FINCH: Evaluating Reverse k-Nearest-Neighbor Queries on Location Data

Wei Wu[1] Fei Yang[2]
[1]Computer Science Programme
Singapore-MIT Alliance
National University of Singapore
wuw@nus.edu.sg

Chee-Yong Chan[2] Kian-Lee Tan[1,2]
[2]Department of Computer Science
School of Computing
National University of Singapore
{yangfei,chancy,tankl}@comp.nus.edu.sg

ABSTRACT

A Reverse k-Nearest-Neighbor (RkNN) query finds the objects that take the query object as one of their k nearest neighbors. In this paper we propose new solutions for evaluating RkNN queries and its variant bichromatic RkNN queries on 2-dimensional location data. We present an algorithm named `INCH` that can compute a RkNN query's search region (from which the query result candidates are drawn). In our RkNN evaluation algorithm called `FINCH`, the search region restricts the search space, and the search region is tightened each time a new result candidate is found. We also propose a method that enables us to apply any RkNN algorithm on bichromatic RkNN queries. With that, our `FINCH` algorithm is also used to evaluate bichromatic RkNN queries. Experiments show that our solutions are more efficient than existing algorithms.

1. INTRODUCTION

With the wide deployment of location sensing devices (such as GPS receivers), location based services are getting popular [9]. Location related queries play an important role in location based services. One such query type is the Reverse k-Nearest-Neighbor (RkNN) query that finds the objects (in the queried dataset) whose k nearest neighbors (NN) include the query point. As an example, a taxi can issue a RkNN query to find the passengers for which the taxi is one of his/her k nearest taxis.

RkNN queries are normally used to help discover the influence sets in a dataset [8]. In location dataset, the distance between two locations q and p gives a hint of q's influence on p (and vice versa). The shorter the distance is, the higher is the influence. In this sense, an object's k NN are the k objects that have the highest influence on it, and an object's reverse k NN are the objects that are highly influenced by it.

RkNN queries come in two flavors: monochromatic and bichromatic. Monochromatic RkNN query involves one type of data objects. For example, a RkNN query issued by a restaurant on a restaurant dataset (to find the restaurants that are influenced by the query restaurant) is a monochromatic RkNN query because both the query data and the queried data are of the same type. Bichromatic RkNN query involves datasets of two types. The previous query involving taxis and passengers is an example of a bichromatic RkNN query.

For monochromatic RkNN queries, researchers have proposed several algorithms. For bichromatic RkNN queries, to the best of our knowledge, there is only one algorithm for R1NN (i.e. k=1), and there is no reported solution that supports arbitrary k values. In this paper, we propose efficient solutions for both monochromatic and bichromatic RkNN queries on 2-dimensional location data where $k \geq 1$.

Our solutions are motivated by the following observation [15]: when k=1, the perpendicular bisector between the query point q and an arbitrary object point p can be used to prune objects in the half-plane that contains p; and when $k > 1$, a set of k such bisectors can also be used to prune certain objects (or R-tree nodes). In [15], Tao et al. exploited this observation to design the TPL scheme, which is the current state-of-the-art RkNN processing scheme. However, it is computationally expensive to find such a set of bisectors. Moreover, for objects that cannot be pruned, the TPL scheme spends considerable time to find out that such a set does not exist.

The key idea we have in our solutions is this: given a RkNN query and a set of data objects S, we can compute a region such that only the objects inside that region and the objects in S can be the query's result objects. We refer to this region as the query's search region. We design an algorithm called `INCH` (INtersections' Convex Hull) for computing the search region.

In our RkNN solution `FINCH` (Fast rknn processing using `INCH`), a query's search region is used to find the query's result candidates; at the same time, these candidates are used to further tighten the query's search region. This search for candidates finishes when the search region is small enough so that no new candidate can be found in the search region, which means that we have found all candidates. A refinement process is subsequently used to get the query's final result from the set of candidates.

We also propose a method in which any monochromatic RkNN algorithms can be applied to process bichromatic RkNN queries. We present the method, and use it to apply our `FINCH` algorithm on the evaluation of bichromatic RkNN queries.

We have conducted an extensive performance evaluation

VLDB '08, August 24-30, 2008, Auckland, New Zealand

of the algorithms. Our results show that the proposed methods are superior to the state-of-the-art technique.

The rest of the paper is structured as follows. In section 2 we give the formal definitions of RkNN queries and survey the related works. The basic INCH algorithm is presented in section 3. Then in sections 4 and 5 we describe the proposed solutions for RkNN and bichromatic RkNN queries. Experimental results are presented in section 6. Finally, section 7 concludes this paper.

2. BACKGROUND AND RELATED WORK

In this section, we introduce definitions and notations, and survey the algorithms that are closely related to RkNN queries.

2.1 Definitions and Notation

Given two points p and q, we use $\mathbf{dist(p,q)}$ to denote the Euclidean distance between p and q.

A **k-Nearest-Neighbors** query [10] kNN(q,k,P) on a dataset P finds the set of k objects that are nearest to the query location q. Formally, an object $p \in P$ is in the result of $kNN(q, k, P)$ if and only if it satisfies the following condition:

$$|\{o \in P | dist(o, q) < dist(p, q)\}| < k \qquad (1)$$

If multiple objects have the same k-th distance to the query object q, they will all be included in the kNN query's result set.

A (monochromatic) **Reverse k-Nearest-Neighbors** query [8] RkNN(q,k,P) on a dataset P finds objects in P whose k nearest neighbors include q. Formally:

$$RkNN(q, k, P) = \{p \in P | q \in kNN(p, k, P)\} \qquad (2)$$

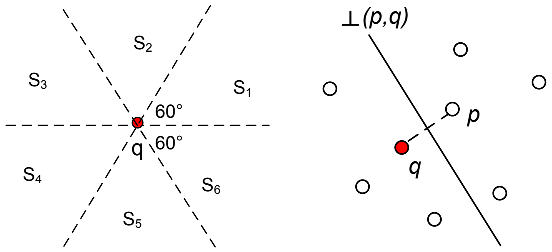
A **bichromatic Reverse k-Nearest-Neighbor** query [8] involves two types of data objects. Let P be the dataset of one type and R be the dataset of the other type. A bichromatic Reverse kNN query issued by a query object q from dataset P finds the objects in R whose kNNs on P includes q. Formally:

$$BRkNN(q, k, P, R) = \{r \in R | q \in kNN(r, k, P)\} \qquad (3)$$

A **Range-k** query [18] Range-k(p,k,r) checks whether there are fewer than k objects within the specified range r around a point p. That is, the query returns true if $|\{o|dist(p, o) < r\}| < k$; otherwise, it returns false. Our proposed solutions for RkNN queries makes use of Range-k queries.

2.2 Algorithms for kNN Queries

Depth-First search based and Best-First search based algorithms have been proposed for answering a kNN query on a R-tree [5] indexed dataset. In these algorithms, the minimum possible distance (min-distance) between the query location q and a R-tree node (or a data point) is used to determine whether the node should be visited. A node is visited if its min-distance to q is smaller than the current known k-th distance. In Depth-First search [10, 4], a stack is used to decide the order of visiting the nodes. In Best-First search [6], a priority queue based on the min-distances is used to decide the visit order.



(a) 60-degree-pruning (b) TPL pruning

Figure 1: Pruning techniques.

2.3 Algorithms for RkNN Queries

Various techniques have been proposed for RkNN queries in various settings [1, 2, 7, 8, 12, 13, 14, 15, 17, 18, 19, 20, 21].

Korn and Muthukrishnan [8] introduced the RNN query and its variants and proposed the RNN-Tree to facilitate the processing of such queries. RdNN-Tree [20] and MRkNNCop-Tree [1] are two other index structures proposed for RkNN processing. These methods rely on certain pre-processing (such as finding and storing each object's distance to its kth nearest neighbor) and incur storage overhead. Singh et al. [12] proposed an algorithm that finds the approximate solution for a RNN query. Yiu et al. studied the problem of RkNN in large graphs (road networks) [21]. [1, 17] proposed solutions for RkNN in metric spaces. [2, 7, 18, 19] studied the processing of continuous RkNN queries on moving objects. The solutions for continuous RkNN queries make use of snapshot RkNN algorithms and focus on the problem of updating a RkNN query's result when moving objects update their locations.

In this work, we are interested in algorithms that: find exact result for RkNN queries on location data, can be applied on general purpose index structures such as the R-tree, and do not rely on pre-processing. The algorithms proposed in [2, 13, 14, 15] fall in this category and are closely related to this work. We briefly introduce these algorithms in the remainder of this section. They all employ the filter-refinement framework. In the filter phase, most data objects are pruned, and the ones that are not pruned become the query's result candidates. In the refinement phase, each candidate is checked to see whether it is a query result.

2.3.1 Filter methods

Two interesting filter methods have been developed for RkNN processing.

One is the 60-degree-pruning method developed by Stanoi et al. in [13] for processing RNN queries (i.e. k=1). Its idea is illustrated in Figure 1(a): the space around a RNN query q can be divided into six equal-size regions (S_1, S_2, $\cdots$, S_6), then in each region only the nearest neighbor of q can possibly be a reverse nearest neighbor of q. By this, q's RNN candidates are restricted to q's nearest neighbor in each subspace. This 60-degree-pruning method can be extended to the case where $k \geq 1$: in each sub-space, only the RkNN query point's k nearest neighbors are result candidates. [2] used this extension to process RkNN queries on moving objects indexed with the TPR-tree [11]. This filter method results in at most 6*k candidates for each RkNN query.

The other is the TPL-pruning method proposed by Tao et al. in [15]. The basic idea of TPL is illustrated in Figure 1(b): the perpendicular bisector between the query point q and an arbitrary object point p divides the space into two half planes, i.e. the q-half-plane that contains q and the p-half-plane that contains p, then the points in the p-half-plane cannot be a RNN of q because p is closer to them than q is. Similarly, for RkNN, an object can be pruned if the object is in at least k such p-half-planes. Given a set S of m data objects, TPL filters as follows: for an R-tree node (or data object), TPL tries to find a subset of S with k objects that can prune the node (or data object) with their bisectors. Because the number of subsets $\binom{m}{k}$ can be very large, exhausting the subsets to prune the node is prohibitive. TPL sorts the objects in S with their Hilbert values, then subsets of consecutive objects are used to check the node. The objects that cannot be pruned are added to candidate set. Both analytical and experimental results in [15] show that TPL has a better pruning power than the 60-degree-pruning method.

Our algorithm also utilizes the property of bisectors. But our algorithm differs from TPL significantly in that we compute the query's search region with the bisectors, then the region (rather than the large number of bisectors) is used to find the query candidates. This saves a lot of computation because the test of whether an object (or an R-tree node) can be pruned can be done efficiently with a containment (or intersection) check[3].

2.3.2 *Refinement methods*

There are also two ways to check whether a candidate p is the RkNN query q's result.

One approach is the kNN based method used in [13, 2]. It first gets p's kNNs and then checks whether q is in it.

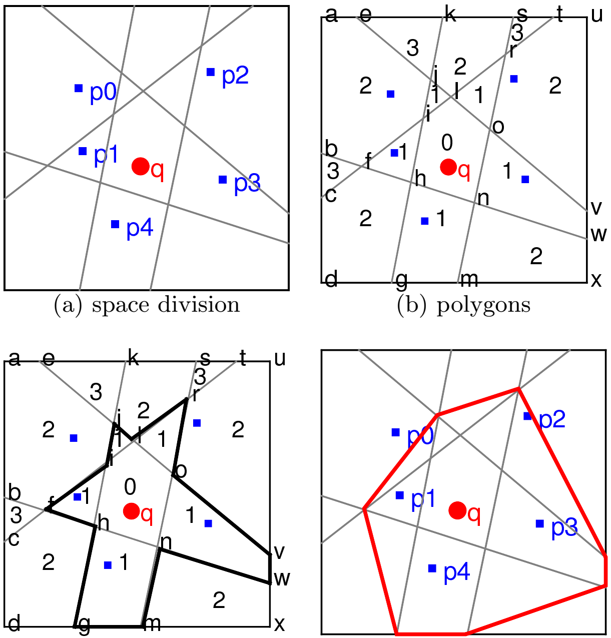
The other approach is the Range-k verification method introduced in [18]. The idea is to use a Range-k query (see section 2.1 for definition) to check whether there are fewer than k objects which have shorter distances to q than the candidate object does. Although [18] is a work that focuses on the *continuous* refinement problem in *continuous* RkNN monitoring, its result that Range-k refinement is more efficient than the kNN based refinement also applies in snapshot RkNN query processing.

2.4 Algorithm for bichromatic RkNN queries

The authors of [14] studied the problem of bichromatic R1NN queries, i.e. k=1. For a BRkNN(q,1,P,R), their solution is to compute q's Voronoi cell [3] using the R-tree on dataset P, then a range query with the Voronoi cell is used to retrieve the query result on the R-tree on dataset R. This solution employs the special property of BRkNN when k=1: the Voronoi cell of q on dataset P gives the exact result region. This solution is not applicable to the general BRkNN queries where $k \geq 1$.

3. INCH: COMPUTING A RKNN QUERY'S SEARCH REGION

Our proposed approach for evaluating RkNN queries is based on the filter-refinement paradigm which computes the query's result objects by first quickly pruning away a set of objects that do not satisfy the query to generate a set of candidate objects, followed by a more precise refinement of the candidates to eliminate false positives. In this section, we present our algorithm, named `INCH` (INtersections' Convex Hull), for generating a *search region*, which is a subspace of the data space. We will explain how this approach is applied to evaluate monochromatic and bichromatic RkNN queries in Sections 4 and 5, respectively.



(a) space division (b) polygons

(c) candidate region (d) search region

Figure 2: Space division and INCH.

The following notations will be used in the rest of this paper. We use P to denote the set of data objects. For a RkNN query, we use q to denote the query point and k to denote the k value of the RkNN query.

3.1 Plane Division and Candidate Region

The key idea of our filtering step is based on using a subset of data objects $S \subseteq P$ to define the query's *candidate region*, which is a subspace of the data space. For now, we will assume that S is a given input set of static data objects; and we will explain in the next section how S is incrementally generated (from an empty set) in the overall RkNN evaluation algorithm.

Given S, how can S be exploited to define a tight candidate region? Clearly, for any data object $p \in P$, if there exists a set of k objects $\{s_1, \cdots, s_k\} \subseteq S$ such that p is closer to each s_i than it is to q, then p is certainly not in the RkNN query's result and can be excluded from the candidate region.

We now explain how the above observation of using S to prune data objects from the candidate region can be generalized to prune regions of the data space to define a tight candidate region. Observe that for any data object $p \in S$, the perpendicular bisector (or bisector for short) of p and q, denoted by $\perp(p,q)$, divides the data space into two half planes such that (1) objects in the half plane of q are nearer to q than they are to p, and (2) objects in the half plane of p are nearer to p than they are to q. We use $\perp^p(p,q)$ to denote the half plane containing p and call it the *p-half-plane*, and $\perp^q(p,q)$ to denote the half plane containing q and call it the *q-half-plane*. More generally, we use the term $\overline{q}$*-half-plane* to refer to a p-half-plane that is induced by some data object $p \in S$ and the query point q.

Thus, each data object $p \in S$ introduces a bisector $\perp(p, q)$, and the entire set of $|S|$ bisectors actually divides the whole data space into a collection of polygons around the query point q. An example of this data space division is depicted in Figure 2(a), where the data space is partitioned into 14 polygons using $S = \{p0, p1, \cdots, p4\}$.

Note that the polygons constructed by the bisectors satisfy the following property: for each bisector $\perp(p, q)$, $p \in S$, each polygon is either completely inside $\perp^p(p, q)$ or completely inside $\perp^q(p, q)$. This important property provides a very simple way to characterize whether a polygon can be safely excluded from the query's candidate region: if a polygon R is inside at least k $\overline{q}$-half-planes, then for every data object $p \in P - S$ that is within R or on the boundary of R, p is not in the RkNN query's result. The fact that R is inside at least k $\overline{q}$-half-planes means that there are at least k objects in S that are closer to p compared to q.

Given a RkNN query (specified by the parameters q and k), we define the query's **candidate region** (with respect to $S \subseteq P$) as the set of polygons constructed by the bisectors induced by S that are inside at most $k - 1$ $\overline{q}$-half-planes.

Referring once more to the example in Figure 2(a), if we label each polygon in the data space division with the number of $\overline{q}$-half-planes that the polygon is in, we obtain Figure 2(b). Consider the top polygon that is defined by the points e, j, and k. This polygon is labeled 3 because the polygon is within three $\overline{q}$-half-planes induced by S: $p0$-half-plane, $p1$-half-plane, and $p2$-half-plane. Notice that there is exactly one polygon that is labeled 0, which is the polygon containing q. Suppose q is the query point for a RkNN query with $k = 2$. Then any data object $p \in P - S$ that is within or on a polygon with a label greater than 1 is certainly not in the query's result; and the query's candidate region is the collection of 6 polygons with label values of 0 or 1 as highlighted in Figure 2(c).

An important point to emphasize is that our definition of candidate region only allows for the pruning of data objects in $P - S$, but not for the objects in S. As an example, consider again the candidate region shown in Figure 2(c), where $k = 2$. Note that although the data object $p0 \in S$ is outside of the candidate region, $p0$ could potentially be in the query's result (e.g., if $P = S$, then $p0$ is in the query's result); therefore, $p0$ should not be pruned based on the candidate region. Data objects in S need to be treated differently from those in $P - S$ because each $s \in S$ is used to generate a bisector for the division of the data space; and consequently, s is obviously within its own s-half-plane. Thus, a stronger condition is actually necessary to prune objects in S; specifically, for each $s \in S$, if s is inside at least $k+1$ $\overline{q}$-half-planes, then s is certainly not in the query's result and can be pruned.

For a RkNN query, we define the **query result candidates** (with respect to S) as consisting of both the data objects in S as well as the data objects in $P - S$ that are within the query's candidate region.

3.2 Search Region: Approximation of Candidate Region

To find the query result candidates in $P - S$ requires identifying all the polygons that define the query's candidate region and checking the objects in $P - S$ against this collection of polygons. Since the number of such polygons can be large, using the candidate region to generate query result candidates from $P - S$ can be inefficient.

In this section, we present a simple but efficient way to approximate the query's candidate region using the smallest convex hull that contains the candidate region. We refer to this approximation as the **search region** of the RkNN query. As illustrated by Figure 2(c) for $k = 2$, the polygons that made up the candidate region for the query are all connected and are clustered around the query point q. Thus, the search region formed using the smallest convex hull that contains the candidate region (shown in Figure 2(d)) generally provides a rather tight approximation of the candidate region.

The key advantage of filtering the data objects in $P - S$ using the search region (instead of the candidate region) is the efficiency of filtering using only a single polygon (instead of many smaller polygons). We next present the algorithm for computing the search region.

Algorithm 1: INCH(q,k,S)

```
input : q: query location
        k: value of $k$ of the RkNN query
        $S \subseteq P$: a subset of data objects
output: the query's search region in the form of a
        convex hull
1 compute bisector for each data point $p$ in S
2 compute the intersections
3 compute the level of each intersection
4 I:= the set of intersections with level smaller than $k$
5 compute and return the convex hull with intersections
  in I
```

3.3 The INCH Algorithm

Our algorithm, called `INCH`, to compute the search region for a RkNN query is shown as Algorithm 1. The algorithm takes as inputs the query location point q, the k value of the RkNN query, and a subset of data objects $S \subseteq P$. It returns a convex hull as the query's search region.

Step 1 computes all the bisectors induced by S. Step 2 then computes all the intersection points formed by the bisectors and the data space boundaries. More specifically, an intersection point is formed by the intersection of two bisectors, or the intersection of a bisector and a data space boundary, or the intersection of two data space boundaries. As an example, each intersection point in Figure 2(b) is labeled with a letter from $\{a, \cdots, o, r, \cdots, x\}$.

Step 3 then computes the *level* of each intersection point. The **level** of a point x (with respect to S), denoted by $level(x)$, is defined to be the number of $\overline{q}$-half-planes that x is in, excluding the $\overline{q}$-half-planes that are due to the bisectors that x intersects (if any). For example, in Figure 2(b), the level of point h is 0; the level of point f is 1 because it is only in $\perp^{p1}(p1, q)$; and the level of point d is 2 because it is only in $\perp^{p1}(p1, q)$ and $\perp^{p4}(p4, q)$.

Step 4 then identifies the set of intersections I that have levels smaller than k; and the search region is the convex hull formed using the points in I. Note that if the number of objects in S is smaller than k, then the search region returned is the entire data space.

The complexity of the `INCH` algorithm is $O(m^3)$, where

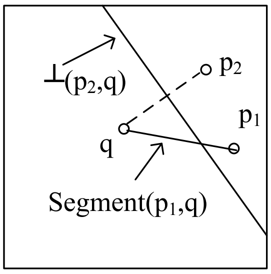
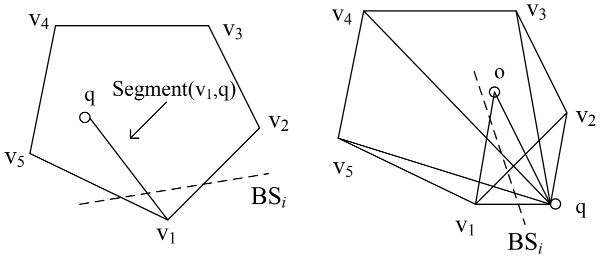


Figure 3: Segment(p_1,q) intersects with $\perp(p_2, q)$.



(a) q is within polygon (b) q is outside of polygon

Figure 4: Illustration for Lemma 1.

m is the number of objects in S. The cost of computing the bisectors (step 1) is $O(m)$. The cost of computing the intersections (step 2) is $O(m^2)$. The cost of computing all intersections' levels is $O(m^3)$ because there are $O(m^2)$ intersections and computing an intersection's level take $O(m)$ time; note that testing whether an intersection point is in a $\overline{q}$-half-plane can be performed in constant time [3]. The size of set I (step 4) is at most $O(m^2)$. The complexity of computing the convex hull is $O(n \log(n))$ for an input size of n; therefore, the cost of step 5 is at most $O(m^2 \log(m))$.

3.3.1 Proof of Correctness

In this section, we establish the correctness of the INCH algorithm. Given two points p_1 and p_2, we use **Segment(p_1, p_2)** to denote the line segment between p_1 and p_2 that excludes the end points p_1 and p_2.

Consider a bisector $\perp(p_2, q)$, where $p_2 \in S$. A point p_1 is inside $\perp^{p_2}(p_2, q)$ if and only if p_1 and q are on two different sides of the bisector $\perp(p_2, q)$. Since q and p_2 are on different sides of $\perp(p_2, q)$, the Segment(p_1,q) must intersect with $\perp(p_2, q)$, as illustrated in Figure 3. Therefore, the level of a point p can also be defined as the number of bisectors that intersects Segment(p,q). That is,

$$level(p) = |\{ \text{ bisectors that intersect with } Segment(p, q)\}|.$$

The following result states the relationship between the levels of points within a polygon and the levels of the polygon's vertices.

Lemma 1. *For any point p within a polygon and for any vertex v of that polygon, the level of p is at least as large as the level of v.*

Proof. The proof is established by contradiction. Consider a polygon R and a query point q. There are two cases to consider depending on whether q is within or outside of R, as illustrated in Figure 4.

Case 1: q is within R. In this case, it is necessary that the the level of any vertex of R must be 0. To see this, suppose that R has a vertex v_1 whose level is not zero. This implies that there must exist at least one bisector BS that intersects with Segment(v_1,q). It follows that BS must also intersect with some edge of R thereby contradicting the fact that R is a polygon formed by the intersection of bisectors.

Case 2: q is outside of R. Let o be a point inside the polygon, and let v_1 be a vertex of R. Suppose that the level of v_1 is larger than the level of o. This implies that there must exist a bisector BS_i that intersects with Segment(v_i,q) but does not intersect with Segment(o,q). Moreover, since the set of three points $\{v_1, o, q\}$ forms a triangle, it is necessary that BS_i intersects with Segment(v_1,o), which means that BS_i intersects with some edge of R. This again contradicts the fact that R is a polygon formed by the intersection of bisectors. □

To establish the correctness of INCH, we need to show that the search region constructed by INCH does not result in any false negatives in $P - S$ as stated by the following result.

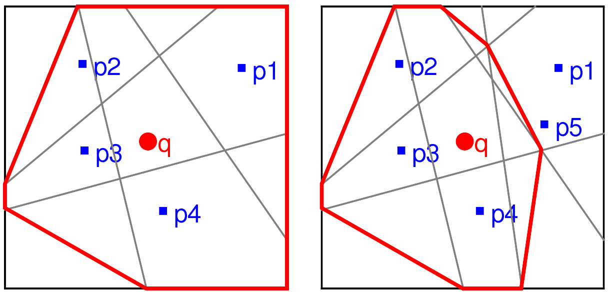
Theorem 2. *Given a data set P, a RkNN query RkNN(q,k), and a set of data objects $S \subseteq P$, let C be the search region (convex hull) constructed by INCH(q,k,S). For each point p in $(P - S)$, level(p) $< k$ iff p is inside or on C.*

Proof. Suppose that there exists a point o that is outside of the convex hull C, where the level of o is smaller than k. There are two cases to consider depending on whether o is within a polygon. Consider the case where o is within a polygon R. By Lemma 1, the level of each vertex of R must also be smaller than k which means that all the vertices of R would have been included in I (step 4 of INCH algorithm) which contradicts the assumption that o is outside of C. Consider the case where o is not within a polygon; i.e., o is on some polygon's edge. Let p be a point on Segment(q,o) where p is outside of C and within some polygon. Since Segment(q,p) is part of Segment(q,o), therefore any bisector that intersects Segment(q,p) must also intersect Segment(q,o). Thus, $level(p) \leq level(o)$ which implies $level(p) < k$. This leads to a contradiction as in the first case. □

4. FINCH: EVALUATING RKNN QUERIES

In the previous section, we presented the INCH algorithm that for a given RkNN query and an input set of data objects $S \subseteq P$, computes a search region SR for the query. Based on S and SR, the query result candidates generated is given by $S \cup S'$, where S' is the set of data objects in $P - S$ that are within the search region SR. Clearly, the choice of S affects the number of query result candidates and the cost of the subsequent candidate refinement step to eliminate false positives from $S \cup S'$. Consider the following two extreme options for S: $S = \emptyset$ and $S = P$. If $S = \emptyset$, then SR becomes the entire data space and $S' = P$ which will incur a costly refinement step to verify each object in P. On the other hand, if $S = P$, then S' is empty and it will incur a costly computation of the minimized SR using the entire set of data objects. Neither of these two options is desirable.

Thus, there are two important issues to be addressed. First, how is S selected to optimize both the cost of computing the query result candidates as well as its size. Second,



(a) $S_1 = \{p1, p2, p3, p4\}$ (b) $S_2 = S_1 \cup \{p5\}$

Figure 5: Incremental refinement of search region.

how can the query result candidates be efficiently refined to compute the query result objects.

In this section, we present our overall approach, called `FINCH`, for evaluating RkNN queries. This approach, which is based on the filter-refinement paradigm, consists of two main algorithms. The `FINCH-Filter` algorithm (which builds on the `INCH` algorithm) computes a set of query result candidates from P, which are then refined by the `FINCH-Refine` algorithm to compute the query result objects.

4.1 Basic FINCH-Filter Algorithm

Our basic `FINCH-Filter` algorithm, which is shown as Algorithm 2, is an iterative approach that starts with S as an empty set and incrementally adds data objects to S and refines the search region. This algorithm is based on the following property of the search region computed by the `INCH` algorithm: if $S_1 \subseteq S_2 \subseteq P$, SR_1 = INCH(q,k,S_1), and SR_2 = INCH(q,k,S_2), then SR_2 is contained in (or equal to) SR_1. This search region refinement property is illustrated by the example in Figure 5: the search region that is induced by a set S_1 of four data objects (Figure 5(a)) is further refined into a smaller search region (Figure 5(b)) when an additional data object p_5 is added to S_1 to form S_2.

```
Algorithm 2: FINCH-Filter(q,k)
  input : q: query location
          k: value of k of the RkNN query
  output: a set of objects containing the query's result
          objects
1 S := empty set
2 SR := entire data space
3 while an object $p \in P - S$ can be found in SR do
4     $S := S \cup \{p\}$
5     SR:=INCH($q$,$k$,$S$)
6 return S
```

The `FINCH-Filter` algorithm begins by initializing S to an empty set (step 1) and the search region SR to be the entire data space (step 2). If a data object $p \in P - S$ is found to be contained in SR (step 3), then p is added to S (step 4). Step 5 then uses the expanded S to refine the search region by invoking the `INCH` algorithm. The algorithm iteratively expands S and refines SR until SR does not contain any object in $P - S$. The set S is then returned as the set of query result candidates.

Note that in the `FINCH-Filter` algorithm as presented, the algorithm terminates only when S', which is the set of objects in $P - S$ that are contained in the search region SR, becomes empty. In this way, the set of query result candidates computed is simply S. Clearly, it is possible to terminate the `FINCH-Filter` algorithm earlier before S' becomes empty (without affecting correctness) and return $S \cup S'$ as the set of query result candidates. We have adopted the variant presented as this produces a smaller set of candidates.

An important optimization issue for the `FINCH-Filter` algorithm is the selection of the next object to add to S at each iteration. This choice affects both the shape and size of the search region (which is formed by the intersection points caused by the bisectors introduced by S) as well as the number of iterations of the while-loop in the `FINCH-Filter` algorithm which determines the the total number of query result candidates computed and hence the computation cost of the candidate refinement step.

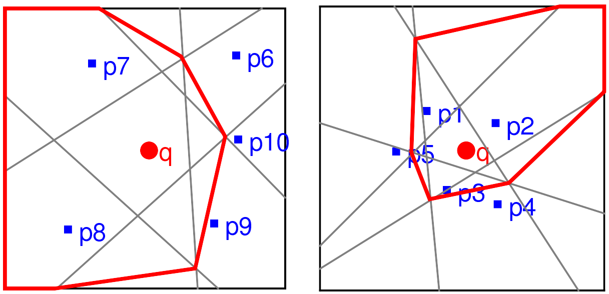
Referring to the example data space partitioning in Figure 2(b), observe that the polygons with smaller label values are closer to q and they are less likely to be pruned as the search region is refined. A good heuristic to optimize the selection of objects to be inserted into S is to give higher priority to objects that are closer to q. Adding to S an object that is closer to q tends to produce a tighter refinement of the search region than adding another object that is further away from q. Figure 6 compares the effect on the search region tightening (with the convex hulls shown by bold lines) when adding five objects to S that are closer to or further from q. By obtaining a tighter search region earlier, the number of candidates produced can also be reduced as illustrated in Figure 6(a): observe that had an object that is closer to q been added earlier to S (before the further objects p7, p8, and p9), the resultant tighter search region would have excluded these three further objects from being added into S. In section 4.2, we explain how this heuristic can be efficiently realized with the use of an R-tree index.

Since the search region is iteratively refined (by calling the `INCH` algorithm) as objects are progressively added to S, another optimization opportunity is to improve the efficiency of the `INCH` algorithm by incrementally maintaining the search region and associated information (e.g., bisectors, intersections and their level values) and reducing its computation cost. In section 4.4, we present two optimizations to improve the performance of the `INCH` algorithm.

4.2 FINCH-Filter with R-tree Index

In this section, we present an efficient realization of our general `FINCH-Filter` algorithm to optimize the iterative selection of data objects to add to S (step 3 of Algorithm 2). As explained in the previous section, a good heuristic for expanding S is to give higher priority to data objects that are closer to q. By indexing the data objects in P using an R-tree index [5] and following a best-first traversal of the index [6], the sequence of data objects added to S will be ordered in non-descending distance from q. We refer to this variant of the `FINCH-Filter` algorithm as the `FINCH-Filter-Rtree` algorithm, which is shown as Algorithm 3.

To support best-first traversal of the R-tree index, a min-heap H is maintained that contains of entires of the form (p, key). Here, p is either an R-tree index node (internal or leaf node) or a data object that is contained in some leaf node; and key is the minimum Euclidean distance between q and p. H is initialized with a single entry corresponding



(a) add five far objects (b) add five near objects

Figure 6: Comparison of search region refinement.

```
Algorithm 3: FINCH-Filter-Rtree(q,k)
 input : $q$: query location
         $k$: value of $k$ of the RkNN query
 output: a set of objects containing the query's result
         objects
 1 S := empty set
 2 SR := entire data space
 3 Let H be a min-heap containing entries of form (e, key)
 4 insert a single entry (R-tree root node, 0) into $H$
 5 while $H$ is not empty do
 6     remove entry (p,key) from H
 7     if $p$.MBR intersects with SR then
 8         if $p$ is an index node then
 9             for each child node $c$ in $p$ do
10                 if $c$.MBR intersects SR then
11                     insert entry ($c$, min-dist($c$,$q$)) into $H$
12         else if $p$ is a leaf node then
13             foreach data point $o$ in $p$ do
14                 if $o$ is inside SR then
15                     insert entry ($o$, dist($o$,$q$)) into $H$
16         else                    /* p is a data object */
17             $S := S \cup \{p\}$
18             SR := INCH($q$,$k$,$S$)
19 return S
```

to the R-tree index's root node (step 4), and the traversal of the R-tree index always selects the entry from H whose node/object is closest to q (step 6). Whenever a data object p is selected from H, p is inserted into S (step 17) and the search region is refined by calling the `INCH` algorithm (step 18).

4.3 FINCH-Refine Algorithm

In this section, we present our candidate refinement approach, the `FINCH-Refine` algorithm (shown as Algorithm 4), that eliminates false positives from an input set S of query candidate results.

The candidate refinement is carried out in two stages. The purpose of the first stage (steps 1 to 4) is to apply an efficient (but approximate) procedure to further eliminate some false positives from S. Any remaining false positives in S are then eliminated using a more costly (but precise) procedure in the second stage (steps 5 to 7).

In the first stage, we apply one more round of the `INCH` algorithm (with parameter values q, k+1, and S) to compute a new search region to eliminate certain false positives from S. Note that we are now using a value of $k + 1$ and not k to compute the new search region. To understand why this makes sense, recall that at the time when an object p is selected for insertion into S (let us refer to this old version of S as S_{old}), p is within the search region that is computed wrt S_{old}. This implies that p is inside at most $k-1$ $\overline{q}$-half-planes (wrt S_{old}). Subsequently, S_{old} is expanded with more objects inserted into it to eventually become the current version S. Clearly, it is possible for p to be now inside more than $k-1$ $\overline{q}$-half-planes (wrt S) and is therefore a false positive. However, since p is certainly inside its own p-half-plane, we now require p to be inside at least $k + 1$ $\overline{q}$-half-planes (wrt S) in order to conclude that p is a false positive; the condition on $k+1$ is necessary to basically take into account of the $\overline{q}$-half-plane contributed by p.

```
Algorithm 4: FINCH-Refine(q,k,S)
 input : $q$: query location
         $k$: value of $k$ of the RkNN query
         $S$: result candidates
 output: RkNN query's result
1 SR := INCH($q$,$k+1$,$S$)
2 foreach data object $p$ in $S$ do
3     if $p$ is not inside SR then
4         $S := S - \{p\}$
5 foreach data object $p$ in $S$ do
6     if Range-k($p$,$k$,dist($p$,$q$)) returns false then
7         $S := S - \{p\}$
8 return S
```

Since the remaining objects in S after the first stage may still contain false positives, in the second stage, a more precise refinement is performed using the Range-k verification method [18]. An object that is not eliminated by this Range-k check is in the result of the RkNN query.

Note that when the the size of the input set of query result candidates S is small, all the objects in S are main-memory resident and the first stage does not incur any disk I/O. In contrast, the Range-k verification procedure used in the second stage requires access to the R-tree index.

4.4 Optimizations for INCH

Recall that in the `FINCH-Filter` algorithm, a new search region needs to be computed (by calling the `INCH` algorithm) each time a new object is added into S. In this section, we present two optimizations for the `INCH` algorithm.

4.4.1 Incremental Maintenance of Search Region

Instead of re-computing the search region from scratch each time a new data object is added to S, a simple optimization is to incrementally maintain and update the information about the bisectors, intersection points and their level values. This optimization procedure, called `Optimized-Add`, is performed when adding a new data object p to S. The details of this optimization are shown as Algorithm 5. Thus, instead of performing "$S := S \cup \{p\}$" (step 4 in Algorithm 2 and step 17 in Algorithm 3), a call is made to "Optimized-Add(q, p, S)".

Algorithm `Optimized-Add` performs four main tasks: (1) computes the new bisector between p and q (step 3); (2) increments the level of each existing intersection point if the

Algorithm 5: Optimized-Add(q,p,S)

```
input  : $q$: query point
         $p \in P - S$: a new data object
         $S \subseteq P$: result candidates
1  Let SG be the set of data space boundaries & existing
   bisectors
2  Let I be the set of existing intersections
3  $bs_{new}$ := perpendicular bisector of $p$ and $q$
4  foreach intersection $it_i$ in $I$ do
5      if $it_i$ is in the $p$-half-plane of $bs_{new}$ then
6          increment $it_i$'s level value by one
7  $I_{new}$ := empty set
8  foreach segment $sg$ in $SG$ do
9      if $bs_{new}$ intersects with $sg$ in the data space then
10         add their new intersection point to $I_{new}$
11 compute the level of each intersection in $I_{new}$
12 $SG := SG \cup \{bs_{new}\}$
13 $I := I \cup I_{new}$
14 $S := S \cup \{p\}$
```

point is inside the new p-half-plane (steps 4-6); (3) computes new intersection points (steps 8-10); and (4) computes the levels of any new intersection points (step 11).

The complexity of the `Optimized-Add` algorithm is $O(m^2)$, where m is the number of data space boundaries and existing bisectors. The first for-loop has a cost of $O(m^2)$; and the second for-loop has a cost of $O(m)$. The cost of computing the new intersections' levels (step 11) is $O(m^2)$. Thus, this optimization reduces the cost of computing intersections and their levels from $O(m^3)$ (Section 3.3) to $O(m^2)$.

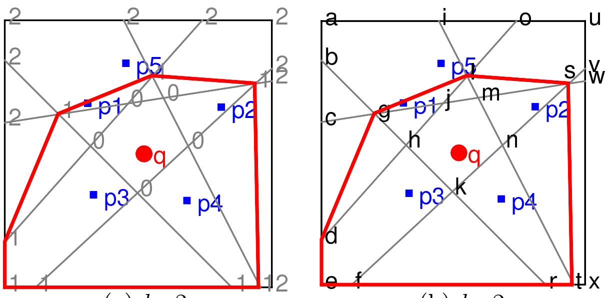
4.4.2 *Optimized Search Region Computation*

Our second optimization is to improve the efficiency of the search region (i.e., convex hull) computation.

First, since the levels of intersection points do not decrease as more points are added into S, a simple optimization is to maintain only the intersection points with level values of at most $k-1$. Second, although there could be many intersection points on a segment (i.e., data space boundary or bisector), only the two "extreme" intersection points (with level values smaller than k) on a segment can potentially become vertices of the convex hull. This observation can reduce the number of intersection points involved for the search region computation from $O(m^2)$ to $O(m)$, where m is the number of bisectors.

As an example, consider the search region (for $k = 2$) shown in Figure 7, where the convex hull is shown with bold lines. Each intersection point is labeled with its level value in Figure 7(a), and is identified with a unique letter identifier in Figure 7(b). Observe that although there are four intersection points (g, j, m, and s) with level values smaller than k on the bisector $\perp(p5, q)$, only the two extreme intersection points, g and s, are vertices of the convex hull.

Thus, by maintaining a sorted list of intersection points (with level values smaller than k and sorted on their location coordinates) for each segment (i.e., data space boundary or bisector), locating the two extreme intersection points on a segment can be perfomed in constant time. Algorithm 6 outlines this optimized `INCH` computation, which we refer to as the `Optimized-INCH` algorithm.



(a) k=2 (b) k=2

Figure 7: Optimized Search Region Computation.

Algorithm 6: Optimized-INCH (q,k,S)

```
input  : q: query point
         k: value of k in RkNN query
         $S \subseteq P$: a subset of data objects
output: the query's search region
$I_{ch}$:= empty set        /* set of intersections for
convex hull computation */
foreach data space boundary or bisector $sg$ do
    Let $I_{left}$ be the leftmost intersection point on $sg$
    with level < $k$
    Let $I_{right}$ be the rightmost intersection point on $sg$
    with level < $k$
    $I_{ch} := I_{ch} \cup \{I_{left}, I_{right}\}$
compute and return the convex hull using $I_{ch}$
```

4.5 Discussions

Note that the two optimizations presented are independent, and one can be applied without applying the other. However, applying only the second optimization (i.e., search region computation) alone without the first optimization (i.e, incremental maintenance) is not very beneficial. This is because the second optimization requires maintaining sorted lists of intersection points and if this is not done incrementally (together with the first optimization), the cost of sorting will be $O(m^2 \log(m))$, where m is the number of segments. This is because there are m sorted lists, each of which has $O(m)$ points; and sorting a list of m elements takes $O(m\log(m))$. Thus, although the second optimization reduces the cost of convex hull computation from $O(m^2\log(m))$ to $O(m\log(m))$, the gain is offset by the sorting cost.

Overall, the first optimization has a significant benefit, and the second optimization should be applied together with the first optimization (to maintain the sorted lists incrementally). The two optimizations together can reduce the cost of search region computation by a factor of k.

5. FINCH-B: EVALUATING BICHROMATIC RKNN QUERIES

In this section, we first describe how our `FINCH` algorithm, which is designed for evaluating RkNN queries, can be extended to a new algorithm, termed `FINCH-B`, for evaluating bichromatic RkNN (or BRkNN) queries (Section 5.1). We then generalize the underlying principle behind this extension to present a general framework for evaluating BRkNN queries using any RkNN approach that is based on the filter-refinement paradigm (Section 5.2).

5.1 Applying FINCH for BRkNN Queries

In contrast to RkNN queries, BRkNN queries involves two sets of data objects, P and R, where the query point q is from P and the result objects are from R. Specifically, $r \in R$ is a result object of the BRkNN query (wrt $q \in P$) if q is one of k objects from P that are the k nearest neighbors of r.

Algorithm 7: FINCH-B (q,k,P,R)

```
input : q ∈ P: query object
        k: value of k of the BRkNN query
        P: dataset that contains q
        R: dataset containing result objects
output: result objects of the BRkNN query
1 S := FINCH-Filter (q,k) on dataset P
2 SR := INCH(q,k,S)
3 R' := subset of objects from R that are contained in
  region SR
4 T: = empty set
5 foreach object r in R' do
6     if Range-k(r,k,dist(q,r)) on P returns true then
7         T := T ∪ {r}
8 return T
```

Our FINCH-B approach of evaluating BRkNN queries is shown as Algorithm 7. Step 1 applies the FINCH-Filter algorithm to compute a set of candidate objects S (from P) for the RkNN query (wrt q and P). Step 2 then computes the search region SR (wrt S) using the INCH algorithm.

Note that while the candidate objects in S are specific to the objects from P (i.e., S can only be used to answer a specific type of RkNN query (wrt q and P) which can be viewed as a special BRkNN query with $R = P$), the search region SR derived from S is more "general" in the following sense. Consider an object $x \in X$, where X is some dataset. If x is not contained in SR, then it can be concluded that the k nearest neighbors (from P) of x certainly does not include q; in particular, if $X = R$, then x is not in the result of the BRkNN query (wrt q, P, and R).

The above property of SR is exploited in step 3 of FINCH-B to compute a set of candidate objects $R' \subseteq R$ for the input BRkNN query. Finally, steps 5 to 7 apply the Range-k verification procedure to eliminate false positives from R'.

5.2 A General Framework for Evaluating BRkNN Queries

Based on the ideas behind our extension of FINCH to FINCH-B for evaluating BRkNN queries, we now present a general framework for evaluating BRkNN queries using any RkNN approach that is based on the filter-refinement paradigm.

Consider a RkNN approach called $\mathcal{A}$ that is based on the filter-refinement paradigm. Let $\mathcal{A}$-Filter and $\mathcal{A}$-Refine denote, respectively, the filter and refinement algorithms of $\mathcal{A}$. A general approach to apply $\mathcal{A}$ to process BRkNN queries is as follows:

1. Apply $\mathcal{A}$-Filter to compute a set of candidate objects $P' \subseteq P$ for the RkNN query (wrt q and P).

2. Based on $\mathcal{A}$-Filter and P', derive a set of candidate objects $R' \subseteq R$ for the input BRkNN query.

3. Apply $\mathcal{A}$-Refine to eliminate false positives from R' using P for the input BRkNN query.

Clearly, the FINCH-B algorithm described in the previous section is an instantiation of this framework.

5.2.1 Extending TPL to TPL-B

Let us now apply the above framework to the TPL approach [15], which is designed specifically for RkNN queries, to create an extended approach, which we refer to as TPL-B, for evaluating BRkNN queries. The details of TPL-B are shown as Algorithm 8.

Algorithm 8: TPL-B (q,k,P,R)

```
input : q ∈ P: query object
        k: value of k of the BRkNN query
        P: dataset that contains q
        R: dataset containing result objects
output: result objects of the BRkNN query
 1 P':= empty set
 2 foreach object p in P do
 3     if k-trim(q,k,P',p) returns ∞ then
 4         P' := P' ∪ {p}
 5 R' := empty set
 6 foreach object r in R do
 7     if k-trim(q,k,P',r) returns ∞ then
 8         R' := R' ∪ {r}
 9 Eliminate false positives from R' using P
10 return R'
```

The filtering function in TPL is called k-trim(q,k,S,p), which is used to determine whether a data object p can be pruned with respect to a set of objects S for a RkNN query; specifically, k-trim(q,k,S,p) returns ∞ iff p cannot be pruned.

In TPL-B, the function k-trim is first used in step 3 to compute a set of candidate objects $P' \subseteq P$ for the RkNN query (wrt q and P). Next, step 7 then applies k-trim (wrt P') to generate a set of candidate objects $R' \subseteq R$ for the input BRkNN query. Finally, step 9 eliminates the false positives from R' using P.

The next section will experimentally compare FINCH-B against TPL-B for evaluating BRkNN queries.

5.2.2 Other Extensions

As a final example of applying our proposed framework, we briefly explain how the 60-degree-pruning method (described in section 2.3.1) can also be extended to evaluate BRkNN queries. In the first step, the set of candidate objects $P' \subseteq P$ for the RkNN query (wrt q and P) is simply q's six sets of k nearest neighbors in the six sub-spaces. In the second step, the set of candidate objects $R' \subseteq R$ for the input BRkNN query is determined by considering each of the six sub-spaces on R, and using the corresponding k-th distance in P' to find all the candidate objects in each subspace. The final third step applies a refinement algorithm to eliminate false positives from R' using P.

6. PERFORMANCE EVALUATION

In this section we study the performance of our proposed algorithms, namely the FINCH algorithm for RkNN queries and the FINCH-B algorithm for bichromatic RkNN queries.

We compare their performance with the TPL[1] algorithm proposed in [15], which is the state-of-the-art RkNN algorithm available for location dataset. [15] shows that it outperforms all other RkNN algorithms that are designed for location dataset. The algorithms are implemented in Java. Experiments are run on a Linux desktop machine with a 2.4GHz CPU and 512M memory.

Two datasets are used in the experiments. We download the TIGER/Line files of Los Angeles (LA) and California state (CA) from the website of U.S. Census Bureau[2], and convert them (line datasets) into point datasets by generating a set of points on each line in the original data files. The datasets are indexed with the R*-Tree with page size set to 4096 bytes. The details of the datasets are provided in Table 1. A 10-page random eviction buffer is used in our program. The buffer size is far smaller than the dataset sizes.

Table 1: Datasets detail

Dataset	Num of objects	R*-Tree size
LA	about 550,000	about 40M
CA	about 25,000,000	about 1.4G

Real running time is used as performance metric. The running time includes CPU time and I/O time. The times shown in the figures are the total running time for 100 queries (with query locations randomly picked in the queried data space). During the experiments, the following information is noted: disk I/O time in filter phase, disk I/O time in refinement phase, total time of filter, total time of refinement.

6.1 Effect of Optimizations on INCH

We here study the effects of our two optimizations for the `INCH` algorithm. Figure 8 shows their effects on `FINCH`. In the figure, line "None" is `FINCH` (using `INCH`) without optimizations, line "Opt1" is `FINCH` with `Optimized-Add` (incremental computing of intersections, section 4.4.1), line "Opt2" is `FINCH` with `Optimized-INCH` (reducing the number of intersections involved in convex hull computation, section 4.4.2), and line "`FINCH`" is `FINCH` with both optimizations. We see that `Optimized-Add` has a big effect when k is not very small, because it reduces the cost of computing the intersections and their levels from $O(m^3)$ to $O(m^2)$. Using `Optimized-INCH` alone introduces some additional cost because we need to sort the intersections (see section 4.5 for detail explanation). When both optimizations are used, `Optimized-INCH` has some benefit because then the sorted lists of intersections can also be maintained incrementally, and it reduces the cost of convex hull computation from $O(m^2 \log(m))$ to $O(m \log(m))$.

6.2 Results of RkNN queries

Figure 9 shows the algorithms' performance on the LA dataset, with the value of k varies from 1 to 10. We see that `FINCH` is more efficient than TPL with all values of k. We also find that `FINCH` scales quite well with k.

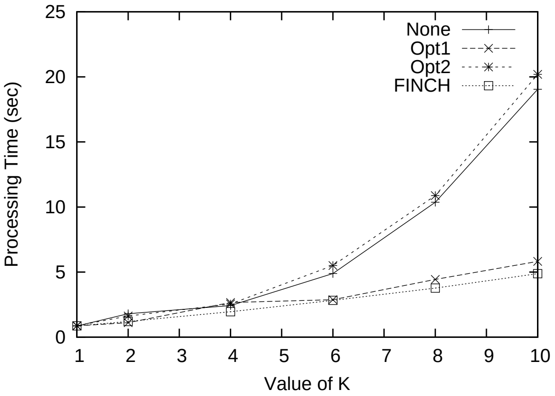


Figure 8: Effects of optimizations on `INCH`.

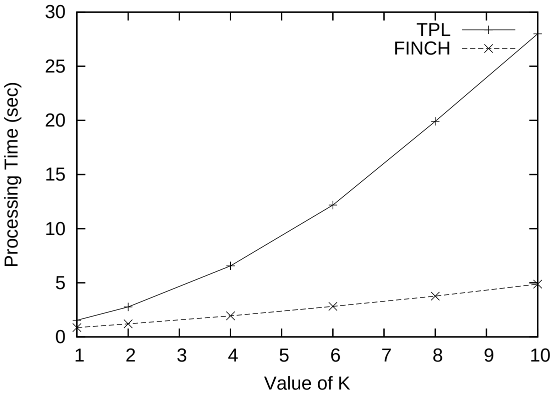


Figure 9: Effect of k using LA dataset.

Figure 10 is a breakdown of the algorithms' running times when k=4. In the figure, each algorithm has two bars. The left one is the time spent in the filter phase, and the right one is the time spent in the refinement phase. And each bar has two components that are the CPU time (white) and disk I/O time (black). From this figure, we observe that `FINCH` is much more efficient than TPL in the filter phase. This is because in `FINCH` each R-tree node (or data point) is checked against a single search region to find whether it can be pruned, while in TPL each R-tree node (or data point) is checked with their k-trim method which basically tries to find a set of k objects that can prune the node. Their pruning method is slow especially when a node cannot be pruned, because the node will be tested against lots of sets of k objects. Our pruning method is fast because we only need to test whether the given node's MBR (minimum bounding rectangle, stored in R-tree) intersects with the search region. Note that the test for pruning is the most basic and important operation in filter, and if a node cannot be pruned then its many children nodes (or data points) need to be tested for pruning.

We also observe from the breakdown figure that in the filter phase CPU time dominates the running time, while in the refinement phase I/O time dominates the running time. As a whole, CPU time dominates[3].

[1] We used a Java port of TPL's original C++ implementation. The Java port strictly follows the C++ implementation.

[2] http://www.census.gov/geo/www/tiger/

[3] This observation is contrary to that in [15]. In [15] I/O time is approximated by counting the number of nodes accessed

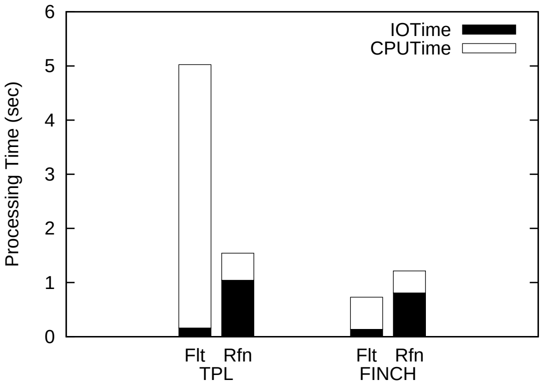


Figure 10: Breakdown of running times on dataset LA and k=4.

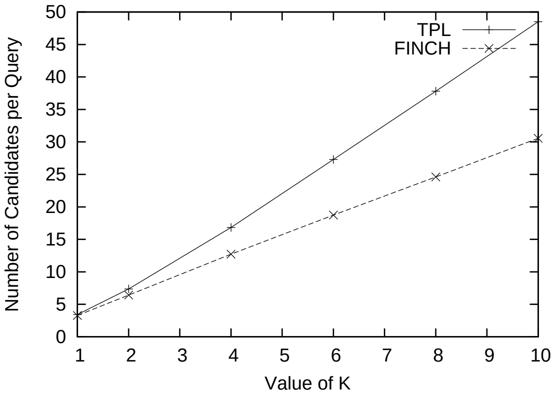


Figure 11: Num of candidates vs. k

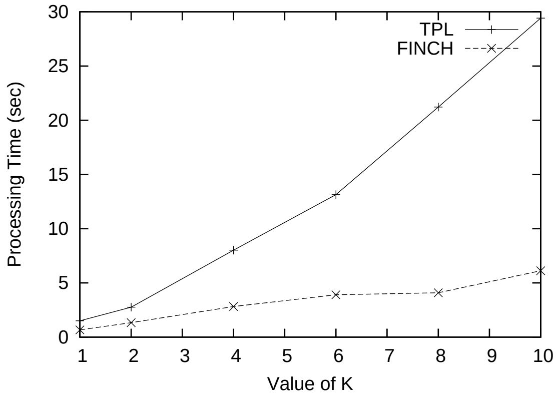


Figure 12: Effect of k using CA dataset

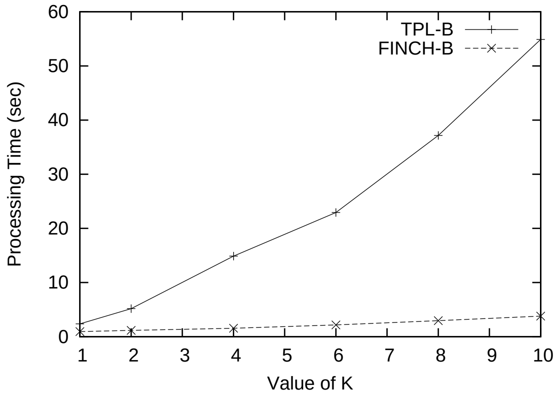


Figure 13: BRkNN, Effect of k using LA dataset

Figure 11 plots the number of candidates we get after the algorithm's filter phase. It shows that FINCH produces smaller number of candidates. It indicates that FINCH has a stronger filter power. It is because our computed search region is tight while TPL trades some pruning power for filtering speed (see section 2.3.1). This also explains why TPL's refinement time in Figure 10 is more than FINCH's.

Figure 12 shows the result of experiments on the CA dataset. The CA dataset is much larger than the LA dataset. Comparing this figure with Figure 9, we find that the performance differences between the FINCH and TPL on the two datasets are similar. This means that dataset size does not have a significant effect on the two algorithms. This also shows that both the algorithms are quite scalable with respect to dataset size.

6.3 Results of bichromatic RkNN queries

Here we present our experiments on bichromatic RkNN queries. Note that there is no existing bichromatic RkNN algorithm that supports an arbitrary value of k (see section 2.4). TPL is designed for monochromatic RkNN and cannot handle bichromatic RkNN queries directly. We compare our FINCH-B algorithm with TPL-B (see section 5.2.1 for detail), which is the adapted version of TPL with our general BRkNN evaluation framework.

The LA dataset is used as dataset P. For dataset R, it is a modified version of the LA dataset. The two datasets have the same data space and similar (but not same) points distributions.

Figure 13 illustrates the algorithms' running times with different k values. It is clear that FINCH-B performs much better than TPL-B. Comparing Figure 13 with Figure 9, we find that the gap between FINCH-B and TPL-B is bigger than the gap between FINCH and TPL. This is explained below along the discussion with Figure 14.

Figure 14 shows the breakdown of the running times when k=4. In the figure, each algorithm has three bars: FltA shows the processing time of the first filter phase, FltB shows the processing time spent in the second filter phase, and Rfn shows the refinement time. We observe that in the second filter phase the two algorithms exhibit very different performance. This explains why the gap between FINCH-B and TPL-B is wider than the one between FINCH and TPL. TPL-B's second filter phase takes similar time as the first filter phase because in TPL-B the pruning is done in the same way as in the first phase. FINCH-B's second filter phase is much faster than its first filter phase. This is because of two reasons: 1) the search region is static during the second filter phase, and this means in the second filter phase we do not need to spend CPU time on computing the search

and multiplying by a constant (10ms), and a node that is accessed multiple times is counted as separate I/Os [16] (i.e. effectively no buffering at all).

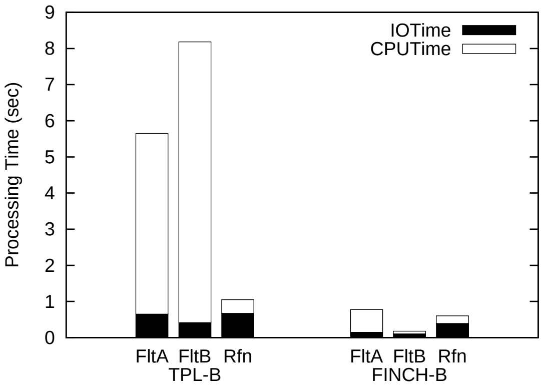


Figure 14: BRkNN, Breakdown of running times on dataset LA and k=4

region; 2) after the first filter phase, the query's search region is small, therefore the search region in the second filter phase has a very strong pruning power. Basically, in FINCH-B the second filter phase is like evaluating a range query with a very small range. This indeed is the most obvious advantage of having a search region calculated.

7. CONCLUSION

A Reverse k-Nearest-Neighbor query finds the objects that are influenced by the querying object. It can be applied in Location-Based Services to answer interesting location related questions. We have presented our solutions for evaluating RkNN queries on location data. We define a RkNN query's search region and propose an algorithm called INCH to compute it based on the query and a set of data objects. INCH is then used in our RkNN solutions to filter and restrict the search space for result candidates. We also presented a method of applying (monochromatic) RkNN algorithms to evaluate bichromatic RkNN queries. Experimental results show that the search region computed by INCH has a strong pruning power, and it speeds up the filter process. These factors make our RkNN solutions much more efficient than the existing state-of-the-art RkNN algorithm.

8. ACKNOWLEDGEMENT

We thank the authors of [15] for providing us the source code of their TPL algorithm. We also thank the anonymous reviewers for their constructive suggestions on improving the clarity of the paper.

9. REFERENCES

[1] E. Achtert, C. Bohm, P. Kroger, P. Kunath, A. Pryakhin, and M. Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *SIGMOD*, 2006.

[2] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *The VLDB Journal*, 15(3):229–249, 2006.

[3] M. d. Berg, M. v. Krefeld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer, 2nd edition, 2000.

[4] K. L. Cheung and A. W.-C. Fu. Enhanced nearest neighbour search on the r-tree. *SIGMOD Rec.*, 27(3):16–21, 1998.

[5] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[6] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[7] J. M. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang. Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In *ICDE*, 2007.

[8] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD*, 2000.

[9] B. Rao and L. Minakakis. Evolution of mobile location-based services. *Commun. ACM*, 46(12):61–65, 2003.

[10] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.

[11] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, 2000.

[12] A. Singh, H. Ferhatosmanoglu, and A. S. Tosun. High dimensional reverse nearest neighbor queries. In *CIKM*, 2003.

[13] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2000.

[14] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of influence sets in frequently updated databases. In *VLDB*, 2001.

[15] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB*, 2004.

[16] Y. Tao and X. Xiao. Private communication, 2008.

[17] Y. Tao, M. L. Yiu, and N. Mamoulis. Reverse nearest neighbor search in metric spaces. *IEEE Transactions on Knowledge and Data Engineering*, 18(9):1239–1252, 2006.

[18] W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan. Continuous reverse k-nearest-neighbor monitoring. In *MDM*, Beijing, 2008.

[19] T. Xia and D. Zhang. Continuous reverse nearest neighbor monitoring. In *ICDE*, 2006.

[20] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *ICDE*, 2001.

[21] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse nearest neighbors in large graphs. *IEEE Transactions on Knowledge and Data Engineering*, 18(4):540–553, 2006.