Approximate Dynamic Nearest Neighbor Searching in a Polygonal Domain

Joost van der Laan
Department of Information and Computing Sciences, Utrecht University, The Netherlands

Frank Staals
Department of Information and Computing Sciences, Utrecht University, The Netherlands

Lorenzo Theunissen
Delft University of Technology, The Netherlands

Abstract

We present efficient data structures for approximate nearest neighbor searching and approximate 2-point shortest path queries in a two-dimensional polygonal domain P with n vertices. Our goal is to store a dynamic set of m point sites S in P so that we can efficiently find a site $s \in S$ closest to an arbitrary query point q. We will allow both insertions and deletions in the set of sites S. However, as even just computing the distance between an arbitrary pair of points $q, s \in P$ requires a substantial amount of space, we allow for approximating the distances. Given a parameter $\varepsilon > 0$, we build an $O(\frac{n}{\varepsilon} \log n)$ space data structure that can compute a $1+\varepsilon$-approximation of the distance between q and s in $O(\frac{1}{\varepsilon^2} \log n)$ time. Building on this, we then obtain an $O(\frac{n+m}{\varepsilon} \log n + \frac{m}{\varepsilon} \log m)$ space data structure that allows us to report a site $s \in S$ so that the distance between query point q and s is at most $(1+\varepsilon)$-times the distance between q and its true nearest neighbor in $O(\frac{1}{\varepsilon^2} \log n + \frac{1}{\varepsilon} \log n \log m + \frac{1}{\varepsilon} \log^2 m)$ time. Our data structure supports updates in $O(\frac{1}{\varepsilon^2} \log n + \frac{1}{\varepsilon} \log n \log m + \frac{1}{\varepsilon} \log^2 m)$ amortized time.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases dynamic data structure, nearest neighbor search, polygonal domain

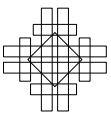
Digital Object Identifier 10.4230/LIPIcs.SoCG.2026.69

Related Version *Full Version*: `https://arxiv.org/abs/2603.11775` [20]

1 Introduction

Nearest neighbor searching is a fundamental problem in which the goal is to store a set S of m point sites so that given a query point q one can quickly report a site in S that is closest to q. For example, the set S may be a set of (locations of) shops, and we may wish to efficiently find the shop closest to our current location. In addition, nearest neighbor searching shows up as an important subroutine in several other fundamental problems. For example, in computing closest pairs [7] and matchings [15]. In these applications, there are often two (additional) difficulties: (i) the point sites are typically embedded in some constrained domain, and (ii) the set of sites may change over time, and hence it may be necessary to insert and delete sites. In the example problem above other buildings, roads, and lakes etc. act as obstacles, and thus we actually have to measure the distance $d(q, s)$ between two points q and s in terms of the length of a shortest obstacle avoiding path $\pi(q, s)$. Furthermore, shops may open and close (or may be too crowded or out of products etc.), and hence we want to consider only the currently open or available shops when querying.

In this paper, we consider dynamic nearest neighbor searching among a set S of m sites in a two-dimensional polygonal domain P with n vertices. We will allow both insertions and deletions in the set of sites S. Agarwal, Arge, and, Staals [1] presented a near linear space data structure for this problem that allows efficient (polylogarithmic) queries and updates, provided that P contains no holes; i.e. it is a simple polygon. However, extending these



results to arbitrary polygonal domains seems to be far out of reach. It seems we need to at least be able to efficiently compute the (geodesic) distance $d(q, s)$ between two arbitrary query points q and s in P. Currently, the best data structures that support such queries in polylogarithmic time use $O(n^9)$ space [11] (and it seems unlikely we can determine the site closest to q without actually evaluating geodesic distances). Hence, we settle on answering queries approximately. Let $\varepsilon > 0$ be some parameter, and let s^* be the site in S closest to a query point q, then our goal is to report a site $s \in S$ whose distance to q is at most $(1 + \varepsilon)d(q, s^*)$. We will say that s is *ε-close* to q. Our main result is then:

▶ **Theorem 1.** *Let P be a polygonal domain with n vertices, let S be a dynamic set of m point sites in P, and let $\varepsilon > 0$ be a parameter. There is an $O(\frac{n+m}{\varepsilon} \log n + \frac{m}{\varepsilon} \log m)$ space data structure that can*

- *compute a site in S that is ε-close to a query $q \in P$ in $O(\frac{1}{\varepsilon^2} \log n + \frac{1}{\varepsilon} \log n \log m + \frac{1}{\varepsilon} \log^2 m)$ time,*
- *support insertions of sites into S in $O(\frac{1}{\varepsilon^2} \log n + \frac{1}{\varepsilon} \log n \log m + \frac{1}{\varepsilon} \log^2 m)$ amortized time,*
- *and support deletions of sites from S in $O(\frac{1}{\varepsilon} \log n \log m + \frac{1}{\varepsilon} \log^2 m)$ amortized time.*

Constructing the initially empty data structure takes $O(\frac{n}{\varepsilon^2} \log^2 n)$ time.

The crucial ingredient in our data structure is a data structure for $(1 + \varepsilon)$-approximate 2-point shortest path queries. Thorup [19] presents an $O(\frac{n}{\varepsilon} \log n)$ space data structure that, given two arbitrary query points s, t in the polygonal domain P, can compute an estimate $\hat{d}(s, t)$ of their distance so that $d(s, t) \leq \hat{d}(s, t) \leq (1 + \varepsilon)d(s, t)$ in $O(\frac{1}{\varepsilon^3} + \frac{\log n}{\varepsilon \log \log n})$ time. Thorup's solution however assumes that distances are encoded as floating point numbers whose bits we can manipulate. This is inconsistent with the Real-RAM model of computation usually assumed in computational geometry. We stick to the Real-RAM model, and show that we can still compute approximate distances efficiently. We can actually improve the dependency on ε, and obtain:

▶ **Theorem 2.** *Let P be a polygonal domain with n vertices, and let $\varepsilon > 0$ be a parameter. In $O(\frac{n}{\varepsilon^2} \log^2 n \log \frac{n}{\varepsilon})$ time we can construct an $O(\frac{n}{\varepsilon} \log n)$ space data structure that, given query points $s, t \in P$ returns an $(1 + \varepsilon)$-approximation of $d(s, t)$ in $O(\frac{1}{\varepsilon^2} \log n)$ time.*

Our overall approach is actually the same as in Thorup's approach: we recursively construct a separator, and estimate distances by going via a discrete set of anchor points on this separator. However, our approach provides stronger guarantees on these anchor points. We believe this may be useful for other problems as well. Indeed, it allows us to efficiently answer (approximate) nearest neighbor queries as well. This also still requires several additional ideas, as simply evaluating the distance between a query point and every site would only give us query time linear in m.

Related Work. Dynamic nearest neighbor searching in the Euclidean plane has a long history, with early results by Agarwal and Matoušek [2]. Chan [6] was the first to achieve expected polylogarithmic update and query times. Following a sequence of improvements, the best structure at this time uses $O(m)$ space, allows for $O(\log^2 m)$ time queries, and for updates in deterministic, amortized $O(\log^4 m)$ time [15, 8]. For more general (constant complexity) distance functions, one can similarly achieve expected polylogarithmic query and update times [15, 16]. When the domain is a simple polygon P with n vertices, Oh and Ahn [17] presented a solution using $O(n + m)$ space achieving $O(\sqrt{m} \log(n + m))$ query time, and $O(\sqrt{m} \log m \log^2 n)$ update time. Agarwal, Arge, and, Staals [1] then showed how to achieve $O(\log^2 n \log^2 m)$ query time and expected $O(\text{polylog}(nm))$ update time using

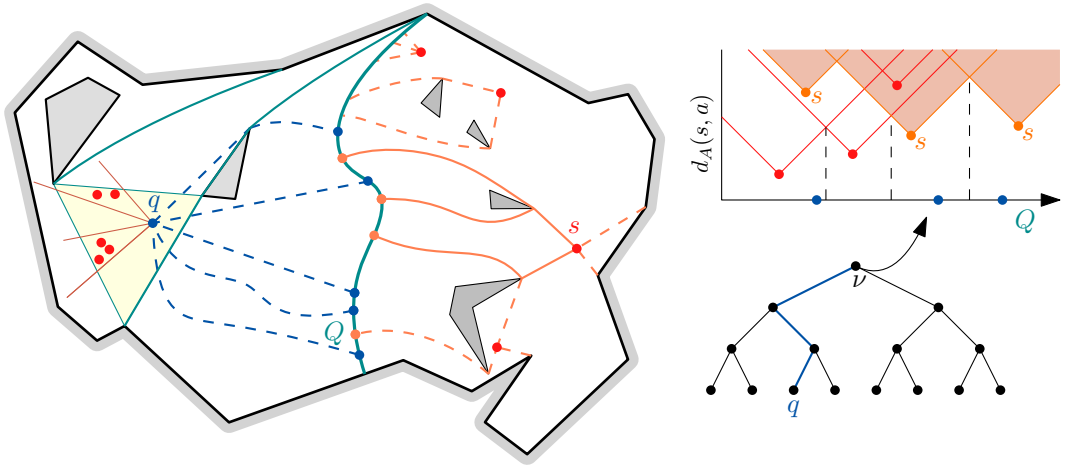


Figure 1 An overview of our approach. We use shortest paths to recursively subdivide P into subpolygons. If s^* lies in the same triangle as q, we can directly compute an ε-close site. Otherwise, the shortest path must cross a shortest path Q, and we find a site s that is sufficiently close to q whose path goes via an anchor point on Q.

$O(n + m \log^3 m \log n)$ space. When the domain may have holes and the set of sites is static, there is an $O(n + m)$ space data structure that allows querying the (exact) nearest neighbor in $O(\log(n + m))$ time [14]. Constructing the data structure uses $O(n \log n)$ time. Wang [21] recently showed that if P is triangulated, the shortest path map of a single site can be computed in $O(n + h \log h)$ time, where h is the number of holes in P. Conceivably, his algorithm could be extended to handle multiple sites (as is the case in the algorithm by Hershberger and Suri). We are not aware of any results that allow updating sites. Even the case where we allow only insertions is challenging: we cannot easily apply a static-to-dynamic transformation [3] since rebuilding the static structure may require $\Omega(n)$ time.

2 Global Approach

We first present an overview of our data structure. See also Figure 1. We start by briefly reviewing the ε-approximate 2-point shortest path query data structure by Thorup [19].

Computing approximate shortest paths. Thorup's approach recursively partitions the domain P using *separators* such that: each separator consists of at most three shortest paths, and the resulting subpolygons are of roughly the same size. This results in a balanced binary tree $\mathcal{T}$ of height $O(\log n)$ in which each node ν corresponds to some subpolygon P_ν that is further subdivided using a separator Q_ν.

Consider the separator Q of the root of this tree $\mathcal{T}$, and let $k = \lceil 1/\varepsilon \rceil$. The main idea is that each polygon vertex v generates a set $A_Q(v)$ of $O(k)$ so called *anchor points*[1] on Q, so that for any point $q \in Q$ the distance via one of these anchor points $A_Q(v)$ to v is at most $(1 + \varepsilon)$ times the true distance $d(q, v)$. So, if a shortest path $\pi(u, v)$ between two vertices u and v intersects Q then there is a path from u to v via two anchor points $a \in A_Q(u)$ and $a' \in A_Q(v)$ (and the subcurve $Q[a, a']$ of Q) of length $(1 + \varepsilon)d(u, v)$. Given the anchor points

[1] Thorup calls these points connections, but we use anchor points to avoid confusion with the paths connecting v to these points.

(and the distances to their defining vertices) we can find the length of such a path in $O(k)$ time. If the path $\pi(u, v)$ does not intersect Q, it is contained in one of the subpolygons of the children of the root, and hence we can compute a $(1 + \varepsilon)$-approximation of the distance $d(u, v)$ in the appropriate subpolygon. As each vertex defines $O(k)$ anchor points on Q, and appears in $O(\log n)$ levels of the recursion. The total space used is $O(nk \log n)$.

To compute the anchor points efficiently, Thorup's actual algorithm uses a graph G in which the distance $d_G(u, v)$ between u and v in the graph is already an $(1+\varepsilon_1)$-approximation of the true distance $d(u, v)$. Using the above approach with parameter ε_0 then actually gives a path in P with total length at most $(1 + \varepsilon_0)(1 + \varepsilon_1)d(u, v)$. Setting $\varepsilon_0 = \varepsilon_1 = \varepsilon/3$ then gives a ε-approximation of the distances as desired.

However, using the graph G results in slightly different (generally weaker) properties compared to the ideal case sketched above. In particular, his approach now guarantees that the distances from vertex v are within a factor $1 + \varepsilon$ of the true distance only for a discrete subset of points on Q. For querying the distances between pairs of vertices this is not an issue. Thorup's data structure can handle such queries in only $O(k)$ time. To support queries between arbitrary points $s, t \in P$, his algorithm connects s and t to $O(k)$ vertices in G each, and then queries the all these pairs. This leads to the query time $O(k^3 + k \log n/ \log \log n)$.

Reducing the query time. Ideally, we would treat s and t identical to polygon vertices, so that we get $O(k)$ anchor points on a separator Q. However, as such a candidate anchor point $p \in Q$ may lie anywhere on Q (in particular, it is not necessarily a vertex of G yet), we may not have the required ε_1-approximation of the distance to such a point yet.

We will show that we can use a slightly different graph G^Q that includes additional "Steiner" points on Q. This leads to a different set of anchor points compared to Thorup's approach, but *does* guarantee that for any vertex v, and *any* point q on Q, the distance to v is at most $(1 + \varepsilon)d(q, v)$. Furthermore, we show that we can efficiently compute an initial set of $O(k^2)$ candidate anchor points for an arbitrary point s, which we then reduce to $O(k)$. This results in $O(k^2 \log n)$ query time for an arbitrary pair of query points $s, t \in P$.

Answering nearest neighbor queries. As in the approximate 2-point shortest path data structure above, we construct a tree $\mathcal{T}$ of separators. We associate every site $s \in S$ with a root-to-leaf path in this tree. Namely, the nodes ν for which s is contained in the subpolygon P_ν. For each node ν on this path, we will additionally maintain a set of anchor points $A_Q(s)$ on the separator $Q = Q_\nu$ corresponding to this node. In particular, we consider Q as a one dimensional space, and maintain an additively weighted Voronoi diagram $Vor(A_Q)$ on Q. The set of sites $A_Q = \bigcup_{s \in S \cap P_\nu} A_Q(s)$ is the set of all anchor points of all sites that appear in P_ν. The weight of an anchor point $a \in A_Q(s)$ is the distance estimate from a to s.

If a shortest path between a query point q and its closest site $s^* \in S$ intersects Q, then there is a sufficiently short path via one of the anchor points of s^* and q on Q. We can then find a sufficiently close site s by querying the Voronoi diagram $Vor(A_Q)$ with all anchor points of q. This leads to an overall query time of $O(k^2 \log n + k \log n \log m + k \log^2 m)$, as we perform such queries in $O(\log n)$ nodes in $\mathcal{T}$.

The graph of the function that expresses the distance from a site $s \in S$ to a point p on Q via a given anchor point $a \in A_Q(s)$ is nicely "V-shaped". Hence, we can maintain $Vor(A_Q)$ by maintaining the lower envelope of these functions. This allows us to insert and remove sites in $O(k \log^2(km))$ time. This leads to an overall update time of $O(k^2 \log n + k \log n \log m + k \log^2 m)$. Our structure uses $O(nk \log n + mk \log n + mk \log m)$ space.

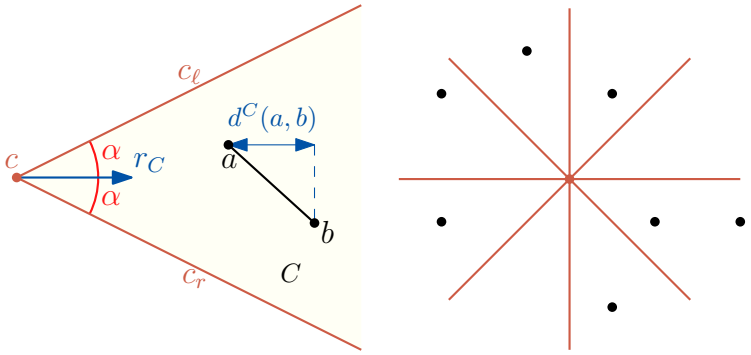


Figure 2 (a) A cone C with apex c, cone direction r_C, and angle α, and an illustration of the cone distance $d^C(a,b)$ between two points $a,b \in C$. (b) A family of cones with angle $2\alpha = 45$ degrees.

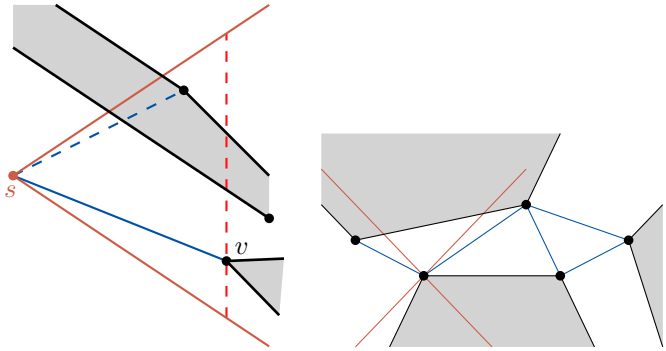


Figure 3 (a) A point s, which has an outgoing cone edge to a vertex v w.r.t the pictured cone C. Hence, there are no visible vertices in C to the left of the red dashed line. (c) An example cone graph G (for a very large ε).

Organization. In Section 3, we define our graph G^Q, and prove that it allows us to accurately estimate distances between any pair of points s,t whose shortest path intersects Q. We then use this to construct our approximate 2-point shortest path data structure in Section 4. Finally, in Section 5 we describe our lower envelope data structure and the remaining tools needed for an efficient dynamic data structure for approximate nearest neighbor searching. Omitted proofs are in the full version [20].

3 Estimating distances using a graph

In this section we discuss how to approximate distances within a polygon P. Our results are based on Clarkson's cone graph [10] on which we can efficiently compute approximate distances between vertices. We review this result in Section 3.1. In Section 3.2 we then extend this approach so that we can represent the distance from any vertex of P to any point on a given shortest path Q in P. This involves computing the anchor points for each vertex of P. Finally, in Section 3.3, we further augment the graph to also represent distances from a set S of arbitrary point sites to (and via) the shortest path Q, and show how to compute the anchor points for such an arbitrary point $s \in S$ efficiently.

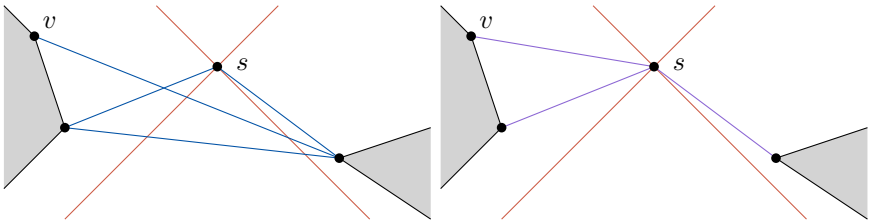


Figure 4 (a) The graph $G[s]$, where only outgoing cone edges from s have been added. (b) The cone graph on $V \cup \{s\}$. Observe that the edge set of neither graph is a subset of the other.

3.1 Clarkson's cone graph

Clarkson [10]'s graph is based on cones. A *cone* C is characterized by an apex point c, a cone direction r_C which is a unit length vector and an angle $\alpha \leq \pi/2$. The boundary of the cone consists of two half lines c_ℓ, c_r who both start at c and the difference in angle compared to r_C is $\alpha/2$ and $-\alpha/2$ respectively. We use C_c to denote a cone C to have its apex at point c. A *cone family* $\mathcal{F}$ is a set of cones that all have their apex at the origin, such that any point in $\mathbb{R}^2$ is contained in at least one cone $C \in \mathcal{F}$. See Figure 2(b) for an example.

Let $\varepsilon > 0$ be a parameter. Clarkson then defines a cone family $\mathcal{F}_\varepsilon$ consisting of $O(k) = O(1/\varepsilon)$ cones, each with angle at most $\varepsilon/8$, and a *cone graph* G based on $\mathcal{F}_\varepsilon$.

Let V be the set of vertices of P. A vertex $b \in V$ is a *minimal cone neighbor* of a point $a \in P$ if and only if there exists a cone $C \in \mathcal{F}_\varepsilon$, so that $b \in S$ is the vertex in the translated cone C_a with minimum *cone-distance* $d^C(a,b) = |r_C \cdot (b-a)|$ to a among all vertices that are visible from a (i.e. for which the line segment $\overline{ab}$ is contained in P). We will also say that a has an *outgoing cone edge* to b w.r.t the cone C. Let $N(a)$ denote the set of these minimal cone neighbors of a over all cones in $\mathcal{F}_\varepsilon$. See Figure 3(a) for an example.

The cone graph $G = (V, E)$ then has an edge $(u, v) \in E$ between two (polygon) vertices $u, v \in V$ if and only if u had v as its *minimal cone neighbor* or vice versa, i.e. $v \in N(u)$ or $u \in N(v)$. Each vertex u has at most $O(k)$ outgoing cone neighbors, and thus the total number of edges in G is at most $O(nk)$. Let $d_G(u, v)$ denote the length of a shortest path $\pi_G(u, v)$ between vertices u, v in G.

▶ **Lemma 3** (Clarkson [10]). *For every pair of vertices u, v in the cone graph G, there is a path $\pi_G(u, v)$ of length $d_G(u, v)$ so that*

$$d(u, v) \leq d_G(u, v) \leq (1 + \varepsilon)d(u, v).$$

In $O(nk \log n)$ time we can build a data structure of size $O(nk)$ so that for any point $s \in P$ one can compute $N(s)$ in $O(k \log n)$ time. Hence, G can be constructed in $O(nk \log n)$ time.

Extending the cone graph. Let $s \in P$ be a point, we define the *extended cone graph* $G[s] = (V \cup \{s\}, E \cup \{(s, v) \mid v \in N(s)\})$ by adding s as a vertex, and connecting it to all of its outgoing cone neighbors. Note that this graph is different from Clarkson's cone graph on $V \cup \{s\}$. See Figure 4 for an illustration. Using essentially same argument as in Lemma 3, we can still prove that $G[s]$ provides an ε-approximation of the distances to and from s.

3.2 Continuous ε-approximation graphs

Let Q be a shortest path between two vertices in P. Our goal is to define and compute, for each vertex v of P, a small set of weighted anchor points $A_Q(v)$ on Q, so that for any point q on Q there is an obstacle avoiding path between q and v of length at most $(1 + \varepsilon)d(q, v)$

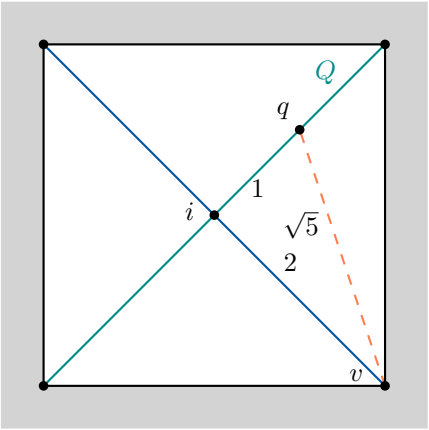


Figure 5 An example that shows that there may be points q on Q (shown in green) for which the graph G' (black, blue and green edges) considered by Thorup does not yield a ε-approximation. For any $\varepsilon \leq 1/10$, Thorup's anchor points give us distance estimate of 3, while the true distance is $\sqrt{5}$. Hence, this is only a $3/\sqrt{5} \geq 1.34$-approximation, not an $(1+\varepsilon)$-approximation.

via one of these anchor points. More specifically, let $\hat{d}_a$ be the weight of an anchor point a. We then want that there is an obstacle avoiding path $\hat{\pi}_a$ from v to a of length $\hat{d}_a$, and that $d_G(v, A_Q(v), q) := \min_{a \in A_Q(v)} d_G(a, q) + \hat{d}_a$ is at most $(1+\varepsilon)d(q, v)$.

Thorup [19] argues that such a set of $O(k)$ anchor points exists. Although his argument (Lemma 5 in his paper) is constructive, it is unclear how to efficiently construct such a set of anchor points. So, he constructs a set of anchor points with respect to Clarkson's cone graph instead. In particular, let $G = (V, E)$ be a cone graph constructed with parameter $\varepsilon_1 = \varepsilon/3$. Thorup defines the graph G' by subdividing every edge in G that intersects Q (at the intersection point), and connecting consecutive vertices along Q, and proves

▶ **Lemma 4** (Thorup [19]). *Let $H = (V, E)$ be an undirected weighted graph, let Q be a shortest path in H, and let $k' \in \mathbb{N}$ be a parameter. For each vertex $u \in H$, we can compute a set $A_Q(u) \subseteq V$ of $O(k')$ discrete anchor points on Q, so that for any vertex $v \in Q$ we have*

$$d_H(u, v) \leq d_H(u, A_Q(u), v) \leq (1 + 1/k')d_H(u, v).$$

Computing all sets of anchor points takes a total of $O\left(k' \log |Q| \left(|V| \log |V| + |E|\right)\right)$ time.

By choosing $1/k' = \varepsilon/3$ and applying Lemma 4 on the graph G' this then allows him to efficiently estimate the distances between the *vertices* of G'. In particular, if the shortest path in G' between two vertices $u, v \in G'$ intersects Q then one can find an $(1+\varepsilon)$-approximation of their distance $d(u, v)$ in P using one of the constructed anchor points in $O(k)$ time.

However, we want an $(1+\varepsilon)$-approximation for the distance between any vertex in G and *any point* on Q; i.e. not just the vertices of Q and the intersection points of Q with edges of G. Thorup's construction does not immediately provide that, see for example Figure 5.

A continuous graph G^Q. We will instead define a slightly different graph G^Q that includes additional "Steiner" points on Q. This *does* allow us to accurately estimate the distance to *any* point on Q, and thus compute appropriate anchor points. We actually construct the cone graph G with parameter $\varepsilon_1 = \varepsilon/9$, and then extend it into a *continuous graph* $G^Q = (V^Q, E^Q)$ (see below). We obtain G^Q from G and Q as follows:

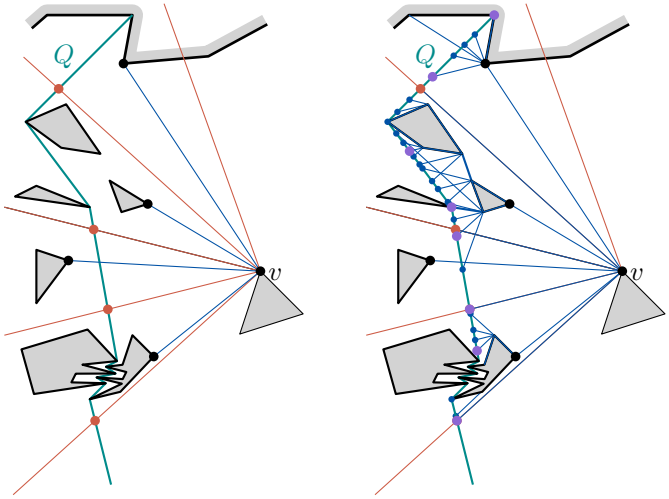


Figure 6 (a) The minimal cone neighbors $N(v)$ of vertex v, and the additional vertices $X_Q(v)$ (brown). (b) We compute a set of anchor points $A_Q(v)$ (purple) on Q based on G^Q (partially drawn).

- We include all vertices of G and of Q as vertices of G^Q.
- For each vertex $v \in G$ we consider the cones in the cone family centered at v. For each cone boundary ρ, we consider the first intersection point q of ρ with Q; if the resulting line segment $\overline{vq}$ lies inside P (i.e. the ray does not intersect the boundary of P before reaching q), we add q as a vertex, and (v, q) as an edge. Let $X_Q(v)$ denote the set of vertices added by v in this way. See Figure 6(a) for an illustration.
- We add all edges of G, and connect each pair of vertices that is consecutive along Q.

Observe that G^Q still has size $O(nk)$; as each vertex in G adds $O(k)$ additional vertices and edges, and Q also has at most $O(n)$ vertices. It is straightforward to construct G^Q from G in $O(nk \log n)$ time, using data structures for fixed-directional ray shooting queries [18].

▶ **Lemma 5.** *In $O(nk \log n)$ time we can construct a data structure of size $O(nk)$ so that for any point $s \in P$ we can compute the extra vertices $X_Q(s)$ in $O(k \log n)$ time. Hence, G^Q can be constructed in $O(nk \log n)$ time.*

Next, we argue that G^Q indeed allows us to ε_1-approximate the distances. We actually interpret G^Q as a *continuous graph* [5]. Every edge $(u, v) \in E^Q$ is a line segment $\overline{uv}$ in P, so we can extend the (graph) distance d_{G^Q} to points in the interior of edges. In particular, for points p, q on the same edge $\overline{uv}$ we define $d_{G^Q}(p, q) = \|p - q\|$ to simply be their Euclidean distance, and for $p \in \overline{uv} \in E^Q$ and $q \in \overline{wz} \in E^Q$ in the interior of different edges we have $d_{G^Q}(p, q) = \min_{u' \in \{u,v\}} \min_{w' \in \{w,z\}} \|p - u'\| + d_{G^Q}(u', w') + \|w' - q\|$. We are only interested in points in the interior of edges of Q. Hence, we prove:

▶ **Lemma 6.** *Let $u, v \in V$ be two polygon vertices in G^Q. Then we have that*

$$d(u, v) \leq d_{G^Q}(u, v) \leq (1 + \varepsilon_1) d(u, v).$$

Furthermore, for any point $q \in Q$ and any vertex $v \in G^Q$ we have that

$$d(v, q) \leq d_{G^Q}(v, q) \leq (1 + \varepsilon_1) d(v, q).$$

Computing and storing anchor points. We can now compute the distances in G^Q, and use them to construct an appropriate set of anchor points. See Figure 6(b). Due to Lemma 6 we can now actually just reuse Thorup's approach (i.e. Lemma 4) again:

▶ **Lemma 7.** *For every vertex v of P, we can compute a set $A_Q(v)$ of $O(k)$ anchor points on Q such that for any point q on Q we have $d(v,q) \leq d_{G^Q}(v, A_Q(v), q) \leq (1+\varepsilon)d(v,q)$.*

This takes $O(nk^2 \log n \log(kn))$ time in total.

3.3 The augmented cone graph

Let S be a dynamic set of point sites inside P. As we are interested in distances to (and between) the points in S, we also want to define and compute anchor points on Q for the points in S. The most straightforward approach would be to just build the above continuous graph G^Q on the set of points (vertices) $V \cup S$ rather than just the polygon vertices V. However, as computing the anchor points involves computing shortest paths in this graph, we cannot easily compute the anchor points for a single new site s. Similarly, maintaining (the distances in) the graph as S changes will be difficult. So, we argue that we can compute suitable anchor points for these new sites using a slightly different method instead.

Let $s \in P$ be a new point site. Similar to Section 3.1, we now consider the (continuous) graph $G^Q[s]$ that we obtain from G^Q by introducing $\{s\} \cup X_Q(s)$ as additional vertices, connecting s to every minimal cone neighbor in $N(s)$ and to every point in $X_Q(s)$, and subdividing the edges of Q to account for the additional vertices in $X_Q(s)$. This (continuous) graph still has $O(nk)$ vertices and $O(nk)$ edges. Moreover, we can use this graph to accurately estimate distances from s to any point on Q.

▶ **Lemma 8.** *Let s be a point in P. For any vertex w of P there is a path from s to w in $G^Q[s]$ of length $d(s,w) \leq d_{G^Q[s]}(s,w) \leq (1+\varepsilon_1)d(s,w)$ that passes through a cone neighbor $v \in N(s)$. Furthermore, for any point $q \in Q$ we have*

$$d(s,q) \leq d_{G^Q[s]}(s,q) \leq (1+\varepsilon_1)d(s,q).$$

Defining anchor points. Our goal is to define a set $A_Q(s)$ of anchor points on Q for point s. To this end, we first define a set of candidate points $X'_Q(s) = X_Q(s) \cup \bigcup_{v \in N(s)} A_Q(v)$ on Q. We can prove that if we assign appropriate weights to this set of $O(k^2)$ points it is indeed a set of anchor points. However, as we argue next, a subset of $O(k)$ of these points suffices.

Let $H = (V', E')$ be an undirected weighted graph with vertex set $V' = \{s\} \cup X'_Q(s)$. Vertex s is connected to each vertex $p \in X_Q(s)$ with an edge of length $\|s-p\| = d_{G^Q[s]}(s,p)$, and to $p \in A_Q(v)$ by an edge of length $\|s-v\| + d_{G^Q}(v,p)$. We connect vertices $u, w \in X'_Q(s)$ that appear consecutively on Q by an edge of length $d(u,w)$ (which is the length of the subpath along Q). Observe that the vertices of $X'_Q(s)$ form a shortest path in the graph H.

▶ **Lemma 9.** *Let $\varepsilon_2 > 0$ be a parameter such that $O(1/\varepsilon_2) = O(k)$. In $O(k^2)$ time we can compute a set $A_Q(s)$ of $O(k)$ discrete anchor points for H, i.e. so that for any $p \in X'_Q(s)$*

$$d_H(s,p) \leq d_H(s, A_Q(s), p) \leq (1+\varepsilon_2)d_H(s,p).$$

We use Lemma 9 with $\varepsilon_2 := \varepsilon/9$ to compute a set $A_Q(s)$ of $O(k)$ anchor points for s with respect to H. Hence, for each point $p \in X'_Q(s)$ (so in particular for those in a set $A_Q(v)$) there is a path of length $(1+\varepsilon_2)d_H(s,p) \leq (1+\varepsilon_2)(\|s-v\| + d_{G^Q}(v,p))$ via an anchor point in $A_Q(s)$. Next, we argue that this set is a valid set of anchor points for any point on Q.

▶ **Lemma 10.** *Let s be a point in P. In $O(k \log n + k^2)$ time, we can compute a set $A_Q(s)$ of $O(k)$ anchor points, so that for any point $q \in Q$ we have*

$$d(s,q) \leq d_{G^Q[s]}(s, A_Q(s), q) \leq (1+\varepsilon)d(s,q).$$

Estimating distances via the shortest path Q. Given a shortest path Q, and points $s, t \in P$ whose shortest path $\pi(s,t)$ in P intersects Q, we can use the above tools to accurately estimate the distance between s and t. More specifically, we compute G^Q and the anchor points as defined above. Additionally, we fix an endpoint u of Q and compute and store, for each vertex w of Q the distance to u (i.e. the path length along Q). This can easily be done in $O(n)$ additional time. Given a pair of points a, b on Q (and the edges that contain a and b, respectively) we can then compute their distance $d(a,b)$ (i.e. the length of the subpath $Q[a,b]$) in constant time.

Let $a \in A_Q(s)$ and $b \in A_Q(t)$ be anchor points of s and t on Q, respectively. The subpath $\hat{\pi}_a$ from s to a in $G^Q[s]$ corresponds to an obstacle avoiding path in P. Similarly, the subpath $\hat{\pi}_b$ from b to t is obstacle avoiding. Furthermore, since Q is a shortest (obstacle-avoiding) path in P, so is its subpath $Q[a,b]$ from a to b. So, the concatenation of $\hat{\pi}_a$, $Q[a,b]$, and $\hat{\pi}_b$ connects s and t and is obstacle avoiding, and has length $\hat{d}_a + \hat{d}_b + d(a,b)$. Now define

$$\hat{d}_Q(s,t) = \min_{a \in A_Q(s), b \in A_Q(t)} \hat{d}_a + \hat{d}_b + d(a,b)$$

to be the length of a shortest such obstacle avoiding path $\hat{\pi}_Q(s,t)$. We then have:

▶ **Lemma 11.** *Let P be a polygonal domain with n vertices, and let Q be a shortest path in P. In $O(nk^2 \log n \log(kn))$ time, we can construct a data structure on P and Q of size $O(nk)$ that can answer the following queries in $O(k \log n + k^2)$ time: given any pair of points $s, t \in P$ for which a shortest path $\pi(s,t)$ intersects Q, return the length $\hat{d}_Q(s,t)$ of an obstacle avoiding path $\hat{\pi}_Q(s,t)$ so that $d(s,t) \leq \hat{d}_Q(s,t) \leq (1+\varepsilon)d(s,t)$.*

4 A data structure for answering distance queries

In this section we develop an $O(nk \log n)$ space data structure that stores the polygonal domain P, so that given a pair of query points s, t we can compute an $(1+\varepsilon)$-approximation of the distance $d(s,t)$ in $O(k^2 \log n)$ time. Our data structure is essentially a balanced hierarchical subdivision of the polygonal domain combined with the graph-based approach to estimate distances from the previous section.

A balanced hierarchical subdivision for polygonal domains. A balanced hierarchical subdivision [9] is a recursive subdivision of a simple polygon into subpolygons. It is a crucial ingredient to obtain efficient solutions to many problems involving simple polygons [1, 13]. As was also recently observed [12], Thorup's approach [19] essentially provides a balanced hierarchical subdivision for polygonal domains. The following lemma summarizes this result:

▶ **Lemma 12** ([19, 12])**.** *Let P be a polygonal domain with n vertices. In $O(n \log n)$ time, one can compute a tree $\mathcal{T}$ so that*

1. *the leaves of $\mathcal{T}$ correspond to triangles in a triangulation of P,*
2. *each node ν of T corresponds to a subpolygon P_ν,*
3. *each internal node ν also corresponds to a set $\mathcal{Q}_\nu$ of shortest paths so that:*
   a. *$\mathcal{Q}_\nu$ consists of at most three shortest paths (see Figure 7),*
   b. *every shortest path $Q \in \mathcal{Q}_\nu$ is also a shortest path in P_ν, and*
   c. *the paths in $\mathcal{Q}_\nu$ partition P_ν into P_μ and P_ω, corresponding to the children of ν,*

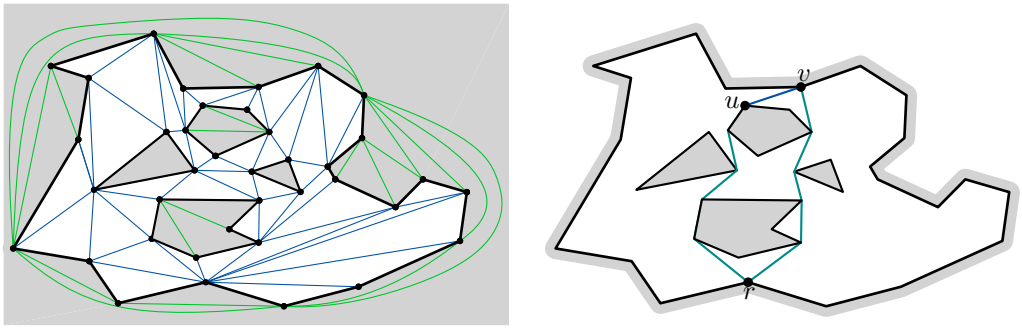


Figure 7 One step in the construction of the balanced hierarchical separator.

4. *the height of the tree is* $O(\log n)$*, and*
5. *the total complexity of the subpolygons* P_ν *on each level of* $\mathcal{T}$ *is* $O(n)$*, and thus the total complexity of* $\mathcal{T}$ *is* $O(n \log n)$*.*

The data structure. Our data structure now stores the balanced hierarchical subdivision $\mathcal{T}$ from Lemma 12. We build a point location data structure [18] on the triangulation induced by the leaves of $\mathcal{T}$. Additionally, for each edge e of this triangulation, let ν be the highest node in $\mathcal{T}$ for which the edge appears on a shortest path $Q \in \mathcal{Q}_\nu$ (if such a node exists). We store a pointer from edge e to this shortest path. We can easily compute these pointers during the construction of $\mathcal{T}$.

Consider a node ν. For each of the $O(1)$ shortest paths $Q \in \mathcal{Q}_\nu$ we will construct the data structure from Lemma 11 (i.e. the graph G^Q and the sets of anchor points) with respect to the subpolygon P_ν. Since the total size of all subpolygons P_ν on each level of the tree is $O(n)$, the total size of all of these data structures on a given level of $\mathcal{T}$ is $O(kn)$, and thus $O(nk \log n)$ in total. Similarly, the total preprocessing time is $O(nk^2 \log^2 n \log(kn))$.

Query. Given a pair of query points s, t we find the leaf triangles Δ_s and Δ_t that contain points s and t, respectively. If s and t lie in the same triangle, we can trivially report the length $\overline{st}$ as their shortest path. Otherwise, a shortest path from s to t must intersect one of the shortest paths $Q \in \mathcal{Q}_\nu$ for a common ancestor ν of the two leaves. In particular, let μ be the highest node of $\mathcal{T}$ for which $\pi(s,t)$ intersects a shortest path $Q \in \mathcal{Q}_\mu$. It follows that the shortest path $\pi(s,t)$ lies inside P_μ, and thus we can use the Lemma 11 data structure for Q to obtain a path of length at most $(1+\varepsilon)d(s,t)$. Unfortunately, we cannot easily find this node μ, so we query all Lemma 11 data structures for all shortest paths $Q \in \mathcal{Q}_\nu$, for all common ancestors ν of the leaves representing Δ_s and Δ_t, and report the shortest distance found. If either s or t lies on an edge e of the boundary of some triangle that is part of some shortest path $Q \in Q_\nu$, we use the pointer stored at e to jump to the highest such node, and query only the common ancestors of these nodes instead. Since every query returns the length of an obstacle-avoiding path, the returned estimate $\hat{d}(s,t)$ satisfies $d(s,t) \leq \hat{d}(s,t) \leq (1+\varepsilon)d(s,t)$ as desired. Since each query takes $O(k \log n + k^2)$ time, the total query time is $O(k \log^2 n + k^2 \log n)$.

Improving the query time. The $O(k \log^2 n)$ term in the query time is only due to computing the ray shooting queries needed to compute the sets $X_Q(s)$ for each shortest path Q considered. Next, we argue that we can compute these sets in only $O(k \log n)$ time by building an additional $O(nk \log n)$ space data structure. The overall space usage thus remains the same, and the final query time becomes $O(k^2 \log n)$ as claimed.

▶ **Lemma 13.** *In $O(nk \log n)$ time, we can build an $O(nk \log n)$ space data structure that given a query point s, allows us to compute the sets $X_Q(s)$ for all shortest paths $Q \in \bigcup_{\nu \in \mathcal{T}(s)} \mathcal{Q}_\nu$, where $\mathcal{T}(s)$ is the path in $\mathcal{T}$ from the root to the leaf triangle containing s.*

Finally, we plug in that $k = \lceil \frac{1}{\varepsilon} \rceil$, which establishes Theorem 2.

▶ Remark 14. Thorup's final approach [19] uses one additional idea: that one can guarantee that the boundary of a subgraph G_ν (used during the construction in Lemma 12) intersects only $O(1)$ shortest paths from ancestor separator paths. Therefore, he can also store anchor points on these paths, and query only the shortest paths of the lowest common ancestor of the leaves corresponding to s and t. It is not clear to us if this idea is still applicable with our approach. For a shortest path B on the boundary of some subpolyon P_ν, it is not clear in which (sub)polygon to place the path B so that we can invoke Lemma 11.

5 Dynamic nearest neighbor searching

In this section, we develop our data structure for approximate nearest neighbor searching with a dynamic set of sites S in a polygonal domain P. In Section 5.1 we first develop a simple solution for when S is simply a set of sites in the plane. In Section 5.2 we consider how to solve a restricted case in which we are given a shortest path Q in P, and we have to only answer queries q for which the shortest path between q and any site in S intersects Q. Then finally, in Section 5.3 we combine these results with the balanced hierarchical subdivision to obtain our data structure for the general problem.

5.1 Dynamic Euclidean ε-close neighbor searching

Let S be a dynamic set of m points in $\mathbb{R}^2$. We present a simple $O(km \log m)$ space data structure that given a query point q can report a site s for which the distance $d(q, s)$ is at most $(1+\varepsilon)d(q, s^*)$ in $O(k \log m)$ time. Here, s^* is a site in S closest to q. Our data structure supports insertions and deletions in $O(k \log m)$ time as well. Our data structure essentially uses the cone based approach from Section 3.1; i.e. for each cone C in the cone family $\mathcal{F}_\varepsilon$, we build a dynamic data structure that can report the site $s \in S \cap C_q$ with minimum cone distance. By the argument from Lemma 3 it follows that this then yields an ε-close site.

▶ **Lemma 15.** *Let S be a set of m sites in $\mathbb{R}^2$. There is an $O(km \log m)$ space data structure that, given a query point $q \in \mathbb{R}^2$ can report an $1/k$-close site from S in $O(k \log^2 m)$ time. Insertions and deletions of sites are supported in amortized $O(k \log^2 m)$ time.*

5.2 Maintaining an additively weighted Voronoi diagram

Let S be the set of m sites in the polygonal domain P, and let Q be a shortest path. We describe a dynamic data structure that, given a query point $q \in P$ can report an ε-close site $s \in S$, provided that the shortest path from q to the closest site $s^* \in S$ intersects Q. More specifically, our goal is to compute a site s which achieves the following minimum

$$\min_{s \in S} \hat{d}_Q(q, s) = \min_{s \in S} \min_{a \in A_Q(q), b \in A_Q(s)} \hat{d}_a + \hat{d}_b + d(a, b). \tag{1}$$

If the shortest path between s^* and q intersects Q, then by Lemma 11 this value is indeed at most $(1+\varepsilon)d(s^*, q)$, and thus the site s that achieves this minimum is ε-close to q. We rewrite Equation 1 to $\min_{a \in A_Q(q)} L_Q(a)$, where we define $L_Q(a) := \hat{d}_a + \min_{s \in S, b \in A_Q(s)} \hat{d}_b + d(a, b)$. Finding the optimal b for the evaluation of $L_Q(a)$ is essentially the closest site problem

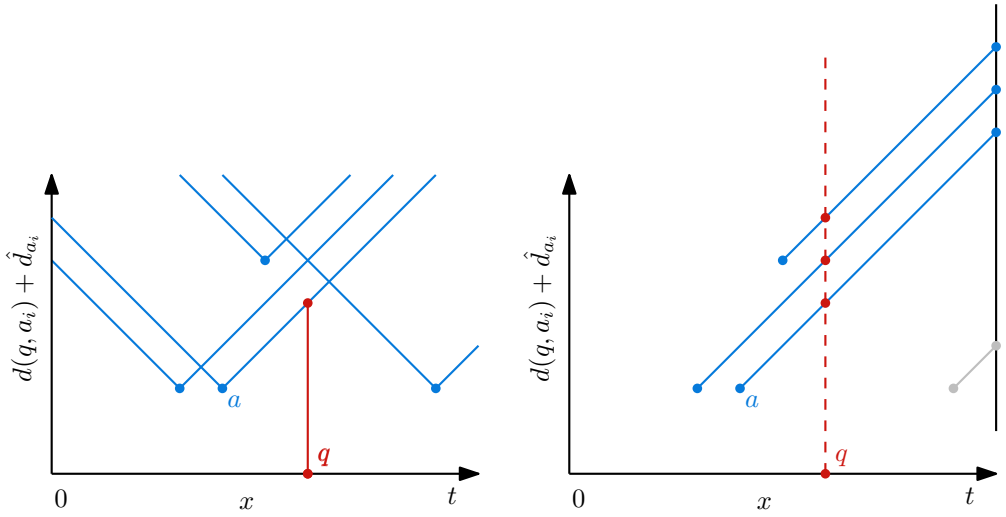


Figure 8 (a) For a query q we want to find the lowest intersection of the line at $q = x$ with the graph of the function $d(a, \cdot)$ of an anchor point $a \in A_Q(s)$. (b) The right half-lines of our V-functions. Note that the half lines intersect the vertical line at q in the same order as at t. Our data structure would find the site to the left of $q = x$ which makes the lowest intersection with $x = t$.

with additive weights $\hat{d}_b$. Furthermore, we only have to consider a 1-dimensional closest site problem as all anchor points lie on a shortest path $Q = \pi(u, v)$. This means we can map the coordinates of each point $a \in Q$ to the one-dimensional coordinate $a_x = d(u, a)$, this preserves distances as for all $a, b \in \pi(u, v)$ we have $d(a, b) = |d(u, b) - d(u, a)| = d(a_x, b_x)$.

Since these functions a have nice structure (see Figure 8), we can relatively easily maintain and query L_Q using priority search trees [4]. We store all $O(mk)$ anchor points (constructed using Lemma 11) and obtain:

▶ **Lemma 16.** *Let P be a polygonal domain with n vertices, let Q be a shortest path in P, and let S be a set of m sites in P. There is an $O(nk + mk)$ space data structure such that:*

- *For any point $q \in P$ we can find $\min_{s \in S} \hat{d}_Q(q, s)$ and the site which achieves this minimum in $O(k \log n + k^2 + k \log(mk))$ time.*
- *We can insert sites in amortized $O(k \log n + k^2 + k \log(mk))$ time, and delete sites in amortized $O(k \log(mk))$ time.*

Constructing an initially empty data structure takes $O(nk^2 \log n \log(kn))$ time.

5.3 The final data structure

Our data structure for general approximate nearest neighbor queries uses the same approach as in Section 4. Our data structure consists of the balanced hierarchical separator $\mathcal{T}$ from Lemma 12, in which we store associated structures at each node. We again preprocess the triangulation formed by the leaves for efficient point location, and store a pointer from each edge e to the highest shortest path Q in some $\mathcal{Q}_\nu$ that uses edge e.

For each internal node ν, and for each shortest path $Q \in \mathcal{Q}_\nu$, we now build an associated Lemma 16 data structure on Q. This structure will store a subset $S_{\nu,Q}$ of the sites. In particular, the sites in $S \cap P_\nu$ that did not lie on shortest paths of ancestors of ν.

For each leaf ν of $\mathcal{T}$, which represents some triangle P_ν, we store (a subset S_ν of) the sites in $S \cap P_\nu$ in the data structure from Lemma 15. We again store only the sites that did not already appear on shortest paths stored at ancestors of ν.

Finally, we build the data structure from Lemma 13 that allows us to efficiently compute the sets $X_Q(s)$ on any root-to-leaf path.

Space and construction time. Since the total complexity of all polygons associated with $\mathcal{T}$ is $O(n \log n)$, it readily follows that the total space of all Lemma 16 data structures is $O(nk \log n + mk \log n)$. As each site also appears in at most one leaf, it then follows that we use a total of $O(nk \log n + mk \log n + mk \log m)$ space. Similarly, constructing an empty data structure takes $O(nk^2 \log^2 n \log(kn))$ time.

Answering a query. Our goal is to report a site $s \in S$ that is ε-close to a query $q \in P$.

We point locate to find a triangle Δ_q that contains q, and compute the sets $X_Q(s)$ for all nodes on the path $\mathcal{T}(q)$ towards this leaf using the Lemma 13 data structure. For each shortest path $Q \in \mathcal{Q}_\nu$ of each node $\nu \in \mathcal{T}(q)$ along this path, we then query the Lemma 16 data structure. Each such query returns a site $s_{\nu,Q}$ that minimizes $\hat{d}_Q(q, s_{\nu,Q})$ among the sites in $S_{\nu,Q}$ (as well as the actual distance estimate $\hat{d}_Q(q, s_{\nu,Q})$). Finally, we query the Lemma 15 data structure of the leaf triangle Δ_q, to compute an ε-close site from the subset of sites stored there. We report the site s with overall minimum distance estimate.

Let s^* be site in S closest to q. We now argue that the site s and the distance estimate $\hat{d}(s, q)$ that we return indeed satisfies $d(q, s^*) \leq d(q, s) \leq \hat{d}(s, q) \leq (1 + \varepsilon)d(q, s^*)$.

If s^* also lies in Δ_q, then Lemma 15 guarantees that s is within distance $(1 + \varepsilon)d(q, s^*)$. Otherwise, the shortest path $\pi(q, s^*)$ from q to s^* must intersect one of the shortest paths $Q \in \mathcal{Q}_\nu$ on one of the nodes on the root-to-leaf path $\mathcal{T}(q)$. In particular, consider a shortest path of the highest such node ν. It follows that s^* is one of the sites in $S_{\nu,Q}$, and thus Lemma 16 guarantees that the candidate $s_{\nu,Q}$ has the minimum $\hat{d}_Q(q, s_{\nu,Q})$ value among all sites in $S_{\nu,Q}$. Hence, our distance estimate satisfies $\hat{d}(q, s) \leq \hat{d}_Q(q, s_{\nu,Q}) \leq \hat{d}_Q(q, s^*)$. Lemma 11 tells us that $\hat{d}_Q(q, s^*) \leq (1 + \varepsilon)d(q, s^*)$ as desired. Finally, since our distance estimate $\hat{d}(q, s)$ corresponds to the length of an obstacle avoiding path, we also have $d(s, q^*) \leq d(s, q) \leq \hat{d}(q, s)$, and thus s is indeed ε-close to q.

Finding the triangle containing q, and traversing the root-to-leaf path corresponding to this triangle takes $O(\log n)$ time. Querying all $O(\log n)$ Lemma 16 data structures along this path would take a total of $O(k \log^2 n + k^2 \log n + k \log(mk) \log n)$ time. Finally, the query to the Lemma 15 data structure stored at the leaf takes $O(k \log^2 m)$ time. As before, the $O(k \log^2 n)$ term in the query time is only due to computing the sets $X_Q(s)$ in every node separately. By querying the Lemma 13 data structure, we can again reduce this term to $O(k \log n)$. This reduces the total time required to query the Lemma 16 data structures to $O(k^2 \log n + k \log(mk) \log n + k \log^2 m)$, which simplifies to $O(k^2 \log n + k \log n \log m + k \log^2 m)$. As this term still dominates the other terms, this is also the final query time.

Updates. We first describe the process for inserting a new site s into S. We point locate to find a triangle P_ν that contains s. When s lies in the interior of such a triangle, then it must lie in the interior of all ancestor subpolygons P_μ as well, so we insert s into the Lemma 15 data structure of the leaf ν, and the Lemma 16 data structures along the root-to-leaf path. If s lies on an edge e of P_ν that is also an edge in a shortest path in one of the separators, we use the pointer stored at this edge to jump to the shortest path $Q \in \mathcal{Q}_{\nu'}$ of the highest such node ν', and insert s in the Lemma 16 data structures of ν' and its ancestors.

We have to insert s into at most one Lemma 15 data structure, (in $O(k \log^2 m)$ time), and at most $O(\log n)$ Lemma 16 data structures. Naively inserting them into every such structure then gives a total insertion time of $O(k \log^2 n + k^2 \log n + k \log(mk) \log n + k \log^2 m)$. As with the queries, we can compute the sets $X_Q(s)$ in $O(k \log n)$ time in total using Lemma 13. This then gives us a total insertion time of $O(k^2 \log n + k \log n \log m + k \log^2 m)$.

When we delete a site, we simply delete the site from all places where it was inserted. This takes a total of $O(k \log n \log m + k \log^2 m)$ time. Plugging in $k = \lceil 1/\varepsilon \rceil$ gives us Theorem 1.

References

1 Pankaj K. Agarwal, Lars Arge, and Frank Staals. Improved dynamic geodesic nearest neighbor searching in a simple polygon. In Bettina Speckmann and Csaba D. Tóth, editors, *34th International Symposium on Computational Geometry, SoCG 2018, June 11-14, 2018, Budapest, Hungary*, volume 99 of *LIPIcs*, pages 4:1–4:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.SOCG.2018.4.

2 Pankaj K. Agarwal and Jiří Matoušek. Dynamic Half-Space Range Reporting and its Applications. *Algorithmica*, 13(4):325–345, 1995. doi:10.1007/BF01293483.

3 Jon Louis Bentley and James B Saxe. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980. doi:10.1016/0196-6774(80)90015-2.

4 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

5 Sergio Cabello, Delia Garijo, Antonia Kalb, Fabian Klute, Irene Parada, and Rodrigo I. Silveira. Algorithms for distance problems in continuous graphs. In Pat Morin and Eunjin Oh, editors, *19th International Symposium on Algorithms and Data Structures, WADS 2025, August 11-15, 2025, York University, Toronto, Canada*, volume 349 of *LIPIcs*, pages 13:1–13:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.WADS.2025.13.

6 Timothy M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *J. ACM*, 57(3):16:1–16:15, 2010. doi:10.1145/1706591.1706596.

7 Timothy M. Chan. Dynamic generalized closest pair: Revisiting eppstein's technique. In Martin Farach-Colton and Inge Li Gørtz, editors, *3rd Symposium on Simplicity in Algorithms, SOSA 2020, Salt Lake City, UT, USA, January 6-7, 2020*, pages 33–37. SIAM, 2020. doi:10.1137/1.9781611976014.6.

8 Timothy M. Chan. Dynamic geometric data structures via shallow cuttings. *Discret. Comput. Geom.*, 64(4):1235–1252, 2020. doi:10.1007/S00454-020-00229-5.

9 Bernard Chazelle and Leonidas J. Guibas. Visibility and intersection problems in plane geometry. *Discrete & Computational Geometry*, 4:551–581, 1989. doi:10.1007/BF02187747.

10 K. Clarkson. Approximation algorithms for shortest path motion planning. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 56–65, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/28395.28402.

11 Sarita de Berg, Tillmann Miltzow, and Frank Staals. Towards space efficient two-point shortest path queries in a polygonal domain. In Wolfgang Mulzer and Jeff M. Phillips, editors, *40th International Symposium on Computational Geometry, SoCG 2024, June 11-14, 2024, Athens, Greece*, volume 293 of *LIPIcs*, pages 17:1–17:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SOCG.2024.17.

12 Sarita de Berg, Frank Staals, and Marc J. van Kreveld. The complexity of geodesic spanners. *J. Comput. Geom.*, 15(1):21–65, 2024. doi:10.20382/JOCG.V15I1A2.

13 Leonidas J. Guibas and John Hershberger. Optimal shortest path queries in a simple polygon. *J. Comput. Syst. Sci.*, 39(2):126–152, 1989. doi:10.1016/0022-0000(89)90041-X.

14 John Hershberger and Subhash Suri. An optimal algorithm for euclidean shortest paths in the plane. *SIAM Journal on Computing*, 28(6):2215–2256, 1999. doi:10.1137/S0097539795289604.

15 Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha Sharir. Dynamic planar voronoi diagrams for general distance functions and their algorithmic applications. *Discret. Comput. Geom.*, 64(3):838–904, 2020. doi:10.1007/S00454-020-00243-7.

16 Chih-Hung Liu. Nearly optimal planar k nearest neighbors queries under general distance functions. *SIAM J. Comput.*, 51(3):723–765, 2022. doi:10.1137/20M1388371.

17 Eunjin Oh and Hee-Kap Ahn. Voronoi diagrams for a moderate-sized point-set in a simple polygon. *Discret. Comput. Geom.*, 63(2):418–454, 2020. doi:10.1007/S00454-019-00063-4.

18 Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986. doi:10.1145/6138.6151.

19 Mikkel Thorup. Compact oracles for approximate distances around obstacles in the plane. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Algorithms – ESA 2007*, pages 383–394, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-75520-3_35.

20 Joost van der Laan, Frank Staals, and Lorenzo Theunissen. Approximate dynamic nearest neighbor searching in a polygonal domain, 2026. arXiv:2603.11775.

21 Haitao Wang. A new algorithm for euclidean shortest paths in the plane. *J. ACM*, 70(2):11:1–11:62, 2023. doi:10.1145/3580475.