Dynamic Data Structures for k-Nearest Neighbor Queries

Sarita de Berg
Department of Information and Computing Sciences, Utrecht University, The Netherlands

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

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 $O(f(n)+k)$ 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 a $O(tk)$ 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 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.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases data structure, simple polygon, geodesic distance, nearest neighbor searching

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2021.14

Related Version *Full Version*: `https://arxiv.org/abs/2109.11854`

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 [4, 5, 10, 18, 19]. We study the dynamic version of the 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 in a city center may want to find the k closest restaurants that currently have seats available, but since he or she cannot walk through walls, this should be reflected by the distances. This introduces complications, as a shortest path in a polygon with m vertices may have complexity $\Theta(m)$. We wish to limit the resulting dependency on m in the space and time bounds as much as possible. In particular, we wish to avoid spending $\Omega(m)$ time every time the availability of the seats in a restaurant changes (which may cause an insertion or deletion of a site in S).

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 a query point q is predicted based on the labels of the k sites nearest to q [11]. When this label turns out to be sufficiently accurate, it is customary to then extend the data set by adding q to S. This naturally gives an interest in insertion-only data structures that can efficiently answer k-NN queries.

The static problem. If the set of sites is static, and k is known a-priori, one option is to build the (geodesic) k^{th}-order Voronoi diagram of S [20]. This yields very fast ($O(\log(n+m)+k)$) query times. However it is costly in space, as even in a simple polygon the diagram has size $O(k(n-k)+km)$. For 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 [1, 10]. Very recently, Liu [19] showed how to achieve the same query time for general constant-complexity distance functions for sites in $\mathbb{R}^2$, using $O(n \log\log n)$ space and roughly $O(n \log^4 n)$ expected preprocessing time (the exact bound depends on the algebraic degree of the functions). In case the domain is a simple polygon $\mathcal{P}$, 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 dynamic 1-NN structure of Agarwal et al. [2]. It uses $O(n \log^3 n \log m + m)$ space, and answers queries in $O(k \operatorname{polylog}(n+m))$ 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 [23] to turn a static k-NN searching data structure into an insertion-only data structure. 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 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. This yields an $O(k \log n)$ term in the query time. We will present a general technique that allows us to avoid this $O(\log n)$ factor.

Fully dynamic data structures. In case we wish to support both insertions and deletions, the problem becomes more complicated, and the existing solutions much more involved. When we need to report only one nearest neighbor (i.e. 1-NN searching) in the plane, efficient fully dynamic data structures exist [6, 8, 16]. Actually, all these data structures are variants of the same data structure by Chan [6]. For the Euclidean distance, the current best result uses linear space, and achieves $O(\log^2 n)$ worst-case query time, $O(\log^2 n)$ insertion time, and $O(\log^4 n)$ deletion time [8]. These results are deterministic, and the update times are amortized. The variant by Kaplan et al. [16] achieves similar results for general distance functions. These data structures can also be used to answer k-NN queries, but when used in this way essentially suffer from the same problem as in the insertion-only case. That is, we get a query time of $O(\log^2 n + k \log n)$ time [6].

Chan argues that the above data structure for Euclidean 1-NN searching can be extended to answer k-NN queries in $O(\log^2 n / \log\log n + k)$ time, while retaining polylogarithmic updates [7]. 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 [9][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.

Very recently, Liu [19] stated that one can obtain $O(\log^2 n+k)$ query time while supporting $O(\text{polylog}\, n)$ expected amortized updates also for general distance functions by using the data structure of Kaplan et al. [16]. However, it is unclear why that would be the case, as all details are missing. Using the Kaplan et al. data structure as is yields an $O(k \log n)$ term in the query time as with Chan's original version [6]. If the idea is to also apply the ideas from Chan's later paper [7] the same issue of choosing the k_i's appears. Similarly, extending the geodesic 1-NN data structure [2] to k-NN queries yields $O(k\,\text{polylog}(n+m))$ query time.

Organization and Results. We develop dynamic data structures for k-NN queries whose query time 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. 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 [7] 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. We do get an $O(\log m)$ factor in the query time, as computing the distance between two sites 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 $O((\log(n + m) \ + k) \log m)$ time queries. In turn, this then leads to a data structure supporting efficient 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 [7, 19]. The amortized update times remain polylogarithmic. We obtain the same query time and similar update times for more general distance functions.

For the geodesic case there is one final hurdle. Chan's algorithm uses partition-tree based "slow" dynamic k-NN data structure of linear size in its subroutines. Liu uses a similar trick after linearizing the distance functions into $\mathbb{R}^c$, for some constant c [19]. Unfortunately, these ideas are not applicable in the geodesic setting, as it is unknown if an appropriate partition tree can be built, and the dimension after linearization would depend on m. Instead, we design a simple, shallow-cutting based, alternative "slow" dynamic (geodesic) k-NN data structure. This way, we obtain an efficient (i.e. $O(\text{polylog}(n + m))$ expected updates, $O(\log^2 n \log^2 m + k \log m)$ queries) fully dynamic k-NN data structure. Omitted proofs are in the full version of this paper [12].

[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 [5] one can then also answer k-NN queries.

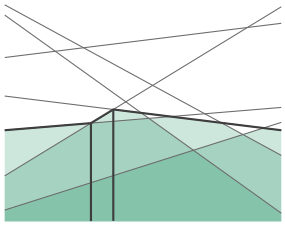

Figure 1 A 2-shallow cutting of a set of lines F in $\mathbb{R}^2$ consisting of 3 prisms. The at most k-level $L_{\leq k}(F)$ is shown in green for $k = 0, 1, 2$.

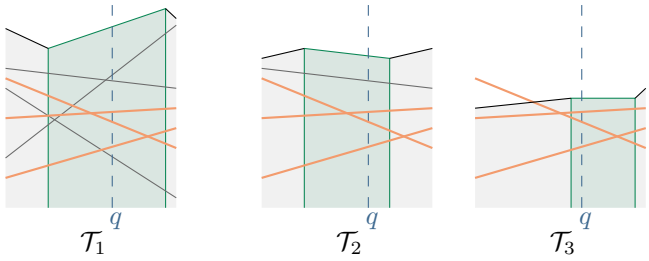

Figure 2 Example of the dynamic 1-NN data structure. Only one shallow cutting (Λ_{k_j}) is shown for each tower. The orange planes in $\mathcal{T}_1$ and T_2 are pruned when building $\Lambda_{k_{j-1}}$, but are not removed from the conflict lists in Λ_{k_j}. When querying for the k-NN, the green prisms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

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 [21]. 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, from the sides by vertical (with respect to the z-direction) planes, 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 Figure 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 [19]. In general, let $T(n, k)$ be the time to construct a k-shallow cutting of 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 $S(n, k) = \frac{n}{k} f(n)$, for some function $f(n)$.

2.2 A dynamic nearest neighbor data structure

We briefly discuss the main ideas used in the existing dynamic 1-NN data structures [6, 8, 16], as these also form a key component in our fully dynamic k-NN data structures. For ease of exposition, we describe the data structure when F is a set of linear functions (planes). To ensure the analysis is correct for our definition of n (the current number of sites 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 [6].

The data structure consists of $t = O(\log_b n)$ "towers" $\mathcal{T}^{(1)}, .., \mathcal{T}^{(t)}$. Each tower $\mathcal{T}^{(i)}$ consists of a hierarchy of shallow cuttings that is built on a subset of planes $F^{(i)} \subseteq F$. For $\mathcal{T}^{(1)}$ we have $F^{(1)} = F$, and a sequence 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 all cuttings computed 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 $\Lambda_{k_j}(F_j)$. 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 $O(k_0)$. By $F^{(1)}_{\text{live}}$ we denote the set of planes that have not been pruned during this process. We then set $F^{(i+1)} = F^{(i)} \setminus F^{(i)}_{\text{live}}$, and recursively build $\mathcal{T}^{(i+1)}$ on the functions in $F^{(i+1)}$. This partitions F into sets $F^{(1)}_{\text{live}}, .., F^{(t)}_{\text{live}}$. Chan [8] recently achieved an overall construction time of $O(n \log n)$, by using information of previously computed cuttings to efficiently build the cuttings later in the sequence. Kaplan et al. [16, Lemma 7.1] prove that for any $\zeta \in (0, 1)$ choosing $c \geq \frac{\gamma}{\zeta}$, for a sufficiently large (but constant) γ, ensures 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. A plane then occurs $O(b \log n)$ times in a tower.

Updates. When updates take place, planes can move from a set $F^{(i)}_{\text{live}}$ to some $F^{(i')}_{\text{live}}$, but the live sets remain a partition of F. To insert a plane f into F, we create a new tower containing only f. When $|F^{(i+1)} \cup ... \cup F^{(t)}|$ reaches $3/4 \cdot |F^{(i)}|$ we rebuild the towers $\mathcal{T}^{(i)}, .., \mathcal{T}^{(t)}$. Such a rebuild occurs only after $\Omega(|F^{(i)}|)$ insertions, so the amortized insertion time is $O(\log^2 n)$.

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 (i.e. d_∇ becomes too large), we *purge* the prism. 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 $F^{(i')}_{\text{live}}$. Chan [8] shows that each increment of d_∇ causes amortized $O(1)$ reinsertions. This gives an amortized deletion time of $O(\log^4 n)$.

Queries. We can answer k-NN queries in $O(\log^2 n + k \log_b n)$ time, and thus 1-NN queries in $O(\log^2 n)$ time, as follows. For each tower we consider the prism containing q in the shallow cutting at level $j_k := \lceil \log(Ck/n) \rceil$, for some large enough constant C. Each such prism has a conflict list of size $O(k)$, and thus we can find the k-lowest *live* planes in each conflict list in $O(k)$ time. Chan [6] proves that considering only these planes is sufficient.

Liu [19] recently claimed the data structure, in particular the version of Kaplan et al. [16], supports k-NN queries in $O(\log^2 n + k)$ time. However, we see an issue with this approach. 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 Figure 2 for an illustration. As there are $O(\log_b n)$ towers, this yields an $O(k \log_b n)$ term in the query time.

General distance functions. Kaplan et al. [16] showed that this data structure is applicable for any set of functions F for which we can compute small k-shallow cuttings. The following lemma summarizes the properties of the data structure in this setting.

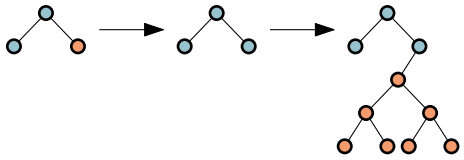

Figure 3 Example of expansion. Blue elements are included in a clan, orange elements are not. The expansion (building the next subheap) occurs when all elements have been included in a clan.

▶ **Lemma 1.** *Let $b \geq 2$ be any fixed value, and $S(n,k)$ be the size of a k-shallow cutting of F. 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 the prisms containing q of the shallow cuttings at level $j_k := \lceil \log(Ck/n) \rceil$, for some large enough constant C.*

3 Querying multiple k-NN data structures simultaneously

We introduce a method to find the k-nearest neighbors of a query point q among t (disjoint) k'-NN data structures together storing a set of sites S. 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 $O(Q(n)t + kt)$ time. Our method allows us to find the k-NN over all these data structures in $O(Q(n)t + k)$ time instead.

Query algorithm. We use the heap selection algorithm of Frederickson [13] 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 by forming groups of elements, called *clans*, in the original heap. Representatives of these clans are then added to another heap, and smaller clans are created from larger clans and organised in heaps recursively. For our purposes, we need to consider (only) how clans are formed in the original heap, because we do not construct the entire heap beforehand. Instead, the heap is expanded during the query only when necessary. See Figure 3 for an example. Note that any (non-root) element of the heap will only be included in a clan by the Frederickson algorithm after its parent has been included in a clan.

The heap H, on which we call the heap selection algorithm, 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_i is the set of sites stored in the i-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 that we obtain this way, see Figure 4. We can now compute the k sites closest to q by finding the $|H_0| + k$ smallest elements in H and reporting 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, any site $s \in S$ will appear 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 k'-NN data structure on S_j. The subheap $H_i(S_j)$ at level $i > 1$ is built only

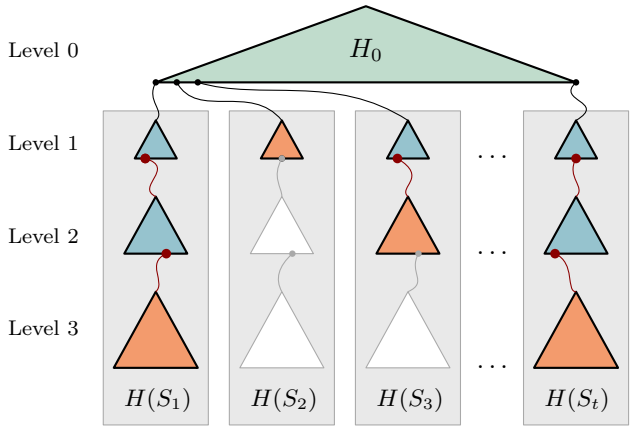

Figure 4 The heap constructed for a k-NN query. Subheaps of which all elements have been included in a clan are *blue*. Subheaps of which not all elements have been included are *orange*. The *white* subheaps have not been built so far, as not all elements of their predecessor are in a clan yet.

after the last element e of $H_{i-1}(S_j)$ is included in a clan. 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 Figure 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. This ensures that 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 [13]. Here, we analyse 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 $k_1 = Q(n)$. In total these queries take $O((Q(n) + k_1)t) = O(Q(n)t)$ time. Retrieving the next k_i elements to build $H_i(S_j)$ for $i > 1$ requires a single query and thus takes $O(Q(n) + k_i)$ time. To bound the time used to build all heaps at level greater than 1, we use the following lemma.

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

To pay for building $H_i(S_j)$, we charge $O(1)$ to each element of $H_{i-1}(S_j)$. Because we choose $k_1 = Q(n)$, Lemma 2 implies that $|H_{i-1}(S_j)| = \Omega(Q(n))$, 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 all elements of $H_{i-1}(S_j)$ have been included in a clan. Thus, we only charge elements of heaps of which all elements have been included in a clan (shown blue in Figure 4). In total, $O(k)$ elements (not in H_0) are included in a clan, 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 $O(1)$ to only $O(k)$ sites.

▶ **Lemma 3.** *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 $O(Q(n_i) + k')$ time. There is a k-NN data structure on $\bigcup_i S_i$ that supports queries in $O(Q(n)t + k)$ 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.*

Throughout this section, we used the standard assumption that for any two points p, q their distance $d(p, q)$ can be computed in constant time. When evaluating $d(p, q)$ takes T time, our technique achieves a query time of $O(Q(n)t + kT)$ by setting $k_1 = Q(n)/T$ and charging $O(T)$ to each site of $H_{i-1}(S_j)$ to pay for building $H_i(S_j)$.

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 [23]. 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 [23]. 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 Lemma 3 directly solves this problem, and we thus obtain the following result.

▶ **Theorem 4.** *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 a k-NN data structure on S of size $O(C(n))$ that supports queries in $O(Q(n)\log n+k)$ time, and insertions in $O((P(n)/n)\log n)$ 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 [1, 10]. Hence:

▶ **Corollary 5.** *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 [19], using $O(n\log\log n)$ space and expected $O(\text{polylog}\, n)$ insertion time.

4.2 Points in a simple polygon

Next, we consider k-NN queries on a set S of n point sites inside a simple polygon $\mathcal{P}$ with m vertices. For any two points p and q the (geodesic) distance $d(p,q)$ is defined as the length of the shortest path $\pi(p,q)$ between p and q fully contained within $\mathcal{P}$. Using $O(m)$ space and preprocessing time, we can store $\mathcal{P}$ so that $d(p,q)$ can be computed in $O(\log m)$ time [15].

To apply Theorem 4, we need a static data structure for geodesic k-NN queries. As we sketch below, we can obtain such a data structure by combining the approach of Chan [5] and Agarwal et al. [2]. However, building this data structure takes $O(nm)$ time. We show that using more ideas from Agarwal et al. [2], together with our algorithm from Section 3, we can obtain a static k-NN data structure that can also be built efficiently. This in turn leads to an efficient insertion-only data structure.

A static data structure. The initial 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

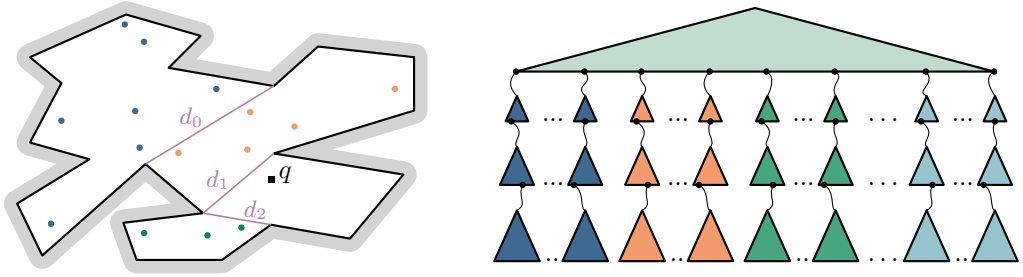

Figure 5 A partial decomposition of $\mathcal{P}$ and the corresponding heap used in a k-NN query for q.

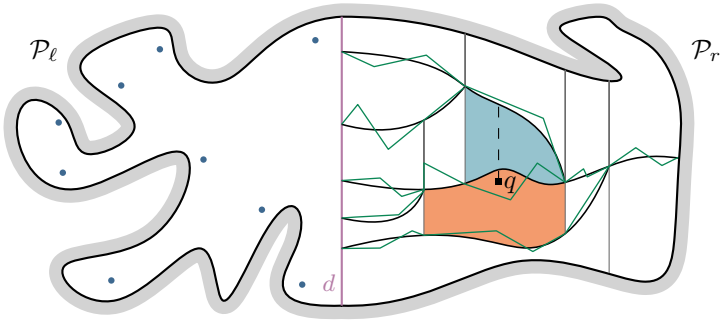

Figure 6 Approximation (in green) of the vertical decomposition of the Voronoi diagram of S_ℓ in $\mathcal{P}_r$. To find the trapezoid containing q, we consider both colored trapezoids.

corresponding (pseudo-)prisms. 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 [5, 22]. This allows us to answer k-NN queries in the same time. The crux in this approach is in how to compute the conflict lists. We can naively compute these in $O(mn)$ time by explicitly constructing the geodesic distance function for each site [14], and intersecting it with each of the $O(n)$ pseudo-prisms. It is unclear how to improve on this bound.

▶ **Theorem 6.** *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 Sketch. To circumvent the issue above, we recursively partition the polygon $\mathcal{P}$ into two subpolygons $\mathcal{P}_r$ and $\mathcal{P}_\ell$ of roughly the same size by a diagonal d [2]. We denote by S_r and S_ℓ the sites in $\mathcal{P}_r$ and $\mathcal{P}_\ell$, respectively. Theorem 22 of [2] provides us with a data structure that can find the k-NN among sites in S_ℓ for a query point in P_r. This is essentially the data structure that was described above. However, because the Voronoi diagram of sites in S_ℓ restricted to $\mathcal{P}_r$ is a Hamiltonian abstract Voronoi diagram [17], we can efficiently compute the conflict lists by only considering the functions intersecting the corners of each prism. We improve the query time of this data structure to $O(\log(n+m)+k\log m)$ by incorporating the idea of Oh and Ahn [22] to approximate a geodesic Voronoi diagram by a polygonal subdivision, see Figure 6. Storing the points of both S_ℓ and S_r at each of the $O(\log m)$ levels of the decomposition in this data structure, and using our technique from Section 3 to query the levels simultaneously, see Figure 5, results in the stated query time. ◀

▶ **Corollary 7.** *Let $\mathcal{P}$ be a simple polygon with m vertices. There is a 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 consider k-NN queries while supporting both insertions and deletions, building on the results of Chan [7]. We first fill in the part missing from Chan [7]'s query algorithm. We then discuss a simple deletion-only k-NN structure. This allows us to adapt Chan's k-NN data structure to more general distance functions like the geodesic distance.

5.1 A dynamic data structure for planes

Chan [7] describes how to adjust his 1-NN data structure to efficiently perform k-NN queries. There are two main changes: the conflict lists are stored in k-NN data structures $\mathcal{D}_0$, 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 the $\mathcal{D}_0$ data structures. Each such structure uses linear space, can perform k'-NN 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 $\mathcal{T}^{(i)}$, each data structure $\mathcal{D}_0$ of a prism ∇ is built on $F_\nabla \cap F^{(i)}_{\text{live}}$. The total space usage is $O(n \log n)$.

The insertions remain unchanged, but deleting a plane h requires extra work. 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 $U(n) = O(\log^{6+\varepsilon} n)$ [7]. The improvement of Kaplan et al. [16] reduces this to $O(\log^{5+\varepsilon} n)$. It follows from Lemma 1 and the above modifications that:

▶ **Lemma 8** (Chan [7])**.** *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 neighbours 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 Lemma 3 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 [7] 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 reporting the t lowest planes above a query point $q \in \mathbb{R}^2$ in $O(n/r + \log r + t)$ time, and deletions in $O(r \log n)$ time. Our entire data structure consists of just $\ell = O(\log r)$ k_i-shallow cuttings $\Lambda_{k_0}, .., \Lambda_{k_\ell}$ of the planes, where $k_i = \lfloor 2^i (n/r) \rfloor$. Hence, this uses $O(n \log r)$ space. We can compute the shallow cuttings along with their conflict lists in $O(n \log n)$ time [10]. Note that when $r > n$, it can be that $k_i = 0$ for some i. In this case, we simply do

not build any of the cuttings that have $k_i = 0$. 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 small, we can make deletions efficient at the cost of making the query time fairly terrible.

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 in this cutting. Hence, the total time to go through all of these prisms is $\sum_{i=0}^{O(\log r)} r/2^i = O(r)$ time. When more than half of the planes from any conflict list are removed, we rebuild the entire data structure. Because every conflict list contains at least n/r planes, at least $\frac{n}{2r}$ 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 $k_i = 2^i(n/r) = O(t)$, 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, we can use a similar approach to Chan [6, 7] to achieve linear space usage, by storing only the prisms of the shallow cuttings, and storing the planes in an auxiliary data structure [3]. We then obtain the following result.

▶ **Lemma 9.** *For any fixed r, we can construct a data structure 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 $O(r \log n)$ time.*

General data structure. The general idea in the above data structure can be applied to any type of functions for which we have an algorithm to compute k-shallow cuttings. 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 it from the conflict lists in $\sum_{i=0}^{O(\log r)} S(n, k_i) = O((r/n)S(n, 1))$ time, and 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 $O(Q(n, n/r) + n/r + t)$. We thus have:

▶ **Lemma 10.** *For any fixed r, we can construct a 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 $O((r/n)(S(n, 1) + T(n, n/r) \log r))$ time.*

5.3 A general dynamic data structure

To generalize the dynamic k-NN data structure from Section 5.1 to other types of distance functions, we replace the $\mathcal{D}_0$ data structures by the data structure of Section 5.2. Queries and updates are performed as before (see Sections 2.2 and 5.1). This results in a dynamic k-NN data structure that can be used for any type of distance functions for which we can construct k-shallow cuttings. Next, we analyze the space usage and the running time in case of the Euclidean distance, as this is somewhat easier to follow, and then generalize to arbitrary distance functions.

Query time. Our $\mathcal{D}_0$ data structure 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 cutting at level j_k, the size of each conflict list we query is $O(k)$. By using our scheme to find the k-nearest neighbors over the substructures simultaneously, the query time becomes:

$$Q(n) = O([\log n + Q_0(O(k))] \log_b n + k) = O((\log n + (\log r + k/r)) \log_b n + k).$$

If we set $r = \log n$, and $b = \log^{\varepsilon} n$, we get $Q(n) = O(\log^2 n / \log\log n + k)$, matching the query time of Chan's approach.

Update time. Lemma 1 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. to the original data structure, since after building the hierarchy of shallow cuttings for a tower, we additionally need to build the structures $\mathcal{D}_0$ on each of the conflict lists. As before, building the shallow cuttings takes $O(n \log n)$ time [8]. Next, we analyse the time to build all data structures $\mathcal{D}_0$. Note that 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 $O(k_j)$. Let α be the constant bounding the size of the conflict lists. Using that $P_0(n') = O(n' \log n')$, we find the following running time:

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

The preprocessing time thus adheres to the recurrence relation $P(n) \leq P(n/b) + O(n \log^2 n)$, 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, with $h \in F^{(i)}_{\text{live}}$, we remove h 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 $\mathcal{D}_0$. By Lemma 1, 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. 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 $D(n) = O(b^2 \log^4 n \log_b n + b^2 \log^3 n \log_b^2 n) = O(\log^{5+\varepsilon} n / \log\log n)$.

Space usage. The space usage of a $\mathcal{D}_0$ data structure storing n' planes is $O(n' \log r)$. The space usage is thus $S(n) = \sum_{j=0}^{\log \frac{n}{k_0}} \frac{n}{k_j} \cdot O(k_j \log r) = O(n \log n \log\log n)$. Note that this can be reduced to $O(n \log n)$ by using the space reduction idea mentioned in Section 5.2.

We can use the same scheme for any distance measure that allows for constructing a k-shallow cutting. In the full version of this paper [12], we prove the following lemma (Lemma 11), which we then apply to constant description complexity distance functions and geodesic distance functions (refer to full version [12] for details) to obtain Theorem 12.

▶ **Lemma 11.** *Given an algorithm to construct a k-shallow cutting of size $S(n,k)$ on n functions in $T(n,k)$ time, such that locating the prism containing a query point takes $Q(n,k)$ time, we can construct a data structure of size $O(S(n,1)^2/n \cdot \log n \log\log n)$ that maintains a set of n functions and allows for k-lowest functions queries in $O(Q(n,1) \log n / \log\log n + k)$ time. Inserting a function takes $O((S(n,1)T(n,1)/n^2) \log^{2+\varepsilon} n)$ amortized time, and deleting a function takes $O((T(n,1)/\log\log n + T(n, n/\log n)) \cdot S(n,1)^2 \log^{4+\varepsilon} n/n^3)$ amortized time.*

▶ **Theorem 12.** *There is a fully dynamic data structure of size $S(n)$ that stores a set of n sites and allows for k-nearest neighbors queries in $Q(n)$ time, insertions in $I(n)$ expected amortized time, and deletions in $D(n)$ expected amortized time. $\mathcal{P}$ is a simple polygon with m vertices, and $\lambda_s(t)$ denotes the maximum length of a Davenport–Schinzel sequence of order s on t symbols and s a constant depending on the functions.*

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

References

1. Peyman Afshani and Timothy M. Chan. Optimal halfspace range reporting in three dimensions. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 180–186. SIAM, 2009. URL: http://dl.acm.org/citation.cfm?id=1496770.1496791.
2. Pankaj K. Agarwal, Lars Arge, and Frank Staals. Improved dynamic geodesic nearest neighbor searching in a simple polygon. In *34th International Symposium on Computational Geometry, SoCG*, volume 99 of *LIPIcs*, pages 4:1–4:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.SoCG.2018.4.
3. Pankaj K. Agarwal and Jirí Matousek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13(4):325–345, 1995. doi:10.1007/BF01293483.
4. Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008. doi:10.1145/1327452.1327494.
5. Timothy M. Chan. Random sampling, halfspace range reporting, and construction of $\leq k$-levels in three dimensions. *SIAM J. Comput.*, 30(2):561–575, 2000. doi:10.1137/S0097539798349188.
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. Three problems about dynamic convex hulls. *Int. J. Comput. Geom. Appl.*, 22(4):341–364, 2012. doi:10.1142/S0218195912600096.
8. Timothy M. Chan. Dynamic geometric data structures via shallow cuttings. In *35th International Symposium on Computational Geometry, SoCG*, volume 129 of *LIPIcs*, pages 24:1–24:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.SoCG.2019.24.
9. Timothy M. Chan. personal communication, 2021.
10. Timothy M. Chan and Konstantinos Tsakalidis. Optimal deterministic algorithms for 2-d and 3-d shallow cuttings. *Discret. Comput. Geom.*, 56(4):866–881, 2016. doi:10.1007/s00454-016-9784-4.
11. Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.
12. Sarita de Berg and Frank Staals. Dynamic data structures for k-nearest neighbor queries, 2021. arXiv:2109.11854.
13. Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993. doi:10.1006/inco.1993.1030.
14. Leonidas Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1):209–233, 1987.
15. 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.

16 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.

17 Rolf Klein and Andrzej Lingas. Hamiltonian abstract voronoi diagrams in linear time. In *Algorithms and Computation, 5th International Symposium, ISAAC, Proceedings*, volume 834 of *Lecture Notes in Computer Science*, pages 11–19. Springer, 1994. doi:10.1007/3-540-58325-4_161.

18 Der-Tsai Lee. On k-nearest neighbor voronoi diagrams in the plane. *IEEE Transactions on Computers*, C-31(6):478–487, 1982.

19 Chih-Hung 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*, pages 2842–2859. SIAM, 2020. doi:10.1137/1.9781611975994.173.

20 Chih-Hung Liu and 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*, pages 1633–1645. SIAM, 2013. doi:10.1137/1.9781611973105.117.

21 Jiří Matoušek. Reporting points in halfspaces. *Computational Geometry Theory and Applications*, 2(3):169–186, 1992.

22 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.

23 Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983. doi:10.1007/BFb0014927.