Novel Forms of Nearest Neighbor Queries in Spatio-Temporal Applications

Dimitris Papadias[+], Yufei Tao[+], Jun Zhang[+], Qiongmao Shen[+] and Nikos Mamoulis*

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

* Dept of Computer Science and Information Systems
University of Hong Kong
Pokfulam Road, Hong Kong
nikos@csis.hku.hk

Abstract Several types of nearest neighbor (NN) search have been proposed and studied in the context of spatial databases. The most common type is the *point NN query*, which retrieves the nearest neighbors of an input point. Such a query, however, is usually meaningless in highly dynamic environments where the query point or the database objects move/change over time. In this paper we study alternative forms of nearest neighbor queries suitable for spatio-temporal applications. In particular, we first describe time-parameterized queries, where in addition to the current nearest neighbors of a query point, the result contains the expected validity period of the result (given the query and object movement), as well as the change that will occur at the end of the validity period. Then, we discuss continuous NN search where the goal is to retrieve all nearest neighbors of a query trajectory, together with the validity interval of each nearest neighbor.

Νέες Μορφές Ερωτήσεων Πλησιέστερου Γείτονα σε Χωρο-χρονικές Εφαρμογές

Περίληψη Αρκετοί τύποι αναζήτησης πλησιέστερων γειτόνων (ΠΓ) έχουν προταθεί και μελετηθεί στο πλαίσιο των χωρικών βάσεων δεδομένων. Ο κοινότερος τύπος είναι η *ερώτηση πλησιέστερων γειτονικών σημείων*, η οποία ανακτά τους ΠΓ ενός σημείου αναφοράς σε ένα στατικό χώρο σημείων. Παρόλα αυτά η συγκεκριμένη ερώτηση συνήθως δεν έχει νόημα σε δυναμικά περιβάλλοντα, όπου οι θέση του σημείου αναφοράς καθώς και αυτές των σημείων στη βάση δεδομένων αλλάζουν με το χρόνο. Στο δοκίμιο αυτο μελετώνται εναλλακτικές μορφές ερωτήσεων ΠΓ, κατάλληλες για χωρο-χρονικές εφαρμογές όπου οι θέσεις των δεδομένων αλλάζουν με το χρόνο. Συγκεκριμένα, αρχικά περιγράφουμε *χρονο-παραμετρικές ερωτήσεις*, όπου, γνωρίζοντας τις κινήσεις των σημείων, εκτός από τους ΠΓ του σημείου αναφοράς, το αποτέλεσμα περιέχει την εκτιμούμενη διάρκειά της εγκυρότητάς τους καθώς και τις αλλαγές που θα γίνουν μετά από τη λήξη της εγκυρότητας των ΠΓ. Στη συνέχεια μελετάμε το πρόβλημα *συνεχούς αναζήτησης ΠΓ*, όπου ο στόχος είναι η ανάκτηση των ΠΓ ενός σημείου αναφοράς που κινείται στο χώρο σε κάθε θέση της τροχιάς του, καθώς και τα διαστήματα εγκυρότητας των ανακτώμενων ΠΓ.

Novel Forms of Nearest Neighbor Queries in Spatio-Temporal Applications

Dimitris Papadias[+], Yufei Tao[+], Jun Zhang[+], Qiongmao Shen[+] and Nikos Mamoulis*

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

* Dept of Computer Science and Information Systems
University of Hong Kong
Pokfulam Road, Hong Kong
nikos@csis.hku.hk

Abstract

Several types of nearest neighbor (NN) search have been proposed and studied in the context of spatial databases. The most common type is the *point NN query*, which retrieves the nearest neighbors of an input point. Such a query, however, is usually meaningless in highly dynamic environments where the query point or the database objects move/change over time. In this paper we study alternative forms of nearest neighbor queries suitable for spatio-temporal applications. In particular, we first describe time-parameterized queries, where in addition to the current nearest neighbors of a query point, the result contains the expected validity period of the result (given the query and object movement), as well as the change that will occur at the end of the validity period. Then, we discuss continuous NN search where the goal is to retrieve all nearest neighbors of a query trajectory, together with the validity interval of each nearest neighbor.

1. INTRODUCTION

Traditional NN queries, such as "which are my nearest gas station now?", are of limited practical use in spatio-temporal applications since, if query point (or the database objects) move, the output may be invalidated immediately (i.e., the NN changes continuously over time). Any result should be accompanied by an expiry period in order to be effective in practice. Motivated by this, we propose *time-parameterized (TP) NN queries* which return: (i) the objects that satisfy the corresponding spatial query, (ii) the *expiry time* of the result, and (iii) the *change* that causes the expiration of the result. Continuing the "gas station" example, the result could be <A, 1, B> meaning that the current NN is A, but at 1 minute (given the direction and the speed of the user's movement) B will become the nearest neighbor.

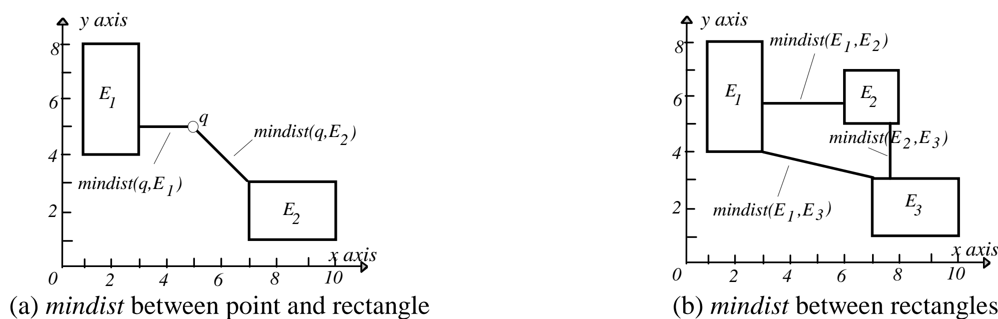
In addition to their importance as standalone methods, TP queries can be used for continuous NN processing. An example of such a query is "find my nearest gas station during my trip to from point *s* to point *e*". A TP query at the starting point *s* will return the first NN, its validity period and the next NN. Then, a second TP query is executed to return the validity period of the second gas station and so on until the ending point *e* is reached. This approach, however, is expected to incur significant overhead due to the potentially high number of queries, especially if the trajectory is long and a large number of

neighbors need to be retrieved. For this case of *continuous nearest neighbor* search we propose an alternative method that applies a single query for the entire trajectory.

The rest of the paper is organized as follows: Section 2 surveys related work in the context of spatial databases concentrating on point NN queries, which constitute the focus of this paper. Section 3 discusses TP NN queries and processing methods. Section 4 describes properties of the continuous nearest neighbor problem and proposes search heuristics that take advantage of these properties to facilitate efficiency. Section 5 presents an extensive experimental evaluation, while Section 6 concludes with directions for future work.

2. RELATED WORK

Following most of the related literature we assume that the (point) datasets are indexed by R-trees. Existing algorithms[1] for processing point NN queries in spatial databases are based on the branch and bound (BaB) paradigm using some bounds to prune the search space. The most common pruning bound is *mindist*, which is the minimum distance between the query object q and any object that can be in the subtree of entry E. Figure 2.1a illustrates *mindist* for the MBRs of E_1 and E_2 with respect to a point query q.



(a) *mindist* between point and rectangle (b) *mindist* between rectangles

Figure 2.1: *Mindist* examples

The first R-tree BaB algorithm, proposed in [RKV95], answers a NN query by traversing the R-tree in a depth-first (DF) manner. Specifically, starting from the root, all entries are sorted according to their *mindist* from the query point, and the entry with the lowest value is visited first. The process is repeated recursively until the leaf level where the first potential nearest neighbor is found. During backtracking to the upper levels, the algorithm only visits entries whose *mindist* is smaller than the distance of the nearest neighbor already found. As an example consider the R-tree of Figure 2.2, where the number in each entry refers to the *mindist* (for intermediate entries) or the actual distance (for point objects) from the query point (these numbers are not stored but computed dynamically during query processing). DF would first visit the node of root entry E_1 (since it has the minimum *mindist*), and then the node of E_4, where the first candidate object (a) is retrieved. When backtracking to the previous level, entry E_6 is

[1] For high dimensional spaces, there exist alternative techniques [KSF+96, SK98, BEK+98] to deal with the "dimensionality curse".

excluded since its *mindist* is greater than the distance of *a*, but E_5 has to be visited before backtracking again at the root level.

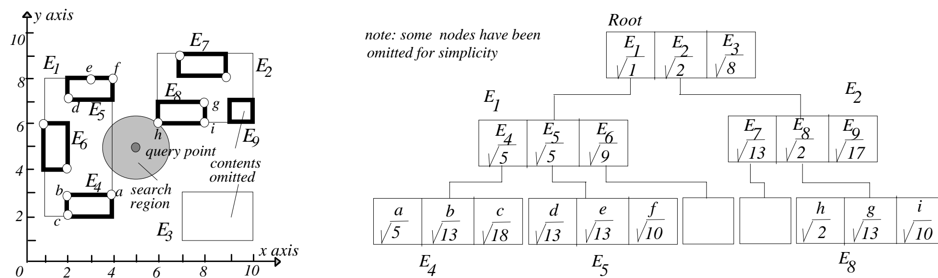


Figure 2.2: Example query

The performance of DF was shown to be suboptimal in [PM97], which reveals that an optimal NN search algorithm only needs to visit those nodes whose MBRs intersect the so-called "search region", i.e., a circle centered at the query point with radius equal to the distance between the query and its nearest neighbor (shaded circle in Figure 2.2). Based on this, [WSB98, BBK+01] investigate cost models for performing NN queries in high-dimensional space.

A best-first (BF) algorithm for KNN query processing using R-trees is proposed in [HS99]. BF keeps a *heap* with the entries of the nodes visited so far. Initially the heap contains the entries of the root sorted according to their *mindist*. The contents of the heap during the processing of the query of Figure 2.2 are shown in Figure 2.3. When E_1 is visited, it is removed from the heap and the entries of its node (E_4, E_5, E_6) are added together with their *mindist*. The next entry visited is E_2 (it has the minimum *mindist* in the heap), followed by E_8, where the actual result (*h*) is found and the algorithm terminates. BF is optimal in the sense that it only visits the nodes necessary for obtaining the nearest neighbor. Its performance in practice, however, may suffer from buffer thrashing if the available memory is not enough for the required heap. In this case part of the heap must be migrated to the disk, which may incur frequent disk accesses.

Action	*Heap*							*Result*
Visit Root	$E_1\sqrt{1}$	$E_2\sqrt{2}$	$E_3\sqrt{8}$					*{empty}*
follow E_1	$E_2\sqrt{2}$	$E_4\sqrt{5}$	$E_5\sqrt{5}$	$E_3\sqrt{8}$	$E_6\sqrt{9}$			*{empty}*
follow E_2	$E_8\sqrt{2}$	$E_4\sqrt{5}$	$E_5\sqrt{5}$	$E_3\sqrt{8}$	$E_6\sqrt{9}$	$E_7\sqrt{13}$	$E_9\sqrt{17}$	*{empty}*
follow E_8	$E_4\sqrt{5}$	$E_5\sqrt{5}$	$E_3\sqrt{8}$	$E_6\sqrt{9}$	$E_7\sqrt{13}$	$E_9\sqrt{17}$		*{(h, $\sqrt{2}$)}*

Report h and terminate

Figure 2.3: Example of BaB algorithms

The BaB framework also applies to closest pair queries that find the pair of objects from two datasets, such that their distance is the minimum among all pairs. Hjaltason and Samet [HS98] and Corral et al, [CMTV00] propose various algorithms based on the concepts of DF and BF traversal. The difference from point-NN is that the algorithms access two index structures (one for each data set) simultaneously. *Mindist* is now defined as the minimum distance between two objects that can lie in the subtrees of two intermediate entries (see Figure 2.1b). If the *mindist* of two intermediate entries E_1 and E_2 (one from each R-tree) is already greater than the distance of the closest pair of objects found so far, the sub-trees of E_1 and E_2 cannot contain a closest pair.

3. TIME PARAMETERIZED NN QUERIES

The output of a spatio-temporal[2] TP query has the general form <**R**,**T**,**C**>, where **R** is the set of objects satisfying the corresponding instantaneous query (i.e., current result), **T** is the expiry time of **R**, and **C** the set of objects that will affect **R** at **T**. From the set of objects in 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. Figure 3.1 shows a TP NN, where objects (points *a* to *g*) are static and query point *q* is moving east with speed 1. Point *d* is the current nearest neighbor of *q*.

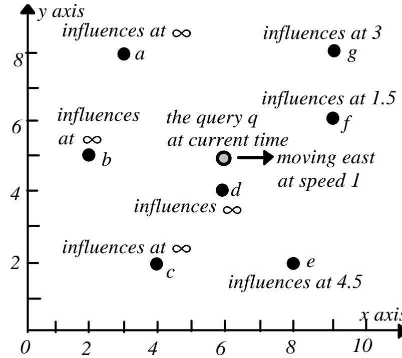


Figure 3.1: TP NN query

The result of a spatial query changes in the future because some objects "influence" its correctness. In the example of Figure 3.1, the influence time of an object is the time that it starts to get closer to the query than the current nearest neighbor *d*. For instance, the influence time of point *g* is 3, because at this time *g* will come closer to *q* than *d*. The influence time of points *a*, *b*, *c* is ∞ because they can never be closer to *q* than its current nearest neighbor *d* (observe that the influence time of *d* is also set to ∞). Notice that a non-infinite (i.e., different from ∞) influence time does not necessarily mean that the object will change the result; *g* will influence the query at time 3, only if the result does not change before due to another object (actually at time 3 the nearest neighbor is object *f*).

We denote the influence time of a point *p* with respect to a query *q* as $T_{INF}(p,q)$. The expiry time of the current result is the minimum influence time of all objects. Therefore, the time-parameterized component of a TP query can be reduced to a nearest neighbor problem by treating $T_{INF}(p,q)$ as the distance metric: the goal is to find the objects (**C**) with the minimum T_{INF} (**T**). These are the candidates that may generate the change of the result at the expiry time (by adding to or deleting from the previous answer set). T_{INF} for intermediate entries *E* is defined in a way similar to *mindist* in NN search: $T_{INF}(E,q)$ is the minimum influence time $T_{INF}(p,q)$ of any point *p* that may lie in the subtree of *E*. The above discussion serves as a high-level abstraction that establishes the close connection between the TP retrieval and NN search. In the sequel we show how to derive suitable $T_{INF}(p,q)$ and $T_{INF}(E,q)$.

[2] The concept of TP queries applies to most types of spatial queries involving selections and joins. Furthermore, although in this paper we assume that the dataset is static (only the query is dynamic), the method can also be applied for moving points using appropriate indexes (see [TP02]).

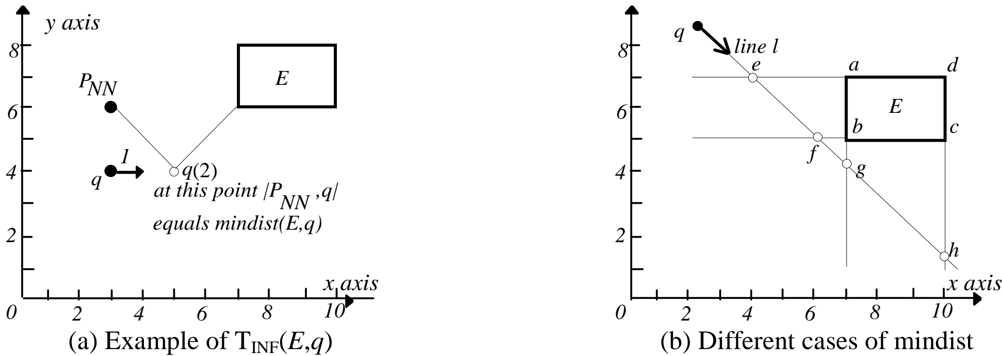
3.1 Derivation of Influence Time

We first consider single nearest neighbor queries before extending the solution to an arbitrary number of neighbors. To facilitate understanding, we present our solution for static point data in 2D space, although the discussion extends to rectangles (where the rationale is the same but the equations more complex). We denote with ($q_1,\dots,q_n$) the coordinates, and with ($q.V_1,\dots q.V_n$) the velocities of query point q on dimensions i=1,…,n.

Let P_{NN} be the current nearest neighbor of q. The influence time $T_{INF}(p,q)$ of a point p is the earliest time t in the future such that p starts to get closer to $q(t)$ than P_{NN}, where $q(t)$ is the position of q at time t. In general, $T_{INF}(p,q)$ is the minimum t that satisfies the following conditions[3]: $|p,q(t)|\le|P_{NN}, q(t)|$ and $t\ge 0$. If the above inequality can be transformed into the standard form $At+B\le 0$, where:

$$A=\sum_{i=1}^{n}2\left[(p_i-q_i)(-q.V_i)-(P_{NNi}-q_i)(-q.V_i)\right], \text{ and } B=\sum_{i=1}^{n}\left[(p_i-q_i)^2-(P_{NNi}-q_i)^2\right]$$

The solution is straightforward and omitted. If no t satisfies the inequality, $T_{INF}(p,q)$ is set to ∞. In case of intermediate entries, $T_{INF}(E,q)$ indicates the earliest time when some object in the subtree of E may start to be closer to q (than P_{NN}). This is illustrated through the example of Figure 3.2a. At time 2, the *mindist* of E to q becomes shorter than $|P_{NN},q|$, which implies that some object in E may start to get closer to q (i.e., $T_{INF}(E,q)=2$). More formally, $T_{INF}(E,q)$ is the minimum t that satisfies the condition: $mindist(E,q(t)) \le |P_{NN}, q(t)|$ and $t\ge 0$. This inequality requires rather complicated case-by-case discussion because the computation of $mindist(E,q(t))$ depends on the relative positions of E and q. Figure 3.2b illustrates an example where the query point is moving along line l. Before q reaches point e, $mindist(E,q)$ should be calculated with respect to point a. When q is on the line segment ef, *mindist* is the distance from q to edge ab of E. Similarly, after q passes points f, g, and h, *mindist* should be computed with respect to point b, edge bc, and point c respectively. The situation can be even more complex when, in addition to the query, MBR E is also dynamic, especially in higher dimensional spaces.



(a) Example of $T_{INF}(E,q)$ (b) Different cases of mindist

Figure 3.2: T_{INF} for intermediate entries

[3] $|a,b|$ denotes the distance between points a and b. Although we use Euclidean distance throughout the paper, other metrics can be applied.

Instead, we follow a simpler (but still efficient as shown in the experiments) conservative approach that underestimates *mindist* (to ensure the correctness of BaB algorithms). In particular, we approximate *mindist* with the perpendicular distance from q to a selected edge (or a plane in high-dimensional space) of MBR E. The edge of E is chosen as follows: (i) If the *mindist* at the current time between E and q is with respect to a corner point of E (e.g., point b in Figure 3.3a), then the selected edge (among the two edges connected to the corner point) is the more distant from q (e.g., edge ab is farther to q than bc); (ii) If the *mindist* is computed with respect to an edge (e.g., edge bc in Figure 3.3b), then we select this edge. In this case, the distance from the query point to the edge is exactly the *mindist* at the current time. The pseudo-code for the algorithm that applies to arbitrary dimensionality is shown in Figure 3.4; the algorithm returns a (hyper) plane of dimensionality n-1.

(a) Underestimation (b) Accurate estimation

Figure 3.3: Approximate *mindist*

Sel_Plane_Approx_*mindist* (E, q)
1. if q is contained in E at the current time
2. return NIL /*no edge selected since $T_{INF}(E,q)=0$*/
3. *sel_dim*=nil /*dimension selected plane is perpendicular to*/
4. *coord*=nil /*the coordinate and velocity of the selected plane on dimension *sel_dim**/
5. *plane_dist*=−∞ /*the distance from q to the selected plane at the current time*/
6. for each dimension i
7. if $q_i<E_{iL}$ /*$[E_{iL},E_{iR}]$ is the extent of E on dimension i*/
8. if ($E_{iL}-q_i$>plane_dist) /*q is further to the plane on this dimension than previous dimensions*/
9. *sel_dim*=i;
10. *plane_dist*=$E_{iL}-q_i$; *coord*=E_{iL};
11. elseif $q_i>E_{iR}$
12. if (q_i-E_{iR}>plane_dist) /*q is further to the plane on this dimension than previous dimensions*/
13. *sel_dim*=i;
14. *plane_dist*=q_i-E_{iR}; *coord*=E_{iR}
15. return the selected plane (at position *coord* on dimension *sel_dim*, moving at *velocity*)

end Sel_Plane_Approx_*mindist*

Figure 3.4: Selecting a plane to approximate mindist

Without loss of generality, assume that the plane l returned by the pseudo-code of Figure 3.4, is perpendicular to the ith dimension at point l_i, and moves along the dimension at speed $l.V_i$. $T_{INF}(E,q)$ is the minimum t that satisfies the condition $mindist(l(t),q(t)) \le |P_{NN}(t),q(t)|$ and $t\geq 0$. Using the usual notation for q, the above inequality is equivalent to:

$$|(q_i - l_i) + t \cdot q.V_i| \le \sqrt{\sum_{i=1}^{n}\left[(q_i - P_{NNi}) + t \cdot q.V_i\right]^2}$$

which can be transformed to the standard (and easily solvable) form $At^2+Bt+C\leq 0$, where

$$A = (q.\mathrm{V}_i)^2 - \sum_{i=1}^{n} (q.\mathrm{V}_i)^2 \quad C = (q_i - l_i)^2 - \sum_{i=1}^{n} (P_{NNi} - q_i)^2$$

and

$$B = -2 \cdot q.\mathrm{V}_i (l_i - q_i) + \sum_{i=1}^{n} 2 \cdot q.\mathrm{V}_i (P_{NNi} - q_i)$$

For TP *k*NN queries the influence time of a point *p* is defined as the earliest time that *p* starts to get closer to *q* than any of the *k* current neighbors. Specifically, assuming that the *k* current neighbors are $P_{NN1}, P_{NN2}, \ldots, P_{NNk}$, we first compute the influence time T_{INFi} of *p* with respect to each P_{NNj} (j=1,2,…, k) following the previous approach. Then, $\mathrm{T_{INF}}(p,q)$ is set to the minimum of $\mathrm{T_{INF1}}$, $\mathrm{T_{INF2}}$, …, $\mathrm{T_{INNk}}$. Similarly, for $\mathrm{T_{INF}}(E,q)$ we first compute the $\mathrm{T_{INFj}}$ of *E* with respect to each P_{NNj} and then set $\mathrm{T_{INF}}(E,q)$ to the minimum of $\mathrm{T_{INF1}}$, $\mathrm{T_{INF2}}$, …, $\mathrm{T_{INFk}}$.

3.2 Query Processing for TP NN queries

As discussed in Section 2, BaB algorithms can be classified in two broad categories: depth- and breadth-first search. Both types can be applied for processing TP NN queries. Figure 3.5a shows the pseudo-code of DF and Figure 3.5b of BF. We assume that initially a regular algorithm is executed to return the current NN (**R**), after which the influence times can be determined; TP-NN just computes the time-parameterized component (**T** and **C**). Both algorithms distinguish between (i) $\mathrm{T_{INF}}(p,q)<\mathbf{T}$ and (ii) $\mathrm{T_{INF}}(p,q)=\mathbf{T}$. In the second case multiple objects influence the result at the same time and the closest one becomes the next neighbor. The extension to TP KNN queries is straightforward.

```
Run traditional NN algorithm to obtain R

TP_NN-DF (current node N)
/*initially: T=∞, C= Ø, D= ∞   */
1.   if N is a leaf
2.   for each point p
3.      if T_INF(p,q)<T
4.         C=p
5.         T= T_INF(p,q)
6.      else if T_INF(p,q)=T
7.            if  |p,q(T)|< |C,q(T)|
8.               C=p
9.   else /*N is an intermediate node*/
10.      sort all the entries E by their T_INF(E,q)
11.      for each entry E
12.         if (T_INF(E,q)≤T)
13.            TP_NN (e.childnode)
13.  end TP_NN-DF
```

(a) DF

```
Run traditional NN algorithm to obtain R

TP_NN-BF ( )
1.  initialize a heap H that accepts <key,entry>
2.  retrieve the root node R
3.  for each entry E in R insert <T_INF(E,q),E> to H
4.  while (H is not empty)
5.   de-heap<key,E>/*E has the minimum key in H*/
6.   if E points to a leaf node
7.      for each point p in E.childnode
8.        if T_INF(p,q)<T
9.          C=p
10.         T= T_INF(p,q)
14.     else if T_INF(p,q)=T
15.           if  |p,q(T)|< |C,q(T)|
11.                        C=p
12.  else /*E points to a non-leaf node*/
13.     if T_INF(E,q) ≤T
14.       for each entry E' in E.childnode
15.          insert <T_INF(E',q),E'> to H
16.  end TP_NN-BF
```

(b) BF

Figure 3.5: BaB algorithms for time-parameterized NN queries

Finally, the framework facilitates performance analysis by utilizing previous findings on nearest neighbor search. Recall from Section 2, that an optimal NN algorithm only needs to visit those nodes, whose MBRs intersect the "search region" around the query point. Such search regions also apply for

processing the time-parameterized component (retrieval of **T** and **C**) of TP queries. Assuming that O_{NN} is the object with the minimum influence time, all entries E to be visited by an optimal algorithm should satisfy the condition: $T_{INF}(E,q) \leq T_{INF}(O_{NN},q)$. Based on this, Figure 3.6 demonstrates the corresponding search region (shaded area) of TP NN queries. The white area does not belong to the search region.

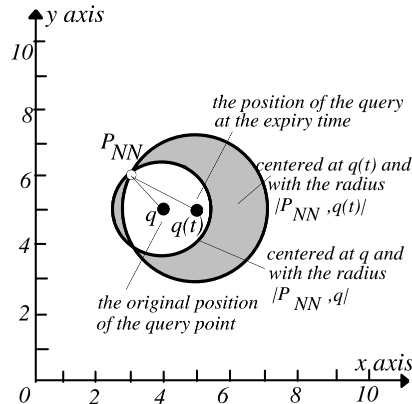


Figure 3.6: Search region for TP NN queries

4. CONTINUOUS NN QUERIES

Let P be a dataset of points in multi-dimensional space and $q = [s, e]$ a line segment. A continuous nearest neighbor (CNN) query retrieves the nearest neighbor (NN) of every point in q. In particular, the result contains a set of <**R**,**T**> tuples, where **R** is a point (or set of points) of P, and **T** is the interval during which **R** is the NN of q. As an example consider Figure 4.1, 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).

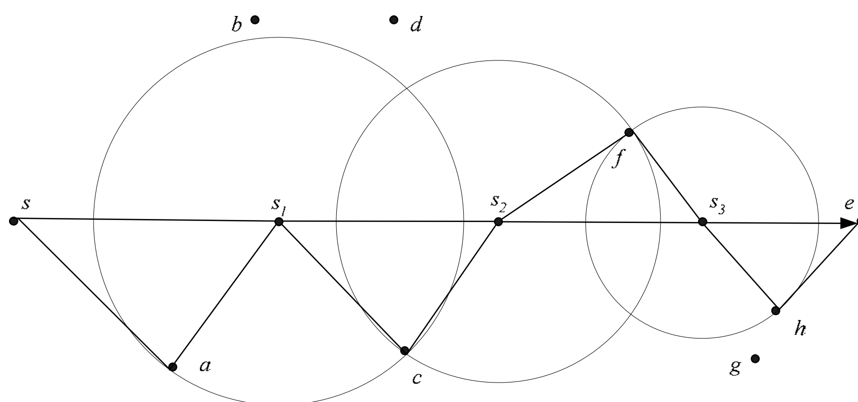


Figure 4.1: Example query

The only existing algorithm for processing CNN queries, proposed in [SR01], employs sampling. 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 4.1).

On the other hand, the TP framework provides a more natural method to process CNN queries: execute a time parameterized NN query at point s, to get the first NN (**R**=$\{a\}$), the validity period of the result (**T** corresponds to point s_1) and the next nearest neighbor (**C**=$\{c\}$). Then, retrieve the TP component (i.e., **C** and **T**) at the points where there is a change in the result (i.e., s_1, s_2, s_3); i.e., the processing of the query involves one (regular) NN search, and four (including the point of origin) computations of the TP component. This technique avoids the drawbacks of sampling, but it is very output-sensitive in the sense that the number of TP queries to be performed grows linearly with the number of split points. Although, these 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. In this section we propose an alternative technique that solves the problem by applying a single query for the whole result. Towards this direction, we first describe some properties of the problem that permit the development of efficient algorithms.

4.1 Definitions and Problem Characteristics

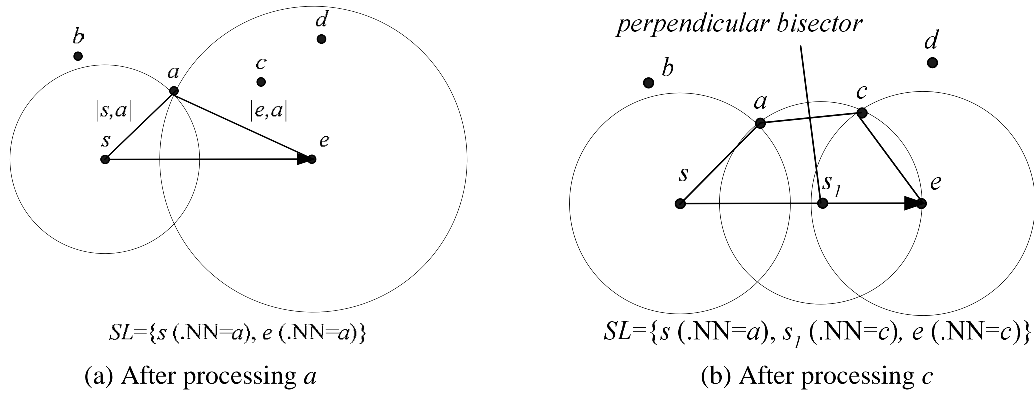
The objective of a CNN query is to retrieve the set of nearest neighbors of a segment $q=[s, e]$ together with the resulting list SL of split points. The starting (s) and ending (e) points constitute the first and last elements in SL. For each split point $s_i \in$ SL ($0 \le i < |\text{SL}|-1$): $s_i \in q$ 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 4.1: Given a split list SL $\{s_0, s_1, \ldots, s_{|\text{SL}-1|}\}$ 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 4.1, consider Figure 4.2a 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)= $|e, a|$. The circle centered at s (e) with radius $|s, a|$ ($|e, a|$) is called the *vicinity circle* of s (e).



SL={*s* (.NN=*a*), *e* (.NN=*a*)}　　　　*SL*={*s* (.NN=*a*), s_1 (.NN=*c*), *e* (.NN=*c*)}

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

Figure 4.2: Updating the split list

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 4.1); hence we ignore b and continue to the next point c.

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 4.2b). 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 4.2a, but not in Figure 4.2b). 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 4.2 (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 4.3, 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 4.2 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).

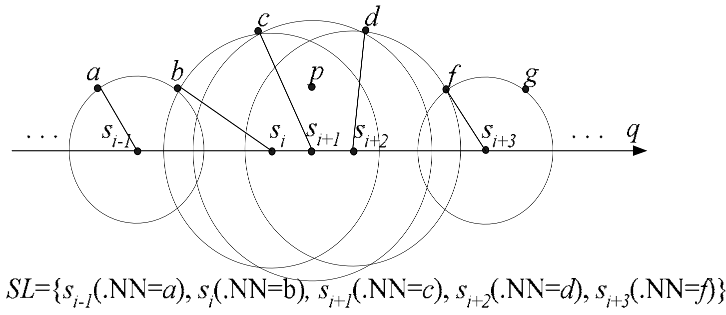


Figure 4.3: Continuity property

Figure 4.4 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 4.3. 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.

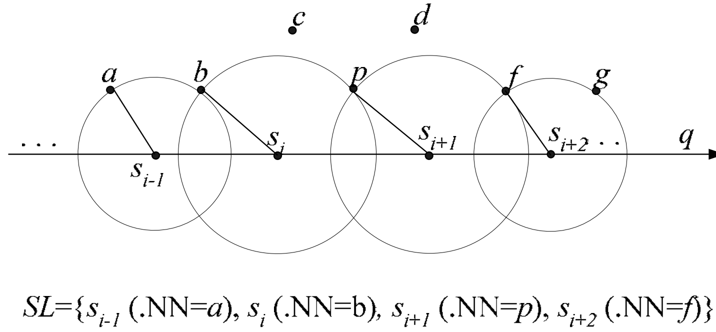


Figure 4.4: After *p* is processed (cont. from Figure 4.3)

The above discussion can be extended to *k*CNN queries (e.g., find the 3 NN for any point on *q*). Consider Figure 4.5, 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*).

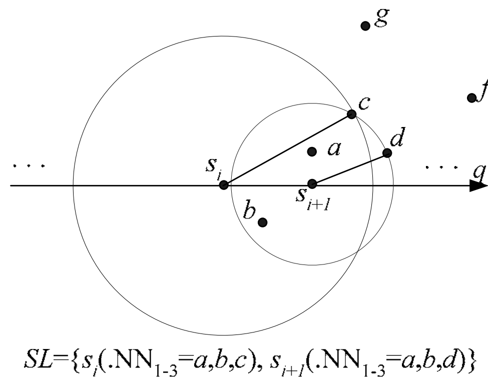


Figure 4.5: Example of *k*CNN (*k*=3)

Lemma 4.1 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). Lemma 4.2, on the other hand, does not apply for *k*>2 (for details see [TPS02]), which means that binary search cannot be employed in this case. The above general methodology can be used for arbitrary dimensionality, where

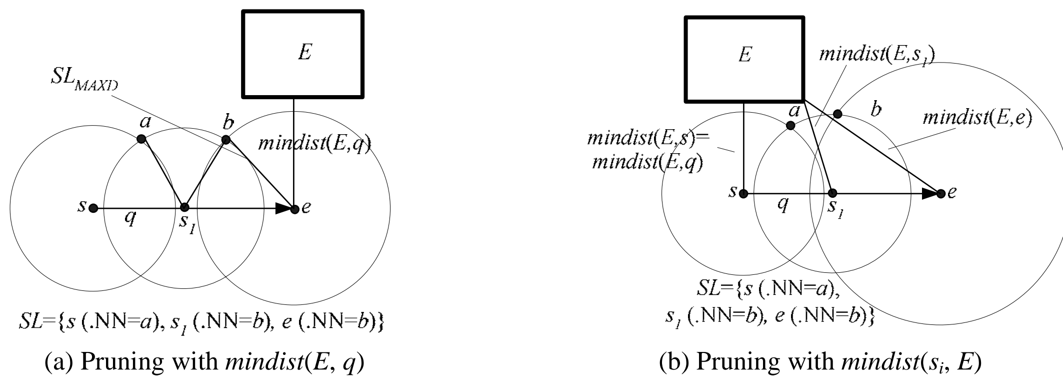
perpendicular bisectors and vicinity circles become perpendicular bisect-planes and vicinity spheres. Its application for processing non-indexed datasets is straightforward, i.e., the input dataset is scanned sequentially and each point is processed, continuously updating the split list. Next we illustrate how the proposed techniques can be used in conjunction with R-trees to accelerate search.

4.2 Query Processing for CNN queries

CNN algorithms are also based on the branch-and-bound paradigm. 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 4.6a 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 4.1 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) Pruning with $mindist(E, q)$ (b) Pruning with $mindist(s_i, E)$

Figure 4.6: Pruning heuristics

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 4.6b, which is similar to Figure 4.6a 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.

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 $dist(s_i, s_i.\text{NN}) > mindist(s_i, E)$.

According to heuristic 2, entry E in Figure 4.6b 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. 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.

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 4.1, 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 4.7, 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.

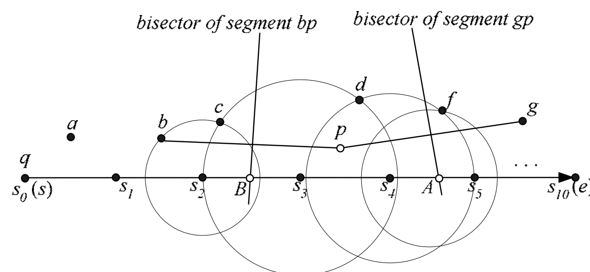


Figure 4.7: Binary search for covered split points

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 4.7) 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 two 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 4.8, 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.

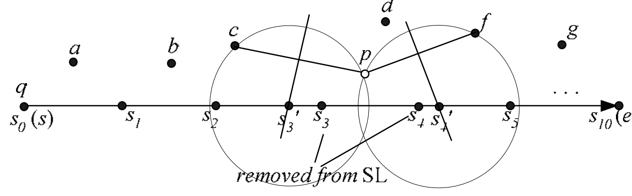
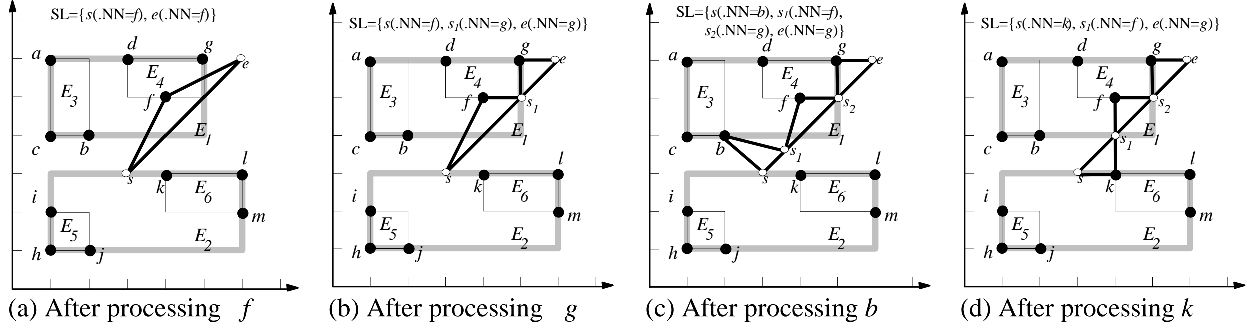


Figure 4.8: After updating the split list

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 2.2. To answer the CNN query [s,e] of Figure 4.9a, the split list SL is initiated with 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 4.9a).



(a) After processing f (b) After processing g (c) After processing b (d) After processing k

Figure 4.9: Processing steps of the CNN algorithm

The next point g covers e and adds a new split point s_1 to SL (Figure 4.9b). 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 4.9c). 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 4.9d). 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]>}.

The proposed algorithms can be extended to support *k*CNN 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 kth (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 4.10 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.

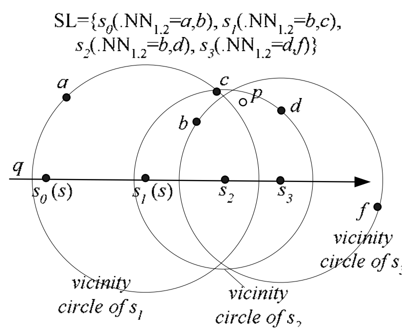
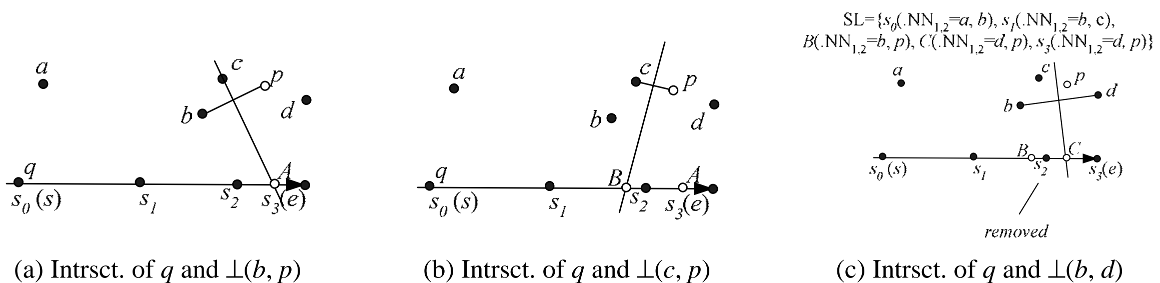


Figure 4.10: Updating SL (k=2) – the first step

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 4.11a), and between q and $\perp(c, p)$ (B in Figure 4.11b) 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.



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

Figure 4.11: Updating SL (k=2) – the second step

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 *k*NN before s_2 is reached. The intersections of q with $\perp(b, c)$ and $\perp(p, c)$ are computed, and since

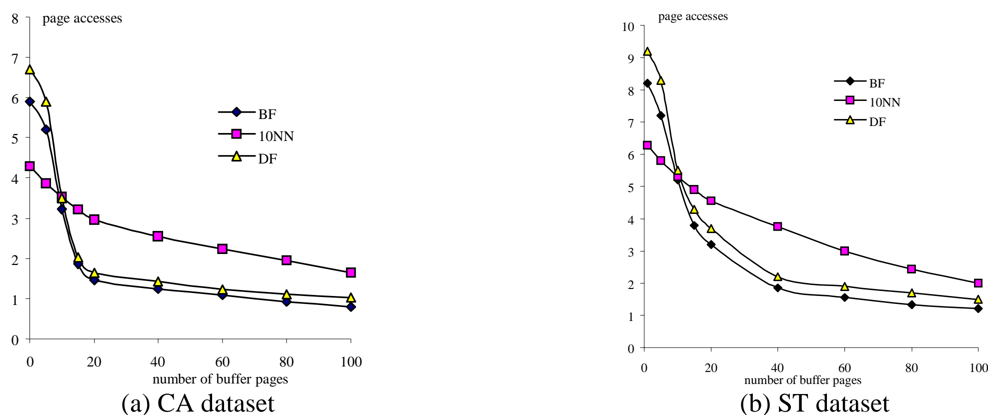
both are outside [B, s_2], the sweeping algorithm terminates without introducing new split point. Similarly, the next interval [s_2, s_3] is handled and a split point C is created in Figure 4.11c. The outdated split points (s_2) are eliminated and the updated SL contains: s_0, s_1, B, C, s_3, whose 2NN are (a,b), (b,c), (b,p), (d,p), (d,p) respectively. The adaptation of the proposed techniques to trajectory nearest neighbor queries (i.e., multiple connected line segments) is straightforward.

5. EXPERIMENTAL EVALUATION

The evaluation consists of two parts: in Section 5.1 we evaluate the overhead of TP NN queries with respect to traditional NN queries, and in Section 5.2 we compare TP NN with the algorithms of Section 4 on continuous nearest neighbor search. For all experiments we use the following real datasets: CA that contains 130K sites and ST that contains the centroids of 2M MBRs representing street segments in California [Web]. The disk size is set to 4K bytes and the maximum fanout of an R-tree node equals 200 entries. Experiments are conducted with a Pentium IV 1Ghz CPU and 256 Mega bytes memory.

5.1 Evaluation of TP NN queries

Recall that processing a TP KNN query is divided into an ordinary KNN query (the first pass), followed by the TP component (the second pass). Figure 5.1 (5.1a for CA dataset and 5.1b for ST dataset) compares the cost of the two passes (for TP 10-NN queries) as a function of number of (LRU) buffer pages. Cost is measured by the average number of disk accesses in performing workloads of 200 dynamic queries. The positions of queries in a workload conform to the distribution of the queried dataset in order to avoid queries in empty space. The query velocities range uniformly in [–0.1, 0.1].



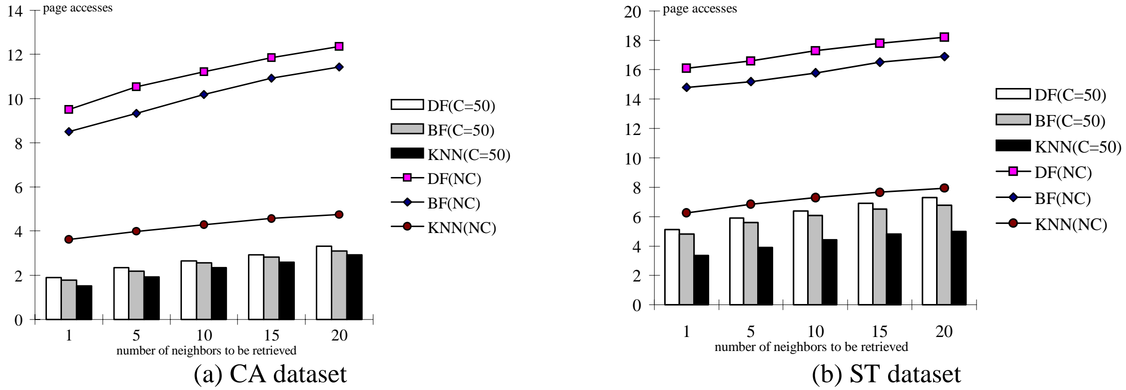
(a) CA dataset (b) ST dataset

Figure 5.1: Comparison of the conventional and the TP component

When there is no buffer, the second pass requires more disk accesses; however, the performance of the second step improves fast even with a very small buffer. This is because the two passes have similar access patterns, and pages loaded for the conventional component are later available for TP processing.

This is further confirmed in Figure 5.2, which shows the cost of a complete TP query (conventional and time-parameterized component) versus that of an ordinary kNN query (costs are shown as a function of k). Notice that, when there is no buffer, a TP query is significantly more expensive than the corresponding kNN query. The addition of a buffer with C=50 pages, reduces this difference

considerably; the cost of a BF TP *k*NN is only 10%-20% higher than that of the regular query. Due to its lower cost, in the sequel we adopt the BF approach for all experiments.

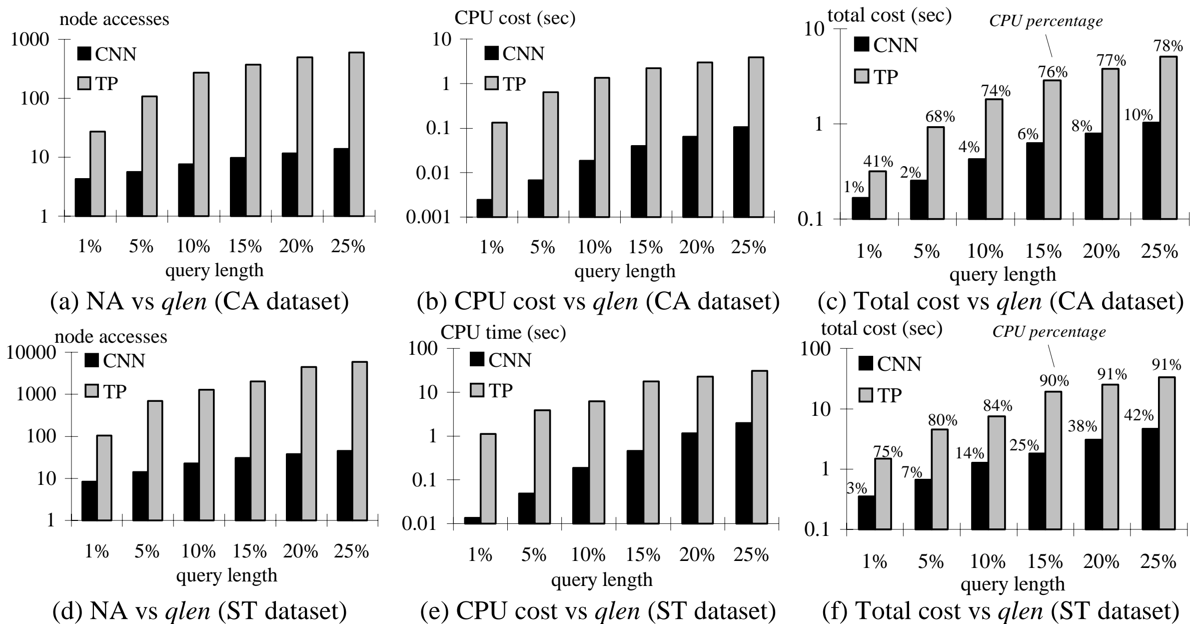


(a) CA dataset (b) ST dataset

Figure 5.2: Comparison of the conventional and the TP component

5.2 Evaluation of CNN queries

The rest of the experiments compare CNN and TP algorithms by executing workloads, each consisting of 200 segment 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. In addition to node/page accesses we include CPU time, which now constitutes an important part of the total cost due to the large number of distance computations. 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 5.3 compares continuous TP NN with CNN processing (NA, CPU time and total cost) as a function of the query length (for k = 5). The first row corresponds to CA, and the second one to ST, dataset. As shown in Figures 5.3a and 5.3d, 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.



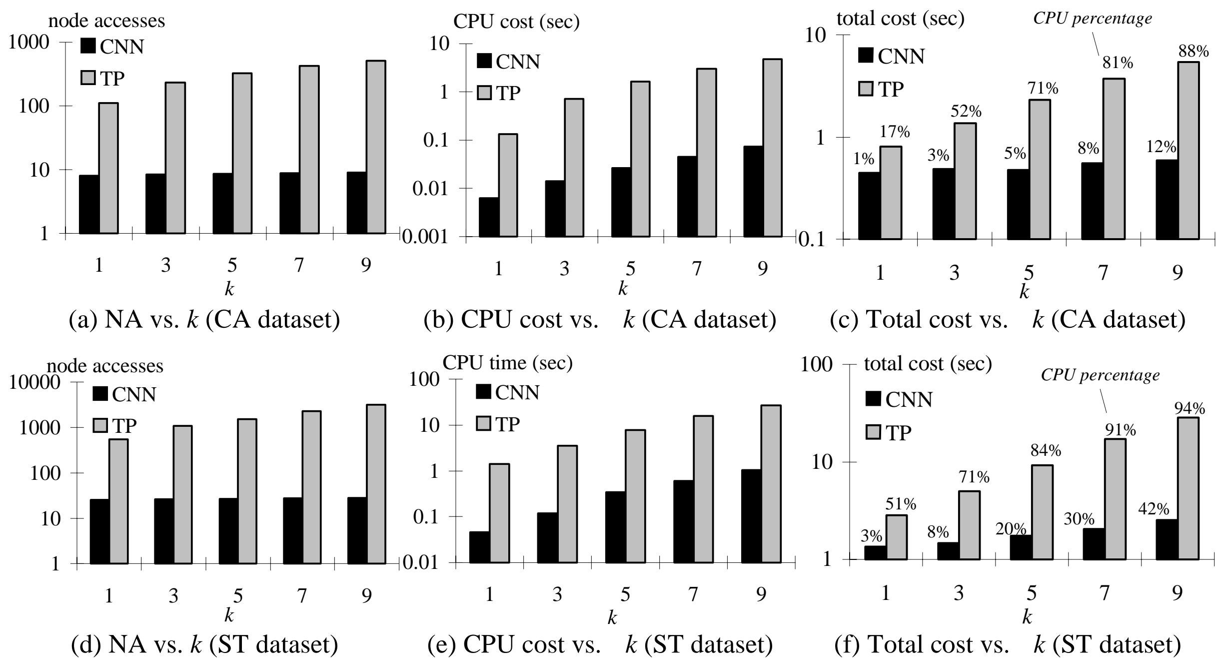
(a) NA vs *qlen* (CA dataset) (b) CPU cost vs *qlen* (CA dataset) (c) Total cost vs *qlen* (CA dataset)

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

Figure 5.3: Performance vs. query length (k=5)

The burden of the large number of queries is evident in Figures 5.3b and 5.3e 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 5.3c and 5.3f 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).

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 5.4, 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 5.4a and 5.4d). 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.



(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 5.4: Comparison with various *k* values (query length=12.5%)

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 5.5 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.

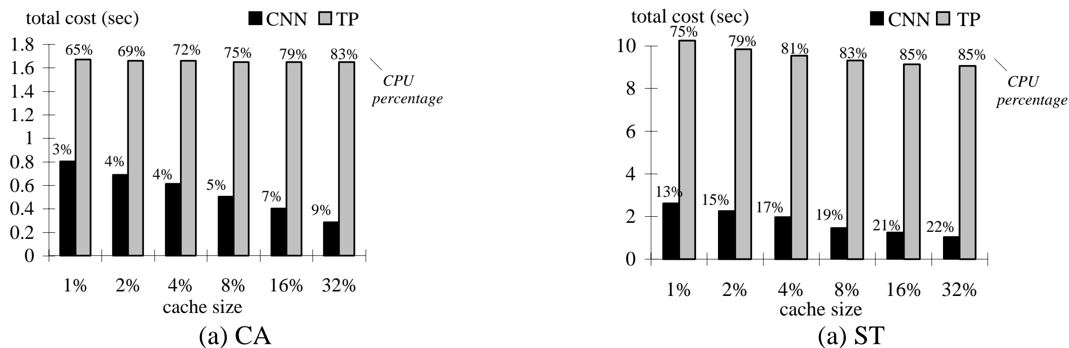


Figure 5.5: Total cost under different cache sizes (*qlen*=12.5%, *k*=5)

6. CONCLUSION

Regular spatial queries are of limited use in dynamic environments, unless the results are accompanied by an expected validity period. In this paper we propose time-parameterized queries, which, in addition to the current nearest neighbor, return the expiry time of the result and the next change. As shown in the experimental evaluation, the extra information is obtained at minimal cost. Then, we apply TP queries for continuous NN search, where the goal is to retrieve all nearest neighbors of an object trajectory together with their validity period. For this problem, we also propose an alternative technique (CNN) that processes the entire trajectory using a single query. CNN outperforms TP significantly (by a factor up to 2 orders of magnitude) especially for long trajectories.

Nevertheless TP queries as standalone methods are crucial for query processing in highly dynamic environments (e.g., mobile communications, weather prediction) where any result should be accompanied by an expiry period in order to be effective in practice. Similarly, CNN queries are essential for 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*").

Given the relevance of the proposed techniques to several spatio-temporal applications, we expect this research to trigger further work in the area. Regarding time parameterized queries, extensions involve additional query types. For example, a TP closest pair (TP CP) query identifies future changes in the closest pairs of objects from two dynamic datasets (e.g., "inform a set of customers about when their nearest cabs will change"). As with TP NN queries, the influence time of a pair of objects (customer, cab) in the TP CP problem depends on the closest pair now. Thus, an efficient definition for $T_{MIN}(E_1,E_2)$ is difficult, because it requires the knowledge of nearest cabs of all customers. Another direction concerns the application of CNN techniques to dynamic datasets. Several indexes have been proposed

for moving objects in the context of spatiotemporal databases [KGT99, 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".

ACKNOWLEDGMENTS

This paper is based on [TP02] and [TPS02] available at: *http://www.cs.ust.hk/~dimitris/*

REFERENCES

[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.

[BEK+98] Berchtold, S., Ertl, B., Keim, D., Kriegel, H., Seidl, T. Fast Nearest Neighbor Search in High-Dimensional Space. *IEEE ICDE*, 1998.

[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.

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

[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.

[KGT99] Kollios, G., Gunopulos, D., Tsotras, V. On Indexing Mobile Objects. *ACM PODS*, 1999.

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

[PM97] Papadopoulos, A., Manolopoulos, Y. Performance of Nearest Neighbor Queries in R-trees. *ICDT*, 1997.

[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.

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

[TPS02] Tao Y., Papadias D., Shen Q. Continuous Nearest Neighbor Search. *VLDB* 2002.

[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.