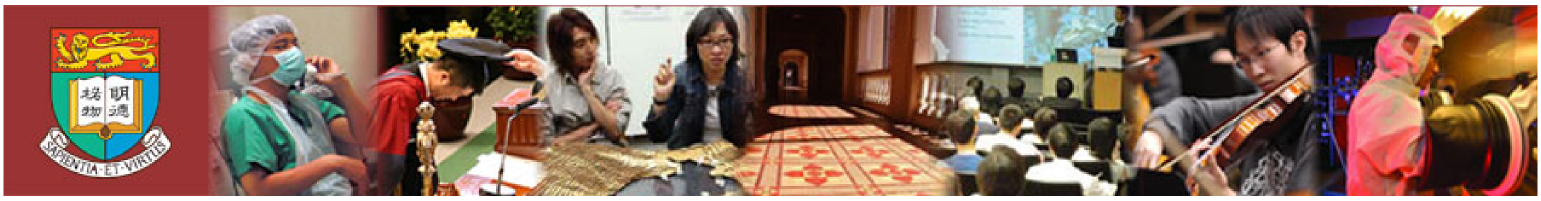


UV-Diagram: A Voronoi Diagram for Uncertain Data

Reynold Cheng #1, Xike Xie #2, Man Lung Yiu *3, Jinchuan Chen +4, Liwen Sun #5

#*Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong*
{1ckcheng,2xkxie,5lwsun}@cs.hku.hk

**Department of Computing, Hong Kong Polytechnic University, Hung Hom, Hong Kong*
3csmlyiu@comp.polyu.edu.hk

+ *Key Lab for Data Engineering and Knowledge Engineering, MOE. Renmin University of China, P.R.China*
4jcchen@ruc.edu.cn

***Abstract*— The Voronoi diagram is an important technique for answering nearest-neighbor queries for spatial databases. In this paper, we study how the Voronoi diagram can be used on uncertain data, which are inherent in scientific and business applications. In particular, we propose the *Uncertain-Voronoi Diagram* (or *UV-diagram* in short). Conceptually, the data space is divided into distinct "UV-partitions", where each UV-partition P is associated with a set S of objects; any point q located in P has the set S as its nearest neighbor with non-zero probabilities. The UV-diagram facilitates queries that inquire objects for having non-zero chances of being the nearest neighbor of a given query point. It also allows analysis of nearest neighbor information, e.g., finding out how many objects are the nearest neighbors in a given area.**

However, a UV-diagram requires exponential construction and storage costs. To tackle these problems, we devise an alternative representation for UV-partitions, and develop an adaptive index for the UV-diagram. This index can be constructed in polynomial time. We examine how it can be extended to support other related queries. We also perform extensive experiments to validate the effectiveness of our approach.

I. Introduction

The Voronoi Diagram, primarily designed for evaluating nearest-neighbor queries over two-dimensional spatial points [1], has raised plenty of research interest. This technique has been extended to handle different related problems, including database services in wireless broadcast environments [2], [3]; high-dimensional query evaluation [4]; continuous location-based services [5]–[7]; and virus spread analysis among mobile devices [8]. Conceptually, the Voronoi diagram partitions the data space into disjoint "Voronoi cells", so that all points in the same Voronoi cell have the same nearest neighbor. The task of finding the nearest neighbor of a query point is then reduced to a point query. Figure 1(a) illustrates a Voronoi diagram of seven points. Since the query point q is located in the Voronoi cell of O_2, O_2 is the nearest neighbor of q.

Is it possible to use the Voronoi diagram to perform nearest-neighbor search on objects whose values are imprecise? Data values can be uncertain for a variety of reasons. Consider a satellite image, which depicts geographical objects like airports, vehicles, and people. Using machine learning and human effort (e.g., community-based systems like Wikimapia), the location of each object on the image can be obtained. Due to the noisy transmission of satellite data, the quality of these images can be affected, and we may not be able to obtain very accurate locations. Moreover, if this location information is released to the public (e.g, for research purposes), it may need to be preprocessed for privacy purposes. In fact, recent proposals like [9], [10] have suggested to represent a user's position as a larger region, in order to lower the likelihood that a user is identified at a particular site. Uncertainty is also inherent in biological data management. For example, microscopy images have been actively used to analyze the thickness of neuron layers in the retina, as well as the extent of the area of a cell. Due to factors like image resolution and measurement accuracy, it is hard to obtain exact values of the objects of interest [11], [12]. For this kind of data, techniques for evaluating range queries, nearest-neighbor queries, and joins, have been developed. These queries return answers with probabilistic guarantees, which reflect the confidence of answers due to data uncertainty. For these applications, tools that resemble the Voronoi diagram can be potentially useful. Specifically, we would like to examine space-partitioning techniques for performing a *Probabilistic Nearest-Neighbor Query* (PNN). Given a query point q, a PNN returns the IDs of objects with non-zero probabilities for being the closest to q, as well as their probabilities. In the sequel, we denote the objects returned by the PNN as *answer objects*, and their probability values as *qualification probabilities*.

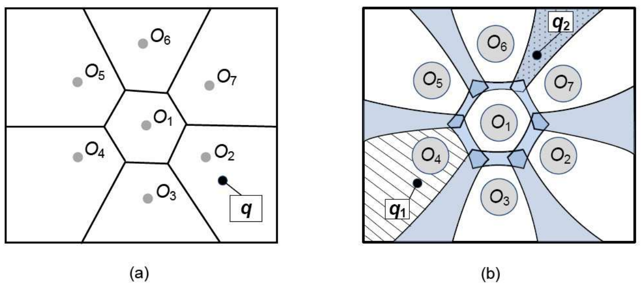


Fig. 1. (a)Voronoi Diagram. (b) UV-Diagram.

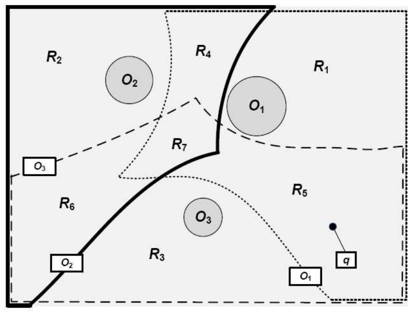
An uncertainty model that has been commonly used is to assume that an object O_i has an "uncertainty region" and a probability distribution function (pdf). This means that the precise position of O_i can only be located inside the (closed) region, with a pdf that describes the distribution of the object's position within the region. The uncertainty region can have any shape, and the pdf is arbitrary (e.g., it can be a uniform distribution, Gaussian, or a histogram). Here we assume that O_i has a two-dimensional circular uncertainty region. However, our solution can be extended to handle non-circular-shaped regions.

Our goals are to investigate how such a diagram should be defined to support nearest-neighbor query execution. Specifically, we propose the *Uncertain-Voronoi* diagram (or *UV-diagram*), where the nearest-neighbor information of every point in the data space is recorded, based on the uncertain objects involved. The UV-diagram provides a basis for studying solutions that used the Voronoi diagram for point data. It could be interesting, for instance, to extend the solution of [2] to support uncertain data in broadcasting services. Figure 1(b) illustrates an example of the UV-diagram of seven uncertain objects, where the space is divided into disjoint regions called *UV-partitions*. Each UV-partition P is associated with a set S of one or more objects. For any point q located inside P, S is the set of answer objects of q (i.e., each object in S has a non-zero probability for being the nearest neighbor of q). The highlighted regions contain points that have two or more nearest neighbor objects. As an example, since q_1 is inside the dashed region, O_4 has a non-zero probability for being the nearest neighbor of q_1; on the other hand, q_2 is located inside the dotted region, and O_6 and O_7 are the answer objects for the PNN with q_2 as the query point. Observe that the Voronoi diagram, which indexes on spatial points, is a special case of the UV-diagram, since a point can be viewed as an uncertainty region with a zero radius. Figure 1 compares the two diagrams.

Besides answering nearest-neighbor queries, the Voronoi diagram is useful for doing data analysis or observing interesting patterns of nearest-neighbor information. In [8], for example, the Voronoi diagram is used to investigate the spreading pattern of bluetooth viruses among mobile users. A UV-diagram can also provide valuable information about these "nearest-neighbor patterns". For instance, in Figure 1(b), if the dashed region is large, then O_4 has high chance to be placed in different clusters, assuming a nearest-neighbor clustering algorithm is used. Another interesting query is: given a region R, display all UV-partitions that intersect with R, as well as the density of objects that can be the nearest neighbor in each UV-partition. Through the UV-diagram, a user can visualize or extract patterns about the nearest-neighbor information.

Drawback of existing solutions. As far as we know, the only indexing method available for nearest-neighbor search over uncertain data is to use an index like the R-tree and the grid. R-tree is a disk-based structure that uses the Minimum-Bounding Rectangles (MBRs in short) to cluster the uncertainty regions of the objects, and organizes MBRs in a hierarchical manner [13]. To evaluate PNN using the R-tree, a branch-and-prune strategy has been proposed in [14], where MBRs that may contain answer objects are traversed. However, this involves a lot of overhead in reading index nodes and leaf pages [14], [15]. Similar issues also occur with grids [16].However, retrieving answer objects from the UV-diagram is essentially a point query search: given a point q, find the objects associated with the UV-partition that contains q. Hence, a UV-diagram can support more efficient PNN search. It is also not clear how an R-tree or grid over uncertain objects can provide pattern analysis of nearest-neighbor information (e.g., displaying the extent of a UV-partition).

Challenges of constructing UV-diagram. It is not trivial to generate a UV-diagram, since this involves producing space partitions based on uncertainty regions, which may not be points. Unfortunately, efficient computational geometry methods for generating the Voronoi diagram (e.g., line-sweeping [17]) cannot be readily used for creating a UV-diagram, since these methods are primarily designed for spatial points, rather than uncertainty regions. Figure 2 depicts the space partition based on three uncertainty regions represented as circles. Each UV-partition (named R_i, where $i = 1, \dots, 7$) is irregular in shape and contains different answer objects, listed on the side of the figure. In general, given a set of uncertain regions, an exponential number of UV-partitions can be created. For example, Figure 2 shows that for three objects, there are seven UV-partitions, each of which contains one of $2^3 - 1 = 7$ combinations of the three objects. To make the problem worse, the number of edges of each UV-partitioncan also be exponentially large! This makes it computationally infeasible to generate and store these partitions. It is also difficult to find out which of these irregular UV-partitions contain a given query point. Indeed, our experimental results show that a brute-force approach of computing and indexing UV-partitions over 50k objects require about 97 hours. Therefore, a scalable method for constructing a UV-diagram is highly desirable.



Partition	Answer Objects
R_1	O_1
R_2	O_2
R_3	O_3
R_4	O_1, O_2
R_5	O_1, O_3
R_6	O_2, O_3
R_7	O_1, O_2, O_3

Fig. 2. A UV-Diagram for 3 uncertain objects.

Our solution. In order to avoid computing UV-partitions directly, we have developed an alternative representation of UV-partitions. Particularly, we propose the novel concept of the *UV-cell*. A UV-cell of an uncertain object O_i is essentially a region, such that a query point inside O_i's UV-cell has O_i as an answer object. Figure 2 illustrates the UV-cells for O_1, O_2, and O_3. The boundary of each UV-cell is labeled with the ID of the object. For example, the UV-cell of O_2 is a region enclosed by solid-line segments. The intersection of one or more UV-cells constitutes a UV-partition. For instance, the UV-cells of both O_1 and O_3 intersect at partitions R_5 and R_7. This means when q is located at any of these partitions, both O_1 and O_3 are the answer objects. Notice that R_7 is intersected by O_2's UV-cell, and hence O_2 is also associated with R_7. Hence, a UV-diagram can be considered as the union of all objects' UV-cells. By finding the UV-cells that contain q, objects with non-zero probabilities can be retrieved.

Although a UV-cell is still expensive to compute, we show how to represent a UV-cell as a set of "candidate reference objects", or *cr-objects* in short. Conceptually, cr-objects are those that define the shape of a UV-cell. These objects can be efficiently obtained. More importantly, by using cr-objects, we devise a polynomial-time method for constructing an index for the UV-partitions. We have adopted an adaptive-grid indexing scheme, which has the advantage of adapting to different distributions of uncertain objects' positions. We will give detail about how this index can be created. Our experimental results show that for both synthetic and real dataset, this index can be constructed in a much shorter time. We also demonstrate how to use this index to support PNN and nearest-neighbor pattern queries.

The rest of the paper is as follows. Section II summarizes related work. In Section III we present basic concepts of the UV-diagram. We explain how to represent UV-cell efficiently in Section IV, and discuss an adaptive index based on the UV-diagram in Section V. We present experimental results in Section VI. Section VII concludes the paper.

II. Related Work

Data Uncertainty Management. Recently, researchers have proposed to consider uncertainty as a "first-class citizen" in a DBMS [15], [18]–[20]. Two models can be used to represent uncertain data: tuple- and attribute- uncertainty. For tuple-uncertainty, each database tuple has a probability of being correct [20]. Here we assume attribute-uncertainty, which represents an attribute as a range of possible values and a probability distribution function (pdf) bounded in the range [18]. Common queries for attribute uncertainty include range queries [21], k-nearest-neighbors [11], skylines [22], [23] and top-k queries [24].

A few works have been proposed to evaluate PNN queries over attribute uncertainty. In [14], numerical integration techniques have been presented. *Probabilistic verifiers*, described in [15], can generate answer objects' probability bounds without performing expensive integration operations. Another way to compute answer probabilities is based on sampling [25]. Here we focus on the efficient retrieval of answer objects. An R-tree-based solution has been proposed in [14], which uses a branch-and-prune strategy to look for nearest neighbors. This solution can involve multiple traversals over the R-tree, resulting in a high I/O cost. With the use of the UV-diagram, we show how answer objects can be retrieved more efficiently.

Other types of nearest-neighbor queries, like the "group nearest-neighbors" [26], "reverse-nearest-neighbors" [27], [28], and "uncertain queries" [29], have also been proposed. In these works, the R-tree was used to support object retreival. An interesting direction is to study how to use the UV-diagram in these solutions.

The Voronoi diagram is an important technique for answering nearest-neighbor queries over spatial points [1]. It has been extended to support other applications (e.g., [2]–[6]). It also facilitates the analysis of spreading patterns of mobile viruses [8]. In [30], the k-th order Voronoi diagram is used to evaluate a k-NN query. The Voronoi diagram has also been defined for boundaries of circular objects in [31]. However, these objects are *not* uncertain, and the method of [31] cannot be used to answer PNN queries.

Few works have studied the application of the Voronoi diagram on uncertain data. [29] consider the "uncertain" nearest neighbor query (UNN) over spatial points. Different from PNN, the query is an uncertain region, not a query point. To evaluate a UNN, the authors propose to use a Voronoi diagram over 2D points. The portions of the Voronoi cells that overlap with the query's uncertainty region are then used to compute answer probabilities. [32] consider the clustering of uncertain attribute data, where a Voronoi diagram is constructed for centroid points. Notice that [29] and [32] do not construct a Voronoi diagram for uncertain data. On the other hand, the UV-diagram is a Voronoi diagram tailored for attribute uncertainty.

In [33], [34], the Voronoi diagram was modified to identify an imprecise object which is surely the nearest object of a query point q. However, the UV-diagram returns object(s) that *may* have chance to be the nearest neighbor of q, and can be used to answer probabilistic nearest-neighbor queries. We also study a database index for the UV-diagram, which has not been examined in these two works.

III. The UV-Diagram

As mentioned in Section I, we can use a "UV-cell" to derive a UV-diagram. Section III-A presents the definition of a UV-cell. We then study a simple method for constructing a UV-cell in Section III-B. The mathematical formulation of a UV-cell is described in Section III-C.

A. The UV-cell

As discussed before, a UV-cell of an object is essentially a region where the object has non-zero chance to be the nearest neighbor of any query point located inside it. Formally, let $O_1, O_2, \ldots, O_n$ be the IDs of a set O of uncertain objects, and D be a two-dimensional space that contains these objects. Notice that D can have any shape in general; for the sake of discussions, we assume that D is a square.

Definition 1: A **UV-cell** of O_i, denoted by U_i, is a region in D such that O_i has a non-zero probability to be the nearest neighbor (NN) of a point q iff q is located in U_i.

Hence, O_i cannot be q's nearest neighbor if q is outside U_i. The UV-cell can be used to recover the UV-partitions (i.e., disjoint regions of a UV-diagram). In fact, a UV-partition that contains q is the intersection of all UV-cells that contain q. This is because the objects associated with these UV-cells have non-zero qualification probabilities. Thus, given the UV-cells of all objects, we can use them to find out which object(s) is/are the nearest neighbor of q with non-zero probabilities.

Notice that if there is at least one uncertain object in domain D, any point in D must be covered by at least one UV-cell. In particular, if O_i is the only object in domain D, then its UV-cell is exactly D.

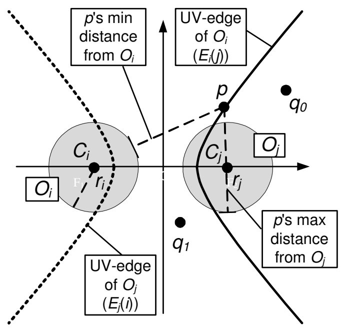


Fig. 3. The UV-edge.

We now study the relationship between a query point and UV-cells. Let p be a point in D, and let $dist_{min}(O_i, p)$ and $dist_{max}(O_i, p)$ be the minimum and the maximum distances of object O_i from p respectively. Figure 3 illustrates two uncertain objects, O_i and O_j. For any point p on the solid line shown, we require the following property to hold:

$$dist_{min}(O_i, p) = dist_{max}(O_j, p) \quad (1)$$

We call this solid line the "UV-edge of O_i with respect to O_j", denoted by $E_i(j)$. A special property of this edge is that any point p at the region on the side of $E_i(j)$ closer to O_j has its maximum distance from O_j, i.e., $dist_{max}(O_j, p)$, shorter than its minimum distance from O_i, i.e., $dist_{min}(O_i, p)$. On the other hand, if p is on the opposite side of $E_i(j)$, then $dist_{max}(O_j, p) \geq dist_{min}(O_i, p)$.

The UV-edge allows us to decide whether an object is an *answer object* (i.e., an object with non-zero qualification probabilities). In Figure 3, q_0 is on the right of $E_i(j)$, which is also closer to O_j than O_i. Thus, $dist_{max}(O_j, q_0) < dist_{min}(O_i, q_0)$. In other words, O_j is *always* closer to q_0 than O_i, and O_i has no chance to be the nearest neighbor of q_0. As another example, q_1 is on the left of $E_i(j)$. Since $dist_{min}(O_i, q_1) \leq dist_{max}(O_j, q_1)$, O_i has a non-zero qualification probability. Hence, given $E_i(j)$, if the query point is on the right of $E_i(j)$, O_i can be pruned.

B. Constructing a UV-cell

We now present a simple method of constructing a UV-cell. Let us define the following:

Definition 2: A **possible region** of object O_i, denoted by P_i, is an area that completely covers the UV-cell of O_i.

An example of an object's possible region is the domain D, since D must cover any UV-cell.

Definition 3: The **outside region** of UV-edge $E_i(j)$, denoted by $X_i(j)$, is the region on one side of $E_i(j)$ such that for any point $q \in X_i(j)$, O_j is always closer to q than O_i.

In Figure 3, the outside region of the UV-edge $E_i(j)$ is the area on the right of the solid line. Notice that since q_0 is in the outside region of $E_i(j)$, O_j is closer to q_0 than O_i, and thus O_i cannot be q_0's nearest neighbor.

Algorithm 1 UV-cell Generation

```
    Input: Uncertain objects $O = \{O_1, O_2, \ldots, O_n\}$
    Output: $U_1, U_2, \ldots, U_n$
 1: for each $O_i \in O$ do
 2:     Let $P_i \leftarrow D$;
 3:     for each $O_j \in O \wedge j \neq i$ do
 4:         $E_i(j) \leftarrow$ UV-edge of $O_i$ w.r.t. $O_j$;
 5:         $X_i(j) \leftarrow$ outside region of $E_i(j)$;
 6:         $P_i \leftarrow P_i - X_i(j)$;
 7:     end for
 8:     $U_i \leftarrow P_i$;
 9: end for
10: return $U_1, U_2, \ldots, U_n$
```

Given an object O_i, if we know all the outside regions $X_i(j)$ (where $j = 1, \ldots, n \wedge j \neq i$), then O_i's UV-cell can be constructed by excluding all these regions from D. Algorithm 1 illustrates the basic method for constructing UV-cell for n objects. The possible region of each object O_i is first initialized as the whole space (Step 2). Then, for each O_j, we compute the UV-edge of O_i and its corresponding outside region (Steps 4 and 5). The possible region, which contains all the points that may have O_i as one of their nearest neighbors, is then "reduced" by the outside region that overlaps with it (Step 6). The UV-cell of O_i is then assigned to be the final possible region (Step 8).

The order of selecting the object for refining O_i's possible region (Steps 4-6) does not affect the correctness of the algorithm. This is because the UV-cell is produced by "shrinking" the possible regions by using the outside regions of other objects. Moreover, as we will see, not all objects are useful in shaping the UV-cell. Once all the UV-cells are generated, then they can be used to answer PNN queries. Table I shows the symbols used in this paper.

TABLE I
NOTATIONS AND MEANINGS.

Notation	Meaning
	Objects and query
D	Domain space (a square)
O	A set of uncertain objects $(O_1, O_2, \ldots, O_n)$
(c_i, r_i)	Center and radius of O_i
q	Query point of a PNN
	UV-diagram
$Cir(c, r)$	A circle centered at c with radius r
$dist(q, c_i)$	Euclidean distance between q and c_i
$dist_{min}(q, O_i)$	min. distance of O_i from q
$dist_{max}(q, O_i)$	max. distance of O_i from q
U_i	UV-cell of O_i
P_i	Possible region of O_i
$E_i(j)$	UV-edge of O_i w.r.t. O_j
$X_i(j)$	Outside region of O_i w.r.t. O_j
F_i	r-objects of O_i, where $F_i \subseteq O$
C_i	cr-objects of O_i, where $C_i \subseteq O$
M	max. no. of non-leaf nodes
T_θ	split threshold

C. The Shape of a UV-cell

Let us assume that the uncertainty region of O_i is a circle, with center c_i and radius r_i. (Later we discuss how other shapes can be supported.) We only present the general case ($r_i > 0$); the special case (i.e., $r_i = 0$) is discussed in our report [35]. For any point $d \in D$, we observe from Figure 3 that:

$$dist_{min}(O_i, q) = \begin{cases} dist(q, c_i) - r_i & q \notin Cir(c_i, r_i) \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

$$dist_{max}(O_j, q) = dist(q, c_j) + r_j \quad (3)$$

where $Cir(c_i, r_i)$ denotes a circle with center c_i with radius r_i. Since $r_i > 0$, $dist_{max}(O_j, q)$ must also be positive. Thus, by substituting Equations 2 and 3 into Equation 1, we have:

$$dist(q, c_i) - dist(q, c_j) = r_i + r_j \quad (4)$$

Let the coordinates of c_i and c_j be (x_i, y_i) and (x_j, y_j). Let $f_x = \frac{1}{2}(x_i + x_j)$ and $f_y = \frac{1}{2}(y_i + y_j)$. Let $cos\theta = \frac{(x_j - x_i)}{dist(c_i, c_j)}$ and $sin\theta = \frac{(y_j - y_i)}{dist(c_i, c_j)}$. Then, Equation 4 becomes:

$$\frac{x_\theta^2}{a^2} - \frac{y_\theta^2}{b^2} = 1 \quad (5)$$

where

- $a = \frac{r_i + r_j}{2}$, $c = \frac{dist(c_i, c_j)}{2}$, and $b = \sqrt{c^2 - a^2}$;
- $x_\theta = (x - f_x)\cos\theta + (y - f_y)\sin\theta$;
- $y_\theta = (f_x - x)\sin\theta + (y - f_y)\cos\theta$.

Essentially, Equation 5 is a hyperbolic equation, with c_i and c_j as the foci, rotated by θ in an anti-clockwise sense [36]. Figure 3 illustrates that the UV-edge of O_i w.r.t. O_j (the solid line) is a hyperbola.

Equation 5 shows that a UV-cell is composed of the intersections of one or more UV-edges, which are hyperbolas. Since a hyperbola is a conic curve, an UV-edge must be *concave* in shape. In Figure 2, apart from the edges of the domain space, the UV-cells of the three objects have concave edges. Note that Equation 5 has two curves, which represent the UV-edges for each pair of objects involved. For example, in Figure 3, the solid line is the UV-edge of O_i w.r.t. O_j, and the dotted line is the UV-edge of O_j w.r.t. O_i.

If two objects overlap, then $dist(c_i, c_j) < r_i + r_j$, and in Equation 5, b is not real. Physically, this means $E_i(j)$ cannot be found, and we can treat $X_i(j)$ as a zero-area region.

Let us revisit Algorithm 1. Step 4 is done using Equation 5. Step 5 is performed by observing that the outside region of a UV-edge must be convex in shape. To perform Step 6 (i.e., cutting the possible region by an outside region), we compute the intersections of hyperbola equations by using linear algebra techniques [36], which are detailed in our report [35].

Non-circular uncertainty regions. Algorithm 1 can be extended to support non-circular uncertainty regions. In particular, we convert the (non-circular) uncertainty region to a circle that minimally contains it. With a larger (circular) uncertainty region, the object has more chance to be the nearest neighbor of any given point, thereby increasing the UV-cell size. Then Algorithm 1 can be used to construct an approximate UV-diagram for these uncertainty regions. A correctness proof of this conversion can be found in [35].

Complexity. The problem of Algorithm 1 is that it is very costly. For each object, its UV-edge with respect to other objects is used to refine its possible region (Step 6). This requires computing the intersections of all edges of the current possible region (P_i) with a new UV-edge $E_i(j)$ from O_j. Note that $E_i(j)$, a hyperbolic curve, can create *three* new edges with each concave edge of P_i. In the worst case, the number of edges of P_i increases by three times whenever a new UV-edge is considered in Step 6. As a result, the number of edges of the UV-cell is $O(3^n)$ (the detailed proof is shown in [35]). Moreover, computing intersections between hyperbolas is complex. In fact, this needs 97 hours to create a UV-diagram of 50K objects in our implementation. Let us investigate how to tackle these problems.

IV. Efficient UV-Cell Generation

Since generating a UV-cell is inefficient, our strategy is to avoid computing it directly. Instead, we represent a UV-cell as a set of *cr-objects*, which can be efficiently derived. Section IV-A outlines the algorithm of yielding cr-objects. We explain the preparation phase of this algorithm in Sections IV-B, and two techniques for finding these objects quickly, in Sections IV-C and IV-D.

A. r-Objects and cr-Objects

Recall from Algorithm 1 that the UV-cell of an object O_i, i.e., U_i, is the result of repeatedly subtracting the outside region of other objects (i.e., $X_i(j)$) from its possible region, P_i. In fact, not all outside regions are useful for refining P_i. In particular, if the UV-edge of O_i corresponding to O_j, i.e., $E_i(j)$, does not intersect with P_i, then P_i cannot be shrinked by X_i(j). We call an object O_j a *reference object* (or *r-object*) of O_i, if O_j defines an edge of O_i's UV-cell. We also denote $F_i \subseteq O$ to be the set of r-objects of O_i. The set F_i contains objects whose outside regions are responsible for defining the UV-cell of O_i. In Figure 2, for example, the set of r-objects of O_3, i.e., F_3, is to $\{O_1, O_2\}$.

Given that the r-objects for each object is known, our solution (to be shown in Section V) can use r-objects to develop an alternative representation of the UV-diagram. This solution is much cheaper than Algorithm 1, which requires exact UV-cells to be computed. However, finding F_i itself is difficult, because we do not know the UV-cell of O_i. Our strategy is to find a small set C_i of objects, where $F_i \subseteq C_i$. We call C_i the *candidate reference objects* (or *cr-objects* in short). We next show how C_i can be derived without acquiring the exact UV-cell of O_i. In Section V, we study an indexing solution based on cr-objects.

Algorithm 2 outlines the three steps required for deriving the cr-objects for O_i. Step 1 (`initPossibleRegion`) creates a possible region P_i based on a small number of objects. In Step 2, the "index level" pruning (or `indexPrune`) yields a set I of objects that may contribute edges to the UV-cell. Step

3 applies "computational level" pruning (or `compPrune`) on I, and produces C_i. Here we assume that an R-tree index has been built on the uncertain objects' uncertainty regions. Each object's information (e.g., uncertainty region and pdf), is stored in the disk.

Algorithm 2 Deriving cr-objects

```
    Input: Uncertain object $O_i$
    Output: cr-object $C_i$
 1: $P_i \leftarrow$ initPossibleRegion($O_i$,$O - \{O_i\}$)
 2: ($P_i$,$I$) $\leftarrow$ indexPrune($P_i$,$O$)
 3: $C_i \leftarrow$ compPrune($P_i$,$I$)
```

B. Step 1: Generating a Possible Region

In Step 1 of Algorithm 2), we retrieve a small number of objects, called *seeds*, from the set $O - \{O_i\}$. These seeds are used to generate an "initial" possible region, using a routine similar to Steps 3 to 7 of Algorithm 1. This region is used by other pruning methods to produce cr-objects.

Seeds have to be selected with care. If seeds are randomly selected, a big initial region can be produced. This region may be intersected by many outside regions, resulting in poor pruning efficiency. To produce small regions, we issue a k-Nearest-Neighbor Query (k-NN) on the R-tree, using the center c_i of O_i's uncertainty region as the query point. The k objects, whose uncertainty regions' minimum distances from c_i are the shortest, are obtained. We then select k_s out of k objects to be the seeds. This is done by dividing the domain D into k_s sectors centered at c_i. For each partition, the object closest to c_i is assigned as a seed.

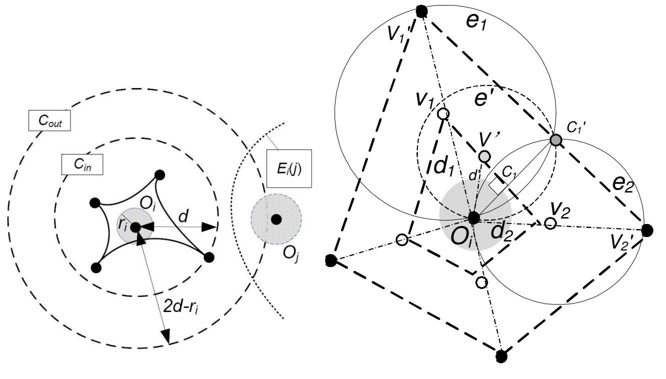
The above method does not guarantee that all k_s seeds can be found (e.g., no seeds can be found if a sector is empty). Even if this happens, however, we can still obtain an initial possible region without affecting the latter steps. This region may be larger though. In our experiments, $k_s = 8$, and in most cases all seeds can be found. For each object, evaluating a k-NN query requires $O(n)$ times, selecting seeds costs $O(k)$ times, and constructing an initial region needs $O(1)$ times. Hence, the cost of this step is $O(n + k)$.

C. Step 2: Index Level Pruning

Once the possible region has been initialized, we perform *I-pruning* (Step 2 of Algorithm 2), in order to remove objects that cannot constitute an UV-edge to the UV-cell. To understand this step, let us consider an object O_i, its possible region P_i, and another object O_j, which has not yet been considered in refining P_i. Our goal is to establish the necessary and sufficient condition(s) for O_j to have effect on the shape of P_i.

Lemma 1: $P_i = P_i - X_i(j)$ if and only if for every point p on the boundary of P_i, $dist_{max}(p, O_j) > dist_{min}(p, O_i)$.

Essentially, if we want to examine whether O_j has any effect on P_i, it suffices to consider the points on P_i's boundary, instead of all points in P_i. Its proof is simple and can be found in our report [35]. The following lemma forms the basis of I-pruning.



(a) I-pruning (b) C-pruning

Fig. 4. Our pruning methods.

Lemma 2: Given an object O_i with center c_i and radius r_i, let d be the maximum distance of P_i from c_i. Let C_{out} be a circle, with center c_i and radius $2d - r_i$. For another object O_j, if $c_j \notin C_{out}$, then $P_i = P_i - X_i(j)$.

Proof: Denote C_{in} be a circle with center c_i and radius d. Figure 4(a) illustrates O_i, its possible region P_i (in solid lines), C_{in} and C_{out}. Let us suppose on the contrary that P_i is not equal to $P_i - X_i(j)$, i.e., P_i can be reshaped by the UV-edge of O_j. Then, using Lemma 1, there must exist a point p on the boundary of P_i such that:

$$dist_{max}(p, O_j) \leq dist_{min}(p, O_i) \tag{6}$$

Using Equations 2 and 3, we have:

$$\begin{aligned} dist(p, c_j) + r_j &\leq dist(p, c_i) - r_i \\ \Rightarrow dist(p, c_j) + dist(p, c_i) + r_j &\leq 2dist(p, c_i) - r_i \\ \Rightarrow dist(p, c_j) + dist(p, c_i) &\leq 2dist(p, c_i) - r_i \\ \Rightarrow dist(c_i, c_j) &\leq 2dist(p, c_i) - r_i \quad (7) \end{aligned}$$

since $dist(c_i, c_j) \leq dist(p, c_j) + dist(p, c_i)$ due to the triangular inequality. Now, $dist(p, c_i) \leq d$, so Equation 7 becomes:

$$dist(c_i, c_j) \leq 2d - r_i \tag{8}$$

This implies that c_j is in the circle C_{out}, contradicting the assumption of Lemma 2. Hence, this lemma is correct. ■

The I-pruning method uses Lemma 2 by issuing a circular range query, centered at c_i with radius $2d - r_i$, on the dataset. This operation can be easily implemented by using the R-tree created for the uncertain objects. The range query first uses the R-tree to filter all objects that do not overlap with the range. For the remaining objects, they are removed if their centers are beyond the circular range. Hence, in this phase, a cost of $O(n)$ is needed for each object.

D. Step 3: Computational Level Pruning

Next, we discuss a simple method, based on distance comparison, for checking whether object O_j can affect the possible region of object O_i. We call this method *C-pruning* (Step 3 of Algorithm 2). Lemma 3, discussed below, serves as the foundation of C-pruning.

Lemma 3: Given an uncertain object $O_i(c_i, r_i)$ and P_i's convex hull $CH(P_i)$, let $v_1, v_2, \ldots, v_n$ be $CH(P_i)$'s vertex. If another object O_j's center c_j is not in any of $\{Cir(v_m, dist(v_m, c_i))\}_{m=1}^{n}$, then $P_i = P_i - X_i(j)$.

Proof: First, the convex hull $CH(P_i)$, which completely contains P_i, must also be O_i's possible region. For every point p on $CH(P_i)$'s boundary, suppose c_j is located outside the circle $Cir(p, dist(p, c_i))$. Then we have:

$$\begin{aligned} dist(p, c_j) &> dist(p, c_i) \\ \Rightarrow dist(p, c_j) + r_j &> dist(p, c_i) - r_i \\ \Rightarrow dist_{max}(p, O_j) &> dist_{min}(p, O_i) \end{aligned} \tag{9}$$

Second, Lemma 1 states that if $dist_{max}(p, O_j) > dist_{min}(p, O_i)$, then $CH(P_i) = CH(P_i) - X_i(j)$. Therefore, if c_j is outside $Cir(p, dist(p, c_i))$ for every p on $CH(P_i)$'s boundary, O_j can be safely pruned.

For convenience, let $Cir(p, dist(p, c_i))$ be a d-bound (where $d = dist(p, c_i)$). We also define a set S of d-bounds for every point p in U_i. We now show that instead of checking all the d-bounds in S, it is only necessary to check those d-bounds constructed for the vertices of $CH(P_i)$. Specifically, the d-bounds of the vertices must contain all other d-bounds of all points on the boundary of $CH(P_i)$. To see this, let d_k be the distance of vertex v_k from O_i's center. We extend each vertex v_k by the distance d_k to obtain a new vertex v'_j (black dot in Figure 4(b)). These new vertices are connected to form a polygon. We use e_1 and e_2 to represent the d-bounds $Cir(v_1, d_1)$ and $Cir(v_2, d_2)$, respectively.

We next show that, for any point v' on $CH(P_i)$'s edge v_1v_2, $Cir(v', dist(v', c_i)) \subseteq e_1 \cup e_2$. (We let $e' = Cir(v', dist(v', c_i))$). We draw a line $c_1c'_1$, which is perpendicular with v_1v_2 and $v'_1v'_2$, and intersects them at points c_1 and c'_1 respectively. As v_1v_2 is the perpendicular bisector of $c_ic'_1$, we see that $c_ic'_1$ is the common chord of e_1, e_2 and e'. Since e_1 or e_2 is bigger than e', e' is contained by e_1 or e_2.

Hence, to check whether O_j can refine P_i, we just need to check the set of d-bounds $S' = \{Cir(v_m, dist(v_m, c_i))\}$ (where $S' \subseteq S$). If c_j is located outside all d-bounds in S', then $CH(P_i) = CH(P_i) - X_i(j)$. Finally, since P_i is completely covered by $CH(P_i)$, $P_i = P_i - X_i(j)$ must also be true. This completes the proof. ■

Step 3 of Algorithm 2 uses Lemma 3 to prune unqualified objects returned by I-pruning. This can be done efficiently, because only the vertices of $CH(P_i)$ are used. Moreover, $|CH(P_i)|$ is small, since the possible region is only derived by eight seeds. The complexity of this phase is $O(n)$.

We consider the objects that are not pruned away in this step as cr-objects (i.e., C_i). The overall complexity of Algorithm 2, for generating C_i's of n objects, is $O(n(n+k))$. Here one may consider to use C_i to generate the exact UV-cell of O_i. However, our experiments showed since $|C_i|$ may be large, generating the UV-cell can still be costly. Next, we show how to use C_i directly to construct an index for the UV-diagram.

V. The UV-Index

We now present a index, called *UV-index*, based on the UV-diagram. Designing the UV-index presents a few technical challenges. The extremely large number of UV-partitions and UV-edges make it infeasible to compute and store a UV-partition. Moreover, the sizes and distributions of the UV-partitions vary significantly (see Figures 1 and 2). Our index solves these problems, and still yields a high query performance. We examine the UV-index and PNN evaluation in Section V-A. We then discuss the construction of the UV-index in Section V-B. We study how to extend the UV-index to support other queries in Section V-C.

A. An Adaptive Grid for UV-partitions

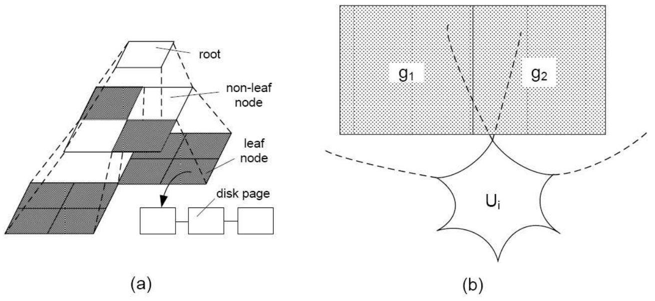


Fig. 5. UV-index: (a) Structure, (b) Overlap checking.

Index Structure. The UV-index adopts a framework similar to a quad-tree [37], in order to index the irregular and non-overlapping UV-partitions. Figure 5 (a) illustrates this index. [1] Each non-leaf node, 16 bytes each, records a pointer to each of its four child nodes, where the square region spanned by each child node is one-fourth of that of its parent. The region covered by the root node is the whole domain D. Each leaf node stores all the objects whose UV-cells overlap with the region defined for the node. To save space, a node's region is not stored, since we can easily derive the dimension of the region based on the level of the node in the tree. Also, due to approximation, a UV-cell that does not overlap with the leaf node's region may be included. However, a UV-cell that truely overlaps with the region will not be excluded. For each leaf node l, we store a linked list of disk pages, which contain tuples $<$ *ID,MBC,pointer* $>$, where:

- *ID* is the identity of object O_i whose UV-cell may overlap with the region covered by l;
- *MBC* is the circle that minimally bounds the uncertainty region of O_i; and
- *pointer* stores the disk page address of the object.

We allocate a maximum of M non-leaf nodes that can be stored in the main memory. The leaf nodes, which contain the lists of pages, are stored in the disk.

[1]Our design adopts quad-tree rather than R-tree. While R-tree MBRs may overlap, quad-tree grids do not. Issuing a point query on non-overlapping UV-partitions in quad-tree is thus more convenient than R-tree.

PNN processing with UV-index. We first use q as the query point, and traverse the index, to find out the leaf l whose region contains q. We then retrieve the disk pages associated with l, which contains the *ID* and the *MBC* of the objects stored in the pages. Since these objects may have their UV cells overlap with the region of l, it is also possible that q is located in their respective UV-cells. Let L be the set of objects associated with l, and A be the answer objects of q. Our goal is to retrieve A from L, where $A \subseteq L$. To do this, we perform a verification method of [14]: based on the *MBC*'s of the objects in L, find out the minimum of the maximum distances of these objects from q. We call this distance d_{minmax}. Any object with the minimum distance larger than d_{minmax} is removed, since this object cannot have a non-zero qualification probability. The remaining objects must be the answer objects, whose probabilities are computed and returned to the user.

B. Index Construction

Recall that a UV-cell can be represented by a set of cr-objects, C_i. Let us examine how this facilitates the construction of the UV-index.

Algorithm 3 InsertObj

```
Input: cr-objects $C_i$; Node $g$;
1:  if (CheckOverlap($C_i$, $g$.region) = true) then
2:      if $g$ is a non-leaf node then
3:          for $k = 1$ to 4 do
4:              InsertObj($C_i, h_k$);
5:          end for
6:      else
7:          $state \leftarrow$ CheckSplit($C_i, g$);
8:          switch ($state$)
9:          case NORMAL:
10:             $g$.list.add($i$, $MBC(O_i)$, $ptr(O_i)$);
11:         break;
12:         case OVERFLOW:
13:             Allocate new page for $g$;
14:             $g$.list.add($i$, $MBC(O_i)$, $ptr(O_i)$);
15:         break;
16:         case SPLIT:
17:             delete $g$.list;
18:             for $k = 1$ to 4 do
19:                 Assign $h_k$ as child of $g$;
20:             end for
21:             $nonleafnum \leftarrow nonleafnum + 1$;
22:         break;
23:     end if
24: end if
```

Framework. Let g be the grid node being examined, and h_k (where $k = 1, \ldots, 4$) be the four child nodes of g. We define a variable *nonleafnum*, which indicates the number of non-leaf nodes allocated to the index and has an initial value of 1. Originally, the `root` of the grid is a leaf node, whose region covered (`root`.*region*) is the domain D.

We use Algorithm 3 (`InsertObj`) to insert an object O_i to the index. This algorithm, whose inputs are C_i and node g, is a recursive procedure, where `InsertObj(`C_i`, root)` is first invoked. In Step 1, `CheckOverlap` investigates if the UV-cell represented by C_i overlaps with the region of grid g. If so, we check whether g is a non-leaf node. If this is true, `InsertObj` is called recursively (Steps 2-4). Otherwise, we perform `CheckSplit` (Step 7), which returns:

1. `NORMAL` (Steps 9-11): g's pages still have space left, and so (i, MBC_i, $ptr(O_i)$) is inserted to g's page, where $ptr(O_i)$ is the pointer to O_i's uncertainty region and pdf.

2. `OVERFLOW` (Steps 12-15): g's pages are full, and a new disk page has to be associated with g, before the information about O_i is inserted to the new page.

3. `SPLIT` (Steps 16-22): g's pages are full. The page list g is removed. Then, g becomes the parent of four nodes (h_k), which have been previously generated by `CheckSplit`. The region of each child node h_k covers each of the four quarters of the region defined for g. Also, *nonleafnum* is incremented by a value of 1. Essentially, The information about the UV-cells previously associated with g are now represented by its child nodes, and g becomes a non-leaf node.

Decision on Splitting. When g's pages are full, either O_i's information is inserted to a new page (`OVERFLOW`), or split into four child nodes (`SPLIT`). Ideally, the region of the leaf node that covers q is completely covered by a true UV-partition. This guarantees that the set of objects returned by the UV-index is the true answer objects. The UV-index, which contains grids, is just an approximation of the UV-diagram. Apparently, the more the splitting is performed, the closer the index can resemble the actual UV-diagram, and yield better query performance.

In fact, splitting is not always useful. Suppose that *g.region* is associated with 100 UV-cells. Moreover, *g.region* is *completely* covered by each of these UV-cells. Then it is not necessary to redistribute g into four child nodes. If splitting is performed in this case, then the UV-cells associated with each child node are exactly the same. Thus, more space is wasted to store duplicated information about the UV-cells. This can happen if the corresponding 100 objects of these UV-cells are close to each other. Then, these UV-cells have similar shapes and significant overlapping. To decide whether to split, we define *split fraction*, θ, as follows:

$$\theta = \frac{\min_{k=1,\ldots,4} |h_k.list|}{|g.list|} \tag{10}$$

which is the minimum fraction of UV-cells in one of the child nodes h_k that are also in g (note that the UV-cells associated with h_k must be the subset of the ones attached to g). A small θ means that the number of UV-cells overlapping with h_k.*region* is small compared with that of g. We now define a splitting condition of a node:

Split if $\theta < T_\theta$

where $T_\theta \in [0, 1]$ is called the *split threshold.* A larger value of T_θ implies a higher tendency of splitting.

Algorithm 4 (`CheckSplit`) implements these ideas. Steps 1-3 return `NORMAL` if the pages of g are not full. Steps 4-5 return `OVERFLOW` if the number of non-leaf nodes allocated is higher than M. In Steps 7-16, we compute the value of θ, by creating four nodes h_k (Step 7), and checking the overlap of each UV-cell with h_k*.region* (Steps 11-12). If the splitting condition is satisfied (Step 17), then the `SPLIT` decision is returned, where Algorithm 3 (Steps 18-19) will assign the nodes h_k to be the child nodes of g. Otherwise, the child nodes are deleted and an `OVERFLOW` decision is made (Steps 20-21).

Algorithm 4 CheckSplit

```
    Input: cr-objects $C_i$; node $g$;
    Outputs: NORMAL, SPLIT, OVERFLOW;
 1: if there is space on any disk page of $g$.list then
 2:     return NORMAL;
 3: end if
 4: if $nonleafnum + 1 > M$ then
 5:     return OVERFLOW;
 6: else
 7:     Create nodes $h_k$ ($k = 1, \dots, 4$) with $h_k$.region equal to each
        quarter of $g$.region;
 8:     Let $A \leftarrow O_i \cup g.list$;
 9:     for each $O_j \in A$ do
10:         for each $h_k$ do
11:             if (CheckOverlap($C_j$, $h_k$.region)) = true then
12:                 $h_k$.list.add($j$, $MBC(O_j)$, $ptr(O_j)$);
13:             end if
14:         end for
15:     end for
16:     Let $\theta \leftarrow (\min_{k=1,\dots,4} |h_k.list|)/|g.list|$;
17:     if $\theta < T_\theta$ then
18:         return SPLIT;
19:     else
20:         delete $h_k$, where $k = 1, \dots, 4$;
21:         return OVERFLOW;
22:     end if
23: end if
```

Algorithm 5 CheckOverlap

```
    Input: cr-objects $C_i$; Region $r$;
    Output: true if $U_i$ and $r$ overlap, false otherwise;
 1: for each $O_k \in C_i$ do
 2:     if $r \subseteq X_i(k)$ then      // Use 4-point testing
 3:         return false;
 4:     end if
 5: end for
 6: return true;
```

Overlap Checking. Algorithm 5 tests if the UV-cell of an object O_i overlaps with a grid g's region r. For every cr-object $O_k \in C_i$, if any of their corresponding outside region ($X_i(k)$) totally contains r, then `CheckOverlap` returns `false` (Steps 1-3). Otherwise, `true` is returned (Step 6). To prove the correctness we use the following lemma:

Lemma 4: If region r is totally covered by $X_i(k)$, where $O_k \in C_i$, then r must not overlap with the UV-cell U_i.

Proof: We want to show that if $\exists O_k$, such that $r \subseteq X_i(k)$, then $r \cap U_i = \phi$. Suppose we have such an object O_k. Now, let us denote $\overline{X_i(j)}$ to be $D - X_i(j)$. Then, U_i is essentially the intersection of all the regions $\overline{X_i(j)}$, for all objects in O, i.e.,

$$U_i = \cap_{j=1 \wedge j \neq i}^{|O|} \overline{X_i(j)} \qquad (11)$$

Moreover, since $r \subseteq X_i(k)$, we have

$$\begin{aligned} r \cap \overline{X_i(k)} &= \phi \\ \Rightarrow (r \cap \overline{X_i(k)}) \cap_{j=1 \wedge j \neq i \wedge j \neq k}^{|O|} \overline{X_i(j)} &= \phi \\ \Rightarrow r \cap (\overline{X_i(k)} \cap_{j=1 \wedge j \neq i \wedge j \neq k}^{|O|} \overline{X_i(j)}) &= \phi \\ \Rightarrow r \cap U_i &= \phi \end{aligned}$$

from Equation 11. Hence, the lemma is correct. ■

To check whether a region r is in the outside region of $X_i(j)$ (Step 2), it is not necessary to generate and test with the UV-edge $E_i(j)$. Instead, we can check this efficiently by using a **4-point test**. To understand this method, observe that r is a square, and the UV-edge of O_i w.r.t. O_j is concave in shape. If all its four corner points are confirmed to be in $X_i(j)$, then we can conclude that $r \subseteq X_i(j)$. For example, Figure 5(b) shows that the region of g_1 must not overlap with U_i, since all the four corner of g are located on the outside region of one of the UV-edges. Moreover, checking whether a point is in $X_i(j)$ is easy, because we can simply check if the point's minimum distance from O_i is larger than its maximum distance from O_j. Hence, we use the four-point test in Step 2.

Notice that Algorithm 5 may incorrectly judge that U_i overlaps with r. Figure 5(b) shows that U_i does not overlap with the region of grid g_2. However, some corners of g_2*.region* are not on the outside region of two of the UV-edges of U_i. If this is true for *all* UV-edges of U_i, then U_i would be decided to be associated with g_2! The consequence is that, during query evaluation, O_i will be retrieved from g_2. This increases the query evaluation time since O_i is not in g_2. However, query accuracy is not affected. In fact, our experimental results show that $|C_i|$ is small with effective pruning, and the scenario in Figure 5(b) is rare. Since checking with C_i is much more efficient than testing with UV-cells, this extra cost is worthwhile. Hence, we use Algorithm 5 to do overlap checking.

Since $|C_i| = O(n)$, Algorithm 5 needs $O(n)$ times to complete. Algorithm 4 uses $O(n^2)$ times, mainly for performing splitting and overlap checking with four child nodes. For Algorithm 3, each UV-cell, in the worst case, needs to perform overlap and split tests with M non-leaf nodes. Hence, its total complexity is $O(Mn^2)$. The index has a maximum height of $M/4$, if, the data distribution is very skewed, and splitting always happen in one single quadrant. However, all non-leaf nodes, 16-byte long, can all be put to the main memory. Thus the tree height has little effect on query performance.

C. Nearest-Neighbor Pattern Analysis

The UV-diagram index can be easily used to retrieve distribution and pattern information about nearest neighbors, which is useful for statistical analysis (e.g., [8]). Let us describe these "pattern-analysis" queries:

1. UV-cell retrieval. This returns the information about O_i's UV-cell (e.g., its area and extent). For example, suppose a user wants to know the approximate area of the region where O_i can be the nearest neighbor. Then, a query that returns the UV-cell U_i of O_i can be useful. To process this query, we scan the leaf nodes that are associated U_i, and compute the total area of the regions covered by these leaf nodes. The process can be sped up by computing and storing these area information offline. A similar procedure can also be used to support the operation of displaying the approximate shape of the UV-cell on the user's screen.

2. UV-partition retrieval. Given a region R, retrieve all UV-partitions inside R, and the approximate "density" of each partition R_i (which is equal to the number of objects associated with R_i, divided by the area of R_i). This allows a user to examine the density distribution of the nearest neighbors in his/her interested area. To support this query, we append a counter to each leaf node, and record the number of objects at that node offline. Then, a range query with range R is issued over the adaptive grid; all regions of the leaf nodes that overlap with R, and their density values, are returned.

VI. Experimental Results

We now report the results on different datasets. Section VI-A describes settings, and Section VI-B discusses the results.

A. Setup

We use Theodoridis et al's data generator [2] to obtain $30k$ objects, which are uniformly distributed in a $10k \times 10k$ space. Each object has a circular uncertainty region with a diameter of 40 units, and a Gaussian uncertainty pdf. For each uncertainty pdf, its mean is the center of the circle, and its variance is the square of one sixth of the uncertainty region's diameter. We represent an uncertainty pdf as 20 histogram bars, where a histogram bar records the probability that the object is in that area. We also use three real datasets of geographical objects in Germany[3], namely *utility*, *roads*, and *rrlines*, with respective sizes 17K, 30K, 36K. These objects are represented as circles before indexing, and has the same uncertainty pdf information as that of the synthetic data.

To compare with R-tree, we use a packed R*-tree [38] to index uncertain objects. The R-tree uses $4k$ disk pages, and has a fanout of 100. We keep all its non-leaf nodes in the main memory. For the UV-index, each non-leaf node has four 4-byte pointers to its children. We also set M, the number of non-leaf nodes in the main memory, to be 4000, and T_θ to be 1. In our experiments, the amount of memory occupied by the R-tree is higher than that of the UV-index. The leaf nodes of both indexes, as well as the uncertainty information about the objects, are stored in the disk.

We examine the running time of 50 PNN queries, whose query points are uniformly distributed in the domain. For simplicity, we use the numerical integration method of [14] to implement probability computation of answer objects. If faster methods such as [15] are used, the fraction of time spent on retrieving answer objects from the index will be higher, and thus it would be important to optimize the index (which is the focus of our work). All our programs were implemented in C++ and tested on a Core2 Duo 2.66GHz PC.

[2]http://www.rtreeportal.org/software/SpatialDataGenerator.zip

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

B. Results

1. Sensitivity Testing. We perform a sensitivity test on the value of T_θ (the splitting threshold). Under a wide range of T_θ, the indexes only have a slight difference. For very small values of T_θ (e.g., 0.2), however, the adaptive grid tends not to split, and degrades into long linked lists of pages. In our experiments, we set T_θ to be 1.

2. Query Performance. We compare the PNN performance of the UV-index and the R-tree on uncertain objects. Figure 6(a) shows the query running time (T_c) against synthetic datasets, with sizes from $10K$ to $80K$. The running time of both queries increase, because with a larger dataset, potentially more objects qualify as query answers, which increase the time for index retrieval and probability computation. The UV-diagram outperforms R-tree in all cases. For example, when $|O| = 60K$, the UV-diagram needs about 50% of the time needed by the R-tree.

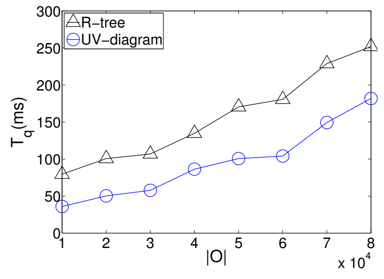
To understand why our method performs better, let us first consider the traversal time of the UV-index, which is composed of the time costs for visiting non-leaf and leaf nodes. Since its non-leaf traversal time takes little time in all experiments (up to 3.9 μs), we only present the I/O overhead. In Figure 6(b) we compare the I/O performance of the UV-index and the R-tree. The UV-index requires significantly less number of I/Os than the R-tree (e.g., when $|O| = 70K$, the UV-index consumes about one-seventh of the I/Os needed by the R-tree). When the R-tree is used to process a PNN query, plenty of leaf nodes needed to be retrieved. For the UV-index, we only need to look for the leaf node that contains the query point. Since the number of disk pages for each leaf node is also small, a high I/O performance can be attained. Also notice that the number of I/Os for the R-tree increases with $|O|$, whereas that of the UV-diagram is relatively stable.

Figure 6(c) shows the time components of T_q: (1) index traversal; (2) retrieval of objects' pdf; and (3) probability computation. While object retrieval and probability computation times are similar for both indexes, R-tree requires a much higher index traversal time. This explains the difference in Figure 6(a). In Figure 6(d) we can see that the query time of both indexes increases with uncertainty region size, since the larger the region, the more probable that the corresponding object is a PNN answer. Again, due to the superiority of I/O performance of the UV-diagram, it performs better than the R-tree.

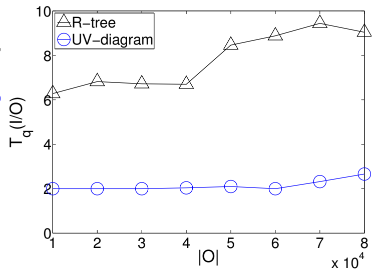
For real datasets, Table II shows that the UV-diagram consistently attains a higher query performance than the R-tree. Since the trends of other results are similar to those of synthetic data, they are omitted here.

TABLE II
Experiment result on real datasets.

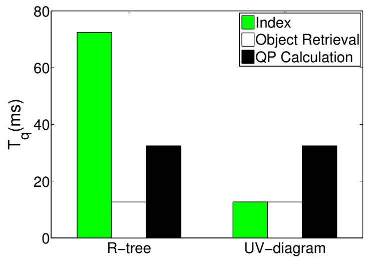
Dataset	$\lvert O\rvert$	T_q(UVD)(ms)	T_q(R-tree)(ms)	T_c(s)	p_c
utility	17K	89	141	784	89%
roads	30K	82	135	2207	88%
rrlines	36K	107	159	2723	86%



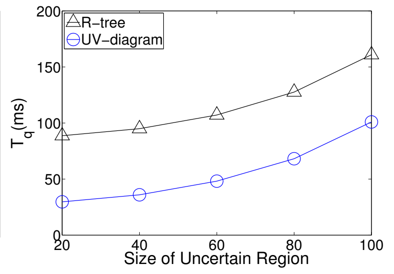
(a) $T_q(ms)$ vs. $|O|$.



(b) $T_q(I/O)$ vs. $|O|$.

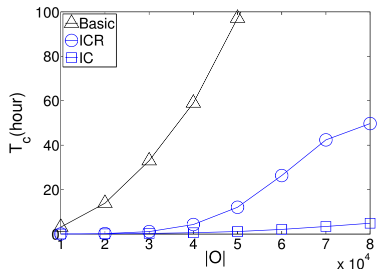


(c)Analysis of T_q.

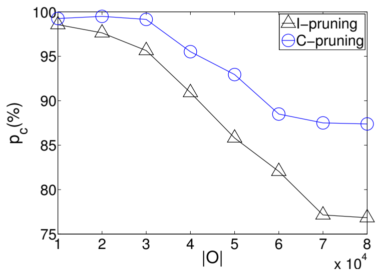


(d) T_q vs. Uncertainty.

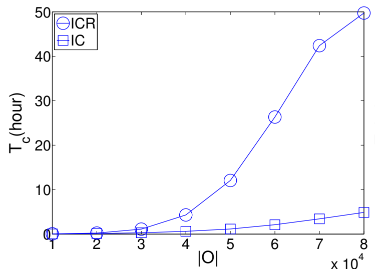
Fig. 6. Query Performance



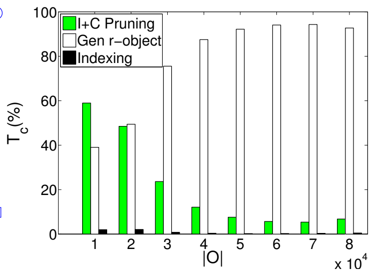
(a) T_c vs. $|O|$.



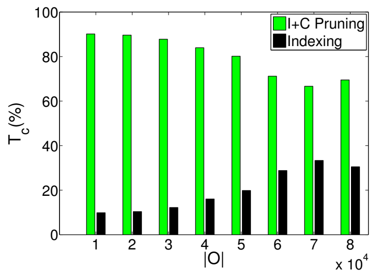
(b)I- vs. C- pruning.



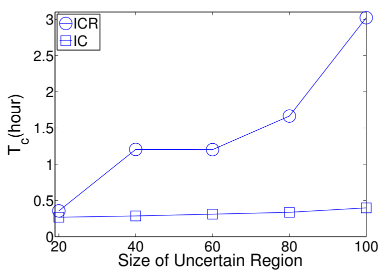
(c)*IC* vs. *ICR*(T_c).



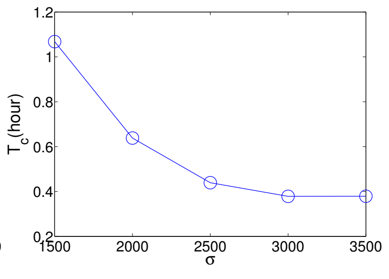
(d) Analysis of *ICR*.



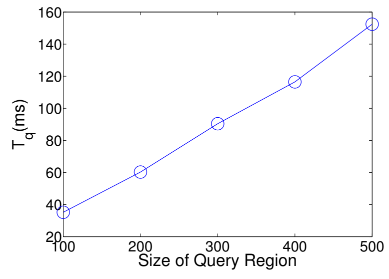
(e)Analysis of *IC*.



(f)T_c vs. uncertainty.



(g)Effect of variance.



(h)UV-partition query.

Fig. 7. UV-Diagram Analysis

3. UV-Diagram Analysis. Next, we examine the UV-diagram construction issues. Let us denote *Basic* as the method which constructs a UV-cell using Algorithm 1, and then indexes the UV-cells with an adaptive grid. An alternative is to collect cr-objects through I-pruning and C-pruning (Algorithm 2), compute UV-cells and obtain the r-objects, and then index them with Algorithm 3. We call this second method *ICR(I- and C-pruning with Refinement)*. The third technique, called *IC*, only uses cr-objects in Algorithm 3. We assume that the R-tree for uncertain objects is available for use by these methods. For generating initial possible regions (used in *IC* and *ICR*), we set k to 300 for performing the k-NN search. Then, the domain D is divided into eight 45^o sectors to obtain the seeds.

Figure 7(a) describes the development time (T_c) of a UV-index for the three methods. *Basic* increases sharply with the dataset size; handling a $50K$ dataset requires about 97 hours. This is because constructing a UV-cell requires an exponential amount of time and numerous complex hyperbola intersections. For *IC* and *ICR*, the use of I- and C-pruning significantly reduces the number of objects examined. Their effects are shown in Figure 7(b), where p_c, the pruning ratio, denotes the fraction of objects from O that has been filtered. At $|O|$=40k, I-pruning and C-pruning achieve a pruning ratio of 90.9% and 95.5% respectively. Hence, a large portion of objects are removed before being considered for constructing the UV-cell. Next, we focus on *IC* and *ICR*.

IC vs. ICR. As shown in Figure 7(c), *IC* performs much better than *ICR*. For example, at $|O| = 70K$, the construction time of *IC* is about 10% of that of *ICR*. To understand why, we analyze their time components in Figures 7(d) and (e). Here we do not show the initial possible region computation time, since it is only about 0.5% of the I- and C-pruning time. Recall the difference between the two methods is that *ICR* needs to find out the exact r-objects (by constructing an exact UV-cell based on the objects returned by pruning), while *IC* does not. For *ICR*, Figure 7(d) shows the fraction of the construction time spent on: (i) I- and C-pruning, (ii) generating r-objects, and (iii) indexing UV-cells. For most datasets, *ICR* spends most of the time to generate exact r-objects, which is very costly. For *IC*, r-object is not produced (Figure 7(e)). Instead, the cr-objects produced by the pruning methods are immediately passed to Algorithm 3 for indexing. Although there are more cr-objects than r-objects, the fact is that the indexing time does not increase much.

In Figure 7(f), the construction time of *ICR* increases sharply with the objects' uncertainty region sizes. With larger uncertainty regions, it is more likely that these regions overlap with each other, making it harder to prune the objects, so that more time is needed to generate r-objects. On the other hand, *IC* is relatively insensitive to the change of uncertainty region sizes. For real datasets, *IC* also achieves high pruning ratio and low construction time (Table II).

We have also measured the query times between the indexes created by IC and ICR. Their performance is almost identical, with a difference of less than 0.01 I/Os. Hence, in other experiments, we assume that IC is used.

Skewness. We next examine the effect of object positions' distribution on the UV-index. Figure 7(g) shows the construction time under different variances (σ) of the uncertainty regions' centers: T_c is higher when data is more skewed (i.e., with a smaller variance). In a dense area where uncertainty regions have high degree of overlap, an object's UV-cell is likely small and associated with many r-objects. Thus T_c is increased. In the most skewed dataset that we tested ($\sigma = 1500$), T_c is around an hour, which is still acceptable if the index is constructed offline.

UV-Partition Query. Finally, we examine the efficiency of our index for answering the UV-partition query. In Figure 7(h), the retrieval time of UV-partitions (T_q) increases with the size of query range R, since more UV-partitions are loaded with larger R. In these experiments, T_q is small.

VII. Conclusions

The UV-diagram is a variant of the Voronoi Diagram designed for uncertain data. To tackle the complexity of constructing and evaluating a UV-diagram, we introduce the concept of UV-cells and cr-objects. We propose an adaptive index for the UV-diagram, and develop efficient algorithms for building it. As our experiments show, this index efficiently supports PNNs and other UV-diagram-related queries.

We plan to extend various Voronoi-diagram-based solutions to handle uncertain data. Also, it would be interesting to study how the UV-diagram can be extended to support multi-dimensional data and incremental updates. Currently, we are investigating the use of the UV-diagram to support other queries (e.g., reverse nearest-neighbor queries).

Acknowledgment

This work was supported by the Research Grants Council of Hong Kong (GRF Projects 5138/06E, 513307, 513508, 711309E). We would like to thank the anonymous reviewers for their insightful comments.

References

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

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

[3] B. Zheng, J. Xu, W.-C. Lee, and L. Lee, "Grid-partition index: a hybrid method for nearest-neighbor queries in wireless location-based services," *VLDB J.*, vol. 15, no. 1, pp. 21–39, 2006.

[4] S. Berchtold, B. Ertl, D. A. Keim, H. peter Kriegel, and T. Seidl, "Fast nearest neighbor search in high-dimensional space," in *ICDE*, 1998.

[5] J. Xu and B. Zheng, "Energy efficient index for querying location-dependent data in mobile broadcast environments," in *ICDE*, 2003.

[6] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik, "The V*-Diagram: a query-dependent approach to moving knn queries," *VLDB*, 2008.

[7] G. Albers et al, "Voronoi diagrams of moving points," *Intl. Journal on Computational Geometry and Applications*, vol. 8, no. 3, 1998.

[8] P. Wang et al, "Understanding the spreading patterns of mobile phone viruses," *Science Express*, vol. 324, no. 5930, 2009.

[9] R. Agrawal and R. Srikant, "Privacy-preserving data mining," in *SIGMOD*, 2000.

[10] C. C. Aggarwal, "On unifying privacy and uncertain data models," in *ICDE*, 2008.

[11] V. Ljosa and A. Singh, "APLA: Indexing arbitrary probability distributions," in *ICDE*, 2007.

[12] ——, "Top-k spatial joins of probabilistic objects," in *ICDE*, 2008.

[13] N. Beckmann et al, "The R*-tree: An efficient and robust access method for points and rectangles," in *SIGMOD*, 1990.

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

[15] R. Cheng, J. Chen, M. Mokbel, and C.-Y. Chow, "Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data," in *ICDE*, 2008.

[16] M. Mokbel, C. Chow, and W. Aref, "The new casper: Query processing for location services without compromising privacy," in *VLDB*, 2006.

[17] M. de Berg et al, *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 1997.

[18] P. Sistla et al, "Querying the uncertain position of moving objects," in *Temporal Databases: Research and Practice*, 1998.

[19] R. Cheng, D. Kalashnikov, and S. Prabhakar, "Evaluating probabilistic queries over imprecise data," in *SIGMOD*, 2003.

[20] N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," in *VLDB*, 2004.

[21] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter, "Efficient indexing methods for probabilistic threshold queries over uncertain data," in *VLDB*, 2004.

[22] J. Pei, B. Jiang, X. Lin, and Y. Yuan, "Probabilistic skylines on uncertain data," in *VLDB*, 2007.

[23] X. Lian and L. Chen, "Monochromatic and bichromatic reverse skyline search over uncertain databases," in *SIGMOD*, 2008.

[24] M. Hua, J. Pei, W. Zhang, and X. Lin, "Ranking queries on uncertain data: A probabilistic threshold approach," in *SIGMOD*, 2008.

[25] H. Kriegel, P. Kunath, and M. Renz, "Probabilistic nearest-neighbor query on uncertain objects," in *DASFAA*, 2007.

[26] X. Lian and L. Chen, "Probabilistic group nearest neighbor queries in uncertain databases," *TKDE*, vol. 20, no. 6, 2008.

[27] M. Cheema et al, "Probabilistic reverse nearest neighbor queries on uncertain data," *TKDE*, vol. 16, no. 9, 2009.

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

[29] G. Beskales, M. Soliman, and I. Ilyas, "Efficient search for the top-k probable nearest neighbors in uncertain databases," in *VLDB*, 2008.

[30] B. Chazelle and H. Edelsbrunner, "An improved algorithm for constructing kth-order voronoi diagrams," *IEEE Trans. Computing*, vol. 36, no. 11, 1987.

[31] M. I. Karavelas, "Voronoi diagrams for moving disks and applications," in *WADS*, 2001.

[32] B. Kao, S. Lee, D. Cheung, W. Ho, and K. Chan, "Clustering uncertain data using voronoi diagrams," in *ICDM*, 2008.

[33] M. Jooyandeh, A. Mohades, and M. Mirzakhah, "Uncertain voronoi diagram," *Inf. Process. Lett.*, vol. 109, no. 13, pp. 709–712, 2009.

[34] J. Sember and W. Evans, "Guaranteed voronoi diagrams of uncertain sites," in *CCCG*, 2008.

[35] R. Cheng, X. Xie, M. Yiu, J. Chen, and L. Sun, "UV-diagram: A voronoi diagram for uncertain data (technical report)," 2009. [Online]. Available: http://www.cs.hku.hk/research/techreps/document/TR-2009-13.pdf

[36] A. Akopyan and A. Zaslavski, *Geometry of Conics*. American Mathematical Society, 2007.

[37] W. Aref and I. Ilyas, "Sp-gist: An extensible database index for supporting space partitioning trees," *JIS*, vol. 17, no. 1, 2001.

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