Circle-Segment Intersection Queries in Connected Geometric Graphs

Peyman Afshani
Aarhus University, Denmark

Yannick Bosch
University of Konstanz, Germany

Sabine Storandt
University of Konstanz, Germany

Abstract

In this paper, we study the problem of efficiently reporting all intersections between a given set of line segments in the plane and a query circle, focusing on the case where the segments form the edges of a connected geometric graph. While previous data structures for circle-segment intersection queries on general segment sets incur high space or query time costs, we exploit the connectivity of the input to obtain significantly improved performance. In fact, we propose a new circle-segment intersection data structure that can be constructed in $\mathcal{O}((n+C)\log^3 n)$ time and space on connected graphs with n edges and C edge crossings. It answers intersection queries in $\mathcal{O}(k\log^3 n)$ time, where k denotes the output size. Our method relies on the construction of efficient circle-graph intersection oracles as well as a novel linear-time algorithm to partition the edges of the graph into balanced, connected components, which might be of independent interest. In a proof-of-concept experimental study on real-world road networks, we show that our novel data structure also performs well in practice. Even on networks with millions of edges, the construction time is within minutes and queries are answered in a few milliseconds.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Data structures design and analysis

Keywords and phrases Intersection data structure, Graph partitioning, Dobkin-Kirkpatrick hierarchy

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2025.3

Acknowledgements This work was initiated at Dagstuhl Seminar 25191 "Adaptive and Scalable Data Structures" (2025-05-04 – 2025-05-09) `https://www.dagstuhl.de/25191`. We thank Schloss Dagstuhl -– Leibniz-Zentrum für Informatik for their support.

1 Introduction

Given a set $\mathcal{S}$ of geometric objects and a query object $\mathcal{Q}$, reporting the intersections of $\mathcal{Q}$ with elements in $\mathcal{S}$ is a fundamental task in computational geometry. In this work, we focus on the scenario where $\mathcal{S}$ is a set of segments in the Euclidean plane, and $\mathcal{Q}$ is a query circle whose center and radius is revealed at query time. The goal is to preprocess $\mathcal{S}$ into a data structure that allows to report circle-segment intersections efficiently.

Notably, there are two different types of intersections that can occur between a segment $S = (p, q)$, specified by its endpoints $p, q \in \mathbb{R}^2$, and a query circle $\mathcal{Q} = (c, r)$, specified by its center $c \in \mathbb{R}^2$ and its radius $r \in \mathbb{R}_0^+$:

- **Type-(i) intersection:** This intersection occurs if $d_2(c, p) \leq r$ and $d_2(c, q) > r$, that is, one endpoint of S is inside the query circle and the other one outside of it.
- **Type-(ii) intersection:** This intersection occurs if $d_2(c, p) > r$ and $d_2(c, q) > r$ but $d_2(c, S) \leq r$, that is, both endpoints of S are outside of the query circle but part of the segment is inside the circle.

Both intersection types are visualized in Figure 1.

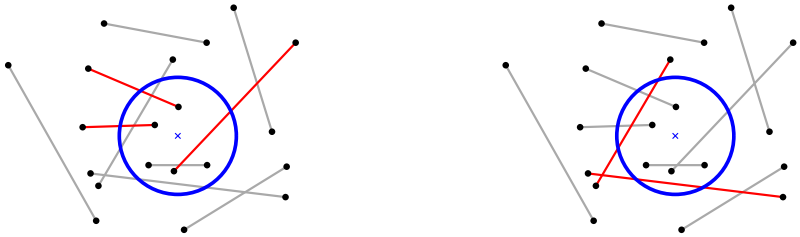

Figure 1 Circle-segment query example. Left: Segments marked red have a type-(i) intersection with the query circle. Right: Segments marked red have a type(ii)-intersection with the query circle.

Closely related to circle queries are disk queries, where segments fully contained within the disk need to be reported as well. So for disk queries, if a data structure identifies the set of segment endpoints that are contained in the disk, one can batch report all incident segments, as those are either completely contained in the disk or have a type(i)-intersection with its boundary circle. Thus, circle queries are somewhat more intricate than disk queries, as here this batch reporting cannot be applied if output-sensitive query times are desired. In [16], different data structures for each intersection type were proposed to cater for circle-segment and disk-segment intersection queries on a given set of n input segments. For fixed radius, after an $\mathcal{O}(n^{3/2}\log n)$ construction phase, a data structure of size $\mathcal{O}(n\log^2 n)$ allows to answer circle or disk queries in time $\mathcal{O}(\sqrt{n}\log^2 n + k)$ or $\mathcal{O}(\sqrt{n}\log n + k\log n)$, where k denotes the output size. For variable radius, a data structure of size $\mathcal{O}(n)$ was shown to answer queries in $\mathcal{O}(n^{2/3+\varepsilon}+k)$ for $\varepsilon > 0$. Significantly improved query times of $\mathcal{O}(\log n + k)$ are also possible but at the cost of a much higher space consumption of $\mathcal{O}(n^{3+\varepsilon})$. Other trade-offs between data structure size and query time can be obtained as well. For example, the query times for the fixed radius case can be matched for the variable radius scenario using a data structure of size $\mathcal{O}(n^{3/2+\varepsilon})$. In all these scenarios, the data structure for detecting type-(ii) intersections dominates the space consumption and the query time.

In this paper, we consider a special case of the circle- or disk-segment intersection problem, in which the segments in $\mathcal{S}$ are the edges of a connected geometric graph G. There are numerous applications of circle-edge or disk-edge intersection queries:

- **Nearest Edges.** Given a geometric graph G and a query point p, identifying the edges of G that are within distance r of p is equivalent to a disk query (p, r) on the edge segment set. This query type is useful, for example, for the construction of segment proximity graphs [1] or in edge facility location problems [20].
- **Graph parameter computation.** There are several graph parameters used to classify geometric graphs whose definition relies on the number and type of disk-segment intersections. This includes the notions of τ-lankiness [18] and λ-low-density [11].
- **Motion planning for circular robots.** Obstacle collision avoidance for robots that have to navigate a mesh or are supposed to sweep a given polygon (e.g. a floor plan) requires to detect whether the robot boundary intersects any obstacle [21]. For circular robots, a circle-segment reporting data structure can be leveraged to detect collisions.
- **Map matching.** Let G be a road network where edge costs describe travel times. A map matching query is specified by a sequence of m measurements $(p_i, r_i, t_i), i = 1, \ldots, m$ (usually obtained via GPS) where p_i denotes the measured position, r_i the error radius (stemming from measurement imprecision) and t_i the time stamp of the measurement. The goal is to match each position p_i to a location l_i in G that is within distance r_i of p_i such that l_{i+1} can be reached from l_i on a path in G with travel time at most $t_{i+1} - t_i$.

Figure 2 Exemplary circle-segment intersection queries with different radii on a connected road network. Query circles are blue and the respective intersecting segments are highlighted in red.

It was proven in [8] that a feasible map matching solution can be fully characterized by the intersection points of the circles (p_i, r_i) with edges in G, and an efficient algorithm was proposed that leverages a circle-segment data structure.

Some example queries on a city road network are visualized in Figure 2. We show that we can utilize the connectivity of G to significantly improve the construction time, the space consumption, and the query times of circle- or disk-segment intersection data structures in theory and practice.

1.1 Related Work

There are interesting special cases of the circle-segment or disk-segment intersection problem, which have been studied in the literature:

- If all input segments have length zero, the input is simply a set of points. Thus, the problem degenerates into reporting all points on the boundary of a query circle or inside a query disk. The latter is also known as the disk range searching problem. Disk range searching in $\mathbb{R}^2$ can be transformed into halfspace range searching in $\mathbb{R}^3$, for which a data structure of size $\mathcal{O}(n \log n)$ allows to answer queries in $\mathcal{O}(\log n + k)$ [5].
- If the circle has a radius of zero, it degenerates into a point. Then, the circle/disk query asks for the segments that contain the query point, also known as segment stabbing problem. If the segments are assumed to be disjoint, queries can be answered in $\mathcal{O}(\log n)$ after a $\mathcal{O}(n \log n)$ preprocessing phase that produces a data structure of linear size [12].

For the general case, the data structures for type-(i) and type-(ii) circle/disk-segment intersection described in [16] are the state-of-the-art. For variable radius, first a partition tree on the segment endpoints is constructed [19] after projecting the points to $\mathbb{R}^3$. Here, the point set is recursively partitioned using simplices, such that in the resulting hierarchical tree, the number of points in each child partition is a constant fraction of the number of points in the parent partition and only a bounded number of children intersect a query object at

each level. To only retrieve the intersecting segments with an endpoint inside the circle, the data structure is augmented such that halfplane composition queries can be answered. Also type-(ii) intersections can be identified with the help of halfplane query compositions. This gives rise to a data structure of size $\mathcal{O}(n)$ with query times in $\mathcal{O}(n^{2/3+\varepsilon} + k)$ for $\varepsilon > 0$, as well as the other trade-offs mentioned in the introduction.

For fixed radius, type-(i) and type-(ii) intersections are computed with two different data structures. For type-(i), first a stabbing path is computed on the segment endpoints. This path is guaranteed to intersect a given query circle at most $\mathcal{O}(\sqrt{n})$ times. On this path, a tree data structure is constructed, which is augmented with a halfplane query data structure at each node. Using geometric transformations and dualization, this allows to identify the segments with an endpoint outside and an endpoint inside the query circle. For type-(ii) intersections, a reduction to triangle stabbing is described and proper data structures for this use case are leveraged. In [8], the data structure for circle-segment queries with fixed radius r from [16] was refined and improved for the case in which the input segments form a path to show improved theoretical running times for map matching. However, for practical implementation, an AABB tree is used in [8], as the nested data structures described in [16] are rather intricate and expensive to construct. The AABB tree, available as a ready-to-use implementation in CGAL, is a hierarchical data structure in which each node corresponds to an axis-aligned bounding box (AABB) that encapsulates the set of geometric primitives associated with its descendant leaf nodes (in this case, the input segments). While often fast in practice, it cannot guarantee output-sensitive query times as the bounding box of a segment set might intersect a given geometric query object $\mathcal{Q}$ (as a circle) while none of the contained segments actually have an intersection with $\mathcal{Q}$.

In [4], the segment-circle intersection problem was studied, in which the roles of the input and the query are switched compared to our setting. Here, the input is a set of n circles and the query object is a segment. It was shown that the circles can be preprocessed into a data structure of size $\mathcal{O}(n \log^2 n)$ such that queries can be answered in $\mathcal{O}(n^{2/3} \log^2 n + k)$ where k denotes the output size. For disks, the query time reduces to $\mathcal{O}(\sqrt{n} \log^2 n + k)$.

There are also other contexts in which the connectivity structure of the input segments plays a crucial role. For example, in [3], the input is a set of segments and the goal is to report the connected components that intersect a given query segment. In [6], it was shown that intersections between two sets of line segments can be computed much faster if each of the two sets is connected.

1.2 Contribution

We propose a new data structure for circle-segment or disk-segment intersection queries on connected planar graphs with n edges, which can be constructed using $\mathcal{O}(n \log^3 n)$ time and space, and answers queries with output size k in time $\mathcal{O}(k \log^3 n)$. This is a significant improvement over the general case, where a data structure of (near-)linear size yields query times in $\mathcal{O}(n^{2/3+\varepsilon} + k)$ for $\varepsilon > 0$. Thus, especially for queries with small output size, our query time is vastly superior. If the input graph is non-planar and there are C crossings among the segments, the construction time and the space consumption increase to $\mathcal{O}((n + C) \log^3 n)$ but the query time stays the same. At the heart of our data structure lies a scheme which partitions the edges of the connected input graph, such that the resulting partitions have balanced size and each still form connected components. We show that such a partitioning can be computed in time $\mathcal{O}(n)$. Applying this result recursively allows us to efficiently construct

an edge partition tree[1] of logarithmic depth. We then equip each such edge partition with an oracle that decides whether for a given query circle $\mathcal{Q}$ there is some edge in the set that intersects $\mathcal{Q}$. Based on this oracle, the tree can be traversed efficiently on query time.

We also carefully describe how to implement the proposed data structure and demonstrate its usefulness in a proof-of-concept evaluation on different road networks. While road networks are non-planar, the number of crossings C between the edges is typically in $\Theta(\sqrt{n})$ [15] and thus we expect our data structure to perform well. This is confirmed in the experiments where we observe fast construction times even on large networks, and improved query times over the AABB tree implementation provided by CGAL.

2 Preliminaries

Throughout the paper, we assume to be given a connected geometric graph $G(V, E)$ as input. The graph nodes are points in $\mathbb{R}^2$ and each edge forms a straight-line segment between its endpoints. The number of pairwise crossings between edge segments is denoted by C. Here, an edge crosses another edge if their intersection contains a point that is not an endpoint of both edges. We assume that no pair of edges intersects in more than one point. Connectivity of G implies that there is a simple path between any pair of nodes in the graph, formed by sequence of edges from E. Thus, connectivity solely relies on the graph structure. This is in contrast to the notion of connected arrangement of segments, in which the set of points induced by the segments needs to be connected [3]. While in a connected geometric graph the edge segment arrangement is also connected, the reverse is not necessarily true (consider a set of segments which all pairwise cross). However, in a planar graph embedding, where we have $C = 0$, these two definitions coincide.

3 An Improved Data Structure for Connected Geometric Graphs

In this section, we introduce our new data structure that leverages the connectivity of the input graph to answer circle-edge and disk-edge intersection queries more efficiently.

In all approaches presented in [16] for the general case, the primary data structure is constructed by only considering the endpoints of the input segments. Either a spanning path with bounded stabbing number is computed on that point set or a partition tree is constructed on the (transformed) endpoints. Thus, any connectivity information is immediately lost. The original segments are only indexed in secondary or even tertiary data structures which are queried individually if the respective point sets fulfill certain requirements.

In contrast, our data structure construction is driven by the edges of G. In Section 3.1, we prove that the edges of a connected graph G can be partitioned into two subsets, such that each partition remains connected and the partition sizes are a constant fraction of the total number of edges. The respective partitioning can be computed in linear time. By applying this method recursively, we obtain an edge partition tree of logarithmic depth in which the leaves correspond to individual segments. In Section 3.2, we describe how to equip each node of the edge partition tree with an oracle that efficiently decides whether a given query circle has a type-(i) or type-(ii) intersection with some edge in the connected subgraph associated with the tree node. This allows to traverse the tree in DFS-order for query answering, only following paths for which the oracle answer is positive. More details on the query answering procedure are given in Section 3.3.

[1] We remark that despite the similar name our data structure differs substantially from the partition tree concept for point sets described in [19].

3.1 Balanced Connected Graph Partitioning

In the construction of our intersection search data structure, we follow the well-established paradigm of recursively dividing our problem into smaller subproblems until they reach a tractable size. To accomplish this while also being able to utilize the connectivity of the input graph, we require an algorithm that partitions the edge set of the input graph such that the connectivity property is maintained in each partition. Existing geometric partitioning methods (for example, median- or grid-based) are oblivious to the connectivity structure and thus are not guaranteed to produce the desired result. The method we propose does the opposite as it completely ignores all geometric information and instead operates only on the abstract graph. It is thus also applicable to non-geometric graphs. Edge partitioning has previously been studied in the context of distributed graph algorithms and parallelization [9, 22]. However, there the goal is to minimize communication cost introduced by the partitioning or to minimize node duplication (which turns out to be NP-hard). However, our focus is on balanced partitioning into connected components. In the remainder of this subsection, we prove the following central theorem.

▶ **Theorem 1.** *Given a connected graph $G[E]$ induced by the edge set E of size n, there is an $\mathcal{O}(n)$-time algorithm that partitions E into E_1 and E_2 such that:*
- $E = E_1 \uplus E_2$,
- $G[E_i]$ *is connected for* $i \in \{1, 2\}$,
- $|E_i| \leq \frac{2}{3}n$ *for* $i \in \{1, 2\}$.

We now describe the partitioning algorithm in detail and then prove its correctness to establish the theorem. The algorithm starts by constructing a spanning tree of G via a DFS run from an arbitrarily selected source node s. Let T_v denote the nodes of the subtree of the DFS-tree that is rooted at node v. The following observation sketches how we can use the notion of T_v to achieve our goal of connected edge partitioning.

▶ **Observation 2.** *Let $E(T_v)$ be the set of edges in E that are incident to nodes in T_v. The graphs induced by the edge partitions $E_1 := E(T_v)$ and $E_2 := E \setminus E_1$ are each connected.*

But we still need to take care of the size bound for each partition. Thus, we would like to efficiently compute the function $\phi(v) := |E(T_v)|$ that assigns to each node the number of edges incident to the nodes in T_v. Initially, we set $\phi(v) = 0$ for all nodes v. We then consider the nodes one after the other using a post-order traversal of the DFS-tree and update the ϕ-values of the current node v as well as its parent node $par(v)$ in the DFS-tree. We will maintain the invariant that at the moment that v is processed, all ϕ-values are less than $\frac{1}{3}n$. This condition is clearly fulfilled after the initialization phase.

Let us now consider the algorithm steps for processing a node v. As we use post-order traversal of the DFS-tree, we know that v is the last node to be processed in T_v. We first check whether $\phi(v) + deg(v) < \frac{1}{3}n$, where $deg(v)$ denotes the degree of v in $G[E]$. If that is the case, we add $deg(v)$ to $\phi(v)$ and then proceed to add the new $\phi(v)$ value to $\phi(u)$ where $u = par(v)$. If we still have $\phi(u) < \frac{1}{3}n$, we are done with v and proceed to the next node in the post-traversal order. Clearly, our invariant is maintained in this case. Otherwise, if $\phi(u) \geq \frac{1}{3}n$, we also know that $\phi(u) \leq \frac{2}{3}n$, as our invariant guaranteed that $\phi(u)$ was smaller than $\frac{1}{3}n$ prior to processing v and we only updated the parent of v as $\phi(v) < \frac{1}{3}n$ held as well. Accordingly, with $v_1, \ldots, v_s$ be the children of u that were already processed up to this point (and thus contributed to $\phi(u)$), we set $E_1 := \bigcup_{i=1}^{s} E(T_{v_i})$ and $E_2 = E \setminus E_1$. The resulting partitions are each connected, as E_1 contains all edges induced by T_u and additionally only edges with at least one endpoint in T_u, while E_2 contains all induced edges of $\{T_s \setminus T_u\} \cup \{u\}$ and additionally only edges with at least one endpoint in this node set.

If, however, $\phi(v) + deg(v) \geq \frac{1}{3}n$, we know that edges incident to nodes in T_v already suffice to form E_1. If additionally $\phi(v) + deg(v) \leq \frac{2}{3}n$, we simply return $E_1 = E(T_v)$ and $E_2 = E \setminus E_1$. If the degree of v is too large to include all of its incident edges in E_1, we only use $\frac{2}{3}n - \phi(v)$ many, giving priority to edges $\{w, v\}$ with $par(w) = v$. Note that there are less than $\frac{1}{3}n$ such prioritized edges, as otherwise $\phi(v)$ would have been equal to or larger than $\frac{1}{3}n$ already, which would contradict our invariant. Therefore, we are sure to include all edges induced by T_v and therefore, again, guarantee connectivity of the resulting partitions.

Thus, the described algorithm reliably produces edge partitions E_1 and E_2, whose induced graphs are connected. However, we did not take double counting of the edges into consideration so far. An edge $e = \{a, b\} \in E$ with $a, b \in T_v$ might contribute twice to $\phi(v)$, as both $deg(a)$ and $deg(b)$ are part of the sum. Therefore, the ϕ-values might overestimate the real number of edges in the respective subtree by a factor of up to 2. Thus, we can so far only guarantee a weaker size bound than the one promised in Theorem 1.

▶ **Lemma 3.** *The partitioning algorithm runs in $\mathcal{O}(n)$ and produces connected edge partitions E_1 and E_2 each of which has a size of at most $\frac{5}{6}n$.*

Proof. The DFS run takes $\mathcal{O}(n)$ time. In the post-order traversal, each node is processed at most once, and the processing step takes constant time if we do not return the result after this step, and at most linear time for the step after which the algorithm terminates. Therefore, the total processing time is in $\mathcal{O}(n)$. By construction, the returned partitions E_1 and E_2 are connected and the estimated size of E_1 is between $\frac{1}{3}n$ and $\frac{2}{3}n$. But as some edges might have contributed twice to this estimation, it follows that we actually have $\frac{1}{6}n \leq |E_1| \leq \frac{2}{3}n$. With $E_2 = E \setminus E_1$, it follows that $\frac{1}{3}n \leq |E_2| \leq \frac{5}{6}n$. ◀

We remark that this size bound is already sufficient for our goal of constructing a partition tree on the edge set with logarithmic depth, and the simplicity of the algorithm also makes it very suitable for practical implementation. However, we can ensure a better partition size balance within the same asymptotic running time as follows. To avoid double counting, we first refine the DFS-tree to a binary tree by introducing for all nodes u with more than two children a complete binary tree rooted at u with the children forming the tree leaves. By the property of complete binary trees the total number of nodes and edges in the DFS-tree is at most doubled by this step. Moreover, we compute a lowest-common-ancestor (LCA) data structure on the refined DFS-tree rooted at s. Such a data structure can be constructed on trees in linear time and it answers LCA queries for node pairs in $\mathcal{O}(1)$ [7]. We then use the processing algorithm described above with two modifications: When processing a dummy node introduced in the binary tree refinement step, we assume its degree to be zero. When processing a real node v, we still add $deg(v)$ to $\phi(v)$, but we additionally iterate over its neighbors w with $\{v, w\} \in E$ and decrement $\phi(LCA(v, w))$ by one if w comes later in the processing order than v. We refer to this algorithm as improved partitioning algorithm.

▶ **Lemma 4.** *The improved partitioning algorithm runs in $\mathcal{O}(n)$ and produces connected edge partitions E_1 and E_2 each of which has a size of at most $\frac{2}{3}n$.*

Proof. The binary refinement step and the LCA data structure construction both run in linear time. The processing of v now takes $\mathcal{O}(deg(v))$, as the LCA data structure needs to be queried for each neighbor. Summed over all nodes, the degrees add up to $2n$ and therefore the overall running time is still linear.

Next, we argue that after v was processed and the algorithm did not yet terminate, we have $\phi(v) = |E(T_v)|$. The proof uses induction over the nodes in post-order traversal. The first node to be processed is a leaf of the DFS-tree and thus $T_v = \{v\}$. If $deg(v) \leq \frac{1}{3}n$, we

set $\phi(v) = deg(v)$ and therefore we also have $\phi(v) = |E(T_v)|$. Let us now assume that the induction hypothesis applies to the first j nodes in the post-order traversal and consider node number $j+1$, which we refer to as u. For each child v of u in T_u, we know that v was processed prior to u and therefore $\phi(v) = |E(T_v)|$. If u has only a single child v, we have $\phi(u) = \phi(v) - |\{\{w,u\} \in E | w \in T_u\}|$ right before u is processed. This is true because v propagates its ϕ-value to its parent and for all $w \in T_u$ with $\{w,u\} \in E$ we have $LCA(w,u) = u$ an u comes later in the processing order than w. Now adding $deg(u)$ to $\phi(u)$, we get $\phi(u) = \phi(v) + |\{\{w,u\} \in E | w \notin T_u\}| = |E(T_u)|$. If u has two children v_1 and v_2, then for all edges $\{a,b\}$ with $a \in T_{v_1}$ and $b \in T_{v_2}$ we have $LCA(a,b) = u$ and thus $\phi(u)$ was decremented by 1 for each such edge. It follows that we have $\phi(u) = \phi(v_1) + \phi(v_2) - |\{\{w,u\} \in E | w \in T_u\}| - |\{a,b\} \in E | a \in T_{v_1} \wedge b \in T_{v_2}\}|$ right before u is processed. As the last term is exactly the number of edges that are counted twice, once in $\phi(v_1)$ and once in $\phi(v_2)$, we get again $\phi(u) = |E(T_u)|$ after adding $deg(u)$ to $\phi(u)$. Therefore, if we terminate while processing node u because $\phi(u) + deg(u) \geq \frac{1}{3}n$, we are now indeed guaranteed that E_1 contains the promised number of edges.

But we might also terminate after processing some node v as the ϕ-value of its parent node u exceeded the threshold after adding $\phi(v)$. We first observe that this can only happen if v is the last child of u to be processed. If it is the first child, then $\phi(u) \leq 0$ holds prior to the processing of v. The value is initialized with zero, but might even become negative it u is the LCA of the endpoints of an edge that connects nodes in the respective subtree. Therefore, after the processing of v, we have $\phi(u) \leq \phi(v)$. And as $\phi(v) < \frac{1}{3}n$ for the propagation step to happen, it follows $\phi(u) < \frac{1}{3}n$ as well. If v is the last child to be processed, we know that $\phi(u) = |E(T_u)| - \{\{u,w\} \in E | w \notin T_u\}$ by the same argumentation as above. Thus, there is no double counting occurring in this scenario and it is safe to construct E_1 as described in the original partitioning algorithm. We conclude that the improved partitioning algorithm takes linear time and guarantees $\frac{1}{3}n \leq |E_i| \leq \frac{2}{3}n$ for $i \in \{1,2\}$. ◀

This completes the proof of Theorem 1. By recursively applying the theorem to a given connected input graph G until we end up with single edges, we can derive in time $\mathcal{O}(n \log n)$ an edge partition tree of depth $\mathcal{O}(\log n)$, in which each node encodes a connected subgraph of G. Figure 3 illustrates this partitioning. Next, we discuss how to augment the resulting edge partitioning tree structure to cater for circle-segment queries.

3.2 Circle-Graph Intersection Oracles

Given a connected graph $G[E]$, our goal is to construct an efficient decision oracle that returns *true* if a given query circle $\mathcal{Q}$ has a type(i)- or a type(ii)- intersection with some edge in G and *false* otherwise.

For our oracle, we utilize the well known geometric transformation in which points $p = (x,y) \in \mathbb{R}^2$ are mapped to points $p' = (x,y,x^2+y^2) \in \mathbb{R}^3$ by a projection function $\psi(p) = p'$, and a circle $\mathcal{Q}$ with center $c = (a,b) \in \mathbb{R}^2$ and radius r is mapped to the plane $z = a(2x-a) + b(2y-b) + r^2$ in $\mathbb{R}^3$ by a function $\rho(\mathcal{Q}) = z$ such that p is inside $\mathcal{Q}$ iff $\psi(p)$ lies below $\rho(\mathcal{Q})$ and outside $\mathcal{Q}$ iff $\psi(p)$ lies above $\rho(\mathcal{Q})$. These transformations are also applied in [16]. Let know $\Psi(E) := \{\psi(p) | \{p,q\} \in E\}$ denote the set of segment endpoints in the graph after applying the transformation via ψ. Further let $CH(\Psi(E))$ denote the convex hull of the resulting three-dimensional point set. We now show how these structures can be utilized to detect intersections of type-(i).

▶ **Lemma 5.** *Iff for a given connected graph $G[E]$ and a circle $\mathcal{Q}$ the plane $\rho(\mathcal{Q})$ intersects $CH(\Psi(E))$ there exists at least one edge $e \in E$ which has a type-(i) intersection with $\mathcal{Q}$.*

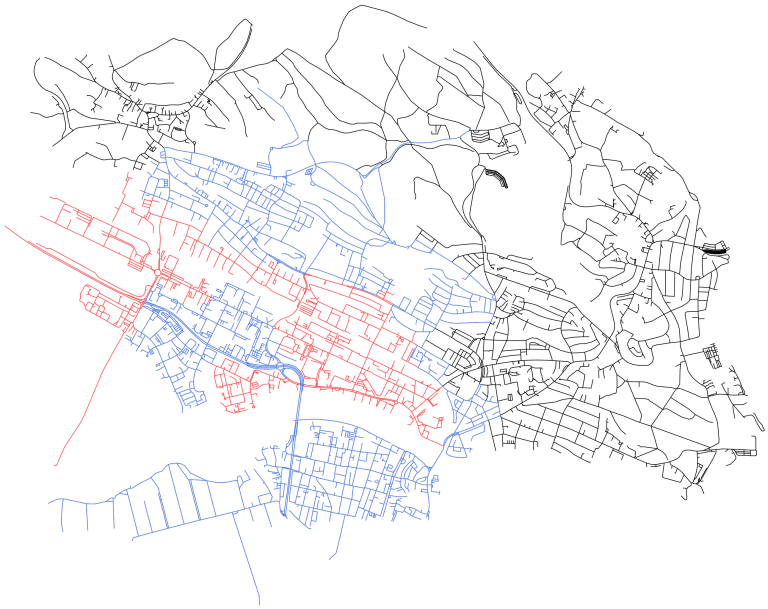

Figure 3 Partition of a graph of the road network of Konstanz with 13500 edges. The black partition represents the smaller of the two initial partitions. The blue and red partitions are the resulting connected subgraphs after subdividing the larger of the initial partitions once more.

Proof. If $\rho(\mathcal{Q})$ intersects $CH(\Psi(E))$, it follows that there is at least one point $p' \in \Psi(E)$ above $\rho(\mathcal{Q})$ as well as at least one point $q' \in \Psi(E)$ below $\rho(\mathcal{Q})$. By means of the applied geometric transformation, we know that $p = \psi^{-1}(p')$ lies outside of $\mathcal{Q}$ while $q = \psi^{-1}(q')$ lies inside of $\mathcal{Q}$. As $G[E]$ is connected, there needs to exist a path from p to q in $G[E]$, that is a path that starts outside the circle and ends in it. Accordingly, the path needs to contain at least one edge e that crosses the circle boundary. It follows that e and $\mathcal{Q}$ have a type-(i) intersection. Vice versa, if an edge that intersects $\mathcal{Q}$ exists, its endpoints are projected to different halfspaces with respect to $\rho(\mathcal{Q})$ and therefore it follows that $\rho(\mathcal{Q})$ intersects $CH(\Psi(E))$. ◀

Accordingly, the oracle returns *true* if $\rho(\mathcal{Q})$ intersects $CH(\Psi(E))$. Otherwise, if the convex hull $CH(\Psi(E))$ is either completely below or above $\rho(\mathcal{Q})$, we know that a type-(i) intersection can not occur but we still have to check for type-(ii) intersections. By definition of a type-(ii) intersection, the endpoints of the respective edge both need to be outside of the query circle. Thus, the oracle can safely return *false* if $CH(\Psi(E))$ is completely below $\rho(\mathcal{Q})$ as in this case all edge endpoints are inside the circle. So from now on, we only consider the case that $CH(\Psi(E))$ lies completely above $\rho(\mathcal{Q})$.

▶ **Lemma 6.** *Given a set of edge segments E with all segment endpoints being outside of a circle $\mathcal{Q} = (c, r)$, then there exists a type-(ii) intersection between a segment in E and $\mathcal{Q}$ iff there exists a segment $S \in E$ such that $d_2(c, S) \leq r$.*

Proof. Let $S^* \in E$ be the segment closest to the circle center c. If $d_2(c, S^*) > r$, we know that no segment in E penetrates the circle and thus there is no type-(ii) intersection. Otherwise, if $d_2(c, S^*) \leq r$ we know that S^* has a type-(ii) intersection with $\mathcal{Q}$ as both segment endpoints are outside the circle but the point on the segment closest to the circle center is inside the circle or on its boundary, which matches the definition of a type-(ii) intersection. ◀

Based on Lemma 5 and Lemma 6, we now have a full description of our decision oracle: Return *true* if $\rho(\mathcal{Q}) \cap CH(\Psi(E)) \neq \emptyset$, or, if $CH(\Psi(E))$ lies completely above $\rho(\mathcal{Q})$ and there is a segment within distance r from the circle center. Return *false* otherwise. The following theorem analyzes the construction and query time of this oracle.

▶ **Theorem 7.** *Given a connected geometric graph G with n edges and a query circle $\mathcal{Q}$, an oracle which decides in time $\mathcal{O}(\log^2 n)$ whether any graph edge intersects $\mathcal{Q}$ can be constructed in $\mathcal{O}((n+C)\log^2 n)$ time and space, where C is the number of edge crossings in G.*

Proof. For the first part of the oracle, we construct a Dobkin-Kirkpatrick hierarchy on the edge segment endpoints projected into $\mathbb{R}^3$. The Dobkin-Kirkpatrick hierarchy data structure can be computed in $\mathcal{O}(n \log^2 n)$ time and space on a set of n points and allows to answer extreme point queries in time $\mathcal{O}(\log n)$ [13, 14]. For the plane $\rho(\mathcal{Q})$, we issue two extreme point queries, one using the normal of the plane as optimization direction and one using the negated normal. The result will always be corner points of $CH(\Psi(E))$. If both points are below/above $\rho(\mathcal{Q})$, then $CH(\Psi(E))$ is fully below/above $\rho(\mathcal{Q})$. If one point is below and the other one above, we know that $\rho(\mathcal{Q}) \cap CH(\Psi(E)) \neq \emptyset$.

For the second part of the oracle, we need a data structure that reports the closest segment to a given query point (the circle center). We use a Segment Voronoi Diagram (SVD) for this purpose. It can be constructed in $\mathcal{O}((n+C)\log^2 n)$ time and space, where C is the number of edge crossings in G [17]. Nearest-segment queries can be answered in time $\mathcal{O}(\log^2 n)$. Thus, the second part of the oracle dominates the construction and query time. ◀

3.3 Query Answering

We are now ready to describe and analyze the full circle-segment data structure and the respective query answering algorithm.

Combining the insights from Theorem 1 and Theorem 7, we first construct an edge partition tree data structure, in which each node represents a connected subgraph of G, and then equip each such node with a decision oracle. To report all segment intersection with a query circle $\mathcal{Q}$, we first apply the transformation to the three-dimensional plane $\rho(\mathcal{Q})$. Then, we use the following recursive algorithm for query answering, with the initial call being to root node of our data structure: For the current data structure node, which represents some connected edge subset E', we ask the associated oracle whether there is any segment in E' that intersects $\mathcal{Q}$. If the answer is *false*, we stop. Otherwise, if the answer is *true*, we recursively call the algorithm on both children of the current node. We also stop if we reach a leaf node, which represents a single segment, and include this segment in the output set if it indeed intersects $\mathcal{Q}$. Figure 4 illustrates the query answering process on a small example graph.

▶ **Theorem 8.** *We can build a circle-segment data structure on the n edges of a connected geometric graph with C edge crossings in $\mathcal{O}((n+C)\log^3 n)$ time and space, which reports the k segments that intersect a given query circle in $\mathcal{O}(k \log^3 n)$ time.*

Proof. The construction time and the size of the data structure is dominated by the oracle computation for each subgraph represented in the edge partition tree. As the tree has logarithmic depth by virtue of Theorem 1 and the computation time is in $\mathcal{O}((n+C)\log^2 n)$ per level by virtue of Theorem 7, we get a total construction time in $\mathcal{O}((n+C)\log^3 n)$.

A query traverses the tree data structure in a DFS-like manner, aborting the search at nodes for which the oracle has concluded that there are no intersections to be detected in the respective subtree. Clearly, if the oracle answers *true* for any internal node of the

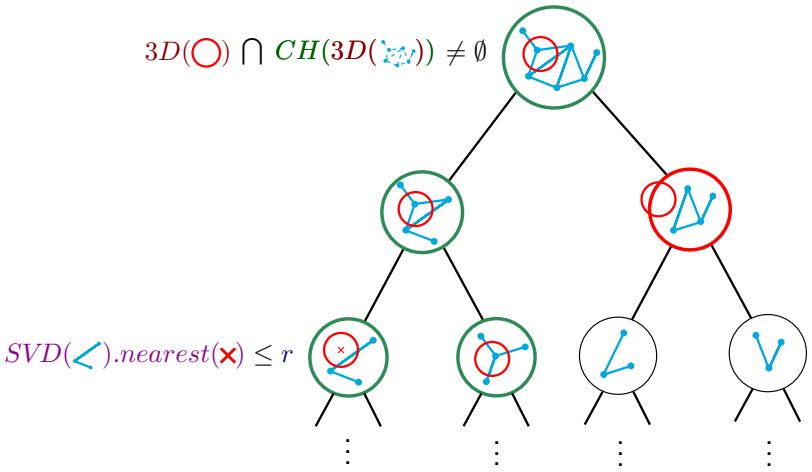

Figure 4 Circle-segment intersection data structure for a graph with 9 edges. The red circle is an exemplary query performed on the data structure. Depicted in green are nodes for which at least one of our intersection predicates (as described in Theorem 7) evaluate to true. For the red node on the right both of them return false as there are no intersecting segments.

tree, then at least for one if its children the answer needs to be *true* as well, as the segment inducing an intersection with the query circle has to be contained in either of the respective subgraphs. Therefore, the oracle calls in which the answer is *false* can always be charged to their respective siblings for which the answer is *true* (with the exception of the answer being *false* immediately on the root node). The number of times the oracle returns *true* is upper bounded by the number of nodes on a root-to-leaf path to segments that do intersect the query circle. As there are k such segments and a root-to-leaf path has a length in $\mathcal{O}(\log n)$, there are at most $\mathcal{O}(k \log n)$ such successful oracle calls. According to Theorem 7, each oracle call takes $\mathcal{O}(\log^2 n)$ time, and therefore we end up with a total query time in $\mathcal{O}(k \log^3 n)$. ◀

We remark that our data structure can also easily be used to answer disk queries with the same performance bounds. The only difference is that during query answering, whenever we encounter a node which represents a subgraph $G[E']$ for which $CH(\Psi(E'))$ lies completely below $\rho(\mathcal{Q})$, we add all edges in E' to the output set and still abort the search at this node.

4 Proof-Of-Concept Study

To demonstrate the usefulness of our circle-segment intersection data structure for practical application, we implemented the construction algorithm and the query answering routine in C++ and tested them on real-world road networks extracted from OSM. Experiments were conducted on an AMD Ryzen Threadripper 3970X with 32 cores.

4.1 Baseline

As baseline, we use the the AABB tree implementation provided by CGAL. It is a hierarchical data structure, in which each leaf node stores a geometric object and each internal node stores the bounding rectangle of all the objects in the respective subtree. It is constructed by dividing the input set of geometric objects roughly in half, computing the respective bounding rectangles, and then continuing recursively on both subsets until the leaf nodes are reached, which store the objects themselves. In our case, the leaves store the segments of the

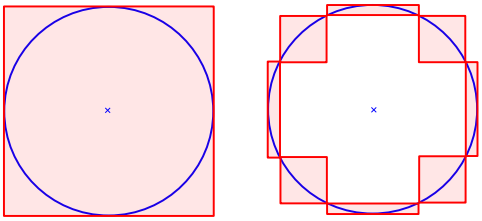

Figure 5 Query answering types using the AABB tree. Left: Simple bounding rectangle. Right: Better approximation of the circle by using eight rectangles.

input graph $G(V, E)$. In a query, given a geometric object, we check whether its bounding rectangle intersects the bounding rectangle associated with the root node of the AABB tree. If this is the case, both children are checked as well. If not, the search is aborted at the respective node. If a leaf node is reached and the final check confirms an intersection, the geometric object stored in said leaf node is added to the output. While often fast in practice, the theoretical running time can be huge in the case that the bounding rectangle of the query object intersects many bounding rectangles associated with the tree nodes, but there are no actual intersections between the query object and the elements stored in the tree leaves. This happens particularly often if the query object is not an area (as a disk) but rather an area boundary (as a circle). Here, any segment fully contained in the circle will create a bounding rectangle that intersects the one of the circle, although the segment itself does not intersect the circle. To improve the performance of our baseline, we therefore do not query the segment AABB tree with the bounding rectangle of the circle, but instead cover the circle with eight rectangles of much smaller total area, see Figure 5. We refer to this method as the *refined* AABB query.

4.2 Comparative Evaluation

As benchmarks, we used two large real-world road networks, namely a part of Germany (Figure 6) with roughly 1.1 million edges and the road network of Taiwan (Figure 8) with about 5 million edges. The Germany graph has $C = 2550$ edge crossings and the Taiwan graph $C = 22920$.

Constructing our circle-segment intersection data structure took less than 12 minutes on the Germany graph. The construction time is dominated by the segment Voronoi diagram computations for all subgraphs contained in the edge partition tree. In Figure 7, we compare the query times of our data structure to those of the refined AABB queries. We observe that except for very small query circle radii, our data structure is clearly superior. Even for large output sizes, the query times stay below 30 milliseconds and are at the peak about a factor of 8 faster than the respective refined AABB queries and two orders of magnitude faster than the naive AABB queries. Thus, we only consider the refined AABB queries in the upcoming experiments. We think that the mild increase of the query time with the output size besides the shown theoretical running time of $\mathcal{O}(k \log^3 n)$ can be explained by many paths from the root to a leaf that contain an intersecting segment share long common prefixes (while the theoretical worst-case analysis assumes them to be disjoint). Moreover, most of the occurring intersections are of type-(i). Our circle-graph oracle confirms such an intersection in time $\mathcal{O}(\log n)$ and then does not need to call the more expensive oracle for a type-(ii) intersection. Thus, in practice we have fewer and more efficient oracle calls than assumed in the theoretical analysis.

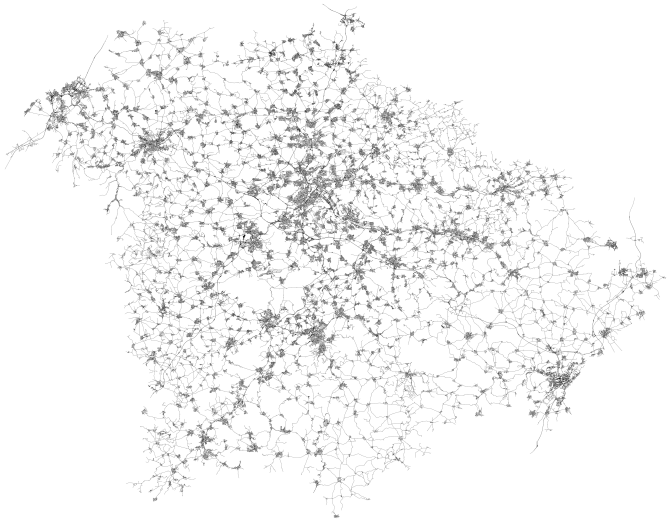

Figure 6 Road network of the metropolitan region around Stuttgart, Germany.

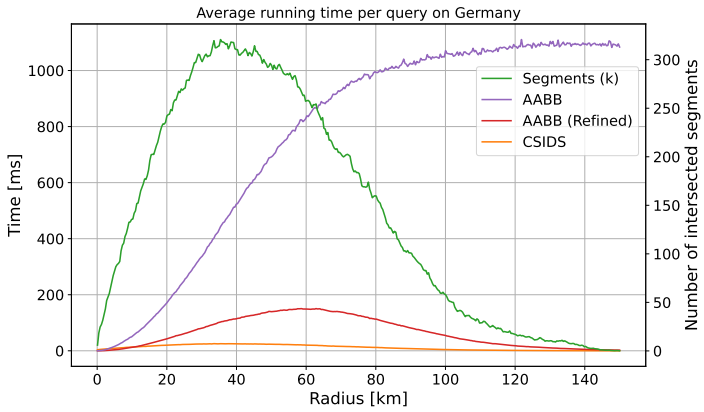

Figure 7 Query time comparison of our Circle-Segment Intersection Data Structure (CSIDS) and the AABB tree on the road network of Germany with approximately $1.1 \cdot 10^6$ edges. Query times are averaged over 100 queries per radius.

To deal with even larger graphs within a reasonable construction time, we observe that we can cut the top of our edge partition tree as well as the bottom levels. If we cut top levels, we then have to deal on query time with multiple root nodes, but we avoid the most expensive oracle construction steps on the large edge partitions close to the root. We remark that multiple root nodes can also be used in case the input graph is not connected but consists of (few) connected components. Cutting the bottom levels implies that we stop the recursive edge partitioning approach not only if we are down to single segments but already when the partition size drops below a certain threshold $t \in \mathbb{N}$. For the resulting leaf nodes, we do not construct the circle-graph intersection oracles but instead perform a simple intersection check for each contained segment with the query circle. As this limits the number of tree nodes, it helps to safe space. Furthermore, it is expected that the rather complex oracles we construct are only effective if the respective segment size is sufficiently large. We applied top and bottom cutting to the road network of Taiwan. This reduces the construction time

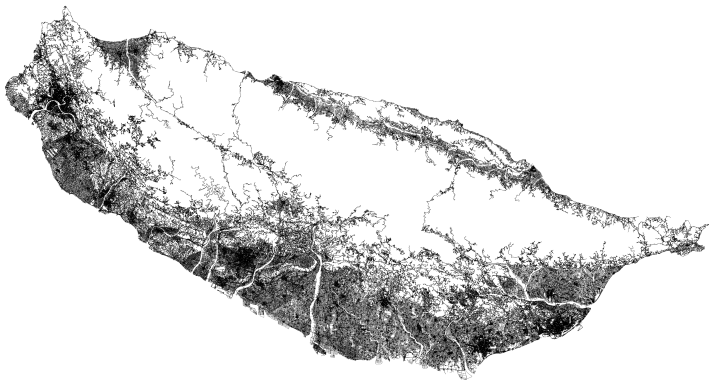

Figure 8 Road network of Taiwan.

significantly, from more than an hour to roughly 9 minutes. In Figure 9, we measure the impact of these data structure engineering techniques on the query time. Again, we observe that the refined AABB query times increase significantly with the query radius and the output size, reaching up to about half a second. Our query times, however, are again in the 20-40 millisecond range, although the search now starts from 76 root nodes (for most of which the oracle immediately returned a negative answer).

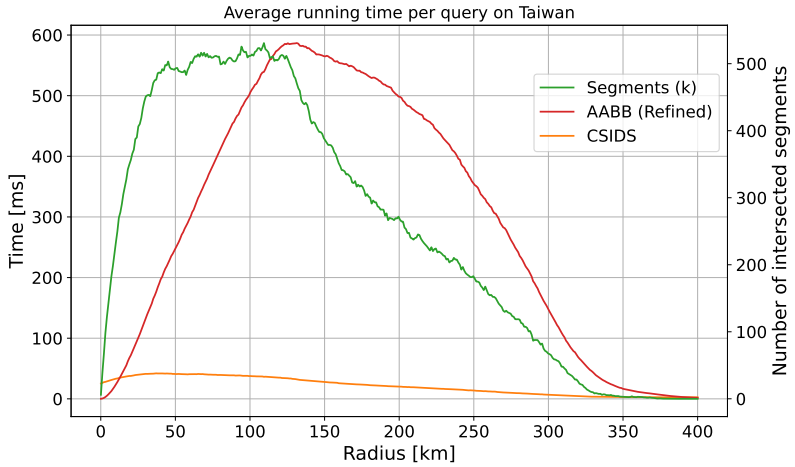

Figure 9 Query time comparison on the road network of Taiwan with approximately $5 \cdot 10^6$ edges. Our CSIDS has 76 top level query trees and the leave construction was stopped at 30 edges.

While it would of course also be interesting to conduct experiments on other types of graphs (for example, triangle meshes), our proof-of-concept study clearly shows the potential of our newly proposed data structure to accelerate the circle-segment queries on connected geometric graphs even for large edge sets.

5 Conclusions and Future Work

In this paper, we introduced a novel data structure for circle-segment intersection queries on connected geometric graphs. We proved a near-linear construction time in the size of the segment set and the number of crossings among the input segments, as well as output-sensitive query times. The data structure was also confirmed to perform well in practice. Nevertheless, there are many ways in which our data structure might be further improved:

- At the moment, we compute the circle-graph intersection oracle independently for each node in our edge partition tree. However, it might be much faster to compute the Dobkin-Kirkpatrick hierarchy and the segment Voronoi diagram bottom-up, using randomized incremental algorithms or clever merging strategies to derive the proper data structures for the level above from the already computed solutions from below. This might allow to decrease the construction time, both in theory and practice.
- Fractional cascading is often used to improve query times if the query involves repeated (binary) searches over similar or nested structures [10]. Recently, it was shown that for an input graph G with bounded degree and a set of hyperplanes in 3D associated with each node in G, one can decide for a query point and a connected subgraph H of G wether the point is below or above the lower envelope of the hyperplanes associated with the nodes in H much faster using fractional cascading [2]. Adapting this methodology to our use case, where we perform searches over nested convex hulls when we traverse the tree, could help to further improve our query times.

It would also be interesting to obtain lower bounds for data size and query time trade-offs in the connected graph setting to see how much improvement is still possible in theory. Moreover, integrating additional engineering ideas could also increase the scalability of the data structure and therefore make it useful for an even larger range of practical applications.

References

1 Ahmed Abdelkader and David M Mount. Approximate nearest-neighbor search for line segments. In *37th International Symposium on Computational Geometry*, 2021.

2 Peyman Afshani, Yakov Nekrich, and Frank Staals. Convexity helps iterated search in 3d. In *41st International Symposium on Computational Geometry (SoCG 2025)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.SoCG.2025.3.

3 Pankaj K. Agarwal and M van Kreveld. Connected component and simple polygon intersection searching. *Algorithmica*, 15(6):626–660, 1996. doi:10.1007/BF01940884.

4 Pankaj K Agarwal, Marc van Kreveld, and Mark Overmars. Intersection queries for curved objects. In *Proceedings of the seventh annual symposium on Computational geometry*, pages 41–50, 1991.

5 Alok Aggarwal, Mark Hansen, and Thomas Leighton. Solving query-retrieval problems by compacting voronoi diagrams. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 331–340, 1990.

6 Julien Basch, Leonidas J Guibas, and GD Ramkumar. Reporting red-blue intersections between two sets of connected line segments. In *Algorithms – ESA'96: Fourth Annual European Symposium Barcelona, Spain, September 25–27, 1996 Proceedings 4*, pages 302–319. Springer, 1996.

7 Michael A Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005. doi:10.1016/J.JALGOR.2005.08.001.

8 Yannick Bosch and Sabine Storandt. Continuous map matching to paths under travel time constraints. In *23rd International Symposium on Experimental Algorithms, SEA 2025, July 22-24, 2025, Venice, Italy*, volume 338 of *LIPIcs*, pages 7:1–7:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.SEA.2025.7.

9 Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1456–1465, 2014. doi:10.1145/2623330.2623660.

10 Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1):133–162, 1986. doi:10.1007/BF01840440.

11 Daniel Chen, Anne Driemel, Leonidas J Guibas, Andy Nguyen, and Carola Wenk. Approximate map matching with respect to the fréchet distance. In *2011 Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 75–83. SIAM, 2011. doi:10.1137/1.9781611972917.8.

12 Mark De Berg. *Computational geometry: algorithms and applications*. Springer Science & Business Media, 2000.

13 David P Dobkin and David G Kirkpatrick. Fast detection of polyhedral intersection. *Theoretical computer science*, 27(3):241–253, 1983. doi:10.1016/0304-3975(82)90120-7.

14 David P Dobkin and David G Kirkpatrick. Determining the separation of preprocessed polyhedra—a unified approach. In *International Colloquium on Automata, Languages, and Programming*, pages 400–413. Springer, 1990. doi:10.1007/BFB0032047.

15 David Eppstein and Michael T Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, pages 1–10, 2008.

16 Prosenjit Gupta, Ravi Janardan, and Michiel Smid. On intersection searching problems involving curved objects. In *Algorithm Theory – SWAT'94: 4th Scandinavian Workshop on Algorithm Theory Aarhus, Denmark, July 6–8, 1994 Proceedings 4*, pages 183–194. Springer, 1994. doi:10.1007/3-540-58218-5_17.

17 Menelaos I Karavelas. A robust and efficient implementation for the segment voronoi diagram. In *International symposium on Voronoi diagrams in science and engineering*, volume 2004, pages 51–62, 2004.

18 Hung Le and Cuong Than. Greedy spanners in euclidean spaces admit sublinear separators. *ACM Transactions on Algorithms*, 20(3):1–30, 2024.

19 Jiří Matoušek. Efficient partition trees. In *Proceedings of the seventh annual symposium on Computational geometry*, pages 1–9, 1991.

20 Harvey J Miller. Gis and geometric representation in facility location problems. *International Journal of Geographical Information Systems*, 10(7):791–816, 1996. doi:10.1080/02693799608902110.

21 A Frank Van der Stappen and Mark H Overmars. Motion planning amidst fat obstacles. In *Proceedings of the tenth annual symposium on Computational geometry*, pages 31–40, 1994.

22 Chenzi Zhang, Fan Wei, Qin Liu, Zhihao Gavin Tang, and Zhenguo Li. Graph edge partitioning via neighborhood heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 605–614, 2017. doi:10.1145/3097983.3098033.