Queries with Segments in Voronoi Diagrams*

Sergei Bespamyatnikh† Jack Snoeyink‡

Abstract

In this paper we consider proximity problems in which the queries are line segments in the plane. We build a query structure that for a set of n points P can determine the closest point in P to a query segment outside the convex hull of P in $O(\log n)$ time. With this we solve the problem of computing the closest point to each of n disjoint line segments in $O(n \log^3 n)$ time. Nearest foreign neighbors or Hausdorff distance for disjoint, colored segments can be computed in the same time. We explore some connections to Hopcroft's problem.

1 Introduction

Since Knuth [13] posed the *post office problem*—preprocess a set of points, or *sites*, in the plane to quickly report the nearest to a query point—and Shamos and Hoey [17] suggested Voronoi diagrams as a solution, there have been a number of proximity problems in the plane whose solution is to build some type of Voronoi diagram and query with a point.

Note: A *Voronoi diagram* of a set of *sites* is the partition of the plane into maximal connected regions that have the same set of nearest sites [4, 8, 15]. In a Voronoi diagram the *Voronoi cell of site p* is the region that has p as its nearest site. We use Euclidean distance in this paper, so for two points a and b, $d(a,b) = \sqrt{(a-b)\cdot(a-b)}$. As is common, we extend this to the distance between sets of points: $d(A,B) = \liminf_{a\in A, b\in B}\{d(a,b)\}$. When the sets are closed, the distance is achieved by two points in the set and lim inf can be replaced by min.

For a further example of the use of the Voronoi diagram, consider the *Hausdorff distance* between two sets of n points A and B, which is defined as $\limsup_{a\in A, b\in B}\{d(a,B), d(b,A)\}$. If we color the points in the two sets red and blue, and then compute for every point the nearest point of the opposite color, the Hausdorff distance is the maximum. It can clearly be computed by locating each point of A in the Voronoi diagram of B and vice versa. Aggarwal et al. [3] generalized this to the *nearest foreign neighbors* problem: given n points, each assigned one of k colors, compute, for each point, a nearest neighbor that is of a different color. Nearest foreign neighbors can also be computed from Voronoi diagrams [3, 11]; we will see this reduction in a more general setting in Section 3.

Since Voronoi diagrams can be constructed for more complex sites, such as line segments or polygons [19, 14], we can represent each post office by the polygon that forms the outline of the building. If our query remains a point, then the approach to solve the post office problem remains the same. However, if the query becomes a line segment or polygon, or if we solve Hausdorff distance or nearest foreign neighbor problems for line segments, then querying a Voronoi diagram may no longer be efficient. In the worst case, our query segment could cross linearly-many cells of a Voronoi diagram and we might have to check the distance to each corresponding site.

One would not expect that segment queries would be as efficient as point queries. In fact, one could use segment queries to solve *Hopcroft's problem:* determine whether any point from a given set of n points lies on any of n given lines. Erickson [10] has shown that any algorithm that can be implemented in a computational model based on partition trees, which includes our algorithms, must take $\Omega(n^{4/3})$ time to solve Hopcroft's problem. We found it surprising, therefore, that the nearest foreign neighbor problem for disjoint segments, which gives rise to sets of disjoint segment queries known in advance, can be solved in $O(n \log^3 n)$ time.

Section 2 describes data structures for solving proximity problems on points in which the queries are segments. Section 3 uses these data structures, together with Voronoi diagrams of line segments, to solve the Hausdorff distance and nearest foreign neighbor problems for sets of disjoint segments.

Chazelle [6] considered the related problem of *segment dragging queries*, which ask to preprocess a set of n points to answer, for a horizontal query segment, what point is hit when the segment is translated vertically. The result of a segment dragging query is the closest point to a segment if there is no point that is closer to a segment endpoint. Chazelle gave two solutions to

*Research supported in part by an NSERC Research Grant. Some work performed during the second author's visit to INRIA Sophia Antipolis, supported by CIES and the Killam Foundation.

†UBC Comp Sci, Vancouver, Canada, besp@cs.ubc.ca

‡UBC Comp Sci, Vancouver, Canada, snoeyink@cs.ubc.ca

the segment-dragging problem: first by a data structure with $O(n \log n)$ space and $O(\log^2 n)$ query time, and then a refinement, in a RAM model, to linear space and $O(\log n)$ query time. Both solutions depend heavily on the ordering of points in the horizontal and vertical directions, and do not adapt to queries with non-horizontal segments. In fact, if they could, we would be able to solve Hopcroft's problem by dragging an infinitesimal-length segment along each line.

2 Segment queries in point Voronoi diagrams

We consider the following problem in three subsections: given a set P of n points, and a set S of m query segments, find the closest point of P to each segment of S. Subsection 2.1 considers the special case in which the query segments are outside the convex hull $ch(P)$. Subsection 2.2 extends this solution to handle the case in which the query segments in S are disjoint, or intersect only at endpoints. For completeness, subsection 2.3 explores the case of intersecting queries and its connection to Hopcroft's problem.

2.1 Queries outside of the convex hull

In this section we show how to construct a data structure for a set P of n points that can report, in $O(\log n)$ time, the closest point to a query segment that is outside the convex hull $ch(P)$. The structure can be constructed in $O(n \log n)$ time, or in linear time if the Voronoi diagram of P is given.

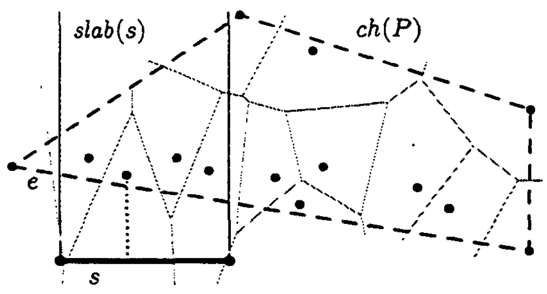

Figure 1: Characterizing the closest point

As mentioned in the introduction, one way to compute the closest point in P to a query segment s would be to compute the Voronoi diagram of P and to inspect the distance to each site whose cell intersects s. To find a more efficient approach, we must further characterize the solution to a query. For a segment s, the *slab of* s, or $slab(s)$, is the region bounded by the perpendiculars to s through the endpoints of s, as shaded in Figure 1.

LEMMA 2.1. *The minimum distance $d(s, P)$ between a point set P and a segment s outside $ch(P)$ is achieved by either an endpoint of s and its closest neighbor in P or by a point p in the slab of s whose Voronoi cell intersects the boundary of $ch(P)$ in the slab.*

Proof. Since s and P are closed sets, the distance $d(s, P)$ is realized by a point of each. Let p be the closest point in P. If the closest point to p is not an endpoint of s, then the shortest segment from p to s is perpendicular to s and remains inside the Voronoi cell of p, as illustrated in Figure 1. This segment is clearly in $slab(s)$.

The closest points to the endpoints of s are easy to obtain from the Voronoi diagram of P. We therefore concentrate on finding the closest point to the interior of s, subject to the constraints of the Lemma. We remark that these constraints make our subproblem different from a segment-dragging query [6], and allow us to use an ordering that depends on the data rather than on the query.

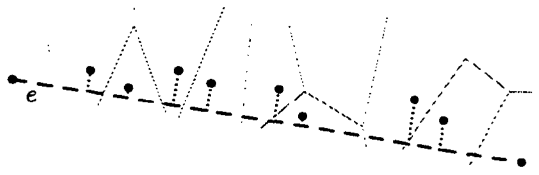

Figure 2: The order of projections onto e

OBSERVATION 2.1. *Let e be an edge of $ch(P)$. The order of the Voronoi cells that intersect e is the same as the order of the projections of the corresponding sites perpendicularly onto e.*

Our data structure consists of three parts. First, we store the Voronoi diagram for P, with a point location structure so that we can report the closest point to segment endpoints that lie outside $ch(P)$. Second, we store the convex hull $ch(P)$ in an array or binary search tree so that standard operations, such as tangents and intersections, can be performed in $O(\log n)$ time by binary search.

Third, for each edge e of the convex hull $ch(P)$, we form a balanced search tree whose leaves correspond to the sites whose Voronoi cells intersect edge e in order. Each internal node implicitly represents the convex hull of the points in its subtree. Before being more precise, it is important to note that the convex hull of a parent node can be obtained from the convex hulls of the two children by computing two common tangents.

LEMMA 2.2. *In the data structure, if two nodes are not related then their convex hulls are disjoint.*

Proof. Let e be the edge of $ch(P)$ for which the structure is built. By construction, the points in the subtrees of two unrelated nodes correspond to non-overlapping sequences of Voronoi cells along e. Observation 2.1 implies that their perpendicular projections onto e are also separable, so the convex hulls of two disjoint subtrees can be separated by a perpendicular to edge e.

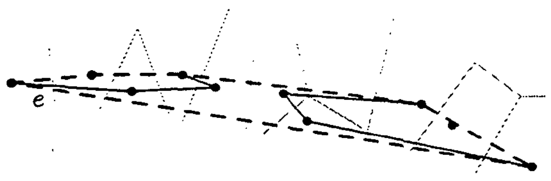

Figure 3: Children of the root in the data structure for e

Now we fully describe the structure built for edge e. The root node stores the convex hull of all points whose Voronoi cells intersect e as an ordered list of vertices. Any other internal node ν stores the portion of its convex hull that is not stored by its ancestors. If we consider a convex hull as an ordered sequence of point/tangent pairs, according to the "kinetic framework" of Guibas, Ramshaw, and Stolfi [12], then we can represent the pairs that are hidden by storing the chain between the common tangents from ν to its sibling, and the directions of these tangents. For example, the dashed lines in Figure 3 show the convex hull and the solid lines show the portions that would be stored with the two children of the root. This structure is closely related to the dynamic hull structure of Overmars and van Leeuwen [16], although we use it as a static structure.

Finally, we apply fractional cascading [7] to this structure so that if a tangent to node ν is known, then tangents of the same slope to the children of ν can be found in constant time. By way of remark, in applications where the queries are known in advance, the fractional cascading can be avoided: since the tangents that we will be concerned with are parallel or perpendicular to query segments, we can sort queries by slope and update these tangents by linear scans.

LEMMA 2.3. *The data structure for computing the nearest point in P to a query segment that does not intersect $ch(P)$ has linear size and can be computed in linear time from the Voronoi diagram of P.*

Proof. The Voronoi diagram and convex hull of P have linear size. Note that the convex hull $ch(P)$ can be extracted from the unbounded cells of the Voronoi diagram in linear time.

There are at most $2n-2$ intersections between the boundary of $ch(P)$ and edges of $VoD(P)$: if we traverse the boundary of $ch(P)$ and list a site whenever we enter its Voronoi cell, we obtain a circular Davenport-Schinzel sequence [18] that contains no *abab* substring. These intersection can be computed in linear time by walking around the boundary of $ch(P)$, through the cells of $VoD(P)$.

The total number of points stored in trees for edges of $ch(P)$ is, therefore, at most $3n-2$. Since each point is internal to a chain once, and only two points per node are chain boundaries, total storage is $O(n)$. It is not hard to compute the tangents in time proportional to the size of stored chains. Preprocessing for fractional cascading is also linear time.

We now describe how to answer a query for a segment s, which we assume is horizontal and lies below the convex hull $ch(P)$. The easy part is to locate the endpoints of s in the Voronoi diagram of P; the harder part is to search for candidates for the closest point to the interior of s.

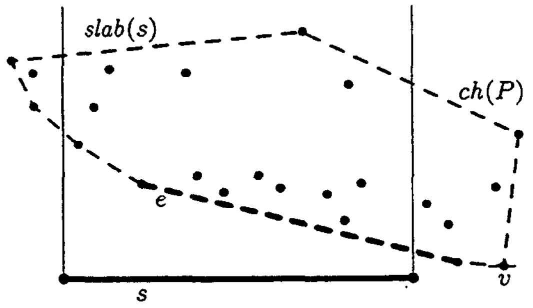

Figure 4: Deciding to search hull edge e

First, determine the vertex v of $ch(P)$ with tangent parallel to s; if v lies in $slab(s)$, then v is the candidate. If $slab(s)$ does not intersect the hull, then there is no candidate. Otherwise, the candidate will be in the structure associated with the convex hull edge that intersects the left or right boundary of the slab, whichever is closer to s. In Figure 4, the candidate will be in the structure for the edge e that intersects the right boundary of the slab.

Now we search down the levels of the tree structure constructed for edge e. We start from the root, and keep at most two nodes per level during our search.

During the search we maintain with node ν the leftmost point l_ν, rightmost point r_ν, and bottommost point b_ν of the convex hull represented by ν. At the root, these can be found by binary search; at subsequent nodes these are either inherited from the parents or obtained from our fractional cascading data structure in constant time apiece.

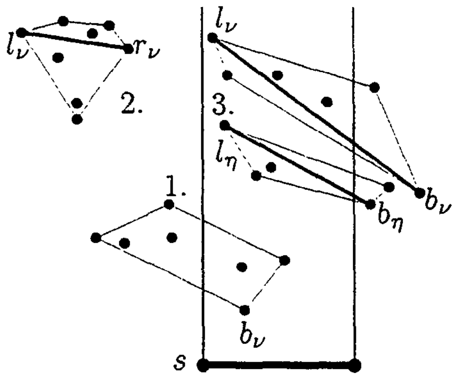

Figure 5: Discards 1–3

To carry out the search, replace the (at most two) nodes on the current level by their children, then apply the following "discarding" operations illustrated in Figure 5:
1. If b_ν is in $slab(s)$, then test if b_ν is the closest point to s found so far and discard ν.
2. If r_ν is left of $slab(s)$, or l_ν is right of $slab(s)$, then discard ν—none of its descendent lie in the slab.
3. If for two nodes, ν and η, the right boundary of $slab(s)$ intersects first $\overline{l_\eta b_\eta}$ and then $\overline{l_\nu b_\nu}$, as in Figure 5, then discard ν.
4. Symmetric to 3: if the left boundary of $slab(s)$ intersects first $\overline{b_\eta r_\eta}$ and then $\overline{b_\nu r_\nu}$, then discard ν.

It is clear that at most two nodes survive the discards: one with bottommost point to the left of $slab(s)$ and one with bottommost to the right. This allows the search to complete in $O(\log n)$ steps. We must show that the closest point to s is inspected.

LEMMA 2.4. *If the closest point to a segment s outside the convex hull ch(P) is closest to the interior of s, then the search described above finds it in O(log n) time.*

Proof. We search the data structure associated with a single hull edge e, as described above. Since only two nodes are kept at any level of the search tree, it is not hard to observe that constant time per level, or $O(\log n)$ total time, is sufficient.

For correctness, we must show that the discard operations never eliminate a closest point to s that lies in $slab(s)$. It is not hard to see that discards 1 and 2 are correct: In discard 1 point b_ν is the best candidate of node ν in $slab(s)$, and in discard 2 no point of ν is in $slab(s)$.

For discard 3, recall that we are searching the structure for the hull edge e, and that, by Observation 2.1,

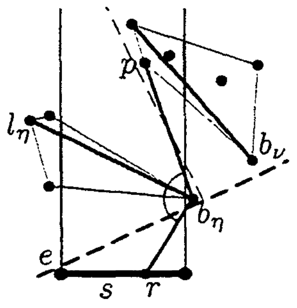

Figure 6: Obtuse angle $\angle pqr$, when $q = b_\eta$

the hulls for ν and η are separable by a line ℓ perpendicular to e. Lines e, ℓ, and the right boundary of $slab(s)$ define a right triangle that must contain a point q of η—we can take $q = b_\eta$ in Figure 6, where the triangle is to the right of $slab(s)$, or take $q = l_\eta$ if the triangle is to the left.

Now, let p be any point of ν in $slab(s)$, and let $r \in s$ be the closest point to p. The angle $\angle pqr$ is obtuse, since $\overline{pq}$ crosses e and qr crosses ℓ, thus,

$$d(p,s) = d(p,r) > d(q,r) \geq d(q,s).$$

We conclude that no point in ν can be closest to s. The proof for the fourth discard operation is, of course, symmetric.

2.2 Disjoint, general queries

Suppose that we are given a set P of n points and a set S of m disjoint line segments in the plane. Thus, we remove the restriction of the previous section that the queries be outside the convex hull, but impose the restriction that they be disjoint. In this section we show how to find the closest point to every segment in $O((n+m)\log^3(n+m))$ time by using a divide-and-conquer algorithm that uses the data structure of the previous section for queries outside the convex hull.

The algorithm proceeds as follows. Compute the convex hull, $H = ch(P)$, and partition the set of segments S into three sets:

- S_0 consists of all segments that lie outside the convex hull H,
- S_1 consists of all segments having at least one endpoint inside the convex hull H, and
- S_2 consists of all segments that intersect the boundary of the convex hull H in two points.

Build the data structure of the previous section for P and, for each segment in S_0, query for the closest point in P. Handle the segments in $S_1 \cup S_2$ by partitioning the points P into sets P' and P'' and recursively call the algorithm for the pairs $(P', S_1 \cup S_2)$ and $(P'', S_1 \cup S_2)$. For a segment s in $S_2 \cup S_1$, we return the closer of the candidate points returned by the recursive calls.

It should be clear that the algorithm will return a closest point to every segment. The key to efficiency is to partition P into a subsets that have disjoint convex hulls, and for which each segment of S_2 intersects at most one of hulls. We will actually form three subsets, although we do so in such a way that we can cut off one at a time so that we can still use a binary tree for our recursion.

LEMMA 2.5. *Given a set of n points P and m segments S_2 that each intersect the boundary of the convex hull $ch(P)$ in two points, we can partition P into three subsets whose sizes are bounded by $\lceil n/2 \rceil$, whose convex hulls are disjoint, and for which any segment in S_2 intersects at most one of these hulls. $O((n+m)\log m)$ time is sufficient if the points are given in lexicographic order.*

Proof. If we cut through the convex hull $ch(P)$ by any line segment, we partition the points into two sets with disjoint hulls. If the cut does not intersect a segment of S_2, then we also have the property that each segment in S_2 intersects at most one hull. We therefore concentrate on bounding the sizes of the sets. Since it is possible that no single segment bisects P and avoids S_2, we may first cut along a segment of $s \in S_2$ to make sure that we can cut what remains without crossing s.

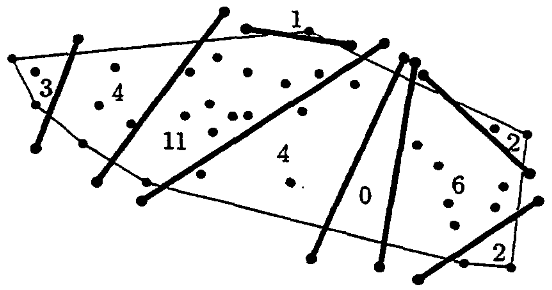

Figure 7: Segments of S_2 partition P

The segments of S_2 partition the interior of the convex hull $ch(P)$ as in Figure 7; the dual graph is a tree, whose vertices are the regions of the partition and edges join adjacent regions. We locate the points of P in these regions and assign a *weight* to each vertex in the dual graph equal to the number of points of P in the corresponding region. Note that some weights may be zero, and that they sum to n.

The dual tree has a *centroid vertex* whose removal leaves no connected component with weight greater than $n/2$; let C be the convex region that corresponds to the centroid, and let $\mathcal{S} \subset S_2$ be the segments that bound C, which correspond to edges incident to the centroid. Note that cutting $ch(P)$ along any $s \in \mathcal{S}$ guarantees that the fragment not containing C has at most $n/2$ points; if for some cut the other fragment has $\lceil n/2 \rceil$ points, we take it.

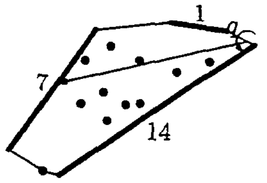

Figure 8: Partitioning C

To decide where to cut, we choose a "pivot" point q on the boundary of C that is also on the boundary of $ch(P)$. By a symbolic perturbation, we may assume that no two points of P lie on a line through q. Now, sweep a line ℓ through q across C. Let L_ℓ denote the *weight to the left of* ℓ, which is the weight of the boundary segments of $\mathcal{S}$ and the number of points in C that lie completely to the left of ℓ. Let R_ℓ denote the *weight to the right of* ℓ, which is defined similarly except that any point on the line is counted as lying on the right. The difference $n - (L_\ell + R_\ell)$ is always the weight of the line segment in $\mathcal{S}$ that ℓ intersects. In Figure 8, $L_\ell = 22$ and $R_\ell = 4$.

If both L_ℓ and R_ℓ are at most $\lceil n/2 \rceil$, we partition $ch(P)$ as follows: if ℓ intersects along a segment $s \in \mathcal{S}$, we first cut along s. Then we cut along ℓ. Neither of these cuts a segment of S_2, and all fragments have the proper size. What remains is to show that such a line ℓ exists.

As ℓ sweeps counter-clockwise, R_ℓ goes from zero to greater than $\lceil n/2 \rceil$: passing a point decreases L_ℓ and increases R_ℓ by one, and passing a segment of $\mathcal{S}$ adds its weight to R_ℓ. In either case, just before R_ℓ becomes greater than $\lceil n/2 \rceil$, we know that $L_\ell \le \lceil n/2 \rceil$.

For the running time, if the points are sorted, then convex hulls can be computed in linear time. Intersecting segments with the convex hull, sorting, and locating the points among the regions takes $O((m+n)\log m)$ time. Centroid computation can be done by a greedy algorithm in $O(m)$ time. The sweep to find ℓ is easy to do in $O((m+n)\log(m+n))$, and can be done in linear time.

To analyze the running time of our recursive algorithm, we consider a *recursion tree* T in which a node ν corresponds to a recursive call of the algorithm. We charge non-recursive computation against points and segments involved on each level; the total computation will be the sum of all charges to all nodes in T.

Denote the sets of points and segments assigned to node ν by $P(\nu)$ and $S(\nu)$, respectively. In a similar manner, let $S_i(\nu)$, for $i = 0, 1, 2$, be the sets of segments outside hull $ch(P(\nu))$, with an endpoint inside, and intersecting hull $ch(P(\nu))$ twice.

By the partitioning of P, the recursion tree T has several properties. For points, the following are relevant:

- T has depth $O(\log n)$.
- A point appears in a set $P(\nu)$ for at most one node per level.

Thus, the total number of points in the tree is $O(n \log n)$.

Now, consider the charges that can be applied against points. Each point is involved in partitioning according to Lemma 2.5, for which it charged $O(\log m)$, and in construction of the query data structure of Lemma 2.3, for which $O(\log n)$ is certainly enough. Thus, the total charges against points are $O(n \log^2 n + n \log n \log m)$.

For segments, there are additional relevant properties:

- For two nodes ν and η that are not on a common path to the root, the convex hulls $ch(P(\nu))$ and $ch(P(\eta))$ are disjoint.
- If a segment $s \in S_2(\nu)$, then s is in S_2 for at most one child of ν.
- If a segment $s \in S_0(\nu)$ then s does not appear in any descendant of ν.

The first property implies that a segment appears in S_1 sets on the two paths to its endpoints; that is, there are $O(m \log n)$ segments in all S_1 sets. Each s in an S_1 set can lead to an s in an S_2 set in a child; by the second this can lead to a path on which s is in S_2 sets down the tree. If this occurs at every level, then there are at most $O(m \log^2 n)$ segments in S_2 sets. Finally, every segment in an S_0 more than once is there because a parent was in an S_1 or S_2, which says that the total number of segments in S_0 sets is also $O(m \log^2 n)$.

The charges against segments are for queries according to Lemma 2.4, which is $O(\log n)$ each. This gives a total of $O(m \log^3 n)$ in the entire recursion tree, and completes the proof of the following theorem.

THEOREM 2.2. *Given n points and m disjoint segments in the plane, the closest point to every segment can be computed in $O(m \log^3 n + n \log^2 n + m \log m)$ time.*

2.3 Intersecting, general queries

For completeness, we note that data structures can be developed for the general problem: locating the closest point to query segments that may intersect. We do not into precise detail because the technology is more standard and the running times are asymptotically slower.

As mentioned in the introduction, the general problem can be used to solve Hopcroft's problem: given n points and n lines, does any point lie on any line. Erickson [10] has shown that any algorithm that can be implemented in a computational model based on partition trees, which includes our algorithms, must take $\Omega(n^{4/3})$ time to solve Hopcroft's problem.

When the points are given in advance and the query segments are given on-line, then we can build a spanning path with $\sqrt{n}$ stabbing number for the points—that is, a path that intersects any query line in at most $\sqrt{n}$ edges [1, 2, 9]. We build the query structures of Subsection 2.1 according to a balanced merge. Then, for any query segment s, we use the line through s to cut the path into at most $\sqrt{n}$ fragments, each of which is covered by $\log n$ query structures for which s is outside the convex hull. This would achieve $O(\sqrt{n} \log^k n)$ query time per segment, for some constant k.

When both points and segments are given in advance, then Agarwal and Procopiu (personal communication) have an algorithm that attains $O(n^{4/3} \log^k n)$ time in total. Such results can be obtained in much the same way that algorithms for Hopcroft's problem are obtained [1, 2, 9]—by using random sampling to reduce the problem to queries outside convex hulls, which can be answered by our query structure of Subsection 2.1.

3 Nearest foreign neighbors and Hausdorff distance for disjoint segments

We note that theorem 2.2 allows us, in $O(n \log^3 n)$ time, to solve the Hausdorff distance for sets of disjoint red segments and disjoint blue segments, and to solve the nearest foreign neighbor problem for disjoint segments. We begin with a simple lemma for nearest red neighbors to blue segments.

LEMMA 3.1. *Given n disjoint red segments and n disjoint blue segments in the plane, the nearest red neighbor for each blue segment can be computed in $O(n \log^3 n)$ time.*

Proof. Note that this problem is asymmetric—every blue segment must discover its nearest red segment, but not the other way around.

The minimum distance between a red and a blue segment is realized in one of three ways: by the intersection of a red and a blue segment, by a blue endpoint with its closest red segment, or by a red endpoint that is closest to the relative interior of the blue segment. (When the interior of two disjoint segments realize the minimum distance, then the segments are parallel and the distance is also realized at an endpoint.)

Those blue segments that intersect red segments can be found by a modification of the Bentley-Ottmann [5] line-sweep algorithm for segment intersection. Whenever an intersection is detected it must be between a red and a blue segment, make the red the nearest neighbor of the blue and delete the blue. This takes $O(n \log n)$ time.

The closest red segment for each blue endpoint can be found by computing the Voronoi diagram of the red segments and quering with blue endpoints in $O(n \log n)$ time.

Finally, the closest red point for each blue segment can be found by Theorem 2.2 in $O(n \log^3 n)$ time.

These computations give at most four candidates for the closest red to each blue segment—taking the minimum completes the computation.

The Hausdorff distance is the maximum of the distances from red to blue and from blue to red.

COROLLARY 3.1. (HAUSDORFF DISTANCE) *Given n disjoint red segments and n disjoint blue segments in the plane, the Hausdorff distance between red and blue sets can be computed in $O(n \log^3 n)$ time.*

The nearest foreign neighbor problem for disjoint segments can also be reduced to instances of the red neighbor problem for segments.

COROLLARY 3.2. (NEAREST FOREIGN NEIGHBORS) *Given n disjoint, colored line segments in the plane, one can compute for every segment the closest neighbor of a different color in $O(n \log^3 n)$ time.*

Proof. Compute the Voronoi diagram of all line segments. Now, choose one of the colors: blue, for example.

We claim that the Voronoi cell for a nearest neighbor to a blue segment will be adjacent to some blue Voronoi cell. Consider a shortest path from a blue segment b to its nearest neighbor s, but trace it starting from s. If this path left the Voronoi cell of S at the boundary of the cell for a segment $t \neq b$, then there is an equal length path from b to t that bends at this boundary. Thus, either t is also blue, or s was not the nearest neigbhor.

Thus, we can form a red neighbor subproblem for the blue segments by taking all blue segments and taking only the sites of neighboring cells as the red segments. We form similar subproblems for each of the other colors.

To bound the total size of all subproblems we can count all neighbor relations in the Voronoi diagram, which is equivalent to determining the sum of the degrees in its dual graph when multiple edges are collapsed. Since the dual is a planar graph, this is at most $6n - 12$, and the total time is bounded by $O(n \log^3 n)$.

Note that if we relax the disjointness restrictions on any of the above problems, we can use them to solve Hopcroft's problem. For example, if we allow intersecting segments of different colors in the nearest foreign neighbors problem, then we could color each point and line a different color. The nearest foreign neighbors for the points would tell us if any point was on any line.

References

[1] Pankaj K. Agarwal. Partitioning arrangements of lines II: Applications. *Discrete & Computational Geometry*, 5:533–573, 1990.

[2] Pankaj K. Agarwal. *Intersection and decomposition algorithms for planar arrangements*. Cambridge University Press, 1991.

[3] A. Aggarwal, H. Edelsbrunner, P. Raghavan, and P. Tiwari. Optimal time bounds for some proximity problems in the plane. *Inform. Process. Lett.*, 42(1):55–60, 1992.

[4] F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23:345–405, 1991.

[5] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[6] Bernard Chazelle. An algorithm for segment-dragging and its implementation. *Algorithmica*, 3:205–221, 1988.

[7] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[8] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[9] Herbert Edelsbrunner, Leonidas Guibas, John Hershberger, Raimund Seidel, Micha Sharir, Jack Snoeyink, and Emo Welzl. Implicitly representing arrangements of lines or segments. *Discrete & Computational Geometry*, 4:433–466, 1989.

[10] Jeff Erickson. New lower bounds for Hopcroft's problem. *Discrete Comput. Geom.*, 16:389–418, 1996.

[11] T. Graf and K. Hinrichs. Algorithms for proximity problems on colored point sets. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 420–425, 1993.

[12] Leonidas J. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 100–111, 1983.

[13] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 1973.

[14] M. McAllister, D. Kirkpatrick, and J. Snoeyink. A compact piecewise-linear Voronoi diagram for convex sites in the plane. *Discrete Comput. Geom.*, 15:73–105, 1996.

[15] Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley & Sons, Chichester, UK, 1992.

[16] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23:166–204, 1981.

[17] M. I. Shamos and D. Hoey. Closest-point problems. In *Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 151–162, 1975.

[18] Micha Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, New York, 1995.

[19] C. K. Yap. An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments. *Discrete Comput. Geom.*, 2:365–393, 1987.