Crawling Hidden Objects with kNN Queries

Hui Yan, Zhiguo Gong, Member, IEEE, Nan Zhang, Member, IEEE
Tao Huang, Hua Zhong, Jun Wei

Abstract—Many websites offering Location Based Services (LBS) provide a kNN search interface that returns the top-k nearest-neighbor objects (e.g., nearest restaurants) for a given query location. This paper addresses the problem of crawling all objects efficiently from an LBS website, through the public kNN web search interface it provides. Specifically, we develop crawling algorithm for 2D and higher-dimensional spaces, respectively, and demonstrate through theoretical analysis that the overhead of our algorithms can be bounded by a function of the number of dimensions and the number of crawled objects, regardless of the underlying distributions of the objects. We also extend the algorithms to leverage scenarios where certain auxiliary information about the underlying data distribution, e.g., the population density of an area which is often positively correlated with the density of LBS objects, is available. Extensive experiments on real-world datasets demonstrate the superiority of our algorithms over the state-of-the-art competitors in the literature.

Index Terms—Hidden Databases, Data Crawling, Location Based Services, kNN Queries.

1 INTRODUCTION

With rapidly growing popularity, Location Based Services (LBS), e.g., Google Maps, Yahoo Local, WeChat, FourSquare, etc., started offering web-based search features that resemble a kNN query interface. Specifically, for a user-specified query location q, these websites extract from the objects in its backend database the top-k nearest neighbors to q and return these k objects to the user through the web interface. Here k is often a small value like 50 or 100. For example, McDonald’s [1] returns the top 25 nearest restaurants for a user-specified location through its locations search webpage.

While such a kNN search interface is often sufficient for an individual user looking for the nearest shops or restaurants, data analysts and researchers interested in an LBS service often desire a more comprehensive view of its underlying data. For example, an analyst of the fast-food industry may be interested in obtaining a list of all McDonald’s restaurants in the world, so as to analyze their geographic coverage, correlation with income levels reported in Census, etc. Our objective in this paper is to enable the crawling of an LBS database by issuing a small number of queries through its publicly available kNN web search interface, so that afterwards a data analyst can simply treat the crawled data as an offline database and perform whatever analytics operations desired.

Here “crawling” is broadly defined, i.e., it can refer to the extraction of all objects from the database, or only those objects that satisfy certain selection conditions, so long as such conditions can be “passed through” to the kNN interface. For example, if the target here is to crawl Google Maps, then the objective may be to crawl all Vietnamese restaurants in Washington, DC. One can see that this condition can be easily passed through to Google Maps by restricting query locations to be from Washington, DC, and specifying “Vietnamese restaurants” as the search keyword

1

It is important to note that the key technical challenge for crawling through a kNN interface is to minimize the number of queries issued to the LBS service. The requirement is caused by limitations imposed by most LBS services on the number of queries allowed from an IP address or a user account (in case of an API service such as Google Maps) for a given time period (e.g., one day). For example, Twitter limits the search rate at 180 queries per 15 minute. Of course, no algorithm can possibly accomplish the task without issuing at least n/k queries, where n is output size (i.e., the number of crawled objects), because each query returns at most k of the n objects. As such, we are bound to have an output-sensitive algorithm, which nevertheless should have a query cost as close to n/k as possible.

While there have been a few prior studies, specifi-

1. Of course, given the nature of a search engine such as Google Maps, not all objects returned by this keyword query may necessarily be Vietnamese restaurants. In this case, we can simply perform a post-processing filtering to obtain the objects we really want.
cally [2] and [3], on crawling through a kNN interface, both are significantly limited in the following aspects. First, their query costs depend on not the output size or the size of the database, but the distribution of objects in the corresponding spatial space, i.e., the minimal distance from an object to its k-th nearest neighbors in the space. This is an unavoidable artifact of the space-partitioning strategy taken by the two techniques - one using QuadTree while the other using Constrained Delaunay Triangulation. Nonetheless, as we shall show in the experimental results, it may lead to serious efficiency problems while running the algorithms in practice, especially when the space size is large but the desired objects are few and congregated in small clusters. Another problem shared by both existing techniques is that they only work on 2D spaces, but not higher-dimensional spaces that expose a kNN interface.

1.1 Outline of Technical Results
Motivated by the deficiencies of the existing techniques, we develop 2D and higher-dimensional crawling algorithms for kNN interfaces in this paper, with the main contributions summarized as follows:

- We start with addressing the kNN crawling problem in 1-D spaces, and propose a 1-D crawling algorithm with upper bound of the query cost being $O(n/k)$, where $n$ is the number of output objects, and $k$ is the top-k restriction.

- We then use the 1D algorithm as a building block for kNN crawling over 2-D spaces, and present theoretical analysis which shows that the query cost of the algorithm depends only on the number of output objects $n$ but not the data distribution in the spatial space.

- We extend the kNN crawling problem to the general case of m-D spaces - which is the first such solution in the literature.

- Our contributions also include a comprehensive set of experiments on both synthetic and real-world data sets. The results demonstrate the superiority of our algorithms over the existing solutions.

1.2 Paper Organization
The rest of the paper is organized as follows: Preliminaries are introduced in Section 2. We start investigating the problem for 1-D spaces in Section 3. Then we present our crawling techniques for 2-D spaces in Section 4 and extend it to general m-D spaces in Section 5. To improve the crawling algorithm further in 2-D spaces, we introduce our external knowledge based techniques in Section 6. Experimental results are demonstrated in Section 7. Related works and conclusion are produced in Section 8 and Section 9 respectively.

2 PRELIMINARIES

2.1 Data Model
Let $D = \{p_1, p_2, \ldots, p_n\}$ be a spatial database in an m-D (dimensional) space, such that $D \subseteq \mathbb{V}^m = [a_{1,1}, a_{1,r}] \times [a_{2,1}, a_{2,r}] \times \cdots \times [a_{m,1}, a_{m,r}]$, where $\mathbb{V}^m$ is in the m-D space. Each point $p_i$ can be specified as $p_i = (p_{i,1}, \ldots, p_{i,m})$ in the m-D space. In fact, many applications are in a 2-D space, where $\mathbb{V}^m$ reduces to a rectangle $\mathbb{V}^2 = [a_1, a_2] \times [b_1, b_2]$. We use an example to show our previously defined query model for a spatial database with a kNN interface in an m-D space. Now let $\mathbb{V}^m(q)$ denote the m-D sphere with center point $q$ and radius $r$. Then, this sphere is called the covered range of $q$ in the m-D space.

2.2 An Example in 2-D space
We use an example to show our previously defined query model for a spatial database with a kNN search interface. Consider a simple database with 10 points in a 2-D space as shown in Fig. 1. Suppose $k = 2$ in this example. After issuing a query $q$, 2 nearest points are returned. These 2 points are covered by a circle $\mathbb{V}^2(q)$ with center $q$ and radius $r$ which is the distance from $q$ to the farthest returned point. And we also have that every point in the covered region has been crawled.

2.3 Problem Definition
With the model as given in subsection 2.1, we are going to design an algorithm which can efficiently crawl all points from a database $D$ with a kNN interface. We formally define the problem as:

**Problem 1.** Given a spatial database $D$ with a kNN search interface in an m-D space, crawling all points of $D$ while minimizing the number of queries issued.

One brute force solution to Problem 1 is to recursively issue query points randomly on yet uncovered regions of the m-D space until no uncovered regions are left. However, it is prohibitively expensive in practice, because there is no guarantee of the upper bound of this method for there are numerous possible query points in the query space. Thus, we need to
carefully design our crawling algorithms with theoretical guarantees.

2.4 Performance Measures

First, we should make sure that all points have been crawled thoroughly. Second, the performance of crawling algorithms is measured by their efficiency. We count the total number of distinct query points issued for crawling the whole database as the query cost. The reason is that many web-based services enforce Per-IP/user limitations which prevent a user from issuing too many queries. So our goal is to minimize the query cost while crawling all points.

2.5 Table of Notations

The notations used in the paper are shown in Table 1.

3 One Dimensional Analysis

In this section, we develop our crawling algorithm for databases with kNN interfaces in 1-D spaces. Specifically, we start with introducing an example of heavily overlapped queries issued with poor crawling strategies. Then we develop our OPTIMAL-1D-CRAWL algorithm for databases in 1-D spaces which can avoid the above mentioned problem. Finally, we give the theoretical analysis of the proposed algorithm.

3.1 Overlapped query problems

In a 1-D (dimensional) space, let \( D = \{p_1, p_2, \cdots, p_n\} \) be a database with kNN interface, bounded in \( V^1 = [a, b] \). In order to efficiently crawl all points from \( D \) through its \( k \)NN interface, an intuitive idea is to incrementally extend the previously covered spaces by issuing new query points at the boundaries of the covered regions. It is clear that the covered subspace is a sub-interval of \( V^1 \). For easy presentation, we use a circle to indicate the covered range of a query as shown in Fig. 2. One of the problems of this incremental method is that a point may be covered by many queries. Fig. 2 shows a situation where point \( p_1 \) is covered by query circles of \( q_1, q_2 \) and \( q_3 \) (here we suppose \( k = 1 \)). Without any bound of the number of query circles covering a point, it will be hard to determine the bound of the query cost to crawl all points in a database. With this in mind, our objective in this work is to design a crawling algorithm, which can cover all points, yet with a small number of overlapped returning points among those queries.

3.2 The OPTIMAL-1D-CRAWL algorithm

Without knowing how the points are distributed within the database, it is intuitive that each new query point should be issued as far as possible to the covered regions in order to produce less overlapped points by the algorithm. To this end, the first query \( q_1 \) should be issued on the mid-point of \( V^1 = [a, b] \), say \( q_1 = (a + b)/2 \), and the corresponding range covered by \( q_1 \) is denoted as \( V^1(q_1) = [q_1 - r_1, q_1 + r_1] \), which is a sub-interval of \( V^1 \) as shown in Fig. 3, where \( r_1 \) is the radius of query \( q_1 \). Then, the uncovered range of \( V^1 \) becomes two parts \([a, q_1 - r_1]\) and \([q_1 + r_1, b]\)
as shown in Fig. 4. The same idea is applied to these two subspaces recursively until no uncovered subspaces are left. The detail of this OPTIMAL-1D-CRAWL algorithm is presented in Algorithm 1. This algorithm targets the midpoints of uncovered regions while the previously described overlapping algorithm targets the boundaries of uncovered regions - just this subtle difference leads to fundamentally different query complexity results.

Algorithm 1 OPTIMAL-1D-CRAWL Algorithm

```
Input: D: 1-D database; V= [a, b] \supseteq D
Output: all points of D
1: U = \{V\} /*the set of uncovered sub spaces*/
2: P = {} /*D points returned currently*/
3: while (U is not empty) do
4:  get V_i = [a_i, b_i] from U (V_i is an element in U )
5:  issue a query at q_i = (a_i + b_i)/2, it covers a range
6:  V_i(q_i) = [q_i - r_i, q_i + r_i] and return k points
7:  add the returned points in V_i(q_i) to P
8:  if r_i < (b_i - a_i)/2 then
9:      U = U \cup \{[a_i, q_i - r_i], [q_i + r_i, b_i]\} /*add two new uncovered spaces to U*/
10: end if
11: U = U \{V_i\} /*remove V_i from U*/
12: end while
13: return P
```

3.3 The upper bound analysis

Now we want to determine the query cost (or number of queries) needed for totally covering V^1 (crawling all points from D) by the algorithm.

Theorem 1. [1-D Query Cost] Let D be a kNN spatial database which is bounded within 1-D space V^1 = [a, b]. Then, the cost (total number of queries) of OPTIMAL-1D-CRAWL algorithm to crawl all points from D is bounded under O(n/k), where n is the total number of points in D, and k is the top-k restriction.

Proof: In each iteration of OPTIMAL-1D-CRAWL algorithm, we always issue a new query q_i on the mid point of a yet uncovered subinterval V_i^1. This new query q_i can either (1) get k new points and produce two new uncovered subintervals V_{i1}^1 and V_{i2}^1 from V_i^1, or (2) fully cover the whole V_i^1. Because there is in total n points in D and each query returns k points, the first kind of queries can only happen at most \lfloor n/k \rfloor + 1 times, correspondingly the second kind of queries can appear no more than \lfloor n/k \rfloor + 2 times which is just the maximum number of uncovered subintervals produced by the first kind of queries, where \lfloor n/k \rfloor is the integral part of n/k. Therefore, this algorithm can fully crawl the database with the cost under 2\lfloor n/k \rfloor + 3 = O(n/k). \qed

4 TWO DIMENSIONAL ANALYSIS

2-D spatial databases are popularly used in the real world, for which users are often allowed to perform kNN queries with a digital map. Take Yahoo Local as an example, we can search for restaurants by simply clicking a point through a map-like interface. Then, k nearest objects (restaurants) will be delivered by the spatial database for us to browse. We now investigate how to design an effective algorithm for crawling all points (objects) from a 2-D database without knowing their data distribution.

In 2-D space, let D = \{p_1, p_2, \ldots, p_n\} be bounded in a region which is a rectangle V^2 = [a_1, a_2] \times [b_1, b_2]. Recall that in 1-D spaces we try to issue a new query far from those covered regions. Now, such an idea becomes much challenging since the shape of a covered region by several 2-D circles can be much irregular. In order to bound the query cost, an intuitive idea is to reduce the complexity of the 2-D crawling problem by using the 1-D techniques. Because we already have the upper bound for the 1-D crawling algorithm, if we can design our 2-D crawling algorithm by applying a limited number of 1-D techniques, then we can also get the upper bound for the 2-D crawling algorithm. For this sake, a natural way is to partition V^2 into a set of slices (thin rectangles) along one side of the rectangle. Each slice will be much “close” to a 1-D line if it is narrow enough. If we issue 2-D queries along the mid line of a slice, the slice may probably be covered when those queries cover its mid line. Motivated by this discussion, we first introduce how to cover a narrow slice of the rectangle by issuing 2-D queries along a line, then further investigate 2-D crawling techniques using such slices.

4.1 Line covering algorithm using 2-D query circles

Let’s first look at an intuitive idea of covering a 1-D line in 2-D space by directly applying the OPTIMAL-1D-CRAWL algorithm.

Let V^1 denote the mid line of V^2 (rectangle), as shown in Fig. 5. To cover V^1 using 2-D queries, following the principles of OPTIMAL-1D-CRAWL algorithm, now we issue 2-D queries along line V^1 in the binary manner (i.e. always issue the new query in the mid point of an uncovered line segment). The only difference comparing with 1-D situation is that, the original covered range of a query q, denoted as V^2(q), is now a circle in 2-D space and the returned points of q can be anywhere within the circle but may not be located on V^1 as shown in Fig. 5. In fact, we use the intersection of V^2(q) and V^1, denoted as V^1(q), to cover V^1. Now the problem is whether or not the upper-bound is still guaranteed by using those 2-D circles (their intersections on the line) to cover a 1-D line segment.

In fact, when we issue 2-D queries in binary manner on a line segment in a 2-D space, a new query can still return k new points of the database if it has no overlap with its two adjacent queries (even
though those \( k \) points may not located on this line segment now). This intuition immediately leads to the following Proposition.

**Proposition 1. [The Query Bound]** The query cost for covering a line segment \( V^1 \) in the rectangle \( V^2 \) by performing the OPTIMAL-1D-CRAWL algorithm on the line is still under \( O(n) \), where \( n \) is the total number of points of \( D \).

**Proof:** The proof can be found in Appendix A. \( \square \)

In 2-D context, though the line covering algorithm is designed for covering the 1-D line \( V^1 \), in fact, those 2-D queries \( q_1, q_2, \ldots, q_k \) can cover a 2-D slice \( S \) (a small rectangle) in 2-D space \( V^2 \) as shown in Fig. 6a, where \( S \) is the slice bounded by those intersection points among those 2-D circles. With this concept, it is intuitive to study if the whole region \( V^2 \) can be covered by a limited number of such slices. However, it is hard to answer this question since a slice \( S \) obtained in this way may contain no points like the scenario shown in Fig. 6b. To address this problem, we introduce a slight revision of the OPTIMAL-1D-CRAWL algorithm to avoid empty slices being generated.

### 4.2 Line covering algorithm with shrunken 2-D query circles

#### 4.2.1 2-D shrunken circles

In the revised OPTIMAL-1D-CRAWL algorithm, instead of using the original 2-D circles, now we use small circles, which are shrunken from the original 2-D query circles, to cover \( V^1 \). The key idea in this revised algorithm is the differentiation between two kinds of coverage regions - one is still what we had in 1-D - i.e., what our algorithm really guarantees completely crawling of points within. The other type, however, is what is used by the 1-D subroutine to determine where to issue the next query. It is this second type of ‘coverage region’ that is shrunk in this new algorithm, so as to ensure that the slice covered by the first-type coverage regions include at least one point in the database.

In detail, let \( V^2(q) \) be the 2-D circle which returns the \( k \) nearest points \( \{p_1, p_2, \ldots, p_k\} \) for query \( q \). Instead of using \( V^2(q) \), now we use a small circle \( V^2_s(q) \) to cover the line \( V^1 \). To know how \( V^2_s(q) \) is designed, we randomly select one point, say \( p_k \), among those returned \( k \) points. Let \( p_k' \) be the projection of \( p_k \) on \( V^1 \). Then, \( V^2_s(q) \) is defined as the circle with center still \( q \) and a small radius \( ||p_k' - q|| \). Therefore \( V^2_s(q) \) can be regarded as a smaller circle shrunk from \( V^2(q) \), which just covers the projection of point \( p_k \). Take Fig. 7 as an example, suppose a query \( q : (0, 0) \) returns 3 points \( p_1 : (10, 10), p_2 : (-20, -6), p_3 : (-9, 15) \). Then the original covered circle \( V^2(q) \) (dotted circle) is centered at \((0, 0)\) with radius \( 2\sqrt{109} \) which is the distance between \( q \) and the farthest point \( p_2 \). The covered segment by \( V^2(q) \) is the line segment \( V^1(q) : [0] \times [2\sqrt{109}, 2\sqrt{109}] \). Suppose \( p_{3}(\sim -9, 15) \) is randomly selected and its projection \( p_{3}' \) is \( (0, 15) \). Then, the corresponding shrunken circle \( V^2_s(q) \) (solid circle) is centered at \((0, 0)\) with radius \( 15 \). The covered segment by \( V^2_s(q) \) is \( V^1_s(q) : [0] \times [15, 15] \), which is thus used to cover \( V^1 \) instead of using \( V^1(q) \).

Now we construct an inscribed rectangle of the circle \( V^2(q) \), denoted as \( C^2(q) \), which just takes line segment \( V^1_s(q) \) as its mid-line as shown in Fig. 7. Then we can easily prove that \( C^2(q) \) should contain at least one point of \( D \) as presented in Proposition 2.

**Proposition 2. [One Point Guarantee in the Covered Rectangle]** Given a query \( q \) issued on line \( V^1 \) of rectangle \( V^2 \). Let \( V^2(q) \) be the 2-D circle covered by \( q \) in the 2-D space, and \( V^2_s(q) \) be the line segment covered by the shrunken circle \( V^2_s(q) \). Let \( C^2(q) \) be the inscribed rectangle of \( V^2(q) \) which takes \( V^1_s(q) \) as its mid-line. Then, \( C^2(q) \) should contain at least one point of \( D \) point returned by \( V^2(q) \).

**Proof:** Without loss of generality, let \( V^2 = [-a, a] \times [-b, b] \), \( V^1 = [0] \times [-b, b], q = (0, 0) \), and \((x, y)\) be the coordinates of a point in \( V^2 \), and \( p_1, p_2, \ldots, p_k \) be all those \( k \) points returned by circle \( V^2(q) \). Let

---

**Fig. 5.** Cover the 1-D line with the original circles

**Fig. 6.** (a) Example of slice with points; (b) Example of slice without points.

**Fig. 7.** Cover the 1-D line with the result-projections
Let \( p_i = (x_i, y_i) \) be the point whose projection \( p_i' \) is just chosen as the point for constructing \( V_s^x(q) \). According to the definition of \( V_s^x(q), (0, y_i) \), the projection point of \( p_i \) should be on the boundaries of \( V_s^x(q) \). Because \( p_i \) is inside \( V^x(q) \), we easily have that \( p_i \) is covered by \( C^2(q) \) (in fact, \( p_i \) is on one side of \( C^2(q) \)).

**Discussions on the Point Selection:** Recall that both the shrunken circle and the inscribed rectangle are determined by the selected points among those \( k \) returned ones. In fact, we can well chose a point \( p_i \) from the \( k \) points such that the corresponding inscribed rectangle \( C^2(q) \) contains the most of those \( k \) points.

In the later part of the paper, we always regard the shrunken circles \( V^x(q) \) and the corresponding inscribed rectangle \( C^2(q) \) obtained in this way.

Now, with the shrunken 2-D circles to cover a 1-D line, one may wonder if the query cost of the revised crawling algorithm is still bounded under \( O(n) \). As a matter of fact, if \( V^x(q) \) cannot fully cover a segment \( V^1 \), at least one new point of \( D \) should be covered by \( C^2(q) \) according to its definition. Therefore, it is intuitive that the cost of OPTIMAL-1D-CRAWL algorithm for covering a segment with shrunken circles is still bounded under \( O(n) \), where \( n \) is the number of points in \( D \). So, we have:

**Proposition 3. [Upper Bound with Shrunken Circles Covering the Line] The query cost of the OPTIMAL-1D-CRAWL algorithm for covering a segment \( V^1 \) with shrunken circles is \( O(n) \), where \( n \) is the number of points in \( D \).**

More importantly, this improved 1-D optimal crawling algorithm guarantees that the corresponding slice obtained contains at least one point of \( D \).

### 4.2.2 Slice covering algorithm

Now we formally discuss how the slice is obtained. Let \( q_1, q_2, \ldots, q_t \) be the query points issued on \( V^1 \) (\( y = 0 \)), such that their corresponding shrunken circles \( V^x(q_1), V^x(q_2), \ldots, V^x(q_t) \) cover line \( V^1 \) completely. Suppose \( C^2(q_1), C^2(q_2), \ldots, C^2(q_t) \) are the corresponding inscribed rectangles of \( V^x(q_1), V^x(q_2), \ldots, V^x(q_t) \), respectively. Then, each \( C^2(q_i) \) can be represented as:

\[
C^2(q_i) = [-\Delta_i, \Delta_i] \times V^x(q_i)
\]

where \( \Delta_i \) is the half width of rectangle \( C^2(q_i) \) as shown in Fig. 7. Let \( \Delta = \min_{1 \leq i \leq t} \{\Delta_i\} \), then we get

\[
C^2_{\text{min}}(q_i) = [-\Delta, \Delta] \times V^x_1(q_i) \subseteq [-\Delta_i, \Delta_i] \times V^x_1(q_i) = C^2(q_i)
\]

Let \( S = [-\Delta, \Delta] \times V^1 \), thus,

\[
S \subseteq \bigcup_{1 \leq i \leq t} [-\Delta_i, \Delta_i] \times V^1(q_i) \subseteq \bigcup_{1 \leq i \leq t} C^2(q_i) \subseteq \bigcup_{1 \leq i \leq t} V^2(q_i)
\]

which leads to the following corollary.

**Corollary 1. [One Point Guarantee in the Slice] For any line \( V^1 \) in rectangle \( V^2 \), let \( S \) be the slice covered by those original 2-D circles of the queries by performing OPTIMAL-1D-CRAWL algorithm on \( V^1 \) using shrunken circles. Then, \( S \) should contain at least one point of \( D \).**

**Proof:** The proof can be found in Appendix B.

The description of the slice covering algorithm is presented in Algorithm 2.

**Algorithm 2 [Slice Covering Algorithm]**

**Input:** \( V^2 = [a_1, a_2] \times [b_1, b_2] \); \( D \): a database;

**Output:** a covered slice \( S \)

1. \( a_m = (a_1 + a_2)/2 \); \( s \) mid point of \( [a_1, a_2] \)
2. \( V^1 = [a_m] \times [b_1, b_2] \); \( \star \) the mid line of \( V^2 \);
3. \( \Delta = \infty \);
4. \( U = \{V^1\} \);
5. while \((U \) is not empty) do
6. get \( V^1 \) from \( U \);
7. set \( q_i \) as the mid point of \( V^1 \);
8. call \( V^2(q_i) \) which returns \( k \) \( D \) points \( p_1, p_2, \ldots, p_k \);
9. get \( C^2(q_i) \) = \( [a_m - \Delta, a_m + \Delta] \times [b_1, b_2] \); \( \star \) the 2-D crawling algorithm for crawling \( V^1 \) using slices \( V^2(q_i) \) that contains the most of \( \{p_1, p_2, \ldots, p_k\} \);
10. get \( V^2(q_i) \); the shrunken circle of \( V^2(q_i) \); get \( V^1(q_i) \); the projection of \( V^2(q_i) \) on \( V^1 \);
11. if \((V^1(q_i) - V^2(q_i)) \) is not empty then
12. \( U = U \cup (V^1(q_i) - V^2(q_i)) \); add the newly generated line segments into \( U \);
13. end if
14. \( \Delta = \min\{\Delta, \Delta_i\} \);
15. \( U = U - \{V^1\} \);
16. end while
17. return \( S = [a_m - \Delta, a_m + \Delta] \times [b_1, b_2] \);

With the Slice Covering Algorithm introduced above, we can cover a non-empty slice of the rectangle \( V^2 \) with cost under \( O(n) \). In the next subsection, we will discuss how to use the techniques to further cover the whole rectangle and study the cost incurred.

### 4.2.3 The 2-D CRAWling Algorithm and the Upper Bound

To crawl all points from \( D \), we perform OPTIMAL-1D-CRAWL algorithm on the mid line of \( V^2 \) and get a slice \( S \), which is fully covered by the queries issued. This process can only have two results: either (1) \( V^2 \) is fully covered by slice \( S \), or (2) it generates two uncovered rectangles \( V^2_1 \) and \( V^2_2 \) from \( V^2 - S \). This process is recursively performed on any uncovered sub-rectangle of \( V^2 \) until no more uncovered sub-rectangles are left. The detailed algorithm is presented in Algorithm 3.

Since the slice produced by performing OPTIMAL-1D-CRAWL algorithm can always contain at least one point of \( D \), we can easily estimate the cost of the algorithm.

**Theorem 2. [Query Cost of the 2-D CRAWling Algorithm]** The cost of the 2-D crawling algorithm for crawling all points of a 2-D spatial database \( D \), which is bounded in a rectangle \( V^2 \), is under \( O(n^2) \), where \( n \) is the number of points in \( D \).
Proof: Each iteration of the 2-D crawling algorithm can only lead to either (1) the corresponding rectangle \( V_2^2 \) is fully covered by the slice \( S \) produced by the OPTIMAL-1D-CRAWL algorithm, or (2) at least one new point of \( D \) is covered by slice \( S \) and two new uncovered sub-rectangles are produced from \( V_2^2 - S \). The second kind of iterations can only happen at most \( n \) times since \( D \) contains only \( n \) points in total. Correspondingly, the first kind of iterations can occur at most \( n + 1 \) times. In other words, the 2-D crawling algorithm can only run the OPTIMAL-1D-CRAWL algorithm at most \( 2n + 1 \) times. Given that each call of the OPTIMAL-1D-CRAWL algorithm incurs at most \( O(n) \) queries, thus, the query cost of the 2-D crawling algorithm is under \((2n + 1) \times O(n) = O(n^2)\). \(\Box\)

Algorithm 3 2-D Crawling Algorithm

Input: \( D \): a database in 2-D space; \( V^2 = [a_1, a_2] \times [b_1, b_2] \supseteq D \)
Output: all points of \( D \)
1: \( U = \{V^2\} \) /*the set of uncovered subspaces (sub-rectangles)/
2: \( P = \{\} \) /*point locations currently*/
3: while \((U (U is not empty))\) do
4: \(V^2_i\) from \( U\) (\(V^2_i\) is an element in \( U\)); \( V^2_i \) is the mid line of \( V^2^2 \)
5: perform binary crawling algorithm on \( V^2_i \); and compute the slice \( S_i \)
6: add returned \( D \) points to \( P \)
7: if \( V^2_i \) is not fully covered by \( S_i \) then
8: two sub-rectangles \( V^2_{i1} \) and \( V^2_{i2} \) are produced from \( V^2_i - S_i \)
9: \( U = U \cup \{V^2_{i1}, V^2_{i2}\} \) /*add the two new subspaces*/
10: end if
11: \( U = U - \{V^2_i\} \) /*remove the processed subspace*/
12: end while
13: return \( P \)

5 Multiple Dimensional Analysis

Though 2-D spatial databases are the most popular ones in the real world, there still exist some applications of \( kn \)NN spatial databases in higher dimensional spaces (three or more dimensions). For example, the coastal.com website [4] allows users to perform \( kn \)NN queries for looking for glasses in 4-D space, with dimensions including temple arm length, lens height, lens width and DBL (distance between lenses). In order to give a solution for higher dimensional spaces and make our approach more complete, in this section we discuss how to extend our 2-D crawling algorithm to an m-D space.

The crawling algorithm for m-D spaces is designed in a recursive manner that an m-D space crawler is conducted by calling (m-1)-D crawling algorithm.

Let \( D = \{p_1, p_2, \ldots, p_n\} \) be a spatial database with \( kn \)NN interface in an m-D space, bounded in \( V^m = [a_1, a_1, r] \times [a_2, a_2, r] \times \cdots \times [a_m, a_m, r] \). Now each query \( q \) covers a sphere \( V^m(q) \) in the m-D space instead of a circle \( V^2(q) \) in the 2-D space. In order to crawl all points of \( D \), we first get the mid hyperplane \( V^{m-1} \) of \( V^m \) and conduct (m-1)-D crawling algorithm on \( V^{m-1} \) (suppose \( m > 3 \)). For each query \( q \), we use shrunken sphere, denoted as \( V^{m-1}(q) \), to cover the hyperplane \( V^{m-1} \).

Similar to the 2-D situation, if \( V^{m-1} \) is fully covered by \( V^{m-1}(q_1), V^{m-1}(q_2), \ldots, V^{m-1}(q_L) \) (shrunken spheres of \( q_1, q_2, \ldots, q_L \)), an m-D slice \( S \) should be covered by the original query spheres \( V^m(q_1), V^m(q_2), \ldots, V^m(q_L) \) in the m-D space. And \( S \) contains at least one database point. The detail of the proposed m-D crawling algorithm is presented in Algorithm 4.

Algorithm 4 m-D Crawling Algorithm, \( Alg(m^2, m) \)

Input: \( D \): a database in an m-D space; \( V^m = [a_1, a_1, r] \times \cdots \times [a_m, a_m, r] \)
Output: all points of \( D \)
1: \( U = \{V^m\} \) /*the set of uncovered subspaces*/
2: \( P = \{\} \) /*point locations currently*/
3: while \((U (U is not empty))\) do
4: get \( V^m_i \) from \( U \) (\( V^m_i \) is an element in \( U \)); \( V^m_i \) is the mid hyperplane of \( V^m \)
5: call \( Alg(V^{m-1}, m-1) \); get slice \( S_i \)
6: add returned \( D \) points to \( P \)
7: if \( V^m_i \) is not fully covered by \( S_i \) then
8: two sub-spaces \( V^m_{i1} \) and \( V^m_{i2} \) are produced from \( V^m_i - S_i \)
9: \( U = U \cup \{V^m_{i1}, V^m_{i2}\} \) /*add the two new subspaces*/
10: end if
11: \( U = U - \{V^m_i\} \) /*remove the subspace*/
12: end while
13: return \( P \)

To study the upper bound of the m-D crawling algorithm, we first look at the upper bound for covering a 1-D line segment using m-D queries. In fact, we can easily extend the claim of Proposition 1 to line covering in multidimensional spaces.

Corollary 2. [Query Bound for Covering 1-D Line using m-D Queries] Let \( V^1 \) be a line segment within \( V^m \) in an m-D space. \( D \) is an m-D database with \( kn \)NN interface, and points inside \( D \) are bounded in \( V^m \). Then, the cost of the OPTIMAL-1D-CRAWL algorithm, which uses m-D queries to cover \( V^1 \), is also bounded under \( O(n) \), where \( n \) is the number of points in \( D \).

In a similar manner, we can extend the rectangle covering algorithm, as well as the claim of Theorem 2 to rectangle covering in multidimensional spaces.

Corollary 3. [Query Cost for Covering 2-D Space in the m-D Space] Let \( V^2 \) be a 2-D rectangle within \( V^m \) in an m-D space. \( D \) is an m-D database with \( kn \)NN interface, bounded in \( V^m \). Then, we can design a crawling algorithm which uses shrunken spheres of m-D queries to cover the 2-D rectangle, with its cost bounded under \( O(n^2) \), where \( n \) is the number of points in \( D \).
Since the algorithm works recursively with respect to the number of dimensions, following the similar ideas as the ones in the 2-D situation we have the theorem below.

**Theorem 3. [Query Cost of the m-D Crawling Algorithm]** The query cost of the m-D crawling algorithm is $O(n^m)$, where $n$ is the number of points in $D$.

**Proof:** The m-D crawling algorithm needs to call $O(n)$ times of (m-1)-D crawling algorithm, which further requires to execute $O(n)$ times of (m-2)-D crawling algorithm. So, we can easily deduce that the cost of m-D crawling algorithm is under $O(n^m)$. □

## 6 Improving the Performance of the Crawling Algorithm using External Knowledge

In previous sections, we have presented our techniques for crawling kNN based databases. With the proposed approach, we can fully crawl all points of a database with kNN interface in 2-D space with cost under $O(n^2)$, independent of the point distribution in the space. Since the 2-D crawling algorithm works well over points that are more or less uniformly distributed over the space but poorly over a space featuring highly skewed geographic distributions.

Take Yahoo Local [5] as an example, points in Fig. 8 represent **restaurants** in New York State. As we can see, those points are heavily skewed in distribution, that most points are densely clustered in several cities, while fewer ones are located in other areas (though those regions are big in size). If we perform our OPTIMAL-1D-CRAWL algorithm on line $AB$ as shown in Fig. 8, the size of query $V^2(q)$ varies dramatically with respect to the locations of $q$ on line $AB$. The circle should be extremely small when $q$ is just located in the dense area (Buffalo City), while it becomes very large when $q$ is in other areas. As the result, the slice obtained by the algorithm should be much thin, which indicates that those bigger query circles on the line could not provide much advantage in covering the regions because the width of the slice is controlled by the smallest query circle of the algorithm.

This motivates us to propose a heuristic idea of first partitioning the entire space into subspaces with more homogeneous point density distributions, and then applying the algorithm over each subspace separately, in order to reduce the overall query cost.

To illustrate the implication of data distribution on the performance of our 2-D algorithm, we start by deriving a theoretical result in Section 6.1 which shows that when the underlying data distribution is uniform, the query cost of our algorithm indeed grows linearly (rather than quadratically as in the worst-case analysis in Section 5) with the number of output points. This result provides the key rationale for our technique described in Section 6.2, which partitions the entire 2-D space into regions of near-uniform data distributions, according to the pre-known auxiliary information about the underlying distribution. The experimental results in Section 7 demonstrate the effectiveness of this external-knowledge based space partitioning strategy on significantly reducing the query cost of kNN crawling.

### 6.1 Relationships between the data distribution and the crawling cost

Before looking into the details of our discussions, we first give a definition of uniform distribution of spatial databases.

**Definition 1. [Uniform Distribution]** Suppose $D \subseteq V^2$ is a kNN based database, where $V^2$ is a rectangle in a 2-D space. Let $c_1$ and $c_2$ be two real numbers such that $c_2 \geq c_1 > 0$. $D$ is called $c_1/c_2$ uniformly distributed in $V^2$ if, for any convex region $V^2 \subseteq V^2$, we always have $N(V^2)/|V^2| \leq c_2$ (when the size of $V^2$ is big enough, i.e. $|V^2| > \delta_0$), where $N(V^2)$ denotes the number of $D$ points inside $V^2$ and $|V^2|$ denotes the area size of $V^2$.

If a kNN database is $c_1/c_2$ uniformly distributed, the cost of the crawling algorithm can be linearly related to the total number of points of $D$.

**Theorem 4. [Crawling Cost of Uniformly Distributed Databases]** Suppose $D$ is a $c_1/c_2$ uniformly distributed database in $V^2$, where $V^2 = [a_1, a_2] \times [b_1, b_2]$. Then, the cost of the proposed 2-D crawling algorithm is bounded by $4c_1^{-3}c_2^{-3}k^{-1}n$, where $k$ is the value of kNN restriction and $n$ is the total number of points of $D$.

**Proof:** Let $S_1, S_2, \ldots, S_L$ denote those slices generated by the 2-D crawling algorithm for fully covering rectangle $V^2$, and they are ordered by their mid line positions (i.e. from the most left to the most right). Further, let $q_{i,1}, q_{i,2}, \ldots, q_{i,J_i}$ be those queries issued by the proposed 2-D crawling algorithm to cover slice $S_i$, which are ordered by their positions on the mid line of $S_i$ (i.e. from the top to the bottom). So, query set $Q = \{q_{i,j} | 1 \leq i \leq L, 1 \leq j \leq J_i\}$ can fully cover the whole space $V^2$. More importantly, those issued queries by the algorithm show the following properties (The detailed proof can be found in Appendix C):
In this section, we describe our experimental setup and conduct evaluations of the proposed crawling algorithms. Since most of the applications and existing works are only 2-D based, our experiments concentrate on the performance measurements of the proposed 2-D crawling algorithms. In particular, we demonstrate the scalability of the algorithms with different size of databases; we show the algorithms’

in longitude and latitude, the corresponding road information of the region can be returned in the format of a set of line segments. Then, we can compute the total length of the roads in by adding the lengths of those segments together, denoted as r.length. Let R be the whole region of the 2-D restaurant database, we regard the ratio r.length/R.length as the density descriptor of region r, denoted as r.density, which can be used for partitioning the whole region into density uniform subregions effectively.

6.3 Region Partitioning

Now, we discuss how to partition the region into subregions in detail. The process includes two steps: (1) uniformly partitioning the whole region R into \( n_1 \times n_2 \) small grids \( r_{i,j} \); then (2) grouping adjacent grids extensively if their densities are much similar in terms of road information of the grids. To simplify our discussions. We refer ‘density’ to ‘road density’ in this section.

The first step is simple, which only requires the grids to be small enough in size. For the second step, we implement a variant of DBSCAN algorithm to group those grids into different regions. In particular, we extensively merge adjacent grids starting at seed grids, which are the grids with the largest density (road density), until no more adjacent grids can be merged.

In detail, a grid \( r_{i,j} \) is density-reachable from a seed grid \( r_{i_0,j_0} \) if there exist a sequence of grids \( r_{i_1,j_1}, r_{i_2,j_2}, \ldots, r_{i_L,j_L} \) with \( r_{i_1,j_1} = r_{i_0,j_0} \) and \( r_{i_L,j_L} = r_{i,j} \), such that every pair of consequence grids \( r_{i_1,j_1}, r_{i_2,j_2}, \ldots, r_{i_L,j_L} \) are adjacent (with a common side), and each grid satisfies density requirement as \( (r_{i_0,j_0}.density - r_{i_1,j_1}.density)/r_{i_0,j_0}.density \leq \alpha \), where \( \alpha \) is a given threshold. All the reachable grids from seed \( r_{i_0,j_0} \) are merged together to formulate a density uniform region \( R(r_{i_0,j_0}) \). This process is recursively applied to the remaining grids \( R \setminus R(r_{i_0,j_0}) \) until no grids are left. The details of the proposed DBSCAN algorithm can be found in Algorithm 5. With this clustering algorithm, finally, all grids are assigned into different density uniform regions \( R_1, R_2, \ldots, R_m \). In fact, each region \( R_i \) may not be a rectangle, however it can be easily divided into several rectangles to be crawled using Algorithm 3.

7 EXPERIMENTS

In this section, we describe our experimental setup and conduct evaluations of the proposed crawling algorithms. Since most of the applications and existing works are only 2-D based, our experiments concentrate on the performance measurements of the proposed 2-D crawling algorithms. In particular, we demonstrate the scalability of the algorithms with different size of databases; we show the algorithms’

...
sensitivity to different parameter settings; we compare the performance of the proposed algorithm with the state of the art of crawling algorithm. Besides the detailed evaluations in 2-D situations, we also show the performance of m-D algorithm with respect to the number of dimensions in general.

7.1 Experiment Setup

1) Hardware and Platform: All experiments were performed on a 3.40 GHz Inter Core machine with 4GB of RAM. All algorithms were implemented in Java.

2) Data Sets: To evaluate the performance of our 2-D crawling algorithms on different data distributions, we generated two kinds of data sets with either uniform or skewed distributions in 2-D and other higher dimensional spaces. We also tested the proposed crawling algorithms on the real data sets Yahoo Local [5] in 2-D space and Eye-glasses [4] in 4-D space. We describe the details of these datasets respectively as follows.

Uniform synthetic data set: To evaluate the 2-D crawling algorithms, we generated uniformly distributed data sets in 2-D space \([0, 1000] \times [0, 1000]\). The coordinate \(p.x\) and \(p.y\) of a point \(p\) were generated uniformly and independently between 0 and 1000. The number of points in each data set was from 500 to 2500 with a step of 500. To demonstrate the performance of the m-D crawling algorithms with respect to the number of dimensions, we generated uniformly distributed synthetic data set in the 1-D space \([0, 1000]\), the 2-D space \([0, 1000] \times [0, 1000]\) and the 3-D space \([0, 1000] \times [0, 1000] \times [0, 1000]\). In each of the dimensional space, we generated 3 data sets with size ranging from 1000 to 5000 with a step of 2000.

Skewed synthetic data set: We also generated skewed distributed data sets in the 2-D space \([0, 1000] \times [0, 1000]\). Points in the skewed data sets were independently drawn from an exponential distribution. Similar to the method used in [3], by using the Inverse Transform Sampling method [7], we first generated a uniform random number \(u, u \in [0, 1]\). Then the coordinate \(p.x\) (or \(p.y\)) of a point \(p\) were generated by \(p.x = -ln(1 - u)/\lambda\), where \(\lambda\) is the rate parameter of the exponential distribution. We set \(\lambda = 10\) in our experiments. After generating points in \([0, 1] \times [0, 1]\), we then scaled the locations of these points to \([0, 1000] \times [0, 1000]\). The number of points in each data set varies from 500 to 2500 with a step of 500. We also generated skewed distributed synthetic data sets with respect to different number of dimensions.

Yahoo Local data sets: In 2-D space, three real data sets of restaurant points were crawled from Yahoo Local [5] for New York State (NY), Oklahoma State (OK) and Utah State (UT), respectively. There are in total 57584 points in NY, 8364 points in OK and 4841 points in UT, respectively. Their minimal bounded rectangles (MBR) are \([-79.76259, -71.777491] \times [40.477399, 45.015865], [-103.002455, -94.430662] \times [33.615787, 37.002312]\) and \([-114.052998, -109.041058] \times [36.997949, 42.001618]\), respectively. In these real data sets, points are highly skewed in distribution that most of points are clustered in several big cities while a few points are located in rural areas. Pink points in Fig. 8 show the distribution of restaurants in NY. In order to test the scalability of the algorithms with different size of the database, for each of the original ones we further generated 4 additional date sets by performing uniform sampling, such that the newly generated ones keep the same distribution, but are with, respectively, the sizes of 20\%, 40\%, 60\% and 80\% of the the original sizes.

Eye-glasses data sets: Eye glasses data set was crawled from the website [4] which supports kNN search on a combination of four attributes, including temple arm length, lens height, lens width and DBL (distance between lenses), respectively. There are in total 3067 points in this data set, which are bounded in \([115.0, 155.0] \times [10.0, 49.0] \times [40.0, 64.0] \times [9.0, 24.0]\). We also generated 2 additional data sets with size of 20\% and 60\% of the original one by uniform sampling.

7.2 kNN Crawling Algorithms

Here we list several algorithms which will be evaluated and tested in our experiments.

The m-D crawling algorithm: This is the general algorithm for crawling the database with a kNN interface in m-D spaces. We demonstrated the performance sensitiveness of the algorithm to the number of dimensions. In particular, we compared the performances of the proposed crawling algorithm in 1-D, 2-D and 3-D spaces.

The 2-D crawling algorithm: This algorithm is a special case of m-D crawling algorithm in a 2-D space.
We extensively evaluated this algorithm with different settings.

The 2-D crawling algorithm with external knowledge: The 2-D crawling algorithm is performed after partitioning the 2-D space using external knowledge. This is the most advanced crawling algorithm we proposed in 2-D space.

The DCDT crawling algorithm: This algorithm was proposed in work [3]. To our best knowledge, this algorithm is the state of the art of crawling algorithm for kNN based databases in 2-D space. In their work, the authors implemented a technique, called constrained delaunay triangulation, to always partition the uncovered regions into triangles, then issued the new query on the center of the biggest triangle. Their algorithm recursively repeated this process until no uncovered triangles are left.

7.3 Performance measure

The performance of the crawling algorithms was measured by the total number of queries issued to crawl all points in a given space.

7.4 Experimental results

We set $k = 10$ as the default top-$k$ restriction for the 2-D synthetic data sets and Eye-glasses, and $k = 100$ for the m-D synthetic data sets and Yahoo Local data sets. For the 2-D crawling algorithm with external knowledge, we partitioned the MBR into $100 \times 100$ grids with some preliminary experiments. Then for NY, each grid was with length 0.04538466 and width 0.07985099 measured in degree. The default value of $a$ for computing density reachable was set as $a = 0.35$ unless otherwise specified (the impact of this parameter is demonstrated later in this section).

1) Performance of the algorithms in 2-D space: We first compared our proposed crawling algorithms with DCDT crawling algorithm in 2-D spaces.

On the synthetic data sets: Since the synthetic data sets were generated without external knowledge for describing density distributions, here we only compared our basic 2-D crawling algorithm (without using external knowledge) with DCDT algorithm. Figs. 9 shows the performance comparisons between the proposed 2-D crawling algorithm and DCDT algorithm on the uniformly distributed data sets and the skewed data sets respectively, where x-axis represents the size of different synthetic data sets while y-axis indicates the crawling cost. We can find that the proposed crawling algorithm outperform DCDT algorithm in all the situations.

We can also find the scalability of the algorithms with different size of the databases from the figure. Besides, it costs more queries to crawl all points when the hidden points are in skewed distribution.

On the real data sets: On the real data sets, we selected not only our basic 2-D crawling algorithm but also our external knowledge based 2-D crawling algorithm to compare with DCDT algorithm on the restaurant data sets Yahoo Local NY, UT and OK. Figs. 10a, 10b, and 10c show the performance comparisons of those crawling algorithms on the three real data sets, respectively. We can see that the 2-D crawling algorithm with external knowledge has the best performance among all. We also find that, on NY, the external knowledge based algorithm outperforms other algorithms with more margin because of the severe skewed distribution of the NY data set. To understand the reasons behind the difference of the performance, we recall the strategies used by those algorithms to issue ‘next query point’ in their crawling. DCDT algorithm assumes the ‘optimal’ query should be the center of the biggest triangle in its partition of uncovered space. However, the skewed distribution of objects often makes the triangles much irregularly shaped, consequently generates heavy overlapped queries. On the other hand, our algorithms issue the next query as the point which is as far as possible to the current covered regions, thus can generate less overlapped queries.

2) Different kNN Restrictions in 2-D space: Now we study the sensitiveness of the algorithm to the value of $k$ in the kNN restriction. To this end, we allowed $k$ to be 100, 200, 300, 400 and 500 respectively, and applied the external knowledge based crawling algorithm on data sets with different sizes (as shown in last subsection). The query cost is shown in Figs. 11a, 11b and 11c for the three data sets NY, UT and OK respectively, where the x-axis represents different values for $k$, and the y-axis is the query cost. Different curves in the same figure show the corresponding performance of the algorithm performed on different size of data sets. From those figures, we can see that a larger $k$ can always reduce the query cost. Especially, the value of $k$ is more significant for reducing the crawling cost of a larger data set.

3) Performance of m-D crawling algorithm in m-D space: In this paper, we had implemented a
general crawling algorithm for databases with a $k$NN interface in m-D spaces. To know the algorithm’s sensitivity to the number of dimensions, we tested the performance of our crawling algorithm on 1-D, 2-D and 3-D databases, respectively.

On synthetic data sets of different dimensions: Figs. 12a and 12b show the performance of the m-D crawling algorithm on the uniform and skewed distribution of data sets in different dimensions, where the x-axis represents the sizes of data sets, while the y-axis indicates the query cost. We can find that the performance of the crawling algorithm drops down dramatically when increasing the number of dimensions.

**Different kNN on Eye-glasses data set:** Eye-glasses is a real data set in 4-D space. Now we demonstrate the performance of our crawling algorithm on this real data set. We tested the performance of the 4-D crawling algorithm on the data sets with different value of $k$NN restriction. From Fig. 12c, we can see that a larger value of $k$ can significantly reduce the query cost of the algorithm. This result is much similar to the one in the 2-D situation.

**4) Impact of parameter settings:** Density threshold, denoted as $a$, is the threshold for controlling the density margin to merge two adjacent grids in the 2-D crawling algorithm with external knowledge. Large value of $a$ may lead to merging grids with much different densities, while small value may produce too many sub-regions which also cause heavy overlapped queries around their boundaries (boundary points may be crawled by both regions). The impact of $a$ to the performance of the algorithm for data set NY is presented in Fig. 13a, where the values of $a$ are set to 0.1, 0.3, 0.5, 0.7 and 0.9. The query cost is high for both large and small value of $a$. Large $a$ still leads to skewed data distribution in the sub-regions. An extreme case is that the partitioned subspace is the MBR itself if we set $a = 1$. On the other hand, the small value of $a$ results in too many sub-regions which also causes heavy overlapped queries around their boundaries. In fact, we find the performance is very stable in a range $a \in [0.1, 0.3]$. The testing of $a$ for UT and OK are in Figs. 13b and 13c respectively, with the similar results.

**8 RELATED WORK**

There have been a significant number of research works in the areas related to crawling contents in web databases. The most relevant studies to our work are presented in [2], [3]. In [3], the authors also dealt with the problem of crawling $k$NN based spatial databases in 2-D situation. The work proposed two methods, Quadtree and Constrained Delaunay Triangulation, to partition an uncovered 2-D space into sub-regions (triangles), and assumed that the best location for the next query was the center of the
largest uncovered triangle. With recursively executing the algorithm, extensive number of small triangles (or sub-regions) were produced which caused heavy overlapped queries. Further more, the upper bound of their algorithms was not related to the database size, but the distribution of the hidden points to be crawled. While in our study, we theoretically proved that the upper bound of our crawling algorithm is only related to size of the database (the number points of the database), without regarding point distribution. We also extend our algorithm to higher dimensional databases as a general solution.

Other works on crawling hidden web databases through restrictive search interfaces dealt with either textual interface (e.g., keyword) \[8\], \[9\], \[10\], \[11\] or structure based interface (e.g., drop-down list) \[12\], \[13\], \[14\], \[15\], \[16\], \[17\], \[18\]. The challenge in the former studies was how to find meaningful queries. For example, \[8\] proposed an approach to siphon hidden web database. It sampled from the acquired documents to discover a set of keywords with high recall. \[9\] also automatically generated meaningful queries to the hidden web database. The authors modeled the problem as a set covering problem. In order to cover maximum number of web pages with minimal cost, they performed a greedy algorithm to issue queries which may return more results. In \[11\], Jiang modeled the crawler as agent and deep web database as the environment, the agent selected an action (query) according to a long-term reward. While the challenge in the latter studies was how to get all hidden contents with a small number of queries which were the combinations of the values of the attributes from the querying interface. For example, in \[12\], Raghavan and Garcia-Molina used a task-specific, human-assisted approach to extract data from the hidden web. \[14\] modeled the structured web database into a distinct attribute-value graph. Then the crawling problem was transformed into a graph traversal one. To improve the qualities of the selected queries, \[18\] generated queries leveraging two data sources which are query logs and knowledge bases like Freebase. Although almost all of the existing studies on crawling hidden web databases were based on heuristic approaches, \[19\] established the theoretical analysis on this problem with structure-based interface. Their crawling algorithm was asymptotically optimal. However, none of those works tackled kNN based web databases except \[2\], \[3\].

9 Conclusion
In this paper, we study the problem of crawling the LBS through the restricted kNN search interface. Although hidden points usually exist in 2-D space, there

Fig. 12. (a) Performance of the m-D crawling algorithm on the uniform synthetic data sets; (b) Performance of the m-D crawling algorithm on the skewed synthetic data set; (c) Sensitivity of the algorithms different $k$NN on Eye-glasses data set.

Fig. 13. Impact of parameter a (a) on NY; (b) on UT; (c) on OK.
are some applications with points in higher dimensional spaces. We extend the 2-D crawling algorithm to the general m-D space, and give the m-D crawling algorithm with theoretical upper bound analysis. For 2-D space, we take external knowledge into consideration to improve the crawling performance. The experimental results show the effectiveness of our proposed algorithms. In this study, the proposed algorithms crawl data objects by given a rectangle (cube) in the spatial space. In the general situation when the bounded region of the objects is irregular, it can be pre-partitioned into a set of rectangles (cubes) before using the techniques proposed in this paper.

REFERENCES


Hui Yan is a PhD student in computer science from University of Science and Technology of China jointly with University of Macau.

Zhiguo Gong is currently an associate Professor in the Department of Computer and Information Science, University of Macau, Macau, China. His research interests include Database, Web Information Retrieval and Web Mining. He is a member of IEEE.

Nan Zhang received the B.S. degree in computer science from Peking University in 2001 and the PhD degree in computer science from Texas A&M University in 2006. He is an Associate Professor of computer science at The George Washington University. His current research interests include databases and information security/privacy. Nan Zhang received the NSF CAREER Award in 2008. He is a member of the IEEE.

Tao Huang received his M.S. and Ph.D. degrees from University of Science and Technology of China in 1991 and 1994, respectively. He is a Professor in the Institute of Software, Chinese Academy of Sciences (ISCAS). His research interests include databases and software engineering.

Hua Zhong received his B.S. and Ph.D. degrees in computer science from the Institute of Software, Chinese Academy of Sciences (ISCAS) in 1996 and 1999, respectively. He is a Professor in ISCAS. His research interests include distributed computing, middleware and software engineering. He has published over 40 papers on international journals and conferences.

Jun Wei received his B.S. degree in Computer Science (1992) and his Ph.D. in Computer Science (1997) from the Wuhan University, China. He was a visiting researcher in the CSE Department of the Hong Kong University of Science and Technology in 2000. He is a Professor in the Institute of Software, Chinese Academy of Sciences (ISCAS). His area of research is software engineering and distributed computing, with emphasis on middleware based distributed software engineering.