Continuous Nearest Neighbor Search

Yufei Tao Dimitris Papadias Qiongmao Shen

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{taoyf, dimitris, qmshen}@cs.ust.hk

Abstract

A continuous nearest neighbor query retrieves the nearest neighbor (NN) of every point on a line segment (e.g., "find all my nearest gas stations during my route from point *s* to point *e*"). The result contains a set of <*point, interval*> tuples, such that *point* is the NN of all points in the corresponding *interval*. Existing methods for continuous nearest neighbor search are based on the repetitive application of simple NN algorithms, which incurs significant overhead. In this paper we propose techniques that solve the problem by performing a single query for the whole input segment. As a result the cost, depending on the query and dataset characteristics, may drop by orders of magnitude. In addition, we propose analytical models for the expected size of the output, as well as, the cost of query processing, and extend out techniques to several variations of the problem.

1. Introduction

Let P be a dataset of points in multi-dimensional space. A continuous nearest neighbor (CNN) query retrieves the nearest neighbor (NN) of every point in a line segment $q = [s, e]$. In particular, the result contains a set of <*R,T*> tuples, where R (for result) is a point of P, and T is the interval during which R is the NN of q. As an example consider Figure 1.1, where $P=\{a,b,c,d,f,g,h\}$. The output of the query is $\{<a, [s,s_1]>, <c, [s_1,s_2]>, <f, [s_2,s_3]>, <h, [s_3,e]>\}$, meaning that point a is the NN for interval $[s,s_1]$; then at s_1, point c becomes the NN etc. The points of the query segment (i.e., s_1, s_2, s_3) where there is a change of neighborhood are called *split points*. Variations of the problem include the retrieval of k neighbors (e.g., find the three NN for every point in q), datasets of extended objects (e.g., the elements of P are rectangles instead of points), and situations where the query input is an arbitrary trajectory (instead of a line segment).

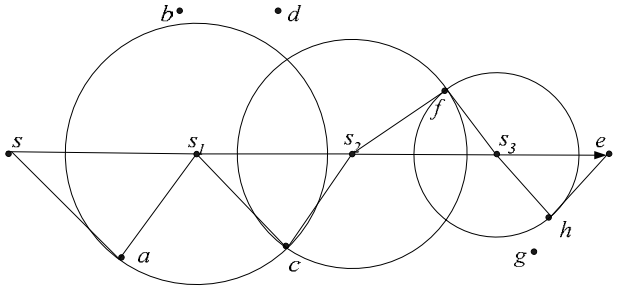


Figure 1.1: Example query

CNN queries are essential for several applications such as location-based commerce ("if I continue moving towards this direction, which will be my closest restaurants for the next 10 minutes?") and geographic information systems ("which will be my nearest gas station at any point during my route from city *A* to city *B*"). Furthermore, they constitute an interesting and intuitive problem from the research point of view. Nevertheless, there is limited previous work in the literature.

From the computational geometry perspective, to the best of our knowledge, the only related problem that has been addressed is that of finding the single NN for the whole line segment [BS99] (e.g., point f for the query segment in Figure 1.1). On the other hand, research in databases (with a few exceptions discussed in the next section) has focused on other variations of NN search in secondary memory. These include kNN for point queries [RKV95, HS99] (e.g., find the three NN of a point q in P), and closest pair queries [HS98, CMTV00] (e.g., find the k closest pairs $<p_i, p_j>$ from two datasets P_1 and P_2, where $p_i \in P_1$ and $p_j \in P_2$).

In this paper we first deal with continuous 1NN queries (retrieval of single neighbors when the query input is a line segment, i.e., the example of Figure 1.1), identifying and proving some properties that facilitate the development of efficient algorithms. Then we propose

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

query processing methods using R-trees as the underlying data structure. Furthermore, we present an analytical comparison with existing methods, proposing models that estimate the number of split points and processing costs. Finally we extend our methods to multiple nearest neighbors and arbitrary inputs (i.e., consisting of several consecutive segments).

The rest of the paper is structured as follows: Section 2 outlines existing methods for processing NN and CNN queries, and Section 3 describes the definitions and problem characteristics. Section 4 proposes an efficient algorithm for R-trees, while Section 5 contains the analytical models. Section 6 discusses extensions to related problems and Section 7 experimentally evaluates our techniques with real datasets. In Section 8 we conclude the paper with directions for future work.

2. Related Work

Like most previous work in the relevant literature, we employ R-trees [G84, SRF87, BKSS90] due to their efficiency and popularity. Our methods, however, are applicable to any data-partition access method. Figure 2.1 shows an example R-tree for point set $P=\{a,\ldots,m\}$ assuming a capacity of three entries per node. Points that are close in space (e.g., *a*, *b*, *c*) are clustered in the same leaf node (N_3). Nodes are then recursively grouped together with the same principle until the top level, which consists of a single root.

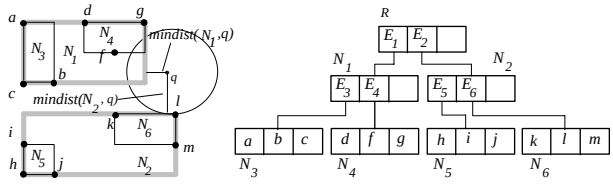


Figure 2.1: R-tree and point-NN example

The most common type of nearest neighbor search is the *point-kNN* query that finds the *k* objects from a dataset *P* that are closest to a query point *q*. Existing algorithms search the R-tree of *P* in a branch-and-bound manner. For instance, Roussopoulos et al [RKV95] propose a depth-first method that, starting from the root of the tree, visits the entry with the minimum distance from *q* (e.g., entry E_1 in Figure 2.1). The process is repeated recursively until the leaf level (node N_4), where the first potential nearest neighbor is found (*f*). During backtracking to the upper level (node N_1), the algorithm only visits entries whose minimum distance is smaller than the distance of the nearest neighbor already found. In the example of Figure 2.1, after discovering *f*, the algorithm will backtrack to the root level (without visiting N_3), and then follow the path N_2, N_6 where the actual NN *l* is found.

Another approach [HS99] implements a best-first traversal that follows the entry with the smallest distance among all those visited. In order to achieve this, the algorithm keeps a heap with the candidate entries and their minimum distances from the query point. In the previous example, after visiting node N_1, best-first traversal will follow the path N_2, N_6 and directly discover *l* (i.e., without first finding other potential NN, such as *f*). Although this method is optimal in the sense that it only visits the necessary nodes for obtaining the NN, it suffers from buffer thrashing if the heap becomes larger than the available memory.

Conventional NN search (i.e., point queries) and its variations in low and high dimensional spaces have received considerable attention during the last few years (e.g., [KSF+96, SK98, WSB98, YOTJ01]) due to their applicability in domains such as content based retrieval and similarity search. With the proliferation of location-based e-commerce and mobile computing, continuous NN search promises to gain similar importance in the research and applications communities. Sistla et al. were the first ones to identify the significance of CNN in spatiotemporal database systems. In [SWCD97], they describe modeling methods and query languages for the expression of such queries, but do not discuss access or processing methods.

The first algorithm for CNN query processing, proposed in [SR01], employs sampling to compute the result. In particular, several point-NN queries (using an R-tree on the point set *P*) are repeatedly performed at predefined sample points of the query line, using the results at previous sample points to obtain tight search bounds. This approach suffers from the usual drawbacks of sampling, i.e., if the sampling rate is low the results will be incorrect; otherwise, there is a significant computational overhead. In any case there is no accuracy guarantee, since even a high sampling rate may miss some split points (i.e., if the sample does not include points s_1, s_2, s_3 in Figure 1.1).

A technique that does not incur false misses is based on the concept of *time-parameterized* (TP) queries [TP02]. The output of a TP query has the general form $<R, T, C>$, where *R* is current result of the query (the methodology applies to general spatial queries), *T* is the validity period of *R*, and *C* the set of objects that will affect *R* at the end of *T*. From the current result *R*, and the set of objects *C* that will cause changes, we can incrementally compute the next result. We refer to *R* as the *conventional*, and (*T*,*C*) as the *time-parameterized* component of the query.

Figures 2.2 and 2.3 illustrate how the problem of Figure 1.1 can be processed using TP NN queries. Initially a point-NN query is performed at the starting point (*s*) to retrieve the first nearest neighbor (*a*). Then, the *influence point* s_x of each object *x* in the dataset *P* is computed as the point where *x* will start to get closer to the line segment than the current NN. Figure 2.2 shows the influence points after the retrieval of *a*. Some of the points (e.g., *b*) will never influence the result, meaning that they will never come closer to [*s*,*e*] than *a*. Identifying the influencing point (s_c) that will change the

result (rendering *c* as the next neighbor) can be thought of as a conventional NN query, where the goal is to find the point *x* with the minimum $dist(s,s_x)$. Thus, traditional point-NN algorithms (e.g., [RKV95]) can be applied with appropriate transformations (for details see [TP02]).

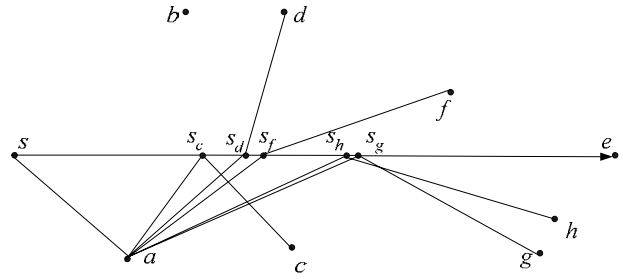


Figure 2.2: CNN processing using TP queries – first step

After the first step, the output of the TP query is $<a, [s,s_c), c>$, meaning that *a* is the NN until s_c, at which point *c* becomes the next NN (s_c corresponds to the first split point s_1 in Figure 1.1). In order to complete the result, we perform repeated retrievals of the TP component. For example, at the second step we find the next NN by computing again the influencing points with respect to *c* (see Figure 2.3). In this case only points *f, g* and *h* may affect the result, and the first one (*f*) becomes the next neighbor.

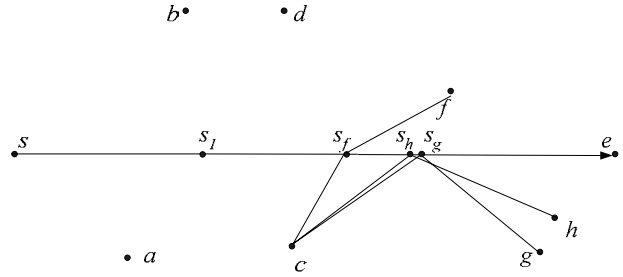


Figure 2.3: TP queries – second step

The method can extend to *k*NN. The only difference is that now the influence point s_x of *x* is the point that *x* starts to get closer to [*s*,*e*] than any of the *k* current neighbors. Specifically, assuming that the *k* current neighbors are $a_1, a_2, \ldots, a_k$, we first compute the influence points s_{xi} of *x* with respect to each a_i (i=1,2,…,k) following the previous approach. Then, s_x is set to the minimum of $s_{x1}, s_{x2}, \ldots, s_{xk}$.

This technique avoids the drawbacks of sampling, but it is very output-sensitive in the sense that it needs to perform *n* NN queries in order to compute the result, where *n* is the number of split points. Although, these *n* queries may access similar pages, and therefore, benefit from the existence of a buffer, the cost is still prohibitive for large queries and datasets due to the CPU overhead. The motivation of this work is to solve the problem by applying a single query for the whole result. Towards this direction, in the next section we describe some properties of the problem that permit the development of efficient algorithms.

Recently, Benetis, et al [BJKS02] address CNN queries from a mathematical point of view. Our algorithm, on the other hand, is based on several geometric problem characteristics. Further we also provide performance analysis, and discuss complex query types (e.g., trajectory nearest neighbor search).

3. Definitions and Problem Characteristics

The objective of a CNN query is to retrieve the set of nearest neighbors of a segment $q=[s, e]$ together with the resulting list SL of split points. The starting (*s*) and ending (*e*) points constitute the first and last elements in SL. For each split point $s_i \in$ SL ($0 \le i < |SL|-1$): $s_i \in q$ and all points in $[s_i, s_{i+1}]$ have the same NN, denoted as s_i.NN. For example, s_1.NN in Figure 1.1 is point *c*, which is also the NN for all points in interval $[s_1, s_2]$. We say that s_i.NN (e.g., *c*) *covers* point s_i (s_1) and interval $[s_i, s_{i+1}]$ ($[s_1, s_2]$).

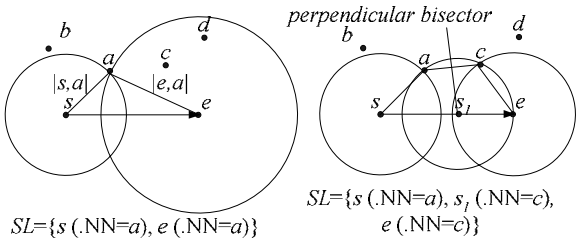
In order to avoid multiple database scans, we aim at reporting all split (and the corresponding covering) points with a single traversal. Specifically, we start with an initial SL that contains only two split points *s* and *e* with their covering points set to $\varnothing$ (meaning that currently the NN of all points in [*s*,*e*] are unknown), and incrementally update the SL during query processing. At each step, SL contains the current result with respect to all the data points processed so far. The final result contains each split point s_i that remains in SL after the termination together with its nearest neighbor s_i.NN.

Processing a data point *p* involves updating SL, if *p* is closer to some point $u \in [s,e]$ than its current nearest neighbor *u*.NN (i.e., if *p* covers *u*). An exhaustive scan of [*s*,*e*] (for points *u* covered by *p*) is intractable because the number of points is infinite. We observe that it suffices to examine whether *p* covers any split point currently in SL, as described in the following lemma.

Lemma 3.1: Given a split list SL $\{s_0, s_1, \ldots, s_{|SL-1|}\}$ and a new data point *p*, *p* covers some point on query segment *q* if and only if *p* covers a split point.

As an illustration of Lemma 3.1, consider Figure 3.1a where the set of data points $P=\{a, b, c, d\}$ is processed in alphabetic order. Initially, SL={*s*, *e*} and the NN of both split points are unknown. Since *a* is the first point encountered, it becomes the current NN of every point in *q*, and information about SL is updated as follows: *s*.NN= *e*.NN= *a* and $dist(s, s.NN) = |s, a|$, $dist(e, e.NN) = |e, a|$, where $|s, a|$ denotes the Euclidean distance between *s* and *a* (other distance metrics can also be applied). The circle centered at *s* (*e*) with radius $|s, a|$ ($|e, a|$) is called the *vicinity circle* of *s* (*e*).

When processing the second point *b*, we only need to check whether *b* is closer to *s* and *e* than their current NN, or equivalently, whether *b* falls in their vicinity circles. The fact that *b* is outside both circles indicates that every point in [*s*, *e*] is closer to *a* (due to Lemma 3.1); hence we ignore *b* and continue to the next point *c*.



(a) After processing *a* (b) After processing *c*

Figure 3.1: Updating the split list

Since *c* falls in the vicinity circle of *e*, a new split point s_1 is inserted to SL; s_1 is the intersection between the query segment and the perpendicular bisector of segment [*a*, *c*] (denoted as $\perp(a, c)$), meaning that points to the left of s_1 are closer to *a*, while points to the right of s_1 are closer to *c* (see Figure 3.1b). The NN of s_1 is set to *c*, indicating that *c* is the NN of points in [s_1, *e*]. Finally point *d* does not update SL because it does not cover any split point (notice that *d* falls in the circle of *e* in Figure 3.1a, but not in Figure 3.1b). Since all points have been processed, the split points that remain in SL determine the final result (i.e., {<*a*, [*s*,s_1]>, <*c*, [s_1,*e*]> }).

In order to check if a new data point covers some split point(s), we can compute the distance from *p* to every s_i, and compare it with *dist*(s_i, s_i.NN). To reduce the number |SL| (i.e., the cardinality of SL) of distance computations, we observe the following *continuity property*.

Lemma 3.2 (covering continuity): The split points covered by a point *p* are continuous. Namely, if *p* covers split point s_i but not s_{i-1} (or s_{i+1}), then *p* cannot cover s_{i-j} (or s_{i+j}) for any value of j>1.

Consider, for instance, Figure 3.2, where SL contains s_{i-1}, s_i, s_{i+1}, s_{i+2}, s_{i+3}, whose NN are points *a*, *b*, *c*, *d*, *f* respectively. The new data point *p* covers split points s_i, s_{i+1}, s_{i+2} (*p* falls in their vicinity circles), but not s_{i-1}, s_{i+3}. Lemma 3.2 states that *p* cannot cover any split point to the left (right) of s_{i-1} (s_{i+3}). In fact, notice that all points to the left (right) of s_{i-1} (s_{i+3}) are closer to *b* (*f*) than *p* (i.e., *p* cannot be their NN).

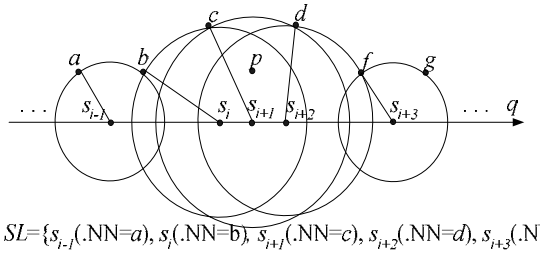


Figure 3.2: Continuity property

Figure 3.3 shows the situation after *p* is processed. The number of split points decreases by 1, whereas the positions of s_i and s_{i+1} are different from those in Figure 3.2. The covering continuity property permits the application of a binary search heuristic, which reduces (to *O*(*log*|*SL*|)) the number of computations required when searching for split points covered by a data point.

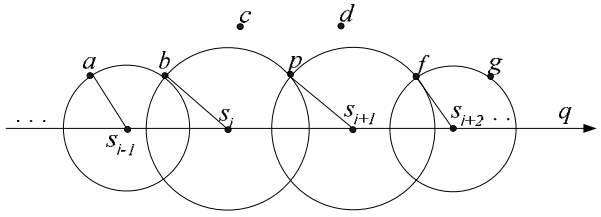


Figure 3.3: After *p* is processed (cont. Figure 3.2)

The above discussion can be extended to *k*CNN queries (e.g., find the 3 NN for any point on *q*). Consider Figure 3.4, where data points *a*, *b*, *c* and *d* have been processed and SL contains s_i and s_{i+1}. The current 3 NN of s_i are *a*, *b*, *c* (*c* is the farthest NN of s_i). At the next split point s_{i+1}, the 3NN change to *a*, *b*, *d* (*d* replaces *c*).

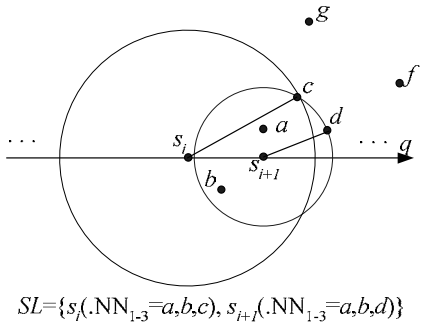


Figure 3.4: Example of *k*CNN (*k*=3)

Lemma 3.1 also applies to *k*CNN queries. Specifically, a new data point can cover a point on *q* (i.e., become one of the *k* NN of the point), if and only if it covers some split point(s). Figure 3.5 continues the example of Figure 3.4 by illustrating the situation after the processing of point *f*. The next point *g* does not update SL because *g* falls outside of vicinity circles of all split points. Lemma 3.2, on the other hand, does not apply to general *k*CNN queries. In Figure 3.5, for example, a new point *h* covers s_i and s_{i+3}, but not s_{i+1}, and s_{i+2} (which break the continuity).

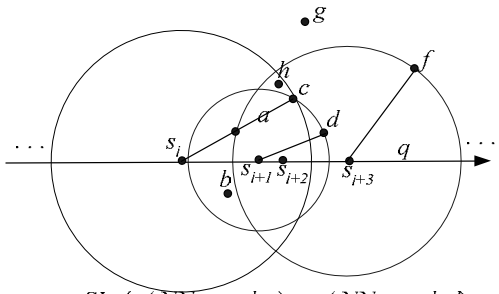


Figure 3.5: After processing *f*

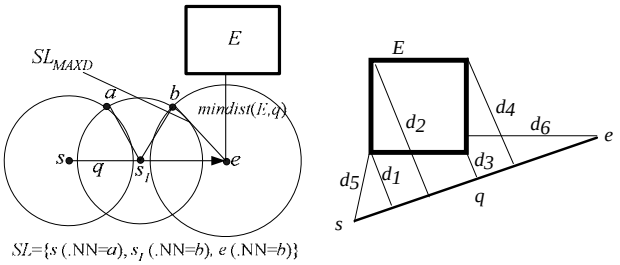
The above general methodology can be used for arbitrary dimensionality, where perpendicular bisectors and vicinity circles become perpendicular bisect-planes and vicinity spheres. Its application for processing non-indexed datasets is straightforward, i.e., the input dataset is scanned sequentially and each point is processed, continuously updating the split list. In real-life applications, however, spatial datasets, which usually contain numerous (in the order 10^5-10^6) objects, are indexed in order to support common queries such as selections, spatial joins and point-nearest neighbors. The next section illustrates how the proposed techniques can be used in conjunction with R-trees to accelerate search.

4. CNN Algorithms with R-trees

Like the point-NN methods discussed in Section 2, CNN algorithms employ branch-and-bound techniques to prune the search space. Specifically, starting from the root, the R-tree is traversed using the following principles: (i) when a leaf entry (i.e., a data point) p is encountered, SL is updated if p covers any split point (i.e., p is a *qualifying entry*); (ii) for an intermediate entry, we visit its subtree only if it may contain any qualifying data point. The advantage of the algorithm over exhaustive scan is that we avoid accessing nodes, if they cannot contain qualifying data points. In the sequel, we discuss several heuristics for pruning unnecessary node accesses.

Heuristic 1: Given an intermediate entry E and query segment q, the subtree of E may contain qualifying points *only if* $mindist(E,q) < SL_{MAXD}$, where $mindist(E,q)$ denotes the minimum distance between the MBR of E and q, and $SL_{MAXD} = \max\{dist(s_0, s_0.NN), dist(s_1, s_1.NN), \ldots, dist(s_{|SL|-1}, s_{|SL|-1}.NN)\}$ (i.e., SL_{MAXD} is the maximum distance between a split point and its NN).

Figure 4.1a shows a query segment $q=\{s, e\}$, and the current SL that contains 3 split points s, s_1, e, together with their vicinity circles. Rectangle E represents the MBR of an intermediate node. Since $mindist(E, q) > SL_{MAXD} = |e,b|$, E does not intersect the vicinity circle of any split point; thus, according to Lemma 3.1 there can be no point in E that covers some point on q. Consequently, the subtree of E does not have to be searched.



(a) E is not visited (b) Computing *mindist*

Figure 4.1: Pruning with $mindist(E, q)$

To apply heuristic 1 we need an efficient method to compute the *mindist* between a rectangle E and a line segment q. If E intersects q, then $mindist(E,q) = 0$. Otherwise, as shown in Figure 4.1b, $mindist(E,q)$ is the minimum (d_3) among the shortest distances (i) from each corner point of E to q (d_1, d_2, d_3, d_4), and (ii) from the start (s) and end (e) points to E (d_5, d_6). Therefore, the computation of $mindist(E, q)$ involves at most the cost of an intersection check, four *mindist* calculations between a point and a line segment, and two *mindist* calculations between a point and a rectangle. Efficient methods for the computation of the *mindist* between <point, rectangle> and <point, line segment> pairs have been discussed in previous work [RKV95, CMTV00].

Heuristic 1 reduces the search space considerably, while incurring relatively small computational overhead. However, tighter conditions can achieve further pruning. To verify this, consider Figure 4.2, which is similar to Figure 4.1a except that SL_{MAXD} ($=|e,b|$) is larger. Notice that the MBR of entry E satisfies heuristic 1 because $mindist(E,q)$ ($=mindist(E,s)$) $< SL_{MAXD}$. However, E cannot contain qualifying data points because it does not intersect any vicinity circle. Heuristic 2 prunes such entries, which would be visited if only heuristic 1 were applied.

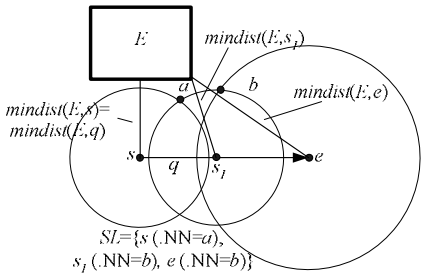


Figure 4.2: Pruning with $mindist(s_i, E)$

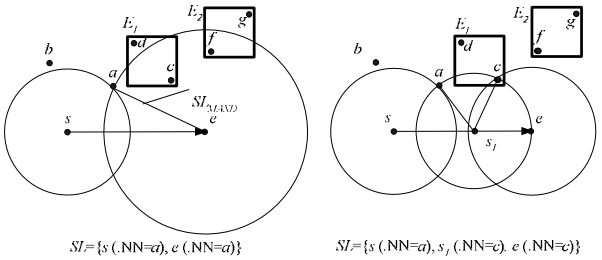
Heuristic 2: Given an intermediate entry E and query segment q, the subtree of E must be searched *if and only if* there exists a split point $s_i \in SL$ such that $dist(s_i,s_i.NN) > mindist(s_i, E)$.

According to heuristic 2, entry E in Figure 4.2 does not have to be visited since $dist(s,a) < mindist(s,E)$, $dist(s_1,b) < mindist(s_1,E)$ and $dist(e,b) < mindist(e,E)$. Although heuristic 2 presents the most tight conditions that a MBR must satisfy to contain a qualifying data point, it incurs more CPU overhead (than heuristic 1), as it requires computing the distance from E to each split point. Therefore, it is applied only for entries that satisfy the first heuristic.

The order of entry accesses is also very important to avoid unnecessary visits. Consider, for example, Figure 4.3a where points a and b have been processed, whereas entries E_1 and E_2 have not. Both E_1 and E_2 satisfy heuristics 1 and 2, meaning that they must be accessed according to the current status of SL. Assume that E_1 is visited first, the data points c, d in its subtree are

processed, and SL is updated as shown in Figure 4.3b. After the algorithm returns from E_1, the MBR of E_2 is pruned from further exploration by heuristic 1. On the other hand, if E_2 is accessed first, E_1 must also be visited. To minimize the number of node accesses, we propose the following visiting order heuristic, which is based on the intuition that entries closer to the query line are more likely to contain qualifying data points.

Heuristic 3: Entries (satisfying heuristics 1 and 2) are accessed in increasing order of their minimum distances to the query segment q.



(a) Before processing E_1 (b) After processing E_1

Figure 4.3: Sequence of accessing entries

When a leaf entry (i.e., a data point) p is encountered, the algorithm performs the following operations: (i) it retrieves the set of split points $S_{COVER}=\{s_i, s_{i+1}, \ldots, s_j\}$ covered by p, and (if S_{COVER} is not empty) (ii) it updates SL accordingly. As mentioned in Section 3, the set of points in S_{COVER} are continuous (for single NN). Thus, we can employ binary search to avoid comparing p with all current NN for every split point. Figure 4.4, illustrates the application of this heuristic assuming that SL contains 11 split points s_0-s_{10}, and the NN of s_0, .., s_5 are points a, b, c, d, f and g respectively.

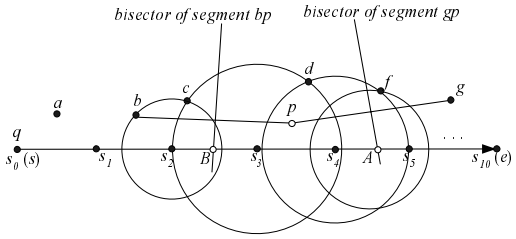


Figure 4.4: Binary search for covered split points

First, we check if the new data point p covers the middle split point s_5. Since the vicinity cycle of s_5 does not contain p, we can conclude that p does not cover s_5. Then, we compute the intersection (A in Figure 4.4) of q with the perpendicular bisector of p and s_5.NN(=g). Since A lies to the left of s_5, all split points potentially covered by p are also to the left of s_5. Hence, now we check if p covers s_2 (i.e., the middle point between s_0 and s_5). Since the answer is negative, the intersection (B) of q and $\perp(p, s_2.\text{NN})$ is computed. Because B lies to the right of s_2, the search proceeds with point s_3 (middle point between s_2 and s_5), which is covered by p.

In order to complete S_{COVER} (=$\{s_3, s_4\}$), we need to find the split points covered immediately before or after s_3, which is achieved by a simple bi-directional scanning process. The whole process involves at most $\log(|SL|)+|S_{COVER}|+2$ comparisons, out of which $\log(|SL|)$ are needed for locating the first split point (binary search), and $|S_{COVER}|+2$ for the remaining ones (the additional 2 comparisons are for identifying the first split points on the left/right of S_{COVER} not covered by p).

Finally the points in S_{COVER} are updated as follows. Since p covers both s_3 and s_4, it becomes the NN of every point in interval [s_3, s_4]. Furthermore, another split point s_3' (s_4') is inserted in SL for interval [s_2, s_3] ([s_4, s_5]) such that the new point has the same distance to s_2.NN=c (s_4.NN=f) and p. As shown in Figure 4.5, s_3' (s_4') is computed as the intersection between q and $\perp(c, p)$ ($\perp(f, p)$). Finally, the original split points s_3 and s_4 are removed. Figure 4.6 presents the pseudo-code for handling leaf entries.

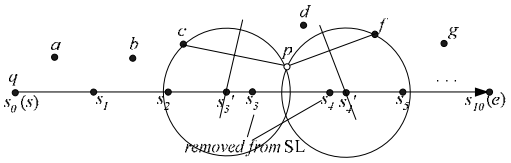


Figure 4.5: After updating the split list

Algorithm Handle_Leaf_Entry
/*p: the leaf entry being handled, SL: the split list*/
1. apply binary search to retrieve all split points covered by p: $S_{COVER}=\{s_i, s_{i+1}, \ldots, s_j\}$
2. let $u=s_{i-1}$.NN and $v=s_j$.NN
3. remove all split points in S_{COVER} from SL
4. add a split point s_i' at the intersection of q and $\perp(u, p)$ with s_i'.NN=p, $dist(s_i', s_i'.\text{NN})=|s_i', p|$
5. add a split point s_{i+1}' at the intersection of q and $\perp(v, p)$ with s_{i+1}'.NN=p, $dist(s_{i+1}', s_{i+1}'.\text{NN})=|s_{i+1}', p|$

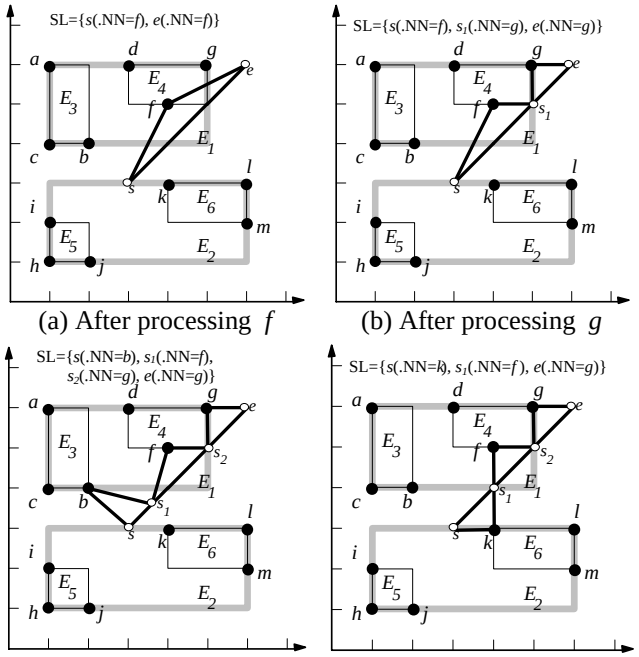
End Handle_Leaf_Entry

Figure 4.6: Algorithm for handling leaf entries

The proposed heuristics can be applied with both the depth-first and best-first traversal paradigms discussed in Section 2. For simplicity, we elaborate the complete CNN algorithm using depth-first traversal on the R-tree of Figure 2.1. To answer the CNN query [s,e] of Figure 4.7a, the split list SL is initiated with 2 entries {s, e} and $SL_{MAXD}=\infty$. The root of the R-tree is retrieved and its entries are sorted by their distances to segment q. Since the *mindist* of both E_1 and E_2 are 0, one of them is chosen (e.g., E_1), its child node (N_1) is visited, and the entries inside it are sorted (order E_4, E_3). Node N_4 (child of E_4) is accessed and points f, d, g are processed according to their distances to q. Point f becomes the first NN of s and e, and SL_{MAXD} is set to $|s, f|$ (Figure 4.7a).

The next point g covers e and adds a new split point s_1 to SL (Figure 4.7b). Point d does not incur any change because it does not cover any split point. Then, the algorithm backtracks to N_1 and visits the subtree of E_3. At

this stage SL contains 4 split points and SL_{MAXD} is decreased to $|s_1,b|$ (Figure 4.7c). Now the algorithm backtracks to the root and then reaches N_6 (following entries E_2, E_6), where SL is updated again (note the position change of s_1) and SL_{MAXD} becomes $|s,k|$ (Figure 4.7d). Since $mindist(E_5,q) > SL_{MAXD}$, N_5 is pruned by heuristic 1, and the algorithm terminates with the final result: {<*k*, [*s*, s_1]>, <*f*, [s_1,s_2]>, <*g*,[s_2, *e*]>}.



(a) After processing *f* (b) After processing *g*

(c) After processing *b* (d) After processing *k*

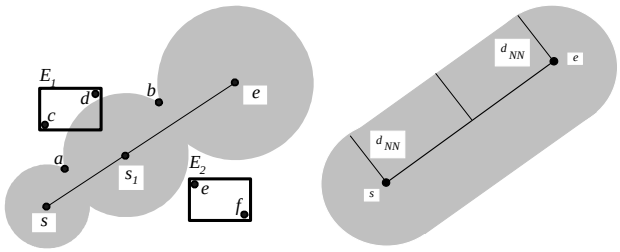
Figure 4.7: Processing steps of the CNN algorithm

5. Analysis of CNN Queries

In this section, we analyze the optimal performance for CNN algorithms and propose cost models for the number of node accesses. Although the discussion focuses on R-trees, extensions to other access methods are straightforward.

The number of node accesses is related to the *search region* of a query *q*, which corresponds to the data space area that must be searched to retrieve all results (i.e., the set of NN of every point on *q*). Consider, for example, query segment *q* in Figure 5.1a, where the final result is {<*a*, [*s*, s_1]>, <*b*, [s_1, *e*]>}. The search region (shaded area) is the union of the vicinity circles of *s*, s_1 and *e*. All nodes whose MBR (e.g., E_1) intersects this area may contain qualifying points. Although in this case E_1 does not affect the result (*c* and *d* are not the NN of any point), in order to determine this, any algorithm must visit E_1's subtree. On the other hand, optimal algorithms will not visit nodes (e.g., E_2) whose MBRs do not intersect the search region because they cannot contain qualifying data points. The above discussion is summarized by the following lemma (which is employed by heuristic 2).

Lemma 5.1: An optimal algorithm accesses only those nodes whose MBRs *E* satisfy the following condition: $mindist(s_i, E) < dist(s_i, s_i.\text{NN})$, for each final split point s_i.



(a) Actual search region (b) Approx. search region

Figure 5.1: The search region of a CNN query

The search region R_{SEARCH}, as shown in Figure 5.1a, is irregular. In order to facilitate analysis, we approximate R_{SEARCH} with a regular region such that every point on its boundary has minimum distance d_{NN} to *q* (Figure 5.1b), where d_{NN} is the average distance of all query points to their NN. For uniform data distribution and unit workspace, d_{NN} can be estimated as [BBKK97, BBK+01] (*N* is the total number points in the data set)[1].

$$d_{NN} \approx \sqrt{1/(\pi N)} \qquad (5\text{-}1)$$

Let *E* be a node MBR with edge lengths $E.l_1$ and $E.l_2$. The *extended region* E_{EXT} of *E* corresponds to the original MBR enlarged by d_{NN} and the query length *q.l* as shown in Figure 5.2.

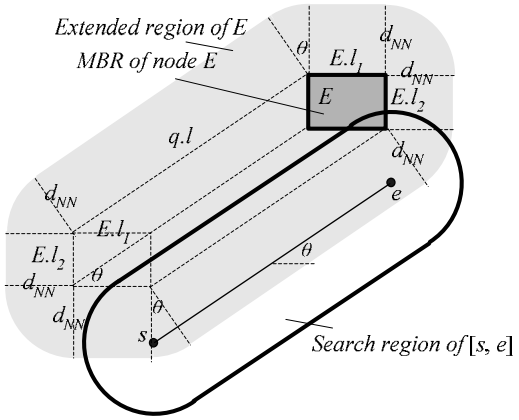


Figure 5.2: The extended region of *E*

Let $P_{ACCESS}(E,q)$ be the expected probability that the MBR *E* of a node intersects the search region. Equivalently, $P_{ACCESS}(E,q)$ denotes the probability that E_{EXT} covers the start point *s* of *q*. For uniform distribution and unit workspace, this probability equals the area of E_{EXT}. Thus,

$$P_{ACCESS}(E,q) = area(E_{EXT}) =$$

[1] Similar approaches have been commonly adopted in previous analysis of point-NN queries. The rationale of equation (5-1) is that the vicinity circle at the query point *q* contains exactly one (out of *N*) point, i.e., $\pi\, d_{NN}^2 = 1/N$.

$$\pi d_{NN}{}^2 + E.l_1 \cdot E.l_2 + 2d_{NN}\left(E.l_1 + E.l_2 + q.l\right) + 2q.l\left(E.l_1 \cdot |\cos\theta| + E.l_2 \cdot |\sin\theta|\right) \quad (5\text{-}2)$$

where d_{NN} is given by equation 5-1. In order to estimate the extents ($E.l_{1i}$, $E.l_{2i}$) of nodes at each level i of the R-tree, we use the following formula [TSS00]:

$$E.l_{1i} = E.l_{2i} = \sqrt{D_i / N_i} \quad 0 \le i \le h-1, \text{ where} \quad (5\text{-}3)$$

$$D_i = \left(1 + \frac{\sqrt{D_{i-1}} - 1}{\sqrt{f}}\right)^2 \quad D_0 = \left(1 - \frac{1}{\sqrt{f}}\right)^2 \quad N_i = N_{i-1}/f \;,\; N_0 = N/f$$

where h is the height of the tree, f the average node fanout, N_i is the number of level i nodes, and N the cardinality of the dataset. Therefore, the expected number of node accesses (NA) during a CNN query is:

$$NA(CNN) = \sum_{i=0}^{h-1} N_i \cdot P_{ACCESS}\left(E.l_i, q\right) = \sum_{i=0}^{h-1} N_i \cdot \begin{bmatrix} \pi d_{NN}{}^2 + E.l_i{}^2 + 2 \cdot d_{NN}\left(2 \cdot E.l_i + q.l\right) \\ +2 \cdot q.l \cdot E.l_i\left(|\cos\theta| + |\sin\theta|\right) \end{bmatrix} \quad (5\text{-}4)$$

Equation 5-4 suggests that the cost of a CNN query depends on several factors: (i) the dataset cardinality N, (ii) the R-tree structure, (iii) the query length $q.l$, and (iv) the orientation angle θ of q. Particularly, queries with $\theta=\pi/4$ have the largest number of node accesses among all queries with the same parameters N and $q.l$.

Notice that each data point that falls inside the search region is the NN of some point on q. Therefore, the number (n_{NN}) of distinct neighbors in the final result is:

$$n_{NN} = N \cdot area(R_{SEARCH}) = N\left(\pi d_{NN}{}^2 + 2d_{NN} \cdot q.l\right) \quad (5\text{-}5)$$

The CPU costs of CNN algorithms (including the TP approach discussed in Section 2) are closely related to the number of node accesses. Specifically, assuming that the fanout of a node is f, the total number of processed entries equals $f \cdot NA$. For our algorithm, the number of node accesses NA is given by equation 5-4; for the TP approach, it is estimated as $NA_{TP} \cdot n_{NN}$, where NA_{TP} is the average number of node accesses for each TP query, and n_{NN} equals the total number of TP queries. Therefore, the CPU overhead of the TP approach grows linearly with n_{NN}, which, (according to equation 5-5) increases with the data set size N, and query length $q.l$.

Finally, the above discussion can be extended to arbitrary data and query distributions with the aid of histograms. In our implementation, we adopt a simple partition-based histogram that splits the space into $m \times m$ regular bins, and for each bin_i we maintain the number of data points $N_{bin\text{-}i}$ that fall inside it. To estimate the performance of a query q, we take the average (N_{bin_avg}) of the $N_{bin\text{-}i}$ for all bins that are intersected by q. Then, we apply the above equations by setting $N = m^2 \cdot N_{bin\_avg}$ and assuming uniformity in each bin.

6. Complex CNN Queries

The CNN query has several interesting variations. In this section, we discuss two of them, namely, kCNN and trajectory NN queries.

6.1 The kCNN query

The proposed algorithms for CNN queries can be extended to support kCNN queries, which retrieve the k NN for every point on query segment q. Heuristics 1-3 are directly applicable except that, for each split point s_i, $dist(s_i, s_i.\text{NN})$ is replaced with the distance ($dist(s_i, s_i.\text{NN}_k)$) from s_i to its k^{th} (i.e., farthest) NN. Thus, the pruning process is the same as CNN queries.

The handling of leaf entries is also similar. Specifically, each leaf entry p is processed in a two-step manner. The first step retrieves the set S_{COVER} of split points s_i that are covered by p (i.e., $|s_i, p| < dist(s_i, s_i.\text{NN}_k)$). If no such split point exists, p is ignored (i.e., it cannot be one of the k NN of any point on q). Otherwise, the second step updates the split list. Since the continuity property does not hold for $k>2$, the binary search heuristic cannot be applied. Instead, a simple exhaustive scan is performed for each split point.

On the other hand, updating the split list after retrieving the S_{COVER} is more complex than CNN queries. Figure 6.1 shows an example where SL currently contains four points s_0,.., s_3, whose 2NN are (a,b), (b,c), (b,d), (b,f) respectively. The data point being considered is p, which covers split points s_2 and s_3.

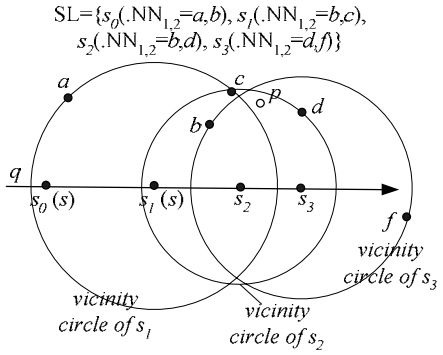
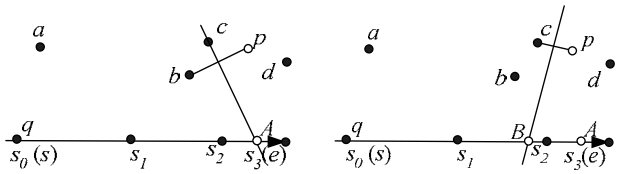


Figure 6.1: Updating SL (k=2) – the first step

No new splits are introduced on intervals [s_i, s_{i+1}] (e.g., [s_0, s_1]), if neither s_i nor s_{i+1} are covered by p. Interval [s_1, s_2], on the other hand must be handled (s_2 is covered by p), and new split points are identified with a sweeping algorithm as follows. At the beginning, the sweep point is at s_1, the current 2NN are (b, c), and p is the *candidate point*. Then, the intersections between q and $\perp(b, p)$ (A in Figure 6.2a), and between q and $\perp$ (c, p) (B in Figure 6.2b) are computed. Intersections (such as A) that fall out of [s_1, s_2] are discarded. Among the remaining ones, the intersection that has the shortest distance to the starting point s (i.e., B) becomes the next split point.



(c) Intrsct. of q and $\perp(a, p)$ (b) Intrsct. of q and $\perp(c, p)$

Figure 6.2: Identification of split point

The 2NN are updated to (b, p) at B, and now the new interval [B, s_2] must be examined with c as the new candidate. Because the continuity property does not hold, there is a chance that c will become again one of the kNN before s_2 is reached. The intersections of q with $\perp(b, c)$ and $\perp(p, c)$ are computed, and since both are outside [B, s_2], the sweeping algorithm terminates without introducing new split point. Similarly, the next interval [s_2, s_3] is handled and a split point C is created in Figure 6.3. The outdated split points (s_2) are eliminated and the updated SL contains: s_0, s_1, B, C, s_3, whose 2NN are (a,b), (b,c), (b,p), (d,p), (d,p) respectively.

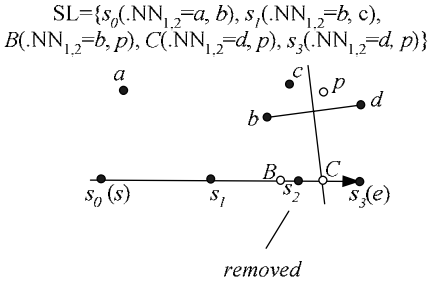


Figure 6.3: Updating SL (k=2) – the second step

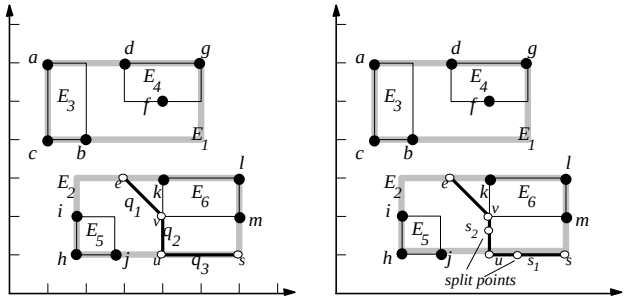
Finally, note that the performance analysis presented in Section 5 also applies to kCNN queries, except that in all equations, d_{NN} is replaced with $d_{k\text{-}NN}$, which corresponds to the distance between a query point and its k-th nearest neighbor. The estimation of $d_{k\text{-}NN}$ has been discussed in [BBK+01]:

$$d_{k-NN} \approx \sqrt{k/(\pi N)}$$

6.2 Trajectory Nearest Neighbor Search

So far we have discussed CNN query processing for a single query segment. In practice, a trajectory nearest neighbor (TNN) query consists of several consecutive segments, and retrieves the NN of every point on each segment. An example for such a query is "find my nearest gas station at each point during my route from city A to city B". The adaptation of the proposed techniques to this case is straightforward.

Consider, for instance, Figure 6.4a, where the query consists of 3 line segments q_1=[s, u], q_2=[u, v], q_3=[v, e]. A separate split list ($SL_{1,2,3}$) is assigned to each query segment. The pruning heuristics are similar to those for CNN, but take into account all split lists. For example, a counterpart of heuristic 1 is: the sub-tree of entry E can be pruned if, for each query segment q_i and the corresponding split list: $mindist(E, q_i) > SL_{i\text{-}MAXD}$. Heuristics 2 and 3 are adapted similarly. When a leaf entry is encountered, all split lists are checked and updated if necessary. Figure 6.4b shows the final results (i.e., <m, [s, s_1]>, <j, [s_1, s_2]>, <k, [s_2, e]>), after accessing E_2, E_6, E_5 (in this order). Notice that the gain of TNN compared to the TP approach, is even higher due to the fact that the number of split points increases with the number of query segments. The extension to kTNN queries is similar to kCNN.



(a) Initial situation (b) Final situation

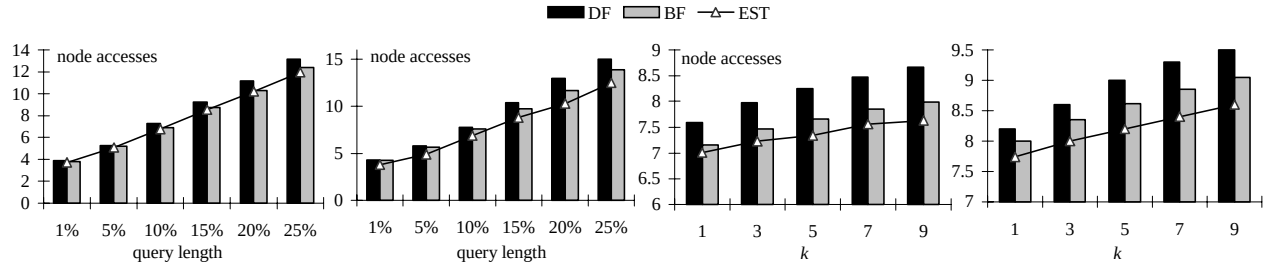
Figure 6.4: Processing a TNN query

7. Experiments

In this section, we perform an extensive experimental evaluation to prove the efficiency of the proposed methods using one uniform and two real point datasets. The first real dataset, CA, contains 130K sites, while the second one, ST, contains the centroids of 2M MBRs representing street segments in California [Web]. Performance is measured by executing workloads, each consisting of 200 queries generated as follows: (i) the start point of the query distributes uniformly in the data space, (ii) its orientation (angle with the x-axis) is randomly generated in [0, 2π), and (iii) the query length is fixed for all queries in the same workload. Experiments are conducted with a Pentium IV 1Ghz CPU and 256 Mega bytes memory. The disk size is set to 4K bytes and the maximum fanout of an R-tree node equals 200 entries.

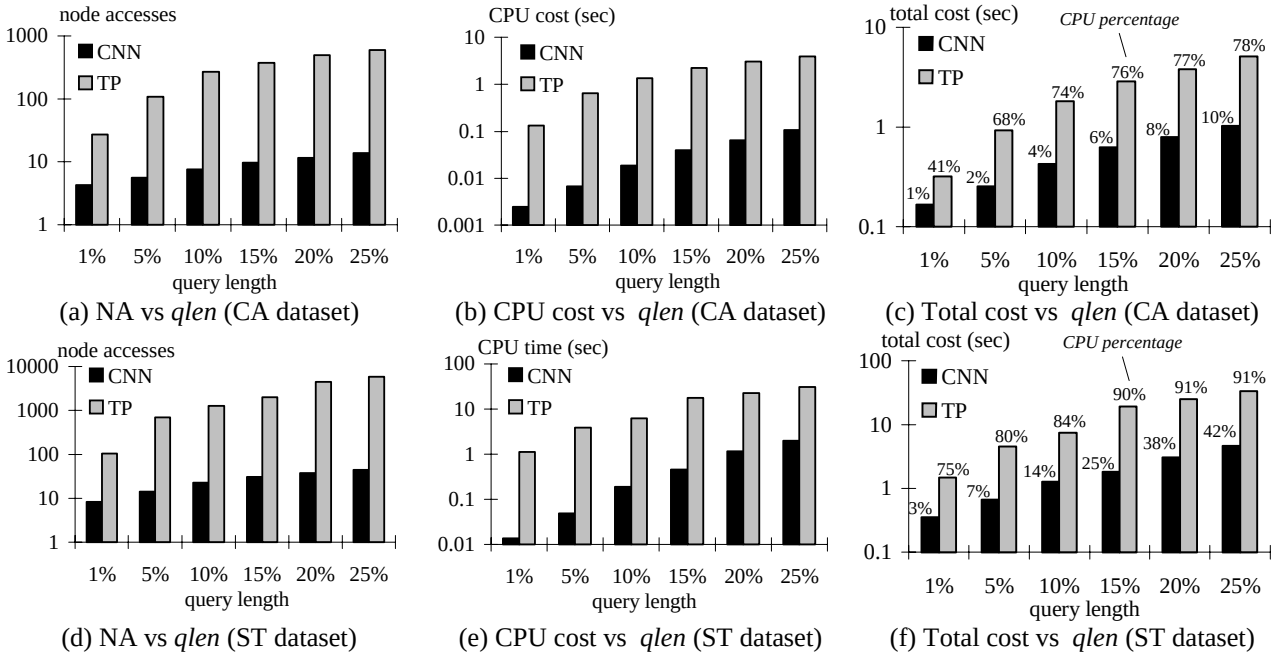
The first set of experiments evaluates the accuracy of the analytical model. For estimations on the real datasets we apply the histogram (50×50 bins) discussed in Section 5. Figures 7.1a and 7.1b illustrate the number of node accesses (NA) as a function of the query length *qlen* (1% to 25% of the axis) for the uniform and CA datasets, respectively (the number of neighbors k is fixed to 5). In particular, each diagram includes: (i) the NA of a CNN implementation based on depth-first (DF) traversal, (ii) the NA of a CNN implementation based on best-first (BF) traversal, (iii) the estimated NA obtained by equation (5-4). Figures 7.1c (for the uniform dataset) and 7.1d (for CA) contain a similar experiment, where *qlen* is fixed to 12.5% and k ranges between 1 and 9.

The BF implementation requires about 10% fewer NA than the DF variation of CNN, which agrees with



(a) Uniform (k=5) (b) CA-Site (k=5) (c) Uniform ($qlen$=12.5%) (d) CA-Site ($qlen$=12.5%)

Figure 7.1: Evaluation of cost models



(a) NA vs *qlen* (CA dataset) (b) CPU cost vs *qlen* (CA dataset) (c) Total cost vs *qlen* (CA dataset)

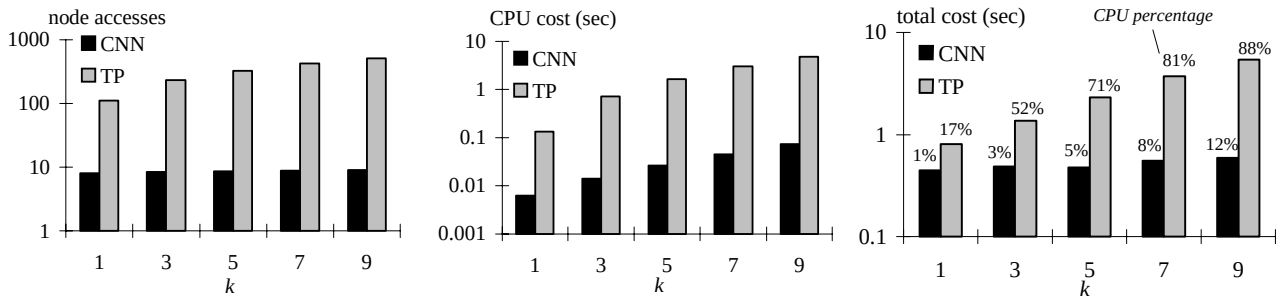
(d) NA vs *qlen* (ST dataset) (e) CPU cost vs *qlen* (ST dataset) (f) Total cost vs *qlen* (ST dataset)

Figure 7.2: Performance vs. query length (k=5)

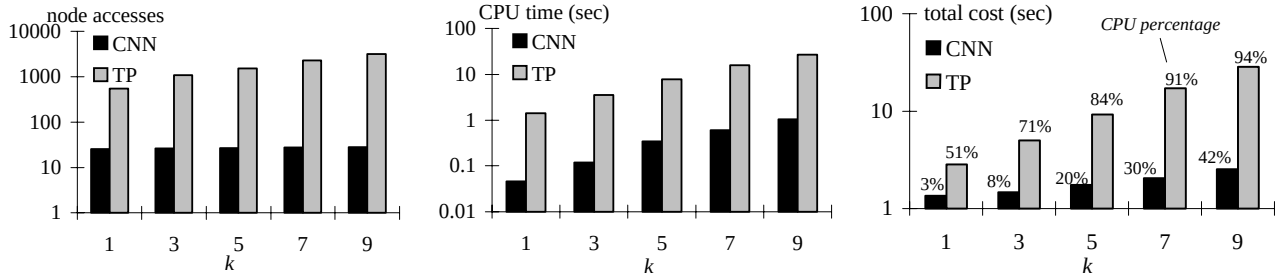
previous results on point-NN queries [HS99]. In all cases the estimation of the cost model is very close (less than 5% and 10% errors for the uniform and CA dataset, respectively) to the actual NA of BF, which indicates that: (i) the model is accurate and (ii) BF CNN is nearly optimal. Therefore, in the following discussion we select the BF approach as the representative CNN method. For fairness, BF is also employed in the implementation of the TP approach.

The rest of the experiments compare CNN and TP algorithms using the two real datasets CA and ST. Unless specifically stated, an LRU buffer with size 10% of the tree is adopted (i.e., the cache allocated to the tree of ST is larger). Figure 7.2 illustrates the performance of the algorithms (NA, CPU time and total cost) as a function of the query length (k = 5). The first row corresponds to CA, and the second one to ST, dataset. As shown in Figures 7.2a and 7.2d, CNN accesses 1-2 orders of magnitude fewer nodes than TP. Obviously, the performance gap increases with the query length since more TP queries are required.

The burden of the large number of queries is evident in Figures 7.2b and 7.2e that depict the CPU overhead. The relative performance of the algorithms on both datasets indicates that similar behaviour is expected independently of the input. Finally, Figures 7.2c and 7.2f show the total cost (in seconds) after charging 10ms per I/O. The number on top of each column corresponds to the percentage of CPU-time in the total cost. CNN is I/O-bounded in all cases, while TP is CPU-bounded. Notice that the CPU percentages increase with the query lengths for both methods. For CNN, this happens because, as the query becomes longer, the number of split points increases, triggering more distance computations. For TP, the buffer absorbs most of the I/O cost since successive queries access similar pages. Therefore, the percentage of CPU-time dominates the I/O cost as the query length increases. The CPU percentage is higher in ST because of its density; i.e., the dataset contains 2M points (as opposed to 130K) in the same area as CA. Therefore, for the same query length, a larger number of neighbors will be retrieved in ST (than in CA).

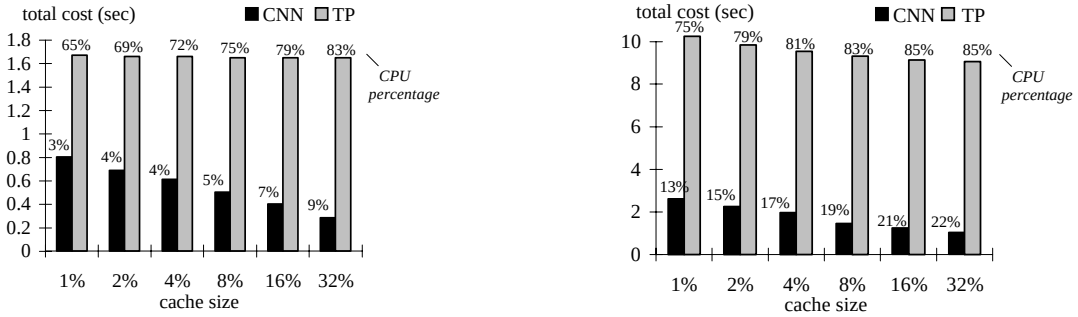


(a) NA vs. k (CA dataset) (b) CPU cost vs. k (CA dataset) (c) Total cost vs. k (CA dataset)



(d) NA vs. k (ST dataset) (e) CPU cost vs. k (ST dataset) (f) Total cost vs. k (ST dataset)

Figure 7.3: Comparison with various k values (query length=12.5%)



(a) CA (a) ST

Figure 7.4: Total cost under different cache sizes (*qlen*=12.5%, k=5)

Next we fix the query length to 12.5% and compare the performance of both methods by varying k from 1 to 9. As shown in Figure 7.3, the CNN algorithm outperforms its competitor significantly in all cases (over an order of magnitude). The performance difference increases with the number of neighbors. This is explained as follows. For CNN, k has little effect on the NA (see Figures 7.3a and 7.3d). On the other hand, the CPU overhead grows due to the higher number of split points that must be considered during the execution of the algorithm. Furthermore, the processing of qualifying points involves a larger number of comparisons (with all NN of points in the split list). For TP, the number of tree traversals increases with k, which affects both the CPU and the NA significantly. In addition, every query involves a larger number of computations since each qualifying point must be compared with the k current neighbors.

Finally, we evaluate performance under different buffer sizes, by fixing *qlen* and k to their standard values (i.e., 12.5% and 5 respectively), and varying the cache size from 1% to 32% of the tree size. Figure 7.4 demonstrates the total query time as a function of the cache size for the CA and ST datasets. CNN receives larger improvement than TP because its I/O cost accounts for a higher percentage of the total cost.

To summarize, CNN outperforms TP significantly under all settings (by a factor up to 2 orders of magnitude). The improvement is due to the fact that CNN performs only a single traversal on the dataset to retrieve all split points. Furthermore, according to Figure 7.1, the number of NA is nearly optimal, meaning that CNN visits only the nodes necessary for obtaining the final result. TP is comparable to CNN only when the input line segment is very short.

8. Conclusion

Although CNN is one of the most interesting and intuitive types of nearest neighbour search, it has received rather limited attention. In this paper we study the problem extensively and propose algorithms that avoid the pitfalls of previous ones, namely, the false misses and the high processing cost. We also propose theoretical bounds for the performance of CNN algorithms and experimentally verify that our methods are nearly optimal in terms of node accesses. Finally, we extend the techniques for the case of *k* neighbors and trajectory inputs.

Given the relevance of CNN to several applications, such as GIS and mobile computing, we expect this research to trigger further work in the area. An obvious direction refers to datasets of extended objects, where the distance definitions and the pruning heuristics must be revised. Another direction concerns the application of the proposed techniques to dynamic datasets. Several indexes have been proposed for moving objects in the context of spatiotemporal databases [KGT99a, KGT99b, SJLL00]. These indexes can be combined with our techniques to process prediction-CNN queries such as "according to the current movement of the data objects, find my nearest neighbors during the next 10 minutes".

Acknowledgements

This work was supported by grants HKUST 6081/01E and HKUST 6070/00E from Hong Kong RGC.

References

[BBKK97] Berchtold, S., Bohm, C., Keim, D.A., Kriegel, H. A Cost Model for Nearest Neighbor Search in High-Dimensional Data Space. ACM PODS, 1997.

[BBK+01] Berchtold, S., Bohm, C., Keim, D., Krebs, F., Kriegel, H.P. On Optimizing Nearest Neighbor Queries in High-Dimensional Data Spaces. ICDT, 2001.

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

[BKSS90] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. ACM SIGMOD, 1990.

[BS99] Bespamyatnikh, S., Snoeyink, J. Queries with Segments in Voronoi Diagrams. SODA, 1999.

[CMTV00] Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M. Closest Pair Queries in Spatial Databases. ACM SIGMOD, 2000.

[G84] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching, ACM SIGMOD, 1984.

[HS98] Hjaltason, G., Samet, H. Incremental Distance Join Algorithms for Spatial Databases. ACM SIGMOD 1998.

[HS99] Hjaltason, G., Samet, H. Distance Browsing in Spatial Databases. ACM TODS, 24(2), pp. 265-318, 1999.

[KGT99a] Kollios, G., Gunopulos, D., Tsotras, V. On Indexing Mobile Objects. ACM PODS, 1999.

[KGT99b] Kollios, G., Gunopulos, D., Tsotras, V. Nearest Neighbor Queries in a Mobile Environment. Spatio-Temporal Database Management Workshop, 1999.

[KSF+96] Korn, F., Sidiropoulos, N., Faloutsos, C., Siegel, E, Protopapas, Z. Fast Nearest Neighbor Search in Medical Image Databases. VLDB, 1996.

[RKV95] Roussopoulos, N., Kelly, S., Vincent, F. Nearest Neighbor Queries. ACM SIGMOD, 1995.

[SJLL00] Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M. Indexing the Positions of Continuously Moving Objects. ACM SIGMOD, 2000.

[SK98] Seidl, T., Kriegel, H. Optimal Multi-Step K-Nearest Neighbor Search. ACM SIGMOD, 1998.

[SR01] Song, Z., Roussopoulos, N. K-Nearest Neighbor Search for Moving Query Point. SSTD, 2001.

[SRF87] Sellis, T., Roussopoulos, N. Faloutsos, C.: The R+-tree: a Dynamic Index for Multi-Dimensional Objects, VLDB, 1987.

[SWCD97] Sistla, P., Wolfson, O., Chamberlain, S., Dao, S. Modeling and Querying Moving Objects. IEEE ICDE, 1997.

[TP02] Tao, Y., Papadias, D. Time Parameterized Queries in Spatio-Temporal Databases. ACM SIGMOD, 2002.

[TSS00] Theodoridis, Y., Stefanakis, E., Sellis, T. Efficient Cost Models for Spatial Queries Using R-trees. IEEE TKDE, 12(1), pp. 19-32, 2000.

[web] http://dias.cti.gr/~ytheod/research/datasets/spatial.html

[WSB98] Weber, R., Schek, H., Blott, S. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. VLDB, 1998.

[YOTJ01] Yu, C., Ooi, B.C., Tan, K.L., Jagadish, H.V. Indexing the Distance: An Efficient Method to KNN Processing. VLDB, 2001.