Supporting Range Queries on Web Data Using *k*-Nearest Neighbor Search

Wan D. Bae[1], Shayma Alkobaisi[1], Seon Ho Kim[1], Sada Narayanappa[1], and Cyrus Shahabi[2,⋆]

[1] Department of Computer Science, University of Denver, USA
{wbae,salkobai,seonkim,snarayan}@cs.du.edu
[2] Department of Computer Science, University of Southern California, USA
shahabi@usc.edu

Abstract. A large volume of geospatial data is available on the web through various forms of applications. However, access to these data is limited by certain types of queries due to restrictive web interfaces. A typical scenario is the existence of numerous business web sites that provide the address of their branch locations through a limited "nearest location" web interface. For example, a chain restaurant's web site such as McDonalds can be queried to find some of the closest locations of its branches to the user's home address. However, even though the site has the location data of all restaurants in, for example, the state of California, the provided web interface makes it very difficult to retrieve this data set. We conceptualize this problem as a more general problem of running spatial range queries by utilizing only k-Nearest Neighbor (k-NN) queries. Subsequently, we propose two algorithms to cover the rectangular spatial range query by minimizing the number of k-NN queries as possible. Finally, we evaluate the efficiency of our algorithms through empirical experiments.

1 Introduction

Due to the recent advances in geospatial data acquisition and the emergence of diverse web applications, a large amount of geospatial data have become available on the web. For example, numerous businesses release the locations of their branches on the web. The web sites of government organizations such as the US Postal Office provide the list of their offices close to one's residence. Various non-profit organizations also publicly post a large amount of geospatial data for different purposes.

⋆ The author's work is supported in part by the National Science Foundation under award numbers IIS-0324955 (ITR), EEC-9529152 (IMSC ERC) and IIS-0238560 (PECASE) and in part by unrestricted cash gifts from Microsoft and Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Unfortunately, access to these abundant and useful geospatial data sets is only possible through their corresponding web interfaces. These interfaces are usually designed for one specific query type and hence cannot support a more general access to the data. For example, the McDonalds web site provides a restaurant locator service through which one can ask for the five closest locations from a given zip code. This type of web interface to search for a number of "nearest locations" from a given geographical point (e.g., a mailing address) or an area (e.g., a zip code) is very popular for accessing geospatial data on the web. It nicely serves the intended goal of quick and convenient dissemination of business location information to potential customers. However, if the consumer of the data is a computer program, as in the case of web data integration utilities (e.g., wrappers) and search programs (e.g., crawlers), such an interface may be a very inefficient way of accessing the data. For example, suppose a web crawler wants to access the McDonalds web-site to retrieve all the restaurants in the state of California. Even though the site has the required information, the interface only allows the retrieval of five locations at a time. Even worse, the interface needs the center of the search as input. Hence, the crawler needs to identify a set of *nearest location* searches that both covers the entire state of California (for completeness) and has minimum overlap between the result sets (for efficiency).

In this paper, we conceptualize the aforementioned problem into a more general problem of supporting spatial range queries using k-Nearest Neighbor (k-NN) search. Besides web data integration and search applications, the solution to this more general problem can be beneficial for other application domains such as sensor networks when sensors have a limited number of channels to communicate with their nearest neighbors. Assuming that we have no knowledge about the distribution of the data sets and no control over the value of k, we propose two algorithms: the Quad Drill Down (QDD) and Dynamic Constrained Delaunay Triangulation ($DCDT$) algorithms. The efficiencies of the algorithms are empirically compared.

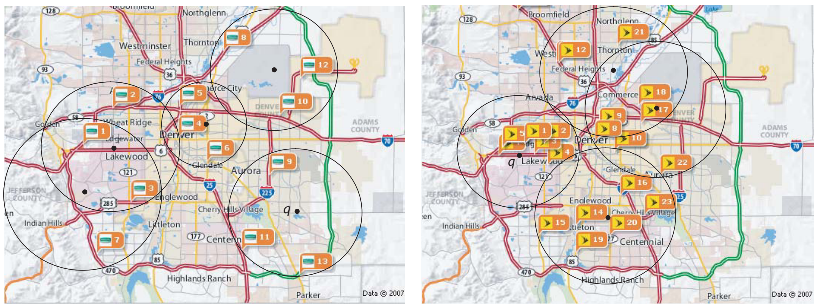
The remainder of this paper is organized as follows: In Section 2, the problem is formally defined. Our proposed algorithms are discussed in Section 3. Next, the algorithms are experimentally evaluated in Section 4. The related work is presented in Section 5. Finally, Section 6 concludes the paper.

2 Problem Definition

A range query in spatial database applications returns all objects inside a query region R. A k-Nearest Neighbor (k-NN) query returns the k closest objects to a query point q. More formal definitions for the range query and the k-NN query can be found in [1,2].

Our focus is on finding all objects within a given rectangular query region R using a minimum number of k-NN searches as supported in various web applications. The complexity of this problem is unknown. Therefore only approximation results are provided. In reality, there can be more than one web site involved in the given range query. Then, the general problem is to find all objects within the query region R using k-NN searches on each of these web sources and to integrate the results from the sources into one final query result. To make the problem more general, no assumptions are made about the data set, i.e., neither the number of objects in the data set nor the data distribution is known.

Figure 1 (a) and (b) illustrate the challenges of this general problem through examples. Suppose that we want to find the locations of all the points in a given region (rectangle) and the only available interface is a k-NN search with a fixed k, $k = 3$ in (a) and $k = 6$ in (b). Hence, the only parameter that we can vary is the query points for the k-NN searches. Given a query point q, the result of the k-NN search is a set of k nearest points to q. It defines a circle centered at q with radius equal to the distance from q to its k^{th} nearest neighbor (i.e., the 3^{rd} closest point in (a)). The area of this circle, covering a part of the rectangle, determines the covered region of the rectangle. To *complete* the range query, we need to identify a set of input locations corresponding to a series of k-NN searches that would result in a set of circles covering the entire rectangle.



(a) A range query using 3-NN search (b) A range query using 6-NN search

Fig. 1. Range queries on the web data

The optimization objective of this problem is to perform as few k-NN searches as possible. This is because each k-NN search results in communication overhead between the client and the server in addition to the actual execution cost of the k-NN search at the server. Hence, the circles should have as few overlaps as possible. In addition, the fact that we do not know the radius of each circle prior to the actual evaluation of its corresponding k-NN query makes the problem even harder. Hence, the *sequence* of our k-NN searches become important as well. To summarize, the optimal solution should find a *minimum set* of query locations that minimizes the number of k-NN searches to completely cover the given query region. In many web applications, the client has no knowledge about the data set at the server and no control over the value of k (i.e., k is fixed depending on the web applications). These applications return a fixed number of k nearest objects to the user's query point. Considering a typical value of k in real applications, which ranges from 5 to 20 [3], multiple queries are evaluated to completely cover a reasonably large region.

Without loss of generality, and to simplify the discussion, we made the following assumptions: 1) a k-NN search is supported by a single web source, 2) the web source supports only one type of k-NN search while there can be multiple k-NN searches based on the value of k, 3) the parameter to the k-NN search is the query point q, 4) the value of k is fixed, i.e., the user has no control over k.

3 Range Queries Using *k*-NN Search

Our approach to the range query problem on the web is as follows: 1) divide the query region R into subregions, $R = \{R_1, R_2, ..., R_m\}$, such that any $R_i \in R$ can be covered by a single k-NN search, 2) obtain the resulting points from the k-NN search on each of the subregions, 3) combine the partial results to provide a complete result.

The main focus is how to divide R so that the total number of k-NN searches can be minimized. We consider the three following approaches: 1) a naive approach: divide the entire query region R into equi-sized subregions (grid cells) such that any cell can be covered by a single k-NN search. 2) a recursive approach: conduct a k-NN search in the current query region, divide the current query region into subregions with same sizes if the k-NN search fails to cover R, and call k-NN search for each of these subregions. Repeat the process until all subregions are covered. 3) a greedy and incremental approach: divide R into subregions with different sizes. Then select the largest subregion for the next k-NN search. Check the covered region by the k-NN search and select the next largest subregion for another k-NN search.

Table 1. Notations

Notation	Description
R	a query region (rectangle)
P_{ll}	the lower-left corner point of R
P_{ur}	the upper-right corner point of R
q	the query point for a k-NN search
R_q	half of the diagonal of R; the distance from q to P_{ll}
C_{Rq}	circle inscribing R; the circle of radius R_q centered at q
P_k	the set of k nearest neighbors obtained by a k-NN search ordered ascendingly by distance from q
r	the distance from q to the farthest point p_k in P_k
C_r	k-NN circle; the circle of radius r centered at q
$C_r^{'}$	tighter bound k-NN circle; the circle of radius $\epsilon \cdot r$ centered at q

The notations in Table 1 are used throughout this paper and Figure 2 illustrates our approach to evaluate a range query using k-NN searches. If $r > R_q$, then the k-NN search must have returned all the points inside C_{R_q}. Therefore, we can obtain all the points inside the region R after pruning out any points of

P_k that are outside R but inside C_r. Then, the range query is complete. Otherwise, we need to perform some additional k-NN searches to completely cover R. For example, if we use 5-NN search for the data set shown in Figure 2, then the resulting C_r is smaller than C_{R_q}. For the rest of this section, we focus on the latter case that is a more realistic scenario. Note that it is not sufficient to have $r \geq R_q$ in order to retrieve all the points inside R when q has more than one k^{th} nearest neighbor since one of them will be randomly returned as q's k^{th}-NN. Hence, we use $r > R_q$ as a terminating condition in our algorithms.

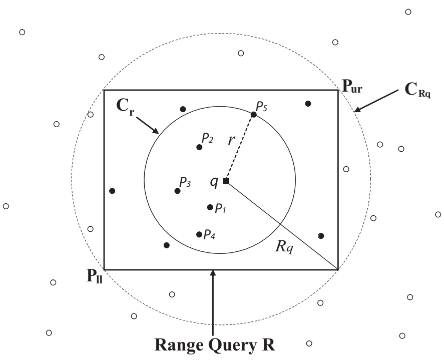


Fig. 2. A range query using 5-NN search

In a naive approach, R is divided into a grid where the size of cells is small enough to be covered by a single k-NN search. However, it might not be feasible to find the exact cell size with no knowledge of the data sets. Even in the case that we obtain such a grid, this approach is inefficient due to large amounts of overlapping areas among k-NN circles and wasted time to search empty cells (consider that most real data sets are not uniformly distributed). Thus, we provide a recursive approach and an incremental approach in the following subsections.

3.1 Quad Drill Down (QDD) Algorithm

We propose a recursive approach, the Quad Drill Down (QDD) algorithm, as a solution to the range query problem on the web using the properties of the quad-tree. A quad-tree is a tree whose nodes either are leaves or have four children, and it is one of the most commonly used data structures in spatial databases [11]. We adopt the partitioning idea of the quad-tree but we do not actually construct the tree. The main idea of QDD is to divide R into equal-sized quadrants, and then recursively divide quadrants further until each subregion is fully covered by a single k-NN search so that all objects in it are obtained.

Algorithm 1 describes QDD. First, a k-NN search is invoked with a point q which is the center of R. Next, the k-NN circle C_r is obtained from the k-NN result. If C_r is larger than C_{R_q}, it entirely covers R, then all objects in R are

Algorithm 1. QDDrangeQuery(P_{ll}, P_{ur})

```
1: $q \leftarrow$ getCenterOfRegion($P_{ll}$, $P_{ur}$)
2: $R_q \leftarrow$ getHalfDiagonal($q, P_{ll}, P_{ur}$)
3: $Knn\{\} \leftarrow$ add kNNSearch($q$)
4: $r \leftarrow$ getDistTo$K^{th}$NN($q, Knn\{\}$)
5: if $r > R_q$ then
6:     $result\{\} \leftarrow$ pruneResult($Knn\{\}, P_{ll}, P_{ur}$)
7: else
8:     clear $Knn\{\}$
9:     $P_0 \leftarrow (q.x, P_{ll}.y)$
10:    $P_1 \leftarrow (P_{ur}.x, q.y)$
11:    $P_2 \leftarrow (P_{ll}.x, q.y)$
12:    $P_3 \leftarrow (q.x, P_{ur}.y)$
13:    $Knn\{\} \leftarrow$ add QDDrangeQuery($P_{ll}, q$) // q1
14:    $Knn\{\} \leftarrow$ add QDDrangeQuery($P_0, P_1$) // q4
15:    $Knn\{\} \leftarrow$ add QDDrangeQuery($P_2, P_3$) // q2
16:    $Knn\{\} \leftarrow$ add QDDrangeQuery($q, P_{ur}$) // q3
17:    $result\{\} \leftarrow Knn\{\}$
18: end if
19: return $result\{\}$
```

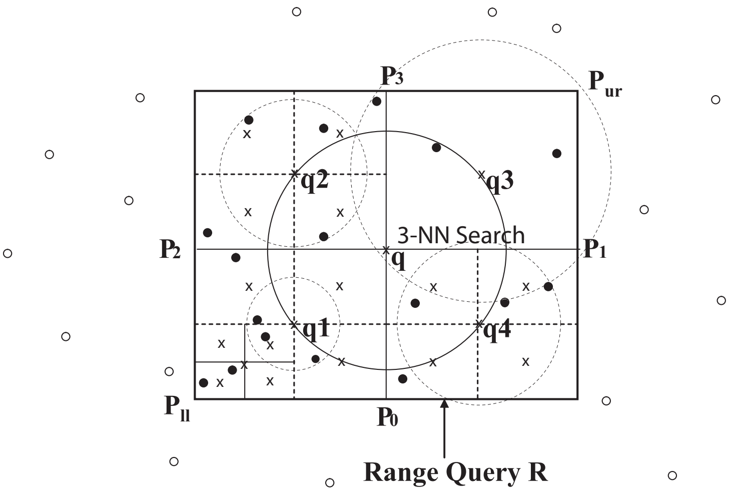


Fig. 3. A *QDD* Range Query

retrieved. Finally, a pruning step is necessary to retrieve only the objects that are inside R – a trivial case. However, if C_r is smaller than or equal to C_{R_q}, the query region is partitioned into four subregions by equally dividing the width and height of the region by two. The previous steps are repeated for every new subregion. The algorithm recursively partitions the query region into subregions until each subregion is covered by a k-NN circle. The pruning step eliminates those objects that are inside the k-NN circle but outside the subregion. An example of QDD is illustrated in Figure 3 (when k=3), where twenty one k-NN calls are required to retrieve all the points in R.

3.2 Dynamic Constrained Delaunay Triangulation ($DCDT$) Algorithm

In this section, we propose the Dynamic Constrained Delaunay Triangulation ($DCDT$) algorithm − a greedy and incremental approach to solve the range query on web data using the Constrained Delaunay Triangulation (CDT)[1]. $DCDT$ uses triangulations to divide the query range and keeps track of covered triangles by k-NN circles using the characteristics of CDT. $DCDT$ greedily selects the largest uncovered triangle for the next k-NN search while QDD follows a pre-defined order. To overcome redundant k-NN searches on the same area, $DCDT$ includes a propagation algorithm to cover the maximum possible area within a k-NN circle. Hence, no k-NN search will be wasted because a portion of a k-NN circle is always added to the covered area of the query range.

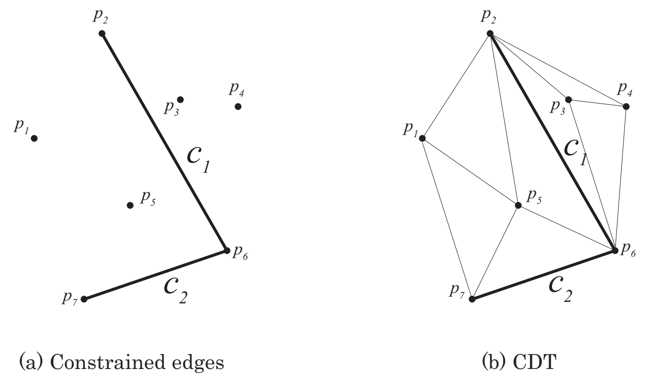


Fig. 4. An example of Constrained Delaunay triangulation

Given a planar graph G with a set of vertices, P, a set of edges, E, and a subset of edges, C, in the plane, CDT of G is a triangulation of the vertices of G that includes C as part of the triangulation [12], i.e., all edges in C appear as edges of the resulting triangulation. These edges are referred to as constrained edges and they are not crossed (destroyed) by any other edges of the triangulation. Figure 4 illustrates a graph G and the corresponding CDT. Figure 4 (a) shows a set of vertices, $P = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7\}$, and a set of constrained edges, $C = \{C_1, C_2\}$. Figure 4 (b) shows the result of CDT that includes the constrained edges of C_1 and C_2 as part of the triangulation.

[1] The terms, CDT and $DCDT$, are used for both the corresponding triangulation algorithms and the data structures that support these algorithms in this paper.

We define the following data structures to maintain the covered and uncovered regions of R:

- *pList*: a set of all vertices of G; initially $\{P_{ll}, P_{rl}, P_{lr}, P_{ur}\}$
- *cEdges*: a set of all constrained edges of G; initially empty
- *tList*: a set of all uncovered triangles of G; initially empty

pList and *cEdges* are updated based on the current covered region by a k-NN search. R is then triangulated (partitioned into subregions) using CDT with the current *pList* and *cEdges*. For every new triangulation, we obtain a new *tList*. $DCDT$ keeps track of covered triangles (subregions) and uncovered triangles; covered triangles are bounded by constrained edges (i.e., all three edges are constrained edges) and uncovered triangles are kept in *tList*, which are sorted in descending order by the area of the triangles. $DCDT$ chooses the largest uncovered triangle in *tList* and calls k-NN search using the centroid (which always lies inside the triangle as opposed to the center) of the triangle as the query point. Our algorithm uses a heuristic approach − to pick the largest triangle from the list of uncovered triangles. The algorithm terminates when no more uncovered triangles exist. For the rest of this section, we describe the details of $DCDT$ shown in Algorithm 2 and Algorithm 3.

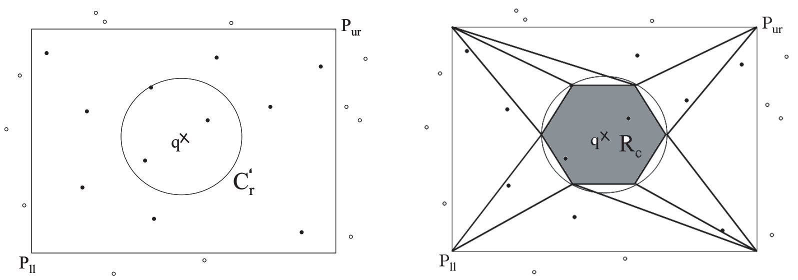
$DCDT$ invokes the first k-NN search using the center point of the query region R as the query point q. Figure 5 (a) shows an example of a 3-NN search. If the resulting k-NN circle C_r completely covers R ($r > R_q$), then we prune and return the result − a trivial case. If C_r does not completely cover R, $DCDT$ needs to use $C_r^{'}$ (a little smaller circle than C_r) for checking covered region from this point for further k-NN searches. We have previously discussed the possibility that the query point q has more than one k^{th} nearest neighbor and one of them is randomly returned. In that case, $DCDT$ may not be able to retrieve all the points in R if it uses C_r. Hence we define $C_r^{'} = \epsilon \cdot C_r$, where $0 < \epsilon < 1$ ($\epsilon = 0.99$ in our algorithm).

$DCDT$ creates an n-gon inscribed into $C_r^{'}$. Choosing the value of n depends on the tradeoff between computation time and the coverage of area; $n = 6$ (a hexagon) is used in our experiments as shown in Figure 5 (b). All vertices of the n-gon are added into *pList*. To mark the n-gon as a covered region, the n-gon is triangulated and the resulting edges are added as constrained edges into *cEdges* (*getConstrainedEdges*() line 15 of Algorithm 2). The algorithm constructs a new triangulation with the current *pList* and *cEdges*, then a newly created *tList* is returned (*constructDCDT*() line 16 of Algorithm 2).

$DCDT$ selects the largest uncovered triangle from *tList* for the next k-NN search. With the new $C_r^{'}$, $DCDT$ updates *pList* and *cEdges* and creates a new triangulation. For example, if an edge lies within $C_r^{'}$, then $DCDT$ adds it into *cEdges*; on the other hand, if an edge intersects $C_r^{'}$, then the partially covered edge is added into *cEdges*. Figure 6 shows an example of how to update and maintain *pList*, *cEdges* and *tList*. $\triangle_1$ ($\triangle_{max}$) is the largest triangle in *tList*, therefore $\triangle_1$ is selected for the next k-NN search. $DCDT$ uses the centroid of $\triangle_1$ as the query point q for the k-NN search. $C_r^{'}$ partially covers $\triangle_1$: vertices v_5 and v_6 are inside $C_r^{'}$ but vertex v_3 is outside $C_r^{'}$. $C_r^{'}$ intersects $\triangle_1$ at k_1 along $\overline{v_3 v_6}$ and

Algorithm 2. DCDTrangeQuery(P_{ll}, P_{ur})

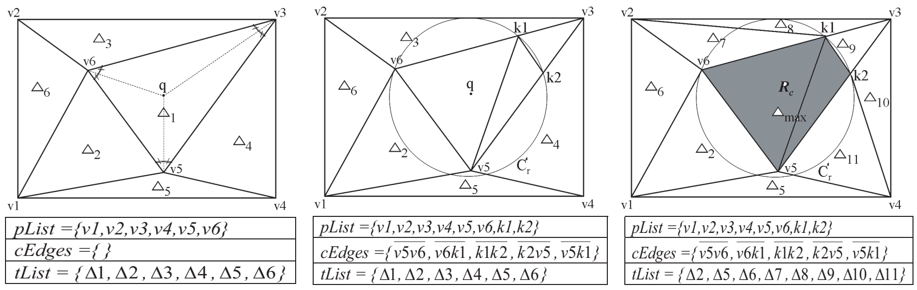
```
1: $pList\{\} \leftarrow \{P_{ll}, P_{rl}, P_{lr}, P_{ur}\}$;
2: $cEdges\{\} \leftarrow \{\}$;
3: $tList\{\} \leftarrow \{\}$;
4: $Knn\{\} \leftarrow \{\}$; all objects retrieved by $k$-NN search
5: $q \leftarrow$ getCenterOfRegion($P_{ll}$, $P_{ur}$)
6: $Knn\{\} \leftarrow$ add kNNSearch($q$)
7: $r \leftarrow$ getDistTo$K^{th}$NN($q, Knn\{\}$)
8: $C_r \leftarrow$ circle centered at $q$ with radius $r$
9: $R_q \leftarrow$ getHalfDiagonal($q, P_{ll}, P_{ur}$)
10: $\epsilon = 0.99$
11: if $r \le R_q$; $C_r$ does not cover the given region $R$ then
12:    $C_r^{'} \leftarrow$ circle centered at $q$ with radius $\epsilon \cdot r$
13:    $N_g \leftarrow$ create an n-gon inscribing $C_r^{'}$ with $q$ as the center
14:    $pList\{\} \leftarrow$ add {all vertices of $N_g$}
15:    $cEdges\{\} \leftarrow$ getConstrainedEdges({all vertices of $N_g$})
16:    $tList\{\} \leftarrow$ constructDCDT($pList\{\}$, $cEdges\{\}$)
17:    while $tList$ is not empty do
18:       mark all triangles in $tList$ to unvisited
19:       $\triangle_{max} \leftarrow$ getMaxTriangle()
20:       $q \leftarrow$ getCentroid($\triangle_{max}$)
21:       $Knn\{\} \leftarrow$ add kNNSearch($q$)
22:       $r \leftarrow$ getDistTo$K^{th}$NN($q, Knn\{\}$)
23:       $C_r^{'} \leftarrow$ circle centered at $q$ with radius $\epsilon \cdot r$
24:       checkCoverRegion($\triangle_{max}, C_r^{'}, pList\{\}, cEdges\{\}$)
25:       $tList\{\} \leftarrow$ constructDCDT($pList\{\}$, $cEdges\{\}$)
26:    end while
27: end if
28: $result\{\} \leftarrow$ pruneResult($Knn\{\}, P_{ll}, P_{ur}$)
29: return $result\{\}$
```



(a) Query point q and R (b) A hexagon as a covered region

Fig. 5. An example of first k-NN call in query region R

at k_2 along $\overline{v_3v_5}$, hence, k_1 and k_2 are added into *pList* (*getCoveredVertices*() line 3 of Algorithm 3). Let R_c be the covered area (polygon) of $\triangle_1$ by $C_r^{'}$ (Figure 6 (c)). Then R_c is triangulated and the resulting edges, $\overline{k_1k_2}$, $\overline{k_1v_6}$,



pList ={v1,v2,v3,v4,v5,v6}
cEdges ={ }
tList = { Δ1, Δ2, Δ3, Δ4, Δ5, Δ6 }

pList ={v1,v2,v3,v4,v5,v6,k1,k2}
cEdges ={ $\overline{v5v6}$, $\overline{v6k1}$, $\overline{k1k2}$, $\overline{k2v5}$, $\overline{v5k1}$ }
tList = { Δ1, Δ2, Δ3, Δ4, Δ5, Δ6 }

pList ={v1,v2,v3,v4,v5,v6,k1,k2}
cEdges ={ $\overline{v5v6}$, $\overline{v6k1}$, $\overline{k1k2}$, $\overline{k2v5}$, $\overline{v5k1}$ }
tList = { Δ2, Δ5, Δ6, Δ7, Δ8, Δ9, Δ10, Δ11 }

(a) Step 1 (b) Step 2 (c) Step 3

Fig. 6. Example of pList, cEdges and tList: covering $\triangle_{max}$

Algorithm 3. checkCoverRegion($\triangle$, C_r', *pList*{}, *cEdges*{})

```
1: if $\triangle$ is marked unvisited then
2:     mark $\triangle$ as visited
3:     $localPList\{\} \leftarrow$ getCoveredVertices($\triangle, C_r'$)
4:     $localCEdges\{\} \leftarrow$ getConstrainedEdges($localPList$)
5:     if $localCEdges$ is not empty then
6:         $pList\{\} \leftarrow$ add $localPList\{\}$
7:         $cEdges\{\} \leftarrow$ add $localcEdges\{\}$
8:         $neighbors\{\} \leftarrow$ findNeighbors($\triangle$)
9:         for every $\triangle_i$ in neighbors{}, $i = 1, 2, 3$ do
10:            checkCoverRegion($\triangle_i, C_r', pList\{\}, cEdges\{\}$)
11:        end for
12:    end if
13: end if
```

$\overline{k_2v_5}$, $\overline{v_5v_6}$ and $\overline{k_1v_5}$ are added into *cEdges*. The updated *pList* and *cEdges* are used for constructing a new triangulation of R (Figure 6 (c)).

As shown in Figure 6 (c), the covered region of $\triangle_1$ can be a lot smaller than the coverage area of C_r'. In order to maximize the covered region by a k-NN search, *DCDT* propagates to (visits) the neighboring triangles of $\triangle_1$ (triangles that share an edge with $\triangle_1$). *DCDT* marks the covered regions starting from $\triangle_{max}$, and recursively visiting neighboring triangles ($findNeighbors()$ line 8-11 of Algorithm 3). Figure 7 shows an example of the propagation process. In Figure 7 (a), $\triangle_{max}$ is completely covered and its neighboring triangles $\triangle_{11}$, $\triangle_{12}$ and $\triangle_{13}$ (1-hop neighbors of $\triangle_{max}$) are partially covered in Figure 7 (b), (c) and (d), respectively. In Figure 7 (e), the neighboring triangles of $\triangle_{max}$'s 1-hop neighbors, i.e., 2-hop neighbors of $\triangle_{max}$, are partially covered. Finally, based on the returned result of *checkCoverRegion*() shown in Figure 7 (e), *DCDT* constructs a new triangulation as shown in Figure 7 (f). Algorithm 3 describes the propagation process.

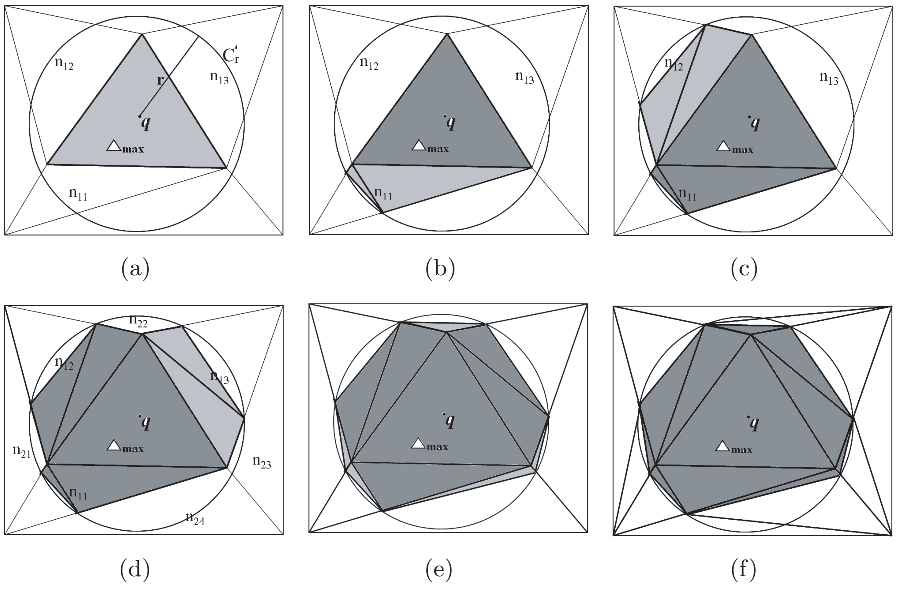


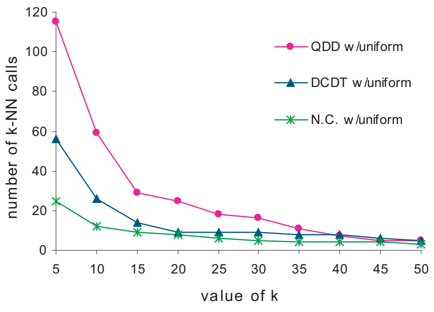
Fig. 7. Example of checkCoverRegion()

Notice that $DCDT$ covers a certain portion of R at every step, and therefore the covered region grows as the number of k-NN searches increases. On the contrary, QDD discards the returned result from a k-NN call when it does not cover the whole subregion, which results in re-visiting the subdivisions of the same region with 4 additional k-NN calls (see line 5-16 in Algorithm 1). Also note that the cost of triangulation is negligible as compared to k-NN search because triangulation is performed in memory and incrementally.

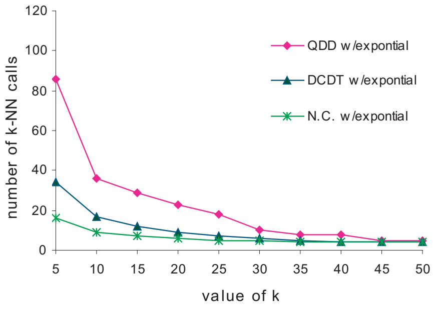
4 Experiments

In this section, we evaluate QDD and $DCDT$ using both synthetic and real GIS data sets. Our synthetic data sets use both uniform (random) and skewed (exponential) distributions. For the uniform data sets, (x, y) locations are distributed uniformly and independently between 0 and 1. The (x, y) locations for the skewed data sets are independently drawn from an exponential distribution with mean 0.3 and standard deviation 0.3. The number of points in each data set varies: 1K, 2K, 4K, and 8K points. Our real data set is from the U.S. Geological Survey in 2001: Geochemistry of consolidated sediments in Colorado in the US [13]. The data set contains 5,410 objects (points).

Our performance metric is the number of k-NN calls required for a given range query. In our experiments, we varied a range query size between 1% and 10% of the entire region of the data set. Different k values between 5 and 50 were used. Each size of range queries was conducted for 100 trials and the average values



(a) Uniformly distributed data set



(b) Exponentially distributed data set

Fig. 8. Number of k-NN calls for 4K synthetic data set with 3% range query

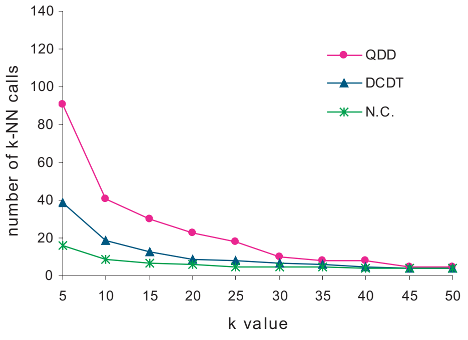


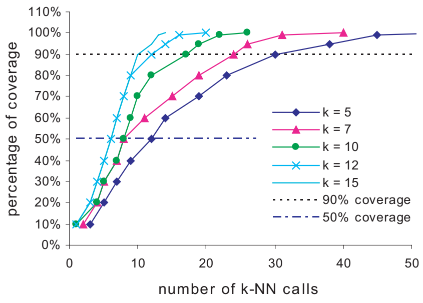
Fig. 9. Number of k-NN calls for real data set with 3% range query

were reported. We present only the most illustrative subset of our experimental results due to space limitation. Similar qualitative and quantitative trends were observed in all other experiments.

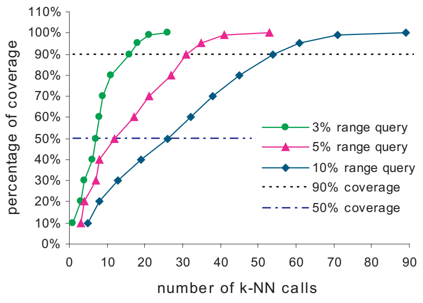
First, we compared the performance of QDD and $DCDT$ with the theoretical minimum, i.e., the necessary condition (N.C.) $\lceil \frac{n}{k} \rceil$, where n is the number of points in R. Figure 8 (a) and (b) show the comparisons of QDD, $DCDT$ and N.C. with uniformly distributed and exponentially distributed 4k synthetic data sets, respectively. For both $DCDT$ and QDD, the number of k-NN calls rapidly decreased as the value of k increased in the range 5-15, and 5-10, for the uniformly and exponentially distributed data, respectively. Then, it slowly decreased as the value of k became larger, approaching to those of N.C. when k is over 35. The results for the real data set have similar trends to those of the exponentially distributed synthetic data set (Figure 9). Note that k is determined not by the client but by the web interface and a typical k value in real applications ranges between 5 and 20 [3]. In both synthetic and real data sets, $DCDT$ needed a significantly smaller number of k-NN calls compared to QDD. For example, with

Table 2. The average percentage reduction in the number of k-NN calls (%)

data distribution	query size	k=5	10	15	20	25	30	35	40	45	50
uniform	3%	51.3	55.9	51.7	52.0	50.0	43.8	27.3	0	0	0
	5%	56.3	53.5	56.9	56.3	55.6	52.5	54.0	43.8	35.7	0
	10%	56.2	53.6	48.2	45.0	37.5	30.0	22.2	22.2	25.0	20.0
exponential	3%	60.5	58.8	58.6	56.9	51.1	40.0	37.5	37.5	20.0	20.0
	5%	68.4	67.2	68.8	65.7	60.7	58.3	50	41.7	25.0	20.0
	10%	69.4	59.7	56.5	56.7	55.8	34.4	46.9	46.4	35.0	31.3



(a) 3% range query with 4K synthetic data set



(b) 4K synthetic data set with k=10

Fig. 10. Percentage of coverage

the exponential data set (4K), *DCDT* showed 55.3% of average reduction in the required number of k-NN searches compared to *QDD*. As discussed in Section 3, *DCDT* covers a portion of the query region with every k-NN search while *QDD* revisits the same subregion when a k-NN search fails to cover the entire subregion. *DCDT* still required more k-NN calls than N.C. On the average, *DCDT* required 1.67 times more k-NN calls than N.C. However, the gap became very small when k was greater than 15.

Next, we conducted range queries with different sizes of query region: 3%, 5%, and 10% of the entire region of data set. Table 2 shows the average percentage reduction in the number of k-NN calls between *DCDT* and *QDD* for 4K uniformly and exponentially distributed synthetic data sets. On the average, *DCDT* resulted in 50.4% and 58.2% of reduction over *QDD* for the uniformly and exponentially distributed data set, respectively. The results show that the smaller the value of k is, the greater the reduction rate.

In Figure 10, we plotted the average coverage rate versus the number of k-NN calls required for *DCDT* on 4K synthetic uniformly distributed data set. Figure 10 (a) shows the average coverage rate of the *DCDT* algorithm while varying the value of k. For example, when k=7, 50% of the query range can be covered by first 8 k-NN calls while the entire range is covered by 40 k-NN calls. The same range query required 24 calls for 90% of coverage and this number is approximately 60% of the total required number of k-NN searches. Figure 10 (b) shows the average coverage rate of *DCDT* while varying query size. For 5% range query, 12 and 31 k-NN searches were required for 50% and 90% coverage, respectively.

5 Related Work

Some studies have discussed the problems of data integration and query processing on the web [4,5]. A query processing framework for web-based data integration was presented in [4]. It also implemented the evaluation of query planning and statistics gathering modules. The problem of querying web sites through limited query interfaces has been studied in the context of information mediation systems in [5].

Several studies have focused on performing spatial queries on the web [6,3,7]. In [6], the authors demonstrated an information integration application that allows users to retrieve information about theaters and restaurants from various U.S. cities, including an interactive map. Their system showed how to build applications rapidly from existing web data sources and integration tools. The problem of spatial coverage using only the k-NN interface was introduced in [3]. The paper provided a quad-tree based approach. A quad-tree data structure was used to check the complete coverage. Our work defines a more general problem of range queries on the web and provides two different solutions with detailed experiments and comparisons. The problem of supporting k-NN query using range queries was studied in [7]. The idea of successively increasing query range to find k points was described assuming statistical knowledge of the data set. This is the reverse of the problem we are focusing on.

The Delaunay triangulation and Voronoi diagram based approaches have been studied for various purposes in wireless sensor networks [8,9,10]. In [9], the authors used the Vornonoi diagram to discover the existence of coverage holes, assuming that each sensor knows the location of its neighbors. The authors in [10] proposed a wireless sensor network deployment method based on the Delaunay triangulation. This method was applied for planning the positions of sensors in an environment with obstacles. It retrieved the location information of obstacles and pre-deployed sensors, then constructed a Delaunay triangulation to find candidate positions of new sensors. *CDT* was dynamically updated while covering the query region with k-NN circles.

6 Conclusions

In this paper, we introduced the problem of evaluating spatial range queries on the web data by using only k-Nearest Neighbor searches. The problem of finding a minimum number of k-NN searches to cover the query range appears to be hard even if the locations of the objects (points) are known.

The Quad Drill Down (*QDD*) and Dynamic Constrained Delaunay Triangulation (*DCDT*) algorithms were proposed to achieve the completeness and efficiency requirements of spatial range queries. For both QDD and $DCDT$, we showed that they can cover the entire range even with the most restricted environment where the value of k is fixed and on which users have no control. The efficiencies of these two algorithms were compared each other as well as with the necessary condition. $DCDT$ provides a better performance than QDD.

We plan to extend our approaches for the cases in the presence of known data distribution and more flexible k-NN interfaces, for example, when users can change the value of k.

Acknowledgments

The following people provided helpful suggestions during this work: Dr. Petr Vojtěchovský and Brandon Haenlein.

References

1. Gaede, V., Günther, O.: Multidimensional access methods. ACM Computing Surveys 30(2), 170–231 (1998)
2. Song, Z., Roussopoulos, N.: k-Nearest Neighbor search for moving query point. In: Jensen, C.S., Schneider, M., Seeger, B., Tsotras, V.J. (eds.) SSTD 2001. LNCS, vol. 2121, pp. 79–96. Springer, Heidelberg (2001)
3. Byers, S., Freire, J., Silva, C.: Efficient acquisition of web data through restricted query interface. In: Poster Prceedings of the World Wide Web Conference, pp. 184–185 (2001)
4. Nie, Z., Kambhampati, S., Nambiar, U.: Effectively mining and using coverage and overlap statistics for data integration. IEEE Transactions on Knowledge and Data Engineering 17(5), 638–651 (2005)
5. Yerneni, R., Li, C., Garcia-Molina, H., Ullman, J.: Computing capabilities of mediators. In: Proceedings of SIGMOD, pp. 443–454 (1999)
6. Barish, G., Chen, Y., Dipasquo, D., Knoblock, C.A., Minton, S., Muslea, I., Shahabi, C.: Theaterloc: Using information integration technology to rapidly build virtual application. In: Proceedings of ICDE, pp. 681–682 (2000)
7. Liu, D., Lim, E., Ng, W.: Efficient k Nearest Neighbor queries on remote spatial databases using range estimation. In: Proceedings of SSDBM, pp. 121–130 (2002)
8. Sharifzadeh, M., Shahabi, C.: Utilizing Voronoi cells of location data streams for accurate computation of aggregate functions in sensor networks. GeoInformatica 10(1), 9–36 (2006)
9. Wang, G., Cao, G., Porta, T.L.: Movement-assisted sensor deployment. In: Proceedings of IEEE INFOCOM, pp. 2469–2479 (2003)
10. Wu, C., Lee, K., Chung, Y.: A Delaunay triangulation based method for wireless sensor network deployment. In: Proceedings of ICPADS, pp. 253–260 (2006)
11. Samet, H.: Data structures for Quadtree approximation and compression. Communications of the ACM 28(9), 973–993 (1985)
12. Chew, L.P.: Constrained Delaunay triangulations. Algorithmica 4(1), 97–108 (1989)
13. USGS mineral resources on-line spatial data (2001), `http://tin.er.usgs.gov/`