CircularTrip and ArcTrip: Effective Grid Access Methods for Continuous Spatial Queries

by

Muhammad Aamir Cheema

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

A THESIS SUBMITTED IN FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTERS IN COMPUTER SCIENCE & ENGINEERING RESEARCH
(March 2007)

Supervisor: Dr. Xuemin Lin

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: **Cheema**	
First name: **Muhammad**	Other name/s: **Aamir**
Abbreviation for degree as given in the University calendar: **Masters by research in Computer Science and Engineering (COMPER2665)**	
School: **School of Computer Science and Engineering (CSE)**	Faculty: **Engineering**

Title: **CircularTrip and ArcTrip: Effective Grid Access Methods for Continuous Spatial Queries**

Abstract 350 words maximum: (PLEASE TYPE)

A k nearest neighbor query q retrieves k objects that lie closest to the query point q among a given set of objects P. With the availability of inexpensive location aware mobile devices, the continuous monitoring of such queries has gained lot of attention and many methods have been proposed for continuously monitoring the kNNs in highly dynamic environment. Multiple continuous queries require real-time results and both the objects and queries issue frequent location updates. Most popular spatial index, R-tree, is not suitable for continuous monitoring of these queries due to its inefficiency in handling frequent updates. Recently, the interest of database community has been shifting towards using grid-based index for continuous queries due to its simplicity and efficient update handling. For kNN queries, the order in which cells of the grid are accessed is very important. In this research, we present two efficient and effective grid access methods, CircularTrip and ArcTrip, that ensure that the number of cells visited for any continuous kNN query is minimum. Our extensive experimental study demonstrates that CircularTrip-based continuous kNN algorithm outperforms existing approaches in terms of both efficiency and space requirement. Moreover, we show that CircularTrip and ArcTrip can be used for many other variants of nearest neighbor queries like constrained nearest neighbor queries, farthest neighbor queries and (k+m)-NN queries. All the algorithms presented for these queries preserve the properties that they visit minimum number of cells for each query and the space requirement is low. Our proposed techniques are flexible and efficient and can be used to answer any query that is hybrid of above mentioned queries. For example, our algorithms can easily be used to efficiently monitor a (k+m) farthest neighbor query in a constrained region with the flexibility that the spatial conditions that constrain the region can be changed by the user at any time.

Approval

This thesis entitled

CircularTrip and ArcTrip: Effective Grid Access Methods for Continuous Spatial Queries

by

Muhammad Aamir Cheema

has been approved by the School of Computer Science and Engineering,
The University of New South Wales, Syndey Australia.

Supervisor: Dr. Xuemin Lin

The final copy of this thesis has been examined by the signatory and I find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Signed ....
Date ....

Copyright Statement

Authenticity Statement

Originality Statement

Dedication

I dedicate this thesis to my parents who have always been my nearest and reverse nearest neighbors[1] [KM00] and have been so close to me that I found them with me whenever I needed. It is their unconditional love that motivates me to set higher targets. I also dedicate this thesis to my sisters (Shazia Cheema and Nadia Cheema) and brothers (Muhammad Omer Cheema, Muhammad Umair Cheema and Muhammad Ammar Cheema) who are my nearest surrounders[2] [LLL06] and have provided me with a strong love shield that always surrounds me and never lets any sadness enter inside.

[1]In a not-so-academic wording, your nearest neighbor is an object that is closest to you and your reverse nearest neighbor is any object for which you are the closest object.

[2]Your nearest surrounders are the objects that are nearest to you and surround you completely.

Acknowledgements

I am very thankful to my parents for their love, support and encouragement and for being with me on each and every step of my life. I am also very thankful to my eldest brother Muhammad Omer Cheema for being very supporting and motivating.

I wish to express my gratitude to my supervisor Dr. Xuemin Lin for his help and guidance. I feel thankful to him for his insightful advice and suggestions. I am also thankful to Dr. Wei Wang for giving me generous amount of time whenever I needed some help. I am grateful to Yidong Yuan with whom I worked during my research and learnt a lot. I feel myself very lucky that I got very friendly colleagues. In particular, I am thankful to Ying Zhang for being very helping and cooperative, Mahady Hasan for being my closest colleague (it is also true literally, he sits very close to me), Yi Luo especially for arranging group meetings, Wenjie Zhang especially for being the only other colleague who is working on moving objects and Bin Jiang especially for being the only masters student in our research group other than me.

I do not want to miss this opportunity to thank people who made my stay in Australia an enjoyable experience without which I would not have been able to concentrate on my research. First of all I am very thankful to Azeem Ahmad Cheema, my cousin who is more like brothers, for being so caring and fun loving that made my stay in Australia a memorable period of my life. I am also grateful to his family, my other brother Waseem Ahmad Cheema and his family for being very caring and affectionate. Their kids gave me so many reasons to smile. I also feel blessed that my brother and best friend Umair was also here with me and I never felt lonely in his presence. I am also thankful to Assad Mehmood for giving me company during coffee breaks.

I feel grateful to all my friends back in Pakistan for being in contact with me and encouraging me. I am thankful to my younger brother and my crime partner Ammar for keeping me informed about family events in his own special way. I am also thankful to my best friends (they are best at finding something in me to complain about), Mohsin Hasan and Samar Cheema, for being very caring and for keeping me up-to-date of issues and events happening in Pakistan especially in our friends' circle. I am thankful to my intimate friend Irfan Ahmad for being too busy to be in contact. I am also thankful to my sweet cousin Farman Ahmad Bajwa for always remembering me and especially for always making me angry by telling me not to be angry. Finally, I am very grateful to Daisy for giving me so many valuable suggestions for this part of my thesis. The best advice she gave me was to include her name in acknowledgments.

I feel obliged to say "thanks" to so many people who have been doing so many good things for me but I am afraid the list may go forever. Same as mathematicians write all positive even numbers in very small space as $\{0, 2, 4, ..., \infty\}$, below I try to write a shorter acknowledgements. Let k be a positive integer, T be my lifetime, and $dist(p, q)$ be the distance between the hearts of any two people p and q reflecting the love the person p has for q, I feel grateful to all people who have ever been among my k nearest neighbors and/or k reverse nearest neighbors over the time period T.

Abstract

A k nearest neighbor query q retrieves k objects that lie closest to the query point q among a given set of objects P. With the availability of inexpensive location aware mobile devices, the continuous monitoring of such queries has gained lot of attention and many methods have been proposed for continuously monitoring the kNNs in highly dynamic environment. Multiple continuous queries require real-time results and both the objects and queries issue frequent location updates. Most popular spatial index, R-tree, is not suitable for continuous monitoring of these queries due to its inefficiency in handling frequent updates. Recently, the interest of database community has been shifting towards using grid-based index for continuous queries due to its simplicity and efficient update handling. For kNN queries, the order in which cells of the grid are accessed is very important. In this research, we present two efficient and effective grid access methods, CircularTrip and ArcTrip, that ensure that the number of cells visited for any continuous kNN query is minimum. Our extensive experimental study demonstrates that CircularTrip-based continuous kNN algorithm outperforms existing approaches in terms of both efficiency and space requirement. Moreover, we show that CircularTrip and ArcTrip can be used for many other variants of nearest neighbor queries like constrained nearest neighbor queries, farthest neighbor queries and $(k + m)$-NN queries. All the algorithms presented for these queries preserve the properties that they visit minimum number of cells for each query and the space requirement is low. Our proposed techniques are flexible and efficient and can be used to answer any query that is hybrid of above mentioned queries. For example, our algorithms can easily be used to efficiently monitor a $(k + m)$ farthest neighbor query in a constrained region with the flexibility that the spatial conditions that constrain the region can be changed by the user at any time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A spatial database system can be defined as a database system that offers spatial data types in its data model and query language, and supports spatial data types in its implementation, providing at least spatial indexing and spatial join methods [Gut94]. The spatial objects are composed of one or more points, lines and/or polygons. Fig. 1.1 shows a map from Google Maps (http://maps.google.com) obtained by entering a query "Find computer related businesses near University of New South Wales". It shows different representations of spatial objects. i.e., points, lines, regions. A point may represent a data object for which only its location is important and its extent in space is not important. For example, the balloons labelled A to J point to the locations of computer shops around the university. The lines represent the facilities of moving through space or connections in space (i.e., roads, rivers). A region represents the spatial object for which its spatial extent is also important. A region may consist of disjoint pieces each containing many polygons. In the figure, University of New South Wales and Prince of Wales Hospital are represented by regions.

Spatial databases are also termed as *image, pictorial, geometric* and *geographic* databases. The application of spatial databases include *Geographic Information Systems* (GIS), *Computer Aided Design* (CAD), *Very-Large-Scale Integration* (VLSI) designs, *Multimedia Information System* (MMIS) and medicine and biological research.

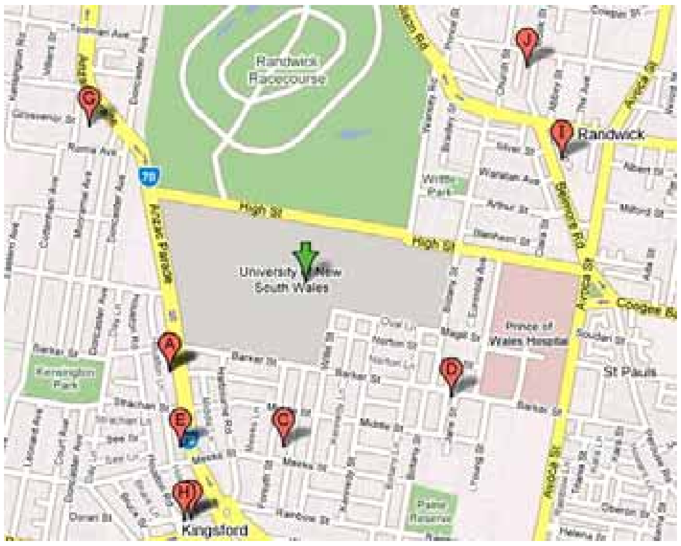


Figure 1.1: Google Maps (http://maps.google.com)

Additional functionality must be added for databases to process spatial data objects because spatial data objects have complex structure and are multidimensional and there is no standard algebra defined for spatial data. Moreover, the spatial data objects are usually dynamic and the storage structure should allow efficient insertions and deletions of data. A spatial database needs to support different kind of spatial queries. For example, a query may be issued to find all police stations in Sydney. Logically, this query is to find one kind of spatial objects (police stations) that are contained by the other type of spatial object (Sydney). Another example which we already have seen is a query to find the computer shops near University of New South Wales. The spatial database needs to find the spatial objects (computer shops) that are closer to some other spatial object (the university).

To support the search operations on spatial data objects, special data structures are needed to be designed. These data structures are usually called *spatial indexes* or *spatial access methods*. In Section 1.1, we briefly describe why traditional one-dimensional indexes cannot be used in spatial databases and we also discuss the important requirements a good spatial access method should meet. In Section 1.2, we describe few basic and advanced spatial queries. Section 1.3 summarizes the contributions of this thesis towards spatial databases. Thesis organization is presented in Section 1.4.

1.1 Spatial Access Methods

The main problem in design of spatial access methods is that there is no total ordering among the spatial data objects that preserves spatial proximity. Consider, for example, a user wants to find 5 restaurants closest to her location. One try to answer this query is to build a one-dimensional index that contains the distances of all restaurants from user's location sorted in ascending order. To answer her query, we can return first 5 entries from the sorted index. However, this index cannot support a query issued by some other user at a different location. In order to answer the query of this new user, we will have to sort all the restaurants again in ascending order of their distances from this user. The difficulty lies in the fact that there is no mapping from multidimensional space into one-dimensional space so that the objects that are close in multidimensional space are also close in the one-dimensional sorted index [GG98].

Another way to answer the queries of above type is to sort each restaurant in two lists. First list ($List_x$) contains the restaurants sorted in ascending order on their x-coordinates. The second list ($List_y$) may contain them sorted according to their y-coordinates. In order to find 5 closest restaurants to a point p located at (p_x, p_y), we may first find few candidates from $List_x$ that are closest to p_x and then we could calculate their actual distance from p by looking their value in $List_y$. However, this approach can be very in-efficient because a restaurant that is closest to p in x-dimension may be the farthest restaurant in y-dimension. Consider the example of Fig. 1.2 where the object closest to p_1 is to be found. The objects p_3 and p_4 are the closest objects to p_1 in x and y dimensions, respectively. However, the nearest object in the two-dimensional space is p_2 which is not the closest object in any single dimension.

For the reason mentioned above, traditional one-dimensional access methods like B-tree [BM72] and extendible hashing [FNPS79] are not suitable for spatial databases. An

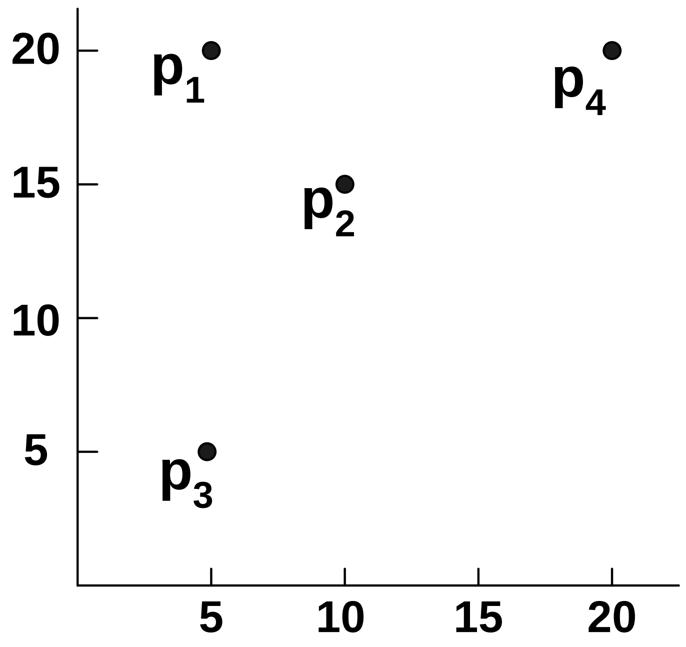


Figure 1.2: Difficulty in Finding the Spatial Proximity of Objects

excellent survey on multidimensional access methods by Gaede *et al.* can be found in [GG98]. Gaede *et al.* describe the requirements that multidimensional access methods should meet, based on the properties of spatial data and their applications.

- **Dynamics:** Spatial data objects can be inserted and deleted in any order and the spatial access methods should be able to continuously record the changes.
- **Secondary/tertiary storage management:** Even though the size of main memory is increasing, it is not always possible to store the complete database in main memory. The access methods need to integrate secondary and tertiary storage in seemless manner.
- **Broad ranges of supported operations:** The access methods should not support only one particular type of operation at the expense of other operations. For example, an access method is not good if insertion of data points into it is efficient but other operations like deletion of the points are slow.
- **Independence of the input data and insertion sequence:** The access meth-

ods should maintain their efficiency even when input data are highly skewed or the insertion sequence is changed.

- **Simplicity:** Complex access methods are usually difficult to implement and are error-prone thus not suitable for large-scale applications.
- **Scalability:** The access method should adapt well with the database growth.
- **Time efficiency:** The access method should provide fast spatial searches.
- **Space efficiency:** An index should be small in size compared to the data to be addressed and therefore it must guarantee a certain storage utilization.
- **Concurrency and recovery:** In modern databases, multiple users run operations on databases concurrently. The access methods should provide robust techniques for transaction management without significant performance penalties.
- **Minimum impact:** The integration of an access method into a database system should have minimum impact on the existing parts of the system.

There have been many spatial access methods proposed in last two decades. Some of the popular spatial indexes are R-tree [Gut84] and its variants(e.g. R*-tree [BKSS90] and R^+-tree [SSH86]), *kd*-tree [Ben75] and the grid file [NHS84].

1.2 Spatial Queries

First we describe some very basic spatial queries and then we will define relatively complex queries. One or more basic queries may be issued to answer these complex queries. Let A and B be two multidimensional spatial data objects (points, lines or regions), below we define some very basic spatial queries.

- **LENGTH(A).** Return the length of a spatial object (line) A.

- **AREA(A).** Return the area of any 2D region A.

- **CENTROID(A).** Return the centroid of a spatial object A. Centroid of an n-dimensional object is the intersection of all hyperplanes that divide it into two parts of equal moment about the hyperplane. Informally, it is the "average" of all points of A.

- **DISTANCE(A, B).** Find the distance between A and B. If A and/or B are not points, the distance definitions may be needed to be defined by the user. For example, possible definitions of distance between the two objects may be the minimum or maximum distance between them. Some other definition can also be applied. For example, find the distance between the centroids of the two objects.

- **EQUALS(A, B).** If A and B have same spatial extent, return true.

- **DISJOINT(A, B).** Return true if A and B are disjoint (they do not have any point in common).

- **INTERSECTS(A, B).** Return true if A and B intersect each other. Note that this is exactly opposite operation to the disjoint operation.

- **CONTAINS(A, B).** If the object B is fully contained in A, return true.

Note that for all of the above queries, the database system does not need to access any other object from the spatial index. These operations involve geometrical computation based on the information of the two spatial objects A and B. Next we briefly describe the queries that are more advanced and in order to answer these queries, the database system may need to traverse spatial indexes or it may need to call few of the basic operations described above.

1. **Spatial Range Query($Q, dist$).** Find all the objects that lie within a given distance $dist$ from the query object Q. In other words, find all objects X such that DISTANCE(Q, X)$\leq$ $dist$. For example, find all the restaurants that are within 10 km from my current location.

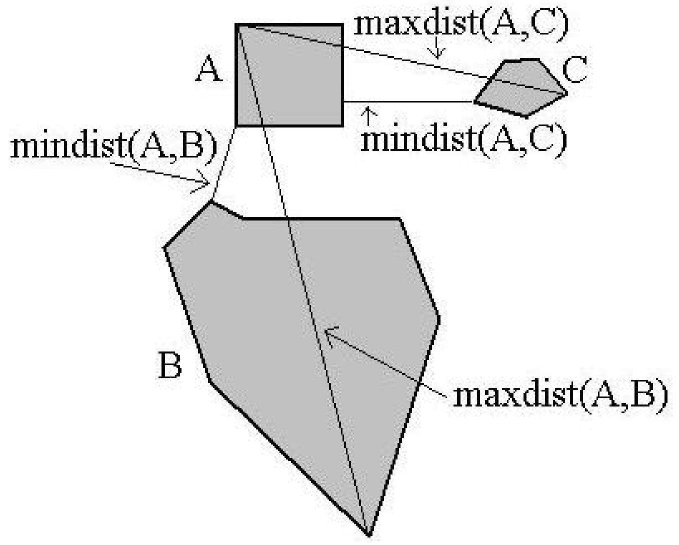


Figure 1.3: Effect of Different Distance Definitions on Nearest Neighbors

2. **Containment Query(Reg).** Find all the data objects that lie within the query region Reg. Formally, return all the objects X for which CONTAINS(Reg, X) returns true. For example, find all the aeroplanes that are currently in New York.

3. **Exact Match Query(Q):** Find all objects that have same spatial extent as the object Q has. More specifically, find all the spatial objects X for which EQUALS(Q, X) returns true. Consider, for an example, a query may be issued to find a house that has same structure as my house.

4. **Nearest Neighbor Query(Q).** Find the closest object to the query object Q. The problem may also be extended to k nearest neighbor (kNN) query, that is to find k closest objects to the query object. More specifically, a k nearest neighbor query is to find a set of objects R that contains k objects such that for all $A \in R$ and for all other objects B, $DISTANCE(Q, A) \leq DISTANCE(Q, B)$. Note that the answer may change depending on the distance definition. Fig. 1.3 illustrates this. The nearest neighbor of A is B if the distance between two points is defined as the minimum distance between them. If the distance is defined as the maximum distance between the two objects, C becomes the nearest neighbor of A.

One example of such queries is to find 10 closest gas stations to my current location. Also note that other metrics than Euclidean distance may also be used. In above example, the road distance may be used to find the nearest neighbors.

5. **Spatial Join Query(R, S).** Let R and S be two sets of objects, a spatial join query is to find object pairs from these sets so that they satisfy some specified join predicate. The examples of possible join predicates may include INTERSECTS, DISJOINT, DISTANCE and CONTAINS operations defined above. An example of such queries is to find all hotel-cinema pairs so that distance between them is less that 1 km.

1.3 Contributions

With the availability of wireless networks and inexpensive mobile devices, continuously monitoring of spatial queries over moving data objects has become a necessity to many recent location-based applications. Consequently, a number of techniques [SR01, TP02, TPS02, ZZP$^+$03, ISS03, XMA$^+$04, YPK05, HXL05, MHP05] have been developed to efficiently process continuous k nearest neighbor (kNN)queries and its variants. Below we summarize the contributions of this thesis to the continuous monitoring of spatial queries[1].

1. For continuous monitoring of spatial queries, the interest of database community has been shifted towards using grid based index because R-tree cannot handle object updates very efficiently. We design CircularTrip and ArcTrip, two novel grid access methods that traverse grid cells according to their proximity to the query point.

2. We propose CircularTrip-based algorithm [CYL07] to continuously monitor kNN queries and our extensive experiments demonstrate that our approach is 2 to 4 times faster than CPM [MHP05] which is currently the best known algorithm for

[1]This work was partially supported by the Australian Research Council (ARC) Discovery Grant (DP0666428)

continuous kNN queries. Moreover, we prove that our CircularTrip based algorithm accesses the minimum number of cells during the continuous monitoring of kNN queries.

3. Our CircularTrip based algorithm is not only time efficient but it is also space efficient. Its space usage is 50% to 85% of CPM's space requirements.

4. We present the techniques based on CircularTrip and ArcTrip to continuously monitor other variants of kNN queries like constrained nearest neighbor queries, farthest neighbor queries and $(k + m)$ nearest neighbor queries. Our proposed techniques can easily be extended to answer any complex query that is hybrid of above mentioned queries. For example, our algorithms can be used to continuously monitor $(k+m)$ farthest neighbors in a constrained region where the spatial conditions that constrain the region can be changed by the user at any time. Moreover, our proposed algorithms preserve the properties that all the algorithms have low memory requirements and the number of cells accessed for each type of query is minimum.

5. We give a detailed survey of previous work on spatial k nearest neighbor queries and its popular variants.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

- A detailed survey of previous work on nearest neighbor queries and its variants has been presented in Chapter 2.

- In Chapter 3, we present our grid access methods named CircularTrip and ArcTrip.

- A CircularTrip based algorithm for continuous monitoring of kNN queries is presented in Chapter 4. Comparison with previous best known approach (CPM) is also presented in Chapter 4.

- In Chapter 5, we briefly describe the techniques to continuously monitor other variants of kNN queries by using CircularTrip and ArcTrip.
- Chapter 6 concludes our research presented in this thesis.

Chapter 2

Related Work

In this chapter, we overview the work related to the nearest neighbor queries. In Section 2.1, we survey the related work on kNN queries. We describe the popular variants of kNN queries in Section 2.2.

2.1 k Nearest Neighbors Queries

Given a set of data points P and a query point q, and an integer $k > 0$, the k nearest neighbors (kNN) query is to find a result set kNN that consists of k data points such that for any $p \in (P - kNN)$ and any $p' \in kNN$, $dist(p', q) \leq dist(p, q)$.

Nearest neighbor queries have received intensive attention in spatial database community in the past decade. A kNN query has various applications in spatial databases. Consider, for example, a set of points in two dimensions representing cities. A kNN query may be issued to find "what are the k closest cities from a point p". In this section, we present a brief overview of the work done on kNN queries. Since kNN queries are widely studied by the researchers in different problem settings, we categorise the related work according to the characteristics of problem settings as follows.

- **Snapshot kNN queries:** Find $k > 0$ objects from a static dataset that are closest to a static query point according to Euclidean distance between them.

- **Continuous kNN queries:** Continuously report $k > 0$ objects, over a time period T, from a continuously moving dataset that are closest to a continuously moving query point. These type of queries are most related with our work.

- **kNN queries in spatial networks:** Find $k > 0$ objects that are closest to a query point q according to their network distances from q. An example is "find k nearest hotels from my current location". Clearly, the underlying network is road network and the network distance between two points may be different from the minimum Euclidean distance between them. The problem can be extended to continuous monitoring of kNNs on networks.

- **Approximate kNN queries:** Sometimes the data is very large or can be accessed only once (e.g. in data streams). In such cases, finding exact kNNs is not efficient so there are many solutions proposed to find the approximate k nearest neighbors of queries with some guaranteed relative error. Such queries are called approximate-kNN queries.

- **Distributed processing of kNN queries:** Different from previous problem settings where a central server is solely responsible for computations, this problem is to find kNNs assuming that the moving data points also have the computation and storage capabilities. Such algorithms utilize the computation and storage capabilities of mobile objects and aim to reduce computation load on the server and/or communication cost between mobile objects and the server.

In Section 2.1.1, we overview the related work on snapshot kNN queries. Previous work on continuous monitoring of kNN queries is given in Section 2.1.2. Section 2.1.3 surveys the work on kNN queries over spatial networks. Approximate kNN queries are presented in Section 2.1.4. Section 2.1.5 reviews the work on distributed processing of kNN queries.

2.1.1 Snapshot kNN Queries

As described earlier, a snapshot kNN query is to find the k nearest neighbors from a static dataset. Snapshot queries are extensively studied by spatial database community [Hen94, RKV95, KSF^{+}96, PM97, KS97, SK98, HS99, CG99]. There have been many spatial indexes proposed for multidimensional points but the R tree [Gut84] and its variants (especially R*-tree [BKSS90]) are the most popular ones. The most widely used approaches to answer kNN queries [RKV95, HS99] use branch-and-bound algorithms on R-tree while maintaining a list of k potential nearest neighbors in a priority queue.

Besides the branch-and-bound algorithms, there have also been attempts [KSF^{+}96] to use range queries to solve kNN queries. Basic idea is to first find a region that guarantees to contain all kNNs and then to use a range query to retrieve the potential kNNs. The improvements to this algorithms were proposed in [CG99] by improving the region estimation. Seidl *et al.* [SK98] improved this algorithm by providing a better search technique of the kNNs in the region.

As described earlier, most popular branch-and-bound algorithms [RKV95, HS99] traverse the R-tree to retrieve k nearest neighbors. R-tree is traversed in depth-first manner in [RKV95] and in best-first search manner in [HS99]. Fig. 2.1 shows spatial objects and their corresponding R-tree. Below, we briefly describe these two popular R-tree traversal methods with the help of Fig. 2.1.

Depth-First Search (DFS)

Starting from the root node, DFS visits the child nodes in a certain order. More specifically, the order is determined by the *mindist* of each node from the query point q, where *mindist* is the minimum distance between q and the node's minimum bounding rectangle (MBR). When a leaf node is reached, the objects are retrieved and the list of nearest neighbor candidates is updated. Let $q.dist_k$ be the distance of k^{th} nearest neighbor from q, clearly the algorithm does not need to visit any node N and its child entries if $mindist(N,q) \geq q.dist_k$.

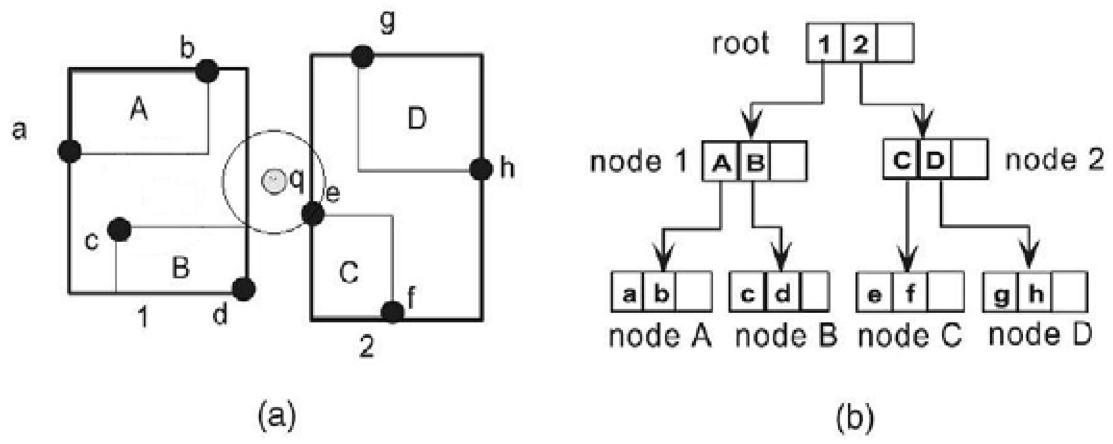


Figure 2.1: An Example of a NN Query on R-tree Index

Fig. 2.1(a) illustrates an example of a nearest neighbor query: $a, b, ..., h$ are the spatial objects and q is the query point. Fig. 2.1(b) shows the corresponding R-tree, including the root, intermediate nodes 1 and 2 and leaf nodes A, B, C and D. The algorithm first visits node 1 because $mindist(1, q) < mindist(2, q)$. Then, DFS visits node B (because $mindist(B, q) < mindist(A, q)$). It computes the *mindist* of the object c and d in node B and inserts d in the nearest neighbor candidate list $q.kNN$. DFS next visits the node A but does not find a better candidate than d. Next node visited is 2. Since $mindist(C, q) < mindist(D, q)$, DFS visits the node C and finds object e that is closer to q than d, so it replaces e with d in $q.kNN$. The next node to be visited is D, DFS prunes it because $mindist(D, q) > mindist(e, q)$. There is no other node to be visited so DFS reports e as the final answer. Table 2.1 shows the nodes in the order they were processed by DFS algorithm.

Node in Process	**candidate set** ($q.kNN$)	$q.dist_k$
1	ϕ	∞
B	d	$mindist(d, q)$
A	d	$mindist(d, q)$
2	d	$mindist(d, q)$
C	e	$mindist(e, q)$

Table 2.1: Depth-First Search

Best-First Search (BFS)

BFS uses a priority queue to store the entries to be explored during the search. The entries in priority queue are sorted by their *mindist* from q. For each popped entry e, all its child are pushed into the queue if e is a node. If e is a leaf, all the objects it contains are considered and the candidate set $q.kNN$ and $q.dist_k$ is updated, accordingly. The algorithm stops as soon as the next popped entry has *mindist* greater than $q.dist_k$

Consider the same example of Fig. 2.1. BFS inserts nodes 1 and 2 in queue. The node 1 is popped first and its child entries A and B are inserted in the priority queue PQ which becomes PQ ={2,B,A}. Since 2 has the smallest *mindist*, it is popped next and its child entries C and D are inserted in the queue (PQ ={C,B,A,D}). The next entry is C, the object e is inserted in candidate set $q.kNN$ and the $q.dist_k$ is updated to $mindist(e,q)$. Algorithm stops because the next entry in queue B has *mindist* greater than $mindist(e,q)$. Table 2.2 shows the nodes in the order they were processed by DFS algorithm.

Priority Queue	**candidate set ($q.kNN$)**	$q.dist_k$
$1,2$	ϕ	∞
$2,B,A$	ϕ	∞
C,B,A,D	ϕ	∞
B,A,D	e	$mindist(e,q)$

Table 2.2: Best-First Search

2.1.2 Continuous kNN Queries

Different from a *snapshot* kNN query, continuous kNN queries are issued once and run continuously to generate results in real-time along with the updates of the underlying datasets. Since mobile devices are now able to detect their postions, the continuous monitoring of the spatial queries have become one of the most interesting problems in spatio-temporal databases. Objects send their location updates and the results of queries are updated accordingly.

The problem of continuously monitoring kNN of moving objects has been investigated largely in past few years. From the modeling and query language perspectives, continuous monitoring of kNN queries was first addressed in [SWCD97]. Kollios *et al.* [KGT99] were first to consider answering kNN queries for moving objects in 1D space. Song *et al.* [SR01] presented the first algorithm for computation of a moving kNN query q over static data set in high dimensional space. Server sends to client a number $m > k$ of neighbors when a kNN query is processed. The k nearest neighbors at a new location q' will be among the m objects of the first query q provided that the distance between q and q' is within a range determined by k and m. For the same setting where queries are moving and data set is static, Zhang *et al.* [ZZP$^+$03] proposed a solution that returns the NN of query q along with its Voronoi cell, where Voronoi cell is the area in which the NN of query will remain same.

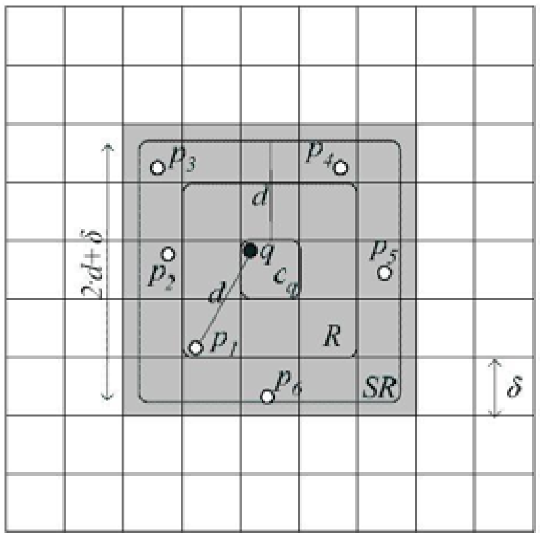
Assuming that the velocity of linearly moving data objects is known, time-parameterized queries [TP02] report the current NN set and its validity period and the next change of the result that will occur at the end of the validity period. Such queries use TPR-tree [SJLL00] or its variants such as TPR*-tree [TPS03](extension of R*-tree by augmenting the indexed objects and the MBRs with the velocity vectors). In [BJKS02], authors propose a solution by depth-first traversal of TPR-tree. Their approach returns all NN sets up to a future time assuming that the velocity vector of the query will not change up to that time. Raptopoulou *et al.* [RPM03] proposed a method by combining the merits of [TP02] and [BJKS02] which significantly reduces the I/O and CPU costs.

All of the above methods and [ISS03] work only when the future trajectories of objects are known at query time by either some linear function or some recursive function [TFPL04]. As discussed in Section 2.1.1, most widely used approach to answer conventional snapshot kNN queries is using R-tree index. R-tree index cannot be used for efficiently monitoring of kNN queries because it cannot support frequent updates of data objects. Though a bottom-up update approach of updating R-tree was introduced in [LHJ$^+$03], recently the interest of database community has been shifted towards using grid-based index to answer continuous kNN queries over highly dynamic data sets. Since grid-based index is simple, it can be easily updated with the updates of data point location updates. Kalashnikov *et al.* [KPH04] used the in-memory grid structure to monitor continuous range queries. The most related and recent works [MHP05, XMA05, YPK05] all employ grid-based index. The space is partitioned into a regular grid of cells $\delta \times \delta$ and each algorithm computes the results of each query every T time units. These algorithms do not make any assumption on the motion pattern of data points and/or query points. Since these are the most recent, most related and the best known algorithms, we briefly describe them below.

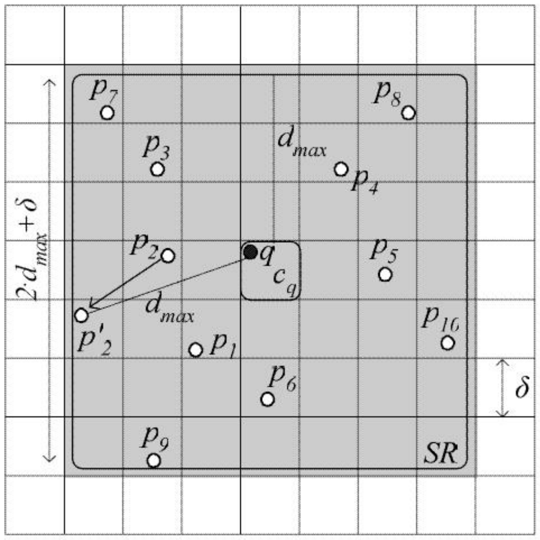
YPK-CNN

Yu *et al.* [YPK05] proposed a method for continuous monitoring of exact kNN queries hereafter referred as YPK-CNN. The new query is evaluated in two steps. First the algorithm searches for k objects around q in iteratively enlarged square R centered at cell c^q, the resident cell of query. Fig. 2.2 shows an example of 1-NN query where in first step p_1 is found with distance d from q. To guarantee the correctness of results, YPK-CNN processes all the cells intersected by a square SR centered at c^q with side length $2.d + \delta$ and updates the answer if necessary. Fig. 2.2(a) shows that the actual answer p_2 is found when all the cells within square of each side $2.d + \delta$ (shaded cells) are visited. YPK-CNN processes all objects from p_1 to p_6 in the running example.

Upon receiving the object updates, YPK-CNN uses the previous result of query to update the new result. Let d_{max} be the maximum distance of the object $p \in q.kNN$ from q that is one of the previous nearest neighbors and has now moved farthest from q, YPK-CNN updates the results as follows. Algorithm visits all the cells within square SR with side length $2.d_{max} + \delta$. Fig. 2.2(b) shows that object p_2 issues an update to location p'_2 so d_{max} is set to $dist(p'_2, q)$. YPK-CNN visits all objects (p_1 to p_{10}) in the shaded cells of the figure and identifies p_1 as the new NN. When a query point changes its location, it is deleted and then handled as new query (i.e., its answer is computed from scratch).



(a) initial NN computation



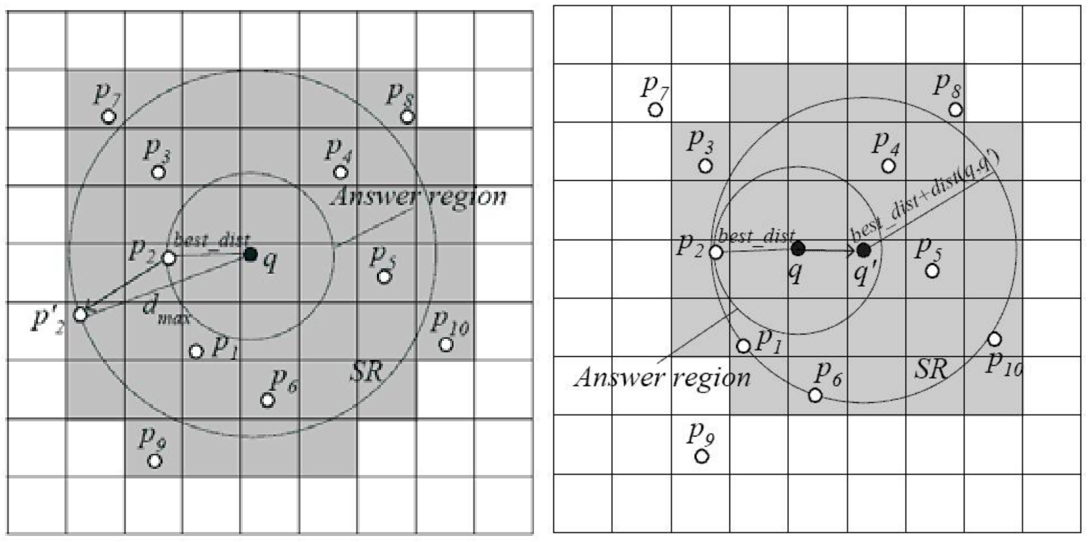
(b) Update Handling

Figure 2.2: YPK-CNN

SEA-CNN

Xiong *et al.* [XMA05] proposed SEA-CNN which focuses exclusively on monitoring the NN changes, without including a module for the evaluation of initial results of a newly registered query. Objects are indexed with grid in secondary memory. The *answer region* of a query q is defined as the circle centred at q with radius *best_dist* where *best_dist* is the distance of k^{th} NN from q. The cells intersecting this circle store book-keeping information to indicate the fact that they are affected by q.

Updates are handled by expanding the circle to radius d_{max}, the maximum distance of all outgoing objects. In Fig. 2.3(a), the object p_2 issues an update and d_{max} is set as $dist(p'_2,q)$. Search region SR is set as the circle with radius r=d_{max} and all the cells intersecting this circle (shaded cells in the figure) are visited and result is updated. Finally, if the query q moves to a new location q' as shown in Fig. 2.3(b), the SEA-CNN sets $r = best\_dist + dist(q, q')$ and computes the new kNN set of q by processing all the cells that lie in the circle centered at q' with radius r. The processed cells are shown shaded.



(a) p_2 issues update (b) q issues update

Figure 2.3: SEA-CNN

CPM

The best known algorithm for continuous kNN queries is CPM proposed by Mouratidis *et al.* [MHP05]. It visits minimal set of cells during initial computation. CPM organizes the cells into conceptual rectangles based on their proximity to q. Each rectangle is defined by a *direction* and a *level number*. The direction is **U, D, L** or **R** (for up, down, left, and right), and the level number indicates the number of rectangles between the rectangle and q. Fig. 2.4(a) illustrates the conceptual partitioning of space for q around the cell $c_{(4,4)}$.

CPM initializes a heap by inserting c^q, the resident cell of q with key 0, and the zero level rectangle for each direction DIR, with key $mindist(DIR_0, q)$ which is the minimum possible distance between the query and the rectangle DIR_0. Then it starts deheaping the entries. If the deheaped entry is a cell, it examines all the objects inside it and updates the $q.kNN$ accordingly, the list of k nearest neighbors. If the deheaped entry is a rectangle DIR_{lvl}, CPM enheaps (1) each cell $c \in DIR_{lvl}$ with key $mindist(c, q)$ and (2) the next level rectangle DIR_{lvl+1} with key $mindist(DIR_{lvl+1}, q)$. The algorithm terminates when the next entry in the heap (either a cell or a rectangle) has key greater than or equal to the *best_dist* (the distance of k^{th} Nearest Neighbor from q). It can be easily seen that the algorithm only processes the cells that intersect with the circle centered at q with radius equal to *best_dist*. This is the minimal set of cells to visit in order to guarantee the correctness. In Fig. 2.4(a), the shaded cells in light color are those that are visited by the algorithm.

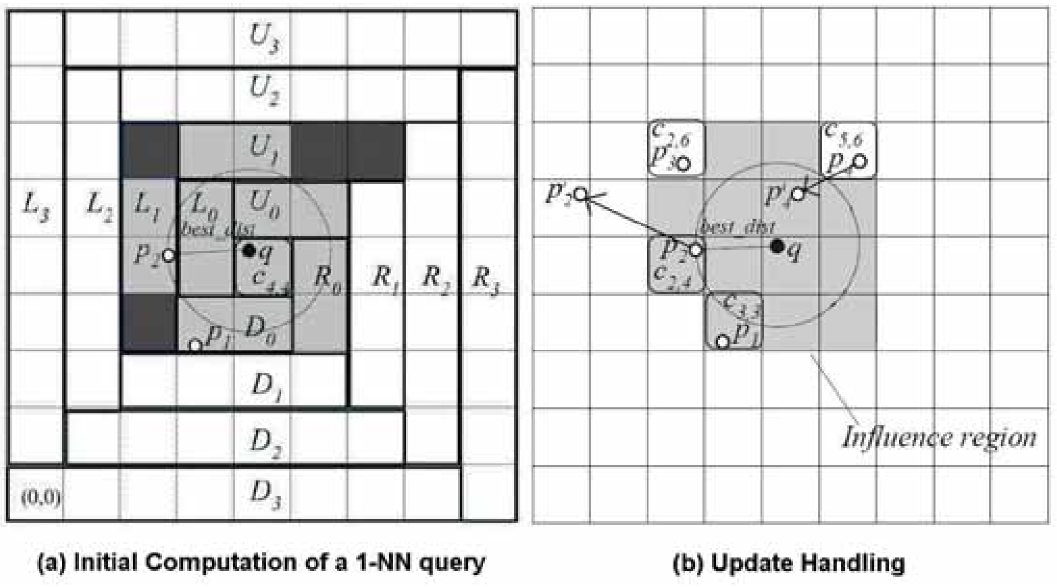


Figure 2.4: CPM

In order to handle the updates efficiently, CPM needs to store (1) *visit list* that contains the cells visited during NN search (in ascending order with respect to their *mindist* from q) and (2) search heap H that contains the entries e (cells and rectangles) which were enheaped but were not deheaped (for which $mindist(e, q) \geq best\_dist$). The visit list, in our running example of Fig. 2.4(a), contains the light shaded cells. The search heap H contains the shaded cells in dark color plus the four boundary rectangles U_2, R_1, D_1 and L_2. The *influence region* of query q are the cells c such that $mindist(c, q) \leq$ *best_dist*. In other words, this is the set of cells that intersect the circle centered at q with radius *best_dist*. Only updates affecting these cells can influence the NN result.

CPM handles the updates as follows: If an object p enters into the influence region (*incoming update*), CPM inserts it in $q.kNN$ and deletes the k^{th} nearest neighbor. If an object p leaves the influence region (*outgoing update*), CPM removes p from $q.kNN$ and calls the re-computation module to compute nearest neighbor set. NN re-computation algorithm first starts visiting the cells in *visit list* of q and then it continues with the entries of search heap H in the same way as it does in NN initial computation. The algorithm stops when the next entry in *visit list* or search heap H has *mindist* greater than or equal to the distance of k^{th} NN found so far. Note that NN re-computation algorithm is same as initial computation algorithm except that it re-uses the previous information. i.e., It does not need to calculate the *mindist* of cells in *visit list* and it avoids enheap and deheap operations.

Consider the example of Fig. 2.4(b) where an object p_4 moves to p'_4 (to illustrate clearly, we ignore the update issued by p_2). Since this is an incoming update, the object p_4 is added to the $q.kNN$ and the old k^{th}NN p_2 is deleted from $q.kNN$. Now take the example where p_2 moves to p'_2 (now we ignore the update of p_4). Since it is an outgoing update, the object p'_2 is removed from the $q.kNN$ and the NN re-computation module is invoked.

To handle multiple updates, CPM counts the number of incoming and outgoing updates for each query q. If incoming updates are greater than outgoing updates, the new nearest neighbors can be found among the incoming objects. Otherwise, re-computation module is called. If query q changes its position, it is deleted from the query list and added as a new query at new location. Then, it is computed from scratch.

2.1.3 kNN Queries in Spatial Networks

Due to the importance of spatial networks in real-life applications, many approaches have been presented to answer spatial queries on spatial networks. In practice, objects can usually only move on a pre-defined set of trajectories as specified by underlying network (road, railway, river etc.). Thus, the importance measure is the network distance, i.e. the length of the shortest trajectory connecting two objects, rather than their Euclidean distance. Since the most popular application of such queries is finding k closest objects on a road-network, such queries are also called kNN queries on road-network. Continuous monitoring of kNN queries on road-networks has also been studied extensively.

Jensen *et al.* [JKPT03] formalize the problem of kNN search in road-networks and present a system prototype for such queries. Papadias *et al.* [PZMT03] describe a framework that integrates network and Euclidean information, and answers various spatial queries including kNN and range queries. They index the data objects with an R-tree and utilize connectivity and location information to guide the search. Their approach is based on generating a search region for the query point that expands from the query. The advantages of their approach are: 1) it offers a method that finds the exact distance in networks, and 2) the architecture can support other spatial queries like closest pair and e-join queries. Since the number of links and nodes that are retrieved and examined are inversely proportional to cardinality ratio of entities and number of nodes in the network, the main disadvantage of this approach is a dramatic degradation in performance when the above cardinality ratio is less than 10% which is the usual case for real world scenarios (real data sets representing the road network and different type on entities in the State of California show that this cardinality ratio is usually between 0.04% and 3% [KS04a]).

Kolahdouzan *et al.* [KS04a] proposed a new approach called VN^3 for NN queries in spatial network databases which precalculates the Network Voronoi Polygons (NVPs) and some network distances. VN^3 is based on the properties of the network Voronoi diagrams. The intuition is that the NVPs can directly be used to find the first nearest neighbor of a query q. Subsequently, the adjacency information of NVPs can be utilized to provide a candidate set for other nearest neighbors of q. Though their approach outperforms INE [PZMT03], but for higher densities of objects it suffers from computational overhead of precalculating NVPs.

A faster but approximate kNN results are retrieved in [KS04b] by an embedding technique that approximates the network distance with computationally simple functions. Given a user-specified trajectory, a path kNN query retrieves the kNNs at any point in the trajectory. Assume that we know the $k + 1$ NNs at some point o in the query trajectory, and that their network distance from o is $dist_i$, for $i = 1, ..., k+1$. Let δ be the smallest difference between the distances of two consecutive NNs, i.e; $\delta = min_{1 \leq i \leq k}\{dist_{i+1} - dist_i\}$. Shahabi *et al.*[SKS02] propose a path NN algorithm based on the observation that any point in the query trajectory within distance $\delta/2$ from o has exactly same NN set as o.

Cho and Chung [CC05] solve the problem by retrieving kNN sets of all network nodes in the query path and uniting them with the data objects falling in the path. It can be observed that the resulting set contains kNNs of any point in the query trajectory. Hu *et al.* [HLX06] proposed a new approach that simplifies the network by replacing the graph topology with a set of interconnected tree-based structures called SPIE's. For each SPIE, they build an index and the algorithm computes kNNs by searching SPIE on a predetermined path to avoid costly network expansions.

Mouratidis *et al.* [MYPM06] address the continuous monitoring of kNN queries in road-networks with arbitrary object and query moving patterns. Their approach also supports the case when the edge weights of underlying network may change at any time. They present two approaches. The first one maintains the query results by processing only updates that may invalidate the current NN sets. The second method follows the shared execution paradigm. More specifically, it groups together the queries that fall in the path between two consecutive intersections in the networks, and produces the results by monitoring the NN sets of these intersections.

2.1.4 Approximate kNN Queries

Most of the work on kNN queries assumes that the data is disk-resident and can be scanned multiple times. However, such work is not suitable for processing data streams that typically require a one-pass algorithms as the data is not stored on disk and it is too large to fit in main memory. So, queries on data streams do not have access to the entire data set and query answers are typically approximate.

Bern *et al.* [Ber93] presented a quadtree based algorithm to answer approximate NN queries. The approximate answer guarantees that the distance from the query point to the returned point is at most $12.d^{1/2}$ times the distance from the query point to any other point in the data structure where d is the dimensionality. However, their approach is limited to find only a single NN and they did not discuss the extension of their approach to answer k nearest neighbors. Moreover, the relative error depends on the dimensionality d.

Problem of $(1+\epsilon)$-approximate kNN queries was studied in [AM93]. Let $q.dist_k$ be the distance of k^{th} actual nearest neighbor of q from q, an approximate $(1+\epsilon)k$NN query is to find a set of objects $(1+\epsilon)k$NN that contains k objects so that $q_{(1+\epsilon)}.dist_k \leq q.dist_k \times (1+\epsilon)$ where $q_{(1+\epsilon)}.dist_k$ is the distance of k^{th} object in $(1+\epsilon)k$NN from q. The relative error ϵ is a user specified constant. The algorithm proposed in [AM93] required exponential time in d and linear space. Follow-up studies [AMN$^+$98, KOR98, LCGMW02] improved its time/space requirements.

Koudas *et al.* [KOTZ04] were the first to propose a solution to approximate kNN queries with absolute error bounds. Given a dataset P, a query point q and a user specified value e, they find a set ekNN that contains k points in P such that for any $p' \in ekNN$ there exists a point $p \in kNN$ (the actual kNN set) such that $dist(p', q) \leq (dist(p, q) + e)$. Their technique is called DISC (aDaptive Indexing on Stream by space-filling Curve). They partition the space into a grid so that the maximum distance between any two points in any cell is at most e. To avoid storing all the points arriving in system, they keep only $K \geq k$ objects in each cell c and discard the rest. They prove that the exact kNN search on these stored objects corresponds to a valid ekNN answer over the original data set. DISC indexes the data points with a B-tree that uses a space-filling curve mechanism to facilitate fast updates and query processing. The index can be adjusted to either 1) optimize memory utilization to answer ekNN queries under certain accuracy requirements or 2) achieve the best possible accuracy under a given memory constraint. DISC can process both snapshot and continuous ekNN queries.

2.1.5 Distributed Processing of kNN Queries

In previous problem settings, a central server is solely responsible for answering spatial queries assuming that the clients have no or very few computation and storage capability. Clients send queries to server, the server computes the results and sends answer back to the clients. However, database community has also investigated the option of utilizing the computation and storage capabilities of clients. One of the main design principles is to develop efficient mechanisms that utilize the computational power at mobile objects, leading to significant savings in terms of server load and/or messaging cost when compared to solutions relying on central processing of location information at the server

Prabhakar *et al.* [PXK$^+$02] proposed a method to continuously monitor range queries, called *Q-index*. They monitor the static range queries over moving data objects. The main idea is to send a *safe region* to each moving object with the guarantee that the result of any query will not change unless the object leaves its safe region. More specifically, each object p is assigned a circular or rectangular region such that p needs to issue a location update only if it leaves its region otherwise it does not affect the results of any query. They build R-tree index on queries instead of moving objects due to intensive updates of moving objects. Each object probes the index to find the query it influences. Cai *et al.* [CHC04] propose MQM, another range monitoring method. The workspace is divided into rectangular sub-domains and each object is aware only of the range queries intersecting its *resident region* which may consist of many sub-domains. Each object reports location update only when it crosses the boundary of any of these queries. When an object leaves its resident region, the server computes a new resident region and sends it to the object. The server computes the resident region by using a binary partitioning tree which maintains for each sub-division of the workspace the queries that intersect it. The number of sub-domains that form an object's resident region depends on how many queries it can store and process.

In both of the above approaches, the messaging cost of mobile objects is reduced because mobile objects report location updates only when the result of some query is

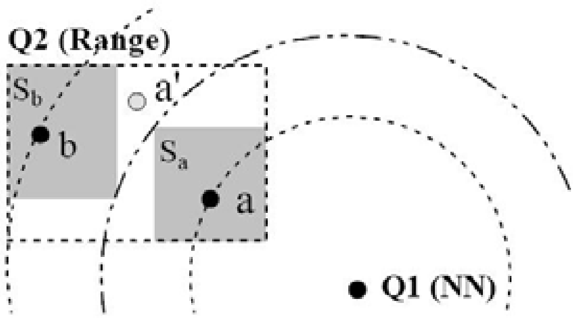


Figure 2.5: Example of Safe Regions

expected to be changed. Gediket *al.* [GL04] propose an approach called *MobiEyes* that significantly reduces the server load and the messaging cost. The idea of *MobiEyes* is similar to that proposed in [CHC04] but *MobiEyes* can answer the moving range queries which latter fails to answer. More specifically, the work space is partitioned using a grid and for each query a *monitoring region* is maintained which is the union of grid cells the query can potentially intersect. Any object that falls in monitoring region of a query receives the information about query position and velocity. The objects report to the server when they enter or leave the predicted query region. Note that this way the objects monitor their spatial relationships with queries locally. Whenever a query moves out of its current cell or changes its velocity, it notifies the server of this change and server relays such position change information to the appropriate subset of objects through broadcasts.

Hu *et al.* [HXL05] presented first distributed approach for monitoring continuous nearest neighbor queries. They also employ the idea of *safe regions* which is a rectangular region that guarantees that the results of any query will not change as long as the object is inside its safe region. Fig. 2.5 shows two queries where Q_1 is a nearest neighbor query and Q_2 is a range query. The safe regions of two objects a and b are S_a and S_b, respectively, shown shaded in the figure. The current results of both queries will not change if both objects remain in their safe regions.

Initial results of query are calculated using best-first search (BFS) on R-tree. The moving objects issue location updates to the server only when they leave their safe regions. Server receives the location updates and incrementally finds the new results of affected queries and computes the new safe regions for the appropriate objects. Consider the example of Fig. 2.5, where a moves out of S_a to a new location a', the result of Q_1 becomes undecided as either of the two objects could be the nearest neighbor. To resolve the ambiguity, the server needs to know the exact location of b so server asks object b to send its current location and computes the new result of Q_1. Safe regions of both the objects a and b are also needed to be updated.

Mouratidis *et al.* [MPBT05] propose a threshold-based algorithm that aims to minimize the communication cost between server and the data objects. Their proposed method can be used for multiple queries with unknown motion pattern and for any distance definition. The basic idea is, for each kNN query q the objects are categorized in two sets. The objects that belong to the result are called *inner* objects and the remaining objects are called *outer* objects. For each object p, a threshold is calculated so that as long as the object is within the range defined by the threshold it does not change the result of the query. For clear illustration, consider the example of Fig. 2.6 where the three nearest neighbors p_1, p_2 and p_3 are inner objects and p_4, p_5 and p_6 are outer objects. The thresholds values are t_1, t_2 and t_3 which define the range for each object such that if its distance from q lies within the range the result is guaranteed to be unchanged. Each threshold is set to the middle of distances of two consecutive neighbors of the query. More specifically, the distance range for p_1 is $[0, t_1)$, for p_2 is $[t_1, t_2)$, for p_3 is $[t_2, t_3)$ and for remaining objects p_4,p_5 and p_6 is $[t_3, \infty)$.

When an object moves out of the range, it reports to the server and the server updates the results and may assign new threshold. Consider the example when p_1 moves to p_1', the order of the two NNs may be changed. Server asks for the location of p_2 and if it is closer to q than p_1, the results are changed accordingly. Otherwise, there is no result change and only the new threshold t_1 is computed and objects p_1 and p_2 are informed of this new threshold. Clearly, all other objects are not required to be involved during this

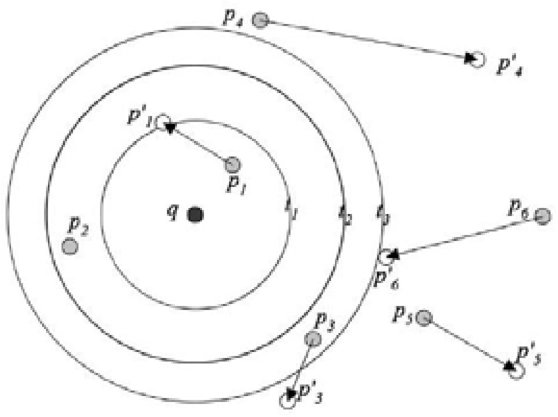


Figure 2.6: Threshold-Based Algorithm for Monitoring a 3-NN Query

update.

Now consider the case where p_3 moves to p'_3 and issues a location update. Any of the outer objects may become the new 3^{rd} nearest neighbor depending on their current locations. Server broadcasts a request to all the outer objects to send their latest locations and updates p_6 as the new 3^{rd}-NN. A new threshold t_3 is computed based on the new positions of 3^{rd}-NN (p_6) and 4^{th}-NN (p_3) and server broadcasts this threshold value to all outer objects and p_6.

2.2 Variants of Nearest Neighbor Queries

In this section, we briefly describe the popular variants of nearest neighbor queries. More specifically, we present the related work on reverse nearest neighbor queries in Section 2.2.1. Constrained nearest neighbor queries are discussed in Section 2.2.2. In Section 2.2.3, we review the problem of aggregate nearest neighbor queries. Finally, Section 2.2.4 presents all-nearest neighbor queries.

2.2.1 Reverse Nearest Neighbor Queries

The most popular variant of nearest neighbor query is reverse nearest neighbor queries that focuses on the inverse relation among points. A reverse nearest neighbor (RNN) query q is to find all the objects for which q is their nearest neighbor. A reverse nearest neighbor query is formally defined below.

Definition: Given a set of objects P and a query object q, a reverse nearest neighbor query is to find a set of objects RNN so that for any object $p \in P$ and $r \in RNN$, $dist(r, q) \leq dist(r, p)$.

RNN set of a query q may be empty or may have one or more elements. Korn *et al.* [KM00] defined the RNN queries and provided a large number of applications. For example, a two-dimensional RNN query may ask the set of customers affected by the opening of a new store outlet location in order to inform the relevant customers. This query can also be used to identify the location which maximizes the number of potential customers. Consider another example, an RNN query may be issued to find the stores outlets that are affected by opening a new store outlet at some specific location. Note that in first example, there are two different sets (stores and customers) involved in RNN query whereas in second example there is only one set (stores). Korn *et al.* defined two variants of RNN queries. A *bichromatic* query (the first example) is to find the reverse nearest neighbors where the underlying data set consists of two different types of objects. A *monochramtic* RNN query (the second exampe) is to find the reverse nearest neighbors where the data set contains only one type of objects. The problem of reverse nearest neighbors is extensively studied in past few years [SAA00, YL01, SRAA01, MZ02, LNY03, SFT03, TPL04, XZKD05, YPMT05, XZ06, TYM06, ABK+06]. Below we briefly describe the most popular and general algorithms only.

Korn and Muthukrishnan [KM00] answer RNN query by pre-calculating a circle of each object p such that the nearest neighbor of p lies on the perimeter of the circle as shown in Fig. 2.7. The MBR of all these circles are indexed by an R-tree called RNN-tree. The problem of RNN query is reduced to a point location query on RNN-tree that returns all the circle containing q. For examples, circle of a and e contain q so both are the reverse nearest neighbors of q. Yang *et al.* [YL01] improved their method by RdNN-tree. Similar to RNN-tree, a leaf node of the RdNN tree contains the circles of

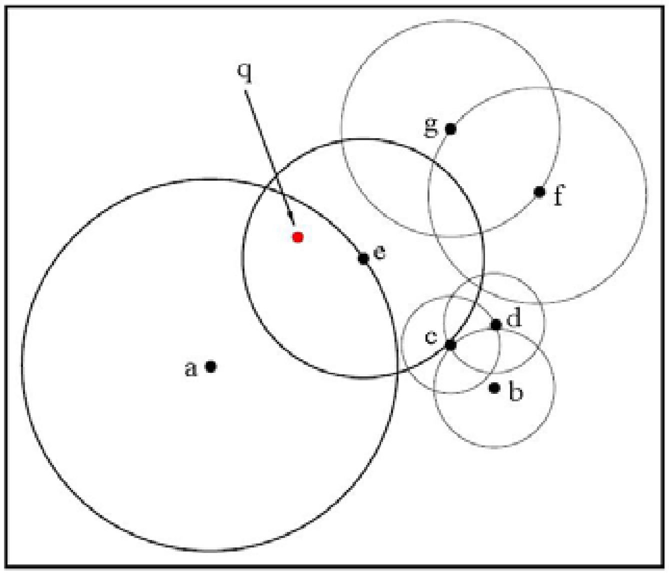


Figure 2.7: The Objects a and e are the Reverse Nearest Neighbors of q

data points. The intermediate nodes contain the minimum bounding rectangles (MBR) of underlying points along with the maximum distance from every point in the sub-tree to its nearest neighbor. The problem with above mentioned techniques is that they rely on pre-computation and cannot deal with efficient updates. In order to alleviate this problem, Lin *et. al* [LNY03] introduced a method for bulk insertion in the RdNN-tree.

Stanoi *et al.* [SAA00] eliminate the need for pre-computing the nearest neighbors of all data points by utilizing an interesting property of reverse nearest neighbor queries. Their approach is called SAA hereafter. SAA utilizes the fact that for any two-dimensional RNN query q, the number of reverse nearest neighbors cannot exceed six. The algorithm divides the space around q in six regions of equal size S_0 to S_5 as shown in Fig. 2.8. They observe that for a nearest neighbor object o_i of q in region S_i; either o_i is the RNN of q or there is no RNN in S_i. For example, o_1 is the nearest neighbor of q in region S_0 but it is not the RNN because o_0 lies closer to it than q. Consequently, there is no RNN of q in S_0 and the algorithm does not need to consider other objects in this region. Based on this interesting property, SAA answers RNN queries in two steps. In first step, for each of the six regions a nearest neighbor is found. All these nearest neighbors form a

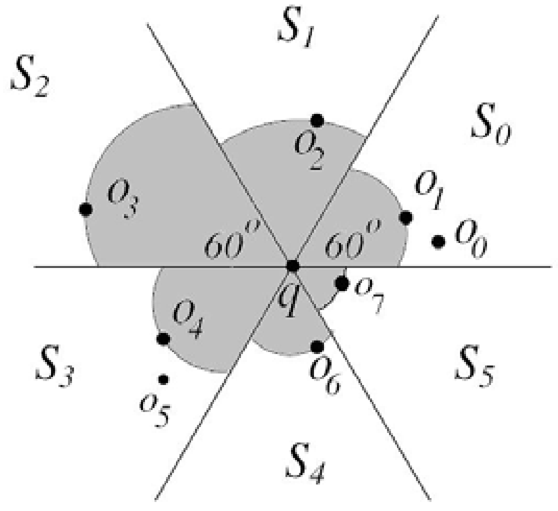
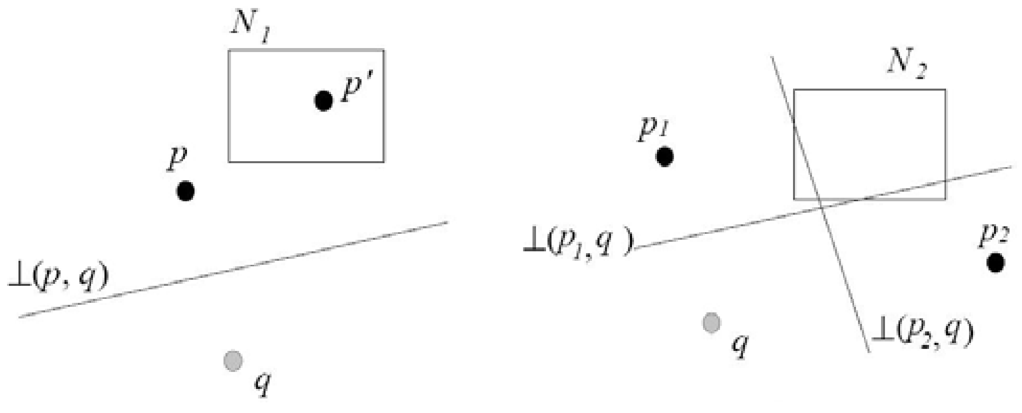


Figure 2.8: Illustration of SAA

candidate list. In second step, a NN query is issued for each object in the candidate list and the algorithm discards the objects from candidate list for which q is not the nearest neighbor.

SAA suffers with the curse of dimensionality. The number of regions to be searched for candidate objects increases exponentially with the dimensionality. Singh *et al.* [SFT03] propose a solution that performs better than SAA in high-dimensional space. Given a system parameter K, their proposed approach first computes K nearest neighbors of q by using conventional R-tree approach. These K objects form the *candidate list*. In the second step, they eliminate the objects from candidate list that are closer to some other object in candidate list than q. Finally, for each object p in candidate list, a *boolean range* query with range $dist(p, q)$ is issued. A boolean range query is different from a conventional range query in the sense that it terminates as soon as the first object is found. Clearly, the objects for which the boolean range query does not return any object are the RNNs of q. However, depending on the value of K, their approach may miss some reverse nearest neighbor that was not retrieved by K nearest neighbor search in first step of the algorithm.

Tao *et al.* [TPL04] utilize the idea of perpendicular bisector to reduce the search space. Let AB be a line segment joining two points A and B, a perpendicular bisector $\perp (A, B)$ of the two points A and B is perpendicular to the line AB and passes through the midpoint M of AB. Consider the example of Fig. 2.9(a) where a perpendicular bisector $\perp (p, q)$ between two points p and q is shown. This perpendicular bisector divides the space into two half planes PL_q and PL_p where PL_q contains q and PL_p contains p. It can be noted that there cannot be any RNN in the plane PL_p other than p (p is closer to all points in PL_p than q). Based on this property, any MBR can be pruned that falls completely in PL_p. The MBR N_1 is pruned in the example of Fig. 2.9(a). Even if some MBR does not fall completely in a half plane PL_{p_1} of an object p_1, it can be pruned if it lies entirely in the union of PL_{p_1} and PL_{p_2} where PL_{p_2} is the half-plane of another object p_2. Fig. 2.9(b) shows the pruning of MBR N_2 because it falls in the union of half-planes PL_{p_1} and PL_{p_2}. Their approach can also be extended to answer RkNN queries that is to find all objects for which q is one of their k nearest neighbors.



(a) Pruning with one point (b) Pruning with two points

Figure 2.9: Half-Plane Pruning

To the best of our knowledge, the only algorithm that can answer continuous reverse nearest neighbor queries is proposed by Xia *et al.* [XZ06]. They utilize the idea presented in [SAA00]. Consider the example of Fig. 2.10 where a continuous reverse nearest neighbor (CRNN) query q is shown along with six regions S_0 to S_5. The monitoring region of q consists of two parts: *pie-regions* and *circ-regions*. A pie-region in a partition S_i is

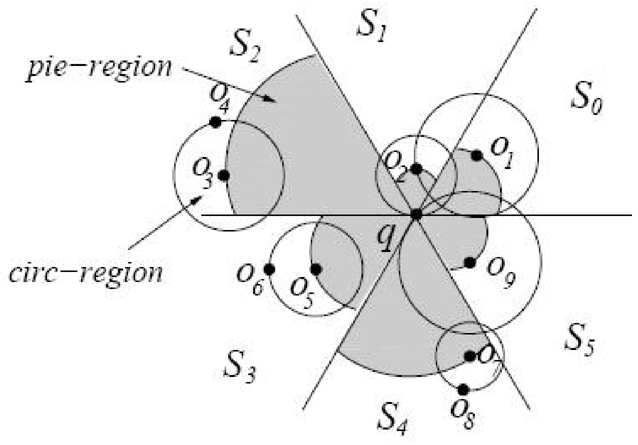


Figure 2.10: The Monitoring Region of a Continuous RNN Query

a pie centered at q having the nearest neighbor of q in S_i on the perimeter. Let p_i be the nearest neighbor of q in a region S_i, the circ-region is a circle centered at p_i having q or an object nearer to p_i than q on its perimeter. The shadowed areas are the six pie-regions and the circles are six circ-regions. Intuitively, the pie-regions monitor the updates that will change the *candidate list* and may possibly change the results. The updates in circ-regions can make a previous result a false positive or can make a previous false positive an RNN. In other words, any update outside the monitoring region (the union of pie-region and circ-regions) cannot affect the results of query and can be ignored.

2.2.2 Constrained Nearest Neighbor Queries

Constrained nearest neighbor queries [FSAA01] can be defined as nearest neighbor queries that are constrained to a specified region. This type of queries is targeted towards the users who are particularly interested in nearest neighbors in a region bounded by certain spatial conditions, rather than in searching for nearest neighbors in the entire data space.

There are many applications of this variant of nearest neighbor queries. For example, a user may wish to find 5 nearest gas stations in San Francisco from his location. Clearly, the constrained region is the map of San Francisco. Similarly, a user may issue a query to find 10 nearest neighbors from his location to the North. In this example, the constrained

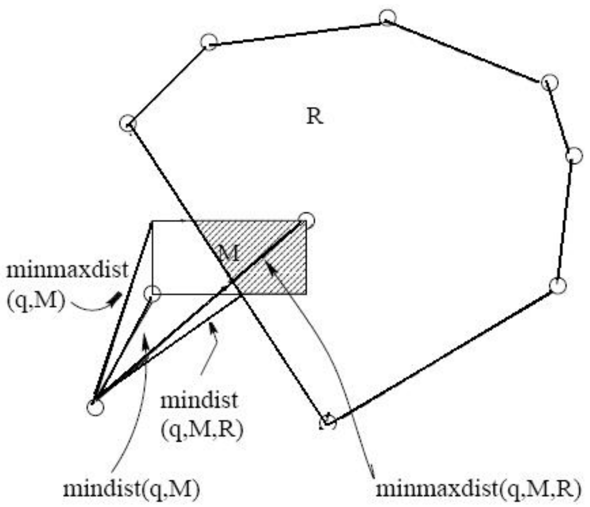


Figure 2.11: Illustration of *mindist*(q, M, R) and *minmaxdist*(q, M, R)

region is the sub-space that lies to the North of the user. Recall, six constrained nearest neighbor queries are issued to find reverse nearest neighbor of a query [SAA00].

Ferhatosmanoglu *et al.* [FSAA01] propose a solution based on the best-first traversal of R-tree. Their proposed pruning criteria makes their technique optimal with respect to the number of I/O accesses. The proximity comparisons in [RKV95], based on the Euclidean distance metric use the notion of *mindist*(q, M) and *minmaxdist*(q, M) where q is a query point and M is a minimum bounding rectangle (MBR). More specifically, *mindist*(q, M) is the shortest distance from q to the given MBR M and *minmaxdist*(q, M) is the minimum distance from q to the furthest point on the closest face of the MBR M. Fig. 2.11 illustrates these definitions. These metrics ensure that for a minimum bounding rectangle M, there is at least one data point within the distance range of $[mindist(q, M), minmaxdist(q, M)]$. Consider a 1-NN query, an MBR M' and its child entries can be pruned if $mindist(q, M') > minmaxdist(q, M)$ because by the definition it is known that M contains at least one object that is closer than all objects in M'. Note that M' and its child can also be pruned if there exists a point p such that $dist(p, q) < mindist(q, M')$.

Ferhatosmanoglu *et al.* notice that the above definitions of *mindist* and *minmaxdist*

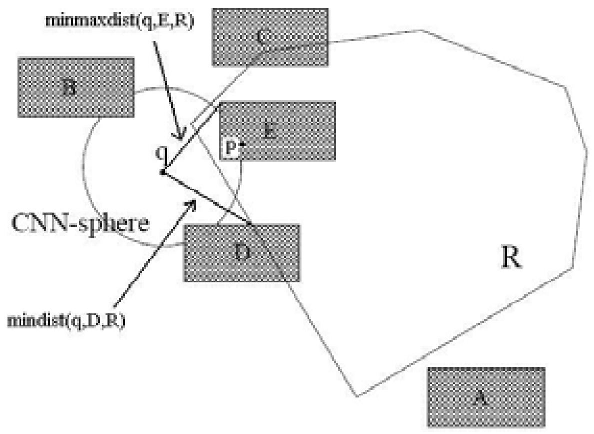


Figure 2.12: Computation of a Constrained Nearest Neighbor Query

used in [RKV95] cannot yield correct results so they re-define these distance metrics as follows: Let R be the constrained region, M be a minimum bounding rectangle (MBR) and q be a query point, $mindist(q, M, R)$ is the minimum distance between q and the part of MBR that lies in R. Consider the example of Fig. 2.11, $mindist(q, M, R)$ is the minimum distance of q to the shaded area of M. Note that $mindist(q, M)$ can never be greater than $mindist(q, M, R)$ ($mindist(q, M) \leq mindist(q, M, R)$) hence this definition provides a tighter pruning condition. $minmaxdist(q, M, R)$ is the distance between q and the furthest point on the closest face of M that is fully contained in R. In Fig.2.11, there is only one edge of M that completely lies inside R, so $minmaxdist(q, M, R)$ is the distance from q to the farthest point on this edge as shown. It can be seen that with the previous definition of $minmaxdist(q, M)$, it cannot be guaranteed that there is at least one object that lies in the range $[mindist(q, M), minmaxdist(q, M)$ because that point may lie outside the region R. According to the new definition, the condition that the edge should be fully contained in R guarantees that there is at least one object that lies in R within the range $[mindist(q, M), minmaxdist(q, M, R)]$.

The authors prove that their approach is optimal with respect to the number of I/O accesses. Consider the example of Fig. 2.12, there algorithm accesses only the MBR E. The MBRs A and B are pruned because they do not lie in the constrained region R. MBR

C is pruned because $mindist(q, C, R) > minmaxdist(q, E, R)$. Since $mindist(q, B, R) < mindist(q, D, R)$, B is accessed first and an object p is found. The MBR D is pruned because $mindist(q, D, R) > dist(p, q)$. The algorithm terminates and reports p as the answer.

To the best of our knowledge, there does not exist any previous work on continuous monitoring of constrained nearest neighbor queries. In chapter 5, we propose an efficient solution for the problem of continuous monitoring of constrained nearest neighbor queries.

2.2.3 Aggregate Nearest Neighbor Queries

Aggregate nearest neighbor (ANN) queries are also called group nearest neighbor queries (GNN) so we will use these names interchangeably hereafter. An aggregate nearest neighbor query retrieves the data points with the smallest sum of distances to all query points in a query group Q. The formal definition is given below.

Definition: Given a set of points P and a group of queries Q, an aggregate k nearest neighbor query is to find a set of points R that contains k objects such that for any $p \in (P - R)$ and any $p' \in R$, $dist(p', Q) \leq dist(p, Q)$ where for any object o, $dist(o, Q) = \sum_{\forall q_i \in Q} dist(o, q_i)$.

As an example, consider few friends want to meet at some city so that the overall distance travelled is minimized. In this case the friends form the query group and the cities are data points. An aggregate nearest neighbor query will return the city so that the total distance travelled by all of them is minimum. Aggregate nearest neighbor queries can also be used for clustering and outlier detection. Papadias *et al.* [PSTM04] give another interesting application of aggregate nearest neighbor queries. The operability and speed of very large circuit depends on the relative distance between various components, the aggregate nearest neighbor queries can be issued to find the abnormalities and guide relocation of components [NO97].

Papadias *et al.* [PSTM04] present three algorithms to answer aggregate nearest neighbor queries called *multiple query method* (MQM), *single point method* (SPM) and *mini-*

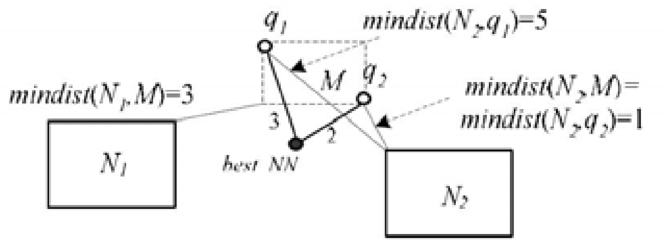


Figure 2.13: Node N_1 can be pruned

mum bounding method (MBM). All these algorithms traverse R-tree to answer aggregate nearest neighbor (ANN) queries. Since their experiments show that MBM outperforms other two algorithms, below we briefly describe MBM.

To prune the search space, MBM uses a minimum bounding rectangle M that contains all query points in Q. The algorithm traverses R-tree and prunes the nodes that cannot contain any candidate point. They prune the nodes based on two observations. Let M be the MBR of Q, and *best_dist* be the distance of the best ANN found so far. A node N can be pruned if $mindist(N, M) \geq best\_dist/n$ where $mindist(N, M)$ is the minimum distance between nodes N and M and n is the cardinality of Q. Consider the example of Fig. 2.13, where the node N_1 can be pruned because $mindist(N_1, M) = 3 >$ $(best\_dist/2 = 2.5)$. This pruning can also be applied on data points. Whenever a data point p is retrieved, first its minimum distance from the MBR M is calculated and the data point p can be pruned if $mindist(p, Q) > best\_dist/n$.

Second observation provides a tighter pruning bound. Any node N can be pruned if $\sum_{\forall q_i \in Q} mindist(N, q_i) \geq best\_dist$ where $mindist(N, q_i)$ is the minimum distance between the node N and a query point q_i. In the example of Fig. 2.13, the node N_2 is pruned because $mindist(N_2, q_1) + mindist(N_2, q_2) = 6 > best\_dist$, .

Li *et al.* [LLHH05] propose two ellipse-based pruning methods for the aggregate nearest neighbor queries. Yiu *et al.* [YMP05] solve the problem of aggregate nearest neighbors on road-network. In all above mentioned queries, the aim is to minimize the total sum of the distances to the query points from a point p. Mouratidis *et al.* [MHP05] introduce two other variants of aggregate nearest neighbor queries. In first variant the aim is to find a point $p \in P$ with the smallest distance from any query point in Q. We call such queries as minimum aggregate nearest neighbor queries and define them as follows.

Minimum ANN Queries: Given a set of data points P and a group of query points Q, a minimum ANN query is to find a point $p \in P$ such that for any other point $p' \in P$, $mindist(p, Q) \leq mindist(p', Q)$ where for any object o, $mindist(o, Q) = min_{\forall q_i \in Q} dist(o, q_i)$.

Consider, for an example, that few friends need to buy one laptop and there are many shops in the city that sells their required laptop. They decide that the person who has some shop nearest to his home will go to buy the laptop. They issue a minimum ANN query. Their houses can be considered as data points and the shops can be treated as query points.

Another variant of ANN is to find the object p that has lowest maximum distance from query points in Q. We name such queries MinMax ANN queries and define as follows.

MinMax ANN Queries: Given a set of data points P and a group of query points Q, a MinMax ANN query is to find a point $p \in P$ such that for any other point $p' \in P$, $maxdist(p, Q) \leq maxdist(p', Q)$ where for any object o, $maxdist(o, Q) = max_{\forall q_i \in Q} dist(o, q_i)$.

Consider as an example that there are many proposed locations to establish a fire brigade unit in a small town. The mayor wants to establish the fire brigade unit at a place so that if any of the houses in town catches fire, the distance to that house from fire brigade unit is minimum compared to the other proposed location. Clearly, the target is to find a location so that the distance of farthest house from the fire brigade unit is minimized. A MinMax ANN query will serve the purpose. The houses in this case are query points and proposed locations are data points.

Mouratidis *et al.* also propose the solution of these variants of aggregate nearest neighbor queries that is very similar to CPM algorithm discussed in Section 2.1.2.

2.2.4 All-Nearest Neighbor Queries

Given two object sets A and B, an all-nearest neighbor (all-NN) query is to find, for each object in A, a nearest neighbor from B. The formal definition is given below.

Definition: Given two object sets A and B, an all-NN query(A, B) is to find for each object $a_i \in A$ an object $b_j \in B$ such that for all other objects $b_k \in B$, $dist(b_j, a_i) \leq dist(b_k, a_i)$.

The result of an all-NN(A, B) query consists of $\mid A \mid$ object pairs $< a_i, b_i >$, where $\mid A \mid$ is the cardinality of set A and b_i is the nearest neighbor of a_i in set B. Note that all-NN queries are not commutative. More specifically, all-NN$(A, B) \neq$ all-NN(B, A).

These queries are common in several applications. For example, a query may be issued to find the nearest hospital for each housing society of a city. This may be an important query for urban planning. Such queries may also be issued for resource allocation problems. Several clustering and outlier detection algorithms also use all-NN queries for efficiency. Similar to aggregate nearest neighbor queries, all-NN queries can also be used to detect abnormalities in very large circuits [NO97].

Few spatial join methods have been proposed that can be used to answer all-NN queries. Brinkhoff *et al.* [BKS93] proposed a join method where both data sets are assumed to be indexed by R-tree. The algorithm returns the objects from the sets A and B that intersect each other. The algorithm traverses both trees and prunes any entry pair $< E_A, E_B >$, if the nodes E_A and E_B do not overlap. The intuition is that these nodes cannot contain any object a_i that intersects some object b_j. Clearly, the problem of spatial join is different from finding all-NN because even if two nodes do not overlap they can still contain two points a_i and b_j so that b_j is the NN of a_i. However, this spatial join technique can be slightly changed to answer closest-pair problem which has been used to answer all-NN queries [HS98, CMTV00]. A closest-pair (CP) problem is to find a pair $< a_i, b_j >$ from two sets A and B so that there is no other pair $< a_x, b_y >$ such that $dist(a_x, b_y) < dist(a_i, b_j)$. The above spatial join method can be used to answer CP queries. However, the difference is that sometimes the nodes that do not overlap will have to be visited. The algorithm traverses the two R-tree indexes and recursively visits the entry pairs $< E_A, E_B >$ that has minimum distance among all other entries. Bohm *et al.* [BK04] propose a k-nearest neighbor join method which searches k NNs from set B for each object in A. If $k = 1$, the problem is same as All-NN query. GORDER [Hon04] is another method to process k nearest neighbor joins.

A straight forward approach to answer an all-NN query is to perfrom one NN query on data set B for each object in A. In [BKSS90], the authors propose several improvements to reduce CPU and I/O costs. Zhang *et al.* [ZPMT04] proposed two approaches for the case when data set B is indexed. *Multiple nearest neighbor* (MNN) method is similar to an index-nested-loop join operation. For each object in A, MNN applies a NN query on the R-tree of data set B. The order in which the objects from A are processed is important because if two objects that are near to each other are processed one after the other, a large percentage of the pages of the R-tree will be available in LRU memory for the processing of the second object. To achieve a better order, the data set A is sorted by some space filling curve [Bia69] which reduces the I/O cost for MNN. However, CPU cost of MNN is still very high. To reduce the CPU cost, they propose *batched nearest neighbor* (BNN) method. BNN splits the points in A into n disjoint groups. Then, the R-tree of B is traversed once only for each group which significantly reduces the number of distance computations hence a lower CPU cost. They also presented a hash-based method, but their experiment proved that BNN outperforms hash-based methods in most of the experiments.

The latest work on all-NN queries is proposed by Chen and Patel [CP07]. They improve the pruning bounds presented by Corral *et al.* [CMTV04]. Let M and N be two minimum bounding rectangles (MBR) for data points in A and B, respectively. Corral *et al.* define $MINMINDIST(M, N)$ as the minimum distance between the two MBRs. The intuition is that this is the minimum possible distance between any two points present in the MBRs (one point in M and one in N). They define $MAXMAXDIST(M, N)$ as the maximum distance between two MBRs M and N. These distance metrics are

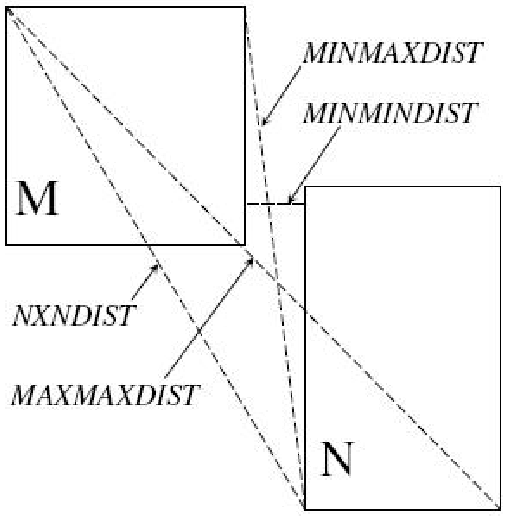


Figure 2.14: Different Pruning Metrics for All-NN Queries

illustrated in Fig. 2.14. The pruning condition is that an MBR N' and its child entries can be pruned if $MAXMAXDIST(M, N) < MINMINDIST(M, N')$ because for every point $a_i \in M$, there always exists a point in N that lies closer to a_i than all points in N'.

Chen and Patel note that the pruning metric $MAXMAXDIST$ is unnecessarily conservative and can be improved. They define another pruning metric $MINMAXMINDIST$ which is termed $NXNDIST$ in short and is shown in Fig. 2.14. Intuitively, $NXNDIST(M, N)$ is the maximum possible distance of any object $a_i \in M$ to its nearest neighbor $b_i \in N$. An MBR N' and its child entries can be pruned if $NXNDIST(M, N) < MINMINDIST(M, N')$ because by the definitions of MBR and $NXNDIST$, for every point $a_i \in M$, there always exists a point in N that lies closer to a_i then all points in N'. We omit the mathematical definition and calculation of $NXNDIST$ due to its complexity and refer interested readers to [CP07].

Chapter 3

Grid Access Methods

In this chapter, we present our grid access methods CircularTrip and ArcTrip. In Section 3.1, we describe the motivation behind these access methods. We present CircularTrip in Section 3.2 and ArcTrip is described in Section 3.3. Finally, Section 3.4 summarizes the chapter.

3.1 Motivation

A grid based index evenly partitions the space into cells. The extent of each cell c on each dimension is δ. $c[i, j]$ indicates the cell at column i and row j and the lower-left corner cell of the grid is $c[0, 0]$. Clearly, point p with location coordinates $(p.x, p.y)$ falls into the cell $c[\lfloor p.x/\delta \rfloor, \lfloor p.y/\delta \rfloor]$. An example of grid based index is shown in Figure 3.1. Compared with other complicated spatial indexes (e.g., R-tree) grid based index is simple and can be maintained efficiently in the dynamic environment. The order in which cells of grid are accessed is important for efficient monitoring of the NN queries hence a need is to develop an access method which minimizes the number of cells accessed. The minimum number of cells that are needed to be accessed to answer any kNN query q are only the cells that lie or intersect the circle with center q and radius $q.dist_k$ (the shaded cells in Fig. 3.2), where $q.dist_k$ is the distance of k^{th}NN from q. Any algorithm that visits a cell that lies outside the circle, visits a cell which cannot have any result object. In contrast, any algorithm that does not access any cell that lies or intersects the circle may potentially miss a valid result object. To the best of our knowledge, CPM [MHP05] provides the only method that tries to minimize the number of cells accessed. CPM minimizes the number of cells accessed in initial computation of any kNN query by employing a conceptual partitioning of grid into rectangles and using a heap as discussed in details in Section 2.1.2. However, CPM accesses unnecessary cells during handling of the updates. Moreover, it also needs to store some data structure for each registered query q.

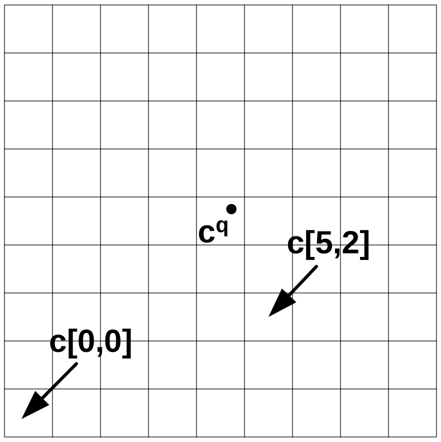


Figure 3.1: Grid Index

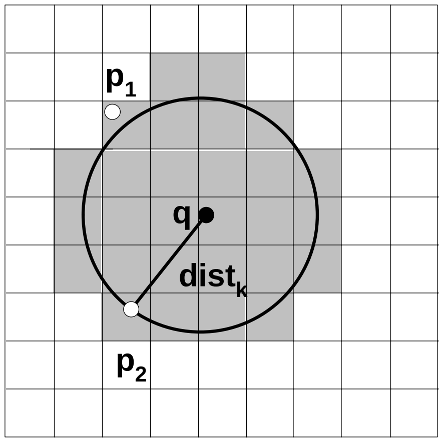


Figure 3.2: Minimal Set of Cells

CPM partitions the space into conceptual rectangles and the rectangles do not reflect spatial proximity to any point. We note that circle centered at query point represents the better proximity and we use this intuition to develop our grid access methods. We design CircularTrip and ArcTrip that access the cells according to their proximity to q and minimize the number of cells accessed not only in initial computation of results but also during the updates handling of nearest neighbor queries.

CircularTrip: Given a radius r, CircularTrip returns all the cells that intersect the circle of radius r with center at q.

ArcTrip: Given a radius r and an angle range $\langle\theta_1, \theta_2\rangle$, ArcTrip returns all the cells that intersect the circle of radius r with center at q and lie between angle range $\langle\theta_1, \theta_2\rangle$.

In next chapters, we will show how these access methods can be used to continuously monitor kNN queries and its variants (i.e., constrained NN queries). We show that by using our access method, these queries can be answered while maintaining the following properties

- The number of cells accessed to continuously monitor kNN query and its variants is minimum. i.e; it does not visit any unnecessary cell even during the handling of updates.
- As opposed to CPM, there is no need to store any additional data structure for each query.

We also prove that our proposed access methods computes the $mindist(c, q)$ exactly the same number of times as the number of cells it returns. Below in Table 3.1 we summarize the math notations used throughout this thesis.

Notation	**Definition**
p	a moving data point
q	the query point
$p.x$, $p.y$, $q.x$, $q.y$	the x-axis (y-axis) coordinate of p (q)
c, $c[i, j]$	a cell (at column i and row j)
c^q	the cell containing q
$c.i$, $c.j$	column (row) value of cell c
δ	cell side length
$dist(p, q)$	the distance between p and q
$q.kNN$	the kNN of q
$q.dist_k$	the distance between the k-th NN and q
$mindist(c, q)$, $maxdist(c, q)$	the minimum (maximum) distance between c and q

Table 3.1: Math Notations

3.2 CircularTrip

In this section, we present and analyze our access method named CircularTrip. As described earlier, given a radius r and a query location q, CircularTrip returns a set of cells C so that all the cells in it intersect the circle of radius r with center at q. Formally, $C_r = \{\forall c \mid mindist(c, q) < r \leq maxdist(c, q)\}$. Figure 3.3 shows an example where CircularTrip returns all the shaded cells. In Section 3.2.1, for the sake of simplicity we present CircularTrip for 2 dimensional grid structure then we will discuss its extension to higher dimensions in Section 3.2.2.

3.2.1 CircularTrip on 2D Grid

To collect a round of cells, CircularTrip starts from the cell c_{start} that is intersected by the given circle and lies left to c^q. CircularTrip starts checking the cells along the circle in direction D_{cur} which is initially set as **up**. In order to make a clock-wise trip, the D_{cur} needs to be changed from **up** to **right** to **down** to **left** to **up**. The direction D_{next} always holds the next direction of D_{cur}.

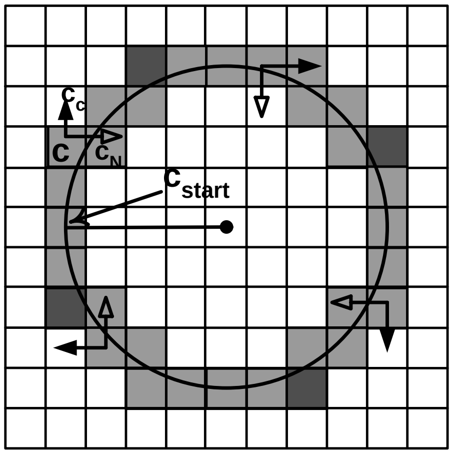


Figure 3.3: CircularTrip

Without loss of generality, consider cell c intersected by the circle which locates in the upper-left quadrant as shown in Fig. 3.3. The key fact is that the next cell intersected by the circle (i.e., the cell in which the arc is connected to one in c) is the adjacent cell either above c (i.e., the cell c_c that is next cell in direction D_{cur}) or right to c (i.e., the c_N that is next cell in direction D_{next}). This is because the outgoing circle crosses either the upper boundary or the right boundary of c. These two adjacent cells, c_c and c_N, are called *candidate adjacent cells* of c. To collect the next cell intersected by the circle, CircularTrip only needs to examine one of the candidate adjacent cells (i.e; check its $mindist(c, q)$ with the given radius r).

Algorithm 1 presents the implementation of CircularTrip algorithm. Though any cell can be selected as c_{start} but in our implementation the left most cell of the round (as shown in Fig. 3.3) is selected. CircularTrip examines the cells clockwise along the given circle until c_{start} is encountered again. When the currently being examined cell c is at the maximum level in D_{cur}, the directions to find its candidate adjacent cells are updated accordingly (i.e., line $10-12$). The cells when the directions D_{cur} and D_{next} are changed are shown dark shaded in Fig. 3.3. Moreover, for some of the cells, D_{cur} and the D_{next} are shown with solid and hollow arrows, respectively.

Algorithm 1 CircularTrip(G, q, r)

```
Input:  G: the grid index; q: query point; r: the radius;
Output: all the cells which intersect the circle with center q and radius r;
1:  $C := \varnothing$; $c := c_{start} := c[i,j]$ ($i := \lfloor (q.x - r)/\delta \rfloor$, $j := \lfloor q.y/\delta \rfloor$);
2:  $D_{cur} := Up$; /* clockwise fashion: $Up \rightarrow Right \rightarrow Down \rightarrow Left \rightarrow Up$ */
3:  $D_{next} := Right$ /* always holds next direction to $D_{cur}$ */
4:  repeat
5:    insert $c$ into $C$;
6:    $c'$ := the adjacent cell to $c$ in $D_{cur}$ direction;
7:    if ($mindist(c', q) \geq r$) /* if $c'$ does not intersect the circle */ then
8:      $c'$ := the adjacent cell to $c$ in $D_{next}$ direction;
9:    $c := c'$;
10:   if (($D_{cur} = (up$ or $down)$ and $c.j = \lfloor (q.y \pm r)/\delta \rfloor$) OR ($D_{cur} = (right$ or $left)$ and
      $c.i = \lfloor (q.x \pm r)/\delta \rfloor$)) then
11:     $D_{cur} := D_{next}$;
12:     $D_{next}$ := the next direction to $D_{cur}$;
13: until $c = c_{start}$
14: return $C$;
```

Lemma 3.2.1 *The total cost of CircularTrip to collect a round C of cells is to compute*

$mindist(c, q)$ of $|C|$ cells, where $|C|$ is the number of cells in round C.

Proof Let c be the current cell being examined and c_c and c_N be the cells next to c in direction D_{cur} and D_{next}, respectively. In order to prove Lemma 3.2.1, it is sufficient to prove that algorithm needs to compute *mindist* of only one of c_c or c_N from q in order to confirm the next intersected cell.

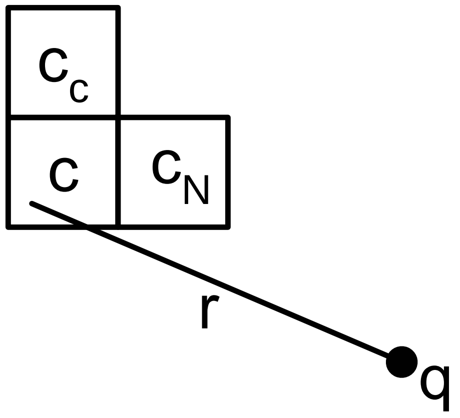


Figure 3.4: Next Intersected Cell is Either c_c or c_N

As can be seen from Fig. 3.4, selection of D_{cur} and D_{next} is made so that following two conditions hold.

$$mindist(c_N, q) < mindist(c, q) < mindist(c_c, q) \tag{3.1}$$

$$maxdist(c_N, q) < maxdist(c, q) < maxdist(c_c, q) \tag{3.2}$$

As cell c intersects the circle, it satisfies that

$$mindist(c, q) < r \leq maxdist(c, q) \tag{3.3}$$

From equations 3.1, 3.2 and 3.3, it can be concluded that

$$mindist(c_N, q) < r < maxdist(c_c, q) \tag{3.4}$$

CircularTrip computes $mindist(c_c, q)$, and if $mindist(c_c, q) < r$, then c_c intersects the circle as can be inferred in conjunction with equation 3.4 that $mindist(c_c, q) < r < maxdist(c_c, q)$.

If $mindist(c_c, q) \geq r$, then it infers that $maxdist(c_N, q) \geq r$ because $mindist(c_c, q) = maxdist(c_N, q)$. So CircularTrip can confirm c_N as next intersected cell without computing its distance from q because for cell c_N in conjunction with equation 3.4, it is confirmed that $mindist(c_N, q) < r \leq maxdist(c_N, q)$. ∎

3.2.2 Extension to Higher Dimensions

CircularTrip can be immediately extended to a higher dimensional space. Take 3D space for example. The only difference is now CircularTrip (Algorithm 1) returns the cells intersected by the sphere of radius r centered at q. Given a query $q(x, y, z)$, CircularTrip is invoked on the plane $z = \lfloor p.z/\delta \rfloor$ and its results is C. Then, for each pair of cells in C intersected by the planes parallel to the plane $y = \lfloor p.y/\delta \rfloor$, call CircularTrip with one of them as c_{start} and half of their distance as radius to collect all the satisfied cells. It is immediately verified that properties of CircularTrip in 2D are all retained.

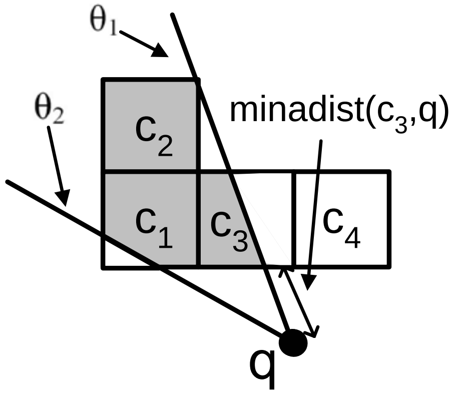


Figure 3.5: Minimum Angular Distance

3.3 ArcTrip

First we define $minadist(q, c, \langle\theta_1, \theta_2\rangle)$ that is minimum angular distance of a cell c from q and then we will give formal definition of ArcTrip.

Minimum angular distance ($minadist(q, c, \langle\theta_1, \theta_2\rangle)$): The minimum angular distance between a cell c and the query point q is Euclidean distance from q to the nearest boundary of the portion of c that lies between angle range $\langle\theta_1, \theta_2\rangle$. Fig. 3.5 clarifies the definition where *minadist* values of c_1 and c_2 are same as their *mindist* whereas *minadist* of c_3 is the minimum distance of the shaded area of c_3 from q. Any cell that completely lies outside the angle range has $minadist = \infty$. i.e; $minadist(q, c_4, \langle\theta_1, \theta_2\rangle) = \infty$.

Now we define ArcTrip$(q, r, \langle\theta_1, \theta_2\rangle)$: given a radius r and an angle range $\langle\theta_1, \theta_2\rangle$, ArcTrip returns every cell c that intersects the circle of radius r with center at q and has $minadist(q, c, \langle\theta_1, \theta_2\rangle)$ smaller than r. Consider the example of Fig. 3.6, ArcTrip(q,r,θ_1,θ_2) returns all the shaded cells.

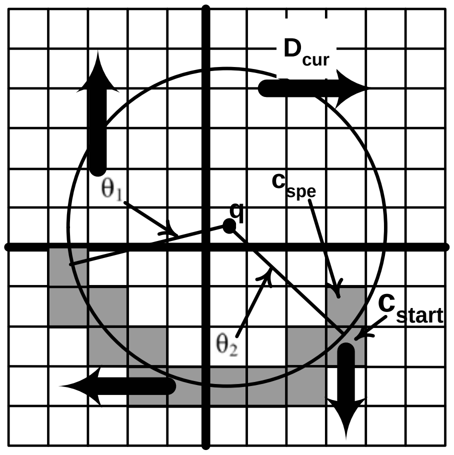


Figure 3.6: ArcTrip

ArcTrip algorithm is similar to CircularTrip. The algorithm starts with the cell c_{start} that intersects the circle and lies at angle θ_2. The cell c_{start} is shown in Fig. 3.6. Mathematically, $c_{start} = c[i,j]$ where $i = \lfloor(q.x + r.Cos\theta_2)/\delta\rfloor$ and $j = \lfloor q.y + r.Sin\theta_2/\delta\rfloor$); The direction D_{cur} is selected according to the quadrant in which c_{start} lies. Fig. 3.6, shows D_{cur} for different quadrants with jumbo arrows. In the example of Fig. 3.6, D_{cur} is set as **down**. The algorithm continues exactly same as CircularTrip and terminates as soon as the next cell that intersects the circle lies outside the range $\langle\theta_1, \theta_2\rangle$.

Note from Fig. 3.6, if we start from c_{start} as mentioned above, we may miss a cell c_{spe} that should have been returned by ArcTrip. We prove next that this special cell c_{spe} is always adjacent to c_{start} in direction opposite to D_{cur}. To address this issue, we always

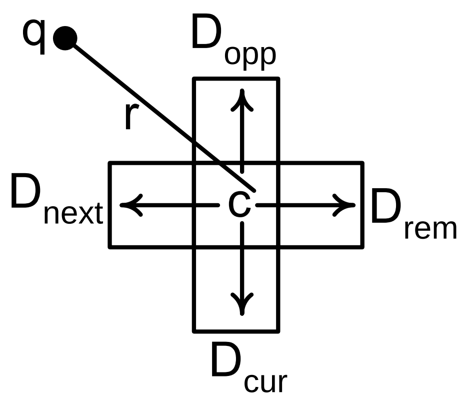


Figure 3.7: The Special Cell c_{spe} is Always the Adjacent Cell of c in D_{opp}

first check whether the cell c_{spe} should be inserted in C or not depending on whether it lies in angle range $\langle\theta_1, \theta_2\rangle$ or not. If it lies in the angle range we include it in result otherwise we ignore it (i.e; line 4 – 5 of Algorithm 2).

Lemma 3.3.1 *Let c_{start} be the starting cell of ArcTrip as defined earlier and D_{opp} be the opposite direction of D_{cur}. The only cell ArcTrip can miss is c_{spe} that is always the adjacent cell to c_{start} in direction D_{opp}.*

Proof It can be immediately verified that the c_{spe} is one of the four cells adjacent to c_{start} (in Fig. 3.7 c_{start} is shown as c). The cells in direction D_{cur} and D_{next} will be checked during execution of ArcTrip and will be inserted if intersect the circle. So c_{spe} can only be the cell either in direction D_{opp} or the remaining direction D_{rem}.

Let c_{opp} and c_{rem} be the adjacent cells to c_{start} in direction D_{opp} and D_{rem}, respectively. It can be immediately verified that $minadist(q, c_{rem}, \langle\theta_1, \theta_2\rangle)$ is always greater than r (otherwise it would have been the starting cell c_{start}), so ArcTrip does not need to check it. We are left only with c_{opp} and hence it is c_{spe}. ▮

Algorithm 2 presents the implementation of ArcTrip. Note that CircularTrip can be considered a special case of ArcTrip where angle range is the whole space. More specifically we can say that CircularTrip(q, r) is exactly same as ArcTrip$(q, r, \langle -180°, 180°\rangle)$.

3.4 Summary

In this chapter we have presented a novel access method CircularTrip which returns the cells according to their proximity to q. We also presented ArcTrip which can be considered a more general form of CircularTrip and we will show in next chapters how these two access methods can be used to efficiently monitor kNN queries, constrained NN queries and other variants.

Algorithm 2 **ArcTrip(** G**,** q**,** r**,** $\langle\theta_1, \theta_2\rangle$ **)**

Input: G: the grid index; q: query point; r: the radius; $\langle\theta_1, \theta_2\rangle$: angle range

Output: all the cells that have *minadist* smaller than r and intersect the circle with center q and radius r

```
1: $c := c_{start} := c[i,j]$ $(i := \lfloor (q.x + r.Cos\theta_2)/\delta \rfloor,\ j := \lfloor q.y + r.Sin\theta_2/\delta \rfloor)$;$C := \varnothing$;
2: $D_{cur}$ :=direction relevant to the quadrant of $c_{start}$ /* (Quadrant:Direction)$\rightarrow (1 : right), (2 : up), (3 : left), (4 : down)$ */
3: $D_{next}$ := the next direction of $D_{cur}$
4: $c_{opp}$ :=the adjacent cell to $c$ in direction opposite to $D_{cur}$
5: if $(minadist(c_{opp}, q, \langle\theta_1, \theta_2\rangle) < r \leq maxdist(c_{opp}, q))$ then insert $c_{opp}$ into $C$
6: repeat
7:   insert $c$ into $C$;
8:   $c'$ := the adjacent cell to $c$ in $D_{cur}$ direction;
9:   if $c'$ does not intersect the circle then
10:    $c'$ := the adjacent cell to $c$ in $D_{next}$ direction;
11:  $c := c'$;
12:  if $((D_{cur} = (up$ or $down)$ and $c.j = \lfloor (q.y \pm r)/\delta \rfloor)$ OR $(D_{cur} = (right$ or $left)$ and $c.i = \lfloor (q.x \pm r)/\delta \rfloor))$ then
13:    $D_{cur} := D_{next}$;
14:    $D_{next}$ := the next direction to $D_{cur}$;
15: until $minadist(c', q, \langle\theta_1, \theta_2\rangle) > r$
16: return $C$;
```

Chapter 4

CircularTrip Based Continuous kNN Algorithm

Different from a snapshot kNN query, continuous kNN queries are issued once and run continuously to generate results in real-time along with the updates of the underlying datasets. Therefore, it is crucial to develop in-memory techniques to continuously process kNN queries due to frequent location updates of data points and query points. In many applications [XMA05, YPK05, MHP05], it is also crucial to support the processing of a number of continuous kNN queries simultaneously; consequently, scalability is a key issue.

To address the scalability, we focus on two issues: (1) minimization of computation costs; and (2) minimization of the memory requirements. We study continuous kNN queries against the data points that move around in an arbitrary way. Similar to the previous work [XMA05, YPK05, MHP05], we assume that the dataset is indexed by an in-memory grid index. Based on CircularTrip presented in Chapter 3 , we present an efficient algorithm to continuously monitor kNN queries.Compared with the most advanced algorithm CPM [MHP05], our CircularTrip-based continuous kNN algorithm has the following advantages.

1. time efficient: although both algorithms access the minimum number of cells for

initial computation, minimum cells are accessed during continuous monitoring in our algorithm.

2. space efficient: our algorithm does not employ any book-keeping information used in CPM (i.e., visit list and search heap for each query).

Our experimental study demonstrates that CircularTrip-based continuous kNN algorithm is 2 to 4 times faster than CPM, while its memory usage is only 50% to 85% of CPM.

4.1 Motivations

As discussed in Section 2.1.2, CPM is the best known approach for continuous monitoring of kNN queries. Though CPM has performance advantages compared to previous continuous kNN algorithms, it has some serious shortcomings described below.

Space Requirement To update the results of any query q, CPM needs to maintain the extra book-keeping information (i.e., visit list and heap).

Recomputation When one point $p \in q.kNN$ moves farther from q than $q.dist_k$, CPM has to recompute all the kNNs to reflect the update. To do that, CPM examines the cells recorded in the book-keeping information corresponding to q (first processes the visit list and then the heap) till the new kNNs are found. Though, those cells form the minimal set of cells required to be accessed during the initial kNN computation, it is not necessary to access all of them in the update computation. For example, in the case where only the k^{th} NN of q moves farther from q, we do not need to look into the previously accessed cells maintained in the visit list as the new k^{th} NN must lie in the cells intersecting the circle with center q and radius $q.dist_k$ only. Therefore, CPM is not optimal during the continuous monitoring.

Query updates Once CPM receives the location update from a query q, q and all of its related information (i.e., $q.kNN$ and the book-keeping information) is deleted and then a new continuous query is issued on the new location and computed from the scratch. Obviously, such computation does not utilize the previous computed information of q.

We show that by employing CircularTrip, our kNN algorithm guarantees that the minimum number of cells are accessed during computation of initial results and continuous monitoring. CircularTrip based algorithm overcomes the problems described above.

4.2 System Overview and Data Structure

Suppose that P is a set of 2-dimensional moving data points. Each data point $p \in P$ is represented by $(p.x, p.y)$. The data points change their location in an unpredictable fashion and frequently. At each timestamp, the data point p which moves from p_{pre} to p_{cur}, issues a location update as $\langle p.id, p_{pre}, p_{cur} \rangle$. Updates of query points are recorded similarly.

Figure 4.1 illustrates our continuous kNN query monitoring system. There are four components in the system, query table, kNN computation module, monitoring computation module, and CircularTrip module. Once a new continuous kNN query arrives, it is first registered in the query table. Then, kNN computation module retrieves the initial kNN results. When either data points or query points move, monitoring computation module receives the location updates and reports the new kNN results to the corresponding queries. CircularTrip module is invoked by both the kNN computation module and monitoring computation module, which guarantees the minimum number of cells are accessed during all the computation. By *accessing* or *visiting* a cell, we mean that all the objects lying inside it are retrieved and their distances from q are calculated. We use another term *encountering* a cell by which we mean to calculate the *mindist* of that cell from q. Clearly, the cost of encountering a cell is negligible as compared to accessing a cell.

Each cell c contains an *object list* P_c that contains all the objects lying in c. *Influence*

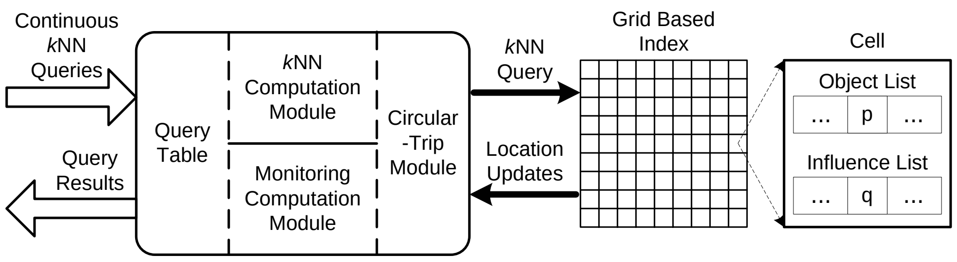


Figure 4.1: System Overview and Data Structure

list Q_c of a cell c contains every query q such that $mindist(c, q) \leq q.dist_k$. Intuitively, influence list of cell c records all the queries q affected by this cell c. Both the object lists and influence lists are implemented as hash tables so that insertion, deletion or retrieval of any element takes constant time.

As shown in Algorithm 3, our CircularTrip-based continuous kNN algorithm consists of two phases. In phase 1, the initial results of each new continuous kNN query are computed. Then, the results are incrementally updated by continuous monitoring module at each time stamp upon the moves of query points and data points. Both phases take advantages of CircularTrip algorithm. Phase 1 and phase 2 are presented in Section 4.3 and Section 4.4, respectively.

Algorithm 3 **Continuous kNN Algorithm**

```
Description:
phase 1: initial kNN computation (Algorithm 4)
    Retrieve the initial kNN results for each new query.
phase 2: continuous monitoring (Algorithm 5) (at each time stamp)
    receive updates
    update the results
```

4.3 Initial kNN Computation

The basic idea of kNN computation algorithm is to access the cells around query point q round by round. A *round* C_i contains all the cells that intersect the circle of radius

$r_i = r_0 + i\delta$ centered at q. To collect a round C_i, we call our CircularTrip algorithm with radius set as r_i. Formally, $C_i = \{\forall c \mid \mathit{mindist}(c, q) < r_i \leq \mathit{maxdist}(c, q)\}$. r_0 is the the first circle's radius. Obviously, r_0 is at most $\mathit{maxdist}(c^q, q)$; otherwise cell c^q will not be accessed by the algorithm. Examples of rounds are shown as the shaded cells in Fig. 4.2. In each round, the algorithm accesses the cells in ascending order of their $\mathit{mindist}(c, q)$. The algorithm terminates when the next cell to be accessed has $\mathit{mindist}(c, q) \geq q.\mathit{dist}_k$.

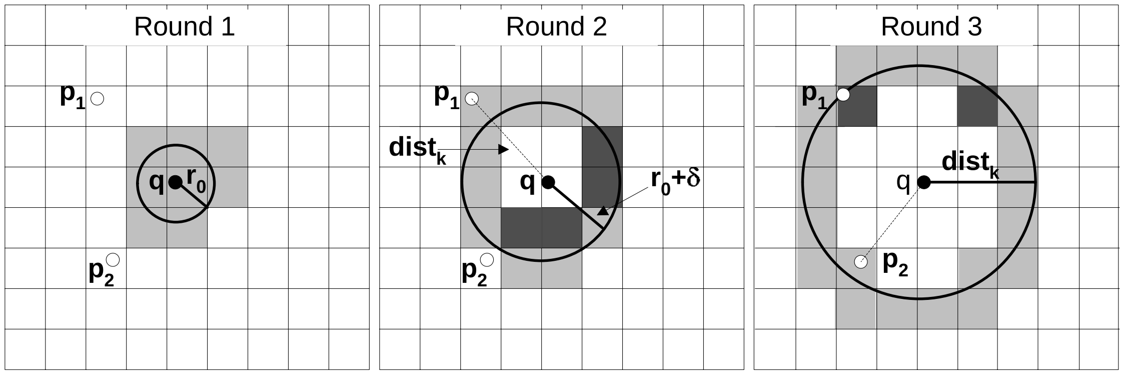


Figure 4.2: A Nearest Neighbor Query

The detailed kNN computation algorithm is shown in Algorithm 4. In the algorithm, the cells c in a round are maintained in a heap H so that they are accessed in ascending order of their $\mathit{mindist}(c, q)$. Initially, all cells of round C_0 are inserted into a heap. In each iteration, if the top entry e^H of the heap with $\mathit{mindist}(e^H, q)$ not smaller than $q.\mathit{dist}_k$ ($= \infty$, initially), it is immediately known that kNN have been found and the computation terminates. Otherwise, $\mathit{dist}(p, q)$ is computed for every data point $p \in e^H$ and $q.kNN$ and $q.\mathit{dist}_k$ are updated if $\mathit{dist}(p, q)$ is smaller than $q.\mathit{dist}_k$. After all the cells in the heap are processed, the current radius r is checked with $q.\mathit{dist}_k$. If r is smaller than $q.\mathit{dist}_k$, all the cells in the next round with radius as $\min\{r + \delta, q.\mathit{dist}_k\}$ are collected. Among them, the cells which were not processed before (i.e., q is not in their influence list) are inserted into the heap and examined in the same way till kNN are found.

Lemma 4.3.1 *In a grid consisting of cells with size $\delta \times \delta$, given a cell c and a query point q where c does not contain q, $\delta \leq \mathit{maxdist}(c, q) - \mathit{mindist}(c, q) \leq \sqrt{2}\delta$.* ∎

According to Lemma 4.3.1, a cell is intersected by at most two consecutive circles (e.g., the dark shaded cells in Fig. 4.2). Although these cells are encountered twice during kNN computation (i.e., these cells appear in two rounds), they are accessed once only. This is because for a query q (1) our kNN algorithm only accesses the cells where q is not in their influence lists; and (2) q will be inserted into its influence list after a cell is accessed. In fact, Theorem 4.3.2 proves the upper bound of the total number of times the cells are encountered in our algorithm.

Theorem 4.3.2 *In kNN algorithm, the total number of times the cells are encountered is at most* 1.27 *times of the number cells in the minimum set of cells.*

Proof Let R be $q.dist_k/\delta$ and $|C_i|$ be the number of cells in round C_i. The total number of times the cells are encountered is at most $4R^2$ and the number of cells in the minimum set of cells is at least πR^2. As a result, the ratio of them is at most $4R^2/\pi R^2 = 1.27$. ▮

Example Fig. 4.2 illustrates a concrete example of an NN query. As no data point is found in the first round, the algorithm continues to process the cells in the next round with radius $(r_0 + \delta)$. In this round, p_1 is found and $q.dist_k$ is updated to be $dist(p_1, q)$. Then, a third round with radius $q.dist_k$ (as $dist(p_1, q) < r_0 + 2\delta$) is processed because the previous radius is smaller than $q.dist_k$. In round 3, $q.kNN$ and $q.dist_k$ are updated after p_2 is found. Computation stops when $q.dist_k$ $(= dist(p_2, q))$ is less than $mindist(e^H, q)$ of the top entry e^H.

4.4 Continuous Monitoring

Recall that at each time stamp, data point p which moves from p_{pre} to p_{cur} issues an update $\langle p.id, p_{pre}, p_{cur}\rangle$. The updates of query points are recorded similarly. Upon these location updates, continuous monitoring module identifies the affected queries and incre-

Algorithm 4 **ComputeNN(** G**,** q**,** k **)**

Input: G: the grid index; q: query point; k: an integer;

Output: the kNN of q;

```
1:  $q.dist_k := \infty$; $q.kNN := \varnothing$; $H := \varnothing$; $r := r_0 := maxdist(c^q, q)$;
2:  insert the cells returned by CircularTrip($G$, $q$, $r$) into $H$;
3:  while $H \neq \varnothing$ and $mindist(e^H, q) < q.dist_k$ do
4:      insert $q$ into the influence list of $e^H$;
5:      $\forall p \in e^H$, compute $dist(p, q)$ and update $q.dist_k$ and $q.kNN$;
6:      remove $e^H$ from $H$;
7:      if $H = \varnothing$ and $r < q.dist_k$ then
8:          $r := \min\{r + \delta, q.dist_k\}$;
9:          cells $C :=$ CircularTrip($G$, $q$, $r$);
10:         $\forall c \in C$, insert $c$ into $H$ if $q \notin$ the influence list of $c$;
11: return $q.kNN$;
```

mentally updates their results. The goal of this phase is to maximize sharing previous computation and results.

4.4.1 Handling Data Point Updates

Regarding a query q, the update of data point p can be classified into 3 categories:

- *internal update:* $dist(p_{pre}, q) \leq q.dist_k$ and $dist(p_{cur}, q) \leq q.dist_k$; Clearly, only the order of $q.kNN$ is affected so we update the order of data points in $q.kNN$ accordingly.

- *incoming update:* $dist(p_{pre}, q) > q.dist_k$ and $dist(p_{cur}, q) \leq q.dist_k$; p may have become a result so we insert it in $q.kNN$.

- *outgoing update:* $dist(p_{pre}, q) \leq q.dist_k$ and $dist(p_{cur}, q) > q.dist_k$; p is deleted from $q.kNN$ because it is no more a part of the answer set.

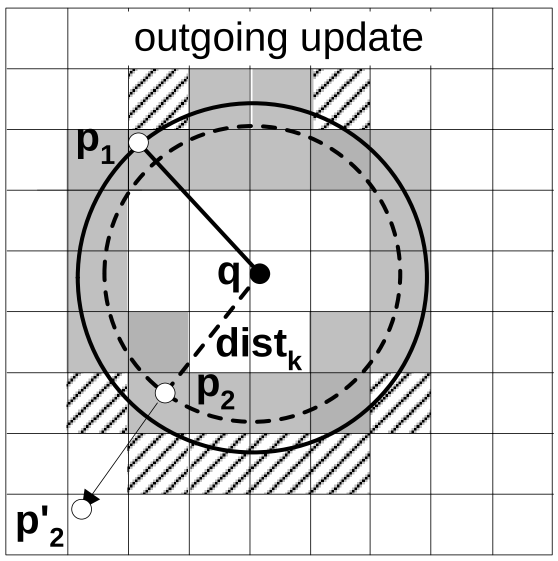
It is immediately verified that only the queries recorded in the influence lists of cell $c^{p_{pre}}$ or cell $c^{p_{cur}}$ may be affected by the update $\langle p.id, p_{pre}, p_{cur} \rangle$, where $c^{p_{pre}}$ ($c^{p_{cur}}$) is the cell containing p_{pre} (p_{cur}). Therefore, after receiving an update $\langle p.id, p_{pre}, p_{cur} \rangle$, continuous monitoring module checks these queries q only and takes the necessary action as described above. Note that a new point registered at system can be treated as an object update assuming p_{pre} be at infinite distance from q. Similarly, a point deletion can be considered as an object moved to a location at infinite distance from q ($dist(p_{cur}, q) = \infty$).

After all the updates are handled as described above, we update the results of each affected query q as follows.

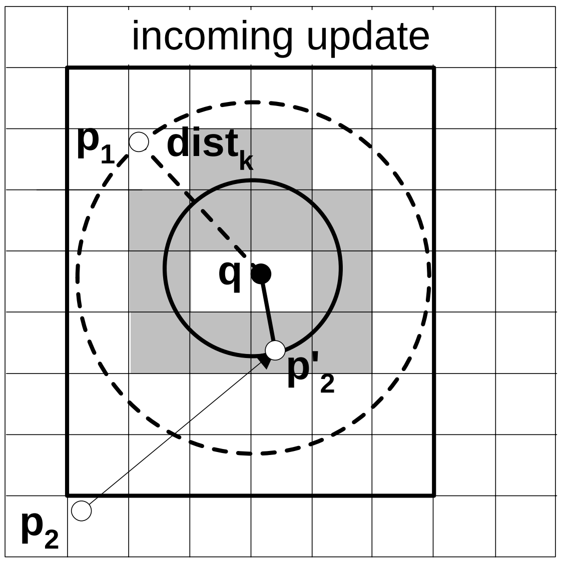
- *Case 1: if* $\mid q.kNN \mid \geq k$. We keep only k closest points to q and delete all others. Then, $q.dist_k$ is updated accordingly. After updating the results, $q.dist_k$ may become smaller. So, we have to remove q from the influence lists of cells c whose $mindist(c, q)$ is greater than current $q.dist_k$. The update of influence lists is discussed in section 4.4.4.

- *Case 2: if* $\mid q.kNN \mid < k$. The recomputation is similar to initial kNN computation but we only need to find the new ($k- \mid q.kNN \mid$) nearest neighbors of q instead of all kNNs. The only difference is that, we set the radius of the first circle r_0 as $q.dist_k$ and then initialize $q.dist_k$ to be ∞. Similar to Algorithm 4, we repeatedly collect cells round by round and examine them in ascending order of their minimum distance to q till all kNNs are found. When accessing a cell, we only consider data points p which do not already belong to $q.kNN$.

Example Fig. 4.3 shows an example where q is an NN query and data point p_2 moves to p'_2. The first update shown in Fig. 4.3(a) is outgoing update as q's NN p_2 moves far away. To handle it, cells of round with radius $q.dist_k$ are collected and accessed (i.e., shaded cells). Since p_1 is found in this round, a second round with radius $dist(p_1, q)$ is processed (i.e., striped cells). Then, computation ends as no new NN is found. To handle the second update of Fig. 4.3(b), as p'_2 is closer to q than its NN p_1, we simply replace $q.kNN$ with p'_2 and remove q from the influence lists of cells c that lie in the bold rectangle but whose $mindist(c, q)$ is greater than $q.dist_k$. The details of updating



(a) Handling an outgoing update p_1'



(b) Handling an incoming update

Figure 4.3: Handling Data Point Updates

influence list will be presented in Section 4.4.4

4.4.2 Handling Query Updates

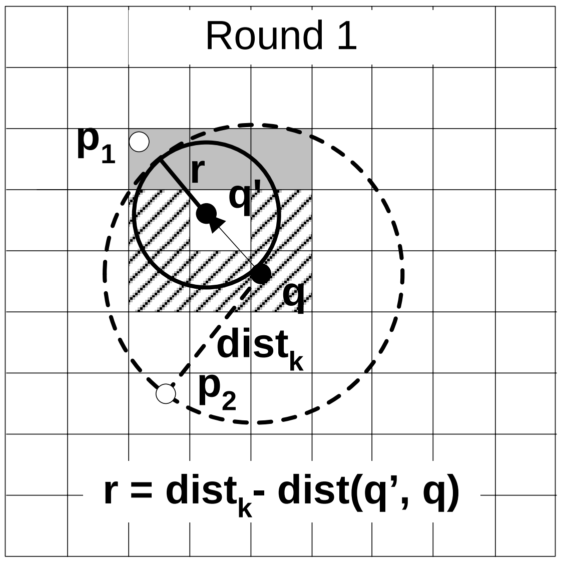
To handle update $\langle q.id, q_{pre}, q_{cur}\rangle$ of query q, the straight forward way is to delete its results and all of its related information (e.g., remove q from the influence lists of related cells), and then issue a new query on location q_{cur} and compute it from scratch. Clearly, this update computation does not utilize the previous computed information. Let $q_{pre}.dist_k$ ($q_{cur}.dist_k$) be the distance of k^{th} NN from q when q at q_{pre} (q_{cur}). The kNN results of q at location q_{pre} and q_{cur} share nothing if and only if $dist(q_{pre}, q_{cur}) > q_{pre}.dist_k + q_{cur}.dist_k$. Without loss of generality, we assume dataset follows uniform distribution. Consequently, $q_{pre}.dist_k = q_{cur}.dist_k$. Note that for other distributions, the similar formula can be derived by density functions. Therefore, in order to maximize sharing previous computed results, we delete the previous query and issue a new query only when query point moves farther than $2q_{pre}.dist_k$.

Handling single query update is described as follows.

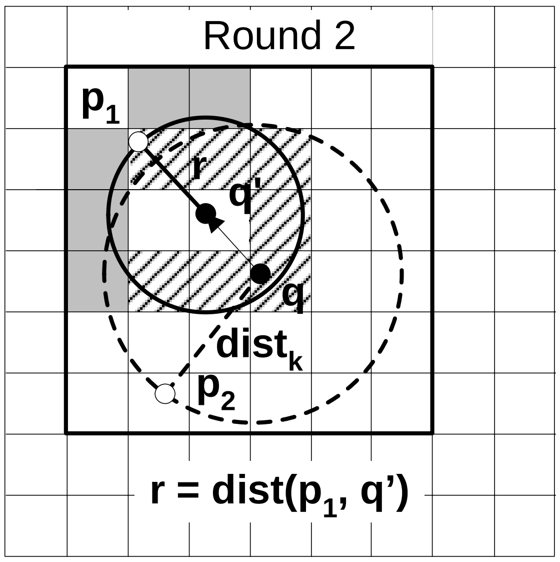
- *Case 1: $dist(q_{pre}, q_{cur}) \geq 2q_{pre}.dist_k$.* Query q and its kNN result are discarded first.

Then, we remove q from the influence lists of cells which lie in the rectangle containing the circle with center at q_{pre} and radius as $q_{pre}.dist_k$. After that, a new continuous kNN query is issued on q_{cur}.

- *Case 2:* $dist(q_{pre}, q_{cur}) < 2q_{pre}.dist_k$. We first calculate $dist(p, q_{cur})$ for every kNN p of q_{pre} and update $q_{cur}.dist_k$ as the maximum of them. i.e., $q_{cur}.dist_k = max_{\forall p \in q.kNN}$ $dist(p, q_{cur})$. Then, recompute kNN for q_{cur} in the similar way to the initial kNN computation. The first circle's radius r_0 is $max\{max(c^q, q), q_{pre}.dist_k - dist(q_{pre}, q_{cur})\}$. During computation, the data points in $q_{pre}.kNN$ are skipped as they have been computed already. Clearly, the cells with $maxdist(c, q_{pre}) \le q_{pre}.dist_k$ are also skipped as all data points in these cells belong to $q_{pre}.kNN$.



(a) p_1 is found in first round



(b) p_1 is confirmed as NN

Figure 4.4: Handling Query Updates

Example In Fig. 4.4, an NN query q moves to q'. To handle this update, we first update $q.kNN$ and $q.dist_k$ by calculating the distance between q' and its current NN p_2. Then, we examine the cells intersected by the circle with center at q' and radius $(q_{pre}.dist_k - dist(q', q))$. Note that only the shaded cells of Fig. 4.4(a) in this round are accessed as others (i.e., striped cells) satisfy $maxdist(c, q_{pre}) \le q_{pre}.dist_k$. Since p_1 is found, the radius of next round is $dist(p_1, q')$. Similarly, in round 2 only the shaded cells of Fig. 4.4(b) are processed. Finally, influence lists of related cells (i.e., cells in the bold square and with $mindist(c, q_{cur}) > q_{cur}.dist_k$) are updated (i.e, remove q from them).

4.4.3 Complete Update Handling Module

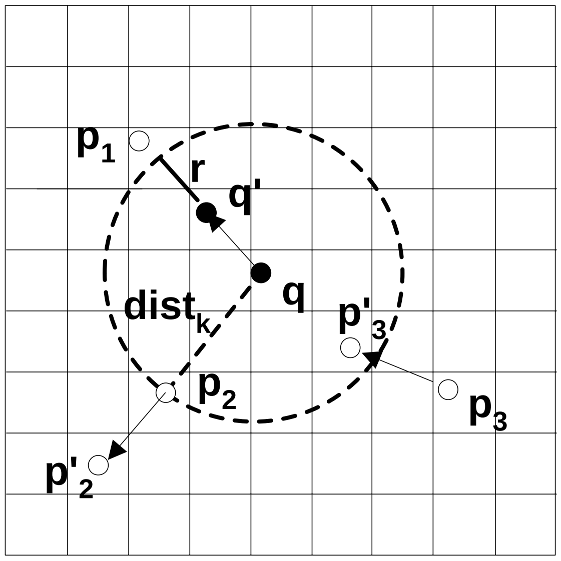
In section 4.4.1 we described how to update the results on receiving data point updates and we discussed handling the updates of queries in section 4.4.2. Now we describe how to deal with data point and query point updates simultaneously at each time stamp. Our update handling module consists of two phases.

•*receive updates:* First we receive the query point updates and then we handle data point updates. For each query update, if $dist(q_{pre}, q_{cur}) \geq 2q_{pre}.dist_k$, we delete q and insert a new query in system. Otherwise, we mark the query as *moving* and calculate $dist(p, q_{cur})$ for every kNN p of q_{pre} and temporarily set $q_{cur}.dist_k$ as $(q_{pre}.dist_k + dist(q_{pre}, q_{cur})$.

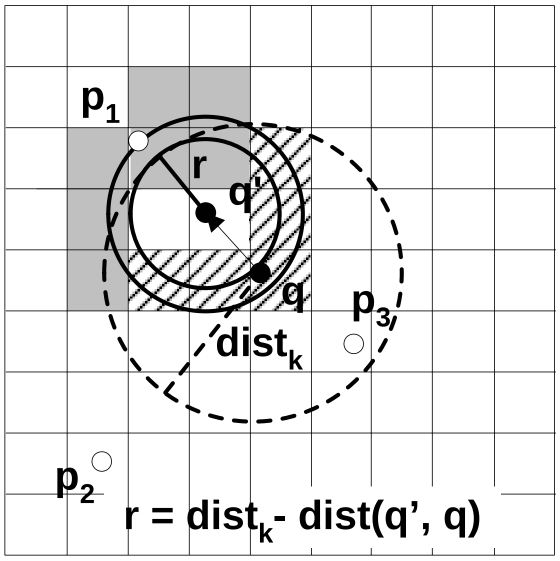
After handling query updates, we receive point updates. For each query q affected by a point update, we insert (delete) p in (from) $q.kNN$ if it is an incoming (outgoing) update. Only the order of $q.kNN$ is changed in case of internal update.

•*update results:* We update the results of all affected queries in this phase. The new queries are updated by calling initial computation module described in Algorithm 4. The affected queries that were not moved are handled as described in section 4.4.1. The queries marked as *moving* are updated by first setting the $q_{cur}.dist_k$ equal to the distance of k^{th} data point in $q.kNN$ (set as ∞ if $\mid q.kNN \mid < k$). The update algorithm is similar to initial kNN algorithm and the radius of first round r_0 is $max\{max(c^q, q), (q_{pre}.dist_k - dist(q_{pre}, q_{cur})\}$. During computation, the data points that are already in $q.kNN$ are ignored. Clearly, the cells with $maxdist(c, q_{pre}) \leq q_{pre}.dist_k$ are also skipped as all the data points in these cells belong to $q.kNN$. The algorithm terminates when the next cell c to be accessed has $mindist(c, q) \geq q_{cur}.dist_k$.

Example Fig. 4.5 presents a concrete example of our complete update handling module. As shown in Fig. 4.5(a), the data points p_2 and p_3 move to p'_2 and p'_3, respectively. The



(a) q, p_2 and p_3 issue location updates



(b) p_1 is found as new NN

Figure 4.5: Handling Multiple Updates

query q also issues location update at q'. Our update handling module first considers the query update and marks it as *moving*. $q.kNN$ is updated by calculating $dist(p_2, q')$. Then $q.dist_k$ is temporarily set as $(dist(q, q') + dist(q, p_2))$. Now algorithm considers the data point updates and p_2 is deleted from $q.kNN$ as it is an outgoing update. The data point p_3 is inserted in $q.kNN$ because it is an incoming update $(dist(q', p_3) < (q.dist_k = dist(q, q') + dist(q, p_2))$. Now the result of q is updated by calling first round with r as shown in Fig. 4.5(b). The data point p_1 is found and to confirm it as answer a last round with radius set as $dist(q', p_1)$ is called. The algorithm accesses shaded cells and the striped cells are ignored because those are fully contained in the dotted circle. i.e; for each striped cell c, $maxdist(c, q) \leq dist_k)$.

Algorithm 5 summarizes the complete update handling module.

4.4.4 Updating the Influence List

After updating the results, we may have to remove q from the influence lists of cells c whose $mindist(c, q)$ is greater than current $q.dist_k$. Consider the running example of Fig. 4.6(a) where after the update of results, both the shaded and striped cells contain q

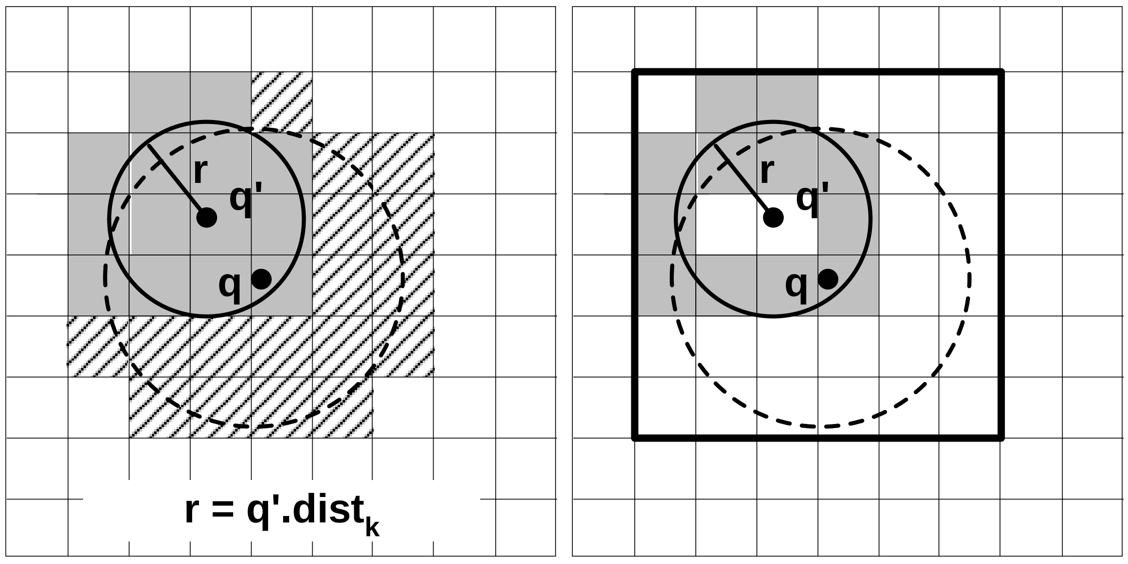
Algorithm 5 Handling Multiple Updates

Description:

```
Step 1: receive updates
1:  for each query update $\langle q.id, q_{pre}, q_{cur} \rangle$ do
2:      case 1: insert $q$ into $Q_{new}$ if $dist(q_{pre}, q_{cur}) \geq 2q_{pre}.dist_k$;
3:      case 2: insert $q$ into $Q_{moving}$ if $dist(q_{pre}, q_{cur}) < 2q_{pre}.dist_k$;
4:  for each $q$ ($\notin Q_{new}$) affected by each data point update $\langle p.id, p_{pre}, p_{cur} \rangle$ do
5:      insert $q$ into $Q_{update}$;
6:      case 1: insert $p$ into $q.kNN$ if incoming update;
7:      case 2: remove $p$ from $q.kNN$ if outgoing update;
Step 2: update results
8:  for each query $q \in Q_{update} \setminus Q_{moving}$ do
9:      if $|q.kNN| \geq k$ then
10:         update $q$ as Case 1 in Section 4.4.1;
11:     else update $q$ as Case 2 in Section 4.4.1;
12: update query $q \in Q_{new}$ ($q \in Q_{moving}$) as Case 1 (Case 2) in Section 4.4.2;
```

in their influence lists whereas only the shaded cells can actually influence the query. So we need to remove q from the influence lists of all striped cells.

A straightforward approach to update influence lists is to first identify a square that contains all the shaded and striped cells. Then, we could check $mindist(c, q)$ of all the cells c in the square and could delete the cells for which $mindist(c, q) > dist_k$. However, this approach requires computing $mindist$ of many cells which can be avoided. We use CircularTrip to efficiently update the influence lists as follows: First we call CircularTrip with radius $r = q'.dist_k$ and mark all the returned cells as shown shaded in Fig. 4.6(b). Then, we identify a square that contains all the shaded and striped cells. The side length of the square can be at most $max\{q_{pre}.dist_k, 2 \times (dist(q_{pre}, q_{cur}) + q_{cur}.dist_k)\}$. Now we remove q from the cells contained in the square that lie outside the marked circle. To do this, for each row of the square, we start processing the cells from the left most cell



(a) before update (b) after update

Figure 4.6: Updating Influence Region

towards the right end cell of the square. For each cell c, we delete q from the influence list. We stop if a cell c is marked or if it resides outside of the square. If the cell c is marked, we continue deleting q from the influence lists from the cells in the same row from right end to left end unless some marked cell is found. The update of influence lists completes when all rows are processed in the way described above. Note that this approach requires computing *mindist* of only the shaded cells of Fig. 4.6(b).s

4.5 Performance Analysis

First we present the proof that our algorithm answers the kNN queries by accessing minimum number of cells and then we compare it with CPM.

Proof of optimality and correctness: Given a query q, in our continuous kNN algorithm, the minimal set of cells are all accessed and only these cells are accessed.[1].

Proof Let $q_{old}.dist_k$ be the distance of k^{th} nearest neighbor from q before the update and $q_{new}.dist_k$ be the distance of k^{th} nearest neighbor after the result has been updated.

[1]The case when query q reports location update and is treated as new query is exception to this claim.

For the case, when $q_{new}.dist_k < q_{old}.dist_k$, our algorithm updates the results without accessing any cell (i.e., in case when number of incoming updates is greater than outgoing updates). So we only consider the case when $q_{new}.dist_k \geq q_{old}.dist_k$.

First we identify the minimal set of cells C that has to be accessed in order to guarantee the correctness then we will show that our algorithm does not access any cell $c' \notin C$.

Corollary 4.5.1 *Only the cells c that has $maxdist(c, q) > q_{old}.dist_k$ can contain the new nearest neighbor.*

Corollary 4.5.2 *Only the cells c that has $mindist(c, q) \leq q_{new}.dist_k$ can contain the new nearest neighbor.*

From corollaries 4.5.1 and 4.5.2, it can be inferred that the minimal set of cells is C, such that for all $c \in C$, $maxdist(c, q) > q_{old}.dist_k$ and $mindist(c, q) \leq q_{new}.dist_k$.

First we show that our algorithm does not access any unnecessary cell. To update the results, our algorithm starts by calling CircularTrip with radius $q_{old}.dist_k$. As the radius of subsequent calls to CircularTrip is always greater than $q_{old}.dist_k$, it can be immediately verified that no cell c can be returned such that $maxdist(c, q) < q_{old}.dist_k$ (satisfies corollary 4.5.1). Moreover, our algorithm accesses cells in strictly ascending order and stops as soon the next cell has $mindist(c, q) \geq q_{new}.dist_k$ (satisfies corollary 4.5.2).

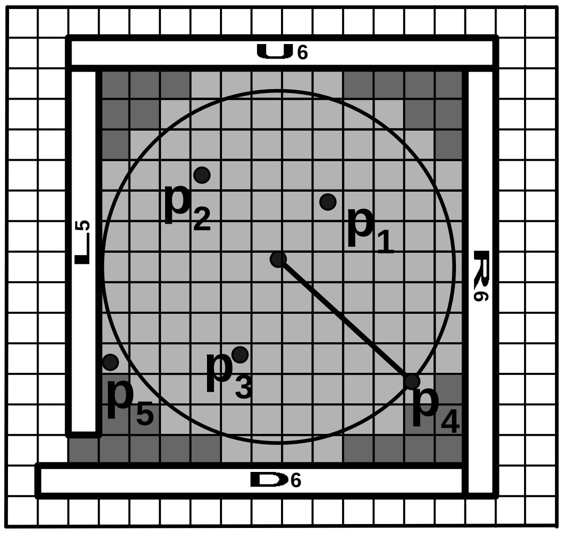
Now as a proof of correctness we show that our algorithm accesses all the cells in minimal set of cells C. The algorithm starts by calling CircularTrip with radius $q_{old}.dist_k$ and the cells c that have $maxdist(c, q) \geq q_{old}.dist_k > mindist(c, q)$ are retrieved. Algorithm iteratively calls CircularTrip with radius increased by δ unless kNNs are found. According to lemma 4.3.1, this increment of δ guarantees that no cell is missed.

Note that initial computation can be considered a special case of update handling where $q_{old}.dist_k$ is zero. ∎

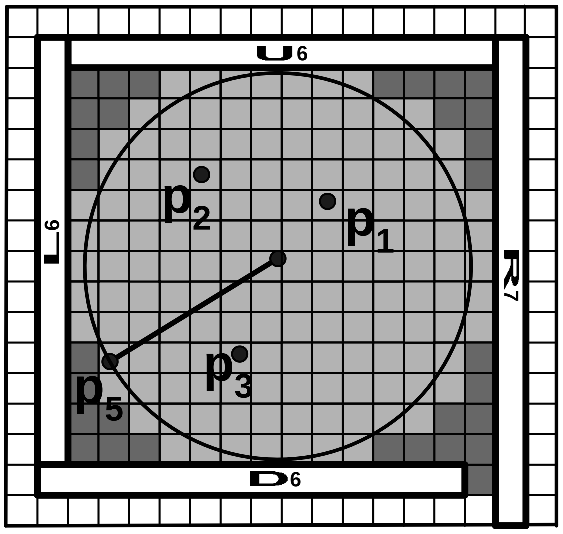
As compared with CPM, our CircularTrip-based algorithm has following advantages.

- CPM accesses minimum possible number of cells only during initial computation of a query but our algorithm accesses minimal set of cells throughout the life of query.

To illustrate that CPM accesses unnecessary cells during update handling, we present a concrete example in Fig. 4.7, where update handling of a $k(=4)$NN query q is shown. Fig. 4.7(a) shows the initial computation of q where the result is $q.kNN = \{p_1, p_2, p_3, p_4\}$. The light-shaded cells are accessed during this computation and form the *visit list*. The dark-shaded cells and four rectangles (L_5, U_6, R_6 and D_6) are the entries in heap H after the initial results have been computed by CPM..



(a) A $k(=4)$-NN query evaluated by CPM

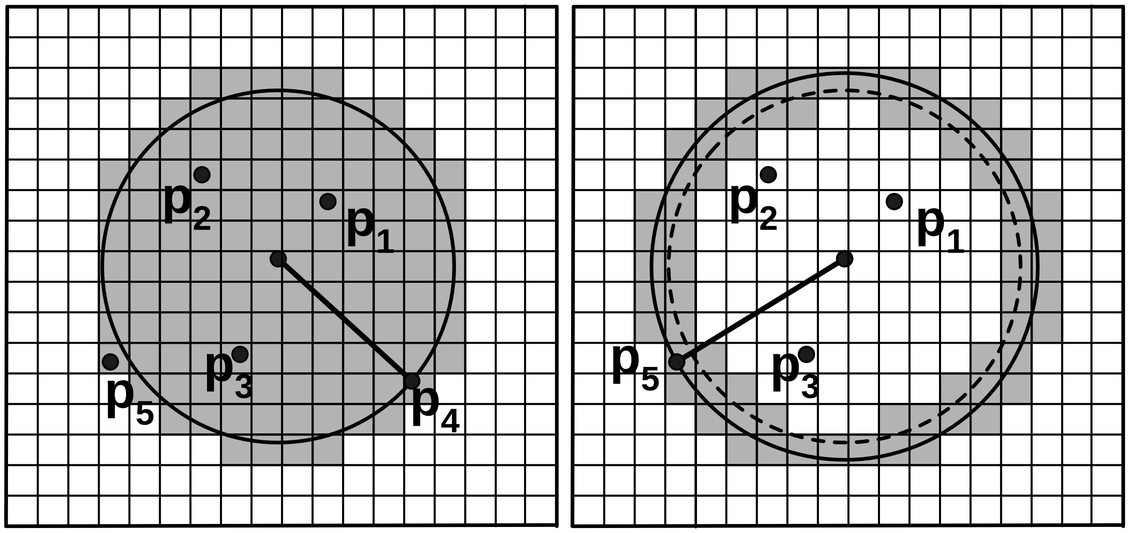


(b) p_4 is deleted, $\{p_1, p_2, p_3, p_5\}$ is updated result

Figure 4.7: An Example of Updates Handling by CPM

Consider an update deletes the object p_4, CPM cannot update the result by starting from the heap H because the new result p_5 is in *visit list* (a cell that was deheaped from H). In order to update the results, CPM first needs to access all the cells in *visit list* and then it continues with heap H. It stops when the next entry (cell or rectangle) in heap has $mindist \geq dist(p_5, q)$. Fig. 4.7(b) shows the update of the results. During this update, CPM accesses light-shaded cells of Fig. 4.7(b). The heap H contains the dark-shaded cells and four rectangles (L_6, U_6, R_7 and D_6) after the result has been updated.

Fig. 4.8 shows the update handling of the same query by our CircularTrip based algorithm. When p_4 is deleted, our algorithm calls CircularTrip with radius set as $dist(p_4, q)$ as shown by dotted circle in Fig. 4.8(b). The new result p_5 is found and the algorithm calls CircularTrip with radius $dist(p_5, q)$ just to confirm that it does not miss any result. The algorithm accesses only the shaded cells of Fig. 4.8(b) and it can be seen that number of cells accessed by our algorithm is much smaller than that of CPM (compare the light-shaded cells of Fig. 4.8(b) and Fig. 4.7(b).



(a) A $k(=4)$-NN query (b) p_4 is deleted from $q.kNN$ and p_5 is added

Figure 4.8: An Example of Updates Handling by CircularTrip-based Algorithm

- Our algorithm does not need to store any book-keeping information for efficient updates as opposed to CPM which stores visit list and heap.

CPM stores visit list and search heap H to use them during updates. The size of visit list and heap H can be approximated to be $(4 + \lceil 2 \times q.dist_k/\delta \rceil^2)$. (i.e., four rectangles + cells in rectangle of side length $2 \times q.dist_k$).

The size of both the visit list and search heap H never decreases unless q reports location update in which case both the visit list and heap H are deleted. Even when the influence region is shrunk, the size of visit list and heap H remains unchanged. On the other hand, when the influence region expands the visit list and heap H also expand.

- CPM does not utilize the previously computed information when a query reports location update.

Once CPM receives the location update from a query q, q and all of its related information (i.e., $q.kNN$ and the book-keeping information) are deleted and then a new continuous query is issued on the new location and computed from the scratch. Clearly, this approach does not utilize the previously computed information. Based on density function, our algorithm makes an optimistic estimate and utilizes the previously computed information if it is expected to be more efficient. Otherwise, the results are updated by deleting and treating q as a new query. In any case, our algorithm performs better than CPM (our experiments demonstrate that the initial computation module of CircularTrip based algorithm is faster than initial computation module of CPM because we do not need to maintain extra book-keeping information).

4.6 Experimental Study

In this section, we evaluate our continuous kNN algorithm. Since CPM significantly outperforms other existing algorithms, it is used as a benchmark algorithm in our evaluation. The following algorithms have been implemented by C++. All experiments were conducted on the PCs with P4 3.2GHz CPU and 2GB memory.

1. *CPM*: CPM continuous kNN algorithm [MHP05].
2. *CircularTrip*: our CircularTrip-based continuous kNN algorithm (Algorithm 3).

In accordance with the experimental study of previous work [XMA05, MHP05], the same spatio-temporal data generator [Bri02] is employed. Specifically, this data generator outputs a set of data points moving on the road network of Oldenburg, a German city. Every data point is represented by its location at successive time stamps. Parameter *data point agility* indicates the percentage of total data points that report their location updates at each time stamp. After reaching its destination, a moving data point randomly selects a new destination and continues moving toward it. Moving speed may be *slow*, *medium*, and *fast*. Data points with slow speed move 1/250 of the extent of space per time stamp. Medium and fast speed are 5 and 25 times faster than slow speed, respectively. Continuous kNN queries are generated in the similar way. All queries are evaluated at each time stamp and the length of evaluation is 100 time stamps. Table below lists the parameters which may potentially have an impact on our performance study. In our experiments, all parameters use default values unless otherwise specified.

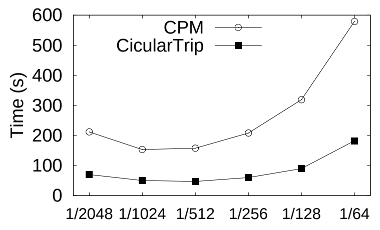
Parameter	Range	Default Values
cell size (δ)	$\frac{1}{2048}, \frac{1}{1024}, \frac{1}{512}, \frac{1}{256}, \frac{1}{128}, \frac{1}{64}$	$\frac{1}{256}$
number of NNs (k)	1, 4, 16, 64, 256	16
number of data points (N)	30, 50, 70, 100, 150, 200 (K)	100K
number of queries (n)	1, 3, 5, 7, 10 (K)	5K
data point agility	10, 30, 50, 70 (%)	50%
query agility	10, 30, 50, 70 (%)	30%
moving speed	slow, medium, fast	medium

4.6.1 Evaluating Efficiency

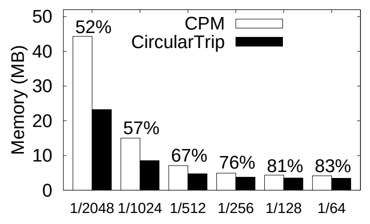
In this subsection, we evaluate efficiency of our continuous kNN algorithm against different settings. The first set of experiments is conducted on various cell sizes. Fig. 4.9 shows the experimental results of CPU time and space requirement.

The larger the cells are, the larger the influence region of a query is. Here, influence region of a query q is the total area of cells c whose $mindist(c, q) \leq q.dist_k$. As a result, the time costs of both algorithms on grid index with larger cells is more than ones with smaller cells. On the other hand, when the cells are too small (i.e., $\delta = \frac{1}{2048}$), many cells are empty which introduces more costs. It is clear that CircularTrip outperforms CPM on all cell sizes and CPM is more sensitive to cell size than CircularTrip. As described before, CircularTrip does not keep any book-keeping information. So, its memory space is always less than CPM's. The numbers above bars in Fig. 4.9 (b) are ratios of their memory spaces.

The second experiment tests an effect of k. From the experiment results as shown in Fig. 4.10, we can see that CircularTrip is always 2 times faster than CPM while its

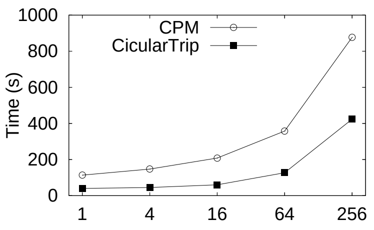


(a) Time

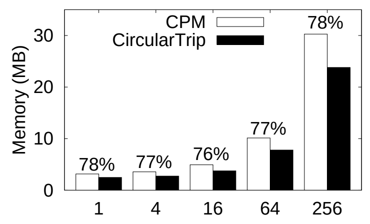


(b) Space

Figure 4.9: Effect of δ

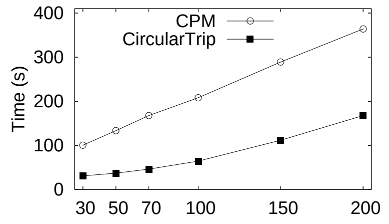


(a) Time

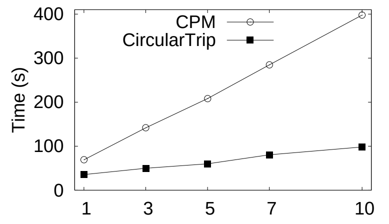


(b) Space

Figure 4.10: Effect of k



(a) Varying N ($\times 1K$)



(b) Varying n ($\times 1K$)

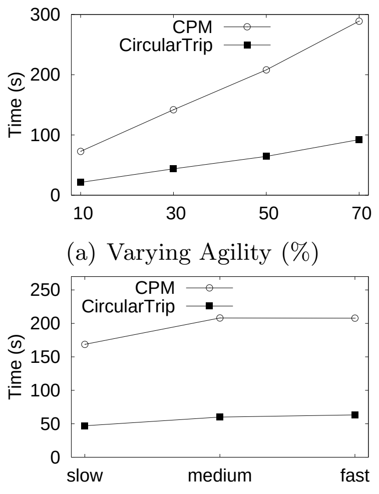
Figure 4.11: Effect of N and n

memory space is at most 78% of CPM's.

The final experiment in this part is to evaluate an impact of cardinality of data points and queries. The experiment results are depicted in Fig. 4.11. Following the trends in the previous experiments, CircularTrip is faster than CPM on all settings. It is interesting that with the number of queries increasing, performance of CPM degrades more significantly than CircularTrip. This is because our algorithm minimizes both initial kNN computation costs and continuous monitoring cost while only the cost of initial kNN computation is minimized in CPM. When $n = 10K$, CPM is 4 times slower than CircularTrip.

4.6.2 Evaluating Effect of Agility

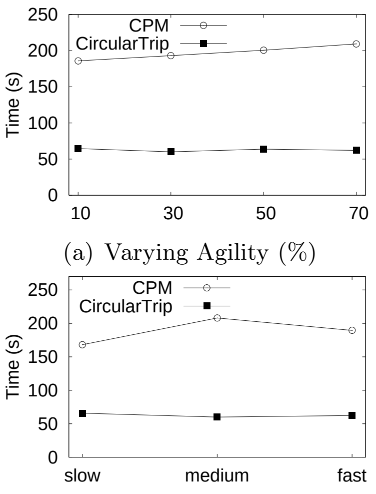
We first evaluate effects of agility and moving speed of data points. The CPU time of both algorithms is reported in Fig. 4.12. It demonstrates that when more location updates are issued at each time stamp, CPM's performance decreases more rapidly than CircularTrip. This is because to update results of the affected queries, CPM has to re-examine cells in their visit lists which causes more cells to be accessed during continuous monitoring. On the other hand, both algorithms are not sensitive to the moving speed of data points.



(a) Varying Agility (%)

(b) Varying Speed

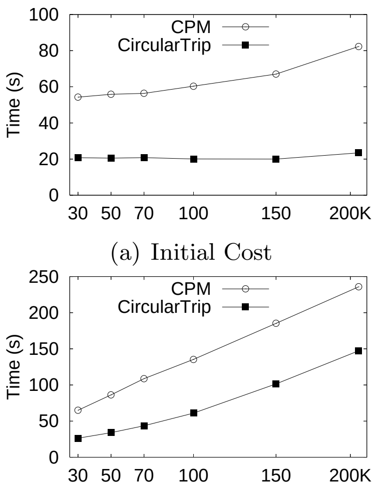
Figure 4.12: Data Movement



(a) Varying Agility (%)

(b) Varying Speed

Figure 4.13: Query Movement



(a) Initial Cost

(b) Monitoring Cost

Figure 4.14: Time Efficiency

Similarly, effects of agility and moving speed of queries are evaluated and the results are shown in Fig. 4.13. Due to the fact that CPM always deletes and issues new queries to update the moving queries, its time cost increases when more queries move. Performance of CircularTrip is stable because it takes advantages of previous computed results when handling query updates. Also, moving speed of queries does not affect both algorithms.

In the last set of experiments, we study the efficiency of initial computation and continuous monitoring separately. In the first experiment, all queries are moving and all data points are static. We slightly modify our algorithm so that it handles moving queries as the same as CPM. Clearly, for both algorithm all costs are initial computation cost only. Fig. 4.14(a) illustrates the results with respect to the data points cardinality. Although both algorithms access minimum number of cells, sCircularTrip's initial cost is 2 to 4 times smaller than CPM's because CPM has to maintain extra book-keeping information for each query. The second experiment studies continuous monitoring cost where all queries are static and their initial costs are omitted. As shown in Fig. 4.14(b), it is confirmed that CircularTrip accesses less cells than CPM during continuous monitoring.

As a short summary, our performance evaluation indicates that compared with CPM, CircularTrip-based kNN algorithm is not only more time-efficient but also more space-efficient. Moreover, our algorithm is more scalable than CPM regarding cell size, k, and cardinality as well as agility and moving speed of data points and queries.

4.7 Conclusion

In this chapter, we proposed an efficient CircularTrip-based continuous kNN algorithm. Compared with the existing algorithm, our technique accesses the minimum set of cells for initial computation and significantly reduces the continuous monitoring cost, while less memory space is required.

Chapter 5

Applications to Other Variants of NN Queries

In this chapter, we show how CircularTrip or ArcTrip can be used to efficiently monitor the variants of NN queries. Our CircularTrip or ArcTrip based algorithms to answer such queries preserve the following properties

1. Algorithms visit minimum number of cells during the continuous monitoring of these queries

2. No extra book-keeping information is required to answer the queries

To the best of our knowledge, there is no existing approach that can efficiently monitor such queries. Though CPM can be used to answer such queries, it is easy to verify that CPM is not efficient.

The rest of the chapter is organized as follows. In section 5.1, we present algorithms for constrained NN queries. We present the technique to continuously monitor farthest neighbors in Section 5.2. $(k + m)$-NN queries are discussed in Section 5.3.

5.1 Constrained Nearest Neighbor Queries

A constrained nearest neighbor query (CNN) [FSAA01] is a nearest neighbor query that is constrained to a specific region. This type of queries is of interest to the users who wish to find the nearest neighbor in some constrained region and not in the whole space. We had discussed the previous work on constrained nearest neighbor queries in Section 2.2.2. In [FSAA01], authors propose different techniques to answer CNN queries but to the best of our knowledge there is no efficient algorithm available for continuously monitoring the CNN queries. Continuous kCNN query can be defined as to monitor the k nearest neighbors in a constrained region. The constrained region is bounded by certain spatial conditions. Below, we categorise continuous kCNN queries based on the constraints that bound the region.

- **Continuous Pie-Region kCNN Queries:** The continuous k constrained nearest neighbors queries where the region is bounded by some angle range $\langle \theta_1, \theta_2 \rangle$ are called continuous Pie-Region kCNN queries. Formally, it monitors the kNNs among a set of objects $P' \subseteq P$ such that all objects in P' lie within the specified angle range. We call it Pie-Region queries because the constrained region is a pie shaped region.

- **Continuous Donut-Region kCNN Queries:** Donut-Region kCNN queries are the continuous kCNN queries where the region is constrained by distance bounds. Let d_L and d_U be the lower and upper distance bounds, respectively. Continuous Donut-Region kCNN queries can be formally defined as monitoring the kNNs among a set of objects $P' \subseteq P$ such that for all $p \in P'$, $d_U \geq dist(p, q) \geq d_L$.

- **Continuous Donut-Pie kCNN Queries:** As apparent from the name, a Donut-Pie kCNN query constrains the region by both the angle and distance bounds. Let $\langle \theta_1, \theta_2 \rangle$ be the angle range and d_L and d_U be the lower and upper distance bounds, respectively. Continuous Donut-Pie queries monitor the kNNs among the set of objects $P' \subseteq P$ such that for all $p \in P'$, $d_U \geq dist(p, q) \geq d_L$ and p lies within the angle range $\langle\theta_1, \theta_2\rangle$. The region looks like a pie taken from a donut, so we name it Donut-Pie queries.

- **Continuous kCNN Queries over Irregular Region:** Given an irregular region R, such queries are to continuously monitor kNNs among the data objects $P' \subseteq P$ that lie inside R.

In this section we present techniques for continuous monitoring of these kCNN queries. We also present the techniques where the spatial conditions that constrain the region can be changed by the user at any time and our approach visits minimum number of cells for all the updates. More specifically, we present our technique to answer continuous pie-region kCNN queries in section 5.1.1, donut-region kCNN queries in section 5.1.2, donut-pie kCNN queries in section 5.1.3 and continuous monitoring of kCNN queries over irregular shaped region is described in section 5.1.4. In section 5.1.5 we give the concluding remarks showing the superiority of our approach.

5.1.1 Continuous Pie-Region kCNN queries

As defined earlier, a continuous pie-region kCNN query q with the angle bounds $\langle\theta_1, \theta_2\rangle$ is to find k nearest neighbors among a set of points $P' \subseteq P$ such that all points in P' lie within the angle range $\langle\theta_1, \theta_2\rangle$. Such kCNN queries have many applications. Consider, for example, a user might be interested in only the points that are in north-east. In this case a query with angle range $\langle 90°, 180°\rangle$ will be issued.

Recall that in order to continuously monitor reverse nearest neighbors, six continuous CNN queries are issued [XZ06] as discussed in Section 2.2.1. Each continuous CNN query monitors the nearest neighbors in its region that covers an angle range of 60° degrees as shown in Fig. 5.1. The algorithm needs to continuously monitor nearest neighbors in the six constrained regions from S_0 to S_5. Continuous CNN query that monitors constrained region S_1 can be denoted as $q_{\langle 60°,120°\rangle}$ because it monitors the nearest neighbors in the region that is bounded by the angle range $\langle 60°, 120°\rangle$.

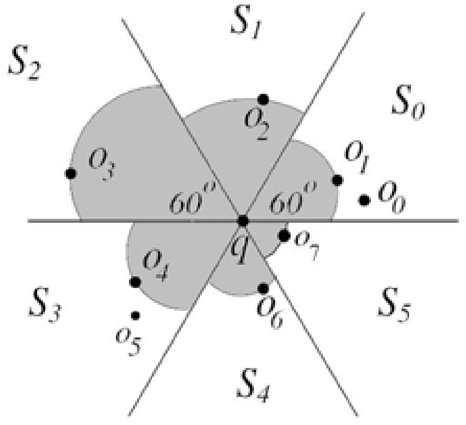


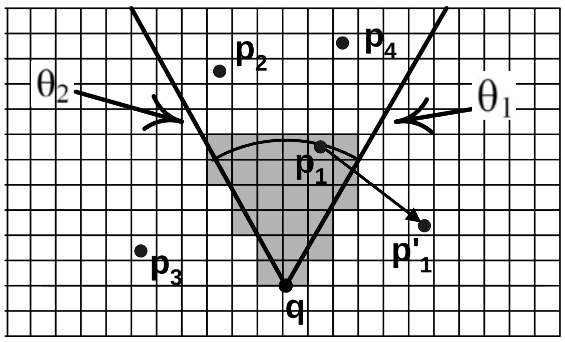
Figure 5.1: Six Pie-Region Constrained NN Queries

Technique

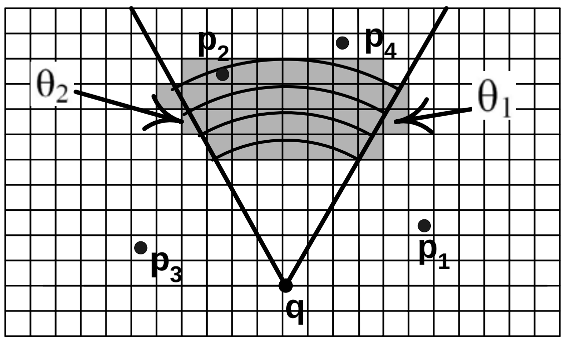
We presented ArcTrip in chapter 3 and now we discuss how ArcTrip can be used to efficiently monitor continuous pie-region kCNN queries. The continuous monitoring of a kCNN query q with angle range $\langle\theta_1, \theta_2\rangle$ is similar to the continuous monitoring of NN queries described in chapter 4. The only difference is that we call ArcTrip$(q, r, \langle\theta_1, \theta_2\rangle)$ instead of CircularTrip(q, r) and for any object p that lies outside the constrained region $dist(p, q) = \infty$.

Below we describe the continuous monitoring of the pie-region kCNN query assuming that initial results are known because the computation of initial results is trivial. Similar to monitoring of NN queries, the update modules handles the updates by categorising them as internal, incoming and outgoing updates. After handling all the updates, if $q.kNN$ contains more than or equal to k objects, the result of query is updated by selecting the k closest objects and deleting others. Otherwise if $q.kNN$ contains less than k objects, ArcTrip with angle range $\langle\theta_1, \theta_2\rangle$ and radius $q.dist_k$ is called and radius is incremented by δ everytime unless kNNs are found.

Example In Fig. 5.2(a), the object p_1 was the constrained nearest neighbor of q and $q.dist_k = dist(p_1, q)$. The shaded cells are those that were visited during initial computa-



(a) p_1 issues an update at p_1'



(b) p_2 is new CNN

Figure 5.2: A Continuous Pie-Region 1-CNN query

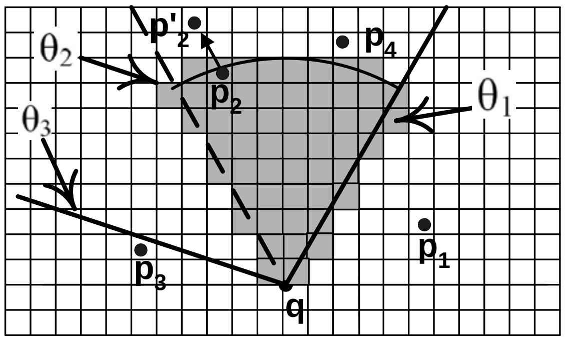
tion. The object p_1 reports update at location p_1' and it is deleted from $q.kNN$ because its distance $(dist(p_1, q) = \infty) > q.dist_k$ (an outgoing update) . To update the results of the query, the algorithm starts calling ArcTrip with angle range $\langle\theta_1, \theta_2\rangle$ and radius set as $q.dist_k$. The radius is iteratively increased by δ every time unless the new CNN p_2 is found. The algorithm visits shaded cells shown in Fig. 5.2(b) to update the results.

Extension to Varying Pie-Region Continuous kCNN Queries

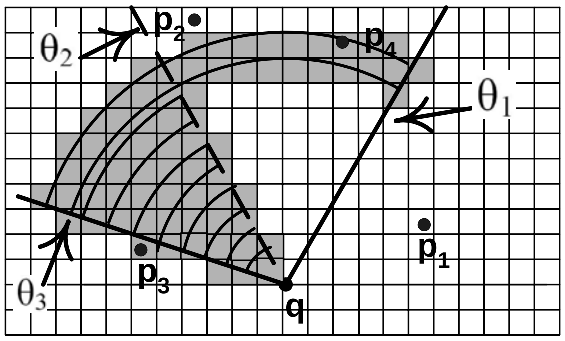
At any time a user may wish to change the spatial conditions that constrain the region of a continuous kCNN query. For example a user may change the angle range that bounds the pie-region which may expand or shrink the constrained region. Consider the running example of Fig. 5.3 where p_2 is the CNN of q. The user changes (expands) the constrained region and the new pie-region is now constrained by angle range $\langle\theta_1, \theta_3\rangle$. Below we describe how we update the result of such queries by visiting the minimum possible number of cells.

If the angle range is decreased, the old kCNNs that lie outside the new angle range are deleted and the algorithm continues with the ArcTrip of new angle range and radius $r = q.dist_k$ until kCNNs are found. If the angle range increases, we visit only the cells that lie in the angle range that is not covered by the old angle range. The example below illustrates how the update works when the constrained region is expanded.

Example In Fig. 5.3, the user has changed the angle range to $\langle\theta_1, \theta_3\rangle$ that has increased



(a) change in constrained region and location of p_2



(b) p_4 is new CNN

Figure 5.3: A Varying Pie-Region Continuous 1-CNN Query

the size of pie-region. The old CNN p_2 also issues update at location p_2' and is deleted from $q.kNN$ because it is an outgoing update. The algorithm first visits the cells that lie in angle range $\langle\theta_2, \theta_3\rangle$ by calling ArcTrip with iteratively increased radius (initially set as $maxdist(c^q, q)$). At this stage p_3 is found but it is not considered because it lies outside the constrained region. The algorithm continues till the radius exceeds $q.dist_k$. The algorithm sets the radius of ArcTrip to $q.dist_k$ and angle range is reset to $\langle\theta_1, \theta_3\rangle$. ArcTrip is iteratively called with this new angle range and radius increased by δ and the new CNN (p_4) is found which is reported as answer. Note that the algorithm visits only the shaded cells of Fig. 5.3(b) and the result cannot be guaranteed if any of these cells is missed.

5.1.2 Continuous Donut-Region kCNN Queries

In this section we present algorithm to continuously monitor a kCNN query where the region of the query is constrained by distance from q. Formally, such a query q continuously monitors the nearest neighbors among a set of objects $P' \subseteq P$ such that for all $p \in P'$, $d_U \geq dist(p, q) \geq d_L$ where d_U and d_L are the upper and lower bounds of distance, respectively. Let C_U and C_L be the circles centered at q and with radii d_U and d_L, respectively. The constrained region is the donut shaped area $C_U - C_L$ shown shaded in Fig. 5.4.

As opposed to simple NN queries where the result would have been p_1, the result

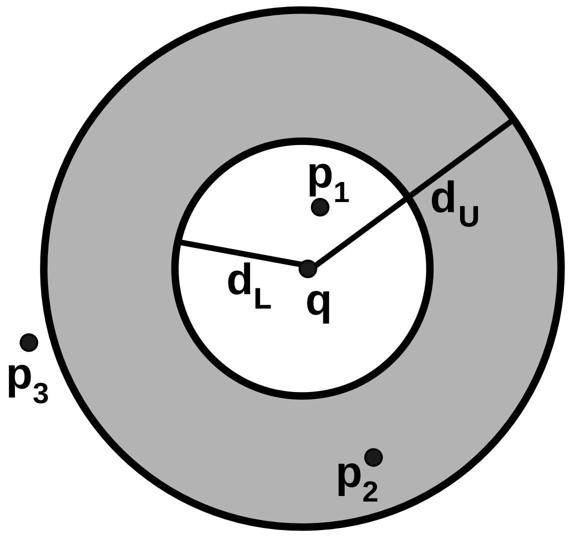


Figure 5.4: A Donut-Region kCNN Query

of such query is p_2 because p_1 does not satisfy the constraint. Such queries have many applications. For example a user might be interested in kNNs that are not nearer to him than 10km and not farther than 20km. Below we present the continuous monitoring of such queries.

Technique

The continuous monitoring of donut-region kCNN queries is exactly same as the continuous monitoring of kNN queries with the only difference that the distance of any object p that lies outside the range is considered infinity and the starting radius of CircularTrip is set as d_L. Consider the example of Fig. 5.5, where a 1-CNN query q is issued with distance bounded by d_L and d_U. The algorithm starts by calling the CircularTrip with radius set as d_L and then increases it by δ everytime unless kNNs are found. Algorithm finds p_2 which is reported as answer. Algorithm visits only the cells that are shown shaded in Fig. 5.5. Note that, this is the minimal set of cells that is required to be visited in order to guarantee the correctness.

The continuous monitoring of kCNN is similar to the monitoring of simple kNN queries. Note that simple kNN queries can be considered a special case of donut-region

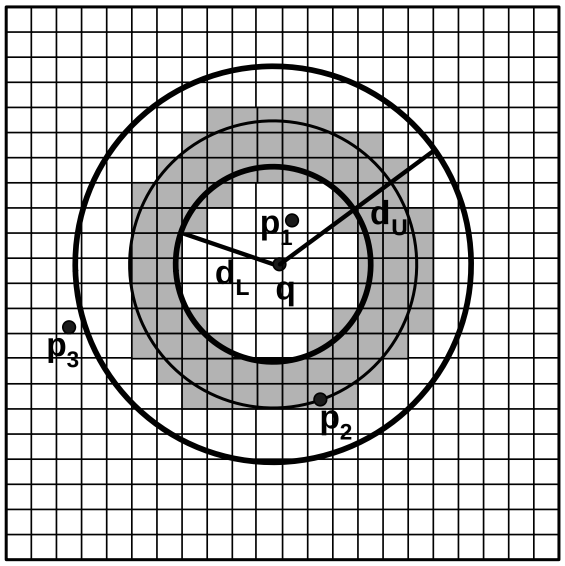


Figure 5.5: Computation of a Donut-Region 1-CNN Query

kCNN queries with $d_L = 0$ and $d_U = \infty$.

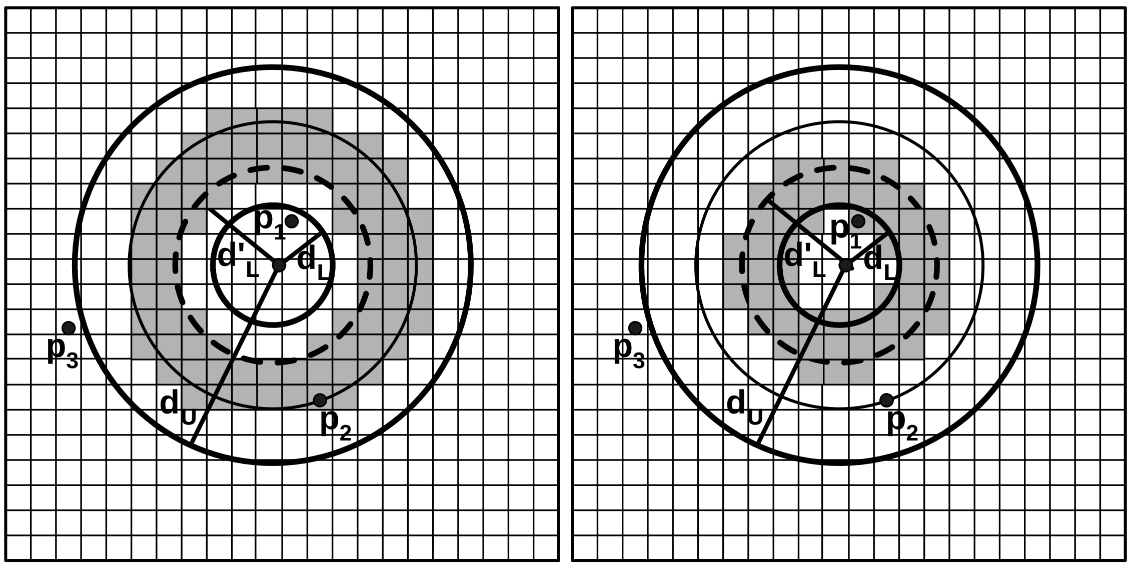
Extension to Varying Donut-Region Continuous kCNN Queries

To show the flexibility of our technique, in this section we present the approach to answer the continuous kCNN queries where the distance bounds d_U and d_L can be changed by user at any time. Consider the running example of Fig. 5.6, where the lower bound is changed from d'_L to d_L. Below we present the technique to efficiently compute the nearest neighbor after such change in constraints.

Suppose the lower distance bound is changed from d'_L to d_L and upper bound is changed from d'_U to d_U. Let $q.dist_k$ be the distance of current k^{th} NN from q. To update the results, first we delete the objects from $q.kNN$ that do not lie in the new range. More specifically we delete every $p \in q.kNN$ where $dist(p,q) < d_L$ or $dist(p,q) > d_U$. If $d_L < d'_L$, the algorithm starts calling CircularTrip with radius d_L and iteratively increases it by δ unless kNNs are found or radius equals or exceeds d'_L. If kNNs are not found and the radius equals to d'_L, the algorithm continues by calling CircularTrip with radius $q.dist_k$ unless kNNs are found or the radius equals to d_U. Note that, if radius equals to

d_U and kNNs have not been found this means there are less than k objects that lie in the constrained region. We illustrate our technique with following example.

Example Consider the example of Fig. 5.6(a), where the lower bound of the donut-region has been changed from d'_L to d_L. Since the current CNN, p_2, still lies in new constrained region we keep it in $q.kNN$. The algorithm starts calling CircularTrip with radius d_L and the object p_1 is found. p_1 is ignored because it does not lie in the constrained region. The algorithm iteratively calls CircularTrip with radius increased by δ everytime unless it equals d'_L. The algorithm should now continue by calling CircularTrip with $q.dist_k$ but because the distance of current CNN p_2 is $dist(p_2, q) = q.dist_k$ so algorithm stops and confirms p_2 as the nearest neighbor. During this update, the algorithm visits the shaded cell of Fig. 5.6(b).



(a) Lower bound changes from d'_L to d_L (b) p_2 is still the CNN

Figure 5.6: A Varying Donut-Region Continuous 1-CNN query

5.1.3 Continuous Donut-Pie kCNN Queries

In section 5.1.1, we presented our technique to continuously monitor the kCNN queries where the region is constrained by some angle range and in section 5.1.2 we discussed how we can use CircularTrip to answer continuous kCNN queries with the region constrained by some distance bound. In this section we discuss the kCNN queries in the region which is constrained by both the angle range and the distance.

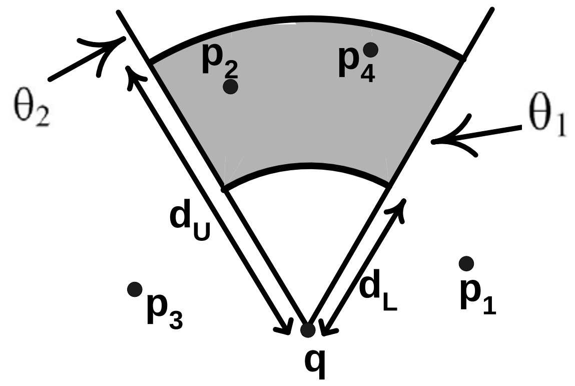


Figure 5.7: Donut-Pie kCNN Query

The user specifies an angle range $\langle\theta_1, \theta_2\rangle$ and the distance range d_L and d_U where d_L is the lower bound and d_U is the upper bound. The query q is to continuously monitor the kNNs in the region bounded by the defined distance and angle range constraints. The constrained region is shown shaded in Fig. 5.7.

Technique

Let $\langle\theta_1, \theta_2\rangle$ be the angle range and d_L and d_U be the lower and upper distance bounds, respectively. The algorithm starts calling ArcTrip with angle range $\langle\theta_1, \theta_2\rangle$ and radius set as d_L. ArcTrip is iteratively called with radius incremented by δ unless kNNs are found. Fig. 5.8 shows an example where p_2 is reported as CNN. The algorithm visits the shaded cells.

Continuous monitoring of such kCNN queries is similar to continuous monitoring of kNN queries with the difference that ArcTrip is called instead of CircularTrip and the distance of any object p that lies outside the constrained region is considered infinity.

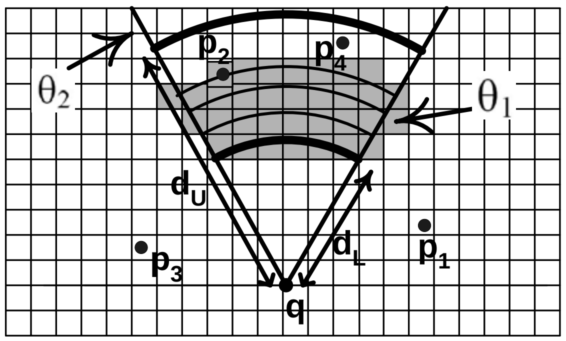
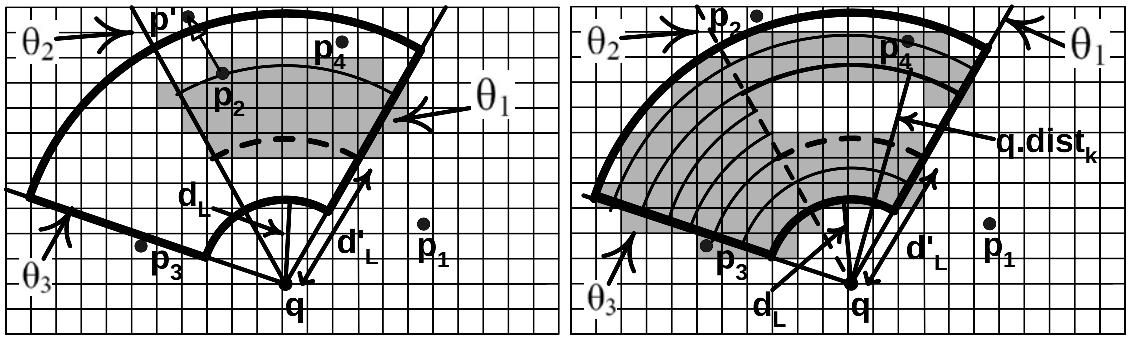


Figure 5.8: Computation of a Donut-Pie 1-CNN Query

Extension to Varying Donut-Pie Region Continuous kCNN Queries

In this section we will extend our approach to answer the queries where the user can change the constraints at any time. Consider the running example of Fig. 5.9(a) where p_2 is the CNN of q. The user changes the constrained region by changing the angle range and distance bounds. The angle range is changed from $\langle\theta_1, \theta_2\rangle$ to $\langle\theta_1, \theta_3\rangle$ and the lower distance bound has been changed from d'_L to d_L. Now the algorithm needs to monitor the nearest neighbors in a bigger constrained region (the closed shape shown with thick boundary). The idea behind updating the result of such queries is simply look in the area that was not looked before. ArcTrip with suitable angle range and radius can be used for this purpose. Below we give an extensive example that will illustrate the monitoring of such queries.



(a) constrained region and location of p_2 changes (b) p_4 is the new CNN

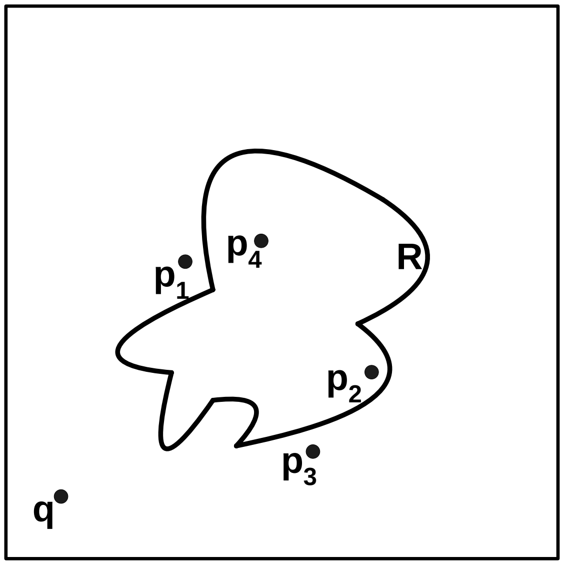
Figure 5.9: Computation of a Varying Donut-Pie Region 1-CNN Query

Example In Fig. 5.9(a), the angle range constraint is changed from $\langle\theta_1, \theta_2\rangle$ to $\langle\theta_1, \theta_3\rangle$ and the lower distance bound is changed from d'_L to d_L. The current CNN p_2 issues update to p'. Since it is an outgoing update so p_2 is deleted from $q.kNN$. Fig. 5.9(b) shows the computation of the new CNN in new constrained region. The algorithm starts by calling ArcTrip with radius d_L and angle range $\langle\theta_1, \theta_3\rangle$. The radius of ArcTrip is increased iteratively unless kNNs are found or when it reaches d'_L. Algorithm finds the object p_3 but it is ignored because it lies outside the constrained region. Now algorithm continues by calling ArcTrip with angle range $\langle\theta_2, \theta_3\rangle$ and radius set as d'_L. ArcTrip is iteratively called with increased radius in the same angle range unless the radius reaches $q.dist_k$. The angle range of ArcTrip is now again reset to $\langle\theta_1, \theta_3\rangle$. The algorithm continues and p_4 is found which is reported as answer. In order to update the result, the algorithm visits the shaded cells of Fig. 5.9(b) which form the minimal set of cells in order to guarantee the correctness.

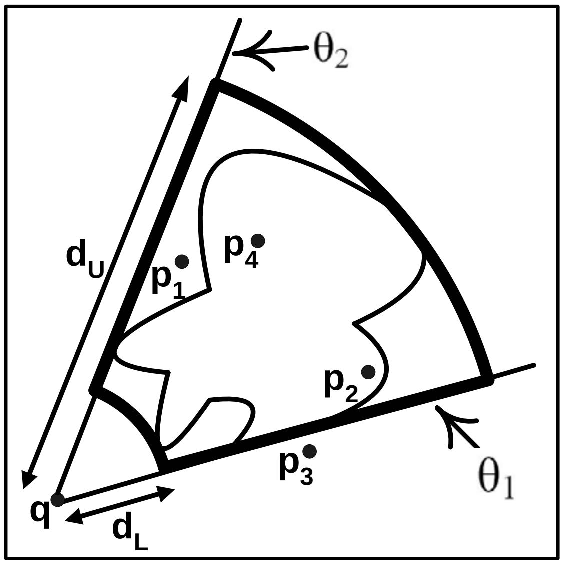
5.1.4 Continuous kCNN Queries Over Irregular Region

In this section we show that our technique can easily be extended to continuously monitor the kCNN queries over irregular region. Consider the irregular region R shown in Fig. 5.10(a). We can easily monitor the nearest neighbors to q in this irregular region R. Let d_L be the minimum distance of R from q and d_U be the maximum distance of R from q. Similarly, let $\langle\theta_1, \theta_2\rangle$ be the minimum angle range that fully covers R as shown in Fig. 5.10(b).

Continuous monitoring of kCNN query over this irregular region R is exactly same as continuous monitoring of a donut-pie region bounded by angle range $\langle\theta_1, \theta_2\rangle$ and distance bounds d_L and d_U as presented in section 5.1.3. The only difference is that the cells and points that lie outside R are ignored.



(a) The irregular region



(b) forming a donut-pie region kCNN query

Figure 5.10: Continuous kCNN Query Over Irregular Region R

5.1.5 Discussion

The continuous monitoring of constrained nearest neighbor queries have become important. Recently, algorithms have been proposed to continuously monitor reverse nearest neighbor queries [XZ06, KMS$^+$07] and each algorithm needs to continuously monitor nearest neighbor in constrained regions. We have shown that our novel access methods CircularTrip and ArcTrip can be used to continuously monitor various continuous kCNN queries. Our approach is flexible, efficient and can easily replace previous algorithms used for continuous monitoring of constrained nearest neighbor queries.

To show the flexibility of our approach we show that it can be extended to answer the continuous kCNN queries where the constrained region can be changed at any time and for all the algorithms presented the number of visited cells is minimum. To the best of our knowledge there is no existing approach that can efficiently answer such queries. Though CPM can be used to compute the initial results of few of such queries by considering only the cells and rectangles that lie in the constrained region but the rectangle based partitioning of CPM is not specifically designed to prioritize cells based on their angle from q so its performance becomes poor for continuous monitoring of such queries and even worse for varying constrained region CNN queries.

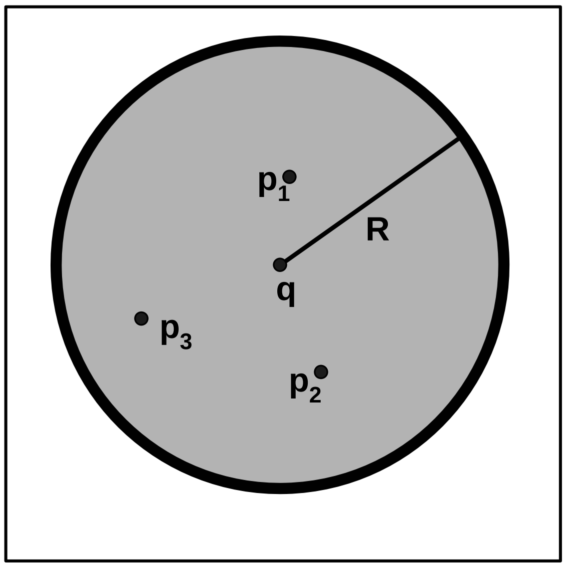
5.2 Farthest Neighbor Queries

Given a set of points P and a query point q, a k farthest neighbor query is to find a result set R which contains k objects such that for any $p \in (P - R)$ and for any $p' \in R$, $dist(p', q) \geq dist(p, q)$. The continuous monitoring of k farthest neighbors (kFN) is to continuously update the results affected by the movement of query point q and data points $p \in P$.

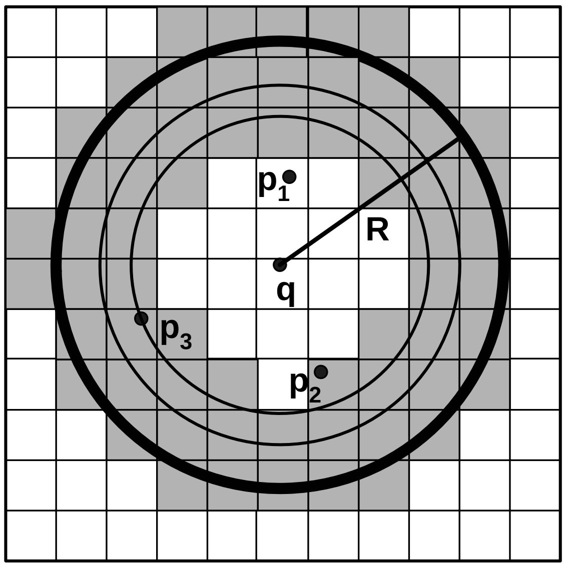
Farthest neighbor queries have various applications. A farthest neighbor query q determines the minimum radius of the circle centered at q which will cover all the data points in P. Similarly, a k farthest neighbors query can be issued to find the minimum radius of the circle around q so that only $k - 1$ points lie outside the circle. Consider another application where a team of commandos is on a mission and the leader wants that no member of the team is more than 10km away from him. Clearly, the members who are farther from him needs more of his attention. He might be interested in k farthest team members so that he can monitor their activities and can advise them not to move too far from him.

5.2.1 Initial Computation

Initial computation of k farthest neighbors is very similar to the computation of nearest neighbors. The algorithm initially calls CircularTrip with radius set as R where R is the maximum distance of q from the region in which the farthest neighbors are to be found. As opposed to nearest neighbor queries, the cells returned by CircularTrip are inserted in a heap according to their $maxdist$ from q. The cells are visited in descending order of their $maxdist$ and the radius of CircularTrip is decreased by δ everytime. Let $q.dist_k$ be the distance between q and k^{th} farthest neighbor found so far, the algorithm stops when the radius of CircularTrip becomes smaller than or equal to $q.dist_k$.



(a) A farthest neighbor query



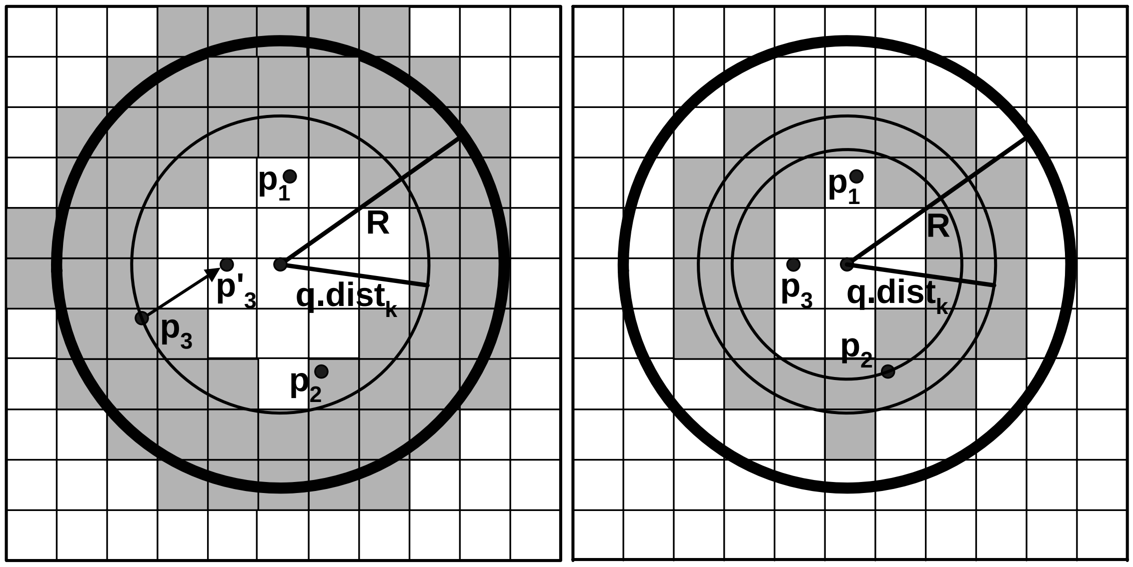
(b) p_3 is the farthest neighbor

Figure 5.11: Computation of a Farthest Nearest Neighbor Query

Example Consider a query q shown in Fig. 5.11(a) where a farthest neighbor is to be found in the shaded area. Algorithm first makes a CircularTrip with radius R and visits the returned cells in descending order of their *maxdist* but no object is found. Next time the algorithm calls CircularTrip with radius $R - \delta$ and finds object p_3. Finally, CircularTrip is called with radius set as $q.dist_k = dist(p_3, q)$ and the object p_2 is found. The newly found object p_2 is not the farthest neighbor because $dist(p_2, q) < dist(p_3, q)$. The algorithm terminates and reports p_3 as the farthest neighbor. The shaded cells of Fig. 5.11(b) are visited during the computation.

5.2.2 Continuous Monitoring

Continuous monitoring of a k farthest neighbors is also very similar to that of a k nearest neighbors query. Let $q.kFN$ be the set of k farthest neighbors of q. In monitoring of k farthest neighbors, an update $< p.id, p_{pre}, p_{cur} >$ is considered outgoing update if $dist(p_{pre}, q) \geq q.dist_k$ and $dist(p_{cur}, q) < q.dist_k$. Similarly, an update is considered incoming if $dist(p_{pre}, q) < q.dist_k$ and $dist(p_{cur}, q) > q.dist_k$. During update handling, the algorithm deletes an object p from $q.kFN$ if it issues an outgoing update and inserts it in $q.kFN$ if it issues an incoming update. After handling all the updates, if $q.kFN$ contains more than or equal to k objects, the algorithm keeps k farthest objects and deletes all other. On the other hand, if $q.kFN$ contains less than k objects, the algorithm updates the result by calling CircularTrip with radius $q.dist_k$ and iteratively decreasing it by δ unless k farthest neighbors have been found.



(a) p_3 issues an update at p'_3 (b) p_2 is new farthest neighbor

Figure 5.12: Update Handling of a Farthest Neighbor Query

Example Consider the running example of Fig. 5.12(a) where the current farthest neighbor p_3 issues an update at location p'_3. It is an outgoing update because $dist(p_3, q) \geq q.dist_k$ and $dist(p'_3, q) < q.dist_k$, so the object p_3 is deleted from $q.kFN$. In order to update the result, the algorithm starts calling CircularTrip with radius set equal to $q.dist_k$ as shown in Fig. 5.12(b). The cells returned by CircularTrip are visited in descending order of their $maxdist$ and the object p_2 is found. p_2 is inserted in $q.kFN$ and $q.dist_k$ is updated to $dist(p_2, q)$. Finally, CircularTrip is called with radius set to $dist(p_2, q)$ and all the returned cells are visited. Since no new object is found, p_2 is returned as result. The algorithm visits the shaded cell of Fig. 5.12(b) and the result cannot be guaranteed if any of the cells is missed.

5.2.3 Discussion

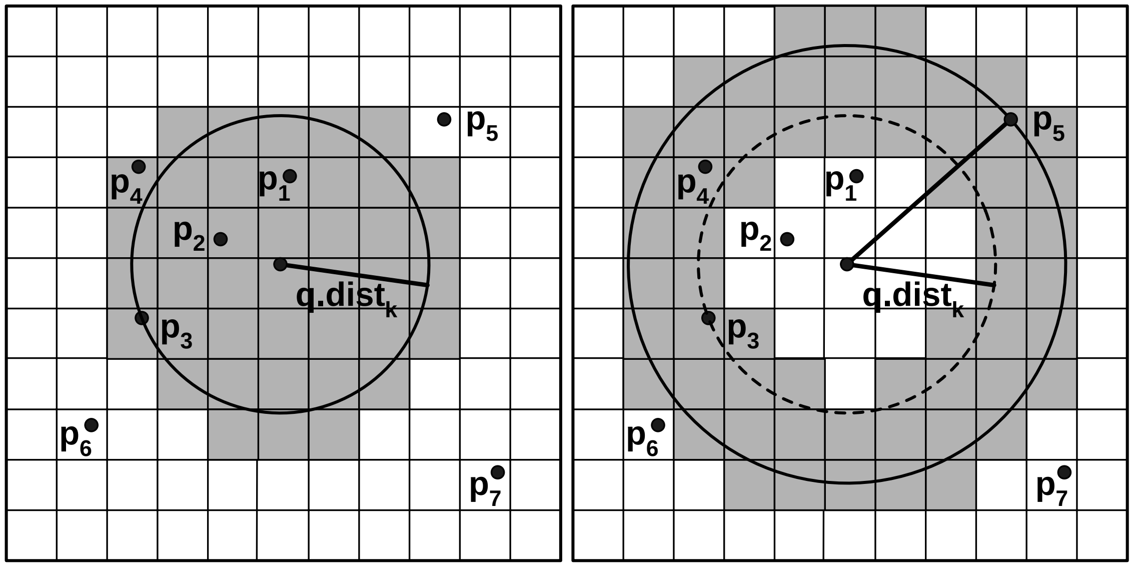
To the best of our knowledge there is no efficient approach that can continuously monitor farthest neighbor queries. SEA-CNN and YPK-CNN are very inefficient because they are designed only to find the nearest neighbors of some query points. Extension of CPM to farthest neighbor queries is non-trivial. Moreover, a rectangle gives a better approximation only for smaller circles. As the size of circle grows, the approximation yielded by rectangle become worse. For this reason, CPM's approximation of proximity of cells to q becomes worse once the rectangles becomes farther from query point which is the case for farthest neighbor queries. On the other hand, our grid access methods can easily monitor farthest neighbor queries without any performance degradation. This shows the effectiveness of our presented grid access methods.

5.3 $(k+m)$ NN Queries

At any time a kNN query can be updated by the user to continuously monitor $(k+m)$ nearest neighbors instead of k nearest neighbors of q where $m > -k$. The queries that support such updates on the number of nearest neighbors are called $(k+m)$NNs queries. It gives flexibility to the user that he can change, at any time, the number of nearest neighbors he wants to continuously monitor. Below we show that by using our approach such kind of queries can be efficiently answered without visiting any unnecessary cell.

5.3.1 Technique

Consider a kNN query has been updated to $(k+m)$NN query where k nearest neighbors of the query are already known. If $0 > m > -k$, then the update of the results is trivial. We only keep $(k+m)$ closest objects among $q.kNN$ and delete all other objects. If $m > 0$, then the results are updated as follows. Let $q.dist_k$ be the distance of k^{th} nearest neighbor of the query q. The algorithm starts calling CircularTrip with radius set equal to $q.dist_k$ and iteratively increases it by δ unless all $(k+m)$ nearest neighbors have been found.



(a) kNN query $(k = 3)$ (b) query updated to $(k+m)$NN query $(m = 2)$

Figure 5.13: A $(k+m)$NNs Query $(k = 3,\ m = 2)$

Example Consider the example of Fig. 5.13 where a user who was monitoring k $(= 3)$ nearest neighbors changes the query so that it now monitors $(k + m = 5)$ nearest neighbors. Fig. 5.13(a) shows $q.dist_k$ that corresponds to the distance of 3^{rd} nearest neighbor from q. In order to update the results such that it includes $(k + m)$ NNs, the algorithm calls CircularTrip with radius set as $q.dist_k = dist(p_3, q)$ and iteratively increases it and p_4 and p_5 are found and reported as 4^{th} and 5^{th} nearest neighbors of q, respectively. In order to update the kNN query to the $(k+m)$NN query, the algorithm has visited the shaded cells of Fig. 5.13(b).

5.3.2 Discussion

To the best of our knowledge, there is no existing approach that can efficiently update a kNN query to a $(k + m)$NN query. Even though CPM stores extra book-keeping information (*visit list* and *heap*), it cannot efficiently update such query. CPM will need to first visit all the cells in *visit list* and than continue with the *heap* in order to find $(k+m)$

NNs. Note that, CPM will visit the same number of cells that it visits to compute the initial result of a newly issued kNN ($k = k + m$) query. i.e; it does not utilize the information of previously known k nearest neighbors. Our algorithm, in contrast, utilizes previous information efficiently and answers the query by visiting minimum number of cells.

5.4 Conclusion

In this chapter, we present techniques for several variants of k nearest neighbor queries. We believe that CircularTrip and ArcTrip can be used to continuously monitor many other variants of nearest neighbor queries (e.g., nearest surrounder queries [LLL06] and aggregate nearest neighbor queries). Though CPM can be used to answer some of these queries but it is not efficient because CPM was designed only for k nearest neighbor queries. Our CircularTrip-based algorithm performs 2 to 4 times better than CPM even for kNN queries and it is easy to verify that for the variants presented in this chapter, our presented approaches are still optimal in accessing the number of cells. Hence the performance of our approaches is expected to be much faster than all previous approaches.

Moreover, it is easy to see that our approach can answer any query that is a hybrid of above mentioned queries. For example, we can use ArcTrip to continuously monitor $(k + m)$ farthest neighbors in a Donut-Pie constrained region where the constraints that define the region may be changed by the user at any time. For all these queries, our proposed algorithms visit minimum number of cells.

Chapter 6

Conclusion

We presented two grid traversal methods named CircularTrip and ArcTrip that return the cells based on their proximty to the query point. CircularTrip returns all the cells around query point q that intersect the circle with center at q and radius r. ArcTrip goes one step further and returns only the cells that intersect the circle of radius r and lie within some specific angle range. We have shown that the access methods are efficient and flexible. More specifically, each access method computes the *mindist* of $\mid C \mid$ number of cells where $\mid C \mid$ is the number of cells returned by the access method.

We show the effectiveness of our grid access methods by introducing an algorithm for continuous monitoring of kNN queries. Our experimental results show that our CircularTrip-based algorithm is 2 to 4 times faster than CPM which is previously best known algorithm. We prove that our algorithm accesses minimum number of cells for any continuous k nearest neighbor query. Moreover, our algorithm uses less memory space. Unlike CPM, CircularTrip-based algorithm does not need any book-keeping information and still achieves a better performance. The space usage of our algorithm is 50% to 85% of CPM's space requirements.

We also proposed algorithms for continuously monitoring other variants of kNN queries like constrained nearest neighbor queries, farthest neighbor queries and $(k+m)$-NN queries. All previous approaches fail to efficiently monitor such queries. Our algorithms are very flexible, efficient and can easily be integrated to answer more complex queries. For example, our proposed algorithms can be easily used to answer $(k + m)$ farthest neighbors in a constrained region where the spatial conditions that constrain the region can be changed by the user at any time. Moreover, all of the presented algorithms preserve the basic properties of visiting minimum number of cells for each continuous query and no book-keeping information is required.

Bibliography

[ABK$^+$06] Elke Achtert, Christian Bohm, Peer Kroger, Peter Kunath, Alexey Pryakhin and Matthias Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, Seiten 515–526, New York, NY, USA, 2006. ACM Press.

[AM93] Arya and Mount. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1993.

[AMN$^+$98] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[Ber93] Marshall Bern. Approximate closest-point queries in high dimensions. *Inf. Process. Lett.*, 45(2):95–99, 1993.

[Bia69] T. Bially. Space-Filling Curves: Their Generation and Their Application to Bandwidth Reduction. *IEEE Transactions on Information Theory*, 15(6):658–664, 1969.

[BJKS02] R. Benetis, C. Jensen, G. Karciauskas and S. Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects, 2002.

[BK04] Christian Böhm and Florian Krebs. The k-Nearest Neighbour Join: Turbo Charging the KDD Process. *Knowl. Inf. Syst.*, 6(6):728–749, 2004.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel and Bernhard Seeger. Efficient processing of spatial joins using R-trees. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, Seiten 237–246, New York, NY, USA, 1993. ACM Press.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, Seiten 322–331, New York, NY, USA, 1990. ACM Press.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1:173–189, 1972.

[Bri02] Thomas Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2):153–180, 2002.

[CC05] Hyung-Ju Cho and Chin-Wan Chung. An efficient and scalable approach to CNN queries in a road network. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, Seiten 865–876. VLDB Endowment, 2005.

[CG99] Surajit Chaudhuri and Luis Gravano. Evaluating Top-k Selection Queries. In *VLDB'99*, Seiten 397–410, 1999.

[CHC04] Y. Cai, K. Hua and G. Cao. Processing Range-Monitoring Queries on Heterogeneous Mobile Objects, 2004.

[CMTV00] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis and Michael Vassilakopoulos. Closest pair queries in spatial databases. In *Proceedings 1994 ACM SIGMOD Conference, Dallas, TX*, Seiten 189–200, 2000.

[CMTV04] A. Corral, Y. Manolopoulos, Y. Theodoridis and M. Vassilakopoulos. Algorithms for Processing K-closest-pair Queries in Spatial Databases, 2004.

[CP07] Yun Chen and Jignesh M. Patel. Efficient Evaluation of All-Nearest-Neighbor Queries. To appear in ICDE, 2007.

[CYL07] Muhammad Aamir Cheema, Yidong Yuan and Xuemin Lin. CircularTrip: An Effective Algorithm for Continuous kNN Queries. To appear in the 12th International Conference on Database Sytems for Advanced Applications (DASFAA), 2007.

[FNPS79] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger and H. Raymond Strong. Extendible hashing- a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

[FSAA01] Hakan Ferhatosmanoglu, Ioanna Stanoi, Divyakant Agrawal and Amr El Abbadi. Constrained Nearest Neighbor Queries. *Lecture Notes in Computer Science*, 2121:257–??, 2001.

[GG98] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[GL04] B. Gedik and L. Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System, 2004.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, Seiten 47–57, New York, NY, USA, 1984. ACM Press.

[Gut94] Ralf Hartmut Guting. An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399, 1994.

[Hen94] Andreas Henrich. A Distance Scan Algorithm for Spatial Access Structures. In *ACM-GIS*, Seiten 136–143, 1994.

[HLX06] Haibo Hu, Dik Lun Lee and Jianliang Xu. Fast Nearest Neighbor Search on Road Networks. In *EDBT*, Seiten 186–203, 2006.

[Hon04] Chenyi Xia Hongjun. GORDER: An Efficient Method for KNN Join Processing, 2004.

[HS98] Gísli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. Seiten 237–248, 1998.

[HS99] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, 1999.

[HXL05] Haibo Hu, Jianliang Xu and Dik Lun Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *SIGMOD*, Seiten 479–490, 2005.

[ISS03] Glenn S. Iwerks, Hanan Samet and Kenneth P. Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In *VLDB*, Seiten 512–523, 2003.

[JKPT03] Christian S. Jensen, Jan Kolarvr, Torben Bach Pedersen and Igor Timko. Nearest neighbor queries in road networks. In *GIS '03: Proceedings of the 11th ACM international symposium on Advances in geographic information systems*, Seiten 1–8, New York, NY, USA, 2003. ACM Press.

[KGT99] George Kollios, Dimitrios Gunopulos and Vassilis J. Tsotras. Nearest Neighbor Queries in a Mobile Environment. In *Spatio-Temporal Database Management*, Seiten 119–134, 1999.

[KM00] Flip Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. Seiten 201–212, 2000.

[KMS$^+$07] James Kang, Mohamed Mokbel, Shashi Shekhar, Tian Xia and Donghui Zhang. Continuous Evaluation of Monochromatic and Bichromatic Reverse Nearest Neighbors. To appear in ICDE, 2007.

[KOR98] Eyal Kushilevitz, Rafail Ostrovsky and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. Seiten 614–623, 1998.

[KOTZ04] N. Koudas, B. Ooi, K. Tan and R. Zhang. Approximate NN queries on streams with guaranteed error /performance bounds, 2004.

[KPH04] Dmitri V. Kalashnikov, Sunil Prabhakar and Susanne E. Hambrusch. Main Memory Evaluation of Monitoring Queries Over Moving Objects. *Distrib. Parallel Databases*, 15(2):117–135, 2004.

[KS97] Norio Katayama and Shin'ichi Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. Seiten 369–380, 1997.

[KS04a] Mohammad Kolahdouzan and Cyrus Shahabi. Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases, 2004.

[KS04b] Mohammad R. Kolahdouzan and Cyrus Shahabi. Continuous K-Nearest Neighbor Queries in Spatial Network Databases. In *STDBM*, Seiten 33–40, 2004.

[KSF$^+$96] Flip Korn, Nikolaos Sidiropoulos, Christos Faloutsos, Eliot Siegel and Zenon Protopapas. Fast Nearest Neighbor Search in Medical Image Databases. In *The VLDB Journal*, Seiten 215–226, 1996.

[LCGMW02] C. Li, E. Chang, H. Garcia-Molina and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces, 2002.

[LHJ$^+$03] M. Lee, W. Hsu, C. Jensen, B. Cui and K. Teo. Supporting frequent updates in R-Trees: A bottom-up approach, 2003.

[LLHH05] Hongga Li, Hua Lu, Bo Huang and Zhiyong Huang. Two ellipse-based pruning methods for group nearest neighbor queries. In *GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems*, Seiten 192–199, New York, NY, USA, 2005. ACM Press.

[LLL06] Ken C. K. Lee, Wang-Chien Lee and Hong Va Leong. Nearest Surrounder Queries. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, Seite 85, Washington, DC, USA, 2006. IEEE Computer Society.

[LNY03] King-Ip Lin, Michael Nolen and Congjun Yang. Applying Bulk Insertion Techniques for Dynamic Reverse Nearest Neighbor Problems. *ideas*, 00:290, 2003.

[MHP05] Kyriakos Mouratidis, Marios Hadjieleftheriou and Dimitris Papadias. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *SIGMOD*, Seiten 634–645, 2005.

[MPBT05] Kyriakos Mouratidis, Dimitris Papadias, Spiridon Bakiras and Yufei Tao. A Threshold-Based Algorithm for Continuous Monitoring of k Nearest Neighbors. *IEEE Transactions on Knowledge and Data Engineering*, 17(11):1451–1464, 2005.

[MYPM06] Kyriakos Mouratidis, Man Lung Yiu, Dimitris Papadias and Nikos Mamoulis. Continuous nearest neighbor monitoring in road networks. In *VLDB'2006: Proceedings of the 32nd international conference on Very large data bases*, Seiten 43–54. VLDB Endowment, 2006.

[MZ02] J. Maheshwari and N. Zeh. On reverse nearest neighbor queries, 2002.

[NHS84] J. Nievergelt, Hans Hinterberger and Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[NO97] K. Nakano and S. Olariu. An Optimal Algorithm for the Angle-Restricted All Nearest Neighbor Problem on the Reconfigurable Mesh, with Applications:. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):983–990, 1997.

[PM97] Apostolos Papadopoulos and Yannis Manolopoulos. Performance of Nearest Neighbor Queries in R-Trees. In *ICDT '97: Proceedings of the 6th International Conference on Database Theory*, Seiten 394–408, London, UK, 1997. Springer-Verlag.

[PSTM04] Dimitris Papadias, Qiongmao Shen, Yufei Tao and Kyriakos Mouratidis. Group Nearest Neighbor Queries. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, Seite 301, Washington, DC, USA, 2004. IEEE Computer Society.

[PXK$^+$02] S. Prabhakar, Y. Xia, D. Kalashnikov, Walid G. Aref and S. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques For Continuous Queries on Moving Objects, 2002.

[PZMT03] D. Papadias, J. Zhang, N. Mamoulis and Y. Tao. Query Processing in Spatial Network Databases, 2003.

[RKV95] Nick Roussopoulos, Stephen Kelley and Frédéric Vincent. Nearest neighbor queries. Seiten 71–79, 1995.

[RPM03] K. Raptopoulou, A. N. Papadopoulos and Y. Manolopoulos. Fast Nearest-Neighbor Query Processing in Moving-Object Databases. *Geoinformatica*, 7(2):113–137, 2003.

[SAA00] Ioana Stanoi, Divyakant Agrawal and Amr El Abbadi. Reverse Nearest Neighbor Queries for Dynamic Databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, Seiten 44–53, 2000.

[SFT03] A. Singh, H. Ferhatosmanoglu and A. Tosun. High Dimensional Reverse Nearest Neighbor Queries, 2003.

[SJLL00] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD Conference*, Seiten 331–342, 2000.

[SK98] Thomas Seidl and Hans-Peter Kriegel. Optimal Multi-Step k-Nearest Neighbor Search. In *SIGMOD Conference*, Seiten 154–165, 1998.

[SKS02] Cyrus Shahabi, Mohammad R. Kolahdouzan and Mehdi Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. In *ACM-GIS*, Seiten 94–10, 2002.

[SR01] Zhexuan Song and Nick Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, Seiten 79–96, London, UK, 2001. Springer-Verlag.

[SRAA01] Ioana Stanoi, Mirek Riedewald, Divyakant Agrawal and Amr El Abbadi. Discovery of Influence Sets in Frequently Updated Databases. In *The VLDB Journal*, Seiten 99–108, 2001.

[SSH86] Michael Stonebraker, Timos K. Sellis and Eric N. Hanson. An Analysis of Rule Indexing Implementations in Data Base Systems. In *Expert Database Conf.*, Seiten 465–476, 1986.

[SWCD97] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain and Son Dao. Modeling and Querying Moving Objects. In *ICDE*, Seiten 422–432, 1997.

[TFPL04] Yufei Tao, Christos Faloutsos, Dimitris Papadias and Bin Liu. Prediction and indexing of moving objects with unknown motion patterns. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, Seiten 611–622, New York, NY, USA, 2004. ACM Press.

[TP02] Yufei Tao and Dimitris Papadias. Time-Parameterized Queries in Spatio-Temporal Databases. In *SIGMOD Conference*, 2002.

[TPL04] Yufei Tao, Dimitris Papadias and Xiang Lian. Reverse kNN Search in Arbitrary Dimensionality. In *VLDB*, Seiten 744–755, 2004.

[TPS02] Yufei Tao, Dimitris Papadias and Qiongmao Shen. Continuous Nearest Neighbor Search. In *VLDB*, Seiten 287–298, 2002.

[TPS03] Yufei Tao, Dimitris Papadias and Jimeng Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries, 2003.

[TYM06] Yufei Tao, Man Lung Yiu and Nikos Mamoulis. Reverse Nearest Neighbor Search in Metric Spaces. *IEEE Transactions on Knowledge and Data Engineering*, 18(9):1239–1252, 2006.

[XMA$^+$04] Xiaopeng Xiong, Mohamed F. Mokbel, Walid G. Aref, Susanne E. Hambrusch and Sunil Prabhakar. Scalable Spatio-temporal Continuous Query Processing for Location-Aware Services, 2004.

[XMA05] Xiaopeng Xiong, Mohamed F. Mokbel and Walid G. Aref. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In *ICDE*, Seiten 643–654, 2005.

[XZ06] Tian Xia and Donghui Zhang. Continuous Reverse Nearest Neighbor Monitoring. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, Seite 77, Washington, DC, USA, 2006. IEEE Computer Society.

[XZKD05] Tian Xia, Donghui Zhang, Evangelos Kanoulas and Yang Du. On computing top-t most influential spatial sites. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, Seiten 946–957. VLDB Endowment, 2005.

[YL01] Congjun Yang and King-Ip Lin. An Index Structure for Efficient Reverse Nearest Neighbor Queries. In *Proceedings of the 17th International Conference on Data Engineering*, Seiten 485–492, Washington, DC, USA, 2001. IEEE Computer Society.

[YMP05] Man Lung Yiu, Nikos Mamoulis and Dimitris Papadias. Aggregate Nearest Neighbor Queries in Road Networks. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):820–833, 2005.

[YPK05] Xiaohui Yu, Ken Q. Pu and Nick Koudas. Monitoring K-Nearest Neighbor Queries Over Moving Objects. In *ICDE*, Seiten 631–642, 2005.

[YPMT05] Man Lung Yiu, Dimitris Papadias, Nikos Mamoulis and Yufei Tao. Reverse Nearest Neighbors in Large Graphs. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, Seiten 186–187, Washington, DC, USA, 2005. IEEE Computer Society.

[ZPMT04] Jun Zhang, Dimitris Papdias, Nikos Mamoulis and Yufei Tao. All-Nearest-Neighbors Queries in Spatial Databases. In *SSDBM '04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, Seite 297, Washington, DC, USA, 2004. IEEE Computer Society.

[ZZP$^+$03] J. Zhang, M. Zhu, D. Papadias, Y. Tao and D. Lee. Location-based spatial queries, 2003.