Computational Geometry: Theory and Applications 111 (2023) 101976

Contents lists available at ScienceDirect

Computational Geometry: Theory and Applications

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

Dynamic data structures for k-nearest neighbor queries

Sarita de Berg *, Frank Staals *

Utrecht University Department of Information and Computing Sciences, Princetonplein 5, 3584CC Utrecht, the Netherlands

ARTICLE INFO

Article history:
Received 14 December 2021
Received in revised form 28 September 2022
Accepted 23 November 2022
Available online 5 December 2022

Keywords:
Data structure
Simple polygon
Geodesic distance
Nearest neighbor searching

ABSTRACT

Our aim is to develop dynamic data structures that support k-nearest neighbors (k-NN) queries for a set of n point sites in the plane in $O(f(n)+k)$ time, where $f(n)$ is some polylogarithmic function of n. The key component is a general query algorithm that allows us to find the k-NN spread over t substructures simultaneously, thus reducing an $O(tk)$ term in the query time to $O(k)$. Combining this technique with the logarithmic method allows us to turn any static k-NN data structure into a data structure supporting both efficient insertions and queries. For the fully dynamic case, this technique allows us to recover the deterministic, worst-case, $O(\log^2 n/\log\log n+k)$ query time for the Euclidean distance claimed before, while preserving the polylogarithmic update times. We adapt this data structure to also support fully dynamic *geodesic* k-NN queries among a set of sites in a simple polygon. For this purpose, we design a shallow cutting based, deletion-only k-NN data structure. More generally, we obtain a dynamic planar k-NN data structure for any type of distance functions for which we can build vertical shallow cuttings. We apply all of our methods in the plane for the Euclidean distance, the geodesic distance, and general, constant-complexity, algebraic distance functions.

© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

In the k-nearest neighbors (k-NN) problem, we are given a set of n point sites S in some domain, and we wish to preprocess these points such that given a query point q and an integer k, we can find the k sites in S 'closest' to q efficiently. This static problem has been studied in many different settings [1–5]. We study the dynamic version of the planar k-nearest neighbors problem, in which the set of sites S may be subject to updates, i.e. insertions and deletions. We are particularly interested in two settings: (i) a setting in which the domain containing the sites contains (polygonal) obstacles, and in which we measure the distance between two points by their geodesic distance, the length of the shortest obstacle avoiding path, and (ii) a setting in which only insertions into S are allowed (i.e. no deletions).

In many applications involving distances and shortest paths, the entities involved cannot travel towards their destination in a straight line. For example, a person walking through the city center may want to find the k closest restaurants that currently have seats available. However, he or she cannot walk through walls, and hence, we need to explicitly incorporate such obstacles into the problem. This introduces additional complications as a single shortest path in a polygon with m vertices may have complexity $\Theta(m)$, and thus it requires $\Theta(m)$ time to compute such a path. We wish to limit the dependency on m in the space, query, and update times of our data structure as much as possible. In particular, we want to avoid having

* Corresponding authors.
E-mail addresses: s.deberg@uu.nl (S. de Berg), f.staals@uu.nl (F. Staals).

https://doi.org/10.1016/j.comgeo.2022.101976
0925-7721/© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

to spend $\Omega(m)$ time and space when we insert or delete a new site in S. In terms of the above example, we wish to avoid having to spend $\Omega(m)$ time every time some seats open up causing a restaurant to become available.

The second setting is motivated by classification problems. In k-nearest neighbor classifiers the sites in S all have a label, and the label of some query point q is predicted based on the labels of the k sites nearest to q [6]. When this label turns out to be sufficiently accurate, it is customary to then extend the data set by adding q to S. Hence, this naturally leads to the question whether there is an insertion-only data structure that can efficiently answer k-nearest neighbor queries.

The static problem. If the set of sites S is static, and k is known a priori, one option is to build the (geodesic) k^{th}-order Voronoi diagram of S [7]. This yields very fast $O(\log(n+m)+k)$ query times, where m is the complexity of the domain $\mathcal{P}$, however it is costly in space, as even in a simple polygon the diagram has size $O(k(n-k)+km)$. Moreover, k needs to be known a priori. In the scenario where the domain is the Euclidean plane, much more space efficient solutions have been developed. There is an optimal linear space data structure achieving $O(\log n + k)$ query time after $O(n \log n)$ deterministic preprocessing time [8,3]. Very recently, Liu showed how to achieve the same query and expected preprocessing time for general constant-complexity distance functions for arbitrary sites in $\mathbb{R}^2$, using $O(n \log\log n)$ space [5]. In case $\mathcal{P}$ is a simple polygon with m vertices, the problem has not explicitly been studied. The only known solution using less space than just storing the k^{th}-order Voronoi diagram is the fully dynamic 1-NN structure of Agarwal et al. [9]. It uses $O(n \log^3 n \log m + m)$ space, and answers queries in $O(k\,\text{polylog}(n+m))$ time (by deleting and reinserting the k-closest sites to answer a query).

Issues when inserting sites. Since nearest neighbor searching is decomposable, we can apply the logarithmic method [10] to turn a static k-NN searching data structure into an insertion-only data structure. For example, in the Euclidean plane this yields a linear space data structure with $O(\log^2 n)$ insertion time. However, since this partitions the set of sites S into $O(\log n)$ subsets, and we do not know how many of the k-nearest sites appear in each subset, we may have to consider up to k sites from *each* of the subsets, thus yielding an $O(k \log n)$ term in the query time. In Section 3 we will present a general technique that allows us to avoid this additional $O(\log n)$ factor.

Fully dynamic data structures. In case we wish to support both insertions and deletions, the problem becomes more complicated, and the existing solutions thereby much more involved. When we again consider the plane, and we wish to report only one nearest neighbor (i.e. 1-NN searching), several efficient fully dynamic data structures exist [11–13]. Actually, all these data structures are variants of the same data structure by Chan [11]. For the Euclidean distance, the current best result using linear space achieves $O(\log^2 n)$ worst-case query time, $O(\log^2 n)$ insertion time, and $O(\log^4 n)$ deletion time [12]. These results are deterministic, and the update times are amortized. The variant by Kaplan et al. [13] achieves similar results for general distance functions: $O(n \log^3 n)$ space, $O(\log^2 n)$ worst-case query time, and expected $O(\text{polylog}\, n)$ amortized update time. Using recent results on shallow cuttings by Liu [5] the space can be reduced to $O(n \log n)$, the insertion time to $O(\log^2 n)$, and the deletion time to $O(\log^4 n)$. The update times remain expected amortized. These data structures can also be used to answer k-NN queries, but when answering a query we run into the same problem as in the insertion-only case. That is, we consider $O(\log n)$ subsets, which results in a query time of $O(\log^2 n + k \log n)$ [11,5].

For the Euclidean case, Chan argues that the above data structure for 1-NN searching can be extended to obtain $O(\log^2 n / \log\log n + k)$ query time, while still retaining polylogarithmic updates [14]. Chan's data structure essentially maintains a collection of k-NN data structures built on subsets of the sites. A careful analysis shows that some of these structures can be rebuilt during updates, and that the cost of these updates is not too large. Queries are then answered by performing k_i-NN queries on several disjoint subsets of sites $S_1, .., S_t$ that together are guaranteed to contain the k nearest sites. However, perhaps because the details of the 1-NN searching data structure are already fairly involved, one aspect in the query algorithm is missing: how to determine the value k_i to query subset S_i with. While it seems that this issue can be fixed using randomization [15],[1] our general k-NN query technique (Section 3) allows us to recover deterministic, worst-case $O(\log^2 n / \log\log n + k)$ query time.

Organization and results. We develop dynamic data structures for k-NN queries in the plane whose query times are of the form $O(f(n)+k)$, where $f(n)$ is some function of n. In particular, we wish to avoid an $O(k \log n)$ term in the query time. To this end, we present a general query technique that given t disjoint subsets of sites $S_1, .., S_t$, each stored in a static data structure that supports k'-NN queries in $O(Q(n)+k')$ time, can report the k nearest neighbors among $\bigcup_{i=1}^{t} S_i$ in $O(Q(n)t + k)$ time. Our technique, presented in Section 3, is completely combinatorial, and is applicable to any type of sites. In Section 4, we then use this technique to obtain a k-NN data structure that supports queries in $O(Q(n) \log n + k)$ time and insertions in $O((P(n)/n) \log n)$ time, where $P(n)$ is the time required to build the static data structure. This result again applies to any type of sites. In the specific case of the Euclidean plane, we obtain a linear space data structure with $O(\log^2 n + k)$ query time and $O(\log^2 n)$ insertion time. At a slight increase of insertion time, we can also match the query time of Chan's [14] fully dynamic data structure. For general, constant-complexity, algebraic distance functions, we obtain the same query and insertion times (albeit the insertion time holds in expectation). In the case where the sites S are points inside a simple polygon $\mathcal{P}$ with m vertices, we use our technique to obtain the first *static* k-NN data structure that uses near-linear space, supports efficient (i.e. without the $O(k \log n)$ term) queries, *and* can be constructed efficiently.

[1] The main idea is that the data structure as is *can* be used to efficiently report all sites within a fixed distance from the query point (reporting all planes below a query point in $\mathbb{R}^3$). Combining this with an earlier random sampling idea [2] one can then also answer k-NN queries.

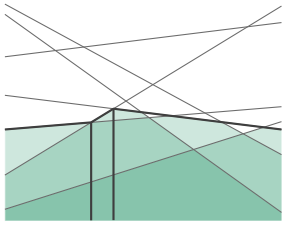

Fig. 1. A 2-shallow cutting of a set of lines F in $\mathbb{R}^2$ consisting of 3 prisms. The at most k-level $L_{\leq k}(F)$ is shown in green for $k = 0, 1, 2$. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

We now do get an $O(k \log m)$ term in the query time, as computing the distance between a pair of points already takes $O(\log m)$ time. Our data structure uses $O(n \log n + m)$ space, can be constructed in $O(n(\log n \log^2 m + \log^3 m) + m)$ time, and supports $O(\log(n+m) \log m + k \log m)$ time queries. In turn, this then also leads to a data structure supporting efficient, $O(\log^2 n \log^2 m + \log n \log^3 m)$ time, insertions.

In Section 5 we argue that our general query algorithm is the final piece of the puzzle for the fully dynamic case. For the Euclidean plane, this allows us to recover the deterministic, worst-case $O(\log^2 n / \log\log n + k)$ query time claimed before [14]. Insertions take amortized $O(\log^{3+\varepsilon} n / \log\log n)$ time, whereas deletions take $O(\log^{5+\varepsilon} n / \log\log n)$ time. We obtain the same query and (expected) update times in case of constant degree algebraic distance functions.

For the geodesic case there is one final hurdle to take. Chan's algorithm uses a partition-tree based "slow" dynamic k-NN query data structure of linear size as one of its subroutines (see Section 5.1 for details). Liu uses a similar trick (after appropriately linearizing the distance functions into $\mathbb{R}^c$ for some constant c) in their static k-NN data structure [16]. Unfortunately, this idea is not applicable in the geodesic setting, as it is unknown if an appropriate (shallow) simplicial partition exists, and the geodesic distance function cannot be linearized into a constant dimensional space (the dimension would need to depend on m). Instead, we design a simple, shallow-cutting based, alternative "slow" dynamic k-NN structure, that does extend to the geodesic setting. This way, we end up with an efficient (i.e. $O(\text{polylog}(n+m))$ expected updates, $O(\log^2 n \log^2 m + k \log m)$ queries) fully dynamic geodesic k-NN data structure.

2. Preliminaries

We can easily transform a k-nearest neighbors problem in $\mathbb{R}^2$ to a k-lowest functions problem in $\mathbb{R}^3$ by considering (the graphs of) the distance functions $f_s(x)$ of the sites $s \in S$. We discuss these problems interchangeably, furthermore we identify a function with its graph.

2.1. Shallow cuttings

Let F be a set of bivariate functions. We consider the arrangement of F in $\mathbb{R}^3$. The *level* of a point $q \in \mathbb{R}^3$ is defined as the number of functions in F that pass strictly below q. The *at most k-level* $L_{\leq k}(F)$ is then the set of points in $\mathbb{R}^3$ that have level at most k.

A *k-shallow cutting* $\Lambda_k(F)$ of F is a set of disjoint cells covering $L_{\leq k}(F)$, such that each cell intersects at most $O(k)$ functions [17]. When F is clear from the context we may write Λ_k rather than $\Lambda_k(F)$. We are interested only in the case where the cells are *(pseudo-)prisms*: constant-complexity regions that are bounded from above by a function in F, from the sides by vertical (with respect to the z-direction) surfaces that pass through an intersection curve between two functions in F, and unbounded from below. For example, if F is a set of planes, we can define the top of each prism to be a triangle. This allows us to find the prism containing a query point q by a point location query in the downward projection of the cutting. See Fig. 1. The subset $F_\nabla \subseteq F$ intersecting a prism ∇ is the *conflict list* of ∇. When, for every subset $F' \subseteq F$, the lower envelope $L_0(F')$ has linear complexity (for example, in the case of planes), a shallow cutting of *size* (the number of cells) $O(n/k)$ can be computed efficiently [5].

In general, let $T(n,k)$ be the time to construct a k-shallow cutting of maximum size $S(n,k)$ on n functions, and $Q(n,k)$ be the time to locate the prism containing a query point. We assume these functions are non-decreasing in n and non-increasing in k, and that $S(n,k) = \frac{1}{k} f(n)$, for some function $f(n)$.

2.2. A dynamic nearest neighbor data structure

We briefly discuss the main ideas of the dynamic nearest neighbor data structure by Chan [11,12] that was later improved by Kaplan et al. [13], as this also forms a key component in our fully dynamic k-NN data structures. For a more detailed description we refer to the original papers. For ease of exposition, we describe the data structure when F is a set of linear functions (planes). To make sure the analysis is correct for our definition of n (the current number of points in S), we rebuild the data structure from scratch whenever n has doubled or halved. The cost of this is subsumed in the cost of the other operations [11].

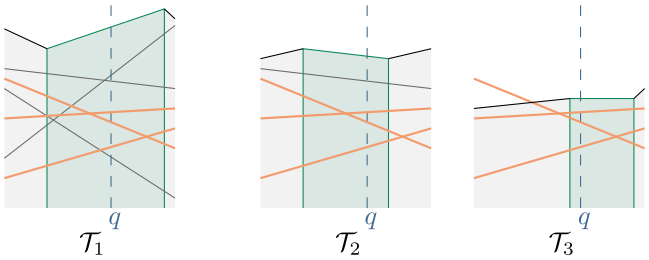

Fig. 2. Example of the dynamic 1-NN data structure. Only one shallow cutting (Λ_{k_j}) is shown for each tower. The orange planes in $\mathcal{T}_1$ and T_2 are pruned when building $\Lambda_{k_{j-1}}$, but are not removed from the conflict lists in Λ_{k_j}. When querying for the k-NN, the green prisms in Λ_{k_j} of each tower are considered. Note that the three orange planes occur in each of the conflict lists.

The data structure consists of $t = O(\log_b n)$ "towers" $\mathcal{T}^{(1)}, .., \mathcal{T}^{(t)}$, for some fixed $b \geq 2$. Each tower $\mathcal{T}^{(i)}$ consists of a hierarchy of shallow cuttings, which is built using a process that involves a subset of planes $F^{(i)} \subseteq F$. For $\mathcal{T}^{(1)}$ we have $F^{(1)} = F$, and a hierarchy of $\ell = \lfloor \log(n/k_0) \rfloor$ shallow cuttings, for a fixed constant k_0. For $j = 0, .., \ell$ we have a k_j-shallow cutting of a subset of the planes $F_j \subseteq F^{(1)}$, where $k_j = 2^j k_0$. We set $F_\ell = F^{(1)}$ and construct these cuttings from $j = \ell$ to 0. After computing $\Lambda_{k_j}(F_j)$, we find the set $F_j^\times$ of "bad" planes that intersect more than $c \log n$ prisms in $\Lambda_{k_\ell}(F_\ell), .., \Lambda_{k_j}(F_j)$, i.e. in all cuttings in the hierarchy so far. We *prune* these planes by setting $F_{j-1} = F_j \setminus F_j^\times$, and removing all planes in $F_j^\times$ from the conflict lists of the prisms in $\Lambda_{k_j}(F_j)$. Note that these bad planes are removed only from the conflict lists of the current cutting, and can still occur in conflict lists of higher level cuttings. In the final $\Lambda_{k_0}(F_0)$ cutting, each conflict list has a constant size of $O(k_0)$. We denote by $F^{(1)}_{\text{bad}} = F_0^\times \cup ... \cup F_\ell^\times$ the set of all bad planes generated during this process. By $F^{(1)}_{\text{live}}$ we denote the set of planes that have not been pruned during the process, so $F^{(1)}_{\text{live}} = F^{(1)} \setminus F^{(1)}_{\text{bad}}$. We then set $F^{(i+1)} = F^{(i)}_{\text{bad}}$ and recursively build $\mathcal{T}^{(i+1)}$ on the functions in $F^{(i+1)}$.

In the end, the set F is partitioned into sets $F^{(1)}_{\text{live}}, .., F^{(t)}_{\text{live}}$. When insertions and deletions take place, planes can move from a set $F^{(i)}_{\text{live}}$ to some $F^{(i')}_{\text{live}}$, but the property that these sets form a partition of F will be preserved. Kaplan et al. prove the following lemma on the size of $F^{(1)}_{\text{live}}$ after the preprocessing phase:

Lemma 1 *(Lemma 7.1 of [13]). For any $\zeta \in (0, 1)$ there exists a sufficiently large (but constant) choice of c, such that $|F^{(1)}_{\text{live}}| \geq (1 - \zeta)n$ after building $\mathcal{T}^{(1)}$.*

When $\zeta = 1/b$, we get $O(\log_b n)$ towers, for some fixed $b \geq 2$, as desired. According to Kaplan et al. [13], this is achieved by choosing $c \geq \frac{\gamma}{\zeta} = b\gamma = O(b)$, for some constant γ. Thus a plane occurs $O(b \log n)$ times in a tower. Here, we first consider only the case where $b = 2$.

To build a single tower, naively we would need to compute $O(\log n)$ shallow cuttings, each of which takes $O(n \log n)$ time. By using information of previously computed cuttings, Chan [12] recently achieved an overall construction time of $O(n \log n)$. The preprocessing time of the entire data structure thus adheres to the recurrence $P(n) \leq P(n/2) + O(n \log n)$. This solves to $P(n) = O(n \log n)$.

Insertions. To insert a plane f into F, we insert it into $F^{(1)}$. When we insert a function f into $F^{(i)}$ we assign it to $F^{(i)}_{\text{bad}}$, and thus recursively insert it in $F^{(i+1)}$. When $|F^{(i)}_{\text{bad}}|$ reaches $3/4 \cdot |F^{(i)}|$ we rebuild the towers $\mathcal{T}^{(i)}, .., \mathcal{T}^{(t)}$. The first tower, $\mathcal{T}^{(i)}$, is built on the planes $\{f\} \cup F^{(i)}_{\text{live}} \cup ... \cup F^{(t)}_{\text{live}}$, and the following towers are again built recursively on the new sets $F^{(i'-1)}_{\text{bad}}$. Only after $\Omega(|F^{(i)}|)$ insertions can such a rebuild occur. The insertion time is thus given by the recurrence $I(n) \leq I(3n/4) + O(P(n)/n)$, where $P(n)$ is the time to build the data structure on a set of n planes. Using $P(n) = O(n \log n)$, this results in an amortized insertion time of $O(\log^2 n)$.

Deletions. Deletions are not performed explicitly on the conflict lists. Instead, for each prism ∇ we keep track of the number of planes in F_∇ that have been deleted so far, denoted by d_∇. When deleting a plane f, we increase d_∇ for all prisms with $f \in F_\nabla$, and remove f from the set $F^{(i)}_{\text{live}}$ that includes f. When too many planes in a conflict list have been deleted, we *purge* the prism. In particular, we purge a prism ∇ in a k_j-shallow cutting when $d_\nabla \geq k_j/2 = 2^{j-1}k_0$. When a prism in $\mathcal{T}^{(i)}$ is purged, we mark it as such, and we reinsert all planes $f' \in F_\nabla \cap F^{(i)}_{\text{live}}$. These planes are effectively moved from $F^{(i)}_{\text{live}}$ to some other $F^{(i')}_{\text{live}}$. Note that we only reinsert planes that have not been deleted so far. This scheme ensures a prism is only purged after at least $k_j/2$ deletions, and this causes at most $|F_\nabla| = O(k_j)$ reinsertions. Thus, each increment of d_∇ causes amortized $O(1)$ reinsertions. This gives an amortized deletion time of $O(\log^4 n)$.

Nearest neighbor queries. When answering a nearest neighbor query for a query point q, we simply find the prism containing q in the lowest ($j = 0$) cutting of each $\mathcal{T}^{(i)}$ by a point location query in $O(\log n)$ time. We then go through each conflict list (of constant size) to find the plane that is lowest at q. If the plane we find for $\mathcal{T}^{(i)}$ is not in $F^{(i)}_{\text{live}}$, we ignore the result. When a prism has been purged, we simply skip it. Finally, we return the plane that is lowest among the $O(\log n)$ planes that are found. As we perform $O(\log n)$ point location queries, the query time is $O(\log^2 n)$.

k-nearest neighbors queries. Answering k-nearest neighbors queries using this dynamic 1-NN data structure is straightforward. For each tower we consider the prism containing q of the shallow cutting at level $j_k := \lceil \log(Ck/n) \rceil$, for some large enough constant C. The size of the conflict list of each of these prisms is $O(k)$, thus we can find the k-lowest *live* planes in each conflict list in $O(k)$ time. Chan [11] proves that it is indeed sufficient to consider only planes in these conflict lists. This query algorithm has a running time of $O(\log^2 n + k \log n)$.

However, even if we were to know the exact number of the k-nearest neighbors that occurs in each tower, we would not be able to support k-NN queries in $O(\log^2 n + k)$ time. When a plane is pruned during the preprocessing, or when a prism is purged, the plane is only removed from the conflict lists of the current shallow cutting. It can thus still occur in other shallow cuttings in the hierarchy. This means that we can encounter the same plane multiple times when querying each tower for the k-lowest planes. See Fig. 2 for an illustration. As there are $O(\log n)$ towers, this yields an $O(k \log n)$ term in the query time.

General distance functions. Kaplan et al. [13] showed how to adapt Chan's data structure to support more general shallow cutting algorithms. The main differences between their data structure and the one from Chan is that planes are only pruned when they appear in $(S(n, 1)/n) \cdot c \log n$ conflict lists. Kaplan et al. essentially prove the following lemma.

Lemma 2 *(Kaplan et al. [13]). Given an algorithm that constructs a k-shallow cutting of size $S(n, k)$ on n functions in $T(n, k)$ time, such that the prism containing a query point can be located in $Q(n, k)$ time, we can construct a data structure of size $O(S(n, 1) \log n)$ that dynamically maintains a set of at most n functions F. Reporting the lowest function at a query point q takes $O(Q(n, 1) \log n)$ time, inserting a function in F takes $O((T(n, 1)/n) \log^2 n)$ amortized time, and deleting a function from F takes $O((T(n, 1) S(n, 1)/n^2) \log^4 n)$ amortized time.*

From now on we consider the general variant, where the data structure consists of $O(\log_b n)$ towers. The following lemma describes the properties of the 1-NN data structure we need to construct our general fully dynamic k-nearest neighbors data structure.

Lemma 3. *Let $b \geq 2$ be any fixed value and $S(n, k)$ be the maximum size of a k-shallow cutting. There is a dynamic nearest neighbor data structure that has the following properties.*

1. *The data structure consists of $O(\log_b n)$ towers.*
2. *A function occurs $O(b \log n \cdot S(n, 1)/n)$ times in a conflict list in a single tower.*
3. *The insertion time is $O(b \log_b n \cdot P(n)/n)$, where $P(n)$ is the preprocessing time.*
4. *A deletion causes amortized $O(b \log_b n \log n \cdot S(n, 1)/n)$ reinsertions.*
5. *To find the k-NN of a query point q, it is sufficient to consider $O(\log_b n)$ prisms, namely for each tower the prism containing q of the shallow cutting at level $j_k := \lceil \log(Ck/n) \rceil$, for some large enough constant C.*

3. Querying multiple k-NN data structures simultaneously

In this section we introduce a method to find the k-nearest neighbors of a query point q spread over t (disjoint) k'-NN data structures storing a set of sites S simultaneously. Suppose the query time of such a k'-NN data structure is $O(Q(n) + k')$, for a non-decreasing function Q. Naively, querying each data structure for the k closest sites would take $O(Q(n)t + tk)$ time. Our method allows us to find the k-NN over all these data structures in $O(Q(n)t + k)$ time instead, thus reducing the $O(tk)$ term to $O(k)$. More formally, we prove the following result.

Theorem 1. *Let $S_1, .., S_t$ be disjoint sets of point sites of sizes $n_1, .., n_t$, each stored in a data structure that supports k'-NN queries in $O(Q(n_i) + k'T)$ time, where T is the time for evaluating $d(p, q)$. There is a k-NN data structure on $\bigcup_i S_i$ that supports queries in $O(Q(n)t + kT)$ time. The data structure uses $O(\sum_i C(n_i))$ space, where $C(n_i)$ is the space required by the k-NN structure on S_i.*

Proof. We first describe the algorithm to query all t data structures simultaneously, and then analyze its running time.

Query algorithm. We use the heap selection algorithm of Frederickson [18] to answer k-NN queries efficiently. This algorithm finds the k smallest elements of a binary min-heap of size $N \gg k$ in $O(k)$ time. Running this algorithm on a heap that contains all sites $s \in S$ exactly once, with the distance $d(s, q)$ as key for each site, would return the k-nearest neighbors. However, building this entire heap takes linear time. To overcome this issue, we do not construct the entire heap we query before starting the algorithm. Instead, the heap is expanded during the query when necessary. See Fig. 3 for an example.

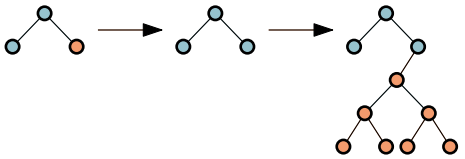

Fig. 3. Example of expansion. Blue elements have been visited by the algorithm, orange elements have not. The expansion (building the next subheap) occurs when the last element is visited.

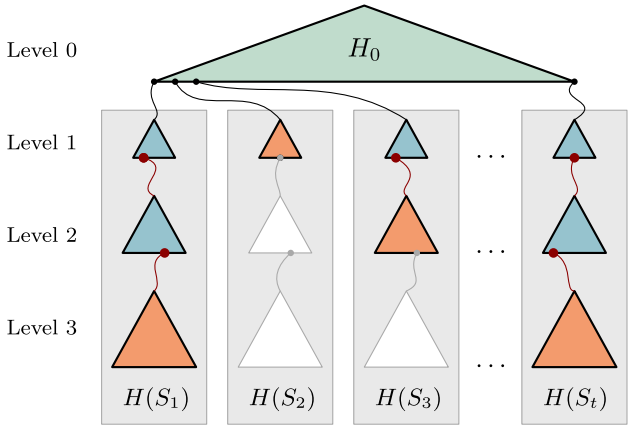

Fig. 4. The heap that we construct for the k-nearest neighbors query. The subheaps of which all elements have been visited are indicated in *blue*. The subheaps that have been built, but for which not all elements have been visited, are indicated in *orange*. The *white* subheaps have not been built so far, because not all elements of their predecessor have been visited.

To know when to expand the heap, we require that the Frederickson algorithm only visits a node *after* the parent of that node has been visited. This implies that once a leaf of the heap is visited by the algorithm, we can expand the heap further below that leaf without hindering the algorithm.

Next, we describe the heap H, on which we call the heap selection algorithm, in more detail. As stated before, H contains all sites $s \in S$ exactly once, with the distance $d(s,q)$ as key for each site. Let $S_1, .., S_t$ be the partition of S into t disjoint sets, where S_j is the set of sites stored in the j-th k'-NN data structure. For each set of sites S_j, $j \in \{1, .., t\}$, we define a heap $H(S_j)$ containing all sites in S_j. We then "connect" these t heaps by building a dummy heap H_0 of size $O(t)$ that has the roots of all $H(S_j)$ as leaves. We set the keys of the elements of H_0 to $-\infty$. Let H be the complete data structure (heap) that we obtain this way, see Fig. 4. It follows that we can now compute the k sites closest to q by finding the $|H_0| + k$ smallest elements in the resulting heap H and reporting only the non-dummy sites.

What remains is how to (incrementally) build the heaps $H(S_j)$ while running the heap selection algorithm. Each such heap consists of a hierarchy of *subheaps* $H_1(S_j), .., H_{O(\log n)}(S_j)$, such that every element of S_j appears in exactly one $H_i(S_j)$. Moreover, since the sets $S_1, .., S_j$ are pairwise disjoint, this holds for any $s \in S$, i.e. s appears in exactly one $H_i(S_j)$. The *level 1* heaps, $H_1(S_j)$, consist of the $k_1 = Q(n)$ sites in S_j closest to q, which we find by querying the static data structure of S_j. The subheap $H_i(S_j)$ at level $i > 1$ is built only after the last element e of $H_{i-1}(S_j)$ is visited by the heap selection algorithm. We then add a pointer from e to the root of $H_i(S_j)$, such that the root of $H_i(S_j)$ becomes a child of e, as in Fig. 3.

To construct a subheap $H_i(S_j)$ at level $i > 1$, we query the static data structure of S_j using $k_i = k_1 2^{i-1}$. The new subheap is built using all sites returned by the query that have not been encountered earlier. It follows that all elements of $H_i(S_j)$ are larger than any of the elements in $H_1(S_j), .., H_{i-1}(S_j)$. Thus, the heap property is preserved.

Analysis of the query time. As stated before, finding the k-smallest non-dummy elements of H takes $O(k + |H_0|)$ time [18]. We now analyze the time used to construct H.

First, the level 0 and level 1 heaps are built. Building H_0 takes only $O(t)$ time. To build the level 1 heaps, we query each of the substructures using $k_1 = Q(n)/T$, where T denotes the time to evaluate the distance function. In total these queries take $O((Q(n) + k_1 T)t) = O(Q(n)t)$ time. Retrieving the next k_i elements to build $H_i(S_j)$ for $i > 1$ requires a single query, and thus takes $O(Q(n) + k_i T)$ time. To bound the time used to build all heaps at level greater than 1, we first prove the following lemma.

Lemma 4. *The size of a subheap $H_i(S_j)$, $j \in \{1, .., t\}$, at level $i > 1$ is exactly $k_1 2^{i-2}$.*

Proof. To create $H_i(S_j)$, we query the static data structure of S_j to find the $k_1 2^{i-1}$ sites closest to q. Of these sites, only the ones that have not been included in any of the lower level subheaps are included in $H_i(S_j)$. The sites previously encountered are exactly the $k_1 2^{i-2}$ sites returned in the previous query. It follows that $|H_i(S_j)| = k_1(2^{i-1} - 2^{i-2}) = k_1 2^{i-2}$. □

Building $H_i(S_j)$ takes $O(Q(n) + k_i T)$ time. To pay for this, we charge $O(T)$ to each element of $H_{i-1}(S_j)$. Because we choose $k_1 = Q(n)/T$, Lemma 4 implies that $|H_{i-1}(S_j)| = \Omega(Q(n)/T)$, and that $k_i = k_1 2^{i-1} = 2^2 k_1 2^{i-3} = O(|H_{i-1}(S_j)|)$. Note that the heap $H_i(S_j)$, $i > 1$, is only built when the final element of $H_{i-1}(S_j)$ is visited. Thus, we only charge elements of the heaps of which all elements have been visited (shown blue in Fig. 4). In total, $O(k)$ elements (not in H_0) are visited, so the total size of these subheaps is $O(k)$. From this, and the fact that all subheaps are disjoint, it follows that we charge $O(T)$ to only $O(k)$ sites. The total running time thus becomes $O(t) + O(Q(n)t) + O(kT) = O(Q(n)t + kT)$. This completes the proof of Theorem 1. □

4. An insertion-only data structure

We describe a method that transforms a static k-NN data structure with query time $O(Q(n)+k)$ into an insertion-only k-NN data structure with query time $O(Q(n)\log n+k)$. Insertions take $O((P(n)/n)\log n)$ time, where $P(n)$ is the preprocessing time of the static data structure, and $C(n)$ is its space usage. We assume $Q(n)$, $P(n)$, and $C(n)$ are non-decreasing.

To support insertions, we use the logarithmic method [10]. We partition the sites into $O(\log n)$ groups $S_1, .., S_{O(\log n)}$ with $|S_i| = 2^i$ for $i \in \{1, .., O(\log n)\}$. To insert a site s, a new group containing only s is created. When there are two groups of size 2^i, these are removed and a new group of size 2^{i+1} is created. For each group we store the sites in the static k-NN data structure. This results in an amortized insertion time of $O((P(n)/n)\log n)$. This bound can also be made worst-case [10]. The main remaining issue is then how to support queries in $O(Q(n)\log n + k)$ time, thus avoiding an $O(k \log n)$ term in the query time. Applying Theorem 1 directly solves this problem, and we thus obtain the following result.

Theorem 2. *Let S be a set of n point sites, and let $\mathcal{D}$ be a static k-NN data structure of size $O(C(n))$, that can be built in $O(P(n))$ time, and answer queries in $O(Q(n) + k)$ time. There is an insertion-only k-NN data structure on S of size $O(C(n))$ that supports queries in $O(Q(n)\log n + k)$ time. Inserting a new site in S takes $O((P(n)/n)\log n)$ time.*

4.1. Points in the plane

In the Euclidean metric, k-nearest neighbors queries in the plane can be answered in $O(\log n + k)$ time, using $O(n)$ space and $O(n \log n)$ preprocessing time [8,3]. Hence:

Corollary 1. *There is an insertion-only data structure of size $O(n)$ that stores a set of n sites in $\mathbb{R}^2$, allows for k-NNs queries in $O(\log^2 n + k)$ time, and insertions in $O(\log^2 n)$ time.*

If we increase the size of each group in the logarithmic method to b^i, with $b = \log^{\varepsilon} n$ and $\varepsilon > 0$, we get only $O(\log_b n)$ groups instead of $O(\log n)$. This reduces the query time to $O(\log^2 n/\log\log n + k)$, matching the fully dynamic data structure. However, this also increases the insertion time to $O(\log^{2+\varepsilon} n/\log\log n)$. For general constant-complexity distance functions, we achieve the same query time using Liu's data structure [5]. The space usage is $O(n \log\log n)$ and the expected insertion time is $O(\log^2 n)$.

4.2. Points in a simple polygon

In the geodesic k-nearest neighbors problem, S is a set of sites inside a simple polygon $\mathcal{P}$ with m vertices. For any two points p and q the distance $d(p, q)$ is defined as the length of the shortest path $\pi(p, q)$ between p and q fully contained within $\mathcal{P}$. The input polygon $\mathcal{P}$ can be preprocessed in $O(m)$ time so that the geodesic distance $d(p, q)$ between any two points $p, q \in \mathcal{P}$ can be computed in $O(\log m)$ time [19].

To apply Theorem 2, we need a static data structure for geodesic k-NN queries. We can build such a data structure by combining the approach of Chan [2] and Agarwal et al. [9]. The data structure consists of a hierarchy of lower envelopes of random samples $R_0 \subset R_1 \subset .. \subset R_{\log n}$. For each sample, we store a (topological) vertical decomposition of the downward projection of the lower envelope, and the conflict lists of the corresponding pseudo-prisms. The downward projection is a geodesic Voronoi diagram, which can be preprocessed for efficient point location queries using the method of Oh and Ahn [20]. From a Clarkson and Shor style sampling argument, it follows that total expected size of all conflict lists of one sample is $O(n)$. Thus, the space of the resulting data structure is $O(n \log n)$ in expectation. We can then find a prism in one of the vertical decompositions that contains the query point and whose conflict list has size $O(k)$ in $O(\log(n+m) + k \log m)$ time [2]. This allows us to answer k-NN queries in the same time. The crux in this approach is in how to compute the

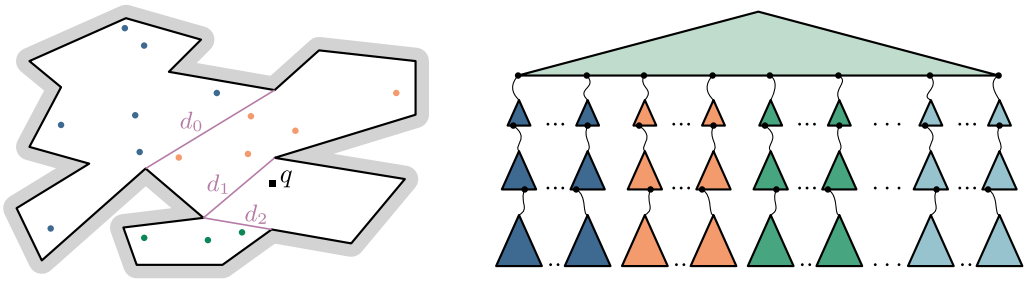

Fig. 5. A partial decomposition of $\mathcal{P}$ and the corresponding heap used in a k-NN query for q.

conflict lists. We can naively compute these in $O(mn)$ time by explicitly constructing the geodesic distance function for each site, and intersecting it with each of the $O(n)$ pseudo-prisms. It is unclear how to improve on this bound.

Theorem 3. *Let S be a set of n sites in a simple polygon $\mathcal{P}$ with m vertices. In $O(n(\log n \log^2 m + \log^3 m))$ time we can build a data structure of size $O(n \log n \log m)$, excluding the size of the polygon, that can answer k-NN queries with respect to S in $O(\log(n+m)\log m + k \log m)$ time.*

Proof. To circumvent the issue we describe above, we partition $\mathcal{P}$ into two subpolygons $\mathcal{P}_r$ and $\mathcal{P}_\ell$ of roughly the same size by a diagonal d. Let S_r and S_ℓ denote the sites in $\mathcal{P}_r$ and $\mathcal{P}_\ell$, respectively. We then build a data structure that can find the k-nearest neighbors of any query point $q \in \mathcal{P}_r$ among the sites in S_ℓ efficiently. To allow us to answer general queries, i.e. to find the k-nearest neighbors for *any* $q \in \mathcal{P}$ among *all* sites in S, we build such a data structure on both $\mathcal{P}_r$ and $\mathcal{P}_\ell$, and then recursively partition the polygon further. This results in a decomposition of the polygon of $O(\log m)$ levels.

To answer k-NN queries for a point in $\mathcal{P}_r$ among the sites in S_ℓ, we consider the Voronoi diagram of sites in S_ℓ (resp. S_r) restricted to $\mathcal{P}_r$. The bisector between any two sites in S_ℓ restricted to $\mathcal{P}_r$ is x-monotone with respect to the vertical diagonal d. Agarwal et al. use this fact to construct an efficient data structure (Theorem 22 of [9]) for our problem. This is essentially the data structure that was described at the start of this section. However, because the bisectors are x-monotone, we can efficiently compute the conflict lists by only considering the functions intersecting the corners of each prism. Because of the monotonicity, a bisector cannot enter and exit again through the same edge of a prism. Therefore, the conflict list of a prism is simply the union of the conflict lists of its corners (Lemma 15 of [9]). For each function f, we compute the vertices it conflicts with by first finding a vertex v on d that conflicts with f, and then performing a breadth first search to find *all* vertices (corners) it conflicts with. Thus, we can build the data structure in $O(n(\log n \log m + \log^2 m))$ time (see [9] for details). The data structure requires $O(n \log n)$ space, excluding the size of the polygon, and it allows us to find the k-nearest neighbors among S_ℓ for a point $q \in \mathcal{P}_r$ in $O((\log n + k)\log m)$ expected time [9].

To answer a k-NN query in $\mathcal{P}$, we first check whether the query point q lies in $\mathcal{P}_r$ or $\mathcal{P}_\ell$. When q lies in $\mathcal{P}_r$ (resp. $\mathcal{P}_\ell$), we query the data structure on S_ℓ to find the k-nearest neighbors among this set. To find the k-nearest neighbors among S_r, we recursively query $\mathcal{P}_r$. At each level of the decomposition, we thus consider a data structure on a single set of sites (S_ℓ or S_r) (see Fig. 5). In the next paragraph, we describe how to improve the query time of the static data structure, which can answer k-NN queries for a query point in $\mathcal{P}_r$ among sites in S_ℓ, to $O(\log(n+m) + k \log m)$ time. It follows that, using our technique from Section 3, we can find the k-NN in S, which are spread over $O(\log m)$ of these data structures, in $O(\log(n+m)\log m + k \log m)$ time.

Improving the query time. The query time of the static data structure is determined by two factors: a point location query in the topological vertical decomposition of a geodesic Voronoi diagram, this takes $O(\log n \log m)$ time, and $O(k)$ distance queries that take $O(\log m)$ time. We can improve the point location time to $O(\log(n+m))$ by incorporating the idea of Oh and Ahn [20] to approximate a geodesic Voronoi diagram by a polygonal subdivision. Given the exact location of the degree-1 and degree-3 vertices of the Voronoi diagram, they approximate each common boundary of two Voronoi regions by connecting the two endpoints using at most three line segments. This allows them to find the (not-approximated) Voronoi region that contains a query point q in $O(\log(n+m))$ time, using $O(n \log(n+m))$ preprocessing time. In our case, we want to find the (pseudo-)prism that contains the query point. This corresponds to finding the (pseudo-)trapezoid, which is the downward projection of the prism, of the (topological) vertical decomposition of a geodesic Voronoi diagram that contains q. Thus, we need to slightly adapt their approach to not only find the Voronoi region of the query point q, but also the exact trapezoid containing q.

Instead of approximating the bisector connecting two Voronoi vertices, we approximate each part of the bisector between two vertices of the vertical decomposition separately, see Fig. 7. To approximate a bisector of sites s_1 and s_2 between vertices u, v, Oh and Ahn first find two points t_1, t_2 such that the geodesic convex hull of t_1, t_2, u, v is contained in the Voronoi regions of s_1 and s_2, and the boundary of this convex hull consists of at most four maximal concave polygonal curves. They then approximate the bisector by either the line segment uv, when this is contained in the convex hull, or a polygonal

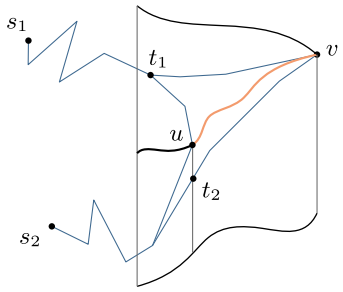

Fig. 6. The four convex chains between u, t_1, v, t_2 are used to approximate the orange bisector between u and v.

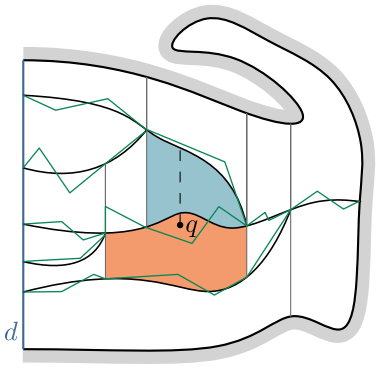

Fig. 7. Approximation (in green) of the Voronoi diagram in $\mathcal{P}_r$. To find the trapezoid containing q, we consider both colored trapezoids.

curve consisting of extensions of the edges incident to u and v of $\pi(u, v)$, and a line segment tangent to $\pi(t_1, v)$ and $\pi(t_2, u)$ that connects them.

We now consider the vertical decomposition of our Voronoi diagram. We want to approximate a bisector of sites s_1 and s_2 between two vertices u, v of the vertical decomposition. Without loss of generality, let v be the vertex furthest from d. Oh and Ahn choose t_i as the junction of $\pi(s_i, u)$ and $\pi(s_i, v)$. When choosing the t_i's like this, it could be that either of the t_i's is not contained in the trapezoid whose boundary we are approximating. In this case, we instead choose t_i as the intersection between $\pi(s_i, v)$ and the left line segment defining the trapezoid, see Fig. 6. As $\pi(s_i, v)$ is contained in the Voronoi region of s_i, this intersection point indeed exists. Note that this also ensures the convex hull is still bound by at most four maximal concave curves. Thus, we can use we can use the algorithm of Oh and Ahn [20] to approximate the bisector within the convex hull.

To find the trapezoid containing a query point q, we use the query algorithm of Oh and Ahn [20]. The query is performed as follows: first, we find the approximated trapezoid containing q by shooting a ray upwards from q, and determining what segment in the approximated Voronoi diagram (or the polygon boundary) is hit. Let s be the site whose Voronoi region we find. Suppose this is not the real trapezoid containing q, then q lies in a region bounded by part of a bisector of s and some other site t, and the approximation of that bisector, see Fig. 7. To find this approximated bisector (and thus the trapezoid in the Voronoi region of t containing q), we shoot a ray from q in direction opposite to the edge of $\pi(s, q)$ incident to q. This does not intersect the real bisector, and thus intersects the approximated bisector of s and t. Finally, we compare the distance from q to the two sites to find the trapezoid of the vertical decomposition containing q. □

Applying Theorem 1 to the data structure of Theorem 3 proves the following corollary.

Corollary 2. *Let $\mathcal{P}$ be a simple polygon with m vertices. There is an insertion-only data structure of size $O(n \log n \log m + m)$ that stores a set of n point sites in $\mathcal{P}$, allows for geodesic k-NN queries in $O(\log(n + m) \log n \log m + k \log m)$ expected time, and inserting a site in $O(\log^2 n \log^2 m + \log n \log^3 m)$ time.*

5. A fully dynamic data structure

In this section, we develop a k-NN data structure that supports both insertions and deletions, building on the results of Chan [14]. The data structure we propose is an extension of the nearest neighbor data structure of Lemma 2, and achieves the following result.

Theorem 4. *Given an algorithm that constructs a k-shallow cutting of size $S(n,k)$ on n functions in $T(n,k)$ time, such that the prism containing a query point can be located in $Q(n,k)$ time, we can construct a data structure that dynamically maintains a set of at most n functions F for k-NN queries. The size of the data structure is $O(S(n,1)^2/n \cdot \log n \log\log n)$, and operations are performed in time:*

$$\begin{aligned} &\textit{Query:} && O(Q(n,1)\log n/\log\log n + k)\\ &\textit{Insertion:} && O((S(n,1)T(n,1)/n^2)\log^{2+\varepsilon} n)^*\\ &\textit{Deletion:} && O((T(n,1)/\log\log n + T(n,n/\log n))\cdot S(n,1)^2\log^{4+\varepsilon} n/n^3)^*\end{aligned}$$

**amortized*

In Section 5.1, we first describe the k-NN data structure for planes by Chan [14], and then fill in the part missing from Chan's query algorithm. This data structure forms the basis for our general k-NN data structure. In Section 5.2, we then discuss a simple deletion-only k-NN structure. This deletion-only structure allows us to adapt Chan's k-NN data structure to more general distance functions like the geodesic distance. In Section 5.3, we analyse the running times of our general k-NN data structure with respect to the Euclidan distance, as this is somewhat easier to follow. In Section 5.4, we prove the general running times with respect to any vertical shallow cuttings as stated in Theorem 4. Finally, in Section 5.5, we apply Theorem 4 to the case of general constant-complexity distance functions and the geodesic distance.

5.1. A dynamic k-NN data structure for planes

Chan [14] describes how to adjust his original 1-NN data structure to efficiently perform k-NN queries. We denote this (adjusted) data structure by $\mathcal{D}$. There are two main changes in this data structure: the conflict lists are stored in k-NN data structures, and the number of towers is reduced by using $b = \log^{\varepsilon} n$. Only the *live* planes of the conflict list of each prism ∇ are stored in a data structure $\mathcal{D}_0$ that uses linear space, can perform k'-NN (or k'-lowest planes) queries in $O(Q_0(|F_\nabla|)+k')$ time, and deletions in $D_0(|F_\nabla|)$ time. A different data structure is used to store small and large conflict lists. After building a tower $\mathcal{T}^{(i)}$, each data structure $\mathcal{D}_0$ of a prism ∇ is built on $F_\nabla \cap F^{(i)}_{\text{live}}$. As the $\mathcal{D}_0$ data structures use linear space, the space usage of the entire data structure is $O(n\log n)$.

Insertions are performed as in the original data structure, but the deletion of a plane h requires more effort than in the original. In addition to increasing d_∇ for each prism containing h, h is explicitly removed from the $\mathcal{D}_0$ data structures. Note that $\mathcal{T}^{(i)}$ for which $h \in F^{(i)}_{\text{live}}$ is the only tower whose $\mathcal{D}_0$ data structures contain h. When a prism in tower $\mathcal{T}^{(i)}$ is purged, we also delete its planes from the other $\mathcal{D}_0$ data structures in $\mathcal{T}^{(i)}$ to retain this property. This gives an amortized expected update time of $U(n) = O(\log^{6+\varepsilon} n)$ [14]. The improvement of Kaplan et al. [13] reduces this to $O(\log^{5+\varepsilon} n)$, and the improvement of Chan [12] makes it deterministic. In Section 5.3 we discuss this update time in more detail w.r.t. our adaption of the data structure. It follows from Lemma 3 and the above modifications that:

Lemma 5 *(Chan [14]). Let q be a query point. In $O(t\log n)$ time, we can find $t = O(\log_b n)$ prisms $\nabla_1, .., \nabla_t$, such that: (i) all prisms contain q, (ii) the conflict list of each prism has size $O(k)$, (iii) the conflict lists are pairwise disjoint and stored in a $\mathcal{D}_0$ data structure, and (iv) the k sites in S closest to q appear in the union of the conflict lists of those prims.*

So, to answer k-NN queries we can use a k_i-NN query on each $\mathcal{D}_0$ data structure of the prisms $\nabla_1, .., \nabla_t$, where k_i is the number of sites from the k-nearest neighbors of q that appear in the conflict list of ∇_i. This takes $O(\sum_{i=1}^{t} Q_0(k) + k_i) = O(\sum_{i=1}^{t} Q_0(k) + k)$ time. However, it is unclear how to compute those k_i values. Fortunately, we can use Theorem 1 to find the k-nearest neighbors over all of the substructures in $O(Q_0(k)\log_b n + k)$ time. Plugging in the appropriate query time $Q_0(k)$ (see Chan [14] and Section 5.3), this achieves a total query time of $O(\log^2 n/\log\log n + k)$ time as claimed.

5.2. A simple deletion-only data structure

Let H be a set of n planes, and let $r \in \mathbb{N}$ be a parameter. We develop a data structure that supports deletions, and reporting the t lowest planes above a query point $q \in \mathbb{R}^2$. For our application, we are mostly interested in the deletion time of the data structure, and less in the query time. By picking r to be somewhat small, we can make deletions efficient at the cost of making the query time fairly terrible.

Lemma 6. *For any fixed r, we can construct a deletion-only data structure $\mathcal{D}_0$ of size $O(n\log r)$, or $O(n)$ when $n \geq r^{1/\varepsilon}$, in $O(n\log n)$ time that stores a set of n planes, allows for t-lowest planes queries in $O(\log r + n/r + t)$ time and deletions in $O(r\log n)$ amortized time.*

Proof. Our entire data structure consists of just $\ell = O(\log r)$ k_i-shallow-cuttings $\Lambda_{k_0}, .., \Lambda_{k_\ell}$ of the planes, for values $k_i = 2^i\lceil n/r\rceil$, for $i = 0, .., \ell$. Hence, this uses $O(n\log r)$ space in total. We can compute the shallow cuttings along with their conflict lists in $O(n\log n)$ time [3]. Note that even when $r > n$, we only need to build $O(\log n)$ shallow cuttings.

Deletions. If we delete a plane, we remove it from all conflict lists in all cuttings. Since cutting Λ_{k_i} has size $O(r/2^i)$, each plane occurs at most $O(r/2^i)$ times. Hence, the total time to go through all of these prisms is $\sum_{i=0}^{\log r} r/2^i = O(r)$ time. When more than half of the planes from any conflict list are removed, we rebuild the entire data structure. Because the smallest conflict list contains at least n/r planes, at least $n/2r$ deletions take place before a global rebuild. We charge the $O(n \log n)$ cost of rebuilding to these planes, so we charge $O(r \log n)$ to each deletion. Deletions thus take amortized $O(r \log n)$ time.

Queries. We report the t-lowest planes at a query point q as follows. We consider the cutting for which $k_i = 2^i \lceil n/r \rceil = O(t)$, so at level $i = \lceil \log(Ctr/n) \rceil$, for some large enough constant C. When $t < n/r$, there is no such cutting, so we query the lowest level cutting instead. We find the prism containing q by a point location query. As the largest cutting has size $O(r)$, this takes $O(\log r)$ time. We then simply report the t lowest planes at q by going through the entire conflict list. This results in a query time of $O(\log r + n/r + t)$.

Reducing space usage. When n is large w.r.t. r, that is $n \geq r^{1/\varepsilon}$, we can use a similar approach to Chan [11,14] to achieve linear space usage. Instead of storing the conflict lists explicitly, we only store the prisms of the shallow cuttings. Additionally, we store the planes in an auxiliary halfspace range reporting data structure [21] with $O(n \log n)$ preprocessing time, $O(n^{1-\varepsilon})$ query time, $O(\log n)$ deletion time, and linear space. This results in a linear space data structure. To delete a plane, we simply delete it in the auxiliary halfspace range reporting data structure in $O(\log n)$ time. After $\frac{n}{2r}$ deletions we rebuild the entire data structure. Thus the amortized deletion time remains $O(r \log n)$.

When performing a query, we first locate the prism ∇ as before. We then query the halfspace range reporting data structure with the intersection point of the vertical line through q and the roof of ∇. We find the t lowest planes by going through the $O(t)$ returned planes. This results in a query time of $O(n^{1-\varepsilon} + \log r + n/r + t)$. Because $n \geq r^{1/\varepsilon}$, we have $n^{1-\varepsilon} \leq n/r$, thus the query time is $O(\log r + n/r + t)$. □

General data structure. The data structure of Lemma 6 can be applied to any type of functions for which we have an algorithm to compute vertical k-shallow cuttings. Refer to Section 2.1 for the definitions regarding the computation of k-shallow cuttings. The data structure uses $O(S(n,1) \log r)$ space. Note that the "lowest" cutting we use is an n/r-shallow cutting. It follows that constructing all shallow cuttings takes $O(T(n, n/r) \log r)$ time. To delete a function, we remove all occurrences of the function from the conflict lists in

$$\sum_{i=0}^{\log r} S(n, k_i) = \sum_{i=0}^{\log r} \frac{1}{k_i} f(n) = \sum_{i=0}^{\log r} \frac{r}{2^i} f(n) = O((r/n) S(n,1))$$

time, where we used the assumption that $S(n,k) = \frac{1}{k} f(n)$, for some function $f(n)$. Additionally, we charge $O((r/n) T(n, n/r) \log r)$ to the deletion to pay for the global rebuild. To answer a query, we simply find the prism containing q in one cutting, so the query time is $O(Q(n, n/r) + n/r + t)$. This results in the following general lemma.

Lemma 7. *For any fixed r, we can construct a deletion-only data structure of size $O(S(n,1) \log r)$ in $O(T(n, n/r) \log r)$ time that stores a set of n functions, allows for t-lowest functions queries in $O(Q(n, n/r) + n/r + t)$ time and deletions in $O((r/n)(S(n,1) + T(n, n/r) \log r))$ amortized time.*

5.3. A general dynamic k-NN data structure

In Section 5.3 and 5.4, we generalize the dynamic k-NN data structure from Section 5.1 to support other types of distance functions. First, in this section, we analyse the space usage and running times of the data structure for planes, as this is somewhat easier to follow. Next, in Section 5.4, we analyse these for the general data structure.

Lemma 8. *There is a fully dynamic data structure of size $O(n)$ that stores a set of n point sites and allows for planar k-nearest neighbors queries using the Euclidean distance in $O(\log^2 n / \log\log n + k)$ time, insertions in $O(\log^{3+\varepsilon} n / \log\log n)$ amortized time, and deletions in $O(\log^{5+\varepsilon} n / \log\log n)$ amortized time.*

Proof. To generalize the dynamic k-NN data structure from Section 5.1 to other types of distance functions, we replace the deletion-only data structure used by Chan [14] as $\mathcal{D}_0$ data structure by the data structure of Section 5.2. In our approach, we use the same data structure for both small and large conflict lists. Queries and updates are performed as before (see Sections 2.2 and 5.1). This results in a dynamic k-lowest functions data structure that can be used for any type of functions for which we can construct k-shallow cuttings. As our $\mathcal{D}_0$ data structure only uses k-shallow cuttings, our approach is also somewhat simpler than Chan's, albeit at a slight increase in space usage. This problem can be solved by using the space saving idea discussed in Section 5.2.

Query time. Our bootstrapping data structure $\mathcal{D}_0$ has query time $Q_0(n') = O(\log r + n'/r)$ and deletion time $D_0(n') = O(r \log n')$. Because we query the shallow cutting at level j_k, the size of each conflict list we query is $O(k)$. By again using our scheme to find the k-nearest neighbors over the substructures simultaneously, the query time becomes:

$$\begin{aligned} Q(n) &= O([\log n + Q_0(O(k))] \log_b n + k) \\ &= O((\log n + (\log r + k/r)) \log_b n + k). \end{aligned}$$

If we set $r = \log n$ (and $b = \log^{\varepsilon} n$ just like Chan) we get

$$\begin{aligned} Q(n) &= O((\log n + \log\log n + k/\log n) \log n / \log\log n + k) \\ &= O(\log^2 n / \log\log n + k). \end{aligned}$$

Thus using our $\mathcal{D}_0$ data structure does not affect the query time.

Update time. In the following we analyze the update time of the data structure in more detail. The update time given by Chan is:

$$U(n) = O(b^{O(1)} \log^6 n + \max_{m \le n} D_0(m) \cdot b^{O(1)} \log^5 n).$$

Note that this update time is based on the old approach of Chan, where a plane can occur $O(b \log^2 n)$ times in a tower. To give a more detailed analysis, we first study the insertion time and then the deletion time of $\mathcal{D}$.

Lemma 3 states that insertion time is given by $I(n) = O(b \log_b n \cdot (P(n)/n))$, where $P(n)$ is the preprocessing time of $\mathcal{D}$. Our preprocessing time increases w.r.t. the original data structure, since after building the hierarchy of shallow cuttings for a tower, we additionally need to build the structure $\mathcal{D}_0$ on each conflict list. As before, building the shallow cuttings takes $O(n \log n)$ time [12]. Next, we analyze the time to build all data structures $\mathcal{D}_0$. The cutting at level j in the hierarchy consists of $O(n/k_j)$ prisms, and the size each conflict list in the cutting is $O(k_j)$. Furthermore, there are $\log(n/k_0)$ cuttings in the hierarchy, as the lowest level cutting is a k_0-shallow cutting. Let α be a constant bounding the size of the conflict lists. Using that $P_0(n') = O(n' \log n')$, we find the following running time for building all $\mathcal{D}_0$ data structures of a single tower:

$$\begin{aligned} \sum_{j=0}^{\log \frac{n}{k_0}} O\left(\frac{n}{k_j}\right) \cdot P_0(\alpha k_j) &= \sum_{j=0}^{\log \frac{n}{k_0}} O\left(\frac{n}{k_j}\right) \cdot O\left(\alpha k_j \log(\alpha k_j)\right) \\ &= \sum_{j=0}^{\log \frac{n}{k_0}} O\left(n \log(\alpha k_j)\right) \\ &= O\left(\log(n/k_0) \cdot n \log\left(\alpha k_0 2^{\log(n/k_0)}\right)\right) \\ &= O(n \log^2 n). \end{aligned}$$

The preprocessing time thus adheres to the recurrence relation $P(n) \le P(n/b) + O(n \log^2 n)$, which solves to $P(n) = O(n \log^2 n)$. It follows that $I(n) = O(b \log_b n \cdot (P(n)/n)) = O(b \log^2 n \log_b n) = O(\log^{3+\varepsilon} n / \log\log n)$. Note that the improvement of building all shallow cuttings in a tower in $O(n \log n)$ time does not improve the insertion time to $O(\log^2 n)$ as in the 1-NN data structure, because building the $\mathcal{D}_0$ data structures is the dominant term.

When deleting a plane h that is live in tower $\mathcal{T}^{(i)}$, we remove the plane from all $\mathcal{D}_0$ of $\mathcal{T}^{(i)}$ with $h \in \mathcal{D}_0$. There are at most $O(b \log n)$ such data structures $\mathcal{D}_0$, because a plane can occur at most $O(b \log n)$ times in a tower. On the other hand, by Lemma 3, deleting a plane causes amortized $O(b \log n \log_b n)$ reinsertions. Each reinserted plane is also removed from the structures $\mathcal{D}_0$ of a single tower. This results in amortized $O(b \log_b n \log n)$ planes that are again removed from $O(b \log n)$ structures. We can thus formulate the deletion time as $D(n) = O(b \log n \log_b n \cdot (b \log n \cdot D_0(n) + I(n)))$. Plugging in $D_0(n) = O(\log^2 n)$ and $I(n) = O(b \log^2 n \log_b n)$, we find $D(n) = O(b^2 \log^4 n \log_b n + b^2 \log^3 n \log_b^2 n) = O(b^{O(1)} \log^4 n \log_b n) = O(\log^{5+\varepsilon} n / \log\log n)$.

Space usage. The space usage of a $\mathcal{D}_0$ data structure storing n' planes is $O(n' \log r)$. The space usage of $\mathcal{D}$ is thus $S(n) = \sum_{j=0}^{\log \frac{n}{k_0}} \frac{n}{k_j} \cdot O(k_j \log r) = O(n \log n \log\log n)$. Using the space reduction idea mentioned in Section 5.2, the space of a $\mathcal{D}_0$ data structure is only $O(n')$ for $n' \ge r^{1/\varepsilon}$. The space usage of the entire data structure is then

$$S(n) = \sum_{j=0}^{j'} \frac{n}{k_j} \cdot O(k_j \log r) + \sum_{j=j'}^{\log \frac{n}{k_0}} \frac{n}{k_j} \cdot O(k_j) = O(j' n \log r + n \log n),$$

where j' is such that $k_{j'} = r^{1/\varepsilon}$. Using that $r = \log n$ we find $S(n) = O(n \log n)$. □

5.4. Analysis of the running times for general shallow cuttings

We can use a similar scheme in a more general setting. In this section, we describe the running times of the data structure w.r.t. any algorithm that can construct vertical k-shallow cuttings. We assume that $S(n,k) = \frac{1}{k} f(n)$, for some function $f(n)$. This implies that the size of a k-shallow cutting including its conflict lists is $S(n,k) \cdot O(k) = O(S(n,1))$.

We will prove the following theorem, which is similar to Lemma 2, but for k-lowest functions queries.

Theorem 4. *Given an algorithm that constructs a k-shallow cutting of size $S(n,k)$ on n functions in $T(n,k)$ time, such that the prism containing a query point can be located in $Q(n,k)$ time, we can construct a data structure that dynamically maintains a set of at most n functions F for k-NN queries. The size of the data structure is $O(S(n,1)^2/n \cdot \log n \log\log n)$, and operations are performed in time:*

Query: $O(Q(n,1)\log n/\log\log n + k)$
Insertion: $O((S(n,1)T(n,1)/n^2)\log^{2+\varepsilon} n)^*$
Deletion: $O((T(n,1)/\log\log n + T(n, n/\log n)) \cdot S(n,1)^2 \log^{4+\varepsilon} n/n^3)^*$
**amortized*

Proof. As in Section 5.3, we apply the $\mathcal{D}_0$ data structure from Section 5.2 to the data structure described in Sections 2.2 and 5.1. Instead of Lemma 6, we apply the general data structure of Lemma 7.

Query time. Using the query time of the general $\mathcal{D}_0$ data structure of Lemma 7, we have $Q_0(n') = O(Q(n', n'/r) + n'/r)$. Just as in the Euclidean case, we use $r = \log n$ and $b = \log^{\varepsilon} n$. The query time then becomes

$$\begin{aligned} Q(n) &= O([Q(n,1) + Q_0(O(k))]\log_b n + k) \\ &= O([Q(n,1) + Q(k, k/r) + k/r]\log_b n + k) \\ &= O(Q(n,1)\log n/\log\log n + k/\log\log n + k) \\ &= O(Q(n,1)\log n/\log\log n + k). \end{aligned}$$

Note that $Q(k, k/r) = O(Q(n,1))$, as $Q(n,k)$ is non-decreasing in n. Also, $O(\log_b n/r) = O((\log n/\log\log n)/\log n) = O(1/\log\log n)$.

Update time. To determine the insertion time, we start by analyzing the preprocessing time of the data structure. In the preprocessing, we first construct the $O(\log n)$ shallow cuttings of a single tower in $O((T(n,1)\log n)$ time. Then, we again build all structures $\mathcal{D}_0$ on (the live part of) the resulting conflict lists. A k_j-shallow cutting consists of $S(n,k_j)$ prisms, each with conflict list of size $\leq \alpha k_j$. The time to build the $\mathcal{D}_0$ data structures is thus:

$$\begin{aligned} \sum_{j=0}^{\log \frac{n}{k_0}} S(n,k_j) \cdot P_0(\alpha k_j) &= \sum_{j=0}^{\log \frac{n}{k_0}} S(n,k_j) \cdot O(T(\alpha k_j, \alpha k_j/r)\log r) \\ &= O\left(S(n,1)\log r \cdot \sum_{j=0}^{\log \frac{n}{k_0}} \frac{1}{k_j} T(\alpha k_j, \alpha k_j/r) \right) \\ &= O\left(S(n,1)\log r \cdot \sum_{j=0}^{\log \frac{n}{k_0}} \frac{1}{k_j} T(\alpha k_j, 1) \right) \\ &= O\left((S(n,1)T(n,1)/n)\log n \log r \right) \\ &= O\left((S(n,1)T(n,1)/n)\log n \log\log n \right). \end{aligned}$$

Here we used that $T(\alpha k_j, 1) \geq \alpha k_j$ and $T(n,k)$ is non-decreasing in n, which implies that $T(\alpha k_j, 1)/k_j \leq T(n,1)/n$. It follows that

$$P(n) = O((S(n,1)T(n,1)/n)\log n \log\log n),$$

and by Lemma 3:

$$I(n) = O(b \log_b n \cdot P(n)/n) = O((S(n,1)T(n,1)/n^2)\log^{2+\varepsilon} n).$$

Note that the insertion time has increased by an $O(\log\log n)$ factor compared to the Euclidean case, because the preprocessing time of the $\mathcal{D}_0$ data structure now depends on $\log r$.

The deletion time is still determined by the number of reinsertions caused by a deletion. According to Lemma 3, the amortized number of reinsertions caused by a deletion is $O((S(n,1)/n)b\log n\log_b n)$. Each reinsertion in turn causes $O((S(n,1)/n)b\log n)$ updates on the $\mathcal{D}_0$ data structures. By Lemma 7 we have that $D_0(n) = O((r/n)(S(n,1) + T(n,n/r)\log r))$. So, the deletion time is given by:

$$\begin{aligned}
D(n) &= O((S(n,1)/n)b\log n\log_b n(D_0(n)(S(n,1)/n)b\log n + I(n))) \\
&= O((S(n,1)/n)\log^{2+\varepsilon} n/\log\log n(D_0(n)(S(n,1)/n)\log^{1+\varepsilon} n + I(n))) \\
&= O((S(n,1)^2 T(n,n/\log n)/n^3)\log^{4+\varepsilon} n \\
&\qquad + S(n,1)^2 T(n,1)\log^{4+\varepsilon} n/n^3\log\log n) \\
&= O((T(n,1)/\log\log n + T(n,n/\log n))\cdot S(n,1)^2\log^{4+\varepsilon} n/n^3).
\end{aligned}$$

Here, we used that $S(n,1) \leq T(n,1)$.

Space usage. The space usage of the data structure is

$$S(n) = \sum_{j=0}^{\log \frac{n}{k_0}} S(n,k_j)\cdot O(S(\alpha k_j,1)\log r) = O(S(n,1)^2/n\cdot\log n\log\log n). \quad \square$$

5.5. Applications

Points in the plane. For general distance functions of constant description complexity, for which the lower envelope of any t functions has $O(t)$ faces, edges, and vertices, Liu [16] recently presented an algorithm to compute k-shallow cuttings of linear size. This improves the shallow cuttings used by Kaplan et al. [13] in the dynamic 1-NN data structure. Liu [16] also shows that in the static setting we can use a circular range query data structure [22] to improve space usage. Like in the Euclidean case, we can use this auxiliary data structure to improve the space usage of the $\mathcal{D}_0$ data structure to linear for large n.

Corollary 3. *There is a fully dynamic data structure of size $O(n\log n)$ that stores a set of n point sites and allows for k-nearest neighbors queries using a fixed constant description complexity distance function in $O(\log^2 n/\log\log n + k)$ time. A site can be inserted in $O(\log^{3+\varepsilon} n/\log\log n)$ expected amortized time and deleted in $O(\log^{5+\varepsilon} n/\log\log n)$ expected amortized time.*

Proof. The k-shallow cutting algorithm by Liu [16] produces shallow cuttings of size $S(n,k) = O(n/k)$ in $T(n,k) = O(n\log^3 n\lambda_{s+2}(\log n))$ expected time. A prism containing a query point can be located using a point location data structure in $Q(n,k) = O(\log n)$ time. We can then apply Theorem 4 directly to obtain a k-NN data structure.

In the recently published journal version of the paper [5], Liu improves upon these results even further. The improved algorithm produces shallow cuttings of size $S(n,k) = O(n/k)$ in $T(n,k) = O(n\log n)$ expected time. To be precise, the construction algorithm of Liu [5] can build *all* shallow cuttings in a hierarchy in $O(n\log n)$ expected time. This means that the preprocessing and deletion time of the bootstrapping data structure (Lemma 7) are improved to $O(n\log n)$ and $O(r\log n)$, respectively. Applying Theorem 4 to these shallow cuttings, and adjusting the update times to account for this improvement in the bootstrapping data structure, proves the result. $\square$

Points in a simple polygon. Agarwal et al. [9] show how to use a data structure based on the structure by Chan to perform geodesic 1-NN queries. We adapt this data structure by again storing the conflict lists in the data structure of Section 5.2. The data structure by Agarwal et al. partitions the polygon recursively into two subpolygons, as described in Section 4.2. It then uses the Chan data structure to find the nearest neighbor in the corresponding subpolygon at each of the $O(\log m)$ levels of the decomposition. Note that in the k-NN case, we can again use our scheme of Section 4 to find the k-NN spread over $O(\log m)$ data structures simultaneously.

The shallow-cuttings that are described in [9] use pseudo-prisms that are no longer of constant complexity. We refer to the paper for the precise definition. For our purposes, we need only the property that we can still find the prism containing a query point efficiently.

Corollary 4. *Let $\mathcal{P}$ be a polygon with m vertices. There is a fully dynamic data structure of size $O(n\log^5 n\log m\log\log n + m)$ that stores a set of n point sites in $\mathcal{P}$ and allows for geodesic k-nearest neighbors queries in $O(\log^2 n\log^2 m/\log\log n + k\log m)$ time, insertions in $O(\log^{8+\varepsilon} n\log m + \log^{7+\varepsilon} n\log^3 m)$ expected amortized time, and deletions in $O((\log^{12+\varepsilon} n\log m + \log^{11+\varepsilon} n\log^3 m)/\log\log n)$ expected amortized time.*

Proof. Theorem 23 of [9] states that a k-shallow cutting $\Lambda_k(F)$ of F of topological complexity $S(n,k) = O((n/k)\log^2 n)$ can be computed in expected time $O((n/k)\log^3 n(\log n + \log^2 m) + n\log^2 m + n\log^3 n \log m)$. Our improvement for the static k-NN data structure of Section 4 also leads to a slight improvement in the construction time of a shallow cutting. In particular, we can build a shallow cutting in $T(n,k) = O((n/k)\log^3 n(\log n + \log^2 m) + n\log^2 m + n\log^2 n \log(n+m))$ expected time. Locating the correct prism takes $Q(n,k) = O(\log n \log m)$ time. By applying Theorem 4 we find for $\mathcal{D}$:

$$Q(n) = O(\log^2 n \log m / \log\log n + k)$$
$$I(n) = O(\log^{8+\varepsilon} n + \log^{7+\varepsilon} n \log^2 m)$$
$$D(n) = O((\log^{12+\varepsilon} n + \log^{11+\varepsilon} n \log^2 m)/\log\log n)$$
$$S(n) = O(n \log^5 n \log\log n)$$

Using this data structure at each of the $O(\log m)$ levels of the decomposition and applying the scheme from Section 4 to achieve $O(\text{polylog}(n,m) + k)$ query time proves the result. □

The following theorem summarizes the results of this section, together with the results from Section 5.3.

Theorem 5. *There is a fully dynamic data structure of size $S(n)$ that stores a set of n sites in the plane and allows for k-nearest neighbors queries in $Q(n)$ time, insertions in $I(n)$ amortized time, and deletions in $D(n)$ amortized time. For the general and geodesic case, the update times are expected running times. $\mathcal{P}$ is a simple polygon with m vertices.*

	Euclidean	*General*	*Geodesic in $\mathcal{P}$*
$Q(n)$	$O\left(\frac{\log^2 n}{\log\log n} + k\right)$	$O\left(\frac{\log^2 n}{\log\log n} + k\right)$	$O\left(\frac{\log^2 n \log^2 m}{\log\log n} + k\log m\right)$
$I(n)$	$O\left(\frac{\log^{3+\varepsilon} n}{\log\log n}\right)$	$O\left(\frac{\log^{3+\varepsilon} n}{\log\log n}\right)$	$O(\log^{8+\varepsilon} n \log m + \log^{7+\varepsilon} n \log^3 m)$
$D(n)$	$O\left(\frac{\log^{5+\varepsilon} n}{\log\log n}\right)$	$O\left(\frac{\log^{5+\varepsilon} n}{\log\log n}\right)$	$O\left(\frac{\log^{12+\varepsilon} n \log m}{\log\log n} + \frac{\log^{11+\varepsilon} n \log^3 m}{\log\log n}\right)$
$S(n)$	$O(n\log n)$	$O(n\log n)$	$O(n\log^5 n \log m \log\log n + m)$

Declaration of competing interest

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

Data availability

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

References

[1] A. Andoni, P. Indyk, Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, Commun. ACM 51 (1) (2008) 117–122.
[2] T.M. Chan, Random sampling, halfspace range reporting, and construction of $\leq k$-levels in three dimensions, SIAM J. Comput. 30 (2) (2000) 561–575.
[3] T.M. Chan, K. Tsakalidis, Optimal deterministic algorithms for 2-d and 3-d shallow cuttings, Discrete Comput. Geom. 56 (4) (2016) 866–881.
[4] D.-T. Lee, On k-nearest neighbor Voronoi diagrams in the plane, IEEE Trans. Comput. C-31 (6) (1982) 478–487.
[5] C. Liu, Nearly optimal planar k nearest neighbors queries under general distance functions, SIAM J. Comput. 51 (3) (2022) 723–765.
[6] T. Cover, P. Hart, Nearest neighbor pattern classification, IEEE Trans. Inf. Theory 13 (1) (1967) 21–27.
[7] C. Liu, D.T. Lee, Higher-order geodesic Voronoi diagrams in a polygonal domain with holes, in: Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, SIAM, 2013, pp. 1633–1645.
[8] P. Afshani, T.M. Chan, Optimal halfspace range reporting in three dimensions, in: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, SIAM, 2009, pp. 180–186.
[9] P.K. Agarwal, L. Arge, F. Staals, Improved dynamic geodesic nearest neighbor searching in a simple polygon, in: 34th International Symposium on Computational Geometry, SoCG, in: LIPIcs, vol. 99, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 4:1–4:14.
[10] M.H. Overmars, The Design of Dynamic Data Structures, Lecture Notes in Computer Science, vol. 156, Springer, 1983.
[11] T.M. Chan, A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries, J. ACM 57 (3) (2010) 16:1–16:15.
[12] T.M. Chan, Dynamic geometric data structures via shallow cuttings, in: 35th International Symposium on Computational Geometry, SoCG, in: LIPIcs, vol. 129, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 24:1–24:13.
[13] H. Kaplan, W. Mulzer, L. Roditty, P. Seiferth, M. Sharir, Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications, Discrete Comput. Geom. 64 (3) (2020) 838–904.
[14] T.M. Chan, Three problems about dynamic convex hulls, Int. J. Comput. Geom. Appl. 22 (4) (2012) 341–364.
[15] T.M. Chan, 2021, personal communication.
[16] C. Liu, Nearly optimal planar k nearest neighbors queries under general distance functions, in: Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, SIAM, 2020, pp. 2842–2859.
[17] J. Matoušek, Reporting points in halfspaces, Comput. Geom. Theory Appl. 2 (3) (1992) 169–186.

[18] G.N. Frederickson, An optimal algorithm for selection in a min-heap, Inf. Comput. 104 (2) (1993) 197–214.
[19] L.J. Guibas, J. Hershberger, Optimal shortest path queries in a simple polygon, J. Comput. Syst. Sci. 39 (2) (1989) 126–152.
[20] E. Oh, H. Ahn, Voronoi diagrams for a moderate-sized point-set in a simple polygon, Discrete Comput. Geom. 63 (2) (2020) 418–454.
[21] P.K. Agarwal, J. Matousek, Dynamic half-space range reporting and its applications, Algorithmica 13 (4) (1995) 325–345.
[22] P.K. Agarwal, J. Matousek, M. Sharir, On range searching with semialgebraic sets. II, SIAM J. Comput. 42 (6) (2013) 2039–2062.