Computer-Aided Design 189 (2025) 103947

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad

P2Seg: Distance query from point to segments

Jiantao Song [a], Rui Xu [b], Wensong Wang [a], Shiqing Xin [a,*], Shuangmin Chen [c,d], Jiaye Wang [a], Taku Komura [b], Wenping Wang [e], Changhe Tu [a]

[a] Shandong University, Jinan, China
[b] The University of Hong Kong, Hong Kong, China
[c] Qingdao University of Science and Technology, Qingdao, China
[d] Shandong Key Laboratory of Deep Sea Equipment Intelligent Networking, Qingdao, China
[e] Texas A&M University, College Station, USA

ARTICLE INFO

Keywords:
Computational geometry
Distance query
Voronoi diagram
KD tree
Bounding volume hierarchy (BVH)

ABSTRACT

Querying the nearest distance from a point to n line segments in 2D is a textbook problem in computational geometry. This paper presents P2Seg, a novel algorithmic strategy that transforms the intricate problem into an accessible linear traversal. Our method precomputes a KD tree and a Voronoi diagram for the site collection S, where S refers to the endpoints of all line segments. Obviously, for a query point q, the nearest site s_i provides a crucial clue for pinpointing the nearest line segment, i.e., the pairing (q, s_i) effectively reduces the search from n line segments to a limited number, represented as $L(q, s_i)$. The key idea of this paper is driven by an insightful observation: if the ray s_iq intersects with s_i's Voronoi cell at a point, say q', then $L(q, s_i)$ is a subset of $L(q', s_i)$. This suggests that preprocessing efforts can be substantially minimized by focusing solely on scenarios where the query point lies on the Voronoi edges, which are fundamentally one-dimensional. We further prove that the challenge of locating the nearest line segment from $L(q', s_i)$ can be distilled down to a simple linear traversal. Testing on datasets of varying complexities shows that P2Seg significantly outperforms state-of-the-art techniques. For example, in scenarios involving 10K segments with an average length of 0.5, our method runs 2.2 times faster than P2M and 60 times faster than AABB, as illustrated in the teaser figure.

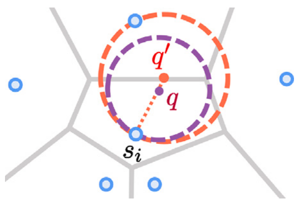
1. Introduction

Given a collection of line segments L in a two-dimensional plane, querying the nearest distance from a user-specified point to L is a fundamental research problem in computational geometry. This problem finds widespread applications across various domains, such as geographic information systems [3], computer vision [4,5], and geometry processing [2].

Traditional methods like the KD tree (KDT) [5] and bounding volume hierarchy (BVH) [6] are effective for small-scale datasets without complicated intersections between line segments but struggle with increasing data sizes and complexities, such as significant overlap and intersections. These challenges can make them as inefficient as brute-force searching, highlighting the need for more efficient algorithms for rapid distance queries.

This paper introduces P2Seg, a novel algorithmic solution that streamlines this complex issue into an accessible linear traversal. Suppose that there are n line segments, it is evident that point-based proximity searches are simpler than point-to-segment queries. Hence, we precompute a KD tree and a Voronoi diagram of S to expedite the query process, where S includes the endpoints of line segments. For any query point q, the KD tree can quickly locate q's nearest site in S. The identified site, s_i, suggests that q resides within the Voronoi cell of s_i, offering valuable clues for identifying the nearest line segment. Simply speaking, the pair (q, s_i) is more informative than a single q, and thus effectively reduces the search scope from n line segments to a more manageable subset, denoted as $L(q, s_i)$.

A pivotal insight of this paper is the observation that if the ray s_iq intersects s_i's Voronoi cell at a specific point, say q', then $L(q, s_i)$ falls within $L(q', s_i)$. This indicates that preprocessing efforts can be substantially minimized by concentrating on scenarios where the query point



is positioned on the Voronoi edges, inherently one-dimensional. As depicted in the inset figure, the circle centered at q suggests that line segments intersecting this circle could offer a distance closer than $\|qs_i\|$. Notably, the circle lies entirely within the circle centered at q', indicating that $L(q, s_i) \subset L(q', s_i)$. See Section 3.2 for details.

* Corresponding author.

https://doi.org/10.1016/j.cad.2025.103947
Received 10 June 2025; Received in revised form 23 July 2025; Accepted 10 August 2025
Available online 20 August 2025

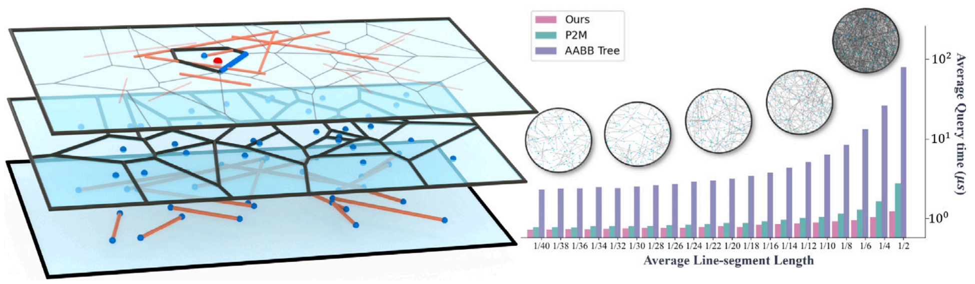


Fig. 1. This paper focuses on rapidly querying the nearest distance from a point to a collection of line segments in 2D. On the left, from bottom to top, are the line segments, the Voronoi diagram of the endpoints, and the candidate line segments associated with the query point. On the right, the timing plot (μs) shows that our algorithm significantly outperforms both AABB [1] and P2M [2] by several times, where we randomly generate 10K line segments in $[0,1] \times [0,1]$. It is important to note that the horizontal axis represents the average line-segment length and the vertical axis is on a log10 scale.

Furthermore, we analyze how q' traverses the Voronoi edge, which reveals three potential situations: (1) a candidate line segment consistently remains within $L(q', s_i)$, (2) it is part of $L(q', s_i)$ temporarily before exiting, and (3) it cycles through entering, exiting, and then re-entering $L(q', s_i)$. This fascinating finding simplifies the challenge of identifying the nearest line segment from $L(q', s_i)$ to a straightforward linear traversal. See Section 3.4. As shown in Fig. 1, on the dataset with 10K line segments, our query efficiency advantage becomes increasingly evident as the average length of the segments increases, reaching 2.2 times that of P2M and 60 times that of AABB.

In summary, our contributions are threefold:

1. We introduce a novel algorithmic strategy for dealing with the point-to-segments query. It achieves more than twice the performance of the latest distance query methods.
2. Our preprocessing is inspired by the observation that it is adequate to consider the query point positioned on the Voronoi edges—defined by S, the endpoints of the line segments.
3. We prove that a line segment l within $L(q', s_i)$ can undergo one of three scenarios as q' moves along the corresponding Voronoi edge. This insight significantly simplifies the point-to-segments query to an easily manageable linear traversal.

2. Related work

Two topics relevant to the theme of this paper are the nearest proximity search and the Voronoi Diagram.

2.1. Nearest proximity search

Precomputation and query stages. Given a collection of primitives in a k-dimensional space, nearest proximity search algorithms aim to find the primitive (such as points, line segments, or triangles) closest to a given input point. These searches can be efficiently performed by organizing the primitives within a tree structure [7,8], significantly narrowing the search space during the query phase. Nearest proximity search algorithms primarily consist of two stages: constructing the tree and conducting queries using this structure. The trees used to organize primitives are categorized into two types based on their partitioning methods: space partitioning trees and bounding volume hierarchies. Each approach facilitates efficient nearest proximity searches through distinct strategies for space partitioning and primitive arrangement.

Space partitioning trees. In geometry, spatial partitioning is the process of dividing the entire space (commonly Euclidean) into two or more disjoint subsets. In practical use, spatial partitioning systems are typically hierarchical, dividing the space into two or more sections at each level through planar partitions. This process is recursively applied to every newly created region, organizing them into a tree-like structure known as a spatial partitioning tree.

During the query phase, the algorithm conducts a downward search, determined by the query point's position relative to the splitting hyperplane. It consults the tree's stored data to identify primitives potentially offering closer proximity, thus updating the nearest distance identified so far. Most tree-based search structures feature a backtracking mechanism to avoid overlooking any primitives that could provide a closer distance during the query.

The KD tree (KDT) is recognized as a classic Binary Search Tree (BST) acclaimed for its effectiveness in nearest-neighbor searches, boasting logarithmic time complexity and linear space efficiency. Its computational advantages have made it a preferred tool for enhancing various graphics algorithms and applications, including data visualization [5], distance queries [4,9], particle tracing [10], isosurface rendering [11,12], and ray tracing [13].

Additionally, numerous other space-partitioning data structures are viable for nearest proximity searches, such as the R-tree [14–17], ball-tree [18], A-tree [19], BD-tree [20], and SR-tree [21]. These structures typically employ partitioning strategies similar to those of KDTs, taking into account data distribution, preprocessing efficiency, and query performance. Owing to their effectiveness in balancing these considerations, KDTs have emerged as the most classic and broadly utilized BST in the field.

Bounding volume hierarchy. Similar to the BST, a Bounding Volume Hierarchy (BVH) [6] is a hierarchical structure designed to manage a collection of geometric primitives. In a BVH, leaf nodes encapsulate geometric primitives, while non-leaf nodes contain bounding volumes that encompass subsets of these primitives. The construction of a BVH can adopt various approaches, including top-down, bottom-up, or incremental insertion methods, ultimately creating a tree structure topped with a bounding volume. BVHs are extensively applied in fields such as distance queries [22], ray tracing [23], and collision detection [24,25].

Regarding BVHs, choosing the appropriate type of bounding volume is critical, as it necessitates a compromise between simplicity and accuracy. Simple bounding volumes like Axis-Aligned Bounding Boxes (AABB) [26,27] and bounding spheres [28–30] offer quick intersection tests but may not snugly fit the geometric primitives. In contrast, more complex bounding volumes like Oriented Bounding Boxes (OBB) [31], k-DOPs (Discrete Oriented Polytopes) [32], zonotopes [33], and others [34–37] aim for a tighter fit. This selection hinges on the specific requirements of the application and the desired balance between ease of computation and the precision of fit. Moreover, the choice of bounding volume is influenced by the nature of the primitives involved. For instance, spherical bounding volumes might not provide a tight fit for line segments as primitives, potentially diminishing the effectiveness of the acceleration in the query process.

2.2. Voronoi diagram

The Voronoi diagram serves as a spatial partitioning technique by assigning points in space to their nearest primitives based on distance.

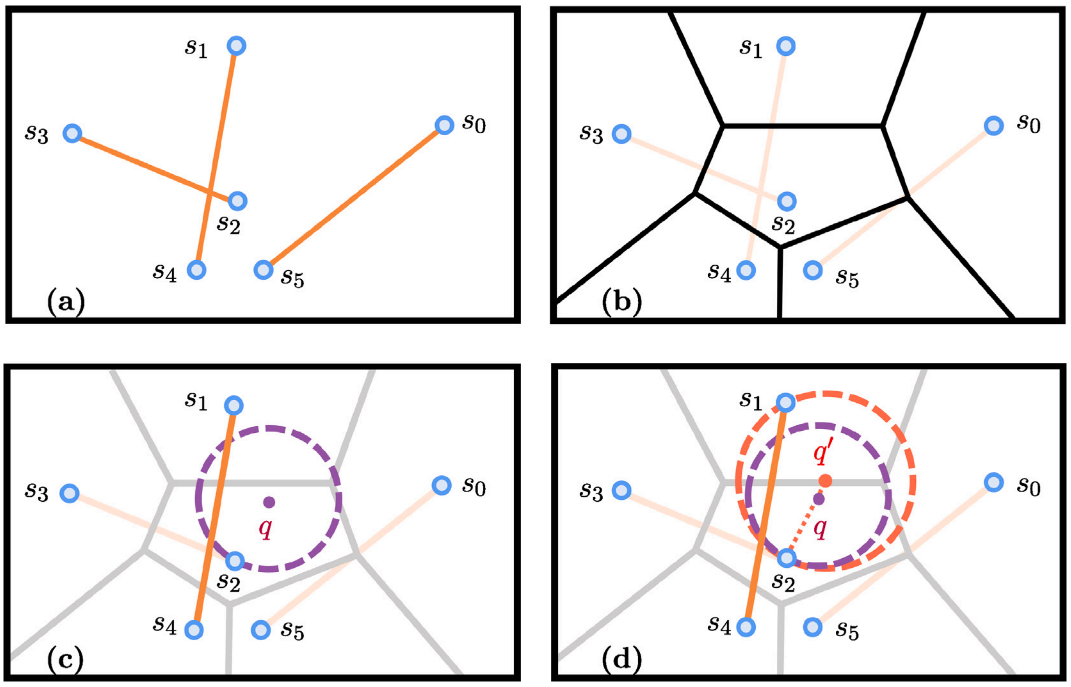


Fig. 2. During the processing phase, it is practical to consider the query point as situated on the Voronoi edges, with the Voronoi diagram constructed from the endpoints of the line segments. (a) a set of line segments L. (b) the Voronoi diagram of the endpoints. (c) a query point q located in s_2's Voronoi cell, identified using the KDT. q's search circle, illustrated in purple, denotes that line segments crossing this circle are potential candidates for providing the actual closest distance. (d) Extending q to q' — where the ray s_2q intersects s_2's cell — defines a larger search circle that fully encompasses q's search circle. This suggests that the candidate set of line segments for q' includes a broader selection of line segments than that for q.

Compared to KDT, Voronoi diagrams provide a more accurate depiction of the proximity between query points and various primitives. However, the construction and querying of Voronoi diagrams are challenging, especially when dealing with generators that are not simple points.

In [2], the approach explores the use of Voronoi diagram for triangle primitives, albeit simplifying it to the more conventional point-based Voronoi diagram to facilitate querying the distance between a point and a polygonal surface. During the query phase, the algorithm first determines the closest mesh vertex to the query point, then consults the vertex's interception table to identify candidate triangles that may offer the closest distance. In addition to processing triangle meshes, the work they referenced is also capable of efficiently managing nearest-distance queries from a point to a collection of line segments. Despite its utility, the preprocessing involved in this method is complex and demands considerable time.

3. Our method

3.1. Problem definition

In a two-dimensional space, the task of finding the nearest distance from a given point q to a set of line segments $L = \{l_i\}_{i=1}^{n}$ can be defined as

$$d_{\min}(q, L) = \min_{l_i \in L} d(q, l_i).$$

Assuming without loss of generality that the n line segments collectively have $2n$ endpoints, constituting a set of sites $S = \{s_i \mid 1 \le i \le 2n\}$.

3.2. Insight

Let s_i be the nearest site for the query point q, with their distance being $\|s_i q\|$. As depicted in Fig. 2(c), we introduce a circle Circ(q) centered at point q and with a radius equal to $\|s_i q\|$. In the following, we name it a *search circle*. It is clear that no sites lie within the interior of Circ(q). If there exists a line segment l_i such that $d(q, l_i) \le d(q, s_i)$, then the segment l_i must intersect the search circle Circ(q). Consequently, we use L_q to denote the line segments that intersect the search circle Circ(q).

The primary challenge in the point-to-segments distance query problem is that q can be any point in 2D space, presenting infinitely many possibilities. However, since finding the nearest site s_i can be efficiently performed using the KDT, it is suitable to start with such a rapid search during the query stage. Obviously, the pair (q, s_i) offers significantly more clues than the solitary point q. In light of this, a straightforward preprocessing might proceed as follows: for each site, s_i, enumerate all possible line segments for which there exists a query point q such that q's search circle intersects the line segment and s_i is identified as the nearest site to q. It can reduce the search space to some extent but may still be inefficient because there may still exist too many relevant line segments.

Extending $s_i q$ to s_i's cell boundary. We further narrow the search space motivated by a noteworthy observation. By extending q to q' along the ray $s_i q$, we encounter an intersection point at the boundary of s_i's Voronoi cell, with q' situated on a Voronoi edge of s_i's cell. Given that s_i, q, q' are colinear, q' induces a larger search circle than q, Circ(q') completely encloses Circ(q), i.e., Circ(q) $\subset$ Circ(q'). This indicates that the candidate set of line segments for q' includes a broader selection of line segments than that for q. Therefore, during the preprocessing stage, it is adequate to consider q as located on Voronoi edges, fundamentally simplifying the search space to one-dimensional.

Variation in search circle along voronoi edge. As depicted in Fig. 3, there exists a Voronoi edge $e_i = v_s v_t$ within the Voronoi diagram. Both v_s and v_t define a search circle, as illustrated by the two dotted blue circles. Given that v_s and v_t are Voronoi vertices, their respective search circles pass through at least three sites but do not contain any sites within their interiors.

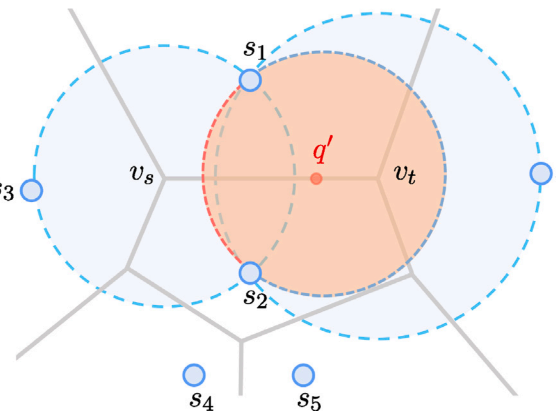


Fig. 3. v_s and v_t each define a search circle. A point q' positioned on the segment v_sv_t defines its own search circle. We have Circ(q') ⊂ Circ(v_s)∪Circ(v_t).

It is known that the segment v_sv_t is the bisector of two sites. In Fig. 3, v_sv_t is the common boundary of s_1's cell and s_2's cell. It can be concluded that both Circ(v_s) and Circ(v_t) pass through both s_1 and s_2. Therefore, s_1 and s_2 are the intersections between Circ(v_s) and Circ(v_t).

Let q' be situated on v_sv_t. q''s nearest site is both s_1 and s_2, with $\|q's_1\| = \|q's_2\|$. Therefore, the search circle of q', i.e., Circ(q'), must pass through both s_1 and s_2, without any sites in the interior of Circ(q'). To this end, s_1 and s_2 are a pair of common points for Circ(v_s), Circ(v_t) and Circ(q').

The search circle Circ(q') is split into two parts by s_1 and s_2. The left part is totally inside Circ(v_s) while the right part is totally inside Circ(v_t). Therefore, it is easy to conclude that Circ(v_t) must be contained by the union of Circ(v_s) and Circ(v_t):

Theorem 3.1. *For any point q' on the Voronoi edge $e_i = v_sv_t$, Circ(q') ⊂ Circ(v_s) ∪ Circ(v_t).*

It suggests that preprocessing should begin by identifying the candidate line segments for each Voronoi vertex, followed by addressing the scenario where the query point is located on a Voronoi edge.

3.3. Pipeline

As illustrated in Fig. 4, the algorithm is comprised of two main stages: the preprocessing stage and the query stage.

ALGORITHM 1: Preprocessing

```
Input: A collection of line segments $L$, whose endpoints
       constitute $S$.
Construct the Voronoi diagram $\mathcal{V}$ of $S$;
for each Voronoi vertex $v_i \in \mathcal{V}$ do
    Identify all segments $l_i$ satisfying $d(v_i, l) \le \min\{d(v_i, s_i)\}$;
    Store them in $L_{v_i}$;
end
for each Voronoi edge $e = v_sv_t \in \mathcal{V}$ do
    Call ArrangeAngleInterval($e_i$);
end
Construct the KDT of $S$;
```

As shown as Alg. 1, the preprocessing stage is further broken down into four steps. Initially, with the given input line segments, we construct a Voronoi Diagram using the endpoints of these segments. Subsequently, for each Voronoi vertex v_i, we determine the line segments that intersect v_i's search circle, which is centered at v_i and passes through the nearest three sites. These candidate line segments are then stored in L_{v_i}. Next, for each Voronoi edge $e_i = v_sv_t$, we identify critical cases where a line segment either emerges as a candidate or is no longer considered one. And for each Voronoi edge $e_i = v_sv_t$, we simulate the movement of the query point from v_s to v_t, analyzing the variation in the set of candidate line segments. This analysis yields three sorted angle interval arrays. The three angular intervals can be subdivided into eight distinct cases, which will be discussed in detail in the following section. Finally, we built a KDT of S.

During the query stage, the nearest site to the query point q is first identified using the KDT. After that, by extending q in the direction of s_iq, we find an intersection q' with the boundary of s_i's cell. Finally, we linearly traverse the candidate line segments based on the radial angle of s_iq' to find the line segment offering the closest distance.

3.4. Critical points

Formulation. Consider the situation as Fig. 5 shows. Suppose p is a point located on the Voronoi edge v_sv_t, the common boundary of s_1's cell and s_2's cell.

Let the two endpoints of the line segment l be s_i and s_j. We denote $\vec{v} = v_t - v_s$, $\vec{u} = v_s - s_2$, $\vec{m} = s_i - v_s$ and $\vec{s} = s_j - s_i$.

Based on the above discussion, l is deemed to undergo a critical situation if it is becoming a candidate or ceasing to be. Substituting $p = v_s + \lambda\vec{v}$ into the above formula, we obtain

$$d(s_2, v_s + \lambda\vec{v}) = d(l, v_s + \lambda\vec{v}).$$

Note that it is not impossible that one of the endpoints of l gives the closer distance due to the assumption that s_2 is the nearest site to q. Hence,

$$d(s_2, v_s + \lambda\vec{v}) = \|\vec{n}\|, \quad d(l, v_s + \lambda\vec{v}) = \frac{|(v_s + \lambda\vec{v} - s_i) \times \vec{s}|}{\|\vec{s}\|},$$

where $\vec{n} = v_s + \lambda\vec{v} - s_2$, and $\vec{w} = v_s + \lambda\vec{v} - s_i$. We further have

$$\|\vec{n}\| = \frac{|\vec{w} \times \vec{s}|}{\|\vec{s}\|},$$

or

$$\|\vec{n}\|^2\|\vec{s}\|^2 = |\vec{w} \times \vec{s}|^2,$$

with λ serving as the variable. The above equation has the following form:

$$f(\lambda) = A\lambda^2 + B\lambda + C = 0, \tag{1}$$

where

$$A = \|\vec{s}\|^2\|\vec{v}\|^2 - |\vec{v} \times \vec{s}|^2,$$
$$B = 2\|\vec{s}\|^2(\vec{u} \cdot \vec{v}) - 2|\vec{v} \times \vec{s}||\vec{m} \times \vec{s}|,$$
$$C = \|\vec{s}\|^2\|\vec{u}\|^2 - |\vec{m} \times \vec{s}|^2.$$

Classification of critical points. We now discuss the solutions to the quadratic equation, noting that the line segment l we consider must be within $L_{v_s} \cup L_{v_t}$. The equation may have 0, 1, or 2 roots, depending on the sign of $\Delta = B^2 - 4AC$. Obviously, a valid solution must satisfy $0 \le \lambda \le 1$. And the most complicated situation occurs when the equation has two valid roots. In this situation, l must be with in L_{v_s} and L_{v_t} simultaneously. When p moves from v_s to v_t, l cycles through entering Circ(p), exiting from Circ(p), and then re-entering Circ(p).

To be more detailed, there are eight different situations, as illustrated in Fig. 6.

1. If $l \in L_{v_s} \cap L_{v_t}$ and $\Delta \le 0$ or $\Delta > 0$ and none of the two roots satisfies $0 \le \lambda \le 1$, it can be observed that like Fig. 6(a,b,c), for any point p on v_sv_t, we have $d(p, s_2) \le d(p, l)$. l remains within the candidate line-segment set L_p for any $p \in v_sv_t$.
2. If $l \in L_{v_s} \cap L_{v_t}$ and $\Delta > 0$ and both roots satisfy $0 \le \lambda_1 \le \lambda_2 \le 1$, as illustrated in Fig. 6(d). In this situation, both L_{v_s} and L_{v_t} contain the line segment l, and when p moves from v_s to v_t, l enters Circ(p), exits from Circ(p), and then re-enters Circ(p).

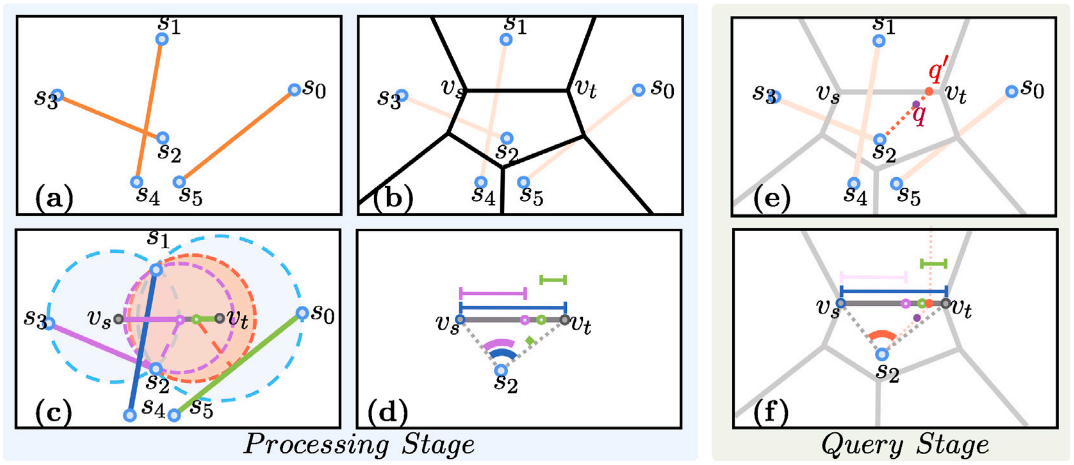


Fig. 4. Preprocessing (a–d) and query (e–f) phases of our approach are outlined as follows. (a) The input line segments. (b) Construction of the Voronoi diagram of the endpoints of the line segments. (c) Computation of critical points on the Voronoi edge $v_s v_t$, as shown in the diagram. This step aims to identify critical cases where a line segment either becomes a candidate or ceases to be one. (d) By allowing the query point to move from v_s to v_t, we examine the change of the set of candidate line segments, resulting in three sorted angle interval arrays. (e) Extend q along the direction of $s_i q$, yielding an intersection q' with the boundary of $s_i's$ cell. (f) Retrieve the candidate line segments according to the radial angle of $s_i q'$.

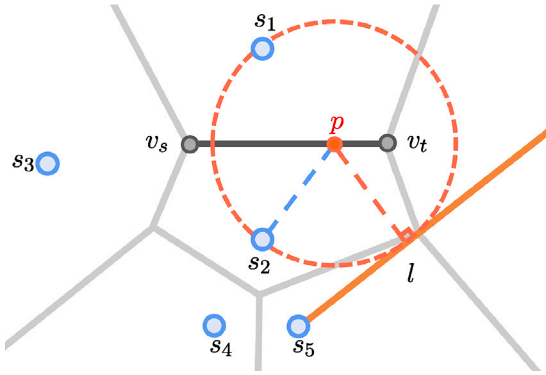


Fig. 5. p is said to be a critical point with regard to the line segment l if and only if Circ(p) is tangent to l.

3. If $l \in L_{v_s} \backslash L_{v_t}$ or $l \in L_{v_t} \backslash L_{v_s}$ and $\Delta \geq 0$ and only one root satisfies $0 \leq \lambda \leq 1$, as shown in Fig. 6(e,f,g,h). In this case, only one of L_{v_s} and L_{v_t} can contain l, and thus when p moves from v_s to v_t, l remains within L_p for a while and then exits, or enters Circ(p) until p arrives at v_t.

3.5. Search structure on Voronoi edge

Based on the eight situations in Fig. 6, for each line segment $l \in L_{v_s} \cup L_{v_t}$, whether l is within L_q has three possibilities: (1) l consistently remains within L_q, (2) it is part of L_q temporarily before exiting, or enters L_q until q arrives at v_t, and (3) l cycles through entering, exiting, and then re-entering L_q. For type #1, the points $p \in v_s v_t$ enabling L_q to contain l are identical to $v_s v_t$. For type #2, the points $p \in v_s v_t$ enabling L_q to contain l must span a continuous part of $v_s v_t$, with one of the endpoints aligning with v_s or v_t. For type #3, the points $p \in v_s v_t$ enabling L_q to contain l must span two intervals, each of which aligning with v_s or v_t. In implementation, we categorize the intervals into two types, depending on aligning with one or two of v_s and v_t.

Therefore, it suffices to maintain three sorted arrays for linear traversal queries. We detail the strategy as follows. As Fig. 7(a) shows, for any point p on the Voronoi edge, the position of p is represented by the radial angle of $s_i p$, with the angle of $s_i v_s$ being set to 0. Based on the above analysis, we store the radial angle intervals for $v_s v_t$, and each line segment may contribute to one or two angle intervals.

In Fig. 7(b), all angle intervals associated with $v_s v_t$ can be arranged in three sorted arrays, IntervalList$_s$, IntervalList$_t$, and IntervalList$_{st}$. It is noteworthy that all intervals align with at least one of v_s or v_t. The three arrays are respectively used to store (1) intervals aligning with v_s but not with v_t, (2) intervals aligning with v_t but not with v_s, and (3) intervals aligning with both v_s and v_t. During the query phase, the ray $s_i q$ can be extended to determine the intersecting Voronoi edge. Subsequently, the radial angle of $s_i q$ is calculated, and all angle intervals encompassing the query angle are retrieved, identifying the candidate line segments. See Fig. 7(c). We use Alg. 2 to depict the pseudo-code.

Query stage. In the query phase, the initial step involves using the KDT to identify the nearest site s_i to the query point q. Extending q along the ray $s_i q$ allows us to locate an intersection point q' at the boundary of s_i's Voronoi cell. Subsequently, we identify the Voronoi edge intersected by the ray $s_i q$ and compute the radial angle α of $s_i q'$ relative to the direction of $s_i v_s$.

Define L_q as an initially empty set of line segments. First, each line segment that contributes to IntervalList$_{st}$ is considered a potential candidate for this query and is added to L_q. Next, we proceed to search IntervalList$_s$ (in descending order) from the beginning until the intervals no longer encompass α, adding all useful candidates to L_q. Similarly, we search IntervalList$_t$ (in ascending order) from the start until the intervals no longer include α, incorporating these useful candidates into L_q. The final step involves examining the candidates in L_q, one by one, to determine which line segment offers the actual closest distance.

3.6. Special case

As Fig. 8 shows, we suppose that q is the query point and q' is a sufficiently distant point along the direction of $s_i q$. It holds true that q' has a larger set of line-segment candidates than q. Since the ray $s_i q$ extends towards infinity, s_i dominates an open Voronoi cell and thus is a vertex of the convex hull of S (the endpoints of the line segments). It can be further deduced that q must be located in the shadow area, bounded by the two rays parallel to the neighboring open Voronoi edges, respectively. To this end, we draw a straight line at s_i such that the line is orthogonal to $s_i q$. Since the convex hull and q are lying on

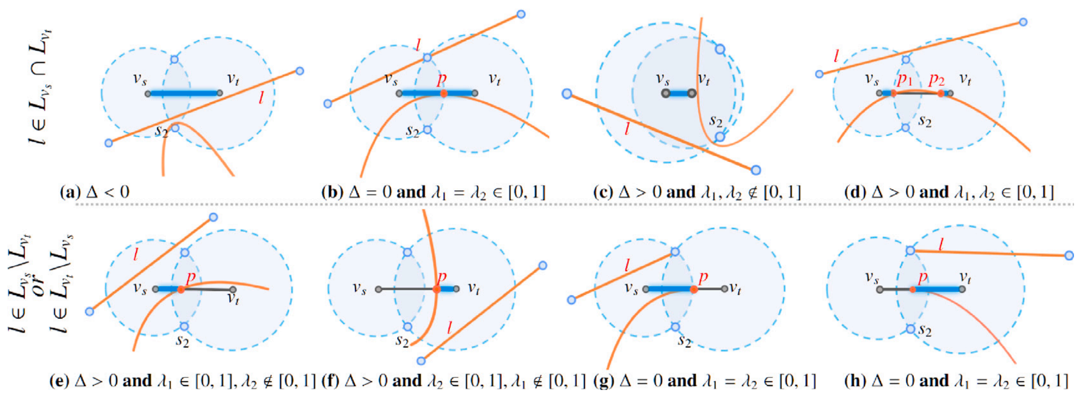

$l \in L_{v_s} \cap L_{v_t}$
$l \in L_{v_s} \setminus L_{v_t}$ or $l \in L_{v_t} \setminus L_{v_s}$
(a) $\Delta < 0$
(b) $\Delta = 0$ and $\lambda_1 = \lambda_2 \in [0,1]$
(c) $\Delta > 0$ and $\lambda_1, \lambda_2 \notin [0,1]$
(d) $\Delta > 0$ and $\lambda_1, \lambda_2 \in [0,1]$
(e) $\Delta > 0$ and $\lambda_1 \in [0,1], \lambda_2 \notin [0,1]$
(f) $\Delta > 0$ and $\lambda_2 \in [0,1], \lambda_1 \notin [0,1]$
(g) $\Delta = 0$ and $\lambda_1 = \lambda_2 \in [0,1]$
(h) $\Delta = 0$ and $\lambda_1 = \lambda_2 \in [0,1]$

Fig. 6. Segment $v_s v_t$ denotes a Voronoi diagram edge, s_2 represents a site, and the orange straight line l is the given input line. Every point on the orange curve is equidistant from the site s_2 and the line l. During the process that the query point p moves from v_s to v_t, there are eight situations when considering whether the line segment is within the candidate set: (1) a candidate line segment consistently intersects Circ(p), (2) it intersects Circ(p) initially and then exits, or enters Circ(p) at $p \in v_s v_t$ until p arrives at v_t, and (3) it cycles through entering, exiting, and then re-entering Circ(p). They can be further divided into 8 cases.

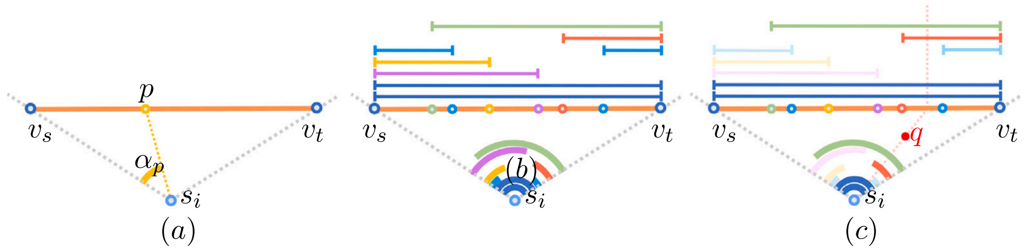


Fig. 7. It suffices to maintain three sorted arrays for linear traversal query. (a) For any point p on the Voronoi edge, the position of p is represented by the radial angle of $s_i p$, with the angle of $s_i v_s$ being set to 0. (b) For all candidate line segments associated with v_s and v_t, we create angle intervals, with each candidate line segment contributing to one or two intervals. A crucial preprocessing step involves organizing these angle intervals into three sorted arrays. It is noteworthy that all intervals align with at least one of v_s or v_t. (c) In the query stage, one can extend the ray $s_i q$ to identify which Voronoi edge is intersected and then compute the radial angle of $s_i q$, followed by retrieving all the angle intervals that contain the query angle, yielding the candidate line segments.

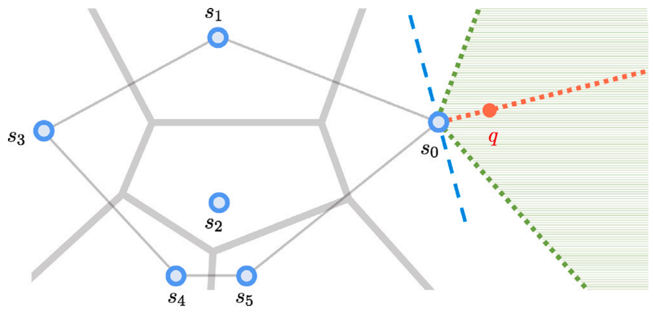


Fig. 8. If $s_i q$ extends towards infinity, then it can be concluded that s_i provides the actual closest distance, i.e., it is impossible to obtain a distance less than $\|s_i q\|$ between q and any other point in the line segments.

different sides of the straight line, we conclude that s_i is exactly the point that provides the actual closest distance.

4. Evaluation

We conducted all the experiments with an AMD Ryzen 9 7900X 12-core CPU and 64 GB of memory. Our implementation was fully developed in C++.

4.1. State-of-the-art methods

Subsequently, we compared our algorithm using various metrics against P2M [2] and AABB Tree [1]. AABB Tree is an implementation of Bounding Volume Hierarchy (BVH) featuring axis-aligned bounding boxes that are straightforward to construct and query. This structure is commonly used for point-to-line segment queries. During the query phase, it identifies which nodes' bounds encompass the query point and decides whether to continue searching their child nodes. Upon reaching a leaf node, it examines whether the contained line segments can provide a closer distance. P2M is a novel query acceleration data structure that constructs an interception table that encodes the proxy relationships between points and other primitives in the preprocessing stage. In the query stage, it first identifies the closest point to the query point using a KDT and then consults the interception table for that point to determine if any primitive can offer a closer distance.

4.2. Dataset

To validate the effectiveness of our method, we randomly generated test segments within the two-dimensional space $[0,1] \times [0,1]$. Different datasets were created by controlling the number of line segments and their lengths. In Fig. 9 top row, we depict data generation scenarios for a fixed proportion of line segment lengths relative to 1/10, while varying the numbers of line segments. In the bottom row of Fig. 9, we illustrate the variation in the dataset with 10K line segments, depicting a gradual increase in the average length of the segments from left to right. In the subsequent experiments, unless otherwise

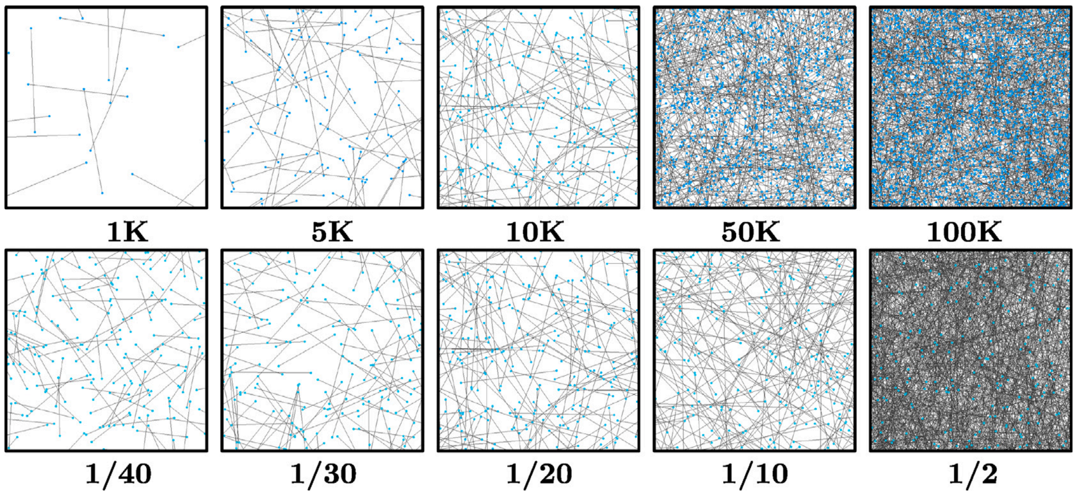


Fig. 9. Partial display of the dataset. The top row shows the situation within an equally sized region is shown for random generation of 1K, 5K, 10K, 50K, and 100K line segments while ensuring that the line segment length is approximately equal to 1/10. The bottom row shows the scenario within an equally sized region is presented for the random generation of line segments with lengths of 1/2, 1/10, 1/20, 1/30, and 1/40 while restricting the line segment number to 10K.

ALGORITHM 2: ArrangeAngleInterval

```
Input: Voronoi edge $e_i = v_s v_t$
Initialize the angle interval list IntervalList$_s$ as empty, to store
  intervals aligning with $v_s$ but not with $v_t$;
Initialize the angle interval list IntervalList$_t$ as empty, to store
  intervals aligning with $v_t$ but not with $v_s$;
Initialize the angle interval list IntervalList$_{st}$ as empty, to store
  intervals aligning with both $v_s$ and $v_t$;
for each $l \in L_{v_s} \backslash L_{v_t}$ do
    Solve Eq. (1) to find the only root $\lambda \in [0,1]$;
    Compute the angle $\alpha$ of $v_0 + \lambda\vec{v}$;
    Push $(\alpha, l)$ into IntervalList$_s$;
end
for each $l \in L_{v_t} \backslash L_{v_s}$ do
    Solve Eq. (1) to find the only root $\lambda \in [0,1]$;
    Compute the angle $\alpha$ of $v_0 + \lambda\vec{v}$;
    Push $(\alpha, l)$ into IntervalList$_t$;
end
for each $l \in L_{v_s} \cap L_{v_t}$ do
    Solve Eq. (1) to find roots $\lambda_i \in [0,1]$;
    if the number of valid roots is less than or equal to 1 then
        push $([], l)$ into IntervalList$_{st}$;
    else
        Compute the angle $\alpha_1$ of $v_0 + \min_{i=1,2}\{\lambda_i\}\vec{v}$;
        Push $(\alpha_1, l)$ into IntervalList$_s$;
        Compute the angle $\alpha_2$ of $v_0 + \max_{i=1,2}\{\lambda_i\}\vec{v}$;
        Push $(\alpha_2, l)$ into IntervalList$_t$;
    end
end
Sort IntervalList$_s$ in descending order;
Sort IntervalList$_t$ in ascending order;
```

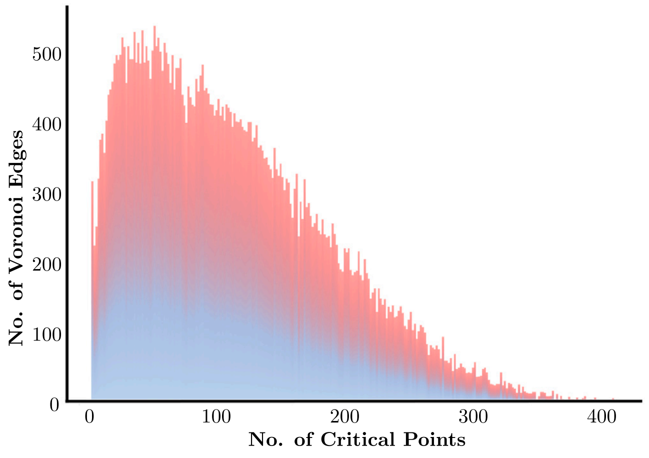


Fig. 10. Different Voronoi edges have varying numbers of proxy line segments associated with them. Here, we illustrate the distribution of Voronoi edges acting as proxies for different quantities of line segments. The horizontal axis represents the number of line segments represented by a single edge, and the vertical axis shows the count of Voronoi edges that serve as proxies for that specific number of line segments.

specified, we randomly generated 20 groups of data, each with different combinations of line segment numbers and lengths. The reported results represent the average timing costs of these 20 groups of data.

4.3. Number of critical points

Reflecting on the preprocessing stage, we calculated critical points to represent intervals for each Voronoi edge. To account for the varying number of critical points for different Voronoi edges, we used input data consisting of 100K randomly generated line segments (with no specific length constraints).

As illustrated in Fig. 10, on the horizontal axis, we represent the number of critical points on each edge, while the vertical axis shows the count of Voronoi edges. It is evident that the distribution of critical points on Voronoi edges is relatively uniform. In this example, the majority of Voronoi edges have critical point counts ranging from 0 to 300, with an average count of approximately 72.19.

4.4. Preprocessing cost

Cost breakdown. The overall preprocessing cost consists of four components: (1) Voronoi diagram generation, (2) computation of which line segments each Voronoi vertex represents, (3) Calculate critical points for each Voronoi edge and organization into the query structure, (4) Calculate the KDT of the line segment endpoint set S. To evaluate the efficiency of each component of our algorithm, we designated the 2D region $[0,1] \times [0,1]$ and produced random line segment data within this

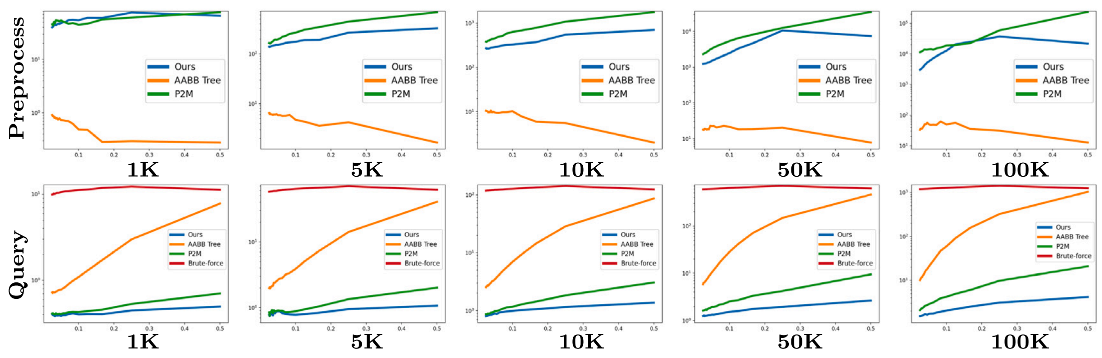


Fig. 11. The horizontal axis of each sub-table represents the line-segments length, with values ranging from $1/40, 1/38, 1/36, \ldots, 1/4, 1/2$. The top row presents a performance comparison of our method and other methods in terms of preprocessing, with the vertical axis measured in milliseconds (ms). The bottom row displays a comparison of the average time taken for a single query between our method and other methods, with units in microseconds (μs). All time data has been log10-transformed for analysis.

Table 1
Our method's breakdown of preprocess stage and query stage. The query stage data is multiplied by 10^3.

No. of segments	Preprocess (ms)				Query (μs)		
	Cal Voronoi	Cal interception	Cal critical	Cal KDT	KDT search	Locating VorEdge	Search candidates
1K	12.34	27.81	9.16	0.34	2.53	0.39	6.93
5K	63.88	154.27	73.58	2.04	8.75	1.11	233.23
10K	140.16	349.96	206.24	4.34	1.28	0.21	332.62
50K	822.37	2407.93	8271.41	26.36	126.53	0.66	1568.58
100K	1815.09	5876.54	28 509.67	53.12	301.55	0.65	2177.42

Table 2
Comparison with existing methods. This table presents the average time required for preprocessing and querying by our method, P2M, AABB Tree, and brute-force (Force) under various numbers and average lengths of line segments. As the number of segments and the average length of segments increase, the preprocessing efficiency of our method significantly surpasses that of P2M, and the advantage in query efficiency becomes increasingly prominent.

Segments length		1/2				1/10				1/20				1/30				1/40			
Method		Our	AABB	P2M	Force	Our	AABB	P2M	Force	Our	AABB	P2M	Force	Our	AABB	P2M	Force	Our	AABB	P2M	Force
Preprocess (ms)	1K	66.2870	**0.27800**	68.9580	\	54.4720	**0.48900**	40.5120	\	48.1650	**0.72900**	48.7590	\	45.9930	**0.82600**	43.2720	\	40.0830	**0.91100**	413 540	\
	5K	326.358	**1.65900**	618.173	\	175.279	**4.73600**	240.701	\	151.980	**5.85700**	175.143	\	14.4147	**6.09300**	155.124	\	139.641	**6.47000**	147.647	\
	10K	699.435	**2.01700**	1600.61	\	328.985	**10.2400**	574.889	\	289.316	**9.62100**	407.773	\	26.8132	**9.92400**	361.330	\	270.534	**10.4810**	340.867	\
	50K	7489.56	**7.80900**	32 265.5	\	2496.79	**21.4670**	4884.65	\	1511.76	**22.5320**	2979.11	\	1291.30	**18.6120**	2311.41	\	1240.40	**17.9130**	2098.68	\
	100K	21 763.8	**12.7510**	214 934	\	12 770.3	**49.6150**	17 183.9	\	5699.66	**50.2430**	12 744.3	\	3695.64	**40.2000**	11 607.9	\	3022.46	**33.2210**	10 391.9	\
Query (μs)	1K	**0.5**	7.8	0.7	11.2	**0.4**	1.1	**0.4**	11.2	**0.4**	0.8	**0.4**	10.6	**0.4**	0.7	**0.4**	10.1	**0.4**	0.7	**0.4**	9.9
	5K	**1.1**	40.5	2.0	61.8	**0.8**	3.9	0.9	63.8	**0.9**	2.6	**0.9**	60.1	**0.8**	2.1	0.9	58.4	**0.8**	2.0	**0.8**	57.6
	10K	**1.4**	86.4	3.1	124.1	**1.0**	7.0	1.2	126.9	**0.9**	3.5	1.0	121.7	**0.8**	2.8	0.9	118.8	**0.8**	2.5	0.9	117.8
	50K	**2.7**	462.5	9.5	622.1	**1.5**	28.4	2.5	637.1	**1.3**	10.5	1.9	610.2	**1.3**	7.0	1.7	602.5	**1.2**	5.9	1.6	598.1
	100K	**4.2**	1032.8	20.9	1243.2	**2.0**	59.5	4.2	1270.4	**1.7**	20.6	2.9	1219.4	**1.6**	12.9	2.4	1199.4	**1.5**	10.1	2.1	1193.5

space (without imposing specific length constraints) to serve as our test dataset. Table 1 illustrates the construction durations for different numbers of line segments in randomly generated data. Specifically, the time costs for the four steps with 10K line segments are as follows 140.16 ms, 349.96 ms, 206.24 ms, 4.34 ms.

Comparison with existing methods. In comparison to the AABB Tree, our algorithm requires more preprocessing time but is more efficient in constructing the Voronoi diagram when dealing with complex data. For instance, the AABB Tree takes approximately 10.240 ms to complete construction, and P2M requires approximately 574.889 ms. In comparison, our method takes 328.985 ms to process data consisting of 10K line segments (with an average length within one-tenth of the data generation range). By controlling the quantity and length of randomly generated test line segments, we provide a comparison of preprocessing times for the AABB Tree, P2M, and our method in Table 2.

And in the top row of Fig. 11, we display a more detailed comparison of the preprocessing efficiency of our method, P2M, and AABB Tree on datasets with different numbers of line segments under the variation of segment lengths.

4.5. Query performance

Cost breakdown. The query phase of our algorithm involves three operations: (1) finding the nearest endpoint s_i through KDT search, (2) determining which Voronoi edge e_i the extended line passes through based on the angle relative to the site point, and (3) identifying the nearest segment by accessing the *query* structure of e_i. As shown in Table 1, the average timing costs for these three parts are 1.28 μs, 0.21 μs, and 332.62 μs, respectively. Notably, we also tested the runtime costs for each part of the query phase under varying numbers of line segments. It can be observed that the majority of the time is spent on examining the information represented by critical points.

Comparison with existing method. To investigate the differences in query performance between our method, P2M, and AABB Tree, we conducted tests comparing their performance under various data scenarios involving different numbers of line segments and segment lengths. In order to demonstrate the efficiency of the algorithms, we also provided the time required for a brute-force query for the same dataset with the same query points. The comparative analysis is presented in Table 2.

First, let us examine the query efficiency as it varies with the number of line segments. It is evident that, regardless of the segment length, our query efficiency consistently outperforms other methods. Furthermore, as the number of line segments increases, our query efficiency advantage becomes increasingly pronounced. For example, when limiting the line segment length to 1/10 and using 1K line segments, our query efficiency is on par with that of P2M and is approximately 2.75 times faster than that of the AABB Tree. When the number of line segments increases to 100K, our query efficiency

Table 3
Comparison of preprocessing time and query time between our method and other methods when the input line segments are long or the line segment lengths are unevenly distributed.

	Preprocess (ms)				Query(μs)			
	Our	AABB	P2M	Force	Our	AABB	P2M	Force
Long segments	56 217.08	27.04	2365.05	\	1.15516	90.5135	2.82452	109.016
Random segments	3497.93	27.49	2397.21	\	1.09594	110.686	3.16098	108.71

surpasses P2M by a factor of 1.1 and is nearly 30 times more efficient than AABB Tree. This clearly indicates that when confronted with datasets containing a substantial number of line segments, our method can provide significantly higher query efficiency compared to existing methods.

Furthermore, we analyze the impact of line segment length on query efficiency under the condition of an equal number of line segments. When dealing with datasets composed of longer line segments, the query efficiency of AABB can be significantly affected due to partitioning issues, with average query times even approaching those of brute-force search. Although P2M has re-encoded the interception table information for each vertex using an R-tree, its capability to handle datasets consisting of longer line segments remains relatively weak. As shown in Table 2, with the increase in average line segment length, our query efficiency becomes even more pronounced in comparison to P2M and AABB Tree. In fact, our query efficiency advantage can reach as high as 4.97 times that of P2M, and it is as much as 245.9 times more efficient than AABB Tree in certain cases.

Additionally, in the bottom row of Fig. 11, we present a nuanced comparison of the query efficiency between our method, P2M, and AABB Tree, evaluated on datasets containing varying numbers of line segments and detailed changes in segment lengths.

4.6. Robustness

In order to verify the robustness of our method, we designed two representative test cases to quantitatively compare its efficiency with that of state-of-the-art techniques. In the first case, line segments with lengths equal to the side length of the square are randomly generated in the square area. Table 3 shows that our preprocessing efficiency has dropped significantly, but in the query stage, our query efficiency is 2.45 times that of P2M. In the second case, the length of the line segment is not restricted, and line segments with lengths between 1/40 and 1 are randomly generated in the square area with a side length of 1. Similarly, we constructed 20 independent datasets, conducted 10,000 queries on each, and reported the mean time as a performance metric. As shown in Table 3, our preprocessing time is 1.45 times that of P2M, but the query efficiency is 2.9 times that of P2M.

5. Conclusions and limitations

In this paper, we introduce a novel algorithmic strategy for handling point-to-segment queries, which outperforms classical methods by several orders of magnitude. Our preprocessing is inspired by the observation that it is sufficient to consider the query point as positioned on the Voronoi edges, where the Voronoi diagram is defined by the endpoints of the line segments. Furthermore, we prove that as the query point moves along a Voronoi edge, for any line segment, there are eight possible situations that determine whether it should be treated as a candidate. This insight significantly simplifies the point-to-segment query to a manageable linear traversal.

However, our algorithm currently has at least two shortcomings. Firstly, the most time-consuming step of the preprocessing involves finding candidate line segments for each Voronoi vertex. Secondly, as shown in Fig. 12, if all input line segments are of equal length and parallel, each site could represent a large number of line segments, resulting in a query efficiency degradation to $O(n)$. This limitation is also present in P2M, whereas the query efficiency of the AABB tree remains unchanged.

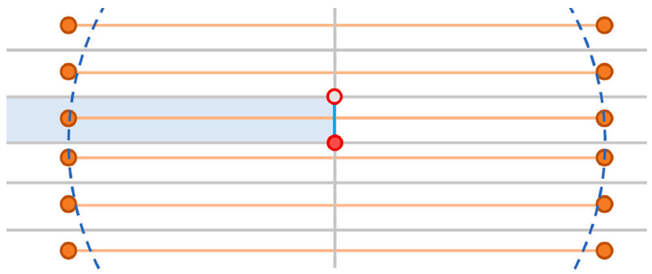


Fig. 12. The orange segments correspond to the input data, and the gray structure represents the Voronoi diagram constructed from their endpoints. For a given light blue Voronoi edge, the circle centered at the red point, with a radius equal to its distance to the nearest site, intersects all input segments.

Although it is possible to extend the same approach to three dimensions—by calculating the proxy relationship of the Voronoi cell plane boundary to the input primitive in the preprocessing stage, identifying the nearest site in the query stage, and querying its proxy triangle primitive based on the angle. But the angle interval in three dimensions involves three dimensions and the situation is more complicated, so extending our algorithm to handle the 3D setting appears to be a non-trivial task.

CRediT authorship contribution statement

Jiantao Song: Writing – original draft, Methodology, Formal analysis, Data curation, Conceptualization. **Rui Xu:** Writing – original draft, Methodology, Formal analysis, Data curation, Conceptualization. **Wensong Wang:** Writing – original draft, Methodology, Formal analysis, Data curation, Conceptualization. **Shiqing Xin:** Writing – original draft, Methodology, Formal analysis, Data curation, Conceptualization. **Shuangmin Chen:** Writing – original draft, Methodology, Formal analysis, Data curation, Conceptualization. **Jiaye Wang:** Methodology, Formal analysis, Data curation, Conceptualization. **Taku Komura:** Methodology. **Wenping Wang:** Methodology. **Changhe Tu:** Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work was supported by the National Key R&D Program of China (2022YFB3303200) and the National Natural Science Foundation of China (U23A20312, 62272277). Additional support was provided by the Research Grants Council of Hong Kong (17210222), the Innovation and Technology Commission of the HKSAR Government under the ITSP-Platform grant (ITS/335/23FP), and the InnoHK initiative (TransGP project).

Data availability

No data was used for the research described in the article.

References

[1] CGAL. The computational geometry algorithms library. 2023, https://www.cgal.org/.
[2] Zong C, Xu J, Song J, Xin S, Chen S, Wang W, Tu C. P2m: A fast solver for querying distance from point to mesh surface. ACM Trans Graph 2023;1–11. http://dx.doi.org/10.1145/3592439.
[3] Roussopoulos N, Kelley S, Vincent F. Nearest neighbor queries. In: Proceedings of the 1995 ACM SIGMOD international conference on management of data. 1995, p. 71–9.
[4] Lu Y, Cheng L, Isenberg T, Fu C-W, Chen G, Liu H, Deussen O, Wang Y. Curve complexity heuristic kd-trees for neighborhood-based exploration of 3d curves. In: Computer graphics forum. vol. 40, Wiley Online Library; 2021, p. 461–74.
[5] Zhao Y, Wang Y, Zhang J, Fu C-W, Xu M, Moritz D. Kd-box: Line-segment-based kd-tree for interactive exploration of large-scale time-series data. IEEE Trans Vis Comput Graphics 2021;28(1):890–900.
[6] Haverkort HJ. Introduction to bounding volume hierarchies [Part of the Ph.D. thesis], Utrecht University; 2004.
[7] Hoel EG, Samet H. A qualitative comparison study of data structures for large line segment databases. In: Proceedings of the 1992 ACM SIGMOD international conference on management of data. 1992, p. 205–14.
[8] Elseberg J, Magnenat S, Siegwart R, Nüchter A. Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. J Softw Eng Robot 2012;3(1):2–12.
[9] Jones MW, Baerentzen JA, Sramek M. 3D distance fields: A survey of techniques and applications. IEEE Trans Vis Comput Graphics 2006;12(4):581–99.
[10] Zhang J, Guo H, Hong F, Yuan X, Peterka T. Dynamic load balancing based on constrained kd tree decomposition for parallel particle tracing. IEEE Trans Vis Comput Graphics 2017;24(1):954–63.
[11] Wang F, Wald I, Wu Q, Usher W, Johnson CR. Cpu isosurface ray tracing of adaptive mesh refinement data. IEEE Trans Vis Comput Graphics 2018;25(1):1142–51.
[12] Parker S, Shirley P, Livnat Y, Hansen C, Sloan P-P. Interactive ray tracing for isosurface rendering. In: Proceedings visualization'98 (cat. no. 98CB36276). IEEE; 1998, p. 233–8.
[13] Zellmann S, Schulze JP, Lang U, Childs H, Cuc-Chietti F. Rapid kd tree construction for sparse volume data. In: EGPGV@ EuroVis. 2018, p. 69–77.
[14] Guttman A. R-trees: A dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD international conference on management of data. 1984, p. 47–57.
[15] Beckmann N, Kriegel H-P, Schneider R, Seeger B. The r*-tree: An efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD international conference on management of data. 1990, p. 322–31.
[16] Kamel I, Faloutsos C. Hilbert r-tree: An improved r-tree using fractals. Tech. rep., 1993.
[17] Berchtold S, Keim DA, Kriegel H-P. The x-tree: An index structure for high-dimensional data. In: Very large data-bases. 1996, p. 28–39.
[18] Liu T, Moore AW, Gray A, Cardie C. New algorithms for efficient high-dimensional nonparametric classification. J Mach Learn Res 2006;7(6).
[19] Sakurai Y, Yoshikawa M, Uemura S, Kojima H, et al. The a-tree: An index structure for high-dimensional spaces using relative approximation. In: VLDB. vol. 2000, Citeseer; 2000, p. 5–16.
[20] White DA, Jain R. Similarity indexing with the ss-tree. In: Proceedings of the twelfth international conference on data engineering. IEEE; 1996, p. 516–23.
[21] Katayama N, Satoh S. The sr-tree: An index structure for high-dimensional nearest neighbor queries. ACM Sigmod Rec 1997;26(2):369–80.
[22] Ytterlid R, Shellshear E. Bvh split strategies for fast distance queries. J Comput Graph Tech (JCGT) 2015;4(1):1–25.
[23] Evangelou I, Papaioannou G, Vardis K, Vasilakis AA. Fast radius search exploiting ray-tracing frameworks. J Comput Graph Tech 2021;10(1):25–48.
[24] Fan P, Wang W, Tong R, Li H, Tang M. Gdist: Efficient distance computation between 3d meshes on gpu. In: SIGGRAPH Asia 2024 conference papers. 2024, p. 1–11.
[25] Wang M, Cao J. A review of collision detection for deformable objects. Comput Animat Virtual Worlds 2021;32(5):e1987.
[26] Lee J, Lee W-J, Shin Y, Hwang S, Ryu S, Kim J. Two-aabb traversal for mobile real-time ray tracing. In: SIGGRAPH Asia 2014 mobile graphics and interactive applications. 2014, p. 1–5.
[27] Larsson T, Akenine-Möller T. A dynamic bounding volume hierarchy for generalized collision detection. Comput Graph 2006;30(3):450–9.
[28] Hubbard PM. Collision detection for interactive graphics applications. IEEE Trans Vis Comput Graphics 1995;1(3):218–30.
[29] Kavan L, Žára J. Fast collision detection for skeletally deformable models. In: Computer graphics forum. vol. 24, Oxford, UK and Boston, USA: Blackwell Publishing, Inc; 2005, p. 363–72.
[30] Palmer IJ, Grimsdale RL. Collision detection for animation using sphere-trees. In: Computer graphics forum. vol. 14, Wiley Online Library; 1995, p. 105–16.
[31] Gottschalk S, Lin MC, Manocha D. Obbtree: A hierarchical structure for rapid interference detection. In: Proceedings of the 23rd annual conference on computer graphics and interactive techniques. 1996, p. 171–80.
[32] Klosowski JT, Held M, Mitchell JS, Sowizral H, Zikan K. Efficient collision detection using bounding volume hierarchies of k-dops. IEEE Trans Vis Comput Graphics 1998;4(1):21–36.
[33] Guibas LJ, Nguyen AT, Zhang L. Zonotopes as bounding volumes. In: SODA. vol. 3, 2003, p. 803–12.
[34] Barequet G, Chazelle B, Guibas LJ, Mitchell JS, Tal A. Boxtree: A hierarchical representation for surfaces in 3d. In: Computer graphics forum. vol. 15, Wiley Online Library; 1996, p. 387–96.
[35] Coming DS, Staadt OG. Velocity-aligned discrete oriented polytopes for dynamic collision detection. IEEE Trans Vis Comput Graphics 2007;14(1):1–12.
[36] Ehmann SA, Lin MC. Accurate and fast proximity queries between polyhedra using convex surface decomposition. In: Computer graphics forum. vol. 20, Wiley Online Library; 2001, p. 500–11.
[37] Liu S, Wang CC, Hui K-C, Jin X, Zhao H. Ellipsoid-tree construction for solid objects. In: Proceedings of the 2007 ACM symposium on solid and physical modeling. 2007, p. 303–8.