Aggregate-Max Nearest Neighbor Searching in the Plane

Haitao Wang*

Abstract

We study the aggregate nearest neighbor searching for the Max operator in the plane. For a set P of n points and a query set Q of m points, the query asks for a point of P whose maximum distance to the points in Q is minimized. We present data structures for answering such queries for both L_1 and L_2 distance measures. Previously, only heuristic and approximation algorithms were given for both versions. For the L_1 version, we build a data structure of $O(n)$ size in $O(n \log n)$ time, such that each query can be answered in $O(m + \log n)$ time. For the L_2 version, we build a data structure of $O(n \log \log n)$ size in $O(n \log n)$ time, such that each query can be answered in $O(m\sqrt{n} \log^{O(1)} n)$ time, and alternatively, we build a data structure in $O(n^{2+\epsilon})$ time and space for any $\epsilon > 0$, such that each query can be answered in $O(m \log n)$ time.

1 Introduction

Aggregate nearest neighbor (ANN) searching [1, 7, 8, 9, 10, 11, 15, 16, 17, 18, 19], also called group nearest neighbor searching, is a generalization of the fundamental nearest neighbor searching problem [2], where the input of each query is a set of points and the result of the query is based on applying some *aggregate* operator (e.g., Max and Sum) on all query points. In this paper, we consider the ANN searching on the Max operator for both L_1 and L_2 metrics in the plane.

For any two points p and q, let $d(p, q)$ denote the distance between p and q. Let P be a set of n points in the plane. Given any query set Q of m points, the ANN query asks for a point p in P such that $g(p, Q)$ is minimized, where $g(p, Q)$ is the *aggregate function* of the distances from p to the points of Q. The aggregate functions commonly considered are Max, i.e., $g(p, Q) = \max_{q \in Q} d(p, q)$, and Sum, i.e., $g(p, Q) = \sum_{q \in Q} d(p, q)$. If the operator for g is Max (resp., Sum), we use ANN-Max (resp., ANN-Sum) to denote the problem.

In this paper, we focus on ANN-Max in the plane for both L_1 and L_2 versions where the distance $d(p, q)$ is measured by L_1 and L_2 metrics, respectively.

Previously, only heuristic and approximation algorithms were given for both versions. For the L_1 version, we build a data structure of $O(n)$ size in $O(n \log n)$ time, such that each query can be answered in $O(m + \log n)$ time. For the L_2 version, we build a data structure of $O(n \log \log n)$ size in $O(n \log n)$ time, such that each query can be answered in $O(m\sqrt{n} \log^{O(1)} n)$ time, and alternatively, we build a data structure in $O(n^{2+\epsilon})$ time and space for any $\epsilon > 0$, such that each query can be answered in $O(m \log n)$ time.

1.1 Previous Work

For ANN-Max, Papadias et al. [16] presented a heuristic Minimum Bounding Method with worst case query time $O(n + m)$ for the L_2 version. Recently, Li et al. [7] gave more results on the L_2 ANN-Max (the queries were called *group enclosing queries*). By using R-tree [6], Li et al. [7] gave an exact algorithm to answer ANN-Max queries, and the algorithm is very fast in practice but theoretically the worst case query time is still $O(n+m)$. Li et al. [7] also gave a $\sqrt{2}$-approximation algorithm with query time $O(m+\log n)$ and the algorithm works for any fixed dimensions, and they further extended the algorithm to obtain a $(1 + \epsilon)$-approximation result. To the best of our knowledge, we are not aware of any previous work that is particularly for the L_1 ANN-Max. However, Li et al. [9] proposed the *flexible* ANN queries, which extend the classical ANN queries, and they provided an $(1 + 2\sqrt{2})$-approximation algorithm that works for any metric space in any fixed dimension.

For L_2 ANN-Sum, a 3-approximation solution is given in [9]. Agarwal et al. [1] studied nearest neighbor searching under uncertainty, and their results can give an $(1 + \epsilon)$-approximation solution for the L_2 ANN-Sum queries. They [1] also gave an exact algorithm that can solve the L_1 ANN-Sum problem and an improvement based on their work has been made in [18].

There are also other heuristic algorithms on ANN queries, e.g., [8, 10, 11, 15, 17, 19].

Comparing with n, the value m is relative small in practice. Ideally we want a solution that has a query time $o(n)$. Our L_1 ANN-Max solution is the first-known exact solution and is likely to be the best-possible. Comparing with the heuristic result [7, 16] with $O(m+n)$ worst case query time, our L_2 ANN-Max solution use $o(n)$ query time for small m; it should be noted that the methods in [7, 16] uses only $O(n)$ space while the space used in our approach is larger.

*Department of Computer Science, Utah State University, Logan, UT 84322, USA. E-mail: haitao.wang@usu.edu.

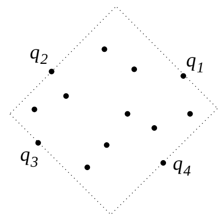

Figure 1: Illustrating the four extreme points q_1, q_2, q_3, q_4.

2 The ANN-Max in the L_1 Metric

In this section, we present our solution for the L_1 version of ANN-Max queries. Given any query point set Q, our goal is to find the point $p \in P$ such that $g(p, Q) = \max_{q \in Q} d(p, q)$ is minimized for the L_1 distance $d(p, q)$, and we denote by $\psi(Q)$ the above sought point.

For each point p in the plane, denote by $p_{\max}$ the farthest point of Q to p. We show below that $p_{\max}$ must be an extreme point of Q along one of the four *diagonal* directions: northeast, northwest, southwest, southeast.

Let ρ_1 be a ray directed to the "northeast", i.e., the angle between ρ and the x-axis is $\pi/4$. Let q_1 be an extreme point of Q along ρ_1 (e.g., see Fig. 1); if there is more than one such point, we let q_1 be an arbitrary such point. Similarly, let q_2, q_3, and q_4 be the extreme points along the directions northwest, southwest, and southeast, respectively. Let $Q_{\max} = \{q_1, q_2, q_3, q_4\}$. Note that $Q_{\max}$ may have less than four *distinct* points if two or more points of $Q_{\max}$ refer to the same (physical) point of Q. Lemma 1, whose proof is omitted, shows that $g(p, Q)$ is determined only by the points of $Q_{\max}$.

Lemma 1 *For any point p in the plane, it holds that $g(p, Q) = \max_{q \in Q_{\max}} d(p, q)$.*

Note that a point may have more than one farthest point in Q. For any point p, if p has only one farthest point in Q, then $p_{\max}$ is in $Q_{\max}$. Otherwise, $p_{\max}$ may not be in $Q_{\max}$, and for convenience we re-define $p_{\max}$ to be the farthest point of p in $Q_{\max}$. For each $1 \leq i \leq 4$, let $P_i = \{p \mid p_{\max} = q_i, p \in P\}$, i.e., P_i consists of the points of P whose farthest points in Q are q_i, and let p_i be the nearest point of q_i in P_i. To find $\psi(Q)$, we have the following lemma, whose proof is omitted.

Lemma 2 *$\psi(Q)$ is the point p_j for some j with $1 \leq j \leq 4$, such that $d(p_j, q_j) \leq d(p_i, q_i)$ holds for any $1 \leq i \leq 4$.*

Based on Lemma 2, to determine $\psi(Q)$, it is sufficient to determine p_i for each $1 \leq i \leq 4$. To this end, we make use of the farthest Voronoi diagram [5] of the four points in $Q_{\max}$, which is also the farthest Voronoi diagram of Q by Lemma 1. Denote by *FVD*(Q) the farthest Voronoi diagram of $Q_{\max}$. Since $Q_{\max}$ has only four points, *FVD*(Q) can be computed in constant time, e.g., by an incremental approach. Each point $q \in Q_{\max}$ defines a cell $C(q)$ in *FVD*(Q) such that every point $p \in C(q)$ is farthest to q_i among all points of $Q_{\max}$. In order to compute the four points p_i with $i = 1, 2, 3, 4$, we first show in the following that each cell $C(q)$ has certain special shapes that allow us to make use of the segment dragging queries [4, 14] to find the four points efficiently. Note that for each $1 \leq i \leq 4$, $P_i = P \cap C(q_i)$ and thus p_i is the nearest point of $P \cap C(q_i)$ to q_i. In fact, the following discussion also gives an incremental algorithm to compute *FVD*(Q) in constant time.

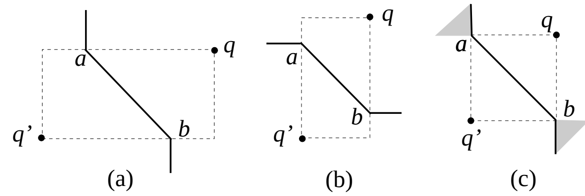

Figure 2: Illustrating the bisector $B(q, q')$ (the solid curve) for q and q'. In (c), since $R(q, q')$ is a square, the two shaded quadrants are entirely in $B(q, q')$, but for simplicity, we only consider the two vertical bounding half-lines as in $B(q, q')$.

2.1 The Bisectors

We first briefly discuss the bisectors of the points based on the L_1 metric. In fact, the L_1 bisectors have been well studied (e.g., [14]) and we discuss them here for completeness and some notation introduced here will also be useful later when we describe our algorithm.

For any two points q and q' in the plane, define $r(q, q')$ as the region of the plane that is the locus of the points farther to q than to q', i.e., $r(q, q') = \{p \mid d(p, q) \geq d(p, q')\}$. The *bisector* of q and q', denoted by $B(q, q')$, is the locus of the points that are equidistant to q and q', i.e., $B(q, q') = \{p \mid d(p, q) = d(p, q')\}$. In order to discuss the shapes of the cells of *FVD*(Q), we need to elaborate on the shape of $B(q, q')$, as follows.

Let $R(q, q')$ be the rectangle that has q and q' as its two vertices on diagonal positions (e.g., see Fig. 2). If the line segment $\overline{qq'}$ is axis-parallel, the rectangle $R(q, q')$ is degenerated into a line segment and $B(q, q')$ is the line through the midpoint of $\overline{qq'}$ and perpendicular to $\overline{qq'}$. Below, we focus on the general case where $\overline{qq'}$ is not axis-parallel. Without loss of generality, we assume q and q' are northeast and southwest vertices of $R(q, q')$, and other cases are similar.

The bisector $B(q, q')$ consists of two half-lines and one line segment in between (e.g., see Fig. 2); the two half-lines are either both horizontal or both vertical. Specifically, let l be the line of slope -1 that contains the midpoint of $\overline{qq'}$. Let $\overline{ab} = l \cap R(q, q')$, and a and b are on the boundary of $R(q, q')$. Note that if $R(q, q')$ is a square, then a and b are the other two vertices of $R(q, q')$ than q and q'; otherwise, neither a nor b is a vertex.

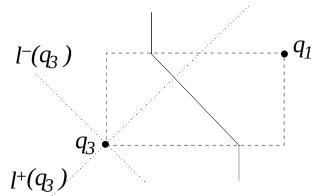

Figure 3: Illustrating an example where q_1 is above $l^-(q_3)$ and below or on $l^+(q_3)$. The bisector $B(q_1, q_3)$ is a v-bisector.

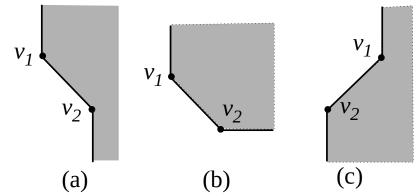

Figure 4: Illustrating three types of regions (shaded).

We first discuss the case where $R(q, q')$ is not a square (e.g., see Fig. 2 (a) and (b)). Let $l(a)$ be the line through a and perpendicular to the edge of $R(q, q')$ that contains a. The point a divides $l(a)$ into two half-lines, and we let $l'(a)$ be the one that doest not intersect $R(q, q')$ except a. Similarly, we define the half-line $l'(b)$. Note that $l'(a)$ and $l'(b)$ must be parallel. The bisector $B(q, q')$ is the union of $l'(a)$, $\overline{ab}$, and $l'(b)$.

If $R(q, q')$ is a square, both a and b are vertices of $R(q, q')$ (e.g., see Fig. 2 (c)). In this case, a quadrant of a and a quadrant of b belong to $B(q, q')$, but for simplicity, we consider $B(q, q')$ as the union of $\overline{ab}$ and the two vertical bounding half-lines of the two quadrants.

We call $\overline{ab}$ the *middle segment* of $B(q, q')$ and denote it by $B_M(q, q')$. If $B(q, q')$ contains two vertical half-lines, we call $B(q, q')$ a *v-bisector* and refer to the two half-lines as *upper half-line* and *lower half-line*, respectively, based on their relative positions; similarly, if $B(q, q')$ contains two horizontal half-lines, we call $B(q, q')$ an *h-bisector* and refer to the two half-lines as *left half-line* and *right half-line*, respectively.

For any point p in the plane, we use $l^+(q)$ to denote the line through q with slope 1, $l^-(q)$ the line through q with slope -1, $l_h(q)$ the horizontal line through q, and $l_v(q)$ the vertical line through q.

2.2 The Shapes of Cells of $FVD(Q)$

A subset Q' of Q is *extreme* if it contains an extreme point along each of the four diagonal directions. $Q_{\max}$ is an extreme subset. A point q of $Q_{\max}$ is *redundant* if $Q_{\max} \setminus \{q\}$ is still an extreme subset. For simplicity of discussion, we remove all redundant points from $Q_{\max}$. For example, if q_1 and q_2 are both extreme points along the northeast direction (and q_2 is also an extreme point along the northwest direction), then q_1 is redundant and we simply remove q_1 from $Q_{\max}$ (and the new q_1 of $Q_{\max}$ now refers to the same physical point as q_2).

Consider a point $q \in Q_{\max}$. Without loss of generality, we assume $q = q_3$ and the other cases can be analyzed similarly. We will analyze the possible shapes of $C(q_3)$. We assume $Q_{\max}$ has at least two distinct points since otherwise the problem would be trivial. We further assume $q_1 \neq q_3$ since otherwise the analysis is much simpler. According to their definitions, q_1 must be above the line $l^-(q_3)$ (e.g., see Fig. 3). However, q_1 can be either above or below the line $l^+(q_3)$. In the following discussion, we assume q_1 is below or on the line $l^+(q_3)$ and the case where q_1 is above $l^+(q_3)$ can be analyzed similarly. In this case $B(q_3, q_1)$ is a v-bisector (i.e., it has two vertical half-lines).

We first introduce three *types* of regions (i.e., *A, B*, and *C*), and we will show later that $C(q_3)$ must belong to one of the types. Each type of region is bounded from the left or below by a polygonal curve ∂ consisting of two half-lines and a line segment of slope ± 1 in between.

1. From top to bottom, the polygonal curve ∂ consists of a vertical half-line followed by a line segment of slope -1 and then followed by a vertical half-line extended downwards (e.g., see Fig. 4 (a)). The region on the right of ∂ is defined as a *type-A* region.

2. From top to bottom, ∂ consists of a vertical half-line followed by a line segment of slope -1 and then followed by a horizontal half-line extended rightwards (e.g., see Fig. 4 (b)). The region on the right of and above ∂ is defined as a *type-B* region.

3. From top to bottom, ∂ consists of a vertical half-line followed by a line segment of slope 1 and then followed by a vertical half-line extended downwards (e.g., see Fig. 4 (c)). The region on the right of ∂ is defined as a *type-C* region.

In each type of the regions, the line segment of ∂ is called the *middle segment*. Denote by v_1 the upper endpoint of the middle segment and by v_2 the lower endpoint (e.g., see Fig. 4). Note that the middle segment may be degenerated to a point. By constructing $C(q_3)$ in an incremental manner, Lemma 3 shows that $C(q_3)$ must belong to one of the three types of regions. The proof of Lemma 3 is omitted.

Lemma 3 *The cell $C(q_3)$ must be one of the three types of regions. Further (see Fig. 5), if $C(q_3)$ is a type-A region, then $C(q_3)$ is to the right of $l_v(q_3)$ and v_2 is on $l_h(q_3)$; if $C(q_3)$ is a type-B region, then $C(q_3)$ is to the right of $l_v(q_3)$ and above $l_h(q_3)$; if $C(q_3)$ a type-C region, then $C(q_3)$ is to the right of $l_v(q_3)$ and v_1 is on $l_h(q_3)$.*

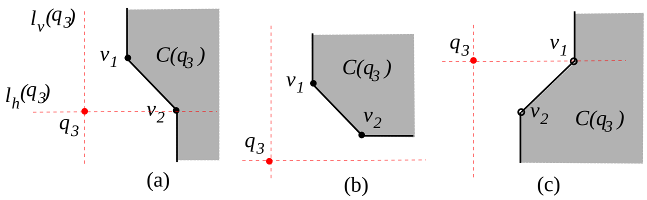

Figure 5: Illustrating the three possible cases for $C(q_3)$: (a) a type-A region; (b) a type-B region; (c) a type-C region.

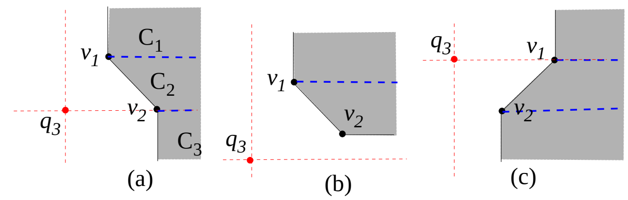

Figure 7: Illustrating the decomposition of $C(q_3)$ for segment-dragging queries.

2.3 Answering the Queries

Recall that our goal is to compute p_3, which is is the nearest point of $P \cap C(q_3)$ to q_3. Based on Lemma 3, we can compute the point p_3 in $O(\log n)$ time by making use of the segment dragging queries [4, 14]. The details are given in Lemma 4.

Lemma 4 *After $O(n \log n)$ time and $O(n)$ space preprocessing on P, the point p_3 can be found in $O(\log n)$ time.*

Proof. We first briefly introduce the *segment dragging queries* that will be used by our algorithm: *parallel-track queries* and *out-of-corner queries* (e.g., Fig. 6).

Let S be a set of n points in the plane. For each parallel-track query, we are given two parallel vertical or horizontal lines (as "tracks") and a line segment of slope ± 1 with endpoints on the two tracks, and the goal is to find the first point of S hit by the segment if we drag the segment along the two tracks. For each out-of-corner query, we are given two axis-parallel tracks forming a perpendicular corner, and the goal is to find the first point of S hit by dragging out of the corner a segment of slope ± 1 with endpoints on the two tracks.

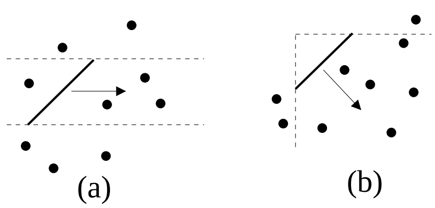

Figure 6: Illustrating the segment dragging queries: (a) a parallel-track query; (b) an out-of-corner query.

As shown by Mitchell [14], after $O(n \log n)$ time and $O(n)$ space preprocessing on S, each of the two types of queries can be answered in $O(\log n)$ [4, 14].

Below, we present our algorithm for the lemma by using the above segment dragging queries. Our goal is to find p_3. Depending on the type of the $C(q_3)$ as stated in Lemma 3, there are three cases.

Type-A If $C(q_3)$ is a type-A region, we further decompose $C(q_3)$ into three subregions (e.g., see Fig. 7 (a)) by introducing two horizontal half-lines going rightwards from v_1 and v_2 (i.e., the endpoints of the middle segment of the boundary of $C(q_3)$), respectively. We call the three subregions the *upper*, *middle*, and *lower* subregions, respectively, according to their heights. To find p_3, for each subregion C, we compute the closest point of $P \cap C$ to q_3, and p_3 is the closest point to q_3 among the three points found above.

For the upper subregion, denoted by C_1, according to Lemma 3, C_1 is in the first quadrant of q_3. Therefore, q_3's closest point in $P \cap C_1$ is exactly the answer of the out-of-corner query by dragging a segment of slope -1 from the corner of C_1.

For the middle subregion, denoted by C_2, according to Lemma 3, C_2 is in the first quadrant of q_3. Therefore, q_3's closest point in $P \cap C_2$ is exactly the answer of the parallel-track query by dragging the middle segment of the boundary of $C(q_3)$ rightwards.

For the lower subregion, denoted by C_3, according to Lemma 3, C_3 is in the fourth quadrant of q_3. Therefore, q_3's closest point in $P \cap C_3$ is exactly the answer of the out-of-corner query by dragging a segment of slope 1 from the corner of C_3.

Therefore, in this case we can find p_3 in $O(\log n)$ time after $O(n \log n)$ time $O(n)$ space preprocessing on P.

Type-B If $C(q_3)$ is a type-B region, we further decompose $C(q_3)$ into two subregions (e.g., see Fig. 7 (b)) by introducing a horizontal half-line rightwards from v_1. To find p_3, again, we find the closest point to q_3 in each of the two sub-regions.

By Lemma 3, both subregions are in the first quadrant of q_3. By using the same approach as the first case, q_3's closest point in the upper subregion can be found by an out-of-corner query and q_3's closest point in the lower subregion can be found by a parallel-track query.

Type-C If $C(q_3)$ is a type-C region, the case is symmetric to the first case and we can find p_3 by using two out-of-corner queries and a parallel-track query.

As a summary, we can find p_3 in $O(\log n)$ time after $O(n \log n)$ time $O(n)$ space preprocessing on P. $\square$

By combining Lemmas 2 and 4, we conclude this section with the following theorem.

Theorem 5 *Given a set P of n points in the plane, after $O(n \log n)$ time and $O(n)$ space preprocessing, we can answer each L_1 ANN-Max query in $O(m + \log n)$ time for any set Q of m query points.*

Proof. As preprocessing, we build data structures for answering the segment dragging queries on P in $O(n \log n)$ time and $O(n)$ space [4, 14].

Given any query set Q, we first determine $Q_{\max}$ in $O(m)$ time. Then, we compute the farthest Voronoi diagram $FVD(Q)$ in constant time, e.g., by the incremental approach given in this paper. Then, for each $1 \leq i \leq 4$, we compute the point p_i by Lemma 4 in $O(\log n)$ time. Finally, $\psi(Q)$ can be determined by Lemma 2. □

3 The ANN-Max in the L_2 Metric

In this section, we present our results for the L_2 version of ANN-Max queries. Given any query point set Q, our goal is to find the point $p \in P$ such that $g(p, Q) = \max_{q \in Q} d(p, q)$ is minimized for the L_2 distance $d(p, q)$, and we use $\psi(Q)$ to denote the sought point above.

We follow the similar algorithmic scheme as in the L_1 version. Let Q_H be the set of points of Q that are on the convex hull of Q. It is known that for any point p in the plane, its farthest point in Q is in Q_H, and in other words, the farthest Voronoi diagram of Q, denoted by $FVD(Q)$, is determined by the points of Q_H [5, 7]. Note that the size of $FVD(Q)$ is $O(|Q_H|)$ [5].

Consider any point $q \in Q_H$. Denote by $C(q)$ the cell of q in $FVD(Q)$, which is a convex and unbounded polygon [5]. Let $f(q)$ be the closest point of $P \cap C(q)$ to q. Similar to Lemma 2, we have the following lemma.

Lemma 6 *If for a point $q' \in Q$, $d(f(q'), q') \leq d(f(q), q)$ holds for any $q \in Q_H$, then $f(q')$ is $\psi(Q)$.*

Hence, to find $\psi(Q)$, it is sufficient to determine $f(q)$ for each $q \in Q$, as follows.

Consider any point $q \in Q$. To find $f(q)$, we first triangulate the cell $C(q)$ and let $Tri(q)$ denote the triangulation. For each triangle $\triangle \in Tri(q)$, we will find the closest point to q in $P \cap \triangle$, denoted by $f_\triangle(q)$. Consequently, $f(q)$ is the closest point to q among the points $f_\triangle(q)$ for all $\triangle \in Tri(q)$. Out goal is to determine $\psi(Q)$. To this end, we will need to triangulate each cell of $FVD(Q)$ and compute $f_\triangle(q)$ for each $\triangle \in Tri(q)$ and for each $q \in Q$. Since the size of $FVD(Q)$ is $O(|Q_H|)$, which is $O(m)$, we have the following lemma.

Lemma 7 *If the closest point $f_\triangle(q)$ to q in $P \cap \triangle$ can be found in $O(t_\triangle)$ time for any triangle $\triangle$ and any point q in the plane, then $\psi(Q)$ can be found in $O(m \cdot t_\triangle)$ time.*

In the following, we present our algorithms for computing $f_\triangle(q)$ for any triangle $\triangle$ and any point q in the plane. If we know the Voronoi diagram of the points in $P \cap \triangle$, then $f_\triangle(q)$ can be determined in logarithmic time. Hence, the problem becomes how to maintain the Voronoi diagrams for the points in P such that given any triangle $\triangle$, the Voronoi diagram information of the points in $P \cap \triangle$ can be obtained efficiently. To this end, we choose to augment the $O(n)$-size simplex range (counting) query data structure in [12], as shown in the following lemma.

Lemma 8 *After $O(n \log n)$ time and $O(n \log \log n)$ space preprocessing on P, we can compute the point $f_\triangle(q)$ in $O(\sqrt{n} \log^{O(1)} n)$ time for any triangle $\triangle$ and any point q in the plane.*

Proof. We first briefly discuss the data structure in [12] and then augment it for our purpose. Note that the data structure in [12] is for any fixed dimension and our discussion below only focuses on the planar case, and thus each simplex below refers to a triangle.

A *simplicial partition* of the point set P is a collection $\Pi = \{(P_1, \triangle_1), \ldots, (P_k, \triangle_k)\}$, where the P_i's are pairwise disjoint subsets (called the *classes* of Π) forming a partition of P, and each $\triangle_i$ is a possibly unbounded simplex containing the points of P_i. The *size* of Π is k. The simplex $\triangle_i$ may also contain other points in P than those in P_i. A simplicial partition is called *special* if $\max_{1 \leq i \leq k}\{|P_i|\} < 2 \cdot \min_{1 \leq i \leq k}\{|P_i|\}$.

The data structure in [12] is a partition tree, denoted by T, based on constructing special simplicial partitions on P recursively. The leaves of T form a partition of P into constant-sized subsets. Each internal node $v \in T$ is associated with a subset P_v (and its corresponding simplex $\triangle_v$) of P and a special simplicial partition Π_v of size $|P_v|^{1/2}$ of P_v. The root of T is associated with P. The *cardinality* of P_v (i.e., $|P_v|$) is stored at v. Each internal node v has $|P_v|^{1/2}$ children that correspond to the classes of Π_v. Thus, if v is a node lying at a distance i from the root of T, then $|P_v| = O(n^{1/2^i})$, and the depth of T is $O(\log \log n)$. It is shown in [12] that T has $O(n)$ space and can be constructed in $O(n \log n)$ time.

For each query simplex $\triangle$, the goal is to compute the number of points in $P \cap \triangle$. We start from the root of T. For each internal node v, we check its simplicial partition Π_v one by one, and handle directly those contained in $\triangle$ or disjoint from $\triangle$; we proceed with the corresponding child nodes for the other simplices. Each of the latter ones must be intersected by at least one of the lines bounding $\triangle$. If v is a leaf node, for each point p in P_v, we determine directly whether $p \in \triangle$. Each query takes $O(n^{1/2} (\log n)^{O(1)})$ time [12].

For our purpose, we augment the partition tree T. For each node v, we explicitly maintain the Voronoi diagram of P_v, denoted by $VD(P_v)$. Since at each level of T the subsets P_v's are pairwise disjoint, comparing with the original tree, our augmented tree has $O(n)$ additional space at each level. Since T has $O(\log \log n)$ levels, the total space of our augmented tree is $O(n \log \log n)$. For the running time, we claim that the total time for building the augmented tree is still $O(n \log n)$ although we have to build Voronoi diagrams for the nodes. Indeed, let $T(n)$ denote the time for building the Voronoi diagrams in the entire algorithm. We have $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + O(n \log n)$, and thus, $T(n) = O(n \log n)$ by solving the above recurrence.

Consider any query triangle $\triangle$ and any point q. We start from the root of T. For each internal node v, we check its simplicial partition Π_v, i.e., check the children of v one by one. Consider any child u of v. If $\triangle_u$ is disjoint from $\triangle$, we ignore it. If $\triangle_u$ is contained in $\triangle$, then we compute in $O(\log n)$ time the closest point of $P \cap \triangle_u$ to q (and its distance to q) by using the Voronoi diagram $VD(P_u)$ stored at the node u. Otherwise, we proceed on u recursively. If v is a leaf node, for each point p in P_v, we compute directly the distance $d(q,p)$ if $p \in \triangle$. Finally, $f_\triangle(q)$ is the closest point to q among all points whose distances to q have been computed above.

Comparing with the original simplex range query on $\triangle$, we have $O(\log n)$ additional time on each node u if $\triangle_u$ is contained in $\triangle$, and the number of such nodes is bounded by $O(n^{1/2}(\log n)^{O(1)})$. Hence, the total query time for finding $f_\triangle(q)$ is $O(n^{1/2}(\log n)^{O(1)} \cdot \log n)$, which is $O(n^{1/2}(\log n)^{O(1)})$. The lemma thus follows. □

Similar augmentation may also be made on the $O(n)$-size simplex data structure in [13] and the recent randomized result in [3]. If more space are allowed, by using duality and cutting trees [5], we can obtain the following lemma, whose proof is omitted.

Lemma 9 *After $O(n^{2+\epsilon})$ time and space preprocessing on P, we can compute the point $f_\triangle(q)$ in $O(\log n)$ time for any triangle $\triangle$ and any point q in the plane.*

Lemmas 7, 8, and 9 lead to the following theorem.

Theorem 10 *Given a set P of n points in the plane, after $O(n \log n)$ time and $O(n \log \log n)$ space preprocessing, we can answer each L_1 ANN-Max query in $O(m\sqrt{n} \log^{O(1)} n)$ time for any set Q of m query points; alternatively, after $O(n^{2+\epsilon})$ time and space preprocessing for any $\epsilon > 0$, we can answer each L_2 ANN-Max query in $O(m \log n)$ time.*

References

[1] P.K. Agarwal, A. Efrat, S. Sankararaman, and W. Zhang. Nearest-neighbor searching under uncertainty. In *Proc. of the 31st Symposium on Principles of Database Systems*, pages 225–236, 2012.

[2] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A.Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45:891–923, 1998.

[3] T.M. Chan. Optimal partition trees. *Discrete and Computational Geometry*, 47:661–690, 2012.

[4] B. Chazelle. An algorithm for segment-dragging and its implementation. *Algorithmica*, 3(1–4):205–221, 1988.

[5] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry — Algorithms and Applications.* Springer-Verlag, Berlin, 3rd edition, 2008.

[6] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[7] F. Li, B. Yao, and P. Kumar. Group enclosing queries. *IEEE Transactions on Knowledge and Data Engineering*, 23:1526–1540, 2011.

[8] H. Li, H. Lu, B. Huang, and Z. Huang. Two ellipse-based pruning methods for group nearest neighbor queries. In *Proc. of the 13th Annual ACM International Workshop on Geographic Information Systems*, pages 192–199, 2005.

[9] Y. Li, F. Li, K. Yi, B. Yao, and M. Wang. Flexible aggregate similarity search. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 1009–1020, 2011.

[10] X. Lian and L. Chen. Probabilistic group nearest neighbor queries in uncertain databases. *IEEE Transactions on Knowledge and Data Engineering*, 20:809–824, 2008.

[11] Y. Luo, H. Chen, K. Furuse, and N. Ohbo. Efficient methods in finding aggregate nearest neighbor by projection-based filtering. In *Proc. of the 12nd International Conference on Computational Science and its Applications*, pages 821–833, 2007.

[12] J. Matoušek. Efficient partition trees. *Discrete and Computational Geometry*, 8(3):315–334, 1992.

[13] J. Matoušek. Range searching with efficient hierarchical cuttings. *Discrete and Computational Geometry*, 10(1):157–182, 1993.

[14] J.S.B. Mitchell. L_1 shortest paths among polygonal obstacles in the plane. *Algorithmica*, 8(1):55–88, 1992.

[15] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *Proc. of the 20th International Conference on Data Engineering*, pages 301–312, 2004.

[16] D. Papadias, Y. Tao, K. Mouratidis, and C.K. Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Transactions on Database Systems*, 30:529–576, 2005.

[17] M. Sharifzadeh and C. Shahabi. VoR-Tree: R-trees with Voronoi diagrams for efficient processing of spatial nearest neighbor queries. In *Proc. of the VLDB Endowment*, pages 1231–1242, 2010.

[18] H. Wang and W. Zhang. The L_1 top-k nearest neighbor searching with uncertain queries. arXiv:1211.5084, 2013.

[19] M.L. Yiu, N. Mamoulis, and D. Papadias. Aggregate nearest neighbor queries in road networks. *IEEE Transactions on Knowledge and Data Engineering*, 17:820–833, 2005.