

Segment Proximity Graphs and Nearest Neighbor Queries Amid Disjoint Segments

Pankaj K. Agarwal 

Department of Computer Science, Duke University, Durham, NC, USA

Haim Kaplan 

School of Computer Science, Tel Aviv University, Israel

Matthew J. Katz 

Department of Computer Science, Ben-Gurion University of the Negev, Beer Sheva, Israel

Micha Sharir 

School of Computer Science, Tel Aviv University, Israel

Abstract

In this paper we study a few proximity problems related to a set of pairwise-disjoint segments in $\mathbb{R}^2$. Let S be a set of n pairwise-disjoint segments in $\mathbb{R}^2$, and let $r > 0$ be a parameter. We define the *segment proximity graph* of S to be $G_r(S) := (S, E)$, where $E = \{(e_1, e_2) \mid \text{dist}(e_1, e_2) \leq r\}$ and $\text{dist}(e_1, e_2) = \min_{(p,q) \in e_1 \times e_2} \|p - q\|$ is the Euclidean distance between e_1 and e_2 . We define the weight of an edge $(e_1, e_2) \in E$ to be $\text{dist}(e_1, e_2)$.

We first present a simple grid-based $O(n \log^2 n)$ -time algorithm for computing a BFS tree of $G_r(S)$. We apply it to obtain an $O^*(n^{6/5}) + O(n \log^2 n \log \Delta)$ -time algorithm for the so-called *reverse shortest path problem*, in which we want to find the smallest value r^* for which $G_{r^*}(S)$ contains a path of some specified length between two designated start and target segments (where the $O^*(\cdot)$ notation hides polylogarithmic factors). Here $\Delta = \max_{e \neq e' \in S} \text{dist}(e, e') / \min_{e \neq e' \in S} \text{dist}(e, e')$ is the *spread* of S .

Next, we present a dynamic data structure that can maintain a set S of pairwise-disjoint segments in the plane under insertions/deletions, so that, for a query segment e from an unknown set Q of pairwise-disjoint segments, such that e does not intersect any segment in (the current version of) S , the segment of S closest to e can be computed in $O(\log^5 n)$ amortized time. The amortized update time is also $O(\log^5 n)$. We note that if the segments in $S \cup Q$ are allowed to intersect then the known lower bounds on halfplane range searching suggest that a sequence of n updates and queries may take at least close to $\Omega(n^{4/3})$ time. One thus has to strongly rely on the non-intersecting property of S and Q to perform updates and queries in $O(\text{polylog}(n))$ (amortized) time each.

Using these results on nearest-neighbor (NN) searching for disjoint segments, we show that a DFS tree (or forest) of $G_r(S)$ can be computed in $O^*(n)$ time. We also obtain an $O^*(n)$ -time algorithm for constructing a minimum spanning tree of $G_r(S)$.

Finally, we present an $O^*(n^{4/3})$ -time algorithm for computing a single-source shortest-path tree in $G_r(S)$. This is the only result that does not exploit the disjointness of the input segments.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Computational geometry

Keywords and phrases segment proximity graphs, nearest neighbor searching, dynamic data structures, BFS, DFS, unit-disk graphs

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.7

Funding Pankaj K. Agarwal: Partially supported by NSF grants IIS-18-14493, CCF-20-07556, and CCF-22-23870.

Haim Kaplan: Partially supported by Israel Science Foundation Grant 1156/23 and the Blavatnik research Foundation.

Matthew J. Katz: Partially supported by Israel Science Foundation Grant 495/23.

Micha Sharir: Partially supported by Israel Science Foundation Grant 495/23.



© Pankaj K. Agarwal, Haim Kaplan, Matthew J. Katz, and Micha Sharir;
licensed under Creative Commons License CC-BY 4.0

32nd Annual European Symposium on Algorithms (ESA 2024).

Editors: Timothy Chan, Johannes Fischer, John Iacono, and Grzegorz Herman; Article No. 7; pp. 7:1–7:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Let S be a set of n geometric (compact) objects in $\mathbb{R}^2$ (e.g., points, disks, segments, rectangles, or convex polygons), and let $r > 0$ be a real parameter. Define the r -proximity graph of S , or *proximity graph* for short, denoted as $G_r(S)$, to be the graph whose vertices are the objects of S , and whose edges are all pairs (e, e') of objects in S with $\text{dist}(e, e') \leq r$, where $\text{dist}(e, e') = \min_{p \in e, q \in e'} \|p - q\|$. For $r = +\infty$, we obtain the complete graph on S . Even for finite values of r , $G_r(S)$ can have a quadratic number of edges. Note that if S is a set of points in $\mathbb{R}^2$, then $G_r(S)$ is the widely studied *unit-disk* graph (where the unit (common radius) is $r/2$); see [23]. On the other hand, for $r = 0$, $G_r(S)$ is the *geometric intersection graph* on S . Because of their numerous applications, geometric proximity (and intersection) graphs have been studied extensively in many different fields.

Since one does not have to specify the edge set E of $G_r(S)$ explicitly, a natural and fundamental question in the theory of proximity graphs is whether basic graph algorithms, such as BFS, DFS, minimum-spanning-tree algorithms, or Dijkstra's shortest-paths algorithm, can be implemented in time that is near linear in $|S|$ (instead of being linear or near linear in $|E|$), or at least subquadratic in $|S|$. Efficient algorithms for these problems require a dynamic data structure on a subset X of S that can handle insertions/deletions of objects of S into/from X , and that can quickly compute a fixed-radius neighbor or a nearest neighbor in X of a query object, a fundamental problem of independent interest. In this paper we study these questions when S is a set of pairwise-disjoint segments, and so are the query segments. By strongly exploiting the disjointness property of input and query segments, we present a data structure that answers nearest-neighbor queries in $O^*(1)$ time, and obtain $O^*(n)$ -time algorithms for some basic graph problems on $G_r(S)$.¹

Related work. A wide range of combinatorial and algorithmic questions have been studied for geometric proximity graphs, e.g., realizability of graphs as geometric proximity graphs (such as unit-disk graphs or segment-intersection graphs), the computational complexity of deciding whether a given graph can be represented as a geometric intersection graph, or studying extremal properties for geometric proximity graphs; see [6, 24, 29, 30, 36, 42, 48] and references therein for a few such results. Motivated by applications in communication networks and data mining, there has been much work on developing faster algorithms for basic graph problems on unit-disk graphs; see [5, 14, 15, 19, 21, 23, 27, 28, 31, 47] for a sample of known results on these graphs.

Despite extensive work on unit-disk graphs, much less is known for algorithmic problems for proximity graphs of more complex objects. An $O^*(n)$ algorithm is known for performing a BFS on a disk-proximity graph (where the disks have arbitrary radii) [20], and the algorithm in [15] for computing a single-source shortest-path tree on unit-disk graphs can be extended to disk-proximity graphs, with $O^*(n)$ running time; see also [18, 35].

An interesting and natural question is whether basic graph algorithms (e.g., BFS or DFS) can be performed in $O^*(n)$ time on segment proximity graphs. Recently, Agarwal *et al.* [3] showed that BFS and DFS on segment proximity graphs can be performed in $O^*(n^{4/3})$ time, in a general setting where the segments can intersect. Furthermore, the known lower-bound results on (planar) simplex range searching and on the so-called Hopcroft's problem suggest that an $O(n^{4/3-\varepsilon})$ -time algorithm, for any $\varepsilon > 0$, is unlikely to exist, even for performing BFS, if the segments in S can intersect each other [1, 26]. However, these lower bounds fail if

¹ As in the abstract, the $O^*(\cdot)$ notation hides $\text{polylog}(n)$ factors.

the segments in S are pairwise disjoint, which raises a natural question whether $O^*(n)$ -time algorithms exist for basic graph problems (such as BFS/DFS/MST) on a segment proximity graph, assuming that the segments are pairwise disjoint.

Bespamyatnikh and Snoeyink [12], and later Bespamyatnikh [11], studied the nearest-neighbor searching problem amid a set of pairwise-disjoint segments in an off-line setting: Let R be a set of red segments and let B be another set of blue segments in $\mathbb{R}^2$, so that the segments in $R \cup B$ are pairwise disjoint; set $n = |R| + |B|$. The work in [11] yields an $O(n \log^2 n)$ -time algorithm to compute the nearest neighbor of every segment of B in R . However, this algorithm is inherently static and off-line – it is not clear how to extend it to the case where the set B is not known in advance, or when we allow to update R dynamically. For the case where the segments are allowed to intersect, Bespamyatnikh [11] solves the problem in $O(n^{4/3} 2^{O(\log^* n)})$ time, which was recently improved to $O(n^{4/3})$ time [50].

Our contributions. Let S be a set of n pairwise-disjoint segments in $\mathbb{R}^2$, and let $r > 0$ be a parameter. We obtain the following results:

Breadth-first search. Our first main result (Section 2) is a simple $O(n \log^2 n)$ -time algorithm for computing a BFS tree of $G_r(S)$ (assuming it is connected), using a grid-based technique. Although it resembles the technique of Chan and Skrepetos [19] for unit-disk graphs, dealing with segments, which may straddle any number of grid cells, requires new ideas, and makes our algorithm different from the one in [19] in many substantial ways.

By combining our BFS algorithm with a novel implementation of parametric search, originally developed in [8], and akin to the recent machinery developed by the authors for disk graphs in [32], we get, also in Section 2, an algorithm for the *reverse shortest path problem*: for a pair of segments $s, t \in S$ and an integer parameter $k > 0$, compute the smallest value r^* of r such that $G_{r^*}(S)$ contains a path from s to t composed of at most k edges. The running time of the algorithm is $O^*(n^{6/5}) + O(n \log^2 n \log \Delta)$, where Δ is the *spread* of S , defined as $\Delta = \max_{e \neq e' \in S} \text{dist}(e, e') / \min_{e \neq e' \in S} \text{dist}(e, e')$.

Nearest-neighbor (NN) queries and bichromatic closest pair. Our second main result (in Section 3) is a dynamic data structure that can maintain a set S of pairwise disjoint segments in the plane under insertions and deletions (so that the segments in S remain pairwise disjoint after each update), so as to answer efficiently NN-queries with segments as queries. We do not need to know the set of query segments in advance. The only assumptions we need are that (i) all the query segments remain pairwise disjoint throughout the process, and (ii) each query segment e is disjoint from the segments of S at the time of the query. A query with a segment e asks for the segment in (the current) S closest to e . The amortized query and update times are $O(\log^5 n)$. If the segments in Q are known in advance (but not necessarily the sequence of queries), then the amortized query and update time can be improved to $O(\log^4 n)$.

We use this data structure for performing DFS on $G_r(S)$, but this result is of independent interest and has potential applications in many other proximity problems involving segments. For example, it is required for efficient implementation of the online facility-location algorithm of Meyerson [41], and for other clustering algorithms, where the items and the centers are pairwise disjoint segments.

We emphasize again that if the input or query segments are allowed to intersect then the known lower bounds on planar simplex range searching suggest that a sequence of n updates and queries might take at least $\Omega(n^{4/3})$ time [1] (or at least time close to it; see

7:4 Segment Proximity Graphs and NN Searching

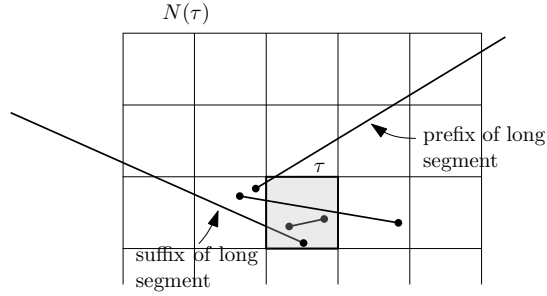
the aforementioned works [11, 50] for matching upper bounds). One thus has to strongly rely on the assumed property that the segments are pairwise disjoint, in order to be able to perform updates and queries in $O(\text{polylog}(n))$ (amortized) time per operation. In other words, this assumption is not just an artifact made to simplify the analysis, but an essential requirement, without which the whole machinery, and the hope for fast algorithms of this kind, collapse. The assumption that the query segments be pairwise disjoint might seem puzzling and counter-intuitive, but, informally speaking, we gain some information from the previous searches, which we use to speed up the current query, exploiting its disjointness from the preceding queries.

Depth-first search and minimum spanning tree. Using the results just mentioned, we obtain (in the full version of the paper) an efficient algorithm, that runs in $O^*(n)$ time, for constructing a Depth-First Search (DFS) tree for $G_r(S)$ (or a forest if $G_r(S)$ is not connected).

Regarding $G_r(S)$ as a weighted graph and using Borůvka’s algorithm, we can compute an MST of $G_r(S)$ in $O(n \log^3 n)$ time.² The algorithm is based on an $O(n \log^2 n)$ -time algorithm for the all “foreign” nearest neighbors (AFNN) problem in a set S of n pairwise-disjoint segments, namely, each segment in S is assigned a color and the goal is to compute the closest segment of a different color for every segment in S (see [11]).

Single-source shortest paths. Finally, we design an $O^*(n^{4/3})$ -time algorithm for computing a single-source shortest-path tree in (the weighted) $G_r(S)$. The algorithm is presented in the full version of the paper.

2 BFS in Segment Proximity Graphs



■ **Figure 1** The (upper part of the) neighborhood $N(\tau)$. S_τ consists of two short segments, a prefix, and a suffix.

Let A be a set of n pairwise-disjoint segments in $\mathbb{R}^2$, $r > 0$ a real parameter, and $s \in A$ a designated start segment.³ We present an $O(n \log^2 n)$ -time algorithm for running BFS from s in $G_r(A)$. We first describe the data structures that we use and then describe the procedure itself.

² We thank an anonymous reviewer of an earlier version of the paper for suggesting to use Borůvka’s algorithm instead of Prim’s algorithm.

³ We use A for the set of segments because S has a different meaning here.

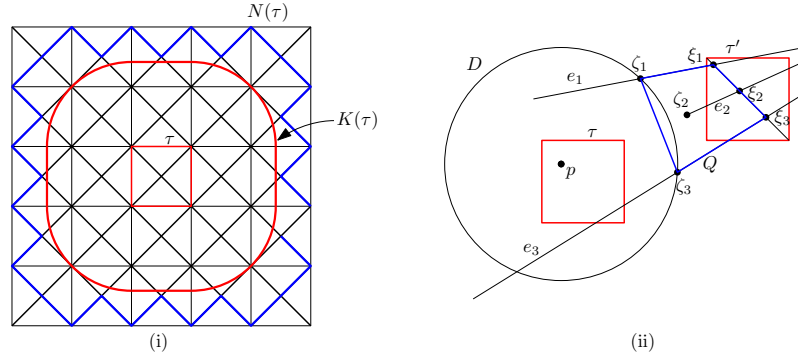
τ (which holds iff they intersect τ); each segment appears either once or twice in this union. We repeat this procedure for all cells in the neighborhood of a non-empty grid cell. Let $\mathcal{C}$ be the set of canonical subsets that are associated with at least one grid cell. Any canonical set in $\mathcal{C}$ has the property that every pair of its segments are within distance r from each other (as they cross the same diagonal of a grid cell).

We keep cross-pointers from each canonical set $C \in \mathcal{C}$ to all its associated cells, i.e., we store $\mathcal{G}(C) = \{\tau \mid C \in \mathcal{C}(\tau)\}$. We also keep cross pointers from each segment $e \in A$ to the set $\mathcal{C}(e)$ of all canonical subsets in $\mathcal{C}$ that contain e . All this data takes $O(n \log^2 n)$ time to compute. We have the following useful lemma

► **Lemma 1.** *Let p be a point in a grid cell τ .*

- (i) *If the middle portion e° of a long segment e is within distance r from p , then e° intersects one of the diagonals of a neighborhood cell $\tau' \in N(\tau)$ of τ .*
- (ii) *Let A_Δ be the sequence of long segments whose middle portions intersect a diagonal Δ of a cell $\tau' \in N(\tau)$, ordered by their intersection points along Δ . The subset $A_{\Delta,p} \subseteq A_\Delta$ of segments that are within distance r from p forms a contiguous subsequence of A_Δ , and it can be computed in $O(\log^2 n + |A_{\Delta,p}|)$ time.*

Proof. (i) Let q be the point on e° closest to p . In particular, the distance of q from τ is at most r . The two families of diagonals of the grid cells of $N(\tau)$ form a portion of a uniform grid (rotated by $\pi/4$) of cell side-length $r/2$. The cells of this rotated grid cover $N(\tau)$, except for *fringe triangles*, each of which is a right-angle triangle whose hypotenuse is a boundary edge of $N(\tau)$. See Figure 3(i). If q lies inside a complete cell σ of the rotated grid then, since e° is of length at least r , it must intersect $\partial\sigma$, i.e., a diagonal of a cell of $N(\tau)$, as claimed. If q lies in a fringe triangle, its distance from τ , and thus from p , is at least as large as the distance from τ of the apex of that triangle (again, refer to Figure 3(i)). But this latter distance is at least $\frac{r}{\sqrt{2}} + \frac{1}{2} \cdot \frac{r}{\sqrt{2}} = \frac{3r}{2\sqrt{2}} > r$, contradicting our assumption. This proves (i).



■ **Figure 3** (i): Illustration of Lemma 1(i). $K(\tau) = \tau + B(o, r)$; the boundary of grid cells formed by diagonals is shown in blue. Fringe triangles, lying outside the blue region, do not intersect $K(\tau)$. (ii): Illustration of Lemma 1 (ii). Three segments of a canonical set intersecting a diagonal of τ' . If e_1 and e_3 intersect $D(p)$ but e_2 does not, then an endpoint of e_2 is too close to ξ_2 .

(ii) Let $D(p)$ be the disk of radius r centered at p , and let $e_1 \prec e_2 \prec e_3$ (where $\prec$ is the order according to the intersection points with Δ) be three segments of A_Δ such that e_1, e_3 intersect $D(p)$ but e_2 does not. See Figure 3(ii). Let $\xi_i = e_i \cap \Delta$, for $i = 1, 2, 3$. Let ζ_1 be the intersection point of $D(p)$ with e_1 closest to ξ_1 ; if $\xi_1 \in D(p)$ then $\zeta_1 = \xi_1$. Similarly define ζ_3 for e_3 . Consider the quadrilateral $Q := \zeta_1 \xi_1 \zeta_3 \xi_3$. The edge $\xi_1 \xi_3$ contains the point ξ_2 , and the edge $\zeta_1 \zeta_3$ lies inside $D(p)$. If e_2 does not intersect $D(p)$, then one of the endpoints of e_2 ,

denoted by ζ_2 , has to lie inside Q (since the segments in A_Δ are pairwise disjoint), which implies that $\|\xi_2 - \zeta_2\| \leq \max\{\|\zeta_1 - \xi_3\|, \|\zeta_3 - \xi_1\|\}$.

Since $\tau' \in N(\tau)$ and $p \in \tau$, $\|p - \xi_i\| \leq 3r$, for $i \leq 3$. Furthermore, for $i = 1, 3$, $\|\xi_i - \zeta_i\| \leq \|p - \xi_i\| \leq 3r$, which implies

$$\max\{\|\xi_1 - \zeta_3\|, \|\xi_3 - \zeta_1\|\} \leq \max\{\|\xi_3 - \zeta_3\|, \|\xi_1 - \zeta_1\|\} + \|\xi_1 - \xi_3\| \leq 3r + r \leq 4r.$$

Hence, $\|\xi_2 - \zeta_2\| \leq 4r$. However, ξ_2 lies on the middle portion of e_2 , implying that $\|\xi_2 - \zeta_2\| \geq 6r$, a contradiction. Therefore e_2 also intersects D . This shows that $A_{\Delta,p}$ is a contiguous subsequence of A_Δ .

Finally, we compute $A_{\Delta,p}$ as follows. Recall that we have A_Δ at our disposal as the union of $O(\log^2 n)$ canonical subsets, collected at $O(\log n)$ nodes of one of the segment trees, say, T^+ . Consider the $O(\log n)$ canonical subsets $C_1, \dots, C_r$, $r = O(\log n)$, obtained from one node v of T^+ . These $O(\log n)$ canonical subsets are linearly ordered along Δ , i.e., all of the segments in one canonical subset appear before those of the other. Let $C_1 \prec C_2 \prec \dots \prec C_r$ be this linear ordering. Since $A_{\Delta,p}$ is a contiguous sequence, by testing the first and the last segments of each C_i , we can determine the first and the last canonical subsets, say C_i and C_j , with $i < j$, whose segments intersect $D(p)$. Then all segments $C_{i+1}, \dots, C_{j-1}$ intersect $D(p)$. Finally, by performing a binary search in each of C_i and C_j , we can determine the first and the last segments of these canonical subsets, respectively, that intersect $D(p)$. By repeating this step for all $O(\log n)$ nodes, we can report $A_{\Delta,p}$ in $O(\log^2 n + |A_{\Delta,p}|)$ time. ◀

2.2 The BFS itself

We proceed layer by layer. For each $i \geq 0$, we compute L_i , the set of segments reached at layer i , starting with $L_0 = \{s\}$. Assume that we have already constructed the i -th layer L_i . Let $U_i = A \setminus \bigcup_{j \leq i} L_j$ denote the subset of *unreached* segments after i layers. A segment $e \in U_i$ is in L_{i+1} if there is a segment $e' \in L_i$ within distance r from e . Since the segments in A are pairwise disjoint, either an endpoint of e is within distance r from e' or an endpoint of e' is within distance r from e . This in turn implies that at least one of e and e' belongs to S_τ for some non-empty cell τ , and the other intersects a cell in $N(\tau)$. Using this simple observation, we compute L_{i+1} as we describe after introducing the following definitions of an active cell and an active canonical set.

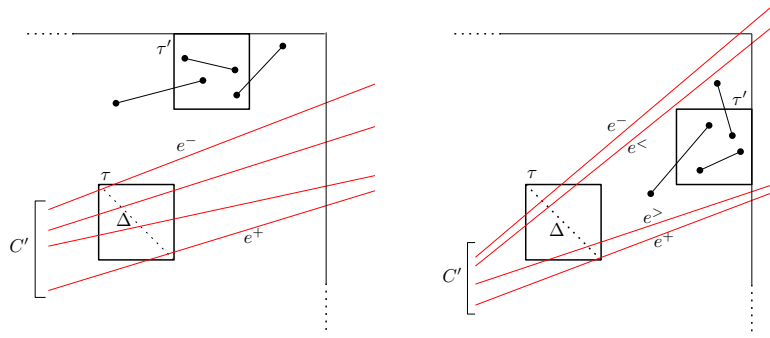
We say that a non-empty cell τ is *active* if $S_\tau \cap L_i \neq \emptyset$. Similarly, a canonical set C is *active* if $C \cap L_i \neq \emptyset$. It is easily seen that a cell or a canonical set is active during the processing of at most two consecutive layers. Once a segment e is added to L_{i+1} , it is deleted from all canonical subsets of $\mathcal{C}(e)$ to which it belongs, to avoid e being reported as a newly reached segment several times. As such, each canonical subset only stores those segments which have not been reached so far.

First, for each active cell τ , we add to L_{i+1} all segments in $S_\tau \cap U_i$ as well as all (long) segments in the canonical subsets of $\mathcal{C}(\tau)$. Similarly, for each active canonical subset C and for every cell $\tau \in \mathcal{G}(C)$, we add all segments of $\bigcup \mathcal{C}(\tau)$ and $S_\tau \cap U_i$ to L_{i+1} . We compute the remaining segments of L_{i+1} in the following three steps.

Step I: Processing $S_{\tau'}$ in the neighborhood of an active cell. Let τ be an active cell. We compute $L_i(\tau) = S_\tau \cap L_i$ and $U_i(\tau) = \bigcup_{\tau' \in N(\tau) \setminus \{\tau\}} S_{\tau'} \cap U_i$, in brute force, in $O\left(\sum_{\tau' \in N(\tau)} |S_{\tau'}|\right)$ time. Next, we compute the segments of $U_i(\tau)$ that are within distance r from some segment of $L_i(\tau)$ (and then add them to L_{i+1}) in two steps. **(1)** We construct the Voronoi diagram $V_i(\tau)$ of $L_i(\tau)$. For each segment $e \in U_i(\tau)$, we locate its endpoints in

$V_i(\tau)$. If the distance of either endpoint from its closest segment in $V_i(\tau)$ is $\leq r$, we add e to L_{i+1} . (2) Let $\mathcal{D}_i(\tau)$ be the set of disks of radius r centered at the endpoints of segments in $L_i(\tau)$. We compute $\mathcal{U}(\mathcal{D}_i(\tau))$, the union of the disks in $\mathcal{D}_i(\tau)$. Using a line-sweep algorithm, we compute all segments in $U_i(\tau)$ that intersect $\mathcal{U}(\mathcal{D}_i(\tau))$ and add them to L_{i+1} . This step takes $O((|L_i(\tau)| + |U_i(\tau)|) \log n) = O\left(\left(\sum_{\tau' \in N(\tau)} |S_{\tau'}|\right) \log n\right)$ time.⁴

Step II: Processing long segments near active cells. Let τ be an active cell. Let $P_i(\tau)$ be the set of endpoints of the segments of $L_i(\tau)$ that lie in τ . We compute the long segments whose middle portions are within distance r from some point p of $P_i(\tau)$. By Lemma 1, the middle portion of such a long segment intersects a diagonal of a cell $\tau' \in N(\tau)$. For each diagonal Δ of every cell $\tau' \in N(\tau)$, using Lemma 1, we report the set $A_{\Delta,p} \subseteq U_i$ of long segments that are within distance r from p and whose middle portions intersect Δ , in time $O(\log^2 n + |A_{\Delta,p}|)$. We add the segments of $A_{\Delta,p}$ to L_{i+1} and, as mentioned above, delete them from all canonical subsets, so that they are not reported again. The total time spent in this step for an active cell τ is $O(|L_i(\tau)| \log^2 n + \lambda_\tau)$, where λ_τ is the number of long segments reported while processing τ . Recall that each segment of L_{i+1} is reported at most once in this step (over all active cells), so $\sum_\tau \lambda_\tau \leq |L_{i+1}|$.



■ **Figure 4** (Left): All segments of a canonical subset C' lie on the same side of a grid cell τ' . (Right): The first segment of a canonical subset C' lies above τ' and the last segment of C' lies below τ' .

Step III: Processing active canonical sets. Finally, we collect the still unreported segments, that should be added to L_{i+1} , whose endpoints are within distance r from the middle portion of a long segment e of L_i ; e belongs to one of the currently active canonical subsets. To do so, we take each active canonical set C and construct the subsequence $C_i = C \cap L_i$; assume C_i is ordered from top to bottom (in the $y = -x$ direction, say). Let e^- (resp., e^+) be the first (resp., last) segment of C_i . We go over all cells $\tau' \in N(\tau)$, for every $\tau \in \mathcal{G}(C)$. Let $U_i^\circ(\tau') \subseteq S_{\tau'}$ be the set of yet unreached segments of $S_{\tau'}$, such that (at least) one of their endpoints lies in τ' . See Figure 4. If both e^- and e^+ lie below (resp., above) τ' then we test every segment of $U_i^\circ(\tau')$ whether it lies within distance r from e^- (resp., e^+), and if so, we add it to L_{i+1} . If either e^- or e^+ intersects τ' , we add all of $U_i^\circ(\tau')$ to L_{i+1} . Otherwise, e^- lies above τ' and e^+ below τ' . We first check, in $O(\log n)$ time, using binary search, whether

⁴ (a) Recall that the Voronoi diagram of m pairwise-disjoint segments has linear complexity, and it can be constructed in $O(m \log m)$ time [7]. (b) As soon as the line sweep detects a segment e that intersects $\mathcal{U}(\mathcal{D}_i(\tau))$, we delete e , to ensure that the number of events processed is $O(|L_i(\tau)| + |U_{i+1}(\tau)|)$.

a segment of C_i intersects τ' . If so, we add all of $U_i^\circ(\tau')$ to L_{i+1} , as above. If not, the binary search returns the last segment $e^<$ of C_i lying above τ' and the first segment $e^>$ of C_i lying below τ' . Again, see Figure 4. We check every segment $e \in U_\tau^\circ$ whether it is within distance r from $e^<$ or $e^>$. If yes, we add e to L_{i+1} .

This completes the description of the $(i+1)$ -st stage of the BFS procedure, and thus of the whole procedure.

Correctness. We prove the correctness by induction on the BFS layers. What one needs to argue is that all the unreached neighbors of segments in L_i are correctly detected and added to L_{i+1} . Using the observation that two segments intersecting the same grid cell are within distance r , it is easily seen that any segment added to L_{i+1} is within distance r from a segment of L_i . Therefore it suffices to argue that any unreached segment $e' \in U_i$ that is within distance r from a segment $e \in L_i$ is added to L_{i+1} . Let $(p, p') \in e \times e'$ be the closest pair of e and e' . Suppose $p \in \tau$ and $p' \in \tau'$, then $\tau' \in N(\tau)$ and vice-versa. Since e and e' do not intersect, at least one of p and p' is an endpoint of that segment.

First consider the case when p is an endpoint of e . If $e' \in S(\tau'')$ for some $\tau'' \in N(\tau)$, then e' will be added to L_{i+1} either in the initial step or Step I. Otherwise p' lies in the middle portion of e' . By Lemma 1, the middle portion of e' intersects the diagonal of a cell $\tau'' \in N(\tau)$ and thus e' belongs to one of the canonical subsets of $\mathcal{C}(\tau'')$. In this case, Step II will add e' to L_{i+1} .

Next, consider the case when p' is an endpoint of e' . If e is short or if p lies on the prefix or suffix of a long segment, then τ is an active cell and Step I will add e' to L_{i+1} . So assume that p lies on the middle portion of e . In this case, all canonical subsets containing e are active. By Lemma 1, there is a cell $\tau'' \in N(\tau)$ such that e lies in a canonical subset $C \in \mathcal{C}(\tau'')$, which implies that $\tau'' \in \mathcal{G}(C)$. Therefore Step III will test τ' while processing C and detect that e' is within distance r from a segment of $L_i \cap C$ and add it to L_{i+1} .

Hence, we conclude that all segments of U_i that are within distance r from a segment of L_i are added to L_{i+1} .

Running time. As for the running time, we spend $O(n \log^2 n)$ time to construct the data structure, canonical subsets, and all the cross pointers.

To bound the time taken by the BFS itself, consider a fixed cell τ of the grid. The segments of S_τ are processed when (i) a cell in $N(\tau)$ (including τ itself) is active, or (ii) a canonical set in $\mathcal{C}(\tau)$ is active.

Case (i) happens for each cell in $N(\tau)$ only during the processing of two consecutive BFS layers, and thus $O(1)$ times overall, since $|N(\tau)| = O(1)$. The work for processing the segments of S_τ in this case is $O(|S_\tau| \log n)$. Case (ii) may occur $O(\log^2 n)$ times, since there are $O(\log^2 n)$ canonical sets in $\mathcal{C}(\tau)$. Following the same argument as in the proof of Lemma 1, we perform a binary search for only $O(\log n)$ of these canonical subsets in Step III, each of which takes $O(\log n)$ time. Furthermore, for an active set C and a cell τ , the cost of processing the segments of S_τ is $O(|S_\tau|)$, to which we have to add a one-time cost of $O(|C|)$, for finding the subsequence C_i of reached segments in C . Summing up over all cells and lists, we get that the total time for processing short segments is $O(n \log^2 n)$.

We process an unreached canonical set $C \in \mathcal{C}(\tau)$ when τ is active. This processing takes $O(|S_\tau| + \lambda_\tau)$ time, where λ_τ is the total number of middle portions of long segments in C that were added to L_{i+1} while processing τ . Since each segment is added to a BFS layer once, we have $\sum_\tau \lambda_\tau \leq n$. Since each non-empty grid cell is active at most twice and is associated with $O(\log^2 n)$ canonical sets, it follows that the total cost of processing unreached canonical sets is $O(n \log^2 n)$.

In conclusion, we thus obtain the main result of this section:

► **Theorem 2.** *Given a set A of n pairwise-disjoint segments in $\mathbb{R}^2$ and a parameter $r > 0$, breadth-first search on the proximity graph $G_r(A)$ can be performed in $O(n \log^2 n)$ time.*

2.3 Reverse shortest path

The reverse shortest path problem in segment proximity graphs is defined as follows. Given a pair of segments $s, t \in S$ and a parameter $k > 0$, compute the smallest value r^* of r such that $G_{r^*}(S)$ contains a path from s to t composed of at most k edges. Kaplan *et al.* [32] recently showed how to carefully adapt a subtle implementation of parametric search introduced by [8] (in a different context) to compute reverse shortest paths in disk graphs. We adapt this method to compute reverse shortest paths in segment proximity graphs.

The first step is to find a value $\hat{r}$ which is a $(1 + \varepsilon)$ -approximation of r^* , for a suitable sufficiently small $\varepsilon > 0$. We do this by a binary search (using the procedure of Section 2) between $\max_{e \neq e' \in S} \text{dist}(e, e')$ and $\min_{e \neq e' \in S} \text{dist}(e, e')$. This takes $O(\log \Delta)$ steps, where $\Delta = \max_{e \neq e' \in S} \text{dist}(e, e') / \min_{e \neq e' \in S} \text{dist}(e, e')$ is the spread of S , for a total cost of $O(n \log^2(n) \log \Delta)$ time. Once we have $\hat{r}$, we simulate the BFS at (the unknown) r^* , using $\hat{r}$ to form a good enough grid for running the procedure on it.

For the simulation to be efficient, we want the critical values of r at which the combinatorial structure of the decision procedure changes, to be distances between endpoints of segments and segments. (More generally, we want each critical value to depend on just two input objects.) To achieve this, we use a dynamic Voronoi diagram over the segments in $U_{i+1}(\tau)$, instead of a sweep over a union of disks, in step (2) of the processing of an active cell τ . This ensures that all critical values of r that the procedure generates are of the desired form, unlike computing the union of disks, in which additional kinds of critical values may arise.

As a preliminary step to this simulation, we use a modified version of standard two-dimensional range searching techniques, as elaborated in [8, 32], to confine r^* to an interval I containing a relatively small number of critical values. Then our simulation bifurcates at critical values in I , carefully balancing the progress of the simulated decision procedure, while narrowing down I . We omit further details, which are fairly similar to those in [32] (and [8]), and summarize with the following theorem.

► **Theorem 3.** *Given a set S of n pairwise-disjoint segments in $\mathbb{R}^2$, a designated pair of segments s, t , and a parameter k , the reverse-shortest-path problem in the proximity graph of S , i.e., finding the smallest r^* such that $G_{r^*}(S)$ contains a path from s to t composed of at most k edges, can be solved in $O^*(n^{6/5}) + O(n \log^2(n) \log \Delta)$ time, where Δ is the spread of S , i.e., $\Delta = \max_{e \neq e' \in S} \text{dist}(e, e') / \min_{e \neq e' \in S} \text{dist}(e, e')$.*

3 Dynamic Nearest-Neighbor Queries for Disjoint Segments

Let S be a set of n pairwise-disjoint segments in $\mathbb{R}^2$, which we refer to as *input segments*. Let P denote the set of endpoints of the segments in S . Let Q be another set of pairwise-disjoint segments, which we refer to as *query segments* and which we may not know in advance. The set S changes dynamically by insertions and deletions of segments, but its current segments are pairwise disjoint at all times. We also assume that when we query with a segment $e \in Q$, it is disjoint from all the segments in the current set S . (See the introduction and below.) For a segment $e \in Q$, let $\text{NN}(e, S) = \arg \min_{e' \in S} \text{dist}(e, e')$ be the segment in S closest to e . Our goal is to store S in a dynamic data structure that supports the following operations:⁵

⁵ Note that S is required to be pairwise disjoint at any given time, so after a segment e has been deleted, a segment intersecting e can be inserted into S as long as it does not intersect any of the current segments.

Insert(e): Given a segment $e \notin S$ that does not intersect any segment of S , insert e into S .

Delete(e): Given a segment $e \in S$, delete e from S .

Query(e): Given a segment $e \in Q$ that does not intersect any segment of (the current) S , return $\text{NN}(e, S)$.

We remark that such a data structure can be used to obtain an efficient, near-linear algorithm for the BFS problem discussed in the previous section. However, the algorithm presented in that section is simpler, as it avoids the need to use the fairly involved NN-structure, and is more efficient.

3.1 Main idea

To design a data structure for efficiently answering NN queries of this kind, we use the (obvious) property that the distance between two disjoint segments is always attained between an endpoint of one segment and some point – endpoint or interior point – of the other segment. Thus, to query with a segment $e \in Q$, we need to perform two subtasks:

(Q1) For each endpoint of e , compute its nearest segment in S .

(Q2) Find the point in P nearest to (the interior portion of) e .

Task (Q1) is simpler to accomplish, using the existing techniques in [2, 33, 37] (see below). Task (Q2) is considerably more difficult. The relevant algorithms described in [11, 12] are inherently static and off-line. An alternative approach (to task (Q2)) is to use range-searching machinery: Let $\Sigma(e)$ be the slab orthogonal to e , namely the slab that is bounded by the two lines that are orthogonal to e and pass through its endpoints. The points of $\Sigma(e) \cap P$ are precisely those points of P whose distance to e is attained at an interior point of e , and this distance is the same as that to ℓ_e , the line supporting e . Computing $P \cap \Sigma(e)$ as the union of a small number of prestored canonical subsets, so that each subset either lies fully above e or lies fully below e (within $\Sigma(e)$), is a major component of the algorithm, but a straightforward approach to this task involves simplex range searching machinery that is “doomed” to run in $O^*(n^{4/3})$ time. Instead, we use a more elaborate, and more efficient, range-searching based technique, that strongly exploits the disjointness of the segments in S and Q .

We now describe the data structure and the query/update procedures in detail. Without loss of generality, we assume that the slopes of the query segments lie in the range $[-1, +1]$. For handling query segments whose slopes do not lie in this range, we construct a similar data structure with the roles of the x - and y -axes flipped. For simplicity, we first describe the procedure for answering a *fixed-radius neighbor* query, namely, given a query segment $e \in Q$ and a parameter $r > 0$, return a segment of S that lies within distance at most r from e if there exists one. (This procedure suffices for our DFS application.) We then extend this procedure to answering NN queries, as described in the full version of the paper.

3.2 Data structure

Our data structure, which consists of multi-level trees, uses the following two classical data structures as building blocks:

Weight balanced trees. We use $\text{BB}[\alpha]$ trees [43] (see also [54]) to construct some of the underlying balanced binary trees in our structure. $\text{BB}[\alpha]$ trees have been extensively used for dynamic multi-level data structures [39, 54]. Roughly speaking, for $\alpha \in (0, 1)$, a $\text{BB}[\alpha]$ -tree T is a binary search tree in which the size of the subtrees is used as a criterion to balance the tree. The *weight* of a node v , denoted by $w(v)$, is 1 plus the number of nodes in the subtree

rooted at v . If u, z are the two children of v , then we require that $w(u), w(z) \geq \alpha w(v)$. The weight-balance condition during updates in T is maintained by performing single and double rotations; see [43, 54] for details. Assume that every internal node v stores a secondary data structure on the elements stored at the leaves of the subtree rooted at v . When a rotation is performed at a node u , we reconstruct the secondary structures stored at u from scratch. A crucial property of $\text{BB}[\alpha]$ -trees, which makes them suitable for multi-level data structures and which we also exploit, is that if the reconstruction of a secondary structure of size m takes $f(m) = \Omega(m)$ time, then an insert/delete operation on T can be performed in $O((f(n)/n) \log n)$ amortized time, where n is the total number of items stored in T [13, 38, 54].

Dynamic convex hull. Second, we use the dynamic convex-hull data structure of Overmars and van Leeuwen [45]. We skip the description of this well-known structure, but note that it supports efficient implementation of many operations involving convex hulls: A point can be inserted/deleted in $O(\log^2 n)$ time, and the following queries can be performed efficiently:

- (CH1) Given a vertical slab Σ and a halfplane γ , determine whether $P \cap (\Sigma \cap \gamma) \neq \emptyset$.
- (CH2) Given Σ, γ as above, return k points of $P \cap (\Sigma \cap \gamma)$ one by one, where $k \leq |P \cap (\Sigma \cap \gamma)|$.
- (CH3) Given a halfplane γ , compute the leftmost (or rightmost) point of $P \cap \gamma$.
- (CH4) Given a vertical slab Σ and a line ℓ that does not intersect $\text{conv}(P \cap \Sigma)$, return the closest point of $P \cap \Sigma$ to ℓ .

Queries (CH1), (CH3), and (CH4) take $O(\log^2 n)$ time each, and (CH2) takes $O((1+k) \log^2 n)$ time. We refer to this data structure as $\mathcal{C}(P)$. It can be constructed in $O(n \log n)$ time.

We are now ready to describe the overall data structure. We construct a data structure Φ for task (Q1), and two data structures Ψ_1, Ψ_2 for task (Q2).

Data structure for task (Q1). The data structure Φ for the task (Q1) is (relatively) simple. Since the segments in S are pairwise disjoint, their Euclidean Voronoi diagram has linear complexity [7]. Therefore we use the general machinery of Kaplan et al. [33], as recently improved by Liu [37] (see also an earlier, slightly less efficient, algorithm of Agarwal et al. [2]), to store S in a dynamic data structure of size $O(n \log n)$, so that a segment can be inserted into S in $O(\log^2 n)$ amortized time and deleted from S in $O(\log^4 n)$ amortized time, and $\text{NN}(q, S)$, for a query point $q \in \mathbb{R}^2$, can be computed in $O(\log^2 n)$ time.

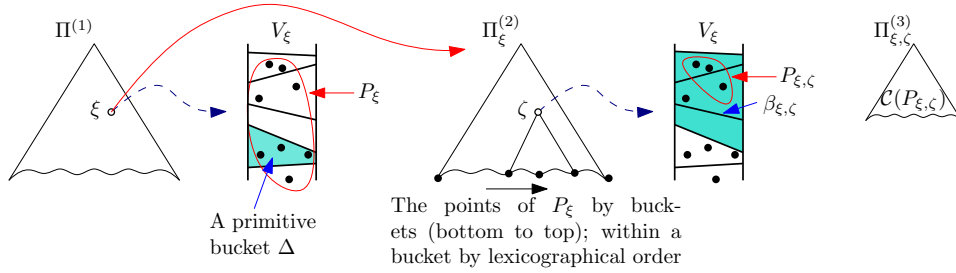
The first data structure for task (Q2). The first data structure, Ψ_1 , for (Q2) is constructed on P and combines a range tree with the aforementioned dynamic convex-hull data structure. It supports insertion and deletion of points, and enables us to add range restriction to (CH1)–(CH4) queries, i.e., perform these queries on a subset of points lying inside an axis-aligned rectangle. In particular, we can perform the following three queries on Ψ_1 :

- (CHY1) Given an axis-aligned rectangle $\square = X_\square \times Y_\square$ and a halfplane γ , determine whether $P \cap \square \cap \gamma$ is nonempty.
- (CHY2) Given a vertical slab $\sigma = X_\sigma \times \mathbb{R}$ and a halfplane γ , return the highest (or the lowest) point in $P \cap \sigma \cap \gamma$.
- (CHY3) Given an axis-aligned rectangle $\square = X_\square \times Y_\square$ and a line ℓ that does not intersect $\square$, return the point of $P \cap \square$ closest to ℓ .

The primary tree Υ of Ψ_1 is a 1-dimensional range tree on the y -coordinates of points in P , constructed as a $\text{BB}[\alpha]$ -tree [54]. Each node ξ of Υ is associated with a y -interval I_ξ and a *canonical* subset $P_\xi \subseteq P$ of points whose y -coordinates lie in I_ξ . At each node ξ of Υ , we

store $\mathcal{C}_\xi := \mathcal{C}(P_\xi)$, the dynamic convex-hull data structure on P_ξ . The size of Ψ_1 is $O(n \log n)$, and it can be constructed in $O(n \log^2 n)$ time. The update and query procedures for Ψ_1 , described in the full version of the paper, take $O(\log^3 n)$ time each, which is amortized for an update.

The second data structure for (Q2). The second data structure, Ψ_2 , is more involved and exploits the fact that the query segments are pairwise disjoint. Roughly speaking, Ψ_2 also enables range restriction to (CH1)–(CH4) queries, where the ranges are now halfplanes bounded by the (lines supporting the) segments of Q . See Section 3.3 and the full version of the paper. Ψ_2 is periodically reconstructed. More precisely, suppose the size of S was n_0 when Ψ_2 was last (re)constructed. Then we reconstruct Ψ_2 after performing a sequence of $n_0/2$ query/insert/delete operations, and charge the reconstruction time to these $n_0/2$ operations. Let R be the set of query segments that arrived after the last reconstruction of Ψ_2 ; we set $R = \emptyset$ when Ψ_2 is reconstructed, so all previous queries are discarded at this moment. It can easily be verified that $|R| \leq |S|/2$ at any given time. Ψ_2 consists of a three-level tree and stores both P and R . A schematic illustration of the structure is given in Figure 5.



■ **Figure 5** A schematic illustration of the data structure Ψ_2 .

The primary tree $\Pi^{(1)}$ is a $\text{BB}[\alpha]$ -tree on the x -coordinates of the points of P and of the endpoints of segments in R , and it is dynamically updated as points are inserted into or deleted from P and as new query segments arrive. We consider $\Pi^{(1)}$ as a segment tree on the x -projections of the segments in R . Each node ξ of $\Pi^{(1)}$ is associated with an x -interval I_ξ . Let $V_\xi := I_\xi \times \mathbb{R}$ be the vertical slab spanned by I_ξ , which we also associate with ξ . Set $P_\xi := P \cap V_\xi$. (Note that the endpoints of segments of R are not included in P_ξ .) By construction, a segment $e \in R$ is stored at ξ if $I_\xi \subseteq e^*$ and $I_{\pi(\xi)} \not\subseteq e^*$, where e^* is the x -projection of e and $\pi(\xi)$ is the parent of ξ . Let R_ξ be the set of segments of R stored at ξ , clipped to within V_ξ . Since the segments of R_ξ straddle V_ξ from side to side and are *pairwise nonintersecting*, they partition V_ξ into trapezoidal regions, which we refer to as *primitive buckets*. See Figure 5. We store the following three auxiliary structures at ξ .

1. We store the segments of R_ξ , sorted from bottom to top, in a balanced binary tree $\mathcal{B}_\xi$, so that we can determine, in $O(\log n)$ time, the primitive bucket of V_ξ that contains a point $p \in P_\xi$. A segment e that straddles V_ξ and that does not intersect any segment of R (namely, a new query segment) can be inserted into $\mathcal{B}_\xi$ in $O(\log n)$ time; e effectively splits the single primitive bucket that it crosses into two primitive buckets.
2. For each primitive bucket Δ of V_ξ , let $P_\Delta := P_\xi \cap \Delta$. We maintain $\mathcal{C}_\Delta := \mathcal{C}(P_\Delta)$, the dynamic convex-hull data structure on P_Δ .
3. Finally, we define a total order $\prec$ on the points of P_ξ as follows: Let $p, q \in P_\xi$. If the primitive bucket of V_ξ containing p lies below the one containing q , then $p \prec q$. If they

lie in the same primitive bucket, we order them lexicographically. We store P_ξ in a BB[α]-tree $\Pi_\xi^{(2)}$ using $\prec$ as the total order. Note that all points of P_ξ lying in the same primitive bucket are stored at consecutive leaves of $\Pi_\xi^{(2)}$.

An insertion of a segment e into the structure destroys the order of the points within the bucket that e crosses, and their new order has to be constructed; see below.

We call a node ζ of $\Pi_\xi^{(2)}$ *redundant* if all points in the subtree rooted at the parent of ζ belong to the same bucket, and *non-redundant* otherwise (the root of $\Pi_\xi^{(2)}$ is always non-redundant). At each internal node $\zeta \in \Pi_\xi^{(2)}$, let $P_{\xi,\zeta}$ be the set of points stored at the subtree rooted at ζ . We store $\mathcal{C}(P_{\xi,\zeta})$, denoted as a third-level substructure $\Pi_{\xi,\zeta}^{(3)}$, at ζ . At each non-redundant node ζ , we store the *bottom edge*, denoted by $\beta_{\xi,\zeta}$, of the lowest primitive bucket of V_ξ that contains a point of $P_{\xi,\zeta}$; the edges $\beta_{\xi,\zeta}$ facilitate the search in $\Pi_\xi^{(2)}$. Recall that $R = \emptyset$ when Ψ_2 is (re-)constructed, so initially V_ξ consists of only one primitive bucket, namely V_ξ itself, and the root of $\Pi_\xi^{(2)}$ is its only non-redundant node, which stores the dynamic convex-hull data structure on the full local set P_ξ .

This completes the description of the second data structure Ψ_2 for (Q2). Since (i) $\Pi_{\xi,\zeta}^{(3)}$ has linear size, (ii) each point of P or segment of R is stored at $O(\log n)$ nodes of $\Pi^{(1)}$, and (iii) each point of P_ξ is stored at $O(\log n)$ nodes of $\Pi_\xi^{(2)}$, it follows that the overall size of Ψ_2 is $O(n \log^2 n)$. Since $\Pi_{\xi,\zeta}^{(3)}$ can be constructed in $O(|P_{\xi,\zeta}| \log |P_{\xi,\zeta}|)$ time, $\Pi_\xi^{(2)}$ and $\Pi^{(1)}$ can be constructed in $O(|P_\xi| \log^2 |P_\xi|)$ and $O(n \log^3 n)$ time, respectively. The procedure for inserting a point into, or deleting from, Ψ_2 , in $O(\log^5 n)$ amortized time, is described in the full version of the paper. Here we describe the procedure for inserting a segment into Ψ_2 .

Inserting a segment into Ψ_2 . Recall that Ψ_2 stores both P and R , so when we query Ψ_2 with a segment $e \in R$ that is disjoint from (the current) S , we first insert e into Ψ_2 . We begin by inserting the endpoints of e into $\Pi^{(1)}$, as described in the full version of the paper, and then insert the segment e itself into Ψ_2 . Recall that $\Pi^{(1)}$ is a segment tree on the segments of R , so e is stored at a node ξ if $I_\xi \subseteq e^*$ but $I_{\pi(\xi)} \not\subseteq e^*$. Let ξ be such a node. Insertion of e at ξ splits a primitive bucket of V_ξ , which in turn updates the ordering of the points in P_ξ . We thus update the secondary structures stored at ξ , as follows. We clip e within V_ξ , and continue to denote by e the clipped segment. Let e^+ (resp., e^-) be the halfplane lying above (resp., below) the line supporting e . By searching with e in $\mathcal{B}_\xi$, we find the primitive bucket Δ of V_ξ that e crosses. We split Δ into two primitive buckets $\Delta^+ = \Delta \cap e^+$ and $\Delta^- = \Delta \cap e^-$. Let $P_{\Delta^-} = P_\Delta \cap \Delta^-$ and $P_{\Delta^+} = P_\Delta \cap \Delta^+$. We construct $\mathcal{C}_{\Delta^-}$ and $\mathcal{C}_{\Delta^+}$, from $\mathcal{C}_\Delta$: by performing (CH2) queries on $\mathcal{C}_\Delta$ with the halfplanes e^+ and e^- , we report the points of P_Δ lying in e^+ and e^- , respectively, in a lock-step manner, until we exhaust all the points in one of the halfplanes. Suppose, without loss of generality, $|P_{\Delta^-}| \leq |P_{\Delta^+}|$. We have P_{Δ^-} at our disposal, so we construct $\mathcal{C}_{\Delta^-}$ from an empty structure by inserting the points of P_{Δ^-} , and we obtain $\mathcal{C}_{\Delta^+}$ by deleting the points of P_{Δ^-} from $\mathcal{C}_\Delta$.

The insertion of e changes the ordering of points in P_Δ , namely, now $p \prec q$ for every $p \in P_{\Delta^-}$ and $q \in P_{\Delta^+}$. We thus delete the points of P_{Δ^-} from $\Pi_\xi^{(2)}$ and re-insert them with their new key, using the procedure described above, so that they appear before P_{Δ^+} in $\Pi_\xi^{(2)}$. Finally, for all non-redundant nodes ζ of $\Pi_\xi^{(2)}$ whose leftmost leaf stores a point of P_{Δ^+} , we update β_ζ to e . There are $O(\log n)$ such nodes and they lie along two root-to-leaf paths of $\Pi_\xi^{(2)}$, so this can be accomplished in $O(\log n)$ time.

We now bound the time spent to insert e . The insertion of the endpoints of e into $\Pi^{(1)}$ takes $O(\log^5 n)$ amortized time, as discussed in the full version. Let ξ be a node of $\Pi^{(1)}$ at which e is stored. We spend $O(\log n)$ time in inserting e into $\mathcal{B}_\xi$. The total time spent

in computing the smaller of the two sets $P_{\Delta-}$, $P_{\Delta+}$ (say $P_{\Delta-}$), and constructing $\mathcal{C}_{\Delta-}$ and $\mathcal{C}_{\Delta+}$ from $\mathcal{C}_{\Delta}$ is $O((1 + |P_{\Delta-}|) \log^2 n)$. Finally, deleting and re-inserting the points of $P_{\Delta-}$ in $\Pi_{\xi}^{(2)}$ takes $O(|P_{\Delta-}| \log^3 n)$ amortized time. Summing up all costs, the total time spent in inserting e at the secondary structures stored at ξ is $O(\log^3 n + |P_{\Delta-}| \log^3 n)$. We charge $O(\log^3 n)$ cost to the insertion of e and pay the $O(|P_{\Delta-}| \log^3 n)$ cost by charging $O(\log^3 n)$ units of credit to each point of $P_{\Delta-}$. One can show (see below) that there is enough credit to pay for these operations.

We obtained $P_{\Delta-}$, $P_{\Delta+}$ by splitting P_{Δ} , in time proportional to $\min\{|P_{\Delta-}|, |P_{\Delta+}|\}$ (times $O(\log^3 n)$). Following a standard argument, it can be shown that this cost (over all insertions of segments to R) can be paid by charging a total of $O(\log^4 n)$ units to each point of P_{ξ} . Since a point p of P is stored at $O(\log n)$ nodes of $\Pi^{(1)}$, $O(\log^5 n)$ units of credit assigned to p when it was inserted into P are sufficient to pay for all the costs charged to it (and these $O(\log^5 n)$ credits are charged against the insertion cost of p into P). Putting all the ingredients together, the amortized cost of inserting a segment into R is $O(\log^5 n)$.

Similar considerations apply to the other operations. See the full version for details.

3.3 Fixed-radius neighbor queries

Let $e = pq \in R$ be a query segment, and let $r > 0$ be a parameter (which is part of the query). We wish to report a segment of S that is within distance r from e if there exists one. Without loss of generality, assume that the slope of e is in the range $[0, 1]$ (other ranges are handled similarly). Let $\Sigma(e)$ be the slab orthogonal to e , as defined earlier.

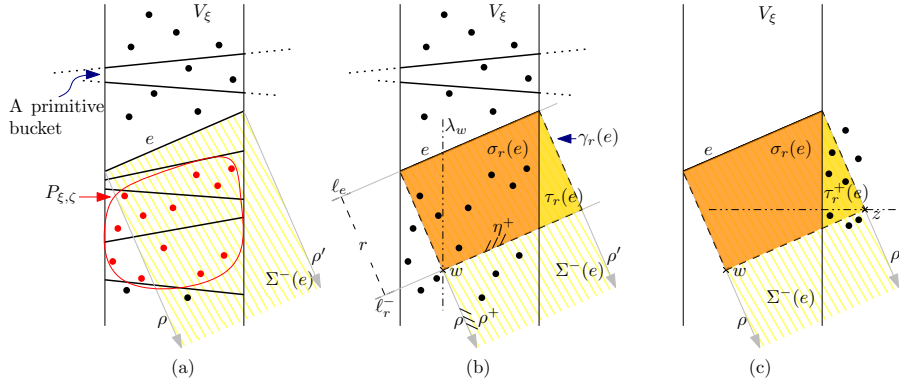


Figure 6 (a) Querying with a segment e , and the slabs V_{ξ} and $\Sigma^-(e)$. (b) Testing $\sigma_r(e)$ for emptiness by splitting $P_{\xi} \cap \Sigma^-(e)$ into a left part and a right part at the vertex w (assuming that $w \in V_{\xi}$), and performing a halfplane emptiness query for each part. (c) Testing $\tau_r(e)$ by splitting it into two right axis-aligned triangles.

We first query Φ with p and q and return their nearest segments in S . If either of them is within distance r from e , we return it and stop. So assume that neither of them is within distance r from e . In this case, it suffices to determine whether there is any point of $P \cap \Sigma(e)$ within distance r from the line ℓ_e supporting e , which we do by querying Ψ_1 and Ψ_2 .

We first insert e into Ψ_2 , as described above. Recall that e is stored at $O(\log n)$ nodes of the primary tree $\Pi^{(1)}$ of Ψ_2 . These nodes partition e into $O(\log n)$ subsegments, each lying within and straddling the vertical slab associated with that node. This split of e also partitions the slab $\Sigma(e)$ into $O(\log n)$ parallel subslabs. We search within each subslab separately. More precisely, let $\xi \in \Pi^{(1)}$ be one of the nodes where e is stored. With a slight abuse of notation, we use e to denote the query segment clipped to within V_{ξ} , and use $\Sigma(e)$ to denote the slab orthogonal to this clipped segment. We now describe the procedure for reporting a point of $\Sigma(e) \cap P$ within distance r from e (or from ℓ_e), if one exists.

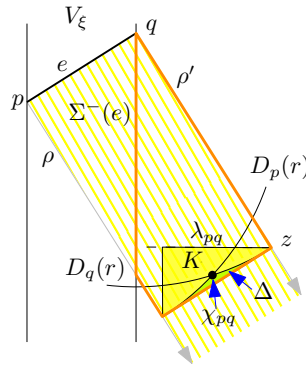
We partition $\Sigma(e)$ into two semi-slabs $\Sigma^-(e)$ and $\Sigma^+(e)$, lying below and above e respectively. We describe the procedure for finding a desired point in $\Sigma^-(e) \cap P$; a symmetric procedure works for $\Sigma^+(e)$. $\Sigma^-(e)$ is bounded by e and by two rays orthogonal to e and emanating from its endpoints, one of which, denoted by ρ' , is fully disjoint from V_ξ ; we denote the other ray as ρ . See Figure 6(a). Let $\gamma_r(e) \subset \Sigma^-(e)$ denote the rectangle with e as its upper edge and its lower edge lying on the line ℓ_r^- parallel to e at distance r below it; $\gamma_r(e)$ determines the set of points in $\Sigma^-(e)$ within distance r from e (or from ℓ_e). One of the side edges of $\gamma_r(e)$ is contained in ρ , and the other side is contained in ρ' and lies outside V_ξ . Our goal is to determine whether $P \cap \gamma_r(e) \neq \emptyset$ and return a point of $P \cap \gamma_r(e)$ if the answer is YES.

We partition $\gamma_r(e)$ into two regions: $\sigma_r(e) = \gamma_r(e) \cap V_\xi$ and $\tau_r(e) = \gamma_r(e) \setminus V_\xi$. Since the slope of e is positive, $\tau_r(e)$ lies to the right of V_ξ . If the right boundary line of V_ξ intersects the lower edge of $\gamma_r(e)$ then $\sigma_r(e)$ is a trapezoid and $\tau_r(e)$ is a right-angle triangle, otherwise (if that line intersects the left side edge of $\gamma_r(e)$) $\sigma_r(e)$ is a right-angle triangle and $\tau_r(e)$ is a trapezoid. We check whether either of $\sigma_r(e)$, $\tau_r(e)$ contains a point of P , as follows.

Testing $\sigma_r(e)$. We focus on the case when $\sigma_r(e)$ is a trapezoid; the case when $\sigma_r(e)$ is a triangle can be viewed as a degenerate instance of this case and is simpler to handle. Since $\sigma_r(e) \subset V_\xi$, $\sigma_r(e) \cap P = \sigma_r(e) \cap P_\xi$, so it suffices to test whether $\sigma_r(e) \cap P_\xi \neq \emptyset$. See Figure 6(b). Note that e is already inserted into the secondary structures at ξ . Hence, using the secondary tree $\Pi_\xi^{(2)}$ stored at ξ , we compute, in $O(\log n)$ time, the points of P_ξ lying below e (i.e., points lying in primitive buckets below e) as the union of $O(\log n)$ canonical subsets, each stored at some node ζ of $\Pi_\xi^{(2)}$. Let $P_{\xi,\zeta}$ be such a canonical subset. Let w be the bottom vertex of $\sigma_r(e)$, let λ_w denote the vertical line through w , and let λ^-, λ^+ be the vertical halfplanes lying to the left and to the right of λ_w , respectively. Again, see Figure 6(b). Let ρ^+ be the halfplane lying above the line supporting the ray ρ , and let η^+ be the halfplane lying above the line ℓ_r^- . A point $p \in P_{\xi,\zeta}$ lying in λ^- (resp., λ^+) lies in $\sigma_r(e)$ if and only if it lies in the halfplane ρ^+ (resp., η^+). See Figure 6(b). Therefore, by performing two (CH1) queries on $\Pi_{\xi,\zeta}^{(3)}$, we determine whether $P_{\xi,\zeta} \cap (\lambda^- \cap \rho^+) \neq \emptyset$ and whether $P_{\xi,\zeta} \cap (\lambda^+ \cap \eta^+) \neq \emptyset$. If the answer of either of these tests is YES, we conclude that $\sigma_r(e) \cap P_\xi \neq \emptyset$, return a point in it and stop. By repeating this procedure for all $O(\log n)$ canonical nodes, we determine, in $O(\log^3 n)$ time, whether $P_\xi \cap \sigma_r(e) \neq \emptyset$, and return a point in this intersection if it is indeed nonempty.

Testing $\tau_r(e)$. Abusing the notation slightly, let p and q be the endpoints of the clipped segment e (within V_ξ). We first consider the case where $\tau_r(e)$ is a right-angle triangle. Let z be the right-angle (rightmost) vertex of the triangle $\tau_r(e)$; z is adjacent to the bottom edge of $\tau_r(e)$, which is contained in ℓ_r^- , and to the right edge of $\tau_r(e)$, which is contained in ρ' . We partition $\tau_r(e)$ into subtriangles $\tau_r^+(e)$ and $\tau_r^-(e)$ by drawing a horizontal edge from z , where $\tau_r^+(e)$ lies above the horizontal edge and $\tau_r^-(e)$ lies below the edge. See Figure 6(c). Since each of these subtriangles is an axis-aligned right-angle triangle (i.e., with horizontal and vertical sides), it can be expressed as the intersection of an axis-aligned rectangle and a halfplane. Therefore, by performing a (CHY1) query on Ψ_1 , we determine, in $O(\log^3 n)$ time, whether either of these subtriangles contains a point of P ; if so we return one such point.

Next, consider the case when $\tau_r(e)$ is a trapezoid. The difficulty in this case is that the two parallel edges of $\tau_r(e)$ lying on ρ, ρ' can be arbitrarily long (when r is much larger than the length of e), and it is not clear how to decompose $\tau_r(e)$ into $O(1)$ right-angle triangles. Instead, we use the following approach. Since, as we have assumed, the slope of e is in the range $[0, 1]$, it can be verified that $|e| \leq r$ in this case (namely, when $\tau_r(e)$ is a trapezoid),



■ **Figure 7** $\tau_r(e)$ is a trapezoid.

which implies that the two disks $D_p(r)$ and $D_q(r)$, of radius r around p and q , intersect, and one of the intersection points of their boundaries, denoted by χ_{pq} , lies in $\Sigma^-(e)$ and to the right of V_ξ ; see Figure 7. Our preliminary querying in Φ with p and q allows us to assume that $D_p(r) \cup D_q(r) \cap P = \emptyset$. Therefore, if $P \cap \tau_r(e) \neq \emptyset$, any point of $P \cap \tau_r(e)$ lies in $\Delta = \tau_r(e) \setminus (D_p(r) \cup D_q(r))$. Let λ_{pq} be the horizontal line that supports Δ from above, i.e., the lowest horizontal line such that Δ lies below λ_{pq} . Since χ_{pq} lies to the right of both p and q , the two disks $D_p(r)$ and $D_q(r)$ have positive slopes at χ_{pq} , and λ_{pq} passes through the rightmost vertex z of $\tau_r(e)$. See Figure 7.

Let λ_{pq}^- be the halfplane lying below λ_{pq} . Let K be the right-angle triangle whose hypotenuse is the bottom edge of $\tau_r(e)$ and whose top side is contained in λ_{pq} . It is easily checked that Δ is contained in K , and that $K \setminus \Delta$ is contained in the union of the two disks. It thus suffices to test whether $K \cap P$ is nonempty, which can be done using the same technique described above, in $O(\log^3 n)$ time.

This completes the description of the procedure for determining whether $\gamma_r(e) \cap P \neq \emptyset$. A symmetric procedure determines in $O(\log^3 n)$ time whether $\Sigma^+(e) \cap P$ contains a point within distance r from e . By repeating this procedure for all $O(\log n)$ nodes of $\Pi^{(1)}$ at which e is stored, the total time spent in querying Ψ_2 is $O(\log^4 n)$. Taking into account the time spent in inserting e into Ψ_2 , and adding the time taken in querying Φ with the endpoints of e , the total amortized query time is $O(\log^5 n)$.

Putting everything together, we obtain the main result of this section:

► **Theorem 4.** *Let S be a set of n pairwise-disjoint segments in $\mathbb{R}^2$, and let Q be another set of pairwise-disjoint segments in $\mathbb{R}^2$ that are not known in advance. S can be stored in a dynamic data structure that can handle insertion and deletion of segments, assuming the set S remains pairwise disjoint after each update, and that can answer fixed-radius-neighbor and nearest-neighbor queries for a query segment $e \in Q$ that does not intersect any segment in (the current) S . The amortized query and update times are $O(\log^5 n)$. If the set Q is known in advance (but not the sequence of queries), the query and (amortized) update time can be improved to $O(\log^4 n)$.*

References

- 1 P. K. Agarwal, Simplex range searching and its variants: A review, in *Journey through Discrete Mathematics: A Tribute to Jiří Matoušek*, M. Loebl, J. Nešetřil and R. Thomas, editors, Springer Verlag, Berlin-Heidelberg, 2017, pages 1–30.

- 2 P. K. Agarwal, A. Efrat and M. Sharir, Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications, *SIAM J. Comput.* 29(3) (1999), 912–953.
- 3 P. K. Agarwal, M. J. Katz, and M. Sharir, On reverse shortest paths in geometric proximity graphs, *Comput. Geom. Theory Appl.* 117 (2024), Art. 102053. Also in *Proc. 33rd Internat. Sympos. on Algorithms and Computation*, 2022, 42:1–42:19.
- 4 P. K. Agarwal and J. Matoušek, Dynamic half-space range reporting and its applications, *Algorithmica* 13 (1995), 325–345.
- 5 P. K. Agarwal, M. H. Overmars, and M. Sharir, Computing maximally separated sets in the plane, *SIAM J. Comput.*, 36(3) (2006), 815–834.
- 6 N. Alon, J. Pach, R. Pinchasi, R. Radoicic and M. Sharir, Crossing patterns of semi-algebraic sets, *J. Combin. Theory Ser. A* 111 (2005), 310–326.
- 7 F. Aurenhammer, R. Klein and D.-T. Lee, *Voronoi Diagrams and Delaunay Triangulations*, World Scientific, Singapore 2013.
- 8 R. Ben Avraham, O. Filtser, H. Kaplan, M. J. Katz and M. Sharir, The discrete and semicontinuous Fréchet distance with shortcuts via approximate distance counting and selection, *ACM Trans. Algorithms* 11 (2015), Art. 29.
- 9 J. L. Bentley and J. B. Saxe, Decomposable searching problems I: Static-to-dynamic transformation, *J. Algorithms* 1(4) (1980), 301–358.
- 10 M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd Edition, Springer Verlag, Berlin, 2008.
- 11 S. Bespamyatnikh, Computing closest points for segments, *Int. J. Comput. Geom. Appl.* 13(5) (2003), 419–438.
- 12 S. Bespamyatnikh and J. Snoeyink, Queries with segments in Voronoi diagrams, *Comput. Geom. Theory Appl.* 16 (2000), 23–33.
- 13 N. Blum and K. Mehlhorn, On the average number of rebalancing operations in weight-balanced trees, *Theor. Comput. Sci.* 11 (1980), 303–320.
- 14 H. Breu and D. G. Kirkpatrick, Unit disk graph recognition is NP-hard, *Comput. Geom. Theory Appl.*, 9(1-2) (1998), 3–24.
- 15 S. Cabello and M. Ježić, Shortest paths in intersection graphs of unit disks, *Comput. Geom. Theory Appl.* 48 (2015), 360–367.
- 16 T. M. Chan, Optimal partition trees, *Discrete Comput. Geom.*, 47 (2012), 661–690.
- 17 T. M. Chan, Dynamic generalized closest pair: Revisiting Eppstein’s technique, *Proc. 3rd Sympos. Simplicity in Algorithms (SOSA)*, 2020, 33–37.
- 18 T. M. Chan, Finding triangles and other small subgraphs in geometric intersection graphs, *Proc. 34th ACM-SIAM Symposium on Discrete Algorithms*, 2023, 1777–1805.
- 19 T. M. Chan and D. Skrepetos, All-pairs shortest paths in unit-disk graphs in slightly subquadratic time, *Proc. 27th Internat. Sympos. on Algorithms and Computation*, pages 24:1–24:13, 2016.
- 20 T. M. Chan and D. Skrepetos, All-pairs shortest paths in geometric intersection graphs, *J. Comput. Geom. Theory Appl.* 10(1) (2019), 27–41.
- 21 T. M. Chan and D. Skrepetos, Approximate shortest paths and distance oracles in weighted unit-disk graphs, *J. Comput. Geom. Theory Appl.* 10(2) (2019), 3–20.
- 22 B. Chazelle, H. Edelsbrunner, L. Guibas and M. Sharir, Algorithms for bichromatic line-segment problems and polyhedral terrains, *Algorithmica* 11 (1994), 116–132.
- 23 B. N. Clark, C. J. Colbourn, and D. S. Johnson, Unit disk graphs, *Discrete Math.* 86(1-3) (1990), 165–177.
- 24 D. Conlon, J. Fox, J. Pach, B. Sudakov, and A. Suk, Ramsey-type results for semi-algebraic hypergraphs, *Trans. Amer. Math. Soc.* 366 (2014), 5043–5065.
- 25 D. Eppstein, Fast hierarchical clustering and other applications of dynamic closest pairs, *Discrete Comput. Geom.* 13 (1995), 111–122.
- 26 J. Erickson, New lower bounds for Hopcroft’s problem, *Discrete Comput. Geom.* 16 (1996), 389–418.

- 27 A. V. Fishkin, Disk graphs: A short survey, *Proc. 1st Internat. Workshop on Approximation and Online Algorithms*, volume 2909 of *Lecture Notes in Computer Science*, pages 260–264, 2003.
- 28 G. D. da Fonseca, V. G. P. de Sá, and C. M. H. de Figueiredo, Shifting coresets: Obtaining linear-time approximations for unit disk graphs and other geometric intersection graphs, *Int. J. Comput. Geom. Appl.* 27(4) (2017), 255–276.
- 29 J. Fox, J. Pach, and A. Suk, Density and regularity theorems for semi-algebraic hypergraphs, *Proc. 26th ACM-SIAM Symposium on Discrete Algorithms*, 2015, pp. 1517–1530.
- 30 W. Gálvez, A. Khan, M. Mari, T. Mömke, M. R. Pittu and A. Wiese, A 3-approximation algorithm for maximum independent set of rectangles, *Proc. 33rd ACM-SIAM Symposium on Discrete Algorithms*, 2022, 894–905.
- 31 J. Gao and L. Zhang, Well-separated pair decomposition for the unit-disk graph metric and its applications, *SIAM J. Comput.*, 35(1) (2005), 151–169.
- 32 H. Kaplan, M. J. Katz, R. Saban and M. Sharir, The unweighted and weighted reverse shortest path problem for disk graphs, *Proc. 31st European Sympos. Algorithms* (2023), 67:1–67:14. Also in arXiv:2307.14663.
- 33 H. Kaplan, W. Mulzer, L. Roditty, P. Seiferth and M. Sharir, Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications, *Discrete Comput. Geom.* 64 (2020), 838–904.
- 34 H. Kaplan, R. E. Tarjan and K. Tsoutsoulouklis, Faster kinetic heaps and their use in broadcast scheduling, *Proc. 12th ACM-SIAM Sympos. on Discrete Algorithms*, 2001, 836–844.
- 35 K. Klost, An algorithmic framework for the single source shortest path problem with applications to disk graphs, *Comput. Geom. Theory Appl.* 111 (2023), Art. 101979.
- 36 J. Kratochvíl and J. Matoušek, Intersection graphs of segments, *J. Combinat. Theory, Ser. B* 62 (1994), 289–315.
- 37 C.-H. Liu, Nearly optimal planar k nearest neighbors queries under general distance functions, *SIAM J. Comput.* 51(3) (2022), 723–765.
- 38 K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer, 1984.
- 39 K. Mehlhorn, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, Springer, 1984.
- 40 K. Mehlhorn, S. Näher, Dynamic fractional cascading, *Algorithmica* 5 (1990), 215–241.
- 41 A. Meyerson, Online facility location, *Proc. 42nd IEEE Sympos. Foundations of Computer Science (FOCS)*, 2001, 426–431.
- 42 J. S. B. Mitchell, Approximating maximum independent set for rectangles in the plane, *Proc. 62nd IEEE Sympos. on Foundations of Computer Science*, 2021, 339–350.
- 43 J. Nievergelt and E. M. Reingold, Binary search trees of bounded balance, *SIAM J. Comput.* 2 (1973), 33–43.
- 44 M. H. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science 156, Springer, 1983.
- 45 M. H. Overmars and J. van Leeuwen, Maintenance of configurations in the plane, *J. Comput. Syst. Sci.* 23 (1981), 166–204.
- 46 F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer, Berlin, 1985.
- 47 L. Roditty and M. Segal, On bounded leg shortest paths problems, *Algorithmica* 59(4) (2011), 583–600.
- 48 M. Schaefer and D. Štefankovič, Decidability of string graphs, *Proc. Sympos. on Theory of Computing (STOC)*, 2001, 241–246.
- 49 M. Sharir and P.K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, Cambridge-New York-Melbourne, 1995.
- 50 H. Wang, Algorithms for computing closest points for segments, *Proc. 41st Internat. Sympos. on Theoretical Aspects of Computer Science (STACS)*, 2024, 8:1–58:17.

- 51 H. Wang and J. Xue, Near-optimal algorithms for shortest paths in weighted unit-disk graphs, *Discrete Comput. Geom.* 64(4) (2020), 1141–1166.
- 52 H. Wang and Y. Zhao, Reverse shortest path problem for unit-disk graphs, *Proc. 17th Internat. Sympos. on Algorithms and Data Structures*, 2021, 655–668.
- 53 H. Wang and Y. Zhao, Reverse shortest path problem in weighted unit-disk graphs, *Proc. 16th Internat. Conf. on Algorithms and Computation*, 2022, 135–146.
- 54 D. E. Willard and George S. Lueker, Adding range restriction capability to dynamic data structures, *J. ACM* 32 (1985), 597–617.