζήτησης. Οι παραπάνω υπηρεσίες παρέχονται από την κλάση rtreetest.cc που βρίσκεται στο αρχείο src. Η εντολή με την οποία δημιουργούμε το R-tree ευρετήριο είναι:

```
./rtreetest -p 4096 -d 2 -f data.txt -i index_file.txt
  -o rtree_image.eps -v true
```

Όπου:

- p, είναι το μέγεθος της σελίδας εισόδου εξόδου,
- d, είναι η διάσταση του χώρου,
- f, είναι το αρχείο των χωρικών δεδομένων, μέσα στο οποίο τα δεδομένα συντάσσονται στη μορφή "id xmin ymin xmax ymax", χωρισμένα με tab. Δηλαδή το αρχείο αυτό περιέχει τις μέγιστες και τις ελάχιστες συντεταγμένες των MBRs,
- i, είναι το όνομα του αρχείου ευρετηρίου που θα δημιουργηθεί,

– o, είναι η εικόνα του ευρετηρίου που θα δημιουργηθεί,
– v, μια μεταβλητή που είναι πάντα αληθής.

Στην εικόνα 6.1 βλέπουμε ένα παράδειγμα του R-tree που δημιουργείται με αυτή την εντολή.

Εικόνα 6.1. R-tree ευρετήριο με τους δρόμους της Ελλάδας.

Πρόκειται για ένα σύνολο δεδομένων που περιέχει τους δρόμους της Ελλάδας. Παρατηρούμε ότι το R-tree περιέχει MBRs τα οποία περιέχουν μικρότερα MBRs τα οποία τελικά περιέχουν τα MBRs των αντικειμένων, δηλαδή τους δρόμους της Ελλάδας. Μετά την εκτέλεση της εντολής rtreetest δημιουργείται μέσα στον φάκελο bin το αρχείο index_file.idx, το οποίο μπορεί να χρησιμοποιηθεί για την εκτέλεση οποιουδήποτε από τους αλγορίθμους ερωτημάτων οπτικού πεδίου του πακέτου κώδικα.

Για την εκτέλεση του αλγορίθμου PostPruningNS δίνουμε στο τερματικό την εντολή:

```
./postpruningns -p 4096 -d 2 -f data.txt -i index_file.idx
  -q query_file.txt -a0 0 -a1 360 -m 1 -t 1 -v true
```

Τα στοιχεία p, d και v είναι τα ίδια που διευκρινίστηκαν για την εκτέλεση του κώδικα rtreetest. Τα υπόλοιπα στοιχεία είναι:

- f, είναι το αρχείο των χωρικών δεδομένων όπου τα δεδομένα συντάσσονται στη μορφή "id #vertices xmin ymin xmax ymin xmax ymin xmax ymax xmax ymax xmin ymax xmin ymax xmin ymin", χωρισμένα με tabs. Περιέχει δηλαδή τις συντεταγμένες των κορυφών των MBRs επαναλαμβανόμενες, διότι ο κώδικας διαβάζει στο αρχείο δεδομένων δύο κορυφές και σχηματίζει μια ακμή, μετά διαβάζει άλλες δύο κορυφές κοκ. μέχρι τον πλήθος των κορυφών που δηλώνεται σε κάθε γραμμή του αρχείου. Επομένως έτσι πρέπει να δομηθεί το αρχείο δεδομένων,
- i, είναι το ευρετήριο που δημιουργήθηκε με τον κώδικα rtreetest,
- q, είναι το αρχείο κειμένου που περιέχει τα σημεία ερωτημάτων στη μορφή "id x y", χωρισμένα με tabs,
- a0, η γωνία έναρξης αναζήτησης κοντινότερων ορατών γειτόνων,
- a1, η γωνία τερματισμού αναζήτησης,
- m, το πλήθος των επιπέδων,
- t, το πλήθος των τομέων.

Στο σημείο αυτό επισημαίνεται ότι όλοι οι αλγόριθμοι υλοποιήθηκαν και ελέγχθηκαν ότι εκτελούνται χωρίς σφάλματα για δύο διαστάσεις (-d 2), για ένα επίπεδο (-m 1) και για ένα τομέα (-t 1) που καλύπτει όλο το γωνιακό εύρος που δίνει ο χρήστης.

Για την εκτέλεση του αλγορίθμου PostPruningkNS δίνουμε την εντολή:

```
./postpruningkns -p 4096 -d 2 -f data.txt -i index_file.idx
  -q query_file.txt -a0 0 -a1 360 -m 1 -t 1 -v true -k 30
```

Στο τέλος της εντολής έχει προστεθεί και η παράμετρος k που δηλώνει το μέγιστο πλήθος των κοντινότερων γειτόνων που ζητείται να επιστρέψει ο αλγόριθμος.

Για τους αλγορίθμους Sweep και PrePruningNS η εντολή είναι όμοια με αυτή του PostPruningNS. Το μόνο που αλλάζει είναι το όνομα του εκτελέσιμου αρχείου. Ομοίως οι αλγόριθμοι SweepkNS και PrePruningkNS εκτελούνται με εντολή όμοια με αυτή του PostPruningkNS.

Τα αποτελέσματα τυπώνονται στην οθόνη εξόδου. Για παράδειγμα ας υποθέσουμε ότι εκτελούμε τον αλγόριθμο PostPruningkNS σε ένα σύνολο δεδομένων με τις λίμνες της Ελλάδας, δίνοντας την παράμετρο k ίση με 5. Ως έξοδο θα πάρουμε:

```
q0: 400083,4.22009e+06
(0, 33.4075):-1
(33.4075, 47.167):65
(47.167, 138.37):-1
(138.37, 146.394):63
(146.394, 217.14):-1
(217.14, 222.307):71
(222.307, 301.511):-1
(301.511, 302.295):74
(302.295, 349.499):-1
```

```
(349.499, 349.821):68
(349.821, 360):-1
```

Αυτό σημαίνει ότι το σημείο ερωτήματος είναι στη θέση (x, y): 400083,4.22009e+06. Με βάση αυτή τη θέση ερωτήματος βρέθηκαν πέντε αντικείμενα. Συγκεκριμένα στο γωνιακό εύρος (0, 33.4075) δεν υπάρχει κανένα κοντινότερο αντικείμενο και εκτυπώνεται το -1. Στη γωνία (33.4075, 47.167) βρέθηκε ως κοντινότερο αντικείμενο αυτό που έχει id 65 κοκ.. Η εκτύπωση των αντικειμένων τελειώνει στη γωνιά λήξης που σε αυτή την περίπτωση είναι 360 μοίρες.

7

Εμπειρική Μελέτη

Η πειραματική μελέτη θα περιλαμβάνει όλους τους αλγορίθμους που υλοποιήθηκαν (PostPruningNS, PostPruningkNS, PrePruningNS, PrePruningkNS, SweepkNS) και τον αλγόριθμο Sweep. Τα πειράματα θα χωριστούν σε τρεις κατηγορίες ανάλογα με την επίδραση της μεταβλητής του άξονα των x, ως προς την οποία γίνονται οι εκτιμήσεις. Αυτές οι κατηγορίες είναι η επίδραση της παραμέτρου k, το μέγεθος του συνόλου δεδομένων και το μέγεθος των αντικειμένων. Τα κριτήρια αξιολόγησης των αλγορίθμων, μετά φυσικά την ορθότητά τους, αναφέρονται στην αποδοτικότητα. Η αποδοτικότητα καθορίζεται από το κόστος εισόδου - εξόδου, τον χώρο μνήμης και το χρόνο εκτέλεσης.

Στην περίπτωση των αλγορίθμων ερωτημάτων οπτικού πεδίου η περισσότερη μνήμη καταναλώνεται για τη διατήρηση της ουράς προτεραιότητας. Στην ουρά προτεραιότητας διατηρούνται οι κόμβοι και τα φύλλα του R-tree, προκειμένου να προσπελαύνονται από τον εκάστοτε αλγόριθμο. Έτσι όσες περισσότερες εισόδους έχει η ουρά προτεραιότητας τόση περισσότερη μνήμη καταναλώνεται. Το κόστος εισόδου - εξόδου αφορά την προσπέλαση των κόμβων/σελίδων στο ευρετήριο R-tree. Το κόστος επεξεργασίας εξαρτάται από την πολυπλοκότητα του εκάστοτε αλγορίθμου και στην περίπτωση των ερωτημάτων οπτικού πεδίου ο υπολογισμός της ορατής περιοχής κάθε σημείου ερωτήματος έχει μεγάλο κόστος επεξεργασίας.

Πριν αναλυθούν τα πειράματα θα πρέπει να διευκρινιστούν τα χαρακτηριστικά του υπολογιστικού συστήματος στο οποίο εκτελέστηκαν. Το λειτουργικό σύστημα που χρησιμοποιήθηκε είναι Ubuntu 11.04 με επεξεργαστή Intel Core 2 Duo 2.0 GHz. Κάθε σύνολο δεδομένων αποθηκεύεται σε ένα R-tree με μέγεθος σελίδας δίσκου στα 4 KB. Το μέγεθος της προσωρινής μνήμης (cache) τίθεται στις 30 σελίδες για όλα τα πειράματα. Κάθε πείραμα διεξάγεται για 20 σημεία ερωτήματος σε τυχαίες θέσεις και τα αποτελέσματα που μετρούνται είναι ο μέσος όρος αυτών των 20 ερωτημάτων.

Για την αξιολόγηση της απόδοσης των αλγορίθμων χρησιμοποιούνται τεχνητά και αληθινά δεδομένα. Χρησιμοποιώντας τεχνητά δεδομένα μπορούμε να αλλάζουμε τις παραμέτρους των πειραμάτων κατά βούληση και έτσι εξετάζοντας διαφορετικές περιπτώσεις να έχουμε μια πιο ολοκληρωμένη πειραματική μελέτη.

Για τα τεχνητά δεδομένα χρησιμοποιήθηκε μια πλατφόρμα παραγωγής χωρικών δεδομένων που αναπτύχθηκε στο πανεπιστήμιο Πειραιώς από την ομάδα του κ. Θεοδωρίδη Γιάννη και είναι διαθέσιμο στην ιστοσελίδα [10]. Τα τεχνητά δεδομένα που δημιουργούνται κατανέμονται τυχαία και ομοιόμορφα σε ένα χώρο 10000 x 10000 τετραγωνικές μονάδες. Το ύψος και το πλάτος του κάθε ορθογωνίου ποικίλει τυχαία από 1 έως 10 μονάδες εκτός από την τελευταία κατηγορία πειραμάτων όπου συγκρίνουμε την απόδοση των αλγορίθμων ως προς κάποια συγκεκριμένα μεγέθη αντικειμένων. Τα αληθινά δεδομένα ανακτήθηκαν από την ιστοσελίδα [10]. Τα δεδομένα αυτά αφορούν δρόμους στην Ελλάδα και στις πόλεις Long Beach και Montgomery των Ηνωμένων Πολιτειών.

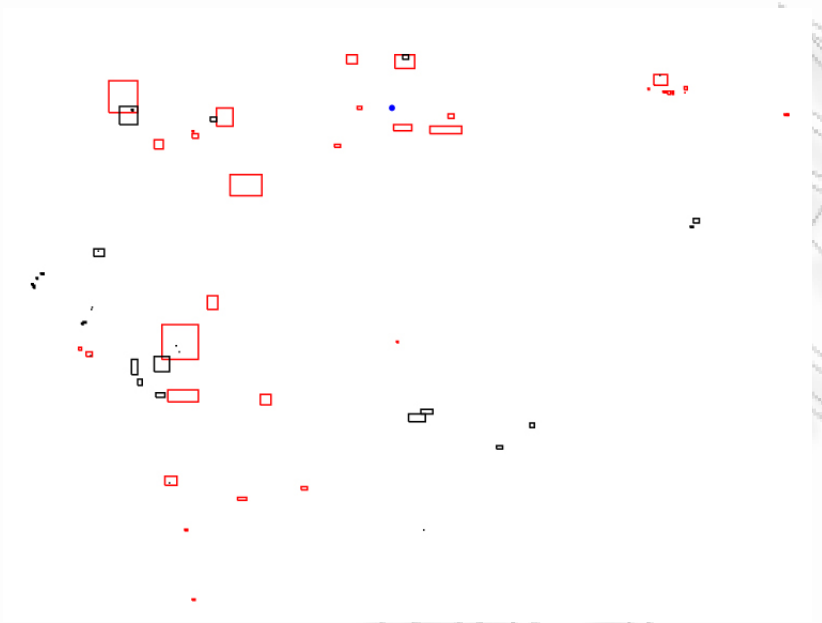
7.1 Έλεγχος Ορθότητας Αποτελεσμάτων

Η ορθότητα των αποτελεσμάτων είναι ένα άλλο κριτήριο αποτίμησης της εργασίας, γι' αυτό πριν από την εκτέλεση των πειραμάτων ελέγχθηκε η ορθότητα. Όλοι οι αλγόριθμοι που υλοποιήθηκαν απαντούν στα ίδια ερωτήματα οπτικού πεδίου. Επομένως με βάση το ίδιο σημείο ερωτήματος και το ίδιο σύνολο δεδομένων θα πρέπει να επιστρέφουν τα ίδια αντικείμενα στις ίδιες ακριβώς γωνίες. Αυτό ελέγχθηκε τόσο από τα αποτελέσματα που τυπώνονται στην οθόνη εξόδου όσο και οπτικά με την εικόνα των αποτελεσμάτων. Επιπλέον με την εικόνα διαπιστώνουμε οπτικά αν τα αντικείμενα που επιστρέφονται ως αποτέλεσμα είναι όντως οι κοντινότεροι ορατοί γείτονες, στο σημείο ερωτήματος και σε συγκεκριμένες γωνίες. Στην εικόνα 7.1 βλέπουμε το αποτέλεσμα που επιστράφηκε από την εκτέλεση των αλγορίθμων PostPruningNS, PrePruningNS και Sweep. Η εικόνα είναι ακριβώς ίδια και για τους τρεις αυτούς αλγορίθμους.

Το σύνολο δεδομένων που φαίνεται στην εικόνα 7.1 αναπαριστά τις λίμνες της Ελλάδας και ανακτήθηκε από την ιστοσελίδα [10]. Πρόκειται για ένα σύνολο δεδομένων με 77 ορθογώνια παραλληλόγραμμα, το οποίο επιλέχθηκε σκόπιμα μικρό για να είναι περισσότερο ευδιάκριτα τα αποτελέσματα που επιστρέφονται από τους αλγορίθμους. Κάθε παραλληλόγραμμο που φαίνεται στην εικόνα αναπαριστά μια λίμνη. Με τους αλγόριθμους (PostPruningNS, PrePruningNS και Sweep) που εκτελέστηκαν σε αυτό το σύνολο δεδομένων αναζητούνται όλοι οι κοντινότεροι ορατοί γείτονες με τις αντίστοιχες γωνίες τους στο γωνιακό εύρος 0 - 360 μοίρες.

Το σημείο ερωτήματος είναι η μπλε τελεία. Με κόκκινο χρωματίζονται τα περιγράμματα των παραλληλογράμμων που ανήκουν στο σύνολο των κοντινότερων ορατών γειτόνων, δηλαδή στο σύνολο των αποτελεσμάτων. Μαύρα είναι τα παραλληλόγραμμα που δεν ανήκουν σε αυτό το σύνολο. Όπως φαίνεται, μαύρα είναι όλα τα αντικείμενα που δεν είναι ορατά από τη θέση ερωτήματος.

Ας δούμε ποια θα είναι η εικόνα των αποτελεσμάτων στο ίδιο σύνολο δεδομένων όταν ο χρήστης επιλέγει να αναζητήσει τους πέντε κοντινότερους ορατούς γείτονες στο ίδιο γωνιακό εύρος (0-360). Για την απάντηση στο ερώτημα αυτό εκτελούνται οι αλγόριθμοι PostPruningkNS, PrePruningkNS και Sweep-



Εικόνα 7.1. Αποτέλεσμα εξόδου αλγορίθμων PostPruningNS, PrePruningNS και Sweep.

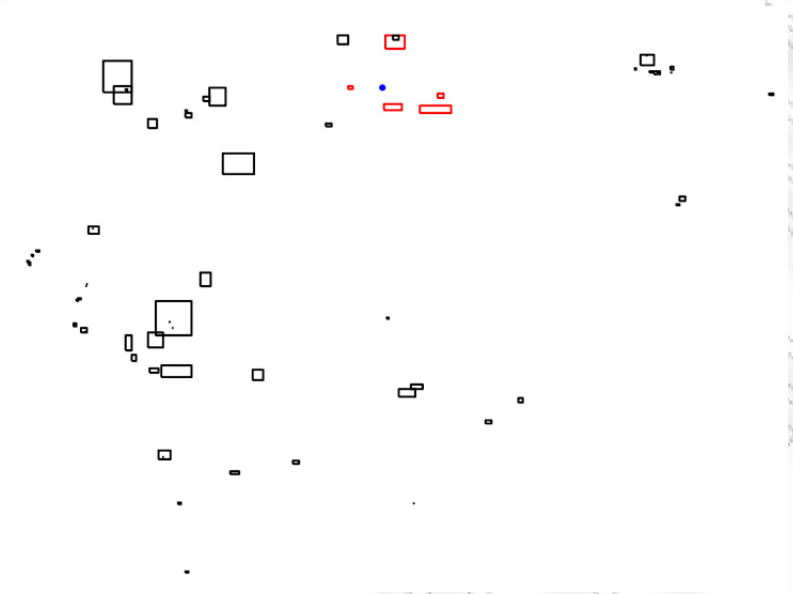
kNS. Η εικόνα του αποτελέσματος είναι η ίδια και για τους τρεις αλγορίθμους και είναι η εικόνα 7.2.

Από την εικόνα 7.2 επιβεβαιώνεται οπτικά ότι οι αλγόριθμοι έχουν επιστρέψει τους σωστούς πέντε κοντινότερους ορατούς γείτονες. Βλέπουμε πέντε παραλληλόγραμμα γύρω από το σημείο ερωτήματος που έχουν χρωματιστεί κόκκινα ενώ όλα τα υπόλοιπα είναι μαύρα.

Έχοντας ολοκληρώσει τον έλεγχο ορθότητας των αποτελεσμάτων με ορισμένα σύνολα δεδομένων μπορούμε να προχωρήσουμε στην διεξαγωγή των πειραμάτων από τα οποία θα πάρουμε πληροφορίες για την απόδοση των αλγορίθμων.

7.2 Επίδραση της Παραμέτρου k

Αυτή η κατηγορία πειραμάτων αφορά μόνο τους αλγορίθμους που επιτρέπουν στον χρήστη να ορίσει τον αριθμό των κοντινότερων γειτόνων. Δηλαδή τους αλγορίθμους PostPruningkNS, PrePruningkNS και SweepkNS. Οι τρεις αλγόριθμοι συγκρίνονται ως προς το κόστος εισόδου - εξόδου, το κόστος επεξεργασίας και το κόστος μνήμης που στην περίπτωση των ερωτημάτων οπτικού σχετίζεται με το μέγεθος της ουράς προτεραιότητας. Για την διεξαγωγή αυτού του πειράματος θα χρησιμοποιηθούν τεχνητά και αληθινά δεδομένα. Για αμφότερα



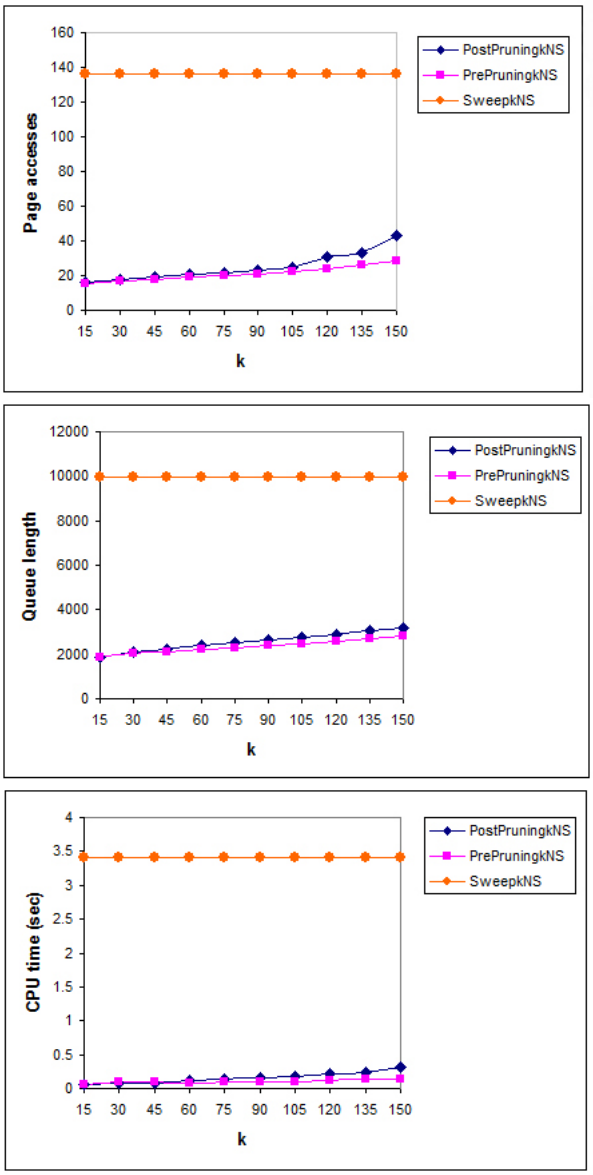
Εικόνα 7.2. Αποτέλεσμα εξόδου αλγορίθμων PostPruning5NS, PrePruning5NS και Sweep5NS.

τα είδη συνόλων δεδομένων το k μεταβάλλεται από 15 έως 150, αυξάνοντάς το κατά 15 αντικείμενα τη φορά.

Τα διαγράμματα που βλέπουμε στην εικόνα 7.3 αφορούν τεχνητά δεδομένα με 100000 MBRs. Τα 20 σημεία ερωτήματος παράχθηκαν επίσης τεχνητά με παραμέτρους ίδιες με αυτές που παράχθηκε το σύνολο δεδομένων. Δηλαδή με ομοιόμορφη κατανομή σε ένα χώρο 10000 x 10000 τετραγωνικές μονάδες.

Όπως ήταν αναμενόμενο, για όλες τις μετρήσεις κόστους ο αλγόριθμος Sweep δεν παρουσιάζει καμία διακύμανση καθ' όλη τη μεταβολή της παραμέτρου k. Όπως αναλύθηκε και σε προηγούμενα κεφάλαια ο αλγόριθμος SweepkNS, εξ' ορισμού, δεν μπορεί να ανακτήσει σταδιακά τους k κοντινότερους γείτονες. Για να γίνει κάτι τέτοιο θα πρέπει να ανακτήσει όλους τους κοντινότερους γείτονες και αφού τους ταξινομήσει σε αύξουσα απόσταση να επιλέξει τους k πρώτους από αυτούς. Επομένως είναι λογικό που τον βλέπουμε να έχει τόσο μεγάλη διαφορά κόστους, για όλες τις μετρήσεις, συγκριτικά με τους αλγορίθμους PostPruningkNS και PrePruningkNS. Εφόσον ο SweepkNS αναζητάει όλους τους κοντινότερους ορατούς γείτονες την ίδια στιγμή που οι άλλοι δύο αλγόριθμοι αναζητούν k γείτονες.

Για όλες τις μετρήσεις κόστους όσο το k αυξάνεται, αυξάνεται και το κόστος των αλγορίθμων PostPruningkNS και PrePruningkNS. Όταν το k είναι μικρό οι δύο αλγόριθμοι συμπεριφέρονται το ίδιο. Καθώς το k αυξάνει το κόστος του PostPruningkNS αυξάνεται πιο γρήγορα από τα κόστος του PrePruningkNS. Αυτό συμβαίνει επειδή καθώς ανακτούνται όλο και περισσότεροι κοντινότεροι



Εικόνα 7.3. Επίδραση της παραμέτρου k σε τεχνητά δεδομένα με 100000 MBRs.

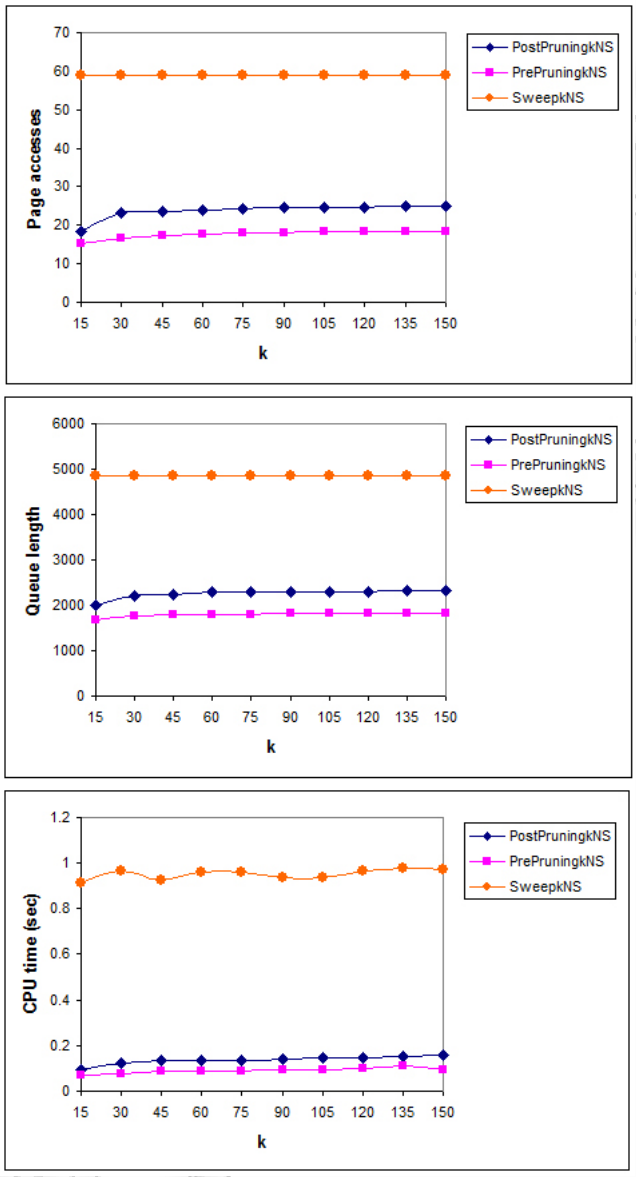
ορατοί γείτονες, η αναλογία μεταξύ ορατών και μη ορατών κόμβων αυξάνεται. Αυτοί οι μη ορατοί κόμβοι κλαδεύονται από τον PrePruningkNS όχι όμως και από τον PostPruningkNS.

Σε όρους κόστους εισόδου - εξόδου ο αλγόριθμος PrePruningkNS κοστίζει λιγότερο διότι είναι βέλτιστος στο κόστος εισόδου - εξόδου. Όπως αναλύθηκε σε προηγούμενο κεφάλαιο, για τον υπολογισμό της ελάχιστης ορατής απόστασης ενός αντικειμένου δεν απαιτείται η επεξεργασία ολόκληρου του συνόλου δεδομένων αλλά ενός μέρους από αυτό. Εφόσον την ελάχιστη ορατή απόσταση ενός αντικειμένου μπορούν να την επηρεάσουν μόνο τα αντικείμενα με μικρότερη απόσταση από αυτό. Επιπλέον επειδή οι κοντινότεροι ορατοί γείτονες για τους αλγορίθμους PostPruning(k)NS και PrePruning(k)NS ανακτούνται σταδιακά, για τον υπολογισμό της ελάχιστης ορατής απόστασης ενός αντικειμένου λαμβάνονται υπόψη ως εμπόδια μόνο τα αντικείμενα του τρέχοντος συνόλου αποτελεσμάτων. Μόνο αυτά τα αντικείμενα έχουν μικρότερη απόσταση από τον εξεταζόμενο κόμβο. Ο αλγόριθμος PrePruning(k)NS ελέγχει την ελάχιστη ορατή απόσταση των κόμβων και δεν τους εξετάζει αν αυτή η απόσταση είναι μεγαλύτερη από αυτή της κεφαλής της ουράς, αλλά τους επανεισάγει στην ουρά προτεραιότητας. Συνεπώς ο επόμενος κοντινότερος ορατός γείτονας δεν μπορεί να ανακτηθεί χωρίς πρώτα να εξερευνηθεί ο κόμβος με την τρέχουσα μικρότερη ελάχιστη ορατή απόσταση. Συμπερασματικά ο PrePruning(k)NS επισκέπτεται τον ελάχιστο αριθμό κόμβων και έτσι είναι βέλτιστος σε κόστος εισόδου - εξόδου.

Σε όρους κόστους CPU, για μικρές τιμές του k ο PrePruningkNS έχει ελαφρώς μεγαλύτερο κόστος από τον PostPruningkNS. Αυτό συμβαίνει επειδή ο PrePruningkNS εφαρμόζει τον έλεγχο της ελάχιστης ορατής απόστασης τόσο σε αντικείμενα όσο και σε κόμβους, ενώ ο PostPruningkNS τον εφαρμόζει μόνο σε αντικείμενα. Καθώς όμως αυξάνει το k αυξάνεται ο αριθμός των κοντινότερων ορατών γειτόνων που πρέπει να ανακτηθούν και επομένως ο PostPruningkNS λαμβάνει υπόψη περισσότερες εισόδους, ενώ ο PrePruningkNS κλαδεύει αποδοτικά πολλούς κόμβους.

Σε όρους κόστους μνήμης που μεταφράζεται σε αριθμό εισόδων στην ουρά προτεραιότητας, για μικρές τιμές του k οι δύο αλγόριθμοι, PostPruningkNS και PrePruningkNS συμπεριφέρονται το ίδιο. Όσο αυξάνεται το k, ο PostPruningkNS παρουσιάζει μεγαλύτερο κόστος από τον PrePruningkNS, διότι ο τελευταίος κλαδεύει πιο αποτελεσματικά τον χώρο αναζήτησης με αποτέλεσμα να διατηρεί μια πιο μικρή ουρά προτεραιότητας.

Παρόμοια αποτελέσματα παρατηρούνται και με την διεξαγωγή αυτών των πειραμάτων σε αληθινά δεδομένα, όπως φαίνεται στην εικόνα 7.4. Τα πειράματα της εικόνας 7.4 διεξήχθηκαν με πραγματικά δεδομένα. Πρόκειται για 53145 MBRs που αναπαριστούν δρόμους της πόλης Long Beach. Τα σημεία ερωτήματος δημιουργήθηκαν λαμβάνοντας τυχαία συντεταγμένες από το σύνολο των αληθινών δεδομένων. Παρόμοια με τα αποτελέσματα των συνθετικών δεδομένων ο PrePruningkNS έχει καλύτερη απόδοση από τον PostPruningkNS, για όλα τα κριτήρια αποδοτικότητας. Η διαφορά κόστους ανάμεσα στους δύο αλγορίθμους είναι μεγαλύτερη από ότι στα συνθετικά δεδομένα. Αυτό συμβαίνει επειδή



Εικόνα 7.4. Επίδραση της παραμέτρου k σε αληθινά δεδομένα με 53145 MBRs από δρόμους στο Long Beach.

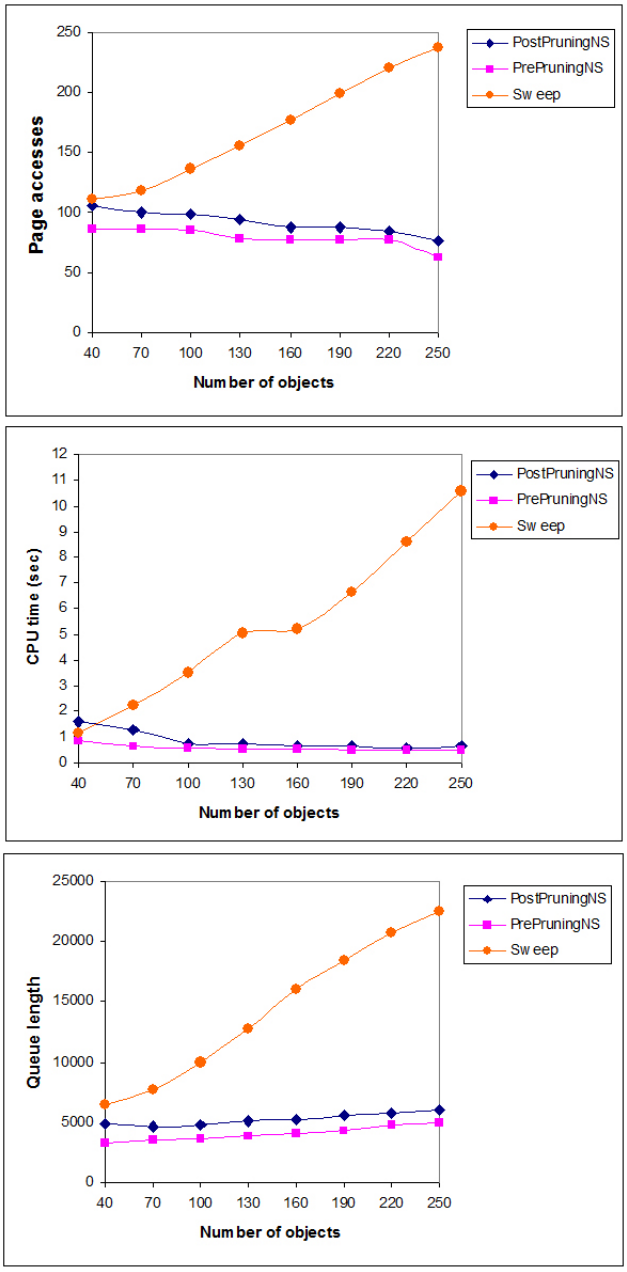
τα αληθινά δεδομένα έχουν μεγαλύτερη πυκνότητα από τα συνθετικά. Παρόλο που το μέγεθος του συνόλου δεδομένων είναι σχεδόν το μισό από αυτό των τεχνητών δεδομένων, το μέγεθος των MBRs είναι μεγαλύτερο με αποτέλεσμα το σύνολο δεδομένων να είναι πιο πυκνό. Η μεγαλύτερη πυκνότητα τονίζει τη διαφορά των αποτελεσμάτων που παράγονται από τις συναρτήσεις MinDist και MinViDist. Ο αλγόριθμος SweepkNS και εδώ δεν επηρεάζεται από τις μεταβολές του k.

7.3 Επίδραση του Μεγέθους του Συνόλου Δεδομένων

Σε αυτή την κατηγορία πειραμάτων μελετάμε την επίδραση του μεγέθους του συνόλου δεδομένων χρησιμοποιώντας τους αλγορίθμους PostPruningNS, PrePruningNS και Sweep για να ανακτήσουμε όλους τους κοντινότερους οδικούς γείτονες. Το μέγεθος του συνόλου δεδομένων ποικίλει από 40000 έως 250000, αυξάνοντας το, κάθε φορά, κατά 30000. Στην εικόνα 7.5 βλέπουμε τη διεξαγωγή των πειραμάτων με τεχνητά δεδομένα. Τα σημεία ερωτήματος έχουν δημιουργηθεί με ομοιόμορφη κατανομή σε ένα χώρο 10000 x 10000.

Στα διαγράμματα της εικόνας 7.5 παρατηρούμε ότι για το κόστος εισόδου - εξόδου, το κόστος CPU και το μήκος της ουράς προτεραιότητας το κόστος του Sweep αυξάνει καθώς αυξάνει το μέγεθος του συνόλου δεδομένων. Αυτό συμβαίνει επειδή ο αλγόριθμος Sweep εξετάζει όλους τους κόμβους του R-tree, εφόσον δεν έχει κάποια συνθήκη τερματισμού. Καθώς το σύνολο δεδομένων αυξάνεται, αυξάνεται επίσης και το μέγεθος του R-tree επομένως ο Sweep έχει περισσότερους κόμβους να επισκεφτεί. Για μικρό σύνολο δεδομένων, όπως βλέπουμε στο πείραμα με τα 40000 αντικείμενα το κόστος επεξεργασίας του Sweep είναι μικρότερο από αυτό του PostPruningNS. Αυτό συμβαίνει γιατί ο χώρος στα 40000 αντικείμενα είναι πιο αραιός και είναι πιθανότερο να υπάρχουν κενές γωνίες χωρίς κανένα NS και ο Sweep έχει την δυνατότητα να ξεπερνάει αυτές τις κενές γωνίες και να μην ψάχνει εξαντλητικά για NS. Αυτό θα αναλυθεί περισσότερο στην επόμενη κατηγορία πειραμάτων που μελετά την επίδραση του μεγέθους των αντικειμένων.

Το κόστος εισόδου - εξόδου και το κόστος CPU τον αλγορίθμων PostPruningNS και PrePruningNS, μειώνεται καθώς το σύνολο δεδομένων αυξάνει. Ο λόγος είναι ότι οι αλγόριθμοι αυτοί έχουν επεκταθεί από τον αλγόριθμο Ripple και έχουν κληρονομήσει την συνθήκη τερματισμού NS - TC του Ripple. Υπενθυμίζεται ότι σύμφωνα με αυτή τη συνθήκη ο αλγόριθμος τερματίζει όταν ικανοποιούνται δύο όροι: (i) σε κάθε κατεύθυνση υπάρχει ένα NS και (ii) όλα τα αντικείμενα στην ουρά προτεραιότητας είναι έξω από το μικρότερο κύκλο (με κέντρο το σημείο ερωτήματος) που περικλείει όλες τις NS απαντήσεις. Ο αλγόριθμος Ripple μπορεί να εφαρμόζει μια τέτοια συνθήκη επειδή ταξινομεί στην ουρά προτεραιότητας τα αντικείμενα σύμφωνα με την αύξουσα απόστασή τους. Ο Sweep δεν το κάνει αυτό και επομένως δεν μπορεί να παραλείψει τον υπόλοιπο έλεγχο της ουράς προτεραιότητας. Συνεπώς οι PostPruningNS και



Εικόνα 7.5. Επίδραση του μεγέθους του συνόλου δεδομένων σε τεχνητά δεδομένα.

PrePruningNS τερματίζουν την αναζήτηση όταν όλοι οι πιθανοί κοντινότεροι ορατοί γείτονες έχουν βρεθεί.

Υπάρχει και άλλος λόγος για τον οποίο το κόστος εισόδου - εξόδου και το κόστος CPU των PostPruningNS και PrePruningNS μειώνεται. Επειδή εφαρμόζουν τον έλεγχο της ελάχιστης ορατής απόστασης ενός αντικειμένου σε σχέση με την απόσταση της κεφαλής της ουράς. Αυτό επιτρέπει ένα καλό κλάδεμα του χώρου αναζήτησης. Επιπλέον ο PrePruningNS εφαρμόζει αυτόν τον έλεγχο και στους κόμβους με αποτέλεσμα να επισκέπτεται μόνο κόμβους που επικαλύπτονται με την ορατή περιοχή. Γι' αυτό παρατηρούμε ότι έχει καλύτερη απόδοση συγκριτικά με τον PostPruningNS. Η αρνητική σχέση που παρατηρούμε στο κόστος εισόδου - εξόδου και στο κόστος CPU των PostPruningNS και PrePruningNS καθώς το πλήθος του συνόλου δεδομένων αυξάνεται, οφείλεται στην αρνητική συσχέτιση μεταξύ του πλήθους των κοντινότερων ορατών γειτόνων και του συνόλου δεδομένων. Καθώς το πλήθος του συνόλου δεδομένων αυξάνεται ο χώρος γίνεται πιο πυκνός με αποτέλεσμα να περιορίζεται η ορατότητα πολλών αντικειμένων. Όταν το σύνολο δεδομένων είναι πιο μικρό ο χώρος αναζήτησης αυξάνεται καθώς η ακτίνα ορατότητας μεγαλώνει και περιλαμβάνει περισσότερα αντικείμενα.

Παρατηρούμε στην εικόνα 7.5 ότι το μήκος της ουράς προτεραιότητας αυξάνεται λίγο και για τους αλγορίθμους PostPruningNS και PrePruningNS. Για τον αλγόριθμο Sweep διευκρινίστηκε ότι αυτό συμβαίνει επειδή δεν έχει τη συνθήκη τερματισμού NS - TC που έχουν οι άλλοι δύο αλγόριθμοι, με αποτέλεσμα η διαφορά κόστους μνήμης σε σχέση με τους άλλους αλγορίθμους να είναι μεγάλη. Για τους άλλους δύο αλγορίθμους το κόστος αυξάνεται λίγο. Όλοι οι αλγόριθμοι γεμίζουν την ουρά προτεραιότητας με κόμβους και αντικείμενα αναδρομικά. Δηλαδή όταν βγάζουν κάποιον κόμβο από την ουρά ανακτούν τα παιδιά του και τα βάζουν στην ουρά προτεραιότητας. Όταν το σύνολο δεδομένων είναι μεγαλύτερο τότε το R-tree είναι μεγαλύτερο, με αποτέλεσμα κάθε κόμβος να έχει περισσότερα παιδιά. Επομένως για τους PostPruningNS και PrePruningNS το κόστος αυξάνεται. Επιπλέον επειδή ο έλεγχος της ελάχιστης ορατής απόστασης κλαδεύει αποδοτικά τους κόμβους που εξετάζονται παρατηρούμε ότι αυτή η αύξηση είναι μικρή.

Σε κάθε περίπτωση κόστους ο PrePruningNS αποδίδει καλύτερα από τους άλλους δύο αλγορίθμους. Ο λόγος είναι ότι συγκριτικά με τους άλλους αλγορίθμους ο PrePruningNS επισκέπτεται λιγότερους κόμβους, εκτελεί λιγότερους ελέγχους ορατότητας και έχει μικρότερο χρόνο επεξεργασίας.

7.4 Επίδραση του Μεγέθους των Αντικειμένων

Η πυκνότητα του χώρου αναζήτησης εξαρτάται τόσο από τον πληθάριθμο των αντικειμένων όσο και από το μέγεθος των αντικειμένων. Όσο μεγαλύτερο το μέγεθος των αντικειμένων τόσο μεγαλύτερη η πυκνότητα του χώρου αναζήτησης. Πυκνός χώρος αναζήτησης σημαίνει ότι ο αλγόριθμος δεν χρειάζεται να ψάχνει πολύ μακριά από το σημείο ερωτήματος για να βρίσκει κοντινότερους

γείτονες και επίσης σημαίνει ότι μειώνονται οι γωνίες του χώρου όπου δεν υπάρχει κανένας κοντινότερος γείτονας.

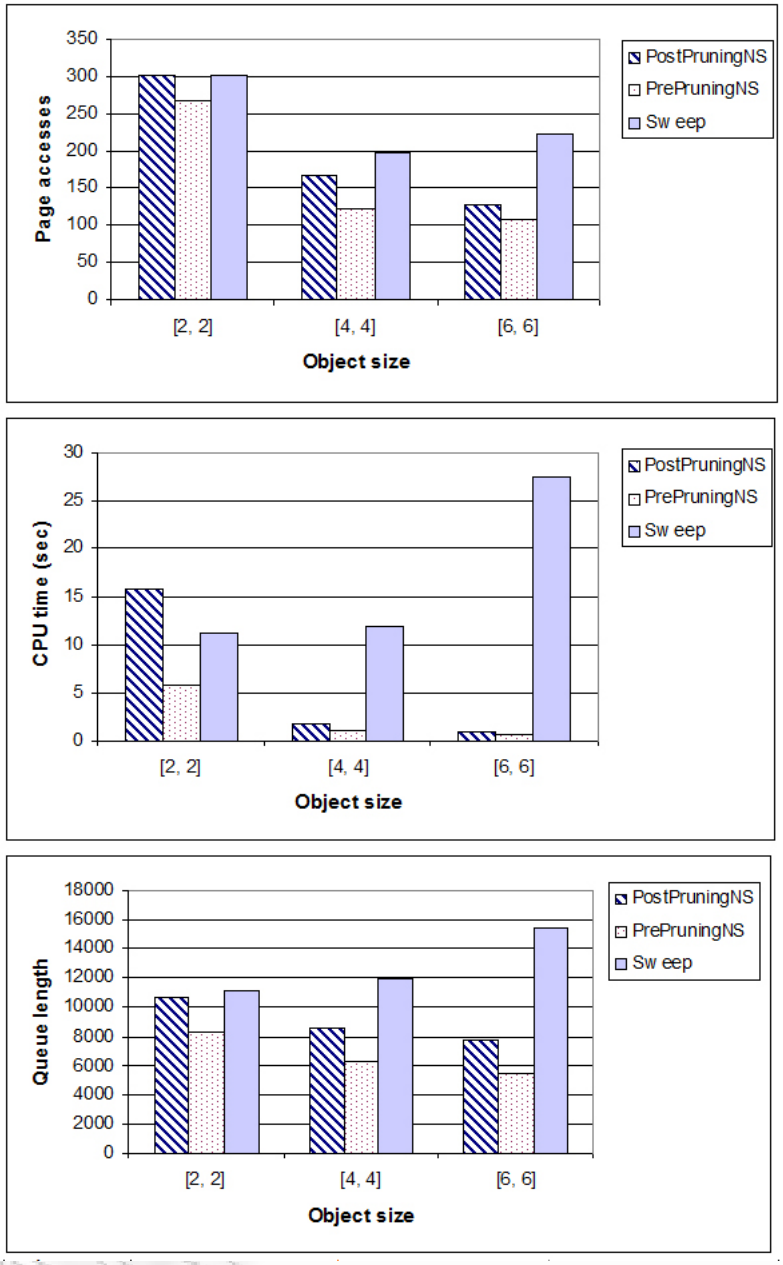
Αυτή η κατηγορία πειραμάτων μπορεί να εκτελεστεί μόνο με συνθετικά δεδομένα καθώς μόνο σε αυτά μπορούμε να καθορίσουμε και να μεταβάλουμε το μέγεθος των αντικειμένων. Θα χρησιμοποιηθούν οι αλγόριθμοι PostPruningNS, PrePruningNS και Sweep για να ανακτηθούν όλοι οι κοντινότεροι ορατοί γείτονες. Το μέγεθος του συνόλου δεδομένων ορίζεται στα 100000 MBRs και χρησιμοποιείται το ίδιο σύνολο σημείων ερωτημάτων όπως και στα προηγούμενα πειράματα με τεχνητά δεδομένα. Το μέγεθος των αντικειμένων μεταβάλλεται από [2, 2], [4, 4] έως [6, 6]. Οι συγκρίσεις των αλγορίθμων θα γίνουν ως προς το κόστος εισόδου - εξόδου, το κόστος μνήμης και το κόστος επεξεργασίας.

Σε όρους κόστους εισόδου - εξόδου, αύξηση του μεγέθους των αντικειμένων σημαίνει καλύτερη απόδοση για όλους τους αλγορίθμους. Με σταθερό το πλήθος των αντικειμένων, όσο αυξάνεται το μέγεθος των αντικειμένων πυκνώνει ο χώρος αναζήτησης και όλο και λιγότεροι κοντινότεροι γείτονες είναι ορατοί με αποτέλεσμα να μικραίνει ο αριθμός των αντικειμένων που χρειάζεται να ανακτηθούν από το δίσκο. Επομένως έχουμε λιγότερες προσπελάσεις στο δίσκο.

Όταν το μέγεθος των αντικειμένων είναι μικρό [2, 2], ο Sweep έχει μικρότερο χρόνο εκτέλεσης από τον PostPruningNS. Όταν τα αντικείμενα είναι μικρά είναι πιθανότερο στο σύνολο των αποτελεσμάτων να υπάρχουν γωνίες χωρίς κανένα κοντινότερο ορατό γείτονα. Ο Sweep επισκέπτεται κόμβους και αντικείμενα του R-tree σύμφωνα με την αύξουσα γωνία έναρξής τους. Αυτό το χαρακτηριστικό του δίνει τη δυνατότητα να ξεπερνάει αυτές τις οπές (περιοχές χωρίς κανένα NS) στο σύνολο των αποτελεσμάτων. Από την άλλη πλευρά ο αλγόριθμος Ripple και κατ' επέκταση οι αλγόριθμοι PostPruningNS και PrePruningNS που δημιουργήθηκαν από αυτόν, όταν υπάρχουν οπές συνεχίζουν τυφλά να εξετάζουν τις εισόδους της ουράς αυξάνοντας το κόστος υπολογισμού.

Επιπρόσθετα στην περίπτωση μικρού μεγέθους αντικειμένων έχουμε πολλούς κοντινότερους ορατούς γείτονες και επομένως πολλά αντικείμενα για να υπολογίσει ο PostPruningNS την ελάχιστη ορατή απόσταση, γεγονός που κοστίζει στον PostPruningNS σε χρόνο εκτέλεσης. Ο PrePruningNS από την άλλη πλευρά υπολογίζει την ελάχιστη ορατή απόσταση και τον κόμβων, δηλαδή σε υψηλότερο επίπεδο από τα αντικείμενα, με αποτέλεσμα να κλαδεύει αποδοτικά το σύνολο των αντικειμένων και έτσι έχει μικρότερο κόστος ακόμα και από τον Sweep που έχει την ικανότητα να ξεπερνάει τις οπές. Αυξανόμενου του μεγέθους των αντικειμένων το κόστος των PostPruningNS και PrePruningNS μειώνεται διότι όπως είπαμε, ο χώρος είναι πιο πυκνός και ανακτούνται λιγότεροι κοντινότεροι ορατοί γείτονες. Δεν συμβαίνει το ίδιο και για τον Sweep.

Όσο αυξάνεται το μέγεθος των αντικειμένων ο χρόνος εκτέλεσης του Sweep αυξάνεται. Το μεγαλύτερο υπολογιστικό κόστος στον Sweep το προκαλεί η συνάρτηση ObjectCompare. Όταν μια είσοδος που ανακτάται από την ουρά προτεραιότητας είναι αντικείμενο, συγκρίνεται με το τρέχον σύνολο των NS αποτελεσμάτων για την κοινή γωνία που έχει η είσοδος με τα αντικείμενα του αποτελέσματος. Αυτή τη σύγκριση την κάνει η συνάρτηση ObjectCompare.



Εικόνα 7.6. Επίδραση του μεγέθους των αντικειμένων σε τεχνητά δεδομένα με 100000 MBRs.

Όταν τα αντικείμενα μεγαλώνουν σε μέγεθος αυξάνεται το γωνιακό τους εύρος. Αυτό σημαίνει ότι είναι πιο πιθανό να έχουν γωνίες που επικαλύπτονται με γωνίες αντικείμενων του τρέχοντος αποτελέσματος, πράγμα που σημαίνει περισσότερες κλήσεις της συνάρτησης ObjectCompare και επομένως αύξηση του υπολογιστικού κόστους.

Όμως συνάρτηση ObjectCompare έχουν και οι αλγόριθμοι PrePruningNS και PostPruningNS. Η έλεγχος της ελάχιστης ορατής απόστασης αυτών των δύο αλγορίθμων γίνεται πριν την κλήση της συνάρτησης ObjectCompare. Επομένως γίνεται ένα καλό κλάδεμα των αντικειμένων ή/και τον κόμβων πριν κληθεί η ObjectCompare και έτσι οι κλήσεις τις τελευταίας συνάρτησης ελαχιστοποιούνται. Το όφελος από τον έλεγχο της ελάχιστης ορατής απόστασης υπερισχύει του κόστους της συνάρτησης ObjectCompare. Συνεπώς οι PrePruningNS και PostPruningNS έχουν μικρό υπολογιστικό κόστος.

Σε όρους μήκους της ουράς προτεραιότητας αυξανόμενου του μεγέθους των αντικειμένων το κόστος των PrePruningNS και PostPruningNS μειώνεται ενώ το κόστος του Sweep αυξάνει. Όπως είδαμε όταν πυκνώνει ο χώρος αναζήτησης μειώνεται ο αριθμός των κοντινότερων ορατών αντικειμένων άρα και ο αριθμός των κόμβων ή/και αντικειμένων που πρέπει να εξεταστούν έτσι η ουρά προτεραιότητας που γεμίζει αναδρομικά με αντικείμενα μειώνεται.

Παρόλο που οι κοντινότεροι ορατοί γείτονες μειώνονται και για τον αλγόριθμο Sweep στην εικόνα 7.6 παρατηρούμε ότι το μήκος της ουράς για τον Sweep αυξάνει. Αυτό συμβαίνει διότι το μεγαλύτερο μέγεθος των αντικειμένων δεν επιτρέπει στη συνάρτηση Lookahead, η οποία υπάρχει μόνο στον Sweep και όχι στον Ripple, να λειτουργήσει αποδοτικά. Η συνάρτηση Lookahead καλείται από τον Sweep για να επιλέξει την καλύτερη είσοδο της ουράς προτεραιότητας σύμφωνα με κάποιες ευρετικές μεθόδους. Καλύτερη είσοδος είναι αυτή που είναι πιο πιθανό, σύμφωνα με κάποιες μεθόδους, να περιέχει κάποιο NS αντικείμενο. Η συνάρτηση Lookahead δίνει τη μεγαλύτερη προτεραιότητα στις εισόδους της ουράς οι οποίες καλύπτονται πλήρως από το γωνιακό εύρος όλων των τρεχόντων NS αντικειμένων. Μεγάλο μέγεθος αντικειμένων συνεπάγεται μεγαλύτερο γωνιακό εύρος για το τρέχον NS σύνολο αποτελεσμάτων. Επομένως είναι πιο πιθανό να βρίσκονται στην ουρά προτεραιότητας είσοδοι που καλύπτονται πλήρως από το γωνιακό εύρος του τρέχοντος συνόλου, χωρίς αυτό να σημαίνει ότι είναι και οι καλύτεροι είσοδοι που πρέπει να επιστραφούν από τη συνάρτηση Lookahead. Αυτό έχει σαν αποτέλεσμα να εξερευνούνται άσκοπα περισσότεροι είσοδοι της ουράς προτεραιότητας και άρα να αυξάνεται το μήκος της, δηλαδή το κόστος μνήμης.

8

Συμπεράσματα

Στο κεφάλαιο 2 παρουσιάστηκαν τα διαφορετικού τύπου ερωτήματα οπτικού πεδίου που έχουν προταθεί και ερευνηθεί από την επιστημονική κοινότητα. Αυτά είναι τα ερωτήματα: (i) κοντινότερου περιβάλλοντος (NS), (ii) κοντινότερου ορατού γείτονα (VNN), (iii) αντίστροφου κοντινότερου ορατού γείτονα (VRNN), (iv) συνεχούς κοντινότερου γείτονα (CNN), (v) συνεχούς κοντινότερου ορατού γείτονα (CVNN), (vi) συνολικά κοντινότερου ορατού γείτονα (AVNN), (vii) εύρους (Range Queries), (viii) όλων των κοντινότερων γειτόνων (All-NN), (ix) k κοντινότερων ζευγών (K-Closest Pairs) και (x) ερωτήματα με αντικείμενα και θέσεις ερωτημάτων σε κίνηση. Ακολούθως, μελετήθηκαν διεξοδικά έξι από τις παραπάνω προσεγγίσεις διαφορετικού τύπου ερωτημάτων οπτικού πεδίου: NS, VkNN, VRNN, CVNN, AVNN και NS σε κινούμενα περιβάλλοντα.

Στο κεφάλαιο 3 παρουσιάστηκαν συγκριτικά τα διαφορετικού τύπου ερωτήματα του προηγούμενου κεφαλαίου. Η αξιολόγηση έγινε ως προς ορισμένα βασικά χαρακτηριστικά του πραγματικού χώρου στα οποία δίνουν σημασία τα ερωτήματα και επιπλέον ως προς συγκεκριμένα χαρακτηριστικά της οντότητας που θέτει το ερώτημα σε κάθε τύπο ερωτήματος. Από τη σύγκριση επιλέχτηκαν οι πρακτικές των άρθρων NS [3] και VkNN [6], για να επεκταθούν. Ο λόγος αυτής της επιλογής είναι ότι και οι δύο πρακτικές αναφέρονται σε σταθερή και σημειακή θέση ερωτήματος, δηλαδή μοιάζουν ως προς τη βάση και διαφέρουν ως προς τα χαρακτηριστικά του πραγματικού κόσμου που υπολογίζουν. Το ερώτημα NS υπολογίζει τη γωνία των αντικειμένων και το VkNN επιτρέπει στο χρήστη να επιλέξει το πλήθος των κοντινότερων ορατών γειτόνων, ενώ και τα δύο ερωτήματα λαμβάνουν υπόψη την ορατότητα των αντικειμένων. Με την επέκταση, ο NS αλγόριθμος δίνει στο χρήστη την επιλογή του πλήθους των κοντινότερων γειτόνων και ο VkNN επιστρέφει τη γωνία των αντικειμένων.

Για την υλοποίηση και επέκταση των VkNN ερωτημάτων του άρθρου [6] επιλέχτηκε να χρησιμοποιηθεί ως βάση ο κώδικας των NS ερωτημάτων. Ο αλγόριθμος Ripple των ερωτημάτων NS μπορεί να τροποποιηθεί ώστε να ανακτά σταδιακά τους k κοντινότερους γείτονες. Αυτή η τροποποίηση έγινε υπολογίζοντας την ελάχιστη ορατή απόστασης των αντικειμένων και επανεισάγοντας

τα αντικείμενα που μπορεί να μην είναι ο επόμενος VNN, στην ουρά προτεραιότητας.

Στο κεφάλαιο 4, σχεδιάστηκε η αρχιτεκτονική των συστημάτων NS και VkNN ώστε να διαπιστωθούν τα σημεία στα οποία πρέπει να επέμβουμε για να υλοποιηθούν οι απαραίτητες αλλαγές για την επέκτασή τους. Στο κεφάλαιο 5 αναλύονται οι νέες προσεγγίσεις που υλοποιήθηκαν.

Η ομοιότητα των αλγορίθμων Ripple και PostPruning είναι ότι και οι δύο αρχικοποιούν την ουρά προτεραιότητας με τη ρίζα του R-tree ευρετηρίου και στη συνέχεια τη γεμίζουν αναδρομικά με κόμβους και αντικείμενα. Όταν ανακτούν μια είσοδο από την ουρά προτεραιότητας ελέγχουν αν η είσοδος αυτή είναι κόμβος ή αντικείμενο. Ο PostPruning όταν βρίσκει ένα αντικείμενο ελέγχει την ελάχιστη ορατή απόστασή του και τη συγκρίνει με την απόσταση της κορυφής της ουράς. Αν το προς εξέταση αντικείμενο έχει μεγαλύτερη ορατή απόσταση από την κορυφή της ουράς επανεισάγεται στην ουρά αλλιώς μπορεί να αποτελέσει μέρος του αποτελέσματος. Ο Ripple δεν έχει αυτόν τον έλεγχο της ελάχιστης ορατής απόστασης και όταν βρίσκει ένα αντικείμενο καλεί μια συνάρτηση που ανανεώνει το NS αποτέλεσμα. Συνεπώς, εισήχθηκε στον Ripple αυτός ο έλεγχος ελάχιστης ορατής απόστασης, όταν βρίσκει κάποιο αντικείμενο και πριν καλέσει τη συνάρτηση ανανέωσης του αποτελέσματος. Για να πραγματοποιηθεί αυτός ο έλεγχος υλοποιήθηκε μια συνάρτηση υπολογισμού της ελάχιστης ορατής απόστασης.

Διατηρώντας τις ιδιότητες της γωνίας και της συνθήκης τερματισμού του Ripple επεκτάθηκε ο αλγόριθμος αυτός στον PostPruning-NS-TC, ο οποίος επιστρέφει και τη γωνία των αντικειμένων. Μεταφέροντας τον έλεγχο της ελάχιστης ορατής απόστασης κατά την ανάκτηση των εισόδων από την ουρά και πριν ελεγχθεί αν η είσοδος είναι κόμβος ή αντικείμενο έχουμε την υλοποίηση του αλγορίθμου PrePruning-NS-TC. Επιπλέον, για να δοθεί στο χρήστη η δυνατότητα επιλογής των k κοντινότερων γειτόνων προστέθηκε ένας έλεγχος που εξετάζει την ουρά αναδρομικά μέχρι να υπάρξουν k αντικείμενα στο τρέχον αποτέλεσμα.

Μέχρι αυτό το σημείο έχει υλοποιηθεί το ένα σκέλος της επέκτασης, δηλαδή τα VkNN ερωτήματα που επιστρέφουν τη γωνία. Για την επέκταση των NS ερωτημάτων ώστε να επιστρέφουν τους k κοντινότερους γείτονες επιλέχθηκε τυχαία από τους Sweep και Ripple, ο αλγόριθμος Sweep. Διαπιστώθηκε ότι ο αλγόριθμος αυτός δεν μπορεί να ανακτήσει σταδιακά τους VkNN καθώς εξετάζει τα αντικείμενα του χώρου ταξινομημένα σύμφωνα με την αύξουσα γωνία έναρξής τους. Έτσι για την επιλογή των VkNN ταξινομούνται τα τελικά αποτελέσματα του Sweep σύμφωνα με την αύξουσα απόστασή τους από το σημείο ερωτήματος και επιλέγονται τα k από αυτά. Έτσι ολοκληρώθηκε η επέκταση των αλγορίθμων.

Στο κεφάλαιο 6 περιγράφονται οι λεπτομέρειες της εκτέλεσης των αλγορίθμων που υλοποιήθηκαν. Στο κεφάλαιο 7 διεξήχθη η πειραματική μελέτη των αλγορίθμων PostPruning(k)NS, PrePruning(k)NS και Sweep(k)NS σε όρους κόστους εισόδου - εξόδου (page accesses), χρόνου εκτέλεσης (CPU time) και μήκους ουράς προτεραιότητας (queue length). Στα ερωτήματα οπτικού πεδίου

η ουρά προτεραιότητας καταναλώνει την περισσότερη μνήμη και έτσι χρησιμοποιείται το μήκος της για την μέτρηση του κόστους μνήμης.

Για τον αλγόριθμο SweepkNS διαπιστώθηκε ότι δεν επηρεάζεται από τις μεταβολές του k, εφόσον δεν μπορεί να ανακτήσει σταδιακά τους VkNN αλλά πρέπει σε κάθε περίπτωση να ανακτήσει όλους τους VNN του συνόλου δεδομένων. Από τα πειράματα όπου μεταβάλλεται το πλήθος των δεδομένων, διαπιστώθηκε ότι με την αύξηση του συνόλου δεδομένων η απόδοση του Sweep χειροτερεύει, για όλα τα κριτήρια κόστους. Αυτό συμβαίνει διότι ο Sweep δεν έχει τη συνθήκη τερματισμού NS-TC που έχουν οι αλγόριθμοι PostPruning(k)NS και PrePruning(k)NS. Το αποτέλεσμα είναι να εξετάζονται εξαντλητικά όλες οι είσοδοι της ουράς προτεραιότητας ακόμα και όταν όλοι οι VNNs έχουν βρεθεί. Αύξηση του πλήθους των αντικειμένων συνεπάγεται αύξηση του μεγέθους του R-tree, άρα περισσότεροι είσοδοι για να εξετάσει ο Sweep, με συνέπεια να μειώνεται η απόδοσή του.

Όταν αυξάνεται το μέγεθος των αντικειμένων το κόστος εισόδου - εξόδου του Sweep μειώνεται γιατί μεγαλώνει η πυκνότητα του χώρου και όλο και λιγότερα αντικείμενα είναι ορατά. Όμως το κόστος επεξεργασίας του Sweep αυξάνεται με την αύξηση του μεγέθους των αντικειμένων διότι αυξάνεται το γωνιακό εύρος των αντικειμένων και επομένως και οι επικαλύψεις των γωνιών μεταξύ του αντικειμένου που εξετάζεται κάθε φορά και των αντικειμένων του τρέχοντος αποτελέσματος. Έτσι ο αλγόριθμος αυτός έχει να κάνει περισσότερες συγκρίσεις μεταξύ των αντικειμένων για να επιλέξει τους κοντινότερους γείτονες. Επιπλέον και το κόστος μνήμης του Sweep αυξάνεται με την αύξηση του μεγέθους των αντικειμένων επειδή το μεγάλο γωνιακό εύρος των αντικειμένων δεν επιτρέπει στη συνάρτηση κλαδέματος του Sweep να λειτουργήσει αποδοτικά.

Υπάρχει μια περίπτωση που ο αλγόριθμος Sweep(k)NS είναι πιο αποδοτικός σε κόστος επεξεργασίας από τους PostPruning(k)NS και PrePruning(k)NS. Αυτή η περίπτωση ισχύει όταν ο χώρος αναζήτησης είναι αραιός δηλαδή υπάρχουν κενές γωνίες στο σύνολο των αποτελεσμάτων. Τότε, ο Sweep έχει τη δυνατότητα να προσπερνάει αυτά τα κενά και να μην ψάχνει εξαντλητικά την ουρά για αντικείμενα που αλληλεπιδρούν με αυτά τα κενά όπως κάνουν οι άλλοι δύο αλγόριθμοι. Σε αυτή την περίπτωση επίσης, ο αλγόριθμος Sweep έχει την ίδια απόδοση με τον PostPruningNS σε κόστος εισόδου - εξόδου.

Το κόστος του PostPruningkNS αυξάνεται με την αύξηση της παραμέτρου k (πλήθος VNNs που επιλέγει ο χρήστης) επειδή αυξάνεται ο αριθμός των VNN που πρέπει να ανακτηθούν και έτσι ο αλγόριθμος λαμβάνει υπόψη του περισσότερες εισόδους της ουράς. Με την αύξηση του πλήθους των συνόλων δεδομένων το κόστος εισόδου - εξόδου και το κόστος επεξεργασίας του PostPruningNS μειώνεται γιατί υπάρχει αρνητική σχέση μεταξύ της αύξησης του αριθμού των αντικειμένων και των VNNs. Επομένως, όλο και λιγότεροι γείτονες είναι ορατοί όταν αυξάνεται το σύνολο δεδομένων. Έτσι ο PostPruningNS έχει λιγότερους ορατούς γείτονες για να υπολογίσει. Το κόστος μνήμης αυξάνεται λίγο για τον PostPruningNS διότι μεγαλύτερο σύνολο δεδομένων σημαίνει περισσότερα παιδιά για κάθε κόμβο του R-tree και επομένως περισσότερα παιδιά για να γεμίσει

η ουρά προτεραιότητας. Επίσης επειδή ο έλεγχος της ελάχιστης ορατής απόστασης των αντικειμένων επιτυγχάνει καλό κλάδεμα του χώρου αναζήτησης, η αύξηση του κόστους μνήμης που παρατηρείται είναι μικρή.

Η πυκνότητα του χώρου αναζήτησης αυξάνει τόσο με την αύξηση του πλήθους των αντικειμένων όσο και με την αύξηση του μεγέθους των αντικειμένων. Επακόλουθα, με την αύξηση του μεγέθους των αντικείμενων η ορατή περιοχή μειώνεται και επομένως και οι VNNs. Για αυτό το λόγο οι μετρήσεις απόδοσης του PostPruningNS καλυτερεύουν όσο αυξάνεται το μέγεθος των αντικειμένων.

Μόνο όταν το μέγεθος των αντικειμένων είναι μικρό ο Sweep αποδίδει λίγο καλύτερα από τον PostPruningNS επειδή, όπως προαναφέρθηκε, ο PostPruningNS δεν έχει την ικανότητα να προσπερνά τις οπές στο σύνολο των αποτελεσμάτων όπως ο Sweep. Επιπλέον, επειδή με μικρό μέγεθος αντικειμένων υπάρχουν πολλοί VNNs, ο υπολογισμός της ελάχιστης ορατής απόστασης των αντικειμένων κοστίζει στον PostPruningNS σε χρόνο επεξεργασίας. Γενικότερα, ο PostPruningNS αποδίδει καλύτερα από τον Sweep και σε ορισμένες περιπτώσεις με μεγάλη διαφορά κόστους. Η μεγάλη διαφορά κόστους οφείλεται στη συνθήκη τερματισμού NS-TC που έχει ο PostPruningNS και επιπλέον επειδή με τον έλεγχο της ελάχιστης ορατής απόστασης επιτυγχάνει καλό κλάδεμα του χώρου αναζήτησης.

Για τον αλγόριθμο PrePruning(k)NS ισχύουν παρόμοια συμπεράσματα με τον αλγόριθμο PostPruning(k)NS. Ο PrePruning(k)NS έχει την ίδια συμπεριφορά με τον PostPruning(k)NS αλλά καλύτερη απόδοση σε όλες σχεδόν τις μετρήσεις. Αυτό συμβαίνει επειδή ο PrePruning(k)NS εφαρμόζει τον έλεγχο της ελάχιστης ορατής απόστασης όχι μόνο στα αντικείμενα όπως ο PostPruning(k)NS, αλλά και στους κόμβους του R-tree. Με αυτό τον τρόπο επιτυγχάνει καλύτερο κλάδεμα του χώρου αναζήτησης. Σε όρους κόστους εισόδου - εξόδου αποδείχτηκε στο άρθρο [6] ότι είναι βέλτιστος, γεγονός που φάνηκε και πειραματικά. Για μικρές τιμές της παραμέτρου k έχει λίγο μεγαλύτερο κόστος επεξεργασίας από τον PostPruningkNS. Αυτό συμβαίνει επειδή υπολογίζει την ελάχιστη ορατή απόσταση για περισσότερες οντότητες από τον PostPruningkNS. Όμως όσο αυξάνει το k το κόστος του PostPruningNS αυξάνει πιο γρήγορα από αυτό του PrePruningNS, σε όλες τις μετρήσεις απόδοσης. Γενικότερα το κλάδεμα του χώρου αναζήτησης που επιτυγχάνει ο PrePruning(k)NS υπερκεράζει το επιπλέον κόστος επεξεργασίας από τον υπολογισμό της ελάχιστης ορατής απόστασης και έτσι εμφανίζεται με καλύτερες αποδόσεις, σε όλες τις μετρήσεις, από τους PostPruning(k)NS και Sweep(k)NS.

Συνοπτικά, όταν στο σύνολο δεδομένων υπάρχουν πολλές γωνίες χωρίς κανέναν κοντινότερο γείτονα, ο Sweep(k)NS έχει μικρότερο χρόνο επεξεργασίας από τον PostPruning(k)NS και ίσο με τον PrePruning(k)NS. Σε όλες τις άλλες περιπτώσεις οι PostPruning(k)NS και PrePruning(k)NS έχουν εξαιρετικά καλύτερη απόδοση, με τον PrePruning(k)NS να είναι καλύτερος από PostPruning(k)NS.

A

Παράρτημα Υλοποίησης των VkNS Ερωτημάτων

Στις επόμενες ενότητες παρουσιάζεται ο κώδικας που υλοποιήθηκε.

A.1 Κλάση prepruningkns.cc

Η κλάση που περιέχει τη συνάρτηση main των αλγορίθμων PostPruning(k)NS και PrePruning(k)NS είναι ίδια. Η μόνη διαφορά είναι μια γραμμή κώδικα που καλεί τον αντίστοιχο αλγόριθμο PostPruning(k)NS ή PrePruning(k)NS. Για λόγους απλοποίησης παρουσιάζεται η κλάση με τη main συνάρτηση μόνο του PrePruningkNS κώδικα.

```
/* ---------------------------------------------------------------------------
Author: Anna Giannoutsou
Email:  anna.giannoutsou@gmail.com
Date:   July, 2011

This program:
1. load an Rtree indexing polygons
2. load a collection of polygons
3. performs nearest surrounder query based on PRE-PRUNING algorithm
4. output the result

Suggested arguments:
> (prog name) -p 4096 -f raw_data.txt -i index_file.idx
  -q queryfile.txt
  -a0 startangle -a1 endangle
  -m #tiers -t #sectors -v true -k #neighbors

explanations:
-p: pagesize, typically, 4096
-f: a tab delimited file, format:
    id #vertice x0 y0 x1 y1 ... xn yn
-i: index file
```

```
-q: query point file
-a0: start angle (default: 0)
-a1: end angle (default: 360)
-m: #tiers (default: 1)
-t: #sectors (default: 1)
-v: verbose mode on
-k: #neighbors to search
------------------------------------------------------------------------------ */

//#include space...

using namespace std;

#define MAXEDGE 1000
#define DIMEN 2

//-----------------------------------------------------------------------
//This function draws the result (objects and query point) at an .eps file,
//Draws the VNNs red and the rest of the objects black. The query point is
//represented by a blue dot.
//-----------------------------------------------------------------------
void drawResult(const Rtree& a_rtree,
          const int a_bottomlevel, const int a_toplevel,
          const char* epsname, vector<int> objid, float c[DIMEN])
{
   RtreeNode* root = a_rtree.m_memory.loadPage(a_rtree.m_memory.m_rootPageID);
   RtreeNodeEntry* entry = root->genNodeEntry();
   Hypercube& hc = entry->m_hc;

   PSDraw psdraw(epsname, hc.getLower()[0], hc.getLower()[1],
                  hc.getUpper()[0], hc.getUpper()[1]);

   const int top = a_toplevel < root->m_level ? a_toplevel : root->m_level;
   delete entry;
   delete root;
   Stack s;
   s.push((void*)a_rtree.m_memory.m_rootPageID);

   while (!s.isEmpty())
   {
     int pageid = (int)s.pop();
     RtreeNode* n = a_rtree.m_memory.loadPage(pageid);
     if (a_bottomlevel <= n->m_level && n->m_level <= a_toplevel)
     {
        for (int i=0; i<n->m_usedSpace; i++)
        {
          int aid = n->m_entry[i]->m_id;
          bool found = false;
          for (vector<int>::size_type h = 0; h < objid.size(); h++)
```

```
            {
              if(objid[h] == aid){
                 found = true;
                 psdraw.box(n->m_entry[i]->m_hc, 0, 1, true);
                 break;
              }
            }
            if(!found)
                psdraw.box(n->m_entry[i]->m_hc, 0, 1, false);
           }
      }
      if (n->m_level > a_bottomlevel)
      {
         for (int i=0; i<n->m_usedSpace; i++)
            s.push((void*)n->m_entry[i]->m_id);
      }
      delete n;
    }
    psdraw.queryPoint(c[0], c[1], true);
}


void helpmsg(const char* pgm)
{
    cerr << "Suggested arguments:" << endl;
    cerr << "> " << pgm << endl;
    cerr << "-p 4096 -d 2 -f raw_polygon.txt -i index_file.idx " << endl;
    cerr << "-q queryfile.txt -a0 startangle -a1 endangle " << endl;
    cerr << "-m tier -t #sectors -v true -k #neighbors" << endl << endl;
    cerr << "explanations:" << endl;
    cerr << "-p: pagesize, typically, 4096" << endl;
    cerr << "-f: raw polygon data file" << endl;
    cerr << "a tab delimited file, format:" << endl;
    cerr << "id #vertice x0 y0 x1 y1 ... xn yn" << endl;
    cerr << "-i: index file (rtree)" << endl;
    cerr << "-q: query file, format" << endl;
    cerr << "id x y" << endl;cerr << "-a0: start angle (default: 0)" << endl;
    cerr << "-a1: end angle (default: 360)" << endl;
    cerr << "-m: tier (default: 1)" << endl;
    cerr << "-t: no. of sectors for partitioned processing (default: 1)" << endl;
    cerr << "-v: verbose mode on" << endl;
    cerr << "-k: number of neighbours to find (default: 1)" << endl;
}


int main(const int a_argc, const char** a_argv)
{
  if (a_argc == 1)
  {
```

```
    helpmsg(a_argv[0]);
    return -1;
  }

  cerr << "PrePruning k NSQ algorithm" << endl;
  //-------------------------------------------------------------------------
  // initialization
  //-------------------------------------------------------------------------
  int pagesize = atol(Param::read(a_argc, a_argv, "-p", ""));
  const char* filename = Param::read(a_argc, a_argv, "-f", "");
  const char* idxname = Param::read(a_argc, a_argv, "-i", "");
  const char* qryfilename = Param::read(a_argc, a_argv, "-q", "");
  const float startangle = (float)atof(Param::read(a_argc, a_argv, "-a0", "0.0"));
  const float endangle = (float)atof(Param::read(a_argc, a_argv, "-a1", "360.0"));
  const int m = atol(Param::read(a_argc, a_argv, "-m", "1"));
  const int t = atol(Param::read(a_argc, a_argv, "-t", "1"));
  const char* vrbs = Param::read(a_argc, a_argv, "-v", "null");
  bool verbose = strcmp(vrbs,"null") != 0;
  const int k = atol(Param::read(a_argc, a_argv, "-k", "1"));

  //-------------------------------------------------------------------------
  // load an R-tree from an index file
  //-------------------------------------------------------------------------
  cerr << "loading Rtree ... ";
  const int maxChild = (pagesize - RtreeNode::size()) /
          RtreeNodeEntry::size(DIMEN);// no. of entries per node
  FileMemory mem(pagesize, idxname, RtreeNodeEntry::fromMem, false);
  Rtree rtree(mem, DIMEN, maxChild, maxChild,
          (int)(maxChild*0.3), (int)(maxChild*0.3), false);

  cerr << "[DONE]" << endl;

  //-------------------------------------------------
  // load polygons
  //-------------------------------------------------
  cerr << "loading polygons ... ";
  fstream fin;
  fin.open(filename, ios::in);
  Hash hPolygon(1000000);
  while (true)
  {
     int id, edge;
     float sx[MAXEDGE], sy[MAXEDGE];
     float ex[MAXEDGE], ey[MAXEDGE];

   //-------------------------------------------------
   // read a polygon record
   //-------------------------------------------------
   fin >> id;
```

```
  if (fin.eof()) break;  // break if eof
  fin >> edge;
  for (int e=0; e<edge; e++)
  {
     fin >> sx[e];
     fin >> sy[e];
     fin >> ex[e];
     fin >> ey[e];
  }

  //-----------------------------------------------
  // form a polygon
  //-----------------------------------------------
  Polygon* plg = new Polygon;
  for (int e=0; e<edge; e++)
  {
     float c[2];
     c[0] = sx[e]; c[1] = sy[e];
     Point ptl(DIMEN, c);
     c[0] = ex[e]; c[1] = ey[e];
     Point ptu(DIMEN, c);
     LineSeg l(ptl, ptu);
     plg->addLineSeg(l);
  }
  hPolygon.put(id, plg);
}
fin.close();
cerr << hPolygon.size() << " polygon objects are loaded." << endl;

//-----------------------------------------------------
// initialize the performance measurement metrics
//-----------------------------------------------------
int numquery = 0;
int totalobj = 0;

float aveProcTime = 0;  // average performance
float aveObjExam = 0;
float aveMemSize = 0;
float aveNumPage0 = 0;
float aveNumPage10 = 0;
float aveNumPage30 = 0;
float aveNumPage50 = 0;
float aveNumObjs[10];  // assume at most 10 tiers

float minProcTime = INFTY;// minimum performance
int minObjExam = INFTY;
int minMemSize = INFTY;
int minNumPage0 = INFTY;
int minNumPage10 = INFTY;
```

```
int minNumPage30 = INFTY;
int minNumPage50 = INFTY;
int minNumObjs[10];

float maxProcTime = -INFTY; // maximum performance
int maxObjExam = -INFTY;
int maxMemSize = -INFTY;
int maxNumPage0 = -INFTY;
int maxNumPage10 = -INFTY;
int maxNumPage30 = -INFTY;
int maxNumPage50 = -INFTY;
int maxNumObjs[10];

for (int i=0; i<m; i++)
{
  aveNumObjs[i] = 0;
  minNumObjs[i] = INFTY;
  maxNumObjs[i] = -INFTY;
}

//-----------------------------------------------------------------
// process PrePruning NSQ algorithm based on read-in query points
//-----------------------------------------------------------------
fstream fqin;
fqin.open(qryfilename, ios::in);
int fileDraw = 0;

while (true)
{
  int id;
  float c[DIMEN];

  //--------------------------------
  // load a query point
  //--------------------------------
  fqin >> id;
  if (fqin.eof()) break;
  fqin >> c[0];
  fqin >> c[1];
  Point q(DIMEN, c);

  //---------------------------------
  // perform the search
  //---------------------------------
  Array res[10];
  int maxexam = 0;
  int maxqlen = 0;
  Array pageaccessed;
  float avewidth = (endangle - startangle)/t;
```

```
 struct timeb starttime, endtime;
 ftime(&starttime);

 for (int i=0; i<t; i++)
 {
    AngleRange ar(i*avewidth+startangle, (i+1)*avewidth+startangle);
    NSSearch::prepruning(rtree, hPolygon, q, ar, m, res[i],
                         maxexam, maxqlen, pageaccessed, k);
 }
 ftime(&endtime);
 float qtime = ((endtime.time*1000 + endtime.millitm) -
         (starttime.time*1000 + starttime.millitm)) / 1000.0f;


//---------------------------------------------------------------------
// This loop is used to print the results uniformly, with the VNN and
// its corresponding whole angular range.
//---------------------------------------------------------------------
if (verbose)
{
  cerr << "q@: " << c[0] << "," << c[1] << ", id:"<<id<<endl;

  for (int i=0; i<m; i++)
  {
    for (int l=0; l<t; l++)
    {
       AngleRange angl;
       int lastobj = -100;
       RippleNSResult* rnsr = (RippleNSResult*)res[l].get(i);
       for (int j=0; j<rnsr->m_numsect; j++)
       {
          for (int k=0; k<rnsr->m_section[j]->size(); k++)
          {
            TierNSResult* r = (TierNSResult*)rnsr->m_section[j]->get(k);

            if (lastobj == -100){
               angl.set(r->m_ar.start(), r->m_ar.end());
            }
            else if (lastobj == r->m_oid && r->m_ar.end()!=360){
               angl.set(angl.start(), r->m_ar.end());
            }
            else if (lastobj == r->m_oid && r->m_ar.end()==360){
               cerr << "("<<angl.start()<<", "<<r->m_ar.end()<<"):"<< lastobj<<endl;
            }
            else if (lastobj != r->m_oid && r->m_ar.end()==360){
               cerr << "("<<angl.start()<<", "<<angl.end()<<"):"<< lastobj<<endl;
               cerr << "("<<r->m_ar.start()<<", "<<r->m_ar.end()<<"):"<< r->m_oid<<endl;
            }
```

```
            else if (lastobj != r->m_oid){
               cerr << "("<<angl.start()<<", "<<angl.end()<<"):"<< lastobj<<endl;
               angl.set(r->m_ar.start(), r->m_ar.end());
            }
            lastobj = r->m_oid;
          }
        }
      }
    }
  }
  cerr << "qtime: " << qtime;
  cerr << ", io: " << pageaccessed.size();
  cerr << ", #res: ";

  int tier[5], obj[5];
  for (int i=0; i<m; i++)
      tier[i] = obj[i] = 0;

  Set allobjid;
  vector<int> myVector;
  for (int i=0; i<m; i++)
  {
     Set objid;
     int prevoid = -100;
     for (int l=0; l<t; l++)
     {
        RippleNSResult* rnsr = (RippleNSResult*)res[l].get(i);
        for (int j=0; j<rnsr->m_numsect; j++)
        {
           for (int k=0; k<rnsr->m_section[j]->size(); k++)
           {
              TierNSResult* r = (TierNSResult*)rnsr->m_section[j]->get(k);
              if (r->m_ar.width() == 0) continue;
              if (r->m_oid != prevoid)
              {
                 tier[i]++;
                 if (prevoid >= 0)
                 {
                    objid.insert((void*)prevoid);
                    allobjid.insert((void*)prevoid);
                    myVector.push_back(prevoid);
                 }
                 prevoid = r->m_oid;
              }
           }
        }
     }
     if (prevoid >= 0)
     {
```

```
        objid.insert((void*)prevoid);
        allobjid.insert((void*)prevoid);
        myVector.push_back(prevoid);
    }

    tier[i]++;
    obj[i] = objid.size();
    cerr << "(tier-" << i << ") " << tier[i] << "(" << obj[i] << ") ";
}
totalobj = allobjid.size();
cerr << endl;
cout<<"---------------------------------------"<<endl;
//convert int to string
std::string s;
std::stringstream out;
out << fileDraw;
s = out.str();
//convert string to char*
const char *p;
p=s.c_str();
//concatanate chars
char str[100];
strcpy (str,"preprunkNSResult_");
strcat (str, p);
strcat(str, ".eps");

drawResult(rtree, 0, 100, str, myVector, c);

//---------------------------------------
// performance measurement
//---------------------------------------

// average performance
int io0 = pageaccessed.size();
int io10 = IOMeasure::lru(pageaccessed, 10);
int io30 = IOMeasure::lru(pageaccessed, 30);
int io50 = IOMeasure::lru(pageaccessed, 50);
aveProcTime += qtime;
aveNumPage0  += io0;
aveNumPage10 += io10;
aveNumPage30 += io30;
aveNumPage50 += io50;
for (int i=0; i<m; i++)
     aveNumObjs[i] += obj[i];
aveObjExam += maxexam;
aveMemSize += maxqlen;
numquery++;

// max performance
```

```
 maxProcTime  = maxProcTime > qtime ? maxProcTime : qtime;
 maxNumPage0  = maxNumPage0  > io0  ? maxNumPage0  : io0;
 maxNumPage10 = maxNumPage10 > io10 ? maxNumPage10 : io10;
 maxNumPage30 = maxNumPage30 > io30 ? maxNumPage30 : io30;
 maxNumPage50 = maxNumPage50 > io50 ? maxNumPage50 : io50;
 for (int i=0; i<m; i++)
     maxNumObjs[i] = maxNumObjs[i] > obj[i] ? maxNumObjs[i] : obj[i];
 maxObjExam = maxObjExam > maxexam ? maxObjExam : maxexam;
 maxMemSize = maxMemSize > maxqlen ? maxMemSize : maxqlen;

 // min performance
 minProcTime  = minProcTime < qtime ? minProcTime : qtime;
 minNumPage0  = minNumPage0  < io0  ? minNumPage0  : io0;
 minNumPage10 = minNumPage10 < io10 ? minNumPage10 : io10;
 minNumPage30 = minNumPage30 < io30 ? minNumPage30 : io30;
 minNumPage50 = minNumPage50 < io50 ? minNumPage50 : io50;
 for (int i=0; i<m; i++)
     minNumObjs[i] = minNumObjs[i] < obj[i] ? minNumObjs[i] : obj[i];
 minObjExam = minObjExam < maxexam ? minObjExam : maxexam;
 minMemSize = minMemSize < maxqlen ? minMemSize : maxqlen;

 //---------------------------------------
 // clean up
 //---------------------------------------
 for (int l=0; l<t; l++)
 {
    for (int i=0; i<m; i++)
    {
       RippleNSResult* rnsr = (RippleNSResult*)res[l].get(i);
       for (int j=0; j<rnsr->m_numsect; j++)
       {
          for (int k=0; k<rnsr->m_section[j]->size(); k++)
             delete (TierNSResult*)rnsr->m_section[j]->get(k);
       }
       delete rnsr;
    }
    res[l].clean();
 }
 pageaccessed.clean();

 fileDraw++;
}
fqin.close();


 cout << "---Prepruning kNS Performance Evaluation---" << endl;
 cout << "#query: " << numquery << endl;
 cout << "m: " << m << endl;
 cout << "t: " << t << endl;
```

```
 cout << "ar: " << startangle << "-" << endangle << endl;
 cout << "proctime(min, ave, max): " << minProcTime << ", " <<
             aveProcTime/numquery << ", " << maxProcTime << endl;
 cout << "objexam(min, ave, max): " << minObjExam << ", " <<
             aveObjExam/numquery << ", " << maxObjExam << endl;
 cout << "memsize(min, ave, max): " << minMemSize << ", " <<
            aveMemSize/numquery << ", " << maxMemSize << endl;
 cout << "io0(min, ave, max): " << minNumPage0 << ", " <<
             aveNumPage0/numquery << ", " << maxNumPage0 << endl;
 cout << "io10(min, ave, max): " << minNumPage10 << ", " <<
             aveNumPage10/numquery << ", " << maxNumPage10 << endl;
 cout << "io30(min, ave, max): " << minNumPage30 << ", " <<
             aveNumPage30/numquery << ", " << maxNumPage30 << endl;
 cout << "io50(min, ave, max): " << minNumPage50 << ", " <<
             aveNumPage50/numquery << ", " << maxNumPage50 << endl;
 cout << "totalobj: " << totalobj<< endl;
 for (int i=0; i<m; i++)
       cout << "res(min, ave, max): " << minNumObjs[i] << ", " <<
              aveNumObjs[i]/numquery << ", " << maxNumObjs[i] << endl;
 cout << endl;

 return 0;
}
```

A.2 Κλάση nsprepruning.cc

Η επόμενη κλάση υλοποιεί τον καθαυτό αλγόριθμο PrePruning(k)NS. Χάριν εξοικονόμησης χώρου έχει αφαιρεθεί από την κλάση αυτή το περιεχόμενο των συναρτήσεων που προϋπήρχαν και παρουσιάζονται μόνο οι συναρτήσεις που υλοποιήθηκαν στην παρούσα εργασία.

```
/* ----------------------------------------
    Author: Anna Giannoutsou
    Email:  anna.giannoutsou@gmail.com
    Date:   July, 2011
---------------------------------------- */
//#include space...

using namespace std;
#define NODE   0
#define EDGE   1
#define DIMEN  2
#define DUMMY  -1

float minViDistance;
```

```
AngleRange angRange;

//------------------------
// edge compare result
//------------------------
class EdgeCompResult


int preprunEdgeCompare(const int a_oid0, const LineSeg& a_lineseg0,
                       const AngleRange& a_ar0,   // lineseg0
                       const int a_oid1,
                       const LineSeg& a_lineseg1,
                       const AngleRange& a_ar1,   // lineseg1
                       const Point& a_pt,         // ref. point
                       Array& a_cmpres,
                       Array& a_hidden)           // comparison result


//-----------------------------------------------------------------------------
// determine a range of existng NS results whose angle ranges are covered by
// a provided angle range (angle range search)
//-----------------------------------------------------------------------------
int preprunTierLookup(Array& a_tier,   // a tier of NS result
            const AngleRange& a_ar,    // angle range of a candidate
            int& a_start,              // starting position
            nt& a_end)                 // ending position


//-----------------------------------------------------------------------------
// preliminary determine if an entry is hidden by existing edges
// true: hidden, and the entry can be discarded
// false: otherwise
//-----------------------------------------------------------------------------
bool preprundistFilter(Array& a_res,
      const Point& a_pt,
      const int a_m,
      const AngleRange& a_queryAR,
      const float a_mindist2q,
      const AngleRange& a_ar)


//-----------------------------------------------------------------------------
// incorporate PrePruning NS Result in a tier-wise fashion
//-----------------------------------------------------------------------------
int prepruningNSIncorp(Array& a_res,  // result set
        const Point& a_pt,           // query point
        const int a_m,               // number of result tiers
        const AngleRange& a_queryAR, // query angle range
        const int a_oid,             // object id
```

```
         const LineSeg& a_lineseg,    // line segment
         const AngleRange& a_ar)      // line segment angle range



void setMinViDistance(float vidist){
   minViDistance = vidist;
}

float getMinViDistance(){
   return minViDistance;
}

void setAnglRange(AngleRange angleRange){
   angRange = angleRange;
}

AngleRange getAnglRange(){
   return angRange;
}

//---------------------------------------------------------------------------
// This function calculates minimum visible distance of an edge and its
// corresponding angle range.
//---------------------------------------------------------------------------
void calcMinViDistance(Array& a_res,// result set
      const Point& a_pt,           // query point
      const int a_oid,             // object id
      const LineSeg& a_lineseg,    // line segment
      const AngleRange& a_ar)      // line segment angle range
{
   class carrier
   {
      public:
         const int m_oid;          // object id
         const LineSeg m_lineseg;  // line segment
         const AngleRange m_ar;    // angle range
         const float m_mindist;    // mindist to a query point
         const int m_level;        // target level
      public:
         carrier(const int a_oid, const LineSeg& a_lineseg,
            const AngleRange& a_ar, const float a_mindist,
            const int a_level):
            m_oid(a_oid), m_lineseg(a_lineseg),
            m_ar(a_ar), m_mindist(a_mindist), m_level(a_level) {};
            ~carrier() {};
         static int distcompare(const void* a0, const void* a1)
         {
            carrier* c0 = *(carrier**)a0;
```

```
            carrier* c1 = *(carrier**)a1;
            if (c0->m_mindist < c1->m_mindist) return -1;
            if (c0->m_mindist > c1->m_mindist) return +1;
            return 0;
         };
};

carrier* c = (carrier*)(new carrier(a_oid, a_lineseg, a_ar,
                                    a_lineseg.mindist(a_pt), 0));

//---------------------------------------------------------------------
// retrieve a target tier
//---------------------------------------------------------------------
RippleNSResult* rnsr = (RippleNSResult*)a_res.get(c->m_level);

//---------------------------------------------------------------------
// retrieve affected sections
//---------------------------------------------------------------------
Array affSection;
rnsr->getSection(c->m_ar, affSection);
float minvidist = c->m_mindist;
AngleRange carrRange = c->m_ar;      //carrier range

for (int k=0; k<affSection.size(); k++)
{
   Array& section = *(Array*)affSection.get(k);

   //------------------------------------------------------------------
   // locate affected portion of a result in a section
   //------------------------------------------------------------------
   int start, end;
   int cnt = preprunTierLookup(section, c->m_ar, start, end);

   for (int j=start; j<=end && cnt > 0; j++)
   {
      TierNSResult* tnsr = (TierNSResult*)section.get(j);
      if (tnsr->m_oid == c->m_oid && tnsr->m_lineseg == c->m_lineseg)
         continue;

      AngleRange common;
      if (AngleRange::common(tnsr->m_ar, carrRange, common) &&
                                              tnsr->m_oid != -1)
      {
         if(tnsr->m_ar.start()==tnsr->m_ar.end())
             continue;

         if (tnsr->m_ar.end() < carrRange.end() &&
               tnsr->m_ar.start() <= carrRange.start()){
            carrRange.set(tnsr->m_ar.end(), carrRange.end());
```

```
                minvidist = c->m_lineseg.mindist(a_pt, carrRange);
            }
            else if (tnsr->m_ar.end() >= carrRange.end() &&
                      tnsr->m_ar.start() > carrRange.start()){
                carrRange.set(carrRange.start(), tnsr->m_ar.start());
                minvidist = c->m_lineseg.mindist(a_pt, carrRange);
            }
            else if (tnsr->m_ar.end() >= carrRange.end() &&
                        tnsr->m_ar.start() <= carrRange.start()){
                //invisible
                carrRange.set(0, 0);
                minvidist = INFTY;
                break;
            }
            else if (tnsr->m_ar.end() < carrRange.end() &&
                         tnsr->m_ar.start() > carrRange.start()){
               AngleRange arange, brange;
               arange.set(carrRange.start(), tnsr->m_ar.start());
               brange.set(tnsr->m_ar.end(), carrRange.end());
               float mindist1 = c->m_lineseg.mindist(a_pt, arange);
               float mindist2 = c->m_lineseg.mindist(a_pt, brange);

               if (mindist1 < mindist2){
                  minvidist = mindist1;
                  carrRange = arange;
               }
               else{
                  minvidist = mindist2;
                  carrRange = brange;
               }
            }
         }
      }
   }
   setMinViDistance(minvidist);
   setAnglRange(carrRange);
}

//-----------------------------------------------------------------------
// Calculates the minimum distance of a hypercube according to a specific
// angle range.
//-----------------------------------------------------------------------
float mindistHC(const Hypercube& a_hc, const Point& a_pt,
                                    const AngleRange& a_ar){
   const Point& lower = a_hc.getLower();
   const Point& upper = a_hc.getUpper();
   float mindist = INFTY;
   float c[2];
   c[0] = upper[0];
```

```
   c[1] = lower[1];
   Point p(2, c);
   LineSeg l(lower, p);
   AngleRange ar = l.angleRange(a_pt);
   AngleRange common;
   if (AngleRange::common(a_ar, ar, common))
   {
      mindist = mindist < l.mindist(a_pt, a_ar) ? mindist : l.mindist(a_pt, a_ar);
   }

   LineSeg l1(p, upper);
   ar = l1.angleRange(a_pt);
   if (AngleRange::common(a_ar, ar, common))
   {
      mindist = mindist < l1.mindist(a_pt, a_ar) ? mindist : l1.mindist(a_pt, a_ar);
   }

   c[0] = lower[0];
   c[1] = upper[1];
   Point p1(2, c);
   LineSeg l2(upper, p1);
   ar = l2.angleRange(a_pt);
   if (AngleRange::common(a_ar, ar, common))
   {
      mindist = mindist < l2.mindist(a_pt, a_ar) ? mindist : l2.mindist(a_pt, a_ar);
   }

   LineSeg l3(p1, lower);
   ar = l3.angleRange(a_pt);
   if (AngleRange::common(a_ar, ar, common))
   {
      mindist = mindist < l3.mindist(a_pt, a_ar) ? mindist : l3.mindist(a_pt, a_ar);
   }

   return mindist;
}

//-----------------------------------------------------------------------------
// Calculates the minimum visible distance of a node.
//-----------------------------------------------------------------------------
void calcMinViDistNode(Array& a_res, // result set
      const Point& a_pt,             // query point
      const int a_oid,               // object id
      const Hypercube& a_hc,
      const AngleRange& a_ar)        // line segment angle range
{
   class carrier
   {
      public:
```

```
      const int m_oid;       // object id
      const Hypercube m_hc;
      const AngleRange m_ar; // angle range
      const float m_mindist; // mindist to a query point
      const int m_level;     // target level
   public:
      carrier(const int a_oid,
         const Hypercube& a_hc,
         const AngleRange& a_ar,
         const float a_mindist,
         const int a_level):
         m_oid(a_oid),
         m_hc(a_hc),
         m_ar(a_ar), m_mindist(a_mindist), m_level(a_level) {};
      ~carrier() {};
      static int distcompare(const void* a0, const void* a1)
      {
         carrier* c0 = *(carrier**)a0;
         carrier* c1 = *(carrier**)a1;
         if (c0->m_mindist < c1->m_mindist) return -1;
         if (c0->m_mindist > c1->m_mindist) return +1;
         return 0;
      };
};

carrier* c = (carrier*)(new carrier(a_oid, a_hc, a_ar, a_hc.mindist(a_pt), 0));

//-------------------------------------------------------------------
// retrieve a target tier
//-------------------------------------------------------------------
RippleNSResult* rnsr = (RippleNSResult*)a_res.get(c->m_level);

//-------------------------------------------------------------------
// retrieve affected sections
//-------------------------------------------------------------------
Array affSection;
rnsr->getSection(c->m_ar, affSection);

float minvidist = c->m_mindist;
AngleRange carrRange = c->m_ar;      //carrier range

for (int k=0; k<affSection.size(); k++)
{
   Array& section = *(Array*)affSection.get(k);

   //----------------------------------------------------------------
   // locate affected portion of a result in a section
   //----------------------------------------------------------------
   int start, end;
```

```
int cnt = preprunTierLookup(section, c->m_ar, start, end);

for (int j=start; j<=end && cnt > 0; j++)
{
   TierNSResult* tnsr = (TierNSResult*)section.get(j);
   if (tnsr->m_oid == c->m_oid){
      continue;
   }
   AngleRange common;

   if (AngleRange::common(tnsr->m_ar, carrRange, common) &&
                                          tnsr->m_oid != -1)
   {
      if(tnsr->m_ar.start()==tnsr->m_ar.end())
         continue;

      if (tnsr->m_ar.end() < carrRange.end() &&
            tnsr->m_ar.start() <= carrRange.start()){
         carrRange.set(tnsr->m_ar.end(), carrRange.end());
         minvidist = mindistHC(c->m_hc, a_pt, carrRange);
      }
      else if (tnsr->m_ar.end() >= carrRange.end() &&
                  tnsr->m_ar.start() > carrRange.start()){
         carrRange.set(carrRange.start(), tnsr->m_ar.start());
         minvidist = mindistHC(c->m_hc, a_pt, carrRange);
      }
      else if (tnsr->m_ar.end() >= carrRange.end() &&
               tnsr->m_ar.start() <= carrRange.start()){
         //invisible
         carrRange.set(0, 0);
         minvidist = INFTY;
         break;
      }
      else if (tnsr->m_ar.end() < carrRange.end() &&
                  tnsr->m_ar.start() > carrRange.start()){
         AngleRange arange, brange;
         arange.set(carrRange.start(), tnsr->m_ar.start());
         brange.set(tnsr->m_ar.end(), carrRange.end());
         float mindist1 = mindistHC(c->m_hc, a_pt, arange);
         float mindist2 = mindistHC(c->m_hc, a_pt, brange);

         if (mindist1 < mindist2){
            minvidist = mindist1;
            carrRange = arange;
         }
         else{
            minvidist = mindist2;
            carrRange = brange;
         }
```

```
            }
         }
      }
   }
   setMinViDistance(minvidist);
   setAnglRange(carrRange);
}

//-----------------------------------------------------------------------
// This function returns the number of objects in the current result.
//-----------------------------------------------------------------------
int objsNoInResult(Array& res, int m){

   Set allobjid;
   for (int i=0; i<m; i++)
   {
      int prevoid= -100;
      RippleNSResult* rnsr = (RippleNSResult*)res.get(i);
      for (int j=0; j<rnsr->m_numsect; j++)
      {
         for (int k=0; k<rnsr->m_section[j]->size(); k++)
         {
            TierNSResult* r = (TierNSResult*)rnsr->m_section[j]->get(k);
            if (r->m_ar.width() == 0) continue;
            if (r->m_oid != prevoid)
            {
               if (prevoid >= 0)
               {
                  allobjid.insert((void*)prevoid);
               }
               prevoid = r->m_oid;
            }
         }
      }
      if (prevoid >= 0)
         allobjid.insert((void*)prevoid);
   }
   return allobjid.size();
}

//-------------------------------------------------------------
// This function checks if an object exists in the current result
//-------------------------------------------------------------
bool existInResult(Array& res, int m, int id){

   for (int i=0; i<m; i++)
   {
      RippleNSResult* rnsr = (RippleNSResult*)res.get(i);
      for (int j=0; j<rnsr->m_numsect; j++)
```

```
      {
         for (int k=0; k<rnsr->m_section[j]->size(); k++)
         {
            TierNSResult* r = (TierNSResult*)rnsr->m_section[j]->get(k);
            if (r->m_ar.width() == 0) continue;
            if (r->m_oid == id) return true;
         }
       }
    }
  return false;
}

//------------------------------------------------------------------------------
// PrePruning NSQ algorithm
//------------------------------------------------------------------------------
int NSSearch::prepruning(Rtree& a_rtree, Hash& a_hPolygon,
                const Point& a_pt,
                const AngleRange& a_queryAR, const int a_m,
                Array& a_res, int& a_exam, int& a_max,
                Array& a_pageaccessed, const int k)
{
   //---------------------------------------------------------------------------
   // carrier (base)
   //---------------------------------------------------------------------------
   class carrier
   {
   public:
      const AngleRange m_ar;
      const float m_mindist2q;
      const int m_type;
   public:
      carrier(const AngleRange& a_ar,
         const float a_mindist2q, const int a_type):
         m_ar(a_ar), m_mindist2q(a_mindist2q), m_type(a_type) {};
      virtual ~carrier() {};

      static int distcompare(const void* a0, const void* a1)
      {
         carrier* p0 = *(carrier**)a0;
         carrier* p1 = *(carrier**)a1;
         if (p0->m_mindist2q  < p1->m_mindist2q)  return -1;
         if (p0->m_mindist2q  > p1->m_mindist2q)  return +1;
         return 0;
      };
   };

   //---------------------------------------------------------------------------
   // carrier of a node entry
   //---------------------------------------------------------------------------
```

```
class nodecarrier: public carrier
{
   public:
      RtreeNodeEntry* m_entry;
      const int m_level;   // level in a Rtree
   public:
   nodecarrier(const RtreeNodeEntry& a_entry, const int a_level,
      const AngleRange& a_ar, const float a_mindist2q):
      m_entry(a_entry.clone()), m_level(a_level),
      carrier(a_ar, a_mindist2q, NODE) {};
   virtual ~nodecarrier() { delete m_entry; };
};

//---------------------------------------------------------------------------
// carrier of a edge entry
//---------------------------------------------------------------------------
class edgecarrier: public carrier
{
  public:
     LineSeg      m_lineseg;
     const int   m_oid;
  public:
     edgecarrier(const LineSeg& a_lineseg, const int a_oid,
        const AngleRange& a_ar, const float a_mindist2q):
        m_lineseg(a_lineseg), m_oid(a_oid),
        carrier(a_ar, a_mindist2q, EDGE) {};
   virtual ~edgecarrier() {};
};

//---------------------------------------------------------------------------
// initialize a m-tier NS result
//---------------------------------------------------------------------------
for (int i=0; i<a_m; i++)
{
   LineSeg l(a_pt,a_pt);         // dummy line segment
   RippleNSResult* rns = new RippleNSResult(DUMMY, l, a_queryAR, 60);
   a_res.append(rns);
}

//---------------------------------------------------------------------------
// create a heap that orders entries based on starting angle
//---------------------------------------------------------------------------
BinHeap h(carrier::distcompare);

//---------------------------------------------------------------------------
// initialize the first heap entry
//---------------------------------------------------------------------------
float cl[DIMEN], cu[DIMEN];
cl[0] = cl[1] = -INFTY;
```

```
   cu[0] = cu[1] = INFTY;
   Hypercube hc(DIMEN, cl, cu);
   RtreeNodeEntry dummyroot(a_rtree.m_memory.m_rootPageID, hc);
   AngleRange ar(a_queryAR);
   h.insert(new nodecarrier(dummyroot, 100, ar, 0));

   while (!h.isEmpty())
   {
      int heapsize = h.size();
      a_max = a_max > heapsize ? a_max : heapsize;
      carrier* c = (carrier*)h.removeTop();

      //-------------------------------------------------------------------
      // filter out carrier that are already hidden by existing edges
      //-------------------------------------------------------------------
      if (preprundistFilter(a_res, a_pt, a_m,a_queryAR,c->m_mindist2q,c->m_ar))
      {
         delete c;
         continue;
      }

      //-------------------------------------------------------------------
      //prepruning check for minimum visible distance
      //-------------------------------------------------------------------
      bool objInserted = false;
      AngleRange commonrange;
      float minimumdist = -1;
      if(c->m_type == NODE)
      {
         nodecarrier* nc = (nodecarrier*)c;
         if (AngleRange::common(a_queryAR, nc->m_ar, commonrange))
         {
            if (!h.isEmpty())
            {
               //calculate the MinViDist of a Node
               calcMinViDistNode(a_res, a_pt, nc->m_entry->m_id,
                                 nc->m_entry->m_hc, commonrange);
               minimumdist = getMinViDistance();
               AngleRange corrRange = getAnglRange();
               carrier* topc = (carrier*) h.top();
               if (minimumdist > topc->m_mindist2q)
               {
                  objInserted = true;
                  if(minimumdist != INFTY   && minimumdist != -1)
                  {
//cout<<"Node has minimum visible distance bigger than the top of the Queue."<<endl;
                     h.insert(new nodecarrier(*nc->m_entry, nc->m_level,
                                              corrRange, minimumdist));
                  }
```

```
         }
      }
   }
}
else // if it is an edge-object
{
   edgecarrier* ec = (edgecarrier*)c;
   if (AngleRange::common(a_queryAR, ec->m_ar, commonrange))
   {
      if (!h.isEmpty())
      {
         //calculate the MinViDist of an Edge
         calcMinViDistance(a_res, a_pt, ec->m_oid,
                           ec->m_lineseg, commonrange);
         AngleRange corrRange = getAnglRange();
         minimumdist = getMinViDistance();
         carrier* topc = (carrier*) h.top();
         if (minimumdist > topc->m_mindist2q)
         {
            objInserted = true;
            if(minimumdist != INFTY   && minimumdist != -1)
            {
            //cout<<"Object id:"<<ec->m_oid<<" has minimum visible
            //distance bigger than the top of the Queue."<<endl;
               h.insert(new edgecarrier(ec -> m_lineseg, ec -> m_oid, corrRange,
                        minimumdist));
            }
         }
       }
    }
}

//---------------------------------------------------------------------
// explore R-tree
//---------------------------------------------------------------------
if (c->m_type == NODE && !objInserted)
{
   nodecarrier* nc = (nodecarrier*)c;

   if (nc->m_level >= 0)
   {
      //-------------------------------------------------------------
      // explore an Rtree node and place its node entries into the
      // heap for later examination
      //-------------------------------------------------------------
      a_pageaccessed.append((void*)nc->m_entry->m_id);
      RtreeNode* node = a_rtree.m_memory.loadPage(nc->m_entry->m_id);

      for (int i=0; i<node->m_usedSpace; i++)
```

```
{
//-------------------------------------------------------
// if a MBB covers a query point, its ar = (0,360)
//-------------------------------------------------------
    if (node->m_entry[i]->m_hc.enclose(a_pt))
    {
       AngleRange ar(0,360);
       h.insert(new nodecarrier(*node->m_entry[i],
                           node->m_level-1, ar, 0));
       continue;
    }

    Polygon plg = Polygon::convert(node->m_entry[i]->m_hc);
    if (plg.acrossX(a_pt))
    {
      //-----------------------------------------------------
      // partition a node if it cuts across +ve axis
      //-----------------------------------------------------
      Polygon above, below;
      plg.partition(a_pt, above, below);
      AngleRange abovear = above.angleRange(a_pt);
      AngleRange belowar = below.angleRange(a_pt);

      //-----------------------------------------------------
      // pending the entry for later examination if it is
      // within a query angle range
      //-----------------------------------------------------
      AngleRange aCommon, bCommon;
      if (AngleRange::common(a_queryAR, abovear, aCommon) ||
         AngleRange::common(a_queryAR, belowar, bCommon))
      {
         h.insert(new nodecarrier(
            *node->m_entry[i], node->m_level-1,
            ar, node->m_entry[i]->m_hc.mindist(a_pt)));
      }
    }
    else
    {
      //-----------------------------------------------------
      // pending the entry for later examination if it is
      // within a query angle range
      //-----------------------------------------------------
      AngleRange ar = plg.angleRange(a_pt);
      AngleRange common;
      if (AngleRange::common(a_queryAR, ar, common))
      {
         h.insert(new nodecarrier(
               *node->m_entry[i], node->m_level-1,
               ar, node->m_entry[i]->m_hc.mindist(a_pt)));
```

```
         }
        }
     }
     delete node;
}
else
{
   a_exam++;   // count the no. of examined objects
   //-------------------------------------------------------------
   // explore the polygon and place its edge for later examination
   //-------------------------------------------------------------
   int oid = nc->m_entry->m_id;

   Polygon* plg = (Polygon*)a_hPolygon.get(oid);
   if (plg->acrossX(a_pt))
   {
      Polygon above, below;
      plg->partition(a_pt, above, below);
      for (int i=0; i<above.numLineSeg(); i++)
      {
         const LineSeg& l = above[i];
         AngleRange ar = l.angleRange(a_pt);
         AngleRange common;
         if (AngleRange::common(a_queryAR, ar, common))
         {   // place edge if it is covered by a query angle range
            float mindist = l.mindist(a_pt, common);
            h.insert(new edgecarrier(l, oid, ar, mindist));
         }
      }
      for (int i=0; i<below.numLineSeg(); i++)
      {
         const LineSeg& l = below[i];
         AngleRange ar = l.angleRange(a_pt);
         AngleRange common;
         if (AngleRange::common(a_queryAR, ar, common))
         {   // place edge if it is covered by a query angle range
            float mindist = l.mindist(a_pt, common);
            h.insert(new edgecarrier(l, oid, ar, mindist));
         }
      }
   }
   else
   {
      for (int i=0; i<plg->numLineSeg(); i++)
      {
         const LineSeg& l = (*plg)[i];
         AngleRange ar = l.angleRange(a_pt);
         AngleRange common;
         if (AngleRange::common(a_queryAR, ar, common))
```

```
                {
                   float mindist = l.mindist(a_pt, common);
                   h.insert(new edgecarrier(l, oid, ar, mindist));
                }
             }
          }
       }
    }
    //-------------------------------------------------------------------
    // investigate edges
    //-------------------------------------------------------------------
    else if(!objInserted) // if c->m_type is EDGE
    {
       edgecarrier* ec = (edgecarrier*)c;
       AngleRange common;
       //-------------------------------------------------------------
       //because the check is being done according to edges not objects.
       //So in the last step we may check another edge of the same kth
       //object-edge examined so far but this isn't k+1 its just kth.
       //-------------------------------------------------------------
       if (AngleRange::common(a_queryAR, ec->m_ar, common))
       {
          if((existInResult(a_res, a_m, ec->m_oid) ||
                   (!existInResult(a_res, a_m, ec->m_oid) &&
                   objsNoInResult(a_res, a_m)<k)))
          {
              prepruningNSIncorp(a_res, a_pt, a_m, a_queryAR,
                       ec->m_oid, ec->m_lineseg, common);
          }
          else return 0;
        }
    }
    delete c;
  }
  return 0;
}
```

A.3 Κλάση nspostpruning.cc

Η κλάση nspostpruning υλοποιεί τον καθαυτό αλγόριθμο PostPruning(k)NS. Ομοίως από την κλάση αυτή παραλείφθηκε το περιεχόμενο των συναρτήσεων που προϋπήρχαν και παρουσιάζονται μόνο οι συναρτήσεις που υλοποιήθηκαν.

```
/* ---------------------------------------------------------------------------
    Author: Anna Giannoutsou
    Email:  anna.giannoutsou@gmail.com
```

```
    Date:   July, 2011
------------------------------------------------------------------------------ */
//#include space...

using namespace std;
#define NODE   0
#define EDGE   1
#define DIMEN  2
#define DUMMY  -1

float minViDist;
AngleRange anglRange;

//----------------------------------------------------------------------------
// edge compare result
//----------------------------------------------------------------------------
class EdgeCompResult

int postprunEdgeCompare(const int a_oid0,
         const LineSeg& a_lineseg0,
         const AngleRange& a_ar0,   // lineseg0
         const int a_oid1,
         const LineSeg& a_lineseg1,
         const AngleRange& a_ar1,   // lineseg1
         const Point& a_pt,         // ref. point
         Array& a_cmpres,

//----------------------------------------------------------------------------
// determine a range of existng NS results whose angle ranges are covered by
// a provided angle range (angle range search)
//----------------------------------------------------------------------------
int postprunTierLookup(Array& a_tier,   // a tier of NS result
               const AngleRange& a_ar, // angle range of a candidate
               int& a_start,           // starting position
               int& a_end)             // ending position

//----------------------------------------------------------------------------
// preliminary determine if an entry is hidden by existing edges
// true: hidden, and the entry can be discarded
// false: otherwise
//----------------------------------------------------------------------------
bool postprundistFilter(Array& a_res,
      const Point& a_pt,
      const int a_m,
      const AngleRange& a_queryAR,
      const float a_mindist2q,
      const AngleRange& a_ar)

//----------------------------------------------------------------------------
```

```
// incorporate PostPruning NSResult in a tier-wise fashion
//---------------------------------------------------------------------------
int postpruningNSIncorp(Array& a_res, // result set
         const Point& a_pt,           // query point
         const int a_m,               // number of result tiers
         const AngleRange& a_queryAR, // query angle range
         const int a_oid,             // object id
         const LineSeg& a_lineseg,    // line segment
         const AngleRange& a_ar)      // line segment angle range


void setMinViDist(float vidist){
   minViDist = vidist;
}

float getMinViDist(){
   return minViDist;
}

void setAngleRange(AngleRange angleRange){
   anglRange = angleRange;
}

AngleRange getAngleRange(){
   return anglRange;
}

//---------------------------------------------------------------------------
// This calculates minimum visible distance of an edge-object and its
// corresponding angle range.
//---------------------------------------------------------------------------
void calcMinViDist(Array& a_res,   // result set
      const Point& a_pt,           // query point
      const int a_oid,             // object id
      const LineSeg& a_lineseg,    // line segment
      const AngleRange& a_ar)      // line segment angle range
{
   class carrier
   {
      public:
         const int m_oid;          // object id
         const LineSeg m_lineseg;  // line segment
         const AngleRange m_ar;    // angle range
         const float m_mindist;    // mindist to a query point
         const int m_level;        // target level
      public:
         carrier(const int a_oid, const LineSeg& a_lineseg,
            const AngleRange& a_ar, const float a_mindist,
            const int a_level):
```

```
         m_oid(a_oid), m_lineseg(a_lineseg),
         m_ar(a_ar), m_mindist(a_mindist), m_level(a_level) {};
      ~carrier() {};
      static int distcompare(const void* a0, const void* a1)
      {
         carrier* c0 = *(carrier**)a0;
         carrier* c1 = *(carrier**)a1;
         if (c0->m_mindist < c1->m_mindist) return -1;
         if (c0->m_mindist > c1->m_mindist) return +1;
         return 0;
      };
};
carrier* c = (carrier*)(new carrier(a_oid, a_lineseg,
                           a_ar, a_lineseg.mindist(a_pt), 0));

//---------------------------------------------------------------------
// retrieve a target tier
//---------------------------------------------------------------------
RippleNSResult* rnsr = (RippleNSResult*)a_res.get(c->m_level);

//---------------------------------------------------------------------
// retrieve affected sections
//---------------------------------------------------------------------
Array affSection;
rnsr->getSection(c->m_ar, affSection);
float minvidist = c->m_mindist;
AngleRange carrRange = c->m_ar;      //carrier range

for (int k=0; k<affSection.size(); k++)
{
   Array& section = *(Array*)affSection.get(k);

   //------------------------------------------------------------------
   // locate affected portion of a result in a section
   //------------------------------------------------------------------
   int start, end;
   int cnt = postprunTierLookup(section, c->m_ar, start, end);


   for (int j=start; j<=end && cnt > 0; j++)
   {
      TierNSResult* tnsr = (TierNSResult*)section.get(j);
      if (tnsr->m_oid == c->m_oid && tnsr->m_lineseg == c->m_lineseg)
                           continue;

      AngleRange common;

      if (AngleRange::common(tnsr->m_ar, carrRange, common)
                           && tnsr->m_oid != -1)
```

```
        {
           if(tnsr->m_ar.start()==tnsr->m_ar.end())
                 continue;

           if (tnsr->m_ar.end() < carrRange.end() &&
                       tnsr->m_ar.start() <= carrRange.start()){
              carrRange.set(tnsr->m_ar.end(), carrRange.end());
              minvidist = c->m_lineseg.mindist(a_pt, carrRange);
           }
           else if (tnsr->m_ar.end() >= carrRange.end() &&
                       tnsr->m_ar.start() > carrRange.start()){
              carrRange.set(carrRange.start(), tnsr->m_ar.start());
              minvidist = c->m_lineseg.mindist(a_pt, carrRange);
           }
           else if (tnsr->m_ar.end() >= carrRange.end() &&
                       tnsr->m_ar.start() <= carrRange.start()){
              //invisible
              carrRange.set(0, 0);
              minvidist = INFTY;
              break;
           }
           else if (tnsr->m_ar.end() < carrRange.end() &&
                       tnsr->m_ar.start() > carrRange.start()){
              AngleRange arange, brange;
              arange.set(carrRange.start(), tnsr->m_ar.start());
              brange.set(tnsr->m_ar.end(), carrRange.end());
              float mindist1 = c->m_lineseg.mindist(a_pt, arange);
              float mindist2 = c->m_lineseg.mindist(a_pt, brange);

              if (mindist1 < mindist2){
                 minvidist = mindist1;
                 carrRange = arange;
              }
              else{
                 minvidist = mindist2;
                 carrRange = brange;
              }
           }
        }
     }
  }
  setMinViDist(minvidist);
  setAngleRange(carrRange);
}

//-------------------------------------------------
// Returns the no of objects founded so far
//-------------------------------------------------
int noOfObjsInResult(Array& res, int m){
```

```
   Set allobjid;
   for (int i=0; i<m; i++)
   {
      int prevoid= -100;
      RippleNSResult* rnsr = (RippleNSResult*)res.get(i);
      for (int j=0; j<rnsr->m_numsect; j++)
      {
         for (int k=0; k<rnsr->m_section[j]->size(); k++)
         {
            TierNSResult* r = (TierNSResult*)rnsr->m_section[j]->get(k);
            if (r->m_ar.width() == 0) continue;
            if (r->m_oid != prevoid)
            {
               if (prevoid >= 0)
               {
                   allobjid.insert((void*)prevoid);
               }
               prevoid = r->m_oid;
            }
         }
      }

     if (prevoid >= 0)
             allobjid.insert((void*)prevoid);
   }
   return allobjid.size();
}

//-----------------------------------------------------------------
// Checks if an object exists in the result that founded so far
//-----------------------------------------------------------------
bool existsInResult(Array& res, int m, int id){

   for (int i=0; i<m; i++)
   {
      RippleNSResult* rnsr = (RippleNSResult*)res.get(i);
      for (int j=0; j<rnsr->m_numsect; j++)
      {
         for (int k=0; k<rnsr->m_section[j]->size(); k++)
         {
            TierNSResult* r = (TierNSResult*)rnsr->m_section[j]->get(k);
            if (r->m_ar.width() == 0) continue;
            if (r->m_oid == id) return true;
         }
      }
   }
   return false;
}
```

```
//---------------------------------------------------------------------------
// PostPruning NSQ algorithm
//---------------------------------------------------------------------------
int NSSearch::postpruning(Rtree& a_rtree, Hash& a_hPolygon,
                const Point& a_pt,
                const AngleRange& a_queryAR, const int a_m,
                Array& a_res, int& a_exam, int& a_max,
                Array& a_pageaccessed, const int k)
{
   //---------------------------------------------------------------------------
   // carrier (base)
   //---------------------------------------------------------------------------
   class carrier
   {
     public:
        const AngleRange m_ar;
        const float m_mindist2q;
        const int m_type;
     public:
        carrier(const AngleRange& a_ar,
           const float a_mindist2q, const int a_type):
           m_ar(a_ar), m_mindist2q(a_mindist2q), m_type(a_type) {};
        virtual ~carrier() {};

       static int distcompare(const void* a0, const void* a1)
       {
         carrier* p0 = *(carrier**)a0;
         carrier* p1 = *(carrier**)a1;
         if (p0->m_mindist2q  < p1->m_mindist2q)  return -1;
         if (p0->m_mindist2q  > p1->m_mindist2q)  return +1;
         return 0;
       };
    };

   //---------------------------------------------------------------------------
   // carrier of a node entry
   //---------------------------------------------------------------------------
   class nodecarrier: public carrier
   {
     public:
       RtreeNodeEntry* m_entry;
       const int m_level;   // level in a Rtree
     public:
       nodecarrier(const RtreeNodeEntry& a_entry, const int a_level,
          const AngleRange& a_ar, const float a_mindist2q):
          m_entry(a_entry.clone()), m_level(a_level),
          carrier(a_ar, a_mindist2q, NODE) {};
```

```
    virtual ~nodecarrier() { delete m_entry; };
 };

//---------------------------------------------------------------------------
// carrier of a edge entry
//---------------------------------------------------------------------------
class edgecarrier: public carrier
{
  public:
     LineSeg m_lineseg;
     const int m_oid;
 public:
    edgecarrier(const LineSeg& a_lineseg, const int a_oid,
       const AngleRange& a_ar, const float a_mindist2q):
       m_lineseg(a_lineseg), m_oid(a_oid),
       carrier(a_ar, a_mindist2q, EDGE) {};
    virtual ~edgecarrier() {};
};

//---------------------------------------------------------------------------
// initialize a m-tier NS result
//---------------------------------------------------------------------------
for (int i=0; i<a_m; i++)
{
   LineSeg l(a_pt,a_pt);                    // dummy line segment
   RippleNSResult* rns = new RippleNSResult(DUMMY, l, a_queryAR, 60);
   a_res.append(rns);
}

//---------------------------------------------------------------------------
// create a heap that orders entries based on starting angle
//---------------------------------------------------------------------------
BinHeap h(carrier::distcompare);

//---------------------------------------------------------------------------
// initialize the first heap entry
//---------------------------------------------------------------------------
float cl[DIMEN], cu[DIMEN];
cl[0] = cl[1] = -INFTY;
cu[0] = cu[1] = INFTY;
Hypercube hc(DIMEN, cl, cu);
RtreeNodeEntry dummyroot(a_rtree.m_memory.m_rootPageID, hc);
AngleRange ar(a_queryAR);
h.insert(new nodecarrier(dummyroot, 100, ar, 0));

while (!h.isEmpty())
{
   int heapsize = h.size();
   a_max = a_max > heapsize ? a_max : heapsize;
```

```
carrier* c = (carrier*)h.removeTop();

//---------------------------------------------------------------------
// filter out carrier that are already hidden by existing edges
//---------------------------------------------------------------------
if (postprundistFilter(a_res, a_pt, a_m, a_queryAR, c->m_mindist2q, c->m_ar))
{
   delete c;
   continue;
}

//---------------------------------------------------------------------
// explore R-tree
//---------------------------------------------------------------------
if (c->m_type == NODE)
{
   nodecarrier* nc = (nodecarrier*)c;
   if (nc->m_level >= 0)
   {
      //-------------------------------------------------------------
      // explore an Rtree node and place its node entries into the
      // heap for later examination
      //-------------------------------------------------------------
      a_pageaccessed.append((void*)nc->m_entry->m_id);
      RtreeNode* node = a_rtree.m_memory.loadPage(nc->m_entry->m_id);

      for (int i=0; i<node->m_usedSpace; i++)
      {
        //-----------------------------------------------------
        // if a MBB covers a query point, its ar = (0,360)
        //-----------------------------------------------------
        if (node->m_entry[i]->m_hc.enclose(a_pt))
        {
            AngleRange ar(0,360);
            h.insert(new nodecarrier(*node->m_entry[i], node->m_level-1, ar, 0));
            continue;
        }

        Polygon plg = Polygon::convert(node->m_entry[i]->m_hc);
        if (plg.acrossX(a_pt))
        {
            //-------------------------------------------------
            // partition a node if it cuts across +ve axis
            //-------------------------------------------------
            Polygon above, below;
            plg.partition(a_pt, above, below);
            AngleRange abovear = above.angleRange(a_pt);
            AngleRange belowar = below.angleRange(a_pt);
```

```
            //-----------------------------------------------------
            // pending the entry for later examination if it is
            // within a query angle range
            //-----------------------------------------------------
            AngleRange aCommon, bCommon;
            if (AngleRange::common(a_queryAR, abovear, aCommon) ||
               AngleRange::common(a_queryAR, belowar, bCommon))
            {
               h.insert(new nodecarrier(
                  *node->m_entry[i], node->m_level-1,
                  ar, node->m_entry[i]->m_hc.mindist(a_pt)));
            }
         }
         else
         {
            //-----------------------------------------------------
            // pending the entry for later examination if it is
            // within a query angle range
            //-----------------------------------------------------
            AngleRange ar = plg.angleRange(a_pt);
            AngleRange common;

            if (AngleRange::common(a_queryAR, ar, common))
            {
               h.insert(new nodecarrier(
                  *node->m_entry[i], node->m_level-1,
                  ar, node->m_entry[i]->m_hc.mindist(a_pt)));
            }
         }
      }
      delete node;
   }
   else
   {
      a_exam++;   // count the no. of examined objects

      //-------------------------------------------------------------
      // explore the polygon and place its edge for later examination
      //-------------------------------------------------------------
      int oid = nc->m_entry->m_id;

      Polygon* plg = (Polygon*)a_hPolygon.get(oid);
      if (plg->acrossX(a_pt))
      {
        Polygon above, below;
        plg->partition(a_pt, above, below);
        for (int i=0; i<above.numLineSeg(); i++)
        {
           const LineSeg& l = above[i];
```

```
            AngleRange ar = l.angleRange(a_pt);
            AngleRange common;
            if (AngleRange::common(a_queryAR, ar, common))
            {   // place edge if it is covered by a query angle range
               float mindist = l.mindist(a_pt, common);
               h.insert(new edgecarrier(l, oid, ar, mindist));
            }
         }
         for (int i=0; i<below.numLineSeg(); i++)
         {
            const LineSeg& l = below[i];
            AngleRange ar = l.angleRange(a_pt);
            AngleRange common;
            if (AngleRange::common(a_queryAR, ar, common))
            {   // place edge if it is covered by a query angle range
               float mindist = l.mindist(a_pt, common);
               h.insert(new edgecarrier(l, oid, ar, mindist));
            }
         }
      }
      else
      {
         for (int i=0; i<plg->numLineSeg(); i++)
         {
            const LineSeg& l = (*plg)[i];
            AngleRange ar = l.angleRange(a_pt);
            AngleRange common;
            if (AngleRange::common(a_queryAR, ar, common))
            {
               float mindist = l.mindist(a_pt, common);
               h.insert(new edgecarrier(l, oid, ar, mindist));
            }
         }
      }
   }
}
//---------------------------------------------------------------------
// investigate edges
//---------------------------------------------------------------------
else // if (c->m_type == EDGE)
{
   edgecarrier* ec = (edgecarrier*)c;
   AngleRange common;
   float minimumdist = -1;

   //because the check is being done according to edges not objects.
   //So in the last step we may check
   //another edge of the same kth object-edge examined so
   //far but this isn't k+1 its just kth.
```

```
            if (AngleRange::common(a_queryAR, ec->m_ar, common))
            {
              //-------------------------------------------------------
              // post-pruning check for minimum visible distance
              //-------------------------------------------------------
              if((existsInResult(a_res, a_m, ec->m_oid) ||
                  (!existsInResult(a_res, a_m, ec->m_oid) &&
                     noOfObjsInResult(a_res, a_m)<k)))   //kCheck
              {
                  if (!h.isEmpty())
                  {
                     calcMinViDist(a_res, a_pt, ec->m_oid, ec->m_lineseg, common);

                     AngleRange corrRange = getAngleRange();
                     minimumdist = getMinViDist();
                     carrier* topc = (carrier*) h.top();
                     float headmindist = topc->m_mindist2q;

                     if (minimumdist > topc->m_mindist2q && minimumdist != INFTY
                                                 && minimumdist != -1)
                     {
          //cout<<"Object id:"<<ec->m_oid<<" has minimum visible distance
          //bigger than the top of the Queue."<<endl;
                        h.insert(new edgecarrier(ec -> m_lineseg, ec -> m_oid,
                                    corrRange, minimumdist));
                     }
                     else{ // renew the result
                        postpruningNSIncorp(a_res, a_pt, a_m, a_queryAR,
                              ec->m_oid, ec->m_lineseg, common);
                     }
                   }
                   else{
                      postpruningNSIncorp(a_res, a_pt, a_m, a_queryAR,
                              ec->m_oid, ec->m_lineseg, common);
                   }
                 }
                 else{
                    return 0;
                 }
               }
             }
             delete c;
          }
          return 0;
        }
```

Α.4 Κλάση sweepkns.cc

Η κλάση sweepkns που παρουσιάζεται στη συνέχεια περιέχει τη main συνάρτηση του αλγορίθμου SweepkNS. Μέσα σε αυτή την κλάση έχει υλοποιηθεί η διαδικασία που επιστρέφει τους k κοντινότερους γείτονες από το σύνολο των αποτελεσμάτων του SweepkNS. Ο ίδιος ο αλγόριθμος Sweep δεν αλλάχτηκε.

```
/* ---------------------------------------------------------------------------
    Author: Anna Giannoutsou
    Email:  anna.giannoutsou@gmail.com
    Date:   July, 2011

    This program:
   1. load an Rtree indexing polygons
   2. load a collection of polygons
   3. performs nearest surrounder query based on
      SWEEP k Nearest Surrounders algorithm
   4. output the result

    Suggested arguments:
    > (prog name) -p 4096 -f raw_data.txt -i index_file.idx
      -q queryfile.txt
      -a0 startangle -a1 endangle
      -m #tiers -t #sectors
      -v true -k #neighbors

    explanations:
   -p: pagesize, typically, 4096
   -f: a tab delimited file, format:
        id #vertice x0 y0 x1 y1 ... xn yn
   -i: index file
   -q: query point file
   -a0: start angle (default: 0)
   -a1: end angle (default: 360)
   -m: #tiers (default: 1)
   -t: #sectors (default: 1)
   -v: verbose mode on
   -k: #neighbors (default: 1)
--------------------------------------------------------------------------- */
//#include space...

using namespace std;
#define MAXEDGE   1000
#define DIMEN   2
#define EDGE   1

//This function draws the result (objects and query point) at an .eps file,
//Draws the VNNs red and the rest of the objects black. The query point is
//represented by a blue dot.
```

```
void drawResult(const Rtree& a_rtree,
                  const int a_bottomlevel, const int a_toplevel,
                  const char* epsname, vector<int> objid, float c[DIMEN])
{
    RtreeNode* root = a_rtree.m_memory.loadPage(a_rtree.m_memory.m_rootPageID);
    RtreeNodeEntry* entry = root->genNodeEntry();
    Hypercube& hc = entry->m_hc;

    PSDraw psdraw(epsname, hc.getLower()[0], hc.getLower()[1],
                   hc.getUpper()[0], hc.getUpper()[1]);

    const int top = a_toplevel < root->m_level ? a_toplevel : root->m_level;
    delete entry;
    delete root;
    Stack s;
    s.push((void*)a_rtree.m_memory.m_rootPageID);

    while (!s.isEmpty())
    {
        int pageid = (int)s.pop();
        RtreeNode* n = a_rtree.m_memory.loadPage(pageid);
        if (a_bottomlevel <= n->m_level && n->m_level <= a_toplevel)
        {
            for (int i=0; i<n->m_usedSpace; i++)
            {
              int aid = n->m_entry[i]->m_id;
              bool found = false;
              for (vector<int>::size_type h = 0; h < objid.size(); h++){
                 if(objid[h] == aid){
                   found = true;
                   psdraw.box(n->m_entry[i]->m_hc, 0, 1, true);
                   break;
                 }
              }
              if(!found)
                  psdraw.box(n->m_entry[i]->m_hc, 0, 1, false);
            }
        }
        if (n->m_level > a_bottomlevel)
        {
            for (int i=0; i<n->m_usedSpace; i++)
                s.push((void*)n->m_entry[i]->m_id);
        }
        delete n;
    }
   psdraw.queryPoint(c[0], c[1], true);
}
```

```
void helpmsg(const char* pgm)
{
   cerr << "Suggested arguments:" << endl;
   cerr << "> " << pgm << endl;
   cerr << "-p 4096 -d 2 -f raw_polygon.txt -i index_file.idx " << endl;
   cerr << "-q queryfile.txt -a0 startangle -a1 endangle " << endl;
   cerr << "-m tier -t #sectors -v true -k #neighbours" << endl << endl;
   cerr << "explanations:" << endl;
   cerr << "-p: pagesize, typically, 4096" << endl;
   cerr << "-f: raw polygon data file" << endl;
   cerr << "    a tab delimited file, format:" << endl;
   cerr << "    id #vertice x0 y0 x1 y1 ... xn yn" << endl;
   cerr << "-i: index file (rtree)" << endl;
   cerr << "-q: query file, format" << endl;
   cerr << "    id x y" << endl;
   cerr << "-a0: start angle (default: 0)" << endl;
   cerr << "-a1: end angle (default: 360)" << endl;
   cerr << "-m: tier (default: 1)" << endl;
   cerr << "-t: no. of sectors for partitioned processing (default: 1)" << endl;
   cerr << "-v: verbose mode on" << endl;
   cerr << "-k: number of neighbours to search (default: 1)" << endl;
}

int main(const int a_argc, const char** a_argv)
{
    if (a_argc == 1)
    {
        helpmsg(a_argv[0]);
        return -1;
    }

    cerr << "SWEEP k NSQ algorithm" << endl;
    //-------------------------------------------------------------------------
    // initialization
    //-------------------------------------------------------------------------
    int pagesize = atol(Param::read(a_argc, a_argv, "-p", ""));
    const char* filename = Param::read(a_argc, a_argv, "-f", "");
    const char* idxname = Param::read(a_argc, a_argv, "-i", "");
    const char* qryfilename = Param::read(a_argc, a_argv, "-q", "");
    const float startangle = (float)atof(Param::read(a_argc, a_argv, "-a0", "0.0"));
    const float endangle = (float)atof(Param::read(a_argc, a_argv, "-a1", "360.0"));
    const int m = atol(Param::read(a_argc, a_argv, "-m", "1"));
    const int t = atol(Param::read(a_argc, a_argv, "-t", "1"));
    const char* vrbs = Param::read(a_argc, a_argv, "-v", "null");
    bool verbose = strcmp(vrbs,"null") != 0;
    const int k = atol(Param::read(a_argc, a_argv, "-k", "1"));

    //-------------------------------------------------------------------------
    // load an R-tree from an index file
```

```
//---------------------------------------------------------------------------
cerr << "loading Rtree ... ";
const int maxChild = (pagesize - RtreeNode::size())/
           RtreeNodeEntry::size(DIMEN); // no. of entries per node
FileMemory mem(pagesize, idxname, RtreeNodeEntry::fromMem, false);
Rtree rtree(mem, DIMEN, maxChild, maxChild,
           (int)(maxChild*0.3), (int)(maxChild*0.3), false);
cerr << "[DONE]" << endl;

//---------------------------------------------------------------------------
// load polygons
//---------------------------------------------------------------------------
cerr << "loading polygons ... ";
fstream fin;
fin.open(filename, ios::in);
Hash hPolygon(1000000);
while (true)
{
  int id, edge;
  float sx[MAXEDGE], sy[MAXEDGE];
  float ex[MAXEDGE], ey[MAXEDGE];

  //-------------------------------------------------------------------
  // read a polygon record
  //-------------------------------------------------------------------
  fin >> id;
  if (fin.eof()) break;   // break if eof
  fin >> edge;
  for (int e=0; e<edge; e++)
  {
     fin >> sx[e];
     fin >> sy[e];
     fin >> ex[e];
     fin >> ey[e];
  }

  //-------------------------------------------------------------------
  // form a polygon
  //-------------------------------------------------------------------
  Polygon* plg = new Polygon;
  for (int e=0; e<edge; e++)
  {
     float c[2];
     c[0] = sx[e]; c[1] = sy[e];
     Point ptl(DIMEN, c);
     c[0] = ex[e]; c[1] = ey[e];
     Point ptu(DIMEN, c);
     LineSeg l(ptl, ptu);
     plg->addLineSeg(l);
```

```
      }
      hPolygon.put(id, plg);
   }
   fin.close();
   cerr << hPolygon.size() << " polygon objects are loaded." << endl;

   //------------------------------------------------------------------------
   // initialize the performance measurement metrics
   //------------------------------------------------------------------------
   int numquery = 0;
   int totalobj = 0;
   float aveProcTime = 0;      // average performance
   float aveObjExam = 0;
   float aveMemSize = 0;
   float aveNumPage0 = 0;
   float aveNumPage10 = 0;
   float aveNumPage30 = 0;
   float aveNumPage50 = 0;
   float aveNumObjs[10];       // assume at most 10 tiers

   float minProcTime = INFTY;  // minimum performance
   int minObjExam = INFTY;
   int minMemSize = INFTY;
   int minNumPage0 = INFTY;
   int minNumPage10 = INFTY;
   int minNumPage30 = INFTY;
   int minNumPage50 = INFTY;
   int minNumObjs[10];

   float maxProcTime = -INFTY;      // maximum performance
   int maxObjExam = -INFTY;
   int maxMemSize = -INFTY;
   int maxNumPage0 = -INFTY;
   int maxNumPage10 = -INFTY;
   int maxNumPage30 = -INFTY;
   int maxNumPage50 = -INFTY;
   int maxNumObjs[10];

   for (int i=0; i<m; i++)
   {
      aveNumObjs[i] = 0;
      minNumObjs[i] = INFTY;
      maxNumObjs[i] = -INFTY;
   }

   //------------------------------------------------------------------------
   // process Sweep NSQ algorithm based on read-in query points
   //------------------------------------------------------------------------
   fstream fqin;
```

```
fqin.open(qryfilename, ios::in);
int fileDraw = 0;
while (true)
{
   int id;
   float c[DIMEN];

   //-------------------------------------------------------------------
   // load a query point
   //-------------------------------------------------------------------
   fqin >> id;
   if (fqin.eof()) break;
   fqin >> c[0];
   fqin >> c[1];
   Point q(DIMEN, c);

   //-------------------------------------------------------------------
   // perform the search
   //-------------------------------------------------------------------
   Array res[10];
   int maxexam = 0;
   int maxqlen = 0;
   Array pageaccessed;
   float avewidth = (endangle - startangle)/t;

   struct timeb starttime, endtime;
   ftime(&starttime);
   for (int i=0; i<t; i++)
   {
      AngleRange ar(i*avewidth+startangle, (i+1)*avewidth+startangle);
      NSSearch::sweep(rtree, hPolygon, q, ar, m, res[i], maxexam,
                                              maxqlen, pageaccessed);
   }
   ftime(&endtime);
   float qtime = ((endtime.time*1000 + endtime.millitm) -
              (starttime.time*1000 + starttime.millitm)) / 1000.0f;


   //-------------------------------------------------------------------
   // carrier (base). It is used to order the results.
   //-------------------------------------------------------------------
   class carrier
   {
     public:
        const AngleRange m_ar;
        const float m_mindist2q;
        const int m_type;
     public:
        carrier(const AngleRange& a_ar,
```

```
        const float a_mindist2q, const int a_type):
        m_ar(a_ar), m_mindist2q(a_mindist2q), m_type(a_type) {};
     virtual ~carrier() {};

  static int distcompare(const void* a0, const void* a1)
  {
      carrier* p0 = *(carrier**)a0;
      carrier* p1 = *(carrier**)a1;
      if (p0->m_mindist2q  < p1->m_mindist2q)  return -1;
      if (p0->m_mindist2q  > p1->m_mindist2q)  return +1;
      return 0;
   };
};

//---------------------------------------------------------------------------
// carrier of a edge entry. It is used to order the results.
//---------------------------------------------------------------------------
class edgecarrier: public carrier
{
  public:
     LineSeg m_lineseg;
     const int m_oid;
  public:
     edgecarrier(const LineSeg& a_lineseg, const int a_oid,
       const AngleRange& a_ar, const float a_mindist2q):
       m_lineseg(a_lineseg), m_oid(a_oid),
       carrier(a_ar, a_mindist2q, EDGE) {};
     virtual ~edgecarrier() {};
};
//-----------------------------------------------------------------
// This loop orders the objects of the result according to
// their distance of the query point
//-----------------------------------------------------------------
BinHeap h(carrier::distcompare);
for (int l=0; l<t; l++)
{
   for (int i=0; i<res[l].size(); i++)
   {
      SweepNSResult* r = (SweepNSResult*)res[l].get(i);
      for (int j=0; j<r->m_result.size(); j++)
      {
        NSResult* nsr = (NSResult*)r->m_result.get(j);
        float mindist = INFTY;
        if (nsr->m_oid != -1){
           mindist = nsr->m_lineseg.mindist(q, r->m_ar);
           h.insert(new edgecarrier(nsr->m_lineseg, nsr->m_oid,
                                                  r->m_ar, mindist));

        }
```

```
        }
    }
}
Set koids; // It keeps the first k objects of the ordered heap.
vector<int> myVector; // used to draw the k nearest objects
while(!h.isEmpty() && koids.size()<k){
    edgecarrier* ec = (edgecarrier*)h.removeTop();
    if(!koids.in((void*)ec->m_oid)){
       koids.insert((void*)ec->m_oid);
       myVector.push_back(ec->m_oid);
    }
    delete ec;
}


//---------------------------------------------------------------------
// This loop is used to print the results uniformly. It prints the k VNNs
// and its corresponding angular range.
//---------------------------------------------------------------------
if (verbose)
{
   cerr << "q@: " << c[0] << "," << c[1] <<", id: "<<id<< endl;
   for (int l=0; l<t; l++)
   {
       AngleRange angl;
       int lastobj = -100;
       for (int i=0; i<res[l].size(); i++)
       {
           SweepNSResult* r = (SweepNSResult*)res[l].get(i);
           for (int j=0; j<r->m_result.size(); j++)
           {
               NSResult* nsr = (NSResult*)r->m_result.get(j);
               int oid = nsr->m_oid;
               if(!koids.in((void*)oid)) oid = -1;
               if (lastobj == -100){
                   angl.set(r->m_ar.start(), r->m_ar.end());
               }
               else if (lastobj == oid && r->m_ar.end()!=360){
                   angl.set(angl.start(), r->m_ar.end());
               }
               else if (lastobj == oid && r->m_ar.end()==360){
                   cerr << "("<<angl.start()<<", "<<r->m_ar.end()<<"):"<<
                                                          lastobj<<endl;
               }
               else if (lastobj != oid && r->m_ar.end()==360){
                   cerr << "("<<angl.start()<<", "<<angl.end()<<"):"<<
                                                          lastobj<<endl;
                   cerr << "("<<r->m_ar.start()<<", "<<r->m_ar.end()<<"):"<<
                                                               oid<<endl;
```

```
                }
                else if (lastobj != oid){
                    cerr << "("<<angl.start()<<", "<<angl.end()<<"):"<<
                                                                lastobj<<endl;
                    angl.set(r->m_ar.start(), r->m_ar.end());
                }
                lastobj = oid;
            }
        }
    }
}

cerr << "qtime: " << qtime;
cerr << " io: " << pageaccessed.size();
cerr << " #res: ";

int tier[5], obj[5], prevoid[5];
for (int i=0; i<m; i++)
{
   tier[i]=obj[i]=0;
   prevoid[i]=-100;
}
Set objid[5];

for (int l=0; l<t; l++)
{
   for (int i=0; i<res[l].size(); i++)
   {
      SweepNSResult* r = (SweepNSResult*)res[l].get(i);
      if (r->m_ar.width() == 0) continue;
      for (int j=0; j<r->m_result.size(); j++)
      {
          NSResult* nr = (NSResult*)r->m_result.get(j);
          if (nr->m_oid != prevoid[j])
          {
             tier[j]++;
             if (prevoid[j] >= 0)
             {
                 objid[j].insert((void*)prevoid[j]);
             }
             prevoid[j] = nr->m_oid;
          }
      }
   }
}
for (int i=0; i<m; i++)
{
    if (prevoid[i] > 0)
    {
```

```
        objid[i].insert((void*)prevoid[i]);
    }
    tier[i]++;
    obj[i] = objid[i].size();
    cerr << "(tier-" << i << ") " << tier[i] << "(" << obj[i] << ") ";
}
totalobj = koids.size();
cerr << endl;

//convert int to string
std::string s;
std::stringstream out;
out << fileDraw;
s = out.str();
//convert string to char*
const char *p;
p=s.c_str();
//concatanate chars
char str[100];
strcpy (str,"sweepkNSResult_");
strcat (str, p);
strcat(str, ".eps");

drawResult(rtree, 0, 100, str, myVector, c);

//-----------------------------------------------------------------------
// performance measurement
//-----------------------------------------------------------------------

// average performance
int io0 = pageaccessed.size();
int io10 = IOMeasure::lru(pageaccessed, 10);
int io30 = IOMeasure::lru(pageaccessed, 30);
int io50 = IOMeasure::lru(pageaccessed, 50);
aveProcTime += qtime;
aveNumPage0  += io0;
aveNumPage10 += io10;
aveNumPage30 += io30;
aveNumPage50 += io50;
for (int i=0; i<m; i++)
   aveNumObjs[i] += obj[i];
aveObjExam += maxexam;
aveMemSize += maxqlen;
numquery++;

// max performance
maxProcTime  = maxProcTime > qtime ? maxProcTime : qtime;
maxNumPage0  = maxNumPage0  > io0  ? maxNumPage0  : io0;
maxNumPage10 = maxNumPage10 > io10 ? maxNumPage10 : io10;
```

```
      maxNumPage30 = maxNumPage30 > io30 ? maxNumPage30 : io30;
      maxNumPage50 = maxNumPage50 > io50 ? maxNumPage50 : io50;
      for (int i=0; i<m; i++)
         maxNumObjs[i] = maxNumObjs[i] > obj[i] ? maxNumObjs[i] : obj[i];
      maxObjExam = maxObjExam > maxexam ? maxObjExam : maxexam;
      maxMemSize = maxMemSize > maxqlen ? maxMemSize : maxqlen;

      // min performance
      minProcTime  = minProcTime < qtime ? minProcTime : qtime;
      minNumPage0  = minNumPage0  < io0  ? minNumPage0  : io0;
      minNumPage10 = minNumPage10 < io10 ? minNumPage10 : io10;
      minNumPage30 = minNumPage30 < io30 ? minNumPage30 : io30;
      minNumPage50 = minNumPage50 < io50 ? minNumPage50 : io50;
      for (int i=0; i<m; i++)
         minNumObjs[i] = minNumObjs[i] < obj[i] ? minNumObjs[i] : obj[i];
      minObjExam = minObjExam < maxexam ? minObjExam : maxexam;
      minMemSize = minMemSize < maxqlen ? minMemSize : maxqlen;

      //------------------------------------------------------------------------
      // clean up
      //------------------------------------------------------------------------
      for (int l=0; l<t; l++)
      {
         for (int i=0; i<res[l].size(); i++)
            delete (SweepNSResult*)res[l].get(i);
         res[l].clean();
      }
      pageaccessed.clean();
   }
   fqin.close();
   cout << "---Sweepkns Performance Evaluation---" << endl;
   cout << "#query: " << numquery<< endl;
   cout << "m: " << m<< endl;
   cout << "t: " << t<< endl;
   cout << "ar: " << startangle << " - " << endangle<< endl;
   cout << "proctime(min,ave,max): " << minProcTime << ", " <<
                        aveProcTime/numquery << ", " << maxProcTime<< endl;
   cout << "objexam(min,ave,max): " << minObjExam << ", " <<
                        aveObjExam/numquery << ", " << maxObjExam<< endl;
   cout << "memsize(min,ave,max): " << minMemSize << ", " <<
                        aveMemSize/numquery << ", " << maxMemSize<< endl;
   cout << "io0(min,ave,max): " << minNumPage0 << ", " <<
                        aveNumPage0/numquery << ", " << maxNumPage0<< endl;
   cout << "io10(min,ave,max): " << minNumPage10 << ", " <<
                        aveNumPage10/numquery << ", " << maxNumPage10<< endl;
   cout << "io30(min,ave,max): " << minNumPage30 << ", " <<
                        aveNumPage30/numquery << ", " << maxNumPage30<< endl;
   cout << "io50(min,ave,max): " << minNumPage50 << ", " <<
                        aveNumPage50/numquery << ", " << maxNumPage50<< endl;
```

```
   cout << "totalobj: " << totalobj<< endl;
   for (int i=0; i<m; i++)
      cout << "res(min,ave,max): " << minNumObjs[i] << ", " <<
                      aveNumObjs[i]/numquery << ", " << maxNumObjs[i]<< endl;
   cout << endl;

   return 0;
}
```

A.5 Αποσφαλμάτωση Συναρτήσεων

Ακολούθως παρουσιάζονται δύο συναρτήσεις του αρχικού κώδικα που μας δόθηκε και στις οποίες χρειάστηκε να γίνει αποσφαλμάτωση διότι εξαιτίας τους επιστρέφονταν null τιμές στο αποτέλεσμα. Οι συναρτήσεις αυτές είναι η acrossX της κλάσης polygon.cc και η συνάρτηση angle της κλάσης point.cc

```
//Determine if a polygon is across the positive x-axis.
bool Polygon::acrossX(const Point& a_pt) const
{
        float min[2];
        float max[2];
        min[0] = min[1] = INFTY;
        max[0] = max[1] = -INFTY;
        for (int i=0; i<m_lineSegs.size(); i++)
        {
                LineSeg* l = (LineSeg*)m_lineSegs.get(i);
                const Point start = l->m_p0;
                const Point end = l->m_p1;

                for (int d=0; d<2; d++)
                {
                        min[d] = min[d] < start[d] ? min[d] : start[d];
                        max[d] = max[d] > start[d] ? max[d] : start[d];
                        min[d] = min[d] < end[d] ? min[d] : end[d];
                        max[d] = max[d] > end[d] ? max[d] : end[d];
                }
        }
        if ((a_pt[1] > min[1]) &&
           (a_pt[1] < max[1])) return true;   // the polygon covers the x-axis
        if ((a_pt[1] > min[1]) &&
           (a_pt[1] == max[1])) return false; // touching and below x-axis
        if (a_pt[1] <= min[1]) return false;   // the polygon above x-axis
        if (a_pt[1] > max[1]) return false;    // the polygon below x-axis
        return true;
}

//Returns the angle of a point according to another point.
float Point::angle(const Point& a_pt) const
```

```
{  // this function considers 2d only
    double xdiff = m_coor[0] - a_pt.m_coor[0];
    double ydiff = m_coor[1] - a_pt.m_coor[1];
    bool xpos = xdiff > 0;
    bool ypos = ydiff > 0;

    if(xdiff==0){
      if(a_pt.m_coor[0]>0) xpos=true;
      else if(a_pt.m_coor[0]==0){
         if(m_coor[1]>0) xpos=true;
         else xpos=false;
      }
      else xpos=false;
    }
    if(ydiff==0){
      if(a_pt.m_coor[1]>0) ypos=true;
      else if(a_pt.m_coor[1]==0){
        if(m_coor[0]>0) ypos=true;
        else ypos=false;
      }
      else ypos=false;
   }

   xdiff = xdiff > 0 ? xdiff : -xdiff;
   ydiff = ydiff > 0 ? ydiff : -ydiff;
   double radian = atan(ydiff/xdiff);
   float degree = (float)(radian * 180 / PI);
   degree = degree > 0 ? degree : - degree;

   if (xpos)
   {
     if (ypos){      // quadrant I (+ve x-axis and +ve y-axis)
           return degree;
     }
     else{           // quadrant IV (-ve x-axis and +ve y-axis)
           return 360 - degree;
     }
   }
   else
   {
     if (ypos){    // quadrant II (-ve x-axis and -ve y-axis)
         return 180 - degree;
     }
     else{        // quadrant III (-ve x-axis and -ve y-axis)
         return degree + 180;
     }
   }
}
```

Βιβλιογραφία

1. S. Nutanong, E. Tanin, and R. Zhang. *Visible Nearest Neighbor Queries.* Advances in Databases: Concepts, Systems and Applications. Lecture Notes in Computer Science, Volume 4443/2007, pp. 876-883, 2007.
2. Y. Gao, B. Zheng, W.-C. Lee, and G. Chen. *Continuous Visible Nearest Neighbor Queries.* Proceeding EDBT '09 Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, ACM, New York, USA, pp. 144-155, 2009.
3. K. C. K. Lee, W.-C. Lee, and H. V. Leong. *Nearest Surrounder Queries.* Proceedings of IEEE 22nd International Conference on Data Engineering (ICDE'06), USA, 85 (April), pp. 85, 2006.
4. K. C. K. Lee, J. Schiffman, B. Zheng, W.-C. Lee, and H.V. Leong. *Round-Eye: A system for tracking nearest surrounders in moving object environments.* Journal of Systems and Software, Volume 80, Issue 12, pp. 2063-2076, December 2007.
5. Y. Gao, B. Zheng, G. Chen, W.-C. Lee, K. C. K. Lee, and Q. Li. *Visible Reverse k-Nearest Neighbor Queries.* Proceedings of IEEE International Conference on Data Engineering (ICDE'A09), Shanghai, Issue Date March 29 2009-April 2 2009, pp. 1203-1206, 2009.
6. S. Nutanong, E. Tanin, and R. Zhang. *Incremental Evaluation of Visible Nearest Neighbor Queries.* Journal of Knowledge and Data Engineering, IEEE Transactions, Volume 22, Issue 5, pp. 665-681, May 2010.
7. S. Shekhar and S. Chawla. *Spatial Databases: A Tour.* Prentice Hall, 2003.
8. R. H. Guting. *An Introduction to Spatial Database Systems.* The VLDB Journal - The International Journal on Very Large Data Bases - Spatial Database Systems, Volume 3, Issue 4, pp. 357-399, October 1994.
9. B. R. Vatti. *A generic solution to polygon clipping.* Commun. ACM, 35(7):56-63, 1992.
10. Y. Theodoridis. *The R-tree Portal (2003)*, http://www.rtreeportal.org, [Tiger1] and [Tiger2] data sets in http://www.rtreeportal.org
11. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. *The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles.* SIGMOD Conference, 1990.
12. A. Guttman. *R-trees: a dynamic index structure for spatial searching.* SIGMOD, pp. 47-57, 1984.

13. Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos, and Y. Theodoridis. *R-trees: Theory and Applications*. Springer-Verlag, 2005.
14. Y. Theodoridis, E. Stefanakis, and T. Sellis. *Efficient cost models for spatial queries using R-trees*. IEEE Trans. Knowl. Data Eng. Volume 12, 1, pp. 19-32, 2000.
15. J. Zhang, D. Papadias, K. Mouratidis, and M. Zhu. *Spatial Queries in the Presence of Obstacles*. In E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Bohm, E. Ferrari, (eds.) EDBT 2004. LNCS, Volume 2992, pp. 366-384, 2004.
16. S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. *The v*-diagram: a query dependent approach to moving kNN queries*. VLDB, pp. 1095-1106, 2008.
17. Man Lung Yiu, E. Lo, and D. Yung. *Authentication of moving kNN queries*. IEEE 27th International Conference on Data Engineering (ICDE), pp.565-576, April 2011.
18. B. Gedik and L. Liu. *MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system*. Proceedings of 9th International Conference on Extending Database Technology (EDBT'A04), Greece, pp. 67-87, March 2004.
19. Z. Song, and N. Roussopoulos. *K-nearest neighbor search for moving query point*. Advances in Spatial and Temporal Databases. Lecture Notes in Computer Science, Volume 2121, pp. 79-96, 2001.
20. W. Wu, W. Guo, and K.L. Tan. *Distributed processing of moving k-nearest-neighbor query on moving objects*. ICDE, pp. 1116-1125, 2007.
21. X. Yu, K. Pu, and N. Koudas. *Monitoring k-nearest neighbor queries over moving objects*. ICDE, pp. 631-642, 2005.
22. G.S. Iwerks, H. Samet, and K. Smith. *Continuous k-nearest neighbor queries for continuously moving points with updates*. Proceedings of 29th International Conference on Very Large Data Bases, Germany, pp. 512-523, September 2003.
23. L. Kulik, and E. Tanin. *Incremental rank updates for moving query points*. GISience, pp. 251-268, 2006.
24. Y. Li, J. Yang, and J. Han. *Continuous k-nearest neighbor search for moving objects*. SSDBM, pp. 123-126, 2004.
25. H. Hu, J. Xu, and D.L. Lee. *A generic framework for monitoring continuous spatial queries over moving objects*. Proceedings of ACM SIGMOD, 2005.
26. E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis. *Algorithms for nearest neighbor search on moving objects trajectories*. Geoinformatica Journal, Volume 11, Issue 2, June 2007.
27. Y. Gao, and B. Zheng. *Continuous obstructed nearest neighbor queries in spatial databases*. SIGMOD, pp. 577-590, 2009.
28. Y. Tao, D. Papadias, and Q. Shen. *Continuous nearest neighbor search*. Proceedings of VLDB, 2002.
29. M.R. Kolahdouzan, and C. Shahabi. *Alternative solutions for continuous k nearest neighbor queries in spatial network databases*. Geo-Informatica 9(4), pp. 321-341, 2005.
30. X. Xiong, M.F. Mokbel, and W.G Aref. *SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatiotemporal databases*. ICDE, pp. 643-654, 2005.
31. Y. Gao, B. Zheng, G. Chen, Q. Li, and X. Guo. *Continuous visible nearest neighbor query processing in spatial databases*. The VLDB Journal - The Inter-

national Journal on Very Large Data Bases. Volume 20, Issue 3, pp. 371-396, June 2011.
32. H. Xu, Z. Li, Y. Lu, K. Deng, and X. Zhou. *Group Visible Nearest Neighbor Queries in Spatial Databases.* Web-Age Information Management, Lecture Notes in Computer Science, Volume 6184, pp. 333-344, 2010.
33. D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. *Group nearest neighbor queries.* ICDE, pp. 301-312, 2004.
34. D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. *Aggregate nearest neighbor queries in spatial databases.* ACM Trans. Database Syst., Volume 30, Issue 2 pp. 529-576, 2005.
35. K. F. Yanmin Luo, Hanxiong Chen and N. Ohbo. *Efficient methods in finding aggregate nearest neighbor by projection-based filtering.* ICCSA, pp. 821-833, 2007.
36. H. Hu and D. L. Lee. *Range nearest neighbor query.* TKDE, Volume 18 Issue 1 pp. 78-91, 2006.
37. R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. *Nearest and reverse nearest neighbour queries for moving objects.* VLDB Journal, Volume 15, no. 3, pp. 229-250, 2006.
38. C. Yang and K.-I Lin. *An index structure for efficient reverse nearest neighbour queries.* ICDE, pp. 485-492, 2001.
39. A. Singh, H Ferhatosmanoglu, and A. Tosun. *High dimensional reverse nearest neighbour queries.* CIKM, pp. 91-98, 2003.
40. Y. Tao, D. Papadias, X. Lian, and X. Xiao. *Multidimensional reverse kNN search.* VLDB J. 16(3), pp. 293-316, 2007.
41. F. Korn, S. Muthukrishnan, and D. Srivastava. *Reverse nearest neighbor aggregates over data streams.* VLDB, pp. 814-815, 2002.
42. J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. *All nearest neighbors queries in spatial databases.* Proceedings of the 16th International Conference on Scientific and Statistical Database Management, pp. 297-306, June 2004.
43. J. Sankaranarayanan, H. Samet, and A. Varshney. *A fast all nearest neighbor algorithm for applications involving large point-clouds.* Computers & Graphics, Volume 31, Issue 2, pp. 157-174, April 2007.
44. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. *Algorithms for processing K-closest-pair queries in spatial databases.* Data & Knowledge Engineering, Volume 49, Issue 1, April 2004.