# Trajectory Possible Nearest Neighbor Queries over Imprecise Location Data

Xike Xie, Man Lung Yiu, Reynold Cheng, and Hua Lu

October 10, 2012

TR-33

A DB Technical Report

| | |
|---|---|
| Title | Trajectory Possible Nearest Neighbor Queries over Imprecise Location Data |
| |  |
| Author(s) | Xike Xie, Man Lung Yiu, Reynold Cheng, and Hua Lu |
| Publication History | |

**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 complex natures of the surroundings. For such data, we consider a common model, where the possible locations of an object are bounded by a closed region, called "imprecise region". Ignoring or coarsely wrapping imprecision can render low query qualities, and cause undesirable consequences such as missing alerts of threats and poor response rescue time. Also, the query is quite time-consuming, since all points on the trajectory are considered. In this paper, we study how to efficiently evaluate trajectory queries over imprecise objects, by proposing a novel concept, $u$-bisector, which is an extension of bisector specified for imprecise data. Based on the $u$-bisector, we provide an efficient and versatile solution which supports different shapes of commonly-used imprecise regions (e.g., rectangles, circles, and line segments). Extensive experiments on real datasets show that our proposal achieves better efficiency, quality, and scalability than its competitors.

# 1 Introduction

Trajectory queries retrieve nearby objects for a given route. Such queries are useful in various domains including transportation and facility management. For example, in the air and shipping industries where safety is the top priority, it is very important to identify potential threats along the route of a flight or a vessel and give alerts in advance. Such threats are exemplified by volcanic ashes for flights in North Europe [1] and icebergs for vessels in US [2]. As another example, trajectories can also represent the pipelines for transporting oil, gas, water, etc. When a section of a pipeline is broken, it causes economic loss and potential hazard. The authority therefore needs to call up the technicians nearest to the damage spot in order to fix the problem [3] as soon as possible.

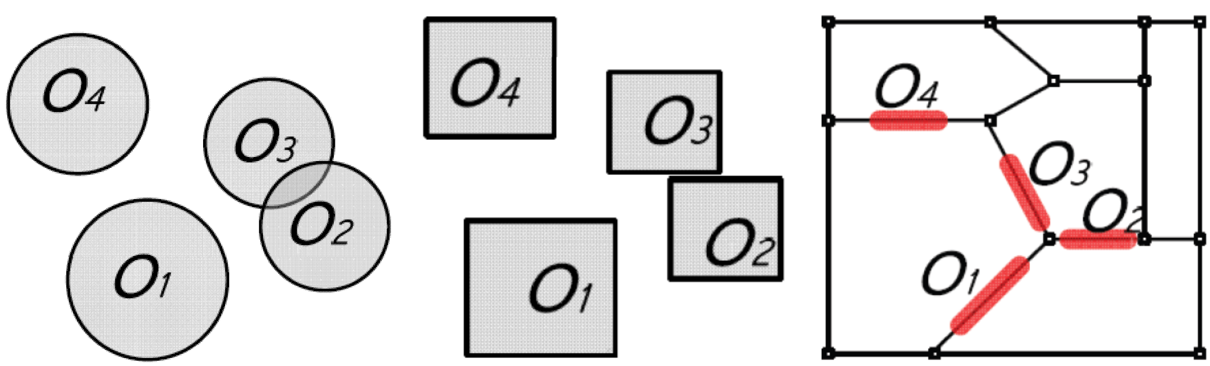
One fundamental challenge in such scenarios is that the measured locations of objects (e.g., clouds of volcanic ash, icebergs, or people) are imprecise. Such imprecise locations result from: (i) limited resolution of the measure device, (ii) infrequent measurement, and/or (iii) environmental factors.

In the transportation example, the threats (icebergs or volcanic ashes) are often detected by remote sensing technologies like satellite imaging. Such technologies usually work at low sensing frequency because of cost constraints, and thus render the measured locations stale for objects. Furthermore, icebergs (volcanic ashes) can move depending on the ocean current (wind) speed. In the pipeline example, a technician may have a GPS device for location tracking [3], where GPS reports locations with measurement errors subject to terrain and climate conditions [4].

Consequently, trajectory queries have to handle such imprecise objects whose locations cannot be precisely determined. Table 1 summarizes these aforementioned two kinds of applications that involve imprecise objects.

Table 1: Summary of Applications

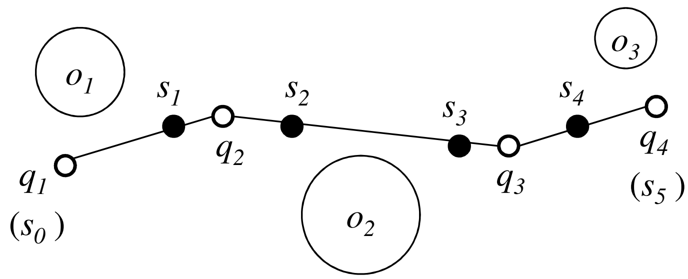
| **Application** | **Route Safety** | **Pipeline Maintenance** |
|---|---|---|
| Trajectory | route of a flight or vessel | fuel or water pipeline |
| Objects | volcanic ashes or icebergs | technicians |
| Localization | remote sensing | GPS |
| Imprecise data source | resolution, environment, infrequent measurement | GPS error, terrain, climate |




(a) circle (b) rectangle (c) line-segment

Figure 1: Imprecise regions. (a) A circle can be used to describe the position uncertainty of a person or vehicle tracked by GPS [5]. (b) A rectangle can be a person's imprecise region when the RFID-based indoor tracking works on the room level [6]. (c) A line segment is used, when a vehicle is moving in a road network [5].

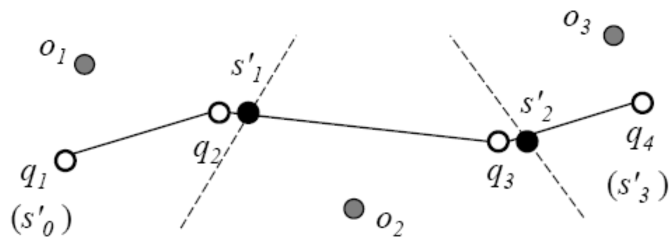
A common way to model an imprecise object is to use so-called *imprecise region* [7, 8, 9, 10, 11, 12, 13], which is a closed region covering all possible position during a time interval. Figure 1 illustrates imprecise regions of different shapes that are seen in GPS, RFID, and road network applications.




(a) imprecise objects

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

(b) result




(c) precise objects

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

(d) result

Figure 2: Example Trajectory Query

In this paper, we study the problem of searching imprecise objects close to a given query trajectory. Figure 2(a) shows a query trajectory $T = \{q_1, q_2, q_3, q_4\}$ and a set of imprecise objects $o_1, o_2, o_3$. The query result (Figure 2(b)) is represented in a compact way by partitioning the query trajectory into segments such that all locations within the same segment share the same result. In this example, $o_2$ is the *definite nearest neighbor* to segment $[s_2, s_3]$. On the other hand, $o_1$ and $o_2$ are *possible nearest neighbors (***PNN***s)* to segment $[s_1, s_2]$ because both of them have potential to be the closest object for any location between $s_1$ and $s_2$.

Determining the query results over imprecise objects is technically challenging, as the geometries of the imprecise regions must be considered. A simple solution is to replace the imprecise region of each object with a central point (shown as a grey dot in Figure 2(c)). Accordingly, the single closest object is associated with the corresponding segment in the query result, as shown in Figure 2(d). For instance, the closest object to location $q_2$ appears to be object $o_1$ only and object $o_2$ is missing from the result. Recall that object $o_2$ also has the possibility to be a closest object to location $q_2$ as shown in Figure 2(a) and (b).

In the aforementioned application scenarios, the "center simplification" approach causes undesirable consequences such as missing threat alerts and poor response time. In the flight/vessel example, modeling threats as imprecise regions prioritizes the safety in all cases, whereas the ignorance of imprecise regions can cause potential dangers. In the pipeline example, a technician seemingly close to (far from) the broken pipeline section may be actually far from (close to) it due to the location imprecision. Calling up such a technician would incur longer time to respond to the emergency. It is important to call up all technicians likely to be close to the damage spot, in order to fix the problem as soon as possible.

An alternative to simplify the trajectory query is the "sampling approach", which considers only those positions at every fixed length on the query trajectory and computes the nearby objects for each such sample. However, deciding the sampling rate is a dilemma in this approach. A high sampling rate incurs huge computation costs, while a low rate can miss many answers. Referring to Figure 2(a), the query result changes only at a few positions ($s_1, s_2, s_3, s_4$). It is not clear how to decide the correct sampling rate in

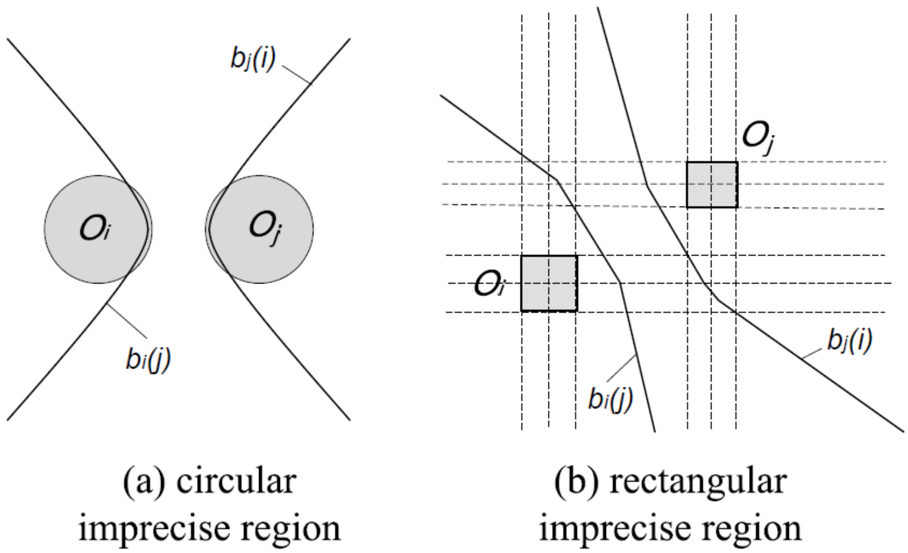



Figure 3: $u$-bisector for imprecise regions.

order to get these answers.

Neither the center simplification approach nor the sampling approach solves the trajectory queries over imprecise objects. In fact, our preliminary experiments show that they cannot guarantee correct and complete query results. Therefore, we develop a solution that can accurately compute a trajectory query on imprecise objects in this paper. A special case of our problem, finding the closest precise points for a given query trajectory, was studied by Tao et al. [14]. The authors used the (perpendicular) *bisectors* of each pair of consecutive points to derive the query answer. For example, in Figure 2(c), the point $s'_1$ is the intersection between the query trajectory and the bisector (in dashed lines) of precise points $o_1$ and $o_2$. Likewise, $s'_2$ is derived by the bisector of $o_2$ and $o_3$.

We extend the bisector concept to $u$-bisector in order to support imprecise objects. Figure 3 illustrates the $u$-bisectors for circular and rectangular imprecise regions. Note that a $u$-bisector is not a straight line anymore for two objects $o_i$ and $o_j$. Instead, it becomes a pair of curves, namely $b_i(j)$ and $b_j(i)$, that partition the domain space into three parts: (1) the left part, where points are absolutely closer to $O_i$ than to $O_j$; (2) the right part, where points are absolutely closer to $O_j$ than to $O_i$; and (3) the middle part, where points can be closer to either $O_i$ or $O_j$. We call the region enclosed by a $u$-bisector half as a *half-space*. For example, in Figure 3(a), the left of $b_i(j)$ is a half-space, and so is the right of $b_j(i)$. We make use of half-spaces and $u$-bisectors to answer a trajectory query.

In practice, it is challenging to compute the intersections between the query trajectory and $u$-bisectors. As shown in Figure 3, $u$-bisectors can be hyperbolic curves (Figure 3(a)), or polylines (Figure 3(b)). Furthermore, these $u$-bisectors may intersect the query trajectory at multiple points. Our solution avoids generating $u$-bisectors for all pairs of imprecise objects by employing a filter-refinement framework. In the filtering phase, *candidate objects* that may be the closest to each query segment are obtained. In the refinement phase, we develop a novel technique called *tenary decomposition* to derive the final answers accurately. We show theoretically and experimentally that our solution is efficient and scalable. Moreover, our solution can easily adapt to imprecise objects of arbitrary shapes to other shapes (e.g., circles, rectangles, line segments, etc.) that are required in different applications.

This paper substantially extends our previous work [15] in several aspects. First, we theoretically prove that a half-space is convex for arbitrary shaped imprecise objects (Section 4.1). Second, we extend the query techniques from supporting circular imprecise objects to objects of arbitrary shapes (Section 4.2). Third, we derive an novel analysis model to estimate the selectivity for trajectory queries (Section 5). Fourth, we conduct extensive additional experiments to evaluate the new proposals (Section 6.3).

The rest of this paper is organized as follows. Section 2 defines the trajectory query we study and presents two query evaluation approaches. Section 3 elaborates on a simplified yet fundamental case where a query trajectory is a single line segment. Section 4 proposes generalized techniques to support different shaped imprecise regions of objects. Section 5 designs an analysis model for trajectory queries. Section 6 presents the experiment results. Section 7 discusses the related works and finally Section 8 concludes the paper. The notations used throughout the paper are listed in Table 2.

Table 2: Notations and meanings.

| Notation | Meaning |
|---|---|
| $D$ | Domain space (a square) |
| $\lvert \cdot \rvert$ | the area of a region |
| $O$ | a set of imprecise objects $(O_1, O_2, \ldots, O_n)$ |
| $MBC(O_i)$ | minimum bounding circle of object $O_i$ |
| $\odot_i(c_i, r_i)$ | circle $\odot_i$ with center $c_i$ and radius $r_i$ |
| $\circledcirc(p, O_i)$ | circle centered $p$ and internally tangent with $O_i$ |
| $\overline{se}$ / $[s, e]$ | line segment with two end points $s$ and $e$ |
| $b_i(j)$ | $O_i$ and $O_j$'s $u$-bisector half, which is closer to $O_i$ |
| $H_i(j)$ | half space cut by $b_i(j)$, which is closer to $O_i$ |
| $s_{i\vdash j}$ | intersection between a line segment and $b_i(j)$ |
| $s_{i\dashv j}$ | intersection between a line segment and $b_j(i)$, which is equivalent to $s_{j\vdash i}$ |
| $q$ | a query point |
| $\oplus$ | Minkowski sum |
| $\mathcal{T}$ / $\lvert\mathcal{T}\rvert$ | trajectory $\mathcal{T}$ / length of trajectory $\mathcal{T}$ |
| $\mathbb{T}(\mathcal{T})$ | trajectory tree constructed for trajectory $\mathcal{T}$ |
| $\Psi(\mathcal{L})$ | ternary tree constructed for line-segment $\mathcal{L}$ |

# 2 Trajectory Possible Nearest Neighbor Queries

## 2.1 Problem Definitions

We first introduce the definition of *PNNQ* (studied in [5]), which is used to define the query studied in this paper. Let $q$ be a point, and $O_i$ an imprecise object from a set $O$. We use $dist_{min}(q, O_i)$ and $dist_{max}(q, O_j)$ to denote the minimum and maximum distances between $q$ and $O_i$, respectively.

**Definition 1** ***Possible Nearest Neighbor Query (PNNQ)*** *Given a set of imprecise objects $O$ and a query point $q$, the result of the* PNNQ *query is a set $PNNQ(q) = \{O_i \in O \mid \forall O_j \in O(dist_{max}(q, O_j) \geq dist_{min}(q, O_i))\}$.*

In Figure 2(a), $PNNQ(q_2) = \{O_1, O_2\}$ implies that either $O_1$ or $O_2$ could be the *NN* of the query point $q_2$. By extending the concept of *PNNQ* to all points in a query trajectory $\mathcal{T}$, we define the *trajectory possible nearest neighbor query* (*TPNNQ*) which returns *PNNQ* for all the points in $\mathcal{T}$. In other words, the query returns $\{\langle q, PNNQ(q)\rangle\}_{q\in\mathcal{T}}$. To get a compact representation of the query result, we merge all consecutive trajectory points that have the same *PNNQ*. The formal definition of *TPNNQ* is given below.

**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 $R = \{\langle T_i, R_i \rangle \mid T_i \subseteq \mathcal{T}, R_i \subseteq O\}$, where PNNQ(q) $= R_i (\forall q \in T_i)$, and $T_i$ is a continuous segment in $\mathcal{T}$.*

In other words, the *TPNNQ* splits $\mathcal{T}$ into a set of consecutive segments $\langle T_1, T_2, ..., T_t \rangle$ where each $T_i$ is a sub-trajectory of $\mathcal{T}$, such that all positions in a given $T_i$ have the same possible nearest neighbors. Formally, $\forall q_i, q_j \in T_i$, *PNNQ*$(q_i)$ = *PNNQ*$(q_j)$. We call each $T_i$ a ***validity interval***. Accordingly, we call the connection point of two consecutive intervals ***turning point***. Such a turning point 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 2(c). The trajectory query $\mathcal{T}(s_0, s_5)$ is split into 5 segments. Also, point $s_1$ is the turning point for segments $T(s_0, s_1)$ and $T(s_1, s_2)$. It is apparent that finding turning points is crucial for evaluating *TPNNQ*. This is however a non-trivial task for imprecise location data. We propose an effective technique for this task in Section 2.2, and develop algorithms on top of it to evaluate *TPNNQ* in Section 2.3.

There are two major differences between the results on imprecise objects and precise objects. Comparing Figures 2(c) and (a): (1) the *imprecise case* could have more result tuples (5 compared to 3); (2) a query point in *imprecise case* might return a set of *PNN*s instead of a single object. These observations indicate that the previous techniques for trajectory queries over precise objects [14] do not solve *TPNNQ*.

## 2.2 Finding Turning Points with $u$-bisectors

Given a set of imprecise objects and a query trajectory, derive the turning points on the trajectory is the crucial step for answering *TPNNQ*. To address that, we first investigate the $u$*-bisector* for imprecise objects. In general, the $u$*-bisector* splits the domain space into several parts, such that query points on different parts could have different *PNN*s. After that, the *turning points* are decided by finding the intersections of the $u$*-bisectors* and the query trajectory.

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

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

The curve $b_i(j)$ splits the domain space into two parts: $H_i(j)$ and $\overline{H_i(j)}$, where $H_i(j)$ is the part covering all points closer to $O_i$ than to $O_j$ and $\overline{H_i(j)}$ is the remaining part of the domain space. We call $H_i(j)$ a **half-space**, and $\overline{H_i(j)}$ as a **half-space complement**. An example is shown in Figure 4. Formally, we have:

$$\begin{aligned} H_i(j) &= \{z : dist_{max}(z, O_i) \leq dist_{min}(z, O_j)\} \\ \overline{H_i(j)} &= \{z : dist_{max}(z, O_i) > dist_{min}(z, O_j)\} \end{aligned}$$

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. The $u$-bisector halves $b_i(j)$ and $b_j(i)$ separate the domain into three parts, including two half-spaces $H_i(j)$ and $H_j(i)$, and a region $V(i, j)$, where

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

Notice that $V(i, j) = V(j, i)$. If $O_i$ and $O_j$ are degenerated into precise points, $V(i, j)$ becomes $\emptyset$ and $b_i(j)$ merges with $b_j(i)$ into a straight line.

If a query line segment is totally covered by $V(i, j)$, $H_i(j)$, or $H_j(i)$, it does not intersect with $b_i(j)$ or $b_j(i)$. Otherwise, the intersections split the line segment into several parts. Different parts correspond to different *PNN*s answers, as those parts are located on different sides of $b_i(j)$ or $H_j(i)$.

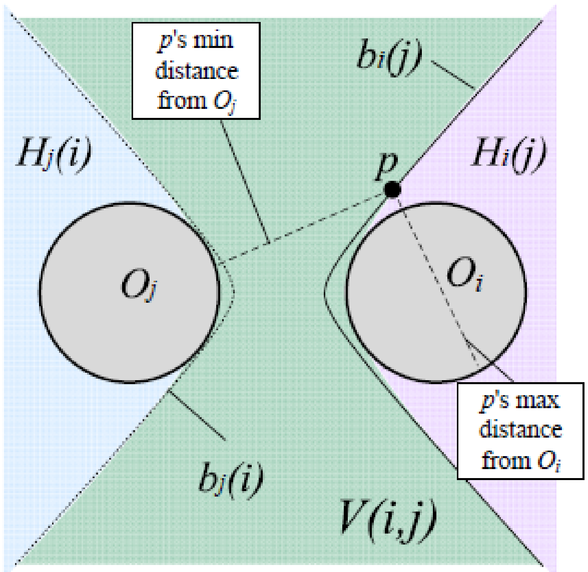



Figure 4: $u$-bisector

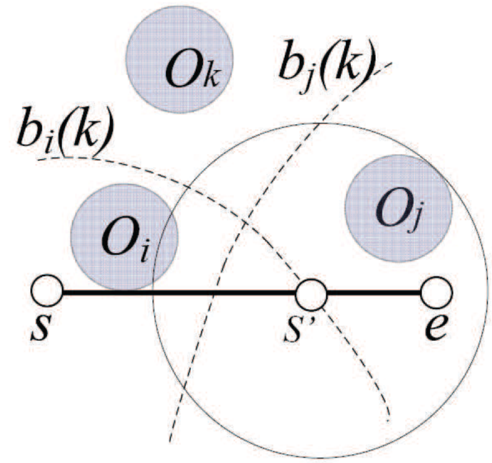



Figure 5: Verification

For circular imprecise objects, it is easy to derive the closed form equations of the $u$-bisectors and evaluate the analytical solution for the intersection points. The procedure to find such intersections is formalized in Algorithm 1. The number of intersections is at most 2, since the equation group (line 8) has at most 2 roots.

As a matter of fact, we find that the "2-intersection" fact holds for arbitrary shaped imprecise regions. For the sake of presentation, we use circular imprecise regions in following sections (Sections 2.3 to 3) and present the generalization to other shapes in Section 4.

## 2.3 Evaluating *TPNNQ*

In this section, we present two approaches for evaluating *TPNNQ*. Section 2.3.1 discusses a nested-loop approach, and Section 2.3.2 presents a more advanced approach that employs the filter-refinement paradigm.

### 2.3.1 Nested-Loop Approach

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 *Nested-Loop* method (Algorithm 2) checks the intersections between the query trajectory and each of the $C_2^n$ $u$-bisectors. The intersections are found by calling Algorithm 1 on line 5.

However, not all the $u$-bisectors intersect with the trajectory. Even if they intersect, not all of the intersections are qualified as *turning points*. For example, in Figure 5, 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 $[s', e] \in H_j(k)$. As a result, $s'$ is not a qualified turning point and $O_k$ is not *PNN* for $p \in [s, e]$. In Algorithm 2, we employ a "*verification*" (line 6) process to exclude those unqualified intersections and their corresponding objects.

The verification works as follows. We use the $s_{i \vdash j}$ to represent an intersection created by $b_i(j)$ ($s_{i \vdash j} = b_i(j) \cap \mathcal{L}$), and $s_{i \dashv j} = b_j(i) \cap \mathcal{L}$. In other words, $s_{i \vdash j}$ can be regarded 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 $\overline{H_i(j)}$. Therefore, $O_i$ should definitely be $s_{i \vdash j}$'s *PNN*, while $O_j$ is not. This can be verified by issuing a *PNNQ* for point $s_{i \vdash j}$.

**Algorithm 1** FindIntersection$^e$

**function** FINDINTERSECTION$^e$(Line segment $\mathcal{L}(s,e)$, Objects $O_i, O_j$)
Let $R$ be a set (of intersection points);
Let $O_i = \odot(c_i, r_i)$ and $O_j = \odot(c_j, r_j)$;
$f_x = \frac{c_i.x + c_j.x}{2} \quad f_y = \frac{c_i.y + c_j.y}{2}$;
$cos\theta = \frac{c_j.x - c_i.x}{dist(c_i, c_j)} \quad sin\theta = \frac{c_j.y - c_i.y}{dist(c_i, c_j)}$;
Construct the hyperbola $h_1$ for $O_i$ and $O_j$: $\frac{x_\theta^2}{a_1^2} - \frac{y_\theta^2}{b_1^2} = 1$, where

$$\begin{cases} a_1 = \frac{r_i + r_j}{2}, \quad c_1 = \frac{dist(c_i, c_j)}{2}, \quad and \quad b_1 = \sqrt{c_1^2 - a_1^2} \\ x_\theta = (x - f_x)cos\theta + (y - f_y)sin\theta \\ y_\theta = (f_x - x)sin\theta + (y - f_y)cos\theta \end{cases}$$

Suppose $\mathcal{L}$ is on straight line $l_1$: $a_2x + b_2y + c_2 = 0$
Let $\Phi$ be the roots of the equation group consisting of $h_1$ and $l_1$:

$$\begin{cases} h_1 : \frac{x_\theta^2}{a_1^2} - \frac{y_\theta^2}{b_1^2} = 1 \\ l_1 : a_2x + b_2y + c_2 = 0 \end{cases}$$

**for** each $\phi \in \Phi$ **do**
  **if** $\phi$ is on $\mathcal{L}(s,e)$ **then**
    $R = R \cup \phi$;
return $R$;

**Algorithm 2** Nested-Loop

**function** NESTED-LOOP(Trajectory $\mathbb{T}$)
**for all** line segment $L \in \mathcal{T}$ **do**
  **for** $i = 1 \ldots n$ **do** ▷ consider object $O_i$
    **for** $j = i + 1 \ldots n$ **do** ▷ consider object $O_j$
      $\mathcal{I}$ = `FindIntersection`$^e(L, O_i, O_j)$ (Algorithm 1);
      Verify $\mathcal{I}$ and delete unqualified elements;
Evaluate *PNN*s for each interval and merge two successive ones if they have same *PNN*s;

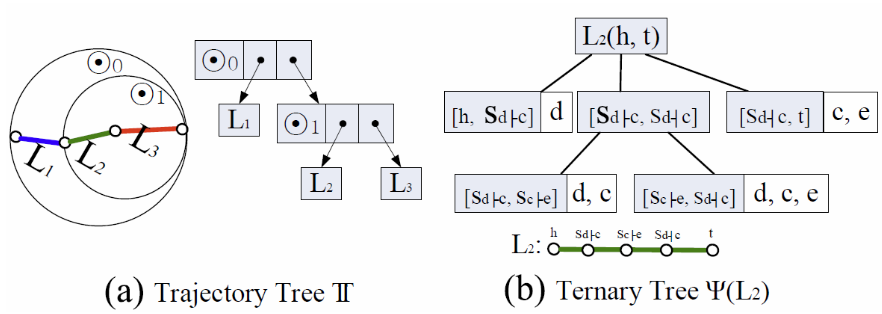



Figure 6: Trajectory Tree $\mathbb{T}(\mathcal{T})$ and Ternary Tree $\Psi(L_2)$

In Algorithm 2, suppose *Step 5* can be done in a constant time $\beta$. *Step 6* can be finished in $O(\log n)$. Suppose $\mathcal{T}$ contains $l$ line segments, then Nested-Loop's total time complexity is $O(l\ n^2(\log n + \beta))$. Nested-Loop is not efficient because it does not prune unqualified objects early in query evaluation but exclude them by late verifications. Next, we present a Filter-Refinement query evaluation approach that effectively prunes those unqualified objects that cannot be *PNN* for any point on the query trajectory.

### 2.3.2 Filter-Refinement Approach

In this section, we present a *filter-refinement* framework for evaluating *TPNNQ*. We assume an R-tree $\mathbb{R}$ is built on the imprecise objects in $O$ and it can be stored in the main memory, as the memeory capabilities increase fast in recent years.

Suppose a query trajectory $\mathcal{T}$ is represented as a series of consecutive line segments, i.e., $\mathcal{T} = \langle L_1, L_2, \ldots, L_l \rangle$, we organize $\mathcal{T}$ using a binary trajectory tree $\mathbb{T}(\mathcal{T})$. Each binary tree node $T_i = \langle L_1, ..., L_{l'} \rangle$ has two children: $T_i.left = \langle L_1, ..., L_{\lfloor \frac{l'}{2} \rfloor} \rangle$ and $T_i.right = \langle L_{\lceil \frac{l'}{2} \rceil}, ..., L_{l'} \rangle$. The trajectory tree for $\mathcal{T} = \langle L_1, L_2, L_3 \rangle$ is shown in Figure 6(a).

The data structure for each binary tree node $T_i$ is a triple: $T_i = \langle L, MBC, Guard \rangle$. Specifically, $L$ is a line segment if $T_i$ is a leaf-node and *NULL* otherwise, $MBC$ is the minimum bounded circle covering $T_i$ or *NULL* for leaf-nodes, and $Guard$ is an entry which keeps minimum and maximum distances to $T_i$. The $Guard$ entry can be either an R-tree node or an imprecise object. Note such $Guard$ entries are not initialized until processing *TPNNQ* is started. Since $\mathcal{T}$ contains $l$ line segments, the trajectory tree $\mathbb{T}(\mathcal{T})$ is constructed in $O(l \log l)$ time.

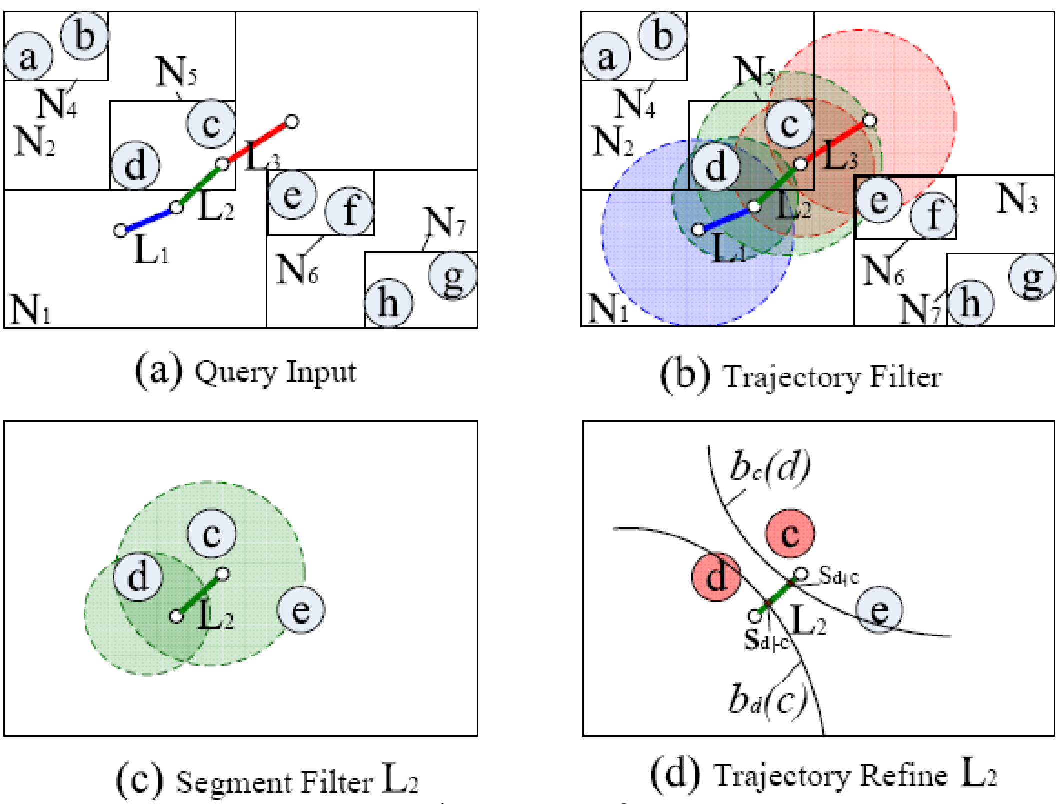
The pseudo code for the filter-refinement framework is shown in Algorithm 3. It takes a trajectory tree $\mathbb{T}$ and an R-tree $\mathbb{R}$ as input. The filtering phase is equipped with two filters. *Trajectory Filter* (line 3) retrieves candidate objects from $O$ such that only those objects that can be the closest objects to the query trajectory $\mathcal{T}$. All other imprecise objects are filtered due to their long distances to $\mathcal{T}$. *Segment Filter* (lines 4–5) further prunes unqualified candidate objects for each line segment $L_i \in \mathcal{T}$. Our previous work [15] elaborates on how the two filters work with trees $\mathbb{T}$ and $\mathbb{R}$. We skip the details here due to the page limit.

The refinement phase evaluates all the *validity interval*s and *turning point*s for each line segment in $\mathcal{T}$. This phase is encapsulated in function `TernaryDecomposition(.)`, to be detailed in Section 3. Finally, all derived *validity interval*s are scanned once and consecutive ones are merged if they belong to different line segments but have the same set of *PNN*s (line 7).

***Example of TPNNQ*** Refer to Figure 7(a). A query trajectory $\mathcal{T} = \{L_1, L_2, L_3\}$ is given, and an R-tree is built on imprecise objects $O = \{a, b, c, d, e, f\}$. We use *trajectory filter* to derive $\mathcal{T}$'s trajectory filtering bound, as shown by shaded areas in Figure 7(b). 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. The *segment filter* is applied for each line segment in $\mathcal{T}$. Taking $L_2$ as an example, the

**Algorithm 3** TPNNQ

**function** TPNNQ(Trajectory $\mathcal{T}$, R-tree $\mathbb{R}$)
  let $\phi$ be a list (of candidate objects);
  $\phi \leftarrow$`TrajectoryFilter`$(\mathbb{T}, \mathbb{R})$;
  **for all** line segment $L_i \in \mathcal{T}$ **do** $\qquad \triangleright\ \mathcal{T} = \{L_i\}_{i \leq l}$
    $\phi_i \leftarrow$ `SegmentFilter`$(L_i, \phi)$;
    $\{\langle L, R \rangle\}_i \leftarrow$ `TernaryDecomposition`$(L_i, \phi_i)$;
  $\{\langle T_i, R_i \rangle\}_{i=1}^{t} \leftarrow$ `Merge`$(\cup_{i=1}^{l} \{\langle L, R \rangle\}_i)$;




(a) Query Input (b) Trajectory Filter

(c) Segment Filter L2 (d) Trajectory Refine L2

Figure 7: TPNNQ

segment filtering bound is shown as Figure 7(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 *Ternary Decomposition* for each line segment to derive the *turning points*. As shown in Figure 7(d), 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. Thus, $L_2$ 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 accordingly, as shown in Figure 6(b). Its root node has three children, each corresponding to a sub-line-segment. These refinement steps recur for each of the three sub-line-segment. Finally, the process stops and a complete ternary tree $\Psi(L_2)$ is constructed when no further split is possible.

Note that the degree of a ternary tree node is at most 3, since a line segment is split into at most 3 sub-line-segments (guaranteed by Theorem 2). Subsequently, the query result for $L_2$ can be fetched by traversing the leaf-nodes of $\Psi(L_2)$. Therefore, we have *TPNNQ*$(L_2)$ $=$ $\{\langle[h, s_{d\vdash c}], \{d\}\rangle, \langle[s_{d\vdash c}, s_{c\vdash e}], \{c, d\}\rangle, \langle[s_{c\vdash e}, s_{d\dashv c}], \{c, d, e\}\rangle,$
$\langle[s_{d\dashv c}, t], \{c, e\}\rangle\}$. The results for $L_1$ and $L_3$ can be obtained likewise.

We proceed to present the refinement process that is done for each line segment in the query trajectory.

# 3 Refinement Process for A Line Segment in Query Trajectory

In the filter-refinement query evaluation framework, we do the refinement for each line segment $L_i$ in the query trajectory $\mathcal{T}$. In particular, we need to find turning points and validity intervals for a line segment $L_i$. We find them a recursive manner. At each iteration, we use a $u$-bisector to split the current line segment into a number of sub-line-segments. We classify the sub-line-segments into different categories and derive the specified pruning bound for each category in order to eliminate disqualified objects. The process repeats until the current intervals can not be further split. Since the current line segment is decomposed into at most 3 parts due to the at most 2 intersections, we name our algorithm *ternary decomposition*. Essentially, the process is equivalent to constructing a *ternary tree* $\Psi(L_i)$ for $L_i$.

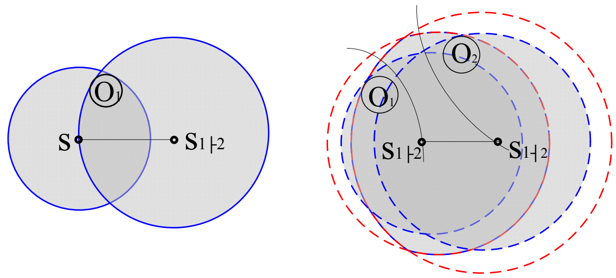
In the sequel, we introduce categories of pruning bounds in Section 3.1. Based on that, we design the ternary decomposition algorithm in Section 3.2.

## 3.1 Pruning Bounds for Three Cases

A query line segment $L_i(s, e)$ can be divided by a $u$-bisector (Definition 3) into at most 3 sub-line-segments. With respect to their positions in half spaces, there are three types of sub-line-segments: *Open Case*, *Pair Case*, and *Close Case*. Refer to Figure 4 for the sake of easy presentation. *Close Case* means the sub-line-segment is totally covered by $H_i(j)$ or $H_j(i)$. *Open Case* means the sub-line-segment is totally covered by $V(i, j)$, except that one of its endpoints is on $b_i(j)$ or $b_j(i)$. *Pair Case* means the sub-line-segment's two endpoints are on $b_i(j)$ and $b_j(i)$ respectively, and all its remaining points are in $V(i, j)$. The three cases are formally described in Table 3.

Table 3: 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 start(end) point of the line segment) omitted |
| close | $[s_{i\vdash j}, s'_{i\vdash j}]$ | $l \in H_i(j)$ and $s_{i\vdash j}, s'_{i\vdash j} \in b_i(j)$ |



(a) Open Case (b) Pair Case

Figure 8: Open Case and Pair Case

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 show the pruning bound derived for $[s, s_{1\vdash 2}]$ and $[s_{1\vdash 2} s_{1\dashv 2}]$ in Figure 8 (a) and (b). The bounds are highlighted by shaded areas. Note that any object $O_i$ beyond the bounds are safely pruned for the corresponding sub-line-segments as it cannot be closer to the The pruning bound of $[s_{1\dashv 2}, e]$ is similar to Figure 8(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 know that $[s_{i\vdash j}, s'_{i\dashv j}]$ must be in $H_i(j)$, so $O_j$ cannot be the *PNN* for each point inside. We design their pruning bounds in the following.

**Lemma 1** **(Pair Case)** *Given two imprecise objects $O_i$ and $O_j$, suppose their $u$-bisector $b_i(j)$ and $b_j(i)$ intersect with a straight line at $s_{i\vdash j}$ and $s_{i\dashv j}$. $\forall q \in [s_{i\vdash j} s_{i\dashv j}]$, an object $O_N$ cannot be $q$'s* PNN *if $O_N$ has no overlap with the pruning bound $\circledcirc(s_{i\vdash j}, O_i) \cup \circledcirc(s_{i\dashv j}, O_j) \bigcap \circledcirc(s_{i\vdash j}, O_j) \cup \circledcirc(s_{i\dashv j}, O_i)$.*

**Lemma 2** **(Open Case)** *Given an Open Case sub-line-segment $[s, s_{i\vdash j}]$, $\forall q \in [s, s_{i\vdash j}]$, an object $O_N$ cannot be $q$'s* PNN*, if $O_N$ has no overlap with $\circledcirc(s, O_i) \cup \circledcirc(s_{i\vdash j}, O_i)$.*

**Lemma 3** **(Close Case)** *Given an Close Case sub-line-segment $[s_{i\vdash j}, s'_{i\vdash j}]$, $\forall q \in [s, s'_{i\vdash j}]$, an object $O_N$ cannot be $q$'s* PNN*, if $O_N$ has no overlap with $\circledcirc(s_{i\vdash j}, O_i) \cup \circledcirc(s'_{i\vdash j}, O_i)$.*

The proof of Lemma 1 is given in Appendix. As the proofs of Lemma 2 and Lemma 3 can be easily derived from Lemma 8 (given and proved in Appendix), they are omitted due to page limit.

The *Pair Case* can also be considered as the union 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 $[s_{i\vdash j}, e]$. Moreover, the *Close Case* can be viewed as the union of $[s, s'_{i\dashv j}]$ and $[s_{i\vdash j}, e]$. The three cases and their combinations cover all possibilities for each piece (validity interval) of a query line segment $L_i$. After the ternary tree $\Psi(L_i)$ is constructed for $L_i$, we can derive the pruning bound of a validity interval. It is the intersection of all its ascender nodes' pruning bounds in the ternary tree $\Psi$.

## 3.2 Ternary Decomposition

The ternary decomposition constructs the ternary tree $\Psi$ in an iterative manner, as shown in Algorithm 4. At each iteration, we select two objects from the current candidate set $\phi_{cur}$ as seeds to divide the current line-segment $L_{cur}$ into two or three pieces. To split $L_{cur}$, we have to evaluate a feasible $u$-bisector, whose intersections with $L_{cur}i$ are *turning points*. Then, to find the $u$-bisector, we might have to try $\frac{C(C-1)}{2}$ pairs of objects, where $C = |\phi_{cur}|$. In fact, the object with the minimum maximum distance to $L_{cur}$, say $O_1$, must be one *PNN*. The correctness is shown in Lemma 4.

**Lemma 4** *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$(L)$.*

---

**Algorithm 4** TernaryDecomposition

---

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

---

Accordingly, the *turning points* on $L_{cur}$ are often derived by $O_1$ and another object among the $C$ candidates. Therefore, the candidates are sorted first in the ternary decomposition. After that, $L_{cur}$ 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 $\phi_{cur}$ (lines 9 to 12). Notice that for each leaf-node $L^i$ of the ternary tree $\Psi(L(s, e))$, $L^i$'s two endpoints must be $s$, $e$, or the *turning points* on $L$. If we traverse $\Psi$ 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 $m + 1$ $\Psi$'s leaf-nodes. Algorithm 4 stops when any pair of objects in $\phi_{cur}^{[L]}$ does not further split $L$.

The complexity of ternary decomposition depends on the size of the *turning points* in the final result. A ternary tree node $T_i$ splits only if one or two intersections are found in $T_i$'s line segment. If no intersections are 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 4 will be called $(1.5m, 2m]$ times. suppose that line 5 in Algorithm 4 is done in time $\beta$ and line 6 is in $O(\log C)$, where $C$ is the number of candidate objects returned by the filtering phase in Algorithm 3. As a result, the complexity of ternary decomposition is $O(mC^2(\log C + \beta))$.

# 4 Supporting Arbitrary Shapes of Imprecise Regions of Objects

So far we have presented our solution for *TPNNQ* where all imprecise objects have circular imprecise regions. It is however possible that imprecise objects take arbitrary shapes of imprecise regions, as illustrated in Figure 1. To handle different shapes, an intuitive way is to enclose an object by a minimum bounding circle (*MBC in short*), and then evaluate the query on the *MBC*s. This makes sense when the imprecise regions can be well represented by MBCs. Otherwise, MBC can introduce considerable dead space, and thus cause many false positives that degrade the query result quality. Hence, it is desirable to have a solution that is more general, reliable, and deployable.

As a matter of fact, the proposed techniques in previous sections can be generalized to arbitrary imprecise region shapes. In particular, to apply the derived techniques (Lemma 1 2 3 and 4), we need to instantiate $dist_{max}(.)$ (or $dist_{min}(.)$) for each specific type of shapes. In addition, we need to consider two important aspects. First, the "2-intersection" fact should hold for other arbitrary. We need to guarantee this in order to make the *Ternary Decomposition* (Section 3) still work. Second, the $u$-bisector's form for arbitrary shaped imprecise regions can be complex. We need to find the turning points (recall Algorithm 1) for the complex case where the $u$-bisector's math representation is not available.

### 4.1 Theories about the $u$-bisector

One important geometric property about the $u$-bisector half $b_i(j)$ is: half space $H_i(j)$ is convex. This property holds even if the imprecise region's shape is concave and irregular. Next, we prove the property formally.

**Theorem 1** ***(Half Space Convexity)*** *Given two imprecise objects $O_i$ and $O_j$, the half space $H_i(j)$ enclosed by the $u$-bisector half $b_i(j)$ is convex.*

**Proof** According to **Midpoint Convexity Theorem** [16], if two arbitrary points $s, e \in H_i(j)$, whose midpoint $m = \frac{s+e}{2}$ satisfies $m \in H_i(j)$, then $H_i(j)$ is convex.

Suppose that two precise points $p_i \in O_i$ and $p_j \in O_j$ satisfy:

$$\begin{cases} dist_{max}(m, O_i) = dist(m, p_i) \\ dist_{min}(m, O_j) = dist(m, p_j) \end{cases} \tag{1}$$

Also,

$$s \in H_i(j) \Rightarrow dist(s, p_i) \leq dist(s, p_j) \tag{2}$$

Similarly,

$$dist(e, p_i) \leq dist(e, p_j) \tag{3}$$

Applying Lemma 10 (see Appendix) to Equations 2 and 3, we have:

$$\begin{aligned} dist(m, p_i) &\leq dist(m, p_j) \\ \Rightarrow dist_{max}(m, O_i) &\leq dist_{min}(m, O_j) \text{ (Equation 1)} \\ \Rightarrow m \in H_i(j) &\Rightarrow H_i(j) \text{ is convex} \end{aligned}$$

The theorem is thus proved.

Based on $H_i(j)$'s convex property, a line segment $\mathcal{L}$ could have at most two intersections with $b_i(j)$. Formally,

**Lemma 5** *Given two imprecise objects $O_i$ and $O_j$, a line segment $\mathcal{L}(s, e)$ has at most two intersection points with the $u$-bisector half $b_i(j)$.*

Since $H_i(j)$ is convex, the query line segment $\mathcal{L}$ is convex, their intersection $l = \mathcal{L} \cap H_i(j)$ must also be convex. Since $l$ is also a part of $\mathcal{L}$, $l$ is a line segment or $\emptyset$. If $l = \emptyset$, $l$ has no intersections with $b_i(j)$. Otherwise, $l$ has at most two intersections with the $u$-bisector half $b_i(j)$, whereas $l$'s two end points are on $H_i(j)$'s boundary.

Likewise, Theorem 1 and Lemma 5 hold for the $H_j(i)$ and $b_j(i)$. Next, we show a more interesting property about the number of intersections between a line segment $\mathcal{L}$ and the $u$-bisector as a whole.

**Theorem 2** ***(Two-intersection Theorem)*** *Given two imprecise objects $O_i$ and $O_j$, a line segment $\mathcal{L}$ has at most two intersections with the $u$-bisector that consists of $b_i(j)$* ***and*** *$b_j(i)$.*

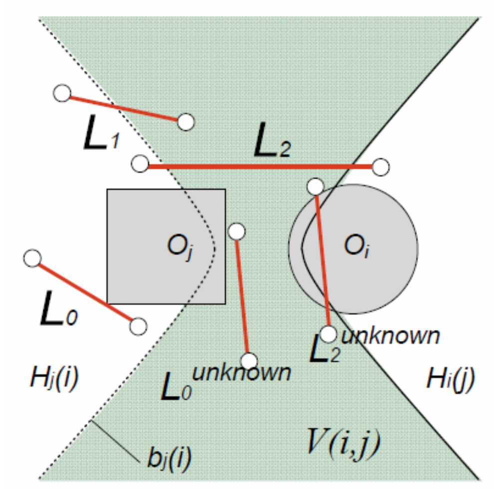



Figure 9: Types of a line segment

**Proof** It is sufficient to show: if $\mathcal{L}$ intersects with $b_i(j)$ at two points, $\mathcal{L} \cap H_j(i) = \emptyset$. In other words, we need to prove for an arbitrary point $t \in \mathcal{L} \wedge t \notin H_i(j)$, $t \notin H_j(i)$.

For circular imprecise regions, the theorem is true according to Lemma 11 (see Appendix). For non-circular imprecise regions, we apply the site decomposition idea [17] to decompose $O_i$ and $O_j$ into two sets of circles $P$ and $Q$. The circles in $P$ or $Q$ may be of different sizes and overlap. An overall half-space $H_j(i)$ is the intersection of all half-spaces $H_{j(q)}(i(p))$ where $p \in P$ and $q \in Q$ (see Lemma 9 in Appendix).

Let $u_i = \{u_{i(p)}\}_{p \in P}$ and $u_j = \{u_{j(q)}\}_{q \in Q}$. For each pair of $u_i(p)$ and $u_j(q)$, we can prove $t \notin H_{j(q)}(i(p))$ according to Lemma 11. Hence, we have:

$$\forall q \in Q \ \ \forall p \in P, t \notin H_{j(q)}(i(p)) \Rightarrow$$
$$t \notin \cap_{p \in P \wedge q \in Q} H_{j(q)}(i(p)) \Rightarrow t \notin H_j(i)$$

Thus, the theorem is true.

Theorem 2 tells that a $u$-bisector can split the query line segment into 3 sub-line segments at most, no matter what shapes the imprecise regions of the two objects have and how complex the form of the $u$-bisector is. Supported by Theorem 2, we proceed to show find to find intersections when arbitrary imprecise region shapes are involved.

### 4.2 Finding Intersections Involving Arbitrary Imprecise Region Shapes

For arbitrary imprecise regions, whose $u$-bisector's mathematical representation is not available, we design an approximated method to find the *intersections*.

Given a line segment and two objects $O_i$ and $O_j$'s $u$-bisector, there can be at most two intersections, as revealed by Theorem 2. We thus classify the line segment into 4 different categories according to the number of intersections, as shown in Table 4. Different cases correspond to different conditions. Referring to the example shown in Figure 9, $L_1$'s two endpoints are located in $H_j(i)$ and $V(i,j)$, so $L_1$ belongs to type 1. Also, $L_0$ and $L_2$ belong to type 0-A and 2-A respectively, according to the conditions listed in Table 4. However, $L_0^{unknown}$ and $L_2^{unknown}$ are two "undetermined" cases. If we only know that one line segment's endpoints are in $V(i,j)$, we can not tell if it is of type 0-B (e.g., $L_0^{unknown}$) or 2-B (e.g., $L_2^{unknown}$). We use $L^{unkown}$ to represent the case that a line segment's two endpoints are in $V(i,j)$. Thus, it is hard to detect which type the $L^{unkown}$ belongs to. We have developed Lemma 7 for type 0-B. Nevertheless, not all cases in type 0-B can be captured. For "undermined" types, we can recursively decompose the line segments, until all the sub-line segments can be classified.

Table 4: Four types of a query line segment

| Intersection count | Condition |
|---|---|
| 0 | A: $\mathcal{L}(s,e) \in H_i(j)$ or $\mathcal{L}(s,e) \in H_j(i)$ (Lemma 6) |
| | B: $\mathcal{L}(s,e) \in V(i,j)$ (Lemma 7) |
| 1 | $s(e) \in H_i(j)/H_j(i)$ and $e(s) \in V(i,j)$ |
| 2 | A: $s \in H_i(j) \wedge e \in H_j(i)$ *or* $s \in H_j(i) \wedge e \in H_i(j)$ |
| | B: $s, e \in V(i,j) \wedge \exists q \in \mathcal{L}, q \in H_i(j)$ *or* $H_j(i)$ |
| Unknown | can be either 0-B or 2-B |

**Lemma 6** *A line segment $\mathcal{L}$ is in the region $V(i,j)$ iff:*

$$\begin{aligned} \forall p \in \mathcal{L}, dist_{max}(p,O_i) > dist_{min}(p,O_j) \\ \wedge\, dist_{max}(p,O_j) > dist_{min}(p,O_i) \end{aligned}$$

**Proof** According to the definition of *half space*:

$$\begin{aligned} p \in V(i,j) \Leftrightarrow p \notin H_i(j) \wedge p \notin H_j(i) \\ \Leftrightarrow dist_{max}(p,O_i) > dist_{min}(p,O_j) \\ \wedge\, dist_{max}(p,O_j) > dist_{min}(p,O_i) \end{aligned}$$

Thus,

$$\begin{aligned} \mathcal{L} \in V(i,j) \Leftrightarrow \\ \forall p \in \mathcal{L}, dist_{max}(p,O_i) > dist_{min}(p,O_j) \wedge \\ dist_{max}(p,O_j) > dist_{min}(p,O_i) \end{aligned}$$

**Lemma 7** *A line segment $\mathcal{L}$ is in the region $V(i,j)$ if:*

$$\begin{aligned} dist_{max}(m,O_i) > dist_{min}(m,O_j) + length(\mathcal{L}) \\ \wedge\, dist_{max}(m,O_j) > dist_{min}(m,O_i) + length(\mathcal{L}) \end{aligned}$$

*where $m$ is the middle point of $\mathcal{L}$.*

**Proof** Let $p$ be an arbitrary point on the line segment $\mathcal{L}$, $x$ be any location in $O_i$, and $r_m = \frac{length(\mathcal{L})}{2}$.

$$\begin{aligned} dist_{max}(m,O_i) > dist_{min}(m,O_j) + length(\mathcal{L}) \Rightarrow \\ dist_{max}(m,O_i) - r_m > dist_{min}(m,O_j) + r_m \end{aligned} \tag{4}$$

We consider the left-hand side of Equation 4 first. Let $y$ be a point of $O_i$ such that $dist_{max}(m,O_i) = dist(m,y)$. By triangle inequality, we have $dist(p,y) \geq dist(m,y) - dist(p,m) = dist_{max}(m,O_i) - dist(p,m)$. As $m$ is the middle point of $\mathcal{L}$, we have $r_m = \frac{length(\mathcal{L})}{2} \geq dist(p,m)$. We also have $dist_{max}(p,O_i) \geq dist(p,y)$. From these three inequalities, we have

$$dist_{max}(p,O_i) \geq dist_{max}(m,O_i) - r_m \tag{5}$$

Likewise, for the right-hand side of Equation 4, we have:

$$dist_{min}(m,O_j) + r_m \geq dist_{min}(p,O_j) \tag{6}$$

Considering Equations 4, 5 and 6 altogether, we have:

$$\forall p \in \mathcal{L},\ dist_{max}(p, O_i) > dist_{min}(p, O_j) \tag{7}$$

Similarly, we can prove

$$\forall p \in \mathcal{L},\ dist_{max}(p, O_j) > dist_{min}(p, O_i) \tag{8}$$

According to Lemma 6, Equations 7 and 8 are sufficient to show $\mathcal{L}$ is in the region $V(i, j)$.

Based on the four types of a line segment, we compute the intersection points approximately using Algorithm 5. The idea of the approximation is to recursively split the query line segment until the current line segment, which contains the type 1 intersection, is shorter than the precision threshold $T_\epsilon$. We thus return the middle point of the line segment as an intersection.

During the decomposition, we classify the line segments into 4 types following Table 4. If the current line segment is of type 1 or 2, it is decomposed for evaluating intersections. If it is of type 0, the branch is stopped. Otherwise, it is of type $Unknown$, the line segment is also decomposed for clarification. The complexity of Algorithm 5 is $O(\log_{T_\epsilon} |L|)$.

**Algorithm 5** FindIntersection

```
function FINDINTERSECTION(Line segment L(s, e), Objects O_i, O_j)
Parameter: the precision threshold T_ε
    if L contains definitely 0 intersection then
        return NULL;
    else
        m = (s+e)/2;
        if length(L) < T_ε then
            if both s and e are in one of H_i(j), H_j(i) and V(i, j) then
                if L contains definitely 0 or 1 intersection then
                    return m;
        else if ¬(L contains definitely 0 intersection) then
            return FindIntersection([s, m], O_i, O_j) ∪ FindIntersection([m, e], O_i, O_j);
```

# 5 Selectivity Estimation for *TPNNQ*

Accurate selectivity estimation is crucial for query processing in database systems. In LBS, the service provider transmits intermediate results (e.g. $\phi$ in Algorithm 3) to the clients through wireless channels. In such a distributed scenario, the estimation can be used to measure the communication cost between the two ends. Precise estimation also helps in efficient load balancing, if the service provider uses multiple processing units for higher efficiency.

In this section, we study selectivity estimation for *TPNNQ*. We start from the simplest case where the query is a point (Section 5.1). Further, we extend it to query line-segments (Section 5.2) and query trajectories (Section 5.3). We consider the *hexagonal lattice model* [18] [19], as shown in Figure 10, where each object has six neighbors whose centers are equidistant from each other, with distance $d_0$[1]. We assume that the imprecise regions are equal-sized and circular shaped with a radius of $r$.

[1]The centers of uncertainty regions form the vertices of $n$ hexagons, each of which has an area of $\frac{\sqrt{3}d_0^2}{2}$. Since $|D| = n \times \frac{\sqrt{3}d_0^2}{2}$, $d_0 = \sqrt{\frac{2|D|}{\sqrt{3}n}}$.

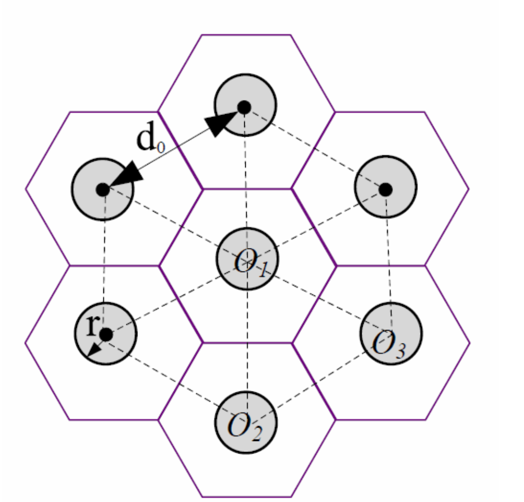



Figure 10: Hexagonal Model

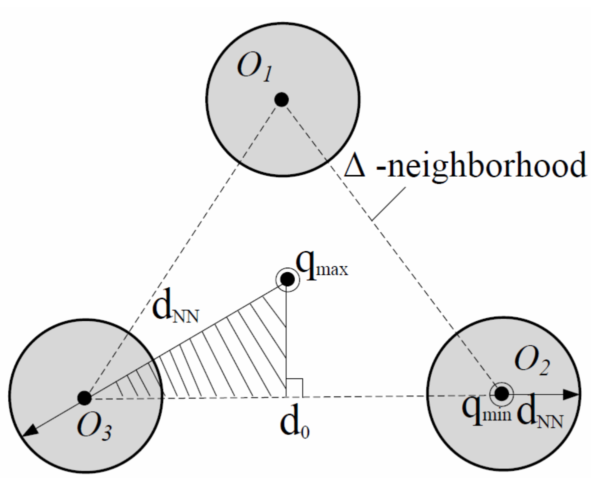



Figure 11: Δ-neighborhood

## 5.1 Result Size Analysis for Query Point

To derive the number of possible nearest neighbors for a given query point $q$, we need to estimate the minimum maximum distance from $q$ to all imprecise objects. We use $d_{NN}$ to denote that distance. Subsequently, the search region of $PNN(q)$ is the circle centered at $q$ with radius $d_{NN}$. Objects that overlap with $\odot(q, d_{NN})$ are qualified as $q$'s possible nearest neighbors [5].

If we connect the centers of two adjacent objects, the domain would be triangulated by dashed lines as shown in Figure 10. Given query point $q$, it must be resided in a triangle. We denote it as **Δ-neighborhood**, which consists of three objects, as shown in Figure 11. Among the three, there must be one object having the minimum maximum distance $d_{NN}$ to $q$, since these three objects are closer than others outside. Different locations in Δ-neighborhood correspond to different $d_{NN}$s. In Figure 11, $q_{min}$'s $d_{NN}$ is $O_2$'s radius, and $q_{max}$'s $d_{NN}$ is the distance from $q_{max}$ to $O_3$'s center plus $O_3$'s radius. Since $d_{NN}$ is changing over $q$'s locations, the number of *PNN*s also varies. If we define the density $\rho$ as the number of objects over a unit area, then the number of *PNN*s can be measured by the density times the area of the search region. Thus, we can get the expected number of *PNN*s. We first derive the $E(|PNN|)$ for the shaded area (*in Figure 11) denoted as* $\Delta_{shaded}$, and repeat 6 times to cover the entire Δ-neighborhood.

$$
\begin{aligned}
& E(|PNN(q)|_{q\in\Delta\text{-neighborhood}}) \\
= \frac{\int_{q\in\Delta}|PNN(q)|dq}{|\Delta\text{-neighborhood}|} = & \frac{6\cdot\int_{q\in\Delta_{shaded}}|PNN(q)|dq}{6\cdot|\Delta_{shaded}|} \\
= & \frac{6\cdot\int_{q\in\Delta_{shaded}}\rho\pi d_{NN}^2 dq}{6\cdot|\Delta_{shaded}|} \\
= & \frac{\int_0^{\frac{d_0}{2}}\int_0^{\frac{x}{\sqrt{3}}}\rho\pi(\sqrt{x^2+y^2}+r)^2 dydx}{\frac{1}{2}\cdot\frac{d_0}{2}\cdot\frac{d_0}{2\sqrt{3}}} \\
= & \rho\pi[\frac{\sqrt{3}d_0(24r+5\sqrt{3}d_0+18r\log_2\sqrt{3})}{108}+r^2]
\end{aligned}
$$

Also, since $\rho$ and $\pi$ are independent, by extracting them we can derive $E(d_{NN})$:

$$E(|PNN(q)|) = \rho\pi E^2(d_{NN}) \tag{9}$$

In the sequel, we simply use $d_{NN}$ to represent $E(d_{NN})$.

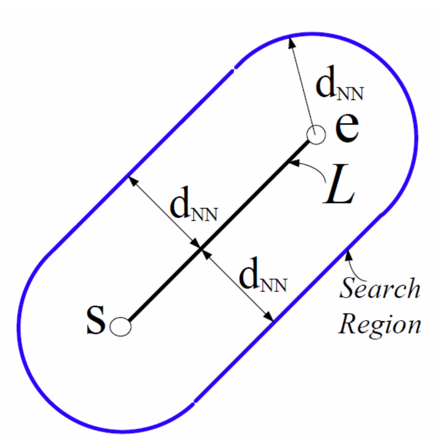



Figure 12: $|PNN(\mathcal{L})|$

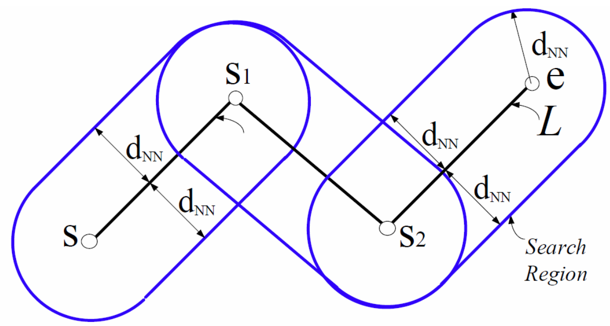



Figure 13: $|PNN(\mathcal{T})|$

### 5.2 Result Size Analysis for Query Line Segment

If the query is a line-segment instead of a point, the search region would be the union of search regions for all points on the line-segment. We show an example of the search region of line-segment $\mathcal{L}$ in Figure 12. The number of $\mathcal{L}$'s *PNN*s can be calculated as the product of density $\rho$ and the search region's area.

$$E(|PNN(\mathcal{L})|) = \rho\pi(2 \cdot d_{NN} \cdot |\mathcal{L}| + \pi \cdot d_{NN}^2) \tag{10}$$

### 5.3 Result Size Analysis for Query Trajectory

Now we extend the estimation from query line-segments to query trajectories. Suppose trajectory $\mathcal{T}$ is represented by $\{\mathcal{L}_1, \ldots, \mathcal{L}_l\}$, where successive line-segments $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$ are connected by point $s_i$. Then, $\mathcal{T}$'s search region equals to the union of all $\mathcal{L}_i$'s search regions, as shown in Figure 13. The union could be well approximated by the summation of all line-segments ($\{L_i\}$)'s search regions subtracting all connecting points ($\{s_i\}$)'s search regions.

$$\begin{gathered} E(|PNN(\mathcal{T})|) \approx \\ \sum_{\mathcal{L}_i \in \mathcal{T}} E(|PNN(\mathcal{L}_i)|) - \sum_{s_i \in \mathcal{T}} E(|PNN(s_i)|) \end{gathered} \tag{11}$$

The analysis above can be extended to other object distributions as follows. We apply an equal-sized histogram which splits the domain into $m \times m$ squares. For each square $s$, we assume the objects are uniformly distributed inside. We count the number of objects $N(s)$ of square $s$. Thus, the density $\rho(s)$ of $s$ is collected by $\frac{N(s)}{|D|/(m \times m)}$. We take the average density $\bar{\rho}$ for all squares overlapping with the query trajectory $\mathcal{T}$ [2], and substitute them into Equation 11 to get the estimation.

# 6 Experimental Evaluation

In this section we report on the experimental results on different datasets. Section 6.1 describes the relevant settings. Section 6.2 gives a metric to measure to quality of query results. Section 6.3 presents the experimental results.

[2] Other parameters such as $\bar{d}_0$ and $\bar{r}$ are obtained similarly.

## 6.1 Experimental Settings

***Queries*** The query trajectories are generated by Brinkhoff's network-based mobile data generator[3]. 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[4], namely *germany*, *LB*, *stream* and *block* with 30K, 50K, 199K, 550K spatial objects, respectively. We also construct the MBC for each object and get 4 other datasets with circular imprecise regions[5]. We use *stream* as the default dataset. Datasets are normalized to the same domain as queries.

To index imprecise regions, we use a packed R*-tree [20]. 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.

All our programs were implemented in C++ and tested on a Core2 Duo 2.83GHz PC enabled by MS Windows 7 Enterprise.

## 6.2 Query Result Quality Metric

As *TPNNQ* queries over imprecise objects, it is interesting to measure the query result quality. We adopt an *Error* function based on the *Jaccard Distance* [21], which measures the similarity between two sets. Recall that the query result of *TPNNQ* is a set of tuples $\{\langle T_i, R_i\rangle\}$. It can be transformed into the *PNN*s for every point on the query trajectory $\mathcal{T}$, i.e., $\{\langle q, PNNQ(q)\rangle\}_{q\in\mathcal{T}}$. Let $R^*(q) = PNNQ(q)$ be the ground-truth query result for a point $q$. We use $R^A(q)$ to represent the *PNN*s returned by algorithm $A$ for the point $q$. The *Error* for algorithm $A$ on query $\mathcal{T}$ is:

$$Error(\mathcal{T}, A) = \frac{1}{|\mathcal{T}|}\int_{q\in\mathcal{T}} 1 - \frac{R^*(q)\cap R^A(q)}{R^*(q)\cup R^A(q)}dq \tag{12}$$

Here, $|\mathcal{T}|$ 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 12 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, it acquires a high error score.

## 6.3 Performance Results

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 defined in Section 6.2. To evaluate the filter-refinement query evaluation framework (Algorithm 3), we list several competitors: *Nested-Loop*, *Sample*, TP-S, TP-TS, and TP-TS$^e$. The suffixes $T$ and $S$ refer to *Trajectory Filter* and *Segment Filter*, respectively. *Nested-Loop* does not use any filter; TP-S does not use *Trajectory Filter*; TP-TS and TP-TS$^e$ (Algorithm 3) use all the filtering and refinement techniques. *Sample* draws a set of uniform sampling points

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

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

[5]We handle other shaped imprecise regions in Section 6.3.4

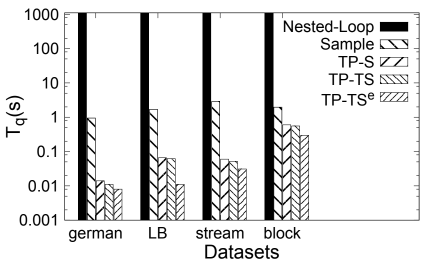



Figure 14: $T_q$(s) vs. Datasets.

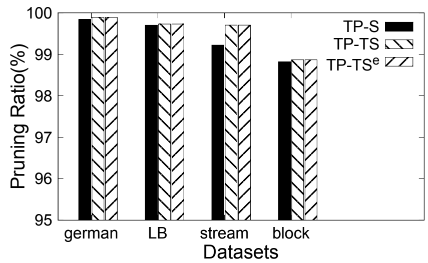



Figure 15: Pruning Ratio vs. Datasets

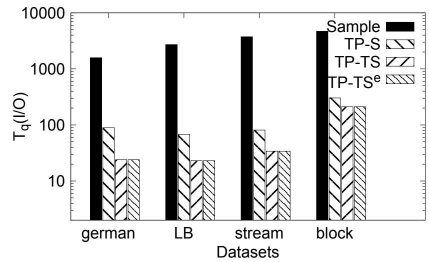



Figure 16: $T_q$(# of node access) vs. Datasets

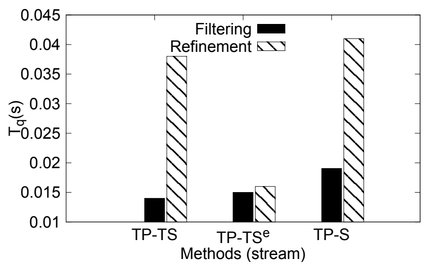



Figure 17: $T_q$'s breakdown

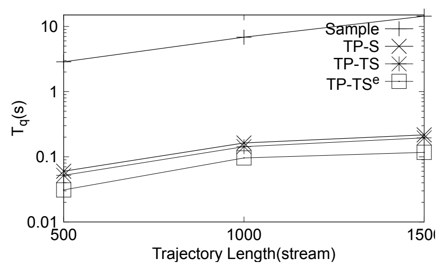



Figure 18: $T_q$ vs. Query Length

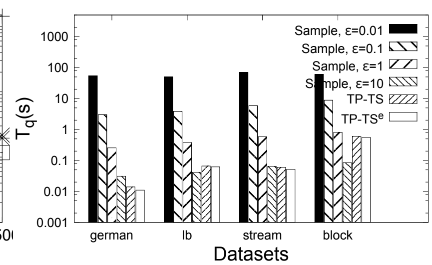



Figure 19: TP-TS vs. Sample ($T_q$)

$\{q\}$ from $\mathcal{T}$. Then, for all $q$, $PNNQ(q)$ is evaluated. The sampling interval, denoted by $\epsilon$, is set to 0.1 unit by default[6].

As discussed, we either use *FindIntersection*$^e$ (Algorithm 1) to find exact turning points or *FindIntersection* (Algorithm 5) to find approximated turning points. The superscript $e$ indicates the exact intersection calculation. So, TP-TS$^e$ derives exact turning points for circular regions, while TP-TS calculates approximate turning points for arbitrary shaped regions. For *FindIntersection*$^e$, we call GSL Library[7] to get the analytical solution. For *FindIntersection*, the default $T_\epsilon$ is set to 0.01 unit.

### 6.3.1 Query Efficiency $T_q$

According to the results shown in Figure 14, the *Nested-Loop* method is the slowest among all. It elaborates all the possible pairs of objects for *turning points* (but most of them do not contribute to validity intervals). Next, *Sample* comes the second slowest. We analyze it in Section 6.3.2.

The other three methods have significant improvement over *Sample* and *Nested-Loop*. One reason is because of the effectiveness of the pruning techniques, as shown in Figure 15. 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 and TP-TS$^e$. Notice that gap would be bigger if the query trajectory consists of many tiny line segments. Also, the combined traversal over R-tree in TP-TS and TP-TS$^e$ save plenty of extra I/O cost, compared to TP-S, shown in Figure 16.

To get a clearer picture about the efficiency, we measure the time costs for *Filtering* and *Refinement* in Figure 17. TP-TS and TP-TS$^e$ are faster than TP-S in both phases. In *Filtering*, the combined R-tree traversal in TP-TS and TP-TS$^e$ save plenty of extra node access, compared to TP-S. The number of node access is shown in Figure 16. In *Refinement*, TP-TS and TP-TS$^e$ are faster, since they has fewer candidates to handle. The observation is consistent with the fact that TP-TS has a higher pruning ratio, shown in

[6] The sampling rate is reasonably high regarding to the trajectory's default length. More details about sampling rates are discussed in Section 6.3.2.

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

Table 5: TP-TS vs. Sample (Error)

| Datasets | Sample | | | | TP-TS |
|---|---|---|---|---|---|
| | $\epsilon$ = 0.01 | $\epsilon$ = 0.1 | $\epsilon$ = 1 | $\epsilon$ = 10 | |
| german | 0.00340 | 0.00457 | 0.01528 | 0.12310 | 6.62e-6 |
| LB | 0.00005 | 0.00029 | 0.00257 | 0.02672 | 5.90e-5 |
| stream | 0.00059 | 0.00090 | 0.00298 | 0.03962 | 6.06e-4 |
| block | 0.01872 | 0.02541 | 0.08516 | 0.44310 | 5.80e-4 |

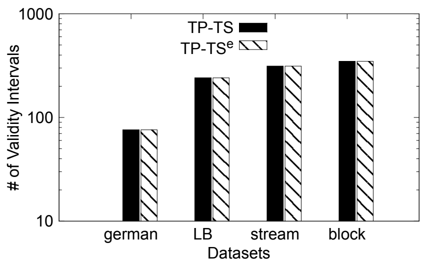



Figure 20: ♯ of Validity Intervals vs. Datasets (TP-TS)

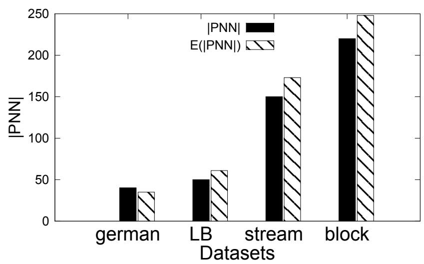



Figure 21: Estimation of $|PNN|$

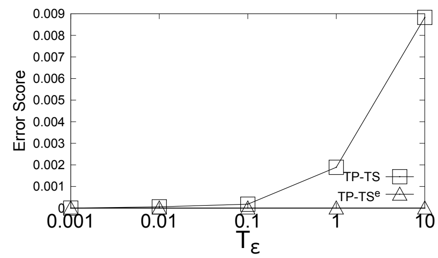



Figure 22: Error vs. Precision (stream)

Figure 15. TP-TS$^e$ directly derives *turning points* by analytical solution, which is more efficient than TP-TS.

We also test the query efficiency by varying the query length in Figure 18. The *Sample* method is slower than others at least one order of magnitude. The costs of the other three increase stably w.r.t. the query length. TP-TS and TP-TS$^e$ are faster.

### 6.3.2 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 requires accessing of R-tree. As shown in Figure 16, *Sample* incurs at least one order of magnitude more node access than our methods.

On the other hand, *Sample* could incur *false negatives*, even with a large sampling rate. Because *Sample* only considers sampled points on the trajectory, whereas *TPNNQ* is for all the points in $\mathcal{T}$. To calculate *Sample*'s error score, we have to infer the *PNN*s for a point $q \in \mathcal{T}$ not being sampled, as required by Equation 12. With limited sampled answers, $q$'s *PNN*s 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 19, where the sampling interval $\epsilon$ 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 $\epsilon$ is very large (e.g. equal to 10 units). Then, is it good if large $\epsilon$ is used? The answer is **NO**. In Table 5, when "*Sample, $\epsilon$ = 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 5. The error of *Sample* is small when $\epsilon$ 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 *Sample* to contain 0 false negative. Compared to them, the error score of TP-TS is much lower.

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

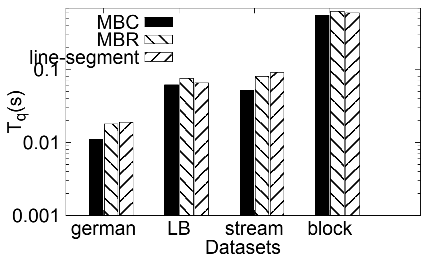



Figure 23: $T_q$ vs. Shapes (TP-TS)

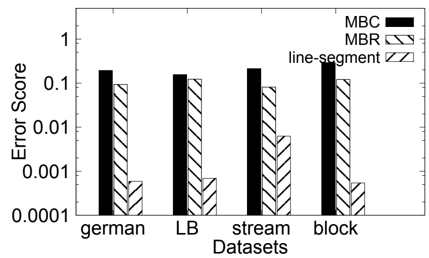



Figure 24: $Error$ vs. Shapes (TP-TS)

### 6.3.3 Analysis of TPNN

Observed from Figure 20, the number of validity intervals increases with the size of the datasets. TP-TS$^e$ has the same number of validity intervals, which means the approximate calculation is capable of deriving the *turning points* within a limited precision $T_\epsilon$.

We also test our proposed analysis model in Figure 21. We split the domain into $25 \times 25$ squares and use average parameters as input. The number of $PNN$s increase with the size of datasets. In all tested cases, the error rate is within 5%, which shows a high accuracy of the selectivity estimation.

We test the *error score* of the TP-TS w.r.t. the increase of precision $T_\epsilon$. As shown in Figure 22, when $T_\epsilon < 0.1$, the *error score* of TP-TS is quite close to the value of TP-TS$^e$, which is 0. This offers us flexibility in choosing the parameter $T_\epsilon$. When $T_\epsilon > 0.1$, the *error score* increases significantly w.r.t. $T_\epsilon$. In our implementation, we set $T_\epsilon$ to 0.01. It is possible to sacrifice some precision for a faster query execution. However, the quality will decrease accordingly. More details are omitted due to page limit.

### 6.3.4 Objects with Different Shaped Imprecise Regions

We model the moving objects on a road network by an imprecise region, whose shape is a line segment. For experiments, we reuse the 4 real rectangular datasets by using each rectangle's two opposite corners as two end-points of a line segment. Then, we test how the quality will be affected by representing the line segment with its enclosed MBC or *MBR*(Minimum Bounding Rectangle). We also investigate how the query performance varies for the three different shapes: circle, rectangle, and line segment.

The queries are implemented by TP-TS method. Figure 23 shows that the $T_q$s are similar for the three shapes we have tested. $T_q$ on the circular dataset is a little bit faster, as the max/min distance evaluation for circular objects requires less distance comparisons than the other two.

However, the quality of a query evaluated over objects' approximated MBCs or MBRs could decrease significantly. In Figure 24, the approximated MBC'e error is as high as 0.15. The error of MBR is lower than MBC because MBR has smaller dead space while enclosing a line segment. But the error is still too large comparing to the result evaluated on the line segments. In real deployment, if the object could be well represented by its MBC or MBR, we suggest to use MBC or MBR for better efficiency. Otherwise, the shape of objects should be considered to achieve better query quality.

In summary, we have shown that TP-TS is much more efficient than *Nested-Loop* and *Sample* methods. It also achieves much better quality than *Sample* method. With the approximated algorithm for finding intersections, the solution for TPNN can be extended to arbitrary shaped imprecise regions (tested by rectangular objects). For some simple shapes (e.g., circular objects), the exact intersections can be found by TP-TS$^e$ with 0 error.

# 7 Related Work

In this section, we review the related work on moving nearest neighbor queries (Section 7.1), as well as the evaluation of trajectory nearest neighbor queries for imprecise location data (Section 7.2).

## 7.1 Moving Nearest Neighbor Query

Nearest neighbor (NN) query for moving query points is a well studied topic [22] [23] [24] [14]. Most existing works focus on reducing the computational cost at the server. They fall into two major categories.

The first category does not require the user's entire trajectory in advance [22] [23] [24], but processes the query online (multiple times) based on the user's moving location.

Song and Roussopoulos [22] 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. Some others [23, 24] use validity region and validity time for the query answer of moving points. Voronoi cells are used 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 only once [14]. In particular, 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 $p_j$.

The query trajectory in our *TPNNQ* setting, such as a flight route or a pipeline, is known in advance. However, the exising technique [14] is not applicable to our problem on imprecise location data. As shown in Figure 2, 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. They use bisectors for specific shapes (circles [8] [9], rectangles [10]) to determine the dominance relationship between objects. This paper distinguishes itself from these works in several important aspects.

First, the query studied in this paper is issued for a trajectory, but not for a single object. Second, the $u$-bisector defined in this paper is extended to support arbitrary shaped imprecise objects. It is however unknown how the existing bisectors [8, 9, 10] can be generalized for similar purposes. Third, our query evaluation partitions the query trajectory into several segments each of which has its own answer set. In contrast, these previous works [8, 9, 10] do not partition their query objects.

## 7.2 Trajectory Nearest Neighbor Query over Uncertain Data

Only a few works have addressed trajectory queries over imprecise data. Chen et al. [11] study the problem of updating answers for continuous probabilistic nearest neighbor queries in the server was studied. Computational overhead is saved if the query answers are within specific probabilistic bounds. Trajcevski et al. [12] investigate the problem of efficiently executing continuous *NN* queries for uncertain moving objects trajectories. Zheng et al. [13] study two variants of k-*NN* query for fuzzy objects. They return the qualified objects satisfying a probabilistic distance threshold or a range of probability thresholds, respectively.

We use the imprecise region model in this paper. It allows us to know which object may be the closest to a given trajectory. In contrast, the uncertainty model described in [11, 12, 13] contains a probability distribution, which describes the chance that an imprecise object is located in each point in the imprecise region. With this more complex uncertainty model, it is possible to quantify the probability that an imprecise object is the nearest neighbor of any point in a given trajectory. Note such a problem is beyond the scope of this paper and therefore we leave it for future research.

Park et al. [7] study a similar problem as we do in this paper. They also use an imprecise region to model the locations of an object and compute the object closest to a given query segment. However, they only compute and return the definite nearest neighbors but ignore objects that may be the closest. This simplification renders significant answer loss in the query result. Also, unlike our solution in this paper, the techniques in [7] are specific to circular objects and are inapplicable to arbitrary shaped imprecise objects.

# 8 Conclusion

In this paper, we study the problem of trajectory possible nearest neighbor query (*TPNNQ*) over imprecise data. To overcome the low quality and inefficiency in simplified methods, we study the geometric properties of $u$-bisector. Based on that, we design an efficient query evaluation approach that follows the filter-refinement paradigm. We also generalize our solution to arbitrary shaped imprecise data. Further, we propose theoretic analysis to estimate the *TPNNQ* query result size. We conduct extensive experiments to evaluate our proposals. The results show that our query evaluation approach is efficient and scalable. Meanwhile, our *TPNNQ* query result size estimation gives very good hints.

# References


[1] D. C. Scott, "Available now: a volcanic ash detector for aircraft," http://www.csmonitor.com/World/Europe/2010/0420/Available-now-a-volcanic-ash-detector-for-aircraft.

[2] U. S. C. Guard, "Announcement of 2011 international ice patrol services," http://www.uscg.mil/lantarea/iip/docs/AOS_2011.pdf.

[3] "Dow employs gps for pipeline worker safety," http://www.automationworld.com/feature-8577.

[4] 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.

[5] R. Cheng, D. V. Kalashnikov, and S. Prabhakar, "Querying imprecise data in moving object environments," *TKDE*, vol. 16, no. 9, 2004.

[6] H. Lu1, B. Yang, and C. S. Jensen, "Spatio-temporal joins on symbolic indoor tracking data." in *ICDE*, 2011.

[7] K. Park, H. Choo, and P. Valduriez, "A scalable energy-efficient continuous nearest neighbor search in wireless broadcast systems," *Wireless Networks*, vol. 16, no. 4, pp. 1011–1031, 2010.

[8] R. Cheng, X. Xie, M. L. Yiu, J. Chen, and L. Sun, "Uv-diagram: A voronoi diagram for uncertain data," in *ICDE*, 2010.

[9] X. Lian and L. Chen, "Efficient processing of probabilistic reverse nearest neighbor queries over uncertain data." in *VLDBJ*, 2009.

[10] M. A. Cheema, X. Lin, W. Wang, W. Zhang, and J. Pei, "Probabilistic reverse nearest neighbor queries on uncertain data." in *TKDE*, 2010.

[11] 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.

[12] 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.

[13] K. Zheng, G. P. C. Fung, and X. Zhou, "K-nearest neighbor search for fuzzy objects," in *SIGMOD*, 2010.

[14] Y. Tao, D. Papadias, and Q. Shen, "Continuous nearest neighbor search," in *VLDB*, 2002.

[15] X. Xie, R. Cheng, and M. L. Yiu, "Evaluating trajectory queries over imprecise location data." in *SSDBM*, 2012.

[16] K. Nikodem, "Midpoint convex functions majorized by midpoint concave functions," *Aequationes Mathematicae*, vol. 32, pp. 45–51, 1987.

[17] A. Okabe, B. Boots, K. Sugihara, and S. Chiu, *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, 2nd ed. Wiley, 2000.

[18] W. Stallings, *Wireless Communications & Networks (2nd Edition)*. Prentice-Hall, Inc., 2004.

[19] X. Xie, R. Cheng, M. Yiu, L. Sun, and J. Chen, "Uv-diagram: a voronoi diagram for uncertain spatial databases," *The VLDB Journal*, 2012.

[20] M.Hadjieleftheriou, "Spatial index library version 0.44.2b." [Online]. Available: http://u-foria.org/marioh/spatialindex/index.html

[21] P.-N. Tan, M. Steinbach, and V. Kumar, "Introduction to data mining," in *Addison-Wesley*, 2006.

[22] Z. Song and N. Roussopoulos, "K-nearest neighbor search for moving query point," in *SSTD*, 2001.

[23] Z. Baihua and L. Dik, "Semantic caching in location-dependent query processing," in *Advances in Spatial and Temporal Databases*, 2001.

[24] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee, "Location-based spatial queries." in *SIGMOD*, 2003.

# 9 *tech report's appendix

**Proof of Lemma 1.**

**Proof** $\forall p \in [s_{i\Vdash j} s_{i\dashv j}]$, both $O_i$ and $O_j$ have chance to be $p$'s PNN. According to Lemma 8, 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:

$$\circledcirc(s_{i\Vdash j}, O_i) \cup \circledcirc(s_{i\dashv j}, O_i) \bigcap \circledcirc(s_{i\Vdash j}, O_j) \cup \circledcirc(s_{i\dashv j}, O_j) \tag{13}$$

**Proof of Lemma 4.**

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

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

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

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

So, $O_1$ definitely belongs to $TPNNQ(\mathcal{L})$.

**Lemma 8** *Given two imprecise objects $O_i$, $O_j$ and a line-segment $\mathcal{L}(s, e)$, $O_j$ can not be $p \in \mathcal{L}$'s $PNN$ if $O_j$ does not overlap with $\circledcirc(s, O_i) \cup \circledcirc(e, O_i)$.*

**Proof** $O_j$ is not $p \in \mathcal{L}$'s $PNN$ given $O_i$, if and only if $p \in \mathcal{L}$ is in half space $H_i(j)$. Since $H_i(j)$ is convex, if $\mathcal{L}$'s two endpoints $s$ and $e$ are in $H_i(j)$, $p \in \mathcal{L}$ is in $H_i(j)$. Formally,

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

So, the lemma is proved.

**Lemma 9** ***(Imprecise Region Decomposition)*** *Given imprecise objects $O_i$ and $O_j$, if their imprecise regions are decomposed into two sets of sub-regions $P$ and $Q$, say $u_i = \{u_{i(p)}\}_{p \in P}$ and $u_j = \{u_{j(q)}\}_{q \in Q}$, $H_i(j) = \cap_{p \in P \wedge q \in Q} H_{i(p)}(j(q))$*

**Proof** For an arbitrary point $t$,

$$\begin{aligned} t \in H_{i(p)}(j(q)) \Leftrightarrow dist_{max}(t, u_{i(p)}) \leq dist_{min}(t, u_{j(q)}) \\ \forall p \in P\ \forall q \in Q, t \in H_{i(p)}(j(q)) \Leftrightarrow \\ max_{p \in P}\{dist_{max}(t, u_{i(p)})\} \leq min_{q \in Q}\{dist_{min}(t, u_{j(q)})\} \quad (14) \\ t \in \cap_{p \in P \wedge q \in Q} H_{i(p)}(j(q)) \\ \Leftrightarrow dist_{max}(t, O_i) \leq dist_{min}(t, O_j) \\ \Leftrightarrow t \in H_i(j) \quad (15) \end{aligned}$$

Thus, $H_i(j) = \cap_{p \in P \wedge q \in Q} H_{i(p)}(j(q))$.

**Lemma 10** *Given two triangle $\Delta ABC$ and $\Delta A''B''C''$, $D$ and $D''$ are two midpoints on $BC$ and $B''C''$, respectively. If $|BC| = |B''C''|$, $|AB| \leq |A''B''|$ and $|AC| \leq |A''C''|$, then $|AD| \leq |A''D''|$.*

**Proof** We show $\Delta ABC$ and $\Delta A''B''C''$ in Figure 25 (a) and (c), respectively. Our purpose is to prove: if $c' \geq c$ and $b' \geq b$, then $d'' > d$. In order to prove that, we increase the length of $\overline{AB}$ to get another triangle $\Delta A'B'C'$, shown in Figure 25 (b). Considering the edge length of the three triangles: (b) has one longer

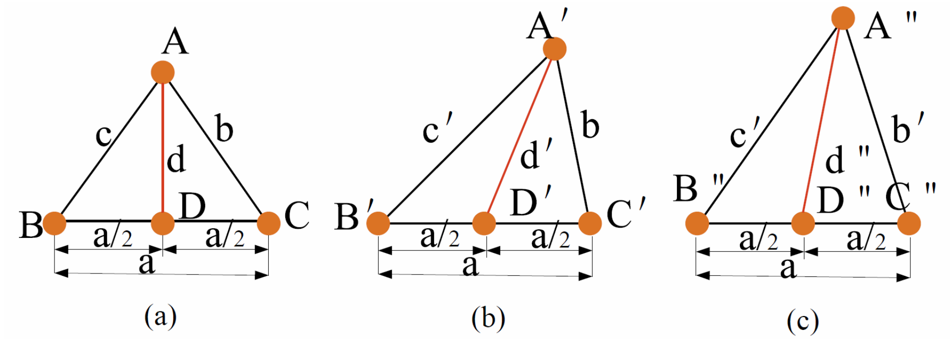



Figure 25: Proof of Lemma 10

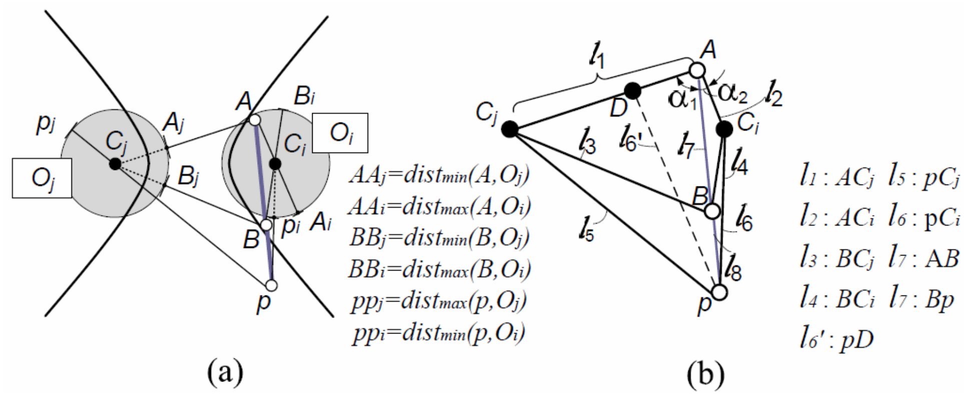



Figure 26: Proof for Lemma 11.

edge than (a) ($c' \geq c$); (c) has one longer edge than (b) ($b' \geq b$). If $d' \geq d$ is true, then $d'' \geq d'$ could be derived similarly. Thus, $d'' \geq d$ could be derived. So, to prove the lemma, we just need to prove $d' \geq d$.

Let $\beta = \angle ACB$ and $\beta' = \angle A'C'B'$. According to the cosine law, we have

$$\cos\beta \geq \cos\beta' \tag{16}$$

Then,

$$\left.\begin{array}{l} d^2 = (\frac{a}{2})^2 + b^2 - 2\cos\beta \cdot \frac{a}{2} \cdot b \\ d'^2 = (\frac{a}{2})^2 + b^2 - 2\cos\beta' \cdot \frac{a}{2} \cdot b \end{array}\right\} \\ \Rightarrow d'^2 - d^2 = ab(\cos\beta - \cos\beta') \geq 0 \\ d' \geq d$$

So, the lemma is correct.

**Lemma 11** *Given two imprecise objects $O_i$ and $O_j$, whose imprecise regions are circles: $O_i(C_i, r_i)$ and $O_j(C_j, r_j)$. A line-segment $\mathcal{L}(s,e)$ has at most two intersection points with the u-bisector: $b_i(j)$ and $b_j(i)$.*

**Proof** Suppose $O_i$ and $O_j$'s uncertainty regions are two circles centered at $C_i$ and $C_j$ with radius $r_i$ and $r_j$, respectively. Since we have known $\mathcal{L}$ intersects with $b_i(j)$ (or $b_j(i)$) at most two points. Then, to prove the lemma, it is sufficient to show: if $\mathcal{L}$ intersects with $b_i(j)$ at two points, $\mathcal{L}$ does not intersect with $b_j(i)$.

Suppose $\overline{AB} = \mathcal{L} \cap H_i(j)$ and $A, B \in b_i(j)$. We extend $\overline{AB}$ to $p$. If we can show for arbitrary point $p$, $p \notin H_j(i)$, the lemma is proved.

Since $A, B \in b_i(j)$, we have $\overline{AA_i} = \overline{AA_j}$ and $\overline{BB_i} = \overline{BB_j}$. $p$ is on the extended line of $\overline{AB}$. According to Equation 1, to show $p \notin H_j(i)$, we just have to show $dist_{max}(p, O_j) > dist_{min}(p, O_i)$, or $\overline{pp_j} > \overline{pp_i}$ as shown in Figure 26(a).

Then, we extend $AA_j$, $BB_j$ to $C_j$ and $pp_i$ to $C_i$. The polygon $\overline{AC_ipC_j}$ is magnified to be Figure 26(b). The problem is change to:

$$\left.\begin{array}{l} l_1 - r_j = l_2 + r_i \\ l_3 - r_j = l_4 + r_i \end{array}\right\} \Rightarrow l_5 + r_j > l_6 - r_i$$

By applying cosine law to $\Delta AC_jB$ and $\Delta AC_iB$:

$$\cos\alpha_1 = \frac{l_1^2 + l_7^2 - l_3^2}{2l_1l_7}, \quad \cos\alpha_2 = \frac{l_2^2 + l_7^2 - l_4^2}{2l_2l_7} \tag{17}$$

After calculation, we can have:

$$\cos\alpha_1 - \cos\alpha_2 < 0 \Rightarrow \alpha_1 > \alpha_2$$

We draw a point $D$ on $\overline{AC_j}$ satisfying $\overline{AD} = l_2$. Then, $\overline{DC_j} = l_1 - l_2 = r_i + r_j$. In $\Delta ADp$ and $\Delta AC_ip$, since $\alpha_1 > \alpha_2$, $\overline{AD} = \overline{AC_i}$, we can have:

$$l_6' > l_6 \tag{18}$$

According to triangle inequality, in $\Delta DC_jp$, $\overline{C_jp} > \overline{pD} - \overline{DC_j}$. It is equivalent to $l_5 > l_6' - (r_i + r_j)$. Together with Equation 18, we have:

$$l_5 > l_6 - (r_i + r_j) \Rightarrow l_5 + r_j > l_6 - r_i$$

Thus, the lemma is proved.