A Dynamic Data Structure for k-Nearest Neighbors Queries

Sarita de Berg[1] **and Frank Staals**[1]

1 **Department of Information and Computing Sciences, Utrecht University, Netherlands**
`s.deberg@uu.nl, f.staals@uu.nl`

Abstract

We present an insertion-only data structure that supports k-nearest neighbors queries for a set of n point sites in $O(Q(n) \log n + k)$ time, based on any static data structure that can perform k'-nearest neighbors queries in $O(Q(n) + k')$ time. The key component is a general query algorithm that allows us to find k-nearest neighbors spread over t substructures simultaneously, thus reducing the $O(tk)$ term in the query time to $O(k)$. Applying this to the logarithmic method yields an insertion-only data structure with both efficient insertion and query time. We apply our method in the plane for the Euclidean and geodesic distance. We then briefly discuss the main difficulties to achieve a similar running time in the fully dynamic case.

1 Introduction

In the k-nearest neighbors (k-NN) problem we are given a set of n point sites S in $\mathbb{R}^d$, and we wish to preprocess these points such that for 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 [3, 4, 8, 12, 13]. In particular, for sites in $\mathbb{R}^2$ and the Euclidean distance metric, Chan and Tsakalidis [8] achieved the optimal $O(\log n + k)$ query time using linear space and $O(n \log n)$ preprocessing time. Very recently, Liu showed how to achieve the same query time for general distance functions (in $\mathbb{R}^2$) using $O(n \log \log n)$ space [13].

In this paper, we study the dynamic version of the k-nearest neighbors problem, in which points can be inserted into or deleted from S, and the points lie in the plane. When we wish to report only one nearest neighbor (i.e. 1-NN searching), several efficient fully dynamic data structures exist [5, 7, 11]. Actually, all these data structures are variants of the same data structure by Chan [5]. For the Euclidean distance, the current best result using linear space achieves $O(\log^2 n)$ query time and polylogarithmic update time [7]. The variant by Kaplan et al. [11] achieves similar results for general distance functions. These data structures can also answer k-NN queries in $O(\log^2 n + k \log n)$ time [5]. Recently, Liu [13] claimed that the version of Kaplan et al. [11] can support such queries in $O(\log^2 n + k)$ time. However, we believe that there are some issues with this approach, as we briefly discuss in Section 4. For the Euclidean distance, Chan achieves a query time of $O(\log^2 n / \log \log n + k)$ for k-NN queries using $O(n \log n)$ space, by adapting his original data structure [6].

We are actually interested mostly in the insertion-only variant of the problem. Since nearest neighbor searching is decomposable, we can directly apply the logarithmic method [14] to turn a static k-NN searching data structure into an insertion-only data structure. However, this again yields an unwanted $O(k \log n)$ term in the query time. Our main goal is to reduce this term to $O(k)$ instead. In Section 2, we show how to achieve this goal. We present a general query algorithm that allows us to find the k-nearest neighbors spread over t substructures in $O(Q(n)t + k)$ time, assuming that the static data structure supports k'-NN

37th European Workshop on Computational Geometry, St. Petersburg, Russia, April 7–9, 2021.
This is an extended abstract of a presentation given at EuroCG'21. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear eventually in more final form at a conference with formal proceedings and/or in a journal.

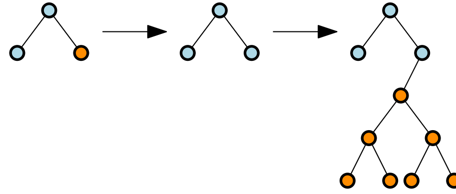

Figure 1 Example of expansion. Blue elements are included in a clan, orange elements are not. The expansion (building the next subheap) occurs when all elements have been included in a clan.

queries in $O(Q(n) + k')$ time. This yields a linear space data structure supporting queries in $O(\log^2 n + k)$ time and insertions in $O(\log^2 n)$ time, when using the Euclidean distance.[1]

Our original interest in the problem stems from a setting in which S is a set of points inside a simple polygon P with m vertices, and we use the geodesic distance as the distance measure. In this setting, Agarwal, Arge, and Staals [2] describe an insertion-only data structure for 1-NN queries that achieves $O(\log^2 n \log^2 m)$ query time, and $O(\log n \log^3 m)$ insertion time. As we show in Section 3, applying our machinery in this setting allows for efficient ($O(\log^2 n \log^2 m + k \log m)$ time) k-NN queries and insertions, as well.

2 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 [14]. 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 [14]. 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.

Query algorithm. Let q be the query point and k the number of nearest neighbors we wish to find. We use the heap selection algorithm of Frederickson [9] 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 (only) need to consider how clans are formed in the original heap, because we do not build the entire heap we query before starting the algorithm. Instead, the heap is expanded during the query when necessary, see Figure 1 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 a partition of S

[1] With a slight variation of this method we can match the $O(\log^2 n/ \log\log n + k)$ query time of Chan's [6] fully dynamic data structure for planes. However, this increases the insertion time to $O(\log^{2+\epsilon} / \log\log n)$.

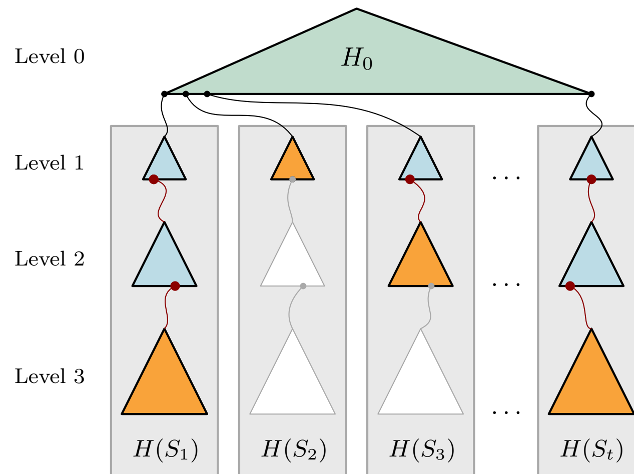

Figure 2 The heap that we construct for the k-nearest neighbors query. The subheaps of which all elements have been included in a clan are indicated in *blue*. The subheaps that have been built, but for which not all elements have been included in a clan, are indicated in *orange*. The *white* subheaps have not been built so far, because not all elements of their predecessor are in a clan yet.

into t disjoint sets. 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 Figure 2. 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 $H_i(S_j)$. Each heap $H_1(S_j)$ consists of the $k_1 = Q(n)$ sites in S_j closest to q, which we find by querying the static data structure of that group. We call these the *level 1* heaps. 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 included in a clan, i.e. e is considered by the heap selection algorithm. When e is included, we 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 1.

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 query time. As stated before, finding the k-smallest non-dummy elements of H takes $O(k + |H_0|)$ time [9]. In this section, we analyse the time used to construct H.

First, the level 0 and level 1 heaps are built. 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. Building H_0 takes only $O(t)$ 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 first prove the following two lemmas.

▶ **Lemma 1.** *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 been 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 $|H_i(S_j)| = k_1(2^{i-1} - 2^{i-2}) = k_1 2^{i-2}$. ◀

▶ **Lemma 2.** *The total size of all subheaps $H_i(S_j)$ at level $i > 1$ is $O(k)$.*

Proof. There are essentially two types of subheaps: *complete* subheaps, of which all elements have been included in a clan (shown blue in Figure 2), and *incomplete* subheaps, of which only part of the elements has been included (shown orange in Figure 2). 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. In total, $O(k)$ elements (not in H_0) are included in a clan, so the total size of all complete subheaps is $O(k)$. Because the size of a subheap is at most twice the size of its predecessor, it follows that the total size of all incomplete heaps at level greater than 1 is also $O(k)$. ◀

Building $H_i(S_j)$ takes $O(Q(n)+k_i)$ time. To pay for this, we charge $O(1)$ to each element of $H_{i-1}(S_j)$. Because we choose $k_1 = Q(n)$, Lemma 1 implies that $|H_{i-1}(S_j)| = \Omega(Q(n))$, and that $k_i = k_1 2^{i-1} = 2^2 k_1 2^{i-3} = O(|H_{i-1}(S_j)|)$. From Lemma 2, and the fact that all subheaps are disjoint, it follows that we charge $O(1)$ to only $O(k)$ sites. We then have:

▶ **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.*

Applying Lemma 3 to the logarithmic method, we 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 that can 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.*

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 $O(T(Q(n) \log n + k))$.

3 Applications

Points in $\mathbb{R}^2$. 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, 8].

▶ **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.*

By using the logarithmic method with only $O(\log_b n)$ groups, where $|S_i| = b^i$, we can improve the query time to $O(\log_b n \log n + k)$, at the cost of increasing the insertion time to $O(b \log_b n \log n)$. Setting $b = \log^\epsilon n$, we match the $O(\log^2 n/ \log\log n + k)$ query time of Chan [6] and still achieve an insertion time of $O(\log^{2+\epsilon} / \log\log n)$. For general distance functions we achieve the same query time using Liu's data structure [13], using $O(n \log\log n)$ space and expected $O(\text{polylog}\, n)$ insertion time.

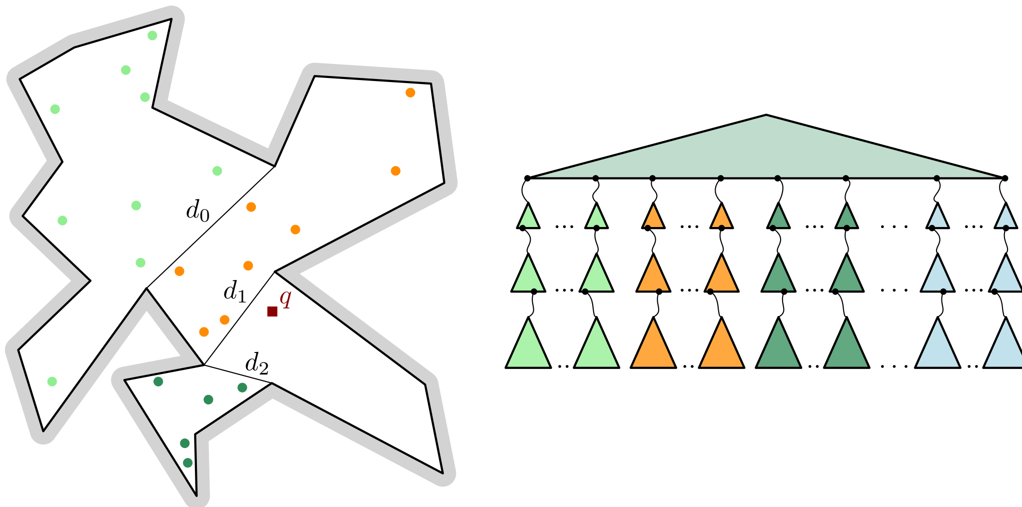

Figure 3 A partial decomposition of P and the corresponding heap used in a k-NN query for q.

Points in a simple polygon. In the geodesic k-nearest neighbors problem, S is a set of sites inside a simple polygon 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 between p and q fully contained within P. The input polygon P can be preprocessed in $O(m)$ time so that the geodesic distance $d(p, q)$ between any two points $p, q \in P$ can be computed in $O(\log m)$ time [10].

We recursively partition the polygon P into two subpolygons P_r and P_ℓ of roughly the same size [2]. We denote by S_r and S_ℓ the sites in P_r and P_ℓ, respectively. This results in a decomposition of the polygon of $O(\log m)$ levels. For both sets, and at each of the levels, we again use the logarithmic method to support insertions. At every level, we store S_ℓ in the static k-NN query data structure of Theorem 22 of [2]. It requires $O(n \log n)$ space, excluding the size of the polygon, and finds the k-nearest neighbors among S_ℓ for a point $q \in P_r$ in $O((\log n + k) \log m)$ expected time. Building the data structure takes $O(n(\log n \log m + \log^2 m))$ time. Insertions using the logarithmic method therefore take $O(\log^2 n \log^2 m + \log n \log^3 m)$ time, as there are $O(\log m)$ levels in the decomposition of P.

During a k-NN query, we have a partition of S into $O(\log n \log m)$ disjoint groups S_j, as we consider one set of sites (S_ℓ or S_r) for each level of the decomposition. An example is shown in Figure 3. Lemma 3 thus states that there is a data structure that allows for k-NN queries in $O(\log^2 n \log^3 m + k \log m)$ time. We can reduce this by setting $k_1 = \log n$ instead of $\log n \log m$ and charging $O(\log m)$ to each site of $H_{i-1}(S_j)$ to pay for building $H_i(S_j)$.

▶ **Theorem 6.** *Let 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 P, allows for geodesic k-NN queries in $O(\log^2 n \log^2 m + k \log m)$ expected time, and inserting a site in $O(\log^2 n \log^2 m + \log n \log^3 m)$ time.*

4 Supporting Deletions

The data structures for 1-NN queries also supporting deletions in $O(\text{polylog}\, n)$ time are all based on an idea of Chan [5]. Liu [13] recently claimed that this data structure (in particular the version of Kaplan et al. [11]) also supports k-NN queries in $O(\log^2 n + k)$ time. We believe there are some issues with this approach, which we sketch below.

The key ingredient for dynamic 1-NN searching is an algorithm that takes a subset S of n of the sites and produces an (abstract) data structure $\mathcal{T}$ storing S, and a partition of S into a set "good" sites G and a set of "bad" sites B. The key properties are that every site in S is stored at most $O(\log n)$ times in $\mathcal{T}$, and that G has size $\Omega(n)$. By recursively applying this algorithm on the bad sites, we obtain a partition of S into $r = O(\log n)$ good sets $G_1, .., G_r$. Each good set G_i is stored in a static 1-NN searching data structure $\mathcal{D}_i$, and thus we can answer queries by querying each of these $O(\log n)$ data structures. Deleting a site may cause some sites in these good sets to become marked as bad. These sites are reinserted into the structure of B, hence we essentially move some sites from a G_i to a new good set G_j. It can be shown that the total number of good sets remains $O(\log n)$. When a query in some static data structure $\mathcal{D}_i$ returns a site marked as bad we simply discard it. Chan shows that this still allows us to answer queries correctly, and that deletions (and insertions) take $O(\text{polylog}\, n)$ amortized time [5, 11]. Note that the data structures $\mathcal{T}_1, .., \mathcal{T}_r$ are used only to collect which functions become bad when performing deletions, not to answer queries.[2]

Liu claims that this data structure can also support k-NN queries in $O(\log^2 n+k)$ time [13]. Presumably, by replacing the 1-NN data structures $\mathcal{D}_1, .., \mathcal{D}_r$ by k-NN data structures (all details are omitted). However, the sites in $\mathcal{D}_1, .., D_r$ are not pairwise disjoint, and thus we may encounter a site in the output to a query in multiple $\mathcal{D}_i$'s. This yields an $O(k \log n)$ term in the query time, which matches the bound given by Chan [5].

To answer k-NN queries efficiently, Chan adapted his original data structure to accomodate k-NN queries. By using the data structures $\mathcal{T}_1, .., \mathcal{T}_r$ to answer queries, and deleting planes that are removed from these structures explicitly, a query time of $O(\log^2 n/ \log\log n + k)$ is achieved. However, it is not straightforward how to generalize this approach for more general distance functions (for example the geodesic distance function). We are currently working on this problem.

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.
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.
3. Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.
4. 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.
5. 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.
6. Timothy M. Chan. Three problems about dynamic convex hulls. *Int. J. Comput. Geom. Appl.*, 22(4):341–364, 2012.

[2] In the presentation of Kaplan et al. [11] $\mathcal{D}_i$ actually coincides with the lowest "layer" of in the "tower" of shallow cuttings $\mathcal{T}_i$. The issue sketched here also applies to this version, as the cuttings in $\mathcal{T}_i$ do not cover the t-level of G_i but of $G_i \cup B$ for some B. The sites/functions in B are "good" w.r.t. some other G_j, and may thus be encountered multiple times.

7 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.

8 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.

9 Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993.

10 Leonidas J. Guibas and John Hershberger. Optimal shortest path queries in a simple polygon. *J. Comput. Syst. Sci.*, 39(2):126–152, 1989.

11 Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha Sharir. Dynamic planar voronoi diagrams for general distance functions and their algorithmic applications. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2495–2504. SIAM, 2017.

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

13 Chih-Hung Liu. Nearly optimal planar k nearest neighbors queries under general distance functions. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2842–2859. SIAM, 2020.

14 Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983.