Evaluating Trajectory Queries Over Imprecise Location Data

Xike Xie[1⋆] Reynold Cheng[2] Man Lung Yiu[3]

[1] Aalborg University, Denmark
xkxie@cs.aau.dk
[2] University of Hong Kong, Pokfulam Road, Hong Kong
ckcheng@cs.hku.hk
[3] Hong Kong Polytechnic University, Hung Hom, Hong Kong
csmlyiu@comp.polyu.edu.hk

Abstract. Trajectory queries, which retrieve nearby objects for every point of a given route, can be used to identify alerts of potential threats along a vessel route, or monitor the adjacent rescuers to a travel path. However, the locations of these objects (e.g., threats, succours) may not be precisely obtained due to hardware limitations of measuring devices, as well as the constantly-changing nature of the external environment. Ignoring data uncertainty can render low query quality, and cause undesirable consequences such as missing alerts of threats and poor response time in rescue operations. Also, the query is quite time-consuming, since all the points on the trajectory are considered. In this paper, we study how to efficiently evaluate trajectory queries over imprecise location data, by proposing a new concept called the u-bisector. In general, the u-bisector is an extension of bisector to handle imprecise data. Based on the u-bisector, we design several novel filters to make our solution scalable to a long trajectory and a large database size. An extensive experimental study on real datasets suggests that our proposal produces better results than traditional solutions that do not consider data imprecision.

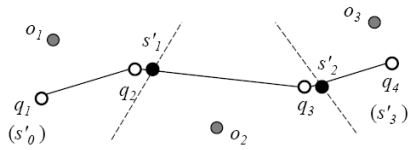
1 Introduction

Given a set P of points, the Trajectory Nearest Neighbor Query (*TNNQ in short*) [1], retrieves the closest object in P for every query point on the given trajectory T. As an example, consider the trajectory $T = \{[q_1, q_2], [q_2, q_3], [q_3, q_4]\}$ and objects $P = \{o_1, o_2, o_3\}$, shown in Figure 1(a). The *TNNQ*'s answer is as Figure 1(b). It means for all points on $[s'_0, s'_1]$, the nearest neighbor is o_1, etc. The *TNNQ* can find applications in location-based service (*LBS in short*), such as "what is the nearest gas station along the travel route".

Unfortunately, the measured location of an object is often imprecise because of: (i) limited resolution of the measure device, (ii) infrequent measurement, (iii) environmental factors. For example, the shipping industries regard safety as their top priority. They hope to identify alerts of potential threats along the route of a vessel in advance, and take appropriate actions if necessary. People in the US and Northern Europe detect the

⋆ This work is done in the University of Hong Kong.

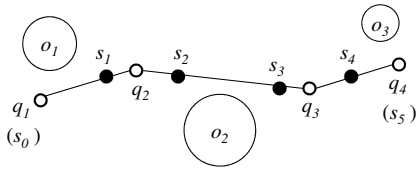
icebergs by remote sensors and satellite imaging [2], which have limited measurement accuracy and frequency. Sensors have limited battery capacity whereas satellite imaging incurs expensive deployment cost. This causes infrequent measurements, rendering the measured location of an object stale. Furthermore, as time passes by, icebergs may move according to the temperature and the ocean current / wind speed. In the *LBS* example, what if the objects being queried are not static but moving constantly (e.g. rescue vehicles positioned by GPS devices)? Again, locations obtained by GPS devices can be contaminated with measurement error, which can be further deteriorated by terrain and climate conditions [3]. Also, the positions could be tracked only periodically due to the limited battery powers [4].



(a) precise objects

Segment	Result
$[s'_0, s'_1]$	o_1
$[s'_1, s'_2]$	o_2
$[s'_2, s'_3]$	o_3

(b) result



(c) imprecise objects

Segment	Result
$[s_0, s_1]$	o_1
$[s_1, s_2]$	o_1, o_2
$[s_2, s_3]$	o_2
$[s_3, s_4]$	o_2, o_3
$[s_4, s_5]$	o_3

(d) result

Fig. 1. Example Trajectory Query

A common way to represent an imprecise location or a moving object is to model the position by an area called *imprecise region* [4,5,6,7,8,9,10]. The possible location of the object is assumed to be within this region. Figure 1(c) shows a query trajectory $T = \{[q_1, q_2], [q_2, q_3], [q_3, q_4]\}$ and some imprecise objects o_1, o_2, o_3. The result (Figure 1(d)) can be represented in a compact way by partitioning the query trajectory into segments such that all locations within the same segment share the same result set. For example, o_2 is *definite nearest neighbor* to the segment $[s_2, s_3]$. On the other hand, o_1 and o_2 are *possible nearest neighbors (***PNN***s)* to the segment $[s_1, s_2]$ because both of them have potential to be the closest object. We define this query as Trajectory Possible Nearest Neighbor Query (*TPNNQ in short*). Note that [1] is a special case of our problem, where the objects being queried are precise points.

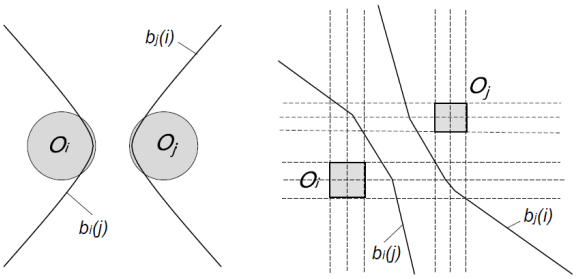
Determining the *TPNNQ* answer can be technically challenging, since the imprecise regions are considered. A simple solution is to replace the imprecise region of each object with a center point (shown as a grey dot), as illustrated in the scenario in Figure 1(a) and (b). The result consists of three segments, each associated with the closest object. For instance, the closest object to location q_2 appears to be o_1 only. The object o_2 is missing from the result. Recall from Figure 1(c) and (d) that o_2 also has possibility to become a closest object to location q_2. This "center simplification" approach causes undesirable consequences such as missing alerts of threats and poor response time in our applications. In the vessel/rescuer example, the ignorance of the imprecise

region could cause potential danger. Thus, it is important to augment each threat with an imprecise region, in order to foresee the worst-case scenario. In the rescuer example, a rescue vehicle seemingly close to / far from the travel path may be actually far from / close to it. Thus, it would take longer time to respond. It is better to call up all rescuers likely to be the closest, in order to handle the emergency as soon as possible.

Another attempt to simplify the problem is to use a "sampling approach", which considers positions at every fixed length on the query trajectory, and compute the potential nearby objects at each position. However, if the sampling rate is high, it incurs a huge computation cost; on the other hand, a low sampling rate can result in many answers missing. Notice that a query trajectory consists of infinite number of possible locations, and it is not easy to determine the sampling rate. As shown in Figure 1(c), the result set changes only at a few positions (s_1, s_2, s_3, s_4). It is not clear how to determine the correct sampling rate to in order to get these answers. In fact, our experimental results show that replacing imprecise regions with points or sampling the trajectory cannot provide an accurate solution. Hence, we develop a solution that can accurately compute a trajectory query on imprecise objects.

The techniques of [1] cannot be readily applied to evaluate *TPNNQ*. [1]'s idea is to use the (perpendicular) *bisectors* of every pair of points to derive the query answer. For example, in Figure 1(a), the point s'_1 is the intersection between the query trajectory and the bisector of objects o_1 and o_2, which are shown as dashed lines. Similarly, s'_2 is derived by o_2 and o_3's bisector. However, the bisector, which forms the basis of [1], is limited to precise points.

We extend the concept of "bisectors" to support imprecise objects, called **u-bisector**. Figure 2 illustrates the corresponding u-bisectors for circular and rectangular imprecise regions. From this figure, we can see that the u-bisector is not a straight line anymore. It becomes a pair of lines, which partition the domain space into three parts: (1) the left area, containing points q where O_i is absolutely closer to q than O_j; (2) the right partition, consisting of points q' where O_j is absolutely closer to q' than O_i; and (3) the middle part, having points q'' where both O_i and O_j can be the nearest to q''. We demonstrate how to use conceptually the intersection points of the query trajectory and the u-bisectors to answer a trajectory query.



(a) circular imprecise region (b) rectangular imprecise region

Fig. 2. u-bisector for imprecise regions.

In practice, it is expensive to compute the intersections points between the query trajectory and the u-bisectors. As shown in Figure 2, these u-bisectors can be hyperbolic curves (Figure 2(a)), or segments of straight lines/curves (Figure 2(b)). Even for one u-bisectors, they can intersect the query trajectory at more than one point. We first design a *Basic* solution, which answers the query in $O(ln^2logn)$ (*n:database size; l:trajectory length*). To make our solution scalable to large datasets and long trajectories, we design a filter-refinement framework. In the filtering phase, *candidate objects* that may be the closest to each answer line-segment are obtained. In the refinement phase, we develop a novel technique called *ternary decomposition*, which can derive the final answers accurately. We show theoretically and experimentally that our solution is efficient and scalable. It is also more accurate than the center simplification and the sampling approaches. We assume the imprecise regions are of circular shapes for simplicity. Actually, our method is also general to other shaped objects. It would not be discussed due to page limitations.

The rest of this paper is as follows. In Section 2 we discuss the related work. Section 3 defines the problem and a basic solution based on the u-bisectors. We present our solution framework in Section 4. The filtering and refinement phases are described in Sections 5 and 6. In Section 7 we present our experiment results. Section 8 concludes.

2 Related Work

Nearest neighbor (NN) query for moving query points is a well studied topic [11] [12] [13] [1]. These works focus on reducing the computational cost at the server. Among these works, there are two major categories.

The first category does not require the user's entire trajectory in advance [11] [12] [13], but processes the query online (multiple times) based on the user's moving location. In [11], the authors propose sampling techniques to answer the moving *NN* query. They study how to calculate the upper-bound distance within which the moving point does not issue a new query to the server. [12] [13] use validity region and validity time for the query answer of moving points. They use Voronoi cells to represent the validity region. The query answer becomes invalid if the validity time is expired or the user leaves the validity region.

The second category assumes that the user's trajectory is known in advance. It evaluates the query once only [1] [4]. In our application, the trajectory, such as sailing routes, is known in advance. Thus, we elaborate the second category in details. In [1], the route of the query point is split into sub-line-segments, such that the NN answer within the same sub-line-segment remains unchanged. A perpendicular bisector $\perp(p_i, p_j)$ between two points p_i and p_j is used to partition the trajectory query into two sub-trajectories, one being definitely closer to p_i and the other being definitely closer to p_j. However, this technique is not applicable to our problem on imprecise location data. As shown in Figure 1, some segments like $[s_1, s_2]$ can have multiple *PNN*s and it is challenging to derive them.

The bisector for imprecise objects has been addressed by a few works recently [5] [6] [7]. They use bisectors to determine the dominance relationship between objects. Our work is different because we consider a query trajectory, but not a query

object. For the trajectory, our solution is capable of answering the query for every point it.

The paper [4] is closely related to our problem. It also uses an imprecise region to model the location of an object and compute the object closest to a given query segment. Unlike our work, [4] only computes the answer for segments with the definite nearest neighbor, such as $[s_0, s_1]$ in Figure 1. It did not study how to compute objects that might be the closest, for some segment like $[s_1, s_2]$ in Figure 1. Furthermore, their method scans the entire database to answer the query, thus it is not very scalable to data volumes.

3 Problem and Preliminaries

In this section, we describe the query semantics in Section 3.1. We introduce the u-bisector in Section 3.2 and propose a basic method in Section 3.3.

3.1 Problem Setting

We first introduce the definition of *PNNQ* (studied in [14]), which is used to define *TPNNQ*, the query studied in this paper. Let q be a point, and let O_i be an imprecise object from an object set O. We use $dist_{min}(q, O_i)$ and $dist_{max}(q, O_j)$ to denote the minimum and maximum distances of object O_i from q, respectively.

Definition 1. ***Possible Nearest Neighbor Query (PNNQ):*** *Given a set of imprecise objects O and a query point q, $O_i \in PNNQ(q)$, if $\nexists O_j \in O$, such that $dist_{max}(q, O_j) < dist_{min}(q, O_i)$.*

In Figure 1(c), $PNNQ(q_2) = \{O_1, O_2\}$ implies that either O_1 or O_2 could be the *NN* of the query point q_2. A query trajectory $\mathcal{T}$ can be represented by a set of query line-segments $\mathcal{T} = \{L_1, ..., L_l\}$, where L_i is a query line-segment. For a query point q, whose trajectory is $\mathcal{T}$, the *trajectory possible nearest neighbor query* (*TPNNQ*) returns PNNs for all the points in $\mathcal{T}$. In other words, the query returns $\{\langle q, PNNQ(q)\rangle\}_{q\in\mathcal{T}}$. If the connected points on the trajectory have the same PNNs, we could merge them into a segment.

Definition 2. ***Trajectory Possible Nearest Neighbor Query (TPNNQ):*** *Given a set of imprecise objects O and a query trajectory $\mathcal{T}$, the answer for the $TPNNQ$ query is a set of tuples $R = \{\langle T_i, R_i\rangle | T_i \subseteq \mathcal{T}, R_i \subseteq O\}$, where PNNQ(q) $= R_i(\forall q \in T_i)$.*

In other words, the *TPNNQ* splits $\mathcal{T}$ into a set of consecutive segments $\{T_1, T_2, ..., T_t\}$. T_i is a sub-trajectory of $\mathcal{T}$. For $\forall q \in T_i$, q has the same possible nearest neighbors (PNNs), then we call each T_i a ***validity interval***. The connection point of two consecutive segments, say T_i and T_{i+1}, is called ***turning point***, which indicates the change of *PNNQ* answers. An example for a *TPNNQ* over three imprecise objects $\{O_1, O_2, O_3\}$ is shown in Figure 1(c). The trajectory query $\mathcal{T}(s, e)$ is split into 5 pieces of segments. Also, point s_1 is the turning point for $T(s_0, s_1)$ and $T(s_1, s_2)$.

Observe that there are two major differences between the results on imprecise objects and precise objects. Comparing Figures 1 (c) and (a): (1) the *imprecise case* could

have more tuples (5 compared to 3); (2) a query point in *imprecise case* might return a set of *PNN*s instead of a single answer.

Thus, the $TPNNQ$ can be answered by finding the *turning points*. Then, how to derive the *turning points* on a trajectory, given a set of imprecise objects? To address that, we first investigate the u-*bisector*, for imprecise objects. In general, the u-*bisector* splits the domain into several parts, such that query points on different parts could have different PNNs. Then, the *turning points* can be evaluated by finding the intersections of the u-*bisectors* and the query trajectory. Next, we discuss the u-bisector.

3.2 u-bisector

Definition 3. *Given two imprecise objects O_i and O_j, their u-**bisector** consists of two lines: $b_i(j)$ and $b_j(i)$. The u-**bisector half** $b_i(j)$ is a set of points satisfying*

$$b_i(j) = \{z : dist_{max}(z, O_i) = dist_{min}(z, O_j)\} \quad (1)$$

The curve $b_i(j)$ splits the domain into two *half-spaces*: $H_i(j)$ and $\overline{H_i(j)}$, where $H_i(j)$ is the half closer to O_i and $\overline{H_i(j)}$ is the complementary half.An example is shown in Figure 3. Thus we have:

$$H_i(j) = \{z : dist_{max}(z, O_i) \leq dist_{min}(z, O_j)\} \quad (2)$$

$$\overline{H_i(j)} = \{z : dist_{max}(z, O_i) > dist_{min}(z, O_j)\} \quad (3)$$

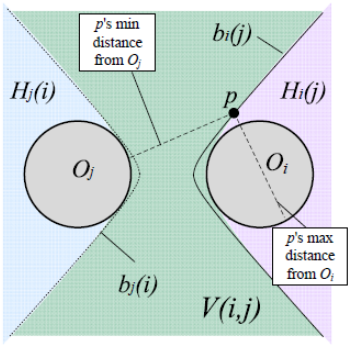


Fig. 3. u-bisector

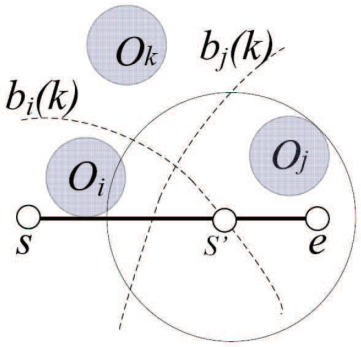


Fig. 4. Verification

Generally speaking, the u-bisector half $b_i(j)$ is a curve in the domain space. If a query point $q \in H_i(j)$, q must take O_i as its nearest neighbor certainly. The u-bisector halves $b_i(j)$ and $b_j(i)$ separate the domain into three parts: $H_i(j)$, $H_j(i)$, and $V(i, j)$, where

$$V(i, j) = \overline{H_i(j)} \cap \overline{H_j(i)} \quad (4)$$

Notice that $V(i, j) = V(j, i)$. If O_i and O_j are degenerated into precise points, $V(i, j)$ becomes 0. Also, $b_i(j) = b_j(i)$, which becomes a straight line.

If a query line-segment is totally covered by $V(i, j)$ or $H_i(j)$, it does not intersect with $b_i(j)$. Otherwise, the intersections split the line-segment into several parts. Different parts might correspond to different PNNs, as they are located on different sides of $b_i(j)$.

For circular imprecise objects, it is easy to derive the closed form equations of the u-bisector and evaluate the analytical solution for the intersection points. The number of intersections is at most 2, since the quadratic equation has at most 2 roots (see Appendix A.1). Next, we present the basic method based on the analysis of the u-bisector's intersections. We focus our discussion on two dimension location data.

3.3 Basic Method

From Definition 2, the *TPNNQ* could be answered by deriving the *turning points*, which are intersections of the query trajectory and the u-bisectors. A u-bisector is constructed by a pair of objects. Given a set O of n objects, there can be C_2^n u-bisectors. The *Basic* method is to check the intersections of the query trajectory with the C_2^n u-bisectors. The intersections can be found by evaluating the equation's roots in Appendix A.1. Here we use *FindIntersection(.)* (in *Step 5*) to represent the process.

However, not all of the bisectors intersect with the trajectory. Even if they intersect, not all of the intersections are qualified as *turning points*. Thus, we need a "*verification*" process to exclude those unqualified intersectinos. For example, in Figure 4, the u-bisector half $b_i(k)$ intersect with $[s, e]$ at s'. For an arbitrary point $q \in [s, e]$, either O_i or O_j is closer to q than O_k, since $[s, s'] \in H_i(k)$ and $[s', e] \in H_j(k)$. Then, O_k is not PNN for $p \in [s, e]$, and s' is not a qualified *turning point*.

We use the $s_{i\vdash j}$ to represent an intersection created by $b_i(j)$ ($s_{i\vdash j} = b_i(j) \cap L$), and $s_{i\dashv j} = b_j(i) \cap L$. In other words, $s_{i\vdash j}$ can be understood as $PNNQ(q)$ answer that turns from containing O_i to both O_i and O_j, if q moves from $H_i(j)$ to $\overline{H_i(j)}$ So, O_i should definitely be $s_{i\vdash j}$'s PNN, while O_j is not. This can be implemented by issuing a $PNNQ$. Thus, we can use this for verification.

Algorithm 1 Basic

```
1: function BASIC(Trajectory 𝕋)
2:     for all line-segment L ∈ 𝒯 do
3:         for i = 1 ... n do                          ▷ consider object O_i
4:             for j = i + 1 ... n do                  ▷ consider object O_j
5:                 ℐ = FindIntersection(L, O_i, O_j);
6:                 Verify ℐ and delete unqualified elements;
7:     Evaluate PNNs for each Interval and Merge two successive ones if they have same PNNs;
```

In Algorithm 1, suppose *Step 5* can be done in time β, which is a constant if we call Appendix A.1. *Step 6* can be finished in $O(\log n)$. Suppose $\mathcal{T}$ contains l line-segments, *Basic*'s total query time is $O(l\ n^2(\log n + \beta))$. In later sections, we study several filters which can effectively prune those unqualified objects, which cannot be PNN for any point on the trajectory, in order to reduce the complexity.

4 Solution Framework

In this section, we propose the framework of the $TPNNQ$ algorithm, which follows the *filter-refinement* framework. We assume an R-tree $\mathbb{R}$ is built on the imprecise objects

O and it can be stored in the main memory, as the storage capabilities increase fast in recent years.

Framework In implementation, we organize the trajectory $\mathcal{T} = \{L_1, L_2, ..., L_l\}$ by constructing a binary tree $\mathbb{T}(\mathcal{T})$. Each binary tree node $T_i = \{L_1, ..., L_{l'}\}$ has two children: $T_i.left = \{L_1, ..., L_{\lfloor \frac{l'}{2} \rfloor}\}$ and $T_i.right = \{L_{\lceil \frac{l'}{2} \rceil}, ..., L_{l'}\}$. We show an example of $\mathcal{T} = \{L_1, L_2, L_3\}$'s trajectory tree in Figure 5(a).

The data structure for each binary tree node T_i is a triple: $T_i = \langle L, MBC, Guard \rangle$. L is a line-segment if T_i is a leaf-node and $NULL$ otherwise. MBC is the minimum bounded circle covering T_i; it is $NULL$ for leaf-nodes. $Guard$ is an entry which has minimum maximum distance to T_i. As we describe later, the entry can be either an R-tree node or an imprecise object. The $Guard$s are not initialized until the processing of $TPNNQ$. Since $\mathcal{T}$ contains l line-segments, the trajectory tree $\mathbb{T}(\mathcal{T})$ could be constructed in $O(l \log l)$ time.

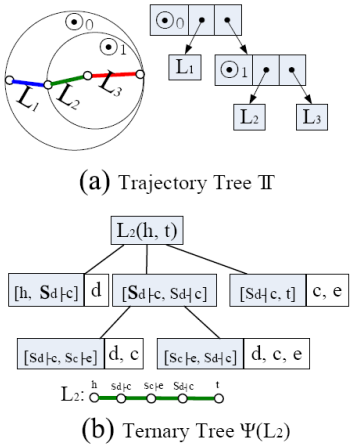


Fig. 5. Trajectory Tree $\mathbb{T}(\mathcal{T})$ and Ternary Tree $\Psi(L_2)$

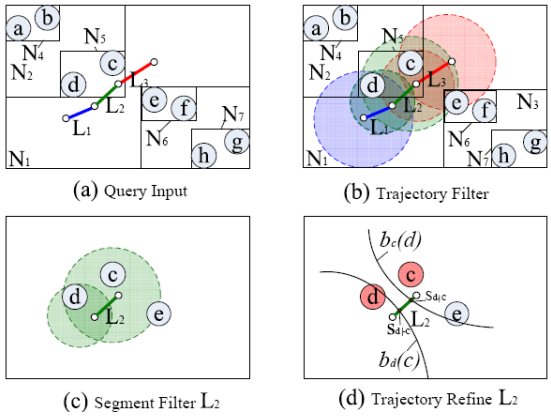


Fig. 6. TPNNQ

Given a constructed trajectory tree $\mathbb{T}$ and an R-tree $\mathbb{R}$, the $TPNNQ$ algorithm, shown in Algorithm 2, consists of two phases. Phase I is the filtering phase, which includes two filters: *Trajectory Filter* and *Segment Filter*. *Trajectory Filter* is to retrieve a set of candidates from the database (*Step 3*). *Segment Filter* prunes away unqualified objects for each $L_i \in \mathcal{T}$ (*Step 4*). Phase II is to evaluate all the *validity interval*s and *turning point*s for each line-segment of the trajectory (*Step 5*). Then, we scan the derived *validity interval*s once and merge two consecutive *validity interval*s if they belong to different line-segments but have the same set of PNNs (*Step 7*).

Example of TPNNQ Suppose an R-tree built on objects $O = \{a, b, c, d, e, f\}$ and a trajectory $\mathcal{T} = \{L_1, L_2, L_3\}$, as shown in Figure 6(a). We use *Trajectory Filter* to derive $\mathcal{T}$'s trajectory filtering bound, as shown by shaded areas in Figure 6(b). The objects $\{c, d, e, f\}$ overlapping with the trajectory filtering bound are taken as candidates. During the process, object d is set to be L_2's $Guard$, and stored in the trajectory tree.

Algorithm 2 TPNNQ

```
1: function TPNNQ(Trajectory 𝕋, R-tree ℝ)
2:     let φ be a list (of candidate objects);
3:     φ ←TrajectoryFilter(𝕋, ℝ);                          ▷ Section 5.1
4:     for all line-segment L_i ∈ 𝒯 do                     ▷ 𝒯 = {L_i}_{i≤l}
5:         φ_i ← SegmentFilter(L_i, φ);                    ▷ Section 5.2
6:         {⟨L, R⟩}_i ← TernaryDecomposition(L_i, φ_i);    ▷ Section 6
7:     {⟨T_i, R_i⟩}_{i=1}^t ← Merge(∪_{i=1}^l {⟨L, R⟩}_i);
```

The *segment filter* is applied for each line-segment in $\mathcal{T}$. Taking $L_2(h,t)$ as an example, the segment filtering bound is shown as Figure 6(c), where f is excluded from L_2's candidates. Because f does not overlap with the filter bound.

In the refinement phase, we call the routine *TernaryDecomposition* to derive the *turning points*. We find the u-bisector halves $b_d(c)$ and $b_c(d)$ intersects with L_2 at $s_{d\vdash c}$ and $s_{d\dashv c}$, respectively. L_2 is split into three sub-line-segments $[h, s_{d\vdash c}]$, $[s_{d\vdash c}, s_{d\dashv c}]$, and $[s_{d\dashv c}, t]$. Meanwhile, the construction of a ternary tree $\Psi(L_2)$ starts, in Figure 5(b). The root node of $\Psi(L_2)$ derives three children correspondingly.

Then, we repeat the above process for each of the three, recursively. Finally, the process stops and we get a ternary tree $\Psi(L_2)$, in Figure 5(b). Observed from $\Psi(L_2)$, the degree of a ternary tree node is at most 3, since a line-segment is split into at most 3 sub-line-segments, as shown in Section 3. The query result on L_2 can be fetched by traversing the leaf-nodes of $\Psi(L_2)$. Then, we have: $TPNNQ(L_2) = \{\langle[h, s_{d\vdash c}], \{d\}\rangle, \langle[s_{d\vdash c}, s_{c\vdash e}], \{c, d\}\rangle, \langle[s_{c\vdash e}, s_{d\dashv c}], \{c, d, e\}\rangle, \langle[s_{d\dashv c}, t], \{c, e\}\rangle\}$. Similarly, the results of L_1 and L_3 can also be evaluated. By merging them we get the answer of $TPNNQ(\mathcal{T})$. After we get the query answer, $\mathbb{T}$ and Ψ are deleted.

In the following sections, we study the *Trajectory Filter* in Section 5.1 and *Segment Filter* in Section 5.2. The refinement step is shown in Section 6.

5 Filtering Phase

The trajectory query consists of a set of consecutive query line-segments. An intuitive way is to: (1) decompose the trajectory into several line-segments; (2) for each line-segment L_i, access R-tree to fetch candidates. Then, apply `TernaryDecomposition` to construct a ternary tree $\Psi(L_i)$ to evaluate its *validity intervals* and *turning points*. We call this method TP-S, which incurs multiple R-tree traversals.

Meanwhile, two consecutive line-segments might share similar $PNNs$. Also, if two line-segments are short, they could even be located within the same *validity interval*. So, considerable efficiency would be saved if the R-tree traversal for each line-segment inside the trajectory could be shared.

5.1 Trajectory Filter

To save the number of R-tree node access, we design Algorithm 3 as the *Trajectory Filter* to retrieve the candidates for the entire trajectory. We start Algorithm 3 by maintaining a heap in the ascending order of maximum distance between an entry to

Algorithm 3 TrajectoryFilter

```
1: function TRAJECTORYFILTER(Trajectory tree 𝕋, R-tree ℝ)
2:     let φ be a list (of candidate objects);
3:     Construct a min-dist Heap ℋ;
4:     push_heap(ℋ, root(ℝ), 0);
5:     while ℋ is not empty do
6:         E ← deheap(ℋ);
7:         if E is a non-leafnode of ℝ then
8:             for E's each child e do
9:                 if isProbable(𝕋, e) then
10:                    push_heap(ℋ, e, dist_max(e, 𝕋));
11:        else
12:            if isProbable(𝕋, E) then
13:                insert E into φ;
14:    return φ;
```

a trajectory tree node T_i's center. If T_i is leaf-node, it is a line-segment. T_i's center is its mid-point. Otherwise, the center is $MBC(T_i)$'s circle center. Then, the top element is popped to test if it/its children could be qualified to be the candidate objects. The process is repeated until the heap is empty.

To determine if an entry is qualified or not, we use Algorithm 4. Let T_i be a $\mathbb{T}$'s node and G be $T_i.Guard$. G is initialized in *Step 4*. Given an R-tree node E, if $T_i \subseteq H_G(E)$, then $\forall O_j \in E$, O_j cannot be T_i's PNNs. Thus, $\forall O_j \in E$ can be rejected. This helps pruning those unqualified objects in a higher index level.

Algorithm 4 isProbable

```
1: function ISPROBABLE(Trajectory tree node T_i, R-tree node E)
2:     Let G be MBC(T_i).Guard;                          ▷ Initialize Guard Obj.
3:     if G is NULL then
4:         G ← E;
5:     elseif T_i is leaf-node and T_i ∈ H_G(E) then
6:         return false;                                 ▷ If E can be rejected, return false
7:     elseif T_i is non-leaf-node and MBC(T_i) ∈ H_G(E) then
8:         return false;                                 ▷ If E can be rejected, return false
9:     elseif dist_max(E, MBC(T_i).c) < dist_max(G, MBC(T_i).c) then
10:        G ← E;                                        ▷ If G can be updated
11:    if T_i is not leaf-node then
12:        return isProbable(T_i.left, E) || isProbable(T_i.right, E);
13:    else return true;
```

In order to check whether E can be rejected, we consider two cases: (i) if T_i is a leaf-node; (ii) if T_i is a non-leaf-node.

(i) When T_i is a leaf-node, we can draw a pruning bound to test whether E is qualified. If we denote $\circledcirc(c, G)$ as a circle centered at c and internally tangent with object G, and $\odot(c, r)$ as a circle centered at c with radius r, then:

$$\circledcirc\,(c, G) = \odot(c, dist_{max}(c, G)) \tag{5}$$

The pruning bound is written as: $\circledcirc(s, G) \cup \circledcirc(e, G)$. The correctness is guaranteed by Lemma 1.

Lemma 1. *Given two imprecise objects O_i, O_j and a line-segment $L(s, e)$, O_j can not be $p \in L$'s PNN if O_j does not overlap with $\circledcirc(s, O_i) \cup \circledcirc(e, O_i)$.*

Proof. To judge if O_j is L's PNN, we first prove it is sufficient to check L's two end points s and e. Then, we show how the pruning bound can be derived by s and e.

Since $b_i(j)$ is a hyperbola half, it has at most two intersections with an arbitrary line. Thus, $H_i(j)$ is convex [15]. So, if s and e are in $H_i(j)$, $p \in L$ must be in $H_i(j)$. It means if O_j is not s and e's PNN given O_i, it is not a PNN for all the points on L.

Next,

$$s \in H_i(j) \Leftrightarrow dist_{max}(s, O_i) < dist_{min}(s, O_j)$$
$$\left.\begin{array}{l} \Leftrightarrow \circledcirc(s, O_i) \cap O_j = \emptyset \\ \circledcirc(e, O_i) \cap O_j = \emptyset \end{array}\right\} \Leftrightarrow O_j \bigcap \circledcirc(s, O_i) \cup \circledcirc(e, O_i) = \emptyset$$

So, the lemma is proved.

If another object O_j does not overlap with the pruning bound defined by Lemma 1, it can not be the PNN of any $p \in L$, since O_i will be always be closer. We also use Lemma 1 as the base to derive other pruning bounds in Section 6.

(ii) When T_i is a non-leaf-node, if $MBC(T_i) \in H_G(E)$, then E can be rejected from candidates. Since $MBC(T_i) \in H_G(E)$, T_i must be in $H_G(E)$. In other words, Equation 6 is satisfied when the condition below is true:

$$dist_{max}(MBC(T_i), G) \leq dist_{min}(MBC(T_i), E) \tag{6}$$

Since $MBC(T_i)$ is a circle, Equation 6 can be rewritten as ($\odot.c$ and $\odot.r$ are $\odot$'s center and radius):

$$dist_{max}(MBC(T_i).c, G) + MBC(T_i).r \leq dist_{min}(MBC(T_i).c, E) - MBC(T_i).r$$

5.2 Segment Filter

After the *trajectory filtering* step of $TPNNQ$, we get a set ϕ of candidates. Before passing ϕ to each line-segment L_i in the *refinement* phase, we perform a simple filtering process to shrink ϕ into a smaller set ϕ_i for L_i. Notice that while deriving the trajectory tree $\mathbb{T}(\mathcal{T})$, we also derive an object called "Guard" for each node T_i. Then, for a $\mathcal{T}$'s line-segment $L_i(s_i, e_i)$, we can reuse the "Guard" O_g to build the pruning bound. According to Lemma 1, the pruning bound is set to $\circledcirc(s_i, O_g) \cup \circledcirc(e_i, O_g)$. An example is shown in Figure 2(c), where the pruning bound for $L_2(h, t)$ is $\circledcirc(h, d) \cup \circledcirc(t, d)$. After that, we get ϕ_i. Empirically, the pruned candidates set ϕ_i is much smaller than ϕ.

6 Trajectory Refinement Phase

For trajectory $\mathcal{T}$, the refinement is done by applying Algorithm 5 *Ternary Decomposition* for each line-segment $L_i \in \mathcal{T}$. Essentially, Algorithm 5 is to construct a *ternary tree* $\Psi(L_i)$ for L_i.

6.1 Trajectory Refinement

Ψ is constructed in an iterative manner. At each iteration, we select two objects from the current candidate set ϕ_{cur} as seeds to divide the current line-segment L_{cur} into two/three pieces.

To split L_i, we have to evaluate a feasible u-bisector, whose intersections with L_i are *turning points*. Then, to find the u-bisector, we might have to try $\frac{C(C-1)}{2}$ pairs of objects, $C = |\phi_i|$. In fact, the object with minimum maximum distance to L_i, say O_1, must be one PNN. The correctness is shown in Lemma 2. Thus, it is often that the *turning points* on L_i is derived by O_1 and another object among the C candidates. So, in Algorithm 5, the candidates are sorted first.

Lemma 2. *If $S = \{O_1, O_2, ...\}$ are sorted in the ascending order of the maximum distance to the line-segment L, then $O_1 \in TPNNQ(L)$.*

Proof. Suppose p is a point of L, such that $dist_{max}(p, O_1) = dist_{max}(L, O_1)$. If O_1 is definitely one PNN of $p \in L$, O_1 must be one PNN of L. Thus, it is sufficient to show $O_1 \in PNNQ(p)$.

To show $O_1 \in PNNQ(p)$ is equivalent to prove $dist_{min}(p, O_1) < dist_{max}(p, O_i)$ $(O_i \in S)$. Then, it is sufficient to show $dist_{max}(p, O_1) < dist_{max}(p, O_i)(O_i \in S)$, as $dist_{min}(p, O_1) < dist_{max}(p, O_1)$.

Notice that $dist_{max}(p, O_i)$ must be no less than $dist_{max}(L, O_i)$. Then,

$$\begin{aligned} dist_{max}(p, O_1) = dist_{max}(L, O_1) &\leq dist_{max}(L, O_i)(O_i \in S) \\ &\leq dist_{max}(p, O_i)(O_i \in S) \end{aligned}$$

So, O_1 definitely belongs to $TPNNQ(L)$.

Algorithm 5 TernaryDecomposition

```
1:  function TernaryDecomposition(Segment L(s,e), Candidates set φ_cur^[L])
2:      Sort φ_cur^[L] in the ascending of maximum distance to L
3:      for i = 1 ... |φ_cur| do                          ▷ consider object O_i
4:          for j = i + 1 ... |φ_cur| do                  ▷ consider object O_j
5:              I = FindIntersection(L, O_i, O_j);
6:              Verify I and delete unqualified elements;
7:              if |I| ≠ 0 then
8:                  Use I to split L(s,e) into |I| + 1 pieces
9:                  for each piece of line segment L^i do
10:                     Use Lemma 3, 4, and 5 to derive pruning bound B_i
11:                     φ_cur^[L^i] ← B_i(φ_cur^[L])
12:                 release φ_cur^[L]
13:                 for each piece of line segment L^i do
14:                     TernaryDecomposition(L^i, φ_cur^[L^i])
```

Then, L_{cur} is split into 2 (or 3) pieces (or children). For L_{cur}'s children L^i, we derive a pruning bound B_i for L^i and select a subset of candidates from ϕ_{cur} , as shown in *Step 9* to *Step 12*.

Notice that for each leaf-node L^i of the ternary tree $\Psi(L(s,e))$, L^i's two end points must be s, e, or the *turning points* on L. If we traverse Ψ in the pre-order manner, any two successively visited leaf-nodes are the successively connected *validity intervals* in L. Suppose we have m *turning points*, we would have $m+1$ *validity intervals*, which corresponds to $m+1$ Ψ's leaf-nodes.

Algorithm 5 stops when any pair of objects in $\phi_{cur}^{[L]}$ does not further split L. The complexity depends on the size of the *turning points* in the final answer. Recall the splitting process of *Ternary Decomposition*, a ternary tree node T_i splits only if one or two intersections are found in T_i's line-segment. If no intersections found in its line-segment, T_i becomes a leaf-node. Given the final answer containing m *turning points*, there would be at most $2m$ nodes in the ternary tree $\Psi(\mathcal{T})$. At least, there are $\lceil 1.5m \rceil$ nodes. So, Algorithm 5 will be called $(1.5m, 2m]$ times. `Step 5` is done in β and `Step 6` is in $O(\log C)$. If the candidate answers returned by *Phase I* contains C objects, the complexity of *Phase II* is $O(mC^2(\log C + \beta))$. Next, we study how to derive the pruning bound B_i mentioned in *Step 11*.

6.2 Pruning Bounds for Three Cases

By a u-bisectors, a query line-segment could be divided into at most 3 sub-line-segments. The sub-line-segments fall into 3 categories according to their positions in half spaces. There are three types of sub-line-segments: *Open Case*, *Pair Case*, and *Close Case*, For example, in Figure 5, $[s_{d\vdash c}, s_{d\dashv c}]$ belongs to the *pair case*. Two examples of *open case* are $[h, s_{d\vdash c}]$ and $[s_{d\dashv c}, t]$. The *Close Case* means the line-segment is totally covered by a half-space. The three cases are formally described in Table 1.

Table 1. Three cases for a line segment

Case	Form	Position
pair	$[s_{i\vdash j}, s_{i\dashv j}]$	$l \in V(i,j)$
open	$[s, s_{i\vdash j}]$ or $[s_{i\dashv j}, e]$	$l \in H_i(j)$(*or* $l \in H_j(i)$) ($s(e)$ is the line-segment's start(end) point)
close	$[s_{i\vdash j}, s'_{i\vdash j}]$	$l \in H_i(j)$ and $s_{i\vdash j}, s'_{i\vdash j} \in b_i(j)$

For *Pair Case* and *Open Case*, we can derive two types of pruning bounds. Suppose the u-bisector between O_1 and O_2 split the query line-segment $[s,e]$ into sub-line-segments: $[s, s_{1\vdash 2}]$, $[s_{1\vdash 2} s_{1\dashv 2}]$, and $[s_{1\dashv 2}, e]$, which are of $Open\ Case$, $Pair\ Case$, and $Open\ Case$, respectively. We shown the pruning bound derived for $[s, s_{1\vdash 2}]$ and $[s_{1\vdash 2} s_{1\dashv 2}]$ in Figure 7 (a) and (b). The bounds are highlighted by shaded areas. The pruning bound of $[s_{1\dashv 2}, e]$ is similar to Figure 7(a), so it is omitted.

Close Case is a special case, when a line-segment has two intersections and totally inside one half-space, say $H_i(j)$. It could be represented by $[s_{i\vdash j}, s'_{i\vdash j}]$, which means the two end-points are on the same u-bisector half $b_i(j)$. In this example, we known $[s_{i\vdash j}, s'_{i\dashv j}]$ must be in $H_i(j)$, so O_j cannot be the PNN for each point inside. Next, we design their pruning bounds.

Lemma 3. (Pair Case) *Suppose two imprecise objects O_i and O_j, whose u-bisector $b_i(j)$ and $b_j(i)$ intersect with a straight line at $s_{i\vdash j}$ and $s_{i\dashv j}$. For another object $\forall O_N \in O$, it cannot be $q \in [s_{i\vdash j} s_{i\dashv j}]$'s PNN, if O_N has no overlap with the pruning bound $\circledcirc(s_{i\vdash j}, O_i) \cup \circledcirc(s_{i\dashv j}, O_j) \bigcap \circledcirc(s_{i\vdash j}, O_j) \cup \circledcirc(s_{i\dashv j}, O_i)$.*

Proof. $\forall p \in [s_{i\vdash j} s_{i\dashv j}]$, both O_i and O_j have chances to be p's PNN. According to Lemma 1, a new object O_N cannot be O_i or O_j's nearest neighbor if

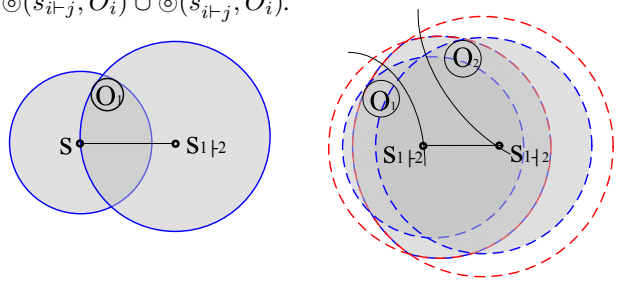
$$O_N \bigcap (\circledcirc(s_{i\vdash j}, O_i) \cup \circledcirc(s_{i\dashv j}, O_i)) = \emptyset, or O_N \bigcap (\circledcirc(s_{i\vdash j}, O_j) \cup \circledcirc(s_{i\dashv j}, O_j)) = \emptyset$$

So, the pruning bound is:

$$\circledcirc(s_{i\vdash j}, O_i) \cup \circledcirc(s_{i\dashv j}, O_i) \bigcap \circledcirc(s_{i\vdash j}, O_j) \cup \circledcirc(s_{i\dashv j}, O_j) \quad (7)$$

Lemma 4. (Open Case) *Given a line-segment* $[s, s_{i\vdash j}]$*, for other objects* $\forall O_N \in O$*, it cannot be query point* $q \in [s, s_{i\vdash j}]$*'s nearest neighbor, if* O_N *has no overlap with the* $\circledcirc(s, O_i) \cup \circledcirc(s_{i\vdash j}, O_i)$.

Lemma 5. (Close Case) *Given two split points* $s_{i\vdash j}$ *and* $s'_{i\vdash j}$*, the pruning bound for* $[s_{i\vdash j}, s'_{i\vdash j}]$ *is* $\circledcirc(s_{i\vdash j}, O_i) \cup \circledcirc(s'_{i\vdash j}, O_i)$.



(a) Open Case (b) Pair Case

Fig. 7. Open Case and Pair Case

Since the proofs of Lemma 5 and Lemma 4 can be easily derived from Lemma 1, they are omitted due to page limitation. The *Pair Case* could also be considered as the overlap of two *Open Case*s. For example, a *Pair Case* $[s_{i\vdash j}, s_{i\dashv j}]$ is equivalent to the overlap part of $[s, s_{i\dashv j}]$ and $[s_{i\vdash j}, e]$. Also, the *Close Case* could be viewed as the overlap of $[s, s'_{i\dashv j}]$ and $[s_{i\vdash j}, e]$. The three cases and their combinations could cover all the cases for each piece(*validity interval*) of the line segment. After Ψ's construction is done, we can view the pruning bound of a *validity interval*. It is the intersection of all its ascender nodes' pruning bounds in the ternary tree Ψ.

7 Experimental Results

Section 7.1 describes settings. We adopt a metric to measure to quality of results in Section 7.2. Section 7.3 discusses the results.

7.1 Setup

Queries The query trajectories are generated by Brinkhoff's network-based mobile data generator [4]. The trajectory represents movements over the road-network of Oldenburg city in Germany. We normalize them into 10k $\times$10k space. By default, the length of trajectory is 500 units. Each reported value is the average of 20 trajectory query runs.

Imprecise Objects We use four real datasets of geographical objects in Germany and US[5], namely *germany*, *LB*, *stream* and *block* with 30k, 50K, 199K, 550k spatial objects,

[4] http://iapg.jade-hs.de/personen/brinkhoff/generator/

[5] http://www.rtreeportal.org/

respectively. We use *stream* as the default dataset. We construct the MBC for each object thus get 4 datasets with circular imprecise regions. Datasets are normalized to the same domain as queries. To index imprecise regions, we use a packed R*-tree [16]. The page size of R-tree is set to 4k-byte, and the fanout is 50. The entire R-tree is accommodated in the main memory.

For the *turning points* calculation, we call GSL Library [6] to get the analytical solution. All our programs were implemented in C++ and tested on a Core2 Duo 2.83GHz PC.

7.2 Quality Metric

To measure the accuracy of a query result, we adopt a *Error* function based on the *Jaccard Distance* [17], which is used in comparing the similarity between two sets. Recall the definition of TPNNQ the query result is a set of tuples $\{\langle T_i, R_i\rangle\}$. It can be transformed into the PNNs for every point on the query trajectory. Formally, the result is $\{\langle q, PNNQ(q)\rangle\}_{q\in\mathcal{T}}$. Let $R^*(q)$ be the optimal solution for the point q, where $R^*(q) = PNNQ(q)$. We use $R^A(q)$ to represent the PNNs derived for the point q in algorithm A. Then, the *Error* for algorithm A on query $\mathcal{T}$ is:

$$Error(\mathcal{T}, A) = \frac{1}{|\mathcal{T}|}\int_{q\in\mathcal{T}} 1 - \frac{R^*(q)\cap R^A(q)}{R^*(q)\cup R^A(q)}dq \tag{8}$$

$|\mathcal{T}|$ is the total length of trajectory $\mathcal{T}$. If $\mathcal{T}$ is represented by a set of line-segments $\mathcal{T} = \{L_i\}_{i=1}^t$, the total length $|\mathcal{T}| = \sum_{i=1}^t |L_i|$.

Equation 8 captures the effect of false positives and false negatives as well. There is a *false positive* when $R^A(q)$ contains an extra item not found in $R^*(q)$. There is a *false negative* when an item of $R^*(q)$ is missing from $R^A(q)$. For a perfect method with no false positives and false negatives, the two terms $R^*(q)$ and $R^A(q)$ are the same, so the integration value is 0.

In summary, the error score is a value between 0 and 1. The smaller an *Error* score is, the more accurate the result is. On the other hand, if a method has many extra or missing results, then it obtains a high *Error*.

7.3 Performance Evaluation

The query performance is evaluated by two metrics: efficiency and quality. The efficiency is measured by counting the clock time. The quality is measured by the *error score*. We compare four methods: *Basic*, *Sample*, TP-S, and TP-TS. The suffixes T and S refer to *Trajectory Filter* and *Segment Filter*, respectively. *Basic* does not use any filter; TP-S does not use *Trajectory Filter*; TP-TS (Algorithm 2) uses all the filtering and refinement techniques. *Sample* draws a set of uniform sampling points $\{q\}$ from $\mathcal{T}$. Then, for all q, $PNNQ(q)$ is evaluated. The sampling interval, denoted by ϵ, is set to 0.1 unit. [7]

Query Efficiency T_q From Figure 8, the *Basic* method is the slowest method among all the four, since it elaborates all the possible pairs of objects for *turning points* (but most of them do not contribute to validity intervals). For the second slowest *Sample*, we analyze it later.

[6] http://www.gnu.org/software/gsl/

[7] The sampling rate is reasonably high regarding to the trajectory's default length. More details about sampling rates are discussed later.

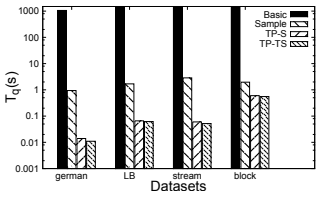


Fig. 8. T_q(s) vs. Datasets

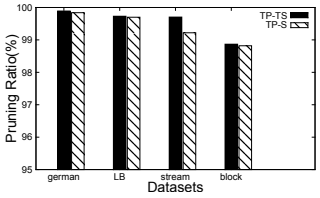


Fig. 9. Pruning Ratio vs. Datasets

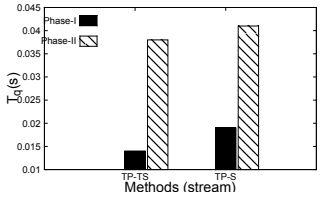


Fig. 10. T_q's breakdown

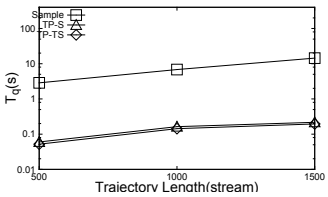


Fig. 11. T_q vs. Query Length

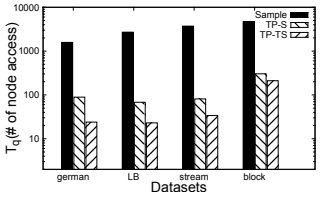


Fig. 12. T_q(# of node access) vs. Datasets

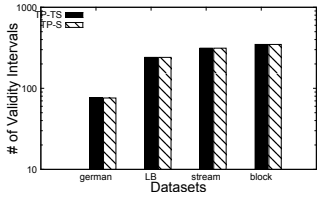


Fig. 13. ♯ of Validity Intervals vs. Datasets

The other two methods have significant improvement over *Sample* and *Basic*. One reason is because of the effectiveness of the pruning techniques, as shown in Figure 9. For all the real datasets, the pruning ratio are as high as 98.8%. TP-S is less efficient, because some candidates shared by different line-segments in trajectory will be fetched multiple times. This drawback is overcome by TP-TS.

To get a clearer picture about the efficiency of our framework, we measure the time costs for *Phase I* and *Phase II* in Figure 10. TP-TS is faster in both phases. In *Phase I*, the combined R-tree traversal in TP-TS saves plenty of extra node access, compared to TP-S. The number of node access is shown in Figure 12. In *Phase II*, TP-TS is faster, since it has fewer candidates to handle. This observation is also consistent with the fact that TP-TS has a higher pruning ratio, shown in Figure 9.

We also test the query efficiency by varying the query length in Figure 11. The *Sample* method is slower than others at least one order of magnitude. The costs of other two methods increase slowly w.r.t. the query length.

TP-TS vs. *Sample* *Sample* method is a straightforward solution to approximate the $TPNNQ$ answer. However, this solution suffers from the extensive R-tree traversals, since every sampling point q requires accessing of R-tree. As shown in Figure 12, *Sample* incurs at least more than one order of magnitude node access than our method.

On the other hand, *Sample* could incur *false negatives*, even with a large sampling rate. Because *Sample* only considers query points sampled on the trajectory, whereas $TPNNQ$ is for all the points in $\mathcal{T}$. To calculate *Sample*'s error score, we have to infer the PNNs for a point $q \in \mathcal{T}$ not being sampled, as required by Equation 8. With limited sampled answers, q's PNNs can only be "guessed" by using its closest sampling point p. In other words, $PNNQ(q)$ has to be substituted with $PNNQ(p)$.

The efficiency is reflected in Figure 14, where the sampling interval ϵ is varied from 0.01 to 10. We can observe that TP-TS outperforms *Sample* in most of the cases. *Sample* is faster only when ϵ is very large (e.g. equal to 10 units). Then, is it good if large ϵ is

used? The answer is **NO**. In Table 15, when "*Sample, ϵ = 10, block*", the error score of *Sample* is as high as 0.443!

We demonstrate the error score of *Sample* and TP-TS in Table 15. Since TP-TS evaluate the exact answer, the error is always 0. The error of *Sample* is small when ϵ is small, (e.g. equal to 0.01, block). However, the query time of that case is 100 times slower than TP-TS. We would like to emphasize that even the *error score* is empirically tested to be 0 over large sampling rates, there is no theoretical guarantee for the *Sample* to contain 0 false negative.

We also test the *error score* of simplifying the imprecise regions into precise points, as mentioned in the introduction. For *german* dataset, the error is as high as 0.76! Thus, the simplified solution could be harmful for applications such as safety sailing.

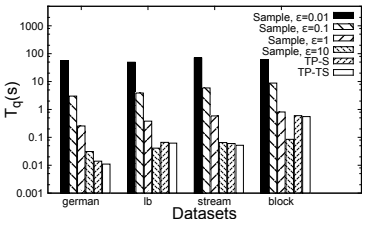


Fig. 14. TP-TS vs. Sample (T_q)

Datasets	Sample				TP-TS
	$\epsilon = 0.01$	$\epsilon = 0.1$	$\epsilon = 1$	$\epsilon = 10$	
german	0.00340	0.00457	0.01528	0.12310	0
LB	0.00005	0.00029	0.00257	0.02672	0
stream	0.00059	0.00090	0.00298	0.03962	0
block	0.01872	0.02541	0.08516	0.44310	0

Fig. 15. TP-TS vs. Sample(Error)

Analysis of TPNN Observed from Figure 13, the number of validity intervals increases with the size of the datasets. TP-S and TP-TS have the same number of validity intervals, which is as expected.

In summary, we have shown that TP-TS is much more efficient than *Basic*, *Sample*, and TP-S methods. It also achieves much better quality than *Sample* method.

8 Conclusion

In this paper, we study the problem of trajectory query over imprecise data. To tackle the low quality and inefficiency in simplified methods, we study the geometric properties of the u-bisector. Based on that, we design several novel filters to support our algorithm. Extensive experiments show that our method can efficiently evaluate the $TPNNQ$ with high quality.

The geometric theories studied in this paper has no limitations in the dimensionality and shape of imprecise regions. In future, we would like to evaluate the algorithm's performance in multi-dimensional space with different shaped imprecise regions. We would also extend our work to support variants queries like k-PNN query, etc.

Acknowledgment

Reynold Cheng and Xike Xie were supported by the Research Grants Council of Hong Kong (GRF Projects 513508, 711309E, 711110). Man Lung Yiu was supported by ICRG grants A-PJ79 and G-U807 from the Hong Kong Polytechnic University. We would like to thank the anonymous reviewers for their insightful comments.

References

1. Y. Tao, D. Papadias, and Q. Shen, "Continuous nearest neighbor search," in *VLDB*, 2002.
2. U. S. C. Guard, "Announcement of 2011 international ice patrol services," http://www.uscg.mil/lantarea/iip/docs/AOS_2011.pdf.
3. L. Jesse, R. Janet, G. Edward, and V. Lee, "Effects of habitat on gps collar performance : using data screening to reduce location error," in *Journal of applied ecology*, 2007.
4. K. Park, H. Choo, and P. Valduriez, "A scalable energy-efficient continuous nearest neighbor search in wireless broadcast systems," *Wireless Networks*, 2010.
5. R. Cheng, X. Xie, M. L. Yiu, J. Chen, and L. Sun, "Uv-diagram: A voronoi diagram for uncertain data," in *ICDE*, 2010.
6. X. Lian and L. Chen, "Efficient processing of probabilistic reverse nearest neighbor queries over uncertain data." in *VLDBJ*, 2009.
7. M. A. Cheema, X. Lin, W. Wang, W. Zhang, and J. Pei, "Probabilistic reverse nearest neighbor queries on uncertain data." in *TKDE*, 2010.
8. J. Chen, R. Cheng, M. Mokbel, and C. Chow, "Scalable processing of snapshot and continuous nearest-neighbor queries over one-dimensional uncertain data," in *VLDBJ*, 2009.
9. G. Trajcevski, R. Tamassia, H. Ding, P. Scheuermann, and I. F. Cruz, "Continuous probabilistic nearest-neighbor queries for uncertain trajectories," in *EDBT*, 2009, pp. 874–885.
10. K. Zheng, G. P. C. Fung, and X. Zhou, "K-nearest neighbor search for fuzzy objects," in *SIGMOD*, 2010.
11. Z. Song and N. Roussopoulos, "K-nearest neighbor search for moving query point," in *SSTD*, 2001.
12. Z. Baihua and L. Dik, "Semantic caching in location-dependent query processing," in *Advances in Spatial and Temporal Databases*, 2001.
13. J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee, "Location-based spatial queries." in *SIGMOD*, 2003.
14. R. Cheng, D. V. Kalashnikov, and S. Prabhakar, "Querying imprecise data in moving object environments," *TKDE*, vol. 16, no. 9, 2004.
15. S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.
16. M.Hadjieleftheriou, "Spatial index library version 0.44.2b." [Online]. Available: http://u-foria.org/marioh/spatialindex/index.html
17. P.-N. Tan, M. Steinbach, and V. Kumar, "Introduction to data mining," 2006.

A Appendix

A.1 Intersection of a hyperbola and a straight line

Given a hyperbola h_1 and a straight line l_1, they could have 0, 1, or 2 intersection points, which is the roots of the following:

$$\begin{cases} h_1 : \frac{x^2}{a_1^2} - \frac{y^2}{b_1^2} = 1 \\ l_1 : a_2 x + b_2 y + c_2 = 0 \end{cases} \tag{9}$$

By solving Equation 9, we can have:

$$\begin{cases} x = \frac{-a_1^2 a_2 c_2 \pm \sqrt{-a_1^2 b_1^2 b_2^2 (a_1^2 a_2^2 - b_1^2 b_2^2 - c_2^2)}}{a_1^2 a_2^2 - b_1^2 b_2^2} \\ y = \frac{-a_2 \pm \sqrt{a_1^2 b_1^2 b_2^2 (-a_1^2 a_2^2 + b_1^2 b_2^2 + c_2^2)} - b_1^2 b_2^2 c_2}{b_2 (b_1^2 b_2^2 - a_1^2 a_2^2)} \end{cases} \tag{10}$$

where

$$\begin{cases} a_1^2 a_2^2 - b_1^2 b_2^2 \neq 0 \\ b_2 \neq 0 \\ a_1 b_1 \neq 0 \end{cases} \tag{11}$$

Notice that if any of the three pre-conditions in Equation 11 is not satisfied, there should be no intersection point for the given curve and line.